《软件测试课程设计》讲稿（第 12 章）

第 14 周 · 2 学时

广东金融学院 计算机学院 | 周宇文

讲稿可照读版 | 由课件逐页撰写

第 12 章　部署测试基础设施

第 14 周 · 2 学时（90 分钟）

《软件测试课程设计》第 12 章 讲稿（第 14 周 · 2 学时 90 分钟）

> 课程：软件测试课程设计（154442008） | 广东金融学院 计算机学院 | 专业选修
>
> 章节：第 12 章 部署测试基础设施 —— 没有地基，测试能力就是一句空话
>
> 学时：**2 学时（90 分钟）**
>
> 教材：朱少民《软件测试方法和技术（第 4 版）》第 3 篇 软件测试项目实践
>
> 配套课件：逐页还原版 **47 页**（`lecture12/`，云端 http://111.231.0.226:2345/lecture12/index.html）
>
> 教案对应：第 14 次课「部署测试基础设施」
>
> 本讲重点：测试基础设施的定义与三重价值（测试的基础、影响研发效能、保障运维）；六类要素（硬件、网络、软件、数据、工具链、环境管理）；测试环境分类与一致性；测试数据管理（造数、脱敏、版本、回滚）；环境即代码；资源调度与成本；四个常见坑
>
> 本讲难点：为什么要区分开发/测试/预发/生产四套环境、以及"一致性"到底要多一致；测试数据为什么比环境更难管

> **本讲稿为可照读完整版**：每页含【口播】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出测试基础设施的三重价值；② 能列出六类要素并各举一例；③ 能说明四套环境的用途与差异点；④ 能说出测试数据管理的四项要求；⑤ 能解释"环境即代码"解决什么问题
能力	① 能为一个课程设计项目写出环境与数据需求清单；② 能识别一次"环境漂移"导致的测试事故并给出整改方案；③ 能设计一份脱敏造数方案
素质（思政）	① 基础设施是"看不见的地基"＝严谨与责任心；② 数据脱敏＝隐私与合规底线；③ 环境一致性＝对结果可信负责；④ 成本意识＝务实节约

本讲要建立的一个总印象：测试做不动、做不准、做不快，八成不是"人不行"，而是**地基不行**——环境不稳、数据没有、工具零散。这一章讲的就是怎么把地基打起来。

二、时间分配总表

> **容量提示**：本章课件 **47 页**，讲稿为素材全量版（口播约 2.1 万字，≈88 分钟），**90 分钟基本可按页讲完**。

时段	页码	内容	分钟	备注
第 1 节	12.1–12.10	基础设施的定义与三重价值、要素总览、硬件与网络	26	**三重价值必讲**
第 1 节	12.11–12.26	软件资源、数据资源、工具链、环境管理	24	当堂列需求清单
——	——	**课间休息 10 分钟**	——	——
第 2	12.27–	环境搭建与一致性、环境即代码、数据管理	22	**数据管理必讲**

节	12.40			
第 2 节	12.41–12.47	资源调度与成本、常见坑、小结与作业	18	**四个坑必讲**

> **压缩优先级**：先压缩产品与工具截图页 → 再压缩硬件规格细节 → 最后才压缩三重价值、环境分类、数据管理、四个坑。

三、逐页讲稿

> 每页第一段是可直接朗读的口播；其后是板书、提问与预设回答、易错点/考点、过渡语。
>
> 页码锚点与课件页一一对应（12.1–12.47）：http://111.231.0.226:2345/lecture12/index.html

【12.1】软件测试方法和技术（1 分钟）

> 各位同学，我们进入第 12 章——**部署测试基础设施**。
>
> 前 11 章，我们一路在讲"怎么测"：怎么设计用例、怎么选方法、怎么走流程、怎么做单元测试集成测试系统测试，第 7 章专门攻了性能和安全这些硬骨头，第 11 章把用例当成团队的资产来维护。讲到这儿你可能有一种感觉：方法我都会了，可为什么一进实验室，半天跑不起来一条用例？
>
> 原因常常不在方法，在**环境**。这一章就解决这件事：**给测试找一个站得住的地方**。
>
> 我给这一章一个定位：前面讲的是**测试的脑子**，这一章讲的是**测试的地基**。地基不牢，脑子再好也用不上劲。很多团队不是不会测，是**没地方好好测**——机器不够、环境不对、数据没有、依赖连不上，四条里占一条，测试就得靠人情和加班硬撑。
>
> 带着一个问题听下去：**你上一次因为环境问题白耗了半天，耗在哪一步？** 这一章讲完，你要能自己回答这个问题，而且知道下一步该怎么改。

> **板书：**中间写"第 12 章 部署测试基础设施"；左侧写"第 2–11 章：怎么测（概念・方法・流程・各级测试・专项・用例）"；右侧写"第 12 章：在哪测、拿什么测（地基）"；底部一行："地基不牢，方法用不上劲"。

> **提问：**回想一次你被环境问题卡住的经历——是装不上、连不通，还是数据跟别人打架？

预设回答：

- 答"数据库连不上、端口被占、依赖服务没起来"——这是最典型的一类，本质是**资源与配置**问题，本章 12.2 的软件资源、网络资源就是讲它。

- 答"两个人同时用一个测试环境，我把他的数据冲了"——这是**共享环境冲突**，本章会反复回到这个坑。

- 答"环境问题属于运维，不该在测试课里讲"——这是最需要纠正的误解：测试人员每天有一半时间花在环境上，你不掌握它，就只能等人来救。

- 收口：这三种回答，正好对应本章的三条主线——**要素、搭建、自动化**。

易错点 / 考点： ① 本章讲的是"测试能不能跑起来"的条件，不是"测得对不对"的方法，别和前面章节混；② 记住本章四个板块的名字：重要性、要素、虚拟化技术、自动部署；③ 判断题常见陷阱："测试基础设施就是买服务器"——它还包括网络、软件、数据，以及把它们组织起来的管理与自动化。

过渡： 先把这一章的地图铺开，你才知道每一步走到哪儿了。

【12.2】12.1 测试基础设施的重要性（1 分钟）

这一章分四块，我按顺序点一遍，你记结构就行。

12.1 测试基础设施的重要性——先解决"为什么值得花钱"。它会讲定义、讲它是测试的基础、讲它怎么影响研发效能、又怎么保障运维。这一块是**观念**，别小看，观念不通，后面所有投入都会被当成浪费。

12.2 测试基础设施的要素——拆开看它由什么组成：硬件资源、网络资源、软件资源、数据资源。这一块是**清单**，你要能在脑子里列出来。

12.3 虚拟化技术的应用——这是**手段**：虚拟机、QEMU-KVM、容器与 Docker、集群与 Kubernetes、服务虚拟化。为什么单独给一节？因为今天的测试环境基本不靠"再买一台机器"来解决，靠的是把资源切出来、复制出来、编排出来。

12.4 测试基础设施的自动部署——这是**落地**：基础设施即代码、基础架构自动部署、应用容器化、自动配置、CI/CD 流水线。落到最后就是一条能自动跑起来的流水线。

四块连起来是一句话：**先想清楚它是什么、由什么组成，再用虚拟化把它造出来，最后用代码把它自动化。**这也是本章的答题骨架。

> **板书：**竖排四块：12.1 重要性（为什么）→ 12.2 要素（有什么）→ 12.3 虚拟化（怎么造）→ 12.4 自动部署（怎么自动）。右侧一行："为什么 → 有什么 → 怎么造 → 怎么自动"。

提问：如果只能给公司提一条改善测试环境的建议，你会提"多买机器"还是"把环境自动化"？为什么？

预设回答：

- 答"多买机器"——短期有效，但如果环境靠人手配，机器越多、配置差异越多，问题只会更乱。

- 答"自动化"——方向对：机器是**一次性投入**，自动化是**把每次重建的成本降到接近零**，本章 12.4 讲的就是这个。

- 答"先看现在卡在哪"——最成熟：先定位瓶颈是资源不足、还是环境不可重复，再决定花钱方向。

- 收口：这一章反复出现的判断标准是——**能不能随时重建出一个一模一样的环境**。

易错点 / 考点：① 四个板块的名称与顺序要能默写；② 12.3 的标题是"虚拟化技术"，不是"云计算"，两者范围不同；③ 简答题若问"本章讲了哪些搭建测试环境的技术"，答虚拟机、容器、集群管理、服务虚拟化四类，别漏服务虚拟化。

过渡：从 12.1 开始，先回答那个最朴素的问题——它到底是什么。

【12.3】12.1 测试基础设施的重要性（1 分钟）

这一页是 12.1 节的地图，四个小节，一条逻辑链。

12.1.1 定义——先把话说准：它到底包含什么。定义不清，后面讨论就会跑偏，有人以为它等于机房，有人以为它等于测试工具。

12.1.2 是测试的基础——讲它在软件生命周期的每个阶段起什么作用，以及一个硬要求：系统测试和验收测试的环境必须最大限度接近真实，也就是**准生产环境**。这一小节还会讲一句狠话：用错环境会引起一系列问题。

12.1.3 影响研发效能——从"快不快、省不省"的角度算账：能不能早点测、环境稳不稳、资源利用率高不高、依赖能不能模拟、能不能并行跑。

12.1.4 保障运维——把它从研发侧延伸到生产侧，讲 DevOps 那条完整的链。

这条链的走向很清楚：**先定义清楚 → 再说它支撑测试 → 再算它带来的效率 → 最后看它对线上运维的价值**。前两小节讲"对不对"，后两小节讲"值不值"。

板书：四行：定义（是什么）→ 是测试的基础（支撑什么）→ 影响研发效能（快不快・省不省）→ 保障运维（线上稳不稳）。右侧标注："前两节讲对不对，后两节讲值不值"。

提问：为什么讲"重要性"要单独占一节四页？直接把技术讲完不行吗？

预设回答：

- 答"因为要向领导申请预算"——很实在，说不清价值就申请不到资源，这一节就是帮你写申请材料的逻辑。

- 答"因为很多人以为它不重要"——说中了。测试环境常被当成"后勤"，出事才想起它。

- 答"重要性属于空话，考试不考"——错，简答与案例分析常从"为什么"切入，比如"某团队测试进度一再延期，请从基础设施角度分析原因"。

- 收口：这一节的四页，就是四类标准答案的素材库。

易错点 / 考点：① 四个小节的标题要记准，尤其"影响研发效能"和"保障运维"这两个提法；② 案例题常见问法："测试总是延期，可能的基础设施原因有哪些"——从环境不可用、环境不一致、数据缺失、依赖不可控四个方向答；③ 别把"重要性"答成"必要性"的空话，要落到**成本、效率、风险**三本账上。

过渡：先把定义说准，这是后面所有讨论的地基。

【12.4】12.1.1 测试基础设施的定义（2 分钟）

课件给的定义是——软件测试基础设施，是指**支持测试运行、支持测试开发、支持测试管理，以及与研发环境、运维环境集成的综合性平台**。这句话里藏了四个对象，我们挨个拆。

第一，**支持测试运行**：有机器、有网络、有被测系统、有依赖服务，测试能真跑起来。这是最容易被想到的一层。

第二，**支持测试开发**：写自动化脚本、调试用例、管脚本代码、跑调试的那套东西。今天测试人员大量时间在写代码，没有代码托管、没有调试环境，自动化就无从谈起。

第三，**支持测试管理**：用例库、缺陷库、需求库、测试报告、流水线看板。它管的不是机器，是**信息**——谁测了什么、哪个版本、结果如何、缺陷到哪一步了。

第四，**与研发环境、运维环境集成**：它不能是一个孤岛。开发提交代码要能自动进测试环境，测试通过的版本要能对上生产部署的那个版本。孤岛式的测试环境，是"测的不是上线的那一版"这种事故的温床。

注意定义最后三个字——**平台**。平台的意思是：它是被共建、被维护、被持续改进的一整套东西；不是某个人临时凑的几台机器。课件紧接着给了结论：**测试基础设施是测试执行的基础**。

我把这句话翻译成日常场景：你要测一个登录，得有能访问的地址、能登录的账号、能看日志的地方、能查数据库的权限。这四样缺一样，用例就只能停在纸面上。工地上最不起眼的是脚手架和水电，可没有它们，砖一块也砌不上去——测试基础设施就是这个角色。

板书：定义拆四块：支持测试运行（机器・网络・被测系统・依赖）｜支持测试开发（脚本・调试・代码管理）｜支持测试管理（用例・缺陷・报告・看板）｜与研发运维集成（进得来・对得上）。底部："是平台，不是几台机器"＋"测试执行的基础"。

提问：一个只有测试机器、没有任何管理系统的环境，算不算合格的测试基础设施？

预设回答：

- 答"算，能跑测试就行"——能跑，但撑不住：结果记在哪、缺陷谁跟、哪个版本测过，全靠人脑和聊天记录，规模一大就崩。

- 答"不算，缺测试管理和集成"——按定义确实缺两块，尤其"与研发运维集成"，缺了就会出现"测的版本和上线的版本不一致"。

- 答"看团队规模，小团队不需要"——有一定道理，但边界要说清：可以**简化**，不能**没有**，最小规模也得有版本记录和缺陷登记。

- 收口：判断标准不是"有没有买系统"，而是**信息有没有闭环**。

易错点 / 考点：① 定义的四个"支持/集成"要能背下来，简答常直接考；② 高频错答：把测试基础设施等同于"测试环境"——环境只是它的一部分，它还包括开发支持、管理支持与集成；③ 记忆口诀：**跑得起来（运行）、写得下去（开发）、管得清楚（管理）、对得上号（集成）**。

【思政落点 1】：脚手架没人夸，可它一塌全工地停摆。测试基础设施就是软件工程里"看不见的地基"——它做好了，功劳记在功能上线；它做差了，责任落在测试头上。愿意把功夫下在这种不显眼的地方，靠的是严谨和责任心：把日志留全、把配置写清、把环境说明书写明白，这些没人检查的细节，恰恰是别人能不能接手你工作的关键。

> **过渡：**定义清楚了，接下来看它在生命周期的每个阶段到底起什么作用，以及那句"用错环境会引起一系列问题"到底指什么。

【12.5】12.1.2 测试基础设施是测试的基础（3 分钟）

第一句：**测试基础设施贯穿了软件开发的各个阶段，每个阶段中它对测试的影响是不一样的**。单元测试阶段，你要的是能跑起来的编译环境、能跑覆盖率工具的运行时、能 Mock 掉外部依赖的框架——一台普通开发机就够，要求是**快**。集成测试阶段，你要的是多个服务能互相调通，要有消息队列、要有数据库、要有注册中心，要求是**通**。系统测试和验收测试阶段，要求突然变成**像**——像真实的生产环境。

第二句就是那个硬要求：**在系统测试和验收测试活动中，测试环境必须最大限度地接近实际环境，因此被称为准生产环境或者类生产环境**。"最大限度接近"具体接近什么？四个维度：**同样的软件版本组合**（操作系统、中间件、数据库版本对齐）、**同量级的数据规模**（生产一千万条，你拿一百条测，执行计划完全不同）、**同样的网络与安全策略**（限流、防火墙、证书、代理）、**同样的部署形态**（集群几台、负载均衡怎么配）。

第三句：**在软件测试中使用错误的测试环境，会引起一系列问题**。哪些问题？第一类，**假阳性**——环境配错了导致用例失败，你花两天查代码，最后发现是配置问题；第二类，**假阴性**——环境太干净、数据太少，本该暴露的性能和并发问题一个都没出来，测试全绿，上线就炸；第三类，**结论不可信**——同一批用例在两个环境跑出两个结果，团队开始怀疑用例本身。

业界常按用途分四级：**开发环境**（开发者自己调，随时可改）、**测试环境**（测试团队专用，版本受控）、**预发环境**（也叫准生产，与生产同构，上线前最后一道）、**生产环境**（真实用户）。四级的边界必须清楚：**开发环境允许乱，生产环境一个字都不能乱，预发环境是两者的桥**。很多团队出事，就是把测试环境当开发环境用——谁都能改配置，最后没人知道当前状态，这就是**环境漂移**。

> **板书：**三句原文＋加厚：① 贯穿各阶段，影响不同（单元要快・集成要通・系统验收要像）；② 准生产四对齐（版本・数据量・网络与安全・部署形态）；③ 错环境三后果（假阳性・假阴性・结论不可信）。下方四环境链：开发 → 测试 → 预发（准生产）→ 生产。

> **提问：**测试环境用生产的一百条样例数据跑通了全部用例，能不能说明系统上线没问题？

> **预设回答：**

- 答"能，功能都对了"——这是最典型的错答。一百条数据跑不出索引失效、跑不出分页深翻、跑不出并发锁等

待，规模一上来结论就翻。

- 答"不能，数据量差太多"——抓住了要点：数据规模是"准生产"的四条对齐之一，缺了它，性能与稳定性结论不可信。

- 答"要看测什么，功能测试可以，性能不行"——这个回答最准确：判据不同，对环境的接近度要求就不同。

- 收口：所以"环境够不够用"没有统一答案，要问"**你要下什么结论**"。

> **易错点 / 考点：**① "准生产环境/类生产环境"这个术语要能写出，并说清它服务的是系统测试与验收测试；② 高频判断："测试环境应与生产环境完全一致"——错在"完全"，课件用词是"最大限度接近"，成本上也不可能完全一致；③ 简答三步法答"错环境的危害"：先说**结果不可信**（假阳性/假阴性），再说**返工成本**，最后说**决策失真**。

> **【思政落点 2】**：环境一致性说到底是"对结论负责"。同一批用例在两个环境跑出两个结果，最省事的做法是挑好看的那个报上去——这一步走偏，后面所有决策都建在假数据上。做测试的底线是：**结果从哪个环境来、版本是什么、数据多大，都要如实写在报告里**。不修饰、不挑选、不隐瞒不利结果，这既是职业诚信，也是让团队敢信你的唯一办法。

> **过渡：**环境是"测得准"的前提，下一页换个角度，看它怎么决定"测得快"。

【12.6】12.1.3 测试基础设施影响研发效能（3 分钟）

> 这一页讲效率，课件给了五条，后面还有省略号，说明这类好处还能继续列。
>
> 第一条，**提供测试工具和环境支持早期测试活动，让软件测试能够尽早开始**。这句话背后是"测试左移"的成本账：缺陷发现得越晚，修复代价越高。可"尽早开始"不是喊口号，它需要条件——代码一提交就能自动构建、能自动跑单元测试、能有环境部署这个中间版本。
>
> 第二条，**提供稳定、可靠的测试环境**。"稳定可靠"这四个字最贵。环境不稳，测试人员的日常就变成"重跑一遍看看是不是环境问题"，一天的有效工作时间可能只剩一半。更糟的是它会训练出一种坏习惯：**看到失败先怀疑环境，而不是先怀疑代码**——这种心态一旦形成，真缺陷就从眼皮底下溜走了。
>
> 第三条，**利用虚拟化技术和云平台提高各种软、硬件资源的利用率**。一台物理机的算力，切成多个虚拟机或容器，按需分配；测试高峰时开出来，闲时释放。

第四条，**利用 Mock 工具等服务虚拟化技术模拟外部依赖对象和系统，提供完整、可靠的自动化测试环境**。这条尤其重要，因为今天几乎没有系统是孤立的：你要调支付、调短信、调征信、调第三方地图。这些依赖你不能随便压测、不能造异常、还可能按次收费。服务虚拟化让你把它们"假装"出来——12.3.5 会专门讲。

第五条，**提供并行测试能力，缩短测试执行需要的时间**。同一批用例拆成十份同时跑，一天变两小时。但要注意，并行不是免费的：用例之间不能互相踩数据，环境要能隔离，否则并行度越高、冲突越多。

五条连起来其实是一句话：**基础设施决定了测试的"起跑线"和"上限"**。起跑线是能不能早点开始，上限是能不能快、能不能大规模重复。省略号那部分你可以自己补：可观测性、可追溯、团队协作，都算。

> **板书：**五行：① 早期测试（左移的条件）② 稳定可靠（不训练"怀疑环境"的坏习惯）③ 虚拟化提利用率 ④ 服务虚拟化模拟依赖 ⑤ 并行测试缩短时间。右侧："基础设施决定测试的起跑线与上限"。

提问：团队说"我们自动化用例写了两千条，但每天只跑得完一次"，问题最可能出在哪？

预设回答：

- 答"用例写太多了，删一些"——方向错了，删用例是拿覆盖率换时间。

- 答"没有并行能力，环境也不够"——正解方向：要么并行拆分，要么加执行节点，要么把慢的资源（数据库、第三方）虚拟化掉。

- 答"机器不够就买机器"——买机器能解决，但如果用例之间抢同一份数据，加机器也提不了速，得先解决**隔离**。

- 收口：性能问题先看瓶颈在哪一环：调度、环境、数据依赖，还是用例本身太慢。

易错点 / 考点：① 五条要能说全，简答常考"测试基础设施如何提升研发效能"；② 特别记牢第四条的关键词"服务虚拟化/Mock"，它与 12.3.5 呼应；③ 选择题陷阱：把"并行测试"说成"分布式测试工具的能力"——课件强调的是**基础设施提供并行能力**，工具只是使用者。

过渡：研发侧讲完了，这一页的价值还有另一半——它对线上运维意味着什么。

【12.7】12.1.4 测试基础设施保障运维（2 分钟）

这一页只有一句话，但信息量很大：测试基础设施**不仅用来支撑研发环境，而且延伸到生产环境，成为支撑产品运维的重要组成部分，从而构成一个贯穿研发和运维的完整的 DevOps 测试基础设施**。

关键词有三个：**延伸**、**支撑运维**、**DevOps**。

先说"延伸"。传统观念里，测试环境是研发的后院，上线就与它无关。今天不是这样：线上的监控告警、日志采集、链路追踪、灰度发布、回滚脚本，用的往往是同一套工具链、同一套配置方式。你在测试环境里没练过监控，生产出事时你连在哪看指标都不知道。

再说"支撑运维"。运维最怕两件事：**不知道自己有什么**和**不知道什么时候坏的**。前者靠配置管理和资产清单，后者靠监控告警。这两件事在测试阶段就该建立起来——环境用代码描述，资产清单自动生成；监控指标在测试环境就接好，生产只是换个地址。

最后说"DevOps"。这个词的核心是打通开发与运维的墙，让一次变更从提交到上线是一条自动流水线，而不是一串邮件和工单。课件给的定语是"**完整的 DevOps 测试基础设施**"——注意"测试"两个字还在，意思是：这条流水线里，测试不是最后一道关卡，而是每一环都嵌着的质量门。

我给一个判断标准：**如果一套基础设施只能让研发用、运维完全用不上，那它只做了一半**。真正成熟的形态是——同一份环境定义，既能拉起测试环境，也能拉起预发和生产；同一套监控模板，测试环境和生产环境通用。这样"测试环境里没发现的问题"才会真正变少。

板书：一条横线贯穿两段：研发环境 ←→ 生产环境；中间三个关键词：延伸・支撑运维・DevOps。下方写"两个怕"：不知道自己有什么（配置与资产清单）／不知道什么时候坏（监控告警）。结论："只给研发用的基础设施，只做了一半"。

提问：为什么说"测试环境里没发现问题"有时不是测试不认真，而是基础设施只覆盖了研发侧？

预设回答：

- 答"因为线上才有的问题，测试环境造不出来"——对，比如真实流量特征、真实网络抖动、真实数据分布，这些需要"准生产"和流量回放（12.2 会讲）来补。

- 答"因为监控没接，问题发生了没人看见"——这是"保障运维"这一条的直接体现：可观测性也是基础设施的一部分。

- 答"测试就该保证线上不出事，出事就是测试的责任"——过于绝对。测试能保证的是"在给定环境和范围内验证过"，环境与生产差距越大，这份保证的边界就越窄——所以要把边界**写清楚**。

- 收口：这也是为什么本章后面反复强调"环境即代码"——只有同源，才能同构。

易错点 / 考点：① 记住"DevOps 测试基础设施"这个完整提法，它把测试基础设施的边界从研发扩到运维；② 常考判断："测试基础设施只服务于测试团队"——错；③ 案例题常从"测试环境通过、生产环境故障"切入，标

准答法包含三段：环境差异（版本/数据/网络）、可观测性缺失、回滚与灰度能力不足。

过渡：知道它重要了，接下来把它拆开——它到底由哪几样东西组成。

【12.8】12.2 测试基础设施的要素（1 分钟）

这一节回答"由什么组成"，课件给了四类要素：**硬件资源、网络资源、软件资源、数据资源**。

我先把这四类的关系点出来，不然你会觉得它们是并列的四张清单。其实是**层层依赖**：硬件是地基（有没有机器），网络是血管（机器之间通不通），软件是器官（机器上跑什么），数据是血液（跑起来处理什么）。任何一层缺失，上面那层就白搭——你机器再多，网络不通，集群起不来；软件装得再全，没有数据，用例跑出来全是空结果。

为什么要素要单独讲？因为**要素就是检查表**。以后你评估一个团队的环境能力，就按这四类打勾：硬件够不够、网络通不通、软件版本对不对、数据有没有且合不合规。四栏打完，短板在哪一目了然。

顺便提醒一句这四类最容易忽略的那一类：**数据**。硬件和网络是采购问题，软件是安装问题，数据是**长期治理问题**——它不像机器那样能一次性买齐，得持续造、持续脱敏、持续维护版本。这也是为什么 12.2.4 之后还要专门讲流量回放。

板书：四层塔：数据（血液）→ 软件（器官）→ 网络（血管）→ 硬件（地基）。右侧："要素＝检查表：够不够・通不通・对不对・有没有且合规"。

提问：这四类要素里，哪一类最容易被团队忽略？为什么？

预设回答：

- 答"数据"——最常见也最正确：机器能买、软件能装，数据得一点点造和治理，还涉及合规。

- 答"网络"——也有道理，尤其是跨机房、跨云、专线和安全策略，往往是"环境通了但连不上"的真凶。

- 答"硬件，因为云上一点就有了"——恰恰相反，硬件最容易解决，所以它最不该成为瓶颈；如果你的瓶颈还在硬件，说明前三个问题没解决就先加机器了。

- 收口：四类都要打勾，短板决定环境能力的下限。

易错点 / 考点：① 四类要素的名称与顺序要能默写：硬件、网络、软件、数据；② 高频错答：把"测试工具"单列成第五类要素——课件把它归在**软件资源**里；③ 简答可用"地基—血管—器官—血液"这个比喻收尾，

好记也不容易漏。

过渡：从最底层开始——硬件到底有哪些形态可选。

【12.9】12.2.1 硬件资源（3 分钟）

硬件资源这一页课件列了四种形态：**机架式服务器、刀片式服务器、传统式服务器、云真机**。前三种是"自己的机器怎么摆"，第四种是"别人的机器怎么租"。我按选型的角度讲，因为你们以后真正要做的决定是"选哪种"。

传统式服务器，也就是我们常说的塔式机，一个机箱一台机器，像台式机放大版。优点是便宜、简单、插上电就能用，噪音大、放办公室都行；缺点是密度低、占地方、电源和散热不共享。它适合什么？适合**小团队的第一台测试机**，或者实验室里给学生用的机器。

机架式服务器，标准宽度能装进机柜，按"几 U"算高度。它的意义是**可规模化**：一个机柜里整齐排十几台，供电、散热、布线都规范，机房能管得过来。中型团队的自建测试机房基本都是这个形态。

刀片式服务器，一个机箱里插多块"刀片"，每块刀片是一台独立服务器，但**共享机箱的电源、风扇、背板和管理模块**。好处是密度最高、布线最干净、管理统一；代价是机箱是单点，机箱坏了整框受影响，而且不同厂商的刀片箱基本不通用，采购容易被绑定。

第四种要重点说：**云真机**。它不是普通服务器，是**跑在云上的真实手机/真机设备**，通过远程连上去操作。为什么测试需要它？因为移动端测试的死结是"机型太多"——你要覆盖不同厂商、不同系统版本、不同屏幕、不同芯片的真实表现。自己买一柜子手机，成本高、折旧快、还管不过来；云真机按需租用，想测哪台连哪台。这是"硬件资源"在移动时代最典型的形态变化。

选型我给三步：**先看规模**（几台还是几十台）→ **再看隔离要求**（能不能共用、要不要独占）→ **最后算总成本**（采购＋电费＋机房＋人力，对比按需租用）。别只比机器单价，机房的电和人是持续的。

板书：四种形态两列对照：自有（传统塔式＝便宜简单・密度低｜机架式＝可规模化・机房标准｜刀片式＝密度最高・共享电源散热・机箱单点）／租用（云真机＝真实机型覆盖・按需・免折旧）。底部选型三步：看规模 → 看隔离 → 算总成本。

提问：一个只有五个人的测试团队，要测一款安卓 App，你建议先买服务器还是先租云真机？

预设回答：

- 答"先租云真机"——多数情况下更合理：机型覆盖是刚需，自购手机折旧快、管理成本高，团队小更需要"用多少租多少"。

- 答"先买服务器，机器在自己手里踏实"——可控性确实更好，但要算总账：五个人维护机房不现实，电费、值守、备件都是隐性成本。

- 答"都要，反正预算够"——预算够也要分优先级，先把**当前最大的瓶颈**解决掉，别一次铺开。

- 收口：硬件选型的本质不是"买好的"，是"买对的"，对的标准是**当前瓶颈**。

易错点 / 考点： ① 四种形态的名称要能写全，尤其"云真机"这个提法，考试常作为"移动测试的硬件资源"出现；② 高频混淆：把"刀片式"和"机架式"混为一谈——区别在**是否共享机箱的电源与散热**；③ 常考判断："自建机房一定比云上便宜"——错，规模小的时候云更省，规模大到一定程度自建才划算。

过渡： 机器有了，机器之间还得通——下一页看网络资源为什么是独立的一类要素。

【12.10】12.2.2 网络资源（3 分钟）

网络资源这一页课件列了四组：**100M／1G／10G 网、局域网／广域网／无线网、网络协议、防火墙／代理服务器或网关**。这四组分别对应四个问题：多快、多远、怎么通、能不能过。

第一组**带宽**，100M、1G、10G，说的是速率。为什么测试要在意带宽？因为很多"性能问题"其实是**带宽问题**：大文件上传下载、视频流、批量数据同步，卡的不是 CPU 而是管道。你在千兆内网测出来的上传速度，放到用户的实际网络里完全不成立。

第二组**覆盖范围**：局域网（同一机房或同一办公网，延迟极低）、广域网（跨地域，延迟和丢包都上来了）、无线网（WiFi 与移动网络，抖动大、切换多）。这一组提醒你：**测试环境所在的位置，决定了你能验证什么**。测试环境全在一个机房内网，你就永远测不出跨地域调用的超时问题。

第三组**网络协议**：HTTP/HTTPS、TCP/UDP、DNS、以及各种内部协议。测试要关心协议，因为协议决定了你怎么抓包、怎么造异常、怎么判断是握手失败还是应用层报错。安全测试更是离不开它。

第四组**防火墙、代理服务器或网关**：这一组最容易被当成"运维的事"，其实是环境问题的高发区。常见故障就三类：**端口没开**（服务起了但连不上）、**代理没配**（访问外网全超时，本地却有网）、**网关做了地址转换**

（日志里看到的来源 IP 全是网关的，排查时分不清是谁）。我在教学中见过太多次"环境不通"，最后一查是防火墙策略。

所以网络资源在测试里的用法有两面：**一面是保障连通**（该通的一定要通，且要写在文档里），**另一面是主动制造不通**（限速、丢包、断网、延迟，用来测系统的健壮性）。第二面常常被忽略，但它才是"网络资源"作为测试手段的价值。

板书：四组对应四问：带宽（多快：100M/1G/10G）｜覆盖（多远：局域网・广域网・无线网）｜协议（怎么通：HTTP・TCP/UDP・DNS）｜边界设备（能不能过：防火墙・代理・网关）。下方写两类故障："端口没开・代理没配・地址转换"；以及"主动制造不通：限速・丢包・断网・延迟"。

提问：服务在本机 curl 一切正常，同事的机器访问超时，你会先查哪三样？

预设回答：

- 答"先查防火墙和端口"——对，第一嫌疑：服务监听的是 127.0.0.1 还是 0.0.0.0，端口有没有对外放行。

- 答"先查代理设置"——也很常见，尤其跨网段访问时，代理白名单里没加这个网段，请求全被拐走。

- 答"先重装服务"——最不该做的第一步：现场还没看清就动手，等于把线索抹掉。

- 收口：先分层定位——**本机通不通 → 同网段通不通 → 跨网段通不通**，一层层缩小范围。

易错点 / 考点：① 四组网络资源要能列全，简答常考"网络资源包括哪些"；② 常考判断："网络资源只指带宽"——错，还包括覆盖范围、协议与边界设备；③ 记住网络资源的两面性：**保障连通**与**主动制造异常**，后者是可靠性测试的基础手段。

过渡：网络通了，机器上还得有东西跑——软件资源这一页开始进入"装什么、装哪个版本"的世界。

【12.11】12.2.3 软件资源（3 分钟）

软件资源这一页列了六项：**操作系统、数据库、Web 服务器、测试工具、应用软件、被测试系统**。前五项是"舞台"，最后一项是"演员"，我分开讲。

操作系统：Windows、各类 Linux 发行版、macOS、以及移动端的 Android 和 iOS。测试要在意操作系统的**版本**，不只是类型。同一个中间件在不同内核版本上行为可能不同；补丁级别不同，某些系统调用和文件权限行为也不同。

数据库：类型（关系型、NoSQL、缓存）、版本、字符集、时区、隔离级别。这五项里任何一项不一致，都可能造出"测试环境正常、生产环境报错"的经典事故。字符集和时区这两个坑，讲十年也还是有人踩。

Web 服务器与应用服务器：反向代理、容器、应用服务器。它们的配置决定了超时时间、连接数上限、请求体大小限制——这些参数在测试环境常常被调得比生产宽松，于是测试全过、上线就超时。

测试工具：性能工具、自动化工具、缺陷管理、用例管理、静态分析工具。这一项常被当成"个人装个软件"，其实它是团队资产：版本要统一、许可证要管理、脚本要能共享。

应用软件：字处理、浏览器、客户端。别小看浏览器——不同浏览器版本对同一段前端代码的解释可以完全不同，兼容性测试就靠这一层。

被测试系统：这一项是核心：你要测的软件本身，以及它的版本。这里最容易出的问题是**版本混乱**——测试环境上跑的是哪个构建、对应哪个代码分支、包含了哪些未合并的改动，必须能查清。我给大家一个硬要求：**测试环境里的每个部署，都要能追溯到一个确定的版本号或提交**。做不到这一点，"复现缺陷"就是一句空话。

六项合起来是一个"软件矩阵"，矩阵的组合是爆炸的——这就是为什么下一节要用虚拟化和容器把这些组合**固化下来**，而不是每次手工装。

> **板书：**六项两栏：舞台（操作系统・数据库・Web 服务器・测试工具・应用软件）／演员（被测试系统）。右侧写"版本矩阵"与一条硬要求："每次部署必须可追溯到确定版本"。

提问：测试环境里的数据库从生产同版本降了一个小版本，会有风险吗？

预设回答：

- 答"小版本无所谓"——典型错答。小版本可能改默认参数、改优化器行为、修安全补丁，执行计划和性能表现会变。

- 答"有风险，可能测出来的性能结论不成立"——对，尤其涉及索引、锁、事务隔离的场景。

- 答"要看测什么，功能测试影响小，性能和兼容性影响大"——最准确的答法，回到"你要下什么结论"这个判断标准。

- 收口：版本差异要不要管，取决于你要验证的结论对版本有多敏感。

易错点 / 考点：① 六项软件资源要能默写，注意"被测试系统"也是软件资源之一，很多同学会漏；② 高频考点：字符集、时区、连接数上限、超时参数这四类"配置差异"引发的环境问题；③ 答题口诀：**装什么（类型）—装哪个版本（版本）—怎么配（参数）**，三层都要对。

> **过渡：**软件资源里"测试工具"那一项，课件单独用一页图来讲——我们看看一个完整的持续集成工具集长什么样。

【12.12】持续集成工具集（2 分钟）

这一页是一张工具集的关系图，没有正文，但它要你看出一件事：**测试工具不是孤立的一个软件，而是一条链上的若干个环节**。

起点是代码仓库。所有环节的输入都是它——没有版本控制，后面全是手工。

第二站是构建。把源码编译、打成制品（jar、war、镜像），这一步的输出必须**可重复**：同一个提交，任何时候构建出来的东西应该一样。

第三站是自动化测试。单元测试、接口测试、回归测试在这一站跑。注意它的位置：**在打包和部署之前**，因为一旦测试挂了，就不该往下走。这就是"质量门"的含义。

第四站是制品库。构建产物不能只躺在某台机器的目录里，要进仓库、带版本号，谁部署的、部署的哪一版，都能查。

第五站是部署与通知。把制品推到测试环境，然后把结果告诉人——成功或失败、谁的责任、日志在哪。

这张图还有一个隐含要求：**每一站都要能被机器自动触发**。任何一站需要人手工点一下，整条链的节奏就被这一下卡住了。所以判断一个团队的持续集成成熟度，有个简单办法——**问他们从提交代码到测试报告出来，中间需要人工操作几次**。零次的，是真流水线；三五次的，是"半自动"；十次以上的，那就还停在手工时代。

这也呼应了 12.4：自动部署不是买一个工具，而是把这条链上每一站都接上，让它们自己往下传。

> **板书：**画一条横向流水线：代码仓库 → 构建（可重复）→ 自动化测试（质量门）→ 制品库（带版本）→ 部署 → 通知。底部一行："成熟度＝从提交到报告，人工操作几次？零次才算真流水线"。

> **提问：**如果一个团队的构建是自动的、测试是自动的，但部署要人手工执行，这条链算不算持续集成？

> **预设回答：**

- 答"算，集成本身就是构建加测试"——严格说持续集成确实以"频繁集成并验证"为核心，但只到这一步，价值只发挥了一半。

- 答"不算完整，缺持续交付/部署那一段"——这个回答更到位：链断在部署，反馈周期就被人工环节拉长。

- 答"手工部署更安全，自动部署风险大"——把"风险"和"自动化"对立起来是误解，手工部署的风险在于**每次都不一样**，自动化恰恰是把步骤固定下来。

- 收口：自动化的价值不在"快"，在"每次都一样"。

> **易错点 / 考点：** ① 要能说出工具链的环节顺序，考点常在"自动化测试应该放在构建之后、部署之前"；② 常考判断："持续集成就是持续部署"——两者层次不同，集成是验证，部署是交付；③ 记住那句可量化的判断标准："人工操作几次"，这是案例题的加分点。

> **过渡：** 工具链解决了"怎么跑"，但跑起来要有东西可处理——下一页讲最容易被忽略的数据资源。

【12.13】12.2.4 数据资源（3 分钟）

数据资源这一页列了四类：**原有数据、正确数据和错误数据、真实的客户数据、大量的数据**。这四类不是随便列的，它们对应四个不同的测试目的，我一条条说。

原有数据，指的是系统里已经存在的数据——历史订单、老用户、上一版本的配置。为什么要专门提它？因为**新版本必须在老数据上跑得起来**。很多事故是升级之后，历史数据格式不兼容、字段含义变了，老数据一读就错。所以升级类测试的第一件事，是把一份真实的老数据灌进去。

正确数据和错误数据，这是一对。只准备正确数据，你测的是"顺利路径"；错误数据才是**异常处理**的试金石——超长字符串、非法字符、边界外的数字、空值、重复值、格式不对的日期。课件把它俩并列，就是在提醒：**测试数据要成对设计**，一条正常、一条异常。

真实的客户数据，最能暴露问题，因为它带着真实世界的脏、乱、杂。但这里有一条红线，我要重点强调：真实客户数据里含个人信息、账号、联系方式、交易记录，**不能直接拿来测**。这就是**脱敏**——把姓名、身份证、手机号、银行卡、地址这些标识替换掉或做不可逆处理，保留数据的分布特征和结构，去掉可识别到人的部分。数据脱敏不是"讲究"，是**合规底线**。

大量的数据，讲的是规模。生产一千万条，测试一百条，索引是否命中、分页深翻是否变慢、批量任务要跑多久，结论完全不同。所以重要的性能与批处理测试，必须准备**同量级的数据**，这也是 12.1.2 讲的"准生产"四条对齐之一。

最后补三件数据管理上必须做的事，课件没展开但工程上必须有：**造数**（脚本批量生成，别手工录）、**数据**

版本（这批数据是哪个版本、配套哪个用例集，要能查）、**回滚**（测试污染了数据，能一键恢复到基线）。这三件事不做，测试数据用两周就会烂掉——行内叫**数据污染**。

板书：四类数据对应四目的：原有数据（升级兼容）｜正确＋错误数据（顺利路径＋异常处理，成对设计）｜真实客户数据（真实分布，但必须脱敏）｜大量数据（规模效应）。下方三件事：造数（脚本）·数据版本（可查）·回滚（可恢复基线）→ 防"数据污染"。

提问：能不能直接把生产库导一份到测试环境用？反正都是自己公司的数据。

预设回答：

- 答"能，最真实"——忽略了两件事：一是**个人信息**，测试环境权限松、日志全，数据一旦外流责任在公司；二是数据量太大时导入本身就会拖垮环境。

- 答"要先脱敏再导，而且最好只导需要的表和字段"——这是标准做法：先脱敏、再裁剪，明确谁能访问。

- 答"公司内部用不算泄露"——这个认识很危险：内部人员范围、测试机权限、备份文件流转，每一环都是泄露面。

- 收口：判断标准不是"是不是自己公司的"，而是"**这份数据能不能追溯到某个具体的人**"。

易错点 / 考点：① 四类数据要能列全并说清各自用途，简答高频；② 必背术语：**脱敏、造数、数据版本、数据回滚、数据污染**；③ 案例题常问"测试数据怎么管理"，标准答法四步：定义基线数据 → 脚本化造数 → 敏感字段脱敏 → 版本化＋可回滚。

【思政落点 3】：把生产数据拷进测试库，图的是省事，赌的却是别人的隐私。手机号、身份证、住址、交易记录，在测试环境里往往没有审计、没有权限控制，一份 Excel 转发出去就收不回来了。数据脱敏看着是技术活，实质是**把"方便"让位给"该不该"**——能不用的真实数据就不用，必须用的先脱敏，用完能删就删。守住这条线，才配接触用户的数据。

过渡：真实数据不好拿、不好用，那有没有办法既真实又安全？下一页的流量回放就是一条思路。

【12.14】流量回放技术（3 分钟）

这一页是"流量回放"的示意图，没有正文。它要你看懂一件事：**还有一条拿到真实数据的路——不拷数据，拷流量。**

传统做法是导数据库。问题很明显：数据是静态的，它只告诉你"库里有什么"，不告诉你"用户怎么用"。流量回放反过来——**它记录的是真实用户发过来的请求**：什么时间、什么参数、什么顺序、什么并发节奏。

它的工作方式分三段。**录制**：在入口处（网关、代理、负载均衡）把请求抓下来，通常是"影子"方式，不影响正常业务。**回放**：把录下来的请求按原来的节奏打到待测环境，可以是新版本、也可以是迁移后的新架构。**比对**：同一个请求，老系统返回什么、新系统返回什么，逐条比差异——差异就是嫌疑点，可能是真 bug，也可能是预期内的行为变化，要人工确认。

它最适合三个场景：**大版本重构后验证行为一致、数据库或中间件迁移后验证结果不变、回归测试补充真实用例**。共同点是：你怕的不是"功能写错了"，而是"改完之后老行为悄悄变了"。

但它有三个前提。**脱敏**：录下来的请求里有账号、手机号、token，回放前必须处理。**副作用隔离**：回放会真的下单、发短信、扣款，所以外部依赖必须换成 Mock、写操作打到影子库，否则回放一次就是一次事故。**时间与环境对齐**：请求带时间戳和外部状态，对不上，比对结果全是噪音。

> **板书：**三段：录制（网关/代理抓请求，影子方式）→ 回放（按原节奏打到待测环境）→ 比对（逐条比差异，人工确认）。三个适用场景：重构验证・迁移验证・回归补用例。三个前提：脱敏・副作用隔离（Mock/影子库）・时间与环境对齐。

提问：回放生产流量时，最危险的一步是什么？

预设回答：

- 答"比对不准"——是麻烦，但不是最危险。

- 答"回放把真实业务写了，比如真下单、真发短信"——这才是最危险的：测试流量打进了真实系统。

- 答"流量太大把测试环境压垮"——确实常见，属于容量问题，可以通过限速、抽样解决。

- 收口：所以回放的第一条工程纪律是——**写操作一律指向影子库，外部依赖一律 Mock**。

易错点 / 考点：① 流量回放的三段式（录制—回放—比对）与三个前提要能说清；② 高频判断："流量回放可以完全替代人工设计用例"——错，它只能覆盖"发生过的行为"，覆盖不了新需求、新分支、新异常；③ 与 12.2.3 的数据资源连起来答：**静态数据看结构，回放流量看行为**。

过渡：四类要素讲完了，接下来看怎么把它们"变出来"——虚拟化技术。

【12.15】12.3 虚拟化技术的应用（1 分钟）

> 这一节是本技术含量最高的一节，五个小节，我先把路线画出来。
>
> **12.3.1 虚拟机技术**——最基础的虚拟化：在一台物理机上跑多个完整的操作系统。要讲清两种类型、为什么用它、它怎么工作、有哪些软件。
>
> **12.3.2 QEMU-KVM 虚拟机解决方案**——把 Linux 上最主流的一套开源方案拆开看：QEMU 管什么、KVM 管什么、为什么两者必须合起来。
>
> **12.3.3 容器技术与 Docker**——换一种思路：不虚拟硬件，只隔离进程。要讲清 Cgroups 与 namespace 这两根支柱，以及容器为什么比虚拟机快。
>
> **12.3.4 集群管理与 Kubernetes**——一台机器不够了怎么办：把一群机器当成一台来用，讲调度、服务发现、动态伸缩。
>
> **12.3.5 服务虚拟化及其工具**——这一节最容易被忽视，但很实用：不虚拟机器，**虚拟"别人的服务"**，把不可控的外部依赖变成可控的假服务。
>
> 这五节的逻辑是**层层抽象**：从虚拟硬件（虚拟机），到虚拟操作系统（容器），到虚拟整个集群（K8s），最后到虚拟外部服务（服务虚拟化）。抽象层级越高，越贴近测试人员的日常——因为你真正要的从来不是"一台机器"，而是"一个可重复、可丢弃、可并行的测试环境"。

> **板书：**五层台阶，从下往上：虚拟机（虚拟硬件）→ 容器（虚拟 OS 环境）→ K8s（虚拟整个集群）→ 服务虚拟化（虚拟外部服务）。右侧："抽象越高，越贴近'可重复、可丢弃、可并行'"。

> **提问：**如果目标是"给每个测试人员一套独立环境"，你会从哪一层开始考虑？

> **预设回答：**

- 答"虚拟机，一人一台最省事"——可行，但资源开销大，十个人就是十套操作系统。
- 答"容器，起停快、占资源少"——多数场景更合适，但要考虑有人需要 root、需要特定内核模块时容器不够用。
- 答"看需求，先看要不要独占和改系统"——最成熟：**要改内核或独占硬件就用虚拟机，只要隔离运行环境就用容器**。
- 收口：选型的分水岭是"**你要隔离到什么程度**"。

> **易错点 / 考点：**① 五个小节的名称与顺序；② 注意区分"虚拟化技术"（虚拟机/容器）与"服务虚拟

化"（Mock 类工具），后者虚拟的不是机器；③ 简答常用问法："虚拟化对测试环境有什么价值"——答资源利用率、快速复制、隔离、可重置四点。

过渡：从最底层的虚拟机开始，先看它有哪两种类型。

【12.16】12.3.1 虚拟机技术（3 分钟）

虚拟机技术这一页给了两个要点：**两种类型的虚拟机**，以及**构建虚拟的测试农场**。

虚拟机按"虚拟化软件装在哪"分成两类。**第一类，直接装在物理硬件上**，虚拟化层本身就是个精简的操作系统，业内叫"裸金属型"，也叫 I 型。它的优点是性能损耗小、安全性高，缺点是要独占整台机器，你没法在一台已经装着 Windows 的笔记本上直接装它。**第二类，装在操作系统之上**，像普通软件一样安装，业内叫"寄居型"。它的优点是装了就能用、和你的日常系统共存，缺点是隔了一层操作系统，性能和管理都多一层开销。**开发机和教学机上用的，基本都是第二类。**

你在自己的笔记本上装一个虚拟机跑 Linux 做测试，那是寄居型；公司的服务器上虚拟出二十台测试机，那通常是裸金属型。考试问"虚拟机的两种类型"，答的就是这个区分标准——**虚拟化层与硬件之间有没有宿主操作系统**。

第二个要点：**构建虚拟的测试农场**。"测试农场"这个词来自持续集成的实践，原意是"一大片机器，像农场一样被统一种、统一收"——你把一批虚拟机整整齐齐管起来，谁要用就分配一台，用完销毁、下次重建。它的关键不是"多"，而是**标准**：每一台都从同一个模板克隆出来，系统版本、依赖库、配置全部一致。

这件事解决了一个折磨测试团队多年的问题——**环境漂移**。什么叫漂移？一台测试机用了三个月，中间有人装了新版本的库、有人改了 hosts、有人手工调了参数，它和生产环境的差距一点点拉开。到最后没人知道它到底是什么状态，缺陷也就没法复现。虚拟测试农场的答案很干脆：**机器是消耗品**。环境不对就删掉，从模板重新拉一台，五分钟回到标准状态。记住这句话——**把服务器当宠物养，还是当牛羊养，这是两种完全不同的运维哲学**；测试农场选的是后者。

板书：两种类型对照：I 型裸金属（虚拟化层直接在硬件上／性能好、安全／独占整机）｜II 型寄居（装在 OS 之上／易部署／多一层开销）。下方写"虚拟测试农场"：统一模板 → 按需分配 → 用完销毁 → 五分钟回到标准态。右侧："机器是消耗品，不是宠物"。

提问：你个人的笔记本上装虚拟机做测试，属于哪一类？为什么机房服务器不这么做？

预设回答：

- 答"寄居型，因为要先有 Windows 再装虚拟机软件"——对，判断标准就是有没有宿主操作系统。

- 答"机房也能装寄居型，只是没必要"——说得准：寄居型多一层开销，服务器追求的是把硬件算力尽量给业务，所以选裸金属型。

- 答"虚拟机就是把电脑分成好几台，类型不重要"——错在"不重要"：类型决定了性能损耗、部署方式和适用场景，是考点。

- 收口：判断三步——**虚拟化层在哪、有没有宿主 OS、性能损耗多大**。

易错点 / 考点：① 两种类型的区分标准要能一句话说清：**虚拟化层与物理硬件之间是否有宿主操作系统**；② 高频错答：把"虚拟机"说成"模拟器"——模拟器是软件层面模拟另一套指令集，虚拟机是硬件资源切分；③ 记住"测试农场"和"环境漂移"这两个术语，前者是解法、后者是问题，案例题常成对出现。

过渡：既然要用虚拟机，就得算清楚它到底给我们省了什么——下一页是四条实打实的账。

【12.17】为什么使用虚拟机？（3 分钟）

这一页回答"为什么要用虚拟机"，课件给了四条理由，而且都带数字，我们一条条看。**先声明一句：下面这些数字是课件给的行业经验值，用来讲量级，不要当成精确测量结果。**

第一条，**充分利用硬件资源**：课件说，有 70% 的服务器利用率只有 20%～30%，借助虚拟机技术可以提高到 85%～95%。为什么会空？因为一台服务器通常只跑一个应用，而那个应用的平均负载远低于峰值——你必须按峰值配机器，于是大部分时间在浪费。虚拟化把多台"低负载"的机器合成一台物理机上的多个虚拟机，空转的算力就被填起来了。请注意这里的两个数字是**利用率**，不是"性能提升 3 倍"，别答错。

第二条，**节约能源和空间**：课件举例说，如果内存加大到 16G 或更高，一台机器可以虚拟 4～8 台服务器。少 4 到 8 台物理机意味着什么？少 4 到 8 份电费、少 4 到 8 份机架空间、少 4 到 8 台机器的风扇噪音和机房空调负荷。

第三条，**提升运作效率**：几分钟就可以装载所需的系统镜像文件。这句话在测试场景里的价值极高——新来一个同事要环境，不用等他装一天系统；某个版本要复测，不用重装机器；环境被搞坏了，删了重来。**"几分钟"这个量级，是把环境准备从"一天的活"变成"一次点击"的关键。**

第四条，**有利于环境的建立和维护**：容易实现添加、移动、变更和重置服务器的操作。这四个动作正好对应测试团队的四种日常需求：加一套并行环境（添加）、把环境从这台宿主机迁到那台（移动）、调整配置或扩容（变更）、以及最常用的——**重置**。重置是测试环境最被低估的能力：一个"随时能回到出厂状态"的环境，意味着测试人员敢做破坏性操作，敢乱改配置，因为他知道代价只是一次还原。

四条合起来，其实是在回答一个选择题：**你要"更多机器"，还是"更灵活的资源"**？虚拟机给的是后者。

板书：四条账：① 利用率 20%～30% → 85%～95%（课件经验值）② 16G 内存可虚拟 4～8 台（省电・省空间）③ 几分钟装载镜像（效率）④ 添加・移动・变更・重置（易维护）。结论："要的不是更多机器，是更灵活的资源"。

提问：如果一台物理机只跑一个测试应用，负载长期只有两三成，最直接的改进是什么？

预设回答：

- 答"虚拟化，把多套环境合到一台物理机上"——对，这就是第一条讲的利用率问题。

- 答"再买一台一起跑，反正不贵"——方向反了：问题不是机器少，是**每台都在空转**，加机器只会让浪费翻倍。

- 答"把应用优化到能跑满 CPU"——技术上正确，但业务系统很难长期跑满，而且测试环境本来就不追求跑满，追求的是"够用且能并行"。

- 收口：先判断瓶颈是"算力不够"还是"算力闲置"，这两种情况的解法完全相反。

易错点 / 考点：① 课件给的 20%～30%、85%～95%、4～8 台、16G 这几个数字要记准，简答常要求"说出虚拟化提高利用率的量级"；② 高频错答：把"利用率提高"说成"性能提高"，两者是**资源使用效率**与**处理速度**两回事；③ 记住四个动作：**添加、移动、变更、重置**——尤其"重置"，它是测试环境区别于生产环境的核心能力。

【思政落点 4】：一台服务器长期只用到两三成算力，账面上看是"公司有钱"，实际上是电、机位、空调、人力全在为闲置买单。虚拟化教的是一种务实的态度：**先把手里的资源用足，再谈添置**。做工程要算账，能不花的钱不花，能复用的东西不重复造；但节约不等于抠门——该投入在可靠性和安全性上的钱，一分都不能省。分清"省该省的"和"花该花的"，这是工程师的基本判断力。

过渡：知道它省了什么，接着看它凭什么能做到——下一页进入工作原理。

【12.18】虚拟机工作原理（2 分钟）

这一页是虚拟机的原理示意图，要你看懂一句话：**虚拟机不是"模拟一台电脑的软件"，而是"把物理资源切开来分给多个操作系统"**。

看图的时候抓住三个层次。**第一层是物理硬件**：CPU、内存、磁盘、网卡，真实的、只有一份。**中间那层是虚拟化层**，也就是 Hypervisor（虚拟机监控器）。它是整个技术的关键——它站在硬件和操作系统之间，把 CPU 的时间、内存的地址空间、磁盘的扇区、网卡的口子，切成一份一份，分给上面的多个虚拟机。**再往上是多个客户操作系统（Guest OS）**，每一个都以为自己独占了一台完整的电脑，装自己的驱动、跑自己的应用、互相看不见。

那 Hypervisor 是怎么"骗"过上层操作系统的？靠的是**特权指令的拦截**。操作系统里有些指令只能在最高特权级执行，客户操作系统执行到这类指令时，CPU 会把它交给 Hypervisor，由 Hypervisor 代为完成或者模拟一个结果，然后再把控制权还给客户系统。这个过程叫"陷入—模拟—返回"。

后来硬件帮了忙：CPU 厂商在指令集层面提供了虚拟化支持，把"拦截特权指令"这件事做成硬件功能，损耗大幅下降。这也是为什么今天的虚拟机跑起来"几乎和物理机一样快"——不是软件变聪明了，是**硬件替软件干了活**。

这一页你要带走的核心判断是：**虚拟化的本质是"资源的时分复用 + 状态的隔离"**。时分复用让多个系统共享一份硬件，隔离让它们互不干扰。

> **板书：**三层结构，从下往上：物理硬件（CPU・内存・磁盘・网卡，只有一份）→ Hypervisor（切分・调度・拦截特权指令）→ 多个 Guest OS（各以为自己独占一台机器）。旁边写机制："陷入—模拟—返回"，以及一句总结："虚拟化＝时分复用＋状态隔离"。

提问：虚拟机里的操作系统，知道自己是虚拟机吗？它为什么能装自己的驱动？

预设回答：

- 答"不知道，它以为自己是真机"——基本正确：这正是虚拟化的目标，让客户系统"无感"。

- 答"知道，因为有虚拟化驱动"——也对，而且更专业：为了性能，客户系统通常装"半虚拟化驱动"，主动配合 Hypervisor，这时它是"知道"的。

- 答"虚拟机就是把真机的系统拷一份"——这是常见误解：拷的是**文件**，跑起来的是**被切分出来的硬件资源**，不是拷贝。

- 收口：两种回答都对，取决于用全虚拟化还是半虚拟化——下一页就讲这个区分。

> **易错点 / 考点：** ① 三层结构（硬件—Hypervisor—Guest OS）要能画出来；② 必背术语：Hypervisor、Guest OS、特权指令拦截；③ 常考判断："虚拟机会让物理机变慢所以不该用"——损耗存在但已被硬件虚拟化大幅降低，且换来的是利用率与灵活性的巨大提升。

> **过渡：** 原理的下一半，是虚拟化在实现上的两条技术路线。

【12.19】虚拟机工作原理 – 续（2 分钟）

> 这一页接着上一页，讲虚拟化实现上的两条路线：**全虚拟化**和**半虚拟化**。没有正文，我按概念补，你听逻辑就行。
>
> **全虚拟化**：客户系统不知道自己在虚拟机里，一条指令都不用改，Hypervisor 靠二进制翻译或硬件辅助把敏感指令换掉。**兼容性极好**，现成镜像拿来就跑；代价是有性能损耗。
>
> **半虚拟化**：客户系统是改过的，它知道自己跑在虚拟机上，会主动调用 Hypervisor 的接口。**性能好**，但只能跑改过的系统，现成的 Windows 镜像用不了。
>
> 今天常常混着用：整体是全虚拟化，I/O 上装半虚拟化驱动（网卡、磁盘），把最耗性能的那块优化掉——"virtio"就是这条路线的产物。
>
> 再补一个问题：虚拟机虚拟了哪几样资源？**内存**（各自连续地址空间，映射到物理内存不同区域）、**存储**（看到一块硬盘，实际是宿主机上的镜像文件）、**网络**（看到一块网卡，实际是虚拟交换机端口）。
>
> 存储这一项还带来一个对测试极有用的能力——**快照**：装好环境、配好数据，拍一张，每次测试前还原一次，这就是"可重复"最省事的实现。带走一句：**全虚拟化要兼容，半虚拟化要性能，快照是最该学会用的功能。**

> **板书：** 两条路线对照：全虚拟化（不改 Guest OS／二进制翻译＋硬件辅助／兼容好・有损耗）｜半虚拟化（改 Guest OS／主动调用接口／性能好・兼容差）→ 实际混用（如 virtio 驱动）。再列三项资源虚拟化：内存・存储（镜像文件＋快照）・网络（虚拟交换机）。底部："快照＝可重复的最省事实现"。

> **提问：** 你想在十分钟内把一套装好数据库和测试数据的 Linux 环境复制出三份给三个同事，最省事的做法是什么？

> **预设回答：**

- 答"从装好的那台拍快照/做模板，克隆三份"——最省事，也是工程上的标准做法。

- 答"三个人各装一遍，反正有安装脚本"——脚本化也可行，但装完还要灌数据、调配置，十分钟不够，而且三次之间可能不一致。

- 答"让他们连同一台机器，分不同账号"——省机器但写进"共享环境冲突"这个坑：数据互相污染，改配置互相影响。

- 收口：选型的判断标准永远是那两个词——**可重复、可隔离**。

> **易错点 / 考点：** ① 全虚拟化与半虚拟化的区别要能说清：**是否修改客户操作系统**；② 常考判断："半虚拟化性能更好所以应该优先选"——要看场景，兼容性有要求时全虚拟化更合适；③ 记住快照的用途，它常与"环境可重复""快速回滚"一起出现在案例题里。

> **过渡：** 原理清楚了，落到工具上——市面上到底有哪些虚拟机软件，怎么选。

【12.20】常见的虚拟机软件（3 分钟）

这一页课件列了九种虚拟机软件，我先给你归个类，不然九个头绪太乱。

第一类是 Linux 上的开源主力：**KVM**（Kernel-based Virtual Machine），它是 Linux 内核自带的虚拟化架构，把内核直接当 Hypervisor 用；**QEMU**，开源的全能模拟器，能模拟各种硬件；**Xen**，开源的老牌虚拟化方案，早期云计算大量使用。这三个是服务器侧的，特点是开源、可控、能上规模，但配置门槛比桌面软件高。

第二类是桌面侧的商业软件：**VMware Workstation/Fusion**，Workstation 是 Windows 和 Linux 上的，Fusion 是 macOS 上的，同一家的产品按平台分两个名字；**Parallels**（macOS），在 Mac 上跑 Windows 体验比较好；**Hyper-V Manager**，微软自家的，Windows 服务器版和专业版里就有。

第三类是轻量或特殊用途的：完全免费和开源的 **Oracle VM VirtualBox**，跨平台、上手快，教学和个人测试用得最多；**Virtuozzo**，SW-soft 公司的产品，走的是容器化路线，偏主机服务商场景；还有 **Colinux**（Windows 上的开源方案），它的思路比较特别，是让 Linux 作为一个进程跑在 Windows 里，不走完整虚拟机的路子。

怎么选？我给三步。**第一步看平台**：你的宿主系统是什么，Mac 上就 Fusion 或 Parallels，Windows 上就 Workstation、VirtualBox 或 Hyper-V。**第二步看用途**：桌面学习、单机测试，VirtualBox 和 Workstation 足够；要在服务器上批量跑、要接集群管理，就上 KVM/QEMU 这一系。**第三步看约束**：有没有许可证预算、

要不要和现有云平台对接、团队会不会维护。注意第三步常常是被忽略的那一步——**装得上不等于维护得住**。

再提醒一个易错点：Hyper-V 和 VirtualBox、VMware 在同一台 Windows 上**同时启用会打架**，因为它们都要占用硬件虚拟化能力。这不是软件缺陷，是资源独占。遇到"虚拟机启动报错、提示虚拟化不可用"，先查有没有别的虚拟化软件在跑。

板书：三类归位：服务器开源（KVM・QEMU・Xen）｜桌面商业（VMware Workstation/Fusion・Parallels・Hyper-V）｜轻量特殊（VirtualBox・Virtuozzo・Colinux）。选型三步：看平台 → 看用途 → 看约束（能不能维护）。备注："硬件虚拟化能力被占用会互相冲突"。

提问：老师让你在自己电脑上装个 Linux 做课程实验，你选哪个？理由是什么？

预设回答：

- 答"VirtualBox，免费、跨平台、够用"——教学场景的标准答案，成本为零、资料多。
- 答"VMware Workstation，性能好、功能全"——也可以，但要看有没有授权，别在机房批量装无授权版本。
- 答"直接装双系统，性能最好"——双系统确实快，但切换成本高、不能与宿主系统同时用，做测试效率反而低。
- 收口：**学生场景优先看成本和易用，生产场景优先看规模和可维护性**。

易错点 / 考点：① 九个名称要能对上归属（尤其 KVM 属 Linux 内核、Xen 是开源、Virtuozzo 属 SW-soft）；② 高频错答：把 QEMU 当成"另一种虚拟化架构"——它是**模拟器/虚拟化软件**，和 KVM 配合构成完整方案，下一页专门讲；③ 记住一条实战经验：多款虚拟化软件同时启用会争抢硬件虚拟化能力，报错先查冲突。

过渡：九种软件里，Linux 上最主流、也最值得讲清楚的是 QEMU 加 KVM 这一组——它们为什么必须一起用？

【12.21】12.3.2 QEMU-KVM 虚拟机解决方案（3 分钟）

这一页把 QEMU 和 KVM 的分工讲得很清楚，三句话，我逐句展开。

第一句：**QEMU 通过动态二进制转换几乎可以模拟任何 IO 设备，再将 Guest OS 的指令转译给宿主机硬件执行。**这句话两个关键词：**IO 设备**和**指令转译**。QEMU 能干的事很全：它可以把一块网卡、一个磁盘控制器、一个显卡"假装"出来，让客户系统以为自己插着这些硬件；它还能把客户系统的指令一条条翻译成宿主机能

执行的指令。所以 QEMU 单独就能跑虚拟机，兼容性极好——这也是它最原始的价值。

第二句：**KVM 是 Linux 内核提供的虚拟化架构，可将内核直接充当 Hypervisor 来使用。**注意"内核"两个字：KVM 不是一个独立软件，它是 Linux 内核里的一个模块，加载之后内核本身就变成了 Hypervisor。这句话解释了为什么 KVM 性能好——**它不需要额外一层，虚拟机就是内核直接管理的进程**。

第三句是关键：**但 KVM 的 kvm.ko 本身只提供了 CPU 和内存的虚拟化，缺少 IO 设备的虚拟化。所以 KVM 必须结合 QEMU 才能构成一个完整的虚拟化技术。**这就是答案：QEMU 补的正是 IO 这一块。

所以这一页真正要你记住的是一个**分工关系**：**KVM 管 CPU 和内存（快），QEMU 管设备和指令翻译（全）**。KVM 负责让虚拟机跑得快，QEMU 负责让虚拟机"看起来像一台完整的电脑"。谁缺了谁都不完整：只有 KVM，你没有设备模型，虚拟机起不来；只有 QEMU，性能上不去，因为所有指令都要软件翻译。

我给一个类比：**KVM 像发动机，QEMU 像车身和仪表盘**。发动机给你动力，车身和设备让你能真正上路。这套组合之所以成为 Linux 上的主流方案，就是因为它既有接近原生的性能，又有极强的设备兼容性——**这两件事本来是矛盾的，QEMU-KVM 用分工把它们各放在了合适的位置。**

> **板书：**两栏分工：QEMU（动态二进制转换／模拟各类 IO 设备／指令转译，兼容性全）｜KVM（Linux 内核模块 kvm.ko／内核即 Hypervisor／只管 CPU 与内存，性能好）。中间一个加号，下方"缺 IO 虚拟化 → 必须结合"。类比："KVM 是发动机，QEMU 是车身"。

提问：既然 KVM 性能好，为什么不干脆只用 KVM，把 QEMU 去掉？

预设回答：

- 答"因为 KVM 不管设备，没有设备虚拟机起不来"——正解：kvm.ko 只做 CPU 和内存虚拟化。
- 答"因为 QEMU 能模拟各种硬件，兼容性好"——这是 QEMU 的价值，但真正的原因是**缺了它不完整**。
- 答"其实可以只用 KVM"——错答，把内核模块当成了完整虚拟化方案，这是本节最典型的混淆。
- 收口：记一句判断：**要性能找 KVM，要设备找 QEMU，成套才叫方案。**

易错点 / 考点：① 分工必须能一句话说清：KVM 管 CPU 与内存、QEMU 管 IO 与指令翻译；② 高频判断："KVM 是独立软件"——错，它是 Linux 内核模块；③ 记住"必须结合"这四个字的原因链：KVM 缺 IO 虚拟化 → 需要设备模型 → QEMU 提供 → 组成完整方案。

过渡：虚拟机是"虚拟硬件"，接下来换一条更轻的路——不虚拟硬件，只隔离进程，这就是容器。

【12.22】12.3.3 容器技术与 Docker（3 分钟）

这一页讲容器，开头先做了一个很重要的对比：**容器和虚拟机类似，都可以看作是虚拟实体，满足隔离性和可管理性**。这句话先承认了两者的共同点——都提供隔离、都能被管理。但紧接着就是"但是"：**容器的核心技术是 Cgroups 与 namespace（命名空间），通过 namespace 实现资源隔离，通过 Cgroups 实现资源控制。**

这两根支柱一定要分清，考试常考。

namespace（命名空间）管"看不见"。它让一个进程拥有自己独立的"视野"：独立的进程列表（看不到外面的进程）、独立的网络（自己的 IP 和端口）、独立的文件系统挂载点、独立的主机名、独立的用户空间。所以在容器里 `ps` 只能看到自己那几个进程，感觉像一台独立机器。这就是**隔离**。

Cgroups（控制组）管"用多少"。它限制和统计一组进程能用多少资源：最多用几个核的 CPU、最多吃多少内存、磁盘读写限速多少、能开多少进程。超出限制就被约束，比如内存超了就被杀掉。这就是**资源控制**。

一个管"看什么"，一个管"用多少"——**namespace 划边界，Cgroups 定配额**。这两句是我给你记的。

那容器和虚拟机的本质区别在哪？一句话：**虚拟机虚拟的是硬件，容器虚拟的是操作系统环境**。虚拟机里每个客户系统都有自己完整的内核，跑在自己的虚拟硬件上；容器里的所有进程**共用宿主机的那一个内核**，只是被 namespace 隔开、被 Cgroups 限住。所以容器里不能装内核、不能改内核参数、不能跑和内核版本强绑定的东西——这也是它的边界。

边界带来的差异很直接：**共享内核 = 启动快、体积小、开销低；共享内核也意味着隔离强度不如虚拟机**，一个容器里的内核级漏洞可能影响整台宿主机。所以选型时问自己一句：**我要的是"隔离到内核"，还是"隔离到进程"？** 前者用虚拟机，后者用容器。

> **板书：**两根支柱：namespace（管"看不见"：进程・网络・挂载・主机名・用户）＝隔离｜Cgroups（管"用多少"：CPU・内存・IO・进程数）＝配额。下方一行对比："虚拟机虚拟硬件（各带内核）／容器虚拟 OS 环境（共享内核）"。角落写："要隔离到内核→虚拟机；隔离到进程→容器"。

提问：容器里能改内核参数吗？为什么？

预设回答：

- 答"不能，它和宿主机共用一个内核"——正解，这是容器与虚拟机的分水岭。

- 答"能，容器是独立环境"——典型错答，把"文件系统独立"当成了"内核独立"。

- 答"有些能，取决于权限和宿主设置"——也对，比如部分网络参数在内核允许的范围内可以调，但本质仍是共享内核。

- 收口：判断依据是"**内核是不是自己的**"，不是"看起来像不像一台机器"。

> **易错点 / 考点：** ① namespace 与 Cgroups 的分工必须能分清：**隔离 vs 资源控制**；② 高频错答："容器是轻量级虚拟机"——错在"虚拟机"，容器不是虚拟化硬件，是进程级隔离；③ 常考判断："容器比虚拟机更安全"——通常相反，共享内核意味着隔离强度较弱。

> **过渡：** 概念懂了，动手怎么做一个容器？下一页从构建流程讲起。

【12.23】基于 Docker 构建容器（3 分钟）

> 这一页标题是"基于 Docker 构建容器"，正文只给了两条参考链接：Docker 官方文档和腾讯云社区的一篇入门文章。课件把细节交给文档，我在这里把**流程与概念骨架**给你搭起来，这样你去看文档就不会迷路。
>
> 先立三个概念，它们是 Docker 的地基。**镜像（Image）**：一个只读的模板，里面装着文件系统、依赖库、环境变量、启动命令。它像"安装光盘"，本身不能运行。**容器（Container）**：镜像跑起来的实例，在镜像之上加了一层可写的薄层。它像"装好的系统正在运行"。**仓库（Registry）**：存放镜像的地方，公共的、私有的都算。三者关系一句话：**从仓库拉镜像，用镜像起容器**。
>
> 再说构建流程，四步。**第一步写 Dockerfile**——一个文本文件，一行一条指令：基于哪个基础镜像、拷贝哪些文件、装什么依赖、暴露哪个端口、用什么命令启动。**第二步 build**——把 Dockerfile 和上下文交给 Docker，它逐层执行，产出镜像。**第三步 run**——用这个镜像起容器，映射端口、挂载目录、传环境变量。**第四步 push**——把镜像推到仓库，别人就能拉。
>
> 这里有个特别值得讲的设计：**分层**。Dockerfile 的每条指令生成一层，层是可以复用的缓存。你把"装依赖"写在"拷代码"前面，改代码时依赖那一层不用重建，构建就快；写反了，改一行代码就要重装一遍依赖。所以**指令顺序本身就是性能设计**，这不是玄学，是缓存机制决定的。
>
> 对测试的实战意义在哪？**你的测试环境可以被写成几十行文本**。以前交接环境靠"说明书 + 口口相传"，现在是"把这个 Dockerfile 拿去 build"。环境不再是一台具体的机器，而是**一份可以被 review、被版本管理、被 diff 的代码**——这就是 12.4 基础设施即代码的雏形。

> **板书：** 三概念：镜像（只读模板，"光盘"）→ 容器（运行实例，加可写层）→ 仓库（存镜像）。流程四步：

> 写 Dockerfile → build → run → push。旁边写"分层缓存"：装依赖在前、拷代码在后 → 改代码不重装依赖。
> 底部："环境＝几十行可 review 的文本"。

提问：为什么 Dockerfile 里"装依赖"要写在"拷贝业务代码"之前？

预设回答：

- 答"利用分层缓存，改代码时不用重装依赖"——标准答案，也是实际优化里最常做的一件事。

- 答"习惯上这样写"——不算错，但说不出理由，遇到构建慢就不知道该改哪。

- 答"顺序无所谓，反正最后结果一样"——结果一样，**代价差很多**：每次改一行代码就重装全部依赖，构建时间可能从几十秒变成几分钟。

- 收口：写 Dockerfile 的第一条性能纪律——**变得慢的放前面，变得快的放后面**。

易错点 / 考点：① 镜像/容器/仓库三者的关系要能说清，常考"容器是镜像的运行实例"；② 高频混淆：把 Dockerfile 当成"脚本"——它是**声明式的构建描述**，逐层执行并缓存；③ 记住分层缓存对构建速度的影响，这是实操题的常见追问。

过渡：构建出来之后，容器到底比虚拟机好在哪？下一页把三条优点摆出来。

【12.24】容器的优点（3 分钟）

这一页给了容器的三条优点，每条都带对比，我按"优点 + 原因 + 对测试的意义"讲。

第一条，**镜像体积小，没有内核，属于轻量级应用**。为什么小？因为容器镜像里只有应用和它的依赖，**不带操作系统内核**——内核用的是宿主机的那一个。体积小的直接好处是**传输快、存得多、拉取快**：CI 机器上拉一个几十兆的测试镜像，几秒钟；拉一个几 G 的系统镜像，可能几分钟。

第二条，**创建和启动更快**：课件给了对比——启动一个虚拟机需要几分钟，Docker 容器的创建和启动只需要几秒钟。原因还是那个：虚拟机要模拟硬件、引导操作系统、启动一堆系统服务；容器只是启动一个被隔离的进程，没有引导过程。对测试的意义非常大：**测试环境可以"用完就扔"**。跑完一批用例，容器销毁；下一批用例，重新起一个干净的。这才叫真正的隔离——不是"清理过"，而是"从来没脏过"。

第三条，**更高的资源利用率**：层次更高，降低额外资源开销，资源控制粒度更小，部署密度更大，占用磁盘空间更小。这句里有四个"更"，我拆开：**层次更高**指抽象层级在操作系统之上，不用管硬件；**额外开销小**指

没有一整套 Guest OS 的 CPU 和内存开销；**控制粒度小**指 Cgroups 能按容器限 CPU 和内存，可以给每个容器精确配额；**部署密度大、磁盘占用小**指同一台机器上能跑几十个容器，而虚拟机可能只跑得下几个。

但优点背后有个代价必须说：**隔离弱、状态难持久、调试方式不同**。容器共享内核，隔离强度不如虚拟机；容器一销毁，里面的数据就没了，所以有状态的东西（数据库数据、测试产物）必须挂载到外面；调试时你得习惯"进容器里看"，而不是"远程登录一台机器"。

一句话总结这一页：**容器把"环境"变成了可以按秒创建、按秒销毁的东西**，这正好命中测试最核心的两个需求——**隔离和可重复**。

板书：三条优点＋原因：① 体积小（无内核，只带应用与依赖）→ 拉取快 ② 启动秒级（无引导过程，直接起进程）vs 虚拟机分钟级 ③ 利用率高（无 Guest OS 开销・Cgroups 精确配额・密度大・省磁盘）。右侧写代价："隔离弱・状态不持久（要挂载）・调试方式不同"。

提问：容器一销毁数据就没了，那测试环境的数据库怎么办？

预设回答：

- 答"挂载到宿主机目录或数据卷，容器删了数据还在"——正解，标准做法。

- 答"每次重新灌一遍测试数据，反正有造数脚本"——也可行，甚至更"干净"，适合无状态测试；但慢，且脚本本身要维护。

- 答"那就别删容器"——这就把容器的最大优势扔了：不删容器，环境又会漂移回"越用越乱"的老路。

- 收口：原则是**无状态的东西用容器，有状态的东西挂出去**。

易错点 / 考点：① 三条优点要能说全，且能给出原因（体积小因为无内核、启动快因为无引导）；② 高频判断："容器启动快是因为它比虚拟机优化得好"——本质原因是**没有操作系统引导过程**；③ 案例题常问"为什么用容器搭测试环境"，答隔离、可重复、秒级创建、密度高四点。

过渡：一台机器上跑几十个容器，谁来决定哪个容器跑在哪台机器上？这就轮到集群管理出场了。

【12.25】12.3.4 集群管理与 Kubernetes（3 分钟）

课件说：**Kubernetes 是目前最具影响力的容器集群管理工具之一，为容器化的应用提供部署运行、资源调度、服务发现和动态伸缩等一系列完整功能，提高了大规模容器集群管理的便捷性。**

第一个功能，**部署运行**：你告诉它"我要三个副本的这个应用"，它就负责把容器跑起来，并且**持续保证有三个**。某个容器挂了，它自动补一个；某个节点宕了，它把上面的容器迁走。这个特性在 K8s 里叫"期望状态"——你描述"要什么"，它负责"做到并保持"。这一点和第 12.4 节的基础设施即代码是一脉相承的：**声明式，不是命令式**。

第二个功能，**资源调度**：集群里有十台机器，新来的容器该放哪台？它要看每台机器还剩多少 CPU 和内存、有没有特殊要求（比如必须有 GPU、必须和某个服务在同一台），然后选一个合适的位置。

第三个功能，**服务发现**：容器是会不停创建销毁的，IP 一直在变，那前端怎么找到后端？K8s 给每个服务一个稳定的名字和虚拟 IP，请求打到这个名字上，它负责转发到当前活着的容器。**没有服务发现，微服务就没法用容器跑**——因为你不能把一堆会变的 IP 写进配置文件。

第四个功能，**动态伸缩**：负载高了自动加副本，低了自动减。这一条对性能测试特别有用：你可以观察系统在"自动扩容"这个机制下，扛住压力的表现如何。

给测试人员的落点是什么？**K8s 让"每次测试都从一套全新的环境开始"变成可能**。一套环境用命名空间隔开，测试跑完整个命名空间删掉，下次重新创建。这就是前面反复讲的"可重复"的最终形态——不是"把环境清理干净"，而是"环境根本不存在过第二次"。

但要提醒一句：**K8s 本身也是有成本的**——学习曲线陡，运维复杂度高。团队只有两三台机器、几个服务，用 Docker Compose 就够，上 K8s 是"用高射炮打蚊子"。

> **板书：**定义＋四功能：部署运行（维持期望状态・挂了自动补）｜资源调度（选节点：剩余资源・特殊要求）｜服务发现（稳定名字／虚拟 IP 转发到活容器）｜动态伸缩（按负载增减副本）。底部："声明式：描述要什么，它负责保持"。右侧一句："小规模别上 K8s"。

提问：容器 IP 一直变，前端服务怎么稳定地找到后端？

预设回答：

- 答"通过服务名，K8s 内部 DNS 解析"——正解，服务发现的核心机制。

- 答"把 IP 写进配置文件"——这就是容器化之后最常见的翻车点：容器重建、IP 变了，配置全废。

- 答"用固定的负载均衡地址"——思路对，K8s 里 Service 就是干这个的，本质是一种稳定的虚拟入口。

- 收口：容器化之后的第一条纪律——**不要依赖 IP，要依赖名字**。

易错点 / 考点：① 四个功能要能说全：部署运行、资源调度、服务发现、动态伸缩；② 高频错答：把 K8s 当

成"容器运行时"——它是**集群管理与编排**工具，真正跑容器的是容器运行时；③ 记住"声明式/期望状态"这个提法，它是 K8s 与脚本式部署的根本差别。

过渡：K8s 里调度的最小单位不是容器，而是 Pod——下一页专门讲它。

【12.26】集群管理与 Kubernetes（2 分钟）

这一页讲 Pod，一句话定义：**Pod 是 K8s 中的最小调度单元，一个 Pod 封装 n 个容器。**

第一层：**最小调度单元**。K8s 调度的时候，调度的对象是 Pod，不是容器。也就是说，一个 Pod 里的所有容器，**一定被放在同一台机器上**。这一点决定了 Pod 的用法——**你希望必须住在一起的东西，才放进同一个 Pod**。

第二层：**一个 Pod 封装 n 个容器**，n 可以是 1，也可以多个。多个容器的场景有个专门的名字叫"边车模式"：主容器跑业务，边车容器干辅助的活，比如收集日志、做代理、同步配置。

那同一个 Pod 里的容器**共享什么**？三样：**共享网络命名空间**——同一个 Pod 里的容器用的是同一个 IP 和端口空间，所以它们之间用 `localhost` 就能互相访问，但端口不能冲突；**共享存储卷**——挂在 Pod 上的数据卷，里面每个容器都能看到；**共享生命周期**——Pod 被删掉，里面的容器全都没了。

为什么要有 Pod 这一层，直接调度容器不行吗？因为**有些进程天生要绑在一起**：主程序和一个日志转发器，必须同机部署、必须共享文件；把它们当成两个独立容器分别调度，就可能被分到两台机器上，文件共享就断了。Pod 这一层解决的正是"**必须同机的一组进程**"这个问题。

对测试的理解落点：你要测一个微服务系统，就得知道**被测对象是 Pod 而不是单个容器**。日志分散在多个容器里、进程之间用 localhost 通信、Pod 重建后 IP 会变——这三件事都会直接影响你写自动化脚本的方式。

板书：Pod 定义："最小调度单元，封装 n 个容器"。两层含义：调度对象是 Pod（同 Pod 必同机）｜n 个容器（可 1 可多，边车模式）。共享三样：网络命名空间（同 IP・localhost 互通・端口不冲突）／存储卷／生命周期。底部："Pod 解决'必须同机的一组进程'"。

提问：同一个 Pod 里的两个容器，能不能都监听 8080 端口？

预设回答：

- 答"不能，它们共享网络命名空间，端口会冲突"——正解，这是最直接的推论。

- 答"能，它们是两个容器互不影响"——典型错答，把"文件系统隔离"当成了"网络隔离"。

- 答"改一个端口就行"——解决办法正确：既然是共享网络，就该按一台机器来规划端口。

- 收口：判断依据是**共享哪一层**——共享网络就要按同机规划端口。

> **易错点 / 考点：**① Pod 的定义要能准确复述："最小调度单元，封装 n 个容器"；② 高频错答：把 Pod 说成"一组必须同机部署的容器"就完了——要补上**共享网络、存储与生命周期**；③ 记住同 Pod 容器用 localhost 互通、端口不能重复，这是面试与考试的高频细节。

> **过渡：**从单个 Pod 到整个集群，再看一眼真实云上的一套基础设施长什么样。

【12.27】基础设施架构（3 分钟）

这一页给了一张基础设施架构图，并列出"**使用的 AWS 服务**"。先说清用途：这不是让你去背亚马逊的产品名，而是让你看清**一套完整的测试基础设施由哪几类云服务拼起来**。课件列的十四个缩写真不少，我按"层"给你归位，归完你就忘不掉了。

网络层：**VPC**（虚拟私有云，相当于你在云上的一块隔离网络）、**Route53**（域名解析）、**ALB**（应用负载均衡，把请求分发到后面多台机器）、**CloudFront**（内容分发，把静态资源推到离用户近的节点）。这四个决定"请求怎么进来、怎么找到目标"。

计算层：**EC2**（弹性计算云，就是虚拟机）、**ECS**（弹性容器服务，跑容器）、**ECR**（弹性容器仓库，存镜像）。这一层是"东西跑在哪"。

数据层：**RDS**（关系型数据库服务）、**S3**（简单存储服务，存文件与静态资源）。这一层是"数据放哪"。

安全层：**IAM**（身份与访问管理，谁能操作什么）、**ACM**（证书管理，管 HTTPS 证书）。这一层最容易被忽略，也最容易出事——权限配错，测试环境就能被外人访问。

运维与消息层：**CloudWatch**（监控与日志）、**SNS**（简单通知服务）、**SQS**（简单队列服务）。这一层负责"看得见"和"传得动"。

为什么要给你看这张图？因为它把前面四类要素落地了：硬件（EC2）、网络（VPC/ALB/Route53）、软件（ECS/ECR/RDS）、数据（S3/RDS）——**你看到的不是产品名，是"要素—服务"的对应关系**。

还有个现实提醒：**云上按用量计费，测试环境最容易烧钱**。忘关的机器、忘删的存储、长期闲置的数据库，

账单会自己长大。所以后面讲自动化时，一定会配套讲"用完即毁"和成本告警——这是同一件事的两面。

> **板书：**五层归位：网络（VPC・Route53・ALB・CloudFront）｜计算（EC2・ECS・ECR）｜数据（RDS・S3）｜安全（IAM・ACM）｜运维与消息（CloudWatch・SNS・SQS）。右侧：要素对应关系（硬件→EC2、网络→VPC、软件→ECS、数据→S3/RDS）。底部提醒："按量计费，忘关就是烧钱"。

提问：测试环境跑完一轮，最容易被忘记清理、也最花钱的通常是哪几类资源？

预设回答：

- 答"忘记关的虚拟机和数据库实例"——最常见，按小时计费，一直开着就是一直烧。
- 答"存储和镜像"——也很典型：快照、镜像、日志堆着不清，单看便宜，累积起来很可观。
- 答"反正是测试环境，不用管"——这是最贵的一句"反正"，账单出来才想起它。
- 收口：**测试环境的成本纪律：能用完即毁的就别长期开着，能自动清理的就别靠人记。**

易错点 / 考点：① 十四个缩写的**归类**比死背名字更重要，考试若给一组服务让你分层，按网络/计算/数据/安全/运维来分；② 高频错答：把 EC2 与 ECS 混为一谈——一个是虚拟机、一个是容器服务；③ 记住"按量计费"这个约束，它直接推导出本章后面的两条工程纪律：**环境用完即毁、成本要能被看见。**

过渡：云上的服务都是自己的，可测试时你要调的那些服务往往不是自己的——这就是服务虚拟化要解决的。

【12.28】12.3.5 服务虚拟化及其工具（3 分钟）

这一节标题是"服务虚拟化及其工具"，课件给了一句话：**当前比较流行的服务虚拟化工具包括开源的 WireMock、Hoverfly 等，以及 Parasoft Virtualize、Tricentis Tosca 等商业软件。**

先解决一个概念混乱：**服务虚拟化和虚拟机、容器不是一回事。**虚拟机虚拟的是"硬件"，容器虚拟的是"操作系统环境"，服务虚拟化虚拟的是"**别人的服务**"。它不管你跑在哪台机器上，它只管一件事：**当你的系统要去调用一个外部服务时，让那个调用得到你想要的结果。**

为什么需要它？我列四类真实困境。**第一类，依赖不可用**：你要调的那个第三方系统还在开发中，接口文档有、服务没有，你的联调没法做；或者它只在办公网能访问，测试环境连不过去。**第二类，依赖不稳定**：第三方服务时快时慢、时不时超时，你的自动化用例一半失败是它的锅，你还没法让它稳定。**第三类，依赖不可控**：你想测"对方返回 500 怎么办""对方 30 秒才响应怎么办""对方返回超长字符串怎么办"——真实服务不

会配合你演这些。**第四类，依赖有成本或风险**：按次收费的接口（短信、地图、支付），你不能拿它压测；涉及真实资金和真实用户的接口，你更不能随便打。

这四类困境，服务虚拟化一次全解：**它给你一个假的服务，行为由你定义**——想让它返回成功就成功，想让它超时就超时，想让它返回脏数据就返回脏数据，而且免费、稳定、随时可用。

工具怎么选？**开源的 WireMock、Hoverfly** 适合大多数团队：轻量、能跑在 CI 里、社区活跃。**商业的 Parasoft Virtualize、Tricentis Tosca** 强在图形化配置、协议覆盖广、和企业既有测试平台集成好，代价是许可证成本。选型判断就一句：**你的依赖协议复不复杂、要不要和现有平台打通**。协议就是 HTTP，开源足够；要覆盖大型机、专有协议、复杂报文，商业工具有优势。

板书：定义一句话："虚拟的不是机器，是别人的服务"。四类困境：不可用・不稳定・不可控（造异常）・有成本/有风险。工具两栏：开源（WireMock・Hoverfly）／商业（Parasoft Virtualize・Tricentis Tosca）。选型判断："协议复杂度 × 平台集成需求"。

提问：支付接口不能拿来压测，但领导要求验证"下单高峰时系统扛不扛得住"，怎么办？

预设回答：

- 答"用服务虚拟化把支付接口假掉，只压自己这边"——正解，这也正是 Mock/服务虚拟化的典型用法。
- 答"少压一点，别把人家打挂"——能减小风险，但压测要的是量级，量小了结论也不成立。
- 答"直接压，出事再说"——这是最危险的做法：真实扣款、真实短信，事故成本由你承担。
- 收口：判断标准是"**这个依赖我能不能控制**"，不能控制的就虚拟掉。

易错点 / 考点：① 必背四个工具名及归属：WireMock、Hoverfly 开源；Parasoft Virtualize、Tricentis Tosca 商业；② 高频混淆：把服务虚拟化与虚拟化技术混为一谈——前者虚拟**服务**，后者虚拟**机器**；③ 案例题常用问法："外部依赖不可用导致测试无法进行，如何解决"，标准答法：服务虚拟化/Mock，并说明可模拟延迟、故障、限流。

过渡：工具名单里 Hoverfly 课件单独给了一页，我们看它能干哪五件事。

【12.29】开源服务虚拟化工具 Hoverfly（3 分钟）

这一页列了 Hoverfly 的五条能力，每一条都对着测试中的一个具体痛点，我一条条对上。

第一条，**可以在 CI 环境中替代缓慢和不稳定的外部服务或第三方服务**。注意"CI 环境"这四个字：CI 里最怕的就是不确定性，一个外部服务抖一下，整条流水线红一次，团队就开始不信流水线了。把外部依赖换成 Hoverfly，流水线就只反映**自己代码**的问题。

第二条，**可以模拟网络延迟、随机故障或速率限制以测试边缘情况**。这一条最能体现服务虚拟化的价值：你想验证"支付接口 3 秒没响应，我们这边会不会无限等待"，真实服务不会配合你；Hoverfly 可以让它每次都慢 3 秒。**异常路径的测试，靠真服务是测不出来的，只能靠假服务。**

第三条，**可以导入/导出、共享和编辑模拟数据**。能导出意味着它能进版本库、能 review、能共享给同事；能编辑意味着产品改了接口，你改配置就行，不用改代码。

第四条，**提供方便易用的命令行界面 hoverctl**。命令行意味着**能进脚本、能进流水线**——这一点很关键，图形界面工具在 CI 里是用不起来的。

第五条，**提供多种运行模式，可以对 HTTP 响应进行记录、回放、修改或合成**。这四种模式我拆开讲，考试爱考：**记录（capture）**——把真实服务的响应抓下来存成模拟数据，这是最省力的起步方式；**回放（simulate）**——按存下来的数据返回，用于稳定的回归测试；**修改（modify）**——在真实响应基础上做改写，比如注入一个错误码、改一个字段；**合成（synthesize）**——完全由你凭空造响应，适合服务还不存在的时候。

这四种模式其实对应测试的三个阶段：**服务已经在跑**用记录、**服务稳定**用回放、**服务要出异常**用修改、**服务还没有**用合成。

最后提醒一个工程细节：**模拟数据要跟着接口一起版本管理**。接口改了、模拟数据没改，测试就会假通过——这是服务虚拟化最隐蔽的坑。

板书：五条能力：① 替代缓慢/不稳定外部服务（保 CI 稳定）② 模拟延迟・随机故障・限流（测异常）③ 导入导出・共享・编辑（进版本库）④ hoverctl 命令行（能进流水线）⑤ 四模式：记录（capture）・回放（simulate）・修改（modify）・合成（synthesize）。底部："模拟数据要与接口同版本管理，否则假通过"。

提问：服务还没开发出来，你又要先写测试用例，用 Hoverfly 的哪种模式?

预设回答：

- 答"合成，因为真实响应根本不存在"——正解，合成就是凭空造响应。

- 答"记录，先抓一份真实的"——抓不到，服务不存在，这正是要区分模式的原因。

- 答"先等着，等服务好了再写用例"——这就把"测试左移"的机会浪费了，接口契约一定，用例就该动起来。

- 收口：四个模式对应四种现实条件，**条件不同，模式不同**。

易错点 / 考点： ① 四种模式的中英文对应要记牢：记录 capture／回放 simulate／修改 modify／合成 synthesize；② 高频错答：把"回放"和"记录"当成一回事——录是抓真实、放是按录制返回，方向相反；③ 简答常问"Hoverfly 在 CI 中有什么用"，答：稳定依赖、可造异常、可版本化、可命令行集成四点。

过渡： 造环境的工具都齐了，剩下的问题是怎么让它们"自动"跑起来——最后一节。

【12.30】12.4 测试基础设施的自动部署（1 分钟）

这一节是本章的落地部分，五个小节，我把顺序和逻辑先讲清。

12.4.1 基础设施即代码——观念先行：把机器、网络、配置都写成文件，进版本库。这一节会讲 IaC 的定义、工具的三个特征、以及四类工具的分工。

12.4.2 基础架构的自动部署——落到具体工具：以 Terraform 为例讲怎么把网络和服务器"声明"出来；再用 TestInfra 讲怎么**验证**基础设施真的建对了。

12.4.3 应用程序容器化及集群部署——把应用打进容器、推到集群，用 Docker 命令和 Kubernetes 落地。

12.4.4 应用程序的自动配置——机器起来了、容器跑了，里面还要装东西、改配置，Ansible 就是干这个的。

12.4.5 CI/CD 流水线——把前面所有环节串成一条自动流水线，用 Jenkins 做调度。

这一节的五个小节，其实是一条**建造顺序**：先把资源用代码描述（IaC）→ 再把资源自动建出来并验证（自动部署 + TestInfra）→ 再把应用装进去（容器与集群）→ 再把配置刷进去（Ansible）→ 最后让整件事自动触发（流水线）。

一句话记住这一节的目标：**从"人配环境"变成"机器建环境"，而且每次建出来的一模一样。**

板书： 五步建造顺序：IaC（写成代码）→ 自动部署＋TestInfra（建出来并验证）→ 容器化与集群（装应用）→ Ansible（刷配置）→ CI/CD（自动触发）。右侧："人配环境 → 机器建环境，且每次一样"。

提问： 五个小节里，哪一步最容易被跳过，但跳过之后问题最多？

> **预设回答：**

- 答"验证那一步（TestInfra）"——很深：建完了不验证，你怎么知道建对了？"建成功"和"建正确"是两件事。

- 答"自动配置（Ansible）"——也可能：手工改配置是环境漂移的主要来源。

- 答"都能跳，先把流水线跑起来"——短期能出效果，但跳过 IaC，流水线建的是一堆手工机器，等于自动化了个壳。

- 收口：**每一环都在防"不一致"，跳过哪一环，不一致就从哪里回来**。

> **易错点 / 考点：**① 五个小节名称与顺序要能默写；② 特别记住 12.4.2 里除了 Terraform 还有 **TestInfra**，很多同学只记前者；③ 简答可这样组织："IaC 是观念，Terraform 建资源，TestInfra 验资源，Docker/K8s 装应用，Ansible 配应用，Jenkins 串流程"。

> **过渡：**从观念开始——什么叫"把基础设施当代码写"。

【12.31】12.4.1 基础设施即代码（3 分钟）

> 这一页给了一个英文全称和一句定义：**Infrastructure as Code，IaC**——"基础设施即代码"是指**通过机器可定义文件管理和配置计算机数据中心的过程。将基础设施以配置文件的方式纳入版本管理，实现更灵活和更快捷的操作。**
>
> 这句话有三个要点，我们一个一个抠。
>
> 第一个要点，**机器可定义文件**。注意"机器可定义"这四个字：它强调的不是"人能看懂"，而是"机器能执行"。你写一份配置文件，机器照着它能把机器建出来，这才叫 IaC。
>
> 第二个要点，**管理和配置数据中心**。范围要划清：它管的不只是服务器，还包括网络（网段、路由、防火墙规则）、存储、负载均衡、甚至监控告警规则。所以 IaC 描述的是**一整套环境**，不是一台机器。
>
> 第三个要点，**纳入版本管理**。这是 IaC 最容易被忽略、也最关键的一半。配置文件进了 Git，就有了这些能力：**能看到谁在什么时候改了什么**（审计）、**能回滚到上一个正确版本**（恢复）、**能对比测试环境和生产环境差在哪**（diff）、**能走代码评审**（把关）。这四件事，手工配环境一件都做不到。
>
> **传统做法**：环境是"配置出来的"，靠人手点、靠经验、靠口口相传的文档；结果是每台机器都略有不同，出问题只能上机器上看，机器坏了重建一次又是一种新状态。**IaC 做法**：环境是"编译出来的"，一堆文本执行一

遍就是一套环境；文本一样，环境就一样，环境坏了删掉重建，重建出来的和原来一模一样。

有一个词特别能概括这个转变——**"环境是牲畜还是宠物"**。当宠物养，得给它起名字、生病了要治、死了很心疼；当牲畜养，编号管理，坏了就换。IaC 就是把环境从宠物变成牲畜的工程手段。

最后一句提醒：**IaC 的难点不是写配置，是"把配置写对且写全"**。漏写一条防火墙规则，环境建出来就是不通的；而手工配置时你随手就点了。所以 IaC 需要配套验证——这就是后面 TestInfra 出场的原因。

板书：定义三要点：机器可定义（机器能执行，不是给人看的文档）｜管整套数据中心（服务器・网络・存储・负载均衡・告警规则）｜纳入版本管理（审计・回滚・diff・评审）。下方对比："手工配置＝环境是宠物；IaC＝环境是牲畜"。

提问：把安装步骤详细写进一份 Word 文档发给大家，算不算基础设施即代码？

预设回答：

- 答"不算，机器读不懂，还得靠人照着做"——正解，"机器可定义"是硬条件。

- 答"算，只要写得足够详细"——详细也只是"文档化"，人执行就会有偏差，第 100 次和第 1 次不一样。

- 答"看情况，如果是脚本就算"——方向对了：**能被机器执行**才算，脚本或配置文件都行，关键是去掉人的手工步骤。

- 收口：判断标准一句话——**能不能让机器照着它把环境建出来，且每次结果一样**。

易错点 / 考点：① IaC 英文全称与中文译名都要会写；② 高频错答：把 IaC 等同于"写部署脚本"——脚本可以是命令式的，IaC 更强调**配置文件＋版本管理＋声明式**；③ 记住版本管理带来的四项能力：审计、回滚、diff、评审，案例题常用。

过渡：说了半天"配置文件"，它到底长什么样？下一页给一个示例。

【12.32】示例（2 分钟）

这一页是 IaC 的配置示例，没有正文，我们看图说话。语法不用记要看懂**这种文件的结构和思路**——它和你们平时写的程序，写法完全不一样。

先看它由哪几块组成。**第一块，声明"用哪个平台"**——要操作哪个云、哪个环境、用哪套凭据。**第二块，声明"要创建什么资源"**——一块资源就是一个对象：一个网络、一个子网、一台服务器、一条防火墙规则、一个

负载均衡，各有参数。**第三块，变量与参数**——把会变的东西提出来：区域、机器规格、副本数量，这样同一份配置能在测试环境和生产环境复用，只是传参不同。**第四块，输出**——建完告诉我关键信息：新机器的IP、负载均衡地址。

这里有个关键区别：**声明式，不是命令式**。命令式是"第一步建网络、第二步建子网、第三步起机器"，顺序和"已经存在怎么办"都得你自己管。声明式是"**我要的最终状态是：一个网络、里面两个子网、挂着三台机器**"——工具自己算：现在有什么、还差什么。所以同一份配置，空环境执行是"从零创建"，已有环境再执行是"只改差异"。

这个特性叫**幂等**——执行一次和十次，结果一样。对测试环境的意义极大：**你可以放心反复执行，不用担心把环境搞坏**。手工操作做不到这一点，你点第二遍就可能建出两套。

那这段配置该放哪？进 Git。谁改的、为什么改，全有记录。**配置进了版本库，"环境为什么是这样"才有答案。**

> **板书：**配置四块：平台声明（云/环境/凭据）→ 资源块（网络・子网・机器・防火墙・负载均衡，各有参数）→ 变量与参数（区域・规格・副本数，环境间复用）→ 输出（IP・地址，供后续使用）。中间大字："声明式 ≠ 命令式；特性＝幂等"。

提问：同一份 IaC 配置在已经建好的环境上再执行一次，会发生什么？

预设回答：

- 答"什么也不做，或者只改差异"——正解，这就是幂等的表现。
- 答"会再建一套，变成两套"——这是命令式脚本的行为，正是 IaC 要解决的问题。
- 答"会报错，说资源已存在"——早期工具可能这样，但现代 IaC 工具会对比状态后决定动作，不会简单报错。
- 收口：判断是否真 IaC，看它**能不能重复执行而结果不变**。

易错点 / 考点：① 声明式与命令式的区别要能举例说清；② 必背术语**幂等**，它是 IaC 反复执行安全的前提；③ 高频错答：把 IaC 配置当成"启动脚本"——脚本按顺序执行，配置描述最终状态。

过渡：看完样子，再看工具该具备哪些特征才算合格的 IaC 工具。

【12.33】IaC 类工具特征（3 分钟）

这一页讲 IaC 类工具应该具备的三个特征，课件写得比较凝练，我逐条加厚。

第一个特征，**版本控制**。课件说：这是将基础结构、配置、容器和管道持久化作为代码最重要的部分，该工具的配置文件应具有**可读性和版本控制性的语法**，所以**不能采用大型虚拟机镜像那样的二进制文件**。这句话点出了一个常见误区：有人以为"我把整台机器的镜像存起来，也叫基础设施即代码"。不行——镜像是个二进制大文件，你没法看它改了什么、没法 diff、没法在代码评审里讨论"这一行该不该删"。**配置文件必须是文本，文本才能被人审阅。**

第二个特征，**模块化**。课件说：最好的代码是可复用的，所以支持"模块化"的工具也要具备复用性，并且**能实现模块的参数化**。举个例子：你们团队要建"一套带数据库的测试环境"，这个组合会被反复用到十次。模块化就是把它打包成一个模块，调用时传参数——环境名、机器规格、数据库版本。不模块化会怎样？十份配置复制粘贴，改一个参数要改十处，必然改漏。**参数化是模块化的灵魂**：不能传参的模块，只是复制粘贴的另一种形式。

第三个特征，**可实例化或可部署**。课件说：该工具必须能够将代码（它可以是管道、配置的实例、容器或对云基础架构的更改）输入并部署到环境中，这样的工具必须是**纯脚本的，而且是全自动的，不需要手工操作或配置**。任何一步要人去点，这一环就不可能被自动化串起来，也就没法进流水线。

三个特征连起来是一条判断链：**是不是文本（能版本控制）→ 能不能复用（模块化＋参数化）→ 能不能全自动执行（无手工步骤）**。

补一句课件没写但很实用的：还要看**状态管理**能力——工具得知道"现在环境里已经有什么"，否则它无法计算差异。

> **板书：**三特征＋判断链：① 版本控制（文本配置・可读・可 diff；**不能是二进制镜像**）② 模块化（可复用＋**参数化**）③ 可实例化/可部署（纯脚本・全自动・**无手工步骤**）。底部判断三问："是文本吗？能复用吗？能全自动跑吗？" 备注：还要有状态管理。

提问：为什么说"把虚拟机镜像存起来"不算基础设施即代码？

预设回答：

- 答"镜像是二进制，看不出改了什么，没法 diff 和评审"——正解，直击第一条特征。
- 答"镜像太大，版本库放不下"——是现实问题，但不是本质原因；本质是**不可读、不可审**。
- 答"镜像也是代码的一种，存起来就行"——典型错答：形式上是文件，工程上丧失了一切可维护性。
- 收口：**代码之所以是代码，是因为它能被人阅读和修改，不只是能被机器执行。**

易错点 / 考点： ① 三个特征的名称要能默写：版本控制、模块化、可实例化/可部署；② 高频考点："为什么不能用二进制镜像"——不可读、不可 diff、不可评审；③ 记住"参数化"是模块化的关键，简答常追问"模块化怎么落地"。

过渡： 特征有了，具体工具有四类，课件把它们分了组——这是本节最该背下来的一张分类表。

【12.34】IaC 四类工具（3 分钟）

这一页是本节最该背下来的一页，IaC 工具分四类，每类都有代表工具，我给你配上"管什么"和"什么时候用"。

第一类，"基础架构即代码"工具，适用于基础架构的自动部署，如 **Terraform、CloudFormation** 等。它管的是"**机器、网络、存储这些资源怎么建出来**"——创建虚拟机、划网段、配负载均衡、开防火墙。Terraform 是跨云的代表，CloudFormation 是 AWS 自家的。**你从零开始搭一套环境，第一步用的就是这一类。**

第二类，"容器即代码"工具，用于应用程序容器化，课件说"只有 **Docker、Kubernetes** 等"。注意"只有"两个字，意思是这一类工具不多但地位稳固。Docker 管"把应用打成镜像"，Kubernetes 管"把镜像跑成集群"。**它管的是应用怎么装进去、怎么跑起来。**

第三类，"配置即代码"工具，用于应用程序的配置管理，有 **Chef、Puppet、Ansible 和 SaltStack**。这四个经常一起考。它管的是"**机器已经有了，里面的软件怎么装、配置怎么刷、服务怎么起**"。区别在于工作方式：Ansible 走 SSH，不用在目标机上装客户端；Chef、Puppet 通常要在目标机上装 agent，走"拉取配置"的模式。

第四类，"管道即代码"工具，用于持续集成和持续交付环境的自动化部署，如 **Jenkins 2.0、Drone.io、ConcourseCI**。它管的是"**这些步骤按什么顺序、什么条件自动触发**"。Jenkins 2.0 的意义是把流水线本身写成了代码（Pipeline as Code），而不是在界面上点来点去。

四类怎么记？我给你一条口诀：**建资源、装容器、配应用、串流程**。对应：Terraform 类建资源 → Docker/K8s 装容器 → Ansible 类配应用 → Jenkins 类串流程。

最后提醒一个实战顺序：这四类不是"选一个"，而是"**按顺序叠起来用**"。先 Terraform 建出机器和网络，再 Docker 打镜像、K8s 跑起来，再 Ansible 刷配置，最后 Jenkins 把它们串成一条流水线。这也是 12.4 后面四小节的顺序。

板书： 四类表格：基础架构即代码（Terraform・CloudFormation）｜容器即代码（Docker・

> Kubernetes）｜配置即代码（Chef・Puppet・Ansible・SaltStack）｜管道即代码（Jenkins 2.0・Drone.io・ConcourseCI）。底部口诀："建资源・装容器・配应用・串流程"。

> **提问：**Ansible 和 Terraform 都能"自动化"，它们的区别在哪？

> **预设回答：**

- 答"Terraform 建机器，Ansible 配机器里的东西"——正解，一句话分清。

- 答"都是自动化工具，差不多"——这是最容易混的错答，混了之后设计流水线就会把两件事塞进一个工具。

- 答"Ansible 也能建云主机"——技术上部分可行（有对应模块），但职责上应该分开：**资源生命周期归 IaC，配置归配置管理**。

- 收口：分工原则——**谁负责"有没有"，谁负责"里面是什么"**。

> **易错点 / 考点：**① 四类工具的名称、代表产品、对应职责三者要对上，这是本章最高频的考点之一；② 高频错答：把 Ansible 归到"基础架构即代码"——课件明确把它放在"配置即代码"；③ 记住四类**叠加使用**的顺序，案例题常问"设计一条自动化环境搭建链路"。

> **过渡：**四类里第一类最基础，课件用 Terraform 做了详细拆解，我们看它怎么组织一套云上环境。

【12.35】12.4.2 基础架构的自动部署（3 分钟）

> 这一页以 Terraform 为例，讲基础架构怎么自动部署。
>
> **先说能力**。课件说：Terraform 具有完成完整的云基础架构创建的能力，并通过 **DSL** 以编程方式将各个组件链接在一起，并能将云基础设施的有用部分定义为**带有参数化输入的模块**，而且可以与其他模块集成，具有良好的复用性。这里三个关键词：**DSL**（领域特定语言，就是它那套配置文件语法，专门用来描述基础设施）、**链接组件**（网络建好，机器挂到网络上，负载均衡挂到机器上——组件之间有依赖关系）、**参数化模块**（前面讲的模块化落地）。
>
> **再说结构**。课件说 Terraform 包括两个主要模块。**一个是管理模块**，定义 **VPC**（虚拟私有云）、**子网**、**NAT 网关**、**路由**、**安全组**和 **PuppetMaster** 等——你看这一串，全是"网络与基础管理"的东西：网段怎么划、子网怎么分、怎么访问外网、流量怎么走、谁能进来，还有一个配置管理服务器。**另一个是服务器模块**，在其子网中定义多个消息代理和多个自定义服务器的层，并将它们**动态链接到公共 ELB**（负载均衡）。

这个两层结构特别值得学，它体现了一个通用设计原则：**先网络后计算**。为什么？因为服务器必须挂在子网上，子网必须在 VPC 里，路由和安全组决定了服务器能不能被访问。**这也是 IaC 的现实难点——依赖顺序**。好在声明式工具会自动推导依赖：你写了"这台机器在某个子网里"，它就知道要先建子网。

最后说范围。课件点明：这里涉及的是**最基本的计算资源、存储资源和网络资源，包括网络和服务器的定义、配置和管理**。注意"最基本"三个字——它告诉你 IaC 的边界：它管资源，不管应用里面的业务逻辑，也不管机器里面的软件安装（那是 Ansible 那一类的事）。

对测试的直接价值：**一套完整的测试环境，从此可以"一键建出来、一键拆掉"**。要复现一个线上问题的环境？改两个参数，重建一套。

> **板书：**三层：能力（DSL 描述・组件链接・参数化模块可复用）｜结构（管理模块：VPC・子网・NAT 网关・路由・安全组・PuppetMaster ／ 服务器模块：消息代理＋多台自定义服务器 → 动态链接公共 ELB）｜范围（最基本的计算・存储・网络资源的定义、配置与管理）。底部原则："先网络后计算；声明式自动推导依赖"。

提问：为什么 IaC 里网络资源通常要写在服务器资源之前？

预设回答：

- 答"服务器必须挂在子网上，子网不通机器就白建"——正解，本质是依赖关系。

- 答"习惯写法，顺序其实无所谓"——对声明式工具来说执行顺序由依赖推导，但**逻辑上**必须先有网络，否则写不出服务器的归属。

- 答"因为 Terraform 要求这样"——不是语法要求，是**资源之间的真实依赖**决定的。

- 收口：记住设计原则——**先把路修好，再跑车**。

易错点 / 考点：① Terraform 两个模块的内容要能说出：管理模块（网络与基础管理）／服务器模块（消息代理＋服务器层＋挂 ELB）；② 高频错答：认为 IaC 会安装应用软件——课件明确其范围是"最基本的计算、存储和网络资源"；③ 记住 DSL 这个词，它常与"声明式"一起出现在判断题里。

过渡：资源建出来了，可你怎么知道它建对了？这就需要一个专门"测试基础设施"的工具。

【12.36】TestInfra（3 分钟）

这一页讲 TestInfra，它是本章一个很容易被略过、但观念上特别重要的工具。课件说：**Testinfra 是一个功能强大的库，可用于编写测试以验证基础设施的状态**，可以使用 **Molecule** 开发 Ansible 角色过程中添加测试关键组件，也可以和虚拟机管理工具 **Vagrant**、CI 工具 **Jenkins** 等集成起来，轻松完成 DevOps 模式下的全自动化流水线式的验证。

先讲观念。前面我们一直在讲"怎么把环境建出来"，但建出来不等于建对了。你 IaC 执行成功了，返回码是 0，可防火墙规则少了一条、某个服务没启动、目录权限不对——**这些都不会让构建失败，但会让测试全部失败**。于是团队开始怀疑代码，查两天，最后发现是环境问题。

TestInfra 要解决的就是这件事：**把"环境应该是什么样"写成断言**。比如断言：这台机器上 80 端口在监听、这个服务在运行、这个目录属主是某个用户、这个配置文件里包含某一行、这个包已安装。跑一遍，全绿说明环境符合预期；哪条红了，直接告诉你是哪一项不符。

这就把"环境验证"从一个靠人肉检查的动作，变成了**可重复执行的测试**。注意它的定位：它不是测业务功能的，它测的是**环境本身**——所以它属于"测试的测试"，或者叫基础设施的自动化验收。

三个集成点也要记住：**Molecule** 是开发 Ansible 角色时的测试框架，可以在角色开发过程中就插入 TestInfra 断言，保证角色写完就是对的；**Vagrant** 管虚拟机，TestInfra 直接连上去验；**Jenkins** 把它塞进流水线，每次环境创建后自动跑一遍验证。

一句话总结：**IaC 负责"建设"，TestInfra 负责"验收"，两个合起来才叫"可信的环境"**。只建不验，等于装修完不验收就交房——住进去才发现水电不通。

> **板书：**定位："编写测试以验证基础设施的状态"。断言示例：端口在监听・服务在运行・目录属主正确・配置文件含某行・包已安装。三个集成点：Molecule（Ansible 角色开发中加测试）・Vagrant（虚拟机）・Jenkins（流水线）。底部一句："IaC 建设，TestInfra 验收"。

提问：IaC 执行成功、返回码 0，是不是就说明环境建对了？

预设回答：

- 答"不一定，成功只说明命令执行完了，不代表状态符合预期"——正解，这正是 TestInfra 存在的理由。

- 答"是，没报错就是对的"——最典型的错答，也是"配置少一条规则"这类事故的根源。

- 答"可以人工上去看一眼"——能发现一部分，但人看不可重复、不可进流水线，规模一大就失效。

- 收口：**"执行成功"与"状态正确"是两件事，中间隔着一层验证。**

> **易错点 / 考点：**① TestInfra 的定位要能一句话说清：**验证基础设施状态的测试库**；② 三个集成对象要记住：Molecule、Vagrant、Jenkins；③ 高频判断："TestInfra 用来测业务功能"——错，它测的是环境状态；④ 案例题常问"如何保证环境搭建的正确性"，标准答法就是"IaC ＋ 基础设施测试"。

> **过渡：**TestInfra 的写法有个特点——参数化，并且能和单元测试框架结合，下一页看图。

【12.37】TestInfra 支持脚本参数化、与 UT 框架集成（2 分钟）

> 这一页是示意图，标题本身就是结论：**TestInfra 支持脚本参数化，并且能和单元测试框架集成**。这这两个特性决定了基础设施测试能不能规模化管理。
>
> **先说参数化**。基础设施测试有个天然麻烦：同一个断言要在很多环境上跑。"80 端口在监听"这条，测试环境要验、预发要验、生产也要验；每条断言都写死 IP，你就得写几十份几乎一样的脚本，改一次端口要改几十处。
>
> 参数化就是把这个变量提出来：**同一份脚本，传不同参数，跑不同环境**。这在工程上叫**测试矩阵**，它让"环境一致性"从口号变成可执行的动作——同一套断言验所有环境，哪个不符，立刻暴露。
>
> **再说与 UT 框架集成**。UT 就是单元测试框架，比如 Python 生态里最常用的 pytest。TestInfra 作为 Python 库，天然能用这些能力：**断言的写法、用例的组织、参数化机制、失败报告、插件生态**。你不用为"测环境"另学一套工具。
>
> 两个特性合起来的意义是：**基础设施测试可以和业务测试跑在同一套流程里**。CI 里先跑环境验证（确认环境对），再跑业务测试（确认功能对）；同样的框架、同样的报告、同样的通知——**这才叫"全自动化流水线式的验证"**。
>
> 最后给个实践建议：环境验证用例不用写多，**抓关键的十条二十条**——关键端口、服务、配置项、权限、依赖连通性。这二十条能挡住大部分"环境不对"的问题；写多了维护不动，反而没人跑。

> **板书：**两特性：① 参数化（同一脚本 × 多环境／多主机＝测试矩阵 → 把"环境一致性"变成可执行动作）② 与 UT 框架集成（断言・用例组织・参数化机制・报告・插件；如 pytest）。合起来："环境验证与业务测试同一套流程"。建议："抓关键 10–20 条断言"。

提问：为什么说"环境验证用例不用写太多，抓关键的就行"？

预设回答：

- 答"写多了维护不动，没人跑就等于没有"——正解，测试资产的第一杀手是维护成本。

- 答"关键项能挡住大部分环境问题"——也对：端口、服务、配置、权限、连通性这五项覆盖了多数故障。

- 答"越多越保险，应该全测"——听着稳妥，实际会让验证本身变成负担，最后被关掉。

- 收口：**能长期跑起来的少量断言，胜过写了不跑的一大堆。**

易错点 / 考点：① 两个特性的名称要能说出：脚本参数化、与 UT 框架集成；② 高频错答：以为 TestInfra 是独立的一套测试框架——它是**库**，依托现有单元测试框架；③ 记住"测试矩阵"这个词，它常与"环境一致性验证"一起考。

过渡：说了这么多工具，课件最后给了一份完整清单，我们按类别过一遍，帮你建立全景。

【12.38】测试基础设施工具清单（3 分钟）

这一页是工具清单，八类，我按"每一类解决什么问题"过一遍。你不用背全，但要记住**类别**，因为选型时你面对的是"我需要哪一类"，不是"我要哪个产品"。

第一类，基础架构类，如 CloudFormation、OpenStack 等。解决"把机器、网络、存储建出来"，对应前面讲的 IaC 第一类工具。**第二类，容器类工具**，如 Docker、Rocket、ElasticBox 等，解决"把应用打成可移植的包"。**第三类，资源编排管理工具**，如 Kubernetes、Apache Mesos、Swarm 等，解决"一群机器怎么当成一台来调度"。

第四类，微服务平台，如 OpenShift、Cloud Foundry、Mesosphere 等。它比编排再高一层：不光管容器怎么跑，还提供应用部署、服务治理、路由、构建打包这一整套开发者体验。可以理解为"带开发平台的集群"。

第五类，日志管理，如 Elastic Stack（ElasticSearch、Logstash、Kibana 和 Beats）、Logentries、Splunk 等。解决"出问题时去哪看日志"。**第六类，系统监控、告警与分析**，如 Prometheus、Icinga 2、Nagios Core、Zabbix、Cacti、Zookeeper 等，解决"系统现在健不健康、坏了通知谁"。**第七类，性能监控**，如 AppDynamics、Datadog、DynaTrace、New Relic、CollectD、StatsD 等，解决"慢在哪一段、哪个调用耗时"。

第八类，知识管理和沟通协作类工具，如 MediaWiki、Confluence、Zoom 等。这一类看着最不像"基础设施"，但课件把它列进来了——因为环境的用法、踩过的坑、操作规程，**必须有人写下来、有人能查到**。没有这一环，前面七类工具的知识全在个别人脑子里，人一走就断档。

八类怎么用？我给一个判断顺序：**先补最缺的那一类**。日志和监控往往最该先有——因为它们是排查一切问题的入口；然后是 IaC 与容器（把环境变成可重建的）；最后才是平台类工具（提升开发者体验）。**别一上手就上平台，你会在还没搞清楚自己要什么的时候，先被平台绑住。**

板书：八类：基础架构（CloudFormation・OpenStack）｜容器（Docker・Rocket・ElasticBox）｜编排（K8s・Mesos・Swarm）｜微服务平台（OpenShift・Cloud Foundry・Mesosphere）｜日志（Elastic Stack・Logentries・Splunk）｜系统监控告警分析（Prometheus・Icinga 2・Nagios Core・Zabbix・Cacti・Zookeeper）｜性能监控（AppDynamics・Datadog・DynaTrace・New Relic・CollectD・StatsD）｜知识管理与协作（MediaWiki・Confluence・Zoom）。

提问：一个团队现在只有代码仓库和一台测试机，先补哪一类工具最划算？

预设回答：

- 答"先上日志和监控"——很实在：没有日志，任何问题都只能靠猜；这是排查的入口。

- 答"先上 IaC，让环境可重建"——也合理，尤其环境经常被改坏的时候。

- 答"先上 Kubernetes，一步到位"——最典型的过度投入：环境还没标准化，先上集群编排，只会把混乱放大。

- 收口：**按"当前最痛"补，不按"最先进"补。**

易错点 / 考点：① 八类的名称要能列出，简答常考"测试基础设施工具可分为哪几类"；② 高频错答：把 Zookeeper 归到"性能监控"——课件把它列在系统监控/告警/分析这一类里，按课件口径记；③ 记住 Elastic Stack 的四个组件：ElasticSearch、Logstash、Kibana、Beats。

过渡：工具选完了，接下来把应用真正装进环境——容器化与集群部署。

【12.39】12.4.3 应用程序容器化及集群部署（3 分钟）

这一节讲应用怎么进容器。课件开头给了一个判断：**目前测试环境中所需要的各种测试工具及其依赖作为 Docker 容器来安装启动已经越来越普遍**。注意它说的是"测试工具及其依赖"，不只是被测应用——**连测试工**

具本身都在容器化。

然后课件给了一个具体例子，这个例子要讲透：**搭建一个 Selenium 的测试环境，不再需要下载 Selenium Server 以及支持特定浏览器的 Web Driver**，Selenium 官方提供了相应的 Docker 镜像，**我们只需要在安装好 Docker 之后执行 docker 命令拉取 Selenium 镜像，然后创建并启动 Selenium 容器即可**。

这个例子好在哪？它把"以前多麻烦、现在多简单"对比得非常清楚。**以前**：装 Java 环境 → 下载 Selenium Server → 为每个浏览器下载对应版本的 WebDriver → 保证驱动版本和浏览器版本匹配（这一步最容易翻车，版本不匹配就报莫名其妙的错）→ 配环境变量 → 启动服务。这一套做完，半天没了，换台机器重来一遍。**现在**：`docker pull` 一个镜像，`docker run` 起一个容器，浏览器、驱动、服务全在里面，版本是官方配好的一对。

这里的关键价值有三个。**第一，依赖打包**：镜像把"应用 + 依赖 + 配置"打成一个整体，不再有"我这能跑你那不能跑"。**第二，版本对应**：镜像由官方维护，驱动与浏览器的版本匹配由镜像保证，你不用再查兼容表。**第三，环境隔离与并行**：想同时跑 Chrome 和 Firefox 两套用例？起两个容器就行，互不干扰；想跑十个并行任务？起十个容器。

但要注意一个坑：**容器里的浏览器默认没有界面**。自动化测试通常用无头模式，可有些场景（比如需要截图对比、需要真实渲染）你得配虚拟显示或远程调试端口。这类细节在实操里很常见，遇到"容器里跑不起来浏览器"，先查这一条。

一句话落点：**容器化把"装环境"这个动作，从一门手艺变成了两条命令。**

> **板书：**课件结论："测试工具及其依赖作为容器安装启动越来越普遍"。Selenium 例子前后对比：以前（装 Java → 下 Server → 下各浏览器 WebDriver → 版本匹配 → 配环境变量 → 启动）／现在（docker pull → docker run）。三价值：依赖打包・版本对应・隔离并行。备注坑："容器内浏览器默认无界面"。

提问：为什么用容器跑 Selenium，比在每台机器上手工装一遍更可靠？

预设回答：

- 答"镜像里浏览器和驱动版本是配套的，不会出现版本不匹配"——正解，这是最实际的收益。

- 答"装得快"——也对，但只是次要收益，真正的价值在**一致性**。

- 答"容器性能更好"——错答：容器不会让 Selenium 跑得更快，甚至因为无界面、资源受限可能略慢。

- 收口：容器化的核心价值排序是——**一致性 > 隔离性 > 速度**。

易错点 / 考点： ① 要能复述课件那个 Selenium 例子，并说清"不再需要下载 Server 与 WebDriver"的原因；② 高频错答：以为容器化能提升测试执行性能——它提升的是**环境一致性与部署效率**；③ 记住"连测试工具也要容器化"这个观点，它常作为简答的加分点。

过渡： 既然要用 Docker，那几条最基本的命令必须张口就来。

【12.40】常用的 docker 命令（3 分钟）

这一页是命令清单，课件列了九条（其中 `docker run` 出现了两次，分别写成"创建并运行容器"和"创建并启动 docker 容器"，说的是同一个命令）。我按"生命周期的顺序"重新给你排一遍，这样好记。

第一步，拿到镜像：`docker pull [OPTIONS] NAME[:TAG|@DIGEST]` ——从镜像仓库拉取镜像。注意后面那个 `[:TAG|@DIGEST]`：**能写 tag 就别用 latest**，因为 latest 会飘，今天拉的和明天拉的可能是两个东西，测试环境最怕这个。要绝对确定，就用 digest（镜像的唯一指纹）。`docker images` 查看本地已有的镜像。

第二步，做自己的镜像：`docker build [OPTIONS] PATH | URL | -` ——从 Dockerfile 制作镜像。PATH 是构建上下文，`-` 表示从标准输入读。这一步的产物是镜像，不是容器，别搞混。

第三步，跑起来：`docker run [OPTIONS] IMAGE [COMMAND] [ARG...]` ——创建并运行容器。这是最常用的命令，也是最容易出问题的命令：端口映射、目录挂载、环境变量、后台运行、容器名，全在参数里。给测试场景一个典型组合的心智模型：**端口要映射出去（不然外面访问不到）、目录要挂载进去（不然数据随容器消失）、环境变量要传进去（不然连不上测试库）**。

第四步，看状态：`docker ps [OPTIONS]` ——获取容器列表。默认只显示运行中的，加参数才显示已停止的。排查"我的容器哪去了"，第一步就是它。

第五步，管生命周期：`docker start CONTAINER` 启动一个已存在的容器，`docker stop CONTAINER` 停止一个容器。注意 start 与 run 的区别：**run 是"创建并启动"（新容器），start 是"启动已有的"**——这个区别考试爱考。

第六步，分发：`docker push [OPTIONS] NAME[:TAG]` ——把本地镜像提交到镜像仓库，别人就能拉。团队共享测试环境镜像，靠的就是这一步。

最后给一条实践纪律：**命令要写成脚本，不要靠手敲**。你今天敲的 `docker run` 加十个参数，明天同事复现不出来；写进脚本、进版本库，才是可重复的。这一条其实就是 12.4 整节的精神——**凡是重复的动作，都该**

被代码固化。

> **板书：**按生命周期排：拉（pull〔写明 TAG 或 DIGEST〕・images）→ 造（build）→ 跑（run〔端口・挂载・环境变量〕）→ 看（ps）→ 管（start／stop；run＝创建并启动，start＝启动已有）→ 发（push）。底部："命令写成脚本，不进脚本的步骤都不可重复"。

提问：`docker run` 和 `docker start` 有什么区别?

预设回答：

- 答"run 是新建一个容器并启动，start 是启动已经存在的容器"——正解，这是最常考的一组。

- 答"一样，都是让容器跑起来"——典型错答：跑起来的结果相似，但一个是**创建新容器**，一个是**复用旧容器**，数据状态完全不同。

- 答"run 会重新拉镜像，start 不会"——这条不准确，run 默认用本地镜像（除非本地没有）。

- 收口：区分标准是"**是不是新容器**"，这直接决定了容器里的数据是不是从零开始。

易错点 / 考点：① 六组命令的作用要能对上，尤其 pull/build/run/push 的输入输出分别是什么（镜像还是容器）；② 高频判断："`docker build` 生成容器"——错，生成的是镜像；③ 记住实践建议：**镜像不要用 latest tag**，用确定的 tag 或 digest，这是环境可重复的前提。

过渡：单机跑容器只是开始，真实场景里要在一群机器上跑——Kubernetes 集群部署。

【12.41】Kubernetes 集群环境中部署（2 分钟）

这一页是集群部署的示意图，没有正文。它要你看懂一件事：**在 K8s 里"部署一个应用"，你写的不是命令，而是一份"期望状态"的描述**，

你提交的是一份资源描述：我要跑这个镜像、要几个副本、要什么端口、要多少 CPU 和内存。这份描述通常写成 YAML 文件，进版本库——这又回到了"基础设施即代码"。

接着 K8s 把它变成一层一层的对象。管这一组资源的是**部署（Deployment）**，它管的不是"一个容器"，而是"**始终保持 N 个副本**"这个承诺；往下是**副本集（ReplicaSet）**，负责数量维护：少了补、多了删；再往下是 **Pod**，真正承载容器的单元；对外则是**服务（Service）**，给这组会变化的 Pod 一个稳定的访问入口。

这条链有个非常实用的能力：**滚动更新**。你改了镜像版本，K8s 不是"全停再全起"——那会造成服务中断，它

是**一批一批换**：起一个新 Pod，等它健康了，再停一个旧的，如此往复。所以升级过程中服务始终可用。

对测试的落点有三个。**第一，每次测试可以从一个干净的命名空间开始**，跑完直接删掉，不留残留。**第二，可以精确控制副本数来构造负载场景**：要压测？把副本数调小，看它在资源紧张时的表现；要验证扩容？调大，看调度和启动耗时。**第三，日志和状态是分散的**：一个请求可能打到任意一个 Pod，排查时必须按"服务"聚合看，而不是登录某台机器。

一句话总结：**K8s 把"部署"从一个动作变成了一个持续维持的状态**。

> **板书：**对象链：Deployment（保持 N 个副本的承诺）→ ReplicaSet（数量维护）→ Pod（承载容器）→ Service（稳定入口）。能力：滚动更新（一批批换，服务不中断）。测试三落点：干净命名空间・用副本数构造负载・日志按服务聚合。

提问：在 K8s 里做性能测试，为什么要特别关注"副本数"这个参数？

预设回答：

- 答"副本数决定了后端能扛多少，副本少压力就打到一个 Pod 上"——正解，负载分布直接由副本数决定。

- 答"副本数越多性能越好，直接拉满"——错答：拉满之后你测的是集群容量，不是应用在目标规模下的表现，结论没法用。

- 答"副本数不影响，压测打的是 Service"——错在忽略了 Service 后面就是这些 Pod，数量直接决定单 Pod 压力。

- 收口：**压测前先确认副本数与生产一致，否则测出来的数字没法对应上线。**

易错点 / 考点：① 对象链的顺序要能说清：Deployment → ReplicaSet → Pod → Service；② 高频考点：滚动更新的含义与价值（升级不中断）；③ 案例题常见问法："集群里日志分散怎么排查"——答按服务聚合、借助日志采集与集中检索，而不是逐台上机。

过渡：机器和容器都就位了，可里面的软件还得装、配置还得刷——这就是自动配置。

【12.42】12.4.4 应用程序的自动配置（3 分钟）

这一页讲 Ansible。课件给的定义是：**Ansible 是一个基于 Python 开发的自动化运维的开源工具，提供远程系统安装、启动/停止、配置管理等服务，并且可以对服务器集群进行批量系统配置、批量部署和批量运行命**

令。

我把这句话拆成四个能力。**第一个，远程系统安装**：在一台控制机上发指令，让几十台目标机同时装软件包。**第二个，启动/停止**：管理服务的运行状态，该起的起、该停的停。**第三个，配置管理**：把配置文件按模板推送到目标机，保证每台的配置内容一致。**第四个，批量执行**：对一批机器同时跑同一条命令，这在排查和巡检时特别有用。

注意三个关键词：**批量**、**集群**、**自动化**。Ansible 的价值不在"能装软件"——手工也能装；价值在"**几十台机器装出来一模一样**"。这正好接上前面的主题：环境不一致的根源，就是"一台一台手工配"。

Ansible 有个很关键的特点，课件没写但你必须知道：**它通常不需要在目标机上安装客户端**，靠 SSH 连过去执行。这个特点的好处很直接：**目标机是干净的**，不用先装代理；**批量上手快**，一批新机器只要 SSH 通就能管。对比前面第三类工具里的 Chef、Puppet，它们一般要在目标机装 agent，走"目标机定期来拉配置"的模式——两种模式各有取舍，agent 模式适合长期稳定的大规模环境，无 agent 模式适合快速铺开和临时环境。

Ansible 的组织方式是"**清单 + 剧本**"：一份清单列出要管哪些机器（可以分组：测试组、预发组），一份剧本描述"对这些机器做什么"（装什么、配什么、起什么服务）。剧本是**可重复执行的**，这就是它的幂等性——跑一次和跑十次，机器状态一样。

最后落到测试：**你的测试环境可以被"重建"而不是"修复"**。环境脏了、配错了、被人改乱了，不用一点点查，直接跑一遍剧本，拉回标准状态。**这是本章"可重复"这条主线在配置层的实现。**

> **板书：**四能力：远程系统安装・启动/停止・配置管理・批量运行命令。三关键词：批量・集群・自动化。特点："无需目标机装 agent，走 SSH"（对比 Chef/Puppet 的 agent 拉取模式）。组织：清单（哪些机器、分组）＋剧本（做什么）。底部："环境可以重建，不用修修补补"。

提问：测试环境被人手工改乱了，用 Ansible 该怎么处理?

预设回答：

- 答"直接重跑剧本，把配置拉回标准状态"——正解，这就是幂等剧本的用法。

- 答"登录上去一条条改回来"——能修好，但下次还会乱，而且你未必知道被改了哪些。

- 答"重装系统最干净"——确实干净，但如果环境是容器或云主机，重建成本低还行；物理机重装就太重了。

- 收口：处理"环境漂移"的正确姿势是——**用声明式配置覆盖回去，而不是逐个修正**。

易错点 / 考点：① Ansible 的四个能力与"批量/集群"特征要能说清；② 高频考点：Ansible 与 Chef/Puppet

的差别（**有无 agent**）；③ 常考判断："Ansible 只能装软件"——错，它还管服务启停、配置推送、批量命令执行。

过渡：单个环节都会了，最后一页把它们串成一条线——CI/CD 流水线。

【12.43】12.4.5 CI/CD 流水线（3 分钟）

课件说：**为了能实现快速交付、持续交付，需要一系列自动化技术的支持和融合，包括持续构建、持续集成、持续测试、持续部署、持续运维。之所以称之为流水线，就是像工厂里面生产产品的流水线一样，把软件应用的整个生命周期里的各项任务以高度自动化的形式快速、有序的执行。**

持续构建：代码一提交就编译打包，保证"能不能编过"这件事随时有答案。**持续集成**：把大家的改动频繁合到一起，并且立刻验证——这是防"合并地狱"的唯一办法。**持续测试**：每一轮构建都自动跑测试，包括单元、接口、回归——它是流水线上的质量门。**持续部署**：测试通过的制品自动推到环境里，不用人手工发版。**持续运维**：上线之后持续监控、持续观察、出问题能快速回滚。

课件用了工厂流水线的比喻，这个比喻要抠一下：工厂流水线的本质不是"快"，是**每一道工序的输出都是下一道的标准输入，且每道工序都有检验。中间任何一道没有检验，缺陷就往下流，流到最后返工成本最高。**

课件点明：**Jenkins 2.0 是最常用的 CI/CD 流水线中的调度管理工具**。注意"调度管理"四个字：Jenkins 本身不编译、不测试，它负责**按顺序触发、按条件判断、收集结果、通知人**。真正干活的是它调用的那些工具——这正好解释了为什么前面要讲那么多工具：**流水线是把它们串起来的胶水，不是替代品。**

我给一条判断流水线好坏的实用标准：**从提交代码到拿到测试报告，需要多久、需要几个人参与**。时间是反馈速度，人是自动化程度的反面指标。理想状态是**几分钟、零人工**。做到这一步，团队才敢频繁提交，而频繁提交正是持续集成的前提——这是一个正循环。

最后一句提醒：**流水线本身也要当代码管理**。配置写进版本库、改动走评审，否则流水线会变成没人敢改的黑盒。

板书：五个持续：构建 → 集成 → 测试（质量门）→ 部署 → 运维。工厂比喻的本质："每道工序输出是下道输入，且每道都有检验"。Jenkins 2.0＝调度管理（按序触发・条件判断・收集结果・通知）。判断标准："提交到报告：几分钟？几个人？" 备注："流水线也要当代码管"。

提问：如果流水线每次都要人工确认才继续，问题出在哪？

预设回答：

- 答"自动化程度不够，某个环节还没法自动判断"——正解方向：要么测试没自动化，要么判断标准不明确。

- 答"人工确认更安全，重要环境就该这样"——有道理，**生产发布**加人工审批是合理设计；但测试环境的每一环都要人点，就失去流水线的意义了。

- 答"没关系，反正能跑通"——能跑通只是及格线，反馈周期被人工卡住，前面所有自动化的收益都被吃掉。

- 收口：区分场景——**测试环境追求全自动，生产发布可以有门禁**。

易错点 / 考点： ① 五个"持续"的名称与顺序要能默写；② 高频错答：把"持续交付"与"持续部署"混为一谈——交付是"随时可发"，部署是"自动发出去"；③ 记住 Jenkins 的定位是**调度管理**，不是构建或测试工具本身。

过渡：所有零件都讲完了，最后一页把这一章串成一句话。

【12.44】Jenkins+Docker 实现持续集成（2 分钟）

这一页是实操页，课件只给了一句：**具体搭建，见教材 P341—350**。也就是说，动手部分交给教材和实验，我在这里把"要搭出什么、容易卡在哪"讲清楚，你回去照着做就不会瞎撞。

先说要搭出什么：**Jenkins 做调度，Docker 做执行环境**。为什么是这两个搭配？因为 Jenkins 的调度能力需要"干净、可重复的执行环境"，而 Docker 正好提供这个。传统做法是在 Jenkins 机器上装一堆构建工具和依赖，时间一长，那台机器就成了"谁都不敢动"的古董；容器化之后，每次构建都在一个**全新的容器**里跑，构建完就销毁，**构建环境本身也变成了可重复的**。

一条典型流水线跑这些阶段：**拉代码 → 构建（编译打包）→ 静态代码分析 → 单元测试 → 打包制品 → 构建镜像 → 推送镜像 → 部署到测试环境 → 通知结果**。这条链上的每一站，都对应本章前面讲过的一类工具：代码仓库、构建工具、静态分析工具、单元测试框架、镜像仓库、容器编排、通知渠道。

三个常见坑，我提前给你。**第一，容器里的权限与目录**：构建产物要能被 Jenkins 取到，就得挂载工作目录，否则容器一销毁，产物跟着没了。**第二，镜像里的依赖要提前装好**：每次构建都现装依赖，时间全耗在这上面，正确做法是把依赖写进镜像，用缓存。**第三，凭据管理**：镜像仓库的密码、部署的密钥，不能明文写进流水线脚本，要用凭据管理功能注入。

还有一个观念要纠正：**不要把测试写在流水线的最后**。很多人习惯"先构建、再部署、最后跑测试"，这是反的。**测试要在部署之前**——测试不过，就不该往下走。这就是"质量门"的位置，也是流水线存在的意义：不是把流程串起来就行，是**在每一道门前替你拦住问题**。

一句话总结这一页：**Jenkins 决定"什么时候做什么"，Docker 保证"在哪做、每次都一样"。**

> **板书：**分工：Jenkins＝调度，Docker＝干净可重复的执行环境。流水线阶段：拉代码 → 构建 → 静态分析 → 单元测试 → 制品 → 镜像 → 推送 → 部署 → 通知。三个坑：工作目录挂载（产物不丢）·依赖预先装进镜像（省时间）·凭据注入（不写明文）。原则："测试在部署之前＝质量门"。

提问：为什么说"测试要放在部署之前"，而不是部署完再测?

预设回答：

- 答"测试不过就不该上线，得先拦住"——正解，这就是质量门的含义。

- 答"部署完再测也一样，反正都要测"——不一样：不合格的版本已经进了环境，可能覆盖掉正在用的好版本，还浪费一次部署时间。

- 答"先部署才能测，因为要跑端到端"——部分成立：端到端测试确实需要部署后的环境，所以完整流水线通常是"单元/接口测试在构建后 → 部署到测试环境 → 端到端测试 → 再往生产"。关键是**每一层都有门**。

- 收口：位置不是绝对的，原则是**越早拦越好，越贵的门越靠后**。

易错点 / 考点：① 要能说出流水线的典型阶段顺序；② 高频考点：为什么用容器做构建环境——**构建环境本身要可重复**；③ 案例题常见问法："流水线里测试总是不稳定"，排查方向包括环境不一致、依赖外部服务、用例间数据冲突（呼应 12.2.4 的数据隔离）。

过渡：整章讲完了，我们把这一章的四句话收拢一下。

【12.45】本章小结（3 分钟）

这一页是本章小结，课件四条，我按"为什么—有什么—怎么造—怎么串"给你重新组织一遍，这四句话也是你复习的骨架。

第一条，测试基础设施很重要，介绍了测试基础设施的各项要素，包括 **IT 基础架构、软件资源和数据资源**。这一条对应 12.1 和 12.2。你要能回答三个问题：它为什么重要（三点：是测试的基础、影响研发效能、保障

运维）；它由什么构成（硬件、网络、软件、数据四类）；判断它好不好的标准是什么（能不能**可重复地**重建一个**准生产**级的环境）。

第二条，提供了测试基础设施搭建的方法和技术，包括**虚拟机技术、容器技术以及服务虚拟化技术**。这一条对应 12.3。你要能分清三者的抽象层次：虚拟机虚拟硬件、容器虚拟操作系统环境、服务虚拟化虚拟外部服务；还要知道每一层的代表工具（QEMU-KVM、Docker、K8s、WireMock/Hoverfly）。

第三条，基础设施即代码（IaC）有利于实现基础架构的自动部署，有四类工具——基础架构即代码、配置即代码、容器即代码、管道即代码。这一条对应 12.4.1 和 12.4.2。这是全章最该背下来的一组分类，因为考选择题、简答题、案例题都可能用到。

第四条，详细讲解了如何搭建一个持续交付流水线。这一条对应 12.4.3 到 12.4.5，也是实验 8 的内容。

如果让我用一句话概括这一章，我会说：**测试基础设施的终极目标，是让"一套干净的、和生产同构的环境"变成一条命令、一次点击、甚至一次代码提交的自动结果。** 达到这一步，测试才可能真正快起来、稳下来；达不到，测试就永远在等环境。

顺便把本章的"常见坑"也收个尾，五个词你记一下：**环境漂移**（越用越不像）、**数据污染**（测试数据互相踩）、**共享环境冲突**（多人一环境互相干扰）、**配置不一致**（参数在环境间不同）、**成本失控**（云资源忘关）。这五个坑，每一个都有对应的解法，而解法全都指向同一件事——**把它写成代码**。

> **板书：** 四句话：① 重要＋要素（硬件・网络・软件・数据）② 搭建技术（虚拟机・容器・服务虚拟化）③ IaC 四类工具（基础架构・配置・容器・管道）④ 持续交付流水线。一句话总结："让干净且同构的环境，成为一条命令的结果"。五个坑：环境漂移・数据污染・共享环境冲突・配置不一致・成本失控 → 解法都指向"写成代码"。

提问： 用本章的话说，"一个好的测试环境"最重要的一个特征是什么？

预设回答：

- 答"可重复，随时能重建出一模一样的"——最贴题，它把 IaC、容器、快照、模板全都串起来了。
- 答"和生产一致"——也重要，但"一致"要靠"可重复"来保证：不能重建，一致性就只能靠运气维持。
- 答"稳定不宕机"——是基本要求，不是核心特征：稳定的机器如果配置是手工调的，照样不可复制。
- 收口：**可重复是根，一致、稳定、快速都是它的结果。**

易错点 / 考点： ① 小结四条要能复述，尤其"四类 IaC 工具"和"三类搭建技术"；② 高频判断："测试基础设施

只需要硬件和软件"——漏了网络与数据；③ 本章综合题的标准答法：先讲要素（四类），再讲手段（虚拟化三层），再讲自动化（IaC 四类＋流水线），最后落到"可重复、准生产、成本可控"三个评价维度。

过渡：理论到这儿收口，下一页把本章的东西变成你手上的一次实验。

【12.46】实验 8: Jenkins+Docker 实现 Java 应用的持续构建（3 分钟）

这一页是实验 8 的任务说明：**采用 Jenkins Pipeline 和 Docker 搭建一个 Java 应用的 Jenkins 流水线，完成一个 Java 应用从编译、静态代码分析、单元测试、打包的过程。**这一页信息量不大，但它是整章的落点，我把任务拆开讲，你照着做。

先看四个动作：**编译、静态代码分析、单元测试、打包**。**编译**验证"代码能不能变成可执行产物"；**静态代码分析**在不动程序的前提下查代码规范与潜在缺陷（第 3 章讲过静态测试，这里是它的工程化形态）；**单元测试**验证最小代码单元的行为，并且要产出覆盖率数据（第 5 章的内容）；**打包**产出可交付的制品。

为什么用 Docker？**为了让这四个动作跑在一个干净、可重复的环境里**。你的本机装了什么 JDK、什么版本的构建工具，都不影响结果——因为流水线跑在镜像里。这也是本章反复强调的那句话：**环境是代码的一部分。**

为什么用 Jenkins Pipeline？因为流水线要**写成代码**（这就是 12.4.1 讲的"管道即代码"）。写成代码的好处是：能进版本库、能评审、能复用、能回滚。在界面上点出来的流水线，换个项目就得重新点一遍。

交付物我给一个清单，你按这个交：**第一，流水线脚本**（Pipeline 文件），要能看出四个阶段；**第二，一次成功构建的日志**，要能看到每个阶段的输出；**第三，单元测试与覆盖率报告**；**第四，静态分析报告**；**第五，一份简短说明**，写清镜像怎么来、凭据怎么配、遇到过什么问题怎么解决的。

评分我会看三点：**流水线能不能一键跑通**（最重要）、**四个阶段是否齐全且顺序合理**、**报告能不能说明质量状况**。第五项"说明"我也看——因为把踩过的坑写清楚，是工程师的基本功，这一条正好呼应本章开头讲的"看不见的地基"。

最后一个提醒：**先让它跑通，再让它跑好**。很多同学一上手就想把并行、缓存、通知全加上，结果一样都没跑通。**最小可用流水线优先**，这是所有工程实践的通则。

板书：任务四动作：编译 → 静态代码分析 → 单元测试 → 打包。为什么 Docker："干净可重复的执行环境"；为什么 Pipeline："管道即代码：进库・评审・复用・回滚"。交付五项：Pipeline 脚本・成功构建日志・单测与覆盖率报告・静态分析报告・问题说明。评分三点：一键跑通・阶段齐全有序・报告能说明质量。

提问：实验里如果单元测试失败了，流水线应该继续往下打包吗？

预设回答：

- 答"不该，测试失败就该停，这就是质量门"——正解，也是流水线的基本纪律。

- 答"可以先打包，反正只是实验"——实验里放松，工作里就会形成坏习惯；而且"打包了但测试没过"的制品一旦流出去，就是事故源头。

- 答"看失败原因，环境问题可以重试"——这个补充很专业：**失败要分类**，代码问题必须停，环境抖动可以重试，但重试要有次数上限。

- 收口：流水线的价值不在"自动跑"，在"**自动判断能不能往下走**"。

易错点 / 考点：① 四个动作的名称与顺序要能写对；② 高频错答：把"静态代码分析"当成"跑程序检查"——它是**不运行程序**的检查，与第 3 章静态测试一致；③ 实验报告的评分重点在**可复现**：别人拿着你的脚本能不能跑出同样结果。

过渡：这一章到这里就讲完了，最后留一句话给大家。

【12.47】感 谢 聆 听（1 分钟）

第 12 章讲完了。

这一章从头到尾其实只在说一件事：**测试能不能被信任，取决于它跑在什么样的地上**。方法再漂亮，环境不对，结论就不可信；用例写得再好，数据不真实，测了也白测。

我希望你带走三个判断：**第一，环境要可重复**——不能重建的环境，一致性只能靠运气；**第二，数据要合规**——真实数据能不用就不用，必须用就先脱敏；**第三，资源要算账**——能复用的不重复造，能自动清理的不靠人记。

下一章我们会走到测试的另一端：怎么把测试的结果管起来、把缺陷跟到底、把质量说清楚。**先把地基打好，后面的楼才立得住。**

实验 8 的流水线请尽快跑通，遇到卡住的地方记下来，下次课我们集中答疑。

板书：三个带走：环境可重复・数据要合规・资源要算账。一句话："测试能不能被信任，取决于它跑在什么样的地上。"

> **提问：**这一章里，你打算先在自己项目里动手改的一件事是什么？

> **预设回答：**

- 答"把环境搭建步骤写成脚本或 Dockerfile"——最有性价比的第一步，从"手工装"变成"跑脚本"。

- 答"先把测试数据脱敏、并且做成可回滚的基线"——很务实，直接解决数据污染。

- 答"先不动，等有机器了再说"——最常见的拖延：脚本化和 Dockerfile 不需要新机器，一台自己的电脑就能开始。

- 收口：**改一件小事，比规划一套大方案更有用。**

> **易错点 / 考点：**① 复习时把本章三组清单过一遍：四类要素、三类搭建技术、四类 IaC 工具；② 记住五个坑的名字与解法方向（环境漂移、数据污染、共享环境冲突、配置不一致、成本失控 → 全部指向"写成代码"）；③ 案例题收尾句可用："**以可重复为核心，以准生产为标准，以自动化为手段，以合规与成本为边界**。"

> **过渡：**下一章进入测试结果与缺陷管理，把这一章搭好的地基真正用起来。

四、思政落点清单（共 4 处）

序号	所在页	落点主题
1	12.4	脚手架没人夸，可它一塌全工地停摆。测试基础设施就是软件工程里"看不见的地基"——
2	12.5	环境一致性说到底是"对结论负责"。同一批用例在两个环境跑出两个结果，最省事的做法
3	12.13	把生产数据拷进测试库，图的是省事，赌的却是别人的隐私。手机号、身份证、住址、交易
4	12.17	一台服务器长期只用到两三成算力，账面上看是"公司有钱"，实际上是电、机位、空调、