《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）

> 有个场面，学完方法的人一定会撞上。用例设计得挺漂亮，一进实验室，半天跑不起一条。

>

> 你以为是代码写错了？多半不是。机器没有、环境不对、数据拿不到、依赖连不上，四条里占一条，测试就得靠人情和加班撑着。撑出来的结果还有个毛病：不可重复。今天你能跑，明天换个同事跑不起来，谁也说不清差在哪。不可重复的活，做一百遍也攒不下经验。

>

> 前十一章讲的是怎么测。用例怎么设计、方法怎么选、流程怎么排，单元、集成、系统一路走下来，第七章攻性能和安全的硬骨头，第十一章把用例当团队资产管起来。这些照样要用。可它们全得落在一块能跑的地方上。方法是一条腿，地方是另一条腿，缺一条就站着晃。

>

> 第 12 章叫部署测试基础设施，管的就是这块地方：在哪测、拿什么测、怎么让它随时重建。四块内容先摆出来：重要性、要素、虚拟化技术、自动部署。前两块想清楚要什么，后两块解决怎么造出来。

>

> 这一章不教新的测试方法。它教的是把方法放上去的那一层。方法课考你会不会设计用例，这里考你能不能

> 让同一批用例稳定地跑一百遍，跑出一样的结果。一条用例卡在环境上，你连它写得对不对都不知道。

>

> 带着一个问题往下听。你上一次因为环境白耗半天，耗在哪一步？装不上，连不通，还是数据跟别人打架？这一章讲完，你得能自己回答，并且知道下一步先改哪儿。

> **提问：**回想一次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"——最典型的一类，落在资源与配置上，12.2 的软件资源、网络资源讲的就是它。

- 答"两个人共用一个测试环境，我把他的数据冲了"——共享环境冲突，本章会反复回到这件事。

- 答"环境问题归运维管，不用在测试课上讲"——这条要改。测试人员一天里相当一部分时间花在环境上，你不掌握它，就只能等人来救。

- 收口：三类回答正好对上本章三条主线：要素、搭建、自动化。

> **易错点 / 考点：**① 本章管的是"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 四个板块的名称要能默写：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，以及把这些组织起来的管理与自动化。

> **过渡：**先把地图铺开，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）

> 四块内容，先说顺序。顺序不是排版安排，它本身就是一条思路：为什么值得花钱、由什么组成、怎么造出来、怎么自动化。

>

> 12.1 讲重要性。它会说清定义，说清它为什么是测试的基础，再说它怎么影响研发效能、怎么保障运维。这一块是观念。观念不通，后面每一笔投入都会被当成浪费，申请机器像讨饭。排期也一样：不说明环境要占掉多少时间，测试就永远排在最后。

>

> 12.2 讲要素。硬件、网络、软件、数据，四类拆开看。这一块是清单，你要能在脑子里列出来，还要能拿它去检查别人的环境缺什么。缺哪一类，整条链都会在某个不起眼的环节上断掉。

>

> 12.3 讲虚拟化技术。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？今天的测试环境基本不靠再买一台机器，靠的是把资源切出来、复制出来、编排出来。切是虚拟机，复制是镜像，编排是集群。

>

> 12.4 讲自动部署。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。落到最后是一条能自己跑起来的流水线。写得下、跑得起来、出错能回滚，三件事都有代码可查。

>

> 四块连起来一句话：先想清楚它是什么、由什么组成，再用虚拟化造出来，最后用代码把它自动化。答题也照这个骨架写，别写成四段互不相干的定义。四块的顺序倒了，后面每一段都会跟着错位。背下来，简答不慌，案例也能照着铺。这一章后面每一页，都挂在这四块里面的某一处。

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

预设回答：

- 答"多买机器"——短期见效。但环境靠人手配，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压到接近零，12.4 讲的就是这个。

- 答"先看现在卡在哪"——最成熟。先分清瓶颈是资源不够，还是环境不可重复，再决定钱往哪花。

- 收口：这一章反复用的判断标准只有一句——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）

为什么讲重要性要花四页？直接把技术讲完不行吗？

>

因为这一节是申请资源时要说的话。你说不清它值多少，就申请不到机器和人。四个小节正好是四套说法，对着四类听众：跟老板讲钱，跟测试组讲能不能测，跟运维讲线上稳不稳，跟开发讲提交完多久能看见结果。

>

12.1.1 讲定义。先把话说准，它到底包含什么。定义不清，讨论立刻跑偏：有人以为它等于机房，有人以为它等于测试工具的别名，还有人以为它只跟开发有关。

>

12.1.2 讲它是测试的基础。讲它在每个阶段起什么作用，还讲一句硬话：系统测试和验收测试的环境要最大限度接近真实，叫准生产环境。用错环境，会引出一串问题。

>

12.1.3 讲它影响研发效能。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，五条都算效率。两千条用例跑一夜，早上出报告，这算效率；跑一夜没人看，那叫浪费。答简答时五条分开写，比合成一句值钱。

>

12.1.4 讲它保障运维。把它从研发侧伸到生产侧，落到 DevOps 那条完整的链上。这一小节最容易答空，答成一句"保障运维很重要"。线上告警响了，谁来看、看哪个指标，测试阶段没定，出事只能互相问。

>

走向很清楚：先定义清楚，再说它撑住什么，再算它省了多少，最后看它对线上的价值。前两节讲对不对，后两节讲值不值。四小节按这个顺序出简答，段落就能对上题眼。

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"要向领导申请预算"——很实在。说不清价值就申请不到资源，这一节给你的就是申请材料的逻辑。

- 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。简答和案例分析常从"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：四页就是四类标准答案的素材。

> **易错点 / 考点：** ① 四个小节的标题记准，特别是"影响研发效能""保障运维"这两个提法；② 案例题常问"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"答成空话，要落到成本、效率、风险三件事上。

> **过渡：** 先把定义说准，后面所有讨论都站在它上面。

【12.4】12.1.1 测试基础设施的定义（2 分钟）

> 先把定义念一遍，念的时候留意四个动词。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，并且与研发环境、运维环境集成的综合性平台。

>

> 第一个，支持测试运行。有机器、有网络、有被测系统、有依赖服务，用例能真跑起来。这一层最容易想到。

>

> 第二个，支持测试开发。写自动化脚本、调试用例、管脚本代码。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，自动化无从谈起。

>

> 第三个，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、缺陷走到哪一步。

>

> 第四个，与研发、运维集成。它不能是一座孤岛。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一版。孤岛式的环境，就是"测的不是上线那一版"这种事故的温床。

>

> 定义最后三个字是平台。平台的意思是被共建、被维护、被持续改进的一整套东西，不是谁临时凑的几台机器。下一句结论很干脆：测试基础设施是测试执行的基础。

>

> 定义里没有一句提到买多少台机器。它说的是能力，不是设备清单。

>

> 落到日常：你要测一个登录，得有能访问的地址、能登录的用户名、能看日志的地方、能查库的权限。四样缺一样，用例就停在纸面上。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。

> **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，规模一大就散。

- 答"不算，缺测试管理和集成"——按定义确实缺两块。缺了集成，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，是信息有没有形成闭环。

> **易错点 / 考点：**① 定义里三个"支持"加一个"集成"要能背，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；③ 口诀：跑得起来、写得下去、管得清楚、对得上号。

> **过渡：**定义清楚了，接着看它在生命周期的每个阶段到底起什么作用。

【12.5】12.1.2 测试基础设施是测试的基础（3 分钟）

> 同一批用例，两个环境，跑出两个结果。你信哪个？

>

> 先看第一句：测试基础设施贯穿软件开发的各个阶段，每个阶段的影响不一样。单元测试阶段要的是能编译、能跑覆盖率、能 Mock 掉外部依赖，一台开发机就够，要求是快。集成测试阶段要多个服务互相调通，要有

> 消息队列、数据库、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。

>

> 这就是那句硬要求：系统测试和验收测试的环境，要最大限度接近实际环境，所以叫准生产环境，也叫类生产环境。"最大限度接近"要接近什么？四个对齐。软件版本组合对齐，操作系统、中间件、数据库版本得对上；数据规模同级，生产一千万条，你拿一百条测，执行计划完全不同；网络与安全策略同样，限流、防火墙、证书、代理都得有；部署形态一样，集群几台、负载均衡怎么配。

>

> 第三句：用错环境会引出一串问题。第一类是假阳性，环境配错导致用例失败，你查两天代码，最后发现是配置；第二类是假阴性，环境太干净、数据太少，性能和并发问题一个都没露出来，测试全绿，上线就炸；第三类是结论不可信，两个环境两个结果，团队开始怀疑用例本身。

>

> 业界按用途分四级：开发环境、测试环境、预发环境、生产环境。开发环境允许乱，生产环境一个字都不能乱，预发是两者之间的桥。很多事故是把测试环境当开发环境用，谁都能改配置，最后没人知道当前是什么状态，这就叫环境漂移。

提问：测试环境用生产的一百条样例数据跑通了全部用例，能不能说明上线没问题？

预设回答：

- 答"能，功能都对了"——最典型的错答。一百条数据跑不出索引失效，跑不出深翻页变慢，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"——抓住要点了。规模是准生产四对齐里的一条，缺了它，性能与稳定性结论不可信。

- 答"要看测什么，功能可以，性能不行"——最准确。判据不同，对环境的接近程度要求就不同。

- 收口："环境够不够用"没有统一答案，要先问你要下什么结论。

易错点 / 考点：① 准生产环境、类生产环境这两个术语要能写出，并说清它服务的是系统测试与验收测试；② 判断陷阱"测试环境应与生产环境完全一致"——错在"完全"，课件用的词是"最大限度接近"，成本上也不可能完全一致；③ 简答答"错环境的危害"分三步：结果不可信、返工成本、决策失真。

【思政落点 1】：同一批用例跑出两个结果，最省事的做法是挑好看的那个报上去。往前一步，后面所有决策

> 就建在假数据上。测试报告里写清结果来自哪个环境、什么版本、多大口径的数据，不修饰、不挑选、不隐瞒不利结果。这不是老实，是让团队敢相信你那份报告的唯一办法。环境一致性，最后就一句：对结论负责。

> **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）

> 两千条自动化用例，一天只跑得完一遍。这算有自动化吗？

>

> 课件给五条，末尾还留着省略号，说明这类好处还能往下列。

>

> 第一条，提供工具和环境支持早期测试，让测试尽早开始。测试左移不是口号，它要条件：代码一提交就能自动构建、自动跑单元测试、自动部署出这个中间版本。缺了这些，早开始只是早开会。

>

> 第二条，提供稳定可靠的测试环境。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，一天能干活的时间剩一半。更糟的是它会养出坏习惯：看见失败先怀疑环境，不先怀疑代码。这个习惯一旦成形，真缺陷就从眼皮底下走过去。

>

> 第三条，用虚拟化技术和云平台提高软硬件资源利用率。一台物理机切成多个虚拟机或容器，按需分配，高峰开出来，闲时放掉。

>

> 第四条，用 Mock 工具这类服务虚拟化技术模拟外部依赖。今天几乎没有系统是孤立的，你要调支付、短信、征信、第三方地图。这些依赖不能随便压测、不能造异常、还可能按次收费。服务虚拟化把它们假装出来，12.3.5 专门讲这一条。

>

> 第五条，提供并行测试能力，缩短执行时间。同一批用例拆十份同时跑，一天变两小时。并行不免费：用例之间不能互相踩数据，环境要隔开，否则并行度越高冲突越多。

>

> 五条收成一句：基础设施决定测试的起跑线和上限。起跑线是能不能早开始，上限是能不能快、能不能大规模重复。省略号那一块可以自己补：可观测性、可追溯、团队协作，都算。

提问：团队说"自动化用例写了两千条，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，要么把慢的依赖虚拟化掉。

- 答"机器不够就买"——能解决一部分，可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：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"基础设施如何提升研发效能"；② 第四条的关键词是服务虚拟化、Mock，和 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，另一半还没讲：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）

线上出事，第一句话常常是：测试怎么没测出来。

>

> 课件只有一句，信息量不小。测试基础设施不只支撑研发环境，还延伸到生产环境，成为支撑产品运维的一部分，构成一条贯穿研发和运维的完整 DevOps 测试基础设施。这句话把测试的边界往前推了一格。

>

> 三个词拆开看。第一个是延伸。老观念里测试环境是研发的后院，上线就与它无关。今天不是：线上监控告警、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。你在测试环境没练过监控，生产出事时连在哪看指标都不知道。

>

> 第二个是支撑运维。运维最怕两件事：不知道自己有什么，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要立起来：环境用代码描述，清单自动生成；监控指标在测试环境接好，生产只换个地址。

>

> 第三个是 DevOps。它的核心是把开发与运维之间那堵墙打通，让一次变更从提交到上线是一条自动流水线，不是一串邮件和工单。定语里还留着一个词：测试。意思是这条流水线上，测试不是最后一道关卡，是每一环都嵌着的质量门。

>

> 判断标准很简单：一套基础设施只有研发用得上、运维完全用不上，那它只做了一半。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，两边通用。

> **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"线上才有的问题，测试环境造不出来"——对。真实流量特征、网络抖动、数据分布，得靠准生产环境和流量回放来补。

- 答"监控没接，问题发生了没人看见"——这是保障运维那一条的直接体现。可观测性也是基础设施的一部分。

- 答"测试要保证线上不出事"——过于绝对。测试能保证的是在给定环境和范围内验证过，环境与生产差得越远，这份保证的边界就越窄，所以边界要写清楚。

- 收口：后面反复强调环境即代码，道理就在这儿：只有同源，才能同构。

> **易错点 / 考点：**① "DevOps 测试基础设施"这个完整提法要记住，它把边界从研发扩到运维；② 判断"测试基础设施只服务测试团队"——错；③ 案例题常从"测试环境通过、生产环境故障"切入，答三段：环境差异、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，看由哪几样东西组成。

【12.8】12.2 测试基础设施的要素（1 分钟）

> 把基础设施拆开，拆到底只有四类东西：硬件资源、网络资源、软件资源、数据资源。四类之外没有第五类，

测试工具、用例管理这些都在软件资源里。

>

先说关系，不然你会觉得它们是并列的四张清单。它们一层压一层。硬件是有没有机器，网络是机器之间通不通，软件是机器上跑什么，数据是跑起来处理什么。任何一层缺，上面那层都白搭：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一节？因为要素就是检查表。以后你去评估一个团队的环境能力，就照四类打勾：硬件够不够、网络通不通、软件版本对不对、数据有没有、合不合规。四栏打完，短板在哪一眼就看出来。打勾的顺序也有讲究：先看有没有，再看对不对，最后看是不是同一个版本、同一口径。

>

四类里最容易被放过去的是数据。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。它不像机器能一次性买齐，得持续造、持续脱敏、持续记版本。这也是为什么数据那一节后面还要接一段流量回放。

>

四类的关系也别记反。有人问"数据重要还是机器重要"，这问题问错了。缺哪一类，环境就在哪儿掉链子，判断的是短板，不是排序。真要排，就按你要下的结论排：测性能，数据在前；测功能，软件版本在前。四类齐了再逐项看选型，才知道哪一项最花钱。

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"——最常见也最对。机器能买、软件能装，数据得一点点造和治理，还牵着合规。

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

- 答"硬件，云上一点就有"——反过来想：硬件最容易解决，所以它最没必要成为瓶颈。瓶颈还停在硬件，多半是先把机器加了，前三类没动。

- 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——课件把它归在软件资源里；③ 简答可以用一句话收：机器、通路、平台、数据，缺一层，上面

全空。

> **过渡：**从最底层开始，硬件到底有哪几种形态可选。

【12.9】12.2.1 硬件资源（3 分钟）

买机器这件事，最容易花错钱。

>

> 课件列四种形态：机架式、刀片式、传统式、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。按选型讲，因为你们以后真正要做的决定就是选哪种。

>

> 传统式服务器就是塔式机，一个机箱一台机器，台式机的放大版。便宜、简单、插电就用，放办公室都行。缺点是密度低、占地方，电源散热不共享。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。

>

> 机架式按 U 算高度，标准宽度能进机柜。它的意义是可规模化：一个柜子里整齐排十几台，供电、散热、布线都规范，机房管得过来。中型团队自建的测试机房基本是这个形态。

>

> 刀片式是一个机箱插多块刀片，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。密度最高，布线最干净，管理统一。代价是机箱成了单点，机箱一坏整框受影响；不同厂商的刀片箱基本不通用，采购容易被绑住。

>

> 云真机要重点说。它不是普通服务器，是跑在云上的真实手机，远程连上去操作。移动端测试的死结是机型太多：不同厂商、不同系统版本、不同屏幕、不同芯片，真实表现差得远。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，想测哪台连哪台。

>

> 选型三步：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、人力一起算，再对比按需租用。别只比机器单价，电和人是持续的。

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

预设回答：

- 答"先租云真机"——多数情况更合理。机型覆盖是刚需，自购手机折旧快、管理成本高，人少更要按需租。

- 答"先买服务器，机器在自己手里踏实"——可控性更好，但五个人维护机房不现实，电费、值守、备件都是隐藏成本。

- 答"都要，预算够"——预算够也要排优先级，先把当前最大的瓶颈解决掉，别一次铺开。

- 收口：硬件选型不是买贵的，是买对的，对的标尺就是当前瓶颈。

易错点 / 考点：① 四种形态名称写全，云真机常作为移动测试的硬件资源出现；② 高频混淆：刀片式和机架式——区别在是否共享机箱的电源与散热；③ 判断"自建机房一定比云上便宜"——错，规模小时云更省，规模很大自建才划算。

过渡：机器有了，机器之间还得通，下一页看网络资源为什么单列一类。

【12.10】12.2.2 网络资源（3 分钟）

本机 curl 通，同事那边超时。这类事一周能碰上好几回。

>

课件列四组网络资源，四组对应四个问题：多快、多远、怎么通、能不能过。

>

第一组是带宽，100M、1G、10G，说的是速率。为什么测试要在意带宽？很多所谓性能问题，根子在带宽：大文件上传下载、视频流、批量数据同步，卡的不是 CPU 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不成立。

>

第二组是覆盖范围：局域网、广域网、无线网。局域网在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；无线网抖动大、切换多。这一组提醒一件事：测试环境在哪儿，决定你能验证什么。全在一个机房内网，你就永远测不出跨地域调用的超时。

>

> 第三组是网络协议：HTTP/HTTPS、TCP/UDP、DNS，还有各种内部协议。协议决定了你怎么抓包、怎么造异常、怎么分清是握手失败还是应用层报错。安全测试更离不开它。

>

> 第四组是防火墙、代理服务器、网关。这一组最容易被推给运维，实际上它是故障高发区。常见三类：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，日志里的来源 IP 全是网关的，排查时分不清是谁。

>

> 网络资源在测试里有两面。一面保障连通，要通的必须通，还要写进文档。另一面主动制造不通：限速、丢包、断网、延迟，拿来测系统的健壮性。第二面常被忽略，可它才是网络作为测试手段的地方。

> **提问：**服务在本机 curl 一切正常，同事的机器访问超时，你先查哪三样？

> **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。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"——最差的第一步。现场还没看清就动手，等于把线索抹掉。

- 收口：分层定位，本机通不通、同网段通不通、跨网段通不通，一层层缩范围。

> **易错点 / 考点：**① 四组网络资源列全，简答常考"网络资源包括哪些"；② 判断"网络资源只指带宽"——错，还包括覆盖范围、协议、边界设备；③ 两面性要记：保障连通，以及主动制造异常，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，下一页进入装什么、装哪个版本。

【12.11】12.2.3 软件资源（3 分钟）

> 装什么、装哪个版本，这两问能吵一下午。

>

> 课件列六项：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。

>

> 操作系统：Windows、各类 Linux、macOS，还有移动端的 Android 和 iOS。要在意的是版本，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，系统调用和文件权限的表现也不同。

>

> 数据库：类型、版本、字符集、时区、隔离级别。这五项里任一项不一致，都能造出"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，实际上是团队资产：版本要统一、许可证要管、脚本要能共享。

>

> 应用软件：字处理、浏览器、客户端。浏览器别小看，不同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。

>

> 被测试系统是核心：你要测的软件本身，还有它的版本。这里最容易出的是版本混乱：测的是哪个构建、对应哪个分支、包了哪些还没合并的改动，必须查得清。给一条硬要求：每次部署都要能追到一个确定的版本号或提交。做不到，复现缺陷就是空话。

>

> 六项合起来是一个软件矩阵，组合数量会炸开。这就是下一节要用虚拟化和容器把这些组合固化下来的原因。

提问：测试环境里的数据库从生产同版本降了一个小版本，有风险吗？

> **预设回答：**

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

- 答"有风险，性能结论可能不成立"——对，特别是索引、锁、事务隔离这些场景。

- 答"看测什么，功能影响小，性能和兼容性影响大"——最准确，回到"你要下什么结论"这个判据。

- 收口：版本差异要不要管，看你要验证的结论对版本有多敏感。

> **易错点 / 考点：** ① 六项要能默写，最容易漏掉的是被测试系统这一项；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；③ 口诀：装什么、装哪个版本、怎么配，三层都得对。

> **过渡：** 六项里测试工具那一项，课件单独用一张图展开，看看一条完整的工具链长什么样。

【12.12】持续集成工具集（2 分钟）

从提交代码到看见测试报告，中间要人点几下?

>

> 图上没有正文，能看出的是一条链：测试工具不是孤立的一个软件，是链上的一串环节。

>

> 起点是代码仓库。所有环节的输入都在它那儿。没有版本控制，后面全是手工。

>

> 第二站是构建。源码编译、打成制品，jar 也好、镜像也好。这一步的输出要可重复：同一个提交，任何时候构建出来的东西都要一样。

>

> 第三站是自动化测试。单元测试、接口测试、回归测试都在这一站跑。位置要看清楚：它在打包和部署之前。测试挂了就不往下走，这就是质量门。

>

> 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，都查得到。

>

> 第五站是部署与通知。把制品推到测试环境，再把结果告诉人：成功还是失败、找谁看、日志在哪。

>

> 这张图顺手把责任也分清了：谁维护仓库，谁盯构建，谁跑测试，谁管制品。

>

> 还有个隐含要求：每一站都要能被机器自动触发。哪一站要人手工点一下，整条链的节奏就被这一下卡住。

>

> 所以判断持续集成的成熟度，有个简单办法：问从提交到出报告，中间人工操作几次。零次是真流水线，三五次是半自动，十次以上还停在手工时代。这个数字比任何成熟度模型都好用，因为它能当面数。这也接得上 12.4：自动部署不是买一个工具，是把每一站接上，让它们自己往下传。

> **提问：**构建自动了、测试自动了，只有部署要人手工执行，这条链算不算持续集成？

> **预设回答：**

- 答"算，集成就是构建加测试"——严格说不错，持续集成以频繁集成加验证为核心。只到这一步，价值只发挥一半。

- 答"不算完整，缺持续交付那一段"——更到位。链断在部署，反馈周期被人工环节拉长。

- 答"手工部署更安全"——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 环节顺序能说清，考点常在"自动化测试放在构建之后、部署之前"；② 判断"持续集成就是持续部署"——层次不同，集成是验证，部署是交付；③ 那句可量化的标准要记：人工操作几次。

> **过渡：**工具链解决怎么跑，跑起来还得有东西可处理，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）

> 生产库导一份到测试环境，行不行？

>

> 课件列四类数据，对应四个测试目的。

>

> 原有数据，指系统里已经存在的那些：历史订单、老用户、上一版的配置。为什么单列？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，一读就错。所以升级类测试的第一件事，是灌一份真实的老数据进去。

>

> 正确数据和错误数据是一对。只准备正确数据，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。课件把它俩并列摆着，说的就是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和杂。这里有一条红线：真实客户数据里有个人信息、用户名、联系方式、交易记录，不能直接拿来测。这就是脱敏。姓名、身份证、手机号、银行卡、地址替换掉，或者做不可逆处理，保留分布特征和结构，去掉能指到人的部分。脱敏不是讲究，是合规底线。

>

> 大量的数据讲规模。生产一千万条，测试一百条，索引命不命中、深翻页慢不慢、批量任务要跑多久，结论完全不同。重要的性能和批处理测试，必须准备同量级的数据。这也是准生产四对齐里的一条。

>

> 再补三件工程上必须有的事：造数，用脚本批量生成，别手工录；数据版本，这批数据是哪个版本、配哪套用例，要查得到；回滚，数据被污染了能一键恢复基线。三件事不做，测试数据用两周就烂，行内叫数据污染。

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"——漏了两件事。一是个人信息，测试环境权限松、日志全，数据流出去责任在公司；二是数据量太大时，导入本身就能把环境拖垮。

- 答"先脱敏再导，最好只导需要的表和字段"——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"，是这份数据能不能追溯到某个具体的人。

> **易错点 / 考点：** ① 四类数据列全并说清用途，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 案例题问"测试数据怎么管理"，答四步：定义基线数据、脚本化造数、敏感字段脱敏、版本化加可回滚。

> **【思政落点 2】**：脱敏这件事，省一步就是拿别人的隐私换自己的方便。手机号、身份证、住址、交易记录一旦进了测试库，往往没人审计、没管权限，一份表格转发出去就收不回来。它考的不是技术，是肯不肯为别人的隐私多花一道工序。能不用的真实数据就不用，必须用的先脱敏，用完能删就删，这才配碰用户的数据。

> **过渡：**真实数据不好拿也不好用，下一页给另一条思路。

【12.14】流量回放技术（3 分钟）

> 数据不好拿，还有一条路：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。流量回放反着来，它记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。这条路不新，做迁移和重构的团队用了很多年。

>

> 工作方式分三段。录制，在入口处抓请求，网关、代理、负载均衡都行，通常走影子方式，不影响正常业务。回放，把录下的请求按原来的节奏打到待测环境，可以是新版本，也可以是迁过架构的新系统。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，得人工确认。

>

> 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，给回归测试补真实

> 用例。共同点是，你怕的不是功能写错了，是改完之后老行为悄悄变了。

>

> 但有三个前提。脱敏，录下来的请求里有用户名、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，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。

>

> 收一句：静态数据看结构，回放流量看行为。前者告诉你库里有什么，后者告诉你用户怎么用。

> **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，不是最危险。比对不准还能靠人工复核补救。

- 答"回放把真实业务写了，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"——常见，属于容量问题，限速、抽样能解决。

- 收口：回放的第一条纪律，写操作一律指向影子库，外部依赖一律 Mock。

> **易错点 / 考点：**① 三段式录制、回放、比对，加三个前提，要能说清；② 判断"流量回放可以完全替代人工设计用例"——错，它只覆盖发生过的行为，盖不住新需求、新分支、新异常；③ 和数据资源连着答：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。

【12.15】12.3 虚拟化技术的应用（1 分钟）

> 五个小节，一层比一层抽象。

>

> 12.3.1 虚拟机技术，最基础的虚拟化：一台物理机上跑多个完整操作系统。要讲清两种类型、为什么用它、它怎么工作、有哪些软件。虚拟机是这一切的起点，不讲清它，后面全是名词。

>

> 12.3.2 QEMU-KVM，把 Linux 上最主流的开源方案拆开看：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，换个思路：不虚拟硬件，只隔离进程。要讲清 Cgroups 和 namespace 这两根支柱，还要讲清容器为什么比虚拟机快。容器不是小号虚拟机，这个误解每年都有人踩。

>

> 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，以及配套工具。这一节最容易被跳过，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。外部依赖最不讲道理：收费、限流、在你最需要的时候挂掉，三样占一样就够受。

>

> 五节的逻辑是层层抽象：虚拟硬件，到虚拟操作系统环境，到虚拟整个集群，最后到虚拟外部服务。抽象的层级越高，越贴近测试人员的日常。五节不是并列的五种工具，是从底往上搭的台阶。

>

> 因为你真正要的从来不是一台机器。你要的是一个可重复、可丢弃、可并行的测试环境。虚拟机给的是机器，容器给的是环境，K8s 给的是规模，服务虚拟化给的是可控的对手。四样凑齐，一个人也能拉起一整套能反复推翻重来的测试环境。

提问：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，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。有人需要 root、需要特定内核模块时，容器就不够了。

- 答"看需求，先看要不要独占、要不要改系统"——最成熟。要改内核或独占硬件就上虚拟机，只要隔离运行环境就用容器。

- 收口：选型的分界线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 分清虚拟化技术（虚拟机、容器）与服务虚拟化（Mock 这类工具），后者虚拟的不是机器；③ 简答常问"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：** 从最底下那层开始，先看虚拟机有哪两种类型。

【12.16】12.3.1 虚拟机技术（3 分钟）

> 笔记本上装个虚拟机跑 Linux，这算哪一类？

>

> 虚拟机按虚拟化软件装在哪分两类。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。性能损耗小、安全性高，代价是独占整台机器，你在已经装着 Windows 的笔记本上装不了它。

>

> 第二类装在操作系统之上，当普通软件装，业内叫寄居型。装了就能用，和日常系统共存，代价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。

>

> 所以你那台笔记本上是寄居型；公司服务器上虚拟出二十台测试机，通常是裸金属型。考题问两种类型，答的就是这个区分标准：虚拟化层和硬件之间，有没有宿主操作系统。

>

> 第二个要点是构建虚拟的测试农场。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统一种、统一收。你把一批虚拟机整整齐齐管起来，谁要用分配一台，用完销毁，下次重建。关键不是多，是标准：每台都从同一个模板克隆，系统版本、依赖库、配置完全一致。

>

> 它治的是一个折磨测试团队多年的毛病，环境漂移。一台测试机用了三个月，有人装了新版本的库，有人改了 hosts，有人手工调了参数，它和生产的差距一点点拉开。到最后没人知道它是什么状态，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，从模板重拉一台，五分钟回到标准状态。把服务器当宠物养，还是当牛羊养，是两种完全不同的运维思路，测试农场选的是后者。

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

> **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，服务器要把算力尽量给业务，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、有没有宿主系统、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有没有宿主操作系统；② 高频错答：把虚拟机说成模拟器——模拟器在软件层模拟另一套指令集，虚拟机是切分硬件资源；③ 测试农场与环境漂移这两个术语要成对背下，一个是解法，一个是问题。

> **过渡：**既然要用虚拟机，就得算清它到底省了什么，下一页是四条带数字的理由。

【12.17】为什么使用虚拟机？（3 分钟）

> 一台服务器，长期只用到两三成。

>

> 这一节回答为什么用虚拟机，课件给四条，都带数字。先说一句：下面这些数字是课件的行业经验值，用来说明量级，别当成精确测量结果。

>

> 第一条，把硬件资源用足。课件说，有 70% 的服务器利用率只有 20% 到 30%，借助虚拟机技术可以提到

> 85% 到 95%。为什么会空？一台服务器通常只跑一个应用，平均负载远低于峰值。你得按峰值配机器，于是大部分时间在浪费。虚拟化把多台低负载机器合到一台物理机上，空转的算力被填起来。这两个数字说的是利用率，不是性能翻了 3 倍。

>

> 第二条，节约能源和空间。内存加到 16G 或更高，一台机器可以虚拟 4 到 8 台服务器。少 4 到 8 台物理机，就是少 4 到 8 份电费、少 4 到 8 份机架空间，风扇噪音和空调负荷一起减。

>

> 第三条，提升运作效率。几分钟就能装载所需的系统镜像。这在测试场景里价值极高：新同事要环境，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，把环境准备从一天的活变成一次点击。

>

> 第四条，有利于环境的建立和维护。添加、移动、变更、重置，四个动作对应测试团队的四种日常：加一套并行环境，把环境从这台宿主机挪到那台，调整配置或扩容，还有最常用的重置。重置是最被低估的能力：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。

>

> 四条收成一句：你要的不是更多机器，是更灵活的资源。

提问：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。

- 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"——技术上不错，业务系统很难长期跑满，测试环境本来也不追求跑满，追求够用且能并行。

- 收口：先分清瓶颈是算力不够，还是算力闲置，两种情况的解法正好相反。

易错点 / 考点：① 20% 到 30%、85% 到 95%、4 到 8 台、16G 这几个数字要记准，简答常问"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，一个是资源使用效率，一个是处理速度；③ 四个动作记牢：添加、移动、变更、重置，重置最容易被忘。

> **过渡：**知道它省了什么，接着看它凭什么做到，下一页进原理。

【12.18】虚拟机工作原理（2 分钟）

虚拟机跑起来，里面的系统以为自己独占一台机器。

>

> 图上三层，一层层看。三层各管一件事：硬件给真资源，Hypervisor 负责切和骗，客户系统只管用。最下面一层是物理硬件：CPU、内存、磁盘、网卡，真的，只有一份。中间一层是虚拟化层，也就是 Hypervisor，虚拟机监控器。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。最上面是多个客户操作系统，每个都以为自己独占一台完整电脑，装自己的驱动、跑自己的应用、互相看不见。

>

> 先说清一件事：它不是模拟一台电脑的软件。模拟器在软件层模拟另一套指令集，指令一条条翻译，慢。虚拟机是把真实资源切开分给几个系统，一台八核的机器切成两台四核，用的是真核。

>

> 它怎么骗过上层？靠特权指令的拦截。操作系统里有些指令只能在最高特权级执行，客户系统执行到这类指令时，CPU 把它交给 Hypervisor，由 Hypervisor 代为完成，或者模拟一个结果，再把控制权还回去。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虚拟化支持，把拦截特权指令做成硬件功能，损耗大幅下降。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。

>

> 带走一句判断：虚拟化是资源的时分复用，加状态的隔离。时分复用让多个系统共享一份硬件，隔离让它们互不干扰。

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，客户系统常装半虚拟化驱动，主动配合Hypervisor，这时它知道。

- 答"虚拟机就是把真机的系统拷一份"——常见误解。拷过去的是文件，跑起来的是被切分出来的硬件资源。

- 收口：两种回答都对，取决于全虚拟化还是半虚拟化，下一页讲这个区分。

易错点 / 考点：① 三层结构硬件、Hypervisor、客户系统要能画出来；② 必背术语：Hypervisor、客户操作系统、特权指令拦截；③ 判断"虚拟机会让物理机变慢所以不能多用"——损耗存在，但硬件虚拟化已经把它压得很低，换来的是利用率和灵活性。

过渡：原理还有下一半，虚拟化在实现上的两条路线。

【12.19】虚拟机工作原理 – 续（2 分钟）

同样是虚拟机，有的快有的慢，差别常在一条线：改不改客户系统。

>

选型时先问一句：现成镜像能不能直接用？能直接用的，走全虚拟化。

>

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 靠二进制翻译或者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：整体走全虚拟化，I/O 上装半虚拟化驱动，网卡和磁盘这两块最耗性能，单独优化掉。

> virtio 就是这条路线的产物。

>

> 还有一个问题要回答：虚拟机虚拟了哪几样资源？内存，各自有连续地址空间，映射到物理内存的不同区域，这一层最容易被忽略，客户系统看到的是连续地址，底下是宿主机上的碎片页；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，实际是虚拟交换机的一个端口。

>

> 存储这一项顺手带来一个对测试极有用的能力：快照。环境装好、数据灌好，拍一张，每次测试前还原一次。装一套数据库加测试数据要半天，还原只要几分钟，中间被人改乱了也不怕。这就是可重复最省事的实现。可重复一旦做到，缺陷复现就不靠运气。

>

> 收一句：全虚拟化要兼容，半虚拟化要性能，快照要先学会用。

> **提问：**想十分钟内把一套装好数据库和测试数据的 Linux 环境复制三份给三个同事，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，有安装脚本"——脚本化可行，但装完还要灌数据、调参数，十分钟不够，三次之间也可能不一致。

- 答"让他们连同一台机器，分不同用户"——省机器，却踩进共享环境冲突：数据互相污染，改配置互相影响。

- 收口：判断标准还是那两个词，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说清：是否修改客户操作系统；② 判断"半虚拟化性能更好所以优先选"——看场景，兼容性有要求时全虚拟化更合适；③ 快照的用途要和环境可重复、快速回滚一起答。

> **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。

【12.20】常见的虚拟机软件（3 分钟）

> 九种虚拟机软件，名字摆一起很乱。先归类。归完类你会发现，学校机房用的和云厂商用的根本不是同一批

东西。

>

> 第一类是服务器侧的开源主力。KVM，也就是 Kernel-based Virtual Machine，Linux 内核自带，把内核直接当虚拟化层用。QEMU 是开源的模拟器，各种硬件都能模拟。Xen 是老牌开源方案，早期云计算大量用过。这三个能上规模、可控，配置门槛比桌面软件高。

>

> 第二类是桌面侧的商业软件。VMware 的 Workstation 和 Fusion，同一家按平台分两个名字，前者在 Windows 和 Linux 上，后者在 macOS 上。Parallels 也在 Mac 上，跑 Windows 体验好。Hyper-V Manager 是微软自家的，服务器版和专业版里就有。

>

> 第三类是轻量或特殊用途的。VirtualBox 完全免费开源，跨平台、上手快，教学和个人测试用得最多。Virtuozzo 走容器化路线，偏主机服务商场景。Colinux 思路特别：让 Linux 当一个进程跑在 Windows 里，不走完整虚拟机那条路，更接近进程级方案。

>

> 怎么选，三步。看平台：宿主是什么就用什么。看用途：桌面学习、单机测试，桌面那几款够用；服务器上批量跑、接集群管理，就上开源那一系。看约束：许可证预算、和现有云平台对接、团队会不会维护。第三步最常被跳过，装得上不等于维护得住。

>

> 还有一条实战经验：几个虚拟化软件在同一台机器上同时启用会打架，因为都要占硬件虚拟化能力。这不是软件缺陷，是资源独占。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。软件选错不致命，维护不住才致命。

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，资料也多。

- 答"VMware Workstation，性能好、功能全"——可以，先看有没有授权，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，切换成本高，不能和宿主系统同时用，做测试反而低效。

- 收口：学生场景先看成本和易用，生产场景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，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——它是模拟器兼虚拟化软件，和 KVM 配合才构成完整方案；③ 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、也最值得讲清的是 QEMU 加 KVM 这一组。

第 12 章讲稿（12.21–12.40，v3）

> 锚点与课件一一对应，不增删页。取消板书。口播按四拍：起 → 推 → 收。

【12.21】12.3.2 QEMU-KVM 虚拟机解决方案（3 分钟）

> 装虚拟机，很多人以为是装一个软件。其实装进去的是两个东西，各干一半的活。

>

> 先说 QEMU。它的原话分两段。QEMU 通过动态二进制转换，几乎可以模拟任何 IO 设备。再把客户系统的指令转译给宿主机硬件执行。两个关键词，IO 设备和指令转译。它能把网卡、磁盘控制器、显卡一个一个假装出来，让客户系统以为自己插着真硬件。它还把客户系统的指令一条条翻成宿主机认的指令。所以单靠 QEMU 就能跑虚拟机，兼容性极好。代价是慢，每条指令都要软件翻一遍。

>

> 再说 KVM。它不是独立软件，是 Linux 内核提供的虚拟化架构。加载之后内核本身就充当 Hypervisor。为什么它快？因为不需要多出一层，虚拟机就是内核直接管的进程。

>

> 可 kvm.ko 只做了 CPU 和内存的虚拟化，IO 设备的虚拟化它没有。所以这一句才是关键：KVM 必须结合 QEMU，才构成一个完整的虚拟化技术。

>

> 分工就清楚了。KVM 管 CPU 和内存，快；QEMU 管设备和指令翻译，全。缺了谁都不完整。只有 KVM，没有设备模型，虚拟机起不来。只有 QEMU，全走软件翻译，性能上不去。

>

> 打个比方，KVM 像发动机，QEMU 像车身和仪表盘。发动机给动力，车身和设备让你真能上路。这两件事本来是矛盾的：接近原生的性能，加上极强的设备兼容。分工把它们各自放到了合适的位置。

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。

- 答"因为 QEMU 能模拟各种硬件，兼容性好"——这是 QEMU 的价值，真正的原因是缺了它方案不完整。

- 答"其实可以只用 KVM"——错答，把内核模块当成了完整方案，这是本节最典型的一处混淆。

- 收口：要性能找 KVM，要设备找 QEMU，成套才叫方案。

易错点 / 考点：分工必须一句话说清：KVM 管 CPU 与内存、QEMU 管 IO 与指令翻译。判断题"KVM 是独立软件"——错，它是 Linux 内核模块。还要能背出"必须结合"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。

【12.22】12.3.3 容器技术与 Docker（3 分钟）

进容器里敲一句 ps，只跳出三四个进程。外面那台上百个。差别在哪?

>

> 容器和虚拟机有共同点，都可以看成虚拟实体，都满足隔离性和可管理性。分界在底座。容器的核心技术是 Cgroups 与 namespace。namespace 做资源隔离，Cgroups 做资源控制。两根支柱，考试常考。

>

> 先说 namespace。它管你看不见的东西。进程列表是自己的，外面的进程看不到。网络是自己的，有独立 IP 和端口。挂载点、主机名、用户空间也各自独立。所以 ps 只看得见自己那几个，感觉像一台独立机器。

> 这就是隔离。

>

> 再说 Cgroups。它管你用多少。最多几个核的 CPU、最多吃多少内存、磁盘读写限多少、能开多少进程。超了就被约束，内存超了直接被杀。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。这两句要能分开说，别混成一句。

>

> 那和虚拟机的分界在哪？虚拟机虚拟的是硬件，每个客户系统带自己完整的内核。容器里的所有进程共用宿主机那一个内核。它们只是被 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的东西。这就是它的边界。

>

> 共享内核带来启动快、体积小、开销低。也带来隔离不如虚拟机：一个容器里的内核级漏洞，可能影响整台宿主机。选型时问自己一句，我要隔离到内核，还是隔离到进程。前者用虚拟机，后者用容器。

> **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，部分网络参数可以在内核允许的范围内调，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：**namespace 与 Cgroups 的分工要分清：一个做隔离，一个做资源控制。高频错答是把容器叫成"轻量级虚拟机"，错在"虚拟机"三个字，它是进程级隔离。判断题"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：**概念清楚了，动手怎么做出来一个容器？从构建流程走。

【12.23】基于 Docker 构建容器（3 分钟）

> 换台电脑跑你的测试，报错一屏。装依赖装到半夜，还是跑不起来。

>

> 三个概念先立住，这是 Docker 的全部底子。镜像，一个只读模板，里面装着文件系统、依赖库、环境变量、启动命令。像安装光盘，本身不能运行。容器，镜像跑起来的实例，在镜像之上加一层可写的薄层。仓库，存镜像的地方，公有私有都算。关系一句话：从仓库拉镜像，用镜像起容器。

>

> 构建走四步。第一步写 Dockerfile，一个文本文件，一行一条指令。要写的是：基于哪个基础镜像、拷哪些文件、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。

>

> 有个设计特别值得讲：分层。每条指令生成一层，层是可复用的缓存。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，缓存机制决定的。

>

> 细节不用背。参考资料就两条，Docker 官方文档，还有腾讯云社区的一篇入门文章。

>

> 对测试的意义在哪？你的测试环境可以写成几十行文本。以前交接环境靠说明书加口口相传，现在是把这个 Dockerfile 拿去 build。环境不再是一台具体机器，而是一份能被 review、被版本管理、被 diff 的代码。这就是后面基础设施即代码的雏形。

> **提问：**为什么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，构建一慢就不知道该改哪。

- 答"顺序无所谓，反正最后结果一样"——结果一样，代价差很多。改一行代码就重装全部依赖，构建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：**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"。另一处混淆是把 Dockerfile 当成脚本，它是声明式的构建描述，逐层执行并缓存。分层缓存对构建速度的影响，是实操题常见的追问。

> **过渡：**容器做出来了，它到底比虚拟机好在哪？

【12.24】容器的优点（3 分钟）

同一台机器，装虚拟机跑几个，装容器能跑几十个。差的不只是数量。

>

> 第一条，镜像体积小，没有内核，属于轻量级应用。为什么小？镜像里只有应用和它的依赖，不带操作系统内核，内核用宿主机那一个。好处很直接：传输快、存得多、拉取快。CI 上拉一个几十兆的测试镜像几秒钟，拉一个几 G 的系统镜像要几分钟。

>

> 第二条，创建和启动更快。启动一台虚拟机要几分钟，容器的创建和启动只要几秒钟。原因还是那个：虚拟机要模拟硬件、引导操作系统、拉起一堆系统服务。容器只是启动一个被隔离的进程，没有引导过程。对测试的意义很大，环境可以用完就扔。跑完一批用例，容器销毁；下一批，重新起一个干净的。这才叫隔离，不是清理过，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。四个"更"拆开说。层次更高，抽象在操作系统之上，不用管硬件。额外开销小，没有一整套客户系统的 CPU 和内存开销。粒度小，Cgroups 能按容器限 CPU 和内存，每个容器精确配额。密度大、省磁盘，一台机器跑几十个容器，虚拟机可能只跑得下几个。

>

> 优点背后有代价。隔离弱，共享内核。状态难持久，容器一销毁里面的数据就没了，数据库数据和测试产物必须挂到外面。调试方式不同，得习惯进容器里看，不是远程登录一台机器。

>

> 收一句：容器把环境变成按秒创建、按秒销毁的东西。测试最核心的两个需求，隔离和可重复，正好对上。

提问：容器一销毁数据就没了，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；代价是慢，脚本本身还要维护。

- 答"那就别删容器"——这就把容器最大的优势扔了，不删容器，环境又会漂移回越用越乱的老路。

- 收口：无状态的东西用容器，有状态的东西挂出去。

易错点 / 考点：三条优点要说全，并且能给出原因：体积小因为无内核，启动快因为没有引导。判断题"容器启动快是因为它比虚拟机优化得好"——错，本质原因是没有操作系统引导过程。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？

【12.25】12.3.4 集群管理与 Kubernetes（3 分钟）

容器一多，问题就换了。不再是它怎么起，是它该起在哪台机器上。

>

> Kubernetes 是目前最具影响力的容器集群管理工具之一。它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。这些功能让大规模容器集群管起来省事很多。四个功能，逐个看。

>

> 部署运行。你告诉它我要三个副本，它负责把容器跑起来，并且持续保证有三个。哪个挂了它自动补一个，

> 哪个节点宕了它把上面的容器迁走。这个特性叫期望状态：你描述要什么，它负责做到并保持。这是声明式，不是命令式。

>

> 资源调度。集群里十台机器，新来的容器放哪台？它看每台还剩多少 CPU 和内存，有没有特殊要求，比如必须有 GPU、必须和某个服务同机。然后选一个位置。

>

> 服务发现。容器不停创建销毁，IP 一直在变，前端怎么找到后端？给每个服务一个稳定的名字和虚拟 IP，请求打到名字上，它转发到当前活着的容器。没有这一条，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。

>

> 动态伸缩。负载高了自动加副本，低了自动减。这一条对性能测试有用：你能观察系统在自动扩容这个机制下扛压的表现。

>

> 对测试的落点：一套环境用命名空间隔开，跑完整个命名空间删掉，下次重建。这是可重复的最终形态，不是把环境清干净，是环境根本没存在过第二次。

>

> 也得说成本。学习曲线陡，运维复杂。团队两三台机器、几个服务，Docker Compose 就够。上 K8s 是用高射炮打蚊子。

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，这是服务发现的核心机制。

- 答"把 IP 写进配置文件"——容器化之后最常见的翻车点。容器一重建，IP 变了，配置全废。

- 答"用固定的负载均衡地址"——思路对，K8s 里的 Service 就是干这个的，本质是一个稳定的虚拟入口。

- 收口：容器化之后的一条纪律，不要依赖 IP，要依赖名字。

易错点 / 考点：四个功能要能说全：部署运行、资源调度、服务发现、动态伸缩。高频错答是把 K8s 当成容

器运行时，它是集群管理与编排工具，实际跑容器的是容器运行时。声明式、期望状态这个提法要说清，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，是 Pod。

【12.26】集群管理与 Kubernetes（2 分钟）

一个 Pod 里塞两个容器，第二个起不来，报端口被占用。为什么？

>

> 定义先给准：Pod 是 K8s 中的最小调度单元，一个 Pod 封装 n 个容器。更细的说明在 Kubernetes 官方文档的 concepts 目录里。

>

> 第一层，最小调度单元。调度的时候，对象是 Pod，不是容器。也就是说，一个 Pod 里的所有容器一定在同一台机器上。这一点决定了用法：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。多个的场景有个名字，边车模式：主容器跑业务，边车容器干辅助的活，收日志、做代理、同步配置。

>

> 同一个 Pod 里的容器共享什么？三样。共享网络命名空间，用的是同一个 IP 和端口空间。所以容器之间用 localhost 就能互访，但端口不能冲突。共享存储卷，挂在 Pod 上的卷，里面每个容器都看得见。共享生命周期，Pod 被删掉，里面的容器全都没了。

>

> 到底为什么要多这一层，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。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 解决的就是"必须同机的一组进程"这个问题。

>

> 对测试的落点：你要测一个微服务系统，被测对象是 Pod，不是单个容器。日志散在多个容器里、进程之间

> 走 localhost、Pod 重建后 IP 会变。这三件事直接影响自动化脚本怎么写。

> **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：看它们共享哪一层，共享网络就要按同机规划端口。

> **易错点 / 考点：**定义要能准确复述：最小调度单元，一个 Pod 封装 n 个容器。只答"一组必须同机部署的容器"会丢分，还要补上共享网络、存储和生命周期。同 Pod 容器用 localhost 互通、端口不能重复，这是面试和考试的高频细节。

> **过渡：**从一个 Pod 到整个集群，再看一眼真实云上那套基础设施长什么样。

【12.27】基础设施架构（3 分钟）

> 打开云控制台，服务列表能拉一屏，全是缩写。背名字没用，得知道它们各在哪一层。

>

> 网络层四个。VPC，虚拟私有云，等于你在云上一块隔离的网络。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。

>

> 计算层三个。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。

>

> 数据层两个。RDS，关系型数据库服务。S3，简单存储服务，存文件和静态资源。这一层是数据放哪。

>

> 安全层两个。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。

>

> 运维与消息层三个。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。

>

> 这张图的用处，是把要素落到服务上。硬件对应 EC2，网络对应 VPC、ALB、Route53。软件对应 ECS、ECR、RDS，数据对应 S3、RDS。你要记的是这个对应关系，不是亚马逊的产品线。

>

> 还有个现实：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，一定会配套讲用完即毁和成本告警。这是同一件事的两面。

> **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"——最贵的一句"反正"，账单出来才想起它。

- 收口：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：**十四个缩写的归类比死背名字重要。给一组服务让分层，按网络、计算、数据、安全、运维消息来分。高频错答是把 EC2 与 ECS 混为一谈，一个是虚拟机，一个是容器服务。按量计费这个约束直接推出两条工程纪律：环境用完即毁，成本要能被看见。

> **【思政落点 3】**：云上按量计费，忘关一台机器，月底就多出一笔钱。这里讲的不是省钱小技巧，是公共资源有主、用者有责这条常识。测试环境最容易变成没人认领的东西：谁建的记不清，谁敢删不敢动。把用完即毁和成本告警写进流程，等于把公司的钱当成自己的钱看待。能省的地方省下来，才有资源做要紧的事。

> **过渡：**云上的服务都是自己的。可测试要调的那些服务，往往不是自己的。

【12.28】12.3.5 服务虚拟化及其工具（3 分钟）

用例一半失败，查到最后，是对面那个接口在抖。你没法让它稳。

>

先把概念分清。服务虚拟化和虚拟机、容器不是一回事。虚拟机虚拟硬件，容器虚拟操作系统环境，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，只管一件事：你的系统去调外部服务时，让那个调用得到你要的结果。

>

为什么需要它？四类困境。依赖不可用，第三方还在开发，接口文档有、服务没有；或者它只在办公网能访问，测试环境连不过去。依赖不稳定，时快时慢、时不时超时，自动化一半失败是它的锅，你还没法让它稳定。依赖不可控，你想测对方返回 500 怎么办、30 秒才响应怎么办、返回超长字符串怎么办。真实服务不会配合你演。依赖有成本或有风险，按次收费的接口，短信、地图、支付，不能拿它压测。涉及真实资金和真实用户的接口，更不能随便打。

>

这四类，服务虚拟化一次全解。它给你一个假的服务，行为由你定义。想成功就成功，想超时就超时，想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。它们轻量、能跑在 CI 里、社区活跃。商业的 Parasoft Virtualize、Tricentis Tosca 强在图形化配置和协议覆盖。和企业既有测试平台也集成得好，代价是许可证成本。

>

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，开源够用。要覆盖大型机、专有协议、复杂报文，商业工具有优势。

提问：支付接口不能拿来压测，可领导要求验证下单高峰扛不扛得住，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，这正是 Mock 和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"——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：**四个工具名及归属必背：WireMock、Hoverfly 开源；Parasoft Virtualize、Tricentis Tosca 商业。高频混淆是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。案例题常这样问：外部依赖不可用导致测试没法做，怎么解决。标准答法是服务虚拟化，并说明可模拟延迟、故障、限流。

> **过渡：**工具名单里 Hoverfly 单独给了一节，看它能干哪五件事。

【12.29】开源服务虚拟化工具 Hoverfly（3 分钟）

> 服务还没写完，你的用例已经要开工了。这时候拿什么测？

>

> Hoverfly 能干五件事，每件都对着一个具体的痛。

>

> 一，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。为什么强调 CI？CI 最怕不确定。外部服务抖一下，流水线红一次，团队就开始不信流水线。换掉之后，流水线只反映自己代码的问题。

>

> 二，可以模拟网络延迟、随机故障或速率限制，用来测边缘情况。你想验证支付接口 3 秒没响应时我们这边会不会一直等，真实服务不会配合你。Hoverfly 可以让它每次都慢 3 秒。异常路径靠真服务测不出来，只能靠假服务。

>

> 三，可以导入导出、共享和编辑模拟数据。能导出，它就能进版本库、能 review、能共享给同事。能编辑，产品改了接口你改配置就行，不用改代码。

>

> 四，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。图形界面工具在 CI 里是

> 用不起来的。

>

> 五，提供多种运行模式，对 HTTP 响应记录、回放、修改或合成。四个模式分开记。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。回放 simulate，按存下来的数据返回，用于稳定的回归。修改 modify，在真实响应上做改写，注入一个错误码、改一个字段。合成 synthesize，完全由你凭空造，适合服务还不存在的时候。

>

> 四个模式对四个阶段：服务已经在跑用记录，稳定了用回放，要出异常用修改，服务还没有用合成。

>

> 最后一句工程细节：模拟数据要跟着接口一起版本管理。接口改了、数据没改，测试就假通过。这是服务虚拟化最隐蔽的一处。

> **提问：**服务还没开发出来，你又要先写测试用例，用哪种模式?

> **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，这正是要分开记四个模式的原因。

- 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：**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。高频错答是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。简答常问 Hoverfly 在 CI 里有什么用，答稳定依赖、可造异常、可版本化、可命令行集成四点。

> **过渡：**造环境的工具齐了，剩下的问题是让它们自己跑起来。

【12.30】12.4 测试基础设施的自动部署（1 分钟）

> 环境配好了，第二个人照着配一遍，配出来不一样。这一节治的就是这个。

>

> 五个小节，顺序先摆清。12.4.1 基础设施即代码，观念先行，把机器、网络、配置写成文件，进版本库。12.4.2 基础架构的自动部署。用 Terraform 把网络和服务器声明出来，再用 TestInfra 验证真建对了。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，再落到 Docker 和 Kubernetes。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 干这个。12.4.5 CI/CD 流水线，把前面串成一条，用 Jenkins 做调度。

>

> 这个顺序不是目录顺序，是建造顺序。先把资源用代码描述，再把资源建出来并验证，再把应用装进去，再把配置刷进去。最后让整件事自动触发。照这个顺序走，出了问题能定位到是哪一环。

>

> 为什么不能倒着来？资源没写成代码，自动部署建出来的是一堆只能手改的机器。建完不验证，建成功不等于建对。配置不刷，容器里还是空的。

>

> 目标收一句：从人配环境变成机器建环境，而且每次建出来的一模一样。五个小节里的每个工具，都为这一句服务。

提问：五个小节里，哪一步最容易被跳过，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。可跳过 IaC，流水线建的是一堆手工机器，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：五个小节的名称与顺序要能默写。12.4.2 里除了 Terraform 还有 TestInfra，很多同学只记前者。简答可这样组织：IaC 是观念，Terraform 建资源，TestInfra 验资源，Docker 和 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）

> 把安装步骤写成一份很详细的 Word，发群里，大家都照着做。这算不算把基础设施当代码？

>

> 先把定义摆上。Infrastructure as Code，中文叫基础设施即代码。它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。把基础设施以配置文件的方式纳入版本管理，实现更灵活更快捷的操作。

>

> 一，机器可定义文件。它强调的不是人能看懂，是机器能执行。你写一份配置，机器照着它能把环境建出来，这才叫 IaC。Word 写得再细，机器读不懂，还是靠人照着做，第 100 次和第 1 次就不一样。

>

> 二，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。描述的是一整套环境，不是一台机器。

>

> 三，纳入版本管理。这一半最容易被忽略，也最关键。配置进了 Git，就多了四件事：谁在什么时候改了什么，能审计。回滚到上一个正确版本，能恢复。对比测试环境和生产环境差在哪，能 diff。走代码评审，能把关。手工配环境，这四件一件都做不到。

>

> 两头对比。传统是环境配出来的，靠人手点、靠经验、靠口口相传的文档，结果每台机器都略有不同。出问题只能上机器看，机器坏了重建一次又是新状态。IaC 是环境编译出来的，一堆文本执行一遍就是一套环境。文本一样，环境就一样；坏了删掉重建，重建出来和原来一模一样。

>

> 有个说法很形象：环境是当宠物养，还是当牲畜养。当宠物要起名字、生病了要治、死了心疼；当牲畜编号管理，坏了就换。IaC 就是把环境从宠物变成牲畜的工程手段。

>

> 还有个难点要说清：难的不是写配置，是写对写全。漏一条防火墙规则，环境建出来就是不通的，而手工配的时候你随手就点了。所以 IaC 必须配套验证。

> **提问：**把安装步骤详细写进一份 Word 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：能不能让机器照着它把环境建出来，并且每次结果一样。

> **易错点 / 考点：**IaC 的英文全称与中文译名都要会写。高频错答是把 IaC 等同于写部署脚本，脚本可以是命令式的，IaC 更强调配置文件加版本管理加声明式。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？

【12.32】示例（2 分钟）

> 一份配置文件，在空环境和已经建好的环境上各执行一次，结果一样吗？

>

> 看图。这种文件由四块组成。

>

> 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，后面还要用。

>

> 关键区别在这：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，已有环境执行是只改差异。

>

> 这个词叫幂等。执行一次和十次，结果一样。对测试环境意义极大：你可以放心反复执行，不怕把环境搞坏。手工操作做不到，你点第二遍就可能建出两套。

>

> 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。

提问：同一份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。

- 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，现代工具会对比状态再决定动作，不会简单报错。

- 收口：判断是不是真 IaC，看它能不能重复执行而结果不变。

易错点 / 考点：声明式与命令式的区别要能举例说清。术语幂等必背，它是反复执行安全的前提。高频错答是把 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：样子看过了，再看工具得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）

> 有人把整台机器的镜像塞进 Git，说这也叫基础设施即代码。不算。

>

> 第一条特征，版本控制。课件讲得直白：这是把基础结构、配置、容器和管道持久化为代码最重要的部分。配置文件要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。

>

> 为什么不行？镜像是一个二进制大文件。你看不出它改了什么，没法 diff，评审里没法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、被人改。

>

> 第二条特征，模块化。最好的代码是可复用的，所以支持模块化的工具要有复用性，并且能实现模块的参数化。

>

> 举个例子。团队要建一套带数据库的测试环境，这个组合会反复用十次。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。参数化是模块化的关键，不能传参的模块，只是复制粘贴换了个形式。

>

> 第三条特征，可实例化或可部署。工具必须能把代码输入并部署到环境。这些代码可以是管道、配置的实例、容器，也可以是对云基础架构的改动。而且必须是纯脚本的、全自动的，不需要手工操作或配置。任何一步要人去点，这一环就串不进流水线。

>

> 三条连起来是一条判断链：是不是文本，能不能版本控制。能不能复用，模块化加参数化。能不能全自动跑，有没有手工步骤。

>

> 再补一条课件没写但实用的：还要看状态管理。工具得知道环境里已经有什么，否则算不出差异。

> **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，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，工程上丢掉了可维护性。

- 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：**三个特征的名称要能默写：版本控制、模块化、可实例化或可部署。为什么不能用二进制镜像，答不可读、不可 diff、不可评审。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，工具分四类，这是本节最该背下来的一张表。

【12.34】IaC 四类工具（3 分钟）

> 问个具体问题。Ansible 和 Terraform 都叫自动化，差在哪？

>

> 第一类，基础架构即代码工具，用于基础架构的自动部署。代表是 Terraform、CloudFormation。它管机器、网络、存储怎么建出来：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。从零搭，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。课件写的是只有 Docker、Kubernetes 等。看清"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，这四个常一起考。它管机器已经有了之后，里面的软件怎么装、配置怎么刷、服务怎么起。工作方式有差别：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，管持续集成和持续交付环境的自动化部署。代表是 Jenkins 2.0、Drone.io、ConcourseCI。它管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代码，而不是在界面上点来点去。

>

> 口诀：建资源、装容器、配应用、串流程。

>

> 这四类不是选一个，是按顺序叠起来用。先 Terraform 建机器和网络，再 Docker 打镜像、K8s 跑起来。再 Ansible 刷配置，最后 Jenkins 串成流水线。这也是后面四小节的顺序。

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"——最容易混的错答。混了之后设计流水线，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；职责上应该分开，资源生命周期归 IaC，配置归配置管理。

- 收口：谁负责有没有，谁负责里面是什么。

易错点 / 考点：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。高频错答是把 Ansible 归到基础架构即代码，课件明确把它放在配置即代码。四类叠加使用的顺序也要会，案例题常问怎么设计一条自动化环境搭建链路。

过渡：第一类最基础，看 Terraform 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）

环境建到一半失败，多数时候不是命令写错，是顺序错。

>

> 先看能力。Terraform 有完成完整的云基础架构创建的能力。它通过 DSL 以编程方式把各个组件链接在一

> 起。能把云基础设施里有用的部分定义成带参数化输入的模块，还能和其他模块集成，复用性好。三个关键词：DSL，领域特定语言，就是它那套专门描述基础设施的配置语法。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。参数化模块，前面讲的模块化落地。

>

> 再看结构，Terraform 分两个主要模块。

>

> 管理模块，定义 VPC、子网、NAT 网关、路由、安全组和 PuppetMaster。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，再加一台配置管理服务器。

>

> 服务器模块，在它的子网里定义多个消息代理和多个自定义服务器的层。这些层动态链接到公共 ELB，也就是负载均衡。

>

> 这个两层结构值得学，它体现一个通用原则：先网络后计算。为什么？服务器必须挂在子网上，子网必须在 VPC 里，路由和安全组决定服务器能不能被访问。顺序错了，机器建出来也连不通。

>

> 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。你写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，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。"最基本"三个字划出边界。它管资源，不管应用里的业务逻辑，也不管机器里的软件安装，那是 Ansible 那一类的事。

>

> 对测试的价值：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。

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：先把路修好，再跑车。

> **易错点 / 考点：**Terraform 两个模块的内容要能说出：管理模块管网络与基础管理，服务器模块是消息代理加服务器层再挂 ELB。高频错答是认为 IaC 会安装应用软件，课件明确它的范围是最基本的计算、存储和网络资源。DSL 这个词常和声明式一起出现在判断题里。

> **过渡：**资源建出来了，怎么知道它建对了？

【12.36】TestInfra（3 分钟）

> 返回码 0，构建是绿的。你上去跑用例，全红。

>

> 这就是 TestInfra 要处理的事。它的定位是一句话：一个功能强大的库，可以编写测试来验证基础设施的状态。它和 Molecule、Vagrant、Jenkins 都能集成。这样就能完成 DevOps 模式下的全自动化流水线式验证。

>

> 先把观念摆正。前面一直在讲怎么把环境建出来，可建出来不等于建对。IaC 执行成功，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，却会让测试全部失败。于是团队开始怀疑代码，查两天，最后发现是环境的问题。

>

> TestInfra 的办法是把环境应该是什么样写成断言。这台机器上 80 端口在监听、服务在运行、目录属主正确、配置文件里含某一行、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。

>

> 于是环境验证从一个靠人肉检查的动作，变成可重复执行的测试。它的定位要清楚：不测业务功能，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。

>

> 三个集成点也要说。Molecule 是开发 Ansible 角色时的测试框架。在角色开发过程中就能插进 TestInfra 断言，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。

>

> 一句话收：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。

> **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，也是少配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、进不了流水线，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：** 定位要一句话说清：验证基础设施状态的测试库。三个集成对象是 Molecule、Vagrant、Jenkins。判断题"TestInfra 用来测业务功能"——错，它测环境状态。案例题常问怎么保证环境搭建的正确性，标准答法是 IaC 加基础设施测试。

> **【思政落点 4】**：建完不验，等于交房不验收。环境建成功和建正确是两件事。多数事故不是不会建，是没人回头确认建成了什么样。工程上的老实就落在这道复查上：不假设一切正常，用断言去证实。这个习惯带出课堂也一样，做完的事自己先验一遍，不把问题留给下一个人。

> **过渡：** TestInfra 的写法有个特点，能参数化，还能和单元测试框架合用。

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

> 同一条断言要在三个环境上跑。你现在是怎么写的？

>

> 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 端口在监听这一条，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，Python 生态里最常用的是 pytest。TestInfra 本身就是 Python 库，天然能用上这些能力。断言写法、用例组织、参数化机制、失败报告、插件生态，都是现成的。你不用为了测环境另学一套工具。

>

> 两个特性合起来是什么意思？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就够：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。

>

> 能长期跑起来的少量断言，胜过写了不跑的一大堆。

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：**两个特性要能说出名字：脚本参数化、与 UT 框架集成。高频错答是以为 TestInfra 是独立的一套测试框架，它是库，依托现有单元测试框架。测试矩阵这个词常和环境一致性验证一起考。

> **过渡：**工具说了一路，最后给一份完整清单，建立全景。

【12.38】测试基础设施工具清单（3 分钟）

> 团队只有代码仓库和一台测试机，预算够补一样工具。补哪样？

>

> 八类，不用背全，要知道每类解决什么问题。

>

> 一，基础架构类，如 CloudFormation、OpenStack，管机器、网络、存储。二，容器类工具，如 Docker、Rocket、ElasticBox，把应用打成可移植的包。三，资源编排工具，如 Kubernetes、Apache Mesos、Swarm，让一群机器当一台调度。

>

> 四，微服务平台，如 OpenShift、Cloud Foundry、Mesosphere。它比编排高一层：不光管容器怎么跑，还管应用部署、服务治理和路由。

>

> 五，日志管理，如 Elastic Stack、Logentries、Splunk，解决出问题去哪看日志。Elastic Stack 四个组件是 ElasticSearch、Logstash、Kibana、Beats。六，系统监控、警告与分析，如 Prometheus、Icinga 2、Nagios Core、Zabbix、Cacti、Zookeeper，管健不健康、坏了通知谁。七，性能监控，如 AppDynamics、Datadog、DynaTrace、New Relic、CollectD、StatsD。

>

> 八，知识管理和沟通协作类工具，如 MediaWiki、Confluence、Zoom。这一类最不像基础设施，可环境怎么用、踩过什么、规程是什么，必须有人写下来。前七类的知识全在个别人脑子里，人一走就断档。

>

> 判断顺序：先补最缺的那一类。日志和监控最该先有，它们是排查的入口。然后 IaC 与容器，把环境变成可

> 重建的。最后才是平台类，别一上手就上平台，环境没标准化，平台只会把混乱放大。

> **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：**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。高频错答是把 Zookeeper 归到性能监控，课件把它列在系统监控、警告与分析这一类，按课件口径记。Elastic Stack 的四个组件也要能说全。

> **过渡：**工具选完，接下来把应用装进环境。

【12.39】12.4.3 应用程序容器化及集群部署（3 分钟）

> 装 Selenium 环境，最烦的一步不是下载，是让版本对上。

>

> 课件开头有个判断。测试环境里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。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。

>

> 例子要讲透。搭一个 Selenium 测试环境，不再需要下载 Selenium Server。也不用下载支持特定浏览器的 Web Driver。Selenium 官方提供对应的 Docker 镜像。装好 Docker 之后，执行 docker 命令拉取镜像，创建并启动容器就完了。

>

> 前后对比一下。以前装一套是这么干的。装 Java 环境，下载 Selenium Server。再为每个浏览器下载对应版本的 WebDriver。接着保证驱动版本和浏览器版本匹配，配好环境变量，启动服务。这一套做完半天没了，

> 换台机器还得重来。最容易翻车的是版本匹配那一步，版本不对，报的错莫名其妙。

>

> 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。

>

> 三个价值。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，驱动与浏览器的匹配由镜像保证，你不用再查兼容表。第三，环境隔离与并行，同时跑 Chrome 和 Firefox，起两个容器就行，互不干扰。跑十个并行任务，起十个容器。

>

> 一处要留意：容器里的浏览器默认没有界面。自动化通常用无头模式，可有些场景要截图对比、要真实渲染，就得配虚拟显示或远程调试端口。遇到容器里跑不起来浏览器，先查这一条。

>

> 收：容器化把装环境这个动作，从一门手艺变成两条命令。

> **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，只是次要收益，价值主要在一致性。

- 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：价值排序是一致性、隔离性、速度。

> **易错点 / 考点：**要能复述 Selenium 这个例子，并说清不再需要下载 Server 与 WebDriver 的原因。高频错答是以为容器化能提升测试执行性能，它提升的是环境一致性和部署效率。连测试工具也要容器化这个观点，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得张口就来。

【12.40】常用的 docker 命令（3 分钟）

> docker run 和 docker start，考过的人一半答反。

>

> 命令按生命周期排，好记。课件列了十条，其中 docker run 出现了两次，写的是同一个命令。

>

> 第一步，拿到镜像。docker pull 从镜像仓库拉取镜像，后面能带 TAG 或 DIGEST。能写 tag 就别用 latest，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 看本地已有的镜像。

>

> 第二步，做自己的镜像。docker build 从 Dockerfile 制作镜像。PATH 是构建上下文，写一个短横线表示从标准输入读。这一步的产物是镜像，不是容器。

>

> 第三步，跑起来。docker run 创建并运行容器。最常用，也最容易出问题：端口映射、目录挂载、环境变量、后台运行、容器名，全在参数里。给个心智模型：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 获取容器列表，默认只显示运行中的。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 启动一个已经存在的容器，docker stop 停止一个容器。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，因为它决定了容器里的数据是不是从零开始。

>

> 第六步，分发。docker push 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。

>

> 最后一条纪律：命令要写成脚本，不要靠手敲。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。

> **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，最常考的一组。

- 答"一样，都是让容器跑起来"——典型错答。结果相似，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"——不准确，run 默认用本地镜像，除非本地没有。

- 收口：区分标准是不是新容器，这直接决定容器里的数据是不是从零开始。

> **易错点 / 考点：**六组命令的作用要能对上，尤其 pull、build、run、push 的输入输出分别是镜像还是容器。判断题"docker build 生成容器"——错，生成的是镜像。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。

第 12 章讲稿（12.41–12.47，v3）

> 锚点与课件一一对应，不增删页。取消板书。口播按四拍：起 → 推 → 收。

【12.41】Kubernetes 集群环境中部署（2 分钟）

> 你在集群里敲一条命令，写上"三个副本"，回车，三份就跑起来了。谁在替你数这三个?

>

> 先把动作看清楚。交上去的不是一串命令，是一份描述。里面写：跑哪个镜像、要几个副本、开什么端口、给多少 CPU 和内存。这份描述一般写成 YAML，进版本库。部署这件事，也就跟着变成了一份文件。

>

> 文件交上去，集群把它翻成一层一层的对象。最上面是部署（Deployment）。它管的不是某一个容器，是"始终保持 N 个副本"这句承诺。往下是副本集（ReplicaSet），专管数量：少一个补一个，多一个删一个。

> 再往下是 Pod，容器待在里头。最外面是服务（Service），给这堆会变的 Pod 一个固定入口。

>

> 为什么非要拆这么多层？一层只管一件事。副本数变了，Pod 不用重写；Pod 换了，入口地址不用改。不拆，改个数量就得动全部配置。

>

> 这条链上有个很实用的能力，滚动更新。你换了镜像版本，它不是全停再全起。全停就断服务。它一批一批换：先起一个新的，等它健康，再停一个旧的，来回几轮。所以升级过程中一直有 Pod 在干活。

>

> 落到测试上，三件事。第一，每轮测试从一个干净的命名空间开始，跑完删掉，不留残留。第二，副本数能当旋钮：压测时调小，看资源紧张时的表现；验证扩容时调大，看调度和启动要多久。第三，日志散在各处。一个请求可能落到任意一个 Pod。排查得按服务聚合着看，别登某台机器翻文件。

>

> 一句话：K8s 把部署从一个动作，变成了一个一直维持着的状态。

> **提问：**做性能测试时，为什么要死盯"副本数"这个参数？

> **预设回答：**

- 答"副本少，压力全砸在一个 Pod 上，测出来当然难看"——对。负载怎么分，先由副本数定，它决定后端能扛多少。

- 答"拉到最大最保险"（典型错答）——拉满测的是集群容量，不是应用在目标规模下的表现，数字对应不上线。

- 答"压测打的是 Service，跟副本无关"——Service 后面就是这些 Pod，数量直接决定单个 Pod 挨多少。

- 收口：压测前先确认副本数和生产一致，否则测出来的东西对不上线。

> **易错点 / 考点：**① 对象链顺序要说得清：Deployment → ReplicaSet → Pod → Service；② 滚动更新考得多，答"一批一批换、升级不断服"；③ 案例题问"集群里日志分散怎么排查"，答按服务聚合、靠集中采集和检索，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。接下来看自动配置。

【12.42】12.4.4 应用程序的自动配置（3 分钟）

> "这台装好了，那台我记不清了。"机房五十台机器要装同一个补丁，最后总有人这么说。一台一台登进去，你会漏，也会装错版本。

>

> Ansible 干的就是这件事。课件给的定义是：它基于 Python 开发，是自动化运维的开源工具。提供远程系统安装、启动/停止、配置管理这些服务。还能对服务器集群做批量系统配置、批量部署、批量运行命令。

>

> 这句定义里有四个能力。远程系统安装：在一台控制机上发指令，几十台目标机同时装包。启动/停止：管服务的运行状态，要起的起，要停的停。配置管理：配置文件按模板推下去，保证每台内容一致。批量执行：一批机器同时跑同一条命令，巡检和排查时最省事。

>

> 三个词要划出来：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。前面讲环境不一致，根源就是一台一台手工配。

>

> 有个特点课件没写，但绕不过去：它一般不在目标机上装客户端。它靠 SSH 连过去执行。好处很直接。目标机是干净的，不用先装代理。一批新机器只要 SSH 通，当天就能管起来。对比前面第三类工具里的 Chef 和 Puppet，那两个一般要在目标机装 agent。它们走的是目标机定期来拉配置的路子。两种模式各有用处。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。

>

> 它组织任务的方式是清单加剧本。清单列清楚管哪些机器，可以分组成测试组、预发组。剧本写对这些机器做什么：装什么、配什么、起什么服务。剧本能反复跑，跑一次和跑十次，机器状态一样，这叫幂等。

>

> 落到测试上就一句：环境可以重建，不用修修补补。脏了、配错了、被人改乱了，跑一遍剧本拉回标准状态。这是"可重复"这条主线在配置层的落法，也是它跟前面虚拟机、容器接得上的地方。

> **提问：**测试环境被人手工改乱了，你怎么处理？

> **预设回答：**

- 答"重跑剧本，把配置拉回标准状态"——对，幂等剧本就是为这个准备的。

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

- 答"重装最干净"——干净是干净。容器和云主机重建便宜，物理机重装就太重了。

- 收口：对付环境漂移，用声明式配置盖回去，别一处一处去修。

> **易错点 / 考点：** ① 四个能力和"批量、集群"这两个特征要说得清；② 考得多的是 Ansible 与 Chef、Puppet 的差别，就落在有没有 agent；③ 判断题"Ansible 只能装软件"——错，服务启停、配置推送、批量命令它都管。

> **过渡：** 单个环节都会了，剩下的是把它们串成一条线。

【12.43】12.4.5 CI/CD 流水线（3 分钟）

> 先看工厂那条线。工位一个接一个，件从一头进去，从另一头出来。

>

> 课件说，快速交付、持续交付要靠一串自动化技术撑起来。清单是：持续构建、持续集成、持续测试、持续部署、持续运维。叫流水线，是因为它像工厂里生产产品的线。软件整个生命周期里的任务，用高度自动化的方式快速、有序地跑完。

>

> 五个持续各管什么。持续构建：代码一提交就编译打包，"能不能编过"随时有答案。持续集成：大家的改动频繁合到一起，合完立刻验，这是防合并地狱的唯一办法。持续测试：每轮构建自动跑测试，单元、接口、回归都算，它是线上的质量门。持续部署：测过的制品自动推进环境，不用人手工发版。持续运维：上线之后持续看，出问题能快速回滚。

>

> 工厂这个比方要抠一下。工厂线的要紧处不是快，是每道工序的输出正好是下一道的输入。而且每道工序都有检验。中间哪一道没检验，缺陷就往下流，流到最后返工最贵。

>

> 课件点明，Jenkins 2.0 是最常用的 CI/CD 流水线调度管理工具。调度管理四个字要看住。Jenkins 自己不编译、也不测试。它按顺序触发、按条件判断、收结果、通知人。干活的是它调用的那些工具。这也解释了前面为什么讲那么多工具：流水线是把它们串起来的胶水，不是替代品。

>

> 判断一条流水线好不好，有个土办法：从提交代码到拿到测试报告，要多久，要几个人。时间是反馈速度，人数是自动化程度的反面。理想状态是几分钟、零人工。做到这一步，团队才敢频繁提交；频繁提交又是持续集成的前提。这是个正循环。

>

> 最后补一句：流水线自己也要当代码管。配置进版本库，改动走评审，不然它迟早变成一个没人敢碰的黑盒。

> **提问：**流水线每一步都要人点一下确认，问题出在哪？

> **预设回答：**

- 答"某个环节还判不了自动，只能靠人"——方向对。要么测试没自动化，要么判断标准没写清。

- 答"人工确认更安全，重要环境这么做没错"——生产发布加人工审批是合理设计；测试环境每一环都要人点，流水线就白搭了。

- 答"能跑通就行"——跑通只是及格线。反馈被人工卡住，前面所有自动化的收益全被吃掉。

- 收口：分开看。测试环境追全自动，生产发布可以有门禁。

> **易错点 / 考点：**① 五个持续的名称和顺序要能默写；② 错得最多的是把持续交付和持续部署混成一个：交付是随时可发，部署是自动发出去；③ Jenkins 的定位是调度管理，不是构建工具，也不是测试工具。

> **过渡：**零件讲完了，接下来看两个最常用的是怎么搭在一起的。

【12.44】Jenkins+Docker 实现持续集成（2 分钟）

> 很多团队的构建机上，装着一堆没人敢动的东西。JDK 三个版本，构建工具两个，环境变量谁改的说不清。

>

> 课件在动手这块只给一句：具体搭建，见教材 P341—350。实操交给教材和实验，这里说清要搭出什么、容

> 易卡在哪。

>

> 要搭出的是：Jenkins 做调度，Docker 做执行环境。为什么是这两个？Jenkins 的调度需要干净、可重复的执行环境，Docker 正好给这个。传统做法是在 Jenkins 那台机器上装一堆工具和依赖。时间一长，它就成了谁都不敢动的古董。容器化之后，每次构建在一个全新容器里跑，跑完销毁。构建环境本身也可重复了。

>

> 一条典型流水线这么走。拉代码、构建（编译打包）、静态代码分析。接着单元测试、打制品、构建镜像。再推镜像、部署到测试环境、通知结果。每一站都对应前面讲过的一类工具：代码仓库、构建工具、静态分析。还有单元测试框架、镜像仓库、容器编排、通知渠道。

>

> 三个地方容易卡。第一，容器里的权限和目录。产物要能被 Jenkins 取走，工作目录就得挂出去，不然容器一销毁，产物跟着没了。第二，依赖要提前装进镜像。每次构建现装依赖，时间全耗在这上面。第三，凭据。镜像仓库的密码、部署用的密钥，不能明文写在脚本里，要用凭据管理注入。

>

> 还有一个观念要正过来：测试别放在流水线的末尾。很多人习惯先构建、再部署、最后跑测试，顺序是反的。测试不过，就不要往下走。质量门的位置在这儿，流水线的意义也在这儿。不是把流程串起来就完事，是在每一道门前替你拦住。

>

> 落点就一句：Jenkins 决定什么时候做什么，Docker 保证在哪做、每次都一样。

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"测试不过就别上线，得先拦住"——对，这就是质量门的含义。

- 答"部署完再测也一样，反正都要测"（典型错答）——不一样。不合格的版本已经进了环境，可能把正在用的好版本盖掉，还白搭一次部署。

- 答"先部署才能测，端到端要环境"——部分成立。端到端确实要部署后的环境，所以完整流水线是构建后跑单

元和接口、部署到测试环境再跑端到端。关键是每层都有门。

- 收口：位置不是绝对的，越早拦越便宜，越贵的门越靠后。

> **易错点 / 考点：** ① 典型阶段的顺序要能说出来；② 考得多的是为什么用容器做构建环境，答"构建环境本身要可重复"；③ 案例题问"流水线里测试总是不稳"，排查方向有环境不一致、依赖外部服务、用例之间数据打架，接得上 12.2.4 的数据隔离。

> **【思政落点 5】**：质量门说的是一件很朴素的事：没验过的东西不往下一站送。测试不过就不部署，代价是这一次慢半天；硬推下去，代价是别人替你收拾。工程上的责任不在口号里，就在自己能拦住的那道门前，不图省事放过去。这条习惯带出课堂，落到岗位上，就是把风险挡在自己这一环，别转给下游的同事和用户。

> **过渡：** 到这儿，这一章的零件和线都走过了。接下来收一收。

【12.45】本章小结（3 分钟）

> 复习最怕的是名词都记得，关系说不出来。四条小结，按"为什么、有什么、怎么造、怎么串"排一遍。这个顺序也是你答题的骨架。

>

> 第一条，测试基础设施很重要。课件列的要素是 IT 基础架构、软件资源、数据资源。这条对应 12.1 和 12.2。你要能答三个问题。它为什么重要：它是测试的前提，影响研发效能，也保障运维。它由什么构成：硬件、网络、软件、数据四类。怎么算好：能不能可重复地重建一个准生产级的环境。

>

> 第二条，搭建的方法和技术，包括虚拟机技术、容器技术、服务虚拟化技术。这条对应 12.3。三者的抽象层次要分得开。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外部服务。代表工具也对得上。QEMU-KVM 是虚拟机那一层。Docker 和 K8s 是容器那一层。WireMock 和 Hoverfly 是服务虚拟化那一层。

>

> 第三条，基础设施即代码（IaC）帮着把基础架构自动部署起来。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。这条对应 12.4.1 和 12.4.2。这组分类是要背下来的。选择题、简答题、案

> 例题都可能从这儿出。

>

> 第四条，怎么搭一条持续交付流水线。对应 12.4.3 到 12.4.5，也是实验 8 的内容。

>

> 用一句话概括整章。测试基础设施要做的，是让干净又和生产同构的环境落地。落到什么程度？一条命令、一次点击、一次提交，环境自己就起来了。做到这一步，测试才快得起来、稳得下来。做不到，测试就一直在等环境。

>

> 本章那五个坑也收个尾。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，几个人一个环境互相干扰。配置不一致，参数在各个环境里不一样。成本失控，云资源忘了关。五个坑的解法指向同一件事：把它写成代码。

> **提问：**用这一章的话说，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。IaC、容器、快照、模板都靠它串起来。

- 答"和生产一致"——也重要，可一致要靠可重复兜住。重建不出来，一致就只能靠运气维持。

- 答"稳定不宕机"——基本要求，不算核心特征。配置是手工调的机器，再稳也复制不出来。

- 收口：可重复是根，一致、稳定、快，都是它长出来的结果。

> **易错点 / 考点：**① 四条要能复述，特别是四类 IaC 工具和三类搭建技术；② 判断题"测试基础设施只要有硬件和软件"——漏了网络和数据；③ 综合题标准答法：先讲要素四类，再讲手段三层虚拟化，再讲自动化（IaC 四类加流水线），最后落到可重复、准生产、成本可控。

> **过渡：**理论收口，剩下的时间交给实验。

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

> 实验 8 的任务书只有一行。用 Jenkins Pipeline 和 Docker 搭一条 Java 应用的流水线。把一个 Java 应用从

编译、静态代码分析、单元测试、打包跑完。

>

> 先拆四个动作。编译，验证代码能不能变成可执行产物。静态代码分析，不动程序，查规范和潜在缺陷。第 3 章的静态测试在这儿变成了工程里的固定动作。单元测试，验最小代码单元的行为，还要产出覆盖率数据，第 5 章的内容。打包，产出可交付的制品。

>

> 四个动作的顺序不是随便排的。编译不过，后面三个都没得谈。静态分析和单测都能在制品出来之前给结论，所以摆在打包前面。反过来，先打包再检查，检查出问题了，这一包就白打。

>

> 为什么用 Docker？让这四个动作跑在一个干净、可重复的环境里。你本机装了什么 JDK、什么版本的构建工具，都不影响结果，活儿在镜像里干。这一章反复说的那句话在这儿落地：环境是代码的一部分。

>

> 为什么用 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：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。

>

> 交付物按这个清单交。第一，流水线脚本，要能看出四个阶段。第二，一次成功构建的日志，每个阶段的输出都在。第三，单元测试与覆盖率报告。第四，静态分析报告。第五，一份短说明，写清镜像怎么来的、凭据怎么配的、遇到过什么问题、怎么解决的。

>

> 评分看三点：一键能不能跑通，最要紧；四个阶段齐不齐、顺序合不合理；报告能不能说明质量状况。第五项我也看。把踩过的坑写清楚，是工程师的基本功。

>

> 最后一句：先让它跑通，再让它跑好。一上手就加并行、加缓存、加通知，往往一样都没跑成。最小可用优先。

提问：实验里单元测试失败了，流水线还要往下打包吗？

> **预设回答：**

- 答"不要，测试失败就停下，这是质量门"——对，也是流水线的基本纪律。

- 答"可以先打包，反正只是实验"（典型错答）——实验里放松，工作里就成习惯。而且"打包了但测试没过"的制品一旦流出去，就是事故的源头。

- 答"看失败原因，环境问题可以重试"——这个补充专业。失败要分类：代码问题就得停，环境抖动可以重试，重试要有次数上限。

- 收口：流水线值钱的地方不是自动跑，是自动判断能不能往下走。

> **易错点 / 考点：** ① 四个动作的名称和顺序要写对；② 错得多的是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，能不能跑出同样的结果。

> **过渡：** 这一章讲到这儿，最后留几句话。

【12.47】感 谢 聆 听（1 分钟）

> 第 12 章到这儿结束。

>

> 回头串一遍。这一章讲的事只有一件：测试能不能被信，取决于它跑在什么上面。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，测了也白测。

>

> 带走三个判断。第一，环境要可重复。重建不出来的环境，一致就只能靠运气。第二，数据要合规。真实数据能不用就不用，非用不可先脱敏。第三，资源要算清。能复用的别重复造，能自动清理的别靠人记。

>

> 这三条都不显眼，也都不在这一次的进度条上，所以总是第一个被省掉。省掉之后也不会当场出事。要等到半年后，有人在深夜发现环境起不来，或者一份报告被追问数据从哪来，才想起它们。

>

> 还有一层你得认下来：这些事做对了没人夸。环境一键起来，没人会为这件事表扬你。它一塌，全组停摆。这一章没有哪条能让你在答辩时多拿十分。它管的是你答辩那天，环境起不起得来。

>

> 交课程设计之前，自查三句就够。环境是不是一条命令就能拉起来。测试数据里还有没有真实的手机号。云主机上还开着几台忘了关的机器。三句都过了，这一章才算学进去。

>

> 还有个顺序别弄反。先把环境做成一条命令就能起来的东西，再谈自动化、谈性能、谈覆盖率。前一步不做，后面全是空转。

>

> 下一章走到测试的另一端：测试结果怎么管，缺陷怎么跟到底，质量怎么讲清楚。

>

> 实验 8 的流水线尽快跑通。卡住的地方记下来，下次课集中答疑。

> **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者 Dockerfile"——最划算的第一步，从手工装变成跑脚本。

- 答"先把测试数据脱敏，做成可回滚的基线"——很务实，直接解决数据污染。

- 答"等有机器了再说"——最常见的拖延。脚本化和 Dockerfile 不用新机器，一台自己的电脑就能开工。

- 收口：改一件小事，比规划一套大方案有用。

> **易错点 / 考点：**① 三组清单过一遍：四类要素、三类搭建技术、四类 IaC 工具；② 五个坑的名字和解法方向要配对，全部指向"写成代码"；③ 案例题收尾可用这句：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。

> **【思政落点 6】**：把生产库整库拷进测试环境，图的是省事，赌的是别人的手机号、住址和交易记录。数据资源是测试基础设施的一部分，合规不是额外手续，是使用它的前提。另一头，一台机器长期只用两三成算力，报表上看着宽裕，花掉的却是电和机位。技术上的自律，一头连着对他人权利的尊重，一头连着对公共资源的节制。

> **过渡：**下一章进入测试结果与缺陷管理，把这一章搭好的东西用起来。

四、思政落点清单（共 6 处）

序号	所在页	落点主题
1	12.5	同一批用例跑出两个结果，最省事的做法是挑好看的那个报上去。往前一步，后面所有决策
2	12.13	脱敏这件事，省一步就是拿别人的隐私换自己的方便。手机号、身份证、住址、交易记录一
3	12.27	云上按量计费，忘关一台机器，月底就多出一笔钱。这里讲的不是省钱小技巧，是公共资源
4	12.36	建完不验，等于交房不验收。环境建成功和建正确是两件事。多数事故不是不会建，是没人
5	12.44	质量门说的是一件很朴素的事：没验过的东西不往下一站送。测试不过就不部署，代价是这
6	12.47	把生产库整库拷进测试环境，图的是省事，赌的是别人的手机号、住址和交易记录。数据资