《软件测试课程设计》讲稿（第 5–8 章）

第 1–9 周 ｜ 各章课时见章首时间分配表

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 5 章　单元测试与集成测试

第 6 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 5 章 讲稿（第 6 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 5 章 单元测试与集成测试 —— 从"写第一行测试代码"到"把系统拼起来还能跑"
>
> 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（本章 = 书页 100 起）
>
> 配套课件：逐页还原版 **100 页**（`lecture5/`，云端 http://111.231.0.226:2345/lecture5/index.html）
>
> 教案对应：第 6 次课「单元测试与集成测试」
>
> 本讲重点：代码评审的价值与优秀实践（互查、检查表、速度与规模控制）、静态分析工具链（SonarLint／FindBugs／Checkstyle／SourceMonitor）、单元测试的目标与"驱动程序＋桩程序"、JUnit 5 的组成与关键组件（Annotation／Assert／Runner）、Mock 家族（Stub／Dummy／Spy／Fake／Mock）与 Mockito、覆盖率工具 JaCoCo 的原理、集成策略（自顶向下／自底向上／混合）、微服务下的契约测试（CDC）与 CI/CD 流水线中的持续测试

> 本讲难点：Mock 与 Stub 的取舍（"够用就行"）；契约测试为什么能把联调成本降到零；覆盖率数字与测试充分性的关系；CI/CD 流水线中各道质量门禁的先后与作用
>
> **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 能说出代码评审的四类关注点（规范与风格、圈复杂度 e−n+2p、信息流复杂度、耦合度）与互查的三条硬指标（每次 <200–400 行、300–500 LOC/h、单次 ≤60–90 分钟）；② 能列举静态分析工具及其定位（SonarLint／FindBugs／Checkstyle／SourceMonitor／Coverity）；③ 能说明单元测试的四条目标与"驱动一桩"的作用；④ 能说出 JUnit 5 = Platform + Jupiter + Vintage 三部分，并解释 @Test／@BeforeEach／Assert／Runner 各自的角色；⑤ 能区分 Dummy／Stub／Spy／Fake／Mock 五种测试替身，并说明 Mock 与 Stub 的核心差别（状态验证 vs 行为验证）；⑥ 能说出 JaCoCo 的六个覆盖率维度与 on-the-fly 插桩原理；⑦ 能说明自顶向下／自底向上／混合三种集成策略的优缺点，以及契约测试（CDC）的两端（Consumer／Provider）；⑧ 能画出 CI/CD 流水线的主要环节与质量门禁
能力	① 能对一个类写出符合 AAA（准备—执行—断言）结构的 JUnit 5 测试；② 能用 Mockito 的 mock／when／verify 写出一条"行为验证"用例；③ 能读 JaCoCo 报告，指出"覆盖率高但分支没测到"的具体位置；④ 能为一个微服务接口写一份最小契约（请求/响应格式），并说明消费者端与提供者端各测什么；⑤ 能在 CI 流水线里定位"测试失败该看哪一段日志"
素质（思政）	① 由代码评审"互为评审、集体评审"体会团队协作与开放心态；② 由"每 1000 行残留 2–6 个 Bug、找 bug 占总投入 30%–60%"体会质量成本意识与精益求精；③ 由第三方开源库漏洞数据（98% 的公司不了解自己在用什么）体会**供应链安全意识与合规责任**；④ 由"单元测试是系统质量的基石"体会脚踏实地、把基础做扎实的工匠精神；⑤ 由持续集成"门禁不通过就发不出去"体会规则意识与对交付负责

本讲要建立的一个总印象：第 4 章讲的是"流程怎么排"，这一章开始讲"**代码这一层到底怎么测**"——先用**评审和静态分析**在不运行代码的情况下抓缺陷，再用**单元测试**把每个函数钉死，最后靠**集成测试与持续集成**把零件拼成能跑的系统。

二、时间分配总表

> ⚠ **容量提示（务必先读）**：本章课件 **100 页**，本讲稿是**素材全量版**（口播约 5 万字，≈200 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）**：
> 5.6、5.9、5.16、5.34、5.36、5.37、5.39、5.46、5.48、5.49、5.53、5.60、5.64、5.69、5.77、5.81、5.82、5.90、5.92、5.93、5.95。
>
> 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（★）**：★5.10–5.15、★5.17–5.28、★5.30–5.31、★5.40、★5.42–5.45、★5.50–5.52、★5.55–5.59、★5.61–5.63、★5.65–5.68、★5.70–5.74、★5.78–5.80、★5.83–5.89、★5.90–5.93、★5.94、★5.96–5.97。
>
> 4. **必须当堂动手页**：5.37（那个 `add` 方法的缺陷，现场找）、5.49（断言写一条）、5.60（Stub/Mock 取舍判断）、5.77（画集成顺序）。
>
> 5. **实验 2（5.96）**务必用 3 分钟交代清楚：用 JUnit 测 `ItemController`，提交测试代码；**5.97 学习资源**可略。

90 分钟课堂推进表

时段	页码	内容	分钟	备注
第 1 节	5.1–5.4	开场、本篇地图、本章三部分	5	讲清"第 2 篇开始讲技术"
第 1 节	5.5–5.14	5.1 代码评审：价值、关注点、代码规范、互查实践	12	**5.6、5.9 必讲**（互查三条硬指标）
第 1 节	5.15–5.31	静态分析工具链与 SAST/SCA、供应链风险	10	**5.16、5.28 必讲**；工具页可略

—	—	**课间休息 10 分钟**	—	—
第 2 节	5.32–5.40	5.2 单元测试：价值、目标、驱动与桩	10	**5.34、5.36、5.37、5.39 必讲**
第 2 节	5.41–5.51	JUnit 5：架构、Annotation、Assert、Runner	9	**5.46、5.48、5.49 必讲**
第 2 节	5.52–5.66	Mock 家族与 Mockito	10	**5.53、5.60、5.64 必讲**
第 2 节	5.67–5.74	覆盖率：JaCoCo 与插桩原理	6	**5.69 必讲**；插桩源码页可略
第 2 节	5.75–5.88	5.3 集成测试：单体策略、微服务、契约测试	10	**5.77、5.81、5.82 必讲**
第 2 节	5.89–5.95	CI/CD 与持续测试、本章小结	6	**5.92、5.93 必讲**
第 2 节	5.96–5.98	实验 2 交代、学习资源、结束	3	实验要求说清

> **压缩优先级**：先压缩 ★ 标记的工具截图页与框架源码页 → 再压缩 JUnit 内部类（TestCase／TestSuite／TestResult）与 EasyMock／JMock 细节 → 最后才压缩
>
> 5.9、5.34、5.37、5.46、5.53、5.60、5.69、5.77、5.81、5.82、5.92。**这十一页是考点密集页，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，不要一次写完。
>
> 页码锚点与课件页一一对应（5.1–5.98），可逐页对照：http://111.231.0.226:2345/lecture5/index.html
>
> **口径提醒**：教材第 5 章标题为"单元测试与集成测试"，课件 5.3 页把教材的 5.7／5.8 两节（系统集成的模式与方法、持续集成及其测试）合并讲成"5.3 持续集成测试"——**按课件顺序讲，按教材口径记**。课件里出现

> 的 `SolarLint`／`SolarQube` 是 `SonarLint`／`SonarQube` 的笔误，以正确拼写为准。

【5.1】第 2 篇 软件测试技术（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 同学们，我们翻到新的一篇了——**第 2 篇，软件测试技术**。
>
> 上一篇讲的是概念、流程和规范，它回答的是"测试这件事该不该做、按什么规矩做"。可真坐到工位上、打开一个项目，你遇到的第一个问题不是流程，而是：**我手上这段代码，究竟怎么测？**
>
> 这一篇就来回答这个问题。它一共五章：**第 5 章 单元测试与集成测试**、第 6 章 系统功能测试、第 7 章 专项测试、第 8 章 国际化和本地化测试、第 9 章 软件测试自动化及其框架。
>
> 你把五章连起来看，就是一条从"最小零件"到"整条流水线"的路：先把一个函数测住，再把模块拼起来测，然后测整个系统的功能与性能安全，最后用自动化把这些活交给机器。
>
> 今天上的第 5 章，是这条路的第一段。第一段没打好，后面全白费。

> **板书：**竖写"第 2 篇 技术篇"→ 下方五章：单元与集成（5）→ 系统功能（6）→ 专项（7）→ 国际化本地化（8）→ 自动化框架（9）；左端标"零件"，右端标"流水线"。

> **提问：**这五章里，你觉得哪一章最"看得见成果"、哪一章最"看不见但最要命"？

> **预设回答：**

- 答"第 9 章自动化最看得见"——确实，自动化跑起来满屏绿灯很直观；但它测什么，取决于前几章设计的用例。
- 答"第 5 章最看不见"——对，单元测试写完谁也不夸你；可它是后面所有层次的底座。
- 答"专项测试最要命"——性能和安全出事最贵。但它依赖单元和集成阶段留下的可测性。
- 收口：**看得见的是结果，看不见的是地基**，这一章讲地基。

> **易错点 / 考点：**本篇五章的顺序不是随便排的，考试常问"哪一章讲集成测试""哪一章讲自动化框架"。记法：**从零件到整机到特种检验到出口到无人车间**。

> **过渡：**既然这一章讲地基，我们先把这一章的名字记住——单元测试与集成测试。

【5.2】软件测试方法和技术（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章的封面，我借它说三件事：这门课、这一章、以及这一章要替你解决的那个"卡点"。

课程是《软件测试课程设计》，我们用的是朱少民老师《软件测试方法和技术》第 4 版。**第 5 章 单元测试与集成测试**——请注意这个名字是两段："单元"是把一个函数、一个类单独拎出来钉死；"集成"是把这些零件装到一起，看它们还认不认对方。

为什么要单开一章讲这两段？因为现实中出问题最多的，恰恰是这两段的中间地带：单个函数都对，一拼起来就错；或者单元层次全是漏洞，一集成全线崩。

这一章学完，你应该能独立做一件事：**拿到一段陌生代码，写出第一批能跑起来的测试，并且说清"我为什么测这几组数据"。**

这句话就是你这一章的学习目标，请你抄在笔记第一行。

> **板书：**中间写"第 5 章 单元测试与集成测试"；上写"单元：先钉死零件"；下写"集成：再拼成整机"；右侧写本章出口一句话"给定代码 → 写出测试 → 说清理由"。

提问：如果只能选一件事作为本章的"毕业标准"，你会选"会把 JUnit 跑起来"，还是"能说清为什么这么测"？

> **预设回答：**

- 答"跑起来"——能跑是底线，工具一周就会，但会跑不等于会测。

- 答"说清理由"——这才是我想听到的。会跑的人多，能说清"为什么测这组数据、为什么漏了那组"的人少。

- 追问：那你觉得"说清理由"需要哪些知识？——用例设计、覆盖率、以及对缺陷类型的敏感度，本章都会给。

易错点 / 考点：章名里的"单元"不是"一个文件"，而是**可独立测试的最小单位**（通常是一个方法或一个类）；"集成"不是"部署上线"，而是**把单元按设计装配起来并测试装配结果**。这两个概念每年都有人答错。

> **过渡：**先把教材的这一章摊开看一遍，你就知道我们要走多少路。

【5.3】教材内容（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是教材第 5 章的目录，八节。我们先按顺序念一遍，然后我告诉你哪几节是一组。

第一节**代码静态测试**、第二节**代码评审案例分析**、第三节**代码静态检测工具**——这三节是一组，共同点是：**它们都不运行代码**。靠人看，靠工具扫，在程序跑起来之前就把毛病挑出来。

第四节**单元测试的目标和任务**、第五节**分层单元测试**、第六节**单元测试工具**——这三节是第二组，共同点是：**它们要运行代码**，把最小单位拉出来单独跑一遍，看它答不答得对。

第七节**系统集成的模式与方法**、第八节**持续集成及其测试**——这两节是第三组，共同点：**它们关心的是"拼"**，而且是"拼了还要天天拼、每次改动都拼一次"。

所以八节看着多，其实就是三块：**不跑代码先挑错、跑了代码钉零件、拼起来还得一直拼**。

你把这三块的分界线记住了，整章的骨架就立住了。教材是按八节写的，我们上课会按三块讲，一会儿那页会讲怎么对上。

再补一句读教材的方法：**教材的节标题，往往就是这一节的结论**。比如"分层单元测试"这五个字，已经告诉你单元测试不是一层，是分层做的；"持续集成**及其**测试"这个"及其"，暗示了持续集成落到测试上有自己的一套做法。你带着这个习惯去读，八节读起来就快得多。

还有一点要提醒：教材写的是"静态测试"和"静态检测工具"，不是"静态的测试"，这两个词别念混——**静态的是被测对象的状态（没运行），不是测试的手段**。

> **板书：**左侧竖写教材八节编号 5.1–5.8；右侧用三条横线切成三段：①静态（5.1–5.3）②单元（5.4–5.6）③集成（5.7–5.8）；每段末尾写一个字："看""跑""拼"。

提问：为什么教材把"代码评审"这种看起来"很土"的人工活动，放在"静态检测工具"前面讲？

预设回答：

- 答"因为它更早出现"——时间上是，但顺序的真正理由是**逻辑顺序**：先知道人会看什么、看哪些维度，才知道该让工具去自动化哪些维度。

- 答"因为工具是评审的自动化"——很好，这句话是本章的一条暗线。工具检查的东西，几乎都能追溯到评审关注点。

- 追问：那反过来，工具能替代评审吗？——不能，工具只认规则，评审还能问"这段代码为什么存在"。

易错点 / 考点：这八节的名字常被考成填空或匹配题。最易错的是把"5.5 分层单元测试"当成集成测试的内容

——它讲的仍然是**单元层次内部的分层**（比如先测最底层算法、再测上层的服务类），不是模块之间的集成。

过渡：教材八节，我们只有九十分钟。所以课件做了整合，怎么整的，下一页说清楚。

【5.4】本章结构：三部分五环节（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是我们这一章的**行程表**。我把教材的八节压成三部分，再补上小结和练习。

第一部分，**5.1 代码评审与分析**——对应教材前三节。学完你要能做到两件事：知道评审该盯哪些点（规范风格、圈复杂度、信息流复杂度、耦合度），以及会用一组工具把能自动查的部分查掉。

第二部分，**5.2 单元测试**——对应教材四到六节。这是本章的重心，也是你实验要动手的部分：目标是什么、驱动程序和桩程序怎么用、JUnit 5 怎么写出第一条用例、Mock 什么时候该上、覆盖率报告怎么看。

第三部分，**5.3 持续集成测试**——对应教材最后两节。它把"集成"从一次性动作变成一条流水线：代码一提交就编译、就跑测试、就过门禁。

后面还有两块收尾：**小结**帮你把重点难点归位，**练习**就是基于 JUnit 的单元测试实验——那是要交代码的，不是交感想。

为什么敢这么压？因为这三部分各自对应一个问句：**不跑代码能查出什么？跑起来怎么测一个单元？拼起来以后靠什么保证它一直是对的？** 三个问句答完，这一章就学完了。

再说时间怎么分。第一部分我给你们十二分钟左右，因为评审的关键点必须当堂讲透，尤其"互查那几条硬指标"——每次看多少行、用什么速度看、看多久，这些是要你以后真照着做的。第二部分最长，二十分钟上下，因为它是实验的直接依据：JUnit 怎么写、Mock 什么时候用，实验课上没人再给你从头讲。第三部分十分钟，重点是"策略选哪一种"和"契约测试解决什么问题"，工具配置可以自己看。最后留几分钟讲小结和实验要求。

也就是说：**今天听不懂的地方，你要当场问；今天听得懂但没动手的地方，实验课上必须补回来。**

板书：横排三格：①代码评审与分析（看）→ ②单元测试（跑）→ ③持续集成测试（拼）；下方两行："小结：重点难点归位"、"练习：JUnit 单元测试实验（交代码）"；右侧竖写三个问句。

提问：三部分里，哪一部分你以前在别的课里接触过、哪一部分完全没碰过？说说你印象里"单元测试"是干什么的。

> **预设回答：**

- 答"没碰过单元测试，以为是测试人员的事"——很典型。单元测试的第一责任人恰恰是**写这段代码的开发人员**，测试人员更多是定标准、看覆盖率、补关键用例。

- 答"写过 JUnit，就是 assert 一下"——入门了，但"assert 一下"背后有 AAA 结构、有边界数据选择、有替身技术，本章会把它们补齐。

- 答"持续集成听过，就是用 Jenkins"——工具只是外壳，内核是"每次变更都必须过同一套自动检查"。

- 收口：这三块里，第二块你要动手，第一块你要养成习惯，第三块你要理解它在团队里怎么落地。

> **易错点 / 考点：**本章的"三部分"是**课件的教学整合**，教材仍是八节。答题时如果题目问"教材第 5 章包括哪八节"，按八节答；问"本章分几部分讲"，按三部分答。两套口径不要混。

> **过渡：**好，从第一部分开始——代码评审与分析。

【5.5】5.1 代码评审与分析（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 我们进入第一部分，**代码评审与分析**。这一部分有一个共同的前提：**代码一行都不用跑**。
>
> 你别小看这个前提。程序不跑，意味着你还没有环境、没有数据、没有依赖——也就是说，**这是你能最早介入的时刻**。代码刚写完，甚至刚提交，你就能看。
>
> 这一部分有三个关键词：**评审**，是人看，同行坐下来读你的代码；**分析**，是工具看，静态扫描器按规则把可疑模式挑出来；**规范**，是照着看，先约定"什么算好代码"，评审才有共同语言。
>
> 记住一句话：**评审解决"人能不能看懂、会不会想错"，分析解决"规则能不能自动兜住"。** 两者不是替代关系，是前后脚关系。
>
> 接下来几页，我们先把"评审值不值"这个问题答掉——因为很多同学心里其实是怀疑的：花时间看别人代码，图什么？

> **板书：**写"5.1 代码评审与分析"，下画三个圈：人看（评审）／工具看（分析）／照着看（规范）；右下角写"前提：不运行代码 → 介入最早"。

提问：不运行代码就能找缺陷，你觉得这件事靠谱吗？凭什么？

预设回答：

- 答"不靠谱，不跑怎么知道错"——这是最常见的直觉。但缺陷分两类：一类是"跑出来才知道"（性能、并发），一类是"读一遍就知道"（空指针没判、资源没关、日志打了密码）。后一类占比很大。

- 答"靠谱，因为很多错误看就能看出"——对，而且越早看越便宜。

- 追问：那评审能替代测试吗？——不能。评审只能证明"我没看出问题"，不能证明"它是对的"。这句话后面还会用到。

易错点 / 考点：评审属于**静态测试**，是测试活动的一部分，不是"开发内部的事"。考试常出判断题："代码评审不属于软件测试"——错。

过渡：既然评审属于测试，那它到底能带来什么？我们看它列出的价值。

【5.6】代码评审的价值（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲"评审到底图什么"，一共列了六条价值，我一条一条讲，但请你注意它们的层次不一样。

第一条，**改善代码质量**。这是最直接的：别人读你的代码，最容易发现"你没想过的输入"和"你写顺手了的坏习惯"。

第二条，**跨团队共享知识**。这条容易被忽略。一个模块只有一个人懂，是项目最大的风险；评审逼着第二个人把这段代码读进去，风险就摊薄了。

第三条，**辅导缺少经验的程序员**。新手最有效的成长方式不是听课，是有人在他代码上写具体的评论："这里为什么要判空？如果传进来的是 null 会怎样？"

第四条，**坚持遵守代码规范**。规范写在文档里没人看，写在评审意见里就必须改。

第五条，**增强协作**。评审是一种低成本的、以代码为中心的技术交流，比开会有效。

第六条，**减少项目代价**。这是总账：缺陷在这一层被拿掉，就不必留到测试、上线、甚至用户投诉那一层去付账。

六条合起来一句话：**评审买到的不只是"少几个 Bug"，还买到了"团队对这段代码的共同理解"。** 后者的价值，往往比前者大。

我再讲一个反面情形，你就知道为什么值得。一个团队不评审，会发生什么？代码写完直接提测，测试人员拿到一个黑盒，只能按需求文档试；代码里那些"我看一眼就知道有问题"的东西——比如一个异常被 `catch` 之后什么都没做、比如一个循环里每次都去查数据库——测试用例根本碰不到，它们会一路带到线上。等到线上出了事再回头定位，你得先把这段没人读过的代码重新读懂一遍。

所以评审的本质是**把"读代码"这件事提前、并且让第二个人读**。它便宜，是因为读代码比调试代码便宜得多。

板书：中间写"代码评审"；六个分支：质量／知识共享／带新人／守规范／协作／降代价；在"降代价"旁标"← 总账"，在"知识共享"旁标"← 最易忽略"。

提问：六条里，哪一条是"老板最不在乎、但对团队长期最重要"的？

预设回答：

- 答"知识共享"——我同意。老板看的是这次交付，知识共享的收益在半年后才显形，可它决定了"这个人走了项目会不会瘫"。

- 答"辅导新人"——也很重要，但要注意：评审不是培训课的替代品，它是"在真实代码上做辅导"。

- 答"减少代价"——这是最容易被接受的理由，也是最容易量化的理由。

- 收口：**对外讲"降代价"，对内讲"知识共享"**——跟不同的人沟通，用不同的价值主张，这也是工程沟通的一部分。

易错点 / 考点：常考多选题"代码评审的价值包括哪些"，六条尽量记全。易错点是把"发现缺陷"当成唯一价值——只答这一条会丢分，因为评审在**知识传递和规范落地**上的作用同样被教材强调。

【思政落点 1】：代码评审最珍贵的地方，是它把"我的代码"变成"我们的代码"。写代码的人天然有防御心理——被人指出问题，第一反应是解释而不是修改。可工程上的事实是：**没有人能独自保证正确**。愿意让别人看、愿意承认自己没想到、愿意把评论当成帮助而不是挑刺，这种开放心态，是技术人最难练也最值钱的品格。团队协作不是一句口号，它就落在你收到评审意见时的那一次选择上：先改，再辩。

过渡：既然要评，评什么？总不能凭感觉说"我觉得这段不好"。下一页给出四个可操作的关注点。

【5.7】关注代码评审（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 评审要看什么？这一页给了四个关注点。我一个一个说，前两个是重点，后两个理解思想就行。
>
> 第一个，**是否符合代码规范、代码风格**。这一条最基础，也最容易自动化。命名是不是有意义、一个方法有没有写成长篇大论、缩进和括号是不是统一。风格问题不影响程序正确性，但它影响**可读性**，而可读性直接影响后面所有的维护成本。
>
> 第二个，**圈复杂度**，公式是 **e − n + 2p**。e 是控制流图的边数，n 是节点数，p 是连通分量数，一般一个方法就是 1。算出来的这个数，含义非常实在：**它等于这个方法的独立路径条数，也就是"最少要写几条测试用例才能把每条路走一遍"**。我给你一个手算的例子：一个方法画出来 9 个节点、10 条边，那么 10 − 9 + 2 = 3，说明它有三条独立路径，你最少的用例数就是三。行业里常用的经验阈值是**一个方法的圈复杂度不超过 10**，超了就该拆。
>
> 第三个，**信息流复杂度**，课件给的式子是**扇入乘扇出再平方**。扇入是"有多少个上层模块调用我"，扇出是"我调用了多少个下层模块"。为什么要平方？因为上下两头的复杂度是互相放大的：一个被二十处调用的方法，只要它自己还依赖五六个服务，改它一处，风险就横跨一大片。**接口越宽，改动越贵。**
>
> 第四个，**模块耦合性**——看功能上、数据上相关联的模块占多大比例。关联越多，越难单独测、单独改。
>
> 四个关注点其实在说同一件事：**评审不只看"这句写得对不对"，还要看"这个结构以后好不好活"。**

> **板书：**四条竖列：规范风格／圈复杂度 e−n+2p／信息流复杂度（扇入×扇出）²／耦合性；圈复杂度下面写手算例"e=10, n=9, p=1 → 3 条路径"；最右侧写一句话"结构决定维护成本"。

> **提问：**一个方法圈复杂度等于 1，说明什么？它是不是就"没问题"了？

> **预设回答：**

- 答"只有一条路径，不需要测试"——**这是最危险的误解**。圈复杂度只数**判定分支**，它不管数据。一个没有任何 if 的方法，照样可能因为除零、空指针、类型溢出出错。

- 答"结构简单，但仍要测数据和边界"——完全正确。圈复杂度低只说明"路径少"，不说明"数据安全"。

- 答"圈复杂度等于 1 就是没有 if"——基本对，但循环、`case`、`&&`／`||` 的短路也算分支，会把数字抬上去。

- 收口：所以圈复杂度是**用例数量的下限估计**，不是质量证明。

> **易错点 / 考点：**① 公式必须记准：**e − n + 2p**，考试常直接把 e、n、p 给你让算，最常错的是漏掉 `2p` 或把 p 当成 0；② p 是**连通分量数**，单个方法通常取 1，所以很多教材写成 e − n + 2；③ 判断题"圈复杂度越高程序越复杂、越难测试"——对；"圈复杂度低就一定没有缺陷"——错。

> **过渡：**四个关注点里，第一个最好落地，因为已经有现成的规范可以照抄。下一页就是两份。

【5.8】首推代码规范（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页只给两个链接，但它的意思比链接重要：**规范不要自己发明，先用业界成熟的。**
>
> 课件首推的是 Google 的代码规范，给了 C++ 和 Java 两份：C++ 那份在 google.github.io/styleguide/cppguide.html，Java 那份是同一站点下的 javaguide.html。为什么首推它？三个原因。
>
> 第一，它**足够细**。细到什么程度？命名怎么起、一个函数多长、注释写在哪、异常怎么用、甚至哪些 C++ 特性禁用，它都写了。细，才可执行；"代码要优雅"这种话是没法执行的。
>
> 第二，它**足够权威且公开**。公开意味着你的团队可以直接引用："我们按 Google Java 规范"，不用再花两个月吵命名该用驼峰还是下划线。
>
> 第三，它**天然适配工具**。规范一旦写死，就能变成 Checkstyle 的配置文件、变成 IDE 的格式化模板——这一点后面讲 Checkstyle 时会接上。
>
> 我的建议是：**团队规范 = 业界规范 + 少量项目约定**。业界规范解决 90% 的争议，剩下 10% 你们自己补一条"本项目额外约定"就够了。别一上来就写一份三十页的部门规范，写完没人看，等于没有。

> **板书：**写"规范三原则：够细 → 可执行；公开权威 → 不吵架；可配置 → 能进工具"；下行写"团队规范 = 业界规范（Google C++／Java）＋ 项目补充"。

> **提问：**如果你们团队已经在用另一套规范（比如阿里的 Java 规范），还要不要换成 Google 的？

> **预设回答：**

- 答"要换，Google 更权威"——没必要。规范的价值在于**统一**，不在于"哪家更正确"。已经统一了就别折腾。

- 答"不用换，能用就行"——正确方向，但要检查一件事：这套规范**有没有配套的自动检查工具**。没有工具支撑的规范，落地率极低。

- 追问：那如果一个项目里一半人按 A 规范、一半人按 B 规范呢？——这才是真正的问题，评审会变成风格吵架。先统一，再谈优劣。

> **易错点 / 考点：**考试可能问"代码规范的作用"，标准答法要落到**可读性、可维护性、团队一致性、可自动化检查**四点，不能只答"好看"。另一个易错点：把"代码规范"和"编码标准/安全编码标准"混为一谈——前者管风格与结构，后者管"哪些写法会引发漏洞"。

> **过渡：**规范有了，评审怎么组织才有效？这一页的四条硬指标，是本章最该背下来的操作细节。

【5.9】代码互查的优秀实践（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是本章最"能直接用"的一页，标题叫"代码互查的优秀实践"，说的是**同行互查怎么做才不流于形式**。我按课件给的几条讲，每条都给你翻成一句人话。
>
> 第一条，**一次检查少于 200 到 400 行代码**。为什么？因为人的注意力是有限的，超过这个量，后面看的都是"扫一眼就放过"。你要是拿着一千行 diff 说"帮我看看"，对方只会回你一句"看着挺好的"。
>
> 第二条，**努力达到一个合适的检查速度：每小时 300 到 500 行**。注意这个数字很反直觉——它意味着看四百行代码要花将近一个小时。**评审不是浏览，是逐行推演**：这个变量此时可能是什么值、这个分支进得去吗、这个异常谁接。
>
> 第三条，**有足够的时间、以适当的速度、仔细地检查，但不宜超过 60 到 90 分钟**。超过一个半小时，人的缺陷发现率会明显下降。所以正确做法是**分多次**，而不是"今天下午我全看完"。
>
> 第四条，**复审前，代码作者应该先给代码加注释**。这一条常被跳过。作者先在关键处写清"这里为什么这么处理"，评审者才不用 reverse engineering 你的思路。
>
> 第五条，**使用检查表（checklist）**。课件说得很明确：检查表肯定能改进双方的结果。因为清单把"凭记忆"变成"逐项确认"——空指针判了吗？资源关了吗？日志里有敏感信息吗？

第六条，**验证缺陷是否真正被修复**。这一条最容易被忽略：评审意见提了，作者说"改了"，然后没人回头看。**没验证的修复等于没修复。**

六条合起来其实是一个意思：**评审是一件有节奏、有清单、有闭环的工程活动**，不是"有空帮我瞄一眼"。

板书：六条压成一行口诀："**少（<200–400 行）· 慢（300–500 LOC/h）· 短（≤60–90 分钟）· 先注释 · 用清单 · 必回验**"。

提问：你们小组做课程设计时，如果只允许你落实其中一条，你选哪条？为什么？

预设回答：

- 答"用检查表"——很务实，成本最低、收益最稳。清单还能沉淀成团队资产。

- 答"少看一点"——也对，这条最容易执行：把大改动拆成几次提交、几次评审。

- 答"必回验"——如果只能选一条，我会选这条。因为**前五条决定"能不能发现问题"，第六条决定"问题有没有被解决"**。

- 收口：现实里最容易崩的是"速度"——大家习惯把评审当浏览，所以第 2、3 条最需要刻意练习。

易错点 / 考点：三个数字必须记准，考试很可能直接考数字：**200–400 行／次、300–500 LOC/h、60–90 分钟／次**。常见错法是把"每小时 300–500 行"记成"每天"或"每分钟"，或者把"单次不超过 60–90 分钟"记成"每天不超过"。判断口诀：**量小、速慢、时短**——这三个字方向一致，都是"少一点、慢一点"。

【思政落点 2】：这几条指标背后是一种很朴素的职业态度：**认真是有成本的，所以要给它留出成本**。"帮我瞄一眼"式评审之所以普遍，不是人懒，是没人给它排时间。请你们以后带项目时记住：给评审排一小时，比事后加班查一个线上 Bug 便宜得多。同时，检查表这件事本身就是一种可贵的习惯——**把靠记性的事变成靠清单的事**。飞行员起飞前要念检查单，不是因为不专业，恰恰因为专业的人也会漏。

过渡：方法有了，工具也得跟上。评审在今天的团队里，多数不是坐在会议室念代码，而是在版本平台上点开一个合并请求。

【5.10】工具支持：Collaborating with pull requests @GitHub（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲工具怎么承载评审。课件给的入口是 GitHub 的 pull request 协作文档，也就是我们常说的 **PR**。

PR 是什么？说白了就是"我改完了，请你们看，看完同意了我再合进主干"。它把评审需要的四样东西放在同一个地方：**改动的差异、逐行的评论、讨论的来龙去脉、以及"同意／要求修改"的结论**。

这四样东西为什么重要？我讲三个实际好处。

第一，**评审对象是"变更"，不是"整个仓库"**。你不是把三万行代码重读一遍，你只读这一次改了什么、为什么改。这就是为什么前面那条"一次 200 到 400 行"能落地——PR 天然把改动切小了。

第二，**留痕可追溯**。半年后有人问"这行为什么这么写"，你翻到那条评论，理由还在。文档会过期，PR 记录不会。

第三，**评审可以和自动检查绑在一起**。CI 一跑，静态检查、单元测试的结果直接挂在 PR 上，人只看"机器查不出来的部分"。这就是本章第三部分要讲的持续集成。

所以我的建议是：**哪怕课程设计只有三个人，也走 PR**。不是为了流程好看，是为了让你体验"我的代码要被人看"这种压力，这个压力会改变你写代码的方式。

> **板书：**中间写"PR = 差异 + 行级评论 + 讨论 + 结论"；左侧写"评审对象 = 变更"；右侧写"CI 结果挂在 PR 上"；下方写"三个好处：切小、留痕、自动化"。

提问：如果团队不用 GitHub，用 GitLab、Gitee 或者公司自建平台，这套做法还成立吗？

预设回答：

- 答"不成立，得用 GitHub"——名字不重要。GitLab 叫 Merge Request，本质一模一样；Gitee、Coding 也都有。

- 答"成立，换个名字"——对。要抓的是**四要素**（差异、评论、讨论、结论），平台只是载体。

- 追问：如果团队连版本管理都没有，只有共享文件夹呢？——那先解决版本管理，评审这件事在这个阶段做不了，因为**你根本说不清"改了哪些行"**。

易错点 / 考点：常考"代码评审的工具支撑方式"，答案要包含**版本控制平台的合并请求、行级评论、与 CI 集成**。易错点：把"PR"和"commit"混为一谈——commit 是一次提交，PR 是**一次待评审的变更集合**，可能包含多个 commit。

过渡：光说流程太抽象，我们看一个真实的变更评审长什么样。

【5.11】示例：change review（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是一次真实的变更评审示例。你不要把它当成"界面截图"看，要把它当成**一次评审的现场记录**，从中读出四件事。

第一件，**看边界**。这次改动动了哪些文件、每个文件改了几行、是新增还是修改。评审的第一分钟就该弄清"改动范围"，因为范围之外的东西你不需要看。

第二件，**看上下文**。好的评审界面不只显示改动的那一行，还显示它前后几行。原因很简单：**一行代码对不对，取决于它周围在做什么**。同一句赋值，放在事务里和放在事务外，风险完全不同。

第三件，**看评论的落点**。评论是挂在具体某一行上的，不是笼统地说"这段写得不好"。挂到行上，作者才知道该改哪；这也是我们训练表达的地方——**把模糊的不满，说成具体的建议**。

第四件，**看结论**。评审最后必须有一个明确的状态：同意合并、要求修改、还是先讨论。最怕的是"看完了，没说话"——那不是通过，那是漏审。

这张示例图要给你们的印象是：**评审是一个结构化的动作，有范围、有上下文、有落点、有结论**。缺一样，它就退化成聊天。

> **板书：**四格：范围（改了什么）／上下文（前后在干什么）／落点（评论挂在哪一行）／结论（同意・改・再议）。

提问：一次评审看完，作者只收到一句"整体还行，建议优化一下"，你作为作者会怎么办？

预设回答：

- 答"那就先不改了"——这正是模糊评论的后果：它不构成可执行的要求，于是被忽略。

- 答"回去问清楚哪里要优化"——对，这是作者该做的。同时也说明评审者没尽到责任。

- 追问：怎么把这句话变有用？——"第 42 行这个查询在循环里，改成批量查一次"，**具体到行、具体到改法**，这才叫评审意见。

易错点 / 考点：这页考的是"评审意见的质量标准"：**可定位、可执行、有理由**。易错点是把"评审"理解成"打分"或"表态"——评审的产出是**可执行的修改项**，不是一个态度。

过渡：评审最该盯的是什么类型的缺陷？下一页给了一个特别典型、也特别常见的：空指针保护。

【5.12】缺陷示例：空指针保护案例分析（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是一个缺陷案例，主题是**空指针保护**。它是评审里最值得练的一类缺陷，因为它有个特点：**不运行也能看出来，运行起来却常常不报错**——只是结果不对，或者干脆悄悄吞掉。
>
> 我们看这类缺陷的几种典型形态。
>
> 第一种，**该判的没判**。方法拿到一个对象就直接调它的方法，注释里声称"调用方保证非空"。可调用方有三个，其中两个不保证。这就是典型的"契约只写在注释里，没写在代码里"。
>
> 第二种，**判错了对象**。判了 A 不为空，用的却是 B；或者判了外层对象不为空，里面那个字段还是空的。这是评审时最容易看走眼的一种，因为"看起来有判空"。
>
> 第三种，**判了之后又拿来用**。比如判空之后走了一个分支，分支里又从别处取了个可能为空的值，直接使用。保护只保护了一半。
>
> 第四种，**用异常吞掉问题**。捕获了空指针异常，然后 `return null` 或者只打一行日志。这段代码从此"永不报错"，但它把一个确定性错误变成了一个随机错误——**错误被推迟到更远、更难查的地方**。
>
> 这四种形态说明一件事：**空指针不是一个"记得判一下"的问题，而是一个"要想清楚谁能保证非空"的问题**。所以评审到这里，问的不该是"这里判空了吗"，而是"**这个参数的为空可能性，由谁负责**"。

> **板书：**四种形态：漏判／判错对象／判一半／异常吞掉；右侧写一行"评审要问的问题：这个值由谁保证非空？"；底写"注释里的契约 ≠ 代码里的契约"。

> **提问：**一个方法在入口处对所有参数都做了判空，抛 `IllegalArgumentException`，这算不算做到了空指针保护？

> **预设回答：**

- 答"算，很规范"——形式上规范，但要追问：**调用方是谁？** 如果调用方全在同一层，这个异常会一路抛到用户面前；如果是给外部用的接口，快速失败是对的。

- 答"不算，得看业务上能不能为空"——这更接近本质。有些参数"可以为空"是合法输入，直接抛异常反而是错的，应该走默认分支。

- 追问：所以判空的正确姿势是什么？——**先定义契约**（可空／不可空、为空时怎么处理），再让代码和契约一致。

> **易错点 / 考点：**常考的判断题是"加了判空就没有空指针风险了"——错，判空只处理了你想到的路径。另一个易错点：`catch (NullPointerException e) {}` 这种写法在评审里属于**必须修改项**，因为它掩盖缺陷；考试若问"捕获空指针异常的坏处"，答"隐藏真实缺陷、把确定性错误变随机错误"。

> **过渡：**除了互查和工具，还有一些做法能让评审真正变成团队习惯。下一页列了五条。

【5.13】开发人员走上台介绍其代码（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲一种很朴素、但效果出奇好的做法：**让开发人员走上台，介绍自己写的代码**。课件给了三条规则：每个人都有机会、随机抽取、现场展示。

我先把这三条拆开说。

每个人都有机会——意思是这不是"高手表演"，是全员参与。新人也要讲，讲得不好也要讲，因为讲的过程就是逼他把自己的设计想清楚。

随机抽取——这一条是关键。如果提前指定，被点到的人会准备一份"演示版代码"；随机抽，你写的每一行都可能被摊开，这会实实在在改变平时的写法。

现场展示——不是念 PPT，是打开代码，从入口一路走下来：这段为什么这么拆、这个分支为什么存在、这里为什么加了缓存。

为什么有效？因为**写代码时你可以糊弄自己，讲代码时糊弄不过去**。讲到某一行你卡住了，说不出它为什么在这儿，那这行八成就该重构。这叫"**用讲解来验证理解**"。

它还顺带解决一个问题：团队里只有一个人懂某块代码，是最大的隐性风险。上台讲一次，第二个人、第三个人就有了基本概念，将来出问题不至于只有一个人能救。

你们课程设计只有三四周，我建议至少做一次：**每人讲五分钟自己的核心类**，不讲界面，讲逻辑。

> **板书：**写"上台讲代码"；下三条：人人有机会／随机抽取／现场展示；旁边写"讲到卡住的那一行 = 该重构的那一行"；再加一句"知识分散 = 风险分散"。

> **提问：**如果你被随机抽到，讲自己上周写的一段代码，你觉得最可能在哪一句上卡住？

> **预设回答：**

- 答"讲不清为什么用这个设计"——非常真实。很多设计是"当时顺手就这么写了"，讲不出理由，就说明这里没有真正的决策。

- 答"讲不清某个分支什么时候会进"——那这段代码要么是死代码，要么是没人测过的路径，两种情况都该处理。

- 答"能讲清，但怕被人挑毛病"——有压力是正常的。我提醒一句：**被挑毛病是这次活动最便宜的部分**，因为它发生在没有用户受损的时候。

> **易错点 / 考点：**这一页常被当成"软技能、不考"。其实考点很明确：**代码评审的形式包括正式会议评审、同行互查、走查（walkthrough）、上台讲解等**，判断题若说"代码评审只能是书面检查"就是错的。另外记住"随机抽取"的作用是**保持压力、避免表演**。

> **过渡：**互查、上台讲，是"怎么评"。还有几条是关于"评审怎么变成组织习惯"的，我们一起收一下。

【5.14】其它优秀实践（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是第一部分关于评审的最后一块拼图，五条实践，条条都指向同一件事：**让评审从"个人美德"变成"组织习惯"**。
>
> 第一条，**关键代码的集体评审**。不是所有代码都值得全员坐下来看，要挑：涉及资金的、涉及权限的、涉及核心算法的、并发和事务相关的。这类代码出错代价最大，评审的投入产出比最高。**评审要有优先级，不能平均用力。**
>
> 第二条，**代码评审作为一种文化**。文化这个词听着虚，其实有很具体的判断标准：新人提交代码，会不会有人主动看？指出问题时，语气是"这里可能有个坑"还是"你怎么连这个都不懂"？**前者是文化，后者是内耗。**
>
> 第三条，**借助工具更有效**。这一条前面已经讲过了：PR、静态检查、CI 报告。工具的价值是**把人从能自动查的事情里解放出来**，让评审的注意力集中在设计和逻辑上。
>
> 第四条，**流程定义**。要写清楚：谁必须评审、多长时间内响应、什么级别的改动需要几个人同意、评审不通过怎么处理。没定义的流程，全靠"看谁好意思催"。

第五条，**培训**。让评审者知道该找什么。这一条和前面的检查表是一对：**检查表告诉你看什么，培训告诉你为什么看。**

五条连起来看，它是从"挑重点（集体评审）→ 立氛围（文化）→ 上工具 → 定规矩（流程）→ 提能力（培训）"的一条链。缺哪一环，评审都会退化成形式。

板书：五行链条：挑重点（关键代码集体评审）→ 立氛围（文化）→ 上工具 → 定规矩（流程定义）→ 提能力（培训）；下方标一句"评审 = 制度，不是美德"。

提问：这五条里，如果你们团队只能补一条，你会补哪条？说说理由。

预设回答：

- 答"流程定义"——务实，先把"谁看、多久回、几个人同意"写下来，评审才有约束力。

- 答"文化"——最根本，但最难，因为它改的是人的态度。

- 答"关键代码集体评审"——投入产出比最高，尤其适合人少的团队。

- 收口：我的建议是先立"关键代码必集体评"这条最小规则，跑顺了再补流程和培训。**先有,再好。**

易错点 / 考点：这一页常考"代码评审的组织形式与实践"，要能说出**集体评审、评审文化、工具支撑、流程定义、培训**五项。易错点是把"集体评审"理解成"所有代码都要集体评审"——恰恰相反，它的前提是**挑关键代码**。

过渡：评审是人看。人看得慢、会累、也会漏。所以接下来我们要给评审配一套自动化工具——代码静态分析工具。

【5.15】代码静态分析工具（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

我们换一个角色——从"人看代码"换成"工具看代码"，这一块叫**代码静态分析工具**。

先把这个词拆开：**静态**，指不运行程序；**分析**，指按规则去理解代码结构；**工具**，指它能自动跑。三样合起来，它干的事就是：**把源码当成数据，用规则去查里面有没有可疑模式。**

它和评审什么关系？一句话：**评审负责"为什么这么写"，工具负责"这么写有没有踩规则"。** 人擅长判断意图，机器擅长不知疲倦地重复检查。这两件事不要互相替代。

这一块内容看着全是工具名，但考点其实不在"你会不会装插件"，而在**哪一类工具查哪一类问题**。我们一会儿会把工具谱系摊开，你重点记"归属"，别记按钮。

还有一句提醒：这些工具**会误报，也会漏报**。所以你看到它报了一堆，不要慌；看到它一个都没报，更不要放心。

板书：写"静态分析 = 不运行 + 按规则 + 自动化"；左右各一行："评审管意图／工具管规则"；下写"会误报、会漏报 → 结果是线索，不是判决"。

提问：一个工具跑完，报告里零缺陷，能不能据此说这段代码是安全的？

预设回答：

- 答"能，零缺陷就是没问题"——这正是最危险的想法。零缺陷只说明**没有命中它的规则**。

- 答"不能，规则有限"——对。规则的覆盖面永远是代码可能性的一小部分。

- 追问：那工具报告还有什么用？——它是**线索清单**，帮你把注意力放到高风险的地方，比人眼全量扫描高效得多。

易错点 / 考点：静态分析属于**静态测试手段**，"静态"指的是**不执行被测程序**，不是"代码不动"。判断题"静态分析可以发现运行时性能问题"——错，运行时问题必须动态测试。

过渡：那么这一块到底有哪些工具？我们按语言和用途把谱系排一遍。

【5.16】代码静态分析工具（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把静态分析工具的谱系排开了。我们不逐个讲用法，只讲"谁管哪一块"，这是考点，也是你以后选工具的依据。

先看通用型这一类：**SonarLint**。它不绑定某一种语言，Java、C、Python、JS 都能查，还能装进各种 IDE。它是这一节的"首推"，下一页单独讲。

然后是**按语言分的**。Java 这边三个老牌选手：**FindBugs** 查字节码层面的缺陷模式、**PMD** 查源码里的坏味道和可疑写法、**Checkstyle** 只管风格规范。请注意这三个的分工：**一个管"可能出错"，一个管"写得不好"，一个管"不符合规范"**——很多同学把它们混成一个东西，其实它们查的不是一回事。

C/C++ 这边，课件列的是 **Clint、Oclint、Cpplint** 和 **TestC++** 这一类，多是从 Google 内部或开源社区长出来的风格与缺陷检查器。

Python 这边是 **Pylint** 和 **PySonar2**——Pylint 管规范和可疑写法，PySonar2 更偏类型推断和语义分析。

最后是**安全类**的大块头：**Coverity、Fortify**。它们面向的是"能被攻击的缺陷"，做的是深度数据流分析，代价是慢、贵，一般放在 nightly 或发布前跑。

课件还给了一个总入口：OWASP 的源码分析工具清单。它的价值是**帮你横向对照**——同一类问题，有哪几个工具能查、各自支持什么语言。

这一页请记住一条选型逻辑：**开发机上装轻的（SonarLint、Checkstyle），流水线上装重的（Coverity、Fortify）。**

板书：三层：通用（SonarLint）／分语言（Java：FindBugs・PMD・Checkstyle；C/C++：Clint・Oclint・Cpplint・TestC++；Python：Pylint・PySonar2）／安全（Coverity・Fortify）；旁注"轻的上开发机，重的上流水线"。

提问：同样是 Java，FindBugs 和 Checkstyle 能不能互相替代？为什么？

预设回答：

- 答"能，都是查 Java 的"——不能。它们查的对象不同：Checkstyle 只看**源码的书写形式**，FindBugs 看**编译后的字节码里的可疑模式**。

- 答"不能，一个管风格一个管缺陷"——对。风格问题不影响运行，缺陷模式可能导致运行出错，严重程度不同，处理优先级也不同。

- 追问：那 PMD 放哪？——PMD 更偏源码级的"坏味道"：没用的变量、空的 catch、复杂的条件。三者是**互补**关系。

易错点 / 考点：① 三个 Java 工具的定位必须分清：**Checkstyle＝风格规范，PMD＝源码坏味道，FindBugs＝字节码缺陷模式**；② 安全类工具（Coverity／Fortify）属于**深度静态分析**，常考"它和轻量检查的区别"；③ 易错：以为 SonarLint 只支持 Java——它几乎是全语言。

过渡：谱系里排第一的那个，我们单独看一眼它凭什么被首推。

【5.17】首推：SonarLint（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲为什么在这么多工具里，课件首推 **SonarLint**。它给了三件事：**发现缺陷、修正缺陷、学习**。我按这个顺序讲，但重点在第三件。

发现缺陷——这是基本功能。你在 IDE 里写代码，它实时在可疑的地方标出来。注意"实时"这两个字很关键：它把检查从"提交后"提前到"书写中"，你刚写完就知道，改起来几乎零成本。

修正缺陷——它不只告诉你"这里有错"，还会给出**具体怎么改**，很多时候能一键修。这一点对新手特别友好：它教的不是"你错了"，而是"应该这样写"。

学习——这是我认为最值钱的一条。每一条规则背后都有解释：为什么这么写有风险、什么场景下例外。你用三个月 SonarLint，等于跟着一本会挑你毛病的教材在学。**工具的最高价值不是替你干活，是让你少犯同样的错。**

课件还强调它**几乎支持所有流行的语言和 IDE**。这一条的现实意义是：你不用为了换语言换工具，团队里用 Eclipse、用 IntelliJ、用 VS Code 的人，规则口径能保持一致。

最后提醒一句：SonarLint 装上之后，第一周你会觉得满屏都是问题，很想把它关掉。**别关。** 先把严重级别调到你关心的那一档，逐个消，你会发现自己写代码的习惯在两三周内明显变化。

> **板书：**三个圈写"发现 → 修正 → 学习"，在"学习"上打星；下写"实时 = 把检查提前到书写中"；再写"几乎全语言、全 IDE → 团队口径一致"。

> **提问：**一个工具既能发现问题又能教你怎么改，那它是不是可以替代老师、替代代码评审？

> **预设回答：**

- 答"差不多可以"——差得远。工具只认规则，它不会问"这个需求该不该这么实现"。

- 答"不能，规则有边界"——对。它能查"这里可能空指针"，查不了"这个业务逻辑本身错了"。

- 追问：那它能替代哪部分工作？——**能替代"机械的、有明确规则的检查"**，正好把评审者的时间省下来看设计。

> **易错点 / 考点：**SonarLint 的三项能力（发现／修正／学习）常考。易错点有两个：① 把它和 **SonarQube** 混为一谈——SonarLint 是 **IDE 里的本地检查器**，SonarQube 是**服务端的质量平台**，下一节会讲两者怎么配合；

② 误以为它只在保存或提交时检查——它是**边写边查**的。

过渡：说它"能装进几乎所有 IDE"，我们就在最常见的 IntelliJ 里看一眼它长什么样。

【5.18】示例：在 IntelliJ 应用 SonarLint 插件（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是在 IntelliJ 里装 SonarLint 的示例。安装动作本身很简单——在插件市场搜 SonarLint、装上、重启，课件给的插件页地址是 JetBrains 插件市场的 7973 号条目。我要讲的不是点哪里，而是**装完以后你应该怎么用它**。

第一，**看提示的层级**。它标出来的问题分严重级别，从一个文件的窗口里能按级别筛。你可以先只看最严重的那一档，把"会出事的"清掉，再处理"不好看的"。

第二，**看规则的说明**。每条提示点开都有解释：这条规则在防什么、什么情况下可以忽略。**忽略也是一门学问**——你要能说清"我为什么认为这里不适用"，而不是"我觉得它烦"。

第三，**注意它查不到的地方**。IDE 插件做的是单文件、轻量、快速的分析，它不做跨模块的深度数据流追踪。所以它报"没问题"，只代表这个文件在本文件范围内看起来没问题。

第四，**别把提示当成必须全清的清单**。全清意味着你要为一个老项目改几千处风格，那不现实。正确做法是：**新代码零新增问题，老代码逐步收敛。** 这条原则在团队落地时特别重要。

板书：写"装插件 → 分级看 → 读规则说明 → 知道它查不到什么"；右下写团队落地原则："新代码零新增，老代码逐步收敛"。

提问：一个已有十万行的老项目，SonarLint 一扫报了两千个问题，你要不要一次性全改？

预设回答：

- 答"要，一次清干净"——风险很大：两千处改动，回归测试根本覆盖不住，很可能改出新的 Bug。
- 答"不改，反正能跑"——那新代码会继续往这个坑里掉。
- 答"分两步：先卡住新增，再慢慢清老的"——这是正确做法，也是业界通行的"棘轮"策略。
- 追问：怎么"卡住新增"？——**只对本次变更的代码做检查**，这就是后面要讲的 CI 门禁。

> **易错点 / 考点：**这一页考的是"工具落地的策略"，而不是安装步骤。标准答法：**分级处理、差异检查、新代码零新增**。易错点是把"清空所有告警"当成目标——那会导致大量无效改动与技术债的假性清偿。

> **过渡：**本地查完，团队怎么保证"每个人的标准一样"？靠服务端。SonarLint 可以和 SonarQube 连起来用。

【5.19】可以和 SonarQube 集成使用（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页讲 SonarLint 和 **SonarQube** 的配合。这两个名字太像了，我先把它们分清楚：**SonarLint 在开发者的 IDE 里，是个本地检查器；SonarQube 在服务器上，是个团队级的质量管理平台。**
>
> 它们为什么要连起来？因为不连会出两个问题。
>
> 第一个问题，**规则不一致**。你本地用默认规则集，觉得挺干净；服务端用团队配置的规则集，一扫一大片。开发者和流水线互相不认账，最后谁也不信谁。连接之后，本地会**同步服务端的质量配置**，本地和服务端用同一套标准。
>
> 第二个问题，**没有全局视角**。本地只能看你打开的这几个文件；服务端能看整个项目、整个版本、甚至多个版本的趋势——**技术债是在涨还是在降，重复代码有多少，覆盖率是多少**，这些都得靠平台。
>
> 所以这套组合的分工是：**SonarLint 负责"我正在写的这段代码"，SonarQube 负责"我们整个项目现在什么水平"。** 一个管即时反馈，一个管长期趋势。
>
> 落到团队实践上，通常是这样：本地装了 SonarLint 并且连上服务端，提交前该清的就清了；CI 上再跑一遍 SonarQube，用**质量门禁**卡住合并。两道关，一道防"写进去"，一道防"合进去"。

> **板书：**左写"SonarLint：IDE 内、单文件、即时"；右写"SonarQube：服务端、全项目、趋势＋门禁"；中间写"连接 → 规则同步"；下写"两道关：写了就查、合了就卡"。

> **提问：**如果团队只允许保留一个，你留 SonarLint 还是 SonarQube?

> **预设回答：**

- 答"留 SonarQube"——它覆盖面更广、能做门禁；但少了本地即时反馈，开发者要等到提交才知道问题。
- 答"留 SonarLint"——反馈快，但没有统一标准和趋势，团队层面管不住。

- 答"看团队规模：小团队留 SonarLint 够用，多人协作要 SonarQube"——这个判断很实用。**工具的选择跟协作规模走。**

- 追问：那如果两个都留，冲突怎么处理？——让服务端当唯一权威，本地跟随服务端配置。

> **易错点 / 考点：**SonarLint 与 SonarQube 的区别是高频考点：**位置（IDE／服务端）、范围（单文件／全项目）、时效（即时／周期性）、职责（个人反馈／团队门禁）**。另注意课件里 `SolarLint` ／ `SolarQube` 是笔误，正确拼写是 **Sonar**Lint／**Sonar**Qube，答题时写对拼写。

> **过渡：**接下来看一个"曾经很红、现在基本退场"的工具，它的兴衰本身就是一课。

【5.20】来自大学的 FindBugs（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页讲 **FindBugs**。它的出身很有意思：**不是商业公司做的，是高校研究团队做出来的**，项目主页在 findbugs.sourceforge.net。
>
> 它的技术路线和前面那些不一样：**它不看源码，看编译后的字节码**。为什么这么做有道理？因为源码里有很多写法是"看起来像缺陷"的，而字节码反映的是编译器真正生成的东西——它直接对应"程序运行时会怎么做"。所以 FindBugs 抓的是**缺陷模式**：空引用可能发生、锁的释放路径不完整、equals 和 hashCode 不配对、一个值被算了但没用上。这些东西有个共同特点：**它们不违法语法，编译器不报错，但运行时可能出事。**
>
> 它的价值在哪？在于**它把"老程序员的经验"变成了可自动执行的规则**。一条规则背后，往往是一个真实事故。
>
> 但这一页真正要讲的是后面那半句：**2015 年 3 月，FindBugs 3.0.1 发布之后，就没有更新了。** 一个曾经被广泛使用的工具，停更了。
>
> 这件事给我们两点提醒。第一，**工具是有寿命的**。它依赖的 JDK 在升级、字节码格式在变，停更意味着慢慢失效。第二，**但缺陷模式不会过期**。空引用、锁未释放、equals 不配对，这些坑十年后还在。所以学工具要学它**在查什么**，别只学它怎么点。

> **板书：**写"FindBugs：高校出身 → 查字节码 → 缺陷模式"；右侧写时间线"2015.03 发布 3.0.1 → 之后停更"；下方写一句"工具会停更，缺陷模式不会过期"。

> **提问：**一个工具已经停止维护了，你还敢在项目里用它吗？

预设回答：

- 答"不敢，风险太大"——合理。停更的工具有两个现实问题：新语法可能解析不了，新 JDK 可能跑不起来。

- 答"能用，规则还是那些"——部分正确，但要限定条件：**只作为辅助参考，不作为门禁依据**。

- 追问：如果它报了一个你觉得很有价值的缺陷，怎么办？——把这条规则的理解搬到仍在维护的工具里去实现，**人要带走知识，不要抱着工具**。

易错点 / 考点：FindBugs 的两个特征常考：**基于字节码分析、2015 年 3 月 3.0.1 之后停止更新**。易错点：把它归为"代码风格检查工具"——它不是，它查缺陷模式；风格检查是 Checkstyle 的活。

过渡：它停更之后，位置被谁占了？答案有点出人意料——被 IDE 自己。

【5.21】FindBugs @ IntelliJ（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页接着讲 FindBugs 在 IntelliJ 里的命运。课件给了一句很直白的话：**IntelliJ IDEA 2021.2 发布之后，就基本把 FindBugs 干掉了。**

为什么会被"干掉"？因为 IDE 厂商把这类检查**内建**了。你想想开发者的选择逻辑：装一个插件要配版本、要跟着 JDK 升级、还会和 IDE 的检查重复报错；而 IDE 自带的检查开箱即用、和编辑器深度集成、跟着 IDE 一起升级。**同等效果下，内建永远赢过插件。**

这件事在教学上很有价值，我给你三层意思。

第一层，**工具会往平台里收**。单点工具活得好不好，很大程度取决于平台厂商是否愿意自己做。这条规律在测试领域反复出现：IDE 收了静态检查，CI 平台收了测试执行，代码托管平台收了评审。

第二层，**能力没有消失，只是换了入口**。字节码级的缺陷检查思路并没有死，它变成了 IDE 的 inspection，也变成了 SonarLint 这类工具里的规则。

第三层，也是最实用的：**你学的应该是"缺陷长什么样"，而不是"某个插件在第几个菜单"**。菜单会变，缺陷模式十年不变。这也是为什么这一章我反复强调"工具会换，模式不会换"。

板书：写"FindBugs（插件）→ IntelliJ IDEA 2021.2 之后基本被内建 inspection 取代"；下三层：工具往平台收／能力换入口／学模式不学菜单。

> **提问：**既然 IDE 自带检查就够了，我们为什么还要单独学 Checkstyle、SonarLint 这些工具？

> **预设回答：**

- 答"因为它们查得更全"——部分对。更重要的是**它们能在 IDE 之外跑**：CI 上、构建脚本里，这些地方 IDE 不在场。

- 答"因为团队要统一标准"——很关键。IDE 自带的检查，每个人的配置可以不一样；CI 上的检查，全团队一个标准。

- 追问：所以工具的真正战场在哪？——**在"离开开发者电脑之后"**。这一句直接指向后面的持续集成。

> **易错点 / 考点：**这一页的考点是"工具演进的原因"，标准答法是 **IDE 内建能力替代第三方插件、平台收敛单点工具**。易错点：以为"被替代 = 被淘汰 = 没用"，其实被替代的是**形态**，不是**能力**。

> **过渡：**IDE 自带的检查到底能查出什么？下一页用一张图说清楚它的能力边界。

【5.22】IntelliJ 分析功能代替了 FindBugs（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是用 IDE 自带的分析功能来替代 FindBugs 的示意。我讲这一页的目的不是让你记住某个菜单，而是让你建立"**IDE 检查的四个维度**"这个框架。
>
> 第一个维度，**语法与语义层**：变量声明了没用、方法返回值被忽略、不可能成立的条件、`switch` 少了分支。这类问题 IDE 靠编译器前端就能发现，几乎零误报。
>
> 第二个维度，**空值与可空性层**：可能为 null 的引用被直接使用、`@Nullable` 标注与实现不一致。这是日常最高频的一类。
>
> 第三个维度，**资源与并发层**：流没关、锁在异常路径上没释放、在同步块里调外部服务。这类问题人眼很难稳定发现，工具却擅长。
>
> 第四个维度，**风格与规范层**：命名、格式、注释缺失。注意这一层**严格来说不算缺陷**，但它影响可读性，所以也放进来了。
>
> 四个维度里，前三个是"会出事"，第四个是"不好看"。你在 IDE 里看到的提示，绝大多数都能归到这四个里。**归好类，你就知道哪些必须当场改、哪些可以排队改。**

> 另外提醒一句：IDE 的分析是**本地的、增量的、快的**，它不做跨模块的全程序分析。所以它的"没报"，只代表这一段代码在它能看到的范围内没问题——这一点和前面 SonarLint 的限制是一样的。

> **板书：**四行写四层：语法语义／空值与可空性／资源与并发／风格规范；右侧标"前三条＝会出事，第四条＝不好看"；底写"本地・增量・快 → 不做全程序分析"。

> **提问：**这四层里，哪一类问题最容易被"我本地跑得好好的"这句话掩盖过去？

> **预设回答：**

- 答"资源与并发"——很准。资源泄漏和竞态在单次运行、数据量小的时候几乎不显形。

- 答"空值"——也对，尤其是"这次传进来的恰好不是 null"的情况。

- 追问：所以这类问题的检查应该放在哪？——**放在流水线上做，不依赖运气**。这就是持续集成的价值。

> **易错点 / 考点：**常考"静态分析能发现哪类缺陷、不能发现哪类"。能：空引用、资源未释放、未使用变量、可疑逻辑。不能：性能瓶颈、并发时序、业务逻辑错误、界面问题——这些必须动态测。答这类题要**同时给出"能"和"不能"**，只答一边会被扣分。

> **过渡：**IDE 的检查是"提高下限"，但如果要把标准变成**团队的、可执行的门禁**，得靠专门的工具。下一页是其中最典型的一个：Checkstyle。

【5.23】代码风格检查 CheckStyle（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲 **Checkstyle**，主页是 checkstyle.org。它的定位特别清楚：**它只管代码风格与规范，不管程序会不会出错。**
>
> 别因为"只管风格"就小看它。它最大的价值是**把规范从文档变成可执行的门禁**。课件给的集成方式很具体：在 Maven 里跑 `mvn checkstyle:checkstyle`；在 Gradle 里 `apply from: 'checkstyle.gradle'`；规则写在 `config/checkstyle/checkstyle.xml`；跑 `./gradlew check` 就会执行检查；结果输出到 `build/reports/checkstyle/main.html`。
>
> 你把这串东西连起来看，它其实是一条很典型的工程链路：**规则写成配置文件 → 构建工具调用 → 输出机器可读的报告 → 报告进 CI 门禁**。这条链路不是 Checkstyle 独有的，几乎所有静态检查工具都是这个结构，你

以后学别的工具只要换个文件名。

为什么强调"进 CI"？因为只在本地跑，结果就是"我今天没空跑"。进了流水线，它才是**所有人必须通过同一道关**。这也是本章第三部分要反复讲的一句话：**规范一旦可执行、可强制，它就真了。**

我再补一个实操建议：**第一版规则别开太多**。先开命名、行长、Javadoc、未使用导入这几类低争议的，跑顺了再逐步加严。一上来开一百条规则，团队第一天就会集体绕过它——**能被执行的严格，比纸面上的完美重要**。

板书：横链写"checkstyle.xml（规则）→ mvn checkstyle:checkstyle ／ apply from: 'checkstyle.gradle'（调用）→ ./gradlew check（执行）→ build/reports/checkstyle/main.html（报告）→ CI 门禁"；右侧写"只管风格，不管对错"。

提问：一个团队用了 Checkstyle 之后，代码风格统一了，是不是代码质量就提升了？

预设回答：

- 答"提升了"——要打个折。风格统一提升的是**可读性和可维护性**，跟"有没有 Bug"没有直接关系。

- 答"没提升，只是整齐"——也不完全对。整齐带来的好处是真实的：评审时不用再讨论格式，注意力能放到逻辑上。

- 追问：那"质量"该怎么提？——靠评审抓意图、靠静态分析抓缺陷模式、靠测试抓行为。**风格检查是地基，不是全部。**

易错点 / 考点：Checkstyle 的定位是高频考点：**只查风格规范，不查缺陷**。常见的混淆是把 Checkstyle 和 PMD 搞反——PMD 会报"空的 catch 块""没用的变量"这类**坏味道**，Checkstyle 报的是"行长超过 120""类名首字母没大写"。另外，`./gradlew check` 与 `checkstyle:checkstyle` 的区别要能说清：一个是任务聚合、一个是直接生成报告。

过渡：规则那么多，具体都能查些什么？看一个规则示例，你就知道它的粒度有多细。

【5.24】CheckStyle 规则（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是 Checkstyle 的规则示意。规则列表很长，但你不用背，你需要建立的是**规则的分类感**——知道它能

在哪些维度上约束你。

我把它归成五类。

第一类，**命名**：类名、方法名、变量名、常量名的书写形式。这一类的争议最小，收益也最直接——名字统一了，读代码的人不用每次重新适应。

第二类，**格式与排版**：缩进宽度、行长度上限、左花括号的位置、空行的使用、运算符两侧的空格。这一类最"琐碎"，但也最适合机器管，因为它完全客观。

第三类，**注释与文档**：公共方法和公共类要不要写 Javadoc、注释的格式对不对。这一类的价值在于**逼你写下"这个方法干什么、参数什么含义"**。

第四类，**结构与语句**：`if` 必须带花括号、`switch` 必须有 `default`、不允许 `import *`、不允许空语句块。这一类的部分规则已经接近"潜在缺陷"了——比如漏了 `default`，可能就是你没想到的情况。

第五类，**度量**：方法长度、参数个数、圈复杂度、嵌套深度。这一类是**把前面讲的评审关注点变成自动检查**——你还记得圈复杂度 e−n+2p 吧？Checkstyle 可以直接把超过阈值的类和方法列出来。

五类看下来你会发现：**Checkstyle 的规则其实就是"团队约定"的可执行版本**。你写一条规范，它就能查一条。

> **板书：**五类竖排：命名／格式排版／注释文档／结构语句／度量（长度・参数数・圈复杂度・嵌套深度）；旁边一行"规范的可执行版本"。

提问：这五类里，哪一类最可能"查出来但你不服"？为什么？

预设回答：

- 答"格式排版"——常见。比如"行长度不超过 100"，你写一个长正则就会被卡。

- 答"参数个数"——真实痛点。有时候参数多是业务决定的，硬拆反而更糟。

- 追问：遇到这种情况正确做法是什么？——要么**在配置里给这一条加例外并写明理由**，要么**改设计**。不能做的是"悄悄在注释里加个忽略标记，谁也不知道为什么"。

易错点 / 考点：这一页的考点在"规则的层次感"：**第五类度量规则和代码质量最相关**（方法太长、太复杂、嵌套太深，本身就是缺陷温床）。易错点：把 Checkstyle 的规则当成"越严越好"——过度严格会诱发大量规避行为，规则要有取舍。

过渡：规则跑完会出报告，我们看一眼报告该怎么读。

【5.25】checkstyle 结果显示（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 Checkstyle 的检查结果。报告怎么读？我给你一个四步顺序。

第一步，**先看总数和分布**。报告会按规则统计违规条数。你要问的第一个问题不是"第一条是什么"，而是"**哪条规则报得最多**"。因为修一条高频规则，往往能一次消掉几百个问题——这是投入产出比最高的动作。

第二步，**看严重级别**。哪些是 error，哪些是 warning。团队应该约定：**error 必须修，warning 可以排队**。没有这个约定，报告就是一堆没人看的红字。

第三步，**定位到文件和行号**。报告会给出具体的文件、行、规则名和说明。这里有个使用技巧：**优先修"新改动涉及的"**。老代码里的历史违规先记账，不急着还——这就是前面说的"新代码零新增"。

第四步，**把高频违规反推成规范问题**。如果一条规则反复被踩，可能不是人的问题，而是**规范本身不合理**，或者**没人在新人入职时讲过**。这时候该改的是规范和培训，不是骂人。

我还想强调一句：报告的意义不在"清零"，在**趋势**。这个版本 800 条、下个版本 700 条，说明在收敛；这个版本 800、下个 1100，说明在恶化。**看趋势比看绝对值有用。**

> **板书：**四步：看分布（哪条规则最多）→ 分级别（error 必修／warning 排队）→ 定位（文件・行号・规则名）→ 反推（改规范或补培训）；右侧写"看趋势，不看清零"。

提问：报告显示某条规则报了 300 次，你第一反应是改代码还是改配置？

预设回答：

- 答"改代码，规范就是规范"——如果这条规则合理，那就该改；但 300 次说明这是系统性问题，改起来要排期。

- 答"改配置，把它关掉"——要看情况。如果这条规则在本项目确实不适用（比如自动生成的代码），关掉是对的；如果只是嫌烦，那是自欺。

- 追问：怎么判断？——**抽十条看**：如果是同一类真实问题，改代码；如果是场景不适用，改配置并写清理由。

易错点 / 考点：这一页常考"静态检查报告的使用方法"，要点是**分级、抓高频、看差异、看趋势**。易错点：把"违规数下降"直接等同于"质量提升"——也可能是规则被关掉了。

> **过渡：**Checkstyle 管风格和度量，但度量里的"复杂度"它只能给个阈值判断。要真正看清复杂度，得用专门的度量工具。

【5.26】SourceMonitor 检测代码复杂度（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲 **SourceMonitor**，一个专门做**代码度量**的工具。它和前面几个的区别是：它主要不是"报错"，而是**算数、画图、看趋势**。

它算哪些数？我们看几类最常用的。

代码行数——包括总行数、有效代码行、注释行。有效代码行和注释行的比例，能看出这个模块"好不好读"。

语句数——比行数更能反映逻辑量，因为一行可以写三条语句，也可以三条语句拆成十行。

复杂度——每个方法、每个类的复杂度值，这是它的核心指标。

最大块深度——嵌套最多嵌套了几层。一个方法里 `if` 套 `for` 套 `if` 套 `try`，深度到五层，那这段代码基本没人敢改。

注释比例和**方法、类的数量**——用来看结构和文档状况。

这些数字单独看没什么意思，**关键是横向比和纵向比**：横向比是同一个项目里哪个模块最"重"；纵向比是同一个模块这三个版本在变好还是变坏。

所以这类工具的典型用法是：**定期跑一次，看趋势图，找出"复杂度突然跳高"的那个模块**。突然跳高通常意味着有人在一个地方堆了太多逻辑——那就是下一次评审、下一次重构的重点。**度量工具的价值是"定位热点"，不是"给分"。**

> **板书：**指标列表：代码行／语句数／复杂度／最大块深度／注释比例／方法与类数量；下方画一条趋势线，标"突然跳高 = 重构热点"；右侧写"定位热点，不给分"。

> **提问：**一个模块的复杂度一直很高但很稳定，另一个模块复杂度不高但每次发布都在涨，你先管哪个？

> **预设回答：**

- 答"先管高的"——直觉答案，但高而稳定的模块可能已经被测试覆盖得很扎实。

- 答"先管在涨的"——我倾向这个。**上涨意味着失控趋势**，越早拦成本越低。

- 追问：那高而稳定的就放着不管？——不是不管，是**排期重构**，而不是紧急处理。这就是用趋势做优先级判断。

> **易错点 / 考点：**常考"代码度量指标有哪些"，要点：**行数、语句数、复杂度、嵌套深度、注释比例**。易错点：把"注释比例越高越好"当成结论——注释过多有时说明代码本身不清晰；教材口径是"适度"，重点看**有效注释**（解释为什么，而不是复述代码在做什么）。

> **过渡：**到目前为止，我们查的都是"自己写的代码"。可现代软件里，大部分代码不是你写的——是第三方库。那部分的风险，比你想的大得多。

【5.27】软件第三方库风险巨大（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是这一部分里最重要的一页，讲**第三方库的风险**。我先念课件给的三个数字，你感受一下量级。
>
> 第一个，**98% 的正在使用开源第三方库的公司，并不了解它们**。注意这句话的措辞——不是"没用"，是"不了解"。用了，但不知道用了什么、什么版本、谁在维护、有没有已知漏洞。
>
> 第二个，**每年有 4000 个新增的开源第三方库漏洞被报告**。这个数字说明漏洞不是一次性事件，是**持续流入**的。你今天扫干净了，下周新披露一个 CVE，你又中招了。
>
> 第三个，课件用一条时间轴给出了开源第三方库的使用比例在急速上升：2010 年是 10%，2015 年是 30%，2020 年是 90%。十年九倍。这意味着什么？意味着**今天一个系统里，你自己写的代码可能只占一成，九成是别人的。**
>
> 那问题就来了：你把九成的信任交给了一群你不认识、也没签合同的人。他们修不修、什么时候修、修了你要不要跟着升——这些你都不控制。
>
> 这就是**供应链风险**。它不是"某个库有 Bug"这么简单，它是**风险的传导链**：上游一个包被投毒或被披露高危漏洞，下游成千上万个系统同时中招，而你连自己用没用它都说不清。
>
> 所以这一页要建立的意识是：**你的系统边界，不是你代码的边界，是你依赖的边界。** 后面讲的 SCA 工具，就是为了把这条边界画出来。

> **板书：**三个数字：98%（不了解在用）／4000（每年新增漏洞）／10%→30%→90%（2010→2015→2020使用比例）；下方写一句"你的系统边界 = 依赖边界"，再写"供应链风险 = 传导链风险"。

> **提问：**你的项目引入了一个开源库，只用了一个工具类里的一个方法。这算不算引入了这个库的全部风险？

> **预设回答：**

- 答"不算，我只用了很小一部分"——这是最常见的误解。**漏洞不看你怎么用，看它在不在你的依赖树里**。它带的传递依赖、它的构建脚本、它的许可证，全都在你的交付物里。

- 答"算，依赖是整体的"——正确。你甚至可能没直接用到出问题的那段代码，但只要它被打进包里，攻击面就在。

- 追问：那怎么办？——**要么升级到修复版本，要么换实现，要么明确接受并记录这个风险**。三种都行，唯独不能"不知道"。

> **易错点 / 考点：**这三个数字是硬考点，尤其"**98% 的公司不了解自己在用的开源库**"和"**每年 4000 个新增漏洞**"。判断题"只要不调用有漏洞的方法就没有风险"——错。另一个易错点：把第三方库风险和"开源不安全"划等号——问题不在开源，在**缺乏可见性与持续跟踪**。

> **过渡：**既然风险来自两个方向——自己写的代码和引用的库，那安全分析自然也要分两条线。下一页把它们摆清楚。

【5.28】代码级安全分析对象（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把代码级的安全分析分成两条线，一张图讲清，我把它翻成一句话：**一条线查"我自己写错的"，一条线查"我引错版本的"。**
>
> 第一条线是 **SAST**，静态应用安全测试。它分析的是**你自己写的源代码**，找的是**未知漏洞**——所谓未知，是指"这个漏洞还没有被公开编号，是你的代码里特有的写法问题"。它依据的标准是 **CWE**，也就是通用缺陷枚举，比如注入、越界、路径穿越、不安全的反序列化。
>
> 第二条线是 **SCA**，软件成分分析。它分析的不是你的代码，是**你引用的组件**，找的是**已知漏洞**——已经有 CVE 编号、已经被公开披露、可能已经有修复版本的那些。它依据的标准是 **CVE**。

这两条线的差别，本质是"**未知 vs 已知**"。你自己的代码写错了，世界上没有第二个人知道，只能靠分析；你引的库出问题了，全世界都知道，只差你有没有去比对。

课件还给了三个比例数字，20%、35%、45%，用来说明这两类问题在整体安全风险里的分布——具体口径以课件为准，你记住"两类都占相当比重、都不能偏废"这个结论就够了。

落到实践上：**SAST 要接在提交和 CI 上，SCA 要接在依赖变更和定期扫描上**。两条线都跑，才叫完整的代码级安全分析。

板书：两列对照表：SAST｜SCA；自己写的代码｜引用的组件；未知漏洞｜已知漏洞；CWE｜CVE；提交与 CI｜依赖变更与定期扫描。旁边写"未知 vs 已知"。

提问：一个项目 SAST 扫出来零问题，SCA 扫出来十七个高危，你怎么判断该先做什么？

预设回答：

- 答"先修 SCA，因为有编号说明很明确"——合理，已知漏洞有现成修复版本，性价比最高。

- 答"先修 SAST，自己的代码更可控"——要看具体漏洞：能被远程触发的注入，比一个只影响内部工具的库漏洞更急。

- 追问：判断依据是什么？——**可达性与暴露面**：这个漏洞在不在攻击者能碰到的路径上。有编号不代表一定危险，没编号也不代表一定安全。

易错点 / 考点：SAST ↔ 未知漏洞 ↔ CWE、SCA ↔ 已知漏洞 ↔ CVE，这个对应关系是必考内容，四条要能一一对上。易错点：把 SCA 当成"代码扫描"——它不读你的代码逻辑，它读的是**依赖清单与版本号**。

【思政落点 3】：第三方库这条链，给工科生上了一堂很现实的责任课。你引一个包，可能只是图省事；但这个包会被打进交付物，进入客户的机房、进入别人的账户体系。**98% 的公司不知道自己用了什么**——这句话背后不是技术问题，是**知情意识**的缺失。将来你们在银行、券商做系统，一份依赖清单就是一份责任清单。对自己交付的每一个组件说得出"它是什么、谁维护、有什么风险"，这叫职业底线，也叫对用户负责。

过渡：SCA 这条线具体怎么做？下一页看它的工具形态。

【5.29】SCA 工具（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 SCA 工具的示意。工具的具体界面不重要，重要的是它的**工作流程**，一共四步，你把这四步记住，用哪家产品都一样。

第一步，**生成依赖清单**。工具先要弄清"你到底用了什么"——直接依赖、传递依赖、各自的版本号。这一步的产物现在有个正式名字叫 **SBOM**，软件物料清单。你可以把它理解成食品包装上的配料表。

第二步，**比对漏洞库**。拿着这份清单去和 CVE 库、厂商公告、开源社区的安全通告比对，看哪些版本命中了已知漏洞。

第三步，**评估影响**。命中了不代表一定可利用：这个组件是不是真的被打进运行包？出问题的那段代码在不在调用路径上？有没有其它防护措施？这一步必须人工参与，工具给的是候选清单。

第四步，**给出修复建议并跟踪**。通常是"升级到某版本""换用替代库""加临时缓解措施"。然后还要跟踪——因为漏洞库天天在更新，今天没问题不代表明天没问题，所以**定期重扫是必须的**。

我特别强调一下第一步的价值：**很多团队连"我用了什么"都答不出来。** 你问一个上线三年的系统用了哪些开源组件、分别什么版本，能立刻答出来的团队不多。这不是技术能力问题，是**可见性**问题。SCA 的第一个贡献不是修漏洞，是让你**看得见**。

> **板书：**四步链：依赖清单（SBOM）→ 比对漏洞库（CVE）→ 评估影响（可达性）→ 修复与跟踪（定期重扫）；旁边写"第一步的价值 = 可见性"。

> **提问：**SCA 报出某个传递依赖有高危漏洞，但你从来没有直接调用过它，该怎么处理？

> **预设回答：**

- 答"忽略，我又没用"——风险在于它可能被别的依赖调用，也可能在特定输入下被触发。

- 答"升级父依赖，让传递依赖跟着升"——这通常是正确路径，因为传递依赖你不能直接改版本，得从上游解。

- 追问：如果上游几年不更新呢？——那就是要**评估替代方案**或**记录并接受风险**，把决定写进文档，而不是让它在清单里躺着没人管。

> **易错点 / 考点：**SCA 的产出物是**依赖清单 + 已知漏洞匹配结果**，不是"代码缺陷列表"。常考的另一个点：**传递依赖也算在你头上**——很多人以为只要不直接引用就没事。

> **过渡：**SAST 和 SCA 都讲完了，那这些工具在工程里到底怎么用、什么时候用？下一页给了三种典型场景。

【5.30】代码分析工具常见的使用场景（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把代码分析工具的使用场景归成三类，还给了配套的检测项和耗时。这是一页"决策页"——它告诉你**什么场合该用什么强度的检查**。

场景一，**在 IDE 中实时分析**。你写代码的时候它就查了。特点是**快，几秒钟**；规则集**可选有限个检测项**——因为它要保证不卡你的编辑器。它的目的是**为开发者快速返回检查结果**。

场景二，**代码每次提交时分析**。提交或合并请求触发一次检查。速度比 IDE 慢，规则可以多一些。它的目的是**在代码进入主干之前拦住问题**，这是性价比最高的一道关。

场景三，**与 CI/CD 流水线集成**。每次构建、每晚定时跑全量。规则最全、分析最深，耗时可能就是**几分钟**。它的目的是**守住发布口**。

三类场景对应三种强度：**实时查轻的、提交查中的、发布前查重的**。这个"三档"结构你以后设计任何质量体系都能套用。

再看检测项举例：**SQL 注入、跨站脚本攻击、资源泄露、硬编码凭据、配置审查**。注意这里既有代码问题（注入、资源泄露），也有配置问题（硬编码凭据、配置审查）——说明现代代码分析不只看 `.java`，还看配置文件。

最后是那组关键词：**规则集、检测项、检测耗时**。三者是互相制约的：**规则越多，耗时越长，能放的位置就越靠后**。这就是为什么重型工具只能放在流水线末尾。

> **板书：**三档表：IDE 实时（几秒・有限检测项・快速反馈）｜每次提交（中等・拦截入库）｜CI/CD 集成（几分钟・全量・守发布口）；下列检测项：SQL 注入／XSS／资源泄露／硬编码凭据／配置审查；底写"规则↑ → 耗时↑ → 位置↓（越靠后）"。

> **提问：**如果只在 CI 上做全量分析，不在提交时做，会有什么后果？

> **预设回答：**

- 答"没后果，反正最后都会查"——有后果：问题会**堆积**。等到发布前才发现两百个问题，你只能选择延期或者忽略，两种都很糟。

- 答"反馈太慢，改起来成本高"——对。提交时发现问题，改动只要几分钟；发布前发现，要重新走一遍测试。

- 追问：那反过来，只在 IDE 里查呢？——更糟，因为 **IDE 里的规则是个人配置**，谁都可以关掉，没有约束力。

> **易错点 / 考点：**三档场景与"耗时—规则数—位置"的制约关系是核心考点。易错点：以为"检测项越多越好"——实际工程里要**按场景裁剪规则集**，否则要么卡死开发机，要么拖垮流水线。

> **过渡：**最后我们掀开盖子看一眼：这些 SAST 工具内部到底是怎么"看懂"你的代码的？下一页是它的通用框架。

【5.31】SAST 工具通用框架（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 SAST 工具的**通用框架**，信息密度很高，但结构其实很清楚。我按数据流向讲，你跟着箭头走一遍。
>
> 起点是**被测源程序**。第一步是**解析**：把源代码变成**抽象语法树**，也就是把"文本"变成"结构"。到这一步为止，工具只知道语法，不知道语义。
>
> 第二步是**基本分析方法**。课件列了一长串，我挑几个解释，你理解它们的共同目的就行：**函数调用**关系、**SSA**（静态单赋值，把每个变量赋值都变成唯一版本，方便追踪数据从哪来到哪去）、**指向分析**（这个指针/引用可能指向哪些对象）、**前后支配**（哪条语句一定在另一条之前执行）、**Mod-effect**（一个函数改了哪些外部状态）、**多态分析**、**全局变量分析**、**抽象解释**（用抽象值概括所有可能的运行状态）、**符号执行**（把输入当成符号，推演路径条件）、**图可达**（从"可控输入"到"危险操作"之间有没有路径）。
>
> 第三步，**缺陷模式**。工具不是凭感觉报错，它有一套**基于缺陷模式的逻辑表达式**，来源是 OWASP、CERT、CWE 这些标准。比如"外部可控的字符串未经校验就进入 SQL 拼接"，这就是一条模式。
>
> 第四步，**逻辑表达式求解**，常用 SMT 求解器。把"这条路径的条件"和"缺陷模式的条件"放在一起解，**能解出满足条件的输入，才算真的可能发生**。
>
> 第五步，**检测结果**。到这里它才给你报一条。
>
> 这个流程解释了一个你一定会遇到的现象：**误报**。因为中间每一步都在做近似——指向分析可能算多了、抽象解释可能放宽了、路径爆炸时只能截断。所以 SAST 的结果永远是"**疑似**"，需要人去确认。这也是为什么它不能替代评审和测试。

> **板书：**竖链：源程序 → 解析（AST）→ 基本分析（函数调用・SSA・指向・前后支配・Mod-effect・多

> 态・全局变量・抽象解释・符号执行・图可达）→ 缺陷模式（OWASP／CERT／CWE 的逻辑表达式）→ 求解（SMT）→ 检测结果；右侧写"每步都在近似 → 必然有误报"。

> **提问：**既然每一步都是近似、必然有误报，那为什么还要用它？

> **预设回答：**

- 答"因为人更不靠谱"——有点糙，但接近事实：人看不完几十万行，工具能。

- 答"因为它不漏那些规则覆盖的路径"——对，工具的价值是**稳定与覆盖**，不是精确。

- 追问：那怎么用好一个会误报的工具？——**降低噪音**：调规则集、按可达性排序、只对变更做门禁。工具的价值取决于你怎么用它，而不是它出厂什么样。

易错点 / 考点：这一页的考点是**流程顺序**和**关键术语**：AST、SSA（Static Single-Assignment）、CWE（Common Weakness Enumeration）、SMT。常考"SAST 的第一步是什么"——**解析成抽象语法树**；"误报产生的原因"——**分析过程做了近似/路径爆炸被截断**。易错点：把 SMT 求解当成"必须的一步"——某些轻量工具跳过它，代价是误报更高。

> **过渡：**第一部分讲完了——我们知道了怎么在不运行代码的情况下抓缺陷。可有些东西，不跑起来是真的看不出来。所以下面进入第二部分：单元测试。

【5.32】5.2 单元测试（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

我们进入第二部分，**单元测试**。这是本章重心，也是你实验课要动手的部分。

前面那部分我们一直在"看代码"，这一段开始**跑代码**。跑的对象很小——一个方法、一个类；但要求很硬：**你要能明确说出它"应该是什么行为"，然后用一段代码去证明。**

为什么它重要？一句话：**它是唯一能在"缺陷还没跑远"的时候把它按住的测试层次。** 到了集成、到了系统，一个缺陷混在一堆模块里，定位成本是几十倍。

这一部分我按六个问题讲：为什么要做？要做到什么程度？上下游没有怎么办（驱动和桩）？用什么工具写？依赖对象怎么处理（Mock）？怎么知道测得够不够（覆盖率）？

这六个问题，其实就是你写第一个测试时会依次卡住的地方。**卡住很正常，我们一个个解。**

> **板书：**写"5.2 单元测试：跑代码"；下方六个问号：为什么做／做到什么程度／上下游怎么办／用什么写／依赖怎么处理／够不够；右侧写"对象最小、反馈最快、定位最准"。

提问：单元测试和生产代码，谁应该先写？

预设回答：

- 答"当然先写代码"——最常见。先写实现再补测试，测试往往变成"照着实现写"，验证的是"代码现在这么干"，而不是"它应该这么干"。

- 答"先写测试"——这就是 TDD 的思路，好处是你先在脑子里把"什么叫对"想清楚。

- 追问：两种都能用吗？——能。**关键是测试和实现不要同一个人照着同一份理解抄一遍**，否则你只是把代码翻译了一次。

> **易错点 / 考点：**单元测试的"单元"指**可独立测试的最小功能单位**，通常是一个方法或一个类，**不是**一个模块、一个服务。判断题"单元测试由测试人员独立完成"——错，主责是开发人员。

> **过渡：**这一部分具体包括哪几块？我们看它的内容清单。

【5.33】单元测试内容（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是第二部分的**内容清单**，六项。它其实也是你这门课的六个知识点，我一项一项交代，顺便告诉你哪项会考、哪项要动手。
>
> 第一项，**为何要进行单元测试**——讲价值与成本。这一项看着像"务虚"，其实是最容易被问到"到底值不值"的地方。下一页专门讲。
>
> 第二项，**单元测试的目标和要求**——讲"测到什么程度算完"。这一项要背：功能、输入输出的容错、内部数据完整性、覆盖率指标。
>
> 第三项，**驱动程序和桩程序**——讲"被测的那个方法，上下游还没有怎么办"。这一项是本章的技术门槛，**驱动**是"我来调你"，**桩**是"你调别人，别人先用假的顶上"。理解这两个词，后面 Mock 就一通百通。
>
> 第四项，**单元测试工具 JUnit 及其实践**——这是你要动手的部分。JUnit 5 的架构、注解、断言、执行器，一条条都要会用。

第五项，**Mock 技术与框架（如 Mockito）**——讲"依赖的对象怎么造"。这是本章的难点，也是面试高频题。

第六项，**测试覆盖率工具 JaCoCo**——讲"我怎么知道测得够不够"。这一项的关键不是会点按钮，是**看得懂报告、知道行覆盖和分支覆盖的差别**。

六项合起来是一条完整的作业链：**想清楚为什么测 → 定标准 → 解决上下游 → 写出来 → 处理依赖 → 度量效果**。你实验交的那份测试代码，就是这条链的产物。

板书：六项竖排：为何测／目标与要求／驱动与桩／JUnit 实践／Mock 与 Mockito／覆盖率 JaCoCo；右侧标"要背：②④"、"要动手：④⑤⑥"、"难点：③⑤"。

提问：六项里，你觉得哪一项最容易被"跳过"？

预设回答：

- 答"覆盖率"——不少团队跑完不看报告，或者只看一个总数。

- 答"驱动和桩"——很多同学一上手就卡在这：被测方法要连数据库，怎么办？

- 追问：跳过哪一项后果最严重？——**跳过"目标与要求"最严重**，因为你连"测到什么程度算完"都没定义，后面全是凭感觉。

易错点 / 考点：这六项是本章的目录级考点，简答题常问"单元测试包括哪些内容"。易错点：把"驱动程序和桩程序"当成工具名——它们是**测试代码的角色**，不是软件产品。

【思政落点 4】：这六项排下来，你会发现单元测试是件"不显眼"的活：没有界面、没有演示效果、写完了也没人鼓掌。可它恰恰是**把地基做扎实**的那部分工作。我们这门课讲"课程设计"，很多同学愿意花两周做界面、花两天写测试。我想说的是：**决定一个系统能活多久的，往往不是它第一天多好看，而是它的每个零件经不经得起推敲。** 愿意在看不见的地方下功夫，这就是工匠精神最朴素的样子——不是把活干得漂亮，是把活干得住。

过渡：第一项我们先讲透——为什么非得在单元这一层花这个力气？

【5.34】为何要进行单元测试？（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页回答"值不值"。课件给了四组论据，我逐条讲，最后给你一个能说服人的算法。

第一条，**编程与编译运行结束后，每 1000 行代码中大约残留有 2 到 6 个 Bug**。注意语境：它不是"新手代码"，是行业常态。一个五万行的系统，天生就带着一两百个缺陷等着你。**缺陷不是意外，是常态。**

第二条，**寻找与修改程序错误的代价占总体开发投资的 30% 到 60%**。这条更扎心：开发投入里有三到六成其实花在"找错和改错"上。**测试不是额外成本，它就是开发成本的主要组成**——区别只在你在便宜的环节花，还是在贵的环节花。

第三条，**尽早发现错误，成本越低**。这条是前面两条的推论，也是最重要的行动指南：同一个缺陷，在单元阶段发现，改一行代码；在系统测试阶段发现，要重新部署、重造数据、重新回归；上线后发现，要发版、要道歉、可能还要赔钱。

第四条，**发现问题比较容易、修正问题更容易、单元质量是系统质量的基石**。为什么容易？单元小、边界清楚、能直接构造输入。为什么是"基石"？课件还补了一句关键的话：**单元测试可以比系统测试测得更彻底**——系统测试受限于场景，方法内部的分支组合你构造不出来，单元测试可以逐个分支喂数据。

最后看这两个算式，这是本页精华。**0.83 的 30 次方，约等于 0.00123；0.99 的 30 次方，约等于 0.74**。什么意思？一个系统由 30 个单元串起来，每个单元"通过率" 83%，整体一次跑通的概率只有千分之一左右；每个单元做到 99%，整体也才 74%。

这两个数字说明：**单元层面的缺陷率，会被集成按指数放大。** 指望系统测试阶段"整体调通"，等于指望 30 个不靠谱零件拼出一台靠谱机器。**所以质量必须在单元这一层做上去——它是唯一能让后面几层站得住的地方。**

> **板书：**四行论据：每千行残留 2–6 Bug／找错改错占总投入 30%–60%／越早越便宜／单元可测得更彻底；右侧写两个算式：$0.83^{30} \approx 0.00123$、$0.99^{30} \approx 0.74$；底部一行大字"单元缺陷率被集成指数放大"。

> **提问：**如果每个单元的通过率都是 99%，30 个单元串起来整体只有 74%。那你觉得"每个单元 99%"这个目标够不够？

> **预设回答：**

- 答"够了，99% 已经很高"——单看一个单元很高，串起来就掉到七成。这说明**"高"是相对的，要按整体目标倒推单元目标**。

- 答"不够，得靠集成测试补"——集成测试能兜一部分，但它兜不住组合爆炸。正确做法是两头一起做：单元抬高，集成专项覆盖接口。

- 追问：那 100% 呢？——100% 覆盖率不等于 100% 正确，这一点后面讲覆盖率时会专门说。**目标不是数字好**

看，是风险可控。

> **易错点 / 考点：**两个数字必背：**每 1000 行残留 2–6 个 Bug、找错改错占总投入 30%–60%**；两个算式要会解释：**$0.83^{30}\approx0.00123$、$0.99^{30}\approx0.74$**。易错点：把 0.83^{30} 理解成"缺陷率"——它算的是**整体一次通过的联合概率**，含义是"每个环节都成的概率相乘"。

> **【思政落点 5】**：这两个算式其实是**成本意识的数学表达**。工程上最贵的一句话是"先这样，回头再说"——因为"回头"意味着这个缺陷已经流到了更贵的环节。请你们记住：**质量不是免费的，但劣质更贵**。花时间在单元测试上，看起来是"慢"，实际上是把后面几轮返工的钱提前省下来。这种"算总账而不是算眼前账"的习惯，将来你们做技术决策、做预算、甚至做人生选择，都用得上。

> **过渡：**知道了为什么必须测，下一个问题就来了——测到什么程度算够？下一页给四条目标。

【5.35】单元测试的其它价值（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 上一页讲"为什么做单元测试"——早发现、好定位、修得便宜，那是算成本账。这一页回答另一个问题：如果单元测试只是抓 bug，凭什么要求每个函数都配一条用例？课件列了七条价值，我挑四条讲透。
>
> 第一条，**支持扩展**。课件原话：没有各种类型测试的支持，软件开发无法支持扩展，单元测试就是它的基础。假设你手里是电商结算模块，一开始只有"单品下单"，半年后要加团购、预售、秒杀。有一套跑得起来的单元测试，你改完跑一遍、绿了就敢提测；没有这套测试，团队第一反应一定是"别动那块代码，谁动谁背锅"。**没有测试的系统，扩展能力是被冻住的**——不是不能加功能，是没人敢加。
>
> 第二条，**引导更好的设计**。课件说得硬气：能够支持单元测试的代码总是不会被人遗弃。你要给一个方法写测试，就必须能把它单独拿出来跑，这一"拿"就照出三种病：参数太多、Monster Method 巨型方法、依赖过多。一个八百行的 `processOrder`，光构造入参就要 new 十几个对象，你有两条路——放弃测试，或者拆成六个职责单一的小方法；一拆，设计就好了。业内叫**可测性倒逼设计**。
>
> 第三条，**避免回归缺陷**。课件说：集成为庞大系统后，靠系统测试穷尽回归几乎不可能。三十万行的系统改一行工具类，系统测试跑一遍六小时只能抽样；单元测试两千条用例八秒跑完，还能跑到每个分支。**它不保证你没写错，它保证你写错了马上知道**。
>
> 第四条，**规范行为和记录代码**。理想状态下，单元测试就是被测代码的行为描述，也是一个例子，解释它怎

么工作、怎么实现某条业务逻辑。新人接手三年前的模块，注释可能过期、设计文档可能没写，但测试用例是真的——输入 5 和 0 期望返回 5。**这是一份可执行的需求文档，而且是唯一不会过期的文档，因为它跑不通就会红**。

剩下三条一句带过：**支持变化**——谁改代码都得知道哪些部分还要验证，测试给了这份名单；**平稳步调**——代码和测试配合着写，很少有最后一分钟的意外；**节省时间**——边界值、输入校验、主逻辑路径，用单元测试做最廉价。课件还引了一句话收尾：人们总愿意写"工作完全正确"的代码，可证明程序完全正确几乎不可能，所以单元测试的目的不是证明正确，是**持续提供证据**。

> **板书：**中间写"单元测试 ≠ 抓 bug"，四周四条：① 支撑扩展（敢改）→ ② 倒逼设计（可测性 = 好设计）→ ③ 回归安全网（八秒两千条）→ ④ 可执行的行为文档（不会过期）；右下角小字"＋变化・步调・省时"。

提问：你接手一个跑了五年、一行测试都没有的模块，领导要你两周内加一个新功能，你的第一步做什么？

预设回答：

- 答"先读代码，读懂了再改"——必要，但不够。读懂的判据不是"我觉得懂了"，而是"我能给它写出一条通过的测试"。

- 答"先补几条测试再动手"——这是正解。业内叫特征测试（characterization test）：先把现有行为钉住，再改。等于给旧代码穿一件防护服。

- 答"先重构，重构完再加功能"——顺序危险。没有测试保护的重构不是重构，是赌博，赌你还记得每一处隐含依赖。

- 收口：这三条其实是一条链——读懂了才写得出测试，有了测试才敢重构，重构干净了才好加功能。

易错点 / 考点：多选题考"单元测试的价值"，学生最容易漏掉"引导更好的设计"和"规范行为和记录代码"这两条，因为它们听起来不像"测试"的活。判断题"单元测试只能发现功能缺陷"是错的——它同样能暴露设计与可测性问题。记忆口诀：**扩展、设计、回归、文档**，四条一组背。

> **过渡：**价值讲起来容易空，落到"目标"就具体了。下一页把单元测试要达成的目标列成四条，其中第四条是一个数字，也是本课程最容易被误用的一条。

【5.36】单元测试的目标（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是考点密集页，四条目标请记准，尤其第四条那个数字。

第一条，**单元模块被正确实现**，主要指编码，包括功能、性能、安全性，课件补一句"一般主要介绍单元功能测试"。为什么？性能受调用方式影响，安全要看整个数据流，单元层面难以完整验证。但"一般不测"不等于"不该测"：一个正则写错导致回溯爆炸、一处 SQL 拼接没参数化，都是单元层面爆的。所以准确说法是**功能是主战场，性能和安全做基础检查**。

第二条，**参数输入是否正确传递并得到保护，也就是容错；输出是否正常**。这条最容易被做成"只测好数据"。什么叫得到保护？空值、超长字符串、类型边界、非法格式进来时有没有被挡住，还是直接把脏数据带给下游。线上大量事故，根子不在算错，而在**根本没检查输入**。

第三条，**内部数据能否保持完整性**，课件给三个点：变量的正确定义与引用、内存及时释放、全局变量的正确处理和影响最低。这是单元测试独有的地盘——系统测试从外面只能看输入输出，看不到"这个对象释放了没有"。举个 Java 例子：`static` 的 SimpleDateFormat 被两个线程同时用，格式化出来的日期会串；这种 bug 黑盒测一百遍可能都不出来，一条并发单元测试就现形。

第四条，**覆盖率达标**：代码行覆盖、分支覆盖或 MC/DC 达到要求，比如高于 80% 或 95%。三个词要分清：行覆盖看这一行执行过没有；分支覆盖看判定为真为假两条路都走过没有；MC/DC 是修正条件/判定覆盖，要求每个条件都能独立影响判定结果，航空轨交这类高安全领域常用。80% 和 95% 不是及格线，而是不同完整性等级的要求——一般业务系统 80% 行覆盖是基本盘，安全关键模块要 95% 以上甚至要求 MC/DC。

但比数字更重要的是这句：**覆盖率是必要不充分条件**。100% 只说明每行都跑过，不说明每行都断言过。一段代码跑了、你没写 assertEquals，它错了你照样绿。所以决定质量的不是覆盖率，是**你断言了什么**。

> **板书：**竖排四条目标：① 正确实现（功能为主，性能/安全做基础）→ ② 输入正确传递＋容错，输出正常 → ③ 内部数据完整性（变量/内存/全局）→ ④ 覆盖率（行・分支・MC/DC，80% 或 95%）；旁注一行红字"覆盖率高 ≠ 测得好，断言才算数"。

提问：一个模块行覆盖率 100%、分支覆盖率 100%，能不能说它测试充分了？请给一个反例。

预设回答：

- 答"可以，都 100% 了"——这是最典型的错答。覆盖率只记录"执行路径"，不记录"判断依据"。

- 答"不能，可能没写断言"——对。执行到但没有断言的代码，等于走过场。

- 答"不能，可能用例的输入选得不对"——也对，而且更狠：输入全用正常值，异常分支靠抛异常"顺便"覆盖过去，覆盖面好看，缺陷还在。

- 收口：所以覆盖率要配着"断言密度"和"输入设计"一起看，这三件套齐了才算测试充分。

> **易错点 / 考点：**选择题最爱考"单元测试的目标包括哪些"，正确答案四条都要选上，包括"内部数据完整性"这条抽象表述。另一个高频陷阱是把 **MC/DC** 说成"多条件覆盖"——它的全称是 Modified Condition/Decision Coverage，核心是"每个条件能独立改变判定结果"，比条件组合覆盖弱、比判定/条件覆盖强。

> **【思政落点 6】**：覆盖率这个数字，最考验一个人的**诚实**。因为它太好看了：写一条把整个方法跑一遍的用例、不写一句断言，覆盖率立刻 90%，报告很漂亮，缺陷一个没抓。工程上这叫"用指标交差"，代价是把风险留给了用户。请记住：**指标是给管理看的，断言才是给质量看的**。做技术的人，宁可报告上写 72%，也不要拿 100% 去骗自己和团队——这种对数据负责、不糊弄的态度，是工程师最基础的职业底线。

> **过渡：**目标讲完了，我们马上动手。下一页这个小练习只有十几行代码，但它是本课程最经典的一道题——四个人里有三个第一次看不出来。

【5.37】一个小练习（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 看这段代码。需求原文：实现两个**正整数或零**的加法，如果其中有 1 个数小于或等于 0，则返回 0。请找出错误，并说出什么测试数据能暴露它。
>
> 先读判定：`if(num1 >= 0 && num2 > 0)`。`num1` 是大于等于 0，`num2` 是**大于** 0。左边带等号，右边没带。而需求说零是合法的。所以 `num2 = 0` 时条件为假，程序走进 else 返回 0；正确答案应该是 `5 + 0 = 5`。**错误就在这一个等号上**。
>
> 这个 bug 为什么难发现？因为平时测函数下意识都用"正常值"：`add(2,3)` 得 5、`add(10,20)` 得 30、`add(-1,5)` 得 0，三个都对，测试通过，代码提交。要暴露它必须用**边界值**：`add(5,0)` 期望 5、实际 0，红了。零不是随便一个数，**零是两个合法区间的交界点**。
>
> 再深一层看覆盖。要让判定两条分支都跑到，最少两组数据：`(5,3)` 为真、`(-1,5)` 为假。可**判定覆盖做到了，这个 bug 依然活着**，因为 `num2=0` 这一格在"假"那一大类里被 `(-1,5)` 代表了。要让每个**条件**都取到

真和假，得补 `(5,0)`——`num1` 真、`num2` 假。这就是条件覆盖比判定覆盖更强的地方，也解释了为什么"分支覆盖率 100%"的代码照样有 bug。

工程教训：修法不是把 `>` 改成 `>=` 就完事，而是**把判定写清楚**——先做参数校验 `if (num1 < 0 || num2 < 0) return 0;` 再放心相加。**边界条件写在开头，比藏在复合判定里安全**。

请当堂在纸上写三件事：错误在哪一行、暴露它的数据是什么、修好之后补哪几条用例。三分钟，我们对答案。

板书：抄判定 `if(num1>=0 && num2>0)`，在 `num2>0` 上画圈打叉，旁边写 `>=`；下方列三行数据：`(5,3)→8 ✅`、`(-1,5)→0 ✅`、`(5,0)→0 ❌ 应为 5`；右侧写"判定覆盖 ≠ 条件覆盖"。

提问：`add(5, 0)` 这条用例，它同时完成了哪两种覆盖？它没完成哪一种？

预设回答：

- 答"完成了判定覆盖"——只对了一半。它让判定取到"假"，但"真"还得靠 `(5,3)` 那一条，两条合起来才叫判定覆盖。

- 答"完成了条件覆盖"——对。`num1>=0` 取真、`num2>0` 取假，两个条件都取到了真假中的一个；配合 `(-1,5)` 就齐了。

- 答"没完成路径覆盖/组合覆盖"——也对，四种条件组合只走到三种组合里的一部分，要全覆盖得再加 `(0,0)`、`(0,3)` 这类数据。

易错点 / 考点：这道题几乎年年考，两个坑：**一是答"没发现错误"，二是答"错误是 num1 应该写成 >0"**（把对称当正确，方向搞反）。答题模板固定三句：错误位置（`num2>0` 的边界）→ 暴露数据（`num2=0`，如 `add(5,0)`）→ 预期与实际（预期 5，实际 0）。判断口诀：**看边界，不看中间；等号两边，必须对齐需求**。

过渡：单个函数好测，因为它没有"邻居"。可真实的函数总要调用别人、也被别人调用。下一页开始讲：怎么给一个函数搭出"假邻居"，让它在孤岛上被测。

【5.38】驱动程序与桩程序（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是标题页，但它是本节最关键的入口：单元测试的两件"脚手架"——**驱动程序（Driver）**和**桩程序（Stub）**。概念我先给一句话版，下一页看图。

先说为什么需要它们。单元测试的对象是一个单元，可能是一个函数、一个类。可现实中的单元不是孤岛：上面有调用者，下面有被调用者——要查数据库、发消息、调支付接口。做单元测试时，**上下两头都不在**：调用者还没写好，或者我不想连带它一起测；被调用者可能是别人的模块、可能是生产库、可能是收费的第三方接口。

怎么办？**上面缺一个"假调用者"，下面缺一个"假被调用者"**。假调用者就是驱动程序：准备数据、调用被测单元、拿回结果。假被调用者就是桩程序：顶替真实的下游，按我需要的样子返回结果。

打个比方：你要考一个学生的计算能力，又不想让其他环节干扰他，于是找个人专门出题收卷，这是驱动；再把"查表""计算器"换成一份固定答案表，让他不用真去查，这是桩。**驱动是上游的替身，桩是下游的替身**，一个喂进来，一个顶出去。

这两个词的方位，考试最爱设陷阱：**驱动力在被测单元之上**，它扮演"主程序"，通常由测试框架提供——你写的 `@Test` 方法本身就是驱动；**桩在被测单元之下**，它扮演"被依赖者"，需要你自己写小类去冒充。

> **板书：**中间画方框写"被测单元 Function under test"；上方箭头写"Driver（假调用者：造数据、发调用、收结果）"，下方箭头写"Stub（假被调用者：按需返回）"；右侧竖排一句"上一个、下一个，都不是真的"。

> **提问：**你写了一个 `@Test` 方法，里面 new 了一个被测类、传了两个参数、断言返回值——这里面哪些部分相当于驱动，哪些相当于桩？

> **预设回答：**

- 答"整个测试方法就是驱动"——对，JUnit 的 Runner 加上你的测试方法，共同构成驱动。

- 答"没用到桩，因为这个类没有下游依赖"——正确。桩不是必需品，只有被测单元真的调用了外部协作者时才需要。

- 答"传进去的两个参数就是桩"——概念串了。参数是**数据**，桩是**对象**（一个实现了相同接口的替身类）。这个区分后面讲 Mock 时还会用到。

> **易错点 / 考点：**名词方位必考。判断三步法：**看它调用谁**——它调用被测单元，是驱动；被测单元调用它，是桩。另一个高频错答是把"桩"和"Mock"当成同义词，这个差别我们在 5.59、5.60 两页专门讲。

> **过渡：**光说方位还是抽象，我们看那张三层结构图，一眼就明白。

【5.39】驱动程序与桩程序（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这张图是单元测试的标准骨架，三层，从中间说起。

中间一层是**被测单元**（Function under test），我们要真正测的那个函数或类，它是舞台中央的主角，不动。

上面一层是 **Driver，驱动程序**，职责三件：准备输入数据、调用被测单元、接收并判断输出。所以你写的 `@Test` 方法、那个 `main` 函数、测试框架的 Runner，本质上都是驱动。**驱动是可以通用的**——框架把"调用、捕获、比对、报告"都封装好了，你只管填数据和期望值。

下面一层是 **Stub，桩程序**。被测单元运行时要去调用别人，那些"别人"在单元测试环境里通常不可用或不该用：数据库没测试数据、支付接口不能真扣钱、消息队列没起。桩就顶在那儿，用固定返回值替代真实调用。图上的箭头方向很关键：**被测单元指向桩**，说明是被测单元去调用桩。

为什么一定要这么搭？三条理由。**隔离**：一个单元测失败，你要能立刻确定问题在它自己身上，而不是"可能数据库挂了"。**可控**：真实下游你控制不了，它今天成功明天超时，你的测试就时红时绿，这种测试没人信；桩把"下游返回什么"变成一行代码。**速度**：连库的一条用例几百毫秒，用桩是微秒级，几千条用例就是半小时和几秒的差距。

补一个工程判据：**什么时候必须上桩？** 满足三条之一——跨进程（网络、数据库、消息）、不可重复（时间、随机数、流水号）、有副作用（扣款、发短信、写外部日志）。三条都不满足的，老实调真的，别为了"纯"而纯。

> **板书：**三层堆叠图：上"Driver（准备数据→调用→判断输出）"、中"被测单元 Function under test"、下"Stub（固定返回值替代真实下游）"；右侧三条"必须上桩"：跨进程／不可重复／有副作用。

提问：如果一个被测方法里调用了 `System.currentTimeMillis()` 取当前时间，你要不要为它做桩？为什么？

预设回答：

- 答"不用，取个时间而已"——那就踩坑了。这类测试会在跨月、跨年、闰秒、夏令时那天莫名其妙地红，而且没人查得出原因。

- 答"要，因为时间不可重复"——正确，这属于"不可重复"那一类。做法是把取时间抽成一个可注入的 `Clock`

接口，测试时注入固定时间。

- 追问：这又回到 5.35 那条"引导更好的设计"——**为了可测，你把时间依赖抽出来了，代码反而更干净**。测试逼设计，就是这个意思。

> **易错点 / 考点：**图上的箭头方向是考点：**驱动 → 被测单元 → 桩**，一条链。如果考生画成"桩 → 被测单元 → 驱动"，说明方位全反了。另常考"驱动和桩由谁来写"：驱动多由测试框架提供（也可手写），**桩必须由测试人员自己写**，因为只有你知道下游该返回什么。

> **【思政落点 7】**：驱动和桩，本质是**为协作搭桥**。被测单元本身没变，是周围的环境被"补"出来了——上游有人喂数据，下游有人应答。这跟我们做团队项目一模一样：一件事推不动，往往不是这个人不行，而是上下游没接上。**愿意去写那个"假的下游"的人，其实是在替整个团队承担衔接成本**。请记住这种角色：不一定出现在架构图里，但没有他，谁都跑不起来。这种"我来补位"的意识，就是协作精神最实的样子。

> **过渡：**桩写起来容易，写好很难。最常见的写法是"返回一个写死的 5"，下一页告诉你为什么这是最不聪明的做法。

【5.40】桩程序的灵活性（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 C# 代码，`ParameterizedStub`——参数化的桩。我们看它比"硬编码的桩"强在哪。
>
> 它长这样：类实现了协作接口 `ICollaborator`，构造函数收一个 `value` 存起来，然后在 `ComputeAndReturnValue()` 里判断——`value` 小于 10 就抛 `InvalidOperationException`，否则返回 `value`。关键在于：**返回什么由构造时传进来的参数决定**，不是写死在方法体里。
>
> 课件底下那句话很直接：不会像上一页返回一个硬编码数值；**让一个桩对象返回单个数值是最简单、但也是最不聪明的做法**，因为其它测试迟早都需要返回一个不同的数值。
>
> 为什么不聪明？第一个测试要返回 100，你写个 `return 100` 的桩；第二个要返回 0，再写一个类；第三个要抛异常，第三个类。三个测试三个类，代码量比被测代码还多，而且 90% 是一样的。这叫**测试代码的复制粘贴**——接口一改，你要改三处。
>
> 参数化之后一个类搞定：`new ParameterizedStub(100)`、`new ParameterizedStub(0)`，要异常就传

5。**变化的部分从"类的个数"变成了"构造参数"**，这就是灵活性。

这段代码还有两个细节值得学：第一，它把**异常路径**也做进了桩——真实下游会失败，你的桩也要能失败，否则容错代码永远测不到；第二，它的判断逻辑是"假业务"，但这是刻意的——桩只需要满足测试需要的行为，不必还原真实逻辑。

一句总结：**桩的价值不在"能返回"，而在"能按需返回"**。写桩前先问自己：下一个测试会不会要一个不同的值？会，就别硬编码。

> **板书：**左侧写"硬编码桩：`return 100` → 每个用例一个类 → 复制粘贴"；右侧写"参数化桩：`new ParameterizedStub(100/0/5)` → 一个类覆盖正常・边界・异常"；中间一句"变化点从『类的个数』移到『构造参数』"。

> **提问：**桩里能不能写业务判断逻辑，比如"金额大于 1000 就打九折"？写了会不会让测试变假？

> **预设回答：**

- 答"不能，桩要越简单越好"——方向对，但过严。桩里可以写逻辑，前提是这段逻辑**只为测试服务**、且你能说清它模拟的是什么。

- 答"能，只要能造出想要的场景"——可以接受，但要当心：桩里的逻辑一旦复杂，你就在测"桩"而不是测"被测代码"，缺陷会被桩掩盖。

- 收口：判断标准是**这段逻辑会不会影响断言的正确性**。会，就说明它该搬到参数里；不会，写一点无妨。

> **易错点 / 考点：**常考"硬编码桩的问题"——标准答案是"复用性差、用例一多就要写多个桩类、维护成本高"。另一个易错点是把"参数化桩"和"Mock 的 when 配置"混为一谈：前者是你自己写的类，后者是框架动态生成的代理对象，机制不同、效果相近。

> **过渡：**到这里，单元测试的"为什么测、测什么、怎么搭台子"都齐了。接下来进入工具，从一份工具清单开始。

【5.41】单元测试工具及其应用（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是本节的分隔页，任务只有一个：把"工具"的地图铺开。

为什么单独立一节讲工具？因为单元测试是**所有测试类型里最依赖工具的一种**。系统测试还能靠人点、靠脚本点；单元测试面对几百上千个函数，手工根本不可能。业内说法是：**没有框架就没有单元测试**——你不可能自己造一套"发现用例、执行、捕获异常、汇总报告"的机制，那本身就是个框架。

这一节带你过四类工具，后面每页都能归位：

第一类**单元测试框架**——JUnit、TestNG、GoogleTest、pytest，负责组织用例、执行、断言、报告，这是地基。第二类**测试替身与 Mock 框架**——JMockit、Mockito、EasyMock，负责造假协作者，把上一页我们手写桩的活自动化。第三类**用例自动生成工具**——EvoSuite、Diffblue Cover，基于遗传算法或符号执行自动生成能提高覆盖率的测试，注意它们是**起点不是终点**。第四类**覆盖率工具**——JaCoCo、gcov/lcov、Coverage.py，回答"我到底测到哪儿了"。

四类的关系一句话串起来：**框架让你写得出，替身让你写得下去，生成工具让你写得快，覆盖率工具让你知道写够了没有**。四件套缺一件，工程上都跑不起来——只有框架没覆盖率，你不知道测够没有；只有覆盖率没替身，稍微复杂的类根本测不动。

顺带说本课程的实操路线：课堂和实验用 Java 生态，**JUnit 5 + Mockito + JaCoCo**。这三样在 Java 项目里占有率最高，学一套够用很久。

> **板书：**四行分类：① 框架（JUnit／TestNG／GoogleTest／pytest）② 替身与 Mock（Mockito／JMockit／EasyMock）③ 自动生成（EvoSuite／Diffblue Cover）④ 覆盖率（JaCoCo／gcov／Coverage.py）；右侧一句话"写得出・写得下去・写得快・知道够没够"。

提问：如果团队只允许你引入一个单元测试相关的工具，你选哪一个？为什么？

预设回答：

- 答"覆盖率工具"——理由通常是想先量化现状。可行，但没有框架就没有用例可跑，覆盖率会是 0，量化的意义不大。

- 答"测试框架，比如 JUnit"——多数情况下的正解。框架是地基，其它三类都是它的插件或配套。

- 答"自动生成工具，一步到位"——听着高效，实际风险最大：生成的用例断言往往是"照抄当前行为"，**它会把 bug 一起固化下来**。

易错点 / 考点：这一页的考点是"工具分类"。最容易错的是把 **EvoSuite／Diffblue Cover** 归到"Mock 工具"里——它们是用例生成工具。另一个常见错答是把 JaCoCo 说成"测试框架"，它是覆盖率库，靠 JVM

agent 插桩工作。

> **过渡：**分类讲完了，下一页把这几类的具体名字摆出来，出处是维基百科那份单元测试框架清单。

【5.42】单元测试工具（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页只给了一个出处——维基百科的"单元测试框架列表"。课件把它当入口，不是当要背的清单。我借它讲三件事。

第一，**"框架列表"本身就说明行业成熟度**。一个语言生态成不成熟，看它有没有成体系的单元测试框架就够了：Java 有 JUnit、TestNG；C++ 有 GoogleTest、Catch2；Python 有 unittest、pytest；JavaScript 有 Jest、Mocha；Go 更狠，`go test` 直接做进语言工具链。**越主流的语言，测试框架越像空气——你不用特意选，它就在那儿**。

第二，**选框架看三件事，不看功能多少**：① 生态——IDE 支不支持、Maven／Gradle 能不能一键跑、能不能出覆盖率报告；② 团队熟悉度——团队用 pytest 用得顺，你非换 JUnit 就是纯增成本；③ 与 CI 的集成成本——能不能只跑失败的用例、能不能产出标准报告格式。这三条比"哪个断言 API 更优雅"重要一百倍。

第三，**这份清单是历史地图**。很多框架名字带 xUnit——JUnit、NUnit、xUnit.net、PHPUnit，都源自 Smalltalk 的 SUnit 那套设计：测试类、测试方法、断言、前后置处理、测试套件、结果汇总。**这套设计活三十年没被推翻**，说明它抓住了本质：测试就是"准备—执行—断言—清理"四步，任何框架都是给这四步换皮。

所以这一页当**坐标系**用：以后换语言、换公司、换项目，先找"这个生态的 JUnit 是谁"，剩下可以平移。

> **板书：**写"xUnit 家族"→ 下面拉四条：JUnit（Java）／NUnit・xUnit.net（.NET）／GoogleTest（C++）／pytest・unittest（Python）；右侧写"选框架看三条：生态・熟悉度・CI 集成"，下方一行"四步不变：准备→执行→断言→清理"。

提问：如果一门语言连一个像样的单元测试框架都没有，你会怎么判断这个技术选型的风险？

预设回答：

- 答"那就别用这门语言"——结论可能对，但理由要更细：没有框架意味着团队要自己搭测试基础设施，成本会翻几倍。

- 答"风险高，因为没法做自动化测试"——对，而且后果是**回归只能靠人**，人一累就漏，漏了就在线上见。

- 追问：所以选技术栈时，"有没有成熟的测试生态"应该和"性能好不好""社区活不活跃"放在同一张评估表里——这也是很多团队踩过坑才学会的。

> **易错点 / 考点：**这一页不考清单，考的是"框架的共同结构"。常考简答："xUnit 系列框架一般包含哪些要素？"标准答案：测试类与测试方法、断言机制、前置/后置处理（setUp/tearDown）、测试套件组织、结果报告。别答成"JUnit 有 @Test"这种具体到某一家的细节。

> **过渡：**清单太宽，我们把镜头收窄到 Java 和 Python 上最常用的那几个名字，看它们的定位差别。

【5.43】最常用的单元测试工具（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是干货清单，我按四类工具给你归位，说清每个工具**解决什么问题**。

第一类**框架**：JUnit、TestNG 是 Java 两大主力，GoogleTest 是 C++ 标杆，pytest 和 unittest 是 Python 两条路——unittest 是标准库自带、风格像 JUnit，pytest 更 Pythonic、直接写 `assert`。职责都是组织、执行、报结果。

第二类**智能化用例自动生成**，课件点了两个：**EvoSuite** 用遗传算法搜索能提高覆盖率的输入，最后生成 JUnit 用例；**Diffblue Cover** 是商业产品，主打给存量 Java 代码自动补测试。定位我说重一点：**它们是起点工具，不是终点工具**。因为它们生成的断言，本质是从当前代码行为反推的——**代码错了，它就把错误行为写成期望值**。所以生成完必须由人审断言，尤其业务规则那几条。

第三类 **Mock 工具**，课件写的是 **JMockit**：能力最强，能 mock 静态方法、构造器、私有方法，但和 JVM 版本耦合最紧。后面重点讲更主流更稳的 Mockito。

第四类**覆盖率工具**：Java 是 **JaCoCo**；C/C++ 是 **gcov**，配 **lcov** 出 HTML、**gcovr** 做汇总和 XML；Python 是 **Coverage.py**。原理都是"埋点统计哪些行被执行过"，具体机制后面讲。

课件那句要划重点的话：Java、C++、Python 里受欢迎的就是这四类——**框架、用例自动生成、Mock、覆盖率**。各认一个主力，工具链就齐了：JUnit ＋ Mockito ＋ JaCoCo，也是我们实验用的组合。

顺手纠一个笔误：前面第 18、19 页把 SonarLint 和 SonarQube 写成了 `SolarLint`、`SolarQube`，那是

> 拼错了，正确是 **SonarLint**（IDE 里的静态检查插件）和 **SonarQube**（服务端代码质量平台），答题写报告按正确拼写来。

> **板书：**四栏对照表：框架（JUnit／TestNG／GoogleTest／pytest／unittest）｜自动生成（EvoSuite／Diffblue Cover）｜Mock（JMockit）｜覆盖率（JaCoCo／gcov・lcov・gcovr／Coverage.py）；底部一行"实验组合：JUnit 5 ＋ Mockito ＋ JaCoCo"；右上角订正框"SonarLint／SonarQube（课件误拼 SolarLint／SolarQube）"。

> **提问：**Diffblue Cover 给一个老模块自动生成了一千条用例，覆盖率从 20% 冲到 85%，能不能直接合并进主干？

> **预设回答：**

- 答"能，覆盖率涨了就是好事"——风险最大。自动生成的断言固化的是**现状**，不是**需求**；如果现状本身有 bug，你就把 bug 写成了"期望行为"。

- 答"不能，要人工审断言"——正解。至少要抽查业务规则相关的用例，确认期望值是从需求来的。

- 答"可以先合并，再慢慢审"——不推荐。一旦进了主干，后面所有人都以为这是"正确行为"，改起来阻力更大。宁可分批合，也要先审。

> **易错点 / 考点：**最爱考的是**工具归类**：EvoSuite／Diffblue Cover 属于"用例自动生成"，不是"Mock 工具"也不是"覆盖率工具"；JMockit 是 Mock 工具，不是框架。第二个考点是 **gcov 家族的分工**：gcov 采集、lcov 出报告、gcovr 出汇总——三者别混答。

> **过渡：**工具认完了，回到 Java 的主线。下一页开始讲 JUnit，先从它的老版本 JUnit 4 的工作原理说起，因为那几个类名到现在还在面试题里出现。

【5.44】JUnit 4 工作原理（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是标题页，我先给总纲，下一页看那四个类的分工。
>
> 为什么讲 JUnit 4 而不是直接讲 5？两个原因。第一，**JUnit 5 是 2017 年才发布的，绝大多数存量项目和面试题还是 JUnit 4 的口径**——你以后接手老代码，看到的还是 `org.junit.Test`、还是 `@Before`／

`@After`／`@RunWith`。第二，JUnit 5 的架构是在 4 的基础上拆出来的，**你懂了 4 的"四件套"，才看得懂 5 的三层架构为什么这么分**。

JUnit 4 的核心设计一句话：**用注解标记测试，用运行器驱动执行，用结果对象收集结论**。注解是`@Test`、`@Before`、`@After`、`@BeforeClass`、`@AfterClass`、`@Ignore`；运行器是`Runner`及其子类；结果是`Result`和`RunListener`。

这里有个设计上的小心思：**一个测试方法失败，不能影响其它方法继续跑**。所以 JUnit 给每个方法套了一层隔离——新建实例、捕获异常、记录下来、继续下一个。你在`@Test`里抛异常，框架不会崩，它只是把这个方法标成"错误"。这就是"框架"和你自己写个 main 循环调用最大的差别。

下一页把参与这件事的四个类摆在一起：`TestRunner`、`TestSuite`、`TestCase`、`TestResult`。课件管它们叫"成员三重唱"，实际上是四个角色。

板书：写"JUnit 4 = 注解 + 运行器 + 结果"；下方三行：`@Test`／`@Before`／`@After`／`@BeforeClass`／`@AfterClass`／`@Ignore`；`Runner`、`Result`、`RunListener`；右侧一句"一个方法失败，不影响其它方法"。

提问：为什么 JUnit 要给每个测试方法都新建一次测试类的实例，而不是所有方法共用一个对象？

预设回答：

- 答"为了隔离，避免上一个方法改了字段影响下一个"——正确，这是最核心的理由，测试方法之间不能有隐式依赖。

- 答"因为 Java 只能这么写"——不是，共用实例技术上完全可行（JUnit 4 里用`@FixMethodOrder`也只是控制顺序，不共享状态）。

- 追问：那如果某个对象**很贵**、所有方法都要用，怎么办？答案是`@BeforeClass`级别的静态资源，或者干脆用依赖注入把共享部分管起来——**共享要显式写出来，不能靠"反正上一个方法已经初始化了"**。

易错点 / 考点：常考"JUnit 4 与 JUnit 5 的注解对应关系"：`@Before`→`@BeforeEach`、`@After`→`@AfterEach`、`@BeforeClass`→`@BeforeAll`、`@AfterClass`→`@AfterAll`、`@Ignore`→`@Disabled`。这个对照表必须背下来，选择题年年出。

过渡：四个角色怎么串成一条流水线，看下一页。

【5.45】JUnit 工作原理 – 续（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 JUnit 4 的核心机制。课件原话：**一个 `TestRunner` 运行一个 `TestSuite`，这个 `TestSuite` 可以由一个或多个 `TestCase`（或由其它 `TestSuite`）组成；运行结果由 `TestResult` 收集，由 `TestRunner` 报告。** 四个角色，我用考试场景对齐，一次记住。

`TestCase` 是考生。一个测试类，里面每个 `@Test` 方法是一道题。它知道怎么答题（执行方法）、怎么自评（断言）。

`TestSuite` 是考场。它把考生组织起来，而且可以嵌套——套件里装套件，就像"上午场"里分"数学考场""语文考场"。这就是你能按包、按类、按标签筛选用例的原因。

`TestResult` 是成绩单。它只干一件事：收集。收集每道题的结论，结论分三类：**成功、失败（failure）、错误（error）**。

这里有个必考区分：**failure 和 error 不是一回事**。failure 是断言不成立——程序跑完了，结果与期望不符。error 是执行中抛了异常——程序根本没跑到断言，这可能是被测代码的问题，**也可能是测试代码自己写错**。所以 `Failures: 2, Errors: 1` 是两种不同性质的信号。

`TestRunner` 是监考老师兼广播员：启动流程，然后报告结果——控制台那一条条绿点红叉、IDE 左边那棵树、CI 上那份报告，都是它报的。

这里还藏着一个架构洞察：`TestSuite` 用**组合模式**把"单个用例"和"一组用例"统一成同一个 `Test` 接口。所以跑整个项目、跑一个包、跑一个类、跑一个方法，对上层是同一种操作。**这个设计被 JUnit 5 继承下来，变成了 `TestEngine` 发现并执行 `TestDescriptor` 树**——你现在理解的这套结构，换个名字还能用十年。

> **板书：**四个角色对齐：`TestCase`＝考生（一个测试类／一个 @Test 一道题）→ `TestSuite`＝考场（可嵌套）→ `TestResult`＝成绩单（成功／failure／error）→ `TestRunner`＝监考＋广播（启动＋报告）；右侧一句"组合模式：单个和一组，都是 Test"。

> **提问：**报告上显示 `Failures: 0, Errors: 3`，你的第一反应应该是什么？这和 `Failures: 3, Errors: 0` 有什么不同的排查路径？

> **预设回答：**

- 答"三个都失败了，去改代码"——太粗。Errors 表示压根没跑到断言，先要看异常栈。

- 答"Errors 是抛异常，可能是测试代码自己写错了"——很准。常见的是桩没配对、构造参数给错、依赖没注入，这类问题先看"是不是测试环境的问题"。

- 答"Failures 才是真失败，Errors 是环境问题"——不完全对。Errors 也可能是被测代码的空指针，那就是真缺陷。所以正确姿势是**先看异常类型和栈顶**，再判断归属。

> **易错点 / 考点：**必考区分 —— **failure（断言失败）vs error（异常中断）**；以及 **TestSuite 可以包含 TestSuite**（组合模式）。还有个常见错答："TestResult 负责打印报告"——不对，**收集是 TestResult，报告是 TestRunner**，这两个职责在 JUnit 5 里被拆得更清楚。

> **过渡：**JUnit 4 是一整块，JUnit 5 把它拆成了三层。这个拆分是理解现代测试框架的钥匙。

【5.46】JUnit 5 架构（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是架构核心，一句话必须背下来：**JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**。这不是三个版本，是**三个职责不同的子项目**，今天拼在一起发布。
>
> 第一块，**JUnit Platform**，基于 JVM 的执行测试的基础框架及其 Test-Engine API。翻成人话：它是**插座和电网**，不管你怎么写测试，只负责发现测试、调度执行、把结果吐出来。它还提供两样：一个**控制台启动器**（命令行就能启动平台），以及给 **Gradle 和 Maven 构建插件的支持**——这就是 `mvn test` 能跑、也能接进 CI/CD 的原因。还有：Platform 自带一个**基于 JUnit 4 的 Runner**，旧测试不用改就能跑。
>
> 第二块，**JUnit Jupiter**，是 JUnit 5 里编写测试和扩展的新编程模型与扩展模型的组合，并提供 Test-Engine 让基于 Jupiter 写的测试在平台上运行。翻成人话：**新国标插头**——你天天写的 `@Test`、`@BeforeEach`、`Assertions.assertEquals`、`@ParameterizedTest` 全来自它，是你最常打交道的一块。
>
> 第三块，**JUnit Vintage**，提供 TestEngine 让平台能跑**基于 JUnit 3 和 JUnit 4 的测试**。翻成人话：**转接头**。公司有十万条存量用例，不可能一夜重写，Vintage 就是那座桥。
>
> 为什么这个拆分是神来之笔？它把**执行**和**编写**彻底解耦了。Platform 只认 Test-Engine 这个接口——谁实现它谁就能上平台，于是 TestNG、Spock 都能挂上来。**这是"面向接口而不是面向实现"的一次设计胜利**。两个

实践点：Vintage 是**可选依赖**，纯新项目可以不引；从 4 迁 5 的标准做法是"先引 Vintage 保证存量能跑，再逐模块改写，最后摘掉"——**不要停机大改**。

板书：三层结构图：上"JUnit Jupiter（新编程模型：@Test／@BeforeEach／Assertions）"、中"JUnit Platform（Test-Engine API＋控制台启动器＋Gradle/Maven 插件＋JUnit4 Runner）"、下"JUnit Vintage（跑 JUnit 3／4 的转接头）"；右侧一句"执行与编写解耦：谁实现 Test-Engine，谁就能上平台"。

提问：一个项目同时有 JUnit 4 的老测试和 JUnit 5 的新测试，需要引哪几块？如果只引 Platform ＋ Jupiter 会发生什么？

预设回答：

- 答"三块都引"——正确。Jupiter 跑新的，Vintage 跑旧的，Platform 调度。

- 答"只引 Platform＋Jupiter 也行，JUnit 4 的用例会被忽略"——现象描述对，后果要说清：**旧用例不是"跑失败"，是"根本没被发现"**，覆盖率会莫名掉一大截，这种"静默丢失"最危险。

- 追问：那怎么确认旧用例真的在跑？看报告里的用例总数和类名，或者给 Vintage 单独跑一次——**数量对不上，就是没跑**。

易错点 / 考点：必考"JUnit 5 由哪三部分组成及各自职责"。三个高频错答：① 说"JUnit 5 就是 Jupiter"（漏了 Platform 和 Vintage）；② 把 Vintage 说成"旧版本兼容包"就完了——要说出它是"提供 TestEngine 让平台跑 JUnit 3/4 测试"；③ 把 Platform 说成"IDE 插件"。记忆口诀：**平台管跑、朱庇特管写、复古管兼容**。

过渡：架构是骨头，接下来是肉——你每天真正会用到的那三个组件。

【5.47】需要了解 JUnit 的关键组件（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把 JUnit 拆成三个你每天真正会碰的组件：**Assert 类、Annotation、Runner**。

Assert 类。课件定义：它是 JUnit 框架中非常重要的类，用于判断所返回的**实际结果**与**预期结果**是否相符。注意主语——它判断的是"结果"，不是"过程"。所以断言是整个测试的**判决环节**：前面准备数据、调用方法都是取证，断言才是宣判。**没有断言的测试不是测试，是"能跑通就算过"的演示代码**。

Annotation。课件定义：JUnit 中以 `@` 引导，用来标注测试类中不同方法的功能。一句话说清它的价值：

它把"这个方法是什么角色"从命名约定变成了显式声明。JUnit 3 时代方法名必须以 `test` 开头才会被执行；JUnit 4 之后你写 `@Test` 就行，方法名叫 `should_return_zero_when_num2_is_zero` 也没问题——名字终于能用来表达意图，而不是讨好框架。

Runner。课件定义：JUnit 通过运行器 Runner 来运行和显示测试执行结果。它是"发动机＋仪表盘"，负责启动，也负责呈现。JUnit 5 里它的职责被 Platform 的 Test-Engine 接管了，但概念没变——总有一个东西在发现并驱动你的测试。

三者是一个闭环：**Annotation 告诉框架有哪些测试，Runner 把它们跑起来，Assert 决定哪条通过**。缺一不可——没注解框架找不到你，没运行器跑不起来，没断言永远绿。

这也是排查"测试没生效"的三步法：先看注解加没加（是不是被误标 `@Disabled`）、再看运行器扫到没（包路径、命名过滤）、最后看断言写没写。

板书：三角关系图：`Annotation`（有哪些测试）→ `Runner`（跑起来＋报结果）→ `Assert`（判通过）；下方写排查三步："注解加没加 → 运行器扫到没 → 断言写没写"。

提问：一个测试方法加了 `@Test`、能跑、报告是绿的，但它其实什么都没验证。你怎么用一分钟发现这件事？

预设回答：

- 答"看代码里有没有 assert"——最快最直接。

- 答"故意把断言改成错的，看它红不红"——更工程化的做法，叫**变异测试**思路：把期望值改错，测试还是绿，就说明这条用例是空壳。

- 答"看覆盖率"——不够。空壳用例照样能把行覆盖率跑满，这正是 5.36 讲的"覆盖率骗人"。

易错点 / 考点：常考"JUnit 三个关键组件及各自作用"，注意 **Assert 判断的是实际与预期是否相符**、**Annotation 以 @ 引导**、**Runner 负责运行并显示结果**这三个标准表述。易错点是把 Assert 说成"捕获异常的工具"——捕获异常是运行器的事，Assert 只做判断。

过渡：三个组件里，注解是你写得最多、也最容易用错的一个。我们把它逐个过一遍。

【5.48】注释 Annotation（2.5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是注解清单，看着像背单词，其实有结构。我先给结构，再点必会的。

课件第一句：Annotation 清晰地表达测试程序的**逻辑结构和功能**。"逻辑结构"这四个字是关键——注解不只标记"这是个测试"，它还在描述这个测试在类里的位置和生命周期。按四组记：

用例本身：`@Test` 是基础；`@ParameterizedTest` 参数化，一条用例跑多组数据，是**减少重复代码最有效的手段**；还有 `@RepeatedTest`、`@TestFactory`、`@TestTemplate`。

生命周期：`@BeforeEach`／`@AfterEach` 每个方法前后各跑一次；`@BeforeAll`／`@AfterAll` 整个类只跑一次，默认必须是静态方法。这四个的差别是本节最常考的点——**"每个方法一次"还是"整个类一次"**。

组织与展示：`@DisplayName` 给用例起人话名字；`@Nested` 嵌套分组；`@Tag` 打标签筛选（CI 里排除 `slow`）；`@TestMethodOrder`／`@TestClassOrder` 控制顺序。

控制与扩展：`@Disabled` 禁用；`@Timeout` 超时即失败——很多人不知道，但它是**防测试挂死的关键**；`@ExtendWith`／`@RegisterExtension` 注册扩展；`@TempDir` 自动建临时目录并在测试后清理，测文件读写必备。

一个使用建议：**注解要"少而准"**。一个类里 `@Disabled` 一大堆、`@Timeout` 一个没有、`@DisplayName` 全不写，那份报告对别人就没有可读性。注解成本极低，收益全在"别人看得懂"和"以后改得动"。

板书：四组分类竖排：① 用例（@Test・@ParameterizedTest・@RepeatedTest・@TestFactory・@TestTemplate）② 生命周期（@BeforeEach・@AfterEach・@BeforeAll・@AfterAll）③ 组织展示（@DisplayName・@Nested・@Tag・@Order）④ 控制扩展（@Disabled・@Timeout・@ExtendWith・@TempDir）；右下角一行"每方法一次 vs 每类一次"。

提问：`@BeforeEach` 和 `@BeforeAll` 各适合放什么？如果一个方法里有 `@BeforeAll` 但方法不是 static，会发生什么？

预设回答：

- 答"`@BeforeEach` 放每个用例都要的初始化，`@BeforeAll` 放贵重的共享资源"——标准答案，比如数据

库连接池、容器启动放 `@BeforeAll`，测试数据构造放 `@BeforeEach`。

- 答"非 static 会报错"——对，JUnit 5 默认生命周期是"每个方法一个新实例"，静态方法才活得比实例长；除非加 `@TestInstance(Lifecycle.PER_CLASS)`。

- 追问：那为什么不干脆全用 `@BeforeAll` 图省事？因为**共享状态会带来用例间的隐式依赖**——第 3 条用例失败可能因为第 1 条改了数据，这种 bug 排查起来极度痛苦。

> **易错点 / 考点：**必背两组对照：JUnit 4 → 5 的注解映射（见 5.44）；以及 **`@BeforeEach`（每方法）与 `@BeforeAll`（每类）的区别**。另一个高频错点：`@ParameterizedTest` 要配合 `@ValueSource`／`@CsvSource`／`@MethodSource` 提供数据，**单独写 `@ParameterizedTest` 是跑不起来的**。

> **【思政落点 8】**：注解看起来只是几个 `@` 符号，但它体现的是**规范意识**。`@Test` 让框架找得到你的测试，`@DisplayName` 让同事看得懂你的意图，`@Timeout` 让一条挂死的用例不至于拖垮整条流水线——这些都不是"技术难度"，而是"愿不愿意按约定把事做完整"。工程里最贵的成本从来不是写不出高深算法，而是**别人接手你的代码时要猜**。请把"让别人少猜一次"当成写代码的默认标准，这就是职业素养的日常形态。

> **过渡：**注解管"怎么组织和跑"，断言管"算不算过"。下一页是断言的老家：Assert 类。

【5.49】Assert 类（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲断言，先把课件那两句核心讲透。
>
> 第一句：**检查并判断程序的运行结果是软件测试中的一项重要工作**。测试程序必然包含断言条件，运行时不满足，就表明结果与期望不一致，程序中存在缺陷。这句话的分量在于——**断言是缺陷判定的唯一出口**，你不写断言，框架就不知道什么叫错。
>
> 第二句是机制：JUnit 通过 Assert 类提供一系列断言方法判断运行结果，该类位于 `junit.framework` 包中；参数包括**期望变量 `expected`** 和**实际变量 `actual`**，两者相等则相符，否则用例失败。
>
> 注意版本口径：`junit.framework` 是 JUnit 3/4 的包路径，JUnit 5 对应 `org.junit.jupiter.api.Assertions`，方法名基本沿用。**答题按教材写 `junit.framework`，写代码用 `Assertions`**，两个都对。

> 实战最容易踩的坑：**`assertEquals(expected, actual)` 的参数顺序**。写反了不会报错——通过时结果一样，只有**失败时**你才会看到一句完全相反的报错："expected: 5, but was: 3"，而你脑子里想的是"应该是 3 啊"，排查方向彻底跑偏。给你一条硬规矩：**第一个参数永远是"我期望的"，第二个永远是"实际跑出来的"**。
>
> 再补三个常用断言：`assertTrue`／`assertFalse` 判真假；`assertNull`／`assertNotNull` 判空；`assertThrows` 判"这段代码应该抛某个异常"——它让异常路径也能被断言。JUnit 5 的 `assertAll` 能把多条断言打包，**一次报出全部失败项**。
>
> 最后一个观念：**断言要具体**。`assertNotNull(result)` 只说明有东西返回，`assertEquals(5, result)` 才说明返回对了。**覆盖靠执行，质量靠断言**。

> **板书：**写 `assertEquals(expected, actual)` 并用箭头标"① 期望 ② 实际"；下方列：`assertTrue/False`、`assertNull/NotNull`、`assertThrows`、`assertAll`；右侧两行红字"参数别写反""断言要具体：`assertNotNull` ＜ `assertEquals`"。

> **提问：**`assertEquals(3, result)` 和 `assertEquals(result, 3)` 都能编译通过，为什么后者是个必须避免的习惯？

> **预设回答：**

- 答"没什么区别，相等就行"——只在通过时没区别。失败时报告会反着说，让人误判。

- 答"失败信息会颠倒，排查方向错"——正确。这是最实际的危害。

- 答"JUnit 5 有重载能传 message，写清楚就行"——补得好，但 message 是辅助，**参数顺序是契约**，不能靠 message 兜底。

> **易错点 / 考点：**三个必考点：① `assertEquals(expected, actual)` 的**参数顺序**；② Assert 类的**包路径**（教材口径 `junit.framework`，JUnit 5 为 `org.junit.jupiter.api.Assertions`）；③ 断言方法"相等则通过、不等则失败"的判定逻辑。判断题常出："断言失败就说明被测程序一定有缺陷"——严格说还要排除"测试代码本身写错"的可能，所以这句话在考试里通常判错。

> **过渡：**断言在 JUnit 4 里不是一个孤立的类，它被 TestCase 继承了——这就把断言和用例绑在了一起。下一页看这三个类的关系。

【5.50】TestCase、TestSuite 和 TestResult 类（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把 5.45 讲的那三个角色落成三个具体的类，我逐条讲定义，并补一句"为什么这样设计"。

`TestCase` 类：JUnit 框架中为测试用例提供服务的核心类，位于 `junit.framework` 包中，并且**继承自 Assert 类**——所以它可以直接使用 Assert 里的断言方法。更关键的是：编写的测试类**都需直接或间接继承于 `TestCase`**，依靠它提供的办法来实现测试方法的运行与判断。

注意"继承自 Assert"的意义：**它把断言能力变成了测试类的天然属性**。写 `extends TestCase` 就能直接 `assertEquals(...)`，不用带类名。JUnit 3 就是这么用的——必须继承、方法名必须 `test` 开头。JUnit 4 之后改注解、不再强制继承，但**设计思路留了下来**：测试类＋断言库这个组合没变。

`TestSuite` 类：实现了 JUnit 的 `Test` 接口，用于管理每一个测试用例。关键词是"实现 `Test` 接口"——这就是**组合模式**：两者实现同一个接口，所以套件能装套件，你能一次跑一个方法、一个类、一个包、一个项目。

`TestResult` 类：收集并记录所有测试用例的运行结果，课件明确分三类：**成功、失败、错误**。实践建议：**看报告先看 Errors 再看 Failures**，因为 error 往往意味着这条用例根本没验到东西。

三个类连起来是一条链：`TestSuite` 组织 `TestCase`，`TestCase` 执行时把结论写进 `TestResult`，最后由 `TestRunner` 读 `TestResult` 报告。**组织—执行—记录—呈现**，四个动作四个对象。

最后提醒概念迁移：JUnit 5 里这三个类基本不直接用了，分别被注解、引擎发现机制、`TestExecutionSummary` 取代。但它们**在结构上仍然存在**，所以考题还在问。

> **板书：**三行三定义：`TestCase`＝测试用例核心类・继承 Assert・测试类需直接或间接继承它；`TestSuite`＝实现 Test 接口・管理用例・可嵌套；`TestResult`＝收集记录结果・成功／失败／错误；底部一条链"组织→执行→记录→呈现"。

提问：`TestCase` 继承 `Assert`，这个设计有什么好处，又有什么代价？

预设回答：

- 答"好处是随取随用，不用 import 断言类"——对，这是便利性。

- 答"代价是测试类被迫继承一个框架类，耦合了"——很到位。这就是 JUnit 4 改用注解、JUnit 5 改成静态导入

的原因：**测试类不应该被框架绑死继承链**。

- 追问：所以从 JUnit 3 到 JUnit 5 的演进主线是什么？——**从"继承框架"到"注解标记"，再到"组合扩展"**，一路在解耦。

> **易错点 / 考点：**必背三句定义（尤其 `TestCase` **继承自 Assert** 这一条，最容易被漏）；必考区分 **`TestResult` 收集 vs `TestRunner` 报告**；以及 `TestSuite` **实现了 Test 接口**这个细节，它是"套件能套套件"的技术前提。

> **过渡：**理论讲够了，看一段真实的 JUnit 5 测试脚本，把注解、断言、准备清理全用一遍。

【5.51】单元测试脚本（以 Junit 5 为例）（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是一段完整的 JUnit 5 测试脚本，我按结构拆，你以后照这个骨架填。

第一层是**类**：通常跟被测类同名加 `Test` 后缀，类上可以用 `@DisplayName` 写人话标题。

第二层是**准备**：`@BeforeEach` 里造数据、建对象。原则是——**每条用例都从同一个干净状态出发**。如果你发现某条用例必须接着上一条的数据才能跑，那不是测试，是连续剧，早晚会崩。

第三层是用例本体，每个 `@Test` 方法内部一定是三段式，业内叫 **AAA：Arrange（准备）— Act（执行）— Assert（断言）**。读代码就找这三段：前面 new 对象设参数是 Arrange，中间那行调用被测方法是 Act，后面 `assertEquals` 是 Assert。

为什么死守 AAA？它是**可读性的骨架**。一条测试挂了，你要在十秒内判断"是数据错了、调用错了、还是结果真错了"，三段分明一眼定位；三段混在一起就得逐行读。

第四层是**清理**：`@AfterEach` 释放资源、删临时文件。更省事的是 `@TempDir`，框架自动建目录并在测试后清掉。

两个实践细节：① **一个测试方法只断言一件事的主题**，围绕同一个行为；一条用例既测加法又测减法，挂了你说不清哪个坏。② **方法名就是文档**：`should_return_zero_when_second_arg_is_zero` 比 `testAdd2` 强一百倍，配 `@DisplayName` 中文描述，报告直接能当验收证据。

一句总结：**注解管结构，AAA 管顺序，断言管判定，命名管可读**。

> **板书：**抄骨架四层：类（`@DisplayName`）→`@BeforeEach`（造数据）→`@Test`（Arrange・Act・Assert）→`@AfterEach`／`@TempDir`（清理）；右侧一行"一个方法一个主题；方法名即文档"。

> **提问：**一条 `@Test` 方法里，Arrange 写了 30 行、Act 1 行、Assert 1 行。这条测试有什么问题？

> **预设回答：**

- 答"太长了，不好读"——表象。真正的问题是：**Arrange 太长说明被测对象的依赖太多**，回到了 5.35 那条"引导更好的设计"。

- 答"可能该抽成 @BeforeEach 或工厂方法"——对，重复的构造逻辑应该提取。

- 答"没问题，只要断言对就行"——短期能跑，长期没人敢改：改一处构造就要动三十行。

> **易错点 / 考点：**实验和卷面都常考 **AAA 三段式**，答题时要说清每段职责（准备数据／调用被测方法／断言结果）。易错点：把 `@BeforeEach` 当成"每个类执行一次"（那是 `@BeforeAll`）；以及忘记 JUnit 5 里 `@BeforeAll` 标注的方法默认必须是 **static**。

> **过渡：**单个类测起来还算舒服，可一旦被测类要调用别人的服务，麻烦就来了——这就进入 Mock 的世界。

【5.52】Mock 技术（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是分隔页，任务是给一个总印象：**Mock 技术解决的是"依赖"这个麻烦**。
>
> 先把问题摆清楚。一个典型的业务方法：查数据库拿订单、调风控判断能不能放行、调支付网关扣款、发消息通知仓库。你要测它，四个下游全在别人手里——数据库没你要的数据、风控在灰度、支付网关收费、消息队列没起。三条路：**搭全套环境**（贵、慢、脆）、**不测**（欠债）、**用替身**（Mock 技术）。
>
> Mock 技术就是第三条，把下游换成可控的替身对象。前面手写桩是最原始的做法；接下来要讲的是一整套替身的分类学：Fake、Dummy、Double、Spy、Stub、Mock，六个词是本节核心，也最容易混。
>
> 先给一句话地图，听后面几页不迷路：**Dummy 是凑数的、Stub 是给答案的、Spy 是记账的、Fake 是能真跑但简化的、Mock 是既给答案又验交互的**。`Test Double` 是这五个的总称——就像"车辆"是轿车卡车摩托的总称。
>
> 为什么值得花一整节？因为它决定你写测试的**成本和质量**。用一个 Fake 能让几百条用例又快又真；而滥用

> Mock 会把测试变成"验证我自己的调用代码"，一重构就全红，业内叫**过度 Mock 的脆性测试**。所以目标不只是会分类，而是**会选**——这是本章的一个难点。

> **板书：**中间写"Test Double（测试替身）"；五个分支：Dummy（凑数）／Stub（给答案）／Spy（记账）／Fake（简化但能跑）／Mock（给答案＋验交互）；右侧写三条路："搭环境（贵）／不测（欠债）／替身（对）"。

> **提问：**如果被测方法要调一个第三方短信接口，你会搭一个真的测试环境，还是用替身？说说你的取舍依据。

> **预设回答：**

- 答"用替身，短信发不出去还会花钱"——对，这是"有副作用"那一类，必须隔离。

- 答"搭真环境更真实"——真实但代价高，而且**不可重复**（运营商限流、延迟抖动），会让测试时红时绿。

- 追问：那"真实性"怎么补？答案是分层——单元测试用替身保证逻辑正确，集成测试再用真环境验证"接线对不对"。**不同层测不同的事，别让单元测试承担集成测试的责任**。

> **易错点 / 考点：**本节最容易考的是**六个替身词的定义与区别**，尤其 Stub 与 Mock。判断口诀先给你：**看它"有没有被验证交互"**——只提供返回值、从不检查"被调用了几次"的，是 Stub；会 `verify` 调用次数／顺序／参数的，是 Mock。

> **过渡：**分类从最弱的一个开始。下一页把 Fake、Dummy、Double、Spy 四个词一次讲清。

【5.53】Fake、Dummy、Double 等（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是定义页，四个词一个一个来，每个给你一条"什么时候用它"的判据。
>
> **Fake，伪对象**——伪造、假货，反义词是 real。课件描述很准：**几乎和产品代码做同样的工作（伪装成跟真的一样），但为了满足测试要求，采用更简单的实现**。关键词是"更简单的实现"，不是"返回假数据"。典型例子：内存数据库代替真库、内存队列代替 Kafka。**它不是骗子，它是精简版**——行为对，性能和省事优先。
>
> **Dummy，哑对象**——仿制品、傀儡。课件说：对被测对象的模拟，特别是**缺乏原有特性或功能而实现欺骗性地替换**，括号里补了句特别生动的话：**完全听命行事，不在具体测试里面起任何作用**。这就是 Dummy 的全部精髓——它只为"把参数位填上"。比如方法必须传一个 `Logger` 才能编译，你传一个什么都不做的空实现，

那个 Logger 就是 Dummy。

Double，测试替身——课件定义：有相同的行为和响应。这是**总称**，就是"替身演员"：长得像、能站位、能演戏，但不是本人。Fake、Dummy、Stub、Spy、Mock 都是它的子类。

Spy，测试间谍——课件说：能够接收参数并返回值。这句太短，我补全：Spy 是**会记账的替身**——既像 Stub 一样返回你设定的值，又把"谁调了我、调了几次、传了什么参数"记下来供你核对。它天然适合验证副作用，比如"提交订单后审计日志接口应被调用一次，参数里必须带用户 ID"。

最后一句归类，必考：**Dummy Object、Test Spy 可归为 Stub**。因为在强弱维度上它们都还属于"提供输入／记录输出"这一档，**都不强制你验证交互**；Mock 不一样，它天生就让你去 `verify`。

日常选用顺序，从便宜到贵：**能不用就不用 → 需要返回值用 Stub → 要看有没有被调用用 Spy → 需要"没按约定交互就失败"才用 Mock**。

板书：四行定义：`Fake`＝精简版真实现（内存数据库）；`Dummy`＝凑数，不参与任何逻辑（空 Logger）；`Double`＝替身总称；`Spy`＝能给值＋会记账；底部红字"**Dummy Object、Test Spy 可归为 Stub**"；右侧一列"由便宜到贵：不用→Stub→Spy→Mock"。

提问：一个方法签名要求传 `UserRepository`，但这条用例根本不查库，你传什么？这算 Fake 还是 Dummy?

预设回答：

- 答"传个空实现，什么都不返回"——正确做法，这是 Dummy。

- 答"这是 Fake"——错了。Fake 是"能真跑的精简实现"，空实现不跑任何逻辑，是 Dummy。**判据：它内部有没有"替代性的真实计算"**。

- 追问：如果这个空实现的某个方法被误调用了、返回 null 导致下游空指针，你怎么办？——让 Dummy 在"不该被调用"的方法里直接抛异常，这条用例就变成了"验证它没被调用"，非常好用。

易错点 / 考点：这一页是纯概念页，考点就是**四词定义 + "Dummy／Spy 可归为 Stub"**。三个高频错答：① 把 Fake 说成"假数据"（Fake 是假**对象**，是简化实现）；② 把 Dummy 说成"返回默认值的桩"（它不返回有意义的值，它根本不该被调用）；③ 把 Double 当成某个具体替身（它是总称）。记忆口诀：**Fake 像真的、Dummy 是摆设、Double 是统称、Spy 会记账**。

过渡：四个词里最容易被误解的是 Fake——很多人以为 Fake 就是"返回假数据"，其实它可能是你整个测试

> 策略里最有价值的一件东西。下一页看它到底怎么用。

【5.54】伪对象（2.5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页讲 Fake 的适用场景。课件第一句就把定位说清楚了：**在某些情况下，插桩不够用。被桩对象替换掉的行为是被测对象所需要的。在这种情况下，伪对象（fake）可能是一种合理的折衷。**
>
> 这句话要慢慢嚼。什么叫"被桩替换掉的行为是被测对象所需要的"？意思是：被测代码不只是拿一个返回值就完事，它需要下游**真的做出某种行为**。最常见的例子——测"下单"流程，它调仓储**保存**订单，然后**查出来**生成发票。你用 Stub 让 `save()` 返回 true，那 `find()` 呢？数据根本没存进去，你只能再写一个桩返回一个假订单——**结果你测的是"桩的剧本"，不是"仓储的行为"**。
>
> 这时候 Fake 就对了：内存仓储，`save()` 真存进一个 Map，`find()` 真从 Map 里取。行为是真的，只是没落磁盘。这就是"折衷"两个字的含义——**比桩真，比真库轻**。
>
> 课件给的例子是这段 `MakePurchase`：先 `CreatePurchase(customer)`，再 `AddProduct(purchase, product)`，再 `CreateInvoice(purchase)`，有折扣就 `ApplyDiscount(discount)`。它的特征是**一串有状态的协作调用**，后一步依赖前一步的结果。这种代码用 Stub 硬顶，你要为每一步编一个"看起来像真的"返回值，编到第三步你自己都不信。
>
> 三条判断标准，什么时候从 Stub 升级到 Fake：① 被测代码有**写后读**依赖（存了再查）；② 依赖下游的**状态机**（未支付→已支付→已发货，桩很难维护）；③ 同一批测试要**反复复用**同一个协作者。
>
> 反面也要说清：**Fake 的成本是自己实现一套简化逻辑**，而且这套逻辑本身可能有 bug。所以只用在价值高、复用多的协作者上；一次性场景，老实写 Stub。

> **板书：**左侧"Stub 不够用的信号"：写后读／依赖状态机／多用例复用；右侧"Fake ＝ 行为真、实现简"，例：内存仓储（Map 代替 DB）；中间一句课件原话"被桩替换掉的行为是被测对象所需要的"；底部"Fake 的成本：自己实现＋自己可能有 bug"。

> **提问：**用内存数据库（Fake）代替真库，会不会掩盖"SQL 写错了"这类缺陷？如果会，你怎么补？

> **预设回答：**

- 答"会，内存库和真库的 SQL 方言不一样"——非常关键的一点。HSQLDB 能跑的 SQL 到了 MySQL 未必能跑。

- 答"所以还要有集成测试跑真库"——正解。**单元测试用 Fake 保逻辑，集成测试用真库保 SQL**，这就是分层。

- 追问：那 Fake 到底值不值？值——因为它把 90% 的用例从"几百毫秒"压到"几毫秒"，让开发者愿意频繁跑；剩下 10% 交给集成测试兜底。

> **易错点 / 考点：**常考"Fake 与 Stub 的适用差别"——标准答案：**Stub 只提供预设返回值；Fake 提供简化但真实的行为实现**，适用于被测对象依赖协作者"行为"而非仅"返回值"的场合。易错点是把 Fake 说成"造假的测试数据"，那是 fixture（测试夹具），不是 Fake。

> **过渡：**光说概念还是飘，下一页那张示意图把"SUT、DOC、前门、往返测试"这四个词摆到了一起。

【5.55】Fake 示意图（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这张图是 Fake 的结构图，要你看出四件事：被测系统在哪、被依赖组件在哪、从哪个门进去、什么叫往返测试。

两个缩写先认：**SUT 是 System Under Test，被测系统**；**DOC 是 Depend-on Component，被测系统所依赖的组件**。单元测试的基本格局就是"SUT 一个、DOC 若干"，桩、假、Mock 全是围着 DOC 打转。

第一个要点是**前门（front door）**，图上标着 Public interface。意思是：**测试应该从被测对象的公开接口进去**，不要绕过接口去动内部字段。从公开接口进去，测的才是真实使用方式；一旦用反射改私有字段，测试就跟实现细节绑死，实现一改测试就红——那种测试是负担，不是资产。

第二个要点是 **Fake 替换的位置**：它站在 DOC 那个位置上，接口一样、行为简化。注意它替换的是"整个协作者"，不是"某一次调用"——这是它和 Stub 在结构上的差别：**Stub 管"某次调用返回什么"，Fake 管"这个协作者整体上是什么"**。

第三个要点是 **round-trip test，往返测试**：一次测试里既写进去又读出来，验证数据在往返中没丢没错。典型场景：保存订单→重新查询→断言字段一致。这种测试用 Fake 最自然，因为 Fake 真有存储；用 Stub 做往返，等于自己跟自己对台词。

第四个要点藏在箭头里：**SUT 只依赖接口，不依赖实现**。这是整个技术能成立的前提——如果被测代码直接

`new MySQLConnection()`，你连替换的机会都没有。所以写不下去时先回头看这条：**不是测试方法的问题，是依赖注入没做好**。

板书：画图：左"前门 Public interface / front door"→ 中"SUT 被测系统"→ 右"DOC 依赖组件"，DOC 位置上标"Fake 站在这里"；下方写 "round-trip test：写进去→读出来→比对"；底部一行"SUT 只依赖接口，才换得动"。

提问：为什么强调"从公开接口进入"？如果为了测试方便，把某个私有方法改成 public 再测，有什么问题？

预设回答：

- 答"没什么问题，能测就行"——短期方便，长期是**测试与实现耦合**，重构时测试必炸。

- 答"会把实现细节暴露成契约"——很准。public 一开，别人就会依赖它，你就再也改不动了。

- 追问：那私有方法里的复杂逻辑怎么测？——通过公开接口覆盖它的所有分支；如果实在测不到，说明它应该被提取成一个独立的类，然后单独测。**又是"测试逼设计"**。

易错点 / 考点：这一页的考点是**缩写与概念**：SUT、DOC、front door、round-trip test 四个词要能对上中文。易错点是把 DOC 说成"被测对象"（DOC 是**被依赖**的组件）；以及把 round-trip 理解成"跑两遍"，它的意思是"写—读—比对"一个来回。

过渡：示意图之后是两个具体的 Fake 例子，第一个看结构，第二个看它怎么顶替数据库。

【5.56】Fake 示例 1（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这个示例是 Fake 的一个具体落地，我们从"为什么这么写"看它，而不是逐行念代码。

场景：被测代码需要一个仓储协作对象，负责存和取。真实实现连数据库，测试里不想连。于是写一个`FakeRepository`，**内部用 Map 或 List 存数据**，对外实现同一套接口——`save` 往 Map 里放，`findById` 从 Map 里取，`count` 返回 Map 大小。

三个设计要点决定它写得好不好。

第一，接口必须和真实实现一致。 最稳的做法是**让 Fake 和真实现实现同一个接口**，这样替换谁都不用改被测代码，编译期就能保证签名对得上。

第二，行为要"够真"。 什么叫够真？`save` 之后 `findById` 必须查得到（这就是往返）；同一个 ID 重复 `save` 应该覆盖而不是新增；`count` 要跟着变。这些不是额外功能，它们恰恰是**被测代码依赖的语义**。少实现一条，测试就给出错误的绿灯。

第三，不要实现真实现不需要的东西。 事务、缓存、分页优化——真实现有，但被测逻辑用不到，写了只是给自己埋 bug。**Fake 的目标是"语义够用"，不是"功能齐全"**。

还有一个工程习惯：**Fake 是生产资产，不是测试脚本里的临时类**。它应该放在 `src/test/java` 下独立的 `fakes` 包里，有名字、有注释，可能还有自己的单元测试。因为它会被几十上百条用例复用，一旦它有 bug，全项目的测试都在骗你。**给测试代码和产品代码同等的待遇**，这是成熟团队和初级团队最明显的差别之一。

板书： 三要点：① 与真实现同接口（编译期保签名）② 语义够真（存后能取・同 ID 覆盖・count 同步）③ 不多实现用不到的功能；右侧一句"Fake 放 `src/test/.../fakes`，是资产不是草稿"。

提问： 如果 Fake 的 `save` 忘了做"同 ID 覆盖"，而是在 List 里又加了一条，哪类测试会假绿？

预设回答：

- 答"测新增的用例"——不一定。真正假绿的是**测"更新"和"数量"的用例**：更新后 `count` 应该还是 1，但 Fake 返回 2，而你写的断言如果是"能查到"就发现不了。

- 答"所有用例都会错"——太绝对，但方向对：**依赖语义的用例都会给出错误信号**。

- 追问：所以 Fake 要不要自己也写测试？要。**给 Fake 写测试**，是本课程希望你建立的一个反直觉习惯。

易错点 / 考点： 这一页不考代码细节，考的是"Fake 与真实现的关系"：**同接口、语义一致、实现更简单**。易错点：学生常把 Fake 写成"另一个完全不同的类，只是名字像"——那样替换时就得改被测代码，等于没做到"可替换"。

过渡： 第一个例子讲"怎么造一个 Fake 仓储"，第二个例子更进一步：直接用现成的内存数据库当 Fake。

【5.57】Fake 示例 2（2.5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这个例子给了一句非常凝练的定义：**fake object = enhanced stub**——伪对象就是"增强版的桩"。为什么说

增强？因为它把"返回一个值"升级成了"表现出一种行为"，但目的仍然是测试，不是生产。

场景是：**验证 DAO（Data Access Object）能正常工作，就构造 fake object 代替数据库**。关键词是"验证 DAO 本身"。前面讲的 Fake 都是当配角、帮别人被测；这一页反过来——**Fake 当背景，让 DAO 成为被测对象**。DAO 的职责就是"把对象翻译成 SQL 再翻译回来"，当然要跟真数据库打交道才能验，可每次连真库太慢、还要清数据。

课件给了两个具体东西：**HSQLDB 是流行的 in-memory database**（内存数据库），**DBUnit** 是配套的数据准备框架。

这两个的组合就是经典的"DAO 单元测试方案"：用 HSQLDB 起一个内存库（进程内、毫秒级、随测试生死），用 DBUnit 在每条用例前把数据灌成一份 XML／CSV 定义的初始状态，测完自动重建。**这样你测的是真 SQL、真 JDBC、真类型映射，只是数据库是"内存版的"**。

这里有个分层观念要说清：**这已经不算纯单元测试了**，业内叫"集成度更高的单元测试"或"轻量集成测试"。它比纯单元测试真（SQL 真跑），比全集成快（不连网络、不起容器），是很好的折衷点，尤其适合 DAO／Repository 这一层——这一层如果全用 Mock 把 JDBC 替掉，你测的只是"我调了 jdbcTemplate.update"，SQL 对不对一点没验。

最后提醒一个坑：**内存库和真库的 SQL 方言不完全一样**。HSQLDB 有 MySQL 兼容模式，但函数、索引、锁行为还有差异。所以这类测试**不能替代**真库上的集成测试，上线前那一轮还得在真库跑。

> **板书：**顶部写课件原话"fake object = enhanced stub"；中部写方案：HSQLDB（内存库）＋ DBUnit（数据准备）；下方三段定位："纯单元（全替身）→ **DAO 用内存库（本页）** → 全集成（真库真网）"；右下角红字"方言有差异，替代不了真库集成测试"。

> **提问：**测一个 DAO，用 Mockito 把 `JdbcTemplate` mock 掉，和用 HSQLDB 真跑，两种做法各能发现什么、各漏什么？

> **预设回答：**

- 答"Mock 能验调用参数，内存库能验 SQL 结果"——准确。Mock 验"我传了正确的 SQL 和参数"，内存库验"这段 SQL 真的返回了正确的行"。

- 答"Mock 更快、内存库更真"——对，这就是取舍。

- 追问：那到底用哪个？——**看你要防什么缺陷**。防"参数传错"用 Mock；防"SQL 写错、映射错"必须真跑。

DAO 这一层的缺陷 80% 在 SQL 和映射上，所以内存库更值。

> **易错点 / 考点：**常考"Fake 的典型实现方式"——答案是内存数据库、内存队列、内存文件系统这类"真实行为的简化实现"。易错点：把 DBUnit 说成"数据库"（它是**测试数据准备**工具），把 HSQLDB 说成"Mock 框架"（它是**真的数据库**，只是驻留内存）。

> **过渡：**到这里，替身家族里"提供状态"的那几个都讲完了。接下来是最容易混、也最重要的一对：Mock 和 Stub。

【5.58】模拟对象（2.5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页是本节的思想转折点，课件用了很重的措辞：**模拟对象（Mock）在某种程度上是游戏的改变者，因为它们将测试的焦点从"状态"转换到"行为"。**
>
> 先说**基于状态的测试**。课件描述：像 fake 会聚焦于状态的测试，以断言结束，该断言会核对返回值，或以某种方式查询被测对象的状态（属性）。翻成人话：**我关心结果对不对**——算出来是 5 就是 5，库存从 100 变成 99 就是 99。这是绝大多数人写测试的默认方式。
>
> 再说**基于行为的测试**。课件说它完全不同，能够验证模拟对象和被测代码、或另一协作者之间**特定的交互行为**，会关注被测对象**是否被调用、调用了多少次、是否查询了 Value 属性**。翻成人话：**我关心你有没有按约定去打招呼**。
>
> 为什么需要第二种？因为有一大类逻辑，结果根本不体现在返回值上。比如用户下单成功后要给风控系统发一条"交易报备"。这个方法返回的只是一个订单对象，你从返回值上看不出报备发没发。状态验证无从下手；行为验证一句话：`verify(riskClient).report(any())`——**报备必须发生一次**。这就是 Mock 的战场。
>
> 课件那段代码就是这个套路：构造一个 `LenientMock`，注入 `PurchaseWorkflow`，走一遍"加商品→用已有客户→输折扣码"，最后 `campaignMock.verify()`。注意这里没有 assertEquals，它的判决完全来自"交互有没有发生"。
>
> 最后给一个平衡：**行为验证更强，也更脆**。它把"怎么调"写进了测试，一旦 `report()` 改名成 `submitRisk()`，或一次调用拆成两次，测试就红——即使业务行为完全正确。所以业内经验是：**优先状态验证，只在结果不可观测时才用行为验证**。

> **板书：**左右对照：左"状态验证（state）：断言返回值／对象属性——问『结果对不对』"；右"行为验证（behavior）：verify 调用次数・参数・顺序——问『有没有按约定交互』"；中间一行"结果不可观测时，才用行为验证"；底部写 `campaignMock.verify()` 圈出来标注"这就是断言"。

提问：一个方法"发送一封欢迎邮件"，返回值是 void。你要怎么验证它真的发了？用状态验证还是行为验证？

预设回答：

- 答"返回值是 void，没法验"——这是没思路的表现。void 恰恰是行为验证的典型场景。

- 答"用行为验证，verify 邮件服务被调用一次，且收件人是新用户"——标准答案，而且要注意"参数也要验"，否则发错人也算通过。

- 答"我改成让它返回发送结果，然后断言"——可以，但要问一句：**为了可测改产品接口，值不值？**如果这个返回值业务上本来不需要，就别为测试污染设计；如果它本来就有价值（比如用于重试），那改得对。

> **易错点 / 考点：**必考"状态验证 vs 行为验证"的定义与适用场景。标准表述：**状态验证检查被测对象最终的状态或返回值；行为验证检查被测对象与协作者之间的交互（是否调用、次数、顺序、参数）**。易错点：把"行为验证"说成"测试行为"（两个"行为"不是一回事，前者指交互，后者指功能）。

过渡：概念讲完，两页纸把 Mock 和 Stub 摆在一起对比——这是本节的必考页。

【5.59】Mock vs. Stub（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是一张对比图，把 Mock 和 Stub 摆在一起。要记住的不是两个名字，而是**三条分界线**。
>
> **第一条，目的不同。** Stub 的目的是让测试跑起来——被测代码要一个值，它给一个值，任务完成。Mock 的目的是让测试做判断——它存在的意义就是被你 `verify`。所以有句话很精辟：**Stub 是被动应招，Mock 是主动待命**。
>
> **第二条，验证什么。** Stub 讲究状态——你给什么输入、它返回什么输出，断言落在被测对象的返回值上。Mock 讲究行为——断言落在"这个协作者被怎么对待了"：调了几次、参数是什么、顺序对不对。同一段代码，用 Stub 你只能问"结果对不对"，用 Mock 才能问"过程守不守规矩"。
>
> **第三条，严格程度。** Mock 默认严格：你声明了期望，它没被满足就失败——即使业务结果完全正确。这就

是"过度规范"（over-specification）的风险。Stub 很宽松：它不管你调没调，反正它在那儿。

再补一条机制上的差别，读代码时很有用：**Mock 通常是动态生成的代理对象**（框架运行时造一个实现接口的类），**Stub 通常是你手写的类**。看到 `when(...).thenReturn(...)`，那是框架造的 Mock；看到 `class FakeXxx implements Xxx`，那是手写 Stub。

给你一个判断题模板，考试直接用：**这条测试挂掉，是因为结果错了，还是因为调用方式变了？** 前者说明该用 Stub；后者说明你用了 Mock，它在向你报警。**如果这个变更业务上完全无害，那这条 Mock 测试就是负资产**——它没保护质量，只保护了实现。

板书：三行对照表：目的（跑起来 vs 做判断）｜验证（状态 vs 行为）｜严格度（宽松 vs 严格可 over-spec）｜机制（手写类 vs 动态代理）；底部口诀"Stub 被动应招，Mock 主动待命"；右下角写判断句"结果错了用 Stub，调用变了才有 Mock 的份"。

提问：重构时把一次 `save()` 调用拆成 `saveDraft()` 和 `confirm()` 两步，业务结果完全一样。用 Stub 写的测试和用 Mock 写的测试，分别会怎样？

预设回答：

- 答"Stub 的测试还是绿的，Mock 的红了"——正是这道题的标准答案，它把两条分界线一次说明白了。

- 答"两个都会红"——不对，Stub 不关心你调了几次。

- 追问：那说明什么？——说明 **Mock 测试的成本要算进重构成本里**。所以你写 Mock 的时候要克制：只对"业务上必须发生的交互"做 verify，别对"实现细节"做 verify。

易错点 / 考点：本节最核心的考点，几乎必出。答题模板：**Stub 提供预设返回值、用于状态验证；Mock 验证交互行为（调用次数／参数／顺序）、用于行为验证；Mock 更严格，滥用会导致测试脆弱**。易错点：说"Mock 是更高级的 Stub"——不是"更高级"，是"关注点不同"。

过渡：这一页是两张图，下一页是课件给的一段总结，它给出了这件事最实用的那句话。

【5.60】Stub 和 mock 小结（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本节的收口，四句话，一句比一句重要，最后一句是本章的名言。

第一句：**Stub 最弱，只是参与测试，为了让测试跑起来。** 它提供输入，不参与判断。这不是贬义——**弱有弱的好处，弱就是稳定**，它不会因为你改了实现就炸。

第二句：**Dummy／Spy 强一些。Test Spy 参与测试，还要验证产生的某种结果。Dummy 是简单、原始的测试替身，比如 Dummy server，给它不同输出会有不同响应，但内部没有实质计算。** 注意"内部没有实质计算"——这是 Dummy 和 Fake 的分界线。

第三句：**Mock 更强，要求"如何参与"，且比较灵活，视情况而定，其行为可能像 Dummy、Stub 或 Spy。** 这句信息量很大：Mock 是万能替身——通过配置，今天可以是 Dummy、明天是 Stub、后天是 Spy。所以**从能力上说，Mock 包含其它**。

第四句，最重要：**够用就行，如果 Stub 够用，就不要用 Mock。**

为什么？因为**替身的强度不是免费的**。用 Mock 就要写 `verify`，就要约定调用次数和参数——这些全是**实现细节**。实现一变测试就红，红了团队就不信任测试，不信任就开始 `@Disabled`，`@Disabled` 一多，测试就名存实亡。**这是测试体系崩塌的典型路径，起点就是"随手多写了一个 verify"**。

反过来，Stub 只承诺"我给这个值"，不管你怎么用，所以它对重构**免疫**。同一场景能用 Stub 表达清楚，就别升级成 Mock。

那什么时候必须用 Mock？三个条件满足其一：① **结果不可观测**（void 方法、发消息、写日志）；② **交互本身是业务要求**（"必须且只能报备一次"，重复报备是事故）；③ **要验证顺序或次数**（先扣库存再扣款，顺序错了就是资损）。这三条之外，都是 Stub 的活。

再深一层：这句话是一种**工程价值观**——用最简单的手段达成目的，不为炫技买单。技术选型不是"谁强用谁"，而是"谁刚好够用用谁"。

板书：强度阶梯：`Stub`（最弱・给值）→ `Dummy／Spy`（参与・记账）→ `Mock`（最强・可扮任意角色）；中间画一条横线写课件名言"**够用就行，Stub 够用就不要用 Mock**"；右下角三条件："结果不可观测／交互即业务要求／需验顺序次数 → 才上 Mock"。

提问：团队里有个同事特别喜欢用 Mock，每个测试都写三条 `verify`。你会怎么说服他改？请给一个基于事实的论证。

预设回答：

- 答"Mock 性能差"——不成立，性能差别可以忽略。

- 答"下次重构你就知道了"——情绪化，不是论证。

- 答"我用一次重构实验证明"——最佳答案：拿一条用例，把实现里的方法调两次改成一次（业务等价），跑测试，红了；再换成 Stub 版本，绿了。**用证据说话，比讲道理有效十倍**。

- 收口：这正好引出"技术选型要能被验证"这个习惯——你说 Mock 好或不好，都要有实验。

> **易错点 / 考点：**这一页几乎就是标准答案原文，四句话要能背：**Stub 最弱、Dummy／Spy 次之、Mock 最强且灵活、够用就行**。判断题常出"Mock 比 Stub 好，应该优先使用"——错，标准答案就是课件那句"如果 Stub 够用，就不要用 Mock"。另一个考点：Dummy 的"内部没有实质计算"。

> **【思政落点 9】**：这句"够用就行"是难得的**务实精神**教科书。技术圈有个通病——**用什么工具代表水平**：写测试必须上 Mock，做架构必须上微服务，选数据库必须分布式。结果系统越来越复杂，问题越来越多。课件这句话给的立场正好相反：**以问题为中心，不以工具为中心**。请把这条带到你们以后的工作里：先问"我要解决什么"，再问"最轻的手段是什么"。**克制，是一种更高级的专业能力**。

> **过渡：**到这里"该用哪种替身"想清楚了，剩下的问题是"用手写还是用框架"。下一页进入 Mock 框架。

【5.61】Mock 框架（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页给了四个名字：**JMock、EasyMock、Mockito、Moco**，括号里还注了"Java API、独立服务器"。它们不是同一类东西。
>
> 前三个是 **Java 的 Mock 框架**，解决"在测试代码里动态造替身对象"，能力边界是**同一个 JVM 进程内**：给接口生成代理、控制返回值、记录调用。
>
> 第四个 **Moco** 不一样，课件标了"独立服务器"——它是**可以独立启动的 HTTP 服务桩**，用来模拟外部 HTTP 接口。给它一份配置，它就监听端口，按匹配规则返回响应。所以 Moco 解决的是**跨进程**的替身。
>
> 这个区分考试和实战都常混：**同进程内的协作者用 Mockito；跨进程的 HTTP 依赖用 Moco（或 WireMock、MockServer）**。判据一句话：**能不能在测试方法里 `new` 出来**。能，用 Mockito；不能（要走网络），用服务桩。
>
> 再看三个 Java 框架的代际关系，帮你记顺序：**JMock → EasyMock → Mockito**。演进主线是**越来越轻**：

> JMock 用一套约束对象（`oneOf`、`allowing`）表达期望，读着像英语但写起来啰嗦；EasyMock 用"录制—回放—检查"三步，必须先声明期望再回放；Mockito 砍成 `when/verify` 两步，还允许先调用再验证。**这就是它胜出的原因：更符合人写测试的思路**。
>
> 实践建议：**除非维护老项目，新项目直接上 Mockito**。EasyMock／JMock 的价值主要是读懂存量代码，以及理解"为什么 Mockito 的设计更好"。

> **板书：**左边一列"进程内 Mock 框架"：JMock → EasyMock → Mockito（演进主线：越来越轻）；右边单列"Moco＝独立 HTTP 服务桩（跨进程）"；中间一行判据"能在测试里 new 出来 → Mockito；要走网络 → Moco"。

> **提问：**被测代码要调一个第三方支付接口（HTTP），你该用 Mockito 还是 Moco？为什么 Mockito 在这里不管用？

> **预设回答：**

- 答"用 Moco，因为要发 HTTP 请求"——正确，跨进程就得服务桩。

- 答"Mockito 也能 mock，mock 掉那个 HTTP 客户端类就行"——这条**部分成立**，而且很常见：把客户端封装成一个接口再 mock 它。但代价是你测不到"序列化、URL、超时、重试"这些真正容易出错的地方。

- 追问：所以正确姿势是两层都测——**逻辑层 mock 客户端，契约层用 Moco 验真实 HTTP**。分层，永远分层。

> **易错点 / 考点：**常考"Mock 框架有哪些"，四个名字要能列全；同时要能说出 **Moco 与前三者的定位差别（独立服务器／HTTP 服务桩）**。易错点：把 Moco 说成"Java Mock 框架"（它也支持 Java API，但核心是独立服务器）；把 JMockit 和 JMock 混成一个（JMockit 能力更强、耦合 JDK 更紧，见 5.43）。

> **过渡：**先看最早被广泛使用的那一个，EasyMock，因为它的"三步法"是理解所有 Mock 框架的钥匙。

【5.62】EasyMock（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> EasyMock 的定义课件写得很准：**通过简单的方法对于给定的接口生成 Mock 对象的类库。它提供对接口的模拟，能够通过录制、回放、检查来完成测试过程，可验证方法的调用种类、次数、顺序，可令 Mock 对象返回指定的值或抛出指定异常。**

三个关键词必须抓住：**录制（record）、回放（replay）、检查（verify）**，这是它的心脏。

用录像机打比方：**录制**是先跟替身对台词——"待会儿有人问你 `execute()`，你就返回这个 ResultSet"；**回放**是把替身放进被测代码里跑，它按剧本应答；**检查**是跑完看剧本有没有演完——该调三次的调了两次，报错。

课件那段代码就是这个骨架：先 `EasyMock.createControl()` 建控制中心，再用 `control.createMock()` 造出 `Connection`、`Statement`、`ResultSet` 三个替身——JDBC 三层对象，正好是"层层依赖"的典型。

为什么这个设计是"必然的"？因为 **Mock 对象的行为必须在被测代码运行之前就定义好**——它一旦开始跑就在被动应答，没有机会改剧本。所以"先录后放"是 Mock 的**逻辑必然**，EasyMock 只是把它显式化了。

那 Mockito 为什么能"先调用后验证"？因为它把"期望"和"验证"拆开了：`when()` 定义应答（相当于录制），`verify()` 是**事后**检查，而且不预先声明就必须满足——你没 verify 的调用它不管，你 verify 了的它才查。**这是宽松模式对严格模式的胜利**。

留一个知识点：EasyMock 当年只能模拟**接口**，Mockito 能模拟类和 final 类，JMockit 连静态方法、构造器都能模拟。**"能模拟什么"这条谱系，是理解三个框架差别的坐标轴**。

板书：三步法大字：**录制（record）→ 回放（replay）→ 检查（verify）**；下方抄骨架：`createControl()` → `createMock(Connection/Statement/ResultSet)`；右侧一行"EasyMock 严格：先声明后回放；Mockito 宽松：when 定义＋事后 verify"。

提问：为什么 Mock 对象的行为必须在被测代码运行"之前"就定义好？如果不能，会发生什么？

预设回答：

- 答"因为运行时它已经在应答了，改不了"——到位。顺序是结构决定的，不是习惯问题。

- 答"其实可以边跑边改"——技术上部分可行（有些框架支持 Answer 回调里改），但那样测试就变成"跟被测代码赛跑"，可读性极差。

- 追问：那 Mockito 的 `verify` 为什么可以事后？——因为 `verify` 不是"设定剧本"，是"核对发生了没有"，它读的是框架记的调用日志。**一个管"演什么"，一个管"演了没"**。

易错点 / 考点：必考 **录制—回放—检查** 三步，以及 EasyMock "可验证方法的调用种类、次数、顺序"。易错点：把 verify 说成"验证返回值"——verify 验的是**调用**，返回值是 when/thenReturn 那段的事。

过渡：EasyMock 之后就是今天的主角。Mockito 的三个核心动作，下一页用一张图给你摆开。

【5.63】Mockito（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 Mockito 的总览图，把用法分成三个动作：**设置依赖性、Stub 方法调用、验证交互性操作**。三个动作正好对应一条测试的生命周期。
>
> **设置依赖性**，就是"把替身造出来、塞进被测对象"。两种写法：代码式 `mock(Xxx.class)`，或注解式 `@Mock` ＋ `@InjectMocks` 自动注入。这一步解决"被测对象的协作者从哪来"。
>
> **Stub 方法调用**，注意这个说法——它是**用 Mockito 的 API 给 mock 对象打桩**：`when(mock.getPrice()).thenReturn(100)`。容易混的点是：**"mock 对象"是角色，"给它打桩"是动作**，打桩在术语上叫 stubbing，所以**一个 Mock 对象也可以被当作 Stub 用**——这就解释了 5.60 那句"Mock 的行为可能像 Stub"。
>
> **验证交互性操作**，就是 `verify(mock).save(any())`——检查方法有没有被调用、调了几次、参数是什么。这是 Mockito 区别于纯 Stub 工具的地方。
>
> 顺序也值得注意：**设置依赖 → 打桩 → 执行 → 验证**。打桩必须在执行之前，因为它改的是应答规则；验证在执行之后，因为它读的是调用记录。
>
> 一个工程建议：**这三个动作要能被一眼看出来**。标准做法是 Arrange 段集中 `mock`＋`when`，Act 段一行调用，Assert 段集中 `verify`＋`assert`。**结构清晰比技巧花哨重要**。
>
> 补一句定位：Mockito 是 Java 生态里**事实标准**，Spring Boot 的 `spring-boot-starter-test` 默认就带它。你不用选，只要会用。

> **板书：**三动作横排：① 设置依赖性（`mock()`／`@Mock`＋`@InjectMocks`）→ ② Stub 方法调用（`when(...).thenReturn(...)`，术语 stubbing）→ ③ 验证交互（`verify(...)`）；下面一行时序："设置依赖 → 打桩 → 执行 → 验证"；右侧一句"Mock 对象也能当 Stub 用"。

> **提问：**`when(mock.save(any()))` 出现在被测方法调用**之后**，会发生什么？为什么？

> **预设回答：**

- 答"没效果，桩打晚了"——基本对，但更准确的说法是：Mockito 会认为你在**执行一次真实调用**（默认返回

null），然后你又对这次调用打了桩，结果既没起到作用，还可能污染调用记录。

- 答"会报错"——某些情况下会（比如对未完成的 stubbing 做检测），但不保证报错，这才是危险的地方：**它可能静默失效**。

- 追问：所以怎么避免？——**把 when 全部写在 Arrange 段**，并且尽量在创建 mock 后立刻打桩，形成肌肉记忆。

> **易错点 / 考点：**三个动作的名称要能对上 API（设置依赖＝mock／@Mock；Stub 调用＝when／given；验证交互＝verify）。易错点：把"设置依赖性"理解成"配置依赖注入框架"，它在这里指的是**准备并注入替身对象**。

> **过渡：**三个动作展开就是 Mockito 的全部 API。下一页逐条过，这一页是本章的操作手册。

【5.64】Mockito 主要功能（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 Mockito 的功能清单，六条，按"你会怎么用"讲，每条给一句实战提醒。
>
> **`mock()`／`@Mock`：创建 mock。** 课件补了重要一句：可以通过 `Answer`／`MockSettings` 指定行为方式。`Answer` 是"方法被调用时执行什么代码"，比 `thenReturn` 灵活，可以返回值依赖入参。**提醒：给 mock 起名字**，出错信息里就能看出是哪个替身。
>
> **`when()`／`given()`：指定 mock 如何行动。** 这是同一件事的两种写法，`when` 是原生风格，`given` 是 BDD 风格。课件还说：返回不符合需求时可以自己写扩展，也就是自定义 `Answer`。
>
> **`spy()`／`@Spy`：部分模拟，调用真实的方法，但仍可被验证。** `mock` 造的是全假替身，所有方法返回默认值；`spy` 是包着真对象的壳，**默认调真方法，你只对需要的方法打桩**。典型场景：一个类十个方法，只让其中一个返回特定值。
>
> **`@InjectMocks`：自动注入带有 `@Spy` 或 `@Mock` 注释的字段。** 它先按类型再按名字注入，构造器、setter、字段注入都试。**注不进去是最高频的报错**，原因通常是类型对不上或用了 final 字段。
>
> **`verify()`：检查方法是否用给定参数调用。** 可配 `times(n)`、`never()`、`atLeastOnce()`、`inOrder()`。**注意 `verify` 默认是 `times(1)`**——很多人以

为是"至少一次"，于是调了两次的 bug 反而测不出来。

参数匹配和 `@Captor`。 匹配器有 `any()`、`eq()`、`anyString()`；`@Captor` 是参数捕获器——需要检查传进去的对象内容对不对，就捕获下来再断言。

有个坑必须说：**参数匹配器不能混用**，一个方法里要么全用匹配器，要么全用具体值，混着写会抛 `InvalidUseOfMatchersException`，这是新手第一大报错。

学习方法：**默写四条**——`mock`、`when`、`verify`、`@InjectMocks`，能覆盖 90% 日常场景；`spy` 和 `@Captor` 会用就行。

板书：六条竖排：① `mock()`／`@Mock`（可配 Answer／MockSettings）② `when()`／`given()` ③ `spy()`／`@Spy`（部分模拟，真方法）④ `@InjectMocks`（自动注入）⑤ `verify()`（次数・参数・顺序）⑥ 参数匹配 `any()`／`@Captor`；右下角红字"匹配器不能混用；verify 默认 times(1)"。

提问：一个类有十个方法，你只想让 `getDiscount()` 返回固定值，其余方法要跑真实逻辑。用 `@Mock` 还是 `@Spy`？如果用 `@Mock` 会怎样？

预设回答：

- 答"用 `@Spy`"——正确，这正是"部分模拟"的定义。

- 答"用 `@Mock`，给 `getDiscount()` 打桩"——那就出问题了：**其余九个方法全变成空实现**，返回 null／0，被测逻辑会莫名其妙地失败。

- 追问：那什么时候反过来该用 `@Mock`？——当被测对象**完全不该执行真实逻辑**时（比如它要连数据库），用 `@Mock` 隔离干净。**判据：真实逻辑对本条测试是"有帮助"还是"有害"**。

易错点 / 考点：必考六个 API 的作用，尤其 **`spy()`＝部分模拟、`@InjectMocks`＝自动注入、`verify()`＝检查调用**。三个高频错点：① `verify` 默认 `times(1)`；② 匹配器混用报错；③ `@Spy` 与 `@Mock` 的差别（前者调真方法）。还有一条判断题常出："Mockito 只能 mock 接口"——现在能 mock 类和 final 类（需 `mockito-inline`／新版默认支持）。

【思政落点 10】：`verify()` 这个 API 的名字值得琢磨——它叫"验证"，不叫"检查过了就行"。它要求你说清楚：**调了几次、参数是什么、顺序对不对**。这背后是一种**证据意识**：判断一件事有没有发生，不能靠"我觉得应该发了"，要靠可复核的记录。工程上大量事故源于"以为做了"——以为发了通知、以为清了缓存、以为关了口子。**把"以为"换成"有记录、可验证"，是责任心最具体的技术形态**。你以后写任何关键逻辑，都问自

> 己一句："这件事怎么才能被证明发生过？"

> **过渡：**清单里有一个注解被单独拎出来讲了一页，就是 `@Mock`。为什么它值得单独一页？

【5.65】Annotation Type Mock（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲 `@Mock` 这个注解的定位，课件给了四条，我把它们翻译成"为什么要有注解，而不是一路 `mock()`"。
>
> 第一条：**将一个字段标记为 Mock**——你声明一个字段，加上注解，Mockito 就帮你造好对象。
>
> 第二条：**允许速记的 Mock 创建**。对照一下就知道省了什么：不用注解是 `OrderRepository repo = mock(OrderRepository.class);`，用注解就是 `@Mock OrderRepository repo;`，省掉重复的类字面量。
>
> 第三条：**尽量减少重复的 Mock 创建代码**。一个测试类常有五六个协作者，每个都写一行 `mock(...)`，光准备段就十行，**把 Act 和 Assert 挤到看不见**；注解把噪音挪到字段声明区，让方法本体只剩业务。
>
> 第四条：**使测试类、验证错误更具可读性**——出错信息里带字段名（如 `orderRepository`），比 `mock for OrderRepository, hashCode: 1234567` 好读得多。
>
> 第五条：**自动检测 `MockedStatic` 类型的静态 Mock**。这是 mock 静态方法的能力，用注解可以把这个静态 mock 的生命周期纳入框架管理，**自动在合适的时候关闭**，不用手写 `try-with-resources`。**这是注解除了省代码之外的第二个价值：管生命周期**。
>
> 所以核心不是记四条，而是理解**注解的本质是声明式**：命令式说"我要做这几步"，声明式说"我声明这些东西是这样，剩下交给框架"。Spring 的 `@Autowired`、JUnit 的 `@Test` 都是同一思路。
>
> 最后提醒配套动作：**注解要生效，必须有初始化那一步**——`@ExtendWith(MockitoExtension.class)`，或手动 `MockitoAnnotations.openMocks(this)`。**忘了这一步，`@Mock` 字段就是 null**，这是新手第二大报错。

> **板书：**四条：① 标记字段为 Mock ② 速记创建（vs `mock(X.class)`）③ 减少重复代码 ④ 提高可读性（报错带字段名）＋⑤ 自动管理 `MockedStatic` 生命周期；底部红字"注解要生效：`@ExtendWith(MockitoExtension.class)` 或 `openMocks(this)`，否则字段是 null"。

> **提问：**为什么说注解除了"省代码"，还解决了"生命周期"问题？请用静态 Mock 举例。

> **预设回答：**

- 答"因为静态 Mock 用完必须关，忘了关会影响其它测试"——对。静态方法被 mock 后是**全局生效**的，不关闭会污染整个 JVM 里的其它用例。

- 答"注解会自动开关"——准确。框架知道什么时候创建、什么时候释放，比你手写 `try-finally` 稳。

- 追问：这告诉我们注解的通用价值是什么？——**把"必须成对出现的操作"（开／关、建／销）交给框架管**，人最容易忘的就是收尾。

> **易错点 / 考点：**常考"使用 @Mock 注解需要做什么前置动作"——答案是**启用 Mockito 扩展**（`@ExtendWith(MockitoExtension.class)`）或调用 `openMocks`。易错点：以为加了 `@Mock` 就自动注入到被测对象——**注入是 `@InjectMocks` 的事**，两个注解解决两个问题，别混。

> **过渡：**注解那一页提到了 `given()` 这种写法，它是 BDD 风格的入口。下一页讲 Mockito 怎么支持 BDD。

【5.66】支持 BDD（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲 Mockito 对 **BDD（行为驱动开发）** 的支持，核心一件事：**它提供了一套 BDD 风格的 API，让测试代码读起来像一句需求描述。**
>
> 先说 BDD 是什么。核心是用业务语言描述行为，典型句式 **Given（给定前提）— When（当发生某事）— Then（那么应该如此）**。比如"给定账户余额 100 元，当我取款 50 元，那么余额应为 50 元"。这句话业务方看得懂，代码也能照着写。
>
> Mockito 的 `BDDMockito` 三个方法正好对上：**`given(...)`** 对应 Given，给定替身的行为，等价于 `when(...)`；**`willReturn(...)`／`willThrow(...)`** 是它的搭档，写法 `given(mock.getPrice()).willReturn(100)`——**语序变了**，"当调用时，就返回"变成"给定调用，将返回"，后者更像英语句子；**`then(...)`** 对应 Then，`then(mock).should().save(any())`，读起来就是"那么它应该保存过"。
>
> 为什么值得单独一页？因为它是**可读性的实质提升**，而且把测试和需求的表述对齐了——方法名可以直接写

`given_balance_100_when_withdraw_50_then_balance_50`，报告生成出来能给业务方看。**测试报告变成验收证据**，在金融、医疗这类受监管行业价值很大。

但要给一个冷静提醒：**BDD 风格是语法糖，不是新能力**。它跟 `when/verify` 完全等价，不存在"用 BDD 测得更准"。所以**团队统一一种风格就行，别在一个类里混着写**——混着写的结果是读代码的人要在两套思维间切换，比只用一种更糟。

一句话收口：**BDD 的语法是为了让人读懂，不是为了更正确**。

> **板书：**三行对照：`given(...)` ≈ `when(...)`（给定前提）｜`willReturn／willThrow` ≈ `thenReturn／thenThrow`（将返回）｜`then(mock).should()` ≈ `verify(mock)`（那么应如此）；右侧写 "Given—When—Then ↔ 需求语句"；底部一句"语法糖，不新增能力；一个类只用一种风格"。

提问：如果团队一半人用 `when/verify`、一半人用 `given/then`，会不会出问题？出在哪？

预设回答：

- 答"不会，功能一样"——功能一样，但**认知成本翻倍**：读代码时要在两套语序间切换，评审时容易漏。

- 答"会有维护问题，风格不统一"——正确，这就是"约定优于配置"的意义。

- 追问：那怎么定？——看团队有没有 BDD 的落地场景（有没有业务方看测试报告）。有，就统一 BDD 风格；没有，`when/verify` 更短更直接。

易错点 / 考点：考点是"Mockito 的 BDD 支持体现在哪"——答案：提供 `given／willReturn／then` 这类 BDD 风格 API（`BDDMockito`），语义与 `when／thenReturn／verify` 等价。易错点：把 BDD 说成"一种测试类型"（它是**开发/协作方法**，不是测试层级）；以及把 `then(...).should()` 说成"断言返回值"（它验证的是**交互**）。

过渡：单元测试的"写"讲完了，接下来要回答一个更硬的问题：**我怎么知道测够了？** 这就进入覆盖率工具。

【5.67】测试覆盖率工具（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是覆盖率工具的入口，任务是把"这类工具干什么、有哪几个代表"讲清，原理下一页展开。

先说它要回答什么问题：**你的测试到底跑到了代码的哪些部分？** 它不判断对错，只做一件事——**统计执行痕**

迹。所以覆盖率工具从来不提高质量，它只是**照镜子**：照出哪些代码从来没被跑过。

课件后面给了一串名字，我按语言归类。**Java：JaCoCo**，生态的标准答案，通过对编译后的 class 文件插桩统计覆盖情况；还有 **Jcov**、**CodeCover**、**CoverageMeter**，多用于特定场景。**C/C++：BullseyeCoverage**，课件特别点了它"支持分支覆盖率（Branch Coverage）的分析"；还有 **Testwell CTC++**（嵌入式很有名，能在目标板上跑）、**Parasoft C++Test**（测试＋静态分析＋覆盖率）。**Python：Coverage.py**，命令行 `coverage run` ＋ `coverage report`，简单直接。

这么多名字，选型三条：① **跟着语言走**（Java 就 JaCoCo，别折腾）；② **看能不能出标准报告格式**（Cobertura／XML／LCOV），因为 CI 平台和 SonarQube 要消费这份报告；③ **看插桩方式会不会影响运行**（改字节码、源码插桩、硬件探针，各有代价）。

最后补一个观念，本页最重要：**覆盖率必须"无侵入地"接进流程才有价值**。它不是"想起来了跑一次"，而是每次构建自动跑、报告自动上传、覆盖率掉了自动告警。**一次性测量毫无意义**，因为覆盖率是**会腐烂的指标**——今天 85%，明天加个新类掉到 70%，没人盯着，三个月后就是 40%。

板书：三列按语言：Java（JaCoCo・Jcov・CodeCover・CoverageMeter）｜C/C++（BullseyeCoverage・Testwell CTC++・Parasoft C++Test）｜Python（Coverage.py）；底部两条："工具只照镜子，不提质量""覆盖率会腐烂，必须进流水线"。

提问：覆盖率工具能不能发现"某段代码逻辑写错了"？如果不能，它的价值到底在哪？

预设回答：

- 答"不能，它只看执行没执行"——正确。它没有"对错"的概念，只有"到没到"的概念。

- 答"它的价值是告诉团队哪里还没测"——对，这是核心价值：**它把"未知"变成"已知的缺口清单"**。

- 追问：所以"覆盖率 90% 但线上出大事故"这种事正常吗？——完全正常，说明那 90% 里**断言不足**，或者**缺陷在输入设计没覆盖的边界上**。覆盖率是**下限保障**，不是质量证明。

易错点 / 考点：常考"覆盖率工具与语言对应"：**JaCoCo（Java）、BullseyeCoverage（C/C++，支持分支覆盖）、Testwell CTC++（C/C++）、Coverage.py（Python）、gcov／lcov（C/C++）**。易错点：把 JaCoCo 说成"测试框架"；把 BullseyeCoverage 说成"Java 工具"；以及把覆盖率工具说成"能提高覆盖率"（它只测量，提高靠写用例）。

过渡：下一页开始逐个看这些工具的实现机制，先看 JaCoCo 是怎么"不改源码也能统计"的。

【5.68】代码覆盖率分析工具（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 我们先把这一节的问题摆出来。假设你写了三百个单元测试，全部绿灯，那能不能说"这段代码被测过了"？不能。因为用例通过只说明**你跑过的那几条路径**是对的，它完全没告诉你**还有多少代码从来没被执行过**。覆盖率工具就是回答这个问题的——它不判断对错，它只统计"哪些指令被走到过、哪些分支一个方向都没走"。
>
> 那这个统计怎么做？靠人眼看代码是做不到的。一次回归可能跑几千个用例，执行路径的分布只有机器能记下来。所以工具的做法是**在代码里埋探针**：程序跑到哪儿，就把哪儿标记一下，跑完把标记汇总成报告。JaCoCo 就是在**编译之后的 Java class 文件**上做这件事，而不是改你的源码——这一点很关键，下一页我展开。
>
> 工具不是只有 Java 一家。做 C/C++ 常用的一个是 **BullseyeCoverage**，它特别强调**分支覆盖率**的分析。为什么强调分支？因为 C/C++ 里一个 `if` 的两个方向、`switch` 的每个 case、`&&` 和 `||` 的短路，都可能是缺陷的藏身处，只看"这行执行过没有"远远不够。同一个生态里还有 **Testwell CTC++**、**Parasoft 的 Jtest 和 C++test**、**CoverageMeter**、**Jcov**、**CodeCover**；Python 那边是 **Coverage.py**。
>
> 你会发现一个规律：**每个语言生态都有自己的覆盖率工具，但原理是同一套**——插桩、记录、汇总、出报告。所以这一页不要背工具名，要记住那条原理链，工具名只是这条链在不同语言上的实现。
>
> 最后留一句提醒：覆盖率是**度量**，不是**目标**。当目标是它的时候，人会去写"能把数字刷上去"的用例；当它是度量的时候，人才会去问"这个分支为什么没被走到"。这句话下一页我给你讲透。

> **板书：**覆盖率工具原理链：插桩 → 记录 → 汇总 → 报告；Java：JaCoCo（对编译后 class 插桩）；C/C++：BullseyeCoverage（专看分支覆盖）、Testwell CTC++；其他：Jtest／C++test、CoverageMeter、Jcov、CodeCover、Coverage.py。右下角写一句"覆盖率是度量，不是目标"。

> **提问：**某团队覆盖率从 62% 涨到 91%，你能不能由此得出"质量变好了"这个结论？

> **预设回答：**

- 答"能，涨了就是好事"——先不否定，但要追问一句：那 9 个百分点是**新增用例**涨的，还是**删掉了难测的代码**涨的？如果是后者，质量并没有变好，只是账变好看了。

- 答"不能，覆盖率只说明执行过，不说明断言对"——这是准确答案。覆盖率高但断言写成 `assertTrue(true)`，等于什么都没验。

- 答"要看分支覆盖涨没涨"——很好，这是内行的判断：语句覆盖涨了、分支覆盖没动，说明只是"多跑了几行直路"。

- 收口：所以看覆盖率要看三件事——**哪个维度涨的、涨在哪些类上、配套的断言有没有跟上**。

> **易错点 / 考点：** ① 最常见的错是"覆盖率高＝测试充分"，判断题里几乎每年出现，正确答案永远是"覆盖率是必要不充分条件"；② 插桩对象是**编译后的 class**，不是源码，选择题常把"修改源文件插入统计代码"当干扰项；③ 工具与语言的对应关系要记牢：BullseyeCoverage 是 **C/C++**，且主打**分支覆盖**；JaCoCo 是 Java；Coverage.py 是 Python。

> **过渡：** 原理说完了，具体到 JaCoCo，它到底给你几个维度的数字？我们一页看全。

【5.69】实现覆盖率度量：Jacoco（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页考得最细，我们拆成四块看：**给什么数字、凭什么能算、怎么装进去、能出什么报告**。
>
> 第一块，**六个分析维度**：指令、分支、行、方法、类、圈复杂度。这六个不是同级概念。**指令覆盖**最细，数的是字节码指令被执行的比例；**分支覆盖**数的是每个判定点的两个出口走了几个——`if` 只走 true 没走 false，这叫"部分覆盖"；**行覆盖**按源码行算；**方法覆盖**和**类覆盖**更粗，方法里有一条指令被执行就算覆盖。而**圈复杂度**不是覆盖率，它是**复杂度指标**，McCabe 的 `e − n + 2p`，JaCoCo 按方法算，告诉你"这个方法最少要几条用例才能把路径走全"。所以报告里**圈复杂度高、覆盖率低**的方法，就是最该补用例的地方。
>
> 第二块，**基于 Java 二进制码、无需源文件**。它统计的是 class，源码只用来展示。那怎么高亮到具体某一行？因为编译时默认生成了行号调试信息，class 里带着"这条指令对应第几行"的对照表。这也解释了一个现象：关掉调试信息编译，你就只剩方法级、类级的数字。
>
> 第三块，**on-the-fly instrumentation 的 java agent 集成**——边跑边插：JVM 启动时挂 agent，类被加载的那一刻改字节码、塞探针，全程不动磁盘上的 class。另一种叫 offline，先插桩再跑。它能集成到 OSGi、Web 容器、EJB 服务器这类基于 Java VM 的应用，兼容所有 class 版本、支持不同 JVM 语言——Kotlin、Scala 编译出来也是 class。

第四块，**报告与数据**：报告有 HTML、CSV、XML 三种，HTML 给人看，另两种给流水线做门禁；执行数据可通过远程协议和 JMX 随时从 agent 里取；构建侧用 ANT task 或 Maven plugin 收集数据并生成报告。

板书：六维度竖排：指令（最细）→ 分支（判定出口）→ 行 → 方法 → 类（最粗）；旁边单独一列写"圈复杂度＝复杂度指标，不是覆盖率（按方法算，e−n+2p）"。第二行写"统计对象＝class；行级高亮靠调试信息"；第三行写"on-the-fly agent：加载时改字节码"；第四行写"报告 HTML／CSV／XML；数据 dump：远程协议・JMX；构建：ANT／Maven"。

提问：一个方法行覆盖 100%、分支覆盖只有 50%，问题出在哪？你应该补什么用例？

预设回答：

- 答"代码有 bug"——不成立，覆盖率和 bug 不是一回事；覆盖率低只说明**没测到**，不说明**测错了**。

- 答"说明有分支没走，要补让判定取反的用例"——对。典型场景是 `if (x > 0)` 只测了正数，负数那一支从没进过，异常处理分支也没进过。

- 答"把行覆盖刷到 100% 就够了"——要点出这是最危险的想法：行覆盖 100% 时，短路运算的右半边、`else` 的空分支照样可能完全没测。

- 收口：补用例的顺序是**先看分支、再看圈复杂度**——分支告诉你缺哪个方向，圈复杂度告诉你这个方法还差多少条路径。

易错点 / 考点：① 六个维度里**圈复杂度不属于覆盖率**，这是最爱考的陷阱，问"JaCoCo 提供哪六个维度"要全答，问"哪些是覆盖率指标"要把圈复杂度摘出去；② "基于二进制码、无需源文件"常被改成"需要源代码"来考判断；③ on-the-fly 与 offline 的区别：前者运行时插桩、不改文件，后者先插桩后运行；④ 报告格式三种要背全：HTML、CSV、XML。

>

记忆口诀："指分行方类圈"——前五个是覆盖率的从细到粗，最后一个是复杂度，别混进去。

【思政落点 11】：覆盖率最容易被用错的地方，是把它当成**交差的凭证**——数字到 80% 就收工，报告一贴，任务完成。但覆盖率真正的用处是**指路**：它告诉你哪一段代码从来没人走过，那些地方往往就是线上事故的产地。我希望大家养成一个习惯：看到红黄格子，第一反应不是"怎么把它变绿"，而是"这块逻辑为什么没人测、它出错会怎样"。**把度量当镜子而不是当奖状**，这就是工程人员该有的诚实。

过渡：下一页开始看源码。别怕，我不带你逐行读代码，我们只看它"在干什么"——因为这三页合起来，就

> 是覆盖率工具的心脏。

【5.70】on-the-fly instrumentation（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这是 JaCoCo agent 的入口方法 `premain`。名字已经说明了它的身份——**在 main 之前**。只要在启动参数里加 `-javaagent:jacocoagent.jar`，JVM 就会去这个 jar 的清单里找 `Premain-Class`，然后调用它的 `premain`，**在你的程序跑第一行代码之前先给它一个动手的机会**。
>
> 它只干三件事。第一，**读配置**：`new AgentOptions(options)`，把启动参数等号后面那串键值对（排除哪些包、数据文件落哪、端口多少）解析成配置对象。第二，**建通道**：`createRuntime(inst)` 造出运行时，再 `runtime.startup(...)`，这个运行时负责收数据、存数据、对外提供 dump 的口子——前面说的远程协议和 JMX 就是这儿暴露的。第三，`inst.addTransformer(new CoverageTransformer(...))`，往 JVM 注册一个**字节码转换器**。从这一刻起，**每个类被加载都要先过这道转换器**：读进来、插探针、再交给 JVM。
>
> 注意一个细节：它传了 `IExceptionLogger.SYSTEM_ERR`，意思是哪个类不兼容、分析失败，**不要把整个应用搞崩**，记到标准错误然后放行。
>
> 所以这一页记住一句话：**premain 不是插桩的地方，是装钩子的地方**。钩子装好，插桩发生在每一次类加载的瞬间。

> **板书：**画一条竖线代表 JVM 启动流程：`-javaagent` → 找 Premain-Class → **premain（本页）** → 应用 main。premain 下挂三个动作：① `AgentOptions` 读配置 ② `createRuntime`＋`startup` 建数据通道 ③ `addTransformer(CoverageTransformer)` 装钩子。旁注："插桩不在这一页，在每次类加载时"。

> **提问：**如果某个第三方 jar 的字节码格式特别老、转换器分析不了，按这段代码的逻辑，会发生什么？

> **预设回答：**

- 答"整个应用启动失败"——不会，因为转换器传了异常记录器，异常被记到 stderr，类原样放行。
- 答"那个类不被统计，其他类照常统计"——对，这是设计意图：**覆盖率工具不能成为应用的故障源**。
- 追问：这体现了一个什么工程原则？答"工具的可降级性"——监控、埋点、日志这类旁路设施，都应该是"出问题不影响主流程"。

易错点 / 考点：① `premain` 是 **agent 入口**，不是插桩实现，问"JaCoCo 在哪一步改字节码"要答"类加载时的 Transformer"；② 启动参数形式是 `-javaagent:xxx.jar=key=value`，考试常把 `-agentlib`、`-Xbootclasspath` 混进来当干扰项；③ `addTransformer` 注册后对**之后**加载的类生效，启动前已经在 JVM 里的核心类不在其列。

过渡：钩子装好了，那"插"这个动作具体怎么写？下一页是 ASM 的活。

【5.71】on-the-fly instrumentation –续（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是 `instrument` 方法：输入一段字节码，输出插好探针的字节码。把它当成一条**字节码加工流水线**：进门是原始 class，出门是带探针的 class。

第一步 **读**：`ClassReader` 把 class 字节流解析成可遍历的结构。第二步 **备写出器**：`new ClassWriter(reader, 0)`，这里重写了 `getCommonSuperClass` 直接抛异常——为什么？因为 ASM 判断两个类型的父子关系时默认会**去加载这两个类**，而我们正处在"类加载"这个环节里，再去触发类加载，轻则死锁重则栈溢出。JaCoCo 的假设是：我只插探针、不做类型变换，栈帧的公共父类不会变，所以这个能力根本不需要。抛异常是一种"声明式拒绝"。

第三步 **决定探针存哪**：`ProbeArrayStrategyFactory.createFor(...)`，探针要有一个数组存"走没走过"，放静态字段、每次新建还是外部注入，得看类的形态，工厂替你选。第四步 **串访问者链**：`ClassProbesAdapter` 套 `ClassInstrumenter`，再套 writer——外层管这个类有几个探针，中层管探针怎么落进字节码，最里层管写出去。

第五步 `reader.accept(visitor, ClassReader.EXPAND_FRAMES)` 开始遍历，`EXPAND_FRAMES` 把栈映射帧展开，方便精确插桩；最后 `toByteArray()` 拿到新字节码。

一句话：它自己实现了"编译器的后半段"——不生成源码，直接在字节码上做等价变换，还要保证结果能过 JVM 校验。

板书：流水线框图：`class 字节 → ClassReader →（ClassProbesAdapter → ClassInstrumenter → ClassWriter）→ 带探针的 class`。旁注三点：① 重写 `getCommonSuperClass` 抛异常＝拒绝触发类加载；② `ProbeArrayStrategy` 决定探针数组存哪；③ `EXPAND_FRAMES` 展开栈帧便于插桩。

> **提问：**为什么插桩工具要这么小心地"不加载别的类"？你在别的地方见过同类的约束吗？

> **预设回答：**

- 答"怕慢"——不完全是，慢是次要的；主要是**类加载本身有锁**，在加载过程中再去加载，很容易死锁。

- 答"类加载器隔离，可能加载错版本"——很到位。同一个类名在不同 ClassLoader 下是两个类，工具贸然加载会拿到错误的类型信息。

- 追问：还有哪儿有类似约束？答：Java agent、热部署、APM 探针、反射框架——**凡是工作在"类加载期"的代码，都要遵守这条纪律**。

> **易错点 / 考点：**① 这一段用的库是 **ASM**（字节码操作框架），考试常问"JaCoCo 用什么改字节码"；② `ClassWriter` 构造参数里的 `reader` 是为了复用常量池、减小输出体积，不是"读源码"；③ `EXPAND_FRAMES` 是遍历选项，不是插桩策略，别混。

> **过渡：**类这一层过去了，方法这一层怎么插？下一页是最后一段，也是最见功力的一段。

【5.72】on-the-fly instrumentation –续（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 `visitMethod`，访问者走到"一个方法"时的回调。它的任务不是马上插，而是**先包一层壳，等方法全部走完再动手**。
>
> 先看结构：问下层"这个方法你接不接"，不接就用空的探测器顶上；接了则返回一个 `MethodSanitizer` 的匿名子类——**名字就是信息：清理器**，有些编译器生成的字节码不规范，得先规整再插，否则过不了校验。
>
> 关键在 `visitEnd`，到这一刻才掌握方法的完整结构，于是按序做四件事。一，`markLabels` **标出所有跳转标签**，搞清哪条指令跳到哪，才能找到控制流的**汇合点**——探针必须挂在分支真正分开又真正汇合的位置，挂别处要么重复计数要么漏计。二，`MethodProbesAdapter` 把探针和标签对应起来。三，`trackFrames` 为真时套 `AnalyzerAdapter`，**模拟操作数栈和局部变量表的变化**：你插了指令，栈深度就变了，不重算栈帧，JVM 校验直接给你 `VerifyError`。这是插桩最硬的门槛——**你改的不是文本，是一个必须严格自洽的栈式程序**。四，`methodProbes.accept(...)` 真正注入。
>
> 三页连起来：**premain 装钩子 → instrument 改类 → visitMethod 改方法**。回头看那六个维度就有底了：指

令和行来自探针位置，分支来自标签配对，圈复杂度来自控制流图的静态计算——**全是这次字节码遍历的副产品**。

板书：`visitMethod` → 返回 `MethodSanitizer` 壳 → `visitEnd` 四步：① `LabelFlowAnalyzer.markLabels` 找汇合点 ② `MethodProbesAdapter` 绑探针 ③ `AnalyzerAdapter` 重算栈帧（防 VerifyError） ④ `methodProbes.accept` 注入。顶部写一条总线："premain 装钩子 → instrument 改类 → visitMethod 改方法"。

提问：为什么探针一定要插在"控制流汇合点"，插在每条指令前面不行吗？

预设回答：

- 答"每条都插更准"——方向对，但代价是**体积和性能爆炸**：一个类可能被插进成千上万个探针，测试跑起来慢到不可用，而且分支计数会重复。

- 答"汇合点少，计数还能对上"——对。分支覆盖要的是"这个判定走没走两个方向"，一个判定一个探针就够。

- 追问：这说明了工具设计的什么取舍？答：**度量精度与运行开销之间的取舍**，这也是所有埋点系统的共同命题。

易错点 / 考点：① 三个方法的分工要能对上号：`premain`＝入口、`instrument`＝类级、`visitMethod`＝方法级；② `AnalyzerAdapter`／栈映射帧是**防校验失败**的，不是"分析业务逻辑"；③ 这三页属于"原理了解"层级，考法以**填空、排序、简答**为主，不会让你写 ASM 代码；④ 常见错误说法："JaCoCo 通过修改源文件插入统计代码"——从头到尾它碰的都是 class。

过渡：理论讲完，我们看一眼真实的报告长什么样——这一页是整节最直观的一页。

【5.73】示例（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这是一张 JaCoCo 报告的实图。我不要你记界面，我要你学会**三秒钟读出一份覆盖率报告**。

第一眼，看**颜色**。绿色是"完全覆盖"，所有分支都走到了；黄色是"部分覆盖"，最常见的情形是一个判定只走了一个方向；红色是"完全没执行"。注意——**真正值钱的是黄色**。红色一眼就知道没测，反而好处理；黄色最容易被忽略，因为它看起来"测过了"，实际上有一半逻辑从没被验证过。

第二眼，看**层级**。报告是按包 → 类 → 方法 → 行逐层下钻的。上面几层给的是汇总数字，用来定位"哪个包最差"；点进去到方法层，才能看到"哪个方法最差"；再点到行，才知道"哪一行是黄的"。**自上而下找问题，自下而上补用例**——不要一上来就翻源码找红线，那样效率最低。

第三眼，看**列**。每一层都有 Missed 和 Covered 两列，按六个维度分栏：指令、分支、行、方法、类、复杂度。判断一个模块该不该补测试，我的经验是看两个组合：**分支覆盖率低、圈复杂度高**——这两个同时出现，说明这块代码逻辑分支多、还没测到，是风险最高区。

还有一个技巧：报告里可以按"覆盖率升序"排。排完排在最前的那几个类，就是这个项目最该补测试的地方。这比"从头到尾看一遍"高效一个数量级。

顺带说一句，为什么这张图能高亮到具体某一行？因为 class 里带着行号调试信息——这就印证了前面那句"基于二进制码，无需源文件"：源码是用来**展示**的，统计口径始终在字节码上。

> **板书：**写三行读图法：① 看颜色（绿＝全覆盖／黄＝部分覆盖＝重点／红＝未执行）② 看层级（包→类→方法→行，自上而下定位、自下而上补用例）③ 看列（Missed／Covered × 六维度）。右下角写判断公式：**分支覆盖率低 + 圈复杂度高 = 最该补测试**。

> **提问：**一份报告里红格子很多，但都是 getter/setter 和自动生成的代码；黄格子只有三个，都在核心业务方法里。你先处理哪边？

> **预设回答：**

- 答"先补红的，数量多"——数字思维，但方向错了：getter/setter 补上去对质量几乎没贡献。

- 答"先看黄的，因为它在核心逻辑里"——对。而且更专业的做法是：**把自动生成的代码从统计里排除**（agent 的 excludes 配置），让报告只反映人写的、有业务含义的代码。

- 追问：这说明覆盖率统计还有一个前置工作是什么？答：**划定统计范围**。范围不划，数字就没有可比性。

> **易错点 / 考点：**① 黄＝部分覆盖（分支只走一边），不是"跑得慢"；② 覆盖率报告的读法是"自上而下定位、自下而上补"，考试若问"如何用覆盖率报告指导补测"，要答出这两个方向；③ 案例题常给一张报告截图问"下一步该做什么"，标准答法是"定位分支未覆盖的方法＋排除自动生成代码＋补判定取反的用例"。

> **过渡：**报告看得懂还不够，工具得能顺手用起来。Eclipse 里那个一键跑覆盖率的插件，就是下一頁。

【5.74】代码覆盖率度量（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲的是 **EclEmma**，一句话定位：**它是基于 JaCoCo 的 Eclipse 插件**。注意这个关系——JaCoCo 是**引擎**，EclEmma 是**外壳**。引擎负责插桩、统计、算数；外壳负责让你在 IDE 里点一下就能跑、跑完直接在源码上染色。

它在开发者的日常里解决什么问题？解决**反馈距离**的问题。你在写一个方法，写完随手点一下"以覆盖率方式运行"，绿黄红立刻出现在编辑器左边那条竖条上，哪个分支没走，一眼就看到。这个反馈只要几秒钟。而如果只能等流水线出报告，反馈要几十分钟——**反馈延迟越长，人越不愿意改**，这是工程心理学，不是技术问题。

但要注意它的局限，这也是考点。第一，**它只覆盖你本机跑的那一次**，别人写的用例、流水线跑的集成测试都不在内；第二，**它依赖 IDE**，CI 服务器上没有 Eclipse；第三，**它容易被当成"我测过了"的证据**。所以企业的正确做法是**两条腿走路**：本地用 EclEmma 做即时反馈，流水线上用 JaCoCo 的 Maven／Gradle 插件出正式报告、做质量门禁。一个是"照镜子"，一个是"过秤"。

我再补一个工程细节：EclEmma 装好之后，Eclipse 里会多出一个"覆盖率"视图，你可以对**整个项目**跑，也可以对**单个测试类**跑。建议养成对"当前正在改的那个类"跑的习惯——范围越小，反馈越快，改起来越有针对性。

那这一节到此收口：**工具能告诉你哪儿没测，但不能替你决定"测得够不够"**。标准要人来定，工具只负责把事实摆出来。

> **板书：**中间写 `JaCoCo（引擎：插桩・统计・算数）`，外面套一个框写 `EclEmma（外壳：IDE 集成・源码染色）`。下面两行对比：本地 EclEmma＝即时反馈／范围仅本次；流水线 JaCoCo 插件＝正式报告／质量门禁。旁边一句："工具摆事实，人定标准。"

提问：既然流水线已经有 JaCoCo 报告了，开发还有必要在本地装 EclEmma 吗？

预设回答：

- 答"没必要，重复了"——忽略了反馈延迟这个变量。等流水线报告出来，人已经切到下一个任务了，改的意愿会大幅下降。

- 答"有必要，本地改起来快"——对，但要补一句：本地看到的数字**不能拿去汇报**，因为它只反映你这一台机器

上跑过的东西。

- 追问：所以两个工具的定位差别是什么？答：**EclEmma 是开发的自检工具，JaCoCo 流水线报告是团队的验收凭证**。同一个引擎，两种用途。

> **易错点 / 考点：**① EclEmma 与 JaCoCo 的关系必考：**EclEmma 是基于 JaCoCo 的 Eclipse 插件**，别答成"两个独立工具"或"JaCoCo 是 EclEmma 的插件"（方向反了）；② EclEmma 历史上曾基于 EMMA，现在用 JaCoCo 引擎——考题若出现"EclEmma 基于 EMMA"要判错；③ 常考判断："IDE 里的覆盖率等于项目的覆盖率"——错，范围不同。

> **过渡：**覆盖率这一节到这儿收住了。从这里开始，我们换一个更大的问题：**一个个方法都测过了，把它们拼起来还能不能跑？**

【5.75】5.3 持续集成测试（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 我们进入本章的第三块——**持续集成测试**。先把口径说清楚：教材里"系统集成的模式与方法"和"持续集成及其测试"是两节，课件把它们合成了一节叫"持续集成测试"，**内容没少，只是编排合并了**，答题时按教材口径写，听课时按课件顺序走。
>
> 这一块要解决什么问题？前面单元测试把每个函数钉死了，可系统出问题，十次有八次不是某个函数算错了，而是**两个都对的东西拼在一起不对**：接口参数顺序对不上、时间格式一个是本地时间一个是 UTC、A 服务以为 B 会重试、B 以为 A 只发一次。这些错误在单元测试里**永远测不出来**，因为单元测试把对方都 Mock 掉了。
>
> 这一块分三层递进。**第一层是单体架构**：所有代码在一个进程里，问题是"按什么顺序把模块拼起来测"，答案是三种策略——自顶向下、自底向上、混合。**第二层是微服务架构**：模块变成了独立部署的服务，拼装变成了跨进程调用，问题升级成"我根本没法把别人的服务随时拉起来"，答案就是**契约测试**。**第三层是持续集成与 CI/CD 流水线**：拼装不再是人工动作，而是每次提交都自动发生，这时候测试必须**又快又准又有序**，这就是持续测试。
>
> 还有一个学习上的提醒。这一块的内容看起来"软"——没有公式、没有算法，全是策略和做法，于是很容易被当成"听听就行"。但恰恰是这块内容最像真实工作：你以后进项目，八成会先被安排去"把这两块拼起来测通"，而不是先让你设计算法。**集成测试是新人最快接触到全局的地方**，因为拼装的时候，你必须同时理解两

边的接口。

请你带着一个问题听这一块：**你们项目上一次联调，卡了几天？卡在谁等谁？**

> **板书：**中间写"5.3 持续集成测试"，下面三行：① 单体：拼装顺序（自顶向下／自底向上／混合）② 微服务：跨进程（契约测试 CDC）③ 流水线：自动拼装（持续测试）。右上角小字备注："教材 5.7／5.8 两节，课件合并为 5.3"。

提问：一个函数单元测试 100% 通过的系统，集成的时候最可能先在哪一类问题上翻车？

预设回答：

- 答"接口对不上"——最常见的一类，参数名、类型、空值约定、编码格式，任何一处不一致都会炸。

- 答"环境问题"——也很真实：配置项、依赖版本、数据库字符集，这些在单元测试里全被 Mock 掉了。

- 答"并发和时序"——最深的一类，单线程测不出，两个服务一并行就暴露。

- 收口：这三类问题的共同点是"**它们都发生在边界上**"，而集成测试的全部工作，就是**盯着边界看**。

易错点 / 考点：① 口径陷阱：课件叫"持续集成测试"，教材分"系统集成的模式与方法"和"持续集成及其测试"两节，答题看题干用哪个词；② 集成测试的对象是**接口与交互**，不是"把所有单元测试再跑一遍"；③ 集成测试**不是**系统测试，它不验证业务需求，只验证"拼起来对不对"。

过渡：先看最小的场景——所有代码都在一个进程里的单体架构，按什么顺序拼？

【5.76】持续集成测试内容（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是这一块的目录，四条：**单体架构的集成测试、微服务架构的集成测试、持续集成及其测试、CI/CD 流水线**。我按"由小到大、由静到动"给你串一遍。

前两条是**架构维度**：单体怎么拼、微服务怎么拼。它们的差别不在技术难度，而在**你能不能控制对方的运行**。单体里所有模块都在你手里，你想先测谁就先测谁；微服务里对方是另一个团队、另一个发布节奏、另一套数据库，你连"把它启动起来"都做不到。所以单体的答案是"策略"，微服务的答案是"契约"。

后两条是**节奏维度**：持续集成讲"多久拼一次"，CI/CD 流水线讲"拼完怎么送到生产"。这两条回答的是同一个

问题——**当拼装从"项目末期干一次"变成"每天干几十次"，测试要怎么变**。

为什么集成测试值得单开一节？因为在测试的投入结构里，**它是最容易被压缩、又最贵的一段**。单元测试写起来快、跑起来快，大家愿意写；系统测试有明确的验收标准，不敢省；只有集成测试卡在中间——不写，出了问题查不出来；写全，环境、数据、依赖全是成本。所以这一节的核心不是"要不要做集成测试"，而是"**怎么用最少的成本，把边界上的风险盖住**"。

再补一句学习方法：这四条不是四个并列的知识点，而是**一条递进的线**。你听的时候不要一条一条记，而要问自己"如果架构从单体变成微服务，上一条的答案哪里失效了？如果节奏从每月一发布变成每天十发布，上一条的成本哪里撑不住了？"——**失效的地方，就是下一条要解决的问题**。这样记，四条就变成一条逻辑，不容易忘。

这四条有个共同的关键词：**边界**。你听后面的内容时，只要盯住"这里的边界是什么、谁在边界两边、边界上传递什么"，这一节就不会乱。

板书：四行目录：① 单体架构集成测试（策略：自顶向下／自底向上／混合）② 微服务架构集成测试（外部通信・数据库访问・契约 CDC）③ 持续集成及其测试（节奏）④ CI/CD 流水线（交付）。左侧竖写一条主线："架构维度（怎么拼）→ 节奏维度（多久拼一次）"，右侧竖写"边界"。

提问：为什么"微服务架构的集成测试"要单独成一条，而不能用单体的三种策略套上去？

预设回答：

- 答"因为服务是分开部署的"——这是表层原因，继续追问：分开部署带来什么后果？答：**你无法控制对方何时可用**，Stub 和 Driver 那套"我编个假的顶上"在跨团队、跨语言时成本极高。

- 答"因为技术栈可能不同"——也对，一个 Java 服务调一个 Go 服务，你的 Stub 要按人家的协议造，改一次协议你就要改一次假服务。

- 追问：所以微服务需要的不是"更好的假模块"，而是什么？答：**一份双方都认的、可自动验证的约定**——这就是契约测试。

易错点 / 考点：① 四条内容要能默写，常考填空；② 别把"持续集成"当成一种测试类型，它是**实践／节奏**；③ 别把 CI/CD 流水线当成"部署工具"，它在这一节里是**测试的执行载体**。

过渡：从最小的场景开始——单体架构，先测谁？

【5.77】单体架构的集成测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

单体架构下，所有模块在一个进程里，问题就变成一道很朴素的题：**先拼哪两个模块、按什么顺序拼**。课件给了两种基本策略，我们把它们对照着看。

第一种，自顶向下。从最上层的**主控模块**开始，往下逐层拼。问题是：主控模块要调用下层，可下层还没拼进来，怎么办？**造一个假的顶上**——这个假模块叫 **Stub（桩）**。桩不干实事，只按约定返回一个固定结果，比如"查询用户"的桩永远返回一个写死的用户对象。这样你就能在**下层一行代码都没写**的时候，先把上层的控制逻辑跑起来。

它的好处很明确：**主控逻辑、业务流程、界面导航这些"骨架"能在最早的时候被验证**，而这类逻辑一旦错，返工最大。坏处也很明确：**底层的关键模块被放到最后才测**——数据库访问、算法、第三方对接，这些最容易出问题的地方反而最晚验证；更要命的是，桩返回的是假数据，上层"跑通了"可能只是**假通过**，真实数据一进来就崩。

第二种，自底向上。反着来：从最底层开始，一层层往上拼。这次的问题变成——底层模块写好了，**谁来调用它**？没有上层，你得自己写一个调用者，这个临时写出来的调用者叫 **Driver（驱动模块）**，它负责准备输入、调用被测模块、打印结果。好处是**底层核心功能被最充分地测过**，算法、数据访问这些硬骨头先啃掉；坏处是**主控逻辑、整体流程要到很晚才能验证**，而且**驱动模块要写很多**，这些代码是测试脚手架，不进产品，但工作量实打实。

把两种放一起，你就看出这是一道**对称的取舍题**：自顶向下缺下层，所以要 **Stub**；自底向上缺上层，所以要 **Driver**。这句对应关系是本节最高频的考点，务必刻进脑子：**自顶向下需要桩，自底向上需要驱动**。

还有一个判断口诀给你：**想早看流程，选自顶向下；想早验核心，选自底向上**。如果项目是"界面和流程先给客户看"，用前者；如果是"算法准确性决定成败"，用后者。

> **板书：**左右对照两栏。左栏"自顶向下"：主控 → 逐层向下 → **缺下层 ⇒ 用 Stub（桩）** → 优点：骨架早验证；缺点：底层晚测、假通过风险。右栏"自底向上"：底层 → 逐层向上 → **缺上层 ⇒ 用 Driver（驱动）** → 优点：核心早验证；缺点：流程晚验证、驱动工作量大。中间一行大字："上缺驱动、下缺桩"。

> **提问：**一个项目的核心风险是"利息计算算法必须绝对准确"，界面下个月才定稿。你选自顶向下还是自底向上？为什么？

预设回答：

- 答"自顶向下，因为要早做演示"——需求里没提演示，这个理由不成立；而且算法在底层，自顶向下会把它排到最后。

- 答"自底向上，先把算法啃下来"——正确。判据是"**风险最大的模块在哪一层**"，在底层就自底向上。

- 答"两个都要"——这其实就是下一页的混合策略，可以顺着接一句："你说到点子上了，下一页正是它。"

- 收口：选策略的判据只有一条——**哪一层的风险最大，就先到那一层去**。

易错点 / 考点：① **Stub 与 Driver 的对应关系**是必考点，选择题、填空题、简答题都出过：**自顶向下→Stub（模拟被调用的下层），自底向上→Driver（模拟调用它的上层）**；② 常见错误：把 Stub 说成"模拟上层"——记住桩是**被调用方**，驱动是**调用方**；③ 优缺点要能各说两条；④ 陷阱题："自顶向下不需要写任何辅助代码"——错，桩也是要写的，只是比驱动轻。

过渡：既然两种各有一半好处，那能不能两头一起上？能——这就是混合策略。

【5.78】单体架构的集成测试 –混合策略（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

混合策略，也叫**三明治策略**。做法很直白：**上层用自顶向下，下层用自底向上，两边往中间推进，最后在中间层汇合**。

为什么这样能省事？因为两种策略的辅助代码是可以省下来的。自顶向下越往下走，桩要造得越多、越复杂；自底向上越往上走，驱动要写得越多。而一旦分成两半，**上层只需要造到某个深度，下层只需要写到某个高度**，中间那一段直接对接——**桩和驱动的总量都下来了**。这是混合策略最实在的收益：省的不是思路，是**脚手架的工作量**。

第二个好处是**风险优先**。分层的时候不是对半切，而是按"哪一层最关键"来切：把**核心模块所在的那一层**作为优先测试的起点，让它尽早在真实代码（而不是桩）上被验证。这就是常说的"关键模块优先"。

第三个好处是**能并行**。上层一个小组、下层一个小组，两边同时推进，最后在中间对接口。对进度紧的项目，这是实打实的收益。

但它有代价，也要说清。第一，**分层要先想清楚**——哪些层归上、哪些层归下、中间那一层谁负责，这个划

分如果拍脑袋，最后会在中间打成一团。第二，**中间层压力最大**——它是两边的汇合点，两边的假设都在这儿对撞，所以中间层的接口定义必须**最早定、最稳**。第三，管理成本高一些，两套推进节奏要协调。

最后必须点一句反面教材：还有一种做法叫**一次性集成**，也叫"大爆炸"，就是所有模块写完，一次性拼起来跑。它的问题不是"一定失败"，而是**失败了不知道怎么定位**——几十个模块同时引入，报一个错出来，你连是哪个模块的错都判断不了。所以它只在极小系统里勉强能用，工程上一般不推荐。

板书：画一个三层方块图：上层标"自顶向下（Stub 逐渐减少）"，下层标"自底向上（Driver 逐渐减少）"，中间画一条粗线标"**汇合层：接口最早定、最稳**"。右侧三条收益：省脚手架／风险优先／可并行；左侧两条代价：分层要先想清／中间层压力大。最下面一行写"一次性集成（大爆炸）＝不推荐：错了定位不了"。

提问：混合策略里，"中间汇合层"应该选在系统的哪一层？选错了会怎样？

预设回答：

- 答"选在正中间，上下各一半"——形式上对，但工程上更该按"**接口最稳定、变更最少**"的那一层来切，而不是按层数平分。

- 答"选在业务逻辑层"——常见的合理答案：因为业务层接口相对稳定，上下两边的数据访问和界面都能各自推进。

- 追问：选错的后果是什么？答：汇合层如果是个**天天变的接口**，两边刚对上就又变了，等于白干——所以混合策略的前提是**架构分层清晰、接口相对稳定**。

易错点 / 考点：① 混合策略又叫**三明治策略**，别名要会；② 它的三个收益（省辅助代码、风险优先、可并行）与两个代价要能成对答出；③ 常考对比题："三种策略怎么选"——答"看风险在哪层、看接口稳不稳、看进度要不要并行"；④ **一次性集成（大爆炸）** 一般作为反面选项出现，判"推荐"即错。

过渡：单体的问题解决了，可现在的系统很少是单体。服务拆开以后，集成测试的难度不是一个量级——下一页我们看微服务。

【5.79】微服务架构（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

先建立一个基本认识：**微服务不是"把单体拆小"，而是把"部署单元"和"团队边界"对齐**。一个服务对应一个

业务能力，自己一个进程、自己一个数据库、自己一条发布流水线。听起来很美，可对测试来说，这是一次**结构性升级**。

升级在哪？在单体里，模块之间是**方法调用**——传的是对象，编译器帮你检查类型，同一个进程，出错了堆栈一路到底。在微服务里，模块之间是**网络调用**——传的是字节流，序列化成 JSON 或 Protobuf，跨进程、跨机器、可能跨语言。**方法调用变成网络调用，出错的方式一下多出好几个量级**：网络会超时、会重试、会乱序；对方会重启、会限流、会版本不一致；序列化会因为一个字段可空性不同而在半夜炸掉。

这带来一个很现实的测试困境：单元测试里，你把依赖都 Mock 掉，测得很舒服，可 **Mock 掉的东西正是最容易出问题的地方**；想端到端测，你得把 A、B、C 三个服务、两个数据库、一个消息队列都拉起来，还得保证它们是"配套的版本"——这个成本，很多团队根本承担不起，于是**联调变成了项目末期的一次大型赌博**。

所以微服务的测试策略要变：**测试金字塔不再往上堆，而是往两边分**——单元测试守住每个服务内部，**接口与契约测试守住服务之间的边界**，端到端只留极少数最关键的场景做冒烟。这一页先建立这个判断，下两页讲具体怎么做：先看"通信与数据"这两类集成测试，再看契约测试。

记住一句话：**微服务的测试难点不在"测得多深"，而在"边界怎么守住"**。

> **板书：**左右对比。左"单体：方法调用"——同进程、对象传参、编译期检查、堆栈连贯；右"微服务：网络调用"——跨进程、序列化、超时重试、版本漂移、部分失败。下面画一个变形的金字塔：底层单元测试（守服务内）→ 中层**契约／接口测试（守边界）**→ 顶层极少量端到端（冒烟）。右下角写"难点不在测多深，在守边界"。

提问：单体应用里，"调用一个方法失败"和微服务里"调用一个接口失败"，最大的区别是什么？

预设回答：

- 答"微服务会超时"——是其中一种，但更本质的是**部分失败**：单体里一个方法炸了，整个请求就失败了，逻辑很干脆；微服务里 A 成功了、B 超时了、C 其实成功了只是响应丢了，系统处在一个**不确定的中间状态**。

- 答"网络不可靠"——对，而且不可靠会连带出重试、幂等、补偿这一整套设计，这些都是**测试要覆盖的新场景**。

- 追问：那单元测试能不能覆盖这些？答：不能，因为单元测试里网络根本不存在——**只有集成测试能碰到这类问题**。

易错点 / 考点：① 微服务集成测试的对象是**通信与数据边界**，不是"每个服务内部的业务逻辑"；② 常考判断："微服务架构下应该加强端到端测试"——错，主流做法是**减少端到端、加强契约与接口测试**；③ "部分失

败"是微服务测试的核心概念，答题时能写出这个词，往往就是得分点。

过渡：边界有两个面：一个是"调用通不通"，一个是"数据存得对不对"。我们分别看。

【5.80】微服务架构的集成测试（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页给出微服务集成测试的两个主攻方向，课件写得很克制，但每句话背后都有血。

第一个方向：测试外部的通信，确认通信是否通畅。注意它的定位——**检查核心功能即可**，它不是让你把业务逻辑再测一遍，而是专盯**协议层次的错误**。课件列了三类典型：**丢失 HTTP 报头、SSL 使用错误、请求／响应不匹配**。

这三类为什么值得单列？因为它们有一个共同特点：**在单元测试里绝对测不出来，在业务测试里又容易被业务错误掩盖**。丢报头，最常见的是认证头、内容类型头、追踪头，开发本地用 Postman 手测时手动带着，一进代码就忘了加；SSL 用错，是证书校验被关掉、或者双向认证只配了一半，本地 HTTP 跑得好好的，一上 HTTPS 全线 500；请求／响应不匹配，是字段名大小写、时间格式、枚举值、可空性——最容易发生在"两个团队各自按自己的理解实现了同一个接口"。

第二个方向：数据库访问测试。它的目标也有三条：**确保微服务使用的数据结构与数据库相符、检查 ORM 中设置的映射关系、数据库访问模块能否妥善处理网络出错**。ORM 就是对象关系映射，把表和对象对起来。这三条其实是在说：**代码里的对象模型和数据库里的表结构之间，有一层翻译，翻译错了不会报编译错误，只会在运行时给出错数据**。

举个最容易踩的例子：代码里一个字段是可空的，数据库里那一列却是 NOT NULL；或者枚举在代码里是字符串，在库里存的是整数，某一方加了新枚举值，另一方不认识。这类问题单元测试永远发现不了，因为它用的是内存里的假数据；只有真的连上库、真的走一次 ORM 映射，才会暴露。第三条"网络出错的处理"更微妙——数据库连不上时，你的服务是快速失败、还是把请求挂在那儿把连接池耗光，这属于**韧性**问题，必须靠集成测试来验证。

把两个方向合起来看，你会发现它们的共同逻辑：**集成测试盯的是"翻译层"**——协议翻译（对象 → 字节流）和数据翻译（对象 → 表）。业务逻辑对不对，单元测试管；翻译对不对，集成测试管。

板书：两栏。左"外部通信"：目标＝通信是否通畅、只查核心功能；典型错误＝**丢失 HTTP 报头／SSL 使用**

> **错误／请求-响应不匹配**。右"数据库访问"：目标＝数据结构与库相符、**ORM 映射**正确、网络出错能妥善处理。下方一句总结："集成测试盯**翻译层**：协议翻译 + 数据翻译"。

提问：一个服务在开发环境一切正常，一上测试环境所有请求都 401。你第一个怀疑什么？

预设回答：

- 答"账号密码错了"——先排除，但环境间最常见的其实是**报头没带**：开发环境的网关可能默认注入了认证头，测试环境没配。

- 答"证书或 HTTPS 配置问题"——很可能，尤其是"本地 HTTP、环境 HTTPS"这种差异，SSL 相关错误经常表现为认证失败。

- 答"接口版本不一致"——也对，特别是请求／响应字段不匹配时，有的框架会直接返回 4xx。

- 收口：三种答案恰好对应课件列的三类协议层错误——**这页的三条不是随便列的，是踩出来的**。

易错点 / 考点：① 两个方向要能默写并各举一例，案例题高频；② 常见错误：把"数据库访问测试"理解成"测 SQL 性能"——它测的是**映射与容错**；③ ORM 的全称 **Object Relational Mapping** 要会写；④ 陷阱："通信测试要把业务功能全部重测一遍"——错，课件明确"检查核心功能即可"。

> **过渡：**通信和数据都盯住了，可还有一个更麻烦的问题没解决——**我根本拉不起对方的服务，怎么测？** 下一页的答案，是本章我最想让你记住的一个方法。

【5.81】微服务架构的集成测试 – 续（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页引出一个方法：**消费者驱动的契约测试**，英文缩写 **CDC**，全称 Consumer Driven Contract Testing。我先把"契约"这个词说准。
>
> 课件给的定义是：**契约规定的是接口的调用者（Consumer，消费者）和被调用者（Provider，提供者）之间约定的 Request 和 Response 数据交互格式**。请注意它规定的是**格式**，不是行为、不是性能、不是业务规则。它回答的问题非常具体：我发给你什么字段、什么类型、什么必填；你还给我什么字段、什么类型、什么情况下返回什么状态码。**就这些，不多不少**。
>
> 那为什么叫"消费者驱动"？因为**契约是从消费者的需求出发写的**。课件写得很清楚：**从 Consumer 的需求出**

发设计测试用例并产生一份契约，然后验证 provider 端的功能。这个顺序非常关键——不是提供者说"我提供这些字段"，而是消费者说"我只需要这些字段、我依赖这些字段"。

这个方向为什么重要？因为提供者天然倾向于"多提供"：多给字段、多加参数，反正不影响自己。可对消费者来说，**多给不是问题，少给和改给才是灾难**。所以契约必须由消费者的真实需求来定义——**我只承诺我用到的，你就不能动这些**。

课件还提了"另一种观点：用契约测试或协议测试来做集成测试"。这句话的意思是，在微服务下，**集成测试的重心可以从"把系统拼起来跑"转移到"验证边界上的约定"**。为什么这是一种合理的观点？因为集成测试要发现的问题，八成就在边界上；而边界上的问题，本质是"两边理解不一致"。既然是不一致，那**把两边的理解显式写下来、再各自验证**，就比"拼起来跑一遍看运气"精准得多。

最后说清它不是什么。契约测试**不能替代**单元测试，也**不能替代**端到端测试。它解决的是"服务之间的约定有没有被遵守"，解决不了"业务算得对不对"。它的定位是**金字塔中间那一层**，把最难搞的跨团队协作，变成两份可以自动跑的测试。

> **板书：**中间写"契约 = Consumer ↔ Provider 约定的 **Request / Response 格式**"。下方三步箭头：① Consumer 需求 → 设计用例 ② **产生一份契约** ③ 验证 Provider。左侧写"消费者驱动＝我只承诺我用到的"；右侧写"边界观点：集成重心从'拼起来跑'转到'验证约定'"。底下一行："契约测试≠单元测试，≠端到端测试"。

提问：为什么契约要由消费者来写，而不是让提供者把自己支持的字段全列出来？

预设回答：

- 答"消费者更清楚自己用什么"——对。提供者列全量字段，看着更"完整"，但没人能保证消费者真的只用这些，也没人能保证提供者以后不改。

- 答"提供者写会更准"——反例很明显：提供者写"我支持 name 字段"，可消费者其实依赖的是 `user_name`，两边一跑就炸，而契约测试如果按提供者的说法写，反而会**给出错误的通过**。

- 追问：这体现了一个什么原则？答：**接口的约束应该由使用方定义**——和"需求由用户定义"是同一个道理。

易错点 / 考点：① CDC 的中英文全称必背：**Consumer Driven Contract Testing／消费者驱动的契约测试**；② 契约规定的是**格式**（Request／Response），不是业务逻辑，这是最爱考的一句话；③ 顺序不能错：**先消费者需求 → 再生成契约 → 最后验证提供者**；④ 常考简答："契约测试能不能替代集成测试"——答"能替代其中验证接口约定那部分，不能替代端到端业务验证"。

> **过渡：**这个方法到底能带来多少好处？课件用三条把它说透了。

【5.82】契约测试及其优势、工具（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是三条优势加四个工具，我们一条一条落地。

第一条，降低服务集成的难度，把它分解为单元测试和接口测试。这句话的分量在于"分解"两个字。原来微服务集成是一个巨大的、不可控的任务：要把 N 个服务拼起来。契约测试把它切成两半——消费者这一侧，用假的服务端跑自己的单元测试；提供者那一侧，用契约回放跑接口测试。**一个 N×M 的联调矩阵，被拆成了 N 加 M 两份独立的活**。这就是它最漂亮的地方：不是让集成变简单了，而是**让集成不再需要"同时在场"**。

第二条，团队能以一种离线的方式完成验证，把联调的成本几乎降到零。"离线"是关键词。传统联调要占环境、要约时间、要人等：前端等后端发版，后端等前端确认，一个字段改三天，两边都卡着。契约测试里，消费者拿着契约自己就能验证，提供者拿着契约自己也能验证，**谁都不用等谁**。课件用"几乎降到零"这个说法，指的就是这部分**等待与协调的成本**。你可以回想上一页我提的那个问题——"联调卡了几天、卡在谁等谁"，契约测试正是冲着这个去的。

第三条，基于契约让接口的变更有迹可循。以前接口改了，怎么发现影响面？靠人记、靠群里喊一句"我把字段改名了啊"。现在契约是一份**受版本管理的文件**，改了它就得跑验证，跑不过就是有消费者会挂。于是"接口变更"从一个**口头事件**变成了一次**可审计的动作**，影响面是查出来的，不是猜出来的。

工具方面，课件列的是 **Janus、Pact、Pacto、Spring Cloud Contract** 这四个框架（课件这里写的是 CDT，实际指的就是 CDC 工具）。其中 **Pact** 是社区里最流行的一套，文档在 docs.pact.io，后面几页的示例都是它；**Spring Cloud Contract** 是 Spring 生态的方案，测试用 Groovy 或 YAML 写契约，和你后面实验里的 Spring 项目更贴近。工具会换，原理不变：**写契约 → 生成桩 → 双向验证**。

> **板书：**三条优势竖排：① **分解**：N×M → N＋M（消费者跑单测、提供者跑接口测试）② **离线**：不用同时在场，联调成本≈0 ③ **留痕**：契约进版本库，接口变更可审计。右侧写工具：Janus、**Pact**（docs.pact.io）、Pacto、Spring Cloud Contract。底部一行口诀："**分解—离线—留痕**"。

> **提问：**假设提供者把响应里的 `userId` 改名成 `id`，但功能完全没变、测试也全绿。契约测试能不能拦住这个问题？

预设回答：

- 答"能，因为契约里写了字段名"——对，这正是第三条第优势的落点：字段名是契约的一部分，改名就是破坏契约。

- 答"不能，改了名但值一样"——要纠正：对消费者来说，**字段名就是接口**，`userId` 读不到就是 null，后面一连串空指针，功能上根本不是"没变"。

- 追问：那提供者怎么安全地改名？答：**先加新字段、两边并行一段时间、消费者迁完再删旧字段**——这叫契约的兼容演进，也是契约测试能支撑的实践。

易错点 / 考点： ① 三条优势要能默写，尤其"**分解为单元测试和接口测试**"和"**离线验证**"这两个提法；② 工具名四个要记全，Pact 与 Spring Cloud Contract 最常考；③ 常见错误：认为契约测试是"提供者写契约、消费者遵守"——方向反了；④ 判断题："契约测试能发现业务逻辑错误"——错，它只保证约定被遵守。

【思政落点 12】：契约这两个字，在工程上和生活里是同一个意思——**我说到做到，你按我说的准备**。契约被遵守，才有"离线验证"：因为我相信你会按约定返回，我才敢不等你。反过来，接口悄悄改名、字段悄悄变可空，看似只少一行通知，实际是**把成本转移给了所有信任你的人**。写接口时请记住：**你发布的每个字段，都是对别人的承诺**。守约、留痕、变更前通知——这是技术规范，也是职业信用。

过渡：说完了道理，我们看它具体怎么跑起来。Pact 的官方文档给了一条很清晰的三段流程。

【5.83】示例：CDC 测试框架（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页给的是 Pact 的示例，地址是 docs.pact.io。我不逐图念，我们把它抽象成一条**三段流水**，你记住这条线，换任何 CDC 工具都通用。

第一段，消费者端产出契约。消费者写测试的时候，不连真实的服务端，而是让 Pact 起一个**本地的假服务端**（mock server），你告诉它"我发这个请求，你应该回这个响应"。测试跑的时候，请求打到假服务端上，消费者代码被完整地跑了一遍；跑完，Pact 把这次交互**记录下来**，写成一份 JSON 文件——**这份 JSON 就是契约**。

第二段，契约被送到一个中枢。这份 JSON 被发布到一个叫 **Broker（契约仓库）** 的地方，带上版本号。有了这个中枢，提供者才知道"现在有哪些消费者、依赖我的哪些行为、依赖的是哪个版本"。

第三段，提供者端验证契约。提供者从 Broker 拉取契约，启动自己的真实服务，**按照契约里记录的请求真的发一次**，然后拿真实响应和契约里约定的响应比对——字段、类型、状态码，一项一项对。全对，这条契约就是"已验证"；有一项不对，构建就红了，而且是**在提供者的流水线里红**，不是等联调时才发现。

三段跑完之后还有一步很关键：Broker 上有一个"**能不能部署**"的查询接口，部署前问一句"我要上的这个版本，它的契约都验证通过了吗"，通过了才允许上线。这一步把契约测试**接进了发布流程**，也就接上了我们后面要讲的 CI/CD。

顺便说一个术语口径：课件这里写的是"CDT 测试框架"，规范写法是 **CDC（Consumer Driven Contract）**，答题以 CDC 为准。

板书：三段流水横排：① **Consumer 测试**：Pact 起 mock server → 跑消费者代码 → 生成 **契约 JSON** ② **发布到 Broker**（带版本）③ **Provider 验证**：拉契约 → 真发请求 → 比对响应 → 红／绿。第四格写"部署前问 Broker：这个版本能上吗"。右上角注："课件写 CDT，规范为 **CDC**"。

提问：如果契约一直是绿的，但上线后还是出了对接问题，最可能是哪一环漏了？

预设回答：

- 答"契约写得不全"——最常见。契约只覆盖了消费者**已经实现的**调用，新写的调用还没进契约，自然测不到。
- 答"提供者验证用的不是生产版本"——很关键，验证必须跑在**准备发布的那个构建**上，否则验的是另一个东西。
- 答"契约没进发布门禁"——最要命的一种：契约测试跑是跑了，但没人卡它，红了也照发。
- 收口：所以判断契约测试有没有真落地，看三件事：**契约覆盖了多少调用、验证跑在哪个构建上、红了能不能拦住发布**。

易错点 / 考点：① 三段流程要能复述（消费者产契约 → Broker 中转 → 提供者验证）；② 术语：Broker＝契约仓库，mock server＝消费者端的假服务端，两者别混；③ 口径提醒：**CDC 不是 CDT**；④ 常考判断："契约测试要求消费者和提供者在同一时间联调"——错，恰恰相反。

过渡：三段里的第一段最容易被误解——消费者到底在测什么？下一页专门讲。

【5.84】Consumer 测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

消费者测试，核心道具是 **Stub server**，也就是桩服务端。

我们要搞清楚它在测谁。**它测的是消费者自己**——我这个客户端代码，在拿到符合约定的响应之后，能不能正确解析、正确转换、正确展示。它**不测**提供者，因为对面是个假的。

那假服务端怎么知道该回什么？靠你在测试里**声明期望**。你写"当我 GET 这个路径、带这个头时，请回 200 和这段 JSON"，Pact 就把这个期望变成行为：**真的起一个 HTTP 服务，监听一个端口，按你声明的规则应答**。所以它叫 Stub server 而不是 Mock 对象——它是**进程内的一个真服务端**，走的是真的 HTTP 协议栈。这一点很重要：因为走了真协议，序列化、报头、状态码这些"协议层的问题"才有机会暴露，而如果只是一个方法级的 Mock，这些全被绕过了。

测试跑起来的顺序是这样：先声明期望 → 起假服务端 → 消费者代码发真请求 → 假服务端按期望应答 → 断言消费者解析结果对不对 → **最后把这次交互写成契约**。

请注意最后这一步的分量：**契约是测试的副产品，不是手写的文档**。这是 CDC 最聪明的地方——文档会过期，测试不会。你改一行消费者代码，跑一次测试，契约自动更新；忘了更新？测试就红。

还有一个实践细节：消费者测试里应该**多造异常场景**。返回 500 怎么办、返回空数组怎么办、返回字段类型和约定不一致怎么办、超时怎么办。因为对面是假服务端，**造这些场景的成本几乎为零**——这在真实联调里是做不到的，你没法让别人的服务专门给你返回一个 503。

> **板书：**中间画"消费者代码 → 真 HTTP 请求 → **Stub server**（按声明的期望应答）"。下方流程：声明期望 → 起服务 → 发请求 → 断言解析 → **生成契约**。右侧两条批注：① 测的是消费者自己，不测提供者；② **契约是测试的副产品，不是手写文档**。底部："异常场景在这里最便宜：500／空数组／类型不符／超时"。

> **提问：**消费者测试里，如果我把期望写错了——比如实际接口返回的是字符串、我期望成数字——测试会不会红？

> **预设回答：**

- 答"不会红，因为假服务端按我写的回"——完全正确，这是这个方法的**固有盲区**：消费者测试只能证明"我能处理我以为的响应"，不能证明"真实响应就是我以为什么样"。

- 答"会红，因为跟真实服务对不上"——那就跑到提供者验证阶段了，消费者这一侧不会红。

- 追问：那这个盲区由谁补上？答：**提供者验证**——它拿真实服务去对契约，一旦真实响应和契约不符，构建就红。**两端互补，才闭环**。

易错点 / 考点：① Stub server 是**真起一个 HTTP 服务**，不是方法级 Mock，这个区别常考；② 消费者测试测的是消费者，别答成"测接口对不对"；③ 契约由测试**自动生成**，这是 CDC 与"手写接口文档"的本质差别；④ 盲区问题："消费者测试全绿能说明接口没问题吗"——不能，要靠提供者验证补。

过渡：那就看另一端——提供者怎么拿契约反过来验自己。

【5.85】Provider 测试（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

提供者测试，核心道具是 **contract verifier**（契约验证器）。课件给了它的仓库地址，是 pact-foundation 的 pact-provider-verifier。

它干的事和消费者正好相反：**从 Broker 把契约拉下来，然后拿真实的服务去"应考"**。具体流程：把契约里记录的每一条请求，**原样发给真实运行的提供者**；拿到真实响应后，和契约里约定的响应**逐项比对**——状态码、报头、响应体的字段和类型。全对就通过，有一项不对就失败，并且会明确告诉你"是哪条契约、哪个字段、期望什么、实际什么"。

课件这一页还写了几个工程上的关键点。**Contract verifier tests** 指的是"由契约驱动的验证测试"，它不是手写的用例，而是**由契约自动生成**的——这就是那个 **auto-generate**。你在图上还能看到 `Contract.groovy` 和 `Call API`：前者是契约（Spring Cloud Contract 里常用 Groovy DSL 写），后者是验证器真正发出调用的那一步。

这里有个特别值得强调的设计：**提供者不需要知道消费者是谁、用什么语言写的**。它只需要读契约 JSON，发 HTTP 请求，比响应。所以一个 Java 服务可以被一个前端、一个 Python 脚本、一个 Go 服务同时"契约验证"，**跨语言是天然的**——这也是它比"造 Stub"高明的地方，Stub 是每种语言各写一套，契约是**一份文件所有人共用**。

最后提醒一个容易忽略的前提：提供者测试必须跑在**你准备发布的那个构建**上，而且**依赖要真实**（真数据库、真配置）。如果验证时又把它 Mock 了一层，那验的还是假的，等于白跑。

板书：流程横排：**Broker 拉契约 → 逐条回放请求 → 打真实 Provider → 比对响应（状态码／报头／字段类型）→ 红／绿**。旁注三个词：contract verifier、auto-generate（契约自动生成用例）、`Contract.groovy`（契约 DSL）＋`Call API`（真发调用）。右下角："跨语言天然：一份 JSON，谁都能

验"；"必须跑在待发布构建上、依赖要真"。

提问：提供者验证失败了，是提供者的错，还是消费者把契约写错了？

预设回答：

- 答"提供者的错"——多数情况是，提供者改了行为没通知。

- 答"可能是消费者写错了"——也存在：消费者按自己的想象声明了期望，而真实接口从来不是那样。这时要判断**谁是对的**：如果契约反映的是合理的消费需求，那提供者要改；如果契约写错了，就改契约。

- 追问：那怎么减少这种争议？答：**契约评审**——把契约当接口文档一样评审，这正是它比口头约定强的地方：**争议有实物可依**。

易错点 / 考点：① 两端对应关系必考：**Consumer 端用 Stub server 生成契约，Provider 端用 contract verifier 验证契约**；② 提供者验证的用例是**自动生成**的，不是手写；③ 常见错误："提供者测试需要启动所有消费者"——错，只需要契约；④ 判断题："契约测试要求两端同语言"——错，跨语言是它的优势。

过渡：两端都讲完了，我们把它们放在一张图上对照，你就能看清这条闭环。

【5.86】示例（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这张图把两端放在一起，标了两个身份：**Server side 是 service provider（服务提供者）**，**Client side 是 service consumer（服务消费者）**。我们顺着看这条闭环是怎么合上的。

左侧是客户端。它在自己的测试里，面对的是一个按契约应答的假服务端，测的是自己的解析和业务处理；测试跑完，产出一份契约文件。这份文件被发布出去，进入共享的契约库。

右侧是服务端。它从契约库拿到同一份文件，用验证器把每条请求打到真实服务上，比对响应。通过了，说明"我仍然满足所有消费者的期待"；不通过，说明"我这次改动破坏了某人"。

这张图最重要的信息，不是流程，而是**那份文件在中间**。请注意：**消费者和提供者之间，传递的不是代码、不是环境、也不是人，而是一份可执行的文件**。这就是 CDC 的本质——**把协作从"人对人沟通"变成"文件对文件验证"**。

再对照一下我们前面讲的单体策略，你会有个顿悟：单体里的 Stub 是**每个调用方各写一个假模块**，微服务里

的契约是**一份共享的约定**。单体里 N 个模块要拼，就有 N 份桩；微服务里 N 个消费者对 1 个提供者，只需要 N 条契约，而且**提供者一次验证全部搞定**。所以 CDC 不是凭空发明的新东西，**它是把"桩"这个老思路，从"各自造假的"升级成了"共享约定的"**。

这也解释了为什么它能"把联调成本降到零"：因为成本的大头从来不是写假服务，而是**协调**——约时间、对版本、等人回消息。契约把这部分成本直接删掉了。

> **板书：**左右两端：**Client side / Consumer**（Stub server → 生成契约）｜**Server side / Provider**（contract verifier → 回放验证）。中间画一个文件图标写"**契约（JSON）**"，箭头从消费者指向契约、从契约指向提供者。下方对比一行："单体 Stub：各写各的假模块；CDC：一份共享约定，提供者一次验证全部"。

提问：为什么说契约测试是把"桩"的思路升级了，而不是发明了一个新东西？

预设回答：

- 答"都是拿假的顶替真的"——有一半对。差别在于：Stub 的"假"是**每个调用方私有的一份想象**，契约的"假"是**双方共同确认的一份约定**，而且能被真实服务反向验证。

- 答"因为契约能自动验证"——这是升级的关键动作：**从"假设对方会这样"变成"验证对方确实这样"**。

- 追问：那这个升级解决了 Stub 的什么老毛病？答：**漂移**——Stub 写完之后，对方改了，你的假模块还在按老样子应答，测试全绿、上线全炸。契约一旦漂移，提供者那边立刻红。

易错点 / 考点：① 两个角色的英文要会对应：**provider＝提供者＝server side，consumer＝消费者＝client side**，考试常故意对调；② 这张图的标准问法是"简述 CDC 的完整闭环"，答题要点必须包含"契约在中间传递"；③ 常考对比："Stub 与契约的区别"——答"私有想象 vs 共享约定 + 可被真实服务反向验证"。

过渡：契约要写，可字段那么多，总不能全靠手敲。工程上有个现成的来源——接口文档。下一页看它。

【5.87】如何获取接口信息（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲的是工具链上的一个衔接点：**接口信息从哪儿来**。课件给的是 **Swagger 生成的动态接口文档**示例。

什么叫"动态"？就是文档不是人写的，是**从代码里生成的**。你在 Controller 的方法上打注解，说明这个接口

的路径、参数、返回类型，工具扫描一遍，就产出一份结构化的接口描述——现在通行的标准叫**OpenAPI**，Swagger 是这套东西最常用的工具链。所以它天然"跟代码同步"：代码改了，文档跟着变，不会出现"文档标 v1、代码跑 v2"这种经典事故。

它和契约是什么关系？我的理解是：**文档描述"我提供什么"，契约规定"你依赖什么"**。这两者不一样。文档是提供者视角的全集，契约是消费者视角的子集。但文档是**写契约最好的原料**——你从文档里把"我这个消费者真正用到的那几个字段"摘出来，就是一份契约的初稿。

反过来，文档还能省掉一大半客户端代码。有了结构化的接口描述，工具可以**自动生成客户端 SDK 和桩服务**：类型定义、序列化、请求方法，全是生成出来的。所以你会看到一个很顺的链路：**注解 → OpenAPI 文档 → 生成客户端／桩 → 写契约 → 两端验证**。这条链上每一步都是自动的，人只负责"确认需求"。

但也要提醒一句：**别把文档当契约用**。文档说"这个字段可能为空"，你的代码按非空处理，文档不会报错，运行时会给你一个空指针。**文档是说明书，契约是考试卷**——说明书可以写得很全，考试卷只考你真用到的那几道题，但考不过就是考不过。

板书：链路横排：**代码注解 → Swagger／OpenAPI 文档（动态生成）→ 生成客户端 SDK／桩服务 → 摘出消费者依赖写成契约 → 两端验证**。中间写一句对比："文档＝提供者视角的全集（说明书）；契约＝消费者视角的子集（考试卷）"。

提问：有了自动生成的接口文档，是不是就不用写契约了？

预设回答：

- 答"不用了，文档已经很全"——常见误解。文档只**声明**，不**验证**；它不会因为提供者改坏了就变红。

- 答"要写，文档只是原料"——对。而且要从文档里只挑**消费者真正依赖的字段**，多挑会让契约过于脆弱，提供者改任何东西你都红。

- 追问：那契约写多细合适？答：**只写你真正依赖的**——这是"消费者驱动"的精髓，也是控制契约维护成本的关键。

易错点 / 考点：① Swagger 文档是**从代码动态生成**的，不是手写，这是选择题常考点；② OpenAPI 是规范名，Swagger 是工具链名，别混答；③ 核心判断："文档 ≠ 契约"——文档声明、契约验证，这是简答题的得分点；④ 契约粒度原则："只写真正依赖的字段"。

过渡：最后一种情况最常见：你要测的服务 A，依赖 B 和 C，可 B、C 你既拉不起来、又还没写契约。怎么

> 办?

【5.88】利用 Mock 技术解除微服务之间的依赖（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页给的是最朴素、也最常用的一招：**用 Mock 把微服务之间的依赖解除掉**。图上标的很清楚：中间是**微服务 A，标注 SUT**，也就是被测系统；它两侧的 **Service B 和 Service C 都被 Mock 掉了**；把这条链打通之后做的，是 **API Test**——在 A 的接口边界上做测试。
>
> 先讲它解决什么问题。你要测 A 的一个接口，A 要调 B 和 C。B 是别的团队维护的，C 还在开发中，两个都不一定在线。如果非要等它们，你的测试就永远跑不起来。Mock 的办法是：**在 A 的外部依赖点上放一个假的**，A 发请求给它，它按预设返回。于是 A 变成了**一个只依赖自己的单元**，随时能跑、随时能重复跑。
>
> 具体 Mock 什么？课件强调"**Mock 请求和响应信息**"——也就是说，你 Mock 的粒度是**协议层的一对报文**：请求长什么样、回什么样。这跟前面契约里的"交互"是同一个粒度。
>
> 这一招的收益有三条。**第一，测试可控**：你可以让 B 返回 500、返回超时、返回一堆脏数据，然后看 A 怎么处理——这在真实环境里根本做不到。**第二，测试快**：不起 B、不起 C，不连它们各自的库，一个测试从秒级降到毫秒级。**第三，解耦进度**：B 没做好，A 照样能测，团队不用互相等。
>
> 但必须讲清它的风险，否则会误用。**最大的风险是"漂移"**：你 Mock 的报文是照着当时 B 的样子写的，B 后来改了，你的 Mock 不知道，测试继续全绿，上线全炸。**第二个风险是"测了假的"**：如果连 A 自己的数据库、消息队列也一起 Mock 掉，那这个测试就退化成单元测试了，集成测试的意义没了。
>
> 所以正确的用法是：**Mock 只用在"我控制不了的远端"，"我自己这一侧的依赖（库、缓存、MQ）要用真的"**。而防漂移的办法，就是我们前面讲的那一套——**把 Mock 的报文固化成契约，让提供者去验证**。你看，绕了一圈，Mock 和契约其实是配套的：**Mock 让你现在能测，契约让你以后还测得准**。

> **板书：**画图：`Service B（Mock） ／微服务 A（SUT） ／ Service C（Mock）`，A 的两侧箭头标"Mock 请求与响应报文"，A 外面画个框标"**API Test 的边界**"。下方三条收益：可控（能造 500／超时／脏数据）／快（毫秒级）／解耦进度；两条风险：**漂移**（Mock 与真实不一致）、**测了假的**（把自家依赖也 Mock 掉）。底部："Mock 管现在，契约管以后"。

> **提问：**一个团队把所有外部依赖都 Mock 了，测试天天绿，可一上线就出问题。问题出在哪?

> **预设回答：**

- 答"Mock 写错了"——可能，但更常见的是 **Mock 从没跟真实服务对过**，也就是漂移。

- 答"没有一个环节验证真实接口"——这才是根因。缺的不是 Mock，缺的是**提供者侧的契约验证**。

- 答"把自家数据库也 Mock 了"——这个也要点出来：那样测的就不是集成，是单元了，集成测试的价值全部丢失。

- 收口：所以判断一套微服务测试是否可靠，问三句：**Mock 的报文是不是契约、契约有没有在提供者端验证、自家依赖是不是真的**。

> **易错点 / 考点：** ① SUT 的含义（System Under Test，被测系统）要会；② 这一页的典型考法是"用什么方法解除微服务依赖"——答 **Mock（测试替身）**，并说明 Mock 的粒度是请求／响应报文；③ 常见错误：认为 Mock 一多就是好事——**Mock 越多，离真实越远**，边界要卡在"我控制不了的远端"；④ 简答题要点：Mock 的收益（可控／快／解耦）与风险（漂移／失真），以及用契约防漂移。

> **过渡：**到这里，集成的"怎么测"讲完了。最后一个问题：这一切在什么时候发生？答案是——**每次提交都发生**。这就是持续集成。

【5.89】CI/CD 与持续测试（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这是本章最后一块，也是把前面所有内容串起来的一块：**CI/CD 与持续测试**。
>
> 先问一个问题：前面讲的单元测试、Mock、覆盖率、契约测试，它们都是"好东西"，可为什么很多团队明明都知道，却还是做不好？原因往往不是不会写，而是**没有一个机制让它们每次都发生**。人是有惰性的：这周赶进度，单元测试先不写了；这个版本急，覆盖率先不看。于是这些实践在文档里存在，在流程里消失。
>
> CI/CD 就是那个**强制机制**。它把"跑测试"从一件靠自觉的事，变成一件**流水线上自动发生、不通过就过不去**的事。所以这一块不是在讲运维，是在讲**怎么让测试真正落地**。
>
> 还有一个更根本的理由：测试能力是**会被消耗的**。一个团队的手工测试经验，往往集中在两三个人身上；人一走，经验就断。而流水线上跑的东西是**沉淀下来的资产**：用例、脚本、门禁标准、报告基线，谁都能接手。所以把测试搬进流水线，不只是提速，也是**把个人经验变成团队能力**。

这一块四页：**流水线长什么样**（两次）、**为什么 CD 会倒逼持续测试**、**一个真实的流水线示例**、**持续测试有哪四个特点**，最后一页是实践框架。学完这一块，你应该能把第 5 章所有技术**摆到一条时间轴上**——每项技术在流水线的哪一段跑、跑完卡什么。

带着一个问题听：**你们项目现在，从提交代码到能上线，中间有几步是自动的？**

板书：中间写"CI/CD 与持续测试"，下面四行：① 流水线形态 ② CD 倒逼持续测试 ③ Amazon 示例 ④ 持续测试四特点 + 实践框架。左侧竖写一句："实践靠机制落地，不靠自觉"。

提问：一个团队单元测试写得很规范，但每次发版前才跑一次。这套实践算不算"落地"了？

预设回答：

- 答"算，写了就是落地"——不算。**测试的价值取决于它多久跑一次**：跑得越晚，发现的问题越贵，而且改起来牵扯越多。

- 答"不算，得进流水线每次跑"——对，这正是这一块的主题。

- 追问：那"每次跑"会不会太慢？答：会，所以才有下一页之后要讲的**分层与精准**——不是所有测试都每次都跑全套。

易错点 / 考点：① CI/CD 在测试课里的定位是**测试的执行与门禁载体**，不是运维话题；② 常考概念区分：CI（持续集成）／持续构建／CD（持续交付与持续部署）——四个词的区别是本节高频考点；③ 判断题："持续测试就是自动化测试"——错，持续测试强调**在流水线中持续发生**，范围比自动化测试更大。

过渡：先看这条流水线整体长什么样，你才有地方安放前面学的每一项技术。

【5.90】CD 倒逼持续测试（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是一条链，五个词：**持续集成 → 持续构建 → 持续交付 → 持续部署**，最后落到**持续测试**。我们一个一个说清，因为这是必考的概念辨析。

持续集成（CI）：开发人员频繁地把代码合到主干，每次合并都自动触发编译和测试。它解决的是"集成地狱"——以前大家各写几个月，最后合并时冲突成堆。关键词是**频繁**和**自动**。

持续构建：每次代码变化都自动构建出可运行的制品。差别在于关注点从"代码能不能合"变成"制品能不能

出"。**持续交付**：保证任何时候都有一份**可以随时发布到生产**的制品，但**发布这个动作由人决定**——一键可发，按不按人说了算。**持续部署**：连"按"都不用人，通过所有门禁就自动上生产。所以交付与部署的区别只有一条：**最后那一下是不是自动的**。

现在说"倒逼"。为什么 CD 会**逼出**持续测试？逻辑很硬：当发布频率从"每月一次"变成"每天十次"，**人工回归测试在物理上就不可能完成**。你不可能每天组织十次回归。于是只有两条路：要么放弃质量，要么**让测试自动化、并且快到能跟上发布节奏**。这就是"倒逼"——**不是测试自己想起来要进步，是交付节奏压着它必须进步**。

这个倒逼会连带改变测试的形态。第一，**测试必须分层**：大量快速的单元测试挡住大部分问题，少量慢的端到端测试守住关键路径。第二，**测试必须精准**：不可能每次都全回归，得能算出"这次改动影响了哪些用例"。第三，**测试必须左移**：等代码写完再测，根本来不及，得在编码阶段就把问题挡掉。

反过来看，这也解释了一个常见困惑：为什么有些团队上了 CI 工具，测试还是老样子？因为**他们只是把手工流程自动化了一遍，没有改变测试的形态**。CI 工具不是目的，**它是一面镜子，照出你的测试够不够快、够不够准**。

板书：横排链条：**持续集成（频繁合并＋自动构建测试）→ 持续构建（自动出制品）→ 持续交付（随时可发，人按发布）→ 持续部署（自动上生产）**。链条下方画一个大箭头写"**倒逼**"，指向"持续测试"。再往下三行：分层／精准／左移。右下角辨析："交付 vs 部署 ＝ 最后那一下是否自动"。

提问：一个团队做到了持续部署——每次合并通过门禁就自动上生产。那他们的测试必须要满足什么条件？

预设回答：

- 答"必须全自动"——必要但不充分。全自动但跑两小时，还是跟不上。

- 答"必须快、必须准、必须能拦住"——完整答案：快（分钟级反馈）、准（只跑受影响的）、拦得住（门禁卡死）。

- 答"还得能回滚"——很好的补充，自动部署必须配自动回滚和监控，这是"测试右移"的部分。

- 收口：所以持续部署不是一个部署动作，它要求**整套测试与监控体系同时到位**——这就是"倒逼"的真实含义。

易错点 / 考点：① **持续交付 vs 持续部署**：唯一的区别是"上生产是否自动"，这是最高频的辨析题；② 四个词要能按顺序说清，且能各用一句话定义；③ "倒逼"的逻辑要能讲出来：**发布频率上升 → 人工回归不可行 → 测试必须自动化、分层、精准**；④ 常见错误：把 CI 等同于"用 Jenkins 之类的工具"——工具只是载体。

过渡：理论讲完，我们看一条真实的流水线是怎么排的——先看一个业界示例。

【5.91】CI/CD 流水线（二）环节排布（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页又是一张流水线图。前面那张讲的是"形态"，这张讲的是"排布"——**同样的门，摆放顺序不同，效果差很远**。我给你三条排布原则。
>
> **第一条：便宜的放前面，贵的放后面**。编译几秒钟，静态扫描一两分钟，单元测试几分钟，集成测试十几分钟，端到端半小时以上。这个顺序不是随便排的，它符合一个经济学直觉：**越早失败越省钱**。一个编译错误不该等到跑完半小时的端到端测试才被发现。
>
> **第二条：确定性的放前面，不确定的放后面**。什么叫不确定？依赖外部环境的测试就是不确定的——网络抖动、第三方接口超时、测试数据被别人改了。这类测试放在前面，会让流水线"时红时绿"，团队很快就学会"红了再点一次重跑"，**门禁就此失效**。所以不确定的测试必须往后放，并且要单独统计它的稳定率。
>
> **第三条：只在必要的分支上跑全套**。主干和发布分支跑全套门禁，功能分支只跑快速门禁。这不是降低标准，而是**把有限的门禁预算花在最需要的地方**——毕竟门禁的目的是拦住问题进入主干和生产，不是给每个分支都做体检。
>
> 最后提一句"失败要快"这个工程原则。它的完整说法是 **fail fast**：不仅流水线要尽早失败，**测试本身也要尽早失败**——一个用例里如果有五个断言，第一个就错了，后面四个不该继续跑；一个测试套件里如果连数据库都连不上，就不该傻等着超时。你在写测试时也要有这个意识：**让错误尽早、尽量明确地暴露**。

> **板书：**横向阶梯图，从便宜到贵：`编译（秒） → 静态扫描（分） → 单元测试（分） → 集成／契约（十分钟级） → 端到端（半小时级）`。下面三条原则：① 便宜在前、贵在后（越早失败越省）② 确定性在前、不确定性在后（防"红了重跑"）③ 主干跑全套、分支跑快速。底部写 **fail fast**：用例内早失败、套件内早失败。

> **提问：**团队里流行一句话——"红了不要紧，点一下重跑就绿了"。这句话暴露了流水线的什么问题？

> **预设回答：**

- 答"测试不稳定"——对，但不稳定只是症状。根因往往是**不确定的测试被放在了前面**，或者测试之间共享了状态。

- 答"门禁形同虚设"——更本质。一旦大家习惯"重跑"，门禁就从"拦截"变成"抽奖"，信任崩塌之后，所有红灯都会被忽略。

- 追问：怎么治？答：**隔离不确定测试、统计不稳定率、超过阈值就修或下线**——宁可少一个用例，不要一条会撒谎的流水线。

> **易错点 / 考点：** ① 排布原则要能说三条，尤其"**便宜在前、贵在后**"和"**确定性在前**"；② fail fast 的含义要准确（尽早失败、明确暴露），别答成"快速失败重试"；③ 常考判断："流水线应该把所有测试每次都跑全"——错，成本上不可行，也不必要。

> **过渡：** 原则有了，看一个业界的具体例子——课件用的是 Amazon。

【5.92】CI/CD 流水线（三）完整环节与门禁（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是流水线的完整展开，我按"入口—检查—放行"三段带你看，然后指出三个最容易被忽略的环节。
>
> **第一段，入口——谁触发。**课件列了四种：**代码提交**（最常见，一 push 就跑）、**定时触发**（每天夜里跑一次全量回归）、**事件触发**（比如依赖库升级、镜像更新、上游服务发了新版本）、**人工触发**（指定某次构建、某个分支）。四种触发的意义不同：提交触发保证"每次改动都被检查"；定时触发补上"没改动但环境变了"的情况；事件触发处理"我没改，但世界变了"；人工触发用于特殊验证。**只配提交触发是不够的**，这是很多团队的漏洞。
>
> 入口之后立刻是**排队限流**。为什么需要它？因为提交一密集，构建就会挤在一起抢资源，最后大家一起慢。限流的作用是**保证关键分支优先、防止资源被打爆**。
>
> **第二段，检查。**顺序是：**编译 → 静态扫描（Sonar 增量扫描）→ 安全扫描 → 单元测试 → 自动化测试**。这里有个专业细节值得单独讲：**增量扫描**。全量扫一遍代码库可能几十分钟，增量扫描只扫本次改动涉及的部分，把反馈压到分钟级——这就是前面说的"便宜在前"的具体实现。安全检查放在单元测试**之前**，意思是"有高危漏洞的代码，连测试都不必跑"，这体现的是**风险前置拦截**的思路。
>
> **第三段，放行。质量门禁**是一道总闸——所有检查的结论在这里汇总，达标才放行。放行之后是**部署 STG**（预发布环境）、**灰度环境**（小流量验证）、**生产环境**。注意这个顺序的用意：**每一级都是一次风险放大**，所以在低风险环境里尽可能把问题暴露完。

还有四个环节容易被忽略，我要点出来：**代码跨数据中心同步**（多地构建的一致性）、**编译结果预推送、制品预推送**（把构建产物提前送到各环境，缩短部署等待）、**动态扩容**（构建高峰时临时加机器）。这四个不是"主流程"，但**没有它们，主流程就会卡**。

板书：三行流程。入口：**代码提交／定时／事件／人工 → 排队限流**。检查：**编译 → Sonar 增量扫描（静态）→ 安全扫描 → 单元测试 → 自动化测试**。放行：**质量门禁 → 部署 STG → 灰度 → 生产**。侧边小字：跨数据中心同步／编译结果预推送／制品预推送／动态扩容。右上角批注："安全扫描放在单测前＝风险前置"。

提问：为什么"安全扫描"要放在单元测试**前面**？按"便宜的在前"原则，单元测试可能更快啊。

预设回答：

- 答"因为安全问题更严重"——方向对，但要说得更工程化：**高危漏洞属于"根本不该进入后续流程"的问题**，它的处置成本与测试结果无关，越早拦越省。

- 答"因为扫描快"——不一定，全量扫描很慢，能放前面靠的是**增量扫描**。

- 追问：这说明排布原则不止"便宜在前"一条，还有什么？答：**按"问题的严重性与处置方式"排**——能被立刻判定为"不可接受"的检查，尽量往前放。

易错点 / 考点：① 流水线环节的顺序是**填空与排序题的高发区**，务必记住"编译→静态→安全→单测→自动化→门禁→STG→灰度→生产"这条主干；② 四种触发方式要能说全，尤其**事件触发**最容易被漏；③ "Sonar 增量扫描"是关键词，全量扫描撑不起持续集成；④ 三个易忽略环节（预推送、跨中心同步、动态扩容）常作为多选题出现。

【思政落点 13】：质量门禁最有意思的地方是**它不商量**：单元测试失败就是失败，覆盖率不达标就是不放行，不会因为"这个版本很重要"就网开一面。这恰恰是它有价值的原因——**规则一旦可以为例外让步，就不再是规则，只是一句建议**。同学们以后一定会遇到"这次特殊，先过了吧"的时刻。希望你能守住那个"不特殊"，因为**破例一次就会有第二次**。守规则不是死板，是对交付负责。

过渡：流水线搭好了，那"持续测试"到底要做到什么样才算合格？课件给了四个词。

【5.93】持续测试的特点（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本节最该背下来的一页：持续测试的**四个特点**——足够快、准确且有效、平滑有序、与 CI/CD 集成。我一个一个讲，每个都给你一个"判断标准"，这样你答题和干活都能用。

第一个，足够快。课件写的是"整个测试过程要快，一方面高度自动化测试，另一方面业务端到端的探索式测试"。这句话有两层，很多人只看到第一层。第一层好理解：能自动的都自动。第二层才是精髓：**自动化解决不了的部分，要靠探索式测试去补**——而探索式测试要快，靠的是**人的经验被复用**（会话式测试、检查表、缺陷模式库），而不是靠多派人。所以"快"不是单一的提速，是**机器管确定性、人管不确定性**。

第二个，准确且有效。课件说"被测系统往往很复杂，不可能做全回归测试，而是要推行**精准测试**"。这句话是持续测试的**技术核心**。全回归的成本随系统规模线性上涨，而发布频率也在上涨，两条线一交叉，全回归必然崩。精准测试要解决的是：**这次改动，到底影响哪些用例**——靠代码变更分析、调用链分析、历史缺陷关联，把回归集缩到最小但够用的子集。判断标准很简单：**你能不能说清"这次改动为什么只跑这些用例"**，说不清就是全回归的伪装。

第三个，平滑有序。课件把"有序"解释成三个打通：**测试左移到测试右移、单元测试到系统测试、静态测试到动态测试**。意思是测试不该是各自为政的孤岛，而应是**一条按成本和风险排好序的链**：左移管预防、右移管监控，单元管内部、系统管整体，静态管不运行就能发现的、动态管必须跑起来才能发现的。判断标准：**能不能把所有测试活动画在一条时间轴上，没有断点、没有重叠浪费**。

第四个，与 CI/CD 集成。课件说"与研发的持续构建、持续集成、持续部署、运维等环境的集成"。这一条是前三条的**落地条件**：再快的测试不进流水线就不会被执行；再精准的选择不接构建系统就选不出范围。判断标准：**测试是构建的一部分，而不是构建之后的一个独立动作**。

四个特点其实是一条逻辑链：**快**是前提，**准**是手段，**有序**是结构，**集成**是保障。

板书：四行，每行"特点 + 判断标准"：

① **足够快**：自动化＋端到端探索式 → 判据：机器管确定性、人管不确定性

② **准确且有效**：不做全回归，推行**精准测试** → 判据：能说清"为什么只跑这些用例"

③ **平滑有序**：左移↔右移、单元↔系统、静态↔动态三个打通 → 判据：所有测试能画在一条无断点时间轴上

④ **与 CI/CD 集成**：接持续构建／集成／部署／运维 → 判据：测试是构建的一部分

底部一条链：快（前提）→ 准（手段）→ 有序（结构）→ 集成（保障）。

提问："我们每次都跑全量回归，所以我们的测试很充分。"这句话在持续测试的语境下有什么问题？

预设回答：

- 答"太慢了，跟不上发布节奏"——对，这是最直接的后果：全回归跑不动，最后一定演变成"发版前跑一次"，持续测试就没了。

- 答"全回归不代表充分"——也对，跑得多不等于测得准，很多缺陷在没被设计的用例里。

- 追问：那精准测试是不是"少测"？答：不是少测，是**把力气放在受影响的地方**——精准的前提是"能证明影响范围"，证明不了就不能缩。

- 收口：所以"全回归"和"精准测试"的差别不在数量，在**有没有依据**。

易错点 / 考点： ① 四个特点的名称与解释是**简答／论述题的高频考点**，要能各配一句原文式的解释；② "精准测试"是关键词，答"减少测试用例"是不准确的，要答"**基于变更影响分析选择用例**"；③ "平滑有序"的三个打通要能背：左移—右移、单元—系统、静态—动态；④ 常见错误：把"持续测试"理解成"一直不停地跑测试"——它是**在流水线中按节奏、分层、有选择地跑**。

过渡： 四个特点讲的是"应该做到什么"，那"怎么一步步做到"？业界有一份专门的白皮书。

【5.94】持续测试实践框架（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页指向一份资料：**《持续测试白皮书》**，课件给了扫码下载的入口。我不逐条念它，讲清楚"为什么需要一份框架"更有用。

因为持续测试这件事，最难的从来不是"会不会写自动化脚本"，而是**怎么让一整个组织持续地做下去**。你只讲技术，会得到一个尴尬的结果：几个人写了一批脚本，半年后没人维护，全部失效。所以成熟的做法一定是**多维度同时推进**——通常包括这么几块：**组织与角色**（谁负责测试资产、谁维护流水线）、**流程与规范**（什么改动跑什么测试、门禁标准是什么）、**技术与工具**（分层框架、环境与数据管理）、**度量与改进**（反馈时间、逃逸缺陷率、用例稳定率）。这四块缺一块，另外三块都会退化。

这份框架对你的直接价值是：**给你一张"能力清单"**。你以后进公司，如果想推动持续测试，不要一上来就谈工具，先拿这张清单对一遍——我们现在缺的是哪一块？缺组织，就先定责任；缺度量，就先把反馈时间和逃逸率统计起来。**先能量化，才能改进**。

顺便说一句：这类白皮书的价值不在"读完"，在**当检查表用**。读一遍记不住，但拿着它一条条对现状，能立刻发现缺口。

再说清一个边界：白皮书是**参考框架**，不是强制标准，也不是成熟度评级。它的作用是给你一套共同语言——当你跟别人说"我们的持续测试卡在度量这一块"，对方能立刻明白你在说什么。**有了共同语言，改进才谈得下去**。

板书：中间写"持续测试实践框架（《持续测试白皮书》）"，四周四个块：**组织与角色／流程与规范／技术与工具／度量与改进**。下面一行："缺一块，三块退化"。底部："先能量化，才能改进——反馈时间、逃逸缺陷率、用例稳定率"。

提问：如果一个团队技术很强、自动化脚本写得很好，但半年后脚本全废了。按这个框架看，最可能缺哪一块？

预设回答：

- 答"缺组织，没人负责维护"——最常见。测试资产如果没有明确 owner，就会随人员流动一起消失。

- 答"缺度量，没人知道它失效了"——也很关键：如果没人统计用例稳定率和反馈时间，脚本烂掉的过程是无声的。

- 追问：这告诉我们什么？答：**持续测试是组织能力，不是个人技能**——这也是它叫"持续"的原因。

易错点 / 考点：① 这一页考法很轻，通常只考"持续测试实践框架／白皮书的四个维度"这类填空；② 记住"持续测试是**组织能力**"这个判断，论述题可以引用；③ 别把白皮书当成标准（它不是强制规范），表述时说"实践框架／参考模型"更准确。

过渡：第 5 章到这里讲完了。我们用一页把整章收口。

【5.95】本章小结（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章的收口，六条，我按"从静态到动态、从零件到系统"的顺序重讲一遍，你对着自己的笔记查漏。

第一条，代码规范与代码评审的价值。要深刻理解它，并且**持续做**——方式是**互为评审、集体评审**。为什么强调"互为"？因为单向评审会退化成"资深的人挑新人的错"，而互为评审是**团队共同建立标准**。课件说"团队

有多方面的收益"，这个收益不只是缺陷变少，还包括知识共享和标准统一。

第二条，代码分析工具的使用。课件说"今天很流行"，两个落点：**开发本地扫描**（写的时候就发现问题，反馈最快）和**与 CI/CD 流水线集成**（保证每次提交都被扫）。这两条要同时有，只有本地扫描靠自觉，只有流水线扫描反馈太慢。

第三条，深刻理解单元测试（UT）的价值和 xUnit 框架的实现机制。这里要注意"实现机制"四个字——JUnit 不是一个魔法，它就是**注解 + 反射 + 断言 + 运行器**这套东西。理解了机制，你换任何语言的 xUnit 都能上手。

第四条，掌握 JUnit 这类工具和 mock 工具的使用技巧。技巧的关键在**取舍**：什么时候用 Stub、什么时候用 Mock、什么时候干脆不 Mock。记住那句原则——**够用就行，Mock 越多离真实越远**。

第五条，单元测试是否充分，需要衡量代码语句／分支覆盖率，一般用覆盖率工具自动分析。注意这句话的两个限定：**语句和分支**（最常用的两个维度）、**自动分析**（人算不出来）。而覆盖率数字要看，但不能只看数字。

第六条，今天提倡持续测试以促进持续交付；契约测试值得关注，从开发自测开始，比较彻底地保证质量。这一条把整章的技术和流程接在了一起：**契约测试让你在开发阶段就能验证跨服务约定**，这是"从开发自测开始"的真正含义。

六条合起来是一句话：**先用评审和静态分析在不运行代码时抓缺陷，再用单元测试把每个函数钉死，用覆盖率衡量够不够，用契约测试守住服务边界，最后靠持续集成让这一切每次都发生。**

> **板书：**竖排六条关键词：① 规范与评审（互为／集体，持续做）② 分析工具（本地＋流水线）③ UT 价值与 xUnit 机制 ④ JUnit／Mock 技巧（够用就行）⑤ 覆盖率（语句／分支，自动分析）⑥ 持续测试＋契约测试。底部收口一句："评审→单测→覆盖率→契约→持续集成"。

提问：这六条里，如果只能保留三条，你保留哪三条？说明理由。

预设回答：

- 答"评审、单元测试、持续集成"——很稳的组合：一个管"不运行就能发现"，一个管"零件正确"，一个管"每次都发生"。

- 答"覆盖率、契约测试、持续集成"——也成立，但要指出：覆盖率是**度量**，它自己不产生质量，得靠单测支撑。

- 追问：为什么"持续集成"几乎所有人都会保留？答：因为它是**机制**——前五条都是"能力"，只有它是"让能力持续发生"的那一环。

易错点 / 考点：① 六条要能默写关键词，简答题常要求"简述本章主要内容"；② 覆盖率那条的两个限定（语句／分支、自动分析）别漏；③ 契约测试在本章的地位要摆对：**它是第 5 章的新增重点**，也是与第 4 章流程衔接的落点；④ 论述题可用"评审→单测→覆盖率→契约→持续集成"这条主线组织答案。

【思政落点 14】：这一章讲的东西都有点"笨"：评审要一行行看，单元测试要一个个写，覆盖率要一格一格补，契约要一条条对。它们不像"上个自动化平台"那样有成就感，可**系统的可靠性恰恰是从这些笨功夫里长出来的**。工程质量没有捷径，只有把基础做扎实的耐心。希望同学们愿意在没人看见的地方多写一个边界用例、多补一个异常分支——**这些不被表扬的部分，才是你专业性的证据**。

过渡：理论讲完了，接下来是动手。实验 2 要求你用 JUnit 把一个 Spring 控制器测出来，我们把要求说清楚。

【5.96】实验 2： 单元测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

实验 2 的任务一句话就能说完：**使用 JUnit 工具，针对 Spring Unit Testing 控制器代码中的 `ItemController` 类进行测试，编写对应的测试类完成单元测试，最终提交测试代码**。但我把要求拆细，否则你会发现"能跑"和"合格"差得很远。

第一，被测对象是谁。`ItemController` 是一个 Spring MVC 的控制器，它的职责是**接收 HTTP 请求、调用 Service、返回响应**。所以你要测的不是"业务算得对不对"（那是 Service 的事），而是**控制器这一层的三件事**：请求参数能不能正确绑定、Service 返回什么它就返回什么、异常和错误码处理对不对。

第二，怎么隔离依赖。控制器依赖 Service，Service 可能又依赖数据库。**不能真连数据库**——那样就不是单元测试了。两种常见做法：一是用 `@WebMvcTest` 只加载 Web 层，把 Service 用 `@MockBean` 换掉；二是不启 Spring 容器，直接 `new ItemController(mockService)`，用 Mockito 造 Service，再直接调用方法。**两种都可以，但你必须在实验报告里说清你选了哪种、为什么**。

第三，用例要覆盖哪些场景。我给你一个最小清单，你要在此基础上自己加：**正常查询**（有数据／无数据）、**新增**（合法参数／缺必填参数／参数类型不合法）、**修改与删除**（存在／不存在）、**异常路径**（Service 抛异常时控制器返回什么）。用 AAA 结构写：准备（Arrange）、执行（Act）、断言（Assert），**一个用例只验一个行为**。

第四，断言要硬。不要只 `assertNotNull`。要断言**状态码、响应体的关键字段、Service 被调用的次数和参数**（`verify`）。这是"行为验证"，也是本实验的评分点之一。

第五，提交物。至少包含：测试类源码（放在 `src/test/java/.../ItemControllerTest.java`）、一份覆盖率报告截图（说明分支覆盖情况）、一份简短的实验说明（测了哪些场景、哪些没测、为什么没测）。

最后提醒三条纪律：**测试方法命名要能读懂意图**（比如 `findById_whenNotFound_returns404`）；**测试之间不能有依赖**，任意顺序都能跑；**不许改被测代码来让测试通过**——如果确实发现被测代码有问题，写进报告，别偷偷改。

板书：五行：① 被测＝`ItemController`（参数绑定／响应／异常处理），不测 Service 业务 ② 隔离＝`@WebMvcTest`＋`@MockBean` 或 `new Controller(mockService)` ③ 场景清单：查（有／无）・增（合法／缺参／类型错）・改删（存在／不存在）・异常 ④ 断言：状态码＋响应字段＋`verify` 调用 ⑤ 提交：测试类＋覆盖率截图＋说明。底部三条纪律：命名可读／用例独立／不改被测代码。

提问：`ItemController` 里有个方法直接返回了 `null` 而没有返回 404，测试该怎么写才算"测出来了"？

预设回答：

- 答"断言返回 null"——那只是**记录现状**，不是测出问题：测试会把错误行为固化下来，以后谁修好了反而测试红。

- 答"断言状态码应该是 404，现在不是，所以测试红"——这才是正确做法：**测试写的是"应该怎样"，不是"现在怎样"**。红了正好证明缺陷存在。

- 追问：那测试红了，实验算失败吗？答：不算——**能用一个失败的用例精准指出缺陷位置，比一堆绿的用例更有价值**，把它写进报告。

易错点 / 考点：① 实验最常见的失分是"**测了 Service 的业务逻辑**"，那不是本实验的目标；② 第二常见的是"**断言太软**"（只判非空、只判不抛异常）；③ 用 `@WebMvcTest` 时容易踩的坑：它不加载 `@Service`，忘了加 `@MockBean` 就会启动失败；④ **不许为了通过测试修改被测代码**，这是纪律问题，也会直接影响评分。

过渡：想在这条路上继续深入，有几本书和两个官方站点值得你收藏。

【5.97】学习资源推荐（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是资源清单，我按"读完能解决什么问题"给你排个序，你别一次全买。

第一本，**《程序开发人员测试指南：构建高质量的软件》**（人民邮电出版社，2018 年 5 月）。这本适合**现在读**——它讲的正是"开发人员自己怎么写测试"，和你这学期做的事完全对口，读完你的单元测试水平会有明显变化。

第二本，**《敏捷测试》**（人民邮电出版社，2021 年 8 月）。这本适合**做完实验之后读**，它讲的是团队层面怎么组织测试、怎么在快速迭代里保证质量，正好接上本章最后的持续测试。

第三本，**《Java 微服务测试》**（电子工业出版社，2019 年 7 月）。这本是**进阶**，讲微服务下怎么测：契约、消费者驱动、测试金字塔在微服务里的变形。你如果要做微服务方向的项目，这本值得细读。

两个站点更重要，因为它们是**活的**：**junit.org/junit5** 是 JUnit 5 的官方文档，你实验里遇到的注解、断言、扩展模型，全在里面，比任何二手教程都准；**docs.pact.io** 是 Pact 的官方文档，契约测试那一套流程和代码示例都在上面。

再说说怎么用这几份材料。**实验卡住的时候查官方文档**，因为报错信息、注解用法、版本差异都在那儿；**想系统补基础的时候读书**，因为书有章节顺序和上下文；**想找方向的时候看白皮书和案例**，因为它们告诉你行业在往哪走。三种材料对应三种需求，别指望一本书解决所有问题。

我给你的建议是：**先读官方文档，再读书**。官方文档更新快、写得准，缺点是它假设你已经知道要解决什么问题——所以配合教材和这几本书打底，效果最好。

> **板书：**两栏。左"书"：《程序开发人员测试指南》→ 现在读；《敏捷测试》→ 实验后读；《Java 微服务测试》→ 进阶。右"站点"：**junit.org/junit5**（注解・断言・扩展）、**docs.pact.io**（契约流程・示例）。底部一句："先读官方文档，再读书。"

提问：如果只有一周时间，你在这三本书和两个站点之间怎么分配？

预设回答：

- 答"先看 JUnit 文档，边做实验边查"——最实用，实验驱动学习效率最高。

- 答"读第一本书的前几章"——也可以，但要配套动手，光读不写记不住。

- 追问：为什么不建议先读微服务那本？答：**没有单元测试和契约的基础，微服务测试那一套读起来是空中楼阁**——顺序很重要。

> **易错点 / 考点：** ① 这一页一般只作了解，考法最多是"推荐一本单元测试相关的书"这类开放题；② 书名、出版社、年份不必死记，但**两个官方站点要记住**（实验和后续作业都要用）；③ 学习路径的判断（先文档后书、先基础后微服务）比书目本身更值得记。

> **过渡：** 这一章就到这里。最后，我把这 90 分钟想留下的东西，用几句话收一下。

【5.98】感 谢 聆 听（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一章我们从"代码这一层怎么测"出发，走完了评审、静态分析、单元测试、Mock、覆盖率、集成策略、契约测试、持续集成这一整条线。如果 90 分钟之后你只记住三句话，我希望是这三句。
>
> 第一句：**测试不是编码之后的补救，是编码过程的一部分**。评审、静态扫描、单元测试，这些都能在代码还没跑起来的时候就发现问题，越早越便宜。
>
> 第二句：**覆盖率是镜子，不是奖状**。它告诉你哪里没人走过，不告诉你"够了"。真正的标准是人定的，工具只负责把事实摆出来。
>
> 第三句：**让好习惯每次自动发生**。会写测试不难，难的是每次提交都跑、每次改动都验。把这件事交给流水线，别交给自觉。
>
> 下一章我们会继续往系统层面走，看性能、安全这些"跑起来之后才暴露"的问题。课下请把实验 2 的 `ItemController` 测出来——**写测试这件事，只有动手才算学会**。
>
> 谢谢大家。

> **板书：** 三行大字：① 测试在编码中，不在编码后 ② 覆盖率是镜子，不是奖状 ③ 让好习惯每次自动发生。下方一行小字："实验 2：`ItemController` 单元测试，提交测试代码。"

> **提问：** 这一章里，哪一个知识点你认为最可能在你的第一个项目里立刻用上？为什么？

> **预设回答：**

- 答"单元测试和 Mock"——最直接，进项目第一周就会写。

- 答"代码评审的检查表"——也很实用，评审是最快能参与的团队活动。

- 答"契约测试"——如果进的是微服务团队，这个最值钱，但需要团队配合才能落地。

- 收口：三种答案都对，取决于你的岗位；但**如果只能先练一样，练单元测试**——它是所有其他实践的基础。

> **易错点 / 考点：**无考点。这一页的作用是**收束全章、指向下一章、明确课下任务**，讲的时候不要拖，一分钟内结束，把时间留给学生提问。

> **过渡：**本章结束。下一次课进入第 6 章，我们从"代码正确"转向"系统在压力下还正确吗"。

四、思政落点清单（共 14 处）

序号	所在页	落点主题
1	5.6	代码评审最珍贵的地方，是它把"我的代码"变成"我们的代码"。写代码的人天然有防御
2	5.9	这几条指标背后是一种很朴素的职业态度：认真是有成本的，所以要给它留出成本。"帮我
3	5.28	第三方库这条链，给工科生上了一堂很现实的责任课。你引一个包，可能只是图省事；但这
4	5.33	这六项排下来，你会发现单元测试是件"不显眼"的活：没有界面、没有演示效果、写完了
5	5.34	这两个算式其实是成本意识的数学表达。工程上最贵的一句话是"先这样，回头再说"——
6	5.36	覆盖率这个数字，最考验一个人的诚实。因为它太好看了：写一条把整个方法跑一遍的用例
7	5.39	驱动和桩，本质是为协作搭桥。被测单元本身没变，是周围的环境被"补"出来了——上游
8	5.48	注解看起来只是几个`@`符号，但它体现的是规范意识。`@Test`让框架找得到你
9	5.60	这句"够用就行"是难得的务实精神教科书。技术圈有个通病——用什么工具代表水平：写
10	5.64	`verify()`这个 API 的名字值得琢磨——它叫"验证"，不叫"检查过了就行
11	5.69	覆盖率最容易被用错的地方，是把它当成交差的凭证——数字到 80%就收工，报告一贴，
12	5.82	契约这两个字，在工程上和生活里是同一个意思——我说到做到，你按我说的准备。契约被

13　5.92　质量门禁最有意思的地方是它不商量：单元测试失败就是失败，覆盖率不达标就是不放行，

14　5.95　这一章讲的东西都有点"笨"：评审要一行行看，单元测试要一个个写，覆盖率要一格一格

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 节	6.23–6.55	UI 自动化：WebDriver、PageObject、Cypress、Android 系与两种机制	16	**6.32、6.42 必讲**；其余按 ★ 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	6.67–	小结、思考三问、实验 3、结束	5	6.68 三问开放讨论

节　　6.71

> **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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲 ☐ 需回看 课堂记录：_____

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）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 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——

第 7 章　专项测试

第 8–9 周 · 两次课各 2 学时（共 180 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 7 章 讲稿（第 8–9 周 · 两次课各 2 学时，共 180 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 7 章 专项测试 —— 性能、安全、兼容、可靠、易用，五条非功能战线
>
> 学时：**2 次课（第 8 周、第 9 周各 2 学时，共 180 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 **212 页**（`lecture7/`，云端 http://111.231.0.226:2345/lecture7/index.html）
>
> 教案对应：第 8 次课「专项测试（一）：性能与安全」＋第 9 次课「专项测试（二）：兼容、可靠与易用」
>
> 本讲重点：性能测试的目标与类型（基准／验证／规划／容量；压力／渗入／峰谷）、负载三要素、性能需求的可测试化、性能测试层次与分段计时、JMeter 体系；安全性测试的 SDLC 贯穿与 Web 十大类漏洞的测法；兼容性（软硬件／系统间／数据／环境组合）；可靠性（容错、全链路压测、混沌工程）；易用性与 A/B 测试
>
> 本讲难点：把"每天一千万笔交易"翻译成可测指标；性能问题的外部表现与内因对应；安全漏洞"原理—为何成立—怎么测—怎么防"的完整链路；全链路压测与混沌工程的爆炸半径控制；A/B 测试的分层分流与统计评价
>
> **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 能说出性能测试的四类目标型测试与三类压力型测试的区别；② 能用「RPS＋并发连接数＋思考时间」解释并发用户数，并说明思考时间为零的后果；③ 能把一句模糊的性能要求改写成带峰值系数的可测指标；④ 能说出性能测试的六个层次与请求各段时间的构成；⑤ 能列出安全性测试贯穿 SDLC 的关键活动，并说明 XSS、SQL 注入、会话劫持、直接对象引用、功能级权限缺失各自的原理与测试点；⑥ 能区分兼容性的四个面向（软硬件／系统间／数据／环境组合）；⑦ 能说明全链路压测与混沌工程的作用、以及爆炸半径为什么要控制；⑧ 能说出 A/B 测试的桶／层／流量三层概念与分层分流的意义
能力	① 能为一个系统写出性能测试方案骨架（目标、负载模型、场景、监控指标、判据）；② 能用 JMeter 组织一个最小测试计划（线程组＋取样器＋断言＋监听器）；③ 能对一个登录接口列出至少 6 个安全测试点；④ 能针对一次升级列出兼容性检查单；⑤ 能设计一个小规模混沌实验（稳态假设、注入故障、爆炸半径、观测指标）；⑥ 能读懂一份 A/B 实验报告的核心指标与显著性
素质	① 由「模糊要求→可测指标」体会严谨求实；② 由资源泄漏与长跑测试体会细节与长期主义；③ 由安

（思政）	全测试体会技术能力必须配责任边界；④ 由数据兼容体会对历史数据与用户资产的敬畏；⑤ 由混沌工程的爆炸半径控制体会「敬畏风险、可控试验」；⑥ 由 A/B 评价体会数据伦理——不用数据骗人；⑦ 由复盘文化体会从失败中学习的诚实

本讲要建立的一个总印象：功能测试回答"**做得对不对**"，专项测试回答"**扛不扛得住、守不守得住、换环境还灵不灵、久用会不会垮、人用着顺不顺**"。这五条战线共同决定一个系统能不能真正上线并长期活着。

二、时间分配总表

> ⚠ **容量提示（务必先读）**：本章课件 **212 页**，是全课程页数第二多的一章，本讲稿是**素材全量版**（口播约 13 万字，≈540 分钟纯朗读）。教案把本章排在**两次课（180 分钟）**，请务必按下表取用：
>
> 1. **重点页全文慢讲（2–3 分钟/页）**：
> 7.4、7.7、7.12、7.13、7.15、7.18、7.22、7.34、7.61、7.63、7.78、7.90、7.91、7.114、7.123、7.149、7.150、7.186、7.191、7.199。
>
> 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（★）**：工具安装配置页（7.24–7.33、7.100–7.105、7.130–7.140、7.165–7.181 中的截图页）、厂商实践页（7.153–7.155）、平台架构细节页（7.192、7.207–208）。
>
> 4. **必须当堂互动页**：7.18（改写一条性能需求）、7.90/7.91（现场找 XSS 与注入点）、7.149（写一个混沌实验的稳态假设）、7.199（读 A/B 指标）。
>
> 5. **实验 4-1／4-2（7.39、7.40）**与压测平台实践页，按课程安排交代清楚。

第 8 次课（第 8 周 · 90 分钟）：性能与安全

时段	页码	内容	分钟	备注
第 1 节	7.1–7.8	开场、本章五节地图、性能测试定位、性能问题内外因、定义与目标、负载三要素	12	**7.4、7.7 必讲**
第 1	7.9–7.14	思考时间、负载模式、四类目标型测试、三类压力型测试	10	**7.12、7.13 必讲**

节			
第 1 节	7.15–7.23	性能层次、全生命周期、性能需求、可测试性、架构与代码评审、设计与执行、分段计时、工具	14	**7.15、7.18、7.22 必讲**；工具清单可略
——	——	**课间休息 10 分钟**	——	——
第 2 节	7.24–7.40	JMeter 体系、性能分析与调优、性能小结、实验 4	10	只讲结构与方法，安装细节课后
第 2 节	7.41–7.76	7.2 安全性测试：渗透测试、SDLC 贯穿、风险分析、代码评审、基于风险	20	**7.61、7.63 必讲**
第 2 节	7.77–7.108	Web 安全：渗透测试点、XSS、SQL 注入、XML 注入、会话劫持、认证与 Session、未验证跳转、直接对象引用、权限缺失；工具与小节	20	**7.90、7.91 必讲**（当堂找点）
第 2 节	7.109–7.121	7.3 兼容性测试：四类兼容 + 小结	4	现场发兼容性检查单

第 9 次课（第 9 周 · 90 分钟）：可靠、压测、易用

时段	页码	内容	分钟	备注
第 1 节	7.120–7.131	7.4 可靠性：可靠性方法、容错性、全链路压测需求与流量控制、压测平台能力	16	**7.123、7.130 必讲**
第 1 节	7.132–7.181	Takin、混沌工程（原则、爆炸半径、局限、实验/故障库/可观察性、五条优秀实践）、ChaosBlade、Chaos Mesh、KeChaos	26	**7.149、7.150 必讲**；截图页可略
——	——	**课间休息 10 分钟**	——	——
第 2 节	7.182–7.185	7.5 易用性测试：UI 元素、用户体验测试方法、模型、A/B 引入	8	——
第 2 节	7.186–7.209	A/B 测试：特点、场景、核心概念、桶层流量、OpenResty 分流、MurmurHash、流程与数据流、SDK、平台、评价、优秀实	26	**7.186、7.191、7.199 必讲**；平台设计

		践、多因素、决策树、架构		可略
第 2 节	7.210–7.212	本节小结、作业与开源清单、学习资源、结束	4	交代作业

> **压缩优先级**：先压缩 ★ 标记的工具截图与厂商实践页 → 再压缩压测平台与 A/B 平台的架构细节（保留"为什么这么设计"的三句话）→ 最后才压缩
>
> 7.12、7.13、7.18、7.78、7.90、7.91、7.123、7.149、7.191、7.199。**这十页是考点密集页，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（7.1–7.212），可逐页对照：http://111.231.0.226:2345/lecture7/index.html
>
> **三处口径提醒**：① 7.28 页课件对 JMeter 的 `-p` 参数有两处不同含义（属性文件／代理端口），实际命令行为 `-P`，讲稿已按命令行事实标注勘误；② 7.3 页"本节内容进行了优化…"是作者备课备注，不作为考点；③ 7.194 写作「MurmurHash」，属哈希算法名，按课件拼写。

【7.1】软件测试方法和技术（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 各位同学，我们进入第 7 章——**专项测试**。
>
> 前面六章我们把"通用测试"这条主线走完了：第 2 章讲概念，第 3 章讲方法，第 4 章讲流程，第 5 章钻进代码内部做单元与集成，第 6 章站到系统外部做功能测试。这条主线回答的是同一个问题：**这个软件做得对不对**——输入进去，输出对不对，业务规则算得对不对。
>
> 可你遇到过这种情况吗：功能一条条都对，用户一多就卡死；功能都对，被人拿一个单引号就把数据库拖走了；功能都对，换个手机型号界面就崩了。这些毛病功能测试一个都抓不住，因为它们不是"对错"问题，是"**在特定条件下还站不站得住**"的问题。
>
> 第 7 章专门收拾这一类问题：性能、安全、兼容、可靠、易用。这一讲我们先攻第一个，也是最烧钱、最容

易在生产上炸的那个：**性能**。带着一个问题听：**你的系统能扛住多少人同时用？这个数字你是从哪知道的？**

> **板书：**中间写"第 7 章 专项测试"；左侧竖写"第 2–6 章：对不对（概念・方法・流程・单元集成・系统功能）"；右侧写"第 7 章：扛不扛得住（性能・安全・兼容・可靠・易用）"。

提问：一个功能完全正确的系统，还有可能上线就出事吗？举一个你自己用过的例子。

预设回答：

- 答"会，双十一抢购页面一直转圈"——这就是性能问题，功能没坏，是扛不住。

- 答"会，某个 App 换了手机就闪退"——兼容性专项，第 7 章后面会讲。

- 答"功能都对就不会出事"——这是最典型的误解：功能测试的判据是"预期结果"，而性能、安全这类问题的判据是"资源、时间、攻击面"，功能用例根本覆盖不到。

- 收口：这三种回答的差别，就是"通用测试"和"专项测试"的分界线。

易错点 / 考点：① 专项测试不是"更细的功能测试"，它的判据不是"输出对不对"，而是"**指标达不达标**"；② 五个专项的名称要能默写：性能、安全性、兼容性、可靠性、易用性；③ 判断题常见陷阱："专项测试只在系统测试阶段做"——它在单元、集成、系统、验收各阶段都有对应活动，下一章讲全生命周期时会再遇到。

过渡：先看这五个专项在整章里怎么排，你心里有个时间预算。

【7.2】7.1 性能测试（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是本章的五个板块，我按顺序点一遍，你记结构就行。

7.1 性能测试——本讲的主体，占的时间最长。要回答：性能问题长什么样、性能测试的定义和三个目标是什么、负载由哪几个要素构成、性能测试有哪几种类型，然后落到工具和结果分析上。

7.2 安全性测试——看的是"软件会不会做它不该做的事"：漏洞从哪来、攻击分几个阶段、安全目标是什么、怎么测。

7.3 兼容性测试——同一套代码换环境还行不行。

7.4 可靠性测试——长时间跑、出故障了能不能恢复。

7.5 易用性测试——人用起来顺不顺。

五块有一条共同的主线：**它们衡量的都不是"对不对"，而是"够不够"**——够快、够稳、够安全、够通用、够好用。"够不够"这三个字，意味着必须有**量化指标**，没有指标就没法判合格。这是整章最重要的一句话。

板书：竖排五块：7.1 性能（快不快）→ 7.2 安全（会不会被攻破）→ 7.3 兼容（换环境行不行）→ 7.4 可靠（长时间稳不稳）→ 7.5 易用（人顺不顺）。右侧一行大字："专项测试＝用指标判'够不够'，不是判'对不对'"。

提问：如果客户只说"这个系统要快一点"，你能开始做性能测试吗？

预设回答：

- 答"能，先测了再说"——测出来的数字没有参照系，报上去也没人认。

- 答"不能，得先有指标"——对。"快一点"要翻译成"首页 95% 的请求在 2 秒内返回"这种可验证的句子。

- 追问：指标从哪来？三条路——用户预期、业务容量、技术约束，7.1 后面专门有一页讲。

- 收口：记住这句话：**没有指标的专项测试，只能算摸底，不能算验收**。

易错点 / 考点：① 五个板块的名称与顺序；② 常考简答："专项测试与功能测试的根本区别"——判据不同（指标 vs 预期结果）、手段不同（工具模拟与监控 vs 输入输出比对）；③ 别把"兼容性"和"可移植性"混为一谈，兼容性测的是"在既定环境下能不能正常工作"，可移植性测的是"搬到新环境要花多少代价"。

过渡：从 7.1 开始。我先给你一张这一节的地图，因为性能这一节概念密度最高。

【7.3】7.1 性能测试（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 7.1 节的分隔，我用一分钟把路线画出来。

这一节分四段。**第一段"认识"**：性能问题有哪些外部表现、内因是什么；性能测试怎么定义、目标有哪三个。**第二段"设计"**：负载由哪三个要素构成、思考时间为什么关键、负载模式怎么设计、性能测试分成哪几类。**第三段"实施"**：从需求到架构评审、代码评审、测试设计与执行，一路到工具——客户端和服务器端各有哪些。**第四段"收口"**：性能分析、调优、监控，以及全生命周期的性能工程。

一句说明：这一节的课件做过精简，压力测试、容量测试和工具的**细节**教材里有更完整的篇幅，课件留下的是**核心概念与判断口径**。所以课上讲骨架，命令与参数你课后对着教材和实验做。

一句话记住这一节的重点：**性能测试不是"跑个工具看数字"，它是"先定义指标，再用负载去逼，最后用数据说话"的闭环。**

板书：一条横线分四段：认识（表现・内因・定义・目标）→ 设计（负载三要素・思考时间・负载模式・测试类型）→ 实施（需求・架构评审・代码评审・设计执行・工具）→ 收口（分析・调优・监控・全生命周期）。底部："先定指标 → 用负载去逼 → 用数据说话"。

提问：为什么性能测试特别强调"先定指标"？换成功能测试，我们会先定指标吗？

预设回答：

- 答"功能测试也有预期结果啊"——对，功能的"预期结果"是**离散的**：对或错；性能的"预期结果"是**连续的**：多快算快、多稳算稳，必须人为划线。

- 答"因为性能没有绝对的对错"——抓住了本质。所以指标不先定，测完就是一堆没人能判读的数字。

- 追问：那指标定错了会怎样？会做出一份"看起来很专业、结论没法用"的报告——这是行业里最常见的浪费。

- 收口：功能测试怕漏用例，性能测试怕**指标含糊**。

易错点 / 考点：① 这一节的四段结构要能说清；② 常考判断："性能测试就是压力测试"——错，压力测试只是性能测试里的一种类型，7.1 后面会分清四种基础类型和三种加码类型；③ 课件提到"内容已优化"，意味着**压力测试、容量测试的工具细节要去教材补**，考试若考细节，以教材为准。

过渡：先建立最基本的认识——为什么今天性能问题变得这么常见。

【7.4】性能测试（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页回答一个前置问题：**为什么性能测试在今天这么重要**。课件给了六条，我按"为什么—怎么变—难在哪"三层讲。

第一条，**性能问题是常见的问题**，特别是在 SaaS 时代、万物互联时代。这句话要拆开看：SaaS 意味着**一套系统同时服务成千上万个租户**，一个人的慢可能是所有人的慢；万物互联意味着**终端数量级从十万级跳到亿**

级，心跳、上报数据把后端压到另一个量级。用户规模上去了，性能问题自然变成高频问题。

第二条，**压力测试、容量测试等也并入性能测试**。也就是说，"性能测试"这个词今天是个**大口袋**：基准、验证、规划、容量、压力、渗入、峰谷，全在里头。考试问"性能测试包括哪些"，别只答一种。

第三条，性能测试**也可以分为静态测试和动态测试，通常意义指动态测试**。静态那部分是架构评审、代码评审里"看出来的性能隐患"——循环里反复创建对象、SQL 没走索引；不运行程序，属静态测试。平时说"做性能测试"，指的是**把系统跑起来、加负载、看指标**，这是动态测试。

第四条，也是最实在的一条：**性能测试一般只能通过工具来实现**。为什么？因为你需要同时模拟几百上千个并发用户、精确控制每秒请求数、同步采集几十项指标——人做不到，手点更做不到。功能测试可以手工点，性能测试手工点等于没测。

第五条，**结果分析有挑战，需要良好的技术功底**。这一点后面会反复出现：工具给你的是曲线和数字，"为什么慢"要靠你对操作系统、数据库、网络、代码的理解去推。工具能测出问题，但**定位问题靠人**。

第六条，**性能测试与性能调优相辅相成，循环往复、不断提升性能**。测出瓶颈→改→再测→再改，这个循环可能走十几轮。所以性能不是"测出来的"，是"**逼出来、改出来的**"。

> **板书：**六行：① SaaS／万物互联 → 规模变大；② 压力・容量并入性能；③ 静态（评审）＋动态（跑负载），通常指动态；④ 只能靠工具（并发・精确・采集）；⑤ 结果分析靠功底；⑥ 与调优循环往复。右侧写一句："性能不是测出来的，是逼出来、改出来的。"

提问：既然性能测试只能靠工具，那是不是装上 JMeter 就会做性能测试了？

预设回答：

- 答"是，工具会跑就行"——工具只解决"施压"和"采集"，不解决"指标定得对不对""数字说明什么"。会跑工具的人很多，能出可用结论的人很少。

- 答"不是，还得会分析"——对，这就是课件第五条讲的"技术功底"。

- 答"还得会设计场景"——补得好。负载多大、怎么加载、跑多久，这些决定你测出来的是不是你想要的东西。

- 收口：工具是**必要条件，不是充分条件**；性能测试的能力，一半在工具，一半在分析和设计。

易错点 / 考点：① 六条要点里，"只能靠工具"和"结果分析有挑战"是简答高频点；② 静态/动态之分要记准：**架构评审、代码评审属于静态性能测试**，这与第 3 章"静态测试不运行程序"的定义一致；③ 常考判断："性能测试做一次就够了"——错，它与调优循环往复，且系统每次大改、数据量每上一个量级都要重做基线。

> **过渡：**既然性能问题这么常见，那它到底长什么样？我们先从用户能感知到的表象看起。

【7.5】性能问题（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页开始进入"性能问题"这个话题，我先把话题的落点说清楚：**接下来两页是一组——先看表象，再看内因**。这一页是引子，下一页把它们对上。
>
> 为什么先讲表象？因为性能问题的发现者，几乎永远不是测试人员，是**用户**。用户不会跟你说"你的线程池打满了"，他只会说"你们这个系统今天怎么这么慢"。所以做性能测试的人，第一项能力是**把用户的抱怨翻译成技术问题**："越来越慢"可能是资源泄漏，"突然卡死"可能是连接池耗尽，"偶尔慢一下"可能是 GC 或者慢查询。
>
> 这节课我给你一个固定的思路，遇到任何"慢"的反馈，都问三句：**慢在什么时候**（启动、查询、下载）、**慢给谁看**（所有人还是个别人）、**慢多久了**（一直慢、越来越慢、还是偶发）。这三句问完，问题的方向基本就定了。
>
> 再强调一句：性能问题最贵的地方不在修，在**发现得晚**。功能缺陷上线可能影响一个功能，性能缺陷上线影响的是**全部用户同时**，而且往往在流量最高的时候发作。

> **板书：**中间写"性能问题"；左侧写"谁先发现：用户"；下方三问："慢在什么时候？慢给谁看？慢多久了？"；右侧留白，准备接下一页的"外部表现 ↔ 内因"。

> **提问：**用户说"你们系统最近变慢了"，你要问的第一句话是什么？

> **预设回答：**

- 答"是所有页面都慢，还是某个功能慢"——好问题，先划范围。

- 答"是一直这样，还是这两天才开始"——好问题，时间维度能直接指向"是不是最近改了什么/数据涨了多少"。

- 答"我马上重启一下服务"——能缓解，但**重启会毁掉现场**：内存里的泄漏证据、线程栈、连接数全没了。性能问题第一原则是**先保现场，再动手**。

- 收口：这三句追问的习惯，比任何工具都值钱。

> **易错点 / 考点：**① 性能问题的第一发现者通常是用户和运维，不是测试；② "重启解决"是缓解不是修复，考试若问"线上变慢的正确处置顺序"，答：**保留现场（日志、线程栈、监控曲线）→ 定位 → 再处置**；③ 别

把"慢"当成单一问题，它至少有"一直慢、越来越慢、偶发慢"三类，对应完全不同的根因。

过渡：用户的抱怨是表象，下一页我们把表象和它背后的三种内因一一对上。

【7.6】性能问题外部表现和内因（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是性能这一节里最实用的一页，它把"用户说的"和"机器里发生的"接上了。课件左边给外部表现，右边给内因，我们一组一组对。

外部表现有三种典型：第一种，**启动系统、打开页面越来越慢**——注意"越来越"这三个字，它是**时间维度上的恶化**，往往指向资源在持续累积；第二种，**查询数据很长时间才显示列表**——这是"单次操作超时"，指向一次请求内部的效率问题；第三种，**网络下载速度很低，比如 5k/s**——这是"带宽或传输链路"的问题，注意它的单位是速度，不是时间，说明瓶颈在传输能力而不在计算。

内因有三种：第一种，**资源耗尽，比如 CPU 使用率达到 100%**——资源被吃干净了，请求排不上队；第二种，**资源泄漏，比如内存泄漏，最终会导致资源耗尽**——注意这个因果：泄漏本身不是立刻出事，它是**慢慢把可用资源吃掉**，最后表现为耗尽；第三种，**资源瓶颈，比如线程、GDI、DB 连接等资源变得稀缺**——这里要特别讲清楚"耗尽"和"瓶颈"的区别：**耗尽是没有了，瓶颈是不够用**。线程池 200 个线程，来了 500 个并发请求，线程没"耗尽"（还能创建），但连接池上限就在那儿，多出来的请求只能排队——这就是瓶颈。

为什么一定要区分这三种？因为它们**修法完全不同**：耗尽要扩容或限流，泄漏要找到没释放的对象，瓶颈要调参或改架构。你把泄漏当瓶颈去扩容，只会让泄漏更快地把新内存吃光。

再给你一个对应关系的口诀：**越来越慢找泄漏，一次就慢找查询，速度很低找带宽**。

板书：左右两栏对照。左（外部表现）：启动/打开页面越来越慢｜查询很久才出列表｜下载速度很低（5k/s）。右（内因）：资源耗尽（CPU 100%）｜资源泄漏（内存泄漏 → 最终耗尽）｜资源瓶颈（线程・GDI・DB 连接稀缺）。中间画箭头连起来，底部写口诀："越来越慢找泄漏，一次就慢找查询，速度很低找带宽"。

提问：一个系统"白天正常、晚上越来越慢，重启就好"，你判断是哪一类内因？为什么不是"资源耗尽"？

预设回答：

- 答"内存泄漏"——方向对。"越来越慢＋重启就好"是泄漏的经典签名：重启把累积的占用清零了。

- 答"资源耗尽"——不够准。耗尽是**结果**，泄漏是**原因**；耗尽通常表现为"直接拒绝服务/报错"，而"越来越慢"说明还没到顶，是**逐步逼近**的过程。

- 答"那就是内存不够，加内存"——加内存只是把发作时间往后推，泄漏的速率不变，下次爆得更晚一点而已。

- 追问收口：所以定位的第一步不是加资源，是**看内存曲线是不是锯齿状往上爬**——爬，就是泄漏；平的，才考虑瓶颈。

> **易错点 / 考点：**① **泄漏 ≠ 耗尽**：泄漏是过程，耗尽（或 OOM）是结果，答题时不要混用；② **耗尽 ≠ 瓶颈**：耗尽是没有了，瓶颈是不够用；③ 三个外部表现要能反向推出内因，这是案例题的高频问法；④ 记忆抓手：外部表现看"时间形态"（越来越／一次／速度），内因看"资源三态"（耗尽／泄漏／瓶颈）。

> **【思政落点 1】**：内存泄漏最值得琢磨的地方是——**它每一次泄漏都很小**。一个请求漏几十 KB，跑一小时看不出来，跑一周才出事。这类问题考验的不是聪明，是**对细节的敬畏**：分配了就要释放，打开了就要关闭，加了缓存就要想淘汰。工程上真正的质量差距，往往不在"会不会写高级算法"，而在"有没有人认真处理这些看起来微不足道的地方"。请记住：**小问题乘以时间，就是大事故**。这种"把每个细节做干净"的习惯，就是工匠精神最朴素的样子。

> **过渡：**知道了性能问题长什么样，接下来要给它一个严格的定义——什么是性能测试。

【7.7】什么是性能测试？（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页给性能测试下定义，定义里有五个关键词，我一个一个抠，因为考试就抠这几个词。
>
> 课件原文是：性能测试就是为了**发现系统性能问题**或**获取系统性能相关指标**而进行的测试。一般在**真实环境**、**特定负载条件**下，通过**工具模拟**实际软件系统的运行及其操作，同时**监控**性能各项指标，最后对测试结果进行**分析**以确定系统的性能状况。
>
> 五个关键词：**真实环境**——性能和机器配置、网络、数据量强相关，在开发机上测出来的数字没有意义；**特定负载**——不是"随便跑跑"，负载要设计，是 100 并发还是 1000 并发，结论完全不同；**工具模拟**——刚才讲过，只能靠工具；**监控指标**——只施压不监控，等于只知道"慢了"不知道为什么慢；**结果分析**——工具出图，人出结论。

然后是**三个目标**，这三条必须背下来：第一，**获取系统性能某些指标数据**——先知道现状，比如首页响应时间到底是多少；第二，**为了验证系统是否达到用户提出的性能指标**——拿数据和指标对，达标还是没达标；第三，**发现系统中存在的性能瓶颈，优化系统的性能**——找出卡在哪，然后改。

这三个目标是**递进**的：先"知道"，再"判定"，最后"改进"。第一个目标对应基准测试，第二个对应验证测试，第三个对应压力、渗入、峰谷那一类——你后面会发现，**测试类型是按目标来的**，不是按工具来的。

还有一句要提醒：定义里"发现性能问题**或**获取指标"用了"或"字，说明性能测试**不一定要有明确的指标才能做**——摸底也是合法的目的。但如果要做**验收**，就必须有指标。这个区别，是很多人写测试报告时含糊的地方。

板书：定义五个关键词竖排：真实环境｜特定负载｜工具模拟｜监控指标｜结果分析。右侧三个目标带箭头：① 获取指标数据 → ② 验证是否达标 → ③ 发现瓶颈・优化性能。底部："先知道 → 再判定 → 后改进"。

提问：老板说"给我测一下这个系统性能怎么样"，这属于三个目标里的哪一个？需要准备什么才能做？

预设回答：

- 答"第三个，找瓶颈"——不一定，这句话更像第一个目标"摸底"。要往下走，得先问清"关心什么指标"。

- 答"第一个，获取指标数据"——更准确。这时需要准备的是：环境（真实配置）、负载模型（大概多少人用、什么操作占比）、监控项。

- 答"没指标就测不了"——错。定义里"获取指标"本身就是合法目的；只是这种结果**只能叫摸底报告，不能叫达标结论**。

- 追问收口：所以接需求时先问一句"**这份报告要用来做什么决定**"，答案决定了你要测什么、测多细。

易错点 / 考点：① 定义里的五个关键词是简答得分点，缺一个扣一分；② 三个目标的**顺序**不能颠倒，它是"知道—判定—改进"的递进；③ 常考判断："性能测试必须在生产环境做"——错，要求的是**真实环境**（配置、数据量、网络接近生产），不是非得在生产上跑；④ 别把"性能测试"和"性能调优"混为一谈：前者是**度量**，后者是**改动**。

【思政落点 2】：三个目标里最容易被糊弄的是第二个——"验证是否达标"。现实中不少报告会悄悄**降低标准**：指标定 2 秒，实测 2.8 秒，就在报告里写"基本满足用户感知"。这在技术上叫口径调整，在职业上叫**不诚实**。工程判断可以有弹性，但**数据不能修饰**：达标就是达标，差 0.8 秒就写差 0.8 秒，再说明原因和风险。请记住：**报告里的每一个数字，都是你职业信誉的一部分**。敢写不好看的数字，才配写重要的报告。

> **过渡：**定义里有"特定负载"，那负载到底是什么？课件把它拆成了三个要素。

【7.8】负载（Workload）（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲负载，是性能测试里最基础也最容易记错的一页。课件给了三个要素，还有一个等式。

第一个要素，**RPS**，课件写的是"每次请求发送的数据量（Request Per Second, RPS）"。这里提醒一句：**括号里的英文是"每秒请求数"，中文描述写的是"每次请求发送的数据量"**，两个口径不完全一致。按英文和行业通用口径记——**RPS 是每秒发出的请求数**，是"速率"；每次请求发多少数据是"请求大小"，通常单独作为参数描述。名词解释写"每秒请求数"最稳。

第二个要素，**SC，并发连接数（Simultaneous Connections）**——同一时刻保持着的连接有多少条。注意它和"并发用户数"不是一回事：一个用户可以开多条连接（浏览器并发下载资源），一条连接也可能服务多个用户（长连接复用）。

第三个要素，**思考时间（thinking time）**，课件定义为"用户发出请求之间的间隔时间"。这是最人性化的参数——真人点一下、看一眼、想一下才会点第二下。工具模拟时忘了这一条，你造出来的就不是"用户"，是"机器"。

然后是课件给的等式：**RPS + SC + Thinking Time = Concurrent users**。请**按课件原样记**，不要自己改写。它的教学含义是：**并发用户数不是孤立数字，由请求速率、连接规模、思考间隔共同决定**。同样"1000 个并发用户"，思考时间 1 秒和 10 秒，压力差十倍——这才需要下一页专门讲思考时间。

补一句：真实场景里三者关系比一个加号复杂（间隔变长，同样人数下 RPS 会下降），所以这个式子更适合当**记忆框架**，不能直接代入计算。真要估算，用"用户数 ÷ 思考时间 ≈ 请求速率"。

> **板书：**三行写要素：RPS＝每秒请求数（速率）｜SC＝并发连接数｜Thinking Time＝请求间隔。下面照抄课件等式："RPS + SC + Thinking Time = Concurrent users"，旁边写"按课件原样记，作记忆框架"。底部写近似式："用户数 ÷ 思考时间 ≈ 请求速率"。

提问：同样 1000 个并发用户，一个人每 10 秒点一次，另一个人每 1 秒点一次，服务器承受的压力一样吗？

> **预设回答：**

- 答"一样，都是 1000 人"——最常见也最要命的错。人数相同，**请求速率差十倍**，压力完全不是一个量级。

- 答"不一样，间隔短的压得狠"——对。按近似关系，10 秒间隔约 100 请求/秒，1 秒间隔约 1000 请求/秒。

- 答"那我把思考时间设成 0，压力最大，测得更保险"——这正是下一页要拆的错误观念：**间隔为零不是"更保险"，是"失真"**。

- 追问收口：所以做场景设计，**思考时间不是可有可无的修饰参数，它是负载模型的一部分**。

> **易错点 / 考点：**① RPS、SC、思考时间三个名词的英文与中文要能对上；② 课件等式**照原样背**，但要知道它表达的是"并发用户由三者共同决定"；③ 高频易错：**并发连接数 ≠ 并发用户数**，一个用户可开多条连接；④ 案例题常给"用户数＋思考时间"让你估请求速率，用近似式算，别忘了说明"这是估算"。

> **过渡：**刚才说到"把思考时间设成零"这个想法，课件专门用一页拆它。

【7.9】思考时间的影响（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页只讲一件事，但它是很多人做性能测试时踩过的第一个坑：**思考时间，会决定你测出来的是"用户"还是"洪水"**。
>
> 课件给了两个对照场景。一个是**间隔时间长**，一个是**间隔时间为零**。原文是这么写的：将系统置于相同的高负载下，将请求之间间隔时间设为零，这样**服务器会立即超载，并形成越来越长的等候队列，直至内存耗尽**。
>
> 为什么会出现"队列越来越长"？我们把服务器的处理能力想成一个收银台：它每秒能结 100 单。如果来的顾客每秒 100 个，队伍不长不短，稳定；如果来的顾客每秒 300 个，收银台还是每秒 100 单，那么**多出来的 200 个就必须排队**。队伍一旦排起来，它只会越来越长——因为消化速度恒定，到达速度不变。
>
> 队伍放在计算机里是什么？是**等待处理的请求对象**：线程池的等待队列、连接池的等待队列、操作系统的 socket 缓冲区。这些都是**内存**。队列越长，占用越多，最后就是课件说的"直至内存耗尽"——然后系统不是"慢"，是**直接崩**。
>
> 所以"间隔设为零更保险"这个想法错在哪？错在它测的不是真实用户，是**极限攻击**。真实用户会看页面、会思考、会走神，思考时间就是那段时间。你把间隔清零，等于假设"每个用户都在疯狂连点"，测出来的数字既不能代表生产表现，也不能用来判断容量。

那什么时候该把间隔设为零？有两种正当用途：一是**做压力测试**，目的本来就是"逼到崩"，看崩溃点在哪；二是**做单接口的吞吐上限测量**，想知道这个接口极限能跑多少 TPS。这两种都要在报告里说清楚"这是极限值，不是用户场景值"。

一句话总结：**思考时间是负载模型的一部分，不是可调的装饰；清零是为了找极限，不是为了更严格。**

板书：左边写"间隔长＝真实用户（队列能消化）"，右边写"间隔为 0＝洪水（到达率 > 处理率）"。中间画一个收银台示意：到达 300／秒、处理 100／秒 → 队列↑ → 内存↑ → 耗尽／崩溃。底部写："清零＝找极限，不是更严格"。

提问：压测报告里写"系统在 5000 并发下崩溃"，这个结论能直接拿去说服老板扩容吗？

预设回答：

- 答"能，说明系统扛不住 5000"——要看这 5000 并发是**带思考时间的用户**还是**间隔为零的请求流**。后者是攻击模型，前者才是业务模型。

- 答"不能，得看场景"——对。同一句"5000 并发崩溃"，两种模型下的业务含义差好几倍。

- 答"崩溃就是不行，没什么好说的"——太粗。崩溃点本身是有价值的数据（容量上限），关键是要标注**在什么负载模型下**测得的。

- 追问收口：所以报告里每一个数字后面都要跟三个定语——**什么场景、什么负载模型、什么环境**。缺一个，这个数字就不能被引用。

易错点 / 考点：① 思考时间清零的后果链条要能完整说出：**到达率超过处理率 → 队列增长 → 内存占用上升 → 耗尽／崩溃**；② 常考判断："思考时间越短，测试越严格"——错，它改变的是**负载模型**，不是严格程度；③ 案例题常问"为什么压测结果和线上表现差很多"，标准答案之一就是**思考时间设置与实际不符**；④ 记住"队列"这个词——性能问题的很多现象（响应时间突然飙升、超时率上升）本质都是队列在起作用。

过渡：思考时间定了，接下来要定"怎么把这些请求铺到时间轴上"，这就是负载模式。

【7.10】负载模式（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲负载模式，课件给了两个维度，很短，但很关键。

第一个维度：**加载的循环次数或持续时间**。也就是"这套脚本跑几轮、跑多久"。这一条决定你能发现什么问题：跑 10 分钟，你只能看到"稳态表现"；跑 8 小时，才可能撞上内存泄漏、连接不释放、日志文件涨满、缓存击穿这些**随时间累积**的问题。所以持续时间的设置不是随手写的，它要对应你的测试目的——验证型跑短，渗入型跑长。

第二个维度：**加载的方式或模式，如均匀加载、峰值交替加载等**。均匀加载是"从头到尾压力恒定"，像一条平线，适合测**稳态吞吐能力和资源占用**；峰值交替加载是"高—低—高—低"地来回突变，像心电图，适合测**系统在突变时的反应**：能不能快速扩起来、缩回去会不会抖、缓存冷了会不会打穿数据库。

为什么模式这么重要？因为真实业务**从来不是均匀的**。早上九点打卡、中午十二点订餐、晚上八点看视频、零点结算跑批——这些都是尖峰。一个系统"平均负载下很稳"完全不代表"尖峰来了不崩"。**平均值会骗人，峰值不会。**

所以设计负载模式时，你要先回答：我这个系统，**压力是平的还是尖的**？平的用均匀，尖的用峰值交替，两种都跑一遍最稳。

> **板书：**两行两个维度：① 循环次数／持续时间（决定能暴露什么问题：短跑看稳态，长跑看累积）② 加载方式（均匀＝平线，测稳态吞吐；峰值交替＝心电图，测突变反应）。底部一句："平均值会骗人，峰值不会。"

提问：一个系统"平均 CPU 使用率只有 30%"，能说明它很健康吗？

预设回答：

- 答"能，30% 很空闲"——平均值会掩盖尖峰。真实情况可能是"平时 10%，每天两次冲到 100%"。
- 答"不能，要看峰值"——对。看 P95／P99 或者峰值时段的曲线，才看得出有没有被顶到天花板。
- 答"还得看是哪个时间段"——补得好，业务有明显时间规律的系统，必须按**时间窗**分别看。
- 追问收口：所以负载模式的设计，本质是**把你担心的那个时间窗，在实验室里复现出来**。

易错点 / 考点：① 两个维度都要答全，很多同学只记得"跑多久"，忘了"怎么加载"；② 均匀加载和峰值交替加载**各自适合什么问题**要能对上；③ 高频概念：**平均值 vs 分位数（P95/P99）**——性能报告只看平均值是典型错误，这一点在后面讲指标时还会强调；④ 持续时间的设置要与测试类型匹配，别用 10 分钟的数据去论证"系统可以连续运行一个月"。

过渡：概念讲完，我们看一个具体的负载测试例子，把刚才这些参数落到场景上。

【7.11】示例：系统负载测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是负载测试的示例，我用它把前面的概念串成一条线，你跟着看一遍，就知道一次负载测试长什么样。

一次负载测试由四样东西组成：**被测系统、负载模型、施压脚本、监控与结果**。被测系统要部署在接近真实的配置上；负载模型就是刚才讲的三个要素加两种模式；施压脚本是把用户操作翻译成工具能执行的请求序列——登录、查列表、下单，中间夹着思考时间；监控与结果就是采集响应时间、吞吐量、错误率和各项资源指标。

结果最典型的呈现方式是一条**随并发上升的曲线**，它一般有三个阶段。**第一阶段是轻载区**：并发增加，响应时间基本不变，吞吐量线性上升——说明系统还有余力。**第二阶段是拐点附近**：吞吐量增长开始变缓，响应时间开始抬头——这是系统**接近饱和**的信号。**第三阶段是过载区**：再往上加并发，吞吐量不升反降，响应时间指数级飙升，错误率开始出现——因为队列排满了，大家互相抢资源，做无效功。

做性能测试最想要的就是**拐点那个数字**：它告诉你这套配置的"经济容量"在哪。你要报给业务方的不是"最大能撑多少"，而是"**在响应时间还达标的前提下，能撑多少**"——这两个数字往往差很远。

还有一个细节：这张图上的曲线不是一次跑出来的，是**阶梯式加压**跑出来的——每加一档，稳定跑一段，记一组数据。所以一次负载测试通常是"多次短跑"拼起来的，不是一个脚本从头跑到尾。

> **板书：**四要素：被测系统｜负载模型｜施压脚本｜监控与结果。下面画一条曲线：横轴并发、纵轴吞吐／响应时间，标三段"轻载区（线性）→ 拐点（接近饱和）→ 过载区（吞吐下降・错误率上升）"。底部写："要报的是'达标前提下的容量'，不是'最大容量'。"

提问：如果曲线在过载区"吞吐量下降"，为什么加更多人反而干得更少？

预设回答：

- 答"因为服务器忙不过来"——太笼统。具体机制是**上下文切换和锁竞争**：线程太多，CPU 时间花在切换和抢锁上，真正干活的时间变少。

- 答"因为队列满了开始拒绝"——这是一种表现（错误率上升），但吞吐下降在拒绝之前就发生了。

- 答"因为内存不够开始换页"——也可能，尤其是有大量对象堆积时。

- 追问收口：所以过载区的"吞吐下降"是**系统在做无效功**的信号，这时候看 CPU、上下文切换、GC 时间，通常能找到证据。

> **易错点 / 考点：**① 三阶段曲线的名称与特征要能画能说；② 高频错答："并发越高吞吐越高"——只在线性区成立；③ 报容量时必须带上**响应时间前提**，只报并发数是耍流氓；④ 阶梯加压的做法要记住：**每档加压后要稳定一段时间再采样**，否则采到的是爬坡过程的数据。

> **过渡：**负载测试只是性能测试的一种。课件把性能测试按目的分成了好几类，我们先看最基础的四种。

【7.12】性能测试类型（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页讲性能测试的四种基础类型，这四种是**按目的分的**，你听完会发现它们各自对应 7.7 那三个目标里的一段。

第一种，**性能基准测试**：在系统标准配置下获得有关的性能指标数据，作为将来性能改进的**基线（Baseline）**。基线这个词很关键——它的作用是"**以后有得比**"。你今天测出首页 800 毫秒，下周优化后测出 500 毫秒，这 300 毫秒的改进是靠基线才看得见的。没有基线，所有优化都是"感觉快了"。

第二种，**性能验证测试**：验证系统是否达到**事先已定义**的系统性能指标、能否满足性能需求。注意"事先已定义"这四个字——指标必须是测试之前就写下来的，不能测完了再往回找。这一条对应 7.7 的第二个目标"验证是否达标"。

第三种，**性能规划测试**：在多种特定环境下，获得不同配置的系统性能指标，从而决定系统部署的**软硬件配置选型**。说白了就是"买几台机器、什么规格"。它的做法是"换配置、重复测、比结果"：4 核 8G 能撑 500 并发，8 核 16G 能撑 1100，那为了明年翻倍的业务量，该买哪一种？这就是规划测试要回答的。

第四种，**容量测试**。课件有一句很重要的话：容量测试可以看作性能测试的一种，**因为系统的容量可以看作是系统性能指标之一**。容量是什么？是"这套系统最多能承载多少用户、多少数据、多少并发"。它和验证测试的区别在于：验证测试问"达不达标"，容量测试问"**上限在哪**"。

四种类型的关系我总结一句：**基准是"记住现在"，验证是"对照标准"，规划是"决定买什么"，容量是"摸到天花板"。**

> **板书：**四行四类：基准（Baseline，记住现在，供以后比较）｜验证（对照事先定义的指标）｜规划（换配置重复测 → 软硬件选型）｜容量（容量也是一种性能指标 → 摸上限）。右侧画一条数轴："现在 → 标准 → 选型 → 上限"。

> **提问：**一个项目从没做过性能测试，第一次做，你会先做哪一种类型？为什么？

> **预设回答：**

- 答"先做验证测试，看达不达标"——可以，但你得先有指标；而指标往往要靠基准数据来定。

- 答"先做基准测试"——更稳。先把现状记下来，后面所有判断都有参照。

- 答"先做容量测试，看能撑多少"——顺序偏了。容量测试成本最高（要压到崩），在还不知道系统基本情况时做，容易白烧时间。

- 追问收口：合理的顺序是**基准 → 验证 → （按需）容量／规划**，基准是所有后续比较的地基。

> **易错点 / 考点：**① 四种类型的**名称＋目的**必须成对记住，简答常考；② 基准测试的关键词是 **Baseline（基线）**，答题要写出来；③ 容量测试与验证测试的区别是高频混淆点：一个是"达标吗"，一个是"上限在哪"；④ 课件明确说"容量可以看作性能指标之一"，所以容量测试**归入性能测试**，不要答成独立门类。

> **过渡：**上面四种是"常规动作"，接下来三种是"加码动作"——压力、渗入、峰谷，专门用来把问题逼出来。

【7.13】性能测试类型（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲三种"加码"的类型：压力测试、渗入测试、峰谷测试。它们的共同点是——**不是验证，是逼供**。

第一种，**压力测试（Stress test）**：模拟实际应用的软硬件环境及用户使用过程的**高负载、异常负载、超长时间运行**，从而**加速系统崩溃**，以检查程序对异常情况的抵抗能力，找出性能瓶颈、不稳定等问题。注意三个词：高负载、异常负载、超长时间；也注意"**加速系统崩溃**"——目的不是证明系统很稳，而是**用最短时间把它逼到崩，看它怎么崩、崩在哪**，模拟的是"最坏的一天"。

第二种，**渗入测试（soak test）**：通过长时间运行，使问题**逐渐渗透出来**，从而发现内存泄漏、垃圾回收（GC）或系统的其他问题，以检验系统的**健壮性**。关键词是"长时间"。很多问题在 10 分钟里根本不会出现：内存一点点涨、连接池一点点漏、临时文件一点点堆。soak 就是把这些**缓慢过程压缩到可观测的时间内**。

第三种，**峰谷测试（peak-rest test）**：采用**高低突变**加载方式进行——先加载到高水平的负载，然后急剧降低，稍微平息一段时间，再加载到高水平的负载，重复这个过程。课件评价它**容易发现问题的蛛丝马迹，最终找到问题的根源**。为什么突变能暴露问题？因为系统在"扩"和"缩"时最脆弱：缓存刚清空就来了大流量（缓存击穿）、连接刚回收就要重建（连接风暴）。

三种类型的分工对照：**压力测试问"极限在哪、怎么崩"；渗入测试问"跑久了会不会坏"；峰谷测试问"忽高忽低扛不扛得住"**。对应三种真实风险：大促、长跑、潮汐流量。

补充一句区分：压力测试和渗入测试都涉及"长时间"，但目的不同——压力测试是**用超长时间去压**，渗入测试是**用正常负载去泡**。一个求崩，一个求漏。

板书：三行：压力（高负载・异常负载・超长时间 → 加速崩溃 → 找瓶颈）｜渗入 soak（长时间 → 问题逐渐渗透 → 内存泄漏・GC・健壮性）｜峰谷 peak-rest（高低突变・反复 → 暴露扩缩瞬间的脆弱 → 找到根源）。底部三句对照："极限在哪／跑久了坏不坏／忽高忽低扛不扛得住"。

提问：一个系统上线三个月后越来越慢、每周要重启一次，你应该用三种类型里的哪一种去复现它？

预设回答：

- 答"压力测试，压到崩"——压到崩能看到崩溃点，但复现不了"三个月慢慢变慢"这个过程。
- 答"渗入测试，长时间跑"——对。这正是 soak 的用途：用几十小时的长跑，把"三个月"的累积过程压缩出来。
- 答"峰谷测试"——峰谷测的是突变响应，不是累积恶化。
- 追问收口：选类型的判断标准是"**线上问题的时间形态**"——累积型选 soak，尖峰型选峰谷，崩溃型选压力。

易错点 / 考点：① 三种类型的**定义关键词**要能默写：压力＝高负载／异常负载／超长时间＋加速崩溃；soak＝长时间＋内存泄漏／GC；peak-rest＝高低突变＋重复；② 高频错答："压力测试就是并发调大"——它的手段是三种负载形态（高、异常、超长），不只是加并发；③ soak 与压力的区分（求崩 vs 求漏）是判断题常考点；④ 峰谷测试容易被忽略，但它是"潮汐业务"（早晚高峰）的标配。

【思政落点 3】：渗入测试最像我们做事的方式——**它不靠聪明，靠熬**。跑一夜、跑一周，盯着曲线看有没有那一点点上翘。这件事没有任何技巧可以替代时间：你不可能用 10 分钟的测试证明"跑一个月没问题"。它教给人的是一种很朴素的品质：**愿意为"看不见的问题"付出等待**。学习也一样，背完一章就能考试，但真正长本事的，是那些需要长期坚持、短期看不到回报的事。请记住：**有些质量只能靠耐心换，急不来。**

过渡：压力测试具体在做什么？我们看一个示例。

【7.14】示例：压力测试（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是压力测试的示例，我重点讲"压力测试要观察什么"，因为很多同学做完压测只会说一句"崩了"。

压力测试的执行方式，是把负载一路加上去，直到系统出现异常，然后**记录它在每一步的表现**。所以它产出的不是"一个数字"，而是**一组临界值和一连串现象**。你要观察四件事。

第一，**崩溃点在哪**：多少并发、多少 TPS 时开始出错——错误是超时、是 500、还是连接被拒。不同的错误形态指向不同的瓶颈：超时多半是下游慢，500 多半是异常没兜住，连接被拒往往是连接池或文件句柄耗尽。

第二，**崩溃前有没有征兆**：响应时间是慢慢爬上去的，还是突然跳的？慢慢爬说明资源在逐步消耗，突然跳往往撞到了某个硬上限（比如线程池满、GC 停顿）。

第三，**崩溃后能不能自愈**：把压力撤掉，系统是几秒钟恢复正常，还是彻底僵死必须重启？这一条最能反映健壮性，也是运维最关心的。**能自愈的系统，事故是"抖动"；不能自愈的系统，事故是"停机"。**

第四，**异常负载下的行为**：课件特意写了"异常负载"——不只是量大，还包括**形态异常**：请求体超大、参数畸形、依赖服务超时、网络抖动。这些才是线上真正让人措手不及的情况。

提醒一句：压力测试是**在受控环境里主动把系统搞崩**，所以一定要在隔离环境做，别在共享的测试环境里跑，否则同事的测试全被你带走。这不是胆小，是专业。

> **板书：**四个观察点：① 崩溃点（多少并发／什么错误形态）② 崩溃前征兆（缓爬 vs 突跳）③ 崩溃后能否自愈（抖动 vs 停机）④ 异常负载行为（超大・畸形・依赖超时・网络抖动）。底部："压力测试产出的是一组临界值和现象，不是一个数字。"

提问：压力测试把系统压挂了，撤掉压力后它自己恢复了——这算通过还是不通过？

预设回答：

- 答"不通过，挂了就是挂了"——要看测试目的。压力测试本来就是要它挂，挂是预期结果。

- 答"通过，能自愈就行"——也不能这么简单。要看恢复时间和恢复后的一致性：数据有没有丢、连接有没有泄漏、恢复后性能是否回到原水平。

- 答"要看恢复时间和数据完整性"——最完整。**自愈 + 快速 + 数据无损**，三条都满足才算健壮。

- 追问收口：所以压力测试的结论永远是"在 X 条件下崩溃，表现为 Y，撤压后 Z 秒恢复"，而不是一个"通过/不通过"。

> **易错点 / 考点：**① 压力测试的目的在于**找瓶颈、验健壮性**，不在于"通过"；② "自愈"要看三个维度：时间、数据、性能是否复原；③ 常考判断："压力测试应该在共享测试环境做，省资源"——错，它会把环境打挂，影响他人；④ 记住"异常负载"包含形态异常，不只数量异常。

> **过渡：**性能问题可能出在系统的哪一层？课件给了六个层次，这就是定位问题的地图。

【7.15】性能测试的层次（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页给的是"定位地图"：一个请求从用户手指到数据库再回来，会穿过六层，性能问题可能卡在任意一层。
>
> 第一层，**Web／App 客户端**。这一层的性能问题是"用户最先感知"的：页面渲染慢、图片太大、脚本阻塞、首屏要等好几秒。它的特点是**换个差一点的手机就暴露**，所以客户端性能测试往往要在低端机上做。
>
> 第二层，**应用服务器**。业务逻辑跑在这里：线程池、JVM／运行时、GC、对象创建、锁竞争、序列化。这一层的典型症状是 CPU 高、GC 频繁、线程阻塞。
>
> 第三层，**数据库访问层**。注意它和"数据库"是分开的两层——访问层指的是连接池、ORM、SQL 生成、事务边界。很多"数据库慢"其实是访问层的问题：连接池太小导致排队、事务开得太大导致锁等待、ORM 生成了 N+1 查询。
>
> 第四层，**RDBMS 数据库**。这一层看的是索引、执行计划、锁、慢查询、表的数据量。数据量是最容易被忽略的变量：一万行和一亿行，同一个 SQL 的命运完全不同。
>
> 第五层，**端口连接**。这一层很底层但极其常见：TIME_WAIT 堆积、端口耗尽、连接复用没做、keep-alive 配置不当。它的典型症状是"服务本身很闲，但客户端连不上"。
>
> 第六层，**网络通信**。带宽、延迟、丢包、DNS 解析、TLS 握手。跨地域部署的系统，这一层经常是主要矛盾。
>
> 为什么要把层次讲这么细？因为**性能问题的定位，本质是"逐层排除"**：先看客户端是不是自己的问题，再看服务端处理时间，再看是不是等数据库，最后看网络。你心里有这六层，排查才有顺序；没有这张地图，就

只能瞎猜。

顺便记住一个判据：**"服务器 CPU 不高但用户很慢"，问题通常不在第二层**，往第五、第六层（连接、网络）或者下游依赖去找。

> **板书：**竖排六层（自上而下）：Web/App 客户端 → 应用服务器 → 数据库访问层 → RDBMS 数据库 → 端口连接 → 网络通信。每层右侧写一个典型症状（渲染慢／CPU・GC／连接池・N+1／索引・慢查询／TIME_WAIT・端口耗尽／带宽・延迟・DNS）。底部："定位＝逐层排除"。

提问：用户反馈"页面打开要 8 秒"，服务器 CPU 只有 10%，你会先查哪一层？

预设回答：

- 答"先查数据库"——可以，但要注意 CPU 低恰恰说明瓶颈可能在"等待"上，比如等 IO、等网络、等下游。

- 答"先查客户端"——合理，尤其首屏 8 秒这种量级，图片和脚本的嫌疑很大。

- 答"看是哪个环节的时间"——最专业。先把这 8 秒拆开：DNS 多久、建连多久、服务端处理多久、传输多久、浏览器渲染多久。

- 追问收口：所以下一页要讲"时间分解"——**把一个请求的耗时切成段，才知道该查哪一层**。

易错点 / 考点：① 六个层次的名称要能按顺序说出；② **数据库访问层 ≠ RDBMS**，前者是连接池／ORM／事务，后者是数据库引擎本身，这是本页最容易被合并的错；③ 高频判断："CPU 高才是性能问题"——错，很多严重问题表现为"CPU 很低但很慢"（等待型瓶颈）；④ 定位方法是逐层排除，答题时要把"分层"这个思路说出来。

过渡：六层是一张横切面。再看一张纵切面——性能测试在系统整个生命周期里怎么分布。

【7.16】全生命周期的性能测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把性能从"一次测试"提升成"一条贯穿项目始终的线"。课件给出了多个环节，我按项目推进的顺序讲。

项目一开始是**性能需求收集与定义**——性能目标在这一步就要写下来，不能等做完再补。接着是**性能建模**和**内建性能（设计中）**：把性能当设计约束，而不是事后补丁——选什么架构、要不要缓存、要不要读写分离，这些决定比任何调优都影响大。

编码阶段做**单元性能测试／代码优化**，把关卡前移到函数级：一段 O(n^2) 的代码在单元测试里就能发现，等到系统测试就晚了。然后是**性能测试**：设计、执行、采集数据，产出**性能数据**。

上线之后是**监控**和**容量管理**：7×24 的性能监控、健康检查、容量规划。课件在图上标了"性能数据"在两个环节之间来回流动——意思是**测试产出的数据要喂给监控和容量管理，监控和容量管理发现的新问题又要回流成新的性能需求**，形成闭环。

这个闭环的意义在哪？在于**性能是持续演进的，不是一次性交付的**。业务量在涨、代码在改、依赖在升级，去年达标的系统今年可能就不达标了。所以课件才有"性能需求调整修改"这个回头箭头——**需求会被现实改写，这是正常的，不是失败**。

一句话总结这一页：**性能工程是一条闭环，性能测试只是其中的一个节点，但它是最能拿出数据的那一个节点。**

板书：画一个环：性能需求收集与定义 → 性能建模／内建性能 → 单元性能测试／代码优化 → 设计・执行性能测试 → 性能数据 → 监控・容量管理 →（回流）性能需求调整修改。环中心写"经验积累"。

提问：一个项目上线后一直没做过性能监控，现在要"补性能"，你会从哪一步开始？

预设回答：

- 答"先压测一遍看看"——可以，但没有监控数据，压出来也说不清瓶颈。

- 答"先把线上监控上起来"——更稳。先知道真实负载长什么样（多少人、什么时段、哪些接口最忙），再设计压测才有的放矢。

- 答"补性能需求"——也成立，指标得先有。

- 追问收口：实操顺序是**监控 → 摸底基准 → 定需求 → 针对压测**；缺了监控，你连"真实负载模型"都画不出来。

易错点 / 考点：① 全生命周期的几个环节要能按顺序说；② 高频判断："性能测试是系统测试阶段的一项活动"——不全面，理想状态是各阶段都有对应活动；③ "内建性能"（design for performance）这个词要会解释：**在设计阶段就把性能作为约束**；④ 记住闭环里的"回流"——需求会被调整，这是性能工程的常态。

过渡：闭环的起点是需求。那性能需求到底怎么从一句"要快"变成可测的指标？

【7.17】从需求开始明确性能（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页回答一个很实际的问题：**性能指标从哪来**。课件给了四类来源、三种关注视角，最后落到一个动作——获取、分析和评审性能需求。

四类来源。第一类，**最终用户的体验，比如 2-5-10 原则**。这是最朴素也最有效的口径：2 秒以内用户觉得很快，5 秒左右还能忍，10 秒基本就关页面走人了。它不精确，但它抓住了性能的本质——**性能是用户体验的一部分，不是机房里的数字**。第二类，**商业需求，比如"比竞争对手的产品好"**。这类需求听着虚，实际很有用：它给了你一个外部参照，你要去测对手、要去做对比，才能把"好"翻译成一个具体的数字。第三类，**技术需求，比如 CPU 使用率不超过 70%**。这类是工程口径，好处是容易测、容易监控，坏处是它和用户体验没有直接对应——CPU 70% 用户可能很流畅，CPU 30% 用户也可能在等锁。第四类，**标准要求，比如数据通信延时小于 80 毫秒**。这类往往来自行业规范或合同条款，是硬约束。

然后是三种关注视角，这一条特别值得记：**用户关注系统的外部表现，如响应时间；业务部门关注系统的容量和数据吞吐量；技术团队关注系统资源占用率、或使用效率**。同一套系统，三种人问三个问题。这解释了一个常见矛盾：技术团队说"资源还很空闲"，业务部门说"高峰期撑不住"，用户说"卡"。三方都没说谎，他们看的是不同层面。

所以课件最后给了括号里的那个动作：**获取、分析和评审性能需求**。三步：先**获取**——把各方嘴里的"要快"收集起来；再**分析**——翻译成可测的指标，检查有没有互相冲突；最后**评审**——让三方坐在一张桌子上确认，别等测完了才说"我不是这个意思"。

一句话总结：**性能需求不是抄来的，是谈出来的；谈出来的东西必须写下来、签下来。**

> **板书：**四类来源：用户体验（2-5-10）｜商业需求（比对手好）｜技术需求（CPU < 70%）｜标准要求（延时 < 80ms）。三种视角：用户→响应时间；业务→容量・吞吐量；技术→资源占用・效率。底部三步："获取 → 分析 → 评审"。

提问：技术团队说"我们的服务器资源只用了 30%，性能没问题"，业务部门说"双十一肯定撑不住"。谁对？

预设回答：

- 答"技术团队对，数据说话"——数据是真的，但口径不对：30% 是**平均值**，双十一看的是**峰值**。

- 答"业务部门对，他们更懂业务"——感觉对，但业务部门给不出瓶颈在哪。

- 答"两个都对，看的是不同层面"——最准确。这正是课件那三种视角要说明的事：**资源占用率、容量、响应时间，是三个不同的问题**。

- 追问收口：所以性能需求要三方面都定：峰值吞吐、响应时间、资源上限，缺一项就会有扯皮。

> **易错点 / 考点：**① 四类来源和三种视角要能对上号，简答常考"性能需求从哪来"；② **2-5-10 原则**是外部体验口径，答题要说明它是"经验阈值"而非精确标准；③ 高频混淆：技术指标（CPU 70%）**不等于**用户感受，两者必须分别定义；④ "获取—分析—评审"三步不能漏"评审"，它是把口头需求变成书面承诺的关键。

> **过渡：**光有来源还不够，指标还得"能测"。下一页就讲什么叫可测试的性能需求。

【7.18】性能需求可测试性（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页用一个对比，讲清"什么叫可测试的性能需求"。课件给了一句话的需求，和一句改写后的需求，我们对着看。

原话是：**"每天要处理一千万笔交易"**。听起来很明确，其实**没法测**。为什么？第一，它没说"一天"是 24 小时还是 8 小时营业时间；第二，它没说交易的分布——是均匀铺开，还是集中在几个时段；第三，它没说响应时间要求——每秒处理得再多，如果每笔要等 30 秒，用户照样骂；第四，它没说峰值。**"平均"这个词在性能里是最危险的词**，因为系统从来不死在平均值上，它死在峰值上。

改写的版本是：先给一句"**清楚而量化的性能指标**"作总纲，然后具体化为"**每秒平均处理约 130 笔交易**"，再加一条"**高峰期间处理要求是平均值四倍，即 520 笔交易／秒**"。

这个改写好在哪？好在它把一句模糊的话，变成了**三个可验证的数**：平均速率、峰值倍数、峰值速率。有了这三个数，测试场景就能设计了：平时按 130 笔／秒跑稳态，峰值按 520 笔／秒跑尖峰，两条曲线都要在响应时间约束内。

顺便做个验算，帮你理解这两个数怎么来的：一千万笔摊到 86400 秒，大约是 116 笔／秒；课件给的 130 笔／秒略高，说明它按"有效服务时间"而不是整 24 小时来折算——**课件没有展开这个折算口径，你答题时按课件给的数字写就行，不要自己另算一套**。而 130 乘 4 正好等于 520，这个"四倍"是配套的。

所以"可测试"的标准是什么？我给你三条：**能定量**（有数字、有单位）、**有前提**（在什么负载、什么环境、

什么数据量下）、**可判定**（测完能明确说达标或未达标）。三条缺一条，需求就还得谈。

> **板书：**左栏写原话"每天要处理一千万笔交易"，旁边打叉，标"不可测：时间口径？分布？响应时间？峰值？"。右栏写改后："清楚而量化的性能指标 → 每秒平均约 130 笔 → 高峰为平均值四倍＝520 笔／秒"，打勾。底部三条标准："能定量・有前提・可判定"。

提问："系统要支持 1000 个并发用户"这句话，算不算可测试的性能需求？

预设回答：

- 答"算，有数字了"——有数字不等于可测。1000 并发下要求响应时间多少？错误率上限多少？跑多久？都没说。

- 答"不算，缺响应时间和错误率"——对，这是最关键的补充。

- 答"还缺负载模型"——很专业。1000 个用户带不带思考时间，压出来的结果天差地别（回想 7.9）。

- 追问收口：所以一条完整的性能需求至少四要素——**并发规模、响应时间、错误率、持续时间**，再加环境与数据量作背景。

易错点 / 考点：① "每天一千万笔"为什么不可测，四条理由要能说全；② **平均值 vs 峰值**是本节高频考点，答题一定要提峰值；③ 可测试三标准（定量、有前提、可判定）可作简答模板；④ 案例题给一句模糊需求让你改写，记住补四个要素：并发、响应时间、错误率、持续时间。

过渡：需求定了、指标量化了，接下来要在设计阶段就把性能"建"进去——这正是架构评审要干的事。

【7.19】架构设计评审（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲架构设计评审，它在性能工程里的位置很特殊：**这一步不运行一行代码，却能决定系统 80% 的性能上限**。

为什么？因为架构决定了"物理可能性"。单体应用再优化也顶不住千万级并发，因为它的扩展方式是"换更贵的机器"；水平扩展的架构一开始就留了加机器的口子。数据库不做读写分离，读请求再多也只能压一台库；缓存层没有，所有请求都直击数据库。这些选择一旦定下来，后面写代码再漂亮也翻不了盘。

架构评审要看什么？我给它归成四问。**第一问：数据放哪、怎么涨？**数据量增长曲线决定要不要分库分表、

要不要冷热分离。**第二问：请求怎么分流？**有没有缓存、有没有限流、有没有降级、有没有异步化。**第三问：瓶颈会移向哪里？**这是最体现水平的一问——你把数据库优化好了，瓶颈就跑到应用服务器；把应用加机器了，瓶颈就跑到网络或连接数。**架构评审就是要提前知道"下一个瓶颈在哪"**。**第四问：可测性够不够？**有没有埋点、有没有监控指标、能不能在测试环境复现生产规模——这一条最容易被忽略，却直接决定你后面能不能做性能测试。

还有一条经验：架构评审里最有效的问题不是"这样行不行"，而是"**这样设计，十倍流量时会先死在哪**"。这个问题没人能含糊过去，它逼着设计者把链路走一遍。

记住这一页的定位：**性能问题最好的修法，是在图纸上修。**

> **板书：**中间写"架构设计评审（不跑代码，定 80% 上限）"。四问竖排：① 数据放哪・怎么涨 ② 请求怎么分流（缓存・限流・降级・异步）③ 瓶颈会移向哪里 ④ 可测性够不够（埋点・监控・可复现生产规模）。底部一句："十倍流量时会先死在哪？"

提问：一个系统架构评审时没人提性能，后面能不能靠调优补回来？

预设回答：

- 答"能，调优就够了"——小问题能补，架构级问题补不了。比如没做水平扩展，调优只能把单机榨到极限。

- 答"不能，架构决定天花板"——抓住了关键。调优是在天花板下面找空间，天花板本身是架构定的。

- 答"要看问题大小"——更准确：**参数级问题调优能救，结构级问题只能改架构**。

- 追问收口：所以评审时要区分"这是参数问题还是结构问题"，别把结构问题留给调优。

易错点 / 考点：① 架构评审属于**静态性能测试**（回想 7.4 的静态/动态之分）；② "瓶颈会转移"这个概念要能举例；③ 常考判断："性能优化主要在编码和调优阶段"——不完整，架构阶段的影响最大；④ 评审四问里"可测性"最容易被漏，答题时补上能加分。

过渡：架构定了，代码还得写。下一页看代码评审里那些最常见的性能坑。

【7.20】代码评审 - 性能问题（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲代码评审中的性能问题，也就是"在代码里用眼睛找慢"。课件给了两个抓手：一个是具体的例子，

一个是清单式的总结。

先说那个具体的例子：**使用 String 的性能是最差**。这句话要放在语境里理解——它说的是**字符串频繁拼接的场景**。在 Java 里字符串是不可变的，每次 `+` 都会创建新对象，拼接一万次就是一万个对象，内存和 GC 压力全上来了；换成 StringBuilder 就只维护一个可变缓冲区。这类问题的特点是：**单次看不出，循环里要命**。

再说清单：课件提到"**35 个 Java 代码性能优化总结**"。我不逐条念，但把它们的共同分类讲给你，你以后看任何一份优化清单都能归类。这些点大致落在五类里：**第一类，对象创建**——循环里 new、装箱拆箱、正则重复编译；**第二类，集合使用**——没定初始容量导致反复扩容、用错数据结构（该用 Map 的地方用了 List 做查找）；**第三类，IO 与数据库**——循环里查数据库（N+1 问题）、没批处理、没关流；**第四类，并发**——锁的粒度太大、无谓的同步、线程池参数不合理；**第五类，字符串与序列化**——就是刚才那个 String 拼接，还有频繁的 JSON 转换。

为什么代码评审能抓性能问题？因为它**成本极低**：不用搭环境、不用造数据、不用跑压测，两个人对着代码看半小时，可能就发现一个会在生产上炸的循环查库。而且它**抓得早**——写代码的当天就能改，比上线后改便宜几个数量级。

但也要说清它的局限：代码评审只能发现"**能看出来**"的问题，看不出"这段逻辑在 1000 并发下会怎样"。所以它和压测是互补的：**评审管"明显不该这么写"，压测管"这么写到底行不行"**。

板书：左侧例子：String 拼接 → 每次 + 都新建对象 → 循环里 O(n²) 级开销 → 换 StringBuilder。右侧五类清单：对象创建｜集合使用（容量・数据结构）｜IO 与数据库（N+1・批处理・关流）｜并发（锁粒度・同步・线程池）｜字符串与序列化。底部："评审管'明显不该这么写'，压测管'这么写到底行不行'"。

提问：代码里有一个循环，每次循环都查一次数据库，功能完全正确，这算不算缺陷？

预设回答：

- 答"不算，功能没问题"——从功能测试角度确实不算，但性能上它是典型的 N+1 查询，数据量一大就崩。

- 答"算性能缺陷"——准确。这类问题在代码评审里就该拦下来。

- 答"要看循环多少次"——很关键。循环 3 次无所谓，循环 3000 次就是灾难。**性能问题永远和规模绑在一起**。

- 追问收口：所以说性能缺陷的判断要带规模：**"在什么数据量下会出问题"**，而不是简单地说"这样写不好"。

易错点 / 考点：① "使用 String 的性能是最差"要能说清**前提是频繁拼接**，不能绝对化；② 五类归类比死记 35 条更有用，答题时给出分类框架得分更高；③ 高频判断："代码评审属于动态测试"——错，它是静态测试；④

性能缺陷必须**带规模描述**，这一点在缺陷报告里常被扣分。

过渡：需求有了、架构评审过了、代码也看过了，接下来才是真正动手：设计性能测试场景并执行。

【7.21】性能测试设计与执行（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲性能测试的设计与执行，课件把要素归成六项环境和资源条件，我逐项讲——任何一项不对，测出来的数字都不能用。

第一项，网络及其配置。带宽多大、延迟多少、有没有经过代理和防火墙。你在公司内网压出来的响应时间，和用户在 4G 上感受到的，完全不是一个东西。

第二项，服务器硬件选型。几核 CPU、多大内存、什么磁盘。SSD 和机械盘在数据库场景下差一个数量级。这一项必须**和生产配置对应**，否则结论无法外推。

第三项，被测试系统部署。几个实例、怎么分布、前置有没有负载均衡、有没有缓存服务。这里最容易出的错是"测试环境只部署了一个实例，生产是八个"，那么你测出的容量只有真实的八分之一。

第四项，数据库数据量。这一项最容易被忽略，也最坑人：测试库一百万行，生产库一亿行，同一个 SQL 在测试库走索引 10 毫秒，在生产库全表扫 10 秒。**数据量是性能测试的隐形变量**，必须专门造数据。

第五项，测试机器池。施压机自己会不会成为瓶颈？一台压测机跑几千并发，它的 CPU 和网络可能先满，那你测到的其实是"压测机的极限"。所以大规模测试要用**多台施压机**——就是后面要讲的分布式压测。

第六项，SUT 监控。SUT 就是被测系统（System Under Test）。监控要覆盖三层：**施压侧**（发了多少请求、成功率）、**被测系统侧**（CPU、内存、GC、线程、连接池）、**依赖侧**（数据库、缓存、消息队列）。只监控一层，等于只知道"疼"，不知道"哪儿疼"。

课件还画了一张部署示意，把施压机池、被测系统的 Web 服务器、应用服务器、数据库服务器分开摆，中间放一个分析器。这张图要你看出的是**角色分离**：施压的、被测的、观测的，三者在物理上要尽量分开，否则互相干扰、数据失真。

板书：六项竖排：网络及配置｜服务器硬件选型｜被测试系统部署｜数据库数据量｜测试机器池｜SUT 监控。监控三层横排：施压侧（请求・成功率）｜被测系统侧（CPU・内存・GC・线程・连接池）｜依赖侧

（DB・缓存・MQ）。底部："角色分离：施压 / 被测 / 观测"。

提问：为什么性能测试环境不能直接用生产的缩水版，比如"生产八台机器，测试就一台"？

预设回答：

- 答"可以啊，按比例换算"——换算不成立。有些瓶颈是**非线性**的：单机连接数上限、单库锁竞争，都不是除以 8 能推出来的。

- 答"不行，测出来的容量只有八分之一"——方向对，但问题更大：一台机器测不出负载均衡、缓存命中率、网络拓扑这些**只在多机时才出现**的行为。

- 答"至少要多机部署同样的拓扑"——最接近专业做法：**拓扑要一致，规模可以缩，但数据量和并发要按目标折算**。

- 追问收口：所以课件把"部署"和"数据量"单列成两项，就是在提醒你：**环境失真，结论全废**。

易错点 / 考点：① 六项要素要能列全，尤其"数据库数据量"和"测试机器池"这两项最容易被漏；② 高频错答："压测环境可以缩水"——拓扑要一致、数据量要接近、并发按目标设定；③ 施压机自身瓶颈是典型的**测试失效**原因，考试常用来出案例题；④ SUT 这个词要会解释：被测系统。

过渡：环境搭好，脚本跑起来了，接下来的关键是"看清楚时间花在哪"。

【7.22】测试执行和监控（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲执行与监控，核心动作是**把一个请求的耗时切成段**。课件列了一串时间名词，我按"从用户按下回车到看到结果"的顺序串一遍。

第一步，**DNS 时间**：把域名解析成 IP 要多久。别小看它，DNS 配错或者解析慢，用户会先卡两秒，而你的服务器一点毛病都没有。

第二步，**网络时间**：请求从客户端发到服务器要多久，以及响应从服务器传回客户端要多久。跨地域、跨运营商的系统，这一段经常是大头。

第三步，**服务时间**：服务器内部处理这次请求花了多久。这是最需要深挖的一段，因为它是拆得最细的一段。

服务时间里还能再拆：**应用处理返回时间**——业务代码自己算的部分；**数据库时间**——等数据库返回的部分。而数据库时间往往又占服务时间的大半，所以"应用慢"十次有八次其实是"数据库慢"。

第四步，**数据返回时间**：响应体从服务端推到客户端的时间，和响应大小成正比。返回一个 10 MB 的 JSON，再快的服务器也救不了用户体验。

第五步，**客户端时间**：浏览器或 App 拿到数据之后，渲染、执行脚本、加载图片的时间。前端同学最熟这一段，服务端同学最容易把它算到别人头上。

为什么要拆这么细？因为**性能问题的定位，就是找"哪一段最长"**。总响应时间 8 秒，可能是 DNS 0.1 秒、网络 0.3 秒、服务 7 秒、返回 0.2 秒、渲染 0.4 秒——那问题就在服务端，而且是服务端里的数据库。不拆，你只能对着"8 秒"发愁。

这一页还有一个方法论上的提醒：**监控要在测试执行的同时做**，不能等跑完了再补。因为性能问题是**瞬时现象**——你不在那一刻采数据，它就永远消失了。

> **板书：**一条时间轴（左＝用户点击，右＝看到结果）：DNS 时间 → 网络时间（请求）→ 服务时间（＝应用处理返回时间 ＋ 数据库时间）→ 数据返回时间 → 客户端时间。底部："定位＝找最长的那一段；监控必须与执行同时进行"。

提问：总响应时间 3 秒，你怎么判断是"服务端慢"还是"网络慢"？给你什么数据你能判定？

预设回答：

- 答"看服务端日志里的处理时间"——对，这是最直接的证据：服务端自己记的处理耗时如果是 0.2 秒，那 2.8 秒就在路上或客户端。

- 答"ping 一下看延迟"——能作参考，但 ping 走的是 ICMP，和 HTTP 的真实链路不完全等价。

- 答"抓包看时间戳"——最严谨，能精确切出各段耗时。

- 追问收口：所以要做性能测试，**埋点和分段计时是基础设施**，不是可选项。

易错点 / 考点：① 五个时间段的名称与顺序；② **服务时间 ⊃ 应用处理时间 ＋ 数据库时间**，这个包含关系常考；③ 高频错答："响应时间长就是服务器慢"——DNS、网络、返回体大小、客户端渲染都可能是元凶；④ 记住"必须在执行同时监控"，事后补数据是无效的。

过渡：这么多指标要靠人动手采是不可能的，所以我们需要工具。看看课件给了哪些。

【7.23】性能测试工具（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页列工具，我不逐个念版本号，我讲**怎么分类、各管什么**，这样你以后遇到新工具也知道往哪放。

课件把工具分成两大类：**客户端性能测试工具**和**服务器端性能测试工具**。这个分法对应我们前面讲的六个层次的前两层——客户端和服务器。

客户端这一组，课件列了 Google Lighthouse、PerfDog、Monkey／Monkeyrunner、mobileperf、Pyroscope、MemoryLeakDetector 等。它们关心的是"用户手上那台设备"：启动耗时、页面渲染、帧率、内存、卡顿、流量、耗电。几个名字要分清：**Lighthouse 面向 Web 页面质量评估**；**Monkey／Monkeyrunner 是 Android 稳定性压测工具**，用随机事件流撞崩溃；**PerfDog 是移动端性能采集工具**，下一节专门看；**Pyroscope 偏向性能剖析（profiling）**；**MemoryLeakDetector 查内存泄漏**。

服务器端这一组，课件列了 JMeter、LoadRunner、Webload、Gatling、k6、Vegeta、Locust、Skywalking 等，可再分两类：**施压类**（JMeter、LoadRunner、Gatling、k6、Vegeta、Locust——造并发、发请求）和**观测类**（Skywalking 这类 APM，看链路、看耗时分布）。**两者必须配合**：只有施压，你知道系统崩了；只有观测，你不知道压力从哪来。

选型原则：**先用开源跑通流程，再考虑商业工具**。JMeter 免费、生态大、插件多，性价比最高，所以课件后面专门讲；LoadRunner 是老牌商业工具，脚本强、报表成熟。

板书：左右两栏。左栏"客户端"：Lighthouse（Web 质量）｜PerfDog（移动端采集）｜Monkey/Monkeyrunner（Android 稳定性）｜mobileperf｜Pyroscope（剖析）｜MemoryLeakDetector（泄漏）。右栏"服务器端"：施压类 JMeter・LoadRunner・Webload・Gatling・k6・Vegeta・Locust；观测类 Skywalking（APM）。底部："施压 ＋ 观测，缺一不可"。

提问：如果只允许你装一个服务器端工具，你选哪个？为什么？

预设回答：

- 答"JMeter"——最常见也最合理：开源、协议支持广、有 GUI 和 CLI、能分布式、插件多。

- 答"LoadRunner"——能力更强，但商业授权成本高，学习和部署重。

- 答"Locust，能写 Python 脚本"——很好，脚本即代码，适合研发自测。

- 追问收口：选型要看三件事——**协议支持、能否分布式、团队会不会用**；工具再强，团队用不起来等于没有。

> **易错点 / 考点：** ① 两类工具的划分与代表产品要能对上；② **施压类 vs 观测类**的区分是加分点，很多同学只会答"性能测试工具就是 JMeter"；③ 高频混淆：Monkey 是**稳定性**压测（随机事件），不是性能指标采集；④ 课件列的具体工具名字要能认，考试可能出"下列哪个不属于服务器端性能测试工具"这类选择题。

> **过渡：** 客户端工具里，课件单独拎出一个来讲——PerfDog。

【7.24】客户端 PerfDog（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页专门讲 PerfDog，一个移动端性能测试工具。它解决的问题很具体：**App 在真机上到底卡不卡、费不费电、占多少内存**。
>
> 为什么客户端性能要单独拿工具测？因为客户端的资源和服务器完全不同：手机的内存以 GB 计、CPU 会降频、电量有限、网络是移动网络还会切换。你用服务器那一套指标（TPS、并发数）根本描述不了客户端的问题。客户端的核心指标是另外一套：**启动耗时、页面帧率（FPS）、内存占用、CPU 占用、流量、电量、卡顿次数**。
>
> PerfDog 的工作方式很直白：**把手机用 USB 连到电脑上，工具在电脑侧采数据、出曲线**。好处是不用改 App 代码、不用埋点，插上就能测。课件给的两条参考信息是：**教材对应的页码范围，以及官方网站**——具体的安装步骤、界面操作、导出报表，以教材和官网文档为准，那是会随版本变化的细节，课上不逐条念，你按教材做实验就行。
>
> 我要提醒的是**用法上的三个要点**，这三条比记界面按钮重要得多。**第一，一定要用真机**，模拟器跑出来的帧率和内存和真机差很远。**第二，一定要用低端机测一遍**，高端机上流畅不代表用户流畅——性能测试要测的是"最差的那台设备"。**第三，场景要贴近真实使用**，比如冷启动、页面来回切换、列表快速滑动、弱网下操作，这些场景才会暴露问题。
>
> 还有一个容易被忽略的对照原则：**同一个操作，改前和改后各测一遍，才有意义**。单看一个"启动耗时 1.8 秒"没法判断好坏，得看基线是多少、目标是多少。

> **板书：** 中间写"PerfDog＝移动端真机性能采集"。左侧指标清单：启动耗时｜FPS｜内存｜CPU｜流量｜电量｜卡顿。右侧三个用法要点：真机（不用模拟器）｜低端机必测｜场景贴近真实（冷启动・切页・快滑・

> 弱网）。底部："改前改后各测一遍，才有对照"。

> **提问：**App 在你的手机上很流畅，能不能说明客户端性能没问题？

> **预设回答：**

- 答"能，我用着挺顺"——你自己的手机往往是中高端机，而且存储、缓存都在"熟"的状态。

- 答"不能，要用低端机测"——对。低端机的内存和 GPU 是短板，很多问题只在它上面出现。

- 答"还得看弱网和长时间使用"——补得好，弱网下的加载策略、用久了的内存增长，都是高发问题。

- 追问收口：客户端性能测试的原则和服务器一样——**测最差的条件，不是最好的条件**。

> **易错点 / 考点：**① 客户端核心指标和服务器端指标**不能混用**，这是本节最容易被合并的错；② 真机 vs 模拟器的差别是高频判断点；③ 常考："客户端性能测试需要改代码埋点吗"——不一定，PerfDog 这类工具外部采集即可，但精细的业务埋点仍需要开发配合；④ 记住对照原则：**没有基线的客户端指标，只是一串数字**。

> **过渡：**工具装上、脚本跑起来，接下来是整章最难的部分——看懂这些数据。

【7.25】性能测试工具 JMeter（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页进入 JMeter，先给它一个整体定位：**JMeter 是一个开源的、用 Java 写的、以"施压"为核心的工具**，它的官方站点课件也给了，你去下载、查文档都在那里。
>
> 它的核心概念只有七个，我按"从外到内"的顺序讲，你把这七个词记住，JMeter 就通了。
>
> **测试计划（Test Plan）**：最外层的容器，一次压测的全部内容都装在里面，相当于一个"项目"。
>
> **线程组（Thread Group）**：**最关键的元件**。它定义"多少个用户、怎么并发、跑多久"。线程数就是虚拟用户数；Ramp-Up 是"用多少秒把用户全放出来"；循环次数或持续时间决定跑多久。**性能测试的负载模型，主要就写在线程组里**。
>
> **取样器／请求（Sampler）**：真正发请求的元件——HTTP 请求、JDBC 请求、FTP 请求等。它决定"你去打谁"。
>
> **逻辑控制器（Logic Controller）**：控制请求的执行顺序和条件——循环、如果、事务、随机。它决定"请求

按什么顺序组合"。

断言（Assertion）：判断响应对不对。这一条最容易被忽略：**没有断言的压测，只统计了"有没有回包"，没统计"回得对不对"**。一个返回 500 的接口，响应时间可以非常快——你要是没断言，报告会漂亮得像假的。

前置／后置处理器：在请求前后做加工，比如前置准备参数、后置提取响应里的 token 传给下一个请求。

监听器（Listener）：收集和展示结果——查看结果树、聚合报告、响应时间曲线。注意它负责"看"，不负责"压"。

一句话把这七件串起来：**测试计划装线程组，线程组里放控制器和取样器，取样器外面挂断言和处理器，结果交给监听器。**

> **板书：**七元件的层级图：测试计划 → 线程组（线程数・Ramp-Up・循环/时长）→ 逻辑控制器 → 取样器（HTTP／JDBC...）→ 断言／前置・后置处理器；旁挂监听器。底部："线程组定负载，取样器定目标，断言定对错"。

提问：一次压测跑完，报告显示"平均响应时间 30 毫秒、错误率 0"，能不能说明这个接口很好？

预设回答：

- 答"能，又快又没错"——先问一句：**你加断言了吗**？很多情况下"错误率 0"是因为工具只判断了 HTTP 层有没有响应。

- 答"不能，得看有没有断言"——对。业务失败但 HTTP 200 的情况非常常见。

- 答"还得看并发是多少"——很关键。10 个并发的 30 毫秒，和 1000 个并发的 30 毫秒，含义完全不同。

- 追问收口：所以读压测报告先看三个前提——**负载多大、断言有没有、跑了多久**。

易错点 / 考点：① 七个元件的名称与职责要能对上，简答高频；② **线程组是负载模型的载体**，这一点必须会说；③ 高频错答："监听器会影响性能所以不能加"——准确说法是 **GUI 模式下的监听器开销大，正式压测应用 CLI 模式并简化监听器**；④ 断言的作用是判断业务正确性，不是判断性能。

过渡：七个元件在界面上是怎么组织的？我们看它的目录结构。

【7.26】JMeter 构成（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 JMeter 的目录结构，也就是"这七类元件在实际界面里长在哪"。我们按树形一层层看。

最上面是**工作台（Workbench）**，里面放的是**非测试元件**——HTTP 代理服务器、录制脚本用的东西。它们不参与压测，只帮你**生成**脚本，所以在工作台里。

往下是**测试计划（Test Plan）**，真正的测试内容都挂在它下面。

测试计划下面最主要的节点是**线程组（Thread Group）**。线程组里有**线程（Thread）**的概念——一个线程就是一个虚拟用户。线程组还可以设 **setup** 和 **teardown**：setup 在线程组开始前跑一次（比如登录拿 token），teardown 在结束后跑一次（比如清理数据）。这两个是**每个线程组一套**，注意别和后面的"前置处理器"混了。

除了线程组，测试计划下还能挂 **Test Fragment（测试片段）**——一段可以被复用的脚本，本身不执行，要被别的元件调用才跑；还有**远程运行控制器**，用来控制分布式压测的从机；以及**其它**杂项。

线程组内部才是我们上一页讲的那套元件：**逻辑控制器、采样器、配置元件、定时器、前置处理器、后置处理器、断言、监听器**。这里多了两个上一页没提的：**配置元件**——比如 HTTP 请求默认值、CSV 数据文件设置、Cookie 管理器，它给请求提供公共配置；**定时器**——控制请求之间的间隔，**思考时间就是在定时器里设的**。

工作台里还有几个辅助工具值得记：**Log View**（看运行日志）、**Templates**（模板，可以直接套一个现成的测试计划，比如录制模板）、**函数助手**（生成随机数、时间戳这类函数，写脚本时很常用）。

这张图要你记住的是**层级关系**：工作台管录制，测试计划管全局，线程组管负载，元件管细节。

> **板书：**树形图：JMeter → 工作台（HTTP 代理・脚本录制・Log View・Templates・函数助手）｜测试计划 → 线程组（Thread・setup・teardown）／Test Fragment／远程运行控制器／其它。线程组内：逻辑控制器・采样器・配置元件・定时器・前置处理器・后置处理器・断言・监听器。底部："工作台管录制，测试计划管全局，线程组管负载"。

提问：思考时间在 JMeter 里设在哪？如果你设错了地方，会出什么问题？

预设回答：

- 答"定时器"——对。常用的是固定定时器和随机定时器（模拟真人的随机间隔）。

- 答"在线程组里设延迟"——线程组的延迟只影响**启动时刻**，不影响请求之间的间隔。

- 答"用断言控制"——断言是判断结果用的，和节奏无关。

- 追问收口：这也说明一件事——**JMeter 的元件是按职责分层的**，你把职责搞清，就知道该往哪儿配。

> **易错点 / 考点：**① 工作台放的是**非测试元件**（录制用），这是常考判断；② **setup／teardown 线程组** 和 **前置／后置处理器** 的区别：前者在测试计划层面跑一次，后者在每个请求前后跑；③ 配置元件与前置处理器容易混：配置元件提供**公共配置**，前置处理器做**请求前的加工**；④ Test Fragment 不独立执行，必须被引用。

> **过渡：**元件讲完，看它到底能打哪些协议——这决定了它能测多少种系统。

【7.27】JMeter 支持的协议（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页很短，但信息量不小：JMeter 支持哪些协议。我按"用得多少"排一遍，你按顺序记。
>
> 第一，**Web——HTTP、HTTPS**。这是绝对主力，绝大多数性能测试都是打 HTTP 接口。
>
> 第二，**SOAP** 和 **REST Webservices**。企业系统里这两类接口很常见，尤其是和外部系统对接的时候。
>
> 第三，**FTP**。文件上传下载场景。
>
> 第四，**Database via JDBC**。这一条很重要：JMeter 可以**直接连数据库发 SQL**，不经过应用层。它的用途有两个——一是单独测数据库的性能（比如一个复杂查询要多久），二是给压测**造数据**或**校验数据**。课件后面讲实验时会专门要求"会使用 JDBC 连接预处理"，说的就是这个。
>
> 第五，**LDAP**。目录服务认证，多见于企业统一登录。
>
> 第六，**JMS**。消息队列，测 MQ 的收发性能。
>
> 第七，**Mail——SMTP**。邮件服务。
>
> 第八，**JSR223**。这一条容易被低估：它允许你用脚本语言（比如 Groovy、JavaScript）写自定义逻辑，**JMeter 的能力边界基本由它决定**——协议不支持？用 JSR223 自己拼。
>
> 第九，**TCP**、**Java Objects** 等。TCP 层面可以测自定义协议；Java Objects 用于 Java 序列化的远程调用。

这张清单的教学意义是：**JMeter 不只测网页**。你要测一个老系统的自定义 TCP 协议、要测一个消息队列、要压一个数据库，它都能干。这也解释了它为什么生态这么大——**覆盖面广，才有人愿意写插件、写教程**。

反过来也要清楚它的短板：**它本质上是"按脚本发请求"的工具，对浏览器真实渲染、对移动端设备资源这类事，它无能为力**——那正是上一页客户端工具的地盘。

板书：竖排协议：HTTP／HTTPS｜SOAP｜REST｜FTP｜JDBC（造数据・校验・单独测库）｜LDAP｜JMS｜SMTP｜JSR223（自定义扩展）｜TCP｜Java Objects。底部一句："JMeter 不只测网页；不会的协议用 JSR223 自己拼"。

提问：如果被测接口用的是公司自研的二进制协议，JMeter 还能压吗？

预设回答：

- 答"不能，只支持标准协议"——错在"只"字。TCP 取样器可以发原始字节流。

- 答"能，用 TCP 取样器"——对，这是最直接的路子。

- 答"能，用 JSR223 自己写"——也对，而且更灵活，可以用 Groovy 拼报文、做校验。

- 追问收口：所以判断一个工具能不能用，别看它"列了哪些协议"，要看它**有没有留扩展口**。

易错点 / 考点：① 协议清单里 **JDBC** 的用途（造数据、校验、单独测库）是高频考点；② **JSR223** 是"扩展点"，答简答题写上能体现深度；③ 常考判断："JMeter 只能测 Web 应用"——错；④ 别把 SOAP 和 REST 当成两个独立取样器家族，它们都属于 Web Service 范畴。

过渡：GUI 点着跑不适合正式压测，JMeter 真正的战斗力在命令行。

【7.28】支持 cli（命令行）模式（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 JMeter 的命令行参数表，看着枯燥，但它是**正式压测的唯一正确姿势**。

三条理由：**GUI 本身吃资源**——界面要渲染曲线、存结果，几千并发时 JMeter 自己先卡死，你测到的是"JMeter 的极限"；**命令行能自动化**——写进脚本、进流水线，每天定时跑；**命令行能分布式**——配合远程负载机把压力摊开来。

参数挑最常用的：**`-n`** 非 GUI 方式运行，命令行模式的总开关，必须有；**`-t`** 指定测试脚本（.jmx）的地址与名称，**相对路径的根是命令窗口的当前路径**，这点最容易踩坑；**`-l`** 把结果记录到文件（.jtl），**这是原始数据，必须保存**；**`-d`** 指定 JMeter home 目录；**`-L`** 定义日志级别。

属性三个要分清作用域：**`-j`** 定义 JMeter 属性，等同于在 jmeter.properties 里设置；**`-D`** 定义系统属性，等同于在 system.properties 里设置；**`-S`** 加载系统属性文件。一句话——**JMeter 属性给工具用，系统属性给 JVM 用**。另外 **`-r`**、**`-s`**、**`-X`** 是分布式三件套，下一页展开。

代理：**`-H`** 代理 IP、**`-p`** 代理端口、**`-u`** 账号、**`-a`** 口令。提醒一句，课件里 `-p` 还兼有"读取 JMeter 属性文件"的用法，**同一字母不同语境含义不同**，敲命令时对着文档确认。

例子值得照抄：`jmeter -n -t my_test.jmx -l log.jtl -H my.proxy.server -P 8000`——**非 GUI 跑脚本、结果写 log.jtl、走代理 8000 端口**。

> **板书：**左侧"为什么用 CLI"：GUI 吃资源｜可自动化｜可分布式。右侧参数分组：运行（-n 非 GUI、-t 脚本、-l 结果、-d home、-L 日志级别、-j/-D/-S 属性）｜分布式（-r 启动从机、-s 服务器模式、-X 停止）｜代理（-H 代理 IP、-p 端口、-u 账号、-a 口令）。底部照抄示例命令。

提问：脚本和数据文件都准备好了，命令行一跑报"文件不存在"，最可能的原因是什么？

预设回答：

- 答"脚本路径写错了"——可能。
- 答"当前目录不对"——最常见。`-t` 的相对路径以**命令窗口当前路径**为根，你 `cd` 到别的目录就找不到了。
- 答"权限不够"——也有可能，尤其 `-l` 输出目录不可写。
- 追问收口：所以正式跑之前，先 `pwd` 确认目录，或者干脆全用绝对路径——这是最低成本的保险。

易错点 / 考点：① `-n` 是命令行模式总开关，必答；② `-t`、`-l` 的作用与相对路径规则；③ 高频混淆：**JMeter 属性（-j）vs 系统属性（-D）**，一个是工具级、一个是 JVM 级；④ 分布式三件套 `-r`／`-s`／`-X` 的对应关系要清楚；⑤ 记住"GUI 只用来调试脚本，正式压测必须 CLI"。

过渡：手写脚本太慢，JMeter 提供了录制功能，我们看怎么用。

【7.29】JMeter 录制功能（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲录制，它解决的是"**脚本从哪来**"的问题。一个真实业务场景可能有几十个请求，手写太慢，录一遍最快。

课件的做法只有三步，但每一步都有坑。**第一步，加逻辑控制单元：录制控制器**。它是一个占位容器，录制下来的请求都塞进它里面。**第二步，加非测试元件：HTTP 代理服务器**——注意它在工作台里，因为它是帮你产生脚本的工具，不是压测内容。**第三步，在浏览器或网络中设置 HTTPS 代理**，把浏览器的流量指向 JMeter 的代理端口，这样你点页面，JMeter 就把请求记下来。

三个必须知道的坑。**坑一，HTTPS 要装证书**。JMeter 代理要解密 HTTPS 流量，就得让浏览器信任它的自签证书，否则你只能录到 CONNECT 请求，内容全是空的。**坑二，过滤静态资源**。"请求"里通常要加 URL 过滤，把图片、CSS、JS 排除掉，否则一次录制就录进几百个无关请求，脚本没法用。**坑三，录完必须整理**。录制下来的是"你点的顺序"，不是"压测需要的顺序"：要删掉无关请求、要把登录抽成 setup、要把变化的参数（时间戳、token、订单号）用关联提取出来，否则第二次跑就失败。

我特别强调第三点，因为它区分了"会录"和"会做性能测试"。**录制只是把手工操作变成脚本的捷径，场景设计还得靠人。**一个没整理过的录制脚本，几乎必然在回放时失败——因为里面写死了上次的 token。

另外提醒一句：**录制出来的脚本要精简**，能合并的合并、能参数化的参数化。脚本越短，跑起来越稳，出错也越好查。

> **板书：**三步：① 逻辑控制单元 → 录制控制器 ② 非测试元件 → HTTP 代理服务器 ③ 浏览器/系统设置 HTTPS 代理。三个坑：HTTPS 装证书｜过滤静态资源（图片・CSS・JS）｜录完整理（删无关・抽登录・参数化/关联）。底部："录制只是捷径，场景还得人设计"。

提问：录制好的脚本第一次回放成功，第二次失败，最可能是什么原因？

预设回答：

- 答"服务器不稳定"——先怀疑脚本，别先怀疑服务器。

- 答"里面有写死的数据，第二次失效了"——最典型。token、验证码、订单号、时间戳，第一次用掉了，第二次就废。

- 答"数据被第一次跑改掉了"——也很常见，比如库存被扣完、账号被锁。

- 追问收口：所以录制脚本后必须做两件事——**参数化**（把变化的换成变量）＋**关联**（把上一个响应的值提取给下一个请求）。

> **易错点 / 考点：**① 录制三个步骤的名称与所在位置（**代理服务器在工作台**）；② HTTPS 证书这个坑必答；③ **参数化与关联**是脚本能反复跑的关键，案例题高频；④ 常考判断："录制完的脚本可以直接用于压测"——错，必须整理和参数化。

> **过渡：**一台机器压不动怎么办？JMeter 支持分布式，我们看它的结构。

【7.30】JMeter 支持分布式环境（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲分布式压测，解决的是"**一台机器压不动**"的问题。
>
> 课件画的结构是这样：一台 **Master（主控，跑 GUI 的那台）**，下面是若干台 **Slave（从机，也就是负载机）**，所有从机同时向**被测系统**发请求。主控只负责分发脚本和汇总结果，真正施压的是从机。
>
> 这个结构的价值有三层。**第一层，突破单机瓶颈**：一台机器可能只能发 2000 个并发（CPU 或网络先满），四台就能到 8000。**第二层，模拟真实分布**：用户是分散在不同地域、不同网络的，从机放在不同网段，压出来的网络行为才接近真实。**第三层，隔离施压与观测**：主控不参与施压，汇总结果时才不会被自己的负载拖慢。
>
> 配置上有几个关键点。**从机要启动成 server 模式**——就是上一页那个 `-s` 参数，或者直接运行 `jmeter-server` 脚本。**主控要知道从机在哪**——从机列表配置在 **jmeter.properties** 里（课件也点了这个文件），主控用 `-r` 参数启动远程负载机。**端口和防火墙要放通**：主从之间要通信，从机还有回传结果的通道，端口不通就表现为"连不上从机"。
>
> 还有一个**特别容易踩的坑**：**主从的 JMeter 版本要一致，脚本里用到的插件和依赖也要一致**。否则从机会报找不到元件，或者结果数据格式对不上。很多人第一次搭分布式就栽在这儿。
>
> 再讲一个反直觉的点：**分布式不是免费的**。从机越多，主控汇总的数据量越大，网络传输和时间同步的开销也越大；而且不同从机的时钟如果不准，结果时间轴就对不齐。所以从机数量要**按需增加**，不是越多越好。

> **板书：**画结构图：Master（GUI，分发脚本＋汇总）→ 多个 Slave（负载机，实际施压）→ 被测系统。右侧

> 要点：从机 `-s`／jmeter-server｜从机列表在 jmeter.properties｜`-r` 启动远程｜端口放通｜**版本与插件必须一致**。底部："分布式不是免费的：汇总开销＋时钟对齐"。

> **提问：**四台从机加起来应该有 8000 并发，但报告里只统计到 3000，可能是什么问题？

> **预设回答：**

- 答"从机没连上"——先看主控日志，有没有 slave 启动失败的记录。

- 答"有的从机先到瓶颈了"——很常见，从机的 CPU 或带宽先满，实际发出的并发远小于设定值。

- 答"结果没汇总全"——也有可能，尤其网络中断或结果文件写失败。

- 追问收口：查分布式问题有个顺序——**先看从机有没有真正启动（日志）→ 再看每台从机自己的负载（CPU／网络）→ 最后看汇总**。

> **易错点 / 考点：**① Master／Slave 的职责划分要清楚（**主控不分担压力**）；② 配置位置：从机列表在 **jmeter.properties**；③ 高频坑：**主从版本、插件、依赖必须一致**；④ 常考判断："从机越多压测越准"——不对，汇总开销和时钟偏差会引入误差。

> **过渡：**JMeter 本身功能有限，靠插件补齐。我们看插件怎么装。

【7.31】JMeter Plugins Manager（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲插件管理器，内容很短，但它是"把 JMeter 从能用变成好用"的关键一步。
>
> 课件给的做法是：把 **jmeter-plugins-manager 的 jar 包放到 jmeter 安装目录下的 lib/ext 目录**里，重启 JMeter，界面上就会出现插件管理器入口，然后你就可以在里面搜索、安装插件了。
>
> 为什么放 `lib/ext`？因为这是 JMeter 加载**扩展元件**的标准目录——JMeter 启动时会扫描这个目录，发现实现了 JMeter 扩展接口的 jar，就把它注册成可用元件。理解这一点，你以后装任何 JMeter 扩展都是同一个动作：**丢进 lib/ext，重启**。
>
> 插件能补什么？我给你归三类。**第一类，图表类**：JMeter 自带的结果展示比较朴素，插件能出更专业的响应时间分布、TPS 曲线、并发对比图，写报告时很有用。**第二类，场景类**：比如阶梯加压（Stepping Thread Group）、Ultimate Thread Group，能画"渐进加压／波浪加压"这类复杂负载模型——回想 7.10 讲的负载

模式，手工配很麻烦，插件直接给你图形化配置。**第三类，协议与元件类**：某些特殊协议的支持、额外的断言、额外的取样器。

但我要提醒一句课件的风格：**插件不是越多越好**。装得越多，版本冲突和兼容问题的概率越大，从机与主控的插件还必须一致（上一页刚讲过）。所以原则是：**需要什么装什么，装完记录版本**。

还有一条实践建议：**插件版本和 JMeter 大版本要匹配**。JMeter 升级后，老插件可能失效，表现为"元件不见了"或者启动报错——这时候先怀疑插件，别怀疑脚本。

板书：一行流程：下载 jmeter-plugins-manager 的 jar → 放进 **JMeter/lib/ext** → 重启 → 插件管理器里搜索安装。右侧三类插件：图表类（响应时间・TPS 曲线）｜场景类（阶梯加压・Ultimate Thread Group，对应负载模式）｜协议/元件类。底部："需要什么装什么，装完记版本；主从插件必须一致"。

提问：装完插件重启 JMeter，发现新元件没出现，你会按什么顺序排查？

预设回答：

- 答"重装一遍"——先别急，先看日志。

- 答"看 jmeter.log 有没有类加载错误"——对，这是最快的证据：版本不兼容通常在这里报。

- 答"看 jar 是不是放错目录"——也很关键，放 `lib/` 和放 `lib/ext/` 效果不一样。

- 追问收口：排查顺序是**目录 → 日志 → 版本匹配**，这三步能解决九成的插件问题。

易错点 / 考点：① 插件 jar 的放置目录 **lib/ext** 是必考点；② 装完必须重启才生效；③ 插件与 JMeter 版本的兼容问题是高频故障原因；④ 记住"主从插件一致"这条，它同时是分布式和插件两个知识点的交叉考点。

过渡：装好插件，我们看一个实际的插件效果示意。

【7.32】插件示例（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是插件的实际效果示例，我借它讲"**同样的数据，不同的画法，结论不一样**"。

为什么图形化插件值得单独占一页？因为性能测试的结论，本质上是**从图里读出来的**。同样的原始数据，用不同的图去看，你会看到不同的东西：

看**响应时间随时间变化的曲线**，你能看出"跑着跑着变慢了"——这是泄漏或资源耗尽的信号；看 **TPS 曲线**，你能看出系统在哪个并发点饱和；看**响应时间分布（比如分位数图）**，你能看出"平均值 200 毫秒，但 P99 是 8 秒"这种被平均值掩盖的灾难；看**阶梯加压的阶梯图**，你能看出每一档加压后的稳态表现。

这里要重点讲一个观念：**永远不要只看平均值**。平均值会把少数极慢的请求藏起来。举个数：100 个请求，99 个 100 毫秒，1 个 10 秒——平均值是 199 毫秒，看起来很好；但那个 10 秒的请求，就是一个真实用户在白屏前等了 10 秒。所以报告里要看 **P90、P95、P99**，以及**最大响应时间**。

这也是为什么性能指标里通常写"**95% 的请求在 X 秒内完成**"，而不是"平均响应时间 X 秒"。前者是用户感受的口径，后者是统计学的自我安慰。

回到这一页的定位：插件示例不是让你记界面长什么样，是让你记住**"该看哪几种图"**。我的建议是四种图必备：响应时间趋势、TPS 趋势、响应时间分位数分布、错误率趋势。四张图放一起，一份压测报告就有了骨架。

板书：四张必备图：① 响应时间趋势（看是否越跑越慢）② TPS 趋势（看饱和点）③ 响应时间分位数（P90／P95／P99＋最大值）④ 错误率趋势。中间写一行大字："平均值会骗人"。底部："指标写法：95% 的请求在 X 秒内完成"。

提问：报告写"平均响应时间 199 毫秒"，这个数字可能掩盖什么问题?

预设回答：

- 答"可能有个别请求特别慢"——正是。平均值对极端值不敏感。

- 答"可能慢的请求集中在某个接口"——也对，平均值会把不同接口混在一起。

- 答"可能慢在某个时段"——补得好，平均值也把时间维度抹平了。

- 追问收口：所以要三个维度同时看——**分位数（尾部）、按接口拆分、按时间看趋势**。

易错点 / 考点：① **平均值陷阱**是性能测试最经典考点，答题必须提分位数；② 四种必备图要能说出各自的用途；③ 常考判断："平均响应时间达标就算通过"——错，要看分位数和错误率；④ 记住行业常用写法"95% 请求在 X 秒内"，它比平均值更能反映体验。

过渡：工具和图表都齐了，课件还给了进一步学习的入口。

【7.33】JMeter 更多参考（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 JMeter 的官方参考入口，课件给了两条路：一条是 Apache 的 JMeter wiki，一条是官方的用户手册（usermanual）。我用一分钟讲"怎么用这两条路自学"，因为工具这东西，课堂只能带你入门。

两者的分工不一样。**用户手册（usermanual）是"元件字典"**：每一个元件、每一个属性、每一个参数的含义，都在里面。你写脚本时卡住了——比如"这个定时器到底有几種""CSV 数据文件怎么配"——查手册最快。**wiki 更像"经验库"**：常见问题、最佳实践、别人的踩坑记录，还有各种"怎么做某件事"的教程。

我给你一个自学的顺序，照着走效率最高：**第一，先把官方手册里的"Building a Test Plan"那一章通读一遍**，它会带你从零做出一个能跑的脚本；**第二，把每个元件的属性页扫一遍**，不求记住，只求"知道有这个东西、需要时知道去哪查"；**第三，遇到具体问题再去 wiki 和社区搜**。

还有一条更重要的建议：**工具的自学能力，比工具本身更值钱**。JMeter 会升级、会被别的工具替代，但"读官方文档、看示例、动手试"这套方法不会过时。你以后遇到 k6、Gatling、Locust，甚至公司自研的压测平台，都是同一套路子。

顺便说一句课件没展开、但你要知道的：**官方文档是全英文的**。别怕，性能测试的术语就那么几十个——thread、sampler、assertion、listener、throughput、latency，把这一节讲过的词对应上，读文档就不费劲了。

> **板书：**两条入口：usermanual（元件字典：每个元件・属性・参数）｜wiki（经验库：FAQ・最佳实践・教程）。下面写自学三步：① 通读 Building a Test Plan ② 扫一遍元件属性 ③ 有问题再搜 wiki／社区。底部："工具的自学能力 > 工具本身"。

提问：同一个问题——"为什么我的脚本跑出来没有结果"——你会在手册里查，还是在 wiki 里查？

预设回答：

- 答"wiki，找别人的经验"——可以，但先查手册更快：可能是监听器没加、结果文件路径不对这类明确的配置问题。

- 答"手册，先看元件说明"——对，先排除"配置错误"，再考虑"用法问题"。

- 答"直接搜引擎"——也行，但搜到的答案常常对应旧版本，容易走偏。

- 追问收口：查文档的顺序是**手册（配置）→ wiki（用法）→ 社区（疑难）**，别一上来就搜。

> **易错点 / 考点：**① 这一页考得少，但"官方文档在哪"偶尔出现在填空；② 更重要的是记住**手册与 wiki 的分工**，这是自学习惯题的标准答法；③ 别把"更多参考"当成可跳过的废话页——它给的是**课后自学路径**，实验课会用到。

> **过渡：**工具会用了，数据也有了，接下来是整章最难的一步——性能分析。

【7.34】性能分析（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲性能分析，也就是"**从数据里找出原因**"。这是性能测试里最考验功底的一环，也是课件在 7.4 就点过的"结果分析有挑战"。

我给你一套四步法，遇到任何"慢"都能用。

第一步，确认现象，别看错指标。先把"慢"量化：是响应时间长，还是吞吐上不去，还是错误率高？三者常常一起出现，但**根因可能完全不同**。同时要看分位数——平均值会骗人。

第二步，拆时间。把一次请求的耗时切成段：DNS、网络、服务时间、数据库时间、返回、渲染。哪一段最长，问题就在哪一层。这一步用的就是 7.22 讲过的那套时间分解。

第三步，找证据。分位数告诉你"哪儿疼"，证据告诉你"为什么疼"。常用的证据有五种：**资源监控曲线**（CPU、内存、IO、网络）、**应用日志**（异常、超时、重试）、**线程栈**（线程都卡在哪个方法上）、**GC 日志**（停顿有多长、频率多高）、**慢查询日志／执行计划**（SQL 走了索引没有）。**没有证据的结论叫猜测，不叫分析。**

第四步，验证假设。这一步最容易被跳过：你怀疑是慢查询，那就单独跑一次这个 SQL 看它到底多久；你怀疑是锁竞争，那就把锁去掉再压一遍看有没有改善。**改一个变量、跑一次、对比**——这是性能分析唯一可靠的验证方式。

再说一个经验：**先排除法，再深挖法。**先用分层（客户端、应用、访问层、数据库、连接、网络）快速排除掉大部分层，再对可疑的那一层深挖。上来就深挖某一层，很容易挖错地方。

最后一句提醒：**性能分析是"证据链"，不是"灵感"。**你能把"现象→数据→证据→结论→验证"这条链说圆，

结论才站得住。

> **板书：**四步法：① 确认现象（响应时间／吞吐／错误率，看分位数）② 拆时间（DNS→网络→服务→DB→返回→渲染）③ 找证据（监控曲线・日志・线程栈・GC 日志・慢查询/执行计划）④ 验证假设（改一个变量，跑一次，对比）。底部："没有证据的结论叫猜测"。

提问：压测发现"响应时间从 100 毫秒涨到 2 秒"，你会先看哪三样东西？

预设回答：

- 答"看 CPU 和内存"——基础，但不够，要先确认是"计算型"还是"等待型"。

- 答"看数据库慢查询"——高频嫌疑，值得先查。

- 答"看线程栈，看线程卡在哪"——很专业，尤其适合"CPU 不高但很慢"的情况。

- 追问收口：顺序建议是**先看资源曲线定类型（计算型/等待型）→ 再看最可疑那一层的证据（DB 或线程栈）→ 最后改一个变量验证**。

易错点 / 考点：① 四步法的名称与顺序可作简答模板；② 高频错答："响应时间变长就是代码有问题"——可能是数据库、可能是网络、可能是 GC；③ **"改一个变量再验证"** 这条方法论常被考到（对照实验思想）；④ 记住五种证据来源，答题写全比写深更容易得分。

【思政落点 4】：性能分析最像科学研究的日常：提出假设、找证据、做对照实验、被数据推翻、再来一轮。它教给人的第一件事是**服从数据**——你原来以为是数据库慢，结果线程栈显示全卡在一个日志锁上，那就得改口。承认"我猜错了"在工程里不是丢脸，是进步的唯一入口。请同学们养成一个习惯：**说结论之前，先问自己"证据是什么"**。这种"让事实说话"的态度，是可以迁移到任何职业里的硬底子。

过渡：找到瓶颈只是第一步，接下来要动手改——这就是调优。

【7.35】调优-Tuning（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲调优，也就是"**改**"。性能测试和调优是一对：测试负责度量，调优负责改进，然后循环。

调优要讲顺序，我给你排一下，从便宜到贵。**第一层，配置参数**：数据库的连接池大小、JVM 堆大小、GC 策略、线程池参数、缓存容量、超时时间。这一层最便宜——改个数字、重启、再测，几分钟一轮。**第二层，**

SQL 与索引：加索引、改 SQL 写法、去掉不必要的联表。这一层收益常常最大，成本也不高，所以是性能调优的"性价比之王"。**第三层，代码**：去掉循环查库、减少对象创建、加缓存、改异步、批处理。这一层要改代码、要回归测试，成本上去了。**第四层，架构**：加缓存层、读写分离、分库分表、服务拆分、水平扩容。这一层最贵、最慢，但天花板也最高。

为什么要按这个顺序？因为**便宜的方案能解决的，就别上贵的**。现实中常见的失误是：一遇到慢就直接加机器，结果加了八台还是不达标——因为瓶颈在一条没走索引的 SQL 上。**加机器解决的是"能力不足"，解决不了"效率低下"。**

调优还有两条铁律。**第一，一次只改一个变量**。同时改三处，性能好了你不知道是哪处的功劳，坏了也不知道该退哪一处。**第二，改完必须重测，而且要跑同一套场景**。场景变了，数据就不可比。

最后提醒：**调优要有基线、要有目标**。没有基线，你不知道改进了多少；没有目标，你不知道什么时候停。无限调优是资源的黑洞——工程上要的是"**达标即止**"，不是"极致性能"。

板书：四层阶梯（便宜→贵）：① 配置参数（连接池・JVM・GC・线程池・缓存・超时）② SQL 与索引（性价比之王）③ 代码（去循环查库・缓存・异步・批处理）④ 架构（缓存层・读写分离・分库分表・扩容）。底部两条铁律："一次只改一个变量"＋"改完跑同一套场景重测"。

提问：系统慢了，团队第一反应是"加两台服务器"，你会怎么回应？

预设回答：

- 答"先加吧，快"——能缓解，但可能白花钱：瓶颈在 SQL 上，加机器只是把同一条慢 SQL 复制到更多机器上跑。

- 答"先定位再决定"——对。定位成本一天，加机器成本几万，顺序明显。

- 答"要看是计算型还是等待型"——很专业：等待型（等 DB、等网络）加机器往往无效。

- 追问收口：所以回应应该说——**"先花半天确认瓶颈在哪一层，如果是能力不足再扩，是效率问题就改代码或 SQL"**。

易错点 / 考点：① 四层调优的顺序与代表手段；② 高频判断："性能问题都可以通过加硬件解决"——错，架构/算法级问题加硬件无效甚至更糟；③ **一次只改一个变量**是实验方法的核心，必考；④ "达标即止"这个工程观值得记，案例题里常用来说明"不要过度优化"。

过渡：调优完了不代表结束——系统上线后还得一直盯着，这就是性能监控。

【7.36】性能监控（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲性能监控。如果说性能测试是"体检"，监控就是"**随身的心率监测**"——它不做事前判断，它负责**一直看着、及时报警**。

监控要盯三类东西。**第一类是资源**：CPU、内存、磁盘 IO、网络带宽，还有连接数、文件句柄、线程数。这一类的特点是"看得见的上限"，快到顶就要报警。**第二类是业务**：TPS、响应时间、错误率、超时率、队列积压长度。这一类直接对应"用户感受到没有"。**第三类是依赖**：数据库慢查询数、缓存命中率、消息队列堆积、第三方接口的可用性。很多"我们的系统慢"，最后查出来是"别人的接口慢"。

业界有个很好用的说法，叫**四个黄金信号**：**延迟、流量、错误、饱和度**。延迟是响应时间，流量是请求量，错误是失败比例，饱和度是资源用到多满。四个信号覆盖了绝大多数故障的前兆，你可以用它当监控面板的骨架。

监控的价值在哪？**在于把"用户投诉"变成"我们比用户先知道"**。这中间的差距，就是运维水平。用户投诉意味着你已经损失了信任；监控报警意味着你还有时间处理。

还有一条实践要点：**监控要有基线，报警要有阈值**。没有基线的曲线，你看不出"今天的 60% 是不是异常"；没有阈值的报警，要么天天响（狼来了），要么永远不响（等于没有）。阈值怎么定？看历史分位数，比如 P95 的正常水平上浮 20% 就报警。

最后串一句：**性能测试、调优、监控，是一个闭环的三段**——测试定基线，调优改现状，监控守长期。

> **板书：**三类监控对象：资源（CPU・内存・IO・带宽・连接数・句柄・线程）｜业务（TPS・响应时间・错误率・超时率・队列长度）｜依赖（慢查询・缓存命中率・MQ 堆积・第三方可用性）。右侧四个黄金信号：延迟・流量・错误・饱和度。底部："监控让你比用户先知道"。

提问：线上没有做任何监控，能靠"用户反馈"来发现性能问题吗？

预设回答：

- 答"能，用户会告诉我们"——能，但代价是**用户已经在受损失**，而且反馈往往是零散的、滞后的。

- 答"不能，太被动"——对。而且用户描述的"慢"没有量化，排查还得从头来。

- 答"至少要有基础监控"——务实。哪怕只监控 CPU、响应时间、错误率三个指标，也比什么都没有强。

- 追问收口：所以监控的最低配置是"**能回答三个问题**"——现在有多快、比平时慢多少、错在哪。

> **易错点 / 考点：**① 三类监控对象要能说全，"依赖"这一类最容易被漏；② **四个黄金信号**（延迟、流量、错误、饱和度）是可加分的外部知识；③ 高频判断："监控就是为了出事时看日志"——错，监控的核心价值是**提前发现**；④ 报警阈值要基于历史分位数，拍脑袋定阈值是常见错误。

> **过渡：**把测试、调优、监控串起来，就得到一张全生命周期的性能工程图。

【7.37】全生命周期的性能测试（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页是 7.1 收尾的"完整版地图"，它把前面零散讲的活动按阶段排好，并且给每个阶段配上了**产出物和度量**。
>
> 按阶段往下走：**性能需求收集与定义**——产出性能需求文档，度量是"指标是否可测"；**内建性能（设计中）、性能建模**——产出架构方案与性能模型，度量是"设计是否满足容量目标"；**单元性能测试、代码优化**——产出优化后的代码，度量是"关键路径的复杂度与耗时"；**性能测试**——产出测试报告与数据，度量是"是否达标"；**监控与容量管理**——产出监控面板与扩容计划，度量是"容量水位与告警准确率"。
>
> 这一页还有几个词要专门解释，因为它们把"测试"提升成了"工程"。
>
> **代码质量与性能、性能测试与工程策略**：意思是性能不是测试部门一家的事，它要写进研发流程——代码规范里要有性能条款，评审清单里要有性能项，发布门禁里要有性能基线。**没有流程支撑的性能，只能靠个人英雄主义。**
>
> **业务度量、DoD、性能度量／KPI Dashboard**：DoD 是"完成的定义"（Definition of Done）——一个需求什么时候算做完？如果 DoD 里写了"性能达标"，那性能就成了交付的一部分，而不是可选项。**KPI Dashboard** 是把性能指标做成看板，让所有人随时能看到。
>
> **环境验证、测试执行（含基线、生产负载、可伸缩性等）**：基线是"标准状态下的一把尺"，生产负载是"真实流量的模型"，可伸缩性是"加资源能不能线性换来能力"。
>
> **容量管理**：这是上线后的长期工作——预测增长、提前扩容、算成本。它的输出不是报告，是**决策**。
>
> 一句话总结这一页：**性能工程的成熟标志，是它进入了流程和看板，而不是停留在某个人的压测脚本里。**

> **板书：**五个阶段＋产出：需求定义（需求文档）→ 性能建模/内建（架构方案・模型）→ 单元性能/代码优化（优化后代码）→ 性能测试（报告・数据）→ 监控・容量管理（看板・扩容计划）。右侧术语：DoD（完成的定义含性能）｜KPI Dashboard｜基线｜生产负载｜可伸缩性。底部："性能进入流程和看板，才算工程化"。

提问：如果团队只有一个人在压测，其他人都不关心性能，这套体系能跑起来吗？

预设回答：

- 答"能，有人做就行"——短期能出报告，长期一定断：需求没人定指标、设计没人管容量、上线没人看监控。
- 答"不能，要进流程"——对。性能必须写进 DoD、评审清单和发布门禁。
- 答"先从一个项目试点"——很务实的路径：先证明价值，再推广成规范。
- 追问收口：所以性能工程的关键动作是"**把个人能力变成流程约束**"，这和我们在第 4 章讲流程的思路是一样的。

易错点 / 考点：① 五个阶段与产出物的对应关系；② **DoD** 这个词要会解释，并能说明"把性能写进 DoD"的意义；③ 高频判断："容量管理是运维的事，与测试无关"——错，容量来自测试数据；④ 记住"基线"在多个页反复出现，它是所有比较的前提。

过渡：7.1 的内容到这里收口，我们用一页小结把它压实。

【7.38】本节小结（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 7.1 的小结，四条，我按"目的—概念—活动—工具"的顺序过一遍，你对着自己的笔记查漏。

第一条，了解性能测试的不同目的（性能测试类型）。这句话的意思是把类型和目标对上：基准是"记住现在"，验证是"对照标准"，规划是"决定买什么"，容量是"摸到上限"，压力是"逼到崩"，渗入是"泡出漏"，峰谷是"突变找根因"。**七个类型，七种目的，别混着背。**

第二条，清楚一些基本概念：负载、负载模式、虚拟用户、性能指标。这四个词是这门课的"性能四件套"。负载的三要素是 RPS、并发连接数、思考时间；负载模式看的是"跑多久、怎么加载"；虚拟用户是工具模拟出来的"假用户"，**它的行为由脚本和思考时间决定**；性能指标则是响应时间、吞吐量、资源利用率、错误率这一组。

第三条，性能测试会涉及架构评审、代码评审、性能调优。这一条最容易被学生忽略：很多人以为性能测试就是"跑工具"，课件明确告诉你，它跨了静态测试（评审）和动态测试（施压），还延伸到调优。**所以做性能的人，不能只会用工具，得懂架构、懂代码、懂数据库。**

第四条，熟练使用性能测试工具（如 JMeter）。注意"熟练"两个字——会打开界面不算熟练，能独立完成"设计场景→写脚本→分布式施压→采集→出报告"才算。

最后给你一个自测方法：**合上讲稿，你能不能在三分钟内把"性能测试是什么、分几类、怎么做、怎么看结果"讲一遍？** 讲得下来，这一节就过关了；卡在哪一处，回去补那一处。

> **板书：**四条：① 目的 → 七种类型（基准・验证・规划・容量・压力・渗入・峰谷）② 概念四件套（负载・负载模式・虚拟用户・性能指标）③ 跨活动（架构评审・代码评审・调优）④ 工具熟练（场景→脚本→分布式→采集→报告）。底部自测："三分钟讲一遍"。

提问："虚拟用户"和"真实用户"最大的区别是什么？

预设回答：

- 答"虚拟用户是假的"——太表面。工具里的虚拟用户就是一段脚本加一个计时器。

- 答"虚拟用户不会思考"——接近了。它的"思考"是人为设定的固定或随机间隔。

- 答"虚拟用户的行为是脚本规定的，真实用户是不可预测的"——最准确。这也是为什么要专门设计负载模式和思考时间，去逼近真实。

- 追问收口：所以性能测试的准确性，取决于**你的负载模型有多像真实用户**，而不是工具有多强。

易错点 / 考点：① 七种类型的名称是本节最可能出简答/填空的地方，必须默写；② **虚拟用户的行为由脚本与思考时间决定**，这句话能解释很多"压测结果和线上不符"的现象；③ 高频判断："性能测试只需要会工具"——错，课件明确列出架构评审、代码评审、调优；④ 小结页本身常被当复习提纲，建议整段背。

过渡：理论讲完，接下来两次实验把它落地。先看后端性能测试。

【7.39】实验 4-1：基于 JMeter 的后端性能测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是实验 4-1 的要求，我逐条解释"为什么要求这一条"，你写实验报告时就知道该写什么。

前置条件：**选一个 Web 应用，需要用户登录的**。为什么要登录？因为登录带来三个真实问题：**会话要保持**（cookie／token）、**请求有依赖**（先登录才能下单）、**数据要参数化**（每个虚拟用户用自己的账号）。这三点是真实压测的核心难点，去掉登录，实验就变成"压一个静态页"，学不到东西。

要求里逐条看。**用代理服务器加录制控制器录制脚本**——练的是"脚本从哪来"。**选用插件，设置线程组相关参数**——练的是负载模型：线程数、Ramp-Up、持续时间。**会用 2-3 种控制器**——练的是流程编排：事务控制器、循环控制器、如果控制器。**会配置 2-3 种采样器**——HTTP 是必备，再加 JDBC 或 FTP 之类。**会使用 HTTP 授权、cookie 管理器**——练会话保持。**会设置断言**——练"判断对错"，这一条决定你的报告是不是可信的。**会使用 JDBC 连接预处理**——练造数据与校验。**会用 2-3 种监听器（查看树、响应时间、吞吐量）**——练结果呈现。

交付物有两样：**性能测试报告（Word 格式，含测试系统、环境、场景设计、执行和结果分析）** 和 **性能测试脚本（.jmx 文件）**。

报告的五个部分，其实就是我们这一节讲的主线：**测试系统**（测什么）、**环境**（六项要素）、**场景设计**（负载三要素＋模式＋类型）、**执行**（怎么跑、跑多久）、**结果分析**（四步法＋证据）。你按这条线写，报告自然就有骨架，不会写成"截图合集"。

一条评分建议：**报告里每个数字都要有出处**——是哪个监听器、哪张图、哪次执行。这既是技术要求，也是职业习惯。

板书：前置：需登录的 Web 应用（会话・依赖・参数化）。要求清单：录制脚本｜线程组参数｜2-3 种控制器｜2-3 种采样器｜HTTP 授权＋cookie 管理器｜断言｜JDBC 预处理｜2-3 种监听器。交付：报告（系统・环境・场景・执行・分析）＋ .jmx。底部："每个数字都要有出处"。

提问：为什么实验强制要求"设置断言"，而不是"能把请求发出去就行"？

预设回答：

- 答"为了报告好看"——恰恰相反，断言会让报告变难看：它会把业务失败暴露出来。

- 答"为了判断结果对不对"——对。没有断言，工具只统计"有没有响应"，不统计"响应对不对"。

- 答"为了能算错误率"——也是关键：错误率这个指标依赖断言。

- 追问收口：所以断言不是"加分项"，是"数据可信的前提"；一份没有断言的压测报告，结论不能采信。

易错点 / 考点：① 实验要求的条目要能逐条对应到知识点（录制→脚本来源、线程组→负载模型、断言→可

信度）；② **报告五部分**是评分重点，缺一项扣分；③ 常见失分：只贴截图不写分析、没有场景设计说明、数据没有版本与环境信息；④ 提交 .jmx 时要保证**别人能直接跑通**（路径用相对路径、外部数据文件一起交）。

过渡：后端测完，客户端也要测。下一次实验用 PerfDog。

【7.40】实验 4-2：基于 PerfDog 的客户端性能测试（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是实验 4-2，客户端性能测试，用 PerfDog 做。课件把步骤列得很清楚，我按步骤讲要点。

第一步，去 PerfDog 官网下载正确的桌面应用程序。"正确"这两个字有讲究：Windows 版和 Mac 版不一样，有的功能还依赖 iTunes 之类的驱动组件，装错了连不上手机。

第二步，USB 连接手机，让它自动检测并添加手机到应用列表。这一步常见的坑是**开发者模式和 USB 调试没开**，或者数据线只支持充电不支持数据传输——工具再好，连不上就是白搭。

第三步，设定测试模式（USB 模式或 WIFI 模式）。USB 模式更稳、延迟更低，适合精细采集；WIFI 模式方便，但网络抖动会影响数据。做实验建议先用 USB。

第四步，选择被测试的 App 开始测试。这里要注意**测什么场景**：冷启动、页面切换、列表滑动、长时间使用，每个场景采集的数据含义不同。

第五步，测试过程中采集性能数据。第六步，保存测试结果。第七步，进行测试数据统计与分析——这一步是重点，工具给你曲线，你要说出"哪一段有异常、为什么"。**第八步，编写测试报告**。

报告里我建议固定放四样东西：**测试机型与系统版本**（客户端性能强依赖硬件）、**测试场景说明**（做了什么操作）、**关键指标曲线**（FPS、内存、CPU、启动耗时）、**结论与建议**（哪里不达标、可能原因）。

最后强调一条实验纪律：**同一个场景至少测两遍**，一遍基线一遍对照，否则你拿到的只是"一次观测"，不是"测试结论"。

板书：八步：下载正确版本 → USB 连接（开发者模式・USB 调试）→ 选测试模式（USB 稳／WIFI 便）→ 选 App → 采集数据 → 保存结果 → 统计与分析 → 写报告。报告四要素：机型与系统｜场景说明｜指标曲线（FPS・内存・CPU・启动耗时）｜结论与建议。底部："同场景测两遍：基线＋对照"。

提问：同一台手机、同一个 App，两次测出来的启动耗时差 0.5 秒，你怎么判断哪个是真的？

预设回答：

- 答"取平均"——可以，但要先排除干扰因素。

- 答"多测几次看波动"——对，至少 3-5 次，看离散程度。

- 答"要看是不是冷启动"——非常关键。冷启动和热启动差好几倍，混着算平均值毫无意义。

- 追问收口：所以客户端测试的第一条纪律是**把测试条件钉死**：冷/热启动、网络、电量、后台进程、系统版本，全部写进报告。

易错点 / 考点：① 八个步骤的顺序；② **冷启动 vs 热启动**的区分是客户端性能的核心概念，常考；③ 高频错答："客户端性能测试用模拟器就行"——真机数据才有意义；④ 报告必须写清机型、系统版本、场景与条件，否则数据无法复现。

过渡：7.1 性能测试到此结束。接下来进入 7.2，看另一类专项——安全性测试。

【7.41】7.2 安全性测试（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

性能讲完，我们进入 7.2，**安全性测试**。先把这个专项的位置说清楚。

性能问"**扛不扛得住**"，安全问"**会不会被人从外面打开**"。共同点是**功能测试都发现不了**：功能测试假设使用者善意、按说明书操作；安全测试假设**对手恶意、会想尽办法绕过规则**。假设一换，姿势就变了。

这一节的路线：先看**威胁模型与安全漏洞模式**——漏洞是什么、从哪来、攻击分几个阶段；再看**范围与方法**——安全目标、测什么、怎么测；然后**贯穿研发生命周期**的测试；最后落到 **Web 安全性测试**与工具。

提醒一个心态问题：很多同学觉得"安全是安全工程师的事"。这个想法站不住——**大多数漏洞是开发阶段写进去的**：少一个参数校验、多一句 SQL 拼接。写代码的人不设防，事后补不完。第一道防线是**每个开发者都有攻击者视角**。

带着一个问题听：**你写过的接口，如果我是攻击者，我会先试哪一下？**

板书：中间写"7.2 安全性测试"；左侧写"性能：扛不扛得住（善意使用者 · 压力）"；右侧写"安全：会不会被打开（恶意对手 · 绕过规则）"。下方四条路线：威胁模型与漏洞模式 → 范围与方法 → 贯穿 SDLC → Web 安全与工具。

> **提问：**功能测试全部通过的系统，安全性上可能是"千疮百孔"的吗？为什么？

> **预设回答：**

- 答"可能，功能测试不测攻击"——对。功能用例里没有"输入一个单引号"这一条。

- 答"不可能，功能都对了说明逻辑没问题"——错在假设：攻击者用的输入，本来就不在功能用例的设计空间里。

- 答"要看有没有做过安全测试"——回到判据问题：功能看"预期结果"，安全看"能不能被利用"。

- 追问收口：所以安全测试的核心动作是**主动去构造"不该出现的输入和环境"**。

> **易错点 / 考点：**① 性能与安全两个专项的**判据差异**要能一句话说清；② 本章安全部分的结构（四级）要能复述；③ 高频判断："安全性测试是安全专家的事"——错，安全要贯穿开发全过程；④ 记住本节的问题意识：**攻击者视角**，这是后面所有内容的钥匙。

> **过渡：**在讲漏洞之前，先分清一类人——黑帽和白帽。

【7.42】知道它们的区分吗？（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲两种人：**黑帽黑客**和**白帽黑客**。为什么要在一门测试课里先讲这个？因为**安全测试的技术和攻击的技术，本质上是同一套**。区别不在手段，在**授权和目的**。

黑帽是未授权入侵：找漏洞是为了牟利、破坏、控制，手段和工具与白帽几乎一样，但**没有授权**，后果由受害者承担。

白帽是授权测试：在**明确授权**的范围内，用攻击者的方法去找漏洞，目的是**在坏人之前发现并修掉**。安全测试、渗透测试、红队演练，都属于这一侧。

这个区分对测试人员不是道德说教，是**职业规范**：做安全测试之前必须有书面授权、有测试范围、有时间窗口、有应急联系人。没有这些，你做的"测试"在法律和职业上都站不住，而且可能把生产系统搞挂。

还有一个中间地带值得知道：**灰帽**——发现了漏洞但没有授权就去验证，还公开出去。很多争议都出在这。**善意不能替代授权**，这是这一行最硬的一条规矩。

所以这一页真正的落点是：**同样一个 SQL 注入手法，在授权范围内是测试，在授权范围外是犯罪。** 你以后做

安全测试，第一件事是把授权文件、范围、时间窗写清楚，第二件事才是敲命令。

板书：左侧写"黑帽：未授权・牟利/破坏"，右侧写"白帽：授权・发现并修复"，中间画一条线标"**授权**"。下方写安全测试的四件套："书面授权｜测试范围｜时间窗口｜应急联系人"。底部："善意不能替代授权"。

提问：一个安全工程师在客户系统里发现了一个高危漏洞，顺手验证了一下能否拿到数据，这算白帽行为吗？

预设回答：

- 答"算，他是好意"——不一定。要看**授权范围里有没有包含"漏洞验证/利用"**。

- 答"要看有没有授权"——对。发现漏洞（扫描）和验证可利用性（利用），在很多授权书里是两件事。

- 答"应该先报告再验证"——很专业。正确顺序是**报告 → 客户确认 → 在约定窗口内验证**。

- 追问收口：所以判断标准不是"善意"，是"**授权边界**"；越界一步，性质就变了。

易错点 / 考点：① 黑帽／白帽／灰帽的区分标准是**授权**，不是技术高低；② 安全测试的四个前置文件（授权、范围、窗口、联系人）是实践题得分点；③ 高频判断："只要目的是好的，验证漏洞不算越界"——错；④ 这一页可能出概念题，答案要落在"授权"两个字上。

过渡：分清人之后，看这一讲（安全性测试）的整体安排。

【7.43】第 13 讲：安全性测试（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是安全性测试的内容目录，五个部分，我按顺序点一遍，你记结构。

第一，威胁模型与安全漏洞模式——先建立"敌人怎么想"的模型：漏洞是什么、从哪来、攻击分几个阶段、有哪些经典的漏洞类型（缓冲区溢出、注入、XSS、CSRF 等）。

第二，安全性测试的范围与方法——测什么、怎么测：安全目标、安全功能测试与漏洞测试的区别、静态与动态方法、模糊测试与渗透测试。

第三，贯穿研发生命周期的安全性测试——把安全活动铺到需求、设计、编码、测试、运维各个阶段，重点是架构风险分析和代码评审。

第四，Web 安全性测试——落到最常见的场景：注入、XSS、CSRF、URL 访问控制这些具体攻击怎么测。

第五，安全性测试工具——扫描器、代理工具、漏洞库这类支撑手段。

五块的关系：**建模 → 定范围 → 铺流程 → 落场景 → 上工具**。一句话主线：**安全测试不是"跑个扫描器"，而是"用威胁模型去设计攻击，再验证防线是否有效"。**

板书：竖排五块：① 威胁模型与漏洞模式 ② 范围与方法（目标・功能/漏洞・静态/动态・模糊・渗透）③ 贯穿 SDLC ④ Web 安全性测试（注入・XSS・CSRF・URL 控制）⑤ 工具。底部一条链："建模 → 定范围 → 铺流程 → 落场景 → 上工具"。

提问：五个部分里，你觉得哪一部分最容易被团队跳过？

预设回答：

- 答"工具，直接跑扫描器最快"——恰恰相反，工具最容易被用，前四部分才最容易被跳过。

- 答"威胁模型"——很准。没有威胁模型，测试就是无的放矢，扫描器报什么修什么。

- 答"贯穿 SDLC"——也对，很多团队只在验收前扫一遍。

- 追问收口：所以这一节的顺序本身就在纠偏：**先想清楚威胁，再谈工具**。

易错点 / 考点：① 五个部分的名称要能默写；② 常考结构题："安全性测试包括哪些内容"；③ 易错：把"安全性测试"等同于"渗透测试"——渗透只是方法之一，还有静态分析、模糊测试、功能验证；④ 记住主线句，答题时用它开头很像样。

过渡：从第一块开始——威胁模型与安全漏洞模式。

【7.44】威胁模型与安全漏洞模式（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是一个分隔，标志我们进入"建模"阶段。我用一分钟说清"威胁模型"到底是什么，因为这个词听着玄。

威胁模型就是一句话：**谁，会从哪，用什么办法，来伤害这个系统，会造成什么后果。** 它由四部分构成：**资产**（要保护什么：用户数据、资金、密钥）、**对手**（谁会来：外部攻击者、内部人员、自动化脚本）、**攻击面**（从哪进：登录口、上传口、接口、第三方集成）、**缓解措施**（现在有什么防线）。

为什么要建？因为**安全测试的资源永远不够**，不可能把所有攻击试一遍。威胁模型的作用是**排序**：把"高价值

资产＋低门槛入口"排前面先测。

它还有一个副产品：**会暴露设计上的假设**。比如"我们假设内网可信"——一旦不成立（终端中毒、VPN 被撞库），整个防线就塌了。**漏洞十有八九是"一条没写下来的假设"破了。**

板书：中间写"威胁模型＝谁・从哪・怎么・后果"。四要素：资产（数据・资金・密钥）｜对手（外部・内部・自动化）｜攻击面（登录・上传・接口・第三方）｜缓解措施（现有防线）。右侧写一句："漏洞＝一条没写下来的假设破了"。

提问：一个只在内网使用的系统，威胁模型里"对手"应该写谁？

预设回答：

- 答"没有对手，内网很安全"——这是最危险的答案。内网里至少有内部人员、有中毒的终端、有被钓鱼的账号。

- 答"内部人员和被攻陷的终端"——对，这是内网系统最主要的威胁源。

- 答"还要考虑供应链和第三方接入"——补得好，VPN、外包、运维通道都是入口。

- 追问收口：所以"内网可信"这条假设，必须写进威胁模型并明确它的**失效条件**。

易错点 / 考点：① 威胁模型的四要素（资产、对手、攻击面、缓解）要能说全；② 高频错答："内网系统不需要安全测试"——错，威胁源主要来自内部与横向移动；③ 加分概念：**"假设失效"是漏洞的常见成因**；④ 记住威胁模型的作用是**排序**，不是"写文档交差"。

过渡：模型里最核心的一格是"资产会被怎么伤害"，我们先从"漏洞"这个词的定义开始。

【7.45】安全性缺陷 — 漏洞（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给"漏洞"下定义，课件用的词是**脆弱性（vulnerability）**，又称弱点或漏洞。原文是这样的：脆弱性是 IT 系统中存在的**可能被威胁利用造成损害的薄弱环节**，脆弱性一旦被威胁成功利用就可能**违犯安全策略**，对 IT 系统（资产）造成损害。

这个定义里有四个要素，必须拆开讲，因为考试就考这四个。

第一个，它是"薄弱环节"——注意，漏洞本身**不是损害**，它是一个"机会"。一扇没锁的窗不是盗窃，但它是盗

窃的机会。这个区别很重要：修漏洞是在**关门**，不是在**抓贼**。

第二个，它必须"被威胁利用"——也就是说，漏洞要造成损害，需要**有人（或某个程序）去用它**。这就解释了为什么同一个漏洞，在隔离的内网里风险低，在公网上风险高：**风险 = 漏洞 × 威胁**。

第三个，后果是"违犯安全策略"——安全策略是我们希望系统遵守的规则（谁能看什么、谁能改什么）。漏洞被利用，本质是**系统做了策略不允许的事**。

第四个，损害的对象是"资产"——数据、资金、服务能力、声誉。

再补一个实践概念：业界常用 **CVE 编号**来标识公开漏洞，用 **CVSS 分数**来量化严重程度。课件没有展开评分细节，你只要知道：**同一个系统里的漏洞要按严重程度排序修**，不能"看见一个修一个"。

最后强调一个观念：**漏洞是"系统的属性"，不是"某次攻击的事件"**。它在被利用之前就一直存在——**这正是测试能提前发现它的原因**。

> **板书：**定义四要素：① 薄弱环节（漏洞≠损害，是机会）② 被威胁利用（风险＝漏洞×威胁）③ 违犯安全策略（做了不允许的事）④ 损害资产（数据・资金・服务・声誉）。右侧补：CVE 标识・CVSS 评分（按严重度排序修）。底部："漏洞是系统属性，不是攻击事件"。

提问：一个漏洞"存在但从未被利用过"，算不算安全问题？

预设回答：

- 答"不算，没出事"——漏洞是属性，不是事件。没被利用只说明"还没人发现或还没人试"。

- 答"算，要被修"——对。修复的依据是**风险**（可能造成多大损害），不是"有没有已经出事"。

- 答"要看暴露面"——很专业：内网/公网、有没有认证、数据敏感度，都影响优先级。

- 追问收口：所以安全测试的产出是"**漏洞清单＋风险评级**"，不是"事故报告"。

易错点 / 考点：① 脆弱性定义的四个要素是简答重点，尤其"薄弱环节"和"被威胁利用"；② 高频判断："漏洞就是 bug"——两者有交集但不相等：bug 是"不符合预期"，漏洞是"可被利用来违反安全策略"；③ 记住公式化的理解：**风险 = 漏洞 × 威胁 × 影响**；④ CVE／CVSS 属于常识性补充，答题点到即可。

过渡：漏洞既然一直存在，那它是怎么被写进系统里的？课件把来源分成两大类。

【7.46】安全性：漏洞来源（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲漏洞从哪来，课件给了两个大分支：**故意**和**无意**。这个分法非常重要，因为**两者的修法完全不同**。

先说**故意**这一支，它又分两种。**恶意**的：木马病毒、陷门（trapdoor，也叫后门）、逻辑炸弹／定时炸弹。这些是**有人刻意埋进去的代码**——陷门是"留一个别人不知道的入口"，逻辑炸弹是"满足某个条件就发作"。这类问题的特点是：**它不在需求里，也不在代码规范里，靠常规测试几乎发现不了**，要靠代码评审、依赖审计、供应链检查。**非恶意**的：比如隐秘通道（covert channel）——设计者并没有恶意，但系统的某个机制可以被用来偷偷传信息（比如通过响应时间差异传递比特）。这类更隐蔽。

再说**无意**这一支，这是**绝大多数漏洞的真实来源**。课件列了六类：**校验错误、范围错误、串行错误、认证／鉴别错误、边界条件错误、其它可利用的逻辑错误**。

我逐类给个画面：**校验错误**是"该检查的没检查"——用户输入没验证就直接进 SQL；**范围错误**是"允许了不该允许的范围"——数组越界、权限范围给太大；**串行错误**是"顺序错了"——先扣款后校验库存，中间被打断就出问题；**认证／鉴别错误**是"把不该认的人认成自己人"——弱口令策略、会话不过期、token 可预测；**边界条件错误**是"临界值没处理"——空字符串、0、最大值、超长输入；**其它逻辑错误**是"业务流程可以被绕"——比如把价格改成负数、跳过支付步骤直接回调。

关键结论：**无意造成的漏洞占绝大多数，而它们几乎都落在"输入校验"和"状态检查"上**。写代码时反复问两句——"这个输入我信得过吗""这个状态我检查了吗"——就能挡掉一大半。

这也解释了为什么安全测试要**从攻击者角度构造输入**：你要测的正是"开发者没想到的输入"。

> **板书：**树形两支。**故意**：恶意（木马病毒・陷门/后门・逻辑/定时炸弹）｜非恶意（隐秘通道）。**无意**：校验错误｜范围错误｜串行错误｜认证/鉴别错误｜边界条件错误｜其它可利用逻辑错误。底部两句："大部分漏洞是无意的"＋"反复问：这个输入信得过吗？这个状态检查了吗？"

提问：六类无意漏洞里，"串行错误"和"逻辑错误"有什么区别？各举一例。

预设回答：

- 答"都是逻辑问题，差不多"——不一样。串行错误强调**执行顺序与中间状态**，逻辑错误强调**规则本身能被绕过**。
- 答"串行：先发货后扣款；逻辑：价格可改成负数"——很到位，两个例子都准确。

- 答"串行错误是并发问题"——有关系但不完全等同：并发会放大串行错误，但串行错误本身是"步骤顺序/事务边界"的问题。

- 追问收口：判断口诀——**看"顺序与中间态"的，是串行；看"规则能不能被绕"的，是逻辑**。

> **易错点 / 考点：**① 两大类（故意／无意）与六小类要能对上，填空题高频；② **陷门／后门**与**逻辑炸弹**的定义要能区分；③ 高频判断："漏洞都是黑客植入的"——错，绝大多数是开发无意引入；④ 记忆抓手：把六类无意漏洞归成"**校验、范围、顺序、身份、边界、业务**"六个字。

> **过渡：**漏洞是"门没锁"，接下来看"贼怎么进来"——攻击的流程。

【7.47】危险、攻击手段（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页把攻击过程拆成三段：**预攻击、攻击、后攻击**。这个划分是安全测试的"地图"，因为**你可以在每一段设防，也可以在每一段做检测**。
>
> **第一段，预攻击（扫描）**。攻击者在没动手之前先做功课。课件列的手段有：**常规信息获取**（搜索引擎、社会工程学）、**漏洞扫描**、**信息收集**（ping、telnet、whois、traceroute）、**端口扫描**（snmp、superscan 这类工具）。这一段的特点是**全是被动或半主动的探测**，很多情况下**不会触发业务异常**，所以最难被发现——你的日志里只有一堆 404 和端口探测。
>
> **第二段，攻击**。真正动手。课件列了：**DoS**（拒绝服务）、**远程渗透**、**口令猜测**、**远程溢出**（比如 getadmin 这类提权工具）、**应用攻击**（比如 SQL 注入）。
>
> **第三段，后攻击**。进去了之后干什么：**权限提升**（本地溢出、口令嗅探）、**留后门**、**痕迹擦除**。这一段最能说明一个残酷的事实：**"被入侵"和"被发现"之间往往隔着几个月**，因为攻击者会清痕迹。
>
> 为什么测试人员要背这个三段式？两个用处。**第一，设计检测点**：预攻击阶段你在日志和 WAF 上做检测（异常扫描、高频 404）；攻击阶段你在输入校验和权限上做拦截；后攻击阶段你在文件完整性、账号行为上做监控。**第二，做渗透测试时按这个顺序走**：先信息收集，再漏洞利用，最后评估影响——这就是标准的渗透流程。
>
> 还要说清一件事：**防御不能只做中间一段**。很多团队的安全投入全在"攻击阶段"（WAF、参数校验），结果

> 预攻击阶段信息泄露（后台地址、测试账号、源码备份），后攻击阶段无人察觉（没有完整性监控）。**真正的防线是全程的。**

> **板书：**三段：**预攻击（扫描）**：常规信息获取・漏洞扫描・社会工程学・搜索引擎・信息收集（ping/telnet/whois/traceroute）・端口扫描。**攻击**：DoS・远程渗透・口令猜测・远程溢出・应用攻击（SQL注入）。**后攻击**：权限提升・后门・痕迹擦除。右侧对应写"检测点：日志/WAF → 输入校验/权限 → 完整性/行为监控"。

提问：为什么"预攻击"阶段最容易被防守方忽略？

预设回答：

- 答"因为还没造成损害"——对，这一段没有业务异常，告警系统不响。

- 答"因为探测看起来像正常访问"——很准确。扫描一个不存在的路径，在你的日志里就是一次 404。

- 答"因为防守方只盯着攻击"——点到了本质：资源都投在了"拦"，没投在"看"。

- 追问收口：所以预攻击阶段的防守手段是**情报与异常检测**——比如大量 404、异常 UA、非常规时间的访问。

易错点 / 考点：① 三段的名称与各自手段要能对应，选择题高频；② **社会工程学属于预攻击**（信息获取），这一点常被错放到"攻击"；③ 高频判断："渗透测试就是直接打"——错，前期信息收集往往占一半以上时间；④ 记住"防御要覆盖三段"，这是案例题的加分点。

过渡：攻击成功之后，会造成哪些类型的安全问题？课件列了一整页。

【7.48】攻击带来的安全性问题（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页列攻击造成的后果，一共九条。我不逐字念，我把它们归成四组，你按组记更牢。
>
> **第一组，信息层面的问题：信息泄露、破坏信息的完整性、业务数据流分析。**信息泄露是"不该看的看到了"；完整性破坏是"不该改的改了"；数据流分析更有意思——它不一定要看到内容，**只看通信的模式**（谁在什么时候和谁通信、数据量多大）就能推断出商业情报。这一条最能说明"加密不等于安全"。
>
> **第二组，可用性问题：拒绝服务（DoS、DDoS）**。让你的服务不可用，是成本最低、效果最直接的一种攻击。

第三组，访问控制与身份问题：**非法使用（非授权访问）**、**窃听**、**假冒**、**旁路控制**、**授权侵犯（内部攻击）**、**抵赖（来自用户的攻击）**。这一组最杂，但可以抓住两条线：**"身份不可信"**（假冒、非授权访问）和**"行为不可追溯"**（抵赖、旁路控制）。内部攻击单列出来，是因为**内部人本来就有权限**，他做的事在系统眼里"完全合法"。

第四组，环境与合规问题：**计算机病毒**、**恶意软件**，以及**信息安全法律法规不完善**。最后这一条很多人觉得"不算技术问题"，但它恰恰是最现实的：法规不完善意味着**取证难**、**追责难**、**赔偿难**，这会让攻击的收益更高、成本更低。

九条合起来，你会发现它们全都能映射回**安全目标**：保密性（信息泄露、窃听）、完整性（篡改）、可用性（DoS）、真实性（假冒）、不可抵赖性（抵赖）。下一页要讲的 STRIED，其实就是把这九条**重新按攻击手法分类**。

板书：四组。信息：信息泄露｜完整性破坏｜业务数据流分析。可用性：DoS／DDoS。访问与身份：非法使用（非授权访问）｜窃听｜假冒｜旁路控制｜授权侵犯（内部）｜抵赖。环境与合规：病毒/恶意软件｜法律法规不完善。右侧竖写映射："保密・完整・可用・真实・不可抵赖"。

提问："业务数据流分析"看不到内容，为什么也算安全问题？

预设回答：

- 答"能看到内容才叫泄露"——不一定。通信模式本身就是情报：某公司半夜向某服务器传了 10GB，不用看内容也知道发生了大事。

- 答"它泄露的是行为模式"——准确。这类攻击防不住，因为它依赖的是**元数据**。

- 答"可以通过流量填充来对抗"——很专业，这就是对抗流量分析的常见手段之一。

- 追问收口：所以"加密"保护的是**内容**，不保护**模式**；两者要分开设计。

易错点 / 考点：① 九条要能说全（填空/多选高频），归类法能帮你记；② **授权侵犯（内部攻击）**和**抵赖**这两条最容易漏；③ "业务数据流分析"是理解性考点，要能解释"元数据也是信息"；④ 记住这九条与五个安全目标的映射，下一页和 7.55 都会用到。

过渡：把这九条按攻击手法重新归类，就得到安全领域最常用的一个模型。

【7.49】软件系统安全威胁：STRIED（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是安全领域最有名的一个分类模型，课件写作 **STRIED**，业内更常见的拼法是 **STRIDE**——字母顺序写法不同，内容一致，答题时按课件写、能说清每个字母的含义就行。

六个字母对应六类威胁，我一个一个讲，每个配一个具体画面。

S——假冒 ID（Spoofing Identity）：冒充别人。冒充用户登录、冒充服务端、伪造 token。**防它靠"认证"**，比如多因素认证、证书。

T——篡改（Tampering）：改了不该改的东西。改请求参数、改数据库、改传输中的数据。**防它靠"完整性校验"**，比如签名、哈希校验、HTTPS。

R——抵赖（Repudiation）：干了不承认。"我没下过这个单""我没批过这笔款"。**防它靠"不可否认性"**，比如审计日志、数字签名。

I——信息泄露（Information disclosure）：不该看到的看到了。报错里带堆栈、接口返回多了字段、日志里存明文密码。**防它靠"保密性"**，比如加密、脱敏、最小化返回。

D——拒绝服务（Denial of Service, DoS）：让你用不了。**防它靠"可用性"设计**，比如限流、降级、容量冗余。

E——特权升级（Elevation of Privilege, EoP）：本来只能看自己的，结果能看所有人的；本来是普通用户，结果成了管理员。**防它靠"授权"**，比如每次操作都校验权限、最小权限原则。

这个模型最实用的地方在于：**它把"威胁"和"防线"一一对应**。你拿一张表，左边六个威胁，右边六种防线（认证、完整性、不可否认、保密、可用、授权），逐项问"我们有吗"。这就成了一份**可执行的安全检查单**。

当一个系统说"我们做了加密"，你要追问：**认证**呢？**授权**呢？**审计**呢？加密只覆盖了 I 那一格。

板书：六行表格：S 假冒 ID → 认证｜T 篡改 → 完整性｜R 抵赖 → 不可否认（审计・签名）｜I 信息泄露 → 保密（加密・脱敏）｜D 拒绝服务 → 可用（限流・冗余）｜E 特权升级 → 授权（最小权限・逐次校验）。底部："威胁 ↔ 防线，一一对应＝检查单"。

提问：一个系统所有数据都加密存储了，能不能说"安全威胁都防住了"？

预设回答：

- 答"能，加密最强"——加密只覆盖 I（信息泄露）那一格，其他五类它管不着。

- 答"不能，还要认证授权"——对。假冒、抵赖、特权升级都得靠另外的机制。

- 答"还要防 DoS"——补得好，可用性是另一条线。

- 追问收口：所以评估安全不能问"有没有加密"，要问"**六类威胁各有哪道防线**"。

> **易错点 / 考点：** ① 六个字母与六类威胁的对应必须默写，这是本节最高频考点；② 拼写差异（课件 STRIED ／业界 STRIDE）别纠结，写清含义即可；③ 高频错答："加密就安全了"——加密只对应保密性；④ 记忆口诀建议自己造一句（如"假、改、赖、泄、拒、升"），考试时按顺序默写。

> **过渡：** 分类模型讲完，我们看具体的漏洞长什么样——课件列了一批经典模式。

【7.50】一些常见的安全漏洞模式（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页列了一串经典漏洞模式，它们是安全测试的"题库"——做测试时你就拿这些模式去对照系统。
>
> **第一类，缓冲区溢出漏洞**。程序往一个固定大小的缓冲区里写超长数据，多出来的部分覆盖了相邻内存——轻则崩溃，重则**被植入并执行攻击者代码**。它是 C／C++ 系统的经典问题，下一页专门看。
>
> **第二类，图像格式（WMF／TIFF／PNG）漏洞**。这一类很有意思：漏洞不在你的代码里，在**你用来解析图片的库**里。用户上传一张特制图片，解析时触发溢出——**你的攻击面包括你依赖的所有第三方组件**。
>
> **第三类，竞争条件漏洞（race condition）**。两个操作同时进行，出现"检查时还没问题，使用时已被改了"（TOCTOU）。典型场景：提现时先查余额再扣减，中间被并发插一刀，就能超额提现。
>
> **第四类，SQL 注入**。把用户输入当成了 SQL 语句的一部分。这是 Web 时代最经典的漏洞，下一页之后会有专门的内容。
>
> **第五类，跨站脚本攻击（XSS）**。把恶意脚本种进"给别人看的页面"，别人一浏览就执行。
>
> **第六类，跨站请求伪造（CSRF）**。借用户的身份，让他在不知情的情况下发出请求——用户是登录状态，攻击者只需要让他"点一下"。
>
> **第七类，Frame-proxy 攻击漏洞**。用框架把真实页面套住，用户在"看起来是你家"的页面里输入密码，属于

界面伪装。

第八类，并发漏洞。并发下的重复提交、库存超卖、幂等缺失。**它在功能测试里几乎看不出来，只在并发时出现**。

我要强调一个共同点：**这些漏洞都不是"功能写错了"，而是"边界没守住"**——输入边界（溢出、注入）、身份边界（XSS、CSRF）、时序边界（竞争、并发）、依赖边界（图像库）。所以防漏洞的思路可以归成一句：**凡是跨越边界的地方，都要检查。**

> **板书：**竖排八类：缓冲区溢出｜图像格式（WMF/TIFF/PNG）｜竞争条件（TOCTOU）｜SQL 注入｜XSS｜CSRF｜Frame-proxy｜并发漏洞。右侧写四类边界：输入边界・身份边界・时序边界・依赖边界。底部："跨边界处必检查"。

提问：XSS 和 CSRF 都涉及"跨站"，它们的区别在哪？

预设回答：

- 答"都是跨站攻击，差不多"——最常见的混淆。两者机制完全不同。

- 答"XSS 是把脚本种进页面，CSRF 是借用户身份发请求"——准确。XSS 攻击的是**页面的可信性**，CSRF 攻击的是**请求的可信性**。

- 答"XSS 需要用户浏览，CSRF 需要用户已登录"——补充得好，两者都依赖用户，但依赖的东西不同。

- 追问收口：一句话区分——**XSS 让"你的浏览器执行我的代码"，CSRF 让"你的浏览器用你的身份发我的请求"**。

易错点 / 考点：① 八类模式要能认（选择题常从里面挑）；② **XSS vs CSRF** 是最高频的混淆点，必须能一句话区分；③ 竞争条件要能举"先查后改"的例子，并说出 TOCTOU 这个术语；④ 加分：把八类归到"四种边界"上，说明你不只是背名词。

过渡：八类里的第一类最古老也最凶险，我们单独看它。

【7.51】常见的 C/C++缓冲区溢出问题（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页专讲缓冲区溢出。为什么要给一个"老问题"单独一页？因为它是**理解"漏洞如何变成攻击"的最佳样本**——它把"写错一行代码"和"攻击者拿到服务器权限"之间的链条完整展示出来了。

先说原理。C 语言的数组不检查边界：你声明一个 8 字节的缓冲区，往里拷 20 字节，编译器不会拦你，多出来的 12 字节就**盖在相邻的内存上**。而相邻的内存里，往往存着**函数的返回地址**——也就是"这个函数执行完该回到哪里继续跑"。

于是攻击链条就成立了：**构造超长输入 → 覆盖返回地址 → 让它指向攻击者自己的代码（或系统里的某个函数）→ 程序"返回"时就跳到了攻击者指定的地方**。这就是"栈溢出攻击"。它厉害的地方在于：**不需要你先登录，一个输入字段就够了**。

常见的出错点有哪些？**第一，`strcpy`／`sprintf`／`gets` 这类不检查长度的函数**——它们只负责拷，不负责数。**第二，长度计算错误**：算错一个字（比如忘了结尾的 `\0`），就会多写一个字节。**第三，整数溢出导致长度校验失效**：先算"还需要多少空间"，结果这个数溢出成了负数，校验就绕过去了。**第四，把用户输入当格式串**：`printf(user_input)` 而不是 `printf("%s", user_input)`，这是格式化字符串漏洞。

怎么防？三条：**用安全版本函数**（`strncpy`、`snprintf`）；**做严格长度校验**，并且校验要在**所有入口**做；**用编译器和运行时的保护机制**（栈保护、地址随机化、不可执行栈）。这三条要一起上——**任何单一防线都能被绕过，组合起来才有意义**。

最后一句：现代语言默认做了边界检查，**但"边界检查"不等于"业务校验"**——数组越界没有了，"金额能不能为负"还得自己管。

> **板书：**原理链：超长输入 → 覆盖相邻内存 → 覆盖返回地址 → 跳转到攻击者指定代码 → 拿到控制权。四个常见错点：不安全函数（strcpy/sprintf/gets）｜长度算错（忘 \0）｜整数溢出绕过校验｜格式化字符串。三条防线：安全函数｜全入口长度校验｜编译器/运行时保护（栈保护・ASLR・NX）。

提问：用 Java 写的系统，是不是就不用担心缓冲区溢出了？

预设回答：

- 答"是，Java 有边界检查"——语言层面确实挡住了数组越界，但**不等于没有同类问题**：字符串拼接过多导致内存暴涨、超大请求体导致 OOM，本质都是"没限制输入规模"。

- 答"不是，还要防依赖库"——很专业：JVM 本身、图像解析库、本地方法调用（JNI）都可能引入原生层漏洞。

- 答"还要做业务层校验"——对，语言给的是"内存安全"，业务给的是"业务安全"。

- 追问收口：所以正确说法是——**换了语言，风险的形态变了，风险本身没消失**。

易错点 / 考点：① 溢出的原理链条要能讲清（覆盖返回地址是关键环节）；② 常见不安全函数要能列举；③

高频判断："用高级语言就没有溢出问题"——错，要看是否涉及原生库与输入规模；④ 三条防线要能说全，注意强调"组合防御"。

过渡：单类漏洞看完了，再看行业给出的"最该防的十类"——OWASP Top 10。

【7.52】OWASP Top10 漏洞模式（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲 **OWASP Top 10**。课件给的信息很简洁：**OWASP 是 Open Web Application Security Project，一个开放的 Web 应用安全项目**，Top 10 是它发布的"最严重的 Web 应用安全风险"清单，官方站点在课件里给了。

我要讲清三件事：它是什么、它为什么有影响力、以及**它不能干什么**。

它是什么：一份**按风险排名的清单**，每隔几年更新一版。它的价值不在"新"，而在"共识"——全世界做 Web 安全的人，都拿它当共同语言。你跟同事说"这是 A01 类问题"，他就知道你在说什么。

它为什么有影响力：因为它**来自真实数据**——OWASP 收集大量应用的实际漏洞统计，按"出现频率 × 影响程度"排序。所以它反映的不是理论上的危险，而是**现实中真被打穿的地方**。

它不能干什么：这一点最关键。**Top 10 不是测试用例集**，它只是"类别"。比如"注入"是一类，具体到你系统里，是哪个接口、哪个参数、怎么注入，清单不会告诉你。**它是地图上的"高危区域"，不是藏宝图上的"具体坐标"。**

那怎么用它？我建议三步：**第一步，拿 Top 10 的类别当检查表**，逐项问"我们有没有这一类风险"；**第二步，把每一类翻译成本系统的具体测试点**（哪个入口、什么输入、预期防线）；**第三步，按业务影响排序**，先测那些"一旦被打穿后果最重"的地方。

有一点要提醒：**清单会更新、条目会调整**，所以别把"第几条第几项"当死知识背。**要记的是那些长期在榜的类别**——注入、失效的访问控制、加密相关缺陷、跨站脚本、安全配置错误、组件漏洞这几类，多年来一直在里面。**这才是考点。**

板书：三问：是什么（按风险排名的 Web 安全风险清单・几年一更）｜为什么有影响力（来自真实数据・全球共识语言）｜不能干什么（不是用例集，是"高危区域"不是"具体坐标"）。用法三步：当检查表 → 翻

成本系统测试点 → 按业务影响排序。底部："长期在榜：注入・访问控制・加密・XSS・配置错误・组件漏洞"。

提问：把 OWASP Top 10 抄进测试用例库，是不是就等于做了安全测试？

预设回答：

- 答"是，照着测一遍就行"——Top 10 是类别，不是用例。抄进去的只是标题。

- 答"不是，要具体到接口和参数"——对。安全测试的落点是"这个接口的这个参数"，不是"注入这一类"。

- 答"还要结合业务风险"——补得好，同一个漏洞在支付系统和公告系统里的优先级完全不同。

- 追问收口：所以正确的用法是"**先按类别扫盲，再按业务翻译，最后按风险排序**"。

易错点 / 考点：① OWASP 的全称与性质要能答；② 高频错答："Top 10 就是十个漏洞"——是十**类**风险；③ 记住"来自真实数据、几年更新一版"这两个特征，能解释它的权威性；④ 别背具体条目的编号，背**长期在榜的类别**。

过渡：威胁和漏洞都看完了，接下来要回答最实际的问题：这些该怎么测？

【7.53】安全性测试的范围与方法（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是分隔，进入"范围与方法"。我用一分钟把这个小节的逻辑讲清楚，因为它是整个 7.2 的枢纽。

这一小节要回答三个问题：**测什么**（范围与目标）、**测到什么程度**（安全要求与等级）、**怎么测**（方法与任务）。

先看"测什么"。这里先立住**安全目标**：课件强调 CIA 三元组——保密性、完整性、可用性。所有安全测试最后都能归到这三条上。**没有目标，测试就变成"能想到什么测什么"。**

再看"测到什么程度"。这里要引入**安全等级**：内部公告系统和人脸识别支付系统，不可能用同一套标准——这就是后面要讲的五个安全保护等级。

最后是"怎么测"：**静态**（工具扫描、人工评审、专家评估）与**动态**（功能测试、模糊测试、渗透测试）；按视角又分**基于威胁**（外部：会发生什么）与**基于漏洞**（内部：存在什么）。

一句话总结这个小节：**先定目标（CIA），再定等级（多严），最后选方法（静态/动态、威胁/漏洞）**。

> **板书：**三问：测什么（安全目标 CIA＋真实性・不可抵赖）｜测到什么程度（安全等级）｜怎么测（静态：扫描・评审・专家评估；动态：功能・模糊・渗透；视角：基于威胁／基于漏洞）。底部："先定目标 → 再定等级 → 后选方法"。

提问：如果客户说"帮我们把系统安全测一遍"，你先问哪三个问题?

预设回答：

- 答"测什么范围"——第一个问题，范围不清就没法报价也没法交付。

- 答"要多严"——第二个问题，对应安全等级和验收标准。

- 答"用什么方法、给什么报告"——第三个问题，方法决定工作量和工具。

- 追问收口：再加一个隐藏问题——"**这次测试的授权和窗口**"，安全测试没有授权不能开工（回想 7.42）。

易错点 / 考点：① 安全目标的核心是 **CIA 三元组**，必须能全称+解释；② 静态方法与动态方法的划分要清楚（评审属静态）；③ 高频混淆：**基于威胁 vs 基于漏洞**——一个是外部视角、一个是内部视角；④ 记住"先目标、再等级、后方法"这个顺序，案例题按它组织答案很清晰。

过渡：从"目标"开始——安全目标到底包含什么。

【7.54】安全目标（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲安全目标，核心是 **CIA 三元组**：**保密性（Confidentiality）、完整性（Integrity）、可用性（Availability）**。课件还有一句话非常重要——**这些目标常常是互相矛盾的，因此需要在这些目标中达到最佳平衡**，并举了个例子：简单地阻止所有人访问一个资源，就可以实现该资源的保密性，但这样就不满足可用性。

我们一个一个说。**保密性**：信息只让该看的人看到。反例最好记——把数据库直接暴露在公网，保密性瞬间归零。**完整性**：信息不被非法篡改，且处理方法是准确、原始的。注意它不只是"防改"，还包括"改了能发现"。**可用性**：该用的时候能用。这一条最容易被低估，但 **DoS 攻击打的正是它**。

然后是这一页真正的考点：**三者会打架**。举三个真实的冲突场景。**冲突一**：为了保密，把日志里的敏感字段

全删了——出事之后没法追溯，**完整性（可审计）受损**。**冲突二**：为了可用，做多副本、多缓存——数据副本越多，**保密性和一致性越难保证**。**冲突三**：为了安全，加很多验证步骤——用户登录要过五道验证，**可用性直线下降**，用户开始用弱密码、开始把密码写在便签上，**整体安全反而变差**。

所以课件那句"最佳平衡"不是客套话，它是**安全工程的核心方法论：安全不是一个可以无限加码的维度，它是有成本的**。每加一道防护，都要问：挡住的威胁值多少？付出的可用性和体验代价多少？

再补三层目标：课件之外行业里常说的还有**真实性**（身份是真的）、**不可抵赖性**（做过不能否认）、**可追溯性**（能查到谁做的）。这三条在第 55 页会正式列出来，这里先埋个伏笔。

板书：中间写"CIA"：保密性（只让该看的看）｜完整性（不被非法篡改・改了能发现）｜可用性（该用能用）。两侧写"冲突"：删日志 vs 可审计｜多副本 vs 一致性｜多验证 vs 体验（弱密码反而更不安全）。底部："安全有成本，要的是平衡"。

提问："为了绝对安全，我们应该给所有操作都加二次验证"——这句话的问题在哪？

预设回答：

- 答"没问题，安全第一"——代价没算。验证越多，用户越烦，最后会用弱密码、会共享验证码，**实际安全度下降**。

- 答"影响可用性"——对，这正是 CIA 的冲突。

- 答"要按风险分级加验证"——专业做法：高风险操作（转账、改密码）强验证，低风险操作弱验证。

- 追问收口：所以安全设计的正确问法是"**这个操作的风险有多高，配多强的验证**"，不是"越多越好"。

易错点 / 考点：① CIA 三个词的中英文与含义必须默写；② **三者会冲突**这个观点是简答/论述高频点，一定要举例；③ 高频错答："安全越高越好"——忽略成本与可用性；④ 记住课件那个"阻止所有人访问"的极端例子，考试直接引用即可。

过渡：CIA 是三条主线，具体展开成要求有八条。下一页看全。

【7.55】安全性要求（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把安全要求摊成八条，比 CIA 更细。我逐条讲，并给每条一个"怎么测"的落点，因为**要求不能测就等**

于没写。

第一，保密性：确保信息只被授权人访问，**信息即使被截取也不能了解真实含义**。后半句是重点——它说明保密不只是"不给看"，还包括"看到了也看不懂"（加密）。**怎么测**：越权访问、抓包看明文、看返回字段是否多余。

第二，完整性：保护信息和处理方法的**准确性和原始性**，包括数据一致性，防止数据被非法用户篡改。**怎么测**：篡改请求参数看服务端是否校验、篡改传输中的数据看有没有签名。

第三，可用性：确保授权的用户在需要时可以访问信息。**怎么测**：压力下的可用性、故障时的降级能力。

第四，真实性：保证信息来源真实可靠。它和"保密性"不同——真实性问"你是谁、你说的真的是你说的吗"，对应认证与签名。**怎么测**：伪造 token、重放请求。

第五，不可抵赖性：用户对其行为不可否定。**怎么测**：关键操作有没有留痕、日志能不能被普通用户删改。

第六，可追溯性（Accountability）：确保实体的行动可被跟踪。它和不可抵赖是一对——**不可抵赖靠"签"，可追溯靠"记"**。**怎么测**：做一次敏感操作，看能不能从日志里还原出"谁、何时、从哪、做了什么"。

第七，可控制性：对信息的传播及内容具有控制能力。比如敏感数据不能导出、不能转发、水印。**怎么测**：尝试导出、截图、批量拉取。

第八，可审查性：对出现的网络安全问题提供**调查的依据和手段**。这一条是"事后能力"——出事之后能不能查清。**怎么测**：模拟一次事件，看现有日志和审计是否够用。

八条里前三条是 CIA，后五条是延伸。记忆抓手：**"保、完、可、真、不、追、控、审"**——念两遍就顺了。

> **板书：**八行：保密性（截取也看不懂）｜完整性（准确・原始・防篡改）｜可用性（该用能用）｜真实性（来源可靠）｜不可抵赖性（行为不可否认）｜可追溯性（行动可跟踪）｜可控制性（传播与内容可控）｜可审查性（调查有依据）。右侧每条对应一行"怎么测"。底部口诀："保完可真不追控审"。

提问："不可抵赖性"和"可追溯性"经常被当成一回事，它们的区别是什么？

预设回答：

- 答"一样，都是留痕"——不一样。一个是"你不能否认"，一个是"我能查到"。

- 答"不可抵赖靠签名，可追溯靠日志"——准确。前者是**事前约定**（你签了就得认），后者是**事后能力**（我能还原过程）。

- 答"日志也能做不可抵赖"——有争议但可讨论：普通日志管理员能改，所以强度不够；要不可抵赖必须用**防篡改**的手段（签名、只追加存储）。

- 追问收口：一句话——**不可抵赖是"法律意义上的证据"，可追溯是"技术意义上的记录"**。

> **易错点 / 考点：**① 八条要求要能说全，前三条（CIA）必答；② **不可抵赖 vs 可追溯**是本节最典型的混淆点；③ **可控制性、可审查性**这两条最容易漏，答题补上能加分；④ 每条都要能说出"怎么测"，这是从概念到实践的关键一步。

> **过渡：**八条要求、六个威胁，两套词容易混。课件用两页练习帮你对齐。

【7.56】练习（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页是练习，把两组词摆在一起让你配对：左边是**六类威胁**（假冒 ID、篡改、抵赖、信息泄露、拒绝服务、特权升级），右边是**六个安全要求／控制手段**（授权、保密、不可抵赖性、可用性、完整性、认证）。
>
> 我先不报答案，讲方法。这类配对题的关键是问一句：**"这个威胁，靠什么机制挡住？"** 注意右边给的是"控制手段"，不是"目标名称"——所以要落到"用什么办法"上。
>
> 给你三次示范，剩下的你自己对。
>
> **假冒 ID** 对应**认证**——因为假冒要解决的是"你到底是谁"，那就要有一个"验证身份"的机制。注意别答"授权"：授权是"你能干什么"，认证是"你是谁"。**这两个词是这一页最大的坑。**
>
> **篡改** 对应**完整性**——防改、能发现改。
>
> **抵赖** 对应**不可抵赖性**——留痕、签名，让他赖不掉。
>
> 剩下的：**信息泄露** 对应 **保密**；**拒绝服务** 对应 **可用性**；**特权升级** 对应 **授权**——因为特权升级的本质是"权限边界被突破"，防它靠的正是授权控制与最小权限。
>
> 你看，六个威胁、六个手段，正好一一对应。**这就是 7.49 讲的 STRIED 的另一半**：威胁模型之所以有用，就是因为它天然带着"该用什么防线"的答案。
>
> 最后提醒一句考试技巧：这类题**先配对最容易的**（篡改—完整性、拒绝服务—可用性），把确定的划掉，剩

下的自然就出来了。**别在难题上卡住**。

板书：左右两列连线：假冒 ID — 认证｜篡改 — 完整性｜抵赖 — 不可抵赖性｜信息泄露 — 保密｜拒绝服务 — 可用性｜特权升级 — 授权。旁边写一行红字："**认证＝你是谁；授权＝你能干什么**"。

提问：为什么"假冒 ID"配"认证"，而不是配"授权"？

预设回答：

- 答"因为假冒是身份问题"——对，身份问题用身份机制解决。

- 答"授权也能防假冒"——逻辑上不通：授权的前提是已经知道你是谁，身份都是假的了，授权无从谈起。

- 答"两个都要"——可以这么说，但**主责是认证**；考试要选最直接的那个。

- 追问收口：记住这条链——**先认证（你是谁）→ 再授权（你能干什么）→ 再审计（你干了什么）**。

易错点 / 考点：① 六组配对必须会，这是本节最直接的考点；② **认证 vs 授权**是安全领域第一号混淆点，务必分清；③ 做题技巧：先连确定的；④ 答题时最好写出"机制"而不只是名词，比如"抵赖→不可抵赖性→审计日志/数字签名"。

过渡：威胁与要求对齐了，接下来看需求怎么写——软件安全性需求包含哪些内容。

【7.57】练习（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页还是同一个练习，是答案页。我们不重复题目，我讲**这六组配对背后的三条规律**，你掌握了规律，下次遇到同样的题不用背。

规律一：威胁和防线是"一对一"的。六类威胁之所以能配上六个手段，是因为每一类威胁都精确对应一种防护机制。这不是巧合——STRIED 模型本来就是"按防护手段"设计的分类法。所以做题时不要瞎猜，问自己："**这个威胁要发生，缺的是哪一道防线？**"

规律二：相似的词要盯住差别。这一页藏着三组最容易混的词。**认证 vs 授权**：认证是"你是谁"，授权是"你能干什么"，假冒配认证、特权升级配授权。**完整性 vs 不可抵赖性**：完整性防"改"，不可抵赖防"赖"。**保密性 vs 真实性**：保密防"看"，真实防"假"。

规律三：目标词和控制词要分清。 列表里既有"保密""可用性""完整性"这种**目标名**，又有"授权""认证"这种**控制手段名**。答题时两者都对，但要能说清"用 XX 手段实现 XX 目标"。比如：**用认证实现真实性，用授权实现访问控制，用审计实现不可抵赖**。

这三条规律的意义超出这一道题。你在读任何安全标准时，都会看到"威胁—控制—目标"这三层词汇混着出现，能分清它们，你才算真的读懂了那份标准。

最后再补一句：**练习题做对不算本事，能在自己的代码里找到这六个威胁的影子才算。** 你回头看看自己写过的登录、下单、改资料的功能，哪一处可能被假冒？哪一处可能被篡改？

板书： 三条规律：① 威胁 ↔ 防线一对一（问"缺哪道防线"）② 盯住形近词（认证 vs 授权・完整性 vs 不可抵赖・保密 vs 真实）③ 分清目标词与控制词（用手段实现目标）。右侧写三条对应句式："认证→真实性｜授权→访问控制｜审计→不可抵赖"。

提问： 如果把"信息泄露"配成"完整性"，错在哪？

预设回答：

- 答"配错了，应该配保密"——对。信息泄露问的是"能不能看到"，对应保密性。
- 答"完整性也有关"——有点关系（泄露常伴随完整性受损），但**主责不对**。
- 答"完整性是防篡改的"——准确，一句话就把边界划清了。
- 追问收口：所以配对题要问"这个威胁**最直接**破坏了哪个目标"，选最直接的那个。

易错点 / 考点： ① 三组形近词（认证/授权、完整性/不可抵赖、保密/真实）必须能一句话区分；② 高频错答：把"信息泄露"配"完整性"——因为两者常同时发生，容易连带；③ 答题格式建议写成"威胁→目标→手段"三段式，得分更稳；④ 记住"用手段实现目标"这个句式，论述题很好用。

过渡： 威胁、目标、手段都清楚了，接下来看这些要求怎么写成"需求"。

【7.58】软件安全性需求（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页把安全需求分成两大类：**数据安全性**和**系统安全性**，每类下面给了一串具体项。这是"把安全目标翻译成开发任务"的关键一步。

第一类，数据安全性，六项：**加密、数据结构、安全存储、安全访问、认证、备份**。这六项其实覆盖了数据的整个生命周期：数据长什么样（**数据结构**——哪些字段敏感、怎么分级）、数据怎么存（**安全存储**——落盘加密、权限隔离）、数据怎么用（**安全访问**——谁能读谁能写）、谁能碰（**认证**）、传的时候怎么保护（**加密**）、丢了怎么办（**备份**）。**注意"备份"也算安全需求**——勒索软件盛行的今天，备份是最后的可用性防线。

第二类，系统安全性，五项：**用户身份认证、用户权限限制、异常处理、入口验证机制、系统强壮性设计**。

这五项值得逐条说。"**用户身份认证**"是门锁；"**用户权限限制**"是门内的分区（最小权限）；"**异常处理**"这一项最容易被当成"编程规范"，其实它是安全需求——**异常信息泄露是最常见的低危漏洞**：一个报错页把堆栈、数据库表名、框架版本全暴露给攻击者，等于给他画了张地图；"**入口验证机制**"是"所有外部输入都要在入口处校验"——注意是**所有入口**，包括 API、文件上传、消息队列、定时任务参数；"**系统强壮性设计**"是"打不坏"——异常输入不崩、压力下不崩、依赖故障时不崩。

这两类的分工我给你一句话：**数据安全性管"数据本身"，系统安全性管"访问数据的那个系统"**。两者缺一不可——数据加密做得再好，系统有个越权漏洞，一样全泄。

写成需求时还有一条规范：**每条安全需求都要可验证**。"系统要安全"没法测；"用户密码必须以加盐哈希存储，且任何接口不得返回密码字段"就能测。这和我们 7.18 讲的性能需求可测试性，是同一个道理。

> **板书：**两大类。数据安全性：加密｜数据结构｜安全存储｜安全访问｜认证｜备份（＝数据全生命周期）。系统安全性：用户身份认证｜用户权限限制｜异常处理｜入口验证机制｜系统强壮性设计。底部："数据管数据本身，系统管访问它的系统"＋"安全需求必须可验证"。

提问："异常处理"为什么会被列为安全需求，而不只是编码规范？

预设回答：

- 答"因为异常会崩"——那是可靠性角度。

- 答"因为报错会泄露信息"——准确。堆栈、SQL、框架版本、内网地址，全是攻击者的情报。

- 答"因为异常路径常被绕过校验"——很专业：很多越权、注入就发生在异常分支里，因为正常分支有校验、异常分支没有。

- 追问收口：所以异常处理的安全要求是两条——**对外不泄露细节，对内不跳过校验**。

易错点 / 考点：① 两大类及各自包含的项要能列举，填空题高频；② **备份属于安全需求**这一点常被漏；③ 异

常处理的双重要求（不泄露、不跳校验）是加分点；④ 高频判断："安全需求可以写得原则一些"——错，安全需求必须可验证。

过渡：需求讲完，我们看一个具体的场景——口令这件事有多少防范措施。

【7.59】示例：系统安全性问题（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页抛出一个问题：**对用户口令，有多少防范措施？** 这是个特别好的题目，因为口令是每个系统都有的东西，而它牵着一长串安全设计。

我们把口令的一生走一遍，每一步都问"这里能出什么事、该有什么措施"。

第一，设置的时候：有没有密码强度要求？有没有禁止使用弱口令库？有没有禁止与用户名相同？——对应"弱口令"这一类最高频的失陷原因。

第二，存储的时候：**绝对不能明文存**。要用**加盐哈希**（每个用户一个随机盐，用 bcrypt／scrypt／PBKDF2 这类慢哈希）。为什么要"盐"？防止彩虹表；为什么要"慢"？让暴力破解的成本高到不可行。这一条是开发最常犯的错——用 MD5 甚至明文。

第三，传输的时候：必须走加密通道（HTTPS），否则中间人直接拿走。

第四，使用的时候：登录失败要不要限制次数？要不要防暴力破解（验证码、锁定、延迟）？要不要记录失败日志？——对应"口令猜测"这类攻击。

第五，找回的时候：这是**最容易被忽略的一环**。安全问题答案能不能猜？重置链接有没有过期？重置令牌可不可预测？很多系统登录做得严，找回做得松，攻击者根本不猜密码，直接走"忘记密码"。

第六，变更与注销的时候：改密码要不要验证旧密码？异地登录要不要提醒？离职账号有没有及时停用？会话有没有超时？

走完这一圈你会发现：**"口令安全"不是一个功能，而是一条链，最弱的一环决定整体强度**。这也解释了为什么很多系统"密码策略很严，还是被盗"——因为链条的另一头（找回、会话）是漏的。

顺带说一句：现代系统越来越多地用多因素认证（短信、TOTP、生物特征）来降低"口令被猜中"的后果，这是"**不指望单点绝对可靠，而是加一层**"的思路。

> **板书：**口令一生六步：① 设置（强度・弱口令库・不与用户名相同）② 存储（加盐哈希・慢哈希・禁明文/MD5）③ 传输（HTTPS）④ 使用（失败限制・防爆破・日志）⑤ 找回（安全问题・链接过期・令牌随机）⑥ 变更/注销（验旧密码・异地提醒・停用・会话超时）。底部："口令安全是一条链，最弱一环决定强度"。

> **提问：**一个系统要求"密码必须 12 位含大小写数字符号"，但允许用"忘记密码"直接重置——它的口令安全算强还是弱？

> **预设回答：**

- 答"强，策略很严"——策略只覆盖了链条的一环。

- 答"弱，找回环节是短板"——对。攻击者不会去爆破 12 位密码，他会去猜安全问题、去撞重置链接。

- 答"要看重置流程怎么设计"——最完整：链接是否一次性、是否过期、是否绑定会话。

- 追问收口：所以评估口令安全要**走完整条链**，任何一环松，前面几环的严格都白费。

> **易错点 / 考点：**① 口令的六个环节要能说全，**"找回"最容易被漏**；② 加盐哈希的三要素（盐、慢哈希、禁明文）必答；③ 高频判断："密码复杂度要求高就安全"——链条思维才是正确答法；④ 记住"最弱一环决定整体强度"，这句话可以套用到所有安全设计题上。

> **过渡：**口令只是入口。系统层面还有一套基本机制，课件用主体、客体这些词来描述。

【7.60】系统安全的基本机制（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页讲系统安全的基本机制，用的是访问控制的经典模型。几个术语先立住，它们是后面所有讨论的语言。
>
> **主体（Subject）**：**主动**的一方——用户、进程。它发起操作。
>
> **客体（Object）**：**被动**的一方——文件、存储设备、数据库表。它被操作。
>
> **访问**：主体对客体做什么——**读、写、执行**。
>
> **权限**：允许不允许做某个访问。**权限是"主体×客体×操作"的三元组**，这一点必须记牢：同一个用户对 A 文件可读、对 B 文件不可读，这才叫权限；笼统地说"这个用户有权限"是不成立的。

再往上三层。**安全策略**：一整套规则，规定"谁在什么条件下可以做什么"。它是**管理层的意图**，比如"财务数据只有财务部能读"。**安全审计**：把"谁做了什么"记下来，用于事后追查。**强制访问控制（MAC）**：由系统统一按标签强制执行的访问控制，用户自己不能改——与之相对的是自主访问控制（DAC，文件主自己决定给谁权限）。MAC 更严，常用于高安全等级的系统。

把这几个词串成一句话：**主体按安全策略，对客体发起访问；权限决定这次访问能不能做；做了之后由审计记录下来；强制访问控制保证策略不被绕过。**

为什么测试人员要懂这个模型？因为**越权漏洞就是这三元组里的某一项被破坏了**：水平越权是"主体换了、客体不该给他"，垂直越权是"操作超出了他的角色"，未授权访问是"根本没验主体"。你拿这个三元组去设计用例，比凭感觉点要系统得多。

板书：画一个三元组：**主体（用户・进程）× 客体（文件・设备・表）× 操作（读・写・执行）= 权限**。上面一层写"安全策略（规则）"，下面一层写"安全审计（记录）"，右侧写"强制访问控制 MAC（系统统一强制）vs 自主访问控制 DAC"。底部："越权＝三元组被破坏"。

提问：一个用户能看到别人的订单，这属于哪一类越权？用三元组怎么描述？

预设回答：

- 答"水平越权"——对，同角色之间横向看别人的数据。

- 答"客体选错了，或者说客体没做归属校验"——很准确：请求里传了别人的订单 ID，服务端没校验"这个订单是不是你的"。

- 答"权限没做数据级隔离"——也对，很多系统只做了功能权限（能不能进这个页面），没做数据权限（能看哪些行）。

- 追问收口：所以修这类漏洞的通用做法是——**每次访问客体都校验归属，不信任客户端传来的 ID**。

易错点 / 考点：① 主体／客体／访问／权限四个术语的定义，以及"权限是三元组"这个理解；② **MAC 与 DAC** 的区别是高频考点；③ 安全策略与安全审计的定位（一个定规则、一个记事实）；④ 加分：能用三元组解释水平越权与垂直越权。

过渡：机制有了，但安全还要靠"防线"来落地。课件给了三道。

【7.61】安全三大保障机制：预防、恢复、支撑（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页把安全保障归成三类机制：**预防、恢复、支撑**。这个三分法很好用，因为它回答的是"安全体系由哪几种能力组成"。

第一，预防。目标是"不让坏事发生"。手段包括：身份认证、访问控制、输入校验、加密、防火墙、权限最小化。**预防是大家最熟悉的一类**，也是投入最多的一类。但要注意：**预防永远有漏**——你不可能预判所有攻击手法，所以只有预防是不够的。

第二，恢复。目标是"出事了能快速回到正常"。手段包括：备份与恢复演练、冗余与故障切换、应急响应预案、回滚机制。**恢复能力是"承认自己会被打穿"之后的设计**——这是安全观念上很重要的一次转变：从"我要建一堵打不穿的墙"到"我要假设墙会被打穿，但我要能快速修好"。

第三，支撑。这一层最容易被忽略，它指的是让前两类能力**持续有效**的基础工作：安全策略与制度、安全培训、审计与监控、漏洞管理流程、合规检查。**支撑是"让安全可持续"的那一层**——没有它，预防措施会随着系统演进而失效（新接口没加校验），恢复措施会随着人员流动而失传（没人知道备份怎么恢复）。

三类的关系：**预防降低事件发生的概率，恢复降低事件的损失，支撑保证前两者长期有效**。三者缺一，体系就是跛的。现实中最常见的失衡是：**预防投入 90%，恢复和支撑几乎没有**——结果一次成功的攻击就变成灾难。

一个检验方法：问团队三个问题——"我们最可能被怎么打穿？""打穿之后多久能恢复？""谁来保证这些措施一年后还有效？"三个问题答不上来，说明某类机制缺失。

> **板书：**三列。预防（认证・访问控制・输入校验・加密・防火墙・最小权限）→ 降低概率。恢复（备份・冗余切换・应急预案・回滚）→ 降低损失。支撑（策略制度・培训・审计监控・漏洞管理・合规）→ 保证长期有效。底部："预防降概率，恢复降损失，支撑保长效"。

提问：一个团队把预算全花在买安全设备上，没有备份也没有应急演练。这个体系缺什么？

预设回答：

- 答"缺恢复能力"——对，一旦被打穿就是灾难，因为回不去。
- 答"缺支撑"——也对，没有制度和流程，设备买了也没人用对。

- 答"预防也未必到位"——有道理：设备堆着不等于配置正确，"买了"和"生效"是两件事。

- 追问收口：所以评估安全体系要看三类机制**是否齐全**，不能只看"花了多少钱"。

> **易错点 / 考点：** ① 三类机制的名称与代表手段要能对上；② 高频判断："安全就是防住攻击"——只覆盖预防，漏了恢复与支撑；③ **"假设会被打穿"这个观念转变**是理解题常考；④ 加分：能说出"预防降概率、恢复降损失、支撑保长效"这句总结。

> **过渡：** 三类机制要落地，得有成文的策略。课件给了一份完整的策略清单。

【7.62】完整的安全策略示例（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是一份"完整安全策略"的清单，条目很多，看着杂，但**它有结构**。我帮你把结构找出来，你就不用死记。
>
> 我们按"管什么"把这些条目归类。
>
> **第一组，密码与身份类**：**口令保护**、**可接受的使用控制**（谁能用什么）、**敏感信息控制**。这一组管"人怎么证明自己、能碰什么"。
>
> **第二组，网络与远程接入类**：**外部网络访问控制**、**远程访问和 VPN 安全控制**、**模拟线路和拨入访问控制**、**无线通信控制**、**Internet DNS 设备控制**、**路由器安全管理**、**第三方网络连接协议**。这一组管"边界"——凡是"从外面连进来"的路，都要有策略。**注意"无线通信控制"和"拨入访问"这两条**，它们最容易被忘，因为现在大家默认"网络就是网线"。
>
> **第三组，主机与服务器类**：**服务器安全管理**、**数据库信任编码**（数据库层的可信代码与权限约束）。
>
> **第四组，流程与管理类**：**防病毒流程**、**审核和风险评估**、**引入评估控制**（引进新系统/新供应商时的安全评估）、**应用程序提供上（ASP）控制**、**自动转发电子邮件控制**、**可接受的加密控制**。
>
> 归类之后你会发现，这份清单其实在回答四个问题：**谁能进（身份）、从哪进（边界）、进来能干什么（主机与数据）、怎么管（流程）**。
>
> 另外注意"**可接受的**"这个词反复出现——可接受的加密控制、可接受的使用控制。它传达一个很重要的态度：**安全策略不是"越严越好"，而是"明确划出可接受范围"**，让员工知道什么能做、什么不能做。**没有明确边界**

的"严"，最后会变成"大家都不遵守"。

> **板书：**四组归类：① 密码与身份（口令保护・使用控制・敏感信息）② 网络与远程接入（外部访问・VPN・拨入・无线・DNS・路由器・第三方连接）③ 主机与服务器（服务器管理・数据库信任编码）④ 流程与管理（防病毒・审核与风险评估・引入评估・ASP・邮件转发・加密控制）。底部四问："谁能进・从哪进・能干什么・怎么管"。

提问：为什么安全策略里要写"可接受的加密控制"，而不是直接写"必须使用最强加密"？

预设回答：

- 答"最强加密性能差"——是一个原因，但不是核心。

- 答"要划清可接受范围"——对。"可接受"意味着给出选项集合和判断标准，让执行者能落地。

- 答"因为场景不同要求不同"——很准确：传输、存储、备份的加密要求本来就不一样。

- 追问收口：所以策略的写法是"**给出可接受区间＋判断依据**"，而不是一句绝对化的口号。

易错点 / 考点：① 策略清单的四组归类法比死记更容易得分；② **"可接受的"这个措辞的含义**是理解题考点；③ 高频漏项：无线通信控制、拨入访问控制、数据库信任编码；④ 记住策略的四个问题（谁能进、从哪进、能干什么、怎么管），可用于组织论述题答案。

过渡：策略是"规矩"，那么测试要验证到什么程度？先看安全性测试的目标。

【7.63】安全性测试的目标（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页把安全性测试的目标分成两层：**通常的目标**和**进一步的目标**，并且点明它们都服务于一件事——**数据的机密性、完整性、可用性**，覆盖**数据存储、传输和处理**三种状态。

先看通常的目标，四条。

第一，（强制）访问控制。验证"该进的能进、不该进的进不去"。注意括号里"强制"两个字——它指的是不依赖用户自觉，而由系统强制执行。测法就是换身份、换角色、换参数，看边界守不守。

第二，身份鉴别。验证"你是谁"这件事判断得对不对。包括口令策略、多因素、会话管理、token 有效性。

测法是伪造、重放、过期复用。

第三，完整性保护。验证数据在存储和传输中不被篡改，且篡改能被发现。测法是改参数、改报文、改文件，看有没有校验和告警。

第四，从反面看还有一条隐含目标：发现潜在的安全漏洞。这条虽然列在"进一步"里，但它是测试工作最直接的产出。

再看进一步的目标，三条。

隐蔽通道分析：找出那些"设计者没打算用来通信、但可以被用来传信息"的路径（比如通过响应时间、通过错误码差异）。这一条属于高等级系统的要求。

可信路径：验证用户与安全功能之间的通信路径不被中间人插入或篡改——简单说，你看到的登录框确实是系统的登录框，不是别人伪造的。

加密算法、代码规范：验证用的是不是合规的算法（不能用 MD5、DES 这类已破的），以及代码有没有遵守安全编码规范。

别忘了课件标的那句：**所有目标最终都归结到数据的机密性、完整性、可用性，覆盖存储、传输、处理**。这句话是"总纲"——你设计任何一条安全用例，都可以问：它验证的是哪种数据状态下的哪个属性？

板书：两层目标。通常：① （强制）访问控制 ② 身份鉴别 ③ 完整性保护 ④ 发现潜在漏洞。进一步：隐蔽通道分析｜可信路径｜加密算法与代码规范。下方写总纲："CIA × 三种数据状态（存储・传输・处理）"。

提问："可信路径"这个目标，具体在防什么？

预设回答：

- 答"防中间人"——对，防的是"你和系统之间被插了一层"。

- 答"防钓鱼页面"——也很贴切：你以为是系统的登录框，其实是攻击者伪造的。

- 答"防会话劫持"——相关，可信路径保证通道不被插入或篡改。

- 追问收口：所以测可信路径，要看证书、看域名、看有没有强制跳转 HTTPS、看登录页有没有被套框架（回想 Frame-proxy）。

易错点 / 考点：① 两层目标（通常四条、进一步三条）的划分要清楚；② **隐蔽通道分析**这个概念要能解释；③ 高频判断："安全性测试的目标就是找漏洞"——不全面，还包括验证访问控制、鉴别、完整性等"功能是否

有效"；④ 记住总纲句"CIA × 存储/传输/处理"，答题时很好用。

过渡：目标定了，范围也要划清——安全性测试到底分几块。

【7.64】软件安全性测试的范围（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页把安全性测试的范围切成两块：**安全功能测试**和**安全漏洞测试**。这个二分法非常重要，因为**很多人只做了后一半，就以为做完了安全测试**。

第一块，安全功能测试（Security Functional Testing）：验证系统**宣称的安全功能真的有效**。课件列了一串：数据机密性、完整性、可用性、不可否认性、身份认证、授权、访问控制、审计跟踪、委托、隐私保护、安全管理等。**注意它的性质——它是"正向验证"**：需求说"密码要加盐哈希"，我去验证是不是真加了；需求说"只有管理员能删除用户"，我去验证普通用户删不了。**它的判据是"需求是否被实现"。**

第二块，安全漏洞测试（Security Vulnerability Testing）：**从攻击者的角度，以发现软件的安全漏洞为目的**。课件给的定义很清楚：安全漏洞是系统在**设计、实现、操作、管理**上存在的**可被利用**的缺陷或弱点。

这两块的区别，我给它三组对照。**视角不同**：功能测试站在"设计者"这一侧，验证"该有的有没有"；漏洞测试站在"攻击者"这一侧，找"不该有的有没有"。**依据不同**：前者依据需求文档，后者依据攻击模式和漏洞库。**完备性不同**：功能测试可以"测完"，因为需求条目是有限的；漏洞测试**永远测不完**，因为攻击手法在演进。

这个"测不完"的性质带来一个实践结论：**漏洞测试必须有优先级**。按什么排？按"资产价值 × 攻击面暴露程度 × 利用难度"。这就是 7.44 威胁模型的用处。

还有一个常见误区要澄清：**安全功能测试不能替代漏洞测试**。一个系统可能把所有安全功能都实现了（有认证、有加密、有审计），但仍然存在越权漏洞——因为**功能是"点"，漏洞在"缝"里**。反过来也一样。**两块必须都做。**

板书：左右两栏。左：**安全功能测试**（机密性・完整性・可用性・不可否认・认证・授权・访问控制・审计・委托・隐私・安全管理）——正向验证，依据需求，可测完。右：**安全漏洞测试**（从攻击者角度，找设计/实现/操作/管理上的可被利用缺陷）——反向寻找，依据攻击模式，测不完。底部："功能管'点'，漏洞在'缝'里，两块都要做"。

提问：一个系统通过了全部安全功能测试（认证、加密、审计都有），能不能说它"没有安全漏洞"？

预设回答：

- 答"能，功能都实现了"——不能。功能实现和"能不能被绕过"是两件事。

- 答"不能，还要做漏洞测试"——对。比如有认证，但认证可以被绕过；有加密，但密钥硬编码在代码里。

- 答"要看有没有做越权和注入测试"——很具体，这两类是"功能齐全但漏洞仍在"的典型。

- 追问收口：所以安全测试的报告要分两部分写——**功能有效性结论**和**漏洞清单**，不能混成一句"安全良好"。

易错点 / 考点：① 两块的名称、视角、依据要能对照说清，简答高频；② 高频错答："做了渗透测试就不用做安全功能测试"——渗透测不到"需求要求的功能是否存在"；③ **漏洞测试永远测不完**这个性质，以及它推出"要排优先级"；④ 记住"功能管点、漏洞在缝"这句口诀。

过渡：说到"功能测试"和"安全测试"，还有一组更基础的区别要讲清。

【7.65】功能性测试 vs. 安全性测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给了一句非常精炼的对照，值得逐字读。

功能性测试：**软件做它应该做的事**，验证正确的输出，发现不正确的输出／行为，产出缺陷（Bug）。

安全性测试：**软件不做它不应该做的事**，应用输入的验证，以避免不安全的事情发生；在测试软件系统中对**危险防止和危险处理设施**进行的测试，以验证其是否有效。

这两句的核心差别在一个词上——**"应该"和"不应该"**。功能测试问"该做的做了吗、做对了吗"；安全测试问"**不该做的，拦住了吗**"。

举个特别直观的例子。转账接口：功能测试关心"转 100 元，扣 100 元，对方到账 100 元"；安全测试关心"**转 -100 元，会不会变成给我加 100 元**""**转别人的账户，会不会成功**""**把请求重放十次，会不会扣十次**"。前者是"按说明书用"，后者是"**反着用、乱着用**"。

再看课件那两个词：**"危险防止"和"危险处理"**。这就是安全测试的两个落点——**防止危险发生**（预防：校验、鉴权、限流）和**危险发生后的处理**（恢复：降级、隔离、告警）。这正好对应 7.61 讲的"预防"和"恢复"两类

机制。**所以安全测试不只测"挡不挡得住"，还要测"挡不住的时候怎么办"。**

还有一个容易忽略的差别：**缺陷性质不同**。功能缺陷往往是"看得见的错"——用户会发现、会报障；安全缺陷往往是"**看不见的对**"——功能全对，但背后已经能被利用了。这决定了安全缺陷更依赖**主动的、专业的**测试，而不是等用户反馈。

一句话总结这一页：**功能测试证明"它能干活"，安全测试证明"它不会乱干"。**

> **板书：**两行对照。功能：做**应该**做的事 → 验证输出是否正确 → 发现不正确的输出/行为 → 产出 Bug。安全：不做**不应该**做的事 → 输入验证 → 验证危险防止与危险处理设施是否有效。中间画竖线写"**应该 / 不应该**"。底部："功能证明它能干活，安全证明它不会乱干"。

提问：同一个"提交订单"接口，功能测试和安全测试各会设计什么样的用例？各举一条。

预设回答：

- 答"功能：正常下单；安全：下单金额改成 0.01 元"——非常典型的对照，一个验规则、一个验绕过。

- 答"功能：库存不足要提示；安全：并发下单会不会超卖"——也很好，安全用例常常是"并发＋边界"组合。

- 答"安全还要测别人的订单能不能看"——对，那是越权，功能用例里通常不会写。

- 追问收口：所以设计安全用例有一个通用手法——**拿功能用例，改一个"不该改"的参数，或者换一个"不该有"的身份**。

易错点 / 考点：① 两句话的原文（"做应该做的事"／"不做不应该做的事"）几乎必考；② **危险防止 vs 危险处理**这两个词对应预防与恢复，要会联系；③ 高频判断："安全测试就是功能测试加一些边界值"——错，视角和依据都不同；④ 记住通用手法："改一个不该改的参数、换一个不该有的身份"。

过渡：范围清楚了，具体任务是什么？课件给了四条。

【7.66】安全性测试的任务（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页列了安全性测试的四项任务，我逐条讲，并且给每条配"怎么落地"。

第一，全面检验软件在需求规格说明中规定的防止危险状态措施的有效性和在每一个危险状态下的反应。这

句话有点绕，拆开看：前半句是"**措施有没有效**"——需求里写了"连续输错 5 次锁定账号"，那我要验证第 6 次真的锁了；后半句是"**危险状态下的反应**"——真到了危险状态（被锁、被限流、依赖宕机），系统反应是否得体：是明确提示，还是白屏、还是暴露内部信息？

第二，对软件设计中用于提高安全性的结构、算法、容错、冗余、中断处理等方案，进行针对性测试。 这一条针对的是"**设计层面的安全机制**"：加密算法用得对不对、冗余切换能不能真的切过去、中断处理会不会留下不一致状态。它的特点是要**专门设计用例**去打这些机制，不能靠随机点。

第三，在异常条件下测试软件，以表明不会因可能的单个或多个输入错误而导致不安全状态。 注意"**单个或多个**"——单个错误输入（一个单引号）和多个错误叠加（超长＋特殊字符＋并发）都要测。这一条是模糊测试和健壮性测试的依据。

第四，对安全性关键的软件单元、组件，单独进行加强的测试，以确认其满足安全性需求。 也就是"**重点部位重点测**"：登录模块、权限模块、支付模块、加解密模块，这些要单独拎出来做更严格的测试——代码评审＋单元测试＋专项攻击测试。**这和"关键模块要加强测试"的一般测试原则是一致的，只是安全场景下"关键"的定义变成了"一旦被攻破后果最重"。**

四条合起来看，其实是四个层次：**需求层的措施验证 → 设计层的机制验证 → 异常条件下的健壮性 → 关键组件的强化测试**。你写安全测试计划时，按这四层列活动，就不会漏。

板书：四条任务：① 需求规定的危险防止措施有效性＋危险状态下的反应 ② 针对安全设计（结构・算法・容错・冗余・中断处理）的专项测试 ③ 异常条件（单个/多个输入错误）下不进入不安全状态 ④ 安全性关键单元/组件的加强测试。底部："需求层 → 设计层 → 异常健壮性 → 关键组件"。

提问："在异常条件下测试"和"模糊测试"是什么关系？

预设回答：

- 答"一回事"——不完全。异常条件测试是**目标**，模糊测试是达成它的一种**手段**（还有手工构造异常、故障注入等）。

- 答"模糊测试是自动化的异常测试"——基本准确，靠随机/变异输入去触发异常。

- 答"异常条件还包括环境异常"——很专业：依赖宕机、网络中断、磁盘满、时钟跳变，这些也算异常条件。

- 追问收口：所以答"怎么在异常条件下测试"时，要写清手段：**手工构造、模糊测试、故障注入**。

易错点 / 考点：① 四条任务的表述要能复述，简答高频；② 第一条的"两半"（措施有效性＋危险状态反应）

常被答漏后半；③ 高频概念：安全性关键组件要**单独加强测试**；④ 加分：把四条归成"需求层—设计层—异常—关键组件"四层。

过渡：不同系统的安全要求不一样，这就引出"安全等级"。

【7.67】安全等级（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲安全等级，五个级别：**用户自主保护级、系统审计保护级、安全标记保护级、结构化保护级、访问验证保护级**。这是我国计算机信息系统安全保护等级的划分，从第一级到第五级，**要求逐级提高**。

我按"每级多了什么能力"来讲，这样你不用死记，能推出来。

第一级，用户自主保护级：靠**用户自主**的访问控制——文件主自己决定给谁权限（就是 DAC）。系统本身不做强约束。这一级适用于"被破坏后对公民、法人权益有损害，但不损害国家安全"的一般系统。

第二级，系统审计保护级：在自主访问控制之上，加了**审计**——谁做了什么要记下来，可以追查。**关键词是"审计"**，这一级的名字就告诉你了。

第三级，安全标记保护级：引入**安全标记**——给主体和客体打上密级标签，访问控制按标签来。这就从"自主"走向"强制"（MAC），因为标签不是用户能随便改的。

第四级，结构化保护级：要求系统的安全机制**结构化**——安全功能要独立、要有清晰的结构、要能形式化地说明。它面向的是"被破坏后会对社会秩序、公共利益造成严重损害"的系统。

第五级，访问验证保护级：最高级。要求对访问**进行验证**，安全机制要能被数学证明（形式化验证），并且能抗高强度的攻击。用于关系国家安全的关键系统。

记这条链的抓手：**自主 → 审计 → 标记 → 结构化 → 验证**。五个词对应五级，顺序不能乱。

为什么测试人员要记这个？因为**等级决定了测试的深度和验收标准**。第二级系统的安全测试和第二级的要求对齐就行，你要是拿第五级的标准去要求它，成本不可承受；反过来，一个该评第三级的系统你按第一级测，就是失职。

板书：五级阶梯（自下而上）：① 用户自主保护级（DAC）② 系统审计保护级（＋审计）③ 安全标记保护级（＋安全标记，走向 MAC）④ 结构化保护级（安全机制结构化）⑤ 访问验证保护级（形式化验证・抗高

强度攻击）。右侧写口诀："自主→审计→标记→结构化→验证"。

提问：第二级和第三级的关键差别是什么？

预设回答：

- 答"第三级更严格"——太笼统，要说出"多了什么"。

- 答"第三级加了安全标记，访问控制从自主变成强制"——准确。标记不是用户能改的，所以控制权从用户手里收到系统手里。

- 答"第二级加了审计"——那是第二级相对第一级的差别。

- 追问收口：所以记这条链的方法是**记"每级新增的能力"**，而不是记名字。

易错点 / 考点：① 五级名称与顺序必须默写，选择题高频；② **DAC 到 MAC 的转变发生在第三级**（安全标记），这是理解性考点；③ 高频错答："等级越高功能越多"——准确说是**安全机制越强、保证越严**；④ 记住"等级决定测试深度与验收标准"，这是它和测试工作的连接点。

过渡：五级各自的安全要求，课件用图来展开。

【7.68】不同等级的安全要求（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把五个等级对应的安全要求摆出来对照。我不逐项念表格，我讲**怎么读这张表**，因为这类"等级—要求"矩阵在标准文档里到处都是，会读比会背有用。

读法三步。

第一步，看行是什么。行是安全要求的类别——一般包括自主访问控制、强制访问控制、身份鉴别、客体重用、审计、数据完整性、隐蔽通道分析、可信路径、可信恢复这些项。**这些名字你在前面都见过：**访问控制、鉴别、审计是 7.63 的目标；隐蔽通道、可信路径也在那里；可信恢复对应 7.61 的"恢复"机制。

第二步，看列是什么。列是五个等级。**要求是"逐级增加"的：**低等级只有基本项（自主访问控制、身份鉴别、审计），高等级才追加强制访问控制、隐蔽通道分析、可信路径、形式化验证这些"贵"的能力。

第三步，看交叉格里写什么。通常写"要求／不要求"或者"部分／全部"。**你只要抓住上升趋势就够了：**等级

越高，要求的项越多、强度越高。

这张表最大的教学价值是让你建立一种感觉：**安全能力是分层的，不是全有全无**。你给一个系统做安全测试，第一步应该是"**它属于哪一级、对应哪些要求**"，然后逐项验证。这比"凭经验想到什么测什么"专业得多。

再补一句实践提醒：**标准里的等级是"应达到"的底线，不是"最高标准"**。业务上如果需要更高（比如涉及支付），那就得在等级要求之上再加。**合规是地板，不是天花板。**

> **板书：**画一个矩阵示意：行＝安全要求类别（自主访问控制・强制访问控制・身份鉴别・客体重用・审计・数据完整性・隐蔽通道分析・可信路径・可信恢复），列＝五个等级，格子里标"要求/不要求"，并画一条向右上的箭头表示"逐级增强"。底部："合规是地板，不是天花板"。

提问：读这类"等级—要求"表格，你会先看哪一部分？

预设回答：

- 答"先看要背的格子"——那样会陷进细节。

- 答"先看行和列分别是什么"——对，先建坐标系。

- 答"先找逐级增加的规律"——更好：抓住趋势，细节用到时再查。

- 追问收口：标准文档的读法是**先结构、再趋势、后细节**，别一上来背格子。

易错点 / 考点：① 常见安全要求类别的名称要认得（尤其**客体重用、隐蔽通道、可信路径、可信恢复**这四个偏门的）；② 高频判断："高等级要求包含低等级要求"——一般成立，但要理解它是"追加"关系；③ 理解"合规是底线"这个观点；④ 本页考得较浅，重点是把类别名词认全。

过渡：不同等级下，安全功能测试的内容也不同。课件用图给出对照。

【7.69】不同安全级别的安全功能测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把"等级"和"安全功能测试的具体内容"接起来。它的意思很直接：**同样是做安全功能测试，等级不同，测的深度和项数不同。**

我把这个对应关系讲成三层。

第一层，基础项——所有等级都要测：身份鉴别（登录、口令策略、会话）、访问控制（权限分配与校验）、审计（有没有记录、记录是否完整）。这三项是任何系统的安全底线，**也是最容易出错的三项**——现实中的漏洞大多数落在这里。

第二层，增强项——中高等级追加：细粒度的访问控制（数据级、字段级）、强制访问控制的标签管理、客体重用（资源释放后不能被下一个使用者读到残留数据——**这一项特别容易被忽略，比如内存、临时文件、连接池里的残留**）、更严格的审计（防篡改、集中存储）。

第三层，高级项——高等级才要求：隐蔽通道分析、可信路径验证、可信恢复（系统被破坏后能否恢复到一个可信状态）、形式化或半形式化的验证。

这三层的读法，其实是在回答一个问题：**"我这个系统，安全功能测试要测到什么程度？"** 答案不是拍脑袋，是**由等级和业务风险共同决定**。

再讲一个实操建议：**不管哪一级，测试用例的组织方式是一样的**——按"鉴别 → 授权 → 访问控制 → 审计 → 数据保护"的顺序，每一项先测正向（该通过的通过），再测反向（不该通过的拦住）。**正向用例保证功能在，反向用例保证边界守得住。** 只做正向，等于没测安全。

最后提醒：**等级越高，越要文档化**。测试方案、用例、结果、残余风险，都要能提交评审——这也是高等级系统测试工作和低等级最大的差别之一。

板书：三层。基础（所有等级）：身份鉴别・访问控制・审计。增强（中高等级）：细粒度访问控制・强制访问控制标签・**客体重用**・审计防篡改。高级（高等级）：隐蔽通道分析・可信路径・可信恢复・形式化验证。底部："用例组织：鉴别→授权→访问控制→审计→数据保护；每项都测正向＋反向"。

提问："客体重用"这一项在测什么？为什么它容易被忽略？

提问：一个删除订单的功能，测"客体重用"要关注什么？

预设回答：

- 答"删了就是删了，没什么可测"——这正是它被忽略的原因：**物理上删了，逻辑上可能还在**。

- 答"要看缓存和备份里还有没有"——对，缓存、备份、日志、临时文件都可能残留。

- 答"还要看连接池/内存复用"——很专业：同一块内存被下一个请求复用，如果不清理就可能读到别人的数据。

- 追问收口：所以客体重用的测试方法是——**A 用完 → 释放 → B 申请 → 检查 B 能不能看到 A 的痕迹**。

易错点 / 考点： ① 三层的划分（基础／增强／高级）便于记忆；② **客体重用**这个概念必须能解释并会设计测试方法；③ 高频判断："安全功能测试只测正向功能"——必须正反都测；④ 记住"等级越高越要文档化"这条实践差别。

过渡： 讲完"测什么"，该讲"怎么测"了——安全性测试方法。

【7.70】安全性测试方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把安全测试方法摊成一张二维图，很值得花时间讲清，因为**它是一张选方法的决策表**。

先看第一维：**静态测试方法**和**动态测试方法**。

静态包括三项：**工具扫描／分析代码**（SAST，不运行程序，找代码里的坏味道和已知模式）；**人工评审：架构／代码**（人读代码，找逻辑漏洞——这是工具替代不了的，因为工具不懂业务）；**专家安全风险评估**（靠经验判断"这个设计会被怎么打"）。

动态包括三项：**功能测试方法**（按需求验证安全功能）；**模糊测试方法**（喂随机/变异数据看会不会崩）；**渗透测试方法**（像攻击者一样打一轮）。

再看第二维，也是这一页最有价值的部分：**基于威胁的方法**和**基于漏洞的方法**。

基于威胁：**从软件外部考察其安全性，识别软件面临的安全威胁并测试其是否能够发生**。它的出发点是"**谁会怎么攻击我**"，对应 7.44 的威胁模型和 7.49 的 STRIED。它关心的是"**场景**"。

基于漏洞：**从软件内部考虑其安全性，识别软件的安全漏洞，如借助特定的漏洞扫描工具**。它的出发点是"**我身上有哪些已知弱点**"，对应 7.50 的漏洞模式和 OWASP 清单。它关心的是"**清单**"。

两者的关系：**威胁方法找"新问题"，漏洞方法找"老问题"**。老问题（注入、越权、弱口令）占实际漏洞的大多数，所以漏洞方法投入产出比高；新问题（这个业务特有的逻辑漏洞）只有威胁方法能发现。**两个都要做，先做漏洞（快、便宜），再做威胁（深、要设计）。**

静态与动态怎么选？一句话：**静态看"写的时候就有问题"，动态看"跑起来才出问题"**。越权的逻辑错误常常静态能看出来（少了一行校验），而并发漏洞只有动态才现形。

板书： 二维表。行＝静态（工具扫描/分析代码・人工评审架构/代码・专家风险评估）｜动态（功能测试・

模糊测试・渗透测试）。列＝基于威胁（外部视角：识别威胁是否会发生）｜基于漏洞（内部视角：识别已知漏洞）。底部："威胁找新问题，漏洞找老问题；静态看'写错'，动态看'跑错'"。

提问：一个新上线的业务系统，时间只够做一轮安全测试，你会先选哪种方法？

预设回答：

- 答"渗透测试，最像真实攻击"——效果直观，但周期长、覆盖有限。

- 答"漏洞扫描＋人工评审"——务实：扫描能快速清掉"老问题"，评审能看清业务逻辑。

- 答"先做基于漏洞的，再做基于威胁的"——最符合投入产出：老问题占比最大。

- 追问收口：所以顺序建议是**扫描（快）→ 人工评审（准）→ 针对高价值模块做威胁分析／渗透（深）**。

易错点 / 考点：① 静态三项、动态三项要能列全；② **基于威胁 vs 基于漏洞**的视角差别是本节核心，简答高频；③ 高频错答："渗透测试能覆盖所有安全问题"——覆盖不了，且不可重复；④ 记住"静态看写错、动态看跑错"，一句顶十句。

过渡：六种方法里，"功能测试方法"最容易被小看，我们看一个具体例子——口令。

【7.71】安全性功能测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲安全性功能测试的具体做法，落脚点是课件最后那三个字：**示例：口令**。它把前面讲过的方法用口令这一个点串起来。

先说功能测试在安全里做什么。课件列了几类：**身份认证测试**（如 DCE／Kerberos、基于公共密钥等认证机制）、**访问控制策略验证**（如入口访问、权限、目录等控制），再加上**规则验证、负面测试、传输检查、存储检查、控件兼容性**这几类通用动作。

这几个词要解释一下。**规则验证**：验证"策略说的是不是真的在执行"——策略说密码 90 天过期，那就验证第 91 天真的要求改。**负面测试**：专门测"不该成功的是否失败"——错误密码、越权请求、过期 token。**传输检查**：抓包看敏感数据在链路上是不是明文。**存储检查**：直接看数据库和文件，密码是不是哈希、关键字段是不是加密。**控件兼容性**：安全控件（验证码、加密控件、证书）在不同浏览器和版本上是否正常工作——**这一条特别容易被忘：控件不兼容，用户就登录不了，可用性直接崩。**

现在落到口令。**口令的安全功能测试清单**：强度策略是否生效；弱口令是否被拒；口令是否加盐哈希存储（存储检查）；传输是否加密（传输检查）；连续失败是否锁定（负面测试＋规则验证）；锁定后是否可解锁、多久解锁（规则验证）；修改口令是否验证旧口令；重置流程是否安全；会话在改密后是否失效——**最后这条最容易漏，改完密码旧会话还有效，等于没改**。

你看，一个"口令"能出十几条用例。**这就是安全功能测试的样子：不是找一个惊天大漏洞，而是把每一条规则逐项验证到底。** 它枯燥，但它挡住的是最常见的那批问题。

一句提醒：这类测试**最好做成检查单**。因为条目多、易漏，靠记忆一定会漏；写成清单，每做一次打个勾，还能沉淀成团队资产。

板书：通用动作：身份认证测试｜访问控制策略验证｜规则验证｜负面测试｜传输检查｜存储检查｜控件兼容性。口令清单：强度策略・弱口令拒绝・加盐哈希・传输加密・失败锁定・解锁规则・改密验旧・重置安全・**改密后会话失效**。底部："把规则逐项验证到底，做成检查单"。

提问：为什么"改密后旧会话要失效"是一条必要的安全用例？

预设回答：

- 答"因为用户体验"——不是体验问题，是安全问题。

- 答"因为攻击者可能已经拿到了旧会话"——正是。用户改密码的动机往往就是"怀疑被盗"，如果旧会话还能用，改密等于没改。

- 答"要同时踢掉其他设备"——更完整：这就是"改密即全端下线"的设计。

- 追问收口：所以这条用例的本质是——**凭据变更必须使旧凭据立即失效**。

易错点 / 考点：① 六类通用动作（规则验证、负面测试、传输检查、存储检查、控件兼容性等）要能列举；② **负面测试**是安全功能测试的核心手法，必答；③ 高频漏项：控件兼容性、改密后会话失效；④ 实践建议"做成检查单"可作为简答的落地措施。

过渡：功能测试是"按规则验"，那"不按规则"怎么验？下一类方法专干这个——模糊测试。

【7.72】模糊测试方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲模糊测试（Fuzz Testing）。课件的定义是：**在一个被测试程序中附加上随机数据（fuzz）作为程序的输入。如果被测试程序出现问题（例如 Crash，或者异常退出），就可以定位程序的缺陷。**

这个定义很短，但它包含了一个非常聪明的思想：**不去猜"哪种输入会出问题"，而是大量地乱喂，让程序自己暴露。**

课件还给了一句评价：**模糊测试的巨大优势：测试设计极其简单，系统的行为先入为主。** 这句话怎么理解？"测试设计极其简单"——你不需要写几十条精心设计的用例，只需要一个能生成输入的生成器和一台会跑的机器；"系统的行为先入为主"——意思是**你不需要预先知道"正确的输出是什么"**，你只看"程序有没有异常"。**这一点很关键：功能测试必须知道预期结果，模糊测试不需要。** 这使它能测那些"说不清正确行为"的输入空间。

模糊测试擅长找什么？**崩溃、内存错误、解析异常、未处理的边界**——也就是"程序被搞挂了"这一类问题。它不擅长找"逻辑越权""业务规则绕过"，因为那些不会让程序崩。

它也有代价：**大量无效输入、覆盖率不一定高、结果要去重**。所以现代模糊测试都配了"覆盖率引导"——只有能触发新代码路径的输入才被保留（这正是下一页要讲的 AFL 的核心思想）。

用在哪？**所有解析外部输入的地方**：文件格式解析（图片、音视频、文档）、网络协议解析、序列化反序列化、命令行参数、API 参数。**凡是"别人给你一段数据、你要读懂它"的代码，都是模糊测试的猎场。**

一句话总结：**模糊测试用"数量"换"发现"，用"机器"换"人的想象力"。**

> **板书：**定义三要素：喂随机数据（fuzz）→ 观察是否崩溃/异常退出 → 定位缺陷。优势："测试设计极简"＋"不需要预知正确行为"。擅长：崩溃・内存错误・解析异常・未处理边界。不擅长：逻辑越权・业务绕过。猎场：文件格式・协议・序列化・API 参数。底部："用数量换发现，用机器换想象力"。

提问：模糊测试不需要知道"正确的输出"，那它凭什么判断"发现了缺陷"？

预设回答：

- 答"靠程序崩溃"——对，这是最基本的判据。

- 答"靠异常退出、超时、内存错误"——补充得好，还有一些不崩但异常的表现。

- 答"还可以靠断言和差异对比"——很专业：给程序加内部断言，或者用不同实现做差分测试。

- 追问收口：所以模糊测试的判据是"**异常**"，而不是"结果不对"；这也决定了它找不到逻辑漏洞。

易错点 / 考点： ① 模糊测试的定义与"测试设计简单"这个优势要能解释清楚；② 高频判断："模糊测试能发现所有类型的漏洞"——错，它擅长崩溃类，不擅长逻辑类；③ 记住它**不需要预期结果**这个特征；④ 加分：说出"覆盖率引导"是它的演进方向（承接下一页 AFL）。

过渡： 模糊器有好几种造法，课件给了三种。

【7.73】几种不同模糊器的构造（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲三种模糊器的构造思路：**黑盒随机模糊、基于语法的模糊、白盒模糊**。这三种代表了"从笨到聪明"的演进，理解这条演进线，比记定义有用。

第一种，黑盒随机模糊：对**正确格式的输入数据**进行随机变异，然后用这些变异的输入运行程序，看是否能够触发异常。关键点在"**从正确格式的输入出发**"——不是从全随机的字节开始，而是拿一个"能跑通"的样本，改几个字节。这样命中率比纯随机高得多。**它的优点是实现极简，缺点是不懂格式，很多变异在解析的第一关就被拒绝，浪费大量算力。**

第二种，基于语法的模糊：是模糊复杂格式输入的**替代方法**，需要指定**输入格式的输入语法、哪些输入部分要进行模糊化**、以及**如何模糊化**。说白了，你先把格式的语法（比如 JSON 的语法、某个协议的报文结构）写出来，然后按语法**生成合法但语义异常的输入**——比如"字段类型对但值超长""必填字段缺失""嵌套深到爆栈"。**它比黑盒聪明，因为它知道"哪一层才是有效的攻击面"，不会在语法层就被挡掉。**

第三种，白盒模糊处理，课件说它**由微软研究院于 2008 年首创**。方法是：**动态地执行被测程序，从执行过程中遇到的条件分支收集输入约束；然后系统地逐个否定所有这些约束，并使用约束求解器求解，其解被映射到执行不同程序执行路径的新输入；使用系统搜索技术重复这个过程，试图扫描程序的所有可行的执行路径。**

这段话信息量很大，我翻译一下：**程序跑到一个 if，就把"要走进这个分支，输入必须满足什么条件"记下来（这叫收集路径约束）；然后把这个条件反过来（否定约束），让求解器算出"什么样的输入能走进另一个分支"；再用算出来的输入去跑，就能到达新路径。反复做，尽量把所有路径走一遍。** 这就是符号执行／白盒 fuzzing 的思路。

三种的取舍：**黑盒最便宜、覆盖最浅；语法模糊中等，适合结构化格式；白盒最强，但需要求解器、开销大、对复杂程序会"路径爆炸"。** 工程上常见的是**混合**：用覆盖率引导的黑盒打底，对关键格式补语法模型。

> **板书：**三行。黑盒随机：从正确样本出发随机变异 → 简单·命中率低。基于语法：先写输入语法 → 按语法生成合法但异常 → 突破语法关。白盒模糊（微软研究院 2008）：动态执行收集路径约束 → 逐个否定约束 → 约束求解 → 生成新输入 → 系统搜索覆盖路径。底部："黑盒浅、语法中、白盒深；工程上混合用"。

提问：为什么"从正确格式的输入出发做变异"，比"完全随机生成输入"更有效？

预设回答：

- 答"因为随机输入大多在解析层就被拒了"——正是。你连业务逻辑都没走到，谈不上发现深层缺陷。

- 答"因为能走得更深"——对，从合法样本出发，变异后仍有机会通过前面的校验。

- 答"因为覆盖率更高"——本质就是这个：**有效输入才能推进到更深的分支**。

- 追问收口：所以模糊测试的效率，取决于"**输入生成器有多懂格式**"——这就自然过渡到基于语法的模糊。

易错点 / 考点：① 三种模糊器的名称与核心思路要能对上；② **白盒模糊的四步**（收集约束→否定约束→求解→搜索）是简答高频；③ 高频错答："白盒模糊就是看源代码"——它是**动态执行＋约束求解**；④ 记住微软研究院 2008 这个出处（课件给的），别记错年份。

过渡：白盒模糊的思想有一个著名的工程实现——AFL。

【7.74】American fuzzy lop 方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲 AFL（American fuzzy lop），课件称它是**一款开源的模糊测试工具，是当今使用最广泛的 Fuzzer**。它值得单独一页，因为它的算法思想非常漂亮，而且**只用六步就说清了**。

先说它的两个前提。**第一，它在程序执行前对程序源码进行插桩（instrumentation）**，以便在程序执行过程中**实时获取程序的执行情况**。插桩就是往代码里插入探针，程序跑到哪个分支、走了哪条边，都被记录下来。这一步是它"聪明"的基础——**它知道自己的输入有没有带来新覆盖。第二，它采用遗传算法对程序的输入进行变异**——把"输入"当成基因，把"新覆盖"当成适应度，一代代进化。

然后是课件的六步算法，我逐步解释。

第一步，将用户提供的初始测试用例加载到队列中。 给它几个能跑通的样本当"种子"。

第二步，从队列中获取下一个输入文件。 轮流处理队列里的每个种子。

第三步，试图将测试用例修剪到不改变程序测量行为的最小尺寸。 这一步很妙：把输入尽量缩短，只要程序行为不变就行。**输入越短，后续变异越容易命中关键字节。**

第四步，使用均衡的、经过充分研究的各种传统模糊策略，重复地对文件进行变异。 也就是对它做各种"改字节、翻位、插入、删除"的操作，一轮又一轮。

第五步，如果任何产生的变异导致插桩记录新的状态转换，将变异的输出作为一个新的条目添加到队列中。 这就是"遗传"的核心：**只有带来新覆盖的变异才被保留下来，成为新的种子。** 有反馈，所以叫"覆盖率引导"。

第六步，转到第 2 步。 循环往复。

这六步的精髓是：**用"覆盖率"当路标，让变异朝着"没走过的路"走**。它把盲目的随机，变成了有方向的搜索。这也解释了它为什么能成为最广泛使用的 fuzzer——**效果好、成本低、几乎不需要人设计用例**。

实践上有两点要注意：**一是初始种子要"好"**（能跑通、覆盖主要功能），种子差则效果差；**二是要有足够的运行时间**——AFL 靠的是成千上万次执行，跑十分钟和跑十天差别巨大。

> **板书：**两个前提：源码插桩（实时获取执行情况）＋遗传算法变异。六步：① 加载初始用例到队列 ② 取下一个输入 ③ 修剪到最小尺寸（不改变行为）④ 用多种策略反复变异 ⑤ 若产生**新状态转换**则加入队列 ⑥ 回到第 2 步。底部："覆盖率当路标，把随机变成有方向的搜索"。

提问：AFL 的第五步"只有新的状态转换才入队"，如果去掉这一步会怎样？

预设回答：

- 答"也能跑，只是慢"——会慢得多，而且会把队列塞满重复的、无价值的输入。

- 答"就退化成普通随机模糊"——本质如此：**没有反馈，就没有引导**。

- 答"浪费算力在同一个分支上"——对，这正是"覆盖率引导"要解决的问题。

- 追问收口：所以 AFL 的核心不是"变异"，是"**筛选**"——用覆盖率做适应度函数。

易错点 / 考点：① 六步流程要能按顺序说出，尤其**第三步修剪**和**第五步入队条件**；② 两个关键词：**插桩（instrumentation）**、**遗传算法／覆盖率引导**；③ 高频判断："AFL 是纯随机测试"——错，它是有反馈引导的；④ 课件给了 GitHub 与文档链接，属于自学入口，记住名字即可。

过渡：模糊测试是"喂坏数据找崩溃"，还有一类方法专门"改代码看测试够不够"——变异测试。

【7.75】变异测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲**变异测试（mutation testing）**。注意名字里也有"变异"，但它和模糊测试的"变异"**完全不是一回事**，这个区别是本页第一个要讲清的点。

课件的定义是：**变异测试是一种在细节方面改进程序源代码的软件测试方法。变异操作是模拟典型应用错误（定位代码的弱点）、或强制产生有效地测试。**

定义有点绕，我换一种说法。变异测试问的问题不是"程序有没有 bug"，而是——**"我的测试用例，真的能发现 bug 吗？"** 它的做法是：**故意往源代码里注入一个小错误（这叫"变异"），然后跑你的测试用例；如果测试失败了，说明用例抓住了这个错误（这个变异被"杀死"）；如果测试还是全绿，说明你的用例有漏洞（这个变异"存活"了）。**

举个具体例子。代码里原本是 `if (a > b)`，变异成 `if (a >= b)`；测试用例跑完还是全过——那就说明**你的用例里没有一条覆盖到"a == b"这个边界。这就是变异测试的价值：它不测程序，它测你的测试。**

几个术语要记住。**变异体（mutant）**：变异后的程序版本。**杀死（killed）**：测试发现了这个变异。**存活（survived）**：测试没发现。**变异得分（mutation score）**＝被杀死的变异体 ÷ 全部变异体。**得分越高，说明用例的"杀伤力"越强。**

课件说变异操作要"**模拟典型应用错误**"——所以变异不是乱改，而是按固定算子改：把 `>` 改成 `>=`、把 `+` 改成 `-`、把 `true` 改成 `false`、删掉一行、把返回值改掉。这些正好对应程序员最常犯的错。

它的成本很高：**每个变异体都要跑一遍完整测试套件**，变异体数量又可能是几百上千。所以实践中要**抽样**、要**只对改动的代码做**（增量变异）、要**并行跑**。这也解释了为什么它在工业界不如单元测试普及——**好，但贵**。

但它的思想极其重要：**测试的质量，要用"能不能发现人为注入的错误"来衡量，而不是用"用例数量"或"覆盖率"来衡量。** 覆盖率只说明"代码被执行了"，变异得分才说明"错误会被发现"。

> **板书：**核心问题："我的用例能不能发现 bug？"流程：注入变异 → 跑用例 → 失败＝杀死 / 通过＝存活 → 变异得分＝杀死数÷总数。术语：变异体・杀死・存活・变异得分。常用算子：> → >=、+ → −、true →

> false、删行、改返回值。底部："覆盖率说'跑到了'，变异得分说'抓得住'"。

> **提问：**一段代码的行覆盖率是 100%，能不能说明测试很充分？

> **预设回答：**

- 答"能，全跑到了"——覆盖率只说"执行过"，不说"检查过"。代码跑了但没断言，等于没测。

- 答"不能，还要看断言"——对。**覆盖率＋断言强度**才是完整的判断。

- 答"可以用变异测试验证"——最专业的回答：往代码里注入错误，看用例抓不抓得住。

- 追问收口：所以三种度量的关系是——**覆盖率看"有没有跑到"，断言看"跑了有没有检查"，变异得分看"错了能不能发现"**。

> **易错点 / 考点：**① 变异测试的定义与目的（**测测试，不是测程序**）必须说清，这是最大得分点；② 四个术语（变异体、杀死、存活、变异得分）要能解释；③ **与模糊测试的"变异"区分开**：一个变异输入，一个变异代码；④ 高频判断："变异测试用来找程序缺陷"——错，它评估用例的有效性。

> **过渡：**7.1 和 7.2 的主体内容到这里告一段落，后面还有渗透测试、贯穿 SDLC 的安全测试和 Web 安全测试，我们下一片继续。

【7.76】变异测试的方法与流程（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 上一页我们给变异测试下了定义，这一页讲它到底怎么走一遍流程。先把动机说透：一套用例全部通过，只能证明"程序在用例覆盖到的路径上没出错"，不能证明"这套用例有本事发现错误"。变异测试换了个问法——**我故意把程序改错一点点，你的用例抓不抓得住？**
>
> 具体五步。第一步，**造变异体**。对源程序做一处极小的语法改动，得到一个新版本，这个版本就叫 mutant（变异体）。改动不是乱改，而是按"变异算子"来改——把 `>` 改成 `>=`，把 `&&` 改成 `||`，把常量 0 改成 1，把一条语句删掉，把一个返回值取反。为什么是这些算子？因为它们模拟的正是程序员最常犯的错：边界写错、条件写反、少了一个分支。
>
> 第二步，**用原来的测试用例去跑变异体**。注意，用例一条都不许改，也不许为变异体新写用例——那样就作弊了，测的就不是原来那套用例的能力。

第三步，**看结果**。用例失败了，说明它抓住了这个错，叫"变异体被杀死"；用例全部照样通过，说明这个错从用例的网眼里漏过去了，叫"变异体存活"。存活才是我们要的信息——**每存活一个，就说明用例集上有一个洞**。

第四步，**排除等价变异体**。有些改动在语义上跟原程序完全一样，比如把 `i++` 写成 `i = i + 1`，程序行为没变，任何用例都杀不死它。这类叫等价变异体，算分的时候必须剔掉，否则分数永远上不去，测的人也会被冤枉。

第五步，**算变异得分**：杀死的变异体数 ÷（变异体总数 − 等价变异体数）。这个百分比，就是这套用例的"检测力"。

一句话记住：**覆盖率问"你跑到没有"，变异得分问"你测出来没有"**。

> **板书：**写"变异测试五步：造变异体 → 原用例跑 → 判生死 → 剔等价 → 算得分"；右下角写公式 `MS = 杀死数 /（总数 − 等价数）`；旁边列算子：常量／算术／关系／逻辑／语句删除／返回值。

> **提问：**一个变异体存活下来了，是不是一定说明这套用例写得不好？

> **预设回答：**

- 答"是，说明用例漏了"——大方向对，但太急，容易误判。

- 答"不一定，可能是等价变异体"——准确。等价变异体怎么都杀不死，必须先剔掉再评价。

- 答"也可能是那段代码根本没被执行"——也对，这种情况要先看覆盖，覆盖不到的变异体讨论生死没有意义。

- 收口：所以看到存活先问三句：**是不是等价变异？变异点在被测范围内吗？那段代码用例走到了吗？** 三问都不是，才是真盲区。

> **易错点 / 考点：**① 变异得分高 **不等于** 软件质量高，它只说明"这套用例对错误敏感"，软件本身可能是错的；② 等价变异体的判定靠人工，带主观性，考大题时要写"需人工或工具辅助识别"；③ 变异测试是**评价用例**的方法，不是"把变异程序拿去上线"。判断题常考"变异测试属于白盒测试技术"——对。

> **过渡：**变异测试是"我们自己造错、看用例能不能发现"。下面换个角色：不由我们造错，而是由**攻击者**去造错——这就是安全性测试里的渗透测试。

【7.77】渗透测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

进入 7.2 安全性测试。开篇这一页，给的是渗透测试的完整工作链，四个阶段：**信息收集 → 威胁分析 → 渗透攻击 → 风险评估与总结**。

先看它的定义，定义里每个词都有信息量：**采用探索式测试方式，模拟黑客可能使用的攻击技术和漏洞发现技术，对目标系统的安全做深入的探测，发现系统最脆弱的环节，直观地发现问题。**"探索式"说明它不是照着用例清单一条条打勾，而是根据反馈随时调整下一步；"模拟黑客"说明它的视角是敌对的；"最脆弱的环节"说明它的目标不是全覆盖，而是找那个一敲就碎的点；"直观"两个字最关键——它不是给你一份"疑似存在漏洞"的列表，而是把系统真的打穿一次给你看。

四个阶段怎么理解。**信息收集**：利用搜索引擎、利用开源平台、收集产品资料，把目标的域名、IP、技术栈、员工、版本号摸清楚。这一步听起来不技术，但它决定了后面所有攻击的命中率。**威胁分析**：找出价值资产在哪、高风险模块是哪个、这次攻击的目标是什么，手段是攻击树分析加上攻击模式库——攻击树是把"我要拿到数据库"这个目标往下拆成一条条可行路径。**渗透攻击**：漏洞挖掘、漏洞利用、权限提升、持久化，四级递进。**风险评估与总结**：出测试报告，写清风险等级、影响面、修复建议。

我要强调一条边界：**渗透测试是获得授权的对抗性测试**。授权文档要写明目标范围、时间窗口、允许的手法、禁止的动作。为什么这条不能省？因为渗透测试和真实入侵在技术动作上几乎一样，唯一的区别就是有没有那张纸、以及有没有人在旁边盯着。

它跟漏洞扫描的区别也在这里：扫描器给的是"可能有这些漏洞"，渗透测试给的是"我已经从这条路走进来了，证据在这里"。

> **板书：**横排四段"信息收集 → 威胁分析 → 渗透攻击 → 风险评估与总结"；每段下挂关键词（搜索引擎／开源平台／产品资料｜价值资产／高风险模块／攻击树／攻击模式库｜漏洞挖掘／利用／提权／持久化｜测试报告）；右上角写"授权范围＋时间窗口"。

提问：如果一家公司只想花两天做一次安全测试，你会建议它做漏洞扫描还是渗透测试？为什么？

预设回答：

- 答"扫描，快"——两天确实只够扫描，但要讲清扫描的产出是"待验证清单"，误报要人工确认。

- 答"渗透，能证明危害"——如果两天只够打一两个点，那就选最值钱的那个点打穿，比如登录或支付。

- 答"两个都做，先扫后打"——这是实务里最常见的组合：扫描负责铺面，渗透负责打点。

- 收口：时间紧就先铺面再打点；但无论哪一种，**必须有人签字确认范围**。

> **易错点 / 考点：**渗透测试的四个阶段顺序是常考填空。易错点：把"威胁分析"和"风险评估"混为一谈——威胁分析在**攻击之前**，回答"打哪里"；风险评估在**攻击之后**，回答"有多严重、怎么修"。另外"探索式"不等于"随便打"，探索式是一种**基于反馈调整策略**的方法论。

> **【思政落点 5】**：渗透测试这门手艺有个天然的两面性：懂它的人既能找出系统的破口，也能利用这个破口。所以我们讲这一节，重点不在"怎么打进去"，而在"打进去之后要做什么"——出报告、定等级、给修复方案，把风险交回到系统负责人手上。同学们将来无论是做测试、做开发还是做运维，只要手上握着用户的数据，就有义务替他们守住这道门。**能力越强，越要清楚自己站在哪一边**，这就是职业责任。

> **过渡：**这条四段链条在工程实践里被切得更细，我们看业界通用的五阶段划分。

【7.78】渗透测试—概览（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页把渗透测试摊成五个阶段，并且很清楚地分成了两大块：左边叫 **exploration（探索）**，右边叫 **exploitation（利用）**。这个二分比上一页的四段更重要，因为它回答了一个问题——**什么时候从"看"切换到"打"**。

探索这一块有三段。**网络发现**：主机探索、端口扫描、服务识别、指纹识别、网络映射。顺序是有讲究的：先知道有哪些机器活着，再知道开了哪些端口，再知道端口后面跑的是什么服务、什么版本，最后画出整张网络拓扑。**指纹识别**是这里的关键词——同一台 Web 服务器，Apache 2.2 和 Nginx 1.24 的漏洞清单完全不同，指纹错了，后面全错。**漏洞评估**：漏洞研究、漏洞发现、威胁分类。发现之后先分类，因为漏洞的价值不一样，一个能远程执行代码的漏洞和一个信息泄漏的漏洞，处理优先级差着数量级。

利用这一块，三步：**利用开发**——把公开的漏洞信息变成针对这套环境能跑通的攻击代码；**概念性证明**——只做到"证明这件事能成立"就停手，比如读到一条不该读的记录、弹一个无害的提示框，而不是真的把数据拖走；**权限提升**——从普通用户提到管理员，从一个系统跳到另一个系统。

最后是**修复与报告**：威胁消除、未来计划、报告。注意"未来计划"这一项，它指的是把这次的发现转成后续的加固路线，而不是打完就完了。

> 这张图的落点是：**渗透测试是一条有起点有终点的流程，不是一次炫技**。PPT 上还留了一个参考站点，说明这类流程业界是标准化的。

> **板书：**中间画一条竖线，左边写"exploration：网络发现（主机／端口／服务／指纹／映射）＋ 漏洞评估（研究／发现／分类）"；右边写"exploitation：利用开发 → 概念性证明 → 权限提升"；线下写"修复与报告：消除／未来计划／报告"。

> **提问：**为什么业界强调"概念性证明"就停手，而不是把漏洞的危害演示到底？

> **预设回答：**

- 答"怕把系统搞坏"——是一层原因，生产环境真的会被打挂。

- 答"证明能进去就够了，取数据没有必要"——对，PoC 的目的是**证明风险存在**，不是**制造损失**。

- 答"取证要留痕，越界会破坏证据"——很专业，日志和取证链被自己搞乱，后面复盘就说不清。

- 收口：所以 PoC 的原则是**最小必要动作**：能读到一条就够，不必读一万条。

> **易错点 / 考点：**① 五个阶段里"网络发现"和"漏洞评估"都属于探索，不要把它们当成"攻击"；② 指纹识别的作用是**缩小漏洞范围**，不是"识别品牌"；③ 常考简答："渗透测试与漏洞扫描的区别"——扫描是自动化铺面、报告疑似；渗透是人工打点、给出证据链。答题要写"扫描是渗透的一个子环节"，而不是"两者互不相干"。

> **过渡：**知道流程，还要知道具体拿什么手段去打。下一页列的是教科书里最典型的五类手法。

【7.79】渗透测试的一些具体方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页列了五类具体方法，我按"由外到内、由浅到深"的顺序讲，同时每讲一条就配一句"怎么防"——因为你将来的岗位大概率是做测试和开发，不是做攻击。
>
> 第一类，**不同网段、VLAN 之间的渗透**。这条针对的是"内网就安全"的错误假设。很多系统把管理后台放在内网，觉得外面进不来，可一旦边界上有一台机器被拿下，横向移动就开始了。**怎么测**：从一台低权限主机出发，看能不能访问不该访问的网段；**怎么防**：网段隔离、访问控制列表、最小权限。
>
> 第二类，**端口扫描**，用 X-Scan、NMAP 这类网络安全扫描器。它的价值是摸清"哪些门是开着的"。**怎么防**：关掉不必要的端口和服务，别把数据库、缓存、管理端口暴露在公网。

第三类，**远程溢出**。课件给的例子是现成工具组合：溢出工具加监听工具，打 SQL Server 的 1433 端口。它的原理是向服务发送超长或畸形数据，覆盖内存里的返回地址，让程序跳去执行攻击者的代码。**怎么防**：打补丁、开 DEP/ASLR 这类编译与系统缓解、把数据库端口收到内网。这里要提醒一句：**这类"现成工具"在实验环境里是教学素材，在真实网络里就是另一回事**，所以我们的课程设计只在自己的靶机上做。

第四类，**本地溢出**。前提是攻击者已经拿到了一个普通账号，然后用一段特殊指令代码把自己提成管理员，课件举的例子是 Wincsrss。它说明一件事：**拿到普通权限不等于安全**，提权往往只需要一个没打补丁的驱动或一个配置错误的服务。**怎么防**：及时打补丁、服务账户降权、开启审计。

第五类，**脚本及应用测试**。利用脚本层面的弱点，去拿 Web 服务器或数据库目录的访问权限，严重的时候能取得系统控制权。这一类就是后面 Web 安全性测试要展开的内容：上传、注入、包含、反序列化。

五类合起来看，是一条从"看见门"到"推开门"再到"走到里屋"的路径。

> **板书：**五行，每行"手法 → 例子 → 防御"：跨网段（VLAN 隔离）｜端口扫描（X-Scan／NMAP → 关闭多余端口）｜远程溢出（1433 端口 → 补丁＋DEP／ASLR）｜本地溢出（普通账号→管理员，Wincsrss → 降权＋审计）｜脚本应用测试（→ 输入校验＋权限最小化）。

提问：这五类手法里，哪一类最不需要"高深技术"，但造成的损失可能最大？

预设回答：

- 答"端口扫描"——它本身不造成损失，但它是所有后续动作的入口，"暴露面"本身就是风险。

- 答"口令猜测"——对，课件在这一页也把它列进了方法清单，一个弱口令就能让前面所有防护作废。

- 答"脚本及应用测试"——从损失看确实最大，因为 Web 是唯一对公网开放的那一层。

- 收口：**门槛最低、危害最大**的往往是同一件事：暴露面没管住 + 弱口令。这也解释了为什么安全加固的第一步永远是"关掉不需要的东西、改掉默认口令"。

易错点 / 考点：远程溢出与本地溢出的区别是常考点——**远程溢出不需要任何账号**，从网络直接打；**本地溢出必须已有普通账号**。简答题里"防御措施"至少写三条：补丁、最小权限、网络隔离。另外注意课件把"口令猜测"写在方法清单里，答题不要漏。

> **过渡：**手法有了，可这些动作都很"重"，一不小心就会把被测系统打挂。所以实施的时候必须有一套监控与保护策略。

【7.80】渗透测试实施策略（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲的是"怎么打才不出事"，五条策略，本质上都是**风险控制**。

第一条，**全程监控**：用类似 Wireshark 的嗅探软件全程抓包。为什么？一是留证据，每一步发了什么包、收到什么响应，事后可复盘；二是可追责，万一系统出现异常，能立刻判断是不是我们打的；三是能止损，看到不该有的响应就停手。

第二条，**择要监控**：扫描过程不录，只在数据分析完、准备发起渗透之前才开抓包。这是全程监控的轻量版，适合规模大、扫描噪声多的场景。它的取舍很清楚：**用完整性换开销**。

第三条，**主机监控**：只盯受测主机的存活状态。这是最省的一种，适合那些"只要不挂就行"的系统。

第四条，**指定攻击源**：多方监控指定的那台源主机，看它的进程、网络连接、数据传输。这条在**多团队协作**时特别有用——大家同时在一个环境里测，出了问题要能定位是谁的动作。

第五条，也是最实用的一条：**对关键系统，采用对目标的副本进行渗透测试**。把生产环境克隆一份到隔离环境里打，打崩了不影响真实用户。代价是副本可能跟生产有差异，所以结论要标注"在副本环境验证"。

这五条合起来回答一个问题：**渗透测试的风险不只来自漏洞，也来自测试动作本身**。一个负责任的测试者，先想的是"我怎么保证不出事"，再想"我怎么打进去"。

板书：五条竖排：全程监控（Wireshark 抓包）→ 择要监控（渗透前开抓）→ 主机监控（只看存活）→ 指定攻击源（多团队定位）→ 副本渗透（隔离环境）。右侧批一句"先控风险，再谈攻击"。

提问：如果客户只允许你在生产环境测，不允许做副本，你会额外加哪些保护措施？

预设回答：

- 答"选深夜做，业务低峰"——务实，但夜间值班人少，出事响应慢，要配好应急联系人。
- 答"先小流量、单请求验证，确认无害再放大"——很专业，这叫"最小冲击"，和上一页的 PoC 原则一致。
- 答"准备好回滚和停止口令"——关键，必须有一个能一句话叫停的人。
- 收口：生产环境测试的三件套：**时间窗口、单点验证、随时可停**。缺一件就不要开工。

易错点 / 考点：五种策略的名称与适用场景要能对上。易错点是把"择要监控"和"主机监控"搞混：择要监控监

控的是**流量**，只是时间上延后；主机监控监控的是**主机存活**，对象不同。简答题常问"为什么关键系统要用副本"，标准答法：降低对生产的影响、可反复验证、便于对比。

> **过渡：**到这里，渗透测试作为一种"手段"讲完了。但安全性测试远不止渗透，它必须铺到整个开发生命周期里。

【7.81】贯穿 SDLC 的安全性测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是安全性测试的**结构性一页**，它讲的是：安全不是上线前的一道关卡，而是一条从需求铺到运维的线。SDLC 就是软件开发生命周期，把它拆开看，每个阶段都有对应的安全动作。

需求阶段：定安全需求、定合规要求、做滥用案例。什么叫滥用案例？正常的用例写"用户如何完成转账"，滥用案例写"攻击者如何绕开限额转账"。这一步做不好，后面所有测试都没有判据。

设计阶段：威胁建模、体系结构风险分析。这时候要问：数据在哪、谁能碰、边界在哪、信任关系有几层。设计阶段的漏洞最贵——改一张架构图可能推翻半个系统，但不改就是带着结构性问题上线。

编码阶段：安全编码规范、静态代码分析、代码评审。缓冲区溢出、SQL 拼接、硬编码密钥，这些都是编码阶段就能拦下的。

测试阶段：安全功能测试、模糊测试、渗透测试，也就是我们这一节讲的主体。

运维阶段：补丁管理、日志监控、应急响应、漏洞预警。

为什么一定要"贯穿"？因为**不同阶段能发现的漏洞类型不一样**，而且越往后越贵。需求阶段漏掉一条安全需求，最后不是"改个代码"，而是"改产品逻辑"。这和我们在第 4 章讲的测试左移是同一个道理，只不过这里的对象换成了安全。

这一页请记住一句判断标准：**如果一个团队只在发布前一周做一次渗透测试，那它做的不是安全性测试，是安全检查**。

> **板书：**一条时间轴：需求（安全需求・滥用案例）→ 设计（威胁建模・体系结构风险分析）→ 编码（安全编码・静态分析・代码评审）→ 测试（安全功能・模糊・渗透）→ 运维（补丁・监控・应急）。轴下写一句"越往后越贵"。

提问：如果一个项目已经进入测试阶段，你这时才被叫去做安全，你能补做哪些"本该更早做"的事？

预设回答：

- 答"补需求"——可以，但要换个形式：把安全需求反向从现有功能里"提取"出来，形成检查单。

- 答"补威胁建模"——可行且值得，架构还在，画数据流图、标信任边界，一两天能出一个初版。

- 答"什么都补不了，只能测"——太消极。测试阶段照样能做代码评审和静态分析，只是成本高。

- 收口：**越晚介入，越要靠"反向提取"补课**；这也说明安全人员应该更早进项目。

易错点 / 考点：这一页最爱出"阶段与动作连线题"。易错点：把"威胁建模"放在测试阶段、把"渗透测试"放在编码阶段。记忆口诀：**需求管"要什么"，设计管"哪里会破"，编码管"别写错"，测试管"打一下"，运维管"补与看"**。

过渡：贯穿 SDLC 是"位置"问题，接下来是"方针"问题——软件安全到底该按什么原则来组织。

【7.82】软件安全实施方针（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页只有两句话，但这两句话是整节安全内容的总纲，我逐句拆。

第一句：**软件安全是一种系统级的问题，需要考虑安全机制（如访问控制）和基于安全的设计（如使攻击难以实施的健壮设计）。**

拆成两半。前半是"安全机制"——认证、授权、访问控制、加密、审计，这些是可以"加上去"的构件。后半是"基于安全的设计"——不是加构件，而是让系统**天生就难打**：权限默认拒绝、失败时安全关闭、关键操作二次确认、敏感数据不落地、组件之间互不信任。这两者的区别，就像"给房子装一把好锁"和"把房子造成易守难攻的结构"。

为什么说是"系统级"？因为安全有一个残酷的性质：**它由最弱的一环决定**。你前端做了防注入，后端拼接 SQL，一样被打穿；你数据库加密了，日志里打了明文密码，一样泄漏。所以安全不能一个模块一个模块地"各管一段"，必须整体设计。

第二句：**软件安全必须成为完整的软件开发生命周期方法的一部分。** 这句话是在否定一种常见做法——"安全是安全部门的事，等项目做完了请他们来把把关"。如果安全是外挂的，它就永远是补丁；只有嵌进流程——

需求有安全条目、设计有威胁分析、编码有规范、测试有用例、发布有检查单——它才是属性。

这两句话合起来，就是这一节的方针：**系统级设计 + 全流程嵌入**。少一条都不成立。

板书：写两行大字：① 系统级：安全机制（访问控制）+ 基于安全的设计（健壮设计）→ 短板效应；② 全流程：安全嵌入 SDLC 各阶段，不做"外挂补丁"。

提问："我们买了最好的防火墙和 WAF，是不是安全就到位了？"这句话的问题在哪？

预设回答：

- 答"还差内部防护"——对，边界设备挡不住内部越权、挡不住业务逻辑漏洞。

- 答"WAF 只是缓解，不是修复"——很准。WAF 拦的是特征，改不了代码里的拼接。

- 答"没有考虑到人"——也算一条，弱口令、配置错误、社工，都是设备管不到的。

- 收口：安全是**纵深防御**：边界、应用、数据、权限、流程，每一层都要有，指望一层解决全部问题，等于没有。

易错点 / 考点：常考判断题"安全性是安全测试人员的工作"——错，安全是**全员、全流程**的属性。另一个易错点是把"安全机制"当成全部，漏掉"基于安全的设计"。答题时要把"访问控制"和"健壮设计"两个例子都写上，这是课件明确给出的。

过渡：既然要嵌进流程，那就要有一个明确的框架。业界最通行的框架叫 SDL，我们看它怎么和安全测试结合。

【7.83】全生命周期的安全性测试（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页给了全生命周期安全性测试的正式表述：**将风险分析、安全测试与软件开发过程（Security Development Lifecycle，SDL）结合起来，在软件开发的各个阶段进行误用模式、异常场景、风险分析以及渗透测试等，尽早地发现高风险的安全漏洞。**

这句话里有四个动作词，我逐个解释，因为它们正好对应四类不同的测试思路。

误用模式：正常用例描述"用户按规矩操作会得到什么"，误用模式描述"有人不按规矩操作会得到什么"。比

如正常是"输入正确的手机号获取验证码"，误用是"同一手机号一秒请求一百次"。误用模式是从需求阶段就能开始写的，它比漏洞清单更贴近业务。

异常场景：接口传空、传超长、传特殊字符、乱序调用、并发调用、超时重试。异常场景测的是"系统的边界和假设"，很多高危漏洞就藏在"开发者以为不会发生"的地方。

风险分析：哪些资产值钱、哪些模块暴露在外、哪些接口没有鉴权。风险分析决定**测试资源的分配**——你不可能把所有地方都测一遍，必须把力气花在高风险处。

渗透测试：前面讲过的，用攻击者视角做验证。

注意这句话的落脚点是"**尽早**发现高风险漏洞"。为什么要强调尽早？因为一个架构级的安全缺陷，在设计阶段发现是"改图"，在上线后发现是"改系统"，而系统已经在跑、数据已经存在、用户已经养成习惯——那时候的修复往往只能打折。

还有一点：SDL 不是一门新技术，它是一套**把安全动作排进开发节奏**的管理方法。它要求每个阶段有明确的交付物和检查点，比如设计阶段必须交威胁模型，编码阶段必须过静态扫描。

> **板书：**中间写"SDL"，四周四个动作：误用模式（需求）／异常场景（设计＋测试）／风险分析（全程）／渗透测试（测试＋运维）；下面写一句"尽早 = 最便宜"。

提问：写"误用模式"和写"测试用例"，思维上最大的差别是什么？

预设回答：

- 答"一个是正面一个是反面"——方向对，再往深一层：误用模式假设**使用者有恶意**，用例假设**使用者会犯错**。

- 答"误用模式要考虑绕过"——很到位，比如绕过前端校验直接调接口，这就是典型的误用思路。

- 答"差不多，换个说法"——不对，思维前提不同，产出也不同：误用模式产出的是"攻击场景清单"，用例产出的是"输入输出对应关系"。

- 收口：所以做安全测试，第一件事是**把"用户都很老实"这个假设从脑子里删掉**。

易错点 / 考点：① SDL 的全称与含义（Security Development Lifecycle）常考填空；② 四个动作的名称要与阶段大致对应，答简答题时至少写全"误用模式、异常场景、风险分析、渗透测试"四项；③ 易错点是把 SDL 当成"某个工具"或"某个标准"——它是**过程框架**。

过渡：SDL 在设计阶段最核心的产出，是体系结构的风险分析。我们看它具体要分析什么。

【7.84】体系结构风险分析（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲设计阶段的安全动作，标题是体系结构风险分析。它的分析对象是**体系结构**，不是代码——这一点很重要，因为结构性问题改代码是改不掉的。

这一页给了三项内容：工件、风险例子、以及一条方法论结论。

工件是"设计和说明书"。也就是说，做这件事不需要代码，只需要架构图、部署图、接口说明、数据流说明。这解释了为什么它能在设计阶段就做——**它靠读文档就能发现风险**。

风险例子给了两个，都很典型。第一个：**对关键数据的区分和保护很糟糕**。什么叫"区分"？就是没有把数据分级——用户的手机号、身份证号、银行卡号，和一条商品评论，敏感度完全不同，保护强度也该不同。全都明文存、全都同一套权限，就是"区分很糟糕"。第二个：**Web 服务未能验证调用代码及其用户**。服务之间互相调用时，如果只认"请求来自内网"而不认"这个调用方是谁、它代表哪个用户"，那么任何能进内网的人都能以任意用户身份调接口。这就是常说的"内部接口裸奔"。

再看那句方法论：**系统体系必须连贯一致，并提供统一的安全防线；应用文档要清晰地记录各种前提假设，并确定可能的攻击。**"连贯一致"是说不能东边做了鉴权、西边忘了；"统一防线"是说安全要有一个收口的地方，而不是散落在每个模块里各写一套；"记录前提假设"是最容易被忽略的——比如"这个接口只被 App 调用""这个字段前端会校验"，这些假设一旦不成立，漏洞就出现了。

结论很清楚：**体系结构风险分析是必需的**，用来揭示结构瑕疵、评估并降低风险；而且风险在生命周期的各个阶段都会出现，所以要**持续**做风险管理、不断监控。

一句话：**结构错了，代码写得再规范也救不回来。**

> **板书：**三行：① 工件＝设计与说明书（不需要代码）；② 典型风险＝关键数据未分级／未校验调用方身份；③ 要求＝体系连贯一致・统一安全防线・写明前提假设・持续风险管理。

> **提问：**"这个接口只给 App 用"，这句话作为设计假设，会带来什么风险？

> **预设回答：**

- 答"别人直接调接口就绕过了 App"——正是。只要接口没有独立鉴权，抓包就能重放。

- 答"App 里的校验等于没校验"——对，客户端校验只是体验优化，永远不能当安全边界。

- 答"那就在网关上加限制"——方向对，但要问清楚：限制的是 IP、频率，还是身份？前两个都能绕过。

- 收口：凡是写成"只有 X 会来调用"的假设，都要追问一句：**如果 Y 来了，系统认不认？**

> **易错点 / 考点：**① 这一页的工件是"设计和说明书"，不是代码——常考判断题；② "未验证调用代码及其用户"这句要能翻译成"服务间调用缺少身份认证"，答题时最好补一句"应引入服务身份与用户身份的双重校验"；③ 注意"持续的风险管理"这半句，很多同学只记"要做风险分析"，漏掉"要持续、要监控"。

> **过渡：**设计阶段看结构，编码阶段看代码。下一页就是代码评审在安全视角下查什么。

【7.85】代码评审示例（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页讲代码评审，题目里带"示例"两个字，重点是"看什么、怎么看出问题"。先把它和第 3 章接上：代码评审属于**静态测试**，不运行程序，靠人读代码找缺陷。它最便宜，也最容易被跳过。

安全视角的代码评审，我给它归纳成一张检查单，六条，你以后做课程设计可以直接拿去用。

第一条，**输入校验**。所有来自外部的数据——请求参数、请求头、上传文件、Cookie、第三方回调——有没有在服务端校验？注意是服务端，不是前端。看代码时找的是"直接用 request 取值然后拼进语句"的地方。

第二条，**SQL 与命令拼接**。看到字符串拼接出来的 SQL、拼接出来的 shell 命令、拼接出来的文件路径，就是红灯。正确做法是参数化查询、白名单校验。

第三条，**权限判断**。每个敏感操作前有没有校验"当前登录人是不是有这个权限、是不是这条数据的主人"？最常见的漏洞不是"没登录也能访问"，而是"登录了但不是你的数据你也能看"，也就是后面要讲的直接对象引用。

第四条，**加密与密钥**。密码是不是用了不可逆的哈希加盐？密钥有没有硬编码在代码或配置文件里？传输有没有走加密通道？

第五条，**日志与错误信息**。日志里有没有打明文密码、身份证号、完整卡号？异常有没有把堆栈直接返回给前端？错误信息越详细，攻击者越省事。

第六条，**错误处理与失败模式**。失败的时候是"拒绝"还是"放行"？很多越权漏洞的根源就是异常分支里忘了return。

评审的产出不是"我看过了"，而是**问题清单 + 责任人 + 修复确认**。没有记录的评审等于没做。

板书：六条检查单竖排：输入校验（服务端）／SQL 与命令拼接／权限判断（含数据归属）／加密与密钥／日志与报错／失败模式。右上写"评审产出＝问题清单＋责任人＋闭环"。

提问：评审时看到这样一段逻辑：先查询订单，如果查询抛异常就跳过后面的归属校验直接返回订单详情。你会怎么判？

预设回答：

- 答"这是 bug，异常了不该返回数据"——对，而且它同时是**功能缺陷**和**安全漏洞**。

- 答"要看异常是不是可被外部触发"——很专业。如果攻击者能构造一个让查询抛异常的入参，这就变成了可利用的越权。

- 答"加个 try-catch 就好了"——方向反了，正确做法是**失败即拒绝**，而不是把异常吞掉继续走。

- 收口：这类问题就是"失败模式不安全"，写进检查单，以后每份代码都查一遍。

易错点 / 考点：① 代码评审是**静态测试**，与"运行程序看结果"的动态测试相对，这是第 3 章的知识点，常跨章考；② 常考简答"代码评审能发现哪类安全问题"——注入、硬编码密钥、越权、信息泄漏、不安全失败模式；③ 易错点是把"前端校验"当安全措施，考试里这样答会被扣分。

过渡：评审是逐行看，但项目那么大，不可能每行都看。所以需要一个"按风险分配力气"的策略。

【7.86】基于风险的安全测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给的是安全测试的**策略层**答案：力气怎么分配。它开头写了"工件：单元和系统"，意思是这套策略既用在单元级，也用在系统级。

策略有两条，注意这两条是**并列**的，不是二选一。

第一条：**用标准功能测试技术来进行的安全功能性测试**。这句话的意思是，安全功能本身也是功能，可以用

等价类、边界值、判定表这些常规技术来测。比如"密码错误五次锁定"，这就是一条功能需求，用边界值测四次、五次、六次；"只有管理员能删除用户"，用判定表把角色和操作组合起来测。**这一条最容易被忽略**：很多同学一听安全测试就想到渗透，其实大部分安全需求是可以用普通功能测试手段验证的。

第二条：**以攻击模式、风险分析结果和滥用案例为基础的基于风险的安全测试**。三个输入：攻击模式（业界已知的攻击手法库）、风险分析结果（哪里值钱、哪里暴露）、滥用案例（前面讲的误用模式）。输出是针对高风险点的定向测试。

为什么需要第二条？课件给了一句很硬的话：**标准的质量保障方法可能不能揭示所有严重的安全问题。**道理不难懂——常规测试验证的是"功能符合需求"，而安全问题是"功能符合需求，但可以被滥用"。你的需求里根本没写"不能被注入"，那按需求测当然测不出来。

最后那句是全页的点睛：**安全测试是为了保证坏的事情不会发生，像攻击者一样考虑问题很重要。**这句话可以当成整个 7.2 节的座右铭。功能性测试问"该发生的发生了吗"，安全性测试问"不该发生的有没有可能发生"。

落到课程设计上：你的安全测试用例表，最好有两栏——一栏来自需求（安全功能），一栏来自攻击模式（风险驱动）。两栏都空的项目，基本等于没做安全测试。

板书：写"两条策略并列"：① 安全功能测试 ← 标准功能测试技术（等价类／边界值／判定表）；② 基于风险的安全测试 ← 攻击模式＋风险分析＋滥用案例。下方写一句"常规方法测不出'合规但可被滥用'"。

提问：需求文档里写着"用户可以修改自己的昵称"，按功能测试这条需求是合格的。从安全角度你还想补什么用例？

预设回答：

- 答"试试昵称里能不能塞脚本"——对，这就是存储型 XSS 的入口。

- 答"试试改别人的昵称"——关键，这就是越权，测试时把用户 ID 换成别人的。

- 答"试试超长昵称、emoji、控制字符"——也算，属于异常输入，可能导致截断或显示错乱。

- 收口：功能需求描述的是"正常路径"，安全测试要补的是"这条路径周围所有的旁门"。

易错点 / 考点：① 两条策略要能同时写出来，只答"做渗透测试"是不完整的；② "安全测试是为了保证坏的事情不会发生"这句话常作为简答题的引语，答的时候要点出"与功能测试的目标相反"；③ 易错点：把"基于风险"理解成"高风险就不测了"——恰恰相反，是**高风险多测、低风险少测**。

过渡：策略讲完，我们把镜头推到最常见的战场——Web。先看 Web 与数据库这一层的测试点。

【7.87】Web 安全性测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

从这里开始进入 Web 安全性测试。为什么单独给 Web 一大块？因为在绝大多数系统里，**Web 是唯一对公网开放的那一层**，也就成了最大的攻击面。

想清楚 Web 为什么天然难防，有三个原因。

第一，**输入不可信而且必须收**。Web 应用的本质就是"把陌生人的字符串收进来，变成数据库操作、文件操作、命令操作"。这个转换过程只要有一处没做对，就是注入类漏洞。

第二，**状态靠客户端维持**。HTTP 本身是无状态的，服务端靠 Cookie 或者 Session ID 认人。而 Cookie 存在用户的浏览器里——存在别人机器上的东西，就必须假设它可能被偷、被改、被伪造。这就带来了会话劫持、CSRF 这一整类问题。

第三，**权限判断分散在每一个接口里**。桌面软件可能只有几个入口，Web 应用动辄几百个接口，每个接口都要自己判断"这个人能不能干这件事"。漏一个，就是一个越权。

所以 Web 安全性测试的思路也就三条：**测输入输出（注入、XSS）、测身份与会话（认证、Session、劫持）、测授权（越权、直接对象引用、功能级权限）**。后面几页就是按这三条展开的。

补一句方法论：Web 安全测试不是"拿工具扫一遍"，工具只能覆盖已知模式。真正的功夫在"理解这个业务的数据流"，因为业务逻辑漏洞——比如"优惠券能不能重复领""订单金额能不能改成负数"——没有任何扫描器扫得出来。

> **板书：**中间写"Web＝最大攻击面"，三根支柱：输入输出（注入・XSS）｜身份与会话（认证・Session・劫持）｜授权（越权・对象引用・功能级权限）。下方写"业务逻辑漏洞：工具扫不出，靠理解数据流"。

提问：为什么说"业务逻辑漏洞是扫描器永远扫不到的一类"？

预设回答：

- 答"因为它是合法请求"——正是，参数类型、长度、格式全都合法，扫描器看不出异常。

- 答"因为它依赖业务规则"——对，扫描器不知道"一个人只能领一张券"这条规则。

- 答"可以用自动化脚本测"——可以，但那已经不是通用扫描，而是**为这个业务专门写的用例**。

- 收口：所以安全测试人员的核心能力，是把业务规则翻译成"可以被违反的假设"。

> **易错点 / 考点：**这一页常用来出"为什么 Web 是安全测试重点"的简答，标准答题点：公网可达、输入不可信、会话在客户端、接口数量多导致授权分散。易错点：只答"Web 用户多"，那不是技术原因。

> **过渡：**Web 层往下就是数据库层。这两层的测试点，课件列了五条。

【7.88】Web/数据库 渗透测试点（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把 Web 与数据库这一条链上的检查点收成五条，我按"从架构到接口"的顺序讲，每条都给出"要防的是什么"。
>
> 第一条，**检查应用系统架构，防止用户绕过系统直接修改数据库**。这句话听起来抽象，实际场景很具体：数据库端口暴露在公网、数据库账号密码写在前端配置里、中间件管理台没有改默认口令。攻击者一旦连上数据库，前面所有的应用层防护全部作废。测试方法：从外网扫端口，看数据库、缓存、管理后台是否可达；查客户端代码里有没有连接串。
>
> 第二条，**检查身份认证模块，防止非法用户绕过身份认证**。要测的是"能不能不做认证就成为已登录用户"：接口是否全部强制鉴权、Token 是否可伪造、验证码是否可复用、登录失败是否有限制、有没有默认账号。
>
> 第三条，**检查数据库接口模块，防止用户获取系统权限**。这里的"系统权限"指的是数据库或操作系统层面的权限。典型问题：注入导致可以执行数据库扩展、可以读写文件、可以调用系统命令；数据库账号权限过大，一个业务账号能建表、能删库。
>
> 第四条，**检查文件接口模块，防止用户获取系统文件**。上传、下载、导入、导出、预览，每一个都是文件接口。要测：能不能上传可执行脚本并访问到、能不能通过路径穿越读到配置文件、下载接口能不能越权拿到别人的文件。
>
> 第五条，**检查其他安全威胁**。这是个兜底项，覆盖前面没提到的：日志泄漏、敏感信息未脱敏、接口返回过多字段、调试接口未关闭、第三方组件已知漏洞。
>
> 这五条的价值在于它给了你一张**按模块划分的检查地图**。做课程设计的时候，你可以直接按这五条各写一组

用例，覆盖面就不会太差。

> **板书：**五条竖排，每条后面标"防什么"：架构（绕过应用直连库）｜身份认证（绕过登录）｜数据库接口（拿到系统权限）｜文件接口（读到系统文件）｜其他（信息泄漏・组件漏洞・调试接口）。

> **提问：**如果测试时发现前端 JavaScript 里写着数据库的连接地址和账号，这属于上面哪一条的问题？严重程度怎么判？

> **预设回答：**

- 答"第一条，架构问题"——对，而且是最严重的那一类，因为它直接给了绕过应用的通道。

- 答"也算第五条，信息泄漏"——可以这么说，但根因在架构，不能只当信息泄漏处理。

- 答"要看这个账号的权限"——关键的追问。如果这个账号只有只读权限，危害有限；如果是业务主账号，等于库已经丢了。

- 收口：判定严重程度看两件事：**暴露面有多大**（外网还是内网）、**这个凭据能干什么**（只读还是可写）。

> **易错点 / 考点：**五条检查点的顺序与内容常考。易错点是把"数据库接口模块"和"文件接口模块"混着答——前者防的是**权限提升**，后者防的是**文件读取**。另外注意"防止用户绕过系统直接修改数据库"这句话，是架构层的考点，答题时要写"数据库不应对外暴露、应用应作为唯一访问入口"。

> **过渡：**五条地图铺好了，接下来逐个看具体的漏洞类型。课件先给了四类最常见的 Web 攻击。

【7.89】Web 安全性 渗透测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把 Web 渗透测试的对象归成四类，这是很经典的归纳：**注入攻击（Injection）、跨站点脚本攻击（XSS）、跨站点请求伪造（CSRF）、未能限制 URL 访问**。四类问题，对应四种"信任被滥用"的方式。
>
> 第一类，**注入攻击**。系统的错误在于：**把用户提供的数据当成了代码**。SQL 注入是把数据当成了 SQL 语句的一部分，命令注入是把数据当成了 shell 命令，XML 注入是把数据当成了 XML 结构，LDAP 注入、表达式注入同理。防御只有一条根本原则：**数据永远是数据，代码永远是代码**——用参数化、用转义、用结构化 API，不要用字符串拼。
>
> 第二类，**XSS**。系统的错误在于：**把用户提供的数据当成了给别人的代码**。你自己看这段数据没事，但它被存

下来、显示给另一个用户时，就在对方的浏览器里执行了。所以 XSS 是"借你的浏览器打你的账户"。

第三类，**CSRF**。系统的错误在于：**把浏览器自动带上的凭证当成了用户本人的意愿**。你登录过 A 站，Cookie 还在，那么任何页面诱导你发一个到 A 站的请求，A 站都会认为这是你本人在操作。防御的关键是加一个攻击者猜不到的令牌。

第四类，**未能限制 URL 访问**。系统的错误在于：**把"页面上没有链接"当成了"访问不到"**。隐藏不等于保护，只要 URL 在，直接敲进去就能开。

这四类的共同点，你听出来了吗？**全部都是"服务端过度相信了客户端"**：相信客户端传来的数据无害、相信客户端显示的就是全部、相信客户端的请求是自愿的、相信客户端看不到的链接就访问不到。这一句，就是整个 Web 安全性测试的总纲。

板书：四行"攻击 → 被滥用的信任 → 根本防御"：注入（数据被当代码 → 参数化）｜XSS（数据被当代码给别人 → 输出编码＋CSP＋HttpOnly）｜CSRF（浏览器凭证被当意愿 → Token＋SameSite）｜未限制 URL 访问（隐藏≠保护 → 服务端逐请求鉴权）。

提问：这四类里，哪一类完全可以在"没有任何输入框"的页面上发生？

预设回答：

- 答"CSRF"——很准。受害者只要访问了一个恶意页面，请求就发出去了，全程没在任何输入框里打字。

- 答"未限制 URL 访问"——也对，连输入都不需要，改地址栏或者直接用工具请求就行。

- 答"XSS 也要输入框吧"——不一定，反射型可以靠链接参数，存储型可以靠昵称、评论、文件名，甚至日志展示页。

- 收口：所以"这个页面没有表单所以安全"是一个典型错觉，攻击面取决于**服务端收哪些参数**，而不是页面上画了什么。

易错点 / 考点：四类名称中英对照要能对上，尤其 CSRF 的中文译名"跨站点请求伪造"常被写成"跨站脚本伪造"。易错点是把 XSS 和 CSRF 搞混：**XSS 是"在受害者浏览器里执行我的代码"，CSRF 是"借受害者浏览器发我的请求"**；前者要执行脚本，后者不需要看到任何返回。

过渡：四类里最常见、也最容易被学生写进课程设计的一类，是 XSS。我们单独看它。

【7.90】针对 XSS 进行测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲 XSS，也就是跨站脚本。定义很简洁：**允许恶意用户将代码植入到"供其它用户使用的 Web 页面"中**。注意这句话的两个要素：一是"代码"，二是"给别人看的页面"。缺一个都不构成 XSS。

课件给的例子是：一般输入域中能输入 `javascript:alert(document.cookie)`，然后在其它页面显示出来。为什么它能成功？因为服务端把用户提交的内容**原样**放进了 HTML 里。浏览器解析 HTML 时，看到 `<script>` 就当代码执行，看到 `onerror=` 就当事件处理器执行——它没法区分"这是程序员写的"还是"这是用户填的"。

课件点了三个危害层次，一层比一层重。第一层：**具有隐藏性**——脚本藏在正常页面的内容里，管理员看页面时根本看不出来。第二层：**受害者的敏感数据、session、cookie 等会泄漏**——脚本能读 `document.cookie`，能读页面上的任意内容，还能悄悄发给攻击者。第三层：**黑客能监控用户的行为、用户被迫访问其它站点**——这就从"偷东西"变成了"控制这个浏览器"。

测试怎么做？按三个位置分。**反射型**：把 payload 放进 URL 参数，看回显页面里有没有被原样输出。**存储型**：把 payload 存进数据库——昵称、评论、收货地址、文件名——再看哪些页面会展示它。**DOM 型**：payload 不进服务端，直接被前端 JavaScript 用 `innerHTML` 之类的方式插进页面，这类要看前端代码而不是看响应体。

防御怎么做？四条一起上：**输出编码**（按上下文做 HTML、属性、JS、URL 编码）、**输入校验**（白名单，尤其是富文本）、**CSP 内容安全策略**（禁止内联脚本）、**Cookie 加 HttpOnly**（就算被注入了，也读不到 Cookie）。记住：**转义要在输出的地方做，不是输入的地方做**。

> **板书：**写"XSS＝把用户的数据当代码，在别人的浏览器里执行"；三型：反射（URL 参数）／存储（入库后展示）／DOM（前端 innerHTML）；防御四条：输出编码・输入白名单・CSP・HttpOnly。

提问：为什么很多教材强调"转义要在输出时做，而不是输入时做"？

预设回答：

- 答"因为同一份数据在不同地方要不同的转义"——非常准。存进数据库要 SQL 转义，输出到 HTML 要 HTML 转义，放进 JS 要 JS 转义。

- 答"输入时转义会把数据改坏"——也对，用户在昵称里打了引号，你给他转成实体，展示出来就变形了。

- 答"两个都做更保险"——可以做，但要明白真正起作用的是输出那一道；只做输入过滤，遇到二次注入照样失效。

- 收口：一句话——**数据保持原样存储，按使用场景在边界处编码**。

> **易错点 / 考点：** ① 三种类型的区分是高频考点：**反射型**要诱导点击、**存储型**无需交互危害最大、**DOM 型**服务端日志里看不到 payload；② 常考判断"对输入做了长度限制就能防 XSS"——错，短 payload 一样能执行；③ 答题要区分"输入校验"和"输出编码"，只写一个不完整。

> **过渡：** XSS 是往页面里塞脚本，下面这一类是往数据库语句里塞条件——SQL 注入。

【7.91】SQL 注入式攻击（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> SQL 注入。课件给的定义非常精炼：**根据 SQL 语句的编写规则，附加一个永远为"真"的条件，使系统中某个认证条件总是成立，从而欺骗系统、躲过认证，进而侵入系统。**课件举的例子是 `' OR '1' = '1`。
>
> 我们把这条 payload 放进一个真实的登录逻辑里看它为什么能成立。假设后台拼出来的语句是：`SELECT * FROM users WHERE name = '` + 用户输入 + `' AND pwd = '` + 密码 + `'`。当用户名填 `' OR '1' = '1`，拼出来就变成了 `WHERE name = '' OR '1' = '1' AND pwd = '...'`。`'1'='1'` 永远为真，于是这个 WHERE 条件对表里**每一行**都成立，查询返回了第一条用户记录，登录就成功了。攻击者连密码是什么都不需要知道。
>
> 注意这里的关键不是那条 payload 有多巧妙，而是**拼接**这个动作。只要 SQL 是拼出来的，攻击者就能往里面塞语法；只要 SQL 是参数化的，无论他塞什么，都只是一个字符串值。这就是为什么防御的根本手段是**参数化查询（预编译）**——它让数据库先编译语句结构，再把参数当数据填进去，结构就改不了了。
>
> 除了绕过登录，SQL 注入还能干更严重的事：**拖库**（用 UNION 把其他表的数据拼进查询结果）、**盲注**（靠页面真假或响应时间一点点猜数据）、**提权与命令执行**（某些数据库支持扩展，能读写文件甚至执行系统命令）。所以它的危害等级通常是最高的那一档。
>
> 测试怎么测？三个层次：**手工验证**——在参数后加单引号看是否报错，加 `AND 1=1` 与 `AND 1=2` 看响应差异；**报错注入**——利用数据库报错信息带出数据；**时间盲注**——构造 `SLEEP` 观察响应时间。工具层面就是 SQLMap 这类，但工具跑出来的结果必须人工确认，误报率不低。

防御再补三条：**数据库账号最小权限**（业务账号不该有建表删库权限）、**关闭详细报错**（别把 SQL 语句返给前端）、**统一收口**（所有查询走同一层数据访问代码，别让某个开发者随手拼一句）。

板书：写拼接示例：`WHERE name = '` + 输入 + `'`；把 payload `' OR '1'='1` 代入，箭头指向"条件恒真 → 返回任意用户"；右侧写防御五条：参数化・最小权限・关报错・统一收口・WAF 兜底。

提问：如果一个系统对所有输入都做了"过滤掉单引号"，是不是就防住了 SQL 注入？

预设回答：

- 答"是"——这是最典型的错误。数字型注入根本不需要单引号，`id=1 OR 1=1` 就够。

- 答"不一定，还有别的写法"——对，编码绕过、宽字节、注释符都能绕。

- 答"应该用参数化，而不是过滤"——这才是根本答案。过滤是黑名单思维，永远补不完；参数化是结构上根治。

- 收口：**黑名单防注入，等于用漏勺挡水**；白名单和参数化才是正路。

易错点 / 考点：① 常考"SQL 注入的原理"——答题必须写出"用户输入被当作 SQL 语法的一部分参与解析"，只写"输入了特殊字符"不得分；② 防御措施里 **参数化查询/预编译** 是必答项，其次是最小权限、错误信息屏蔽；③ 易错点：把"前端做了长度和字符校验"当作防御，考试里这是典型的错答。

【思政落点 6】：SQL 注入这一页，技术上只有一行 payload，但它背后的道理值得多说两句。同一个知识点，写测试用例的人用它证明系统能被绕过，写代码的人用它知道"拼接"这两个字有多危险。技术本身没有立场，**立场在使用技术的人身上**。所以我要求同学们做课程设计时，凡是要演示注入，只在自己的靶机、自己的测试库上做，用例里写清"验证点、预期结果、修复建议"三栏。这就是"能力配得上责任边界"最具体的做法——你的手艺越锋利，越要知道刀口朝哪。

过渡：SQL 注入是往关系型语句里塞条件，同样的思路换个载体，就是 XML 注入。

【7.92】XML 注入（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

XML 注入。这一页给了两段 XML 对照，我们看差别在哪里。

正常的一段是：一个 `USER role=guest` 节点，里面有 name 和 email。注意 `role` 这个属性——它是权限标记。

被注入之后的那段，多出来一个 `USER role=admin` 的节点，里面是一个叫 test 的用户。也就是说，**原本只有一个访客用户的 XML 文档，凭空多出了一个管理员**。

它是怎么发生的？和 SQL 注入一模一样：程序把用户输入**拼接**进了 XML 字符串。假设注册接口是"把用户填的 name 和 email 拼进一个 `<USER role=guest>` 节点里"，那么攻击者在 email 字段里填上"闭合当前节点、再补一个新的 admin 节点"的内容，拼出来的文档结构就被改写了。系统后面读这个 XML 做鉴权，读到那个 admin 节点，就把他当管理员了。

所以 XML 注入的本质，和 SQL 注入、命令注入、LDAP 注入是**同一件事**：**数据里含有结构性字符，被拼进了结构化文档，于是数据的边界被突破，变成了新的结构。**

危害有三个方向。第一，**越权**——像这个例子，凭空多一个高权限用户。第二，**数据篡改**——改写已有节点的值，比如把金额、状态改掉。第三，**信息泄漏与实体攻击**——XML 支持自定义实体，如果解析器允许外部实体（XXE），攻击者就能让解析器去读服务器上的文件，甚至发起内网请求。

防御三条：**用 XML 库的构造 API 而不是字符串拼接**（这和参数化查询是一个道理）、**对写入的内容做实体转义**（把 `<`、`>`、`&`、引号转掉）、**关闭外部实体解析**（禁用 DTD 和外部实体，这一条专门防 XXE）。

测试怎么做？在每一个会进入 XML 的字段里，尝试输入 `</name><USER role=admin>` 这类闭合结构，然后看生成的文档里有没有多出节点；再试 `<!DOCTYPE` 与实体引用，看解析器会不会去访问外部资源。

> **板书：**画两段 XML 对照，用箭头标出"多出来的 admin 节点"；下面写一句本质："结构化字符 + 字符串拼接 = 结构被改写"；防御三条：构造 API・实体转义・禁用外部实体。

提问：XML 注入和 SQL 注入，你觉得哪一个更容易被开发者忽视？为什么？

预设回答：

- 答"XML，因为用得少"——有道理，很多人以为 XML 是"配置文件"，不觉得它是攻击面。

- 答"XML，因为大家不知道它也能注入"——对，安全意识往往只覆盖 SQL 和 XSS。

- 答"SQL 更常见，所以更容易被忽视"——不完全对，SQL 注入现在有大量现成防护和框架默认参数化，反而是 XML、JSON 这类"看起来无害"的格式容易裸奔。

- 收口：判断风险不看"用得多不多"，看**有没有把外部数据拼进结构化文档**。只要拼了，就有注入。

易错点 / 考点：① 这一页的图要能读懂："注入后多出一个 role=admin 的 USER 节点"是必考细节；② 常考"XML 注入与 XXE 的区别"——XML 注入是**改写文档结构**，XXE 是**利用外部实体读取资源**，两者相关但不是

一回事；③ 防御必答"不要用字符串拼接构造 XML""禁用外部实体"。

过渡：注入类问题解决的是"伪造数据"，下面这一类解决的是"伪造身份"——会话劫持。

【7.93】会话劫持（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

会话劫持。先讲清楚 HTTP 为什么会有这个问题：HTTP 是无状态的，服务器不记得你是谁，所以每次请求都要带一个凭证——Cookie 里的 Session ID。服务器靠这个 ID 判断"这还是刚才那个人"。**谁拿到这个 ID，谁就是你。**

课件写得很直接：**由于 Cookie 失效时间很长，攻击手段一般是盗取用户 Cookie 然后伪造 Cookie 冒充该用户，从而劫持会话。**注意这里点了"失效时间很长"——这是一个设计上的取舍失误。为了方便，很多系统把 Cookie 的有效期设成三十天甚至更久，于是这三十天里，只要这个字符串泄漏一次，攻击者就能长期冒用。

盗取的方式有哪些？课件点了一个最典型的：**通过 XSS 来劫持，用 `document.cookie` 拿到 Cookie**。除此之外还有：中间人窃听（没走 HTTPS 的明文流量）、物理接触（公用电脑没退出登录）、日志或 URL 里带了 Session ID。

防御要分四层，我按重要性排。

第一，**Cookie 加 HttpOnly**。这一条直接掐断 XSS 读 Cookie 的路径，成本极低，收益极高。

第二，**全程 HTTPS，Cookie 加 Secure 属性**。让凭证不在明文信道里跑。

第三，**缩短会话有效期，并且登出即失效**。服务端要真的把会话销毁，而不是只清浏览器的 Cookie。

第四，**绑定上下文**。把会话和 IP、User-Agent 之类做弱绑定，或者每次敏感操作要求二次验证。注意"弱绑定"——绑太死会让用户换个网络就掉线。

还有一条工程实践：**Session ID 要在登录成功后重新生成**。这防的是"会话固定"——攻击者先给你一个他知道的 ID，等你登录后，这个 ID 就变成了已登录状态，他也就能用了。

一句话总结这一页：**会话凭证就是临时的身份证，它的安全性等于整个系统的安全性。**

板书：写"HTTP 无状态 → 靠 Session ID 认人 → 谁有 ID 谁就是你"；左侧列泄漏途径：XSS・明文传输・公

> 用电脑・URL 带 ID；右侧列防御四层：HttpOnly・HTTPS＋Secure・短有效期＋登出销毁・弱绑定＋登录后重生成。

> **提问：**为什么"登出"这个动作，在安全上比很多人想的更重要？

> **预设回答：**

- 答"防止别人用我的账号"——对，尤其在公用电脑上。

- 答"要服务端真的销毁会话，不能只清前端"——非常关键，很多系统只删了浏览器 Cookie，服务端那个 Session 还活着。

- 答"还能防会话固定"——思路对，登录和登出都应该触发 ID 重生成。

- 收口：判断标准很简单——**登出之后，拿旧 Cookie 再发一次请求，如果还能成功，这个功能就是没做完**。

> **易错点 / 考点：**① 会话劫持与 CSRF 的区别常考：劫持是**拿到你的凭证自己用**，CSRF 是**借你的浏览器发请求、不需要拿到凭证**；② 防御措施里 **HttpOnly** 是最常考的一条，其次是 HTTPS、登出销毁、登录后重生成 Session ID；③ 易错点：以为"Cookie 加密了就安全"——加密只防内容泄漏，不防被整体盗用重放。

> **过渡：**会话劫持是"偷到凭证"，还有一类更基础的问题：凭证和会话管理本身就没设计对。

【7.94】无效认证和 Session 管理方式（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页列了五条"无效的认证与会话管理"，属于设计层面的缺陷。我逐条讲，每条都给出"为什么错"和"怎么改"。
>
> 第一条，**用户密码在数据库中用明文保存、用户登录使用未加密连接**。明文存密码的后果不是"被管理员看到"，而是"一旦库被拖走，所有用户在所有网站的账号都危险"——因为人会复用密码。正确做法是加盐哈希（比如 bcrypt、scrypt 这类慢哈希），并且登录走 HTTPS。注意"加盐"和"慢"这两个词都重要：加盐防彩虹表，慢哈希让离线爆破变得不划算。
>
> 第二条，**用户密码可以被特殊操作覆盖**。课件举的例子很具体：不正确地实现密码更改功能、恢复密码功能，都可能造成密码被直接更改。典型漏洞是"改密码接口只校验了新密码，没校验旧密码"，或者"找回密码的验证码可以爆破、可以复用"。改密码必须验旧密码，重置必须验一次性凭证。

第三条，**网页 URL 中包含 Session ID**。为什么错？URL 会被记进浏览器历史、会被记进服务器访问日志、会通过 Referer 头泄漏给第三方站点、会被用户复制粘贴发出去。Session ID 正确的存放位置是 Cookie（加 HttpOnly 和 Secure），不是 URL。

第四条，**Session ID 不会 timeout，或者 session／token／SSO token 在登出的时候未失效**。这一条在单点登录场景里特别常见：用户在 A 系统点了登出，B 系统那边的 token 还活着。要改的是"统一登出"——登出要通知所有依赖方。

第五条，**用户认证信息、Session ID 使用未加密连接传输**。这一条和第一条后半句是一回事，但它单独列出来，是因为它覆盖的不只是登录页——任何一个带着 Cookie 的请求走了 HTTP，凭证就暴露在链路上。

五条合起来看，都是"认证与会话"这个地基上的裂缝。地基裂了，上面加多少防护都白搭。

> **板书：**五条竖排，每条右边标"正确做法"：明文存/明文传 → 加盐慢哈希＋HTTPS；密码可被覆盖 → 改密验旧、重置用一次性凭证；URL 带 Session ID → 只放 Cookie（HttpOnly＋Secure）；不超时/登出不失效 → 超时＋统一登出；凭证明文传输 → 全站 HTTPS。

提问：为什么"把 Session ID 放在 URL 里"在移动端和分享场景下尤其危险?

预设回答：

- 答"链接会被分享出去"——正是，用户一转发，等于把登录态送人。

- 答"会被日志记下来"——对，Nginx 访问日志、CDN 日志、代理日志，全都会留一份。

- 答"Referer 会带出去"——很专业，跳转到第三方时 URL 会作为 Referer 发出去。

- 收口：所以判断标准是——**这个凭证会不会出现在任何"会被记录或被分享"的地方**。会，就不合格。

易错点 / 考点：五条内容常考多选与简答。易错点：① 把"密码加密存储"当成正确答案——应是**加盐哈希**，可逆加密仍然不安全；② 忽略"登出未失效"这一条，它常出现在 SSO 相关题目里；③ 常考判断"Session ID 放在 URL 里便于多标签页共享"——错，这是典型的危险设计。

> **过渡：**认证和会话之外，还有一类很隐蔽的问题：页面跳转。它看起来只是"跳一下"，实际能把你送到钓鱼站。

【7.95】未验证跳转（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

未验证跳转，英文常叫 Open Redirect（开放重定向）。定义：**有些页面跳转会根据用户输入来决定，这就给了攻击者可乘之机，从而将用户导向恶意网站或者未授权链接。**

课件给了两个例子，一个偏"钓鱼"，一个偏"越权"，我们分开看。

第一个例子：`http://www.example.com/redirect.jsp?url=evil.com`。这个页面收到 `url` 参数就直接跳过去，不做任何校验。攻击者把这个链接发给受害者，受害者看到的是**自己信任的域名**开头，点进去之后却被送到了 `evil.com`——一个做得一模一样的登录页。这里的关键在心理层面：**用户判断链接是否可信，看的是域名开头那一段**。所以这个漏洞本质上是在"借用你域名的信誉"。

第二个例子：`http://www.example.com/boring.jsp?fwd=admin.jsp`。这个更有意思，它跳的是站内页面。一个看起来无害的 `boring.jsp`，通过 `fwd` 参数能跳到 `admin.jsp`。如果 `boring.jsp` 本身不做权限校验（因为它"只是个无聊页面"），而 `admin.jsp` 依赖"访问者是从有权限的入口进来的"这种错误假设，那么这就变成了一次**绕过授权检查**。

防御怎么做？三条。第一，**不要用用户输入直接做跳转目标**；如果业务必须支持，就用**白名单**——只允许跳到预先登记过的地址，或者只允许跳站内相对路径。第二，**跳转前做归属校验**，尤其要拦住 `//evil.com`、`\/evil.com`、`%2f%2f` 这类变形。第三，**关键页面不依赖"来源"做鉴权**，每一个请求都独立校验身份和权限。

测试怎么做？把每一个带 `url`、`redirect`、`return`、`next`、`fwd`、`callback` 之类参数的接口找出来，逐个替换成外部域名，看会不会跳出去；再试各种编码变形，看白名单能不能被绕过。

> **板书：**写两条链接示例；右侧写"本质：借你域名的信誉 + 绕过入口鉴权"；防御三条：白名单・归属校验（含变形）・逐请求鉴权。参数名清单：url／redirect／return／next／fwd／callback。

> **提问：**为什么"只允许跳转到以本站域名开头的地址"这种校验，仍然可能被绕过？

> **预设回答：**

- 答"可以用 `@` 符号骗过解析"——例如 `http://www.example.com@evil.com`，人看是本站，浏览器实际去 evil.com。

- 答"可以用 `//evil.com`"——对，协议相对 URL，浏览器会当成外部域名。

- 答"可以大小写或编码变形"——也对，`%2F%2F`、反斜杠、Unicode 同形字都是常见绕过。

- 收口：所以正确做法不是"检查字符串开头"，而是**解析成 URL 对象后比对主机名**，或者干脆用映射表（给每个允许的目标一个 ID）。

> **易错点 / 考点：**① 这两个例子的区别要能说清：前者是**钓鱼**（把人送到外部），后者是**越权**（跳过授权检查到站内高权限页）；② 常考判断"跳转参数只要做了非空校验就是安全的"——错；③ 防御必答**白名单**，答题时最好补一句"校验应在服务端完成"。

> **过渡：**跳转是"送你去别处"，接下来这一类是"让你看到不该看的"——直接对象引用。

【7.96】直接对象引用（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

直接对象引用，英文是 Direct Object Reference，这一类漏洞在业务系统里出现频率极高，而且特别好理解。

课件给的代码是这样的：先拼一条查询 `SELECT * FROM accts WHERE account = ?`，用 `PreparedStatement` 预编译，然后把 `request.getParameter("acct")` 塞进第一个参数，执行查询。注意——**这段代码在防注入上是正确的**，它用了参数化。但它有一个致命的问题：**它没有校验这个 acct 是不是当前登录用户的**。

于是攻击者只要改一下 querystring，把 `acct` 换成别人的账号，就能查到任何人的信息：`http://example.com/app/accountInfo?acct=notmyacct`。这就是"直接对象引用"：**客户端直接把数据库里的对象标识（ID、账号、文件名）传上来，服务端照着用，不检查归属。**

为什么它这么常见？因为它长得不像漏洞。开发者会想："用户要改的是他自己的东西，他为什么要传别人的 ID？"——可攻击者偏偏就会传。这是典型的"以正常用户的行为模式做假设"。

防御有三条，缺一不可。

第一，**服务端做归属校验**。任何"按 ID 取数据"的接口，必须用**当前会话的身份**去过滤，而不是只靠传上来的 ID。写成 `WHERE account = ? AND owner_id = ?`，那个 owner 从会话里拿，不从请求里拿。

第二，**用间接引用**。不要暴露数据库主键，给每个对象发一个"临时映射令牌"，服务端维护"令牌 → 真实 ID"的映射，并且这个映射是和当前用户绑定的。这样就算令牌被猜，也只能被人用自己的身份用。

第三，**统一收口**。把"取对象"这件事收到一个数据访问层里做，别让每个接口自己写一遍——因为一定会有人忘。

一句话记住：**ID 是客户端给的，权限是服务端判的。这两件事永远不能混。**

板书：抄那段代码，然后在 `WHERE account = ?` 后面画红圈写"缺 owner 校验"；右侧写"改 acct 参数 → 看到别人的数据"；防御三条：归属校验（owner 从会话取）・间接引用令牌・统一收口。

提问：这段代码已经用了 `PreparedStatement`，为什么还会出问题？这是不是说明"用了参数化就安全了"？

预设回答：

- 答"参数化只防注入，不防越权"——完全正确，这是本题的核心。参数化解决的是"数据被当代码"，解决不了"这个数据该不该被你看"。

- 答"它没校验登录"——不完全是，可能校验了登录，但没校验"这条数据是不是你的"。这两个是不同层次的检查。

- 答"应该加个 owner 条件"——对，而且 owner 必须**从会话里取**，不能从请求参数里取，否则攻击者连 owner 一起改。

- 收口：安全有三问——**你是谁（认证）、你能干什么（功能权限）、这条数据是不是你的（数据权限）**。这一段代码只做了前两个中的问题，漏了第三个。

易错点 / 考点：① 直接对象引用的定义常考："客户端提交对象标识，服务端未校验访问权限"；② 易错点是把"参数化查询"当成防越权手段——它只防注入；③ 防御必答"服务端校验数据归属"与"使用间接引用/随机令牌"，答题时强调 owner 来自会话。

过渡：课件下一页给了一个更直观、也更刺眼的例子——在线银行的账户号。

【7.97】直接对象引用（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页只有一个链接：`https://www.onlinebank.com/user?acct=6065`。就这么一行，杀伤力比上一页那段代码更直观。

你看这个 URL 的结构：`user` 是接口，`acct` 是参数，`6065` 是账户号——一个**连续的、可枚举的整数**。

这意味着什么？意味着攻击者不需要任何技巧，只要把 6065 改成 6066、6067，就能一个账户一个账户地看过去。这叫**遍历**，是最朴素也最有效的一种攻击。

这一页想让你体会三件事。

第一，**连续 ID 本身就是一种信息泄漏**。它告诉你系统里有多少用户、增长速度多快、哪个号段是新注册的。所以安全设计里有一条原则：对外标识尽量用不可预测的值（UUID、随机串），而不是自增主键。

第二，**"HTTPS + 登录"不等于安全**。这个 URL 是 https 的，用户大概率也登录了，但只要没有归属校验，登录只证明"你是某个用户"，不证明"这个账户是你的"。

第三，**这类漏洞的检测成本极低、修复成本也不高，但它出现的频率极高**。因为它是"业务功能写完了、能用了"，而不是"写错了"。开发者心里想的是"接口通了"，测试心里想的是"页面能打开"，没有人问一句"换成别人的 ID 会怎样"。

所以这一类漏洞，最适合作为你们课程设计里的一个必测项。做法很简单：准备两个账号 A 和 B，用 A 登录，把 A 能访问的所有带 ID 的接口抓下来，把 ID 换成 B 的对象的 ID，逐条重放。**成功返回 B 的数据，就是漏洞；返回 403 或空，才算合格。** 这个测试方法有个名字，叫水平越权测试。

> **板书：**写那一行 URL，把 `6065` 圈出来写"连续、可枚举 → 遍历"；下方写测试方法："A 登录 → 抓带 ID 接口 → 换成 B 的对象 ID → 重放"；结论三句：连续 ID 即泄漏・HTTPS≠授权・必测项。

提问：如果这个系统把 `acct=6065` 换成了 `acct=8f3a...`（一串随机字符），漏洞就没了吗？

预设回答：

- 答"没了，猜不到"——只解决了一半：猜不到，但**泄漏之后照样能用**（比如从聊天记录、截图、日志里拿到）。

- 答"还能从别的接口拿到别人的 ID"——很关键，很多系统在列表页、消息通知里就把别人的 ID 发出去了。

- 答"所以还是要做归属校验"——正确。随机化只是提高门槛，**授权才是根治**。

- 收口：安全设计的原则是**不依赖"攻击者不知道"，而依赖"知道了也用不了"**。

易错点 / 考点：① 这一页的结论常作为案例题：给一个 URL，问存在什么漏洞、怎么测、怎么修；② 答题三件套：**漏洞名称（直接对象引用/水平越权）、验证方法（换 ID 重放）、修复方案（服务端归属校验＋不可预测标识）**；③ 易错点：只说"ID 要加密"——加密后的 ID 仍然可以被重放，归属校验不能省。

过渡：对象级权限漏了会怎样？功能级权限漏了会怎样？下一页是后者。

【7.98】功能级权限控制缺失（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

功能级权限控制缺失。上一页讲的是"数据级"——你能不能看这一条；这一页讲的是"功能级"——你能不能调这个接口。层级不一样，但危害一样大。

课件先说了原因，这句话很值得抄下来：**权限控制一般是写在代码中或者通过程序的配置文件来完成，但开发人员容易忘记添加一些功能的权限控制代码。**注意"容易忘记"四个字——这不是设计能力问题，是**人的注意力问题**。一个系统几十上百个接口，权限判断散落在每个接口的开头，忘一个就漏一个。

课件给的例子：一个本该只有 admin 才能访问的链接 `http://example.com/app/admin_getappInfo`，非 admin 用户可以直接在浏览器里访问，说明网站存在功能级权限控制漏洞。

为什么"直接在浏览器里访问"就能成功？因为**前端隐藏菜单不是权限控制**。很多系统的做法是：登录后根据角色渲染菜单，普通用户看不到"系统管理"这一项。看起来安全了，其实只是"没给你入口"。攻击者根本不需要菜单，他只要知道 URL，或者从 JavaScript 文件里把接口清单翻出来，直接请求就行。

所以这一类漏洞的测试方法非常明确：**先把接口清单拿到**（看前端 JS、看接口文档、看抓包记录），**然后用低权限账号逐个请求**，看哪些返回了 200 和真实数据。返回 200 的那些，如果本该只有管理员能调，就是漏洞。

防御怎么做？三条。

第一，**权限校验必须在服务端，且在每个受保护接口上**。前端隐藏只是体验。

第二，**别在每个接口里手写 if 判断**——那样一定会漏。正确做法是**统一拦截**：用拦截器、过滤器、注解或网关，把"这个接口需要什么权限"声明在配置里，由一个地方统一判。

第三，**默认拒绝**。新加的接口如果没声明权限，默认谁都不能访问，逼着开发者去声明。这条叫"安全默认值"，是安全设计里最重要的一条原则。

一句话：**功能级权限的关键不是"写对"，而是"不可能写漏"。**

> **板书：**写"前端隐藏 ≠ 权限控制"；写漏洞示例 URL；右侧写防御三条：服务端逐接口校验・统一拦截（拦截器/注解/网关）・默认拒绝。下方写测试法：拿接口清单 → 低权限账号逐个请求 → 看 200。

> **提问：**为什么"每个接口自己写 if 判断权限"这种做法，长期看一定会出漏洞？

> **预设回答：**

- 答"人会忘"——就是课件说的那句"容易忘记"，这是最真实的原因。

- 答"新人不知道规矩"——对，人员流动后，新加的接口没人提醒。

- 答"代码审查能补上"——能补一部分，但评审也是人做的，同样会漏。

- 收口：所以工程上的解法是**把"要记住的事"变成"忘不掉的结构"**——统一拦截 + 默认拒绝。这就是"用设计解决问题，而不是用纪律解决问题"。

> **易错点 / 考点：**① 功能级权限与数据级权限（直接对象引用）的区别是高频考点：**前者是"能不能调这个功能"，后者是"能不能碰这条数据"**；② 常考判断"前端菜单隐藏了就安全了"——错；③ 简答必答"服务端统一拦截 + 默认拒绝"，只写"加权限判断"不够。

> **过渡：**讲了这么多漏洞类型，接下来该看工具了。工具分两类：写代码时用的和打系统时用的。

【7.99】安全性测试工具（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是安全性测试工具的总入口，它不列具体工具，而是先建立一个分类框架。我给大家一个四象限，你可以把后面见到的所有工具都往里放。
>
> 第一个维度：**看代码还是跑系统**。看代码的叫**静态分析**（SAST），跑系统的叫**动态分析**（DAST）。前者不需要运行程序，靠解析源码或字节码找缺陷；后者要在运行中的系统上发请求、看响应。
>
> 第二个维度：**找已知还是找未知**。找已知的是**模式匹配**——拿漏洞特征库去比对，快但漏报多；找未知的是**基于模型或形式化**的方法——建立程序的行为模型，再验证安全属性，慢但能发现深的问题。
>
> 这四个格子对应四类工具：
>
> **静态＋已知**：词汇语法分析类工具，模式匹配找可疑函数调用。
>
> **静态＋未知**：约束分析、基于模型的工具，分析数据流和控制流。
>
> **动态＋已知**：漏洞扫描器、Web 扫描器，发已知的攻击载荷。

动态＋未知：模糊测试（Fuzzing）、渗透测试，靠变异和人的创造力。

为什么先讲框架？因为学生最容易犯的错是"以为装个扫描器就等于做了安全测试"。扫描器只覆盖"动态＋已知"那一格——已知模式的动态检测。它扫不出业务逻辑漏洞，扫不出需要多步交互才能触发的漏洞，也理解不了你的权限模型。

还有一个实践建议：**SAST 放左边（开发阶段），DAST 放右边（测试阶段）**。左边早、便宜、误报多；右边晚、真实、覆盖面受用例限制。两者不是替代关系，是接力关系。

板书：画 2×2 矩阵，横轴"静态/动态"，纵轴"已知/未知"；四格分别填"词汇语法分析（ITS4 等）｜约束分析与模型检测（SLAM 等）｜Web 扫描器（Nikto/ZAP）｜模糊测试与渗透测试"；下方写"SAST 靠左，DAST 靠右，接力不替代"。

提问：如果一个团队只有预算买一个工具，你会选静态分析还是动态扫描？依据是什么？

预设回答：

- 答"静态，因为早"——有道理，越早发现越便宜，而且能覆盖到没被用例走到的代码。

- 答"动态，因为能看到真实危害"——也有道理，动态扫描的误报通常更容易判断。

- 答"看团队成熟度"——这是最专业的答法：如果代码规范、CI 完善，静态能发挥最大价值；如果系统已经上线、历史包袱重，动态扫描能立刻找到暴露面。

- 收口：没有标准答案，但答题要给出**判断依据**，而不是"我觉得"。

易错点 / 考点：① SAST 与 DAST 的区别是必考点：**是否需要运行程序、能否定位到行、误报率高低**；② 易错点：把"渗透测试"归到静态类——它是动态的，而且要人工；③ 常考判断"静态分析工具能发现运行时才出现的漏洞"——错，那正是它的盲区。

过渡：四象限里最能体现"技术在进步"的一格，是静态分析。它有三代技术，课件分得很清楚。

【7.100】静态分析工具（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把静态分析工具分成三类，正好对应三代技术演进：**词汇语法分析、约束分析、基于模型**。三代的能力和局限差别很大，考试很喜欢考这个层次。

第一类，**词汇语法分析类工具**。原理是"通过模式匹配找出特定的函数中可能导致安全问题的缺陷"。说白了就是查关键词：看到 `strcpy`、`gets`、`sprintf` 就报"可能有缓冲区溢出"；看到 `system(` 就报"可能有命令注入"。课件点了它的特点：**使用简单、快速，但误报率比较高、很难检测出比较复杂的安全问题。**为什么误报高？因为它不知道上下文——`strcpy` 的源字符串如果是常量，那根本没问题，可它看不懂。代表工具有 ITS4、Flawfinder、RATS、Pscan。

第二类，**约束分析类工具**。原理是"更好地分析函数的安全性，通过对解析树的分析，确定缺陷的位置"。它比第一代进了一步：会做数据流分析，追踪"这个变量从哪来、经过什么处理、到哪去"。所以它能判断"这个参数确实来自用户输入，且没有做过长度检查"。局限也很明确：**检查缺陷范围比较有限**——它擅长某一类特定属性（比如数组越界、空指针、格式化字符串），不太擅长跨模块的复杂问题。代表工具有 BOON、CQUAL、Splint、UNO、ESC/JAVA。

第三类，**基于模型测试的工具**。原理是"构建程序的有限状态模型，使用形式化方法表示需要检查的安全属性，设计分析算法检测模型中存在的缺陷"。这一代把程序抽象成状态机，把"不能出现什么"写成形式化属性，然后用算法去搜。能力强、能发现深层的时序类问题，代价是建模成本高、状态爆炸。代表工具有 SLAM、BLAST、MOPS、ESP、FindBugs。

三代的关系不是替代，而是**分工**：第一代适合快速筛查大量代码，第二代适合精确检查某几类缺陷，第三代适合验证关键模块的严格属性。实际项目里通常三代混用，再加一层"人工确认"。

最后提醒一句：静态分析的输出必须人工过一遍。报告里十条问题，可能有七条是误报——但剩下三条，往往是你用测试用例一辈子也碰不到的。

> **板书：**三代竖排，每代写"原理 → 特点 → 代表工具"：词汇语法（模式匹配／快但误报高／ITS4・Flawfinder・RATS・Pscan）｜约束分析（解析树＋数据流／精确但范围窄／BOON・CQUAL・Splint・UNO・ESC/JAVA）｜基于模型（有限状态＋形式化属性／深但成本高／SLAM・BLAST・MOPS・ESP・FindBugs）。

提问：为什么第一代工具看到 `strcpy` 就报警，误报会那么高？

预设回答：

- 答"因为它不知道参数是什么"——对，源是不是常量、目标缓冲区多大、有没有先检查长度，它都不看。

- 答"因为它只看函数名"——准确，这叫"没有上下文敏感"。

- 答"所以要多报，宁可错杀"——这是这类工具的设计取舍：**用高误报换低漏报**，让人的注意力去筛。

- 收口：所以评估静态分析工具，不能只看"报了多少"，要看**准确率**和**能不能定位到行**。

> **易错点 / 考点：**① 三类的名称、原理、代表工具是标准考点，尤其要记住 **ITS4/Flawfinder/RATS/Pscan 属第一代、FindBugs 属基于模型**（这点最容易记错）；② 常考判断"静态分析误报率低"——错，第一代误报很高；③ 简答题要写出"静态分析不能替代动态测试，两者互补"。

> **过渡：**理论工具分类讲完，看一份具体的工具清单——这些是工程里真会用到的。

【7.101】常见的安全漏洞检测工具（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给了一份工具清单，十一个，我按用途分成四组，这样你记起来有结构。

第一组，**代码级静态检查**：CppCheck、FindBugs、C++Test／Jtest、Klocwork Insight。这四个都是看代码的。CppCheck 针对 C/C++，FindBugs 针对 Java 字节码，C++Test／Jtest 是 Parasoft 的商业套件，除了静态检查还能生成单元测试，Klocwork 偏向大型 C/C++ 工程的深度分析。**选型看语言，看是否需要集成到 CI。**

第二组，**渗透与攻击平台**：Metasploit、BackTrack5。Metasploit 是漏洞利用框架，把"发现漏洞 → 选择载荷 → 建立会话"标准化了；BackTrack5 是早期的渗透测试专用发行版（后来演化为 Kali）。这一组属于"进攻方工具"，学习时务必只在靶场里用。

第三组，**Web 应用测试**：Burp Suite、Nikto Web Scanner、W3af、ZAP。Burp 是代理型手工测试平台，Nikto 是老牌 Web 服务器扫描器，W3af 是"Web 应用攻击与审计框架"，ZAP 是 OWASP 的旗舰开源项目。这四个是 Web 安全测试的主力，后面几页会专门讲 Burp 和 W3af。

第四组，**企业级平台**：Fortify SSC。它不是单机工具，是软件安全中心，负责把扫描结果汇总、去重、定级、跟踪修复，通常和 CI/CD 集成，管的是"整个组织的漏洞治理"。

怎么看这份清单？我给你三句话。

第一，**工具是分层的**：代码层、应用层、平台层，各管一段。

第二，**开源与商业各有位置**：开源灵活、可定制、适合学习和中小项目；商业工具规则库全、支持好、适合合规场景。

第三，**工具解决"发现"，人解决"判断"**。同一份报告，有经验的人能看出哪三条是真的，没经验的人只会把报告转发给开发。

课件在这里留了一句"详见我之前写的文章"，说明这份清单是作者实践经验的汇总，不是抄来的名录。

板书：四组并列：代码静态（CppCheck・FindBugs・C++Test/Jtest・Klocwork）｜渗透平台（Metasploit・BackTrack5）｜Web 测试（Burp Suite・Nikto・W3af・ZAP）｜企业平台（Fortify SSC）。下方写"工具分层，人做判断"。

提问：如果一个团队已经在用 Fortify 做代码扫描，还有必要在测试阶段做 Burp 手工测试吗？

预设回答：

- 答"有必要，一个看代码一个看运行"——正是。Fortify 看不出"两个接口组合起来能越权"这种问题。

- 答"没必要，重复了"——这不成立，静态与动态的盲区不重合。

- 答"看预算"——现实考量，但即便不买商业工具，手工测关键接口也是必须的。

- 收口：静态工具告诉你"这里可能有问题"，动态测试告诉你"这里确实能被利用"。前者是线索，后者是证据。

易错点 / 考点：① 工具与用途的对应是常考连线题，特别注意 **FindBugs 是 Java、CppCheck 是 C/C++**；② 易错点：把 Metasploit 当成"扫描器"——它是**利用框架**；③ 常考简答"静态工具与动态工具的区别"，答题要落到"是否运行程序"和"能否证明可利用"。

过渡：清单里最值得花时间的是 Burp 和 W3af。但在讲它们之前，先看一张更实用的表——十大漏洞各自对应什么工具。

【7.102】十大漏洞对应的工具（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是"漏洞类型 → 工具"的对照表，非常实用。先说清楚怎么读它：**它不是一对一映射，是多对多**。同一个工具能测好几种漏洞，同一种漏洞也要好几个工具配合。所以不要死记"哪个对哪个"，要记"这一类问题用什么手段"。

先看漏洞那一列，十项：注入式攻击（注入类漏洞）、跨站点脚本 XSS、无效的认证及会话管理功能、对不安全对象的直接引用、伪造的跨站点请求 CSRF、安全设置错误、加密存储的不安全因素、不限制访问者的

URL、传输层面的保护力度不足、未经验证的重新指向及转发。

这十项你发现没有，**几乎全部和前面几页讲的内容对得上**：注入、XSS、认证与会话、直接对象引用、CSRF、URL 访问限制、未验证跳转——就是我们刚讲过的那些。剩下三项是新的：安全设置错误（配置问题）、加密存储不安全（数据保护）、传输层保护不足（信道问题）。这三项的共同点是：**它们不在业务代码里，而在配置和部署里**。这也解释了为什么"代码审计做得很认真"的系统照样会出事。

再看工具那一列：SQL Inject Me、Zed 攻击代理（ZAP）、Firefox HackBar、Burp Suite、Tamper Data／Seride、Watobo、MatriXay、Nikto／Wikto、Calomel Add-on、Watcher。

按手段归一下类：**改请求**——HackBar（在地址栏快速改参数）、Tamper Data（拦截并修改请求）、Burp Suite（全能的拦截代理）；**扫漏洞**——ZAP、Nikto／Wikito、Watobo、MatriXay；**专测 SQL 注入**——SQL Inject Me；**查 TLS 配置**——Calomel（SSL/TLS 配置检查）；**查证书与站点信息**——Watcher。

怎么用这张表？我建议的教学用法是：**先按漏洞定测试项，再按测试项选工具**。反过来——先装一堆工具再想测什么——是最常见的低效做法。

还有一句必须说：工具能覆盖的，永远只是"已知模式"。这十项里最容易被工具漏掉的，是"安全设置错误"和"加密存储不安全"——因为它们需要看配置、看数据，而不是发请求。

> **板书：**画两列对照，左侧十类漏洞（用简称），右侧按手段分组：改请求（HackBar・Tamper Data・Burp）／扫漏洞（ZAP・Nikto・Watobo・MatriXay）／专测注入（SQL Inject Me）／查 TLS（Calomel）／查证书（Watcher）。顶部写"多对多，不是一对一"。

提问：这十项里，哪几项是"扫描器基本扫不出来、必须靠人看配置"的？

预设回答：

- 答"安全设置错误"——对，比如目录列表没关、默认口令没改、错误页暴露堆栈。
- 答"加密存储的不安全因素"——对，要看密码是不是哈希、密钥存在哪，这得读代码和配置。
- 答"传输层保护不足"——这个其实有工具能测（TLS 配置扫描），但需要专门扫，普通 Web 扫描器不管。
- 收口：所以安全测试的能力结构是"**工具铺面 + 人工看配置 + 人工想业务**"三层，缺一层就有盲区。

易错点 / 考点：① 十大漏洞的清单常考多选，注意"安全设置错误""加密存储不安全""传输层保护不足"这三项容易被漏；② 易错点是硬记一对一映射——答题时写"工具与漏洞是多对多关系"更准确；③ 常考判断"有了扫描器就不需要人工测试"——错，理由要写"配置类与业务逻辑类漏洞工具覆盖不到"。

> **过渡：**这张表里的工具名，考试不要求背全，但有两个必须认得——Burp Suite 和 W3af，它们代表了两种不同的技术路线。

【7.103】Burp Suite- web security testing（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

Burp Suite 是 Web 安全测试里最常用的一套工具，它的定位不是"扫描器"，而是**一个能看、能改、能重放的中间人工作台**。课件列了它的七个组件，我一个一个讲，每个都告诉你"它解决什么测试问题"。

第一个，**Intercepting Proxy（拦截代理）**：让你检查并修改浏览器与目标应用之间的流量。这是 Burp 的核心。它把浏览器发出的每个请求都停下来，你可以看、可以改、可以放行。为什么这一步关键？因为**安全测试的本质是发出"正常用户发不出的请求"**，而浏览器会自动加很多限制——隐藏字段、禁用的按钮、前端校验。有了代理，这些限制全部失效。

第二个，**Spider（爬虫）**：抓取内容与功能，把站点里所有的链接、表单、参数都摸出来。它的产出是"攻击面清单"。

第三个，**Scanner（扫描器）**：自动化检测多种漏洞类型。这是"铺面"用的，跑完你会得到一份待确认清单。

第四个，**Intruder（入侵者）**：执行强大的定制化攻击，用来发现和利用不寻常的漏洞。说白了就是**高度可配置的批量重放**：把请求里某个位置标成变量，灌入字典，看响应差异。测越权、测爆破、测参数枚举，全靠它。

第五个，**Repeater（重放器）**：手工修改并重发单个请求。它是最常用的一个——你怀疑某个参数有注入，就在 Repeater 里改一下、发一下、看响应，反复试探。

第六个，**Sequencer（序列器）**：测试会话令牌的随机性。这一条直接对应前面讲的会话劫持——如果 Session ID 生成得不随机，就能被预测，那 HttpOnly 再好也没用。

第七个，**Extensibility（可扩展性）**：可以自己写插件，完成复杂而高度定制的任务。因为业务逻辑漏洞是通用的，必须自己写检查逻辑。

用一句话概括 Burp 的价值：**它把"猜测"变成"验证"**——你不再是"觉得这里有洞"，而是"我发了这个请求，它回了这个响应"。

> **板书：**七个组件竖排，右边各配一句用途：Proxy（拦截改包）／Spider（摸攻击面）／Scanner（铺面）／Intruder（批量定制重放）／Repeater（单请求手工试探）／Sequencer（令牌随机性）／Extensibility（自写插件）。

> **提问：**为什么说"有了拦截代理，前端的校验就等于不存在"？

> **预设回答：**

- 答"因为能直接改请求"——对，代理在浏览器和后端之间，前端做什么都拦不住你手改。

- 答"因为可以直接调接口"——也是，跳过页面，用工具构造请求。

- 答"前端校验本来就不算安全措施"——这句最准确，前端校验是**体验优化**，不是安全边界。

- 收口：所以安全测试的默认假设是"客户端完全不可信"，服务端必须自己再校验一遍。

> **易错点 / 考点：**① 七个组件的名称与用途是常考连线；② 易错点是把 Burp 等同于"漏洞扫描器"——它的核心是**代理 + 手工重放**，Scanner 只是其中一个组件；③ 常考判断"Sequencer 用来爆破密码"——错，它测的是**令牌随机性**，爆破靠 Intruder。

> **过渡：**工具的能力讲完了，我们看它在真实界面里长什么样、按什么顺序用。

【7.104】Burp Suite- web security testing（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 Burp 的操作界面，重在让你建立"工作流"的印象，而不是记按钮位置。我把一次完整的安全测试在 Burp 里的走法串一遍。
>
> 第一步，**配代理**。让浏览器流量经过 Burp，装好它的 CA 证书以便解密 HTTPS。这一步有个前提必须说清楚：**只在你自己的靶机或授权范围内的目标上做**。代理本质是中间人，用在别人的流量上就是另一回事了。
>
> 第二步，**开拦截，走一遍业务流程**。注册、登录、下单、查询、退出，每个动作都抓下来。这一步的目的不是找漏洞，而是**建立业务地图**：有哪些接口、每个接口带哪些参数、哪些参数是 ID、哪些是金额、哪些是状态。
>
> 第三步，**丢给 Scanner 铺面**。等它跑完，得到一份按风险排序的清单。注意：**扫描结果只是线索**。

第四步，**用 Repeater 逐条验证**。把可疑请求发到 Repeater，改一个参数、看一眼响应。改 ID 看越权，改金额看逻辑，去掉 Token 看鉴权，加个引号看注入。这一轮下来，清单会大幅缩水，剩下的才是真问题。

第五步，**用 Intruder 批量验证**。比如要确认 100 个用户 ID 里哪些能越权，就标好变量、灌入字典、看返回长度或状态码的差异。

第六步，**整理证据**。每条漏洞保留请求原文、响应片段、复现步骤、影响评估。这是报告的核心，也是开发愿意修的前提。

我要特别强调第三步和第四步的关系。很多同学跑完扫描器就把报告交了，结果开发一看全是误报，从此不信安全测试。**扫描器负责"找可能性"，人负责"证明确实性"**——这个分工不搞清楚，工具越用越没威信。

板书：六步流程：配代理（装 CA）→ 拦截走业务（建地图）→ Scanner 铺面 → Repeater 验证 → Intruder 批量 → 整理证据。下方写"扫描给可能性，人给确实性"。

提问：为什么"先手工走一遍业务，再开扫描器"这个顺序不能反？

预设回答：

- 答"扫描器不知道业务"——对，它不知道哪个接口是转账、哪个是查询，只能按通用规则扫。

- 答"先走业务能拿到登录态"——很实际，很多接口必须先登录才能扫到。

- 答"顺序反了会漏"——也对，扫描器爬不到需要多步交互才出现的页面。

- 收口：所以人工的角色是**给工具画地图**，工具的角色是**替人跑量**。

易错点 / 考点：① 这一页常考"Burp 的典型使用流程"，答题顺序要是"代理→爬取/走业务→扫描→重放验证→报告"；② 易错点：把"扫描结果"直接当"漏洞清单"；③ 常考判断"Burp 可以解密 HTTPS，所以能测任何站点"——技术上能，但**授权**是前提，答题要提这一句。

过渡：Burp 是手工工作台，下一页的 W3af 是另一条路线——把攻击和审计做成框架。

【7.105】W3af（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

W3af 的全名是 **Web Application Attack and Audit Framework**，Web 应用攻击与审计框架。它的思路和

Burp 不同：Burp 是"给你一套手工工具"，W3af 是"给你一条可编排的流水线"——把扫描、审计、利用串起来，用插件组合出攻击链。

课件先讲了它的**模糊测试引擎（Fuzzing engine）**，并且列出了它能注入的七个位置，这七个位置非常值得记，因为它就是 Web 输入点的完整清单：

Query string（URL 查询串）、**POST-data**（表单体）、**Headers**（请求头）、**Cookie values**（Cookie 值）、**Multipart/form file content**（多部分表单里的文件内容）、**URL filename**（URL 里的文件名）、**URL path**（URL 路径本身）。

为什么这七个位置重要？因为**只要一个位置进了服务端并且被使用，它就是潜在的注入点**。学生做测试时最容易只盯着查询串和表单，漏掉 Header（比如 `X-Forwarded-For`、`Referer`、`User-Agent` 被直接写进日志或 SQL）、漏掉 Cookie（很多系统拿 Cookie 里的字段直接做权限判断）、漏掉文件名（上传后的重命名、路径拼接）。

第二块讲它的**代理服务器实现**：它内置了 Web／代理服务器，便于集成到你自己的代码里，用来识别并利用漏洞。翻成教学语言：**它不只是个界面工具，还能当库用**，可以把扫描能力嵌到自己的流水线里。

第三块是 **Fast HTTP Client**，也就是高性能 HTTP 客户端，列了一串能力：以极快速度发送特制 HTTP 请求、代理支持、HTTP Basic/Digest 认证、伪造 User-Agent、添加自定义请求头、Cookie 处理、HTTP 响应处理、DNS 缓存、多部分文件上传。

这一串能力看着琐碎，其实每一行都对应一个测试需求：**伪造 UA** 是为了测移动端和爬虫的不同逻辑；**自定义请求头**是为了测 Header 注入和绕过；**DNS 缓存**是为了压测时不被打爆；**多部分上传**是为了测文件上传漏洞。所以别把它当参数表背，要当"测试动作清单"看。

> **板书：**三块：Fuzzing 七注入点（Query string／POST-data／Headers／Cookie／Multipart file／URL filename／URL path）｜代理服务器（可嵌入代码）｜Fast HTTP Client（速度・代理・认证・UA 伪造・自定义头・Cookie・DNS 缓存・多部分上传）。

提问：七个注入点里，你最可能在做课程设计时漏掉哪一个？漏掉会有什么后果？

预设回答：

- 答"Header"——最常见。`X-Forwarded-For` 被直接拼进 SQL 或写进权限判断，是真实存在的一类漏洞。

- 答"Cookie"——也很常见，很多系统把用户 ID、角色放在 Cookie 里还做明文判断。

- 答"URL filename / path"——容易漏，尤其是下载接口和静态资源路由。

- 收口：判断方法很简单——**凡是服务端会读的地方，都在清单里**；先从代码里搜"取请求数据"的 API，再对照这七项。

> **易错点 / 考点：** ① W3af 全称（Web Application Attack and Audit Framework）常考；② 七个注入点是很好的多选题素材，注意 "URL filename" 与 "URL path" 是两个不同项；③ 易错点：把 W3af 只当扫描器——它同时提供**框架/库**能力；④ 常考简答"Burp 与 W3af 的定位差异"：Burp 偏手工交互，W3af 偏框架化自动编排。

> **过渡：** 安全性测试这一节到这里讲完了。我们花一页把整节的方法盘一遍。

【7.106】本讲小结（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是安全性测试的方法清单，十项，我按"从早到晚、从静到动"的顺序重排一遍，你会看到一条清晰的能力阶梯。
>
> **基于模型的安全功能测试**：先给系统建一个行为模型，再验证安全属性在这个模型上是否成立。它回答的是"设计层面允不允许坏事情发生"。
>
> **基于故障注入的安全性测试**：人为制造故障——断网、超时、返回错误、篡改数据——看系统会不会因此失守。它测的是**异常路径上的安全性**，因为大量漏洞只在异常分支里出现。
>
> **模糊测试（Fuzz Testing）**：用变异出来的畸形输入去轰接口，看会不会崩、会不会执行、会不会泄漏。它不需要你先知道漏洞长什么样，这是它最大的价值。
>
> **基于白盒的安全测试**：拿到代码，做数据流分析、污点分析，追踪"用户输入有没有不加处理地流到危险函数"。
>
> **威胁模型与攻击树理论**：不是测试执行，而是**测试设计**——把"攻击者想达到什么"拆成一条条路径，每条路径就是一个测试场景。
>
> **形式化安全测试方法**：用数学方法描述安全属性并证明。成本高，用在最关键的模块上。
>
> **语法测试**：针对解析器——JSON、XML、协议报文、文件格式——用合法但异常的语法结构去测。前面讲的

XML 注入就属于这一类的相邻问题。

基于属性的测试：先声明"任何情况下都必须成立的性质"（比如"金额不能为负""一个人只能有一张券"），再自动生成数据去试图推翻它。

动态污点分析方法：给外部输入打"污点"标记，跟踪它在程序里的流动，看它有没有流到敏感操作——这是工业界查注入最有效的手段之一。

基于风险的安全测试：按风险分配资源，高风险的模块多测、深测。

这十项你不用全用，但要能说出**各自解决什么问题**。考场上给你一个场景，你要能选出两三种组合。

板书：十项分三组：设计层（威胁模型与攻击树・基于模型・形式化）｜实现层（白盒・动态污点・语法测试）｜运行层（故障注入・模糊测试・基于属性・基于风险）。右侧写"选方法的依据＝风险＋阶段＋可见性"。

提问：如果只允许你选两种方法去测一个已经上线、源码拿不到的系统，你选哪两种？

预设回答：

- 答"模糊测试＋基于风险的安全测试"——很合理：一个负责用畸形输入轰，一个负责把力气压在高风险接口上。

- 答"渗透测试＋故障注入"——也可以，但要说明"渗透"其实是模糊与手工的结合。

- 答"白盒＋污点分析"——源码拿不到，这两项做不了，说明审题很关键。

- 收口：选方法的第一约束不是"哪个先进"，而是**你能拿到什么**（源码、日志、设计文档、测试环境）。

易错点 / 考点：① 十项名称常考多选与填空，注意"动态污点分析"和"基于属性的测试"这两项容易被忘；② 易错点：把"威胁模型"当成一种测试执行技术——它是**设计与分析**活动；③ 简答题要求"针对某场景选方法并说明理由"，答题结构：场景约束 → 选 2–3 种 → 各自解决什么。

过渡：方法讲完了，最后留两页推荐——不是推荐工具，而是推荐"接下来该往哪里学"。

【7.107】特别推荐（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是作者给的学习推荐，属于"课后自己走的路"。我按"先打底、再动手、最后看格局"的顺序给大家一

个读法。

先打底：安全测试不像功能测试，它的知识面是"跨层"的——你要懂 HTTP 协议、懂浏览器同源策略、懂数据库权限模型、懂操作系统权限。这些东西不补，工具会用但不知道为什么。建议的底子是：HTTP 报文结构、Cookie 与 Session 机制、SQL 基础、Linux 权限。

再动手：安全这东西光看书没用，一定要有靶场。课件这一节推荐的资源，本质上都是让你在**合法可控的环境**里把攻击链走一遍。走一遍之后，你对"输入校验""输出编码"这些词的理解会完全不同——因为你见过不加会怎样。

最后看格局：看业界每年发布的漏洞榜单和真实事件复盘。你会慢慢发现一个规律：**排在前面的漏洞类型，十几年没怎么变**——注入、XSS、失效的访问控制、配置错误。技术在变，人在同一个地方反复摔跤。

我要在这里补一句课程要求：**所有这些练习，只在自己的靶机、自己的测试库、自己搭的环境里做**。这不是保守，这是这个行业的底线——你的能力越强，越需要一个清晰的边界感，否则一次"试试看"就可能变成别人的损失。

还有一条学习建议：每学一个攻击手法，强迫自己写三条东西——**原理一句话、防御一句话、怎么测一句话**。这三条写不出来，说明还没学会。

> **板书：**三层：打底（HTTP・Cookie/Session・SQL・Linux 权限）→ 动手（靶场走一遍攻击链）→ 看格局（漏洞榜单＋事件复盘）。右下写"每学一个手法，写三句：原理／防御／怎么测"。

提问：为什么"排在前面的漏洞类型十几年没怎么变"这件事，对测试人员来说是好消息还是坏消息？

预设回答：

- 答"好消息，说明经验可复用"——是一面，检查单确实可以沉淀下来反复用。
- 答"坏消息，说明大家没吸取教训"——也是事实，同样的错误一代代重复。
- 答"对个人是好消息，对行业是坏消息"——这个回答最完整。
- 收口：所以把检查单做成**流程里绕不过去的一步**，比指望每个人自觉更有效。

易错点 / 考点：这一页不直接出题，但常作为**课程设计选题**的引导。注意区分"学习资源推荐"和"考试知识点"——考的是方法名与原理，不是资源清单。易错点：把推荐当成"必须装这些工具"，工具只是手段。

过渡：推荐还有第二页，讲的是更偏工程体系的那一层。

【7.108】特别推荐 2（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页接着推荐，方向从"学什么"转到"怎么把安全放进工程里"。三个层次，我按从个人到组织的顺序讲。

第一层，**规范与标准**。安全测试不能靠个人灵感，要靠清单。业界的漏洞榜单给了"该防什么"，安全编码规范给了"该怎么写"，测试指南给了"该怎么验"。这三样东西的共同作用是：**把专家的判断固化成普通人也能执行的步骤**。

第二层，**工具链集成**。安全要被真正执行，必须嵌进研发流水线：提交代码时跑静态扫描，构建镜像时查依赖漏洞，部署前跑一次动态扫描，上线后持续监控。这叫 DevSecOps——安全不是最后一个关卡，而是每一道关卡上的一小步。判断标准很朴素：**如果安全检查需要人专门去点一下，它迟早会被跳过**。

第三层，**组织与责任**。谁对安全负责？答案不是"安全部门"，而是"每个提交代码的人 + 每个设计架构的人 + 每个做发布决策的人"。安全团队的角色是提供工具、制定标准、做应急响应，不是替所有人做检查。

我给同学们一个可以立刻用上的建议：**在你的课程设计里，加一份"安全检查单"**，十到十五条，每条写清"检查什么、怎么查、不合格怎么改"。这份单子比任何工具都值钱，因为它是你自己的知识沉淀，下次做项目可以直接复用。

最后一句话：安全不是一门课，是一种**工作习惯**——每写一个接口，问一句"如果别人传了不该传的参数会怎样"。

> **板书：**三层：规范与标准（榜单・编码规范・测试指南）→ 工具链集成（提交扫描・依赖检查・部署前扫描・上线监控＝DevSecOps）→ 组织与责任（人人负责，安全团队做标准与应急）。下方写"要人专门点一下的检查，迟早被跳过"。

提问：为什么"需要人专门去点的安全检查"最后一定会被跳过？

预设回答：

- 答"因为赶进度"——最真实的原因，发布压力下第一个被砍的就是"额外步骤"。

- 答"因为没人为它负责"——也对，没人负责的事就没人做。

- 答"所以要自动化"——正解，把它变成流水线里必过的一步，不过就发不出去。

- 收口：这就是"用机制保证，而不是用自觉保证"，和第 4 章讲的流程规范是一回事。

> **易错点 / 考点：**这一页的考点在"DevSecOps 的含义"和"安全责任归属"。易错点：认为安全是安全部门的事——答题要写"全员、全流程"。另外注意区分"规范"（写什么）与"工具"（怎么查），两者互补。

> **过渡：**安全性测试这一节到此结束。下面进入 7.3，一个听起来简单、做起来极容易漏的专项——兼容性测试。

【7.109】7.3 兼容性测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

进入 7.3 兼容性测试。先讲清楚它为什么值得单独成一节，因为很多人觉得"兼容性"就是"多试几个浏览器"。

兼容性测试要回答的问题是：**同一个软件，换一个环境还对不对？**这里的"环境"有四个维度：**硬件**（不同机型、不同外设、不同配置）、**软件系统之间**（操作系统、中间件、数据库、第三方组件、客户端与服务端的版本组合）、**数据**（老版本产生的数据、别的系统产生的数据）、**软件环境组合**（浏览器 + 操作系统 + 分辨率 + 语言 + 网络）。

为什么它容易出事？因为**开发环境是单一的，用户环境是发散的**。开发机上跑得好好的，用户那边可能是三年前的安卓机、可能是 IE 内核的浏览器、可能是没有管理员权限的电脑、可能装着一个改了默认行为的输入法。

兼容性问题的代价也很特别：**它不影响功能正确性，只影响"某些用户用不了"**。这类问题在测试报告里往往不显眼——"某机型闪退"看起来像小问题，但对那批用户来说就是 100% 不可用。而且它极难复现，开发和测试经常因为"我这儿没问题"吵起来。

这一节的讲法我按课件的顺序：先看兼容性问题和升级失败的典型现象，再看软硬件兼容、系统间兼容、数据兼容三条主线，最后讲环境组合怎么用组合测试的方法来压缩用例量。

请带着一个问题听：**你们做过的项目，兼容性测试是怎么确定"测哪些环境"的？是拍脑袋，还是有依据？**

> **板书：**中间写"兼容性＝换环境还对不对"；四个维度：硬件｜系统之间｜数据｜环境组合。右侧写"开发环境单一 vs 用户环境发散"，下方写"代价：不影响功能，只让部分用户用不了"。

> **提问：**为什么兼容性缺陷在缺陷报告里常常被低估？

预设回答：

- 答"因为只影响少数机型"——正是这种想法导致它被降级，但对那批用户是 100%。

- 答"因为难复现"——对，复现不了就容易被判"无法重现"而关闭。

- 答"因为不影响主流程"——要小心这个判断，支付页面在某个浏览器上白屏，就是主流程断了。

- 收口：评价兼容性缺陷，要看**受影响用户的比例 × 该功能的业务重要性**，而不是看"看起来严重不严重"。

易错点 / 考点：兼容性测试的四个维度常考多选。易错点：把兼容性测试等同于"浏览器兼容性测试"——那只是一个子集。常考判断"兼容性问题不影响功能正确性，所以优先级低"——错，要看受影响用户比例与业务重要性。

过渡：先看现实中最常见的两种现象：一升级就出问题，一换环境就不对。

【7.110】碰到兼容性问题、升级失败（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲的是兼容性问题最常见的两种表现形态：**升级失败**，和升级之后的**兼容性问题**。它们经常同时出现，但根因不同。

升级失败指的是升级动作本身没完成：安装包装不上、数据库脚本执行报错、配置文件格式不认、依赖的组件版本不匹配、升级到一半中断导致系统处于半新半旧的状态。最麻烦的是最后一种——**部分模块升级成功、部分失败**，系统还能启动，但行为已经不一致了。

升级后的兼容性问题指的是升级成功了，但和别人对不上了：新版本客户端连不上老版本服务端、新版本写出的数据老版本读不了、新版本依赖的中间件老服务器上没有、第三方接口的协议变了。

这两类问题的共同根源是：**升级不是替换一个文件，而是同时改变了四样东西**——代码、数据格式、接口协议、运行环境。任何一样没对齐，就会出问题。

所以升级测试的检查清单应该是四问：**旧数据能不能被新版本正确读取？新写的数据老版本还能不能用（如果需要回滚）？接口协议变了吗，调用方都同步了吗？依赖的组件版本在目标环境里有吗？**

再补一条工程实践：**升级必须可回滚**。上线前要验证"回滚脚本能跑通、回滚后数据不丢"。因为升级出问题时，业务等不了你排查，先回滚止损才是正确顺序。

这一页没有复杂理论，但它对应的是真实事故里最高发的一类。**升级失败往往不是技术难题，而是清单没列全。**

板书：两块对照：升级失败（装不上・脚本报错・配置不认・依赖不匹配・半新半旧）｜升级后不兼容（客户端/服务端不匹配・数据读不了・中间件缺失・协议变更）。下方四问清单＋"必须可回滚"。

提问：为什么"升级到一半中断"比"升级彻底失败"更危险?

预设回答：

- 答"因为系统还能跑，大家以为没事"——正是，这种"看起来正常"最危险。

- 答"因为数据可能一半新一半旧"——对，数据不一致比服务不可用更难修。

- 答"因为回滚也回不干净"——很关键，部分模块已经写了新格式数据，回滚会读不了。

- 收口：所以升级要做**幂等＋事务化**：要么全成功，要么能干净退回，中间状态必须能被检测出来。

易错点 / 考点：① 要能区分"升级失败"与"升级后兼容性问题"，前者是过程失败，后者是结果不匹配；② 常考简答"升级测试要检查什么"，答题按四问展开（旧数据・新数据・协议・依赖）；③ 易错点：忽略"回滚验证"，这是升级测试里最常被漏掉的一项。

过渡：升级问题常常落在硬件这一层。我们先看软件与硬件的兼容。

【7.111】软件与硬件兼容性问题（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲软件与硬件的兼容性。很多人以为这一块只跟"驱动"有关，其实它覆盖的范围很宽，我按从上到下的顺序讲五层。

第一层，机型与硬件配置。同一个 App，在不同 CPU 架构（ARM／x86）、不同内存大小、不同屏幕分辨率、不同传感器配置的机器上，表现可能不同。低内存机型上被系统杀掉后台、小屏机型上按钮被挤掉、没有某个传感器的机型上功能直接崩——这些都是兼容性问题。

第二层，外设与接口。打印机、扫码枪、读卡器、摄像头、USB 设备、串口设备。这一层的问题特别多，因为外设的驱动行为差异很大，而且往往不在开发者的测试环境里。

第三层，驱动与固件。显卡驱动版本不同，渲染结果可能不同；固件版本不同，同一个指令的行为可能不同。这一层最典型的症状是"某批机器上画面撕裂/花屏/闪退"。

第四层，资源与性能边界。磁盘空间不足、内存不足、电量低、温度高时的降频——这些"硬件状态"也会触发兼容性问题，因为它们会改变软件的运行条件。

第五层，虚拟化与云环境。虚拟机、容器、云主机的硬件是"抽象"的，某些依赖特定硬件特性的功能（GPU加速、硬件加密、特定指令集）可能不可用。

怎么测？三条务实做法。**一是按市场占有率选机型**，别按"我手上有哪些机器"选。**二是抓关键路径**，安装、启动、登录、核心业务、退出，每条路径在低端机和高端机各跑一遍。**三是建立机型矩阵并沉淀下来**，每次发版复用，别每次重新讨论。

一句话：**硬件兼容性的关键不是"测多少台"，而是"测对哪几台"。**

> **板书：**五层：机型配置｜外设接口｜驱动固件｜资源与性能边界｜虚拟化与云。下方三条做法：按市场占有率选机型・抓关键路径・机型矩阵沉淀。

提问：如果预算只够买三台测试机，你会怎么选？

预设回答：

- 答"一台低端、一台主流、一台旗舰"——很实用，覆盖性能边界的两端和中间。
- 答"按用户机型分布选覆盖最多的"——更专业，先看数据再买机器。
- 答"买最新最好的"——方向错了，兼容性问题恰恰多发在老设备和低配设备上。
- 收口：选机的依据是**用户分布 × 业务关键度**，不是"配置越高越能发现问题"。

易错点 / 考点：① 硬件兼容的五个层次常考多选，注意"资源与性能边界""虚拟化"这两层容易被漏；② 易错点：认为"配置越高越能测出兼容性问题"——低配和异常状态才是高发区；③ 常考简答"如何选择测试机型"，答题要提"市场占有率数据"和"关键路径"。

过渡：硬件之上是软件。软件系统之间的兼容，课件用两张图分别讲了服务端和客户端。

【7.112】软件系统之间的兼容性 -1（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲服务端这一侧的兼容性。它给的是一个典型的三层堆栈，从上往下依次是**应用软件**、**中间件/OS**、**第三方软件**（比如数据库）。层与层之间都标着"兼容"，还标了两个"upgrade"箭头，旁边是具体的版本对：Red Hat Enterprise Linux 7.1 与 6.6，MySQL Enterprise Edition 5.7 与 5.6。

这张图要你看出的第一件事：**兼容性不是一个点，是一条链**。你的应用依赖中间件，中间件依赖操作系统，应用还直接依赖数据库。链条上任何一环换了版本，整条链的兼容关系都要重新验证。

第二件事：**升级方向是不对称的**。课件画的是 RHEL 6.6 升到 7.1、MySQL 5.6 升到 5.7。往上走的时候，你要回答两个问题：**新版本还认不认老版本的配置和数据？我的应用在新版本上还能不能跑？**这两个问题的答案经常不一样——数据库升上去了，但应用用的驱动还是老版本，照样连不上。

第三件事：**兼容性有方向性**，这就是后面小结要讲的"向后兼容"和"向前兼容"。向后兼容指的是新版本能处理老版本的东西；向前兼容指的是老版本能处理新版本的东西。工程上通常只保证向后兼容——**新版本要认旧数据，但旧版本不保证认新数据**。所以升级前必须确认"回滚窗口内，老版本还能读新数据吗"。

具体测什么？我给四条：**安装与启动**（在目标 OS 版本上能否正常安装、依赖是否齐全）、**接口与协议**（中间件、数据库驱动版本是否匹配）、**配置兼容**（老配置文件在新版本是否仍被识别）、**性能与行为**（新版本默认参数变了没有，比如连接池大小、字符集、超时时间）。

最后一条最容易被忽略：**默认值变了，也是一种不兼容**。代码没改，行为变了，这种问题最难查。

> **板书：**画三层堆栈（应用／中间件・OS／第三方软件），层间写"兼容"；右侧写版本对：RHEL 6.6 → 7.1、MySQL 5.6 → 5.7；下方四测：安装启动・接口协议・配置兼容・性能与默认值。

> **提问：**为什么"数据库升级成功、应用却连不上"这种事故很常见？

> **预设回答：**

- 答"因为驱动版本没跟着升"——最常见的原因，JDBC/ODBC 驱动不匹配。
- 答"因为认证方式变了"——很专业，新版本数据库常改默认认证插件。
- 答"因为字符集或时区默认值变了"——也对，这类"默认值变更"极难排查。
- 收口：所以升级验证要覆盖**整条链**，而不是只验证被升级的那一个组件。

易错点 / 考点： ① 三层堆栈与"兼容是链式"的结论常考；② 版本对（RHEL 7.1/6.6、MySQL 5.7/5.6）是课件给的例子，答题可引用；③ 易错点：把"组件升级成功"等同于"系统兼容"；④ 常考判断"兼容性只需要验证新版本"——错，回滚场景下老版本也要验证。

过渡： 服务端是"往上叠"的兼容，客户端是"多版本并存"的兼容。这两种问题的性质完全不同。

【7.113】软件系统之间的兼容性 -2（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲客户端侧的兼容性，它跟服务端那张图的最大区别是：**客户端不是一套，而是好几套同时在用**。图上画了三个版本——Ver 2.1、Ver 2.2、Ver 3.0，每一个都要和中间件/OS 兼容，也都要和第三方软件兼容，同时它们之间还要"兼容"。

这里的"兼容"有两层含义。

第一层，**同一个服务端要同时服务多个客户端版本**。这是真实世界的常态：用户不会因为你们发了新版本就立刻升级，App Store 上有历史版本，企业内网有锁版本，老旧设备装不了新版。所以服务端必须做到"**新老客户端都能用**"——这叫服务端对客户端的向后兼容。

第二层，**不同版本的客户端之间的数据要能互通**。比如 A 用 2.1 发了一条消息，B 用 3.0 能不能正确显示？3.0 新增的字段，2.1 收到会不会崩？这就是"跨版本数据兼容"，也是下一页要讲的数据兼容问题的一个具体场景。

由此引出两个工程要求。

第一，**接口设计要能演进**。新增字段要可选、不能因为多了字段就报错；删除字段要先停止使用、再下线；枚举值要允许"未知值"存在。这些规则合起来叫"健壮性原则"——**发送要保守，接收要宽容**。

第二，**要定义版本支持策略**。同时支持几个大版本？最低支持到哪一版？老版本什么时候强制升级？这些问题如果没有明确答案，测试就没法确定"测哪些组合"。

一句话总结这一页：**服务端的兼容是"往上长"，客户端的兼容是"横向铺"**。前者要防升级踩坑，后者要防版本碎片。

板书： 画三个版本框（2.1／2.2／3.0）横向排列，每个都连到"中间件/OS"与"第三方软件"；下方写两层含

> 义：多版本同时服务·跨版本数据互通；再写两条要求：接口可演进（发送保守／接收宽容）·版本支持策略。

> **提问：**"接收要宽容"具体到接口设计上，你会怎么做？

> **预设回答：**

- 答"多余字段忽略不报错"——对，这是最基本的一条。

- 答"枚举遇到未知值不能崩"——很关键，新增枚举值是最常见的破坏性变更。

- 答"字段缺失时用默认值"——可以，但要小心默认值本身带来语义错误。

- 收口：三条合起来就是"**结构可扩展、缺省有定义、未知不致命**"。

> **易错点 / 考点：**① 这一页常与上一页对比出题："服务端兼容与客户端兼容的区别"——一个是纵向升级链，一个是横向多版本并存；② 易错点：认为"用户会及时升级"——现实中版本碎片长期存在；③ 常考简答"接口演进要注意什么"，答题要提"新增可选字段、未知枚举值容错、版本策略"。

> **过渡：**版本兼容说到底是"代码之间"的事，而数据兼容是"数据之间"的事——它更容易被忽略，也更难修。

【7.114】最常见的数据兼容性问题？（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页用一个设问开头：最常见的数据兼容性问题是什么？课件给了三个现象，都很朴素，但每个都对应一大类故障。
>
> **第一个现象：文件打不开。**老版本生成的文件，新版本打不开；或者别的软件生成的文件，本软件打不开。原因通常是格式变了、编码变了、或者新版本对格式做了更严格的校验——**严格化本身也是一种不兼容**。
>
> **第二个现象：数据显示不对。**文件能打开，但内容错位：中文变乱码、日期差一天、金额小数点错位、列表顺序变了、字段串行。这一类最阴险，因为**系统不报错**，用户以为是对的，直到某天对账对不上。数据兼容问题里，静默错误比崩溃危险得多。
>
> **第三个现象：导致系统崩溃。**读到预期之外的结构就直接崩了——空指针、数组越界、解析异常没处理。这说明代码对"外部数据"的健壮性不足，而这类数据往往来自旧版本或其他系统。

三个现象背后有一个共同的判断标准，我给你们一句话：**数据兼容性测试的核心不是"能不能读"，而是"读出来的东西对不对，以及读错了会不会崩"。**

所以数据兼容测试要问四个问题：**能读吗？读全了吗？读对了吗？读错的时候系统怎么反应？**最后这一问最容易被漏——正确处理方式是"明确报错并拒绝处理"，最糟的是"带着错误数据继续算下去"。

还有一个容易忽略的点：**数据是会"往回走"的。**用户升级后可能回滚、可能把新版本文件发给老版本用户、可能用新版本导出再导进老系统。所以测试不能只测"新读旧"，还要测"旧读新"。

板书：三个现象：文件打不开（格式/编码/严格化）｜数据显示不对（乱码・日期・金额・顺序，静默错误）｜系统崩溃（健壮性不足）。下方四问：能读吗・读全吗・读对吗・读错怎么反应。再写"新读旧 + 旧读新"。

提问：为什么"数据显示不对"比"文件打不开"更危险？

预设回答：

- 答"因为不报错，没人发现"——正是核心。打不开至少知道坏了，显示错了会一路错下去。

- 答"因为可能影响决策"——对，报表数字错了，业务决策就错了。

- 答"因为难追溯"——也是，等发现时已经产生了大量错误数据，清理成本极高。

- 收口：所以数据兼容测试必须做**内容比对**，不能只看"有没有异常"。

易错点 / 考点：① 三个现象是常考填空；② 易错点：只测"能否打开"，不测"内容是否正确"——这是数据兼容测试最典型的漏测；③ 常考判断"只要程序不崩溃，数据兼容就没问题"——错，静默数据错误危害更大。

过渡：数据兼容分两种情况：同一个系统的新旧版本之间，和不同系统之间。先看前者。

【7.115】相同系统之间的数据兼容（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲同一个系统内部的数据兼容，它是数据兼容里最基础、也最常出事的一种。图上有两个方向，我按"数据怎么流"来讲。

第一个方向，客户端与服务端之间的数据文件。图中写了"上传旧版／下载新版数据文件"。这指的是：客户

端升级了，但它要处理的文件可能是旧版客户端上传的；反过来，服务端生成的新格式文件，旧版客户端还得能打开（如果还支持老客户端）。这一层的关键是**格式版本号**——文件里应该带一个版本标记，程序读到之后按对应规则解析，而不是"一律按最新格式读"。

第二个方向，软件与数据库之间。图中写了两条：**新版本软件存取历史数据**，和**原版本软件存取 DB 新 schema 的数据**。前者是"新读旧"，后者是"旧读新"。这两条的难度完全不同：新读旧是设计时就该保证的（向后兼容），旧读新通常做不到——所以工程上的做法是**保证可回滚窗口内的兼容**，比如加字段时给默认值、不改字段含义、不删字段。

还有一条贯穿两者的动作：DB schema 升级。数据库结构变更（加表、加字段、改类型、加索引）是最容易出事的环节。经验规则有四条：**加字段要给默认值或允许为空；不要改字段类型和含义；不要删字段，先标记废弃；大批量数据变更要可重入、可中断、可续跑。**

为什么这一页值得慢慢讲？因为它对应的是"升级后数据全乱"这类事故。这类事故的可怕之处在于**不可逆**——代码可以回滚，数据写错了要一条条修。

所以我把这一页的结论说重一点：**动数据结构之前，先想清楚怎么退回去。**这不是保守，这是对已经在系统里的那批数据负责。

> **板书：**两个方向：客户端↔服务端（上传旧版/下载新版，靠格式版本号）｜软件↔DB（新读旧＝必须保证；旧读新＝靠回滚窗口）。下方 schema 四规则：加字段给默认・不改类型含义・不删只废弃・批量变更可重入。

提问："旧版本软件读取新 schema 的数据"为什么通常做不到？那工程上怎么处理？

预设回答：

- 答"因为旧代码不认识新字段"——对，而且可能因为多余字段直接报错。

- 答"所以只能保证回滚期内兼容"——正是实务做法：不改语义、只加可选字段。

- 答"可以用视图或中间层隔离"——很专业的思路，用数据库视图或 API 层做适配。

- 收口：不能保证"永远可回滚"，就保证"**回滚窗口内可回滚**"，并把窗口写进升级方案。

易错点 / 考点：① 两个方向（新读旧／旧读新）与对应结论是高频考点：**新读旧必须保证，旧读新靠回滚窗口**；② schema 变更的四条规则常考简答；③ 易错点：认为"加了字段不影响老代码"——很多序列化框架对未知字段是严格模式，会直接失败。

【思政落点 7】：数据兼容这一页，我想请大家对"历史数据"保持一点敬畏。系统里那批数据不是测试数据，是用户几年积累下来的订单、病历、成绩、账目——它们没有备份可重来，也没有"重跑一遍"的机会。技术上，加一个字段、改一个类型只是几秒钟的事；但对数据来说，那可能是不可逆的改写。所以我们定规矩：**改结构前先想回滚，跑脚本前先做备份，批量变更必须可中断可续跑**。这种"动手前先想后果"的谨慎，就是工程人员对他人财产的尊重。

过渡：同一个系统内部尚且如此，不同系统之间的数据兼容，难度还要再上一个台阶。

【7.116】不同系统之间的数据兼容（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲不同系统之间的数据兼容。图上列了三组来源：来自 Microsoft、运行在 Windows 11 上的 Word Editor；来自 Apple、运行在 Mac OS X 上的 Word Editor；来自 IBM Lotus、运行在 Windows 11 上的 Spreadsheet。右边列出了五种互通方式：**网络导入/导出、文件导入/导出、文件加载/保存、剪切复制粘贴、备份**。

这五种方式，本质上是**五条数据进入本系统的通道**，而每一条通道都是兼容性测试的对象。很多人只测"文件导入导出"，漏掉另外四条，结果在真实使用中出问题。

网络导入/导出：通过接口或协议交互，注意编码、时区、精度、字段长度限制。

文件导入/导出：格式版本、字符集、换行符、BOM、分隔符、日期格式。

文件加载/保存：直接打开第三方格式，或者在第三方软件里打开本系统的文件。这一条最考验格式解析的健壮性。

剪切复制粘贴：最容易被忽略的一条。从 Word 粘贴带格式的文本到网页编辑器、从 Excel 粘贴一片单元格到表格控件——剪贴板里可能有 HTML、RTF、纯文本多种格式，系统取哪一种、怎么清洗，直接决定会不会出乱码、会不会把脚本一起粘进来（这又回到了 XSS）。

备份：备份出来的数据能不能在新版本、新环境里恢复。**没验证过恢复的备份，等于没有备份。**

这一页还要引出一个重要判断：**跨系统兼容的目标不是"完美还原"，而是"关键信息不丢、语义不错"**。字体、行距、复杂表格样式这类东西，跨系统几乎不可能完全一致，测试要有取舍，把力气放在"金额、日期、编号、

状态"这些不能错的地方。

测试设计上，我建议按"格式 × 通道"做矩阵：每一种来源格式，走每一条通道，各测一遍。

板书：左侧三种来源（MS Word／Apple Word／Lotus Spreadsheet）；右侧五种通道：网络导入导出・文件导入导出・加载保存・剪贴粘贴・备份恢复。下方写"目标＝关键信息不丢、语义不错"，测法＝格式×通道矩阵。

提问：为什么"剪切复制粘贴"这条通道最容易被漏测，却又最容易出问题？

预设回答：

- 答"因为它不算功能"——正是这种想法，它没有专门的按钮，藏在用户习惯里。

- 答"因为剪贴板内容格式复杂"——对，同一份内容可能同时有 HTML、RTF、纯文本三个版本。

- 答"因为还会带进脚本"——很关键，粘贴是最经典的存储型 XSS 入口之一。

- 收口：所以粘贴要测两件事：**格式还原得对不对，危险内容有没有被清洗**。

易错点 / 考点：① 五种互通通道是常考多选，注意"备份"也算一种数据兼容场景；② 易错点：只测导入导出，漏掉剪贴板与备份恢复；③ 常考判断"跨系统兼容要求显示效果完全一致"——错，目标是关键信息与语义正确。

过渡：真实的系统往往不只两种格式，而是"能读很多种、能写很多种"，这就是下一页的复杂案例。

【7.117】数据兼容性更复杂的案例（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲一个更复杂的场景：系统要**导入下列格式文件**，也要**读取/导出下列格式文件**，同时还要**读取已有文件**，并且最好能**另存为**。这四个动作叠在一起，兼容性的组合数就上来了。

我把它拆成四种能力，逐个讲测试要点。

第一种能力：导入。把外部格式读进来，转成内部模型。测试要点是"畸形输入"——字段缺失、字段顺序颠倒、多余字段、超长字段、空文件、超大文件、编码不是 UTF-8、含特殊字符和公式。**导入功能的健壮性，决定了系统会不会因为一个脏文件而崩。**

第二种能力：读取/导出。把自己的数据写成外部格式给别人用。测试要点是"对方能不能正确打开"——这就

要真的拿目标软件去打开验证，而不是看自己系统里显示正常。**导出的正确性只能由接收方来判定。**

第三种能力：读取已有文件。这里的关键词是"已有"——历史文件、旧版本文件、别人给的文件。这一类文件往往不符合你当前的假设：没有版本号、字段缺失、用了废弃的取值。测试要点是"向后兼容的边界"——最老支持到哪个版本？超出范围时是明确报错还是静默出错？

第四种能力：另存为。课件在这里标了"（better）"，意思是推荐支持。为什么推荐？因为"另存为"是用户的逃生通道——新版本打不开或者写坏了，还能另存成通用格式保住数据。**一个不提供导出通道的系统，等于把用户的数据锁死了。**

这四种能力合起来，构成一个"格式 × 动作"的矩阵：行是格式，列是导入、导出、读取、另存。每一格都要有测试用例。你会发现用例数量增长很快——这正是下一页要用组合测试方法来压缩的原因。

还要提醒一点：**格式兼容是有成本的**。支持越多格式，维护成本越高，安全风险也越大（解析器是最容易被攻击的组件）。所以产品要明确"支持哪些格式、支持到什么程度"，而不是"尽量都支持"。

板书：四种能力四行：导入（畸形输入）｜导出（由接收方验证）｜读取已有（向后兼容边界）｜另存为（用户逃生通道）。下方画"格式 × 动作"矩阵示意，右下写"解析器＝高危组件，支持范围要有边界"。

提问：为什么说"导出的正确性只能由接收方来判定"？

预设回答：

- 答"因为自己系统里看着正常不代表文件合规"——正是，可能用了私有扩展、可能编码不对。
- 答"因为对方软件更严格"——对，很多软件对格式的校验比生成方严格得多。
- 答"所以要拿目标软件实际打开"——这是唯一可靠的验证方式，也常被省略。
- 收口：测试导出功能的正确姿势是**闭环验证**：导出 → 用目标软件打开 → 比对关键字段 → 再导回来自比对。

易错点 / 考点：① 四种能力与各自测试要点是常考简答；② 易错点：把"导出成功"当成"导出正确"；③ 注意"另存为"的价值——它是数据可迁移性的保障，也是避免厂商锁定的手段；④ 常考判断"支持格式越多越好"——错，要看维护成本与解析安全风险。

过渡：格式和动作一乘，用例就爆炸了。怎么在有限时间里覆盖尽可能多的组合？这就是组合测试要解决的问题。

【7.118】软件环境(组合)兼容性测试（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲软件环境组合的兼容性测试，核心方法两个：**Pairwise 组合测试**和**正交实验法**。为什么需要它们？因为环境是多个维度的笛卡尔积。

举个具体例子：操作系统 4 种（不同 Windows 版本、macOS、Linux）× 浏览器 5 种 × 分辨率 4 种 × 语言 3 种 × 网络条件 3 种，乘起来是 4×5×4×3×3＝720 种组合。全测一遍，不现实。

Pairwise（两两组合）的思路是：绝大多数缺陷是由**两个因素之间的交互**触发的，三个及以上因素同时交互才触发的缺陷比例很低。所以只要保证"任意两个因素的每一对取值组合都至少出现一次"，就能用很少的用例覆盖大部分风险。上面那个 720 种的环境，用 pairwise 通常能压到二三十条用例。

正交实验法的思路相近，但更强调用**正交表**来保证各因素各水平的均衡搭配，适合因素和水平都比较规整的场景。两者的区别在考试里常考：**pairwise 关注"两两覆盖"，正交表关注"均衡与可分析"**——正交表还能做极差分析，判断哪个因素影响最大。

课件还给了关键的一句：**根据市场调查数据，设计组合矩阵表**。这句话的意思是，组合不能凭感觉选——要先知道用户实际在用什么。如果 80% 的用户在 Windows 上用 Chrome，那 Windows + Chrome 这一格必须优先、必须深测；某个只占 0.3% 的组合，用 pairwise 覆盖到就行，不必投入太多。

所以这一页的完整方法链是：**列出因素与水平 → 查市场占用数据 → 用 pairwise／正交表生成组合 → 按占用率分配测试深度 → 形成组合矩阵表并沉淀**。

最后提醒一句：组合测试能压缩用例，但它**假设缺陷主要来自两两交互**。有些缺陷确实要三个因素同时满足才出现，比如"特定浏览器 + 特定语言 + 特定时区"。所以高风险场景仍然需要**针对性补测**，不能全靠组合。

> **板书：**写"4×5×4×3×3＝720 种"；下行写"pairwise：任意两因素的取值对至少出现一次"；再写"正交表：均衡搭配 + 可做极差分析"；最后写方法链：列因素 → 查市场数据 → 生成组合 → 按占用率分配深度 → 矩阵表沉淀。

提问：为什么"两两组合"能在用例数量大幅下降的同时，还保持较高的缺陷发现率？

预设回答：

- 答"因为大部分 bug 是两个因素交互引起的"——这是 pairwise 的理论前提，也是它的边界。

- 答"因为三因素交互的 bug 很少"——对，比例低但不为零，所以高风险处要补测。

- 答"因为随机测也能测出来"——不对，pairwise 的价值恰恰是**系统性地保证覆盖**，而不是碰运气。

- 收口：说清楚它的**假设**，比记住它能省多少用例更重要——假设不成立的地方，就是它失效的地方。

> **易错点 / 考点：**① pairwise 与正交实验法的区别是常考简答：**前者保证两两覆盖、用例最少；后者用正交表保证均衡、可做因素影响分析**；② 易错点：认为组合测试可以完全替代针对性测试——它对三因素以上的交互覆盖不足；③ 常考计算题：给因素与水平数，问全组合数，再问 pairwise 大致量级。

> **过渡：**组合测试把环境的矩阵压缩了，但它压的是"因素之间"的关系。还有一种矩阵是"系统之间"的关系，我们收个尾。

【7.119】小结（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 7.3 兼容性测试的小结，它把前面十页的内容压成了三组概念，我按"概念 → 动作 → 范围"的顺序讲。
>
> **第一组概念：向后兼容与向前兼容。**课件写得很清楚：**向后兼容＝可使用之前版本的数据；向前兼容＝可使用未来版本的数据。**这两个词最容易搞反，我给一个记忆办法——**看"兼容"这个词的主体是谁**。"新版本兼容旧数据"叫向后兼容，因为它是往后（旧方向）看；"旧版本能读新数据"叫向前兼容，因为它是往前（新方向）看。
>
> 这里有个实务结论要记牢：**向后兼容是必须保证的，向前兼容通常只在不破坏结构的前提下尽量做**。因为新版本知道你过去写过什么，旧版本不可能预知你未来会写什么。
>
> **第二组动作：升级、迁移、备份、恢复、导入/导出。** 这五个动作都是"数据在动"的场景，也就是数据兼容问题的高发区。请注意它们的关系：升级和迁移是"主动改变"，备份和恢复是"保险"，导入导出是"与外界交换"。测试的时候，这五个动作要两两组合着测——比如"升级失败后恢复备份"、"导出后导入到新环境"，这类组合场景才是真实事故的样子。
>
> **第三组范围：硬件兼容、系统之间兼容、数据之间兼容。**这三条就是本节的主线，也对应三个不同的问题层次：硬件层是"跑不跑得起来"，系统层是"能不能配合"，数据层是"读出来对不对"。
>
> 我把这一节浓缩成一句话：**兼容性测试的本质，是在"环境发散"和"测试资源有限"之间做取舍**——先用组合

方法压缩，再用市场数据排序，最后把结论固化成矩阵表。

还有一个容易被忽略的产出：**兼容性测试的结论应该写成"支持矩阵"**——明确写"支持哪些环境、不支持哪些、不支持时表现如何"。这份矩阵既是测试结论，也是给用户的承诺。

板书：三组并列：概念（向后兼容＝读旧数据／向前兼容＝读新数据）｜动作（升级・迁移・备份・恢复・导入导出，两两组合测）｜范围（硬件・系统之间・数据之间）。下方写"产出＝支持矩阵"。

提问：如果一个产品文档只写"支持 Windows"，你会追问哪几个问题？

预设回答：

- 答"哪个版本，Win7 还是 Win11"——必要，版本差异很大。

- 答"32 位还是 64 位"——也很关键，很多兼容问题就出在这里。

- 答"不支持的时候是什么表现"——这一问最专业，明确失败方式比扩大支持范围更重要。

- 收口：所以"支持矩阵"要三列：**环境、支持状态、不支持时的表现**。三列齐全才算写完。

易错点 / 考点：① 向后兼容与向前兼容的定义是**高频易错点**，务必按"新读旧／旧读新"来记；② 五个动作（升级・迁移・备份・恢复・导入导出）常考多选；③ 易错点：认为"兼容性测试做完就完了"，漏掉"支持矩阵"这个交付物。

过渡：兼容性讲完，7.3 结束。下面进入 7.4——可靠性测试，这一节里有一半内容是近几年才真正流行起来的。

【7.120】7.4 可靠性测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

进入 7.4 可靠性测试。这一节在教材的经典内容之外，课件明确加了一句：**增加了现在流行的全链路压测和混沌工程**，并且给了一个定位说明——**全链路压测属于稳定性工程，但也可以算性能测试的一类**。

这句话信息量不小，我拆一下。第一，可靠性测试的经典内容是"成熟性、容错性、易恢复性"，偏"系统能不能稳定运行"；第二，近几年业界把它扩展成了"稳定性工程"，关注的是"在真实流量、真实故障下，系统还能不能扛住"；第三，全链路压测和混沌工程是这套工程实践的两个抓手——**一个负责把真实规模的流量灌进去，一个负责把真实世界的故障注进去**。

为什么这两件事重要？因为传统性能测试有个致命缺陷：**它测的是"我们的测试环境在模拟流量下表现如何"**，而生产环境的数据量、调用链深度、依赖数量、网络拓扑，跟测试环境完全不是一回事。你在测试环境压出 1000 TPS 很开心，上线后 300 TPS 就雪崩——中间差的就是"全链路"和"真实数据"。

所以这一节的逻辑是：先讲可靠性的三个度量维度，再讲容错性测试，然后花大力气讲全链路压测——从需求、流量控制、平台原理、数据处理、录制回放、全流程，一直到开源实现 Takin。

请带着一个问题听：**如果要给一个日活百万的系统做一次"能不能扛住大促"的验证，你会怎么设计？**这一节讲完，你应该能答出来。

> **板书：**写"7.4 可靠性＝经典三性 + 稳定性工程"；左侧写"成熟性／容错性／易恢复性"；右侧写"全链路压测（灌真实流量）＋混沌工程（注真实故障）"；下方写"测试环境 ≠ 生产环境，差在链路与数据"。

提问：为什么"在测试环境压出很高的 TPS"这件事，说服力有限？

预设回答：

- 答"因为数据量不一样"——对，索引命中率、缓存命中率全变了。
- 答"因为调用链短"——很关键，测试环境常常把下游 mock 掉了，真实环境要穿透十几个服务。
- 答"因为网络和中间件配置不一样"——也对，连接池、超时、重试策略的默认值经常不同。
- 收口：所以压测结论必须标注**在什么环境下得出**，否则这个数字没有意义。

易错点 / 考点：① 课件给的定位句要记住："全链路压测属于稳定性工程，也可以算性能测试的一类"；② 易错点：把可靠性测试等同于"跑久一点看会不会挂"——那只是成熟性的一部分；③ 常考简答"为什么需要全链路压测"，答题要点：环境差异、链路完整性、真实数据分布。

过渡：先把可靠性的度量口径讲清楚，后面所有讨论才有依据。

【7.121】可靠性测试方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给可靠性的定义和度量口径，是本节的理论地基。定义是：**在规定的一段时间和条件下，软件能维持其性能水平的能力有关的一组属性**，可用**成熟性、容错性、易恢复性**三个基本子特性来度量。请注意定义里的两个限定词——"规定的一段时间"和"规定的条件"。可靠性从来不是绝对的，**脱离时间和条件谈可靠性没**

有意义。

三个子特性，逐个讲。

成熟性：系统不出故障的能力。课件给了度量方式：**通过错误发现率 DDP（Defect Detection Percentage）来表现，DDP 越小，软件越成熟**，公式是 **DDP＝测试发现的错误数量 ÷ 已知的全部错误数量**。这里要特别注意方向——**DDP 小是好事**。为什么？因为 DDP 衡量的是"测试阶段发现的错误占全部已知错误的比例"，DDP 小说明大部分错误在更早的阶段就被消灭了，或者根本没被引入，说明过程成熟。这个方向很容易记反，考试常考。

容错性：出了故障还能继续提供正确服务的能力。比如某个下游服务超时，系统降级返回缓存数据而不是直接报错。这一项单独有一页讲。

易恢复性：故障之后恢复的速度和完整度。课件的测法是：**先设法（模拟）使系统崩溃、失效等，然后计算其系统和数据恢复的时间，来做出易恢复性评估**。注意这里有两个指标——**恢复时间**和**恢复完整度**（数据有没有丢、状态有没有错）。

三个子特性的关系可以这么理解：**成熟性管"少出事"，容错性管"出事不倒"，易恢复性管"倒了能快起来"**。三者合起来才是完整的可靠性。

落到测试设计上：成熟性靠长时间运行和缺陷数据分析；容错性靠故障注入；易恢复性靠"破坏—恢复"的演练。三种手段完全不同，不能混着用。

> **板书：**写定义关键词"规定的时间＋规定的条件"；三个子特性竖排：成熟性（DDP＝测试发现/已知全部，越小越成熟）｜容错性（出事不倒）｜易恢复性（恢复时间＋恢复完整度）。下方写口诀"少出事・出事不倒・倒了快起"。

提问：DDP 越小越成熟，这个方向和很多指标相反。你能解释为什么吗？

预设回答：

- 答"因为它算的是测试阶段发现的比例"——对，分母是"已知的全部错误"，包括更早阶段发现的。

- 答"测试阶段发现得少，说明前面拦住了"——准确，这正是"过程成熟"的含义。

- 答"那是不是测试越弱 DDP 越小"——这是一个很好的反问！确实存在这个陷阱：如果前期没发现、测试也没发现，DDP 也会小。所以 DDP 要配合"逃逸缺陷数"一起看。

- 收口：任何单一指标都能被"操作"，所以度量要看一组指标，而不是一个数。

易错点 / 考点：① **DDP 越小越成熟**是高频易错点，务必记牢方向；② 三个子特性的名称与含义常考连线；③易恢复性要答出"恢复时间"和"数据完整性"两个维度，只答"恢复得快"不完整。

过渡：三个子特性里，"出事不倒"这一项需要特殊手段才能测——你得先把系统弄坏。

【7.122】容错性测试（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲容错性测试。课件第一句就点出了它的性质：**容错测试是一种对抗性的测试过程**。注意"对抗性"这三个字——它不是"验证正常功能"，而是"主动搞破坏"。

具体做法是：**通过各种手段让软件强制性地发生故障，或者把应用程序或系统置于（模拟的）异常条件下，以产生故障，例如设备输入/输出（I/O）故障或无效的数据库指针和关键字等。**

这句话里给了两类故障源：**环境类**（I/O 故障、网络中断、磁盘满、时钟漂移）和**数据类**（无效的指针、无效的关键字、脏数据）。前者模拟"世界不正常"，后者模拟"数据不正常"。

课件还给了一句很有分量的判断：**之前有故障注入测试，现在有混沌工程，目前掌握混沌工程更有意义**。这句话是在告诉你技术演进的方向。故障注入是"我知道哪里可能坏，我去把它弄坏"；混沌工程是"我不知道哪里会坏，我用受控实验的方式在生产环境里随机制造真实故障，观察系统整体表现"。

两者的差别有三个：**范围**（单点 vs 系统级）、**环境**（测试环境 vs 生产环境）、**方法**（预设故障 vs 实验驱动）。混沌工程有几个必须遵守的前提，我列给大家：**有明确的稳态指标**（比如成功率、延迟分位数）、**有爆炸半径控制**（先小范围、可随时中止）、**有监控与回滚预案**。没有这三条，那不是混沌工程，那是事故演练。

测什么、怎么判？我给三条验收标准：**故障发生时，核心功能是否仍可用**（降级是否生效）、**故障恢复后，数据是否一致**（有没有丢单、重复）、**用户是否得到明确反馈**（不是白屏、不是无限转圈）。

一句话：**容错性测试考的不是"不出故障"，而是"出了故障之后系统怎么表现"。**

板书：写"对抗性测试"；左侧两类故障源：环境类（I/O・网络・磁盘・时钟）／数据类（无效指针・无效关键字・脏数据）；右侧写"故障注入 → 混沌工程"对比：单点/系统级・测试环境/生产・预设/实验驱动；下方三条前提：稳态指标・爆炸半径・监控回滚。

提问：为什么"混沌工程"必须在生产环境做，而不能只在测试环境做？

预设回答：

- 答"因为测试环境没有真实流量"——对，没有流量就观察不到影响。

- 答"因为生产环境的依赖关系才是真的"——很准，测试环境常常缺依赖、走 mock。

- 答"所以风险很大，要控制爆炸半径"——正是关键，所以才有"先小范围、可中止"的规矩。

- 收口：混沌工程不是"更大胆的破坏"，而是"**更严格的实验设计**"——它把风险控制写进了方法本身。

易错点 / 考点：① 容错性测试是"对抗性测试"这个定性常考；② 故障注入与混沌工程的差别是高频考点，答题按"范围／环境／驱动方式"三点对比；③ 易错点：认为混沌工程就是"随便杀进程"——必须先有稳态指标和爆炸半径控制。

过渡：容错性测的是"坏了怎么办"，而下一页开始的这一块，测的是"真实规模的流量来了扛不扛得住"。

【7.123】全链路压测（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是全链路压测的入口。前面 7.1 讲的性能测试，主要是在测试环境里用 JMeter 这类工具压单个服务或单条链路；全链路压测要解决的是另一个层次的问题。

什么叫"全链路"？三层含义。

第一，链路完整。从用户入口（CDN、网关、负载均衡）一直压到最底层的数据库、缓存、消息队列，中间每一个服务、每一次 RPC 调用都在链路上，一个都不能 mock。因为瓶颈往往不在你以为的地方——应用层 CPU 才 30%，数据库连接池已经满了。

第二，流量真实。请求的比例、参数的分布、用户的思考时间、数据的倾斜程度，都要接近生产。压测不是"把同一个请求重复一百万次"，那样缓存全命中，什么也测不出来。

第三，数据隔离。这是全链路压测最难的工程问题：**压测会产生写操作**，写进去的数据会污染真实业务——订单、库存、账户余额都不能乱。所以必须有"影子"机制：影子库、影子表、流量标记、挡板。这也解释了后面 Takin 为什么把"0 侵入 + 数据隔离"当作核心卖点。

全链路压测的价值在哪？它是**大促、秒杀、开学选课、春运抢票**这类场景唯一可信的验证手段。因为这些场景的特点都是"短时间、极高并发、全链路承压"，测试环境永远模拟不出来。

这一节后面会一层层展开：为什么要做（需求）、流量怎么控、平台怎么搭、数据怎么准备、流量怎么录制回放、整个流程怎么组织、开源实现长什么样。**注意顺序——先想清楚"为什么"，再谈"怎么做"，否则很容易变成堆工具。**

> **板书：**写"全链路＝链路完整＋流量真实＋数据隔离"；下方列"入口 CDN/网关 → 服务 → 中间件 → DB/缓存/MQ"一条链；右侧写"数据隔离手段：影子库表・流量标记・挡板"。

提问：为什么"把同一个请求重复一百万次"这种压测方式，结论不可信？

预设回答：

- 答"因为缓存全命中了"——最直接的原因，第二次之后就不查库了。

- 答"因为没有数据倾斜"——对，真实流量的热点分布很不均匀。

- 答"因为写操作比例不对"——也很关键，纯读和读写混合的瓶颈完全不同。

- 收口：所以压测脚本必须有**流量模型**：接口比例、参数分布、读写比、思考时间，四样齐全才算"模拟"。

易错点 / 考点：① 全链路压测的三个特征（链路完整、流量真实、数据隔离）是核心考点；② 易错点：把全链路压测等同于"用更多压力机"——关键在链路与数据，不在并发数；③ 常考简答"全链路压测与普通性能测试的区别"，答题要落到"环境、链路、数据、隔离"四点。

过渡：那到底什么情况下必须做全链路压测？课件把需求单列了一页。

【7.124】全链路压测的需求（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲"什么时候需要全链路压测"。这个判断很重要，因为全链路压测的成本极高——要改造中间件、要准备影子库、要协调多个团队、要在生产环境操作，不是所有项目都值得做。

我按四个触发条件来讲。

第一，有明确的、可预期的流量峰值。大促、秒杀、抢购、开学选课、节假日出行——这些场景的流量曲线

是已知的，峰值是必须扛住的。既然知道要来，就必须提前验证。

第二，系统规模大到"单点压测失去意义"。当一条业务链路要经过十几个服务、几个中间件、多个数据源的时候，单独压某一个服务，得出的数字跟整体表现没关系。**瓶颈在链路里，不在服务里。**

第三，容量决策需要数据支撑。要扩容几台机器？数据库要不要分库分表？缓存要多大？这些问题如果靠"拍脑袋 + 上一年的经验"，很容易估错。全链路压测给的是**容量水位**：当前配置能扛多少，瓶颈在哪个节点，扩哪里最有效。

第四，稳定性有硬性要求。金融、医疗、政务、电商交易，这类系统一次故障的代价极高，验证的投入是值得的。

反过来，什么情况可以不做？**流量平稳、链路短、可降级、故障代价小**的系统，做常规性能测试加监控就够了。做全链路压测不是"更专业"的标志，而是"成本与风险匹配"的选择。

还有一个现实约束要讲：全链路压测需要**跨团队协作**——业务、测试、运维、DBA、中间件团队都要参与。所以它往往由专门的"稳定性团队"牵头，而不是某一个测试人员能独立完成的。这也是为什么后面要讲"压测平台"——平台的作用就是把这种复杂性收进去。

> **板书：**四个触发条件竖排：有可预期峰值｜链路长到单点压测失效｜容量决策需数据｜稳定性硬要求。下方写"不做的情形：流量平稳・链路短・可降级・代价小"；再写"组织前提：跨团队 + 平台化"。

提问：一个内部管理系统，日活五百人，要不要做全链路压测？

预设回答：

- 答"不用"——基本正确，但补一句：仍要做基本的性能验证和容量评估。

- 答"要看有没有峰值"——很专业的判断，比如月末结算、批量导入，也可能有尖峰。

- 答"做了更保险"——投入产出不划算，全链路压测的成本可能超过系统本身的价值。

- 收口：判断标准是**"故障代价 × 峰值概率"是否大于"压测成本"**，而不是"别人都做"。

易错点 / 考点：① 四个触发条件是常考简答；② 易错点：认为"重要系统就必须做全链路压测"——要看链路复杂度与峰值特征；③ 常考判断"全链路压测只能由测试团队独立完成"——错，它需要多团队协作与平台支撑。

过渡：决定要做之后，第一个技术难题就是：怎么保证灌进去的流量不把真实业务搞乱。

【7.125】系统流量控制（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲流量控制，是全链路压测最关键的安全阀。图里画的是一条完整的下单链路：**订单 → 管道 → MQ 消息 → 预分拣 → MQ 消息 → 任务分发 → 下游系统**，下面是被压的**应用集群**（画成一排 Docker 容器），流量来自**互联网的生产流量**，中间有**憋洪&泄洪**、**阻塞&泄洪**这样的标记。

我按三个机制讲。

第一个机制：流量标识与路由。 压测流量进来时必须带标记（通常是 HTTP 头里的一个特殊字段），中间件识别这个标记，把请求路由到影子库、影子表、影子缓存。这样压测的写操作落在"另一套"存储上，真实数据不受影响。这就是"数据隔离"的技术实现。

第二个机制：憋洪与泄洪。 这是这张图的关键词。"憋洪"是把流量先攒住——比如压测开始时，先把请求压在入口，等所有节点就绪、监控到位，再统一放行。"泄洪"是放行。为什么要这么做？因为压测要求**瞬时并发**，如果流量是慢慢涨上去的，你就测不到峰值那一刻的系统表现。憋洪还有一个好处：**可以随时喊停**，流量还没放出去，风险是可控的。

第三个机制：阻塞与限流。 图里还标了"阻塞&泄洪"，指的是当系统水位超过阈值时，主动把流量挡住，避免把系统压垮。这在生产环境压测里是必须的——**你可以在生产环境压测，但不能把生产环境压死**。

为什么这一页重要？因为它回答了一个矛盾：**全链路压测必须在生产环境做（否则不真实），但生产环境不能出事故**。解决这个矛盾靠的就是流量控制：**标记隔离数据、憋洪制造峰值、限流兜住底线**。

一句话总结：**全链路压测的第一原则不是"压得多狠"，而是"随时能停"。**

> **板书：**画链路：订单 → 管道 → MQ → 预分拣 → MQ → 任务分发 → 下游系统；下方画应用集群（Docker1…9）与"互联网生产流量"；标注三个机制：流量标识（影子库表）｜憋洪&泄洪（造峰值、可喊停）｜阻塞&限流（兜底线）。

> **提问：**为什么"憋洪"这个动作，对压测结论的有效性是必需的？

> **预设回答：**

- 答"因为要制造瞬时并发"——正是，慢慢涨上去测不到峰值表现。

- 答"因为要等所有节点就绪"——对，压测前要确认监控、影子库、挡板都到位。

- 答"因为可以随时取消"——很关键，流量攒在入口，风险可控。

- 收口：所以憋洪既是"技术手段"，也是"风险控制手段"——这一点在设计压测方案时必须写进流程。

易错点 / 考点： ① 憋洪/泄洪、阻塞/限流的含义是常考名词解释；② 易错点：把"限流"当成"压测能力不足的妥协"——它是**保护生产**的必要设计；③ 常考简答"全链路压测如何避免污染生产数据"，答题要提"流量标识 + 影子库表 + 挡板"。

过渡： 流量控制的策略清楚了，我们看支撑这一切的压测平台，它整体是怎么搭起来的。

【7.126】压测平台原理示意图（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是压测平台的总体架构图，信息量很大，我按"谁发压、发到哪、怎么控"三个问题带你看。

第一个问题：谁发压？ 图中间是**流量引擎集群**，下面分成两大类节点：**站外分区域节点**和**站内各机房节点**。

站外分区域节点挂的是各家运营商的 CDN 节点——联通、电信、移动，还有其它 CDN 节点。为什么要在站外发压？因为真实用户就是从这些入口进来的，**压测如果只从内网发，就测不到 DNS、CDN、公网带宽这些环节**。而且分运营商发压能暴露"某一路带宽打满"这类问题。

站内各机房节点就是 IDC 机房 A、B、C、D，每个机房里部署压力机引擎。为什么机房要多点部署？两个原因：一是**压力机自身的网络出口有限**，单机房发不出足够大的流量；二是要模拟**跨机房调用**，真实流量不会只打一个机房。

第二个问题：发到哪？ 右侧是**业务系统**：首页、购物车、结算页……这些是被压对象。注意它有两个入口——**公网入口**和**内网入口**。公网入口走 CDN 和域名，测的是全链路；内网入口直连服务，测的是后端容量。两种入口对应两种测试目的。

第三个问题：怎么控？ 左侧是压测平台本体，它管三样东西：**脚本、数据、流量控制台**。脚本是"发什么请求"，数据是"用什么参数发"（包括压测账号、商品 ID、优惠券），流量控制台是"发多少、什么时候发、什么时候停"。

这套架构的设计思想，说到底就一句：**把"发压能力"和"业务系统"解耦，用平台统一调度。** 压力机是可以横向扩的，脚本和数据是可以复用的，流量是可控的。这样一次压测的组织成本才能降下来，否则每次都要临

时拼机器、临时写脚本。

还有一个细节值得注意：图里压力机引擎画了很多个，但**它们不是各自为战**——必须由统一的控制器同步启动，才能形成瞬时并发。这就引出了下一页的平台内部结构。

板书：三问三答：谁发压（流量引擎集群：站外 CDN 分区域＋站内 IDC 多机房）｜发到哪（业务系统：首页/购物车/结算页，公网入口＋内网入口）｜怎么控（压测平台：脚本・数据・流量控制台）。

提问：为什么压力机要部署在多个机房、多个运营商节点，而不是集中在一个机房？

预设回答：

- 答"单机房出口带宽不够"——最直接的原因，压力机自己会先成为瓶颈。

- 答"要模拟跨机房调用"——对，真实流量会分散到多个机房，集中打一个不真实。

- 答"要测不同运营商的链路"——很专业，某一路 CDN 或某个运营商链路的问题只有分区域发压才暴露。

- 收口：所以"发压能力"本身也是需要设计和验证的——压不上去，结论就不成立。

易错点 / 考点：① 三大组成（流量引擎集群、压测平台、业务系统）与两个入口（公网/内网）是常考填空；② 易错点：认为"压力机越多越好"——关键是分布与同步，而不是数量；③ 常考简答"为什么要在站外 CDN 节点发压"，答题要点：覆盖公网链路、分运营商、模拟真实入口。

过渡：外部看是"集群发压"，内部看是一套完整的调度系统。下一页拆开看。

【7.127】压测平台原理示意图-2（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是压测平台的内部结构，从"人怎么用"到"机器怎么执行"，一条链走下来。我按数据流讲。

整条链从人开始：**压测人员**通过**前端页面**操作，平台侧存着**脚本仓库**和**数据库**——脚本仓库存压测脚本（哪个场景、哪些接口、什么参数），数据库存场景配置、任务记录、结果数据。

中间是三个服务，这是平台的核心：**任务调度服务**负责"什么时候、把哪个任务、发给哪些 Agent"；**结果采集服务**负责从各个 Agent 收集压测数据；**实时计算服务**负责把这些数据算成指标（TPS、响应时间、错误率、分位数）。

为什么要有"实时"两个字？因为压测是动态的——你看到 TPS 曲线掉了、错误率上来了，必须**当场决定**是加压、减压还是停。如果结果要等压测结束才出来，那这次压测就白做了。

再往下是数据通道：**Kafka** 做消息缓冲，**ES** 做存储与检索。为什么用这套组合？因为压测产生的数据量极大——成千上万个 Agent，每秒上报多条采样，直接写数据库会打爆。Kafka 削峰、ES 支持聚合查询，这是标准做法。

再往执行侧走：**VIP** 是统一入口，后面挂多个 **Agent**，最后一个环节是**控制器**。控制器负责**同步启动**——所有 Agent 必须在同一时刻开始发压，否则并发数上不去。这就是上一页说的"统一调度"的具体实现。

这一页给我们的启发是：**压测平台本质上是一个分布式任务系统**，它要解决的是"调度、采集、计算、存储"四个通用问题。理解了这一点，你再看任何压测平台（包括后面的 Takin），都能快速定位它的模块。

还有一句实践提醒：**平台自身的容量也要评估**。几万个 Agent 同时上报，采集服务扛不住，你看到的数据就是失真的。

板书：从左到右画链：压测人员 → 前端页面 → 脚本仓库／数据库 → 任务调度服务 → VIP → Agent×N（控制器同步）→ 结果采集服务 → Kafka → 实时计算服务 → ES。

提问：为什么"实时计算"这一环不能省，等压测结束再统一算行不行？

预设回答：

- 答"不行，压测要动态调整"——正是核心，看到曲线异常必须马上决策。
- 答"因为要保护生产"——对，实时看到水位才能及时停。
- 答"事后算也可以，但会错过止损时机"——这个回答最平衡，事后分析仍需要，但实时是刚需。
- 收口：所以压测平台的能力指标里，**"秒级反馈"**比"能压多高"更实用。

易错点 / 考点：① 平台内部四大模块（调度、采集、实时计算、存储）是常考填空；② 易错点：把 Kafka／ES 当成"压测工具"——它们是**数据通道组件**；③ 常考判断"压测结果可以压完再统一分析"——错，动态调整依赖实时数据。

过渡：平台能发压了，但发什么数据？压测数据的准备，是全链路压测里最容易被低估的一环。

【7.128】压测数据处理过程（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲压测数据的处理过程。很多人以为压测数据就是"随便造一批用户和商品"，其实不是——**数据的真实性直接决定压测结论的有效性**。

图里给的流程是这样：**线上流量数据**先经过**脱敏、筛选、排序**，再进**数据修正模型**做加工，然后过**数据验证模块**，最后形成**压测数据池**，再"存储、分发"给压力机集群和压测平台。右侧还画了真实的系统构成：**网关、Nginx、应用、大数据平台、核心交换机、MQ、Redis、网盘**，并注明了"MQ 是消息队列，Redis 代表缓存"。

我逐个环节讲它的必要性。

脱敏：生产数据里有手机号、身份证、地址、金额，必须脱敏才能用于压测。这不只是合规问题，也是安全问题——压测环境的防护等级通常低于生产。

筛选：不是所有线上数据都适合做压测输入。要挑出"有代表性的"——活跃用户、真实存在的商品、有效的优惠券。用一批已注销用户的数据压测，等于测了个寂寞。

排序：这一条容易被忽略。排序的目的是**制造真实的数据倾斜**——真实流量里 20% 的商品承载 80% 的访问，如果你均匀分布地压，热点问题永远暴露不出来。

数据修正模型：把脱敏、筛选后的数据调整成"压测需要的规模"，比如要把 10 万用户的量级放大到 100 万，就得按规则生成新数据，同时保持分布特征。

数据验证模块：检查这批数据"能不能用"——字段完整吗、关联关系对吗、会不会触发业务校验失败。**没验证过的压测数据，压出来的错误率是假的。**

压测数据池：最终产物，统一存储、按需分发到各个压力机。为什么要分发？因为压力机在地域上分散，让它们就近取数据，避免数据下载成为瓶颈。

这一页的落点是一句话：**压测数据不是"造数据"，而是"复制真实世界的分布"。**

> **板书：**流程链：线上流量数据 → 脱敏 → 筛选 → 排序 → 数据修正模型 → 数据验证模块 → 压测数据池 → 存储/分发 → 压力机集群＋压测平台。右侧标组件：网关・Nginx・应用・大数据平台・MQ・Redis。

> **提问：**为什么"排序"这个环节会影响压测结论的有效性？

> **预设回答：**

- 答"因为要模拟热点"——正是，真实流量高度倾斜，均匀压测测不出热点。

- 答"因为缓存命中率会不一样"——很专业，热点集中时缓存命中高，均匀分布时缓存穿透严重。

- 答"因为数据库行锁会不一样"——也对，热点商品会产生锁竞争，这是最典型的瓶颈。

- 收口：所以压测数据必须保留**分布特征**：热点比例、读写比、参数分布，三样都要像真的。

> **易错点 / 考点：**① 处理流程的六个环节常考排序题；② 易错点：把"脱敏"只当合规要求——它同时是**压测环境的安全要求**；③ 常考简答"压测数据为什么要脱敏和验证"，答题分两条：安全合规、结论有效性。

> **过渡：**数据准备好了，脚本呢？如果靠人一个个写接口，成本太高。业界更常见的做法是——把真实流量录下来，再放回去。

【7.129】流量录制回放的过程（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲流量录制与回放，它是"让压测脚本自动长出来"的一种方法。图里分成两块：**录制部分**和**回放部分**。

录制部分：通过**分光分流**和 **Agent** 把真实流量采下来，经过 **GoReplay** 这类工具处理，落到**云盘**或者直接进**压测平台**。图里还画了 **MQ** 和**缓存**，说明采集的不只是 HTTP 请求，也可能包括消息。

关键词是**分光分流**。它指的是在网络设备上把流量复制一份出来，**不影响原始请求**——这一点很重要，因为录制绝不能拖慢生产。相比之下，Agent 方式是在应用进程里埋点采集，信息更全（能拿到方法级参数），但对应用有侵入。

回放部分：把录下来的流量按需重新发出去。图里还标了"**手工上传**"，说明除了自动录制的流量，也可以人工准备一批请求导入平台。

录制回放的价值有三个。

第一，脚本自动生成。几百个接口的脚本，人工写要几周，录一遍就有了。

第二，参数真实。录下来的是真实用户的参数组合，覆盖了各种边界。

第三，能复现线上问题。线上出现异常时，把那段流量录下来在测试环境回放，是定位问题的高效手段。

但它有三个必须注意的限制，我强调一下。

限制一：录制流量不能直接放大回放。原样回放只是"重演"，要压测就得放大倍数，一放大就出现了"同一批订单重复提交""同一批优惠券重复使用"的问题。所以回放前必须做**参数化改造**：把幂等相关的字段替换掉。

限制二：写操作的流量要小心。录制时是真实下单，回放时如果打到真实库，就重复下单了。必须走影子库。

限制三：流量有"过期"问题。录的是上周的流量，这周商品下架了、接口改了，回放会大量报错，需要定期重录。

一句话：**录制回放解决"脚本从哪来"，但解决不了"数据隔离"和"参数唯一性"，这两件事还得靠平台。**

> **板书：**上半"录制"：分光分流／Agent → GoReplay → 云盘／压测平台（旁路复制不影响原请求）；下半"回放"：压测平台 → 被测应用；旁注"手工上传"。右下三条限制：需参数化・写操作走影子库・流量会过期。

提问：为什么"录制下来的流量不能原样重复回放一百万次"？

预设回答：

- 答"因为订单号会重复"——最直接，业务校验会挡住，压出来的错误率是假的。
- 答"因为优惠券只能用一次"——对，这类幂等约束到处都是。
- 答"因为要放大并发，原样回放只是重演"——很到位，回放与压测是两个目的。
- 收口：所以回放前必须做**参数改写**：唯一键、时间戳、金额、状态，该换的换，该算的算。

易错点 / 考点：① 录制与回放两部分的组件常考连线；② 易错点：认为"录制回放等于压测"——它只是**脚本与参数的来源**；③ 常考简答"录制回放的局限"，答题要提参数唯一性、数据隔离、流量时效。

过渡：工具链都有了，但全链路压测从来不是一个技术动作，而是一次跨团队的项目。我们看它的完整流程。

【7.130】全链路压测全流程（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是全链路压测的全流程，课件把它分成三个阶段：**准备阶段、校验阶段、实施阶段**。这是本节最该背下来的一页，因为它是"做事的方法"，不是"工具的参数"。

准备阶段，七件事：**被压系统范围界定**——哪些系统在链路上、哪些不在，边界必须先画清；**场景、流量、**

方案设计评审——压什么场景、灌多少流量、怎么灌，要评审，不能一个人拍；**基础数据构造**——商品、促销、优惠券这些业务基础数据；**脚本、模型创建调试**——脚本要跑通，流量模型要调对；**压测方案配置**——场景创建、资源准备；**压测数据分发**——把数据推到各压力机；**发布压测实施计划**——时间、人员、联系机制、应急预案。

注意这七件事里有三件是"非技术"的：评审、计划、范围界定。这说明什么？说明全链路压测失败的原因，多数不是技术不够，而是**组织没到位**。

校验阶段，三件事：**小流量验证**——先灌一点点流量，确认链路通、影子库生效、监控正常；**线上压测流量验证**——确认流量标识被正确识别、数据真的落到影子库；**机房流量切换**——验证切换过程中的可用性。这三件事的共同目的只有一个：**在放大流量之前，确认"不会出事"**。

实施阶段，四件事：**压测执行、动态调整流量**——一边压一边看曲线，该加加该减减；**应用降级演练**——顺便验证降级开关有没有效；**机房流量恢复**——把流量切回去；**压测报告生成**——出结论、给容量建议。

我要特别点出"**动态调整流量**"和"**降级演练**"这两项。前者说明压测是**探索式**的，不是"设定 10 万并发然后等结果"；后者说明压测不只是"测容量"，还是"**验证预案**"的时机——平时不敢碰的降级开关，压测时正好拉一下。

这一页的方法论价值，一句话：**全链路压测是"准备占七成、执行占三成"的工作。**

板书：三阶段竖排。准备（范围界定・方案评审・基础数据・脚本模型・方案配置・数据分发・实施计划）｜校验（小流量验证・流量验证・机房切换）｜实施（执行与动态调整・降级演练・流量恢复・报告）。右侧写"准备七成，执行三成"。

提问：为什么"小流量验证"这一步不能省，直接上大流量不是更快吗？

预设回答：

- 答"怕把生产打挂"——正是，先小流量是控制爆炸半径。

- 答"要确认影子库真的生效"——非常关键，流量标识没生效，大流量一上去就污染真实数据。

- 答"要确认监控能看到"——对，看不到就没法判断该不该继续。

- 收口：所以小流量验证验的是三件事：**链路通、隔离生效、监控可见**。三件都确认了，才允许放大。

易错点 / 考点：① 三阶段及其子项是高频简答／排序题，注意"小流量验证"和"降级演练"容易被漏；② 易错点：把"压测执行"当成整个流程——准备阶段的工作量最大；③ 常考判断"压测就是设定并发数然后跑"——错，

它是"验证—放大—调整"的探索过程。

【思政落点 8】：全链路压测这一页，最能体现"严谨"两个字。它要求我们先界定范围、再评审方案、再造数据、再小流量验证，最后才敢放大流量；压的过程中要盯着曲线随时调整，压完还要出报告、给容量建议。这一整套流程看起来很"慢"，但每一条都是用事故换来的。我想说的是：**面对真实规模，任何"应该没问题"的直觉都不可靠**。工程师的可信度不来自自信，而来自"我把每一步都验证过了"。请把这种"面向真实、逐步放大、留好退路"的习惯，带进你们以后做的每一个系统。

过渡：流程讲完，最后看平台应该提供哪些能力，才算"够用"。

【7.131】压测平台提供的整体能力（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页把压测平台应该提供的能力分成六块，从"发什么"到"得出什么结论"，形成一条完整的能力链。我按这条链讲。

第一块，压测模型。 这一块包含四种发压方式：**动态发压**（压测过程中随时调整流量大小）、**峰值模拟**（模拟真实流量的尖峰曲线，而不是恒定压力）、**流量回放**（把录制的真实流量按倍数放出去）、**智能寻值**（自动试探系统的容量上限，一步步加压直到找到拐点）。前两个解决"像不像"，后两个解决"省不省力"。

第二块，压测控制。 分**人工**和**自动**两种。人工就是测试人员看着曲线手动调，自动就是按预设策略跑。为什么要保留人工？因为首次压测时你不知道系统的脾气，必须人盯着；等摸清了，就可以自动化。

第三块，数据采集。 采六类数据：**压测数据**（TPS、响应时间、错误率）、**日志数据**、**链路数据**（每一跳的耗时）、**资源数据**（CPU、内存、IO、网络）、**调用数据**（下游依赖的调用量与耗时）、**健康状态**。为什么这么全？因为**只压不采等于白压**——你只知道"慢"，不知道为什么慢。

第四块，智能分析。 四个产出：**水位告警**（哪个资源快满了）、**结论预研**（这次压测大概能得出什么结论）、**容量预期**（按当前配置能扛多少）、还有对上面六类数据的综合分析。这一块的价值是把"数据"变成"判断"。

第五块，评估策略。 四个层级：**单点容量**（单个服务）、**场景容量**（一条业务场景）、**集群容量**（整个集群）、**链路容量**（全链路）。分层评估的意义是：出问题时能定位到"是哪一层不够"。

第六块，评估结论。 三样交付物：**评估报告**、**优化方案**、**风险建议**。注意后两个——压测的目的不是"给个数

字"，而是"告诉你要改什么、有什么风险"。

这一页的整体逻辑：**平台的价值不在于"能压多高"，而在于"能把结果变成决策"。**

> **板书：**六块横向排：压测模型（动态发压・峰值模拟・流量回放・智能寻值）→ 压测控制（人工/自动）→ 数据采集（压测・日志・链路・资源・调用・健康）→ 智能分析（水位告警・结论预研・容量预期）→ 评估策略（单点・场景・集群・链路）→ 评估结论（报告・优化方案・风险建议）。

提问：为什么"只压不采"的压测几乎没有价值？

预设回答：

- 答"因为不知道瓶颈在哪"——正是核心，只知道 TPS 低，不知道该改哪里。

- 答"因为没法定位到具体服务"——对，链路数据才能告诉你哪一跳慢。

- 答"因为下次还得从头摸"——很实际，没有数据沉淀就无法对比。

- 收口：所以压测的产出应该是**"数字 + 归因"**，只有数字的压测报告是不合格的。

易错点 / 考点：① 六块能力与各自子项是常考多选／连线，注意"智能寻值"和"水位告警"容易被漏；② 易错点：把"压测平台"理解成"压力机集群"——前者是管理平台，后者只是执行端；③ 常考简答"压测报告应包含什么"，答题要提"容量结论、瓶颈归因、优化建议、风险提示"。

过渡：能力清单讲完，我们看一个具体的实现——开源的 Takin。

【7.132】现在来看开源的 Takin（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页介绍 Takin，中文名"羚牛"。课件给的定义是：**Takin 是基于 Java 语言开发的一套生产全链路压测的系统，可以在无业务代码侵入的情况下，嵌入到各个应用程序节点，实现生产环境的全链路性能测试，适用于复杂的微服务架构系统。**

这句话里有四个关键点，我逐个讲，因为每一个都对应一个工程难题。

第一，"基于 Java"。这意味着它的接入方式是 Java Agent——通过字节码增强在运行时织入逻辑，所以你不用改代码。这也决定了它的适用范围：Java 微服务系统。

第二，"无业务代码侵入"。课件专门列了一条：**业务代码 0 侵入：在接入、采集和实现逻辑控制时，不需要修改任何业务代码。**为什么这件事重要？因为如果每接入一个系统都要改代码，那全链路压测就推行不下去——几十个团队、上百个服务，改一遍要一年，还得每次发版都维护。**0 侵入是"能落地"的前提。**

第三，"数据安全隔离"。课件的原话是：**可以在不污染生产环境业务数据情况下进行全链路性能测试，可以在生产环境对写类型接口进行直接的性能测试。**注意最后半句——**可以对写接口直接压测**。这是全链路压测的分水岭：很多方案只能压读接口，因为写操作会脏数据；能做到安全地压写接口，才算真正能验证下单、支付这类核心链路。

第四，"安全性能压测"和"性能瓶颈快速定位"。前者指在生产环境压测不影响业务，后者指压测结果能直接展现业务链路中性能瓶颈的节点。

这四条其实是递进关系：**0 侵入让它能接进来，数据隔离让它敢在生产跑，安全压测让它敢压写接口，瓶颈定位让它压完有用。**四条缺一条，这套方案就不完整。

最后一句提醒：Takin 是开源项目，课件给了 GitHub 地址。开源的价值不只是"免费"，更在于**你能看到它怎么解决影子库、怎么打流量标、怎么织入字节码**——这些设计思路比工具本身更值得学。

板书：四个卖点竖排：Java（Agent 字节码增强）→ 0 侵入（不改业务代码）→ 数据隔离（可压写接口）→ 安全压测＋瓶颈定位。右侧写"递进关系：能接入 → 敢生产 → 敢写 → 有用"。

提问：为什么"能对写接口做压测"这件事，比"能压到很高的 TPS"更能说明一个压测方案成熟？

预设回答：

- 答"因为读接口压测不真实"——对，真实业务一半以上是写操作。

- 答"因为写操作会脏数据，需要影子库"——正是技术难点所在，涉及中间件改造。

- 答"因为下单链路才是核心"——很到位，压测的价值就在验证交易主链路。

- 收口：所以判断一套全链路压测方案，先问"写接口敢不敢压"，再问"能压多高"。

易错点 / 考点：① Takin 的四个特点（0 侵入、数据隔离、安全压测、瓶颈定位）是标准考点；② 易错点：把"0 侵入"理解成"不需要部署"——它仍需部署 Agent，只是不改业务代码；③ 常考判断"全链路压测只能压读接口"——有了影子库机制，写接口同样可以压。

过渡：四个卖点背后是一套具体架构。下一页看它的组成。

【7.133】Takin 架构（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 Takin 的架构图。我不逐框念，而是带你抓四条主线，因为任何全链路压测平台的架构，都跑不出这四条。

第一条主线：控制面。 平台需要一个"大脑"，负责管理应用、配置影子库表、配置挡板与白名单、编排压测场景、下发任务。这一层是 Web 控制台加后端服务，是人和平台交互的地方。

第二条主线：数据面。 压测流量要在中间件层面被识别和改写——数据库访问被导向影子库、缓存访问被导向影子缓存、消息被加上标记。这就是"数据隔离"的实现位置。注意关键点：**改写发生在中间件/客户端 SDK 层，而不是业务代码里**，这才叫 0 侵入。

第三条主线：采集面。 各个应用节点上的 Agent 采集链路数据、调用数据、资源数据，汇总回平台。没有这一层，压测就只有"结果"没有"归因"。

第四条主线：执行面。 压力机按场景和流量模型发压，并把结果回传。

这四条主线对应四个问题：**谁指挥、数据去哪、怎么看见、谁来压。** 你去看任何一个压测平台——商业的、开源的、自研的——都能这样拆。

这一页还有一个隐含的设计思想值得说：**Takin 把"压测"做成了"平台 + Agent + 中间件增强"的组合**，而不是一个大一统的单体系统。这样每个应用只要接入 Agent，就能参与全链路压测，扩展性好。代价是**对中间件版本有要求**，你需要确认自己的数据库连接池、缓存客户端、消息客户端在支持列表里。

所以评估这类平台时，第一个要问的问题不是"它能压多高"，而是"**我的技术栈它支持吗**"。

> **板书：**四条主线竖排：控制面（应用/影子库表/挡板/白名单/场景）｜数据面（中间件增强 → 影子库・影子缓存・消息标记）｜采集面（Agent → 链路・调用・资源）｜执行面（压力机发压回传）。下方写"谁指挥・数据去哪・怎么看见・谁来压"。

提问：为什么这类平台的"技术栈支持范围"，往往比"性能上限"更影响落地？

预设回答：

- 答"因为不支持就接不进去"——最直接，中间件不在列表里，影子库就打不通。

- 答"因为改造中间件成本极高"——对，很多团队宁可放弃方案也不改中间件。

- 答"因为业务代码不能动"——很关键，0 侵入的前提就是中间件层能兜住。

- 收口：所以选型的第一张表是"兼容性矩阵"，第二张才是"性能指标"。

> **易错点 / 考点：** ① 架构的四个层面（控制、数据、采集、执行）是常考简答；② 易错点：把"影子库"当成"另建一个数据库"——它是**同一套逻辑下的隔离存储**，靠流量标记路由；③ 常考判断"全链路压测平台与业务代码强耦合"——0 侵入方案正是为了解耦。

> **过渡：** 架构讲完，看一眼它实际长什么样。

【7.134】Takin 界面截图（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 Takin 的控制台界面。界面本身不用记，但界面上有哪几类信息，直接对应它的能力，值得一看。
>
> 我把这类压测平台的界面归纳成五个区域，你对照着看就明白了。
>
> **第一个区域：应用与接入管理。** 有哪些应用接入了平台、每个应用的 Agent 状态如何、版本是否一致。这一块是"能不能压"的前提。
>
> **第二个区域：配置管理。** 影子库表配置、挡板配置、白名单配置都在这里。这三样是数据隔离的三件套：影子库管"数据写到哪"，挡板管"某些下游调用要不要拦掉"，白名单管"哪些流量不参与压测"。**配置错了，压测就会污染生产，或者压不出真实结果。**
>
> **第三个区域：场景与脚本。** 一个"场景"描述一次压测：压哪些接口、按什么比例、用多少并发、跑多久。脚本则描述单个请求怎么发。
>
> **第四个区域：任务与执行。** 发起压测、查看进度、动态调整流量、紧急停止。这里最重要的是那个"停止"按钮——它是最后一道保险。
>
> **第五个区域：结果与报表。** TPS 曲线、响应时间分位数、错误率、链路耗时分布、瓶颈节点。这一块是压测的产出。
>
> 看界面的时候请建立一个观念：**压测平台的界面，本质上是"风险控制面板"**。你在这里做的每一个配置，都在回答"我怎么保证这次压测不出事"。所以别把它当"点点按钮就出报告"的工具，要当"操作一台有风险的机器"。

还有一点很实际：**第一次用这类平台，先在测试环境把整套流程跑通**——应用接入、影子库建好、挡板配好、小流量验证通过，再考虑生产环境。跳过这一步，风险不可控。

板书：五个区域：应用与接入｜配置（影子库表・挡板・白名单）｜场景与脚本｜任务与执行（含紧急停止）｜结果与报表。下方写"界面＝风险控制面板"。

提问：为什么"白名单配置"在生产环境压测里是必需的?

预设回答：

- 答"要把某些真实用户排除掉"——对，避免压测流量影响真实交易。

- 答"要排除某些特殊接口"——也对，比如对账、清算这类不能重复执行的接口。

- 答"要保证压测流量能被识别"——很专业，白名单和流量标识经常配合使用。

- 收口：白名单的本质是"**划一条线，线内是压测，线外是真业务**"，线划不清就别压。

易错点 / 考点：① 三件套配置（影子库表、挡板、白名单）的作用是常考简答；② 易错点：认为"平台有界面所以很安全"——风险取决于配置，不取决于工具；③ 常考判断"压测可以直接在生产环境开始，不需要测试环境预演"——错。

过渡：它从哪来、怎么用？课件给了开源地址。

【7.135】Takin 开源介绍（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页很简单，只给了一个地址：Takin 的 GitHub 仓库。但我想借这一页讲三件事，因为"怎么用开源项目"本身就是一项能力。

第一，开源项目的价值在"可读"。 你拿到 Takin 的代码，能看到它怎么实现流量标记、怎么改写数据源、怎么在字节码层面织入、怎么设计影子表的命名规则。这些东西在文档里通常只有一句话，在代码里是完整的设计。**学压测平台的正确方式，是先读架构，再读关键实现，最后才跑起来。**

第二，跑起来的方式要正规。 一般步骤是：看 README 了解模块划分和部署要求 → 用 Docker Compose 或官方脚本起一套最小可用环境 → 按文档接入一个示例应用 → 跑通一次最简单的压测。**不要一上来就想着接自己的生产系统**，先在示例上把链路走通。

第三，评估开源项目要看四个维度：活跃度（最近提交、issue 响应）、**文档质量**（有没有部署与接入指南）、**依赖复杂度**（要装多少东西、对中间件版本有什么要求）、**可维护性**（代码结构清不清楚、能不能自己改）。这四条决定了它能不能真的用在项目里。

我要提醒一句：**开源不等于零成本**。你要投入人力去部署、调试、改造、维护，遇到问题时只能自己查。所以选开源还是选商业，本质是"用钱换时间"还是"用时间换钱"。

还有一条与课程直接相关的建议：**把 Takin 当成"读源码学架构"的样本**，而不是必须部署成功的任务。你们的时间有限，读它的架构设计、理解影子库与流量标记的实现思路，收获比装一遍更大。

板书：三个层次：可读（流量标记・数据源改写・字节码织入・影子表命名）→ 可跑（README → 最小环境 → 示例应用 → 单场景压测）→ 可评（活跃度・文档・依赖・可维护性）。下方写"开源≠零成本：用时间换钱"。

提问：如果让你评估"这个开源压测平台能不能用在我们的项目上"，你会先看哪三样东西？

预设回答：

- 答"看它支不支持我们的中间件"——第一优先，接不进去一切免谈。

- 答"看社区还活不活跃"——很重要，停更的项目接手风险大。

- 答"看部署复杂度"——务实，装不起来就用不了。

- 收口：三样合起来就是"**能接入、有人维护、装得起来**"，这三条过了再谈功能。

易错点 / 考点：① 这一页本身知识点少，但常与"开源工具评估"结合出简答；② 易错点：把"开源"等同于"免费可用"——忽略部署与维护成本；③ 记住 Takin 的仓库地址作为课程设计参考资料即可，不必背具体路径。

过渡：真要用起来，得先做完一套配置。课件把配置项列成了七项。

【7.136】Takin 使用（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页列了 Takin 使用时要做的七项配置，课件还给了两篇参考文章。我把这七项按"先后顺序"重排，并说明每一项在解决什么问题——因为**配置顺序错了，后面全错**。

第一，应用配置。 把要参与压测的应用接入平台：应用名、部署节点、Agent 版本。这是"哪些系统在链路上"的落地。

第二，链路梳理。 这一项最容易被跳过，但最关键。它要回答：一次请求经过哪些应用、每个应用调用哪些下游、哪些是同步调用哪些是异步消息。**没有链路图，压测结果就没法归因**——你只知道整体慢，不知道慢在哪一跳。

第三，影子库表配置。 指定压测数据写到哪些库、哪些表。通常做法是"同实例不同库"或"同库不同表前缀"，让压测数据和生产数据物理隔离但结构一致。**这是数据隔离的核心配置。**

第四，挡板配置。 挡板（Mock）用来拦住那些"不能真的执行"的下游调用——比如真实的支付网关、真实的短信通道、真实的外部对账。压测时让这些调用返回预设结果，既避免真实副作用，又能控制变量。

第五，白名单配置。 划清"哪些流量参与压测、哪些不参与"。典型用法：只让带压测标记的流量走影子库，其余走正常路径。

第六，脚本配置。 压测脚本：请求怎么发、参数从哪来、断言是什么。脚本质量决定压测的"像不像"。

第七，场景配置。 把脚本、流量模型、并发策略、持续时间组合成一个可执行的场景。

这七项里，**链路梳理和影子库表配置是最容易出事的两个**：前者影响结论有没有用，后者影响生产会不会被污染。我的建议是把这两项单独做评审。

另外提醒一句：课件给的两篇参考文章是官方社区的实践文章，读的时候重点看"他们踩了哪些坑"，比看操作步骤更有价值。

> **板书：**七项按序：应用配置 → 链路梳理 → 影子库表 → 挡板 → 白名单 → 脚本 → 场景。右侧标两项高危：链路梳理（影响归因）·影子库表（影响数据安全）。

> **提问：**为什么"挡板配置"在压测里是不可少的，而不是"能省则省"？

> **预设回答：**

- 答"因为有些下游不能真调"——正是，支付、短信、外部对账，真调会产生真实副作用。
- 答"因为要控制变量"——很专业，把外部不稳定因素挡掉，才能测出自己系统的问题。
- 答"因为下游扛不住压测流量"——也对，第三方接口通常有严格限流。
- 收口：挡板的作用是"**把不可控变成可控**"，它是压测实验设计的一部分。

易错点 / 考点： ① 七项配置是常考多选；② 易错点：把"链路梳理"当成"画个图看看"——它是归因的基础；③ 常考简答"挡板的作用"，答题要提"避免真实副作用""控制变量""保护下游"。

过渡： Takin 自己的脚本体系有门槛，所以它还提供了另一条路——和 JMeter 集成。

【7.137】与 JMeter 集成 -1（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲 Takin 与 JMeter 的集成。为什么要集成？道理很直接：**JMeter 已经是业界最普及的压测工具，团队的学习成本最低；而 Takin 提供的是全链路压测需要的"平台能力"——影子库、流量标记、链路采集。**把两者接起来，就能用熟悉的 JMeter 写脚本，用 Takin 的平台能力做全链路压测。

这个组合解决了三个现实问题。

第一，脚本资产复用。 团队过去几年积累的 JMeter 脚本不用重写。这一条对落地极其重要——工具替换的最大阻力从来不是功能，而是"我以前的脚本怎么办"。

第二，学习成本低。 压测人员不用学一套新的脚本 DSL，还是 JMeter 的线程组、取样器、断言、定时器。

第三，能力互补。 JMeter 强在"发压与脚本编排"，弱在"生产环境的数据隔离与全链路追踪"；Takin 强在后者。两者结合，正好互补。

集成的技术形态通常是：**Takin 提供一个 JMeter 的插件或后端监听器**，让 JMeter 发出的请求带上压测标记，并把结果回传给 Takin 平台。这样平台就能知道"这批流量是压测流量"，从而路由到影子库；也能收集到压测结果，做汇总分析。

从学习方法上说，这一页给了一个通用思路：**做平台不要什么都自己造。** 发压引擎这种成熟度极高的部件，直接复用开源生态；平台把力气花在"隔离、标记、采集、分析"这些真正难的地方。这也是为什么后面很多企业自研压测平台时，底层都还是 JMeter 或 Gatling。

一句话：**JMeter 解决"怎么发"，Takin 解决"发到生产怎么不出事"。**

板书： 写"JMeter（发压＋脚本编排）＋ Takin（隔离＋标记＋采集＋分析）"；下方三条收益：脚本复用・学习成本低・能力互补；技术形态：JMeter 插件／后端监听器 → 带标记 → 回传平台。

提问： 为什么企业自研压测平台时，很少自己从头写一个发压引擎？

预设回答：

- 答"没必要，JMeter 已经够用"——对，发压引擎的成熟度很难超越。

- 答"自研成本太高，收益低"——很准确，这是典型的"非核心竞争力"。

- 答"自研的价值在隔离和调度"——正是，钱要花在难的地方。

- 收口：技术选型的原则是**复用成熟部件，自研差异化能力**。

易错点 / 考点： ① 集成的动机（脚本复用、学习成本、能力互补）是常考简答；② 易错点：认为"集成后 JMeter 就能做全链路压测"——全链路能力来自 Takin 的平台侧；③ 常考判断"发压引擎必须自研才能满足需求"——多数情况下不成立。

过渡： 集成的细节，课件还留了一页参考。

【7.138】与 JMeter 集成 -2（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是集成的第二部分，课件给了一个 README 地址，说明具体做法要看官方文档。我借这一页讲"读技术文档"的方法，因为这是你们做课程设计时最需要的能力。

读这类集成文档，按四步走。

第一步，先看"前置条件"。 需要什么版本的 JMeter？需要装什么插件？Takin 平台要先配好什么？**90% 的集成失败都发生在这一步被跳过。**

第二步，看"数据流"。 请求从 JMeter 发出去，中间经过谁，标记加在哪里，结果怎么回传。把这条链画出来，你就理解了集成的本质——它无非是"加标记"和"回传结果"两件事。

第三步，看"配置项清单"。 哪些参数必填、哪些有默认值、哪些填错会静默失效。特别注意"静默失效"这类——配置错了不报错，压测跑完了你才发现数据没进影子库。

第四步，看"验证方法"。 文档里通常会写"如何确认集成成功"，比如查一条日志、看一个页面、对一次数据。**没有验证步骤的集成，等于没集成。**

这四步不只适用于 Takin，适用于你以后读的任何技术文档。

还有一个观念要建立：**集成类工作的风险主要在"边界"上**——JMeter 的版本边界、Takin 的版本边界、中间件的版本边界。三个边界两两组合，就是你要验证的矩阵。所以做集成测试时，先列版本矩阵，再动手。

最后说一句和课程作业相关的话：如果你们的课程设计要演示全链路压测，**不必真的搭一套 Takin**。用 JMeter 压一个由两三个服务组成的链路，加上"流量标记 + 影子表"的简化实现，把原理讲清楚，比部署一套完整平台更有价值。**课程设计考的是理解，不是部署能力。**

> **板书：**读文档四步：前置条件 → 数据流（标记加在哪・结果怎么回）→ 配置项（注意静默失效）→ 验证方法。下方写"风险在边界：JMeter 版本 × Takin 版本 × 中间件版本"。

提问：为什么"配置错了但不报错"比"配置错了一启动就失败"更危险？

预设回答：

- 答"因为你会以为成功了"——正是，压测跑完才发现数据没隔离，那时已经污染了生产。

- 答"因为排查成本高"——对，没有报错就没有线索，只能靠对比数据发现异常。

- 答"所以要有验证步骤"——这就是第四步存在的意义。

- 收口：判断一个系统做得好不好，看它**出错时是否容易被发现**——这叫"可观测性"。

易错点 / 考点：① 这一页偏方法，考点常落在"集成测试要验证什么"；② 易错点：只看"能不能跑通"，不看"数据是否正确隔离"；③ 常考判断"集成成功＝压测能发出请求"——错，还要验证标记生效与结果回传。

过渡：集成好了，压出来的数据怎么看？最后两页讲结果展示。

【7.139】压测结果展示（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲压测结果怎么展示。很多人以为"结果展示"就是把图表画漂亮，其实它的核心任务是**让人在最短时间内做出判断**。

一份合格的压测结果，要能回答四个问题，对应四类视图。

第一类：系统整体扛住了吗？ 看 TPS 曲线和错误率曲线。判断标准不是"峰值多高"，而是**在目标 TPS 下，错误率是否为零、响应时间是否在 SLA 内**。TPS 冲到很高但错误率 30%，那不叫扛住，那叫雪崩前的一瞬间。

第二类：响应时间分布如何？ 必须看**分位数**（P90、P95、P99），不能只看平均值。平均值是最会骗人的指标——90% 的请求 50 毫秒，10% 的请求 5 秒，平均值可能只有 500 毫秒，但用户体验是"十个人里有一个卡死"。

第三类：瓶颈在哪？ 看链路耗时分布——每一跳服务、每一次数据库调用、每一次缓存访问的耗时占比。**找到占比最高的那一跳，就是优化目标。**

第四类：资源水位如何？ CPU、内存、IO、网络、连接池、线程池。注意一个常见现象：应用 CPU 才 40%，但 TPS 上不去了——这时瓶颈往往在**连接池满、线程池满、下游限流**这些"看不见的队列"上。

展示形式上，好的压测报告通常有一条主线：**结论先行**（能不能扛住目标流量）→ **证据支撑**（曲线与分位数）→ **瓶颈归因**（链路与资源）→ **优化建议**（扩哪里、改什么、预期收益）。

我要强调"结论先行"这四个字。**报告的第一页就该给出判断**，而不是让人翻二十页图表自己总结。这是工程沟通的基本素养。

板书：四问四视图：扛住了吗（TPS＋错误率）｜体验如何（P90/P95/P99）｜瓶颈在哪（链路耗时占比）｜水位如何（CPU・内存・IO・连接池・线程池）。下方写报告主线：结论先行 → 证据 → 归因 → 建议。

提问：为什么压测报告里"平均响应时间"这个指标的说服力很弱？

预设回答：

- 答"因为被极端值拉平了"——正是，平均值掩盖长尾。

- 答"因为用户体验看的是最慢的那部分"——对，卡一次就记住了。

- 答"因为 SLA 通常按分位数定"——很专业，SLA 写的是"P99 小于 500ms"这类表述。

- 收口：所以看性能必须看**分布**，平均值只能当"整体量级"的粗略参考。

易错点 / 考点：① 分位数（P90/P95/P99）的意义是高频考点，要能解释"为什么不用平均值"；② 易错点：只看 TPS 峰值不看错误率；③ 常考简答"如何定位性能瓶颈"，答题要提"链路耗时分布 + 资源水位 + 队列类指标"。

过渡：结果展示还有续页，讲的是"压测概览"这个汇总视图。

【7.140】压测结果展示 – 续（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是结果展示的续页，核心词是**压测概览**。概览的作用是把一次压测的关键信息压缩到一屏，让没参与压测的人也能快速理解。

一个合格的压测概览，我建议包含六项内容。

第一，压测基本信息。 什么时间压的、压了多久、压的是哪个场景、用了多少并发、目标 TPS 是多少。**没有这些信息，任何数字都无法解读**——"8000 TPS"是好是坏，取决于目标和环境。

第二，关键指标汇总。 实际达到的 TPS、成功数／失败数、错误率、响应时间分位数、最大响应时间。这一屏要能让人一眼看出"达到目标没有"。

第三，SLA 达成情况。 把实际值与事先约定的指标做对比，逐条打勾或打叉。这一项最能体现专业性——**压测不是"看看能跑多快"，而是"验证是否满足承诺"**。

第四，瓶颈排序。 按耗时占比或资源占用排序，列出前几个瓶颈节点。这一项决定了压测之后要干什么。

第五，异常与告警。 压测过程中触发了哪些告警、有没有出现降级、有没有错误日志突增。这些信息往往比平均指标更有价值。

第六，结论与建议。 一句话结论（能扛／不能扛／有条件能扛）加上具体建议（扩容多少、优化哪个接口、调整哪个参数）。

这一页还有一个观念要传达：**概览不是"简化版报告"，而是"决策界面"**。它的读者可能是不懂技术的业务负责人，也可能是要排期改代码的开发负责人——两类人看同一屏，都要能拿到自己需要的信息。

所以在做课程设计时，如果你的项目涉及性能或压测，报告里最好有这么一页概览。**这一页做得好，说明你真的理解了压测的目的。**

> **板书：**概览六项：基本信息（时间・场景・并发・目标）｜关键指标（TPS・成功率・分位数）｜SLA 达成｜瓶颈排序｜异常与告警｜结论与建议。下方写"概览＝决策界面，不是简化报告"。

> **提问：**为什么"目标 TPS"必须写在概览里，而不是放在报告后面？

> **预设回答：**

- 答"因为没有目标就无法判断好坏"——正是，8000 是高是低要看目标是多少。

- 答"因为读者要先知道结论"——对，概览就是给结论的地方。

- 答"因为目标本身也要被检验"——很专业，目标定得合不合理，也是压测的产出之一。

- 收口：所以压测的第一步是定目标（SLA），而不是先跑起来——这又回到了 7.1 讲的"从需求开始明确性能"。

> **易错点 / 考点：** ① 概览应包含的要素是常考简答；② 易错点：把"概览"理解成"几张图"——它是**结论页**；③ 常考判断"压测报告越长越专业"——错，结论先行、证据可查才是专业。

> **过渡：** 到这里，全链路压测这一块讲完了。再往下就是压测平台特点的小结，以及 7.4 后半段的混沌工程——那是下一片的内容。

【7.141】小结：压测平台特点（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把"压测平台"这件事收个口。前面几页我们讲的是全链路压测怎么组织、流量怎么染色、影子表怎么隔离，落到工具和平台这一层，一个能真正干活的压测平台，必须同时具备四个本事。
>
> 第一个本事是**支持协议广泛**。真实系统里不是只有 HTTP，还有 HTTPS、JSF 这类 RPC、HBase、数据库、各类中间件。压测平台如果只会发 HTTP，那它压不到落库、压不到缓存、压不到消息，测出来的结论只是半个系统的结论。这一点你们做课程设计时特别容易犯：压了半天首页，接口全绿，一压下单链路就崩，因为链路里最慢的那一段根本没被压到。
>
> 第二个本事是**多种发压模式**：动态发压、智能寻点、标准模式、梯度模式、集合点模式。为什么需要这么多？因为问题不一样，压法就不一样。要找容量上限就用梯度模式，一级一级往上加；要复现秒杀那种瞬时洪峰，就用集合点模式让一批请求同时到达；要边压边调参数，就用动态发压。
>
> 第三个本事是**资源合理分配**：资源灵活调度、资源重复使用以减少拉取时间、统筹压力机调度。压测机是稀缺资源，能不能重复用，直接决定你一天能跑几轮。
>
> 第四个本事是**多维度结果查看**：压测数据秒级反馈，动态增减压的结果自动分段展示，能算结果区间，还能把应用监控结果采集回来，和压测曲线放在一起看。

> **板书：** 四行竖排 —— **①协议广泛**（HTTP/HTTPS・JSF・HBase・数据库・中间件）→ **②发压模式**（动态／智能寻点／标准／梯度／集合点）→ **③资源分配**（灵活调度・重复使用省拉取时间・统筹压力机）→ **④**

> **结果查看**（秒级反馈・分段展示・区间计算・监控采集）。

右侧一句："压不全 = 白压。"

> **提问：**如果你们组只有一个只支持 HTTP 的压测工具，要去压"下单→扣库存→落库→发消息"这条链路，你会漏掉哪些风险？

> **预设回答：**

- 答"漏掉数据库压力"——对，而且这是最常见的一层：HTTP 接口的 QPS 上去了，数据库连接池先满。

- 答"消息队列没压到，但队列一般不会挂"——这就想当然了。异步化之后瓶颈往往从同步链路上挪到了队列消费端，消费积压会让整条链路"看起来没超时、实际没结果"。

- 答"那就用脚本自己拼协议"——可行但代价高，等于你自己在造一个平台，这正是本节讲平台特点的原因：这些能力应该由平台提供，不该每个项目重复造。

- 收口：压测的覆盖面由协议的覆盖面决定，压不到的组件等于没测。

> **易错点 / 考点：**这一页容易和"性能测试类型"混着考。**发压模式是工具能力**（压力怎么施加），**性能测试类型是测试目的**（负载、压力、稳定性、并发）。判断题"梯度模式属于压力测试"——不严谨，梯度只是一种加压方式，可以用在负载测试里，也可以用在压力测试里。答题三步法：先问"这条考的是目的还是手段"，再看选项落在哪一侧。

> **过渡：**压测能回答"系统扛得住多大流量"，但它回答不了另一个问题——某个组件**突然死掉**的时候，系统会怎样。

【7.142】混沌工程（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 从这里开始，我们进入这一章的第三个专项：**混沌工程**。
>
> 先说它和前面两节的关系。性能测试和全链路压测，问的是"压力大了会怎样"；混沌工程问的是"**东西坏了会怎样**"。这两个问题看着像，其实完全不是一回事。压测是把水龙头开到最大，看管道会不会爆；混沌工程是拿锤子照着某一根管子敲一下，看整套系统还能不能供水。
>
> 为什么这个能力现在变得特别重要？因为我们的系统已经从"一台机器上一个程序"，变成了几十上百个服务、

上千个实例、跨机房跨云的分布式系统。这种系统里，**故障不再是意外，而是常态**：硬盘会坏、网络会抖、某个实例会僵死、某个依赖会超时。你没法保证它不出故障，你只能保证"出了故障，系统仍然可用"。

所以混沌工程这个词你一定要理解准：它不是"制造混乱"，恰恰相反，它是**用可控的人造故障，去换真实故障面前的确定性**。一句话——**故意让它坏，是为了让它真的坏的时候不会塌**。

这一节我们按这条线走：先看复杂系统为什么必然出问题，再看混沌工程的定义、五项原则、关键工作和爆炸半径控制，然后看它自身的局限，最后落到工具和实践。

板书：中间写"混沌工程"；左边写"压测：加大水压 → 管道会不会爆"；右边写"混沌：敲一根管子 → 系统还能不能供水"；下面一行粗体："**故障是常态，不是意外**"。

提问：一个系统做了完整的压力测试，各项指标都达标，是不是就不需要混沌工程了？

预设回答：

- 答"是，压测都过了就稳了"——这是最典型的误判。压测验证的是"满负载下的表现"，它不验证"某个依赖突然不可用时的降级能力"。

- 答"不一样，压测里没有故障"——准确。压测的输入是流量，混沌的输入是故障。

- 答"那混沌工程是不是就是拔网线"——拔网线只是最粗的一种注入手段，混沌工程的重点不在"拔"，而在"实验设计、稳态判定、爆炸半径控制"这一整套方法。

- 收口：这两个能力是互补的，谁也不能替代谁。

易错点 / 考点：名词辨识是必考点。**故障注入是一种技术手段，混沌工程是一套工程方法**——后者包含假设、实验设计、稳态标准、爆炸半径控制、报告改进，不只是"注个故障看看"。判断题"混沌工程等于随机破坏生产环境"——错，恰恰相反，它的第一原则就是最小化影响范围。

过渡：要说清"为什么必须做"，得先看清今天这套系统复杂到了什么程度。

【7.143】从复杂到混沌（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是一张示意图，课件标注的来源是 Data Center Knowledge，它要你看出的是一个因果链：**复杂度上去了，故障的影响面就跟着放大**。

> 我们看这条链子怎么走的。系统的组件越来越多，调用关系从"三层架构"变成网状：一个用户请求进来，经过网关、认证、商品、库存、订单、支付、消息、缓存、数据库，中间还夹着服务发现、配置中心、监控探针。组件一多，**每一次请求要经过的环节就多**，而每个环节都有一个失败概率。数学上很清楚：单个环节 99.99% 可用，串上 50 个环节，整体可用性就被拉下来了。
>
> 更麻烦的是**耦合**。以前一个模块挂了，用户看到的是"这个功能不可用"；现在一个底层组件挂了，可能是全站雪崩，因为所有服务都在等它。这就是图里"影响"两个字的分量——**故障的影响半径，跟着依赖的半径一起扩大**。
>
> 还有第三个变化：**人的认知跟不上了**。系统大到一定程度，没有人能在脑子里完整推演"如果这里坏了会发生什么"。你不知道，就只能等它真的坏了才知道。而那时候代价已经发生。
>
> 这张图给我们的结论只有一句：**在复杂系统里，"不出故障"是不可承诺的，"出了故障还活着"才是可设计的**。这就是混沌工程要解决的问题。

> **板书：**一条链 —— 组件增多 → 调用链变长 → 单点失败概率累积 → 耦合放大影响面 → 人无法推演。

右侧写一个算式："0.9999 的 50 次方 ≈ 0.995"；下方红字："**故障影响半径 = 依赖半径**"。

> **提问：**如果一个系统里每个服务的可用性都是 99.99%，把它串成 50 个环节的一条链路，整体可用性大概是什么水平？

> **预设回答：**

- 答"还是 99.99%"——不会。串联系统的可用性是各环节可用性相乘，环节越多，整体越低。

- 答"大概 99.5% 左右"——方向对。这正是"每个组件都很稳、整体却不够稳"的由来。

- 答"那就少串几个环节"——思路对，这就是"减少强依赖、把同步改异步、把强依赖改弱依赖"这些架构手段的动机，也是后面讲降级和熔断的原因。

- 追问：所以可用性不是一个组件的事，是**整条链路的事**。这句话请记住，后面讲稳态标准时还要用。

> **易错点 / 考点：**这里常出计算题或判断题。记住**串联相乘、并联取"1 减都失效的概率"**这两个公式口径。另一种考法是概念题："为什么分布式系统必须做混沌工程"——答题要落到三点：调用链变长、故障不可完全避免、人工推演不可行。

> **过渡：**既然故障避不掉，那就要算一笔账——一次中断到底值多少钱。

【7.144】企业服务中断 1 小时所带来的损失（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页是算账的。课件引用 Statista 2022 的数据，来源标注为混沌工程实验室，它想让你建立的不是"记住某个数字"，而是"**可用性每提高一个 9，代价和收益都急剧变化**"这个直觉。
>
> 课件给出了四档可用性水平对应的年停机时长：**大于 87.6 小时、大于 8.7 小时、大于 52 分钟、小于 5 分钟**。你们把这四个数字和"几个 9"对一下就很清楚了：87.6 小时对应的是 99%，8.7 小时对应 99.9%，52 分钟对应 99.99%，5 分钟对应 99.999%。**每多一个 9，允许的停机时间就少一个数量级**。
>
> 这就是行业里说的"**从三个 9 到四个 9，是最贵的一步**"。为什么贵？因为 99% 到 99.9% 靠"把明显的 bug 修掉、加个监控"就能做到；99.9% 到 99.99% 就必须做冗余、做故障自动转移、做降级预案；再往上，就要靠混沌工程这种"主动验证你的冗余和预案真的有效"的手段。
>
> 再看"一小时损失"这件事。不同行业差异极大，课件给的是行业的量级差异，具体金额我不在这里念，你们记住一个判断方法：**损失 = 直接交易额 + 恢复成本 + 用户流失 + 品牌与合规代价**，后两项往往比第一项大得多。
>
> 所以这一页真正的结论是：**可靠性是要花钱买的，而混沌工程的用处，是让你买的每一分冗余都真的有效**。你不验证，冗余就只是纸面上的冗余。

板书：四档表 —— 99% → >87.6 小时/年；99.9% → >8.7 小时；99.99% → >52 分钟；99.999% → <5 分钟。

下方："**每多一个 9，停机时间少一个数量级**"；再下："损失 = 直接交易额 + 恢复成本 + 用户流失 + 品牌合规"。

提问：如果你们的产品现在是"三个 9"，老板要求明年做到"四个 9"，你会先做哪三件事？

预设回答：

- 答"多买几台服务器"——不够。加机器解决的是容量，不是"故障时的切换"。
- 答"做主备、做自动切换、做降级预案"——方向对，这三件是标准答案的主体。
- 答"还要验证这些手段真的管用"——这才是关键补充：主备切换写了代码不等于切得过去，预案写了文档不等于执行得下去，必须演练。
- 追问：所以"四个 9"不是靠买来的，是靠**设计 + 验证**换来的，验证那一半就是混沌工程。

易错点 / 考点：数字类考点只要记"每多一个 9，年停机时间除以 10"即可，不必背精确秒数。易错点是把"可

用性"和"可靠性"混用：可用性是**时间维度上能用的比例**，可靠性还包括**不出错的程度**（比如数据对不对）。

案例题里如果问"提升了可用性是否等于质量提升"，答案是不等于，还要看正确性与数据一致性。

过渡：要把可用性往上推，先得知道"到底哪些地方会坏"。课件把它做成了一张故障画像。

【7.145】稳定性风险集中治理 – 故障画像（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是本节最实用的一张图，叫**故障画像**——把"稳定性风险"从一句口号，拆成可以逐条对照的清单。

先看它归的三类问题。

第一类，异常场景类：关注的是节点异常状态下的表现，以及异常场景下的预案是否完善。说白了就是——某台机器、某个实例出问题的时候，系统表现成什么样？预案写了没有？写了管不管用？

第二类，链路设计类：关注各类链路异常预案的正确性、有效性，以及预案验证的常态化问题。"常态化"这三个字很重：预案只有演练过才算存在，写在文档里不算。

第三类，预案设计类：故障发生缺乏标准化流程，而且**快速恢复止损优于问题排查**。这一条是很多团队的认知盲区——出事那一刻，第一目标不是"搞清楚为什么"，而是"先止血"。

再看右边的风险清单，它按来源分了四组。**应用问题**：进程僵死、线程池满、连接池满、依赖超时、依赖异常、内存泄漏、慢查询、异步阻塞、缓存异常、热点问题、同步问题。**组件问题**单独列了一组。**网络问题**：网络抖动、延时/丢包、重试风暴、带宽问题、网卡问题、跨机房。**系统问题**：CPU 争抢、内存争抢、IO 异常、宕机、节点异常、集群异常。

这份清单你要当成"体检表"用。做稳定性评审时，一条一条问"我们这一条有没有风险、有没有预案、演练过没有"。它同时也是**故障库的原料**——后面讲"基于历史问题抽象通用故障库"，抽象的就是这张表。

板书：左侧三类：**异常场景类｜链路设计类｜预案设计类**（旁注：预案要常态化验证；止损优先于排查）。

右侧四组风险树：应用（僵死・线程池满・连接池满・依赖超时・内存泄漏・慢查询・异步阻塞・缓存异常・热点・同步）｜组件｜网络（抖动・延时丢包・重试风暴・带宽・网卡・跨机房）｜系统（CPU 争抢・内存争抢・IO 异常・宕机・节点/集群异常）。

提问："重试风暴"为什么会被单独列成一条风险？重试不是提高可用性的手段吗？

预设回答：

- 答"重试太多会把下游压垮"——抓住核心了。下游已经变慢，上游再放大三倍请求量，就是自己把下游打死的。

- 答"所以要加退避和熔断"——对，指数退避 + 抖动 + 熔断 + 限流，是标准解法。

- 答"重试是好事，不该算风险"——典型误解：任何提高可用性的手段，用到极端都会变成新的风险源，这正是后面讲混沌工程局限性时那条"前进一步、后退两步"。

- 收口：所以画故障画像的意义，是让你在**出事之前**就把这些"好心的坏设计"找出来。

易错点 / 考点：这张图的考法通常是"归类题"：给你一个现象，问属于应用问题、网络问题还是系统问题。判断口诀：**看是谁的资源被耗尽**——线程/连接/内存/缓存是应用层；延时/丢包/带宽/跨机房是网络层；CPU/IO/宕机是系统层。另一个考点是"预案设计类"的内涵：标准化流程 + 止损优先。

过渡：把这些风险一条条兜住，靠人工排查是不可能的。于是有了混沌工程这个思路。

【7.146】混沌工程出现：以毒攻毒（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页的标题起得很妙——"**以毒攻毒**"。课件还放了一张 Google 搜索趋势图，标的是 "Chaos Engineering"，意思是这个词在工业界的关注度是逐年往上走的。

什么叫以毒攻毒？我们平时做测试，是"给正常输入，看输出对不对"。混沌工程反过来：**主动制造本来不该发生的坏事，看系统能不能扛住**。你担心数据库主库挂了会怎样？那就真的把主库切走一次，选在业务低峰、选在小范围流量上，看它多久恢复、有没有数据错乱。你担心某个机房网络抖？那就注入延时和丢包，看熔断会不会触发、降级有没有生效。

为什么这个思路能成立？因为**故障你已经躲不掉了**。分布式系统里，硬件会坏、网络会抖、进程会僵死，这些不是"如果"，是"什么时候"。既然早晚要来，那就不如**由你自己挑时间、挑范围、挑强度地把它引出来**——可控地坏一次，好过不可控地坏一次。

课件给的定义请你们记准：混沌工程是**分布式系统的实验学科**，是一套**基于系统的实验**，用来**提高技术架构的弹性能力**，是应对生产中突发事件、提升抵御能力的复杂技术解决方案。

注意定义里的三个词：**实验**（不是破坏）、**学科**（有方法，不是拍脑袋）、**弹性**（目标是韧性，不是零故

障）。

> **板书：**中间大字"以毒攻毒"；左侧"常规测试：正常输入 → 看输出对不对"；右侧"混沌工程：主动造故障 → 看系统扛不扛得住"。

下方框住定义关键词：**实验学科｜基于系统的实验｜提高架构弹性｜抵御生产突发事件**。

> **提问：**为什么说"由你自己挑时间挑范围地把故障引出来"比"等它自己坏"更安全？

> **预设回答：**

- 答"因为你在场，能马上处理"——对，这是最直接的一条：有人盯着、有预案在手，和凌晨三点被动接警完全两回事。

- 答"因为范围可控"——对，这是第二条：你可以只影响 1% 的流量、只影响一台实例。

- 答"反正都要坏，主动坏一次没意义"——错在没有意识到"演练过的团队"和"没演练过的团队"在真实故障面前的差距，这个差距就是混沌工程要买的东西。

- 追问：所以混沌工程卖的不是"制造故障"，是"**把不可控变成可控**"。

> **易错点 / 考点：**定义题必考，关键词缺一不可。常见错答是"混沌工程就是随机杀进程"——这只描述了手段，漏了"实验学科、稳态假设、最小影响范围"这些方法要素。另一种考法是对比题："混沌工程与故障注入测试的区别"——故障注入是**技术**，混沌工程是**工程实践体系**，前者是后者的一部分。

> **过渡：**那么这套东西到底有没有用？业界做过调研，我们看数据。

【7.147】混沌工程确实能提升系统可用性（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页课件引用的是 Gremlin 那份《State of Chaos Engineering》2021 年的调研，标题就一句话：**混沌工程确实能提升系统可用性**。
>
> 这张图要你读出的不是某个具体百分比，而是三件事。
>
> **第一，做和不做有差距。** 调研里把受访团队按"是否系统性地开展混沌工程"分组，结论是开展了的团队，在可用性目标达成、故障恢复时间、团队对系统韧性的信心这几项上都更好。注意最后那个"信心"——它不是虚

> 的指标，出事的时候，**知道系统会怎么坏的团队，处置速度快得多**。
>
> **第二，成熟度不同，收益不同。** 只是偶尔手动注入一次故障的团队，收益有限；把混沌实验做成常态化、自动化、进了 CI 的团队，收益明显。这跟健身一样，一年跑一次体检不叫锻炼。
>
> **第三，最大的阻力不是技术，是人和流程。** 调研里团队反馈的主要障碍通常是"怕影响生产""没有明确的负责人""缺少稳态指标"，而不是"没有工具"。工具是最好解决的，共识最难。
>
> 这里我要提醒一句学术规范：**调研数据要看口径**。样本量多少、受访者是谁（多是已经对混沌工程有兴趣的团队）、指标怎么定义，都会影响结论。课件给的是行业趋势判断，你写报告引用时要说清"据 Gremlin 2021 年调研"，不要把它当成严格对照实验的因果结论。
>
> 一句话总结这一页：**方向是确定的——混沌工程有用；程度是要看的——用得对不对、深不深，差别很大。**

> **板书：**三行 —— **①做 vs 不做**：可用性达成、恢复时间、团队信心都有差距；**②成熟度**：手动偶发 < 常态化 < 自动化入 CI；**③阻力**：怕影响生产／无负责人／缺稳态指标（技术不是主要障碍）。

角上写："引用要报口径：Gremlin《State of Chaos Engineering》2021。"

> **提问：**调研里团队说最大的障碍是"怕影响生产"。这个担心合理吗？如果合理，混沌工程怎么回答它？

> **预设回答：**

- 答"合理，生产不能乱动"——合理，而且这正是"爆炸半径控制"存在的理由，不是让你别做，是让你**小范围地做**。

- 答"不合理，怕就别做"——太粗暴。工程决策要看代价对比：可控的小故障演练，和不可控的大故障，哪个代价高？

- 答"可以先在沙盒、金丝雀环境做"——完全正确，这就是"从线下到生产、从小到大"那条实践路径。

- 收口：所以"怕影响生产"的正确回应不是"别怕"，而是"我们有办法把影响框住"。

> **易错点 / 考点：**这一页容易被当成"背数据"，其实考点在**方法**：调研结论只能作趋势参考，不能当因果证明。答题时如果问"如何论证混沌工程的收益"，标准思路是三条：可用性指标对比、故障恢复时间（MTTR）对比、演练发现的预案缺陷数量。

> **过渡：**这个思路不是一天长出来的，它有一个清晰的发展脉络。

【7.148】混沌工程发展历程（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是一张时间轴，讲混沌工程怎么从"一些团队的经验"变成一个"有名字、有原则、有工具、有基金会的学科"。
>
> 我按阶段给你捋。**第一阶段是萌芽**：最早是大型互联网公司发现"系统太复杂，只能在生产里试"，开始有组织地做故障演练。这些实践起初没有统一名字，各家叫法不一样。**第二阶段是命名与原则化**：混沌工程这个名字被提出来，随后业界把做法提炼成几条原则——就是下一页要讲的五项原则，这一步很关键，它把"经验"变成了"可传授的方法"。**第三阶段是工具化**：出现了一批开源工具和平台，把故障注入变成写配置、点按钮就能做的事，门槛大幅下降。**第四阶段是平台化与云原生化**：随着 Kubernetes 普及，混沌工程被做成声明式 API 和 Operator，进了云原生基金会的项目体系，变成研发流程里的一环。
>
> 这条线说明一个规律：**任何工程方法成熟的标志，都是"有原则、有工具、有平台、能自动化"**。只有个人经验，没有原则，传不下去；只有原则，没有工具，落不了地；有工具但进不了流水线，就只能当表演。
>
> 你们在课程设计里做的那点故障注入，正处在这条线的最前端——手工的、单点的。别小看它，所有平台都是从一条命令开始的。

> **板书：**时间轴四段 —— **萌芽（生产里试）→ 命名与原则化（五项原则）→ 工具化（开源注入工具）→ 平台化/云原生化（声明式 API・Operator・CNCF）**。

轴下写一句："方法成熟的标志 = 原则 + 工具 + 平台 + 自动化。"

> **提问：**如果一个团队只有"出事之后复盘"的机制，没有任何主动演练，他在这条时间轴上大概处在哪个阶段？缺的是什么？

> **预设回答：**

- 答"萌芽阶段"——大体对，但严格说是"连萌芽都还没到"，因为萌芽的标志是**主动**去试。

- 答"缺工具"——不准确，先缺的是**原则和稳态标准**：不知道该看什么指标判断"系统还算正常"，工具给他也用不对。

- 答"缺的是把复盘结论变成实验的能力"——很到位，这正是下一页"从灾难中学习"要讲的。

- 收口：所以顺序是原则 → 稳态 → 工具，别倒过来，倒过来就是"为了用工具而用工具"。

易错点 / 考点：发展历程类内容一般考"阶段特征"，不考具体年份。记住四个关键词：**生产实践 → 原则化 → 工具化 → 平台化**。常见错答是把"混沌工程"和" Netflix 的 Chaos Monkey"等同——那只是一个早期代表性工具，不是整个学科。

过渡：说到原则，这就是本节最该背下来的一页。

【7.149】混沌工程的原则（5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是**必背页**。混沌工程的五项原则，课件原文写得很清楚，我一条一条讲，讲完给你一个口诀。

第一条，建立一个围绕稳定状态行为的假说。 注意"假说"这两个字。做实验之前，你得先写下一句可以被推翻的话，比如"在单机房故障时，下单成功率不低于 99.9%"。这句话里必须包含**可观测的指标**和**可接受的阈值**。没有这句话，你注入完故障只能看着屏幕说"好像还行"，那不叫实验，叫瞎试。

第二条，在实验中引入反映真实事件的变量，比如服务器崩溃、硬盘故障、网络连接断开。为什么强调"真实"？因为你造的故障越像真实会发生的，实验结论越有用。你去注入一个现实中不可能出现的故障，测出来的结论也就没意义。

第三条，在生产环境中运行实验。 这一条最反直觉，也最容易被质疑。理由是：**只有生产环境才有真实的流量、真实的依赖、真实的配置**。测试环境永远"太干净"，你在里面验证通过的容错能力，到了生产可能根本不成立。注意，这条原则和第五条是一对——是**在生产里做**，但**范围要最小**。

第四条，持续自动化运行实验。 一次演练只能证明"那一天那个版本还行"。系统天天在变，昨天能扛的故障今天未必能扛，所以实验要进流水线、要定期跑。

第五条，最小化影响范围，也就是后面专门讲的**爆炸半径控制**。

再看这张分层图：**基础设施、基础平台、业务应用**三层，横向穿过的是**故障注入**，上方由**人员组织/实践**统领，纵向贯穿的是**计划/报告、演练可观测、安全审计、量化评价、报告改进**，最外层是**风险保护**。这张图的意思是：混沌工程不是测试一个组件，而是**从基础设施到业务应用、从技术到组织的一整套闭环**——注入只是中间那一刀，前面有计划，后面有观测、审计、评价和改进。

口诀给你：**有假说、用真故障、在生产、要自动、控范围**。

> **板书：**五条竖排 —— **①稳态假说**（指标＋阈值，可被推翻）｜**②真实事件变量**（宕机・硬盘・断网）｜**③在生产运行**（真流量真依赖）｜**④持续自动化**｜**⑤最小影响范围**。

右侧分层图：人员组织/实践 ｜ 业务应用 ｜ 基础平台 ｜ 基础设施；贯穿箭头写"故障注入 → 计划/报告 → 可观测 → 安全审计 → 量化评价 → 报告改进"，外框"风险保护"。

> **提问：**第三条说"在生产环境中运行实验"，第五条说"最小化影响范围"。这两条矛盾吗？

> **预设回答：**

- 答"矛盾，生产就不能乱动"——不矛盾，这是**同一件事的两个约束**：场所选生产（为了真实性），范围压到最小（为了安全）。

- 答"不矛盾，是在生产里选一小部分做"——准确，这就是"分级发布、逐步扩大"。

- 答"那能不能只在测试环境做，算不算违反原则"——违反第三条，但可以作为**起点**；优秀实践里明确写了"从线下到生产"的进阶路径。

- 追问：所以这两条要一起理解——**真实性靠第三条，安全性靠第五条**，缺一条这实验就不成立。

> **易错点 / 考点：**五项原则是**简答和填空的高频点**，必须能一字不差地说出五条。最常错的是把"最小化影响范围"漏掉，或者把"建立假说"写成"设定目标"——**假说必须是可证伪的陈述句**，不是目标。另一个考点是理解"为什么要在生产做"：答案是真实性（流量、依赖、配置、数据规模）。

> **过渡：**原则是"该怎么做"，落到执行层，真正要干的活是四块。

【7.150】混沌工程关键工作（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把原则变成了四件具体的活：**实验设计、爆炸半径控制、注入故障能力、稳态标准**。四块缺一不可，我给你串成一条流水线。
>
> **第一块，实验设计。**三个动作：**故障画像**（照着上一页那张风险清单，挑出你要验证的风险点）、**上下游依赖梳理**（这件事坏了，谁会受影响，影响是强依赖还是弱依赖）、**链路受力分析**（故障打在这条链路的哪个位置，压力会往哪里传导）。设计没做好，注入了也是白注——你不知道自己在验证什么。
>
> **第二块，爆炸半径控制。**四个手段：**节点可控、链路可控、演练隔离机制、兜底安全策略**。节点可控是"只动

我指定的实例"，链路可控是"只影响我指定的调用路径"，隔离机制是"演练流量和真实流量分开"，兜底策略是"一旦越界立刻自动停"。这一块下面单独一页展开。

第三块，注入故障能力。 这一块最容易被误解成"只有技术团队的事"。课件把它拆成三条线：**【运】私有云、容器化、贝壳云**这类基础设施层；**【研】Java、Go、Cpp、PHP** 这类应用层，能做到方法级、代码行级的注入；**【测】接口、性能、巡检自动化**这类验证层。三条线合起来，才叫"注入能力齐备"——只有运维能拔网线，你就只能做最粗的故障。

第四块，稳态标准。 三句话：**线上服务定制化稳定标准定义、服务/业务/链路级稳态定义和管理、灵活的稳态可接入能力**。翻成大白话：每个服务的"正常"长什么样，得有一份写得出来的标准；而且要能分层次——单服务的稳态、一个业务的稳态、整条链路的稳态不是一回事；最后这套标准要能方便地接进平台，而不是每个团队自己手写一套。

四块的关系是：**设计决定验什么，注入决定怎么验，稳态决定凭什么判合格，爆炸半径决定你敢不敢验**。

板书：四象限 —— **①实验设计**（故障画像・依赖梳理・链路受力）｜**②爆炸半径控制**（节点可控・链路可控・隔离机制・兜底策略）｜**③注入能力**（运：私有云/容器/贝壳云；研：Java/Go/Cpp/PHP；测：接口/性能/巡检自动化）｜**④稳态标准**（定制化定义・服务/业务/链路分级・灵活接入）。

下方一句："设计定验什么，注入定怎么验，稳态定凭什么判合格，半径定敢不敢验。"

提问：为什么"稳态标准"要分服务级、业务级、链路级三层，只定一个全局指标不行吗？

预设回答：

- 答"一个指标太粗，看不出是哪坏了"——对，全局指标掩盖局部问题。

- 答"业务级才是用户感知的"——也对，但只有业务级就没有定位能力，出了问题不知道该修哪个服务。

- 答"三层是'定位链'：链路异常 → 定位到业务 → 定位到服务"——这就是标准答案的核心，**分级是为了既能判断好坏、又能定位原因**。

- 收口：所以稳态标准不是"定几个指标"，是"建一套能判断、能定位的指标体系"。

易错点 / 考点：四块工作常考"缺一不可"的论述题。经典错答："混沌工程的关键是有一个好的注入工具"——工具只是第三块，而且是最容易买到的。另一个高频考点是**稳态标准与假设的关系**：假设里写的阈值，就来自稳态标准；没有稳态标准，假设就是拍脑袋。

过渡：四块里最需要敬畏的是第二块——爆炸半径控制，我们专门讲。

【7.151】爆炸半径控制（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是混沌工程里**最需要敬畏**的一页。它的核心命题一句话：**在最小的实验范围内，让系统潜在的问题暴露出来，而不会意外造成更大规模的故障。**

课件给了四条实践经验，我逐条说。

第一，选择最小爆炸半径。 起步就从最小的地方开始——一个实例、一个接口、1% 的流量。

第二，混沌实验要"分级发布"。 就是随着你对系统的信心增加，逐步扩大实验范围。这句话的意思是：**范围不是一开始就定死的，是随着证据长出来的。**第一次只敢动一台机器，验证通过之后，才敢动一个集群。

第三，混沌并不意味着一定要在全量生产环境实施。 这一条是在纠正一个常见误解——好像不做全量生产就不算混沌工程。不是的，关键是"真实"，不是"全量"。

第四，要有完备的自动化和高效预案，具备快速熔断能力。 你要有一个"红色按钮"：一旦监控发现指标越过阈值，实验自动停止、故障自动恢复。没有这个按钮，就不要开始。

再看这张漏斗图，它把"范围"拆成四个维度，这是最实操的部分。

环境维度：实验环境 → 线下环境 → 沙盒环境 → 金丝雀环境 → 生产环境，越往后越真实、越危险。

流量维度：压测流量 → mock 流量 → 引流流量 → 回放流量 → 流量抽样 x%。注意"抽样 x%"——生产环境的常用做法是只让一小部分真实流量进入实验路径。

粒度维度：请求粒度 → 用户粒度。请求粒度是"这个用户的这一条请求走实验分支"，用户粒度是"这个用户的所有请求都走实验分支"。前者更细，后者更接近真实体验。

实例维度：单实例 → 多实例。最后落到"测试中间件 - 请求粒度实验"这种组合上。

一句话记法：**四个旋钮一起拧——环境越真越小心，流量越少越安全，粒度越细越好控，实例越少越好退。**

> **板书：**中央写命题："最小范围内暴露问题，不造成更大故障。"

四条实践：**最小半径｜分级发布（随信心扩大）｜不必全量生产｜自动化＋预案＋快速熔断。**

漏斗四维度：**环境**（实验→线下→沙盒→金丝雀→生产）｜**流量**（压测→mock→引流→回放→抽样 x%）｜**粒度**（请求→用户）｜**实例**（单→多）。

> **【思政落点 9】**：爆炸半径控制讲的是"**敢试，但试得有边界**"。工程上的勇气从来不是"什么都敢在生产上动"，而是"我知道边界在哪、越界会自动停"。这份敬畏对应的是责任：一次没有兜底的演练，可能把几十万用户的订单打掉。请同学们记住，真正的专业不是把风险说小，而是把风险**量化、框住、留退路**——这是工程伦理里最朴素的一条。

提问：如果领导说"这次演练必须全量生产，一次到位才有说服力"，你会怎么回应？

预设回答：

- 答"照做，领导说了算"——不行，这是把风险无边界地放大，出了问题没人兜得住。

- 答"我坚持只做 1% 流量"——立场对，方式太硬，容易变成对抗。

- 答"我先在金丝雀环境跑通、给出数据和熔断指标，再申请扩到 1%、5%，逐步放开"——这就是标准做法：**用证据换范围**，把"要不要全量"变成"证据够不够"。

- 追问：所以爆炸半径不只是技术参数，它还是**你和组织之间的一份风险契约**。

易错点 / 考点：这一页是案例分析高频点。答题框架要背住四维度：**环境、流量、粒度、实例**。最常错的答法是只说"少影响点用户"，答不出"分级发布"和"快速熔断"。另一个考点是区分"金丝雀环境"和"金丝雀发布"：前者是环境，后者是发布策略，名字像但不是一个东西。

过渡：说了这么多好处，我也得把它的短处讲清楚——混沌工程并不是万能药。

【7.152】混沌工程的局限性（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是本节最"清醒"的一页。课件第一句就定了调：**混沌工程有技术、有价值，但它的局限性也很明显**。然后给了六条，一条比一条扎心，我逐条讲。

第一条，用疫苗打比方：过时的疫苗不一定能增强免疫力，经过混沌工程的系统**可能还会继续宕机**。原因说得很直白——头痛医头、脚痛医脚，不够全面；而且**故障模式是人为设计的，局限于过去发生过的故障，无法发现未来出现的新故障**。这句话是核心：你只能演练你想得到的故障。

第二条，"泥浆大球"。这是《人月神话》里的老比喻：系统已经烂成一团，无数的混沌工程也修不好它。而且在混沌实验里发现问题之后，**想修正已经变得很困难**——你知道了，但你改不动。

第三条，前进一步、后退两步。在某一方面的优化，可能增加其他方面的脆弱性。比如你为了抗住下游超时加了重试，结果引出了重试风暴——上一页的故障画像里就有这一条。

第四条，注入本身带来新的复杂性。以"优美的、自动的方式"注入故障，同时也在给系统增加复杂度；实验做得越多，可能对系统造成进一步伤害。这句话在提醒我们：**演练工具本身也是要被测试的系统**。

第五条，过多关注故障测试，容易忽视前期设计和事后反思。这句话是在纠偏——混沌工程不能替代架构设计，也不能替代复盘。你天天演练，架构该烂还是烂。

第六条，"一键回滚"具有欺骗性，数据可能在回滚时已经遭到破坏。这一条最要命：代码能回滚，数据库里写坏的数据回不去，消息已经发出去了也收不回。

所以我对这一页的总结是：**混沌工程是一面镜子，不是一剂药**。它照出问题，不负责治好；它能验证你的韧性，不能替代你的设计。

板书：六条竖排（每条压成四字）：**过时疫苗｜泥浆大球｜进一退二｜注入增复｜偏测轻设｜回滚欺骗**。

右侧写两句原话要点："故障模式是人为设计的，只能覆盖过去"；"数据可能在回滚时已被破坏"。

底部一行："**混沌工程是镜子，不是药**。"

提问："一键回滚具有欺骗性"这句话，你能举出一个具体场景说明吗？

预设回答：

- 答"回滚代码但数据已经改了"——正确，这是最典型的一种：新版本写入了错误格式的订单，回滚代码不会把订单改回来。

- 答"消息已经发出去了"——也对，异步链路一旦投递，回滚方无从撤回。

- 答"回滚本身也要时间，这段时间业务是停的"——补充得好，回滚不是瞬时的，期间的用户请求怎么办？

- 追问：所以正确的做法是**回滚 + 数据修复预案**一起准备，并且**先做数据兼容设计**（新版本写的数据，老版本要能读）。这就是"可回滚"真正的含义。

易错点 / 考点：这页考"批判性思维"，常出论述题："既然混沌工程有这么多局限，为什么还要做？"标准答法是**两面都答**：一面是它能解决的问题（验证冗余与预案、提升团队信心、暴露未知依赖），一面是它解决不

了的（设计缺陷、数据损坏、新故障模式），结论落到"它是必要但不充分的手段"。六条局限最好能背出三条以上，其中"过时疫苗"和"回滚欺骗"最高频。

过渡：局限归局限，业界已经有不少做得很成熟的实践。我们先看一个最有画面感的。

【7.153】国外的混沌工程实践：Slack 的灾难片剧场（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲 Slack 的做法，名字叫"**灾难片剧场**"，很形象——把故障演练当成一场有剧本、有导演、有观众的演出。

课件说它 **2018 年 1 月启动**，而且强调"**严格的流程**"，这些流程的目的写得很明确：**最大限度地提高学习效果，同时把生产事故的风险降到最低**。注意这两个目标是一对：既要学到东西，又不能真出事。

具体怎么做的？课件给了四个细节。

第一，详细的计划和精心的组织。 每次演习都在**指定的时间**内进行，**专家集中在一起**。内容包括**探索性的演习**，以及在运营团队中**传阅响应计划**。注意"传阅"这个动作——它不是把文档发出去就完了，是让每个可能被叫到的人**事先读过、知道自己要干什么**。

第二，主持人要写"剧本"。 主持人记录**如何"incite the failure"**（怎么把故障挑起来）、**运行哪些命令、哪些 Amazon EC2 实例会被涉及**，还记录了**所有应该被监控的日志、指标和警报**。这几项合起来就是一份可复现、可审计的实验记录——出问题能查，成功了能复用。

第三，主持人会当场提问题。 课件举的那个问题特别值得琢磨："**对'基于开发环境的容错性预测生产环境的容错性'有多大信心？**"这个问题问的是——你在测试环境验证过的东西，你敢不敢保证它在生产也成立？

第四，为什么叫"剧场"？ 因为有人在看。让运营、研发、甚至管理层在场，演练就变成了一次**共同的经验积累**，而不是某个团队关起门来做的技术练习。

这套做法最值得学的不是工具，是**组织方式**：有剧本、有导演、有观众、有记录、有复盘。

板书：标题写"灾难片剧场（2018.01 启动）"；下面四行 —— **严格流程**（学习效果最大 + 生产风险最小）｜**计划组织**（指定时间・专家集中・探索性演习・传阅响应计划）｜**主持人剧本**（如何 incite the failure・运行哪些命令・涉及哪些 EC2 实例・应监控的日志指标告警）｜**当场提问**（"对开发环境容错性预

> 测生产容错性的信心？"）。

右侧："有剧本・有导演・有观众・有记录 → 组织能力。"

> **提问：**为什么主持人要把"运行哪些命令、涉及哪些 EC2 实例"写进记录？这看起来像流水账。

> **预设回答：**

- 答"方便复现"——对，第二次做同样的实验不用重新摸索。

- 答"出事能追责"——有一层，但更重要的是**出事能快速回退**：知道动了哪些实例，才能在失控时精确恢复。

- 答"这也是安全审计的凭证"——很到位，上一页分层图里就有"安全审计"这一项。

- 追问：所以这份记录同时服务三个目的：**复现、回退、审计**。这也是为什么它必须由主持人写，不能事后补。

> **易错点 / 考点：**这一页常考"演练的组织要素"，答题要能说出：**固定时间窗口、专家集中、剧本化记录、响应计划事先传阅、问题驱动**。经典错答是把"灾难片剧场"理解成"随机制造事故看大家反应"——恰恰相反，它的特征是**高度计划性**。

> **过渡：**Slack 关注的是"演练怎么组织"，Google 关注的是另一件事——"灾难恢复的目标怎么定"。

【7.154】Google's Disaster Recovery Program（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页讲 Google 的灾难恢复项目，它给的是一个**分层定义**的思路，特别适合拿来做架构题的答题框架。
>
> 先看三个层级。**层级 1 是"零故障"**——服务不中断。**层级 2 是中间态**。**层级 3 是"灰色的、需要恢复"**——服务已经不可用，要靠恢复手段把它拉回来。这个分级的价值在于：**不是所有服务都必须做到零故障**，你要先给自己的每个服务定级，再决定投入多少。
>
> 课件还讲了一句方法论：**构建专用架构模式，以标准化灾难恢复方法，让恢复能力的期望与服务中断场景相匹配**。这句话有三个关键词。"专用架构模式"——不要每次出事临时想办法，要把 DR 做成一套标准架构；"标准化"——同样的故障用同样的套路处理；"**期望与场景相匹配**"——这是最省钱的一条：用户能容忍 4 小时的服务，你没必要为它做零中断。
>
> 接下来是两个必须记住的指标：**RTO 是恢复时间目标**——你要多久能恢复；**RPO 是恢复点目标**——你能容忍

丢多少数据。这两个指标是所有灾备方案的度量单位，考试必考。

课件给了三组数字，正好演示"分级"：**区域级：RTO 12 小时、RPO 24 小时**；再往上一档是 **RTO 4 小时、RPO 零**；最高一档是 **RTO 零、RPO 零**。你把这三组连起来看就明白了：**级别越高，RTO 越短、RPO 越小，成本也越高**。RPO 为零意味着每一条数据都不能丢，那就要同步复制；RTO 为零意味着不能中断，那就要多活、要流量秒级切换。

这一页和混沌工程什么关系？关系很直接：**RTO 和 RPO 是写在纸上的承诺，混沌工程是验证这个承诺能不能兑现的手段**。你不做演练，永远不知道你的"12 小时"到底是 12 小时还是 12 天。

板书：三层级 —— **层级 1 零故障｜层级 2｜层级 3 灰色、需要恢复**。

两个指标大字：**RTO = 恢复时间目标｜RPO = 恢复点目标**。

三组数据：**区域级 RTO 12h / RPO 24h → RTO 4h / RPO 0 → RTO 0 / RPO 0**；箭头标"级别↑ 成本↑"。

下方："DR 承诺写在纸上，混沌工程验证它兑不兑现。"

提问：一个内部报表系统，用户一天看一次。你会给它定什么级别的 RTO 和 RPO？如果是一个支付系统呢？

预设回答：

- 答"报表系统 RTO 几小时都行，RPO 一天也行"——合理，因为它本来就是按天更新的，中断半天没人受不了。

- 答"支付系统必须 RTO 零、RPO 零"——方向对，但要说清代价：零 RPO 要同步复制，零 RTO 要多活加秒级切流，成本是数量级的差别。

- 答"都按最高标准做最保险"——这是最典型的过度设计，成本上去、复杂度也上去，反而更容易出故障。

- 追问：所以定级的依据是**业务影响分析**——这个服务停 1 小时，谁受影响、损失多少。这就是"期望与场景相匹配"。

易错点 / 考点：RTO 和 RPO 的区别是必考判断题。口诀：**RTO 管"多久能回来"（时间），RPO 管"能丢多少"（数据）**。常见错法是把两者记反，或者认为 RPO 为零就一定 RTO 为零——两者独立。另一个考点是"为什么不是所有服务都要最高级别"：因为成本与复杂度，且过度设计本身增加脆弱性。

过渡：Google 讲的是目标怎么定，微软给的是工具怎么用——而且是云上的做法。

【7.155】微软的混沌工程（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲微软的做法，代表了一种典型的云厂商思路：**把混沌工程做成云上的托管服务**，用户不装软件、不搭平台，直接在控制台上配一个实验就行。
>
> 课件给的那句英文是关键：**用基于 agent 的故障，通过 Azure 门户给一台 Linux 虚拟机加 CPU 压力**。这句话里有三个信息。
>
> **第一，"基于 agent"**。云厂商不可能在你的机器里随便乱动，它需要在目标资源上装一个代理，由代理去执行注入。这是云上混沌工程和自建混沌工程最大的形态差别：自建可以随便侵入，云上必须走代理。
>
> **第二，"CPU 压力"是注入内容**。这就是我们前面讲的"注入故障能力"的一种，攻击面很明确——压 CPU，看你的服务在算力被抢的情况下会不会超时、会不会雪崩。
>
> **第三，"通过门户"说明它是产品化、界面化的**。降低门槛是云厂商的核心价值，代价是你只能在它提供的故障类型里选。
>
> 再看下面那条 CLI：`az vm identity assign --ids $VM_RESOURCE_ID --identities $MANAGED_IDENTITY_RESOURCE_ID`。这条命令做的事是**给虚拟机分配一个托管标识**。为什么混沌实验前要先做这一步？因为**注入是需要权限的**——平台要能操作这台机器、要能读它的监控数据，而你绝不能给它一把万能钥匙。所以标准做法是：**用托管标识做最小授权**，只给它做实验需要的那点权限。
>
> 这一页的技术含量其实就在这条命令上。它告诉你一件很重要的事：**在云上做混沌工程，第一个要解决的不是"怎么注入"，而是"授权怎么给得又够又不多"**。给多了，混沌平台自己就成了最大的风险源。

> **板书：**三信息 —— **基于 agent**（目标资源装代理执行）｜**CPU 压力**（注入内容之一）｜**门户化**（产品化、门槛低、可选故障类型受限）。

CLI 一行：`az vm identity assign --ids $VM_RESOURCE_ID --identities $MANAGED_IDENTITY_RESOURCE_ID` → 旁注"**最小授权的托管标识**"。

底部："云上做混沌，第一关是授权，不是注入。"

> **提问：**为什么云上做混沌实验，不直接用管理员账号密码去操作目标机器，而要先配一个托管标识？

> **预设回答：**

- 答"管理员权限太大，出事就是大事"——正确，这是最小权限原则。

- 答"托管标识不用管密钥，避免泄露"——也对，这是它相对"写死密钥"的优势：凭据由平台托管、自动轮换。

- 答"配了标识就能审计了"——很好，谁在什么时候用这个标识做了什么，有记录。

- 追问：所以这条命令背后是三件事：**最小权限、免密钥、可审计**。这也是所有自动化运维平台的通用要求。

> **易错点 / 考点：**这一页考"云上混沌工程的形态差异"，落在三点：**代理式注入、托管服务化、基于标识的最小授权**。常见错答是把"云上混沌"理解成"云厂商帮你制造故障"——它是**给你工具让你自己造**，责任主体仍是你的团队。若考权限设计，标准答案必含"最小权限原则"与"可审计"。

> **过渡：**混沌工程的实验、故障库、可观察性三件事，是把"敢在生产上捣乱"变成"有章法地捣乱"的关键，我们逐个看。

【7.156】实验（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 混沌工程落到操作层面，最小的单位不是"一次测试"，而是"一个实验"。这两个词差别很大：测试用例问的是"结果对不对"，实验问的是"我那个假设站不站得住"。
>
> 一个完整的实验有五件事。先写清楚**要验证的假设**，比如"订单服务挂掉一个实例，下单成功率不掉"。再**定义稳态指标**——成功率、P99 时延、错误率。然后**设计注入**：注什么故障、注在哪个实例、注多久、影响多大范围。接着**运行并观察**。最后**得出结论**：假设成立，系统的韧性得到一次证明；假设被推翻，那就找到了一个真实的风险点，转成待修项。
>
> 这个顺序不可以反过来。很多团队一上来就问"我能注什么故障"，那是从工具出发；正确的起点是"我担心什么、我怎么知道它没发生"。**先有稳态，才有实验**——没有稳态指标，注完故障你连"这是不是异常"都判断不了。
>
> 还有一点很关键：实验是**可以重复**的。今天验证过的假设，下个版本还要再验一遍，因为架构在变、依赖在变。所以实验要留档：假设、参数、结论都要能复现。这就是它和"随手压一下"的根本区别。

> **板书：**一条五段式箭头 —— **假设** → **稳态指标** → **故障注入** → **运行观察** → **结论**（成立✓／推翻✗→转风险待办）；右侧一句："实验 ≠ 测试用例；先有稳态，才有实验。"

> **提问：**如果你只写得出"注入 CPU 80% 压五分钟"，却写不出一句"我认为业务不会受影响"的假设，这个实验

缺了什么？

预设回答：

- 答"缺稳态指标"——对了一半，稳态指标是"怎么判断有没有受影响"的依据；缺了它，实验做完只剩一堆曲线。

- 答"缺假设"——更根本。没有假设，实验就不是验证，只是"制造了一次故障看看会怎样"，结论无从归档、无从复现。

- 答"注了 CPU 压力不算故障吧"——典型的错答。CPU 被抢占正是最典型的资源类故障，问题不在故障种类，而在你没有事先说清"受影响与不受影响的界线在哪"。

- 收口：假设和稳态指标是一对，缺一个，实验就退化成演练。

易错点 / 考点：判断题常考"实验必须以假设为前提"——正确。选择题常把"定义稳态指标"排在"设计注入"之后当干扰项，**正确顺序是先定稳态**。记忆口诀：**"先立标尺，再动手"**。

过渡：实验里的"注入什么"这一栏，靠的不是灵感，而是一个沉淀下来的库——下一页就是故障库。

【7.157】故障库（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页看一个"故障库"的实物：微软 Azure Chaos Studio 的故障清单。为什么要看别人家的清单？因为故障库直接决定混沌工程能做到多大——**你能注入的故障种类，就是你能发现的风险种类**。

逐类看。第一类是**资源压力**：CPU 压力、物理内存压力、虚拟内存压力、磁盘 I/O 压力，再加一条"任意 Stress-ng 压力"。第二类是**时间与进程**：时间改变（Time change）、杀进程、时间延时、停掉一个 Windows 服务。第三类是**网络**：DNS 失败、网络延迟、网络断开、带防火墙规则的网络断开；往上还有改网络安全组规则。

第四类是**云资源本身**：ARM 虚机关机、虚拟机规模集实例关机、Cosmos DB 故障切换、Azure Cache for Redis 重启、经典云服务关机、Key Vault 禁止访问（Deny Access）。第五类是**在 AKS 上借 Chaos Mesh 注入的八种**：网络、Pod、压力、IO、时间、内核、HTTP、DNS。AKS 即 Azure Kubernetes Service。

这张清单最值得学的不是长，是**颗粒度**。同样是"网络故障"，它拆成延迟、断开、带防火墙规则的断开、DNS 失败、改安全组五档。为什么拆这么细？因为不同的拆法对应完全不同的失效模式：延迟是"变慢"，断

开是"报错"，DNS 失败是"找不到"，安全组改错是"全断"。不拆开，测来测去只能测出同一类问题。

> **板书：**五栏竖排 —— **①资源压力**（CPU／物理内存／虚拟内存／磁盘 IO／Stress-ng 任意）→ **②时间与进程**（时间改变・杀进程・延时・停服务）→ **③网络**（DNS 失败・延迟・断开・断开+防火墙・安全组）→ **④云资源**（关机・规模集・DB 故障切换・Redis 重启・Key Vault 拒绝）→ **⑤ AKS 上 Chaos Mesh 八类**。
> 旁注："AKS = Azure Kubernetes Service"。

> **提问：**清单里"网络延迟"和"网络断开"是两条。如果你只做了延迟实验，业务全绿，能否说明网络故障已经验证过了？

预设回答：

- 答"能，延迟更难忍"——典型的错答。延迟考验的是超时与重试，断开考验的是熔断、重连、降级，两条路径完全不同。

- 答"不行，断开更狠，会直接报错"——正确。断开之后流量瞬间失败，考验的是熔断器和降级预案有没有真的生效。

- 答"那就两条都跑一遍"——对，而且还要加"带防火墙规则的断开"和"DNS 失败"，因为它们的失败信息不一样，排障路径也不一样。

- 收口：颗粒度决定覆盖度，一个故障类型只测一档，等于只测了最容易的那一档。

> **易错点 / 考点：**考点是"故障库的分类维度"。常见错答是把它们按"轻重"分（严重／一般），正确维度是**按资源与失效模式分**：资源、时间/进程、网络、云资源、容器编排层。答题三步法：**看故障作用在谁身上 → 看它改变了什么（量／连通性／可见性／生命周期）→ 归到库里哪一栏**。

> **过渡：**注入了故障，怎么知道"业务到底有没有受影响"？这就要靠下一件事——可观察性。

【7.158】可观察性（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页只有一个词，但它其实是混沌工程能否成立的前提：**可观察性**。
>
> 先把三个词分清。**监控**是"我事先想到要看的指标，我盯着它"；**可观察性**是"我没事先想到的问题，也能从系统吐出来的数据里问出来"。差别在哪？监控回答的是已知问题，可观察性回答的是未知问题。混沌工程制造

的恰恰是"你没预料到的那一种异常"，所以只靠预设的几张监控大盘是不够的。

可观察性靠三类数据支撑：**指标**（Metrics，数字随时间的变化，适合看趋势和告警）、**日志**（Logs，离散事件，适合看"到底发生了什么"）、**链路追踪**（Traces，一次请求跨了哪些服务、每段花了多久，适合定位"慢在哪一跳"）。三类合起来，才能回答"故障注进去之后，用户那边到底感受到了什么"。

这一页要你建立的判断是：**实验的结论质量，上限由可观察性决定**。指标做得糙，你只能说"好像没大事"；链路追踪缺失，你只能知道"失败了"却不知道失败在哪一跳。所以做混沌工程之前，先问一句：我的观测手段够不够支撑一次判定？

板书：中间写"可观察性"；下方三支 —— **Metrics**（趋势・告警）｜**Logs**（发生了什么）｜**Traces**（跨服务・慢在哪一跳）；上方一句："监控＝已知问题的答案；可观察性＝未知问题的提问能力。"

提问：如果一套系统只有 CPU、内存这类主机监控，没有业务成功率、没有链路追踪，你能用它给混沌实验下结论吗？

预设回答：

- 答"能，看 CPU 没飙就没事"——错得很典型。主机指标正常而业务已经大面积失败，是分布式系统里最常见的景象：机器好好的，链路断了。

- 答"判不了，缺业务级指标"——正确。判定稳态必须用**用户视角的指标**，比如成功率、时延分位值。

- 答"那就先补监控再实验"——最务实的答案。顺序就是：先有观测，再谈注入。

- 收口：观测能力是实验的地基，地基不够，实验做了也白做。

易错点 / 考点：常考"监控与可观察性的区别"，标准表述是**预设指标 vs 可任意提问**。另一常见错答是把"可观察性"等同于"日志多"。日志多不等于能提问，关键是有没有把指标、日志、追踪三者关联起来（同一个 trace id 串起来）。

过渡：从这一页开始进入优秀实践。第一条实践，讲的是"混沌工程到底在为什么服务"。

【7.159】优秀实践 1：复杂系统韧性实现之道（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是优秀实践的第一条，标题就把立场摆明了：**混沌工程是复杂系统韧性的一条实现之道**，不是万能药。

> 课件标注来源是中国信息通信研究院 2021 年的成果，是行业层面归纳出的实践共识。
>
> 为什么强调"复杂系统"？因为简单系统不需要混沌工程。一个单体应用、一个数据库、一台机器，它的失效模式你数得过来，做压测、做容灾演练就够了。复杂系统不一样：服务几十上百个、依赖交叉、调用链深、还会自动扩缩容和实例迁移。它的故障往往不是"某个零件坏了"，而是"几个正常的零件凑在一起出了怪事"——这种问题靠读代码和开会讨论是推不出来的，只能靠**在真实环境里做受控实验**把它逼出来。
>
> "韧性"这个词也要说清。**可靠性**讲的是"尽量不出事"，**韧性**讲的是"出了事能扛住、能快速恢复"。。混沌工程干的正是后一件事：它假定故障一定会发生，然后问"发生了会怎样"。
>
> 所以这一条实践的含义是：把混沌工程放进**架构治理**的盘子里，而不是当成测试团队的一个附加动作。它的产出不是"通过了/没通过"，而是"我们在哪一层还扛不住"。

> **板书：**中间一行大字："**可靠性＝尽量不出事；韧性＝出了事扛得住**"。左侧写"复杂系统：服务多・依赖交叉・链路深・自动扩缩容"；右侧写"混沌工程＝受控实验 → 暴露'正常零件凑出的怪事'"；底部标"来源：中国信息通信研究院，2021 年"。

> **提问：**一个只有三个服务的系统，值得专门上混沌工程吗？如果值得，理由是什么？

> **预设回答：**

- 答"不值得，服务太少没什么可混沌的"——可以理解，但如果这三个服务里有一个是支付或账务，它一挂业务就停，那就值得做**针对性的少数实验**。

- 答"值得，因为依赖第三方"——这个思路很好。系统复杂度不只来自自己的服务数量，还来自**外部依赖的数量和不确定性**。

- 答"值得，因为要合规"——动机偏了。混沌工程的价值是发现风险，不是为了凑一份材料；动机错了，实验就会做成表演。

- 收口：判断标准不是服务几个，而是"**它挂了，业务停不停、损失多大**"。

> **易错点 / 考点：**选择题常考"可靠性 vs 韧性"的区分，正确表述是**可靠性偏预防、韧性偏承受与恢复**。另一易错点是把混沌工程当成"故障演练"的同义词——演练是照着剧本走，混沌工程是**先有假设、再验证**，有没有假设是分界线。

> **过渡：**既然假定故障一定会发生，那第一个要学的动作就是——出过的事故，别浪费。

【7.160】优秀实践 2：从灾难中学习、细化实验目标（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

第二条实践是一句话：**从灾难中学习，并且把实验目标细化**。这句话里其实有两个动作，我们分开说。

第一个动作是"**从灾难中学习**"。任何一次真实故障，都是一次免费且昂贵的实验——昂贵是因为它已经造成了损失，免费是因为你不需要再设计注入，故障自己来了。所以复盘的价值不在写报告，而在把它**转化成下一次可以主动复现的实验**：当时是哪个依赖先超时？哪一层没有熔断？告警是几分钟后才来的？把这些问清楚，就得到了一组实验参数。

第二个动作是"**细化实验目标**"。什么叫细化？"验证系统高可用"是空目标；"验证订单服务在两个实例中挂掉一个、且注入持续 30 秒的情况下，下单成功率不低于 99%"才是可执行的目标——有对象、有故障、有范围、有阈值。目标越细，结论越硬；目标越空，实验就越容易变成"跑了一遍，好像没事"。

这两件事是连着的：**灾难提供问题的方向，细化把它变成可以验证的命题**。没有第一步，你不知道该测什么；没有第二步，你测了也不知道算不算通过。

顺便说清一个态度问题：复盘的目的不是找责任人，而是找**系统的薄弱环节**。要是每次复盘都变成追责会，以后就没人愿意讲真话了，你也拿不到真实的故障画像。

板书：左右两栏 —— 左："**从灾难中学习**：真实故障 → 时间线还原 → 哪一层没扛住 → 转成实验参数"；右："**细化实验目标**：空目标'验证高可用' ✗ → 具体目标'挂 1/2 实例 · 30 秒 · 成功率≥99%' ✓"。底部："复盘找薄弱环节，不找责任人。"

提问：把"验证系统高可用"写进实验目标，问题出在哪？请你现场把它改写成一句可执行的。

预设回答：

- 答"没问题，方向对就行"——错。这样的目标没法判定通过与否，实验做完只能凭感觉下结论。

- 答"应该写清楚注什么故障、注在哪、注多久、多少指标算通过"——完全正确，这就是细化的四个要素。

- 答"应该写成'系统要 99.99% 可用'"——这是 SLA 目标，不是实验目标。它描述的是长期结果，实验要的是**一次可判定的观测**。

- 收口：实验目标的模板就是一句："在【故障】【范围】【时长】下，【指标】不低于【阈值】。"

易错点 / 考点：高频考点是**实验目标的可判定性**。判断题"实验目标可以写'验证系统稳定性'"——错，不可判

定。另一易错点是把 SLA／SLO 当实验目标：SLA 是承诺，实验目标是**验证承诺是否成立的手段**。

【思政落点 10】：从灾难中学习，前提是**允许把问题讲出来**。复盘文化最怕两件事：一是把复盘开成追责会，二是把"没出事"当成"没问题"。工程上的每一次事故，都是系统在用损失换来的信息；如果团队因为怕担责而隐瞒细节、只写"操作失误"，这份信息就白费了。请记住：**敬畏风险，不等于回避风险；敢于暴露问题、如实记录、把教训固化成实验，才是真正的负责**。

过渡：事故给的教训如果只停在一份报告里，下一个人还会踩同一个坑。所以下一步是把它们抽象成通用的东西——故障库。

【7.161】示例：基于历史问题抽象建设通用故障库（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给的是一张"从历史问题抽象出通用故障库"的示意图，我认为它是本章最值得抄回自己项目的一页。

图的纵向是**三个层次**。底层是**基础设施**，指机房那一层的东西——课件在这里用"风：空调、火：消防、水：恒湿"来代表机房环境类风险，也就是断电、温度、水患这类物理层面的问题。中间是**基础架构**，指服务器、网络、存储、中间件这些支撑件。顶层是**业务应用**，指你自己的服务。

图的横向，是从历史问题里抽出来的**故障类型**。基础架构这一档列得很密：CPU 抢占、内存抢占、IO 抢占、存储热点、连接打满、网络抖动、网络拥塞、网络中断、DNS 异常、宕机、假死、磁盘读写慢、网卡打满、实例迁移、实例部署、OOM、core。业务应用这一档列的是：下游失败、依赖超时、性能下降、格式错误、配置错误、环境错误、业务攻击。

关键在于"**抽象**"这两个字。你出过一次"某个接口超时"，记成"接口超时"就只能防这一处；抽象一层，记成"依赖超时"，它就能套到所有下游调用上；再抽象一层，记成"下游失败"，它连"下游返回错数据"都覆盖了。**故障库是按抽象层级建的，不是按事故编号建的**。所以每来一次新故障，你的动作应该是：先判断它属于哪一类，能归类就补参数，归不进去才新增一类。

板书：三层横带 —— 上"**业务应用**"（下游失败・依赖超时・性能下降・格式错误・配置错误・环境错误・业务攻击）｜中"**基础架构**"（CPU／内存／IO 抢占・存储热点・连接打满・网络抖动／拥塞／中断・DNS 异常・宕机・假死・OOM・core）｜下"**基础设施**"（风：空调・火：消防・水：恒湿）。右侧竖写一句："**按抽象层级建库，不按事故编号建库**"。

> **提问：**你出了一次事故："某次大促，商品详情页因为评论服务响应慢而整体变慢。"这条应该收进故障库的哪一类、记成什么？

> **预设回答：**

- 答"记成'评论服务慢'"——太具体，只能防这一个服务，换个服务照样出事。

- 答"记成'依赖超时'，归到业务应用层"——正确。抽象一层之后，它就能套到所有同步依赖上，还能顺手补上"超时阈值设多少、有没有降级"这两个参数。

- 答"记成'性能下降'就行"——抽象过头了。太粗的类别没法指导注入，你得知道注的是"被调方变慢"还是"本机变慢"。

- 收口：好的故障条目，要**抽象到能复用，具体到能注入**。

> **易错点 / 考点：**考点是"故障库的分层与抽象粒度"。常见错答把"故障库"理解成"事故台账"——台账是记录历史，故障库是**可注入的故障类型清单**。判断三步法：**这条能否转化成注入动作？→ 能否套到别的服务上？→ 参数写不写得出来？** 三问都过，才算合格条目。

> **过渡：**库建好了，接下来最容易被忽略、也最容易出事的一步——在哪儿做实验，做多大。

【7.162】优秀实践 3：从线下到生产、从小到大（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 第三条实践讲的是**推进路径**，两条线同时往前走：一条是"从线下到生产"，一条是"从小到大"。
>
> 第一条线是**环境递进**：测试环境 → Staging 环境 → 预览环境里的特定流量 → 生产集群全流量。为什么不直接上生产？因为生产上做实验有真实用户，一旦爆炸半径失控，实验本身就成了事故。所以先在测试环境把注入动作跑通，再到 Staging 验证服务间的影响，然后在预览环境只放一小部分真实流量进来，确认线上行为一致，最后才放到生产全流量。
>
> 第二条线是**范围递进**：可控流量 → 单个接口 → 单机 → 单集群 → 单机房 → 全链路。这条线量的是"一次实验能影响多大"。注意它和第一条线不是一回事：环境说的是"在哪儿做"，范围说的是"影响多大"。两头都要一格一格往上走，**每一格都要有明确的回退开关**。
>
> 为什么必须这么慢？这就是"**爆炸半径**"的概念。爆炸半径＝一次实验可能波及的用户与业务范围。半径小的

> 实验，最坏结果是你自己发现一个问题；半径大的实验，最坏结果是全体用户一起帮你发现问题。所以推进的节奏应该由**证据**决定：上一格的实验结果稳定、回退动作验证过、监控能覆盖，才允许进入下一格。
>
> 一句话：**先在最小半径里把方法练熟，再用可控的半径去换真实的信息**。

> **板书：**两条平行箭头 —— 上行"**环境**：测试 → Staging → 预览环境特定流量 → 生产全流量"；下行"**范围**：可控流量 → 单接口 → 单机 → 单集群 → 单机房 → 全链路"。中间竖线标"**爆炸半径**"，右侧写"每一格都要有回退开关；上一格稳了才进下一格"。

> **提问：**团队已经能稳定在生产上对单机做 CPU 压力实验了，能否直接跳到"全链路网络延迟"？

> **预设回答：**

- 答"能，反正都是注故障"——危险。全链路延迟会同时影响所有调用方，一旦超时与重试叠加，很容易演变成雪崩，而这恰恰不是你想验证的那件事。

- 答"不行，中间缺了单集群、单机房两级"——正确，跳级等于放弃了中间两级能给你的信息：集群内的调度与迁移、机房级的容灾切换。

- 答"可以先做，反正有开关能停"——开关存在不等于开关有效。回退动作本身也要在低半径里验证过，才敢在大半径里依赖它。

- 收口：半径的扩大必须由证据推动，不可以由进度表推动。

> **易错点 / 考点：**高频考点是"**爆炸半径的定义与控制手段**"：定义＝实验可能影响的用户与业务范围；控制手段＝限制注入对象（单实例／单集群）、限制流量（小流量／特定用户）、限时（duration）、熔断回退开关、避开高峰。常见错答是把"爆炸半径"理解成"故障的严重程度"——它是**影响范围**，不是故障强度。

> **【思政落点 11】**：爆炸半径这四个字，本质上是**对风险的敬畏**。做工程的人容易有一种自信：我懂这套系统，我知道注进去会怎样。可复杂系统的教训恰恰是"你不知道"。所以，凡是可能影响到真实用户的操作，都要先问三句：**影响谁、影响多大、出问题怎么退**。把试验关在可控的笼子里，不是胆小，而是把"敢试"和"负责"同时做到的唯一办法——**可控才敢试，有退路才敢往前**。

> **过渡：**环境和范围都在往上走，那"看得见"的能力跟不跟得上？下一条实践专门说这件事。

【7.163】优秀实践 4：混沌工程可观察性（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 第四条实践又回到可观察性，但视角变了：前面那一页讲的是"可观察性是实验的前提"，这一页讲的是"**混沌工程本身也要被观察**"。
>
> 第一层意思是，**实验过程要被记录下来**。什么时候开始注的、注了哪些对象、参数是多少、什么时候恢复的、恢复是否成功——这些都要有日志。为什么？因为你在排查"实验期间业务为什么抖了一下"的时候，需要一条精确的时间线，把注入动作和业务曲线对起来。
>
> 第二层意思是，**实验的影响要被度量到业务侧**。不该只看被注入的那台机器，还要看它上游的调用方、下游的被调方，以及最终的用户成功率。故障是注在一个点上的，影响却是沿着调用链传的，观测必须跟着传。
>
> 第三层意思是，**观测本身要覆盖实验窗口的前后**。注之前要有一段基线，注之后要看到恢复是否回到基线。只看中间那一段，你判断不了"是恢复了还是本来就在抖"。
>
> 所以这一条实践可以收成一句操作要求：**每做一次实验，都要能拿出一张"注入—指标—结论"对齐的图**。拿不出来，这次实验就只是一次事故演练，不是一次工程验证。

> **板书：**三行 —— ①**过程留痕**（注入时间・对象・参数・恢复时间・恢复结果）；②**影响沿链路度量**（被注对象 → 上游调用方 → 下游被调方 → 用户成功率）；③**覆盖实验窗口前后**（基线 → 注入期 → 恢复期）。
> 底部："拿不出'注入—指标—结论'对齐图，就不算验证。"

提问：实验结束后业务指标回到了正常值，是不是就可以判定"实验通过"了？

预设回答：

- 答"是，恢复了就说明系统扛住了"——不严谨。还要看注入期间指标有没有掉出阈值，以及恢复是自动的还是靠人工重启的。

- 答"要看注入期间有没有跌破稳态阈值、恢复是不是自动完成"——正确。**扛住**的判据在注入期，**韧性的质量**还要看恢复方式。

- 答"只要日志里没报错就行"——错。业务失败往往不体现在被注机器的日志里，而是体现在上游的超时与重试上。

- 收口：判据要落在"注入期的稳态"和"恢复期的方式"两处。

易错点 / 考点：常考"实验判定需要哪些观测数据"，标准答案要包含**基线、注入期指标、恢复过程、调用链上下游**四项。易错点是把"实验日志"和"业务监控"混为一谈——前者记录你做了什么，后者记录系统发生了什么，两者要能按时间对齐才有用。

过渡：方法讲完了，落到工具。第五条实践先给一个原则：别从零造。

【7.164】优秀实践 5：采用开源混沌工程平台（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

第五条实践是个很实在的建议：**采用开源的混沌工程平台**，不要一上来就自研。

课件给了六个项目，我按它们的定位点一遍。**Chaos Toolkit** 是一个偏"实验即代码"的框架，用配置文件描述实验步骤，语言无关，适合把实验接进流水线。**Litmus Chaos** 面向 Kubernetes，用 CRD 定义混沌实验，和 GitOps 的配合比较顺。**Chaos Mesh** 后面会专门讲，是云原生方向上生态最完整的一个。**chaoskube** 的思路很轻：按标签随机杀掉 Pod，用来做"随机制造噪声"的常态化实验。**kube-monkey** 借的是 Netflix 那只"混沌猴子"的思路，按计划随机关掉 Pod。**KubeInvaders** 则把实验做成了可视化的小游戏界面。

为什么建议先用开源？三个理由。**第一，故障场景的实现是最难的部分**——怎么在网络命名空间里加延迟、怎么让 cgroup 超限，这些细节踩过的坑最多，开源项目已经帮你踩过了。**第二，生态和标准一致**——大家都在用 CRD、都用 YAML 描述实验，你的实验定义将来换平台也能带走。**第三，成本**——自研一个能覆盖基础资源、容器、应用层的注入平台，投入远超一个课程设计的工作量。

但要说清一点：开源给的是**能力**，不是**方法**。装了 Chaos Mesh 不等于会做混沌工程，假设、稳态、爆炸半径这些还是要你自己想。

板书：六格 —— **Chaos Toolkit**（实验即代码・语言无关）｜**Litmus Chaos**（K8s・CRD・GitOps）｜**Chaos Mesh**（云原生・生态完整）｜**chaoskube**（按标签随机杀 Pod）｜**kube-monkey**（计划式随机杀 Pod）｜**KubeInvaders**（可视化演示）。底部："开源给能力，方法还得自己想。"

提问：既然开源平台都现成，为什么很多公司最后还是自研了一套？

预设回答：

- 答"为了炫技"——少数情况是这样，但通常不是主因。

- 答"因为要接自己的发布系统和监控"——说到点子上了。平台的价值在于"和你的流水线、你的指标体系打通"，开源工具在这两处往往要二次开发。

- 答"因为开源的不安全"——这个理由站不住。成熟开源项目的注入能力本身是可控的，风险主要来自**用的人把爆炸半径开太大**。

- 收口：合理的路径是"先用开源跑通方法 → 再按自己的流水线做集成"，而不是从零造轮子。

> **易错点 / 考点：**考点是"工具与方法的区别"。常见错答是"用了混沌平台就等于做了混沌工程"——平台只提供注入能力，**假设、稳态、半径、结论**四件事仍然要人来做。若考工具定位，记住三个锚点：Chaos Mesh 与 Litmus 偏 K8s、Chaos Toolkit 偏实验编排、chaoskube 与 kube-monkey 偏随机杀 Pod。

> **过渡：**六个项目里有一个是国内团队做的、也是本教材后面重点讲的——ChaosBlade。

【7.165】ChaosBlade（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页介绍 **ChaosBlade**。它是阿里内部 **MonkeyKing** 开源出来的，按课件说法是在**十年故障测试和演练实践**基础上演化而成。值钱的是"十年""实践"两个词——它是被真实故障反复打磨出来的工具。
>
> 实验场景分五类。**基础资源**：CPU、内存、网络、磁盘、进程。**Java 应用**：数据库、缓存、消息、JVM 本身、微服务，而且可以指定**任意类方法**注入，这就意味着你能精确到"让某个 service 的某个方法多花 500 毫秒"。**C++ 应用**：可以指定任意方法或某一行代码注入延迟、篡改变量和返回值——注意"篡改返回值"这四个字，它意味着你能验证"下游返回了错误数据时，我的代码会不会崩"。**Docker 容器**：杀容器、容器内的 CPU／内存／网络／磁盘／进程。**云原生平台**：Kubernetes 节点上的各类实验，以及 kill Pod、kill Docker。
>
> 它能做得这么细，靠的是**分层实现**：基础资源靠操作系统接口，Java 靠 Java Agent 挂载，C++ 靠 GDB 调试器。不同语言用不同手段，所以才能做到"方法级"和"代码行级"。
>
> 对比 Chaos Mesh，两者定位不同：Chaos Mesh 的根在 Kubernetes，ChaosBlade 的根在**单机与进程**，往上再长到容器和 K8s。所以如果你的系统里有大量非容器化的老服务，ChaosBlade 更顺手。

> **板书：**中间写"**ChaosBlade**（阿里 MonkeyKing 开源・十年故障测试实践）"；下方五类场景 —— **基础资源**（CPU／内存／网络／磁盘／进程）｜**Java**（DB・缓存・消息・JVM・微服务・任意类方法）｜**C++**（任意方法／代码行・延迟・篡改变量与返回值）｜**Docker**（杀容器・容器内各类资源）｜**云原生**（K8s

节点・kill Pod・kill Docker）。右下角："Java Agent ／ GDB ／ OS 接口 = 分层实现"。

提问："篡改某个方法的返回值"这种注入，能验证什么别的手段验证不了的东西?

预设回答：

- 答"验证代码写得对不对"——太笼统。它验证的是**对异常数据的容错**，不是逻辑正确性。

- 答"验证下游返回脏数据时，上游会不会把脏数据写进库"——很好，这就是数据链路容错。这类问题用断电、延迟都测不出来。

- 答"这种注入太危险，不该用"——恰恰相反，它是最便宜的一种验证：你可以在测试环境里让某个方法稳定地返回错误值，看系统怎么反应。

- 收口：注入的粒度越细，你能验证的假设就越具体——这就是方法级注入的价值。

易错点 / 考点：考点是"ChaosBlade 的场景分类与实现手段的对应关系"：**基础资源→系统调用／内核接口，Java→Java Agent，C++→GDB，容器→Docker API／CRI，K8s→Operator**。常见错答是把 ChaosBlade 与 Chaos Mesh 当同类替代品——一个是"从单机长到云原生"，一个是"生在云原生"，适用系统不同。

过渡：它凭什么能做到方法级注入？答案在组件拆分上。

【7.166】ChaosBlade 关键组件（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页拆的是 ChaosBlade 的组件，一共八个模块，看着多，其实按"一个入口 + 一组执行器 + 一份模型定义"来记就清楚了。

入口是 `chaosblade`：它是混沌实验的执行工具，提供 CLI 和 HTTP 两种执行方式，包含创建实验、销毁实验、查询实验、实验环境准备、实验环境撤销这些命令。注意"环境准备"和"环境撤销"——很多注入要前置条件，由它统一处理。

执行器按场景分家：`exec-os` 管基础资源；`exec-docker` 通过调用 Docker API 标准化实现容器场景；`exec-cri` 通过 CRI 标准化实现容器场景；`chaosblade-operator` 管 Kubernetes 平台，用 K8s 资源操作的方式来创建、更新、删除实验，可以用 kubectl、client-go 或 chaosblade CLI 触发；`exec-jvm` 管 Java 应用，靠 **Java Agent 技术动态挂载**；`exec-cplus` 管 C++ 应用，靠 **GDB** 做到方法与代码行级别的注入。

模型定义是 `chaosblade-spec-go`：混沌实验模型的 Golang 语言定义，方便用 Go 实现的场景复用同一套模型。

这八个模块的组织方式值得学：**入口统一、执行分离、模型共用**。所以加一个新场景，只需要写一个执行器，不用动入口；而所有场景的参数结构是一致的，用起来不会一个命令一个样。

板书：三组 —— **入口**：`chaosblade`（CLI＋HTTP；create／destroy／query／prepare／revoke；命令与场景参数说明）｜**执行器**：`exec-os`（基础资源）·`exec-docker`（Docker API）·`exec-cri`（CRI）·`chaosblade-operator`（K8s 资源方式，kubectl／client-go／CLI）·`exec-jvm`（**Java Agent 动态挂载**）·`exec-cplus`（**GDB**，方法／代码行级）｜**模型**：`chaosblade-spec-go`（Golang 模型定义）。右侧："入口统一·执行分离·模型共用"。

提问：为什么 Java 场景要单独做一个 `exec-jvm`，而不是像 CPU、内存那样用操作系统接口去做？

预设回答：

- 答"因为 Java 跑在虚拟机里，外面看不到方法"——正确。JVM 内部的方法调用对操作系统是不可见的，要动它只能进到 JVM 里面。

- 答"因为 Java 慢，需要专门优化"——不是原因。分开的理由是**注入点不同**：一个是进程外的资源，一个是进程内的字节码。

- 答"那用 GDB 也行"——技术上能挂，但操作字节码不如 Java Agent 干净；这正是 `exec-cplus` 用 GDB、`exec-jvm` 用 Agent 的原因：**按语言选手段**。

- 收口：注入手段必须匹配注入点所在的层次。

易错点 / 考点：考点是"组件与场景／技术手段的对应"。常考三对：**exec-jvm ↔ Java Agent**、**exec-cplus ↔ GDB**、**exec-docker ↔ Docker API**／**exec-cri ↔ CRI**。易错点是把 `chaosblade-operator` 当成独立平台——它是 **K8s 场景的执行器**，走的是 K8s 资源的创建、更新、删除这条路。

过渡：这些模块拼在一起长什么样？下一页给一张构成图。

【7.167】ChaosBlade 构成（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是 ChaosBlade 的构成图。要看**它们之间是什么关系**，分三层说。

最外层是**调用方**：你可以从命令行直接敲，也可以通过 HTTP 接口调用，还可以从 K8s 那边用资源的方式下发。三个入口，最后都收进同一个地方——**实验模型**。`chaosblade-spec-go` 那份模型定义就是干这个的：不管你从哪儿发起，实验都被规整成同一套结构。

中间是**分发**：模型按场景类型分派给对应的执行器。基础资源走 OS 执行器，容器走 Docker 或 CRI 执行器，K8s 走 Operator，Java 走 JVM 执行器，C++ 走 C++ 执行器。注意 Operator 这一支有个特别之处：它跑在集群里，把"实验"变成 K8s 的一种资源对象，所以实验的创建、更新、删除都走 K8s 的机制。

最内层是**目标**：被注入的进程、容器、Pod、节点和它们承载的应用。

这张图要你记住的一句话是：**ChaosBlade 的可扩展性来自"模型在上、执行在下"**。加场景只加执行器，入口和模型都不用动。

> **板书：**三层结构 —— 上"**调用方**：CLI ／ HTTP ／ K8s 资源" → 中"**统一实验模型**（场景・动作・参数）" → 下"**执行器**：OS｜Docker／CRI｜Operator｜JVM｜C++" → 底"**目标**：进程・容器・Pod・节点・应用"。
> 右侧："模型在上，执行在下 → 加场景只加执行器"。

提问：如果明天要给 ChaosBlade 加一个"Python 应用方法级注入"的场景，按这个构成，你该动哪一层？

预设回答：

- 答"改入口，加一个命令"——不用。入口是统一的，命令按模型自动生成。

- 答"加一个 Python 执行器"——正确，模型里描述"哪个模块、哪个函数、注什么"，执行器负责真正挂上去。

- 答"改模型定义"——只有在现有模型表达不了这种场景时才需要，通常方法级注入用现有模型就能描述。

- 收口：这就是分层的好处——扩展点被收敛到了一个地方。

易错点 / 考点：考点是"统一模型"的作用：**让不同场景的实验定义结构一致、可复用、可编排**。常见错答是把 CLI 和 HTTP 当成两套实现——它们是同一个入口的两种调用方式。判断方法：看这张图里**唯一被所有分支共用**的那一层是什么，答案就是"实验模型"。

过渡：结构清楚了，我们跑一个最小的例子，看看命令长什么样。

【7.168】一个简单的演示（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页给的是一个最小演示。课件只写了标题和官方文档地址，我按 ChaosBlade 的标准用法把流程走一遍，命令的具体参数以官方文档为准。

第一步是**创建实验**。CPU 场景：`blade create cpu load --cpu-percent 80`——让 CPU 负载压到 80%。网络场景：`blade create network delay --time 3000 --interface eth0 --local-port 8080`——在指定网卡、指定端口上注入 3 秒延迟。这两条命令的共同点是：**只说要什么效果，不说怎么做**，具体怎么落到系统调用上，是执行器的事。

第二步是**拿到实验标识**。创建成功会返回一个 UID，这个 UID 是你后面查询和销毁的唯一凭据。注意这一步很容易被忽略：演示完忘了记 UID，实验就一直挂着，第二天还在压你的 CPU。

第三步是**查询**：用 `blade status` 之类的方式确认实验还在不在、注了多久、参数是什么。第四步是**销毁**：`blade destroy <UID>`。有些场景注入前需要准备环境，对应 `blade prepare`，撤销用 `blade revoke`。

这个演示要你记住的不是命令本身，而是那个**四条动作的节奏**：创建 → 记录 → 观察 → 销毁。真实演练里还要加两条：注之前说清假设和稳态指标，注之后把结论归档。**没有销毁动作的实验，等于给自己埋了一颗定时炸弹。**

> **板书：**四步 —— ① **create**（`blade create cpu load --cpu-percent 80` | `blade create network delay --time 3000 --interface eth0 --local-port 8080`）→ ②**记 UID**（查询与销毁的唯一凭据）→ ③ **status 观察**（是否在跑・参数・持续时长）→ ④ **destroy <UID>**（环境类场景配 `prepare` / `revoke`）。右上角标："参数以官方文档为准"。底部："没销毁的实验 = 定时炸弹。"

提问：演示时创建了实验，忘了销毁，第二天业务出现莫名其妙的抖动，你会先查什么？

预设回答：

- 答"先查最近有没有发布"——常规思路，但会白查半天。

- 答"先查有没有残留的混沌实验"——正确。`blade status` 一敲就知道，这是演练环境里最常见的"人为故障"。

- 答"重启机器就好了"——能好，但你把证据也一起销毁了，下次还会再犯。

- 收口：演练环境的第一排查项，永远是"有没有人还在注故障"。

> **易错点 / 考点：**考点是"实验生命周期管理"：**创建—查询—销毁**，以及"环境准备—环境撤销"这一对。判断题"实验执行完会自动恢复"——错，多数注入需要显式销毁。答题要点：任何注入动作都要配一个**回退动作**，并且回退动作要在注入前就确认可用。

> **过渡：**ChaosBlade 的根在单机。可现在的系统跑在云上、跑在容器里，故障的形态变了——下一页讲这个变化带来了什么新要求。

【7.169】云原生架构下混沌工程的引入（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页讲的是"为什么云原生架构下需要重新想一遍混沌工程"。课件用四组对照来说明：**每注入一次故障，同时验证四项能力**。
>
> 第一项，**单点故障容忍能力**：注入单实例故障，业务 SLA 没有波动，业务不受影响。注意这里的前提是"单实例"——云原生架构默认认为实例会挂，所以架构上就该设计成挂一个不影响整体。
>
> 第二项，**状态可感知**：注入故障之后，PaaS 平台能及时感知到实例异常，异常实例能自动迁移成功，而且业务指标不受影响。这一项考的是**平台的感知能力**，不是你的代码。
>
> 第三项，**通过 naming service 访问**：异常实例迁移之后，实例列表变了，上游对这个服务的访问不受影响。这一项考的是**服务发现**——如果上游把实例地址写死了，迁移等于断链。
>
> 第四项，**部署满足单一包描述**：注故障引发了服务扩容，PaaS 平台自动执行扩容，不需要人工做前置或后置操作。这一项考的是**声明式部署**——你描述"要几个副本"，平台自己补齐。
>
> 这四项合起来说明一件事：**云原生把"故障处理"的大部分责任交给了平台**，于是混沌实验的验证对象也就不只是应用代码，还包括平台能力：调度、迁移、服务发现、弹性伸缩。正因为验证对象变了，才需要一个原生于 K8s 的工具——这就引出了 Chaos Mesh。

> **板书：**四组"注入 → 验证" —— ① 单实例故障 → **SLA 无波动**｜②注入故障 → **PaaS 感知 • 异常实例自动迁移**｜③注入故障 → **实例迁移不影响上游（naming service）**｜④注入故障 → **自动扩容，无人工前后置（单一包描述）**。底部箭头："验证对象从'应用代码'扩到'平台能力' → 引出面向云原生的 Chaos Mesh"。

> **提问：**在 K8s 上注入了 Pod 故障，Pod 被杀掉、新 Pod 起来了、业务也没受影响——这个实验能说明什么，又说明不了什么？

> **预设回答：**

- 答"说明高可用做到了"——只说对一半。它说明**副本冗余和自动重建**生效了，不代表服务发现、连接池、会话状态都没问题。

- 答"还要看上游有没有报错、有没有长连接断连"——很到位。Pod 重建的过程中，上游的连接池和注册中心缓存是典型的薄弱点。

- 答"说明平台没问题，代码也没问题"——过度推断。这次实验只覆盖了"杀 Pod"这一种失效模式，网络、IO、时钟都没验。

- 收口：一次实验只证明它验过的那一件事，别扩大解释。

> **易错点 / 考点：**考点是"云原生下混沌实验的验证对象扩展"：从**应用自身**扩展到**调度、迁移、服务发现、弹性伸缩**四项平台能力。常见错答是"云原生平台自带高可用，不需要混沌工程"——恰恰相反，平台能力同样会失效，而且失效时更难察觉。

> **过渡：**所以要有一个原生于 K8s 的工具。它就是下一页的主角。

【7.170】Chaos Mesh（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是 **Chaos Mesh** 的发展时间线，我按"出身—开源—进 CNCF—版本"四个节点讲。
>
> **出身**：它最早是 TiDB 团队在 2019 年做的一个测试框架，名字叫 "Schrödinger's Cat"（薛定谔的猫）。**开源**：2019 年 12 月 31 日在 GitHub 上开源。**进 CNCF**：2020 年 2 月 6 日进入 CNCF 云原生全景图；2020 年 7 月 14 日成为 CNCF 沙箱项目；2022 年 2 月成为孵化项目。**版本**：2020 年 9 月 26 日发布 1.0 GA，2021 年 7 月 23 日 2.0 GA，2021 年 11 月 30 日 2.1 GA。
>
> 这里要补一个背景：CNCF 用**鸿沟理论**管理项目，分**沙箱（Sandbox）→ 孵化（Incubating）→ 毕业（Graduated）**三级。这个分级不是荣誉称号，它反映的是项目的采用广度和治理成熟度。所以看一个云原生工具靠不靠谱，先看它在哪一级，是个很实用的判断方法。

从这条时间线还能读出两件事。一是**它从测试框架长成了平台**：起点是"给某个数据库做测试"，现在覆盖网络、IO、时间、内核、压力各类故障。二是**它的迭代节奏很快**：不到两年从 1.0 走到 2.1。

顺带说一句：起点是 TiDB，意味着它一开始就是为"分布式有状态系统"设计的，这类系统对一致性、时钟、IO 尤其敏感，所以它的故障类型里才有 TimeChaos、IOChaos 这种细分类。

板书：一条时间线 —— **2019** 内部测试框架 "Schrödinger's Cat"（为 TiDB）→ **2019.12.31** GitHub 开源 → **2020.2.6** CNCF 全景图 → **2020.7.14** CNCF Sandbox → **2020.9.26** 1.0 GA → **2021.7.23** 2.0 GA → **2021.11.30** 2.1 GA → **2022.2** CNCF Incubating。下方一行："CNCF 分级（鸿沟理论）：Sandbox → Incubating → Graduated"。

提问：一个工具从"CNCF 沙箱"升到"孵化"，通常意味着什么变化？

预设回答：

- 答"说明它更成熟了，可以用了"——方向对，但理由要说准：升级看的是**采用者数量、社区治理、发布规范**这些指标，不只是代码质量。

- 答"说明有更多公司在生产环境用"——这是核心指标之一。生产采用是分级评估里权重很高的一项。

- 答"沙箱项目就是玩具，别用"——过头了。沙箱意味着"早期、接口可能变"，不代表不可用；关键看你愿不愿意承担接口变动的成本。

- 收口：分级是"采用风险的提示牌"，不是"能否用的许可证"。

易错点 / 考点：常考"CNCF 三级分级的名称与顺序"，标准答案：**Sandbox → Incubating → Graduated**。易错点是把 Chaos Mesh 的出身记成"某云厂商的产品"——它源自 **TiDB 团队的测试框架**。若考时间，记住 2019 开源、2020 沙箱、2022 孵化三个锚点即可。

过渡：它在 K8s 上怎么表达一个实验？靠的是声明式 API。

【7.171】声明式 API 定义简化混沌实验管理（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲 Chaos Mesh 最关键的一个设计选择：**用声明式 API 来定义混沌实验**。

什么是声明式？命令式是"你告诉系统怎么做"，声明式是"你告诉系统要什么状态"，剩下交给系统达成。

Chaos Mesh 把这个思路搬到了混沌实验上：你写一份 YAML，说明"我要对哪批 Pod 注入什么故障、持续多久"，控制器负责把它变成现实。

落地手段是 **CRD（Custom Resources）扩展 Kubernetes API**。课件列了几种：**PodChaos**（杀 Pod、让 Pod 不可用）、**NetworkChaos**（网络延迟、丢包、分区）、**IOChaos**（文件读写延迟、错误）、**TimeChaos**（时钟偏移）、**StressChaos**（CPU、内存压力）、**KernelChaos**（内核级故障），类型还在增加。

为什么能简化管理？三个好处。**第一，实验成了普通资源**：创建、查看、删除都用 kubectl，和你管 Deployment 是一套手感。**第二，实验进版本库**：YAML 能提交、能 review、能追溯"谁在什么时候改过实验参数"。**第三，天然可自动化**：任何能操作 K8s 的东西——CI 流水线、Operator、GitOps 工具——都能操作实验。

对比命令式：命令行敲一条命令，没留痕、难复用、换个人重现不了。**声明式把实验从"一次操作"变成了"一份资产"。**

板书：左："命令式＝告诉系统**怎么做**（一次性・难复用・难追溯）" ｜ 右："**声明式**＝告诉系统**要什么状态**（YAML・可版本化・可自动化）"。中间："**CRD 扩展 K8s API**"；下列六类：`PodChaos`・`NetworkChaos`・`IOChaos`・`TimeChaos`・`StressChaos`・`KernelChaos`……。底部："实验从'一次操作'变成'一份资产'。"

提问：为什么 Chaos Mesh 选择"给 K8s 加一种资源类型"，而不是"提供一个命令行工具"？

预设回答：

- 答"因为 K8s 就是声明式的，顺着生态走"——准确。复用了 K8s 的控制器模式、权限体系、审计日志，等于免费继承了一整套基础设施。

- 答"因为 YAML 好写"——表层理由。好写只是副作用，核心是**状态可被持续调谐**：实验被删了、节点重启了，控制器还能把状态拉回你想要的样子。

- 答"因为命令行不方便"——不成立。好的命令行工具一样好用，差别在**可复用与可审计**。

- 收口：选择声明式，本质是选择"接入生态"，而不是"多一种写法"。

易错点 / 考点：考点是"声明式 vs 命令式"的辨析，以及"CRD 的作用"。常见错答是把 CRD 说成"一种配置文件格式"——它是**扩展 K8s API 的机制**，注册之后 `kubectl get podchaos` 这样的查询才成立。若考六类资源，

记住 Pod／Network／IO／Time／Stress／Kernel 六个前缀即可。

过渡：有了资源类型，还得有人去读它、执行它。我们看架构。

【7.172】Chaos Mesh 架构（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 Chaos Mesh 的架构图，四个组件，我按"人怎么用—谁来指挥—谁去动手"这条线讲。

Chaos Dashboard 是给人用的：通过 UI 管理和监控混沌实验。它解决的是"降低门槛"——写 YAML 对熟手很轻松，对第一次用的人就是一道坎，有个界面能点选实验类型、选目标、看状态。

Chaos Controller Manager 是大脑：它做调度和控制，同时是**编排引擎**。前面讲的声明式资源就是它来读的——它监听那些 Chaos 资源的创建、更新、删除，然后决定"现在该注入、该恢复，还是该等着"。

Chaos Daemon 是 K8s 环境里的执行组件：真正动刀子的活它在干。因为要侵入命名空间、改 cgroup、动网络规则，这些动作需要节点上的高权限，所以它是以 DaemonSet 的形式跑在每个节点上的。

Chaosd 是 non-K8s 环境的执行组件：给那些还没上 K8s 的机器用。说明它的使用面不限于集群内——虚拟机、物理机上也能用同一套方法做实验。

把这四个组件连起来看，就是一条清晰的分工线：**Dashboard 管体验，Controller Manager 管决策，Daemon 与 Chaosd 管执行**。这也是所有云原生控制器类系统的标准分层：**期望状态（声明）→ 决策（控制器）→ 实际动作（agent）**。你以后看 Operator、看各种 K8s 工具，基本都是这个骨架。

板书：四个方块 —— **Chaos Dashboard**（UI・管理与监控）｜**Chaos Controller Manager**（调度与控制・**编排引擎**）｜**Chaos Daemon**（**K8s 环境执行**，节点级）｜**Chaosd**（**non-K8s 环境执行**）。中间一条流程："声明（资源）→ 决策（Controller Manager）→ 执行（Daemon／Chaosd）"。底部："体验／决策／执行，三段分工。"

提问：为什么执行组件要跑在每个节点上，而不是集中部署成一个服务？

预设回答：

- 答"因为它要动本机的网络和 cgroup，必须在本机"——正确。网络命名空间、cgroup、文件系统这些资源都是**节点本地**的，跨节点操作做不到。

- 答"因为要分摊压力"——不是主因。真正的理由是**权限与位置**：注入动作需要节点级权限，也只能在节点上生效。

- 答"这样可以少开端口更安全"——有点道理，但主要还是能力和位置决定的。

- 收口：执行组件的位置，由它要操作的对象决定。

> **易错点 / 考点：**考点是"四个组件的职责划分"，特别是 **Dashboard（界面）／Controller Manager（决策与编排）／Daemon（K8s 执行）／Chaosd（非 K8s 执行）**。最常见的错答是把 Chaos Daemon 和 Chaosd 混成一个——前者在集群里，后者在集群外。答题要点：先问"这个组件在哪儿跑"，再问"它负责决策还是执行"。

> **过渡：**Controller Manager 具体在盯什么？下一页说得很具体。

【7.173】监听资源变化（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲控制器的工作方式，三句话就能说清，但它解释了整个系统"为什么能自动"。
>
> 第一句：**监听** PodChaos、NetworkChaos 等资源的创建、更新、删除。控制器不主动去轮询"现在该做什么"，而是一直盯着这些资源对象的变化。你提交一份 PodChaos，它立刻知道；你把 duration 改短，它也知道；你删掉它，它同样知道。
>
> 第二句：**决定当前该注入、恢复，还是等待**。收到变化之后，它要判断此刻的真实意图：新建的资源意味着"开始注入"；被删除或到期意味着"恢复"；如果资源还在但条件没满足（比如目标 Pod 还没就绪），那就是"等待"。注意"等待"也是决策的一种，这一步最容易被忽略——很多自动化系统出问题，就出在"该等的时候它动手了"。
>
> 第三句：**分发**。如果是简单注入，比如 PodKill，它自己就能做；如果需要动节点上的东西，就向 **Chaos Daemon 或 Sidecar** 发送请求，由它们去执行。
>
> 这三句话合起来就是一个"**监听—判断—执行**"的循环。为什么这个循环重要？因为它让系统具备了**自我纠偏**能力：实验资源被误删了，恢复动作会自动执行；节点重启了，控制器再看到资源还在，会重新把状态调谐回来。这就是声明式的好处落到实处的地方——你描述状态，系统负责守住它。

> **板书：**三步循环 —— ①**监听**（PodChaos／NetworkChaos… 的创建・更新・删除）→ ②**判断**（注入／恢复／**等待**）→ ③**执行**（简单注入如 **PodKill** 自行处理；需节点操作的转 **Chaos Daemon／Sidecar**）。旁注："自我纠偏 = 声明式的价值落地"。

提问：控制器判断出"该等待"和判断出"该注入"，哪一个更容易出错？为什么？

预设回答：

- 答"注入更容易出错，因为动作危险"——危险是事实，但"等待"的判断错了后果更隐蔽：明明条件没满足就注入，实验结论全是假的。

- 答"等待更容易错，因为条件很多"——很到位。等待要看目标是否存在、是否就绪、上一个实验是否恢复完，任何一个条件判错，都会导致误注入或漏注入。

- 答"两个一样"——不对，两者的失败模式不同：注入错了是"制造了不该有的故障"，等待错了是"该测的没测到"。后者不会报错，最难发现。

- 收口：自动化的最大风险，往往不是"做错"，而是"悄悄没做"。

易错点 / 考点：考点是"控制器三种决策状态：注入／恢复／等待"，以及"简单注入与 Daemon 执行的区分"。常见错答是只记"注入和恢复"两种——**等待**这一项是理解声明式系统的关键。判断方法：把实验当成"期望状态"，控制器做的永远是"让实际状态追上期望状态"，追不上的时候就该等。

过渡：分发到执行器之后，故障究竟是怎么"进到"容器里的？这是最硬的一页。

【7.174】故障如何注入？（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页回答一个很本质的问题：**故障到底是怎么注进去的**。课件给了三行关键内容，我们一行一行拆。

第一行，**监听**——和上一页一样，先感知资源变化。这是前提。

第二行，**容器的实体是什么**：它不是一个黑盒，它由三样东西构成——**进程**、**Namespace**、**Cgroup**。进程是真正在跑的东西；**Namespace 控制可见性**，也就是这个进程能看到哪些网络、哪些挂载点、哪些进程；**Cgroup 限制资源**，也就是它能用多少 CPU、多少内存、多少 IO。

第三行，**注入的实质**：**侵入 Namespace 和 Cgroup，进行干扰**。你要让网络变慢，就去动它的网络命名空间，

加一条 tc 规则；你要让磁盘变慢，就去动它的挂载与 IO 限制；你要让它 CPU 不够用，就去动 cgroup 的配额，或者直接起一堆抢占进程；你要让它"看不见"别的服务，就在网络命名空间里改路由或 DNS。

所以"注入故障"听起来玄，落到系统层面就是两件事：**改它能看到什么，改它能用多少**。这也解释了为什么这类工具必须有节点级权限——命名空间和 cgroup 都是内核资源，没有权限碰不到。

反过来说，理解了这一层，你就能自己判断"某个故障类型能否实现"：凡是能通过改可见性或改资源限制表达出来的故障，都能注入；表达不出来的，就得靠应用层手段，比如 Java Agent 改方法、GDB 改返回值。

板书：三行 —— ①**监听**资源变化 → ②**容器实体 = 进程 + Namespace（控制可见性）+ Cgroup（限制资源）** → ③**注入实质 = 侵入 Namespace／Cgroup 进行干扰**。右侧举例：网络变慢→网络命名空间加 tc｜CPU 不足→cgroup 配额／抢占进程｜找不到服务→改路由／DNS。底部："改它能看到什么、改它能用多少。"

提问："让容器内的 DNS 解析失败"这个故障，按这一页的框架，是改 Namespace 还是改 Cgroup?

预设回答：

- 答"改 Cgroup，因为是资源问题"——错。DNS 解析属于"能否看到/找到"，是**可见性**问题。

- 答"改 Namespace，比如改它的 DNS 配置或网络命名空间里的规则"——正确。诸如此类的"找到不找到""通不通"都归可见性一侧。

- 答"改应用配置也行"——也能实现，但那是应用层手段，不是平台级注入；两者适用场景不同。

- 收口：先判断这个故障影响的是"看得见"还是"够不够用"，手段自然就出来了。

易错点 / 考点：考点是"注入的底层实质"：**Namespace 管可见性、Cgroup 管资源限额**。这是本节最容易被跳过、也最能区分"会用工具"和"懂原理"的一页。常见错答是把注入理解成"发一个特殊请求给应用"。答题三步法：**这个故障改变了什么 → 归到可见性还是资源 → 对应动哪个内核机制**。

过渡：原理清楚了，我们看一份真实的实验定义长什么样。

【7.175】使用 Chaos Mesh 定义混沌实验（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是一份 PodChaos 定义，逐行读。它回答五个问题：类型、名字、空间、怎么注、注谁。

`apiVersion: chaos-mesh.org/v1alpha1` 是接口版本；`kind: PodChaos` 声明这是 Pod 级故障。

`metadata` 两个字段：`name: pod-failure-example` 是实验名字，起名要看得出意图，别用 `test1`；`namespace: chaos-testing` 是**实验资源所在的空间**，不是被注入 Pod 的空间。

`spec` 是核心，四字段。`action: pod-failure`——动作是"让 Pod 失败"，即一段时间内不可用，而不是删掉它；删除会被重建，失败则是"原地不动但不可用"，更接近真实的部分故障。`mode: one`——从目标里挑一个；这一行是**爆炸半径**的直接体现，改成 `all` 就是全选，风险上一个量级。`duration: '30s'`——三十秒，到点自动恢复。

最后 `selector` 用 `labelSelectors` 选中 `app.kubernetes.io/component: tikv` 的 Pod——**用标签选目标，不写 Pod 名字**。Pod 名字每次重建都会变，标签是稳定的：**按标签圈范围，不按个体点名**。

最该记的是两行：`mode` 和 `duration`——爆炸半径的两个旋钮：**影响几个**、**影响多久**。

板书：五行注释 —— `apiVersion`（接口版本）｜`kind: PodChaos`（实验类型）｜`metadata`：`name`（意图化命名）·`namespace`（实验空间）｜`spec.action: pod-failure`（不可用 ≠ 删除）｜`spec.mode: one`（**半径旋钮①：影响几个**）·`duration: '30s'`（**半径旋钮②：影响多久**）｜`selector.labelSelectors`（**按标签圈范围，不按名字点名**）。

提问：把 `action` 从 `pod-failure` 改成删除 Pod，实验验证的能力有什么不同?

预设回答：

- 答"没区别，反正 Pod 都没了"——错。`pod-failure` 是让 Pod 不可用但对象还在，删除是对象消失后被重建；前者考验"流量摘除与超时"，后者考验"重建与调度"。

- 答"删除会触发重建，验证的是自愈；failure 验证的是故障隔离"——准确。两种失效模式对应两套机制。

- 答"那就都注一遍"——对，这也说明"动作"本身就是实验设计的一部分，不是随手选的。

- 收口：选动作，等于选你要验证的那条恢复路径。

易错点 / 考点：考点是"YAML 各字段含义"与"爆炸半径的两个旋钮（mode／duration）"。常见错答是以为 `mode: one` 表示"注入一次"——它表示**选中一个目标**，注入时长由 duration 决定。另一易错点：`namespace` 是实验资源所在空间，不是目标 Pod 的空间，选目标靠 `selector`。

过渡：一个实验只有一种故障。真实事故往往是好几个故障叠在一起——所以需要编排。

【7.176】编排混沌实验（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲 **Workflow**：把多个实验编排成一个流程。为什么要编排？真实事故很少是单一故障：先网络抖动，重试打满下游，线程池耗尽，链路雪崩。只测单点，测不出连锁反应。

构成四块：**名称**；**Entry**，入口，引用任一声明过的 Template——是"引用"不是"复制"，同一模板可多处复用；**Template 数组**，定义各行为；**五种模板类型**：**Serial**（串行）、**Parallel**（并行）、**Chaos**（具体实验）、**Suspend**（挂起，模拟"故障已发生、系统在挣扎"的窗口）、**Task**（任务，做检查或触发外部动作）。Serial、Parallel、Task 可把其他节点**以引用方式作为子节点**，于是整个 Workflow 成了一张**有向图**。

两个要点要说。**第一，必须唯一入口**：从 entry 能走到每个节点，执行顺序才确定，也才好排查"卡在哪一步"。**第二，用引用不用嵌套**：定义一次、多处引用，改一处全改，维护几十个实验很好用。

骨架两行：`entry` 指向某个模板名，`templates` 里逐个列 `name` 和 `templateType`——**入口 + 模板清单**，剩下控制器按图推。

`Suspend` 看起来"什么都没干"，其实是编排里最有用的一环——注入后要**留观察窗口**，让指标暴露影响，再决定加大还是收手。

板书：构成 —— **名称**｜**entry（入口，引用任一 Template）**｜**templates 数组**｜五种类型：`Serial`（串行）·`Parallel`（并行）·`Chaos`（具体实验）·`Suspend`（挂起＝观察窗口）·`Task`（任务／检查）；Serial／Parallel／Task 可**引用**子节点 → 形成**有向图**。骨架两行：`entry: <模板名>`、`templates: [{name, templateType}, …]`。底部："模板定义一次、多处引用。"

提问：编排里为什么要专门提供 `Suspend` 这种"什么都不做"的模板？

预设回答：

- 答"用来做延时，模拟慢"——不是。延时是故障的一种（TimeChaos 或网络延迟），Suspend 是编排层的暂停。

- 答"用来留观察窗口，等指标反映出来"——正确。故障注进去到业务指标恶化往往有滞后，没有这个窗口，你会在"还没显现代价"的时候就把实验停掉。

- 答"用来人工确认，需要人点一下继续"——也有这种用法，把人工审批卡在流程里。

- 收口：编排不只是"排顺序"，还要排"什么时候停一停"。

> **易错点 / 考点：**考点是"五种模板类型"与"有向图 + 唯一入口"的结构。常见错答是把 `Task` 当成"一个混沌实验"——混沌实验是 `Chaos`，`Task` 一般用来做检查或调用外部动作。记忆口诀：**串、并、混、挂、任**（Serial／Parallel／Chaos／Suspend／Task）。

> **过渡：**Workflow 把故障按时间摆好，但真实用户遇到的是"一连串麻烦"。下一页把实验的组织单位从"故障"换成"场景"。

【7.177】用户场景演练（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页把视角从"组件"切到"用户"。前面讲的都是"注入一个故障、看一个服务"，这一页要问的是：**用户能否把一件事做完**。

为什么要有这一层？因为系统可以做到"每个组件都没挂、每个接口成功率都 99.9%"，而用户依然完不成下单——可能一个非关键的推荐服务在拖慢主流程，可能一个校验接口超时后静默返回空，用户看到的是"提交失败，请稍后再试"。**组件的健康度不等于业务的可用性**，这中间有一条很宽的缝。

做法是：按用户旅程把动作串起来——打开页面、登录、搜索、加购、下单、支付，每步对应一条链路和一组指标；然后把多个故障编排进同一条旅程里，让它们在时间上有先有后，模拟"一件事还没缓过来，第二件事又来了"。这时候你观察的不是某个服务的错误率，而是**这条旅程的整体完成率、每一步的耗时、以及失败时用户看到什么**。

关键词是"**编排**"——上一页的 Workflow 就是为这个准备的：串行把故障按时间叠上去，并行模拟多处同时出问题，挂起留观察窗口。**单点实验回答"这个零件会不会坏"，场景演练回答"用户这件事还能否办成"。**

> **板书：**一条用户旅程箭头 —— **打开页 → 登录 → 搜索 → 加购 → 下单 → 支付**；下方标"每一步＝一条链路＋一组指标"；上方标"故障按时间叠加（串行）／多处同时（并行）／观察窗口（挂起）"。右侧一句："**组件健康 ≠ 业务可用**；单点实验看零件，场景演练看旅程。"

> **提问：**所有服务的成功率都是 99.9%，用户却反馈"下单老是失败"，这种情况最可能的原因是什么？

> **预设回答：**

- 答"统计数据不准"——有可能，但更常见的不是统计问题。

- 答"失败集中在某一步，而那一步不是核心服务"——很到位。比如校验、风控、推荐这类"非核心"环节失败时，主流程可能被直接拦住。

- 答"重试把延迟放大了"——也对，这是典型连锁：单次调用没超阈值，重试叠加后整体超时。

- 收口：所以要按旅程看指标，而不是按服务看指标。

> **易错点 / 考点：**考点是"组件可用性与业务可用性的区别"，以及"场景演练需要编排能力"。常见错答是把场景演练等同于"多注几个故障"——关键差别在**是否按用户旅程组织、是否观察端到端指标**。判断三步法：**这条链路用户能否感知 → 断了用户看到什么 → 有没有替代路径**。

> **过渡：**场景演练是在"上线前或演练窗口"做的。有没有办法让它更早、更自动地发生？

【7.178】Pull Request 场景（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲的是一个很"左移"的用法：**把混沌实验挂到 Pull Request 上**。也就是每次有人提代码，实验自动跑一遍。
>
> 逻辑很简单：韧性是一种会被"改坏"的属性。今天你的服务能扛住下游超时，是因为代码里有超时控制和降级分支；明天有人为了图快，把超时时间从 2 秒改成 30 秒，或者把一个降级分支删掉"反正没触发过"，编译能过、单测能过、功能测试也能过——因为正常路径没变。**只有注入一次故障，才知道韧性还在不在。**
>
> 所以做法是：PR 提上来，自动部署到预览环境（这就是上一节讲的"预览环境特定流量"那一格），跑一组**小而快**的混沌实验——半径限制在这一个 PR 涉及的服务的单个实例上，时长几十秒，只验几条最关键的假设：下游超时有没有被兜住、依赖不可用时有没有降级、有没有把重试放大成风暴。通过了才允许合并。
>
> 这里有两个工程约束必须说清。**第一，实验必须快**：PR 门禁如果跑二十分钟，开发就会绕过它；所以只跑"必过集"，不跑全量。**第二，实验必须有稳定判据**：门禁不允许"看起来还行"，必须是指标越线就红。噪声大的实验放在门禁里，很快就会被团队用"重跑一次"给废掉。
>
> 这一页的意义在于：**混沌工程从"季度演练"变成了"每次提交都要过的一关"**，韧性和功能一样，进入了持续验证的轨道。

> **板书：**一条流水线 —— **提交 PR → 部署预览环境 → 跑"必过集"混沌实验（小半径 · 单实例 · 几十秒）→ 指标越线则阻止合并**。右侧两条约束："**必须快**（否则被绕过）"｜"**判据必须稳**（噪声大就被'重跑一次'废掉）"。底部："韧性也会被改坏 → 韧性要进 CI。"

提问：把混沌实验放进 PR 门禁，最大的现实阻力是什么？

预设回答：

- 答"环境不够用"——是真实阻力之一，所以通常只在预览环境跑，并且只注一个实例。

- 答"太慢，开发等不了"——最普遍的阻力。解决办法是拆成"必过集"和"全量集"，门禁只跑前者。

- 答"实验不稳定，一会儿红一会儿绿"——这个最致命。不稳定会被当成噪声，然后被忽略，最后被关掉。

- 收口：门禁类实验的第一要求不是覆盖广，而是**判据稳、跑得快**。

易错点 / 考点：考点是"混沌实验的两种节奏"：**演练式（周期性、大半径、深度验证）**与**门禁式（每次提交、小半径、快速回归）**。常见错答是把两者混为一谈，要求门禁也做全链路验证——那会让流水线彻底堵死。判断方法：先问"这次实验的目的是发现新风险，还是防止已有韧性退化"。

过渡：从工具到实践，我们看到了一条完整的路。国内有团队把这套东西做成了一个体系，下一页开始讲它。

【7.179】从故障注入到混沌工程 —— KeChaos 2.0（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页的标题本身就是结论：**从故障注入到混沌工程**。这中间差的是什么？差的是"体系"。故障注入是动作，混沌工程是方法加流程加度量。KeChaos 2.0 就是把这套体系化的过程画了出来。

它把一次完整的实验拆成五步。**第一步，定义实验场景**——先想清楚要验证什么业务场景，而不是先想"我能注什么"。**第二步，关联注入能力**——场景定下来之后，再看手上哪些工具能实现它，把能力和场景对上。**第三步，确定监测稳态标准**——哪些指标、什么阈值算正常，这一步必须在注入前完成。**第四步，实验运行 + 爆炸半径控制**——跑起来，同时把影响范围摁住。**第五步，实验终止产出报告**——结束，并把结论沉淀下来。

这五步里有三个细节值得专门点出来。**"关联注入能力"放在第二步**，说明它是"按需找工具"，不是"拿工具找场景"——这个顺序决定了你是问题驱动还是工具驱动。**"爆炸半径控制"和"运行"写在同一格里**，说明半径控制不是运行前的一次性设置，而是运行过程中持续要做的事。**"产出报告"是流程的正式一环**，不是可选项

——没有报告，实验就没有进入下一次决策的输入。

还有一个词要解释：**"稳态标准"**。它和前面讲的"稳态指标"是一回事，强调"标准"两个字，意思是这些指标要有明确的、事先约定的阈值，而不是事后看着曲线说"大概还行"。

板书：五步竖排 —— ①**定义实验场景**（先想验证什么）→ ②**关联注入能力**（按场景找工具，不是拿工具找场景）→ ③**确定监测稳态标准**（指标＋阈值，注入前定）→ ④**实验运行＋爆炸半径控制**（运行中持续控半径）→ ⑤**实验终止产出报告**（结论入库，成为下次输入）。标题旁注："故障注入＝动作；混沌工程＝体系（方法＋流程＋度量）"。

提问：五步里"关联注入能力"排在"确定稳态标准"之前。如果把它挪到最后，会有什么后果？

预设回答：

- 答"没什么区别，都是准备"——有区别。顺序错了，你会先被工具能做什么框住，场景设计不自觉地向"工具好实现"倾斜。

- 答"会变成工具驱动，场景被工具裁剪"——正确。这就是很多团队"做了很多实验，却没回答业务关心的问题"的根源。

- 答"会导致稳态标准定得不合适"——也有这一层：标准应该跟着业务场景走，而不是跟着能注入的故障走。

- 收口：流程顺序本身就是一种方法论，它决定了你是问题驱动还是工具驱动。

易错点 / 考点：考点是"五步流程的名称与顺序"，以及"故障注入与混沌工程的区别"。常见错答是把两者当同义词——**故障注入是技术手段，混沌工程是包含假设、稳态、半径控制、报告沉淀的完整体系**。若考顺序，记住"**场景 → 能力 → 标准 → 运行控半径 → 报告**"，其中"标准必须在运行前定"是最硬的规则。

过渡：这套流程落到具体目标上，KeChaos 选的第一块阵地是基础服务的稳定性。

【7.180】基础服务稳定性风险治理（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲的是 KeChaos 怎么用混沌工程做**基础服务的稳定性风险治理**，其中还标出了它的两个演进阶段。

先看演进。**KeChaos 0.1 的定位是"核心基础服务稳定性风险探索"**——探索的意思是，还不知道风险在哪，先做一批实验把它们找出来。**KeChaos 1.0 的定位是"核心基础服务常态化的故障注入和异常测试、故障演练例**

行化"——注意"常态化""例行化"这两个词，意思是这件事从项目制的专项，变成了按周期自动跑的常规工作。这个演进路径很有代表性：**先探索找风险，再把验证变成例行**。

再看治理闭环，五个动作：**定义实验 → 做出假设 → 执行实验 → 修复问题 → 实验验证**。这五步不是一条直线，是一个圈——最后一步"实验验证"验完，如果发现新的薄弱点，又回到"定义实验"。

这个闭环里有两个点值得强调。**"做出假设"被单列成一步**，说明它是必须显式写下来的东西，不是心里想想。**"修复问题"后面必须跟"实验验证"**，这是最容易断的一环：问题修了，但没人回头用同一个实验复验一次，于是"修好了"只是感觉。

为什么选"基础服务"先做？因为基础服务是所有业务的地基——注册中心、配置中心、缓存、消息、数据库代理这类东西，一挂就是一大片。对它们做混沌工程的性价比最高：**影响面大、故障模式相对稳定、验证方法容易复用。**

板书：上排演进 —— **KeChaos 0.1**：核心基础服务稳定性**风险探索** → **KeChaos 1.0**：**常态化**故障注入与异常测试、故障演练**例行化**。下排闭环（画成圈）—— **定义实验 → 做出假设 → 执行实验 → 修复问题 → 实验验证**（回到定义实验）。右侧："基础服务＝地基：影响面大・故障模式稳定・方法可复用。"

提问：治理闭环里为什么要把"实验验证"再放回圈里，而不是"修复完成"就结束？

预设回答：

- 答"因为修复可能没修对"——对，这是最常见的情况：改的是现象，根因还在。
- 答"因为修复可能引入新问题"——也很真实。改超时、加降级、换依赖，都可能改变别的链路行为。
- 答"因为要留证据给领导看"——动机偏了。复验的价值是**确认假设成立**，不是攒材料。
- 收口：没有复验的修复，只能叫"改过了"，还说不上"修好了"。

易错点 / 考点：考点是"两个阶段的定位区别"：**0.1 探索（找风险）、1.0 常态化与例行化（守风险）**。常见错答是把"例行化"理解成"自动化"——例行讲的是**按周期纳入工作流**，自动化只是实现手段。另一考点：治理闭环是**循环**而非一次性流程，**修复后必须复验**。

过渡：闭环讲的是"怎么管"，下一页讲"具体怎么注"，也就是那套动作细则。

【7.181】基于 KeChaos 的故障注入（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把故障注入拆成四阶段，每阶段都有具体动作，是本章最"可照着做"的一页。
>
> **第一阶段，依赖梳理。** 四个动作：**链路分析**（请求经过了哪些服务）、**强弱依赖梳理**（哪些断了业务就停，哪些只是体验下降）、**依赖降级梳理**（每个依赖有降级方案吗）、**熔断限流降级能力梳理**（阈值、上限、开关在哪）。注意这一步一次故障没注。为什么先做？**不知道依赖关系，你连"会波及谁"都算不出来**，爆炸半径控不住。
>
> **第二阶段，故障注入。** 五个动作：**报备周知**（先告诉相关方）、**历史故障画像**（照过去出过的问题去注，命中率高）、**专注自身服务**（先验自己，别一上来就动别人）、**模拟下游异常**、**模拟依赖异常**。顺序有讲究：先报备，再照画像注，范围从自己往外扩。
>
> **第三阶段，报警验证。** 三个指标：**报警触达率**（该收到的人收到了吗）、**报警及时性**（多久收到的）、**报警有效性**（这条报警有没有指向真问题）。注入故障是验告警体系最便宜的办法，比等真事故便宜。
>
> **第四阶段，降级验证。** 三项：**降级能力建设**（有没有）、**限流有效性**（限得住吗）、**熔断、降级有效性**（真的会跳闸吗）。
>
> 四个阶段的共同逻辑：**先弄清结构，再小心注入，然后验"看得见"和"扛得住"**。注意后两阶段验的是**运维能力**，不是业务代码——这是很多人漏掉的一半。

> **板书：**四阶段（每阶段列子项）—— ①**依赖梳理**：链路分析｜强弱依赖｜依赖降级｜熔断限流降级能力 ②**故障注入**：报备周知｜历史故障画像｜专注自身服务｜模拟下游异常｜模拟依赖异常 ③**报警验证**：触达率｜及时性｜有效性 ④**降级验证**：降级能力建设｜限流有效性｜熔断降级有效性。底部："前两阶段弄清结构、小心注入；后两阶段验'看得见'和'扛得住'。"

> **提问：**为什么"报警验证"要单独做成一个阶段，而不是顺带看一下监控有没有报？

> **预设回答：**

- 答"顺带看就行，反正会报"——最常见的想法，也是最容易漏的。告警链路本身会坏：规则过期、通道失效、值班表没更新。

- 答"因为告警是给人看的，人不在就等于没报"——很到位。触达率和及时性量的就是这件事。

- 答"因为要看报警准不准"——对，这是有效性：一条每次都响的报警，等于没有报警。

- 收口：告警体系也是被测对象，注入故障是它最便宜的体检方式。

> **易错点 / 考点：**考点是"四阶段及各自子项"，尤其**报警验证的三性（触达率／及时性／有效性）**与**降级验证的三项**。常见错答是把"故障注入"当成整个流程——它只是第二阶段，前后都还有工作。答题要点：**先梳理再注入，注入后既验业务也验告警与降级**。

> **过渡：**到这里，7.4 混沌工程就讲完了。接下来进入 7.5，换一个完全不同的问题：功能都对，用户却不会用。

【7.182】7.5 易用性测试：A/B 测试（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 我们进入 7.5 节。前面几节问的都是"系统扛不扛得住""功能对不对"，这一节换的问题很短：**好不好用**。
>
> 为什么值得单独成节？因为"能用"和"愿意用"是两件事。你能登录、能下单、能支付，功能全对；但按钮放的地方找不到、错误提示是一串错误码、一个流程要点五次，用户就走了。这类问题**功能测试一条都抓不到**——用例写的是"点击提交，订单创建成功"，它不会问"用户知不知道该点哪里"。
>
> 易用性测试回答三层问题：**看得懂吗**（界面元素是否合习惯）、**走得通吗**（任务能否完成）、**愿意继续用吗**。三层对应三种方法和三种度量。
>
> 本节最后落到一个很"工程化"的手段——**A/B 测试**：把用户分组，同时在线跑不同版本，用真实行为数据判断哪个方案更优，把"我觉得更好"变成"数据显示更好"。**易用性本来是主观的，A/B 给它装了一把尺子。**

> **板书：**中间写"**7.5 易用性测试**"；上方一行："功能对 ≠ 用户会用"；下方三层问题 —— **看得懂吗**（界面元素）｜**走得通吗**（任务完成）｜**愿意继续用吗**（体验满意度）；右下角箭头指向"**A/B 测试**：把'我觉得'变成'数据显示'"。

> **提问：**一个功能所有测试用例都通过了，上线后用户却大量放弃，最可能是哪一类问题没测？

> **预设回答：**

- 答"性能问题"——有可能，但性能问题通常会表现为慢和超时，而不是"放弃"。

- 答"易用性问题"——正确。用户没找到入口、看不懂提示、流程太长，都会直接放弃。

- 答"需求做错了"——也对，但需求错通常表现为"用户不用这个功能"；"用了一半放弃"更像易用性问题。

- 收口：功能测试验"通不通"，易用性测试验"愿不愿"。

> **易错点 / 考点：**考点是"易用性测试的定位"：它属于**非功能测试**，验证的是用户使用过程中的效率、可理解性与满意度，**功能测试覆盖不到**。常见错答是把易用性等同于"界面好看"——好看只是其中一项，易用性还包括一致性、效率、容错提示。

> **过渡：**既然是"界面"起的作用最大，那就先把"好的界面"到底好在哪说清楚。

【7.183】良好的 UI 元素（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页给的是"好的用户界面"的七个要素，课件原话是七个：**符合标准和规范、直观性、一致性、灵活性、舒适性、正确性、实用性**。我按性质分三组，好记。
>
> 第一组是"**守规矩**"：**符合标准和规范**、**一致性**。前者是符合平台与行业的通用约定——手机上的返回手势、菜单位置、表单必填项的标记方式；后者是产品内部同类操作表达要一样：这页用"保存"，那页就写"提交"，按钮颜色、提示语气、错误码格式都要统一。**一致性之所以重要，是因为用户是靠"上次的经验"来用你的软件的**；每次都要重新学，成本就变成了摩擦。
>
> 第二组是"**好理解**"：**直观性**、**正确性**。直观性是看一眼就知道这是干什么的、点了会怎样；正确性包含两层，界面上的信息必须是对的（数字、状态、文案与实际一致），操作的结果也必须是可预期的（点了删除就真的删了，并且有确认与可撤销）。
>
> 第三组是"**好用**"：**灵活性**、**舒适性**、**实用性**。灵活性是指对不同用户给不同路径——新手有引导，熟手有快捷键，批量操作有入口；舒适性是视觉与操作负担要低：对比度够、点击区够大、不刺眼、不让人紧张；实用性是每一个元素都要有存在的理由，不做"看起来高级但没人用"的功能。
>
> 这七条不是美学偏好，它们是**可以被检查的**。做易用性测试时，拿这七条做一张检查单，一条条问"我们做到了吗"，比凭感觉说"我觉得不丑"靠谱得多。

> **板书：**七要素分三组 —— **守规矩**：符合标准和规范｜一致性 ｜ **好理解**：直观性｜正确性 ｜ **好用**：灵活性｜舒适性｜实用性。右侧："一致性＝用上次的经验就能用；正确性＝信息真、结果可预期；实用性＝每

个元素都要有理由。"

提问："这个页面用'保存'，那个页面用'提交'，功能都正常。"这违反七要素里的哪一条？会造成什么实际损失？

预设回答：

- 答"违反了直观性"——不准确。直观性讲的是"看一眼知道是什么"，这里讲的是同类表达统一，属于**一致性**。

- 答"违反一致性，用户会犹豫、会点错"——正确。措辞不统一，用户每次都要判断"这两个按钮是不是一回事"。

- 答"无所谓，用户会习惯"——这正是把摩擦当理所当然。摩擦积累起来就是放弃率。

- 收口：一致性问题看起来最像"小事"，却是最容易量化收益的一类改进。

易错点 / 考点：考点是"七要素的名称与归类"。常见错答是把"正确性"只理解成"程序不出错"——在易用性语境里，它还包括**信息真实、结果可预期**。另一错答是把"灵活性"当成"功能多"——灵活性强调的是**面向不同熟练度用户的多条路径**，不是功能堆叠。

过渡：有了检查标准，下一步是怎么测——三种方法，用途各不相同。

【7.184】用户体验测试方法（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页给三种用户体验测试方法，它们的区别不在技术难度，而在**什么阶段用、回答什么问题**。

第一种，**探索式测试**：可以确定新产品应该包含哪些内容和功能，也可以评估初步设计或原型的有效性和易用性。注意两个关键词——"新产品"和"原型"。它用在**还没有成品的时候**，目的是把"该做什么"和"这个方向对不对"搞清楚。所以它常和低保真原型、纸质草图一起用，成本极低，改起来也快。

第二种，**评估性测试**：在发布前或发布后对最新版本做测试，确保用户能直观使用并获得良好的体验。它用在**有成品的时候**，回答的是"这一版能否上、上了会不会被骂"。发布前做叫可用性测试，发布后做就是拿真实用户的表现来评估。

第三种，**比较性测试**：比较两种或更多产品或设计的易用性，区分各自优缺点，以确定哪种设计能提供最佳的用户操作体验。它用在**有多个候选方案的时候**，回答的是"选 A 还是选 B"。注意它的产出不只是"哪个赢"，还包括**为什么赢、在哪个维度赢**——否则你下次还是不知道该怎么设计。

三种方法连起来是一条时间线：**探索（还不知道做什么）→ 评估（做出来了能否上）→ 比较（有两个方案选哪个）**。选错方法最常见的情形是：已经有了成品，却还在用探索式的方法问"你觉得怎么样"——这种问法拿到的多是客套话，不是可用的结论。

板书：三行 —— **探索式**（阶段：新产品／原型｜回答：该做什么・方向对不对｜手段：草图・低保真）｜**评估性**（阶段：发布前／发布后｜回答：这一版能否上）｜**比较性**（阶段：多方案并存｜回答：选哪个・为什么・哪个维度赢）。底部一条时间线："探索 → 评估 → 比较"。

提问：产品已经上线了，团队还在做"给用户看原型、问你觉得好不好"的测试。问题出在哪？

预设回答：

- 答"没问题，多听听用户总没错"——方向上不错，但方法错了阶段：上线后该问"真实使用中哪里卡住"，而不是"你觉得这个设想好不好"。

- 答"方法用错了阶段，应该做评估性或比较性测试"——正确。上线后的优势正是**有真实行为数据可用**。

- 答"应该直接问用户喜欢哪个颜色"——这属于比较性测试里最浅的一层，只问偏好不问任务完成情况，结论很弱。

- 回扣：方法要匹配阶段，阶段决定你能拿到什么数据。

易错点 / 考点：考点是"三种方法对应的阶段与问题"。常见错答是把"比较性测试"等同于"A/B 测试"——A/B 测试是**在线、真实流量、看行为数据**的比较；比较性测试范围更宽，也可以在实验室里用少量用户做。另一考点：探索式测试强调**早期与低成本**。

过渡：测完得有指标。用户体验怎么量化？下一页给三套模型。

【7.185】用户体验测试模型（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页给三套模型，从"怎么想"到"看什么数"，层次不同。

第一套是**谷歌的 GSM 模型**：先明确**目标（Goals）**，再分析目标会转化成哪些**信号（Signal）**，最后据信号建立具体的**度量指标（Metric）**。举例：目标"让用户快速找到商品"，信号是"搜索到点击的间隔""首次点击命中率"，指标落成"搜索到点击的中位耗时"。价值在于**强制把目标翻译成可测的东西**，中间那层"信号"是关

键——很多人从目标直跳指标，落出来的指标和目标无关。

第二套是**传统网站 UX 指标 PULSE**：Page view（浏览）、Uptime（在线时间）、Latency（延迟）、Seven-day active user（周活）、Earning（收益）。特点是**偏运营和系统层面**，好采集、好汇报，适合做整体看板。短板明显：这些数字讲"网站怎么样"，不讲"用户感受怎么样"——浏览量高可能是设计得好，也可能是用户迷路了乱点。

第三套是 **HEART 模型**，以用户为中心：Happiness（愉悦度）、Engagement（参与度）、Adoption（接受度）、Retention（留存率）、Task success（任务完成度）。

关系记作：**GSM 是方法（从目标推指标），PULSE 是看板（系统与运营），HEART 是用户视角框架**。选指标先问"你要回答系统问题还是用户问题"，答案自然出来。

板书：三栏 —— **GSM**：Goals → Signal → Metric（"强制把目标翻译成可测项"）｜**PULSE**：Page view・Uptime・Latency・Seven-day active user・Earning（系统／运营视角，好采集）｜**HEART**：Happiness・Engagement・Adoption・Retention・Task success（用户视角）。底部："GSM 是方法；PULSE 是看板；HEART 是框架。"

提问：老板要求"用数据证明我们的改版让体验变好了"。你会从 PULSE 里挑指标，还是从 HEART 里挑？为什么？

预设回答：

- 答"用 PULSE，数据现成"——省事，但证明力弱：浏览量涨了未必是体验变好，可能只是投放加了。

- 答"用 HEART，尤其 Task success 和 Retention"——更贴合"体验"这个词。任务完成度和留存率直接反映用户能否办成事、愿不愿意回来。

- 答"两个都用，PULSE 看整体，HEART 看用户"——最完整。前提是先把 GSM 那一层想清楚：这次改版的目标是什么，对应哪个信号。

- 收口：指标不是越多越好，要能对上"你想证明的那句话"。

易错点 / 考点：考点是三套模型的**定位差异与要素名称**。常见错答是把 GSM 当成一组指标——它是**从目标推导指标的方法**，产出的是指标而不是固定清单。另一错答是把 PULSE 当作用户体验模型——它更偏系统与运营。记忆口诀：**GSM 推、PULSE 看、HEART 问用户**。

过渡：HEART 和 GSM 能告诉你"体验有没有变好"，但它们都要跑一段时间才有结论。有没有办法在几天内

> 就判断出"新版和老版哪个更好"？下一节讲 A/B 测试。

【7.186】A/B 测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

各位同学，我们进入本章最后一个专项——**A/B 测试**。先看课件那句提纲挈领的话："有些东西需要 Sense，但大部分东西是可以用 Science 来做判断的。"翻译过来就是：产品上确实有一部分靠直觉、靠品味，但**大部分决定其实可以靠数据来判断**。A/B 测试就是把"靠品味"的那一部分，尽量转成"靠证据"。

定义我先给全：**A/B 测试是把用户分成不同的组，同时在线试验产品的不同版本，通过用户反馈的真实数据来确定哪一个方案更好。**

这里有三层意思。第一层**分组**——用户被分到不同组，不同组看到不同版本。第二层**同时在线**——不是上个月用方案 A、这个月用方案 B，而是同一时间两拨人各看一个，这样季节、活动、竞品这些外部因素对两组的影响是一样的。第三层**真实数据**——判断依据是用户真实行为，不是会议室里的争论。

为什么要专门讲它？因为不讲会出什么事，你们在课程设计里马上就会遇到：按钮放左边还是放右边、标题写"立即注册"还是"免费试用"，产品经理和设计师各有一套理由，最后谁职级高听谁的。上线之后数据不好，谁也说不清是方案错了还是运气不好。A/B 测试解决的就是这个"**无法归因**"的问题。

生活里的类比就是**临床试验**：新药不能"医生觉得有效就上市"，必须设对照组、同期观察、看统计结果。软件比药幸运得多——药做不了对照，软件可以同一秒钟给一半用户旧版本、一半用户新版本。

所以记住一句：**A/B 测试不是"测 bug"的测试，它是"测方案优劣"的测试**。功能测试问"对不对"，A/B 测试问"**哪个更好**"。

> **板书：**A/B 测试 = 分组 + 同时在线 + 真实数据 → 选优；Sense（品味·争论）→ Science（对照·数据）；对照组 A ｜ 实验组 B ｜ 判据：指标差异是否显著。

> **提问：**一个注册按钮，产品经理说改成绿色转化更高，设计师说红色更醒目。下周就要上线，你怎么在两天内给出一个让两个人都服气的答案？

> **预设回答：**

- 答"做用户调研，问用户喜欢哪个"——问出来的偏好不等于真实行为；用户说喜欢绿色，点的时候可能还是点红色那个。要看行为数据。

- 答"先上绿色一周，再上红色一周，比两段数据"——这是最典型的错答。两周的外部环境不同（周末、活动、竞品），差异归因不了。A/B 必须**同时**跑。

- 答"各放 50% 用户，看点击率谁高"——方向对了，但还要补两句：分流要随机且稳定（同一用户每次看到同一个版本），以及要看差异是否**统计显著**，否则 1% 的波动可能只是噪声。

- 点评：第三种回答已经站在正确答案门口。后面讲"特点"补第一层、讲"分流算法"补第二层、讲"评价"补第三层，就是把这个答案补严。

> **易错点 / 考点：**三个必考区分。① **A/B 测试 ≠ 灰度发布**：灰度发布是"逐步放量降低风险"，目的是稳；A/B 测试是"对照组比较优劣"，目的是选。② **A/B 测试 ≠ 埋点分析**：埋点只是采数据，没有"分组对照"就没有因果推断。③ 判断题高频陷阱——"先上线 A 版观察一周，再上线 B 版观察一周，比较两周数据"：**不是 A/B 测试**，违反"并行性"（环境不一致）。口诀：**同时间、两组人、比数据**。

> **过渡：**定义里那三个要素，其实分别对应它的三个特点，我们逐条拆。

【7.187】A/B 测试特点（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> A/B 测试的可信度，靠三个特点撑着：**先验性、并行性、科学性**。这三个词考试要能默写，而且要能各说一句解释。
>
> 第一个，**先验性**。课件写的是"采用流量分割与小流量测试的方式，先让线上部分小流量用户使用以验证设计，再根据数据反馈来推广到全流量，减少产品损失"。说白了就是：**先拿百分之一到百分之几的用户试，试对了再全量**。为什么不直接全量？因为全量是一次不可逆的赌注——方案错了，全体用户都受损，损失被放大。小流量试错的代价被限制在一个可控的池子里。注意"先验"不是"先知先觉"，它指的是**在全面推广之前先取得证据**。
>
> 第二个，**并行性**。课件说"同时运行两个或两个以上版本的试验完成对比分析，而且保证每个版本所处的环境是一致的，避免流程周期长的问题，节省验证时间"。这里两个词最关键：**同时**和**环境一致**。同一时间窗口里，促销、节假日、版本发布、竞品动作对两组的影响是同一份，因此可以被抵消掉。要是串行做，环境一变，你分不清是方案带来的差异还是环境带来的差异。并行还有一个附带好处：快。串行两轮要四周，并行两周出结果。

第三个，**科学性**。课件说"基于统计的数据来做出决策，避免主观或经验的错误决策"。注意"统计"两个字——不是"数据多就科学"。数据多但抽样有偏，照样错。科学性体现在：随机分流、样本量足够、看置信区间、考虑显著性。

打个比方：先验性是"**先小范围试吃**"，并行性是"**两组人同时吃不同的菜**"，科学性是"**用称重和打分表判定，不靠谁嗓门大**"。三者缺一个，结论就软。

再补一句现实感受：很多团队号称在做 A/B，其实只做了第一条——小流量放出去了，但没有对照组、没有显著性检验，最后拿"感觉还行"收尾。那不叫 A/B，那叫**带风险的灰度**。

> **板书：**先验性：小流量 → 验证 → 全量（控损失）｜并行性：同时跑 + 环境一致（可归因、周期短）｜科学性：统计决策（随机・样本量・显著性）；底部一行："缺一条 → 结论不可信"。

> **提问：**我们把新推荐算法先给 5% 用户用，两周后看起来效果不错，就推全量。这里面"先验性"做到了，还有什么没做到？

> **预设回答：**

- 答"没做并行对照，5% 用户是自己跟自己比"——抓住了要害。5% 用户的历史数据和现在比，中间夹着时间因素，归因不了。正确做法是同时留一组用老算法做对照。

- 答"5% 太小，样本不够"——有可能，但要看基线指标和效应量；有的指标 5% 也够，有的 50% 都不够。不能一刀切，要做样本量估算。

- 答"两周太短"——也对，要考虑新奇效应：用户刚看到新东西会多点几下，两周后回落。时长要覆盖至少一个完整的用户行为周期。

- 点评：三个回答分别指向"对照缺失""样本量""时长"，正好是后面"评价"一节的三个要素。

> **易错点 / 考点：**① "先验性"容易误答成"提前预测结果"，正确是"**先小流量取得证据再推广**"。② "并行性"的采分点是"**同时＋环境一致**"，只答"两个版本"不给分。③ 选择题常把"科学性"换成"数据量大就科学"来设错——科学性指的是**统计推断的规范性**（随机化、样本量、显著性），不是数据规模。三步判断法：**有对照吗？同时吗？显著吗？**

> **过渡：**特点讲完，接下来最实际的问题：哪些事值得拿 A/B 去试？

【7.188】ABTest 适应的场景（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

A/B 测试适用的场景，课件列了四类，我按"从技术到业务"的顺序排。

第一类，**灰度发布——用于技术和算法迭代**。新模型、新架构、新依赖库，最先关心的是"会不会崩、性能会不会掉"。这时候 A/B 的价值是**把故障面控制在小流量**，同时对比新旧版本的错误率与耗时。这一类里，A/B 与灰度发布高度重叠。

第二类，**功能优化——界面模块、样式风格、交互方式**。这是最经典的 A/B 战场：按钮位置、配色、导航层级、表单字段多少。判据是点击率、完成率、停留时长。

第三类，**内容优化——推广海报、落地页、内容模块、文案**。同一个活动，标题写"限时五折"还是"仅剩 3 天"，转化差别可能很大。内容类实验的特点：**成本极低、迭代极快**，一天能跑好几组。

第四类，**运营优化——运营策略、沟通话术**。推送时间、优惠券面额、召回话术、客服应答模板。这一类直接对着收入，所以也最容易吵架——因为它涉及"给谁发多少"，光靠经验往往拍不准。

为什么要把场景分类讲清楚？因为**不同场景的判据指标、所需样本量、可承受的实验时长完全不同**。技术类看稳定性和耗时，几天就够；运营类看转化和收入，往往要跑完一个完整周期。

这里我要提醒一句边界：A/B 不是万能的。如果一件事**没法同时给两组不同待遇**（比如价格必须一致、合规不允许差别对待）、或者**流量太小**根本攒不出样本、或者**效果要几个月后才显现**，那就不该硬上 A/B。

> **板书：**灰度发布（技术/算法：稳定性・耗时）→ 功能优化（界面・样式・交互：点击・完成率）→ 内容优化（海报・落地页・文案：转化）→ 运营优化（策略・话术：收入）；旁注"成本↑ 反馈周期↑，判据指标随之变"。

提问：如果公司想验证"把会员年费从 198 降到 168 会不会提升总收入"，能不能用 A/B 测试？

预设回答：

- 答"能，一半用户看 198，一半看 168"——技术上能做，但这是**价格歧视**，多数业务与合规上不允许；用户发现同商品不同价，信任损失远大于收益。

- 答"不能，涉及定价"——方向对，但更准确的说法是：可以测，但要换设计，比如按**地区**或**新老用户**分组，且必须事先评估公平性与合规。

- 答"那就别测了，拍脑袋定"——也不对，还可以用历史数据、价格弹性模型、小范围市场试点。

- 点评：这道题的意义是让大家记住 A/B 的**适用边界**由三件事决定：能否同时差异化对待、流量够不够、指标能不能在合理周期内观察到。

> **易错点 / 考点：**四类场景的名称与对应指标是常考填空。易错：把"灰度发布"只当发布手段，忽略它也是 A/B 的一种应用；把"内容优化"和"运营优化"混答——前者改的是**页面上的东西**，后者是**对用户做的动作**（推送、券、话术）。判断能不能做 A/B 的三问：**能同时分组吗？流量够吗？周期够吗？**

> **过渡：**这一页说的是"有哪些场景"，下一页说"哪些场景真的值得做"。

【7.189】ABTest 适应的场景（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页接着摊场景，但你注意重点变了——上一页回答"有哪些"，这一页回答"**哪些值得做、哪些做不了**"。四类场景之外，我们要加四个判断条件。

第一个条件，**改动成本**。内容优化落在成本最低的一头：改一句标题、换一张海报，开发量近乎为零，样本需求也不大，两三天能出结论。所以成熟的增长团队会在这里做**高频小实验**，一周跑十几组。功能优化往中间走：改交互要设计、要开发、要回归测试，成本上一个台阶，效果通常几天到两周能看出来。灰度发布这一类比较特殊，它的目的不是"选优"而是"控险"，所以它落在"必须做"的区域——新架构上线，你不做小流量验证，就是拿全量用户赌。

第二个条件，**流量是否够攒样本**。一个日活只有两千的 App，做前端按钮实验，一周可能只攒到几百次曝光，这种量级根本看不出 2% 的差异。流量不足时的替代做法是：拉长实验周期、放大改动幅度、或者干脆用定性方法（可用性测试、访谈）。

第三个条件，**反馈周期是否可接受**。运营优化常常落在这一条上：发券、改推送策略，直接动收入，但很多效果要等一个复购周期才看得出来，实验窗口往往要几周。如果效果要半年才显现，A/B 就在时间成本上不划算了。

第四个条件，**能不能同时差异化对待**。有些场景流量够、指标也清楚，但业务上不允许两组不同待遇：价格、会员权益、对外承诺的服务水平，这些不能一半一半。这类事情落在"**不能做 A/B**"的清单里。

所以把上一页和这一页合起来记：**A/B 的适用性不是由"想不想测"决定的，而是由"成本、流量、反馈周期、能否差异化"这四个约束共同决定的。** 四个约束里任一不满足，就该换方法，而不是硬做一个"看起来科学"的实验。

板书：四约束 → ① 改动成本 ② 流量能否攒够样本 ③ 反馈周期 ④ 能否同时差异化；写一行判断："四选一不满足 → 换方法（定性/试点/历史数据）"。

提问：一个内部 OA 系统，全公司三千人，你想验证"新版首页布局能不能减少审批耗时"。这个 A/B 做得成吗？

预设回答：

- 答"做得成，一半人用新版"——先别急。三千人里真正每天走审批的可能只有几百人，样本被进一步稀释；而且审批耗时受流程、审批人习惯影响很大，方差很高。

- 答"做不成，人太少"——方向对。但要给出替代方案：可以先按部门分组做**前后对比**，或者先做小范围试点（一个部门用一个月），再收集定性反馈。

- 答"那就直接全量上，出问题再改"——最危险的一种，内部系统全量切换的返工成本往往比想象的高。

- 点评：这道题的关键不是"能不能分两组"，而是"**每组能不能攒够有效样本、指标方差能不能压住**"。样本不足时，宁可做试点也不要做一个结论不可信的实验。

易错点 / 考点：这一页考"适用性判断"，标准答案要给出**四个约束**，不能只答"流量要大"。易错：认为"只要有数据就能做 A/B"——数据量不等于有效样本量，还要看方差与效应量。另一常见错答是把"不能做 A/B"等同于"不能改进"，实际替代路径有三条：试点、前后对比、定性研究。

过渡：场景讲完，进入这一节的骨架——六个核心概念。

【7.190】核心概念（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是本节的地基，六个概念：场景、桶、层、实验、流量、流量正交。我用一栋楼来类比，你跟着记。

场景：对应业务场景，场景之间**完全独立**。比如"首页推荐""搜索排序""支付收银台"就是三个场景。类比：三栋独立的楼，互不干扰。

桶：圈定了特定的用户流量，**不同的桶之间流量是互斥的**；它从属于场景，一个场景可以有多个桶。类比：

楼里的房间，一个人只能待在一间房里。

层：一类实验的集合，从属于场景，一个场景可以有多个层；**处于同一层的各实验之间流量互斥，处于不同层的各实验之间流量正交，各实验流量之和为总流量**。类比：楼层。同一层的房间互斥，但一个人可以同时在"二楼"和"五楼"各占一间——这就是正交。

实验：用来验证某个决定请求处理方式的功能或策略的一部分流量，通常用来验证某个功能或策略对系统指标（如 PV/UV、CRT、下单转化率等）的影响。类比：房间里摆的那套家具方案。

流量：指所有用户请求。注意是**请求**，不是"用户数"——一个用户一天来十次，就是十份流量。

流量正交：不同层之间流量分配方式**完全独立**，不会互相影响，符合 **MECE**，也就是相互独立、完全穷尽。

为什么要正交？因为两层不独立，实验就会互相污染：你在 UI 层测按钮颜色，同时在算法层测推荐模型，如果两层用同一批人、同一套分配逻辑，那"点击率变了"到底该归功于谁？正交让每个实验都能**独立归因**。

一句话记：**场景装层，层装实验，实验分桶，桶切流量；同层互斥，异层正交。**

> **板书：**场景（业务，完全独立）→ 层（一类实验集合：同层互斥 · 异层正交 · 各实验流量和＝总流量）→ 桶（互斥流量块）→ 实验（一部分流量 + 一个策略 → 看指标 PV/UV · CRT · 转化率）；旁注：流量＝所有用户请求；正交＝独立分配＝MECE。

提问："一个用户在 UI 层进了实验 A，在算法层就不能进实验 B"——这句话对不对？

预设回答：

- 答"对，用户只能进一个实验"——这是最典型的错答。互斥只发生在**同一层内部**；不同层之间是正交的，同一个用户**可以同时**在多个层里各进一个实验。

- 答"不对，不同层可以同时进"——正确，而且要补一句理由：正因为不同层分配独立，一个用户同时参与多个实验也不会让结论互相污染。

- 答"要看是不是同一个场景"——也对，正交与互斥的前提都在**同一个场景内**；跨场景的流量本来就完全独立。

- 点评：这三层其实是一句话——**互斥管同层，正交管异层，独立管跨场景**。

> **易错点 / 考点：**六个概念必考，最易错的是**桶与层的混淆**（桶是流量块，层是实验集合）和**互斥/正交的对象搞反**。口诀：**同层互斥、异层正交、场景独立**。判断题陷阱："各层流量之和等于总流量"——课件原话是"各实验流量之和为总流量"（指同一层内），把"各实验"偷换成"各层"就是错。

> **过渡：**光有名词不够，下一张图把"层怎么切、桶怎么分"画出来。

【7.191】A/B 测试：桶、层、流量（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是上一页那个"楼宇模型"的具体切法，你要看出三件事：**总流量怎么往下切、同一层的实验怎么互斥、不同层怎么叠加。**

先说切法。总流量是百分之百，先按场景隔开；场景内部再切层。这里有个最反直觉的点：**层不是把流量切走一块，而是同一批流量在每一层各被重新分配一次。** 所以 UI 层可以是 A 组一半、B 组一半；算法层也可以是 C 组一半、D 组一半——两层的"一半"**不是同一批人**。理解了这句话，正交就懂了。

再说互斥。同一层里，如果你要同时跑两个实验，比如"按钮颜色"和"按钮文案"，你不能让两个实验各占一半再各占一半——那加起来就是百分之二百。正确做法是**把这一层的流量切成互斥的桶**：实验一拿 1 到 50，实验二拿 51 到 100，每个桶内部再和对照组对半分。

最后说叠加。一个用户请求进来，网关依次在每一层里算出他属于哪个桶，把各层的结果拼成一个完整的实验组合。这就是"正交"在工程上的样子——**每层独立算一次哈希，互不影响**。

看图时请盯住那个"流量之和"的约束：同一层内所有实验（含对照）的流量加起来必须等于该层总流量。多了说明有重叠，少了说明有流量没被分配到，两种情况都会让统计结论失真。

给个数字感受一下。假设某场景一天一百万请求，UI 层开两个实验各占两成、其余六成留空；算法层开一个实验占三成、七成留空。这不是浪费——**留空是给未参与实验的多数用户保留稳定体验**，也是给后续实验留出流量预算。

> **板书：**总流量 100% → 场景 → 层（**每层都用全部流量重新分配**）→ 桶（同层互斥切块）→ 实验 A/B；约束：同层 Σ 实验流量（含对照）= 该层总流量；正交实现：每层独立哈希。

> **提问：**同一层里已经有一个占 60% 流量的实验，你还能不能再加一个占 50% 的实验进这一层？

> **预设回答：**

- 答"能，各分一半用户"——错。同一层是互斥的，60 + 50 = 110 已经超了；而且"各分一半"和"占 60%"是两种互相矛盾的分法。

- 答"不能，最多再给 40%"——正确。这一层剩下的 40% 可以给新实验，也可以留一部分作对照或留空。

- 答"那就把它挪到另一层去"——这是很专业的回答。换层就变成正交关系，不再抢同一层的流量；但要注意换层后归因口径也变了。

- 点评：三个回答分别踩中"互斥约束""流量预算""用正交解决容量问题"，最后一条是工程上最常用的解法。

> **易错点 / 考点：**考图必考"同层之和等于总流量"这条约束。常见错答：把"层"当成"流量的一大块"（错，层是全量再分配）；把"留空"当成"浪费"（留空是体验保护与流量预算）。若出计算题：同层已有 30%＋25% 的实验，问还能给多少——答 45%，且建议其中保留对照。

> **过渡：**模型讲清了，接下来看工业界怎么把它落成一套网关架构。

【7.192】基于 OpenResty 的多层分流模型（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲一个工业级的实现方案：**基于 OpenResty 的多层分流模型**。先把 OpenResty 是什么说清——课件给的定义是"**OpenResty 是一个基于 Nginx 与 Lua 的高性能 Web 平台，用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关**"，课件还给了官网地址 openresty.org/cn。一句人话：**把 Nginx 的并发能力和 Lua 的脚本能力缝在一起**，让网关能跑业务逻辑。

再看这条链路，我按请求的走向讲。链路的起点是**终端**：App、H5、小程序三种形态。它们带着**终端唯一标识**和**请求 URI** 进来，走 Cookie 协议，同时可以带一个 **ABTest header** 或 **Mfw_project** 标识。

流量接着到**网关**——课件称它为"跨平台、跨语言的 ABTest 网关"。网关在这里做三件事：第一，用 **MurmurHash** 算分流；第二，从 **Redis / 实验策略**里取当前生效的流量配置，比如 1 到 50 给版本 A、51 到 100 给版本 B；第三，做 **URI 匹配**，把请求路由到"具体微服务"。

链路终点是**后端具体服务程序**，也就是真正的微服务。它对 A/B 实验基本是**无感**的——它看到的就是一个已经打好标的请求。

为什么要把分流放在网关，而不是放在每个服务里？因为终端是**跨平台**的（App、H5、小程序），后端是**跨语言**的（Java、Go、Node）。每个服务自己实现一套分流，会出现三件事：算法不一致、口径不一致、改一次要发很多次版。放到网关，就是**一份分流逻辑、一处配置、全局一致**。

还有两个细节值得注意。"1 到 50、51 到 100"这种区间写法，说明分流是**按哈希值取模后落区间**，而不是"取随机数"——随机数每次请求都变，同一个用户一会儿看到 A、一会儿看到 B，实验就废了；哈希保证**同一标识永远落在同一区间**。第二个细节是**策略放 Redis**：配置放在外面，改流量比例不用重启网关，这才支撑得起"一天跑十几组实验"。

板书：终端（App/H5/小程序）＋ 唯一标识 ＋ 请求 URI ＋ Cookie → OpenResty 网关（跨平台・跨语言）：MurmurHash 分流 ｜ Redis 取策略（1～50→A，51～100→B） ｜ URI 匹配 → 具体微服务（对实验无感）；旁注：OpenResty ＝ Nginx ＋ Lua。

提问：为什么分流要用"哈希取模落区间"的固定算法，而不能用"随机数"？

预设回答：

- 答"随机数更公平"——概率上公平，但**一致性**没了：同一用户刷新两次可能看到两个版本，体验割裂，数据也脏。

- 答"哈希能保证同一用户稳定"——正确，这是核心，哈希的输入固定，输出就固定。

- 答"那用哈希会不会某一桶特别大"——会，所以选哈希函数有要求：分布要均匀，下一页讲 MurmurHash 就是这个原因。

- 点评：这道题的答案就是三个词——**稳定、均匀、可复算**。可复算很重要：出了问题，拿用户标识重算一遍就知道他在哪个桶，这叫"分流可还原"。

易错点 / 考点：必记两点：**OpenResty ＝ Nginx ＋ Lua**；分流的关键输入是**设备标识 ＋ 实验/策略标识 ＋ 层标识**（下一页展开）。常见错答："A/B 测试要在每个业务服务里改代码"——正好相反，主流做法是**网关统一分流、业务无侵入**。选择题还可能考"为什么策略放 Redis"，答案：动态下发、免重启、多实例一致。

过渡：这个模型到底解决了什么问题？下一页逐条列它的优势。

【7.193】OpenResty 分流模型的优势（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把上面的架构摊成"好处"，一共五点，我按对团队的影响从大到小排。

第一，**接入成本低，业务无侵入**。分流在网关做，业务代码几乎不用改。这一点对老系统尤其重要——一个

跑了五年的系统，你让它改造一遍分流 SDK，风险和工期都受不了；网关层加一段 Lua，改一次全局生效。

第二，**跨平台、跨语言统一**。App、H5、小程序，Java、Go、Node，全都走同一套分流算法。如果各端各写一套，最常见的事故就是"App 算出来是 A，H5 算出来是 B"，同一个用户在两端看到不同版本，数据互相污染。统一在网关，这个坑从根上就不存在。

第三，**性能开销小**。Nginx 加 Lua 本身是为高并发设计的，一次哈希运算的成本远小于一次业务请求。分流不该成为瓶颈，这是架构上的硬要求。

第四，**策略动态化**。实验配置放 Redis，运营改流量比例、开关实验、加白名单，都不需要发版重启。"快"是 A/B 的生产力——一天只能改一次配置，就别谈什么高频实验。

第五，**不产生二次流量**。分流在入口处一次决定，请求只走一条路径。如果靠前端发两次请求再比结果，流量翻倍、后端压力翻倍，而且两次请求的条件并不一致，比出来的结果也不可信。

同时要看到代价，做工程不能只讲好处：**网关成了关键路径**，它挂了全站都受影响，所以必须多副本、健康检查、配置降级——Redis 不可用时要能按默认策略放行，而不是拒绝服务。另外，Lua 代码的可测性、可观测性都比普通业务代码差，实验逻辑必须做单元测试和灰度验证。

一句话：**这个方案的强项是"统一"和"轻量"，代价是"网关要足够可靠"。**

> **板书：** ① 业务无侵入（接入成本低）② 跨平台跨语言统一口径 ③ 性能开销小 ④ 策略动态化（Redis，免重启）⑤ 不产生二次流量；代价：网关是关键路径 → 多副本 · 健康检查 · 配置降级。

提问：如果网关挂了，A/B 实验会怎样？你认为这时系统应该"拒绝服务"还是"按默认版本放行"？

预设回答：

- 答"拒绝服务，至少不会出错"——这是最危险的错答。实验是增强功能，不是核心功能；为了实验正确性把整个站点打挂，本末倒置。

- 答"按默认版本放行"——正确。这叫**降级**：配置取不到就按线上主版本走，用户无感，实验数据这一小段缺失可接受。

- 答"应该提前把配置缓存在本地"——非常专业，这是多级缓存思想：本地兜底，Redis 常态，网关聚合。

- 点评：这道题把"实验系统"和"核心链路"的优先级关系讲透了——**实验系统必须能优雅地不存在**。

易错点 / 考点：五点优势要能列全，其中"**无侵入**""**跨平台跨语言**""**不产生二次流量**"最常被考到。易错：

把"无侵入"理解成"不用改任何代码"——SDK 或埋点仍要接，只是业务逻辑不用改。另一考点是"配置服务挂了怎么办"，标准答案必含**降级＋本地兜底**。

过渡：优势里反复出现"均匀""稳定"，这靠的是算法，下一页专门讲它。

【7.194】MurmurHash 算法（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲分流的算法核心：**MurmurHash**。

先看输入。课件说"参与算法的 Hash 因子有**设备 id、策略 id、流量层 id**"。三个因子各有分工：**设备 id** 决定"你是谁"，保证同一个用户稳定落桶；**策略（实验）id** 决定"在测哪件事"，让不同实验的分配互不相关；**流量层 id** 决定"在哪一层"，让不同层正交。三个拼起来做哈希，就同时拿到了"稳定"和"正交"两个性质。记住这句：**盐不同，分布就不同；盐固定，结果就稳定。**

再看为什么选它。课件给了两个特点。

第一个，**快，比安全散列算法快几十倍**。MD5、SHA 这类算法是为防篡改设计的，要抗碰撞、要不可逆，代价就是慢。而分流场景根本不需要密码学强度——没人会为了看 B 版本去攻击你的哈希。分流在请求链路上，每一毫秒都要算，所以必须快。

第二个，**变化足够激烈**。课件举的例子是"abc"和"abd"这两个只差一个字符的串，能**均匀散布在哈希环上**。这条比"快"更关键：如果哈希对相近输入给出相近输出，那设备 id 连号的一批用户就会全落进同一个桶，桶大小失衡，实验结论直接失真。MurmurHash 的雪崩效应能让一个比特的变化把输出彻底打散。

最后是两个关系的定义，务必分清。**互斥**：两个实验流量独立，用户只能进入其中一个——这是"同一层"的关系。**正交**：用户进入所有的实验之间没有必然关系——这是"不同层"的关系。前者靠"同层共用一份哈希结果的不同区间"实现，后者靠"层 id 进哈希因子"实现。

一句话收口：**MurmurHash 用三个因子、几十倍的速度、足够激烈的分布，把"稳定、均匀、正交"三件事一次算完。**

板书：输入：设备 id ＋ 策略（实验）id ＋ 流量层 id → MurmurHash → 取模 100 → 落 1～50 / 51～100；理由：① 快（比安全散列快几十倍）② 分布激烈（"abc"/"abd" 均匀散布哈希环）；互斥＝同层・只能进

一个；正交＝异层・无必然关系。

提问：为什么不用 MD5 或 SHA 来算分流哈希？反正都是哈希。

预设回答：

- 答"MD5 更安全"——分流不需要安全性。哈希结果本来就发给前端、写进 header，谈不上保密。

- 答"MD5 太慢，扛不住高并发"——正确，这是主因。安全散列为了抗碰撞牺牲了速度，在每请求都要算的链路上是纯浪费。

- 答"那用 CRC32 行不行"——可以讨论：CRC 快，但分布质量不一定够均匀，尤其对连号的设备 id 容易聚簇。选算法要看**速度与分布**两个指标，不是只看一个。

- 点评：这道题的标准答法是"**分流要的是快和均匀，不是抗碰撞**"，一句话点明取舍。

易错点 / 考点：三个哈希因子必须默写全（**设备 id、策略 id、流量层 id**），少一个就丢了采分点。易错：把"互斥"和"正交"的对象弄反；把 MurmurHash 当成"加密算法"（它是散列，不提供安全性）。考点还可能落到"为什么要均匀"——因为桶大小失衡会导致两组基线不可比。

过渡：机制齐了，接下来看人怎么按流程把这套机制跑起来。

【7.195】A/B 测试流程（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

前面讲的是机制，从这一页开始讲"人怎么干"。A/B 测试的流程五步：**产生实验想法 → 评估实验优先级 → 设计和开发实验 → 分析数据 → 应用结果**。我一步步说清每步的产出与常见的错。

第一步，**产生实验想法**。想法从哪来？三个来源：数据分析（哪里流失高、哪里点击低）、用户反馈（客服工单、调研）、竞品与经验。这一步最常见的病是**拍脑袋生成想法**——产品经理昨晚想到一个点子就当实验做，跑两周才发现根本没人注意这个改动。判断一个想法值不值得做，先问一句："它预期影响哪个指标、影响多大、凭什么这么认为。"

第二步，**评估实验优先级**。想法永远比资源多，必须排序。常用的排法是看三件事：**预期影响有多大、我们对这个预期有多确信、做它要多少成本**。影响大、把握足、成本低的先做。这一步的价值是防止团队把两周花在一个 0.1% 的改动上。

第三步，**设计和开发实验**。要定下来的是：假设（改什么、期望什么变化）、**主指标**（唯一判定依据）与护栏指标（不能变坏的指标）、分流比例、样本量与实验时长、埋点方案。这一步最容易出的错是**指标定太多**——指标一多，总能挑出一个"显著"的，那就成了自己骗自己。

第四步，**分析数据**。看三件事：差异有多大、**是否统计显著**、有没有分群差异——比如新用户有效、老用户无效。还要检查**样本比例失调**：本该对半分，实际 47 比 53，说明分流或埋点有问题。

第五步，**应用结果**：全量上线、下线，或改设计再测。课件用的是"应用结果"，重点在"**应用**"——很多团队报告写完，代码里的实验分支留了三年没人清，技术债就是这么来的。同时要把结论**沉淀成知识**：这个人群、这个场景下，什么改动有效。

口诀：**想法要能落到指标，优先看影响乘把握，设计只留一个主指标，分析先看显著再看分群，应用要清代码留结论。**

板书：想法 → 优先级（影响 × 把握 ÷ 成本）→ 设计开发（假设・主指标・护栏・样本量・时长・埋点）→ 分析（显著？分群？比例失调？）→ 应用（上线/下线/再测 + 清代码 + 沉淀结论）。

提问：为什么第三步一定要"只留一个主指标"？多几个指标不是看得更全吗？

预设回答：

- 答"指标多了更全面"——这是最典型的错答。指标一多，事后总能挑出一个"显著"的，等于把判定标准交给了运气。

- 答"因为多重检验会抬高假阳性"——正确且专业：测二十个指标，按 5% 显著性水平，纯随机也会有一个"显著"。

- 答"主指标定一个，其他的当护栏"——最佳答案。护栏指标不是用来比优劣的，而是用来一票否决的：点击率涨了但退款率也涨了，就不能上。

- 点评：三答的差别就是"裁判是一个还是二十个"。**一个裁判，一套事先定好的规则**，这是实验和讲故事的分界线。

易错点 / 考点：五步顺序是必考填空。易错：把"评估优先级"和"设计实验"顺序颠倒；把"分析数据"答成"看报表"（分析含显著性与分群）。考点还可能问"实验结束后还要做什么"——标准答案：**全量或下线、清理实验代码、归档结论与元数据**，很多人漏掉清理代码这一条。

过渡：流程讲完，下一页看数据在这五步之间怎么流动。

【7.196】A/B 测试 数据流（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把 A/B 测试的数据链路摊开，你要看出的是**两个闭环**，而且这两个闭环必须用**同一个标识**对齐。

先看第一个闭环，**分流闭环**：实验配置存在配置中心，SDK 从平台拉取配置；用户请求进来时算出桶号，命中哪个版本就返回哪个版本。这个闭环的特点是**快**——它必须在一个请求的生命周期内完成，通常几十毫秒。

第二个闭环，**数据闭环**：用户看到版本之后的行为——曝光、点击、下单——由前端埋点和后端日志记录下来，经采集管道进数仓，再算出指标，最后进报表。这个闭环的特点是**慢**，从产生行为到报表可见，可能是分钟级，也可能隔天。

两个闭环怎么对上？靠同一个**分流标识**：SDK 在分流时给这次请求打上 `abTraceId`（请求唯一标识）和 `bcm`（结果标识），埋点上报时把这个标识一起带上。数据侧一看到标识，就知道"这条行为属于哪个实验、哪个版本、哪个桶"。

如果不打标会怎样？这就是最常见的"**实验结论不可信**"的根因：你有一堆点击日志，也知道两个版本各自的转化率，但分不清哪条日志来自哪一组，只能按时间或用户 id 去猜归属，一猜就错。

这张图还要注意三个工程细节。第一，**配置要有版本和下发时间**，否则"实验中途改了流量比例"，前后两段数据混在一起就没法解释。第二，**分流要能复算**，出问题时拿用户标识重算一遍，验证他在哪个桶——这叫回放能力，是排查线上争议的杀手锏。第三，**上报要能抗丢**，日志丢了会让两组的样本量不对等，而样本量不对等会直接破坏统计结论。

一句话总结：**分流决定用户看到什么，埋点决定我们能看到什么，标识决定这两件事能不能对上。**

> **板书：**分流闭环：配置中心/Redis → SDK 拉配置 → 算桶 → 返回版本（快，毫秒级）；数据闭环：埋点/日志 → 采集 → 数仓 → 指标 → 报表（慢，分钟到天）；对齐钥匙：`abTraceId` + `bcm`；三要求：配置带版本 · 分流可复算 · 上报抗丢。

提问：报表里两组的样本量差得很多，本该接近对半分，你觉得问题最可能出在链路的哪一段？

预设回答：

- 答"分流算法不够均匀"——可能，但要先排除更常见的两类原因：埋点丢失和上报过滤。

- 答"埋点丢了一批，或者某个版本加载失败没上报"——很可能。版本 A 页面加载失败率高，用户压根没看到曝光，日志自然少，样本就偏了。

- 答"分流本身比例就配的不是 50/50"——也很常见，尤其实验中途调过流量比例，前后两段混算就出现比例失衡。

- 点评：排查顺序应该是"**先查配置比例，再查埋点丢失，最后才怀疑算法**"。因为前者最常见，后者最少见；反过来查会浪费时间。

> **易错点 / 考点：**两个闭环＋一个标识是核心考点。易错：认为"分流做对了数据就一定对"——分流正确只保证分组正确，样本量、去重、流失都归数据侧管。常见问法："实验数据不可信，可能有哪些原因"，标准答案要覆盖：**分流不均匀、埋点缺失、口径不一致、配置中途变更、样本量不足**。

> **过渡：**这条链路的两端，一端是平台，一端是代码里的 SDK，我们先看 SDK 做什么。

【7.197】A/B 测试 SDK（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲 SDK 干的活——它是 A/B 平台伸进业务系统的"手"，一共三件事。

第一件，**实验配置动态分发**。课件原话是"实现针对场景标识＋用户分流标识的**一致性哈希算法**，根据场景的实验流量配置，进行实验配置的动态分发"。这里两个词要抓住：**一致性哈希**和**动态**。一致性哈希的意思是，当你调整流量比例、下线一个实验、新增一个实验时，**大部分用户的归属不变**，只有落在被调整区间的那部分会迁移。这一条对业务太重要了——你不可能每次改配置就把一半用户的体验重置一遍。动态是指配置不写死在代码里，SDK 定期拉取或接收推送，改配置不用发版。

第二件，**上报 ABTest 数据**。课件列了三类日志：**请求日志**（每次分流的结果，用于校验流量比例）、**业务自定义日志**（转化、下单金额等业务指标）、**监控日志**（SDK 自身的健康度：请求数、失败数、延迟）。三类各有用途：请求日志验"分流对不对"，业务日志算"效果好不好"，监控日志查"SDK 活得好不好"。很多团队只上报业务日志，结果发现数据对不上时无从排查——缺的就是请求日志和监控日志。

第三件，**基于埋点规范生成前端埋点标识**。课件给了三个具体标识：**abTraceId**（请求唯一标识）、**bcm**（请求结果标识），以及**事件标识**（曝光 view、点击、收藏、转发等）。这些标识的作用是**透传到前端埋点，用来追踪用户行为**。为什么要 abTraceId？因为一次请求可能带来多个行为事件——曝光一次、点击三次，

有了请求唯一标识，才能把"这一次体验"和"这一串行为"绑在一起，避免重复计数和漏计。

一句话记：**SDK 三件事——拉配置（一致性哈希、动态）、报数据（请求/业务/监控三类）、打标识（abTraceId、bcm、事件）。**

板书：① 配置动态分发（场景标识 + 用户分流标识 → 一致性哈希 → 少迁移）② 上报三类日志（请求 / 业务自定义 / 监控）③ 生成埋点标识（abTraceId 请求唯一 · bcm 结果 · 事件：曝光/点击/收藏/转发）。

提问："一致性哈希"里的"一致性"到底保证了什么？如果换成普通取模，改一次流量比例会发生什么？

预设回答：

- 答"保证同一用户看到同一版本"——这只说对了一半。普通取模也能保证同一用户稳定，除非分母变了。

- 答"保证调整流量时大部分用户不迁移"——正确。普通取模的分母一变（比如从 100 改成 120），几乎所有用户的余数都变，等于全员重新洗牌，实验数据的连续性就断了。

- 答"一致性和稳定性是一回事"——不是。**稳定性**指"同一输入的输出不变"，**一致性**指"输入"或"名额"小幅变化时，已有映射尽量保持。这正是它比取模高明的地方。

- 点评：这道题的落点是"**扩容/调比例不洗牌**"。面试里问一致性哈希，八成就在问这个。

易错点 / 考点：SDK 三件事＋三类日志＋三类标识（abTraceId、bcm、事件）都是采分点。易错：把 abTraceId 和 bcm 混为一谈——前者标识"这一次请求"，后者标识"这一次请求的结果"；把"监控日志"漏掉。如果考简答"SDK 为什么必须上报请求日志"，答案是：**用于校验分流比例是否正确、支持复算与排查**。

过渡：SDK 在业务侧干活，另一头是平台本身，我们看平台该有哪些功能。

【7.198】A/B 测试平台功能（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲 A/B 平台作为一个产品，要有哪四块功能。这四块其实就是上一页 SDK 的对立面——SDK 在业务侧，平台在服务侧。

第一块，**ABTest 元数据管理**：应用、场景与实验的创建、编辑，以及配置的发布。所谓"元数据"，就是"我们在测什么、怎么分、分多少"这些描述信息。这一块看着简单，实际最考设计：应用和场景要能分级管理，实验要有状态机——草稿、运行中、已暂停、已结束、已归档；配置发布要有版本和回滚。没有状态机和版

本，线上就会出现"以为下线了其实还在跑"的事故。

第二块，**ABTest 接入的开发和测试支持**。课件说得很具体：用户可以方便地查看**完整可用的接入示例代码**，可以**输入分流用户标识进行测试**，以及**查询实时日志**。这三件事对应三个真实痛点：示例代码决定接入要几天还是几小时；输入标识测试让开发能自己验证"这个用户到底进哪个桶"，不必每次找平台同学；实时日志决定了排查一次线上问题要五分钟还是五小时。

第三块，**实时报表和离线效果报表**。实时报表包含实时请求、点击（率）、转化率、千次曝光转化等指标，以及**对比分析**；离线报表做更完整的统计与分群。为什么要两套？实时报表用于**早发现问题**，比如分流比例不对、某个版本报错暴涨；离线报表用于**下结论**，比如显著性、置信区间、长期效果。用实时报表下结论，是新手最常犯的错。

第四块，**异常监控和告警**。平台实时监控 SDK 上报的**请求数、失败数和延迟时间**等数据，一旦发现异常即发出告警。这一块最容易被忽略，但它是平台的"刹车"：分流服务挂了，用户看到的是错误页，不是"少了一个实验"。

一句话：**元数据管"测什么"，接入支持管"接得快"，报表管"看得清"，告警管"出事能知道"。**

> **板书：** ① 元数据管理（应用/场景/实验 + 配置发布 + 状态机/版本）② 接入支持（示例代码 · 输入分流标识测试 · 实时日志）③ 实时报表 + 离线效果报表（请求 · 点击率 · 转化率 · 千次曝光转化）④ 异常监控告警（请求数 · 失败数 · 延迟）；旁注："实时看异常，离线下结论"。

提问： 为什么平台要同时提供"实时报表"和"离线报表"？只保留一个不行吗？

预设回答：

- 答"实时就够，看得快"——最典型的错答。实时数据往往只是初步结果，样本不足、未经清洗，用它下结论就是拿噪声当结论。

- 答"离线更准，只要离线"——也不行。离线报表有延迟，分流比例配错、某版本报错暴涨这类问题要等到第二天才发现，损失已经发生。

- 答"实时看异常，离线下结论"——标准答案。两者分工不同：一个是**监控**，一个是**统计推断**。

- 点评：这也是所有数据系统的通例——**流式负责发现问题，批式负责回答问题**。

易错点 / 考点： 四块功能必须列全，且要能各说一句用途。易错：把"实时报表"当成"效果判定依据"；漏掉"配置发布/回滚"这一条（元数据管理不只是增删改）。若考"平台为什么需要状态机"，答：**防止运行中实**

验被随意修改，保证数据可比与可追溯。

> **过渡：**功能齐了，最关键的问题来了——实验结论怎么评价才站得住。

【7.199】A/B 测试评价（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲怎么评价一个 A/B 实验的结论，也就是"**凭什么说 B 比 A 好**"。这是整节最容易出错的地方，也是最容易被拿来做文章的地方。

第一件事，**看差异是否统计显著**。两组转化率 3.1% 和 3.3%，差 0.2 个百分点——这个差是真的，还是随机波动？答案要靠显著性检验和置信区间。要点是：**先定样本量，再看结果**，而不是看到结果再算"还差多少样本"。

第二件事，**看样本量够不够、时长够不够**。样本太小，任何差异都可能是噪声；时长太短，会撞上**新奇效应**（用户对新鲜事物短期更活跃）和周期效应（周末与工作日行为不同）。经验做法是至少覆盖一个完整的用户行为周期。

第三件事，**看指标选得对不对**。主指标只能有一个，作为判定依据；另外配**护栏指标**，比如不能因为提升点击而抬高退款率。指标一多，多重比较的问题就来了：测二十个指标，纯靠运气也会有一两个"显著"。

第四件事，**看效应在人群上是否一致**。整体不显著，不代表对某类用户不显著；反过来，整体显著，也可能是某一类用户被另一类稀释了。这就是后面决策树要解决的问题。

第五件事，**看结论能不能落地**。一个"提升 0.3%、需要多花两倍服务器"的方案，未必值得上。

这五件事里，最危险的不是算错，而是**算对了却选择性地说**。所以评价的核心不是"会不会算"，而是"敢不敢把全部结果摆上桌"。这一点我们单独讲。

【思政落点 12】：A/B 测试的说服力来自"数据不撒谎"，可数据不会自己说话，是人在解释它。同一份报告，只报显著的指标、把 2 个百分点的差异说成"大幅增长"、或者跑到一半提前偷看结果再决定停不停——这些做法在技术上都不算伪造数据，却足以让错误决策披上科学的外衣。请守住一条底线：**报结论时，把口径、样本量、置信区间和不利指标一起报**。会做实验是能力，如实报告是品行。

> **板书：**评价五看：① 显著？② 样本/时长够？③ 指标对（1 主 + 护栏）？④ 分群一致？⑤ 可落地？；红线

> 一行：**口径 · 样本量 · 置信区间 · 不利指标 —— 一起报**。

> **提问：**实验跑了一周，B 组转化率比 A 组高 0.4 个百分点，p 值 0.06。产品经理说"就差一点，算通过吧"。你怎么回答？

> **预设回答：**

- 答"0.06 很接近 0.05，可以算显著"——典型的"凑显著性"。阈值必须事先定，事后放宽等于把结论变成主观意愿。

- 答"不能算，p=0.06 说明还有 6% 是噪声"——表述不严谨但方向对；更准确的说法是这个结果**不足以拒绝原假设**。

- 答"再跑一周看会不会更显著"——可以做，但要事先声明这是"延长实验"，并且接受"可能更不显著"；不能一边延长一边反复偷看，那会放大假阳性。

- 点评：三答的差别就在"规则是事先定的还是事后改的"。**先定规则再跑是科学，跑完再定规则是讲故事。**

> **易错点 / 考点：**高频错点：① 把"相对提升"和"绝对百分点"混着说——3% 到 3.3% 是 0.3 个百分点，也是 10% 的相对提升；② 认为"两组数据一算就知道谁好"，忽略显著性与样本量；③ 认为"指标越多结论越可靠"，实际多重比较会抬高假阳性。若考简答"如何评价一个 A/B 实验"，按五看作答即可拿满分。

> **过渡：**知道了怎么评价，再看成熟团队在工程上到底做对了哪些事。

【7.200】A/B 测试优秀实践（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是行业里被反复验证过的做法，我一条条过，重点是每条**解决了什么问题**。
>
> 第一条，**采用流量拦截分发的方式，摒弃了原有接口的形式，对业务代码没有侵入，性能没有明显影响，且不会产生二次流量**。对比"接口形式"：业务代码里调一次取配置的接口，拿到版本再分支；侵入性强、加一个实验就要改一次业务代码，还多一次远程调用。流量拦截反过来——请求进网关就打标，业务只管读标。"不产生二次流量"值得单独说：接口形式会引发额外调用，量大时就是实打实的成本。
>
> 第二条，**采用流量分层并绑定实验的策略，更精细直观地去定义分流实验**。分层之后，每个实验明确声明"我在哪一层、占多少流量"，避免实验之间互相抢流量。

第三条，**数据传输：通过在 HTTP 头部增加分流信息，业务方无需关心具体的实现语言**。这是解耦的关键——Java 读 header，Node 也读 header，语言无关。

第四条，**监控体系**。实验本身要能被监控：分流是否生效、比例是否准、SDK 是否异常。没有监控，实验就是黑盒。

第五条，**用户画像等精细化定制 AB**。不是简单对半分，而是按人群定向：新用户、老用户、高价值用户分别做实验。这解决了"整体看不出效果、某类用户却有效"的问题。

第六条，**统计功效支持置信区间、特征值等**。这是把"科学"落到工具上——平台自己要能算置信区间，而不是让分析师手算。

第七条，**通过 AARRR 模型评估实验对北极星指标的影响**。课件解释了 AARRR：**Acquisition 获取、Activation 激活、Retention 存留、Revenue 收益、Referral 推荐**；并指出"AARRR 模型把控产品整体的成本/收入关系，用户生命周期价值 LTV 远大于用户获取成本 CAC 与用户经营成本 COC 之和，就意味着产品的成功"。判据就是：**LTV 大于 CAC 加 COC**。实验把注册率提上去了，但拉来的是低质量用户、LTV 掉下来，那是"指标赢了、生意输了"——这正是需要北极星指标和护栏指标的原因。

> **板书：** ① 流量拦截分发：无侵入 · 无二次流量 ② 分层绑定实验 ③ HTTP header 传分流信息（语言无关） ④ 监控体系 ⑤ 用户画像精细定制 ⑥ 统计功效：置信区间 · 特征值 ⑦ AARRR → 北极星指标；判据：**LTV ＞ CAC ＋ COC**。

> **提问：** 一个实验把新用户注册率提升了 8%，但三个月后这批用户的付费率明显低于老渠道。这个实验算成功吗？

> **预设回答：**

- 答"算成功，注册率是明确的指标提升"——只看第一层指标，这是最典型的错答。注册是手段，不是目的。

- 答"不算，付费率掉了"——方向对。但要注意是"付费率"还是"付费总额"掉了，以及这批用户本来就该是低付费人群（渠道结构变化）。

- 答"要看 LTV 和 CAC 的关系"——最专业。如果获客成本没降、LTV 却降了，那就是"用数量换质量"，长期是亏的。

- 点评：这道题的落点就是课件的 AARRR：**任何单点指标的提升，都要放回 AARRR 漏斗和 LTV/CAC 的账里去核算**。

> **易错点 / 考点：**七条实践要能列全，其中"**无侵入＋无二次流量**""**HTTP header 传分流信息**""**AARRR**"最常考。易错：把 AARRR 的五个字母记错（Acquisition、Activation、Retention、Revenue、Referral），把 R 混成 Retention 与 Referral；把 LTV/CAC/COC 三者的关系说反。答题模板：**先看北极星，再算 LTV 与 CAC 加 COC 的账**。

> **过渡：**还有一组实践，落点在"平台自己好不好用"，我们接着看。

【7.201】A/B 测试优秀实践-续（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是"续"，讲**平台自己怎么做好用、怎么扛得住**，也就是工程侧的实践。课件最后那句提醒放到最后说。

第一组，**提升平台易用性**：支持完备的 ABTest 元数据管理、**应用级的角色和权限**、实验的**商家 id 或者用户分流标识的白名单**。三件事各对应一个真实需求：元数据完备才可追溯；角色权限才不至于"谁都能改线上流量"；白名单让开发和测试先把指定账号拉进组里，不必等真实流量。

第二组，**系统优化**：多级缓存、优化 SDK 性能、实现 SDK 配置自动化和透明化、优化一致性哈希算法。多级缓存解决"每个请求都查一次配置"的压力——SDK 本地一份、网关一份、Redis 一份；SDK 性能影响业务耗时，分流必须廉价；配置自动化透明化，是指接入方不用手工配一堆参数，变更也能自动生效；优化一致性哈希，是为了调流量时**尽量少迁移用户**。

第三组，**支持分别针对 Java 和 Node 端的示例代码**，包括引入代码库、初始化、获取实验配置以及前端埋点。接入成本的一大半花在"照着文档试错"上，有能直接跑的示例，接入能从两天变两小时。

第四组，**支持测试、预发、生产等不同的应用环境**。实验配置必须分环境隔离，否则测试环境改的流量比例会打到生产上——那是全量事故。

第五组，**支持输入分流用户标识获取实验配置，以测试和还原 A/B 分流结果**。这就是"可复算"：拿一个用户标识，平台告诉你他进哪个实验，争议一查就清。

第六组，**支持实时日志明细查询**，查询条件包括日志类型、实验以及分流用户标识。缺了它，"数据不对"就成了无解的悬案。

最后，课件点了要害：**需要关注 ABTest 接入成本与易用性、ABTest 数据价值**。翻成大白话：**平台再漂亮，**

> **接入要两周、报表没人看，它就没有价值**。衡量平台成败的，不是"支持多少功能"，而是"多少实验真跑起来了、多少结论真被采纳了"。

> **板书：**易用性：元数据 · 应用级角色权限 · 白名单（商家 id/分流标识）；系统优化：多级缓存 · SDK 性能 · 配置自动化透明化 · 一致性哈希优化；工程化：Java/Node 示例 · 测试/预发/生产隔离 · 分流可还原 · 实时日志明细；末尾一句：**接入成本与易用性、数据价值**。

> **提问：**为什么"测试、预发、生产三类环境"的支持，被列为平台的必备能力而不是加分项？

> **预设回答：**

- 答"方便开发调试"——只是表层。真实原因是**配置会串环境**：同一个实验配置中心如果被三个环境共用，测试环境调的比例就会打到生产。

- 答"因为生产不能随便改，所以要先在别的环境验证"——正确。这是环境隔离的基本逻辑，和代码分支隔离是一回事。

- 答"三类不够，还得有灰度环境和多机房"——很专业，说明环境维度还不止三个：地域、机房、灰度池都要能隔离。

- 点评：这道题的关键词是"**隔离**"。实验配置是一种"会直接改变用户所见"的配置，它比普通业务配置更需要环境隔离与权限控制。

> **易错点 / 考点：**本条实践的核心是"**易用性与接入成本**"，考题常问"为什么接入成本决定 A/B 平台的价值"——答：**实验数量与迭代速度取决于接入成本，成本高则实验少，平台的投资回报就无从谈起**。易错：把"白名单"答成"白名单用户不参与实验"（正好相反，白名单常用于**强制进组**）；把"多级缓存"只答成"提升性能"，漏掉"降低配置服务依赖、支持降级"。

> **过渡：**实践讲完，我们看真实平台长什么样——先看一块实验看板。

【7.202】A/B 测试 Dashboard（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是一块真实的实验看板，上面有几张数字卡片。别去念数字，先看**它在用什么口径讲故事**——这是这一页的教学重点。

看标着"Feature Y"的那一组卡片：**加权满意度成绩从 120.9 提到 153.8，（正面）满意比例从 27% 提高到 78%**，另外还有一个 **"Off vs On" 的 6.60%**。三个数，三种口径。

第一个是**绝对分值**：153.8 比 120.9 高，但"高 32.9 分"这件事本身说明不了好坏，因为分值量纲是自定义的。第二个是**比例**：27% 到 78%，涨了 51 个百分点——这是最能让人激动的一个数。第三个是**相对提升**："Off vs On 6.60%"，说的是开与关之间的相对差异。

同一份实验，三个数能讲出三种情绪。所以第一件要练的事是：**看到一个提升数字，先问它是分值、百分点，还是相对提升**。这三者混着说，就很容易夸大或缩小结论。

再看标着"Feature X"的那一组：**加权满意度 147.3 到 154.4**，满意比例 **85% 到 87%**，"Off vs On" 是 **2.3%**，另外还有一个 **5.5% 的满意度提升**。注意这一组的基线本来就高——85%，从 85 到 87 只涨 2 个百分点。**基线越高越难提升**，这是实验设计时就要有的心理预期：不是每个实验都能有前面那种量级的跃升。

第三点，**看板只是"看板"，不是"判据"**。它告诉你趋势，但"能不能上线"要靠统计检验：样本量够不够、置信区间是否跨零、护栏指标有没有变坏。看到绿色箭头就上、看到红色箭头就砍，中间没有检验这一步，是很多团队的常见病。

最后一点观察方法：**把两组卡片并排看**。同一块看板上，一组的提升是几十个百分点，另一组是两三个百分点，说明这两件事的"性价比"完全不同——资源该往哪投，一眼就看得出来。

板书：三种口径：绝对分值（120.9→153.8）｜比例/百分点（27%→78%、85%→87%）｜相对提升（Off vs On：6.60% / 2.3%）；读图三步：先问口径 → 再看基线高低 → 最后才谈显著性；一行：**看板 ≠ 判据**。

提问：产品经理指着看板说"这个实验提升了 78%，必须马上全量"。作为测试人员，你先问哪三个问题？

预设回答：

- 答"先问样本量多大"——对。样本太小的 78% 可能是几个人算出来的。

- 答"先问 78% 是百分点还是相对提升"——非常关键，这是本题的核心陷阱：27% 到 78% 是百分点，而"提升 78%"这种说法本身口径不明。

- 答"先问这是不是实时数据、跑够时间没有"——也很重要，实时看板往往是初步结果。

- 点评：三问合起来就是标准动作——**问口径、问样本、问时长**，之后才轮到显著性检验。

易错点 / 考点：常考"绝对变化、百分点、相对提升三者的区别"，这是数据分析的基本功。口诀：**百分点看减法，相对提升看除法**。易错：把看板数据当作上线判据；忽略基线差异（85% 的基线涨 2 个百分点，含金量

可能高于低基线涨 10 个百分点）。若考判断题"相对提升越大效果越好"——错，要看基线与显著性。

过渡：一块看板能看几个指标，但真实实验往往要同时看一整套指标，下一页就是那种情形。

【7.203】A/B 测试分析（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页列的是一组实验指标和它们的相对变化：**Average session length（平均会话时长）1%、Views Content occurrences per session（每次会话内容浏览次数）28%、Sessions per daily user（日均用户会话数）7%、Lifetime value per daily user（日均用户生命周期价值）60%、Subscribing occurrence per day（日均订阅发生次数）24%、Time in subscribing per daily user（日均订阅时长）24%、Sign In occurrences per daily user（日均登录次数）32%、Subscribed occurrences per daily user（日均订阅次数）9%、Daily Active Users（日活跃用户数）**。

数字我不逐条解释，重点讲**怎么读这一堆指标**，因为这里藏着 A/B 分析最常见的错误。

第一，**指标是有层级的**。这组指标从规模到商业价值大致是：日活（规模）→ 会话与浏览（参与度）→ 登录与订阅（关键行为）→ 生命周期价值（商业价值）。看的时候要有顺序：**先看规模和参与度有没有被影响，再看商业指标**。原因很简单——如果日活掉了 5%，那个 60% 的 LTV 提升就要打问号，很可能是分母变小造成的假象。

第二，**注意"每人""每次"这种归一化口径**。`per daily user`、`per session` 都是把总量除掉了人数或次数。这类指标有个特点：**分子分母可能同时变**。比如"日均登录次数提升 32%"，可能是登录变多了，也可能是活跃人数变少了、分母缩小。所以归一化指标**必须和总量指标一起看**。

第三，**防止"指标太多"的陷阱**。这里列了九个指标，如果每个都做显著性检验，按 5% 的显著性水平，纯随机也会有将近半个指标"显著"。这就是多重比较问题。规范做法是：**事先指定一个主指标**，其余作为解释性指标或护栏指标，只有主指标用来下结论。

第四，**看指标之间的因果链**。浏览变多（28%）→ 会话时长变动很小（1%）→ 订阅变多（24%）→ 生命周期价值提升（60%），这是一条大致合理的链路。如果出现"浏览涨了、时长没变、LTV 暴涨"，那多半是别的因素在起作用，或者埋点口径变了。

一句话：**指标不是越多越好，而是要有一条能自圆其说的因果链，并且事先定好谁是裁判。**

> **板书：**指标分层：日活（规模）→ 会话/浏览（参与度）→ 登录/订阅（关键行为）→ LTV（商业价值）；
> 两条纪律：归一化指标必配总量指标；**先定主指标，其余为解释/护栏**；一句：指标要有因果链。

> **提问：**"日均用户登录次数提升 32%"，这个数能不能直接当成好消息？

> **预设回答：**

- 答"能，登录多了说明用户更活跃"——最典型的错答。这是 `per daily user` 的归一化指标，分母是日活；如果日活掉了一半，登录次数照样能涨。

- 答"不能，要先看日活有没有掉"——正确。这是"分母陷阱"。

- 答"还要看是不是登录策略改了，比如强制重新登录"——非常专业。指标异动先排除"机制变化"，再谈"效果提升"。

- 点评：这道题的核心就是一句话——**凡是"人均""次均"的指标，必须和总量一起读**。

> **易错点 / 考点：**考"如何解读一组实验指标"，标准答案要包含：**分层看、归一化配总量、先定主指标、看因果链**。易错：把"相对变化百分比"当成绝对差异；忽略分母效应。另一常考点是"为什么指标多了反而不好"——多重比较导致假阳性上升。

> **过渡：**一次只测一个改动太慢，工程上会想一次测多个，这就是多因素实验。

【7.204】多因素的 A/B 实验（5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页讲**多因素的 A/B 实验**——一次不只测一个变量。课件给了四个实验和它们的结果：**Button Color Test（按钮颜色测试）+1.5、New Headline Test（新标题测试）+0.8、Totally New Landing Page（全新落地页）−4.0%、Another All-new Landing Page（另一个全新落地页）+6.0%**；旁边还标了"**A/B Testing Improvements to Statistically Significant Confidence**"，也就是"提升达到统计显著置信度所需要的时间"，对应 **6、8、4.5、2** 周。
>
> 先看结论层面的三件事。
>
> 第一，**改动幅度和收益不是线性关系**。改个按钮颜色，提升 1.5 个单位的指标；换个标题，0.8；而"全新落地页"一个 −4.0%、另一个 +6.0%。这说明什么？**大改动的方差大**：可能大赚，也可能大亏。所以大改动更需要

A/B，而不是更不需要——你凭直觉根本判断不了。

第二，**同样是"全新落地页"，两个结果一正一负**。这就是 A/B 的价值所在：它不告诉你"大改好不好"，它告诉你"**这一个**大改好不好"。设计质量才是决定因素，不是改动幅度。

第三，**达到统计显著所需的时间差别很大：2 周到 8 周**。为什么？因为达到显著需要的样本量取决于**效应量**：效应大，很快能看出来；效应小，就要更多样本。全新落地页 +6.0% 只需 2 周，新标题 +0.8 要 8 周——**小改动往往要付出更长的时间成本**，这笔账要在排期时就算进去。

再讲"多因素"本身。一次只测一个变量，好处是归因干净，坏处是慢：四个改动串行测，就是 6 加 8 加 4.5 加 2 周。多因素实验用**析因设计或正交设计**，一次把多个因子组合起来测，除了能算出每个因子的主效应，还能看出**交互效应**——比如"新标题"配"新落地页"可能相互增强，也可能相互抵消。代价是组合数增长很快、每组样本被摊薄。

给一句实务建议：**改动小、想快速筛，用单因素；改动多、想找组合最优，用多因素，但要先把因子数和水平数压住。**

板书：四个实验 → 按钮颜色 +1.5（6 周）｜新标题 +0.8（8 周）｜全新落地页 −4.0%（4.5 周）｜另一全新落地页 +6.0%（2 周）；三条结论：改动越大方差越大 · 大改≠更好 · 效应小则周期长；方法：单因素（归因干净、慢）vs 多因素（析因/正交、看交互、样本摊薄）。

提问：四个改动，你会串行一个一个测，还是一次设计一个多因素实验一起测？各有什么代价？

预设回答：

- 答"一起测最快"——快，但代价是两个：样本被摊到很多组，单组样本不足；交互效应会让"主效应"的解释变复杂。

- 答"一个一个测最清楚"——归因干净，但按课件给的周期算，四个串起来要二十周左右，产品等不起。

- 答"先多因素粗筛，再对胜出的组合做单因素确认"——最优解。先用较少的样本找出"哪些因子有用"，再对关键因子做一次更精细的验证。

- 点评：三答对应三种工程取舍——**速度、清晰度、成本**。成熟的实验体系一般是"两级走"：先筛后验。

易错点 / 考点：必记三点：**改动幅度大 ≠ 收益大**；**同类型改动结果可能相反**；**效应量决定达到显著所需时长**。易错：把"−4.0%"当成"负面结论"就放弃大改的思路（正确解读是"这一个方案不好"）；把多因素实验等同于"同时跑多个独立实验"（错，多因素是一个实验里组合多个因子，能算交互效应）。

> **过渡：**多因素的结果怎么读才不误判？下一页给了一个具体的分析场景。

【7.205】A/B 测试结果分析（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是结果分析的具体形态：两个落地页版本——Landing Page 01 和 Landing Page 02——对着测，每个版本上又叠了几个设计选项：**Show Product Video / No Product Video（放不放产品视频）**、**Remove Navigation / Add Navigation（去掉还是保留导航）**、**Longer Form Page / Shorter Form Page（长表单还是短表单）**，最后看两个结果：**Sum of newsletter Sign up（邮件订阅注册总数）** 和 **SUM OF AMOUNT（金额合计）**。

这张图要教的是"**看两个指标，而不是一个**"。

先说为什么要有两个结果指标。注册数代表**量**，金额代表**质**。一个方案完全可能"注册数上去了、金额下来了"——比如把表单缩短、把必填项砍掉，注册的人确实多了，但来的人意向弱、付费少。反过来，长表单会挡住一批人，但留下的人更精准。**只看注册数，你会选错方案；只看金额，你会漏掉规模效应。**

再说**主指标与副指标**的配合方式。规范做法是事先定一个主指标，比如金额；另一个作为约束条件，比如注册数不能掉超过某个幅度。两个指标同向，皆大欢喜；反向，就要做**权衡决策**，而这个决策是业务判断，不是统计判断——统计只能告诉你"差异有多大、是否显著"，不能告诉你"该不该接受这个交换"。

第三点，**选项之间会相互影响**。视频、导航、表单长度这三个选项不是独立的：去掉导航可能让用户更专注，也可能让他找不到路；加视频提高理解度，但拖慢加载。所以正确做法不是"逐个挑最好的"，而是**组合起来测**——这正好接上上一页的多因素实验。

第四点，**看结果要回到假设**。实验开始前写了什么假设？"加视频能提高注册"——那就先看视频这个因子的主效应，再看它和表单长度的交互。没有假设的分析，就会变成"在数据里翻找故事"，总能翻出一个看起来合理的解释，但那不是科学。

一句话：**量看注册、质看金额，先定主指标，再做组合，最后回到假设。**

> **板书：**两个版本（LP01 / LP02）× 三个选项（视频 有无 ｜ 导航 有无 ｜ 表单 长短）→ 两个结果（注册总数 = 量 ｜ 金额合计 = 质）；决策规则：主指标 + 约束指标；方法：组合测试 + 回到假设。

提问：实验结果是"注册数涨了 12%，但金额合计跌了 6%"。你会建议上线吗？

预设回答：

- 答"注册涨了就是好事，上"——最典型的错答。注册是中间指标，金额才是结果指标；12% 的量换 6% 的质，可能是亏的。

- 答"金额跌了就不上"——也片面。如果这批新增注册用户的复购周期还没到，金额下跌可能只是"时间未到"，不是"质量差"。

- 答"要先看这是哪个渠道、哪类用户、以及金额的时间窗"——正确。这道题的正确答案是"分情况"，前提是把渠道结构、用户类型、金额统计窗口都摊开看。

- 点评：这就是"业务判断与统计判断"的分工——统计告诉你差异显著，判断"这个交换值不值"是产品和商业的事。测试工程师的职责是**把两个数都摆上桌，不让任何一方被藏起来**。

易错点 / 考点：常考"主指标与护栏指标的关系"，答：**主指标判优劣，护栏指标一票否决**。易错：把"注册数"当最终目标（它是过程指标）；把"多个选项分别最优"等同于"组合最优"（存在交互效应）。若考简答"结果分析要回答哪几个问题"，答：**差异多大、是否显著、在谁身上有效、代价能不能接受**。

过渡：如果整体实验看不出结论，或者结论只对一部分人成立，怎么办？下一页给出办法。

【7.206】逐层递进的 A/B 实验：决策树（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是全节最"硬"的一张图：**逐层递进的 A/B 实验——决策树**。它解决的问题是：**整体实验没结论，或者结论太粗，怎么办。**

先看这张图长什么样。根节点上标着一组数：**158，n=38，100%**——158 是这个分组的指标值，n=38 是样本数，100% 是它占总体的比例。往下按条件分裂：**Pageviews 小于 8460** 分成两支，一支是 **117（n=7，18%）**，另一支是 **167（n=31，82%）**。再往下继续用 **Clicks 大于等于 775**、**Pageviews 小于 7720**、**Pageviews 小于 9907**、**Pageviews 小于 9765**、**Experiment 大于等于 0.5**、以及 **DOW 等于周一、周三** 或 **DOW 等于周日、周二、周五、周六** 这些条件一层层分下去，叶子上是一组组的数值和 n 值。

它要你看出三件事。

第一，**决策树在做"人群切片"**。整体平均 158，但左边那 18% 的人群只有 117，右边 82% 的人群是 167。也就是说，**这个改动对大多数人有效、对少数人群有害**，平均下来被稀释了。这正好印证前面对"评价"的讲法：要看效应在人群上是否一致。

第二，**分裂条件就是可执行的运营规则**。条件里有流量规模（Pageviews）、行为强度（Clicks）、实验标识（Experiment）、**星期几（DOW）**。这意味着最终结论不是"新版本更好"，而是"**新版本对周一周三访问、页面浏览量低于 8460 的那批用户更好**"。这个结论能直接落到投放策略、灰度范围和产品设计上。

第三，也是最要紧的一条——**小样本的叶子不能当结论**。图里有些叶子的 n 只有 1、2、3，它们的指标值可能很夸张，但那只是一两个人或三五个人的结果。**n 太小就是噪声**。所以读决策树必须同时看两个数：叶子上的指标值，和括号里的 n。

还要提醒一个风险：**逐层分裂会放大假阳性**。你分了二十个分支，总有一两个"显著"的，这属于数据挖掘式的过度拟合。规范做法是：**用一段数据建树，用另一段数据验证**，验证通过了才敢上线。

一句话：**决策树把结论从"要不要上"变成"对谁上"，但它必须配两个东西——样本量门槛和独立验证。**

板书：根：158（n=38，100%）→ Pageviews＜8460 → 117（n=7，18%）｜167（n=31，82%）→ 继续按 Clicks≥775、Pageviews＜7720/9907/9765、Experiment≥0.5、DOW 分裂；读法：**指标值 ＋ n 一起看**；两道闸：样本量门槛 · 独立数据验证。

提问：图里有个叶子的指标值高得离谱，但括号里写着 n=2。这个发现能作为上线依据吗？

预设回答：

- 答"能，数值这么高说明有效"——最典型的错答。n=2 意味着两条样本，任何两个人行为特殊一点都能造出极端值。

- 答"不能，样本太小"——正确。要给出经验门槛：分群分析时每组至少要能支撑一次显著性检验。

- 答"可以用来提出假设，但不能当结论"——最佳答案。这是决策树的正确用法：**用它发现线索，再用一次独立实验去验证**。

- 点评：这道题把"探索性分析"和"确认性结论"的区别讲透了。**探索可以大胆，结论必须保守。**

易错点 / 考点：三个采分点：**人群切片、条件即规则、小样本不可信**。易错：把决策树当成"发现最优人群就直接上线"的工具（缺独立验证）；只看叶子的指标值不看 n。若考"逐层递进实验的风险"，标准答案：**多重比较导致的假阳性、过度拟合、样本被切碎后每组不足**。

> **过渡：**理论讲完，最后看一个真实的商用 A/B 平台是怎么搭的。

【7.207】A/B 测试 Testin 技术架构（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是 **Testin（云测）的 A/B 测试技术架构**，看它是怎么把一个 A/B 平台分层搭起来的。看完你应该能自己画出一个平台的骨架。

按请求和数据的走向，大致是这么几层。**接入层**：SDK 与 API，负责和业务系统对接，拉配置、报数据。**分流/决策层**：实验配置的存储与计算、分流算法、规则匹配——这一层就是前面 OpenResty 模型干的事。**数据采集与处理层**：埋点日志的接收、清洗、落库、聚合。**分析计算层**：实时指标与离线指标两套计算链路。**展示与运营层**：看板、报表、告警、权限管理。

为什么要分层？三个理由。

第一，**关注点分离**。分流的可用性要求是"毫秒级、不能错"，分析的可用性要求是"数据全、能重算"，两者的技术要求完全不同。混在一起，改报表逻辑会牵连分流服务，风险不可控。

第二，**可以独立扩容**。分流层的压力跟请求量走（比如大促），分析层的压力跟数据量和查询量走。分开部署，各自扩容，成本才可控。

第三，**便于替换与演进**。分流算法可能要换、报表可能要接新的 BI 工具，分层之后，换一层不影响其他层。

看图还要注意两条"横切"的东西：一条是**配置与元数据中心**，它被分流层和分析层同时依赖，所以必须高可用；另一条是**监控与告警**，它横跨所有层，任何一层出问题都要能被看见。

最后给一句评价：**这张架构图和前面讲的机制是对得上的**——终端与 SDK 对应接入层，OpenResty 网关对应分流层，abTraceId 与埋点对应采集层，实时与离线报表对应计算层。你把它们对上，这一节就通了。

> **板书：**接入层（SDK/API）→ 分流决策层（配置＋算法＋规则匹配）→ 采集处理层（日志接收・清洗・落库・聚合）→ 分析计算层（实时链路 / 离线链路）→ 展示运营层（看板・报表・告警・权限）；横切：配置元数据中心（高可用）・监控告警。

> **提问：**如果把分流服务和报表统计放在同一个服务里，会带来什么具体问题？

> **预设回答：**

- 答"会慢一点"——不只是慢。最严重的是**故障传导**：报表的一次慢查询把连接池占满，分流接口跟着超时，用户直接看到错误页。

- 答"扩容不方便，两者压力来源不同"——正确。分流跟 QPS 走，报表跟数据量与查询复杂度走，绑在一起只能按峰值配置资源。

- 答"发布互相牵连"——也对。改个报表口径就得发一次版，而发版期间分流是不该中断的。

- 点评：三答分别对应**故障隔离、弹性伸缩、独立发布**——这也是所有分层架构的三个通用理由。

> **易错点 / 考点：**分层名称与职责要能对应，最常考"为什么分流与分析要解耦"。易错：把"实时/离线"当成两套独立系统（它们是同一分析层的两条链路，共用数据源）；漏掉"配置中心必须高可用"这一条。若考画图题，按五层加两条横切画即可。

> **过渡：**看完别人的架构，我们落回自己——如果让你设计，要定清哪几件事？

【7.208】A/B 测试平台设计（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲**平台设计**——如果让你从零设计一个 A/B 平台，要定哪几件事。这一页我按"设计清单"讲，因为课程设计要求你们自己动手搭一个简化版。

第一件，**领域模型**。至少要有的实体：应用、场景、层、实验、桶、指标、指标快照。它们之间的关系就是前面那张图：应用下有场景，场景下有层，层下有实验，实验分桶。模型定错，后面所有功能都要返工。

第二件，**状态机**。一个实验要有明确的生命周期：草稿、待审核、运行中、已暂停、已结束、已归档。每个状态谁能操作、能改什么，都要定死。没有状态机的平台，线上一定会出现"改了运行中实验的流量比例，导致前后数据不可比"的事故。

第三件，**流量与互斥校验**。发布实验前必须校验：同一层内所有实验的流量之和不超过百分之百，桶之间不重叠，白名单用户不冲突。这是平台的"安全带"，靠人不靠系统一定出错。

第四件，**分流与配置下发**。要定清楚：分流算法（如 MurmurHash 三因子）、配置存储（如 Redis）、下发方式（拉取还是推送）、缓存策略、降级策略（配置服务不可用时按默认版本放行）。

第五件，**埋点与数据模型**。要定清指标口径：分子分母是什么、时间窗口按自然日还是滚动 24 小时、去重规

则是什么。口径不定，两个团队算出来的转化率能差一倍。

第六件，**权限与审计**。应用级角色、谁能发布、谁只能看，操作日志全留。这一条在真实公司里往往决定平台能不能被信任。

第七件，**报表与检验**。至少要有：样本量、指标值、置信区间、显著性、护栏指标。没有显著性计算的平台，只能算"看板"，不能算"实验平台"。

最后一句：**A/B 平台的核心不是界面好不好看，而是四件事——分流准、口径清、数据全、结论可复算。**

板书：七项清单：① 领域模型（应用-场景-层-实验-桶-指标）② 状态机（草稿→运行→暂停→结束→归档）③ 流量与互斥校验（同层 ≤100%、桶不重叠、白名单不冲突）④ 分流与下发（算法・存储・拉/推・缓存・降级）⑤ 埋点与口径（分子分母・时间窗・去重）⑥ 权限与审计 ⑦ 报表与检验（样本量・置信区间・显著性・护栏）；底部四字：**准 ・ 清 ・ 全 ・ 可复算**。

提问：为什么"流量与互斥校验"必须做成系统强制，而不是靠发布同学自己检查？

预设回答：

- 答"人会算错"——对，但这只是表层。真实原因是**流量是动态的**：今天这一层有 60%，明天加一个实验变 80%，第三天再改一个桶，靠人脑记根本不可能。

- 答"校验能防止上线事故"——正确。同层流量之和超过 100% 时，落在这个区间的用户会被两个实验同时命中，数据直接作废。

- 答"可以靠评审会"——不够。评审会能防设计错误，防不了"临时改配置"。**约束要落在系统里，不落在流程里**，这是工程化的基本要求。

- 点评：这就是"把制度写进代码"的典型例子——**凡是能被系统拦住的事故，都不要指望人记住**。

易错点 / 考点：七项清单里，"**领域模型**""**状态机**""**流量互斥校验**"是设计题的高频采分点。易错：把"实验"和"桶"当成同一层概念（实验是策略，桶是流量容器）；漏掉"降级策略"（配置服务挂了怎么办）。若考"平台设计中最容易忽略什么"，答：**指标口径（分子分母、时间窗、去重）**，因为它不出现在界面上，却决定所有结论。

过渡：这一节的内容到这儿就讲完了，我们把整个第 7 章收一下。

【7.209】本节小结（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

到这儿，第 7 章的内容就收完了。这一页的小结把六个要点串起来，我按"三个专项加 A/B 三件事"来讲。

第一条，**兼容性测试覆盖软件与硬件、软件与软件、数据之间的兼容性，而且数据兼容更为重要，因为数据更具价值**。为什么数据兼容最重要？因为硬件不兼容换台机器就行，软件不兼容升个版本也行，但数据一旦乱了、丢了、读错了，是补不回来的。这就是"数据更具价值"的含义。

第二条，**全链路压测可看作是性能测试的右移，全面地提升系统的稳定性**。关键词是"右移"——把压测从测试环境搬到生产环境，用真实流量、真实依赖、真实数据去验证容量。

第三条，**混沌工程可看作是故障注入测试的右移，并提升为工程层次，遵守五项原则，践行优秀实践，熟练使用开源工具**。关键词是"提升为工程层次"：它不只是随手注入一个故障，而是要有假设、有爆炸半径控制、有观测、有复盘。

第四条，**A/B 测试是为了提高用户体验采用的科学方法，让数据说话**。"科学方法"和"让数据说话"这两个词组是它的定位——它是本章唯一一个**面向产品和运营决策**的测试类型，不是面向缺陷的。

第五条，**A/B 测试包括实验设计、流量分发控制算法等关键内容**。这两块正好对应今天讲的两半：实验设计（假设、指标、样本量、时长、显著性）和流量分发（场景、层、桶、正交互斥、MurmurHash、OpenResty 网关）。

第六条，**建立一个灵活支持 A/B 测试的平台**。这一条是给你们的作业预告——平台化是让 A/B 从"某个团队的特殊技能"变成"全公司基础设施"的唯一路径。

把这一章拉高一点看：**性能测试问"扛不扛得住"，安全测试问"守不守得住"，兼容性问"到处能不能跑"，全链路压测和混沌工程问"真实环境里崩不崩"，A/B 测试问"哪个方案更好"。** 五个问题，构成"专项测试"的完整拼图。

> **板书：**六条：兼容（数据兼容最重要）｜全链路压测＝性能右移｜混沌＝故障注入右移并工程化｜A/B＝科学方法、让数据说话｜A/B 两要素＝实验设计 ＋ 流量分发算法｜平台化；底部一行：**五个问题 → 专项测试的完整拼图**。

> **提问：**本章五个专项里，哪一个最"贵"？哪一个最"便宜"？为什么？

预设回答：

- 答"性能最贵，要买压测环境和机器"——有道理，全链路压测还要把生产流量导进来，成本确实高。

- 答"A/B 最便宜，改个按钮就行"——前半句对。小实验确实便宜，但**平台建设不便宜**：分流网关、埋点体系、数据链路、报表，是一次性重投入。

- 答"贵不贵要看它拦下的事故值多少钱"——最成熟的回答。专项测试的成本必须和它避免的损失放在一起算：一次大促宕机、一次数据串号、一次错误的定价决策，损失都是数量级的。

- 点评：三答的层次是"资源成本 → 建设成本 → 风险成本"。考试若问"为什么企业愿意投入专项测试"，答：**因为它拦下的风险远大于投入**。

易错点 / 考点：小结页是期末复习的第一入口，六条要能默写。易错：把"全链路压测"和"混沌工程"都记成"右移"却不区分内容——前者右移的是**性能验证**，后者右移的是**故障注入**。另一常考点是"数据兼容为什么最重要"，答：**数据不可重建、错误会持久化并扩散**。

过渡：这一章讲的是机制，下一页给的是入口——你们要自己动手搭一个。

【7.210】去搭建一个简单的 A/B 测试平台（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是给你们的动手任务：**去搭建一个简单的 A/B 测试平台**。课件给了两个入口，都是"开源/免费 A/B 测试工具"的清单页：一个是 convert.com 的博客文章，一个是 posthog 的博客文章，地址在课件上，你们照着去翻。这两篇的价值不在于它推荐了哪个工具，而在于让你看到**这个领域的方案光谱**。

清单里的方案大致分三类，你看的时候可以按这三类去归类。

第一类，**轻量前端脚本型**：一段脚本嵌进页面，改文案改样式，自带分流和统计。优点是十分钟能上手；缺点是只能测前端，服务端实验、多端一致性都做不到。

第二类，**自建/开源自托管型**：把平台部署在你自己的服务器上，数据不出内网，能接自己的埋点体系。优点是可控、可改造；缺点是要自己运维、要自己保证分流服务的可用性。

第三类，**商业 SaaS 型**：功能最全，统计功效、分群分析、样本量计算都有；代价是数据要出去、费用按流量算。

选型建议很实际：**课程设计用第二类或第一类**——你们的目的是把机制跑通，不是买服务。

具体作业我建议按五步做。第一步，**选一个场景**：比如课程设计里那个小程序的一个页面，或者一个注册流程。第二步，**做两个版本**，明确差在哪一个变量上。第三步，**实现分流**：可以用 MurmurHash 三因子，也可以先用最简化的哈希取模，但必须保证同一用户稳定落桶。第四步，**埋点与统计**：至少记录曝光、点击、转化三个事件，并且带上实验标识和桶号。第五步，**跑满一个周期出报告**：报告里必须有样本量、指标值、置信区间，以及你的结论和"如果结论相反你会怎么做"。

最后提醒一句评分要点：**做得漂亮不如做得对**。分流不稳定、口径不清楚、样本量不够就下结论——这三条任何一条踩了，功能再花哨也是零分。

板书：三类方案：前端脚本型（快・只测前端）｜开源自托管型（可控・要运维）｜商业 SaaS（全・要出网要付费）；作业五步：选场景 → 做两版 → 实现分流（稳定落桶）→ 埋点（曝光/点击/转化 + 桶号）→ 跑周期出报告（样本量・置信区间・结论）；评分红线：**分流稳、口径清、样本够**。

提问：如果让你用两天时间做出一个"能跑通"的 A/B 平台，你会砍掉哪些功能、保留哪些功能?

预设回答：

- 答"先做报表，界面对评分重要"——最典型的错答。报表是展示，分流和埋点错了，报表再漂亮也是错的。

- 答"保留分流＋埋点＋一个最简单的对比报表，砍掉权限、状态机、多因素"——正确。这就是最小可用闭环（MVP）：**能分组、能记录、能对比**。

- 答"还要保留'输入用户标识查桶号'这个功能"——很好，这是排查能力，两天也能做出来，性价比极高。

- 点评：三答的差别就是"什么是骨架、什么是装饰"。判断标准只有一条：**去掉它，实验结论还可信吗？** 去掉分流稳定性，结论不可信；去掉权限管理，结论仍然可信（只是不安全）。

易错点 / 考点：这一页是实践页，考点通常落在"三类方案怎么选"和"最小可用平台包含哪几块"。易错：把"装一个开源工具"当成"搭建平台"（缺自己的埋点与口径就没有可比数据）；忽略"同一用户稳定落桶"这个最低要求。作业评分常看三条：**分流可复算、埋点带标识、报告有统计量**。

过渡：动手之前，先看看有哪些书可以当参考。

【7.211】学习资源推荐（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

最后推荐几本书，我按"你这门课接下来会用到什么"来说，不按出版顺序。

第一本，**《软件性能测试、分析与调优实践之路》，清华大学出版社，2020 年 7 月**。这本对本章性能测试那部分是直接延伸，特点是"实践"——它不只讲怎么压，还讲压出问题之后怎么定位到代码和数据库。你要是课程设计做后端性能实验，这本最对症。

第二本，**《全栈性能测试修炼宝典 JMeter 实战（第 2 版）》，人民邮电出版社，2021 年 5 月**。课上讲的 JMeter 只是骨架，这本是完整手册：元件、插件、分布式、报告。做性能实验的时候当工具书翻。

第三本，**《Web 安全攻防：渗透测试实战指南》，电子工业出版社，2018 年 7 月**。对应安全性测试那一节。提醒一句：这本书讲的是攻击手法，看的时候请始终带着"我在学怎么防"的立场——知道怎么打，是为了知道怎么守。

第四本，**《超大流量分布式系统架构解决方案》，电子工业出版社，2020 年 3 月**。对应全链路压测那部分。性能问题到了"全链路"这个层次，本质是架构问题：缓存、限流、降级、分库分表。这本书讲的是架构侧怎么配合。

第五本，**《混沌工程：复杂系统韧性实现之道》，机械工业出版社，2021 年 6 月**。对应混沌工程。它是这个领域比较系统的一本，原则、实验设计、爆炸半径控制都讲得比较细。

第六本，**《A/B 测试：创新始于试验》，机械工业出版社，2019 年 4 月**。对应今天这一节。书名里"创新始于试验"这五个字，其实就是 A/B 测试的方法论内核：**别争论，去试验**。

六本书，覆盖本章六个方向。我的建议不是全买，而是**先挑一本和你课程设计题目最相关的精读，其余当参考**。技术书的价值在于"你遇到问题时它在手边"，不在于"你读完了几本"。

> **板书：**六本 → 性能实践｜JMeter 实战｜Web 安全攻防｜超大流量架构｜混沌工程｜A/B 测试：创新始于试验；一句：**先精读一本对题的，其余当参考**。

提问：这六本书里，如果只能带一本进课程设计答辩现场，你带哪本？为什么？

预设回答：

- 答"带最厚的那本，显得准备充分"——形式主义。答辩看的是你能不能回答问题，不是书有多厚。

- 答"带我课程设计题目对应的那本"——正确。比如做性能优化就带性能实践，做混沌实验就带混沌工程。

- 答"哪本都不带，带自己的实验记录和缺陷清单"——最佳答案。**答辩最有力的材料是你自己的数据和复现步骤**，书只是支撑。

- 点评：这道题想说的是——**资料的价值在"能支撑你的判断"，不在"读过"**。这也是整个课程设计的态度。

> **易错点 / 考点：**资源页一般不作为考点，但常出现在"课程设计参考资料"要求里。要注意的是**书名的准确性**：五本都是"出版社＋年份"，写报告时引用格式别错。另一个常见问题是把"性能测试"和"性能调优"混为一谈——前者是发现，后者是解决，第一本书名里两个词都有，正好说明它们是两件事。

> **过渡：**这一章就到这里，我用三句话收个尾。

【7.212】感 谢 聆 听（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 第 7 章到这里结束，最后我用三句话收这一章。
>
> 第一句，**专项测试的判据不是"对不对"，而是"达不达标"**——性能看响应与容量，安全看攻击面，兼容看环境适配，可靠看故障下的表现。
>
> 第二句，**越往后走，测试越像工程而不是手艺**——全链路压测、混沌工程、A/B 测试，靠的都是平台、数据、算法，而不是某个人的经验。
>
> 第三句，**让数据说话，但别让数据替你说谎**——口径要清、样本要够、结论要经得起复算。
>
> 下一次课我们进入第 8 章，**软件本地化测试**——同一套软件换语言、换地区、换文化，还能不能用、好不好用。这一章会用到今天讲的兼容性思路，但问题更细：文字长度、日期格式、货币、排序规则、翻译质量。
> 带着一个问题回去想：**一个中文界面做得再好，翻成德语、阿拉伯语之后，还可能坏在哪里？**

> **【思政落点 13】**：这一章教了性能压测、漏洞攻防、混沌注入、A/B 分流——这四样东西换个场合使用，就是打垮一个系统的四种手段：压测变拒绝服务，漏洞变入侵，故障注入变破坏，分流变操纵用户。技术本身没有立场，立场在使用它的人。请记住：**能力越强，越要问一句"我这么做，是在让别人变好，还是只在证明我厉害"**。把本事用在把系统做稳、把用户当人上，这门手艺才立得住。

> **板书：**三句：**达不达标 → 工程不是手艺 → 让数据说话但别说谎**；下一站：第 8 章 软件本地化测试（语言・

> 地区·文化）。

> **提问：**这一章学完，你手上多了四件"能伤人的工具"——压测、攻防、故障注入、分流。你给自己定的使用边界是什么？

> **预设回答：**

- 答"只在授权范围内用"——必要但不充分。"授权"是外部约束，很多伤害发生在授权范围之内（比如用数据诱导用户）。

- 答"不做损害用户利益的事"——更进一层，这是把判断标准从"允不允许"换成"该不该"。

- 答"用之前问自己：如果这件事被公开报道，我还站得住吗"——很实用的自检法。工程伦理很多时候不需要复杂推演，只需要一个"公开检验"。

- 点评：三答的层次是"合规 → 良知 → 可公开"。**技术能力的天花板由学习决定，职业高度由这条边界决定。**

> **易错点 / 考点：**结束页不作知识点考点，但课程设计答辩常问"你在项目中如何保证测试结论可信"。可以借本章的三句话作答：**判据明确、过程可复现、结论可复算**。另一常被问的是"专项测试和通用测试的分工"，答：**通用测试保证"做对"，专项测试保证"在真实条件下仍然成立"**。

> **过渡：**本章到此结束。下一次课，我们把这套"专项"的思路带到本地化测试里去。

四、思政落点清单（共 13 处）

序号	所在页	落点主题
1	7.6	内存泄漏最值得琢磨的地方是——它每一次泄漏都很小。一个请求漏几十 KB，跑一小时看
2	7.7	三个目标里最容易被糊弄的是第二个——"验证是否达标"。现实中不少报告会悄悄降低标
3	7.13	渗入测试最像我们做事的方式——它不靠聪明，靠熬。跑一夜、跑一周，盯着曲线看有没有
4	7.34	性能分析最像科学研究的日常：提出假设、找证据、做对照实验、被数据推翻、再来一轮。
5	7.77	渗透测试这门手艺有个天然的两面性：懂它的人既能找出系统的破口，也能利用这个破口。
6	7.91	SQL 注入这一页，技术上只有一行 payload，但它背后的道理值得多说两句。同一

7	7.115	数据兼容这一页，我想请大家对"历史数据"保持一点敬畏。系统里那批数据不是测试数据
8	7.130	全链路压测这一页，最能体现"严谨"两个字。它要求我们先界定范围、再评审方案、再造
9	7.151	爆炸半径控制讲的是"敢试，但试得有边界"。工程上的勇气从来不是"什么都敢在生产上
10	7.160	从灾难中学习，前提是允许把问题讲出来。复盘文化最怕两件事：一是把复盘开成追责会，
11	7.162	爆炸半径这四个字，本质上是对风险的敬畏。做工程的人容易有一种自信：我懂这套系统，
12	7.199	A/B 测试的说服力来自"数据不撒谎"，可数据不会自己说话，是人在解释它。同一份报
13	7.212	这一章教了性能压测、漏洞攻防、混沌注入、A/B 分流——这四样东西换个场合使用，就

第 8 章　软件本地化测试

第 10 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 8 章 讲稿（第 10 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 8 章 软件本地化测试 —— 同一套代码，能不能在另一个语言和地区"像本地产品一样好用"
>
> 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 **46 页**（`lecture8/`，云端 http://111.231.0.226:2345/lecture8/index.html）
>
> 教案对应：第 10 次课「软件本地化测试」
>
> 本讲重点：国际化（i18n）与本地化（l10n）的区别；本地化测试的测试项（翻译质量、界面布局与截断、日期时间与货币格式、排序与检索、输入法与字符集、图片图标、法规与习俗）；测试组织（单语言/多语言环境、伪本地化、翻译验证与回归）
>
> 本讲难点：为什么"能显示中文"不等于"本地化做对了"；伪本地化如何在不翻译的情况下提前暴露布局问题；字符集与编码问题的定位思路

> **本讲稿为可照读完整版**：每页含【口播】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能区分国际化与本地化，并说明各自发生在哪个阶段、由谁负责；② 能列出本地化测试至少 8 类测试项；③ 能解释伪本地化的原理与价值；④ 能说出编码/字符集问题的常见现象与排查方向；⑤ 能说明本地化回归测试的范围怎么定
能力	① 能为一个已上线的中文系统列出"出海到英文与阿拉伯语市场"的本地化测试清单；② 能设计一套伪本地化验证方案；③ 能判断一个界面问题是翻译问题、布局问题还是编码问题
素质（思政）	① 由编码/乱码问题体会"细节决定成败"；② 由习俗与法规禁忌体会尊重文化差异与合规意识；③ 由翻译质量体会对用户负责；④ 由测试组织体会流程与严谨

本讲要建立的一个总印象：本地化测试测的不是"翻译对不对"，而是"**这个产品在这个市场里，像不像一个本地产品**"——语言、格式、习惯、法规，四样都要对。

二、时间分配总表

> **容量提示**：本章课件 **46 页**，讲稿是素材全量版（口播约 2 万字，≈85 分钟），课时 90 分钟，**基本可以按页讲完**；仍建议把纯图示页与工具页快速带过。

时段	页码	内容	分钟	备注
第 1 节	8.1–8.12	开场、i18n 与 l10n、本地化测试的定位与测试项	26	**测试项清单必讲**

第 1 节	8.13–8.26	界面布局与截断、格式本地化（日期/时间/货币/数字）、排序与检索	24	现场找截断例子
——	——	**课间休息 10 分钟**	——	——
第 2 节	8.27–8.38	字符集与编码、输入法、图片图标、法规与习俗	20	**编码问题必讲**
第 2 节	8.39–8.46	测试组织与执行、伪本地化、回归、小结与作业	15	伪本地化当堂设计

> **压缩优先级**：先压缩工具截图页与流程图示页 → 再压缩具体语种细节 → 最后才压缩"测试项清单""编码问题""伪本地化"三处。

三、逐页讲稿

> 每页第一段是可直接朗读的口播；其后是板书、提问与预设回答、易错点/考点、过渡语。
>
> 页码锚点与课件页一一对应（8.1–8.46）：http://111.231.0.226:2345/lecture8/index.html

【8.1】软件测试方法和技术（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 各位同学，我们进入第 8 章——**软件本地化测试**。
>
> 前面几章我们一直在解决同一个问题的不同侧面：代码里的单元对不对、模块拼起来通不通、系统整体功能对不对、性能和安全扛不扛得住。这些工作有一个共同的前提，你可能从来没注意过——**我们默认这个软件是给中国人用的**：界面是中文，日期是年-月-日，货币是人民币，姓名是姓在前、名在后。
>
> 这一章把这个前提掀掉。同一个软件装到阿拉伯语用户的机器上，界面要从右往左排；装到德语用户的机器上，同一句话会长出一截，按钮装不下；装到美国用户的机器上，2/3/2009 到底是二月三日还是三月二日，居然说不清。这都不是"功能对不对"的问题，是"**换个地方还能不能用、用得顺不顺**"的问题。
>
> 这一章回答三件事：国际化 i18n 和本地化 l10n 差在哪、本地化测试要查什么、这些测试怎么组织和执行。
>
> 带着一个问题听：**你的软件如果明天要卖到德国和沙特，你手上的测试用例，有几条还能直接用？**

> **板书：**正中写"第 8 章 软件本地化测试"；左侧竖写"前面各章：默认说中文（对错・性能・安全）"；右侧写"第 8 章：换个国家还行不行（i18n / l10n）"。

提问：一个在国内用得好好的软件，直接拿到国外卖，你猜第一批用户投诉会集中在哪些地方？

预设回答：

- 答"翻译错了、翻译得很怪"——这是最直接的一类，但它只是本地化问题的一部分，后面会看到还有格式、布局、文化三类。

- 答"界面乱码、日期看不懂、钱的面额不对"——很准，这三条正好对应字符集、数据格式两个技术板块。

- 答"把界面翻译成英文就能卖了"——这是最典型的错答。翻译只是 L10N 六项工作里的一项，代码没做国际化改造、日期和货币仍是硬编码，翻译完了照样用不了。

- 收口：所以本章的顺序是——**先有国际化能力，才谈得上本地化质量**。

易错点 / 考点：① 本章两条主线必须分清：i18n 是"让软件具备被本地化的能力"，l10n 是"把它变成某个地区的版本"；② 本地化不等于翻译，这是全章反复考的判断；③ 判断题陷阱："本地化测试只在测试阶段做"——错，它从设计、编码就开始埋下成败。

过渡：先不谈定义，我们看几个你我都遇到过的生活场景，你就知道这一章为什么必须单独成章。

【8.2】国际化生活的体验（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页课件很客气，只列了四个小场景，但这四个场景把本地化测试的全部难点都点到了。我们一个一个过。

第一个，**去美国旅行，最感不适应的是什么**。不是语言，是那些"看起来一样、规则不一样"的东西：插座电压、温度用华氏度、距离用英里、重量用磅、纸张尺寸不一样、星期一算一周的第一天还是星期日算第一天。这些都不是对错问题，是**约定不同**。软件的本地化，本质上就是把这些约定一条条对齐。

第二个，**2/3/2009 代表哪一天**。这个问题值得停三秒。美国人写月/日/年，所以是二月三日；欧洲人、很多亚洲国家写日/月/年，所以是三月二日。同一串数字，两个答案，**而且两个都"对"**。这就是本地化测试里最阴的一类缺陷：程序没报错、没崩溃、数据也存进去了，只是含义错了。用户看到账单日期差一个月，这在金融系统里是事故。

第三个，**George Bush 属于哪个家族**。这个问题问的是姓名和称谓。中文习惯姓在前，英文习惯名在前；中文叫"布什先生"，英文叫 Mr. Bush，称谓的位置反过来。还有音译问题：Bush 译成"布什"，同一个家族的人译法要一致，否则检索、排序、去重全乱。看似小事，客户档案系统里一个姓名规则错了，几十万条数据就得重洗。

第四个，**收到邮件出现乱码**。这是字符编码问题，也是国内用户最熟悉的一种本地化故障。乱码的本质是"**写的时候用一种编码，读的时候用另一种编码**"。

四个场景归成一句话：**本地化测试要防的，不是程序算错，而是程序按别人的规矩算对了、按你的规矩算错了。** 这句话是本章的总纲。

板书：中间写"国际化生活的体验"；四角写四个场景：① 美国旅行（度量衡・纸张・一周起点）② 2/3/2009（日期顺序歧义）③ George Bush（姓名・称谓・音译）④ 邮件乱码（编码不一致）。底部一行大字："不是算错，是规矩不同"。

提问：2/3/2009 这个日期，如果让你在程序里存下来，你怎么保证不产生歧义？

预设回答：

- 答"按美国习惯存，月/日/年"——那欧洲用户填进来的日期就会被读错，方向反了。

- 答"用四位年-两位月-两位日的写法"——对，这是国际上最不容易误读的写法，把"年"放最前面，因为年的位数最多、最不可能被当成日或月。

- 答"加个下拉框让用户选格式"——可以缓解输入歧义，但存储层如果还是字符串，导出、排序、跨系统交换时又会出问题，治标不治本。

- 追问收口：真正可靠的做法是**存储用无歧义格式，显示用本地格式**，两个层面分开管。这个"存储与显示分离"的思想，后面在数据格式那节还会再讲一次。

易错点 / 考点：① 日期歧义是本地化案例题的最爱，答题要点是"**显示本地化、存储标准化**"；② 姓名问题的三个子项：顺序、称谓位置、音译一致性，考试常只答"顺序"而丢分；③ 乱码的根因一句话：**编码与解码用的字符集不一致**，不是"字库坏了"。

过渡：光说生活场景还不够，我们看两张真实的界面，同一个软件在两个地方长什么样。

【8.3】具体例子（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章的第一组对照素材，请你带着"找不同"的心态看：**同一个软件、同一份数据，在中文简体的北京环境和英文的纽约环境里，呈现出来的东西差在哪。**

先看差异的类型，不要急着看具体字。第一类是**文字本身**：界面上的词换了语言，这是最容易被发现的。第二类是**格式**：日期怎么写、时间用 12 小时制还是 24 小时制、钱用什么符号、千分位用什么符号、小数点用点还是逗号。第三类是**排列**：谁在前谁在后——日期是年月日还是月日年，姓名是姓在前还是名在前，地址是从大到小还是从小到大。第四类是**度量**：长度、重量、温度、纸张。

这四类里，只有第一类会被用户立刻发现并且投诉；**后三类才是最贵的**，因为它们不报错。举个真实感很强的场景：一份报表在北京导出是"1,234.56"，在纽约导出可能变成"1,234.56"看着一样，但到了德语环境就可能是"1.234,56"——逗号和点的含义整个反过来了。如果下游还有个程序去解析这个字符串，它会把一千二百三十四读成一点二三。

所以看这组对照图，我要求你练一个动作：**每看到一处差异，就问一句"这个差异是显示层的，还是会流进数据层的"**。只停在显示层的，改起来便宜；流进数据层、存进数据库、传给下游系统的，就是缺陷，而且往往是发布以后才炸的缺陷。

再补一句方法上的话：这类对照不要靠眼睛扫，要**列成清单逐项过**——日期、时间、数字、货币、度量、姓名、地址、电话、排序。这张清单就是后面 8.3.1 那节要展开的六项数据格式。

> **板书：**左右两栏："中文简体・北京"｜"英文・纽约"。中间画四条横线，分别标：文字｜格式｜排列｜度量。底部写一句："显示层差异 = 便宜；数据层差异 = 缺陷"。

提问：这两张图里的差异，哪些属于"改翻译文件就能解决"，哪些改翻译文件解决不了？

预设回答：

- 答"文字变了的那种，改翻译文件就行"——对，这是资源文件层面的工作。

- 答"日期和钱的写法，也能在翻译文件里改"——错。格式是**代码按区域设置算出来的**，翻译文件里改不了；代码里写死了格式串，就得改代码。

- 答"说不好，得看它是怎么实现的"——这个回答最专业。判断依据就是：这个值是**格式化出来的**，还是**拼字符串拼出来的**。拼出来的，本地化必出问题。

- 收口：所以本地化测试看到一个可疑显示，第一反应应该是"**它是怎么生成的**"，而不是"翻译得对不对"。

> **易错点 / 考点：**① 四类差异（文字／格式／排列／度量）要能分类举例；② 常考判断："界面文字都翻译了，本地化就完成了"——错，格式类问题一条都没解决；③ 记住"显示层／数据层"这条分界，它是判断缺陷严重度的实用尺子。

> **过渡：**中文和英文好歹都是从左往右读，接下来这个例子会把方向也翻过来。

【8.4】示例—阿拉伯语（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是阿拉伯语的界面示例。阿拉伯语属于**从右到左**的语言，也就是常说的 RTL（Right-to-Left）。这一页要你看出来的，不是"这些字我不认识"，而是**整个界面的空间逻辑被镜像了**。
>
> 具体镜像在哪些地方？第一，**阅读方向反过来**：文字从右边起头，段落右对齐。第二，**控件顺序反过来**：原来左边的导航跑到右边，原来右边的滚动条跑到左边，进度条的起点在右边。第三，**标点和成对符号要翻转**：左括号和右括号在视觉上要对调，问号、逗号的位置也跟着变。第四，**数字和混合文本更麻烦**：阿拉伯文里夹着英文单词、夹着数字的时候，会出现"双向文本"的排序问题，一段话里既有右到左又有左到右，边界在哪儿、数字该按哪边读，都需要算法去判断。
>
> 为什么这件事对测试特别重要？因为它**不可能靠翻译解决**。你把界面文字换成阿拉伯语，如果布局代码是按"左对齐、左边距、控件从左往右排"写死的，翻完就是一场灾难：文字压在图标上、按钮顺序颠倒、进度条反着走。这类问题在中文和英文环境里**永远暴露不出来**，只有真的切到 RTL 语言才会出现。
>
> 所以业界有个做法：在做国际化改造时就把布局写成"**跟着语言方向走**"——该用起始边、结束边的地方，不要写死左右。测试的时候也一样，别只测中文和英文，至少挑一个 RTL 语言过一遍，把镜像逻辑验掉。这一条，是很多团队第一次出海时最惨的教训。

> **板书：**中间写"RTL：从右到左"；四支箭头分别写：阅读方向 → 右起・右对齐；控件顺序 → 导航/滚动条镜像；成对符号 → 括号翻转；混合文本 → 双向排序。底部："翻译解决不了方向，布局要跟着语言走"。

> **提问：**把界面文字全部换成阿拉伯语，但布局代码一行不改，你预计最先崩的是哪个地方？

> **预设回答：**

- 答"文字和图标会重叠"——很可能，因为文字变长、方向反了，原来的留白全对不上。

- 答"滚动条跑到左边，用户会找不到"——对，控件镜像没做就是这样。

- 答"应该没什么问题，字体能显示就行"——这是典型错答。方向、对齐、控件顺序全都要单独处理，光能显示字符离"能用"差得远。

- 追问收口：判断一个界面有没有真正支持 RTL，最省事的办法是看它**有没有把"左/右"换成"起始/结束"**这种与方向无关的表达。

> **易错点 / 考点：** ① RTL 的四个镜像点要能说出来；② 常考判断："支持双字节字符就等于支持阿拉伯语"——错，阿拉伯语是字符集问题加方向问题两件事；③ 记住术语 RTL／LTR，写简答题时用术语比用"从右往左"更得分。

> **过渡：** 场景看完了，我们回到教材的顺序，先把这一章的地图铺开。

【8.5】8.1 什么是软件本地化（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是本章的四块内容，我按顺序点一遍，你记结构。
>
> **8.1 什么是软件本地化**——定义与基础。要分清国际化 i18n 和本地化 l10n 的关系，要讲字符集、国际化标准、本地化的基本步骤，最后落到"本地化测试包含哪几类工作"。这一块是地基，占的时间最长。
>
> **8.2 翻译验证**——翻译是本地化里最显眼的一项工作，但"翻完了"和"翻对了"是两件事，这一节讲怎么验证翻译，以及翻译之外的文化层面要改什么。
>
> **8.3 本地化测试的技术问题**——三块：数据格式、页面显示和布局、配置与兼容性。这是本章最"干货"的部分，几乎每条都能直接变成一条测试用例。
>
> **8.4 本地化的功能测试**——把本地化版本当成一个完整软件去测：集成、索引排序、联机文档，以及"本土测试"和"翻译验证测试"这两种典型的组织形式。
>
> 四块有一条共同的主线：**本地化测试查的不是"程序算得对不对"，而是"程序在别人的规矩下还对不对"**。所以我们既要懂测试，也要懂一点别人的规矩。

> **板书：** 竖排四块：8.1 什么是本地化（i18n/l10n・字符集・标准・步骤・测试范围）→ 8.2 翻译验证（翻译

> 内容・文化心理・特殊符号）→ 8.3 技术问题（数据格式・显示布局・配置兼容）→ 8.4 功能测试（集成・排序・文档・本土测试・TVT）。右侧一句："别人的规矩"。

> **提问：**这四块里，你觉得哪一块最难自动化、最依赖人？

> **预设回答：**

- 答"8.2 翻译验证"——有道理，语言好不好要人判断，机器只能查术语一致性。

- 答"8.3 技术问题"——也对，很多是"看着别扭"，判据难写。

- 答"都能自动化，写脚本扫一遍就行"——错。格式、编码、布局的**检测**可以自动化，但"这个译法在这个文化里合不合适"，脚本给不出答案。

- 收口：所以本地化测试的典型配置是**工具做筛查、人做判断**，这一点和性能测试很像。

> **易错点 / 考点：**① 四块的名称与顺序；② 8.3 与 8.4 容易混：8.3 是"技术问题清单"，8.4 是"把本地化版本当完整软件测的组织方式"；③ 简答题常问"本地化测试包括哪些内容"，标准答案是 8.1.5 那六类，不要拿 8.3 的三块去答。

> **过渡：**先看 8.1 这一节内部的路线，五小节，一层比一层落地。

【8.6】8.1 什么是软件本地化（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 8.1 这一节分五小节，我给你画一条从"概念"到"能干活"的路线。
>
> **8.1.1 软件本地化与国际化**——先把 i18n 和 l10n 这两个词掰开。这是全章最容易混的一对概念，也是考试必考的一对。
>
> **8.1.2 字符集问题**——从 ASCII 讲到 Unicode、UTF-8。为什么要在这个位置讲编码？因为乱码是本地化故障里出现频率最高、用户最不能忍的一类，而它的根子全在字符集。
>
> **8.1.3 软件国际化标准**——五条设计要求：语言切换机制、与语言无关的输入输出接口、资源文件国际化、包容本地数据格式。这五条是"一个软件够不够格被本地化"的检查表。
>
> **8.1.4 软件本地化基本步骤**——八步流程，从配置管理、术语表，到提取、翻译、检入、编译、调 UI、再测试。

这一节回答的是"本地化这件事怎么组织"。

8.1.5 软件本地化测试——落到六类测试工作：功能性、翻译、可用性、兼容性、文化宗教、手册验证。

一句话串起来：**概念分清楚（8.1.1）→ 底层看编码（8.1.2）→ 设计看标准（8.1.3）→ 执行看流程（8.1.4）→ 验收看六类测试（8.1.5）**。

板书：五格横排：8.1.1 概念（i18n/l10n）→ 8.1.2 编码（ASCII・Unicode・UTF-8）→ 8.1.3 标准（五条）→ 8.1.4 流程（八步）→ 8.1.5 测试（六类）。下方一行："概念 → 底层 → 设计 → 执行 → 验收"。

提问：为什么讲本地化测试，要花一整节去讲字符编码？

预设回答：

- 答"因为乱码最常见"——对，这是最直接的理由。

- 答"因为编码不对，后面的测试全是假的"——这个更深一层：编码错了，你测出来的"翻译不准确"可能只是显示错了。

- 答"编码属于开发的事，测试不用管"——错。测试人员至少要能判断"这是编码问题还是字体问题还是翻译问题"，否则提的缺陷单会被开发退回来。

- 收口：**能定位到层次，才提得出有效的缺陷**。

易错点 / 考点：① 五小节的顺序与关键词；② 8.1.2 与 8.3.2 的关系：前者讲原理，后者讲表现为乱码的现象与检查点；③ 别把 8.1.4 的"八步"和 8.4 的"本地化功能测试"混成一件事，前者是生产过程，后者是验证活动。

过渡：进入 8.1.1，先把 i18n 和 l10n 这两个词彻底分开。

【8.7】I18N vs. L10N（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是本章最重要的一页，四个判断句，我一句一句抠。先解释这两个怪词怎么来的：i18n 是 internationalization，首字母 i 和尾字母 n 中间隔 18 个字母；l10n 是 localization，l 和 n 中间隔 10 个字母。业界图省事就这么缩写。再加一个词 globalization，缩成 g11n，它把 i18n 和 l10n 都包进去，指"全球化"这件整体的事。

第一句，**I18N 是 L10N 的基础和前提，为 L10N 做准备**。这句话定的是先后关系：没有国际化能力的软件，做本地化是硬改，成本极高。打个比方，i18n 是盖楼时把水电管道预埋好，l10n 是装修。管道没埋，装修的时候就只能砸墙。

第二句，**L10N 是 I18N 向特定本地语言环境的转换**。注意"特定"两个字。国际化是一次性的、通用的；本地化是多次的、一个地区一份。中文简体一份、中文繁体一份、德语一份、阿拉伯语一份，每一份都是一次独立的本地化。

第三句，**I18N 是软件产品源语言开发的一部分，属于 Engineering**。这句话很关键，它决定了责任归属：国际化是研发的活，写代码的时候就要做，不是测试阶段补的。你让测试去发现"这个字符串硬编码了"，测试能发现，但改还是研发改。

第四句，**L10N 可以独立于 Engineering，可由第三方完成**。这就是为什么行业里有专门本地化公司：把资源文件发出去，翻译、回填、出多语言包。它像一个外包工序，可以独立发包、独立验收。

四句话合起来是一张组织图：**研发负责 i18n，本地化团队负责 l10n，测试两头都要验——验研发有没有留好接口，验本地化有没有填对内容。** 这也是本章后面所有内容的分工依据。

> **板书：**三行对照：I18N＝internationalization（通用能力・Engineering・研发负责・一次做好）｜L10N＝localization（特定环境・可第三方・一地区一份）｜G11N＝globalization（i18n＋l10n 的总和）。中间一句箭头："i18n 是前提 → l10n 是转换"。底部："预埋管道 vs 装修"。

提问：一个软件先做了德语版，再想做日语版，需要重新做国际化吗？

预设回答：

- 答"需要，每种语言都要重新国际化"——错。国际化是通用能力，做一次就够，多做几遍是浪费。

- 答"不需要，只要再做一次本地化"——对。前提是当初的国际化做得干净：字符串抽出去了、格式没硬编码。如果当初是硬改出来的德语版，那日语版就得再硬改一遍，这时候"要不要补国际化"就成了成本决策。

- 答"看情况，得看第一版是怎么做的"——最专业的回答。这正是判断"技术债要不要还"的典型场景。

- 收口：所以第四句话的现实含义是——**i18n 做得好，l10n 是复制；i18n 做得差，l10n 是重做**。

易错点 / 考点：① 四句判断必背，尤其"i18n 属 Engineering、l10n 可第三方"这两句最容易考；② 别把 g11n 当成第三种技术，它是前两者的合集；③ 简答高频："简述国际化与本地化的区别与联系"——答题结构就是"前提与转换＋责任归属＋复用性"。

> **过渡：**既然 i18n 是前提，那"具备国际化能力"具体指哪些能力？课件给了八条。

【8.8】I18N（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是国际化的能力清单，八条。我按"数据、代码、界面、语言习惯"四组给你归一下，这样好记。

第一组，数据层：第一条，**支持 Unicode 字符集、支持双字节的字符**。这是地基中的地基。字符集不支持，后面所有翻译都是乱码。第二条其实也在数据层：**支持各个国家的度量衡、时区、货币单位格式等的设置**——注意它说的是"设置"，意思是这些不该写死，要能配。

第二组，代码层：**分离程序代码和显示内容、消除 Hard code**。这两条是一件事的两面：凡是要给用户看的文字，都不能写死在代码里，要放到资源文件里去。为什么？因为翻译的人不该去改源代码，改源代码就要重新编译、重新测，一轮下来成本翻几倍。第三条，**使用 Header files 去定义经常被调用的代码段**——课件这里说的是把常量的定义集中起来管理，好处是一处改动全局生效，不会漏改。

第三组，界面层：**改善翻译文本尺寸，具有调整的灵活性**。这条特别容易被忽略。翻译后的文字长度和原文不一样，界面必须留出伸缩余地——按钮能变宽、标签能换行、对话框能长大。如果界面是按英文字符数精确排版写死的，德语一上就崩。还有一条，**国际化用户界面设计（自我定义）**，意思是界面设计本身要考虑不同文化的阅读习惯和审美，不要默认全世界都跟你一样。

第四组，语言习惯层：**支持各个国家的键盘设置**——键盘布局不一样，快捷键的可用字符就不一样；**支持文字排序和大小写转换**——不同语言的字母表顺序不同，排序规则也不同，有些语言压根没有大小写。这两条在测试里非常容易漏，因为中文环境下你根本感知不到。

八条合起来是一句话：**国际化就是把"跟地区有关的一切"从代码里赶出去，变成可配置、可替换的东西。**

> **板书：**四组纵列。数据层：Unicode/双字节｜度量衡・时区・货币可设。代码层：代码与显示分离｜消除硬编码｜头文件集中定义。界面层：文本尺寸可伸缩｜UI 设计国际化。语言习惯层：键盘设置｜排序与大小写转换。底部："把地区相关的一切赶出代码"。

> **提问：**八条里，哪一条最容易被团队"以为自己做到了"？

> **预设回答：**

- 答"消除硬编码"——很可能。界面上看得见的字符串容易抽出去，看不见的才要命：日志里的提示、异常信息、导出的文件名、报表表头。

- 答"文本尺寸可伸缩"——也非常典型。源语言下每个按钮都刚刚好，一翻译全挤在一起。

- 答"这些都做了，我们代码里没有中文"——这个回答本身就是陷阱。代码里没有中文，不等于没有硬编码的格式：`yyyy-MM-dd`、`$`、`kg`、`12/24 小时`，这些全是硬编码。

- 追问收口：所以检查硬编码，不能只搜中文，要搜**格式串和单位符号**。

> **易错点 / 考点：** ① 八条要能归组记忆（数据・代码・界面・语言习惯）；② 高频错点：把"消除硬编码"只理解成"界面文字"，漏掉日志、异常、文件名、格式串；③ 简答题若问"国际化包含哪些方面"，答这八条比答"翻译"得分高得多。

> **过渡：** 清单是抽象的，我们看一张真正体现国际化能力的实例。

【8.9】国际化功能实例（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是国际化能力的实例，也就是刚才那八条在真实界面上长什么样。你要练的是"**从现象倒推能力**"。

看第一类现象：**同一个数据，显示的样子随区域变**。比如一个数字，在一种区域设置下是 1,234.56，换一种设置可能变成 1.234,56——千分位和小数点的符号对调了。这说明程序没有自己拼字符串，而是调用了系统的区域格式。这就是"支持货币单位格式设置"落地的样子。

第二类现象：**日期时间的显示随区域变**，24 小时制和 12 小时制、星期几的写法、月份的名字。第三类现象：**排序结果随区域变**——同样一组人名，在两种语言下的先后顺序不一样，因为字母表顺序不同。第四类现象：**界面上不该出现的字符出现了**，比如本该显示一个字符的位置出现一个方框，这叫缺字形，属于字体或字符集问题。

怎么用这一页？我建议你把它当成一张"**验收照片**"：将来你测一个号称支持多语言的软件，就按这四类现象各挑一个点，切换区域设置，看它变不变。变了，说明能力在；不变，说明格式被写死了。

这里有个方法论要强调：**国际化测试最有力的手法就是"切换"**。切换区域、切换语言、切换时区、切换键盘，然后看同一个功能的表现是否随之改变。不切换，你测的只是源语言那一种情况。

> **板书：**四类现象竖排：数字格式随区域变｜日期时间随区域变｜排序随区域变｜缺字形出现方框。右侧写："手法：切换区域 / 语言 / 时区 / 键盘"。

提问：你把系统区域从中文切到德语，发现日期显示没变，这算缺陷吗？

预设回答：

- 答"算，说明格式写死了"——大概率算，但要先排除一种情况：这个字段可能是**业务要求固定格式**的，比如对外的接口报文。

- 答"不算，可能是缓存"——有这个可能，重启或清缓存再验一次，这是应有的严谨。

- 答"不算，日期格式无所谓"——错。日期格式在跨地区场景下会直接导致误读，前面 2/3/2009 那个例子已经说明后果。

- 收口：所以提这类缺陷时，缺陷单里要写清"**在什么区域设置下、期望什么格式、实际什么格式**"，三个要素缺一不可。

> **易错点 / 考点：**① "切换"是国际化测试的核心手法，要能说出切换哪四个维度；② 判断"显示不变"是不是缺陷，要先确认该字段有没有"固定格式"的业务要求；③ 缺字形（方框、豆腐块）与乱码是两种不同现象，考试常混。

过渡：i18n 的能力说完了，接下来看另一半——l10n 到底要做哪些事。

【8.10】L10N（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是本地化的工作内容，六项。注意它不是"翻译"一项，而是六项，这个数字本身就是考点。
>
> 第一项，**翻译**。这是主体工作量，包括界面文字、提示信息、错误信息、帮助文档。翻译要讲术语一致、语气统一，后面 8.2 会专门讲怎么验证。
>
> 第二项，**地区文化、宗教**。这一项最容易被工程团队忽略，也最容易出事。图标、颜色、手势、动物形象、节日、地图边界，在不同文化里的含义可能完全相反。有些内容在某个国家是禁忌，你不改，产品就上不了架，甚至引发公关危机。
>
> 第三项，**度量衡和时区等**。公里还是英里、摄氏度还是华氏度、千克还是磅；时区不只是显示偏移，还包括

夏令时——有些国家有、有些没有，切换的那一周特别容易出日期错乱。

第四项，**软件用户界面（UI）**。翻译完了要重新排版：控件变宽、对话框变高、菜单换行、图标里的文字要重画。这一项在 8.3.2 会展开。

第五项，**联机文档**，包括帮助文档和功能性的 PDF 文档。为什么强调"功能性 PDF"？因为有些软件的交付物本身就是 PDF——合同模板、报表模板、说明书，这些文件里的字体、编码、排版都要单独处理，而且往往不在研发的测试范围里，是最容易漏的一角。

第六项，**热键设置**。这一项很技术但很实际：菜单项的快捷键通常跟字母绑定，翻译之后字母变了，快捷键就得重新分配，还要避免和系统保留键冲突。中文用户习惯 Ctrl+C/V，德语用户键盘上 Z 和 Y 的位置都不一样，这些都要验。

六项记法：**翻译、文化、度量、界面、文档、热键**——前两个字能凑一句"翻文度界文热"，你按自己的方式记，关键是别只记翻译。

板书：六行竖排：① 翻译 ② 地区文化・宗教 ③ 度量衡・时区 ④ UI ⑤ 联机文档（帮助・功能性 PDF）⑥ 热键设置。右侧批注："只做第①项＝没做本地化"。

提问：六项里，哪一项最可能"研发完全不知道要做"？

预设回答：

- 答"文化宗教"——很对，这属于产品与市场范畴，研发通常不碰。

- 答"联机文档里的 PDF"——也很对，PDF 往往由文档团队产出，测试根本不在链条上。

- 答"热键不用改，Ctrl+S 全世界都一样"——这是常见错答。功能键确实通用，但**菜单助记符**（按 Alt 后出现的那个带下划线的字母）是按语言分配的，翻译后必须重排，否则两个菜单项可能抢同一个字母。

- 收口：这六项的价值就在于提醒你——**本地化的验收面，比"界面翻译"宽得多**。

易错点 / 考点：① 六项内容必背，"只做翻译"是典型错答；② 热键问题的准确说法是"菜单助记符需重新分配并避免冲突"，不是"功能键要改"；③ 常考判断："PDF 文档不属于软件本地化范围"——错，课件明确把它列进联机文档。

过渡：为什么本地化里编码问题总是排第一？我们下到最底层，看字符集。

【8.11】8.1.2 字符集问题（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲字符集。课件给的定义很短：**字符集是操作系统中所使用的字符映射表**。"映射表"三个字是全页的关键——它说的是"**字符和数字之间的对应关系**"。计算机里没有"字"，只有数字；你看到的每一个汉字，本质上是一个编号。字符集就是这本编号簿。

课件列了五种：**ASCII**、**GB2312**、**ISO8859-X**、**Unicode**，以及 Unicode 的三种具体编码 **UTF-8 / UTF-16 / UTF-32**。它们的关系不是并列的，是**演进的**：ASCII 只装英文，一个字节，128 个位置；GB2312 是中国自己的国标，为了解决汉字，用两个字节；ISO8859-X 是欧洲的一堆单字节字符集，一个语系一份；Unicode 想一统天下，给全世界所有字符编号；UTF-8/16/32 是 Unicode 的三种"存法"。

为什么必须讲这个？因为**乱码的根因永远是同一句话：写的时候用 A 号簿，读的时候用 B 号簿**。举个最常见的场景：一个 UTF-8 保存的中文文件，被一个按 GBK 解码的程序打开，每个汉字都被拆成两半重新解释，出来的就是一串谁也不认识的字。反过来也一样。这不是"字库坏了"，是**两边用的编号簿不一样**。

更麻烦的是它会产生"**半个字符**"的问题。双字节字符占两个字节，如果程序在截断字符串时按字节切，正好切在一个汉字的中间，剩下的那半个字节谁也解释不了——这就是为什么本地化测试里，"字段长度限制"是个高危测试点：英文字段限 20 个字符，中文可能只能放 10 个字，按字节算又会切坏。

所以后面 8.3.2 讲乱码、8.3.3 讲 Non-Unicode 路径，根子都在这一页。这一页听懂了，后面那些现象都是它的表现。

> **板书：**中间写"字符集＝字符↔数字的映射表"；下方一行演进：ASCII（1 字节・128 位）→ GB2312（2 字节・汉字）→ ISO8859-X（欧洲单字节）→ Unicode（全球编号）→ UTF-8/16/32（三种存法）。右侧红字："乱码＝写用 A 簿，读用 B 簿"。

提问：同一段中文，为什么在有的电脑上正常、在有的电脑上是乱码？

预设回答：

- 答"因为两台机器的系统语言不一样"——方向对，更准确的说法是**默认代码页或解码方式不一样**。

- 答"因为文件坏了"——错。文件一个字节都没变，变的是"怎么解释这些字节"。

- 答"装个字体就好了"——这是最典型的错答。字体只解决"有没有这个字形"，解决不了"这段字节代表哪个字"。乱码和缺字形是两件事。

- 追问收口：所以排乱码的顺序是——**先确认编码，再看字体，最后看翻译**。顺序反了，白折腾。

> **易错点 / 考点：**① 字符集的定义与五种类型的演进关系；② 乱码与缺字形（方框、豆腐块）必须分清：前者是解码错，后者是字体缺；③ 高频考点："为什么双字节字符会导致字符串截断出错"——因为按字节切可能切在字符中间。

> **【思政落点 1】**：字符集这一页最值得琢磨的地方是——**它出错的时候，程序一切正常**。没有异常、没有崩溃、日志也不报错，只是用户看到一堆乱码。这类问题考的是"**细节决定成败**"：一个编码参数、一个字节序、一次截断，看起来都是小事，可它直接决定了几亿用户能不能看懂你的界面。工程质量的差距，常常不在会不会写复杂算法，而在有没有人愿意为一个"看起来不影响运行"的字节较真。请大家养成这个习惯：**越是没人盯着的地方，越要自己盯。**

> **过渡：**知道了有哪几种字符集，接下来看国际上用什么标准把它们管起来。

【8.12】字符集问题标准（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页列了五个标准，看着像背编号，其实每一个都在回答一个具体问题。我把"它管什么"讲清楚，编号你眼熟就行，考试很少要求你默写编号，但要求你知道**有哪几类标准、各管什么**。
>
> 第一个，**ISO/IEC 10646-1:2003**，它定义的是四字节编码的通用字符集，英文缩写 UCS，也叫"通用多八位编码字符集"。它回答的问题是：**全世界所有字符，怎么放进同一套编号体系里**。四个字节，空间足够大，这也是 Unicode 的编号基础。
>
> 第二个，**ISO 639-1:2002**，两字母语种代码标准。中文是 zh，英文是 en，日文是 ja，德语是 de。它管的是"**这是哪种语言**"。
>
> 第三个，**ISO 3166-1:1997**，国家代码标准。中国是 CN，美国是 US，台湾地区是 TW。它管的是"**这是哪个地区**"。
>
> 为什么语言和地区要分开？因为同一种语言在不同地区规则不同。中文在简体环境里用"软件"，在繁体环境里用"軟體"；日期习惯、货币、度量都可能不一样。所以本地化标识通常是"**语言 + 地区**"两段拼起来，这样才够精确。

第四个，**RFC 3066**，语言鉴定标签标准。它管的是"**这个标签怎么写才算合法**"，也就是把前面两个标准拼起来的语法规则。你在程序里看到的那些语言标识，依据的就是这一类标签规范。

第五个，**RFC 3629**，UTF-8。它管的是"**Unicode 怎么用变长字节存下来**"，下一页会专门讲。

五个标准记法：**编号（10646）、语言（639）、地区（3166）、标签（3066）、编码（3629）**。这五个词就是本地化标识体系的五根柱子。

再补一句考试口径：这五个标准里，测试人员真正会用到的只有两个——**知道语言代码和地区代码怎么拼出本地化目标，知道 UTF-8 有正式规范可查**。其余三个，你能说出它管什么就够了，不必背全名。

板书：五行：ISO/IEC 10646-1:2003 → 通用字符集 UCS（4 字节编号）｜ISO 639-1:2002 → 语言代码（zh/en/ja/de）｜ISO 3166-1:1997 → 国家/地区代码（CN/US）｜RFC 3066 → 语言标签语法｜RFC 3629 → UTF-8。右侧一行："语言 + 地区 = 精确的本地化目标"。

提问：为什么本地化目标不能只写"中文"，要写成"中文简体"或"中文繁体"？

预设回答：

- 答"因为字不一样，一个简体一个繁体"——对，这是最直接的原因。

- 答"因为用词也不一样，比如软件和软体"——补得好，这就是术语层面的差异。

- 答"中文就是中文，一样的"——错。只写"中文"，翻译团队没法判断用哪套字形、哪套术语、哪套日期习惯，交付出来必然有一半用户不满意。

- 追问收口：同理，"英语"也要分英式美式——颜色是 colour 还是 color，日期顺序也不同。所以本地化目标的最小单位是"**语言 + 地区**"。

易错点 / 考点：① 五个标准各自"管什么"要能对上，别把 639（语言）和 3166（地区）搞反；② 课件给的是标准编号与版本年份，考试若考填空照抄，别自己改年份；③ 高频简答："为什么本地化标识要包含语言和地区两部分"。

过渡：标准讲完了，我们从最小的字符集看起——ASCII，所有编码故事的起点。

【8.13】示例：ASCII（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 ASCII 码表。这张表看着枯燥，但它是整个字符编码世界的起点，值得你花两分钟看懂它的结构。

ASCII 一共 128 个码位，用 7 个二进制位就能表示完，所以早期一个字节（8 位）的最高位是空着的——这个"空着的一位"后面会引出很多故事。表里的排列有规律，记住规律比背表有用：

前面一段是**控制字符**，编号最小的那批，它们不显示成字，而是控制行为——回车、换行、制表、响铃。你在程序里见的 \n、\t 就是它们。

中间是**数字**，从 0 到 9 是连续排列的，所以"字符 '7' 减字符 '0' 等于数字 7"这种写法才成立。

然后是**大写字母**，再是几个标点，然后是**小写字母**。大写和小写的编号差是固定的 32，这就是为什么大小写转换在英文里可以靠加减一个常数完成。

最后是英文标点符号。

看这张表要看出它的**局限**：128 个位置，装得下英文、数字、英文标点，**装不下任何一个汉字**。中文有几万个常用字，一个字节不够，必须两个字节。这就是 GB2312 出现的原因，也是"双字节字符"这个词的由来。

还有一条隐含信息：ASCII 里没有"€"、没有重音字母、没有阿拉伯文，欧洲人要写自己的语言就得扩展它——这就是 ISO8859-X 系列存在的原因。你把这条演进链记住：**一个字节不够 → 加字节 → 各国自己加 → 最后统一到 Unicode**。

> **板书：**画一条横轴：控制字符 → 数字（连续）→ 大写字母 → 标点 → 小写字母 → 标点。上方标注"128 个码位 / 7 位"。下方红字："装不下汉字 → 双字节的由来"。右侧写演进链："1 字节 → 2 字节 → 各国扩展 → Unicode"。

提问：既然 ASCII 只用 7 位，一个字节还剩 1 位，为什么当初不直接用它装更多字符？

预设回答：

- 答"因为当时没考虑别的语言"——对，这是历史事实层面的回答，标准是按英语世界的需求定的。

- 答"因为剩下那位要用来做校验"——这是真实存在的一种做法，早期通信确实用最高位做奇偶校验，所以不能随便拿来装字。

- 答"因为 128 个够用了"——在当时够用，但正是这个"够用"给后来的国际化埋了坑。

- 收口：这段历史给我们的启示很实际——**标准的边界，往往就是后来所有兼容性问题的源头**。

易错点 / 考点：① ASCII 是 7 位、128 个码位，别写成 8 位 256 个；② 大小写差 32、数字字符连续，这两个

规律常被用来出小题；③ 常考判断："ASCII 可以表示中文"——错，这是双字节讨论的起点。

过渡：ASCII 不够用，于是就有人想一次把全世界装进去——这就是 Unicode。

【8.14】示例：Unicode（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲 Unicode 的编号结构，课件给的信息不多，但每一条都是关键，我拆开讲。

第一，"**UCS-4，范围 0x0 到 0x10FFFF**"。这句话的意思是：Unicode 给每个字符一个编号，编号空间从 0 一直到十六进制的 10FFFF。为什么是这个范围？因为四个字节理论上能装到 FFFFFFFF，但没必要用那么多，划到这个上限足够覆盖现有和未来可预见的全部字符。

第二，"**1st 字节：0-16 个平面，后两个字节：0-FF 行，0-FF 列**"。这是一个坐标式的理解方式：整个编号空间被分成若干个"**平面**"，每个平面里有若干"**行**"，每行里有若干"**列**"。一个字符的位置就是"第几平面、第几行、第几列"。这种说法你听着像地图坐标，其实本质就是——**编号是分层的**。

第三，"**0 平面：BMP（Basic Multilingual Plane）＝ UCS-2**"。这是全页最重要的一句。第 0 个平面叫"基本多语言平面"，几乎所有日常用的字符都在这里：拉丁字母、汉字、日文假名、阿拉伯文、标点符号。所谓 UCS-2，就是"**用固定的两个字节表示这个平面里的字符**"。前面说的"双字节字符"，指的就是这一层。

由此可以推出一个很实用的结论：**绝大多数情况下，一个汉字占两个字节**（在 UCS-2 的世界里）；但在 UTF-8 里它占三个字节。同一句话，不同编码下字节数完全不同——这正是"按字节限制长度"会出事的原因。

再补一句：基本平面之外的字符（比如一些生僻字、古文字、表情符号）要用"代理对"来表示，也就是两个双字节拼起来。这类字符是本地化测试的高危区：长度算不对、截断截坏、排序乱掉，全从这儿来。

板书：画一个坐标示意：平面（0–16）× 行（00–FF）× 列（00–FF），范围标 0x0–0x10FFFF；在"平面 0"处写"BMP ＝ UCS-2（双字节区）"。下方一行："汉字：UCS-2 两字节 / UTF-8 三字节"。

提问：一个字段限长 10 个字节，能存几个汉字？

预设回答：

- 答"10 个汉字"——错，把"字节"当成"字符"了，这是最普遍的错误。

- 答"5 个汉字"——如果按 UCS-2 每字两字节算，对；但如果程序内部是 UTF-8，汉字占三个字节，就只能放 3 个。

- 答"要看编码方式"——最准确的回答。**先问编码，再算字数**，这才是专业反应。

- 追问收口：所以本地化测试遇到"长度限制"，第一件事是问清"**限的是字节还是字符**"，第二件事是拿最长的目标语言去试。

> **易错点 / 考点：**① BMP、UCS-2、"双字节"三者的关系要能串起来讲；② 高频判断："一个汉字就是一个字节"——错，这是本地化缺陷的高发根源；③ 编号范围 0x0–0x10FFFF 常作为填空点，注意是 16 个平面。

> **过渡：**Unicode 解决了"编号"，还没解决"怎么存"，这就是 UTF-8 要回答的问题。

【8.15】示例：UTF-8（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲 UTF-8，课件只给了一个 RFC 的链接，但内容必须讲清楚，因为它是今天互联网上最常用的编码。

UTF-8 的核心思想是"**变长**"：一个字符用几个字节，取决于它是哪个字符。英文的 26 个字母还是 1 个字节，和 ASCII 完全一样；带重音的欧洲字符 2 个字节；常用汉字 3 个字节；更生僻的、或者表情符号 4 个字节。

变长带来三个后果，每个后果都是一类测试点。

第一个后果，**向后兼容**：纯英文的文本，用 UTF-8 存和用 ASCII 存，字节完全一样。这就是为什么 UTF-8 能迅速取代各种本地编码——老系统不用改就能读。

第二个后果，**长度不再等于字符数**。这是最要命的一条。程序里所有"取前 N 个字符""截断到 N 字节""按长度校验"的地方，在 UTF-8 下都必须小心。截断如果正好切在一个三字节汉字的中间，剩下两个字节就是非法序列，显示出来就是乱码或者问号。

第三个后果，**解码可以自同步**：UTF-8 的设计让每个字节的高位能表明"我是单字节字符，还是多字节字符的第几个字节"。好处是即使数据流中间丢了一段，程序还能从下一个合法起点重新对齐，不会一路错到底。这个特性是它比很多老编码健壮的原因。

课件给的那个 RFC 链接，是 UTF-8 的规范原文。课上不要求你读，但你要知道一件事：**编码是有正式规范的，不是各家自己商量**。遇到编码争议，最后都要回到规范上说话。这也是一种职业习惯——**有标准的地方，就**

别靠猜。

> **板书：**画一张对照表：英文 1 字节｜欧洲带重音字符 2 字节｜常用汉字 3 字节｜生僻字/表情 4 字节。右侧三条：向后兼容 ASCII｜长度≠字符数｜解码自同步。底部："截断切在字符中间 = 乱码"。

提问：一个用 UTF-8 存的文件，被一个只认 ASCII 的老程序读，会发生什么？

预设回答：

- 答"中文会变乱码"——对，非 ASCII 部分全部读不出来。

- 答"英文部分正常，中文变乱码"——更准确。UTF-8 的 ASCII 段兼容，所以英文毫发无损，只有多字节部分出问题。

- 答"文件会打不开"——不一定，多数情况是能打开但内容错，这种"看起来打开了其实是错的"才是最危险的。

- 收口：这正是为什么本地化测试里要专门有一条"**在非 Unicode 环境下的表现**"，8.3.3 会讲。

易错点 / 考点：① UTF-8 是变长编码，1–4 字节；② "长度≠字符数"是本地化缺陷的高发点，务必能举例；③ 常考判断："UTF-8 是 Unicode 的另一种字符集"——不准确，Unicode 是字符集与编号，UTF-8 是它的**编码方式**之一。

过渡：底层讲完，我们把视角抬到设计层——一个软件要做到哪五条，才算"具备国际化标准"。

【8.16】8.1.3 软件国际化标准（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是国际化的五条设计要求。前面 8.1 那八条讲的是"能力清单"，这五条讲的是"**设计上的硬要求**"，两者角度不同：八条回答"要具备什么"，五条回答"接口做成什么样才算合格"。这五条我一条一条过，每一条我都会告诉你它对应的测试点。

第一条，**切换语言的机制**。软件要能换语言，而且换完不能要求重装、重启。这一条对应的测试点是：切换入口在哪儿、切完界面有没有立刻变、有没有一部分文字没跟着变（通常是缓存或者写死的地方）。

第二条，**与语言无关的输出接口**。意思是：程序往外吐的东西——日志、导出文件、报表、接口报文——格式不能依赖当前语言。为什么？因为一旦依赖，同一个系统在不同语言环境下产出的数据就不一致，下游系统解析不了。测试点是：同一份数据在两种语言环境下导出，结果该一样的必须一样。

第三条，**与语言无关的输入接口和标准的输入协议**。这条跟第二条对称。输入不能只认一种编码、一种格式。用户可能在中文输入法下敲全角符号，可能从别处粘贴一段 UTF-8 文本，程序都要能正确接收。测试点：全角半角混输、粘贴多语言文本、导入外来文件。

第四条，**资源文件的国际化**。所有要翻译的内容要放在资源文件里，而且要有一套管理机制——分文件、分模块、能增量更新、缺项能回退到源语言。测试点：某个资源项缺失时，界面是显示空白、显示键名、还是回退显示源语言？三种表现里，"显示键名"是最容易漏测的。

第五条，**支持和包容本地化数据格式**。这是数据层的总要求：数字、货币、日期、时间、度量衡，都要能按目标地区的习惯来，而不是硬编码成一种。

这五条合起来，就是"**界面可切、输出中立、输入宽容、资源可换、格式可配**"。你在评审一份国际化设计方案时，就拿这五句话去对。

> **板书：**五行：① 切换语言机制（不重装不重启）② 输出接口与语言无关 ③ 输入接口与语言无关・标准协议 ④ 资源文件国际化（缺失可回退）⑤ 包容本地格式。右侧竖排记忆句："界面可切・输出中立・输入宽容・资源可换・格式可配"。

提问：第四条"资源文件国际化"，如果某个词条忘了翻译，你希望界面显示什么？

预设回答：

- 答"显示英文原文"——这是比较常见的降级策略，用户好歹能猜。

- 答"显示空白"——最差的选择，用户看到一块空白，既不知道是什么也不知道出错了。

- 答"显示一个像 %s 或者 BTN_OK 的东西"——这是错答里最不理想的，用户完全看不懂；但对测试来说它反而**是好事**，因为它一眼就能被看出来，比"静默回退成英文"更容易发现漏译。

- 追问收口：所以验收时要定一个明确策略：**漏译时回退到源语言，并且让测试能定位到是哪一条漏了**。策略不定，测试就没法判对错。

易错点 / 考点：① 五条要求要能默写关键词；② 输入接口与输出接口是成对的两条，答题时别只写一半；③ 高频判断："国际化只是界面的事"——错，第二、三条明确要求输入输出接口与语言无关。

过渡：设计标准有了，接下来看测试怎么切分——i18n 测试和本地化测试，查的不是同一批东西。

【8.17】I18N 测试（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是本章结构最清楚的一页，它把"国际化支持"和"本地化测试"两列并排放，一共十四个问号。这一页你要带走的是**分工**：左列查能力，右列查填充。

先看左列，**国际化支持**，七个问号：多字节字符集和脚本支持吗？多字节字符的输入和显示正常吗？多字节的文件名、文件夹名及其处理对吗？区域设置生效吗？索引和排序正确吗？本地化操作系统上能跑吗？键盘支持吗？

这七个问题的共同特征是：**它们都在源语言版本上就能测**。不需要等翻译交付。比如"文件名支持多字节"，你拿一个中文文件名在英文系统上建一下就知道；"区域设置生效"，你切换区域看日期变不变就知道。这就是 i18n 测试的最大优势——**早，而且便宜**。

再看右列，**本地化测试**，七个问号：本地化资源齐全吗？区域设置有没有被硬编码？链接和串联的字符串有没有问题？资源文件里有没有夹着非本地化内容？文字有没有扩展余地？非层次图像（就是图片里画死的文字）里的文字处理了吗？其它组件呢？

右列的共同特征是：**必须等翻译交付之后才能测**，而且它的缺陷往往更"贵"——因为改一处要重新翻译、重新排版、重新回归。

这一页还特别标了一条：**Pseudo-translation（伪翻译），重要部分**。什么叫伪翻译？就是在翻译还没交付之前，先把资源文件里的英文**机械地替换成一段变长的、带特殊字符的假文本**，然后跑一遍界面。它能在几分钟内暴露三类问题：文字扩展装不下、硬编码没抽出来、编码处理有问题。成本几乎为零，收益极高。

最后课件给了个术语：**MBC = Multiple Byte Character，多字节字符集**。以后看到 MBC 你就想到"中文、日文、韩文这类占多个字节的字符"。

板书：左右两列。左"国际化支持（源语言版就能测）"：MBC 字符集/脚本｜MBC 输入显示｜MBC 文件（夹）｜区域设置｜索引排序｜本地化 OS｜键盘。右"本地化测试（翻译后测）"：本地化资源｜区域设置硬编码｜链接/串联字符串｜资源含非本地化内容｜文字扩展余地｜非层次图像文字｜其它组件。中间标红："Pseudo-translation ＝ 提前引爆"，底部注 "MBC = Multiple Byte Character"。

提问：为什么"链接/串联字符串"会成为一个单独的测试项？

预设回答：

- 答"因为句子拼起来语序会乱"——对。程序里常写"共 " + 数量 + " 条记录"，翻译成别的语言后语序可能整个反过来，拼出来就是病句。

- 答"因为翻译只翻了片段，前后接不上"——也对，这是另一种表现：句子被切成几段分别翻译，译者看不到上下文，译文风格和术语就不统一。

- 答"字符串拼接没什么问题，能显示就行"——错。这是本地化最典型的隐性缺陷，源语言下读着通顺，换语言就成了机器翻译腔甚至语义错误。

- 收口：所以正确做法是**用带占位符的完整句子**，让译者看到整句；测试时要专门去找界面上那些"由几段拼出来的话"。

> **易错点 / 考点：** ① 两列各七个问号要能说出大意，重点是分清"哪一列能在源语言版上测"；② 伪翻译（pseudo-translation）是本章的高频名词，要能说出它的作用和成本优势；③ MBC 的英文全称常考填空。

> **过渡：** 既然左边那一列能提前测，具体用什么方法测？下一页给了三种。

【8.18】国际化测试方法（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页给的是国际化测试的三种方法，很简短，但每一种都对应一个明确的场景，我补上"什么时候用、怎么用"。
>
> 第一种，**设计评审和代码审查**。这是静态方法，不运行程序。评什么？评设计里有没有把格式、语言、地区这些东西写死；审代码里有没有硬编码字符串、有没有按字节截断、有没有用系统默认编码去读写文件。为什么把它放第一位？因为这类问题**越早发现越便宜**。等到测试阶段发现"整个模块的日期格式都是拼字符串拼出来的"，改造成本可能是重写。
>
> 第二种，**针对源语言的功能测试，比如不同的区域设置、不同的时区显示**。注意这句话的用词——"针对源语言"。意思是：界面还是英文（或中文），但我把系统区域切到德国、把时区切到纽约，看功能还对不对。这是最划算的一类测试，因为它不需要任何翻译交付，却能暴露大量区域相关缺陷。时区那一项尤其值得测：跨时区、跨夏令时切换点、跨日期边界，最容易出"差一天"的 bug。
>
> 第三种，**针对伪翻译版本的测试**。伪翻译版本就是把资源文件里的文本换成变长的假文本，跑起来看界面。它专治三类病：布局被撑破、文字被截断、以及那些"没被抽进资源文件、所以没被伪翻译覆盖到"的硬编码

——后者尤其宝贵，因为**凡是没变成假文本的地方，就是硬编码的地方**。这一招我叫它"反向定位"。

三种方法的关系是：**评审管"设计上对不对"，源语言测试管"区域相关功能对不对"，伪翻译测试管"界面和资源处理对不对"**。三层都做完，国际化这块基本就干净了。

板书：三行：① 设计评审 + 代码审查（静态・最便宜）② 源语言功能测试（切区域/时区，不需翻译）③ 伪翻译版本测试（撑破・截断・反向定位硬编码）。右侧写："凡是没变成假文本的地方＝硬编码"。

提问：伪翻译测试为什么能"反向定位"硬编码？

预设回答：

- 答"因为假文本很显眼，一眼就能看出哪些没被替换"——对，这就是核心逻辑。界面上还留着正常英文的地方，就是没走资源文件的。

- 答"因为它会把所有文字变长，撑破的地方就是布局问题"——这是它的第二个作用，但不是"反向定位"。

- 答"伪翻译就是随便乱码一下，测不出什么"——错。伪翻译是**有规则地**替换：变长、加特殊字符、加边界标记，目的性很强，不是乱码。

- 收口：所以伪翻译既是"压力测试"（用变长文本压布局），也是"覆盖率探针"（找出漏抽的字符串）。

易错点 / 考点：① 三种方法要能按"静态/动态"分类；② 伪翻译的两大作用（压布局、找硬编码）是简答得分点；③ 常考判断："国际化测试必须等翻译完成才能做"——错，前两种方法都不需要翻译。

过渡：方法有了，具体查哪些点？下一页给了一张十项的清单。

【8.19】国际化测试点（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是国际化测试的检查清单，十项，最后还有个省略号——意思是还能往下加。我按"输入、显示、转换、交互"四组给你归，每一项都说一句"怎么测"。

输入组：**多字节和单字节文字的混合**——在一句话里同时敲中文、英文、数字、全角符号，看显示和存储对不对；**输入法编辑器（IME）**——中文输入法的候选框、联想、上屏，在某些界面里会出问题，比如输入框被候选框挡住、上屏后光标位置跳错。

显示组：**双向识别功能**——就是前面讲的 RTL，一段文字里混了从左到右和从右到左两种方向，程序要能正确判断每一段的走向；**大小写转换、换行**——有些语言没有大小写，转换函数遇到它们会不会出错；换行规则也不一样，中文可以在任意字之间断行，英文必须按词断，日文还有自己的一套禁则。

转换组：**硬编码**——找那些没抽进资源文件的文字和格式；**语言切换方式**——切换入口、切换后是否立即生效、是否有残留没变的文字。

交互组：**快捷组合键**——翻译后菜单助记符要重新分配，避免冲突；**纸张大小**——A4 和 Letter 尺寸不同，打印排版会错位，这一项在打印报表、发票、合同类软件里是必测；**电话号码**——位数、分组、区号、国际前缀，各国都不同，输入校验如果按中国手机号写死 11 位，出国就废。

最后一项 **词序问题**——主谓宾的顺序、形容词和名词的先后、数量和单位的组合方式，各国都不一样。程序如果靠拼接字符串来组句子，词序一定出问题。

这十项我给你一个记忆抓手：**"混、法、切、硬、换、键、纸、电、序"**——混合输入、输入法、切换、硬编码、大小写换行、快捷键、纸张、电话、词序，再加一个双向识别。你把这十项抄在测试计划的第一页，做国际化测试时逐项打勾。

> **板书：**四组纵列。输入：多字节单字节混排｜IME。显示：双向识别｜大小写・换行。转换：硬编码｜语言切换方式。交互：快捷组合键｜纸张大小｜电话号码｜词序。底部："十项逐项打勾，别凭印象"。

提问：电话号码为什么会被列进"国际化测试点"？它不就是一串数字吗？

预设回答：

- 答"因为各国位数不一样"——对，中国手机 11 位，美国 10 位，有些国家还带前导零。

- 答"因为格式不一样，有的加括号有的加横线"——也对，显示格式和校验规则都要跟着地区走。

- 答"电话号码是业务字段，跟本地化没关系"——错。凡是有"固定长度、固定格式、固定校验规则"的字段，都是本地化的高危点：邮政编码、身份证号、税号、车牌，全是同一类问题。

- 追问收口：所以判断一个字段要不要本地化，就问一句"**这个规则在别的国家还成立吗**"。

易错点 / 考点：① 十项要能归类记忆，考试常以多选题形式出现；② "纸张大小"最容易被忘，记住它属于打印排版类；③ 高频案例："输入框写死手机号 11 位"是本地化缺陷的经典题。

过渡：清单讲完了，我们看一组截图，把这些抽象的点落到眼睛里。

【8.20】示例（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是把上一页那十项落到具体界面上的示例。看图的方法我在第 3 页给过：**每看到一处异常，先判断它是哪一类问题，再判断它是显示层还是数据层**。这一页我带着你走一遍这个判断流程。

第一类，看**文字装不装得下**。界面上如果有文字被截断、有省略号、有半个字露在外面，这就是布局问题，属于 8.3.2 的内容。判断要点：是控件太小，还是文本太长？如果是控件按源语言长度写死的，那是设计问题，不是翻译问题。

第二类，看**字符显不显示得出来**。有方框、有问号、有一串怪字符，这是编码或字体问题。这里要分清：方框（豆腐块）通常是字体缺字形；成串的怪字符通常是解码用错了字符集。两者修法完全不同。

第三类，看**方向和对齐**。RTL 语言该右对齐的地方左对齐了、镜像没做、括号反了，都属于这一类。

第四类，看**语言混排**。中英文数字混在一起时，间距、基线、换行位置是否正常。这一条最考验实现质量，也最难自动检测。

我想强调一个职业习惯：**看截图不能只看"哪里不对"，要能说出"为什么不对、应该改成什么"**。缺陷单里写"界面显示异常"是没法处理的；写"德语环境下'保存'按钮文字超出边框约两个字符宽度，建议按钮宽度自适应或允许换行"，开发一看就知道怎么改。**缺陷描述的精度，就是测试人员的专业度。**

板书：四类判断：装不装得下（布局）｜显不显示得出（编码/字体）｜方向对齐（RTL）｜语言混排（间距基线换行）。右侧写缺陷描述三要素："现象 + 环境 + 建议"。

提问：截图里一个按钮上的字被切掉了，你怎么判断是翻译太长还是控件太小？

预设回答：

- 答"把语言切回源语言看是否正常，正常就是翻译太长"——这是最快的判断法，很实用。

- 答"看控件有没有固定宽度"——这是从实现角度的判断，更根本。

- 答"反正就是显示问题，让开发自己看"——错。不区分原因，开发可能只改翻译（缩短译文），而真正的问题是控件写死了宽度，换个更长的语言还会犯。

- 收口：所以好的缺陷单会写"**建议控件宽度自适应，而不是缩短译文**"——这就是测试人员给出的专业判断。

易错点 / 考点：① 四类问题的分类判断是案例题的基本功；② 缺陷描述"现象+环境+建议"三要素，考试若给

一段糟糕的缺陷描述让你改，就按这三要素补；③ 别把"字体缺字形"和"编码错误"混为一谈。

过渡：从这一页开始进入流程——本地化这件事，从头到尾分八步。

【8.21】8.1.4 软件本地化基本步骤（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是本地化的八步流程。这是本章唯一的完整流程，考试很可能让你排序或者补步骤，所以我按"准备、提取、翻译、回填、验证"五个阶段给你串一遍。

准备阶段，两步。第一步，**建立配置管理体系，跟踪目标语言各个版本的源代码**。为什么这一步排第一？因为本地化是"多语言、多版本并行"的活：中文版在改、英文版在改、德语版也在改，如果没有配置管理，你根本说不清"这个德语版是从哪个源版本翻出来的"。一旦源语言改了而译文没跟上，就会出现"新功能没翻译"的缺口。第二步，**创造和维护术语表**。术语表是本地化的命根子：同一个词在全产品里必须统一译法，否则用户在界面、帮助、文档里看到三种叫法，会以为是三个不同功能。

提取阶段：**源语言代码和资源文件分离、或提取需要本地化的文本**。这一步做得好不好，直接决定后面所有工作的成本。提取不干净，译者就会拿到一堆不该翻的东西（变量名、日志、内部代码），翻完了还会被误用。

翻译阶段：**把分离或提取的文本、图片等翻译成目标语言**。注意课件写的是"文本、图片等"——图片里的文字也要处理，这就是前面说的"非层次图像的文字"，往往要重画图。

回填阶段：**把翻译好的文本、图片重新检入目标语言的源代码版本**。这里的关键词是"检入"——走配置管理，不是手工粘贴。

验证阶段，三步：**如果需要，编译目标语言的源代码**；**测试翻译后的软件，调整 UI 以适应翻译后的文本**——注意这一步是"调 UI"，属于返工；**测试本地化后的软件，确保格式和内容都正确**——这一步才是完整的功能验证。

八步走完你会发现：真正"翻译"只占一步，其余七步都是**工程和管理**。这就是为什么本地化项目的成败，往往不取决于译者水平，而取决于流程。

板书：五阶段八步：准备（配置管理・术语表）→ 提取（代码与资源分离）→ 翻译（文本・图片）→ 回填

（检入源版本）→ 验证（编译・调 UI・测格式与内容）。右侧批注："翻译只占 1/8，其余是工程与管理"。

提问：为什么"术语表"要放在翻译之前，而且要求"维护"而不只是"创造"？

预设回答：

- 答"因为术语不统一，用户会以为功能不一样"——对，这是最直接的用户体验后果。

- 答"因为翻译公司可能换人，得有依据"——很实际。人员流动、外包更换，术语表是唯一能保证一致性的东西。

- 答"术语表是翻译的事，测试不用管"——错。测试恰恰要用术语表当判据：译文里出现的词，和术语表不一致，就是缺陷。

- 追问收口：所以测试团队要拿到术语表，把它变成一条可执行的检查规则。**没有术语表，翻译质量的判定就只能靠"感觉"**。

易错点 / 考点：① 八步的顺序与名称，尤其"调 UI"在"完整验证"之前，别颠倒；② 高频判断："本地化的主要工作是翻译"——错，八步里只有一步是翻译；③ 术语表的作用（一致性 + 可判定）是简答常见得分点。

【思政落点 2】：这八步里最不起眼的是第一步——配置管理。它不产出任何看得见的东西，却决定了后面七步有没有依据。现实中的本地化事故，很多不是译错了，而是"**译文对的，但对应的是上一个版本的界面**"。这类问题靠聪明解决不了，只能靠流程：谁改的、什么时候改的、改在哪个分支、对应哪一版译文，每一步都留痕。请记住：**严谨的流程意识，是让普通人也能稳定交付高质量的唯一办法**。做测试的人，尤其要尊重流程——因为流程保护的正是"不出事"这件事本身。

过渡：八步是"做事"的顺序，下一页把这些步骤放进整个产品生命周期里看。

【8.22】本地化过程（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把本地化放进产品的大流程里，五个环节加一条主线：**国际化软件设计 → 核心功能测试 → 国际化测试 → 翻译测试 → 本地化测试 → 全球化软件发布**。

注意顺序里的两个关键点。

第一，**国际化软件设计排在最前面**。这印证了前面那句话：i18n 属 Engineering，是设计阶段的事。设计没做，后面的测试只能发现缺陷，不能创造能力。

第二，**核心功能测试在国际化测试之前**。为什么？因为要先确认"这个软件在源语言下功能是好的"。如果功能本身就有问题，你去测国际化，测出来的失败你分不清是功能缺陷还是本地化缺陷。这就是测试里最基本的一条纪律：**先确认基线，再测增量**。

然后是**国际化测试**——查能力，能不能被本地化。接着是**翻译测试**——检查译文本身：完整性、术语、语境。最后是**本地化测试**——把翻译好的版本当完整软件测一遍，包括安装、升级、打印、帮助文档。

这条链子最后落在**全球化软件发布**上。注意"发布"也是一个环节而不是终点：多语言版本要一起发、版本号要对齐、语言包要能独立更新。

我给你一句总结：**这条链子上，越靠前的环节改起来越便宜，越靠后的越贵。** 设计阶段改一行配置，成本是一；发布之后发现方向错了，成本是重做。这就是为什么本章要花那么多篇幅讲 i18n 的设计标准。

还有一个现实要提醒你：这条链子上的每个环节都会「回流」。翻译测试发现译文和界面对不上，要退回到回填；本地化测试发现某个控件装不下，要退回到调 UI 再重新构建。所以真实的本地化项目不是一条直线，而是一条带着回环的螺旋。你将来写计划，一定要给返工留出时间，别把排期排成直线。

板书：一条横向流程：国际化软件设计 → 核心功能测试 → 国际化测试 → 翻译测试 → 本地化测试 → 全球化软件发布。流程下方画一个从低到高的成本箭头："越靠后越贵"。

提问：为什么"翻译测试"要单独成一个环节，而不是合到"本地化测试"里？

预设回答：

- 答"因为翻译问题太多，需要单独集中查"——有道理，译文问题量大且集中在语言层面。

- 答"因为做的人不一样，翻译测试常由语言专家做"——很准确，这是组织上的原因，也呼应了"l10n 可第三方完成"。

- 答"没必要分开，一起测就行"——在很小的项目里可以合并，但一旦多语言并行，混在一起会导致缺陷归属不清：到底是译文错，还是程序把译文显示错了？

- 收口：分开的目的，是让**缺陷归因**变得干脆：语言问题给译者，显示问题给开发。

易错点 / 考点：① 六个环节的顺序必背；② "核心功能测试在本地化之前"这条纪律常考；③ 高频判断："本地化测试可以替代翻译测试"——错，两者的判据和责任人不同。

过渡：链条走完了，我们回头看 8.1 的最后一小节：本地化测试到底包含哪六类工作。

【8.23】8.1.5 软件本地化测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本地化测试的六类工作，也是本章最可能出简答题的地方。我一条一条讲，每条都给你一个"具体查什么"的例子。

第一，**功能性测试**，包括所有基本功能、安装、升级等测试。这一条最容易被跳过——很多人以为"功能在原版测过了，本地化版不用重测"。错。本地化版是**重新编译过的另一个产品**：资源文件换了、有时候代码也动了，安装包不一样、升级路径不一样。安装和升级尤其要测：语言包能不能装、装错了语言能不能切、升级之后译文会不会被覆盖回英文。

第二，**翻译测试**，包括语言完整性、术语准确性等的检查。完整性指有没有漏译、有没有多余的英文残留；准确性指术语是否统一、语气是否合适、有没有机翻痕迹。

第三，**可用性测试**，包括用户界面、度量衡和时区等。注意课件把"度量衡和时区"放在可用性里——这不是笔误。对用户来说，看到"3 磅"还是"1.4 公斤"，是好不好用的问题；看到时间差 8 小时，也是好不好用的问题。

第四，**兼容性调试**，包括硬件兼容性、版本兼容性等。这里的关键词是"调试"——本地化环境下的兼容问题往往需要反复试：不同版本的操作系统、不同的语言包、不同的输入法、不同的打印机驱动。

第五，**文化、宗教、喜好等适用性测试**。这一条最软，但杀伤力最大：地图边界、宗教符号、颜色寓意、手势、节日、历史表述。这类问题一旦出错，不是 bug 而是事故，可能直接导致产品下架。

第六，**手册验证**，包括联机文件、在线帮助、PDF 文件等测试。文档里的截图是不是旧版本、链接能不能跳、索引能不能搜、PDF 里的字体有没有嵌进去，都要验。

六类记法：**功能、翻译、可用、兼容、文化、手册**。前四类是技术活，后两类是判断题——尤其第五类，考的是你的见识和分寸。

板书：六行：① 功能性（基本功能・安装・升级）② 翻译（完整性・术语准确性）③ 可用性（UI・度量衡・时区）④ 兼容性（硬件・版本）⑤ 文化宗教喜好 ⑥ 手册（联机文件・在线帮助・PDF）。右侧写："本地化版是重新编译的另一个产品"。

提问：为什么"安装和升级"必须重测？本地化版和原版在安装上有什么不同？

预设回答：

- 答"因为安装包不一样，里面多了语言文件"——对，这是最直接的原因。

- 答"因为要测语言选择、语言切换"——也对，安装向导本身也要翻译，而且要在安装时就能选语言。

- 答"安装程序没改，不用测"——错。安装向导里的文字也要翻译；安装路径里可能有非 ASCII 字符；升级时旧版本的配置会不会覆盖新译文，这些全是新风险。

- 追问收口：所以本地化测试的第一条用例，往往是"**在一台目标语言的干净机器上，从零装一遍，再升级一遍**"。

易错点 / 考点：① 六类工作必背，顺序不必死记但内容不能漏；② "度量衡和时区"归在可用性里，这是课件的归类，答题时按课件口径写；③ 高频判断："功能在原版测过，本地化版可以免测"——错，本地化版是另一个构建产物。

过渡：六类是"测什么"，接下来看"怎么组织"，业界有两份公开的本地化测试计划可以参照。

【8.24】软件本地化测试（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页把本地化测试再细分成六块，并且给了两份业界公开的测试计划作参照。先看这六块，它比上一页的六类更贴近"执行分工"。

I18n 测试——查国际化能力，源语言版就能做。**L10n 测试**——查本地化版本本身。**语言/翻译的测试**——纯语言层面：漏译、错译、术语、语气。**美观/界面测试**——排版、对齐、截断、控件布局。**功能测试**——业务功能在本地化环境下还对不对。**发布测试**——打包、安装、升级、许可、版本号、语言包分发。

这六块的划分很有意思：它把"语言"和"美观"分开，把"功能"和"发布"分开。为什么？因为**责任人和修法不同**：语言问题返给译者，美观问题返给 UI/前端，功能问题返给开发，发布问题返给构建和运维。**缺陷分类分对了，流转就快；分错了，来回踢皮球。**

课件给了两个参考链接：一个是 Mozilla 的 Firefox 2.0 L10n 测试计划，一个是 OpenOffice 的本地化测试计划。这两个都是开源项目公开的文档，价值在于——你能看到**真实的、工业级的本地化测试计划长什么样**：有哪些测试项、怎么组织多语言、怎么记录每种语言的状态。

我建议你课后真的去翻一翻。看别人成熟的测试计划，比看十页理论都快。看的时候带着三个问题：**它列了**

哪些测试项？每种语言怎么标记完成度？缺陷怎么归类？ 这三问的答案，就是你自己写计划时的模板。

还有一个观察：这类公开计划里通常有一张「语言状态表」，把每种语言标成未开始、翻译中、待验证、已通过。这张表是本地化项目管理的核心工具，因为多语言并行时，你随时得知道哪一种语言卡在哪一步。没有这张表，项目经理只能凭感觉汇报进度。

板书：六块：I18n 测试｜L10n 测试｜语言/翻译测试｜美观/界面测试｜功能测试｜发布测试。右侧标注责任人：译者｜UI/前端｜开发｜构建运维。底部写参考："Mozilla Firefox L10n Test Plan｜OpenOffice L10n Test Plan"。

提问：一个界面上"德语文案读起来别扭"和"德语按钮文字被截断"，这两条缺陷应该分到哪一块？

预设回答：

- 答"都是翻译问题"——错。前者是语言问题，后者是布局问题，修的人和方法完全不同。

- 答"别扭的给译者，截断的给前端"——完全正确，这就是分类的意义。

- 答"都提给开发，让他们判断"——可以，但效率最低。测试的分类能力，直接决定缺陷流转的速度。

- 收口：所以这一页真正的考点不是六个名词，而是"**分类即分派**"这个意识。

易错点 / 考点：① 六块与上一页六类的区别：这页按"测试类型/责任人"分，上页按"测试内容"分；② 两个参考计划的出处要能说出（Mozilla、OpenOffice）；③ 案例题常给一条缺陷让你分类，注意"语言"与"美观"的界线。

过渡：六块里，前两块最容易混——i18n 测试和 l10n 测试到底差在哪？下一页专门做对比。

【8.25】I18N 测试 vs. L10N 测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是对比页，把两个测试放在一起看。我给你一张"五维对照表"，这五个维度就是考试问"两者区别"时的答题骨架。

第一维，测的对象。i18n 测的是"**能力**"——这个软件有没有被本地化的资格；l10n 测的是"**成品**"——这个特定语言的版本做得好不好。

第二维，什么时候做。i18n 早，源语言版就能做，甚至在设计评审阶段就开始；l10n 晚，必须等翻译交付、重新构建之后。

第三维，语言环境。i18n 通常在**单语言环境**下做——界面还是源语言，只是切换区域、时区、键盘；l10n 必须在**目标语言环境**下做——装目标语言的操作系统、目标语言的输入法，最好在目标国家实测。

第四维，缺陷归属。i18n 的缺陷基本都是**开发**的：硬编码、按字节截断、默认编码写死；l10n 的缺陷可能属于译者（错译、术语不一致）、可能属于 UI（截断、重叠）、也可能属于开发（资源没加载）。**归属不同，返工成本差一个数量级。**

第五维，复用性。i18n 测一次，对所有语言都有效；l10n 每个语言都要测一遍，而且语言越多，总工作量线性增长。

这五维里最关键的是第二维。因为**越早发现问题越便宜**，这是软件工程里唯一一条几乎无例外的规律。所以业界的推荐做法是：把 i18n 测试做扎实，让 l10n 测试专注于语言和排版——前者像体检，后者像装修验收。

最后提醒一句：两者不能互相替代。i18n 全过，l10n 照样可能因为译文质量差而不合格；l10n 全过，也不证明这个软件能顺利支持第十种语言。

板书：五维对照表（两列）：对象（能力/成品）｜时机（早/晚）｜语言环境（单语切区域/目标语言环境）｜缺陷归属（开发/译者・UI・开发）｜复用性（一次通吃/每语言一遍）。底部："i18n 像体检，l10n 像装修验收；不能互相替代"。

提问：一个软件通过了全部 i18n 测试，能不能说它"本地化没问题"？

预设回答：

- 答"能，国际化都过了说明底子好"——错。底子好只说明"能装"，不说明"装得对"。译文质量、排版效果都不在 i18n 的检查范围里。

- 答"不能，还得看具体语言的译文和排版"——对，这正是 l10n 测试的职责。

- 答"不能，还得看目标国家的文化禁忌"——补得好，这属于 l10n 的文化适用性。

- 追问收口：一句话记住——**i18n 决定"能不能"，l10n 决定"好不好"**。

易错点 / 考点：① 五维对照是简答题的标准骨架；② 高频判断："i18n 测试通过即代表本地化质量合格"——错；③ "单语言环境 vs 多语言环境"这一维最容易被漏，答题时补上能加分。

> **过渡：**8.1 到此结束。进入 8.2，先看翻译验证要验哪三样东西。

【8.26】8.2 翻译验证（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

8.2 讲翻译验证。这一节只有三项，但每一项都值得认真对待，因为翻译是本地化里唯一"用户每天都在读"的产出。

第一项，**翻译的内容**。这是基本盘：有没有漏译（源语言有、目标语言没有）、有没有错译（意思歪了）、有没有多译（源语言删了、译文还留着）、术语是否统一、数字和变量占位符有没有被译坏。最后这一条特别技术：句子里如果有个 %s 或者 {0}，译者一动它，程序运行时就会崩或者显示错位。

第二项，**目标语言的文化心理**。这一项最微妙。同一句话，语法全对，但读起来"不像人话"，或者语气不对——比如把面向消费者的轻松语气译成了公文腔，或者把正式场合的敬语译成了口语。还有一种情况是"**政治正确与敏感度**"：某些表达在某个文化里带有冒犯意味，虽然字面没错。这种问题机器查不出来，只能靠目标语言的母语者读。

第三项，**特殊符号**。引号（中文的""和英文的""）、破折号、省略号、货币符号、度量单位符号、数学符号、版权符号。这些符号在不同语言里的形态和用法不一样，翻译时容易原样照抄，结果在目标语言里显得很别扭，甚至显示不出来。

三项合起来是一句话：**翻译验证不是"查错别字"，是判断"这段文字在这个文化里，像不像一个本地人写的"**。这也是为什么本地化行业的黄金标准是"**由目标语言的母语者做审校**"，而不是靠双语词典对照。

> **板书：**三项：① 翻译的内容（漏译・错译・多译・术语・占位符）② 目标语言的文化心理（语气・敬语・敏感度）③ 特殊符号（引号・破折号・货币・单位）。底部："像不像本地人写的"。

> **提问：**译文语法全对、意思也对，但用户说"读起来很别扭"，这算缺陷吗？

> **预设回答：**

- 答"算，属于可读性问题"——算，这类缺陷在本地化项目里很常见，通常定为中低级但必须改。

- 答"不算，意思对就行"——这是典型错答。软件是给人用的，"读着别扭"直接影响信任感，用户会怀疑这软件是不是山寨的。

- 答"要看是哪类产品，正式文档要求高一些"——很专业的补充，产品定位决定语言标准的严格程度。

- 收口：所以验收标准要提前写清楚：是"能看懂"还是"读起来自然"，这两档的返工量差很多。

易错点 / 考点：① 三项内容必背；② "占位符被译坏"是技术性极强的考点，要能举 %s、{0} 这类例子；③ 高频判断："翻译验证可以由不懂目标语言的人用工具完成"——只能完成一部分（漏译、术语一致性），语气和文化心理必须母语者判断。

【思政落点 3】：翻译这一项最考验"**对用户负责**"这四个字。一段机翻痕迹很重的界面，功能一个不少，可用户第一眼就给它贴了个"不用心"的标签。反过来，一个术语从头到尾统一、语气从头到尾一致的软件，用户会觉得"这家公司是认真的"。做技术的人容易有个错觉：语言是"软"的部分，不如代码重要。可用户接触产品的**唯一通道就是这些文字**。请记住：**你交付的每一个字，都是产品的一部分；对文字敷衍，就是对用户敷衍。**

过渡：翻译只是本地化的一半，另一半是"翻译之外的改动"。下一页看这两层分别改什么。

【8.27】软件本地化与翻译（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把本地化拆成两个层面：**技术层面的更改**和**文化层面的更改**。这个二分法是本章的骨架之一，答题时非常好用。

先说**技术层面**，五项：**调整大小**——控件、对话框、列宽要能容纳译文；**调整默认设置**——默认语言、默认区域、默认纸张、默认货币，都要按目标地区设；**重新编译**——本地化版是重新构建的产物，这句话我重复第三遍了，因为它太重要；**创建新的图形**——图片里带文字的、图标带英文缩写的，都要重做；**重新编排文档格式**——帮助文档、PDF 的页面尺寸、字体、目录、索引都要重排。

这五项的共同点：**都是工程活，都可以量化验收**。控件宽度是不是够、默认值对不对、编译有没有报错、图片文字有没有换、文档页数有没有错乱，一条条都能写成用例。

再说**文化层面**，课件列了：**包装、图标、宣传、样品、政治敏感的术语、地方规章和宗教信仰**。这一层的共同点是：**没有对错，只有合适不合适**。同一个手势，在有些国家是"OK"，在有些国家是侮辱；同一个颜色，在有些文化里代表喜庆，在另一些文化里代表丧事；地图上的边界画法，在有些国家是法律问题。

为什么要把这两层分开？因为**处理方式完全不同**：技术层靠测试和返工，文化层靠**评审和本地顾问**。技术缺陷可以靠用例抓，文化问题必须靠"懂当地的人"提前审。你让一个不懂当地文化的测试团队去测"这个图标在当地合不合适"，他们测不出来——不是不专业，是信息不够。

所以成熟的做法是：技术层由测试团队兜住，文化层建立一道"**本地审校 + 合规评审**"的关卡，两道关都过，才允许发布。

板书：左右两栏。左"技术层面（可量化、靠测试）"：调整大小｜调整默认设置｜重新编译｜创建新图形｜重排文档格式。右"文化层面（靠评审、靠顾问）"：包装｜图标｜宣传｜样品｜政治敏感术语｜地方规章与宗教信仰。底部："技术靠用例，文化靠人"。

提问：一个"竖起大拇指"的图标，作为"点赞"功能放进产品，需要本地化评审吗？

预设回答：

- 答"需要，有些地区这个手势是冒犯"——对，这就是文化层面的典型例子。
- 答"不需要，这就是个通用图标"——这是最常见的错答。正是这种"想当然的通用"，造成了大量出海事故。
- 答"看目标市场，如果只发国内就无所谓"——合理，文化评审的范围应该跟发布范围挂钩，这是成本意识。
- 追问收口：所以文化层面的检查项，要**按目标市场逐条过**，而不是全球一套标准。

易错点 / 考点：① 两层各自的五项/六项要能分开举例，别把"重新编译"写进文化层；② 高频判断："文化适用性问题是缺陷"——更准确的说法是"适用性/合规问题"，处理路径与功能缺陷不同；③ 简答常问"本地化为什么不能等同于翻译"，就用这两层来答。

【思政落点 4】：文化层面这一栏，考的是"**尊重差异、守住合规**"。有些内容在我们看来稀松平常，在别人的文化里却是禁区；有些表述在国内没问题，到了别的国家就触碰法律。这不是"要不要迁就别人"的问题，而是两条底线：**尊重，是跨文化合作的基本教养；合规，是产品能不能在当地合法存在的前提。** 做技术的同学将来很可能参与出海项目，请把这两句话记住：不确定的地方，去问当地的人；拿不准的内容，按最严格的标准处理。**敬畏规则，才走得更远。**

过渡：技术层面第一件要改的就是"大小"，因为译文一长，界面就装不下。下一页专门看文字扩展。

【8.28】翻译问题 – 文字扩展（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲**文字扩展**，也就是译文比原文长导致的界面问题。这是本地化测试里出现频率最高、最容易被用户看到的一类缺陷。

先讲为什么会扩展。不同语言的"信息密度"不一样：同一个意思，用不同语言写出来，字符数可以差很多。前面 8.3.2 会提到一句课件原话——**德语最长，汉语比较精炼**。这不是印象，是结构差异：德语喜欢把词组合成很长的复合词，一个词能顶中文一句话；中文是单字表意，同样的信息用更少的字符就能承载。反过来，如果把中文译成英文，多数情况下也会变长，因为英文需要更多的词和冠词介词。

扩展会打到哪些地方？第一，**按钮和标签**——宽度是设计时按源语言定的，一长就截断或者被挤成两行，两行又撑高工具栏。第二，**表格列**——列宽固定，长文字要么截断要么把整张表撑到屏幕外。第三，**对话框和提示**——文字超出边界、按钮被挤出可视区，用户点不到"确定"。第四，**菜单**——菜单项换行、子菜单跑到屏幕外。第五，**帮助文档和错误信息**——段落长度变了，页码、索引、交叉引用全要重排。

测试这一类问题的方法很简单，就两个字：**用最长的语言去试**。业界的做法是先做伪翻译（把文本机械拉长），把布局问题在翻译交付前就炸出来；然后在真实语言里重点测德语、俄语这类"长"的语言。

还有一条更根本的建议：**布局不要按字符数写死**。控件要能自适应、能换行、能滚动；设计时就要给译文留出余量。这一条属于 i18n 的设计标准，也是前面 8.1 第八条"改善翻译文本尺寸，具有调整的灵活性"的落地。

> **板书：**一句原理："德语最长，汉语精炼"；五个受灾点：按钮标签｜表格列｜对话框提示｜菜单｜文档与错误信息。底部两行："用最长的语言去试"｜"布局不按字符数写死"。

提问：界面上一个按钮在德语下文字被切成两行，把按钮撑高了，导致整个工具栏错位，这条缺陷该提给谁？

预设回答：

- 答"提给译者，让他把德语缩短"——有时可行，但治标：换成俄语还会犯，而且为了排版牺牲译文准确性是本末倒置。

- 答"提给前端，让按钮宽度自适应或允许换行"——这是治本的方向。

- 答"提给 UI 设计，因为设计时没留余量"——最根本的一层，也是最有价值的判断。

- 追问收口：所以合理的缺陷单会写"**建议工具栏按钮宽度自适应；短期可由译者提供简短译法作为缓解**"——治

本加治标，两条都给。

> **易错点 / 考点：** ① "德语最长、汉语最精炼"是课件原话，考试可直接引用；② 文字扩展的五个受灾点是案例题素材；③ 高频判断："译文太长就让译者改短"——只在极少数场景可接受，根本解法是布局自适应。

> **过渡：** 8.2 讲完，进入 8.3——本地化测试的技术问题，三块，第一大块是数据格式。

【8.29】8.3 软件本地化测试技术（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

8.3 是本章最"能直接用"的一节，三块内容，我先把地图给你。

8.3.1 数据格式——六项：数字、货币、时间、日期格式、度量衡单位、复数问题。这一块的特点是**全部可以用"对照"来测**：同一份数据，在两种区域设置下各显示一次，比对是否符合该地区的习惯。

8.3.2 页面显示和布局——三件事：译文长度差异（德语最长、汉语精炼）、乱码（双字节语言 GB/BIG5/JP）、字符索引与排序。这一块的特点是**看着测**：截图、比对、走查。

8.3.3 配置和兼容性问题——键盘布局、打印机配置、硬件与版本兼容、数据库问题、热键。这一块的特点是**搭环境测**：装目标语言的系统、换键盘、插打印机。

三块合起来，正好对应本地化缺陷的三种"发现方式"：**对照数据、看界面、搭环境**。你以后拿到一个本地化版本的测试任务，就按这三个动作依次做：先对照数据格式，再走查界面布局，最后在目标环境里跑一遍兼容性。

一句话总结这一节的价值：**8.1 告诉你"为什么"，8.3 告诉你"查什么"，8.4 告诉你"怎么组织"。**

> **板书：** 三块：8.3.1 数据格式（数字・货币・时间・日期・度量衡・复数）→ 8.3.2 显示与布局（长度・乱码・索引排序）→ 8.3.3 配置与兼容（键盘・打印机・硬件版本・数据库・热键）。右侧写三种发现方式："对照数据｜看界面｜搭环境"。

> **提问：** 这三块里，哪一块最可能靠自动化脚本覆盖？

> **预设回答：**

- 答"数据格式，因为规则明确"——对，格式可以写断言，输入一组区域设置，比对输出字符串。

- 答"乱码，可以扫"——部分可以，比如扫资源文件里的非法字节序列，但界面渲染出来的乱码要靠截图比对。

- 答"都能自动化"——过于乐观。布局重叠、文字被遮挡、方向错误，这些机器很难判"好不好看"。

- 收口：所以本地化测试的现实是：**格式类尽量自动化，界面类靠人眼和截图比对**。

> **易错点 / 考点：** ① 三块的名称与内容不能混，尤其"字符索引排序"属于 8.3.2 而不是 8.3.1；② 三种发现方式（对照/看/搭环境）是很好用的记忆抓手；③ 简答若问"本地化测试的技术问题有哪些"，就答这三块。

> **过渡：** 进入 8.3.1 之前，先看一个几乎每个本地化项目都会踩的坑——姓名。

【8.30】最常见的问题（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页拿姓名开刀，因为姓名是"最常见的问题"，也是最能说明本地化思维的一页。

课件给的对照很简洁：中文的"周宇文"、"朱先生"；英文的"Shaomin Zhu"、"Mr. Zhu"。这一组对照里藏着四层差异，我逐层说。

第一层，顺序。中文是姓在前、名在后；英文是名在前、姓在后。所以"周宇文"转成英文写成 Shaomin Zhu 的时候，两个部分的位置换了个个儿。程序如果按"第一个词是姓"去解析，遇到英文名就全错。

第二层，称谓的位置。中文说"朱先生"，称谓在姓**后面**；英文说 Mr. Zhu，称谓在姓**前面**。如果程序用模板拼"某某先生"，换到英文就拼成了"Mr. 放在后面"，非常别扭。

第三层，称谓的选择。中文的"先生/女士"和英文的 Mr./Ms./Mrs. 不是一一对应：英文里 Mrs. 还带婚姻状况信息，而现代用法里很多人更愿意用 Ms.。这一层涉及文化习惯，靠规则表解决不了，只能靠本地化规范。

第四层，也是最要命的一层：排序和检索。通讯录按什么排？中文按拼音排、按笔画排、还是按 Unicode 编码排？英文按姓氏的字母序排。同一个"周"，拼音是 Z，在字母表最后；如果按编码排，位置又完全不同。**用户找一个人的时候，排序错了等于功能不可用。**

所以姓名这个字段，看着是"两个文本框"，实际上牵动四件事：**显示顺序、称谓模板、音译一致性、排序规则**。这就是本地化测试的思维方式——**一个普通字段，也要问一遍"这个规则在别的国家还成立吗"**。

顺便说一句，课件为什么把它叫「最常见的问题」：因为姓名几乎出现在每个系统的第一屏——注册、登录、

通讯录、订单、发票、报表。它被使用的次数最多，出错的机会自然也最多。你将来做本地化走查，从姓名入手是性价比最高的起点。

板书：四层：① 顺序（姓+名 / 名+姓）② 称谓位置（朱先生 / Mr. Zhu）③ 称谓选择（先生女士 / Mr. Ms. Mrs.）④ 排序与检索（拼音・笔画・编码 / 字母序）。底部："一个字段，四种规则"。

提问：通讯录里"周宇文"和"Shaomin Zhu"是同一个人，系统该怎么存才不会乱？

预设回答：

- 答"存成一个字符串就行，显示的时候再排"——错，一个字符串没法可靠地拆出姓和名，排序和称谓全做不了。

- 答"分成姓、名、称谓、显示名四个字段"——这是比较标准的做法，显示名单独存，避免每次拼。

- 答"存拼音就够了"——不行，中文用户在界面上要看到汉字，拼音只是排序和检索用的辅助。

- 追问收口：所以姓名设计至少要有"**结构化字段 + 本地显示名 + 排序键**"三部分，这是很多系统吃亏之后才补上的。

易错点 / 考点：① 姓名的四层差异要能说全，只答"顺序不同"是最常见的失分点；② 高频判断："姓名就是一个文本字段"——错，它牵动显示、称谓、音译、排序；③ 排序规则（拼音/笔画/编码）是常考细节。

过渡：姓名只是一个字段，8.3.1 要把所有"格式类"字段系统过一遍，六项。

【8.31】8.3.1 数据格式（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是数据格式的六项清单：数字、货币、时间、日期格式、度量衡的单位、复数问题。这六项是本地化测试的"基础套餐"，我一项一项给你判据。

第一，数字。两个点：千分位用什么符号（逗号、空格、点、还是不用），小数点用什么符号（点还是逗号）。这两个符号在有些地区是**互换**的，所以同一串字符在不同地区读出来的数值差一千倍。

第二，货币。符号是什么（¥、$、€）、符号放在数字前面还是后面、小数保留几位、负数怎么表示（减号、括号、还是红色）。还有汇率和金额精度——金额的舍入规则如果按源语言写死，财务报表会对不上。

第三，时间。12 小时制还是 24 小时制、上午下午怎么标、时分隔符是冒号还是点。还有时区：显示本地时

间还是服务器时间，跨时区跨夏令时的时候日期会不会跳。

第四，日期格式。年月日的顺序、分隔符、月份是数字还是名称、星期从哪天开始。这就是第 2 页那个 2/3/2009 的问题。

第五，度量衡的单位。长度、重量、面积、温度、速度、纸张尺寸。公里/英里、摄氏度/华氏度、千克/磅、A4/Letter。这一项在工业软件、物流、医疗类系统里是硬需求。

第六，复数问题。这一项最容易被忽略，也最能体现语言差异。英语只有单数和复数两态（one apple / two apples）；有些语言有多达好几种复数形式，不同数量用不同词形；中文、日文基本不分单复数。程序如果硬编码"数量 > 1 就加 s"，在那些语言里就错了。

六项的测试手法是统一的：**准备一组"边界数据"，在两种以上的区域设置下各跑一遍，逐项对照是否符合当地习惯**。边界数据要包括：负数、零、很大很小的数、带小数的金额、跨年的日期、夏令时切换点。

> **板书：**六行：① 数字（千分位・小数点）② 货币（符号・位置・小数位・负数）③ 时间（12/24・时区・夏令时）④ 日期（顺序・分隔符・月份・起始星期）⑤ 度量衡（长度・重量・温度・纸张）⑥ 复数（单复数・多复数形态）。底部："边界数据 + 两种区域设置对照"。

提问：为什么"复数问题"会被单列成一项？它不是语法问题吗？

预设回答：

- 答"因为程序里到处是'共 N 条'这种拼接，复数处理错了整句话就错"——对，这是最实际的理由。

- 答"因为有些语言的复数形式不止两种"——非常准确，这是它必须单列的技术原因：不是"加不加 s"，而是"选哪种词形"。

- 答"复数跟软件测试没关系"——错。凡是"数量 + 单位"的界面文字，都涉及复数规则，这在电商、报表、消息提示里到处都是。

- 追问收口：所以正确做法是把复数规则交给本地化框架去处理，而不是在代码里写 if (n > 1)。

易错点 / 考点：① 六项要能默写并各举一例；② "小数点与千分位互换"是最经典的案例，务必能讲清后果；③ 高频判断："复数就是加 s"——错，多复数形态语言里这是错的。

过渡：这六项都跟"区域设置"有关，下一页看区域设置在哪里配、影响什么。

【8.32】区域与语言（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是区域与语言设置的例子，也就是操作系统里那个"你住在哪儿、你说什么语言"的配置界面。别小看这两个下拉框，本地化测试的大部分现象都由它决定。

它一般分三块，你要知道每块管什么。

第一块，**格式**。它决定日期怎么写、时间用不用 24 小时、数字的千分位和小数点、货币符号、度量衡单位。这一块是纯显示层的，改它不影响程序逻辑，只影响呈现——这正是 8.3.1 那六项的实验台。

第二块，**位置**。它决定你所在的国家/地区，进而影响一周从哪天开始、默认纸张、默认度量衡。有些系统还把"位置"用于首页内容、天气、时区推断。

第三块，**管理里的"非 Unicode 程序的语言"**。这一项最关键，也最容易被忽略。它的作用是告诉系统："那些不支持 Unicode 的老程序，用哪套编码去解释它们的文字。"改这一项通常要重启。**很多"某个老软件打开中文全是乱码"的问题，根子就在这里**——老程序用的是系统默认代码页，而默认代码页跟你的文件编码不一致。

测试上怎么用这一页？我建议你做三组实验：**只改格式不改语言**（看格式类显示变不变）、**只改语言不改格式**（看界面文字变不变）、**改非 Unicode 程序的语言**（看老模块会不会乱码）。三组实验分别对应显示、翻译、编码三个层次，做完你对这个软件的区域处理能力就有底了。

还有一个坑要提醒：**改完设置要重启或者重新登录才生效，而且不同版本的 Windows 表现不一样**。测试时如果没重启就下结论，很容易误判成缺陷。

> **板书：**三块：格式（日期・时间・数字・货币・度量衡）｜位置（国家地区・一周起点・纸张）｜管理→非 Unicode 程序语言（代码页，需重启）。底部三组实验："只改格式｜只改语言｜改默认代码页"。

提问："某个老模块打开中文全是乱码，其它模块正常"，你最该先查什么？

预设回答：

- 答"查那个模块是不是不支持 Unicode"——方向对，这类老模块通常按系统默认代码页解码。

- 答"查系统的非 Unicode 程序语言设置"——非常准确，这就是最常见的原因。

- 答"重装字体"——错。字体解决字形，不解决解码。

- 收口：所以这类问题的修复往往有两种：改代码支持 Unicode（治本），或调整系统默认代码页（缓解，且影响其他程序）。

> **易错点 / 考点：**① 格式/位置/非 Unicode 语言三块各管什么，别混；② "改设置需重启才生效"是测试执行细节，常被忽略导致误判；③ 高频考点：非 Unicode 程序语言与代码页的关系。

> **过渡：**配置讲完，我们看这些格式在真实界面上出错的样子。

【8.33】本地化问题 - 数据格式（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是数据格式出错的实例。我把常见的错误形态归成五类，你以后看到就能立刻认出来。

第一类，符号错位。比如该用逗号做千分位的地方用了点，或者小数点用了逗号。这类错误在数字上最危险，因为它不报错，只是数值含义变了。

第二类，顺序颠倒。日期写成月/日/年但目标地区习惯日/月/年；姓名写成名在前但目标地区习惯姓在前。这类错误用户能看出来，但会显得很不专业。

第三类，单位没换。该显示公里显示英里，该显示摄氏度显示华氏度，纸张该用 A4 用了 Letter。这类错误在打印和报表上后果最明显——直接印错。

第四类，格式串硬编码。整块界面上只有某几个字段的格式不对，其它都对。这通常是那几个字段用了自己拼的字符串，没走统一的格式化函数。**这一类是最值得提缺陷的，因为它暴露的是架构问题**。

第五类，时区处理错误。显示的是服务器时间、跨时区时日期差一天、夏令时切换那天的记录时间错乱。这类缺陷特别隐蔽，只有在特定日期、特定时区才能复现。

怎么测这五类？核心手法还是"**对照**"：同一份数据，在两种区域设置下各截一张图，把两边的格式逐字比对。比对的时候不要凭记忆，要拿一张"目标地区的正确格式"参照表——这张表可以从目标地区的官方习惯、或者系统自带的格式里取。

最后一句：这一类缺陷**最容易在评审时被当成"小问题"放过**，但它们在金融、医疗、物流系统里是会出真事故的。判严重度的时候，看它有没有流进数据层。

> **板书：**五类：符号错位｜顺序颠倒｜单位没换｜格式串硬编码｜时区处理错误。右侧写测试手法："同一数

据・两种区域・逐字比对"。

提问：界面上只有两三个字段的日期格式不对，其它都对，你判断根因是什么？

预设回答：

- 答"这两个字段是特殊处理的"——对，通常就是有人图省事自己拼了字符串。

- 答"可能是翻译错了"——不太可能。日期格式不是翻译出来的，是格式化函数算出来的。

- 答"可能是这两个字段的数据类型不对，存的是字符串"——非常专业，这是根因层面的判断：字符串就没法格式化。

- 追问收口：所以这条缺陷的价值不在"改这两个字段"，而在"**全项目搜一遍还有多少地方在拼格式字符串**"。

易错点 / 考点：① 五类错误形态要能认出来；② "局部格式不对"往往指向硬编码，这是案例题的高分点；③ 严重度判断标准：是否流入数据层、是否影响金额或时间。

过渡：数据格式说完，进入 8.3.2——页面显示和布局，第一件事就是文本长度。

【8.34】8.3.2 页面显示和布局（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是显示与布局的总纲，三句话，每句都能展开成一堆测试点。

第一句，**德语最长，汉语比较精炼**。这句话是本地化布局问题的总根源。德语靠复合词承载信息，一个词能很长；中文一字一义，同样的意思用的字符少。所以把中文界面译成德语，几乎必然出现"装不下"；反过来把英文译成中文，往往会出现"留白太多"。测试的含义是：**做布局测试，一定要用"最长的语言"和"最短的语言"各测一遍**，中间的语言自然会过。

第二句，**乱码（双字节语言 GB/BIG5/JP/...）**。课件括号里列的这三个，代表的是三类典型的双字节编码：GB 是简体中文的国标，BIG5 是繁体中文，JP 是日文。它们的共同点是"一个字符用两个字节"，也共同面临"编码不一致就乱码"的风险。这一句还提示了一个现实：**多语言环境里往往同时存在好几种编码**，一个系统里既有 UTF-8 的数据，又有历史遗留的 GB 数据，还有从日文系统导出的文件——这才是真实项目的复杂度。

第三句，**字符索引、排序**。索引指按什么建目录：按拼音、按笔画、按部首、按字母。排序指按什么规则比

大小：不同语言的字母表顺序不同，还有变音符号、大小写、数字混排的处理规则。这一项在中文环境里就已经很复杂（拼音序和笔画序结果不同），多语言下更复杂——一个列表里同时有中文、英文、日文，按什么排？

三句话归一句：**布局测试要问三件事——长度够不够、编码通不通、顺序对不对。** 这三件事分别对应三类工具：截图比对、编码扫描、排序结果核对。

板书：三句：① 德语最长・汉语精炼（用最长最短各测一遍）② 乱码（GB / BIG5 / JP 双字节）③ 字符索引与排序（拼音・笔画・部首・字母序）。右侧："长度够不够｜编码通不通｜顺序对不对"。

提问：一个列表里同时有中文、英文、日文的名字，你打算按什么规则排序？

预设回答：

- 答"统一按 Unicode 编码排"——技术上可行、结果稳定，但用户看不懂这个顺序（汉字按编码排既不是拼音也不是笔画）。

- 答"按当前界面语言的规则排"——这是主流做法，中文界面按拼音、英文界面按字母，比较符合用户预期。

- 答"随便排，用户会自己搜"——错。排序是检索的基础功能，排序乱意味着用户找不到东西。

- 追问收口：所以排序规则要**明确写进需求**，并且写清"混合语言时怎么办"，否则开发和测试各按各的理解做，必吵。

易错点 / 考点：① "德语最长、汉语精炼"是课件原话；② GB/BIG5/JP 分别对应简体、繁体、日文，别搞混；③ 索引（拼音/笔画/部首）与排序（字母序/编码序）是两个概念，考试常混。

过渡：装不下的时候，界面上具体是什么样子？下一页给的是"不能完全显示"。

【8.35】示例：不能完全显示（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是"不能完全显示"的实例。这是本地化缺陷里用户最容易发现、也最影响观感的一类。我把它的四种典型形态说清楚，你以后看图就能对号入座。

第一种，硬截断。文字被直接切掉，末尾没有任何提示。用户看到的是半句话，而且不知道后面还有内容。这是最差的一种。

第二种，加省略号。文字被截断但末尾有省略号，用户知道"还有内容"，但看不到。这种比硬截断好，前提是有悬浮提示或者能点开看全文；如果没有，用户就只能猜。

第三种，撑破容器。文字没被截断，但把控件撑变形了——按钮变成长条、对话框被顶宽、工具栏换行导致下面所有内容下移。这种问题往往连带一片：一个控件变宽，整行布局全乱。

第四种，压盖。文字盖住了旁边的图标或其它文字，两层内容叠在一起，两边都读不了。

这四种形态的严重度不一样：**压盖和撑破最严重**（因为影响其它元素），硬截断次之，加省略号最轻（前提是能看全文）。

测试的时候要注意一个细节：**同一个界面，在不同分辨率、不同字号、不同 DPI 下，表现可能完全不同**。你在 1920 宽屏上看着正常，用户在 1366 的笔记本上就截断了。所以本地化布局测试一定要**在目标地区的常见分辨率下测**，而不是只在自己的开发机上瞄一眼。

还有一条经验：截图时要**同时截源语言版和目标语言版的同一个界面**，两张并排放进缺陷单。开发一看就知道差多少，不用来回问。

> **板书：**四形态：硬截断｜省略号｜撑破容器｜压盖。标严重度：压盖・撑破 > 硬截断 > 省略号。右侧："多分辨率・多字号・多 DPI 都要试"。

提问：界面文字加了省略号，算不算合格？

预设回答：

- 答"算，至少有提示"——不完全算。要看能不能看到全文：悬停提示、点开详情、或者能拖动列宽。看不到全文，用户还是没法用。

- 答"不算，应该让控件自适应"——这是治本的要求，但现实中列宽固定是常态，省略号加完整提示是常见的折中方案。

- 答"看是什么字段，关键信息不能省"——非常专业。金额、日期、编号这类字段被截断，后果远比描述性文字严重。

- 收口：所以判合格的标准是：**信息能不能被用户完整获取**，而不是"看起来有没有提示"。

易错点 / 考点：① 四形态与严重度排序；② "同一界面在不同分辨率/DPI 下表现不同"是执行要点，常被漏；③ 缺陷单要"双语截图并排"，这是实操规范。

> **过渡：**装不下是一种问题，读不出来是另一种——下一页看乱码。

【8.36】示例：乱码（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是乱码的实例。前面 8.1.2 讲过原理，这一页看现象，并且教你**从现象反推原因**——这是测试人员最值钱的一项技能。

看现象分三种。

第一种，成串的怪字符。原来的中文位置变成了一堆莫名其妙的符号，比如"锟斤拷"、"Ã¤Â¸Â"这样的东西。这种的根因几乎总是**编码不一致**：数据是 UTF-8，程序按 GBK 解了；或者数据是 GBK，程序按 UTF-8 解了。特征是同一条数据在别的机器、别的入口能正常显示。

第二种，方框或者豆腐块。每个字的位置是一个空心方框，或者一个带问号的方块。这不是解码错，是**字体缺字形**：编码解对了，但这个字体里没有这个字符的形状。特征是换个字体就正常。

第三种，半个字或者问号。一个汉字显示成半个，或者干脆变成"?"。这通常是**截断造成字节断裂**，或者经过了一次"无法表示就替换成问号"的有损转换。特征是这个字段往往有长度限制。

三种现象，三种根因，三种修法：**第一种改编码声明或转换逻辑，第二种装字体或换字体，第三种改截断方式（按字符截，不按字节截）**。这就是为什么我反复强调"先定位层次，再提缺陷"——你报"界面有乱码"，开发只能猜；你说"该字段在 UTF-8 数据源下按 GBK 解码，建议统一为 UTF-8"，开发十分钟就能改完。

补一条经验：乱码问题有一个特点——**它往往不是全乱，而是局部乱**。因为只有经过某条特定路径的数据才会错。所以复现时要记清"**数据从哪来、经过哪些环节、在哪个环节被转码**"，这条链路就是根因所在。

> **板书：**三现象三根因：成串怪字符 → 编码不一致｜方框豆腐块 → 字体缺字形｜半个字或问号 → 字节截断/有损替换。右侧："先定位层次，再提缺陷"；底部："记清数据链路：来源 → 环节 → 转码点"。

> **提问：**同一段文字，在网页上正常，导出成文件再用记事本打开就乱码，你怎么解释？

> **预设回答：**

- 答"文件编码和网页编码不一样"——对，网页有编码声明（响应头或页面里的声明），文件没有，记事本只能猜。

- 答"记事本太老了"——这是事实层面的补充：老版本记事本对编码的自动识别很弱，容易猜错。

- 答"文件坏了"——错。文件本身完好，是"解释方式"错了。

- 追问收口：所以导出功能要保证"**文件里带编码声明**"，或者统一用兼容性最好的编码。这一条可以直接写成一条测试用例。

> **易错点 / 考点：** ① 三种现象与三种根因的对应关系必须能背；② "局部乱码"的复现要记录数据链路，这是实操要点；③ 高频判断："乱码装个字体就好"——只对第二种现象成立。

> **过渡：** 显示问题还有一类不在文字上，而在图片和控件里——下一页叫"其它"。

【8.37】示例：其它（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页的标题叫"其它"，其实是把所有不属于"文字长度"和"乱码"的界面本地化问题收在一起。我按四类给你补全，因为这一类在实际项目里漏得最多。

第一类，图片里的文字。图标上写着 "Save"，按钮背景图里印着 "Submit"，截图里带着英文的界面元素。这类文字**抽不进资源文件**，只能重画图。测试时要专门去找"图片里的文字"，包括图标、徽标、示意图、帮助文档里的截图——这是最容易被漏的一类，因为测试人员的注意力都在文本上。

第二类，图标的隐喻。邮筒、手势、动物、颜色、节日元素。同一个图标在不同文化里含义可能完全不同。这类问题不属于"显示错误"，属于"文化不适"。

第三类，隐含的格式假设。比如地址栏按"省-市-区-街道"排；比如姓名字段假设"两个字就是姓名"；比如"邮编"字段只留 6 位数字。这些假设在源语言环境里天经地义，换个国家就全崩。测试时要把每个输入框都问一遍"**这个格式在别的国家还成立吗**"。

第四类，与语言绑定的行为。大小写转换（有些语言没有大小写）、首字母大写（有些语言不这么用）、连字符断词（只对拉丁语言有效）、文字方向、以及"按字母分组"的联系人列表——中文该按拼音分组，日文按假名，这些都算行为层面的本地化。

这一页的教学目的，是让你建立"**本地化不只是文本**"的意识。文本只占界面的一部分，图片、图标、布局、行为假设，都是本地化的工作面。你在做本地化走查时，眼睛要能"分层看"：文字一层、图片一层、控件一

层、行为一层。

> **板书：**四类：图片里的文字（需重画）｜图标隐喻（文化）｜隐含格式假设（地址・邮编・姓名）｜与语言绑定的行为（大小写・断词・方向・分组）。右侧："本地化不只是文本，分层看：文字｜图片｜控件｜行为"。

提问：一个"按首字母分组"的联系人列表，在中文环境下该怎么改？

预设回答：

- 答"改成按拼音首字母分组"——这是主流做法，中文用户也熟悉。

- 答"改成按笔画或者部首"——可行，但用户认知成本高，实际产品很少这么做。

- 答"不用改，中文也能按编码分组"——错。按编码分组出来的分组名对用户毫无意义。

- 追问收口：所以这类"与语言绑定的行为"，改动量往往比翻译大得多——它要改逻辑，不只是改文本。

易错点 / 考点：① "图片里的文字抽不出来，必须重画"是高频考点；② 隐含格式假设的四类例子（地址、姓名、邮编、电话）要能举；③ 常考判断："本地化就是把文字翻译一遍"——这一页就是反驳它的四条证据。

过渡：把上面这些零散的点系统化，就是下一页的 UI 验证细节，六条。

【8.38】UI 验证的细节（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章最实用的一页，六条 UI 验证细节，每条都能直接变成检查项。讲完给你一张走查表。

第一条，控件相互重叠或排列间隔不均衡。重叠是硬伤——两个控件压在一起，用户点不准；间隔不均衡是"看着别扭"——有的地方挤、有的地方空。根因通常是布局用了绝对坐标，译文一变长就挤到别人地盘上。

第二条，文字遮挡图像、文字超过边界、控件中字符没有完整显示。这三种是"装不下"的三个变体，但要注意区分：遮挡是层级问题（z-order），超边界是容器问题，字符不完整是裁剪问题。修法不同。

第三条，文字方向的问题，比如希伯莱文和阿拉伯文是从右到左显示。这里课件点出了两种 RTL 语言：希伯莱文和阿拉伯文。这一条要验的是：整段方向对不对、控件顺序有没有镜像、标点位置对不对、数字在 RTL 段落里的位置对不对。

第四条，左右对齐问题。课件给了两个很好的例子：阿拉伯文应右对齐；中英文之间有区别——**中文段落开头要空两个字，英文开头不空**。这个细节特别能说明问题：首行缩进不是"美观偏好"，是**排版规则**，按语言不同而不同。如果程序统一加了缩进，英文段落就会显得很怪；反之中文段落不缩进，读起来也不符合习惯。

第五条，连字符对多数拉丁语言有效，但对东方语言一般无效。连字符断词是为了在行末把长单词拆开，拉丁语言按音节拆是合理的；中文、日文没有"词间空格"，也不需要连字符断词，中文可以在任意字之间换行。所以如果排版引擎给中文加了连字符，那就是错的。

第六条，拉丁语言的大小写问题、多字节语言的显示乱码问题。大小写：某些界面会做"全部大写"的样式处理，遇到没有大小写的语言（中文、日文、阿拉伯文）会怎样？乱码：还是前面讲的那三类现象。

六条走查完，它们覆盖的正是 UI 本地化"看得见"的全部：**位置、边界、方向、对齐、断行、字符**。口诀：**"挤不挤、装不装、朝哪边、对不对齐、怎么断、显不显"**。

> **板书：**六条：① 控件重叠・间隔不均 ② 文字遮挡图像・超边界・字符不完整 ③ 文字方向（希伯莱・阿拉伯 RTL）④ 左右对齐（阿拉伯右对齐；中文首行空两字，英文不空）⑤ 连字符对拉丁有效、东方无效 ⑥ 大小写・多字节乱码。底部口诀："挤不挤・装不装・朝哪边・对不对齐・怎么断・显不显"。

提问：中文段落首行空两个字、英文不空，这个差异该由谁保证——翻译、设计，还是代码？

预设回答：

- 答"翻译的时候在每段前面加两个空格"——这是最常见的土办法，但很脆弱：换字号、换字体、复制粘贴都会错位。

- 答"由排版/样式控制，按语言设置首行缩进"——这是正确做法，样式跟着语言走。

- 答"无所谓，看不出来"——错。排版规则是本地化的基本礼貌，专业用户一眼就能看出"这不是本地做的"。

- 追问收口：所以样式也要国际化——**同一个样式类，在不同语言下可以有不同的缩进、行高、字体**。

易错点 / 考点：① 六条要能按"位置・边界・方向・对齐・断行・字符"归类；② "中文首行缩进两字、英文不缩进"是课件明确给的细节，非常适合出选择题；③ 高频判断："连字符断词对所有语言都适用"——错，对东方语言一般无效。

过渡：界面看完了，还有一层在"环境"里——配置与兼容性。

【8.39】8.3.3 配置和兼容性问题（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

8.3.3 讲配置和兼容性。这一块的特点是：**问题不在代码里，在环境里**。同一份安装包，装在不同配置的机器上表现不同。课件给了三句话，我逐句展开。

第一句，**配置性包括键盘布局设计，它是语言依赖性最大的硬件，还有打印机配置等**。注意"语言依赖性最大的硬件"这个说法——键盘。为什么？因为键盘是物理的：德语键盘的 Z 和 Y 位置对调，法语键盘是 AZERTY 布局，阿拉伯语键盘上还有专门的键。软件如果按物理位置或者按字符绑快捷键，换个键盘就可能失效或者串键。打印机配置也是配置的一部分：纸张尺寸、默认边距、驱动语言，都会影响输出。

第二句，**兼容性包括与硬件的兼容性、与上一版本的数据兼容及与其他本地化软件的兼容性等等**。三类兼容要分清：**硬件兼容**（在目标地区的常见机型上能不能跑）、**版本兼容**（从上一版升级上来，数据和设置会不会坏）、**与其它本地化软件的兼容**（同机装了别的语言包、别的输入法、别的办公套件，会不会冲突）。第三类最容易被忽略，但现实中最常见——用户机器上永远不止你一个软件。

第三句，**关注一些点，如数据库问题、热键**。数据库问题指的是：数据库的字符集、排序规则、字段长度，跟本地化数据能不能对上。一个 GBK 排序规则的库，存进 UTF-8 的数据，轻则乱码重则截断。热键前面讲过：菜单助记符要重新分配，还要避开系统保留键和其它软件的占用。

三句话合起来：**配置看键盘和打印机，兼容看硬件版本和邻居软件，细节看数据库和热键**。这一块没有巧办法，只能**搭环境、真机跑**。

> **板书：**三句：① 配置性（键盘布局＝语言依赖最大的硬件・打印机配置）② 兼容性（硬件・上一版本数据・其它本地化软件）③ 关注点（数据库・热键）。底部："配置靠搭环境，兼容靠真机跑"。

提问：为什么课件说键盘是"语言依赖性最大的硬件"？

预设回答：

- 答"因为不同语言的键盘布局不一样，键位和字符的对应关系不同"——对，这是核心原因。

- 答"因为键盘上有当地语言的字母"——也算一层：阿拉伯语、俄语键盘上印的是当地字母。

- 答"键盘就是输入设备，跟语言没关系"——错。同一台电脑插上不同布局的键盘，敲同一个物理键出来的字符可能完全不同。

- 追问收口：所以测"快捷键"时，不能只在自己的键盘上按一遍，要**在目标语言的键盘布局下按一遍**。

> **易错点 / 考点：**① "键盘是语言依赖性最大的硬件"是课件原话，常出填空或判断；② 兼容性三类（硬件・版本数据・其它本地化软件）要能说全；③ 数据库的字符集与排序规则是高频细节考点。

> **过渡：**配置和兼容性具体怎么测？下一页给了三个非常具体的场景。

【8.40】配置和兼容性测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给的是配置与兼容性测试的三个具体切入点，加上一个参考资料。这三个点都很"工程"，我把它们讲透。

第一，本地化操作系统。测本地化版本，最好在**目标语言的操作系统**上测，而不是在自己机器上装个语言包。为什么？因为语言包只改界面语言，系统的默认代码页、区域格式、字体集合、输入法、甚至某些 API 的行为可能都没变。你在中文系统上装个德语语言包，测出来的"通过"是不可靠的。这一条是所有本地化测试规范里的第一条。

第二，Non-Unicode 数据路径。指的是程序处理"非 Unicode 编码的数据"时走的那些路径：读一个 GBK 的文本文件、导入一个 BIG5 的 CSV、访问一个老数据库。这类路径在现代代码里往往是历史遗留，最容易出乱码和截断。测试方法很直接：**准备一批不同编码的数据文件，逐个导入，看结果**。

第三，欧洲系统本地化 OEM vs. Windows "ANSI"。这一条讲的是编码生态的复杂性：同一个 Windows，在不同地区的 OEM 版本里，默认的"ANSI"代码页可能不一样——西欧是 1252，中欧是 1250，中文是 936，日文是 932。所谓"ANSI"其实是各家代码页的统称，不是一种编码。所以一个程序如果依赖"系统 ANSI 代码页"，它在不同地区的机器上行为就不同。这正是"**同一份程序，在不同国家的机器上表现不一样**"的技术根源。

课件还给了一篇 W3C 的文章，标题是 "Migrating to Unicode"——迁移到 Unicode。这篇文章值得看，它讲的是"**怎么把一个老的单字节/本地编码系统，安全地迁到 Unicode**"，包括数据迁移、代码改造、以及迁移过程中怎么避免数据损坏。这类文章对理解 8.3.3 的全部问题非常有帮助。

三个点归一句：**本地化测试必须在"真环境"里做，而真环境的核心差异，一半在操作系统，一半在编码。**

> **板书：**三点：① 本地化操作系统（不是装语言包）② Non-Unicode 数据路径（老编码文件・老库）③ 欧洲

> OEM vs Windows "ANSI"（代码页 1252/1250/936/932）。右侧写参考资料："W3C — Migrating to Unicode"。底部："真环境＝操作系统 + 编码"。

> **提问：**在中文 Windows 上装德语语言包，能不能替代德语版 Windows 做测试?

> **预设回答：**

- 答"能，界面都变德语了"——错。界面语言只是表层，代码页、字体、输入法、区域默认值都可能没变。

- 答"不能，但可以作为初筛"——这是很务实的判断：语言包环境能发现一部分翻译和布局问题，但编码类问题必须上真系统。

- 答"看测试目的，测翻译够用，测编码不够"——完全正确，这就是分层测试的思路。

- 收口：所以测试计划里要写清"**哪一类用例在语言包环境跑，哪一类必须上目标语言系统**"，这是资源有限时的现实取舍。

> **易错点 / 考点：**① "ANSI 不是一种编码，是一堆代码页的统称"这一点极易考；② 三个切入点的名称要能说出；③ 高频判断："装语言包等于本地化环境"——错。

> **过渡：**8.3 的技术问题到此结束，进入 8.4——把本地化版本当成一个完整软件来测。

【8.41】8.4 本地化功能测试（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 8.4 只有一页标题，但它标志着一个转折：前面 8.3 讲的是"**一类一类地查问题**"，8.4 讲的是"**把本地化版本当成一个完整软件，从头到尾测一遍**"。
>
> 这两件事不一样。8.3 是清单式的：查格式、查布局、查兼容。8.4 是场景式的：一个用户拿到德语版，从安装、登录、用功能、打印、导出、看帮助、升级，整条链路走下来，会不会在某一步卡住。
>
> 为什么必须做这一步？因为清单再全，也覆盖不了"组合"。举个典型场景：日期格式在列表页显示正常（8.3.1 过了），但把这个日期导出成 Excel 再导入回来，就错了——这个缺陷既不在格式清单里，也不在布局清单里，它藏在"**导出导入这条链路**"上。只有按业务流程走一遍，才会撞见。
>
> 所以 8.4 的关键词是"**集成**"：不只是界面翻译对了，而是整条业务链在目标语言环境下跑得通。

下一页我们看这一节的具体工作怎么分。

板书：中间一条横线：左边"8.3 = 清单式（查一类一类问题）"，右边"8.4 = 场景式（按业务链路走一遍）"。下方写："清单覆盖不了组合，组合缺陷只在链路里出现"。

提问：清单式测试和场景式测试，哪个更容易漏缺陷？漏的是哪一类？

预设回答：

- 答"清单容易漏，因为清单是有限的"——对，清单只能覆盖"想到的"。

- 答"场景容易漏，因为没有明确判据"——也有道理，场景测试依赖测试人员的经验和想象力。

- 答"两种都会漏，但漏的东西不一样：清单漏组合，场景漏细节"——这个回答最完整。

- 收口：所以成熟的测试策略是**两者叠加**：先按清单扫一遍保证覆盖率，再按场景走一遍保证连通性。

易错点 / 考点：① 8.3 与 8.4 的定位差别（清单 vs 场景）是理解本章结构的关键；② 高频判断："本地化测试就是检查界面翻译"——8.3 和 8.4 都在反驳它；③ 案例题常给"导出再导入出错"这类跨环节缺陷，考点就是"组合缺陷"。

过渡：进入具体工作，本地化功能测试分成三块，还对应两种行业通行的组织形式。

【8.42】本地化功能测试的具体工作（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页把本地化测试分成三块——**语言、功能、美观/界面**——然后给出两种组织形式：**In-country testing（本土测试）**和 **Translation verification testing（TVT，翻译验证测试）**。这一页信息密度很高，我分两半讲。

先讲三块。**语言**：查译文本身，漏译、错译、术语、语气、符号。**功能**：查业务功能在目标语言环境下是否正常，包括输入、处理、输出、打印、导出。**美观/界面**：查排版、对齐、截断、方向、图标。

这三块的划分很实用，因为它直接对应三类执行者：语言由译审或本地语言专家做，功能由测试工程师做，美观由 UI/前端或有审美判断的人做。**分对了人，效率就上来了。**

再讲两种组织形式。

In-country testing，本土测试，也叫"在当地测"。意思是把测试活动放到目标市场去做：用当地的机器、当地的网络、当地的系统版本、当地的用户习惯，最好由当地的人来测。它的价值在于"**真实**"：你在国内模拟不出来的东西——当地的网速、当地的支付习惯、当地常见的机型、当地用户的操作直觉——只有在当地才能撞见。代价是贵、协调难。

Translation verification testing，TVT，翻译验证测试。它的关注点窄而深：**只验翻译**。典型做法是语言专家拿着源语言和目标语言对照的界面，逐条检查译文是否准确、是否符合语境、术语是否统一。TVT 通常有专门的工具支持：双语对照界面、术语库比对、上下文截图。

两种形式的关系：**TVT 保证"字对"，In-country testing 保证"用得对"**。预算有限时，先用 TVT 把语言问题清干净（便宜、可并行），再用本土测试做最终把关（贵、但不可替代）。

板书：三块：语言｜功能｜美观/界面（对应译审・测试・UI）。两种组织形式：In-country testing（本土测试・真实环境・当地用户）｜TVT（翻译验证测试・双语对照・只验翻译）。底部："TVT 保字对，本土测试保用得对"。

提问：TVT 和普通的"翻译审校"有什么区别?

预设回答：

- 答"TVT 是在软件界面上看，审校是看文档"——很准确。TVT 的关键在于"**在上下文里看**"：同一句话在按钮上和在提示框里，译法可能不同。

- 答"TVT 要检查术语一致性，还要看有没有截断"——补充得好，TVT 往往连带发现布局问题。

- 答"没区别，都是看翻译"——错。脱离界面的审校看不到语境、看不到长度限制、也发现不了占位符错误。

- 收口：所以 TVT 是"**在真实界面里验翻译**"，这是它不可替代的价值。

易错点 / 考点：① 三块内容与三种执行者的对应关系；② In-country testing 与 TVT 的英文全称、中文译名、定位差别都要能说；③ 高频判断："TVT 可以替代本土测试"——错，两者互补。

过渡：这一节最后落到 L10N 功能测试的具体内容：集成、索引排序、联机文档。

【8.43】L10N 功能测试（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页是 L10N 功能测试的三项内容，加一段很重要的说明，我把说明放在最后讲，因为它最容易被忽略。
>
> 第一项，**集成测试**。本地化版本要和什么集成？和操作系统、和数据库、和打印机、和外部系统接口、和其它语言版本共存。典型的集成问题是：装了两个语言版本互相覆盖、语言包和主程序版本不匹配、和本地的办公软件交换文件时编码错乱。这一项的本质是"**本地化版本不是孤岛**"。
>
> 第二项，**索引和排序**。前面 8.3.2 提过原理，这里是功能层的验证：搜索能不能搜到、按字母/拼音/笔画建的索引对不对、排序结果符不符合当地习惯、分页和"跳到某个字母"能不能用。这一项在通讯录、文档管理、电商商品分类里是核心功能，排序错了等于搜索废了。
>
> 第三项，**联机文档的功能测试**。注意是"功能"不是"内容"：帮助能不能打开、目录能不能跳转、搜索能不能用、索引链接有没有失效、PDF 里的书签和字体正不正常。文档的内容准确性属于翻译验证，文档的**可用性**属于功能测试，这两件事要分开提缺陷。
>
> 然后是那段说明，我认为是全章最实用的一句：**不仅要查看用户界面，而且要对文件保存、打印等类似的功能进行测试，特别要注意语言环境特定的组件，比如时间、日期格式以及文字处理等相关方面的功能进行测试。**
>
> 这句话在纠什么偏？纠的是"本地化测试就是看界面"这个偏见。真正的本地化缺陷，一大半在"**界面之外**"：保存下来的文件名对不对、导出文件的编码对不对、打印出来的纸张和边距对不对、复制粘贴到别的软件里会不会乱码。这些都是"看不见界面"的地方，却恰恰是用户真正在干活的地方。
>
> 所以这一页我给你一句行动指令：**测本地化版本时，把"保存、打开、打印、导出、复制粘贴、发邮件"这六个动作，每个都用目标语言的数据做一遍。**

> **板书：**三项：集成测试｜索引和排序｜联机文档功能测试。下方框内写："界面之外：保存・打开・打印・导出・复制粘贴・发邮件——每个都要用目标语言数据做一遍"。右侧批注："语言环境特定组件：时间・日期・文字处理"。

> **提问：**为什么"打印"在本地化测试里要被单独强调？

> **预设回答：**

- 答"因为纸张尺寸不一样，A4 和 Letter 会错位"——对，这是最直接的原因。

- 答"因为打印出来的字体可能不对，中文可能变方框"——也对，打印链路用的是另一套字体和渲染。

- 答"打印跟本地化没关系"——错。打印涉及纸张、边距、字体、编码、驱动，四样都可能跟地区有关。

- 追问收口：所以打印机这一环，是"界面看着好、输出却错"的典型场景，也是 8.3.3 里"打印机配置"被列出来的原因。

> **易错点 / 考点：**① 三项内容要能说出；② 那段说明是全章的高频引用句，"界面之外的功能"是核心观点；③ 高频判断："本地化测试只需要看界面"——这一页就是标准反例。

> **过渡：**本章的内容到这里讲完了，我们收一下，把整章压缩成三句话。

【8.44】本章小结（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章小结，三句话，正好对应本章的三层结构。我按"研发、测试、执行"的顺序给你串一遍。

第一句，**软件国际化是研发上多工程问题，在设计、编程上需要遵守 I18N 相应的规范和要求**。注意"多工程"这个说法——它不是说"很多工程"，而是强调国际化是**贯穿设计、编码、构建、文档的系统性工程**，不是一个可以后期补上的功能。这一句对应 8.1：i18n 属 Engineering，做一次管所有语言。

第二句，**国际化测试需要验证双向识别功能、语言切换方式、多字节和单字节文字的混合、输入法编辑器、大小写转换、快捷组合键、词序问题、无硬编码**。这一串就是 8.1 那十项测试点的浓缩版。注意最后三个字"**无硬编码**"——它是整句的落脚点，因为硬编码是绝大多数国际化缺陷的共同根因。

第三句，**本地化测试会涉及"数据格式、UI 显示和布局、配置和兼容性问题、翻译内容等"**。这一句对应 8.3 和 8.4：四类问题，加上前面 8.1.5 的六类测试工作、8.2 的翻译验证，构成完整的执行面。

三句话合起来，就是一张责任图：**研发负责"能做到"，测试负责"验证做到了"，本地化团队负责"做得好"。**

最后我给你一个"三问自检"，考试和面试都用得上。拿到任何本地化任务，先问三句：

一、**这个功能有没有把地区相关的东西硬编码？**（查 i18n 根子）

二、**同一份数据在目标地区显示成什么样？**（查格式与布局）

三、**在目标环境里，整条业务链跑得通吗？**（查集成与兼容）

三问都过了，本地化质量就基本靠得住了。

> **板书：**三句：① 国际化＝研发的多工程问题（设计・编程守 I18N 规范）② 国际化测试八项（双向・切换・

> 混排・IME・大小写・快捷键・词序・无硬编码）③ 本地化测试四类（数据格式・显示布局・配置兼容・翻译内容）。底部三问："硬编码了吗？显示成什么样？整条链跑得通吗？"

提问：如果只允许你保留一条本地化测试用例，你保留哪一条？为什么？

预设回答：

- 答"切换语言后逐页走查"——覆盖面最广，一条用例能把翻译、布局、方向、编码都过一遍。

- 答"用德语在德语系统上跑一遍主流程"——很有价值，一条用例同时压到长度、格式、环境。

- 答"随便留一条，反正测不完"——这种心态要纠正。测试资源永远不够，**取舍能力才是专业能力**：留哪条，取决于这个产品最怕哪类事故。

- 收口：所以真正专业的回答是"**看业务风险**"——金融产品留格式和金额，出海消费产品留布局和翻译。

易错点 / 考点：① 三句话要能背，尤其"无硬编码"和"多工程"这两个词；② 本章的四组数字容易混：8.1 的八条能力、五条标准、八步流程、六类测试，答题前先对号；③ 高频综合题："结合本章，谈谈如何组织一个多语言版本的测试"——就按"i18n 先行 → 伪翻译 → TVT → 本土测试 → 发布验证"这条线答。

过渡：最后留几分钟，把可以继续深挖的资料给你们。

【8.45】参考资源（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章的参考资源，五个入口，我按"用处"给你排个序，你课后按需取用。

第一个，**W3C 的 International 站点**。这是国际化领域最权威的公共资源之一，讲字符编码、语言标签、文字方向、本地化最佳实践。8.1.2 和 8.3.3 那些标准，很多能在这里找到面向开发者的解释。**如果你只想看一个地方，就看它。**

第二个，**Sakai 项目的 I18N 主页**。这是一个开源教学系统项目的国际化资料，价值在于"**工程实践**"：他们怎么抽资源、怎么管理多语言、怎么处理翻译流程。看真实项目怎么做，比看规范更接地气。

第三个，**GALA Global**，全球化与本地化行业协会。这里能看到本地化这个行业的全貌：产业分工、质量标准、术语、职业路径。对想了解"本地化是一门生意"的同学特别有用。

第四个，**MSDN 上 Kieran 的博客**。这是一个长期写国际化与编码问题的技术博客，特点是"**具体**"：一个乱码案例能写清来龙去脉。这类博客最适合在遇到实际问题时去翻。

第五个，**Java 官方的国际化入门文章**。讲 Java 里怎么做国际化：资源包、区域设置、格式化类。如果你要做实验，从这里入手最快——Java 的国际化框架是最经典的实现之一，看懂了它，其它语言的框架都是同一套思路。

五个资源，用法是：**W3C 打底、Sakai 看流程、GALA 看行业、博客查案例、Java 文章做实验**。别指望一次看完，遇到具体问题时去查，效率最高；而且看的时候记得把「它讲的是原理、流程还是案例」分清楚，别拿一篇博客当标准，也别拿一份标准当操作手册。

板书：五个入口：W3C /International｜Sakai I18N｜GALA Global｜MSDN 博客｜Java 国际化入门。右侧写用法："W3C 打底・Sakai 看流程・GALA 看行业・博客查案例・Java 做实验"。

提问：如果你要给团队写一份"本地化测试入门材料"，你会先看这里的哪一个？

预设回答：

- 答"W3C，最权威"——权威但偏原理，写入门材料可能太硬。

- 答"Sakai，因为是完整项目的实践"——很实用的选择，能直接看到测试计划和组织方式。

- 答"都不用看，我自己总结"——不建议。本地化是个成熟行业，别人踩过的坑已经写成文档了，重踩一遍很浪费。

- 收口：写材料的最佳组合是"**一份原理（W3C）+ 一份实践（Sakai）+ 一份自己的项目案例**"。

易错点 / 考点：① 五个资源各自的性质要能对上（标准组织・开源项目・行业协会・技术博客・语言框架文档）；② 考试一般不考网址，但可能考"本章推荐了哪几类参考资料"；③ 记住 GALA 是行业协会，别和标准组织混。

过渡：这一章就讲到这里，最后说两句收尾的话。

【8.46】感 谢 聆 听（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

第 8 章讲完了，最后我用三句话把这一章收住。

第一句，**本地化测试的难点不在技术，在"换位"**。你要能站到一个德国用户、一个阿拉伯语用户、一个美国用户的位置上，去问"这样写我看不看得懂、用不用得顺"。这个能力，比会写脚本更难得，也更值钱。

第二句，**越早做越便宜**。国际化是设计阶段的事，本地化是发布前的事。你在设计评审时多问一句"这个格式硬编码了吗"，可能省掉后面几周的返工。这是软件工程里少数几条几乎无例外的规律。

第三句，**细节就是专业**。一个编码参数、一个首行缩进、一个称谓位置、一个复数词形——每一件都不起眼，但用户看到的全部，就是这些不起眼的东西。

从下一章开始，我们换一个视角，从"测一个软件"转到"管一个测试项目"。这一章希望大家带走的不只是知识点，还有一种习惯：**看到一个理所当然的设计，先问一句"换个地方还成立吗"**。

谢谢大家。

> **板书：**三句：① 难点在换位，不在技术 ② 越早做越便宜 ③ 细节就是专业。底部留一行："换个地方还成立吗？"

提问：这一章里，哪一个知识点最可能在你将来的工作中直接用到？

预设回答：

- 答"字符集和乱码的排查思路"——很实用，几乎每个项目都会遇到。

- 答"文字扩展和布局自适应"——也很实用，做过出海项目的人都会认同。

- 答"这些都用不上，我们只做国内市场"——不一定。国内也有繁体版、少数民族语言版、多终端多分辨率，方法是一样的。

- 收口：所以这一章的方法论，其实是"**如何在差异中保证一致性**"，这个能力在哪儿都用得上。

易错点 / 考点：① 本章的三条主线（i18n 能力・l10n 执行・文化合规）要能在脑子里连成一张图；② 复习时优先背四组数字：八条能力、五条标准、八步流程、六类测试；③ 案例题的通用答法：**先判层次（i18n 还是 l10n）→ 再判类型（格式/布局/编码/文化）→ 最后给建议**。

过渡：本章结束，下一章进入测试项目管理，我们从"怎么测"走向"怎么管"。

四、思政落点清单（共 4 处）

序号	所在页	落点主题
1	8.11	字符集这一页最值得琢磨的地方是——它出错的时候，程序一切正常。没有异常、没有崩溃
2	8.21	这八步里最不起眼的是第一步——配置管理。它不产出任何看得见的东西，却决定了后面七
3	8.26	翻译这一项最考验"对用户负责"这四个字。一段机翻痕迹很重的界面，功能一个不少，可
4	8.27	文化层面这一栏，考的是"尊重差异、守住合规"。有些内容在我们看来稀松平常，在别人