《软件测试课程设计》讲稿（第 7 章）

第 8–9 周 · 两次课各 2 学时（共 180 分钟）

广东金融学院 计算机学院 | 周宇文

讲稿可照读版 | 由课件逐页撰写

第 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）】【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（思政）	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）＋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 节	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**
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	7.41–	7.2 安全性测试：渗透测试、SDLC 贯穿、风险分析、代码评	20	**7.61、7.63 必讲**

节	7.76	审、基于风险		
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、优秀实践、多因素、决策树、架构	26	**7.186、7.191、7.199 必讲**；平台设计可略
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（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）

> 功能一条条都对，用户一多就卡死。这种事你遇到过吗？

>

> 前面六章走的是同一条线：这个软件做得对不对。第 2 章讲概念，第 3 章讲方法，第 4 章讲流程。第 5 章钻到代码里做单元和集成，第 6 章站到系统外面做功能测试。输入进去，输出对不对，业务规则算得对不对。这条线到第 6 章，已经算走完整了。

>

> 可它只回答一半。一个单引号能把数据库拖走，功能测试抓不住。换个手机型号界面崩了，也抓不住。为什么抓不住？因为这些不是对错问题。它们的判据是资源、时间、攻击面、运行环境，功能用例压根覆盖不到这些地方。功能测试问"输出对不对"，它们问"在特定条件下还站不站得住"。

>

> 还有个背景。七八年前，一套系统服务一家公司。现在一套系统可能同时服务上万家租户。用户规模上去了，稳不稳这件事就从加分项变成及格线。

>

> 第 7 章收拾的就是这一类。五个专项：性能、安全性、兼容性、可靠性、易用性。这一讲先攻第一个，也是最烧钱、最容易在生产上炸的那个。它的判据是指标，不是预期结果。这一个区别没打通，后面五节全会听成更细的功能测试。

>

> 听的时候带一个问题：你的系统能扛住多少人同时用？这个数字，你是从哪儿知道的？

> **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，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，换个手机就闪退"——兼容性专项，后面几节会讲到。

- 答"功能都对就不会出事"——最常见的误解。功能测试的判据是预期结果，性能看的是资源、时间、攻击面，用例覆盖不到。

- 收口：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"，判据不是"输出对不对"，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 常见陷阱："专项测试只在系统测试阶段做"——错，单元、集成、系统、验收各阶段都有对应活动，第 8 章讲全生命周期时会再遇到。

> **过渡：**先把这五块在整章里怎么排看清，你心里才有个时间预算。

【7.2】7.1 性能测试（1 分钟）

> 五块内容，一块一块点过去，你看着记结构就行。

>

> 7.1 性能测试，本讲主体，时间最长。要回答四件事：性能问题长什么样、性能测试怎么定义、负载由哪几个要素构成、测试分成哪几类。7.2 安全性测试，看软件会不会越界：漏洞从哪来、攻击分几个阶段、安全目标是什么。7.3 兼容性，同一套代码换个环境还认不认人，换浏览器、换机型、换系统版本都算。7.4 可靠性，长时间跑、出了故障能不能恢复。7.5 易用性，人用起来顺不顺。

>

> 五块底下有一条共同的主线。它们衡量的都不是"对不对"，是"够不够"。够快、够稳、够安全、够通用、够好用。

>

> "够不够"这三个字有代价。它意味着你先得有一个能比的刻度，也就是量化指标。没有指标，测完只能说"我感觉还行"，验收单上签不了字。

>

> 五个方向对着五类反例：卡、被攻破、换环境就崩、跑久了坏、没人愿意用。

>

> 还有一点：五块里没有一块是选修。验收报告五项都要有结论，缺一项，这份报告就不完整。

>

> 所以客户只丢一句"要快一点"，你能开工吗？先把它翻成"首页 95% 的请求 2 秒内返回"这种句子。翻不出来，先谈，别先跑工具。

> **提问：**客户只说"这个系统要快一点"，你现在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"要翻成可验证的句子。

- 追问：指标从哪来？三条路，用户预期、业务容量、技术约束，后面有一页专门讲。

- 收口：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 五个板块的名称和顺序；② 常考简答"专项测试和功能测试的根本区别"——判据不同，指标对预期结果；手段不同，工具加监控对输入输出比对；③ 别把兼容性和可移植性混成一件事：兼容性问在既定环境下能不能正常工作，可移植性问搬到新环境要花多少代价。

> **过渡：**从 7.1 开始，先给你一张这一节的路线图。

【7.3】7.1 性能测试（1 分钟）

> 性能这一节，概念密度是全章最高的。路线不给，容易听散。

>

> 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：负载的三个要素、思考时间为什么关键、负载模式怎么设计、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。第四段收口：分析、调优、监控，还有全生命周期的性能工程。

>

> 顺序不能倒。指标没定就去跑负载，跑出来的数没人能判；负载没设计就去看曲线，看完也不知道从哪改。四段是递着来的，第二段概念最密，也最常被跳过。课上按这个顺序走，记笔记也按这个顺序记。复习时照四段过一遍，比逐页翻快。

>

> 有一句说明要放在前面。这一节的课件做过精简，压力测试、容量测试和工具的细节，教材里篇幅更完整。课上讲骨架和判断口径，命令和参数你课后对着教材和实验补。

>

> 为什么先交代这个？因为考细节时以教材为准，课件留下的是"怎么判断"。骨架立住了，细节你自己能补；骨架散了，细节背再多也用不上。

>

> 这一节的重点可以收成一句：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。

提问：为什么性能测试特别强调"先定指标"？功能测试也先定指标吗？

预设回答：

- 答"功能测试也有预期结果啊"——对，但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，多快算快得人为划线。

- 答"因为性能没有绝对的对错"——抓住了这一点。

- 追问：指标定错了会怎样？会做出一份看着很专业、结论没法用的报告。

- 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 四段结构要能说清；② 常见判断"性能测试就是压力测试"——错，压力测试只是一种类型，这一节后面会分清四种基础类型和三种加码类型；③ 课件写了"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，考细节时以教材为准。

> **过渡：**先建立一个前提，为什么今天性能问题变得这么常见。

【7.4】性能测试（2 分钟）

为什么这几年性能问题突然变多？课件第一句就点了：SaaS 时代、万物互联时代。

>

> 拆开看。SaaS 意味着一套系统同时服务上千个租户，一个人慢可能是所有人慢。万物互联意味着终端规模从十万级跳到亿级，心跳和上报把后端压到另一个量级。用户规模变了，性能问题就从偶发变成常见。

>

> 第二句，压力测试、容量测试这些都并进了性能测试。今天这个词是个大口袋，基准、验证、规划、容量、压力、渗入、峰谷全在里头。所以问"性能测试包括哪些"，只答一种要吃亏。

>

> 第三句，性能测试也分静态和动态，平时说的指动态。静态那部分在评审里：循环里反复建对象、SQL 没走索引，程序不运行就看出来。动态是把系统跑起来、加负载、看指标。

>

> 第四句最实在：性能测试一般只能靠工具。为什么？几百上千并发、每秒请求数要精确、几十项指标要同步采集，手点做不到。功能测试能手工点，性能测试手工点等于没测。

>

> 第五句，结果分析靠功底。工具给曲线，为什么慢，要靠你对操作系统、数据库、网络、代码的理解。工具能测出问题，定位问题靠人。

>

> 第六句，测试和调优循环往复。所以性能不是测出来的，是逼出来、改出来的。

提问：既然只能靠工具，那装上 JMeter 是不是就会做性能测试了？

> **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。

- 答"不是，还得会分析"——对，这就是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，决定你测出来的是不是你要的东西。

- 收口：工具只是必要条件。会跑工具，离会做性能测试还差分析和设计。

> **易错点 / 考点：** ① 六条里"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：架构评审和代码评审属于静态性能测试，跟第 3 章"静态测试不运行程序"一致；③ 常见判断"性能测试做一次就够了"——错，它和调优循环往复，系统大改、数据量上一个量级都要重做基线。

> **过渡：** 性能问题这么常见，那它到底长什么样？先从用户能感觉到的表象看。

【7.5】性能问题（1 分钟）

> 用户不会跟你说"你的线程池打满了"。他只会说"你们系统今天怎么这么慢"。

>

> 性能问题的第一发现者，几乎从来不是测试，是用户和运维。所以干这一行，第一项能力是把抱怨翻成技术问题。"越来越慢"可能是泄漏，"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。

>

> 给你一个固定套路。遇到任何"慢"，都问三句。慢在什么时候，启动、查询还是下载。慢给谁看，所有人还是某个地区某一类机型。慢多久了，一直慢、越来越慢，还是偶发。三句问完，排查方向基本就定了。为什么管用？它们对应三个坐标：时间、范围、趋势。坐标一定，问题就从"很慢"缩到"某个接口每晚八点开始变慢"。

>

> 还有一笔代价要说清。性能缺陷最贵的地方不在修，在发现晚。功能缺陷上线，坏的是一个功能。性能缺陷上线，坏的是全部用户，而且常在流量最高的时候发作。同一个毛病，在设计阶段改一行，上线后要停机、要发版、要向用户解释。所以这一块，反应快不如记录全。

>

> 留证这件事得提前说。日志、监控曲线、线程栈、连接数，平时不采，出事那一刻现找就来不及。

>

> 有人第一反应是重启。重启能缓解，但现场没了。内存里的泄漏证据、线程栈、连接数，全清空。先保现场，再动手。

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，时间维度直接指向是不是刚改过什么。

- 答"我马上重启一下服务"——能缓解，但重启会毁掉现场。

- 收口：先保现场，再动手。这三句追问的习惯比工具值钱。

易错点 / 考点：① 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，问"线上变慢的正确处置顺序"要答：保留现场、定位、再处置；③ 别把"慢"当成一件事，它至少有一直慢、越来越慢、偶发慢三类，根因完全不同。

过渡：用户的抱怨是表象，下面把表象和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）

左边是人话，右边是机器话。这两栏对不上，排查就只能靠猜。

>

> 外部表现三种。启动系统、打开页面越来越慢，看"越来越"三个字，这是时间上的恶化，指向资源在持续累积。查询数据很长时间才出列表，这是单次操作超时，指向一次请求内部的效率。网络下载速度很低，比如5k/s，单位是速度不是时间，说明瓶颈在传输能力。

>

> 内因三种。资源耗尽，CPU 打到 100%，请求排不上队。资源泄漏，比如内存泄漏。看因果：泄漏本身不立刻出事。它一点点吃掉可用资源，最后才表现成耗尽。资源瓶颈，线程、GDI、DB 连接这些资源变稀缺。

>

> 耗尽和瓶颈差在哪？耗尽是没有了，瓶颈是不够用。线程池两百个线程，进来五百个并发，线程没耗尽，还能建，但上限就在那儿，多出来的只能排队。

>

> 为什么要分这么细？修法完全不同。耗尽要扩容或者限流，泄漏要找到没释放的对象，瓶颈要调参或者改结构。你把泄漏当瓶颈去扩容，只会让泄漏更快吃光新内存。

>

> 三个内因还会连锁。泄漏跑久了变成耗尽，耗尽再触发限流，用户看到的是页面刷不出来。看曲线的时候，往上爬的那一段比崩掉那一刻信息更多。

>

> 送一句口诀：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，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。越来越慢加重启就好，是泄漏的经典签名。

- 答"资源耗尽"——不够准。耗尽是结果，泄漏是原因；耗尽常表现为直接报错，越来越慢说明还没到顶。

- 答"加内存"——只把发作时间往后推，泄漏速率不变。

- 追问：定位第一步看内存曲线是不是锯齿状往上爬。爬，就是泄漏；平的，再考虑瓶颈。

易错点 / 考点：① 泄漏不等于耗尽，泄漏是过程，耗尽或者 OOM 是结果；② 耗尽不等于瓶颈，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，案例题常这么问；④ 记忆抓手：外部表现看时间形态，内因看资源三态。

【思政落点 1】：泄漏最值得琢磨的地方，是每一次都很小。一个请求漏几十 KB，跑一小时看不出来，跑一周才出事。它考的不是谁聪明，是有没有人愿意认真处理这些看着微不足道的地方：分配了就释放，打开了就关闭，加了缓存就想好淘汰。小问题乘上时间，就是大事故。把每个细节做干净，是工匠精神最朴素的样子。

过渡：知道了性能问题长什么样，接下来给它一个严格的定义。

【7.7】什么是性能测试？（3 分钟）

定义这种东西，背下来不难。难的是知道哪几个词会丢分。

>

课件原话：性能测试就是为了发现系统性能问题，或者获取系统性能相关指标而做的测试。一般在真实环境、特定负载条件下，通过工具模拟实际运行和操作。同时监控各项指标，最后分析结果，确定系统性能状况。

>

五个关键词。真实环境：性能跟机器配置、网络、数据量强相关，开发机上测出来的数没意义。真实环境不好搭，最麻烦的是数据量。生产的量造不出来，只能缩小规模再加折算。报告里要写明折算比例。特定负载：一百并发和一千并发，结论完全不同。工具模拟：刚讲过，只能靠工具。监控指标：只施压不监控，你知道慢了，不知道为什么慢。结果分析：工具出图，人出结论。

>

三个目标要背。一，获取系统性能某些指标数据，先知道现状。二，验证系统是否达到用户提出的性能指标。三，发现瓶颈、优化性能。三条是递进的：先知道，再判定，后改进。测试类型也是按目标分的，不是按工具分的。

>

定义里用了一个"或"字，发现性能问题，或者获取指标。这说明摸底也是合法目的。三个目标里，只有第二个要求事先有指标，另外两个可以边测边定。但要做验收，就必须有指标。这个区别，写报告的人最容易含糊。

提问：老板说"给我测一下这个系统性能怎么样"，属于三个目标里的哪一个？要准备什么？

预设回答：

- 答"第三个，找瓶颈"——不一定，这句更像摸底。

- 答"第一个，获取指标数据"——更准确。要准备环境、负载模型、监控项。

- 答"没指标就测不了"——错。摸底本身合法，只是结论只能叫摸底报告。

- 收口：接需求先问一句，这份报告要用来做什么决定。

易错点 / 考点：① 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；③ 常见判断"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了，前者是度量，后者是改动。

> **过渡：**定义里有"特定负载"，负载到底是什么，下面拆成三个要素。

【7.8】负载（Workload）（3 分钟）

> 一千个人同时用，和一千个人轮流用，压力一样吗？差十倍。

>

> 三个要素各管一头。RPS 管压力有多大，SC 管连接资源占多少，思考时间管请求来得有多密。

>

> 第一个 RPS。课件中文写的是"每次请求发送的数据量"，括号里英文是每秒请求数，两个口径不一致。按英文和行业习惯记：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，另算。

>

> 第二个 SC，并发连接数，同一时刻保持的连接有多少条。它和并发用户数不是一回事。一个用户能开好几条连接，一条长连接也能服务多个用户。

>

> 第三个思考时间，用户发出请求之间的间隔。真人点一下、看一眼，才点第二下。工具模拟时忘了这一条，你造出来的不是用户，是机器。同一个脚本，思考时间从一秒改到十秒，压出来能差一个量级。

>

> 课件给了个等式：RPS 加 SC 加思考时间，等于并发用户数。照课件原样背，别自己改写。它想说的是：并发用户数不是孤立数字，由速率、连接规模、间隔共同决定。同样一千个并发，间隔一秒和间隔十秒，压力差十倍。

>

> 真要估算，用"用户数除以思考时间，约等于请求速率"。真实关系比一个加号复杂，这个式子不当计算工具用，当框架用。

> **提问：**同样一千个并发用户，一个每十秒点一次，一个每秒点一次，服务器压力一样吗？

> **预设回答：**

- 答"一样，都是一千人"——最常见也最要命的错。人数相同，请求速率差十倍。

- 答"不一样，间隔短的压得狠"——对。十秒间隔约一百请求每秒，一秒间隔约一千。

- 答"那我把思考时间设成零，更保险"——正好相反，清零不是更保险，是失真。

- 收口：思考时间是负载模型的一部分，不是可有可无的修饰参数。

> **易错点 / 考点：**① 三个名词的英文和中文要能对上；② 课件等式照原样背，同时知道它表达的是"并发用户由三者共同决定"；③ 高频错：并发连接数不等于并发用户数，一个用户能开多条连接；④ 案例题给用户数和思考时间让你估速率，用近似式算，并说明这是估算。

> **过渡：**刚才那句"把思考时间设成零"，下一页专门拆它。

【7.9】思考时间的影响（3 分钟）

> 同一套脚本，改一个参数，结论能从"合格"变成"崩溃"。

>

> 课件给了两个对照：间隔时间长，和间隔时间为零。原文写得很直接。把系统置于同样的高负载，把请求之间的间隔设为零。服务器会立即超载，形成越来越长的等候队列，直至内存耗尽。

>

> 队列为什么会越来越长？把服务器想成一个收银台，每秒能结一百单。来的顾客也是每秒一百个，队伍不长不短。来的是每秒三百个，多出来的两百只能排。消化速度不变，到达速度不变，队伍就只会越来越长。

>

> 队伍在机器里是什么？等待处理的请求对象：线程池的等待队列、连接池的队列、socket 缓冲区。这些都是内存。队列越长，占得越多，最后就是课件说的内存耗尽。到那一步，系统不是慢，是直接崩。

>

> 所以"清零更保险"错在哪？错在它测的不是用户，是洪水。真实用户会看、会想、会走神，思考时间就是那

> 段时间。清零等于假设所有人都在疯狂连点。测出来的数字既不代表生产，也不能拿来定容量。

>

> 什么时候能清零？两种正当用途。做压力测试，目的本来就是逼到崩，看崩在哪。做单接口的极限吞吐，看它最多能跑多少。这两种都要在报告里写清：这是极限值，不是用户场景值。

> **提问：**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，说明扛不住 5000"——得看这 5000 是带思考时间的用户，还是间隔为零的请求流。

- 答"不能，得看场景"——对。同一句话在两种模型下，业务含义差好几倍。

- 答"崩了就是不行"——太粗。崩溃点本身是容量上限，关键是标注在什么负载模型下测的。

- 收口：每个数字后面要跟三个定语，什么场景、什么负载模型、什么环境。

> **易错点 / 考点：**① 后果链条要能完整说：到达率超过处理率，队列增长，内存占用上升，耗尽或崩溃；② 常见判断"思考时间越短，测试越严格"——错，它改的是负载模型，不是严格程度；③ 案例题问"为什么压测结果和线上差很多"，标准答案之一是思考时间和实际不符；④ 记住"队列"这个词，响应时间飙升、超时率上升，多数是队列在起作用。

> **过渡：**思考时间定了，接下来定怎么把这些请求铺到时间轴上。

【7.10】负载模式（2 分钟）

> 一条平线，一条心电图。你的业务像哪一种？

>

> 课件给了两个维度。第一个，加载的循环次数或者持续时间，也就是这套脚本跑几轮、跑多久。它决定你能撞见什么问题。跑十分钟，只看得到稳态。跑八小时，才可能撞上内存泄漏、连接不释放、日志涨满、缓存击穿这些随时间累积的毛病。所以时长要对着目的写，验证型跑短，渗入型跑长。

>

> 第二个，加载的方式。均匀加载是压力从头到尾恒定，像平线，测稳态吞吐和资源占用。峰值交替是高、低、

> 高、低来回突变，像心电图。它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，会不会把库打穿。

>

> 两个维度还能组合。短跑配均匀，测稳态；长跑配均匀，测泄漏；短跑配峰值交替，测扩容反应。三个组合，对着三类问题。

>

> 模式为什么这么要紧？真实业务从来不是均匀的。早九点打卡、中午订餐、晚上看视频、零点跑批，全是尖峰。平均负载下很稳，不代表尖峰来了不崩。

>

> 平均值会骗人，峰值不会。设计时先回答一句：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。把这几个组合写成一张场景表，比嘴上说"压一下"强得多。

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。

- 答"还得看是哪个时间段"——补得好。业务有时间规律的系统，要按时间窗分别看。

- 收口：设计负载模式，就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，很多人只记得跑多久，忘了怎么加载；② 均匀加载和峰值交替各自适合什么问题要能对上；③ 平均值对分位数是高频概念，报告只看平均值是典型错误；④ 时长要和测试类型匹配，别用十分钟的数据论证"能连跑一个月"。

过渡：概念铺完，看一个具体例子，把这些参数落到场景上。

【7.11】示例：系统负载测试（2 分钟）

报告上那条曲线，不是一次跑出来的。

>

> 一次负载测试有四样东西：被测系统、负载模型、施压脚本、监控与结果。被测系统要放在接近真实的配置上。负载模型就是刚才那三个要素加两种模式。施压脚本把用户操作翻成请求序列：登录、查列表、下单，中间夹着思考时间。监控采集响应时间、吞吐量、错误率、各项资源指标。

>

> 四样缺一样都会失真。缺监控，你只知道慢；缺负载模型，数没法跟别人比；缺脚本，这次跑的结果复现不出来。

>

> 结果最典型的呈现，是一条随并发上升的曲线，一般分三段。轻载区，并发增加，响应时间基本不变，吞吐量线性上升，说明还有余力。拐点附近，吞吐量涨得慢了，响应时间开始抬头，这是接近饱和的信号。过载区，再加并发，吞吐量不升反降，响应时间飙升，错误率冒出来。

>

> 最想拿到的是拐点那个数。它告诉你这套配置的经济容量在哪。报给业务方的不是"最大能撑多少"，是"响应时间还达标的前提下能撑多少"。这两个数常常差很远。

>

> 还有个细节：曲线是阶梯加压跑出来的。每加一档，稳定跑一段，记一组数。档位要密一点，拐点才看得清。所以一次负载测试是多次短跑拼起来的，不是一个脚本从头跑到尾。

提问：曲线在过载区吞吐量下降，为什么加了更多人反而干得更少？

预设回答：

- 答"服务器忙不过来"——太笼统。机制是上下文切换和锁竞争，CPU 时间花在切换和抢锁上。

- 答"队列满了开始拒绝"——那是错误率上升，吞吐下降在拒绝之前就发生了。

- 答"内存不够开始换页"——也可能，特别是对象大量堆积时。

- 收口：过载区的吞吐下降，是系统在做无效功的信号。

易错点 / 考点：① 三阶段的名称和特征要能画能说；② 高频错答"并发越高吞吐越高"，只在线性区成立；③ 报容量必须带响应时间前提，只报并发数没有意义；④ 阶梯加压的做法：每档加压后稳定一段时间再采样，

否则采到的是爬坡过程的数据。

> **过渡：**负载测试只是一种，课件按目的把性能测试分成了好几类。

【7.12】性能测试类型（3 分钟）

四种类型，名字里都带"性能测试"，考试最爱在这儿设坑。

>

> 第一种，性能基准测试。在系统标准配置下拿到指标数据，作为将来改进的基线。基线的作用是有得比。今天首页八百毫秒，优化后五百毫秒，这三百毫秒的改进是靠基线才看得见的。没有基线，所有优化都只是"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指标。这五个字要紧：事先已定义。指标得在测试之前写下来，不能测完了往回找。

>

> 第三种，性能规划测试。在多种特定环境下拿到不同配置的指标，决定部署的软硬件选型。说白了就是买几台、什么规格。四核八 G 撑五百并发，八核十六 G 撑一千一，明年业务翻倍买哪个，这就是规划要回答的。

>

> 第四种，容量测试。课件有句话：容量可以看作系统性能指标之一，所以它算性能测试的一种。容量是这套系统最多能承载多少用户、多少数据、多少并发。它和验证测试的差别：验证问达不达标，容量问上限在哪。

>

> 选哪种跟着问题走。想知道现状用基准，想签验收用验证，想买机器用规划，想报上限用容量。基准还有个前提：配置、数据量、脚本要一致，不然两次数值不能比。

>

> 四个词收一下：基准记现在，验证对标准，规划定选型，容量摸天花板。

> **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？

> **预设回答：**

- 答"先做验证测试"——可以，但你得先有指标，而指标常常要靠基准数据来定。

- 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"——顺序偏了。它成本最高，基本情况不明时容易白烧时间。

- 收口：基准、验证、按需容量和规划。基准排在前面。

> **易错点 / 考点：** ① 四种类型的名称和目的要成对背下来，简答常考；② 基准测试的关键词是 Baseline，答题要写出来；③ 容量测试和验证测试的区别是高频混淆点，一个问达标吗，一个问上限在哪；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，别答成独立门类。

> **过渡：** 上面四种是常规动作，下面三种是加码动作，专门把问题逼出来。

【7.13】性能测试类型（4 分钟）

> 有三个类型，目的不是验证，是让系统说真话。

>

> 压力测试。模拟实际应用的软硬件环境和用户使用过程。高负载、异常负载、超长时间运行，加速系统崩溃。检查它对异常情况的抵抗能力，找出瓶颈和不稳定。四个字要紧：加速崩溃。目的不是证明它稳，是用最短时间把它逼崩，看它怎么崩、崩在哪。

>

> 渗入测试，soak。通过长时间运行，让问题一点点渗透出来，发现内存泄漏、垃圾回收或者别的问题，检验健壮性。关键词是长时间。很多毛病十分钟里根本不出现：内存一点点涨，连接一点点漏，临时文件一点点堆。soak 把几个月的缓慢过程，压到能观测的时间里。

>

> 峰谷测试，peak-rest。高低突变加载：先压到高水平，急剧降下来，平息一段，再压上去，反复。课件说它容易发现问题的蛛丝马迹，最后找到根源。为什么突变能暴露问题？系统在扩和缩的时候最脆弱。缓存刚清空就来了大流量，连接刚回收就要重建。

>

> 分工对照：压力问极限在哪，渗入问跑久了坏不坏，峰谷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。

>

> 还有一处容易混。压力和渗入都涉及长时间，差别在目的：一个求崩，一个求漏。

> **提问：**系统上线三个月越来越慢、每周要重启一次，用三种类型里的哪一种去复现？

> **预设回答：**

- 答"压力测试，压到崩"——能看到崩溃点，复现不了三个月慢慢变慢的过程。

- 答"渗入测试，长时间跑"——对。用几十小时把三个月的累积压出来。

- 答"峰谷测试"——峰谷测突变响应，不测累积恶化。

- 收口：判断标准是线上问题的时间形态，累积选渗入，尖峰选峰谷，崩溃选压力。

> **易错点 / 考点：**① 定义关键词要能默写：压力是高负载、异常负载、超长时间加加速崩溃；渗入是长时间加内存泄漏和 GC；峰谷是高低突变加重复；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，不只有加并发；③ 渗入和压力的区分，求崩对求漏，判断题常考；④ 峰谷容易被忽略，它是潮汐业务的标配。

> **过渡：**压力测试具体在做什么，看一个示例。

【7.14】示例：压力测试（2 分钟）

> 压测跑完只写一句"系统崩了"，这份报告等于没写。

>

> 压力测试的做法，是把负载一路加上去，直到系统异常，同时记录每一步的表现。所以它产出的不是一个数字，是一组临界值和一串现象。要看四件事。

>

> 这四件事最好压测之前就列好。压完再想，很多现象已经过去了。

>

> 一，崩溃点在哪。多少并发、多少 TPS 开始出错，错误是超时、是 500、还是连接被拒。错误形态指向不

> 同瓶颈。超时多半是下游慢，500 多半是异常没兜住，连接被拒常常是连接池或者句柄见底。

>

> 二，崩之前有没有征兆。响应时间是慢慢爬上去的，还是突然跳起来的。慢慢爬说明资源在逐步消耗，突然跳往往撞了某个硬上限，比如线程池满、GC 停顿。

>

> 三，崩完能不能自愈。撤掉压力，几秒钟恢复，还是彻底僵死必须重启。这一条最反映健壮性。能自愈，事故是抖动；不能自愈，事故是停机。记录要带时间戳，响应时间抬头和某个资源见底才容易对上。

>

> 四，异常负载下的行为。课件特意写了"异常负载"。不只是量大，还有形态异常：请求体超大、参数畸形、依赖服务超时、网络抖动。这些才是线上最让人措手不及的。

>

> 最后一句：压力测试是主动把系统搞崩，一定要在隔离环境里做。共享测试环境里跑，同事的测试全被你带走。

提问：压力测试把系统压挂了，撤掉压力它自己恢复了，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"——不能这么简单。要看恢复时间和恢复后的一致性。

- 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。

- 收口：结论永远是"在什么条件下崩溃、什么表现、撤压后几秒恢复"，不是一个通过与否。

易错点 / 考点：① 目的在找瓶颈、验健壮性，不在"通过"；② 自愈要看三个维度：时间、数据、性能有没有复原；③ 常见判断"压力测试放在共享测试环境做，省资源"——错，它会把环境打挂，影响别人；④ 异常负载包含形态异常，不只有数量异常。

> **过渡：**性能问题可能出在哪一层，下面给一张六层的地图。

【7.15】性能测试的层次（3 分钟）

> 一个请求从手指落下到数据回来，中间要过六道关。
>
> 一，Web 和 App 客户端。用户最先感知：渲染慢、图片太大、脚本阻塞、首屏等好几秒。特点是换台差一点的手机就暴露，所以客户端性能常在低端机上测。
>
> 二，应用服务器。业务逻辑在这里跑：线程池、运行时、GC、对象创建、锁竞争、序列化。典型症状是 CPU 高、GC 频繁、线程阻塞。
>
> 三，数据库访问层。它和数据库是两层。访问层指的是连接池、ORM、SQL 生成、事务边界。很多"数据库慢"就是这一层：连接池太小排队，事务开太大锁等待，ORM 生成 N+1 查询。
>
> 四，RDBMS 数据库。看索引、执行计划、锁、慢查询、表的数据量。数据量最容易被忽略，一万行和一亿行，同一条 SQL 命运完全不同。
>
> 五，端口连接。底层但常见：TIME_WAIT 堆积、端口耗尽、连接复用没做、keep-alive 配错。典型症状是服务本身很闲，客户端却连不上。
>
> 六，网络通信。带宽、延迟、丢包、DNS 解析、TLS 握手。跨地域部署的系统，这一层常是主要矛盾。
>
> 为什么要拆这么细？定位靠逐层排除。先看客户端是不是自己慢，再看服务端处理时间，再看是不是在等库，最后看网络。心里有这六层，排查才有顺序。
>
> 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。往下游依赖、连接、网络去找。

> **提问：**用户说页面打开要 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 CPU 低恰恰说明瓶颈可能在等待上，等 IO、等网络、等下游。

- 答"先查客户端"——合理，8 秒这个量级，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 8 秒拆开：DNS、建连、服务端处理、传输、渲染。

- 收口：把一个请求的耗时切成段，才知道去查哪一层。

> **易错点 / 考点：**① 六层名称要能按顺序说出；② 数据库访问层不等于 RDBMS，前者是连接池、ORM、事务，后者是数据库引擎本身，这是最容易合并的错；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；④ 定位方法是逐层排除，答题要把分层思路说出来。

> **过渡：**六层是一张横切面，再看一张纵切面，性能在项目一生里怎么分布。

【7.16】全生命周期的性能测试（2 分钟）

> 性能不是一次测试，是一条一直在转的线。

>

> 项目一开始是性能需求的收集与定义。目标在这一步写下来，不能等做完再补。接着是性能建模和设计阶段的内建性能：把性能当设计约束，不是事后补丁。选什么结构、要不要缓存、要不要读写分离，这些决定的影响比任何调优都大。

>

> 越往前走越便宜。需求阶段改一句指标，编码阶段改一行代码，上线后要停机发版。

>

> 编码阶段做单元性能测试和代码优化，把关卡前移。一段平方级复杂度的代码，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，产出性能数据。

>

> 上线之后是监控和容量管理：7×24 的监控、健康检查、容量规划。课件图上"性能数据"在两个环节之间来

> 回走。意思是测试数据要喂给监控和容量管理。那边发现的新问题，又回流成新的性能需求。

>

> 测出来慢，还得有人定位、有人优化。课件把定位瓶颈和优化放在测试后面，那是一段路，不是一个动作。测试只负责把问题摆出来。

>

> 环中间写着经验积累。每转一圈，团队对"这套系统什么情况下会慢"就多一层判断。这份东西长在人身上，不长在文档里。

>

> 这个闭环的意义在哪？性能是持续演进的，不是一次交付的。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。

>

> 所以图上那个"性能需求调整修改"的回头箭头是正常的，不是失败。需求会被现实改写。

提问：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。

- 答"补性能需求"——也成立，指标得先有。

- 收口：实操顺序是监控、摸底基准、定需求、针对压测。缺了监控，连真实负载模型都画不出来。

易错点 / 考点：① 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 内建性能这个词要会解释：在设计阶段就把性能当约束；④ 闭环里的回流也要知道，需求被调整是常态。

> **过渡：**闭环的起点是需求。一句"要快"怎么变成可测的数？

【7.17】从需求开始明确性能（3 分钟）

> "系统要快一点"，这句话能写进合同吗？写不进。

>

> 课件给了四类来源。第一类，最终用户的体验，2-5-10 原则。两秒以内觉得快，五秒左右还能忍，十秒基本关页面走人。它不精确，但抓住了要点：性能是体验的一部分，不是机房里的数字。第二类，商业需求，"比竞争对手的产品好"。听着虚，有实际用处，它给了外部参照，你得测对手才能把"好"变成一个数。第三类，技术需求，CPU 不超过 70%。工程口径，好测好监控，但它和体验没有直接对应。第四类，标准要求，数据通信延时小于 80 毫秒，来自行业规范或者合同，是硬约束。

>

> 然后是三种视角，这条最值得记。用户关注外部表现，比如响应时间。业务部门关注容量和数据吞吐量。技术团队关注资源占用率和使用效率。

>

> 同一套系统，三种人问三个问题。这也解释了一个常见矛盾：技术说资源还很闲，业务说高峰撑不住，用户说卡。三方都没说谎，看的是不同层面。

>

> 四类来源常常打架。用户要快，业务要省，技术要稳，标准要合规。评审就是把打架的地方摆到桌面上。

>

> 所以课件最后那个动作：获取、分析、评审性能需求。先收集各方嘴里的"要快"，再翻成可测指标、检查有没有冲突，最后让三方坐一桌确认。别等测完才说"我不是这个意思"。

> **提问：**技术团队说资源只用了 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"技术对，数据说话"——数据是真的，口径不对。30% 是平均值，双十一看峰值。

- 答"业务对，他们更懂业务"——感觉对，但业务说不出瓶颈在哪。

- 答"两个都对，看的是不同层面"——最准确。资源占用率、容量、响应时间，是三个不同的问题。

- 收口：峰值吞吐、响应时间、资源上限，三项都得定。

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 原则是经验阈值，不是精确标准；③ 高频混淆：技术指标不等于用户感受，两者要分别定义；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那一步。

过渡： 来源有了还不够，指标还得能测。

【7.18】性能需求可测试性（3 分钟）

"每天要处理一千万笔交易"，听着很硬，真拿去测却测不了。

>

为什么没法测？四点。它没说一天是二十四小时还是八小时营业。它没说交易的分布，均匀铺开还是挤在几个时段。它没说响应时间，每秒处理得再多，每笔要等三十秒，用户照样骂。它没说峰值。

>

改写过的版本长这样。先一句清楚而量化的性能指标作总纲。然后落到每秒平均处理约 130 笔。再加一条：高峰期间是平均值四倍，也就是每秒 520 笔。

>

好在哪？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。场景就能设计了。平时按 130 跑稳态，高峰按 520 跑尖峰。两条曲线都要落在响应时间约束内。

>

顺手验一下。一千万摊到 86400 秒，大约 116 笔每秒。课件给 130，略高，说明它按实际服务时间折算。课件没展开这个口径，答题按课件数字写，别自己另算一套。130 乘 4 正好 520，四倍是配套的。

>

可测试的标准三条。能定量，有数字有单位。有前提，什么负载、什么环境、什么数据量。可判定，测完能明确说达标或者不达标。三条缺一条，需求还得接着谈。

>

"平均"是性能里最危险的词。系统从来不死在平均值上，死在峰值上。

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。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。一千个用户带不带思考时间，压出来天差地别。

- 收口：一条完整的需求至少四要素，并发规模、响应时间、错误率、持续时间。

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 平均值对峰值是高频考点，答题一定要提峰值；③ 可测试三标准可以当简答模板；④ 案例题给一句模糊需求让你改写，补四个要素：并发、响应时间、错误率、持续时间。

【思政落点 2】：可测试性背后是一条职业底线。指标定两秒，实测 2.8 秒，报告里写成"基本满足用户感知"，技术上叫口径调整，职业上叫不诚实。工程判断允许有弹性，数据不能修饰。差 0.8 秒就写差 0.8 秒，再说明原因和风险。报告里每个数字都是你信誉的一部分，敢写不好看的数字，才配写要紧的报告。

过渡：需求定了、指标量化了，接下来要在设计阶段把性能建进去。

【7.19】架构设计评审（2 分钟）

一行代码没跑，性能上限已经被定掉大半。

>

为什么？结构决定物理可能性。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。水平扩展的结构，一开始就留了加机器的口子。库不做读写分离，读请求再多也只能压一台。没有缓存层，所有请求直击数据库。这些选完，后面代码写得再漂亮也翻不了盘。

>

评审便宜，因为改图纸不用停机。等上线再改结构，代价是重做一遍，还要迁移数据。

>

评审看什么？归成四问。一，数据放哪、怎么涨。增长曲线决定要不要分库分表、要不要冷热分离。二，请

> 求怎么分流。有没有缓存、限流、降级、异步。三，瓶颈会移向哪儿。这一问最见水平：把库优化好，瓶颈跑到应用服务器；应用加了机器，瓶颈跑到网络和连接数。评审就是提前知道下一个瓶颈在哪。四，可测性够不够。有没有埋点、有没有指标、测试环境能不能复现生产规模。这一条最容易被漏，却直接决定你后面能不能测。

>

> 还有个提问方式比"这样行不行"管用：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，它逼着设计者把整条链路走一遍。

>

> 所以性能问题最好的修法，是在图纸上修。结构定错了，调优只能在很窄的范围里挪。

> **提问：**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，结构问题补不了。

- 答"不能，架构决定天花板"——抓住了。调优是在天花板下面找空间。

- 答"要看问题大小"——更准确。参数级问题调优能救，结构级问题只能改结构。

- 收口：评审时要分清这是参数问题还是结构问题。

> **易错点 / 考点：**① 架构评审属于静态性能测试；② 瓶颈会转移这个概念要能举例；③ 常见判断"性能优化主要在编码和调优阶段"——不完整，架构阶段影响最大；④ 四问里可测性最容易被漏，答题补上能加分。

> **过渡：**结构定了，代码还得写，下面看代码里最常见的那些性能坑。

【7.20】代码评审 - 性能问题（3 分钟）

> 一个循环里查一次库，功能全对。这算缺陷吗？

>

> 课件给了两个抓手：一个具体例子，一份清单。例子是"使用 String 的性能是最差"。这句话得放在语境里：它说的是字符串频繁拼接的场景。Java 里字符串不可变，每次加号都建新对象。拼一万次就是一万个对象，

> 内存和 GC 全压上来。换成 StringBuilder，只维护一个可变缓冲区。这类毛病的特点是单次看不出，循环里要命。

>

> 清单是"35 个 Java 代码性能优化总结"。不逐条念，讲它们落在哪五类。一，对象创建：循环里 new、装箱拆箱、正则重复编译。二，集合使用：没定初始容量反复扩容。数据结构用错，用 Map 能查到的场景拿了 List 去遍历。三，IO 和数据库：循环里查库、没批处理、没关流。四，并发：锁粒度太大、无谓同步、线程池参数不合理。五，字符串和序列化：刚才那个拼接，还有频繁的 JSON 转换。

>

> 为什么评审能抓性能？成本低。不搭环境、不造数据、不跑压测，两个人对着代码看半小时，可能就拦下一个会在生产上炸的循环查库。而且抓得早，写代码当天就能改。

>

> 局限也要说清。它只看得出"能看出来"的，看不出这段逻辑在一千并发下会怎样。所以它和压测互补：评审管明显不能这么写，压测管这么写到底行不行。

>

> 还有一句：性能缺陷要带规模。循环三次无所谓，三千次就是灾难。

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"——功能角度确实不算，性能上它是典型的 N+1，数据量一大就崩。

- 答"算性能缺陷"——准确。这类问题在代码评审阶段就不能放过。

- 答"要看循环多少次"——很关键。三次无所谓，三千次是灾难。

- 收口：性能缺陷的判断要带规模，说清在什么数据量下会出问题。

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 条有用，答题给分类框架得分更高；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，缺陷报告里常在这扣分。

过渡：需求有了、架构评审过了、代码也看过了，接下来才是动手设计与执行。

【7.21】性能测试设计与执行（3 分钟）

> 压测报告上写着平均响应 200 毫秒，这份报告你敢直接拿去上会吗？数字对不对，取决于它是在什么环境里测出来的。

>

> 课件把环境条件拆成六项。网络和配置排第一。带宽多大、延迟多少、中间有没有代理和防火墙。内网压出来的 200 毫秒，用户在地铁上用 4G 可能是两秒。链路不一样，数字就不是一个数字。

>

> 第二项，服务器硬件选型。几核、多大内存、什么磁盘。机械盘和 SSD 在数据库场景下差一个数量级。这一项要跟生产对上，对不上，结论就外推不出去。

>

> 第三项，被测试系统部署。部署几个实例、怎么分布、前面有没有负载均衡、有没有缓存。最容易出的错是测试环境一个实例，生产八个。那你测出来的容量，只有真实的八分之一。

>

> 第四项，数据库数据量。这一项最容易被跳过，也最伤人。测试库一百万行，生产库一亿行。同一条 SQL，在测试库走索引十毫秒，到生产库全表扫十秒。数据量是个隐形变量，得专门造。

>

> 第五项，测试机器池。它的活儿是产生虚拟用户和负载。可施压的机器自己会不会先到顶？一台机器跑几千并发，CPU 或网卡先满，你测到的其实是压测机的极限，不是被测系统的极限。所以大规模压测要多台施压机，这就是后面要讲的分布式。

>

> 第六项，SUT 监控。SUT 是被测系统。监控要盖三层：施压侧看发了多少请求、成功率多少；被测系统侧看 CPU、内存、GC、线程、连接池；依赖侧看数据库、缓存、消息队列。只监控一层，你只知道疼，不知道哪儿疼。

>

> 课件还画了一张部署示意图。左边测试机池，中间分析器，右边 Web 服务器、应用服务器、数据库服务器，框在一起就是被测系统。这张图要你看出的是一件事：施压的、被测的、观测的，物理上尽量分开。混在一台机器上，它们互相抢资源，数据就失真了。

> **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"按比例换算就行"——换算不成立。有些瓶颈不是线性的，单机连接数上限、单库锁竞争，都不是除以八能推出来的。

- 答"不行，容量只有八分之一"——方向对，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、拓扑一致"——最接近专业做法。拓扑要一致，规模可以缩，数据量和并发按目标折算。

- 收口：课件把部署和数据量单列成两项，就是在提醒环境失真、结论全废。

> **易错点 / 考点：**六项要素要列全，数据库数据量和测试机器池最容易被漏。判断题"压测环境可以缩水"是错的，拓扑要一致、数据量要接近、并发按目标设定。施压机自己到瓶颈，是典型的测试失效原因，案例题常从这里出。SUT 要会解释：被测系统。

> **过渡：**环境搭好、脚本跑起来，接下来要看清时间花在哪一段。

【7.22】测试执行和监控（3 分钟）

> 用户在 4G 上点一下，八秒才出结果。你打开服务器日志，处理只用了 0.2 秒。剩下那 7.8 秒去哪了？

>

> 不知道去哪，就没法优化。所以第一个动作是把一次请求切成段。课件给了一串时间名词，按"按下回车到看见结果"的顺序串一遍就顺了。

>

> 先是 DNS 时间。域名解析成 IP 要多久。别小看它。DNS 配错或者解析慢，用户先卡两秒，而你服务器一点毛病没有。

>

> 然后是网络时间。请求从客户端走到服务器要多久，响应再走回来要多久。跨地域、跨运营商的系统，这一段经常是大头。

>

> 再是服务时间。服务器内部处理这次请求花了多久。这一段最值得深挖，因为它还能继续拆。拆成应用处理返回时间，也就是业务代码自己算的部分；和数据库时间，也就是等数据库返回的部分。数据库时间往往占服务时间的大半。所以"应用慢"，十次有八次其实是"数据库慢"。

>

> 接着是数据返回时间。响应体从服务端推到客户端的时间，跟响应大小成正比。返回一个 10 MB 的 JSON，服务器再快也救不了体验。

>

> 最后是客户端时间。浏览器或 App 拿到数据之后，渲染、跑脚本、加载图片。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？因为定位性能问题，就是找哪一段最长。总响应 8 秒，DNS 0.1 秒、网络 0.3 秒、服务 7 秒、返回 0.2 秒、渲染 0.4 秒。问题在服务端，而且具体在数据库。不拆，你只能对着"8 秒"发愁。

>

> 还有个方法论上的提醒。监控必须和执行同时做，不能等跑完再补。性能问题是瞬时现象，你不在那一刻采数据，它就消失了。

>

> 再补一层。设备本身也在变。同一个接口，在电脑上测和在一部三四年前的手机上测，数字能差出几倍。所以看响应时间，得先问一句：在哪台机器上、什么网络下测的。

> **提问：**总响应 3 秒，你怎么判断是服务端慢还是网络慢？给你什么数据能判定？

> **预设回答：**

- 答"看服务端日志里的处理时间"——最直接的证据。服务端自己记的处理耗时 0.2 秒，那 2.8 秒就在路上或者客户端。

- 答"ping 一下看延迟"——能参考，但 ping 走 ICMP，跟 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。

- 收口：所以埋点和分段计时是基础设施，不是可选项。

> **易错点 / 考点：**五个时间段的名称和顺序；服务时间包含应用处理时间和数据库时间，这个包含关系常考。高频错答是"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。监控必须和执行同时做，事后补的数据无效。

> **过渡：**这么多指标靠人手工采不可能，得看工具。

【7.23】性能测试工具（2 分钟）

> 工具清单摆出来，几十个名字。常用的其实就那两三个。所以这里不背名单，讲怎么分类，你以后遇到没见过的工具也知道往哪儿放。

>

> 课件分两大类：客户端性能测试工具，服务器端性能测试工具。这个分法对应前面讲的层次，一层是用户手上那台设备，一层是你机房里的服务器。两边测的东西根本不是一回事。

>

> 客户端这一组，课件列了 Google Lighthouse、PerfDog、Monkey、Monkeyrunner、mobileperf、Pyroscope、MemoryLeakDetector。它们关心的是用户手上那台设备：启动多久、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 和 Monkeyrunner 是 Android 的稳定性压测，用随机事件流去撞崩溃。PerfDog 是移动端性能采集，下一节专门讲。Pyroscope 偏性能剖析。MemoryLeakDetector 查内存泄漏。

>

> 服务器端这一组，JMeter、LoadRunner、Webload、Gatling、k6、Vegeta、Locust、Skywalking。这些还能再分两类。施压类：JMeter、LoadRunner、Gatling、k6、Vegeta、Locust，负责造并发、发请求。观测类：Skywalking 这类 APM，负责看链路、看耗时分布。

>

> 两类必须配合。只有施压，你知道系统崩了，不知道为什么崩。只有观测，你知道哪儿慢，可压力从哪来、能扛多少，还是不知道。

>

> 选型的原则：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，课件后面专门讲它。LoadRunner 是老牌商业工具，脚本能力强，报表成熟，代价是授权贵、部署重。

>

> 怎么用这张分类表？先把被测对象定下来。测网页，两边都要，客户端那组看页面，服务器那组看容量。测 App，客户端那组是主力。先定对象，再挑工具，顺序别反。工具装错类别，等于用称体重的秤去量身高，数字有了，意思没有。

> **提问：**只允许你装一个服务器端工具，你选哪个？

> **预设回答：**

- 答"JMeter"——最常见也最合理。开源、协议支持广、有界面也有命令行、能分布式、插件多。

- 答"LoadRunner"——能力更强，授权贵、部署重，学习和维护都重。

- 答"Locust，能写 Python 脚本"——脚本即代码，适合研发自测。

- 收口：选型看三件事，协议支持、能不能分布式、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：**两类工具的划分和代表产品要对上。施压类和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"。Monkey 是稳定性压测，用随机事件，不是性能指标采集。"下列哪个不属于服务器端工具"这类选择题，名字要认得。

> **过渡：**客户端工具里，课件单独拎出一个来讲。

【7.24】客户端 PerfDog（2 分钟）

> 手机在你手里很顺，这不能说明客户端性能没问题。你自己的手机往往是中高端机，缓存是热的，后台也干净。用户的手机不是。

>

> 客户端性能要单独测，因为它的资源和服务器完全不是一套。手机内存按 GB 算，CPU 会降频，电量有限，网络是移动网络，还会在 Wi-Fi 和 4G 之间来回切。服务器那套 TPS、并发数，描述不了客户端的问题。客户端有自己的一套指标：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 干的事很直白。手机用 USB 连到电脑，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。课件给的两条参考是教材 P201 到 203 页，和 PerfDog 的官网 perfdog.wetest.net。安装步骤、界面按钮、报表导出，这些随版本会变，以教材和官网为准，课上不逐条念。

>

> 用法上有三条比记按钮重要得多。第一，用真机。模拟器跑出来的帧率和内存，跟真机差得很远。第二，用低端机测一遍。高端机流畅，不代表用户流畅。性能测试测的是最差的那台设备，不是你自己那台。第三，场景贴近真实。冷启动、页面来回切、列表快速滑、弱网下操作，这些才会把问题逼出来。

>

> 还有一条对照原则，最容易被忽略。同一个操作，改前和改后各测一遍，才有意义。单看一个"启动 1.8 秒"，你判断不了好坏。得知道基线是多少、目标是多少。

>

> 客户端还有个特点，问题常常是累积出来的。刚装上的 App 很快。用一个月，缓存堆起来，就慢了。所以长时间使用这个场景不能省。

提问：App 在你手机上很流畅，能不能说明客户端性能没问题?

预设回答：

- 答"能，我用着挺顺"——你的手机往往是中高端机，缓存是热的，后台也干净。

- 答"不能，要用低端机测"——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。

- 收口：客户端和服务器一个道理，测最差的条件，不测最好的条件。

> **易错点 / 考点：**客户端指标和服务器指标不能混用，这是最容易合并的错。真机和模拟器的差别是高频判断点。"客户端性能测试要不要改代码埋点"，PerfDog 这类外部采集不用，但精细的业务埋点仍要开发配合。没有基线的指标只是一串数字。

> **过渡：**工具装上、脚本跑起来，接下来是整章最难的部分。

【7.25】性能测试工具 JMeter（2 分钟）

> 先说结论：JMeter 的七个元件，考试要考，干活也绕不开。这七个词理顺了，工具就算通了。

>

> JMeter 是什么？开源、Java 写的、以施压为核心。官网在课件上，jmeter.apache.org，下载和文档都在那儿。

>

> 七个元件按从外到内讲。最外层是测试计划，一次压测的全部内容都装在里头，相当于一个项目。

>

> 往里是线程组，这是最关键的元件。它定三件事：多少个用户、怎么并发、跑多久。线程数就是虚拟用户数。Ramp-Up 是多少秒把用户全放出来。循环次数或者持续时间，决定跑多久。负载模型主要就写在线程组里。

>

> 线程组里面是取样器，也叫请求。HTTP 请求、JDBC 请求、FTP 请求，发请求的就是它。它决定你去打谁。

>

> 逻辑控制器管顺序和条件。循环、如果、事务、随机。它决定请求按什么顺序组合。

>

> 断言判断响应对不对。这一条最容易被省掉。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，报告漂亮得像假的。

>

> 前置处理器和后置处理器，在请求前后做加工。前置准备参数，后置把响应里的 token 提出来，传给下一

> 个请求。

>

> 监听器收集和展示结果。查看结果树、聚合报告、响应时间曲线。它负责看，不负责压。

>

> 一句话串起来：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。

>

> 还有个层级问题容易忽略。断言和处理器挂在取样器下面，不是挂在测试计划下面。挂错层级，它管的范围就变了。本来只想管一个请求，结果管了全部。

> **提问：**一次压测跑完，报告显示平均响应 30 毫秒、错误率 0，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"错误率 0"是工具只看了 HTTP 层有没有响应。

- 答"不能，得看有没有断言"——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，不是一个意思。

- 收口：读报告先看三个前提，负载多大、断言有没有、跑了多久。

> **易错点 / 考点：**七个元件的名称和职责要能对上，简答高频。线程组是负载模型的载体，这句话必须会说。监听器不是不能加，准确说法是图形界面下的监听器开销大，正式压测走命令行并简化监听器。断言判断业务正确性，不判断性能。

> **过渡：**七个元件在界面上怎么组织，看那棵树。

【7.26】JMeter 构成（2 分钟）

> 装完 JMeter，打开界面，左边那棵树第一次看是懵的。一层套一层，不知道哪个放在哪儿。这一节把树看明白。

>

> 最上面是工作台，里面放非测试元件。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。判断依据很简单：它让脚本从无到有，不是让请求发出去。

>

> 往下是测试计划，要跑的内容都挂在它下面。

>

> 测试计划下面主要是线程组。线程组里有线程，一个线程就是一个虚拟用户。线程组还能配 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，别跟前置处理器混了。

>

> 线程组之外，测试计划下还能挂 Test Fragment。一段可以被复用的脚本，本身不执行，得被别的元件调用才跑。还有远程运行控制器，管分布式压测的从机。还有别的杂项。

>

> 线程组内部才是上一页那套元件：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。这里多了两个前面没提的。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，思考时间就设在这里。

>

> 工作台里还有三个辅助工具。Log View 看运行日志，Templates 是模板，能直接套一个现成的测试计划，函数助手生成随机数、时间戳这类函数，写脚本常用。

>

> 这棵树记四句就够：工作台管录制，测试计划管全局，线程组管负载，元件管细节。

>

> 层级还有个用处，报错时定位。元件挂错地方，症状往往是明明配了却不生效。这时候别急着加配置，先看它挂在哪一层。层对不上，改多少遍都一样。

提问：思考时间在 JMeter 里设在哪？设错地方会出什么问题？

> **预设回答：**

- 答"定时器"——常用固定定时器和随机定时器，后者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"——断言判断结果，和节奏无关。

- 收口：元件是按职责分层的，职责搞清，就知道往哪儿配。

> **易错点 / 考点：**工作台放的是非测试元件，录制用，这是常考判断。setup 和 teardown 线程组跟前置、后置处理器的区别：前者在线程组层面跑一次，后者在每个请求前后跑。配置元件提供公共配置，前置处理器做请求前的加工，两者容易混。Test Fragment 不独立执行，必须被引用。

> **过渡：**元件讲完，看它能打哪些协议。

【7.27】JMeter 支持的协议（2 分钟）

> 有人问，JMeter 能不能压我们公司自研的二进制协议。能。但理由不在协议清单上，在它留的那个扩展口。

>

> 先把清单过一遍，按用得多少排。

>

> 第一，Web，HTTP 和 HTTPS。绝对主力，绝大多数压测都是打 HTTP 接口。

>

> 第二，SOAP 和 REST Webservices。企业系统里这两类接口常见，特别是跟外部系统对接的时候。

>

> 第三，FTP。文件上传下载。

>

> 第四，Database via JDBC。这条重要。JMeter 能直接连数据库发 SQL，不经过应用层。用途有两个：单独测数据库的性能，比如一个复杂查询要多久；给压测造数据或者校验数据。后面实验会要求用 JDBC 连接做预处理，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。

>

> 第七，Mail，SMTP，邮件服务。

>

> 第八，JSR223。这条最容易被低估。它允许你用脚本语言，Groovy、JavaScript 都行，写自定义逻辑。工具的能力边界基本由它决定。协议不支持？自己拼。

>

> 第九，TCP、Java Objects 这些。TCP 层面能测自定义协议，Java Objects 用于 Java 序列化的远程调用。

>

> 这张清单的意思不是让你背九个名字，是告诉你 JMeter 不只测网页。老系统的自定义 TCP 协议、消息队列、数据库，它都能碰。覆盖面广，才有人愿意写插件、写教程。

>

> 短板也要清楚。它是按脚本发请求的工具。浏览器真实渲染、手机设备资源，它管不了。那是客户端工具的地盘。

>

> 课件这份清单看一眼就够。真要动手，去官网查当前版本支持什么。版本一变，清单会变。今天背下来的名字，明年可能多两个、少一个。会用清单的人，不如会查清单的人。

> **提问：**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"——最直接的路子。

- 答"能，用 JSR223 自己写"——更灵活，可以用 Groovy 拼报文、做校验。

- 收口：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：**JDBC 的用途是高频考点，造数据、校验、单独测库。JSR223 是扩展点，简答题写上能体现深度。"JMeter 只能测 Web 应用"是错的。SOAP 和 REST 都归 Web Service 范畴，别当成两个独立的取样器家族。

> **过渡：**界面点着跑不适合正式压测，最强的玩法在命令行。

【7.28】支持 cli（命令行）模式（3 分钟）

图形界面点着跑，跑到两千并发，JMeter 自己先不动了。这时候你测到的不是被测系统，是 JMeter 的极限。

>

> 所以正式压测只有一个正确姿势：命令行。三条理由。GUI 要渲染曲线、要存结果，自己就吃资源。命令行能自动化，写进脚本、进流水线，每天定时跑。命令行还能分布式，配合远程负载机把压力摊开。

>

> 参数挑常用的。`-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，也就是 .jmx 文件的位置和名字。相对路径的根是命令窗口的当前路径，这点最容易踩坑。`-l` 把结果记到文件，.jtl，这是原始数据，必须存。`-d` 指定 JMeter home 目录。`-L` 定日志级别，DEBUG、INFO、ERROR。

>

> 属性有三个，作用域要分清。`-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。一句记法：JMeter 属性给工具用，系统属性给 JVM 用。

>

> 分布式三件套是 `-r`、`-s`、`-X`，下一页展开。`-r` 开启远程负载机，从机列表在 jmeter.properties 里配。`-s` 以服务器方式运行，也就是启动 agent。`-X` 停止远程运行。

>

> 代理四个：`-H` 代理 IP，`-p` 代理端口，`-u` 用户名，`-a` 口令。这里有个歧义，课件里 `-p` 还兼有"读取 JMeter 属性文件"的用法。同一个字母，语境不同含义不同，敲命令时对着文档确认。

>

> 课件给的例子值得照抄：`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。

>

> 两种模式的分工别搞混。命令行跑出来的数据，是留给你事后分析的，不是现场看的。所以先在图形界面里把脚本调通，再用命令行压。顺序反了，两件事都做不好。

> **提问：**脚本和数据都准备好了，命令行一跑报文件不存在，最可能是什么原因？

> **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，你切到别的目录就找不到了。

- 答"权限不够"——也可能，特别是 `-l` 的输出目录不可写。

- 收口：跑之前先 pwd 确认目录，或者干脆全用绝对路径。这是成本最低的保险。

> **易错点 / 考点：**`-n` 是命令行模式总开关，必答。`-t`、`-l` 的作用和相对路径规则要清楚。JMeter 属性和系统属性容易混，一个给工具用，一个给 JVM 用。分布式三件套 `-r`、`-s`、`-X` 的对应关系要记。图形界面只用来调脚本，正式压测走命令行。

> **过渡：**手写脚本太慢，JMeter 有录制功能。

【7.29】JMeter 录制功能（2 分钟）

> 一个真实业务，从登录到下单，几十个请求。手写脚本，一天写不完。录一遍，十分钟。

>

> 课件给的做法三步，每一步都有坑。

>

> 第一步，加逻辑控制单元，录制控制器。它是个占位容器，录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。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。位置放错，你自己都找不着。

>

> 第三步，在浏览器或系统网络设置里配 HTTPS 代理，把流量指向 JMeter 的代理端口。这样你点页面，JMeter 把请求记下来。

>

> 三个坑。坑一，HTTPS 要装证书。代理要解密 HTTPS 流量，就得让浏览器信任它的自签证书。不装，你只能录到 CONNECT 请求，内容全是空的。坑二，过滤静态资源。请求里加 URL 过滤，把图片、CSS、JS 排掉。不过滤，一次录制录进几百个无关请求，脚本没法用。坑三，录完必须整理。

>

> 第三个坑要单独说，因为它区分了"会录"和"会做性能测试"。录下来的是你点的顺序，不是压测需要的顺序。要删无关请求，要把登录抽成 setup，要把会变的参数，时间戳、token、订单号，用关联提取出来。不整理，第二次跑必然失败，因为里面写死了上次的 token。

>

> 录制只是把手工操作变成脚本的捷径，场景设计还得靠人。脚本越短越稳，能合并的合并，能参数化的参数化。出错了也好查。

>

> 录制的适用范围也要清楚。它适合表单提交、页面跳转这类有明确请求序列的业务。纯接口、没有页面可点的系统，录制帮不上忙，只能手写，或者从接口文档生成。

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"——也常见，库存被扣完、用户被锁。

- 收口：录完必须做两件事，参数化把会变的换成变量，关联把上一个响应的值提取给下一个请求。

> **易错点 / 考点：**三个步骤的名称和位置，代理服务器在工作台。HTTPS 证书这个坑必答。参数化和关联是脚本能反复跑的关键，案例题高频。"录制完的脚本可以直接用于压测"是错的，必须整理和参数化。

> **过渡：**一台机器压不动，JMeter 有分布式。

【7.30】JMeter 支持分布式环境（3 分钟）

> 一台笔记本压到两千并发就上不去了，机器还有余量，是网络先满。这种时候，要加的不是被测系统，是施压机。

>

> 课件的结构是这样：一台 Master，跑 GUI 的那台，下面挂若干台 Slave，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。施压的是从机。

>

> 这个结构值三层。第一层，突破单机瓶颈。一台发两千并发，四台就是八千。第二层，模拟真实分布。用户散在不同地域、不同网络，从机放在不同网段，压出来的网络行为才像真的。第三层，隔离施压与观测。主控不参与施压，汇总时才不会被自己的负载拖慢。

>

> 配置有几个关键点。从机要启动成 server 模式，就是上一页那个 `-s`，或者直接跑 jmeter-server 脚本。主控要知道从机在哪，从机列表配在 jmeter.properties 里，课件点过这个文件。主控用 `-r` 启动远程负载机。端口和防火墙要放通，主从要通信，从机还有回传结果的通道。端口不通，表现就是连不上从机。

>

> 有个坑特别容易踩：主从的 JMeter 版本要一致，脚本里用到的插件和依赖也要一致。不一致，从机报找不到元件，或者结果格式对不上。很多人第一次搭分布式就栽在这儿。

>

> 还有个反直觉的点。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。不同从机的时钟不准，结果的时间轴就对不齐。所以从机按需增加，不是越多越好。

>

> 再补一句，从机上别做别的事。一边压测一边下载文件、开着录屏，样本就脏了。分布式最怕的不是机器少，是每台机器都不知道自己为什么慢。

提问：四台从机加起来，按设定是八千并发，报告里只统计到三千，可能是什么问题？

> **预设回答：**

- 答"从机没连上"——先看主控日志，有没有从机启动失败的记录。

- 答"有的从机先到瓶颈了"——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"——也可能，特别是网络中断或者结果文件写失败。

- 收口：排查有个顺序，先看从机有没有真启动，再看每台从机自己的负载，最后看汇总。

> **易错点 / 考点：**主控和从机的职责要分清，主控不分担压力。从机列表配在 jmeter.properties。主从版本、插件、依赖必须一致，这是高频坑。"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 自带功能有限，靠插件补。

【7.31】JMeter Plugins Manager（2 分钟）

> 装插件这件事，动作只有一步，位置只有一个。放对了就成，放错了什么都不发生。

>

> 课件给的做法：把 jmeter-plugins-manager 的 jar 包，放进 JMeter 安装目录下的 lib/ext 目录，重启 JMeter，界面上就出现插件管理器的入口。然后你在里面搜索、安装。

>

> 为什么是 lib/ext？因为这是 JMeter 加载扩展元件的标准目录。启动时它会扫这个目录，发现实现了 JMeter 扩展接口的 jar，就注册成可用元件。理解这一点，以后装任何 JMeter 扩展都是同一个动作：丢进 lib/ext，重启。

>

> 插件补什么？归三类。图表类：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，写报告时有用。场景类：阶梯加压、Ultimate Thread Group，能画渐进加压、波浪加压这类复杂负载模型。回想前面讲的负载模式，手工配很麻烦，插件直接给你图形化配置。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。

>

> 有一条要提醒。插件不是越多越好。装得越多，版本冲突的概率越大。而且主从的插件必须一致，上一页刚讲过。原则是：需要什么装什么，装完把版本记下来。

>

> 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，表现是元件不见了，或者启动报错。这种时候先怀疑插件，别先怀疑脚本。

>

> 装插件的时机也有讲究。先跑通一个最小脚本，再装插件。脚本还没通就装一堆，出了问题你分不清是脚本错还是插件错。插件解决的是效率，不是替你解决设计。

提问：装完插件重启，新元件没出现，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"——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"——关键。放 lib 和放 lib/ext，效果不一样。

- 收口：顺序是目录、日志、版本匹配，三步能解决九成的插件问题。

易错点 / 考点：插件 jar 放 lib/ext 是必考点，装完必须重启才生效。插件和 JMeter 版本的兼容问题是高频故障原因。主从插件一致这条，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，看一个实际效果。

【7.32】插件示例（2 分钟）

同一批原始数据，换一张图，你能得出相反的结论。这不是夸张，是性能报告里天天发生的事。

>

> 为什么图形化插件值得单独占一页？因为性能测试的结论，是从图里读出来的。同样一份原始数据，看的角度不同，看到的东西不同。

>

> 看响应时间随时间变化的曲线，你能看出跑着跑着变慢了。这是泄漏或者资源耗尽的信号。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、P99 却是 8 秒这种被平均值盖住的灾难。看阶梯加压的阶梯图，你能看出每一档加压之后的稳态表现。

>

> 这里有个观念要重点讲：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。平均值算下来 199 毫秒，看着挺好。可那 10 秒的请求，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，还要看最大响应时间。行业里指标通常写成"95% 的请求在 X 秒内完成"，不写"平均响应时间 X 秒"。前者是用户感受的口径，后者是统计上的自我安慰。

>

> 插件示例不是让你记界面长什么样，是让你记住盯哪几种图。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。四张摆在一起，一份压测报告就有了骨架。

>

> 图的数量也要控制。一张图里叠十条曲线，谁也看不出哪条是哪个接口。能进评审的报告，图不超过四张，每张只回答一个问题。

> **提问：**报告写"平均响应 199 毫秒"，这个数字可能盖住什么问题？

> **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，平均值把不同接口混在一起。

- 答"可能慢在某个时段"——补得好，平均值把时间维度也抹平了。

- 收口：三个维度一起看，分位数的尾部、按接口拆、按时间看趋势。

> **易错点 / 考点：**平均值陷阱是最经典的考点，答题必须提分位数。四种必备图要能说出各自用途。"平均响应时间达标就算通过"是错的，要看分位数和错误率。行业常用写法是"95% 的请求在 X 秒内完成"。

> **过渡：**工具和图表都齐了，课件还给了自学入口。

【7.33】JMeter 更多参考（1 分钟）

> 工具这东西，课堂上只能带到门口。剩下的路，得你自己会走。

>

> 课件给了两条入口。一条是 Apache 的 JMeter wiki，一条是官方用户手册 usermanual。地址都在课件上。一分钟讲怎么用这两条路自学。

>

> 两者分工不一样。用户手册是元件字典。每个元件、每个属性、每个参数的含义都在里面。写脚本卡住了，比如这个定时器有几种、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。

>

> 给一个自学顺序。第一步，把手册里 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。第二步，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。第三步，遇到具体问题再去 wiki 和社区搜。

>

> 还有一条更要紧的。工具的自学能力，比工具本身值钱。JMeter 会升级，会被别的工具替代。读官方文档、看示例、动手试，这套方法不会过时。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的。官方文档是全英文的。别怕，性能测试的术语就那几十个：
> thread、sampler、assertion、listener、throughput、latency。把这一节讲过的词跟它们对上，读文档

> 就不费劲了。

>

> 再说一句查文档的姿势。带着具体问题去查，别从头读。手上有一个跑不通的脚本，比读十页概述有用得多。手册的目录本身就是元件的分类，顺着走一遍，你对工具的边界就有数了。

提问：同一个问题，"为什么我的脚本跑出来没有结果"，你查手册还是查 wiki?

> **预设回答：**

- 答"wiki，找别人的经验"——可以，但先查手册更快。多半是监听器没加、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。

- 收口：顺序是手册排配置、wiki 看用法、社区问疑难，别一上来就搜。

易错点 / 考点：这一页考得少，"官方文档在哪"偶尔出现在填空。更值得记的是手册和 wiki 的分工，这是自学习惯题的标准答法。别把它当可跳过的页，它给的是课后自学路径，实验课会用。

> **过渡：**工具会用了，数据也有了，接下来是整章最难的一步。

【7.34】性能分析（2 分钟）

曲线摆在那儿，慢是肯定的。慢在哪，会上没人说得清。性能分析就是解决这件事的。

>

> 给一套四步法，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。三个常常一起来，根因可能完全不同。同时看分位数，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。这就是前面讲过的时间分解。

>

> 第三步，找证据。分位数告诉你哪儿疼，证据告诉你为什么疼。常用证据五种：资源监控曲线，看 CPU、内存、IO、网络；应用日志，看异常、超时、重试；线程栈，看线程都卡在哪个方法上；GC 日志，看停顿多长、多频繁；慢查询日志和执行计划，看 SQL 走没走索引。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，就单独跑一次这条 SQL，看它到底多久。你怀疑是锁竞争，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。这是唯一可靠的验证方式。

>

> 还有个经验：先排除，再深挖。先用分层把大部分层排掉，再对可疑那一层深挖。上来就扎进某一层，很容易挖错地方。

>

> 一句提醒。性能分析要的是证据链，不是灵感。现象、数据、证据、结论、验证，这条链说圆了，结论才站得住。

>

> 四步里最容易偷懒的是第三步。人一旦有了猜测，就去找支持猜测的证据，把不支持的忽略掉。这不叫分析，这叫给自己的结论找台阶。

提问： 压测发现响应时间从 100 毫秒涨到 2 秒，你先看哪三样东西？

预设回答：

- 答"看 CPU 和内存"——基础，但不够，得先分清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，特别适合 CPU 不高但很慢的情况。

- 收口：先看资源曲线定类型，再看最可疑那一层的证据，最后改一个变量验证。

易错点 / 考点： 四步法的名称和顺序可以当简答模板。"响应时间变长就是代码有问题"是错的，可能是数据库、网络、GC。"改一个变量再验证"这条对照实验的思想常考。五种证据来源写全比写深更容易得分。

> **【思政落点 3】**：性能分析最像科研的日常：提假设、找证据、做对照实验、被数据推翻、再来一轮。它教人的第一件事是服从数据。你原以为是数据库慢，线程栈却显示全卡在一个日志锁上，那就得改口。承认"我猜错了"在工程里不丢人，是往前走的唯一入口。请养成一个习惯：说结论之前，先问自己证据是什么。

> **过渡：**找到瓶颈只是第一步，接下来动手改。

【7.35】调优-Tuning（2 分钟）

> 系统慢了，会上第一个提议通常是加机器。加八台，还是不达标。钱花了，问题还在。

>

> 调优要讲顺序，从便宜到贵排。

>

> 第一层，配置参数。数据库连接池大小、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。所以它是调优里性价比最高的一档。

>

> 第三层，代码。去掉循环里查库、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，天花板也最高。

>

> 为什么按这个顺序？因为便宜的方案能解决的，别上贵的。现实里常见的失误是：一遇到慢就加机器，加了八台还不达标，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。

>

> 两条铁律。第一，一次只改一个变量。同时改三处，好了你不知道是哪处的功劳，坏了也不知道撤哪一处。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。

>

> 最后提醒一句。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。无限调优是资源黑洞。工程上要的是达标即止，不是极致性能。

>

> 还有一条常被忘的，调优之前先把监控打开。没有监控，改完只看得到总时间变没变，看不到是哪一段变了。改对了不知道为什么对，改错了也修不回来。

提问：系统慢了，团队第一反应是加两台服务器，你怎么回应？

> **预设回答：**

- 答"先加吧，快"——能缓解，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，顺序很明显。

- 答"要看是计算型还是等待型"——等待型，等数据库、等网络，加机器往往无效。

- 收口：先花半天确认瓶颈在哪一层，能力不足就扩，效率问题就改代码或者改 SQL。

易错点 / 考点：四层调优的顺序和代表手段。"性能问题都可以靠加硬件解决"是错的，架构和算法层面的问题加硬件无效甚至更糟。一次只改一个变量是实验方法的核心，必考。达标即止这个工程观，案例题里常用来讲不要过度优化。

> **过渡：**调优完了不算结束，上线之后还得一直盯着。

【7.36】性能监控（2 分钟）

凌晨两点，用户打电话说系统卡。你才知道。这个时间差，就是监控要拿回来的东西。

>

> 性能测试像体检，一年一次。监控是随身的心率监测，一直看着，及时报警。

>

> 监控要盯三类。第一类，资源：CPU、内存、磁盘 IO、网络带宽，还有连接数、文件句柄、线程数。这类东西有看得见的上限，快到顶就得报警。第二类，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，依赖：数据库慢查询数、缓存命中率、消息队列堆积、第三方接口可用性。很多"我们系统慢"，最后查出来是别人的接口慢。

>

> 业界有个好用的说法，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。四个信号盖住了大多数故障的前兆，可以拿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，说明信任已经损失了。监控报警，说明你还有时间处理。

>

> 两条实践要点。监控要有基线，报警要有阈值。没有基线的曲线，你看不出今天这个 60% 算不算异常。没有阈值的报警，要么天天响，响到没人看，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。

>

> 最后串一句。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。

>

> 还有一层，监控的粒度要能下钻。总数难看的时候，得能按接口、按机房、按版本拆开看。只能看一个总量的面板，报警响了，你还是不知道先改哪。

提问：线上没做任何监控，能靠用户反馈发现性能问题吗？

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，反馈还零散、滞后。

- 答"不能，太被动"——对。而且用户说的"慢"没有量化，排查还得从头来。

- 答"至少要有基础监控"——务实。哪怕只盯 CPU、响应时间、错误率三个指标，也比什么都没有强。

- 收口：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：三类监控对象要说全，依赖这一类最容易被漏。四个黄金信号是延迟、流量、错误、饱和度，写上能加分。"监控就是为了出事时看日志"是错的，核心价值在提前发现。阈值要基于历史分位数，拍脑袋定是常见错误。

过渡：测试、调优、监控串起来，就是一张全生命周期的图。

【7.37】全生命周期的性能测试（3 分钟）

把前面这些活动按时间排开，你会发现有一半不发生在测试阶段。这就是全生命周期这张图的意思。

>

按阶段往下走。性能需求收集与定义，产出性能需求文档，度量是指标能不能测。内建性能，也就是设计中的性能，加上性能建模，产出架构方案和性能模型，度量是设计能不能满足容量目标。单元性能测试和代码优化，产出优化后的代码，度量是关键路径的复杂度和耗时。性能测试，产出测试报告和数据，度量是达没达标。监控与容量管理，产出监控面板和扩容计划，度量是容量水位和告警准确率。

>

几个词要专门解释，它们把"测试"抬成了"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，它得写进研发流程。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。没有流程撑着，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。一个需求什么时候算做完？如果 DoD 里写了性能达标，性能就成了交付的一部分，不是可选项。KPI Dashboard 是把指标做成看板，让所有人随时看得见。性能监控也是这套东西的一部分，它把看板上的数变成每天的日常。

>

环境验证、测试执行，含基线、生产负载、可伸缩性。基线是标准状态下的参照，生产负载是真实流量的模型，可伸缩性是加资源能不能换来能力。

>

> 容量管理是上线后的长期活。预测增长、提前扩容、算成本。输出不是报告，是决策。

>

> 一句总结。性能工程成熟的标志，是它进了流程和看板，不是停在某个人的压测脚本里。

>

> 评审一个需求，先看它在哪个阶段落空。需求里没写指标，后面全在补。设计里没算容量，测试时才发现扛不住。上线后没有看板，问题只能等投诉。哪一段空着补哪一段。

提问：团队只有一个人在压测，别人都不关心性能，这套体系跑得起来吗？

> **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。性能得写进 DoD、评审清单和发布门禁。

- 答"先从一个项目试点"——务实。先证明价值，再推广成规范。

- 收口：关键动作是把个人能力变成流程约束，跟前面讲流程的思路一样。

易错点 / 考点：五个阶段和产出物的对应关系。DoD 要会解释，还要能说出把性能写进 DoD 的意义。"容量管理是运维的事，跟测试无关"是错的，容量数据来自测试。基线在多处反复出现，它是一切比较的前提。

过渡：7.1 到这里收口，用一页小结压实。

【7.38】本节小结（2 分钟）

> 合上讲稿，三分钟能不能把性能测试是什么、分几类、怎么做、怎么看结果讲一遍？讲得下来，这节就过关。卡在哪一处，回去补那一处。

>

> 先对四条。

>

> 第一条，了解性能测试的不同目的，也就是性能测试类型。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，清楚几个基本概念：负载、负载模式、虚拟用户、性能指标。这四个词是这一节的四件套。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。这条最容易被忽略。很多人以为性能测试就是跑工具。课件明确说，它跨了静态测试和动态测试，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，比如 JMeter。关键在"熟练"两个字。会打开界面不算熟练。能独立走完设计场景、写脚本、分布式施压、采集、出报告，才算。

>

> 这四条不是四个孤立的知识点，是一条链。目的决定你选哪种类型，概念决定你怎么描述负载，跨活动决定你找谁配合，工具决定你怎么落地。

>

> 还有个自测的办法。拿一张纸，七个类型、四个概念、四层调优，各写一行。写完能讲出每个词指什么，这一节就真在你手里了。讲不出来的，回去翻对应那页。

> **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，真实用户不可预测"——最准确。这也是为什么要专门设计负载模式和思考

时间。

- 收口：压测的准确性，取决于你的负载模型有多像真实用户，不取决于工具多强。

> **易错点 / 考点：**七种类型是本节最可能出简答和填空的地方，必须默写。虚拟用户的行为由脚本和思考时间决定，这句话能解释很多"压测结果和线上不符"的现象。"性能测试只需要会工具"是错的，课件明确列了架构评审、代码评审、调优。小结页常被当复习提纲。

> **过渡：**理论讲完，两次实验把它落地。先看后端。

【7.39】实验 4-1：基于 JMeter 的后端性能测试（3 分钟）

> 实验报告交上来，最常见的问题是一堆截图，没有一句分析。所以这一节把每条要求为什么这么提，讲清楚。

>

> 前置条件：选一个需要用户登录的 Web 应用。为什么要登录？因为登录带来三个真实问题。会话要保持，cookie 或者 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登录凭证。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。

>

> 要求逐条看。用代理服务器加录制控制器录脚本，练的是脚本从哪来。选用插件、设置线程组参数，练的是负载模型：线程数、Ramp-Up、持续时间。会用两三种控制器，练流程编排：事务、循环、如果。会配两三种采样器，HTTP 必备，再加 JDBC 或者 FTP。会用 HTTP 授权和 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可不可信。会用 JDBC 连接预处理，练造数据和校验。会用两三种监听器，查看结果树、响应时间、吞吐量，练结果呈现。

>

> 交付两样：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；还有 .jmx 脚本文件。

>

> 报告那五个部分，就是这一节的主线。测试系统是测什么。环境是那六项要素。场景设计是负载三要素加模式加类型。执行是怎么跑、跑多久。结果分析是四步法加证据。按这条线写，报告自然有骨架，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。哪个监听器、哪张图、哪次执行。

>

> 还有一项最容易漏，环境。同一份脚本，换一台机器跑，数据就不一样。所以报告里写清机器配置、JDK 版本、JMeter 版本、被测系统部署了几台。这些不写，别人复现不了你的结果。

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。

- 收口：断言不是加分项，是数据可信的前提。没有断言的压测报告，结论不能采信。

易错点 / 考点：实验要求要能逐条对应到知识点，录制对应脚本来源，线程组对应负载模型，断言对应可信度。报告五部分是评分重点，缺一项扣分。常见失分是只贴截图不写分析、没有场景设计说明、数据缺版本和环境信息。交 .jmx 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。

【思政落点 4】：实验报告里每个数字都要有出处，这句话不只是技术要求。它说的是：你写下的结论，得有人能顺着它找到证据。工程里最贵的成本，往往不是改一个缺陷，是信了一个来路不明的数字。请把"数据可追溯"当成职业习惯：测了什么、在什么条件下、谁跑的、原始文件在哪。这套习惯换个行业照样成立，它是诚实的一种具体形状。

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）

同一台手机，同一个 App，两次测出来的启动耗时差半秒。哪个是真的？先别急着取平均。

>

> 实验 4-2 用 PerfDog 做客户端性能测试。课件给了八步，按步骤说要点。

>

> 第一步，去 PerfDog 官网下载正确的桌面应用，官网是 perfdog.wetest.net。"正确"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 连手机，让它自动检测、把手机加进应用列表。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、不支持传数据。工具再好，连不上就是白搭。

>

> 第三步，定测试模式，USB 或者 WIFI。USB 更稳、延迟更低，适合精细采集。WIFI 方便，但网络抖动会污染数据。做实验先用 USB。

>

> 第四步，选被测试的 App 开始测。这里要定场景：冷启动、页面切换、列表滑动、长时间使用。每个场景采到的数据含义不同。

>

> 第五步，测试过程中采集数据。第六步，保存结果。第七步，统计与分析。这一步是重点。工具给你曲线，你要说出哪一段异常、为什么。第八步，写报告。

>

> 报告里固定放四样：测试机型和系统版本，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、可能是什么原因。

>

> 一条实验纪律：同一个场景至少测两遍，一遍基线一遍对照。只测一遍，你拿到的是观测，不是结论。

>

> 还有个细节，采集周期。采样太稀，抓不到卡顿那一下。采样太密，工具自己就占资源。默认值先按工具给的来，看到异常再调。

> **提问：**同一台手机、同一个 App，两次启动耗时差 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，至少三五次，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。

- 收口：客户端测试第一条纪律是把条件钉死，冷热启动、网络、电量、后台进程、系统版本，全写进报告。

> **易错点 / 考点：**八个步骤的顺序。冷启动和热启动的区分是客户端性能的核心概念，常考。"客户端性能测试用模拟器就行"是错的，真机数据才有意义。报告要写清机型、系统版本、场景和条件，否则数据无法复现。

> **过渡：**性能测试到这里收口。7.2 换一类专项，安全性测试。

【7.41】7.2 安全性测试（2 分钟）

> 性能那一讲，我们问的是系统扛不扛得住。到安全这里，问题换成了一句更难受的：有人存心要进来，它挡不挡得住。

>

> 功能测试有个默认前提，使用者按说明书操作。安全测试的前提反过来，对手不看说明书，专门找说明书没写的那几条路。

>

> 于是功能用例里永远不会出现"用户名里填一个单引号"这一条。用例全绿，门可能还开着。

>

> 两个专项的判据也不同。性能看压力下的响应，安全看能不能被利用。同一段代码，功能上输出正确，安全上可能已经把错误堆栈、表名、框架版本一起交出去了。

>

> 很多人心里有个想法，安全是安全工程师的活。这个想法撑不住。漏洞大半是在开发阶段写进去的，少一个参数校验，多一句 SQL 拼接。写的人不设防，事后补不完。

>

> 课件把内容分成四块。威胁模型与安全漏洞模式、范围与方法、贯穿研发生命周期、Web 安全与测试工具。

> 四块的顺序是建模、定范围、铺流程、落场景。先想清楚敌人会从哪来，再决定拿什么工具去挡。反过来的做法很常见，上来就开扫描器，扫完一屏报告，不知道先修哪个。

>

> 再补一层差异，报障的方式也不一样。功能缺陷会有人报，用户一用就骂出来。安全缺陷没人报，在被人利用之前，系统一切正常，监控也不响。所以它只能靠人主动去找，等不到有人投诉。

>

> 产出形式也不同。功能测试交的是通过还是不通过。安全测试交的是一份带评级的问题清单，外加还没排上的风险。

>

> 一句可以带走的：安全测试的核心动作，是主动构造那些不该出现的输入和环境。

> **提问：**功能测试全部通过的系统，安全性上可能千疮百孔吗？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：功能看预期结果，安全看能不能被利用。

- 收口：所以安全测试的核心动作，是主动去造那些不该出现的输入和环境。

> **易错点 / 考点：**性能与安全的判据差异要能一句话说清：一个看压力下的表现，一个看能不能被利用。本章安全部分的结构要能复述成四块。判断题"安全性测试是安全专家的事"是错的，安全要贯穿开发全过程。本节的问题意识是攻击者视角，后面十几页都靠它。

> **过渡：**讲漏洞之前，先把人分清——黑帽和白帽。

【7.42】知道它们的区分吗？（2 分钟）

> 同一个人，同一套工具，同一个漏洞。为什么在这家公司做叫渗透测试，换个地方做就叫犯罪。

>

> 分界不在技术，在授权。

>

> 黑帽是未授权进入。他找漏洞是为了牟利、破坏、控制，手段和白帽几乎一样。差的是没人允许他动这个系统，后果由别人承担。

>

> 白帽是在授权范围内动手。用攻击者的办法找问题，目的是在坏人之前发现并修掉。安全测试、渗透测试、红队演练都在这一侧。

>

> 这个区分对测试人员不是道德课，是职业规范。开工之前四样东西要有：书面授权、测试范围、时间窗口、应急联系人。少一样，事后你说不清自己做过什么，还可能把生产系统打挂。

>

> 中间还有一层，叫灰帽。发现了漏洞，没拿到授权就去验证，验证完还公开。争议多半出在这儿。善意不能替代授权。

>

> 给三个画面，把这条线画清楚。

>

> 自己搭环境，扫自己的练习系统，随便扫，这是白帽。

>

> 在授权范围里，扫到登录口就停手，授权书没让你试口令。这也是白帽，边界守住了。

>

> 同样一套工具，对着别人的站扫，哪怕你只想看看有没有漏洞。这已经是黑帽，报不报告都不改变性质。

>

> 灰帽麻烦的地方在这儿：漏洞是真的，动机也可能不坏，可验证那一步越过了授权。事后说不清你有没有碰过数据。

>

> 落点是一句话：同一记 SQL 注入，在授权范围内是测试，出了范围性质就变了。以后做安全测试，第一步把授权、范围、窗口落实成文字，第二步才敲命令。

提问：安全工程师在客户系统里发现一个高危漏洞，顺手验证了一下能不能拿到数据。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这一项。

- 答"要看有没有授权"——对。发现漏洞和验证可利用性，在很多授权书里是两件事。

- 答"应该先报告再验证"——很专业。顺序是报告、客户确认、在约定窗口内验证。

- 收口：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：黑帽、白帽、灰帽的区分标准是授权，不是技术高低。安全测试的四个前置文件是实践题得分点：授权、范围、窗口、联系人。判断题"只要目的好，验证漏洞不算越界"是错的。概念题答案要落在"授权"两个字上。

【思政落点 5】：授权这件事，落到的是规矩意识。同一套技术，越界一步性质就变了，"我没恶意"从来不是免责的理由。同学们以后进金融、医疗这类行业，签字那一刻就是承诺。先把范围、窗口、联系人写清楚再动手，这份自觉比技术更难养成，也更能说明一个人专不专业。

过渡：人分清了，看这套内容整体怎么排。

【7.43】第 13 讲：安全性测试（1 分钟）

五块内容摆在这儿，如果只能记一块，你记哪一块。

>

> 先按顺序点一遍。第一块，威胁模型与安全漏洞模式：漏洞是什么、从哪来、攻击分几段、有哪些经典类型。

>

> 第二块，安全性测试的范围与方法。里面是安全目标、功能测试和漏洞测试的区别、静态与动态、模糊与渗透。

>

> 第三块，贯穿研发生命周期的安全性测试。需求、设计、编码、测试、运维各阶段分别做什么，重点是架构风险分析和代码评审。

>

> 第四块，Web 安全性测试：注入、XSS、CSRF、URL 访问控制这些具体攻击怎么测。

>

> 第五块，安全性测试工具：扫描器、代理工具、漏洞库这类支撑手段。

>

> 五块连成一条线：建模、定范围、铺流程、落场景、上工具。

>

> 最容易被跳过的是前三块。工具最顺手，装上就能跑，所以团队从工具开始，扫完一堆报告，不知道先修哪个。顺序反了，工具只会放大慌乱。

>

> 第四块单独拿出来，有它的道理。Web 最容易被外面直接够到，入口多、变得快。别的系统只有一个登录口，一个网站可能有几十个接口。

>

> 五块之间也互相牵着。没有第一块的建模，第二块的范围定不下来，只能按清单平均用力。没有第三块的流程，安全活动全挤在验收前那两周，前面几个月没人管。

>

> 五块也可以当成五道工序看。前面不做，后面做的都是补救。

>

> 答题时先把五块名字写全，再补一句主线，结构分就稳了。

>

> 主线句是这句：安全测试不是跑个扫描器，是拿威胁模型去设计攻击，再回来验证防线有没有用。

> **提问：**五个部分里，你觉得团队最常跳过哪一块？

> **预设回答：**

- 答"工具，直接跑扫描器最快"——正好相反。工具最容易被用起来，前四块最容易被跳过。

- 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"贯穿研发生命周期"——也对，很多团队只在验收前扫一遍。

- 收口：这一节的顺序本身就在纠偏，先想威胁，再谈工具。

> **易错点 / 考点：**五块名称要能默写，结构题常问"安全性测试包括哪些内容"。最容易错的是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。

> **过渡：**从第一块开始，先说威胁模型到底是个什么东西。

【7.44】威胁模型与安全漏洞模式（1 分钟）

> "威胁模型"四个字，很多人念完还是不知道它是什么东西。

>

> 拆开就是一句话：谁，从哪，用什么办法，来伤这个系统，伤了会怎样。

>

> 它由四块构成。资产，要护住什么，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。

>

> 建它不是为了交一份文档。安全测试的资源永远不够，不可能把所有攻击都试一遍。威胁模型干的是排序，把值钱的资产加门槛低的入口拎到前面先测。

>

> 它还有个副产品，会把设计里没写下来的假设翻出来。比如"我们假设内网可信"。这条一旦不成立，终端中毒、VPN 被撞库，整条防线跟着塌。

>

> 具体怎么建，三步，一节课能出一版。

>

> 第一步画数据流。数据从哪进、经过哪些处理、最后存到哪，一条一条画出来。

>

> 第二步标信任边界。跨过这条线，可信程度就变了，要重新检查。浏览器到服务端是一条，服务端到第三方接口又是一条。

>

> 第三步把假设写成句子。写完念一遍，看它有没有失效条件。写不出失效条件的假设，多半是没想清。

>

> 排序也靠它。高价值资产加低门槛入口，先测。值钱但门口挂着五道锁的，可以往后放。

>

> 漏洞十有八九，就是某条没写下来的假设破了。所以建模的时候，除了写有哪些威胁，还要写每条假设在什么条件下会失效。

> **提问：**一个只在内网使用的系统，威胁模型里的"对手"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，这是内网系统主要的威胁源。

- 答"还要算供应链和第三方接入"——补得好，VPN、外包、运维通道都是入口。

- 收口：所以"内网可信"这条假设要写进模型，还要写清它的失效条件。

> **易错点 / 考点：**四要素要能说全：资产、对手、攻击面、缓解措施。判断题"内网系统不需要安全测试"是错的，威胁主要来自内部和横向移动。加分点是"假设失效是漏洞的常见成因"。威胁模型的作用是排序，不是写文档交差。

> **过渡：**四块里最核心的一格是资产会被怎么伤，这就要先把"漏洞"这个词说准。

【7.45】安全性缺陷 — 漏洞（3 分钟）

> 一扇没锁的窗，算不算已经被盗了？不算。可它离被盗只差一个路过的人。

>

> 课件的定义就是这个意思。脆弱性，又称弱点或漏洞。它是 IT 系统中存在的、可能被威胁利用造成损害的薄弱环节。它一旦被威胁成功利用，就可能违犯安全策略，对 IT 系统也就是资产造成损害。

>

> 这句话拆成四块，考试就考这四块。

>

> 第一，它是薄弱环节。漏洞本身不是损害，是一个机会。修漏洞是在关门，不是在抓贼。

>

> 第二，它要被威胁利用。没人用，损害不发生。同一个漏洞放在隔离内网和放在公网，风险差一个量级。风险等于漏洞乘威胁。

>

> 第三，后果是违犯安全策略。策略是我们希望系统守的规矩，谁能看什么、谁能改什么。被利用，就是系统做了规矩不许做的事。

>

> 第四，受损的是资产：数据、资金、服务能力、声誉。

>

> 再补两个行话。公开漏洞用 CVE 编号标识，严重程度用 CVSS 打分。评分细节不用背，只要知道漏洞要按严重度排队修，看见一个修一个等于没排。

>

> 再往细一层看，风险不只两个乘数。漏洞、威胁之外还有影响。同一条注入，打在测试库上和打在生产库上，影响差一个量级。评级不能只看漏洞本身。

>

> 定义里那个"可能"两个字别漏。可能是说它有条件，条件就是威胁。没有威胁来用它，它只是一个弱点。

>

> 这一句请划下来：漏洞是系统的属性，不是某次攻击的事件。它在被利用之前一直在那儿，这正是测试能提前发现它的原因。

> **提问：**一个漏洞存在，但从来没被利用过，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，或者还没人试。

- 答"算，要修"——对。修的依据是风险，不是有没有已经出事。

- 答"要看暴露面"——很专业。内网还是公网、有没有认证、数据多敏感，都影响优先级。

- 收口：所以安全测试的产出是漏洞清单加风险评级，不是事故报告。

> **易错点 / 考点：**定义的四个要素是简答重点，尤其是薄弱环节和被威胁利用这两条。判断题"漏洞就是 bug"不成立：bug 是不符合预期，漏洞是可被利用来违反安全策略，两者有交集但不相等。CVE 与 CVSS 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。

【7.46】安全性：漏洞来源（3 分钟）

> 写代码的人不会故意留后门。可绝大多数漏洞，确实是写代码的人带进来的。

>

> 课件把来源分成两支：故意和无意。这两支的修法完全不同，所以这个分法比记名词重要。

>

> 故意这一支再分两种。恶意的：木马病毒、陷门也叫后门、逻辑炸弹和定时炸弹。都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，逻辑炸弹是满足条件就发作。这类东西不在需求里，也不在编码规范里，常规测试基本抓不到。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。设计者没坏心，可系统某个机制能被拿来偷偷传信息，比如靠响应时间的差异传

> 比特。

>

> 无意这一支，才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、认证与鉴别错误、边界条件错误、其它可利用的逻辑错误。

>

> 每类给个画面。校验错误是该查的没查，输入直接拼进 SQL。范围错误是允许了不该允许的范围，数组越界、权限给太大。串行错误是顺序错了，先扣款后查库存，中间被打断就出事。认证错误是把不该认的人认成自己人，弱口令、会话不过期、token 可预测。边界条件错误是临界值没管，空串、0、最大值、超长输入。其它逻辑错误是流程能被绕，价格改成负数、跳过支付直接回调。

>

> 六类几乎都落在两处：输入校验和状态检查。写代码时反复问两句，这个输入我信得过吗，这个状态我查了吗。

> **提问：**六类无意漏洞里，串行错误和逻辑错误差在哪儿？各举一例。

> **预设回答：**

- 答"都是逻辑问题，差不多"——不一样。串行错误看执行顺序和中间状态，逻辑错误看规则本身能不能被绕。

- 答"串行是先发货后扣款，逻辑是价格能改成负数"——很到位，两个例子都准。

- 答"串行错误就是并发问题"——有关系但不相等。并发会放大它，它本身是步骤顺序和事务边界的问题。

- 收口：判断口诀是看顺序和中间态的是串行，看规则能不能被绕的是逻辑。

> **易错点 / 考点：**两大类和六小类要能对上，填空题高频。陷门与逻辑炸弹的定义要能分开。判断题"漏洞都是黑客植入的"是错的，绝大多数是开发无意引入。六类可以压成六个字记：校验、范围、顺序、身份、边界、业务。

> **【思政落点 6】**：无意造成的漏洞占绝大多数，这条事实指向的是一种职业习惯。少一次校验、漏一个边界值，单看都不算偷懒，凑在一起就是被人打开的门。工程上的负责，常常不是干出多漂亮的设计，而是把枯燥的那几行检查做全。认真在这里不是态度问题，是能不能交付的问题。

> **过渡：**门没锁说清了，接下来看贼怎么进来。

【7.47】危险、攻击手段（3 分钟）

> 入侵一个系统，第一步常常什么都没干，只是看。

>

> 课件把过程拆成三段：预攻击、攻击、后攻击。拆它有实际用处，每一段都能设防，也都能留检测点。

>

> 预攻击就是踩点。常规信息获取、搜索引擎、社会工程学，都在这儿。还有漏洞扫描、信息收集，ping、telnet、whois、traceroute 这一类。端口扫描也是这一步，snmp、superscan 这类工具。这一段全是探测，业务上没有异常。日志里只有一堆 404 和端口试探，所以最难被发现。

>

> 攻击段才动手。DoS、远程渗透、口令猜测，还有远程溢出，像 getadmin 这类提权工具。应用攻击也算这一段，比如 SQL 注入。

>

> 后攻击段是进去之后干什么：权限提升，本地溢出、口令嗅探；留后门；擦痕迹。这一段说明一件冷的事，被入侵和被发现在往隔好几个月。

>

> 测试人员背这个三段式有两个用处。一个是设计检测点。预攻击段看日志和 WAF，攻击段卡输入校验和权限，后攻击段盯文件完整性和账号行为。另一个是渗透测试照着走，先收集，再利用，最后评估影响。

>

> 三段的投入常常不成比例。预算大多砸在攻击段，买设备、上 WAF、加参数校验。预攻击段没人管，后攻击段没人看。

>

> 可事故复盘里，往往追得到预攻击段的痕迹。测试环境的地址写在招聘帖里，源码备份挂在公网目录，内部 Wiki 里留着默认口令。这些不是攻进来的，是摆在那儿的。

>

> 常见的偏差是防线只做中间一段。WAF 买了，参数校验写了，两侧的口子还开着，中间的拦截也撑不了多久。防线得铺满三段。

提问：预攻击阶段为什么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，告警不响。

- 答"探测看起来像正常访问"——很准。扫一个不存在的路径，日志里就是一次 404。

- 答"防守方只盯着攻击"——点到了：资源都投在拦，没投在看。

- 收口：所以这一段的防守靠情报和异常检测，大量 404、异常 UA、非常规时间的访问，都是信号。

易错点 / 考点：三段名称与各自手段要能对应，选择题高频。社会工程学属于预攻击段的信息获取，常被错放到攻击段。判断题"渗透测试就是直接打"是错的，前期信息收集常占一半以上时间。案例题可以加一句：防御要覆盖三段。

过渡：攻击成功之后会留下哪些类型的问题，课件列了一整页。

【7.48】攻击带来的安全性问题（3 分钟）

一条加密的通信，和一条谁都不看内容的通信，哪条更危险？答案有点反直觉，后一条有时候更危险。

>

> 课件列了攻击造成的后果，一共九条。我不逐字念，归成四组，按组记更牢。

>

> 信息层面三条：信息泄露、破坏信息的完整性、业务数据流分析。泄露是不该看的看到了。完整性破坏是不该改的改了。数据流分析不碰内容，只看通信模式，谁在什么时候和谁通信、数据量多大。商业情报就出来了。加密护的是内容，护不住模式。

>

> 可用性层面一条：拒绝服务，DoS 和 DDoS。成本最低，效果最直接。

>

> 访问控制与身份层面六条。非法使用也就是非授权访问、窃听、假冒、旁路控制。还有授权侵犯即内部攻击、抵赖即来自用户的攻击。这一组杂，抓两条线就够：身份不可信，行为不可追溯。内部攻击单列出来，是因为内部人本来有权限，他做的事在系统眼里完全合法。

>

> 环境与合规层面两条：计算机病毒和恶意软件，还有信息安全法律法规不完善。最后这条听着不像技术问题，可它最现实。法规不完善意味着取证难、追责难、赔偿难。攻击的收益就高，成本就低。

>

> 四组的顺序也值得看一眼。信息、可用、身份、合规，越往后越不像纯技术，也越容易在复习时被跳过。

>

> 还有一点，这九条里有一半不是技术能单独解决的。法规、取证、追责，属于制度层面。做题时别顺手把它们划掉。

>

> 九条合起来都能映射回五个目标：保密、完整、可用、真实、不可抵赖。下一页那六个字母，就是把这九条按手法又切了一遍。

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：某公司半夜向某台服务器传了 10GB，不用看内容也知道出了事。

- 答"它泄露的是行为模式"——准确。这类攻击难防，因为它依赖的是元数据。

- 答"可以用流量填充对抗"——很专业，这是常见手段之一。

- 收口：加密保护内容，不保护模式，两件事要分开设计。

易错点 / 考点：九条要能说全，填空和多选都高频，归类法能省力气。最容易漏的是授权侵犯即内部攻击，还有抵赖。业务数据流分析是理解性考点，要能解释元数据也是信息。九条和五个目标的映射，下一页还要用。

过渡：把这九条按攻击手法重新切，就是安全领域最常用的那个模型。

【7.49】软件系统安全威胁：STRIED（3 分钟）

六个威胁、六道防线，这个模型在业界用了二十多年，为什么一直没被换掉。

>

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。答题按课件写，把每个字母的含义说清就行。

>

S，假冒 ID：冒充别人。冒充用户登录、冒充服务端、伪造 token。挡它靠认证，多因素、证书。

>

T，篡改：改了不该改的东西。改请求参数、改数据库、改传输中的数据。挡它靠完整性，签名、哈希、HTTPS。

>

R，抵赖：干了不承认。我没下过这个单，我没批过这笔款。挡它靠不可否认性，审计日志、数字签名。

>

I，信息泄露：不该看到的看到了。报错带堆栈、接口多返回字段、日志里存明文口令。挡它靠保密，加密、脱敏、少返回。

>

D，拒绝服务：让你用不了。挡它靠可用性设计，限流、降级、容量冗余。

>

E，特权升级：本来只能看自己的，结果能看所有人的。挡它靠授权，逐次校验、最小权限。

>

它好用的地方在于，威胁和防线一一对应。左边六个威胁，右边六种机制，逐项问"我们有吗"，这就成了一份能照着做的检查单。

>

> 有人会问，六类会不会重叠。会。一次越权往往既是假冒又是特权升级。答题看它靠哪道机制被挡住，就归到哪一类。

>

> 检查单怎么用？画一张表，左边六个威胁，右边六道防线。每个格子问三句：有没有这道机制，覆盖到哪些接口，上次验证是什么时候。

>

> 答不上来的格子，就是这次要测的地方。这比凭感觉挑几个接口点一遍靠得住。

>

> 有人说我们做了加密。那就接着问：认证呢，授权呢，审计呢。加密只盖住 I 那一格。

> **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 那一格，其余五类它管不着。

- 答"不能，还要认证授权"——对。假冒、抵赖、特权升级都得靠别的机制。

- 答"还要防 DoS"——补得好，可用性是另一条线。

- 收口：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：**六个字母和六类威胁的对应必须默写，是本节最高频考点。拼写差异别纠结，STRIED 和 STRIDE 写清含义即可。错答最多的是"加密就等于安全"，加密只对应保密性。口诀可以自己造一句，假、改、赖、泄、拒、升，按顺序默出来。

> **过渡：**分类讲完，看具体漏洞长什么样，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）

> 八个漏洞模式，名字你大概都听过。我更想你看到它们共同的毛病。

>

> 缓冲区溢出，往固定大小的缓冲区里写超长数据。多出来的盖住相邻内存，轻则崩，重则执行攻击者的代码。

> 这个下一页单独看。

>

> 图像格式漏洞，WMF、TIFF、PNG：漏洞不在你的代码里，在你解析图片的那个库里。用户传一张特制图片，解析时溢出。你的攻击面包括你依赖的所有第三方组件。

>

> 竞争条件，race condition：检查的时候没问题，用的时候已经被改了。提现先查余额再扣减，中间被并发插一刀，就能超提。

>

> SQL 注入：把用户输入当成了语句的一部分。Web 时代最经典的漏洞。

>

> XSS，跨站脚本攻击：把脚本种进给别人看的页面，别人一浏览就执行。

>

> CSRF，跨站请求伪造：借用户已经登录的身份，让他在不知情的时候发出请求。

>

> Frame-proxy：用框架把真页面套住，用户在"看起来是你家"的页面上输口令。

>

> 并发漏洞：重复提交、库存超卖、幂等缺失。功能测试几乎看不出来，只在并发的时候冒头。

>

> 八类里只有前两类跟语言和第三方库有关。后面六类基本出在业务和框架层。换语言解决不了大部分问题。

>

> 边界思维比背八类管用。换个系统，八类可能只中三类，边界永远是那四种。

>

> 依赖边界最容易被忘。列一份组件清单，标出版本，查有没有已知漏洞。这件事在项目里几乎没人主动做。

>

> 共同点在这儿：这些都不是功能写错了，是边界没守住。输入边界、身份边界、时序边界、依赖边界。凡是跨边界的地方，都得查一遍。

> **提问：**XSS 和 CSRF 都带"跨站"，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。一个攻页面的可信性，一个攻请求的可信性。

- 答"XSS 要用户浏览，CSRF 要用户已登录"——补得好。都依赖用户，依赖的东西不同。

- 收口：XSS 是让你的浏览器执行我的代码，CSRF 是让你的浏览器用你的身份发我的请求。

> **易错点 / 考点：**八类要能认，选择题常从里面挑。XSS 与 CSRF 是最高频的混淆点，必须一句话分清。竞争条件要能举先查后改的例子，并说出 TOCTOU。加分做法是把八类归到四种边界上，说明你不只是背名词。

> **过渡：**八类里的第一类最老也最凶，单独看它。

【7.51】常见的 C/ C++缓冲区溢出问题（2 分钟）

> 一个八字节的数组，往里写二十个字节，会发生什么？

>

> C 的数组不查边界。声明 8 字节，拷进去 20 字节，编译器不拦，多出来的 12 字节盖到相邻内存上。相邻内存里常常存着函数的返回地址，也就是这个函数跑完该回哪儿继续。

>

> 攻击链就这么成立。构造超长输入，盖掉返回地址，让它指向攻击者自己的代码，或者系统里某个现成函数。程序"返回"的时候就跳过去了。这叫栈溢出。它狠的地方在于不要你先登录，一个输入框就够。

>

> 常见错点四个。一是不查长度的函数，strcpy、sprintf、gets，只负责拷不负责数。二是长度算错，忘了结尾那个 \0，就多写一个字节。三是整数溢出把校验绕过去，先算还差多少空间，算成负数，校验就失效。

> 四是把用户输入当格式串。写 printf(user_input)，而不是 printf("%s", user_input)。

>

> 怎么防？三条。换成 strncpy、snprintf 这类安全版本。长度校验做在所有入口，不只做在页面上。再开编译器和运行时保护，栈保护、地址随机化、不可执行栈。三条要一起上，单靠一条都能被绕过去。

>

> 写代码时可以养一个自查习惯。凡是从外部进来一个长度、一个数量，先问上限是多少。写不出上限的地方，通常就是溢出可能出现的地方。

>

> 还有一点时间上的差别。溢出这类漏洞一旦成功，攻击者拿到的往往是整台机器的控制权，不是一条数据。所以它的优先级一直排在前面，哪怕听起来很老。

>

> 最后一句：Java 这类语言帮你挡住了数组越界，可边界检查不等于业务校验。金额能不能为负，还得你自己管。

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"——语言层确实挡住了数组越界，可同类问题还在。拼接巨量字符串、超大请求体打进内存，本质都是没限制输入规模。

- 答"不是，还要防依赖库"——很专业。JVM、图像解析库、JNI 调用的本地方法，都可能带进原生层漏洞。

- 答"还要做业务层校验"——对。语言给的是内存安全，业务得自己管业务安全。

- 收口：换了语言，风险的形态变了，风险本身没消失。

易错点 / 考点：溢出的原理链条要讲清，覆盖返回地址是关键一环。不安全函数要能列举。判断题"用高级语言就没有溢出问题"是错的，要看是否涉及原生库和输入规模。三条防线要能说全，并且强调组合起来才有效。

过渡：单类漏洞看完，再看行业给出的那份最该防的清单。

【7.52】OWASP Top10 漏洞模式（2 分钟）

> 有一份清单，全世界做 Web 安全的都在用。可照着它抄进用例库，你一条都测不出来。

>

> 先说是什么。OWASP 的全称是 Open Web Application Security Project。中文叫开放 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，owasp.org/Top10。

>

> 它为什么有分量？因为它按真实数据排。收集大量应用的实际漏洞统计，按出现频率乘影响程度排序，几年更新一版。所以它写的是现实里真被打穿的地方，不是理论上的危险。它还是同一种语言，你说这是 A01 类问题，同事就知道你在说什么。

>

> 它不能干什么，这条最关键。Top 10 是类别，不是测试用例集。注入是一类，具体到你系统里是哪个接口、哪个参数、怎么注，清单不告诉你。它是地图上的高危区域，不是具体坐标。

>

> 怎么用？三步。拿类别当检查表，逐项问我们有没有这一类。把每一类翻成本系统的测试点，哪个入口、什么输入、预期哪道防线。再按业务影响排序，先测打穿后果最重的地方。

>

> 它也公开可查，学生能拿到全文，不必等公司给账号。这一点对自学很关键。

>

> 再补一层用法。把 Top 10 和 CVE 编号对着看。清单告诉你该防哪一类，CVE 告诉你这一类里哪个组件刚出了事。

>

> 条目编号别死背，清单会更新。长期在榜的那几类要认得。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。

>

> 考题常问三件事：它是什么、凭什么有分量、能不能直接当用例用。背编号没有分。

> **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，要落到接口和参数"——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，优先级差很远。

- 收口：正确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：**OWASP 的全称与性质要能答。错答最多的是"Top 10 就是十个漏洞"，它是十类风险。它有两个特征能解释权威性：来自真实数据，几年更新一版。具体编号不必背，长期在榜的类别要认得。

> **过渡：**威胁和漏洞都看完了，接下来是最实际的问题：这些怎么测。

【7.53】安全性测试的范围与方法（1 分钟）

> 客户说"帮我们把系统安全测一遍"。这句话里没有一个字能直接用。

>

> 得先把它拆成三问：测什么，测到什么程度，怎么测。

>

> 测什么，先立目标。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。

>

> 测到什么程度，要定安全等级。内部公告系统和人脸识别支付，不可能用一套标准。这就是后面五个安全保护等级的用处，它决定投多少人力、验收按什么口径。

>

> 怎么测，两条轴。静态还是动态。静态是工具扫描、人工评审、专家评估，程序不跑。动态是功能测试、模糊测试、渗透测试，程序要跑。视角还分两种：基于威胁，从外部问会发生什么；基于漏洞，从内部问已经存在什么。

>

> 顺序别倒：先定目标，再定等级，最后选方法。反过来先挑工具，做完的报告说服不了任何人。

>

> 两套分法要能对上。静态发现的是代码里写着的毛病，动态发现的是跑起来才暴露的毛病。同一个越权，静态能看出没做归属校验，动态才能证明真能拿到别人的数据。

>

> 基于威胁的用例从场景出发，写出来像攻击剧本。基于漏洞的用例从清单出发，写出来像检查表。两种都得有。

>

> 还有一问常被跳过：这份报告交给谁。交给开发的，要能指到行；交给管理层的，要能说清风险和成本。同一个结论，两种写法。

>

> 还有一件事要提前问，这次的授权和窗口。没有授权不能开工，这条在讲黑帽白帽的时候已经说过一遍。

> **提问：**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，既报不了价也交不了付。

- 答"要多严"——第二个。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。方法决定工作量和工具。

- 收口：再加一个隐藏问题，这次的授权和窗口。没授权不能开工。

> **易错点 / 考点：**安全目标的核心是 CIA 三元组，全称加解释都要会。静态与动态的划分要清楚，代码评审属于静态。最容易混的是基于威胁和基于漏洞，一个是外部视角，一个是内部视角。"先目标、再等级、后方法"这个顺序，案例题按它组织答案很顺。

> **过渡：**从目标说起，看看安全目标到底包含什么。

【7.54】安全目标（3 分钟）

> 三条目标，经常互相打架。这不是设计没做好，它本来就这样。

>

> CIA 是保密性、完整性、可用性。保密性，信息只让该看的人看到。完整性，信息不被非法篡改，处理方法是准确、原始的。可用性，该用的时候能用。

>

> 课件有一句话要抄进笔记：这些目标常常互相矛盾，需要在这些目标中达到最佳平衡。它举的例子很干脆，简单地阻止所有人访问一个资源，保密性满分，可用性归零。

>

> 三个真实冲突。为了保密，把日志里的敏感字段全删，出了事没法追溯，可审计性受损。为了可用，做多副本多缓存，副本越多，保密性和一致性越难保证。为了安全，登录加五道验证。用户烦了，开始用弱密码，开始把验证码贴在显示器上。整体安全反而更差。

>

> 所以平衡不是客套话。安全不是一个能无限加码的维度，它带着成本。每加一道防护，都得问两句：挡住的威胁值多少，付出的可用性和体验代价多少。

>

> 还有一层，任何一条目标做到满分，另外两条一定吃亏。这不是技术不行，是设计必须做的选择。所以安全方案都带一份取舍说明，挡了什么、让掉了什么。

>

> 再补一个冲突，留存与保密。为了出事能查，日志要留久、留全。可日志里全是手机号、地址、重置记录，留得越久，泄露那天的损失越大。合规要求留存，隐私要求少留，两边都得交代。

>

> 行业里还常提三条：真实性、不可抵赖性、可追溯性。这三条下一页会正式列出来，现在先有个位置感。

> **提问：**"为了绝对安全，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，最后会共用验证码、会用弱口令，实际安全度下降。

- 答"影响可用性"——对，这正是 CIA 之间的冲突。

- 答"要按风险分级加验证"——专业做法。转账、改密码强验证，看公告弱验证。

- 收口：所以正确问法是这个操作风险多高、配多强的验证，不是越多越好。

> **易错点 / 考点：**CIA 三个词的中英文和含义必须默写。三者会冲突这个观点是简答和论述的高频点，一定要举例。错答最多的是"安全越高越好"，忽略了成本和可用性。课件那个"阻止所有人访问"的极端例子可以直接引用。

> **过渡：**CIA 是三条主线，摊成要求是八条。

【7.55】安全性要求（3 分钟）

> CIA 是三条，安全要求摊开是八条。多出来的五条从哪来，讲完就清楚。

>

> 每一条我都给一个怎么测，要求不能测，等于没写。

>

> 保密性：信息只被授权人访问，即使被截取也不能了解真实含义。后半句是重点，看到了也看不懂。怎么测：越权访问、抓包看明文、看返回字段是不是多了。

>

> 完整性：保护信息和处理方法的准确性与原始性，包括数据一致性，防止被非法篡改。怎么测：篡改请求参数看服务端校不校验，改传输中的数据看有没有签名。

>

> 可用性：授权用户在需要时能访问信息。怎么测：压力下的可用性、故障时的降级。

>

> 真实性：来源真实可靠。它问的是你是谁、这话真是你说的吗，对应认证与签名。怎么测：伪造 token、重放请求。

>

> 不可抵赖性：用户对自己的行为不能否认。怎么测：关键操作有没有留痕，日志普通用户能不能删改。

>

> 可追溯性：实体的行动能被跟踪。它和不可抵赖是一对，一个靠签，一个靠记。怎么测：做一次敏感操作，看日志能不能还原出谁、何时、从哪、做了什么。

>

> 可控制性：对信息的传播和内容有控制力，敏感数据不能导出、不能转发、加水印。怎么测：试着导出、截图、批量拉。

>

> 可审查性：出了问题能提供调查的依据和手段，这是事后能力。怎么测：模拟一次事件，看现有日志够不够用。

>

> 八条之间也不是并列的。可追溯做不住，不可抵赖就只剩口头承诺。可审查做不住，前面七条出了事也没法复盘。

>

> 前三条是 CIA，后五条是延伸。口诀：保、完、可、真、不、追、控、审。

> **提问：**不可抵赖性和可追溯性常被当成一回事，区别在哪？

> **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，后者是事后能力。

- 答"日志也能做不可抵赖"——可以讨论，但普通日志管理员能改，强度不够。要不可抵赖得用防篡改手段，签名、只追加存储。

- 收口：不可抵赖是能当证据，可追溯是能还原过程。

> **易错点 / 考点：**八条要能说全，前三条 CIA 必答。不可抵赖与可追溯是本节最典型的混淆点。最容易漏的是可控制性、可审查性，补上能加分。每条都要能说出怎么测，这是从概念走到实践的那一步。

> **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）

> 六对六连线，我不报答案，先给方法。

>

> 左边六类威胁：假冒 ID、篡改、抵赖、信息泄露、拒绝服务、特权升级。右边六个手段：授权、保密、不可抵赖性、可用性、完整性、认证。

>

> 这类题只问一句：这个威胁要发生，缺的是哪道防线。右边给的是控制手段，不是目标名，所以答案要落到用什么办法上。

>

> 示范三次，剩下的你自己对。

>

> 假冒 ID 配认证。假冒要解决的是你到底是谁，那就得有一个验证身份的机制。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混淆点。

>

> 篡改配完整性。防改，还得改了能发现。

>

> 抵赖配不可抵赖性。留痕、签名，让他赖不掉。

>

> 剩下三条：信息泄露配保密，拒绝服务配可用性，特权升级配授权。特权升级是权限边界被突破，防它靠的正是授权控制和最小权限。

>

> 六对六，正好一一对应。这就是 STRIED 的另一半：模型之所以好用，是因为它天生带着该用什么挡的答案。

>

> 这道题还有变体。给你六个手段，问各自能挡哪几类威胁。答案不成一对一，比如审计能挡抵赖，也能在事后追责。变体考的还是同一件事：机制和目标之间不是一根线。

>

> 答题格式也补一句。别只写"假冒 ID 配认证"这一行，把中间那层补上：假冒 ID、真实性、认证。三个词写全，看卷的人一眼就知道你懂机制。

>

> 考试技巧补一句。先连最容易的，篡改和拒绝服务最好认，把确定的划掉，剩下的自然浮出来。别在难题上卡住。

>

> 时间也别全花在连线上。留两分钟，把三组形近词各写一句区别，那才是会考的地方。

提问：为什么假冒 ID 配认证，不配授权？

> **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，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。

- 收口：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：六组配对必须会，这是本节最直接的考点。认证和授权是安全领域第一号混淆点。做题时先连确定的。答题最好把机制也写上，比如抵赖、不可抵赖性、审计日志和数字签名这样一串写完。

> **过渡：**答案页不重复题目，只讲这六组背后的三条规律。

【7.57】练习（2 分钟）

同一道题的答案页。答案本身不值钱，值钱的是这三条规律。

>

规律一，威胁和防线是一对一。六类威胁能配上六个手段，不是巧合。这个模型本来就是按防护手段切出来的分类法。做题时别猜，问自己：这个威胁要成，缺的是哪一道防线。

>

规律二，长得像的词要盯住差别。认证对授权，你是谁，你能干什么。完整性对不可抵赖性，一个防改，一个防赖。保密性对真实性，一个防看，一个防假。

>

规律三，目标词和控制词要分开。列表里既有保密、可用性、完整性这种目标名，也有授权、认证这种手段名。答题时两者都对，但要能说成一句话：用什么手段实现什么目标。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。

>

这三条不只对这道题有用。你以后读任何安全标准，威胁、控制、目标三层词都混着出现，能分清，才算读懂了那份标准。

>

还有个操作值得做。六组配完回头检查一遍，看有没有两个威胁配了同一个手段。真出现这种情况，多半是某个词你理解偏了。

>

这三条规律写需求时也用得上。你写一条安全需求，最好能说清它挡的是哪类威胁、实现的是哪个目标。三样写全，这条需求就糊弄不过去。

>

再补一句，题做对不算本事，能在自己写过的代码里找到这六个威胁才算。回头看看你的登录、下单、改资料，哪一处可能被假冒，哪一处可能被篡改。

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，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。

- 收口：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：**三组形近词必须一句话分清：认证与授权、完整性与不可抵赖、保密与真实。错答最多的是把信息泄露配给完整性。答题可以写成威胁、目标、手段三段式，得分更稳。

> **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）

> "系统要安全"，这句话写进需求文档，验收的时候谁都没法签。

>

> 课件把安全需求分成两类：数据安全性和系统安全性。

>

> 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。这六项盖住数据的整个生命周期。数据结构管数据长什么样、哪些字段敏感怎么分级。安全存储管怎么落盘、怎么隔离权限。安全访问管谁能读谁能写。认证管谁能碰，加密管传的时候怎么护，备份管丢了怎么办。备份也算安全需求，勒索软件这几年把这条顶到了最前面。

>

> 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。认证是门锁，权限限制是门里的分区。异常处理最容易被当成编码规范，它其实是安全需求。一个报错页把堆栈、表名、框架版本全交出去，等于给对手画了张地图。入口验证要记全，接口、文件上传、消息队列、定时任务参数，都算入口。系统强壮性是打不坏，异常输入不崩，压力下不崩，依赖挂了也不崩。

>

> 分工一句话：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。数据加密做得再好，系统有个越权漏洞，一样全泄。

>

> 两类之间还有交叠。备份属于数据安全性，可备份文件要不要加密、谁能恢复，又落回系统安全性。

>

> 写需求还有一条规矩，每条都要可验证。"系统要安全"没法测，"口令必须加盐哈希存储，任何接口不得返回口令字段"就能测。这和性能需求要可测量是同一个道理。

> **提问：**异常处理为什么被列成安全需求，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"因为异常路径常跳过校验"——很专业。很多越权和注入就发生在异常分支，正常分支有校验，异常分支没有。

- 收口：所以异常处理的安全要求有两条，对外不泄露细节，对内不跳过校验。

> **易错点 / 考点：**两大类和各自包含的项要能列举，填空题高频。备份属于安全需求这一点常被漏掉。异常处理的双重要求是加分点。判断题"安全需求可以写得原则一些"是错的，安全需求必须可验证。

> **过渡：**需求讲完，挑一样每个系统都有的东西走一遍，就是口令。

【7.59】示例：系统安全性问题（2 分钟）

> 一个十二位、大小写加符号的密码，安全吗？先别答。

>

> 课件问的是，对用户口令，有多少防范措施。把口令的一生走一遍，每一步都问这里能出什么事。

>

> 设置的时候：有没有强度要求，有没有挡住弱口令库，有没有禁止和用户名相同。

>

> 存储的时候：不能明文。用加盐哈希，每个用户一个随机盐，配 bcrypt、scrypt、PBKDF2 这类慢哈希。加盐是防彩虹表，慢是让爆破的算力成本高到不划算。这一条是开发最常犯的错，用 MD5，甚至明文。

>

> 传输的时候：走加密通道，否则中间人直接拿走。

>

> 使用的时候：失败要不要限次数，要不要防爆破，验证码、锁定、延迟，失败日志记不记。这一条对着口令猜测。

>

> 找回的时候，最容易被漏。安全问题答案能不能猜，重置链接有没有过期，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，直接点忘记密码。

>

> 变更与注销的时候：改密码验不验旧密码，异地登录提不提醒，离职账号停不停，会话超不超时。

>

> 一圈走完，结论是，口令安全不是一个功能，是一条链，最弱一环决定整体强度。所以"密码策略很严还是被盗"不奇怪，链条另一头是漏的。

>

> 还有一件事常被漏，默认口令。设备、后台、测试环境装完就带着 admin 这类初始口令，很多事故从这儿开始。交付前要把默认口令清一遍。

>

> 设计时还有个取舍。慢哈希会让登录变慢，参数要调在一个平衡点上：用户等得起，爆破的人算不起。

>

> 现在的做法是加一层多因素，短信、TOTP、生物特征，不指望单点绝对可靠。

提问：一个系统要求密码 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"——策略只盖住了链条的一环。

- 答"弱，找回是短板"——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。链接是不是一次性、会不会过期、绑不绑会话。

- 收口：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：**六个环节要能说全，找回最容易被漏。加盐哈希三要素必答：盐、慢哈希、禁明文。判断题"密码复杂度高就安全"是错的，链条思维才是正确答法。最弱一环决定整体强度，这句话可以套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、客体这些词来讲。

【7.60】系统安全的基本机制（3 分钟）

> 用户点一下"查看订单"，系统内部要判断几件事？至少三件：你是谁，这个订单是不是你的，你能不能看。

>

> 课件用的是访问控制的经典模型，几个词先立住。

>

> 主体，主动的一方，用户、进程，它发起操作。客体，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，允许不允许做这次访问。

>

> 权限是主体乘客体乘操作的三元组。这一点要记牢：同一个用户对 A 文件可读、对 B 文件不可读，这才叫权限。笼统说"这个用户有权限"，不成立。

>

> 再往上一层。安全策略是一整套规则，规定谁在什么条件下能做什么。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 由系统按标签统一强制执行，用户自己改不了。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，用在高

> 安全等级的系统里。

>

> 串成一句。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。做完由审计记下来，强制访问控制保证策略不被绕过。

>

> 这三个词不是拿来背定义的。设计用例时，把主体、客体、操作三列写出来，一行一行问该不该允许，缺的用例自己就冒出来了。

>

> 测试人员懂这个模型有直接用处。越权漏洞就是这个三元组某一项坏了。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角色。未授权访问是根本没验主体。拿三元组去设计用例，比凭感觉点要系统得多。

> **提问：**一个用户能看到别人的订单，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"只做了功能权限，没做数据权限"——也对。能进这个页面，不等于能看这些行。

- 收口：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：**主体、客体、访问、权限四个术语的定义，加上"权限是三元组"这个理解，是最容易被问倒的地方。MAC 与 DAC 的区别是高频考点。安全策略定规则、安全审计记事实，两者的定位不要混。加分项是用三元组解释水平越权和垂直越权。

> **过渡：**机制有了，防线还得落地，课件后面给了三道。

【7.61】安全三大保障机制：预防、恢复、支撑（2 分钟）

> 楼道里的灭火器一年一检，标签贴得整整齐齐。可真问一句，上次疏散演练是什么时候？没人接得上话。安

> 全上最常见的一种自我安慰，就藏在这个场面里。

>

> 安全保障其实分三类。第一类预防，目标是不让坏事发生。手段你们都熟：身份认证、访问控制、输入校验、加密、防火墙、权限最小化。钱也是这一类花得最多。可预防一定会有漏。攻击手法在变，你不可能把所有路提前堵死。

>

> 第二类恢复，目标是出了事能快些回到正常。备份与恢复演练、冗余与故障切换、应急响应预案、回滚机制，都算。它背后是一次观念上的转向：从"砌一堵打不穿的墙"，改成"假设墙会被打穿，但我能很快修好"。承认自己会被打穿，听着不体面，可这是设计恢复能力的前提。

>

> 第三类支撑，最容易被忽略。它管的是让前两类长期有效：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。没有它，预防会随系统演进而失效，新接口上线没加校验，谁也没发现；恢复会随人员流动而失传，备份文件在，没人知道怎么恢复。

>

> 三类各管一段：预防降概率，恢复降损失，支撑保长效。现实里最常见的失衡，是预防占九成，恢复和支撑几乎没有。结果一次成功的攻击，直接变成灾难。

>

> 有个省事的检验法，问团队三句话。我们最可能被怎么打穿？打穿之后多久能恢复？谁保证这些措施一年后还有效？三句里有一句答不上，就是某类机制缺了。

提问：一个团队把预算全花在买安全设备上，没有备份，也没做过应急演练。这个体系缺什么？

预设回答：

- 答"缺恢复"——对，一旦被打穿就回不去，小事故会变成灾难。

- 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，设备堆着不等于配得对，"买了"和"生效"是两件事。

- 收口：评估一套安全体系，看的是三类齐不齐，不是看花了多少钱。

易错点 / 考点： ① 三类机制的名称和代表手段要能对上，选择题常配对；② 高频判断"安全就是防住攻击"——只覆盖预防，漏了恢复和支撑；③ "假设会被打穿"这个观念转变是理解题常考；④ 加分句：预防降概率、恢复降损失、支撑保长效。

过渡： 三类机制要落地，得有成文的策略，下面看一份完整的清单。

【7.62】完整的安全策略示例（2 分钟）

有人把安全策略写成三十页，打印装订，放进抽屉。半年后审计要看，翻出来一对，条目和现在的系统已经对不上。问题不在长短，在这份文档有没有结构。

>

课件那份清单条目很多，看着杂。按"管什么"归成四组，就顺了。

>

第一组管身份：口令保护、可接受的使用控制、敏感信息控制。它回答的是人怎么证明自己、能碰什么。第二组管边界：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。还有无线通信控制、Internet DNS 设备控制、路由器安全管理、第三方网络连接协议。凡是"从外面连进来"的路，都得有策略。这里最容易忘的是无线通信和拨入这两条。现在大家默认网络就是一根网线，可会议室的 AP、运维的 4G 卡、供应商的专线，都是入口。

>

第三组管主机与数据：服务器安全管理、数据库信任编码。后者说的是数据库这一层的可信代码和权限约束，不是应用挡住了就完事。第四组管流程：防病毒流程、审核和风险评估、引入评估控制、应用程序提供上（ASP）控制。还有自动转发电子邮件控制、可接受的加密控制。引进新系统、新供应商要做安全评估，这条常被当成采购流程，其实它属于安全策略。

>

归完类你会发现，这份清单在回答四个问题：谁能进、从哪进、进来能干什么、怎么管。

>

还有个措辞值得停一下。"可接受的"出现了两次，可接受的加密控制、可接受的使用控制。它传达的态度很

> 明确：策略不是越严越好，是划出可接受的范围。范围不清的严，最后会变成大家都不遵守。

> **提问：**策略里为什么写"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"——是个原因，但不是核心。

- 答"要划清可接受范围"——对。"可接受"给的是选项集合和判断标准，执行的人能落地。

- 答"场景不同要求不同"——很准，传输、存储、备份的加密要求本来就不一样。

- 收口：策略的写法是给出可接受区间加判断依据，不是喊一句绝对化的口号。

> **易错点 / 考点：**① 四组归类比死记条目省力，答题也好看；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 四个问题（谁能进、从哪进、能干什么、怎么管）能直接搭论述题的骨架。

> **过渡：**策略是规矩，那测试要验到什么程度？先看目标。

【7.63】安全性测试的目标（3 分钟）

> 一条安全用例写完，怎么判断它测的是重点，还是在边角上打转？

>

> 先看目标分几层。通常的目标四条，进一步的目标三条，它们都指向同一件事。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。这句是总纲，每条用例最后都要回到它。

>

> 第一条，强制访问控制。验证该进的能进、不该进的进不去。括号里"强制"两个字有分量，它指不靠用户自觉，由系统执行。测法很直接：换身份、换角色、换参数，看边界守不守。

>

> 第二条，身份鉴别。验证"你是谁"这件事判得对不对，包含口令策略、多因素、会话管理、token 有效性。测法是伪造、重放、过期以后再拿来复用。

>

> 第三条，完整性保护。数据在存储和传输中不能被改，改了要能发现。测法是改参数、改报文、改文件，看有没有校验和告警。

>

> 第四条藏在"进一步"里，却是测试最直接的产出：发现潜在的安全漏洞。这条不用多解释，你们实验交的报告主要就写它。

>

> 进一步的目标三条。隐蔽通道分析，找那些设计者没打算用来通信、却能被用来传信息的路径。比如响应时间的细微差别，比如两种错误码的差异。这条是高等级系统的要求。可信路径，验证用户和安全功能之间那条路没被插一层。你看到的登录框，得真是系统的登录框。加密算法、代码规范，验证用的是不是合规算法。MD5、DES 这类已经破了的不能再上，代码有没有守住安全编码规范。

>

> 收一句：设计任何一条安全用例，先问它验的是哪种数据状态下的哪个属性。问不出来，这条多半是凑数的。

> **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，防的就是你和系统之间被插了一层。

- 答"防钓鱼页面"——很贴切，你以为是系统的登录框，其实是伪造的。

- 答"防会话劫持"——相关，通道不被插入、不被篡改，会话才守得住。

- 收口：测可信路径，看证书、看域名、看有没有强制跳 HTTPS、看登录页能不能被套框架。

> **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 隐蔽通道分析要能自己解释并举例；③ 高频判断"安全测试的目标就是找漏洞"——不全面，还要验访问控制、鉴别、完整性这些功能是否有效；④ 总纲句 CIA 乘存储、传输、处理，答题时好使。

> **过渡：**目标定了，范围还得划清，看它分几块。

【7.64】软件安全性测试的范围（3 分钟）

一个系统认证有、加密有、审计也有，报告上敢不敢写"安全良好"？

>

范围分两块。第一块安全功能测试，验证系统宣称的安全功能真有效。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。再加访问控制、审计跟踪、委托、隐私保护、安全管理。它的性质是正向验证：需求说密码要加盐哈希，我去看是不是真加了；需求说只有管理员能删用户，我去看普通用户删不删得掉。判据是需求有没有被实现。

>

第二块安全漏洞测试，从攻击者角度，以发现漏洞为目的。课件给了定义：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。四个面都要看，很多人只盯实现，设计和运维的洞一样致命。

>

两块差在哪，三组对照。视角不同：功能测试站在设计者这边，看该有的有没有；漏洞测试站在攻击者那边，看不该有的有没有。依据不同：前者拿需求文档，后者拿攻击模式和漏洞库。完备性不同：需求条目有限，功能测试可以测完；攻击手法在演进，漏洞测试永远测不完。

>

测不完这件事有个直接后果：漏洞测试必须排优先级。按什么排？资产价值、攻击面暴露程度、利用难度，三个乘起来。前面讲威胁模型时用的是同一套算法。

>

还得澄清一个误解：功能测试替代不了漏洞测试。系统可能把安全功能全实现了，认证、加密、审计一样不缺，越权漏洞照样在。功能是一点一点的点，漏洞长在点与点之间的缝里。反过来也一样。两块都得做。

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。功能实现了，和能不能被绕过是两件事。

- 答"不能，还要做漏洞测试"——对。有认证，认证可以被绕过；有加密，密钥硬编码在代码里。

- 答"要看有没有做越权和注入"——很具体，这两类正是功能齐全但洞还在的典型。

- 收口：报告要分两部分写，功能有效性结论一份，漏洞清单一分，别合成一句"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"做了渗透就不用做安全功能测试"——渗透测不出需求要求的功能在不在；③ 漏洞测试永远测不完，由它推出要排优先级；④ 口诀：功能管点，漏洞在缝里。

> **过渡：** 说到功能测试和安全测试，还有一组更基础的区别得掰开。

【7.65】功能性测试 vs. 安全性测试（3 分钟）

有两句原话，值得一字一字读。功能测试是"软件做它应该做的事"，安全测试是"软件不做它不应该做的事"。

>

> 差别就压在"应该"和"不应该"上。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。

>

> 拿一个转款接口说。功能测试关心转一百块，扣一百，对方收到一百，余额算得对。安全测试关心另外几件事：转负一百，会不会变成给我加一百；把收款人换成别人，会不会成功；同一个请求重放十次，会不会扣十次。前面是照着说明书用，后面是反着用、乱着用。

>

> 课件还有两个词：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，靠校验、鉴权、限流；危险发生之后的处理，靠降级、隔离、告警。它们正好对上前面说的预防和恢复两类机制。所以安全测试不只要测挡不挡得住，还要测挡不住的时候怎么办。

>

> 还有个差别容易被忽略：缺陷的性质不同。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，背后已经能被利用了。这一条决定了安全缺陷更依赖主动的、专业的测试，等用户反馈就太晚。

>

> 落到手法上，有一条通用路子：拿一条功能用例，改一个不该改的参数，或者换一个不该有的身份。订单金额改成一分，用户 id 换成别人的，角色从普通用户改成管理员。改完还能通过，问题就出来了。

> **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，安全把金额改成 0.01"——典型对照，一个验规则，一个验绕过。

- 答"功能验库存不足提示，安全验并发会不会超卖"——也好，安全用例常是并发加边界的组合。

- 答"安全还要看别人的订单能不能看"——对，那是越权，功能用例里一般不写。

- 收口：设计安全用例有个通用手法，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 两句原文几乎必考，"做应该做的事"对"不做不应该做的事"；② 危险防止对预防，危险处理对恢复，要能连起来说；③ 高频判断"安全测试就是功能测试加边界值"——视角和依据都不一样；④ 通用手法要背：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）

> 写安全测试计划，最常见的写法是列一张扫描清单。清单能交差，可它盖不住任务的全貌。

>

> 课件给了四条任务。第一条，全面检验需求规格说明里规定的防止危险状态措施是否有效，以及每一个危险状态下的反应。这句有点绕，拆两半。前半是措施有没有效：需求写了连续输错五次就锁定用户，那就验第六次真锁了。后半是危险状态下的反应。真被锁、真被限流、依赖真宕机的时候，系统给一句明确提示，还是白屏，还是把内部信息漏出去。

>

> 第二条，针对设计里用来提高安全性的结构、算法、容错、冗余、中断处理这些方案，做针对性测试。它打的是设计层的机制：加密算法用得对不对，冗余能不能真切过去，中断处理会不会留下不一致状态。这类要专门设计用例，随机点两下碰不到。

>

> 第三条，在异常条件下测，证明不会因为单个或多个输入错误就进不安全状态。注意"单个或多个"这四个字：一个单引号是单个，超长加特殊字符再加并发是叠加。这一条是模糊测试和健壮性测试的依据。

>

> 第四条，对安全性关键的单元、组件单独做加强测试。登录、权限、支付、加解密，这几块要拎出来，评审、单元测试、专项攻击测试一起上。这跟关键模块重点测的一般原则一致，只是安全场景里"关键"的意思变成了一旦被攻破后果最重。

>

> 四条排下来是四个层次。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。计划按这四层列活动，不容易漏项。

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。

- 答"模糊测试是自动化的异常测试"——基本准确，靠随机或变异输入去触发异常。

- 答"异常还包括环境异常"——很专业，依赖宕机、网络断、磁盘满、时钟跳变，都算。

- 收口：答"怎么在异常条件下测"，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 第一条的后半（危险状态下的反应）最常被答漏；③ 安全性关键组件要单独加强测试，这是概念题；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，这就引出等级。

【7.67】安全等级（3 分钟）

验收会上有人问，这套系统按几级测？答不上来，后面的深度就没法谈。

>

> 我国把计算机信息系统的安全保护分成五级，从低到高。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。

>

> 死记五个名字容易混。换个记法，记每一级多了什么。

>

> 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，也就是 DAC。系统不做强约束。它适用于被破坏后损害公民、法人权益，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。谁做了什么要留下来，事后能追。名字里就写着关键词。

>

> 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，访问控制按标签走。这一步是从自主走向强制，因为标签不是用户能随手改的，控制权从用户手里收到系统手里。

>

> 第四级，结构化保护级，要求安全机制本身结构化：安全功能独立、结构清晰、能形式化地说明。它面向被破坏后严重损害社会秩序和公共利益的系统。

>

> 第五级，访问验证保护级，最高一级。访问要被验证，安全机制要能被数学证明，还得扛得住高强度攻击。用在关系国家安全的关键系统上。

>

> 五个词连起来：自主、审计、标记、结构化、验证。顺序不能乱，乱了说明你记的是名字，不是能力。

>

> 测试人员为什么要背这条链？等级决定测试深度和验收标准。该评三级的系统你按一级测，是失职；二级的系统你拿五级标准去要求，成本扛不住。

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"——太笼统，得说出多了什么。

- 答"多了安全标记，访问控制从自主变强制"——准确。标记用户改不了，控制权就收到系统手里。

- 答"第二级加了审计"——那是第二级相对第一级的差别。

- 收口：记这条链的办法是记每级新增的能力，不是记名字。

> **易错点 / 考点：** ① 五级名称与顺序要能默写，选择题高频；② DAC 到 MAC 的转变发生在第三级，这是理解性考点；③ 高频错答"等级越高功能越多"——准确说是安全机制更强、保证更严；④ 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，课件用一张表摊开。

【7.68】不同等级的安全要求（2 分钟）

> 标准文档里那种大表，一页几十个格子，多数人从第一格开始背，背到第三行就放下了。

>

> 这类"等级—要求"矩阵到处都有，会读比会背管用。读法三步。

>

> 第一步看行是什么。行是安全要求的类别。自主访问控制、强制访问控制、身份鉴别、客体重用、审计。再加数据完整性、隐蔽通道分析、可信路径、可信恢复。这些名字你在前面都碰过。访问控制、鉴别、审计是刚才那几条目标；隐蔽通道和可信路径也在；可信恢复对应的就是恢复机制。

>

> 第二步看列是什么。列是五个等级，要求逐级追加。低等级只有基本项，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。

>

> 第三步看格子里写什么。通常写要求或不要求，也有写部分或全部的。抓住上升趋势就够了：等级越高，要求的项越多，强度也越高。

>

> 这张表教给你的，是一种感觉：安全能力是分层的，不是全有全无。给一个系统做安全测试，第一步该问它

> 属于哪一级、对应哪些要求，再逐项验。这比凭经验想到哪测到哪专业得多。

>

> 再补一句实践上的提醒：标准里的等级是应达到的底线，不是最高标准。业务上需要更高，比如涉及支付，就得在等级要求之上再加。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"——那样会陷进细节里。

- 答"先看行和列分别是什么"——对，先把坐标系建起来。

- 答"先找逐级增加的规律"——更好，抓住趋势，细节用到时再查。

- 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"高等级包含低等级"——一般成立，但要理解它是追加关系；③ "合规是底线"这个观点要能展开；④ 这一页考得浅，重点是把名词认全。

> **【思政落点 7】**：合规是地板，不是天花板，这句说的是标准和自觉的关系。等保测评过了，只证明你踩到了最低那条线，不证明系统守得住。有人把过审当终点，测评一过就松手，新接口、新供应商、新数据都不再回头看，等于把责任外包给一张证书。站得住的态度是：标准给底线，底线之上的判断自己做，还得说得出理由。守规矩只是入门，替用户多想一步才是本分。

> **过渡：**等级不同，安全功能测试的内容也跟着变。

【7.69】不同安全级别的安全功能测试（2 分钟）

> 同一个删除订单的按钮，一级系统和五级系统的测法，工作量能差出十倍。

>

> 差别在深度。把安全功能测试按等级摊成三层看。

>

> 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，权限分配和校验；审计，有没有记、记得全不全。这三项是底线，也是现实里最容易出洞的三项。绝大多数漏洞落在这一层。

>

> 第二层增强项，中高等级追加：细粒度访问控制，细到数据级、字段级；强制访问控制的标签管理；客体重用；更严的审计，防篡改、集中存储。客体重用最容易被忽略，它管的是资源释放之后，下一个使用者不能读到上一个人的残留。内存、临时文件、连接池，都算。

>

> 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、形式化或半形式化的验证。

>

> 三层其实在回答一个问题：这套系统的安全功能测试要测到什么程度。答案不是拍脑袋，是等级和业务风险一起定。

>

> 不管哪一级，用例的组织方式一样。按鉴别、授权、访问控制、审计、数据保护排下去。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。

>

> 还有一条实践差别：等级越高，越要文档化。方案、用例、结果、残余风险，都要能拿去评审。

提问：一个删除订单的功能，测"客体重用"要关注什么？这一项为什么容易被忽略？

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"还要看连接池和内存复用"——很专业，同一块内存被下一个请求复用，不清理就会读到别人的数据。

- 收口：测法是 A 用完、释放、B 申请，再查 B 能不能看到 A 的痕迹。

易错点 / 考点：① 三层划分（基础、增强、高级）便于记忆，也能撑起简答；② 客体重用要能解释并设计测法；③ 高频判断"安全功能测试只测正向"——必须正反都测；④ 等级越高越要文档化，这条实践差别常被漏。

> **过渡：**测什么讲完了，该讲怎么测。

【7.70】安全性测试方法（3 分钟）

> 工具买了一堆，真到排测试计划，还是不知道先上哪一个。

>

> 方法可以摊成两维来看。第一维是静态和动态。

>

> 静态三项：工具扫描、分析代码，不跑程序，按规则找可疑模式；人工评审架构和代码，人读，找逻辑漏洞，这一项工具替不了，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。

>

> 动态三项：功能测试方法，按需求验安全功能；模糊测试方法，喂随机或变异的数据看会不会崩；渗透测试方法，像攻击者那样打一轮。

>

> 第二维更要紧：基于威胁，还是基于漏洞。基于威胁是从软件外部看安全性，识别软件面临哪些威胁，再测这些威胁会不会真发生。它的出发点是"谁会怎么打我"，关心的是场景。基于漏洞是从软件内部看，识别已知的安全漏洞，借助漏洞扫描工具。它的出发点是"我身上有哪些已知弱点"，关心的是清单。

>

> 两者分工清楚：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，投入产出比高。新问题，这个业务特有的逻辑漏洞，只有威胁方法挖得出来。所以顺序是先漏洞后威胁，先快后深。

>

> 静态和动态怎么选，也有一句话：静态看写的时候就有问题，动态看跑起来才出问题。少一行校验这种错，读代码能看出来；并发下的竞态，只有跑起来才现形。

>

> 时间只够一轮的时候，务实顺序是这样。扫描铺面，人工评审看业务逻辑，再挑高价值模块做威胁分析和渗透。

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，最像真实攻击"——效果直观，可周期长、覆盖有限。

- 答"扫描加人工评审"——务实，扫描清老问题，评审看业务逻辑。

- 答"先做基于漏洞的，再做基于威胁的"——最合投入产出，老问题占比最大。

- 收口：顺序是扫描快、评审准、高价值模块做威胁分析和渗透，一层比一层深。

> **易错点 / 考点：**① 静态三项、动态三项要能列全；② 基于威胁和基于漏洞的视角差别是本节核心，简答高频；③ 高频错答"渗透能覆盖所有安全问题"——覆盖不了，还不可重复；④ 静态看写错、动态看跑错，一句顶十句。

> **过渡：**六种方法里，功能测试最容易被小看，看一个具体例子。

【7.71】安全性功能测试（2 分钟）

> 一个口令能出多少条用例？十几条，条条都能抓出真问题。

>

> 先说功能测试在安全里做什么。课件列了两类主项：身份认证测试，比如 DCE/Kerberos、基于公钥的认证机制；访问控制策略验证，比如入口访问、权限、目录这些控制。再加几类通用动作：规则验证、负面测试、传输检查、存储检查、控件兼容性。

>

> 这几个词得掰开。规则验证，验策略说的是不是真在执行：策略写密码九十天过期，就看第九十一天是不是真要求改。负面测试，专测不该成功的是否失败：错误口令、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，直接翻数据库和文件，口令是不是哈希、关键字段是不是加密。控件兼容性，验证码、加密控件、证书这些，换个浏览器、换个版本还能不能用。最后这条最容易被忘，控件不兼容，用户连登录页都过不去，可用性直接崩。

>

> 落到口令上，清单是这样：强度策略生效吗；弱口令被拒吗；口令加盐哈希存储吗；传输加密吗；连续失败锁定吗；锁定之后怎么解、多久解；改口令要不要验旧的；重置流程安全吗；改完密码，旧会话失效吗。

>

> 最后一条最要命。用户改密码，动机往往是怀疑被盗。旧会话还能用，等于没改。

>

> 一个口令十几条，这就是安全功能测试的样子：不是找一个惊天大洞，是把每条规则逐项验到底。它枯燥，挡住的却是最常见的那批问题。条目多、易漏，靠脑子记一定漏，做成检查单，做一次打个勾。

> **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"为了体验"——不是体验问题，是安全问题。

- 答"攻击者可能已经拿到旧会话"——正是。改密的动机常常就是怀疑被盗。

- 答"要同时踢掉其他设备"——更完整，这就是改密即全端下线。

- 收口：这条用例的本质是，凭据一变，旧凭据立刻作废。

> **易错点 / 考点：**① 通用动作要能列举：规则验证、负面测试、传输检查、存储检查、控件兼容性；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"能当简答的落地措施写。

> **过渡：**按规则验完了，那不按规则怎么验？下一类方法专干这个。

【7.72】模糊测试方法（3 分钟）

> 让一个没看过需求的人乱点你的系统。他点到崩的那个地方，往往就是缺陷。

>

> 模糊测试干的就是这件事。课件定义：在一个被测试程序上附加随机数据（fuzz）作为输入。程序出问题，比如崩溃或者异常退出，就能定位缺陷。

>

> 定义短，想法聪明。它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：测试设计极其简单，系统的行为先入为主。这句要解释。"测试设计极其简单"，指你不用写几十条精心设计的用例。只要一个能生成输入的生成器，加一台会跑的机器。"系统的行为先入为主"，指你不用预先知道正确的输出是什么，只看程序有没有异常。这一点很关键：功能测试必须先知道预期结果，模糊测试不需要。所以它能测那些说不清正确行为的输入空间。

>

> 它擅长找崩溃、内存错误、解析异常、没处理的边界，也就是程序被搞挂这一类。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。

>

> 代价也有：大量无效输入，覆盖率不一定高，结果还得去重。所以现代模糊测试都配覆盖率引导，只有能触发新代码路径的输入才留下。

>

> 用在哪？所有解析外部输入的地方：文件格式，图片、音视频、文档；网络协议；序列化和反序列化；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，都是它的猎场。

>

> 它的路子，是拿机器的时间和数量，去换人想不到的那几个输入。

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"——补得好，有些不崩但也算异常。

- 答"还能靠断言和差异对比"——很专业，加内部断言，或者拿两套实现做差分。

- 收口：它的判据是异常，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 定义和"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漏

洞"——错，它擅长崩溃类；③ 不需要预期结果这个特征要记住；④ 加分：说出覆盖率引导是它的演进方向。

> **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）

同一个程序，三种喂法，挖出来的洞能差出一个数量级。

>

> 第一种，黑盒随机模糊。对正确格式的输入数据做随机变异，再拿变异后的输入跑程序，看会不会触发异常。关键在"从正确格式出发"：不是从一堆随机字节开始，而是拿一个能跑通的样本，改几个字节。命中率比纯随机高得多。它实现极简，缺点是不懂格式，很多变异在解析第一关就被拒了，算力白烧。

>

> 第二种，基于语法的模糊，是模糊复杂格式输入的替代办法。要先指定三件事：输入格式的语法是什么，哪些部分要模糊化，怎么模糊化。说白了，你先把格式的语法写出来，比如 JSON 的语法、某个协议的报文结构。再按语法生成合法但语义异常的输入。字段类型对、值超长；必填字段缺一个；嵌套深到爆栈。它比黑盒聪明的地方，是知道哪一层才是有效攻击面，不会在语法层就被挡回去。

>

> 第三种，白盒模糊处理，课件说出自微软研究院，2008 年首创。做法是：动态执行被测程序，从执行中遇到的条件分支收集输入约束；再系统地逐个否定这些约束，用约束求解器求解，解映射成能走不同路径的新输入；用系统搜索反复做，尽量把程序所有可行路径扫一遍。

>

> 翻成人话：程序跑到一个 if，就把"要进这个分支，输入得满足什么"记下来；把条件反过来，让求解器算出什么样的输入能走另一条路；拿算出来的输入去跑，就到达新路径。反复做，路径就铺开了。这就是符号执行那一套。

>

> 取舍很直白：黑盒最便宜，覆盖最浅；语法模糊居中，适合结构化格式；白盒最强，但要求解器，开销大，复杂程序还会路径爆炸。工程上多是混着用。

> **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"随机输入大多在解析层就被拒了"——正是，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，变异后还有机会过前面的校验。

- 答"因为覆盖率更高"——说到底就是这个，有效输入才推得到更深的分支。

- 收口：模糊测试的效率，取决于输入生成器有多懂格式，这就自然引出语法模糊。

> **易错点 / 考点：**① 三种模糊器的名称和核心思路要能对上；② 白盒模糊四步（收集约束、否定约束、求解、搜索）是简答高频；③ 高频错答"白盒模糊就是看源码"——它是动态执行加约束求解；④ 微软研究院 2008 这个出处别记错年份。

> **过渡：**白盒模糊的思想有个著名的工程实现，就是 AFL。

【7.74】American fuzzy lop 方法（3 分钟）

> 一个开源工具，把瞎试变成有方向地试。它靠两个前提，六步循环。

>

> 工具是 AFL，American fuzzy lop，课件说它是当今用得最广的 Fuzzer。两个前提：一，它在程序执行前对源码插桩，执行过程中实时拿到程序的执行情况；二，它用遗传算法对输入做变异，把输入当基因，把新覆盖当适应度。

>

> 插桩得说清楚。往代码里插探针，程序跑到哪个分支、走了哪条边，都被记下来。这一步是它聪明的基础，它知道自己这次的输入有没有带来新覆盖。

>

> 六步。第一步，把用户给的初始测试用例加载到队列里，也就是给它几个能跑通的种子。第二步，从队列里取下一个输入文件。第三步，把用例修剪到不改变程序测量行为的最小尺寸。这一步很妙：输入越短，后面变异越容易命中关键字节。第四步，用均衡的、经过充分研究的各种传统模糊策略，反复对文件做变异。改字节、翻位、插入、删除，一轮接一轮。第五步，如果某个变异让插桩记录到新的状态转换，就把这个变异

> 输出作为新条目加进队列。第六步，回到第二步。

>

> 第五步是全部的关键。只有带来新覆盖的变异才被留下，成为新种子。有反馈，才叫覆盖率引导。去掉它，队列会被重复又没价值的输入塞满，工具退化成普通随机模糊。

>

> 所以它的核心不是变异，是筛选。用覆盖率当路标，让变异朝没走过的路走。

>

> 实践两点：种子要好，能跑通、覆盖主要功能；时间要给够，跑十分钟和跑十天，结果不是一个量级。

> **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，队列还会被重复又没价值的输入塞满。

- 答"退化成普通随机模糊"——就是这样，没有反馈就没有引导。

- 答"算力浪费在同一个分支上"——对，这正是覆盖率引导要解决的问题。

- 收口：AFL 的核心不是变异，是筛选，用覆盖率当适应度函数。

> **易错点 / 考点：** ① 六步流程要按顺序说得出，尤其第三步修剪和第五步入队条件；② 两个关键词：插桩（instrumentation）、遗传算法与覆盖率引导；③ 高频判断"AFL 是纯随机测试"——错，它是有反馈的；④ 课件给的 GitHub 和文档链接是自学入口，记名字就行。

> **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用例够不够。

【7.75】变异测试（3 分钟）

> 用例全绿，可能是程序对，也可能是用例软。这两种情况在报告上长得一模一样。

>

> 变异测试就是来分这两种情况的。先说名字撞车：模糊测试变异的是输入，变异测试变异的是代码，两回事。

>

> 课件定义：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，或者强制产生有效的测试。定义绕，换个说法。它问的不是程序有没有 bug，是我的用例真能发现 bug 吗。

>

> 做法：故意往源码里注入一个小错，这叫变异；然后跑你原来那套用例。用例失败了，说明它抓住了，这个变异体被杀死；用例照样全绿，说明错从网眼里漏过去了，变异体存活。

>

> 举个例子。代码里原本是 if (a > b)，变异成 if (a >= b)。跑完还是全过，就说明你的用例里没有一条覆盖到 a 等于 b 这个边界。这一条边界，恰恰是线上最常见的错。

>

> 几个术语要分清。变异体，变异后的程序版本。杀死，测试发现了它。存活，测试没发现。变异得分，杀死的变异体数除以全部变异体数。分越高，这套用例的杀伤力越强。

>

> 变异不是乱改，是按算子改。大于改成大于等于，加改成减，true 改成 false，删掉一行，改掉返回值。这些正好对上程序员最常犯的错：边界写错、符号写反、漏了分支。

>

> 它贵。每个变异体都要跑一遍完整测试套件，变异体动辄几百上千。所以实践里要抽样，只对改动的代码做增量变异，能并行就并行。这也是它在工业界不如单元测试普及的原因。

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。跑了没断言，等于没测。

- 答"不能，还要看断言"——对，覆盖加断言强度才是完整判断。

- 答"可以用变异测试验证"——最专业，往代码里注入错误，看用例抓不抓得住。

- 收口：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 定义和目的（测测试，不是测程序）必须说清，这是最大得分点；② 四个术语要能解释：变异体、杀死、存活、变异得分；③ 和模糊测试的变异区分开，一个变输入，一个变代码；④ 高频判断"变异测试用来找程序缺陷"——错，它评估用例。

过渡：定义有了，落到操作是五步，一步步走。

【7.76】变异测试的方法与流程（3 分钟）

上一页给了定义，落到手上是五步，第三步才是信息最多的地方。

>

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。改动按算子来：大于改成大于等于，与改成或，常量零改成一，删一条语句，返回值取反。为什么是这些？因为它们模拟的正是最常犯的错：边界写错、条件写反、少一个分支。

>

第二步，拿原来的测试用例去跑变异体。用例一条都不许改，也不许为它新写。改了就是作弊，测的就不是原来那套用例的本事。

>

第三步看结果。用例失败，变异体被杀死；用例全过，变异体存活。存活才是我们要的信息，每存活一个，用例集上就有一个洞。

>

第四步，排除等价变异体。有些改动在语义上跟原程序一样，比如把 i++ 写成 i 等于 i 加一，行为没变，任何用例都杀不死。这类要剔掉再算分，不然分数永远上不去，写用例的人还被冤枉。

>

第五步算分：杀死的变异体数，除以变异体总数减掉等价变异体数。这个百分比，就是这套用例的检测力。

>

有个地方容易混：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。程序本身可能是错的，用例照样杀得死变异体，因为变异体就是从错的程序上改出来的。这一点答大题要写清楚。

>

> 还有，等价变异的判定靠人，带主观性。答题时补一句"需要人工或工具辅助识别"，分数会好看些。

> **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，说明漏了"——方向对，但太急，容易误判。

- 答"不一定，可能是等价变异体"——准确。等价变异怎么都杀不死，得先剔掉再评价。

- 答"也可能那段代码根本没被执行"——也对，覆盖不到的变异体，谈生死没意义。

- 收口：看到存活先问三句：是不是等价变异、变异点在测试范围内吗、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，它只说明用例对错误敏感；② 等价变异体的判定靠人工，带主观性，大题要写"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；④ 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，看用例抓不抓得住。下面换个角色，让攻击者去造错。

【7.77】渗透测试（3 分钟）

> 扫描器给你一份疑似有漏洞的清单，客户看完还是不知道该不该怕。

>

> 渗透测试补的就是这一块。课件定义：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探测，发现最脆弱的环节，直观地发现问题。每个词都有信息量。探索式，说明它不是照清单打勾，是按反馈随时调下一步。模拟黑客，说明视角是敌对的。最脆弱的环节，说明它不追求全覆盖，找那个一敲就碎的点。直观最要紧，它不给你一份疑似列表，是把系统真打穿一次给你看。

>

> 工作链四段。信息收集：用搜索引擎、开源平台、产品资料，把域名、IP、技术栈、员工、版本号摸清。这一步看着不技术，可它决定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、这次打什么，手段是攻击树分析加攻击模式库。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：出报告，写风险等级、影响面、修复建议。

>

> 有一条边界绕不开：渗透测试是拿到授权的对抗性测试。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？因为它在技术动作上和真实入侵几乎一样，区别只有那张纸，和旁边盯着的人。

>

> 它跟漏洞扫描的差别也在这。扫描给的是可能有这些漏洞，渗透给的是我已经从这条路走进来了，证据在这。

> **提问：**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，但要讲清产出是待验证清单，误报得人工确认。

- 答"渗透，能证明危害"——两天只够打一两个点，那就挑最值钱的那个点打穿。

- 答"先扫后打"——实务里最常见的组合，扫描铺面，渗透打点。

- 收口：时间紧就先铺面再打点，无论哪一种，范围得有人签字。

> **易错点 / 考点：**① 四阶段顺序常考填空；② 易错：把威胁分析和风险评估混了——威胁分析在攻击之前，回答打哪里；风险评估在攻击之后，回答多严重、怎么修；③ 探索式不等于随便打，它是按反馈调整策略；④ 扫描与渗透的关系要写"扫描是渗透的一个子环节"。

> **【思政落点 8】**：渗透这门手艺天然有两面。同一套手法，用来找破口是测试，用来撬别人的系统就是入侵，动作几乎一样，分界在授权和目的。所以这一节的重点不在怎么打进去，在打进去之后做什么：出报告、定等级、给修复方案，把风险交回系统负责人手上。将来不管做测试、开发还是运维，手上握着别人的数据，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。

> **过渡：**这条四段链在工程里被切得更细，看业界通用的五阶段。

【7.78】渗透测试—概览（3 分钟）

> 从看到打，切换点在哪儿？图上那根竖线就是答案。

>

> 业界把渗透测试切成五个阶段，左边一块叫 exploration，探索；右边一块叫 exploitation，利用。这个二分比四段更实用，它回答的是什么时候该动手。

>

> 探索有三段。网络发现：主机探索、端口扫描、服务识别、指纹识别、网络映射。顺序有讲究：先知道哪些机器活着，再看开了哪些端口。再看端口后面跑什么服务、什么版本，最后画出整张拓扑。指纹识别是这里的关键。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，因为漏洞的价值差得远。一个能远程执行代码的洞，和一个信息泄漏的洞，处理优先级差着数量级。

>

> 利用这一段三步。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，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，弹一个无害的框，而不是真把数据拖走。权限提升，从普通用户提到管理员，从一个系统跳到另一个。

>

> 最后是修复与报告：威胁消除、未来计划、报告。未来计划这一项常被跳过，它的意思是要把这次的发现转成后续的加固路线，不是打完就完。

>

> 整张图的落点：渗透测试是有起点有终点的流程，不是一次炫技。

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，生产环境真会被打挂。

- 答"证明能进去就够了"——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，日志和证据链被自己搅乱，复盘就说不清。

- 收口：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，不是攻击；② 指纹识别的作用是缩小漏洞范围，不是认品牌；③ 常考简答"渗透与漏洞扫描的区别"，要写扫描是渗透的子环节；④ 概念性证明的分寸是理解题考点。

过渡： 流程清楚了，具体拿什么手段去打？看教科书里最典型的五类。

【7.79】渗透测试的一些具体方法（3 分钟）

内网就安全，这句话在攻防里最经不起推敲。

>

具体手法有五类，由外到内看。

>

跨网段、VLAN 之间的渗透。它打的就是那个错误假设。很多系统把管理后台放内网，觉得外面进不来。可边界上一台机器被拿下，横向移动就开始了。测法是从一台低权限主机出发，看能不能碰到不该碰的网段。防线怎么设：网段隔离、访问控制列表、最小权限。

>

端口扫描，用 X-Scan、NMAP 这类扫描器。它的价值是摸清哪些门开着。堵法很土也有效：关掉不必要的端口和服务，别把数据库、缓存、管理口挂在公网上。

>

远程溢出。课件给的例子是一套现成工具组合。溢出工具 SQL2.exe 加监听工具 nc.exe，打 SQL Server 的 1433 端口。原理是往服务发超长或畸形的数据，覆盖内存里的返回地址，让程序跳去执行攻击者的代码。应对是打补丁、开 DEP 和 ASLR 这类缓解、把数据库端口收回内网。这些工具在实验环境里是教学素材，换个地方就是另一回事，我们的实验只在自己的靶机上做。

>

本地溢出。前提是已经拿到一个普通用户的权限。再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。提权常常只需要一个没打补丁的驱动，或者一个配错的服务。做法是及时打补丁、把服务进程降权、把审计开起来。

>

> 最后一类，脚本及应用测试。利用脚本层面的弱点，拿 Web 服务器或数据库目录的访问权限，严重的时候能拿到系统控制权。上传、注入、包含、反序列化，都属于这一类，后面 Web 安全测试会展开。

>

> 五类连起来是一条路：看见门、推开门、走到里屋。

> **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，可它是后面所有动作的入口，暴露面本身就是风险。

- 答"口令猜测"——对，课件也把它列进了方法清单，一个弱口令能让前面的防护全作废。

- 答"脚本及应用测试"——从损失看确实最大，Web 是唯一对公网开放的那一层。

- 收口：门槛最低、危害最大常常是同一件事，暴露面没管住加强口令。加固第一步永远是关掉不要的东西、改掉默认口令。

> **易错点 / 考点：**① 远程溢出和本地溢出的区别必考：远程溢出不需要任何用户身份，本地溢出必须先有普通用户权限；② 防御措施至少写三条：补丁、最小权限、网络隔离；③ 课件把口令猜测写进方法清单，答题别漏；④ 脚本及应用测试对应后面的 Web 安全测试，位置要清楚。

> **过渡：**手法有了，可这些动作都很重，一不小心就把系统打挂，所以实施得配监控策略。

【7.80】渗透测试实施策略（2 分钟）

> 打得进去不算本事，打完系统还活着才算。

>

> 实施策略五条，条条都在控风险。

>

> 全程监控：用类似 Wireshark 的嗅探软件全程抓包。为什么？留证据，每一步发了什么包、收到什么响应，事后能复盘；能追责，系统一出异常，立刻判断是不是我们打的；能止损，看到不该有的响应就收手。

>

> 择要监控：扫描过程不录，等数据分析完、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。取舍很清楚：拿完整性换开销。

>

> 主机监控：只盯受测主机的存活状态。最省的一种，适合只要不挂就行的系统。它跟择要监控不是一回事：择要监控盯的是流量，只是时间上推后；主机监控盯的是主机还活着没有。

>

> 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。多团队同时在一个环境里测的时候，这条特别有用，出了问题要能定位是谁的动作。

>

> 第五条最实用：对关键系统，拿目标的副本做渗透。把生产克隆一份到隔离环境里打，打崩了不影响真实用户。代价是副本和生产可能有差异，所以结论要标注"在副本环境验证"。

>

> 五条合起来回答一个问题：风险不只来自漏洞，也来自测试动作本身。一个负责的测试者，先想怎么保证不出事，再想怎么打进去。

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"挑深夜做，业务低峰"——务实，可夜间值班人少，出事响应慢，得配好应急联系人。

- 答"先单请求验证，确认无害再放大"——很专业，这叫最小冲击，和 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：生产环境测试三件套，时间窗口、单点验证、随时可停。缺一件就别开工。

易错点 / 考点：① 五种策略的名称和适用场景要能对上；② 易错是把择要监控和主机监控搞混：前者监控流量，只是时间上延后，后者监控主机存活，对象不同；③ 简答常问"为什么关键系统用副本"，答降低对生产的影响、可反复验证、便于对比；④ 五条的共同点是控制测试动作本身的风险。

过渡：渗透只是安全测试的一种手段，它还得铺到整个开发生命周期里，下一段接着讲。

【7.81】贯穿 SDLC 的安全性测试（2 分钟）

> 安全这件事，很多团队是这么安排的：上线前一周，请人打一遍。打完了出份报告，改几个高危，发布。这个安排不算错，但它只能抓住一部分问题。

>

> SDLC 是软件开发生命周期。把它摊开，五个阶段各有各的安全动作。

>

> 需求阶段，定安全需求、定合规要求、写滥用案例。什么叫滥用案例？正常用例写"用户怎么转账成功"，滥用案例写"攻击者怎么绕过限额转账"。这一步不做，后面所有测试都没有判据。你说不安全，依据在哪？没有依据，测试人员只能凭感觉。

>

> 设计阶段，威胁建模、体系结构风险分析。要问四件事：数据放在哪、谁能碰、边界划在哪、信任关系有几层。为什么这一步最贵？改一张架构图，可能推翻半个系统；不改，就是带着结构性问题上线。

>

> 编码阶段，安全编码规范、静态分析、代码评审。缓冲区溢出、SQL 拼接、密钥写死在代码里，这些读代码就能拦下，不用等人打进来。

>

> 测试阶段，安全功能测试、模糊测试、渗透测试。这一节后面讲的主体都在这儿。

>

> 运维阶段，补丁、日志监控、应急响应、漏洞预警。上线不是终点，是新风险的起点。

>

> 为什么非得"贯穿"？因为不同阶段能发现的漏洞类型不一样，代价也不一样。需求阶段漏一条安全需求，最后不是改代码，是改产品逻辑。流程要重画，界面要重做，培训要重来。

>

> 判断标准给一句：只在发布前一周做一次渗透测试的团队，做的不是安全性测试，是安全检查。检查问的是"有没有明显的大洞"，测试问的是"这一路的风险有没有被管住"。

> **提问：**一个项目已经进了测试阶段，你才被叫去做安全。哪些本该更早做的事，你还补得回来？

> **预设回答：**

- 答"补需求"——可以，换个形式：从现有功能反推安全需求，整理成一份检查单。

- 答"补威胁建模"——可行，也值得。架构还在，画数据流图、标出信任边界，一两天能出初版。

- 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、时间挤。

- 收口：越晚介入，越要靠反推补课。反过来看，安全人员进项目的时间得再往前挪。

> **易错点 / 考点：**阶段和动作的连线题最爱考。最常见的错：把威胁建模放到测试阶段，把渗透测试放到编码阶段。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：**位置讲完了，剩下一个方针问题：软件安全按什么原则组织。

【7.82】软件安全实施方针（2 分钟）

> "我们买了最好的防火墙和 WAF"——这话我在答辩里听过好几次。问下去，内部接口没有鉴权，日志里打着明文手机号。

>

> 课件给的两句话，是这一节的总纲。第一句：软件安全是系统级的问题，既要考虑安全机制，也要考虑基于安全的设计。

>

> 拆成两半。安全机制是认证、授权、访问控制、加密、审计，这些是能加上去的部件。基于安全的设计不是加部件，是让系统天生难打。权限默认拒绝，出错时往安全的方向关，关键操作二次确认，敏感数据不落地，组件之间互不信任。

>

> 差别在哪？一个是给房子换一把好锁，一个是从结构上让它不好进。锁能撬，结构不好撬。

>

> 为什么强调系统级？安全有个很硬的性质：它由最弱的一环决定。前端做了校验，后端照样拼 SQL，一样被打穿。库里加密了，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。一个模块守住自己那一亩地，缺口却常常出现在两块地之间：接口、日志、备份、运维通道，谁都不认领。

>

> 第二句：软件安全必须是完整开发生命周期方法的一部分。这句在否定一种常见做法——安全是安全部门的事，等项目做完请他们来把把关。安全如果是外挂的，它永远只是补丁。只有嵌进流程里，它才成为系统本身的一部分。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。

>

> 两句合起来，方针就八个字：系统级设计，全流程嵌入。少一条都不成立。

> **提问：**"我们买了最好的防火墙和 WAF，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，也挡不住业务逻辑漏洞。

- 答"WAF 只是缓解，不是修复"——准。它拦的是特征，改不了代码里那句拼接。

- 答"没算上人"——也算一条。弱口令、配置错误、被社工，设备都管不到。

- 收口：安全是分层防御，边界、应用、数据、权限、流程各守一段。指望一层解决全部，等于一层都没有。

> **易错点 / 考点：**判断题"安全性是安全测试人员的工作"——错，它是全员、全流程的事。另一个错法：只答安全机制，漏掉基于安全的设计。答题时"访问控制"和"健壮设计"两个例子都要写上，这是课件明确给的。

> **过渡：**要嵌进流程，就得有个说得清的框架。业界最通行的是 SDL。

【7.83】全生命周期的安全性测试（3 分钟）

> 同一条缺陷，早发现和晚发现，代价差得不是一点。设计阶段发现是改一张图，上线后发现是改一个系统。

>

> 课件给的定义是这样。把风险分析、安全测试和软件开发过程结合起来，也就是 SDL，Security

> Development Lifecycle。在各个阶段做误用模式、异常场景、风险分析、渗透测试，尽早发现高风险的安全漏洞。

>

> 里面四个动作词，对应四种思路，逐个拆。

>

> 误用模式。正常用例写"用户按规矩操作得到什么"，误用模式写"有人不按规矩操作能得到什么"。正常是"输入手机号获取验证码"，误用是"同一号码一秒请求一百次"。它从需求阶段就能开始写，比一张漏洞清单更贴业务。

>

> 异常场景。传空、传超长、传特殊字符、乱序调用、并发调用、超时重试。这一项测的是系统的边界和假设。高危漏洞最爱藏在"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。它决定测试资源怎么分。你不可能把每个角落都测一遍，力气必须压在高风险处。

>

> 渗透测试。用攻击者视角做验证，前面讲过。

>

> 这句话的落脚点在"尽早"两个字。为什么？架构级的安全缺陷拖到上线后修，系统在跑、数据已经有、用户已经养成习惯，修复只能打折。

>

> 再补一层：SDL 不是一门新技术，是一套把安全动作排进开发节奏的管理办法。它要求每个阶段有交付物、有检查点。设计阶段交威胁模型，编码阶段过静态扫描。没有交付物，"做了安全"就只是一句口头声明。

> **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"一个正面一个反面"——方向对，再深一层：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"——到位。绕过前端校验直接调接口，就是典型的误用思路。

- 答"差不多，换个说法"——不一样。前提不同，产出也不同：误用模式产出攻击场景清单，用例产出输入输出对应关系。

- 收口：做安全测试的第一件事，是把"用户都很老实"这个假设从脑子里删掉。

> **易错点 / 考点：**SDL 的全称和含义常考填空，Security Development Lifecycle。四个动作的名称要写全：误用模式、异常场景、风险分析、渗透测试。错法是把 SDL 当成某个工具或某个标准，它是过程框架。

> **过渡：**SDL 在设计阶段最重的产出，是体系结构的风险分析。

【7.84】体系结构风险分析（3 分钟）

> 改代码能改掉一个 bug，改不掉一个结构。这一节看的就是结构。

>

> 分析对象是体系结构，不是代码。这个区别重要在哪？代码层面改得再干净，结构错的地方还在。结构性问题，写多少行代码都救不回来。

>

> 课件给了三样东西：工件、风险例子、一条方法论。

>

> 工件是设计和说明书。意思是干这件事不需要代码，只要架构图、部署图、接口说明、数据流说明。这也解释了它为什么能在设计阶段就做——它靠读文档就能看出风险。

>

> 风险例子两个，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？就是没分级。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，保护强度当然也不同。全用明文存、全用一套权限，就是区分很糟糕。

>

> 第二个，Web 服务没有验证调用方代码，也没有验证它代表的用户。服务之间互相调，只认"请求来自内网"，不认调用方是谁、它替谁办事。那内网里随便哪个人，都能以任意用户的身份调接口。这就是内部接口裸奔。

>

> 方法论的句子有四层意思。体系必须连贯一致，不能东边做了鉴权、西边忘了。要有统一的安全防线，安全得有个收口的地方，不是每个模块各写一套。文档要写清前提假设——"这个接口只给 App 调""这个字段前端会校验"，这类话一旦不成立，漏洞就出现了。风险要持续管，各阶段都会冒新风险，得一直看。

>

> 所以这份分析得在设计阶段做。等系统跑起来再回头看结构，要改的就不只是文档了。

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。接口没有独立鉴权，抓个包就能重放。

- 答"App 里的校验等于没校验"——对。客户端校验只算体验优化，别当安全边界。

- 答"那就在网关上限制"——方向对，但要说清限制什么：IP、频率，还是身份？前两个都能绕。

- 收口：凡是写成"只有 X 会来调"的假设，都要追问一句——如果 Y 来了，系统认不认？

易错点 / 考点：工件是设计和说明书，不是代码，常考判断。那句"没验证调用方代码、也没验证它代表的用户"要能翻译成"服务间调用缺身份校验"，答题时补一句"服务身份和用户身份都要验"。还有"持续的风险管理"这半句，多数人只记要做风险分析，漏了要持续、要监控。

过渡：设计阶段看结构，编码阶段看代码。往下就进代码里了。

【7.85】代码评审示例（3 分钟）

打开一个文件，给你一小时，让你看出里面的安全问题。你从哪一行看起？

>

代码评审是静态测试，不跑程序，靠人读。这是第 3 章的内容，接在这儿：它最便宜，也最容易被跳过。

>

> 安全视角的评审，我收成一张检查单，六条。做课设可以直接拿去用。

>

> 第一条，输入校验。外部来的数据——请求参数、请求头、上传文件、Cookie、第三方回调——服务端有没有校验？看代码时找的是"直接从请求里取值、然后拼进语句"的地方。前端校验不算，前端在别人的浏览器里。

>

> 第二条，SQL 和命令拼接。看到字符串拼出来的 SQL、拼出来的 shell 命令、拼出来的文件路径，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。敏感操作前有没有查"当前登录人有没有这个权限、这条数据是不是他的"？最常见的漏洞不是"没登录也能访问"，是"登录了，但不是你的数据你也能看"。下面就有例子。

>

> 第四条，加密和密钥。密码有没有用不可逆的哈希加盐？密钥有没有写死在代码或配置文件里？传输有没有走加密通道？

>

> 第五条，日志和错误信息。日志里有没有明文密码、身份证号、完整卡号？异常有没有把堆栈直接返给前端？错误信息越详细，攻击者越省事。

>

> 第六条，失败模式。出错的时候是拒绝还是放行？很多越权漏洞的根就在异常分支里少了一个 return。

>

> 评审的产出不是"我看过了"。是问题清单、责任人、修复确认。没有记录的评审，等于没做。

> **提问：**评审时看到这么一段：先查订单，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"要看这个异常能不能被外部触发"——很专业。构造一个让查询抛异常的入参，这就成了可利用的越权。

- 答"加个 try-catch 就好"——方向反了。正确做法是失败即拒绝，不是把异常吞掉继续走。

- 收口：这类问题叫失败模式不安全，写进检查单，每份代码都过一遍。

> **易错点 / 考点：**代码评审是静态测试，和"跑起来看结果"的动态测试相对，这是第 3 章的知识点，常跨章考。简答问"评审能发现哪类安全问题"，答注入、硬编码密钥、越权、信息泄漏、不安全失败模式。错法是把前端校验当安全措施。

> **【思政落点 9】**：一个 return 漏在异常分支里，看着是笔误，代价是别人的数据被看到。这六条检查单说的其实是同一件事：替不在场的人想一遍。写代码的人只走正常路径，读代码的人替用户、替同事、替半年后的自己去看那些走不通的路。这份自觉落到价值观上，就是对自己交付的东西负责。你签下的不是"能跑"，是"我查过了，查的就是这几处"。

> **过渡：**逐行看，看得准，可项目那么大，不可能每行都看。得有个分配力气的办法。

【7.86】基于风险的安全测试（3 分钟）

> 力气永远不够。放在安全测试上，这句话最实在。一个系统几百个接口，你只有两周。

>

> 课件给的是策略层的答案，工件写的是单元和系统——这套策略既用在单元级，也用在系统级。策略两条，并列，不是二选一。

>

> 第一条，用标准功能测试技术做安全功能性测试。意思是安全功能本身也是功能，可以用等价类、边界值、判定表来测。"密码错五次锁定"是一条功能需求，用边界值测四次、五次、六次。"只有管理员能删用户"，用判定表把角色和操作组合起来。这一条最容易被学生漏掉。一听安全测试就想到渗透，其实大半安全需求用普通功能测试手段就能验。

>

> 第二条，以攻击模式、风险分析结果、滥用案例为基础的基于风险的安全测试。三个输入。攻击模式是业界

> 已知的手法库。风险分析结果告诉你哪里值钱、哪里暴露。滥用案例就是前面讲的误用模式。输出是针对高风险点的定向测试。

>

> 为什么必须有第二条？课件那句话很硬：标准的质量保障方法，可能揭示不了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。

>

> 全页点睛的一句：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。功能测试问"需求要的发生了吗"，安全测试问"没写进需求、但可能发生的，防住了吗"。

>

> 落到课设：你的安全测试用例表最好两栏，一栏来自需求，一栏来自攻击模式。两栏都空的项目，等于没做安全测试。

提问：需求文档写"用户可以修改自己的昵称"，按功能测试这条需求合格。从安全角度，你还想补什么用例？

预设回答：

- 答"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。把用户 ID 换成别人的。

- 答"超长昵称、emoji、控制字符"——也算，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，安全测试补的是这条路径周围所有的旁门。

易错点 / 考点：两条策略要一起写，只答"做渗透测试"不完整。那句"安全测试是为了保证坏的事情不会发生"常作简答引语，答的时候点出"和功能测试的目标相反"。另一个错法：把"基于风险"理解成"高风险就不测"，反了，是高风险多测、低风险少测。

过渡：策略清楚了，看最常见的战场。Web 为什么单独占一大块？

【7.87】Web 安全性测试（2 分钟）

> Web 这一块在课件里单独占一大截，为什么？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。

>

> 它为什么天然难防？三个原因。

>

> 第一，输入不可信，而且必须收。Web 应用天天干的事，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。这个转换里只要有一处没做对，就是注入。

>

> 第二，状态靠客户端维持。HTTP 本身无状态，服务端靠 Cookie 或 Session ID 认人。Cookie 存在用户自己的浏览器里。存在别人机器上的东西，就得假设它会被偷、被改、被伪造。会话劫持、CSRF 这一整类问题都从这儿来。

>

> 第三，权限判断散在每一个接口里。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。接口是后来一个个加的，权限判断也是后来一个个补的。补的时候，谁也说不清全系统一共多少个入口。

>

> 所以测试思路也就三条。测输入输出，管注入和 XSS。测身份和会话，管认证、Session、劫持。测授权，管越权、直接对象引用、功能级权限。后面几页就按这三条往下走。三条里最容易松手的是第三条，前两条还有工具帮着看，第三条全靠人懂业务。

>

> 再补一句。Web 安全测试不是拿工具扫一遍。工具只覆盖已知模式。真功夫在理解这个业务的数据流。业务逻辑漏洞——"优惠券能不能重复领""订单金额能不能改成负数"——没有哪个扫描器扫得出来。

> **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"可以写自动化脚本测"——可以，但那不是通用扫描，是为这个业务专门写的用例。

- 收口：安全测试的核心能力，是把业务规则翻译成"可以被违反的假设"。

> **易错点 / 考点：**简答"为什么 Web 是安全测试重点"，答题点是公网可达、输入不可信、会话在客户端、接口多导致授权分散。只答"Web 用户多"不是技术原因。

> **过渡：**Web 往下就是数据库。这条链上的检查点，课件收了五条。

【7.88】Web/数据库 渗透测试点（3 分钟）

从浏览器到数据库，中间隔着好几层。每一层都有它的破口。课件把这一条链上的检查点收成五条。

>

> 第一条，检查应用系统架构，防止用户绕过系统直接改数据库。这话听着抽象，场景很具体。数据库端口对公网开着，连接串写在前端配置里，中间件管理台还用着默认口令。攻击者一旦连上库，前面所有应用层防护全部作废。测法是从外网扫端口，看数据库、缓存、管理后台通不通；翻客户端代码里有没有连接串。

>

> 第二条，检查身份认证模块，防止非法用户绕过认证。要测的是"能不能不做认证就成为已登录用户"。再往下看：接口是不是全部强制鉴权？Token 能不能伪造？验证码能不能复用？登录失败有没有次数限制？有没有默认账号？

>

> 第三条，检查数据库接口模块，防止用户拿到系统权限。这里的系统权限指数据库或操作系统那一层。典型问题是注入之后能执行数据库扩展、能读写文件、能调系统命令。再一个是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，防止用户拿到系统文件。上传、下载、导入、导出、预览，每个都是文件接口。要测三件事。能不能传一个可执行脚本再访问到？能不能靠路径穿越读到配置文件？下载接口能不能越权拿

> 别人的文件？

>

> 第五条，检查别的安全威胁。这是兜底：日志泄漏、敏感信息没脱敏、接口返回字段过多、调试接口没关、第三方组件有已知漏洞。

>

> 五条的价值在于它给了你一张按模块划分的检查地图。课设里按这五条各写一组用例，覆盖面不会太差。

提问：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。这是最严重的那一类，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，但根在架构，不能只当信息泄漏处理。

- 答"要看这个账号有什么权限"——关键的追问。只有只读权限，危害有限；是业务主账号，等于库已经丢了。

- 收口：判严重程度看两件事：暴露面多大（外网还是内网）、这份凭据能干什么（只读还是可写）。

易错点 / 考点：五条的顺序和内容常考。别把"数据库接口模块"和"文件接口模块"混着答：前者防权限提升，后者防文件读取。"防止用户绕过系统直接修改数据库"是架构层考点，答题要写数据库不对公网暴露、应用是唯一访问入口。

过渡：地图铺好了，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）

四种攻击，名字都听腻了。可它们错的地方各不相同。课件把这四类列出来：注入攻击、跨站点脚本攻击、跨站点请求伪造、未能限制 URL 访问。

>

> 第一类，注入。系统的错在：把用户给的数据当成了代码。SQL 注入是把数据当成 SQL 语句的一部分，命令注入是把数据当成 shell 命令。XML 注入是把数据当成文档结构，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、用结构化接口，别用字符串拼。

>

> 第二类，XSS。系统的错在：把用户给的数据当成给别人的代码。你自己看这段数据没事。它被存下来、显示给另一个用户，就在对方浏览器里跑起来了。所以 XSS 是借你的浏览器，动你的账户。

>

> 第三类，CSRF。系统的错在：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。任何页面诱导你发一个到 A 站的请求，A 站都认为是你本人在操作。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。系统的错在：把"页面上没有链接"当成了"访问不到"。隐藏不是保护。URL 在，直接敲进去就能开。后台管理系统里这类情况最多：菜单不给你，页面还挂在那儿。

>

> 四类的共同点，听得出来吗？全是服务端过度相信客户端。相信它传来的数据无害。相信它显示的就是全部。相信它的请求是自愿的。相信它看不见的链接就访问不到。这一句，就是整个 Web 安全的总纲。

提问：这四类里，哪一类完全可以在一个"没有任何输入框"的页面上发生？

> **预设回答：**

- 答"CSRF"——很准。受害者只要访问了恶意页面，请求就发出去了，全程没在哪个输入框里打字。

- 答"未能限制 URL 访问"——也对，连输入都不用，改地址栏或者直接用工具发请求就行。

- 答"XSS 总得有输入框吧"——不一定。反射型靠链接参数，存储型靠昵称、评论、文件名，甚至日志展示页。

- 收口："这个页面没有表单所以安全"是典型错觉。攻击面取决于服务端收哪些参数，不是页面上画了什么。

> **易错点 / 考点：**四类的中英对照要对上。CSRF 的译名是"跨站点请求伪造"，常被写成"跨站脚本伪造"。XSS 和 CSRF 最容易混：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，后者不用看到任何返回。

> **过渡：**四类里最常见、也最容易被写进课设的，是 XSS。

【7.90】针对 XSS 进行测试（3 分钟）

一行 `javascript:alert(document.cookie)`，为什么能在别人的浏览器里跑起来?

>

课件给的说明很简洁：允许恶意用户把代码植入到"给别的用户看的 Web 页面"里。这句话有两个要素，一是代码，二是给别人看的页面。缺一个都不构成 XSS。

>

课件那个例子是：在一般输入域里能输入这段脚本，然后在别的页面显示出来。为什么能成功？因为服务端把用户提交的内容原样放进了 HTML。浏览器解析时看到脚本标签就当代码执行，看到 onerror 就当事件处理器执行。它分不清哪句是程序员写的、哪句是用户填的。

>

危害分三层，一层比一层重。第一层，有隐藏性。脚本藏在正常页面的内容里，管理员看页面根本看不出来。第二层，受害者的敏感数据、session、cookie 会泄漏。脚本能读 document.cookie，能读页面上任何内容，还能悄悄发给攻击者。第三层，攻击者能监控用户行为，还能让用户被迫访问别的站点。到这一层，就不是偷东西，是控制这个浏览器。

>

测试按三个位置分。反射型：payload 放进 URL 参数，看回显页面有没有原样输出。存储型：payload 存进库——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 之类的手段插进页面，这类要看前端代码，不是看响应体。

>

防御四条一起上。输出编码，按上下文做 HTML、属性、JS、URL 编码。输入校验走白名单，富文本尤其要。CSP，禁掉内联脚本。Cookie 加 HttpOnly，就算被注入也读不到 Cookie。

>

转义要做在输出的地方，不是输入的地方。

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"同一份数据在不同地方要不同的转义"——非常准。进数据库要 SQL 转义，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，但要清楚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：数据保持原样存储，按使用场景在边界处编码。

> **易错点 / 考点：**三型的区分是高频考点：反射型要诱导点击，存储型不用交互、危害最大，DOM 型在服务端日志里看不到 payload。判断"对输入做了长度限制就能防 XSS"——错，短 payload 一样能执行。答题要分开写"输入校验"和"输出编码"。

> **过渡：**XSS 是往页面里塞脚本，下面这一类是往 SQL 语句里塞条件。

【7.91】SQL 注入式攻击（3 分钟）

> 一个单引号，能登进别人的账号。听着夸张，往下看就不夸张了。

>

> 课件的定义是这样。按 SQL 语句的编写规则，附加一个永远为真的条件。让系统里某个认证条件总是成立，骗过系统、躲开认证，再进入系统。课件给的例子是 `' OR '1' = '1`。

>

> 把这条 payload 放进一段真实的登录逻辑里看。后台拼出来的语句是：SELECT 全部字段 FROM users WHERE name 等于用户输入，再接 AND pwd 等于密码。用户名填 `' OR '1' = '1`，拼完就变成 WHERE name = '' OR '1' = '1' AND pwd = '...'。'1'='1' 永远为真，这个 WHERE 对表里每一行都成立。查询返回第一条用户记录，登录成功。密码是什么，攻击者根本不需要知道。

>

> 关键不在 payload 有多巧，在拼接这个动作。SQL 是拼出来的，攻击者就能往里塞语法。SQL 是参数化的，他塞什么都只是一个字符串值。所以防御的根本手段是参数化查询、预编译。数据库先编好语句结构，再把参数当数据填进去，结构就改不了。

>

> 除了绕登录，它还能干更重的事。拖库，用 UNION 把别的表的数据拼进结果。盲注，靠页面真假或响应时间一点点猜。提权和命令执行，某些数据库支持扩展，能读写文件，甚至跑系统命令。所以它的危害等级通常在最上面那一档。

>

> 怎么测？三个层次。手工验证：参数后加单引号看报不报错，加 AND 1=1 和 AND 1=2 看响应差。报错注入：借数据库报错把数据带出来。时间盲注：构造 SLEEP 看响应时间。

>

> 防御再补三条。数据库账号最小权限，业务账号没有建表删库的权。关掉详细报错，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。

提问：如果一个系统对所有输入都过滤掉单引号，是不是就防住了 SQL 注入？

预设回答：

- 答"是"——最典型的错。数字型注入根本不用单引号，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"要用参数化，不是过滤"——这才是根本答案。过滤是黑名单思路，永远补不完；参数化是结构上根治。

- 收口：黑名单防注入，等于拿漏勺挡水。白名单加参数化才是正路。

易错点 / 考点：常考"SQL 注入的原理"，答题要写出"用户输入被当成 SQL 语法的一部分参与解析"，只写"输入了特殊字符"不得分。防御里参数化查询、预编译是必答项，往下是最小权限、错误信息屏蔽。错法：把前端长度和字符校验当防御。

【思政落点 10】：同一条 payload，写用例的人拿它证明系统能被绕过，写代码的人拿它知道"拼接"这两个字有多危险。技术本身没有立场，立场在使用它的人身上。所以课设里要演示注入，只在自己的靶机、自己的测试库上做，用例写清验证点、预期结果、修复建议三栏。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。同样的思路换一个载体，就是 XML 注入。

【7.92】XML 注入（3 分钟）

两段 XML，差别只有一个节点。可这一个节点是管理员。

>

> 正常的那段：一个 USER 节点，role 是 guest，里面有 name 和 email。role 这个属性是个权限标记。

>

> 被注入之后那段，多出来一个 USER，role 是 admin，里面是一个叫 test 的用户。原本只有访客的文档，凭空多出一个管理员。

>

> 怎么发生的？和 SQL 注入一模一样：程序把用户输入拼进了 XML 字符串。假设注册接口是"把用户填的 name 和 email 拼进一个 role=guest 的节点"。那攻击者在 email 字段里填一段内容：先把当前节点闭合，再补一个 admin 节点。拼出来的文档结构就被改写了。系统后面读这份 XML 做鉴权，读到那个 admin 节点，就把他当管理员。

>

> 所以 XML 注入和 SQL 注入、命令注入、LDAP 注入是同一件事。数据里带了结构性字符，被拼进结构化文档。数据的边界被突破，变成了新的结构。

>

> 危害三个方向。越权，像这个例子，凭空多一个高权限用户。数据篡改，改写已有节点的值，把金额、状态改掉。信息泄漏和实体攻击。XML 支持自定义实体。解析器如果允许外部实体，攻击者就能让它去读服务器上的文件，甚至发内网请求。

>

> 防御三条。用 XML 库的构造接口，别用字符串拼，这和参数化是一个道理。对写入的内容做实体转义，把尖括号、and 符、引号转掉。关掉外部实体解析，禁用 DTD 和外部实体，这条专门防 XXE。

>

> 怎么测？每个会进 XML 的字段里，试输入一段闭合结构，看生成的文档有没有多出节点。再试 DOCTYPE 和实体引用，看解析器会不会去访问外部资源。

提问：XML 注入和 SQL 注入，你觉得哪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 只是配置文件，不觉得它是攻击面。

- 答"XML，因为不知道它也能注入"——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"——不太对。SQL 注入现在有大量现成防护，框架默认参数化；反而是 XML、JSON 这类看着无害的格式容易裸奔。

- 收口：判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。拼了，就有注入。

> **易错点 / 考点：**图要能读懂，"注入后多出一个 role=admin 的 USER 节点"是必考细节。常考"XML 注入和 XXE 的区别"：前者改写文档结构，后者利用外部实体读资源，相关但不是一回事。防御必答"不用字符串拼接构造 XML""禁用外部实体"。

> **过渡：**注入类解决的是伪造数据，下面这一类解决伪造身份。

【7.93】会话劫持（3 分钟）

> 登录之后，服务器怎么知道还是你?

>

> 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，通常就是 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：因为 Cookie 失效时间很长，攻击手段一般是盗取用户 Cookie，再伪造 Cookie 冒充该用户，会话就被劫持了。失效时间长这一点，是设计上的取舍失误。为了少让用户重新登录，很多系统把 Cookie 的过期时间设成三十天甚至更久。这三十天里，这个字符串泄漏一次，攻击者就能长期冒用。

>

> 盗取的路子有哪些？课件点了最典型的一条：通过 XSS 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。还有一种最土的：截屏、拍照，把地址栏复制进聊天群。

>

> 防御分四层，按重要性排。

>

> 第一，Cookie 加 HttpOnly。这一条直接掐断 XSS 读 Cookie 的路，成本极低，收益很高。

>

> 第二，全程 HTTPS，Cookie 加 Secure 属性。让凭证不在明文信道里跑。

>

> 第三，缩短会话时长，登出即失效。服务端要真的把会话销毁，不是只清浏览器那头。

>

> 第四，绑定上下文。会话和 IP、User-Agent 做弱绑定，或者敏感操作要求二次验证。绑太死，用户换个网络就掉线。

>

> 还有一条工程做法：Session ID 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，这个 ID 就成了已登录状态，他也能用。

>

> 会话凭证被人拿走，等于他拿着你的身份在系统里走。

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，公用电脑上尤其明显。

- 答"服务端要真销毁会话，不能只清前端"——很关键。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，登录和登出都要触发 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，还能成功，这个功能就是没做完。

易错点 / 考点：会话劫持和 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、不用拿到凭证。防御里 HttpOnly 最常考，再往下是 HTTPS、登出销毁、登录后重生成 Session ID。别以为"Cookie 加密了就安全"，加密只防内容泄漏，不防被整体盗走重放。

> **过渡：**劫持是凭证被别人偷走。还有一类更基础：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）

有的系统不是被攻破的，是原本就没锁。课件列了五条无效的认证和会话管理方式，都在设计层。

>

> 第一条，密码在数据库里明文存，登录走未加密的连接。明文存密码的后果不是被管理员看到。是库一旦被拖走，这些人在别的网站的账号也危险，因为人会复用密码。正确做法是加盐哈希，用 bcrypt、scrypt 这类慢哈希，登录走 HTTPS。加盐和慢，两个词都要紧。加盐防彩虹表，慢哈希让离线爆破不划算。

>

> 第二条，密码可以被特殊操作覆盖。课件举的例子很具体：密码更改功能、密码恢复功能实现得不正确，都可能让密码被直接改掉。典型漏洞是改密码接口只验新密码不验旧密码，或者找回密码的验证码能爆破、能复用。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 里带 Session ID。为什么错？URL 会进浏览器历史、进服务器访问日志、通过 Referer 头漏给第三方站，还会被用户复制粘贴发出去。Session ID 的正规位置是 Cookie，加 HttpOnly 和 Secure，不是地址栏。

>

> 第四条，Session ID 不超时，或者 session、token、SSO token 在登出时没失效。这在单点登录里特别常见：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，认证信息、Session ID 走未加密连接传输。这一条和第一条后半句是一回事，单独列出来，是因为它管的不只是登录页。任何一个带着 Cookie 的请求走了 HTTP，凭证就摊在链路上。

>

> 五条说的是同一层的事：认证和会话是系统的入口，这里松一寸，上面加多少防护都白搭。

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，都会留一份。

- 答"Referer 会带出去"——很专业。跳到第三方时，URL 会作为 Referer 发出去。

- 收口：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：**五条常考多选和简答。错法一：把"密码加密存储"当正确答案，应是加盐哈希，可逆加密仍然不安全。错法二：漏掉"登出未失效"，它常出现在 SSO 题里。错法三：认为"Session ID 放 URL 便于多标签页共享"，这是危险设计。

> **过渡：**认证和会话之外，还有一类更隐蔽的问题：页面跳转。看着只是跳一下，实际能把你送到钓鱼站。

【7.95】未验证跳转（3 分钟）

> 点一个看起来是自己学校的链接，落到了钓鱼页。问题出在跳转参数上。

>

> 有些页面跳转根据用户输入决定，这就给了攻击者可乘之机，把用户导向恶意站点或者未授权链接。英文常叫 Open Redirect，开放重定向。

>

> 课件给了两个例子，一个偏钓鱼，一个偏越权。

>

> 第一个：`http://www.example.com/redirect.jsp?url=evil.com`。这个页面收到 url 参数就直接跳，不做校验。攻击者把链接发给受害者。受害者看到的是自己信任的域名开头，点进去却被送到 evil.com，一个做得一模一样的登录页。关键在心理层面：人判断链接可不可信，看的是域名开头那一段。所以这个漏洞是借你域名的信誉。

>

> 第二个：`http://www.example.com/boring.jsp?fwd=admin.jsp`。这个跳的是站内页面。一个看着无害

> 的 boring.jsp，通过 fwd 参数能跳到 admin.jsp。boring.jsp 自己不做权限校验，因为它"只是个无聊页面"。admin.jsp 又依赖一个错误假设：访问者是从有权限的入口进来的。这就成了一次绕过授权检查。站内跳转常被当成小事，正因为它看着只是换个页面。

>

> 防御三条。不要拿用户输入直接当跳转目标。业务必须支持，就用白名单，只允许跳预先登记过的地址，或者只允许站内相对路径。跳转前做归属校验，尤其要拦住 `//evil.com`、`\/evil.com`、`%2f%2f` 这类变形。关键页面不靠"来源"做鉴权，每个请求独立校验身份和权限。白名单也要写死在服务端配置里，不能让它从请求参数来。

>

> 怎么测？把带 url、redirect、return、next、fwd、callback 这类参数的接口找出来，逐个换成外部域名，看会不会跳出去；再试各种编码变形，看白名单能不能绕。

> **提问：**为什么"只允许跳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"——比如 `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：正确做法不是检查字符串开头，是解析成 URL 对象再比主机名，或者干脆用映射表，给每个允许的目标一个 ID。

> **易错点 / 考点：**两个例子的区别要说清：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。判断"跳转参数做了非空校验就安全"——错。防御必答白名单，答题补一句校验在服务端做。

> **过渡：**跳转是把你送到别处。接下来这一类，是让你看到不属于你的东西。

【7.96】直接对象引用（3 分钟）

> 这段代码用了 PreparedStatement，防注入是对的。可它还是漏了。

>

> 课件那段代码是这样：拼一条查询，SELECT 全部字段 FROM accts WHERE account 等于问号，用 PreparedStatement 预编译，把请求里的 acct 参数塞进第一个占位符，执行查询。参数化没问题，注入防住了。问题在另一处：它没校验这个 acct 是不是当前登录用户的。

>

> 于是攻击者改一下 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`。这就叫直接对象引用。客户端把库里的 ID、账号、文件名传上来，服务端照着用，不查归属。

>

> 为什么这么常见？因为它长得不像漏洞。开发者的想法是"用户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。这是拿正常用户的行为模式做假设。

>

> 防御三条，缺一不可。

>

> 第一，服务端做归属校验。任何"按 ID 取数据"的接口，都要用当前会话的身份去过滤，不能只靠传上来的 ID。条件里再加一个 owner 等于会话里的用户，那个 owner 从会话里拿，不从请求里拿。

>

> 第二，用间接引用。别暴露数据库主键。给每个对象发一个临时映射令牌，服务端维护令牌到真实 ID 的映射。这个映射还绑在当前用户身上。令牌被猜中，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，别让每个接口自己写一遍，因为一定有人忘。三条里第一条是必须做的，后两条是把它做得更省事、更不容易漏。

>

> ID 是客户端给的，权限是服务端判的。这两件事不能混。

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"这条数据你能不能看"。

- 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。两回事。

- 答"补一个 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、你能干什么、这条数据是不是你的。这段代码漏了第三问。

> **易错点 / 考点：**定义常考："客户端提交对象标识，服务端没校验访问权限"。别把参数化查询当防越权手段，它只防注入。防御必答"服务端校验数据归属"和"间接引用/随机令牌"，答题时强调 owner 来自会话。

> **过渡：**下面给了一个更直观的例子，在线银行的账户号。

【7.97】直接对象引用（2 分钟）

> 一行 URL，一个数字往上加，就能把别人的账户翻过去。

>

> `https://www.onlinebank.com/user?acct=6065`。就这一行，杀伤力比上一段代码还直观。

>

> 看结构：user 是接口，acct 是参数，6065 是账户号，一个连续的、可枚举的整数。这意味着攻击者不需要任何技巧，把 6065 改成 6066、6067，一个账户一个账户看过去。这叫遍历，最朴素，也最管用。写个脚本改最后一位，几百个账户几分钟跑完，日志里看着跟正常访问一样。

>

> 这里想让人体会三件事。

>

> 第一，连续 ID 本身就是信息泄漏。它告诉你系统里有多少用户、增长多快、哪个号段是新注册的。所以安全设计里有一条：对外标识尽量用不可预测的值，UUID、随机串，别用自增主键。自增主键在库里是对的，

> 摆到 URL 上就成了路线图。

>

> 第二，HTTPS 加登录不等于安全。这个 URL 是 https，用户大概率也登录了。可只要没有归属校验，登录只证明"你是某个用户"，不证明"这个账户是你的"。

>

> 第三，这类漏洞检测成本极低、修复成本也不高，出现的频率却极高。因为它是"功能写完了、能用了"，不是"写错了"。开发者想的是接口通了，测试想的是页面能打开，没有人问一句"换成别人的 ID 会怎样"。

>

> 所以它适合当课设的必测项。做法很简单。准备两个账号 A 和 B，用 A 登录。把 A 能访问的带 ID 接口全抓下来，ID 换成 B 的对象的 ID，逐条重放。返回 B 的数据，就是漏洞；返回 403 或空，才算合格。这个方法有名字，叫水平越权测试。跑完把结果记下来，附进测试报告。

> **提问：**如果系统把 acct=6065 换成一串随机字符，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。随机化只是抬高门槛，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：**结论常作案例题：给一个 URL，问什么漏洞、怎么测、怎么修。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（服务端归属校验加不可预测标识）。只说"ID 要加密"不够，加密后的 ID 照样能被重放。

> **过渡：**数据级权限漏了会出事，功能级权限漏了同样出事。下面这一类是后者。

【7.98】功能级权限控制缺失（3 分钟）

> 菜单里没有"系统管理"，不等于接口调不通。

>

> 上一页讲数据级：你能不能看这一条。这里讲功能级：你能不能调这个接口。层级不同，危害一样大。

>

> 课件先说原因，这句话值得抄。权限控制一般写在代码里，或者靠配置文件完成。可开发人员容易忘记给某些功能加权限控制代码。容易忘记四个字最真实。这不是设计能力问题，是注意力问题。一个系统几十上百个接口，权限判断散在每个接口开头，忘一个就漏一个。

>

> 课件给的例子是这个链接 `http://example.com/app/admin_getappInfo`，只有 admin 才能访问。非 admin 用户直接在浏览器里就能打开，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按角色渲染菜单，普通用户看不到"系统管理"这一项。看着安全了，其实只是没给你入口。攻击者不需要菜单。他知道 URL，或者从 JavaScript 文件里把接口清单翻出来，直接发请求就行。

>

> 所以测试方法很明确。先把接口清单拿到，看前端 JS、看接口文档、看抓包记录。再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，只有管理员才能调的，就是漏洞。

>

> 防御三条。

>

> 第一，权限校验在服务端，而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。那样一定会漏。正确做法是统一拦截。拦截器、过滤器、注解或者网关，把"这个接口要什么权限"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

> 安全设计里最要紧的一条。

>

> 功能级权限的关键不是"写对"，是"不可能写漏"。

> **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"——就是课件那句"容易忘记"，最真实的原因。

- 答"新人不知道规矩"——对。人员流动之后，新加的接口没人提醒。

- 答"代码评审能补上"——能补一部分，可评审也是人做的，同样会漏。

- 收口：工程上的解法是把"靠记性的事"变成"忘不掉的结构"——统一拦截加默认拒绝。用设计解决问题，别用纪律解决问题。

> **易错点 / 考点：**功能级权限和数据级权限（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。判断"前端菜单隐藏了就安全"——错。简答必答服务端统一拦截加默认拒绝，只写"加权限判断"不够。

> **过渡：**漏洞类型讲了一轮，轮到看工具了。工具分两类：写代码时用的，打系统时用的。

【7.99】安全性测试工具（2 分钟）

> 装一个扫描器，是不是就等于做了安全测试？

>

> 工具这一块，课件不列具体工具，先搭一个分类框架。给一个四象限，往后见到的工具都能往里放。

>

> 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。前者不用运行程序，靠解析源码或字节码找缺陷；后者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。快，漏报多。找未知的靠基于

> 模型或形式化的方法，先建行为模型，再验证安全属性。慢，能挖出深的问题。

>

> 四个格子对应四类工具。静态加已知：词汇语法分析类，模式匹配找可疑函数调用。静态加未知：约束分析、基于模型的工具，分析数据流和控制流。动态加已知：漏洞扫描器、Web 扫描器，发已知的攻击载荷。动态加未知：模糊测试、渗透测试，靠变异和人的创造。四格的分界不是工具牌子，是它看什么、它找什么。

>

> 为什么先讲框架？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"动态加已知"那一格。它扫不出业务逻辑漏洞，扫不出要多步交互才触发的漏洞，也理解不了你的权限模型。

>

> 还有一条实践建议：静态分析放左边，开发阶段；动态分析放右边，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，是接力。

提问：如果一个团队只有预算买一个工具，你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，而且能覆盖到用例没走到的代码。

- 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"——最专业的答法。代码规范、CI 完善，静态能发挥最大价值；系统已经上线、历史包袱重，动态能立刻找到暴露面。

- 收口：没有标准答案，但答题要给出判断依据，不是"我觉得"。

易错点 / 考点：SAST 和 DAST 的区别是必考点：要不要运行程序、能不能定位到行、误报率高不高。错法是把渗透测试归到静态类，它是动态的，而且要人工。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：四象限里最能看出技术在进步的一格，是静态分析。它有三代。

【7.100】静态分析工具（3 分钟）

> 看到 strcpy 就报警，这种工具你们大概都见过。它属于第一代。

>

> 课件把静态分析工具分三类，正好对应三代技术：词汇语法分析、约束分析、基于模型。三代的能力和局限差得很远，考试爱考这个层次。

>

> 第一类，词汇语法分析。原理是通过模式匹配，找出特定函数里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、速度快，但误报率比较高，很难查出复杂的安全问题。为什么误报高？它不知道上下文。strcpy 的源字符串如果是常量，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。原理是更好地分析函数的安全性，靠对解析树的分析确定缺陷位置。它比第一代多走了一步：会做数据流分析，追这个变量从哪来、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，而且没做长度检查"。局限也明确：检查的缺陷范围比较有限。它擅长某一类属性，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。代表工具是
> BOON、CQUAL、Splint、UNO、ESC/JAVA。

>

> 第三类，基于模型。原理是构建程序的有限状态模型，用形式化方法表示要检查的安全属性，再设计算法去检测模型里的缺陷。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，用算法搜。能力强，能挖到时序类的深层问题，代价是建模成本高、状态爆炸。代表工具是
> SLAM、BLAST、MOPS、ESP、FindBugs。

>

> 三代不是替代，是分工。第一代适合快速筛大量代码，第二代适合精确查某几类缺陷，第三代适合验证关键模块的严格属性。实际项目常常三代混用，再加一层人工确认。

>

> 最后一句：静态分析的输出必须人工过一遍。报告里十条问题，可能七条是误报。但剩下三条，往往是你写用例一辈子也碰不到的。

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"所以宁可多报"——这是这类工具的设计取舍：用高误报换低漏报，让人去筛。

- 收口：评估静态分析工具，不能只看报了多少条，要看准确率和能不能定位到行。

> **易错点 / 考点：**三类的名称、原理、代表工具是标准考点。ITS4、Flawfinder、RATS、Pscan 属第一代，FindBugs 属基于模型，这一点最容易记错。判断"静态分析误报率低"——错，第一代误报很高。简答要写出"静态分析不能替代动态测试，两者互补"。

> **过渡：**分类讲完，往下看一份工程里真会用的工具清单。

【7.101】常见的安全漏洞检测工具（3 分钟）

> 十一个工具名摊在一行，硬背下来没用，考完就忘。按用途分成四组，四个词就能拎住一组。

>

> 先看代码这一组，四个。CppCheck 管 C 和 C++，FindBugs 读 Java 字节码。这两个都开源，都不运行程序，只读代码。C++Test 和 Jtest 是 Parasoft 的商业套件，除了静态检查还能生成单元测试。Klocwork Insight 偏大型 C/C++ 工程的深度分析。选型只看两条：你用什么语言，要不要接进 CI。

>

> 第二组是进攻方的工具，Metasploit 和 BackTrack5。Metasploit 是漏洞利用框架，把找漏洞、挑载荷、建会话串成一条线。BackTrack5 是早期的渗透发行版，后来演成 Kali。这一组只在靶场里碰，别的场合不碰。

>

> 第三组做 Web：Burp Suite、Nikto Web Scanner、W3af、ZAP。Burp 是代理型的手工工作台，Nikto 是

> 老牌的 Web 服务器扫描器。W3af 自称攻击与审计框架，ZAP 是 OWASP 的旗舰开源项目。这四个是主力，后面两页细讲 Burp 和 W3af。

>

> 第四组是企业平台，Fortify SSC。它不是单机工具，是软件安全中心。扫描结果汇总、去重、定级、跟踪修复，全在这一层。它通常和 CI/CD 接在一起，管的是整个组织的漏洞。

>

> 清单后面跟了一句"详见我之前写的文章"。说明这是作者自己趟出来的，不是抄名录。

>

> 三句判断收尾。工具是分层的，代码层、应用层、平台层各管一段。开源灵活、能改、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。最后一句最要紧：工具负责发现，人负责判断。同一份报告，老手挑得出哪三条是真的，新手只会转发给开发。

提问：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"这类问题。

- 答"没必要，重复了"——不成立，静态与动态的盲区不重合。

- 答"看预算"——现实考量。即便不买商业工具，关键接口也得手工测。

- 收口：静态工具说"这里可能有问题"，动态测试说"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，这里最容易混的是 **FindBugs 对 Java、CppCheck 对 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：**清单里最值钱的是 Burp 和 W3af。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）

> 知道有 SQL 注入，然后呢？下一步不是翻书，是找工具。课件给了一张对照表，左边十种漏洞，右边一串工具名。

>

> 先说怎么读这张表。它不是一对一映射，是多对多。同一个工具能测好几种漏洞，同一种漏洞也要几个工具配着用。所以别死记"哪个对哪个"，要记"这一类问题用什么手段"。

>

> 先读漏洞那一列。第一项注入式攻击，第二项跨站脚本 XSS。第三项无效的认证及会话管理，第四项对不安全对象的直接引用。第五项伪造的跨站请求 CSRF，第六项安全设置错误。第七项加密存储的不安全，第八项不限制访问者的 URL。第九项传输层保护不足，第十项未经验证的重新指向与转发。

>

> 这十项里七项前面都讲过。注入、XSS、会话、对象引用、CSRF、URL 限制、未验证跳转，全是老面孔。剩下三项是新的：配置错误、加密存储、传输层。它们的共同点很清楚，不在业务代码里，在配置和部署里。所以代码审计做得再认真，系统照样可能出事。

>

> 工具那一列按手段归一下。改请求的三个。HackBar 改地址栏，Tamper Data 和 Seride 拦下来改。Burp 全能拦截。扫漏洞的四个：ZAP、Nikto 和 Wikto、Watobo、MatriXay。专测 SQL 注入的是 SQL Inject Me。查 TLS 配置的是 Calomel，查证书和站点信息的是 Watcher。

>

> 用法就一句：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，最费时间也最没产出。

>

> 还有一句得说清。工具只能覆盖已知模式。这十项里最容易被漏的是配置错误和加密存储，因为要看配置、看数据，不是发个请求就能看出来。

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"——有工具能测，但要专门扫 TLS，普通 Web 扫描器不管。

- 收口：所以能力结构是三层：工具铺面、人工看配置、人工想业务。缺一层就有盲区。

> **易错点 / 考点：** ① 十大漏洞的清单常考多选，"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——答题写"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：** 工具名考试不要求背全，但有两个必须认得——Burp Suite 和 W3af，它们代表两条不同的技术路线。

【7.103】Burp Suite- web security testing（3 分钟）

> 很多人第一次打开 Burp，会以为它是个扫描器。不是。它的核心是中间人：能看、能改、能重放。

>

> 课件列了七个组件，每个都对应一个测试动作。

>

> 第一个，Intercepting Proxy，拦截代理。浏览器和目标应用之间的每个请求，它都能停下来。你能看、能改、能放行。为什么这一步关键？浏览器替你加了很多限制：隐藏字段、禁用的按钮、前端校验。代理一开，这些全不算数。安全测试要的就是发出正常用户发不出的请求。

>

> 第二个，Spider，爬虫。把链接、表单、参数摸出来，产出一张攻击面清单。

>

> 第三个，Scanner，扫描器。自动跑一批漏洞类型，给你一份待确认清单。

>

> 第四个，Intruder。它是可配置的批量重放：标一个变量，灌一本字典，看响应差异。测越权、测爆破、测

> 参数枚举，都靠它。

>

> 第五个，Repeater。改一个请求，发一次，看一眼。这是用得最多的一个。你怀疑某个参数有注入，就在这里反复试。

>

> 第六个，Sequencer，测会话令牌的随机性。这一条直接接上前面讲的会话劫持。令牌能被猜出来，HttpOnly 设得再好也没用。

>

> 第七个，Extensibility。能自己写插件。业务逻辑漏洞千变万化，通用规则接不住，只能自己写判断。

>

> 七个收成一句：Burp 把"我觉得这里有洞"变成"我发了这个请求，它回了这个响应"。

> **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，代理夹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。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 只是里面一个组件；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：**组件清楚了，看它按什么顺序用。

【7.104】Burp Suite- web security testing（2 分钟）

> 装上 Burp，第一件事不是点扫描。先把代理配好，把业务走一遍。

>

> 第一步，配代理。浏览器流量过 Burp，装它的 CA 证书，HTTPS 才解得开。前提说清楚：只在自己的靶机或授权范围里做。代理就是中间人，拿它看别人的流量是另一回事。

>

> 第二步，开拦截，把业务走一遍。注册、登录、下单、查询、退出，每个动作都抓下来。这一步不找漏洞，是画地图：有哪些接口，各自带哪些参数，哪些是 ID，哪些是金额，哪些是状态。

>

> 第三步，丢给 Scanner 铺面。跑完得到一份按风险排序的清单。清单只是线索。

>

> 第四步，Repeater 逐条验。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，Intruder 批量验。一百个用户 ID 里哪些能越权，标好变量灌字典，看返回长度和状态码的差异。

>

> 第六步，整理证据。请求原文、响应片段、复现步骤、影响评估。开发愿意动手修，靠的是这个。

>

> 第三步和第四步的关系要单独说。很多人跑完扫描器就交报告，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。这个分工错一次，工具在团队里就没威信了。

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扫得到。

- 答"顺序反了会漏"——也对，多步交互才出现的页面，爬虫爬不到。

- 收口：所以人给工具画地图，工具替人跑量。

易错点 / 考点： ① 常考"Burp 的典型使用流程"，顺序要是"代理→走业务/爬取→扫描→重放验证→整理证据"；② 易错点：把"扫描结果"直接当"漏洞清单"；③ 常考判断"Burp 能解密 HTTPS，所以能测任何站点"——技术上能，**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 走另一条路——把攻击和审计做成框架。

【7.105】W3af（3 分钟）

W3af 的全称考试爱考，四个英文词。Web Application Attack and Audit Framework。翻成中文是 Web 应用攻击与审计框架。

>

它和 Burp 的路线不同。Burp 给你一套手工工具，W3af 给你一条能编排的流水线，用插件把扫描、审计、利用串成一串。

>

先看它的模糊测试引擎，能注入七个位置。这七个位置就是 Web 输入点的完整清单。前四个是 Query string 和 POST-data。还有 Headers 和 Cookie values。第五个是 Multipart 表单里的文件。第六、第七是 URL filename、URL path。

>

为什么要记全？一个位置只要进了服务端还被用到，它就是潜在的注入点。学生最容易只盯查询串和表单。Header 呢？X-Forwarded-For、Referer、User-Agent 被写进日志。这样的代码一抓一把，也有直接拼进 SQL 的。Cookie 呢？很多系统拿 Cookie 字段直接判权限。文件名和路径更少人看，上传后的重命名、下载接口的路径拼接，都落在这两处。

>

第二块是代理服务器。它自带 Web 和代理服务器，好嵌进你的代码。翻成白话：它不只能当界面用，还能当库。

>

第三块是 Fast HTTP Client。前三项是极快地发特制请求、支持代理、Basic 和 Digest 认证。中间几项管请

> 求构造：伪造 User-Agent、自定义请求头、处理 Cookie。最后还有响应处理、DNS 缓存、多部分文件上传。

>

> 别当参数表背，每行都是一个测试动作。伪造 UA 是为了测移动端和爬虫的不同逻辑。自定义头是为了测 Header 注入和绕过。DNS 缓存是为了压测时不打爆 DNS。多部分上传是为了测文件上传漏洞。

> **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，下载接口和静态资源路由最常见。

- 收口：判断法很简单——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"的接口，再对照这七项。

> **易错点 / 考点：**① W3af 全称常考；② 七个注入点是很好的多选题素材，"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：**安全性测试这一节讲完，收一页把整节的方法盘一遍。

【7.106】本讲小结（3 分钟）

> 安全性测试的方法，教材列了十项。十项摊开像清单，按"从设计到运行"排一遍就成了台阶。

>

> 基于模型的安全功能测试：先给系统建一个行为模型，再验安全属性在这个模型上成不成立。它回答的是设计层面允不允许坏事发生。

>

> 基于故障注入：人为造故障，断网、超时、返回错误、篡改数据，看系统会不会因此失守。它打的是异常路径，大量漏洞只在这一路上露头。

>

> 模糊测试：拿变异出来的畸形输入轰接口，看它崩不崩、会不会执行、会不会泄数据。你不用先知道漏洞长什么样，这是它最大的价值。

>

> 基于白盒：拿代码做数据流和污点分析，追用户输入有没有不加处理就流到危险函数。

>

> 威胁模型与攻击树：这不是执行，是设计。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。

>

> 形式化方法：用数学描述安全属性再证明。贵，只用在最关键的模块上。

>

> 语法测试：打解析器，JSON、XML、协议报文、文件格式，用合法但异常的结构去试。前面讲的 XML 注入就挨着这一类。

>

> 基于属性的测试：先声明必须成立的性质。比如金额不能为负、一个人只能领一张券。再自动造数据去推翻它。

>

> 动态污点分析：给外部输入打标记，跟踪它流到哪，看有没有流到敏感操作。工业界查注入，这是最有效的手段之一。

>

> 基于风险的安全测试：按风险分资源，高风险的模块多测、深测。

>

> 十项不必全用，得说得出各自解决什么问题。考场上给你一个场景，你要选得出两三种组合。

> **提问：**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：一个用畸形输入轰，一个把力气压在高风险接口上。

- 答"渗透测试 + 故障注入"——也行，但要说明"渗透"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，还是一个能登录的账号。

> **易错点 / 考点：** ① 十项名称常考多选与填空，"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"当成测试执行技术——它是**设计与分析**活动；③ 简答题"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，最后留两页推荐。推荐的不是工具，是接下来往哪儿学。

【7.107】特别推荐（2 分钟）

> 这一节收尾，有人会问：接下来自己练什么。课件把推荐留在这儿，我按先打底、再上手、最后看局面的顺序说。

>

> 先打底。安全测试的知识是跨层的。你得懂 HTTP 报文怎么长，懂浏览器的同源策略，懂数据库的权限模型，懂操作系统的权限。这些不补，工具会用，但不知道为什么。最低的底子是四样。HTTP 报文结构、Cookie 与 Session 机制、SQL 基础、Linux 权限。

>

> 再上手。这一块看书没用，必须有靶场。课件推荐的资源，作用都是让你在合法可控的环境里把一条攻击链走完。走完一遍，你对"输入校验""输出编码"这几个词的理解会变，因为你见过不加会怎样。

>

> 最后看局面。看每年发布的漏洞榜单，看真实事件的复盘。看几年会发现一个规律：排在前面的类型十几年没怎么变，注入、XSS、失效的访问控制、配置错误。技术在换，人在同一个地方反复摔。

>

> 课程要求补一句。所有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。能力越强，边界越要清楚，一次"试试看"可能就是别人的损失。

>

> 最后一条学习法：每学一个手法，逼自己写三句。原理一句、防御一句、怎么测一句。三句写不出来，就是还没学会。

> **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，经验可复用"——是一面，检查单确实能沉淀下来反复用。

- 答"坏消息，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管用。

> **易错点 / 考点：**这一页不直接出题，常作为**课程设计选题**的引导。要区分"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。

> **【思政落点 11】**：靶场和别人的系统之间，只隔着一个授权。同一条攻击链，放在自己的靶机上是练习，放到没打招呼的机器上就是违法。这门课让你们练手法，也请顺手练一个习惯：动手前先问这台机器是谁的、我有没有书面许可。守住这条线，能力才叫本事；守不住，能力就是给别人添的风险。

> **过渡：**推荐还有第二页，方向从"学什么"转到"怎么把安全放进工程里"。

【7.108】特别推荐 2（2 分钟）

> 工具装了一堆，事故照样出。问题多半不在工具上，在流程上。这一条线走三层，从个人一直到组织。

>

> 第一层，规范与标准。安全测试靠清单，不靠灵感。漏洞榜单告诉你防什么，安全编码规范告诉你怎么写，测试指南告诉你怎么验。三样东西合起来干一件事：把专家的判断变成普通人照着能做的步骤。

>

> 第二层，工具链集成。安全要真被执行，就得嵌进流水线。提交代码跑静态扫描，构建镜像查依赖漏洞，部署前跑一次动态扫描，上线后持续监控。这套叫 DevSecOps。它比上线前集中测一遍更省，因为问题在最便宜的地方就被拦下了。判断标准很朴素：需要人专门去点一下的检查，迟早被跳过。

>

> 第三层，组织与责任。谁对安全负责？不是安全部门。是每个提交代码的人、每个设计架构的人、每个做发布决定的人。安全团队给工具、定标准、做应急，不替所有人做检查。

>

> 给一个马上能用的建议：课程设计里加一份安全检查单，十到十五条。每条写清查什么、怎么查、不合格怎么改。这份单子比任何工具都值钱，它是你自己的知识沉淀，下次做项目直接拿来用。

>

> 最后一层窗户纸：安全不是一门课，是习惯。每写一个接口，问一句——别人塞了个奇怪的参数进来，会怎样。

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，不是用自觉保证，跟第 4 章讲的流程规范是一回事。

易错点 / 考点：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"。另外区分"规范"（写什么）与"工具"（怎么查），两者互补。

> **过渡：**安全性测试到此结束。下面进 7.3，一个听着简单、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）

> 同一个软件，换台机器就打不开。这种事你们遇到过吗？这就是兼容性测试要管的事。

>

> 它要回答一个问题：同一套代码，换一个环境还对不对。环境有四个维度。硬件：机型、外设、配置。软件系统之间：操作系统、中间件、数据库、第三方组件，还有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：浏览器加系统加分辨率加语言加网络。

>

> 为什么容易出事？因为开发环境只有一套，用户环境是发散的。你机器上跑得顺，用户那边可能是三年前的安卓机。可能用的是 IE 内核的浏览器，可能电脑没有管理员权限。还可能装了个改了默认行为的输入法。

>

> 代价也很特别。它不影响功能正确性，只让一部分用户用不了。报告里写"某机型闪退"，看着像小问题，对那批用户就是百分之百不可用。而且它难复现，开发和测试经常因为"我这儿没问题"吵起来。

>

> 这一节怎么走：先看最典型的两种现象，升级失败和升级之后的不兼容；再看软硬件、系统之间、数据三条主线；最后讲环境组合怎么用组合测试压用例。

>

> 带着一个问题听：你们做项目时，"测哪些环境"是拍脑袋定的，还是有依据的？

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"——正是这种想法让它被降级，可对那批用户是 100%。

- 答"因为难复现"——对，复现不了就容易被判"无法重现"关掉。

- 答"因为不影响主流程"——要小心这句，支付页面在某个浏览器上白屏，主流程就断了。

- 收口：评价它要看受影响用户比例乘上这个功能的业务重要性，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：先看现实里最常见的两种现象：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）

> 升级最怕的不是彻底失败，是升到一半。

>

> 先把两种形态分开。第一种，升级失败。安装包装不上，数据库脚本报错，配置文件格式不认，依赖版本对不上，或者升到一半断了。最难的是最后那种：一部分模块升上去了，一部分没升。系统还能启动，行为已经不一致了。

>

> 第二种，升级后的兼容性问题。升级成功了，但和别人对不上。新客户端连不上老服务端，新版本写的数据老版本读不了。依赖的中间件老服务器上没有，第三方接口的协议也变了。

>

> 两种现象的根子是一个。升级不是换个文件，它同时动了四样东西：代码、数据格式、接口协议、运行环境。哪一样没对齐，都出问题。

>

> 所以升级测试的清单就是四问。旧数据新版本能不能正确读？新写的数据老版本还能不能用？接口协议变了没有，调用方都同步了吗？依赖的组件在目标环境里有吗？

>

> 这四问最好画成一张表，每次发版照着勾。半新半旧的系统最坑，它既不能当旧的测，也不能按新的测。

>

> 再补一条工程做法：升级得可回滚。上线前先验一遍，回滚脚本跑得通，回滚后数据不丢。真出问题时，业务等不了你排查，先退回去止损才是对的顺序。

>

> 这类事故里，硬技术难题很少，多半是清单没列全。

> **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、要事务化：要么全成功，要么能干净退回，中间状态必须能查出来。

> **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"，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。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）

> 很多人以为硬件兼容就是驱动的事。不止。从上往下数，有五层。

>

> 第一层，机型和配置。同一个 App，换 ARM 或 x86，换内存大小，换屏幕分辨率，换传感器，表现就可能不同。低内存机上后台被杀，小屏上按钮被挤掉，没有某个传感器的机型功能直接崩。这些都是兼容问题。

>

> 第二层，外设和接口。打印机、扫码枪、读卡器、摄像头、USB 设备、串口设备。这层问题特别多，因为外设的驱动行为差异大，而且常常不在开发者的环境里。

>

> 第三层，驱动和固件。显卡驱动版本不同，渲染结果就不同。固件版本不同，同一条指令的行为就不同。典型症状是某批机器上画面撕裂、花屏、闪退。

>

> 第四层，资源和性能边界。磁盘满了、内存紧了、电量低了、温度高了降频。这些状态也会触发兼容问题，因为它们改了软件的运行条件。

>

> 第五层，虚拟化和云。虚拟机、容器、云主机的硬件是抽象的。依赖特定硬件特性的功能，GPU 加速、硬件加密、特定指令集，可能直接不可用。

>

> 怎么测？三条务实的。按市场占有率选机型，别按你手上有哪些机器选。抓关键路径，安装、启动、登录、核心业务、退出，每条在低端机和高端机上各跑一遍。把机型矩阵沉淀下来，每次发版复用，别每次重新开会。

>

> 一句话：关键不是测多少台，是测对哪几台。

> **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乘业务关键度，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，"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）

> 一张三层堆栈的图，右边标着两个 upgrade 箭头。从上往下是应用软件、中间件和操作系统、第三方软件。层与层之间写着兼容。

>

> 版本对给了两组。一组是 Red Hat Enterprise Linux 7.1 和 6.6。另一组是 MySQL Enterprise Edition 5.7 和 5.6。

>

> 看这张图先看出一件事：兼容不是一个点，是一条链。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。链上任何一环换了版本，整条链的兼容关系都要重验。

>

> 第二件事：升级方向不对称。图上是往上走，6.6 到 7.1，5.6 到 5.7。往上走要回答两个问题。新版本还认不认老的配置和数据？我的应用在新版本上还跑不跑？这两个答案经常不一样。数据库升上去了，应用用的驱动还是老的，照样连不上。

>

> 第三件事：兼容有方向。新版本能处理老版本的东西，叫向后兼容。老版本能处理新版本的东西，叫向前兼容。工程上通常只保证向后兼容。所以升级前要确认：回滚窗口里，老版本还读不读得动新数据。

>

> 具体测四条。安装与启动，在目标系统版本上装得上吗，依赖齐吗。接口与协议，中间件和驱动的版本匹配吗。配置兼容，老配置文件新版本还认吗。性能与行为，默认参数变了没有，连接池、字符集、超时时间。

>

> 最后一条最容易被忽略：默认值变了也是不兼容。代码一行没改，行为变了，这种问题最难查。

> **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，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，不能只验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。

> **过渡：**服务端是往上叠的兼容，客户端是多版本并存的兼容。这两种问题性质完全不同。

【7.113】软件系统之间的兼容性 -2（2 分钟）

App 发了新版，用户三个月不升。服务端照样得认他。图上画了 Ver 2.1、2.2、3.0 三个版本。每个都要和中间件、操作系统兼容。也要和第三方软件兼容，彼此之间还得兼容。

>

这里的兼容有两层意思。

>

第一层，一个服务端要同时伺候多个客户端版本。这是常态。用户不会因为你发了新版就立刻升，应用商店里留着历史版本，企业内网锁版本，老设备装不上新版。所以服务端得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，不同版本客户端之间的数据要通。A 用 2.1 发一条消息，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？这就是跨版本的数据兼容，也是下一页那条数据兼容的一个具体场景。

>

由此有两个工程要求。

>

接口设计要能演进。新增字段必须可选，不能因为多了字段就报错。删字段先停用，再下线。枚举值要允许未知值存在。这三条合起来就是健壮性原则：发送要保守，接收要宽容。

>

第二个要求，版本支持策略要写清楚。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？这几个问题没有答案，测试就定不了组合。

>

> 收一句：服务端的兼容是往上长，客户端的兼容是横向铺。前者防升级踩坑，后者防版本碎片。

> **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：三条合起来是结构可扩展、缺省有定义、未知不致命。

> **易错点 / 考点：**① 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：**版本兼容是代码之间的事，数据兼容是数据之间的事。后者更容易被忽略，也更难修。

【7.114】最常见的数据兼容性问题？（2 分钟）

> 老版本导出的表格，新版本一打开，中文全是问号。这就是数据兼容问题最常见的脸。

>

> 课件问的是：最常见的数据兼容性问题是什么。它给了三个现象。

>

> 第一个，文件打不开。老版本生成的文件新版打不开，或者别的软件生成的本软件打不开。原因通常是格式变了、编码变了，或者新版本把校验收严了。严格化本身也是一种不兼容。

>

> 第二个，数据显示不对。文件打开了，内容错位：中文乱码，日期差一天，金额小数点挪位，列表顺序变了，字段串了。这一类最阴，因为系统不报错。用户以为是对的，直到某天核数对不上。静默的错误比崩溃危险得多。

>

> 第三个，系统崩溃。读到预期之外的结构直接崩：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够，而这类数据多半来自旧版本或别的系统。

>

> 三个现象背后是一句判断。数据兼容测试的核心不是能不能读，是读出来的东西对不对。读错了会不会崩，同样要问。

>

> 所以测的时候问四句：能读吗？读全了吗？读对了吗？读错的时候系统怎么反应？最后一句最常被漏。正确的做法是明确报错并拒绝处理，最糟的是带着错数据继续往下算。

>

> 还有一个容易忘的方向：数据会往回走。用户升级后可能回滚，可能把新文件发给老版本的人，可能用新版导出再导进老系统。所以不能只测新读旧，还得测旧读新。

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，业务决策就跟着错。

- 答"因为难追溯"——也是，等发现时已经攒了一堆错数据，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"有没有抛异常"。

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）

代码可以回滚，数据不行。这一节最重的一句话，先放在这儿。

>

> 同一个系统内部的数据兼容，图上有两个方向。

>

> 第一个方向，客户端和服务端之间的数据文件。图上写的是上传旧版、下载新版数据文件。客户端升级了，它要处理的文件可能是旧客户端传的。反过来，服务端生成的新格式文件，还支持的老客户端也得能打开。这一层的关键是格式版本号。文件里带一个版本标记，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，软件和数据库之间。图上两条：新版本软件存取历史数据，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。难度完全不同。新读旧是设计时就得保证的，向后兼容。旧读新通常做不到，所以工程上只保证回滚窗口内的兼容。

>

> 贯穿两者的动作是 DB schema 升级。加表、加字段、改类型、加索引，最容易出事。四条经验规则：加字段给默认值或允许为空；不改字段类型和含义；不删字段，先标废弃；大批量变更要可重入、可中断、可续跑。

>

> 为什么这页值得慢讲？因为它对应"升级后数据全乱"这类事故。可怕的地方是不可逆。代码能回滚，数据写错了要一条一条修。

>

> 所以结论说重一点：动数据结构之前，先想清楚怎么退回去。这不是保守，是对已经在库里那批数据负责。

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，数据库视图或接口层做适配。

- 收口：保证不了"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 12】：库里那批数据不是测试数据。它们是用户几年攒下来的订单、病历、成绩、流水，没有"重跑一遍"的机会。技术上，加一个字段只是几秒钟；对数据来说，可能是不可逆的改写。所以定三条规矩：改结构前先想回滚，跑脚本前先做备份，批量变更要能中断能续跑。这种动手前先想后果的习惯，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，难度还要再上一格。

【7.116】不同系统之间的数据兼容（2 分钟）

从 Word 里复制一段，粘进网页编辑器，格式全散了。这事天天发生，很少有人把它当兼容性缺陷去测。

>

不同系统之间的数据兼容，图上列了三组来源。微软的 Word Editor，跑在 Windows 11 上。苹果的 Word Editor，跑在 Mac OS X 上。IBM Lotus 的 Spreadsheet，跑在 Windows 11 上。右边五种互通方式：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。

>

这五种就是五条数据进本系统的通道。每条通道都是测试对象。很多人只测文件导入导出，另外四条不测，真实使用时就出事。

>

网络导入导出，走接口或协议。编码、时区、精度、字段长度都得对。

>

文件导入导出，盯格式版本、字符集、换行符、BOM、分隔符、日期格式。

>

文件加载保存，直接打开别人的格式，或者在别人的软件里打开你的文件。这条最考验解析器的健壮性。

>

> 剪切复制粘贴，最容易被漏的一条。剪贴板里同一份内容可能同时有 HTML、RTF、纯文本三个版本。系统取哪一个、怎么清洗，决定会不会乱码，也决定会不会把脚本一起粘进来。这又回到 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。

>

> 还要引一个判断：跨系统兼容的目标不是完美还原，是关键信息不丢、语义不错。字体行距这类东西跨系统几乎不可能一致。力气要放在金额、日期、编号、状态这些不能错的地方。

>

> 测法按格式乘通道做矩阵，每种格式过每一条通道。

> **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，"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）

> 格式是越多越好吗？做过导入导出的人都明白，多支持一种就多一堆坑。图上四个动作叠在一起：导入下列格式文件、读取或导出下列格式文件、读取已有文件、另存为。最后一个还标了 better。

>

> 拆成四种能力看。

>

> 导入：把外部格式读进来，转成内部模型。要点是畸形输入。字段缺失、顺序颠倒、多余字段、超长字段都要试。空文件、超大文件、编码不是 UTF-8、带特殊字符和公式，也都要试。导入的健壮性，决定系统会不会被一个脏文件搞崩。

>

> 导出：把数据写成外部格式给别人。要点是对方能不能正确打开。这得真拿目标软件去开，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判定。

>

> 读取已有文件：关键词是已有。历史文件、旧版本文件、别人给的文件。它们往往不合你现在的假设：没版本号、字段缺失、用了废弃的取值。要点是向后兼容的边界。最老支持到哪一版？超出范围是明确报错，还是静默出错？

>

> 另存为：课件标了 better，意思是推荐支持。为什么推荐？它是用户的逃生通道。新版打不开或者写坏了，还能另存成通用格式把数据保下来。不给导出通道的系统，等于把用户的数据锁死。

>

> 四种能力乘上格式，就是一个矩阵。行是格式，列是四个动作，每格都要有用例。用例数量涨得很快，这正是下一页要用组合测试压下来的原因。

>

> 最后一句：格式兼容有成本。支持得越多，维护越贵，安全风险越大，解析器本来就是最容易被攻击的组件。所以产品要写明支持哪些格式、支持到什么程度，而不是尽量都支持。

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，也最常被省略。

- 收口：正确姿势是闭环验证：导出、用目标软件打开、比对关键字段、再导回来比一遍。

> **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 "另存为"的价值要答得到——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，用例就炸了。怎么在有限时间里盖住尽可能多的组合？这就是组合测试要解的问题。

【7.118】软件环境(组合)兼容性测试（3 分钟）

> 四个系统乘五个浏览器乘四种分辨率，你先算算多少种。

>

> 操作系统 4 种，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。全测一遍，不现实。

>

> 课件给两个方法：Pairwise 组合测试，正交实验法。

>

> Pairwise 的思路是：绝大多数缺陷由两个因素的交互触发。三个以上同时交互才出的缺陷，比例很低。所以只要保证任意两个因素的每一对取值至少出现一次，就能用很少的用例盖住大部分风险。720 种压下来，通常二三十条就够。

>

> 正交实验法思路相近，但它用正交表保证各因素各水平的均衡搭配，适合因素和水平都规整的场景。区别是常考点。pairwise 管两两覆盖，正交表管均衡。正交表还能做极差分析，看出哪个因素影响最大。

>

> 课件还有一句关键的：根据市场调查数据，设计组合矩阵表。意思是组合不能凭感觉选，得先知道用户真在用什么。八成用户在 Windows 上用 Chrome，那一格必须优先、必须深测。只占千分之三的组合，pairwise 盖到就行。

>

> 完整的方法链是五步。先列因素和水平，再查市场占用数据。用 pairwise 或正交表生成组合，按占用率分配测试深度。最后把矩阵表沉淀下来。

>

> 最后一句要说清：组合测试能压用例，是因为它假设缺陷主要来自两两交互。有些缺陷要三个因素同时满足才出现，比如特定浏览器加特定语言加特定时区。高风险的地方还是得补针对性用例，不能全指望组合。

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，不是碰运气。

- 收口：说清它的假设，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：组合测试压的是因素之间的关系。还有一种矩阵是系统之间的关系，收个尾。

【7.119】小结（3 分钟）

向后兼容，向前兼容。这两个词每年都有人答反。

>

> 课件写得很干脆：向后兼容是能使用之前版本的数据，向前兼容是能使用未来版本的数据。记法看方向。新版本读旧数据，叫向后，因为它往后看。旧版本读新数据，叫向前，因为它往前看。

>

> 实务结论要背下来：向后兼容必须保证，向前兼容只在不破坏结构的前提下尽量做。新版本知道你过去写过

> 什么，旧版本猜不到你未来会写什么。

>

> 第二组是动作：升级、迁移、备份、恢复、导入导出。五个动作都是数据在动，也就是数据兼容的高发区。它们的关系不一样：升级和迁移是主动改，备份和恢复是保险，导入导出是跟外界交换。测的时候要两两组合，比如升级失败后恢复备份、导出后导入到新环境。真实事故就长这个样子。

>

> 第三组是范围：硬件兼容、系统之间兼容、数据之间兼容。三条对应三个问题层次。硬件层问跑不跑得起来，系统层问能不能配合，数据层问读出来对不对。

>

> 收一句：兼容性测试是在环境发散和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。

>

> 还有一个容易被跳过的产出：兼容性结论要写成支持矩阵。明确写支持哪些环境、不支持哪些、不支持时什么表现。这份矩阵既是测试结论，也是对用户的承诺。

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儿。

- 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。三列齐了才算写完。

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 7.4 可靠性测试，这一节有一半内容是近几年才流行起来的。

【7.120】7.4 可靠性测试（2 分钟）

日活百万的系统，大促前怎么证明它扛得住？这一节回答这个问题。

>

> 课件在经典内容之外加了一句：增加了现在流行的全链路压测和混沌工程。还给了定位：全链路压测属于稳定性工程，但也可以算性能测试的一类。

>

> 这句信息量不小，拆三层。可靠性的经典内容是成熟性、容错性、易恢复性，问的是系统能不能稳定运行。近几年业界把它扩成稳定性工程，问的是真实流量、真实故障下还扛不扛得住。全链路压测和混沌工程是这套实践的两个抓手。一个把真实规模的流量灌进去，一个把真实世界的故障注进去。

>

> 这两个为什么必要？传统性能测试有个致命缺陷：它测的是测试环境在模拟流量下表现如何。生产环境的数据量、调用链深度、依赖数量、网络拓扑，跟测试环境不是一回事。测试环境压出一千 TPS 很高兴，上线三百 TPS 就雪崩。中间差的就是全链路和真实数据。

>

> 这一节的顺序是：先讲可靠性的三个度量维度，再讲容错性测试，然后重点讲全链路压测。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。

>

> 带着一个问题听：给一个日活百万的系统做一次"扛不扛得住大促"的验证，你会怎么设计？讲完这一节，你要答得出来。

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，测试环境常把下游 mock 掉，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准："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）

不出故障，就叫可靠吗？这个直觉一半对，一半很危险。

>

课件给的定义是：在规定的一段时间和条件下，软件把性能水平维持住的一组属性。定义里藏着两个限定词，"规定的一段时间"和"规定的条件"。为什么非要有这两句？因为可靠性没法脱离时间和环境谈。同一套系统，跑一周没事，跑一年呢？测试环境没事，双十一呢？把这两句删掉，"可靠"就成了空话。

>

这组属性拆成三个子特性度量：成熟性、容错性、易恢复性。三个各管一段。

>

成熟性，说的是系统不出故障的能力。课件给的量法是错误发现率 DDP，公式是测试发现的错误数量，除以已知的全部错误数量。这里方向容易记反：DDP 越小，软件越成熟。小反而好，为什么？因为分母是"已知的全部错误"，里面算进了更早阶段就抓到的那些。测试阶段发现的占比低，说明前面的评审和单元测试把大部分挡住了。

>

容错性，说的是出了故障还能接着提供正确服务。比如下游超时，系统降级返回缓存，而不是直接报错。这一项下面单独一节讲。

>

易恢复性，说的是坏了以后恢复得多快、多完整。课件的测法是：先设法模拟让系统崩溃、失效，再算系统和数据恢复的时间，据此评估。所以它有两个维度，恢复时间，还有恢复的完整度，数据丢没丢、状态错没错。

>

> 三个收到一句：成熟性管少出事，容错性管出事不倒，易恢复性管倒了能起来。落到测试设计上，三种手段不通用。成熟性靠长时间运行加缺陷数据分析。容错性靠故障注入。易恢复性靠破坏再恢复的演练。

提问：DDP 越小越成熟，这个方向和大多数指标相反。你能讲清为什么吗？

预设回答：

- 答"因为它算的是测试阶段发现的占比"——对，分母是已知的全部错误，包含更早阶段拦下的。

- 答"测试阶段发现得少，说明前面拦住了"——准确，这就是"过程成熟"的意思。

- 答"那测试越弱，DDP 不也越小"——这个反问很好，陷阱确实在。前期没发现、测试也没发现，DDP 一样小。所以它得配合逃逸缺陷数一起看。

- 收口：单个指标都能被操作，度量要看一组数，不是一个数。

易错点 / 考点：① DDP 越小越成熟，方向高频错，别记反；② 三个子特性的名称和含义常考连线；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"这一项得先把系统弄坏才测得出来。

【7.122】容错性测试（2 分钟）

有一类测试，天生就是去搞破坏的。

>

> 课件先给了定性：容错测试是一种对抗性的测试过程。对抗性三个字要抓住。它不验证正常功能，它主动制造故障。

>

> 具体怎么做？通过各种手段让软件强制性地发生故障，或者把应用、系统放到模拟的异常条件下，逼它出故障。课件举了两个例子：设备输入输出故障，无效的数据库指针和关键字。

>

> 从这两个例子能看出两类故障源。一类是环境类：I/O 出错、网络断、磁盘满、时钟漂。另一类是数据类：

> 无效指针、无效关键字、脏数据。前者模拟世界不正常，后者模拟数据不正常。两类都得造。只造一类，容错能力只测了一半。

>

> 课件还有一句判断分量很重：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。这是在指方向。

>

> 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，我在受控的实验里制造真实故障，看系统整体怎么反应"。差别落在三点：范围，单点还是系统级；环境，测试环境还是生产；驱动方式，预设故障还是实验驱动。

>

> 但混沌工程不是胆子大。它有三个前提。一是稳态指标要明确，比如成功率、延迟分位数。二是爆炸半径要控，先小范围、随时能中止。三是有监控和回滚预案。三条缺一条，那就不叫混沌工程，那叫闯祸。

>

> 测完怎么判？三条。故障发生时核心功能还在不在，降级有没有生效。恢复之后数据一致不一致，有没有丢单、重复。用户有没有拿到明确反馈，不是白屏，不是无限转圈。

>

> 所以这项测试考的不是"不出故障"，是"出了故障以后系统怎么表现"。

提问：混沌工程为什么强调在生产环境做，只在测试环境不行？

预设回答：

- 答"测试环境没有真实流量"——对，没流量就观察不到影响面。

- 答"生产环境的依赖关系才是真的"——很准，测试环境常缺依赖、走 mock。

- 答"所以风险大，得控爆炸半径"——正因如此才有先小范围、能中止这条规矩。

- 收口：混沌工程不是更大胆的破坏，是更严格的实验设计，风险控制写在方法里。

易错点 / 考点：① 容错性测试等于对抗性测试，这个定性常考；② 故障注入与混沌工程按范围／环境／驱动方式三点对比；③ 别把混沌工程当成随便杀进程，它得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，接下来这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）

> 测试环境里压出来的那个漂亮数字，敢不敢拿去汇报？

>

> 前面 7.1 讲的性能测试，是在测试环境用 JMeter 这类工具压单个服务、单条链路。全链路压测要回答的是另一层问题。

>

> "全链路"这三个字有三层意思。

>

> 第一层，链路要完整。从用户入口压起，CDN、网关、负载均衡，一直压到最底下的数据库、缓存、消息队列。中途每一次 RPC 都在链路上，一个都不许 mock。为什么这么较真？因为瓶颈常常不在你以为的地方。应用层 CPU 才三成，数据库连接池已经满了。

>

> 第二层，流量要真实。请求比例、参数分布、用户的思考时间、数据的倾斜程度，都得像生产。压测不是同一个请求重复一百万次。那样缓存全命中，第二次之后连库都不查，你测的是缓存，不是系统。

>

> 第三层，数据要隔离。这是全链路压测最难的工程问题。压测会产生写操作，写进去的订单、库存、余额会污染真实业务。所以得有影子库、影子表、流量标记、挡板。后面讲 Takin，它把 0 侵入加数据隔离当核心卖点，原因就在这儿。

>

> 那它值钱在哪？大促、秒杀、开学选课、春运抢票，这类场景的特点是短时间、极高并发、全链路承压。测试环境永远模拟不出来，所以它是唯一可信的验证手段。

>

> 这一节后面会一层层展开。为什么要做，流量怎么控，平台怎么搭。数据怎么准备，流量怎么录怎么放。流程怎么组织，开源实现长什么样。顺序别倒。先想清楚为什么，再谈怎么做，否则就成了堆工具。

> **提问：**同一个请求重复一百万次那种压法，结论为什么不可信？

> **预设回答：**

- 答"缓存全命中了"——最直接，第二次起就不查库了。

- 答"没有数据倾斜"——对，真实流量的热点分布很不均匀。

- 答"读写比例不对"——也在点上，纯读和读写混合的瓶颈完全是两件事。

- 收口：所以脚本里得有流量模型，接口比例、参数分布、读写比、思考时间，四样齐了才算模拟。

> **易错点 / 考点：**① 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，关键在链路和数据，不在并发数；③ 简答"与普通性能测试的区别"，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？课件把需求单列了一页。

【7.124】全链路压测的需求（2 分钟）

> 不是所有系统都值得做全链路压测，这句先说在前面。

>

> 因为它太贵。要改中间件、要备影子库、要几拨团队一起上、要在生产环境动手。所以"要不要做"本身就是一个判断考点。

>

> 四个触发条件。

>

> 第一，有明确、可预期的流量峰值。大促、秒杀、抢购、开学选课、节假日出行，曲线是已知的，峰值躲不过去。既然知道要来，就得提前验。

>

> 第二，系统大到单点压测没意义。一条业务链路经过十几个服务、几个中间件、多个数据源。你单独压一个服务，那个数字跟整体表现没关系。瓶颈在链路里，不在服务里。

>

> 第三，容量决策要数据支撑。扩几台机器、数据库要不要分库分表、缓存开多大，靠拍脑袋加去年经验，很容易估错。压测给的是容量水位：当前配置扛多少、卡在哪个节点、扩哪里最划算。

>

> 第四，稳定性有硬要求。金融、医疗、政务、电商交易，一次故障代价太高，验证这笔钱花得值。

>

> 反过来，什么情况可以不做？流量平稳、链路短、能降级、故障代价小，做常规性能测试加监控就够了。做不做全链路压测，不是专业不专业的分界线，是成本和风险配不配的问题。

>

> 还有个现实约束：它要跨团队。业务、测试、运维、DBA、中间件团队都在场。所以通常是专门的稳定性团队牵头，一个测试人员独立干不了。这也是后面要讲压测平台的原因，平台就是把这摊复杂性收进去。

> **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"——基本对，补一句：常规的性能验证和容量评估还是得有。

- 答"要看有没有峰值"——很专业的判断，月末结算、批量导入也可能是尖峰。

- 答"做了更保险"——投入产出不划算，压测成本可能超过系统本身的价值。

- 收口：判断标准是故障代价乘峰值概率，跟压测成本比一比，不是看别人做没做。

> **易错点 / 考点：**① 四个触发条件是常考简答；② 易错点：以为重要的系统就得做，要看链路复杂度和峰值特征；③ 判断"全链路压测只能测试团队独立完成"——错，它靠多团队协作加平台支撑。

> **过渡：**决定要做，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）

压测最怕的一件事，是把生产压出事。所以这里先讲怎么刹车，再讲怎么加力。

>

图里画的是一条完整的下单链路。订单进管道，出 MQ 消息，到预分拣。再出 MQ 消息，到任务分发，最后落到下游系统。下游还分系统、机房、分组。链路下面是线上应用集群，画成一排 Docker，一号到九号。流量来源标着互联网生产流量。链路中段两处标了憋洪和泄洪，下面集群两侧标了阻塞和泄洪。

>

三个机制，一个一个说。

>

第一个，流量标识与路由。压测流量进来的时候必须带标记，一般是 HTTP 头里的一个特殊字段。中间件认出这个标记，把请求路由到影子库、影子表、影子缓存。压测的写操作落在另一套存储上，真实数据不受影响。数据隔离就是这么实现的。

>

第二个，憋洪和泄洪。憋洪是把流量先攒在入口，等所有节点就绪、监控到位，再统一放。为什么非憋不可？因为压测要的是瞬时并发。流量慢慢涨上去，你永远看不到峰值那一刻系统的样子。憋洪还带一个好处：想停就停。流量没放出去，风险还在自己手里。

>

第三个，阻塞和限流。图上标的是阻塞加泄洪。系统水位过了阈值，就主动把流量挡住，别把自己压垮。这一条在生产环境里是硬要求。你可以在生产上压测，但不能把生产压死。

>

这里回答的是一个矛盾：全链路压测必须在生产环境做，不然不真实；生产环境又不能出事故。解法就是这三样。标记隔离数据，憋洪造出峰值，限流兜住底线。

>

一句话，压测的第一原则不是压得多狠，是随时能停。

提问：憋洪这个动作，对压测结论的有效性为什么是必需的？

> **预设回答：**

- 答"要制造瞬时并发"——正是，慢慢涨上去测不到峰值表现。

- 答"得等节点都就绪"——对，监控、影子库、挡板得先确认到位。

- 答"出问题能马上取消"——很关键，流量攒在入口，风险可控。

- 收口：所以憋洪一半是技术手段，一半是风险控制手段，写方案时得写进流程。

> **易错点 / 考点：** ① 憋洪／泄洪、阻塞／限流的含义常考名词解释；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 简答"怎么避免污染生产数据"，要提流量标识加影子库表加挡板。

> **过渡：** 流量怎么控清楚了，接下来看支撑这一切的压测平台是怎么搭起来的。

【7.126】压测平台原理示意图（3 分钟）

> 一次压测，几万台压力机同时发请求。这些机器从哪来，往哪发，谁喊开始？

>

> 三个问题，对着图上的三块看。

>

> 谁发压？中间是流量引擎集群，下面分两大类节点。一类是站外分区域节点，挂的是联通、电信、移动，还有第三方 CDN 节点。另一类是站内各机房节点，IDC 机房 A、B、C、D，每个机房里放压力机引擎。

>

> 为什么非要在站外发？因为真实用户就是从这些入口进来的。只从内网发，DNS、CDN、公网带宽这几段你就测不到。分运营商发还有个用处，某一路带宽被打满这类问题，只有分开压才暴露得出来。

>

> 为什么机房也要多点？两个原因。压力机自己的网络出口有限，一个机房发不出那么大的量。还有，真实流量不会只打一个机房，跨机房调用这一层得模拟。

>

> 发到哪？右边是业务系统，首页、购物车、结算页这些。它有两个入口：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。两个入口对应两个目的，别混着用。

>

> 怎么控？左边是压测平台，管三样东西：脚本、数据、流量控制台。脚本定发什么请求。数据定用什么参数发，压测用的用户、商品 ID、优惠券都在里面。流量控制台定发多少、什么时候发、什么时候停。

>

> 整套架构的思路就一句：把发压能力和业务系统拆开，用平台统一调度。压力机可以横向加，脚本和数据可以复用，流量可控。这样一次压测的组织成本才降得下来。否则每次都临时找机器、临时写脚本。

>

> 还有一点。图里压力机引擎画了一排，它们不是各干各的。得由统一控制器一起启动，才形成瞬时并发。

> **提问：**压力机为什么要摊到多个机房、多个运营商节点，集中在一个机房不行？

> **预设回答：**

- 答"单机房出口带宽不够"——最直接，压力机自己先成瓶颈了。

- 答"要模拟跨机房调用"——对，真实流量分散在多个机房，集中打一个就失真了。

- 答"要测不同运营商的链路"——很专业，某一路 CDN 或某个运营商的问题只有分区发压才出来。

- 收口：所以发压能力本身也要设计和验证，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）加两个入口（公网／内网）是常考填空；② 易错点：以为压力机越多越好，关键是分布和同步；③ 简答"为什么在站外 CDN 节点发压"，答覆盖公网链路、分运营商、模拟真实入口。

> **过渡：**外面看是集群发压，推进去看，里面是一套完整的调度系统。

【7.127】压测平台原理示意图-2（2 分钟）

> 外面看是集群发压，推进去看，里面是一条从人到机器的链。

>

> 链子从人开始。压测人员在页面上操作，平台侧存着脚本仓库和数据库。脚本仓库存脚本，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。

>

> 中间三个服务是核心。任务调度服务管什么时候、把哪个任务、发给哪些 Agent。结果采集服务从各个 Agent 收数据。实时计算服务把这些数据算成指标，TPS、响应时间、错误率、分位数。

>

> 实时两个字为什么要单说？因为压测是动的。你盯着曲线，TPS 掉了、错误率上来了，得当场决定是加压、减压还是喊停。结果要是等压完才出来，这次压测就白跑了。

>

> 再往下是数据通道。Kafka 做消息缓冲，ES 做存储和检索。为什么用这个组合？压测的数据量太大。成千上万个 Agent 每秒上报好几条采样，直接往数据库写，库会被打爆。Kafka 削峰，ES 能聚合查询，这是常见做法。

>

> 再往执行侧。VIP 是统一入口，后面挂多个 Agent，最后一环是控制器。控制器管同步启动。所有 Agent 必须同一时刻开始发，否则并发数根本推不上去。上一页说的统一调度，落到这儿就是这台控制器。

>

> 这条链顺下来，压测平台就不神秘了。它是个分布式任务系统，解决调度、采集、计算、存储四件事。你以后看任何压测平台，包括后面的 Takin，都能很快找到对应模块。

>

> 最后一句实操提醒：平台自己的容量也要算。几万个 Agent 一起上报，采集服务先扛不住，你看到的数据就是失真的。

提问：实时计算这一环能不能省，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"——正是核心，曲线异常时得马上做决策。

- 答"为了保住生产"——对，实时看到水位才来得及停。

- 答"事后也得算，但会错过止损时机"——这个回答最平衡，事后分析仍要做，实时是刚需。

- 收口：所以平台的能力指标里，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）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 判断"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，可发什么数据？这一环最容易被低估。

【7.128】压测数据处理过程（3 分钟）

> 很多人以为压测数据就是随便造一批用户和商品。这么想，压出来的结论是假的。

>

> 流程是这样。线上流量数据先进来，过脱敏、筛选、排序。再进数据修正模型加工，然后过数据验证模块。最后形成压测数据池，存下来、分发出去。一部分给压力机集群，一部分回压测平台。右边画的是真实的系统构成。网关、Nginx、应用、大数据平台、核心交换机、MQ、Redis、网盘。课件还专门注了一句，MQ 是消息队列，Redis 代表缓存。

>

> 每个环节为什么在，我一个个说。

>

> 脱敏。生产数据里有手机号、身份证、地址、金额，不脱敏不能用。这不只是合规，也是安全。压测环境的防护等级一般比生产低。

>

> 筛选。不是所有线上数据都适合当输入。得挑有代表性的：活跃用户、真实在架的商品、有效的优惠券。拿一批已注销用户的数据去压，等于没压。

>

> 排序。这一条最容易被跳过。排序是为了造出真实的数据倾斜。真实流量里少数商品扛住大部分访问，你均匀分布地压，热点问题一辈子露不出来。

>

> 数据修正模型。把筛完的数据调成压测要的规模。十万用户要放大到一百万，就按规则生成，同时把分布特征保住。放大不能变成摊平。

>

> 数据验证模块。查这批数据能不能用，字段全不全、关联对不对、会不会一压就触发业务校验失败。没验过的数据，压出来的错误率是假的。

>

> 压测数据池。这是最终产物，统一存、按需分发。为什么要分发？压力机分散在各地，让它们就近取数，别让下载数据成了瓶颈。

>

> 落到一句：压测数据不是造数据，是把真实世界的分布复制一份进来。

> **提问：**排序这个环节为什么会影响压测结论的有效性？

> **预设回答：**

- 答"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"缓存命中率会不一样"——很专业，热点集中时命中高，均匀分布时穿透严重。

- 答"数据库行锁会不一样"——也对，热点商品会产生锁竞争，那是最典型的瓶颈。

- 收口：所以数据得保住分布特征，热点比例、读写比、参数分布，三样都得像真的。

> **易错点 / 考点：**① 处理流程六个环节常考排序题；② 易错点：把脱敏只当合规要求，它同样是压测环境的安全要求；③ 简答"为什么要脱敏和验证"，分安全合规、结论有效两条答。

> **过渡：**数据有了，脚本呢？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）

> 几百个接口的脚本，三个人写两周。有没有更省的办法？有，把真实流量录下来再放回去。

>

> 图分两块，录制和回放。

>

> 录制这块，靠分光分流和 Agent 把真实流量采下来。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，说明采的不只是 HTTP 请求，消息也可能被采。

>

> 关键词是分光分流。它是在网络设备上把流量复制一份出来，原始请求不受影响。这一点很要紧，录制绝不能拖慢生产。Agent 是另一条路，在应用进程里埋点采。拿到的信息更全，能到方法级参数。代价是对应用有侵入。

>

> 回放这块，把录下来的流量按需重新发出去。图上还标了手工上传，意思是除了自动录的，也能人工准备一批请求导进去。

>

> 它值钱的地方三条。第一，脚本自动生成。人工要几周，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，各种边界天然覆盖。第三，能复现线上问题。线上出现异常，把那段流量录下来在测试环境回放，比猜快得多。

>

> 但它有三个限制，得说清。

>

> 一，录下来的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，一放大就出事：同一批订单重复提交，同一批优惠券重复使用。所以回放前必须做参数化改造，把和幂等有关的字段换掉。

>

> 二，写操作的流量要小心。录的时候是真下单，放的时候打到真库，就是重复下单。必须走影子库。

>

> 三，流量会过期。上个月录的，这个月商品下架、接口改了，一回放全是报错，得定期重录。

>

> 所以它解决的是脚本从哪来，解决不了数据隔离和参数唯一，这两件还得靠平台。

> **提问：**录下来的流量为什么不能原样重复回放一百万次？

> **预设回答：**

- 答"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"优惠券只能用一次"——对，这类幂等约束到处都是。

- 答"放大并发是另一个目的，原样回放只是重演"——很到位。

- 收口：所以回放前得改参数，唯一键、时间戳、金额、状态，要换的换，要重算的重算。

> **易错点 / 考点：**① 录制与回放各自的组件常考连线；② 易错点：认为录制回放等于压测，它只是脚本和参数的来源；③ 简答"录制回放的局限"，答参数唯一性、数据隔离、流量时效。

> **过渡：**工具链都有了，可全链路压测从来不是一个技术动作，是一次跨团队的项目。

【7.130】全链路压测全流程（3 分钟）

> 全链路压测做砸，原因常常不在技术上。

>

> 流程分三段：准备、校验、实施。这几句值得背，因为它讲的是做事的方法，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，边界先画清。场景、流量、方案设计评审，压什么、灌多少、怎么灌，得评审，不能一个人拍。基础数据构造，商品、促销、优惠券这些业务数据。脚本和模型创建调试，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，把数据推到各台压力机。最后发布压测实施计划，时间、人员、联系机制、应急预案。

>

> 这七件里，三件不是技术活：范围界定、评审、计划。这就说明失败原因多半出在组织上。谁负责、谁签字、出事打谁的电话，没定清楚，技术再强也压不下去。

>

> 校验阶段三件事。小流量验证，先灌一点点，确认链路通、影子库生效、监控正常。线上压测流量验证，确认流量标记真被识别、数据真落进影子库。机房流量切换，验证切换过程中业务还可用。三件的共同目的只有一个：放大之前，先确认不会出事。

>

> 实施阶段四件事。压测执行加动态调整流量，一边压一边看曲线，要加就加，要减就减。应用降级演练，顺手把降级开关拉一下。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。

>

> 动态调整和降级演练这两项要单说。前者说明压测是探索出来的，不是设个并发数然后等结果。后者说明压测不光测容量，还是验证预案的时机。平时不敢碰的开关，这时候正好拉一下。

>

> 方法就一句：准备占七成，执行占三成。

> **提问：**小流量验证这一步能不能省，直接上大流量不是更快?

> **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"要确认影子库真生效"——非常关键，标记没生效，大流量一上去就污染真实数据。

- 答"要确认监控看得见"——对，看不见就没法判断还要不要继续。

- 收口：小流量验的是三件：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 三阶段及子项是高频简答和排序题，小流量验证、降级演练最容易被漏；② 易错点：把压测执行当成整个流程，准备阶段的工作量才是最大的；③ 判断"压测就是设个并发数跑一遍"——错，它是验证、放大、调整的探索过程。

> **【思政落点 13】**：这套流程里最不显眼的是校验阶段那三件小事，最容易被砍的也是它们。可"先小流量、再确认隔离、最后才放大"，正是工程上对风险的基本态度：不赌直觉，把每一步都验证过再往前走。落到你们身上，课程设计里少一点"大概没问题"，多一点"我拿一小批试过了"。可靠不是自信换来的，是留好退路换来的。

> **过渡：**流程讲完，最后看平台要提供哪些能力，才算够用。

【7.131】压测平台提供的整体能力（3 分钟）

一个压测平台，光能压还不够。压完得能说出话。

>

课件把能力分成六块，从发什么到得出什么，正好一条链。

>

第一块，压测模型。四种发压方式。动态发压，压的过程中随时调流量大小。峰值模拟，模拟真实曲线的尖峰，不是恒定压力。流量回放，把录下来的流量按倍数放出去。智能寻值，自动一步步加压，试探上限，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。人工是测试人员看着曲线手动调，自动是按预设策略跑。为什么还留人工？第一次压你根本不知道系统的脾气，得人盯着。摸清了再自动化。

>

第三块，数据采集。采六类。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，下游依赖的调用量和耗时。还有健康状态。为什么要这么全？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。产出四样。水位告警，哪个资源快满了。结论预研，这次大概能得出什么。容量预期，按当前配置能扛多少。再加上对上面六类数据的综合分析。这一层干的事，是把数据变成判断。

>

第五块，评估策略。四个层级：单点容量、场景容量、集群容量、链路容量。分层的好处是出问题时能定位到哪一层不够。

>

第六块，评估结论。三样交付物：评估报告、优化方案、风险建议。后两样最要紧。压测的目的不是给个数

> 字，是告诉你要改什么、哪里还有风险。

>

> 整条链看下来，平台的价值不在能压多高，在能不能把结果变成决策。

> **提问：**只压不采的压测，为什么几乎没有价值？

> **预设回答：**

- 答"不知道瓶颈在哪"——正是核心，只知道 TPS 低，不知道改哪里。

- 答"定位不到具体服务"——对，链路数据才能告诉你哪一跳慢。

- 答"下次还得从头摸"——很实际，没有数据沉淀就没法横向对比。

- 收口：所以压测的产出是数字加归因，只有数字的报告不合格。

> **易错点 / 考点：**① 六块能力及各子项常考多选和连线，智能寻值、水位告警最容易漏；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 简答"压测报告要包含什么"，答容量结论、瓶颈归因、优化建议、风险提示。

> **过渡：**能力清单摆完了，看一个具体实现，开源的 Takin。

【7.132】现在来看开源的 Takin（3 分钟）

> 有一套方案，敢在生产环境直接压下单接口。它凭什么敢？

>

> 这是 Takin，中文名羚牛。课件的定义是这一句。Takin 基于 Java 语言开发，是一套生产全链路压测系统。它能在无业务代码侵入的情况下，嵌入各个应用程序节点。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。

>

> 这句定义里有四个点，每个点背后都是一个工程难题。

>

> 第一，基于 Java。这决定了它的接入方式是 Java Agent，靠字节码增强在运行时织入逻辑，所以不用你改

> 代码。也决定了它的适用范围：Java 微服务系统。

>

> 第二，业务代码 0 侵入。课件专门列了一条：接入、采集和实现逻辑控制时，不需要改任何业务代码。为什么这件事这么要紧？每个系统接入都得改代码的话，全链路压测根本推不动。几十个团队、上百个服务，改一遍要一年，每次发版还得维护。0 侵入是能落地的前提。

>

> 第三，数据安全隔离。课件原话是：不污染生产环境业务数据，可以在生产环境对写类型接口做直接的性能测试。后半句是分水岭。很多方案只能压读接口，写操作会脏数据。敢压写接口，才算真能验证下单、支付这类主链路。

>

> 第四，安全性能压测和性能瓶颈快速定位。前者是在生产上压测不影响业务，后者是结果能直接指出链路里哪个节点卡住。

>

> 这四条是递进关系。0 侵入让它接得进来，数据隔离让它敢上生产，安全压测让它敢压写，瓶颈定位让它压完有用。少一条，这套方案就不完整。

>

> 最后一句：Takin 是开源项目，仓库地址课件给了。开源的价值不只在免费，更在你能看到它怎么做出影子库、怎么打流量标、怎么织字节码。这些设计思路比工具本身更值得看。

提问：敢对写接口做压测，为什么比压到很高的 TPS 更能说明一套方案成熟？

预设回答：

- 答"读接口压测不真实"——对，真实业务一半以上是写操作。

- 答"写会脏数据，得有影子库"——正是难点所在，涉及中间件改造。

- 答"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：判断一套方案，先问写接口敢不敢压，再问能压多高。

易错点 / 考点：① Takin 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

> 侵入理解成不用部署，Agent 还是要装的，只是不改业务代码；③ 判断"全链路压测只能压读接口"——有了影子库，写接口照样能压。

> **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）

> 看架构图最容易犯的错，是从左上角开始，一个框一个框念。

>

> 换个方式，抓主线。任何全链路压测平台的架构，都跑不出四条线。

>

> 第一条，控制面。平台得有个大脑，管应用、配影子库表、配挡板和白名单、编排场景、下发任务。这一层是控制台加后端服务，人和平台打交道的地方。

>

> 第二条，数据面。压测流量要在中间件层被认出来、被改写。数据库访问导向影子库，缓存访问导向影子缓存，消息加上标记。数据隔离就发生在这儿。关键点是改写发生在中间件和客户端 SDK 层，不落在业务代码里。这才叫 0 侵入。

>

> 第三条，采集面。各个应用节点上的 Agent 采链路数据、调用数据、资源数据，汇总回平台。少了这一层，压测只有结果，没有归因。

>

> 第四条，执行面。压力机按场景和流量模型发压，把结果回传。

>

> 四条线对应四个问句：谁指挥，数据去哪，怎么看得见，谁来压。拿这四个问句去拆任何一个压测平台，商业的、开源的、自研的，都拆得开。

>

> 还有个设计取向值得说。它是平台加 Agent 加中间件增强拼起来的，不是一个大一统的单体系统。好处是每个应用接个 Agent 就能参与，扩展性好。代价是对中间件版本有要求，你得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，第一个问题不是它能压多高，是我的技术栈它认不认。

> **提问：**这类平台的技术栈支持范围，为什么常常比性能上限更影响落地?

> **预设回答：**

- 答"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"改中间件成本极高"——对，很多团队宁可放弃方案也不动中间件。

- 答"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 四个层面（控制、数据、采集、执行）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 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）

> 打开一个压测平台的控制台，第一眼会觉得它像个后台管理系统。别被这个印象带走。

>

> 界面上的信息，说白了就五块。

>

> 一，应用与接入管理。哪些应用接进来了，每个应用的 Agent 状态怎么样，版本一致不一致。这块决定能不能压。

>

> 二，配置管理。影子库表、挡板、白名单都在这里。这三样是数据隔离的配套。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。配错了，要么污染生产，要么压出来的结果不是真的。

>

> 三，场景与脚本。一个场景描述的是一次压测：压哪些接口、什么比例、多少并发、跑多久。脚本描述单个请求怎么发。

>

> 四，任务与执行。发起、看进度、动态调流量、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。

>

> 五，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。

>

> 五块看完，有个观念得立起来。这类界面不是点点按钮就出报告的工具，它是一块风险控制面板。你在这里做的每个配置，回答的都是同一个问题：我怎么保证这次压测不出事。

>

> 还有个很实在的建议。第一次用，先在测试环境把整套流程走一遍。应用接进来、影子库建好、挡板配好、小流量验证通过，再去碰生产。这一步省掉，风险就不在你手里了。

提问：白名单配置在生产环境压测里为什么是必需的?

预设回答：

- 答"要排除某些真实用户"——对，避免压测流量影响真实交易。

- 答"要排除特殊接口"——也对，结算、清算这类接口不能重复执行。

- 答"要保证压测流量能被识别"——很专业，白名单常和流量标识配合用。

- 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）的作用常考简答；② 易错点：以为有界面所以很安全，风险取决于配置，不取决于工具；③ 判断"压测可以直接在生产开始，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）

课件只给了一个 GitHub 地址，看着没什么可讲的。可怎么用开源项目，本身就是一项能力。

>

> Takin 的仓库在 github.com/shulieTech/Takin。地址记下来，课程设计的参考资料里能用上。

>

> 拿到一个开源压测平台，三件事。

>

> 第一，它可读。你翻它的代码，能看到流量标记怎么实现、数据源怎么改写。还有字节码在哪织进去、影子表怎么命名。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。所以学它的顺序是：先读架构，再读关键实现，最后才跑起来。

>

> 第二，它可跑。一般路子是这样。看 README 弄清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。接一个示例应用，跑通一次最简单的压测。别一上来就想着接自己的生产系统，先在示例上把链路走顺。

>

> 第三，它可评。四个维度。活跃度，最近有没有提交、issue 有没有人回。文档质量，有没有部署和接入指南。依赖复杂度，要装多少、对中间件版本有什么要求。可维护性，代码结构清不清楚、你能不能自己改。这四条决定它能不能真进项目。

>

> 一句实话，开源不等于零成本。部署、调试、改造、维护都要投人力，出了问题也只能自己查。所以选开源还是选商业，就看你是想用钱换时间，还是用时间换钱。

>

> 给课程设计一句建议：把它当读源码学架构的样本，别当必须部署成功的任务。你们时间有限，读明白影子库和流量标记怎么设计，收获比装一遍大。

> **提问：**让你判断这个开源压测平台能不能用在项目上，你先看哪三样？

> **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 这块知识点少，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，忽略部署和维护成本；③ 仓库地址作为课程设计参考资料记下来就够，不必背具体路径。

> **过渡：**真要用起来，得先把一套配置做完。课件列了七项。

【7.136】Takin 使用（3 分钟）

> 配置项列了七条，从上往下一条条填，多半要返工。

>

> 课件列的七项是：应用配置、影子库表配置、挡板配置、白名单配置、链路梳理、脚本配置、场景配置。还给两篇实践文章，news.shulie.io 的 ?p=2987 和 ?p=3661。

>

> 一项一项说它在解决什么。

>

> 应用配置。把参与压测的应用接进来，应用名、部署节点、Agent 版本。这是"哪些系统在链路上"的落地。

>

> 链路梳理。这一项最容易跳过，也最要紧。它要回答：一次请求经过哪些应用、每个应用调了哪些下游、哪些是同步调用哪些走消息。没有链路图，压测结果没法归因。你只知道整体慢，不知道慢在哪一跳。

>

> 影子库表配置。指定压测数据写到哪些库、哪些表。常见做法是同实例不同库，或者同库不同表前缀，让压测数据和生产数据结构一样、物理分开。数据隔离的核心就在这一条。

>

> 挡板配置。挡板用来拦住那些不能真调的下游，真实的支付网关、真实的短信通道、外部结算。压测时让它们返回预设结果，既没有真实副作用，又能把变量控住。

>

> 白名单配置。划清哪些流量参与压测。典型用法是只让带标记的流量走影子库，剩下的走正常路径。

>

> 脚本配置。请求怎么发、参数从哪来、断言是什么。脚本质量决定这次压测像不像真的。

>

> 场景配置。把脚本、流量模型、并发策略、持续时间拼成一个能跑的场面。

>

> 七项里最容易出事的是两项：链路梳理和影子库表。前者决定结论有没有用，后者决定生产会不会被污染。这两项别顺手做，单独拉个评审。至于动手顺序，把链路梳理提到前面。它不清楚，后面全是猜。

提问：挡板配置为什么是压测里不可少的一环，而不是能省则省？

预设回答：

- 答"有些下游不能真调"——正是，支付、短信、外部结算真调会产生真实副作用。

- 答"要控住变量"——很专业，把外部不稳定因素挡掉，才测得出自己系统的问题。

- 答"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，它是实验设计的一部分。

易错点 / 考点：① 七项配置常考多选；② 易错点：把链路梳理当成画个图看看，它是归因的基础；③ 简答"挡板的作用"，答避免真实副作用、控制变量、保护下游。

过渡：Takin 自己的脚本体系有门槛，所以它还留了一条路，和 JMeter 集成。

【7.137】与 JMeter 集成 -1（2 分钟）

> 团队里已经攒了几百个 JMeter 脚本，上全链路压测，这些脚本要扔掉吗？

>

> 不用。Takin 能和 JMeter 集成，道理很直白。JMeter 是业界最普及的压测工具，团队上手成本最低。Takin 给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。两边接上，用熟悉的 JMeter 写脚本，用 Takin 的平台能力跑全链路。

>

> 这个组合解决三个现实问题。

>

> 一，脚本资产复用。过去几年攒的 JMeter 脚本不用重写。这一条对落地特别重要。换工具最大的阻力从来不是功能不够，是"我以前的脚本怎么办"。

>

> 二，学习成本低。压测人员不用再学一套脚本语言，还是线程组、取样器、断言、定时器那几样。

>

> 三，能力互补。JMeter 强在发压和脚本编排，弱在生产环境的数据隔离和全链路追踪。Takin 正好反过来。两边拼上，谁都不缺。

>

> 技术上通常是这么接的。Takin 提供一个 JMeter 插件或后端监听器。让 JMeter 发出的请求带上压测标记，结果回传给平台。平台认得出这批是压测流量，就路由到影子库；也收得到结果，能做汇总分析。

>

> 从学习方法上说，这里给了个通用思路：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，直接用开源生态。力气花在隔离、标记、采集、分析这些真正难的地方。很多企业自研压测平台，底层还是 JMeter 或者 Gatling，原因就在这儿。

>

> 一句话，JMeter 解决怎么发，Takin 解决发到生产怎么不出事。

> **提问：**企业自研压测平台，为什么很少从头写一个发压引擎？

> **预设回答：**

- 答"没必要，JMeter 够用"——对，发压引擎的成熟度很难超越。

- 答"自研成本高、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：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）常考简答；② 易错点：以为集成后 JMeter 就能做全链路压测，全链路能力来自 Takin 的平台侧；③ 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）

> 同一份集成文档，有人半小时跑通，有人三天还在报错。差别在读的顺序。

>

> 课件给的是 Takin 和 JMeter 集成的 README。地址在 github.com/shulieTech/Takin-jmeter。具体步骤看文档，我讲怎么读这类文档。

>

> 四步。

>

> 第一步，先看前置条件。要哪个版本的 JMeter，要装什么插件，Takin 平台侧得先配好什么。集成失败十有八九败在这一步被跳过。

>

> 第二步，看数据流。请求从 JMeter 发出去，中间经过谁，标记加在哪，结果怎么回传。把这条链画出来就明白了，集成本质上就两件事：加标记、回传结果。

>

> 第三步，看配置项清单。哪些必填、哪些有默认值、哪些填错不报错。最后这类最坑。配置错了它不吭声，压完了你才发现数据根本没进影子库。

>

> 第四步，看验证方法。文档里一般会写怎么确认集成成功，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。

>

> 这四步不只对 Takin 有用，你以后读任何技术文档都能这么走。

>

> 还有个观念：集成类工作的风险主要在边界上。JMeter 有版本边界，Takin 有版本边界，中间件也有版本边界。三个边界两两组合，就是你要验的矩阵。所以动手之前先列版本矩阵。

>

> 和课程作业相关的一句：课程设计要演示全链路压测，不必真搭一套 Takin。用 JMeter 压一个两三个服务的链路。加上流量标记和影子表的简化实现，把原理讲清楚。这比部署一套完整平台有价值。考的是理解，不是部署能力。

> **提问：**配置错了但不报错，为什么比一启动就失败更危险？

> **预设回答：**

- 答"你会以为成了"——正是，压完才发现数据没隔离，那时已经污染生产了。

- 答"排查成本高"——对，没有报错就没线索，只能靠比对数据发现异常。

- 答"所以得有验证步骤"——这就是第四步存在的理由。

- 收口：判断一个系统做得好不好，看它出错时是不是容易被发现，这叫可观测性。

> **易错点 / 考点：**① 这块偏方法，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，不看数据有没有正确隔离；③ 判断"集成成功等于压测能发出请求"——错，还要验证标记生效和结果回传。

> **过渡：**集成好了，压出来的数据怎么看？最后两块讲结果怎么展示。

【7.139】压测结果展示（2 分钟）

压测跑完，一堆曲线摆在面前。有人第一件事是找最好看的那一条。

>

> 结果展示的任务，不是把图做漂亮，是让人在最短时间里做出判断。

>

> 一份结果要回答四个问题，对应四类视图。

>

> 一，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，是在目标 TPS 下错误率是不是零、响应时间在不在 SLA 里。TPS 冲得很高、错误率三成，那不叫扛住，那是雪崩前的一瞬间。

>

> 二，响应时间怎么分布的。必须看分位数，P90、P95、P99，不能只看平均值。平均值最会骗人。九成请求 50 毫秒，一成请求 5 秒，平均值可能只有 500 毫秒，可用户体验是十个人里有一个卡死。

>

> 三，瓶颈在哪。看链路耗时分布，每一跳服务、每一次数据库调用、每一次缓存访问占多少。占比最高的那一跳，就是优化目标。

>

> 四，资源水位怎么样。CPU、内存、IO、网络、连接池、线程池。有个常见现象：应用 CPU 才四成，TPS 就是上不去。这时候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。

>

> 形式上，好的报告有一条主线。结论先行，先说能不能扛住目标流量。再给证据，曲线和分位数。再给归因，链路和资源。最后给建议，扩哪里、改什么、预期多少收益。

>

> 结论先行这四个字要单说。报告第一页就要给出判断，不能让读者翻二十页图表自己总结。这是工程沟通的基本功。

提问：压测报告里，平均响应时间这个指标为什么说服力很弱？

> **预设回答：**

- 答"被极端值拉平了"——正是，平均值掩盖长尾。

- 答"用户感受到的是最慢那几次"——对，卡一次就印象深。

- 答"SLA 通常按分位数定"——很专业，SLA 常写成 P99 小于某个值。

- 收口：看性能必须看分布，平均值只能当整体量级的粗参考。

> **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 简答"怎么定位性能瓶颈"，答链路耗时分布加资源水位加队列类指标。

> **过渡：**结果展示还有一块，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）

> 一份压测报告，业务负责人只翻开头那一块。他看的是什么？

>

> 看压测概览。概览干的事，是把一次压测的关键信息压成一页，让没参与的人也能看懂。

>

> 六项内容。

>

> 一，基本信息。什么时候压的、压了多久、压的哪个场景、多少并发、目标 TPS 多少。没有这些，任何数字都没法解读。八千 TPS 是好是坏，取决于目标和环境。

>

> 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。这几项要能一眼看出达到目标没有。

>

> 三，SLA 达成情况。拿实际值和事先约定的指标逐条对比，打勾或者打叉。这一项最见专业度。压测不是看看能跑多快，是验证有没有兑现承诺。

>

> 四，瓶颈排序。按耗时占比或者资源占用排，列出前几个瓶颈节点。这一项决定压测之后干什么。

>

> 五，异常与告警。压的过程中触发了哪些告警、有没有降级、有没有错误日志突增。这些信息往往比平均指标更有用。

>

> 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：扩多少、优化哪个接口、调哪个参数。

>

> 有个观念得立住：概览不是简化版报告，它是决策界面。看它的人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。两类人看同一份东西，都得拿到自己要的那部分。

>

> 课程设计里如果你的项目涉及性能，报告里就留这么一块。这块做好了，说明你想清楚了压测是为什么服务的。

> **提问：**目标 TPS 为什么必须写在概览里，不能放到报告后面?

> **预设回答：**

- 答"没有目标就没法判好坏"——正是，八千是高是低，要看目标是多少。

- 答"读者要先看到结论"——对，概览就是放结论的地方。

- 答"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：所以第一步是定目标，不是先跑起来，这又回到 7.1 讲的从需求开始明确性能。

易错点 / 考点：① 概览应包含的要素常考简答；② 易错点：把概览理解成几张图，它是结论页；③ 判断"压测报告越长越专业"——错，结论先行、证据可查才算专业。

【思政落点 14】：概览写得好不好，考的是诚实。数据不好看的时候，最容易做的事是挑一条好看的曲线放上去，把分位数换成平均值，把有条件能扛说成能扛。可这份东西是给人做决策用的。你美化一次，别人就照着错的判断去投钱、去排期。把失败数、把没达标的 SLA 写在最前面，是技术人最基本的职业底线。

过渡：全链路压测讲到这儿收完，再往后是平台特点的小结和混沌工程，那是下一片的内容。

【7.141】小结：压测平台特点（3 分钟）

压测最容易骗人的地方，是首页全绿。首页接口漂亮，QPS 撑到八千，你就以为这套系统能扛。真去按下单，链路里最慢的那一段根本没被压到。结论只覆盖了半个系统。

>

一个能干活儿的压测平台，第一件事是协议覆盖够宽。现实系统里不只 HTTP。还有 HTTPS，还有 JSF 这类 RPC。还有 HBase、数据库、各类中间件。为什么这条排第一？压不到落库，就压不出连接池先满；压不到消息，就压不出消费积压。你的结论，只能覆盖你发得出去的那几种协议。

>

第二件是发压模式够多。动态发压、智能寻点、标准模式、梯度模式、集合点模式。五种摆在这儿，不是凑数。问题不一样，压法就不一样。要找容量上限，就一级一级往上加，这叫梯度。要复现秒杀那种瞬时洪峰，就让一批请求同时到达，这叫集合点。要边压边调参数，那是动态发压。手上只有一种模式，你就只能回答一种问题。

>

第三件是资源分配。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。为什么这也算平台本事？压测机是稀缺的。能不能重复用，直接决定你一天跑几轮。跑不动的组，常常不是没脚本，是每一轮都在等机器。

>

第四件是结果查看。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，跟压测曲线放一起看。为什么非要放一起？只看压力曲线，你分不清是系统到顶，还是压力机自己先到顶。这两件事的处置完全不一样。

>

> 四件事收成一句：压不全，等于白压。

> **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挪到队列消费端，消费积压会让整条链看起来没超时、实际没结果。

- 答"那就用脚本自己拼协议"——可行，代价高。等于你在自己造一个平台，也正是要讲平台特点的原因：这些能力交给平台，别每个项目重造一遍。

- 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：**容易和"性能测试类型"混着考。**发压模式是工具能力**，讲的是压力怎么施加；**性能测试类型是测试目的**，讲的是负载、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"，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）

> 压测会给你一个很舒服的答案：这台机器能扛八千。它不回答另一个问题：把一台拔掉，系统还剩下什么。

>

> 两个问题看着像，其实不是一回事。压测是把水龙头开到最大，看管道会不会爆。混沌工程是照着某一根管子敲一下，看整套系统还能不能供水。一个测承载，一个测残缺。

>

> 为什么这个能力现在变得要紧？系统已经不是一台机器上一个程序了。几十上百个服务，上千个实例，跨机

> 房跨云。这种系统里，故障不是意外，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。你保证不了它不出故障，只能保证出了故障它仍然可用。

>

> 所以这个词得理解准。它不是制造混乱。它是用可控的人造故障，去换真实故障面前的那点确定性。故意让它坏，是为了它真坏的时候不塌。

>

> 定义给一句。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。它也是生产里抵御突发事件的复杂技术方案。三个词各有用处。实验，不是破坏。学科，有方法，不是拍脑袋。弹性，目标不是零故障。

>

> 这一节按这条线走。先看复杂系统为什么必然出问题，再看定义、五项原则、四块关键工作、爆炸半径怎么控。然后看它自己的短处，最后落到工具和实践。顺序别打乱。原则没立住就上工具，做出来的只是随机破坏。

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"——最典型的误判。压测验的是满负载下的表现，它不验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。重点不在拔，在实验设计、稳态判定、爆炸半径控制这一整套方法。

- 收口：两套能力互补，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）

> 一个请求进来，要过几道关卡？你数一遍。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、配置中心和监控探针。

>

> 组件一多，每次请求要过的环节就多，而每个环节都有一个失败概率。串联的可用性是相乘的。单个环节四个 9，就是 0.9999。串五十个环节，0.9999 的 50 次方约等于 0.995。整条链掉到三个 9 以下。每个零件都稳，整体不稳，原因就在这个乘法上。

>

> 第二个变化是耦合。以前一个模块挂了，用户看到的是这个功能不能用。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。故障的影响半径，跟着依赖的半径一起扩大。课件图上"影响"那两个字的分量，就在这儿。图注的来源标的是 Data Center Knowledge。

>

> 第三个变化不写在任何架构图上：人的认知跟不上了。系统大到某个程度，没有谁能在脑子里推演出这里坏了会怎样。推不出来，就只能等它真坏一次。而那时候代价已经发生。

>

> 有人会问，那就少串几个环节行不行？行。这正是把同步改异步、把强依赖改弱依赖这些手段的动机，也是后面讲熔断和降级的原因。但环节减不到零，风险也就减不到零。并联能救一部分，并联算的是一减都失效的概率，可真实的分布式系统里，能并联的部分比你想的少。

>

> 链条走到这儿，只剩一句结论。复杂系统里，不出故障不可承诺，出了故障还活着才可以设计。后者就是接下来要讲的东西。

> **提问：**如果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、同步改异步这些架构手段的动机。

- 追问：所以可用性不是一个组件的事，是整条链路的事。这句话后面讲稳态标准时还要用。

> **易错点 / 考点：**常出计算题或判断题。**串联相乘、并联取"1 减都失效的概率"**，两个口径别混。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，那就得看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）

> 先看四个数字，别急着看结论。大于 87.6 小时、大于 8.7 小时、大于 52 分钟、小于 5 分钟。这四个数是一年里允许停机的总时长，按四档可用性排下来。数据标的是 Statista 2022，来源混沌工程实验室。

>

> 对上几个 9 就清楚了。87.6 小时是 99%，8.7 小时是 99.9%，52 分钟是 99.99%，5 分钟是 99.999%。每多一个 9，允许的停机时间少一个数量级。

>

> 为什么从三个 9 到四个 9 最贵？因为前一步便宜。99% 到 99.9%，把明显的 bug 修掉、加几个监控，基本就到。99.9% 到 99.99%，要上冗余、上自动切换、上降级预案，三样都得有人维护。再往上，还要有人验证这些手段在故障里真的生效。那一步的钱，花在验证上。

>

> 一小时的损失，行业之间差得远。金额我不念，方法你带走：损失等于直接交易额加恢复成本，加用户流失，加品牌与合规代价。后两项常常比第一项大得多。

>

> 所以可靠性是花钱买的。混沌工程在这笔支出里的位置很具体：让你买的每一分冗余都真能用。不验证，冗余只是纸面上的，故障一来，切换脚本自己先挂。

>

> 答辩里如果被问四个 9 怎么做到，先答主备、自动切换、降级预案这三件，再补一句这些都演练过。少最后

> 那一句，前三条就只是文档。

> **提问：**如果产品现在是三个 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。加机器解决容量，不解决故障时的切换。

- 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"——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。

- 追问：所以四个 9 不是买来的，是设计加验证换来的，验证那一半就是混沌工程。

> **易错点 / 考点：**数字类考点只需记"每多一个 9，年停机时间除以 10"，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）

> 真出事那一刻，你团队第一件事干什么？不少组的答案是先查日志。这张图里有一句话跟这个习惯相反：快速恢复止损优于问题排查。

>

> 故障画像做的事，是把稳定性风险从一句口号，拆成能逐条对照的清单。左边三类问题。异常场景类，看节点异常状态下系统表现成什么样，以及异常场景下的预案是否完善。链路设计类，看各类链路异常预案的正确性、有效性，还有预案验证的常态化。常态化这三个字最重：预案只有演练过才算存在，写在文档里不算。预案设计类，看故障发生时有没有标准化流程，以及止损和排查谁先。

>

> 右边是风险清单，按来源分四组。应用问题一长串：进程僵死、线程池满、连接池满、依赖超时、依赖异常。还有内存泄漏、慢查询、异步阻塞、缓存异常、热点问题、同步问题。组件问题单独列一组。网络问题：网络抖动、延时与丢包、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。

>

> 这份清单怎么用？当体检表。做稳定性评审，一条一条问三句：这一条我们有风险吗，有预案吗，演练过吗。三句都答得出来，这条才算过去。答不出第三句的，基本都是空预案。

>

> 它还是故障库的原料。后面讲基于历史问题抽象通用故障库，抽的就是这张表。所以这张图别只当背景，抄下来，对着自己的系统勾一遍。

> **提问：**"重试风暴"为什么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。

- 答"所以要加退避和熔断"——对，指数退避加抖动，再加熔断和限流，是标准解法。

- 答"重试是好事，不算风险"——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，这就是后面讲的"前进一步、后退两步"。

- 收口：画故障画像的意义，是在出事之前把这类好心的坏设计找出来。

> **易错点 / 考点：**常见考法是归类题：给一个现象，问属于应用、网络还是系统问题。判断口诀：**看谁的资源被耗尽**——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：标准化流程加止损优先。

> **过渡：**这些风险一条条兜，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）

> 有一个词，第一次听会觉得不对劲：以毒攻毒。

>

> 平时做测试，是给正常输入，看输出对不对。混沌工程反过来。主动造一件本来不会发生的坏事，看系统扛不扛得住。担心主库挂了怎么办？那就真把主库切走一次，挑业务低峰、挑小范围流量，看多久恢复、数据

> 有没有乱。担心机房网络抖？那就注延时和丢包，看熔断触发没有、降级生效没有。

>

> 这个思路为什么成立？因为故障躲不掉了。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，不如自己挑时间、挑范围、挑强度把它引出来。可控地坏一次，好过不可控地坏一次。

>

> 课件上方那张是 Google 搜索趋势，标的是 Chaos Engineering。曲线往上走，说明工业界的关注度在涨。趋势图证明不了因果，但它能说明一件事：这不是某一家厂商的口号。

>

> 定义再念一遍，念的时候盯住三个词。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。三个词缺一个，定义就答不全。答辩里只写"主动注入故障"的，多半要扣分。

>

> 跟压测比一句：压测的输入是流量，混沌的输入是故障。两套能力互补，谁也替不了谁。

>

> 最后一句可以带走：主动坏，是为了别被动塌。

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，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"——也对。可以只影响百分之一的流量、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，在真实故障面前差得很远。

- 追问：所以它卖的不是制造故障，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"，那只描述了手段，漏了实验学科、稳态假设、最小影响范围这些方法要素。对比题常问"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：**这套东西到底有没有用？业界做过调研，看数据。

【7.147】混沌工程确实能提升系统可用性（3 分钟）

有人会说，这套东西听着有道理，可真有用吗。有调研。

>

> 课件引用的是 Gremlin 2021 年的那份调研。英文标题就叫 State of Chaos Engineering。这个标题本身就是一句判断：混沌工程确实能提升系统可用性。这是厂商做的调研，性质先说清楚。

>

> 图里要读的不是某个百分比，是三件事。第一，做和不做有差距。调研按是否系统性开展分组。开展了的团队，在可用性目标达成、故障恢复时间、团队对韧性的信心上都更好。最后那个信心不是虚的。知道系统会怎么坏的团队，出事时下手快得多。

>

> 第二，成熟度不同，收益不同。偶尔手动注一次故障的，收益有限。把实验做成常态化、自动化、进了流水线的，收益明显。一年体检一次不叫锻炼。

>

> 第三，最大的阻力不是技术，是人和流程。团队反馈的障碍是怕影响生产、没有明确负责人、缺少稳态指标，不是没有工具。工具最好解决，共识最难。

>

> 引用这类数据要报口径。样本是谁，指标怎么定，都会影响结论。写报告时写清"据 Gremlin 2021 年调研"，别当成严格对照实验的因果结论。

>

> 如果考题问怎么论证收益，答三条就够。可用性指标对比、故障恢复时间对比、演练发现的预案缺陷数量。第三条最容易忘，也最能说明问题。

> **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？合理的话，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"——正确，这就是从线下到生产、从小到大的那条实践路径。

- 收口：所以对"怕影响生产"的正确回应不是"别怕"，是"我们有办法把影响框住"。

> **易错点 / 考点：**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。问"如何论证混沌工程的收益"，标准思路三条：可用性指标对比、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，它有一条清晰的脉络。

【7.148】混沌工程发展历程（3 分钟）

> 一个做法从手艺变成学科，中间要过几道门。混沌工程走了四段。

>

> 第一段是萌芽。最早是几家大型互联网公司发现，系统太复杂，只能在生产里试。于是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。

>

> 第二段是命名与原则化。名字被提出来，做法被提炼成几条原则，就是下一页那五条。这一步最关键，它把某家公司的经验，变成别人也能学的方法。没有这一步，这东西只能靠师傅带徒弟。

>

> 第三段是工具化。一批开源工具和平台出来了，故障注入变成写配置、点按钮。门槛一降，做的人就多。也是从这一段起，混沌工程开始被当成一个产品品类。

>

> 第四段是平台化与云原生化。Kubernetes 普及以后，混沌工程被做成声明式接口和 Operator。它进了云原

> 生基金会的项目体系，成了研发流程里的一环。

>

> 四段连起来是一个规律。方法成熟的标志，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，只能当表演。

>

> 顺便纠一个常见错答：把混沌工程等同于 Netflix 的 Chaos Monkey。那只是一个早期代表性工具，不是整个学科。

>

> 你们课设里做的那点故障注入，处在这条线的最前端，手工的、单点的。别嫌它土。所有平台都是从一条命令开始的。

> **提问：**一个团队只有"出事之后复盘"的机制，没有任何主动演练，它在这条时间轴上大概在哪个位置？缺的是什么？

> **预设回答：**

- 答"萌芽阶段"——大体对，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，这正是后面"从灾难中学习"要讲的。

- 收口：顺序是原则、稳态、工具，别倒过来。倒过来就是为用工具而用工具。

> **易错点 / 考点：**考阶段特征，不考具体年份。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 Chaos Monkey 等同，那只是一个早期工具。

> **过渡：**说到原则，这五条是必背的。

【7.149】混沌工程的原则（5 分钟）

> 五条原则，值得背下来，一条一条说，最后给口诀。

>

> 第一条，建立一个围绕稳定状态行为的假说。假说这两个字最要紧。做实验之前，你先写下一句能被推翻的话。比如：单机房故障时，下单成功率不低于 99.9%。这句话里必须有可观测的指标，和可接受的阈值。没有这句话，注完故障你只能盯着屏幕说好像还行。那不叫实验。

>

> 第二条，在实验中引入反映真实事件的变量。服务器崩溃、硬盘故障、网络连接断开。为什么强调真实？你造的故障越像真会发生的，结论越有用。注一个现实里不可能出现的故障，结论也就没意义。

>

> 第三条，在生产环境中运行实验。这条最反直觉。理由是：只有生产才有真流量、真依赖、真配置、真数据规模。测试环境永远太干净，在里面验证通过的容错，到生产可能根本不成立。它跟第五条是一对：在生产里做，范围压到最小。

>

> 第四条，持续自动化运行实验。一次演练只证明那一天那个版本还行。系统天天在改，昨天扛得住的今天未必。所以实验要进流水线，定期跑。

>

> 第五条，最小化影响范围，也就是爆炸半径控制。

>

> 分层图看一眼。基础设施、基础平台、业务应用三层，横向穿过的是故障注入。上面由人员组织与实践统领。纵向串着计划与报告、演练可观测、安全审计、量化评价、报告改进，最外面一圈是风险保护。意思是：注入只是中间那一刀，前面有计划，后面有观测、审计、评价、改进。缺后面，实验就只剩下折腾。

>

> 口诀：有假说、用真故障、在生产、要自动、控范围。

> **提问：**第三条说在生产环境运行实验，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"——违反第三条，但可以作为起点。优秀实践里写了从线下到生产的进阶路径。

- 追问：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，要能一字不差说出五条。最常错的是漏掉"最小化影响范围"，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**。另一个考点是为什么要在生产做：真实性，含流量、依赖、配置、数据规模。

> **过渡：**原则讲的是立场。落到执行层，要干的活是四块。

【7.150】混沌工程关键工作（4 分钟）

> 原则讲的是立场，落到执行层，真正的活是四块。

>

> 第一块，实验设计。三个动作。故障画像，照着那张风险清单挑出要验证的点。上下游依赖梳理，这件事坏了谁会受影响，是强依赖还是弱依赖。链路受力分析，故障打在这条链的哪个位置，压力往哪里传导。设计没做好，注入了也是白注，因为你不知道自己在验什么。

>

> 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。隔离是演练流量和真实流量分开。兜底是一旦越界立刻自动停。

>

> 第三块，注入故障能力。这块最容易被当成只有运维的事。它有三条线。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。测，接口、性能、巡检自动化这类验证层。三条线齐了才叫能力齐备。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、链路级的稳态定义和管理。灵活的稳态可接入能力。翻成人话：每个服务的正常长什么样，得写得出来。单服务的稳态、一个业务的稳态、

> 整条链的稳态不是一回事。这套标准还得方便接进平台，别让每个组自己手写一套。

>

> 四块的关系一句话：设计决定验什么，注入决定怎么验，稳态决定凭什么判合格，半径决定你敢不敢验。

> **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，出了问题不知道修哪个服务。

- 答"三层是一条定位链：链路异常，定位到业务，再定位到服务"——这就是核心，分级是为了既能判断好坏，又能定位原因。

- 收口：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说的关系：假说里写的阈值来自稳态标准；没有稳态标准，假说就是拍脑袋。

> **过渡：**四块里最需要敬畏的是第二块，单独讲。

【7.151】爆炸半径控制（4 分钟）

> 如果领导说这次演练必须全量生产，一次到位才有说服力，你怎么接？

>

> 先把命题说清：在最小的实验范围内，让系统潜在的问题暴露出来，又不至于意外造成更大规模的故障。

>

> 四条实践经验。第一条，选最小爆炸半径。起步就从一个实例、一个接口、百分之一的流量开始。第二条，实验分级发布。随着对系统的信心增加，逐步扩大范围。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，验证过了，才敢动一个集群。

>

> 第三条，混沌不意味着一定要在全量生产环境实施。这句在纠一个误解，好像不做全量就不算混沌。关键在真实，不在全量。第四条，要有完备的自动化和高效预案，具备快速熔断能力。要有一个红色按钮：监控发现指标越线，实验自动停，故障自动恢复。没有这个按钮，别开始。

>

> 再看那张图，四个维度是最实操的部分。环境：实验环境、线下环境、沙盒环境、金丝雀环境、生产环境，越往后越真实也越危险。流量：压测流量、mock 流量、引流流量、回放流量、流量抽样百分之 x。生产里常用的是只让一小部分真流量进实验路径。粒度：请求粒度和用户粒度。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。实例：单实例到多实例。最后落到"测试中间件-请求粒度实验"这种组合上。

>

> 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。

> **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"我坚持只做百分之一流量"——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，再申请扩到百分之一、百分之五"——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约定。

> **易错点 / 考点：**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，不是一个东西。

> **【思政落点 15】**：爆炸半径讲的是敢试，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，是我知道边界在哪、越界会自动停。这份敬畏对应责任：一次没有兜底的演练，可能把几十万用户的订单打掉。请把风险量化、框住、留退路当成习惯，别把风险说小了当成本事。这是这一行最朴素的职业底线。

> **过渡：**说了这么多好处，也得把它的短处讲清楚。

【7.152】混沌工程的局限性（4 分钟）

> 说了这么多好处，得说说它不管用的时候。

>

> 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，也可能继续宕机。原因写得直白：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，发现不了将来才出现的新故障。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，再多的混沌工程也修不好。而且在实验里发现问题之后，想改已经很困难。你知道了，但你改不动。

>

> 第三条，前进一步，后退两步。某一方面的优化，可能增加别处的脆弱性。为了抗住下游超时加了重试，于是引出重试风暴，故障画像里就有这一条。

>

> 第四条，注入本身带来新的复杂性。用漂亮、自动的方式注入故障，同时也在给系统加复杂度。实验做得越多，可能对系统伤害越深。这话在提醒：演练工具自己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。演练天天做，架构照样烂。第六条，一键回滚有欺骗性，数据可能在回滚时已经被破坏。这条最要命。代码能回滚，写坏的订单改不回来，发出去的消息也收不回。

>

> 六条收一句：它是镜子，不是药。照出问题，不负责治好；能验证韧性，不能替代设计。

> **提问：**请举一个具体场景，说明"一键回滚具有欺骗性"。

> **预设回答：**

- 答"回滚了代码，数据已经改了"——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：所以正确做法是回滚加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。

> **易错点 / 考点：**考批判性思维，常出论述题："既然有这么多局限，为什么还要做？"标准答法是两面都答：一面是它能解决的，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；结论落到必要但不充分。六条最好背出三条以上，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。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）

> 先看一个最有画面感的做法。Slack 管它叫灾难片剧场。

>

> 2018 年 1 月启动。为什么叫剧场？因为有人在看。有剧本，有导演，有观众。它把故障演练当成一场演出，不是某个团队关起门来的技术练习。

>

> 流程是严格的，目的也写得明白：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。两个目标是一对，既要学到东西，又不能真出事。

>

> 四个细节。第一，详细的计划和精心的组织。每次演习在指定时间内做，专家集中在一起。内容包括探索性的演习，还有在运营团队里传阅响应计划。传阅这个动作别滑过去。它不是把文档发出去就算完，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，主持人事先写剧本。记录怎么把故障挑起来，运行哪些命令，哪些 Amazon EC2 实例会被涉及。还有所有需要盯的日志、指标和警报。这几项合起来是一份可复现、可审计的记录。出问题能查，成功了能复用。

>

> 第三，主持人当场提问题。课件举的那个问题挺值得琢磨：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。问的是，你在测试环境验过的东西，敢不敢保证它在生产也成立。第四，让运营、研发、甚至管理层在场。演练因此变成一次共同的经验积累。

>

> 这套做法最值得学的不是工具，是组织方式。

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？这看起来像流水记录。

> **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，但更要紧的是出事能快速回退：知道动了哪些实例，才能精确恢复。

- 答"这也是安全审计的凭证"——很到位，第五章那张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。这也是它必须由主持人写、不能事后补的原因。

易错点 / 考点：常考演练的组织要素：**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 关注演练怎么组织，Google 关注另一件事——灾难恢复的目标怎么定。

【7.154】Google's Disaster Recovery Program（4 分钟）

> 承诺写得再漂亮，也要有人验。Google 的灾难恢复项目给的是一个分层定义的办法，很适合拿来当架构题的框架。

>

> 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，要靠恢复手段拉回来。分级的意义在这儿：不是所有服务都必须做到零故障。你先给每个服务定级，再决定投多少钱。

>

> 方法论还有一句：构建专用架构模式，把灾难恢复的方法标准化。让恢复能力的期望与服务中断的场景相匹配。三个关键词。专用架构模式，别每次出事临时想办法。标准化，同样的故障用同样的套路。期望与场景相匹配，这是最省钱的一条，用户能容忍四小时的服务，没必要为它做零中断。

>

> 两个指标必须记。RTO 是恢复时间目标，多久能恢复。RPO 是恢复点目标，能容忍丢多少数据。

>

> 课件给了三组数字，正好演示分级。区域级是 RTO 12 小时、RPO 24 小时。往上一档是 RTO 4 小时、RPO 为零。最高一档是 RTO 为零、RPO 为零。连起来看：级别越高，RTO 越短，RPO 越小，成本越贵。RPO 为零意味着一条数据都不能丢，那要同步复制。RTO 为零意味着不能中断，那要多活、要秒级切流。

>

> 跟混沌工程什么关系？RTO 和 RPO 是写在纸上的承诺，混沌工程是验证它能不能兑现的手段。不演练，你永远不知道那个 12 小时是 12 小时还是 12 天。

> **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。它本来就按天更新，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，成本是数量级差别。

- 答"都按最高标准做最保险"——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：定级的依据是业务影响分析：这个服务停一小时，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，看时间；RPO 管能丢多少，看数据。常见错法是记反，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：成本、复杂度，加上过度设计本身增加脆弱性。

> **过渡：**Google 讲目标怎么定，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）

> 云厂商做混沌，形态跟自建完全不一样。

>

> 课件给的那句英文是关键。用基于 agent 的故障，在 Azure 门户给一台 Linux 虚拟机加 CPU 压力。一句话里三个信息。

>

> 第一，基于 agent。云厂商不可能在你的机器里随便乱动，它要在目标资源上装代理，由代理去执行注入。这是云上和自建最大的形态差别。自建可以侵入，云上必须走代理。

>

> 第二，CPU 压力是注入内容。这就是前面说的注入能力的一种。攻击面很明确：压 CPU，看服务在算力被抢时会不会超时、会不会雪崩。

>

> 第三，通过门户，说明它产品化、界面化。降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挑。

>

> 下面那条命令：az vm identity assign，给虚拟机分配一个托管标识。做实验之前为什么要先做这步？因为注入需要权限。平台要能操作这台机器，要能读它的监控数据，而你不能给它一把万能钥匙。标准做法就是用托管标识做最小授权，只给做实验需要的那点权限。

>

> 还有个误解要拆掉。云上做混沌，不是云厂商替你制造故障，是它给你工具、让你自己造。责任主体还是你的团队，出事还是你兜。

>

> 这条命令里藏着一句更重要的话。在云上做混沌，第一个要解决的不是怎么注入，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。

> **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。凭据由平台托管、自动轮换，比写死在脚本里强。

- 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。这也是自动化运维平台的通用要求。

> **易错点 / 考点：**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。若考权限设计，标准答案必含最小权限原则与可审计。

> **过渡：**实验、故障库、可观察性这三件事，是把敢在生产上捣乱变成有章法地捣乱的关键。逐个看。

【7.156】实验（3 分钟）

> 混沌落到操作层，最小的单位不是一次测试，是一个实验。

>

> 两个词差得远。测试用例问结果对不对，实验问我那个假设站不站得住。

>

> 一个完整的实验五件事。先写清要验证的假设，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。最后给结论：假设成立，韧性得到一次证明；假设被推翻，那就找到一个真实风险点，转成待修项。

>

> 顺序不能反过来。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，注完故障你连这算不算异常都判不了。

>

> 举个反例。只写得出注入 CPU 百分之八十、压五分钟，写不出一句"我认为业务不会受影响"。那这次做的

> 不是实验，是一次故障演练。做完只剩一堆曲线，没人能说清这是异常还是正常。

>

> 还有一点容易被忽略：实验是可以重复的。今天验过的假设，下个版本还得再验，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。这就是它跟随手压一下的区别。

>

> 一句话：先定标准，再动手。

> **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。它是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 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，问题不在种类，在你没事先说清受影响与不受影响的界线。

- 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：先立标准，再动手。

> **过渡：**实验里"注入什么"那一栏，靠的不是灵感，是一个沉淀下来的库。

【7.157】故障库（4 分钟）

> 实验里注什么，靠的不是灵感，是一个沉淀下来的库。看一份实物：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你能做多大。你能注入的故障种类，就是你能发现的风险种类。

>

> 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 IO 压力。Windows 一条、Linux 一条，再加一条任意 Stress-ng 压力。第二类是时间与进程。时间改变、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，再往上是改网络安全组规则。

>

> 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。还有经典云服务关机、Key Vault 禁止访问。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 就是 Azure Kubernetes Service。

>

> 这份清单最值得学的不是长，是颗粒度。同样叫网络故障，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？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。

> **提问：**清单里"网络延迟"和"网络断开"是两条。只做了延迟实验、业务全绿，能不能说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。延迟考验超时与重试，断开考验熔断、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生效。

- 答"那就两条都跑一遍"——对，还要加带防火墙规则的断开和 DNS 失败，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、时间与进程、网络、云资源、容器编排层。答题三步：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？靠下一件事。

【7.158】可观察性（3 分钟）

前面几件事做完，还有一件决定成败的事没提。

>

> 三个词得先分清。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。差别就在这儿：监控回答已知问题，可观察性回答未知问题。混沌工程制造的恰恰是没预料到的那一种异常，靠几张预设大盘不够用。

>

> 可观察性靠三类数据撑起来。指标，是数字随时间的变化，适合看趋势和告警。日志，是离散事件，适合看到底发生了什么。链路追踪，是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：故障注进去以后，用户那边到底感受到了什么。

>

> 再往深一层。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。追踪缺失，你只知道失败了，不知道失败在哪一跳。三类数据还要能串起来，同一个 trace id 从网关一路查到底。串不起来的三份数据，只是三份孤证。

>

> 所以做实验之前先问一句：我的观测手段，够不够支撑一次判定。这个问法把工具问题变成了能力问题。买一套监控不难，做出一套能判定稳态的观测，要有人在指标上花心思。

>

> 不够，就先补观测，再谈注入。

> **提问：**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，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。顺序就是先有观测，再谈注入。

- 收口：观测能力是实验的前提，观测不够，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**预设指标对可任意提问**。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，关键在有没有把指标、日志、追踪用同一个 trace id 关联起来。

> **过渡：**从这儿开始进入优秀实践。第一条讲的是一件更根本的事。

【7.159】优秀实践 1：复杂系统韧性实现之道（4 分钟）

> 简单系统不需要混沌工程。这话听着像劝退，实际上是在划适用面。

>

> 一个单体应用、一个数据库、一台机器，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、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。这种问题读代码读不出来，开会也讨论不出来，只能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。可靠性讲尽量不出事。韧性讲出了事能扛住、能快速恢复。前者靠预防，后者靠承受。混沌工程干的是后一件事：假定故障一定发生，然后问，发生了会怎样。

>

> 课件标注的来源是中国信息通信研究院 2021 年的成果，属于行业层面归纳的实践共识。所以这一条的含义是：把混沌工程放进架构治理的盘子里，别当成测试团队的一个附加动作。它的产出不是通过或者没通过，而是我们在哪一层还扛不住。

>

> 判断一个系统要不要做，不看服务有几个。看它挂了业务停不停、损失多大。三个服务里有一个是支付，那就值得做几个针对性的实验。复杂度不只来自自己的服务数，还来自外部依赖的数量和不确定性。

> **提问：**一个只有三个服务的系统，值得专门上混沌工程吗？值得的话，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。但这三个里有一个是支付或资金，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"——思路很好。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。价值在发现风险，不是凑一份材料，动机错了实验就做成表演。

- 收口：判断标准不是服务几个，是它挂了业务停不停、损失多大。

> **易错点 / 考点：**选择题常考可靠性与韧性的区分，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：演练照着剧本走，混沌工程先有假设再验证，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）

> 事故已经发生了，钱已经赔了，接下来最亏的一件事是什么？写一份报告，然后归档。

>

> 第二条实践两个动作。第一个是从灾难中学习。任何一次真实故障，都是一次昂贵又免费的实验。昂贵，因为它已经造成了损失。免费，因为你不用设计注入，故障自己来了。复盘的价值不在写报告，在把它转成下一次可以主动复现的实验。当时哪个依赖先超时，哪一层没熔断，告警几分钟后才来。问清这三件，就得到一组实验参数。

>

> 第二个动作是细化实验目标。验证系统高可用，这是空目标。验证订单服务挂掉一个实例、注入持续 30 秒、下单成功率不低于 99%，这才是可执行的目标。有对象、有故障、有范围、有阈值。目标越细，结论越硬。目标越空，实验越容易变成跑了一遍好像没事。

>

> 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，不知道测什么。没有第二步，测了也不知道算不算过。

>

> 顺带区分一组容易混的：SLA 是承诺，实验目标是验证承诺的手段。把"系统要 99.99% 可用"写成实验目标，是把承诺当成了方法。

>

> 还有个态度问题：复盘找的是系统的薄弱环节，不是责任人。每次复盘都开成追责会，以后没人讲真话，你也拿不到真实的故障画像。

提问：把"验证系统高可用"写进实验目标，问题出在哪？请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"要写清楚注什么故障、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"写成系统要 99.99% 可用"——这是 SLA 目标，不是实验目标。它描述长期结果，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——在某个故障、某个范围、某个时长下，某个指标不低于某个阈值。

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 16】：从灾难中学习的前提，是允许把问题讲出来。复盘最怕两件事：开成追责会，和把没出事当成没问题。每一次事故都是系统用损失换来的信息；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。敢暴露问题、如实记录、把教训固化成实验，这才是负责。

过渡：事故的教训只停在一份报告里，下一个人还会踩同一个坑。下一步是把它们抽象成通用的东西。

【7.161】示例：基于历史问题抽象建设通用故障库（4 分钟）

先看一次复盘。有个组把事故报告写到第十七页，最后一行是：商品详情页变慢，起因是评论服务响应慢，属偶发。

>

> "偶发"这两个字最费钱。下次换个服务慢，你还得从零查一遍。想跳出这个圈，得把每次事故往上收一层，收成一个能复用的类型。

>

> 图的纵向是三层。最底下是基础设施，就是机房那一层，课件拿风、火、水来代表：风是空调，火是消防，水是恒湿。中间是基础架构，服务器、网络、存储、中间件这些支撑件。最上面是业务应用，你自己写的服务都在这一层。

>

> 图的横向是从历史问题里抽出来的故障类型，列得很密。基础架构那一档：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 异常也在。宕机、假死、磁盘读写慢、网卡打满，加上实例迁移、实例部署、OOM、core。业务应用那一档：下游失败、依赖超时、性能下降。再看格式错误、配置错误、环境错误、业务攻击。

>

> 这么长一串不是拿来背的，是拿来归类的。差别在收到哪一层。你出过一次"某个接口超时"，记成"接口超时"，它只防那一个接口。往上收一层记成"依赖超时"，所有同步调用都套得上。再收一层记成"下游失败"，连下游返回错数据也覆盖住了。

>

> 所以动作是：新故障来了先判断它归哪一类，能归就补参数，归不进去才加一类。库按抽象层级长，不按事故编号长。

>

> 合格的条目有个硬判据：抽象到能复用，具体到能注入。两头都够，才算真收进去了。

提问：某次大促，商品详情页整体变慢，起因是评论服务响应慢。这条要收进哪一类，记成什么？

预设回答：

- 答"记成评论服务慢"——太具体。它只能防这一个服务，换个服务照样出事。

- 答"记成依赖超时，放在业务应用层"——对。收一层之后，所有同步依赖都套得上，还能顺手补两个参数：超时阈值设多少、有没有降级。

- 答"记成性能下降就行"——又收过头了。这么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。最常见的错答把故障库当成一本事故流水——流水只记历史，故障库是可注入的故障类型清单。判断一条能不能入库，三问就够：能转成注入动作吗？能套到别的服务上吗？参数写不写得出来？三问都过，才算合格条目。

> **过渡：**库有了，下一个问题是在哪儿注，一次注多大。

【7.162】优秀实践 3：从线下到生产、从小到大（4 分钟）

能不能直接上生产？这问题年年有人问，理由也现成：测试环境造不出真实流量和真实数据。

>

> 课件给的不是"上或不上"，是两条线一起往前挪。

>

> 第一条线是环境：测试环境，Staging 环境，预览环境里的一股特定流量，生产集群全流量。为什么不跳？每一步验的东西不一样。测试环境验注入动作本身跑不跑得通。Staging 验服务和服务之间的牵连。预览环境放一小股真实流量进来，验线上行为跟线下是否一致。三关都过了，才轮到全流量。

>

> 第二条线是范围：可控流量，单个接口，单机，单集群，单机房，全链路。这条线量的不是"在哪儿做"，是"一次实验能波及多大"。它跟环境那条不是一回事：环境回答位置，范围回答影响。

>

> 影响多大，有个词叫爆炸半径。半径小，最坏结果是你自己先发现一个问题。半径大，最坏结果是全体用户陪你一起发现问题。差别就在这儿。

>

> 所以节奏由证据定，进度表定不了。上一格的实验稳住了，回退动作真跑过一遍，监控也盖得住，才准进下一格。每一格都得有回退开关，开关还得提前验过。开关存在，不等于开关管用。

>

> 收一句：先在最小半径里把方法练熟，再拿可控的半径去换真实的信息。

> **提问：**团队已经能稳定在生产上对单机压 CPU 了，能不能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。全链路延迟同时压所有调用方，超时叠上重试很容易演成雪崩，而那恰好不是你想验的那件事。

- 答"不行，中间缺了单集群和单机房两级"——对，跳级等于放弃这两级能给你的信息：集群内的调度与迁移、机房级的容灾切换。

- 答"先做，反正有开关能停"——开关存在不等于管用。回退动作本身也要在低半径里验过，才敢在大半径里指望它。

- 收口：半径扩大由证据推动，不由进度表推动。

> **易错点 / 考点：**高频考点是爆炸半径：定义是一次实验可能波及的用户与业务范围，控制手段是限制注入对象（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。错答多半把爆炸半径理解成故障的严重程度——它说的是影响范围，不是故障强度。

> **【思政落点 17】**：爆炸半径讲的是对风险的估计，不是对自己能力的怀疑。做工程的容易有这种自信：这套系统我熟，注进去会怎样我清楚。复杂系统的教训恰恰反过来，你不知道。所以凡是要碰真实用户的操作，先问三句：影响谁、影响多大、出问题怎么退。把试验关在划好的圈里，不是不敢试，是让敢试和负责同时成立。

> **过渡：**半径往上挪，眼睛跟不跟得上？

【7.163】优秀实践 4：混沌工程可观察性（3 分钟）

> 实验跑了四十分钟，业务指标抖一下又回来了。问为什么抖，在场没人答得上。

>

> 这就是能注不能看。混沌工程的可观察性要管三层。

>

> 第一层，实验过程留痕。什么时候开始注、注了哪些对象、参数是多少、什么时候恢复、恢复成没成功，都要有记录。为什么？因为你要把注入动作和业务曲线对上时间。没有这条线，"实验期间业务为什么抖"永远查不清。

>

> 第二层，影响要量到业务侧去。别只盯被注的那台机器。故障注在一个点上，影响顺着调用链往外传：上游的调用方、下游的被调方、最后落到用户成功率。观测得跟着这条链走。缺了这一段，注入点上的曲线好看，用户那边可能已经掉了一截。

>

> 第三层，观测得盖住实验窗口的前后。注之前要有一段基线，注之后要看有没有回到基线。只看中间那一段，你分不清"恢复了"和"本来就在抖"。

>

> 这三层不是三套系统，是同一条时间轴上的三种数据。实验日志记你做了什么，业务指标记系统怎么反应，链路追踪记影响怎么传过去。缺一样，这张图就拼不齐。

>

> 三层合起来是一条能带走的要求：每做一次实验，都要能拿出一张"注入—指标—结论"对得上的图。拿不出来，这次就只是演了一遍，不是验了一遍。

提问：实验结束，业务指标回到正常值了，能不能判实验通过？

预设回答：

- 答"是，恢复了就说明系统扛住了"——不严谨。还得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重启的。

- 答"要看注入期有没有跌破稳态阈值、恢复是不是自动完成"——对。扛住的判据在注入期，韧性的质量还要看恢复方式。

- 答"日志里没报错就行"——错。业务失败常常不写在被注机器的日志里，而是写在上游的超时和重试里。

- 收口：判据落在两处，注入期的稳态，恢复期的方式。

易错点 / 考点：常考"实验判定需要哪些观测数据"，标准答案得包含基线、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成一件事：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：**看得见之后，工具从哪儿来。

【7.164】优秀实践 5：采用开源混沌工程平台（3 分钟）

> 第一个反应是自己写一个，理由也现成：我们的场景特殊，现成工具不够用。

>

> 课件建议先看开源的。六个项目，按定位点一遍。

>

> Chaos Toolkit 走"实验即代码"。用配置文件写实验步骤，语言无关，适合塞进流水线。Litmus Chaos 面向 Kubernetes。它用 CRD 定义实验，跟 GitOps 配合得顺。Chaos Mesh 是云原生方向生态最全的，后面单独讲。chaoskube 很轻，按标签随机杀 Pod，做常态化噪声用。kube-monkey 借 Netflix 那只猴子的思路，按计划随机关 Pod。KubeInvaders 把实验做成了可视化的小游戏。课件底部还挂了一篇横向对比，container-solutions 那篇。

>

> 为什么不建议从零造？三条。

>

> 场景实现是最难的那部分。怎么在网络命名空间里加延迟，怎么让 cgroup 超限，坑最多，开源已经替你踩过了。生态和标准也一致，大家都拿 CRD、都拿 YAML 描述实验，将来换平台，实验定义能带走。成本上更明显，做一个覆盖基础资源、容器、应用层的注入平台，投入远超一个课程设计。

>

> 边界要说清：开源给的是能力，不是方法。装了 Chaos Mesh 不等于会做混沌工程。假设怎么提、稳态怎么定、半径怎么控，还得自己想。

> **提问：**开源平台都现成了，为什么不少公司最后还是自研一套？

> **预设回答：**

- 答"为了炫技"——少数情况是，通常不是主因。

- 答"因为要接自己的发布系统和监控"——说到点上了。平台的价值在和你那条流水线、那套指标体系打通，开源在这两处往往要二次开发。

- 答"因为开源的不安全"——站不住。成熟项目的注入能力本身可控，风险主要来自用的人把半径开太大。

- 收口：合理的路子是先用开源跑通方法，再按自己的流水线做集成。

> **易错点 / 考点：**考点是工具和方法的区别。错答多是"用了混沌平台就等于做了混沌工程"——平台只给注入能力，假设、稳态、半径、结论四件事仍然要人来做。考工具定位的话记三个锚点：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，教材后面重点讲它。

【7.165】ChaosBlade（4 分钟）

> 阿里内部有个系统叫 MonkeyKing，外面知道的人不多。它的开源版叫 ChaosBlade。

>

> 课件说它是在十年故障测试和演练实践上一步步演化出来的。十年、实践这两个词值钱，它意味着这工具是被真实故障磨出来的，不是设计文档里推出来的。

>

> 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。它还能指定任意类方法注进去。也就是你能精确到让某个 service 的某个方法多花 500 毫秒。C++ 应用：指定任意方法甚至某一行代码，注延迟，篡改变量和返回值。Docker 容器：杀容器，容器内的 CPU、内存、网络、磁盘、进程。云原生平台：K8s 节点上的各类资源实验，kill Pod，kill docker。

>

> 能做到这么细，靠分层实现。基础资源走操作系统接口。Java 走 Java Agent 挂载，C++ 走 GDB 调试器。容器走 Docker API 和 CRI，K8s 走 Operator。手段跟着注入点所在的层次换。

>

> 跟 Chaos Mesh 比，定位不一样。Chaos Mesh 生在 K8s。ChaosBlade 的根在单机和进程，往上再长到容器和集群。系统里要是还挂着一批没容器化的老服务，ChaosBlade 更顺手。

提问："篡改某个方法的返回值"这种注入，能验什么别的手段验不了的东西？

预设回答：

- 答"验证代码写得对不对"——太笼统。它验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"——好，这就是数据链路的容错。断电和延迟都测不出这一类。

- 答"这种注入太危险，不能用"——恰恰相反，它最便宜：在测试环境里让某个方法稳定地返回错值，看系统怎么反应。

- 收口：注入粒度越细，能验的假设就越具体。

易错点 / 考点：考点是场景分类和实现手段的对应：基础资源走系统调用与内核接口，Java 走 Java Agent，C++ 走 GDB，容器走 Docker API 与 CRI，K8s 走 Operator。错答多是把 ChaosBlade 和 Chaos Mesh 当同类替代品——一个从单机长到云原生，一个生在云原生。

过渡：它凭什么做到方法级？看组件拆分。

【7.166】ChaosBlade 关键组件（4 分钟）

八个模块，看着像八个项目。按三组记就清楚了：一个入口，一组执行器，一份模型定义。

>

入口是 chaosblade 本体。混沌实验的执行工具，CLI 和 HTTP 两种执行方式。命令覆盖创建、销毁、查询实验，还有实验环境准备与环境撤销。它还带完整的命令、场景、参数说明。准备和撤销这两个容易被略过：不少注入要前置条件，统一由它处理。

>

执行器按场景分家。exec-os 管基础资源。exec-docker 调 Docker API 标准化实现容器场景。exec-cri 走 CRI 标准化实现容器场景。chaosblade-operator 管 K8s 平台。它用 K8s 资源操作的方式创建、更新、删除实验。kubectl、client-go、chaosblade cli 三条路都能触发。exec-jvm 管 Java，靠 Java Agent 动态挂载。

> exec-cplus 管 C++，靠 GDB 做到方法和代码行级别。

>

> 模型定义是 chaosblade-spec-go，混沌实验模型的 Golang 语言定义。用 Go 写的场景能共用这一套。

>

> 这个拆法的好处值得学：入口统一，执行分离，模型共用。加一个新场景只写一个执行器，入口不用动；所有场景的参数结构一致，用起来不会一个命令一个样。

提问：Java 场景为什么要单独一个 exec-jvm，不跟 CPU、内存一样走操作系统接口？

预设回答：

- 答"Java 跑在虚拟机里，外面看不到方法"——对。JVM 内部的方法调用对操作系统不可见，要动它只能进到 JVM 里面。

- 答"Java 慢，需要专门优化"——不是理由。分开的原因是注入点不同：一个在进程外的资源，一个在进程内的字节码。

- 答"那用 GDB 也行"——技术上能挂，动字节码不如 Java Agent 干净。这正是 exec-cplus 用 GDB、exec-jvm 用 Agent 的原因：按语言挑手段。

- 收口：注入手段得配得上注入点所在的那一层。

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 对 Java Agent，exec-cplus 对 GDB，exec-docker 对 Docker API、exec-cri 对 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 资源创建、更新、删除这条路。

过渡：这些模块拼起来长什么样，下一张图给。

【7.167】ChaosBlade 构成（3 分钟）

一张图里有箭头，箭头比框更值得看。

>

> 构成分三层。最外层是调用方，三条路：命令行直接敲，HTTP 接口调用，从 K8s 那边用资源的方式下发。

> 三条路最后都收进同一个地方，实验模型。chaosblade-spec-go 那份定义干的就是这件事。不管你从哪儿发起，实验都被规整成同一套结构：场景、动作、参数。

>

> 中间是分发。模型按场景类型派给对应的执行器。基础资源走 OS，容器走 Docker 或 CRI。K8s 走 Operator，Java 走 JVM，C++ 走 C++。Operator 那支有个特别之处：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，还有它们承载的应用。同一份模型落到这四种目标上，动手的方式完全不一样。

>

> 为什么统一模型这么要紧？因为实验要能被复用、被编排、被工具读。同一个场景换个入口发起来，结构一变，所有上层的东西都得跟着改。模型在上，执行在下，扩展点就被收敛到一处：加场景只加执行器，入口和模型都不用碰。

>

> 一句判断留在这儿：这张图里被所有分支共用的那一层，就是实验模型。

> **提问：**明天要给 ChaosBlade 加一个 Python 应用方法级注入，按这个构成，动哪一层？

> **预设回答：**

- 答"改入口，加一条命令"——不用。入口统一，命令按模型自动生成。

- 答"加一个 Python 执行器"——对。模型里描述哪个模块、哪个函数、注什么，执行器负责把动作挂上去。

- 答"改模型定义"——只有现有模型表达不了这种场景才需要。方法级注入用现有模型基本能描述。

- 收口：分层的好处就是这个，扩展点被收到一个地方。

> **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。错答多是把 CLI 和 HTTP 当成两套实现——它们是同一个入口的两种调用方式。判断方法还是那句：看图上唯一被所有分支共用的那一层。

> **过渡：**结构清楚了，跑一个最小的例子。

【7.168】一个简单的演示（4 分钟）

敲第一行命令之前，先想清一件事：这条命令改的是哪台机器上的什么。

>

> 创建实验。CPU 场景的命令是 blade create cpu load，参数 --cpu-percent 80。它把负载压到八成。网络场景是 blade create network delay。参数三个：--time 3000、--interface eth0，还有 --local-port 8080。意思是在这块网卡的这个端口上注三秒延迟。两条命令的共同点：只说要什么效果，不说怎么做，落到系统调用上那一步是执行器的事。

>

> 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，最容易被忽略。演示完忘了记，实验就一直挂着。第二天它还在压你的 CPU。

>

> 然后是查询，blade status 之类，确认实验还在不在、注了多久、参数对不对。最后销毁，blade destroy 加 UID。有些场景注之前要 prepare，撤的时候 revoke。

>

> 演示不要求你背命令，参数以官方文档为准。课件底部给了 chaosblade-io.gitbook.io 那个地址。要留下的是四拍节奏：创建、记录、观察、销毁。真实演练还得加两条：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。

提问：演示时建了实验忘了销毁，第二天业务出现莫名其妙的抖动，你先查什么？

> **预设回答：**

- 答"先查最近有没有发布"——常规思路，能白查半天。

- 答"先查有没有残留的混沌实验"——对。blade status 一敲就知道，演练环境里最常见的"人为故障"就是这个。

- 答"重启机器就好"——能好，可你把证据也一起销了，下次还会再犯。

- 收口：演练环境的第一排查项，永远是还有没有人在注故障。

> **易错点 / 考点：**考点是实验生命周期：创建、查询、销毁，外加环境准备与环境撤销这一对。判断题"实验执行完会自动恢复"——错，多数注入要显式销毁。答题要点：任何注入动作都得配一个回退动作，而且回退动作要在注入前确认过能用。

> **过渡：**单机玩明白了，系统跑到云上又会怎样。

【7.169】云原生架构下混沌工程引入（4 分钟）

> 同一套代码搬到 K8s 上，故障的样子就变了。

>

> 原因是责任被挪走了。以前实例挂了你自己兜，现在平台替你兜。兜不兜得住，得一次一次注进去才知道。课件拿四组对照说明，每注一次，同时验四项能力。

>

> 第一项，单点故障容忍。注单实例故障，业务 SLA 没有波动，业务不受影响。前提是单实例：云原生默认实例会挂，架构上就得设计成一个挂了不影响整体。

>

> 第二项，状态可感知。注进故障之后，PaaS 能及时感知实例异常，异常实例自动迁移成功，业务指标不受影响。这一项考平台的感知，不是你的代码。

>

> 第三项，通过 naming service 访问。实例迁移之后列表变了，上游对这个服务的访问不受影响。这一项考服务发现：上游要是把地址写死，迁移等于断链。

>

> 第四项，部署满足单一包描述。注故障引发了扩容，PaaS 自动执行，不需要人做前置或后置。这一项考声明式部署，你写要几个副本，平台自己补齐。

>

> 四项合起来是一句话：验证对象从应用代码扩到了平台能力。调度、迁移、服务发现、弹性伸缩都得验。于是需要一把生在 K8s 上的工具。Chaos Mesh 就是接着往下讲的。

提问：在 K8s 上注了 Pod 故障，Pod 被杀、新 Pod 起来、业务也没事，这个实验说明什么，又说明不了什么？

预设回答：

- 答"说明高可用做到了"——说对一半。它说明副本冗余和自动重建生效，不代表服务发现、连接池、会话状态都没问题。

- 答"还得看上游有没有报错、有没有长连接断连"——很到位。Pod 重建过程中，上游的连接池和注册中心缓存是典型薄弱点。

- 答"说明平台没问题，代码也没问题"——推过头了。这次只覆盖杀 Pod 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：考点是云原生下验证对象的扩展：从应用自身扩到调度、迁移、服务发现、弹性伸缩四项平台能力。错答多是"云原生平台自带高可用，不需要混沌工程"——恰恰相反，平台能力一样会失效，而且失效时更难察觉。

过渡：于是要一把生在 K8s 上的工具。

【7.170】Chaos Mesh（3 分钟）

2019 年 12 月 31 日，GitHub 上多了一个仓库。

>

> 往前追一步，它最早是 TiDB 团队 2019 年做的一个测试框架。名字叫 Schrödinger's Cat，薛定谔的猫。往后走，2020 年 2 月 6 日进 CNCF 云原生全景图。7 月 14 日成为 CNCF 沙箱项目，9 月 26 日 1.0 GA。2021 年 7 月 23 日 2.0 GA，11 月 30 日 2.1 GA。2022 年 2 月成为孵化项目。

>

> 补个背景。CNCF 按鸿沟理论管项目，分三级。沙箱 Sandbox、孵化 Incubating、毕业 Graduated，一级比一级稳。这不是荣誉称号，它反映的是采用广度和治理成熟度。看一个云原生工具靠不靠谱，先看它站在

> 哪一级，挺实用。

>

> 这条线还读出两件事。一是它从测试框架长成了平台。起点是给一个数据库做测试，现在覆盖网络、IO、时间、内核、压力各类故障。二是迭代节奏快，不到两年从 1.0 走到 2.1。

>

> 还有一点藏在出身里。起点是 TiDB，一个分布式有状态系统，对一致性、时钟、IO 特别敏感。所以它的故障类型里才有 TimeChaos、IOChaos 这种细分类，不是拍脑袋加的。

> **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"更成熟了，可以用了"——方向对，理由得说准：升级看采用者数量、社区治理、发布规范这些指标，不只看代码质量。

- 答"有更多公司在生产环境用了"——这是核心指标之一。生产采用在分级评估里权重很高。

- 答"沙箱就是玩具，别碰"——过头了。沙箱意味着早期、接口可能变，不等于不能用，看你愿不愿承担接口变动的成本。

- 收口：分级是一块风险提示牌，不是使用许可证。

> **易错点 / 考点：**常考 CNCF 三级分级的名称与顺序：Sandbox、Incubating、Graduated。易错点是把 Chaos Mesh 的出身记成某云厂商的产品——它来自 TiDB 团队的测试框架。考时间就记三个锚点：2019 开源，2020 沙箱，2022 孵化。

> **过渡：**它怎么表达一个实验？靠声明式。

【7.171】声明式 API 定义简化混沌实验管理（4 分钟）

> 写 YAML，还是敲一条命令？两种都能把同一个故障注进去。

>

> 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 选了后者。

> 你写一份 YAML，说清对哪批 Pod 注什么故障、注多久。控制器负责把它变成现实。

>

> 落地手段是 CRD，用自定义资源扩展 Kubernetes 的 API。课件列了六类。PodChaos 杀 Pod，或者让 Pod 不可用。NetworkChaos 做延迟、丢包、分区。IOChaos 做文件读写延迟和错误，TimeChaos 做时钟偏移。StressChaos 压 CPU 和内存，KernelChaos 做内核级故障。类型还在加。

>

> 为什么这样能简化管理？三点。实验成了普通资源。创建、查看、删除都用 kubectl，跟你管 Deployment 一套手感。实验能进版本库，YAML 可以提交、可以 review、可以追溯谁在什么时候改过参数。天然可自动化。任何能操作 K8s 的东西都能操作实验：流水线、Operator、GitOps 工具都算。

>

> 反过来看命令行：敲完就过去了，没留痕，换个人重现不了。

>

> 一句话：声明式把实验从一次操作变成了一份资产。

> **提问：**Chaos Mesh 为什么选"给 K8s 加一种资源类型"，而不是"发一个命令行工具"？

> **预设回答：**

- 答"因为 K8s 本来就是声明式的，顺着生态走"——准确。控制器模式、权限体系、审计日志都是现成的，等于白捡一整套基础设施。

- 答"因为 YAML 好写"——表层的理由。好写只是副作用，核心是状态能被持续调谐：实验被删了、节点重启了，控制器还能把状态拉回来。

- 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：选声明式，选的是接进生态，不是多一种写法。

> **易错点 / 考点：**考点是声明式与命令式的辨析，以及 CRD 的作用。错答多是把 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 这种查询才成立。考六类资源，记 Pod、Network、IO、Time、Stress、Kernel 六个前缀就够。

> **过渡：**有了资源类型，还得有人读它、有人执行它。

【7.172】Chaos Mesh 架构（4 分钟）

有人问过：这些执行器为什么不集中部署成一个大服务？

>

> 先看四个组件各管什么。Chaos Dashboard 给人用，UI 上管理和监控实验。它的作用是降门槛。写 YAML 对熟手轻松，对第一次上手的人就是一道坎。界面能点选类型、选目标、看状态。

>

> Chaos Controller Manager 管决策，做调度和控制，同时是编排引擎。前面那些声明式资源就是它读的，它盯着创建、更新、删除，然后决定现在动手还是先等着。

>

> Chaos Daemon 是 K8s 环境里的执行组件，真动刀子的活它干。要侵入命名空间、改 cgroup、动网络规则，这些动作要节点级高权限。所以它以 DaemonSet 的形式跑在每个节点上。

>

> Chaosd 是 non-K8s 环境的执行组件，给还没上 K8s 的机器用。虚拟机、物理机上也能拿同一套方法做实验。

>

> 连起来是一条分工线。Dashboard 管体验，Controller Manager 管决策。Daemon 和 Chaosd 管执行，各管一段。这也是云原生控制器类系统的标准骨架：期望状态、决策、实际动作。以后看 Operator、看别的 K8s 工具，大半是这个结构。

提问：为什么执行组件要跑在每个节点上，不集中部署成一个服务？

> **预设回答：**

- 答"它要动本机的网络和 cgroup，必须在本机"——对。网络命名空间、cgroup、文件系统都是节点本地的，跨节点操作不到。

- 答"为了分摊压力"——不是主因。真正的理由是权限和位置：注入要节点级权限，也只能在节点上生效。

- 答"少开端口更安全"——有点道理，主要还是能力和位置决定的。

- 收口：执行组件放在哪儿，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：Dashboard 管界面，Controller Manager 管决策与编排，Daemon 管 K8s 执行，Chaosd 管非 K8s 执行。最常见的错答是把 Chaos Daemon 和 Chaosd 混成一个——一个在集群里，一个在集群外。答题先问它跑在哪儿，再问它负责决策还是执行。

> **过渡：**控制器具体在盯什么？

【7.173】监听资源变化（3 分钟）

> 控制器一直在做三件事，顺序还是固定的。

>

> 第一件，监听。盯 PodChaos、NetworkChaos 这些资源的创建、更新、删除。它不是隔一会儿去问一遍现在要做什么，而是一直看着这些对象。轮询的代价是延迟和空转，监听拿到的是事件本身。你提交一份 PodChaos，它立刻知道；你把 duration 改短，它也知道；你删掉，它同样知道。

>

> 第二件，判断。收到变化之后要认出此刻的意图：新建的资源是要开始注；被删掉或者到期是要恢复；资源还在但条件没满足，比如目标 Pod 还没就绪，那就先等着。等着也是一种决策，这一步最容易被漏。不少自动化系统出事，就出在还没到动手的时候它动手了。

>

> 第三件，执行。简单注入像 PodKill，它自己就能干；要动节点上的东西，就向 Chaos Daemon 或者 Sidecar 发请求，由它们落地。

>

> 三件合起来是一个循环：监听、判断、执行。循环带来的是自我纠偏。实验资源被误删了，恢复动作会自动跑。节点重启了，控制器再看到资源还在，会把状态重新拉回来。声明式的好处就落在这儿：你描述状态，系统守着它。这套结构里没有"实验列表"这种东西，只有资源对象和它们的变化。

> **提问：**判断出"先等着"和判断出"要注"，哪一个更容易出错？

> **预设回答：**

- 答"注入更容易错，因为动作危险"——危险是事实，可等着判错了后果更隐蔽：条件没满足就注，实验结论全是假的。

- 答"等着更容易错，因为条件多"——很到位。目标在不在、就绪没有、上一个实验恢复完没有，任一条判错，都会误注或漏注。

- 答"两个一样"——不一样。注入错了是多造一个本来没有的故障，等着错了是本来要测的没测到，后者不报错，最难发现。

- 收口：自动化的风险常常不是做错，是悄悄没做。

> **易错点 / 考点：**考点是控制器的三种决策状态：注入、恢复、等待，以及简单注入与 Daemon 执行的区分。错答多是只记注入和恢复两种——等待这一项才是理解声明式系统的关键。判断方法：把实验当成期望状态，控制器做的就是让实际状态追上它，追不上就先等。

> **过渡：**分发出去以后，故障怎么进到容器里？

【7.174】故障如何注入？（4 分钟）

> 故障是注进去的，可注到哪儿去了？

>

> 前一步还是监听资源变化。接着要问：容器到底是个什么东西。它不是黑盒，它由三样构成：进程、Namespace、Cgroup。进程是真正在跑的那部分。Namespace 管可见性，决定这个进程能看到哪些网络、哪些挂载点、哪些别的进程。Cgroup 管资源，决定它能用多少 CPU、多少内存、多少 IO。这三样摊开，一个容器在系统层面就没有第四样东西了。

>

> 所以注入的实质是两件事：侵入 Namespace 和 Cgroup，然后干扰它。

>

> 具体到动作上：要让网络变慢，就去动它的网络命名空间，加一条 tc 规则。要让磁盘变慢，就动挂载和 IO 限额。要让 CPU 不够用，就动 cgroup 配额，或者直接起一堆抢占进程。要让它找不到别的服务，就在网络命名空间里改路由或者 DNS。

>

> 于是判断一个故障类型能不能做，有了一条通用路子。靠改可见性或者改资源限额能表达出来的，都能注。表达不出来的，得走应用层手段，比如 Java Agent 改方法、GDB 改返回值。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，没权限碰不到。

提问："让容器内的 DNS 解析失败"这个故障，按这层框架，是动 Namespace 还是动 Cgroup?

预设回答：

- 答"动 Cgroup，因为是资源问题"——错。DNS 解析是找不找得到、通不通，属可见性一侧。

- 答"动 Namespace，改它的 DNS 配置或网络命名空间里的规则"——对。这一类"看得见看不见"的故障都归可见性。

- 答"改应用配置也能实现"——能，但那是应用层手段，不是平台级注入，适用场景不一样。

- 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。这一节最容易被跳过，也最能分出"会用工具"和"懂原理"。错答多是把注入理解成发一个特殊请求给应用。答题三步：这个故障改了什么，归可见性还是资源，对应动哪个内核机制。

过渡：原理清楚了，看一份真实的实验定义。

【7.175】使用 Chaos Mesh 定义混沌实验（4 分钟）

这份 YAML 只有十几行，五个问题都装在里头：什么类型、叫什么、放哪个空间、怎么注、注谁。

>

> apiVersion 是 chaos-mesh.org/v1alpha1，接口版本。kind 是 PodChaos，说明这是 Pod 级故障。

>

> metadata 两个字段。name 是实验名，起名要看得出意图，别写 test1。namespace 是实验资源所在的空间，不是被注 Pod 的空间，这两个容易混。

>

> spec 是核心，四个字段。action 是 pod-failure，让 Pod 失败，一段时间内不可用，而不是删掉它。差别在哪？删掉会被重建，失败是原地不动但不可用，更接近真实的部分故障。mode 是 one，从目标里挑一个。这一行就是爆炸半径的第一个旋钮。改成 all 是全选，风险上一个量级。duration 是 30s，到点自动恢复，这是第二个旋钮。

>

> 最后是 selector，用 labelSelectors 挑目标。标签写的是 app.kubernetes.io/component 等于 tikv。按标签选，不写 Pod 名字。Pod 名字每次重建都变，标签是稳的。

>

> 两个旋钮记住就够用：影响几个，影响多久。

> **提问：**把 action 从 pod-failure 改成删除 Pod，实验验的能力差在哪？

> **预设回答：**

- 答"没区别，反正 Pod 都没了"——错。failure 是对象还在但不可用，删除是对象消失再被重建，两条失效路径。

- 答"删除会触发重建，验的是自愈；failure 验的是故障隔离"——准确。两种失效模式对应两套机制。

- 答"那就都注一遍"——对，这也说明动作本身是实验设计的一部分，不是随手挑的。

- 收口：选动作，等于选你要验的那条恢复路径。

> **易错点 / 考点：**考点是 YAML 各字段的含义，以及爆炸半径的两个旋钮：mode 和 duration。错答多是以为 mode: one 表示注一次——它表示选中一个目标，注多久由 duration 管。另一个易错点是 namespace：那是实验资源的空间，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，事故往往几个一起来。

【7.176】编排混沌实验（4 分钟）

> 一次事故很少只由一个故障引起。

>

> 常见顺序是这样：先网络抖，重试把下游打满，线程池耗尽，链路雪崩。只测单点，测不出这种连锁。所以要编排，Workflow 干的就是这件事。

>

> 构成四块。名称。Entry 是入口，引用任意一个声明过的 Template。是引用不是复制，同一个模板能多处用。Template 数组，定义各个行为。五种模板类型：Serial 串行，Parallel 并行，Chaos 具体实验。还有 Suspend 挂起，Task 任务。Serial、Parallel、Task 能把别的节点当子节点，方式是引用。于是整个 Workflow 成了一张有向图。

>

> 两个要点。第一，入口唯一：从 entry 能走到每个节点，执行顺序才定得下来，排查卡在哪一步才有依据。第二，用引用不用嵌套：定义一次多处引用，改一处全改，几十个实验也好维护。

>

> 骨架就两行。第一行 entry，指向某个模板名。第二行 templates，逐个列 name 和 templateType。入口加模板清单，剩下交给控制器按图推。

>

> Suspend 看着什么都没干，却是编排里最管用的一环。注进去之后要留观察窗口，等指标把影响显出来，再决定加大还是收手。

提问：编排里为什么要专门给一个 Suspend，什么都不做？

预设回答：

- 答"用来做延时，模拟慢"——不是。延时是故障的一种，走 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"——对。故障注进去到业务恶化有滞后，没这个窗口，你会在还没显现代价的时候就把实验停掉。

- 答"用来卡人工确认，点一下才继续"——也有这种用法，把审批卡在流程里。

- 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，以及有向图加唯一入口的结构。错答多是把 Task 当成一个混沌实验

——混沌实验是 Chaos，Task 一般用来做检查或者调外部动作。记五个字：串、并、混、挂、任。

过渡：故障摆好了，用户那边看到的是一连串麻烦。

【7.177】用户场景演练（3 分钟）

所有服务的成功率都是 99.9%，用户还是说下单老失败。

>

系统可以做到每个组件都没挂、每个接口成功率都好看，而用户完不成一件事。可能一个不关键的推荐服务在拖主流程。也可能一个校验接口超时后静默返回了空。用户看到的只有"提交失败，请稍后再试"。组件的健康和业务的可用，中间隔着一条挺宽的缝。为什么按旅程看反而更难？因为旅程跨好几个服务，指标得能把它们串成一条。手上只有按服务分的仪表盘，这条旅程的完成率你得自己算。

>

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。每一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，让它们有先有后，模拟一件事还没缓过来、第二件又来了。

>

这时候看的不是某个服务的错误率，是这条旅程的整体完成率、每一步的耗时、失败时用户看到什么。

>

关键词还是编排。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，挂起留出观察窗口。

>

两句话分清楚：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办不办得成。前者问的是系统，后者问的是人。

提问：所有服务成功率都 99.9%，用户却反馈下单老是失败，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。校验、风控、推荐这类环节一挂，主流程可能被直接拦住。

- 答"重试把延迟放大了"——也对。单次调用没超阈值，重试叠起来整体就超时，这是典型连锁。

- 收口：所以按旅程看指标，别按服务看指标。

> **易错点 / 考点：**考点是组件可用性与业务可用性的区别，以及场景演练要编排能力。错答多是把场景演练等同于多注几个故障——差别在是不是按用户旅程组织、是不是看端到端指标。三问可用：这条链路用户能不能感知？断了用户看到什么？有没有替代路径？

> **过渡：**场景演练一般在上线前或者演练窗口做，能不能更早？

【7.178】Pull Request 场景（3 分钟）

> 有人提了个 PR，改两行：把一个超时从 2 秒改成 30 秒。编译过，单测过，功能测试也过，因为正常路径没变。

>

> 这就是韧性的特点，它会被人改坏，而且改坏的时候不报警。今天这个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。有人图快把它删了，理由还挺硬："反正没触发过。"只有注一次故障，才知道韧性还在不在。

>

> 于是有了这个用法。PR 提上来，自动部署到预览环境。就是前面说的预览环境那股特定流量，在那儿跑一组小而快的实验。半径压到这一个 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：下游超时兜没兜住，依赖不可用有没有降级，重试会不会被放大成风暴。过了才允许合并。

>

> 两条工程约束得说清。实验要快，门禁跑二十分钟，开发就绕过去了，所以只跑必过集，不跑全量。判据还得稳，门禁不接受"看起来还行"，指标越线就得红。噪声大的实验放进门禁，很快会被"重跑一次"废掉。

>

> 这条路子还有个前提。预览环境跟生产得说得上话，配置、依赖版本、数据规模都不能差太远。差太远，注

> 出来的结论带不走。

>

> 变化就在这儿：混沌工程从季度演练，变成了每次提交都要过的一关。

> **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"——真实阻力之一。所以一般只在预览环境跑，而且只注一个实例。

- 答"太慢，开发等不了"——最普遍的阻力。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，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）。错答多是把两者混成一件事，要求门禁也做全链路验证——流水线会彻底堵死。判断先问一句：这次是想发现新风险，还是防止已有韧性退化。

> **【思政落点 18】**：韧性是别人写的代码在替你兜底，而兜底的东西最容易被人图省事拆掉。把故障注入挂到每次提交上，等于把"我记得这儿还有降级"变成一个别人能查的结论。对自己那两行改动负责，具体动作就是改完先想办法把它弄坏一次，再交出去。肯让别人拿故障来试自己的代码，比嘴上说质量可靠实在得多。

> **过渡：**工具和实践都齐了，国内有团队把它做成了一套体系。

【7.179】从故障注入到混沌工程 —— KeChaos 2.0（4 分钟）

> 标题前后各是一件事，中间隔着的就是这一节。

>

> 故障注入是动作，混沌工程是方法加流程加度量。KeChaos 2.0 把这套体系画成了五步。

>

> 第一步，定义实验场景。先想清楚要验哪个业务场景，不是先想我能注什么。第二步，关联注入能力。场景定了，再看手上哪些工具能实现，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，这一步得在注入之前做完。第四步，实验运行加爆炸半径控制。跑起来，同时把影响摁住。第五步，实验终止产出报告。结束，把结论沉下来。

>

> 三个细节值得点出来。关联注入能力排在第二步，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，说明控半径不是跑之前设一次，是跑的过程中一直要做。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，稳态标准。它跟稳态指标是一回事，多"标准"两个字，意思是阈值得事先约定，不能事后看着曲线说大概还行。

>

> 五步走完，一份实验才算闭环。场景、能力、标准、运行、报告，五样齐全。换成工具清单，就只剩中间那一样了。

提问：五步里"关联注入能力"排在"确定稳态标准"前面。把它挪到最后，会出什么后果？

预设回答：

- 答"没什么区别，都是准备"——有区别。顺序一错，你先被工具能做什么框住，场景设计不自觉往"工具好实现"那边偏。

- 答"会变成工具驱动，场景被工具裁剪"——对。很多团队做了不少实验，却回答不了业务关心的问题，根子在这。

- 答"稳态标准会定得不合适"——也有这一层：标准得跟着业务场景走，别跟着能注的故障走。

- 收口：流程顺序本身就是方法，它决定你从问题出发还是从工具出发。

易错点 / 考点：考点是五步的名称与顺序，以及故障注入和混沌工程的区别。错答多是把两者当同义词——故障注入是技术手段，混沌工程是把假设、稳态、半径控制、报告沉淀都包进去的完整体系。记顺序：场景、能力、标准、运行控半径、报告。

过渡：这套流程落到具体目标上，第一块阵地是基础服务。

【7.180】基础服务稳定性风险治理（4 分钟）

治理这两个字听着大，落到动作上就是五步。

>

> 先看演进。KeChaos 0.1 的定位是核心基础服务的稳定性风险探索。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。常态化、例行化这两个词重要，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。两代之间差的不只是功能，是这件事在团队里的位置：一边是临时立项，一边是排进排期表。

>

> 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。这五步不是直线，是个圈。最后一步验完，发现新的薄弱点，又回到定义实验。

>

> 两个点要强调。做出假设被单列成一步，说明它得显式写下来，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。问题改了，没人回头用同一个实验复验。于是"修好了"只是感觉。

>

> 为什么先挑基础服务？注册中心、配置中心、缓存、消息、数据库代理这一类，挂一个就是一大片。对它们做混沌工程最划算：影响面大，故障模式相对稳定，验证方法容易复用。

>

> 一句话：先探索找风险，再把验证变成例行。

提问：闭环里为什么要把"实验验证"放回圈里，不是修完就结束？

> **预设回答：**

- 答"因为修复可能没修对"——对，最常见的情况就是这样：改的是现象，根因还在。

- 答"因为修复可能引入新问题"——也真实。改超时、加降级、换依赖，都可能改到别的链路。

- 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 探索是找风险，1.0 常态化与例行化是守风险。错答多是把"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。另一考点是闭环为循环，修复之后必须复验。

> **过渡：**闭环讲的是怎么管，下一页讲具体怎么注。

【7.181】基于 KeChaos 的故障注入（4 分钟）

> 故障注入这四个字，很多人脑子里只有一个动作：把某个服务打死，看它崩不崩。真上手会发现，注之前一堆功课，注之后还有一半活没人干。

>

> 课件把这件事拆成四个阶段，每个阶段都有具体动作。第一个阶段，依赖梳理。四个动作。链路分析，看请求经过哪些服务。强弱依赖梳理，哪些依赖断了业务就停，哪些只是体验下降。依赖降级梳理，每个依赖有没有降级方案。最后是熔断限流降级的能力梳理，阈值在哪、上限多少。

>

> 这一步一次故障都没注。为什么先做？你不知道调用关系，就算不出会波及谁。半径控不住，实验就不敢在生产上跑。

>

> 第二个阶段才是注入。五个动作。先报备周知，告诉相关方什么时候注什么。再照历史故障画像去注，过去真出过的问题命中率最高。范围上先专注自身服务，别一上来就动下游。然后模拟下游异常、模拟依赖异常。

>

> 顺序有讲究。报备在前，出事才有人配合。画像在前，注的是真出过的问题，不是凭想象编的。专注自身在前，先证明自己扛得住，再去碰别人的链路。

>

> 第三个阶段，报警验证。三个指标。报警触达率，收到的人收没收到。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。为什么单占一个阶段？因为告警链路自己也会坏。规则过期、通道失效、值班表半年没更新，平时都看不出来。注一次故障，是给告警体系做体检最便宜的办法。

>

> 第四个阶段，降级验证。三项：降级能力建设，有没有这个东西；限流有效性，限得住吗；熔断、降级有效性，真会跳闸吗。写在文档里的降级方案，和按下去会动的开关，是两件事。

>

> 四个阶段连起来，逻辑只有一句：先弄清结构，再小心注入，然后验看得见和扛得住。后两个阶段验的是运维能力，不是业务代码。这一段最常被漏掉，因为它不在代码里。

提问：为什么报警验证要单独占一个阶段，而不是注入完顺带看一眼监控有没有报？

预设回答：

- 答"顺带看就行，反正会报"——最常见的想法，也最容易漏。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，和降级验证的三项。常见错答是把故障注入当成整个流程，它只是第二阶段，前后都还有活。答题要点：先梳理再注入，注入后既验业务，也验告警和降级。

过渡：混沌工程这一块走完了，接下来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）

有个功能上线两周，数据看着别扭。点进来的人不少，走完流程的不到三成，客服那边也没人投诉。

>

> 这种情况查用例是白查的。用例写的是点提交、订单创建成功，它全绿。可它从来不问一句：用户知不知道该点哪里，看不看得懂那句提示。

>

> 7.5 节换的就是这个问题，好不好用。它值得单独成节，因为能用和愿意用是两件事。能登录、能下单、能支付，功能全对。可按钮藏在第二屏，错误提示是一串错误码，一个流程要点五次。人就走了。

>

> 易用性测试回答三层问题。看得懂吗，界面元素合不合用户的习惯。走得通吗，一个任务能不能从头做到尾。愿意继续用吗，做完之后他下次还来不来。三层对应三种方法和三种度量，后面几页分开说。

>

> 这一节收尾的地方是一个很工程化的手段，A/B 测试。把用户分成组，同时在线跑不同版本，用真实行为判断哪个更好。易用性本来是主观的，A/B 给它装了一套能比的判断依据。你觉得好不算，看数据。

>

> 还有一句边界先摆清。功能测试验通不通，易用性测试验愿不愿。两件事都得做，但别拿一边的结果去替另一边结案。

>

> 真要说分工，一句话够。前面问的是系统扛不扛得住，从这里开始问的是人愿不愿意用。

> **提问：**一个功能所有用例都通过，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，但性能问题通常表现为慢和超时，不是放弃。

- 答"易用性问题"——对。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根本不用这个功能；用了一半放弃，更像易用性。

- 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，验的是使用过程中的效率、可理解性和满意度，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，还包括一致性、效率、容错提示。

> **过渡：**既然界面起的作用最大，那就先把好的界面到底好在哪说清。

【7.183】良好的 UI 元素（4 分钟）

> 两个按钮，一个写保存，一个写提交，功能都对，都能把数据存下来。用户在两页之间来回看，最后还是点错了。

>

> 好的用户界面七要素，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。按性质分三组，好记。

>

> 第一组守规矩。符合标准和规范，是跟平台和行业的通用约定走：手机上的返回手势、菜单放哪、必填项怎么标。一致性，是产品内部同类操作的表达要一样。这页写保存，那页就写提交，按钮颜色、提示语气、错误码格式都得统一。

>

> 一致性为什么重要？用户是靠上次的经验来用你的软件的。每次都要重新学一遍，多出来的那几秒就是摩擦。刚才那两个按钮的损失就在这儿：用户得停下来判断，这两个词是不是一个意思。

>

> 第二组好理解。直观性，看一眼就知道这是干什么的，点了会怎样。正确性有两层。界面上的信息必须对，数字、状态、文案跟实际一致。操作的结果也要可预期，点了删除就真删了，还得有确认和撤销。

>

> 第三组好用。灵活性是对不同的人给不同的路：新手有引导，熟手有快捷键，批量操作有入口。舒适性是视觉和操作的负担要低：对比度够、点击区够大、不刺眼、不让人紧张。实用性是每个元素都得有存在的理由，不做看起来高级但没人用的东西。

>

> 七条摆在一起，作用是把审美问题变成可以检查的问题。做易用性测试时，拿这七条当一张单子，一条条问我们做到没有。这比凭感觉说我觉得不丑靠谱。

>

> 其中两条最容易被当成小事：一致性和实用性。它们的改动量最小，收益却算得出来。

> **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，会造成什么损失？

> **预设回答：**

- 答"违反直观性"——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，属一致性。

- 答"违反一致性，用户会犹豫、会点错"——对。措辞不统一，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"——这就是把摩擦当理所当然。摩擦攒起来就是放弃率。

- 收口：一致性问题看着最像小事，却是最容易算出收益的一类改进。

> **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。另一个错答是把灵活性当成功能多，灵活性强调面向不同熟练度用户的多条路径。

> **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）

> 手上还没有成品，你却要回答这个方向值不值得做。这时候拉几个用户问好不好用，问出来的多半是客套话。

>

> 用户体验测试有三种方法，区别不在难度，在先用在哪个阶段、回答哪个问题。

>

> 第一种，探索式测试。它能确定新产品该包含哪些内容和功能，也能评估初步设计或原型的有效性和易用性。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、这个方向对不对搞清楚。所以它常跟纸质草图、低保真原型一起用，成本极低，推翻重来也不心疼。

>

> 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，并且体验良好。它用在有成品的时候，回答这一版能不能上、上了会不会被骂。发布前做，是可用性测试；发布后做，就看真实用户的实际表现。

>

> 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，确定哪种能给最好的操作体验。它用在有几个候选方案的时候，回答选 A 还是选 B。产出不只是哪个赢，还要说清为什么赢、在哪个维度赢。否则下次设计还是拍脑袋。

>

> 三种连起来是一条时间线：探索，还不知道做什么；评估，做出来了能不能上；比较，两个方案选哪个。

>

> 最容易出错的情形反而不在选方法，是已经有了成品，还在用探索式的问法问你觉得怎么样。这种问法拿到的是偏好，不是任务能不能完成。上线之后的优势恰好是有真实行为可看，把它浪费在问感受上，太亏。

>

> 还有一条界线。比较性测试不等于 A/B 测试。A/B 是在线、真实流量、看行为数据；比较性测试范围更宽，实验室里几个用户也能做。

> **提问：**产品已经上线，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，多听用户总没错"——方向不错，方法错了阶段。上线后该问真实使用中哪里卡住。

- 答"方法用错了阶段，该做评估性或比较性测试"——对。上线后的优势正是有真实行为数据可用。

- 答"直接问用户喜欢哪个颜色"——这是比较性测试最浅的一层，只问偏好不问任务完成情况，结论很弱。

- 收口：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 是在线、真实流量、看行为数据，比较性测试范围更宽。另一考点：探索式测试强调早期与低成本。

> **过渡：**测完得落到指标上。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）

> 改版上线，老板要一句用数据证明体验变好了。报表打开，第一个问题就卡住：看哪个数。

>

> 三套模型，层次不同。先看谷歌的 GSM。从产品或功能的目标出发，分析这些目标会转化成哪些信号，再据信号建立具体、适用的度量指标。目标、信号、指标，三层。

>

> 举个具体的。目标是让用户快点找到商品。信号是搜索到点击的间隔、首次点击的命中率。指标落成搜索到点击的中位耗时，一个能算的数。它的价值在中间那一层。很多人从目标直接跳到指标，落出来的数跟目标没关系，跑半年也不知道在证明什么。

>

> 第二套是传统网站 UX 指标 PULSE：页面浏览量、在线运行时间、延迟、周活用户数、收益。五个都好采集、好汇报，适合做整体看板。短板也明显：这些数讲的是网站怎么样，不是用户感受怎么样。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，以用户为中心：愉悦度、参与度、接受度、留存率、任务完成度。

>

> 三者的关系记成一句。GSM 是方法，从目标推指标。PULSE 是看板，看系统和运营。HEART 是框架，站在用户那边。

>

> 所以选指标之前先回答一个问题：这次要回答的是系统问题还是用户问题。答完，用哪套自然就定了。

>

> 老板那句用数据证明体验变好，最贴的其实是 HEART 里的任务完成度和留存率。前者说用户能不能办成事，后者说他愿不愿意回来。PULSE 可以一起报，但它证明不了体验。

提问：老板要用数据证明改版让体验变好了，你从 PULSE 挑指标，还是从 HEART 挑？

> **预设回答：**

- 答"用 PULSE，数据现成"——省事，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，尤其任务完成度和留存率"——更贴体验这个词。前者说能不能办成事，后者说愿不愿回来。

- 答"两个都用，PULSE 看整体，HEART 看用户"——最完整。前提是先用 GSM 想清这次目标是什么、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。另一个错答是把 PULSE 当用户体验模型，它更偏系统与运营。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，可它们都要跑一段时间。有没有几天内就分出新旧版本谁更好的办法？

【7.186】A/B 测试（3 分钟）

> 按钮放左边还是右边，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，谁也说服不了谁，最后谁职级高听谁的。上线之后数据不好，也说不清是方案错了还是运气不好。

>

> 课件第一句是句英文：有些东西需要 Sense，但大部分东西可以用 Science 来判断。翻过来就是：产品上确实有一部分靠直觉和品味，但大部分决定其实能靠数据判。A/B 测试就是把靠品味的那部分，尽量挪到靠证据这边。

>

> 定义要给全。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。判断依据是用户反馈的真实数据，看哪个方案更好。

>

> 三句话拆开，每一句都在堵一个漏洞。分组，不同组看到不同版本，否则无从比较。同时在线，不是这个月用 A、下个月用 B，而是同一时间两拨人各看一个。季节、大促、竞品动作对两组的影响是同一份，能抵消掉。真实数据，判断依据是用户真点了什么，不是会议室里谁的嗓门大。

>

> 生活里的类比是临床试验。新药不能医生觉得有效就上市，得设对照组、同期观察、看统计结果。软件比药幸运，同一秒钟就能给一半人旧版、一半人新版。

>

> 所以有一句要分清：A/B 测试不是测 bug 的。功能测试问对不对，A/B 问哪个更好。它测的是方案优劣，两个版本都能正常跑，只是一个更值。

提问：注册按钮，产品经理要改绿，设计师要改红，下周上线，两天内怎么给一个两人都服的答案？

预设回答：

- 答"做用户调研，问用户喜欢哪个"——偏好不等于行为。用户说喜欢绿色，点的时候可能还是点红色那个。

- 答"先上绿色一周，再上红色一周比两段数据"——最典型的错答。两周外部环境不同，差异归不了因。

- 答"各放一半用户，看点击率谁高"——方向对，还要补两点：分流要随机且稳定，同一用户每次看到同一版；差异要看统计显著，百分之一的波动可能只是噪声。

- 收口：第三种已经站在正确答案门口，后面讲特点、讲分流、讲评价，就是把这三层补严。

易错点 / 考点：三个必考区分。A/B 不等于灰度发布，灰度是逐步放量降风险，目的是稳；A/B 是对照比较，目的是选。A/B 不等于埋点分析，埋点只采数据，没有分组对照就没有因果推断。判断题高频陷阱：先上 A 一周再上 B 一周比较两周数据——不是 A/B，违反并行性。口诀：同时间、两组人、比数据。

过渡：定义里那三个要素，正好对应它的三个特点，逐条拆。

【7.187】A/B 测试特点（3 分钟）

有些团队号称在做 A/B，问下去只有一件事：新版本先放给了百分之五的用户。

>

> 这不是 A/B，这是小流量灰度。可信度要靠三个特点撑着：先验性、并行性、科学性。

>

> 先验性。课件写的是，用流量分割和小流量测试，先让线上部分小流量用户使用，验证设计。再根据数据反馈推广到全流量，减少产品损失。说白了是先拿百分之一到几的用户试，试对了再全量。为什么不直接全量？全量是一次不可逆的赌注，方案错了全体用户受损，损失被放大。小流量把试错的代价关在一个可控的池子里。先验不是先知先觉，它指在全面推广之前先取得证据。

>

> 并行性。同时运行两个或两个以上版本的试验做对比分析。而且保证每个版本所处的环境一致，避免流程周期长，节省验证时间。两个词最关键：同时，环境一致。串行做，环境一变，你分不清差异是方案带来的还是环境带来的。并行还附带一个好处，快。串行两轮四周，并行两周出结果。

>

> 科学性。基于统计的数据做决策，避免主观或经验的错误决策。这里要咬住统计两个字。不是数据多就科学。数据多但抽样有偏，照样错。科学体现在随机分流、样本量足够、看置信区间、考虑显著性。

>

> 三个特点缺一个，结论就软。三个都占，才算一个实验。

>

> 现实里最常见的半成品，就是只做第一条。小流量放出去了，没有对照组，也没有显著性检验，最后拿感觉还行收尾。

> **提问：**新推荐算法先给百分之五用户用，两周后看着不错就推全量。先验性做到了，还差什么？

> **预设回答：**

- 答"没做并行对照，这百分之五自己跟自己比"——抓住要害。中间夹着时间因素，归不了因。正确做法是同时留一组用老算法。

- 答"百分之五太小，样本不够"——有可能。要看基线指标和效应量，有的指标百分之五够，有的百分之五十都不够，得估算。

- 答"两周太短"——也对。要考虑新奇效应，用户刚见到新东西会多点几下，两周后回落。时长要覆盖一个完整的用户行为周期。

- 收口：三个回答分别指向对照缺失、样本量、时长，正好是评价一节的三个要素。

> **易错点 / 考点：**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。并行性的采分点是同时加环境一致，只答两个版本不给分。选择题常把科学性换成数据量大就科学来设错，科学性指统计推断的规范性。三步判断：有对照吗？同时吗？显著吗？

> **过渡：**特点说完，最实际的问题来了：哪些事值得拿 A/B 去试。

【7.188】ABTest 适应的场景（3 分钟）

> 哪一类改动值得拿两组用户去比？课件给了四类场景。

>

> 第一类，灰度发布，用在技术和算法迭代。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 的价值是把故障面关在小流量里，同时对比新旧版本的错误率和耗时。这一类里，A/B 和灰度发布高度重叠。

>

> 第二类，功能优化，界面模块、样式风格、交互方式。这是最经典的战场：按钮位置、配色、导航层级、表单几个字段。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三天，转化能差一截。内容类实验成本极低、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、券面额、召回话术、客服应答模板。这一类直接对着收入，所以最容易吵架，因为它涉及给谁发多少，光靠经验拍不准。

>

> 分这四类不是为了背名字，是因为不同场景的判据指标、所需样本量、能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。

>

> 边界也得说。有些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，攒不出样本。或者效果要几个月后才显现。这三种硬上 A/B，只会得到一个不成立的结论。

>

> 一句话收：四类场景决定你测什么，三个条件决定你能不能测。

> **提问：**想验证会员年费从 198 降到 168 会不会提升总收入，能不能用 A/B?

> **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。更准的说法是可以测，但得换设计，比如按地区或新老用户分组，事先评估公平性。

- 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。

- 收口：这道题的意义是记住三条边界——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。判断三问：能同时分组吗？流量够吗？周期够吗？

> **过渡：**场景摆开了，下一页换个角度问：这些场景里哪些真值得做。

【7.189】ABTest 适应的场景（4 分钟）

> 场景摊开之后，落到手上还有个更硬的问题：这个实验做不做得成。

>

> 第一个条件，改动成本。内容优化在成本最低的一头。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。所以成熟团队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，成本上一个台阶，效果通常几天到两周看得出来。灰度发布比较特殊，它的目的不是选优，是控险。新架构上线不做小流量验证，等于拿全量用户赌。

>

> 第二个条件，流量够不够攒样本。一个日活两千的 App，做前端按钮实验，一周可能只攒到几百次曝光。这个量级看不出百分之二的差异。流量不足时的替代做法有三条：拉长周期、放大改动幅度、改用定性方法，比如可用性测试和访谈。

>

> 第三个条件，反馈周期能不能接受。运营优化常卡在这。发券、改推送，直接动收入，可效果往往要等一个复购周期才看得出来。实验窗口要几周。效果要是半年后才显现，时间成本就不划算了。

>

> 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，业务上就是不允许两组不同待遇。价格、会员权益、对外承诺的服务水平，都不能一半一半。这类事落在不能做 A/B 的清单里。

>

> 四个条件合起来看，适用性不是由想不想测决定的。它由成本、流量、反馈周期、能否差异化共同决定。任一条不满足，就换方法。换方法不等于放弃改进，试点、前后对比、历史数据、定性研究，都是路。

> **提问：**一个内部 OA 系统，全公司三千人，想验证新版首页布局能不能减少审批耗时。做得成吗？

> **预设回答：**

- 答"做得成，一半人用新版"——先别急。三千人里每天真走审批的可能只有几百人，样本被进一步稀释；审批耗时还受流程和审批人习惯影响，方差很高。

- 答"做不成，人太少"——方向对，但要给替代方案：按部门做前后对比，或者先小范围试点一个部门跑一个月，再收定性反馈。

- 答"直接全量上，出问题再改"——最危险。内部系统全量切换的返工成本往往比想象的高。

- 收口：关键不是能不能分两组，是每组能不能攒够有效样本、方差压不压得住。

> **易错点 / 考点：**这一页考适用性判断，标准答案要给四个约束，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。另一错答是把不能做 A/B 等同于不能改进，替代路径有三条：试点、前后对比、定性研究。

> **过渡：**场景和边界都清楚了，接下来进这一节的骨架，六个概念。

【7.190】核心概念（4 分钟）

> 先看一个具体的。同一个用户，在推荐实验里被分到 B 组，他还能不能在按钮实验里进另一个组？

>

> 这个问题要先弄清六个概念：场景、桶、层、实验、流量、流量正交。

>

> 场景对应业务场景，场景之间完全独立。首页推荐、搜索排序、支付收银台，就是三个场景，三件事互不干扰。

>

> 桶圈定特定的用户流量，不同的桶之间流量互斥。它从属于场景，一个场景可以有多个桶。一个人在同一个场景里只能待在一个桶。

>

> 层是一类实验的集合，也从属于场景。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。所以刚才那个用户，同层只能进一个，不同层可以各进一个。

>

> 实验，是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。它通常用来验证某个功能或策略对系统指标的影响。课件举的是 PV／UV、CRT、下单转化率。

>

> 流量指所有用户请求。这里咬住请求两个字，不是用户数。一个用户一天来十次，就是十份流量。

>

> 流量正交，是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？两层不独立，实验就互相污染。你在界面层测按钮颜色，同时在算法层测推荐模型。两层用同一批人、同一套分配逻辑，点击率变了算谁的功劳？正交让每个实验都能单独归因。

>

> 六个概念收成一句：场景装层，层装实验，实验分桶，桶切流量；同层互斥，异层正交，场景独立。

> **提问：**一个用户在界面层进了实验 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"——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。

- 答"不对，不同层可以同时进"——正确，还要补一句理由：不同层分配独立，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。

- 收口：三层就一句话——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，还有互斥与正交的对象搞反。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：**名词有了，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）

> 总流量一百，先按场景隔开，场景内部再切层。这一步切错，两个实验的数字一起废。

>

> 切法里最反直觉的一点：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。界面层可以是 A 组一半、B 组一半；算法层也可以是 C 组一半、D 组一半。这两层的一半，不是同一批人。想通这一句，正交就懂了。

>

> 再说互斥。同一层里要同时跑两个实验，比如按钮颜色和按钮文案。不能让两个实验各占一半再各占一半，那加起来百分之二百。正确做法是把这一层的流量切成互斥的桶。实验一拿一到五十，实验二拿五十一到一百。每个桶内部再跟对照组对半分。

>

> 最后说叠加。一个请求进来，网关依次在每一层算出它属于哪个桶，把各层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。

>

> 看图时盯住那条约束：同一层内所有实验含对照的流量加起来，必须等于这一层的总流量。多了说明有重叠，少了说明有流量没分出去。两种都会让统计结论失真。

>

> 给个数字找感觉。某场景一天一百万请求，界面层开两个实验各占两成，剩下六成留空。算法层开一个实验占三成，剩下七成留空。留空不是浪费。它给没参与实验的多数用户保留稳定体验，也给后面的实验留出流量预算。

>

> 所以容量不够时，工程上最常见的解法不是硬挤，是把实验挪到另一层。挪过去就变成正交关系，不再抢同一层的流量。代价是归因口径也跟着变。

提问：同一层里已有一个占六成流量的实验，还能不能再加一个占五成的？

预设回答：

- 答"能，各分一半用户"——错。同一层互斥，六十加五十已经超了，而且各分一半和占六成是两种互相矛盾的分法。

- 答"不能，最多再给四成"——正确。剩下的四成可以给新实验，也可以留一部分作对照或留空。

- 答"把它挪到另一层去"——很专业。换层就变成正交关系，不再抢同一层流量，但归因口径也跟着变。

- 收口：三答分别踩中互斥约束、流量预算、用正交解决容量，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。计算题：同层已有三成加两成半，还能给多少——答四成半，并建议其中保留对照。

过渡：模型清楚了，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）

分流画在纸上是一张图，落到工程上要先回答一个问题：这段判断代码放在哪儿跑。

>

> 课件给的方案是基于 OpenResty 的多层分流模型。先把 OpenResty 说清。课件说，它是一个基于 Nginx 与 Lua 的高性能 Web 平台。用来方便地搭建能处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。官网在 openresty.org/cn。一句人话：把 Nginx 的并发能力和 Lua 的脚本能力接在一起，让网关能跑业务逻辑。

>

> 链路按请求走。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。

>

> 到网关，这是课件说的跨平台、跨语言的 ABTest 网关。网关做三件事。用 MurmurHash 算分流。从 Redis 或实验策略里取当前生效的配置。比如一到五十给版本 A，五十一到一百给版本 B。再做 URI 匹配，把请求路由到具体微服务。

>

> 终点是后端具体服务程序。它对实验基本无感，看到的就是一个已经打好标的请求。

>

> 为什么放网关，不放每个服务里？终端跨平台，后端跨语言，Java、Go、Node 都有。每个服务自己写一套，会出三件事：算法不一致、口径不一致、改一次要发很多次版。放网关，就是一份分流逻辑、一处配置、全局一致。

>

> 两个细节。一到五十、五十一到一百，这种区间写法说明分流是按哈希值取模后落区间，不是取随机数。随机数每次请求都变，同一个用户一会儿看见 A、一会儿看见 B，实验就废了。哈希保证同一标识永远落同一个区间。

>

> 还有一个，策略放 Redis。配置在外面，改流量比例不用重启网关。这才撑得起一天跑十几组实验。

提问：分流为什么必须用哈希取模落区间，不能用随机数？

> **预设回答：**

- 答"随机数更公平"——概率上公平，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"——对，这是核心。输入固定，输出就固定。

- 答"那用哈希会不会某个桶特别大"——会，所以哈希函数有要求：分布要均匀。下一页讲 MurmurHash 就是为这个。

- 收口：答案就是三个词——稳定、均匀、可复算。可复算很要紧：出了问题，拿标识重算一遍就知道他在哪个桶。

> **易错点 / 考点：**必记两点：OpenResty 等于 Nginx 加 Lua；分流的关键输入是设备标识、实验或策略标识、层标识。常见错答：A/B 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。选择题还可能问为什么策略放 Redis，答动态下发、免重启、多实例一致。

> **过渡：**这套模型到底解决了什么问题，下一页逐条摆出来。

【7.193】OpenResty 分流模型的优势（4 分钟）

> 一套架构值不值得用，不看它功能多，看它把哪些麻烦从业务方手里拿走了。

>

> 第一，接入成本低，业务无侵入。分流在网关做，业务代码几乎不动。这一条对老系统格外要紧。一个跑了五年的系统，让它改一遍分流 SDK，工期和风险都受不了。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，全走同一套算法。各端各写一套，最常见的事故是 App 算出来 A、H5 算出来 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，这个坑从根上就没有了。

>

> 第三，性能开销小。Nginx 加 Lua 本来就为高并发设计，一次哈希的成本远小于一次业务请求。分流不该变成瓶颈，这是架构上的硬要求。

>

> 第四，策略动态化。配置放 Redis，运营改比例、开关实验、加白名单，都不用发版重启。快是 A/B 的生产

> 力，一天只能改一次配置，就别谈高频实验。

>

> 第五，不产生二次流量。分流在入口一次决定，请求只走一条路。要是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，比出来的也不可信。

>

> 好处说完，代价也得看。网关成了关键路径，它挂了全站都受影响，所以要多副本、健康检查、配置降级。另外 Lua 代码的可测性和可观测性比普通业务代码差，实验逻辑得自己做单元测试和灰度验证。

>

> 所以这个方案的强项是统一和轻量，弱项是网关必须足够可靠。两个词一起记，才是一套完整的选型理由。

> **提问：**网关挂了，A/B 实验怎么办？系统这时候要拒绝服务，还是按默认版本放行？

> **预设回答：**

- 答"拒绝服务，至少不会出错"——最危险的错答。实验是增强功能，不是核心功能，为实验正确性把站点打挂，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，实验数据缺一小段可以接受。

- 答"提前把配置缓存在本地"——很专业，这是多级缓存：本地兜底，Redis 常态，网关聚合。

- 收口：这道题把优先级讲透了——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考。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，标准答案必含降级和本地兜底。

> **过渡：**优势里反复出现均匀和稳定，这靠算法，下一页专门讲它。

【7.194】MurmurHash 算法（4 分钟）

> 同一个用户的 id 不变，他凭什么每次都进同一组？

>

> 答案在算法的输入里。参与哈希的因子有三个：设备 id、策略 id、流量层 id。三个各有分工。设备 id 决定你是谁，保证同一个用户稳定落桶。策略 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。三个拼起来做一次哈希，稳定和正交同时拿到。

>

> 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 这类是为防篡改设计的，要抗碰撞、要不可逆，代价就是慢。分流不需要密码学强度，没人会为了看 B 版本去攻击你的哈希。而分流在请求链路上，每一毫秒都在算。

>

> 第二个，变化足够激烈。课件举的例子是 abc 和 abd，只差一个字符，能均匀散布在哈希环上。这条比快更关键。要是哈希对相近输入给出相近输出，设备 id 连号的一批用户会全落进同一个桶。桶大小失衡，结论直接失真。雪崩效应让一个比特的变化把输出彻底打散。

>

> 最后两个定义必须分清。互斥，是两个实验流量独立，用户只能进其中一个，这是同一层的关系。正交，是用户进的所有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做到，后者靠层 id 进哈希因子做到。

>

> 三个因子、几十倍的速度、足够激烈的分布，把稳定、均匀、正交一次算完。换成 MD5，这三个性质能剩下的不多。

> **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。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速度，在每请求都算的链路上是

纯浪费。

- 答"那 CRC32 行不行"——可以讨论。CRC 快，但分布质量不一定够均匀，对连号的设备 id 容易聚簇。选算法要看速度和分布两个指标。

- 收口：标准答法就一句——分流要的是快和均匀，不是抗碰撞。

> **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，因为桶大小失衡会让两组基线不可比。

> **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）

> 两周过去，结论是没变化。回头看记录，问题出在第一步：那个想法本来就没人关心。

>

> 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。

>

> 第一步，想法从哪来。三个来源：数据分析，看哪里流失高、哪里点击低；用户反馈，客服工单和调研；竞品和自己的经验。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就当实验做，跑两周才发现没人注意这个改动。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，评估优先级。想法永远比资源多，必须排序。排的时候看三件事：预期影响多大、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。这一步防的是把两周花在一个百分之一的改动上。

>

> 第三步，设计和开发实验。要定下来的是这几样。假设，改什么、期望什么变化。主指标，唯一的判定依据。护栏指标，不能变坏的那些。还有分流比例、样本量和时长、埋点方案。这一步最容易出错的地方是指标定太多。指标一多，事后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，实际四十七比五十三，说明分流或埋点有问题。

>

> 第五步，应用结果。全量上线、下线，或者改设计再测。重点在应用这两个字。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。结论还要沉下来：这个人群、这个场景，什么改动有效。

> **提问：**第三步为什么一定要只留一个主指标？多几个不是看得更全吗？

> **预设回答：**

- 答"指标多了更全面"——最典型的错答。指标一多，事后总能挑出一个显著的，等于把判定交给运气。

- 答"多重检验会抬高假阳性"——正确也专业。测二十个指标，按百分之五的显著性水平，纯随机也会冒出一个显著。

- 答"主指标一个，其余当护栏"——最佳。护栏不是用来比优劣，是用来一票否决的：点击率涨了但退款率也涨，就不能上。

- 收口：三答的差别就是裁判有一个还是二十个。一个裁判、一套事先定好的规则，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的顺序颠倒；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，清理代码这一条最多人漏。

> **过渡：**流程说完，下一页看数据在这五步之间怎么流。

【7.196】A/B 测试 数据流（4 分钟）

> 报表上两组样本量四十七比五十三，本该对半分。先别急着怀疑算法。

>

> 这张图要看出两个闭环，而且两个闭环必须用同一个标识对齐。

>

> 第一个闭环是分流。实验配置存在配置中心，SDK 从平台拉配置；请求进来算出桶号，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走完，通常几十毫秒。

>

> 第二个闭环是数据。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 分流时给这次请求打上 abTraceId 和 bcm。一个标识这次请求，一个标识这次结果。埋点上报时一起带上。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。

>

> 不打标会怎样？这就是实验结论不可信最常见的根因。你有一堆点击日志，也知道两个版本各自的转化率。可分不清哪条来自哪一组，只能按时间或用户 id 去猜。一猜就错。

>

> 三个工程细节值得点出来。配置要有版本和下发时间，否则实验中途改了比例，前后两段数据混在一起没法解释。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，这是排查线上争议最有效的一招。上报要能抗丢，日志丢了会让两组样本量不对等，而样本量不对等直接破坏统计结论。

>

> 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。

提问：报表里两组样本量差得很多，本该接近对半分，问题最可能出在链路哪一段？

预设回答：

- 答"分流算法不够均匀"——可能，但先排除更常见的两类：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。

- 答"分流比例本来就不是一半一半"——也常见，尤其实验中途调过比例，前后两段混算就失衡。

- 收口：排查顺序是先查配置比例，再查埋点丢失，最后才怀疑算法。反过来查要浪费很多时间。

> **易错点 / 考点：**两个闭环加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：**这条链路两端，一头是平台，一头是代码里的 SDK。先看 SDK 做什么。

【7.197】A/B 测试 SDK（4 分钟）

> SDK 这个词听着像个库，它干的其实是三件很具体的活。

>

> 第一件，实验配置动态分发。课件原话是，实现针对场景标识加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，做实验配置的动态分发。两个词要抓住，一致性哈希和动态。一致性哈希的意思是，你调流量比例、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，你不可能每次改配置就把一半用户的体验重置一遍。动态是说配置不写死在代码里，SDK 定期拉或者接收推送，改配置不用发版。

>

> 第二件，上报数据。三类日志。请求日志，记每次分流的结果，用来校验流量比例。业务自定义日志，记转化、下单金额这类业务指标。监控日志，记 SDK 自己的健康度，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。很多团队只报业务日志，数据对不上时无从排查，缺的就是另外两类。

>

> 第三件，基于埋点规范生成前端埋点标识。课件给了三个。abTraceId，请求唯一标识。bcm，请求结果标识。还有事件标识，曝光 view、点击、收藏、转发这些。这些标识透传到前端埋点，用来追踪用户行为。

>

> 为什么要 abTraceId？一次请求可能带来多个行为，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一串行为绑在一起，避免重复计数，也避免漏计。

>

> 三件事收一句：拉配置、报数据、打标识。

> **提问：**一致性哈希里的"一致性"保证了什么？换成普通取模，改一次流量比例会怎样？

> **预设回答：**

- 答"保证同一用户看到同一版本"——只说对一半。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，比如从一百改到一百二，几乎所有人的余数都变，等于全员重新洗牌，数据连续性就断了。

- 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。

- 收口：落点就一句——扩容和调比例不洗牌。

> **易错点 / 考点：**SDK 三件事、三类日志、三类标识都是采分点。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。若问 SDK 为什么必须上报请求日志，答案是校验分流比例是否正确、支持复算和排查。

> **过渡：**SDK 在业务侧干活，另一头是平台本身，看平台要有哪些功能。

【7.198】A/B 测试平台功能（4 分钟）

> 一个 A/B 平台要有哪几块，问每天用它的人最准。接入要几天、这个用户到底进了哪个桶、昨天那条曲线为什么抖。三个抱怨正好对上四块功能。

>

> 第一块，元数据管理，包括应用、场景与实验的创建、编辑，以及配置的发布。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，实验要有状态：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，接入的开发和测试支持。课件说得很具体：用户能查看完整可用的接入示例代码，能输入分流用户

> 标识做测试，能查询实时日志。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，让开发自己就能验证这个用户进哪个桶，不必每次找平台同学。实时日志决定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。实时报表含实时请求、点击率、转化率、千次曝光转化这些指标，以及对比分析。离线报表做更完整的统计和分群。为什么两套都要？实时报表用来早发现问题，比如比例不对、某个版本报错暴涨。离线报表用来下结论，比如显著性、置信区间、长期效果。拿实时报表下结论，是新手最常犯的错。

>

> 第四块，异常监控和告警。平台实时盯着 SDK 上报的请求数、失败数和延迟，发现异常就告警。这一块最容易被省掉，可它是刹车。分流服务挂了，用户看到的是错误页，不是少了一个实验。

>

> 四块收成一句：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。

> **提问：**平台为什么要同时给实时报表和离线报表？只留一个不行吗？

> **预设回答：**

- 答"实时就够，看得快"——最典型的错答。实时数据往往是初步结果，样本不足、没清洗，拿它下结论就是拿噪声当结论。

- 答"离线更准，只要离线"——也不行。离线有延迟，比例配错、某版本报错暴涨这类问题要等第二天才发现，损失已经发生。

- 答"实时看异常，离线下结论"——标准答案。一个监控，一个统计推断，分工不同。

- 收口：这也是数据系统的通例——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，还要能各说一句用途。易错：把实时报表当效果判定依据；漏掉配置发布和回滚这一条，元数据管理不只是增删改。若问平台为什么需要状态机，答防止运行中的实验被随意修改，保证数据可比可追溯。

> **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）

实验跑了一周，B 组转化率比 A 组高零点四个百分点，p 值零点零六。产品经理说，就差一点，算通过吧。

>

怎么回答？第一件事，看差异是不是统计显著。两组转化率百分之三点一和百分之三点三，差零点二个百分点。这个差是真的还是随机波动，要靠显著性检验和置信区间。要点是先定样本量再看结果，不是看到结果再算还差多少样本。

>

第二，看样本量和时长够不够。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，用户对新鲜东西短期更活跃。还会撞上周期效应，周末和工作日行为不一样。经验做法是至少覆盖一个完整的用户行为周期。

>

第三，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，不能因为提升点击就把退款率抬上去。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，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，也可能是某类用户的效果被另一类稀释了。

>

第五，看结论能不能落地。提升零点三个百分点、要多花两倍服务器，这个方案未必值得上。

>

五件事里最危险的不是算错，是算对了却选择性地说。同一份报告，只报显著的指标，把两个百分点说成大幅增长，或者跑到一半偷看结果再决定停不停。这些做法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。

提问：实验跑了一周，B 组高零点四个百分点，p 值零点零六，产品经理说算通过。怎么回答？

预设回答：

- 答"零点零六很接近零点零五，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，还有百分之六是噪声"——表述不严谨但方向对。更准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。一边延长一边反复偷看，会放大假阳性。

- 收口：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：把相对提升和绝对百分点混着说，三点一到三点三是零点三个百分点，也是百分之十的相对提升；认为两组数据一算就知道谁好，忽略显著性和样本量；认为指标越多结论越可靠，实际多重比较会抬高假阳性。简答按五看作答即可。

> **【思政落点 19】**：数据不会自己说话，是人在解释它。口径、样本量、置信区间、不利指标，一起报，才算把结论交出去。只挑对自己有利的那几条，技术上挑不出毛病，可决策已经被带偏了。诚实在这里不是一句口号，是让别人能复查你的判断。做实验的人守住这条，团队才敢拿数据当依据。

> **过渡：**评价标准有了，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）

> 这几条不是理论，是别人踩过坑之后留下来的做法。

>

> 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，也不会产生二次流量。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，业务只读标。不产生二次流量值得单说，接口形式会引发额外调用，量大了就是实打实的成本。

>

> 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，每个实验要明确说清我在哪一层、占多少流量，避免互相抢。

>

> 第三条，数据传输。在 HTTP 头部增加分流信息，业务方不用关心实现语言。Java 读 header，Node 也读

> header，语言无关，这是解耦的关键。

>

> 第四条，监控体系。实验本身要能被监控：分流生效没有、比例准不准、SDK 有没有异常。没有监控，实验是黑盒。

>

> 第五条，用户画像这类精细化定制。不是简单对半分，而是按人群定向，新用户、老用户、高价值用户分开做。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。平台自己就能算，不必让分析师手算。

>

> 第七条，通过 AARRR 模型评估实验对北极星指标的影响。AARRR 是获取、激活、存留、收益、推荐。课件指出，这个模型把控产品整体的成本和收入关系。判据是用户生命周期价值 LTV，远大于获客成本 CAC 与经营成本 COC 之和。

>

> 判据就是 LTV 大于 CAC 加 COC。注册率提上去了，拉来的却是低质量用户、LTV 掉下来，那是指标赢了、生意输了。这就是为什么要有北极星指标和护栏指标。

提问：一个实验把新用户注册率提升了百分之八，三个月后这批用户付费率明显低于老渠道。算成功吗？

> **预设回答：**

- 答"算成功，注册率是明确的提升"——只看第一层指标，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。还要分清是付费率掉了还是付费总额掉了，以及这批用户本来就该是低付费人群。

- 答"要看 LTV 和 CAC 的关系"——最专业。获客成本没降、LTV 却降了，那就是用数量换质量，长期是亏的。

- 收口：任何单点指标的提升，都得放回 AARRR 漏斗和 LTV、CAC 的关系里再看一遍。

易错点 / 考点：七条实践要能列全，其中无侵入加无二次流量、HTTP header 传分流信息、AARRR 最常考。易错：把 AARRR 五个字母记错，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 的关系。

【思政落点 20】：单个指标涨了，不等于事情做对了。注册率提八个点，三个月后付费率掉下去，报表上好看，长期是亏的。工程上的价值判断常常得往后看一步：这次改动让谁受益、代价落在谁身上、能撑多久。把 LTV 和成本放在一起算，是一种不被短期数字牵着走的习惯。

过渡：定义、机制、流程、评价都过了一遍，剩下的工夫全在实践里。

【7.201】A/B 测试优秀实践-续（4 分钟）

先想一个场面。平台演示的时候很漂亮，业务方接了三天，第四天说算了，我们还是拍脑袋。

>

接入这件事，输的往往不是功能少，是每一步都要人教。前面讲的是实验怎么做对，这里讲平台自己怎么才算好用、扛得住。

>

第一组，易用性。元数据要全：这个实验谁建的、改过几次、当时假设是什么。不全，三个月后没人说得清那次调整为什么做。应用级的角色和权限要分开。不然谁都能改线上流量，一次误操作就是全量。白名单要能按商家 id 指定人，也能按分流标识指定。开发和测试先把指定用户拉进组里，不必等真实流量慢慢进来。

>

第二组，系统优化。多级缓存，SDK 本地一份、网关一份、Redis 一份。不然每个请求都去查一次配置。SDK 性能要压住，分流挂在业务请求上，它慢一点，全站都慢。配置要自动化、透明化。接入方不用手工填一堆参数，改了也能自动生效。一致性哈希要优化，图的是调流量比例时尽量少迁移用户。一个用户被搬一次，他的数据就断成两段。

>

第三组，Java 和 Node 两端都得有能跑的示例。引入代码库、初始化、获取实验配置、前端埋点，四样齐。接入成本的一大半花在照文档试错上。有示例，两天能变两小时。

>

第四组，测试、预发、生产要分开。配置不隔离，测试环境调的比例会打到生产，那是全量事故。

>

> 第五组，输入一个用户标识，平台告诉你他进了哪个实验、拿到什么配置。数据一有争议，查一遍就清。

>

> 第六组，实时日志明细查询，按日志类型、实验、分流标识去查。缺了它，数据不对就成了没有答案的悬案。

>

> 底部那句最要紧：接入成本和数据价值。平台功能再全，接入要两周、报表没人看，它就是个零。

> **提问：**为什么"测试、预发、生产三类环境"的支持，被列成平台的必备能力，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：三个环境共用一个配置中心，测试环境调的比例就会打到生产。

- 答"生产不能随便改，得先在别处验证"——对，这是环境隔离的基本逻辑，和代码分支隔离是一回事。

- 答"三个还不够，还得有灰度池和多机房"——很专业。环境这个维度不止三个：地域、机房、灰度池都要能隔开。

- 收口：这道题的关键词是隔离。实验配置会直接改变用户所见，它比普通业务配置更需要隔离和权限。

> **易错点 / 考点：**本条的核心是易用性与接入成本。常问"为什么接入成本决定 A/B 平台的价值"——实验数量和迭代速度取决于接入成本，成本高则实验少，投入就收不回来。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"降低对配置服务的依赖、支持降级"。

> **【思政落点 21】**：平台好不好用，接入方最有发言权。你多写一份示例、多留一个默认值，别人就少熬一个通宵。你把参数全摊给对方，就是拿自己的省事换别人的麻烦。工程上的善意很具体：让下一个人少走一步弯路。落到价值观上，就是做事替使用者想一层，别把成本悄悄转给看不见的人。

> **过渡：**平台要有什么说完了，看一块真实看板长什么样。

【7.202】A/B 测试 Dashboard（4 分钟）

27% 到 78%，这一跳足够让会议室安静下来。先别急着高兴，看它用的是哪种口径。

>

标着 Feature Y 的那组卡片上有三个数。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%。还有一个 Off vs On 的 6.60%。三个数，三种讲法。

>

第一个是绝对分值。153.8 比 120.9 高，可"高 32.9 分"说明不了好坏。分值的量纲是自己定的。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 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。三者混着说，结论就会被放大，也会被缩小。

>

再看 Feature X 那组。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。这组的基线本来就高。从 85 到 87 只涨两个百分点，可比从 27 涨到 78 难得多。基线越高越难提，这是排实验时就得有的预期。不是每个实验都能有前一种量级的跃升。

>

第三点，看板只是看板，不是判据。它告诉你趋势，能不能上线要靠统计检验。样本够不够、置信区间跨不跨零、护栏指标有没有变坏。看见绿箭头就上、看见红箭头就砍，少掉的正是检验这一步。这是很多团队的通病。

>

最后是看法：两组卡片并排看。一组涨几十个百分点，一组涨两三个。这两件事的性价比完全不同，资源往哪投，一眼就分出来。

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"。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%，可能是十几个人算出来的。

- 答"先问 78% 是百分点还是相对提升"——这是本题的核心陷阱。27% 到 78% 是涨了 51 个百分点，而"提升 78%"这个说法本身口径不明。

- 答"先问这是不是实时数据、跑够时间没有"——也很要紧，实时看板往往只是初步结果。

- 收口：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：把看板数据直接当上线判据；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。判断题"相对提升越大效果越好"——错，要看基线和显著性。

> **过渡：**一块看板只放几个指标，真实实验往往得同时盯一整套。

【7.203】A/B 测试分析（4 分钟）

> 日均登录次数涨了 32%，能不能当成好消息？先别答，这组数里埋着一个陷阱。

>

> 表上是九个指标和它们的相对变化。平均会话时长 1%。每次会话内容浏览次数 28%。日均用户会话数 7%。日均用户生命周期价值 60%。日均订阅发生次数 24%，日均订阅时长 24%。日均登录次数 32%，日均订阅次数 9%。最后是日活跃用户数。数不逐条念，讲怎么读。

>

> 第一，指标是有层级的。从规模到商业价值大致这么排。日活是规模，会话和浏览是参与度。登录和订阅是关键行为，生命周期价值是商业价值。看的时候有顺序：先看规模和参与度有没有被动到，再看商业指标。日活掉了 5%，那个 60% 的生命周期价值就得打问号。很可能是分母变小撑起来的。

>

> 第二，留意"每人""每次"这种归一化口径。per daily user、per session，都把总量除掉了人数或者次数。这类指标的脾气是分子分母同时会变。日均登录次数涨 32%，可能是登录变多，也可能是活跃人数变少、分母缩了。所以归一化指标必须配总量指标一起看。

>

> 第三，防"指标太多"。九个指标，每个都做显著性检验，按 5% 的水平，纯随机也会有将近半个指标"显著"。这就是多重比较。规范做法是事先指定一个主指标。剩下的当解释性指标或者护栏指标，只有主指标用来下结论。

>

> 第四，看指标之间的因果链。浏览多了 28%，会话时长几乎没动，只涨 1%。订阅多了 24%，生命周期价值涨 60%。这条链大致说得通。要是出现"浏览涨了、时长没变、价值暴涨"，那多半是别的因素在动，或者埋点口径改了。

>

> 收一句：指标不是越多越好。要有一条说得通的因果链，还要事先定好谁当裁判。

> **提问：**"日均用户登录次数提升 32%"，这个数能不能直接当好消息？

> **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；日活掉一半，登录次数照样能涨。

- 答"不能，要先看日活有没有掉"——对，这就是分母陷阱。

- 答"还要看登录策略有没有改，比如强制重新登录"——很专业。指标异动先排除机制变化，再谈效果提升。

- 收口：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"，标准答案要有四条：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"——多重比较把假阳性抬上去。

> **过渡：**一次只测一个改动太慢，工程上就想一次测多个。

【7.204】多因素的 A/B 实验（5 分钟）

> 四个改动里有两个都叫"全新落地页"，一个 −4.0%，一个 +6.0%。同一个名字，相反的结果，这事本身就值得停一下。

>

> 四个实验和结果是这样。按钮颜色测试 +1.5，新标题测试 +0.8。全新落地页 −4.0%，另一个全新落地页 +6.0%。旁边还标了达到统计显著置信度所需的时间，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，涨 1.5 个单位；换个标题，0.8。两个"全新落地页"一个大亏一个大赚。这说明大改动的方差大，可能大赚，也可能大亏。所以大改动更需要 A/B，不是更不需要——凭直觉你根本判不了。

>

> 第二，同样是全新落地页，一正一负。这就是 A/B 值钱的地方。它不回答"大改好不好"，它回答"这一个方案好不好"。决定结果的是设计质量，不是改动幅度。

>

> 第三，达到显著要的时间差得远，2 周到 8 周。为什么？达到显著需要的样本量取决于效应量。效应大，很快看得出来；效应小，就得更多样本。+6.0% 只要 2 周，+0.8% 要 8 周。小改动常常要付更长的时间成本，这段时间在排期时就得算进去。

>

> 再讲"多因素"本身。一次只测一个变量，归因干净，代价是慢。四个改动串着测，6 加 8 加 4.5 加 2 周。多因素实验用析因设计或者正交设计，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。新标题配新落地页，可能互相增强，也可能互相抵消。代价是组合数涨得快，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，但先把因子数和水平数压住。

提问：四个改动，你会串着一个一个测，还是一次设计一个多因素实验一起测？各自要付什么代价？

> **预设回答：**

- 答"一起测最快"——快，但代价有两个：样本被摊到很多组，单组不够；交互效应会让主效应不好解释。

- 答"一个一个测最清楚"——归因干净，可按课件给的周期算，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。先用较少样本找出哪些因子有用，再对关键因子做一次细验证。

- 收口：三答对应三种取舍——速度、清晰度、成本。成熟的实验体系一般两级走：先筛后验。

> **易错点 / 考点：**必记三点：改动幅度大不等于收益大；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：看到 −4.0% 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"同时跑几个独立实验"，那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？下面有个具体的分析场面。

【7.205】A/B 测试结果分析（4 分钟）

两个落地页，01 和 02，各自上面还叠了三个开关。你要在几个数里挑一个方案，挑错的方式有很多种。

>

> 图上是这么摆的。Landing Page 01 和 02 对着测。每个版本上叠三个选项：放不放产品视频，去掉还是保留导航，长表单还是短表单。最后看两个结果：邮件订阅注册总数，和金额合计。

>

> 这张图要教的是看两个指标，不是一个。

>

> 先说为什么要有两个。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。把表单缩短、必填项砍掉，注册的人确实多了，可来的人意向弱、付费少。反过来，长表单挡住一批人，留下的那批更精准。只看注册数，你会选错方案；只看金额，你会漏掉规模效应。

>

> 再说主指标和副指标怎么配合。规范做法是事先定一个主指标，比如金额；另一个当约束条件，比如注册数不许掉超过多少。两个指标同向，皆大欢喜；反向，就得做权衡。而这个权衡是业务判断，不是统计判断。统计只能告诉你差异多大、显不显著，它没法告诉你这个交换值不值得接受。

>

> 第三点，选项之间会互相影响。视频、导航、表单长度这三件事不是独立的。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，是把它们组合起来测。这正好接上前面那个多因素。

>

> 第四点，看结果要回到假设。实验开始前写的是什么假设？"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，会变成在数据里翻故事，而且总能翻出一个看着合理的解释。可那不是科学。

>

> 收一句：量看注册、质看金额，先定主指标，再做组合，最后回到假设。

> **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，可能亏。

- 答"金额跌了就不上"——也片面。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、金额的时间窗多长"——对。这道题的答案是分情况，前提是把渠道结构、用户类型、统计窗口都摊开。

- 收口：统计负责说差异显著，值不值得换是产品和商业的事。测试工程师的活是把两个数都摆上桌，别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"每个选项分别最优"当成"组合最优"，中间隔着交互效应。简答题问"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：**整体看不出结论，或者结论只对一部分人成立，怎么办？

【7.206】逐层递进的 A/B 实验：决策树（5 分钟）

> 一张树状图，根上一个数 158，括号里 n=38，100%。往下每分一层，括号里的 n 就越小。

>

> 它治的是这么个毛病：整体实验看不出结论，或者结论太粗。

>

> 根节点按 Pageviews 小于 8460 分成两支。一支是 117，n=7，占 18%。另一支是 167，n=31，占 82%。再往下靠这些条件一层层切。Pageviews 小于 7720、9907、9765、9418。Clicks 大于等于 775、大于等于 730。Experiment 大于等于 0.5。还有 DOW，等于周一周三，或者等于周日、周二、周五、周六。叶子上是一组组数值和 n。

>

> 要看出三件事。

>

> 第一，决策树在做人群切片。整体平均 158，可左边那 18% 的人只有 117，右边 82% 的人是 167。翻成人话：这个改动对多数人有用、对少数人有害，平均下来被稀释了。前面讲评价时说过，要看效应在人群上一致不一致。

>

> 第二，分裂条件就是能执行的运营规则。条件里有流量规模 Pageviews、行为强度 Clicks。也有实验标识 Experiment，还有星期几 DOW。所以结论不是"新版本更好"。它是"新版本对周一周三访问、浏览量低于 8460 的那批用户更好"。这个结论能直接落到投放、灰度和产品设计上。

>

> 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，指标值高得离谱，可那也就是一两个人的结果。n 太小就是噪声。读这棵树得同时看两个数：叶子上的指标值，和括号里的 n。

>

> 还有个风险得提：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"，这是数据挖掘式的过拟合。规范做法是一段数据建树，另一段验证，验证过了才敢上线。

> **提问：**图里有个叶子的指标值高得离谱，括号里写着 n=2。这个发现能当上线依据吗？

> **预设回答：**

- 答"能，数值这么高说明有效"——最典型的错答。n=2 就是两条样本，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"——对。再给一个门槛：分群时每组至少要能支撑一次显著性检验。

- 答"可以拿它提假设，不能拿它当结论"——最佳答案。这是决策树的正确用法：用它找线索，再用一次独立实验去验。

- 收口：这就是探索性分析和确认性结论的分界。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"找到最优人群就直接上线"的工具，少了独立验证；只看叶子的指标值不看 n。问"逐层递进实验的风险"，标准答案：多重比较抬假阳性、过度拟合、样本被切碎后每组不够。

> **过渡：**机制讲到这儿，最后看一家商用平台是怎么搭起来的。

【7.207】A/B 测试 Testin 技术架构（4 分钟）

> 把一家商用 A/B 平台拆开看，里面是五层。Testin 云测这张架构图就是这么搭的，看完你就能自己画出一个骨架。

>

> 按请求和数据的走向。接入层，SDK 和 API，负责跟业务系统对接，拉配置、报数据。分流决策层，管实验配置的存储和计算、分流算法、规则匹配。前面 OpenResty 网关干的就是这个活。数据采集与处理层，接收埋点日志、清洗、落库、聚合。分析计算层，实时指标和离线指标两条链路。展示与运营层，看板、报表、告警、权限。

>

> 为什么非得这么分？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。分析的要求是数据全、能重算。两套要求完全不一样。混在一起，改个报表逻辑会牵连分流服务，风险收不住。

>

> 第二，能独立扩容。分流层的压力跟着请求量走，大促就顶上来。分析层的压力跟着数据量和查询量走。分开部署，各自扩，成本才可控。

>

> 第三，好替换。分流算法可能要换，报表可能要接新的 BI 工具。分好层，换一层不惊动别的层。

>

> 看图还得盯住两条横切的东西。一条是配置与元数据中心，分流层和分析层都依赖它，所以它必须高可用。另一条是监控与告警，它横跨所有层，哪一层出事都得被看见。

>

> 最后核对一下，这张图跟前面讲的机制对得上。终端和 SDK 对应接入层，OpenResty 网关对应分流层。abTraceId 和埋点对应采集层，两张报表对应计算层。对上了，这一节就通了。

提问：如果把分流服务和报表统计塞进同一个服务里，会出什么具体问题？

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"——对。分流跟 QPS 走，报表跟数据量和查询复杂度走，绑在一起只能按峰值配资源。

- 答"发布互相牵连"——也对。改个报表口径就得发一次版，而发版期间分流不能中断。

- 收口：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通用理由。

易错点 / 考点：分层名称和职责要能对上，最常考"为什么分流和分析要解耦"。易错：把实时和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"这条。画图题按五层加两条横切画。

过渡：别人的架构看完了，落回自己——让你设计，要先定清哪几件事？

【7.208】A/B 测试平台设计（5 分钟）

> 让你从零设计一个 A/B 平台，第一件事定什么？不是界面，是模型。

>

> 第一件，领域模型。至少要有这些实体：应用、场景、层、实验、桶、指标、指标快照。它们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，后面所有功能返工。

>

> 第二件，状态机。一个实验得有明确的生命周期：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，线上一定出事。改运行中实验的流量比例，前后数据就不可比。

>

> 第三件，流量与互斥校验。发布前必须校验三样。同一层内所有实验的流量加起来不超过百分之百。桶之间不重叠。白名单用户不冲突。这是安全带，靠人靠不住。

>

> 第四件，分流与配置下发。要定清楚四件事。分流算法，比如 MurmurHash 三因子。配置存哪，比如 Redis。下发是拉还是推，缓存怎么做。还有降级：配置服务挂了就按默认版本放行。

>

> 第五件，埋点和数据模型。指标口径要定死：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。这条在真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，那只能叫看板，不能叫实验平台。

>

> 收一句：平台的核心不是界面好不好看，是四件事——分流准、口径清、数据全、结论可复算。

> **提问：**为什么"流量与互斥校验"必须做成系统强制，不能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：今天这层 60%，明天加个实验变 80%，第三天再改一个桶，靠脑子记不可能。

- 答"校验能防上线事故"——对。同层流量加起来超过百分之百，落在这段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。

- 收口：凡是能被系统拦住的事故，都别指望人靠记性。

> **易错点 / 考点：**七项里，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。问"平台设计最容易忽略什么"，答：指标口径，分子分母、时间窗、去重，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收口，把整个第 7 章归拢一下。

【7.209】本节小结（4 分钟）

> 六句话，把第 7 章收住。小结这几条基本是教材原话，考试按原话答。

>

> 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。为什么数据最要紧？硬件不兼容换台机器，软件不兼容升个版本。数据乱了、丢了、读错了，补不回来。

>

> 第二条，全链路压测可以看作性能测试的右移，全面提升系统的稳定性。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、真实数据去验容量。

>

> 第三条，混沌工程可以看作故障注入测试的右移，并提升为工程层次。还得遵守五项原则，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。不是随手注入一个故障，而是有假设、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。这两个词组就是它的定位。它是这一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的两半。实验设计管假设、指标、样本量、时长、显著性。流量分发管场景、层、桶、正交互斥，还有 MurmurHash 和 OpenResty 网关。

>

> 第六条，建立一个灵活支持 A/B 测试的平台。这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子。

>

> 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，拼成专项测试的整张图。

提问：这一章五个专项，哪一个最贵？哪一个最便宜？凭什么这么算？

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，可平台建设不便宜：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。专项测试的成本得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。

- 收口：三答的层次是资源成本、建设成本、风险成本。问"企业为什么愿意投专项测试"，答：它拦下的风险远大于投入。

易错点 / 考点：小结页是期末复习第一入口，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，下面给的是入口，你们要自己动手。

【7.210】去搭建一个简单的 A/B 测试平台（5 分钟）

两天，做一个能跑通的 A/B 平台，你砍什么？先别答，先看有哪些现成的。

>

课件给了两个入口，都是开源免费 A/B 工具的清单页。一个在 convert.com 的博客，一个在 posthog 的博客。地址在课件上，照着翻。这两篇的价值不在它推荐了哪个工具，在于让你看到这个领域的方案光谱。

>

清单里的方案大致三类。

>

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，自建或者开源自托管型。平台部署在自己的服务器上，数据不出内网，能接自己的埋点体系。优点是可控、能改。缺点是要自己运维，分流服务的可用性也得自己保证。

>

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。

>

选型建议很实在：课程设计用第二类或者第一类。你们的目标是把机制跑通，不是买服务。

>

> 作业按五步做。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，也可以先用最简单的哈希取模。但同一个用户必须稳定落桶。第四步，埋点和统计。至少记曝光、点击、转化三个事件，都带上实验标识和桶号。第五步，跑满一个周期出报告。报告里要有样本量、指标值、置信区间，还有你的结论。再补一问：如果结论相反你会怎么做。

>

> 评分说一句：做得漂亮不如做得对。分流不稳、口径不清、样本不够就下结论。这三条踩任何一条，功能再花哨也是零分。

> **提问：**给你两天时间做一个"能跑通"的 A/B 平台，你砍掉哪些功能、留下哪些？

> **预设回答：**

- 答"先做报表，界面对评分要紧"——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"留分流、埋点、一个最简的对比报表，砍掉权限、状态机、多因素"——对。这是最小可用闭环：能分组、能记录、能对比。

- 答"还要留'输入用户标识查桶号'"——很好，这是排查能力，两天做得出来，性价比极高。

- 收口：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，结论仍然可信，只是不安全。

> **易错点 / 考点：**这一节考实践，通常落在"三类方案怎么选"和"最小可用平台含哪几块"。易错：把"装一个开源工具"当成"搭平台"，缺自己的埋点和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。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）

> 六本书，别全买。先挑一本跟你课设题目对得上的精读，剩下那几本当工具书。

>

> 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。它接的是本章性能测

> 试那段，特点在"实践"。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。课设做后端性能实验，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。做实验时当工具书翻。

>

> 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。接的是安全性测试那一节。这本书讲的是攻击手法，看的时候带着"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。接全链路压测那段。性能问题到了全链路这个层次，就是架构问题：缓存、限流、降级、分库分表。这本讲架构侧怎么配合。

>

> 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。接混沌工程。这个领域里它算系统的，原则、实验设计、爆炸半径控制都讲得细。

>

> 第六本，《A/B 测试：创新始于试验》。机械工业出版社，2019 年 4 月。接今天这节。书名里"创新始于试验"五个字就是 A/B 的方法论内核：别争论，去试验。

>

> 六本覆盖六个方向。技术书的价值在你遇到问题时它在手边，不在你读完了几本。

提问：这六本里，只能带一本进课程设计答辩现场，你带哪本？

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"带我课设题目对应的那本"——对。做性能优化带性能实践，做混沌实验带混沌工程。

- 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。

- 收口：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。书名要写准，六本都是"出版社加年份"，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成一件事。前者是发现，后者是解决，第一本书名里两个词都有。

过渡：这一章到这儿，我用三句话收尾。

【7.212】感 谢 聆 听（2 分钟）

最后三句话。

>

第一句，专项测试的判据不是"对不对"，是"达不达标"。性能看响应和容量，安全看攻击面，兼容看环境适配，可靠看故障下的表现。对错是功能测试管的事。专项测试管的是另一件：在真实条件下，它还成不成立。所以它要的是阈值，不是感觉。

>

第二句，越往后走，测试越像工程，越不像手艺。全链路压测、混沌工程、A/B 测试，靠的都是平台、数据、算法，不是某个人的经验。手艺的毛病是人一走就没了。工程的规矩是换谁来都跑得动。你现在写的每一份报告、每一条复现步骤，都在往工程那边挪一步。

>

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。这三句话，答辩的时候一定会被问。这一节讲的三种口径、分母陷阱、n=2 的叶子，说的都是同一件：结论得站得住。

>

下一次课进第 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。这一章会用上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，一个都跑不掉。带一个问题回去想：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？

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，但不够。授权是外部约束，很多伤害恰恰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"——进一层，把标准从"允不允许"换成"值不值得"。

- 答"用之前问自己：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，只要一个公开检验。

- 收口：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，但答辩常问"你在项目里怎么保证测试结论可信"。可以用本章三句话作答：判据明确、过程可复现、结论可复算。另一常问的是"专项测试和通用测试怎么分工"，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。

> **【思政落点 22】**：这一章教了性能压测、漏洞攻防、混沌注入、A/B 分流。这四样换个场合用，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。能力越强，越要多问一句：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。

> **过渡：**本章到此结束。下一次课，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 22 处）

序号	所在页	落点主题
1	7.6	泄漏最值得琢磨的地方，是每一次都很小。一个请求漏几十 KB，跑一小时看不出来，跑一
2	7.18	可测试性背后是一条职业底线。指标定两秒，实测 2.8 秒，报告里写成"基本满足用户感
3	7.34	性能分析最像科研的日常：提假设、找证据、做对照实验、被数据推翻、再来一轮。它教人
4	7.39	实验报告里每个数字都要有出处，这句话不只是技术要求。它说的是：你写下的结论，得有
5	7.42	授权这件事，落到的是规矩意识。同一套技术，越界一步性质就变了，"我没恶意"从来不
6	7.46	无意造成的漏洞占绝大多数，这条事实指向的是一种职业习惯。少一次校验、漏一个边界值
7	7.68	合规是地板，不是天花板，这句说的是标准和自觉的关系。等保测评过了，只证明你踩到了

渗透这门手艺天然有两面。同一套手法，用来找破口是测试，用来撬别人的系统就是入侵，

一个 return 漏在异常分支里，看着是笔误，代价是别人的数据被看到。这六条检查单

同一条 payload，写用例的人拿它证明系统能被绕过，写代码的人拿它知道"拼接"

靶场和别人的系统之间，只隔着一个授权。同一条攻击链，放在自己的靶机上是练习，放到

库里那批数据不是测试数据。它们是用户几年攒下来的订单、病历、成绩、流水，没有"重

这套流程里最不显眼的是校验阶段那三件小事，最容易被砍的也是它们。可"先小流量、再

概览写得好不好，考的是诚实。数据不好看的时候，最容易做的事是挑一条好看的曲线放上

爆炸半径讲的是敢试，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，是我知

从灾难中学习的前提，是允许把问题讲出来。复盘最怕两件事：开成追责会，和把没出事当

爆炸半径讲的是对风险的估计，不是对自己能力的怀疑。做工程的容易有这种自信：这套系

韧性是别人写的代码在替你兜底，而兜底的东西最容易被人图省事拆掉。把故障注入挂到每

数据不会自己说话，是人在解释它。口径、样本量、置信区间、不利指标，一起报，才算把

单个指标涨了，不等于事情做对了。注册率提八个点，三个月后付费率掉下去，报表上好看

平台好不好用，接入方最有发言权。你多写一份示例、多留一个默认值，别人就少熬一个通

这一章教了性能压测、漏洞攻防、混沌注入、A/B 分流。这四样换个场合用，就是打垮一