《软件测试课程设计》讲稿（第 1–3 章）

第 1–2 周 · 各 2 学时（90 分钟）

广东金融学院 计算机学院 | 周宇文

讲稿可照读版 | 由课件逐页撰写

第 1 章　引论

第 1 周 · 2 学时（90 分钟） | 教材：朱少民《软件测试方法和技术（第 4 版）》

《软件测试课程设计》第 1 章 讲稿（第 1 周 · 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 1 章 引论 —— 为什么必须做软件测试?
>
> 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）》，清华大学出版社 2022
>
> 配套课件：逐页还原版 61 页（`lecture1/`，云端 http://111.231.0.226:2345/lecture1/index.html）
>
> 本讲重点：软件缺陷的代价与质量成本观、软件测试的必要性（真实事故案例）、软件测试的定义与学科形成
>
> 本讲难点：区分测试/调试/质量保证三者的边界；理解测试左移与成本非线性增长
>
> **本讲稿为可照读完整版**：每页含【口播（可直接朗读）】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为时间紧张时可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出软件测试的定义（在指定条件下运行或评审系统与组件、观察记录结果、对照预期作出评价）；② 能列举至少 5 个真实缺陷事故并归纳其教训；③ 能说清测试与 SQA、与开发的关系，以及测试左移的含义；④ 能复述 TDD"测试在前、开发在后"的基本思想
能力	① 能用"是否运行程序"判断静态测试与动态测试；② 能区分测试、调试、质量保证三者的边界；③ 能针对给定事故案例指出缺失了哪一类测试（集成/兼容/性能/安全等）
素质（思政）	① 通过 Therac-25、波音星际客机、骑士资本等事故建立"质量即责任"的职业底线；② 通过《血狮》与国产软件案例理解工程诚信与工匠精神；③ 通过测试左移与缺陷成本曲线体会严谨求实、精益求精

本讲要建立的一个总印象：软件缺陷客观存在且代价高昂，**测试是发布前的质检关卡**，也是测试人员的职业责任。

二、时间分配总表

> ⚠ **容量提示**：本讲稿为**素材全量版**，口播合计约 3 万字量级（≈2 小时纯朗读）。90 分钟课堂扣除互动、练习、课间约 20 分钟后，纯讲授容量约 70 分钟。
>
> **请按以下方式取用**：① **重点页全文慢讲（2–3 分钟/页）**：1.11、1.13、1.14、1.15、1.17、1.23、1.31、1.32、1.35、1.37、1.49、1.51、1.54；② **常规页快速带过（0.5–1 分钟/页）**：只讲口播第一段＋板书＋1 个提问；③ **可课后阅读页**：★1.5、★1.6、★1.7、★1.8、★1.9、★1.60。

时段	页码	内容	分钟	备注
第 1 节	1.1–1.10	封面、课程导学、教材与三篇结构	10	讲清考核口径与课程设计任务
第 1	1.11–	案例导入：真实缺陷事故（迪士尼/Intel/火星/波音/骑士资本	16	1.13–1.15、1.17 重点

节	1.21	/AWS/Uber 等）		讲
第 1 节	1.22–1.34	1.2 为什么要测试；1.3.1 学科形成；1.3.2 正反思维	14	1.23、1.31、1.32 是考点
——	——	**课间休息 10 分钟**	——	——
第 2 节	1.35–1.45	1.3.3 定义（IEEE/ISO 29119）；1.3.4 其它视角（质量/风险/经济/Oracle/批判性思维）	16	1.35、1.37 必讲
第 2 节	1.46–1.52	1.4 测试与 SQA；1.5 测试与开发（并行协作、测试左移）	12	1.49、1.51 是难点
第 2 节	1.53–1.61	1.6 TDD 思想；问题、小结、思考与练习、学习资源、结束	12	1.54、1.57、1.59 必做

> **压缩优先级（时间不够按序压）**：★1.5–1.9（课程导学细节）→ ★1.60（学习资源改为课后阅读）→ 1.3–1.4（就业市场与敏捷时代合并为一句带过）→ 1.40–1.45（其它视角只讲风险与经济两个）。**1.13–1.15、1.17、1.23、1.31、1.32、1.35、1.37、1.49、1.51、1.54 是考点密集页，不要压缩。**

三、逐页讲稿

> 使用说明：每页第一段是**可直接朗读的口播**；其后是板书、提问与预设回答、易错点/考点、过渡语。板书建议边讲边写，不要一次写完。

【1.1】软件测试方法和技术（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 同学们好，欢迎来到《软件测试课程设计》。我是这门课的任课老师周宇文。这门课的教材是朱少民老师主编、清华大学出版社 2022 年出版的《软件测试方法和技术（第 4 版）》。先交个底：这是专业选修课，32 学时、2 学分，理论 16 学时加实验 16 学时，一半时间讲原理，一半时间在机房动手。考核方式是考查，看平时的过程性表现，加上最后的课程设计作品与答辩。今天先不急着学工具，我们回答一个最根本的问题——软件为什么要测？不测会怎样？把这个问题想明白，后面所有方法才有落脚点。

> **板书：**课程名 → 教材＝朱少民《软件测试方法和技术（第 4 版）》清华 2022 → 32 学时（理论 16＋实验

16）→ 考查（过程性＋作品答辩）

提问：你们觉得，一个软件完全不测试就上线，最坏会发生什么？

预设回答：①"大不了不好用，用户骂两句"——这是最典型的低估，我会追问：如果它管的是钱、是电、是医疗设备呢？②"会崩溃、会赔钱"——方向对但太笼统，要求他说清"哪一类损失、谁来承担"。③"现在都敏捷了，边开发边测，不用专门测"——这句话先记在黑板上，讲到敏捷那一页再回来算账。

易错点 / 考点：最常见的错，是把软件测试等同于"点点鼠标看有没有报错"。记三句话：测试是有理论、有方法、有流程的学科；测试的目标是发现缺陷、提供质量信息，不是证明软件是对的；测试不是开发之后的收尾工序。判断三步法：一问测什么（对象与范围），二问谁来测、何时测（主体与时机），三问测到什么程度算完（准则）。

过渡：坐标定下来了。接下来看一张图，它会告诉我们：软件测试这座山，到底是什么形状。

【1.2】软件测试全景图（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这张图叫"软件测试全景图"。它不是让大家背的，而是要在脑子里建一张地图。所谓全景图，就是把和软件测试有关的东西按几个维度铺开。横着看，是软件的生命周期：需求、设计、编码、集成、部署、运维，测试在每个阶段都有自己的活儿。竖着看，是测试的不同层次：单元测试、集成测试、系统测试、验收测试。再往里看，是测试的类型与方法：功能、性能、安全、兼容性、易用性，以及黑盒方法、白盒方法。最后还有一列是支撑体系：测试流程与规范、测试管理、缺陷跟踪、自动化工具。为什么先看这张图？因为新手最常犯的错，是手里只有一把锤子，就把所有东西都当钉子——只学一种方法，什么问题都拿它去套。有了全景图，你才知道自己站在哪儿，手上这把工具适合解决哪一类问题。打个比方，这就像去医院：全景图告诉你有内科、外科、影像科、检验科，你挂号之前得先知道自己大概是什么毛病。软件也一样，性能问题拿功能用例去测，是永远测不出来的。

板书：画一张四层图 → ①生命周期（需求→设计→编码→集成→部署→运维）②测试层次（单元→集成→系统→验收）③类型方法（功能/性能/安全/兼容/易用＋黑盒/白盒）④支撑（流程规范→管理→缺陷跟踪→自动化工具）

提问：如果用户投诉"这个 App 一到晚上八点就卡"，按全景图，你该去哪个格子找工具？

预设回答：①"找功能测试"——错，功能测试看的是"做得对不对"，不看"快不快"，我会引导他往性能格子

走；②"找性能测试"——对，但要追问：是响应时间、吞吐量还是并发用户数的问题？③还有同学会说"先看看是不是网络问题"——这个思路很好，说明他有了分层排查的意识，可以表扬并补一句：先用监控确认瓶颈在客户端、网络还是服务端，再选工具。

易错点 / 考点：容易混的是"层次"和"类型"——层次回答"在哪一级测"，类型回答"测什么属性"。判断题常考"性能测试属于系统测试阶段"这类说法，要按"层次＋类型"两个坐标分别作答。

过渡：地图看完了，我们把镜头拉近：这张地图上的那些词，彼此是什么关系？

【1.3】系统地理解软件测试（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页请大家跟着我一起看，标题叫"系统地理解软件测试"。图上有二十来个词：软件、质量、缺陷、测试、目标、测试用例、方法、设计、实施、发现、清除、阶段、管理、思想、源泉、确定、寻求、指导。看着散，其实是一条完整的逻辑链。第一，测试的对象是"软件"，追求的是"质量"，所以质量是整套活动的目标；没有质量这个目标，测试就变成"为测而测"。第二，质量的敌人是"缺陷"，缺陷就是测试直接要对付的东西——做测试，本质上就是找缺陷、报缺陷、推动清除。第三，"源泉"和"确定"回答的是测试从哪里开始：测试的源泉是需求与用户期望，需求不确定，测试就无从下手，所以测试要尽早介入，先把"测什么"确定下来。第四，"寻求""设计""方法""测试用例"是一条技术链：先寻求怎么测，再设计测试用例，再选合适的方法，用例是这件工作的核心产物。第五，"实施""发现""清除"是执行链：把用例跑起来，发现缺陷，推动清除。第六，"阶段""管理""思想""指导"是保障链：测试要分阶段推进，要靠管理组织资源、进度和缺陷，还要靠测试思想来指导决策。再打个比方，这整张图就像看病：目的是健康，也就是质量；症状是发烧，也就是缺陷；你不能看一眼就开药，得先问诊、验血、拍片，这就是分析与用例设计；确诊后治疗是清除缺陷，康复靠作息管理就是阶段与管理。放到软件行业里，拿一个登录功能说：需求写着"手机号加验证码可以登录"，这是源泉；用例要覆盖正常登录、验证码错误、账号被锁、网络中断、重复点击，这是设计；每个异常都要记成缺陷单、跟踪到关闭，这是管理与执行。所以一句话口诀：目标管质量，对象抓缺陷，源泉看需求，核心是用例，落地靠执行，保障靠管理和思想。

板书：质量（目标）→ 缺陷（对象）→ 需求（源泉）→ 用例（核心）→ 方法/设计（怎么测）→ 实施/发现/清除（执行）→ 阶段/管理/思想（保障）

提问：请用一句话说说，"缺陷"和"测试用例"是什么关系？

预设回答：①"用例是用来找缺陷的"——对，但不够深，我会补充：用例是把"希望发现的缺陷"提前翻译成

可执行步骤；②"用例就是操作步骤"——典型的窄化，要提醒用例还必须写明前置条件、输入数据、预期结果和判定准则；③有同学会说"没有预期结果就不算用例"——这句很到位，可以顺势强调：没有预期结果，就无法判断实际结果是"对"还是"错"。

易错点 / 考点：这一页最爱考"关系"而不是"定义"。记住一条主线：需求是源泉，用例是核心，缺陷是对象，质量是目标。选择题常把"测试的目的是证明软件没有缺陷"当干扰项，这句话是错的——测试只能证明缺陷存在，不能证明缺陷不存在。

【思政落点 1】：这二十来个词里，"清除"两个字最见职业底线。找缺陷是技术活，报缺陷是良心活。行业里有真实教训：发现缺陷却因为怕影响绩效、怕得罪人而选择隐瞒，最后问题流到用户手上，代价由公众承担。请同学们记住——测试人员手里握着的是别人看不见的风险，如实记录、不隐去不利结果，是这个岗位的第一条职业底线，也是一名工程师对国家、对社会负责的起点。

过渡：原理讲完了，大家可能更关心一个现实问题：学这个，将来干什么？我们看两张招聘市场的图。

【1.4】从就业市场看软件测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是两张图，都来自招聘市场。第一张是岗位与需求分布，第二张是岗位要求里的技能关键词。据课件数据，招聘平台上与软件测试相关的岗位长期保持稳定需求，测试工程师、自动化测试工程师、测试开发工程师是三类最常见的职位名称。请大家重点看第二张图里的技能词：功能测试、用例设计、缺陷管理、数据库、接口测试、性能测试、自动化脚本、Linux、Python 或 Java。有没有发现？这些词跟咱们这门课的大纲几乎一一对应——用例设计、缺陷管理、接口测试、性能测试、自动化工具，后面都会讲到。为什么要先看就业？因为学习最怕"不知道学了有什么用"。这张图告诉大家：测试不是"点鼠标的岗位"，而是一个有明确能力栈的岗位。招聘方要的不是"你会不会用某个工具"，而是"你能不能分析需求、设计用例、定位问题、说清缺陷并推动它被修掉"。把这张图记在心里，以后每学一个方法，都问自己一句：这一条能不能写进简历，能不能在面试里讲出一个具体例子？

板书：三类岗位（测试工程师／自动化测试／测试开发）→ 技能栈：功能测试・用例设计・缺陷管理・数据库・接口测试・性能测试・自动化脚本・Linux・Python/Java

提问：招聘要求里为什么不直接写"会用某某工具"，而写"用例设计能力"？

预设回答：①"工具学得快，能力难练"——很准，工具两星期能上手，设计思维要一学期；②"工具会过

时"——也对，方向一致；③"因为公司用的工具各不相同"——这是最实在的回答，正好引出结论：工具是载体，方法是本事；方法迁移得走，工具只是脚手架。

易错点 / 考点：别把"软件测试"理解成"软件测试工具"。考试如果问"测试工程师最核心的能力是什么"，落在"测试分析与用例设计"上，而不是某个工具的操作步骤。

过渡：既然岗位在变，那今天的行业跟十年前比，到底变了什么？我们看敏捷时代这一页。

【1.5】在敏捷时代，如何看待软件测试？（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页标题是个问题：敏捷时代，怎么看待软件测试？课件给了五条判断，我逐条讲。第一条，敏捷开发模式提倡整个团队对质量、对测试负责。注意这句话的分量——测试不再是"测试组的事"，产品、开发、测试都得为质量签字。第二条，开发与测试越来越融合，比如课件提到的微软测试人员转型：过去微软有庞大的专职测试岗，后来大量测试岗位被并入开发角色，测试能力变成开发者的基本功。第三条，持续交付倒逼持续测试，但测试最容易成为瓶颈。为什么？因为代码一天能集成十次，自动化测试要是跑一次要两个小时，交付就被卡住了。这就像高速公路：路修得再宽，收费站只有一个窗口，车照样堵。第四条，测试左移、测试前移，让开发做更多的测试——把缺陷拦在需求和设计阶段，越早越便宜。第五条，测试驱动开发，英文缩写 TDD：测试在前、开发在后，先写测试再写实现。这里再补一个生活化的类比：敏捷时代的质量，就像一支球队的防守——不是后卫一个人的事，前锋也要回防，教练也要布阵；如果大家都觉得"丢球是后卫的问题"，这支球队一定丢球。软件行业里还有一个很典型的现象：很多团队每天多次把代码合并到主干，每次合并都会自动触发一轮测试，谁把测试跑红了，谁就得马上修，绝不允许红色状态过夜。这就是"持续测试"的真实样子——测试不是一道关卡，而是开发流水线上的传送带。所以课件才提醒我们：这条路最怕的就是瓶颈。一旦自动化测试跑得比开发还慢，团队就会开始"绕过测试"，而一旦绕过，质量就会以看不见的方式持续下滑。大家把这一页记成一句：质量是全员责任，测试要与开发并行，而不是排在开发后面。

板书：①全员质量责任 ②开发测试融合（微软测试转型）③持续交付→持续测试（瓶颈风险）④测试左移/前移 ⑤ TDD：测试在前、开发在后

提问：为什么说"测试最容易成为瓶颈"？如果你们团队的自动化测试要跑两个小时，你会怎么改？

预设回答：①"因为测试最慢"——太表面，我会追问慢在哪里；②"因为每次提交都要跑全量测试"——这就说到点子上了：可以分层，提交时跑冒烟与单元测试，夜间跑全量与性能；③"可以并行、可以分布式跑"——也对，属于工程手段；④有同学说"那就少测一点吧"——这是危险回答，绝不能靠减少测试来提速度，

要换成"提高测试的性价比"，说清哪一层该测什么。

易错点 / 考点：TDD 的顺序一定要记准——先写测试，再写实现，最后重构；顺序反了就变成"补测试"。判断题常把"测试左移"错误理解为"把测试人员提前招进来"，左移的核心是测试活动在时间轴上向前移，测试人员更早介入需求与设计。

过渡：讲到这里，有位同学可能会想：那我这门课到底是要背概念，还是要练本事？下面两页，一页讲能力，一页讲方法。

【1.6】如何借助这门课培养学生的分析问题和解决问题的能力？（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页的问题很直接：这门课怎么培养大家分析问题和解决问题的能力？课件给了三个抓手。第一个抓手是"强调分析能力，测试分析是基础"。请注意，测试分析是基础——拿到一个功能，先要分析它有哪些输入、哪些分支、哪些边界、哪些异常，分析不清楚，用例就只能靠拍脑袋。第二个抓手是"批判性思维，比如探索式测试"。探索式测试要求你一边学习系统、一边设计用例、一边执行、一边根据结果调整下一步——它不相信"需求写完就万事大吉"，而是主动去质疑："这里是不是有问题？"第三个抓手是"工程思维，找到不同的解决方案，从中选出最优的"。这句话是这门课的灵魂：现实中的测试问题，从来没有唯一解。时间有限、人手有限，你是全量回归，还是按风险挑重点回归？这不是聪明不聪明的问题，是工程权衡的问题。举个生活中的类比：装修房子，预算和工期都有限，你是全屋精装还是先把水电做好？会算账的人，才是好工程师。那批判性思维怎么练？就是刚才说的探索式测试，它不是"随便点点"，而是带着问题去试探系统：先花二十分钟熟悉功能，再列出一批"我怀疑会出问题"的点，比如改价格后再改数量、优惠券和积分同时用、页面刷新一半就断网，然后一边试一边记录，试完复盘的发现又变成新的怀疑。这个过程就像侦探查案——不是等案情自己送上门，而是主动去找矛盾之处。工程思维又怎么练？还是拿购物车举例：如果测试时间只够覆盖一半功能，你是按页面从头测到尾，还是先把"下单、支付、退款"这条主链路测透？成熟的答案一定是后者，因为主链路出错的代价最大。这就是风险驱动：把有限的测试资源，压在出错代价最高的地方。我们这门课的实验和作业，都会逼着大家做这种权衡，并说明你权衡的理由。

板书：分析能力（测试分析是基础）→ 批判性思维（探索式测试）→ 工程思维（多方案比较→选最优，并说明理由）

提问：同一条需求，张三设计了 8 条用例，李四设计了 30 条。谁的测试更好？

预设回答：①"李四更全面"——这是最常见的直觉，我会反问：如果上线时间只剩两小时，30 条跑得完吗？

②"张三更好，因为效率高"——也未必，如果那 8 条漏掉了核心分支，效率就是假效率；③比较理想的回答是"要看覆盖了什么，而不是看数量"——这就讲到关键：用覆盖率和风险来评价，而不是用条数。顺势提醒：用例数量不是绩效，漏测才是事故。

易错点 / 考点：别把"探索式测试"当成"随便点点"。它是有方法、有记录、有章程的：先明确测试目标与时间盒，边测边记，事后复盘并沉淀为可复用的用例。简答题若问"分析能力与工程思维的区别"，答：分析能力解决"有哪些问题要测"，工程思维解决"在约束下先测哪些、怎么测最划算"。

过渡：能力怎么练出来？光靠听讲不行，得动手。下一页讲实验教学。

【1.7】如何通过实验教学培养学生的实际能力？（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页讲实验。课件第一句就说：软件测试是一门实践性很强的课程。这句话不是套话——你可以把等价类划分的定义背得滚瓜烂熟，但面对一个真实的登录页面，你还是会漏掉手机号带空格、密码含中文、验证码过期这些情况。所以这门课的方法是"做中学"：听一遍，不如自己做一遍。课件还给了两条方法：问题驱动教学与实验；实验就是分析问题、解决问题的过程。什么是问题驱动？就是先给你一个"跑不通""算不对""会崩"的现象，然后你自己去定位：是环境问题、数据问题，还是程序逻辑问题。这个过程跟将来上班是一模一样的——领导给你一个 bug 单，上面只有一句"用户反馈下不了单"，剩下的全靠你自己查。其实道理跟学游泳、学开车一样：教练讲一百遍换气和打方向盘都没用，你得自己下水、自己上路；而实验课就是你们的泳池和练车场，在这里犯错不要钱，在工作里犯错很贵。软件行业里有个很朴素的经验：同一个缺陷，在需求评审时发现，改一句话；在设计阶段发现，改一段设计；在编码阶段发现，改几行代码；上线之后才发现，就要改代码、改数据、发版本、安抚用户，甚至上门道歉——越往后越贵。所以我们做实验，练的不只是"会不会用工具"，而是在练"我能多早发现它"。我给大家一个忠告：实验课别抄步骤、别只截图。真正长本事的是三件事——第一，把现象记录清楚（什么环境、什么数据、怎么操作、出现什么）；第二，把猜想和验证过程写下来，哪怕猜错了也写；第三，把结论沉淀成可复用的用例或检查清单。这门课的平时分，很大一部分就看这三件事做得实不实。

板书：做中学 → 问题驱动（给现象，找原因）→ 实验＝分析问题＋解决问题 → 记录三件事：现象/环境/数据、猜想与验证、沉淀用例清单

提问：实验报告里只写"运行成功，结果正确"，为什么拿不到高分？

预设回答：①"因为没分析"——对；②"因为没有记录过程"——也对；③"因为不知道他到底测了什么数

据"——这正是关键：没有输入数据和预期结果，这份报告无法被复现，也就无法被信任。补一句：可复现是测试报告的生命线。

易错点 / 考点：常考"缺陷报告必备要素"：环境、版本、前置条件、操作步骤、实际结果、预期结果、复现概率、附件证据。判断三步法：别人按你写的步骤能不能复现？缺陷能不能被定位？不写预期结果的报告，等于没写。

过渡：方法有了，那这门课用什么教材、学哪几块内容？我们看下一页。

【1.8】《软件测试方法和技术》（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

介绍一下我们手里的这本教材。据课件数据，这本书 2005 年出版，现在是第 4 版，是"十一五""十二五"国家级规划教材，也是上海市普通高校优秀教材；有 300 多所大学在使用，销量超过 20 万册，并连续三年获得清华大学出版社畅销书奖。这几个数字说明什么？说明它是一本被长期验证过的教材，不是速成手册。再看结构，本书共分三篇：第一篇"软件测试的原理与方法"，第二篇"软件测试技术"，第三篇"软件测试项目实践"。这个三篇结构其实对应了我们学习的三个阶段：先懂原理（知道为什么这么测），再学技术（知道用什么工具、什么方法测），最后做项目（在真实的约束下把前面学的用起来）。所以大家在读这本书的时候，不要一上来就跳到工具章节，工具是学得最快的，原理才是决定你上限的东西。这个三篇结构，很像医学院的培养路径：第一篇是基础医学，教你人体是怎么运转的；第二篇是临床技术，教你怎么用各种仪器去查、去治；第三篇是临床实习，把你扔到真实的病人面前做判断。没有第一篇，你看到症状只会乱开药；没有第三篇，你背再多书也不敢上手术台。我们这门课同样如此：第 3 章的方法如果脱离了原理，就会变成"照着公式套用例"；而课程设计如果不动手，你也永远不知道真实的软件有多不听话。所以我的建议是，读教材的时候手里要有三支笔：第一支划概念，第二支记工具怎么用，第三支写"这个方法在什么场景下会失效"。第三支笔最重要，因为任何测试方法都有它测不到的东西，知道边界，才知道什么时候该换方法。也提醒一句：教材第 4 版跟我们这门课的课件是对齐的，课后作业里出现的概念，都能在书上找到出处，遇到不懂的先翻书，再问老师。

板书：2005 年首版 → 第 4 版（2022）→ 国家级规划教材 · 上海市优秀教材 → 300+ 所高校使用 · 销量 20 万册+ · 连续三年清华畅销书奖 → 三篇：原理与方法 | 技术 | 项目实践

提问：为什么教材要分成"原理与方法""技术""项目实践"三篇，而不是直接把工具从第一页讲到最后一页？

预设回答：①"因为要先懂原理"——对；②"因为技术会过时，原理不会"——这个回答很成熟，可以表扬；

③"因为最后要做项目"——也对：知识只有在项目里被串起来才会变成能力。可以把三种回答合成一句：原理决定你会不会想，技术决定你会不会做，项目决定你能不能交付。

易错点 / 考点：注意区分"教材版本"与"课程课时"这类事实性信息，别把第 4 版记成第 3 版，也别把 32 学时记成 64 学时。客观题常考教材结构与出版信息，记住"三篇"和"2022 年第 4 版"。

过渡：三篇里我们最先进入的是第一篇。第一篇包含哪四章？看下一页。

【1.9】第 1 篇 软件测试的原理与方法（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这是第一篇的目录页，包含四章。第 1 章"引论"，也就是我们今天开始讲的内容，回答的是"为什么必须测、什么是测试"这些根本问题；第 2 章"软件测试的基本概念"，把测试的分类、级别、静态与动态、黑盒与白盒这些概念系统化；第 3 章"软件测试方法"，进入具体方法：等价类划分、边界值分析、判定表、因果图、路径覆盖等等；第 4 章"软件测试流程和规范"，讲测试怎么在团队里组织起来，从测试计划、测试设计到缺陷管理、测试报告。大家注意这个顺序：先讲必要性（为什么），再讲概念（是什么），再讲方法（怎么做），最后讲流程（怎么组织起来做）。这是一条从"想明白"到"做得对"再到"做得稳"的路线。很多同学做课程设计时最痛苦的不是不会用工具，而是不知道从哪儿下手，根源就是跳过了前两章。所以第一篇这四章，请务必跟着走扎实。

板书：第 1 篇 = 第 1 章 引论（为什么）→ 第 2 章 基本概念（是什么）→ 第 3 章 测试方法（怎么做）→ 第 4 章 流程和规范（怎么组织）

提问：为什么"流程和规范"要放在"方法"之后讲，而不是一开始就讲？

预设回答：①"因为要先会方法才知道流程管什么"——很到位；②"因为流程是给团队用的"——也有道理，个人可以先会方法，团队才需要流程；③"因为流程不重要，可以最后看"——要纠正：流程不是不重要，而是它依赖于方法；没有方法的流程只是一堆表格。

易错点 / 考点：常考"四章的顺序与各自主题"的对应关系。记忆口诀：一论（引论）二概（概念）三法（方法）四流程。

过渡：好，教材结构清楚了，我们正式进入第 1 章。

【1.10】软件测试方法和技术（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

翻到这一页，我们正式进入第 1 章"引论"。这一章在整个课程里的位置，相当于盖房子的地基。它不教你某个具体的用例设计技巧，而是回答三个问题：第一，为什么必须做软件测试——用真实事故说话；第二，什么是软件测试——把它作为一个学科来定义；第三，测试和质量保证、和开发到底是什么关系。这三问答不清楚，后面学得越多越乱：你会把测试当成开发的附属品，会把质量保证和测试混为一谈，也会在执行时失去判断标准。所以接下来这一章，我会带着大家看一批真实案例，每个案例都问一句：如果当时有规范的测试，哪些问题本来可以被提前发现？请大家带着这个问题听。

板书：第 1 章 引论 → 三问：为什么必须测／什么是测试／测试与质保、开发的关系

提问：如果让你只用一个词概括第 1 章的主题，你选哪个词？

预设回答：①"定义"——只答对了一半，定义只是其中一问；②"案例"——案例是手段不是主题；③"必要性"——这个最贴切，第 1 章的主线就是回答"为什么非测不可"。

易错点 / 考点：第 1 章的考点集中在"测试的必要性""测试的定义""测试与调试、质量保证的区别"三处。注意"测试≠调试"：调试是定位并修复缺陷的开发活动，测试是发现缺陷、提供质量信息的活动。

过渡：第 1 章要讲哪六节？我们看全章目录。

【1.11】1.1 软件测试的必要性（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这是第 1 章的目录，一共六节：1.1 软件测试的必要性；1.2 为什么要进行软件测试；1.3 什么是软件测试；1.4 测试和质量保证的关系；1.5 测试和开发的关系；1.6 测试驱动开发的思想。请特别注意 1.1，它下面挂着一串案例：迪斯尼的多媒体游戏、一个缺陷造成数亿美元损失、火星探测飞船坠毁、软件测试走捷径导致灾难再次发生、错误指令造成骑士资本集团损失（课件数据为 4.4 亿美元）、AWS 宕机整整 4 个小时、预定的酒店住不进去露宿街头、Uber 泄漏个人隐私导致用户要求赔偿 3 亿多元，还有更多悲剧。我先把这一串名字报出来，大家不要急着记数字，先记住一件事：这些不是故事会，而是我们这门课的"病历本"。每看完一个案例，我都要问你们一句——这件事如果放在今天，用我们后面要学的方法，能不能提前发现？

板书：1.1 必要性｜1.2 为什么测｜1.3 什么是测试｜1.4 测试与质量保证｜1.5 测试与开发｜1.6 TDD → 1.1

案例串：迪斯尼・奔腾缺陷・火星飞船・捷径重演・骑士资本・AWS 宕机・酒店预订・Uber 隐私

提问：这些事故来自游戏、芯片、航天、金融、云计算、出行、酒店等完全不同的行业，它们的共同点是什么？

预设回答：①"都是软件出问题"——太笼统；②"都造成了很大损失"——对，但还要再进一步；③"共同点是：缺陷已经流到了用户端才被发现"——这个回答最好，正好点出测试的价值：把发现缺陷的时点往前移。

易错点 / 考点：容易把"缺陷"和"故障"混着用。缺陷是软件中存在的问题，故障是缺陷被触发后表现出来的异常。案例题答题模板：现象→影响范围→根因（哪一类缺陷）→如果做了哪一级测试可以拦截。

过渡：案例从哪里讲起？我们先看 1.1 这一节的开篇。

【1.12】1.1 软件测试的必要性（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这是 1.1 节的开篇页。按课件的说明，这一节要先列写本课时的学习要点，再依次深入讲解；如果有学习要求，就写在要点之后。课件还给了我们讲每一节要走的六个动作：讲概念——这个概念是什么，工作中什么场景会用到；讲讲解——要胜任这个能力，需要掌握哪些知识和技术；举例子——用一个案例或一个演示帮大家真正看懂怎么用；分享经验——特别是理论和实际有差异的地方；给学习建议——告诉大家在真实学习或工作场景里可以怎么用；提供实用工具——这一节对应的工具或模板。请大家把这六个动作记成一条学习路径：概念→知识→案例→经验→建议→工具。这也是我要求你们做课程设计时的顺序：先弄明白要解决什么问题，再选方法，再动手，最后沉淀成模板。接下来的 1.1.1，我们就从第一个案例开始。

板书：六步学习路径：概念（什么场景用）→ 知识技术（要会什么）→ 案例/demo（怎么用）→ 经验分享（理论与实际的差）→ 学习建议 → 实用工具/模板

提问：为什么课件要把"经验分享——理论和实际的差异"单独列成一个动作？

预设回答：①"因为实际比理论复杂"——对，这是核心；②"因为书上写的做不到"——过于绝对，应该说"现实中要受时间、人力、环境的约束"；③"因为老师有工作经验"——也算一个原因，但重点不在老师，而在你们将来要面对的取舍。

易错点 / 考点：常考"学习方法论"类简答题，答"概念—知识—案例—经验—建议—工具"六步即可拿分；若问"理论与实际的差异体现在哪"，答：约束条件（时间、成本、环境、人手）导致的取舍与优先级排序。

过渡：好，第一个案例来了——圣诞节的迪斯尼。

【1.13】迪斯尼并不总是带来笑声（5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这个案例的标题叫"迪斯尼并不总是带来笑声"。据课件数据，1994 年圣诞节前夕，迪斯尼公司发布了第一个面向儿童的多媒体光盘游戏"狮子王童话"。圣诞节后的第一天，客户支持部的电话就开始响个不停，不断有人咨询、抱怨：为什么游戏总是安装不成功？为什么装上了也没法正常使用？事后发现，这个游戏软件只能在少数系统上正常运行，课件把它归为一类：兼容性问题。圣诞节前发布，是典型的"节日档期"发布：发布时间定死，留给测试的窗口被压缩。买游戏的是家长，圣诞早晨孩子拆开礼物等着玩，结果装不上、进不去、没声音，家长的第一反应不是"这是兼容性缺陷"，而是"迪斯尼的东西怎么这么差"。这就是质量问题的真实代价：它不只是返工成本，更是品牌信誉的损失，而信誉的修复成本远高于当初把测试做完整的成本。兼容性问题为什么难躲？因为它本质是组合爆炸：操作系统版本、显卡与声卡驱动、光驱型号、解码器、区域语言、分辨率，每一项都可能不同。你不可能把所有组合都买齐了测，所以兼容性测试的核心不是"全都测"，而是先分析用户群，找出主流配置与高风险组合，按优先级覆盖，这正是我们后面要学的等价类划分和风险驱动测试的思想雏形。如果当年有规范的测试流程，哪些问题可能被提前发现？至少三件事：第一，需求与设计阶段就明确"支持的运行环境清单"，并写进验收条件，不在清单里的系统就明确不支持、在包装上写清楚；第二，系统测试阶段针对主流配置矩阵做真实环境实测，而不是只在开发机上验证；第三，发布前让没参与开发的人从"拆包装"开始走一遍真实安装体验。这类缺陷放到今天的手机 App 上就是"某个机型闪退"，放到网页上就是"某个浏览器排版错乱"——本质没变。

板书：1994 圣诞前夕发布"狮子王童话"→ 节后客服电话不断（装不上/用不了）→ 根因：兼容性问题（只能在少数系统正常运行）→ 代价：用户流失＋品牌受损 → 对策：环境清单进需求 → 配置矩阵按风险覆盖 → 第三方真实安装体验

提问：如果你是迪斯尼当时的测试负责人，只有一周时间，你会怎么安排兼容性测试？

预设回答：①"把所有系统都买来测"——时间成本都不允许，我会追问：一周够吗？钱够吗？②"按市场占有率排，先测主流的三个系统"——这是很实际的答案，可以表扬，并补充还要看"用户群特征"：买儿童游戏的家长，家里电脑往往偏旧或偏杂牌；③"测一下开发机不就行了"——要立刻纠正，这正是案例翻车的根源：只在开发环境验证，等于没测兼容性；④"先做需求评审，把支持范围写清楚"——这个回答境界最高，说明他把测试往左移了，可以不测的组合就不测，但不能装作支持。

易错点 / 考点：这一页最常考"缺陷类型的判断"：安装不上、运行不了、只在少数系统正常 → 兼容性缺陷，而不是功能性缺陷。判断三步法：一，换环境问题就消失吗？是→兼容性；二，功能在支持的环境里是否按规格实现？是→不是功能缺陷；三，有没有明确的"支持环境清单"？没有→需求与文档缺陷，也要记一笔。简答题如果问"兼容性测试怎么做"，答题要点：确定支持矩阵→分析用户配置分布→识别高风险组合→按优先级与等价类覆盖→形成清单并写入发布标准。

【思政落点 2】：迪斯尼这个案例里，损失最大的是什么？是那些圣诞节早晨失望的孩子和他们父母的信任。测试人员的岗位之所以重要，是因为我们守着产品交给公众前的最后一道关。课件后面还会讲到放疗仪致死、航天器坠毁这类更沉重的案例，也会有国产游戏《血狮》的教训——把没有经过充分验证的产品当作成熟产品去卖，透支的是用户的信任，也是行业的信誉。请把"质量责任"四个字写进职业观：你签下的每一个"测试通过"，都可能关系到别人的时间、财产，甚至安全。

过渡：迪斯尼的代价是口碑，下一页我们要看的这个案例，代价直接用亿美元来计算——一个缺陷，究竟能贵到什么程度？

【1.14】一个缺陷造成了数亿美元损失（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

我们接着往下看。屏幕上这道算式很奇怪，大家跟我一起读一遍：4195835 除以 3145727，再乘以 3145727，最后减去 4195835。学过小学数学的同学都知道，一个数先除以某个数、再乘回来，等于什么都没做，所以结果应该是 0。可是当年有一批计算机，算出来的偏偏不是 0。这就是著名的奔腾浮点除法缺陷——CPU 在做浮点除法的时候，某些特定数值的组合会算错。

大家注意，这不是小毛病。CPU 是计算机的心脏，它算错，就意味着所有跑在它上面的软件都可能算错。如果是我做作业算错一道题，最多扣两分；可如果这个 CPU 装在银行系统里、装在飞机的飞控里、装在医院的放疗设备里，那就是钱算错、飞机失控、剂量算错。

那么 Intel 是怎么发现的？不是自己测出来的，是一位数学教授在做研究的时候发现自己的计算结果对不上，一路排查才定位到 CPU。这就要命了——它说明这个问题在出厂前没有被测出来。最后的代价是什么？课件写得很清楚：Intel 公司付出很大代价，回收 CPU，造成 4 亿美元损失。这个数字是按课件给的，4 亿美元，而且请注意，这是上世纪九十年代中期的 4 亿美元。

我们打个生活里的比方：这就像一家奶粉厂，出厂检验只抽检了几罐，其余全凭信心发货，等消费者吃出问题才召回。召回的成本，远远大于出厂前把每一批都检一遍。

那这一页要带走什么？第一，缺陷的代价不是线性的，越往后越贵；第二，一个底层的小数点错误，可以放大成几亿美元的商业灾难。这也正是我们这门课反复要讲的一件事——测试不是给开发挑毛病，测试是替用户和公司把风险挡在出厂之前。

板书：奔腾浮点除法缺陷 → （a÷b）×b−a ≠ 0 → 出厂前未发现 → 回收 CPU → 4 亿美元损失（据课件数据）→ 缺陷代价：越晚发现越贵

提问：如果这道算式在你自己的电脑上算出来不是 0，你判断问题出在软件、硬件还是编译器？为什么？

预设回答：① 多数同学第一反应是"软件 BUG"——点评：思路可以理解，但这次错在 CPU 的浮点运算单元。这说明缺陷不只藏在代码里，硬件、固件、编译器都可能带缺陷，我们讲的"被测对象"边界比想象中宽。② 有同学说"编译器优化错了"——很好，这可以追问他怎么验证：如果是编译器问题，换一个编译器结果就该对；而当年的现象是换语言、换程序都错，所以定位指向硬件。③ 有同学说"不知道从哪下手"——正好给他一个判断三步法：先看能否稳定复现 → 再看换软件环境是否还错 → 最后把范围缩到硬件或固件。

易错点 / 考点：一是数字要记清，4 亿美元是"回收 CPU 造成的损失"，不是"赔偿用户的金额"。二是概念，本案例对应的是浮点计算（精度/计算类）缺陷。三是常考结论："缺陷发现得越晚，修复代价越大"，给一个"早期改一行代码 vs 上线后召回"的情境让你判断成本高低。口诀：**先除后乘，回不到原点就是缺陷。**

【思政落点 1】这个案例最扎心的地方在于：问题不是测不出来，而是一开始选择了"先发货，再说"。企业算的是召回成本，用户承担的是算错的后果。要让学生记住一条职业底线——测试人员手里拿的不是一张验收单，而是用户的钱包、健康甚至生命。将来到了企业做测试，如实记录、如实上报是最基本的操守：发现缺陷不报，比没发现更严重。

过渡：奔腾的缺陷好歹只是算错数。接下来这一页，代价从"算错钱"变成了"船毁人亡"——我们看火星探测飞船。

【1.15】火星探测飞船坠毁（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

上一页的损失是钱，这一页的损失是人命和几十亿的投入。课件讲的是"火星探测飞船坠毁"。课件给出的原因链是这样的：飞船上的着地开关，本应该在真正触地的时候才被触发，可是机械震动在大多数情况下也会触发这个开关，于是就设置了一个错误的数据位。设想一下，飞船开始着陆的时候，计算机看到这个错误的数据位，极有可能判断"我已经落地了"，于是关闭推进器。而实际上飞船还在半空中，距离地面还有 1800 米。

没有反推进器的帮助，飞船从 1800 米的高度冲向地面，必然会撞成碎片。这里的"着地开关""错误的数据位""1800 米"都是课件原文给的数字和描述。

但这一页真正值得我们记住的，是课件的最后一句话："两个小组本身的工作都没什么问题，就是没有合在一起测试，其接口没有被测，而问题就在这里。"课件把这一页的缺陷类型标成四个字——集成测试不足。这才是这一页的教学落点，飞船只是代价。

什么叫做集成？我打一个比方：一个乐队，鼓手自己练得很好，小提琴手自己也练得很好，可是从来没有合排过一次。上台那天，各弹各的，节奏对不上，曲子就崩了。软件也一样：A 小组写的模块对外输出的是"米"，B 小组接收的时候以为是"英尺"，各自单元测试全绿，一合起来就出事。

这就是单元测试和集成测试的区别：单元测试回答"我这个零件好不好"，集成测试回答"这些零件接在一起转不转得动"。很多同学写程序，测试就是把自己那几个函数跑一遍，这叫单元测试；真正的风险往往藏在接口、参数、时序、数据格式这些"接缝"上。所以记一个判断三步法：一看模块本身对不对，二看接口约定一致不一致，三看连起来跑通没跑通——第三步不做，前面两步都是自娱自乐。

板书：火星探测飞船坠毁 → 着地开关被机械震动误触发 → 写入错误数据位 → 计算机误判"已着陆"、关闭推进器 → 从 1800 米坠毁（据课件数据）→ 病根：**集成测试不足（接口没测）**

提问：两个小组各自测试都通过了，为什么合起来还会崩？请举一个你在做课程作业时遇到过的"接口对不上"的例子。

预设回答：① "他们没沟通好参数含义"——对，这就是接口约定问题。追问一句：这个约定该在什么时候定下来？答：设计阶段就该冻结，测试用例要围绕接口来设计。② "这不是测试的问题，是设计的问题"——这是很好的回答，要点评：缺陷的根因可能出在设计，但拦截它靠的是集成测试，测试也要前移去评审设计。③ 典型的错误回答："只要每个模块都测好了，系统就不会有问题"——这是最高频的错误认知，模块正确推不出系统正确。

易错点 / 考点：最易混的是单元测试与集成测试。"每个模块都正确"推不出"系统正确"，接口未测是集成测试缺位的典型症状。考试常给"分工开发、单独测试全通过、联调失败"的情境，问缺了哪个级别的测试——答集成测试。口诀：**零件都好，装起来未必好；接口不测，等于没测。**

过渡：如果说火星飞船是"没合起来测"，那下一页波音的故事更值得琢磨——它不是没时间测，而是主动把测试拆碎了，走了捷径。

【1.16】软件测试走了捷径导致灾难再次发生（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页的标题很刺眼——"软件测试走了捷径导致灾难再次发生"。主角是波音公司的载人飞船"星际客机"。课件的原文是：波音公司测试载人飞船星际客机软件系统的程序存在严重缺陷，现在计划对测试程序进行修改；造成程序存在严重缺陷的主要原因就是软件测试走了捷径——该公司缩短了对该飞行器软件的一次关键测试，他们把整个飞行过程分成了几个小单元分别进行测试，但最后却没有做完整的、端到端的集成测试，也就是没有进行时长为 25 个小时的整体测试。这里的"25 个小时"是课件给的数字，注意它是那一次整体测试的时长，别记成飞行时长。

请特别注意"缩短"这两个字。它不是"忘了测"，而是"时间不够，先把关键测试砍一段"。这在工程上非常常见：进度一紧，第一个被砍的往往是测试；测试里第一个被砍的，往往是那个最花时间、最不"出彩"的端到端整体测试。而且理由听起来还挺合理——"每个小单元我们都测过了呀"。可是把所有小单元各测一遍，加起来不等于整体跑一遍。就像一部电影，每个镜头单独看都没问题，但没有人把整部片子连着放一遍，剪出来的节奏、音画同步、字幕时间轴，照样能出大问题。

课件在这一页同样标了四个字：集成测试不足。上一页是"没合起来测"，这一页是"合不起来测"。上一页偏组织之间的接口没测，这一页偏过程与过程之间的衔接没测——分段测试覆盖的只是每一段内部的逻辑，段与段之间的状态传递、时序、异常恢复，全都没人管。

再往深一层想，这是一个决策问题，不是技术问题。"走捷径"是人在进度压力下做出的取舍。所以课件用了很重的说法："灾难再次发生"。火星飞船的教训早就写进教科书了，为什么还会再来一次？因为每一个项目都有它自己的"这次不一样"。而测试人员的专业价值，恰恰就是在大家都说"这次来不及了"的时候，把风险讲清楚、把取舍记录清楚。

> **板书：**波音 星际客机 → 关键测试被"缩短" → 把飞行过程拆成小单元分别测 → 未做端到端集成测试 → 缺失：25 小时整体测试（据课件数据）→ 本质：进度压力下的决策失误 → 集成测试不足

> **提问：**假设你是这个项目的测试负责人，只剩三天，团队说"整体测试要 25 小时，来不及了，能不能先跳过"，你怎么回答？

> **预设回答：**① "必须测，风险太大"——态度对，但光表态不够。追问：你怎么让项目经理接受？要能说出"不测的后果"和"替代方案"，比如先跑关键路径的短时冒烟版本。② "听项目经理的，出了事他负责"——这是典型的错误回答：测试人员要把风险如实呈现、把决策记录在案，而不是把责任一推了之。③ "只测最

关键的几段"——这是比较专业的回答：范围可以裁剪，但必须显式声明裁掉了什么、风险有多大、由谁签字确认。

易错点 / 考点：常考判断："为赶进度缩短测试时间属于合理的进度管理"——错；正确的说法是"在风险可控、显式记录的前提下裁剪测试范围"。第二个混淆点是把"25 小时整体测试"记成飞行时长。口诀：**分段测完 ≠ 端到端跑通；砍测试不能砍接口。**

【思政落点 2】波音这个案例讲的不是技术不够，而是把质量控制让位给了进度。可以让学生代入一个具体处境：如果你是那个被要求"先跳过整体测试"的工程师，你会怎么做？正确的做法既不是硬顶，也不是随大流，而是把风险讲清楚、把替代方案给出来、把决定留痕。这就是工程人员的质量责任——**按时交付很重要，但交付一个自己知道有问题、却没说出来的东西，不可原谅。**

过渡：上面两个案例的根子都在"没合起来测"。下一页换一种味道——代码没写错、测试也没少做，只是一条指令发错了，45 分钟，几亿美元。

【1.17】错误指令造成骑士资本集团损失 4.4 亿美元（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页的主角是美国的骑士资本集团。课件给的时间线非常精确：2012 年 8 月 1 日上午 9 点，纽约证券交易所开盘交易，骑士资本的第一位散户投资者发出了买卖其投资头寸的指令；仅仅 45 分钟后，骑士资本的服务器就执行了 400 万笔交易，使公司损失了 4.6 亿美元，濒临破产。45 分钟、400 万笔、4.6 亿美元，这三个数字是这一页的核心。另外提醒一句：这一页的标题写的是"4.4 亿美元"，正文写的是"4.6 亿美元"，两个数字不一致，我们以正文的 4.6 亿为准，后面讲这个案例时也统一按正文口径。

45 分钟能发生什么？我们平时点个外卖，45 分钟刚够送达。可是在一家做高频交易的机构那里，45 分钟足够让服务器把几亿美元打进市场，让一家公司走到破产边缘。这就是软件缺陷在金融领域的放大效应：它的运行速度不是人按鼠标的速度，而是机器每秒成千上万笔的速度，缺陷一旦上线，损失是按秒计的。

课件给这一页标的缺陷类型是"功能容错性问题"。什么叫容错？通俗说，就是"出了意外，系统还能不能兜住"。这一页的起因描述是"错误指令"——也就是说，系统面对一条非预期的、异常的指令时，没有拒绝、没有拦截、没有熔断，而是照单执行，还执行了 400 万次。生活里的类比很好找：家里的电路如果只有一个总开关、没有保险丝，短路一次就可能烧掉整栋楼。保险丝平时一点用都没有，出事那一秒，它替你保住了整栋楼。

所以这一页要带走两个词：**容错**和**止损**。容错是让系统在异常输入、异常状态下不崩溃、不放大错误；止损是万一真的错了，能不能快速停下来。做测试的时候，我们不仅要测"正常流程走得通吗"，更要测"给它异常输入，它会怎么样"——这就是后面要讲的异常测试、边界测试、失效恢复测试的来由。

板书：骑士资本 2012-08-01 09:00 开盘 → 45 分钟 → 400 万笔交易 → 4.6 亿美元（据课件数据；标题写 4.4 亿，以正文为准）→ 濒临破产 → 缺陷类型：功能容错性 → 关键词：熔断 · 止损 · 异常输入

提问："错误指令"是操作者发出来的，为什么这笔账要算在软件缺陷头上？

预设回答：① "因为软件应该能识别出这是错误指令"——对，这正是容错的核心：软件不能假设输入永远正确。② "是客户操作失误，跟软件无关"——要纠正：操作失误是常态而不是例外，软件设计的前提就是"人会犯错"，所以要有校验、限额、二次确认和熔断。③ "不知道算谁的责任"——顺势引导学生区分两件事：责任可能在人，但缺陷是系统缺少对异常的保护，二者不能相互抵消。

易错点 / 考点：一是数字，45 分钟、400 万笔、4.6 亿美元，并注意标题与正文不一致。二是概念，容错不等于可靠：可靠性讲"不出故障"，容错讲"出了故障也不造成严重后果"。三是案例题常问"这个案例暴露了哪类测试缺失"，答案指向异常/容错测试以及上线前的风控验证。口诀：**正常流程测不出胆子，异常输入才见真章。**

【思政落点 3】骑士资本这类案例里，真正的风险往往不是"没人知道有毛病"，而是"有人知道了但心存侥幸，觉得不会那么巧"。测试人员最可贵的一种品质，就是不做这种侥幸者：缺陷等级怎么定就怎么报，风险多大就说多大，不粉饰、不缩水。**如实记录、如实上报、不隐瞒不利结果，这是诚信，也是这个职业能被信任的根基。**

过渡：2012 年的券商如此，2017 年的云计算巨头也没能幸免——下一页，AWS 宕机整整 4 个小时。

【1.18】AWS 宕机整整 4 个小时（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

前面几个案例，都是某一家公司自己出事。这一页不一样，出事的是一家"替别人保管系统"的公司——亚马逊云，也就是 AWS。课件写得很清楚：2017 年 3 月 2 号，亚马逊云出现严重的宕机；亚马逊云之前也出现过宕机，但一般在一个小时之内解决问题，而这次非常严重，宕机整整四个小时。缺陷类型标注是"性能、稳定性问题"。

这里要请同学们注意，出问题的是云上的基础存储服务。说白了，它就是云上的"硬盘"——很多网站、APP 的

图片、视频、备份数据都放在里面。它一停，不是一家网站打不开，而是很多网站图片显示不出来、很多APP 直接报错。这就是云时代的放大效应：以前一个软件缺陷影响一个用户、一家公司；现在一个底层服务的缺陷，会沿着依赖链传到成千上万个系统上，这也叫级联失效。

再看时间。课件说，以前一小时能解决，这次四个小时。为什么这次这么久？因为它出问题的地方太底层了——修一个跑在最上面的应用很快，可当你发现是地基出了问题，能做的事情就非常有限，只能等它慢慢恢复。这是稳定性问题的一个典型特征：**越底层的组件，故障影响面越大、恢复时间越长。**

那我们做测试的同学从这里要学到什么？第一，性能与稳定性是独立的测试维度，跟功能测试不是一回事：功能测试回答"这个按钮点了有没有反应"，性能稳定性测试回答"一万人同时点会怎么样、连续跑一整天会怎么样"。第二，测试要关注依赖：你的系统再健壮，如果它依赖的那个云服务挂了，你也会挂。所以现代测试里有依赖分析和故障注入这类专门的工作，就是去模拟依赖失效，看系统还能不能撑住。

板书：AWS · 2017-03-02 → 宕机 4 小时（以往 1 小时内解决，据课件数据）→ 缺陷类型：性能 / 稳定性 → 云时代放大效应：级联失效 → 越底层，影响面越大、恢复越慢

提问：一个"存储服务"坏掉，为什么会让很多看起来毫不相关的网站一起打不开？

预设回答：① "因为大家的数据都存在里面"——对，这是依赖关系。追问：如果你的系统要抗住这种风险，测试阶段应该做什么？答：做依赖分析、故障注入，验证降级预案。② "说明他们测试做得不好"——要引导得更准确：更可能是稳定性与容灾能力的不足，而不只是"没测"；四小时里他们要处理恢复流程、依赖启动顺序、流量限制，这些都需要提前演练。③ "云计算就是不可靠"——纠正：云不是不可靠，而是把风险集中了，管理风险的前提是承认依赖、度量依赖。

易错点 / 考点：数字记 2017 年 3 月 2 日、4 小时、以往 1 小时内。概念上，性能问题与稳定性问题常被混为一谈——性能看"快不快"，稳定性看"长时间、高负载下稳不稳"。案例题常考"云服务故障对下游系统的影响属于什么风险"，答依赖风险（级联失效）。

过渡：宕机影响的是"用不了"；下一页 Uber 这个案例，影响的是更敏感的东西——你的隐私。

【1.19】Uber 泄漏个人隐私，导致用户要求赔偿 3 亿多元（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲的是隐私。课件的原文是：2017 年 2 月，根据法国《费加罗报》报道，一名法国商人起诉 Uber，要求该公司赔偿 4500 万欧元；按当时汇率 7.4 计算，合计人民币 3.33 亿，以弥补隐私漏洞对自己婚姻造成

的伤害。他的理由是，Uber 专车 App 中存在这样的安全性漏洞。标题说"3 亿多元"，跟正文的 3.33 亿是一致的。

我们先把数字换算一遍，大家跟着算：4500 万欧元乘以 7.4，等于 3.33 亿人民币。这个数字很有冲击力，但更值得琢磨的是一个细节——**索赔的理由不是"我的钱被偷了"，而是"我的隐私漏洞伤害了我的婚姻"。** 这说明泄露出去的绝不是一串没用的字符，而是能定位一个人、能还原一个人生活轨迹的信息：他什么时候叫车、从哪里上车、到哪里下车、多久去一次某个地址。这些数据拼起来，就是一张生活地图。

什么叫隐私保护问题？一句话：系统收集了用户的个人信息，却没有能力保证这些信息只被该看的人、在该看的时候看到。它属于安全性缺陷的一个分支。生活里的类比就是酒店的房卡：你住 301，房卡只能开 301；如果前台把一张万能卡随手放在大堂，问题不在于"卡坏了"，而在于权限管理失效了。这一页的问题，就是这把万能卡没管好。

从测试角度看，安全性测试跟前面讲的性能、功能都不太一样。它要回答三个问题：谁能看到什么、谁能改什么、谁能以别人的身份做什么。这类测试要求一种特别的思维方式——"攻击者思维"，站在坏人的角度想：如果我想拿到别人的数据，我会从哪里入手？后面讲安全性测试时，我们会专门讲越权访问、权限校验、传输加密、日志与审计这些检查点。还要提醒一句：隐私问题的后果往往是慢性的，泄露那一天没感觉，某一天你才发现有人知道你的全部行踪。

板书： Uber · 2017-02 · 法国《费加罗报》 → 索赔 4500 万欧元 ≈ 3.33 亿人民币（汇率 7.4，据课件数据）→ 缺陷类型：安全性 / 隐私保护 → 核心问题：权限与访问控制 → 攻击者思维：看得到 · 改得动 · 冒充得了

提问： 一个打车 App 记录你的上车点和下车点，为什么这可能构成隐私漏洞？

预设回答： ① "因为这些数据能暴露我的住址"——好回答，追问：除了住址还能推出什么？工作地、作息规律、社交关系、甚至健康状况。② "只要有密码就安全了"——典型误区。密码只解决"是不是本人登录"，解决不了后台有没有越权看数据、接口有没有绕过校验。③ "数据在我手机里，别人拿不到"——提醒他：这些数据实际存在服务端，还要经过网络传输，服务端、传输、终端三个环节都可能出问题。

易错点 / 考点： 换算必考——4500 万欧元 × 7.4 ≈ 3.33 亿人民币。概念上，安全性缺陷不等于功能缺陷，它的特点是"功能都正常，但数据被不该看的人看到了"。口诀：**看得到、改得动、冒充得了——三个方向查安全。**

过渡： 大公司出大事故，普通人的遭遇往往更具体。下一页是 2019 年国庆假期，很多人订了酒店，却住不

> 进去。

【1.20】预定的酒店住不进去、露宿街头（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是一个发生在我们身边的故事。课件写的是：正值 2019 年国庆假期出行高峰期，通过某知名旅行网预定酒店的不少网友遭遇"人在囧途"——到达酒店却无法入住，已支付的订单却显示未支付或者订单不存在；与此同时，客服电话打不通，想退房也退不掉；有些旅客想重新预定，在这样的出行高峰期又很难订到房。最后的结果，就是标题里写的：预定的酒店住不进去、露宿街头。

这一页课件没有标注缺陷类型，所以下面这个判断是我们根据现象做的分析，不是课件原文：从"已支付的订单却显示未支付或订单不存在"来看，这是一个典型的功能与数据一致性缺陷——钱付掉了，订单状态没同步；再叠加客服打不通、退不掉，就是可用性和服务响应能力的问题。

我想请大家特别体会"假期高峰期"这四个字。为什么这类缺陷平时不出现，一到高峰期就爆发？因为高峰期的本质是**并发**：平时每分钟十个人下单，服务器轻轻松松；高峰期每分钟几万人下单，系统在压力下就暴露出一堆平时藏着的毛病——请求超时不处理、消息丢了、状态写了一半、重试把订单写重了。这跟上页讲的稳定性问题是同一家人，只不过这次被击穿的是业务数据的一致性。

再算一下这次事故的"代价"。课件没有给乘客的经济损失数字，我们也不编。但可以算另外一种账：一家平台，用户在寒夜里站在酒店门口却住不进去，第二天他还会用这个平台吗？**这就是信任成本。** 功能缺陷可以靠一个补丁修好，用户对你的信任一旦掉了，补丁修不回来。生活里的类比就是餐厅订位：你打电话订了位，到店里说没有你的记录，经理也找不到人核实，旁边又没有空桌——这不只是"系统出故障"，这是让顾客站在门口。所以测试用例不能只在"一个人、一条数据、正常流程"下设计，必须要有高并发、状态流转、异常中断这些场景。

> **板书：** 2019 国庆 · 某知名旅行网 → 已支付订单显示"未支付 / 不存在" → 无法入住、退款无门、客服不通 → 判断（非课件原文）：功能与数据一致性缺陷 + 可用性不足 → 触发条件：高峰期并发 → 代价：信任成本

> **提问：** 为什么这类"订了房住不进去"的缺陷，平时的测试很难发现？

> **预设回答：** ① "因为平时没那么多人用"——对，这是并发场景缺失。追问：测试时能不能造出高并发？答：能，压力测试工具就是干这个的。② "因为订单系统太复杂"——部分成立，要顺势引导到可测性：越复杂越

要把状态机拆开，把"待支付→已支付→已确认→已完成"每一次状态转移都测到。③ 典型的错误回答："这只是小概率事件，不用管"——要驳回去：涉及钱和承诺的业务，小概率乘以大用户量就等于必然发生。

易错点 / 考点：注意课件这一页**没有**给缺陷类型标签，答题时不要一句"性能问题"就完事，更准确的说法是数据一致性问题叠加可用性问题。考点常以"给现象判断缺陷类型"的形式出现。口诀：**钱付了、单没了——先查状态同步，再查并发。**

【思政落点 4】这个故事没什么高科技含量，但它最能让学生理解"用户视角"。对我们来说，那只是一个订单状态字段错了；对那位旅客来说，那是全家人半夜拖着行李站在街头。做测试要有一种把用户请到心里的能力：**你在设计用例时多想的那个异常场景，可能就是某个真实用户免于受困的那一晚。** 质量责任不是抽象口号，它落在每一条用例的细节里。

过渡：讲到这里，我们已经看了六个案例。下一页，课件把镜头拉开，告诉我们——这样的悲剧还有很多。

【1.21】更多的悲剧（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页的标题只有四个字："更多的悲剧"。课件给了两个例子，我念一遍，请大家把数字记准。

第一个是放射性治疗仪 Therac-25。它的软件存在缺陷，导致几个癌症病人受到非常严重的过量放射性治疗，其中 4 个人因此死亡。关键在"过量"两个字——放疗设备的剂量是精确控制的，软件出错让它给出的剂量远远超出安全范围。这不是"仪器不好用"，这是把治疗变成了伤害。

第二个是爱国者导弹防御系统。课件写的是：当这个防御系统的时钟累计运行超过 14 小时之后，系统的跟踪系统就不准确了，从而导致拦截伊拉克"飞毛腿"导弹的几次失败，其中一枚在沙特阿拉伯的多哈爆炸的"飞毛腿"导弹造成 28 名美国士兵死亡。这一页有两个数字必须记住：**14 小时**和 **28 人**。

我们把这两个案例分开看，它们各代表一种非常典型、也非常可怕的缺陷。Therac-25 代表的是"安全关键系统中的软件缺陷"：在这类系统里，软件不是辅助工具，软件直接控制物理世界——剂量、速度、开关。软件缺陷不再是"少算了几毛钱"，而是直接指向人的身体。所以在医疗、航空、核电、汽车这类行业，测试的标准和强度跟普通应用完全不是一个量级，通常还有专门的标准和认证要求。

爱国者导弹代表的是"跟时间有关的缺陷"，也就是我们后面会专门讲的时间/日期类缺陷。为什么运行 14 小时之后才不准？因为系统内部用了跟时间相关的量，运行时间一长，精度就丢掉了，跟踪随之出现偏差。这类缺陷最阴险的地方在于：**它不在刚上线的时候出现，而是在系统跑了很久之后才出现。** 如果我们的测试只

跑十分钟就收工，这种缺陷永远发现不了。这两件事合起来，给了这门课一个最硬的理由：软件测试不是锦上添花的工序，在某些行业它就是人命关天的最后一道闸门。

板书： Therac-25 放疗仪 → 过量辐射 → **4 人**死亡（据课件数据） ｜ 爱国者导弹 → 累计运行 **> 14 小时**后跟踪不准 → 拦截飞毛腿失败 → **28 名**美国士兵死亡（据课件数据）→ 两类典型缺陷：安全关键系统缺陷 · 时间/精度相关缺陷

提问： 爱国者导弹的缺陷"跑 14 小时才出现"，这说明测试设计上缺了什么？

预设回答： ① "缺了长时间运行的测试"——准确，这叫耐久性/稳定性测试。追问：那要跑多久才够？答：按业务可能的最长连续运行时间来定，不是拍脑袋，要看真实的连续工作场景。② "应该一发现就重启系统"——这是运维上的权宜之计，不是测试；而且这种"重启掩盖"会让缺陷更难被发现，可以顺势讨论"用临时办法遮住缺陷"的危害。③ "这是硬件问题吧"——纠正：时钟累计运行导致的精度丢失与时间处理有关，账要算在软件上。

易错点 / 考点： 必背数字——Therac-25 致 **4 人**死亡；爱国者系统累计运行 **14 小时**后跟踪不准，造成 **28 名**美国士兵死亡。常见混淆：把 Therac-25 说成"操作人员失误"，课件明确写的是软件存在缺陷。判断题高频："只有功能错误才会造成严重后果"——错，时间、精度、安全类缺陷同样致命。口诀：**剂量、时钟，两条命门。**

【思政落点 5】 讲到 28 名士兵、4 位病人的时候，我们要停一下。这些数字背后，是一个个再也回不来的家庭。软件测试在有些行业是一份"沉默的守护"——做得好，没有人会记得你；做不好，代价由别人承担。要让学生明白：**我们将来提交的每一份测试报告，都可能成为某条产品线的安全底线。** 认真，是这个职业最高级的品质。

过渡： 六个案例、至少几十条人命、十几亿美元的损失，都指向同一个问题——我们为什么要做软件测试？下一页，课件正式给出这一节的标题。

【1.22】1.2 为什么要进行软件测试？（1.5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

前面六页案例看下来，我相信大家心里已经有答案了。不过我们还是要正式地把这个问题提出来：**1.2，为什么要进行软件测试？** 这一页是本节的开场，课件上没有堆结论，只留了一个标题，等着我们往下回答。

在往下走之前，我先交代一下我们这门课"讲义"的用法，课件的备注里也专门强调了这一点：讲义不是语音

讲解的辅助材料，它本身就是学习的主体。也就是说，你课后复习靠的是这份讲义，而不是靠回忆我上课说了什么。所以每一页我都会尽量把话写完整，你单独看讲义也能学得下去。

我们这门课讲每一个学习要点，都会按这样一条线索走：第一，先讲概念——这个知识点是什么，在工作中什么场景会用到它；第二，再展开知识和技术——要胜任这个能力，你需要掌握什么，我会讲细；第三，举例子，用一个案例或一个演示帮你理解怎么用；第四，分享经验，讲讲理论和实际工作的差别在哪里，这些往往是踩过坑才知道的；第五，给你们学习建议；第六，如果有对应的工具和模板，我也会一并给出来。

那么这一节到底要解决什么问题？就是一句话：把"测试"从一个"工序"变成一种"投资判断"。学完这一节，你们要能回答三件事：第一，测试为什么是保证质量的必要手段；第二，不做测试的代价是怎么一层层放大的；第三，测试这笔钱为什么值得花。

所以接下来这一节，我们就要正面回答"为什么要测试"。请大家带着前面那几个案例去想：如果当初多做一道测试，哪些悲剧是可以避免的？

板书： 1.2 为什么要进行软件测试？ → 讲义＝学习主体 → 学习六步：概念 → 知识/技术 → 示例或演示 → 经验分享 → 学习建议 → 实用工具

提问： 请你用一句话回答——为什么必须做软件测试？（不许用"为了找 BUG"这一句）

预设回答： ① "为了保证质量"——直接命中下一节课件的第一句话，可以表扬并追问：质量和缺陷是什么关系？② "为了发现缺陷、把缺陷清出去"——也对，追问：那发现了不修会怎样？（引出劣质成本）③ "因为软件一定会出错，用户会骂人"——很实在，把它接到"用户期望"上：质量就是满足用户规定和隐含期望的程度。对"为了交差""公司规定要测"这类回答，要明确纠偏：那是形式，不是目的。

易错点 / 考点： 这一页本身是过渡页，不产考点，但"讲义是学习主体"这句话要记住——课后复习以讲义为准，考试范围也以讲义和教材为界。另一个提醒：不要用"为了找 BUG"这种手段性表述来回答目的性问题。

过渡： 好，下一页，我们把这个问题正式回答一遍。

【1.23】为什么要进行软件测试？（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

我们正式回答这个问题。课件的答案很朴实，第一句话就是："答案很简单，就是为了保证软件质量。"注意"简单"这两个字——测试的目的不是"证明程序没错"，更不是"让开发难看"，而是为了保证质量。

第二句话是：软件总存在缺陷。只有通过测试，才可以发现软件缺陷；也只有发现了缺陷，才可以把软件缺陷从软件产品或软件系统中清理出去。请大家把这条逻辑链捋一下，它有三个环节：**软件一定有缺陷 → 测试才能发现缺陷 → 发现之后才能清除缺陷。**这条链子断在任何一环，结果都一样：缺陷留在产品里。如果不承认"软件总存在缺陷"，就会觉得测试是多此一举；如果发现了缺陷却不清理，测试就成了走过场。

第三句话讲的是成本：软件中存在的缺陷给我们带来的损失是巨大的，软件测试是软件质量保证的关键步骤；测试作为一种"预防和评估成本"的投入，从而降低缺陷造成的劣质成本。这里出现了三个成本概念，是本章的一个重点，我把它写在黑板上：**预防成本和评估成本，是我们主动花的钱；劣质成本，是缺陷漏出去之后被迫花的钱**——召回、赔偿、返工、客户流失，全算劣质成本。前面几页案例里那些几亿美元的损失，全都是劣质成本。测试的逻辑就是：花一笔小小的预防与评估成本，去换掉一大笔劣质成本。

那为什么不干脆不测、把钱省下来？因为这两笔钱不对等。业界的经验规律是：缺陷发现得越晚，修复它的代价就越高。为什么？因为越晚发现，要连带改动的东西越多——代码写完了、集成进去了、文档写了、客户培训做了，甚至已经上线了；这时候改一个缺陷，等于返工一整套东西。

课件最后一句是：软件测试在产品开发中占据着相当重要的位置，这也是软件行业几十年的实践所证明的一个道理。

> **板书：**为什么测试？→ 保证质量 → 软件总存在缺陷 → 测才能发现 → 发现才能清除 → 缺陷损失巨大 → 测试＝预防成本＋评估成本 → 降低劣质成本 → 几十年工程实践证明

提问："测试是一种成本"——那是不是测试投入越多越好？

> **预设回答：**① "越多越好，测到没有缺陷为止"——要引导：测试有边际效益递减，无限测会把项目拖死；正确说法是在质量目标与成本之间找平衡。② "投太多不划算"——对了一半，追问：那怎么判断该投多少？答：看缺陷后果的严重程度、看质量目标、看风险等级。③ "看项目大小"——补充：还要看行业，医疗、航空的测试强度远高于普通应用，因为他们漏出去的缺陷代价完全不同。

易错点 / 考点：三类成本的名字必须记准——预防成本、评估成本、劣质成本，测试属于前两类的投入。这是选择题高频点。案例分析常给"某公司为省测试费直接上线，结果赔偿若干万元"，问赔偿属于哪类成本——答劣质成本。口诀：**预防＋评估是投资，劣质成本是学费。**

> **过渡：**既然测试是为了保证质量，那"软件测试"这件事本身，到底该怎么定义？我们进入 1.3 节。

【1.24】1.3 什么是软件测试？（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

接下来这一节，题目叫"什么是软件测试"。别看只有五个字，这是本章、也是这门课的地基。如果连"什么是测试"都说不清楚，后面讲用例设计、讲测试级别、讲自动化工具，全都是空中楼阁。

这一节我们分四个小节走，课件和教材的安排是这样：第一小节 1.3.1 软件测试学科的形成，教材在第 11 页；第二小节 1.3.2 正反两方面的争辩，教材在第 12 页；第三小节 1.3.3 软件测试的定义，教材在第 13 页；第四小节 1.3.4 软件测试的其它观点。这四个小节是一条线：先看历史，再看争论，然后才落定义，最后补充其它视角。

这个顺序本身就有讲究。为什么不直接甩一个定义出来？因为软件测试的定义不是天上掉下来的，它是几十年的实践和争论沉淀出来的。你先看争论，才会明白为什么定义里要有某些限定；你先看历史，才会明白为什么测试从"程序写完之后的事"变成了"贯穿开发全过程的事"。所以我希望大家听这一节的时候不要急着抄定义，先跟着走一遍"为什么会有不同的说法"。

这一节学完，你们应该能回答三个问题：测试是什么？测试和调试有什么区别？"测试是为了证明程序是正确的"这句话，错在哪里？

最后提醒一句学习方式：这一节的概念会成为后面所有章节的"公理"。比如后面讲测试用例设计，为什么要设计异常用例？根子就在"测试是发现错误"这个认识上；再比如讲测试级别，为什么测试要贯穿单元、集成、系统、验收？根子就在"测试有不同视角和目标"。所以这一节看起来最简单，实际上最不能打折扣。

板书：1.3 什么是软件测试？→ 1.3.1 学科的形成（教材 p11）→ 1.3.2 正反两方面的争辩（p12）→ 1.3.3 定义（p13）→ 1.3.4 其它观点 → 路线：历史 → 争论 → 定义 → 补充

提问：凭直觉说一句，你认为"软件测试"是什么？

预设回答：① "就是找 BUG"——最常见。点评：方向不错，但"找 BUG"是手段不是目的，追问一句"找到 BUG 之后呢？"，把话题引到"提供质量信息、清除缺陷"。② "就是运行程序看看能不能用"——这属于动态测试里的功能验证；提醒他测试还包括静态测试，不运行程序也能测，比如需求评审、代码走查。③ "测试就是保证软件没有错"——典型错误，要坚决纠正：测试不能证明软件没有错，只能发现其中存在的错。

易错点 / 考点：本节最大的误区是"测试是为了证明程序是对的"，先把这个误区立住，后面正式定义时再来拆。另外记住四小节与教材页码的对应（11/12/13）。口诀：**测试是发现错误，不是证明正确。**

> **过渡：**那我们就从第一小节开始，看看"软件测试"这门学科是怎么一步步形成的。

【1.25】1.3.1 软件测试学科的形成（1.5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是把 1.3 节的四个小节摊开，先给大家一张地图。我们刚才已经点过一次，这里再明确一遍：1.3.1 软件测试学科的形成，1.3.2 正反两方面的争辩，1.3.3 软件测试的定义，1.3.4 软件测试的其它观点。

这四块内容，知识密度是递增的，但认知难度是先降后升的，我解释一下。1.3.1 讲学科怎么形成，是故事，最好懂，可它决定了你后面所有判断的底线；1.3.2 讲正反两方面的争辩，也就是正向思维和反向思维的对立，这块最容易被忽略，但它恰恰最有用——因为它决定了你测的时候是"顺着验证"还是"故意找茬"；1.3.3 给出定义，这是要背、要考、要能准确表述的；1.3.4 是其它观点，告诉你定义不只一家之言，帮你建立更完整的认识。

所以我给大家一个学习建议：这一节的四小节，不要只背 1.3.3 那一句定义。真到了考试或者面试问你"你怎么理解软件测试"，光有一句定义是撑不住场面的；你要能说出测试的历史演变、正反两种思维、以及不同观点的差异，这才叫理解。

另外提醒一句，页码我也写在这里了：教材上分别是第 11 页、第 12 页、第 13 页。课后读完这一节，第一次作业里"用自己的话写出软件测试的定义"那道题，你们就能答得更像样。

顺便说一下这四个小节在整门课里的位置：它们是"地图的图例"。后面几周我们讲的测试级别、测试类型、静态与动态、黑盒与白盒，全都可以挂到这张地图上。所以现在多花几分钟认清路，后面就少走弯路。

> **板书：**1.3 学习地图 → 形成（p11）→ 争辩（p12）→ 定义（p13）→ 其它观点｜学习建议：不要只背定义，要能说出演变、思维与观点差异

> **提问：**这四个小节里，你觉得哪一个最可能成为考点？为什么？

> **预设回答：**① "定义，因为要背"——承认它是考点，但追问：如果只考定义，教材为什么还要花两节讲历史和争辩？引导学生理解知识之间的支撑关系。② "争辩，因为难"——好回答，可以表扬：正反思维是测试思维方式的根，案例分析题经常考。③ "其它观点，因为听起来高级"——提醒他观点要多、但主线要清，回答"你如何理解软件测试"时主线仍然是定义。

> **易错点 / 考点：**四小节的顺序与教材页码（11 / 12 / 13）容易记混。记忆口诀：**先看形成（历史），再看争辩**

（思路），然后定名（定义），最后广角（其它观点）。

过渡：那我们从第一小节开始——软件测试学科是怎么形成的。

【1.26】1.3.1 软件测试学科的形成（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

我们进入 1.3.1，软件测试学科的形成。这一页课件上只有标题，但它要回答的问题很重要：测试这件事古已有之，可测试作为一个学科、一个专业，是很晚才出现的。为什么？这就跟它的形成过程有关。

这条形成过程，大致可以理解成三个阶段。第一个阶段，测试还不是一件独立的事，它藏在"调试"里面。程序写完，运行一下，看看哪里不对，改一改，再运行——这是开发人员自己做的事，没有专门的岗位，也没有系统的方法。这个阶段最大的问题是：测试的范围完全等于"开发者能想到的范围"。

第二个阶段，到了上世纪七十年代前后，业界开始把"测试"和"调试"分开来看。这里有一个理念上的大转弯：以前大家觉得测试是为了"证明程序是对的"，后来慢慢认识到，测试的本意是"发现程序里的错误"。教材 1.3.1 节里讲了这段演变的过程和相关的人物与年代，具体以教材为准，我这里只讲主线。这个转弯为什么重要？因为它决定了你测试的姿势——如果你想着"证明它是对的"，你就会顺着写程序的人的路子走，很容易漏；如果你想着"找出它的错"，你就会去挑边界、挑异常、挑组合。

第三个阶段，是测试的工程化和规范化。有了独立的测试角色、独立的测试团队、专门的流程和标准，测试从一门"手艺"变成了一门"工程"。今天我们看到的各种测试级别、测试类型、测试标准与认证，都是这个阶段的产物。再往前一步，就是我们当下说的"测试左移"和"全程测试"——测试不再是开发完成之后的最后一道工序，而是从需求阶段就开始介入。这个概念后面还会专门讲。

所以这一节给我们的整体印象是：软件测试不是某个人拍脑袋发明的，它是被一次次事故、一次次返工逼出来的一个专业。

板书：1.3.1 学科的形成 → ① 藏在调试里（开发自测，范围＝开发者的想象）→ ② 与调试分离（从"证明正确"到"发现错误"）→ ③ 工程化 / 规范化（独立团队 · 流程 · 标准）→ ④ 当代：测试左移、全程测试（详见教材 p11）

提问：为什么早期的"测试"会藏在调试里，而不是一开始就独立出来？

预设回答：① "因为那时候程序小，一个人就能搞定"——对，规模决定分工，这是最根本的原因。② "因为

大家觉得测试不重要"——部分成立，但可以引导得更准确：不是不重要，而是当时没有人意识到它有独立的价值。③ "因为那时候没有测试工具"——有道理，工具也是被需求催生出来的，可以顺带提一句"先有分工，才有工具"。

易错点 / 考点：这一节最容易考的是"测试与调试的区别"（下一节展开）：调试是发现错误之后定位并修复的过程，属于开发活动；测试是发现错误、评价质量的过程。另一个易错点是仍把"测试是为了证明程序正确"当成对的。具体的年代、人物与事件以教材 1.3.1 节（教材 p11）为准。

过渡：学科形成之后，关于"测试到底是干什么的"，业界的争论一直没有停。下一页我们就进入 1.3.2——正反两方面的争辩。

【1.27】从狭义的软件测试到广义的软件测试（7 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

同学们，前面我们讲了软件缺陷带来的惨痛代价。从这一页开始，我们要回答一个更基础的问题：软件测试到底测什么？先看一个很常见的误解——很多人一说测试，脑子里出现的画面就是：程序写完了，运行起来点一点，看看有没有报错。这就是最狭义的软件测试，一句话概括：软件测试等于程序测试。但请注意，软件不只是可执行的程序。按照软件工程的观点，需求、设计、代码这三样东西，都属于软件的组成部分。需求说清楚要做什么，设计决定怎么做，代码才是最后的实现。既然这三样都是软件，那它们出了问题，是不是软件缺陷？当然是。而检查这三样东西有没有问题，靠的不是把程序跑起来，靠的是评审——需求评审、设计评审、代码审查。这些评审工作有一个统一的名字，叫静态测试。

>

那么问题来了：为什么过去我们不觉得评审算测试？因为早期观念里，只有"运行程序、看到结果"才算测试。可是大家想一想，如果在需求阶段就发现"这个功能压根没写清楚"，你还需要等代码写完再跑一遍去发现吗？完全不需要。所以，当业界正式提出"动态测试"这个词的时候，其实已经从反面承认了：还有一类不需要运行程序的测试，也就是静态测试。于是软件测试的范围扩大了——软件测试既包含静态测试，也包含动态测试，这就是广义的软件测试。

>

为什么一定要强调"广义"？因为它直接决定成本。有一句话请大家记牢：测试进行得越早，成本越低；反过来，缺陷发现得越迟，研发成本越高。你早一步在评审桌上指出需求里的二义性，代价可能只是改一句话；

等系统上线了才发现，改的就是代码、接口、文档，甚至用户的信任。这笔账具体怎么算，我们看下一页的成本曲线。

板书：狭义：软件测试＝程序测试 → 广义：软件＝需求＋设计＋代码 → 对需求/设计/代码的评审＝静态测试 → 软件测试＝静态测试＋动态测试 → 测试越早，成本越低

提问：如果一个功能的需求文档里写"系统应快速响应用户请求"，这条需求本身有没有问题？它属于静态测试还是动态测试的检查对象？

预设回答：①答"没问题，写得挺清楚"——要追问：多快算快？1 秒还是 10 秒？谁来判断达标？指出"快速"是二义性需求，测试无法据此判定通过与否。②答"这是需求问题，应该让开发去改"——要强调这正是测试人员的活：需求评审就是静态测试的战场，测试要在需求阶段就把不可测的词挑出来。③答"属于动态测试"——纠偏：需求文档不运行程序，属于静态测试。点评要点：能在需求阶段发现二义性，就是最便宜的缺陷修复。

易错点 / 考点：常见混淆是把"软件测试"等同于"程序测试"（狭义），漏掉对需求、设计的静态测试。考点多以判断题出现："只有运行程序才能叫测试"——错。判断三步法：一看对象（需求／设计／代码／程序）→ 二看是否运行程序（不运行即静态测试，运行即动态测试）→ 三看时机（越早越便宜）。

过渡：既然早发现能省钱，那到底能省多少？我们把成本曲线摊开来看。

【1.28】广义的软件测试可以极大地降低研发成本（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

大家看这张图。横轴是软件开发的时间轴——需求、设计、编码、测试、上线运维；纵轴是修复一个缺陷所要付出的成本。图上这条曲线，请特别注意它的形状：它不是一条斜着往上的直线，而是一条越往右越陡的曲线。这就是我们反复强调的四个字：非线性增长。

>

什么叫非线性？打个比方。装修房子的时候，你在图纸阶段跟设计师说"这面墙我想敲掉"，设计师改一张图纸，几分钟的事，成本几乎为零。等到水电都铺好了你才说要敲墙，得重新布线、重新贴砖，又花钱又费时。要是房子已经住进去三年了才想起来要敲，那要搬家具、要临时找住处，成本翻好几倍。软件是一模一样的道理，而且软件更狠，因为软件是"牵一发而动全身"的。

>

> 具体怎么个牵动法？一个缺陷在需求阶段被发现，你改的是文档上的一句话，成本就是一次讨论。到了设计阶段，你得改设计文档、改接口定义，上下游同事都要跟着对齐。到了编码阶段，你要改代码，还要改相应的单元测试。到了测试阶段才发现，代码要改、已经跑过的测试要重跑、相关模块要回归。要是等产品上线、用户已经用起来了才爆出来，那就不只是改代码的问题了：要紧急发版，要写公告，要处理用户投诉，严重的话还要赔偿、丢客户、丢口碑。这就是为什么这条曲线在后期会陡得吓人。

>

> 所以我们说，广义的软件测试之所以能极大地降低研发成本，道理就在这里：它把测试这件事从"最后一道关"变成了"贯穿始终的一条线"。需求评审、设计评审、代码审查这些静态测试，做的是最左边、最便宜的那一段；动态测试在中间和右边兜底。左边的成本压下去了，整个项目的总成本才降得下来。
>
> **板书：**缺陷发现越迟 → 修复成本越高，且非线性增长（曲线左低右陡）→ 成本构成：改文档／改设计／改代码＋回归／发版＋公告＋投诉＋赔偿 → 广义测试＝把测试铺满全过程，压住最左端

> **提问：**同样一个"登录后跳转错误"的缺陷，在需求评审时发现，和在产品上线后被用户发现，修复成本差在哪里？请至少说出三项差异。

> **预设回答：**①只说"花的钱不一样多"——要追问具体差在哪，引导说出：改动对象（文档／设计／代码／已上线系统）、连带工作（重测与回归范围）、外部影响（用户投诉、公告、口碑与赔偿）。②答"反正都要改代码，成本差不多"——典型错误，纠偏：越早发现，越可能只改文档或设计，根本不进代码。③答"上线后发现问题更严重所以贵"——肯定方向，补充"非线性"这一层：不是贵一点，而是成倍地贵，因为回归范围随系统规模扩大。点评：把三者串成一句——发现越迟，要动的东西越多、要重做的验证越多、要面对的外部人越多。

> **易错点 / 考点：**易错是把成本增长理解成线性，认为"晚发现无非多花一点时间"。考点：给一个事故情境问"该缺陷本应在哪个阶段被发现，能省下哪些成本"（案例分析）。记忆口诀：早一步在文档上改一句话，晚一步在系统里改一圈。

> **【思政落点 1】**：讲国外真实事故——Therac-25 放疗仪因软件缺陷导致患者受到过量辐射致死，骑士资本因交易系统缺陷在 45 分钟内亏损约 4.6 亿美元。这些代价背后共同的一点是：缺陷在最贵的那一段才被发现。要让学生先估算损失量级，再对照真实数据，体会"没有测试把关就交付"意味着什么。落到职业观：测试人员的质量责任不是口号，你手里那张用例表，可能对应着使用者的生命安全与公众利益。

> **过渡：**知道了成本曲线，我们再把镜头拉远，看看软件测试这门学科是怎么一步步走到今天的。

【1.29】软件测试学科的发展（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲软件测试学科的发展脉络，四个阶段、四个关键词，请大家把年份和"导向"对应起来记。

>

> 第一个阶段，1957 到 1978 年，以功能验证为导向。这个时期大家对测试的理解是：测试就是证明软件是正确的。注意这句话背后的思维方式，是正向思维——我先相信软件是对的，测试的任务是把这个"对"证明出来。所以做法通常是跑一遍功能，看看结果跟预期是否一致，一致就算通过。

>

> 第二个阶段，1978 到 1983 年，以破坏性检测为导向。这是一个观念上的转折点。人们发现，你按说明书跑一遍功能、跑通了，并不能说明软件没问题——你只是没找到问题而已。于是测试的目的变了：测试是为了找到软件中的错误。这是逆向思维：先假定软件有错，然后想尽办法把它找出来。

>

> 第三个阶段，1983 到 1987 年，以质量评估为导向。这一时期，测试的意义又被抬高了：它不只是找错，还要提供产品的评估和质量度量。也就是说，测试要回答"这个产品现在到底处于什么质量水平"——缺陷有多少、严重程度如何、哪些模块风险高。测试开始从"找 bug 的手艺"变成"提供质量信息的工作"。

>

> 第四个阶段，1988 年起，以缺陷预防为导向。测试是为了展示软件符合设计要求，发现缺陷、预防缺陷。请注意最后两个字：预防。发现缺陷是治已病，预防缺陷是治未病——通过分析缺陷的分布和模式，回头去改需求和开发的流程，让同一类缺陷不再反复出现。

>

> 四个阶段连起来，是一条很清晰的线：证明对 → 找出错 → 评估质量 → 预防缺陷。测试的地位，也从"事后挑毛病"一路升到"过程改进的引擎"。
>
> **板书：**1957–1978 功能验证（正向：证明正确）→ 1978–1983 破坏性检测（逆向：找错误）→ 1983–1987

质量评估（提供质量度量）→ 1988– 缺陷预防（发现＋预防缺陷）

提问：这四个阶段里，哪些阶段的测试目标已经超出了"找缺陷"本身？它们分别多做了什么？

预设回答：①只答"第四个，1988 年起"——部分正确，要追问：第三个阶段其实也超了，它提供的是"质量度量"，回答的是产品处于什么质量水平。②答"第二阶段最重要，因为开始找错了"——肯定其转折意义，但提醒：找错仍是"治已病"，第四个阶段才引入预防。③答"都差不多，就是找 bug"——纠偏：四个阶段的目标从"证明正确"到"预防缺陷"，思维方式和工作产出都不一样，后期还要出质量度量和缺陷模式分析。点评：答案要落在"质量评估（提供信息）"和"缺陷预防（改流程）"这两级上。

易错点 / 考点：易错是把 1957–1978 的"正向思维"与后面 Myers 的反向思维混为一谈；把"质量评估"当成"质量保证"。考点：年份与导向的匹配（选择题）、"测试目的的演变"（简答）。记忆口诀：证明—找错—度量—预防，四步走。

过渡：这是一种划分方法。同样这段历史，换一个观察角度，还能切成三个阶段。

【1.30】不同的阶段划分（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

刚才我们用"导向"把软件测试的历史切成了四段，这一页换成另一种切法，按学科成熟度切成三个阶段。两种切法讲的其实是同一段历史，只是观察角度不同，考试里两种都要认得。

>

初级阶段，1957 到 1971 年。这个阶段最典型的特征是一句话：测试通常被认为是对产品进行事后检验。什么叫事后检验？就是产品做完了，拉出来检一遍，合格就交付。它跟工厂里质检员拿卡尺量零件是一个思路。而且这个阶段缺乏有效的测试方法——没有系统的用例设计方法，没有覆盖率的说法，靠的是个人经验和直觉。所以这个阶段的测试，地位低、效果差，还经常背锅。

>

发展阶段，1972 到 1982 年。这十年里有一个标志性事件：1972 年召开了第一次关于软件测试的正式会议。为什么一次会议这么重要？因为在这之前，测试是每个程序员各自的手艺活，没有统一的术语、没有公认的方法、也没有交流的平台。有了正式会议，大家开始把测试当成一个值得研究的对象：怎么设计用例、怎么

> 评价测试是否充分、测试和开发怎么分工。这些问题被摊到桌面上讨论，软件测试才真正开始"长学问"，也促进了它的发展。

>

> 成熟阶段，1983 年到现在。标志是国际标准 Std 829-1983 的发布。这个标准把测试文档规范了下来——测试计划、测试说明、测试用例、测试报告该写什么、怎么写，都有了统一要求。有了标准，测试就从"手艺"变成了"专业"：它可以被教学、被检查、被审计、被认证。于是软件测试形成一门独立的学科和专业，成为软件工程学科中的一个重要组成部分。

>

> 三个阶段连起来：事后检验的手艺 → 有交流、有方法的研究对象 → 有标准、有分工的独立专业。这也解释了为什么今天的企业会设专门的测试部门、测试岗位和测试职级。
>
> **板书：**初级阶段 1957–1971：事后检验，缺有效方法 → 发展阶段 1972–1982：1972 年第一次软件测试正式会议，促进发展 → 成熟阶段 1983–：国际标准 Std 829-1983，独立学科与专业，软件工程重要组成部分

提问：为什么说 1972 年的第一次软件测试正式会议，是测试从"手艺"走向"学科"的关键一步？

> **预设回答：**①答"因为开会很重要"——空话，要追问：会议到底提供了什么？引导说出"统一术语、交流方法、形成研究共同体"。②答"因为这个会议发布了标准"——纠偏：Std 829-1983 是成熟阶段的标志，1983 年才发布，不要张冠李戴。③答"因为从此测试有了专门的人做"——肯定，补充：有了交流平台，个人经验才能沉淀成可传授的方法，方法才能变成标准和专业分工。点评：一次会议的价值，在于把个人经验变成公共知识。

易错点 / 考点：易错是把两种阶段划分的年份和标志事件搞混（尤其 1972 年会议与 1983 年标准），以及把初级阶段的"事后检验"误记成"缺陷预防"。考点：阶段—年份—标志事件三连线（选择／填空）。记忆三步：先定时段（1957–1971／1972–1982／1983–）→ 再定标志（事后检验／首次正式会议／Std 829-1983）→ 最后定地位（缺方法／促发展／独立学科）。

> **过渡：**历史的脉络清楚了，接下来进入 1.3.2，看一个至今还在争论的问题：测试到底是为了证明对，还是为了找出错？

【1.31】1.3.2 正反两方面的争辩（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

我们进入 1.3.2 节，正反两方面的争辩。

>

> 先解释一下这个标题。软件测试从诞生到现在，围绕"测试的目的到底是什么"，一直有两种针锋相对的观点。一种说：测试是为了证明软件能正常工作，让用户和开发团队对它建立信心。另一种说：测试是为了证明程序有错，你要是没找到错，那说明这次测试做得还不够狠。这就是所谓的正向思维和反向思维。

>

> 大家可能会问：这不就是个定义之争吗？咬文嚼字有什么意义？意义很大。因为你怎么理解测试的目的，直接决定你怎么设计用例、怎么判断测试做完了没有。

>

> 如果按正向思维，你的动作是：把需求说明书上的功能一条一条跑一遍，全都通过了，就认为测试完成、可以交付。如果按反向思维，你的动作完全不同：你要专门去想"这个功能在什么情况下会出错"——边界值、异常输入、并发、断网、极端数据，你要主动去攻击这个系统，直到想不出新的攻击方式为止。同样一个登录功能，两种思维写出来的用例，数量和质量都天差地别。

>

> 所以这一节我们不是死记概念，而是要搞清楚三件事：两种思维方式各自长什么样、各自出自谁的观点、它们的价值分别在哪里。最后我们还要落到一个更实际的结论——它们不是二选一，而是要在不同场景下配合使用。接下来两页，我们先分别认识两位代表人物。
>
> **板书：**1.3.2 正反两方面的争辩 → 正向思维：验证软件"能工作"，跑遍功能直至全部通过 → 反向思维：假定软件有错，"找出错"，攻击薄弱点直至找不出问题 → 认知决定用例怎么设计、何时算测完

提问：如果只允许你用一句话向新人解释"测试的目的"，你会站正向还是反向？为什么？

> **预设回答：**①站正向："测试就是确认功能都对"——追问：功能都通过了，能保证没有隐藏问题吗？引导认识"通过≠无错"。②站反向："测试就是找 bug"——肯定这是 Myers 的经典立场，再追问：一个查不出 bug 的测试就等于失败吗？引导到"测试的价值也在于提供质量信息"。③答"看情况"——这个答案最有前途，顺

> 势引导到下一页的结论：两种思维各有适用场景，工程上要结合使用。点评：先让学生亮明立场，再用后面的内容让他们自己修正立场。

> **易错点 / 考点：**易错是把"正向思维"理解成"积极的态度"、把"反向思维"理解成"消极的态度"——两者说的是方法论立场，不是工作态度。考点：给出一段测试行为描述，判断属于正向还是反向思维。

> **过渡：**先看正向思维的代表人物和他的原话。

【1.32】软件测试的正向思维（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是正向思维的代表，Bill Hetzel 博士。请大家把他的名字和"正向思维"绑在一起记，这是考点。

>

> 先看他最核心的一句话：软件测试就是为程序或系统能够按预期设想运行而建立信心的过程。请注意这句话的落点——"建立信心"。在 Hetzel 看来，测试的产出不只是"发现了几个问题"，更是"让相关的人相信这个系统能按预期跑起来"。如果测试做完了，大家对这个系统能不能上线还是心里没底，那测试就没达到目的。

>

> 他还有一句更书面、更像定义的表述：软件测试是一系列活动，以评价一个程序或系统的特性或能力，并确定是否达到预期的结果。这句话里有三个动作，请大家拆开来记：第一，它是"一系列活动"，不是随手点几下；第二，它的对象是"特性或能力"，比如性能、可靠性、易用性；第三，它的结论是"是否达到预期的结果"——也就是说，测试要给出一个有依据的判断。

>

> 接着看第三句：测试是为了验证软件是否符合用户需求，即验证软件产品是否能正常工作。这句话把"预期"的来源点清楚了——预期不是开发人员自己拍脑袋定的，而是用户需求。所以正向思维的本质，是拿需求当尺子去量产品。

>

> 我们用一个生活场景来体会。你去餐厅点了一份牛排，要求七分熟。菜端上来，你切开看一眼，确实是七分熟，于是你满意了。这个"看一眼、确认符合要求"的过程，就是正向思维。它回答的问题是：该做的，做到

> 了吗?

>

> 但大家马上能想到一个问题：我看这一眼，只能证明这一块牛排是七分熟，不能证明后厨没有别的隐患——比如食材新不新鲜、有没有交叉污染。也就是说，正向思维能验证"符合要求"，但它天然不擅长发现"没想到的问题"。这个短板，就是下一页反向思维要补的。
>
> **板书：**正向思维代表：Bill Hetzel → 测试＝为"按预期运行"建立信心的过程 → 测试＝一系列活动，评价特性/能力，确定是否达到预期结果 → 测试＝验证软件是否符合用户需求 → 回答：该做的，做到了吗?

> **提问：**按 Hetzel 的说法，"测试是一系列活动以评价特性或能力并确定是否达到预期结果"。这句话里的"预期结果"应该由谁提供?

> **预设回答：**①答"开发人员"——纠偏：预期来自用户需求，开发人员自己的假设不能当尺子。②答"测试人员自己判断"——追问：测试人员的个人判断如果和需求不一致怎么办？引导到"以需求文档、规格说明为依据，需求不清就回到需求阶段解决"。③答"用户需求"——正确，追问：如果需求文档本身就写得含糊呢？正好呼应前面讲的静态测试——需求二义性要在评审阶段挑出来。点评：测试的尺子是需求，尺子本身不准，量出来的结论就没有意义。

> **易错点 / 考点：**易错是把"建立信心"理解成"测试就是走个过场让人放心"。考点：Hetzel 与 Myers 各自观点与人名的对应（选择题／连线题）；"测试是为了验证软件是否符合用户需求"出自谁的观点。判断三步法：看落点——落在"符合预期／建立信心"是正向，落在"发现错误／证伪"是反向。

> **过渡：**正向思维把测试当"确认"，反向思维则把测试当"进攻"。我们看 Myers 怎么说。

【1.33】软件测试的反向思维（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是反向思维的代表，Glenford J. Myers。如果说 Hetzel 的落点是"建立信心"，Myers 的落点就是四个字：证明有错。

>

> 先看他的第一句话，也是最有冲击力的一句：测试是为了证明程序有错，而不是证明程序无错误。按常识，

> 我们做测试不就是为了确认软件没毛病吗？Myers 说不。他的逻辑是：你把软件跑一遍没发现问题，只能说明这一次没找到，不能证明软件没有问题；而只要你能找到一个错误，你就确凿地证明了"这个程序有错"这件事。所以从逻辑上讲，测试能够证明的是"有错"，不能证明的是"无错"。

>

> 再看第二句：一个好的测试用例在于它能发现至今未发现的错误。这一句把"好用例"的标准给改了。过去我们评价用例，看的是它覆盖了哪个功能、写得规不规范；Myers 说，这些都不是关键，关键是你这条用例有没有真的抓到过问题。一条跑一万遍都通过的用例，和一条抓出严重缺陷的用例，价值完全不一样。这个观点直接催生了后来的一整套用例设计方法——等价类、边界值、错误推测，全都是为了让用例"更容易抓到错"。

>

> 第三句更彻底：一个成功的测试，是发现了至今未发现的错误的测试。注意"成功"这两个字的位置——测试的成功不在于"全部通过"，恰恰在于"逮到了问题"。这就带来一个推论：如果一轮测试下来一个缺陷都没发现，这轮测试到底算成功还是失败？按 Myers 的立场，得先怀疑是不是做得太软了。

>

> 生活里也有类似的场景：安检员一天下来什么都没查出来，我们不会觉得他工作很成功，反而会担心他是没认真查。所以反向思维的本质是：先假定系统有问题，再带着"我要把它找出来"的心态设计测试。正向和反向到底谁对？下一页我们用一张表把两者摆在一起看。
>
> **板书：**反向思维代表：Glenford J. Myers → 测试是为了证明程序有错，而不是证明程序无错误 → 好的测试用例＝能发现至今未发现的错误 → 成功的测试＝发现了至今未发现的错误 → 回答：怎么能把它弄坏？

提问：按 Myers 的观点，如果一轮测试执行完毕，一个缺陷也没有发现，你该如何评价这轮测试？

> **预设回答：**①答"说明软件质量很好"——最典型的错误，纠偏：也可能说明用例设计得太弱、只跑了主流程、没有攻击边界和异常。②答"说明这轮测试不合格"——太绝对，追问：如果用例已覆盖边界、异常、组合，仍然没发现缺陷，那它的价值在哪里？（提供质量信息，说明该范围的缺陷密度低。）③答"要看用例是怎么设计的"——最成熟的回答，肯定并总结：先看测试的充分性，再谈软件的质量。点评：把"没发现缺陷"当成需要解释的现象，而不是结论。

易错点 / 考点：易错是把"Myers 说测试是为了证明程序有错"误解成"测试人员的工作就是挑刺、和开发对立"——它说的是方法论，不是人际关系。考点：Myers 三句名言的填空与理解；"成功的测试"的判断（易出

判断题，选项常写成"测试全部通过即成功"——错）。记忆口诀：Hetzel 建信心，Myers 找错误。

过渡：两种思维摆在一起，差别就一目了然了。看这张对比表。

【1.34】认知决定着行为（9 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页的标题叫"认知决定着行为"，它把正向和反向两种思维放在一张表里对照。为什么要对照？因为这张表要证明一件事：你怎么认知测试，就直接决定你手上敲出来的用例长什么样。

>

我们先看表的上半部分。左边一路是正向思维：它的前提是验证软件正常工作；对应的行为描述是——在设计规定的环境下运行软件的所有功能，直至全部通过。请注意这里面的两个关键词："所有功能"和"全部通过"。正向思维追求的是覆盖面加通过率，它是一张清单式的做法：需求上有多少功能，我就跑多少功能，跑完都绿了，收工。右边一路是逆向思维：它的前提是假定软件有错误；对应的行为描述是——寻找容易犯错误的地方和系统的薄弱环节，试图破坏系统，直至找不出问题。

>

两句话一对比，差别非常清楚。正向是"遍历清单"，逆向是"攻击弱点"：一个按需求文档往下走，一个按风险和经验去找。正向问"这个功能正常吗"，逆向问"我怎么能把它弄坏"。

>

再往上看一点，这里还并列了两句关于测试的说法：一句是"评价一个程序或系统的特性或能力并确定是否达到预期的结果"，另一句是"测试是为发现错误而针对某个程序或系统的执行过程"。这两句话各自代言一种思维，最后都指向同一个词——软件测试。所以这一页真正想告诉大家的是：软件测试这个概念本身就同时包含两层含义，既有度量的成分，也有证伪的成分。它不是非此即彼，而是一体两面。

>

那工程上应该怎么用？我的建议是分场景：对安全攸关的系统，比如医疗设备、金融交易，要更多用逆向思维——漏掉一个缺陷的代价不可承受；对需求明确、迭代快速的普通业务系统，正向思维能保证基本功能可靠，但也不能只做正向，否则线上一定会被"没想到的路径"教育。

板书：

认知（前提）	行为（做法）	终止条件
正向：验证软件正常工作	在设计规定的环境下运行软件的所有功能	直至全部通过
逆向：假定软件有错误	寻找容易犯错误的地方和系统的薄弱环节，试图破坏系统	直至找不出问题

→ 结论：软件测试＝度量＋证伪，一体两面

提问：同样测一个"转账"功能，正向思维和逆向思维分别会写出什么样的用例？请各举一条。

预设回答：①正向举例："输入正确账号和金额 100 元，点击转账，检查余额扣减 100 元"——典型的按规格验证，肯定它。②逆向举例："余额刚好等于转账金额""金额输入 0 或负数""转账过程中断网""并发两笔同时扣款"——这正是对薄弱环节的攻击，表扬。③答"逆向就是随便乱输"——纠偏：乱输不叫逆向思维，逆向是有目的地找易错处（边界、异常、并发、状态），错误推测法也是讲方法的。追问：这两条用例哪一条更可能在真实项目里抓到线上事故？引导学生体会逆向用例的价值。

易错点 / 考点：易错是把正向思维等同于某种用例设计方法（如等价类）、把逆向思维等同于错误推测法——两两相关但不等同，正向／逆向是目的层面的立场。考点：给出行为描述判断思维类型；简答"为什么说软件测试是一体两面"。判断三步法：看前提（假设软件对／错）→ 看动作（遍历功能／攻击薄弱点）→ 看终止条件（全部通过／找不出问题）。

过渡：认知讲完了，接下来要落到最关键、也是考得最多的一件事：给软件测试下一个严谨的定义。进入 1.3.3。

【1.35】1.3.3 软件测试的定义（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

我们进入 1.3.3，软件测试的定义。

>

可能有同学会想：前面都讲了两节课了，怎么到现在才讲定义？这恰恰是有意安排的。因为"软件测试"这四个字在不同人嘴里含义差别很大——有人指跑一遍程序，有人指找 bug，有人指评估质量，还有人把它和质

> 量保证混在一起。如果不在前面把历史、把静态动态、把正反思维讲清楚，直接甩一个定义给大家，大家只会背下来，不会真正理解。

>

> 所以这一节的任务是：先看看大家心里默认的定义是什么，再用国际标准给出的定义把这个概念框住，最后用"软件测试的价值"来收口，回答"它到底给项目带来什么"。

>

> 我先给大家透个底：这一节里的定义来自 IEEE 和 ISO/IEC 29119 相关标准，都是英文原文加中文翻译。看到英文不要怕，我们逐句拆，重点是把定义里的每个要素抠出来——在什么条件下、对什么东西、做什么动作、得到什么结果。定义之所以是定义，就是因为这些要素缺一不可。以后你们写测试计划，实际上就是在回答这几个要素：条件是什么、对象是什么、动作是什么、结果怎么评价。

>

> 另外提醒一句，这一节后面还有"软件测试的价值"，那四点请大家务必记牢。它不仅是考试内容，更是你们在课程设计答辩时被问到"你这个测试有什么意义"时的标准答案。待会儿看到英文原文也别跳过——考试里定义要素的识别，就藏在那句话的每一个词里面。
>
> **板书：**1.3.3 软件测试的定义 → 为什么放到现在讲：先有历史／静态动态／正反思维 → 依据：IEEE、ISO/IEC 29119（对应 ISO/IEC 24765 词汇）→ 收口：软件测试的价值

提问：在看标准定义之前，请你用自己的话说一句"软件测试是什么"。你觉得你的说法里，缺了哪个要素？

> **预设回答：**①"软件测试就是运行程序找 bug"——缺"在特定条件下""观察或记录结果""做出评价"这几个要素。②"软件测试就是确认软件没问题"——这是正向思维的口径，缺"评价"和"发现差别"的成分。③"软件测试就是保证质量"——把测试和质量保证混了：测试提供质量信息，不"保证"质量。点评：把学生的答案先写在黑板上，讲完标准定义再回头对照，让他们看到自己漏了哪几个词，这是最有效的定义教学法。

易错点 / 考点：易错是把"软件测试"和"质量保证"当成同义词。考点：标准定义中要素的识别；"测试能不能保证质量"（判断题：不能，测试只能提供质量信息）。

> **过渡：**那我们先来看，大家心里想的那些答案，究竟哪个才算是软件测试。

【1.36】什么是软件测试？（8 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是一张图，图上的问题很直白：什么是软件测试？然后列了一串大家脑子里的答案，我们一个一个过，看看哪些对、哪些不全对。

>

第一个答案：检查、检验，Check。这个说法很朴素，就是"看一眼有没有毛病"。它对，但太笼统——检查什么？怎么检查？检查到什么程度算够？都没回答。

>

第二个：验证软件能否正常运行。这就是我们前面说的正向思维口径，它的短板也很清楚：正常运行只说明主流程没问题，异常路径、边界情况都没碰到。

>

第三个：发现问题，Detect error。这是反向思维口径，抓问题、找错误。但它只讲了测试的一个动作，没讲测试还要给结论。

>

第四个：证明是对的，Correction proof。这个说法最危险。因为软件测试在逻辑上做不到"证明软件无错"——你测过的路径是有限的，没测过的路径是无限的。所以严格讲，测试不能证明正确，只能提供证据。

>

第五个：质量评估，Quality evaluation。这个说法往前迈了一大步：测试要给出对质量的判断，比如缺陷密度、严重程度分布、风险模块。这是 1983 到 1987 年那个阶段的成果。

>

第六个：质量保证，Quality Assurance。注意，这个最容易混淆。质量保证管的是过程——规范、评审、度量、流程改进，从源头上让缺陷少产生；测试管的是产品——验证和确认，提供质量信息。测试是质量保证的重要手段之一，但测试不等于质量保证。

>

所以你看，这六个答案从"看一眼"一路走到"保证质量"，本身就是一个认知不断升级的过程。它们不是互相

否定，而是层层包含：检查是动作，找错是目标，评价是结论，而质量保证是更大的范畴。下一步，我们就用国际标准的定义，把这些零散的认知收成一个完整、严谨的说法。

板书：什么是软件测试？→ 检查/检验 Check？→ 验证能否正常运行？→ 发现问题 Detect error？→ 证明是对的 Correction proof？→ 质量评估 Quality evaluation？→ 质量保证 Quality Assurance？→ 六者层层包含，最后用标准定义收口

提问：这六个答案里，哪一个说法在逻辑上最站不住脚？为什么？

预设回答：①选"证明是对的"——正确，追问为什么：测试只能执行有限条路径，无法穷尽，所以只能提供证据，不能给出证明。②选"质量保证"——也抓住了要点，但要说清它不是"站不住脚"而是"范畴混淆"：质量保证包含测试，不能说测试就是质量保证。③答"检查、检验"——可以讨论：它不算错，只是太笼统。点评：要区分两类问题——逻辑错误（证明正确）与概念混用（测试 vs 质量保证）。

易错点 / 考点：易错是认为"测试通过"就等于"软件没有缺陷"；把质量评估与质量保证混为一谈。考点：给出一句关于测试的说法，判断是否正确并说明理由（高频简答）。记忆口诀：查→找→评→保，四层递进；但"证明正确"这一层永远做不到。

过渡：大家心里的答案过完了，下面看权威标准怎么定义。

【1.37】软件测试 IEEE/ISO29119 的定义（8 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是权威定义，出自 ISO/IEC 24765《系统和软件工程词汇》，也是 IEEE 与 ISO/IEC 29119 采用的口径。英文原文我先读一遍，再逐要素拆。

>

原文是：An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

>

中文译文是：在特定的条件下运行系统或组件，观察或记录结果，对系统或组件的某个方面做出评价。

>

> 这个定义每个词都有讲究，我们拆成四个要素。

>

> 第一，它是一项活动。这提醒我们：测试是有组织、有计划的过程，不是随手点两下；是活动，就意味着有输入、有步骤、有产出。

>

> 第二，在特定的条件下运行。这个"特定条件"太重要了。测试绝不是"随便跑跑"，你必须先想清楚：用什么数据、什么环境、什么配置、什么前置状态。条件不明确，结论就没有意义。反过来说，测试用例的本质，就是把"特定条件"固化下来。

>

> 第三，观察或记录结果。这里的关键词是"记录"。光看见不行，要有可追溯的记录：实际结果是什么、截图在哪、日志在哪。这也是为什么我们课程实验要求大家截图留证。

>

> 第四，对系统的某个方面做出评价。注意两点：一是"做出评价"，测试必须给判断，不能只报现象；二是"某个方面"，说明测试是有侧重的，一次测试通常只针对功能、性能、兼容性这类特定方面，不可能一口气覆盖全部。

>

> 四个要素连起来，就是完整的测试活动：明确条件下运行 → 观察并记录结果 → 对特定方面做出评价。以后写用例、写测试报告，都可以拿它来检查有没有漏。
>
> **板书：**ISO/IEC 24765（IEEE、ISO/IEC 29119 口径）→ 四要素：① Activity 一项活动 ② executed under specified conditions 特定条件下运行 ③ results observed or recorded 观察或记录结果 ④ evaluation of some aspect 对某个方面做出评价

提问：这个定义里，哪个要素是很多同学做实验时最容易忽略的？漏了它会有什么后果？

> **预设回答：**①答"记录结果"——最常见，追问后果：没有记录就无法复现、无法证明缺陷存在，开发一句"我这边没问题"就能把缺陷打回去。②答"特定条件"——也很好，追问：条件不写清楚，别人照你的用例跑不出同样结果，用例就没有可重复性。③答"做出评价"——提醒：只报"点了一下没反应"是现象，不是评价，要

> 说清"这违反了哪条需求、严重程度如何"。点评：把要素落到课程设计的具体动作上——用例要有前置条件、步骤、预期结果、实际结果、截图和结论。

> **易错点 / 考点：**易错是把定义背成"运行软件找错误"，漏掉"特定条件""记录""评价"三个要素。考点：定义的填空与要素识别；判断某段工作是否属于测试活动（如"只运行不记录、不评价"——不完全符合定义）。记忆四要素口诀：有条件、要运行、留记录、给评价。

> **过渡：**这条定义讲的是"运行和评价"，下一页的续讲，补上定义的另一半——测试到底在比较什么。

【1.38】软件测试 IEEE/ISO29119 的定义 – 续（8 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页是上一条定义的续，它回答了一个更本质的问题：测试在做比较的时候，到底拿什么跟什么比？

>

> 先看英文：Testing is comparing what the test item does with what it is expected to do. 翻译过来就是：测试就是把测试项实际做的事情，和它被期望做的事情进行比较。这句话是全章最该记住的一句话之一。它把测试的动作抽象成了一个"比较"。

>

> 这个比较有两边。左边是"实际表现"——被测对象真实跑出来的行为，由运行环境、输入数据和代码逻辑决定，是客观、可观察的。右边是"期望表现"——需求规格、设计文档、接口契约、用户预期所规定它应该有的行为。测试要做的，就是设法让这两边对齐，然后找出它们的差。

>

> 第二条表述更正式：分析某个软件项，以发现现存的和要求的条件之差别（即错误），并评价此软件项的特性。请注意这里的括号——"差别（即错误）"。也就是说，缺陷的定义可以直接从这句话推出来：所谓错误，就是实际的条件和要求的条件之间的差别。这一下就把"缺陷"的判定标准讲清楚了：有差别才有缺陷，没有要求就没有差别，也就无从判定缺陷。这就是为什么需求不清楚的项目没法测——你连比较的右边都画不出来。

>

> 还有一个词不能漏：评价此软件项的特性。前面说找差别，这里说评特性。说明测试的输出有两类：一类是具体缺陷，一类是对特性水平的整体评价，比如性能达到什么量级、可靠性处于什么水平。它们对应两种报告：缺陷单和质量报告。

>

> 最后强调一点：既然是比较，测试的严谨程度就取决于两块——测得够不够全（左边取样够不够），拿来比的标准对不对（右边尺子准不准）。这两条正是后面要展开的测试用例设计与测试需求分析。
>
> **板书：**Testing＝comparing what the test item does（实际）with what it is expected to do（期望）→ 分析软件项，发现现存的条件与要求的条件之差别（即错误），并评价此软件项的特性 → 缺陷＝实际与要求的差别 → 输出：缺陷单＋质量报告

> **提问：**按照"测试是实际与期望的比较"这一定义，如果一个项目的需求文档写得很笼统，会直接导致什么后果？

> **预设回答：**①答"测试做不了"——方向对，要具体化：没有明确的"要求条件"，就没有比较的右边，无法判定通过或失败，测试结论只能靠个人感觉。②答"那就按开发说的算"——纠偏：拿开发的口头说明当标准，等于让被测方给自己定标准，独立性丧失。③答"先测着，发现问题再说"——追问风险：漏掉的问题无从发现，也无法度量测试充分性。点评：这正是测试人员必须尽早参与需求评审的原因——把不可判定、二义性的需求在评审阶段挑出来。

> **易错点 / 考点：**易错是把"缺陷"理解成"程序崩溃"这类明显故障，漏掉"与要求不符"这个判定核心（界面与设计稿差两像素、日志格式不符要求，同样是缺陷）。考点：由定义推导缺陷定义（简答）；"没有需求文档能不能测"（判断／案例）。判断三步法：找到"要求"→ 找到"实际"→ 两者有差别即缺陷。

> **过渡：**定义和比较都清楚了，最后我们回答一个务实的问题：投入这么多做测试，究竟换回了什么？

【1.39】软件测试的价值（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是本节前一部分的收口：软件测试的价值。课件给了四条，我一条一条讲，并且告诉你们每一条在实际项目里对应什么动作。

>

> 第一条，全面评估产品质量，获得有关产品质量的全面、客观的信息。注意"全面"和"客观"这两个词。全面，是说测试不能只盯功能，还要看性能、兼容性、易用性、可靠性这些方面；客观，是说结论要有数据支撑，不能凭感觉说"我觉得差不多"。这一条对应的产出，是测试报告里的质量评估部分。

>

> 第二条，发现问题，督促问题解决，提高产品质量。这一条强调"督促"。测试人员找到缺陷只是第一步，缺陷不修，测试的价值就等于零。所以测试工作里有一大块是缺陷跟踪：确认缺陷被受理、被修复、被验证关闭。这也是我们课程设计要求大家写缺陷单、跟踪缺陷状态的原因。

>

> 第三条，持续提供质量反馈、及时揭示质量风险，有助于控制项目风险，提高构建的质量。这一条最容易被忽略，但项目经理最看重。软件测试像什么？像汽车仪表盘。它不能替你开车，但它告诉你油量、水温、车速，让你在抛锚之前做出决策。测试就是在每个版本、每个迭代，把"当前质量到了什么程度、哪些模块风险高"这个信息交给决策者。没有这条信息，项目管理者就是在盲开。

>

> 第四条，通过缺陷分析，获得缺陷模式，有助于缺陷预防。这是最高层次的价值：把已经发生的缺陷归类、统计、找规律——比如某一类缺陷反复出现在某几个模块，或者集中在某个开发阶段——然后回头去改流程，让这一类缺陷不再产生。发现缺陷是治已病，缺陷预防是治未病。

>

> 四条连起来看，测试的价值是层层递进的：给信息 → 促修复 → 控风险 → 防缺陷。这也正好呼应了我们前面讲的学科发展四个阶段，说明测试早就不只是"最后一道检验工序"了。
>
> **板书：**软件测试的价值：①全面评估产品质量，获得全面、客观的质量信息 ②发现问题、督促解决、提高质量 ③持续质量反馈、及时揭示风险、控制项目风险、提高构建质量 ④缺陷分析→缺陷模式→缺陷预防 → 层层递进：给信息→促修复→控风险→防缺陷

提问：这四条价值里，哪一条最容易被团队忽略，但对项目风险控制最关键？为什么？

> **预设回答：**①答"第三条，持续提供质量反馈"——正确，追问：如果测试只在最后出一份报告会怎样？项目管理者在中间做不了决策，风险只能到最后一起爆。②答"第二条，督促问题解决"——可以讨论，补充：督

促偏"事后"，而第三条是"过程性"的，价值在于及时。③答"第一条，评估质量"——提醒：评估是基础，但如果没有持续反馈，评估结果来得太晚，只能"验尸"。点评：测试真正的价值是让风险早暴露，而不是让报告更漂亮。

易错点 / 考点：易错是把"测试的价值"说成"保证软件没有缺陷"——测试不保证无缺陷，它提供质量信息、降低风险。考点：四条价值的记忆与排序（简答）；结合案例说明缺陷预防的价值。记忆口诀：评质量、促修复、控风险、防缺陷。

【思政落点 2】：讲国产游戏《血狮》（1997）——宣传声势远超其实际被验证的质量，交付后口碑崩塌；再对照波音星际客机因测试未做到端到端而出问题。两个方向一个共同点：把"没测到"当成了"没问题"。联系知识点：测试的价值在于四点，而缺陷预防是最高一层。落到价值观：测试是替用户把最后一道关，如实记录缺陷、不隐瞒不利结果，是从业者的职业底线与质量责任；家国情怀要靠过硬的质量来承载。

过渡：到这里，"测试是什么、为什么值得做"已经讲清楚了。带着这四条价值，我们从"为什么测"转向"怎么测"，继续沿着教材的思路往下展开。

【1.40】1.3.4 软件测试的其它观点（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

前面几页我们走了三步：1.3.1 看软件测试这门学科是怎么形成的，1.3.2 看围绕测试的正反两方面的争辩，1.3.3 给出软件测试的定义。到了这一页，1.3.4，我们要接着追一个问题：定义有了，那能不能从更多角度把测试看得更立体一点？答案是能。这一节给大家五个视角，等于从五个不同的坡面去爬同一座山。第一，质量视角，测试是对软件质量进行全面评估；第二，风险视角，测试是对潜在的各种质量风险进行评估；第三，经济视角，测试是用最小的代价换最高的软件产品质量；第四，基于 Test Oracle 的认知，判断输出结果对不对，得有判断准则；第五，基于批判性思维的认知，测试是借助观察、经验、反思、推理和沟通来收集信息、做出结论的不断探索的过程。有同学会问：定义不就够了吗，为什么要费劲讲五个视角？因为定义是一句话，观点是一整套看问题的方式。你在企业里跟人讨论"这个版本要不要多测一轮"，只背定义是说服不了人的：跟项目经理要讲风险，跟老板要讲成本，跟开发要讲判断准则，跟客户要讲质量。不同角色的语言不一样，测试人员必须会切换语言。打个比方，同一辆车，司机关心好不好开，修车师傅关心哪里容易坏，保险公司关心出事故的概率，二手车商关心值多少钱——车是同一辆，问题不同，结论自然不同。还要提醒一句：这五个视角不是五套互相打架的说法，而是互补的。质量视角回答"测什么"，风险视角回答"先测什么"，经济视角回答"值不值得测"，Test Oracle 回答"怎么判定对错"，批判性思维回答"用什么态度去测"。考试爱考"观

点与核心主张"的对应，给你一句话让你判断它属于哪个视角，所以接下来每一页，请重点抓那句话的关键词。

板书： 1.3.4 其它观点 → 质量视角｜风险视角｜经济视角｜Test Oracle｜批判性思维 → 互补、不冲突

提问： 如果项目经理说"这个版本时间很紧，少测一点"，五个视角里你更可能用哪一个去跟他沟通？为什么？

预设回答： ①答"风险视角"——正确，且能说出"按风险排序，砍掉低风险项而不是平均砍"。教师点评：方向对，要追问一句"那你怎么证明哪些是低风险"，逼学生给出风险判断的依据。②答"质量视角"——只对了一半。教师追问：你讲质量，项目经理会说"质量我当然要，但时间不够"，这句话没有说服力；质量视角讲的是"测什么"，不是"砍多少"。③答"经济视角也行"——可以接受，但要提醒：经济视角解决的是"这笔测试投入值不值"，是用成本收益算账，跟"先测哪块"是两个问题。把三种回答并排写在黑板上，正好演示五个视角各管一段，不是随便挑一个都行。

易错点 / 考点： 最典型的错，是把五个视角背成五句口号，却分不清谁回答哪个问题。判断三步法：先看这句话在谈"测什么"还是"先测什么"还是"值不值"；再看它有没有提到判断准则；最后看它是不是在讲收集信息、做结论的过程。选择题常考"从经济视角看软件测试的核心是什么"这类对应题，判断题常把"测试就是找Bug"当正确项来设置陷阱。

过渡： 五个视角，我们从最容易被挂在嘴边、也最容易被理解偏的一个开始——质量视角。

【1.41】从质量视角认知软件测试（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页进入质量视角。课件上那句话是：软件测试被认为是对软件质量进行全面评估的活动，给出质量信息，从而确定质量是否满足设计和用户的需求。请抓住三个关键词：全面评估、给出质量信息、满足需求。第一层，测试是一项评估活动，不是"找茬活动"。大家看这张图，中间画着"发现 Bug？"，后面专门带了一个问号，这就是在提醒我们：测试的目的不只是找 Bug，而是给出质量信息，让决策者知道这个软件现在到底能不能交付。你把 Bug 报上去，开发改完，测试再验，这一整套动作的最终产出，是一份关于质量的判断，而不是一堆缺陷单。第二层，为什么会有缺陷？看图的下半部分：左边是用户，带着"要求/期望"；右边是质量，也就是实际做出来的东西；中间标着"矛盾/对立"。软件缺陷的本质，就是做出来的东西和用户要求的、期望的东西之间出现了落差。这句话很重要，它解释了后面为什么要做需求评审、为什么要写验收标准——因为只有当"要求和期望"被写清楚，落差才可判定。第三层，靠什么评估？图里写了"质量模型"。质量模型就是一把尺子。国际上常用的软件质量模型，会把质量拆成功能适合性、性能效率、兼容性、易用性、可靠性、安全性、可维护性、可移植性这些特性。没有这把尺子，"质量好"就是一句空话，A 说挺好，B 说不行，吵到

最后谁嗓门大谁赢。生活里的道理是一样的：体检报告不会只写"你身体不错"，它给出血压、血糖、血脂这些指标，让医生和你都能对比、能追踪。测试报告也一样，要给出可对比、可追踪的质量信息。为什么这个视角重要？因为太多团队的测试报告只写一句"测试通过，未发现严重问题"，这句话对决策几乎没有价值——它是结论，不是信息。真正有用的写法是：测了哪些质量特性、覆盖了多少条需求、剩余风险集中在哪里。这一页还埋下了后面反复要用的三个概念：需求是测试的依据，质量特性是测试的维度，评估结论必须建立在证据之上。

板书：质量视角 → 全面评估 + 给出质量信息 → 用户"要求/期望" ↔ 质量（矛盾/对立）→ 评估靠"质量模型" → 发现 Bug？（不只是找 Bug）→ 需求是依据、质量特性是维度、结论靠证据

提问：课件的图里，"发现 Bug"后面为什么要加一个问号？如果测试只以"找到 Bug"为目的，会漏掉什么？

预设回答：①答"因为测试的目的不只是找 Bug"——正确。教师追问：那除了找 Bug，测试还产出什么？引导学生说出"质量信息、能否交付的判断、剩余风险"。②答"因为不是每次都能找到 Bug"——这是典型错误，把目的和结果搞混了。教师点评：找不到 Bug 的那一轮测试一样有价值，它提供了"当前版本在这些范围内没有发现问题"的信息，这是发版决策的证据；要找 Bug 才做测试，等于把测试当成抽奖。③答"因为 Bug 这个词有歧义，需求理解错了也算 Bug"——有深度，可以顺势引到下一句"矛盾/对立"：需求本身有歧义时，判断对错的标准就不牢靠，所以要提前评审需求。三种回答按"结论—证据—标准"三层排一下，学生就明白为什么测试报告要写信息而不是只写结论。

易错点 / 考点：两个混淆点。第一，把"测试"等同于"找 Bug"，忽略"给出质量信息"，这是最常见的选择题陷阱。第二，把质量模型当成考试名词硬背，却不会用它组织测试：考案例题时，要求你针对某个系统列出该测哪些质量特性，只写"功能"两个字是拿不到分的。记忆口诀：要评估、给信息、比需求、靠模型。

过渡：质量视角回答的是"测什么"，但它回答不了"先测什么"。要回答这个问题，我们得换到风险视角去看。

【1.42】从风险视角认知软件测试（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页讲风险视角。课件上写着：软件测试被认为是对软件系统中潜在的各种质量风险进行评估的活动。后面还有一句非常关键的话：测试是样本实验而不能穷尽，其风险总是存在的。再往下，是"基于风险的测试"这个说法——它强调对软件开发的全过程进行检测，随时发现问题、报告问题，减少对客户不利影响的风险。我们先拆"测试是样本实验而不能穷尽"。这句话的意思是：你不可能把用户所有可能的输入、所有可能的操作顺序、所有可能的运行环境都试一遍。输入组合是爆炸式增长的，一个几百行的小程序，穷举路径都

做不到，何况一个几十万行的系统。既然测不完，那就必须选择测什么——这就是样本实验。既然是抽样，就一定会有漏网之鱼，所以风险永远存在，测试的目标不是"消灭风险"，而是把风险压到可以接受的水平。这就引出基于风险的测试：哪里出事概率高、出了事后果重，就优先测哪里。风险大致等于"发生的可能性"乘以"一旦发生的损失"。举个身边的例子：一个电商系统，支付和退款模块一旦出错就是真金白银的损失，而"关于我们"页面的错别字损失极小。时间只够测一半的时候，先测支付，这叫按风险排序，不叫偷懒。再看图上这条链：上面是需求、设计、代码，中间是功能和非功能特性，往下是测试，最后到交付；左边站着产品经理、项目经理和开发人员，图里向下的箭头旁边写着"持续反馈质量风险"，而且是两处，意思是从头到尾一直在反馈。这一点是理解本页的钥匙：风险不是等到测试阶段才暴露，需求写得含糊、设计接口对不上、代码没有异常处理，这些都是风险源，测试人员要尽早介入，随时把风险信息反馈回去。所以风险视角下的测试，是贯穿开发全过程的一项检测活动，不等同于最后阶段的验证。对客户来说，提前发现一个高风险问题，可能就避免了一次线上事故、一次口碑崩塌。

板书：风险视角 → 测试＝样本实验，不能穷尽 → 风险＝可能性 × 影响 → 基于风险的测试：全过程检测、随时发现、随时报告 → 需求→设计→代码→（功能/非功能特性）→测试→交付，向上"持续反馈质量风险"

提问：同样的测试时间，为什么"平均分配给每个功能模块"是一种不负责任的做法？请举一个你熟悉的系统来说明。

预设回答：①答"因为有的模块出问题损失大，平均分配等于高风险的地方没测够"——正确，能说出概率和影响两个维度更好。教师点评：让他具体说出"哪个模块、什么后果"，把抽象原则落到场景。②答"因为测试时间不够，从来测不完"——只答对了一半，这只解释了为什么要选择，没解释为什么不能平均。教师追问：那如果某个模块又简单又不重要，你给它同样的人力，损失的是什么？引导学生说出"机会成本"。③答"平均分配没什么不好，覆盖面广"——这是很典型的错误直觉。教师点评：覆盖面广不等于风险低，覆盖了十个不重要功能，不如覆盖一个支付功能；把这句话写在黑板上，作为"风险排序"的反面教材，再让学生把它改写成"按风险×影响排序"。

易错点 / 考点：第一，把"基于风险的测试"误解成"高风险模块多测几轮"这么简单，漏掉"全过程检测、持续反馈"这一半。第二，误以为测试可以做到零风险、穷尽验证，判断题常拿"只要测试足够充分就能保证软件无缺陷"来设坑，答案是错的。判断三步法：先问"能不能穷尽"（不能），再问"先测哪里"（风险高的），最后问"风险从哪来"（需求、设计、代码全过程）。

过渡：既然测不完，就必须做取舍；而只要谈取舍，就绕不开钱。接下来我们从经济视角算一算这笔账。

【1.43】从经济视角认知软件测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页从经济视角看测试。课件原话是这样四句：测试的经济观点就是以最小的代价获得最高的软件产品质量；经济观点也要求软件测试尽早开展工作；发现缺陷越早，返工的工作量就越小，所造成的损失就越小；还有一句是最扎心的——测试的成本要小于缺陷造成的损失，测试才有意义。我们一句一句来。第一句，"以最小的代价获得最高的质量"，注意不是"不计代价追求零缺陷"。测试是要花钱的：要人、要时间、要环境、要机器。如果为了一百万分之一的概率去投入巨大成本，那这笔钱花得不值。所以测试从来不是"测得越多越好"，而是"花得值"。第二句和第三句连起来看，是这一页的核心：为什么要尽早测？因为发现得越早，返工的工作量越小，损失越小。缺陷像滚雪球，需求阶段一句话的含糊，到了设计就变成接口定义错误，到了编码就变成一片实现，到了测试甚至上线才暴露，这时候要改的不只是一个字，而是设计、代码、文档、数据，甚至已经交付给用户的产品。生活里的类比很好懂：牙上有个小黑点，早去看，补一下就好；拖到牙神经坏了，就要根管治疗加牙冠，钱和罪都翻好几倍。汽车刚有异响就去检查，也许只是换个垫片；拖到高速上抛锚，损失的就是拖车费、误工费甚至安全。软件行业最典型的例子，就是英特尔当年奔腾处理器的浮点除法缺陷，据本课程案例素材，最后召回换货付出的代价在四亿美元这个量级。一个除法算错的细节，最后变成了一场大规模的召回。第四句给了我们一把决策尺子：测试的成本小于缺陷造成的损失，测试才有意义。反过来，如果某类缺陷造成的损失远小于测试它的成本，那就要重新考虑测到什么程度。这也解释了为什么测试要有优先级、要做风险排序——钱要花在刀刃上。同学们以后写测试计划，里面一定会有"范围和取舍"，那时你要能说清：我为什么测这些，为什么这一段先测，为什么那一段只做抽样。

板书：经济视角 → 最小代价 × 最高质量 → 尽早测试：发现越早，返工越小、损失越小 → 决策尺子：测试成本 < 缺陷损失，测试才有意义 → 测试不是"越多越好"，是"花得值"

提问：有人说"质量是免费的"。结合这一页的经济视角，你同意吗？如果不完全同意，请把这句话补完整。

预设回答：①答"不完全同意，预防的投入是必要的，但质量不是零成本"——很好。教师点评：这话的原始含义是"把缺陷挡在前面比事后返工便宜"，而不是"一分钱不花就有高质量"。②答"同意，做好质量就不需要返工"——典型错误。教师追问：那测试人员、测试环境、回归测试的机器要不要钱？引导学生区分"预防成本低"和"成本为零"。③答"只有测试成本小于损失才值得测"——抓住了课件的尺子。教师点评：这句话是对的，但要追问边界——损失怎么估？让学生意识到"损失估算"本身就是一件需要判断的事，不能拍脑袋。

易错点 / 考点：常见错误有两个。一是把经济视角读成"少测省钱"，其实它的落脚点是"尽早测"和"算总账"。二是忽略"测试也是有成本的"，答案例题时只写"要测得更充分"，不提成本与收益的平衡。考试常考判断

题："缺陷发现得越晚，修复成本越高"——正确；"测试投入越多，质量一定越高"——错误。记忆口诀：早发现、小返工、算总账。

【思政落点 43】讲什么故事：课件这句"测试成本要小于缺陷造成的损失，测试才有意义"，背后是企业实实在在的钱，也是用户实实在在的安全。怎么联系知识点：经济视角不是教大家省钱省测试，恰恰相反，它告诉我们缺陷一旦流到用户手里，代价会成倍放大——英特尔奔腾的浮点除法错误，最后是四亿美元量级的召回。落到什么价值观：工程人员的每一次"这个先不测了，赶进度"，都不是一句轻飘飘的话，最终买单的是企业和用户；把缺陷挡在交付之前，既是职业能力，也是质量责任的底线。

过渡：经济视角告诉我们要算总账，可要算账，前提是能判断"这一段结果到底对不对"。这就引出了 Test Oracle 的认知。

【1.44】基于 Test Oracle 的认知（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页很短，字也很少，但它是整个测试活动的地基。课件上只有一句话：输出结果是否正确，需要判断准则。图也只有一张，画的就是这个意思。什么叫判断准则？就是当你把一组输入喂给程序、程序吐出一个结果的时候，你凭什么说这个结果是对的。这个"凭什么"，在测试领域有一个专门的术语，叫 Test Oracle，中文一般翻译成测试预言或测试判定准则。为什么它这么重要？因为没有 Oracle，测试就变成了"看着像对的就算对"。举个最简单的例子：一个计算器程序，你输入 2 加 3，它输出 5，你说对；如果它输出 6，你说错。你的判断准则来自小学数学，这就是 Oracle。再比如登录功能：输入正确的账号密码，期望结果是登录成功、跳转到首页、并且页面右上角显示用户名。这一串"期望结果"，才是你的判断准则。如果只写"登录成功"，那系统提示"登录成功"却把你导到一个空白页，你算不算通过？没有准则，就判不了。那 Oracle 从哪里来？常见的有几类：需求规格和验收标准，这是最正规的来源；业务规则和领域知识，比如财务系统里借贷必须相等；旧系统的实际行为，做系统改造时拿老系统当参照；已有的参考实现或者同类产品的表现；还有测试人员的人工经验判断。这里要提醒一个现实困难，业内叫"Oracle 问题"：有些输出很难有唯一的正确标准。比如界面的美观程度、语音识别的结果质量、算法给用户推荐得合不合理，甚至是今天大量的 AI 输出——这些地方"对"的边界很模糊。遇到这种情况，通常的做法是把它转化成可判定的形式：把美观转成设计规范的检查项，把推荐质量转成一批标注好的样本和一条通过线，或者引入多人评审。所以这一页想让大家记住的是：测试不是"跑一遍看看"，测试是先有准则、再执行、最后比对。你在本课程的实验里写测试用例，表头一定会有"预期结果"这一列，那一列就是你的 Oracle，它不是形式主义，它是你能不能判定通过的依据。

板书： Test Oracle → 输出结果是否正确，需要判断准则 → 准则来源：需求规格/验收标准｜业务规则与领域知识｜旧系统行为｜参考实现｜人工经验 → 有准则才能判"通过/不通过" → 难判定的输出要转化为可判定形式

提问： 测试用例表里"预期结果"这一列，如果空着或者写成"正常显示"，会带来什么后果？

预设回答： ①答"没法判断测试通过还是失败，两个人可能得出两个结论"——正确。教师点评：再追一步，这种不一致在团队协作里会导致什么？引导到"缺陷争议、返工、回归标准不统一"。②答"没关系，执行的人自己知道对错"——典型错误。教师点评：写用例的人知道，不代表执行的人知道，更不代表三个月后回归测试的人知道；用例是给团队用的，不是给个人备忘的。③答"可以事后补上"——可接受但不是好做法。教师追问：如果开发改完代码，你又补了新的预期结果，那这次改动到底算通过还是没通过？引导学生理解：Oracle 是先于执行确定的，事后补等于给结果找解释。

易错点 / 考点： 第一，把 Test Oracle 当成某个工具或某个测试类型，其实它是一个概念——判定准则。第二，写用例时把预期结果写成"结果正确""正常"这类无法判定的表述，这是实验和课程设计里最常见的扣分点。第三，误以为 Oracle 只能来自需求文档；事实上领域知识、旧系统、参考实现、人工经验都可以，但来源越客观，争议越小。判断口诀：先有准则、再执行、后比对。

过渡： 有了准则，还差一样东西——态度。因为再好的准则，也挡不住"我觉得没问题"这种随意判断。

【1.45】基于批判性思维的认知（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页把测试和一种思维方式绑在一起。课件上是这么说的：软件测试就是借助观察、经验、反思、推理或沟通等收集信息，并对软件产品相关的质量信息进行分析，以此评估软件质量，并做出结论。后面还跟着一句短语：不断探索的过程。我们先看前半句里的五个手段：观察、经验、反思、推理、沟通。观察，是看现象——点下去有没有反应、返回的数据对不对、界面有没有错位；经验，是你见过类似的系统怎么出错，比如日期、金额、并发；反思，是回头想"我刚才这个判断是不是太快了""这里没出问题是不是因为我根本没测到"；推理，是从已知推未知——这个输入框限制了长度，那粘贴超长文本会怎样；沟通，是找开发、产品、用户把需求问清楚。这五种手段合起来，是在收集信息，而不是在收集感觉。后半句"做出结论"，说明测试最终要给出判断，但判断必须建立在分析过的质量信息之上。那"批判性思维"体现在哪里？体现在一个根本立场：不轻信。程序说"保存成功"，不一定真保存了，要去数据库里核实；界面显示"操作已完成"，不一定真完成，要看状态有没有变；开发说"这个功能没问题"，那就要用证据来确认。这跟前面 risk 视角里那句"测

试是样本实验"是一脉相承的。反过来，如果测试人员的思维方式是"我写的用例都跑过了，应该没问题"，那就从批判性思维滑回了自我安慰。这也是本页两个图要表达的对照：一个是不断提出疑问、不断探索的循环，一个是一旦停止怀疑就停止发现缺陷。生活里我们其实每天都在用批判性思维：买东西看配料表，不听广告听成分；看到"限时特价"先想是不是原价就那样。做测试就是把这种较真的习惯，变成一种有方法、有记录、有结论的工作方式。请记住这句话：测试人员不是软件的敌人，但一定是"未经证实的结论"的敌人。

板书：批判性思维 → 观察｜经验｜反思｜推理｜沟通＝收集信息 → 分析质量信息 → 评估质量、做出结论 → 不断探索的过程 → 核心立场：不轻信、要证据

提问：界面上弹出"保存成功"，为什么还不能直接判定这个功能测试通过？你会再去看什么？

预设回答：①答"要去数据库或列表里查一下数据真的写进去了"——正确。教师点评：这就是观察加推理，把界面提示当作线索而不是结论。②答"把提示截个图，用例就算过了"——典型错误，属于只验证了现象、没验证结果。教师追问：如果程序只弹了提示、实际什么都没存，你这条用例是不是漏掉了缺陷？③答"看开发怎么说"——不可取。教师点评：沟通是手段之一，但沟通用来澄清需求，不能用来替代验证；开发说存了，你仍然要自己看到证据。

易错点 / 考点：第一，把批判性思维理解成"跟开发抬杠""什么都怀疑"，其实它的落点是收集信息、分析信息、做出有依据的结论。第二，把"探索"当成漫无目的地乱点，忽略课件列的五个手段和后续的分析动作。第三，考试里常把"测试是证明程序没有错误"当正确项，这是反的——测试是带着怀疑去求证，不是去证明"我写对了"。记忆口诀：不轻信、找证据、做结论。

【思政落点 45】讲什么故事：批判性思维这一页，考验的其实是一个人的诚信与科学态度。怎么联系知识点：课件要求测试借助观察、经验、反思、推理、沟通去收集信息，再基于信息做出结论——这就是用证据说话。落到什么价值观：发现缺陷不报告、为了赶进度隐去不利结果、把"没测"写成"通过"，都是对事实的背叛；如实记录、如实评价，是测试人员最基本的职业操守，也是我们这门课过程性考核看重的态度。

过渡：到这里，软件的"其它观点"讲完了。接下来要处理一个在课堂上和工作中都极易混淆的关系——测试和质量保证到底是什么关系。

【1.46】1.4 测试与质量保证的关系（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 1.4 节的开头，标题就是本节要回答的问题：测试与质量保证的关系。大家先看这张图，它把本节要

建立的框架先摆出来了。为什么单独拿一节来讲这个关系？因为这是最容易被混为一谈的一对概念。很多同学一听说"质量保证"，脑子里想的就是"测试"；也有同学以为"测试"和"质量保证"是两个各自为政的部门。这两种理解都会出问题：前一种会导致你把质量保证做成了事后检测，后一种会导致两边互相推责任——测试说"你怎么没保证质量"，质量保证说"我提醒过你，是你没测出来"。我们先看课件备注里列的内容：1.3.1 软件测试学科的形成、1.3.2 正反两方面的争辩、1.3.3 软件测试的定义、1.3.4 软件测试的其它观点——这是刚才这一大段的回顾，说明我们前面的讨论都还在 1.3 节里。现在转进 1.4，接下来三页要依次回答三个问题：第一，什么是 SQA，也就是软件质量保证；第二，SQA 到底做哪些活动；第三，软件测试和 SQA 之间谁包含谁、谁指导谁、谁给谁提供数据。这三个问题的答案，都会在实验报告和课程设计文档里用到——比如你写测试计划，如果分不清"测试活动"和"SQA 活动"，范围就会写乱；你写质量报告，如果分不清"我提供数据"和"我负责整个流程"，职责就会越界。所以这一节虽然概念性强，但它是后面所有文档写作的语法。最后把本节最容易混的一点先点出来：以后做题，凡是问"谁负责流程与规范"的，答案指向质量保证；凡是问"谁给出产品的验证结论"的，答案指向测试。这两个答案千万不要对调。

板书：1.4 测试与质量保证的关系 → 三问：什么是 SQA？SQA 有哪些活动？测试与 SQA 谁包含谁？→ 前面回顾：1.3.1 学科形成｜1.3.2 正反争辩｜1.3.3 定义｜1.3.4 其它观点

提问：你们觉得"测试"和"质量保证"是一回事吗？如果是两回事，差别在哪一句话上？

预设回答：①答"不是一回事，测试是查产品，质量保证是管流程"——很接近正确答案。教师点评：这句直觉很好，先记在黑板上，等第 49 页再严格对照。②答"是一回事，测试就是保证质量"——最常见的错误。教师追问：如果测试就是质量保证，那一个没有独立测试团队的公司，是不是就没有质量保证了？引导学生发现两者活动范围不同。③答"质量保证更大，包含测试"——方向对了。教师点评：正确但不完整，要补上"包含"之外还有"指导和监督"这层关系，留到第 49 页揭晓。

易错点 / 考点：本节的混淆点集中在层级关系上——软件测试是 SQA 的活动之一，但 SQA 不等于测试；"测试是 SQA 的重要手段"和"SQA 指导监督测试"是两句话、两个方向，不能只答一半。判断三步法：先看它说的是流程还是产品，再看它是给数据还是提要求，最后落到对应的那一栏。选择题常把两者对调来设坑，务必看清主语。

过渡：要把这个关系讲清楚，先得知道 SQA 本身是什么——下一页我们就给它一个正式的定义。

【1.47】什么是 SQA？（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页回答什么是 SQA。SQA 是软件质量保证的英文缩写，全称 Software Quality Assurance。课件给出了一个完整定义：软件质量保证活动是通过对软件产品有计划的进行评审和审计来验证软件是否合乎标准的系统工程，通过协调、审查和跟踪以获取有用信息，形成分析结果以指导软件过程。这个定义里有三个动作请画下来：有计划的评审、审计、以及验证是否合乎标准；还有一条目的：指导软件过程。注意最后这四个字——指导软件过程，它说明 SQA 关心的不只是"这个产品行不行"，更是"我们造产品的这套流程行不行"。围绕这个定义，课件又给了三句展开。第一句：对软件工程各个阶段的进展、完成质量及出现的问题进行评审、跟踪。注意"各个阶段"，从需求到设计到编码到测试到上线，每个阶段都要有检查点，而不是最后一次性大检查。第二句：审查和验证软件产品是否遵守适用的标准、规程和要求，并最终确保符合标准、满足要求。这里说的是"合规"，标准可能是国际标准、行业标准，也可能是公司自己定的开发规范和编码规范。第三句：建立软件质量要素的度量机制，了解各种指标的量化信息，向管理者提供可视信息。这一句最容易被忽略，但它非常实在——质量保证要拿出数据，比如缺陷密度、评审发现率、需求变更次数、用例通过率，用这些量化信息让管理者看清楚现状。把三句合起来看，SQA 的工作方式可以概括成三个词：定标准、查执行、报数据。生活里最好懂的例子是食品安全检查：不是等吃坏肚子才去查，而是从原料采购、生产车间、包装、出厂检验，一路都有标准和抽查记录。软件也一样，SQA 是给软件生产过程装的一套"质量体系"，测试只是这套体系里的一环。所以如果你以后做测试工程师，会经常收到 SQA 发来的评审通知、度量表格和规范更新；而如果你做质量保证，就要学会从测试数据里读出流程的问题，而不是只盯着缺陷数。

板书： SQA＝Software Quality Assurance 软件质量保证 → 定义：有计划地评审、审计，验证是否合乎标准 → 目的：获取信息、分析结果、指导软件过程 → 三句展开：①各阶段进展/质量/问题 评审跟踪 ②审查验证是否遵守标准、规程、要求 ③建立度量机制，向管理者提供可视信息 → 三词概括：定标准、查执行、报数据

提问： "向管理者提供可视信息"为什么也要算 SQA 的工作？只提交一份缺陷清单不行吗？

预设回答： ①答"不行，管理者要做决策，需要看趋势和量化指标，不只是零散的问题"——正确。教师点评：顺势点出缺陷密度、通过率这类指标比单条缺陷更能反映流程状态。②答"因为管理者看不懂技术细节"——有点道理但角度偏了。教师点评：不是看不懂，而是管理决策需要的是"趋势、对比、风险"，单条缺陷回答不了"这个版本能不能发"。③答"这是形式主义，交清单就够"——典型错误。教师追问：如果缺陷有 200 条但都集中在同一个模块，清单能告诉你什么？引导学生说出"集中度、根因、流程改进方向"。

易错点 / 考点： 第一，把 SQA 缩写成"质量检测"，忽略它是过程性的、计划性的系统工程。第二，只记住定义，答不出 SQA 的三个工作面向（评审跟踪、合规验证、度量报告）。第三，考试中常出现"以下哪项不属

于 SQA 的工作"，要把 SQA 的度量、评审、审计、标准执行和"具体写代码"区分开。记忆口诀：定标准、查执行、报数据，目的是指导过程。

过渡：定义讲完了，那 SQA 具体要做哪些活动呢？下一页列出了七项。

【1.48】SQA 活动（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页列出 SQA 的具体活动，一共七项，我们一项一项过。第一，技术方法的应用。就是说开发过程要用对方法和工具，用什么开发方法、什么设计方法、什么建模手段，SQA 会关注这些方法有没有被正确使用，因为它直接影响质量。第二，正式技术评审的实施。注意"正式"两个字——它和随便找人看两眼完全不同，正式技术评审要有计划、有角色、有检查单、有记录、有结论。第三，软件测试。这一项请大家重点画下来。软件测试是 SQA 的活动之一，也就是说测试在质量保证的框架之内。第四，标准的执行。编码规范、文档模板、命名约定、配置管理要求，这些标准要落地执行，SQA 负责推动和检查。第五，修改的控制。也就是变更控制，代码、需求、文档的每一次修改都要走流程、要评估影响、要留痕。为什么这一项重要？因为大量缺陷不是"一开始就写错"，而是"改出来的"——修了 A 功能结果弄坏了 B 功能。第六，度量。要建立指标，收集数据，比如缺陷密度、评审效率、测试通过率。没有度量，改进就没有方向。第七，质量记录和记录保存。评审记录、测试报告、变更记录，这些都要归档保存，一方面是为了追溯，另一方面是为了体系审核时能提供证据。把这七项放在一起看，你会发现一个规律：它们大部分都不是"直接检查产品"，而是在检查"做事的方式"——方法对不对、评审做没做、标准守没守、变更管没管、数据有没有。只有第三项"软件测试"是直接对产品做验证和确认。这就解释了为什么测试是 SQA 的重要手段之一，但绝不是全部。同学们在做课程设计的时候，其实可以拿这七项当自查表：你的项目有没有走评审？代码有没有规范？需求变更有没有记录？测试数据有没有统计？测试报告有没有归档？这七问答下来，基本就知道一个团队的质量管理水平在哪里。也提醒一句，考试里非常喜欢问"软件测试是不是 SQA 的活动之一"，答案是肯定的，别答成"两回事"。

板书：SQA 活动七项 → ①技术方法的应用 ②正式技术评审的实施 ③软件测试 ④标准的执行 ⑤修改的控制 ⑥度量 ⑦质量记录和记录保存 → 多数查"做事方式"，第③项直接查产品 → 结论：测试是 SQA 的重要手段之一，不是全部

提问：这七项里，哪一项最容易被小团队省略？被省略之后，通常在什么时候付出代价？

预设回答：①答"质量记录和记录保存，或者度量"，也有人答"正式技术评审"——都合理。教师点评：请他给出一个具体后果，比如没有记录，出了线上问题无法追溯是哪次改动引入的。②答"测试最容易被省"——

不准确。教师点评：测试往往是小团队唯一保留的"质量动作"，恰恰是被裁到只剩它；真正的空白通常是评审、度量和记录。③答"七项都得做，一项都不能省"——态度可嘉但要引导到现实：资源有限时应该按风险取舍，但"不留记录"这种省法等于把可追溯性一起省掉了，代价最大。

易错点 / 考点：第一，漏项。七项里"修改的控制""度量""质量记录"最容易被忽略，考试多选常在这里设置正确项。第二，把"技术方法的应用"理解成"使用测试工具"，它的范围更宽，指开发全过程中技术方法的正确使用。第三，判断题常考"软件测试是 SQA 活动之一"——正确；"SQA 就是软件测试"——错误。记忆口诀：法（方法）评（评审）测（测试）标（标准）改（变更）度（度量）记（记录）。

过渡：七项活动里有测试，也有评审、标准、变更、度量。那测试在整个 SQA 里到底站在什么位置？下一页我们就正面对比。

【1.49】软件测试 vs. SQA（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是本节的核心，把测试和 SQA 摆在一起对照。课件给了三层关系，我们逐层看。第一层：SQA 指导、监督软件测试的计划和执行，督促测试工作的结果客观、准确和有效，并协助测试流程的改进。请注意这三个词——客观、准确、有效。SQA 不替测试干活，它盯着测试干得对不对：你的测试计划有没有覆盖关键需求，你的测试结果是不是如实记录，你的测试流程有没有可以改进的地方。第二层：软件测试是 SQA 重要手段之一，为 SQA 提供所需的数据，作为质量评价的客观依据。也就是说，SQA 要判断"这批产品的质量行不行""这条流程改得对不对"，它靠什么下判断？靠测试给的数据。测试是 SQA 的眼睛和手。第三层是最容易考的对照：SQA 是一项管理工作，侧重于对流程的评审和监控；测试是一项技术性的工作，侧重对产品进行评估和验证。一个管过程，一个验产品；一个关注"我们是怎么做的"，一个关注"做出来的东西对不对"。把三层连起来记，就是一句话：SQA 给测试定规矩、看规矩有没有被守，测试给 SQA 交数据、当质量判断的依据。这里要澄清两个常见误解。第一个误解是"测试做得越多，SQA 就越好"。不对——测试发现了一大堆缺陷，恰恰说明前面的需求、设计、评审环节有问题，SQA 该做的是去改进流程，而不是庆祝测试团队发现的缺陷多。第二个误解是"有了 SQA 就不用测试，或者有了测试就不需要 SQA"。也不对——没有测试，SQA 的结论没有客观数据支撑，只能靠感觉；没有 SQA，测试容易变成自说自话，标准不统一、结果不可比、流程不改进。这一页的知识在你们的课程设计里会直接用到：写测试计划时，范围、准则、记录方式其实都受质量保证要求约束；写质量报告时，你要清楚自己提供的是"数据与验证结论"，流程评审和改进建议属于另一条线。分得清这两条线，文档就不会写串。

板书：软件测试 vs. SQA → SQA 指导、监督测试的计划与执行（客观、准确、有效）→ 测试是 SQA 重要手段之一，为 SQA 提供数据 → SQA＝管理工作，侧重流程的评审与监控；测试＝技术工作，侧重产品的评估与验证 → 一句话：SQA 定规矩看规矩，测试交数据当依据

提问：如果某个版本测试发现缺陷特别多，是先表扬测试团队，还是先检查流程？为什么？

预设回答：①答"先检查流程，说明前面环节没拦住缺陷"——正确。教师点评：追一句"那测试团队该不该表扬"，引导学生区分"发现缺陷有价值"和"缺陷多说明前期失控"两件事。②答"表扬测试，说明测试认真"——情绪上合理，但漏了重点。教师追问：如果需求评审做扎实了，这批缺陷里有多少根本不会产生？让学生体会"质量是设计出来的，不是测出来的"。③答"两个都要查"——稳妥，可以接受。教师点评：这个回答不错，但要问清优先级：先看缺陷的引入阶段分布，才能判断是流程问题还是个别疏忽。

易错点 / 考点：第一，把两者的关系答成"包含关系"就停手，漏掉"指导、监督、提供数据"三层互动。第二，答选择题时把"SQA 侧重产品验证、测试侧重流程监控"选成正确项——这是把两者对调了，属于高频陷阱。第三，判断题"SQA 就是测试部门""测试人员就是 SQA 人员"都是错的。记忆口诀：SQA 管过程，测试验产品；SQA 给测试定规矩，测试给 SQA 交数据。

【思政落点 49】讲什么故事：课件特意强调 SQA 要"督促测试工作的结果客观、准确和有效"，这句话说的是监督，落点其实是诚信。怎么联系知识点：测试向 SQA 提供质量评价的客观依据，数据一旦被修饰，整个质量判断就失去意义。落到什么价值观：真实是质量工作的生命线。企业里为了过一个质量门而修改测试结论、遮掩失败用例，短期看是"顺利交付"，长期看是拿用户和公司信誉做赌注。我们做课程设计也一样，测出来的问题如实写、测不到的地方如实标明范围，这才是合格的工程素养。

过渡：说清了测试和质量保证的关系，还有一个关系同样重要，而且更贴近日常——测试和开发的关系。

【1.50】1.5 测试与开发的关系（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页进入 1.5 节，标题是测试与开发的关系。同样先放一张图，把这一节的问题摆出来。为什么这个关系也值得单独讲一节？因为它直接决定测试在项目里被摆在什么位置。在不少团队和不少人的直觉里，开发是"造东西的"，测试是"检查东西的"，造完了才轮到检查，所以测试天然排在开发后面。这种想法听起来很自然，实际代价非常大：测试时间被排在最后，一旦前面延期，被压缩的永远是测试；缺陷暴露得晚，返工成本高；测试成了"把关的唯一一道墙"，压力大、效果差。这一节要纠正的就是这个排序。课件备注里仍然是刚才 1.3 节的四个小节作为回顾，说明现在我们正式转入 1.5。本节接下来用两页来对照：第一页讲"测试不是开发下

一道工序"，把传统的那条串联链摆出来，让大家看清它的危害；第二页讲"测试与开发是并行、协作的关系"，给出正确的组织方式。这两页合起来，也是本课程后面很多内容的思想基础——比如为什么要有测试左移，为什么单元测试要由开发来写，为什么缺陷修复过程需要开发与测试反复沟通。请大家带着一个问题听：如果测试不是最后一道工序，那它在需求阶段、设计阶段应该做什么？这个问题在第 52 页会给出答案。最后还要纠一个态度：这一节不是要否定开发，也不是要把测试抬到开发之上，而是纠正一种排序。开发和测试是同一支队伍里的两种专业能力，没有谁比谁低一等，谁先谁后也不是身份问题，而是时机问题。

板书： 1.5 测试与开发的关系 → 传统直觉：开发造、测试查，测试排最后 → 代价：时间被挤、缺陷晚暴露、返工大 → 本节两问：测试是下一道工序吗？测试与开发该是什么关系？→ 回顾：1.3.1—1.3.4

提问： 回想一下你做过的课程项目或小组作业，测试是安排在哪一步做的？结果如何？

预设回答： ①答"最后交之前随便点几下"——非常真实，也是本页要打的靶子。教师点评：不要批评，先问"那几次点出来的问题好改吗"，多数学生会说难改，因为结构已经定了。②答"边写边测"——好回答。教师点评：这正是第 52 页要讲的并行协作，可以先表扬并让他说明具体做法。③答"我们分工，有人专门测"——这是"流程上的顺序"和"人员上的分工"混在一起了。教师追问：专门测的那个人，是从需求阶段就参与，还是等着代码写完才开始？把混点拆开，学生才能分清"角色分工"和"阶段排序"是两回事。

易错点 / 考点： 第一，把"并行协作"当成口号，答不出在哪些阶段交换哪些信息；第二，把"测试不是下一道工序"理解成"测试不必在开发之后执行"，把介入时机和执行时机混为一谈；第三，误以为测试左移就是要测试人员去替开发写代码。判断口诀：活动提前介入、执行按版本推进、责任团队共担。

过渡： 先看最常见、也最需要纠正的那张图——把测试当成开发的下一道工序。

【1.51】测试不是开发下一道工序（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页只有一条链：需求定义、设计、编程、测试、交付，五个方框一字排开。大家注意，这条链本身没有错——从时间上看，需求确实在设计和编程之前，代码确实在测试之前。它错在哪？错在我们把它理解成"做完上一格才能进下一格，做完就没我的事了"。这条链最大的问题是可以被读成一条流水线：需求分析员交需求，设计师交设计，程序员交代码，测试员交测试报告，然后交付。每一格都只管自己那一环，前一格交出去就结束。这在软件里会带来三个后果。第一个后果，缺陷发现得晚。需求里的一处含糊、设计里的一处接口不一致，一路上没人质疑，一直走到测试阶段才被发现，这时候要改的东西已经扩散到设计、代码、数据、文档。第二个后果，返工量成倍放大。经济视角那一页我们讲过：发现越早，返工越小。第三个后果，最实

际——测试时间被压到最短。因为它在链条末端，前面每一格延期都会吃掉它的时间，而它又是最后一格，没有地方可以再往后推，于是只能压缩用例、压缩回归、压缩环境准备，最后连"测完"都做不到。真实世界里的教训不少。据本课程案例素材，波音星际客机的软件问题，反映出的就是测试被拆散、端到端验证不足；更早的火星探测器事故，则与模块接口没有做充分的集成测试有关。这些事故的共性不是"程序没写出来"，而是"没有人从整体上验证它"。所以这一页要建立的观念是：这条链描述的是**工作产品的依赖顺序**，不是**人员的篱笆**。你可以按顺序产出需求、设计、代码，但测试的思维和活动必须提前介入，从需求阶段就盯着每一样工作产品的质量。有同学会问：那测试到底放在哪？答案是：测试无处不在，下一张图会讲清楚。

板书：传统串联链：需求定义 → 设计 → 编程 → 测试 → 交付 → 误读＝"做完上一格就没我的事" → 三后果：①缺陷发现晚 ②返工成倍放大 ③测试时间被压缩（末位、无路可退）→ 案例：系统集成与端到端验证不足引发事故 → 正解：这是工作产品的依赖顺序，不是人员的篱笆

提问：为什么这条链上最容易"牺牲"的总是测试？如果不想被牺牲，测试应该在前面做哪些事？

预设回答：①答"因为它在最后，没有下游可以顺延"——准确。教师点评：这句话点出了排程上的本质，写进板书。②答"因为测试不重要"——错误但值得讨论。教师追问：如果测试不重要，为什么出了线上事故大家第一个问的是"测了没有"？引导学生发现：说它不重要，只是因为它"看起来不产出功能"。③答"前面做需求评审和设计评审，把问题提前拦下来"——很好的回答。教师点评：这正是测试左移，可以顺势让大家举一个"需求评审里发现的问题"的例子，把抽象概念落到自己的项目上。

易错点 / 考点：第一，把"测试不是下一道工序"误读成"测试不需要在开发之后进行"——测试的执行确实在产品可用之后，但测试的介入必须提前，这是两件事。第二，考试里常把"测试是开发完成后的最后一个阶段"当作正确说法，这是典型的错误选项。第三，写文档时把测试计划安排在编码结束后才启动，属于实践扣分点。判断三步法：问顺序（时间上有先后，正常）→ 问介入（活动是否提前，必须提前）→ 问人员（是不是各扫门前雪，不能）。

过渡：既然不该是一条单向流水线，那测试和开发到底该怎么排、怎么配合？最后一张图给出答案。

【1.52】测试与开发是并行、协作的关系（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是 1.5 节的结论：测试与开发是并行、协作的关系。这张图要大家看出的，是两条轨道同时向前推进，并且在每个节点上都在交换信息。我们把"并行"和"协作"分开讲。先说并行。并行指的是测试活动和开发活动在时间上重叠，而不是一前一后排队。怎么重叠？需求阶段，测试人员就参与需求评审，专门盯那几个问

题：这条需求能不能验证？验收标准写清楚了吗？有没有互相矛盾或者含糊的地方？这些正是后面写用例的依据。设计阶段，测试人员参与设计评审，关注接口约定、异常处理、数据边界，同时开始做测试需求分析和测试计划。编码阶段，开发人员自己写单元测试，测试人员准备测试数据和用例，双方对接口契约达成一致，甚至可以先把接口的测试用例冻结下来，等实现完成就能马上开跑。到了系统测试和验收阶段，测试人员主导，开发人员待命修缺陷，缺陷修复后由测试人员回归验证，形成闭环。再说协作。协作的意思不是"客气"，而是双方对同一个目标负责。测试人员不是来挑刺的，他的产出是质量信息，帮团队在交付前发现问题；开发人员也不是把代码一丢就完事，他要提供可测的版本、清楚的变更说明、真实的环境和日志。落到具体做法上有几条特别实用：一是缺陷报告要写得能被复现——步骤、数据、环境、期望与实际，缺一样开发就要来回问；二是修复后要说明影响范围，方便测试判断回归哪些用例；三是遇到需求歧义，一起找产品确认，而不是互相赌气。这套并行协作的机制，业内常叫测试左移和持续反馈，前面 1.3.4 风险视角里那句"持续反馈质量风险"讲的就是它。给大家一句话总结：开发和测试不是接力赛的上下两棒，而是同一辆车上的两个仪表——一个负责把车开起来，一个负责告诉你车况怎么样；只有并排看着同一块路，才知道什么时候该踩刹车。这也正是我们课程设计里要求团队协作、要求测试与开发文档互相配套的原因。

板书：测试与开发＝并行 + 协作 → 并行：需求评审介入（可验证性/验收标准）→ 设计评审介入（接口、异常、边界）→ 编码期：单元测试 + 用例与数据准备 + 冻结接口契约 → 系统/验收：测试主导，开发修复，回归闭环 → 协作：缺陷可复现、变更讲影响、歧义一起找产品 → 关键词：测试左移、持续反馈

提问：测试左移最直接的好处是什么？请从"缺陷的引入阶段"和"修复成本"两个角度各说一句。

预设回答：①答"越早发现问题，改动越小、成本越低"——正确，覆盖了成本角度。教师点评：再补一句引入阶段的角度——在需求阶段拦下一个歧义，等于同时拦住了由这个歧义派生出的设计错误、代码错误和一批用例。②答"测试人员可以早点开始写用例，后面轻松点"——只说到进度，没说质量。教师追问：如果测试人员只是在旁边写用例，不提需求问题，算不算左移？引导学生区分"提前干活"和"提前发现问题"。③答"左移就是让开发多写代码少测试"——方向反了。教师点评：左移不但不减测试，反而让开发承担更多的验证责任（比如单元测试），是总量增加、位置前移。

易错点 / 考点：第一，把"并行"理解成"同时开始、各干各的"，漏掉"协作"和信息交换这一半。第二，认为测试左移就是"测试提前介入执行"，忽略了需求与设计评审阶段的介入形式。第三，判断题常见"开发和测试是两个独立的、互不干涉的阶段"——错误；"测试人员只对产品负责，不对流程负责"——也不准确，因为测试结果同时是 SQA 评价流程的依据。记忆口诀：并行不排队，协作不甩锅；左移一步，省下十倍。

【思政落点 52】讲什么故事：这一页讲协作，最能体现一个团队的质量文化。怎么联系知识点：测试与开发

并行协作，意味着缺陷信息要如实、及时地在两边流动，谁的锅先放一边，先把问题解决。落到什么价值观：软件是集体作品，推责比缺陷更伤人。开发对测试隐瞒改动、测试对开发夸大问题，都会让团队失去互信；反过来，把"我们一起把这个缺陷解决掉"当成默认姿态，团队的质量水平和交付能力都会上一个台阶——这也是本课程团队协作与工程表达能力这一目标想训练的东西。

过渡：到这里，第 1 章关于"测试是什么、为什么必须测、测试与质量保证是什么关系、测试与开发是什么关系"的讨论就讲完了。下面我们把镜头拉近，进入软件测试的基本概念——测试的分类与级别、静态测试与动态测试、黑盒白盒与灰盒，看看一次具体的测试工作到底是怎么分类、怎么分层展开的。

【1.53】1.6 测试驱动开发的思想（1.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

好，同学们，我们进入本章的最后一节——**1.6 测试驱动开发的思想**。这一节不摆新概念，它承担的任务是**给前面五节收口，再把顺序翻转过来**。

大家回想一下，这一章我们一路走来，讲了测试的必要性、测试的价值、从五个视角认识测试、正向思维和逆向思维、测试与质量保证的分工、测试与开发的并行协作。你会发现，所有这些讨论都指向同一个结论：**测试的地位，必须从"事后把关"提到"事前定义"。**

那么问题来了，怎么在工程上把这个结论落地呢？靠喊口号吗？不是。靠的是一套人人都能执行的具体做法——**这就是测试驱动开发，英文缩写 TDD。**

我给大家先透一个底：这一节只有四页，但它是本章的压轴——**前面五节解决的是"测试重不重要"，这四页解决的是"测试放在哪里做"。**一个解决认识，一个解决动作。认识不对，动作就白做；动作不落地，认识就是空谈。

所以这一节回答的是本章最后那个问题：**测试这件事，到底该在什么时候开始？**

板书：黑板左侧竖写"需求 → 设计 → 编码 → 测试 → 交付"这条传统顺序；右侧并排写"**1.6 TDD：测试前移、顺序反转**"；中间画一个反向箭头，从"测试"指回"需求"，箭头旁写"← 提前介入"。

提问：TDD 这个名字里，"驱动"两个字最值得我们琢磨。它驱动的到底是什么——是驱动写测试的手，还是驱动写代码的手？换句话说，TDD 最有价值的产出，是那些测试用例本身，还是那条被反转过来的顺序？

预设回答：第一种回答会说"最值钱的是测试用例，攒下来就是回归测试的资产"——这个回答只对了一半，

要肯定它看到了**资产价值**，然后追问："如果只为了攒用例，那为什么不先写完代码再补用例呢？反正用例最后还是那些用例。"这一问就能把学生推到关键点上。第二种回答会说"最值钱的是那个先后顺序，顺序一变，人做事的思路就变了"——这是我们要的答案，要直接表扬："对，先写测试，你在心理上就被迫先回答'它应该做什么'，而不是'我做了什么'，这个心理转向才是 TDD 的命门。"第三种，也会有同学说"不就是把测试提前吗，我提前写就行了"——这是**典型的误读**，要点破：TDD 不只是时间上提前，而是要**先写一个会失败的测试**，看着它红、再让它绿，这个"红—绿"的节奏是有硬性要求的，下一两页我们就讲这个节奏。

易错点 / 考点：第一个混淆点，是把 TDD 当成"一种测试技术"——错，**TDD 是一种开发思想、一种设计方法**，它产出的东西是代码，不是测试报告。第二个混淆点，是把 TDD 和"先写测试再写代码"划等号——不够，还要求**测试先行、失败为先、小步快跑、随时可重构**。判断口诀记四个字：**"先、败、小、重"**——先写测试、允许失败、小步进行、随时重构。

过渡：名字听过了，位置也说清了，那 TDD 到底怎么个"测试在前、开发在后"？我们翻到下一页，看它那张最经典的循环图。

【1.54】测试驱动开发的思想（4.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页的标题还是"测试驱动开发的思想"，正文只有一行字，但这一行字是整节的定盘星——**TDD，test driven development，测试在前，开发在后。**请注意这八个字的分量：它不是"测试和开发同时进行"，更不是"开发完再补测试"，而是**测试必须排在整个动作序列的最前面。**

屏幕下半部分画的是经典的 TDD 循环图，我用大白话带大家走一圈，一共八步，一圈一圈地转。

起点叫"开始"，旁边写"**为新性能写一个测试**"。注意，还没有任何实现代码，先把"我期望它做到什么"写成一条测试。

第二步是"**编译**"，第三步是"**修订编译错误**"——这一步特别有意思：因为实现还不存在，测试根本编译不过。但大家注意，**这里的报错不是事故，是路标**，它明确告诉你"还差什么"。

第四步"**运行测试并发现错误**"，第五步"**编写代码**"——只写让这条测试通过所需要的最少代码。第六步"**运行测试并通过**"。第七步"**如果需要就进行重构**"。第八步，箭头回到开头，为下一个新功能再写一个测试，循环往复。

我把这个循环概括成四个字，大家记下来：**红、绿、重构。**测试失败是红，测试通过是绿，在绿灯的保护下整理内部结构，就是重构。

这里我必须讲透一个最容易讲错的地方：**为什么一定要先看到"红"？**因为如果一条测试你从没见过它失败，你就根本不知道它有没有在验证东西——它可能永远都是绿的，那你写了等于没写。这是 TDD 里最容易被跳过、也最不能跳过的一步。

再打个生活化的比方。你去体检，是先定好"血压低于 140 算正常"这个标准，还是等检查完了再回头说"哦，我这份报告看着还行"？当然是先定标准。**TDD 里那条测试，就是你先写下来的体检合格线。**写代码的过程，就是照着这条合格线去做。

还有一个词要解释：**重构**。重构不是"重写"，它的定义是**在不改变外部行为的前提下，改善代码的内部结构**。那问题来了，谁敢在项目里随手改结构？答案是：**手里有测试网的人才敢。**测试全绿，你改完再跑一遍，绿的就还敢继续走；绿变红了，说明你改坏了，立刻退回。**测试网是重构的安全绳**，这就是"测试驱动"这四个字最实在的收益。

最后提醒大家看图上的循环形状——它是一个**闭合的圆**，不是一条直线。直线意味着"做完就完了"，圆意味着**每一小圈都留下了一层测试**。转一圈加一个功能，也加一层回归保护。这就是为什么 TDD 出来的项目，越往后越不怕改。

板书：画一个顺时针圆圈，标八步：**为新功能写测试 → 编译 → 修编译错误 → 运行测试、发现失败（红）→ 编写代码 → 运行测试、通过（绿）→ 需要则重构 → 回到起点**。圆圈中央写三个大字：**红 · 绿 · 重构**。右下角补一句：**测试 = 先定好的合格线**。

提问：TDD 循环的第四步是"运行测试并发现错误"。假如我图省事，跳过这一步，直接写代码、写完了再一次性运行测试，会丢掉什么？请大家具体说出丢掉的到底是什么。

预设回答：第一种回答会说"丢掉了一次运行而已，问题不大"。这个回答要当场纠正，因为它把"运行"当成了成本：丢掉的是**对测试本身有效性的验证**——一条从没红过的测试，你没有证据证明它在真的检查东西，它可能只是恒为真的空断言。这时可以追问："怎么证明一条测试不是摆样子？"答案是："看它红过。"第二种回答会说"丢掉的是对需求的确认，就是那个编译不过、测试失败的阶段，本来是在逼我想清楚接口长什么样"。这个回答层次很高，要表扬并放大：**TDD 的第一个产物其实不是测试，而是接口设计**，是先想清楚"外面怎么用我"，再想"我里面怎么写"。第三种回答会说"省了时间，效率更高"。这个要借势讲代价：省下的是几十秒的运行，赔上的是**没有回归网**——后面每次改代码你都不知道有没有改坏别处，最终会付出更

大的返工代价，这跟我们前面讲过的"缺陷发现越迟、成本非线性增长"是同一条规律，只是缩短到了分钟级。

易错点 / 考点：三个高频错误。第一，**把"测试在前"理解成"测试文档在前"**——不是，是**可运行的测试代码在前**，必须是能编译、能执行的。第二，**忽略"红的"那一步**，直接写代码再补测试，这不是 TDD，这叫"测试稍后（test after）"，考试里经常拿它做干扰项。第三，**把重构和重写混为一谈**——重构**不改变外部行为**，重写会改变；多选题常把"重构=重写"设成错误选项。判断三步法：一看**有没有先写测试**，二看**有没有先看到失败**，三看**有没有在小步循环里积累回归测试**。

过渡：TDD 这条循环看着挺简单，但它不是哪个人凭空想出来的。它有明确的出身和出处——下一页我们就去看看，它最早来自哪个开发方法。

【1.55】TDD 的实践最早来自极限编程（4.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页的标题是一句结论：**TDD 的实践最早来自极限编程。**极限编程的英文是 Extreme Programming，缩写 XP，它是 1990 年代末敏捷运动中非常有代表性的一种开发方法。TDD 并不是哪本教材坐在办公室里设计出来的，它是**在 XP 的工程实践中先做出来、后被总结成思想的**。这一点很值得让学软件的同学记住：**工程方法往往先有实践，后有理论。**

屏幕上这张图是课件里给出的 XP 实践流转图，信息量很大，我带着大家一步一步读。

我们从最左边开始。起点是"**下一个任务或失败的验收测试**"——注意，一上来就有"失败"两个字，这说明**这个流程是被"还没满足的东西"推着往前走的**，不是被"已经做完的东西"推着走，这个出发点就决定了它的性格。

接着箭头分成两层：**简单设计**指向 **CRC 卡**，**复杂问题**也指向 CRC 卡。CRC 就是 Class-Responsibility-Collaborator，类－职责－协作者卡片，是一种很轻量的设计工具。它要说明的是：**不管问题简单还是复杂，设计这一步都不能省，只不过用的工具可以很轻。**

再往右，从"下一个任务或失败的验收测试"出发，**结对**之后"**创建一个单元测试**"。这里"结对"两个字很关键——**两个人一起干，一个人写、一个人看**，这是 XP 的标志性做法，也正好回应了我们马上要讲的那个问题：**一个人测自己写的东西会有什么毛病。**

接下来进入中间那个核心循环，我们看它的每一步：**创建单元测试 → 修订代码 → 失败或通过的单元测试 → 不断的集成 → 运行所有单元测试 → 100% 单元测试通过。**

大家仔细看"修订代码"这个节点，它下面挂着一个词叫"**代码重构**"，而且重构旁边标了两条："**简单代码**"和"**复杂代码**"。这个细节很值得讲：**简单代码就直接改，复杂代码就要靠重构来梳理**——但两条路都通向同一个目标，就是让代码保持可维护。

图上还有几条反馈回路，我也点一下：一条向上写"**改变结构、对人员**"，一条写"**人员调整**"，再往上一条写"**我们需要帮助**"。这几条线说明 XP 不只是技术实践，它同时管人：**结构不合适就调结构，人干不动就调人，搞不定就喊帮助。**这是一种非常坦诚的团队文化。

再看最右边：**新的单元测试 → 新功能** 这条线，是"不断集成"时新功能带来的新测试；而最上方一条弧线写着"**运行失败的验收测试**"，从"100% 单元测试通过"绕过去，最后落到"**通过验收测试**"。这一段是整个流程的出口——**单元测试全绿只是内部过关，验收测试通过才叫对外交付。**

板书：左侧竖排写三条主线——**① 任务/失败的验收测试 ② 简单设计与复杂问题 → CRC 卡 ③ 结对 → 创建单元测试**；中间画核心循环：**创建单元测试 ⇄ 修订代码（重构：简单代码/复杂代码）→ 不断集成 → 运行所有单元测试 → 100% 通过**；右侧写出口：**100% 单元测试通过 → 运行失败的验收测试 → 通过验收测试**；最上方批注三条旁线：**人员调整、改变结构、我们需要帮助**。

提问：这张流程图的入口写的是"下一个任务**或失败的验收测试**"。为什么一个开发流程，要用"失败"来当起点？换成"下一个要开发的功能"，不是更顺耳吗？

预设回答：第一种回答会说"因为还没有实现，所以测试当然是失败的，这是正常状态"。这个回答已经摸到门了，要顺着往下压一句："对，但请你再说深一层——**把'失败'摆在起点，等于承认了'不满足'才是工作的常态**，这和你习惯的'我做完一个就勾一个勾'是完全不同的心态。"第二种回答会说"用失败来驱动，能保证每一步都是有目标、可验证的"。这个回答很好，把它升级成一句板书：**“从「还没满足」出发，每一步都有一个可以说清楚的判据。”**第三种是典型的抵触回答："这不就是先写个必然失败的测试，多此一举吗？"这时候要正面接住：这不是多此一举，恰恰是因为**人天生倾向于保护自己的成果**，先写失败测试，就能把"我写得对不对"这个主观判断，换成一个客观的、机器来判的绿灯/红灯。

易错点 / 考点：第一，**极限编程（XP）** 这个名字的写法与归属要记准——考题常问"TDD 最早来自哪种开发方法"，答案就是**极限编程（Extreme Programming）**。第二，**CRC 卡（类－职责－协作者）** 是设计工具，容易被误当成编码工具或测试工具。第三，**"100% 单元测试通过"与"通过验收测试"是两个关卡**，不能合并——单元测试是内部的、盯代码的，验收测试是对外的、盯需求的，这个区别在第 56 页还要再往前推一步。

过渡：讲到这儿大家应该发现了，TDD 在 XP 里是"一个动作"。后来它长大成了一种"思想"，还分出了两个

> 不同的实践层次。下一页，我们就把这三个名字摆在一起说清楚：TDD、UTDD、ATDD。

【1.56】TDD 与 UTDD、ATDD（4.5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页的标题是"**TDD 与 UTDD、ATDD**"，页面上还写着一行总结性的话，我先把这行话念清楚：**TDD 成为思想，UTDD 单元测试驱动开发、ATDD 验收测试驱动开发则成为实践。**请大家把这句话记在笔记最上面，因为考试里最爱考的就是这三个词的分工。

我们先看屏幕这张图。它是两个同心圆：**里面那个小圆标着 UTDD**，圆里画的是"**单元测试**"和"**代码**"两个方块，两个方块之间是**双向箭头**；**外面那个大圆标着 ATDD**，大圆里面包含了整个小圆。

大圆左上方还有一个"**验收指标**"的椭圆，它用箭头指向"**验收测试**"；同时它还有一条虚线连到左下角那个**笑脸**图标上。另外还有一条虚线，从笑脸连向"验收测试"的方向。

这张图的读法，我教大家一句话：**思想在圆心，实践分两层；内圈管代码，外圈管指标。**

先说**内圈 UTDD**，Unit Test Driven Development，**单元测试驱动开发**。它的对象是"代码"这个最小的可测单元，做法是先用单元测试把"这个函数应该输入什么、输出什么"钉住，再去把代码写出来，然后让测试变绿。**注意那两个方块之间是双向箭头**——这说明单元测试和代码之间是**相互塑造**的关系：测试约束了代码的行为边界，代码的实现细节又会反过来催生新的测试。

再说**外圈 ATDD**，Acceptance Test Driven Development，**验收测试驱动开发**。它站在比单元更高的位置：从"**验收指标**"出发，先把"这个东西要满足什么条件才算交付合格"讲清楚，再把这些指标变成**验收测试**。大家看图上的位置关系——**ATDD 这个大圆把 UTDD 整个装在里面**，这个包含关系非常重要：它说明**验收测试驱动开发并不排斥单元测试，它是把单元测试包在自己的循环里**。业务上的验收条件往下分解，落到代码层面就变成一条条单元测试。

最后说**圆心上的 TDD**。为什么说 TDD 是"思想"而不是"实践"？因为思想只有一个内核——**测试先行、以测试来驱动设计与实现**；至于在哪个层次上先写测试，那是实践层面的选择：**在最细的代码层次上先写测试，就是 UTDD；在验收指标的层次上先写测试，就是 ATDD。**两个实践的出发点不同、颗粒度不同、面向的人也不同，但它们共享同一个思想内核。

图上那个笑脸，我理解课件想表达的是"**最终结果让使用者满意**"——验收指标是要对着"用户满意"这条线来定

的，而不是对着开发自己方便来定的。这个视角一摆正，ATDD 和 UTDD 的分工就清楚了：**UTDD 保证里面是对的，ATDD 保证外面是用户要的。**

板书：先画一个大圆标 **ATDD（验收测试驱动开发）**，圆内再画一个小圆标 **UTDD（单元测试驱动开发）**；小圆内并排写"**单元测试 ⇄ 代码**"；大圆内小圆外写"**验收指标 → 验收测试**"，并引一条线连到"**用户满意**"；大圆圆心处写一个大字："**TDD＝思想（测试先行）**"。旁边竖排口诀：**思想在圆心，实践分两层；内圈管代码，外圈管指标。**

提问：假设一个团队说"我们做的是 TDD，但我们只写验收测试、不写单元测试"。按这一页的图来判断，他们的话站得住吗？为什么？

预设回答：第一种回答会说"站得住，因为 ATDD 把 UTDD 包在里面了，做外圈自然就包含内圈"。这个回答**混淆了"包含"和"替代"**，必须点破：包含关系说的是**范围**，不是说**自动等价**——外圈大并不等于内圈可以被跳过，图里小圆是**实打实画在里面的**，那意味着单元测试这一层仍要存在。第二种回答会说"站不住，ATDD 和 UTDD 是两个不同颗粒度的实践，一个管验收指标、一个管代码细节，验收测试过了不能说明每个单元的逻辑分支都对"。这是标准答案，要追一句把考点夯实："对，所以这两层是**互补**，不是**互替**。"第三种回答会说"只要产品能用，写不写单元测试无所谓"。这个回答在工程上很危险，正好用它讲代价：没有单元测试，缺陷一旦在集成阶段暴露，**定位成本会成倍上升**，因为你不知道是哪一块坏了——这又是我们第一章反复出现的那条规律：**越晚发现，代价越高。**

易错点 / 考点：第一，**三个缩写的全称必须能默写**：TDD＝Test Driven Development，UTDD＝Unit Test Driven Development（单元测试驱动开发），ATDD＝Acceptance Test Driven Development（验收测试驱动开发）。第二，**"TDD 是思想，UTDD 和 ATDD 是实践"**这句话是原句，判断题常把"TDD 是一种实践"设成错误选项。第三，**内圆外圆的关系是包含，不是并列，也不是替代**——画图题常考：请把 TDD、UTDD、ATDD 用一个图表示出来，标准答案就是**同心圆**。第四，别把 ATDD 和"验收测试"混为一谈：ATDD 强调**驱动**，也就是**在开发之前就用验收标准来牵引**，它是动作；验收测试是**最终执行的检验**，它是结果。

过渡：这一章讲了这么多，从测试的价值讲到 TDD 的思想，最后我要把一个最尖锐的问题摆在大家面前——**让开发人员测自己的产品，到底行不行？TDD 凭什么能解决它？**下一页就是这个问题。

【1.57】问题（5.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页没有别的，页面上就是两个字——"**问题**"。下面两行小字是两个问句：第一，**如果开发人员测试自己的产品，会有哪些障碍？**第二，**TDD 又是如何克服这些障碍的？**这一页是我们第 1 章的思考高点，也是考试、答辩都爱问的地方，我按参考答案的口径，把它一层一层讲透。

先说第一个问题，障碍分三层，我们逐层看。

第一层，是思维盲区。人写代码的时候，脑子里已经装着自己那套实现思路。等到回头测试，他会不自觉地**顺着自己的实现思路去测**。这一顺，问题就来了——**他只会去走自己"想到过"的路径，而缺陷恰恰最爱藏在"他没想到"的路径和输入组合里。**说白了，**用你自己的思路去找自己的错，就像一个从来没迷过路的人去画地图，他会把走过的那条路画得很细，把没走过的岔路口漏掉。**

第二层，是心态偏差。这一点更隐蔽、也更普遍。人在面对自己亲手做出来的东西时，**天然倾向于证明"我写的是对的"**。这不是道德问题，这是心理机制。所以同样一条失败的测试，别人跑出来会追问"代码哪里错了"，自己跑出来第一反应往往是"是不是测试写错了"。**这个倾向会让缺陷被解释掉、被绕过、被改成"设计如此"。**这里我要提醒一句：这才是"开发人员不适合测自己产品"这句话的真正含义——不是能力不够，而是**立场天然不中立**。

第三层，是独立性与时间不足。独立性说的是：**开发和测试最好不是同一个人**，这样评审视角才客观，这在测试组织里叫"独立性原则"。时间不足说的是现实里的排期——开发任务压得最紧的时候，最先被砍掉的往往就是自测和用例设计，因为**它们不直接产出新功能，交付压力一来，第一个被牺牲的就是它们。**

现在讲第二个问题，**TDD 是怎么把这三层障碍逐一化解的**。它一共给了我们三件武器，正好对着三个障碍。

第一件武器，对应思维盲区：先写测试、再写实现。这个顺序的意义在于，它**逼着你从需求与接口出发**——因为写测试的时候，实现代码一行都还没有，你手里只有"这个功能应该表现成什么样"，你只能从外部往里看。**这就把"顺着实现思路测"的路给堵死了。**

第二件武器，对应独立性与时间不足：测试先行形成自动回归网。测试是在写功能之前就落下来的，它不会等到交付前才被临时砍掉——**它已经变成了代码库的一部分。**而且这个回归网是自动的，随时能跑，这就把"没时间测"从一个主观借口，变成了一个客观事实：**你的测试覆盖了多少，跑一遍就知道。**

第三件武器，对应心态偏差：用例就是"可执行规格"，减少主观判断。这是三件武器里最漂亮的一件。原来"这算不算缺陷"是个**主观问题**，靠人吵；现在它变成了**客观问题**——测试通过还是不通过，机器说了算。**规格是可执行的，判断就不再有解释空间。**所以 TDD 不只是让测试更早，它是**把"质量由谁说了算"这个问题，从人手里交给了测试。**

最后我给大家一句总纲，请记牢：**障碍的根子是"自己人评自己"，TDD 的解法是"用先写下来的客观标准去评"。**先把期望写成测试，再把实现写出来——立场问题被顺序解决了，盲区问题被外部视角解决了，时间问题被自动化解决了。

板书：左侧写"**障碍三层**"：① **思维盲区**——顺着自己的实现思路测，漏掉没想到的路径与组合；② **心态偏差**——倾向证明"我写的是对的"；③ **独立性与时间不足**。右侧写"**TDD 三件武器**"，用箭头一一对应：① 先写测试再写实现 → **迫使从需求与接口出发**；② 测试先行 → **自动回归网，用例就是可执行规格**；③ 用例即规格 → **减少主观判断**。底部写总纲：**用先写下来的客观标准，去评自己写的东西。**

提问：三个障碍里，"心态偏差"是最难承认的一个。请你举一个你身上真实发生过的场景——你是怎么在"发现自己的东西不对"的那一瞬间，说服自己"其实这样也行"的？

预设回答：第一种回答是典型的："我写作业的时候跑出来结果不对，第一反应是检查输入数据，改了几次数据凑出正确答案，就交了。"这个回答太真实了，要立刻接住并顺势把它变成教学素材："**注意他刚才做了什么——他改了输入去迁就输出。**这就是心态偏差最标准的表现形式：不是改代码，是改标准。而 TDD 的做法刚好反过来，**标准是事先锁死的，你只能改代码。**"第二种回答会说"我会跟自己说这个边界情况不常见，用户不会这么用"。这也是极好的素材，要追问："那你凭什么判断用户不会这么用？"答案往往就是"凭感觉"——那就点破：**"凭感觉"正是主观判断，而 TDD 就是把这条"凭感觉"的路用测试堵住。**第三种，也会有同学很硬气地说"我从来都要求自己严格自测，不存在这个偏差"。这时候不要否定他，要用提问引导："好，那我问你——**你怎么证明你测到了你没想到的地方？**"这个问题他答不上来，因为**你无法用"想到"去覆盖"没想到"**，这恰恰是外部视角和自动化回归网不可替代的原因。

易错点 / 考点：第一，**"开发人员不能测自己的产品"这句话的标准理由，是"独立性"与"客观性"，不是"水平不行"**——选择题常把"开发人员技术能力不足"设成错误选项。第二，**TDD 克服障碍的机制要能对上号**：先写测试对应"思维盲区"，自动回归网对应"时间与独立性"，可执行规格对应"心态偏差"，这三条是**一一对应**关系，不要交叉。第三，**别忘了"用例即规格"这个提法**——它把主客观判断的边界讲清楚了，简答题里是加分点。判断三步法：**谁在评？评什么？用什么标准评？**——TDD 的答案是"用事先锁死的外部标准评实现"。

过渡：这一章从"为什么要测试"一路讲到"TDD 怎么解决自测的障碍"，到这里，1.1 到 1.6 的六节内容就全部讲完了。下面我们用一页时间，把这一章的五个理解点收拢起来。

【1.58】本章小结（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

好，这一页是**本章小结**。屏幕上列了五条，我就按这五条，一条一条把第 1 章的要点钉一遍。大家注意，**这五条里的每一条，都对应着我们这一章前面讲过的一整节**，所以我讲的时候，你们可以顺手在心里回放一下那一节的关键句。

第一条，理解测试的定义与价值。测试是什么？按国际标准的说法，是在特定条件下运行系统或组件，观察或记录结果，对它的某个方面做出评价。而测试的价值有四个层面：**全面评估产品质量，拿到客观的质量信息；发现问题、督促问题解决，从而提高质量；持续提供质量反馈、及时揭示质量风险；通过缺陷分析得到缺陷模式，进而预防缺陷。**大家看，从"评估"到"发现"到"反馈"到"预防"，这四个词其实是一条**由被动到主动的上升线**——测试走到最高层，不是抓错，而是**让错误不再发生**。

第二条，从不同视角认识软件测试。我们这一章一口气给了五个视角：**质量视角**，测试是对软件质量的全面评估，给出质量信息；**风险视角**，测试是对系统潜在质量风险的评估，而且因为测试是抽样实验、不可能穷尽，风险总是存在的；**经济视角**，测试追求以最小代价获得最高质量，所以**测试成本必须小于缺陷造成的损失，测试才有意义**；还有 **Test Oracle 视角**，输出是否正确需要有判断准则；以及**批判性思维视角**，测试是一个借助观察、经验、反思、推理去收集信息、不断探索的过程。五个视角，其实是五把尺子——**同一件事，换个视角看，你关注的东西完全不同。**

第三条，正向思维、逆向思维，也会决定测试的行为。这一条是本章的思想核心。正向思维的代表是 Bill Hetzel 博士，认为测试是**为程序能按预期运行而建立信心的过程**；逆向思维的代表是 Glenford J. Myers，讲得更锋利——**测试是为了证明程序有错，而不是证明程序无错；一个好的测试用例在于它能发现至今未发现的错误；一个成功的测试是发现了至今未发现的错误的测试。**落到行为上就是一个对比：正向思维是"**在设计规定的环境下运行软件的所有功能，直至全部通过**"；逆向思维是"**寻找容易犯错误的地方和系统的薄弱环节，试图破坏系统，直至找不出问题**"。**认知决定行为**——你心里假设它是对的，你就在走过场；你心里假设它有错，你才会去挑刺。

第四条，理解测试、SQA、质量、开发之间的关系。SQA 是一项**管理工作**，侧重于对流程的评审和监控，它指导、监督测试的计划与执行；测试是一项**技术性工作**，侧重对产品进行评估和验证，**测试是 SQA 的重要手段之一**，为 SQA 提供质量数据。至于开发，**测试不是开发的下一道工序，两者是并行、协作的关系**——需求、设计、编码的每一步，测试都在场。

第五条，理解先进的 TDD 思想。一句话：**测试在前，开发在后**；它最早来自极限编程的实践，TDD 是思想，UTDD 和 ATDD 是它的两种实践；它靠"先写测试、看到失败、再写实现、然后重构"的循环，把"自己人评自己"这个根子上的障碍给拆掉。

最后我给大家留一句本章的总结：**这一章我们学到的不是"怎么测"，而是"该在什么时候、以什么心态去测"。**答案是：**越早越好，并且先假设它有错。**

板书：竖排五条：**① 定义与价值**（评估 → 发现 → 反馈 → 预防）；**② 五个视角**（质量 / 风险 / 经济 / Test Oracle / 批判性思维）；**③ 正反思维决定行为**（Hetzel 正向 vs. Myers 逆向）；**④ 测试—SQA—开发**（管理 vs. 技术；并行而非下道工序）；**⑤ TDD**（思想在内，UTDD/ATDD 在外）。黑板最下方写本章一句话：**越早越好，并先假设它有错。**

提问：这五条里如果要你只留一条带走，你留哪一条？请用一句话说出来，并说明你为什么选它。

预设回答：第一种回答会选第①条"定义与价值"，理由是"这是基础，后面全都要用它"。这个回答稳妥，要肯定，但可以把它的层次往上推一层："基础确实重要，不过你再想想——**定义讲的是'测试是什么'，心态讲的是'测试的人是谁'**，哪一个更决定成败？"第二种回答选第③条"正向逆向思维"，理由是"这一条最反直觉，也最能改变我的做事方式"。这就是我们要的答案，要直接落到板书上："对，**正向思维是「验证它没问题」，逆向思维是「证明它有问题」，这两种心态下写出来的用例，质量完全不是一个量级。**"第三种回答也会有人说"我选第⑤条 TDD，因为它最具体，能马上用"。这也很好，要顺势补一句：TDD 之所以能落地，正是因为它是把第③条那种逆向心态，**固化成了流程动作**——思想一旦变成动作，就不会因为人的惰性而丢掉。

易错点 / 考点：第一，**"测试是 SQA 的重要手段之一"与"测试就是 SQA"要分清**——这是判断题的高频陷阱，两者是**手段与体系**的关系，不是等同关系。第二，**正反思维的代表人物别归错**：正向是 **Bill Hetzel**，逆向是 **Glenford J. Myers**。第三，**经济视角的那句不等式要能默写**："**测试的成本 < 缺陷造成的损失，测试才有意义**"，考试里常把不等号方向设成错误项。第四，**"测试不是开发的下一道工序"**这个判断，是选择题的常客。记忆口诀对第③条尤其有用，四个字：**"假设有错"**——答题时先写这四个字，再展开正反两面的行为差异。

过渡：课本上的内容到这儿就总结完了，但学习不能停在"听懂了"。下一页我们留几道题，检验一下这一章是不是真的学进去了。

【1.59】提问、思考与练习（5.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

好，最后一页练习——**提问、思考与练习**。屏幕上有四道题，我要提醒大家一句：**这四道题不是随便凑数的问题，它们的答案基本就是我们这一章所有考点的浓缩，期末和课程设计答辩里都可能以不同形式出现。**所以我不念题、等你们答，我直接把讲法和答案都讲透，你们对照着检查自己的思路。

第一题：在日常使用软件的过程中，遇到哪些软件质量问题？这道题看起来随便，其实考的是"能不能把生活现象对应到缺陷类型"。我按参考答案的口径归类：**闪退、卡顿这类，属于性能与可靠性问题；数据丢失或者计算结果错误，属于功能与数据缺陷；界面错位、乱码，属于兼容性与易用性问题**；还有一类特别常见——**更新之后老功能失效了，这叫回归缺陷**。请大家留意这四类的名字，**它们就是缺陷分类的入门口径**，也是你们第一次作业要用的标签。我建议你们回去建一个"质量问题随手记"，连续记一周，把遇到的每个现象都往这四类里归一次——**归不上类的，那本身就是个值得讨论的例子。**

第二题：软件测试的正反两方面观点，会如何影响测试工作？这道题考的是第 1.3.2 节。**正向思维是按规格去验证，"该做的都做到了"；反向思维是先假设程序有错，专门挑边界、异常输入和组合去证伪。**这两个思维落到具体工作上，差别非常具体：同样一个"登录"功能，正向思维会写"输入正确账号密码，登录成功"；反向思维会写"空密码怎么办、超长输入怎么办、特殊字符怎么办、连续错十次怎么办"。**一个是照着需求清单打勾，一个是照着'哪里可能出错'去挖。**你们做实验的时候，我会明确要求**用例里必须有一半以上是反向用例**，就是这个道理。

第三题：软件测试和软件开发的关系是怎样的？如何更好地利用这种关系？这道题背后是我们讲的"**测试不是开发的下一道工序**"。**它们是并行、协作的关系**：需求的定义、设计、编程、交付，每一段测试都在场。那怎么"更好地利用"呢？答案就是**测试左移**——**在需求和设计阶段，测试就介入进来。**我举两个最实在的例子：需求评审的时候，测试人员专门负责挑**二义性**，一句话如果有两种读法，就当场问清楚；接口还没实现的时候，先把**接口契约和用例冻结下来**，让开发和测试照着同一份规格各干各的。**你越往前介入，你能拦住的问题就越多，返工就越少**——这正是我们第 1.2 节讲过的经济视角。

第四题：软件测试和质量保证之间的联系和区别？这一题最容易答糊，我用一句话把它钉死：**SQA 面向过程，测试面向产品。**SQA 关心的是"**做事的过程合不合规矩**"——规范有没有执行、评审有没有开、度量有没有做、过程有没有在改进；测试关心的是"**做出来的东西对不对**"——验证和确认，把质量信息交出来。**联系在于：测试是 SQA 的重要手段之一，测试跑出来的数据，是 SQA 判断质量、评价过程的客观依据；反过来，SQA 会指导、监督测试的计划和执行，督促测试的结果客观、准确、有效。**大家看，一边是管理体系，一边是技术

手段，它们是**相互支撑**的。答题三步法也给大家：**先判属性（管理还是技术）→ 再比侧重（流程还是产品）→ 最后说产出（过程改进的建议，还是质量信息与缺陷数据）。**

四道题讲完了。屏幕备注里写着"重点学习内容"，确实如此——**这四道题，就是这一章的四根支柱。**

板书：左侧写四个题号；右侧写答题关键词：**① 质量问题四归**（性能与可靠性 / 功能与数据 / 兼容与易用性 / 回归）；**② 正反思维**（按规格验证 vs. 专挑边界与异常证伪）；**③ 并行协作 + 测试左移**（需求评审挑二义性、接口契约与用例先行）；**④ SQA 面向过程 vs. 测试面向产品**（测试是 SQA 的重要手段）。底部写答题三步法：**判属性 → 比侧重 → 说产出**。

提问：用第④题的框架考你们一下——如果我发现"这次迭代的缺陷特别多"，这个信息应该反馈给谁？是让测试把用例写得更狠一点，还是该去查一查过程和规范哪里出了漏洞？

预设回答：第一种回答会说"当然先让测试补用例，先把 bug 都抓出来"。这个回答只做了一半，要肯定它的紧迫性，同时提醒：**缺陷多是一个结果，不一定是一个原因**。追问一句："如果同一类缺陷反复出现，补用例能解决它第二次出现吗？"——这就是把话题引到 SQA 上。第二种回答会说"两边都要做，短期靠测试补齐覆盖，长期要靠 SQA 去查根因、改过程"。这是最完整的答案，要表扬并帮他说完："对，**测试解决这一次，SQA 解决下一次**，这正好就是 SQA 要做度量、要做过程改进的原因。"第三种回答会说"缺陷多是开发的问题，跟测试无关"。这个回答要温和地纠偏：**缺陷多也反映测试自己的覆盖与准入把关**——测试有没有在需求阶段就介入、有没有把准入标准卡住。这也正好回应第③题"测试左移"的价值。

易错点 / 考点：第一，**第①题的分类口径要能对上号**——"闪退卡顿"对应**性能与可靠性**，不要答成功能缺陷；"更新后旧功能失效"对应**回归缺陷**，不要答成兼容性。第二，**第②题千万别把正反思维答成"好与坏"**——它们不是对错关系，是**两种互补的思维取向**。第三，**第③题的关键词是"并行""协作""测试左移"**，如果只答"测试在开发之后进行"，整题零分。第四，**第④题最忌把 SQA 和测试说成"一个东西的两种叫法"**，一定要落到"**过程 vs. 产品**"和"**手段 vs. 体系**"这两组对比上。

过渡：四道题答完，这一章的知识就闭环了。学完一门课，光靠课件不够，最后我给大家推荐三本可以继续读下去的书。

【1.60】学习资源推荐（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是**学习资源推荐**，课件上列了三本。我一本一本说，告诉大家**什么时候该翻哪一本**，这样你们回去才用得上。

第一本，《软件测试方法和技术（第 4 版）》，清华大学出版社，2022 年。这就是我们这门课的教材，也是我们所有术语口径的来源。我提醒一句：**我们课上讲的每一个概念，最后都要回到这本教材的说法上**，尤其是定义、分类、流程这些容易有不同版本表述的地方。所以大家备考的时候，**以这本第 4 版为准**，不要去网上随便找一份说法就背下来。这本书的结构我们也已经在讲第一章的时候带大家看过一遍了——**原理与方法、技术、项目实践这三篇**，正好对应我们这门课"理论加实践"的安排。

第二本，《全程软件测试（第 3 版）》，人民邮电出版社，2019 年。这本书的名字里有两个字最关键——"**全程**"。它讲的不是"测试阶段要做什么"，而是**测试怎么贯穿软件开发的整个过程**。你们现在听完第 1 章应该已经有感觉了：我们这一章反复强调的就是**测试左移、测试与开发并行**。等你做课程设计的时候，要写测试计划、要安排测试介入的时机，这本书会给你很完整的参照。我把它定位成**"工程实践视角的补充读物"**。

第三本，《致命 Bug 软件缺陷的灾难与启示》，人民邮电出版社，2016 年。这本书是本**案例集**。前面两本讲"应该怎么做"，这一本讲"**没做好会怎样**"。我们课上提到过的那些真实事故——放疗仪致人死亡、处理器浮点除法错误导致大规模召回、航天探测器因为接口没有做集成测试而坠毁，这一类案例在这本书里有更完整的来龙去脉。我要专门说一句：**这一本对你们最有用的场景，是写课程设计报告里的"缺陷影响分析"，以及答辩时回答"这个缺陷如果不修会有什么后果"。**

三本书的分工，我给大家总结成一句：**教材用来定口径，全程测试用来学工程做法，致命 Bug 用来长教训、找案例。**

最后补一句自己的建议：书不用一次看完。**未来两周，请先把教材第 1 章重读一遍，边读边把你在生活里遇到的软件质量问题记进那个"随手记"里。**这一步做到了，这一章就算真的落地了。

> **板书：**竖排三本，每本后面批注用途：①**《软件测试方法和技术（第 4 版）》（2022，清华）→ 定口径 / 备考依据**；②**《全程软件测试（第 3 版）》（2019，人邮）→ 全程视角 / 写测试计划的参照**；③**《致命 Bug：软件缺陷的灾难与启示》（2016，人邮）→ 案例库 / 缺陷影响分析与答辩素材**。旁边写一句：**教材定口径，全程学做法，Bug 书长教训。**

> **提问：**这三本书里，如果你现在只能选一本、并且只有两周时间，你会选哪一本来支撑你的课程设计？请连理由一起说。

预设回答：第一种回答选第①本教材，理由是"考试以它为准，先把口径对齐"。这个选择很务实，要肯定，同时补一句："口径对齐之后，别忘了**测试计划是要写给别人看的**，所以第②本迟早要翻。"第二种回答选第②本《全程软件测试》，理由是"课程设计要写测试计划、要安排测试介入时机，这本最直接"。这个判断很到位，要顺势提醒一个风险："工程做法好学，但**术语口径如果跟教材不一致，答辩时容易被追问**，所以两本要配着看。"第三种回答选第③本《致命 Bug》，理由是"案例最好看、也最好用"。这个回答要鼓励但要纠偏：**案例的价值在于你说得出"启示"**，如果只会复述事故经过，报告里那一段是撑不起来的——**要把事故对应回本章的知识点**，比如"这正是缺少集成测试的代价""这正是经济视角下测试成本小于损失的反面案例"，这才叫把案例用活了。

易错点 / 考点：这里不是背书的出版社和年份，而是要建立"**什么场景查什么资料**"的判断力。唯一需要精确记忆的是**教材的版本信息**：《软件测试方法和技术（第 4 版）》，清华大学出版社，2022 年——**课程所有概念的最终口径以它为准**。另外提醒两点：一是**别把"全程软件测试"和"软件测试方法和技术"混为一谈**，前者强调测试贯穿全程的工程视角，后者是本课程的系统教材；二是**引用案例时一定要写出启示**，只写事故描述在报告里是不合格的。

过渡：这一章我们从"为什么要测试"一直讲到"测试该在什么时候做"，中间经过了测试的价值、五个视角、正反思维、SQA 的分工、开发的协作，最后落在 TDD 上。课就上到这里，我最后说两句收尾的话。

【1.61】感 谢 聆 听（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

好，同学们，这一章到这里就讲完了。我用最后两分钟把第 1 章收个尾。

按课件备注里给的口径，我在这里应该说的话是："**以上是本节课的内容**""**这一章讲到这里**"——这两句话我今天就照着说：**以上是本节课的内容，第 1 章"引论"讲到这里。**

我们从头回顾一下这一章走的路：我们先问了"**为什么必须做软件测试**"，用真实事故说明**缺陷的代价是真实的、而且发现得越晚代价越高**；接着我们从**质量、风险、经济、Test Oracle、批判性思维**五个视角认识了测试；然后讲了**正向思维和逆向思维**——它们不是对错之分，而是两种会直接决定你测试行为的取向；再往下区分了**测试与 SQA**：一个是技术、一个是管理，一个面向产品、一个面向过程；又讲了**测试与开发**：不是上下道工序，而是并行协作，要**测试左移**；最后落在 **TDD 思想**上——**测试在前，开发在后**，用先写下来的客观标准，去评自己写的东西。

这一章的名字叫"引论"，我希望大家现在就记住这一章真正留下的东西，它不是某个定义，而是**一个态度**：**面对软件，不要假设它是对的；作为测试的人，你的价值不在于证明它对，而在于比别人更早发现它不对。**

课下的两件事我再说一遍：**第一，整理 5 个真实软件缺陷事故，标注类型；第二，用自己的话写出软件测试的定义，并说明它和调试、质量保证的区别。**同时别忘了装好实验环境，完成第一次测试体验——**找出给定小程序里至少两个缺陷，截图记录下来。**

我们下一次课讲**第 2 章 软件测试的基本概念**，会讲清楚测试的分类、测试的级别、静态测试和动态测试、黑盒白盒灰盒。**带着今天这个"先假设它有错"的心态来，第二章会更好懂。**

周宇文，广东金融学院。**以上就是本节课的全部内容，谢谢大家，下课。**

板书：正中间写"**感谢聆听**"；右上角竖排写本章六节回放：**1.1 必要性 → 1.2 价值与成本 → 1.3 定义与正反思维 → 1.4 SQA → 1.5 测试与开发 → 1.6 TDD**；左下角写本章一句话态度：**不假设它对，只求更早发现它不对**；右下角写作业与预习：**作业：5 个事故 + 定义辨析；实验：找 2 个缺陷并截图；预习：第 2 章 测试的分类与级别。**

提问：这一章我们从事故讲到了 TDD。我想请大家用一句话回答：如果只能从这一章带走一样东西，你希望带走的是哪一个判断——是"测试很重要"，还是"测试要早"，还是"测试要抱着怀疑的心态"？请说你选它的理由。

预设回答：第一种回答会说"测试要早，因为成本是越晚越高的，早就是省钱"。这个回答很实在，要肯定它的经济逻辑，同时补一层："**早**是效率层面的答案，但'早'是为了什么？是为了让发现缺陷的那一秒钟尽量靠前，所以还得配上怀疑的心态，早才有意义。"第二种回答会说"要抱着怀疑的心态，因为正向思维容易走过场"。这是最贴近本章立意的答案，要直接落到板书上："对，**这一章所有的方法论，最后都要靠这个心态来落地**——心态不对，再好的流程也是走形式。"第三种回答会说"测试很重要"。这个回答没错，但太笼统，可以善意地推一把："'重要'是一个结论，**这一章我们要的不是结论，而是可执行的态度**——所以请你再往前一步，把'重要'翻译成一个具体动作：比如，从下次写作业开始，先写一份'它应该怎样'的清单，再去实现。"这样一问，学生就能把"重要"落到"测试先行"上，正好和下一章的测试分类衔接起来。

易错点 / 考点：一是**本章标题是"引论"，考纲里的落点是"理解"层级**，重点在**概念、思维与关系**（测试与 SQA、测试与开发、正反思维），而不是具体技术，答题时不要跑到第 3 章之后的测试方法上去。二是**回顾本章的六节顺序**，别把 **1.4 测试与 SQA** 和 **1.5 测试与开发** 调换——考纲里的顺序就是教材顺序。三是**收尾语的要求**：按课件备注，每课结束时要用"**以上是本节课的内容**""**这一章讲到这里**"这两句话，这是课堂规范的

固定动作，答辩或录课时不要漏。

> **过渡：**第 1 章到此全部结束。下一节课我们进入**第 2 章 软件测试的基本概念**，从"软件测试分哪几类、有几个测试级别"开始，把今天建立的这些认识落到具体的方法体系上。我们下次课见。

四、思政落点清单（共 13 处）

序号	所在页	落点主题
1	1.3	这二十来个词里，"清除"两个字最见职业底线。找缺陷是技术活，报缺陷是良心活。行业
2	1.13	迪斯尼这个案例里，损失最大的是什么？是那些圣诞节早晨失望的孩子和他们父母的信任。
1	1.14	这个案例最扎心的地方在于：问题不是测不出来，而是一开始选择了"先发货，再说"。企
2	1.16	波音这个案例讲的不是技术不够，而是把质量控制让位给了进度。可以让学生代入一个具体
3	1.17	骑士资本这类案例里，真正的风险往往不是"没人知道有毛病"，而是"有人知道了但心存
4	1.20	这个故事没什么高科技含量，但它最能让学生理解"用户视角"。对我们来说，那只是一个
5	1.21	讲到 28 名士兵、4 位病人的时候，我们要停一下。这些数字背后，是一个个再也回不来的
1	1.28	讲国外真实事故——Therac-25 放疗仪因软件缺陷导致患者受到过量辐射致死，骑
2	1.39	讲国产游戏《血狮》（1997）——宣传声势远超其实际被验证的质量，交付后口碑崩塌
43	1.43	讲什么故事：课件这句"测试成本要小于缺陷造成的损失，测试才有意义"，背后是企业实
45	1.45	讲什么故事：批判性思维这一页，考验的其实是一个人的诚信与科学态度。怎么联系知识点
49	1.49	讲什么故事：课件特意强调 SQA 要"督促测试工作的结果客观、准确和有效"，这句话说
52	1.52	讲什么故事：这一页讲协作，最能体现一个团队的质量文化。怎么联系知识点：测试与开发

第 2 章　软件测试基本概念

《软件测试课程设计》第 2 章 讲稿（第 2 周 · 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 2 章 软件测试基本概念 —— 软件缺陷、软件质量、测试分类与层次
>
> 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）》，清华大学出版社 2022
>
> 配套课件：逐页还原版 87 页（`lecture2/`，云端 http://111.231.0.226:2345/lecture2/index.html）
>
> 本讲重点：测试分类与测试级别、静态测试与动态测试、黑盒测试与白盒测试
>
> 本讲难点：不同测试级别的目标与执行主体；黑盒与白盒方法的选择依据
>
> **本讲稿为可照读完整版**：每页含【口播（可直接朗读）】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为时间紧张时可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出软件质量的内涵与主流质量模型（ISO 9126 三层模型、Boehm 模型、ISO 25000 系列）；② 能给出软件缺陷的定义与判断准则，说明缺陷产生原因与修复成本随阶段的变化；③ 能按多个维度给测试分类（静态/动态、主动/被动、手工/自动化、脚本/探索式、黑盒/白盒/灰盒）；④ 能复述测试的四个层次及各自的任务与执行主体
能力	① 能用"是否运行程序"判断静态与动态测试；② 能针对给定功能点选择黑盒或白盒方法并说明理由；③ 能画出测试工作流程（需求分析→策略→计划→设计→执行→评估）并定位任一活动
素质（思政）	① 通过"质量＝客户满意度"与缺陷成本曲线，建立"质量即责任、越早越省"的工程观；② 通过产品评审（需求/设计评审）体会严谨求实、批评与自我批评的团队文化；③ 通过验证与确认（V&V）理解工程规范与法治意识（按标准与契约办事）

本讲要建立的一个总印象：质量是可定义、可度量的；**缺陷是质量的对立面**，而测试是用分类清晰、层次分明的活动去发现与预防缺陷。

二、时间分配总表

> ⚠ **容量提示（务必先读）**：本章课件 **87 页**，是全书页数最多的一章，本讲稿为**素材全量版**（口播约 4 万字量级，≈3 小时纯朗读），**90 分钟课堂绝无可能逐页念完**。请务必按下表取用：
>
> 1. **重点页全文慢讲（2–3 分钟/页）**：
> 2.6、2.11、2.14、2.17、2.18、2.23、2.27、2.32、2.37、2.43、2.51、2.52、2.58、2.59、2.71、2.75、2.85。
>
> 2. **常规页快速带过（0.5–1 分钟/页）**：只讲口播第一段（结论）＋板书＋1 个提问。
>
> 3. **可课后阅读页（★）**：
> ★2.7、★2.8、★2.9、★2.12、★2.13、★2.15、★2.16、★2.19、★2.20、★2.21、★2.22、★2.62、★2.65、★2.78–2.81。
>
> 4. **练习/作业页**：2.84 作业与 2.86 实验要在课上交代清楚；2.85 思考题留作课后（答案口径见本稿该页）。

时段	页码	内容	分钟	备注
第 1 节	2.1–2.2	封面、第 1 章回顾	3	用 2 分钟建立本章地图
第 1 节	2.3–2.22	2.1 软件缺陷与质量内涵（质量模型・内部/外部/使用质量）	21	2.6、2.11、2.14、2.17、2.18 重点；★2.7–2.9、2.12–2.13、2.15–2.16、2.19–2.22 可略
第 1 节	2.23–2.32	缺陷定义・判断准则・产生・构成・分布・成本	16	2.23、2.27、2.32 必讲
——	——	**课间休息 10 分钟**	——	——
第 2 节	2.33–2.40	2.2 软件测试的分类（多维度）	10	2.37 静态/动态是核心
第 2 节	2.41–2.53	2.3 静态测试与动态测试（产品评审・静态分析・V&V）	14	2.43、2.51、2.52 是难点
第 2	2.54–	2.4 主动/被动测试；2.5 黑盒/白盒	8	2.56 讲清选择依据

节	2.57			
第 2 节	2.58–2.69	2.6 软件测试层次（单元→集成→系统→验收）	12	2.58、2.59 必讲；★2.62、2.65 可略
第 2 节	2.70–2.82	2.7 软件测试工作范畴（需求分析→策略→计划→设计→执行→评估）	10	2.71、2.75 重点
第 2 节	2.83–2.87	本章小结、作业、思考题、实验、结束	6	2.84、2.85、2.86 必交代

> **压缩优先级**：★2.7–2.9、2.12–2.13、2.15–2.16、2.19–2.22（质量模型细节，可布置课后阅读）→ ★2.62、2.65（示例页）→ ★2.78–2.81（脚本/探索式细节）→ 2.60、2.67、2.68（层次细节）。
> **2.23、2.27、2.32、2.37、2.43、2.52、2.56、2.58、2.59、2.71、2.75、2.85 是考点密集页，不要压缩。**

三、逐页讲稿

> 使用说明：每页第一段是**可直接朗读的口播**；其后是板书、提问与预设回答、易错点/考点、过渡语。板书建议边讲边写，不要一次写完。

【2.1】软件测试方法和技术（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 同学们好，我们继续上课。上一周我们用了整整两节课，把"为什么必须做软件测试"这个问题讲透了。今天开始，我们进入第 2 章——**软件测试的基本概念**。课件上署了我的单位和姓名——广东金融学院，周宇文；这不只是课件格式，也算一份责任声明：这一章讲的质量观、缺陷观，最后要落到每一个测试人员身上。请先记住一句话：第 1 章回答"为什么测"，第 2 章回答"测什么、怎么分类、站在哪个位置测"。前者是态度，后者是概念与方法，两章合起来就是我们这门课的地基。今天我们要把"软件缺陷""软件质量""测试分类""静态与动态""黑盒与白盒""测试层次""测试工作范畴"这一串概念一个一个钉牢。请大家把教材翻到第 2 章，我们开始。

> **板书：**中间横写"第 2 章 软件测试的基本概念"；左侧竖排写"第 1 章＝为什么测（观念）"，右侧写"第 2 章＝测什么・怎么分・谁来测（概念与方法）"。

> **提问：**请用一句话说说，上一章我们反复强调的"为什么必须测试"的那个核心理由是什么？

> **预设回答：**

- 回答"因为软件一定有缺陷"——点评：方向对，但更准确的说法是"缺陷不可避免，而且缺陷越晚被发现，修复代价越高"，这才是必须测的完整理由。

- 回答"为了交付、为了不出事故"——点评：这是目的，不是理由；测试的直接价值是**尽早发现缺陷、为质量提供信息**。

- 回答"课本上这么写的"——点评：不批评，但要求补一句自己的理解，防止把课程学成背结论。

- 追问：如果缺陷不可避免，测试能把缺陷"清零"吗？——答案是"不能，测试是在降低风险，不是消灭风险"，这个口径今天要反复用。

> **易错点 / 考点：**本页是封面页，不会直接考；但课程口径要记准——本课程是《软件测试课程设计》（课程代码 154442008），专业选修、32 学时 2 学分（理论 16＋实验 16），考查课，过程性考核＋课程设计作品与答辩；教材是朱少民《软件测试方法和技术（第 4 版）》，清华大学出版社 2022 年。后面所有名词表述一律以这本教材为准。

> **过渡：**在进入新概念之前，我们先把第 1 章的六块内容快速过一遍——下一页就是"第 1 章回顾"。

【2.2】第 1 章回顾（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

上一章我们跑了六站，现在我一站一站带你回看，你对照自己的笔记检查一遍。第一站，**什么是软件测试**。我们给过一个可以背下来的定义：软件测试是在规定的条件下对软件进行操作，以发现程序错误、衡量软件质量，并对其是否能满足设计要求进行评估的过程。说白了，它是"为了发现缺陷而运行软件"，同时顺手给质量提供证据。第二站，**软件测试的正反两面性**。正面是"验证"——证明软件做对了它该做的事，给开发者和用户信心；反面是"发现缺陷"——带着怀疑去证伪，专挑边界、异常和你想不到的组合。这两个方向看着矛盾，其实是一枚硬币的两面。第三站，**验证软件**，也就是 Verification，回答"我们有没有把产品正确地做出来"，对的是规格和设计。第四站，**发现缺陷**，这里要提醒大家，测试的目的不是证明软件没有错，而是尽可能早地把错误找出来。第五站，**V&V**——Verification 与 Validation，前者问"做得对不对"，后者问"做的是不是用户要的"，这两句话每年都考，务必分清。第六站，**软件测试和开发的关系**，以及 **TDD 测试驱动开发**：测试

不是开发的尾巴，它可以从需求阶段就并行介入，先写测试，再写实现。这六站合起来，就是第 1 章的骨架。

板书：竖排六行——① 软件测试定义 → ② 正反两面性（验证／发现缺陷）→ ③ 验证 Verification → ④ 发现缺陷 → ⑤ V&V → ⑥ 测试与开发关系／TDD。右侧留一块"待解决区"，写三个词：**缺陷？ 质量？ 分类？**

提问：Verification 和 Validation 最核心的区别是什么？请各用一句大白话说清楚。

预设回答：

- 回答"一个验证一个确认，差不多"——点评：这正是最常见的错误，两者依据不同、对象不同，不能混用；中文通常译成"验证"和"确认"，译名相近但含义不同。

- 回答"Verification 检查代码对不对，Validation 检查功能对不对"——点评：抓住了对象感，接近正确，但要补上"依据"：Verification 依据是规格、设计、接口约定；Validation 依据是用户需求和使用场景。

- 回答"两个都是测试的意思"——点评：承认它俩都在测试范畴内，但一个是"造得对不对"，一个是"造得对不对路"；用盖房子类比：Verification 是照着图纸检查墙有没有砌歪，Validation 是交房前问用户住得舒不舒服。

- 追问：如果图纸本身就错了呢？——引出"图纸错了，Verification 再严格也救不了"，这就是 Validation 必须存在的原因。

易错点 / 考点：V&V 混淆是最高频考点。判断三步法：**一看依据**（依据规格／设计＝Verification，依据用户需求／使用场景＝Validation）；**二看提问**（"Are we building the product right?" 对的是 Verification，"Are we building the right product?" 对的是 Validation）；**三看阶段**（Validation 更靠后、更靠用户场景，常与验收测试绑定）。考试形式多为选择、判断或名词解释，也可能出在案例分析里让你指出某活动属于哪一类。记忆口诀：**"验证看规格，确认看用户"**。

过渡：第 1 章给了我们"为什么测"，那么"缺陷"到底怎么定义、"质量"到底怎么衡量？这就是第 2 章要解决的问题。我们先看本章的完整地图——下一页。

【2.3】2.1 软件缺陷（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章的地图，七个小节念一遍：2.1 软件缺陷、2.2 软件测试的分类、2.3 静态测试与动态测试、2.4 主动测试与被动测试、2.5 黑盒测试与白盒测试、2.6 软件测试层次、2.7 软件测试工作范畴。其中 2.1、2.2

是地基，2.5、2.6 是考试与课程设计重点。页面上还列了整门课更远的框架：后面有**四类测试方式**——静态与动态、主动与被动、手工与自动化、基于脚本与探索式；然后是**结构化测试方法**，包括语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖、MC/DC 覆盖和基本路径覆盖；最后是**基于需求的测试方法**，包括等价类划分、边界值分析、判定表、因果图、Pairwise（工具 ATCS，据课件数据）、正交试验法和基于场景的方法。记住：先立地基——什么算缺陷、从哪来、修它多贵。

板书：板书列写七个节标题（2.1～2.7），在 2.1、2.2 下画双横线标"地基"；右侧另起一栏预先写下三个大标题：**四、测试方式｜五、结构化测试方法（ST）｜六、基于需求的测试方法（RBT）**，并注明"后续课时展开"。

提问：如果你要给"缺陷"下一个定义，你会从哪几个角度入手？

预设回答：

- 回答"程序运行不对就是缺陷"——点评：这是现象层，方向没错但太窄，忽略了文档缺陷、需求缺陷、性能缺陷这些"不报错但错"的情况。

- 回答"和需求不一致就是缺陷"——点评：抓住了核心（对照标准），值得表扬；追问一句"那需求本身写错了算不算缺陷"，引出"缺陷的对象不只是代码"。

- 回答"用户不满意就是缺陷"——点评：从用户视角看是对的，但太主观，测试要能落到可判定的依据上，所以要结合质量模型和需求。

- 点评收口：这三种回答恰好对应我们接下来要讲的三层——现象、依据、期望。

易错点 / 考点：本页是目录页，考点在于**章节结构的记忆**：2.1 软件缺陷、2.2 软件测试的分类、2.3 静态／动态、2.4 主动／被动、2.5 黑盒／白盒、2.6 测试层次、2.7 工作范畴。常见混淆是"2.4 主动与被动"记成"自动与手动"——注意主动测试与被动测试是另一对概念，手工与自动化是**另一组分类维度**，两者不在一个层次上。记忆口诀：**"缺陷、分类、动静、主被、黑白、层次、范畴"**，十四字背下来。

过渡：地图看完了，我们从第一站出发——2.1 软件缺陷。先看这一节的学习路线怎么走。

【2.4】2.1 软件缺陷（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 2.1 这一节的门牌。接下来这一整节都围绕"缺陷"两个字，但不会只给一个定义就结束。这一节按六个层次推进：先搞清楚**概念**——缺陷与质量是什么关系，工作中什么场景会用到；再做**讲解**，把胜任这件事必须掌握的知识和技术讲细；然后**举例**，用一个案例帮你真正看懂怎么用；接着**分享经验**，我会讲理论和实际有落差的地方；再给出**学习建议**；最后如果有，给大家**实用工具**和模板。展开来就是五个小标题：**2.1.1 软件质量的内涵、2.1.2 软件缺陷的定义、2.1.3 软件缺陷的产生、2.1.4 软件缺陷的构成、2.1.5 修复软件缺陷的代价**。换成五连问就是：质量是什么？缺陷是什么？从哪来？长什么样？修它有多贵？把这五问回答清楚，你就具备了做测试的第一块基本盘。

板书：板书六个关键词竖排——**概念 → 讲解 → 举例 → 经验分享 → 学习建议 → 实用工具**；右侧再竖排五个小标题：2.1.1 质量的内涵／2.1.2 缺陷的定义／2.1.3 缺陷的产生／2.1.4 缺陷的构成／2.1.5 修复代价。

提问：在"缺陷是什么"和"缺陷从哪来"之间，为什么我们要先花力气讲"质量"？

预设回答：

- 回答"因为质量更重要"——点评：方向对但不完整，要说清因果关系：缺陷是**相对质量**而存在的，没有质量标准，就没有"不合格"这个判断。

- 回答"因为考试要考"——点评：坦诚，但请把"考"换成"用"：没有质量标准的团队，缺陷单会被开发一句"这不算 bug"打回来。

- 回答"因为要先有需求"——点评：很接近了，需求其实就是质量标准的一种表达形式，值得表扬。

- 追问：那么"用户说不好用"算不算缺陷？——引出下一页要讲的核心命题。

易错点 / 考点：本页是分隔页，考试不直接考，但要注意**顺序不能颠倒**：先质量、后缺陷。答题时如果先写缺陷定义再补质量，容易被判逻辑不清。判断三步法记住：**先立标准（质量）→ 再对标准找差异（缺陷）→ 最后评估严重度与代价**。

过渡：好，我们现在就走进 2.1 的第一句话——缺陷是质量的对立面。

【2.5】缺陷是质量的对立面（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页只有一句话，但这句话是整节 2.1 的钥匙：**缺陷是质量的对立面**。我们先看课件原文怎么说的：要了解

什么是缺陷（defect），就必须先清楚"质量"（Quality）这个概念，因为缺陷是相对质量而存在的；违背了质量、违背了客户的意愿、不能满足客户的要求，就会引起缺陷或者产生缺陷。大家看屏幕上这张关系图，我按箭头念一遍：最左边是**用户**，用户有**要求／期望**；要求和期望汇聚起来，就形成了**质量**的标准；质量标准和现实产品之间是**矛盾／对立**的关系；在矛盾里**发现**了问题，于是出现了我们天天挂在嘴边的那个词——**Bug?** 注意这个问号：发现现象不等于判定缺陷。接着往下走：**软件测试**去**发现**，再通过**评估**去对照**质量模型**，最后才能下结论——这到底算不算缺陷、严重到什么程度。

为什么非得从质量讲起？因为**缺陷不是绝对的，它是相对于质量要求而存在的**。同一段代码，放在只给三个内部员工用的小工具里可能无所谓，放在银行转账系统里就是重大事故。抛开质量谈缺陷，就像抛开尺子谈长短。

生活里的类比最直白：同一件衬衫，地摊标准下"能穿"就算合格，出口欧标下纽扣拉力不够就是不合格。不是衣服变了，是**标准变了**。

软件行业的例子更清楚：一个页面在 4G 下 1.2 秒打开、在弱网下 8 秒才打开，如果需求写明"弱网环境首屏不超过 3 秒"（据课件数据口径，具体指标以你们项目的需求文档为准），8 秒就是缺陷；如果需求里根本没写性能指标，那它现在只能算一条"待确认项"，还不能直接判缺陷。所以我们做测试，第一步永远不是点按钮，而是拿到那把尺子——需求与质量模型。

> **板书：**中间画箭头链：**用户 → 要求/期望 → 质量 ⊥（矛盾/对立）→ 发现 → Bug? → 软件测试 → 评估 → 质量模型**。右上角写一句结论："**没有质量标准，就没有缺陷判定**"。左下角写两个例子关键词：**衬衫纽扣拉力（生活）／弱网首屏 3 秒（软件）**。

> **提问：**如果一个用户抱怨"这个功能很难用"，但需求文档里完全没提易用性指标，这条抱怨能不能直接登记成缺陷？

预设回答：

- 回答"能，用户说不满意就是缺陷"——点评：用户视角值得尊重，但测试记录要可判定；更规范的做法是先登记为"待确认问题"，再回到需求和隐含期望去确认，确认后升级为缺陷。这里正好对应课件那句"违背了客户的意愿"。

- 回答"不能，需求没写就不算"——点评：太绝对，会漏掉**隐含需求**。教材口径是"规定的或隐含的要求"，用户对易用性的常识性期望属于隐含要求。

- 回答"看情况/看严重程度"——点评：有判断意识，但顺序反了，严重程度是判定**之后**才评估的。

- 追问：那"隐含需求"怎么变成可判定的标准？——引出下一页要讲的"软件质量的内涵"，以及后面章节的 ISO 质量模型。

> **【思政落点 1】**讲一个真实的工作场景：某团队上线后收到大量用户投诉，工程师在周报里写"功能符合需求文档，判定通过"。知识点联系——课件上那个"Bug？"的问号，本质上就是**科学精神**的落点：发现异常不等于下结论，结论必须由证据、可复现步骤和明确标准支撑；同时它也是**诚信**的落点。价值观——测试人员的职责不是"证明自己没漏"，而是如实记录、如实上报，哪怕这个结果让团队难堪。写下"不通过"三个字需要勇气，而这份勇气就是这一行的职业底线。

> **易错点 / 考点：**本页高频考点有两个。第一，**缺陷的定义必须带上"相对于质量"这个前提**，只答"程序错误"是不完整的，标准答案要写"缺陷是相对质量而存在的，违背了质量、违背客户意愿、不能满足客户要求"。第二，**判定缺陷的四个来源**：需求（含隐含需求）、设计规格、用户期望、质量模型。判断三步法：**① 找到标准 → ② 对比实际 → ③ 判定是否违背**。案例题常见问法："发现某现象，能否判为缺陷？请说明依据"——拿分点是写出"依据是需求/质量模型"，而不是写"我觉得是"。

> **过渡：**既然判定缺陷要有一把尺子，那这把尺子——"软件质量"——本身是怎么定义的？我们把 2.1 的内容再展开一层，看下一页的五个小标题。

【2.6】2.1.1 软件质量的内涵（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 把 2.1 再往里拆一层，这五个小标题就是接下来的行军路线。第一站 **2.1.1 软件质量的内涵**：不满足于"质量就是好"，要拿到能写进报告的定义。第二站 **2.1.2 软件缺陷的定义**：把缺陷从"感觉不对"变成有明确构成要素的判断对象。第三站 **2.1.3 软件缺陷的产生**：缺陷来自需求、设计、编码、测试、配置和文档，我们会分析它产生的环节。第四站 **2.1.4 软件缺陷的构成**：一份缺陷报告要写哪些字段才合格，这直接影响课程设计打分。第五站 **2.1.5 修复软件缺陷的代价**：同一缺陷在需求阶段发现和在运维阶段发现，代价差几个数量级，这就是测试要"左移"的经济学依据。这五节的关系打个比方：**质量是尺子，缺陷是刻度上的偏差，产生是偏差的来源，构成是记录偏差的格式，代价是偏差的价格**。

> **板书：**横排五个方框：**质量的内涵（尺子）→ 缺陷的定义（偏差）→ 缺陷的产生（来源）→ 缺陷的构成（记录）→ 修复的代价（价格）**；下方写一行提示：考试重点在 2.1.2 与 2.1.5。

> **提问：**这五节里，你认为哪一节最直接影响你的课程设计得分？为什么？

> **预设回答：**

- 回答"2.1.4 缺陷的构成"——点评：非常实际，缺陷报告字段不全，开发和老师都无法复现，直接扣分；表扬其工程意识。

- 回答"2.1.5 修复代价"——点评：有战略眼光，因为它决定了测试什么时候介入、投入多少；但代价这一节更常出现在论述题里。

- 回答"2.1.3 产生的环节"——点评：也对，懂缺陷来源才能设计出有针对性的用例，属于"预防优于检测"的思路。

- 追问：那如果只能选一节背下来呢？——给建议：定义与代价两节必须能默写，其余理解即可。

> **易错点 / 考点：**本页是目录页。要注意区分 2.1.1 是"**软件**质量"的内涵，而不是泛泛的"质量"；很多同学在简答题里把两者混写，会丢掉"隐含要求""使用要求"这些关键词。判断三步法：**看到"内涵"想定义与出处（IEEE／ISO）→ 看到"产生"想环节（需求到维护）→ 看到"代价"想阶段（越晚越贵）**。

> **过渡：**我们从最根上的那一问开始——什么是质量？看下一页。

【2.7】2.1.1 软件质量的内涵（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页又出现"2.1.1 软件质量的内涵"，它不是重复，而是**转场**：前面是路线图，现在真正进入这一小节的正文。接下来连续几页，我们都在回答同一个问题——什么是质量？你会看到三种讲法：第一种是**图片和直觉**，先对着画面自己感受；第二种是**引文和观点**，用一段很有味道的话告诉你质量不是什么、又是什么；第三种才是**定义和标准**，IEEE 怎么说、ISO 怎么说。我特意把顺序安排成"感受—观点—定义"，是有原因的：质量这个概念，一上来就背定义，你会觉得它是一句空话；先有画面感，定义才会落地。所以下几页请不要急着抄定义，先跟着看图、跟着想，最后我们一起把定义抄下来。这一小节是本节的**概念地基**，后面的缺陷定义、质量模型、质量标准体系全都建立在它之上。

> **板书：**板书居中写大标题"**2.1.1 软件质量的内涵**"；下方画三格：**感受（图）→ 观点（引文）→ 定义（IEEE／ISO）**，并在"定义"格下标注"本页为概念地基"。

提问：不看课件，请用你自己的一句话给"质量"下个定义，不许说"质量就是好"。

预设回答：

- 回答"质量就是满足用户要求"——点评：抓住了"要求"，已经接近教材口径，值得表扬；再追问"用户没说的要求算不算"，引出"隐含要求"。

- 回答"质量就是不出问题、没有 bug"——点评：这是从测试视角的朴素理解，但"没有 bug"是结果不是定义，而且统计上不可能达到。

- 回答"质量就是好用、耐用"——点评：这是生活语感，属于体验层描述；提醒学生：能写进报告的表述必须可衡量，比如"平均无故障时间""缺陷密度"。

- 追问：那"贵"是不是质量的一部分？——先不回答，留到下一页那段引文里找答案。

易错点 / 考点：本页是转场页。易错点是**把"质量"与"等级""档次"混为一谈**——高档不等于高质量，高质量指的是"满足要求的程度"，而不是"用料多贵、界面多华丽"。这一点在第 2.9 页的引文里会被正面强调，考试常出判断题：**"价格高的软件质量一定更高"——错**。

过渡：那么，质量到底长什么样？我们先不看定义，看一张图。

【2.8】什么是"质量"？（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页只有一幅图和四个大字——什么是"质量"？这是纯图形页，课件故意不给文字，就是要我们**先看、先想、先说**。给大家三十秒，看三处。**一看整体感**：这幅图有没有"被认真对待过"？质量往往不是某个零件亮眼，而是整体上没一处将就。**二看细节**：边缘、接缝、比例、留白——体现质量高低的，常常是这些说不清但看得出的小地方。**三看一致性**：各部分像不像出自同一人之手？在软件里，这就是"风格统一、交互一致"。

为什么要看图？因为**质量首先是能被感知的**。用户评判软件，往往不是靠读需求文档，而是靠直觉：按钮位置一会儿左一会儿右、加载图标时有时无，用户说不清哪里不对，但会说"这个软件很糙"；反之，间距、颜色、返回逻辑都一致，他只会说"挺好用"。

看软件实例：同一功能在三个页面都有入口，按钮文案却写成"提交""确定""保存"。功能全对、测试全过，用户就是觉得别扭——这不是功能缺陷，而是一致性体验缺失。看图训练的正是测试人员最稀缺的能力：**把"感**

觉不对"变成可描述、可复现、可判定。

板书：中间写四个大字"**什么是质量？**"；下方画三个观察维度：**整体感 ／ 细节 ／ 一致性**；右下角写一行待答问题："这幅图的'认真对待'体现在哪里？"——留白，等学生回答后补写。

提问：这幅图里，哪个细节让你觉得"做的人是用心的"？请指出来，并说说你为什么这么判断。

预设回答：

- 回答"整体看起来很舒服，但我说不出哪里好"——点评：非常真实，这正是质量的特征——**说不出好，但缺了就立刻发现**；引导他把"舒服"拆成比例、间距、配色三件事。

- 回答"某个局部特别精致"——点评：观察力好，但提醒他反向思考：如果只有一个局部精致、其余潦草，整体质量是升还是降？答案是降，因为**不一致本身就是缺陷**。

- 回答"我看不出来"——点评：不要慌，看图和找缺陷一样需要训练；给方法：把图分成九宫格，逐格问"这里有没有一处显得随意"。

- 追问：如果把这幅图的某个元素换掉，你多久能发现？——引出"显著性"与"严重程度不是一回事"这个后续考点。

易错点 / 考点：本页是纯图形页，考试不会直接考。但要记住两个口径：① **质量可以被感知，但不能只靠感觉判定**，判定仍要回到要求与标准；② **一致性是质量的组成部分**，界面风格、术语、交互逻辑不统一，可以登记为缺陷（通常是易用性或一致性类缺陷）。常见的错误认知是"界面问题不算缺陷"——在真实项目和课程设计评审里，这类问题一样会被记录和扣分。

过渡：刚才我们用眼睛感受了质量，接下来听听前辈怎么用文字描述它——下一页有一段很精彩的引文。

【2.9】什么是"质量"？（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

屏幕上这段引文，我慢慢念一遍，请记关键词："质量的概念很难定义，因为它不仅是可见的，而且在体现它的作品中以某种方式被**直觉地**呈现出来。质量与大众的审美、品味或风格无关；与地位、体面或奢华无关。相反，它是在一种**接受、得体和克制**的气氛中显露出来的。"

这段话有三层意思。**第一层，"难以定义"但又"可见"**。课件专门注了英文词 intuitive，直觉。什么意思？就

是刚才看图时那种感觉——你说不出标准，但一眼就能看出一个东西是被认真做过还是糊弄的。软件也一样：同一需求，两个团队都能实现功能，用户上手三分钟就能觉出差别。这种"说不出但看得见"的特性，正是质量最难教的地方。

第二层，"与审美、品味、风格无关，与地位、体面、奢华无关"。这一句是重点，也最容易考。它明确告诉我们：**质量不是"好看""高级""贵"**。界面朴素但贴合使用习惯的软件，质量可能高于动画炫酷却操作反人类的软件；功能简单的小工具，把该做的事做到位，就是高质量的。反过来，堆料、堆特效、堆价格都不是质量的证明。对测试人员尤其重要——**判定不能被"看起来高端"带跑**。

第三层，"在一种接受、得体和克制的气氛中显露出来"。这三个词很讲究：**接受**，用户愿意接受它，用起来没有抵触；**得体**，分寸上不多不少，比如提示语不吓人、报错不甩锅、弹窗不打扰；**克制**，不炫技、不过度设计，把注意力留给用户真正要做的事。

举个大家都有的体验：点删除时，是轻轻问一句"确定删除吗"，还是弹一堆术语让你自己猜？前者得体，后者不合格。**质量的最高境界是让人感觉不到它的存在**——用得顺，就是质量好。

板书：板书三段式。上："**质量难定义，但可直觉感知（intuitive）**"；中："**≠审美 ≠品味 ≠风格 ≠地位 ≠体面 ≠奢华**"（"≠"要写得醒目）；下："**接受 · 得体 · 克制**"；右侧写一行提炼："**用得顺，就是质量好**"。

提问：这段话明确说"质量与地位、体面或奢华无关"。那么一个界面炫酷、动画华丽、价格昂贵的软件，能不能因此就判它质量高？

预设回答：

- 回答"不能，还要看它是否满足要求"——点评：完全正确，请他把这句完整说出来，作为全班结论；这才是教材口径。

- 回答"能，贵有贵的道理"——点评：这是典型的直觉误判，正好用课件这句话反驳：价格和奢华属于"体面"，而质量讲的是"满足要求的程度"；贵可能说明成本高，不能说明质量高。

- 回答"看用户喜不喜欢"——点评：方向对，但要补一句"喜欢"要落到可用性与需求满足度上，例如操作是否顺手、报错是否可理解，否则就成了纯主观。

- 追问：那"克制"和"功能少"是一回事吗？——不是。克制是不做无用打扰，功能少是能力不足，别把"偷懒"说成"极简"。

易错点 / 考点：本页是**高频概念考点**。必背三个词：**接受、得体、克制**；必背三个"无关"：**审美、品味、风格**（以及地位、体面、奢华）。常见考法：① 判断题——"界面华丽、价格昂贵的软件质量更高"（错）；② 简答题——"如何理解'质量是在接受、得体和克制的气氛中显露出来的'"。判断三步法：**看它是否满足要求 → 看它是否打扰用户 → 看它是否炫技过度**。记忆口诀：**"三无三有"**——无关审美奢华，有关接受得体克制。

过渡：感受和观点都有了，但写在测试报告里的东西不能靠直觉——下一页，我们把质量变成一句能引用的定义，而且要从客户和品牌的角度再补一刀。

【2.10】质量 =品牌 =客户满意度（5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

屏幕上这一页给出了一个等号链：**质量 = 品牌 = 客户满意度**。两个等号，我们一个一个说。为什么说**质量等于客户满意度**？因为质量最终是由用户来兑现的。你内部的代码再优雅、需求文档再规范，如果用户用着别扭、出错、不敢用，那在他心里这个软件就是"差"。我们刚才讲过，质量是"满足要求或期望的程度"，而"期望"住在用户脑子里，不在你的代码里。所以测试人员在写用例的时候，除了对着需求文档，还要反复问自己一句：**用户真正会怎么用它？**

为什么又说**质量等于品牌**？因为品牌是什么？品牌就是用户对一家公司、一个产品的**长期信任的总和**。这份信任从哪来？从每一次使用的体验里一点点攒起来。质量稳定，用户下次还敢用；质量出一次大事，十年口碑可能一夜归零。你们可以回想自己的经历：某个 App 曾经卡死丢过你的数据，之后你是不是很长一段时间不敢再打开它？这就是质量对品牌的杀伤力。

我用生活里的例子把这条链说透。一家小吃店，第一次去味道好、上菜快，你会记住它；第三次去还是好，你会推荐给朋友；第十次去依然稳定，你就成了老顾客——**这就是品牌**。中间只要有一次吃出问题还糊弄你，前面九次都白攒。软件行业更极端：软件是可以瞬间复制给几千万人的商品，**一次严重缺陷的放大倍数，远远超过一家小吃店**。

所以这一页真正想告诉大家的是：**质量不是一个技术指标，它是商业结果**。它同时决定了用户的满意度、企业的品牌和收入。这也是为什么测试岗位不是成本中心，而是品牌的守门人。同学们做课程设计时，如果只把"跑通用例"当目标，那就把这一页的意义丢了。

板书：中间写等式链：**质量 = 品牌 = 客户满意度**；等式下方画一条向下的箭头写"**长期信任的累积（慢）**"，

> 再画一条向上的箭头写"**一次严重缺陷的崩塌（快）**"；左下角写生活例"小吃店"，右下角写行业例"App 丢数据 → 卸载"。

> **提问：**为什么说软件的质量事故比传统行业的质量事故"放大倍数"更大？

> **预设回答：**

- 回答"因为用的人多"——点评：对了一半，还要补上"分发成本几乎为零、版本统一、缺陷瞬间同步给所有用户"，这才是软件独有的放大器。

- 回答"因为软件 bug 多"——点评：方向偏了，缺陷数量不是关键，关键是**影响半径**和**不可逆性**（数据丢了就没了）。

- 回答"因为修起来快，所以也无所谓"——点评：纠正一个常见误解：回归上线快不代表损失小，用户流失和信任损失修不回来。

- 追问：那测试能不能保证品牌不出事？——不能保证，但能把"出事概率"压到可接受水平，这就是风险管理的说法。

> **易错点 / 考点：**本页考点是这条等式的**因果方向**：是**质量决定**满意度与品牌，不是品牌决定质量。常见错误答案："品牌好所以质量好"（本末倒置）。另一个易错点是把"客户满意度"等同于"客户当时不投诉"——不投诉也可能是用户已经放弃、默默卸载了。判断三步法：**谁在用 → 用得好不好 → 会不会再来**。

> **过渡：**感受、观点、商业关系都讲完了，现在到了最关键的一步：把"软件质量"写成一个可以引用的正式定义。看下一页。

【2.11】软件质量 的内涵（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页进入正题，把"软件质量"变成可引用的定义。课件给了三个口径，逐条读、逐条拆。第一句，**IEEE 的定义**：质量是系统、部件或过程**满足明确需求**，以及**客户或用户需要或期望的程度**。请注意结构——它把质量拆成两边：一边是"明确需求"，写下来的、可验证的要求；另一边是"客户或用户需要或期望"，可能有、但没写下来的东西。课件在这个位置标注了"客户或用户需要或期望的程度不同"，意思是同一产品面对不同用户，期望不同，满足程度也不同，所以质量是个**程度**问题，不是"有或没有"的开关。

第二句，**ISO 口径的软件质量**：软件产品具有满足**规定的或隐含的要求**能力有关的**特征与特征的总和**。四个关键词必须抠出来：**规定的**，需求文档、合同、标准里写清楚的；**隐含的**，用户不说但理所当然的期望，比如计算器不能算错数、保存按钮不能点了没反应；**特征**，具体属性，比如响应时间、并发数、易用性；**总和**最关键——**质量是这些特征的合力，不是某一项突出**。一个软件响应快但天天崩，不叫高质量——某一项拖了总分。课件标注的标准编号是 **ISO 8492**——这里要给大家提个醒：**这是课件的一处笔误**。该定义出自 **ISO 8402《质量管理和质量保证——术语》**；若写作 **ISO 9126**，那是指质量模型标准（2.14 页要讲的三层模型）。**课堂与作业统一按 ISO 8402 讲**；遇到课件原文时可以指出这处笔误，这也是一次"以标准原文为准、不盲从讲义"的训练。

第三句最朴素：**软件质量，就是软件产品满足使用要求的程度**。它把前两句收拢到"使用"上——**标准再多，最后都要落到能不能用、好不好用、敢不敢用**。

合起来就是可以写进课程设计报告的定义：**软件质量，是软件产品满足规定的和隐含的要求（也就是用户需要与期望）的程度，它是相关特征与特征的总和，最终体现在使用要求的满足程度上**。这句话背下来，2.1.1 就拿下了。

板书：板书三行定义：① **IEEE**：质量＝满足**明确需求** ＋ 客户/用户**需要或期望的程度**；② **软件质量（课件标注 ISO 8492，实为 ISO 8402）**：满足**规定的或隐含的**要求能力有关的**特征与特征的总和**；③ **软件质量＝满足使用要求的程度**。右侧抠四个关键词：**规定 · 隐含 · 特征 · 总和**；底下一行结论："质量是**程度**问题，不是有/无开关"。

提问：一个软件功能全对、响应很快，但每周要崩溃一次。按课件的定义，它的质量算高还是低？

预设回答：

- 回答"不高，因为可靠性差，特征总和拉低了"——点评：完全正确，请他把"特征与特征的总和"这七个字说出来，这是拿分关键词。

- 回答"高，因为大部分时候能用"——点评：这是典型的"用平均值掩盖风险"的思路；提醒他质量的"总和"不是算术平均，关键特征的缺失会直接否决整体评价。

- 回答"看用户能不能忍"——点评：有商业现实感，但作为测试结论太主观；应当写成"可靠性特征不达标，判定为高风险缺陷"。

- 追问：那"隐含要求"能不能举一个你身边的例子？——引导学生举"保存后不丢数据""点了要有反应""中文不乱码"这类例子。

易错点 / 考点：本页是**定义必考页**。三个高频失分点：① 漏掉"**隐含要求**"——只写"满足需求"，丢掉一半分数；② 把质量说成"**没有缺陷**"——定义里从来没有这句话；③ 把"特征总和"理解成"各项加权平均"，忽略了关键特征的一票否决效应。考试形式多为名词解释、简答或案例分析；记忆口诀：**"规定＋隐含，特征成总和，落回使用中"**。判断三步法：**有没有写下来（规定）→ 用户是不是理应能得到（隐含）→ 各项特征合起来够不够（总和）**。

过渡：定义有了，可"特征"到底有哪些？这就是下一页要给的——高质量软件的标准体系，我们从产品、流程、商业环境三个面去看。

【2.12】高质量软件标准体系（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页最有体系感：谈质量不能只盯着代码，要分三层看。课件把高质量软件的标准体系画成三个维度——**产品、流程、商业环境**，逐个展开。

先看**产品质量**。课件说得很明确：产品质量是人们实践产物的属性和行为，**可以认识、可以科学地描述**，并能通过方法和人类活动**改进**。这话在打气——质量不是玄学，可测量、可改进。那怎么描述？靠**质量模型**。课件列了三个经典模型：**McCall 模型、Boehm 模型、ISO 9126 模型**。它们的共同思路是把"质量"拆成若干可度量的特性，比如功能性、可靠性、易用性。写课程设计报告时，如果能在"测试范围"一节引用一个质量模型，说明测了哪几个特性、为什么选，报告立刻上档次。

再看**过程质量**。核心是：**产品的质量，是过程质量的产物**；过程不靠谱，产品靠运气。课件列了三样：**CMM 软件能力成熟度模型**，评估组织的开发过程成熟到什么等级；**ISO 9000** 国际标准过程模型，通用质量管理体系标准；以及 **SPICE**，软件过程改进和能力决断，用于评估和改进软件过程能力。共同点是**把"人治"变成"制度化"**——不指望高手救场，而是让流程稳定地产出合格产品。

最后看**商业环境**。课件列举了商业过程中的质量内容：**培训、成品制作、宣传、发布日期、客户、风险、成本、业务**等。这一层最容易被技术同学忽略：一个技术上没问题的产品，如果发布日期被强行提前导致测试被压缩，或者宣传夸大到产品根本做不到，质量照样出问题。所以"高质量"是**产品、流程、商业环境三者的合围**，缺一个角都不成立——记住：**测产品、看流程、认环境**。

板书：板书三列框架——**产品（质量模型：McCall／Boehm／ISO 9126）｜流程（CMM／ISO 9000／**

> **SPICE）｜商业环境（培训・成品制作・宣传・发布日期・客户・风险・成本・业务）**；产品列下方补一句："**可认识、可描述、可改进**"；流程列下方补一句："**产品的质量是过程质量的产物**"。

> **提问：**既然产品质量可以通过测试来保证，为什么还要关心"过程质量"？

> **预设回答：**

- 回答"因为测试只能发现缺陷，不能预防缺陷"——点评：非常到位，这正是过程质量存在的理由；请他把"测试是检测、过程是预防"这一对比写在笔记上。

- 回答"因为要过 CMM 认证、要拿单"——点评：这是商业动因，确实存在，但只是手段；根本目的仍是让质量稳定可复制，而不是靠个别高手。

- 回答"过程质量是管理层的事，跟我们测试没关系"——点评：明确纠正。测试用例评审、缺陷跟踪规范、回归策略，本身就是过程质量的一部分，测试人员恰恰是过程质量的重要执行者。

- 追问：SPICE 和 CMM 有什么区别？——提示二者都面向过程能力评估与改进，记忆时抓住"能力成熟度"与"过程改进和能力决断"这两个名称即可。

> **易错点 / 考点：**本页考点集中在**名称与归属的对应**，非常容易张冠李戴。请记牢：**质量模型**是 McCall、Boehm、ISO 9126（面向**产品**）；**过程质量**是 CMM、ISO 9000、SPICE（面向**流程**）；商业环境的内容是培训、成品制作、宣传、发布日期、客户、风险、成本、业务（面向**环境**）。常考题型是连线题或选择题，比如"下列属于过程质量范畴的是"。记忆口诀：**"产品三模型，流程三体系，环境八要素"**（八要素按课件列举计数，据课件数据）。另一个易错点是把 CMM 说成"软件测试模型"——它评的是**组织开发过程的能力成熟度**，不是测试技术。

> **过渡：**体系讲完了，但测试人员最常打交道的还是产品那一列。下一页我们把"产品质量的标准"具体摊开，看看有哪几项是必须盯住的。

【2.13】产品质量的标准（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把"产品质量"落到条目上，课件列了九项，我念一遍：**功能性 Functionality**、**可用性 Usability**、**可靠性 Reliability**、**性能 Performance**、**容量 Capacity**、**可伸缩性 Scalability**、**可维护性 Service manageability**、**兼容性 Compatibility**、**可扩展性 Extensibility**。课件把这九项列在"**非功能特性**"栏目下，

须提醒：按通行 ISO 口径，**功能性属于功能特性**，其余八项才是非功能特性；答题按教材口径写。

再挑四组最容易混的讲清。**可用性和功能性**：功能性回答"有没有、对不对"，可用性回答"能不能顺利用起来"——能算对却要绕八步，功能性没问题，可用性很差。**可靠性和性能**：性能看"快不快"，可靠性看"稳不稳、出错后能否恢复"；快而不稳，比慢而稳更可怕。**容量和可伸缩性**：容量是"现在能装多少"，比如并发数、数据量；可伸缩性是"需要时能不能扩上去"，加机器后能力能不能跟着涨。**可维护性和可扩展性**：前者偏"好不好查、好不好修"，后者偏"好不好加新功能"。

为什么九项要一起看？它们常常**互相拉扯**：加缓存性能上去了，可能一致性出问题；校验做严，可能可用性下降；为可扩展把结构做复杂，可能可维护性变差。测试人员的角色是**在特征之间帮团队做取舍的裁判**——说清"为保住 A 牺牲了多少 B"。

做课程设计时，别九项平均用力，而是**选三到四项最相关的特性**，写明理由，再设计对应验证方法——这比"我全都测了"更有说服力。

板书：板书九项，横排三行三列：**功能性／可用性／可靠性｜性能／容量／可伸缩性｜可维护性／兼容性／可扩展性**；右侧写四组对比：**功能 vs 可用（有没有 ↔ 顺不顺）· 性能 vs 可靠（快不快 ↔ 稳不稳）· 容量 vs 伸缩（装多少 ↔ 能不能扩）· 可维护 vs 可扩展（好不好修 ↔ 好不好加）**；角落标注"课件栏目：非功能特性（功能性按 ISO 口径属功能特性）"。

提问：一个系统"响应很快、功能很全"，但每隔几天就需要重启一次。请指出它在九项标准中哪一项最不达标，并说明可能连带影响哪些项。

预设回答：

- 回答"可靠性不达标"——点评：正确，请补充理由："稳定运行能力不足"，并追问他会不会连带影响可用性；引导他答出"频繁重启会直接伤害可用性与用户体验"。

- 回答"性能不达标"——点评：这是最常见的误判，因为"响应快"给了错觉；提醒他性能看的是速度与吞吐，重启看的是稳定性，别把两者混为一谈。

- 回答"可维护性不达标"——点评：有洞察力，频繁重启往往意味着缺陷定位困难；但这是**连带影响**，主因仍是可靠性。

- 追问：如果这个系统是银行核心系统，你会在测试报告里怎么定级？——引导学生用"影响范围＋严重程度＋发生频率"三要素定级。

【思政落点 2】讲一个课堂可讨论的情境：同一次测试里，你发现了三个缺陷，其中两个是界面小瑕疵，一个是概率极低但后果严重的隐患——比如涉及数据准确性。知识点联系——这一页九项标准里，可靠性、容量这类特性往往"平时看不出来，出事就是大事"，测试人员要有**科学的定级判断**，不能只挑好复现、好交差的缺陷报。价值观落点：**如实记录、按标准定级、不为好看的数字隐去不利结果**，这就是第 2 次课思政设计里最看重的诚信与质量责任；一个团队的质量文化，就体现在敢不敢把那个"概率低但后果重"的缺陷摆在桌面上。

易错点 / 考点：本页是**高频考点页**。必背九项中英文对照（功能性、可用性、可靠性、性能、容量、可伸缩性、可维护性、兼容性、可扩展性），以及四组对比（功能 vs 可用、性能 vs 可靠、容量 vs 伸缩、可维护 vs 可扩展）。常见错误：① 把"性能"当"可靠性"，把"容量"当"可伸缩性"；② 把"功能性"归入非功能特性（注意课件页面本身把它列在"非功能特性"栏目下，答题按教材口径区分）；③ 认为"功能全就等于质量高"。考试形式多为选择题、判断题或案例分析题；记忆口诀：**"功能可用可靠，性能容量伸缩，维护兼容扩展"**——九个词，三三成句。判断三步法：**先问是功能还是非功能 → 再问是"现在的表现"还是"未来的能力" → 最后问它和哪一项容易被混淆**。

过渡：质量的内涵与标准我们讲完了，下一片开始，我们将进入 2.1.2——软件缺陷的定义，把"缺陷"这个东西从概念变成一份可以填写、可以评审、可以追责的记录。这是你们课程设计里马上要用到的本事。

【2.14】软件质量特征（ISO 9126）（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

前面我们讨论的是"质量是什么"，这一页把它落到国际标准上：软件的质量到底由哪几个可检查的侧面组成。请大家记住这个标准号——ISO 9126，它是影响最大、被引用最多的软件质量模型之一，后面很多标准都是从它发展出来的。它把软件质量归纳成六大特征，我逐个讲。

第一，功能性。课件上的定义是"与一组功能及其指定性质有关的一组属性"，功能就是满足明确或隐含需求的能力。说人话：该做的事，做没做到、做得对不对。第二，可靠性，"在规定的一段时间和条件下，软件维持其性能水平的能力"——请注意"一段时间"和"规定条件"这两个限定，程序跑一次不崩不叫可靠，长期稳定运行才算。

第三，易用性，"规定或潜在的用户为使用软件所需付出的努力和所作的评价"。它有两半：一半是客观的努力，学多久、点几下；一半是主观的评价，用起来舒不舒服。第四，效率，"在规定条件下软件的性能水平与所使用资源量之间关系"，说白了就是同样的活儿，干得快不快、吃得少不少。

第五，可维护性，"与进行指定的修改所需的努力有关的属性"，改一个需求要动多少代码，这就是可维护性。

第六，可移植性，"软件从一个环境转移到另一个环境的能力"，换操作系统、换数据库还能不能跑起来。

最后一句课件上的话很重要：每一个质量特征都分别与若干子特征相对应。也就是说，这六项还太粗，不能直接拿来测量，必须继续往下拆。拆下去长什么样？就是下一页那张三层模型。

这里我再提醒一句：这六大特征不是并列的口号，而是有优先级的。面对不同产品，团队必须根据用户场景给它们排序，否则所谓"质量"就成了没有重点的一团。

> **板书：**写"ISO 9126 六大质量特征"，一边讲一边竖着写：**功能（做没做到）→ 可靠（稳不稳）→ 易用（好不好用）→ 效率（快不快・省不省）→ 可维护（好不好改）→ 可移植（好不好搬）**；下面补一句："**每一个特征往下还有若干子特征**"。

提问：如果让你给"学生选课系统"评质量，这六个特征里你会先盯住哪两个？为什么是这两个？

预设回答：

- 答"可靠性和效率"：✅ 很到位。点评并追问："选课系统为什么 reliability 重要？"引导到业务本质——**抢课高峰期系统崩了，学生和教务全部受影响，这是关键业务时间窗（如开学第一周）最不能出的问题**，所以可靠性优先级最高；紧接着就是效率，因为并发量决定选课能不能在几分钟内结束。

- 答"易用性，因为界面要好看"：⚠ 只说到了外观。点评：易用性在标准里指的是"用户为使用软件所需付出的努力和评价"，**重点是任务完成成本（几步能选上课、会不会选错）**，不只是好看；界面美观只是其中一小部分。

- 答"六个都很重要，全部同等对待"：❌ 典型但错误的答案。点评：资源永远有限，**质量特征必须有优先级，依据是产品定位和用户场景**；教务内部系统的可移植性权重就很低，而面向公众的 App 则完全不同。

易错点 / 考点：①中英文对应必须记牢——功能性 functionality、可靠性 reliability、易用性 usability、效率 efficiency、可维护性 maintainability、可移植性 portability；②最常见的混淆是把"效率"和"性能"划等号，标准里效率＝性能水平与资源消耗的关系，**是"性价比"不是单纯的快**；③把"易用性"等同于"界面美观"；④考点形式：给一段场景描述让你判断属于六大特征中的哪一个（选择题），或者让你补全"特性—子特性"的对应关系。**判断三步法**：先问"这个属性管什么"→ 再对照六大特征的定语（功能实现/稳定运行/用户努力/资源效率/修改成本/环境迁移）→ 最后排除"性能就是效率"这个陷阱。

过渡：六大特征还要往下拆，ISO 9126 把它拆成了三层——我们看下一页这张结构图。

【2.15】ISO 9126 软件质量三层模型（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是一张图，请大家把目光放到图上，它要你看出的就是一件事：软件质量不是一句话，而是一棵三层树。第一层在最上面，是总目标——软件质量本身；第二层是刚才讲的六大质量特性：功能性、可靠性、易用性、效率、可维护性、可移植性；第三层在每根枝条的末端，是质量子特性，以及再往下的度量指标。

为什么非要分三层？因为"质量好"这句话没法验证，"可靠性好"同样没法验证，必须一路拆到能测量的东西上。比如可靠性往下拆，可以拆出成熟性、容错性、易恢复性；成熟性还可以用平均无故障时间来度量，容错性可以用"注入故障后系统能否继续工作"来度量。再比如易用性往下拆，是易理解性、易学性、易操作性，可以用"新用户完成第一个任务要多久"来度量。这样一拆，质量就从一句形容词变成了可以打分、可以验收的东西。

这张图还有一个用途：它是后面所有质量模型的"母版"。无论是 Boehm 模型，还是 ISO 25000 系列，走的路子都是"总目标—特性—子特性—度量"这条主线，区别只在特性的划法和层数。

再补一句它和测试的直接关系：三层模型不是学术分类，它决定了测试设计的方法——你选定哪一个度量指标，就等于宣布了要设计哪一类测试项。

板书：画三层：**第一层 软件质量（总目标）／第二层 六大特性／第三层 子特性＋度量指标**；旁边写口诀"**一层看目标、二层看特性、三层看度量**"，再举一例：**可靠性 → 容错性 → 故障注入后能否继续服务**。

提问：为什么"软件质量好"这句话在验收会上没人能反驳，也没人能签字？这张图怎么解决这个问题？

预设回答：

- 答"因为太笼统，没有标准"：✅ 抓住要害。点评：所以三层模型的第二层把"好"拆成六个可讨论的维度，第三层再拆成可测量的指标，**质量评价从口水战变成数据对话**。

- 答"那就直接测第三层的指标，不用管前两层"：⚠ 只见树木。点评：度量指标是手段，第二层的特性是目的；**只盯指标会出现"指标好看、产品难用"的情况**，例如启动时间达标但核心任务路径极长。

- 答"这张图是给开发看的，跟测试没关系"：❌ 需要纠正。点评：恰恰相反，**测试用例的设计就是从第三层的度量指标倒推出来的**——你打算怎么度量，就决定了要设计哪些用例。

易错点 / 考点：①三层的顺序不能颠倒（总目标→特性→子特性/度量）；②"特性"与"子特性"的层级容易混，记住**子特性是能落到测量上的那一层**；③考点形式：给出一个子特性让你归类到六大特性之下（如"易恢复

性"归可靠性、"易学性"归易用性、"资源特性"归效率）；④记忆口诀：**一句形容词拆不成用例，拆到第三层才叫质量**。

> **过渡：**ISO 9126 的三层结构讲完了，但质量模型不止这一个。在它之前还有一位前辈，提出了软件质量模型史上第一个层次化结构——我们看 Boehm 模型。

【2.16】Boehm 软件质量模型（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 Boehm 软件质量模型。Boehm 是软件工程领域的大家，他在 20 世纪 70 年代末提出这个模型，是软件质量模型里较早、也很有影响力的一套层次化结构。课件这一页是一张关键词地图，信息密度很大，我带大家按三条线索来读。

第一条线索是"产品操作"，也就是软件在运行时表现出来的质量。上面有正确性、可靠性、效率、完整性、可用性，还有容错性、执行效率、储存效率、存取控制、存取检查这些更细的条目。可以看出，这条线索回答的是"软件跑起来对不对、稳不稳、快不快、数据安不安全"。

第二条线索是"产品修改"，也就是将来要改它的时候，成本高不高。对应的关键词有可维护性、可测试性、灵活性、扩展性、模块性、一般性、简单性、连贯性。请注意 Boehm 模型的一个历史贡献：它把可测试性和可理解性这类"开发友好"的属性也纳入了质量，而不只盯着运行结果。

第三条线索是"产品维护"和迁移，也就是软件换环境、被别的系统接着用的时候行不行。关键词有可移植性、互用性、机器独立性、软件系统独立性、通讯公开性、数据公开性、重复性、阐述性。

另外还有一组偏人的属性：可训练、沟通良好、易操作的、自我操作性、工具——这就是人和软件打交道时的质量。大家抓一个总印象就够了：Boehm 模型第一次把质量从"代码对不对"扩展到了"好不好改、好不好测、好不好搬、好不好学"，这就是它比单纯的正确性检查高明的地方。

还有一个细节值得注意：这份模型里出现了"工具""可训练""沟通良好"这样的条目，说明早在几十年前，软件质量就已经被理解为"技术加人"的合力，而不只是纯粹的技术问题。

> **板书：**黑板上横着画三条线：**产品操作（正确性・可靠性・效率・完整性・可用性）／产品修改（可维护性・可测试性・灵活性・扩展性・模块性）／产品维护与迁移（可移植性・互用性・机器独立性・软件系统独立性・通讯公开性）**；右侧补一列"**人机属性：可训练・沟通良好・易操作**"。

> **提问：**Boehm 模型把"可测试性"写成一项质量属性。请问：一个软件"可测试性差"是什么意思？它会给项目带来什么实际后果？

预设回答：

- 答"就是很难测"：✅ 方向对，但要说透。点评：可测试性差的典型表现是——**模块之间耦合死、没有接口、没人能造出输入去观察输出**；后果是测试成本飙升、缺陷漏到生产环境，你在课程设计里也会遇到：写得越死，用例越难设计。

- 答"可测试性是测试人员的事，跟开发质量无关"：❌ 常见误解。点评：可测试性恰恰是**设计质量的体现**，它由架构和接口设计决定，测试人员只能被动承受。所以业界有句话：**测不动的东西，往往就是设计得不好的东西**。

- 答"把它当作一项功能需求写进规格书就行"：⚠ 想法不错但落不了地。点评：可测试性不是一条功能，而是一组设计要求（接口可注入、状态可观察、依赖可替换），需要落实在**设计评审的检查项**里。

> **易错点 / 考点：**①别把 Boehm 模型和 ISO 9126 记混——**Boehm 是层次化质量模型的早期代表，ISO 9126 是国际标准的六特性结构**；② Boehm 模型的特色在于同时覆盖"运行表现"和"修改/迁移成本"，并纳入可测试性、可理解性；③考点形式：判断"可测试性属于哪一类质量属性"（产品修改类）、或比较两个模型的异同（简答题）；④记忆口诀：**跑得好（操作）、改得动（修改）、搬得走（维护迁移）、学得会（人机）**。

> **过渡：**Boehm 模型是"前辈"，ISO 9126 是"经典"，那么今天企业真正在用的是哪一套标准？我们看最新的一代——ISO 25000 系列。

【2.17】最新质量标准：ISO 25000 系列（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页叫"最新质量标准：ISO 25000 系列"，它的正式名字是 ISO/IEC 25000《软件产品质量要求和评定》，英文缩写 SQuaRE，全称 Software product Quality Requirements and Evaluation。请大家记住这几个字母，它是现在国际上评价软件质量的主干标准。
>
> 为什么要出这个新标准？因为老的 ISO 9126 只管"质量模型"，老的标准 ISO 14598 只管"质量评价"，两套标准各说各话；SQuaRE 把它们整合成一条链，从"定要求"一直走到"做评价"。SQuaRE 分成几个分部：2500n 是质量管理，2501n 是质量模型，2502n 是质量测量，2503n 是质量要求，2504n 是质量评价，后面还有

25050 到 25099 的扩展部分。大家不用背全编号，但要抓住它的逻辑顺序：**先定模型（测什么维度）→ 再定测量（怎么量）→ 再定要求（量到什么程度算合格）→ 最后做评价（按证据下结论）**。

在我们国家，这套标准被等同采用为 GB/T 25000 系列，下一页要讲的产品质量模型，对应的就是 GB/T 25000.10-2016。课件这一页给出的链接是 ISO 官方的在线浏览平台，感兴趣的同学可以课后去查标准原文，考试不会考那么细，但将来做测试方案时，引用标准号是很加分的。

为什么加分？因为笼统地说"我们按照行业标准做测试"，对方没法追问；而说出"我们的质量要求和评价依据 GB/T 25000 系列"，对方马上可以追问到具体分部，这就是专业与业余的差别。

一句话总结这一页：质量模型从"一家之言"变成了"国际通用语言"，好处是——当你说"这个系统可靠性不达标"时，所有人对"可靠性"的理解是一致的。

板书：写 **ISO/IEC 25000 SQuaRE**，下面竖排：**2500n 质量管理 → 2501n 质量模型 → 2502n 质量测量 → 2503n 质量要求 → 2504n 质量评价**；右侧写对应关系："**国内等同采用：GB/T 25000 系列（产品质量模型＝GB/T 25000.10-2016）**"；再写一句逻辑链："**定模型 → 定测量 → 定要求 → 做评价**"。

提问：如果换作你给一个课程设计项目写测试方案，为什么"先引用一个质量标准"比"直接开始设计用例"更稳妥？

预设回答：

- 答"因为有依据，别人不好反驳"：✅ 说到了要点。点评：标准提供的是**共同语义和可追溯的评价口径**——你说"性能不达标"，标准会追问"你测的是时间特性还是资源特性、在什么条件下、阈值多少"。

- 答"引用标准就是走形式，最后还是要自己测"：⚠ 一半对。点评：标准不替你测，但它替你**定义验收的语言**；没有这层语言，需求方说"要流畅"，开发说"够快了"，测试夹在中间无法裁决。

- 答"越新的标准越好，老标准可以完全丢掉"：❌ 需要纠正。点评：**ISO 9126 的六特性思维今天仍然在用**，SQuaRE 是在它基础上升级整合；学习路径应该是"先懂 9126 的骨架，再看 25000 的整合"，而不是简单地新替旧。

易错点 / 考点：①标准号与名称对应：**ISO/IEC 25000＝SQuaRE＝软件产品质量要求和评定**；**GB/T 25000.10-2016 是我国等同采用的产品质量模型**；②别把 25000 与 9126 的关系说反——**9126 是前身，25000 是整合与升级**；③考点形式：给出标准编号让你说分部职责（如 2501n 管模型、2504n 管评价），或判断"ISO 25000 系列只包含质量模型"（错，它是一条从要求到评价的完整链）；④记忆口诀：**先模型、再测量、后要求、终评价**。

> **过渡：**这页讲的是标准的"编号体系"，接下来的问题更本质：一个软件的质量，其实分层存在——内部质量、外部质量、使用质量。它们是层层影响的关系，我们看下一页那张链条图。

【2.18】内部质量→外部质量→使用质量（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 这一页是一张关系图，讲的是质量的三层存在形态。请大家看图上从左到右的走向：内部质量、外部质量、使用质量，三者的箭头是"影响"——内部质量影响外部质量，外部质量影响使用质量；反过来看，箭头是"依赖于"——使用质量依赖于外部质量，外部质量依赖于内部质量。同时右边还有一个"使用语境"作用于使用质量。
>
> 怎么理解这三层？内部质量是"看不见的质量"，指的是代码、文档这些中间产品本身的属性，靠内部度量来看；外部质量是"看得见的质量"，指的是软件运行起来之后表现出来的行为，靠外部度量来看；使用质量是"用得着的质量"，指的是用户在他的真实使用语境下能不能顺利完成任务，靠在使用中度量。
>
> 举个我们身边的例子：一个登录模块，代码里密码是明文比较、没有失败次数限制，这是内部质量问题；用户实际登录时能被暴力破解、后台报错，这是外部质量问题的表现；而在真实业务里，用户因此被盗号、被投诉，这就是使用质量被破坏的结果。
>
> 所以这三层不是三种无关的质量，而是同一件事的三个观察距离：**离代码越近越早能发现，离用户越近越能说明危害**。测试的价值就在于把远处的危害，用近处的、早期可测的指标提前识别出来。
>
> 还要说清一点：这三层不是三张彼此独立的检查表，而是一条证据链——内部指标异常可以预测外部行为的风险，外部行为异常可以预测用户使用效果的损失。测试报告里如果能把这条链串起来，说服力会强得多。

> **板书：**画一条横向链：**内部质量 —影响→ 外部质量 —影响→ 使用质量**；在下方画反向箭头，标注"**依赖于**"；右侧写"**使用语境 → 影响 → 使用质量**"；再补三层度量名：**内部度量｜外部度量｜在使用中度量**，旁注一句"**越靠内部越早可测，越靠使用越能说明危害**"。

> **提问：**如果测试资源只够盯一层，你应该盯哪一层？请说出理由。

> **预设回答：**

- 答"盯使用质量，因为用户只在乎这个"：⚠ 重要但不可行。点评：使用质量最贴近价值，但它**只能在真实语

境里长时间观察，早期的课程设计项目根本没有真实用户；所以实践中要靠外部度量做替身。

- 答"盯内部质量，代码写好了自然没问题"：❌典型的"技术自负"。点评：内部干净不等于用户能用，**代码质量只是必要条件**；而且内部指标容易自己糊弄自己（注释率高不代表注释有用）。

- 答"三层都要有指标，但可以用内部和外部指标提前预测使用质量"：✅这是最专业的回答。点评：这正是模型画的"影响/依赖于"关系要表达的——**三层是层次递进、可传递的证据链**，早期用内部、中期用外部、上线后用实际使用度量。

> **易错点 / 考点：**①箭头方向常考：**内部影响外部、外部影响使用；使用依赖外部、外部依赖内部**，别把"影响"和"依赖"方向写反；②"使用语境"这个要素只有使用质量这一层才有；③考点形式：判断题"内部质量好则使用质量一定好"（错，是影响不是等同）；④记忆口诀：**内影响外、外影响用；用依赖外、外依赖内；语境只在最外层**。

> **过渡：**三层里最靠内、也最早能测的那一层就是内部度量，它到底量什么、有什么好处？我们看下一页。

【2.19】内部度量（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲内部度量。课件上给了两张图，主题是内部度量的对象和它的价值，备注里有一句英文提示——Economic risk mitigation，也就是"缓解经济风险"，另外还点出 ISO/IEC 9126-1:2001 把软件质量模型分成内部质量与外部质量模型、使用质量模型。

内部度量量的是什么呢？量的是**中间产品自身的可测量属性**：代码和设计文档的规模、结构复杂度、模块之间的耦合程度、注释情况、需求到代码的可追溯性等等。这些东西有一个共同特点——**不需要把系统完整跑起来就能测**，静态评审、静态分析工具就能给出数据。

它的价值就在那句"缓解经济风险"上。软件工程里有个普遍规律：缺陷发现得越晚，修复代价越高，因为越晚就意味着越多的设计、代码和测试已经建立在错误的基础之上。内部度量让团队在编码阶段、甚至在评审阶段就拿到质量数据，从而把修复成本压在最便宜的那个时间点上。

但请注意内部度量的局限：它测的是"中间产品的属性"，不是"用户能感知的效果"。圈复杂度低不代表用户能顺利下单，注释率高也不代表注释说得对。所以内部度量只能告诉你"哪里可能有问题"，不能替你做"能不能交付"的最终判断。

再给一个实操建议：内部度量最好成组使用。只看复杂度容易被误导，但把"复杂度偏高＋被频繁修改＋评审中问题集中"三个信号叠加起来，高风险模块基本就锁定了，测试和评审资源就该优先压到这些模块上，这就是最朴素的数据驱动测试。

板书：写"**内部度量＝量中间产品本身**"，下面列：**规模 · 复杂度 · 耦合度 · 可追溯性 · 注释/规范符合度**；再写两个关键词："**不需要运行程序**"＋"**Economic risk mitigation（缓解经济风险）**"；旁边画一条代价曲线并标注"**越晚发现，修复越贵**"。

提问：既然内部度量不能证明用户满意，为什么还要花人力去做静态分析和代码评审？

预设回答：

- 答"为了提前发现问题，省钱"：✅ 对了，就是"缓解经济风险"。点评：追问一句"省在哪里"——**省在还没有把错误放大器做大的时候修复**，一个需求理解偏差在需求阶段改是写几行字，在上线后改可能是全量回归。

- 答"因为标准要求做，属于流程合规"：⚠ 只答了形式。点评：合规只是手段，真正目的是**让不可见的东西变得可见**；代码质量数据是团队判断"能不能进入下一阶段"的门禁依据。

- 答"内部度量好，那就以圈复杂度为唯一指标考核开发"：❌ 非常危险的做法。点评：单一指标一定被"优化"（程序员把函数拆碎，复杂度好看但可读性更差），**内部度量要成组看、结合评审判断**，不能拿来当 KPI 硬压。

易错点 / 考点：①内部度量**不需要运行程序**，这是它与外部度量最本质的区别；②不要把内部度量与静态测试混为一谈：**内部度量是"量数据"，静态测试是"找缺陷"**，两者常同时进行但目的不同；③考点形式：判断某项活动属于内部度量还是外部度量（如"统计圈复杂度"属内部，"测量登录响应时间"属外部）；④口诀：**内部量自己（代码文档）、外部量行为、使用量效果**。

过渡：内部、外部、使用这三级到 ISO 25010 里被重新组织成了两个模型：一个管产品本身，一个管用户使用。先看产品质量模型。

【2.20】产品质量模型（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是产品质量模型。课件给了两个关键信息：一个是 ISO 官方的标准在线浏览地址（iso.org 的 OBP 平台，标准号是 ISO/IEC 25010 第 1 版），另一个是 **GB/T 25000.10-2016 质量模型**——也就是说，我们国家

等同采用的标准就是它，考试和写测试方案时引用这个国标号最稳。

产品质量模型要回答的问题是：**软件这个"产品"本身，应该从哪几个维度评价？** 在 GB/T 25000.10 里，产品质量特性有八个：功能适合性、性能效率、兼容性、易用性、可靠性、安全性、可维护性、可移植性。

我们来对比一下它和 ISO 9126 的六大特征：可靠性、易用性、可维护性、可移植性这四项保留了下来；功能性细化为"功能适合性"，强调的是"功能是否满足规定和隐含需求"，而不只是"有没有这个功能"；效率改称"性能效率"，把时间行为和资源利用都纳进来；新增了两项——**兼容性**（与其他产品共享环境、交换信息的能力，比如能不能和现有系统对接、能不能和别的软件共存）和**安全性**（信息保密性、完整性、抗抵赖、可核查、真实性）。

为什么要新增这两项？看看今天的软件形态就明白了：系统都是联网的、都要集成、都存着用户数据。**在 9126 的年代，安全性和兼容性是"附加项"；在今天，它们是"入场券"**——一个不能对接的单机系统，在业务上几乎没有价值；一个泄露用户数据的系统，功能再全也是负分。

还有一层：产品质量模型里的这些特性都是"产品属性"，必须靠需求、设计、编码、测试共同实现，不能指望某个阶段事后补出来。所以需求评审时就该问：这个需求的兼容性要求是什么、安全要求是什么，而不是等系统做完再补一轮安全测试。

> **板书：**写"**产品质量模型（GB/T 25000.10-2016 / ISO/IEC 25010）**"，下面竖排八个：**功能适合性 · 性能效率 · 兼容性 · 易用性 · 可靠性 · 安全性 · 可维护性 · 可移植性**；右侧写"**相比 ISO 9126：保留 4 项，功能性→功能适合性，效率→性能效率，新增 兼容性＋安全性**"。

提问：为什么标准要把"安全性"从功能性下面的一个子特性，提升为一个独立的质量特性？

预设回答：

- 答"因为安全问题越来越严重"：✅ 方向对。点评：更准确地说，是**安全已经不只是"某个功能的属性"，而是贯穿全部功能的横切要求**——登录、支付、上传、日志都涉及，放在功能性子项里会被漏掉。

- 答"提升层级只是标准改版的形式调整，没有实质意义"：⚠ 低估了标准演进。点评：标准属性的"层级位置"决定了**评审和测试的关注度**；一个条目从子特性升到特性，意味着它必须有独立的测试类型、独立的验收门槛。

- 答"安全是运维和安全团队的事，测试不用管"：❌ 必须纠正。点评：安全性作为产品质量特性，**在测试范围内**，常见的有认证授权、输入校验、敏感信息保护等测试项；出了问题，测试同样是责任方之一。

易错点 / 考点：①标准号对应要准：**产品质量模型＝ISO/IEC 25010（GB/T 25000.10-2016）**；②八个特性最容易漏掉"兼容性"和"安全性"；③不要把"性能效率"写成"性能"或"效率"就草草带过，标准含义是**时间行为＋资源利用**；④考点形式：给出场景判断属于哪个特性（"换数据库后系统仍能运行"→可移植性；"与第三方支付接口正常交互"→兼容性；"越权访问被拦截"→安全性）；⑤记忆口诀：**功能性能、兼容易用、可靠安全、可维护可移植**。

过渡：产品质量模型管的是"产品本身好不好"，可用户其实不关心产品，只关心"我用得顺不顺"。这就轮到另一个模型——使用质量模型。

【2.21】使用质量模型（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是使用质量模型。它和上一页的产品质量模型是一对：产品质量模型站在**产品**角度问"这个软件本身有哪些质量维度"，使用质量模型站在**用户**角度问"用户在实际使用语境里，能不能达成他的目标"。课件这一页也是以图为主，备注同样提示了 Economic risk mitigation 和 ISO/IEC 9126-1:2001 的两分法。

使用质量的特性，按现行的 GB/T 25000.10／ISO 25010，一般列五个：有效性、效率、满意度、免于风险、语境覆盖。有效性是说用户能不能准确、完整地达成目标；效率是说达成目标要花多少时间、多少成本；满意度是用户的主观感受，包括有用性、可信度、愉悦感、舒适度；免于风险是说软件有没有给用户的经济、健康、数据安全带来风险；语境覆盖是说它能不能在用户真实的使用场景下（不同设备、不同环境、不同任务）都成立。

我提醒大家注意一点：在早期的 ISO 9126 里，使用质量是四项——有效性、生产率、安全性、满意度；到了 25010 时代扩展成上面这五项，把"风险"和"语境覆盖"单独提出来。这背后的变化很有味道：**标准开始承认——同一个软件在不同的使用语境下，使用质量可以是完全不同的**。手机上流畅，不代表在弱网的山村里可用；办公室里好用，不代表戴着手套的车间工人能用。

再把它和验收测试挂上钩：验收测试的判定准则，本质上就是使用质量的指标化——任务完成率、平均完成时间、严重问题数、无障碍可用程度等等。如果一份验收方案里只有"功能是否符合需求"这一条，那它评的是产品，不是使用。

板书：写"**使用质量模型（用户视角）**"，横排五个：**有效性 · 效率 · 满意度 · 免于风险 · 语境覆盖**；下

面写对比："**ISO 9126 时代四项：有效性·生产率·安全性·满意度 → 25010 扩展为五项**"；右侧写一句"**同一软件，不同语境，使用质量不同**"。

提问：请举一个"产品质量不错、但使用质量很差"的例子，并说明差在哪一项上。

预设回答：

- 答"功能齐全但老年人不会用，满意度低"：✅ 很好的例子。点评：这属于**易用性与满意度**问题，它提醒我们——产品的功能适合性达标，不代表使用质量达标，**用户的达成成本和感受是独立维度**。

- 答"产品质量和使用质量是一回事，只是叫法不同"：❌ 概念混淆。点评：两者观察对象不同——**前者看产品属性，后者看使用结果**；一个产品可以特性全达标却在真实语境里失败（比如强光下屏幕看不清）。

- 答"只要用户投诉多就是使用质量差"：⚠ 不完整。点评：投诉只是满意度的信号，使用质量还要看**有效性（任务是否真的完成）和免于风险（有没有造成损失）**；很多失败的用户根本不投诉，直接流失。

易错点 / 考点：①两套模型不要张冠李戴：**产品质量模型 8 特性，使用质量模型 5 特性**；②"语境覆盖"只出现在使用质量模型里，是 25010 的新增项；③考点形式：给出用户场景判断属于使用质量的哪一项（"新用户 3 分钟完成注册"→有效性/效率；"系统不会误导用户造成资金损失"→免于风险）；④口诀：**有效、高效、满意、无风险、语境覆盖**。

过渡：光讲模型还是抽象的，下一页课件给了一个具体例子——把它套到 Web Portal 上看，使用质量到底怎么评。

【2.22】示例：Web Portal 的使用质量（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是示例，课件用一个 Web Portal（门户网站）来说明"使用质量"怎么落地。所谓门户，通常是企业的信息入口：登录、导航、公告、单点登录到各个业务系统。我们就把刚才那五项使用质量特性，一项一项套到这个门户上。

第一项有效性：用户能不能真的完成他要做的事——登录成功、找到要办的事项、点进正确的业务系统，而不是在导航里迷路。第二项效率：完成这些事要花多少时间和步骤，比如从首页到目标系统要不要点五次、加载要不要等十秒。第三项满意度：用户对界面的信任感和舒适度，包括信息是不是准确、样式是不是一致、提示是不是清楚。第四项免于风险：门户里往往挂着单点登录和员工信息，一旦被冒用或者跳转到钓鱼页面，

损失是直接的，所以风险要单独评。第五项语境覆盖：员工在电脑上用、在手机审批时用、在弱网的外勤现场用，效果可能完全不同。

大家可以发现，这里用的方法很朴素：**把抽象的质量特性，翻译成具体用户能感知的问题**。这正是使用质量评价的核心动作——不是问"这个系统质量好不好"，而是问"哪一类用户，在什么语境下，完成哪一件事，会失败"。

再补一层：门户的使用质量评价通常不是一次性结论，而要分语境、分角色来看。同一套指标，对天天使用的老员工和偶尔登录的新员工，答案可能完全相反；对电脑端和对手机端，也可能一个通过、一个不通过。所以报告要按语境分组给结论，而不是给一个全局平均分——平均值最擅长掩盖问题。

板书：写"**示例：Web Portal 的使用质量**"，左侧列五特性，右侧并列写出问句：**有效性→能否完成登录/办事；效率→要几步・等多久；满意度→是否信任舒适；免于风险→账号与信息是否安全；语境覆盖→PC/手机/弱网是否都可办**；下方写一句总结："**把特性翻译成用户能感知的问题**"。

提问：如果只能选三项指标来评价这个门户的使用质量，你选哪三项？为什么？

预设回答：

- 答"有效性、效率、免于风险"：✅ 很务实。点评：这三项**能直接量化、且直接对应业务损失**——办事失败、耗时过长、账号被冒用，都是能被统计和追责的。

- 答"满意度最重要，发个问卷就行了"：⚠ 只靠主观数据不够。点评：问卷受样本和情绪影响大，**要与客观指标互相印证**；满意度高但任务完成率低，往往说明用户根本没意识到自己用错了。

- 答"门户是内部系统，风险可以不用评"：❌ 危险的判断。点评：门户通常是**单点登录的入口，是攻击价值最高的一环**；它一旦失守，后面所有业务系统一起暴露。

易错点 / 考点：①示例类题目的答题结构：**先定用户与语境 → 再按特性逐项分解 → 最后给可测指标**；②最常见的错误是只说界面感受，不给可测口径；③考点形式：给一个系统（如选课系统、外卖 App）让你按使用质量五特性列评价角度；④口诀：**先问谁在用、在哪用，再问能不能办成、要多久、信不信、险不险**。

过渡：从这一页开始，我们的关注点从"什么是好质量"转到反面：质量出了问题，就是缺陷。先给缺陷下一个正式定义。

【2.23】2.1.2 软件缺陷的定义（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

我们进入 2.1.2，软件缺陷的定义。课件上给的核心表述是：**任何程序、系统中的问题，比如与产品设计书的不一致性、不能满足用户的需求，都属于软件缺陷。**这句话虽然短，但包含了两个很关键的判断来源，我拆开讲。

第一个来源是"与产品设计书的不一致性"。也就是说，规格说明、设计文档是基准，程序的行为跟基准不符，就是缺陷。比如需求书写着"密码长度不少于 8 位"，程序却允许 6 位，这就是不一致，不需要讨论。

第二个来源是"不能满足用户的需求"。请注意这一句的分量——它意味着**即使程序完全符合设计书，只要用户的需求没有被满足，仍然算缺陷**。为什么？因为设计书本身可能写错了、写漏了。用户说"我要能导出 Excel"，需求文档里没写，程序自然也没有这个功能，从"一致性"角度它没错，从"满足需求"角度它就是缺陷。

这两个来源正好构成了我们后面反复用的两条判定思路：**对照规格（正向思维：该做的做了没有）**和**对照用户期望（反向思维：还有什么情况让他失望）**。课件这一页还配了图，表达的也是同样的意思——缺陷的判定不能只看代码对错，还要看它是否偏离了设计和用户的期望。

所以大家记住一个判断口径：**缺陷不是"程序里有错误"这么窄，而是"任何让产品与设计或用户期望产生偏离的问题"**，它可能藏在代码里，也可能藏在需求、设计、文档甚至配置里。

还有一个实务提醒：判定缺陷时一定要写清"期望"和"实际"。只说一句"这样不对"，开发会反问"那应该怎样"；把规格条款或用户场景引出来，问题才是可讨论、可裁决的。这也是缺陷报告里最容易被新手省略、又最不能省略的一栏。

> **板书：**写"**2.1.2 软件缺陷的定义**"；下面两条并列：**① 与产品设计书不一致（对照规格）／② 不能满足用户需求（对照期望）**；右侧补一句："**判定缺陷的两把尺子：规格 + 用户期望**"；再写"**缺陷可以藏在需求、设计、代码、文档、配置里**"。

提问：需求文档漏写了一个功能，程序按文档做完了，测试该不该把它记为缺陷？请说明理由。

预设回答：

- 答"该记，因为用户需求没被满足"：✅ 符合课件口径。点评：但要点出处理方式——**这类问题的根因在需求，不在代码**，报告时要写清"需求遗漏"，让责任流向正确的位置，而不是直接指责开发。

- 答"不该记，程序符合规格说明就是对的"：❌ 很常见，但只对了一半。点评：符合规格是"通过下限"，不

是"质量上限"；**用户期望才是最终验收标准**，漏需求造成的问题最终仍会以用户投诉的形式回到团队。

- 答"记也可以，但要等需求方确认再说"：⚠ 流程上没错，时机上要小心。点评：**先把发现记录下来，再走确认流程**，不要因为"口径未定"就不记录；很多缺陷就是这样在讨论中消失的。

> **易错点 / 考点：**①缺陷定义的两个来源要能同时说出来，只答"程序错误"是不完整的；②"与设计书不一致"和"不满足用户需求"可能同时成立、也可能只成立一条，考试常给场景让你判断属于哪一种；③考点形式：判断"符合规格说明但用户不满意"是否算缺陷（算）；④记忆口诀：**两把尺子量缺陷——一把量规格，一把量期望**。

> **过渡：**定义讲完了，我们来看一个有意思的历史现场：软件缺陷这个词，最早是怎么来的？请看"First Bug"。

【2.24】First Bug（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页标题叫 First Bug，"第一只虫子"。图上的人物是 Grace Hopper，格蕾丝·霍珀，1906 到 1992 年，她是计算机史上的传奇人物：参与早期计算机的研制、推动了编译器的发展，被称为 COBOL 之母，也是美国海军的少将。
>
> 课件这一页讲的是那个著名的故事：早期计算机使用大量机电继电器，某次机器故障排查时，工程师从继电器里取出一只飞蛾，把它贴进维护日志，并写下"发现 bug 的第一个实际案例"。从此，"bug"这个词在计算机领域就流行开来，排除故障也被叫做 debugging——"捉虫"。这里补一个更严谨的细节：bug 这个说法在更早的工程领域就已经用来指代故障了，但这只飞蛾让它在计算机行业里彻底流行，成为我们今天通用的术语。
>
> 我想请大家注意这个故事里最易被忽略的一点：**为什么工程师要把一只飞虫郑重地贴进日志本、还写上一句话？** 因为这只虫子不是一个笑话，这是一份**缺陷记录**——记录现象、保留证据、说明原因、供后来人查阅。这正是本课要培养的基本功：发现问题要如实记录，留下可追溯的证据，而不是口头说一句"刚才有点问题，重启一下好了"。
>
> 同学们，测试工作看起来是在跟代码打交道，本质上是在跟"诚实"打交道。把不利于自己的结果隐去、把偶发问题当作没发生，短期看省事，长期看就是把风险留给用户。霍珀那一代人留下的不只是"bug"这个词，还有一种一丝不苟的记录习惯。

再深一层看，记录习惯背后是工程的可重复性：只有把现象、环境、步骤写下来，别人才能复现、才能验证修复、才能在下次遇到相似问题时快速定位。测试人员交出去的最有价值的资产，往往不是"我发现了bug"，而是一份别人照着就能复现的问题记录。

板书：写"**First Bug —— Grace Hopper（1906–1992）**"；下面写 **1947 年哈佛 Mark II 继电器中的飞蛾 → 贴入维护日志 → "第一个实际的 bug" → debugging（捉虫）**；右侧写一句落点："**发现 → 记录 → 留证据 → 可追溯**"。

【思政落点 1】讲什么故事：霍珀团队把一只飞蛾贴进日志本并写明"这是第一个实际发现的 bug"。怎么联系知识点：软件缺陷定义要求"可追溯的问题记录"，缺陷报告的最小要素就是现象、环境、证据、影响。落到什么价值观：**科学精神与职业诚信——如实记录每一个问题、不隐去不利结果，是测试人员的职业底线，也是工程文化的基石。**

提问：如果一次测试中发现的问题是偶发的、复现不了，你会怎么处理？直接忽略还是记录？

预设回答：

- 答"记录下来，标注偶发和已尝试的复现步骤"：✅ 正确做法。点评：偶发问题往往是**并发、时序、资源竞争**类缺陷的前兆，最容易被忽略，也最容易在生产环境放大；记录时要把环境、时间、操作序列都留下来。

- 答"复现不了就当没有，省得浪费开发时间"：❌ 这是典型的隐瞒式处理。点评：把"我没法复现"变成"系统没问题"，本质上是**用省略换安心**；真出事故时，团队连线索都没有。

- 答"先记，但优先级放低，等再出现时合并"：✅ 稳妥。点评：补充一句——**允许低优先级，但不允许不记录**；同时要注明"暂不可复现"，方便后续出现同类现象时快速关联。

易错点 / 考点：①年代与人名对应：**Grace Hopper，1906–1992，First Bug 故事**；②"bug 一词由这只飞蛾首创"是流传很广的误解，准确说法是它让该词在计算机领域流行；③考点形式：名词解释"debugging"，或简答"从 First Bug 看缺陷记录应包括哪些要素"；④口诀：**先记下来，再想办法复现**。

过渡：认识了"第一只虫"，我们回到术语本身——缺陷在中文和英文里有非常多的说法，它们之间有微妙的差别，这正是下一页那张对照表要解决的问题。

【2.25】缺陷 – Defect, Bug（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页的标题是"缺陷 – Defect, Bug"，课件用一张表列出了缺陷在中英文里的各种说法，一共九组：缺点 defect、偏差 variance、谬误 fault、失败 failure、问题 problem、矛盾 inconsistency、错误 error、毛病 incident、异常 anomy。大家第一眼可能觉得"这些不都差不多吗"？请注意，正是这些词的混用造成了团队沟通和缺陷统计里的大量扯皮，所以要把它们的关系理清楚。

所以这三者不是同义词替换，而是一条因果链。**error（错误）**通常指人犯的错，比如需求理解错了、代码写错了一位。**fault 或 defect（谬误／缺陷）**指的是留在产品里的那个错误状态，比如代码里那行错误的判断条件，它是静态存在的。**failure（失败／失效）**指的是运行时表现出来的不正常结果，比如用户提交表单后系统崩溃。

三者构成一条链：**人犯 error → 产品里留下 fault/defect → 运行时出现 failure**。这也是为什么缺陷管理里必须区分"根因"和"现象"：用户看到的是 failure，工程师要找的是 fault，而真正要改进流程去解决的是 error 的产生原因。

剩下的几个词也有各自的语境：variance（偏差）常用在"实际结果与预期不一致"；inconsistency（矛盾）强调与需求、文档之间的相互冲突；problem（问题）是最宽泛的口语说法；incident（毛病／事件）在运维和缺陷追踪系统里常指一次故障事件；anomy（异常）指偏离常规的状态。**术语可以多，但描述缺陷时的口径必须统一**，这也是缺陷报告要使用标准字段和统一分级的原因。

> **板书：**写"**缺陷的多种说法**"，左中右三列对照：**缺点 defect｜偏差 variance｜谬误 fault｜失败 failure｜问题 problem｜矛盾 inconsistency｜错误 error｜毛病 incident｜异常 anomy**；下面画链条：**error（人为错误）→ fault / defect（产品中的缺陷状态）→ failure（运行中的失效）**；旁注"**用户看到 failure，工程师找 fault，流程要治 error**"。

提问：线上用户报"系统崩溃了"，工程师说"这是环境问题不是缺陷"。这句话问题出在哪？

预设回答：

- 答"环境问题也是缺陷，因为用户受到了影响"：✅ 直击要点。点评：要补充专业表述——**用户感知到 failure，就必须先立案追踪**，至于根因是代码、配置、环境还是数据，是调查结论，不能作为"不记录"的理由。

- 答"工程师说得对，代码没问题就不是 bug"：❌ 混淆了 fault 和 failure 的层次。点评：**缺陷的定义看的是"偏离期望"，不是"代码有没有错"**；环境适配失败本身就是可移植性或兼容性上的质量问题。

- 答"两边说的不是一回事，各说各的"：⚠ 看到了术语分歧，但止步于此。点评：这恰恰说明**团队需要统一的缺陷定义和分级口径**，否则同一件事有人算 bug、有人算配置问题，统计数字永远对不上。

易错点 / 考点：① error / fault(defect) / failure 三者的层次关系是本节最核心的考点，务必按"人—产品—运行"这条线记；②九个英文词不要张冠李戴，尤其 fault 与 failure 常被互换；③考点形式：给出场景判断属于 error、fault 还是 failure（"程序员把 '>=' 写成 '>'"→error/fault，"边界值输入时结果错误"→failure）；④记忆口诀：**人错 error、物错 fault、跑错 failure**。

过渡：术语口径统一之后，还要有权威定义做依据。下一页我们就看两个标准的原话：IEEE 729 和 ISO 29119。

【2.26】软件缺陷（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给出软件缺陷的权威定义，课件引了两处。第一处是 IEEE 729（1983 年）给出的标准定义，它从两个角度描述：**从产品内部看，软件缺陷是软件产品开发或维护过程中所存在的错误、毛病等各种问题；从外部看，软件缺陷是系统所需要实现的某种功能的失效或违背。**请大家注意这个"内外两看"的结构：内部看是"问题存在"，外部看是"功能失效"，一个偏静态、一个偏动态，正好和前面讲的 error/fault/failure 呼应。

第二处是 ISO 29119 的定义，也是我们这门课后面做测试要常引用的标准。它把缺陷描述为：**组件或系统中会导致其无法执行所要求功能的瑕疵**；同时给出扩展说法：**任何依据需求规格说明、设计文档等基准而产生偏离的情况**，都算缺陷。下面还有一条非常重要的注（NOTE）：缺陷可能是在**评审、测试、分析、编译或使用软件产品及相关文档**的过程中被发现的——但不仅限于这些活动。

这条注把三件事说透了。第一，缺陷的发现手段不止"运行测试"一种，**评审、静态分析、编译告警乃至用户使用**都是发现渠道；第二，缺陷的载体不止程序代码，**文档、需求、设计**同样可能有缺陷；第三，只要偏离了基准或期望，就是缺陷，不需要等到它造成线上事故。

所以这一页请记住一句话：**缺陷是偏离期望的任何瑕疵，发现它的时机和手段可以多种多样，但判定它的尺子只有两把——基准（规格、设计、文档）和期望（用户需求）。**

最后给一个学习建议：这两个定义不要死背英文原句，要抓住它们共同的内核——偏离基准或期望，并且导致应有的功能无法实现。抓住内核，遇到没见过的表述，你也能判断它算不算缺陷。

板书：写"**软件缺陷的权威定义**"；左侧写 **IEEE 729（1983）：内部看＝开发/维护中存在的错误、毛病；外部看＝所需功能的失效或违背**；右侧写 **ISO 29119：会导致组件或系统无法执行所要求功能的瑕疵；任何偏**

> **离需求规格说明、设计文档等基准的情况**；下面写 NOTE 的三个发现渠道：**评审 · 测试 · 分析 · 编译 · 使用（不限于此）**，再画一条底线："**基准 + 期望＝判定缺陷的两把尺子**"。

> **提问：**ISO 29119 的注里说缺陷可能在"编译"和"使用"阶段被发现。请各举一个例子，并说明它们为什么容易被团队忽视。

> **预设回答：**

- 答"编译告警说明代码有问题，但大家习惯性忽略告警"：✅ 非常真实。点评：大量空指针、未初始化变量、类型截断都藏在编译告警里，**把告警当噪音，就是把缺陷放进版本库**。

- 答"用户使用中发现的问题不算缺陷，因为没有用例"：❌ 需要纠正。点评：用户是真实语境下的测试者，**生产环境的用户反馈是缺陷的重要来源**；区别只在于它是最昂贵的一种发现渠道。

- 答"评审发现的只能算建议，不能记缺陷"：⚠ 概念不准。点评：评审发现的需求二义、设计错误都是缺陷，**只是缺陷的载体不是代码**；记录时应标明发现阶段，便于做缺陷来源分析。

> **易错点 / 考点：**① IEEE 729 的"内部／外部"两分法是经典考法，要能准确复述；② ISO 29119 定义里的两个要点——"无法执行所要求的功能"和"偏离需求规格说明、设计文档等基准"；③最容易漏的就是那条 NOTE：**缺陷发现不限于测试**，还可能是评审、分析、编译、使用；④考点形式：名词解释、判断"只有运行程序发现的问题才叫缺陷"（错）、简答"缺陷可能在哪些阶段被发现"；⑤记忆口诀：**内看问题、外看失效；偏离基准、即为缺陷；发现渠道、不止测试**。

> **过渡：**到这里，第 2 章关于"质量"和"缺陷"的基本概念就讲完了——质量模型告诉我们什么是好，缺陷定义告诉我们什么是坏。接下来我们会用这些口径进入下一部分：缺陷是怎么分类的、测试又是怎么分类的。

【2.27】软件缺陷的现象（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 前面我们已经知道什么是软件缺陷。可到了实际工作里，摆在测试人员面前的问题非常具体：屏幕上出现的这个现象，到底算不算缺陷？要回答它，先得学会认缺陷的"长相"。课件把软件缺陷的现象归纳成六类。第一类，功能、特性没有实现，或者只实现了一部分——需求说要支持批量导入，结果只能一条一条录，这就是没实现。第二类，设计不合理，本身就存在缺陷——两个模块都往同一张表里写数据，谁也没加锁，代码每行看着都对，可设计从一开始就是错的。第三类，实际结果和预期结果不一致——你输入三加五，它给你九，这是最典型的缺陷现象。第四类，运行出错，包括运行中断、系统崩溃、界面混乱——点一下按钮程序

直接退出，或者弹出一堆重叠的窗口，用户根本没法用。第五类，数据结果不正确、精度不够——金额算出来多了两分钱，坐标算出来偏了几米，功能"跑通了"，但结果是错的。第六类，用户不能接受的其他问题，比如存取时间过长、界面不美观——这一类最容易吵架，因为需求文档里往往没写"多快算快""多好看算好看"，可用户就是不接受。这六类，你就把它当成一张筛查清单：拿到一个现象，先往这六类里对号入座，能对上号的，基本可以判定为缺陷；对不上号的，先别急着报，回去查需求、问用户。注意了，前三类和第四、第五类属于"硬缺陷"，有明确对错，可以直接判定；第六类属于"软缺陷"，要靠用户感受和业务约定来判定，这正是我们下一页要解决的问题。

板书：缺陷六现象 → ①功能未实现/部分实现 ②设计不合理 ③实际结果≠预期结果 ④运行出错（中断/崩溃/界面混乱）⑤数据不正确/精度不够 ⑥用户不能接受（存取慢/界面不美观）

提问：登录页面上，密码输入框比用户名输入框窄了 2 个像素。这算不算缺陷？依据是我们刚讲的哪一类现象？

预设回答：第一种答"不算，功能全都正常"——先肯定他关注了功能，再把他拉回第六类"用户不能接受的其他问题"，追问：如果不算缺陷，那 UI 走查时提的这个意见该叫什么？提示它属于界面设计与一致性范畴。第二种答"算，因为界面不好看"——肯定结论，但要求他说出对应哪一类现象，训练"用清单说话"的习惯，不能凭感觉定性。第三种答"看需求或 UI 规范怎么写"——这是最佳答案，请他说出需求里没写时怎么办，自然引到"判断依据"上，为下一页的判断准则做铺垫。

易错点 / 考点：常见混淆有三种：把"数据结果不正确、精度不够"当成"功能没实现"；把"存取时间过长"当成性能没做，忽略了它在这页属于"用户不能接受"这一类；把界面问题一律判为缺陷，不看设计规范和用户约定。考试形式多为选择或判断："下列哪项不属于软件缺陷的现象"。判断三步法：一查需求有没有要求，二看结果对不对，三问用户能不能接受。

过渡：现象能对上号，可"预期结果"到底由谁说了算？同一段行为，有人说是缺陷，有人说是特性，这就引出了下一页——软件缺陷的判断准则。

【2.28】2.1.3 软件缺陷的判断准则（8 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

上一页我们学会了从现象上认缺陷，但现象只是表面。一句"实际结果和预期结果不一致"，难点从来不在"实际结果"——跑一下就有；难点在"预期结果"从哪儿来。这个负责给出预期、并判定测试通不通过的机制，业界有个专门的名字，叫测试预言，Test Oracle。课件给的定义是：Test Oracle 就是决定一项测试是否通过的

一种判断机制；它要求把被测试系统的实际输出与所期望的输出进行比较，从而判断有没有差异，也就是判断是不是缺陷。那么预期从哪儿来？课件列了七类依据，我们分三组记。第一组是最有分量的两个来源：需求规格说明书和其他需求、设计规范文档，这是第一位的依据；还有竞争对手的产品——用户常说"人家某某软件就是这么做的"，竞品可以当参照，但要谨慎，别人的行为不等于你的需求。第二组是四类偏自动化的预言：启发式测试预言，靠经验规则判断，比如"金额不可能为负数"；统计测试预言，靠统计规律判断，比如响应时间的分布有没有偏离历史基线；一致性测试预言，靠同一系统不同实现之间的一致性判断，比如新旧版本对同一个请求的结果应当一致；基于模型的测试预言，把一个模型当作正确答案的来源，拿实际输出跟模型推演的结果比。第三组是人类预言，Human oracle，说白就是由人来判断，靠测试人员的经验、业务知识，甚至最终靠用户的感受。这一页要建立一个意识：判断缺陷不是拍脑袋，而是有依据、可追溯的。你报一个缺陷，就必须说清楚"我拿什么当预期"。

板书：判断准则（Test Oracle）＝决定测试是否通过的比较机制：实际输出 vs 期望输出 → ①需求规格说明书/需求与设计规范文档 ②竞争对手产品 ③启发式预言 ④统计预言 ⑤一致性预言 ⑥基于模型的预言 ⑦人类预言

提问：测试一个计算器应用，输入 0.1 加 0.2，屏幕上显示 0.30000000000000004。你用什么当预期结果？请说出依据属于课件里的哪一类。

预设回答：答"预期是 0.3，所以是缺陷"——肯定他发现问题的敏锐，追问依据：需求文档有没有规定精度和舍入规则？若有，属于第一类依据；若没有，就要靠启发式预言（常识规则）或人类预言来判断，并说明这类问题在不同业务里结论可能不同。答"浮点数就这样，不算缺陷"——引导他区分"实现限制"与"用户可接受"，请他从用户视角判断这句结果用户能不能接受，再强调结论必须有依据、能说服人。答"用另一个计算器比对"——这是很聪明的答案，指出这属于一致性或竞品参照的思路，同时提醒：参照物的行为同样要先确认是对的。

易错点 / 考点：最常错的是把 Test Oracle 理解成"测试工具"或者"测试用例"，其实它是"判断是否通过的机制"。第二个混淆是把"人类预言"当成不专业、靠感觉，实际上人工判断在易用性、业务规则、界面审美上是不可替代的。考试形式常见为名词解释、选择（下列哪个不属于缺陷判断依据）和案例题（给出场景问依据是什么）。记忆口诀：物、竞、启、统、一、模、人——文档为准，竞品参照，四类预言兜底，人能拍板。

【思政落点 1】：讲"判断缺陷必须有依据"，可以联系工程中的标准意识：需求文档、设计规范、行业标准就是判断是非的尺子。测试人员手里的尺子如果有偏差，报出去的缺陷就站不住脚，团队会因此反复扯皮。做质量工作的人，既要有自己的判断，更要尊重客观依据，用数据和规范说话，而不是用嗓门大小说话——这

就是科学精神在测试岗位上的具体落点。

过渡：知道了怎么判断缺陷，接下来要回答一个更根本的问题：这些缺陷到底是从哪儿冒出来的？

【2.29】2.1.4 软件缺陷的产生（7 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

缺陷从哪儿来？课件把产生原因归成三大类。第一类是技术问题。里面又分三种：算法错误，思路本身就是错的；计算和精度问题，比如浮点数累积误差、单位换算错误；接口参数传递不匹配，A 模块传来的单位是厘米，B 模块当成米收下，单看两边都没错，一对接就出事。第二类是团队工作。这一条最容易被忽视，杀伤力却最大——沟通不充分、误解。评审会上大家都点头，散会后三个人脑子里装着三个版本的需求；接口约定只写在聊天记录里，没人落进文档；跨部门、跨地域的团队各做各的。这类缺陷不是谁技术差，是信息传递断了。第三类是软件本身的问题：文档错误，说明书写错了参数；用户使用场合，也就是 user scenario 考虑不周，没想过用户会在地铁里断网操作；时间上不协调或不一致带来的问题，比如定时任务和用户操作撞在一起、几台机器的时钟不同步；还有系统的自我恢复、数据的异地备份、灾难性恢复等，这些功能平时不跑，一跑就是大事故。课件这一页还挂了参考文献，主题是"缺陷驱动的过程改进"。它在告诉我们：分析缺陷产生的原因，不是为了追责某个人，而是为了回头改流程。同一个原因反复出现，说明流程有洞，而不是某个人不小心。给你一个判断经验：如果某个缺陷让你觉得"这怎么能怪开发"，那它大概率属于团队工作或软件本身这两类，解决方案在流程和文档里，不在某一行代码里。

板书：缺陷产生三来源 → 技术（算法错误／计算与精度／接口参数传递不匹配）｜团队（沟通不充分、误解）｜软件本身（文档错误、用户使用场合、时间不协调或不一致、自我恢复与数据异地备份/灾难恢复）

提问：一个接口联调失败：前端按"秒"传超时时间，后端按"毫秒"解释，结果请求 3 秒就超时。这个缺陷属于课件所说的哪一类原因？根子上该怎么修？

预设回答：答"技术问题，接口参数不匹配"——正确，但要追问他会不会把它只当代码 bug 修，提醒他真正的修法是补上接口契约与单位约定，属于流程改进。答"沟通问题"——也说得通，表扬他从团队角度看问题，说明技术类与团队类常常并存，评审与文档化是共同解药。答"前后端都有责任"——把它变成教学素材：缺陷归因的目的是定位可改进的环节，不是分摊责任，请他给出具体改进动作（统一单位、接口文档、联调用例）。

易错点 / 考点：常错在把"缺陷产生原因"与"缺陷现象"混为一谈：现象回答"看到什么"，原因回答"为什么会有"。另一处混淆是把"文档错误"归到技术问题，其实课件把它放在"软件本身"一类。考试常考分类题（下列

哪项属于团队工作导致的缺陷）和简答（简述缺陷产生的三类原因）。记忆抓手：技术、协作、软件——写错了、没沟通清、想漏了场景。

过渡：三类原因是从性质上分，把缺陷按来源归好类之后，我们还得知道每一类各占多少、哪一类是重灾区，这就有了缺陷的构成。

【2.30】2.1.5 软件缺陷的构成（6 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

接着看 2.1.5 软件缺陷的构成。上一页讲的是原因的大类，这一页要做的是把缺陷变成可以统计、可以量化的结构。所谓构成，就是把收集到的缺陷按来源或者根源做归类统计，看每一类各占多少。课件这一页配了两张图，出处标得很清楚：一张来自 2010 年 1 月发表在 International Journal of Industrial Engineering and Management 第一卷第四期第 155 到 161 页的论文，题目是《Requirements-based testing process in practice》，讲的是需求驱动的测试流程实践；另一张来自 scopemaster 的技术博客，题目是《软件缺陷的根本原因》。这里我要特别提醒：课件图上的百分比我不要求你们背，不同项目、不同统计口径差别很大，背数字没有意义；你要学的是看图的方法。图上的分类通常包括需求类、设计类、编码类、文档类、环境与配置类、人员与沟通类。看这张图要问三个问题：第一，哪一类贡献的数量最多？第二，严重级别最高的那些缺陷集中在哪一类？第三，上一次做过程改进以后，这一类有没有下降？这三个问题回答清楚了，过程改进才有靶子，否则所谓改进就是拍脑袋。回到课程设计上，你们后面要记录缺陷，我要求每条缺陷都标注来源类别和发现阶段，目的就是让你们最后能画出属于自己项目的那张构成图，用数据说明"我们的问题主要出在需求还是编码"。

板书：缺陷的构成＝按来源/根源归类统计 → 需求｜设计｜编码｜文档｜环境与配置｜人员与沟通；图谱出处：IJIEM 2010,1(4):155-161《Requirements-based testing process in practice》＋scopemaster《root causes of software bugs》；三问：哪类最多？严重缺陷集中在哪类？改进后是否下降？

提问：如果你们项目统计出来的缺陷构成图显示：需求类缺陷占的数量最多，而且严重级别高的缺陷也几乎都在需求类。下个迭代你打算改哪三件事？

预设回答：答"让开发小心点"——点评这个答案没抓住根源，需求类缺陷靠开发小心是治不住的，引导他往评审和需求确认上想。答"加强需求评审、需求要让用户确认、需求变更要走流程"——这是好答案，帮他排优先级：先做实需求评审的检查单，再补需求基线确认。答"测试人员提前介入需求评审"——很好，进一步

点出这就是测试左移，让测试人员在需求阶段就提出二义性和可测性问题。

【思政落点 2】：缺陷构成分析里有一个职业底线问题：不能只统计"好修好说"的缺陷，把难看的、自己漏掉的、指标不利的隐去。数据一旦被美化，过程改进就失去了方向，用户拿到了虚假的质量信号。将来你们做缺陷记录和测试报告，要如实记、如实报，不因为影响考核就删除或降级——诚信是质量工作者的第一块基石。

易错点 / 考点：常错有两个：把"构成"当成"某一类缺陷的详细步骤"，其实它讲的是各类缺陷的比例结构；另一个是把构成图当成背数字的材料。考试常考简答（什么是缺陷的构成分析，它的作用是什么）和案例题（根据给定的缺陷清单做归类统计，并给出改进建议）。记忆抓手：归好类、看比例、找靶子、改流程。

过渡：构成回答的是"哪一类多"，但缺陷还有一个更隐蔽的维度——它是在什么时候被引入、又在什么时候才被发现。

【2.31】软件缺陷在不同阶段的分布（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

有了构成，再看分布。分布讲的是时间维度：缺陷是在哪个阶段被引入的，又是在哪个阶段被发现的。课件这一页给了两句关键结论。第一句：在真正的程序测试之前，通过审查、评审，就可以发现更多的缺陷。注意这个"真正的程序测试之前"，指的是代码还没跑起来、甚至还没写完的时候。为什么强调这一点？因为很多同学脑子里有个默认流程：写完代码交给测试，测试就是点点点。可事实上，评审和静态检查能抓到的缺陷数量相当可观，而且成本低得多。第二句更有意思：需求的缺陷会在需求评审、设计、编码、测试等过程中逐步被发现，很难在需求分析这一个阶段就全部发现。这句话包含两层含义。一层是需求缺陷的影响会一路向下传：设计、编码、测试各个阶段都可能因为它出问题，所以你在编码阶段看到的一个 bug，根子可能在一周前的需求文档里。另一层是别指望需求评审一次就把需求缺陷清干净，缺陷的发现是一个随时间逐步收敛的过程，所以要反复评审、持续跟踪。这页的图把两个分布画在一起：一条是缺陷被引入的阶段分布，一条是缺陷被发现或被修复的阶段分布，两条线错开的那片区域，就是缺陷潜伏的时间。潜伏得越久，暴露得越晚，代价就越大——这就直接接到下一页。

板书：分布＝引入阶段 × 发现阶段 → 引入曲线 vs 发现曲线 → 两线错开区＝缺陷潜伏期；结论①真正的程序测试之前，审查/评审已能发现更多缺陷 ②需求缺陷会经需求评审→设计→编码→测试逐步发现，难在需求分析一次清空

提问：一个上线后才被用户发现的计算错误，追查发现是需求文档里"折扣按四舍五入"这句话没写清楚。请

说出这个缺陷的引入阶段和发现阶段，并解释为什么它的代价特别高。

预设回答：答"引入在需求阶段，发现在上线后"——正确，追问中间的三个环节（需求评审、设计、编码、测试）为什么都没有拦住它，让学生看到每个环节都本可以拦住。答"是开发写错了"——借此纠正归因偏差：需求二义性没有被评审识别，问题在信息传递和评审有效性上，不全在编码。答"这种缺陷没法避免"——点评他悲观的部分有现实依据（需求缺陷很难一次清空），但要求他说出可以改善的动作，比如需求评审检查单、原型确认、用例反查需求覆盖。

易错点 / 考点：最常错的是把"缺陷引入阶段"和"缺陷发现阶段"当成同一个东西，画图时两条线重合。第二个易错点是把评审当成"走形式、浪费时间"。考试常考判断（缺陷大多是在编码阶段引入的）、识图题（读出两条曲线的错位区并说明含义）和简答（为什么说需求缺陷很难在需求分析阶段全部发现）。记忆抓手：引入是源头，发现是关口，两个关口错开多大，返工就多贵。

过渡：潜伏期越长代价越大——那到底大成什么样？课件用两个关键词给出了答案：劣质成本，以及非线性增长。

【2.32】2.1.6 修复软件缺陷的成本（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

2.1.6 修复软件缺陷的成本。这一页有两个关键词。第一个叫劣质成本，英文缩写 COPQ，Cost of Poor Quality。缺陷带来的成本就叫劣质成本。注意这个名字的立场：它不是"测试成本"，也不是"开发成本"，而是"因为质量不好而额外付出的钱"。它包含什么？返工的人力、重新测试的时间、紧急发版的加班、线上事故的赔付、客户流失、品牌受损，甚至包括你反复开会澄清、写事故报告所花的时间。第二个关键词是：非线性增长。课件明确说，缺陷不及时处理带来的成本很高，而且是呈非线性增长的。什么意思？缺陷留在系统里的时间越长，修复它的代价不是每天多一点点，而是越往后越陡。原因很直观：需求阶段的一个缺陷，当场改，就是改一句话、动一段文档；拖到设计阶段改，要改设计、改接口；拖到编码阶段改，要改代码，而且可能已经被几十处引用；拖到测试或上线以后才发现，还要加上定位成本、回归成本、错误数据的修复成本，以及对已经造成后果的追责成本。你还会发现一个规律：越晚发现的缺陷越难定位，因为代码和设计已经改了很多轮，现场都被破坏了。所以测试左移不是一句口号，它背后是一条经济规律。这里我要提醒一个常见误区：不要以为"测试花的时间越少，项目就越省"。省下来的测试时间，往往会在后期以几倍甚至几十倍的返工还回去。测试不是单纯的成本项，它是把风险成本提前锁住的保险。

板书：修复成本 → ①劣质成本 COPQ（Cost of Poor Quality）：返工＋重测＋紧急发版＋事故赔付＋客户

流失＋品牌损失 ②非线性增长：需求→设计→编码→测试→上线，越往后越陡（定位/回归/数据修复/追责成本叠加）→ 结论：测试左移＝经济规律

提问：项目快到截止日期了，有人提议"这轮回归测试砍掉一半，能省三天时间"。请你用这一页的两个概念回应他，并给出你的判断。

预设回答：答"省三天，但上线后可能要赔三十天"——好答案，让他说清 COPQ 里具体包含哪些项目，把直觉变成可计算的账：三天测试人力对比潜在的返工、赔付与流失。答"砍了就砍了，风险我不管"——不要简单否定，把它当教学素材，请他写下这个决定的风险清单，再讨论谁为劣质成本最终买单。答"只能砍低风险模块的用例，高风险必须保住"——这是最职业的答案，表扬并补充：要基于缺陷分布和影响面排序，而不是平均砍一半，这就是风险驱动的测试。

【思政落点 3】：成本曲线讲的是"欠下的质量债早晚要还"，这正好对应职业责任：明知道某个功能没有验证充分，却为了赶进度签字放行，代价最终由用户承担——历史上不少事故就是这么发生的。同学们将来在岗位上，可能会面对"能不能先上线"的压力，请记住，你有责任把风险讲清楚、写清楚，而不是替别人承担不该承担的沉默。

易错点 / 考点：常错有两个：把 COPQ 理解成"测试部门的花费"，其实它是质量不好造成的全部额外损失；把非线性增长理解成"线性增长得更快一点"，忽略了曲线后半段的陡增。考试常见为选择题（COPQ 的含义）、判断题（缺陷修复成本随阶段线性增长——错）和案例题（比较在不同阶段修复同一缺陷的代价并给结论）。记忆抓手：质量不好多花的钱＝COPQ；越晚越贵，非直线。

过渡：到这里，2.1 软件缺陷就讲完了——从现象、准则、产生、构成、分布到成本。接下来我们要问：用什么办法把这些缺陷找出来？这就是 2.2 软件测试的分类。

【2.33】2.2 软件测试的分类（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

从这一页开始进入 2.2，软件测试的分类。前面 2.1 讲的是缺陷：是什么、怎么判断、从哪儿来、怎么分布、代价多大。讲透了缺陷，接下来自然要问：用什么办法把它找出来？办法有很多，名字也很多——功能、性能、静态、白盒、回归……不先建立分类框架，这些名词就是一盘散沙，到了项目上也不知道该选哪个。所以 2.2 的任务，就是把这些方法挂到一棵分类树上去。这一节按六个步骤走：概念，这个维度是什么、什么场景用；做法，要掌握哪些知识和技术；举例，用案例帮你看懂怎么用；经验分享，讲理论与实际的差距；学习建议；以及对应的实用工具和模板。重点是分类维度和选择依据，不是背名词。学完你要能做到两件事：

拿到一个项目，能说出该做哪几类测试；听到一个陌生名词，能判断它挂在哪个维度上。

板书：2.2 软件测试的分类 → 学习路径：概念（什么场景用）→ 讲解（要掌握什么）→ 举例（案例/Demo）→ 经验分享（理论与实际的差距）→ 学习建议 → 实用工具/模板；目标：拿到项目能选类型，听到名词能定位维度

提问：在进入分类之前先热身：请凭直觉说出你知道的三种测试名称，并试着说说它们分别是从什么角度来划分的。

预设回答：答"功能测试、性能测试、安全测试"——表扬，指出这三个都是从"测试目的"这个角度划分的，属于同一根主枝，让学生体会同一维度下可以有很多类型。答"单元测试、集成测试、系统测试"——指出这是从"测试对象或范围"划分的，与上一组不是同一个角度，正好说明为什么需要多个维度。答"白盒、黑盒"——指出这是从"是否针对内部结构"划分的，属于另一个视角。三种回答都记在黑板不同位置，直接变成分类树的雏形。

易错点 / 考点：常见错误是把不同维度的名词混在一起当成平级分类（比如"单元测试和性能测试哪个先做"），考核中常有"下列分类不属于同一维度的是"。判断三步法：先问"按什么分"，再问"这一维有哪些取值"，最后才问"该选哪个"。

过渡：框架交代完了，我们先看这张分类总览图，把整节的骨架装进脑子里。

【2.34】软件测试的分类（5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是 2.2 的分类总览图，一张图把整节的骨架画出来了。看这种图有个方法：先看它从哪几个方向分叉，再看每一根分叉下面挂了哪几片叶子。图的主干就是"软件测试"这四个字，往外伸出几根主枝，对应我们后面要逐个讲的分类维度。第一根主枝，按测试的对象或范围分：单元测试、集成测试、系统测试，一直到验收测试，越往后测的对象越大、越接近用户。第二根主枝，按测试的目的分，也就是我们常说的测试类型：功能、性能、可靠性、安全性、兼容性、回归等等。第三根主枝，按被测程序有没有被执行来分：静态测试和动态测试。第四根主枝，按测试有没有盯着内部结构和实现算法来分：白盒和黑盒，中间还有灰盒。第五根主枝，按测试方法分：精准测试、变异测试、蜕变测试、MBT 这些相对进阶的方法。同学们看这张图的时候，脑子里的动作应该是：先把五根主枝记住，再把每一个你听说过的测试名词往枝上挂。挂得上去，说明你的知识是有结构的；挂不上去，说明你只是记住了名字。这里有一个非常重要的提醒：这些维度不是互相排斥的单选题，而是可以叠加的坐标。同一个测试活动，可以同时是系统测试、性能测试、动态测试、黑盒

测试，并且用基于模型的方法来设计。所以以后不要问"这个项目到底该做黑盒还是白盒"，而要问"在哪个层次、为了什么目的、用什么视角、借助什么方法"。

板书：分类总览图 → 主干：软件测试 → 五主枝：①对象/范围（单元→集成→系统→验收）②目的/类型（功能・性能・可靠性・安全・兼容・回归）③是否执行程序（静态/动态）④是否针对内部结构（白盒/黑盒/灰盒）⑤方法（精准・变异・蜕变・MBT）；维度可叠加＝坐标系，不是单选题

提问：请给"用 Selenium 脚本跑一遍电商网站的下单流程，看页面和订单数据是否正确"这个活动打上四个坐标标签。

预设回答：答"系统测试＋功能测试＋动态测试＋黑盒测试"——标准答案，请他逐个说明理由，尤其解释为什么是黑盒（脚本只操作界面、不关心内部实现）。答"自动化测试、回归测试"——认可这两点，但指出自动化属于"执行方式"这一对，回归属于"测试目的"，维度不同，帮他区分。答"白盒测试，因为脚本能看到代码"——这是典型错误，借此讲清白盒的关键是"是否针对内部结构和实现算法来设计用例"，而不是"手里有没有代码"，脚本走的是界面，仍是黑盒。

易错点 / 考点：常错在把分类维度当成分等级、分先后，或者认为一个测试只能属于一类。考试常见为多选题（下列哪些维度可以用来给同一个测试活动打标签）和识图题（从分类图中指出某名词所属维度）。记忆口诀：范围、目的、跑不跑、看里看外、用什么法。

过渡：五根主枝先记牢，接下来我们逐个拆开看，先从最常用的按对象和目的分类讲起。

【2.35】不同维度的分类（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把五个分类维度展开讲。第一个维度，按测试的对象或范围分，范围从小到大排：底层测试，也就是单元测试，测的是最小的可测单位，通常是一个函数、一个类或者一个模块；接口测试或集成测试，测的是模块拼起来之后能不能正确协作，重点在接口、数据传递和调用顺序；系统测试，把整个系统装进接近真实的环境，按需求从头到尾验证；再往上就是业务层的验收测试，由用户或者业务方确认"这是不是我要的东西"。记住一条主线：越往下，测的是"零件对不对"，执行主体多是开发人员；越往上，测的是"整机能不能交付"，执行主体逐渐转向测试人员和用户。第二个维度，按测试的目的分，也就是测试类型：功能测试看功能对不对；性能测试看快不快、扛不扛得住；可靠性测试看长时间运行稳不稳；安全性测试看会不会被攻破；兼容性测试看在不同浏览器、操作系统、设备、数据下是否都正常；回归测试看改了一处有没有碰坏别处。第三个维度，按被测软件在测试过程中是否被执行，分为静态测试和动态测试。第四个维度，按是否针对系

统的内部结构和具体实现算法，分为白盒测试和黑盒测试。第五个维度，按测试方法分：精准测试、变异测试、蜕变测试、MBT，也就是基于模型的测试，这些属于进阶方法，后面章节会逐个展开。这五个维度的顺序和课件一致，请大家按这个顺序记，不要打乱。

> **板书：**五个维度 → ①对象/范围：单元（底层）→ 接口/集成 → 系统 → 业务层验收 ②目的/类型：功能・性能・可靠性・安全・兼容・回归 ③是否执行程序：静态 / 动态 ④是否针对内部结构与算法：白盒 / 黑盒 ⑤方法：精准测试・变异测试・蜕变测试・MBT

> **提问：**一个银行 App 要上线新版本。请分别从"对象/范围"和"目的/类型"两个维度，各说出两个必须做的测试，并说明理由。

> **预设回答：**答"单元测试和集成测试；功能测试和性能测试"——达标，请他把每个测试对应的对象说清楚，特别是集成测试要盯接口和转账对账的数据一致性。答"就做一个系统测试，覆盖全部功能"——借这个答案讲"层次不能互相替代"：系统测试跑通了，不代表单元内部的分支和边界都被覆盖，缺陷定位也会困难得多。答"安全性测试最重要，因为涉及资金"——认可风险判断，同时补充：安全和性能再重要也不能替代功能与回归，要按风险排优先级而不是做单选题。

易错点 / 考点：最容易错的是把"对象/范围"和"目的/类型"混为一谈，比如认为"系统测试"和"功能测试"是同一个维度的平级名词。第二个易错点是把"验收测试"只当成一个流程节点，忘了它是测试范围维度上的最高层。考试常见为分类连线题、多选题（下列哪些属于按测试目的分类）。判断三步法：先问"测的是谁"（范围），再问"为了什么"（目的），最后问"怎么测"（静态动态、黑白、方法）。

> **过渡：**五个维度搭好了架子，但还有一组更细的说法需要单独讲——就是同样是做测试，方式上可以差得很远。

【2.36】软件测试方式（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲软件测试方式，课件列了四对，一共八种：静态测试和动态测试；主动测试和被动测试；手工测试和自动化测试；基于脚本的测试和探索式测试。为什么要把它们成对摆出来？因为它们回答的是四个不同的问题：程序跑不跑？——这是静态对动态。测试人员要不要主动去驱动系统？——这是主动对被动。用例由人来执行还是由机器执行？——这是手工对自动化。用例是提前设计好了照做，还是边测边设计？——这是基于脚本的测试与探索式测试。请注意，每一对里的两个选项之间，不是"新的取代旧的"，也不是"好的和坏的"，而是各有各的适用场景，实际项目里往往是组合使用的。举个例子说明它们能叠加：一个电商系统的支

付模块，你既可以做动态的功能测试，也可以做静态的代码评审；回归用例可以交给自动化跑，同时用探索式的手工测试专门去找自动化脚本想不到的路径；上线以后还要靠被动测试，从监控和日志里发现问题。所以学这四对，目标不是选边站，而是建立一个坐标——拿到一个测试任务，你要能说清楚它在四个坐标上分别落在哪里。接下来我们逐对来看。首先是静态测试和动态测试，这是最基础、也最能体现测试左移的一对。

板书：测试方式四对 → 跑不跑：静态 / 动态｜谁驱动：主动 / 被动｜谁执行：手工 / 自动化｜用例何时设计：基于脚本 / 探索式；四对＝四个坐标轴，可叠加组合使用

提问：请判断这句话对不对，并说明理由："自动化测试比手工测试先进，所以项目里应该尽量用自动化测试替代手工测试。"

预设回答：答"不对，有些场景必须手工"——追问是哪些场景，引导他说出探索式测试、易用性判断、需求频繁变动的部分，把结论落到"场景决定方式"。答"对，自动化效率高"——不直接否定效率这个事实，而是请他算一笔账：只跑一次的用例，写脚本的成本能不能收回来，让他自己发现自动化的前提是"可重复、稳定、量大"。答"两者结合，回归用自动化，新功能用探索式手工"——最佳答案，表扬并补一句：企业里这正是常见的分工方式，自动化守住既有功能，人去找未知的问题。

易错点 / 考点：常错在把四对方式当成互斥的选项，认为一个测试只能属于每对中的一边；第二个易错点是认为"自动化一定优于手工"。考试常见为判断、选择和简答（简述软件测试的几种方式并各举一例）。记忆口诀：跑不跑、谁驱动、谁执行、怎么设计。

过渡：四对里先看第一对，也是最基础、最容易考的一对——静态测试和动态测试。

【2.37】静态测试 vs. 动态测试（10 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

静态测试和动态测试的区别，一句话就能说清：程序有没有跑起来。动态测试是通过运行程序发现错误——你设计输入、执行、观察输出，再跟预期比，我们在上一节做的练习基本都是这一类。静态测试不执行程序，而是通过人工评审、代码扫描和静态分析来发现错误，文档评审、代码走查、工具扫描都算。这里要划清几个边界。第一，静态测试查什么？查需求文档里的二义性和遗漏，查设计里不合理的地方，查代码里的空指针、数组越界、资源没释放、命名和编码规范问题。这些问题如果留到运行时才发现，代价会大得多——有些问题比如内存泄漏，跑一次两次根本看不出来。第二，静态测试不等于"随便看看"。课件这一页特意列出了评审会议里的角色分工：主持人负责组织会议、控制节奏、确保议题不跑偏；作者是被评审对象的作者，负责讲解、答疑并记录问题；记录员负责如实记录每一个问题和决议，不能漏记、不能美化；列席人员旁听

学习并提供补充视角；内审员负责检查过程是否合规；技术专业人员提供领域内的技术判断；用户代表则从需求和业务角度把关。有了这套分工，评审才是"有产出、可跟踪"的活动，而不是一次聊天。第三，不要以为静态测试就是评审，代码的静态分析主要靠工具来做，但人工的代码评审同样不可或缺，两者是互补的。最后给一个判断口径：凡是"我读了、我看了、我扫了"的，都是静态测试；凡是"我运行了、我点了、我发请求了"的，都是动态测试。

板书：静态 vs 动态 → 静态＝不执行程序：人工评审（走查/会议评审/互为评审）＋代码扫描/静态分析 → 查需求二义、设计不合理、代码规范与隐患；动态＝运行程序：设计输入→执行→比对输出；评审角色分工：主持人（组织控场）・作者（讲解答疑记录）・记录员（如实记录）・列席人员（旁听补充）・内审员（查合规）・技术专业人员（技术判断）・用户代表（需求业务把关）

提问：一个模块刚写完，还没运行过一次。用静态方法你能查出哪些类型的问题？请至少说出三类，并各举一个具体例子。

预设回答：答"规范问题、空指针、资源没释放"——达标，请他补充"需求或设计层面的问题"，比如接口参数单位没写清、异常分支没有设计，让他意识到静态测试不只看代码。答"没运行怎么知道对不对"——这是很有代表性的误解，借此讲清：静态测试查的是"能不能对"，而不是"跑出来对不对"，规范、结构、逻辑路径上的矛盾在纸上就能看出来。答"只能靠工具扫描"——补充人工评审的不可替代性，比如变量命名歧义、注释与代码不符、业务规则理解偏差，工具往往扫不出来。

易错点 / 考点：最常错的是把"静态测试"等同于"测试文档"或者"不重要的测试"；第二个易错点是以为静态测试只针对代码，漏掉需求与设计文档。考试常见为选择（下列哪项属于静态测试）、判断（静态测试不执行程序——对）和案例题（给定一段代码，指出可用静态分析发现的隐患）。记忆抓手：不动程序就是静态；评审看文档、走查看代码、扫描靠工具。

过渡：静态和动态讲的是"程序跑不跑"，接下来这对讲的是"测试人员要不要主动去驱动系统"——主动测试与被动测试。

【2.38】主动测试 vs. 被动测试（8 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

第二对是主动测试和被动测试。主动测试，是测试人员主动出击：主动设计用例、构造输入、驱动系统、观察输出。我们平时课堂练习做的几乎都是主动测试——测什么、怎么测、什么时候测，控制权都在你手里。被动测试不一样，它不主动去打扰系统，而是在系统正常运行的过程中收集数据、观察行为，从日志、监控

指标、用户反馈、缺陷报告、线上数据里发现问题。打个比方：主动测试像体检，你自己走到医院，医生按项目给你逐项查；被动测试像戴手环，不打扰你的生活，却持续记录心率、睡眠，异常了才报警。为什么需要被动测试？因为有些缺陷只在真实环境、真实数据、真实并发下才暴露，你在实验室里构造不出来；而且被动测试不会干扰用户，特别适合线上系统、生产环境和已经在跑的产品。它当然也有局限：不主动，就不知道系统在你没观察到的地方发生了什么；没有现成的预期结果可以对照，判断更多要靠基线、统计和趋势。所以在企业里两者是配合使用的：上线前靠主动测试把关，上线后靠被动测试感知。顺带说一句，很多同学把"被动测试"理解成"消极测试、随便测测"，这是完全错的——要做好被动测试，前提是对系统非常了解，知道该采集哪些指标、该盯哪些日志、该关注哪些异常模式，工作量一点都不小。

板书：主动 vs 被动 → 主动：测试人员设计用例、构造输入、驱动系统、观察输出（控制权在测试方，如功能/回归测试）｜被动：不干扰系统运行，采集日志・监控・用户反馈・线上数据，靠基线与统计判断（适合生产环境、灰度发布、Beta 与线上监控）｜关系：上线前主动把关，上线后被动感知，配合使用

提问：产品已经上线三个月，用户投诉"偶尔下单失败，但好像又自己好了"，而且无法稳定复现。这时你更该用主动测试还是被动测试？具体怎么做？

预设回答：答"被动测试，去看日志和监控"——正确，追问看什么：失败请求的时间分布、并发峰值、下游依赖的响应时间与错误码、是否有重试和自动恢复的痕迹。答"主动测试，我写脚本一直点下单"——肯定这个动作的价值，但指出难点在于"无法稳定复现"，先靠被动测试收集线索缩小范围，再用主动测试验证假设，让两者形成闭环。答"这种问题查不出来"——借机讲灰度与埋点的重要性：可观测性本身就是设计的一部分，没有日志和监控，被动测试就无从下手。

易错点 / 考点：最容易错的是把"被动测试"理解成被动消极、不设计、不投入；第二个易错点是认为被动测试只能发现问题不能定位问题，其实通过日志与监控的关联分析，被动测试经常能给出非常精确的线索。考试常见为选择、判断（被动测试需要运行系统但不主动输入——对）和案例题（给定线上问题场景，选择测试方式并说明理由）。判断三步法：谁在驱动？系统在不在真实运行？判断依据是预期还是基线？

过渡：主动和被动讲的是"谁来驱动"，接下来这对讲的是"谁来执行"——手工测试和自动化测试。

【2.39】手工测试 vs. 自动化测试（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

最后一对，手工测试和自动化测试。手工测试，就是人按用例一步一步执行，自己观察、自己判断结果。自动化测试，是用脚本或工具来执行并比对结果，机器跑、机器判，或者机器跑、人来判。先破除一个常见误

解：自动化测试不是"更高级的手工测试"，它不是万能的。哪些适合自动化？回归测试，因为用例稳定、要反复跑很多遍；性能测试和压力测试，要模拟成百上千并发，人根本做不了；大批量的数据驱动测试；以及持续集成里的冒烟测试。哪些必须靠手工？需求还在频繁变动、脚本写了马上就废的；探索式测试，靠人的直觉、经验和联想去找问题；易用性、界面美观这类靠主观感受判断的；还有一次性的、只跑一遍的验证。从投入上看，自动化有前期成本：写脚本、搭环境、准备测试数据，还有持续的维护成本。用例只跑一两次，自动化是亏的；用例要跑上百次，自动化才划算。还有一个很重要的经验：脚本会腐化，界面一改、接口一变，脚本就红一片，所以自动化测试本身也是需要长期维护的资产，不是写完就一劳永逸。课件这一页配了三张图，画的就是手工与自动化在不同场景下的分工，以及投入与产出的对比，大家结合图来理解：图的重点不是告诉你哪个更好，而是告诉你边界在哪里。

板书：手工 vs 自动化 → 手工：人执行、人判断（适用：需求多变、探索式、易用性主观判断、一次性验证）｜自动化：脚本/工具执行与比对（适用：稳定回归、性能与并发、数据驱动、持续集成冒烟）｜成本：自动化有编写＋环境＋数据＋维护成本，重复次数足够多才划算；脚本会腐化，需持续维护

提问：你们要在两周内交付一个学生选课系统，功能还在改，回归用例大约 30 条。你会选择把回归全部自动化吗？请说明理由。

预设回答：答"会，自动化效率高"——请他算账：30 条用例、两周内可能只跑两三次，而写脚本加调试可能要三四天，还要跟着需求变化改脚本，让他自己发现不划算。答"不会，需求还在变，脚本维护成本太高，先手工跑几轮，等需求稳定再挑高频用例自动化"——这是最职业的答案，表扬并补充：可以只自动化"核心且稳定"的若干条，作为冒烟用。答"自动化能发现更多缺陷"——纠正这个误解：自动化擅长"验证已知的预期"，不擅长发现未知问题，发现新缺陷更多依赖人的探索与思考，这正是探索式测试的价值所在。

易错点 / 考点：常错有三个：认为自动化一定优于手工；把"自动化测试"等同于"性能测试工具"，忽略它只是一种执行方式；忽略脚本的维护成本，以为写完就完了。考试常见为选择、判断（所有回归测试都应当自动化——错）和简答（比较手工测试与自动化测试的适用场景）。记忆抓手：机器擅长"反复核对已知"，人擅长"发现未知"；量大稳定交给机器，变化探索留给人。

过渡：四种测试方式我们讲了三对——静态与动态、主动与被动、手工与自动化，还剩最后一对：基于脚本的测试与探索式测试，下一页接着讲。

【2.40】基于脚本的测试 vs. 探索式测试（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

上一页我们对比了手工测试和自动化测试，这一页把第 2.2 节最后一组测试方式补齐，也是最有意思的一组：基于脚本的测试和探索式测试。

先说基于脚本的测试（Scripted Testing）。它的做法很直观：先把测试设计做完，把用例一条条写清楚，再按计划执行，执行完了再分析结果。也就是说，"测试设计、测试执行、分析"这三件事被拆成了先后阶段——先想清楚，再动手，最后复盘。

探索式测试（Exploratory Testing）刚好相反，它把学习、设计、执行、分析这四件事拧成一个循环：一边了解被测软件，一边设计下一步怎么测，马上执行，看到结果再回头修正自己的理解和思路。请注意，测试设计并没有消失，它只是从"事前一次性做完"变成了"执行过程中持续进行"。这个概念最早由测试专家 Cem Kaner 提出，他强调的是测试者的自由与责任并存。

打个比方。基于脚本的测试像照着菜谱做菜：菜谱写好了，按步骤下锅，味道不对就回去改菜谱。探索式测试像老厨师试新食材：先尝一口，再换个火候，边做边判断，手上没有现成菜谱，但脑子里一直在设计。

软件行业里这两种方式都在用，而且常常混着用。需求稳定、接口明确的回归场景，比如每次发版都要跑的登录、下单、支付主流程，适合用脚本化的用例守住底线；而产品刚上线、需求还很模糊、文档只有薄薄几页的时候，你写不出准确的用例，硬写出来的用例测的往往不是真风险，这时候就该让有经验的测试人员去"边学边测"，快速把主要风险点摸出来。

所以选择依据可以记三条：需求稳定不稳定、被测对象熟不熟悉、时间紧不紧。需求稳、对象熟、要长期回归，就脚本化；需求在变、对象生、要快速摸风险，就探索式。

> **板书：**基于脚本：设计 → 执行 → 分析（先写后跑；可复用、可回归）｜探索式：学习 ↔ 设计 ↔ 执行 ↔ 分析（循环；边学边测）｜选择依据：需求稳定度 · 对象熟悉度 · 时间压力 · 是否长期回归

> **提问：**一个刚上线的新 App，需求文档只有一页，团队里也没人测过，你手上没有任何历史用例。这时候你是先花两天写一份两百条的用例库，还是先动手点一小时？为什么？

> **预设回答：**

- 答"先动手点一小时"：✅ 多数情况下对。追问：那点之前要不要有目标？——要，探索式测试不是瞎点，得带着"这次要摸清哪块风险"的任务去点，并且边点边记录。

- 答"先写两百条用例"：❌ 典型误区。点评：需求只有一页，你写出来的用例大概率是对着自己想象的规格在测，覆盖率看着高，真风险碰不到；而且两天时间对"刚上线"这种节奏太贵了。

- 答"两个都要，先探索再补脚本"：✅✅ 最接近实务。探索阶段摸到的问题，正好沉淀成后面可回归的用例。

> **易错点 / 考点：**最大的坑是把探索式测试等同于"随意乱点"（ad hoc 测试）。考试常见判断题："探索式测试就是不做测试设计、不用写用例的测试"——错。探索式测试的设计工作一直存在，只是它和执行交织在一起，并且要求测试者记录过程、复盘发现、把有效路径沉淀成可回归的用例。判断三步法：有没有带着目标去测？有没有边测边设计？有没有记录与沉淀？

> **过渡：**到这里，第 2.2 节"软件测试方式"的四组对比就全讲完了。接下来我们进入 2.3 节，讲一对更基础、更本质的划分：静态测试和动态测试。

【2.41】2.3 静态测试和动态测试（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

从这一页开始，我们进入 2.3 节：静态测试和动态测试。

这一节要回答的核心问题只有一个：测试到底要不要把程序跑起来。不运行被测程序，靠人看文档、看代码，靠工具扫代码，这叫静态测试；把程序运行起来，喂数据、看输出、比结果，这叫动态测试。这一条分界线，是后面所有测试技术的一级分类，务必先立住。

再补一句它的分量：这组分类不是学术摆设，它直接决定你下一步能拿出什么手段——是坐在评审桌上挑问题，还是到环境里去跑用例。选错了手段，力气就白费了。

本节我们按三步走：第一步，讲产品评审，也就是对需求文档、设计文档和代码的评审——解决"该不该这么写"；第二步，讲静态分析，也就是不运行程序的代码检查——解决"代码里藏着什么问题"；第三步，讲验证与确认，也就是 V&V——解决"我们做的到底对不对、是不是用户真正要的"。

请大家带着一个问题听本节：静态测试不跑程序，它凭什么能发现缺陷？如果这个问题想通了，你就会明白教材为什么反复强调"越早发现越便宜"，也会明白测试为什么要往需求、设计这些"纸面产物"上延伸。顺便提醒：这一节的关键词——评审、静态分析、验证、确认——全是后面考试的高频词，听到一个就在脑子里归好类。

> **板书：**2.3 静态 vs 动态 ｜ 分界线：是否运行被测程序 ｜ 路线：产品评审（需求/设计/代码）→ 静态分析（不跑程序查代码）→ 验证与确认（V&V）

提问：一个程序运行起来一切正常，是不是就说明它没问题？

预设回答：

- 答"是"：❌只能说明"跑到的那些路径"正常。追问：没跑到的路径、异常分支、界面文字错误、文档与代码不一致，动态测试能发现吗？

- 答"不是，还要看代码和文档"：✅对，这正是静态测试的用武之地。

- 答"得看需求"：✔️已经触及 V&V 的核心，本节最后一块内容就会展开。

易错点 / 考点：别把"静态"理解成"不动脑子地看"，也别把"动态"理解成"随便跑跑"。判断口径只有一条：被测程序有没有被执行。选择、判断题常把"代码评审""代码走查""静态扫描"和"运行程序调试"混在一起考，按这条口径一一对号即可。

过渡：先看第一块——产品评审。它管的是文档和代码的质量，也是所有测试活动里最能省钱的环节。

【2.42】静态测试（6 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

什么是静态测试？教材给的定义很干净：静态测试包括对软件产品的需求和设计文档、代码的评审，比如技术评审、文档评审等，以及对代码的静态分析。一句话，凡是"不运行被测程序、靠人对产物进行检查"的活动，基本都算静态测试。

这里有一条特别容易被忽略的边界，请注意：管理评审、流程评审不属于静态测试，它们属于质量保证，也就是 QA。为什么？因为静态测试盯的是"产品"——需求、设计、代码这些交付物；而管理评审、流程评审盯的是"过程"——项目怎么管的、流程执行得规不规范。对象不一样，归属就不一样。这条边界考试很爱考。

再看评审的主要形式，教材列了三种：互为评审（Peer review）、走查（walk-through）、会议评审（Inspection）。我们可以按"正式程度"给它们排个队。互为评审最轻，就是同事之间互相看一遍，随时可以发生；走查是作者带着几个人把自己的文档或代码过一遍，边讲边收集意见，介于正式与非正式之间；会议评审最正式，有明确的角色分工——主持人、作者、记录员、检查人员，要按检查单逐条核对，发现问题要记录、要跟踪、要闭环。前面那张图里出现的那些角色，说的就是会议评审的阵势。

最后还有一句：代码的静态分析主要采用工具进行，但人工的代码评审也不可或缺。为什么人工不能省？因

为工具只认规则，不认业务。工具能告诉你"这个变量定义了没用"，但告诉不了你"这段业务逻辑把用户等级算反了"——后者只有懂需求的人看得出来。

生活类比：静态测试像交稿前的校对。你不是把文章念给别人听（那是动态测试），而是拿着红笔逐字看——错别字、句子不通、前后矛盾、引用不准。文章还没"运行"，错误就能挑出一大半。

板书：静态测试＝不运行程序，检查"产物" ｜ 对象：需求文档 · 设计文档 · 代码 ｜ 手段：评审（Peer review / Walk-through / Inspection）＋ 代码静态分析 ｜ 边界：管理评审、流程评审 → QA，不属于静态测试 ｜ 工具为主，人工不可省

提问：下面四项里，哪一项不属于静态测试？A 需求文档评审；B 代码走查；C 管理评审；D 用工具扫描代码。

预设回答：

- 答 C：✅ 正确。管理评审评的是管理过程和项目状态，属于质量保证（QA）。

- 答 D：❌ 常见错误，以为"用工具"就不算静态测试。点评：判断标准是有没有运行被测程序，跟用不用工具无关。

- 答 B：❌ 走查是评审的主要形式之一，是标准的静态测试。

易错点 / 考点：三个高频混淆点。①把管理评审、流程评审错当成静态测试——记住"评产品是测试，评过程是 QA"。②把"互为评审 / 走查 / 会议评审"的正式程度记反——口诀"互看最轻、走查居中、会议最重"。③以为静态分析全靠工具——教材明确说人工的代码评审不可或缺。考试可能出选择题判断归属，也可能在案例题里问"以下哪些活动属于静态测试"。

过渡：三种评审形式里，用得最多、收益也最大的，是产品评审。它评的到底是什么？下一页我们专门讲。

【2.43】2.3.1 产品评审（5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

2.3.1 讲产品评审。先明确评审对象：需求文档、设计和代码。也就是说，从"要做什么"到"打算怎么做"再到"真的怎么写的"，每一个阶段的产物都要被检查一遍。

教材有两句话要划重点。第一句：通过软件评审，可以更早地发现需求工程、软件设计等各个方面的问题，大大减少大量的后期返工，将质量成本从昂贵的后期返工转化为前期的缺陷发现。说白了，就是把"花钱

改"提前成"花时间看"。第二句：评审是对软件元素或者项目状态的一种评估手段，以确定其是否与期望的结果保持一致，并使其得到改进。请注意后半句"并使其得到改进"——评审的目的不是挑毛病，更不是追责，而是让产物变好。

为什么"更早"这么值钱？回到第 1 章讲过的缺陷修复成本曲线：同一个缺陷，在需求评审时发现，改的是一句话；在设计阶段发现，改的是一张图；到编码阶段发现，改的是代码加联调；等上线之后被用户发现，改的是补丁、是数据、是口碑，甚至可能是召回和赔偿。中间差的不是一点点工作量，而是数量级。

举个身边的例子。学生选课系统里有一句需求写着"系统应支持高并发选课"。评审时有人追问：高并发是多少？一千人同时点，还是两万人同时点？课程容量怎么算、超卖怎么处理？这些问题在评审桌上问出来，答案是改几行文字；如果等到选课当天系统卡死、课程被超选，再回头找原因，那就是一场事故。

所以测试人员参加产品评审，不是去"旁听"的，而是去提问、去挑二义、去预判可测性。你在评审会上问的每一个问题，都可能是后面省下来的一轮返工。

板书： 2.3.1 产品评审 ｜ 对象：需求文档 → 设计 → 代码 ｜ 价值：后期返工 → 前期缺陷发现（质量成本前移） ｜ 定义：评估产物是否与期望一致，并使其得到改进 ｜ 测试人员角色：提问 · 挑二义 · 预判可测性

提问： "把质量成本从后期返工转化为前期发现"，这句话落到你自己的项目里，具体是哪一笔钱、哪一段时间？

预设回答：

- 答"评审多花两天，省下上线后通宵修 bug"：✅ 方向对。追问：能不能举出你们小组真实的一次返工？

- 答"评审就是开会，浪费时间"：❌ 这正是要纠正的观念。点评：不评审不等于省时间，只是把成本推到了后面；越靠后越贵，还可能贵到修不起。

- 答"需求改一句话，代码要改一片"：✅✅ 好，说明理解了"缺陷扩散"——需求阶段的错会被设计、编码逐层放大。

易错点 / 考点： 两个易错点。①记错评审对象：产品评审的对象是需求文档、设计和代码，不包括"人员绩效""项目进度"。②把评审等同于"找茬、考核"——教材强调评审是评估并改进产物。案例题里问"如何降低后期返工成本"，标准答案里一定有"尽早开展需求与设计评审"这一条。

过渡： 产品评审里最先做、收益最大的是需求评审。需求评审判到底要解决哪些问题？下一页给了五类清单。

【2.44】需求评审解决的问题（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页列出需求评审要解决的问题，一共五类。我把它当作一张"找茬清单"来用，五个字：错、漏、糊、多、乱。

第一类，不正确的需求认识。需求方说的是 A，写进文档变成了 B，大家还都以为写的是 A。比如学生选课系统里"允许学生退课"，被写成了"允许管理员替学生退课"，一个主语的变化，权限就全错了。

第二类，丢掉的需求点。该有的漏了——需求里通篇讲选课，却没人提"退课之后名额什么时候释放""密码忘了怎么找回"。这类问题的可怕之处在于，漏掉的东西不会在评审桌上喊疼，它会在上线后用户投诉时才出现。

第三类，模糊的描述。像"界面友好""操作简便""响应速度快""尽量兼容主流浏览器"，这些词人人能念、人人解释不同，全是不可测的表述。模糊的需求等于把决策权偷偷交给了开发，最后验收时谁都不认账。

第四类，多余的或没意义的需求。需求文档里塞进了根本用不上、或者与目标无关的东西：校园系统只有两万师生，却要求"支持百万级并发"。多写的不只是文字，是设计复杂度、开发工作量和测试成本。

第五类，不一致的理解。同一句话，开发读出一个意思，测试读出另一个意思，客户心里还有第三个意思。评审桌上不说穿，等到测试提了缺陷，开发一句"我按需求做的"，就变成了扯皮。

你可以把需求评审理解成给图纸做"体检"：这五类问题，就是最常查出的五种病。查得越早，图纸越好改。

> **板书：**需求评审五类问题：① 不正确的需求认识 ② 丢掉的需求点 ③ 模糊的描述 ④ 多余的 / 没意义的需求 ⑤ 不一致的理解 ｜ 口诀：错 · 漏 · 糊 · 多 · 乱

> **提问：**"系统应具有良好的用户体验"——这句话属于这五类里的哪一类？为什么必须改掉才能继续往下做？

> **预设回答：**

- 答"模糊的描述"：✅ 对。但它同时还可能引发"不一致的理解"——开发觉得是响应快，测试觉得是界面好看，客户想的是流程简单。

- 答"多余的需求"：❌ 不准确。它不是多余，而是本该说清楚却没说清楚，属于模糊。点评：区分的诀窍是问"这句话能不能验收"——不能验收的就是模糊。

- 答"改不了，用户就是这么说的"：⚠ 这正是评审要干预的地方。点评：把用户的原话翻译成可验收的指标，是需求评审的产出之一，比如"95% 的请求在 2 秒内返回"。

易错点 / 考点：五类问题容易混，尤其"模糊的描述"和"不一致的理解"——前者是单句话本身说不清，后者是多个人的理解对不上，而一句话模糊往往顺带造成理解不一致。判断题常把"模糊"归到"多余"里去设陷阱。答题时按"错、漏、糊、多、乱"五个字逐条对照，就不容易漏项。

过渡：知道了要解决什么问题，还得知道按什么尺子来判断。下一页给出需求评审的六条标准。

【2.45】需求评审的标准（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

需求评审不能凭感觉，得有一把尺子。教材给了六条标准：正确性、完备性、易理解性、一致性、易测试性、易追溯性。我给大家一个口诀：正确完备好理解，一致可测可追溯。

正确性，是说需求真的反映了用户的真实需要，没有写错、没有写偏。一条错误的需求，写得再漂亮也是负分。

完备性，是说该有的都有，功能、性能、约束、异常处理、边界情况，一个都不漏。判断完备性有个笨办法：拿业务流程从头走到尾，每一步都问一句"需求里写了吗"。

易理解性，是指描述清楚、没有二义，开发、测试、客户看同一句话，理解一致。凡是需要"再解释一下才明白"的需求，都不算过关。

一致性，是指需求之间不打架：文档内部不矛盾，跟上一版不矛盾，跟相关标准、跟已有系统也不矛盾。常见的坑是同一份文档里，前面写"学生最多选 6 门课"，后面写"最多选 8 门"。

易测试性，是指这条需求可以被验证、可以量化。这一条对在座的各位最直接——需求不可测，测试就没法设计用例，也没法判定通过与否。

易追溯性，是指每条需求都有唯一标识，能从需求追到设计、追到代码、追到测试用例，反过来也能从用例追回需求。有了追溯，才能回答"这条需求到底测了没有"和"这个用例是为哪条需求写的"。

这六条不是理论摆设。你们做课程设计的时候，拿这六条去审自己写的需求，能挑出一堆问题。

板书：需求评审六标准：正确性 · 完备性 · 易理解性 · 一致性 · 易测试性 · 易追溯性 ｜ 口诀：正确完备好理解，一致可测可追溯

提问：这六条里，哪一条最直接决定"测试能不能开工"？请说明理由。

预设回答：

- 答"易测试性"：✅ 方向正确。追问：如果一条需求不可测，你作为测试人员该怎么办？——不是放弃，而是推动把它改写成可量化、可判定的表述。

- 答"完备性"：✔️ 也很关键，缺了需求就没法设计覆盖；但完备性缺的是"内容"，易测试性缺的是"可验证性"，后者直接卡住用例设计。

- 答"一致性"：⚠ 一致性差会让用例互相冲突，但通常可以靠澄清解决，卡不死测试。

易错点 / 考点：六条标准几乎每年都会以简答或选择的形式出现，务必按口诀背全，尤其别把"易测试性"漏掉——它最容易被学生忽略，却和工作最相关。另有一个高频概念题：需求"可测试"的标准表述是什么？答题要点是"可量化、有明确判定准则"。选择题里常把"可维护性""可移植性"混进来当干扰项——那是 ISO/IEC 25010 的质量特性，不是需求评审标准。

过渡：标准有了，方法呢？下一页讲怎么把需求评审真正做好——八条注意事项。

【2.46】如何做好需求评审？（6 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲方法：怎么才能把需求评审做好。教材给了八条，我按"会前、会中、会后"帮大家归一下类。

会前两条。第一，有明确的评审标准，比如准备一份需求质量 checklist——就把上一页那六条当尺子。第二，熟悉评审内容，为评审做好准备。这一条最实在：参会前把需求文档读一遍，把看不懂的地方标出来，把想问的问题写下来。临时进会场才第一次读文档的人，只能跟着别人点头，评审会就变成了走过场。

会中四条。第三，该参加的人都需要参加。需求评审最怕两种缺席：写代码的没来，和真正用系统的人没来。第四，针对问题阐述观点，而非针对个人。可以说"这段描述会导致权限判断错误"，不要说"这是谁写的"。第五，从客户角度想问题，多问几个为什么。不要只问"这个功能怎么实现"，更要问"用户为什么要这个功能、不做行不行"。第六，在会前或会后提出自己建设性的意见。有些复杂意见当场说不清，写成书面意见反而更有效。

会后两条。第七，对发现的问题跟踪到底。记录下来的问题必须有人改、有人复核、有人关闭，不能散会即结束。第八，针对需求文档等报告问题——把问题落成书面记录，而不是停留在口头，否则过两天就没人认账。

最后还有一句总要求：检查需求定义是否合理、清楚，是否具有可测试性。这三问可以作为每次评审的自检清单。

板书：会前：明确 checklist · 熟悉内容做准备 | 会中：该到的都到 · 对事不对人 · 站在客户角度多问为什么 · 提建设性意见 | 会后：跟踪到底 · 书面报告问题 | 自检三问：是否合理 · 是否清楚 · 是否可测

提问：评审会上，你发现同事负责的那段需求有严重二义性，很可能引发返工。你打算怎么开口？

预设回答：

- 答"直接说，这段写得有问题"：⚠ 事实没错，但把"问题"和"人"绑在了一起。点评：换成"这段描述可能有两种理解，我们确认一下是哪种"，效果完全不同。

- 答"不说，免得得罪人"：❌ 这是最坏的选择。点评：含糊的需求一定会变成后期的返工或线上缺陷，代价由整个团队和用户承担。

- 答"先私下沟通，再在会上正式提出来"：✅✅ 很成熟。既照顾了对方的感受，又保证问题被记录和跟踪。

易错点 / 考点：八条里最常考的是"针对问题而非针对个人"和"对发现的问题跟踪到底"，案例题常设"评审会开成了批斗会"或"问题记录了没人管"的情境，让你分析并给出改进措施。另外要记住：评审不是一次性会议，而是一个"发现—记录—修改—复核—关闭"的闭环，缺了后半段，前面全白做。

【思政落点 1】请记住评审席上的两条底线。第一条叫"对事不对人"：我们用证据和事实说话，指出的是文档里的缺陷，不是同事的能力。第二条叫"跟踪到底"：发现了问题却因为怕麻烦、怕担责而放过，等于把风险转嫁给用户和团队。将来你们做测试、报缺陷，一定会遇到"这个 bug 报上去领导不高兴"的时刻——请记住，如实记录、如实上报，是对用户负责，也是对自己的职业信誉负责。质量这条线，是靠一个个不肯将就的人守住的。

过渡：需求评审判的是"做什么"。接下来，我们要评"打算怎么做"——设计评审。

【2.47】设计评审（5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

需求评审核对了"要做什么"，设计评审要核对"打算怎么做"。教材把设计评审的目标讲得很明确：保证需求

能在设计中得到准确和完整的表示，也就是保证系统架构设计和产品功能规格说明书的质量。它评的核心产物，是设计规格说明书。

这一页的图给了四个抓手，我逐一拆开。

第一，系统架构。看分层是否清晰、模块之间耦合紧不紧、有没有单点、扩展和降级怎么考虑。架构评审是"高层"的评审，改起来最贵，所以最应该在纸面上就改掉。

第二，可测试性。这是测试人员最该盯的地方：接口有没有暴露、能不能 mock、关键路径有没有日志、内部状态可不可观测、能不能构造出需要的数据。可测试性差的设计，测试阶段只能靠"点点点"，覆盖率上不去，问题也定位不了。

第三，系统部署。部署拓扑、环境依赖、配置管理、升级与回滚方案，这些看着离测试很远，其实直接决定你能不能搭出一套可复现的测试环境。

第四，从高层到底层。设计评审不是一次过，而是层层往下：先审总体架构，再审模块设计、接口设计、数据库设计，最后审到产品功能规格说明书。借助 UML 等建模工具，用类图、时序图、状态图把设计"画出来"，画不清、看不懂的地方就是风险点。

这里面最需要各位记住的一句话是：不断从测试角度去问开发。设计评审不是开发的独角戏，测试人员问的每一个"这里怎么测""出错了怎么发现"，都是在为后面的测试省时间。

> **板书：**设计评审 ｜ 目标：保证需求在设计中得到准确、完整的表示（系统架构设计＋功能规格说明书） ｜ 抓手：系统架构 · 可测试性 · 系统部署 ｜ 方式：从高层到底层 · 借 UML 等建模工具 · 不断从测试角度问开发

提问：作为测试人员参加设计评审，你最该问的三个问题是什么？请给出你的清单。

预设回答：

- 答"这个模块怎么测？出错了怎么发现？"：✅ 直击可测试性。可追问：如果没有日志、没有独立接口，你还能怎么测？

- 答"需求里这条功能在设计里有没有体现？"：✅ 这正是核对"需求的准确、完整表示"，属于设计评审的本职。

- 答"这段代码为什么这么写？"：❌ 越位了。点评：代码怎么写是编码阶段和代码评审的事；设计评审关注结构、接口、部署与可测试性，顺序错了会失焦。

易错点 / 考点：常见混淆是"设计评审"和"代码评审"的边界：设计评审看架构、接口、数据、部署与可测试性

这些"纸面设计"，代码评审看具体实现。判断题还爱考"设计评审只在编码前做一次"——错，它随设计逐步细化、从高层到底层持续进行。记忆抓手："架构—可测—部署"三问，再加一句"从测试角度问开发"。

过渡：设计评审到底能拦下哪些问题？下一页给了一张很典型的清单。

【2.48】设计评审解决的问题（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页把设计评审常能拦下的问题列了出来，我逐条讲，你会发现每一条都对应后面测试阶段的一场噩梦。

第一，是否有设计规范。团队有没有统一的分层方式、命名约定、接口约定、异常处理约定。没有规范，代码就是"每人一套写法"，后期维护和测试都无从下手。

第二，系统架构设计的不合理。比如该分的层没分、耦合过紧、职责不清，一个改动要牵动半个系统。架构一旦定型，改造成本极高。

第三，单点失效。某个组件、某个服务、某个数据库一旦挂掉，整条业务链就断了，既没有备份也没有降级。这类问题在设计图上看得见，在代码里反而看不出来。

第四，数据的不完整性、不一致性。字段该有的没有、该唯一的没约束、同一份数据在多处存储口径不一，最后表现成"同一个学生在两个页面看到的成绩不一样"。

第五，缺乏可测试性。没有对外接口、不能单独运行、状态不可观测、测试数据造不出来。这一条直接决定测试的效率和可信度。

第六，具体功能设计的问题。比如流程走不通、异常分支没考虑、权限设计有漏洞——这些是"在设计层面就能看出来的功能缺陷"。

结论很清楚：这些问题在设计评审阶段被发现，代价是改图纸；留到系统测试阶段，代价就是改代码、改数据、改部署，甚至推翻重来。

板书：设计评审常见问题：设计规范缺失 · 架构设计不合理 · 单点失效 · 数据不完整 / 不一致 · 缺乏可测试性 · 具体功能设计缺陷 ｜ 一句话：图纸上改便宜，代码里改昂贵

提问：为什么"缺乏可测试性"必须在设计评审阶段解决，而不是等测试执行时再抱怨？

预设回答：

- 答"测试阶段已经改不动了"：✅ 对。设计定型后再加接口、加日志、加 mock 点，等于返工重构，还要重新回归。

- 答"可测试性是测试人员的事"：❌ 需要纠正。点评：可测试性首先是设计属性，测试人员只能提前提要求、给建议，却没法替开发做设计。

- 答"只要功能对了，可测试性差点没关系"：❌ 危险。追问：一个没有日志、没有独立接口的模块线上出问题，你怎么定位？——功能对不代表可维护、可诊断。

易错点 / 考点："单点失效"和"数据不一致"是设计评审的经典考点，案例题里常给一段架构描述让你找问题，答题时要能说出"问题是什么、会造成什么后果、怎么改"。另外记住判断口径：凡是"改起来要动结构"的问题，都该在设计评审解决；凡是"改一行代码"的问题，可以留到代码评审。

过渡：评审是"靠人看"，静态分析是"靠技术和工具看"。2.3.2 节，我们从这里开始。

【2.49】2.3.2 静态分析（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

进入 2.3.2，静态分析。这一页只有两张图，我请大家把它当成一张"全景图"来看，看图的时候抓三件事。

第一件，静态分析管什么。它不运行程序，而是把需求和设计模型、源代码当成分析对象，从中提取信息，检查程序逻辑上的各种缺陷和可疑的程序构造。所以它的产出通常不是"通过 / 不通过"，而是一批"值得你去看一眼"的发现。

第二件，谁来分析。一头是人，一头是工具。人靠经验和理解，工具靠规则和算法，两者不是替代关系，而是配合关系——下一页我们就把这两种情况拆开讲。

第三件，它站在整个测试流程的什么位置。静态分析站在最前面，紧挨着编码。这时候程序还没跑起来，问题还在"纸面"上，改起来的代价最小。

请带着一个判断标准去看这两张图：凡是"不运行被测程序、直接对代码或模型做检查"的，都是静态分析。至于它具体分成哪两种情况、用了哪些技术、怎么从代码里一层层抽信息，接下来两页会逐层展开。

板书：2.3.2 静态分析 ｜ 不运行程序，直接分析需求 / 设计模型 / 源代码 ｜ 两类执行者：人工（经验）＋

工具（规则、算法）｜位置：紧贴编码，越早越便宜

提问：静态分析的产出，为什么通常不是"合格 / 不合格"，而是一批"可疑点"？

预设回答：

- 答"因为工具会误报"：✅ 关键原因。静态分析从程序里提取信息做推断，推断不等于事实，所以结果需要人来判断。

- 答"因为它不运行程序，拿不到真实结果"：✅ 也对。没有运行结果，就只能提示"可疑"，不能判定"失败"。

- 答"因为它只检查格式"：❌ 太窄了。点评：静态分析不只查格式和风格，还能查程序逻辑缺陷与可疑构造，这一点下一页会展开。

易错点 / 考点：别把静态分析窄化成"代码风格检查"。教材的定义包含两层：既看代码是否满足规范性、质量要求，也检查程序逻辑中的缺陷与可疑构造。选择、判断题常把"静态分析能发现运行时的性能瓶颈"设为错误选项——性能必须运行程序才能测，属于动态测试。

过渡：静态分析具体分成哪两种情况？下一页给出答案：人工检测和计算机辅助静态分析。

【2.50】静态分析分为两种情况（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

静态分析分两种情况，教材讲得很清楚：一种是人工检测，一种是计算机辅助静态分析。

先说人工检测。它偏重于编码风格、质量的检验，由人对设计、代码进行分析，能有效地发现逻辑设计和编码错误。注意"有效"这两个字——人工最擅长发现的是"需要理解意图才能判断"的问题，比如命名与语义不符、注释和代码打架、边界条件想漏了、异常处理写反了。这些问题工具几乎看不出来，因为它需要知道"这段代码本来想干什么"。

再说计算机辅助静态分析。它利用静态分析工具对被测程序进行特性分析，从程序中提取一些信息，以便检查程序逻辑的各种缺陷和可疑的程序构造。工具的优势是快、全、不知疲倦：几十万行代码几分钟扫完，规则一视同仁，还能出各种度量报告。但工具的短板也很明显：它不懂业务，只会按规则报警，容易误报，也可能漏报。

所以两者是配合关系，不是替代关系。现实中的用法是：先用工具做"初筛"，让它把可疑点全扫出来、按严

重程度排好序；再由人做"复判"，逐条判断哪些是真问题、哪些是误报、哪些虽然报警但业务上无所谓。工具负责广度，人负责深度。

打个比方，这就像体检。仪器负责测指标——血压、血糖、心电图，又快又客观；医生负责看报告下结论——这个指标确实高，但结合你的具体情况要不要紧。只信仪器会吓人，只凭感觉会漏病。

> **板书：**静态分析两种情况 | ① 人工检测：偏重编码风格与质量，擅长发现逻辑设计和编码错误 | ② 计算机辅助静态分析：工具提取程序信息，检查逻辑缺陷与可疑构造 | 关系：工具初筛（广度）＋人工复判（深度）

提问：你用一个静态分析工具扫了项目，报出 300 条告警。你打算怎么处理这 300 条？

预设回答：

- 答"逐条从头看一遍"：⚠ 态度对，但方法笨。点评：先按规则类型和严重级别归类、排序，优先看高危项和集中出现的规则，效率高得多。

- 答"先按严重程度分级，再逐类确认"：✅ 实务做法。追问：如果某一类规则误报率很高，你会怎么处理？——可以调规则或加白名单，但要有记录，不能直接关掉不看。

- 答"300 条太多，先不管"：❌ 最危险。点评：告警堆积正是静态分析工具被弃用的头号原因；正确做法是建基线——新代码不新增告警，存量逐步收敛。

易错点 / 考点：高频考点是把两种方式的能力边界搞混：人工检测偏重编码风格与质量，能发现逻辑设计和编码错误；工具负责特性分析、提取程序信息、发现可疑构造。还要记住工具不能替代人工。考试若问"静态分析能否完全自动化"，答案是否定的。

过渡：两种方式里，"计算机辅助静态分析"背后是一整套代码分析技术。下一页我们走进代码静态分析的流水线。

【2.51】代码静态分析（6 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页讲代码静态分析的原理。教材的定义是：通过词法分析（Lexer）、语法分析（Parser）、控制流分析、数据流分析等技术对程序代码进行扫描，验证代码是否满足规范性、质量要求等。

我们顺着流水线走一遍。第一步，词法分析。程序源代码经过词法分析器（Lexer），被切成一个个 Token，也就是各种不同种类的单词：关键字、标识符、常量、运算符、分隔符。这一步只认"词"，不认"句"。第二步，语法分析。Token 交给语法分析器（Parser），它做分析和语法检查，产出一棵抽象语法树（AST）——树形结构把"谁是谁的组成部分"表达清楚。走到这里，工具才算"读懂了句法"。

再往下是两座桥。一是控制流：通常用控制流图（CFG）来表示程序的控制流，节点是基本块，边是可能的跳转，分支和循环一眼就能看出来。二是数据流：通常用静态单赋值（SSA）来表示程序中数据的使用-定义链（Use-Def Chain），每个变量只被赋值一次，于是"这个值从哪儿来、到哪儿去"被追得清清楚楚。前面说的"变量定义了没用""用了没定义"，就是在这两座桥上被抓出来的。

教材还提到更高级的一类技术：静态分析可以采用模拟程序执行的技术，比如符号执行、抽象解释、值依赖分析，并采用数学约束求解工具进行路径约减或可达性分析，以减少误报、增加效率。通俗地说，符号执行就是拿"符号值"代替具体输入，把程序当数学题去推；路径一多就会"路径爆炸"，这时候用约束求解做路径约减和可达性判断，把走不到的路径砍掉，效率才上得来。

回头看前面那两张图：左边是"程序怎么被读懂"，右边是"读懂之后能查出什么"。这条流水线的每一级，都对应着一类能查出来的缺陷。

板书：程序 → Lexer（词法分析）→ Token → Parser（语法分析）→ AST ｜ 控制流：CFG ｜ 数据流：SSA（Use-Def Chain） ｜ 高级：符号执行 · 抽象解释 · 值依赖分析 + 约束求解（路径约减 / 可达性分析 → 减少误报）

提问：为什么"变量定义了没用""变量用了没定义"这类问题工具能轻松发现，而"这个循环少跑了一次"却很难？

预设回答：

- 答"前者是语法和数据流问题，后者是语义问题"：✅ 抓到要害。追问：数据流分析为什么能抓到"定义未使用"？——因为它建了 Use-Def 链，赋值和使用的关系是显式的。

- 答"循环跑几次要看运行结果"：✅ 对。循环次数往往依赖具体输入或复杂条件，静态推不出来，得靠符号执行，或者干脆交给动态测试。

- 答"工具不行，人工扫一遍就行"：⚠ 部分对。人工也能看出循环边界问题，但几十万行代码靠人扫不现实；正确姿势是工具负责可判定的模式，人负责需要理解意图的判断。

易错点 / 考点：这一页的专业词最容易记混：Lexer 做词法分析、产出 Token；Parser 做语法分析、产出 AST；CFG 管控制流，SSA 管数据流。答题时别把 CFG 和 SSA 的功能对调。另一个高频点：符号执行、抽象

解释属于"模拟程序执行"的静态分析技术，注意它们仍然不真正运行被测程序，别误判成动态测试——这是选择题的经典陷阱。

过渡：静态测试的手段讲完了——评审和静态分析。但它们有一个共同的局限：只能验证"符合不符合规格"。那"规格本身对不对"由谁来管？下一页，验证与确认。

【2.52】2.3.3 验证与确认（V&V）（6 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是 2.3 节的收口，也是整章最经典的一对概念。教材说：软件测试是由"验证（Verification）"和"有效性确认（Validation）"活动构成的整体。注意"整体"两个字——它是两件事合起来的，缺一不可。

先看验证，Verification，教材给的原问句是：Are we building the product right？我们是否正确地构造了软件？也就是"是否正确地做事"，验证开发过程是否遵守已定义好的内容，验证产品满足规格设计说明书的一致性。它的参照物是——规格设计说明书。

再看有效性确认，Validation，原问句是：Are we building the right product？我们是否构造了正是用户所需要的软件？也就是"是否正在做正确的事"，验证产品所实现的功能是否满足用户的需求。它的参照物是——用户需求。

一个是"正确地做事"，一个是"做正确的事"。口诀记牢：验证看文档，确认看用户——跟规格说明书比是 Verification，跟用户需求比是 Validation。

举个例子。团队按设计说明书严格实现了"成绩报表导出"：字段、格式全对，验证 100% 通过。可用户真正想要的是"成绩一出来就自动推送到手机"，报表根本没人下载。功能没错，方向错了——这就是典型的"验证通过、确认失败"。

再打个比方：盖房子时监理拿图纸核对钢筋型号、承重墙位置，这是验证；你搬进去住才发现厨房小得转不开身——图纸不合你的需要，这是确认出了问题。

完整的 V&V 告诉我们：既要问"有没有按规格做对"，也要问"这个规格是不是用户真正要的"。前者靠评审、静态分析和测试查一致性，后者靠需求评审、用户验收和真实使用查价值。

板书：2.3.3 V&V ｜ Verification（验证）：Are we building the product right？正确地做事 → 对照"规格

> 设计说明书"｜Validation（有效性确认）：Are we building the right product？做正确的事 → 对照"用户需求"｜口诀：验证看文档，确认看用户

> **提问：**一个项目的测试用例 100% 全部通过，能说明这个软件成功吗？为什么？

> **预设回答：**

- 答"不能，用例只覆盖了规格，没覆盖用户真实需要"：✅✅ 标准答案。追问：那这些用例白跑了吗？——不白跑，验证通过说明实现与规格一致，这是必要但不充分的条件。

- 答"能，测都过了还不好？"：❌ 混淆了 Verification 和 Validation。点评：用例的设计依据通常是规格说明书，它证明的是"正确地做事"，证明不了"做正确的事"。

- 答"还要看用户验收"：✅ 对，验收测试正是 Validation 的主要手段，用户说好用才算数。

> **易错点 / 考点：**这是全章最容易考、也最容易错的一对概念。①中译名易混：Verification 一般译"验证"，Validation 译"确认（有效性确认）"；英文题干问哪个对应哪个，一律按参照物判断。②参照物易混：记住"验证看文档、确认看用户"。③常见陷阱选项："验证和确认是一回事"——错，两者的手段与视角不同，但目标统一，都是为了提高质量。判断三步法：先找参照物 → 再定 V 还是 V → 最后判断这句描述对不对。另外记住：评审是静态的 V&V，测试是动态的 V&V。

> **【思政落点 2】**这一页其实藏着一个很硬的职业态度：不要满足于"文档上通过了"。规格说明书也是人写的，它也可能错、也可能过时。真正严谨的工程思维，是既不迷信文档，也不迷信自己的实现，而是始终追问一句"用户到底要什么"。将来你们做测试、写报告，如果发现"功能全部通过但没人愿意用"，请如实写出来，而不是把不利结论藏起来——诚实报出真相，比给出一个漂亮的通过率更重要。这就是科学精神在测试岗位上的样子。

> **过渡：**到这里，2.3 节的三块内容——产品评审、静态分析、验证与确认——全部讲完，静态测试与动态测试这条主线也就闭合了。下一页，我们把视角从测试方法转到测试工作本身，看看软件测试到底分哪些层次、有哪些岗位分工。

【2.53】验证和确认（6 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页的标题只有四个字——验证和确认，配了一张图。别看字少，它是整个测试行业的两个基本坐标，我

> 们后面讲的每一个测试活动，归根到底都在回答这两个问题里的一个。

>

> 先说验证。验证回答的是：我们是不是把产品做对了。说得完整一点，就是检查产品有没有按照事先定好的规格说明、设计文档、需求基线去实现。它的参照物是文档，是"我们当初说好要这么做"。审阅需求、审查设计、代码走查，单元测试里按语句和分支设计用例，这些都属于验证。

>

> 再说确认。确认回答的是：我们是不是做了对的产品。它的参照物不是文档，而是用户和真实需求——用户拿到这个东西，能不能解决他的问题。验收测试、用户试用、在线试运行，这些都属于确认。

>

> 为什么必须把这两件事分开？因为它们会各自出错。只做验证，可能把一份本来就写歪的规格说明百分之百地实现出来——程序一行没错，用户却用不了。只做确认，可能用户说"能用"，但内部实现全是临时补丁，一改就崩。两个都做，才既有过程质量又有结果质量。

>

> 生活里的类比很好记：盖房子，验证是拿图纸对尺寸、对钢筋标号，一条一条核对；确认是搬进去住几天，看采光、看漏水、看一家人住得顺不顺。图纸全对，不代表住得舒服。

>

> 落到我们的学生选课系统：需求写"支持按学号查询选课结果"，验证就是检查代码有没有按设计实现这个查询；确认就是请教务老师真的用一遍，看她能不能顺手找到她要的那个学生的记录。

>

> **板书：**验证 Verification → 做对了吗（符合规格）→ 参照物＝文档／设计｜确认 Validation → 做对的产品吗（满足需求）→ 参照物＝用户／真实需求｜口诀：验证对内看文档，确认对外看用户

> **提问：**我们的单元测试里，按设计文档检查一个函数有没有实现对非法输入的校验，这是验证还是确认？为什么？

> **预设回答：**①说是确认——因为校验的是用户数据；②说是验证——因为对照的是设计与规格；③说两个都算。教师点评：标准答案是验证，因为手里的参照物是文档。但如果"非法输入必须拦截"这条规则最初来自用户投诉，那这个活动其实已经带上了确认的成分——这说明验证与确认在实践中经常发生在同一个活动里。

> 判断的关键就靠一句话：你手里的参照物是文档，还是用户？追问一句：那验收测试的参照物是什么？

> **易错点 / 考点：**①把"验证"和"确认"当同义词，中英文都容易混，要注意 verification 与 validation 是两个词；②以为两者严格分阶段、先验证后确认——实际上在需求阶段两者就可以同时开始，这正是测试左移。判断三步法：一看参照物（文档／用户）→ 二看问题（做对了吗／做对的产品吗）→ 三看结论（符合规格／满足需求）。考试常出情景选择题："开发人员按详细设计评审代码"属验证；"邀请用户试用新版选课系统"属确认。

> **过渡：**有了验证和确认这两个坐标，我们再换一个维度——测试人员到底出不出手？下一页讲主动测试和被动测试。

【2.54】2.4 主动测试 vs. 被动测试（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页正式进入 2.4 节，按"谁在驱动被测对象"把测试分成两类。

>

> 课件给的定义很清楚。主动测试，是测试人员主动操作被测对象，比如输入数据、发送请求等来驱动被测对象，从而验证被测对象的响应或者输出结果。被动测试，是测试人员不干预产品的运行，而是被动地监控产品在实际环境中运行而获得系统运行的数据，然后进行分析。

>

> 主动测试我们最熟：点按钮、填表单、发接口请求、写自动化脚本跑上千条用例，全是主动测试。它的好处是能构造场景，能把边界值、非法值、极端组合都喂进去，而且有明确的预期结果可以比对。它的短板也在这儿——你只能测到你想得到的路径，真实用户那些"想不到"的操作序列，你构造不出来。

>

> 被动测试正好补上这一块。它不打扰系统，去监控线上真实的流量、日志、崩溃上报、响应时间，从海量真实数据里找异常模式。它最大的好处是数据真、场景全、对用户零打扰；最大的短板是没有标准答案——你看到一个响应时间是 1.8 秒，这算慢还是正常？没有需求基线、没有预期输出，只能靠趋势和对比来判断，也就是我们常说的"预言问题"。

>

> 打个比方：主动测试像体检，医生让你抬手、深呼吸，按项目逐条测；被动测试像给你戴一个 24 小时心率手环，你该干嘛干嘛，它默默记录。体检能查出专项指标，手环能发现你自己都没察觉的日常异常，两者谁也替代不了谁。

>

> 在工程里，主动测试主要用于研发和测试阶段，被动测试主要用于线上运行阶段，也就是我们下一页要讲的在线测试。考试常这样出：给一段活动描述，让你判断属于哪一类，判据就是"有没有人为注入输入"。

>

> **板书：**主动测试 → 测试人员注入输入／发请求 → 驱动对象 → 比对预期输出（可构造场景、有预言）｜被动测试 → 不干预 → 监控真实运行数据 → 事后分析（数据真实、无预言）｜判据一句话：有没有人为注入输入

提问：我们上线后接了崩溃上报和 APM 监控，运维同事每天看错误率曲线。这属于主动测试还是被动测试？它能不能替代功能测试？

> **预设回答：**①"被动测试，不能替代"——正确；②"这是运维不是测试"——需要纠正：按教材口径，它正是被动测试／在线测试的一类，属于测试工作范畴；③"有了线上监控就不用写用例了"——典型错误。追问：如果没有一条功能用例说清"按学号查询应该返回什么"，监控发现接口报 500，你能不能判断它错在哪儿？借此讲清被动测试缺少预期结果这个先天短板。

易错点 / 考点：①把"被动"理解成"不测"——被动测试依然是测试，只是不注入输入；②把"自动化测试"等同于主动测试，其实线上拨测也是自动化的；③误以为被动测试只能发现性能问题，它同样能发现崩溃、数据异常和业务流程断点。记忆口诀："主动动手，被动动眼"。判断题常见陷阱："被动测试不需要设计"——错，监控点、判定阈值、对比组同样是设计出来的。

> **过渡：**讲了被动测试，自然要问：产品在真实环境里跑、真实用户在用的那种测试，具体长什么样？下一页给了一个名字——在线测试。

【2.55】实例：在线测试（Product-in Testing）（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页只有一个标题加一张图，讲的是一个非常现代的测试形态：Product-in Testing，直译是"产品内测试"，我们课堂上就叫它在线测试。

>

它的核心思想一句话：把测试从实验室搬到真实的生产环境，让真实用户、真实数据、真实并发来当测试的输入源。传统做法是我们在测试环境里模拟用户，模拟得再像也只是模拟；在线测试是直接把新版本用灰度、用功能开关的方式放给一小部分真实用户，一边跑一边采集数据，用数据判断它到底行不行。

>

课件这张图，要大家看出三件事。第一，被测对象不再是代码分支，而是跑在真实环境里、有真实数据的产品。第二，测试数据不再是我们编的，而是真实用户产生的流量、点击、报错和反馈。第三，判断标准从"用例通过没通过"变成"线上指标有没有变坏"——错误率、响应时间、崩溃率、业务转化率。

>

为什么企业越来越依赖它？因为有些问题只在真实规模下才出现：真实的并发量、真实的数据脏乱程度、真实机型的兼容性、真实的网络抖动。这些东西在测试环境里复现不出来，或者复现成本高得离谱。

>

但代价必须讲清楚：在线测试的对象是真实用户，一旦出问题就是真事故。所以它有一整套安全措施——先小比例灰度，观察再放量；关键功能挂功能开关，随时可以关掉；预设止损阈值，指标一超标立刻回滚；同时留对比组，才能区分"是版本变坏了"还是"今天流量本来就大"。

>

回到我们的课程项目：选课系统正式开放前，先让一个班试用，盯住接口错误率和响应时间，没问题再放开给全院。这就是最小规模的在线测试。

>

板书：在线测试 = 生产环境 + 真实用户 + 真实数据 → 采集指标 → 决策（放量／回滚）｜三道安全阀：灰度放量 → 功能开关 → 止损回滚，另加对比组

提问：我们把新版本放给 5% 的用户做在线测试，结果错误率比对照组高了。这时候第一件事是继续观察还是立刻回滚？依据是什么？

预设回答：①"立刻回滚"——止损优先，正确；②"再观察一会儿看是不是偶然"——要区分场景：如果影响

> 的是关键业务路径、影响面还在扩大，先回滚再复盘；如果只是轻微波动且有完善监控，可以缩短观察窗，但必须设死线；③"回滚了就说明这个功能不能上"——纠正：回滚只是止血，问题定位修好之后可以再灰度。

> **易错点 / 考点：**①把在线测试等同于"不做测试直接上线"——恰恰相反，它需要更完善的监控和更清晰的判定基线；②忽略对比组，把大盘波动误判成版本质量问题；③只讲收益不讲风险。考点多为案例判断："灰度发布加埋点监控"属于哪种测试？答：被动测试／在线测试。记忆口诀："先小后大、可关可回"。

> **过渡：**主动与被动，说的是在实验室还是在线上的差别。接下来换个维度：我们看不看得见程序的内部结构？这就是 2.5 节——黑盒与白盒。

【2.56】2.5 黑盒测试方法和白盒测试（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 2.5 节，黑盒测试方法和白盒测试。这一页的关键词有两组，请大家对照着记。

>

> 第一组，黑盒对应的是"基于需求的测试"，也叫"数据驱动测试"。看图上那个示意：左边是客户需求，中间是输入，右边是输出，中间那个盒子是黑的不透明。黑盒测试就是不看程序内部怎么写的，只从需求和界面出发，准备输入数据，看输出对不对。它为什么叫数据驱动？因为驱动你测试的就是那一组组输入数据以及它们对应的期望输出——用例表就是它的核心资产。事件驱动也是同一个意思：你按一个按钮触发一个事件，然后观察系统响应。

>

> 第二组，白盒对应的是"结构化测试"，也叫"逻辑驱动测试"。这里盒子是透明的，你盯着代码里的语句、分支、条件、循环、路径去设计用例，追求的是"覆盖"——语句覆盖、判定覆盖、条件覆盖、路径覆盖。覆盖得够不够，要靠工具统计覆盖率来说话。

>

> 为什么这两种方法都必须有？因为它们能发现的问题不一样。黑盒站在用户角度，能发现"功能没实现、需求理解错了、界面不友好、业务流程走不通"；白盒站在实现角度，能发现"某个分支写错、边界判断漏了、异常没处理、代码里有死代码"。同一段代码，用两种视角去测，能找出完全不同的缺陷。

>

> 还有一个中间地带叫灰盒测试，既知道一点内部结构——比如知道接口协议、知道数据库表结构，又主要从外部去验证。它是现在接口测试和集成测试的主要形态。

>

> 企业里的粗略分工是这样：单元测试多用白盒，因为代码就在手边；系统测试、验收测试多用黑盒，因为测试人员关心的是业务；接口层常用灰盒。

>

> **板书：**黑盒 = 基于需求／数据驱动 → 输入—[黑盒]—输出｜白盒 = 结构化／逻辑驱动 → 语句・分支・条件・路径覆盖｜灰盒 = 知接口不知实现｜选择依据：信息可得性 + 测试阶段 + 测试目标

> **【思政落点 1】**同一段代码，用黑盒视角和白盒视角去测，能发现完全不同的缺陷——这正是科学精神里"多视角求证、不轻易下结论"的训练。要特别提醒学生：发现缺陷是团队的共同资产。发现后如实记录，不因为"这是我写的模块"就轻描淡写，也不因为"不是我负责的"就随手关掉，这是测试从业者最基本的诚信底线。

> **提问：**一个登录功能，输入框什么都不填直接点登录，页面白屏。用黑盒视角你能发现什么？用白盒视角又能发现什么？

> **预设回答：**①"黑盒发现白屏这个现象，白盒发现空指针判断缺失"——很好，这正是标准答案；②"白盒测试是开发人员做的，测试人员不用管"——纠正：白盒是方法，谁做取决于阶段和技能，测试人员完全可以做接口级白盒；③"黑盒就是手工测试"——错，黑盒照样能自动化，关键看有没有利用内部结构信息。

> **易错点 / 考点：**①把黑盒／白盒当成"手工／自动化"的分类——两码事，分类维度是"是否利用程序内部结构信息"；②误以为白盒只测代码、黑盒只测界面；③考试爱考对应关系："数据驱动测试"对应黑盒，"逻辑驱动测试"对应白盒，选择题常把两者对调。判断三步法：一看有没有利用内部结构信息 → 二看用例依据是需求还是代码逻辑 → 三看覆盖度量是需求覆盖还是结构覆盖。

> **过渡：**到这里，2.4 和 2.5 两节的主要概念就讲完了。下一页是一张小结图，我们把它整理成两张对照表。

【2.57】小结（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是小结，一张图，把 2.4 和 2.5 两节的知识点收在一起。我们不重复念标题，直接把它整理成两张对照表，方便大家抄下来复习。

>

> 第一张表，按"谁驱动被测对象"来分。主动测试，测试人员注入输入、发请求、驱动对象、验证响应；特点是有明确预期、能构造边界和异常，缺点是想得到的才测得到。被动测试，测试人员不干预，监控生产环境的运行数据再分析；特点是数据真实、场景完整、零打扰，缺点是没有标准答案、难判定对错。工程上，前者主要在研发测试阶段用，后者主要在线上运行阶段用。

>

> 第二张表，按"看不看得见内部结构"来分。黑盒，基于需求和数据，做输入输出比对，覆盖的是功能与业务路径；白盒，基于代码逻辑，追求语句、分支、条件、路径覆盖；中间还有灰盒，知道接口和数据结构，但主要从外部验证。

>

> 两张表放在一起，其实构成了四个象限。同一个测试活动可以同时落在两个维度上——比如我们写一段自动化接口脚本，发请求去验证返回，同时我们清楚这个接口会读哪张表：那它既是主动测试，又是灰盒测试。再比如线上的灰度监控，是被动测试，同时是黑盒测试，因为你不看代码只看指标。

>

> 给大家一个口诀收尾："主动被动看谁驱动，黑盒白盒看透不透。"再加一个判断方法：拿到任何一个测试活动，先问"输入是谁给的"，再问"我看得见里面吗"，两次追问，分类就清楚了。

>

> 这部分内容是后面所有具体方法的地基。等价类、边界值、判定表、因果图，属于黑盒方法族；语句覆盖、判定覆盖、基本路径测试，属于白盒方法族。今天先把族谱认清楚，后面才不会把方法用错地方。

>

> **板书：**维度一：谁驱动 → 主动（注入输入）／被动（监控数据）｜维度二：是否利用内部结构 → 黑盒（需求・数据驱动）／白盒（逻辑・结构驱动）／灰盒（知接口）｜四象限举例：自动化接口脚本 ＝ 主动＋灰盒；线上灰度监控 ＝ 被动＋黑盒｜口诀：主动被动看谁驱动，黑盒白盒看透不透

提问：请把这个活动归到两个维度上：测试人员用工具向支付接口重复发送 1000 笔请求，观察返回码和金额，

同时他知道这个接口背后会调用第三方通道。这是什么测试?

预设回答：①"主动测试加灰盒测试"——标准答案；②"压力测试"——说的是测试类型，不是分类维度，要引导他分清"分类维度"和"测试类型"；③"黑盒测试"——可以追问：他知道会调用第三方通道这条信息，到底用上了没有？如果没用上，说黑盒也不算错。这正好说明黑盒与灰盒的边界，取决于你实际利用了哪些信息。

易错点 / 考点：①混淆分类维度——"主动／被动"按驱动方式分，"黑盒／白盒"按信息可见性分，"功能／性能"按测试目标分，不能混在一层里答；②误以为被动测试不用设计，实际上监控点、阈值、对比组都是设计出来的；③小结页最容易出简答题："简述主动测试与被动测试的区别""黑盒测试与白盒测试的适用场景"。答题模板：定义 → 特点 → 适用阶段 → 局限。

过渡：分清楚了"怎么测"，下面就要问"分几层测"。2.6 节，软件测试层次。

【2.58】2.6 软件测试层次（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 2.6 节的开篇，标题是软件测试层次，配了一张图。它是一页课时要点页，课件备注里其实写明了这节课的讲法：先讲概念，再详细讲解，然后举例，接着分享工作中的经验、特别是理论和实际有差异的地方，最后给学习建议和实用工具。我们就按这个路子走。

>

先讲概念。测试层次，指的是按被测对象的规模和组装程度，把测试分成若干级别。为什么非要分层？两个理由。

>

第一，缺陷发现得越晚，代价越大。一个在编码阶段改一行代码就能解决的错误，如果拖到系统测试阶段才暴露，可能要重新设计、重新联调，成本成倍往上翻；拖到上线以后，就是真实用户买单。分层就是为了让不同性质的问题，在各自最合适的阶段暴露出来。

>

第二，不同层次的问题性质本来就不一样。函数内部的逻辑错误、模块之间的接口不匹配、整个系统在真实

> 业务下的表现、用户能不能接受，这是四类完全不同的风险，用同一种方式去测是测不出来的。

>

> 这里有一条工程上特别重要的经验，也是理论跟实际最容易分叉的地方：教科书上层次之间是清清楚楚的，实际项目里往往是混着走的。敏捷团队可能今天写一个函数、明天就联调、下周就灰度，四个层次被压缩在几天之内。但注意——压缩的是时间，不是内容。该问的问题一个都不能少：这段逻辑对吗？接口对得上吗？系统在业务场景下行吗？用户认吗？

>

> 所以这一页图，大家要看出的是一个"由内到外、由小到大"的递进：从最小的代码单元，到单元之间的接口，再到单元组成的系统，最后到系统承载的用户业务。下一页我们就把这四层具体展开。

>

> **板书：**测试层次 = 按被测对象规模与组装程度分级｜为什么分层：① 缺陷越晚发现代价越大 ② 不同层次问题性质不同｜讲义六步：概念 → 讲解 → 举例 → 经验 → 建议 → 工具｜关键提醒：时间可压缩，问题不能省

【思政落点 2】分层背后是一种"把大问题拆开、逐层把关"的工程思维，也是工匠精神里"每道工序都有责任人"的体现。可以引导学生反思：很多质量事故并不是没人测，而是四层里某一层被"赶进度"跳过了。守规矩、按流程做事，本身就是质量责任感的体现。

提问：项目工期紧，领导说"单元测试和集成测试先跳过，直接上系统测试，反正最后测通就行"。你同意吗？怎么反驳？

预设回答：①"不同意，越晚发现越贵"——方向对，但要补上具体理由：单元层没测干净的逻辑错误，在系统层会被业务场景掩盖，定位成本极高；②"跳过也能发现所有问题"——典型错误，因为系统测试用黑盒方法，覆盖不到内部结构，很多分支错误永远暴露不出来；③"可以压缩但不能取消"——这是比较成熟的回答，可以据此展开"时间可压缩、内容不可省"。

易错点 / 考点：①把"测试层次"和"测试类型"混为一谈——层次是纵向分级，类型是横向分类；②误以为层次越高越重要——每层目标不同，缺一层就是漏洞；③考点多问"V 模型中每个层次对应哪个开发阶段"，记住对应关系：需求↔验收、设计↔系统与集成、编码↔单元。判断口诀："由小到大看对象，由内到外看信息"。

过渡：好，四层具体是哪四层、每层测什么，下一页那张层次图说得很清楚。

【2.59】软件测试 4 个层次（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是软件测试的四个层次，图上标得很齐，我们一层一层看。

>

> 第一层，单元测试。被测对象是组件、模块、类或者函数，也就是程序里最小的可测单位。目标是确认这个最小单元的功能实现了没有、有没有编码错误。执行主体以开发人员为主。因为信息最全，代码就在手边，所以主要用白盒方法。

>

> 第二层，集成测试。被测对象是单元之间的接口。为什么单独立一层？因为每个单元自己测通了，合在一起不一定能工作。参数顺序写反了、数据格式一个是字符串一个是数字、调用时序不对、异常没有正确传递，这些都属于接口问题，单元测试发现不了，系统测试又太粗、定位不到。

>

> 第三层，系统测试。被测对象是由单元构成的整个系统，关注的是系统功能、安全性、健壮性、效率这些整体质量属性，是在接近真实的环境里，对完整产品做端到端的验证。

>

> 第四层，验收测试。被测对象是系统承载的用户业务，站在用户和业务的角度，确认产品能不能被接受。它问的不是"程序对不对"，而是"用户愿不愿意用、能不能用这个系统把活干完"。

>

> 把这四层串起来看，它是一条从内到外、从小到大的链：组件 → 单元之间的接口 → 由单元构成的系统 → 系统承载的用户业务。对应的信息可见性也在变化：单元层能看到代码，集成层看得到接口协议，系统层基本是黑盒，验收层完全站在用户一侧。

>

> 我特别提醒一个常见误解：不是"层次越高越高级"。这四层是互补关系，不是替代关系。你系统测试做得再漂亮，也测不出一个分支条件写反的逻辑缺陷，因为它藏在黑盒里；反过来，单元测试覆盖率做到百分之百，

> 也不能保证两个模块拼起来就能跑通。

>

> 我们的课程设计就按这条链走：先给功能写单元测试，再集成联调，最后拼成完整的选课系统做系统测试和验收演示。

>

> **板书：**单元（组件／模块／类／函数）→ 集成（单元之间接口）→ 系统（由单元构成的系统）→ 验收（系统承载的用户业务）｜由内到外、由小到大｜互补关系，不是替代关系

> **提问：**我们小组三个人各写一个模块，每个人自己的模块都测过、都能单独跑通。把三个模块拼起来，一登录就报"用户不存在"，登录模块和数据库模块都觉得自己没错。这个问题该在哪个层次发现？为什么单元测试没抓到？

> **预设回答：**①"集成测试，因为这是接口问题"——标准答案；②"在系统测试发现就行"——可以追问定位难度：系统层只能看到"登录失败"，要在三个模块里定位，得靠日志一点点排，成本高得多；③"单元测试没抓到说明单元测试没用"——纠正：单元测试测的是模块内部逻辑，接口契约本来就不属于它的范围，这正是分层存在的意义。

> **易错点 / 考点：**①把集成测试理解为"把单元测试再跑一遍"——错，它的目标是接口；②把验收测试等同于系统测试——区别在视角和主体：系统测试由测试团队在模拟环境做，验收测试由用户在其真实业务场景中确认；③考试常考层次与被测对象的对应连线题，也考"某缺陷属于哪一层"的案例题。判断三步法：一看被测对象是什么（代码／接口／系统／业务）→ 二看谁在做（开发／测试／用户）→ 三看有没有利用内部结构信息。

> **过渡：**四个层次分好了，那每一层到底要完成哪些任务？下一页用一张表把任务列得明明白白。

【2.60】不同测试层次的任务（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是一张表，把四个测试层次各自要完成的任务列出来了，信息量很密，我们一行一行过。

>

> 第一行，单元测试有三项任务：组件的功能、健壮性、效率。功能好理解，就是这段代码该算对的算对了没有。健壮性指的是异常情况下的容错能力——输入为空、输入超长、除数为零、文件打不开，这时候程序是优雅地报错还是直接崩掉。效率则是这个最小单位上的性能，比如一个函数的时间复杂度和资源占用。

>

> 第二行，集成测试的任务只有一项：组件之间的接口。任务少不等于简单，恰恰说明这一层的目标非常聚焦——它不管单个组件的功能，专门盯接口契约：参数个数和类型对不对、数据格式一致不一致、调用顺序有没有依赖、异常能不能正确传递、共享数据会不会打架。

>

> 第三行，系统测试的任务有四项：系统功能、安全性、健壮性、效率。跟单元测试比，多了安全性；而且这里的功能、健壮性、效率都是系统级的——系统功能是端到端的业务流程能不能走通；安全性是权限、注入、越权访问、敏感数据保护；健壮性是高并发、异常流量、部分服务不可用时的表现；效率是响应时间、吞吐量、资源使用。

>

> 第四行，验收测试的任务是：功能及用户界面、安全性、效率、用户的可接受性。注意最后一项——用户的可接受性，这是验收测试独有的，最主观也最终极：功能都能用，但界面乱、流程绕、术语看不懂，用户就是不接受，那这个系统在验收这一环就是不通过。

>

> 竖着看还有一条规律：越往上，测的维度越多、越贴近整体和用户；而"接口"只在集成这一层出现，它是这一层专属的靶心。

>

> 记忆方式：别死背四行，记三个关键词——组件三项、接口一项、系统四项、验收四项加可接受性。考试里最常见的题型，是给你一个缺陷现象，问你它属于哪一层的哪项任务。

>

> **板书：**单元 → 组件功能・健壮性・效率｜集成 → 组件之间的接口｜系统 → 系统功能・安全性・健壮性・效率｜验收 → 功能及用户界面・安全性・效率・用户可接受性｜竖看规律：越往上维度越多、越贴近用户

提问：三个现象，请分别归到层次和任务上。①某函数传入空字符串时抛未捕获异常，导致进程退出；②订

单模块传给库存模块的数量是字符串"3"，库存模块按数字处理，扣减结果错误；③选课系统在 2000 人同时抢课时，响应时间从 0.5 秒涨到 30 秒。

预设回答：①单元测试的健壮性——注意不是功能，因为它没算错，是异常处理缺失；②集成测试的接口——典型的数据格式不一致；③系统测试的效率，也可以说性能测试。如果学生答成"都是 bug"，要追问他：这三个问题分别在哪个阶段最容易发现、由谁负责修，让他体会分层的实际意义。如果学生把③答成"验收测试的效率"，可以让他比较两种答法的差别：谁在测、在什么环境测。

易错点 / 考点：①把"健壮性"和"功能"混起来——判据是：有没有产生错误结果。产生错误结果是功能问题，异常场景下扛不住是健壮性问题；②忘记"用户的可接受性"只在验收层，看到界面问题就答系统测试；③安全性出现在系统和验收两层，容易漏答一层。记忆口诀："组件三、接口一、系统四、验收四加一"。案例题答题结构：先定层次，再定任务项，最后给一句话理由。

过渡：表格里四层都有了，接下来我们把最底层的单元测试单独拎出来细讲，它的做法和分工很有讲究。

【2.61】单元测试（8 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页专门讲单元测试，课件上这段话几乎是定义级的标准表述，我们逐句拆开看。

>

第一句：单元测试针对程序系统中的最小单元——模块或组件进行测试，一般和编码同步进行。有两个关键词。"最小单元"，在实际的面向对象开发里，通常就是一个类或者一个方法；"和编码同步进行"，意思是不要等整个模块写完再补测试，而是写一个函数就测一个函数，边写边测，问题在几分钟内就被发现并修掉。

>

第二句：主要采用白盒测试方法，从程序的内部结构出发设计测试用例，检查程序模块或组件的已实现功能与定义的功能是否一致，以及编码中是否存在错误。请注意，它同时验两件事：一是功能对不对，二是代码本身有没有写错，比如分支条件写反、变量没初始化、循环边界差一。

>

第三句，这条最容易被忽略：通常要编写驱动模块和桩模块。为什么？因为你要测的函数往往不是独立的

——它要调用别人，也要被别人调用。桩模块用来替代"它调用的下层模块"，比如你的函数要查数据库，测试时就用一个假函数直接返回固定数据；驱动模块用来"调用它"，负责传参数、接收返回值、打印结果。有了桩和驱动，一个函数就能脱离整个系统单独跑起来，这才叫单元测试。

>

第四句：单元测试一般由编程人员和测试人员共同完成，以开发人员为主。原因是他们最了解自己的代码，而且刚写完就测，修复成本最低。

>

最后一句是个很硬的数据：单元测试包括代码评审，代码评审可以发现程序 50% 到 70% 代码的缺陷。这个比例来自教材，说明很多缺陷根本不用运行程序，人眼看代码就能发现，这是性价比最高的一道防线。

>

落到课程里：你写一个"根据选课学分判断能否毕业"的函数，用桩挡掉数据库，用驱动喂几组数据再看返回结果，这就是一次完整的单元测试。

>

板书：对象：最小单元（模块／组件）｜时机：与编码同步｜方法：白盒为主｜目标：已实现功能 = 定义功能 + 无编码错误｜工具件：桩模块（替代被调用的下层）+ 驱动模块（调用本单元）｜分工：开发为主、测试参与｜补充：代码评审可发现 50%～70% 的缺陷（教材数据）

【思政落点 3】代码评审能发现 50%～70% 缺陷这个数据，正好用来说明"同行评审不是挑刺，而是互相保命"。要讲清两点职业素养：评审别人的代码时对事不对人，指出问题要给依据；被别人评审时坦然接受、如实修改。反过来，明知有问题却因为怕丢面子而不说，最后承担后果的是用户——这就是我们这门课反复强调的质量诚信。

提问：我要测一个"计算订单总价"的方法，它内部会去数据库查商品单价，还会调用优惠券服务算折扣。为了做单元测试，我需要写哪两个东西？它们分别替代谁？

预设回答：①"桩模块替代数据库和优惠券服务，驱动模块负责调用计算总价的方法"——标准答案；②"直接把数据库连上测就行了"——要追问：数据库里有脏数据、优惠券服务今天正好挂了，你的测试结果还稳定吗？借此讲"单元测试要隔离外部依赖"；③把桩和驱动说反了——这是最常见的错误，纠正口诀："桩在下被调用，驱动在上调用它"。

易错点 / 考点：①桩模块与驱动模块混淆——判断方法是看它在调用链上的位置：被被测单元调用的是桩，调用被测单元的是驱动；②误以为单元测试必须写代码，其实人工代码评审也是单元测试的一部分；③误以为单元测试是测试人员的事——教材明确以开发人员为主。考点形式：名词解释、简答"单元测试为什么需要桩和驱动"、判断"单元测试一定要在全系统集成之后做"（错）。口诀："同步做、白盒为主、桩下驱动上、开发挑大梁"。

过渡：光说概念还是抽象，下一页课件给了示例图，我们看着图把驱动和桩怎么搭，具体走一遍。

【2.62】示例（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是示例，课件放了两张图。我不照着图念，而是带大家把单元测试的完整套路走一遍。你们看图的时候，就对着这三步走：搭架子、设计用例、看结果。

>

第一步，搭架子。看图上的调用关系：最外面是驱动模块，它负责准备输入数据、调用被测函数、接收返回值。被测函数内部如果要访问数据库、要读文件、要调用别的服务，就在那个位置换成桩模块。你要能一眼在图上指出：哪个框是被测单元，哪个框是被调用方替换出来的假货，哪个框是调用方。这是读这类图最核心的能力。

>

第二步，设计用例。单元测试主要用白盒方法，所以用例不是随便挑几个数字，而是盯着代码结构去设计：每条语句至少走到一次；每个判断的真假两个方向都要走到；多个条件组合的时候要覆盖关键组合；循环要测零次、一次、多次。用这些用例去覆盖，才能回答"我到底测到了多少"这个问题。

>

第三步，看结果、看覆盖。跑完以后不只看通过没通过，还要看覆盖率报告：哪些分支没被走到。没被走到的分支只有两种可能——要么是测试用例不够，要么是这段代码根本多余、或者永远进不去。两种情况都值得处理。

>

> 两张图放在一起，通常是一张讲被测单元与外部依赖的替换关系，一张讲用例的输入与预期输出。你们读图的时候注意一个细节：图里有"预期输出"这一列，这说明单元测试的判定标准是客观的——输入、预期输出、实际输出三者比对，不靠人主观判断，这也是单元测试能够自动化的前提。

>

> 最后再提醒一条工程上的经验：单元测试的用例应该和被测代码放在一起，跟着版本走，每次提交都跑。写一次只跑一次，那不叫单元测试，那只是一次性的手工验证。

>

> **板书：**读图三步：① 找被测单元 ② 找被替换的依赖（桩）与调用方（驱动）③ 找输入／预期／实际三列｜白盒用例设计：语句 → 分支真假 → 条件组合 → 循环 0／1／多次｜结果判定：通过与覆盖双看｜落地要求：用例随代码入库、每次提交都跑

提问：覆盖率报告显示某个 if 分支从来没被执行过。请你给出两种可能的原因，以及分别该怎么处理。

> **预设回答：**①"用例没覆盖到"——处理是补一条能走进去的输入；②"这段代码进不去，是死代码或者防御性代码"——处理是确认后删除，或者加注释说明；③"覆盖率不重要，能跑通就行"——典型误解，要纠正：覆盖率不是目的，但它能暴露"我根本没测过这里"这个事实，是发现测试盲区的手段。可以追问：覆盖率 100% 是不是就没有缺陷了？引导学生自己说出"覆盖的是结构，不是需求"。

易错点 / 考点：①把覆盖率当成质量结论——覆盖率 100% 只说明结构走到了，不代表需求全实现、边界全对；②设计用例时只按代码结构，忽略规格说明里的边界——白盒也要结合需求；③把示例图里的"预期输出"当摆设，实际工作中最常见的错误就是只看程序跑没跑通、不比对预期值。考点："单元测试用例设计的主要依据是什么"——答：程序内部结构（白盒），并结合模块的功能说明。

> **过渡：**单元自测通过以后，下一步就是把这些单元装到一起——这就是集成测试。

【2.63】集成测试（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲集成测试。课件给的定义是：集成测试，也称联调，在单元测试的基础上，将模块按照设计要求组装起来同时进行测试，主要目标是发现与接口有关的模块之间的问题。现在提倡持续集成测试。

>

> 把这个定义里的三个要点抠出来讲。
>
> 第一，前提是单元测试已经做过。集成测试不是拿一堆没测过的模块来试，而是在单个模块已可用的基础上，专门看"组装"这件事。顺序反了，出问题就分不清是模块内部错还是接口错。

>

> 第二，被测对象是接口，由此决定了集成测试用例长什么样：不深挖模块内部逻辑，而是构造跨模块的数据流和调用序列，检查参数传递、数据格式、字段含义、调用时序、错误码约定、异常传播、全局数据和资源竞争。举几个场景：A 模块传时间戳用秒，B 模块按毫秒解析，差了 1000 倍；订单和库存模块的调用顺序写反，出现超卖；一个模块抛的异常另一个模块接不住，流程静默中断。

>

> 第三，为什么叫"联调"。工程里，集成测试往事故意是开发人员坐在一起，把各自模块接上调通——它天然带一点协同色彩，需要接口契约先谈清楚。

>

> 集成策略顺便记一下：一次性组装，也就是"大爆炸"，风险最大；自顶向下，从主控模块往下接，需要桩；自底向上，从底层往上拼，需要驱动；三明治，两头往中间合。这四种策略的名字和优缺点，是考试常客。

>

> "现在提倡持续集成测试"这句话很关键：以前集成是阶段性的动作，等所有模块写完才拼；现在每完成一个小功能就自动构建、自动跑集成用例，让接口问题在几小时内暴露。

>

> 第一章的火星探测器事故就是接口问题的极端版本：两个团队各自算得都没错，但一个用公制、一个用英制，接口没对齐，最终整机坠毁。它讲的不是"谁写错了代码"，而是"接口没约定清楚，谁都没错，系统却错了"。

>

> **板书：**集成测试 = 联调｜前提：单元测试完成｜目标：与接口有关的模块之间的问题｜检查清单：参数／格式／含义／时序／错误码／异常传播／全局数据／资源｜策略：大爆炸 · 自顶向下（需桩） · 自底向上（需驱动） · 三明治｜趋势：持续集成测试

> **提问：**单元测试全部通过，集成的时候发现下单流程偶发失败，日志显示库存扣减成功了但订单没生成。这最可能是哪类问题？集成测试该怎么设计用例才能抓到它？

> **预设回答：**①"接口时序或者事务问题"——对，可能是下单与扣库存的顺序问题，也可能是事务边界没包住两个操作，还可能是异步调用的超时与重试；②"再跑一遍单元测试"——要指出单元测试测不出跨模块的事务边界，跑一百遍也没用；③"这是偶发问题，先不管"——典型的危险回答，要追问：偶发意味着与条件有关，通常和并发、时序相关，集成测试恰恰要用并发和异常注入的用例去复现它。

> **易错点 / 考点：**①把集成测试与系统测试混同——区别在范围与目标：集成测接口、范围是模块组合；系统测整体、范围是完整产品；②以为集成测试只能等所有模块写完才能做，这与持续集成理念相悖；③策略名称与所需辅助模块混淆，记住口诀："自顶向下要桩，自底向上要驱动"。考点常见：简答"集成测试的主要目标"、案例题"该缺陷属于集成还是系统"。

> **过渡：**定义里最后那半句"现在提倡持续集成测试"，值得单独用一页讲清楚，因为它是现代软件交付的基本功。

【2.64】持续集成、持续测试（7 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲持续集成和持续测试，配了一张流程图。我们先把这个流程顺着图讲一遍：开发人员把代码提交到版本库，集成服务器检测到变更，自动拉取代码、自动构建、自动部署到测试环境、自动跑测试，然后把结果反馈给所有人。整个过程不需要人手工点，几十秒到几分钟就出结果。

>

> 为什么要这么做？先说不这么做会怎样。传统模式下每个人在自己机器上写代码，各自都认为自己是对的，等到集成日那天一起合并，冲突、编译不过、接口对不上，一下子全炸出来，业内管这叫"集成地狱"。最要命的是问题堆在一起，你分不清是谁引起的，排错时间比写代码时间还长。

>

> 持续集成的核心其实不是工具，而是习惯：小步、频繁地提交，每次提交都保证可构建，主干始终可用。这样每次集成的差异都很小，一出问题，一眼就能定位到刚提交的那几行。

>

> 持续测试是持续集成的升级：把测试嵌进这条流水线。提交代码后自动跑单元测试；构建成功后自动跑集成测试和接口测试；部署到测试环境后跑冒烟测试；再往前是自动化回归和部分性能测试。它的价值在于反馈速度——缺陷在提交后几分钟就被发现，而不是等到测试阶段才被发现。这正好呼应我们前面讲的"越晚发现越贵"。

>

> 这里必须讲两个代价，不然就是只讲好处了。第一，自动化用例本身需要维护，接口一变用例就红，用例资产和产品代码一样需要人管。第二，会出现"不稳定用例"，一会儿过一会儿不过，大家慢慢就不看红灯了——一旦团队开始忽略红灯，这条流水线就形同虚设。所以工程上有一条铁律：流水线红了先修，绝不带着红灯往主干部署。

>

> 落到我们课程设计：你们小组可以把代码放到 Git 仓库，约定每次提交前必须自己先跑通测试，提交后互相检查。这就是最小规模的持续集成。

>

> **板书：**提交 → 自动拉取 → 自动构建 → 自动部署 → 自动测试（单元／集成／接口／冒烟／回归）→ 反馈｜核心习惯：小步提交、主干常绿、失败即停｜两大代价：用例维护成本、不稳定用例｜铁律：红灯不放过

【思政落点 4】持续集成这条流水线最考验的其实是诚信和责任感——测试红了却直接合入、失败用例被注释掉、报喜不报忧地宣布"构建通过"，这些做法短期看像是"效率"，长期看是给用户埋雷。要引导学生建立"红灯就是命令，谁打破谁负责修"的职业习惯，这正是质量文化落地的具体表现。

提问：小组里有个同学说："只要我本地测试通过，就可以提交了。"这句话有没有问题？

> **预设回答：**①"有问题，本地环境和集成环境不一样"——对，还要补一句：持续集成要求提交后整条流水线是绿的，个人通过不等于整体通过；②"有问题，应该先拉最新代码再测"——这是很成熟的回答，可以顺势讲代码同步与冲突；③"没问题啊"——典型错误，追问：如果你的改动让别人的模块挂了，而你没跑别人的测试，这个问题该由谁来发现？引出"主干常绿是团队共同责任"。

易错点 / 考点：①把持续集成等同于"装个构建工具跑一下"——它首先是开发习惯和纪律；②把持续测试等同于"测试自动化"——自动化是用例的执行方式，持续测试强调的是嵌入流程、每次变更都跑、快速反馈；③忽略不稳定用例的危害。考点多为简答："持续集成的核心实践有哪些""为什么说持续测试能降低质量成本"。

> **过渡：**讲完了理念，我们看一个真实项目的例子——下一页的示例。

【2.65】示例（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页又是示例，配了一张图，课件备注给了一个具体的项目：DotNetNuke，一套非常优秀的基于 ASP.NET 的开源门户网站程序，网址是 www.dotnetnuke.com，我们可以课后点开去看。

>

这个示例为什么选一个开源项目？因为开源项目的质量过程是透明的，你能看到很多平时在公司里看不到的东西。你去观察这样一类项目，我建议盯四个地方。

>

第一，看它的代码仓库怎么组织。有没有专门的测试工程目录，测试代码和产品代码是不是放在一起、一起提交。如果一个开源项目的测试代码和产品代码同步演进，说明它把测试当资产；如果测试目录很久没更新，那它的质量基本只能靠用户报 bug。

>

第二，看它的持续集成状态。这类项目通常会在仓库首页挂一个构建状态的标识，每次提交都自动构建、自动跑测试。绿色代表主干可用，红色代表有人刚把主干弄坏了——这就是我们上一页讲的"主干常绿"。

>

第三，看它的缺陷跟踪。提一个 bug 要求写清楚：复现步骤、期望结果、实际结果、环境版本。修完之后要求补一条回归用例，防止同一个问题再次出现。这个"缺陷—用例"的闭环，是质量持续改善的关键机制。

>

第四，看它的代码提交评审。合并请求要有别人审过、要有测试跟着进。这就是我们前面讲的"代码评审能发现 50% 到 70% 缺陷"，在工程里的落地方式。

>

这四点合起来，其实就是开源社区用流程和自动化，把一个没有统一管理者的大型项目维持住的秘诀。大家做课程设计的时候完全可以照这个套路来：建仓库、写测试、把测试和代码一起提交、用缺陷清单跟踪问题、

> 让组内同学互相评审。你们这个学期要交付的不只是一个能跑的系统，还有一套能说明"你是怎么保证它是对的"的过程材料。

>

> **板书：**示例：DotNetNuke（基于 ASP.NET 的开源门户网站，www.dotnetnuke.com）｜观察四点：① 测试代码是否随产品代码一起提交 ② CI 构建状态是否常绿 ③ 缺陷跟踪是否闭环（复现 → 修复 → 回归用例）④ 合并是否经过评审并带测试｜迁移到课程设计：仓库 + 测试 + 缺陷清单 + 组内互评

> **提问：**你到开源项目仓库里一看，发现它的测试目录最近三个月一次都没改过，而产品代码这三个月提交了两百多次。这说明什么？

> **预设回答：**①"测试没跟上，质量只能靠用户报 bug"——标准答案；②"可能这个项目不需要测试"——要纠正：功能在持续变化，没有测试跟着变，等于回归验证缺位，老功能被改坏的几率很高；③"提交次数多说明项目活跃、质量好"——引导他区分"活跃度"和"质量保障强度"这两个不同的指标。

> **易错点 / 考点：**①把"示例"当故事听，不提炼可迁移的做法——考试和答辩里，能不能把案例讲成"做法 + 效果 + 启示"才是得分点；②误以为持续集成、代码评审是大公司的奢侈品——开源项目用免费工具就能做；③混淆"缺陷多"和"质量差"：缺陷发现得多、跟踪得规范，反而是质量过程健康的信号，真正危险的是没人报缺陷。考点：案例题常要求"结合开源项目的做法，说明如何在小组项目中落地持续集成与测试"。

> **过渡：**到这里，第 2 章关于测试分类、测试层次与黑盒白盒方法的主要内容就讲完了。下一片我们继续往下走，进入测试过程与更多具体方法的内容。

【2.66】系统功能测试（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 前面几次课，我们一路走过了单元测试、集成测试，现在把镜头拉远，站到用户那一边。系统测试是软件测试四个层次里的第三层，它上面承着集成测试，下面接着验收测试。这一页讲的是系统测试里的第一块内容——系统功能测试。

>

> 先给定义：系统功能测试，一般是在完成集成测试之后进行的，它依据的是产品功能说明书，针对产品所实现的功能，从用户的角度来做功能验证，确认每一个功能是不是都能正常使用。

>

> 这段话里有三个词要抠住。第一个是"集成测试之后"——顺序不能乱。第二个是"产品功能说明书"——测试依据是它，不是代码，也不是开发同事口头说的那句"这块没问题"。第三个最关键，是"从用户角度"。同一个功能，开发人员想的是"我是怎么实现的"，用户想的是"我要拿它办成什么事"。

>

> 为什么一定要强调用户角度？因为集成测试查的是模块之间的接口通不通，它关心的是"零件装得上装不上"；系统功能测试关心的是"这台机器能不能干活"。举个生活里的例子：一辆车在总装线上，每个零件都装好了、螺丝都拧紧了，这叫集成没问题；可这车到底能不能开、空调冷不冷、刹车灵不灵，得把它开到路上去跑一跑——这就是系统功能测试。

>

> 再看软件行业的实例。一个电商网站的购物车模块，集成测试关心的是购物车和库存、订单模块之间的接口参数对不对；系统功能测试关心的是一个真实用户能不能把商品放进购物车、改数量、用优惠券、下单、再取消订单，每一步的结果是不是符合功能说明书的描述。

>

> 还有一点必须讲清楚：这一层用的是黑盒方法。你不去管内部怎么实现，只通过界面和接口去驱动它，看输出对不对。这也意味着，功能说明书要是写得不清不楚，这一层测试就注定是糊的——所以我们在测试需求分析里反复强调"可测试性"。
>
> **板书：**系统功能测试 → 位置：完成集成测试之后（系统测试层）→ 依据：产品功能说明书 → 视角：用户角度 → 目标：每个功能都能正常使用 → 方法：黑盒。

提问：购物车"删除商品"功能，集成测试已经通过，系统功能测试还要测哪些点？

> **预设回答：**第一种回答："点删除，商品没了，就算通过。"这是典型错误——只测正常路径，还默认"接口通了功能就没问题"。第二种回答："还要看数量、总价有没有刷新，优惠券要不要重算，库存有没有释放，删除后能不能恢复。"这是正确方向。第三种回答："这些是单元测试该管的。"说明层次没分清。教师点评：先肯定第二种，再带大家把"接口通"和"功能对"这两个概念分开；然后追问库存、优惠券的联动为什么要放到系统层来看——因为它跨了多个模块，只有在系统这一层才观察得到。

易错点 / 考点：最容易和集成测试混淆。判断三步法：一看依据——功能说明书还是设计/接口约定；二看视

角——用户视角还是模块之间；三看对象——每个功能可用与否还是接口问题。考试常以选择题问"系统功能测试在哪个阶段之后进行"，或给一个场景让你判断属于哪一测试层次。口诀：功能说明为依据，用户视角验功能，集成之后才登场。

过渡：功能都正常了，用户就满意了吗？如果点一下要等十秒、一千人同时用就崩了，用户照样跑掉。下一页我们看系统测试的另一半——系统非功能性测试。

【2.67】系统非功能性测试（6 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

上一页我们说功能测试回答的是"能不能用"，这一页要回答的是"好不好用、扛不扛得住"。

>

先看定义：系统非功能性测试，是把软件放在整个计算机环境下——注意这个"整个计算机环境"包括软硬件平台、某些支持软件和数据等——在实际运行环境下验证系统的非功能性。

>

这句话里有两个地方要圈出来。第一个是"实际运行环境"。功能测试可以在你的开发机上跑，非功能测试不行。为什么？因为性能、稳定性这类特性，跟机器的 CPU、内存、磁盘、网络、数据库版本，甚至同一台机器上还跑了什么别的东西都有关系。你在开发机上测出来响应五十毫秒，用户那台装了三个杀毒软件的老电脑上可能是五秒。

>

第二个是"软硬件平台、支持软件和数据"。这几样凑在一起才叫环境，少一样，测出来的结论就不算数。

>

那它包含哪些内容？课件列了四项，后面还跟着省略号：性能测试、安全性测试、稳定性测试、兼容性测试。省略号意味着不止这些，还有易用性、可靠性、可维护性等等，也就是我们前面讲 ISO/IEC 25010 产品质量模型时提到的那些使用质量特性。

>

为什么非功能测试越来越重要？因为今天软件之间在功能上往往拉不开差距——你有的按钮，别人也有。真正让用户流失的是：双十一零点支付一直转圈、上传的照片被人拖走、系统连着跑三天就内存泄漏重启。非

> 功能缺陷的代价，往往比功能缺陷大得多，因为它一坏就是全局性的，而且很难靠改几行代码补回来——它本质上是架构问题。

>

> 打个生活里的比方：一家餐厅，菜品齐全、每道菜都做得对，这是功能合格；可要是这家店只有十个座位，中午来一百个人就排到马路上，空调还坏了，你下次还去吗？
>
> **板书：**系统非功能性测试 → 环境＝整个计算机环境（软硬件平台＋支持软件＋数据）→ 前提＝实际运行环境 → 内容：性能 / 安全 / 稳定 / 兼容 / ……（省略号＝ISO/IEC 25010 其他质量特性）→ 特征：多为架构层面问题，代价全局化。

提问：同一款 App，在开发同事的高配手机上测响应很快，能不能据此判断"性能没问题"？

> **预设回答：**第一种："能啊，实测就是快。"这是错误回答——忽略了环境代表性和负载条件。第二种："不能，要约定目标机型、网络条件、并发量和数据量。"这是正确回答。第三种："性能就是没有卡顿。"理解过于片面。教师点评：由第二种引出"环境＋负载＋指标"三要素；然后强调，非功能测试如果不先把这三样定义清楚，结论就是不可复现、不可比较的，写进报告也没人信。

易错点 / 考点：常见混淆是把功能与非功能混为一谈，以及把"稳定性"和"可靠性"随手混用；最容易漏掉的是"实际运行环境"这个前提条件。考试常考选择题"下列哪项不属于非功能性测试"，或案例题要求为某系统列出非功能测试项。判断三步法：能不能用→功能；好不好用、扛不扛得住→非功能；换台机器、换个负载结论就变→非功能测试必须写清环境与指标。

> **过渡：**系统测试把功能和非功能都验完了，最后谁说了算？是掏钱的用户。下一页，我们讲交付前的最后一道闸门——验收测试。

【2.68】验收测试（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

验收测试是软件测试四个层次里的最后一层，也是交付前的最后一道闸门。

>

> 它的目的说白了就一句话：向未来的用户表明，系统能够像预定要求那样工作。注意"表明"这两个字——验收测试带有很强的证明色彩，你要拿出证据告诉用户：你要的功能都在，性能也和你的合理期待一致。

>

> 定义里还有两个词要抠。第一个是"未来的用户"，也就是验收测试站在真正掏钱、真正要用这套系统的人这一边。第二个是"合理期待"。为什么不说"达到需求规格说明书"就够了？因为用户不会拿着你的说明书逐条比对，他心里的标准是——我上次用的系统是这样，我同事说那个功能应该这样，宣传册上写的是这样。这些没说出口的东西，就是隐含需求。而验收测试要管的，恰恰是这些"没写下来、但用户认为理所当然"的部分。

>

> 验收测试还有一项很具体、也最容易被忽略的工作——安装测试，也叫部署验证。它是按照软件产品的安装手册或相应文档，在一个和用户使用该产品完全一样的环境中，或者相当于用户使用环境中，一步一步地做安装操作测试。为什么强调"一步一步"？因为用户就是照着手册一步步做的。手册里少写一句"先停服务"，用户就能给你装崩。所以安装测试不只测软件，也在测文档。

>

> 打个比方：你网购了一套衣柜，图纸上写"先装底板再装侧板"，你照着做，结果发现孔位对不上——这不是你笨，是这套图纸和产品没有经过"安装测试"。软件也一样。
>
> **板书：**验收测试 → 目的：向未来的用户表明系统能像预定要求那样工作 → 验证：功能＋性能＝用户的合理期待（含隐含需求）→ 安装测试（部署验证）：按安装手册/文档，在与用户环境完全一样或相当的环境中，一步一步做安装操作 → 一句话：一层管"用户认可"，一层管"装得上"。

提问：需求规格说明书上写的功能全部测通过了，验收测试是不是就一定能过？

> **预设回答：**第一种："能，都按需求验过了。"这是典型错误——把"符合规格"当成了"用户满意"。第二种："不一定，还要看隐含需求、真实数据与业务量、实际环境和用户的使用习惯。"这是正确方向。第三种："验收测试是我们测试组自己跑一遍。"这是把主体搞错了，验收的主体是用户或客户。教师点评：借第二种回答讲清验证与确认的区别——验证是"把东西做对了"（符合规格），确认是"做了对的东西"（满足用户）；验收测试偏向确认，所以规格之外的合理期待同样要核。

> **【思政落点 1】**讲什么：定义里那句"如同用户所合理期待的那样"。怎么联系知识点：验收测试要管隐含需求，说明质量责任不只在纸面上那几行条款。落到什么价值观：诚实交付、对用户负责，不做"说明书上没写就不算我的事"的推诿——把用户没想到的风险替用户想到，这才是工程人员的职业良心。

易错点 / 考点：第一，验收测试与系统测试混淆——判断点在主体与视角：验收由用户/客户主导、看"能否接

受"，系统测试由测试团队主导、看"系统是否符合规格"。第二，漏掉安装测试其实属于验收测试。第三，把"合理期待"只理解成需求文档。考试高频点：α、β 测试都属于验收测试的形式（下一页面）。案例题常问"上线验收前应做哪些准备"。

> **过渡：**用户到现场来验，是一回事；用户不在现场，甚至成千上万个用户分散在全国各地，又该怎么验？下一页我们看 α 测试、β 测试，以及一个更有争议的做法——在线测试。

【2.69】α、β 测试 &在线测试（6 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页有三个概念：α 测试、β 测试，还有在线测试。课件用的是英文原文，我们一条一条来读。

>

> 第一段，α 测试。原文是：Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site，后面跟了一句，Is a form of internal acceptance testing。翻过来就是——α 测试是由潜在用户、客户，或者一支独立的测试团队，在开发方的场所进行的模拟的或实际的操作性测试；它属于一种内部验收测试形式。三个要点：谁来做（潜在用户、客户或独立测试团队）、在哪做（开发方场所）、什么性质（内部验收测试）；而且它既可以是模拟的，也可以是实际操作的。

>

> 第二段，β 测试。原文说：Beta testing comes after α testing，Versions of the software, known as beta versions, are released to a limited users outside of the programming team。翻过来——β 测试在 α 测试之后进行；把软件的某个版本，也就是我们说的 β 版本，发布给编程团队之外的有限用户使用。三个关键词：α 之后、有限用户、编程团队之外。说白了，α 是在自家院子里试车，β 是把车交给一批志愿车主开上路，让真实世界帮你挑毛病。

>

> 第三个概念是 Testing in production，在线测试，也常译成生产环境测试。它的思路是：软件已经在真实环境里跑了，那就直接在真实环境里测试、观测、比对，而不是另搭一套测试环境。为什么会出现这种做法？因为再像的测试环境也不是生产环境——真实的用户量、真实的数据分布、真实的地域网络差异，你复制不出来。

>

> 课件还给了两个延伸阅读的链接：一个是微软 MSDN 上 seliot 的博客，另一个是 thetestingplanet 在 2011 年 11 月的那篇《The Future of Software Testing Part One: Testing in Production》。从这两条链接能看出，这个做法讨论得很早，不是新潮概念。

>

> 但要注意分寸：在线测试能做，前提是有兜底——灰度发布、功能开关、实时监控、随时能回滚。这几样缺了，在线测试就是拿用户当小白鼠。
>
> **板书：**α 测试：开发方场所 / 潜在用户・客户・独立测试团队 / 模拟或实际操作 / ＝内部验收测试。β 测试：在 α 之后 / β 版本 / 编程团队之外的有限用户 / ＝外部真实环境试运行。在线测试（Testing in production）：在真实生产环境测试与观测；配套：灰度、开关、监控、回滚。延伸阅读：MSDN seliot 博客；thetestingplanet 2011.11《The Future of Software Testing Part One: Testing in Production》。

提问：一个团队想在正式上线后做在线测试，于是直接把新版本全量发布给所有用户使用，行不行？

> **预设回答：**第一种："行，真实用户测出来的问题最真实。"这是错误回答——只看到真实性，忽略了风险与兜底手段。第二种："不行，要先小流量灰度、加功能开关、布好监控和回滚预案，用少数用户先验证。"这是正确回答。第三种："在线测试就是让用户帮忙报 bug。"理解片面。教师点评：由第二种讲"真实"与"可控"这对矛盾——在线测试的价值在真实性，底线在可控性，两者靠工程手段同时满足；再提醒一句：没有回滚能力的全量发布，不叫测试，叫赌博。

易错点 / 考点：第一，α 与 β 的顺序和场所最容易搞反，口诀是"α 在自家、β 在门外，α 先 β 后"。第二，把 α、β 排除在验收测试之外——课件明确 α 属于内部验收测试形式，β 是交给外部有限用户。第三，把在线测试理解成"上线了就不用测试"。考试常考选择题"下列关于 β 测试的说法正确的是"，干扰项多在场所、参与人、先后顺序上做文章。

> **过渡：**到这里，测试层次这条线就走完了：单元、集成、系统、验收。接下来我们要换一个视角——不看被测对象，而是看测试这份工作本身：测试人员一年到头到底在忙什么？这就是 2.7 软件测试工作范畴。

【2.70】2.7 软件测试工作范畴（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 从这一页开始，我们进入 2.7 节——软件测试工作范畴。前面 2.1 到 2.6，讲的是概念、方式、层次；从这一

节起，我们把视角从"测什么、怎么测"转到"测试这份工作到底由哪些环节构成"。

>

课件在这一节的开头放了一张结构图，另外配了一段讲义编写说明。这段说明本身很值得念一遍，因为它规定了接下来每一个知识点要从六个角度展开。第一是概念：这个概念是什么，在工作中什么场景会用到。第二是讲解：要胜任这个能力，需要掌握哪些知识和技术。第三是举例：用一个案例或 demo 帮大家理解，并且知道怎么用。第四是分享经验：特别是理论和实际有差异的地方。第五是学习建议：在相关学习和工作场景中怎么用。第六是实用工具：这个知识点对应的工具和模板。

>

为什么要按这六步走？因为测试是一个工程实践性极强的岗位。你光记住"测试计划包含七个要素"没有用，你得知道在你们公司这份计划是给谁看的、评审会上谁最可能挑刺、写完之后靠什么跟踪。所以后面每一节，我都会尽量把这六件事补齐。

>

那么测试工作一共有哪些环节？按教材的划分，软件测试的工作范畴包括六块：测试需求分析、测试策略制定、测试计划、测试设计、测试执行，以及测试结果和过程评估。请大家先记在心里，这是一个闭环——先想清楚测什么，再定怎么测，再排计划，再做设计，然后执行，最后评估并反馈回需求。

>

先看这张全局结构图，下一页我们把测试自身的工作流程展开。

板书：2.7 软件测试工作范畴 → 六环节闭环：需求分析 → 策略 → 计划 → 设计 → 执行 → 结果与过程评估 →（反馈回需求）。讲义六步法：概念 → 讲解 → 举例 → 分享经验 → 学习建议 → 实用工具。

提问：这六个环节里，你觉得哪一个最容易被团队跳过或者敷衍？为什么？

预设回答：第一种："测试执行，因为最累。"这说明只把执行当成产出环节，忽略了前面的设计决定执行价值。第二种："测试需求分析和策略，赶进度的时候容易被跳过，直接开始写用例。"这是正确方向。第三种："评估，项目一结束大家就散了。"也有道理。教师点评：肯定第二种，指出需求分析与策略是"地基"，跳过它们，用例写得再多也可能是无效覆盖——地基没打，砖砌得再多也白搭；同时表扬第三种观察，说明评估环节确实常被形式化。

易错点 / 考点：六环节的名称与顺序（常以多选题或排序题出现）；最容易混的是"测试策略"和"测试计

划"——策略回答"怎么测"，计划回答"谁、在什么时候、用什么资源做什么"；另一个常见遗漏是只记到"执行"，把"评估"丢了。口诀：需、策、计、设、执、评。

> **过渡：**知道了有哪六块，还要看它们是怎么串起来的、各自的产出物是什么——下一页是软件测试自身的工作流程。

【2.71】软件测试自身工作流程（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是一张流程图，页面上没有文字说明，所以我要带着大家把这张图读出来。

>

> 图的左边是入口，起点是测试需求——从需求文档、设计文档、产品功能说明里提炼出"要测什么"。往右走第一站是测试策略：决定手工还是自动化、黑盒还是白盒、先测哪块后测哪块。第二站是测试计划：把范围、测试项与优先级、风险、进度、资源、跟踪控制机制定下来，落到文档和排期上。第三站是测试设计：先做总体设计，把测试方案和测试结构定下来；再做详细设计，把一条条测试用例写出来。再往右是测试执行：手工的、自动化的，按用例跑，也允许探索式地去试。最后一站是测试结果与过程评估：看覆盖率够不够、缺陷是不是在收敛、当前版本质量如何，同时把计划里说的活动和实际做的事对一对。

>

> 评估的结果要往回反馈：缺陷多的模块，下一轮要加测；覆盖不到的路径，要回到设计补用例；需求变了，要回到需求分析重新界定范围。所谓"测试自身的工作流程"，重点在"自身"两个字——它和开发流程是两条并行的线，但两者要互相咬合。

>

> 读这张图，请大家盯住三样东西：起点在哪里、每一站的产出物是什么、哪些箭头是回头的。
>
> **板书：**流程主线：测试需求 → 测试策略 → 测试计划 → 测试设计（总体/详细）→ 测试执行（手工/自动化/探索）→ 测试结果与过程评估 →↩ 反馈回需求与设计。读图三问：起点在哪？每站产出物？哪些箭头回头？

提问：照着这张图，如果你手上只有一份需求文档，下一步应该产出什么？再下一步呢？

> **预设回答：**第一种："直接开始写测试用例。"这是典型错误——跳过了需求分析、策略和计划。第二种："先做测试需求分析，明确范围与优先级；再定策略、排计划，然后才设计用例。"这是正确回答。第三

种："先搭测试环境。"这是把手段当成了流程起点。教师点评：重点强调"产出物"意识——每一步都要有可评审、可交接的产物，没有产出物的流程等于没走；然后追问一句：测试用例是这张图里哪一站的产品？把答案落在"测试详细设计"上。

易错点 / 考点：流程顺序与各阶段产出物的对应关系（考试爱考"测试用例属于哪个阶段的产物"→测试详细设计；"测试计划属于哪一站"→第三站）；另一个易错点是把测试环境搭建当成流程起点。判断口诀：先想测什么（需求）→ 再想怎么测（策略）→ 再排怎么做（计划）→ 再写下来（设计）→ 再动手（执行）→ 最后回头看看（评估）。

过渡：流程的第一站，也是最容易被敷衍的一站，就是测试需求分析——下一页我们把它讲透。

【2.72】测试需求分析（5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

测试需求分析，是整个测试工作的第一站。它要回答的是最原始的那个问题：到底测什么。

>

课件给了三件事。第一件，明确测试范围——哪些功能点要测试、哪些功能点不需要测试。注意后半句，"哪些不需要测"同样重要。为什么？因为测试资源永远是有限的，把范围写清楚，既是对测试负责，也是对项目和用户负责：白纸黑字说明这个版本不测这个模块，出了问题是大家都知道的风险，而不是测试漏了。

>

第二件，知道哪些测试目标优先级高、哪些优先级低。范围定了，还要排序。核心流程、用户高频使用的功能、涉及资金和隐私的部分，优先级最高；边缘的、内部的管理功能，可以往后排。为什么排序这么重要？因为现实是，测试时间永远会被压缩。会压缩到什么程度？压缩到只能测最重要的那几条。所以优先级不是锦上添花，它是你在最后关头决定"先测什么"的依据。

>

第三件，要完成哪些相应的测试任务，才能确保目标的实现。这句话是从目标倒推任务：要实现"核心支付流程可用"这个目标，需要哪些任务？准备测试数据、搭建环境、设计多少条用例、执行几轮回归、分别由谁来做？这些想清楚了，后面的计划才有东西可排。

>

> 我给大家一个判断标准：测试需求分析做得好不好，就看它能不能回答三个问句——测什么？不测什么？先测什么？

>

> 打个生活里的比方：装修房子的时候，设计师上来不会先问你要买什么牌子的油漆，他会先问几口人住、要不要书房、预算多少、什么时候入住。范围、优先级、条件先对齐，后面才不会返工。测试需求分析就是这个"先对齐"的环节。

>

> 做这一步时还要同步考虑一个词——可测试性。需求里写"系统应具有良好的用户体验"，这条你没法测。遇到这种情况，正确做法是回到需求评审把二义性问清楚，而不是假装它不存在。
>
> **板书：**测试需求分析＝测什么 → ①范围：要测的功能点 / 不测的功能点 → ②优先级：高 / 低 → ③从目标倒推测试任务 → 检验三问：测什么？不测什么？先测什么？→ 同步识别"可测试性"（含糊需求回评审澄清）。

提问：某条需求写着"系统应具有良好的用户体验"，你怎么把它变成可测试的测试需求？

> **预设回答：**第一种："照抄进测试需求，列成一条测试项。"这是典型错误——无法判定通过与否，等于没测。第二种："追问具体场景与指标，比如首页首屏加载时间、完成关键操作的步数、错误提示是否看得懂。"这是正确方向。第三种："直接删掉这条需求。"做法太粗暴，应当回评审确认，而不是单方面删除。教师点评：把"二义性"当成一类缺陷来处理，说明测试需求分析的实质是把需求翻译成可验证的语句；同时强调必须找需求方确认，测试人员不能自己给需求加戏。

易错点 / 考点：第一，把"测试需求"等同于"软件需求"——测试需求是测试视角的转化结果。第二，只列要测的，不列不测的。第三，漏掉优先级。考试常见题型：给出需求文档片段，要求写出测试范围与优先级，或判断"该需求是否可测"。口诀：范围清、优先级明、任务能落地。

> **过渡：**知道自己要测什么、先测什么之后，下一个问题就是怎么测才最划算——这就是软件测试策略。

【2.73】软件测试策略（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

测试策略，是这一章里最能体现"测试是一门工程"的一页。它要做的决定，是"怎么测最划算"。

>

> 课件把要考虑的因素分成三类。第一类，测试方式。这里面有几组权衡：手工方式还是自动化方式，静态方式还是动态方式；用探索式测试还是基于脚本的测试；是自己团队测，还是众测、外包。第二类，测试方法。是黑盒方法还是白盒方法，是基于数据流还是基于控制流，是做完全组合测试还是做组合优化测试。第三类，测试过程。先测什么、后测什么，测试阶段怎么划分。

>

> 我们挑几组讲透。第一组，手工还是自动化。大家容易有个误解，觉得自动化高级、手工落后。其实不是。自动化适合那些稳定、重复、回归性的工作——每次发版都要跑一遍的登录、下单、支付主流程；手工适合一次性的、需要人的判断和感受的工作——界面好不好看、提示语读起来别扭不别扭、新手会不会迷路。拿自动化去测"这句话说得顺不顺"，那是拿锤子拧螺丝。

>

> 第二组，完全组合还是组合优化。假设有十个参数，每个参数三个取值，完全组合就是三的十次方，将近六万种组合，这还没算操作顺序。你根本跑不完。所以工程上常用正交试验、成对组合这类组合优化方法，用很少的用例覆盖大部分缺陷。这里的关键认知是：测试不可能穷尽，策略就是在"测不完的世界"里做取舍。

>

> 第三组，自己团队测还是众测、外包。自己团队了解产品，但视角容易固化、存在盲区；众测人多面广、贴近真实用户，但结果质量参差、保密风险要评估；外包省人力，但沟通成本和知识沉淀是代价。

>

> 课件最后给了一句目的性很强的话：制定或选择更合适、更有效的测试方式、方法和技术，目的是以最低的时间或人力成本，达到最大程度地揭示产品的质量风险，尽快完成测试。请注意，这句话里有三个诉求同时存在——成本要低、风险要露出来、时间要快。它们在互相打架，策略就是在打架的三者之间找平衡点。

>

> 所以请记住：策略不是"选最先进的技术"，策略是"为这个项目、这个阶段、这批人做出最合适的取舍"。
>
> **板书：**测试策略 → 三类因素：①测试方式（手工↔自动化 / 静态↔动态 / 探索式↔基于脚本 / 自测↔众测・外包）；②测试方法（黑盒↔白盒 / 数据流↔控制流 / 完全组合↔组合优化）；③测试过程（先测什么后测什么 / 测试阶段划分）。目标：最低时间人力成本 + 最大程度揭示质量风险 + 尽快完成（三角平衡）。

> **提问：**一个每月发版四次的产品，主流程回归每次都靠两个人手工跑两天，你觉得策略该怎么改？为什么不是"全都自动化"？

> **预设回答：**第一种："全都自动化，手工不要了。"这是错误回答——忽略了一次性测试、体验类测试，也没算脚本的开发和维护成本。第二种："把稳定的回归主流程自动化，探索式和体验类仍然手工，同时评估脚本的开发与维护成本。"这是正确方向。第三种："直接加人扛。"这是在回避问题。教师点评：由第二种引出"自动化成本＝开发＋维护＋失效后修复"，强调自动化是有投资回报门槛的；选择标准就三句——稳定、高频、结果可判定。

> **易错点 / 考点：**第一，把测试策略和测试计划混为一谈——策略回答"怎么测"，计划回答"谁、何时、用什么资源做"。第二，把"组合优化"理解成"少测几个功能"——它其实是方法层面的覆盖设计。第三，以为自动化一定优于手工。考试常以案例题出现："请为该产品制定测试策略，并说明你选择这种测试方式的理由。"

> **过渡：**策略定了，接下来要把它变成可执行的安排——排期、分人、定资源。这就是测试计划。

【2.74】测试计划（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 测试计划，是把策略落到纸面和排期表上的那一步。课件列了七项内容，我们一项一项看，每一项都问一句：不写会怎样？

>

> 第一，目标和范围。这份计划要达成什么，管到哪儿为止。不写，测试做完了没人知道算不算完成。

>

> 第二，测试项及其优先级。这次要测哪些功能、哪些模块、哪些非功能特性，谁先谁后。不写优先级，一遇到延期就只能慌。

>

> 第三，测试风险识别与分析。这一条最容易被写成空话。我给大家几个常问的风险方向：需求变更频繁、测试环境不稳定、开发提测延迟、人手不足、第三方接口不可用。写风险不是凑数，是要连着写应对措施——比如提测延迟，就要写明每延迟一天，测试时间怎么压缩、优先保哪些测试项。

>

> 第四，指定测试策略。也就是把上一页讲的取舍写进来，作为本次测试的方法依据。

>

> 第五，进度安排。什么时候测哪个阶段，里程碑是什么，什么时候能给出验收结论。进度要跟开发进度对齐，关键节点最好倒排。

>

> 第六，资源配置。人力、环境、设备、工具、测试数据。这里"测试数据"经常被遗漏，可它常常是最大的瓶颈——真实数据拿不到、脱敏规则没定，执行阶段就卡住了。

>

> 第七，跟踪和控制机制。进度怎么跟踪，日报还是看板？谁来汇总？偏差到什么程度要启动调整？测试计划不是写完就归档的文档，它是活的：需求变了要改，进度偏了要调，所以必须有跟踪和控制机制兜住。

>

> 最后强调一点：测试计划不是测试组长一个人的作业。它要跟项目计划对齐，要经过评审，要让开发、产品、运维都知道里面关于他们的承诺。
>
> **板书：**测试计划七要素 → 目标与范围 → 测试项及优先级 → 风险识别与分析（必须带应对措施）→ 指定测试策略 → 进度安排 → 资源配置（人/环境/设备/工具/数据）→ 跟踪与控制机制。口诀：目标范围、项与序、风险、策略、进度、资源、跟踪控。

提问：如果你们的测试计划只能删掉一项，删哪项损失最小？如果只能保留一项，留哪项？

> **预设回答：**第一种："删跟踪控制机制，文档写完就完了。"这是错误回答——计划会因此失去活性，没人跟踪就等于没有计划。第二种："保留目标与范围，因为其他各项都围绕它展开。"这个判断比较合理。第三种："删资源配置，人不够是项目经理的事。"这是错误回答——资源需求不写进计划，测试就没法争取到人。教师点评：强调"目标是纲"，其余六项都是它的支撑；同时点出跟踪控制是计划能落地的关键，也是考试里最常被漏掉的一项。

易错点 / 考点：七项内容与"策略 vs 计划"的边界；常考多选题"以下属于测试计划内容的有"，干扰项多为测试用例细节（那属于测试设计）。另一处易错：风险评估只列风险清单、不写应对措施。判断口诀：目标定方向、范围划边界、项序排先后、风险带对策、策略定方法、进度定时间、资源定投入、跟踪保落地。

> **过渡：**计划里写着"要做哪些测试"，可具体一条一条到底怎么测出来？这就进入测试设计。

【2.75】测试设计（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 测试设计，解决的是"如何测"这个问题。前面测试需求分析回答了"测什么"，策略和计划回答了"用什么方式、什么时间、多少资源测"，到了测试设计，就要把它落成一条条具体的、可执行的测试。

>

> 课件把测试设计分成两层。第一层，测试总体设计。它主要指测试方案的设计和测试结构的设计。测试结构是什么意思？就是把整个测试对象组织成一个层次——测哪几个大块，块与块之间是什么关系，每个块下面再分哪些功能点、哪些测试项，形成一棵树。有了这棵树，你才知道覆盖率该怎么算：叶子节点覆盖了多少。

>

> 测试方案的设计，范围比较大。课件列了五件事：选择测试方法、明确测试策略、设计测试技术路线、选择测试工具、规划测试环境。我把它们串成一句话：用什么方法、走什么技术路线、拿什么工具、在什么环境里、按什么策略去测。测试环境这一项常常被低估——环境规划包括要几套环境、硬件配置、数据库与中间件版本、测试数据从哪来、环境之间怎么隔离和复用。环境不对，前面设计得再好也白搭。

>

> 第二层，测试详细设计，主要指测试用例的设计。这是最接地气的一层：一条用例要有编号、所属模块、前置条件、输入数据、操作步骤、预期结果、优先级，必要时还要关联需求编号。

>

> 这里有个常见的认知误区：很多人以为"测试设计"就是"写用例"。其实写用例只是详细设计，前面还有总体设计。为什么总体设计不能省？因为它决定了覆盖的思路——如果连测试结构都没搭起来，用例就是一堆散装条目，覆盖率没法算，哪一块遗漏了也看不出来。这就是我们课程设计里为什么要求先画测试项结构，再写用例。

>

> 打个比方，测试设计就像装修的施工图：总体设计相当于平面布局图——哪里是客厅、哪里是卧室、水电怎么走；详细设计相当于每个柜子的加工尺寸图。只有尺寸图、没有布局图，装出来的房子是住不了人的。
>
> **板书：**测试设计＝解决"如何测" → ① 总体设计：测试方案设计（选测试方法 / 明确测试策略 / 设计技术路线 / 选测试工具 / 规划测试环境）＋测试结构设计（测试项树形分解→可算覆盖率）；②详细设计：测试用

例设计（编号/模块/前置条件/输入/步骤/预期/优先级/关联需求）。类比：布局图 vs 加工图。

提问：给"用户登录"功能做测试设计，总体设计和详细设计分别要产出什么？

预设回答：第一种："上来就写十条用例。"这是典型错误——跳过了总体设计。第二种："总体设计产出登录相关的测试项结构（正常登录、异常登录、安全、兼容、性能等）、方法选择（等价类、边界值）、环境与数据准备；详细设计产出一条条用例，写清账号密码取值、预期结果和优先级。"这是正确回答。第三种："只画结构不写用例。"这是另一个极端。教师点评：强调两层缺一不可；再联系本课的实践任务"用等价类和边界值为登录功能设计十条用例"——那十条属于详细设计，而它前面的分析属于总体设计。

易错点 / 考点：第一，把"测试设计"等同于"写用例"，漏掉总体设计。第二，总体设计与详细设计各自包含什么（高频选择题）。第三，把测试环境规划当成运维的事而不写进测试方案。口诀：总体定方案与结构，详细出用例；先有结构后有例，覆盖率才算得清。

过渡：详细设计的产物就是测试用例。下一页我们专门说说，为什么测试用例这么值得花时间去写。

【2.76】测试用例（5 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

测试用例，是测试详细设计的产物，也是测试人员手里最实在的东西。课件给了它四层价值，我们一层一层来看。

>

第一层，它是测试人员在测试过程中的重要参考依据。什么依据？执行的时候照着它做：前置条件是什么、输入什么、怎么操作、期望看到什么。有了它，测试就不再依赖某个人的记忆。这一点在多轮测试里尤其重要——第一轮谁测的、第二轮换个人来测，结论必须一致。

>

第二层，它有助于节约测试时间、提高测试效率。这里要澄清一个误解：写用例看起来费时间，怎么反而省时间？因为它把"想"和"做"分开了。设计的时候集中精力想清楚测什么、怎么判断对错；执行的时候就是照做、记录、比对。如果边想边做，一个人一上午可能只推进两三个功能点，而且判断标准飘忽不定。

>

第三层，良好的测试用例会不断被重复使用，使得测试过程事半功倍。这句话的关键词是"重复使用"。一个

> 版本发一次，回归一轮；发十次，就回归十轮。同一批用例被用了十遍，它的开发成本就被摊得很薄。这也是自动化脚本能够落地的前提——没有稳定、描述清晰的用例，自动化无从谈起。

>

> 第四层，测试用例是一个知识积累的过程。一位老测试人员的价值，很大一部分就沉淀在他们积累的用例库里：那些踩过的坑、遇到过的边界、闻所未闻的异常输入，都写在用例里。新人来了，读用例往往比读需求文档学得快。用例库越厚，团队的整体能力就越不依赖个别人。所以用例不仅要写，还要评审、维护、归档、复用。

>

> 一句话总结这一页：用例是测试的"规格说明书＋知识库＋回归资产"。

【思政落点 2】讲什么：用例库是团队一点一点积累起来的知识资产。怎么联系知识点：正因为良好用例会不断被重复使用、成为回归与自动化的基础，它才必须被认真评审、长期维护。落到什么价值观：踏实记录、精益求精的工匠精神——反对"能用就行、随手一测"的应付态度，也不藏私，让个人经验变成团队能力。

> **板书：**测试用例四层价值 → ①执行的参考依据（统一标准、不依赖记忆）→ ②节约时间、提高效率（把"想"与"做"分离）→ ③不断重复使用、事半功倍（回归资产）→ ④知识积累（团队能力沉淀，要评审、可维护、可复用）。

提问：为什么说"没有用例也能测"，但长期看仍然必须写用例？

> **预设回答：**第一种："因为公司要求写文档。"这是表面回答。第二种："为了统一标准、支撑回归复用、沉淀团队知识，还能作为自动化的基础。"这是正确回答。第三种："有了用例就不需要探索式测试了。"这是把两者对立起来，错误。教师点评：借第二种承接探索式测试与基于脚本的测试的关系——临场探索可以没有用例，但要把探索的发现固化成团队资产，就必须回写用例；最后强调"可重复使用"才是用例最大的价值。

易错点 / 考点：第一，把"节约时间"理解成"写用例很快"，它省的是执行与回归的时间。第二，漏掉"知识积累"这一层（简答题常考这个要点）。第三，用例要素记不全。判断口诀：依据—效率—复用—沉淀。考试常考：测试用例的作用（简答/多选）；以及判断题"测试用例是测试详细设计的产物"。

> **过渡：**用例写好了，接下来就是照着它去跑——测试执行。这里面有两条路：手工执行和自动化执行。

【2.77】测试执行（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

测试执行，终于到了动手的环节。课件把它分成两种：手工执行和自动化执行。

>

先看手工执行。定义是：基于详细设计的测试用例来完成测试，也可以在没有测试用例的情况下进行探索式测试。这两句话要拆开看。前半句是我们最熟悉的——拿到用例，按步骤操作，比对预期结果与实际结果，记录通过还是失败。这里我要强调执行的质量：一个负责任的执行，不只是"照着点"。你要如实记录实际结果，包括那些"虽然过了、但总觉得怪怪的"现象；遇到失败，要留下可复现的步骤、截图、日志和环境信息。为什么？因为缺陷报告的质量直接决定开发能不能一次修对。很多扯皮，都源于一句"反正就是不行"。

>

后半句是探索式测试：在没有测试用例的情况下进行的测试。注意，"没有用例"不等于"没有方法"。探索式测试靠的是测试人员的经验、对产品的理解和即时的判断，边学、边设计、边执行、边调整。

>

再看自动化执行。定义是：采用测试工具来完成，一般都需要开发自动化测试脚本，然后由工具执行脚本。这意味着自动化执行的前期投入在脚本开发上——所以它适合反复执行的场景，只用一次的场景不值得为它写脚本。

>

课件还提示我们，关于自动化的更多内容，会在后面"单元测试与集成测试""系统测试"和"自动化测试框架"等章节详细讨论，这一页先把概念建立起来。

>

执行阶段还有几件配套的事要记住：一是执行的纪律——用例没跑完不能报"测试完成"，跳过的用例要写明原因；二是缺陷的记录与跟踪——发现、提交、跟踪到关闭；三是执行证据的留存——日志、截图、测试数据。我们课程设计里要求大家"发现缺陷并记录、完成一个非规范的测试报告"，练的就是这一段功夫。

【思政落点 3】讲什么：执行时发现了一个对自己不利的结果——是如实记录，还是轻轻放过。怎么联系知识点：测试执行要求如实记录实际结果、保留失败证据、跟踪缺陷到关闭。落到什么价值观：诚信是测试人员的职业底线；隐去不利结果，等于把风险转嫁给用户，也辜负了团队对测试的信任。

> **板书：**测试执行 → 手工执行：基于详细设计的用例（操作—比对—记录）＋无用例的探索式测试（靠经验与判断，不是靠蛮点）→ 自动化执行：先开发脚本 → 再由工具执行 → 后续章节详述 → 执行三配套：纪律（不跳不瞒）、缺陷记录与跟踪、证据留存。

> **提问：**一位测试同事跑用例时发现某条用例失败，但重跑又通过了，他觉得"重跑是好的，就记通过"——你怎么看？

> **预设回答：**第一种："同意，重跑通过说明没问题。"这是典型错误——把间歇性缺陷掩盖掉了。第二种："应如实记录，标注为偶现，补充日志、时间、环境、数据、出现次数等信息，作为缺陷提交并跟踪。"这是正确回答。第三种："直接报严重缺陷。"方向对，但不留证据，开发无法定位。教师点评：把"偶现缺陷"讲成典型案例——这类缺陷往往最危险，上线后就是偶发故障，且极难复现；用"如实记录＋留证据＋标偶现"三步法收口。

> **易错点 / 考点：**第一，把自动化执行理解成"不用写脚本、工具自己会测"。第二，把探索式测试等同于随机乱点。第三，用"重跑通过即通过"处理偶现失败。考试常考：手工执行与自动化执行的适用场景对比；或"下列属于测试执行阶段工作的是"，干扰项常常是设计用例、编写测试计划这类其他阶段的工作。

> **过渡：**执行里的手工那条路，本质上就是照着脚本走的；自动化那条路，脚本更是核心。下一页我们把"基于脚本的测试"这个传统但基础的方式讲清楚。

【2.78】基于脚本的测试（6 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲基于脚本的测试，英文是 Scripted Testing，简称 ST。它是我们理解后面探索式测试的参照物。

>

> 先看它的核心特征，课件写得很简洁：先设计后执行。也就是说，测试动作不是临场发挥，而是先把要测什么、怎么测、怎么判断结果写清楚，写完了再去执行。这个"先"字，就是 ST 与探索式测试最大的分野。

>

> 第二个要点，Script 到底是什么？课件给了两种情况：在手工测试里，Script 就是 Test case，测试用例；在自动化测试里，Script 就是 Test Script，测试脚本。所以同一个词，在两种执行方式下的产物不一样——一个是给人看的步骤，一个是给工具跑的代码。

>

> 第三个要点，它的阶段很明显。课件列了四步：分析、设计、执行、报告。分析阶段理解需求、界定测试项；设计阶段把用例写出来；执行阶段按用例跑；报告阶段汇总结果、写测试报告。四个阶段一个接一个，边界清楚，所以课件说它属于较传统的测试方式。但传统不等于过时——今天大量的测试工作，尤其是回归测试和对合规性要求高的行业，仍然是基于脚本的。

>

> 第四个要点，课件写了一句很传神的话："有什么开发就有什么测试。"什么意思？开发的产物是需求、设计、代码，ST 就对应地有需求分析、测试设计、测试执行，一一映射。开发走瀑布，测试也走瀑布；开发按模块划分，测试就按模块划分用例。它的好处是结构清晰、可管理、可度量、可交接；它的代价是——需求一变，前面写好的用例就要返工，响应变化比较慢。

>

> 顺带说一下，课件在这一页的最后还附了一句 WebEx Meeting Center 的客户收益描述："使用 WebEx Meeting Center，可以加快业务决策，同时减少差旅成本和时间。"这句话本身是产品文档里的宣传语，不是测试内容。我把它摆在这里是想提醒大家：产品文档、宣传材料里对用户的承诺，恰恰是验收测试要逐条对照核验的依据。文档写了却做不到，测试就要把它标出来。

>

> 把这一页收成一句话：ST 是"先想清楚、写下来、再照着做"，它最大的优点是可控和可复用，最大的短板是应对变化慢。
>
> **板书：**基于脚本的测试 ST → 先设计后执行 → Script：手工测试＝Test case（用例）／自动化＝Test Script（脚本）→ 阶段明显：分析 → 设计 → 执行 → 报告 → 特点：较传统、结构清晰、可管理可度量可交接；响应变化慢 → 口诀"有什么开发就有什么测试"。附：WebEx 客户收益语（产品承诺＝验收核验依据）。

提问：一个需求频繁变更的项目，团队坚持全流程严格的基于脚本的测试，会遇到什么问题？该怎么调整？

> **预设回答：**第一种："没问题，规范总是好的。"这是忽略成本的典型错误。第二种："用例维护成本高、返工多、执行进度被拖累；应该分层——核心稳定的部分保持脚本化，变化快的部分用探索式测试加轻量用例，并缩短设计与执行之间的间隔。"这是正确方向。第三种："干脆全部改成探索式测试。"这是走到另一个极端。教师点评：借第二种提前铺垫 ST 与探索式测试的对比关系，强调"不是选边站，而是按场景配比"；同时把下一节的问题抛出来。

易错点 / 考点：第一，把 Script 只理解成自动化脚本，漏掉手工测试的 Test case。第二，把 ST 的四个阶段记混——是分析、设计、执行、报告，不是需求、计划、设计、执行。第三，认为 ST 已经被淘汰。考试常考：ST 的特征（先设计后执行、阶段性明显），以及 ST 与探索式测试的对比（下一节展开）。记忆口诀：先设后行、四步分明、开发测试一一对应。

过渡：ST 要求先设计后执行，可如果项目节奏快到没时间先写用例呢？下一节我们看另一种风格——探索式测试，它是怎么把学习、设计、执行并行起来做的。

【2.79】探索式测试定义（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

前面几页我们把静态测试、动态测试、手工测试、自动化测试这些"测试方式"逐一过了一遍，现在来到一个很有意思、也特别容易被误解的概念——探索式测试。这一页给出的定义，来自 Cem Kaner 在 2006 年 6 月 6 日软件测试协会会议上的主题演讲《Exploratory Testing After 23 Years》。标题里的"23 年"往前一推，就是 1983 年——也就是说，这个概念从出现到被系统总结，走了二十多年。

定义是这样说的：探索式测试是一种软件测试风格（Style），它强调独立测试人员（Individual Tester）的个人自由和职责（Personal Freedom and Responsibility），为了持续优化其工作的价值（Value），将测试相关学习（Test-related Learning）、测试设计（Test Design）、测试执行（Execution）和测试结果分析（Analysis）作为相互支持的活动，在整个项目过程中并行地执行。

请大家抓住四个关键词。第一是"风格"，它不是一种具体技术，也不是一个工具，而是一种做事的方式；第二是"独立测试人员"，强调个人的自由和职责——自由给你了，责任也一起给你，你不能说"用例没写我就不会测"；第三是四个活动：学习、设计、执行、分析，它们是相互支持、并行进行的，不是先学完再设计、设计完再执行那种串行；第四是"持续优化工作的价值"，这句是灵魂——探索式测试不是乱测，它每一步都在问"我这一轮测下来，对项目到底有什么价值"。

生活里最好的类比是"老医师问诊"。新手照着检查单一项一项勾，老医师一边听你说、一边追问、一边当场决定下一个该查什么，学、想、查、判断是同时发生的。软件行业里也一样，一个刚上线、需求天天改的移动端产品，就特别适合用这种风格去探。

课件备注里还给了两个词：Interpretation 是"解释、说明"，Mutually 是"互为、相互地"。后者正好对应定义里那句"相互支持的活动"，请大家顺手记住。

板书：探索式测试（ET）→ 风格 Style ≠ 技术 → 独立测试人员：个人自由＋职责 → 学习→设计→执行→分析（相互支持、并行）→ 持续优化价值｜出处：Cem Kaner 2006.6.6 演讲（概念源头 1983 年）

提问：定义里说四个活动是"并行执行"的，那它跟"想到哪测到哪"的乱测，差别到底在哪里？

预设回答：①"没差别，就是随机点一点"——典型错误。要追一句：随机点的人有没有在记录、分析、调整下一轮策略？没有分析就没有探索，那只是碰运气。②"差别在于有没有用例"——部分对，探索式测试也可能有松散的用例甚至测试章程，关键是学习和设计在执行现场实时发生。③"差别在于测试人员的能力和责任心"——说得好，这正是定义里"个人自由和职责"的落点：自由越大，越依赖自律。

易错点 / 考点：①把探索式测试等同于 Ad-hoc（即兴）乱测——错，它是"有结构的即兴"。②把"并行"理解成"不做设计"——错，设计还在，只是挪到执行过程中去做。③选择、判断题常考这个定义的出处：Cem Kaner，2006 年主题演讲，概念源头可追到 1983 年。记忆口诀：**风格、独立、四活动并行、价值驱动**。

过渡：知道了探索式测试是什么，我们自然会问：它跟传统的、按用例一条条执行的测试，到底该怎么摆？下一页课件给出一条谱系。

【2.80】基于脚本的测试 vs. 探索式测试（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是课件里第一张"对照图"，它把各种测试做法排成了一条从"规范"到"自由"的连续谱。谱的最左端是 Fully Scripted Testing，完全脚本化的测试，用例写得满满的，一步一步照着执行；往右走是 Automated Tests，自动化测试；再往右进入中段，出现了 Test Design、Test Execution，以及松散的测试用例和测试场景；接着是 Bug Hunting，缺陷猎捕；再往右是 Exploratory Testing 探索式测试、Product Exploration 产品探索；最右端是 Ad-hoc Testing，完全自由的探索，想到哪测到哪。课件还摆出了"角色扮演测试"这一类做法，提示我们中段存在大量介于规范与自由之间的测试形式。

这张图最想告诉大家的一句话是：**这不是一道二选一的判断题，而是一条光谱**。左端追求可控、可重复、可度量；右端追求灵活、应变、靠人的判断。真实项目里你几乎不会只用一端，而是看阶段、看风险、看被测对象，来决定自己站在谱上的哪一段。回归测试、验收测试、需要反复执行的冒烟测试，就往左边站，越自动化越好；一个全新的功能、一份含糊的需求、一个还没被任何人碰过的模块，就往右边站，靠人去探。

为什么会有这么一条谱？因为左右两端的成本结构不一样。规范脚本前期投入大：要分析需求、要写用例、

要评审，但一旦写好就能反复执行，边际成本很低；自由探索前期几乎不用投入，立刻就能开始，可它高度依赖执行者的水平和状态——今天这个人能发现的问题，换个人可能一个都发现不了。

所以判断标准很朴素：**这段测试会不会被重复执行很多次？需求是不是已经稳定？**两个都"是"，就往规范一侧走；两个都"否"，就往自由一侧走。

顺带说一句，课件把 Ad-hoc 放在最右端，也是在提醒我们：探索式测试虽然位于谱的右侧，但不等于最右端那个"完全自由的探索"，它是有方法、有记录、有分析的——这一点上一页讲定义时已经埋了伏笔。

板书：规范 ←————————————→ 自由

Fully Scripted Testing → Automated Tests → Test Design / Test Execution / 松散的测试用例、测试场景 → Bug Hunting → Exploratory Testing → Product Exploration → Ad-hoc Testing（完全自由的探索）｜中段还有角色扮演测试等形式

提问：同一个登录功能，如果它是长期维护的核心模块，你把它放在谱上的哪一段？如果它是本周才加的一个实验性开关呢？

预设回答：①"核心模块放左边，实验性功能放右边"——标准答案。追问为什么：核心模块回归次数多、需求稳定，脚本化和自动化的收益最高。②"都放右边，人工测发现的问题更多"——要纠正：人工发现的问题"看起来多"，但不可重复、不可度量，回归时无法保证不漏。③"都放左边，全写成自动化"——也要纠正：需求都没定的实验功能写脚本是浪费，需求一改脚本全废。

易错点 / 考点：①最易错：以为探索式测试就等于 Ad-hoc 测试。课件把两者分列在谱上的不同位置，Ad-hoc 是"完全自由的探索"，ET 是右侧但有方法的一档。②选择题常考两端的名称：最规范端是 Fully Scripted Testing，最自由端是 Ad-hoc Testing。③简答题常问"什么时候用脚本化、什么时候用探索式"，答题三步法：**看重复次数 → 看需求稳定性 → 看风险与人员水平**。

过渡：谱系图讲完了，下一页课件把两端拉成一张逐条对照表，我们一条一条看。

【2.81】ET vs. ST -2（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是探索式测试（ET）和基于脚本的测试（ST）的第二张对比页。上一页看的是它们在谱系上的位置，

这一页看的是一条一条的工作方式差异。先看左边的 Scripted Testing，脚本化测试：第一，先设计、后执行，顺序不能倒；第二，强调逻辑分析，靠文档和推理把用例推出来；第三，关注需求和测试文档，文档是它的主战场；第四，有明确的测试标准——什么算通过、什么算失败，事先定死；第五，强调评审、可控，用例要评审，执行要可控；第六，总体上严谨、规范。

再看右边的 Exploratory Testing：第一，学习、设计和执行并行，边学边设计边测；第二，上下文驱动，测什么、怎么测，由当前这个产品、这个版本、这段时间的上下文决定，而不是照搬模板；第三，强调个人能力，同样的时间，高手和新人的产出差距极大；第四，Test Oracle，测试判定准则，这是重点——脚本化测试的判定标准写在用例里，而探索式测试常常要测试人员现场判断"我看到的这个现象算不算缺陷"；第五，关注与产品的交互，测试人员是真的像用户一样在用；第六，拥抱变化、乐趣，需求变了不慌，反而觉得有新东西可探。

我们把它压成一句话：**脚本化测试是把不确定性提前消灭在文档里，探索式测试是把不确定性留到现场、用人的判断去消化。**左边赌的是"我事先想得够全"，右边赌的是"我现场反应够快"。所以左边最怕需求变更，一变，用例全废；右边最怕人员不行，人一弱，测试就变成瞎点。

页面上标的是"ET vs. ST -2"，说明这是对比的第二页——第一页看谱系位置，第二页看工作方式，两页合起来才是完整的一对。

板书：ST：先设计→后执行｜逻辑分析｜重需求和测试文档｜标准明确｜强调评审、可控｜严谨规范

ET：学习＋设计＋执行并行｜上下文驱动｜靠个人能力｜Test Oracle 现场判定｜关注与产品交互｜拥抱变化、乐趣

一句话：ST 把不确定消灭在文档里；ET 把不确定留到现场消化

提问：右边提到探索式测试要有 Test Oracle，那如果一个测试人员说"我觉得这里不对劲，但我一时说不清为什么"，这算不算发现了缺陷？

预设回答：①"不算，说不清就不是缺陷"——要纠正：发现异常只是第一步，接下来要做的是最小化复现、对照需求文档和同类产品的行为，把它变成可描述、可举证的缺陷，说不清只是起点。②"算，凭直觉就行"——也要纠正：缺陷必须能复现、能说明影响，否则开发无法修、也无法验证。③"要看能不能稳定复现、是否偏离需求和用户期望"——好答案，这正好落到缺陷报告的要素上。

易错点 / 考点：① Test Oracle 常被译错成"测试预言"，标准说法是"测试判定准则/预期结果来源"，考试要能解释它回答的是"凭什么判断这次测试通过了"。②简答高频：ST 与 ET 的差别，至少要答出"是否先设计后执

行""是否上下文驱动""对文档和标准的依赖程度"三条。③易错：把"拥抱变化"当成"不做计划"——ET 同样要有测试章程和时间盒。

> **过渡：**测试做完了，结果就摆在那里，那我们怎么判断"测够了没有、这个版本能不能发"？下一页讲测试结果与过程评估。

【2.82】测试结果和过程评估（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

前面几页讲的是怎么测，这一页讲的是测完之后怎么看。课件把它分成两件事：测试结果评估和测试过程评估。这两件事经常被混为一谈，其实一个看的是产品，一个看的是我们自己的工作。

先看测试结果评估，课件给了三个分析角度。第一，分析测试覆盖率，目的是了解测试是否充分——覆盖率高不代表质量一定好，但覆盖率低一定说明还有大片区域没人碰过，这是"测没测到"的问题。第二，做缺陷的趋势分析，看缺陷是越来越多还是越来越少，从而了解缺陷是否已经收敛——如果每天发现的缺陷还在往上走，说明这个版本远没到稳定的时候。第三，做缺陷的分布分析，看缺陷集中在哪几个模块、集中在哪几类严重程度，再基于缺陷来评估当前被测试版本的质量。注意这里的落脚点是"版本质量"，也就是为"这个版本能不能发布"提供依据。

再看测试过程评估。课件讲得很清楚：结合测试计划来进行评审，相当于把计划的测试活动和实际执行的活动做比较，了解测试计划执行的情况和效果。说白了就是"计划 vs 实际"的偏差分析：计划里要执行的用例，实际只跑了一部分，差在哪里？计划的回归时间被压缩了，是人力不够还是环境卡住了？过程评估的目的不是追责，而是发现流程里的堵点，让下一轮计划定得更靠谱。

记住这三层顺序：**先看结果够不够（覆盖率），再看问题收敛没有（趋势与分布），最后回过头看自己干得怎么样（过程偏差）。**很多同学写测试总结只写"测了多少条用例、发现多少缺陷"，那只是结果的一半；把覆盖率、趋势、分布和计划偏差补齐，才叫会评估。

> **板书：**测试结果评估 → 覆盖率（是否充分）→ 缺陷趋势（是否收敛）→ 缺陷分布（集中在哪里）→ 版本质量判断

测试过程评估 → 计划的活动 vs 实际的活动 → 偏差与效果 → 改进下一轮计划

三层顺序：测没测到 → 问题收没收敛 → 自己干得怎么样

提问：如果 A 版本发现 100 个缺陷，B 版本发现 30 个缺陷，能不能直接说 A 的质量比 B 差？

预设回答：①"当然更差，缺陷多"——典型错误。追问：这 100 个是随便点出来的，还是照着系统化的用例测出来的？测试强度不同，发现数就不可比。②"要看缺陷是否收敛、分布在哪里、测试投入有多少"——好答案，三个角度全用上了。③"要看严重程度"——对，同样是 100 个，90 个界面文案错误和 90 个数据计算错误，结论完全相反。

易错点 / 考点：①最大误区：把"缺陷总数"直接等同于"质量水平"，必须结合测试充分性和投入来看。②"收敛"要会解释：新增缺陷数随时间下降，并且严重缺陷已基本关闭。③选择题常考两种评估的区分：结果评估看产品与版本，过程评估看计划执行。口诀：**覆盖看充分，趋势看收敛，分布看集中，过程看偏差。**

过渡：到这里第 2 章的概念就全部讲完了，我们花几分钟把整章串成一张全景图。

【2.83】本章小结（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是全章"收口"的地方，一共五条，我们一条一条过。

第一条：缺陷是质量的对立面。这句话的意思是，谈缺陷之前必须先谈质量——质量是什么？是软件满足规定需求和隐含需求、也就是满足用户期望的程度。所以课件提醒我们，要借助产品质量模型和使用质量模型来理解质量：产品质量模型看的是软件本身的属性，使用质量模型看的是软件被真正使用起来以后，对用户任务产生了什么效果。

第二条讲缺陷的因果链条：缺陷是内部错误，它在外部表现为失效。这一点非常关键——用户看不到你代码里那行写错的条件判断，用户只看到"点结算没反应"。而缺陷的来源很多，课件点了三个大源头：需求定义、设计和代码。需求写歪了，后面怎么努力都歪；设计漏了边界，代码再漂亮也补不回来。课件还强调一句：缺陷要尽早发现、尽早修正，否则带来的劣质成本越大——这就是我们前面讲过的成本放大效应，越往后修越贵。

第三条列了软件测试方式的几组对比：静态 vs 动态、主动 vs 被动、手工测试 vs 自动化测试、基于脚本的测试 vs 探索式测试。这一条是"横向"的，讲的是同一件事可以用哪些不同方式去做。

第四条列了单元测试、集成测试、系统测试和验收测试。课件这里字面写的是"软件测试方式"，但列出来的这四项其实是"测试级别"，也有教材叫测试阶段。易错点就在这里：方式回答"怎么测"，级别回答"在哪一层

测、由谁来测"，这是两条不同的轴线。后面讲单元测试与集成测试的章节会专门展开这四级。

第五条讲软件测试的工作范畴，一共六件事：测试需求分析、测试策略制定、测试计划、测试设计、测试执行、测试结果和过程评估。这六件事就是大家做课程设计时要走完的完整链路，也是本课程第 3 篇各章的主题。

课件的落脚点写得很清楚：帮助同学们建立软件测试的整体全景图。这张图请务必装进脑子里——纵向是级别，横向是方式，中间贯穿的是六项工作，最上面压着的是质量与缺陷这条主线。

【思政落点 1】科学精神・质量观・诚信。质量模型和缺陷分类不是背概念，而是在训练一种"凡事讲证据、讲分类、讲因果"的科学态度：说质量要有指标，说缺陷要有复现和影响。再往前一步就是职业诚信——测试人员发现了缺陷却隐去不报，等于把风险留给用户、留给同事。所以这一章真正要立的规矩是：数据如实记录，不利结果不隐瞒。

板书：①缺陷＝质量的对立面（产品质量模型＋使用质量模型）②缺陷：内部错误→外部失效；来源＝需求/设计/代码；越早修越便宜 ③方式：静态 vs 动态、主动 vs 被动、手工 vs 自动化、脚本 vs 探索 ④级别：单元→集成→系统→验收 ⑤工作范畴：需求分析→策略→计划→设计→执行→结果与过程评估

提问：请用一句话说出"方式"和"级别"的区别，并举一个例子。

预设回答：①"方式是怎么测，级别是在哪一层测。比如对登录模块做单元测试——单元测试是级别；用白盒还是黑盒、手工还是自动化，是方式。"——标准答案。②"级别就是阶段，方式就是方法，差不多"——要纠正：混用会导致计划里写不清"谁在什么阶段用什么手段"。③"方式包含级别"——错误，正好用课件第四条那句表述来纠偏。

易错点 / 考点：①课件第四条文字写的是"软件测试方式"，但列的是单元/集成/系统/验收，实质是级别——考"测试级别有哪些"就答这四项。②顺序题：六项工作范畴的先后顺序（需求分析在计划之前）不能颠倒。③本章小结本身是简答题高发区，建议按"质量—缺陷—方式—级别—工作范畴"五段来背。

过渡：概念都清楚了，接下来是两件要动手做的事——课后作业和本章的思考题。

【2.84】作业（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本课程的第一次课后作业，我把要求念清楚，也讲清楚我到底想看什么。作业要求是：选择同类的两到三个产品，比如词典类、视频播放器这一类，对它们做一次质量比较。具体四件事：第一，详细地分析它们的外部质量和使用质量；第二，列出它们之间有明显差异的质量属性；第三，列出这类软件在需求评审时需要关注的功能点；第四，列出这类软件在设计评审时需要关注的非功能特性。

先解释两个词，不然大家容易写偏。外部质量，指的是从外面看得见、可以测量的质量特性，比如功能是否齐全、界面是否好用、运行是否流畅、在不同设备上是否兼容。使用质量，指的是放到真实使用场景里去看：它能不能帮用户把任务顺利完成、效率高不高、用户满不满意、有没有风险。同样一个词典软件，外部质量看它词库多大、查询多快；使用质量看你查一个生词到看懂例句，一共花了多少秒、中间有没有被广告打断。

第三条和第四条是把"比较"转换成"评审关注点"，这是本次作业最有价值的部分。需求评审关注功能点，就是在需求阶段要问清楚哪些功能、哪些流程、哪些边界；设计评审关注非功能特性，就是性能、安全、兼容、易用、可维护这些"不是功能却决定体验"的属性。为什么这么设计？因为这两个正是测试人员最应该提前介入的会议。

交作业时我提醒三点：一是写清你比较的是哪几个产品、什么版本、什么时候测的，这叫测试环境信息；二是质量属性的差异要给出证据，比如截图或具体操作步骤，不能只写"感觉 A 比 B 好"；三是把"我认为需要关注的点"和"它实际上做到了什么"分开写，前者是你的判断，后者是事实，混在一起就分不清哪是分析、哪是臆测。

这门课的考核方式是过程性考核加上课程设计作品与答辩，这份作业就是过程分的一部分，请认真做。

板书：同类产品 2–3 个（词典、播放器…）→ ①外部质量＋使用质量分析 ②有明显差异的质量属性 ③需求评审关注的功能点 ④设计评审关注的非功能特性｜作业要求：写清版本与环境、差异要给证据、判断与事实分开写

提问：如果让你比较两款词典 App，你会用哪个指标来判断"使用质量"的差别？

预设回答：①"看谁的词库大"——那是外部质量里的功能属性，还没到使用质量。②"查同一个生词，从输入到看完释义和例句的总耗时、总步骤数"——好答案，这就是任务级的效率指标。③"看应用商店评分"——可以当参考信号，但要追问：评分受价格、广告、运营影响，不能当作质量测量的主证据。

易错点 / 考点：①易错：把"使用质量"当成"用户体验好不好"这种纯主观感受，它其实是围绕任务效果、效率、满意度、风险的可观察指标。②功能点与非功能特性容易写反：需求评审数功能，设计评审谈非功能。③这份作业的内容与期末课程设计里的"测试需求分析"直接相关，别当形式作业交。

> **过渡：**作业之外，课件还留了三道思考题，这三道题就是本章的考点清单，我们把它讲透。

【2.85】思考题（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页的三道思考题，请大家一定当成复习提纲来用，因为本章期末要考的点基本都在里面。我把参考答案一题一题讲。

第一题：为什么说缺陷是质量的对立面？先明确质量的定义——质量是软件满足规定需求和隐含需求、也就是满足用户期望的程度。而缺陷恰恰是违背需求与期望、导致结果不正确的问题。所以两者是同一个坐标轴上的两端：质量是"满足期望"的度量，缺陷是"违背期望"的实证。缺陷越多、越严重，质量就越低；反过来，想让质量高，最直接的动作就是减少缺陷，尤其是减少严重缺陷。答题时一定要走三步：先给质量下定义，再讲缺陷是"违背"，最后落到"数量与严重程度共同决定质量水平"，三步缺一不可。

第二题：为什么静态测试和动态测试是一对对立统一体？"对立"讲的是手段：静态测试不运行程序，靠阅读、评审、检查文档和代码来发现问题；动态测试必须运行程序，通过输入数据、观察输出来发现问题。一个不动、一个要动，这是手段上的对立。"统一"讲的是目标和互补关系：两者的目标完全一致，都是发现缺陷、评价质量；而且它们能覆盖对方覆盖不到的地方——静态测试擅长抓需求里的二义性、设计上的缺陷、编码规范与逻辑问题，这些问题程序跑起来未必暴露；动态测试擅长验证运行时的真实行为，比如计算错误、性能瓶颈、并发问题，这些问题光看代码很难断定。所以标准答案是：手段对立，目标统一，功能互补，缺一不可。

第三题：为什么说测试需求分析是测试计划、测试设计的基础？一句话，测试需求分析回答的是"测什么"——测试项有哪些、范围到哪里、优先级怎么排、可测性如何。这些答案一旦有了，测试计划才有依据：范围定了才能估工作量和资源，优先级定了才能排进度、安排人力，风险点清楚了才能定应对措施。测试设计同样依赖它：只有明确了测试项和它的可测性，才能决定用哪些方法、覆盖到什么程度、判定准则是什么。反过来想，如果跳过测试需求分析直接写计划，就会出现"计划里排满了场次，却不知道要测什么"；直接写用例，就会出现"用例写得漂亮，却漏了最重要的业务场景"。这正是后面"测试需求分析与测试计划"那一章要专门解决的问题。

三道题串起来其实就是本章的主线：先有质量与缺陷的概念，再有静态与动态两类手段，最后用测试需求分析把概念接进工程流程。

> **板书：**①质量＝满足规定需求＋隐含需求（期望）；缺陷＝违背需求与期望 → 同一坐标轴两端，缺陷的数量与严重程度决定质量水平

②静态（不运行程序：阅读、评审、检查；抓需求二义性、设计缺陷、规范问题）↔ 动态（运行程序：抓计算错误、性能、并发）→ 手段对立、目标统一、功能互补

③测试需求分析＝回答"测什么"（测试项、范围、优先级、可测性）→ 计划据此定范围、资源、进度、风险；设计据此定方法、覆盖与判定准则

> **提问：**如果只允许你做静态测试或者只允许你做动态测试，你选哪一个？为什么？

> **预设回答：**①"选动态，能跑起来才算真测试"——要纠正：跑起来之前，需求二义性和设计缺陷已经在里面了，动态测试往往只能看到症状，说不出根因。②"选静态，成本低"——要纠正：不运行就永远无法确认运行时行为和系统级质量。③"两个都不能只选一个"——正确，正好回扣"对立统一"。

> **易错点 / 考点：**①把静态测试等同于"看代码"——不完整，需求文档、设计文档、测试用例甚至用户手册的评审都属于静态测试。②只答"一个运行一个不运行"而漏掉"互补"，最多得一半分。③判断题高频：静态测试也能发现缺陷（对）；缺陷是内部错误、失效是外部表现（对）；测试需求分析在测试计划之后进行（错，必须在前）。口诀：**对立看手段，统一看目标，基础看"测什么"。**

> **过渡：**三道思考题之后，本章还留了一个动手实验，我们把它的要求说清楚。

【2.86】实验：完成一个简单的测试过程（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这是本章的实验，名字很朴素——完成一个简单的测试过程。注意"过程"两个字，它不是让你背概念，而是让你完整走一遍"分析—执行—记录—报告"的闭环。
>
> 第一步，选被测对象。课件给了两个来源：优先用你在软件工程课或其他课程里开发过的软件系统，从里面选定一到两个功能模块；如果手上没有系统，就用课件给的两个练习站点——saucedemo 和 automationpractice。这两个都是业界经典的练习用电商站点，功能完整、又没有真实业务风险，随便点、随便试，非常适合第一次做功能测试。
>
> 第二步，先做初步的功能测试分析。课件提了两个具体问题：一是了解功能操作的路径，也就是从哪进、点

几步能到；二是明确要输入哪些数据，以及有哪些特殊、异常的数据或操作。这一步就是"设计"，不要一上来就乱点，先把入口、路径、输入项写清楚。比如登录功能，你要问：输入框接受什么格式？空值、超长值、错误密码、特殊字符算不算异常？能不能只用键盘 Tab 和回车走完全流程？

第三步，执行手工测试。课件的要求很形象：像用户使用产品那样操作软件，进行手工测试，发现缺陷并记录。请务必做到"操作有路径、缺陷有步骤、现象有截图"。记录一条缺陷至少要有四样东西：怎么操作、期望什么结果、实际什么结果、能不能稳定复现。

第四步，完成一个非规范的测试报告。这里的"非规范"我要特别强调：它的意思是第一次不要求你套正式模板、不要求完整的测试计划与评审流程，而不是"可以随便写"。报告里至少要有被测对象与版本、测试的功能模块、用了哪些数据和异常输入、发现的缺陷清单，以及你自己的一句结论。

具体细节见教材第 40 到 41 页。这个实验是后面课程设计的基础动作，工具上先不做硬性要求，先把"发现缺陷并把它说清楚"这件事做到位。

板书：①选对象：自研系统 1–2 个功能模块／saucedemo、automationpractice ② 初步分析：操作路径＋输入数据（含特殊、异常） ③手工测试：像用户一样用，发现并记录缺陷 ④非规范报告（非规范 ≠ 随便写）｜详见教材 P40–P41｜一条缺陷四要素：操作步骤、期望结果、实际结果、可复现性

提问：在登录功能里，你会设计哪些"特殊、异常"的数据或操作？

预设回答：①"输入错误的密码"——对，但太基础，继续追问：空用户名、超长字符、前后空格、大小写、SQL 特殊字符呢？②"连续点五次登录按钮"——好例子，属于异常操作，考验防重复提交。③"点登录后再按浏览器后退，看状态对不对"——很好的思路，这是流程与状态类的异常路径。教师点评：特殊与异常输入通常从"格式、长度、边界、顺序、频率、状态"六个方向去想。

易错点 / 考点：①把"非规范报告"理解成可以只写结论——报告必须有证据链。②缺陷记录只写"这里有 bug"——工程和考试都要求给出复现步骤与期望/实际的对比。③容易漏掉"分析"这一步直接开测，导致测试没有针对性；而这一步恰恰是本章测试设计意识落到手上的地方。

过渡：这节课的内容就到这里，最后我们用一句话帮大家收个尾。

【2.87】感谢聆听（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

好，以上就是本节课的内容。这一章讲到这里，我们把软件测试的基本概念过了一遍：缺陷是质量的对立面，缺陷是内部错误、在外部表现为失效；测试方式有静态与动态、主动与被动、手工与自动化、基于脚本与探索式；测试级别有单元、集成、系统、验收；工作范畴从测试需求分析、测试策略、测试计划、测试设计、测试执行，一直到测试结果和过程评估。请回去把本章小结那张全景图和三道思考题结合起来复习，作业和实验按要求提交。

我是广东金融学院的周宇文，课件和资料会发到课程群，有问题随时找我。感谢聆听，下次课我们进入新的内容。

> **板书：**质量与缺陷 → 方式（静态/动态、主动/被动、手工/自动化、脚本/探索）→ 级别（单元/集成/系统/验收）→ 工作范畴（需求分析→策略→计划→设计→执行→评估）｜下次课预告 · 作业与实验提交提醒 · 联系方式

提问：请用一句话说出本章你最有收获的一个概念。

> **预设回答：**①"缺陷是内部错误，失效是外部表现"——能把这两层分开，说明因果链理解了。②"静态测试不运行程序、动态测试运行程序，两者互补"——分类思维到位。③"测试需求分析必须先于测试计划"——工程顺序意识有了，这正是后面课程设计的起点。

易错点 / 考点：收尾前再点一遍本章三个高频错点——探索式测试不等于 Ad-hoc 乱测；"软件测试方式"那一条列出的其实是测试级别；测试需求分析在计划与设计之前，顺序不能颠倒。

> **过渡：**第 2 章到此结束，下次课我们进入新的一章，开始讲具体的测试方法与技术。

四、思政落点清单（共 16 处）

序号	所在页	落点主题
1	2.5	讲一个真实的工作场景：某团队上线后收到大量用户投诉，工程师在周报里写"功能符合需
2	2.13	讲一个课堂可讨论的情境：同一次测试里，你发现了三个缺陷，其中两个是界面小瑕疵，一
1	2.24	讲什么故事：霍珀团队把一只飞蛾贴进日志本并写明"这是第一个实际发现的 bug"。怎
1	2.28	讲"判断缺陷必须有依据"，可以联系工程中的标准意识：需求文档、设计规范、行业标准

2	2.30	缺陷构成分析里有一个职业底线问题：不能只统计"好修好说"的缺陷，把难看的、自己漏
3	2.32	成本曲线讲的是"欠下的质量债早晚要还"，这正好对应职业责任：明知道某个功能没有验
1	2.46	请记住评审席上的两条底线。第一条叫"对事不对人"：我们用证据和事实说话，指出的是
2	2.52	这一页其实藏着一个很硬的职业态度：不要满足于"文档上通过了"。规格说明书也是人写
1	2.56	同一段代码，用黑盒视角和白盒视角去测，能发现完全不同的缺陷——这正是科学精神里"
2	2.58	分层背后是一种"把大问题拆开、逐层把关"的工程思维，也是工匠精神里"每道工序都有
3	2.61	代码评审能发现 50%～70%缺陷这个数据，正好用来说明"同行评审不是挑刺，而是互
4	2.64	持续集成这条流水线最考验的其实是诚信和责任感——测试红了却直接合入、失败用例被注
1	2.68	讲什么：定义里那句"如同用户所合理期待的那样"。怎么联系知识点：验收测试要管隐含
2	2.76	讲什么：用例库是团队一点一点积累起来的知识资产。怎么联系知识点：正因为良好用例会
3	2.77	讲什么：执行时发现了一个对自己不利的结果——是如实记录，还是轻轻放过。怎么联系知
1	2.83	科学精神·质量观·诚信。质量模型和缺陷分类不是背概念，而是在训练一种"凡事讲证据

第 3 章　软件测试方法

第 3–4 周 · 两次课各 2 学时（共 180 分钟） | 教材：朱少民《软件测试方法和技术（第 4 版）》

《软件测试课程设计》第 3 章 讲稿（第 3–4 周 · 两次课 180 分钟）

> 课程：软件测试课程设计（154442008） | 广东金融学院 计算机学院 | 专业选修
>
> 章节：第 3 章 软件测试方法 —— 从直觉经验到形式化，七大方法论与覆盖准则
>
> 学时：**2 次课（第 3 周、第 4 周各 2 学时，共 180 分钟）**
>
> 教材：朱少民《软件测试方法和技术（第 4 版）》，清华大学出版社 2022（本章＝书页 42–75）
>
> 配套课件：逐页还原版 **109 页**（`lecture3/`，云端 http://111.231.0.226:2345/lecture3/index.html）
>
> 教案对应：第 3 次课「软件测试方法（一）：基于直觉与经验」＋第 4 次课「软件测试方法（二）：基于模

> 型与形式」
>
> 本讲重点：测试方法分类框架、等价类划分与边界值分析、判定表与因果图、逻辑覆盖系列与基本路径、基于模型的测试、方法选择与组合
>
> 本讲难点：组合方法的约束处理（因果图／判定表化简／Pairwise）；覆盖强度的递进关系与圈复杂度；模型抽象与状态迁移覆盖准则；方法成本与收益的权衡
>
> **本讲稿为可照读完整版**：每页含【口播（可直接朗读）】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为时间紧张时可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能按依据说出测试方法的类别（基于直觉与经验／输入域／组合与优化／逻辑覆盖／缺陷模式／模型／形式化），并说明 SWBOK 的分类口径；② 能给出等价类划分与边界值分析的划分原则与取点规则；③ 能画出判定表与因果图、说明条件约束（E/I/O/R/M）的含义；④ 能逐一说明语句、判定（分支）、条件、判定-条件、条件组合、MC/DC、基本路径覆盖的定义与包含关系；⑤ 能说出 MBT 的建模要素与常见覆盖准则（全状态／全迁移／全路径）、形式化方法的三部曲与适用场景
能力	① 能针对一个具体输入规格划分有效／无效等价类并列出边界值用例；② 能对含复合条件的判定设计满足 MC/DC 的用例；③ 能由源代码画控制流图、计算圈复杂度并导出基本路径集；④ 能用 Pairwise／正交表把组合爆炸压到可执行规模；⑤ 能根据需求形态、风险等级、时间与人员能力选择方法组合并说明理由
素质（思政）	① 由覆盖率数字与缺陷发现的关系，建立「不为指标而测试、如实报告结果」的职业诚信；② 由安全关键系统（航空 DO-178C、医疗、轨交）采用 MC/DC 与形式化方法，体会敬畏生命、精益求精的工程伦理；③ 由错误猜测法「经验必须沉淀为检查表与缺陷模式」，理解个人经验要转化为团队资产、知识共享的协作精神；④ 由探索式测试的会话章程与记录，训练规范留痕、可追溯的法治意识与责任担当

本讲要建立的一个总印象：测试方法没有优劣，只有**适用**——所有方法服务于同一个目标：**以可接受的成本，尽早发现更多、更严重的缺陷**；方法的进步方向是**从依赖个人经验，走向可度量、可复用、可自动化**。

二、时间分配总表

> ⚠ **容量提示（务必先读）**：本章课件 **109 页**，是全课程页数最多的一章，本讲稿为**素材全量版**（口播约 5 万字量级，≈3 小时纯朗读）。教案把本章排在**两次课（180 分钟）**，两次课也讲不完全部页，请务必按下表取用：
>
> 1. **重点页全文慢讲（2–3 分钟/页）**：
> 3.4、3.8、3.13、3.16、3.23、3.25、3.27、3.32、3.39、3.48、3.49、3.52、3.53、3.56、3.57、3.61、3.63、3.67、3.68、3.73、3.76、3.79、3.82、3.85、3.89、3.91、3.96、3.99、3.105、3.106。
>
> 2. **常规页快速带过（0.5–1 分钟/页）**：只讲口播第一段（结论）＋板书＋1 个提问。
>
> 3. **可课后阅读页（★）**：★3.5–3.7、★3.12、★3.14、★3.17–3.18、★3.20、★3.26、★3.28–3.31、★3.33–3.38、★3.40–3.41、★3.44–3.45、★3.47、★3.50–3.51、★3.54、★3.58–3.60、★3.62、★3.64–3.66、★3.69–3.72、★3.74–3.75、★3.77–3.78、★3.80–3.81、★3.83、★3.86–3.88、★3.90、★3.92–3.95、★3.97–3.98、★3.100–3.104。
>
> 4. **练习/作业页**：3.15、3.21、3.25、3.98 的练习当堂做（答案口径见本稿对应页）；3.106 思考题留作课后；3.107、3.108 两次作业必须用 5 分钟交代清楚提交要求与评分点。

第 3 次课（第 3 周 · 90 分钟）：从经验到组合

时段	页码	内容	分钟	备注
第 1 节	3.1–3.8	本章地图与测试方法分类框架（方法论 vs 具体方法・SWBOK 分类）	12	**3.4、3.8 必讲**；★3.5–3.7 可略
第 1 节	3.9–3.11	3.1 基于直觉和经验的方法（ALAC／随机测试・错误猜测法）	12	**3.11 错误猜测法是教案重点**，配「文件上传」案例讨论
第 1 节	3.12–3.21	3.2 基于输入域的方法（等价类划分・边界值分析）	22	**3.13、3.16 必讲**；3.15／3.21 当堂练习
——	——	**课间休息 10 分钟**	——	——

第 2 节	3.22–3.31	3.3 基于组合及其优化的方法（判定表法・因果图法）	18	**3.23、3.25、3.27 必讲**；★3.28–3.31 可略
第 2 节	3.32–3.41	Pairwise 方法・正交实验法・工具与小结	12	**3.32、3.39 必讲**；工具页可略
第 2 节	3.42	小结与第一次课收口＋作业二布置	4	交代 3.108 作业要求

第 4 次课（第 4 周 · 90 分钟）：从代码到模型

时段	页码	内容	分钟	备注
第 1 节	3.43–3.56	3.4 基于逻辑覆盖的方法（语句・判定/分支・条件・判定-条件・条件组合・MC/DC）	26	**3.48、3.49、3.52、3.53、3.56 必讲**（覆盖强度递进是考点）
第 1 节	3.57–3.65	基本路径覆盖（控制流图・圈复杂度・独立路径集）	14	**3.57、3.61、3.63 必讲**；3.59–3.60 现场画图
——	——	**课间休息 10 分钟**	——	——
第 2 节	3.66–3.70	3.5 基于缺陷模式的测试（DPBT 与自动化）	10	**3.67、3.68 是教案重点**
第 2 节	3.71–3.83	3.6 基于模型的测试（MBT・功能图／状态迁移・模糊测试・变异测试）	16	**3.73、3.76、3.79、3.82 必讲**
第 2 节	3.84–3.95	3.7 形式化方法（形式化规范・形式化验证・有限状态机・Spec Explorer）	12	**3.85、3.89、3.91 必讲**；★3.92–3.95 可略
第 2 节	3.96–3.104	补充：基于场景的测试方法（事件流图・示例）	8	**3.96、3.99 必讲**，与敏捷测试呼应
第 2 节	3.105–3.109	本章小结・思考题・作业一/二・结束	4	3.106 思考题课后完成；3.107 作业一当堂说明

> **压缩优先级**：先压缩 ★ 标记的示例／工具页与历史演进页 → 再压缩同类方法的第二个示例（保留第一个）→ 最后才压缩 3.4 节逻辑覆盖与 3.2 节输入域的核心页。
>
> **3.13、3.16、3.23、3.27、3.32、3.48、3.52、3.56、3.57、3.61、3.67、3.73、3.76、3.79、3.85、3.91、3.96、3.99 是考点密集页，不要压缩。**

三、逐页讲稿

> 使用说明：每页第一段是**可直接朗读的口播**；其后是板书、提问与预设回答、易错点/考点、过渡语。板书建议边讲边写，不要一次写完。
>
> 页码锚点与课件页一一对应（3.1–3.109），可用课件网址逐页对照：http://111.231.0.226:2345/lecture3/index.html

【3.1】软件测试方法和技术（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 各位同学，我们开始上课。我是这门课的任课教师周宇文，接下来两次课由我带大家把这一章过一遍。先说一个容易被忽略的细节：这一章的章名，课件上写的是"第 3 章 软件测试基本方法"，教材上写的是"第 3 章 软件测试方法"——同一章两种写法，答题以教材为准，这个差别后面我还会再点一次。
>
> 从这一节课开始，我们进入全课程方法密度最高的一章。前面两章我们干了什么？第 1 章讲的是"为什么必须测"，第 2 章讲的是"软件缺陷是什么、测试分成哪几类、我们从哪个位置介入"。那些问题回答的都是"是什么、分几类"。而从今天起，我们要回答的是"到底怎么测"——具体到一个输入框、一段判定语句、一条业务链路，你手上应该拿起哪一件工具，用几组数据，凭什么说测够了。
>
> 这一章课件有一百多页，我们会用两次课讲完：第一次课从直觉经验走到组合优化，第二次课从代码逻辑走到模型和形式化。请大家先把心态摆正：这一章不需要你背很多名词解释，但需要你能拿着方法去设计用例。

> **板书：**中间写"第 3 章 软件测试方法"；左边写"第 1、2 章：为什么测 · 测什么（观念与概念）"，右边写"第 3 章：怎么测（方法与用例设计）"。

> **提问：**如果只给你两个小时测试一个从没见过的新系统，你会从哪儿下手？请说出你的第一步动作。

> **预设回答：**

- 答"先测登录、注册、下单这些最常用的功能"——很好，这正是后面要讲的 ALAC 思路：时间有限就把火力压到用户最常走的地方。

- 答"一个功能一个功能从头过一遍"——不现实。两小时连需求文档都读不完，这种答法缺的是优先级判断，不是勤奋。

- 答"先看代码，从代码里找问题"——可以，但前提是你有代码、看得懂、而且知道要看什么；如果面对的是第三方系统，这条路直接堵死。

- 追问收口：这三种答法其实对应三种不同的测试切入点，本章 3.1 到 3.7 节讲的就是怎么在这些切入点之间做取舍。

> **易错点 / 考点：**本页是封面页，一般不会直接出题，但口径要记准——课件写的是"第 3 章 软件测试基本方法"，教材写的是"第 3 章 软件测试方法"，同一章不同写法，答题以教材《软件测试方法和技术（第 4 版）》为准。另外要分清：本章讲的是方法（怎么设计用例），不是工具（用什么软件跑）；考试常把方法名和工具名混在一起做选择题。

> **过渡：**在打开本章地图之前，我们先把第 2 章的账结一下——因为第 3 章正好长在第 2 章的两个结论上面。

【3.2】第 2 章回顾（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是上一章的快速回看，我一条一条带你过，你对照自己的笔记检查有没有漏洞。
>
> 第一条，软件缺陷是软件质量的对立面。我们给缺陷下过定义：软件产品中不符合需求、或者虽然符合需求但用户不满意的地方，都可以算缺陷。记住这个"对立面"的提法，它意味着讨论缺陷时必须先有一个质量标准，没有标准就谈不上缺陷。
>
> 第二条，软件测试的分类可以从三个维度切：按层次分、按类型分、按方法分。这三个维度千万不能串线。
>
> 按层次分，是单元、集成、系统和验收四级——分的是被测对象有多大。按类型分，我们讲了静态测试与动态测试，分界线是程序有没有跑起来；讲了主动测试与被动测试，分界线是测试者有没有主动向被测对象施加输入；还讲了黑盒测试与白盒测试，分界线是你依据的是需求规格还是程序内部结构。按方法分，就是今天要展开的七个板块：基于直觉经验、基于输入域、基于组合及其优化、基于逻辑覆盖、基于缺陷模式、基于模型、形式化。

最后一条容易被同学们忽略：软件测试的工作范畴包括测试需求分析、测试计划与设计、测试执行、测试结果评估。请大家记住，测试不等于"执行用例"那一个动作，前面有分析和设计，后面还有评估。少了任何一环，测试就退化成了点鼠标。而在这四个环节里，需求分析和计划设计最省钱——在需求里改一句话，代价远远小于上线之后去改一行代码。

板书：缺陷＝质量的对立面 ｜ 三维度：层次（单元→集成→系统→验收）・类型（静／动・主动／被动・黑／白）・方法（今天展开） ｜ 工作范畴：需求分析 → 计划与设计 → 执行 → 结果评估。

提问：黑盒测试的依据是需求规格，白盒测试的依据是程序内部结构。那么验收测试有没有可能用白盒的方法？

预设回答：

- 答"不可能，验收测试当然是黑盒"——主流答案，方向对。追问一句为什么：因为验收测试面对的是客户视角的整套系统，看的是需求满足度，客户手里通常也没有源码。

- 答"如果客户方有能力看代码，也可以做白盒"——这是很好的补充，说明你已经理解黑盒白盒是按"依据"划分，而不是按"阶段"绑定的。

- 答"白盒就是单元测试"——典型混淆。把"层次"和"类型"两个维度绑死了。点评：层次按对象大小分，黑白盒按依据分；单元测试里照样可以用等价类划分（那是黑盒），系统测试里照样可以做代码级剖析（那是白盒）。

- 这个错误回答恰好是教学素材：请这位同学自己举一个"单元测试阶段用黑盒方法"的例子，他马上就会发现两个维度本来是独立的。

易错点 / 考点：三个维度不要串线。判断题高频："黑盒测试只能用于系统测试""静态测试就是评审""白盒测试比黑盒测试更高级"——前两句把维度绑死，第三句把手段当等级，都是错的。判断三步法：一问被测对象多大（层次），二问依据什么信息（类型），三问用例怎么设计（方法）。

过渡：账结完了，我们把第 3 章的全景地图打开。

【3.3】3.1 基于直觉和经验的方法（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本章的地图，七个板块，我念一遍：3.1 基于直觉和经验的方法；3.2 基于输入域的方法；3.3 基于组

合及其优化的方法；3.4 基于逻辑覆盖的方法；3.5 基于缺陷模式的测试；3.6 基于模型的测试；3.7 形式化测试方法。

请大家特别注意这个排列顺序，它不是随手排的。越往下走，对人的经验依赖越少，对模型和数学的依赖越多；越往下走，方法越严格、越容易被度量、也越容易被自动化。所以 3.1 到 3.7 这条线，本质上是一条"从依赖个人经验，走向可度量、可复用、可自动化"的演进路线。

课件的讲者备注还给了各节的页码：3.1 在 35 到 37 页，3.2 在 38 到 44 页，3.3 在 45 到 52 页，3.4 在 53 到 61 页，3.5 在 62 到 64 页，3.6 在 65 到 70 页，3.7 从 71 页开始。课后复习照这个页码定位，能省很多时间。

板书：竖排七行——3.1 直觉经验 → 3.2 输入域 → 3.3 组合优化 → 3.4 逻辑覆盖 → 3.5 缺陷模式 → 3.6 模型 → 3.7 形式化；右侧竖写一条主线：经验 → 可度量 · 可复用 · 可自动化。

提问：这七个板块里，你觉得哪一个最"靠人"，哪一个最"靠数学"？

预设回答：

- 答"3.1 最靠人，3.7 最靠数学"——正确。追问：那夹在中间的 3.4 呢？它靠的是代码结构，人和数学都不算主角，靠的是"可分析的产物"。

- 答"3.6 基于模型就是自动化测试"——常见错误。点评：模型是抽象，它带来的是"用例可以从模型里生成"，能不能自动化是随之而来的结果，不是它的定义。

- 答"越靠后的方法越好"——要纠正的价值观错误。方法没有高低，只有适用；越靠后的方法成本越高，用在不需要它的系统上就是浪费。

易错点 / 考点：本节地图的排序与归属是选择题高频；另一个坑是把"探索式测试"当成 3.1 的一种具体方法——教材认为它是一种测试方式，理由是在探索式测试里可以采用各种各样的方法。判断题常考："探索式测试属于基于直觉和经验的方法"——严格说教材不把它当方法，答题时按题目给的框架表述。

过渡：我们从最左边这一格开始，先看 3.1 基于直觉和经验的方法。

【3.4】方法论和具体方法（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

在钻进具体方法之前，我们先站高一层，把"方法论"和"具体方法"这两个词分开。这两个层次一分清，后面

很多争论就不会发生了。

从方法论这个层面看，它更多体现的是一种哲学思想，最典型的就是辩证统一。测试里有大量的对立统一体：静态测试和动态测试、白盒测试和黑盒测试、自动化测试和手工测试。请大家把"对立统一"四个字拆开理解——对立，是说它们看问题的角度正好相反；统一，是说它们服务于同一个目标，谁也替代不了谁。你不能说黑盒比白盒高级，也不能说自动化一定比手工强，它们是一枚硬币的两面。刚才有位同学答"越靠后的方法越好"，犯的就是这个毛病：把手段当成了等级。

第二句话更实在：软件测试的方法论来源于软件工程的方法论。有什么样的开发方法，就有什么样的测试方法跟它配套。有面向对象的开发方法，就有面向对象的测试方法；有敏捷方法，就有和敏捷方法对应的敏捷测试。为什么？因为开发方法决定了交付产物长什么样——面向对象开发产出的是类和对象，你的测试切入点自然也是类和对象；敏捷开发两周一个迭代，你的测试节奏就不可能还是"三个月后一次性系统测试"。说白了，测试方法不是凭空发明的，它是被开发方式逼出来的。

所以这一页留给大家一个判断动作：以后你听到任何一个新测试名词，先问一句——它是方法论层面的思想，还是具体到能落地写用例的方法？层次混了，讨论就会变成吵架。

板书：方法论（哲学 · 辩证统一）→ 对立统一体：静／动 · 黑／白 · 自动／手工 ｜ 具体方法（能直接落地写用例）｜ 来源关系：软件工程方法论 → 测试方法论（OO 开发 → OO 测试；敏捷 → 敏捷测试）。

提问："自动化测试一定优于手工测试"这句话错在哪里？

预设回答：

- 答"错在自动化也有成本"——对。追问他成本在哪：脚本开发与维护、环境与数据准备、一次性投入的回收周期，这些都是真金白银。

- 答"探索性、易用性这类测试自动化做不了"——更好的答案，已经说到方法论层面了：自动化擅长确定性校验，人的经验擅长发现意外。

- 答"没错，趋势就是全自动化"——典型错误，把手段当目标。把它当教学素材：请这位同学描述一个"自动化跑了一万条全绿、用户一上手就骂"的场景，他描述完自己就明白了对立统一的含义。

- 收口一句：对立统一不是和稀泥，而是提醒你，任何一侧的极端化都会带来新的风险。

易错点 / 考点：判断题两类。①把方法论的对立统一理解成"谁替代谁"——错，是互补；②把开发方法与测试方法的关系记反——不是测试方法决定开发方法，而是开发方式决定测试方式。口诀：开发怎么造，测试怎

么拆；造出类就测类，造出迭代就测迭代。

过渡：既然对立统一里最先被反复提到的就是黑盒和白盒，我们先把这两个"盒子"的边界划清楚。

【3.5】黑盒测试方法和白盒测试（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页用两张图把黑盒和白盒摆在一起，请大家跟着图看。

左边这个灰色的长方体就是黑盒。为什么叫黑盒？因为你看不见里面。这张图想告诉我们的是：同样一个"盒子"，看它的角度不同，测试的切入点就不同——进去的是数据、出来的是结果，而驱动这个盒子的，要么是客户需求，要么是事件。也就是说，黑盒测试只关心三件事：我喂进去什么、你吐出来什么、这两者是不是符合需求说好的关系。至于里面怎么算的，它一概不看。对应到黑盒，就分出两条路径：基于需求的测试，是照着需求文档一条条对；数据驱动测试，是拿不同的数据去喂。

而白盒那边画的是一个弯着腰、拿着放大镜盯流程图的人——功夫全下在程序内部结构上。白盒测试的依据是程序内部的结构：语句怎么走、判定怎么分叉、条件怎么组合。所以它又叫结构化测试，也叫逻辑驱动测试——"结构"和"逻辑"这两个词就是白盒的身份证。

一句话对照：黑盒看外面，白盒看里面；黑盒的依据是需求，白盒的依据是代码。再补一句最重要的：这两条路线不是二选一，而是同一个项目里的两道筛子——先用黑盒确认功能对不对，再用白盒确认代码里有没有藏着走不到的坑。

举个软件行业的实例。同样是测一个"购物车满 200 元减 30 元"的促销功能：黑盒的做法是拿不同金额的订单去试，199 元、200 元、201 元，看优惠有没有按需求触发；白盒的做法是去看判定语句里那个"大于"有没有被写成"大于等于"，条件之间的逻辑与有没有被写成逻辑或。两个视角发现的其实是两类不同的问题——黑盒发现"结果和需求不一致"，白盒发现"实现和设计不一致"。

板书：黑盒：输入 → 黑箱 → 输出；依据＝客户需求／事件驱动；别名＝基于需求的测试 · 数据驱动测试 ｜ 白盒：拿着放大镜看流程图内部；别名＝结构化测试 · 逻辑驱动测试 ｜ 一句话：外看需求，内看代码。

提问：同一个函数，我想知道"它算得对不对"和"它里面有没有永远走不到的代码"，分别该用哪一边的办法？

预设回答：

- 答"前者黑盒，后者白盒"——正确。请他把依据补全：前者依据需求规格，后者依据控制流。

- 答"都用白盒，白盒更彻底"——错误。点评：白盒看不到需求对不对。如果需求本身写错了，代码完全照着错误需求实现，你白盒覆盖率做到 100%，系统照样是错的。

- 答"都用黑盒"——同样偏。黑盒发现不了"这段代码永远走不到""这个判定写反了但结果凑巧对了"这类问题。

- 收口：所以两道筛子缺一不可，这也是为什么这一章后面既讲输入域方法，也讲逻辑覆盖。

> **易错点 / 考点：**最常见的是把"黑盒／白盒"和"手工／自动化"或"功能／性能"混为一谈。判断口径只有一条：看你的依据是需求还是代码结构。选择题常给四个活动问哪个属于白盒；判断题常把"界面测试就等于黑盒测试"绝对化——界面测试通常依据需求，但也要看具体做法，别把话说死。

> **过渡：**那过去大家习惯的黑盒白盒分法，具体能装进哪些方法？看下一页这张分类图。

【3.6】过去常提"黑盒和白盒"方法（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这张图是过去教材里最常见的分法：先分白盒和黑盒，再把具体方法往里塞。
>
> 左边走白盒这条路，落到"结构化测试方法"，装进去六个覆盖标准：语句覆盖、判定覆盖、条件覆盖、判定条件覆盖、条件组合覆盖、基本路径覆盖。这六个是 3.4 节的主角，它们的共同点是都要对着代码的控制流说话。
>
> 右边走黑盒这条路，落到"基于需求的测试方法"，装进去七个：等价类划分法、边界值分析法、判定表方法、因果图法、Pairwise 方法、正交试验法、功能图法。它们的共同点是都从需求、从输入输出规格出发。
>
> 这张图很好用，但这一章后面我们还要告诉大家：这种两分法已经不够用了。为什么？因为像基于缺陷模式的测试、基于模型的测试、形式化的方法，你硬塞进黑盒或白盒都很别扭。所以下一页我们换一个更现代、也被国际知识体系承认的框架。
>
> 这两列的区别，用一句话就能记住：白盒那一列问的是"代码里的每条路走过了没有"，黑盒那一列问的是"需求里的每条规则对过了没有"。再举个例子，一个"满 18 岁才能注册"的功能：黑盒会去试 17 岁、18 岁、19 岁，看注册按钮该亮还是该灰；白盒会去看那个年龄比较的分支有没有被两种结果都执行到。左边管功能对不对，右边管代码走没走全，两者都要，谁也代替不了谁——这两列合起来，就是过去所有测试方法的家底。

> **板书：**白盒方法 → 结构化测试方法：语句 · 判定 · 条件 · 判定条件 · 条件组合 · 基本路径覆盖 ｜ 黑盒方法 → 基于需求的测试方法：等价类 · 边界值 · 判定表 · 因果图 · Pairwise · 正交试验 · 功能图。

> **提问：**黑盒这一列里，哪几个方法的共同点是"处理多个输入条件的组合"？

> **预设回答：**

- 答"判定表、因果图、Pairwise、正交试验"——正确。追问：那等价类和边界值处理的是几个变量？答案是单个变量的取值域。

- 答"全部都是组合"——混淆了。等价类和边界值是单变量思路，功能图是从功能出发建模，都不是组合问题。

- 答"判定表其实是白盒"——错误。判定表依据的是输入条件的组合规则，属于黑盒侧。

> **易错点 / 考点：**两个坑。①功能图法在这张图上放在黑盒列，但教材后文把因果图、功能图归为基于模型的方法——归属会随分类框架变化，考场上以题目给的框架为准。②左边六个覆盖标准必须背全，漏一个就丢分。口诀："语句、判定、条件、判定条件、条件组合、基本路径"——三层递进，最后加路径。

> **过渡：**那么有没有一个更完整、也更被国际标准承认的分类框架？有，就在下一页。

【3.7】方法论：测试方法（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把方法论层面的测试方法又列了一遍，五条：上下文驱动方法、基于需求验证的方法、基于场景的测试方法、基于模型的方法、基于经验的方法。
>
> 请注意，这五条和上一页的黑盒白盒不是同一层的东西。它们回答的是"我根据什么来决定测什么"。上下文驱动，是说测试策略跟着项目上下文走——同样是电商系统，银行系的支付模块和初创公司的商品列表，测法不可能一样；基于需求验证，是拿需求文档当标尺；基于场景，是把用户真实的业务链条串起来完整走一遍；基于模型，是先把系统抽象成状态机、判定表这类模型，再从模型生成用例；基于经验，就是靠人。
>
> 右边这三只互相咬合的齿轮是一张配图，它想表达的意思是：测试能力不是靠单一方法堆出来的，而是基础、服务、产品一层带一层转起来的。这跟我们这门课的立场一致——讲方法，最终是为了让你在具体项目里把几种方法配起来用。
>
> 再打个生活的比方。这五条方法论就像你决定"今天去哪儿吃饭"的五种依据：看今天是什么场合（上下文驱

动），照着昨晚列好的清单执行（基于需求验证），按一条完整的动线走完（基于场景），先画一张地图再挑路线（基于模型），或者干脆凭经验去常去的那一家（基于经验）。手段不同，目标却是同一个——把肚子填饱。测试也一样，五条路通向同一个目标：以可接受的成本，尽早发现更多、更严重的缺陷。

所以这一页先埋一个伏笔：本章最后我们要讲的，不是"哪个方法最好"，而是"方法怎么选、怎么组合"。

板书：方法论五条：上下文驱动 · 基于需求验证 · 基于场景 · 基于模型 · 基于经验 | 它们回答的问题：我根据什么决定测什么 | 配图寓意：能力分层咬合，不是单点堆方法 | 伏笔：方法的选择与组合。

提问：一个手机银行 App 要测"转账"，上面这五条里至少能同时用上哪几条？

预设回答：

- 答"基于需求验证＋基于场景"——很好。需求对规则，比如单笔限额、手续费、跨行到账时间；场景走完整链路，选卡、输金额、验证、成功或失败。

- 答"还要加基于经验"——对。老测试员会想到：中途断网、转账时切到后台、重复提交、金额输成负数，这些正是经验的用武之地。

- 答"用上下文驱动就够了"——错误。它不是一种具体的测法，而是"先判断项目处境、再决定用哪些方法"的元策略，不能拿它当用例设计手段。

易错点 / 考点：最大的坑是把"上下文驱动"当成一种具体测试技术。判断三步法：它是否在回答"根据什么决定测什么"——是，则属于方法论层；它是否能直接产出一条可执行用例——能，则属于具体方法。选择题常问"下列哪一项属于方法论层面的测试方法"。

过渡：好，把三张框架图放在一起，我们用国际知识体系的口径再对一次表。

【3.8】SWBOK 对测试方法的分类（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是本节的定盘星，请大家把这张表抄进笔记本，它后面会被反复引用。

表格六行，每一行都是一个"基于什么"的测试类别。

第一行，基于输入域的测试，缩写 I-D-B-T，装的是等价类、边界值、两两组合也就是 pairwise、随机测试，它对应的传统口径是黑盒测试。

第二行，基于代码的测试，缩写 C-B-T，装的是基于控制流的标准、基于数据流的标准、CBT 参考模型，对应白盒测试。

第三行，基于故障模式的测试，缩写 F-B-T，装的是故障模型、错误猜测法、变异测试。请大家注意有一类后面是空的——它既不算纯黑盒，也不算纯白盒，这正是旧的两分法装不下的东西。

第四行，基于使用的测试，缩写 U-B-T，装的是操作配置（operational profile）和用户观察启发，对应黑盒测试。

第五行，基于模型的测试，缩写 M-B-T，装的是决策表、有限状态机、形式化验证、TTCN3、工作流模型。

第六行，基于应用特性的测试，缩写 T-B-N-A，装的是面向对象、Web、实时、SOA、嵌入式、安全攸关这些内容，右边标注的是"应用领域"——这四个字很关键，意思是它根本不算一种测试方法，而是各个应用领域里怎么把前面那些方法用起来。

这张表还有一个特别值得琢磨的细节：错误猜测法在这里被放在基于故障模式，而我们教材把它归到基于直觉和经验。两种归法都成立，因为分类框架不同。这就是我一开始提醒大家的：别把归属当成真理，要看题目用的是哪一套框架。

板书：SWBOK 六类：输入域 IDBT → 代码 CBT → 故障模式 FBT → 使用 UBT → 模型 MBT → 应用特性 TBNA ｜ 右侧标注：黑盒 · 白盒 · （空）· 黑盒 · （空）· 应用领域 ｜ 记忆：输入、代码、故障、使用、模型、应用。

提问："错误猜测法"在这张表里属于哪一类？在我们教材里又属于哪一类？为什么可以不一样？

预设回答：

- 答"表里在基于故障模式，教材在基于直觉和经验"——很好，说明两个框架都看懂了。追问：差异的根源在哪？在于划分依据不同——SWBOK 看它"利用已知缺陷模式去猜"，教材看它"靠人的经验去猜"。

- 答"一定有一个是错的"——典型错误。点评：分类是工具，不是真理，取决于划分依据。把它变成教学素材：考试时以题干给出的框架为准，不要把教材口径硬套到 SWBOK 表上。

- 答"属于基于输入域"——错误。错误猜测法不靠划分输入域，它靠经验直接猜，这跟等价类完全是两条路。

易错点 / 考点：三个高频点。①六个英文缩写的对应关系是选择题常客；② TBNA 不是方法，而是应用领域；③"随机测试"被 SWBOK 归入基于输入域，而不是基于直觉和经验——这一条最反直觉。另外提醒一处书写差异：课件缩写为 SWBOK，教材原文写作 SWEBOK 3.0，指的是软件工程知识体系（Software Engineering

Body of Knowledge），答题以教材口径为准。口诀：输入代码故障，使用模型应用。

【思政落点 1】讲一个关于知识体系与话语权的故事：SWEBOK 是国际上通行的软件工程知识体系，我们先把它学透，是为了掌握这门行当的共同语言。值得同学们想一想的是，今天中国的软件规模和应用复杂度已经走在世界前列——从移动支付到全民级的电商大促，这些系统对测试提出的要求，很多在国际教材里找不到现成答案。知识点联系：分类框架是别人总结的，但框架背后的经验要靠我们自己的工程实践去补充。落到价值观：先把别人的知识体系吃透，才有资格把我们在超大用户量、超高并发场景里积累的测试经验写成标准、讲给世界听。工程自信不是喊出来的，是测出来的。

过渡：六个大类立住了，接下来我们逐类展开。第一类，也是看上去最不像"方法"的一类——基于直觉和经验的方法。

【3.9】3.1 基于直觉和经验的方法（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

我们进入 3.1，基于直觉和经验的方法。这一节有两个小节：3.1.1 讲 Ad-hoc 测试方法和 ALAC 测试，3.1.2 讲错误推测法。

先讲一句可能让同学们意外的话。教材认为，基于直觉和经验的测试方法不是严格意义上的科学测试方法，带有一定的随机性，测试结果不够可靠，甚至可以看作是没有办法的办法。那为什么还要花一整节讲它？因为软件测试具有社会性，呈现一定的不确定性。有些东西很难靠一种明确的科学方法测完——比如产品好不好用、用户体验顺不顺，很大程度上仍然依赖测试人的直觉和经验。

还有一个概念必须在这里澄清：业界很流行的"探索式测试"，教材认为它是一种测试方式，而不是一种方法。理由很简单——在探索式测试里面，你可以采用各种各样的测试方法。方式和方法，层次完全不同。

板书：3.1 ＝ 3.1.1 Ad-hoc／ALAC ＋ 3.1.2 错误推测法 ｜ 定位：不是严格意义上的科学方法，带随机性、结果不够可靠 ｜ 为什么还要讲：软件测试具有社会性 ｜ 探索式测试＝一种测试方式，不是一种方法。

提问：为什么教材说"探索式测试是一种测试方式，而不是一种方法"？方式和方法差在哪里？

预设回答：

- 答"因为探索式测试里面可以用各种方法"——正确。追问他举个例子：你在探索的过程中，照样可以随时用上

等价类划分和边界值。

- 答"因为探索式测试不用写用例"——误区。探索式测试不预先写用例，但它一直在做设计，而且要求记录过程。

- 答"方式和方法的差别不重要"——要纠正。术语的层次错了，整句判断就会错，考试里这就是一道判断题的命门。

> **易错点 / 考点：**判断题高频。"探索式测试属于 3.1 基于直觉和经验的方法"——严格按教材口径，它不被认为是一种方法；"自由测试（Ad-hoc）等同于探索式测试"——不等价，Ad-hoc 更随意、更没有结构。判断三步法：它是一个具体可复用的测法，还是一个能容纳多种测法的做事方式？

> **过渡：**先看 3.1.1，从 ALAC 和随机测试说起。

【3.10】3.1.1 ALAC 测试和随机测试（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> ALAC 是 Act-like-a-customer 的简写，翻译过来就是"像客户那样做"。它是一种基于客户使用产品的知识开发出来的测试方法——依据不是需求文档，而是"客户平时到底怎么用这个产品"。
>
> 它的出发点是著名的 Pareto 80/20 规律。教材把这条规律落到测试上，概括为两条：一是全部功能中 20% 是常用功能，用户 80% 的时间都花在这 20% 上，剩下 80% 的功能只有 20% 的时间被用到；二是测试发现的错误里，80% 很可能集中在 20% 的程序模块中，另外 20% 的错误集中在剩下的 80% 模块里。
>
> 看这张原理图。外面这个圆圈代表整个系统；里面那块浅色椭圆标注着"20% 最常用的部分"，说明是"有针对性的测试，测试的重点"；其余部分的说明则是"虽然更大范围也存在缺陷，但不是测试的重点"。这张图要说的就是一句话：测试时间是有限的，火力必须往用户最常走的那条路上压。
>
> ALAC 适合什么场合？教材列了几种：产品只是演示版、预算很低、时间不够、开发日程表很紧。它最大的益处是降低测试成本、缩短测试时间，而且缺陷的查找和改正会针对客户最容易遇到的错误。从这个角度看，ALAC 测试和基于风险的测试方法很接近。
>
> 那标题里的"随机测试"呢？教材在 3.1.1 里对应的概念是自由测试（Ad-hoc Testing）——强调测试人员根据自己的经验，不受测试用例的束缚，放开思路、灵活地进行各种测试；这类测试是严格测试方法的补充手段，思维活跃，常能发现隐藏很深的缺陷。但要提醒一个分类细节：在 SWBOK 那张表里，随机测试被归到基于输入域的测试，而不是基于直觉和经验。同一个词在不同框架里归属不同，答题以题目给的框架为准。

> **板书：**ALAC = Act-like-a-customer（像客户那样做）｜依据：客户使用产品的知识｜出发点：Pareto 80/20（20% 功能→80% 使用时间；80% 缺陷→20% 模块）｜适用：演示版 · 预算低 · 时间紧 · 日程紧｜近亲：基于风险的测试｜随机测试：SWBOK 归"输入域"。

> **提问：**一个团队只有三天时间测一个电商 App，产品经理说"每个页面都要点到"。你会怎么说服他把火力集中起来？

> **预设回答：**

- 答"先测下单和支付主链路"——标准答案。追问依据：这两条是 80% 用户 80% 时间在做的事，也是失败代价最大的事。

- 答"三天不够，要求加时间"——部分合理，但没解决问题。点评：ALAC 的前提就是时间紧、预算低，先把最常用的 20% 测扎实，再谈加人加时间。

- 答"那就随机点，点到哪儿算哪儿"——典型错误，而且正好踩在本页的混淆点上。点评：那不是 ALAC，那正是教材批评的漫无边际的随机测试。ALAC 的"像客户那样做"是有依据、有重点的，不是没有策略。

- 追问收口：如果这个 App 的核心功能是"查余额"而不是"下单"，你的重点会变吗？——会。重点取决于用户的实际使用分布，这也说明了为什么 ALAC 必须建立在客户使用知识之上。

> **易错点 / 考点：**①把 ALAC 的 80/20 记成"要测 80% 的功能"——正好反了，重点压在 20% 的常用功能上。②把 ALAC 等同于随机测试——依据不同：前者靠客户使用知识，后者靠随机取值。③把 ALAC 等同于基于风险的测试——教材说的是"接近"，不是"等于"：ALAC 手里那把尺子主要是使用频率，风险还要额外考虑后果的严重程度。口诀：客户怎么用，我就怎么测；二八分火力，风险是一家。

> **过渡：**ALAC 给的是"往哪儿压火力"的思路，但它依赖的仍然是知识，而不是一项可以照着做的规则。下一页，我们看一种更依赖个人经验的方法——错误推测法。

【3.11】3.1.2 错误猜测法（6 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页讲错误推测法，也叫错误猜测法。教材给的定义是：测试者根据经验、知识和直觉来发现软件错误，推测程序中可能存在的各种错误，从而有针对性地进行测试。有一句老话很贴切——"只可意会，不可言传"，说的就是这类方法的味道。

它背后有一个思想值得记住：某处发现了缺陷，则可能隐藏更多的缺陷。说白了，看见一只虫子往往意味着一窝。所以操作时先把可能出现的错误和容易出错的地方列出来，再由测试者凭经验做选择。

课件给了四种"程序经常出现的错误"的来源，请记牢：第一，单元测试中发现的模块错误；第二，产品以前版本曾经发现的错误；第三，输入数据为 0 或者字符为空；第四，软件要求输入时，用户不是没输对，而是根本没有输入任何内容，单单按了 Enter 键。

教材还补了几个方向：特殊字符（星号、百分号、斜杠、井号、@、美元符号等）输入系统后产生的例外情况；把多个边界值组合起来测也可能让程序出错；客户端正常连接没问题，可以试试断掉连接看它能否重连；还有空指针、内存没及时释放、session 失效、JavaScript 字符转义这些程序里的老面孔。

接下来是这一页最重要的评价：它能充分发挥人的直觉和经验，在小组中集思广益，方便实用，特别在测试基础较差时组织小组做错误猜测很有效。但它不是一个系统的测试方法，所以只能用作辅助手段——先用系统的测试方法设计用例，再用它补充一些额外的例子。

优缺点同样鲜明：优点是能快速容易地切入，还能体会到程序易用与否；缺点是难以知道覆盖率，可能丢失大量未知的区域，而且带有主观性、难以复制。最后补一个分类伏笔：SWEBOK 把它归到基于故障模式的测试，我们教材认为它更适合归到基于直觉和经验。

> **板书：**错误推测法 = 经验 + 知识 + 直觉 → 有针对性地测 ｜ 背后思想：一处有缺陷，可能藏更多 ｜ 四类常见来源：单元测试的模块错误 · 历史版本的缺陷 · 输入 0／字符为空 · 未输入直接按 Enter ｜ 定位：辅助手段（先用系统方法，再用它补） ｜ 缺点：覆盖率不可知 · 主观 · 难复制。

提问：我们做一个"用户注册"页面，只准你用错误推测法，你能列出五条最可能出问题的地方吗？

预设回答：

- 答"用户名为空直接点注册、密码全是空格、手机号输入 11 位以外的数字、两次密码不一致、什么都不填直接按 Enter"——很好，其中几条正好对上课件里的"输入 0／字符为空／未输入直接回车"，说明抓住了要点。

- 答"把所有输入框都用边界值跑一遍"——跑题了。点评：那是 3.2 节的系统化方法，不是推测法。推测法的价值在于想别人想不到的，比如用特殊字符当用户名、注册到一半断网、重复提交。

- 答"我不知道从哪儿猜"——很正常。给他抓手：先翻上一个版本的缺陷列表，再看这个模块的开发人员最容易在哪儿犯错，最后想"用户会怎么乱来"。

易错点 / 考点：①最常考的判断题："错误推测法可以作为主要的测试方法"——错，它只能作辅助手段。

②"错误推测法等于随机乱测"——错，它是"有依据地猜"，依据是经验、历史缺陷和已知的易错点。③归属有两个答案：SWEBOK 归 FBT，教材归基于直觉和经验，答题看题干给的框架。判断三步法：有没有依据？依据是不是个人经验？能不能说清覆盖率？——靠经验、说不清覆盖率，就是它。

【思政落点 2】请大家注意课件里的第二条来源——"产品的以前版本曾经发现的错误"。这句话的分量在于：同一个人在同一个坑里摔第二次可以原谅，但一个团队在同一个坑里摔第二次，就是管理问题。知识点联系：错误推测法靠的是个人经验，而个人经验会随着人员流动而蒸发，所以成熟的团队会把每次错误猜测的结果沉淀成检查单、缺陷模式和回归用例，让老测试员的直觉变成新人能直接调用的资产。落到价值观：知识共享不是情怀，是职业操守——你的经验不只属于你个人，把它写下来、传下去，团队的整体质量水位才会抬起来。

过渡：3.1 讲完了，它最大的软肋就是"覆盖率说不清"。要解决这个问题，就得把输入域本身拆开来看——我们进入 3.2 基于输入域的方法。

【3.12】3.2 基于输入域的方法（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

我们进入 3.2，基于输入域的方法。这一节两个小节：3.2.1 等价类划分法，3.2.2 边界值分析法。

先说立足点。教材讲得很朴素：软件系统本质上就是对输入数据的处理，并转化成所期望的结果。所以用数据来验证系统的功能性，是很自然的思路。不论被测对象是整个系统、一个模块还是一个函数，它总有数据输入或参数调用——喂进不同的数据，看输出的数据对不对，这类方法都归为基于输入域的方法。

这里有一个归属上的细节，考试容易绕：决策表、因果图、两两组合、正交实验法其实也可以归到广义的输入域验证，因为它们测的是多变量组合的数据。但在本教材里，决策表、Pairwise、正交法被划到组合方法，只有因果图、功能图归到基于模型的方法，我们按教材口径走。

换句话说，这一节只讲两件兵器：一件负责把输入域切开，一件负责在切口上取点。

板书：3.2 ＝ 3.2.1 等价类划分 ＋ 3.2.2 边界值分析 ｜ 立足点：软件＝对输入数据的处理 → 用数据验证功能 ｜ 归属口径（本教材）：单变量 → 输入域；决策表／Pairwise／正交 → 组合；因果图／功能图 → 基于模型 ｜ 本节两件兵器：切开输入域 · 在切口上取点。

提问：既然叫"基于输入域"，那如果被测对象是一个没有参数的函数，这个方法是不是就用不上了？

预设回答：

- 答"用不上"——典型错误。点评：输入域不只包括函数参数，还包括全局变量、配置项、外部读入的文件、接口返回的数据。只要它是"喂进去的东西"，就有输入域。

- 答"还能测，比如它读的配置文件"——正确，这就是把输入域的概念理解宽了。

- 答"还可以从输出域反推"——更完整。教材明确说，确定输入等价类时通常要分析输出数据的等价类，再据此导出对应的输入数据等价类。

- 收口一句：输入域的边界不是代码里的参数表，而是"一切影响结果的外部数据"。

易错点 / 考点：①"基于输入域"不等于"只看函数参数"，等价类划分既要划输入也要看输出。②判断题常考"决策表法属于基于输入域的方法"——按本教材口径归组合方法，答题以教材为准。③别把 3.2 和 3.3 的顺序记反：3.2 解决单变量，3.3 解决多变量组合。

过渡：先看 3.2.1，等价类划分法。

【3.13】3.2.1 等价类划分方法（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

等价类划分法是本章第一个系统化方法，请抓住三个关键词：子集、等效、代表值。

先看定义：等价类是某个输入域的子集，子集中每个输入数据的作用是等效的。做法就是把输入数据分成若干等价类，从每个等价类里选一个代表性数据作为测试用例。等价类又分两类：有效等价类，也就是合理的数据；无效等价类，也就是异常的数据。

有同学会问：凭什么一个值能代表一整个类？教材的答案是——如果代表值作测试用例没发现错误，这个类里其他数据也不会发现错误。举教材的例子：测实数开方程序，实数域分成正实数、负实数和 0，各取一个代表值就够。

为什么非要这么做？因为穷举根本做不到。教材算过一笔账：两个整数输入量在 32 位机器上，测试数据最大可能是 2 的 64 次方组；跑一组要 1 毫秒，不停歇跑完也要 5 亿年。所以只能选少量有代表性的数据，用较小代价暴露较多缺陷——这正是等价类划分要解决的问题。

划分过程一般经过两步：第一步分类，把输入域按相同特性或类似功能划分；第二步抽象，在各子类中抽象

出相同特性并用实例表征。看右边这张图：all inputs 被绿线切成几块，橙色椭圆里的 i1 到 i4 就是划分出的等价类——块内均匀，说明块内数据在揭露错误的能力上等效。

有效类和无效类怎么用，这个规则常考：对有效等价类，一条用例要尽可能多地覆盖尚未被覆盖的类，重复直到全部覆盖；对无效等价类则设计若干条用例分别覆盖。注意，一条用例通常只覆盖一个无效等价类——因为程序碰上两个非法值很可能只报第一个错就退出，第二个错有没有被处理，你根本看不出来。

最后是评价：优点是能用较少的用例实现完整覆盖，减少了重复性；缺点是缺乏特殊用例的考虑，还要有深入的系统知识才能选出有效的数据。

板书：等价类 = 输入域的子集（类内等效）→ 每类取一个代表值 | 两类：有效（合理数据）· 无效（异常数据） | 为什么：穷举 $2^{32}\times2^{32}=2^{64}$ 组，需 5 亿年（教材数据） | 两步：分类 → 抽象 | 覆盖规则：有效类尽量合并，无效类一条一个 | 优点：用例少、覆盖全；缺点：缺特殊用例、依赖系统知识。

提问：一个输入框要求填"手机号"。请你划出等价类，并说明哪几个无效等价类必须一条用例一个。

预设回答：

- 答"有效类是 11 位数字手机号；无效类是长度不对、含非数字、为空、号段不存在"——结构正确，和课件口径一致。追问：有效类能不能只留一条用例？可以，但要覆盖所有有效等价类。

- 答"把所有无效情况放进一条用例，省时间"——典型错误，也是案例题的高频扣分点。点评：这条用例只能证明程序对第一个错有反应，后面的错全被掩盖了。请这位同学自己数一数，他列的四个无效类合并成一条之后，有三个再也没有被验证过。

- 答"只测有效类就行，正常用户不会乱输"——严重误区。点评：无效等价类测的是容错性，在金融类系统里，非法输入本身就是攻击面的一部分。

易错点 / 考点：①把等价类划分和边界值分析混为一谈——等价类解决"划几块"，边界值解决"在块的边上取哪几个点"，两者是补充关系，常配合使用。②无效等价类必须逐条设计用例，这是案例题的主要扣分点。③"等价"指的是揭露错误的能力等效，不是数值相等或写法相似。判断三步法：看去重（块内是否去重）→ 看覆盖（有效类能否合并、无效类是否单条）→ 看补充（边界值补上了没有）。

【思政落点 3】刚才那笔账——2 的 64 次方组数据、5 亿年——把一个工程现实摆在所有人面前：资源永远是有限的，追求绝对完备的测试是不可能的。知识点联系：等价类划分教给我们的态度，正是承认约束之后做出的理性取舍——不空喊"零缺陷"，而是把有限的资源投到最有代表性的地方去。落到价值观：这在金融行业尤其重要，风控、清算、支付这些系统的测试预算同样有限，一个测试工程师最值钱的能力不是承诺测

全，而是能说清楚"哪些测了、哪些没测、为什么先测这些"，这就是实事求是、量力而行。

> **过渡：**等价类划分解决了"划几块"，但它故意忽略了一件事——块的边上。而恰恰是这些边上最容易出错，下一页我们就看划分原则怎么落到几种具体情形上。

【3.14】等价类划分方法应用示例（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

上一页我们讲了等价类划分的基本思想，这一页看它怎么落到具体的输入条件上。课件把常见情形归纳成两句：第一句，输入条件规定了取值范围或者值的个数，就可以划分成一个有效等价类和两个无效等价类；第二句，输入条件规定了输入值的集合，也就是规定了"必须怎样"，那么就是一个有效等价类加一个无效等价类。最后还留了一句"N 个条件……"，意思是条件一多，等价类就按每个条件分别铺开，再分别取代表值。

右边三张图很直观。上面一张是范围型：中间那段是 in range，即有效等价类，左右两边 less than range 和 greater than range 是两个无效等价类。中间一张是单值型：只有 value 这一个点是有效的，小于它的 less than value、大于它的 greater than value 都属于无效。下面一张是集合型：member of set 是有效等价类，外面那一圈 not member of set 是无效等价类。

为什么这个分类值得记？因为等价类的个数直接决定用例条数。教材的报表日期例子最典型：三条有效等价类合成一条用例，输入 201006 就全覆盖了；无效等价类要分别输入 200aob、20102、1012012、198802、203011、200000、202013 这七个非法数据，加起来八条数据就把日期输入测完了；不用等价类划分，输入值则要几百个。所以等价类划分的功劳是：把漫无边际的随机测试变成了有针对性的测试，用少量代表值实现合理覆盖。

最后注意"多条件"这一层——现实中的输入几乎都不是单一条件，比如日期同时受"6 位数字字符""年份 2000 到 2020""月份 1 到 12"三条规则约束，这三条要分别划类、分别取代表值。

> **板书：**范围型 → 1 有效（in range）＋2 无效（小于／大于）｜单值型 → 1 有效（value）＋1 无效（≠value）｜集合型 → member of set（有效）／not member of set（无效）｜N 个条件 → 逐条划类、分别取代表值｜报表日期：有效 1 条（201006）＋无效 7 条＝8 条数据

> **提问：**既然等价类可以有很多个，为什么我们说"有效等价类可以一条用例多覆盖几个，无效等价类必须一种错误一条用例"？

> **预设回答：**

- 答"有效类可以合起来测"：✅ 方向对。追问：为什么无效类不能也合起来测？——因为程序遇到第一个不合法输入往往就直接报错返回了，后面那几种错误根本没机会被检查到，失败时也分不清是哪条规则被违反。

- 答"两类没有区别"：❌ 回到报表日期的例子。200aob 是"含非数字字符"，20102 是"少于 6 个数字字符"，写成一条用例，报错了你只知道"有问题"，查不出问题出在哪条规则上。

- 答"划分要看完备性和互斥性"：✅✅ 这正是要抓的两条原则——完备性要求所有可能的输入都归入某一类，一个不能漏；互斥性要求同一个输入不能同时属于两类，不能重叠。

> **易错点 / 考点：**第一个坑是把"范围型"的无效等价类只写一个。范围型一定是两个无效类：小于下界的和大于上界的。教材第 46 页用 10<x<100 举例，明确写出有效类 10<x<100、无效类 x<10 和 x>100，这是判断题和填空题的高频考点。第二个坑是学生对着多规则需求只划出一个有效类。第三个坑是分不清"类型"和"范围"——同一段需求里两种条件都要划。判断三步法：先识别条件类型（范围／单值／集合／布尔／枚举）→ 再按类型套"几个有效类、几个无效类"→ 最后用完备性和互斥性自查一遍。

> **【思政落点 4】**等价类划分的完备性要求"所有可能的输入都必须归入某一类，一个都不能漏"，互斥性要求"同一个输入不能同时属于两类"。这两条像极了做事要有边界、有规矩：分工要清楚、责任要分明，既不能留下没人管的空白，也不能两拨人抢同一件事。把它练成习惯，将来写用例、定接口契约、做任务分工，都会少很多扯皮和返工。

> **过渡：**等价类划分做到了把输入域切成几块，可它只告诉我们"哪一块"，没有告诉我们"一块里面哪一点点最危险"。经验告诉我们，翻车的地方常常不在块的中间，而在块的边缘。下一页我们就把镜头推到边缘上去，讲边界值分析方法。

【3.15】练习（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是当堂练习。题目给的是一个计算器的界面，请你想一想：对这样一个计算器，有效等价类怎么划分？无效等价类又怎么划分？课件右边给了四条参考划分：等价类 1 是 Integer 整数，等价类 2 是 Decimal fraction 小数，等价类 3 是 Negative 负数，等价类 4 是 Invalid input 非法输入。
>
> 先说要求，再说时间。我给三分钟独立做题；做完用两分钟和同桌交换、专门挑对方的漏；最后我们用两分钟一起对答案，请同学站起来讲你的划分理由。做的时候按刚才那套动作走：第一步判断计算器的输入条件属于哪种类型——它的输入域是数值，既带类型性质又带范围性质；第二步把每一类都写明是有效还是无效，

并且给每一类配一个代表值。请大家不要只写"整数、小数"四个字，要写成"有效等价类 1：整数，代表值 42"这样的格式，这才叫用例设计，才是能交给别人执行的成果。

还有一句提醒：课件给的四条是"一类一格"的参考口径，你们完全可以划得比它更细——比如小数可以再分正小数和负小数，非法输入可以再分"非数字字符""超出显示位数""除数为零"。今天真正要过关的不是数出几格，而是你能不能说明白：你为什么要这么切？你的切法满足不满足完备性和互斥性？先自己做，我一会儿点评。

板书：练习：计算器输入域 → 有效／无效等价类？ ｜提示：先判类型 → 再分有效无效 → 每类给代表值｜书写格式：编号＋名称＋有效/无效＋代表值

提问：请一位同学回答——"负数"在这道题里是有效等价类还是无效等价类？你的判断依据是什么？

预设回答：

- 答"有效，因为计算器本来就能算负数"：✅ 正确。依据是规格说明，不是我们的直觉：对计算器来说负数是合理、有意义的输入，属于有效等价类。这里 Integer、Decimal fraction、Negative 三类都属于有效输入，各取一个代表值。

- 答"无效，因为负数是负数"：❌ 典型误判。点评：判断有效还是无效，唯一依据是"满足不满足规格说明、对程序有没有意义"，跟数值正负没有关系。

- 补充追问"那什么是无效等价类"：无效等价类是不合理、无意义的输入集合，也就是题里的 Invalid input——非数字字符、空输入、超出范围的值等，它检验的是程序的异常处理和容错能力。

- 追问"划分原则是什么"：两条——完备性（所有可能输入都要归入某一类，不能漏）与互斥性（同一输入不能同时属于两类，不能重叠）。经验做法是：有效等价类按"合法区间／合法类型"划分，可以有几个；无效等价类按"每一种错误各算一类"划分，通常会更多。

- 迁移提醒：这条原则放到表单规则上更好理解。像后面练习题里 Username 的"不能为空""只能用字母数字下划线连字符""不能用空格""必须以字母开头"，每一条都要各算一个独立的无效等价类——一个无效类只覆盖一种错误，否则失败时分不清是哪条规则被违反了。

易错点 / 考点：最常见的错是把"无效等价类"写成"所有非法输入合成一类"，也就是把题里的 Invalid input 只写一格。考试里这么答要丢分：一个无效类只覆盖一种错误，合起来写就失去了定位能力。第二个坑是只划有效类、不划无效类，或者把两者写在同一列里，违反互斥性。考点通常以案例题出现——给一段需求，要求列出有效／无效等价类表并编号，再各取一个代表值。口诀：有效看规格、无效看错误，一条错误一类。

> **过渡：**等价类把输入域切成了块，也给了每块的代表值。但真实项目里的翻车点，往往不在块的中央。下一页我们把目光移到每一块的边缘，讲边界值分析方法。

【3.16】3.2.2 边界值分析方法（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页正式开始第 3.2.2 节，边界值分析方法，英文缩写 BVA，Boundary Value Analysis。先记它的一句话结论：很多错误发生在输入或者输出范围的边界上，因此针对各种边界情况设置测试用例，可以更有效地发现缺陷。请注意这句话里的两个关键词，一个是"边界"，一个是"更有效"——它没有说边界值能替代等价类，而是说同样的力气花在边界上，命中缺陷的概率更高。

为什么边界这么邪门？教材给的解释是：实践证明，程序往往在输入输出的边界值情况下发生错误。回到上一页的报表日期例子，等价类划分法恰恰漏掉了边界——它只告诉我们 2000 到 2020 这一段是有意义的，却没有专门去试 200001 这个有效最小值、202012 这个有效最大值，也没有试 199901、199912、202101、202112 这些刚好越界的无效值。而程序往往就在这些地方出错：循环少跑一次、比较符号该写大于等于却写成大于、数值截断与进位处理不当，全都发生在边界上。

具体怎么设计？课件给了两个动作。第一步，确定边界情况，也就是输入或者输出等价类的边界；第二步，选取正好等于、刚刚大于或者刚刚小于边界值的数据作为测试数据。说白了就是三档：压在界上、界内一步、界外一步。三档各来一条，边界就被钉住了。

最后提醒一句它的边界——注意这个词用在这里正好。边界值分析法也有管不着的地方：教材明确说，它对布尔值或者逻辑变量无效，因为布尔的取值只有真和假，没有"刚好"和"差一点"；它也不能很好地测试不同的输入组合。所以它的正确定位是等价类划分法的一种补充，两者结合起来使用才更有效。

> **板书：**BVA＝Boundary Value Analysis｜结论：错误多发生在输入/输出范围的边界上｜设计两步：①确定边界情况（输入或输出等价类的边界）②取正好等于、刚刚大于、刚刚小于边界值的数据｜三档：压界／界内一步／界外一步｜局限：对布尔值与逻辑变量无效、不善长组合 → 是等价类划分的补充

提问：给一个"允许输入 1 到 255 个字符"的文本输入框，按边界值方法，你应该取哪几个值？

预设回答：

- 答"1、255、0、256"：✅✅ 完全正确。教材表 3-2 的原话就是：输入 1 个和 255 个字符作为有效等价类，输入 0 个和 256 个字符作为无效等价类，这几个数值都属于边界条件值。追问：还能不能再加？——可以补 2

和 254 作为界内一步，更稳。

- 答"取 1、2、128、254、255"：⚠ 部分对。128 是中间值，属于等价类里的普通代表值，不是边界；真正必须有的是 0、1、255、256 这四个。

- 答"只要测 1 和 255 就行"：❌ 只测有效边界，等于没测容错。离点——也就是刚好越界的 0 和 256——才是最容易暴露缺陷的地方。

> **易错点 / 考点：**最大的错是把边界值测试做成"只测有效边界"。判断标准很简单：有没有取到"刚好越界"的点。第二个易错点是漏掉输出侧的边界——教材第 47 页明确要求，要根据规格说明的每一个输出条件分别使用范围规则和个数规则，输入侧的边界不是全部。第三个易错点是把边界值当成万能的：考试常出判断题"边界值分析法可以替代等价类划分法"——错，教材的定位是"常被看作是等价类划分法的一种补充，两者结合起来使用更有效"。记忆口诀：边界三取点——上点、内点、离点；判断三步法：找范围 → 定上下界 → 上下各取压界与越界。

> **【思政落点 5】**边界值分析讲的就是"刚刚好"和"差一点点"——多一个字符、少一分钱、差一天。真实工作里的事故，往往正发生在这些差一点点的地方：一个金额的进位、一次还款日的跨月、一条记录的第 255 个字符。把边界值思维带回日常，就是养成严谨求实的习惯：不满足于"大概没问题"，而是把临界那一档亲自试过、把结果如实记下来。

> **过渡：**原理讲完了，可是"正好等于、刚刚大于、刚刚小于"具体怎么取？下一页用两个示例把取点规则写成可以照着抄的清单。

【3.17】示例 1（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页用示例把边界值的取点规则落成清单，一共两条，请大家跟着我记。
>
> 第一条：如果输入条件规定了值的范围，就要取刚达到这个范围边界的值，以及刚刚超越这个范围边界的值，作为测试输入数据。课件给的表达式是 (1,20]，读作大于 1、小于等于 20，也就是 1<x≤20。这个区间的两端不一样：右端是闭的，20 在范围里；左端是开的，1 严格来说不在范围里。所以如果我们讨论整数输入，压在界上的有效值是 20，界内最近的一个有效值是 2；而刚刚越界的无效值是 21 和 1。请大家体会这一对：一个是"踩在线内"，一个是"踩在线外"，两者只差一个数字，程序的判断却必须区分开——这就是边界值测试的杀伤力所在。

第二条：如果输入条件规定了值的个数，就用最大个数、最小个数、比最小个数少一、比最大个数多一的数据作为测试数据。举个例子，某个列表要求"至少选 1 项、最多选 5 项"，那么你要测的是 1 项、5 项、0 项和 6 项——四个数字，把上下边界各钉一遍。

课件还给了第三个素材：{X}=（1，5，10，15，20）。这是一个有序集合。教材第 48 页的第四条规定：如果规格说明给出的输入域或者输出域是有序集合，比如有序表、顺序文件，就应该选取集合的第一个元素和最后一个元素作为测试数据——也就是这里的 1 和 20。为什么？因为有序集合一旦在首尾出错，比如排序取不到第一个、或者越界读到第 6 个，通常就整段数据都废了。

把这三条合起来看，你会发现它们说的是同一件事：边界不只是"数值的大小"，还包括"个数多少"和"位置首尾"。

板书：①值的范围 → 刚达到边界的值＋刚超越边界的值｜(1,20] 即 1<x≤20 → 有效 2、20；无效 1、21｜②值的个数 → 最大个数、最小个数、最小−1、最大＋1（选 1～5 项 → 1、5、0、6）｜③有序集合 {1,5,10,15,20} → 取第一个 1 与最后一个 20

提问：为什么 (1,20] 这个区间，取 20 是有效数据，而取 1 反而是无效数据？两个端点看上去都在"边上"。

预设回答：

- 答"因为左端是小括号，右端是中括号"：✅ 答到点子上了。小括号表示开区间，1 不包含在内；中括号表示闭区间，20 包含在内。判断边界的有效无效，先看开闭，再看取值。

- 答"1 和 20 都有效，反正都在边界附近"：❌ 典型误区。点评：区间符号不是装饰，它直接决定程序里的比较运算是"大于"还是"大于等于"。1 恰好是那个"差一点不合格"的值，正是最该测的无效数据。

- 追问"如果输入是小数而不是整数呢"：那 1 和 20 之间就没有"最近的一个整数"了，这时要靠规格说明里的精度要求去取，比如 1.01 和 20.00。核心动作不变：一侧压在界上，一侧刚刚越界。

易错点 / 考点：第一，看错区间开闭，把开区间端点当成有效值，这是填空题最常见的失分点。第二，只对"范围"取边界，忘了"个数"和"有序集合首末"。第三，把"比最小个数少一"写成"最小个数减一"而忘了它可能是零或负数——0 往往是被单独拦截的一类，也要单独出一条用例。教材表 3-2 的字符例子可以直接背下来：允许 1～255 个字符，有效取 1 和 255，无效取 0 和 256。口诀：（1，20]——左开右闭，界内取 2，界外取 1 和 21。

过渡：示例 1 讲的是数值和个数的边界。下一页的示例 2 换一种输入形态——不是填数字，而是在一堆复选框里做选择，看看边界值在那种界面上还剩几个点可抓。

【3.18】示例 2（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

> 示例 2 换了场景，手法没变。这一页截的是会议系统设置流程里的第 8 步，Attendee Privileges，也就是参会者权限设置。中间是一列复选框：Save 保存、Print 打印、Annotate 标注、View participants list 查看参会者名单、Control next or previous images 控制上一张下一张图片、View thumbnails 看缩略图、Control applications/Web browser or desktop remotely 远程控制应用和桌面、View any document 看任意文档、View any page 看任意页面、Contact operator privately 私聊主持人等等。下面还有一栏"与谁私聊"：Host 主持人、Presenter 演讲者、All attendees 全体参会者。
>
> 这类界面怎么用边界值？这里没有"数值大小"的边界，要抓的是"选择个数"的边界。课件给出的用例清单正是这么做的：第一，任意的正常值，随机选择几个选项，这是等价类里的普通代表值；第二，边界值，选择所有选项，是个数的上界——全选，最满的状态；第三，边界值，一个都不选，即空状态，是个数的下界；第四，边界值，只选择一个选项，比下界多一档。
>
> 请特别注意"一个都不选"。很多人写用例时下意识跳过它，觉得"用户总得选一个吧"。可程序只要没有对空选择做防护，用户一路点下一步就能触发它。这里还藏着组合边界：全选加全体参会者、一个不选加主持人，都容易让权限判断逻辑翻车。
>
> 这个示例的启发是：边界值方法的核心不是"数字"，而是"极值状态"。只要能回答出"什么是最小、什么是最大、什么是空、什么是满"，边界就找到了。

> **板书：**场景：Attendee Privileges 权限复选框｜取点：正常值→随机选几项｜边界：全选（个数上界）／一个都不选（下界＝空）／只选一个（比下界多一）｜边界＝极值状态，不限于数值

> **提问：**在这类复选框界面里，"一个都不选"到底算有效等价类还是无效等价类？会不会因为它"不合法"就不用测了？

> **预设回答：**

- 答"无效，但必须测"：✅ 这是最实用的回答。追问：为什么要测无效的？——因为无效等价类检验的正是程序的容错与异常保护。如果界面允许用户一步跳到下一步，程序就必须给出合理的提示，而不能崩掉或者把权限置成空。

- 答"无效，所以不用测"：❌ 这是把"无效等价类"当成"不值得测的数据"。教材第 46 页讲得很清楚：不但要保

证所有有效数据输入产生正确输出，还要保证输入错误或空输入时能有异常保护，这样的测试才能保证软件的可靠性。

- 答"不选也行，那它是有效的"：⚠ 要看规格说明。如果产品明确允许"不授予任何权限"，它就是有效等价类；如果一定要选，它就是无效类。判断依据永远是规格，而不是我们的猜想。

> **易错点 / 考点：**第一，认为"复选框界面没有边界值可测"——错，个数边界就是它的边界。第二，漏掉空状态和满状态。第三，只测单个控件，不测组合（全选＋某个下拉框的取值）。考点常见于案例题：给出一个多选界面，要求写出边界值用例，评分点通常是"有没有全选、有没有空选、有没有只选一个"。口诀：多选界面三档——全选、单选、不选。

> **过渡：**到这里，边界值方法已经能用在数值、个数、位置和多选场景上了。但还有一类边界特别阴，它们不是业务规则规定的，而是计算机本身的表示能力决定的。下一页我们专门看这些特殊边界。

【3.19】一些特殊的边界值（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲一些特殊的边界值，也就是那些不由业务规则决定、而由计算机自身的表示能力决定的边界。为什么必须认识它们？因为这类边界一旦越过去，程序不是报个错那么简单，而是直接发生溢出、截断、乱码。右边这两张表，就是两套工业界的"常识刻度"。

先看左边的二进制单位表。Bit 位，取值 0 或者 1；Nibble 半字节，0 到 15；Byte 字节，0 到 255；Word 字，单字 0 到 65535，双字 0 到 4294967295；再往上 Kilo 是 1024，Mega 是 1048576，Giga 是 1073741824，Tera 是 1099511627776。这些数字不是拿来背的，是拿来取边界的。教材就给了现成的用法：对字节做检验，边界值条件可以设成 254、255 和 256——255 是能存的最大值，256 就是那个刚刚越界、会溢出的值。

再看右边的 ASCII 表。Null 是 0，Space 空格是 32，斜杠是 47，数字字符 0 是 48、9 是 57，大写 A 是 65、Z 是 90，左中括号是 91，单引号是 96，小写 a 是 97，z 是 122，左花括号是 123。请重点记两件事：第一，大写字母和小写字母的编码是分开的两段，A 是 65、a 是 97，它们之间还夹着 91 到 96 这六个符号；第二，教材强调，0 到 9 的边界字符是斜杠"/"和冒号":"——斜杠 47 排在 0 的前面，冒号 58 紧跟在 9 的后面。做文本输入或文本转换的测试，如果能想到拿这几个边界字符去试，很多隐藏的类型判断缺陷就藏不住了。

一句话总结这一页：数值边界看数据类型，字符边界看编码表。

> **板书：**二进制：Bit 0/1｜Nibble 0–15｜Byte 0–255｜Word 0–65535 或 0–4294967295｜K 1024｜M 1048576｜G 1073741824｜T 1099511627776｜字节检验取 254、255、256｜ASCII：Null 0｜Space 32｜/ 47｜0 48｜9 57｜A 65｜a 97｜z 122｜0–9 的边界字符＝"/"与":"

> **提问：**一个用单字节存储年龄字段的系统，允许输入 0 到 255。按今天讲的，你会挑哪几个值去测？为什么其中有一个值一定不能少？

> **预设回答：**

- 答"254、255、256"：✅✅ 直接对上了教材第 49 页的原话。追问：为什么这三个？——255 是字节能表示的最大值，是压在界上的有效值；254 是界内一步；256 表示不了，是刚越界值，用来验证程序有没有做溢出防护。

- 答"0、1、128、254、255"：⚠ 有一半对。0 是下边界，值得测；128 是普通中间值，价值有限；真正的要害是"256"这个越界值，不能漏。

- 答"测 255 就够了"：❌ 只测上界不测越界，等于把最危险的一档放过去了。历史上大量溢出缺陷就是这么漏掉的。

> **易错点 / 考点：**第一，把"特殊边界值"理解成背数字。要记的是取点方法：类型上限本身就是边界，上限加一就是离点。第二，混淆字符和编码：0 是数字字符还是空字符？在 ASCII 表里，Null 是 0，数字字符 0 是 48，二者完全不同，考试常拿这个设陷阱。第三，忽略大小写之间的符号区间 91 到 96——做邮箱、用户名之类的大小写不敏感校验时，这一段是最容易出错的。考点形式多为填空题与选择题，比如"字节型字段的边界值测试应取哪三个值"。

> **过渡：**刚才这些边界还算有章可循，可真实的边界名单比这长得多，而且很多边界是"语义"上的。下一页我们把常见边界的成对说法一次列全，帮你建一个自己的边界清单。

【3.20】一些特殊的边界值 –续（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页把常见边界继续列全，而且用的是一对对的说法——你只要记住这些"对"，边界就成清单了。先读一遍：First 和 Last，即首和末，对应 First−1 和 Last＋1；Min 和 Max，最小最大，对应 Min−1 和 max＋1；Star 和 Finish，开始和结束，对应 Start−1 和 Finish＋1；Empty 和 Full，空和满；Less than empty 和 more than full，比空还少、比满还多；Slower 和 Faster，更慢和更快；Largest 和 Smallest，最大和最小；

Over 和 Under，以及 just Over 和 Just Under，也就是刚刚超过和刚刚低于；Shortest 和 Longest，最短和最长。最后跟着一串省略号——这个清单是开放的，你遇到的每一个"极值"都可以往里加。

旁边那一列绿色字是分类维度：数值、字符、位置、数量、速度、体积。两列对着看，会发现边界值方法的适用面比想象中宽：数值有大小边界，字符有编码边界，位置有首末边界，数量有多少边界，速度有快慢边界，体积有空满边界。最后课件还点了一组容易被忽略的：缺省值、空格、(none)、0、Null 等。这些值在很多系统里是"默认可不填"的，恰恰最容易被程序错误处理。

左边的图是 Word 的段落设置对话框，里面有一堆看起来不起眼的输入：缩进 Left 和 Right 的磅值、行距 Before 和 After、特殊格式下拉框里的 (none)、行距下拉框里的 Single。这张图正好当练习素材：这些控件里，哪几个的边界值最值得一测？我建议先盯住 (none) 和 0 ——一个是"没设"，一个是"设为零"，很多排版缺陷就出在两者的混淆上。

板书：成对清单：First/Last（±1）｜Min/Max（∓1）｜Start/Finish（±1）｜Empty/Full｜Less than empty / more than full｜Slower/Faster｜Largest/Smallest｜Over/Under、just Over/Just Under｜Shortest/Longest｜维度：数值・字符・位置・数量・速度・体积｜另加：缺省值・空格・(none)・0・Null

提问：为什么"缺省值、空格、(none)、0、Null"这几个看上去无害的值，反而要专门列出来当边界？

预设回答：

- 答"因为它们经常被程序当成同一个东西处理"：✅✅ 说到根子上了。0 和"没填"在很多数据库里是不同的状态，可代码里一个默认值就可能把它们抹平，于是"用户没填"被读成"用户填了 0"，报表全错。

- 答"因为用户经常不填"：✅ 也对，但只说了使用频率。真正的原因是要验证程序能不能区分"没有"和"零"。

- 答"这些不算边界吧"：❌ 纠正。教材第 49 页把它们明确列为"其他边界值检验"，原话就包括默认值、空值、空格、未输入值、零、无效数据、不正确数据和干扰（垃圾）数据等。

易错点 / 考点：第一，只把边界理解成"数字的大小"，忽略位置、数量、速度、体积这些维度。第二，记不住成对说法的方向：First−1 是比第一个还靠前，Last＋1 是比最后一个还靠后，考试中常把符号搞反。第三，漏掉"无值"这一族。记忆口诀：极值成对，两侧各加一；空与满、有与无，都要各来一条。

过渡：理论、规则、特殊边界都齐了，下面用一道综合练习题把边界值方法真正用起来——请注意，这道题的输入不是数字，而是两个表单字段。

【3.21】练习（4 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是当堂练习，题目是：用边界值方法为两个字段设计测试数据。第一个字段 Username 用户名，规则有四条：12 个字符以内、不能为空；只能用字母、数字、下划线和连字符，不能用空格；必须以字母开始；大小写不敏感。第二个字段 Password 密码，规则两条：不少于 6 个字符；大小写敏感。

先说时间和做法。给大家四分钟独立做题，按"字段 → 规则 → 上点／内点／离点"的格式列一张小表；再用两分钟和同桌互查，重点查"有没有离点"；最后两分钟我们一起对答案，我会请两位同学分别报 Username 和 Password 的数据。做题方法照刚才学过的那样：先按规则划分出合法区间，再取三种点——上点就是边界本身，内点就是区间内的普通值，离点就是刚刚越界的值。所谓"恰好取到边界"和"刚刚越过边界"，是这道题的两个评分要害。

提醒几句容易踩的地方。第一，Username 的四条规则不是一体的，每一条都要单独取边界：长度有长度的边界，首字符有首字符的边界，字符集有字符集的边界。第二，不要只给有效值，没有离点就不叫边界值测试。第三，看到"大小写不敏感"和"大小写敏感"这四个字，请想一想，它们对测试数据意味着什么？先自己做，我稍后点评。

板书：练习：边界值法设计测试数据｜Username：≤12 字符・非空・字母数字_-・不以空格・字母开头・大小写不敏感｜Password：≥6 字符・大小写敏感｜表格列：字段｜规则｜上点｜内点｜离点｜预期结果

提问：Password 的规则是"不少于 6 个字符，大小写敏感"。请一位同学只回答一个问题——你会取哪几个长度值？

预设回答：

- 答"5、6、7，再加一个超长串"：✅✅ 完全符合要求。5 位是无效数据，刚刚低于下界；6 位是有效数据，正好取下界；7 位是有效数据，正常长度；再补一条超长串，验证系统有没有长度上限或者会不会截断。

- 答"6、8、10"：❌ 这三个值全在合法区间里，属于内点，同处一个等价类，测一个就够了，其余两个是浪费。点评：边界值测试的价值不在"多"，而在"卡在界上"。

- 追问"有效数据给哪些"：abc123 和 Abc123 都是有效的 6 位密码。但请注意，"大小写敏感"意味着 Abc123 与 abc123 是两条不同的用例，两条都要测，不能只测一条就认为覆盖了——这正是本题埋的考点。

预设回答（Username 要点，教师对答案时逐条给出）：

- 有效数据：abc（3 位，正常）；a1_-b（5 位，把字母、数字、下划线、连字符四类合法字符都含进去）；abcdefghijkl（正好 12 位，取上界）；AbC 与 abc（用来验证"大小写不敏感"是不是真的把两者当作等价输入）。

- 无效数据（每一条对应一个无效等价类的边界）：空串（空输入的下界，0 位）；abcdefghijklm（13 位，刚刚越过上界）；1abc（数字开头）；_abc（连字符开头）；abc d（含空格）；abc#（含非法字符）。

- 收尾提醒两个易错点：只测有效值不算边界值测试，必须给出离点，也就是刚好越界的值；另外"空"在很多系统里是被单独拦截的一类，要单独出一条用例，不能和"长度不够"混为一谈。

> **易错点 / 考点：**第一，把"大小写不敏感"当作可以少测一条的理由——恰恰相反，它要求你用同一条数据的大小写两种写法去验证 BUG 是否存在。第二，把"不能为空"和"至少 1 个字符"当成同一条规则，实际实现里它们常常是两段不同的校验代码。第三，多规则字段只测长度、不测字符集和首字符。考试中这类题目按点给分，评分点通常是：上界、下界、上下界的离点、非法字符、空值、大小写各一条。

> **过渡：**到这里，第 3.2 节基于输入域的方法就讲完了——等价类负责"分块"，边界值负责"卡边"，两者配合，单因素的输入就被管住了。下一页开始，我们面对一个更难的问题：多个输入条件一起来的时候怎么办？

【3.22】3.3 基于组合及其优化的方法（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 从这一页开始进入第 3.3 节，基于组合及其优化的方法。这一节底下有四个小节：3.3.1 判定表方法，3.3.2 因果图法，3.3.3 Pair-wise 方法，3.3.4 正交试验法。四把工具，解决的是同一个难题。
>
> 难题是什么？教材讲得很直白：等价类划分法和边界值分析法，处理的是单因素、单变量的数据；可多因素、多变量的输入情况就不一样了，这些因素之间会相互组合。虽然各种输入条件可能出错的情况都已经被考虑到了，但多个输入情况组合起来可能出错的情况，却被忽视了。也就是说，你把用户名测透了、把密码测透了，不代表"用户名合法＋密码合法＋验证码过期"这组组合就不出事。而检验各种输入条件的组合并不容易——就算把每个条件都划分成等价类，它们之间的组合情况也相当多，这就叫组合爆炸。
>
> 所以教材引入了组合分析。它的定位是：一种基于每对参数组合的测试技术，主要考虑参数之间的影响是主要的错误来源，而大多数的错误起源于简单的参数组合。这句话是这一节的灵魂，请划下来。
>
> 组合分析的优点是：低成本实现、低成本维护、易于自动化、易于用较少的测试案例发现更多的错误，而且

用户可以自定义限制条件。缺点是：经常需要专家领域知识、不能测试所有可能的组合、也不能测试复杂的交互。请大家注意这个态度——教材没有把组合方法吹成万能药，它明确列出了局限。这也是我们这门课反复强调的：方法没有优劣，只有适用。

接下来的四页，我们依次看这四把工具。

> **板书：** 3.3 基于组合及其优化的方法｜3.3.1 判定表法｜3.3.2 因果图法｜3.3.3 Pair-wise 方法｜3.3.4 正交试验法｜单因素：等价类・边界值 → 多因素：组合爆炸｜组合分析：基于每对参数组合｜主要错误来源＝参数之间的影响；大多数错误起源于简单的参数组合｜优点：低成本・易自动化・用例少｜缺点：需领域知识・不能覆盖全部组合・不善复杂交互

提问： 教材为什么说"大多数错误起源于简单的参数组合"？如果这句话成立，对我们的测试策略意味着什么？

预设回答：

- 答"所以两两组合就够用，不用做全组合"：✅✅ 这正是 Pair-wise 方法的立论基础——不必覆盖所有组合，只要覆盖所有"两两组合"，就能抓住大部分因参数交互引起的缺陷。

- 答"那是不是只测单因素就行了"：❌ 纠正。单因素都测过，恰恰会漏掉"单独看都合法、合起来出问题"的情况，组合测试补的就是这个缺口。

- 追问"那全组合不是更保险吗"：保险但不可行。一百个因素、每个取两个值，全组合是 2 的 100 次方，穷举在时间上不可能完成——这和第一章讲穷举法不可行是同一个道理。

易错点 / 考点： 第一，混淆"组合方法"与"等价类／边界值"的适用场景，答成"组合方法更高级"。第二，误以为组合测试能覆盖复杂的交互——教材明确说它的缺点之一就是不能测试复杂的交互。第三，把"每对参数组合"理解成"每个参数取值两两配对测一遍数值"，其实指的是任意两个参数的取值对都要在用例集中出现。考点多为概念题：组合分析的定义、优点与缺点各写两条。

过渡： 四把工具里的第一把，也是最严格、最有逻辑性的一把，就是判定表方法。它用一张表格，把多因素的条件组合和对应动作一次性摊开。

【3.23】3.3.1 判定表方法（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲第 3.3.1 节，判定表方法。先看它的适用条件：在实际应用中，许多输入是由多个因素构成的，而不

是单一因素，这时就需要多因素组合分析。对于多因素输入，如果它们之间的关系比较简单——根据某一个输入组合就能直接判断输出结果，而且每个输入条件或者输出结果都可以用"成立"或"不成立"来表示，也就是只有 1 和 0 两个取值——那么就可以直接对输入条件进行组合设计，不需要再做因果分析，这时就用判定表方法。

判定表由"条件和活动"两部分组成：条件作为输入，活动作为输出。换句话说，它列出了一个测试活动执行所需的全部条件组合；所有可能的条件组合定义了一系列选择，而测试活动（也就是结果输出）需要考虑每一个选择。比如说，"打印机能不能打印出正确的内容"，受驱动程序、纸张、墨粉这几个因素影响，每个因素都有成立和不成立两种状态，两两相乘就是一张完整的组合表。

为什么说它严格？教材的原话是：判定表从输入条件的完全组合来满足测试的覆盖率要求，具有很强的逻辑性，所以基于判定表的测试用例设计方法是最严格的组合设计方法之一，其测试用例具有良好的完整性。请注意"完全组合"这四个字——它是这一节里唯一做到全覆盖的方法，后面讲的 Pair-wise 和正交试验法，都是在组合数爆炸之后不得不做的取舍。

但也正因为"完全组合"，它的代价也很清楚：条件一多，表的列数就按 2 的 n 次方增长。所以判定表用在小规模、高风险的逻辑上最合适——比如订单优惠规则、权限授予规则、费用结算规则。这些地方逻辑一错就是真金白银，值得把它摊开。

> **板书：**判定表方法｜适用：多因素输入，关系简单，输入/输出只有成立（1）与不成立（0）｜结构："条件（输入）"＋"活动（输出）"｜目标：完全组合覆盖 → 最严格的组合设计方法之一，用例完整性强｜代价：n 个二值条件 → 2^n 列｜例：打印正确内容 ← 驱动程序 ・ 纸张 ・ 墨粉

提问：一个业务模块有 4 个二值条件，用判定表做完全组合要考虑多少种情况？这个数字告诉我们什么？

预设回答：

- 答"16 种"：✅ 2 的 4 次方等于 16，这就是判定表的列数。追问：那 10 个条件呢？——1024 种，已经很难手工维护了。

- 答"4 种"：❌ 把"条件个数"和"组合个数"搞混了。一个条件产生两种状态，四个条件相互组合，是乘法关系不是加法关系。

- 答"16 种，所以判定表其实不实用"：⚠ 结论下得太早。这里的正确态度是：判定表的价值在于"完整"，代价在于"列数"，所以它适合条件不多但逻辑重要的场景；条件多的时候，才轮到下一页要讲的"合并相似规则"和后面的 Pair-wise。

易错点 / 考点：第一，把判定表当成"画个表格填一填"的文书工作，忽略它"完全组合覆盖"的实质。第二，记不住适用前提：输入输出都能用成立与不成立表示、关系简单、无需因果分析——如果条件之间有复杂的因果链条，那就要用下一页之后要讲的因果图法。第三，忘了判定表是要出用例的：每一列就是一条用例。考点常考适用条件与 2^n 的规模估算。

过渡：知道了判定表长什么样，还得知道它的五个零件分别叫什么。下一页我们逐一定义条件桩、动作桩、条件项、动作项和规则。

【3.24】判定表方法术语及应用（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲判定表的术语。课件列了五个概念，我逐个解释，请大家对照右边的表格看。

第一个，条件桩：列出问题的所有条件。在右边的表里，就是 C1 到 C4 这几行——注意"桩"字，它指的是问题的条件清单，是一张表最左边那一列。

第二个，动作桩：列出可能针对问题所采取的操作。表里就是 A1 到 A5 这几行，也就是程序可能做出的各种响应。

第三个，条件项：针对所列条件的"具体赋值"，也就是每个条件分别取真值还是假值。表里每一列的 Y 和 N，就是条件项。

第四个，动作项：列出在条件项各种取值组合的情况下应该采取的动作。表里那些打叉的格子，就是动作项——它回答的是"这一列的情况下，程序该做哪几件事"。

第五个，规则：任何一个条件组合的特定取值及其相应要执行的操作。一句话说清楚：在判定表中，贯穿条件项和动作项的一列，就是一条规则。

最后请把课件底部标红的那句记住：最后每一列需要设计一条测试用例覆盖，有几条规则（组合）就有几条测试用例。这是判定表与用例之间唯一的换算关系——表上有几列，你就要写出几条用例，不多不少。

至于怎么造表，教材给了四个步骤：第一步列出所有的条件桩和动作桩；第二步填入条件项；第三步填入动作项，制定初始判定表；第四步简化、合并相似规则或相同动作。这四步的顺序不要乱：先有清单，再填取值，再定动作，最后才谈简化。

板书：条件桩＝问题的所有条件（C1…）｜动作桩＝可能采取的操作（A1…）｜条件项＝条件的取值

（Y/N）｜动作项＝该组合下应执行的动作（×）｜规则＝贯穿条件项与动作项的一列｜红字：每一列＝一条测试用例，几条规则就几条用例｜四步：列桩 → 填条件项 → 填动作项 → 简化合并

提问：右边这张初始判定表有 16 条规则，是不是意味着要写 16 条测试用例？如果觉得太多，下一步该怎么办？

预设回答：

- 答"是 16 条，但可以先做简化"：✅✅ 标准答案。追问：怎么简化？——把动作项完全相同、只有某一个条件取值不同的规则合并，用"—"表示这个条件无关。教材给的四步里，第四步就是"简化、合并相似规则或相同动作"。

- 答"不是，随便挑几条重要的就行"：❌ 这是在用例还没生成之前就先削减覆盖率，等于把判定表最值钱的"完整性"扔掉了。简化必须是有依据的合并，不是拍脑袋删列。

- 追问"表里那行 impossible situation 是什么意思"：说明有些条件组合在业务上根本不可能出现。这类列要先标注出来，它本身就是需求分析的成果——如果业务上明明可能出现却标成了不可能，那就是漏测。

易错点 / 考点：第一，把"条件项"和"动作项"记反——条件项是条件的取值（Y／N），动作项是该组合下要执行的动作（1／0）。第二，以为这些术语是文字游戏，其实这一页最硬的考点就是那句红字：几条规则就几条用例，常以填空或计算题出现，n 个二值条件对应 2^n 条规则。第三，混淆"动作桩"和"动作项"：桩是操作清单，项是具体取值。第四，跳过第四步"简化合并"直接出用例，把完整的初始判定表原样当结果。记忆口诀：桩是清单、项是取值；条件桩配条件项，动作桩配动作项；一列一条用例。

【思政落点 6】判定表最可贵的，是把口口相传的业务规则变成一张谁都看得懂的表格：条件、取值、动作一目了然，换了人接手也能照着执行。测试设计要为他人可复用，这是团队协作的基本功。同一份规格，各人脑子里的理解往往不同；把理解写进表格、把分歧摊到纸面上，团队才能在同一个事实基础上对齐。认真留痕、可追溯，也是对同事负责。

过渡：刚才说简化是有依据的合并，那么依据到底是什么？下一页我们不急着看新知识，先把这张 16 列的表拿来做一次当堂讨论。

【3.25】判定表组合优化讨论（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是讨论题，题目就叫"判定表组合优化讨论"。这一页讨论的是判定表怎么简化。这张表一共 16 列，也

就是 4 个条件桩（C1 到 C4）的全部组合；动作桩有 5 条（A1 到 A5），其中 A5 标注着 impossible situation——意思是某几种条件组合在业务上根本不可能出现。

请先看我用红框圈出来的地方——第 10 列和第 12 列。我把它们并排比较一下：C1，两列都是 N；C2，两列都是 Y；C4，两列都是 N；唯一的差别是 C3，第 10 列是 Y，第 12 列是 N。再看动作：这两列只有 A4 打了叉，其余动作全空。结论是什么？C3 取真还是取假，对这两列该做什么动作完全没有影响。既然没影响，就没有必要分成两列——把 C3 标成"—"，表示"无关项"，两列合并成一列。

现在布置讨论。给大家三分钟，同桌两人一组，做三件事：第一，把这张表的条件桩、动作桩、条件项、动作项、规则五个术语各指一次，确认你们找对了；第二，找出除了第 10、12 列之外，还有哪些列可以合并，并说清依据；第三，用一句话总结"什么情况下可以合并"。三分钟之后我请两三个组来汇报，然后我们一起收口。注意，先讨论、后点评，我不会提前给答案。

讨论的时候请把这条判断标准念给自己听：若两条或多条规则的动作项完全相同，而且只有某一个条件的取值不同，就说明这个条件对该动作没有影响，可用"—"标记后合并。这句话就是合并的唯一依据。另外还要守住一条底线：合并不能改变语义——合并前后，针对任何一个具体的输入组合，判定表给出的动作必须完全一样，否则就是过度合并，会造成漏测。

> **板书：**讨论三步：①指认条件桩/动作桩/条件项/动作项/规则 ②找出可合并的列并说明依据 ③一句话总结合并条件｜依据：动作项完全相同＋仅一个条件取值不同 → 该条件记"—"，合并｜例：第 10 列（C3＝Y）与第 12 列（C3＝N）动作相同（A4＝×）→ 可合并｜底线：合并不得改变语义

> **提问：**请汇报——除了第 10、12 列，你还能找出哪些列可以合并？判断的依据是什么？如果有一列标着 impossible situation，它能不能参与合并？

> **预设回答：**

- 答"动作项逐列完全一致，且只有一个条件取值不同的两列可以合并"：✅✅ 说得准确。可以直接用课件上的第 10、12 列作证据：C1、C2、C4 三行完全相同，只有 C3 一个是 Y 一个是 N，动作行又都是 A4，所以合并、把 C3 记为"—"。

- 答"把动作一样的列都合起来"：⚠ 不完整。只对了一半——还要看条件是否只差一个取值。如果两列有三个条件都不同而动作恰好相同，那很可能是规格说明本身有歧义，不能简单合并了事。

- 答"impossible situation 那一行不影响合并"：❌ 要提醒。不可能出现的条件组合要单独标注、单独确认，它既不能当作正常规则参与合并，也不能因为"反正不会发生"就删掉——它的正确用途是反过来验证需求：这

个组合真的不可能吗？

- 教师补充（对答案时给出完整口径）：判定表由条件桩、动作桩、条件项、动作项和规则组成，n 个二值条件理论上产生 2^n 条规则，条件一多就组合爆炸；优化的依据只有一条，就是"动作项完全相同、且只有某一个条件的取值不同，说明这个条件对该动作没有影响"，用"—"标记后合并。合并有三个前提：动作项逐列完全一致；合并后不遗漏任何动作；被合并的规则在同一条件下互为取值，一般是成对或者成 2 的幂数的组合并。

- 教师举例（教材的打印机判定表）：驱动、纸张、墨粉三个条件共 8 条规则；当"无纸"和"无墨粉"同时成立时，无论驱动程序是否正确，动作都是提示没有纸张——因为业务约定是优先警告缺纸——于是这几条就可以合并成一列。

- 教师收口讲价值与风险：合并的价值是规则数从 2^n 降到实际有区分度的条数，用例数随之下降；而且合并过程本身就能暴露规格说明里没写清楚的情况——合并不了的规则，往往正是需求有歧义的地方。但要注意，合并不能改变语义，合并前后针对任一具体输入组合给出的动作必须相同，否则就是过度合并，会漏测。

> **易错点 / 考点：**第一，把"简化"理解成"删掉不重要的规则"，这是最常见的错。第二，合并时把"动作项完全相同"这个前提漏掉，只看条件不同就合。第三，忘记合并后要用"—"明确标出无关项，导致表格读不出信息。第四，对 impossible situation 处理草率。考点形式上一般考两道：一是不定项选择（给出几列，问哪两列可以合并、依据是什么），二是简答题（判定表简化的条件与注意事项）。

> **过渡：**优化讲完了，我们用一张真实业务的判定表把这一节收口。下一页这张商品订单的判定表，正好可以把条件桩、动作桩、无关项和用例数四件事一次串起来。

【3.26】判定表方法示例（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 最后用一个完整示例把判定表落地。这张表的业务背景是商品订单处理，先看条件项那三行：C1 是"此商品在经营范围"，C2 是"此商品可以发货"，C3 是"此客户没有拖欠过付款"。再看动作项那五行：A1 货到后允许客户转账；A2 货到客户必须立即付款；A3 重新组织货源；A4 电话通知；A5 书面通知。
>
> 我们一列一列读。R1 这一列：C1 是 N，也就是商品不在经营范围，而 C2 和 C3 都填了"—"，这就是上一页说的无关项：商品根本不在经营范围，讨论能不能发货、客户有没有欠款都没有意义，所以这两条不参与判断。对应动作是 A5，书面通知。
>
> R2：C1 是 Y，商品在经营范围；C2 是 Y，可以发货；C3 是 Y，客户没有拖欠过付款。三条都成立，动作是

A1，货到后允许客户转账——这是对信用良好的客户给的待遇。

R3：C1 是 Y，C2 是 Y，但 C3 是 N，也就是客户拖欠过付款。商品能发货，但信用有问题，所以动作变成 A2，货到客户必须立即付款。

R4：C1 是 Y，C2 是 N，商品不能发货，C3 是 Y。动作有两个：A3 重新组织货源，A4 电话通知。

R5：C1 是 Y，C2 是 N，C3 也是 N。动作是 A3 重新组织货源，加 A5 书面通知。

注意 R4 和 R5 的差别：商品都不能发货，处理方式都是重新组织货源，区别只在通知方式——客户信用好就电话通知，信用不好就书面通知。所以 C3 这个条件在"不能发货"这个分支里的作用，是决定通知方式，而不是决定要不要发货。这层业务含义，正是判定表逼着你写清楚的。

最后做一次换算：这张表一共 5 条规则，对应 5 条测试用例——R1 到 R5，一条不多、一条不少。这就是判定表的价值：把散落在需求文档和业务人员脑子里的规则，变成一张可直接执行、可逐列验收的表。

板书：条件桩：C1 此商品在经营范围｜C2 此商品可以发货｜C3 此客户没有拖欠过付款｜动作桩：A1 货到后允许客户转账｜A2 货到客户必须立即付款｜A3 重新组织货源｜A4 电话通知｜A5 书面通知｜R1：C1＝N（C2/C3 为"—"）→ A5｜R2：Y/Y/Y → A1｜R3：Y/Y/N → A2｜R4：Y/N/Y → A3＋A4｜R5：Y/N/N → A3＋A5｜5 条规则 → 5 条用例

提问：R2 和 R3 的差别只有 C3 一个条件，结果却是"允许转账"和"必须立即付款"两种完全不同的处理。这说明判定表在测试里真正的价值是什么？

预设回答：

- 答"它能发现单个条件变化引起的处理差异"：✅✅ 说得很好。单个条件从 Y 变 N，业务动作就整条改变，这正是最需要被用例锁住的逻辑分支——客户少还一笔钱，授信规则就该变。

- 答"它把需求写清楚了"：✅ 也对。追问：那如果业务方看到这张表说"R3 应该也允许转账，只是额度小一点"，说明什么？——说明原来的规格说明是有歧义或者遗漏的，判定表在测试之前就把这个洞照出来了。

- 答"5 列就是 5 条用例，很好数"：✅ 这是最直接的收益。补充：R1 里那两个"—"也提醒我们，用例只需覆盖业务上说得通的条件组合，不必硬凑 2 的 n 次方种。

易错点 / 考点：第一，看到"—"就以为那一格漏填了——"—"是有明确含义的无关项。第二，只读动作不读条件，把 R4 和 R5 当成同一类，忽略通知方式（电话／书面）这条差异。第三，忘了用例数等于规则数，考试里要求"据此设计测试用例"，标准答案就是逐列列出 R1 到 R5。第四，把条件桩和动作桩写反——条件桩回

答"判定什么"，动作桩回答"做什么"。判断三步法：看条目内容是判定还是动作 → 条件项只有 Y/N/—，动作项只有 1/0 → 一列一条用例。

过渡：判定表适合条件之间关系简单、每个条件都能用成立或不成立表示的场景。可现实里还有一类规格说明：条件之间层层依赖，某个输入会遮住另一个输入。这时候表格就不够直观了。下一次课我们接着讲因果图法，它用图形把这些依赖关系画出来，再转成判定表。今天讲的第 3.2 节和第 3.3 节前半部分就到这里，请大家课后把第 3.21 页这道练习重新做一遍，并整理成可以直接提交的用例表。

【3.27】3.3.2 因果图方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

上一节我们停在判定表上。判定表严格，但它有个隐含前提：每个条件的取值只有"成立"和"不成立"两种，也就是 1 和 0。可现实中的规格说明书不是这么写的，它写的是自然语言："第一列字符必须是 A 或 B，第二列必须是一个数字，这种情况下才修改文件；第一列不对就输出信息 L，第二列不是数字就给出信息 M。"一堆"或"、一堆"与"、一堆"如果……就……"，还夹着隐性限制——人读得懂，表格读不懂。

因果图法，就是把这套自然语言规格翻译成一张能算出组合的图。定义是：因果图法借助图形着重分析输入条件的各种组合，每种组合条件就是"因"，它必然有一个输出结果，这就是"果"。教材评价很高——因果图是一种形式化的图形语言，不仅能发现输入、输出中的错误，还能指出程序规范的不完全性和二义性。注意后半句：它不只是拿来生成用例的，更是一面照妖镜，规格里哪里含糊、哪里漏了情况，一画图就露出来了。

适用场景课件写得很直接："针对更为复杂的‘多种输入条件、产生多种结果’设计组合测试用例。"条件少、结果单一，判定表够用；一旦多输入、多输出，还要区分哪些组合合法、哪些不成立，就该请因果图出场。

怎么做？课件给了五步。第一步，分析软件规格说明，判断哪些是原因也就是输入条件、哪些是结果也就是输出，给每个原因和结果赋予一个标示符，原因记 C1、C2，结果记 E1、E2。第二步，找出因与果、因与因之间的对应关系，画出因果图。第三步，标出哪些因果关系不可能发生，也就是标上约束或限制条件。第四步，把因果图转化为判定表，让复杂的逻辑关系变成一张简单的组合矩阵。第五步，把判定表的每一列转化为一条测试用例。教材归纳为四步，差别只是把第一步拆开，两套答法都对。

板书：因果图法（Cause-effect Diagram）｜因＝输入条件（C1、C2…），果＝输出结果（E1、E2…）｜适用：多输入条件、多输出结果 ｜五步：①分析 Spec、赋标示符 → ②找出因与因、因与果关系、画因果图 → ③标注不可能的因果关系＝约束／限制 → ④转化为判定表（组合矩阵）→ ⑤每列＝一条用例 ｜价值：还能暴露规范的不完全性与二义性

提问：规格说明里写着"第一列字符必须是 A 或 B"，如果只用判定表，这条"或"你打算怎么处理？它跟因果图里的"或"关系有什么不一样？

预设回答：①答"判定表里也写 1 和 0 就行"——点评：判定表能表达"或"，但它表达的是条件之间的逻辑组合，画不出"哪些组合不允许出现"，而本页第三步的约束恰恰是判定表写不出来的部分。②答"因果图就是画个图好看，最后还是要转成判定表"——点评：方向对了一半，判定表确实是因果图的下游产物，课件第四步就是"创建（转化为）判定表"；但因果图的独立价值在于用图形把约束外化出来，让人一眼看出规范里的漏洞与二义。③答"A 或 B 要拆成两个条件，再加一个互斥约束"——这是最专业的回答，正好接上后面两页。

易错点 / 考点：三个高频错点。第一，把因果图的"因"和"果"记反——因是输入条件，果是输出结果，标示符一般用 C 打头表示 Cause、E 打头表示 Effect。第二，漏掉第三步"标注约束"，画完图就直接转判定表——那样得到的组合里会混进根本不可能出现的非法组合，用例看着多，其实是无效的。第三，把因果图与判定表的关系搞反——顺序是"规格 → 因果图 → 判定表 → 用例"，判定表是中间产物，不是起点。考试常考选择题（因果图的适用场景／设计步骤排序）和案例题（给一段规格说明，要求画出因果图并转判定表）。记忆三步法：先分清因与果，再补上约束，最后才转表出用例。

过渡：要画图，先得认识笔和纸——下一步我们就来认因果图的四个基本符号：有因必有果、非、或、与。

【3.28】因果图的基本符号（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

要画因果图，先得认识它的四个基本符号。第一个，有因必有果关系。画法最简单：一个节点 i，一条直线箭头指向节点 j，表示 i 成立则 j 出现。这是最基本的连线，也是所有因果关系的地基。

第二个，非-关系。课件的表述是"只有当因 i 不存在时，果 j 才出现"。图上画法是在直线上加一条小波浪线，教材里用 ∽ 表示取反。生活里很好找例子：设"因 i＝订单已经支付"，那么"果 j＝提示待付款"，只有当 i 不存在、也就是钱还没付的时候，这个提示才出现。注意，非关系不是说"i 成立则 j 不出现"，而是说"j 出现的唯一前提就是 i 不成立"。

第三个，或-关系：如果因 i_1、或因 i_2、或……或因 i_n 存在时，结果 j 才出现。画法是多条输入线汇到同一条弧线上，再由这条弧线指向结果 j，弧线上标一个 ∨ 号。关键词是"只要有一个成立就行"。类比一下：登录

系统时，账号密码、手机验证码、第三方扫码，三种方式里任何一种成功，登录这个结果就出现——这就是或。

第四个，与-关系：只有当因 i_1 与因 i_2 与……与因 i_n 同时存在时，结果 j 才出现。画法同样是多条输入线汇成一条弧线再指向 j，弧线上标一个 ∧ 号。关键词是"缺一不可"。类比：转账成功这个结果，要求"账户余额充足"与"支付密码正确"同时成立，少一个都不行。

四个符号给一句口诀记牢：直线表示有因必有果，波浪线表示非，∨ 号弧线表示有一个就行，∧ 号弧线表示缺一不可。最后提醒一句：这四个符号只描述了条件成立之后结果怎么变，还没有描述"哪些取值组合根本不允许出现"。那部分内容，要靠下一页的约束符号。

板书：①有因必有果：i ——→ j ② 非（∽）：只有 i 不存在，j 才出现（直线＋波浪线）③或（∨）：i_1 或 i_2…或 i_n 存在 → j（弧线标 ∨）④与（∧）：i_1 与 i_2…与 i_n 同时存在 → j（弧线标 ∧）｜口诀：直线有果・波线取非・∨ 有一个就行・∧ 缺一不可

提问：手机"静音模式"打开之后，来电就没有铃声，只有震动。这里面哪个是因、哪个是果？它是与、或、非里的哪一种？

预设回答：①答"静音是因、没有铃声是果，属于与关系"——点评：因和果分对了，"静音"确实是一个输入条件；但只列了一个因，谈不上与关系。②答"属于非关系，因为静音打开之后铃声就不出现了"——点评：这是最典型的错误。非关系说的是"i 不成立时 j 才出现"，而这里恰恰是"i 成立"导致了结果变化；正确做法是把它建成"静音模式＝1"这个因，再加"铃声＝0"这个结果条件。③答"还缺一个条件，比如有没有来电"——这个补充很好，说明学生已经在按"多因一果"的方式思考，正好引出下一页的约束与组合。

易错点 / 考点：最大的坑是把"非-关系"读成"因果之间没有关系"。非关系是一种实实在在的因果，只是逻辑上取了反；判断口径只有一条——果 j 出现的前提是 i 不存在。第二个坑是把 ∨ 和 ∧ 的弧线画反：∨ 是"有一个就够"，∧ 是"全部都要"，考卷上写错一个符号，整张因果关系表就全错。判断三步法：数输入是一个还是多个（一个就用直线或波浪线）→ 多个输入时问"够一个就行还是要全都满足"→ 选 ∨ 还是 ∧。选择题常把"或""与"的适用场景互换来考，案例题则要求按文字描述画出正确的符号。

过渡：到这里，条件与结果之间的四种逻辑关系就认全了。但这还不够——下一页的六个约束符号，才是因果图真正的难点，也是考试最容易失分的地方。

【3.29】条件之间的关系（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是本节最难也最爱考的一页。刚才的与、或、非回答的是"条件成立以后结果怎么变"；这一页的约束符号回答的是另一个问题——"哪些取值组合根本不允许出现"。课件给了六个符号：E、I、O、R、M 和 IR。

第一个，E，互斥（Exclusive）：最多只能有一个条件被满足。注意"最多"，它允许一个都不满足。类比：一张电影票上的座位最多只能卖给一位观众，也可能根本没卖出去。

第二个，I，包含（Inclusive）：至少有一个条件被满足，允许多个同时成立——一份报销单至少要有一位负责人签字。第三个，O，唯一（Only）：正好也只能有一个条件满足，所以 O 等价于 E 加 I：E 是"最多一个"，I 是"至少一个"，O 是"正好一个"——一张选票必须投给一位候选人，且只能投一位。

第四个，R，要求（Require）：满足了条件 a，就要求满足条件 b，有方向。类比：要打印彩色文档，就要求打印机里装了彩色墨盒。

第五个，M，屏蔽（Mask）：满足条件 a，隐含地要求不满足条件 b——方向与 R 正好相反。类比：手机打开飞行模式，就要求移动数据必须关闭——这就是"屏蔽"；软件里就是置灰，选了"到店自提"，"配送地址"就要求不填。

第六个，IR，无关：满足了条件 a，条件 b 就无关重要了。类比：顾客选了"到店自提"，"期望送达时间"填上午还是下午就无关重要。分清这三个：R 是"b 必须成立"，M 是"b 必须不成立"，IR 是"b 随便，都行"。

三步判断法：先问能不能同时成立，不能就是 E；再问是否必须有一个成立，必须就是 I、正好一个就是 O；最后看 a 成立对 b 提的是"必须有""必须没有"还是"无所谓"，对应 R、M、IR。

> **板书：** E 互斥＝最多一个 ｜ I 包含＝至少一个 ｜ O 唯一＝正好一个（O＝E＋I） ｜ R 要求＝a 成立 → b 必须成立 ｜ M 屏蔽＝a 成立 → b 必须不成立 ｜ IR 无关＝a 成立 → b 无所谓 ｜ 判断三步法：能否同时成立？是否必须至少一个？a 对 b 提的是"必须有／必须没有／无所谓"？

提问： 一个下拉框里能选的值是"男、女、保密"。这三个取值之间，应该标 E、I 还是 O？

> **预设回答：** ①答"O，唯一，因为只能选一个"——多数学生会这么答，要追问：那用户不选、保持空白，允许吗？如果业务上允许留空，就是 E 而不是 O；如果业务规定必选，才是 O。②答"E，最多选一个"——也成立，前提是允许不选；说明这个学生抓住了"最多"和"正好"的区别，值得表扬。③答"I，至少选一个"——错，I 不限制上限，那样就允许"既男又女又保密"同时成立，与下拉框的互斥语义不符。追问一句：如果这

个控件是复选框而不是单选下拉框，答案会变成什么？答案是 I（至少选一个）；如果同时要求不许全选，那才回到 E。

易错点 / 考点：三个必考辨析。第一，E 与 O 的差别只在"能不能一个都不满足"：E 允许全不满足，O 不允许。第二，I 与 O 的差别只在"能不能多个同时满足"：I 允许，O 不允许。记住 O＝E＋I 这个等式，考场上可以当场推。第三，R、M、IR 的方向与极性：R 是要求 b 成立，M 是要求 b 不成立，IR 是 b 无所谓；M 常被误读成"a 不成立时 b 才成立"，那是非关系，不是屏蔽。常见题型是判断（"E 表示有且仅有一个条件被满足"——错）、选择（给出业务规则，选对应符号）和案例（在因果图上标注约束）。记忆口诀：E 最多、I 至少、O 正好、R 要有、M 要没有、IR 无所谓。

【思政落点 7】：六个符号训练的是"把话说到没有歧义"的能力。工程上最贵的错误往往不是写错代码，而是需求里一句含混的话被两个人理解成两个意思："最多一个"和"正好一个"只差两个字，落到系统里就是允许与不允许空白的差别。将来写用例、写报告都要像标 E、I、O 一样较真。这种一丝不苟的规范意识，正是金融行业最看重的素养。

过渡：六个符号认完了，光记住不够，我们来看一个把因果图和判定表串起来的完整例子。

【3.30】示例（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把刚才学的符号连起来用一遍。先看这张因果图。它描述的逻辑关系可以写成一个式子：(A or B or C) And (D or E)。意思是：A、B、C 三个条件里有一个成立，并且 D、E 两个条件里有一个成立，结果才成立。图上还标了两个约束——A、B、C 之间标的是 E，也就是互斥，最多只能有一个成立；D 和 E 之间标的是 R，也就是要求，有些取值组合是不允许出现的。

右边这张判定表，就是把左边这张图"翻译"出来的结果。步骤是：先按条件数列出各种取值组合，每个组合占一列；再把每一列代入逻辑表达式算一次，把结论填进"结果"行；结果等于 1 的列，就是真正能触发功能的组合，等于 0 的列是功能没被触发的组合；最后，每一列都可以写成一条测试用例，输入是这一列各条件的取值，预期输出就是这一列的"结果"。

怎么读这张表？我教大家一个不会错的方法——代进去算。(A or B or C) 和 (D or E) 是两个括号，括号里面是"或"，两个括号之间是"与"。比如某一列里只有 A 成立、其余全不成立，那么第一个括号成立、第二个括

号不成立，结果是 0；再比如某一列里 B、D、E 都成立，两个括号同时成立，结果就是 1。不管规格写得多绕，转成判定表之后，任何一列都可以用一次代入验算，对错一目了然。

还要注意这张表里那些"不能出现"的列。A、B、C 互斥，意味着这三列里同时出现两个 1 的组合是非法的；D、E 之间有要求约束，同样有一批组合不允许出现。这些非法列必须先划掉，剩下的列才值得写成用例——这正是上一页那些约束符号存在的意义，也是因果图比单纯罗列组合高明的地方。

板书：逻辑表达式 (A or B or C) And (D or E) ｜ 约束：A｜B｜C 之间 E（互斥） · D 与 E 之间 R（要求） ｜ 判定表读法：一列＝一种组合 → 代入表达式 → 结果 1 表示功能被触发、0 表示未触发 → 先划掉约束禁止的非法列 → 余下每列＝一条用例

提问：五个条件 A、B、C、D、E 每个都是"成立／不成立"两种取值，如果完全展开，判定表应该有多少列？为什么课件这张表列出来的列数明显少得多？

预设回答：①答"2 的 5 次方，32 列"——✅ 计算正确，这是完全组合数，奖励。②答"因为有互斥和要求约束，很多组合根本不可能出现，所以被删掉了"——✅ 这正是本题要点，因果图的价值就体现在这种"先约束、后组合"的裁剪上。③答"因为只挑了有代表性的列"——⚠ 说法含糊：不是"挑代表"，而是"删非法"；有代表性的裁剪是后面 Pairwise 和正交实验法要做的事，两者不要混。追问一句：如果 A、B、C 之间没有互斥约束，列数会变多还是变少？——变多，因为那些"同时成立"的组合也变成合法组合了。

易错点 / 考点：三个易错点。第一，把判定表的列数直接当成 2^n——一旦有约束，列数一定小于 2^n，写答案时要把约束带来的删减说清楚。第二，忘记"结果"行的含义：结果行不是重复条件，而是每条组合的判定结论，考试时漏填这一行等于全错。第三，把因果图和判定表割裂开：正确口径是"因果图提供约束，判定表提供组合"，先画图定约束，再转表列组合，最后逐列出用例。记忆抓手：一图定关系，一表列组合，一列一用例。

过渡：图会画了，表会转了，那现实中我们用不用手画呢？不用——下一页看一个专门做这件事的工具。

【3.31】因果图方法设计工具（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

因果图手工画得多了就会发现，条件一多，图会画得很乱，判定表也容易抄错。所以这个方法是典型"工具比人快"的场景。课件推荐的工具是 BenderRBT 的 Cause-Effect Graphing，网址在 benderrbt.com 上。

看它的界面，就能明白这类工具的价值在哪里。这个工具把画因果图和出判定表连成了一条流水线。例子叫 Check Overdraft Protection（透支保护校验）：画布上用 EXCL、MASK 这类节点表达互斥和屏蔽约束，再用 OR、AND 把多路条件汇成一个结果 Give_OD。

同一份图还能自动生成判定表：TESTS #01 到 #05 一共五条测试，说明工具已经把组合裁过了；Causes 是各个输入条件，Effects 是各个输出结果，单元格里填 T 或 F 就是真和假。

所以这类工具做的事情可以概括成一句话：你只负责把因果关系和约束画对，组合、化简、列表这三件重复劳动交给工具。这正是我们要建立的分工观——人的价值在于把规格想清楚，机器的价值在于不知疲倦地穷举。

板书：工具：BenderRBT Cause-Effect Graphing（benderrbt.com）｜界面三块：左侧画图区（EXCL／MASK／OR／AND／Not 节点）→ 右侧自动生成判定表（TESTS #01–#05 · Causes · Effects · T／F）｜分工：人画关系与约束，工具做组合、化简、列表

提问：工具已经能自动把因果图变成判定表了，那我们还有必要在课堂上手画因果图吗？

预设回答：①答"没必要了，工具更快"——❌要点评：工具只会忠实地执行你画的图，图画错了它给你一张看起来很像样的错表；判断约束该标 E 还是标 O、D 和 E 之间到底是谁要求谁，这些只能由懂业务的人决定。②答"图要人来画，表可以让工具出"——✅这是本页最想听到的答案：把"理解规格"和"机械穷举"两件事分开，前者是人的核心竞争力，后者交给工具。③答"工具能发现规格里的矛盾吗"——这是个很好的追问，可以顺势说：工具能帮你把约束形式化，从而让矛盾暴露得更快，但"这话到底什么意思"仍然要靠需求评审和测试人员的判断。

易错点 / 考点：本页是工具页，直接考的概率低，但有一个知识点容易考判断题：工具的作用是"生成与化简组合"，不是"替你分析需求"；因果图这一步的分析工作仍然是人做的。另外，不要把 BenderRBT 记成"判定表工具"——它是因果图工具，判定表是它的输出。记忆抓手：人手画因，机器算果。

过渡：因果图处理的是"条件是／否成立"的世界。可现实中还有一类问题，条件不是两个取值，而是三个、四个甚至八个——那该怎么办？下一页我们进入 Pairwise。

【3.32】3.3.3 Pairwise 方法（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这是本节第二个重点页。先看课件给的四行骨架。背景：大部分缺陷是在两个变量取值冲突的测试时被发现的。处理的问题：多变量、每个变量多取值的组合数太大——注意"不再是条件成立、不成立两种情况"。定义：确保某个变量取值和另一个变量取值成对出现都会被覆盖。效果：可大幅度降低组合的数量。

翻译成大白话。判定表和因果图面对的条件只有"成立"或"不成立"；可真实软件里，一个因素往往有好几个取值。比如测一个客户端能不能跑：操作系统有 Windows、Linux，CPU 有 Intel、AMD，网络协议有 IPv4、IPv6——三个因素、每个两个取值，完整组合是 2×2×2＝8 条。

课件右边这张 Table 1 就是 Pairwise 给出的答案，只有四条：Windows＋Intel＋IPv4，Windows＋AMD＋IPv6，Linux＋Intel＋IPv6，Linux＋AMD＋IPv4。验一验：OS 和 CPU 这一组，Windows-Intel、Windows-AMD、Linux-Intel、Linux-AMD 四种配对全出现；OS 和协议、CPU 和协议两组同样齐全。三组两两配对共十二种，四条用例全部覆盖，八条压到四条，这就是 Pairwise 的最小规模。

那为什么"两两覆盖就够"？因为经验数据告诉我们，绝大部分缺陷由最多两个因素的相互作用引起。这既是 Pairwise 的依据，也是它的边界——它赌的是"两因素交互"这个统计规律，不是数学完备性。

组合数一大，测试就不是"多花点时间"，而是根本做不完。本例 8 条变 4 条还不算什么；因素更多时，完整组合以指数速度涨到天文数字，而 Pairwise 只按最大两个因素取值数之积的量级增长。

板书： Pairwise（成对／两两组合）｜背景：多数缺陷来自两个变量取值的冲突｜问题：多变量×多取值 → 组合数太大｜定义：任一变量取值与另一变量取值"成对出现"都要被覆盖｜效果：大幅降低组合数｜例：OS（Windows／Linux）×CPU（Intel／AMD）×协议（IPv4／IPv6）＝8 条 → 4 条，三组两两配对全覆盖｜依据的边界：赌"两因素交互"这一经验规律

提问： 刚才那个例子，八个组合压到四条，凭什么说这四条就够？如果一个缺陷偏偏是"Windows＋Intel＋IPv6"三者凑在一起才出现的，这四条能测出来吗？

预设回答： ①答"测不出来，所以 Pairwise 有风险"——✅ 完全正确，要大力表扬：这正是 Pairwise 的适用边界，它不保证覆盖三因素交互。追问：那怎么办？答案是提高强度，做三三组合甚至更强，或者对该组合追加针对性用例。②答"四条已经覆盖了所有两两组合，所以没问题"——⚠ 说法只对了一半：两两覆盖做到了，但三因素交互确实可能漏；要点评"覆盖"这个词必须带强度，说"覆盖了"之前先问清楚是几因素覆盖。③答"那就干脆测完整的八条"——可以，八条确实不贵；但要把成本感建立起来：本例只有八条，换成课件后面那个相机参数的例子，完整组合是三千多条，就不能这么说了。

易错点 / 考点： 三个必考点。第一，Pairwise 的强度是"2"，它保证的是任意两个因素的所有取值配对至少出

现一次，不保证三因素及以上；考试常出判断题（"Pairwise 能保证发现所有缺陷"——错）。第二，Pairwise 与传统边界值、等价类的分工：等价类和边界值解决"单个变量取哪些值"，Pairwise 解决"多个变量怎么组合"，两者是配合关系不是替代关系。第三，算组合数时不要多算：完整组合是各因素取值数连乘，两两组合的规模按最大两个因素取值数之积这个下界估量级。记忆口诀：强度是二，两两必配，三三不保，量级可控。

【思政落点 8】：Pairwise 背后是一种朴素又高级的做事方法：资源有限，不能平均用力，要抓住主要矛盾——绝大多数缺陷由两个因素交互引起，就优先把两两组合测透，这就是"好钢用在刀刃上"。课程设计里也一样：用例只够跑三十条，是平均撒在一百个功能上，还是按风险排序把核心组合打穿？能为取舍讲出理由，才算真正的工程师。

过渡：道理讲完了，我们用课件给的一个更规整的例子，把"两两组合"这四个字亲手算一遍。

【3.33】5. Pairwise 具体示例（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这个例子很短，但很典型。有三个因素：A 有 A1、A2、A3 三个取值，B 有 B1、B2、B3，C 有 C1、C2、C3。课件先抛出一个问题：完全组合测试数是多少？我们算一下——3×3×3＝27 条。

再看右边，课件把 Pairwise 的设计结果列出来了，一共九条：A1B1C1、A1B2C2、A1B3C3、A2B1C2、A2B2C3、A2B3C1、A3B1C3、A3B2C1、A3B3C2。请大家跟我一起验一验这九条是不是真的把两两组合覆盖全了。

先看 A 和 B 的配对：A1B1、A1B2、A1B3、A2B1、A2B2、A2B3、A3B1、A3B2、A3B3——A 的三个取值和 B 的三个取值两两相配，九种组合一种不缺。再看 A 和 C：同样是九种全齐。再看 B 和 C：B1C1、B2C2、B3C3、B1C2、B2C3、B3C1、B1C3、B2C1、B3C2，九种也全齐。也就是说，二十七种配对，九条用例全部覆盖。

进步在哪里？二十七条压到九条，工作量降到三分之一。而且请注意，九条已经不能再少了——因为 A 和 B 这两列本身就有 3×3＝9 种取值配对，任何一组测试想覆盖它们，列数都不可能少于九。也就是说，这九条已经踩到了理论下界，是"最少且够用"的极限设计。

再补一个观察：这九条不是随手挑的，它让任意两列之间的每种配对恰好出现一次，行和列的取值也分布得很均匀。这种"均匀"的感觉，就是下一节正交实验法的出发点。手工挑是很难挑出来的，规模一大必须靠工

具算。

板书：A：A1／A2／A3　B：B1／B2／B3　C：C1／C2／C3 ｜ 完全组合＝3×3×3＝27 ｜ Pairwise＝9 条（A1B1C1、A1B2C2、A1B3C3、A2B1C2、A2B2C3、A2B3C1、A3B1C3、A3B2C1、A3B3C2） ｜ 校验：A×B、A×C、B×C 各 9 种配对全覆盖 ｜ 下界＝3×3＝9，已最优

提问：如果 A、B、C 每个因素都有四个取值，完全组合是多少条？两两组合至少需要多少条？

预设回答：①答"完全组合 4×4×4＝64 条，两两组合至少 4×4＝16 条"——✅ 两个都算对了，说明已经掌握了"完整组合连乘、两两组合取下界"这两套算法。②答"两两组合也是 64 条"——❌ 说明还没理解"配对"这个词，要回到定义重讲：覆盖的对象不是三个取值的整体组合，而是任意两个因素的取值配对，所以规模取决于两个因素，不是三个。③答"16 条肯定够"——⚠ 要提醒：16 条是下界，实际工具生成的条数一般会比 16 条多一些，因为要在同一批用例里同时满足三组配对约束，不一定能正好压到下界；下界用来估量级，不用来当结论。

易错点 / 考点：两个高频错误。第一，把完全组合公式用错地方：完整组合是各因素取值数连乘，本例 3×3×3＝27；不要算成 3＋3＋3 或者 3×2。第二，误以为"两两组合的条数就等于最长的那一列的长度"——本页的正确算法是取两个最大因素取值数之积作为下界。考试常出填空题（求完全组合数）和简答题（给出一组用例，判断它是否满足 Pairwise 覆盖），判断方法只有一条：任取两个因素，把它们的取值配对列全，看是不是每一种都至少出现过一次。

过渡：三因素三水平还只是小场面。我们换成真实相机的设置项，来看组合爆炸到底有多吓人。

【3.34】Pairwise 处理更复杂的情况（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把问题放大到真实产品的规模，对象是一部相机的拍摄参数设置。一共五个因素：曝光值，有 -2.0、-1.5、-1、-0.5 和 +0.5、+1、+1.5、+2，八个取值；白平衡，有白炽灯、荧光灯、闪光灯、阴天等，课件按四个取值计；感光度，有自动、100、200、400、800、1600、3200，七个取值；分辨率，低、中、高、超高，四个；效果，无、负片、灰度、深褐色，四个。

课件算给我们看：完整组合是 8×4×7×4×4＝3584 条。3584 条是什么概念？一条用例假设平均五分钟——包括设置参数、拍一张、看结果、记录，这已经算快的——就要将近三百个小时，一个人不吃不睡干一

个多月。这就是组合爆炸的可怕之处：它不是"多干几天"的问题，而是根本不可执行。

课件给出的 Pairwise 口径是 8×7＝56 条，也就是取取值最多的两个因素相乘；56÷3584＝1.56%——据课件数据，用不到百分之二的工作量，换掉绝大部分的两因素交互风险。

这里我要补一句严谨的话：8×7＝56 是两两组合规模的**理论下界**——曝光值的八个取值和感光度的七个取值本身就要求 56 种配对，任何测试集都不可能少于这个数；而工具实际生成的结果一般会比 56 条略多一些，因为要在同一批用例里同时满足五组两两配对，往往压不到下界。所以大家记住：下界用来估算量级，实际条数以工具输出为准。

把这个例子的教训说透：面对 3584 条，穷举方案在第一轮就被否掉了；而 56 条量级的方案，一个下午能跑完。方法的选择，直接决定了项目做不做得成。

板书：相机参数 5 因素 ｜ 曝光 8 · 白平衡 4 · 感光度 7 · 分辨率 4 · 效果 4 ｜ 完整组合 8×4×7×4×4＝3584 ｜ 课件 Pairwise 口径 8×7＝56，占比 1.56%（据课件数据） ｜ 56＝两个最大因素取值数之积＝理论下界；工具实际出表一般略多于下界

提问：既然 3584 条太多、56 条又可能漏掉三因素交互，那今天下午就要交付，你会怎么安排这批测试？

预设回答：①答"直接跑 56 条"——✅ 可执行，但要点评：先做风险识别，把三因素交互风险最高的区域（比如高感光度＋闪光灯＋某档曝光）单独补几条，这是按风险加权的做法。②答"先随机抽 200 条跑"——❌ 要指出：随机抽样既不能保证两两覆盖，也不能保证风险高的区域被抽到，是"看着省事、实际最浪费"的方案。③答"分成两轮，第一轮跑 Pairwise 全覆盖，第二轮对发现缺陷的模块做完全组合"——✅✅ 这是最职业的答案，业内叫"先粗后细"：第一轮用 Pairwise 全局扫，第二轮在缺陷聚集处做局部穷举。

易错点 / 考点：三个易错点。第一，把 Pairwise 的条数算成"各因素取值数之和"或者"某两个因素的乘积随便挑"——正确做法是取**最大的两个**因素取值数之积作为下界。第二，误以为这个下界就是工具输出的确切条数——下界是估计，实际一般略多。第三，忘记乘除的口径：算占比时分母必须是完整组合数，本例是 3584，不是 56 的某种变形。考试可能出计算题（求完整组合数、求占比）和方案题（给定因素与取值，用 Pairwise 设计并说明依据）。

过渡：有同学可能会问：8×7＝56 这个"两两组合就够"的说法，是谁定的？是拍脑袋吗？下一页我们看经验数据的来源。

【3.35】Pairwise 方法来自于经验数据（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页回答一个关键质疑：凭什么说两两组合就够？课件给的答案是——大部分缺陷是在两个变量取值冲突的测试时被发现的，而不需要测试所有的完整组合。请注意，这不是拍脑袋，背后有一批统计研究。

来看左边这张图。横轴是测试因素个数，从 2 到 10；纵轴是用例条数，从 0 到 1200 以上。图里有两条线：虚线是穷举（Exhaustive），实线是 Pairwise。请看两条线的走势差别——因素个数从 2 增到 10，穷举那条虚线急速上扬，到十个因素时已经冲到上千条以上，这就是我们上一页说的组合爆炸；而 Pairwise 那条实线增长非常平缓，同样的因素规模下，用例数还停留在很低的水平。两条线越拉越开，说明因素越多，Pairwise 省得越狠。这个图要给大家建立的就是"量级感"三个字。

再看右边这张图。横轴是交互强度（Interactions），从 1 到 6；纵轴是累计发现的故障百分比，从 0 到 100。图里有四条曲线，分别来自医疗设备、浏览器、Web 服务器和 NASA 分布式数据库。四条曲线的共同形状是：从 1 起步，到 2 的地方就已经明显跃升——只要覆盖到两个因素的交互，就能发现很大一部分故障；再往后到 4、5、6，四条曲线都逐渐贴近 100，继续增加交互强度的收益越来越小。

两张图放在一起，Pairwise 的合理性就出来了：左边说明"穷举的代价随因素数爆炸"，右边说明"收益在强度到 2 时已经拿到大头"。代价和收益在强度 2 附近达成平衡，这就是工程上选择两两组合的统计依据。具体的覆盖率数字，下一页课件给了一张更细的表；课件的讲者备注还给了两个来源，一个是微软 MSDN 上的材料，一个是 Hexawise 的案例研究，课后可以按这两条线索去查原始数据。

板书：命题：大部分缺陷来自两个变量取值冲突，无需测所有完整组合 ｜ 左图（因素数 2→10）：穷举虚线急速上扬 vs Pairwise 实线平缓上涨 → 组合爆炸 vs 量级可控 ｜ 右图（交互强度 1→6 · 累计故障覆盖率）：到 2 已明显跃升，到 4 以后趋于饱和 ｜ 来源（课件讲者备注）：MSDN 材料 · Hexawise 案例研究

提问：右图里四条曲线在交互强度等于 2 的地方就已经升得很高，那是不是说明强度取 2 一定是最划算的？

预设回答：①答"是，2 就够了"——⚠ 要点评：对一般商业系统成立，但不能一刀切；要看系统类型，图里四条曲线的抬升幅度本身就不一样，有的系统在强度 2 拿到的覆盖率明显低于另外一些。②答"要看系统的重要程度，安全关键系统就该往上加"——✅ 这正是工程判断：医疗、轨交、航空这类系统一旦失效代价极高，宁可多做几轮三因素、四因素组合。③答"曲线到 4 以后就平了，做更高强度没意义"——⚠ 补充：曲线接近饱和是说"平均收益变小"，但对特定高风险模块，那最后几个百分点往往就是最贵的缺陷所在。

易错点 / 考点：本页是数据依据页，考点是两句结论。第一，Pairwise 的合理性来自经验统计，不是数学完备性——所以它有风险、有边界，强度是可以调的。第二，读图能力也是考点：横轴是什么、纵轴是什么、哪条线是穷举哪条是 Pairwise，考试可能给一张曲线图让学生判断"哪条是 Pairwise 的用例数曲线"。常见判断题"Pairwise 组合数随因素数线性增长"——错，它是增长缓慢，但仍随因素数增加，只是相对穷举的指数增长而言非常平缓，表述时不要把它说成"不增长"。

过渡：图上只能看出趋势，具体能覆盖多少故障，下一页课件给了一张硬数据表。

【3.36】Pairwise 方法来自于经验数据 – 续（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页把上一页右边的曲线换成了数字，标题就是"不同强度的组合故障覆盖率"。表的行是交互强度（Vars），从 1 到 6；列是五类被测系统：医疗设备、浏览器、服务器、NASA GSFC、网络安全。

我们逐行看。强度 1，也就是只看单个因素，五类系统的故障覆盖率分别是 66、29、42、68、20——据课件数据。请注意浏览器只有 29、网络安全只有 20，只测单个因素，连三分之一的故障都抓不到。这就是为什么只做等价类、边界值这类单变量方法是不够的。

强度 2，也就是两两组合，五类系统分别升到 97、76、70、93、65——据课件数据。和强度 1 一比，提升非常明显：浏览器从 29 跳到 76，服务器从 42 跳到 70。这组数据是整张表最值得记住的。强度 3 是 99、95、89、98、90；强度 4 是 100、97、96、100、98——据课件数据。到四因素交互，大部分系统已经接近满分；再往下的强度 5、6，只有部分系统还有数据，浏览器和服务器在强度 6 都到了 100——据课件数据。

这张表怎么用？第一，它告诉我们强度是可以选的，2 是性价比的甜点，3、4 是给高可靠系统加保险的档位。第二，它提醒我们强度 2 不是万能：表里网络安全在强度 2 只到 65，浏览器是 76，都还有明显的提升空间，所以对安全类、金融类系统，把强度提到 3 甚至 4，是有数据支撑的合理决策。第三，它也回答了一个常见的疑问：为什么课堂上讲的是 Pairwise 而不是三三组合？因为对大多数应用系统，两两组合已经拿走了绝大部分收益，继续加强度的边际收益下降得很快。

板书：不同强度的组合故障覆盖率（据课件数据）｜强度 1：66／29／42／68／20｜强度 2：97／76／70／93／65｜强度 3：99／95／89／98／90｜强度 4：100／97／96／100／98｜列名：医疗设备・浏览

器・服务器・NASA GSFC・网络安全 | 结论：强度 2 是性价比甜点，强度 3–4 给高可靠系统加保险

提问：这张表里网络安全在强度 2 只有 65，而医疗设备在强度 2 已经到 97。同样做两两组合，为什么差别这么大？这对我们的测试策略意味着什么？

预设回答：①答"说明网络安全的缺陷更多是多个因素一起作用才出现的"——✅ 这正是这张表最深的含义，也解释了为什么安全类软件普遍要求更高的组合强度。②答"数据不准吧"——提醒学生：这是课件引用的公开研究数据，不同被测对象的缺陷分布本来就不一样，覆盖率差异恰恰是研究结论的一部分，不是数据错误。③答"那以后统一都做强度 4 就好了"——⚠ 要算账：强度提高，用例数会显著增加，时间和人力都要跟着涨；正确做法是按模块风险分层——安全、支付这类核心模块上强度 3 到 4，普通展示类模块强度 2 就够。

易错点 / 考点：两个易错点。第一，把这张表里的数字当成"任何系统都适用"的普适结论——它是若干被测系统的统计汇总，正确用法是"看趋势、估量级、按系统类型调整"，不是背下某个具体数字往别的项目上套。第二，把"强度"和"覆盖率"两个概念搞混：强度（Vars）是设计用例时覆盖的因素交互个数，覆盖率是跑完之后发现的故障比例，前者是手段，后者是效果，答题时不要互换主语。本页最常考的是一句话判断："提高组合强度一定能提高故障发现率，但边际收益递减"——对。

过渡：方法讲到这儿，动作层面就差最后一步了——真到项目里，谁也不会手算五十六组配对。所以我们来看看现成的工具都有哪些。

【3.37】Pairwise 方法工具（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是工具清单，课件给的入口是 pairwise.org 的 tools 页面，上面列了二十款工具。我不逐个念，只带大家看懂三件事。

第一件事，看来源。CATS 来自贝尔实验室，OATS 来自 AT&T，TCG 来自 NASA，CTS 和 Intelligent Test Case Handler 来自 IBM，ReduceArray2 来自美国空军，PICT 来自微软。通信巨头、航天机构、军队、软件巨头都在造这类工具——这说明组合测试不是课堂练习题，而是工业界的刚需。

第二件事，看形态和授权。有的标网页版 Web-based，有的标命令行 Command-line，有的标 GUI。授权方面，有商业收费的，也有明确自由的：AllPairs 是 Perl 脚本、免费、GPL，PICT 是命令行、免费，Jenny 免费、公有领域。学生做课程设计，用免费的几款完全够。

第三件事，工具列表里标注了"replaces CTS"，说明这是新一代工具替代老的 CTS。建议大家课后至少挑一款跑一次，把上两页那个相机参数的例子喂进去，看它到底给你出多少条用例。

板书：入口 http://www.pairwise.org/tools.asp（共 20 款）｜来源：Bell Labs（CATS）· AT&T（OATS）· NASA（TCG）· IBM（CTS／Intelligent Test Case Handler）· 美国空军 STSC（ReduceArray2）· 微软（PICT）｜形态：Web／命令行／GUI／电子表格｜授权：商业收费 · 免费（AllPairs—Perl／GPL · PICT—命令行／免费 · Jenny—命令行／公有领域）

提问：如果你的课程设计要测一个后台管理系统的查询页面，条件有六个维度、每个维度三到四个取值。你会用哪一类工具，为什么？

预设回答：①答"用免费的 PICT 或 AllPairs，命令行跑一下，把因素和取值写成文件"——✅ 很务实，命令行工具最适合课程设计这种小规模场景，输入输出都是文本，还能写进实验报告。②答"用网页版，方便"——也可以，网页版不用装环境；但要提醒：把项目参数传到外部网站时要考虑数据敏感性，仿真数据无所谓，真实业务数据就不要外传。③答"我自己写个脚本穷举"——❌ 要点评：把精力花在造轮子上不划算，组合测试的生成算法（贪心、IPO 之类）不是本课程的重点，直接用成熟工具，把时间留给测试设计和结果分析。

易错点 / 考点：本页是工具页，考点很轻，通常只考"知道有这类工具、知道工具的作用是自动生成组合并支持约束"。一个容易记混的点：AllPairs 有两条同名记录，分别来自 Satisfice 和 McDowell，这不是课件出错，而是两个不同实现恰好同名。另外要区分"工具名"和"算法名"：下页我们会看到 IPOG 这类算法，它不是一个独立的软件产品，而是工具里可以选择的生成算法。记忆抓手：工具负责生成，约束必须你自己写。

过渡：工具清单里我只挑一款重点看——微软的 ACTS，它也是课件明确推荐的那一款。

【3.38】Pairwise 工具推荐:ACTS（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

课件在工具清单之后单独用一页推荐了 ACTS。这一页是界面截图，我按三个区域看。

先说"确定因素与取值"这一步：System View 把参与组合的因素列成一棵树，ServerType 服务器型号、OS 操作系统、VMSizeType 虚拟机规格、StorageMirrorType 存储镜像类型、VCardType 网卡类型等十项，这就是"确定因素与取值"。

生成结果在 Test Result 表里：表头就是这些因素名，每一行一条用例，这一版截图排到三十一行以上，另外还有 Statistics 页签看覆盖统计。

中间弹出的 Options 窗口有三个参数：Algorithm 默认是 IPOG（Recommended），另有 IPOG-F、IPOG-F2 和 Base Choice；Strength 强度可选 1 到 6，截图里是 2，也就是两两组合，选 3 就是三三组合；Constraint Handling 约束处理当前是 Mixed。

总结成一句话：因素、取值、强度、约束填完，点一下 Build，几十上百条用例就排好了；人力从排列组合里解放出来，用在把因素和约束想清楚上。

板书：ACTS（微软，课件推荐）｜① System View＝因素树（ServerType／OS／VMSizeType／StorageMirrorType／VCardType／Multi-Path／BusinessSimulate／VMDiskType／VMHostTimeSync／VirtualSwitchConf）② Test Result ／ Statistics＝用例与统计 ③ Options：Algorithm＝IPOG（Recommended）／IPOG-F／IPOG-F2／IPOG-D／Base Choice ・ Strength＝1–6（选 2 即两两组合）・ Constraint Handling＝Mixed ｜ 四步：填因素取值 → 定强度 → 写约束 → Build

提问：ACTS 的 Options 里有两个参数——Strength 和 Constraint Handling。如果 Strength 已经选了 2，为什么还需要 Constraint Handling？它解决的是什么问题？

预设回答：①答"约束是业务规则，跟强度没关系"——✅ 正确。强度决定覆盖几因素的组合，约束决定哪些组合根本不允许出现；前者管"覆盖面"，后者管"合法性"。②答"加了约束可以进一步减少用例数"——✅ 也对，这是约束的副产品；但要点评：约束更是为了保证生成的用例在业务上真的能构造出来，比如"未登录用户不能是 VIP 会员"这种组合，测不了也不该测。③答"两个差不多，都是组合强度"——❌ 混淆了概念，要回到 3.3.3 的定义：Strength 是 t-way 里的 t，约束是条件之间的依赖与互斥，两者一个是数量参数、一个是逻辑限制。

易错点 / 考点：本页是工具页，考点集中在两个词。第一，Strength 就是组合强度，选 2 是两两组合、选 3 是三三组合，这正是前面反复讲的"强度可调"的落地形式；考试可能把它与 Pairwise 的定义连起来考。第二，约束（Constraint）不是可有可无的装饰——不加约束，工具会生成一批业务上不可能存在的组合，白跑一批用例；这与因果图里标 E／I／O／R／M 是同一件事，只是换成了工具能读的写法。记忆抓手：强度管覆盖，约束管合法。

【思政落点 9】：ACTS 界面上默认选中的 IPOG 和一旁的 Recommended，背后是几代研究者积累的算法：用尽量少的用例覆盖尽量多的配对，今天只是一个下拉框。技术创新的价值就在这里：把复杂留给自己，把

简单留给使用者。我们要的正是"站在工具之上"的能力：懂原理与边界，又能借它从重复劳动里脱身，把精力投到设计上。

过渡：Pairwise 讲完了，但它其实是"组合优化"这个大题目的一个解。下一个方法处理的是同一类问题，用的却是一套来自数学的正交表——也就是本节最后一个内容：正交实验法。

【3.39】3.3.4 正交实验法（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这是本节最后一页，也是第三个重点页。正交实验法处理的问题，课件说和 Pairwise 类似；但来路完全不同。

Pairwise 的依据是经验数据；正交实验法的依据是数学。课件写道：它依据 Galois 理论，从大量测试用例中挑选适量、有代表性的点，从而合理安排测试的一种科学实验设计方法（缩写 OTDM）。"Galois 理论"来自代数学，说明它不是为软件发明的，而是从农业、制造业的试验设计借来的。

具体怎么做？三步。第一步，确定因子与状态：因子是变量，状态是取值，取值个数叫水平数。第二步，选一张合适的正交表。第三步，用正交表构造测试数据集，每一行是一条用例，再补一些可疑用例。

正交表长什么样？教材的例子是员工查询：员工号、姓名、邮件地址各有"填"和"空"两种水平，因子数 3、水平数 2，于是选 $L_4(2^3)$。它按"均衡分散"的原则挑出四行，再加一条三项全空直接查询，一共五条，而完全组合是 8 条——这就是"用最小的测试用例集合去获取最大的测试覆盖率"。通用记法：常用 $L_8(2^7)$、$L_9(3^4)$、$L_{16}(4^5)$；$L_8(2^7)$ 里 7 是因子数，2 是水平数，8 是测试次数；选表挑次数最少的，对不上就用"包含"和"组合"找最接近的。

最后一个问题：正交实验法和 Pairwise 什么关系？正交表是一种更强的均衡性要求：它不但要求任意两个因素的取值配对都出现，还要求每列各水平出现次数相同、任两列的水平组合次数也相同，术语叫"均衡分散、整齐可比"。所以正交设计一定满足两两覆盖，反之不成立。代价是行数受规格化表格限制，通常比 Pairwise 条数更多。一句话：Pairwise 是"够用就好"的组合覆盖，正交表是"均衡可分析"的实验设计。

板书：正交实验法（Orthogonal experimental design, OTDM）｜依据：Galois 理论——从大量测试用例中挑选适量、有代表性的点 ｜ 三步：① 确定因子与状态（因子数／水平数，可加权）→ ② 选合适的正交表 → ③ 用正交表构造测试数据集（每行一用例＋补可疑用例）｜记法 $L_8(2^7)$：列数 7＝因子数，2＝水平数，行数 8＝测试次数；常用 $L_8(2^7)$／$L_9(3^4)$／$L_{16}(4^5)$ ｜例：员工查询 3 因子 × 2 水平 → $L_4(2^3)$ 四条＋全空一条

＝5 条（穷举为 8 条）｜与 Pairwise：正交表是更强的均衡性要求，正交设计必满足两两覆盖，反之不成立

提问：既然正交表一定满足两两覆盖，那我们以后是不是都该优先用正交实验法，把 Pairwise 淘汰掉？

预设回答：①答"对，正交表更严格所以更好"——❌要纠偏：更严格不等于更合适；正交表的行数受规格化表格限制，常常要凑成 4、8、9、16 这样的行数，遇到水平数参差不齐（比如一个因素 8 个取值、另一个 4 个）时，往往会多出不少用例，而 Pairwise 加上约束条件能裁得更紧。②答"看情况，条件规整就用正交表，水平数杂乱就用 Pairwise"——✅这是最中肯的答案，可以进一步点出：金融、通信这类行业的老标准里正交表用得多，是因为它的统计性质便于分析主效应；而现代工具化的组合测试大多直接做 t-way 覆盖。③答"两个都做一遍对比一下"——可以，但要提醒课时成本：方法选择本身也要算成本，选型阶段做小规模对比是合理的，正式执行时不能两套全跑。

易错点 / 考点：四个必考点。第一，正交表的记法别记反：$L_8(2^7)$ 里 8 是行数也就是测试次数，7 是列数也就是因子数，2 是水平数；考试最爱在这里设错。第二，不要把正交表当成"随便一张表格"：正交表的核心性质是每一列各水平出现次数相同、任两列的水平组合出现次数相同，这两条不满足就不是正交表。第三，正交实验法与 Pairwise 的关系是本节的高频简答题，标准口径是"正交表是更强的均衡性要求，正交设计必然满足两两覆盖，两两覆盖不一定是正交设计"。第四，步骤别漏：确定因子与状态、选正交表、构造数据集，三步缺一不可，第一步尤其不能省——因素和取值都没定清楚，后面选什么表都是空谈。记忆口诀：先定因子与水平，再挑行数最少的表，最后一行一用例。

过渡：到这里，3.3 节基于组合及其优化的四个方法就全部讲完了——判定表、因果图、Pairwise、正交实验法。它们解决的是同一个问题：多因素多取值时，怎么用最少的用例抓住主要风险。下一节我们换个维度，从"输入怎么组合"转向"代码走了哪条路"，进入基于逻辑覆盖的方法。

【3.40】正交实验法示例（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

上一页我们讲了正交实验法的三个步骤，这一页用一张图把"省"这件事算给大家看。

请大家先看左边那个立方体。它有三个因子 A、B、C，每个因子的水平数都是 3，也就是 A1、A2、A3，B1、B2、B3，C1、C2、C3。图上那些蓝点，是三个坐标轴三个方向全都铺满的组合点。数一数，全面测试是 3×3×3＝27 个用例。27 个用例意味着什么？意味着光这一个查询功能，你就要造 27 组数据、跑 27 遍、对 27 次结果。

再看箭头右边这个立方体：从 27 个点里，只挑了 9 个，正交测试 3×3=9。凭什么挑这 9 个？看右边列出的这 9 行——A1B1C1、A1B2C2、A1B3C3、A2B1C2、A2B2C3、A2B3C1、A3B1C3、A3B2C1、A3B3C2。注意了，这 9 行不是随手圈的，它有两条硬性质：第一，每个因子的每个水平，在这 9 行里出现的次数完全一样，都是 3 次；第二，任意两个因子的水平配对，在这 9 行里恰好出现一次、不重不漏。

第二条件才是关键。这就是"与两两组合一致"这五个字的全部含义——你把它读三遍：正交表的 9 行，正好覆盖了 A 与 B、A 与 C、B 与 C 三个方向上的全部两两组合。用 27 个用例能做八九成的事，现在 9 个用例就够了，而且凭什么只测 9 个是有数学依据的，不是拍脑袋。

所以考试写正交实验法的优点，不要只写"用例少"。要写成两句：一是用例数从全面组合的乘积降到与水平数同量级；二是省下来的每一个用例都不白省，任意两个因子的组合一个不漏。前一句是"省"，后一句是"省得起、省得有据"。

板书：三因子三水平 → 全面 3×3×3=27 ｜ 正交 3×3=9 ｜ 9 行↑ 每水平各出现 3 次 ｜ 任意两因子配对恰好一次 =「与两两组合一致」｜ 优点=用例少 + 两两组合不漏

提问：这 9 行里，因子 A 取了 3 个水平，因子 B 也取了 3 个水平。请你数一数，所有 9 个（A，B）组合在表里出现了几次？再数数（A，C）组合出现了几次？

预设回答：

- 答"各出现一次"：✅ 完全正确，这正是正交表的均衡性。追问：那如果我只挑了 9 行，却没保证这个均衡，还能叫正交实验法吗？——不能，那就只是随机抽样。

- 答"（A，B）出现 3 次、因为每行都有 A 和 B"：❌ 把"出现"理解成了"存在"。点评：（A1，B1）这个具体配对只出现一次，出现在第一行；你不能说 A 和 B 出现了 3 次就算覆盖了，咱们数的是配对的种类。

- 答"从 27 降到 9，省了三分之二"：✔️ 数字对。补一句：因子数和水平数越多，省得越狠，这才是在高维组合场景用它的理由。

易错点 / 考点：三个高频错。第一，把正交实验法和 Pairwise 混为一谈——正交表要求"均衡"，即各水平出现次数相等、各两两配对出现次数相等；Pairwise 只要求每个两两配对至少出现一次，允许不均衡，所以在某些场景 Pairwise 的用例数还能比正交表更少。第二，把"水平数"记成"因子数"——水平是**取值个数**（A1/A2/A3 是 3 个水平），因子是**变量个数**（A/B/C 是 3 个因子）。第三，以为水平数必须相等——不等的时候要用包含与组合的办法选表。判断口诀：**因子是"几个变量"，水平是"每个变量几个值"，行数是"跑几次"。**

【思政落点 10】正交实验法最能体现科学精神的，是它敢给自己划一条可验证的红线：9 个用例不是"差不多够用"，而是"两两组合一个不漏"。省是可以被证明的，不是靠感觉说省。我们做测试也一样——说"测过了"没用，"测了什么、漏了什么、依据是什么"才有用。把依据讲清楚、把结论摆在可核查的标准上，这是科学精神，也是工程师的职业底线。

过渡：手工构造正交表毕竟还是个体力活。下一页我们看看，现成的工具怎么把这张表直接吐出来。

【3.41】正交表生成工具（1.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页很短，但它解决的是"你知道怎么做、但来不及做"的问题——正交表生成工具。

手工挑正交表，因子少、水平少的时候还行；一旦因子数到七八个、水平数还不齐，你会发现自己算不动，而且极容易算错。所以实用做法是：先按上一页讲的步骤确定因子与水平，然后交给工具出表，拿到每一行之后，再把取值映射回具体的业务数据，变成可执行的测试用例。

讲义上给了两个参考地址：一个是 Weibull 网站上的热点话题页，讲正交表怎么读怎么用；PPT 讲者备注里给的是 Minitab 官方帮助文档里"正交设计基础"那一节。Minitab 是统计与实验设计领域的常用商业工具，它可以从因子和水平直接生成正交设计表。另外教材 3.3.4 节里还提到过一个国产工具"正交设计助手 II"，也在做同一件事。

我要给大家一个提醒：工具的定位是替你算表，不是替你决定测什么。因子选哪几个、每个因子分几个水平，这两件事工具永远不会帮你做，它只能吃你给的输入。所以这部分工作做得好不好，责任在你身上。

顺便给一个更省事的信息：如果你要的效果是"两两组合至少覆盖一次"，很多缺陷管理或测试管理工具本身就内置了 Pairwise 生成器，直接用它就行，不一定非要先手工画出正交表。

板书：定位＝替你算表，不替你决定测什么 ｜ 流程：定因子与水平 → 工具出表 → 取值映射 → 生成用例 ｜ 参考：Weibull 正交表话题页 · Minitab 正交设计帮助 · 教材提及「正交设计助手 II」

提问：工具能给我们一张漂亮的正交表。那如果一开始因子就选错了——比如漏掉了"并发用户数"这个影响因素，工具出的表还有救吗？

预设回答：

- 答"没救，得重新选因子再出表"：✅ 对。追问：所以因子选择这一步的输入从哪来？——从规格说明书和业务理解里来，这一步是人的工作。

- 答"不影响，正交表还是有覆盖的"：❌ 典型误区。点评：表本身确实均衡，可它均衡的是一个漏了关键因子的模型；漏掉的因子在哪个组合下会暴露缺陷，这张表永远不会告诉你。

- 答"可以用错误猜测法把漏掉的场景补进去"：✅✅ 非常实务。工具出的表是骨架，可疑场景按经验补几条，正是教材说的"再增加一些没有生成的但可疑的测试用例"。

> **易错点 / 考点：**考试里工具页很少直接出题，但常以概念题的形式出现："下列哪项不是正交实验法的步骤？"或者"正交表的行数代表什么？"记住行数＝测试次数，列数＝因子数，水平数＝每列取值的个数。另一个坑是把"工具生成"等同于"自动化测试"——工具生成的是**用例设计**，跟用例能不能自动执行完全是两件事。

> **过渡：**好，到这里 3.3 节基于组合及其优化的方法就全部讲完了。下一页是本讲的第一次课收口，我们把这两节课走过的路串成一条线。

【3.42】小结（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页是第 3 次课的收口页，请大家翻到笔记第一页，把今天的路回放一遍。
>
> 这张小结图把三条主线摆在一起看：一条是"基于需求的测试方法"，往下分三支。**静态**分单因素和多因素——单因素有等价类划分和边界值分析，多因素有因果分析法、决策表和正交试验法。**动态**有功能图和有限状态机。**其它**就是错误推测法。
>
> 把两节课的路线对着这张图复述一遍。第一节课开场，我们先立起了测试方法的分类框架；接着进入 3.1 节基于直觉和经验的方法，重点展开错误猜测法，配了文件上传那个案例。然后进入 3.2 节基于输入域的方法：等价类划分讲有效与无效等价类怎么分，边界值分析讲上点、离点、内点怎么取。这就是"单因素"这一支。
>
> 第二节从单因素跨到多因素：判定表法把条件桩、动作桩、条件项、动作项摆成一张表；因果图法处理互斥、包含、唯一、要求几类约束；Pairwise 把组合数从乘积压下来；最后是正交实验法。图中"多因素"下挂的三个方法，我们都讲到了。
>
> 现在看最后一个方框：**其它——错误推测法**。它为什么不能塞进静态或动态？前两支都有明确的输入——静态靠规格和结构，动态靠运行和状态；错误推测法靠人的经验与直觉，没有形式化依据。这个位置也在提醒

我们：**这一整页的方法，讲的都是"基于需求"**，从需求、从外部行为、从输入域和组合出发，全部站在软件**外面**看它。

下一次课要换一套完全不同的思路：**打开代码，站到软件里面去看它**——哪些语句真的被执行过？哪些真假分支被走全了？哪些路径被走漏了？这就是 3.4 节基于逻辑覆盖的方法，也是全章考点最密集的一节。作业二的提交要求、格式和评分点请看课件最后的作业页，按时提交。下课。

板书：基于需求的测试方法 → 静态：单因素（等价类划分・边界值分析）／多因素（因果分析・判定表・正交试验） ｜ 动态：功能图・有限状态机 ｜ 其它：错误推测法 ｜ 一句话：全部站在软件**外面**看它 ｜ 下次课：打开代码，站到**里面**看它

提问：错误推测法为什么被单独放进"其它"，而不是归到静态或者动态里？说说你的理由。

预设回答：

- 答"因为它靠经验，没有形式化的依据"：✅ 正是。追问：那它有价值吗？——有，它能在需求不完整、结构无法建模的时候，凭经验直接命中高风险点，所以它常在方法组合里当"补漏"的角色。

- 答"因为它是最差的、不重要的方法"：❌ 误区。点评：工具没有好坏，只有适用；错误推测法在集成测试、回归测试里命中率往往很高，因为老手知道哪块最容易坏。

- 答"因为它既可以静态用也可以动态用"：✔️ 观察得很细，这个说法站得住。补充：它不是按"跑不跑程序"来分类的，所以硬塞进任何一支都不合适。

易错点 / 考点：本节最常考的是**归属题**。请把这张图的层级背成一句口诀：**单因素两个——等价类、边界值；多因素三个——因果、判定表、正交；动态两个——功能图、有限状态机；其它一个——错误推测。** 高频错点：把因果图法划到"静态"以外的分支（错，因果图与判定表同属静态多因素）；把边界值分析说成"多因素方法"（错，它只看单个输入的边界）；把等价类划分和边界值分析说成两个独立方法（更准确的说法是两者是互补的一对，常配合使用）。

【思政落点 11】这张小结图值得多说一句：同一个系统，从"输入域"看、从"组合"看、从"状态"看，得到的是三套不同的测试方案，但说的是同一个软件。这提醒我们，任何单一视角都会留下盲区，能看见自己没看见的东西，靠的是主动换视角。做学问、做工作、看自己都一样——敢于从别人的角度重新审视自己的判断，这就是批评与自我批评的方法论价值。

过渡：休息过后，我们进入本次课的重头戏。下一页是一张章节分隔页：3.4 基于逻辑覆盖的方法。请先把白盒测试里那条"覆盖强度阶梯"记在脑子里。

【3.43】3.4 基于逻辑覆盖的方法（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

休息回来，我们正式进入 3.4 节，基于逻辑覆盖的方法。这是第 4 次课的主线，也是全章考点最密集的一节。

先给这一节一个总的定位。前面 3.1 到 3.3，不管我们用等价类、边界值，还是判定表、因果图、正交表，我们始终站在软件**外面**：给它一组输入，看它给什么输出。软件内部怎么实现的，我们不管。这套方法叫黑盒测试。从这一页开始，我们要把盖子**打开**：直接盯着代码，问一个完全不同的问题——**这段代码，到底有没有真的被跑到过？**

为什么要问这个问题？因为有一类缺陷，黑盒测试从外面极难看见，而它就在代码里明摆着：有一段代码你从来没执行过，那你凭什么说它是对的？测试领域有一句近乎常识的推论——**没有被执行过的代码，里面藏着缺陷的概率远高于被执行过的代码**。所以覆盖率这个指标，本质上是在回答"我们的测试，证据到底铺了多宽"。

这一节我们要逐步建立起七个覆盖标准：语句覆盖、判定覆盖、条件覆盖、判定-条件覆盖、条件组合覆盖、MC/DC 覆盖、基本路径覆盖。请大家注意，这七个不是七个并列的选项，它们是一条**强度阶梯**：越往下，要求越严、用例通常越多、留给缺陷的藏身处越少。我们这节课的任务，就是沿着这条阶梯一级一级往上爬，并且随时回答同一个问题——站在这一级，还有哪种错误是我抓不到的？

最后先敲一句预防针。**覆盖强度高，不等于测试质量一定好。** 一个用例把同一段代码跑一百遍，语句覆盖率是 100%，可它一个新分支都没多碰——覆盖率数字漂亮，测试价值几乎为零。所以这一节学的不是"把数字刷到多少"，而是"我知道这个数字能证明什么、不能证明什么"。

> **板书：**3.4 基于逻辑覆盖（＝白盒）｜视角切换：**从外面看输入输出 → 打开盖子看代码有没有跑到**｜前提：没被执行的代码，是缺陷的高发区｜七级阶梯：语句 → 判定 → 条件 → 判定-条件 → 条件组合 → MC/DC → 基本路径｜警告：覆盖率高 ≠ 测试质量好

> **提问：**一段代码，测试报告上写着"语句覆盖率 100%"，能不能直接推出"这个模块的测试做得很充分"？请给出你的理由。

> **预设回答：**

- 答"不能，覆盖率只说明语句都跑到了，不说明每条分支、每个条件都试过"：✅ 这正是本次课要展开的核心。追问：那报告上写 100% 有没有意义？——有，它至少说明没有"死代码"漏测，但它是一块地板，不是天花板。

- 答"能，100% 就是全覆盖"：❌ 典型错误，也是本节的第一个陷阱。点评：语句覆盖是七级阶梯里最弱的一级，它连判定的假分支都没保证碰过，下一页马上会给反例。

- 答"要看测的是什么模块，核心模块要更高标准"：✅✅ 已经带上了风险思维。补充：安全关键领域常用 MC/DC 作为准入门槛，正是这个道理。

> **易错点 / 考点：**第一，**白盒不等于逻辑覆盖**——白盒测试还包括数据流测试、静态分析等，逻辑覆盖只是其中最常用的一支。第二，覆盖标准的中文名与英文缩写要对上：MC/DC 是 Modified Condition/Decision Coverage，中文常译"修正的条件/判定覆盖"。第三，本条最重要：**覆盖强度递进不是"包含全部"，而是"逐级加严，但有两个方向互不包含"**。递进的主线是语句 ⊂ 判定 ⊂ 判定-条件；但**条件覆盖与判定覆盖互不包含**，这一条我们到 3.52 会专门算给你看。

> **【思政落点 12】**覆盖率是"证据"，不是"目标"，这句话请写进笔记。行业里真实存在的坏做法是：为了让报表好看，挑最容易跑的用例去堆覆盖率，或者把没跑到的分支用排除规则掩掉，报告上仍然是漂亮的一行数字。数字是真的，结论是假的。测试人最值钱的东西不是会写用例，而是说的话可信。如实报告覆盖率、如实写明未覆盖部分和原因，这就是职业诚信，也是日后别人敢把关键系统交给你的前提。

> **过渡：**有了这个总地图，我们把阶梯上的七级先完整列出来看一眼。下一页是结构化测试方法。

【3.44】结构化测试方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页把刚才说的七级阶梯，正式列成一张清单，标题叫"结构化测试方法"。为什么叫结构化？因为这一套方法全部依赖程序的结构：判断在哪、条件怎么构成、路径怎么走。它们看的是代码的组织形态，所以都属于白盒一侧。
>
> 我们从上往下过一遍这七项。第一，语句覆盖，要求每个可执行语句至少被执行一次。第二，判定覆盖，也叫分支覆盖，要求每个判断的真分支和假分支都至少走一次。第三，条件覆盖，要求每个判断里**每个条件的取真取假**都至少满足一次。第四，判定/条件覆盖，把前两者的要求合起来。第五，条件组合覆盖，要求判断里各条件取值的**所有组合**都出现。第六，MC/DC 覆盖，修正的条件/判定覆盖，要求每个条件都能**独立地**影响

一次判定的结果。第七，基本路径覆盖，从控制流的路径角度来要求覆盖。

请大家特别留意第四项和第六项的区别，这是最容易糊掉的地方。判定/条件覆盖的口径很朴素：判定覆盖的要求做到，条件覆盖的要求也做到，两条都满足，就算过了。而 MC/DC 更进一步，它要求"独立性"：其他条件都不变，只翻动这一个条件，判定的结果要跟着变。所以 MC/DC 比判定/条件覆盖强，而且它在航空、航天这类安全关键领域的软件适航标准里是硬性门槛。

那为什么还要讲这套阶梯？因为它是给测试工作定"工作量刻度"用的。同一个小模块，按语句覆盖做可能 2 个用例，按判定-条件覆盖做可能 2 到 4 个，按条件组合覆盖做要 8 个。你在项目里报工时、排计划、谈"测到什么程度算够"，依据就是这张清单。

板书：结构化测试方法（白盒，看代码结构）｜1 语句 · 2 判定/分支 · 3 条件 · 4 判定/条件 · 5 条件组合 · 6 MC/DC · 7 基本路径 ｜易混：判定/条件＝两条要求都满足；MC/DC＝要求"独立影响"

提问：判定/条件覆盖和 MC/DC 覆盖，听起来都像是"判定加条件"。如果只按判定/条件覆盖做，可能漏掉什么类型的问题？

预设回答：

- 答"可能漏掉某个条件单独起作用的情况"：✅ 抓到了要点。追问：为什么"单独起作用"这么重要？——因为真实缺陷常常就是某一个条件的判断写错或写反了，而别的条件的取值恰好把它盖住了。

- 答"两个是一回事，只是叫法不同"：❌ 常见混淆。点评：判定/条件覆盖不要求独立性，只要每个条件真假都出现过、每个判定真假都出现过就行；MC/DC 要求每个条件能独立翻转判定结果，所以 MC/DC 严格强于判定/条件覆盖。

- 答"MC/DC 更难写用例，所以用得少"：✔️ 事实层面成立，但要补一句：在普通商业软件里确实少用，可在安全关键领域它是**准入条件**，不是可选项。

易错点 / 考点：本页的考点是**名词与会话对应**。考试常见的选项组合会让你从七个名字里挑出"不属于结构化测试方法"的一个——注意，像等价类划分、边界值分析、判定表法、正交实验法这些基于输入域和组合的方法，不属于这一列。另一个高频点：**判定覆盖＝分支覆盖**，这两个词在考试里可以互换。记忆口诀：**语句跑一遍，判定真加假，条件各取真与假，判定条件两相加，组合全排上，MD 要独立，路径走基本。**

过渡：七项清单里，有一项和前六项的血缘关系稍远一点，就是最后那个"基本路径覆盖"。下一页我们专门把它和前六项摆在一起对比。

【3.45】逻辑覆盖 vs. 路径覆盖（2.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页处理一个边界问题：逻辑覆盖和基本路径测试，到底是不是一家人？请大家先看讲义上的两段定义。

逻辑覆盖：以程序或系统的内部逻辑结构为基础，分为语句覆盖、判定覆盖、判定-条件覆盖、条件组合覆盖等。关键词是"内部逻辑结构"——它盯的是判断、条件这些**逻辑单元**。

基本路径测试：在程序或业务控制流程的基础上，分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计出测试用例。关键词是"控制流程"和"环路复杂性"——它盯的是**执行轨迹**，而且用圈复杂度这个可算的数，来决定到底要导出多少条路径。

两者的区别，用一句话说清楚：**逻辑覆盖管的是"每个路口有没有走到"，路径覆盖管的是"每条路线有没有走完"。** 前者关心一个判定、一个条件有没有被碰过；后者关心从入口到出口，整条轨迹有没有被走遍，并且路径数量不是随便定的，是由圈复杂度算出来的。

那它们的关系呢？教材的说法很值得注意：基本路径覆盖一般不归为逻辑覆盖，但从它们的密切关系看，可以都统一为逻辑覆盖。也就是说，严格分类时它另立一支，实际应用中它们常被放在一起讲、一起用。

还有一个更开阔的点，请大家写进笔记：教材特别指出，逻辑覆盖**不局限于代码这个层次**，它可以扩展到业务流程图、数据流图，让测试覆盖需求层次的业务逻辑。这句话的分量很重——它意味着上面这七级阶梯不只适用于单元测试的代码，也适用于你画在需求文档里的那张业务流程图。

> **板书：**逻辑覆盖＝基于**内部逻辑结构**：盯判定与条件（语句・判定・判定-条件・条件组合…）｜基本路径测试＝基于**控制流程**：算环路复杂性（圈复杂度）→ 导出基本可执行路径集合 ｜一句话：逻辑看节点，路径看路线 ｜拓展：逻辑覆盖可上移到业务流程图、数据流图

> **提问：**一个模块有 2 个判定，每个判定真假两种情况。逻辑覆盖要你保证"每个判定的真假都出现"；基本路径测试要你做的是另一件事。请你用一句话说出这个差别。

预设回答：

- 答"路径测试要把从入口到出口的走法覆盖到，不是只看单个判定"：✅准确。追问：那走法有多少条、要测几条，靠什么定？——靠圈复杂度算出来的基本路径集。

- 答"路径覆盖就是逻辑覆盖加强版，把所有组合都测一遍"：❌混淆了路径覆盖与条件组合覆盖，而且"所有路径"在有循环时是无穷的。点评：基本路径测试恰恰是要避开路径爆炸，它只要一组线性无关的独立路径，靠组

合就能覆盖全部边。

- 答"两个一样，只是叫法不同"：❌ 两者视角不同，且基本路径测试多了圈复杂度这个量化步骤，下一页之后的 3.57 会专门算。

> **易错点 / 考点：**第一，**"基本"不等于"最简单"**——指的是线性无关的一组独立路径，不是挑最容易跑的那条。第二，路径覆盖不等于"穷举所有路径"，循环会让路径数变成无穷，所以必须用圈复杂度截断。第三，考试里的一句经典判断："基本路径覆盖属于逻辑覆盖。"——严格按教材口径，**一般不归为逻辑覆盖，但可统一看待**。答这类题要写清前提，别硬答"是"或"不是"。

> **过渡：**分类问题解决了，我们正式从上阶梯的第一级开始。下一页：语句覆盖。

【3.46】（代码行）语句覆盖（5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这是阶梯的第一级，也是最常被引用的一句话指标：语句覆盖。
>
> 先看定义，很简单：**设计若干测试用例，运行被测程序，使程序中的每个可执行语句至少被执行一次。** 就这么一句话，没有别的条件了。
>
> 我们把它落到右边这段示例程序上。程序共 9 行：第 1、2 行是变量声明，第 3 行是第一个判断——如果 a 大于 0 并且 b 大于 0，就执行第 4 行的 c＝c/a；第 5 行结束第一个判断；第 6 行是第二个判断——如果 a 大于 1 或者 c 大于 1，就执行第 7 行的 c＝c＋1；第 8 行结束；第 9 行 c＝b＋c 是出口前的最后一句。
>
> 右边这张图就是它的**程序控制流图**。怎么读？节点就是可以被执行的语句或者判定，箭头就是执行的方向。图上有两条分流：从节点 3 出去，一条边直接到 5，另一条经过节点 4 汇到 5——这是第一个判断的真假两条路。从节点 6 出去，一条边直接到 8，另一条经过节点 7 汇到 8——这是第二个判断的真假两条路。最后从 8 到 9 收口。
>
> 现在来设计用例。图中间那个箭头写的是 (a, b, c)＝(1, 1, 2)，后面跟着两个字：**覆盖**。我们来验算这组值能不能把 9 条语句全跑一遍。a＝1、b＝1，第一个判断 a＞0 and b＞0 成立，进入第 4 行算出 c＝2/1＝2；接着第二个判断 a＞1 or c＞1，a＝1 不满足 a＞1，但 c＝2 满足 c＞1，所以成立，进入第 7 行算出 c＝2＋1＝3；最后第 9 行 c＝1＋3＝4。

结论：**一组用例 (1, 1, 2)，语句覆盖率就达到了 100%。** 这条路径经过了节点 3、4、5、6、7、8、9，第一级的门槛就这么低——一个用例就够。

请大家记住这个数字：**一个用例，100%。** 下一页我们就用这个"漂亮"的数字，来找它的麻烦。

板书：语句覆盖＝每个**可执行语句**至少执行一次 ｜ 示例：(a,b,c)＝(1,1,2) → M 真 → c＝c/a＝2 → N 真（c＞1）→ c＝c＋1＝3 → c＝b＋c＝4 ｜ 路径 3-4-5-6-7-8-9，**9 条语句全覆盖，1 个用例** ｜ 地位：七级阶梯的第一级（最弱）

提问：(1, 1, 2) 这一组用例，把两个判断都走成了"真"。那这两个判断的"假"分支，在这一轮测试里被走过吗？

预设回答：

- 答"没走过，两个判断都是真"：✅ 观察正确。追问：那语句覆盖率是多少？——仍然是 100%，因为跳到假分支并不影响后面的语句照常执行。这个反差就是下一页的主题。

- 答"走过，因为程序有两个判断，总会走假的那边"：❌ 把"可能"当成了"已经"。(1,1,2) 这一组具体用例并没有触发任何假分支。

- 答"要看控制流图才能确定"：✅✅ 好习惯。附图说明：从节点 3 直接到 5 的那条边（假分支）、从节点 6 直接到 8 的那条边（假分支），这轮都没走。

易错点 / 考点：三处高频坑。第一，**"每个可执行语句"要排除声明语句**——很多语言里变量声明不产生可执行代码，考卷上如果问"这段程序有几条可执行语句"，要按控制流图上的节点来数，不要照抄行号。第二，把"语句覆盖"和"代码行覆盖"严格等同——日常语境里两者常混用，但在有分支、有循环的语言里，一行可能对应多条语句，考试若同时出现两个说法，以"可执行语句"为准。第三，**语句覆盖是七级里最弱的一级**，选择/判断题里说"语句覆盖是最强的覆盖标准"一定错。判断口诀：**语句覆盖只管"跑到过"，不管"跑得全"。**

过渡：既然一个用例就能把语句覆盖率刷到 100%，那这个 100% 到底值多少钱？下一页我们用同一个例子，让你亲眼看见它能漏掉什么。

【3.47】语句覆盖不能发现的问题（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是 3.4 节最有价值的一页之一，它用一个反例，把语句覆盖的软肋彻底摊开。

左侧是我们刚才那段正确的程序：第 3 行是 `a > 0 and b > 0`，第 6 行是 `a > 1 or c > 1`。右边这段是"如果开发写错了"的版本：注意看图上的标注，第 3 行的 `and` 被写成了 `or`，第 6 行的 `or` 被写成了 `and`。这是真实开发中极常见的一类错误——布尔连接符写反。现在关键的问题来了：我们手里还是那个用例 (a, b, c)＝(1, 1, 2)。请跟我一起算，看它会不会发现这个错误。

> 先跑左边正确的代码，刚才算过：第一个判断真，执行第 4 行；第二个判断真，执行第 7 行；最后第 9 行。语句覆盖率 100%。再跑右边写错的代码。(a, b, c) 还是 (1, 1, 2)。第一个判断现在变成了 `a > 0 or b > 0`：a＝1 大于 0，只要一边成立就成立，所以判断结果是**真**——和正确代码的结果**一模一样**，照样执行第 4 行。第二个判断变成了 `a > 1 and c > 1`：a＝1 不满足 a＞1，`and` 要求两边都成立，所以判断结果是**假**——这里和正确代码的结果不一样了，第 7 行不会执行。
>
> 那语句覆盖率受影响吗？不受。第 7 行本来就是判断为真时才执行的可选语句，判断为假就直接跳到第 8 行、再到第 9 行，仍然是实实在在执行了语句。所以这个用例在错误代码上的语句覆盖率**依然是 100%**，而两个布尔连接符被写反了，一个都没被发现。
>
> 请把结论抄下来，这是本节的第一个考点：**语句覆盖只要求语句被执行，不要求判定的真、假两种结果都被检验，因此它无法发现判定的逻辑运算错误（例如 and 与 or 写反），也无法发现判定内部条件的错误。** 要抓住这类错误，必须上升到下一级——判定覆盖。
>
> **板书：** 正确：`a>0 and b>0` / `a>1 or c>1` ｜ 写错：`a>0 or b>0` / `a>1 and c>1`（and ↔ or 写反） ｜ 用例仍为 (1,1,2)：M 真（or 仍为真）· N 由真变假，但第 7 行是可选语句 → **语句覆盖率都还是 100% → 缺陷未被发现** ｜ 结论：语句覆盖不检验判定真假，抓不到逻辑运算错误

> **提问：** 既然 (1, 1, 2) 抓不到这个错误，请你再给一组 (a, b, c)，让正确的代码和写错的代码跑出不同的结果，从而把这处 `and`/`or` 写反暴露出来。

> **预设回答：**

- 答"(a,b,c)＝(-1, -1, 0)"：✅ 可用。左侧 M 为假、N 为假；右侧 `a>0 or b>0` 仍为假，但 `a>1 and c>1` 也仍为假——注意，这一组也可能区分不出来，说明**单选一组不够，还要看两边配合**。点评：这正是教材强调"必须让判定的真分支和假分支都走到"的原因。

- 答"(2, -1, 6)"：✅✅ 思路对。左：M＝假（b＜0），N＝真（a＞1）；右：`a>0 or b>0`＝真（a＞0），于是第 4 行被执行；`a>1 and c>1`＝真（都＞1），第 7 行被执行。必要语句的执行情况出现差异。追问：这一组其实同时把"条件"层面也动过了，这正是下一页要继续讲的内容。

- 答"随便再补一组就行"：❌ 点评：补用例不是凑数量，得说清"这组要触发哪个判定的假分支、要暴露哪种错误"，说不清就是碰运气。

> **易错点 / 考点：**本页几乎必考。常见题型有三种。① 判断题："语句覆盖率达到 100%，说明程序中的逻辑判断都测试过了。"——**错**。② 案例题：给出代码，问"某组用例达到语句覆盖，能否发现 and/or 写反的错误？"——**不能**，理由要写"该组用例未覆盖判定的假分支"。③ 概念题："要发现判定内部逻辑运算错误，至少需要达到哪种覆盖？"——**判定覆盖**，且教材进一步指出，AND 关系需用"真真"与"真假（或假真）"的组合才能暴露，OR 关系需用"假假"与"真假（或假真）"的组合。判断口诀：**语句覆盖挡不住"符号写反"，因为符号写反不一定改变语句执行不执行，只改变走哪条边。**

> **过渡：**既然问题出在"只跑真分支"，那我们把要求提一级——判定的真和假，都要走到。下一页：判定／分支覆盖。

【3.48】3.4.1 判定／分支覆盖（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是阶梯的第二级，判定覆盖，也叫分支覆盖。请注意讲义标题里写的是"判定／分支覆盖"，这两个词在考试里可以画等号。

先看定义：**设计若干用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次，即判断真假值均曾被满足。** 和语句覆盖比，差别就在最后那句"判断真假值均曾被满足"。这一条正好补上上一页的缺口：每个判定真假都走过，`and`/`or` 写反就会让某个用例的执行结果出现差异。讲义第二句解释了这个名字的由来：**一个判定代表着程序的一个分支，所以判定覆盖也被称为分支覆盖。**

用例怎么设计？右边流程图上面给了一组 (a, b, c)＝(1, 1, 2)，蓝线指向 `c＝c/a`，走的是判断 M 为真的边；下面一组 (a, b, c)＝(-1, 1, 0)，红线走的是 M 为假的边，再到判断 N 看真假。

我们来验算这两组。第一组 (1, 1, 2)：M＝真，执行 c＝c/a；N 中 c＞1 成立，N＝真，执行 c＝c＋1；最后 c＝b＋c，所以 M 取真、N 取真。第二组 (-1, 1, 0)：a＝-1，M 中 a＞0 不成立，M＝假，跳过 c＝c/a；接着看 N＝`a>1 or c>1`，a＝-1 不满足、c＝0 也不满足，N＝假，跳过 c＝c＋1，最后执行 c＝b＋c，所以 M 取假、N 取假。

合起来看：M 真有了、假也有了；N 真有了、假也有了。**两个用例，判定覆盖率 100%。** 和上一级比：语句

覆盖 1 个用例，判定覆盖 2 个，强度涨一级、成本也涨一级——这就是"强度阶梯"。

最后留一个钩子。第一组是 M 真 N 真，第二组是 M 假 N 假。两个判定"一起真、一起假"，会不会留下漏洞？下一页从"条件"这一层来找它。

板书：判定（分支）覆盖＝每个判定的**真分支与假分支**至少各经历一次 → 判定真假值均被满足 ｜ 判定＝分支（一个判定＝程序的一个分支） ｜ 用例：(1,1,2)→M.T.N.T.；(-1,1,0)→M.F.N.F. → **2 个用例，判定覆盖 100%** ｜ 与语句覆盖对比：语句 1 个用例，判定 2 个用例，强度↑成本↑

提问：判定覆盖要求每个判定的真假都走到。而根据上一页教材的提示，如果要专门暴露 `and` 与 `or` 互换的错误，两个用例的真假搭配还不能随便选。请你想想，为什么"两个判定一起真、一起假"这种搭配可能不那么好？

预设回答：

- 答"因为它没有单独检验某一个条件的作用"：✅ 触及关键。追问：判定覆盖管的是整个判定的结果，谁在里面起了作用它并不区分。——对，这就是下一级条件覆盖要解决的问题。

- 答"因为还有别的真假组合没覆盖，比如 M 真 N 假"：✅✅ 说得更细。补充：M、N 两个判定各有真有假，四种搭配我们只走了两种；至于这四种搭配是不是都必须走，那是 3.52 及之后的内容。

- 答"没什么不好，判定覆盖已经够了"：❌ 提醒他回看上一页：(1,1,2) 单独一组就能 100% 语句覆盖，判定覆盖确实更强，但它**不保证每个条件的取值都被检验过**，条件写错仍可能藏住。

易错点 / 考点：三处高频。第一，**判定的真假要"每个判定"分别满足**，不是"整个程序里真假都出现过一次"——考试常给一个只让两个判定都取真的用例组，问是否达到判定覆盖，答案是**没有**。第二，判定覆盖隐含语句覆盖：每个判定真假都走到了，那么判定下的两条语句路径必然都被执行，所以**满足判定覆盖一定满足语句覆盖**（真包含关系），但反之不成立。第三，考试高频结论句：**判定覆盖能发现判定层面的逻辑运算错误（如 and/or 互换），但发现不了判定内部条件的错误。** 判断口诀：**判定覆盖看"整个判定的真假"，不看"里面每个条件的真假"。**

过渡：那么，把镜头再拉近一层，直接盯住判定里的每一个条件，会得到什么？下一页：条件覆盖。

【3.49】3.4.2 条件覆盖（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这是阶梯的第三级，条件覆盖。请注意它的观察单位比判定覆盖又细了一层：判定覆盖看"整个判定"，条件覆盖看"构成判定的每一个条件"。

先看定义：**条件覆盖的基本思想是设计若干测试用例，执行被测程序以后，要使每个判断中每个条件的可能取值至少满足一次。** 请把"每个判断中每个条件的可能取值"这句话咬住——它的要求对象是**条件**，不是判定。

什么叫条件？看这张图就明白了。左边写的是判定 `(a > 0 and b > 0)`，右边用一个大括号把它拆开，里面的 `a > 0` 和 `b > 0` 分别用虚线箭头指向"条件"两个字。也就是说，一个判定可以由若干个条件用逻辑运算符连接起来，这些**不能再拆的布尔表达式**就是条件。在我们这个例子里，判定 M 拆成 `a > 0` 和 `b > 0` 两个条件；判定 N 拆成 `a > 1` 和 `c > 1` 两个条件。

那条件覆盖具体要求什么？要求 `a > 0` 取过真、也取过假；`b > 0` 取过真、也取过假；`a > 1` 取过真、也取过假；`c > 1` 取过真、也取过假。一共四个条件，每个条件两态，算起来是八项要求。

这里请大家先记住一个直觉：**判定覆盖看"结果"，条件覆盖看"零件"。** 一个判定结果错了，可能是里面某一个零件错了；只测结果，你只能知道"这个判定出过问题"，测零件，你才能知道"是哪个条件出过问题"。所以条件覆盖比判定覆盖更细致，用例的构造也更讲究——你不能随便凑两组值，得盯着每个条件的真假去凑。

下一页我们就动手凑，先把这四个条件的真假记号编好号，再去找能同时满足八项要求的用例。

> **板书：**条件＝判定里不能再拆的布尔表达式（`a>0`、`b>0`、`a>1`、`c>1`） ｜ 判定 `(a>0 and b>0)` → 拆成两个**条件** ｜ 条件覆盖＝每个判断中**每个条件的可能取值**至少满足一次（4 个条件 × 真/假 ＝ 8 项要求） ｜ 一句话：判定覆盖看**结果**，条件覆盖看**零件**

提问： `(a > 0 and b > 0)` 这一整块，算一个条件还是两个条件？为什么？

预设回答：

- 答"两个条件，因为里面有两个不能再拆的布尔表达式 a＞0 和 b＞0"：✅ 正确。追问：那 `and` 算什么？——它是把两个条件连起来的逻辑运算符，本身不是条件。

- 答"一个条件，因为它是一个完整的判断"：❌ 混淆了条件与判定。点评：整块是**判定**（判断），拆开的小块才是**条件**；条件覆盖的计数对象是小块。

- 答"要看语言怎么实现"：✔️ 有一定道理，可以提示：本质上按语义上的独立布尔子表达式来拆，和我们后面算 MC/DC 时的拆法是同一套。

易错点 / 考点：第一，**"条件"与"判定"必须分得清**，这是全节的计数基础：本例 2 个判定、4 个条件。第二，

> 条件覆盖的计数是"每个条件的真和假"，不是"每个条件出现一次"。第三，考试常问："条件覆盖能不能保证判定覆盖？"——**不能**，两者互不包含，下一页的示例会给出反例。判断口诀：**判定数"结果的两个面"，条件数"零件的两个面"。**

> **过渡：**光说要求还不够，我们把它写到纸上。下一页我们把四个条件的真假全部编号列出来。

【3.50】示例：列出所有条件（4 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页是做条件覆盖的准备，也是做后面所有覆盖最基础的一步：**把所有条件列全，把每个条件的真、假各起一个名字。**
>
> 先看左边的流程图，两张图放一起对照着看。第一张是原始的流程图：上面一个菱形写 `a>0 and b>0`，真分支（T）指向 `c=c/a`，假分支（F）往下；下面一个菱形写 `a>1 or c>1`，真分支（T）指向 `c=c+1`，假分支（F）往下；最后两个分支合并到 `c=b+c`。第二张是它的抽象版：上菱形标 M，真分支到 K，假分支到 3；下菱形标 N，真分支到 J，假分支到 5；最后是 P。也就是说，**M 就是判定 `(a>0 and b>0)`，N 就是判定 `(a>1 or c>1)`；K 就是 `c=c/a`，J 就是 `c=c+1`，P 就是 `c=b+c`。**
>
> 现在看右边，我们把四个条件一个个点名，每个条件都给它两个编号。判定 M 里的条件 `a>0`：取 .T. 时记为 **T1**，取 .F. 时记为 **F1**。判定 M 里的条件 `b>0`：取真记为 **T2**，取假记为 **F2**。判定 N 里的条件 `a>1`：取真记为 **T3**，取假记为 **F3**。判定 N 里的条件 `c>1`：取真记为 **T4**，取假记为 **F4**。
>
> 请大家把这八个符号抄在笔记本上，抄的时候按这个顺序：**T1、F1、T2、F2、T3、F3、T4、F4**。为什么要编号？因为条件覆盖的要求"每个条件的可能取值至少满足一次"，翻译成编号语言就是：**这八个记号，在你的用例集里全部要出现。** 有了它们，找用例就从"凭感觉试数"变成了"看还缺哪几个记号"——这就是形式化方法的好处，它把判断变成了核对。坐标习惯：取值一律写成 (a, b, c) 三元组，顺序固定。

> **板书：**M＝`(a>0 and b>0)`，K＝`c=c/a`；N＝`(a>1 or c>1)`，J＝`c=c+1`；P＝`c=b+c` ｜ 编号：`a>0`→**T1/F1**；`b>0`→**T2/F2**；`a>1`→**T3/F3**；`c>1`→**T4/F4** ｜ 抄写顺序：**T1 F1 T2 F2 T3 F3 T4 F4**

判定	条件	取 .T.	取 .F.
M	a>0	T1	F1

M　　b>0　T2　　F2

N　　a>1　T3　　F3

N　　c>1　T4　　F4

> **提问：**为什么我们要给"取假"也专门编一个号，比如 `a>0` 取假叫 F1？不就是"没取真"吗？

> **预设回答：**

- 答"因为条件覆盖要求真假都出现，取假和取真一样是一项必须核对的要求"：✅ 说到点上了。追问：如果不编号，你怎么知道漏没漏？——只能靠记，记不住就会漏，这正是条件覆盖最常见的失分点。

- 答"因为假分支也要测"：⚠ 部分对，但概念要注意：F1 说的是**条件 `a>0` 取假**，不是**判定 M 取假**。两件事不能混。

- 答"编号只是为了好看"：❌ 点评：编号是核对清单。后面做条件组合覆盖时，八大组合也是靠这套记号写出来的，没有它寸步难行。

> **易错点 / 考点：**本页必考的是**编号规则本身**。请记牢：**T 表示条件取真、F 表示条件取假；下标的数字是按条件在代码中出现的顺序编的，从 1 开始，一个条件占一个数字，不分判定。**所以本例如：第 1 个条件 `a>0`→1，第 2 个 `b>0`→2，第 3 个 `a>1`→3，第 4 个 `c>1`→4。高频错误有三个：① 把 `a>1` 编成 T1（它在判定 N 里，序号要接着往下排）；② 把 `c>1` 编成 T2（它的序号必须接着往下排）；③ 把 F1 理解成"判定 M 为假"。判断口诀：**数字跟着条件的出场顺序走，T/F 跟着条件的取值走。**

> **过渡：**清单列好了，接下来就是照单找数。下一页我们用两组用例，把这八个记号全部覆盖掉，然后你会发现一个不太妙的问题。

【3.51】示例续：覆盖所有条件（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页我们照着上一页的清单找用例。任务很明确：让 T1、F1、T2、F2、T3、F3、T4、F4 八个记号全部出现。

讲义给了两组值。第一组：**(a, b, c)＝(2, -1, 0)**，覆盖 **T1、F2、T3、F4**。核对：`a>0` 因 a＝2＞0 成立 →

T1 ✅；`b>0` 因 b＝-1 不成立 → **F2** ✅；`a>1` 因 a＝2＞1 成立 → **T3** ✅；`c>1` 因 c＝0 不成立 → **F4** ✅。一组值就落实四个记号。第二组：**(a, b, c)＝(-1, 1, 2)**，覆盖 **F1、T2、F3、T4**。核对：`a>0` 假 → **F1** ✅；`b>0` 真 → **T2** ✅；`a>1` 假 → **F3** ✅；`c>1` 真 → **T4** ✅。

八个记号一个不缺。**条件覆盖 100%。** 但讲义最后那行红字问：**"但有什么问题吗？"** 问题还真不小，而且是这一节最重要的一个反例。

回头看这两组用例在**判定层面**跑成了什么。用例 1：`a>0` 真但 `b>0` 假，`and` 要求两边都真，所以 **M＝假**；`a>1` 真，`or` 只要一边真就真，**N＝真**。用例 2：`a>0` 假 → **M＝假**；`a>1` 假、`c>1` 真 → **N＝真**。

看出问题了吗？**两个用例里 M 恒假、N 恒真。** 条件覆盖 100%，可 M 的**真分支没走过**、N 的**假分支没走过**。结论请抄笔记：**条件覆盖不一定满足判定覆盖。** 反过来，判定覆盖也不一定满足条件覆盖——3.48 那两组用例做到了判定真假全覆盖，条件层面却没有全照顾到。所以这一对标准**互不包含**："谁更强"的答案是**谁也不包谁**。那把两条要求合起来呢？这就是下一页的判定-条件覆盖。

板书： 用例 1 (2,-1,0) → T1 F2 T3 F4 ｜ 用例 2 (-1,1,2) → F1 T2 F3 T4 ｜ 合计 8 个记号全中 → **条件覆盖 100%（2 个用例）** ｜ ⚠ 但：两组里 **M 恒假、N 恒真** → **条件覆盖不保证判定覆盖** ｜ 结论：**条件覆盖与判定覆盖互不包含**

用例	a>0	b>0	a>1	c>1	条件覆盖	判定 M	判定 N
(2, -1, 0)	T1	F2	T3	F4	✔	.F.	.T.
(-1, 1, 2)	F1	T2	F3	T4	✔	.F.	.T.

提问： 我们做到了条件覆盖，可 M 的真分支一次都没走。请你只改动用例 2 里的一个数，让 M 变成真，同时尽量不破坏已有的条件覆盖。

预设回答：

- 答"把用例 2 的 a 从 -1 改成 2"：❌ 这一改，F1 就丢了（`a>0` 变成真，T1 重复），条件覆盖立刻不满。点评：这正是**条件覆盖与判定覆盖冲突**的直观来源——你满足了零件，未必满足结果。

- 答"把用例 2 的 a 改成 1"：✅✅ 可行。a＝1 时 `a>0` 真（此处要注意 F1 由 T1 顶替，需再补一例或调整，

因此要点是"必须重新核对八项记号"）。追问：这说明什么？——说明要同时满足两条要求，往往需要**增加用例数**，这正是判定-条件覆盖成本更高的原因。

- 答"那就再加第三组用例"：✅ 实务做法。补一句：判定-条件覆盖的标准做法就是"两条要求同时满足"，用例数一般会多于单纯的判定覆盖或条件覆盖。

> **易错点 / 考点：**本页是**全章最高频的考点页之一**，务必吃透。三条要记死：① **条件覆盖不保证判定覆盖**（本页反例：M 恒假、N 恒真）；② **判定覆盖不保证条件覆盖**（见 3.48 的用例搭配）；③ 因此 **条件覆盖与判定覆盖互不包含，不存在"谁更强"**。考试常见问法："下列覆盖标准中，与判定覆盖互不包含的是？"答案是**条件覆盖**。判断口诀：**判定覆盖管"整个判定真不真"，条件覆盖管"每个条件真不真"，两套账各记各的，谁也算不上谁的加强版。**

> **过渡：**既然两条要求各管一头，那就索性把两条全写上。下一页：判定-条件覆盖。

【3.52】3.4.3 判定-条件覆盖（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页给出的是"两条都别漏"的方案：判定-条件覆盖。
>
> 先看定义，请逐字读：**判定-条件覆盖是判定和条件覆盖设计方法的交集，即设计足够的测试用例，使得判断条件中的所有条件可能取值至少执行一次，同时，所有判断的可能结果至少执行一次。** 抓两个关键词：一是"**交集**"，二是"**同时**"。条件覆盖做到了，判定覆盖可以漏；判定覆盖做到了，条件覆盖也可能漏。判定-条件覆盖的态度很直接：**两个都别漏，一起来。**
>
> 然后看讲义这张表，它把满足要求的用例列得很清楚，一共两个。第一行，取值 **(a, b, c)＝(2, 1, 2)**：a＝2＞0 → T1；b＝1＞0 → T2；a＝2＞1 → T3；c＝2＞1 → T4，四个条件全取真。分支这一列，`a>0 AND b>0` 两支都真，所以 M＝.T.；`a>1 OR c>1` 也两支都真，所以 N＝.T.。**一组用例同时立了两功。**
>
> 第二行取值 **(a, b, c)＝(-1, 0, 1)**：`a>0` 假 → F1；`b>0` 假 → F2；`a>1` 假 → F3；`c>1` 不成立（注意是严格大于，1 不大于 1）→ F4，四个条件全假。判定层面：`a>0 AND b>0` 两支都假，M＝.F.；`a>1 OR c>1` 两支都假，N＝.F.。
>
> 合起来看：条件层面八个记号全出现 → 条件覆盖 ✔️；判定层面 M、N 都有真有假 → 判定覆盖 ✔️。**两个用例，判定-条件覆盖 100%。**

和上一页对照：上一页同是两个用例、条件覆盖 100%，可 M 恒假、N 恒真；这一页换两组值就补齐了。**用例数还是两个，强度却提了一级。** 最后留一句提醒：这一页的四种判定搭配只走了"都真"和"都假"两种，剩下两种没走到会不会漏缺陷？这是下一级条件组合覆盖要回答的问题。

板书：判定-条件覆盖＝判定覆盖 ∩ 条件覆盖（**交集**，两条要求**同时**满足）｜用例 1 (2,1,2) → T1 T2 T3 T4，**M＝.T. N＝.T.**｜用例 2 (-1,0,1) → F1 F2 F3 F4，**M＝.F. N＝.F.**｜**2 个用例，两账齐平**｜对比 3.51：同样 2 个用例，那页只满足条件覆盖，这页两样都满足

条件	取值（a, b, c）	覆盖条件	分支	覆盖判定
a>0, b>0, a>1, c>1	(2, 1, 2)	T1、T2、T3、T4	a>0 AND b>0	M＝.T. N＝.T.
a⩽0, b⩽0, a⩽1, c⩽1	(-1, 0, 1)	F1、F2、F3、F4	a>1 OR c>1	M＝.F. N＝.F.

提问：上一页和这一页都是两个用例。请你对比一下，为什么同样是两个用例，上一页只算条件覆盖，这一页却算判定-条件覆盖？

预设回答：

- 答"因为这一页选的用例让两个判定也分别取到了真和假"：✅ 完全正确，说到根子上了。追问：那说明覆盖强度不只取决于用例数量，还取决于什么？——取决于**用例怎么挑**，挑之前先想清楚每一组要顶哪几项要求。

- 答"因为这一页的用例更多"：❌ 事实都不对，两页用例数一样，都是两个。点评：这个错觉恰是本节想纠正的——**覆盖率不是靠堆用例堆出来的，是靠核对要求核出来的。**

- 答"判定-条件覆盖就是判定覆盖加条件覆盖，当然更强"：✔️ 结论对，但要说清楚"强"在哪里：它的强，强在**两条要求同时被作为验收条件**，而不是在同一个用例里多做点什么。追问：那它比条件组合覆盖强吗？——不强，下一页见。

易错点 / 考点：三条必记。① **判定-条件覆盖不保证条件组合覆盖**——它只要求每个条件真假各出现一次、每个判定真假各出现一次，并**不要求条件取值之间的各种搭配都出现**；本页两个用例只走了"四条件全真"和"四条件全假"两种搭配，像"a>0 真而 b>0 假"这类组合在判定 M 里就没出现。② 判定-条件覆盖的字面是"交集"，但实务语义是"两条要求各自都被满足"，不要理解成"用例的集合取交集"。③ 考试常见问法："要达到判定-条件覆盖，至少需要几个用例？"——本例是**两个**；但要提醒学生：用例数取决于条件的取值能否被同一组数

据同时满足，条件之间互相约束时，用例数会增加。判断口诀：**条件看零件、判定看结果，两条都要＝判定-条件；搭配不管，组合还早。**

过渡：判定-条件覆盖把"判定"和"条件"两套账合到一起，但还没解决"所有条件组合都要取到"的问题。下一页我们看条件组合覆盖，以及它为什么在真实项目里常常不划算。

【3.53】3.4.4 条件组合测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

接下来是逻辑覆盖里最卖力气的一种——条件组合覆盖，也叫多重条件覆盖，英文 Multiple Condition Coverage，缩写 MCC。

它的基本思想一句话讲清：设计足够多的测试用例，使判断中每个条件的所有可能取值至少出现一次，并且每个判断本身的判定结果也至少出现一次。前半句和条件覆盖听着差不多，真正的差别在后面：它不是简单地要求每个条件都出现"真"与"假"两种结果，而是要求这些结果的所有可能组合都至少出现一次。

拿前面那个判定 M 来看。M 等于 a>0 and b>0，展开成两个条件，每个条件两种取值，两两组合就是四种：真真、真假、假真、假假。课件上用 1 和 −1 表示真和假，写成（1，1）、（1，−1）、（−1，1）、（−1，−1）。这四种组合，每一条都要有用例覆盖。

为什么非要组合到这么细？因为"与"和"或"写错了，单看每个条件的真假是看不出来的。假如代码本来写的是 and，开发者手一抖写成了 or。你只测了真真和假假两条，程序结果居然是对的，缺陷就漏过去了；可你只要补上真假这一条，and 输出假、或者 or 输出真，差别立刻暴露。这就好比医生开药，只测"两种药一起吃"和"两种药都不吃"是看不出问题的，真正的坏反应往往出在只吃某一种的时候。

行业里的例子更直白：编译器解析表达式、计费系统判定优惠条件、权限系统校验访问规则，到处都是复合判断。一个 and 和 or 写反，可能就是一笔费用算错、一次越权放行。

代价也要同时说清楚：一个判定里有 n 个条件，组合数就是 2 的 n 次方。条件越多，用例数是指数级往上翻——这就是它后面要被打折扣的原因。

板书：MCC（条件组合覆盖）＝ 每个条件所有取值至少一次 ＋ 每个判定结果至少一次 ＋ 取值所有可能组合至少一次 ｜ M＝a>0 and b>0 → 4 种组合：（1，1）（1，−1）（−1，1）（−1，−1） ｜ 代价：n 个条件 → 2^n 条用例

提问：判定 M＝a>0 and b>0 有 4 种组合。那么请算一下：如果程序里有两个这样的判定 M 和 N，条件组合覆盖一共要覆盖多少种组合？

预设回答：

- 答"16 种"：❌ 把两个判定的条件混在一起按 2^4 算了。点评：组合是在各自判定内部做的，M 内部两两组合，N 内部两两组合。

- 答"8 种"：✅ 对，就是 4＋4。追问：为什么不是 2×2×2×2 的整体组合？——因为 M 与 N 是先后关系，M 的条件不会和 N 的条件交叉组合。

- 答"看每个判定有几个条件，2 的 n 次方，再相加"：✔️ 思路完全正确，要点明的是"各自算、再相加"。

易错点 / 考点：最大的混淆是把条件组合覆盖和条件覆盖混为一谈。判断三步法：① 每个条件真、假是不是都出现了？② 每个判定真、假是不是都出现了？③ 每个判定内部的取值组合是不是全都出现了？前两条是条件覆盖的底线，第三条才是条件组合覆盖的加码。考点常出计算题：给一个含 n 个条件的判定，问条件组合覆盖至少要多少条用例——答案是 2 的 n 次方。

过渡：组合全都覆盖到了，是不是就等于把程序的每条路径都走到了？下一页我们用一张表来对账。

【3.54】组合覆盖 vs 路径覆盖（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页只有一张表，但它要讲一个很重要的结论：组合覆盖得再全，也不等于路径覆盖。

先看表的结构：第一列是测试用例，第二列是它覆盖了哪些条件的真假，第三列是它走过的路径，第四列是它覆盖的组合编号。四行用例分别覆盖组合 1 和 5、2 和 6、3 和 7、4 和 8——八个组合一个不漏，条件组合覆盖的要求确实达到了。右边那张小图画的也是这件事：菱形 M 走真出口到 K，走假出口落到下面的菱形 N，N 再分真假走到 J 和 P。

现在把目光挪到第三列。第一行走的是 P1，也就是 1-2-4；第二行和第三行走的都是 P3，也就是 1-3-4，重复了；第四行走的是 P4，也就是 1-3-5。回头数一数这个程序一共有几条路径：P1、P2、P3、P4 四条。P2 是 1-2-5，它一次都没被走到。

所以课件在下面写了一行红字：覆盖了所有组合，但覆盖路径有限，1-2-5 没被覆盖。

这说明两件事。第一，最强的逻辑覆盖也不是充分测试，它管的是判定内部的条件组合，管不到判定与判定之间的路径组合；第二，条件和路径是两个不同的维度，一个维度拿了满分，另一个维度可能还是零分。

打个比方：组合覆盖像是把这台机器上每个零件的各种配对都试了一遍，路径覆盖像是把整条产线的走法都跑了一遍。两本是不同的账，一本做得再满，也不能替另一本交差。

还有一句要提醒大家：这里说的"没覆盖"，不是用例写少了，而是这类方法天生管不到那个维度。后面我们既有办法补上它，也要明白补上它的代价。

> **板书：**组合覆盖 8/8 全中 ｜ 路径：P1（1-2-4）、P3（1-3-4）出现两次、P4（1-3-5） ｜ **P2（1-2-5）未被覆盖** ｜ 结论：组合全 ≠ 路径全

> **提问：**这四条用例里，第二行（a，b，c）＝（2，−1，−2）和第三行（−1，2，3）覆盖的条件组合完全不同，为什么走的却是同一条路径 P3？

> **预设回答：**

- 答"因为它们都让 M 假、N 真"：✅ 对。P3 是 1-3-4，对应的正是判定 M 取假、判定 N 取真。

- 答"巧合"：❌ 点评：不是巧合，是必然——路径由一组判定结果唯一确定，而判定结果只看条件组合"合"出来的那个值，不看组合内部是怎么凑的。

- 答"表格写错了"：❌ 提醒：表格没错，这正是本页要你看出的"多对一"关系。

> **易错点 / 考点：**常见错误是认为覆盖强度越高的方法一定覆盖路径越多。逻辑覆盖的强度是就判定内部的条件组合而言的，路径是另一个维度，两者不等价、也不能互相替代。考点形式：给出一张组合覆盖的用例表，问哪条路径没被覆盖——把第三列去重，再和程序的所有路径比差集。

> **过渡：**既然组合覆盖又贵又有重复，那有没有办法把没必要的用例砍掉，同时保住那份严格？下一页我们先把疑点摆到桌面上。

【3.55】问题（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页的标题就叫"问题"，专门用来挑条件组合覆盖的毛病。请带着一个问题听：条件组合覆盖到底哪里不划算？

课件给了两条判词。第一条，条件组合效率不高，有些测试是不必要的。第二条，判定/条件覆盖还不够强。

我们先验证第一条。回到那 8 种组合：组合 3 是 F1、T2，组合 4 是 F1、F2。把它们放在一起看——第一个条件都固定为假，只把第二个条件从真改成假，判定的结果有变化吗？没有，M 都是假。既然结果不变，这两条测试就等于在做同一件事，其中一条是白花的钱。

再看组合 1 和组合 2：第一个条件都固定为真，第二个条件一真一假，结果一个真一个假。这一对测试就有价值，它证明了第二个条件确实能左右判定结果。

用同样的眼光看"或"的关系：组合 5 和 6 放在一起没有意义，组合 7 和 8 放在一起有意义。

这个道理其实和生活里的体检套餐一样：你花了钱做了十项检查，其中有两项是同一件事换个名字，那这两项里就必然有一项是白做的。条件组合覆盖的四组里，恰好也有一组是这种情况。

换个角度说，判断"这一对用例有没有价值"，看的不是它覆盖了几个格子，而是它能不能证明某件事。能让结果翻转的那一对，证明了"这个条件说了算"；结果纹丝不动的那一对，什么也没证明，只是把表格填满了。

结论就出来了：对每一个逻辑运算，真正必要的测试组合只有三组，不是四组。课件在原表基础上做了优化，把这叫"必要的测试条件组合"。第二条判词我们下一页展开。

板书：问题：① 效率不高，有些用例不必要 ② 判定/条件覆盖还不够强 ｜ 无效对：F1・T2 与 F1・F2（M 均为假） ｜ 有效对：T1・T2 与 T1・F2（M 真假不同） ｜ 每个逻辑运算：必要组合 3 组（不是 4 组）

提问：既然"F1、T2"和"F1、F2"这一组测试没意义，那是不是干脆只留"T1、T2"和"T1、F2"两条就够了？

预设回答：

- 答"够了"：❌ 典型错误。点评：只留这两条，F1 这个条件一次都没取到过，条件覆盖都不满足，判定为假的分支也没走到。

- 答"不够，还得有一条让判定为假"：✅ 对。标准答案就是三组：（真，真）→真、（真，假）→假、（假，真）→假。这样每个条件真假都出现过，判定真假都出现过，而且第二个条件能独立影响结果这件事被证据证明了。

- 答"把四组都留着最保险"：⚠ 保险但昂贵。追问：如果有 10 个条件，$2^{10}=1024$ 条，你留得下来吗？——这正是下一页 MC/DC 要解决的问题。

易错点 / 考点：本页的核心结论是"每个逻辑运算的必要测试组合是 3 组"，它是下一页 MC/DC 的入口。注意别把"三组"记成"三个条件"；也别以为被删掉的那组是随便删的——删的判据是"固定其他条件不变、只改变

某一个条件时，判定结果是否翻转"，结果不翻转的才是冗余。选择题常把"F1、T2 与 F1、F2"和"T1、T2 与 T1、F2"对调来迷惑你。

过渡：把"必要的三组"这个结论正式定义下来，就是下一页的 MC/DC——航空软件强制要用的那个覆盖准则。

【3.56】修正条件/判定覆盖（MC/DC）（3 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

修正条件/判定覆盖，英文 Modified Condition/Decision Coverage，缩写 MC/DC。这是本节的最高潮，也是实际工程里最常被写进合同的覆盖率指标。

它由四条要求组成，我们一条条读。第一，每个判定的所有可能结果至少能取值一次——这是判定覆盖的底线。第二，判定中每个条件的所有可能结果至少取值一次——这是条件覆盖的底线。第三，一个判定中的每个条件独立地对结果产生影响——这是 MC/DC 独有、也是最难做到的一条。第四，每个入口和出口至少执行一次——保证没有走不到的语句。

第三条到底是什么意思？说白了就是：对每一个条件，你要能找出两条用例，它们其他条件全都一样、只有这一个条件取值相反，而判定的最终结果也跟着翻转。这时我们才承认"这个条件确实能独立影响判定结果"，它不是摆设，不是那种怎么变都不影响结果的废条件。

用课件那个判定 a>0 and b>0 来验证。先看条件 a>0：用例一（1，1）结果为真，用例二（−1，1）结果为假，只有 a 从真变假、b 都是真，结果翻转了——a 的独立性成立。再看 b>0：用例一（1，1）为真，用例三（1，−1）为假，只有 b 变了、结果翻转——b 的独立性也成立。三条用例就把这个判定的 MC/DC 撑起来了，课件上写的正是"对于含 n 个输入的判定，至少需要 n＋1 条用例"。

这就是它最大的价值：条件组合覆盖光这一个判定就要 2 的 n 次方条用例，MC/DC 只要 n＋1 条，用例数量从指数级掉到线性级。也正因为如此，航空软件质量标准 DO-178C 明确规定：会影响飞机起飞及降落安全性的 A 等级软件必须满足 MC/DC。

板书：MC/DC 四条：判定真假各一次 · 条件真假各一次 · **每个条件独立影响结果** · 入口出口各一次 ｜ 判定 a>0 and b>0 → 3 条用例：（1，1）真、（1，−1）假、（−1，1）假 ｜ 数量：≥ n＋1（线性） vs 条件组合 2^n（指数） ｜ DO-178C A 级软件强制

提问：如果判定写成 a>0 or b>0，也给它设计满足 MC/DC 的用例，三条够吗？请把这三条给出来。

预设回答：

- 答"够，照抄与的那三条"：❌照抄就错了。点评：（1，1）真、（1，−1）真、（−1，1）真——判定结果全是真，连判定覆盖都没满足。or 的性质是一真即真，要翻转结果，必须让另一个条件保持假。

- 答"（1，1）真、（−1，−1）假、（1，−1）真"：⚠方向对但表述不完整，要逐条验证独立性。

- 教师给出的标准答案：三条用例覆盖（真，假）→真、（假，真）→真、（假，假）→假。对 a 验证时固定 b 为假，a 由真变假、结果由真翻假；对 b 验证时固定 a 为假，b 由真变假、结果由真翻假。两个条件的独立性都成立。

易错点 / 考点：三个高频错误。一是把 MC/DC 当成条件组合覆盖的别名——记住 MC/DC 是它的"必要子集"，与运算只要三组而不是四组。二是把"每个条件独立影响结果"理解成"把每个条件都跑一遍"，独立性的判据是"成对用例、只差一个条件、判定结果翻转"。三是记混适用对象：MC/DC 是安全关键软件的强制项，DO-178C 的 A 等级才要求，不是所有软件都必须做到。口诀：判定看两头，条件看真假，独立看配对，入口出口不能漏。

【思政落点 13】航空软件质量标准 DO-178C 明确规定，影响飞机起飞与降落安全的 A 等级软件必须满足 MC/DC，医疗、轨道交通等安全关键领域也采用类似要求。"每个条件都能独立影响判定结果"这条看似苛刻的规定，背后是一句朴素的话：软件里任何一处逻辑写错，都可能对应着天上的一架飞机、手术台上的一位病人。希望同学们由此体会"敬畏生命、安全第一"的工程伦理——严格的标准不是拿来为难工程师的，而是拿来保护使用者的。

过渡：判定内部的条件，我们算是挖到底了。但把视角抬高一层去看整个程序的执行路径，还有一件更重要的事没做——下一页进入 3.4.5 基本路径覆盖。

【3.57】3.4.5 基本路径覆盖（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

从这一页开始，进入逻辑覆盖的最后一块，也是收束全局的一块：基本路径覆盖，Basic Path Coverage，简称 BPC。

定义很直白——设计所有的测试用例，来覆盖程序中基本的执行路径。

这里先解释一个词："基本"。为什么不是"所有路径"？因为程序里只要有循环，路径数量就可能是无穷多条，

你永远也列不完。而"基本路径"是指线性无关的那么一小组，它的条数是有限的、可以算出来的——这个条数就是下一页要讲的圈复杂度。所以基本路径覆盖是"用有限的条数，把所有路径的构成要素都铺到"，是一种务实的折中。

课件这一页重新搬出了前面那张表，标题却换了。我们对比着看：表里 5 条用例覆盖了 P1、P2、P3、P3、P4，四条路径一条不漏，上一页缺失的 P2（1-2-5）被补上了。

但请注意第三、第四列：组合覆盖的是 1、5，1、8，2、6，3、7，4、8。数一下，八个组合也都齐了。为什么多了一条用例就能两头都顾上？因为它把原来重复走 P3 的那两条改成了一条走 P2、一条走 P3，顺手把组合 1、8 补了进来，路径和组合于是同时全了。

所以教材得出结论：采用其中任何一种方法都不能完全覆盖所有测试要求，实际设计用例时应当把不同方法组合起来交叉使用，才能达到最高的覆盖率。

还要交代一句定位：教材说基本路径覆盖一般不归入逻辑覆盖，但从它们密切的关系看，可以统一到逻辑覆盖这个大类里来讲。另外注意一个包含关系：做到了基本路径覆盖，判定覆盖一定也满足，因为每条基本路径都会把某个判定的一个出口走掉；反过来，判定覆盖满足，路径却可能还是漏的。

测试用例	覆盖路径	覆盖条件	覆盖组合
输入 a=2，b=1，c=6｜输出 a=2，b=1，c=5	P1（1-2-4）	T1，T2，T3，T4	1，5
输入 a=1，b=1，c=−3｜输出 a=1，b=1，c=−2	P2（1-2-5）	T1，T2，F3，F4	1，8
输入 a=2，b=−1，c=−2｜输出 a=2，b=−1，c=−2	P3（1-3-4）	T1，F2，T3，F4	2，6
输入 a=−1，b=2，c=3｜输出 a=−1，b=2，c=6	P3（1-3-4）	F1，T2，F3，T4	3，7
输入 a=−1，b=−2，c=−3｜输出 a=−1，b=−2，c=−5	P4（1-3-5）	F1，F2，F3，F4	4，8

板书：BPC＝覆盖程序中基本的执行路径 ｜ 5 条用例 → P1、P2、P3、P4 全覆盖（补上缺失的 1-2-5） ｜ 单一方法都不够 → 组合交叉使用 ｜ 定位：通常不归入逻辑覆盖，但关系密切，可统一讲

> **提问：**上一页四条用例漏掉了 P2，这一页多了一条就把 P2 补上了。那么"多一条用例"和"换一条用例"，哪一种做法更好？

> **预设回答：**

- 答"多一条更保险"：⚠ 保险但成本高。追问：如果路径多到几百条，你是靠加用例还是靠优化用例？

- 答"换一条更好，因为原来那两条本来就重复"：✅ 对。上一页第二、第三条都走 P3，本身就冗余，把它改成走 P2，等于零成本补了一条路径。

- 答"要看覆盖率目标和时间预算"：✅✅ 最接近实务。基本路径覆盖解决"路径漏没漏"，条件组合解决"条件组合漏没漏"，取舍要看这段代码的风险等级——关键组件宁可多测，边缘模块可以少测。

> **易错点 / 考点：**别把"基本路径覆盖"和"覆盖所有路径"画等号。程序里的循环会让路径数量变成无穷多，基本路径覆盖覆盖的是"线性无关的基本路径集合"，条数正好等于圈复杂度，这是下一页的重点。另外记住教材那句判断：任何一种单一方法都不够，方法要组合交叉使用。

> **过渡：**这个"基本路径"到底怎么一条条找出来？下一页给出一个可以照着做的四步流程。

【3.58】基本路径覆盖的设计过程（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页就是基本路径覆盖的操作手册，一共四步，建议直接背下来，后面几页全都是这四步的展开。
>
> 第一步，依据代码绘制流程图。注意是依据代码，不是依据文档——基本路径测试是白盒方法，图的来源是程序真实的控制结构。
>
> 第二步，确定流程图的圈复杂度。走完这一步，我们手里就有了一个数字，这个数字后面会告诉我们到底有几条基本路径。
>
> 第三步，确定线性独立路径的基本集合。注意这里的用词是"基本集合""基本集"，英文 basis set——它是独立路径构成的一个集合，不是随手挑几条路径凑数。
>
> 第四步，设计测试用例覆盖每条基本路径。图算完了、路径列出来了，最后一定要落到用例上，否则前面三步都是纸上功夫。
>
> 这四步可以压缩成一句口诀：画图、算数、列路径、写用例。顺序也不能颠倒——不画图就没法算复杂度，

不算复杂度就不知道要列几条路径，路径没列出来就不知道该写几条用例。

它很像装修前的算料：先量房画图，再算面积，再列出要买哪几样主材，最后才下单。顺序错了，要么买多了浪费，要么买少了停工——测试用例设计也是一个道理。

再补一句为什么要"依据代码"而不是"依据需求"：需求文档写的是"应该怎么做"，代码里藏的是"实际怎么做"。两者一旦不一致，那本身就是缺陷，而只有照着代码画的图，才能把这个不一致暴露出来。

板书： ① 依据代码绘制流程图 → ② 确定圈复杂度 V(G) → ③ 确定线性独立路径的基本集合（basis set） → ④ 设计测试用例覆盖每条基本路径 ｜ 口诀：画图 · 算数 · 列路径 · 写用例

提问： 这四步里，哪一步最容易被跳过？跳过的后果是什么？

预设回答：

- 答"第一步，直接从代码写用例"：⚠ 常见。点评：有经验的测试人员确实能直接看出分支，但代码一长、嵌套一深，人脑就会漏；画图是把结构外化的过程，外化了才漏不掉。

- 答"第二步，觉得算复杂度没必要"：❌ 危险。点评：圈复杂度就是"我到底要准备几条路径"的答案来源，不算它，你根本不知道自己准备够了没有；它还顺带告诉你这段代码多复杂、出错概率有多大。

- 答"第四步，路径列完就交差"：❌ 点评：测试的最终产物一定是可执行的用例和可判定的预期结果；只交一张路径图，等于给出了方案没给出证据。

易错点 / 考点： 四步的顺序和名称是默写题的常客，注意第三步的关键词是"线性独立"和"基本集合"。还要分清两个概念：圈复杂度算出来的是独立路径的**条数**，不是路径本身；同一个程序，基本集**不唯一**，但里面路径的条数是固定的。判断题常见说法"基本路径集合是唯一的"——错。

过渡： 理论讲完了，我们拿一段真实代码走一遍全程。先看代码和它对应的流程图。

【3.59】示例 – 从源代码到流程图（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把"映射"这个动作画出来了：一段伪代码、一张对应的程序流程图，中间那个大箭头就是映射本身。

先读代码。过程名叫 process records，一共 11 行。第一行 Do While records remain 是个循环；第二行 Read record 读一条记录；第三行 If record field 1 ＝ 0 判断第一个字段是不是零，是的话第四、五行把记录

存进缓冲区、计数器加一；第六行是 Else If record field 2 ＝ 0，也就是第一个条件不成立时再看第二个字段，成立就 reset counter；第八行 Else store in file，前两条都不成立就写文件；第九、十行是两个 End If，第十一行 End Do 收循环。

再看右边的图，关键在于记住红色编号和左边行号的对应：圆圈 1 是循环判断，方框 2 是读记录，菱形 3 是第一个 If，方框 4、5 是存缓冲区和计数器加一，菱形 6 是那个 Else If，方框 7 是 reset counter，方框 8 是写文件，9、10 这两个黑点是把支路合回来的汇合点，11 是出口。

请大家特别盯住三处。第一，菱形代表判断，方框代表处理，代码里有几层判断，图里就有几个菱形。第二，9 和 10 本身不做任何事，只是"多条路并成一条路"的汇合点，但它们在路径表达式里必须算进去。第三，从 10 有一条线绕回 1，那就是循环的回边——正因为有它，这个图里才有了"环"，圈复杂度才有得算。

板书：代码行 1→节点 1（循环判断） ｜ 2→2（读记录） ｜ 3→3（菱形 If） ｜ 4、5→4、5 ｜ 6→6（菱形 Else If） ｜ 7→7 ｜ 8→8 ｜ 9、10→汇合点 ｜ 11→11（出口） ｜ 三盯：菱形＝判断、方框＝处理 ｜ 汇合点不做处理但计入路径 ｜ 10→1 是循环回边

提问：代码明明只有 11 行，图里也编到 11 号；那为什么后面算出来的圈复杂度是 4，而不是 11 或者别的数？请你先猜一个理由。

预设回答：

- 答"因为判断只有几个"：✅ 方向对，但说法要精确——图里的判断节点是 1、3、6 三个，圈复杂度等于判定节点数加一，正好 4。

- 答"因为有 11 行代码"：❌ 点评：行数和路径数是两回事，顺序执行的一百行代码也可能只有一条路径。

- 答"有循环所以路径无穷多"：⚠ 没错，路径确实可能无穷多，所以基本路径覆盖不去数无穷，而是数"线性无关的基本路径"，这正是圈复杂度的意义。

易错点 / 考点：本页最常考的是代码行与流程图节点的对应关系，三个坑尤其要防：① 把两个 End If 当成节点（它们是配对的收尾，不是独立处理）；② 漏掉 9、10 这两个汇合点，导致后面边数、节点数全算错；③ 忘了 10→1 这条回边，漏了它圈复杂度就少 1。看图三步法：定节点、定边、定回边。

过渡：图是画出来了，但这么绕的一张图不好数也不好讲。下一页我们先把它简化一下。

【3.60】流程图简化（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页做的是减法：把流程图里那些只会顺次往下走、不会产生分叉的节点合并掉。

左边是原图，右边是简化后的图，我们对照着看。原图里 2 号是 Read record，3 号是菱形判断，2 完成之后无条件进入 3，中间没有任何别的出路，所以这两个节点可以合并成一个，课件写成"2,3"。同样的道理，4 号存缓冲区之后必然进入 5 号计数器加一，中间也没有分支，于是合并成"4,5"。合并之后，右边这张图就剩 1、2,3、6、7、8、9、10、4,5、11，一共 9 个节点。

有同学会担心：合并了会不会把路径弄丢？不会。因为这些节点组成的是一条"独木桥"，凡是经过 3 的路径必然经过 2，合并只是把两段必然相连的边压成一个点，路径条数和判断逻辑一点没变。

简化的好处马上就能看出来。右边这张图点少、边清，数节点、数边、看区域，眼睛一扫就能数完；而左边那张图框套框、线绕线，人工数边极易出错。这也正是工程上的常见做法——先把图整理干净，再去算复杂度。就像看地图找路，先把小胡同隐掉，主干道立刻就清楚了。

但请注意：简化只动"画法"，不动"逻辑"。判断节点一个都不能少，汇合点和回边一条都不能丢；只要你合并的是那些无条件相连的执行节点，结论就完全等价。

板书：合并规则：顺序执行、无分叉的相邻节点 → 合并为一个节点 ｜ 2＋3 →「2,3」 ｜ 4＋5 →「4,5」 ｜ 简化后 9 个节点：1 ｜ 2,3 ｜ 6 ｜ 7 ｜ 8 ｜ 9 ｜ 10 ｜ 4,5 ｜ 11 ｜ 合并不改变路径条数与判断逻辑

提问：既然顺序执行的节点可以合并，那 9 号这个汇合点和 10 号能不能也并成一个节点？为什么？

预设回答：

- 答"能，它们都在中间"：❌ 点评：9 收的是 7 和 8 两条支路，10 收的是 9 和 4,5 两条来路，来路不同，合并就抹掉了路径信息。

- 答"不能，因为 4,5 到 10 那条边要单独接进来"：✅ 对，判断依据是结构作用是否等价。

- 教师补充：合并的唯一判据是"这两个节点之间只有一条无条件的边"，只有满足它才敢并。

易错点 / 考点：简化是手段不是目的。考试里如果题目给的就是简化后的图（比如本页右图），你必须按简化后的节点数和边数算圈复杂度，不能自己再加回去。另外记住简化后的节点名写法「2,3」「4,5」，路径写法里也要照写，不要拆成两个。判断题常见陷阱：说"流程图简化会降低路径覆盖率"——错，简化前后路径集合

是一一对应的。

过渡：图干净了，接下来就可以请出这一节最重要的一个数字：圈复杂度。

【3.61】计算圈复杂度（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

圈复杂度，英文 Cyclomatic Complexity，是代码逻辑复杂度的度量。课件的定位非常明确：它提供了被测代码的路径数量，而且复杂度越高，出错的概率越大。

先看中间那张直方图，横轴是 V(G)，纵轴是模块数量。它长成一个明显的右偏分布——绝大多数模块的圈复杂度都挤在左边很低的几格里，右边拖出一条长尾，被阴影标出来的正是高复杂度模块。工程上这张图的含义是：少数几个又大又绕的模块，承担了大部分出错风险。所以圈复杂度的第一个用途，是帮你把评审和测试的注意力集中到那条尾巴上；很多团队干脆给圈复杂度定一条警戒线，越线的模块必须重点评审、必须拆函数。

下面三条公式是本页核心，它们是同一个量的三种算法。第一条，V(G) 等于区域数量——这里的"区域"是由节点和连线围起来的封闭形状，并且一定要把图形外部那一整片也算作一个区域，漏了它结果就少 1。第二条，V(G) 等于连线数量减节点数量再加 2，也就是 E 减 N 加 2。第三条，课件写的是"简单可预测节点数量加 1"，也就是教材上说的判定节点数加 1——数一数图里有几个会产生分支的判断节点就行了。

课件还给了一个很实用的讲者提示：算圈复杂度不一定非要画出流程图，直接数代码里的条件语句也可以；而且复合条件要按布尔运算符个数来拆，比如 a>0 and b>0 这种带一个运算符的复合条件，要算 2 个判断，记作"布尔运算符个数加 2"，也就是 1 加 2 等于 3。

举个教材上的小例子对照：一个 5 节点、6 条边的图，V(G) 等于 6 减 5 加 2，等于 3；数判定节点正好 2 个，2 加 1 也等于 3。两种算法互相印证——这就像体检时几个指标互相验证，一个数看着不对，另一个数马上能提示你重新测。

板书：圈复杂度 V(G)＝代码逻辑复杂度的度量 → 提供路径数量，越高越易错 ｜ ① V(G)＝区域数（含图形外部区域） ② V(G)＝E－N＋2 ③ V(G)＝判定节点数＋1 ｜ 不开图：数条件语句；复合条件按「布尔运算符个数＋2」拆 ｜ 教材例：6－5＋2＝3，判定节点 2＋1＝3

提问：一段代码里写了 if (a>0 and b>0)，这个复合条件该算 1 个判断节点，还是 2 个，还是 3 个？请结合课

件给的算法算一算。

预设回答：

- 答"算 1 个，它就是一个 if"：❌典型错误。点评：按判定节点数你确实只数到一个 if，但这样一来条件里的两个子判断就被掩盖了。

- 答"算 2 个，因为它有两个条件"：⚠ 接近但不合课件口径。课件讲者提示的口径是"布尔运算符个数加 2"，这里一个 and，1 加 2 等于 3。

- 教师讲评：两种口径都要会说——按"判定节点＋1"是图形口径，按"布尔运算符＋2"是数条件语句时的口径。答题时写清你用哪一种，避免自相矛盾。

易错点 / 考点：三个必背的坑。① "区域数"必须包含图形外部区域，这是最常见的少算 1 的原因。② E－N＋2 里的 2 是常量，别丢，也别写成加 1。③ 复合条件按运算符个数展开，这是拿满分的关键细节，计算题里只要出现了 and 或 or，就先把它折成判断数再算。验算技巧：三种算法结果必须一致，不一致就是数错了。

过渡：公式会用了，我们把刚才那张简化后的图代进去，亲手算一遍。

【3.62】圈复杂度计算示例（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页就是把上一页的三条公式，全部套到我们前面那张简化图上，看它们是不是给出同一个答案。

先看区域法。课件用"数区域"的办法把答案直接算了出来：这张流程图被围成四块封闭区域——1→2,3→4,5→10 再回到 1 围出第一块，中间一大块，6、7、8、9 围出一个小菱形块，最外层再算一块，合计四块，圈复杂度就是 4。四块区域，所以 V(G) 等于 4。注意这里已经数进了图形外部的那一大片区域，否则你只会看到 3 块。

再看边数减节点数。简化后是 9 个节点，我们数边：1 到 2,3、2,3 到 6、2,3 到 4,5、4,5 到 10、6 到 8、6 到 7、7 到 9、8 到 9、9 到 10、10 到 1、10 到 11，一共 11 条。换成公式再算一遍：11 条边减 9 个节点加 2，同样等于 4。两种算法互相印证。

最后用判定节点法验算：图里的判断节点是 1（循环判断）、3（第一个 If，已并入 2,3）、6（Else If），共 3 个，3 加 1 等于 4。

三种算法，同一个答案：V(G) 等于 4。数边时有个小技巧：先把每个节点出发的箭头数一遍，再和进入每个节点的箭头数对一遍，出入对得上就说明没漏边。这个 4 的含义非常重要——它告诉我们，这段代码的线性独立路径有 4 条，也就是说，下一节我们至少要准备 4 条测试用例。用得多了你会发现：一个模块的圈复杂度一旦上到两位数，基本路径测试的用例成本就很难接受了，这时候要做的是先拆分模块，而不是硬写用例。

算法	代入过程	结果
区域数法	Region 1、Region 2、Region 3、Region 4（含图形外部区域）	4
边点法	E－N＋2 ＝ 11 － 9 ＋ 2	4
判定节点法	判定节点 1、3、6 共 3 个 → 3 ＋ 1	4

板书：区域法 → V(G)＝4 ｜ 边点法：11－9＋2＝4 ｜ 判定节点法：3＋1＝4 ｜ **结论：独立路径 4 条 → 至少 4 条用例**

提问：如果数边的时候漏掉了 10 到 1 这条循环回边，V(G) 会算成多少？这个错误会造成什么后果？

预设回答：

- 答"会算成 3"：✅ 对，只有 10 条边时，10 减 9 加 2 等于 3。

- 追问后果：会少列一条独立路径，直接后果就是测试用例少一条，某段代码可能一次都没被执行到。

- 答"反正有 Region 4，不影响"：❌ 点评：三种算法必须互相印证，一个算法出问题，另外两个能把它抓出来——这就是课件要同时给三条公式的原因。

易错点 / 考点：计算题的标准解法是"先用区域法或边点法算出数，再用判定节点法验算"，两个数不一致就回头重数。最常见的三个失分点：漏算图形外部区域、漏掉循环回边、把已经合并的 2,3 或 4,5 又拆成两个节点。另外要背下结果的物理含义：V(G) 既等于独立路径条数，也约等于所需测试用例的最少条数。

过渡：拿到 4 这个数，接下来就是把这 4 条独立路径一条条写出来。

【3.63】确定独立路径集合（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页要立住两个名词：独立路径和基本集。

独立路径的定义是：至少引入一系列新的处理语句或条件的任何路径。这句话的关键词是"至少引入新的东西"——这条路径和已经挑出来的路径比起来，不能是别人的翻版，它必须带来至少一处此前没走过的语句或判断。

由独立路径构成的集合就叫基本集，英文 basis set。课件给了它一条非常重要的性质：由基本集导出的测试用例，保证每行代码语句至少被执行一次。也就是说，基本集不只是"看起来不重复"，它自带一个覆盖保证。

这个"独立"可以类比成给一栋楼规划疏散路线：每条路线都得带出一段别人没走过的通道，才算一条新路线；如果两条路线从头到尾重合，那第二条就没有存在的意义。

还有几句实务提醒，都是踩过坑的经验。第一，基本集合不一定唯一——同一个程序可以给出好几组不同的基本集，答案不唯一，别一看自己的答案和参考书不一样就慌。第二，不需要活动图，但最好绘制程序流程图——流程图是把结构可视化的最低成本工具。第三，最好每个单元都进行基本路径测试，对关键组件则是必要的——这句话点明了投入的优先级。

课件还给了操作提示：确定基本集时，可以先把判定节点标出来，因为每一条新的独立路径，往往就诞生在某个判定节点的一个此前没走过的出口上。

最后提醒一句它的适用范围：基本路径覆盖最适合"单元"这一级，也就是一个函数、一个模块。模块越大，圈复杂度越高，要准备的用例就越多；这也是为什么实务中会先做基本路径测试，再由上层的集成测试和系统测试去覆盖模块之间的交互。

> **板书：**独立路径＝至少引入一系列新的处理语句或条件的路径 ｜ 基本集＝由独立路径构成的集合 ｜ 性质：导出的用例保证每行语句至少执行一次 ｜ 要点：基本集不唯一 · 不必用活动图但最好画流程图 · 每个单元都做，关键组件必须做 · 先标出判定节点

提问：你照着课件的方法列出了 4 条独立路径，同学列出了另外 4 条，两条路线完全不一样。谁对谁错？

预设回答：

- 答"肯定有一个错了"：❌ 点评：基本集不唯一，条数相同、各自都满足"引入新语句"这个条件，就都对。

- 答"都对，条数都是 4"：✅ 对。补充判据：条数必须等于圈复杂度，且每条路径相对前面几条都能带来新的语句或条件。

- 答"谁的用例更容易构造谁的好"：✔️ 这是更高一层的思路——在保证 4 条的前提下，选一组更容易构造出输入数据的，工程上更划算。

易错点 / 考点：三个易错点。① 把"独立路径"理解成"互不相交的路径"——错，独立是线性无关的意思，两条路径完全可以共用大部分节点。② 把"基本集不唯一"理解成"条数也可以不一样"——错，条数恒等于 V(G)。③ 以为基本路径覆盖必须用活动图——错，课件说的是不需要活动图，但最好画程序流程图。判断三步法：条数对不对？每条有没有新东西？每条能不能落到可执行的用例？

【思政落点 14】基本集导出的用例，是要拿来说"这段代码我确实测到了"的证据。这里有一条职业底线值得强调：覆盖率是结果的度量，不是可以凑出来的数字。为了把覆盖率刷上去而写空用例、删掉断言、把没跑到的地方标成"不适用"，短期看报表很漂亮，长期看是在给用户埋雷。测试报告上的每一个百分比，背后都应当能追到具体用例、具体路径、具体证据——这就是测试人的诚信。

过渡：有了基本集，最后一步就是给每条路径配上能真正跑出这条路的输入数据。

【3.64】示例：基本路径测试用例（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页把前面所有准备工作收口，给出这段代码的 4 条基本路径，并要求保证每条基本路径被执行一次。左边那张图上用不同颜色的粗线把每条路径描了一遍，右边就是路径清单，可以对照着看。

我们一条条念。Path1 是 1-2-3-6-7-9-10-1-11。走法是：进循环读一条记录，第一个字段不是零，走 3 的假出口到菱形 6，第二个字段是零，于是执行 7 号 reset counter，经过汇合点 9、10 回到循环头，处理完所有记录后从 11 退出。

Path2 是 1-2-3-6-8-9-10-1-11。它和 Path1 只差一处：两个字段都不是零，所以走 8 号 store in file。

Path3 是 1-2-3-4-5-10-1-11。第一个字段就是零，走 3 的真出口，执行 4、5，再经 10 回到循环。

Path4 是 1-11，最短的一条——循环体一次都没进，也就是 records remain 一开始就是假，一条记录都没有。

这 4 条路径正好对上 V(G) 等于 4，一条不多一条不少。注意 Path1 和 Path2 的区别只在 7 和 8，Path3 的区别在 4、5——它们各自引入了前几条没走过的处理语句，独立性就是这么体现出来的。落成用例时，关键是构造输入让程序真的走这条路，并且给出可判定的预期输出；只有输入没有预期结果，用例是没法执行的。

顺便说一个实务习惯：路径清单和用例最好一一对应编号，比如 Path1 对应用例 TC-01，这样万一某条路径执行失败了，你能准确说出是哪条路径出了问题，而不是对着几百条用例发呆。

路径	途经节点	走这条路的输入条件	预期结果
Path1	1-2-3-6-7-9-10-1-11	字段 1 ≠ 0、字段 2 = 0	执行 reset counter，计数器归零
Path2	1-2-3-6-8-9-10-1-11	字段 1 ≠ 0、字段 2 ≠ 0	执行 store in file，记录写入文件
Path3	1-2-3-4-5-10-1-11	字段 1 = 0	执行 store in buffer 与 increment counter
Path4	1-11	记录集为空，循环条件一开始即为假	循环体不执行，直接结束

> **板书：**Path1：1-2-3-6-7-9-10-1-11（字段 2=0 → reset counter）｜Path2：1-2-3-6-8-9-10-1-11（写文件）｜Path3：1-2-3-4-5-10-1-11（字段 1=0 → 存缓冲区、计数）｜Path4：1-11（无记录）｜4 条 = V(G)｜要求：每条路径至少执行一次

> **提问：**Path4 只有 1 到 11 两个节点，短得像是凑数的。这条路径有必要单独写一条用例吗？不给它设计用例会漏掉什么？

预设回答：

- 答"没必要，它什么都没执行"：❌ 典型错误。点评：正因为"什么都没执行"才是缺陷高发区——空集合、空文件、零记录这类边界输入，恰恰是程序最容易崩的地方，很多系统就是被"零条记录"打挂的。

- 答"有必要，它测的是循环一次都不进的情况"：✅ 对。它是四条路径里唯一覆盖"循环入口判断为假"这一分支的用例。

- 答"不仅要有，还应该再加边界数据"：✅✅ 更进一步。教材专门提醒过，这段代码还要考虑对取值边界的设计，路径覆盖之外还得补上输入域的边界值测试。

> **易错点 / 考点：**本页最容易失分的地方是路径写法——必须把汇合点（9、10）和回边（10 到 1）原样写进路径串里，少写一个节点整条路径就错了。另外两个高频判断：① "基本路径覆盖能覆盖所有条件组合"——错，教材明确说它不能保证覆盖所有条件组合，要跟条件组合覆盖交叉使用；② "Path4 是冗余路径"——错，空输入是最需要测的边界场景。

> **过渡：**到这里，从语句覆盖一路走到基本路径覆盖，逻辑覆盖的全家福就齐了。最后一页我们把它们排成一

> 张图。

【3.65】逻辑覆盖小结（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页是本节的总账，请把这张图记进脑子里。
>
> 中间这条链讲的是覆盖强度的递进关系，从下往上依次减弱：条件组合覆盖 MCC 最强，往上是条件/判定覆盖 CC/DC，再到判定覆盖 DC，最上面是语句覆盖——越往上要求越低、越容易满足。箭头朝上，表示下面这层能覆盖的，上面那层也能覆盖——强度一层一层递减，这就是逻辑覆盖的强弱阶梯。
>
> 链条两侧还挂着两个。还要单独记住条件覆盖 CC：它和条件/判定覆盖是互补关系，单独用条件覆盖并不比判定覆盖强，所以它摆在这一层，而不在链条里。右边有两个旁支：一个是基本路径覆盖 BPC，箭头指向判定覆盖，意思是它至少能把每个判定的两个分支都走到；另一个是 MC/DC，箭头指向条件/判定覆盖，意思是它比 CC/DC 更严，但严格程度又低于把组合全部走完的 MCC。换句话说，MC/DC 是从 MCC 里精挑细选出来的那一小部分，既省了用例，又保住了独立性这条精髓。
>
> 下面四行英文别跳过，考试里经常用缩写考你：Condition Coverage 是 CC，Decision Coverage 是 DC，Multiple Condition Coverage 是 MCC，Modified Condition/Decision Coverage 是 MC/DC。
>
> 最后给一句总结：逻辑覆盖这一族方法的目标是同一个——用一组用例，把语句、判定、条件、组合、路径这几个层面的漏网之鱼同时逼出来；区别只在于你愿意为更高的覆盖强度付多少用例成本。够用就好，用对地方。
>
> 记这张图有个窍门：先记住中间那条"语句—判定—条件/判定—条件组合"的主干阶梯，再把条件覆盖和 MC/DC 挂在左手边，把基本路径覆盖挂在右手边，就不会记串了。

> **板书：**强弱阶梯（自下而上）：MCC ⊃ CC/DC ⊃ DC ⊃ 语句覆盖 ｜ 旁支：条件覆盖 CC 与 CC/DC 互补 ｜ 基本路径覆盖 BPC → 判定覆盖 ｜ MC/DC → 条件/判定覆盖（严于 CC/DC，弱于 MCC） ｜ 英文对照：CC／DC／MCC／MC-DC

> **提问：**如果项目时间只够你选一种覆盖准则来做单元测试，你选哪个？请给出理由，而不是只给名字。

> **预设回答：**

- 答"当然选最强的 MCC"：❌点评：强度最高的往往最贵，2 的 n 次方会让用例数量失控，时间不够反而什么都没测透。

- 答"选判定覆盖 DC，性价比最高"：✅实务中常见答案。它保证每个分支都走到，成本线性，是很多团队的最低门槛。

- 答"按模块风险分级：核心模块上 MC/DC 或基本路径覆盖，普通模块用判定覆盖"：✅✅最标准的答案。这也呼应本章开场那句话——方法没有优劣，只有适用。

易错点 / 考点：本页是覆盖面最广的考点页。必背两组关系：① 强弱顺序 MCC 最强，MC/DC 介于 CC/DC 与 MCC 之间，语句覆盖最弱；② 缩写与英文全称一一对应，别把 MCC 和 MC/DC 的字母顺序记混。提醒一句：箭头方向表示包含关系，不是相等——满足 MCC 一定满足 DC，反过来不成立。

【思政落点 15】从语句覆盖到判定覆盖、条件覆盖，再到 MC/DC 和基本路径覆盖，这一族方法是典型的由简到繁、层层加码。它给我们两点启示：一是学习和做事都要循序渐进，先把最简单的语句覆盖做扎实，再谈更高的覆盖强度；二是要抓主要矛盾——不是所有模块都值得上最贵的准则，把最强的功夫用在最关键的代码上，这既是工程判断力，也是资源有限条件下的责任意识。

过渡：逻辑覆盖这一节到此收口。下一页我们换一个完全不同的视角——不再问"代码哪一行没跑到"，而是问"代码里常常犯哪些错"，进入 3.5 基于缺陷模式的测试。

【3.66】拓展（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

到这里，3.4 节基于逻辑覆盖的方法就讲完了。这一页"拓展"不是考试内容，但很有价值，它告诉我们覆盖还有两个可以往外延伸的方向：一个叫数据流覆盖，一个叫控制流覆盖。

先说数据流覆盖。前面我们画的都是控制流图，关注的是"这条语句有没有被执行到"。数据流覆盖换了个视角，它盯的是数据本身：这个变量在哪里被赋值——也就是定义，又在哪里被读出来使用——也就是引用，从定义到引用的这条链有没有被完整地走过一遍。为什么需要多这个视角？因为语句覆盖、判定覆盖只管"走到没走到"，管不了"值传错了"。比如一个变量在前面被重新赋了值，后面用的却是旧值，控制流覆盖全绿，缺陷照样漏。数据流覆盖就是专门补这个洞的，它需要数据流程图，也需要代码里变量的定义与引用关系。

再说控制流覆盖。这个词有两个层次：往下看，就是我们在代码里做的逻辑覆盖——语句、分支、条件、基本路径，这一层我们已经讲过；往上看，就是业务流程基本路径覆盖——把整个业务流程画成图，让用例把

主干流程和各个异常分支都走一遍。一个测的是代码内部怎么跳转，一个测的是业务上怎么流转，方法内核是同一个。

大家记住一句话：控制流覆盖问"路走全了吗"，数据流覆盖问"值传对了吗"。两者合起来，才既管住了结构，又管住了数据。

> **板书：**数据流覆盖 → 数据流程图 + 代码中变量的定义（赋值）与引用（读取）→ 查"值"传得对不对｜控制流覆盖 → ① 代码层：逻辑覆盖（语句／分支／条件／路径）② 业务层：业务流程基本路径覆盖 → 查"路"走全没走全｜一句话：控制流问"路走全了吗"，数据流问"值传对了吗"

> **提问：**我们前面用基本路径覆盖测一个函数，语句、分支都覆盖到了，测试全部通过。这时有人提出：这个函数里有个变量算错了。基本路径覆盖为什么可能发现不了它？数据流覆盖又能从哪个角度补上？

> **预设回答：**

- 答"路径走到就行，不管值"：✅ 方向对。点评：控制流覆盖只保证结构被执行，不保证数据的定义与使用关系正确；补一句——所以覆盖率高不等于缺陷少。

- 答"基本路径覆盖能发现，只要断言写好"：⚠ 部分对。点评：断言写得好确实能发现，但这依赖测试人员想到去断言那个变量；数据流覆盖是从方法上系统地找出"定义—引用"对，不靠个人灵光。

- 答"数据流覆盖就是看数据对不对"：❌ 太笼统。点评：它不判断数据"对不对"，那是预期结果的活儿；它判断的是"定义到引用的传递链有没有被覆盖"。

> **易错点 / 考点：**第一，别把数据流覆盖与控制流覆盖混为一谈——前者看变量的定义与引用，是数据的流转；后者看语句与分支的执行顺序，是控制的流转。第二，控制流覆盖有两个层次，代码层的逻辑覆盖与业务层的基本路径覆盖，答题时能分层次说明是加分项。判断题常见："数据流覆盖属于黑盒方法"——错，它需要在代码或数据流程图层面分析。一句话口诀：控制流管路径，数据流管数值。

> **过渡：**这两条延伸线大家了解即可。接下来我们换一个完全不同的思路——不从结构出发，而是从"以前犯过的错"出发来找缺陷，这就是 3.5 节基于缺陷模式的测试。

【3.67】3.5 基于缺陷模式的测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页开始 3.5 节，基于缺陷模式的测试。这一节的前半段是本次课的重点，请大家跟紧。

先说这个思路是怎么来的。如果过去测试人员犯了不少错误，也就是产生了不少缺陷，团队自然会想：能不能从中吸取教训？把过去发现的各种具体缺陷归纳整理、抽象出共性，形成"缺陷模式"，然后拿这个模式去预防问题，也拿它去检查当前被测对象，看有没有相互匹配的问题。这就是基于缺陷模式的测试，英文缩写 DPBT，全称 Defect-Pattern-Based Testing。

它的关键动作只有一步：匹配。我知道有哪些典型缺陷、它们长什么样，就去被测对象里找有没有长得像的。像医生看病：先有病历库，再看眼前的症状对不对得上。

那它和我们第 3 次课讲的错误猜测法是什么关系？错误猜测法靠的是测试人员自己脑子里的经验，想到哪儿测到哪儿，是手工的、个人的。缺陷模式把这件事往前推了一大步：把常见缺陷模式固化到工具里，工具就能对代码做静态分析、做扫描，这时候"经验"就从个人脑子里搬进了规则库。教材提到的 FindBugs、Flawfinder、Klocwork Insight、Fortify Static Code Analyzer 这些静态测试工具，都是基于缺陷模式实现的。

好处很直接：测试效率高、效果好、容易自动化，而且只要建立规则，不需要开发测试脚本。

但要注意它的两个代价。第一是误报：静态分析天然会报出很多"看着像、其实不是"的问题，都要人工确认；误报一多，确认量可能大到承受不了，实践中常用动态测试来补。第二是漏报：因为缺陷模式本身没有规范、统一、形式化的定义，很多只是自然语言的描述，不同工具对同一个程序跑出的结果集差别很大。这两条是高频考点。

> **板书：** DPBT（Defect-Pattern-Based Testing）＝归纳历史缺陷 → 抽象共性 → 缺陷模式 → 匹配被测对象 → 发现同类问题｜一句话：过去摔过的坑，变成今天的检查单｜与错误猜测法的关系：个人经验（手工）→ 模式固化进工具（静态分析・代码扫描）｜代表工具：FindBugs・Flawfinder・Klocwork Insight・Fortify Static Code Analyzer｜优点：效率高・效果好・易自动化・只需建规则不需写脚本｜两个代价：误报（需人工确认）・漏报（模式定义不统一，工具结果集差异大）

> **提问：** 既然基于缺陷模式的工具一跑就能扫出成百上千条"疑似缺陷"，是不是意味着以后测试只要跑工具就行了？

> **预设回答：**

- 答"是，工具扫一遍就够了"：❌ 典型误区。点评：工具报的是"疑似"，不是"确诊"；误报要人工确认，漏报更危险——没报不等于没有。

- 答"不行，误报太多，要人工确认"：✅ 抓到一半。追问：把误报都确认完，剩下的就一定是真缺陷吗？——不一定，还有漏报，模式的覆盖范围决定工具能力的上限。

- 答"工具管静态的常见缺陷，业务逻辑和高风险场景还得靠人"：✅✅ 最完整的回答。点评：工具擅长重复、可模式化的问题，人擅长判断业务合理性与新风险，二者是分工而不是替代。

> **【思政落点 16】**缺陷预防优于事后返工。怎么联系知识点：DPBT 的做法是把历史缺陷抽象成模式，再拿模式去事前筛查，而不是等故障发生再救火。落到什么价值观：中医讲"上医治未病"，软件工程讲"缺陷发现得越早、修复成本越低"——把责任心往前放一步，主动给未来设防，比事后通宵补漏更有价值。

> **易错点 / 考点：**第一，DPBT 与错误猜测法同属"基于缺陷"的思路，区别在载体——一个是人的经验，一个是固化的模式与工具，考试常出对比题。第二，误报和漏报要分清：误报是"报了但其实没问题"，漏报是"有问题但没报"，两者的成因与代价不一样。第三，"工具报出来的就是缺陷"是判断题的高频错项。判断三步法：问题可模式化吗？工具覆盖了这个模式吗？报出来的人工确认了吗？

> **过渡：**知道了 DPBT 是怎么回事，下一个问题自然是：常见的缺陷模式到底有哪些？这正是下面 3.5.1 要展开的内容，也是我们这次课要结合案例讨论的地方。

【3.68】3.5.1 常见的缺陷模式（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页列出了七个常见的缺陷模式族：故障模型、安全漏洞模型、性能模型、并发故障模型、不良习惯模型、代码国际化模型、易诱骗代码模型。请大家先把这个清单记住，再逐条理解。

第一类，故障模型，也是教材里讲得最细的一类，指的是会引起缺陷的常见用法类问题，典型的就是空指针、数组越界、非法计算、使用未初始化的变量、不完备的构造函数。这些都是老熟人，但每年都还在发生。

第二类，安全漏洞模型，这类缺陷为他人攻击软件提供了可能，一旦被攻陷可能瘫痪，造成的危害更大，所以更应当尽量避免，常见的比如缓冲区溢出、被感染数据、竞争条件等。

第三类，性能模型，代码能跑对，但跑得低效：不必要的基本类型包装类、字符串的比较与复制、未声明为 static 的内部类、参数为常数的数学方法、声明了却未使用的属性与方法。

第四类，并发缺陷模型，针对多线程与虚拟机工作机制引起的问题，线程启动的任意性和不确定性让人无法

确定执行结果，对公共区域的错误使用会导致死锁。

第五类，不良习惯模型，源自程序员的不良编码习惯：文件的空输入、垃圾回收、类与方法及域的命名、方法调用、对象序列化、域初始化、参数传递、代码安全性等。

第六类，代码国际化模型，也就是硬编码，在语言国际化的过程中容易造成本地设置与程序需求不符、匹配错误。

第七类，易诱骗代码模型，指那些容易引起歧义、迷惑人的写法：无意义的比较、永远为真的判断、条件分支里写相同的代码、声明了却未使用的域——混淆视听，让人无法判断程序的真正意图。

怎么用？就一句话：认模式、找症状。下面我们就用这次课的案例讨论来练一遍。

> **板书：**七个缺陷模式族 → ① 故障模型（空指针・数组越界・非法计算・未初始化变量・不完备构造函数）② 安全漏洞模型（缓冲区溢出・被感染数据・竞争条件）③ 性能模型 ④ 并发缺陷模型（死锁・公共区域误用）⑤ 不良习惯模型（空输入・命名・序列化・参数传递）⑥ 代码国际化模型（硬编码）⑦ 易诱骗代码模型（无意义比较・永真判断・同码分支・未使用域）｜用法：认模式 → 找症状 → 设计用例

> **提问：**案例讨论。我们的课程设计里有一个"文件上传"功能，允许用户上传作业文件。请你用错误猜测法配合今天讲的缺陷模式，列出最可能出错的十个点，并说明每一条对应哪个模式。

> **预设回答：**（教师在黑板上按"错误点 → 缺陷模式"两栏写）

- 不选文件直接点上传 → 空输入（不良习惯模型）
- 上传 0 字节的空文件 → 边界处理错误（故障模型）
- 上传超过大小上限的文件 → 边界处理错误（故障模型）
- 上传 .exe、.jsp 等可执行或脚本文件未被拦截 → 危险文件类型（安全漏洞模型）
- 把 .exe 改名成 .jpg、或改 Content-Type 绕过后缀校验 → 校验可绕过（安全漏洞模型／易诱骗代码模型）
- 文件名含中文、空格、#、%、& 等特殊字符，或文件名超长 → 编码与命名处理（代码国际化模型）
- 文件名里带 ../ 试图跳出上传目录 → 路径穿越（安全漏洞模型）
- 已存在同名文件再传一次，是覆盖、重命名还是报错，行为未定义 → 结果不确定（故障模型）
- 上传中途取消或断网，服务端残留半个文件、临时文件不清理 → 资源未释放（不良习惯模型）
- 并发多次上传或上传后立即删除，文件句柄、连接、内存不释放 → 资源未释放／并发缺陷模型

- 上传失败后重试，是否重复入库、产生重复记录 → 幂等性（故障模型）

- 分片上传时分片乱序、缺片、重复片 → 边界与并发（故障模型／并发缺陷模型）

- 教师点评：多列几条更好，关键是每条都能对上模式。能对上模式的猜，才是可复用、能进检查单的经验，而不是碰运气。

> **【思政落点 17】**讲什么：把个人踩过的坑写成缺陷模式、写进检查单，而不是留在自己脑子里。怎么联系知识点：缺陷模式的价值在于"共性"——一个人的教训只有被抽象、被写下、被共享，才能变成全团队的排查能力。落到什么价值观：团队协作与知识沉淀。不藏私、肯复盘，把个人经验转化为组织资产，这既是对同事负责，也是对用户负责。

> **易错点 / 考点：**第一，七个模式族的名称要能对上号，考试常出"下列哪个属于安全漏洞模式"这类归属题。第二，最容易混的是"故障模型"与"不良习惯模型"：前者是会引起缺陷的具体用法错误（空指针、越界），后者是长期编码习惯带来的隐患（命名、序列化、参数传递）。第三，注意性能模型里的问题不是功能错，而是效率低——能跑对不代表没问题。第四，案例题常见考法：给出功能，要求列出出错点并归类，评分点是"能不能对上模式"，不是点子多。记忆口诀：故障安全性能、并发习惯国际、诱骗收尾。

> **过渡：**这些模式目前是人来认、人来查。那么能不能让机器来查？下面 3.5.2 就看 DPBT 在工具里是怎么自动实现的。

【3.69】3.5.2 DPBT 的自动化实现（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页讲 DPBT 怎么落到工具里。教材把过程拆成七步，从源代码输入开始，最后自动生成 IP 报表。IP 是 Illegal Pattern 的缩写，非法模式，意思是"这里出现了不该出现的模式"。
>
> 第一步，预处理，也叫预编译。源程序里有宏定义、文件包含、条件编译这些预处理命令，必须先把宏展开，这样才有利于变量的查找。
>
> 第二步，词法分析。把预编译阶段产生的中间代码分解开，形成各种符号表，为语法分析做准备。符号表主要有六张：标识符表、类型表、关键字表、常数表、运算符表、分界符表。
>
> 第三步，语法分析和语义处理。把输入字符串识别为单词符号流，按标准语法规则进一步分析，区分出变量定义、赋值语句、函数等等，结果是生成语法树并提供对外接口；同时从语法树可以生成程序控制流图和变

量的定义使用链，为下一步的故障查找做准备。

第四步，抽象语法树生成。源程序里所有语句都成为语法树的节点，树里带着后续处理所需的各种信息，比如语句类型、变量名称及类型，它是后面所有操作的基础。

第五步，控制流图生成。在抽象语法树的基础上生成控制流图，描绘程序结构，而且它必须反映源程序的结构。

第六步，IP 扫描，这是整个测试过程的关键步骤。先定义测试模型，然后在控制流图上遍历，用测试模型去匹配，从而生成 IP 报表。这里有一句话要划重点：工具所能检测的故障集合，取决于定义的测试模型集合——你没想到的模式，工具永远查不出来。

第七步，人工确认。因为误报的存在，生成的 IP 需要由人来确认。

我们把这七步串一遍：预编译 → 词法分析 → 语法分析与语义处理 → 抽象语法树 → 控制流图 → IP 扫描 → 人工确认。它的本质是，把"人眼看代码找毛病"这件事，翻译成"机器在图上匹配模式"。

板书： DPBT 自动化七步 → ① 预处理／预编译（宏展开）② 词法分析（六张符号表：标识符・类型・关键字・常数・运算符・分界符）③ 语法分析与语义处理（语法树＋控制流图＋变量定义使用链）④ 抽象语法树生成（语句＝节点，含语句类型／变量名称及类型）⑤ 控制流图生成（必须反映源程序结构）⑥ IP 扫描（定义测试模型 → 在控制流图上遍历匹配 → 生成 IP 报表）⑦ 人工确认（消误报）｜关键句：工具能查出什么，取决于定义了哪些测试模型

提问： 这七步里，哪一步决定了这个工具"能不能发现某个缺陷"？如果你要让它学会查一个新的缺陷模式，你要改的是哪一步？

预设回答：

- 答"第六步 IP 扫描"：✅ 方向对。点评：扫描只是执行匹配，真正决定能力的是扫描之前定义的测试模型——模型里有，才可能匹配到。

- 答"第一步预处理"：❌ 偏了。点评：预处理是前提，宏没展开会导致变量找不到，但它决定的是"查得准不准"，不是"查不查得到某个模式"。

- 答"第七步人工确认"：❌ 那是把关环节，不是能力来源。追问：那为什么还要它？——因为误报客观存在，机器给的是线索，最终判定要人负责。

易错点 / 考点： 第一，七步顺序要能默写，考试常出排序题或填空题。第二，"IP 报表"里的 IP 指 Illegal

Pattern（非法模式），不是网络里的 IP 地址，这是高频混淆点。第三，最容易漏掉第七步人工确认——自动化不等于全自动，缺了它就不是完整的 DPBT 流程。判断三步法：预编译了吗？模型定义了吗？报出来确认了吗？

过渡：这七步只是流程，真正把这些步骤串起来、还能继续往下做求解的，是一套通用的实现框架。我们看下一页的图。

【3.70】实现的通用框架（1 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是一张架构图，标题叫"实现的通用框架"。我们按顺序过。

最左边黄色爆炸图形是"被测源程序"，经"解析"变成"抽象语法树"。从这里分析分成几条路走：抽象解释、符号执行、图可达，最后汇到"数据流等基本分析方法"；它右边挂着一排具体技术：函数调用、Mod-effect、SSA（静态单赋值）、多态分析、指向分析、全局变量分析、前后支配。

分析结果往下形成"程序状态空间"，再往下是"基于缺陷模式的逻辑表达式"。注意它不是凭空来的：左边的"缺陷模式"用虚线指向它，而缺陷模式的来源是三个业界公认的知识库——OWASP、CWE、CERT。CWE 是通用缺陷枚举，一个安全漏洞词典；OWASP 是开放 Web 应用安全项目；CERT 是安全编码标准。

逻辑表达式再往右交给 SMT 求解——SMT 是可满足性模理论。教材注明，这类求解器已被集成到 HOL/Isabelle、ESC/Java2、ACL2 等大型工具中。求解出来的结果，就是最后的"检测结果"。

记住三个抓手：抽象语法树是起点，缺陷模式是知识来源，SMT 求解是判定手段。

板书：被测源程序 →解析→ 抽象语法树 →（抽象解释／符号执行／图可达）→ 数据流等基本分析方法 →（函数调用・Mod-effect・SSA・多态分析・指向分析・全局变量分析・前后支配）→ 程序状态空间 → 基于缺陷模式的逻辑表达式 ←缺陷模式← OWASP／CWE／CERT → SMT 逻辑表达式求解 → 检测结果｜三个抓手：语法树是起点・缺陷模式是知识来源・SMT 是判定手段

提问：图上有一条虚线，是从"缺陷模式"指向"基于缺陷模式的逻辑表达式"的。如果改成从缺陷库直接连到"检测结果"，会丢掉什么？

预设回答：

- 答"会丢掉中间的分析"：✅ 对。点评：缺陷模式是"要查什么"，程序状态空间是"程序可能是什么样"，逻辑表达式把两者接起来，才能交给求解器判定。

- 答"没什么区别，反正结果都是检测结果"：❌ 误区。点评：没有中间这层，"模式"就退化成字符串匹配，语义层面的缺陷根本查不出来。

- 答"会丢掉对程序的分析结果，变成只按关键字匹配"：✅✅ 最准确。点评：这正是静态分析工具从"全文搜索"升级为"语义分析"的关键一跳。

> **易错点 / 考点：**第一，SSA 全称 Static Single-Assignment（静态单赋值），SMT 全称 satisfiability modulo theories（可满足性模理论），CWE 全称 Common Weakness Enumeration（通用缺陷枚举），考试常考英文缩写与中文名的对应。第二，别把 OWASP、CWE、CERT 当成工具名，它们是缺陷模式的知识来源与标准。第三，这张图的顺序不能颠倒：解析在前、求解在后。

> **过渡：**缺陷模式这条路，本质还是"从错误出发"。接下来我们换一种更主动的思路——先把系统该有的行为建成模型，再从模型里生出用例来。这就是 3.6 节基于模型的测试。

【3.71】3.6 基于模型的测试（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页开始 3.6 节，基于模型的测试，英文 Model-Based Testing，缩写 MBT。这一节是第 4 次课的重点之一，我们分两小节讲：3.6.1 功能图法，3.6.2 模糊测试方法。本片先把 MBT 的总体思路和功能图法讲清楚，模糊测试方法下一片继续。
>
> 先给一个概括的印象。前面几节，我们要么从输入数据出发——等价类划分、边界值分析；要么从代码结构出发——逻辑覆盖、基本路径覆盖；要么从过去的错误出发——基于缺陷模式。而 MBT 走的是第四条路：先把被测系统的行为抽象成一个模型，再让模型来告诉我们该测哪些情况。用例不是一条条"想"出来的，是从模型里"长"出来的。
>
> 为什么要有这条路？因为当系统行为复杂、组合很多的时候，靠人脑枚举路径一定会漏。模型一旦建准了，覆盖的完整性和一致性就有了保障；需求变了，改模型、重新生成用例，比手工去改一堆用例可靠得多。
>
> 但也要先把丑话说在前面：MBT 不是要替代前面那些方法。教材明确说，基于模型的软件测试技术不能替代已有的其他测试技术，而是对其他测试技术的有力补充，目前已经应用于通信协议测试、API 测试等领域。微

> 软研究院用 C# 开发过相应的工具 Spec Explorer，教材上有它的界面截图，也给了官网和团队博客地址，感兴趣的同学可以课后去看。
>
> 接下来几页，我们先把 MBT 的定义和它的实施过程讲透，再看它由哪些角色和构件支撑，最后回到最常用的功能图法。

> **板书：**3.6 基于模型的测试 MBT（Model-Based Testing）｜小节目录：3.6.1 功能图法｜3.6.2 模糊测试方法｜定位：不能替代其他测试技术，是有力补充（已用于通信协议测试・API 测试）｜代表工具：微软 Spec Explorer（C#）

> **提问：**前面几节的思路分别是"从数据出发""从代码出发""从错误出发"。MBT 是"从模型出发"。你觉得"从模型出发"最大的好处和最大的风险分别是什么？

> **预设回答：**

- 答"好处是不漏、可自动生成"：✅ 对。追问：那风险呢？——模型错了，生成的所有用例都跟着错。

- 答"风险是建模成本高"：✅ 抓到重点。点评：建一个准确的模型要投入人力和时间，模型维护更是长期成本，所以 MBT 更值钱的地方是行为复杂、变更频繁的系统。

- 答"MBT 可以完全替代手工测试"：❌ 典型误区。点评：教材说得很清楚——它是补充，不是替代；模型只描述了你写进去的行为，模型之外的东西还得靠其他方法。

> **易错点 / 考点：**第一，MBT 三个字母对应 Model-Based Testing，别写成"基于模块的测试"。第二，本节两小节的归属要分清：功能图法属于 3.6.1，模糊测试方法属于 3.6.2，考试常考归属题。第三，"MBT 能替代其他测试方法"是高频错项，教材口径是"有力补充"。

> **过渡：**那么，MBT 到底怎么定义？它凭什么能从模型生出用例？我们看下一页。

【3.72】什么是 MBT？（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页给出 MBT 的正式定义，请大家跟着我读一遍：基于模型的测试（MBT），是通过构建能够正确描述被测软件系统功能特性的模型，然后基于这个模型产生测试用例并执行这些测试用例的过程。
>
> 这个定义里有三个动词要抓住：构建模型、产生用例、执行用例。少了任何一个都不完整——只建模不生成

用例，那叫建模，不叫测试；生成用例不去执行，那叫设计，也不叫测试。

请看右边的图，它把定义画成了一个闭环。起点是"文字版本的需求或规范"，经过"抽象转换"变成"模型"。模型再往下"生成"出两样东西：一样是"输入——测试操作序列"，一样是"输出——判断准则"，也就是Oracle，测试预言。左边这一路"控制"被测系统 SUT 执行测试；右边这一路去"观察、比较"实际结果。判断准则和需求都要送进"验证"这个菱形框里对照，验证的结果再"反馈"回模型和需求。所以 MBT 是一个能自我修正的循环：模型 → 用例 → 执行 → 比较 → 反馈 → 改模型。

这里有两个词要专门解释。第一个是"抽象转换"：需求是自然语言写的，里面有歧义；建模就是把自然语言翻译成没有歧义的、形式化的描述。第二个是"判断准则 Oracle"：测试用例不只要有输入，还要有"什么才算通过"的判定标准，而这个标准是从模型推出来的，不是测试人员临时拍脑袋定的。

教材还补充了一句很重要的话：MBT 先从概念上形成模型，然后用数学的方法来描述这个模型，逐渐形成仿真模型，完成所需的测试。也就是说，模型有三个层次——概念上的、数学描述上的、可仿真的，越往后越能自动化。

另外提醒一句：模糊测试方法和变异测试，我们在讲安全性测试时已经讨论过了，这里是它们作为 MBT 的常见方法再次出现。

板书： MBT 定义三动词 → ① 构建能正确描述被测系统功能特性的模型 ② 基于模型产生测试用例 ③ 执行这些测试用例｜闭环：需求／规范 →抽象转换→ 模型 →生成→ 输入（测试操作序列）＋ 输出（判断准则 Oracle）→ 控制／观察比较 → 针对 SUT 执行测试 → 验证 → 反馈 → 修正模型与需求｜模型三层次：概念模型 → 数学描述 → 仿真模型

提问： 定义里说"构建能够正确描述被测软件系统功能特性的模型"。如果这个模型本身就是错的——它描述的行为跟系统实际该有的行为不一样，会发生什么？这种错误能在 MBT 流程里被发现吗？

预设回答：

- 答"会生成一堆错的用例"：✅ 对，而且危险在于用例是自动生成的，看起来又多又整齐，人容易以为覆盖得很充分。

- 答"能发现，因为验证环节会反馈"：⚠ 只对一半。点评：图上的"验证"比的是判断准则和需求，如果需求本身就没写清、或者模型和需求犯了同样的理解错误，这个环就发现不了——它只能发现模型与需求不一致，发现不了"需求和模型一起错"。

- 答"发现不了，得靠评审需求和模型"：✅✅ 最到位。点评：所以 MBT 的前提是模型要经过评审，模型的可

信度决定了 MBT 的上限。

> **易错点 / 考点：**第一，MBT 的三要素是模型、用例、执行，答题时不能只答"用模型生成用例"。第二，Oracle（判断准则／测试预言）是从模型推导出来的，不是人工临时定的。第三，最容易混的是"模型不等于代码"：模型是系统预期行为的抽象描述，通常比代码抽象、比需求形式化。第四，判断题常见"MBT 生成的用例不需要人工审核"——错。

> **过渡：**定义清楚了，接下来看它落到工程里怎么干。下一页是 MBT 的实施过程，也是本节的重点页。

【3.73】MBT 基本原理（实施过程）（3 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页是本节的重点，讲 MBT 的实施过程，一共五步，请务必记牢。
>
> 第一步，为被测试系统 SUT 建模。对应图上的"1. Model"：需求文档进来，由建模者操作，输出是模型 Model；旁边虚线连着测试计划，说明建模与测试计划相互参照。
>
> 第二步，基于模型产生测试用例。对应"2. Generate"：用例生成器按覆盖准则从模型里导出测试用例，同时产出两份很重要的材料——需求追溯矩阵和模型覆盖率。这是 MBT 相比手工设计用例最大的增量价值：每条用例对应哪条需求、模型覆盖到什么程度，都是自动记录、可查的。
>
> 第三步，将抽象的测试具体化，使测试用例具有可执行性。对应"3. Concretize"：抽象用例只说了"该发生什么"，要落到具体数据和操作步骤才能跑，所以由脚本生成器把用例翻译成测试脚本，再通过适配器 Adaptor 接到实际环境上。适配器这一层很关键，它是模型世界和真实系统之间的桥。
>
> 第四步，执行测试。对应"4. Execute"：由测试执行工具驱动被测系统运行脚本。
>
> 第五步，分析测试结果。对应"5. Analyze"：把结果拿回来分析，判定通过还是失败，并反馈回去修正模型和需求。
>
> 五步串成一句口诀：建模、生成、具体化、执行、分析。特别注意第 2 步和第 3 步是两回事——生成出来的是抽象用例，具体化之后才是可执行脚本，把这两步并成一步是典型失分点。
>
> 最后看收益与成本。收益是需求与用例同步：需求变了改模型、用例重新生成，不必人工一条条去改；追溯矩阵与模型覆盖率自动产出，变更可重生成、可追溯。成本是建模要投入、模型要长期维护、具体化还要写

适配器。所以 MBT 不是"免费的自动化"，而是用前期的建模成本换后期的可重复与可追溯。

> **板书：** MBT 实施五步 → ① 为 SUT 建模（需求 → 模型；与测试计划相互参照）② 基于模型产生用例（用例生成器 → 测试用例 + 需求追溯矩阵 + 模型覆盖率）③ 具体化（脚本生成器 + 适配器 → 可执行脚本）④ 执行测试（执行工具 → 被测系统）⑤ 分析测试结果（判定 → 反馈修正）｜口诀：建模・生成・具体化・执行・分析｜收益：需求与用例同步・变更可重生成・可追溯｜成本：建模投入・模型维护・适配器开发

> **提问：** MBT 说自己有个大好处——"需求变了不用改用例，改模型重新生成就行"。这个好处在什么样的情况下会不成立？

预设回答：

- 答"需求变得太大、模型结构都变了"：✅ 对。点评：小改动能靠重新生成省力；如果变更导致模型要重画，那建模成本就白省了。

- 答"如果需求是自然语言、又没人维护模型，就做不到"：✅✅ 说到根子上。点评：模型是唯一事实来源这件事必须成立，需求文档和模型脱节，MBT 就退化成一次性工作。

- 答"用例不用改，但脚本要重新生成"：✅ 准确的补充。点评：抽象用例可以重生成，具体化那一步——脚本和适配器——往往还得跟着调。

> **【思政落点 18】** 讲什么：MBT 的需求追溯矩阵与模型覆盖率是自动产出的。怎么联系知识点：它要求需求、模型、用例三者一一对应、可查证，模型改了用例能重新生成，整个链条留痕。落到什么价值观：这就是工程规范化与可追溯意识——做事讲流程、讲证据、讲责任到人，而不是靠拍脑袋、靠记忆。

> **易错点 / 考点：** 第一，五步的顺序与名称是必背内容，注意"具体化"是独立的一步。第二，要能说出第 2 步的两个附加产出：需求追溯矩阵与模型覆盖率。第三，第 3 步里的适配器 Adaptor 不可省，它是模型与真实系统之间的接口，也是 MBT 落地成本的主要来源之一。第四，案例题常问"MBT 的收益与成本"，答收益必须同时答成本才算完整。

> **过渡：** 五步讲完，我们再往下一层看：这套流程在工程里由哪些角色承担、靠哪些构件支撑？下一页是 MBT 的架构。

【3.74】MBT 架构（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是 MBT 的总体架构图，我们分四块看。

先看参与这套架构的角色。Test Manager 测试经理负责统筹；Test Modeler 建模者负责把需求变成模型；Test Model Execution Specialist 测试执行专家负责让测试真正跑起来。另外还有 Model Designer，也就是模型设计者。这说明 MBT 是有分工的工程活动，不是一个人写完模型就完了。

再看中间那个大圆柱，叫 Model Repository 模型仓库，是整个架构的核心。里面放着测试数据、测试配置、测试日志，还有精化机和动作机；建模者往里创建模型，测试生成再从里面取模型。

第三块是右边的测试生成：模型组合器把模型拼成测试模型，测试控制与测试引擎执行模型并记录状态。

第四块是执行侧：关键字执行经由 DevHost、DevControl 到适配器，驱动 SUT 1 到 SUT N——它可以同时接多台被测设备；上面还有测试控制通过 WWW GUI 做执行、控制与观察。

记住一句话：MBT 的核心资产是模型仓库，围绕它分成建模、生成、执行三条流水线。

板书： MBT 架构四块 → ① 角色：测试经理・建模者・模型设计者・测试执行专家 ② 核心：Model Repository 模型仓库（测试数据・测试配置・测试日志・精化机・动作机）③ 测试生成：模型组合器 → 测试模型 → 测试控制与测试引擎 ④ 执行：关键字执行 → DevHost／DevControl → Adapter → SUT 1..N（WWW GUI 执行、控制与观察）｜一句话：核心资产是模型仓库，围绕它分建模・生成・执行三条流水线

提问： 这张图上，"模型仓库"被放在正中间，两边的人都要跟它打交道。如果团队里没有这个统一仓库，每个建模者各存一份模型，会出什么问题?

预设回答：

- 答"会版本混乱、对不上"：✅ 对。点评：模型是生成用例的唯一来源，来源不统一，用例就不一致，追溯矩阵也失去意义。

- 答"会重复劳动"：✅ 补充得对。点评：测试数据、测试配置、日志都放在仓库里，正是为了让建模、生成、执行三条线共用同一份事实。

- 答"没什么影响，各自跑各自的"：❌ 误区。点评：这正是"能跑起来但不可维护"的典型，模型一改，没人知道该改哪一份。

易错点 / 考点： 第一，四个角色的英文名与职责要能对上，考试常出匹配题。第二，模型仓库里装的东西容易记混：测试数据、测试配置、测试日志属于仓库；模型组合器、测试引擎属于测试生成侧，不要混。第三，

注意架构支持多台 SUT（SUT 1 到 SUT N），这是 MBT 在设备兼容性测试场景中的优势。第四，一句话总结：模型仓库是 MBT 的心脏。

过渡：架构说清了，最后落到方法层面：常见的 MBT 方法到底有哪些？请看下一页。

【3.75】常见的 MBT 方法（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页列出常见的 MBT 方法，一共六条。

第一条，有限状态机，finite state machines，缩写 FSM，还包括扩展的有限状态机 EFSM。这是最经典、最常用的 MBT 模型，思路是把系统的行为描述成一组状态以及状态之间的迁移。我们下一页要讲的状态迁移图，就是它的图形化表达。

第二条，符号执行，Symbolic Execution。一种使用符号值代替数字值执行程序的程序分析技术。普通执行是我们给具体输入、看具体输出；符号执行是把输入当成符号变量，沿着路径把约束积累下来，最后求解这些约束，得到"什么样的输入会走这条路"。它的好处是一条符号路径可能代表成百上千个具体输入。

第三条，定理证明，Theorem proving。做法是通过一组能够明确定义系统行为的逻辑表达式，也就是谓词，来完成模型构建，然后去证明性质成立。

第四条，模型检验，Model checking。它是对属性的测试：如果属性在模型中是有效的，模型检验能发现证据；如果无效，它能给出反例。这一条在后面的形式化方法一节还会深化。

第五条，随机或半随机模型，比如模糊测试方法、变异测试、马尔科夫链。这一类不追求把行为完全刻画清楚，而是靠随机、半随机的生成去撞出问题，适合发现意料之外的缺陷。

第六条，其它方法，包括基于 UML 的 MBT、因果图方法等。

教材的讲者备注还补充了几个相关概念：约束逻辑编程可以和符号执行配合，用于选择测试用例；随机或半随机模型的典型代表还有模糊控制器、马尔科夫链；模糊测试还可以细分为基于变异的生成器和基于生成的生成器。

记忆抓手：状态机、符号执行、定理证明、模型检验、随机模型，最后加一个"其它"。

板书：常见 MBT 方法六条 → ① 有限状态机 FSM／扩展有限状态机 EFSM（最常用）② 符号执行：使用符

> 号值代替数字值执行程序 ③ 定理证明：通过一组逻辑表达式（谓词）定义系统行为 ④ 模型检验：对属性测试，有效给证据、无效给反例 ⑤ 随机／半随机模型：模糊测试・变异测试・马尔科夫链 ⑥ 其它：基于 UML 的 MBT・因果图方法｜配套概念：约束逻辑编程（选用例）・模糊控制器・马尔科夫链

> **提问：**符号执行的定义是"使用符号值代替数字值执行程序"。请你用一句话说明：它跟我们平时跑一遍程序比，多得到了什么？

> **预设回答：**

- 答"得到了路径条件"：✅ 对。点评：一次具体执行只能验证一个输入；符号执行把输入变成变量，积累出这条路径的约束条件，一个约束就能代表一批输入。

- 答"得到了更快的速度"：❌ 偏了。点评：符号执行通常比直接执行更慢、更复杂，它换来的不是速度，是覆盖面与可推导性。

- 答"能得到什么样的输入会走哪条路，从而反推用例"：✅✅ 最完整。点评：这正是符号执行被列入 MBT 方法的原因——它能直接支撑用例的生成。

> **易错点 / 考点：**第一，六条方法的名称要能默写，尤其 FSM 与 EFSM、模型检验与定理证明容易混。第二，定理证明是"用逻辑表达式（谓词）定义系统行为"，模型检验是"对属性做检验、给证据或反例"，两者的区别是高频考点。第三，别把模糊测试、变异测试漏掉——它们属于"随机／半随机模型"这一类。第四，因果图在教材里被归为基于模型的方法，所以出现在这里；而决策表、两两组合、正交实验被教材归为组合方法，不要混。

> **过渡：**方法列完了，我们从最常见的那一类往下钻——有限状态机。教材用"功能图法"来组织它，下一页开始 3.6.1。

【3.76】3.6.1 功能图法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页开始 3.6.1 功能图法，这是本次课的重点页，请大家认真听。
>
> 先讲程序的功能通常由哪两部分说明组成。第一部分是静态说明，描述输入条件和输出条件之间的对应关系——给什么输入、得到什么输出。第二部分是动态说明，描述输入数据的次序或者转移的次序——先做什么、后做什么。

为什么要区分这两者？因为大量的组合情况存在，如果仅使用静态说明来组织测试往往是不够的，必须通过动态说明来补充测试。功能图法就是因此而产生的一种测试用例设计方法。

功能图法的做法是：使用功能图形式化地表示程序的功能说明，并按照图覆盖原理生成测试用例。它有两个要素——一个是"图"，一个是"覆盖准则"。图决定你画出什么，覆盖准则决定你从图里取多少用例。

教材说，图覆盖准则可以分为三种：第一，节点覆盖，覆盖图中所有可达的节点；第二，边覆盖，覆盖图中所有可达的节点连线；第三，路径覆盖，覆盖从初始节点到结束节点形成的路径。放到状态迁移的语境里，节点覆盖就是我们常说的全状态覆盖，边覆盖就是全迁移覆盖，路径覆盖就是全路径覆盖。路径覆盖还可以细分为主路径（简单路径）覆盖、完全路径覆盖、指定路径覆盖等。

但这里有一个很重要的提醒：完全路径覆盖往往会产生无限的路径，是无法满足的，所以实践中一般选主路径覆盖。教材还给了生成用例的三种策略：顺路游历策略、绕路游历策略和遍历策略。

最后是功能图的构成：它由状态迁移图和逻辑功能模型两部分组成。状态迁移图表示输入数据序列以及相应的输出数据，由输入和当前状态决定输出数据和后续状态；逻辑功能模型表示状态输入条件和输出条件之间的对应关系，只适合描述静态说明，输出数据仅由输入数据决定。

> **板书：**程序功能的两类说明 → 静态说明：输入条件与输出条件的对应关系｜动态说明：输入数据（转移）的次序｜功能图法 = 用功能图形式化表示功能说明 + 按图覆盖原理生成用例｜图覆盖三准则：① 节点覆盖（全状态）② 边覆盖（全迁移）③ 路径覆盖（全路径；含主路径／简单往返／完全往返／完全路径／指定路径）→ 完全路径往往无限，一般取主路径｜用例生成策略：顺路游历・绕路游历・遍历｜功能图 = 状态迁移图（动态说明）+ 逻辑功能模型（静态说明）

> **提问：**完全路径覆盖"往往会产生无限的路径，是无法满足的"，所以一般退而取主路径覆盖。既然做不到，为什么教材还要把它列为一个覆盖准则？它的意义在哪里？

> **预设回答：**

- 答"它是个理想上界，用来衡量实际覆盖了多少"：✅ 到位。点评：准则的价值不只在"达到"，也在"参照"——知道理论上限在哪，才知道自己离完备有多远。

- 答"因为简单程序能全覆盖"：✅ 补充得对。点评：路径有限的小系统确实可以完全覆盖，路径爆炸是复杂系统才出现的问题。

- 答"考试要考，所以列出来"：❌ 别这么想。点评：三个准则是层层递进的强度阶梯——节点覆盖最弱、边覆盖居中、路径覆盖最强，这个递进本身就是考点。

易错点 / 考点：第一，静态说明与动态说明分别对应逻辑功能模型与状态迁移图，这个对应关系最容易记反，是高频失分点。第二，图覆盖三准则的名称与强度递进（节点弱于边、边弱于路径）要记牢，注意它们就是状态迁移语境下的全状态、全迁移、全路径覆盖。第三，完全路径覆盖不可行、实用取主路径覆盖，这个结论常考。第四，别把"功能图"理解成"程序的流程图"，功能图由状态迁移图和逻辑功能模型构成。

过渡：功能图的两半里，动态的那一半更关键，也是我们下面要重点看的——状态迁移图。

【3.77】状态迁移图（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲状态迁移图。定义是这样的：状态迁移图描述系统状态变化的动态信息，也就是动态说明，由状态和迁移来描述；状态指出数据输入的位置或时间，而迁移则指明状态的改变。

请大家把这句话拆成四个建模要素来记。第一是状态，系统当前处在什么情况，比如"等待""成功""失败"；第二是事件，也叫输入，是什么触发了变化，比如"输入正确的用户名和口令"；第三是迁移，就是状态之间的那条箭头，它指明从一个状态变到另一个状态；第四是条件，有些教材和工具把箭头必须满足的条件叫守卫条件，只有条件成立，这条迁移才允许发生。

看图。这是一个登录功能的状态迁移图。起点是"等待状态"，从它出发有几条线：输入正确的用户名和口令，迁移到"成功状态"，成功状态再通到"下一个状态"；用户名和口令输错，箭头回到"等待状态"自己——这叫自迁移，是状态迁移图里非常典型的一种画法；输错超过 3 次，迁移到"失败状态"；在失败状态点击"忘记密码"，迁移到"新的等待状态"。

这张图里最值得说的是那个自迁移和一个带条件的迁移。自迁移告诉我们：输错一次并不改变系统所处的状态，但业务上它一定改变了某些东西——比如错误次数加一。带条件的迁移"输错超过 3 次"告诉我们：迁移不是随便发生的，它有触发条件，而且这个条件通常是某个计数器累积到阈值。

所以状态迁移图的价值就在这里：它把"藏在状态里的数据"和"什么时候允许变化"一起画出来了，而这些恰恰是文字需求最容易漏掉的部分。大家自己画图的时候，一定要问自己三个问题：状态全了吗？每个状态的出口事件全了吗？有没有哪个迁移是有条件的，条件写清楚了吗？

板书：状态迁移图 = 描述系统状态变化的动态信息（动态说明）｜建模四要素：① 状态（数据输入的位置／时间）② 事件（输入，触发变化）③ 迁移（状态之间的箭头，指明状态的改变）④ 条件／守卫条件（迁

移成立的前提）｜登录示例：等待状态 —正确口令→ 成功状态 → 下一个状态；等待状态 —输错→ 等待状态（自迁移）；等待状态 —输错超过 3 次→ 失败状态 —点击"忘记密码"→ 新的等待状态｜自检三问：状态全吗？出口事件全吗？条件写清了吗？

提问：图上"用户名和口令输错"这条箭头是从"等待状态"出发、又回到"等待状态"的。既然状态没变，这条迁移为什么必须画出来？如果不画，会漏掉什么？

预设回答：

- 答"会漏掉错误次数的累积"：✅ 抓住关键。点评：状态没变，但状态里的数据变了——错误次数正是后面"输错超过 3 次"那个条件的基础，不画这条线，这条业务规则就悬空了。

- 答"会漏掉对自迁移本身的测试"：✅ 补充得对。点评：自迁移要单独设计用例——连续输错一次、两次、三次，边界刚好落在阈值上。

- 答"不画也没关系，反正状态一样"：❌ 典型错误。点评：状态迁移图不只画状态，还画状态之间的变化关系；漏掉自迁移，等于漏掉了一整类用例。

易错点 / 考点：第一，状态迁移图的四要素（状态、事件、迁移、条件／守卫条件）要能默写，考试常用填空或判断。第二，自迁移与"同状态不同条件"是常考的细节，比如输错一次与输错超过 3 次都从等待状态出发，但目标状态不同。第三，注意定义里那句"状态指出数据输入的位置（或时间）"，这句话常作为单选题的选项出现。第四，画图题评分点：状态是否完备、迁移是否有事件标注、条件是否写明。

过渡：图画出来了，覆盖准则也有了，最后一个问题是：怎么从这张图真正变出测试用例？下一页专门回答这个。

【3.78】如何设计测试用例？（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页回答一个很实的问题：有了功能图，测试用例到底怎么设计出来？教材给了一句总纲：功能图法设计测试用例，就是如何覆盖软件所表现出来的所有状态，可以转化为两个层次的测试用例。

第一个层次是局部的测试用例，从功能逻辑模型——也就是决策表或者因果图——导出。它要覆盖的是各个状态下的各种输入数据的组合。请注意"各个状态下的组合"：同样是输入用户名和口令，在"等待状态"下和在"失败状态"下，预期结果是不一样的，所以要分状态设计。这一层用的是黑盒方法。

第二个层次是整体的测试用例，从状态迁移图导出，要覆盖系统（程序）控制的逻辑路径。也就是说，它关心的是从初始状态出发，经过哪些迁移、最终到达哪里。这一层用的是白盒方法——路径覆盖、分支覆盖、条件覆盖这些。我们前面讲的基本路径覆盖法在这里正好派上用场：从逻辑功能表中，根据所有的输入及状态生成所需要的节点和路径，形成实现功能图的基本路径组合。

所以教材给了一句很精辟的总结：功能图法是综合运用黑盒方法和白盒方法来设计测试用例——整体上选用白盒方法，做路径覆盖、分支和条件覆盖；局部上选用黑盒方法，用决策表或者因果图方法。

这句话建议大家当成口诀背下来："整体白盒走路径，局部黑盒配组合"。它也是案例题的高频答案。为什么必须两层结合？因为只做整体覆盖，你会走过所有路径，但每条路径上的输入数据是随便挑的；只做局部覆盖，你把每个状态下的输入组合都测遍了，但状态之间的转移顺序可能从来没串起来过。两层合起来，才既管住"怎么走"，又管住"给什么"。

还要提醒一句：功能图法的整体路径覆盖，本质上就是我们前面讲的基本路径覆盖法在状态图上的应用，前后知识是打通的。

> **板书：**功能图法设计用例 = 覆盖软件表现出的所有状态 → 分两层｜① 局部用例：从逻辑功能模型（决策表／因果图）导出 → 覆盖各状态下输入数据的组合 → 黑盒方法｜② 整体用例：从状态迁移图导出 → 覆盖系统（程序）控制的逻辑路径 → 白盒方法（路径覆盖・分支覆盖・条件覆盖）｜口诀：整体白盒走路径，局部黑盒配组合｜落地：由逻辑功能表生成节点与路径 → 用基本路径覆盖法设计用例

提问：只用一层行不行？比如我只做从状态迁移图导出的整体用例，把每条路径都走一遍，是不是就够了？

预设回答：

- 答"不够，输入组合没管"：✅ 对。点评：路径走对了，每条路径上的数据却是随手挑的，等价类和边界值上的缺陷照样漏。

- 答"只做局部也不够"：✅✅ 两边都说到了。点评：局部把每个状态的输入测细了，但转移的次序、状态之间的衔接从没被验证过——比如"输错三次之后再输对"这种时序问题就漏了。

- 答"够，路径覆盖最强"：❌ 典型误区。点评：覆盖强度说的是"结构被执行的程度"，不等于"数据被验证的程度"，这两件事不能互相替代。

易错点 / 考点：第一，"整体白盒、局部黑盒"这个分工方向最容易记反，一定要背准：整体用状态迁移图加路径覆盖（白盒），局部用决策表或因果图（黑盒）。第二，局部用例的来源是"逻辑功能模型"，它只适合描述静态说明，输出仅由输入决定——这一点和 3.76 那页的静态说明是一回事，要能串起来。第三，案例题常

考"请说明功能图法如何设计测试用例"，标准答案必须包含两个层次及其对应方法。第四，别把功能图法说成"纯黑盒方法"或"纯白盒方法"，它是综合运用。

过渡：到这一页，3.6.1 功能图法就完整了——从静态说明与动态说明，到功能图的两个组成部分，再到两个层次的用例设计。下一页开始 3.6.2 模糊测试方法。

【3.79】3.6.2 模糊测试方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

从这一页开始，我们进入 3.6 节第二块内容——模糊测试方法，英文 Fuzz Testing。定义是：模糊测试就是在一个被测试的程序中附加上随机数据，这些随机数据就叫 fuzz，把它们作为程序的输入送进去；如果被测试程序出现问题，例如崩溃（Crash），或者异常退出，就可以定位程序的缺陷。

模糊测试的判据不是"输出对不对"，而是"程序有没有扛住"。我们事先不需要写出每个输入对应的正确输出，而是先入为主地认定：程序不管收到什么输入，都不该崩溃、不该异常退出。课件在页面上专门点了这一点——模糊测试的巨大优势，就是测试设计极其简单，系统的行为先入为主。

讲到这儿大家容易把它和前面学过的随机测试混起来。两者都靠随机数据，但盯的信号不一样。随机测试是从输入域里抽数据，把实际输出和预期输出比对，它找的是"算错结果"的缺陷，必须有预期结果当参照；模糊测试不比对结果，它盯的是程序崩不崩、挂不挂，找的是健壮性、容错性上的缺陷。一句话：随机测试管"答得对不对"，模糊测试管"扛不扛得住"。

据教材数据，在键盘或鼠标大量随机输入的情况下，Windows NT 4.0 有 21% 的程序会崩溃，还有 24% 的程序会挂起。这说明相当多的程序很少认真处理畸形输入。

典型战场是两个：安全性测试和容错性测试。它可以模拟黑客发动攻击测试，也可以检验服务器的容错能力。它的思路缺乏严密逻辑：不去推导到底哪个数据把系统搞坏了，而是在基本框架内造出尽可能多的杂乱数据，撞出意想不到的缺陷。

数据海量，模糊测试不能手工做，必须用工具自动执行，过程很简单，就四步：生成大量数据；把数据发给被测系统；检测被测系统的状态；最后根据状态判断是否存在潜在的安全漏洞。

板书：3.6.2 模糊测试（Fuzz Testing）｜做法：附加随机数据 fuzz → 触发 Crash／异常退出 → 定位缺陷 ｜信号＝崩溃，不是"输出对不对" ｜优势：测试设计极简单、行为先入为主 ｜战场：安全性 + 容错性 ｜必用工具：四步（造数据 → 输入 → 查状态 → 判漏洞）

> **提问：**随机测试和模糊测试都用随机数据，请问它们判断"发现问题"的依据有什么不同？如果只让你保留一条判据，你留哪条？

> **预设回答：**

- 答"模糊测试看有没有崩溃，随机测试看输出对不对"：✅ 对。追问：那模糊测试要不要预期结果？——不需要，这正是它测试设计简单的原因。

- 答"两者是一样的，都是随机造数据"：❌ 典型误区。点评：判据不同决定了两者能发现的缺陷类型不同，随机测试发现功能错误，模糊测试发现健壮性与安全性问题；说"一样"就丢掉了一个考点。

- 答"模糊测试也能发现功能错误"：⚠ 偶尔能，但那不是它的主战场。点评：它没有预期结果作参照，功能正确性主要还得靠等价类、边界值、逻辑覆盖这些方法。

> **易错点 / 考点：**最大的坑是把模糊测试等同于"随机测试"或"随便乱测"。考点常见判断与选择：模糊测试的判据是程序崩溃、异常退出等异常行为，它不需要预期输出（缺乏直接的测试预言），因此主要面向容错性与安全性；而随机测试属于基于输入域的方法，仍要拿实际输出与预期输出比对。记忆三步法：数据从哪来（随机／半随机）→ 异常当信号（崩溃＝缺陷）→ 用在什么场景（安全、容错）。另外注意课件上写的是"Crush"，那是笔误，正确拼写是 Crash，考试写 Crash。

> **过渡：**既然模糊测试要靠工具自动造数据，那造数据的部件长什么样、有几种做法？下一页我们就来看模糊器的几种构造。

【3.80】几种不同模糊器的构造（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 上一页我们知道了模糊测试是干什么的，这一页看它的核心部件——模糊器，也就是那个自动产生数据的模板。课件把模糊器相关做法分成三类：黑盒随机模糊、基于语法的模糊、白盒模糊处理。三者的共同目标一样，都是造出畸形输入去撞程序，区别在于"数据从哪里来、按什么规则造"。
>
> 第一类，黑盒随机模糊。做法是拿一份正确格式的输入数据，对它做随机变异，然后用这些变异后的输入运行程序，看是否能够触发异常。课件讲得很实在：这是一种简单的 hack；如果一个应用从未进行过模糊测试，用这种技术就能有效地发现应用中的漏洞。它的优点是几乎零门槛，缺点是变异像瞎撞，撞进深层逻辑的概率低。

第二类，基于语法的模糊，是模糊复杂格式输入的替代方法。它要求我们先指定三件事：输入格式的输入语法、哪些输入部分要进行模糊化、以及如何模糊化。基于语法的模糊生成器会生成许多新的输入，每个输入都满足语法编码的约束——说白了，造出来的数据"格式上像模像样，内容上稀奇古怪"。那这种模糊的效率靠什么指导？靠模糊生成器的用户的创造力和专业知识。

第三类，白盒模糊处理，由微软研究院于 2008 年首创。它的思路和前面两类完全不同：动态地执行被测程序，从执行过程中遇到的条件分支收集输入约束；然后系统地逐个否定所有这些约束，并使用约束求解器求解；求解得到的解，被映射到执行不同程序执行路径的新输入；再使用系统搜索技术重复这个过程，试图扫描程序的所有可行的执行路径。

所以课件的结论是：与黑盒随机模糊相比，白盒模糊通常更精确，可以运行更多的代码，从而发现更多的 bug。

板书：模糊器三路 | ① 黑盒随机：正确格式 → 随机变异 → 跑 → 看异常（简单 hack，未测过就有效） | ② 基于语法：指定输入语法 + 模糊哪些部分 + 怎么模糊（靠人的创造力指导） | ③ 白盒模糊（微软研究院 2008）：执行 → 收集条件分支约束 → 逐个否定 → 约束求解 → 新输入走新路径 → 系统搜索扫可行路径 | 白盒更精确、跑更多代码、发现更多 bug

提问：同样是造畸形数据，如果被测对象是一个复杂的文件格式解析器，你会先选"黑盒随机变异"还是"基于语法的模糊"？为什么？

预设回答：

- 答"基于语法的模糊"：✅ 对。追问：它比黑盒随机多花了什么代价？——要先写出输入语法，人工成本更高，但造出来的数据能真正走到格式解析的深处。

- 答"黑盒随机变异"：✔️ 起步阶段可以，成本最低。点评：如果连一份正确样本都拿不到，黑盒随机反而无从下手；而如果格式复杂、嵌套很深，纯随机变异大多会在第一层格式校验就被挡回来。

- 答"白盒模糊最好，直接用白盒"：⚠ 方向对但要讲条件。点评：白盒模糊要有源码、要能符号执行和约束求解，代价最高；工程上常常是黑盒随机先摸底，再用语法和白盒做深水区。

易错点 / 考点：三个易混点。第一，把"白盒模糊"理解成"基于语法的模糊"——不是，白盒模糊的关键是收集条件分支的输入约束并逐个否定、用约束求解器求新输入，属于反馈驱动的符号执行路线。第二，记错首创者与年份，课件明确写的是微软研究院、2008 年。第三，"基于语法的模糊需要指定什么"是高频多选：输入语法、哪些输入部分要模糊化、如何模糊化，三条缺一不可。判断口诀：黑盒改数据、语法守格式、白盒解

约束。

> **过渡：**三类模糊器讲完，接下来看一个把"反馈驱动"做到极致的经典工具：AFL。

【3.81】American fuzzy lop 方法（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲模糊测试工具里最有名的一个：American fuzzy lop，简称 AFL。课件对它的定位是：一款开源的模糊测试工具，是当今使用最广泛的 Fuzzer，还被称为"效率之王"，因为它部署简单，旨在尽量降低对系统的影响。

它凭什么效率高？关键在插桩和遗传算法两个词。先说插桩：AFL 在程序执行之前，先对程序源码进行插桩——实际操作就是用 afl-gcc 或 afl-clang 这类工具重新编译被测对象，把统计代码注入进去。这样程序运行时，AFL 就能实时获取执行情况，知道这次输入走到了哪些分支。

再说遗传算法。AFL 用遗传算法对程序输入进行变异，在程序运行时注入自己的代码，自动产生测试用例进行模糊测试。它不是漫无目的地随机轰炸，而是"变异—观察—保留"的循环。

课件把整个算法简化成六步，我们一起过一遍。第一，将用户提供的初始测试用例加载到队列中；第二，从队列中获取下一个输入文件；第三，试图将测试用例修剪到不改变程序测量行为的最小尺寸，这一步是为了省时间；第四，使用均衡的、经过充分研究的各种传统模糊策略，重复地对文件进行变异；第五，如果任何产生的变异导致插桩记录新的状态转换，就将变异的输出作为一个新的条目添加到队列中；第六，转到第 2 步，继续循环。

把六步连起来看，AFL 的反馈驱动思想就清楚了：插桩记录到的"新的状态转换"就是奖励信号，只有能走到新路径的变异才配进入队列继续变异。用反馈把盲目的随机，变成有方向的探索。

课件还给了两个入口：AFL 的文档站与 Google 的 AFL 仓库，课后可以自己拿个开源小工具试跑一轮。

> **板书：**AFL（American fuzzy lop）＝ 开源、使用最广、效率之王 ｜ 两大抓手：插桩 instrumentation（afl-gcc／afl-clang 编译）+ 遗传算法变异 ｜ 六步：① 初始用例入队 → ② 取下一个 → ③ 修剪到最小尺寸 → ④ 传统模糊策略反复变异 → ⑤ 出现新状态转换就入队 → ⑥ 回第 2 步 ｜ 核心：反馈驱动（新路径＝保留信号） ｜ 兼容其他 fuzzer，可喂给半自动工具

> **提问：**AFL 的第五步说"只有导致插桩记录到新状态转换的变异，才作为一个新条目加进队列"。如果去掉这一条，AFL 会退化成什么？

> **预设回答：**

- 答"退化成黑盒随机模糊"：✅ 对。点评：队列里不再有新路径的种子接着变异，就只剩对初始用例反复随机改，失去了反馈驱动这条命脉，也就没有"效率之王"了。

- 答"速度会变快，因为不用判断新状态"：❌ 典型错误。点评：判断新状态的开销相对总执行次数很小，去掉它省不了什么，却会让覆盖率停止增长。

- 答"会变成纯遗传算法"：⚠ 不准确。点评：遗传算法的"选择"依据正是这条新状态转换记录，去掉它选择就失效了，剩下的只是随机变异。

> **易错点 / 考点：**考试常考三组对应关系：插桩靠重新编译（afl-gcc／afl-clang），变异靠遗传算法，保留靠新状态转换。易错点有两个：一是把"插桩"说成"在运行时打补丁"——插桩发生在程序执行前，是对源码编译时的处理；二是把 AFL 说成纯黑盒工具——它有插桩和反馈，属于反馈驱动（覆盖率引导）的模糊测试。另请记住第三步"修剪到最小尺寸"的目的：不改变程序测量行为，同时提高执行效率。

> **【思政落点 19】**AFL 把人从"一条条手写畸形数据"的重复劳动里解放出来：机器负责海量试错与状态记录，人负责挑种子、定策略、看结果。这正体现了技术创新的价值——工具不是来替代人的，而是替人扛走机械重复的部分，让工程师把精力投到更有创造性的问题上。同学们将来写代码、做测试，也要有这种"能用工具固化下来的经验，就不要靠人熬夜顶"的劳动观与效率意识。

> **过渡：**模糊测试衡量的是"程序扛不扛得住畸形输入"，那有没有方法衡量"我们这套测试用例本身够不够强"？下一页的变异测试，回答的就是这个问题。

【3.82】变异测试（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 接下来这一页，是 3.6 节里我最想让大家记住的方法——变异测试，英文 mutation testing。先读课件上的定义：变异测试是一种在细节方面改进程序源代码的软件测试方法；变异操作是模拟典型应用错误（定位代码的弱点），或者强制产生有效的测试。
>
> 变异测试要回答的不是"程序对不对"，而是"这套用例够不够狠"。做法是把手上的程序改坏一点点，每改一

处得到一个"变异体"；再拿同一套用例去跑原有程序和变异体，比较结果。用例发现了这个改动，变异体就被"杀死"了；跑完两个程序结果一样，变异体就"活"了。

课件例子很直白：原有代码写 if(p>q)，改成 if(p<q) 就得到一个变异体。右边表里小程序先是 z=x，再 z=z+y，然后 if(a>0) 返回 z，否则返回 2*x+z。变异体 1 把第一行变成 z=++x，变异体 2 把第二行变成 z=z+--y，这两个叫一阶变异体；变异体 12 是两者合并得到的二阶变异体。表格用两组用例 a=1 和 a=-1，给出四种程序在两种取值下的输出，请大家自己对比：哪个变异体被哪组用例杀死了。

由此引出两个概念。第一个是变异得分：被用例杀死的非等价变异体数量 ÷ 非等价变异体总数；得分越高，用例集越强。分母为什么强调"非等价"？这就带出等价变异体：有些变异改完后语义一点没变，和原程序行为一样，任何用例都杀不死它。这类变异体如果不识别剔除，变异得分就会被永远压低，让人误以为自己的用例不行。课件分类树里专门列了"等价变异体检测"这一支，讲的就是这个道理。

所以记住它的价值：衡量的不是程序正确性，而是用例集的强度。变异得分低不一定是程序有错，很可能是用例太温柔；把活的变异体一个个杀掉，就是在给用例集"补钙"。

板书：变异测试（mutation testing） | 问题：用例够不够狠（不是程序对不对） | 三步：取原程序 p → 用变异算子改一小处 → 得变异体；原程序 vs 变异体 比结果 | 杀死 mutant＝用例发现差异；存活＝用例没碰到 | 一阶／二阶变异体 | 变异得分＝杀死数 ÷ 非等价变异体总数 | 等价变异体：语义未变、杀不死 → 必须识别剔除

提问：某同学说自己写的用例把功能跑通了、bug 也报了几个，所以用例质量没问题；现在我给他的程序植入了 20 个变异体，结果只杀死了 5 个。请问这说明什么？如果他反驳说"那是变异体改得太偏，跟我的用例无关"，你怎么回应？

预设回答：

- 答"说明用例覆盖不够深，很多分支没碰到"：✅ 对。追问：那要不要立刻加用例？——要，而且加用例的过程正是变异测试的用途，加到把所有非等价变异体都杀死为止。

- 答"变异体是人为改的，本来就不该被发现"：❌ 典型误区。点评：变异操作模拟的正是"典型的应用错误"，真实开发者也常犯这种错；如果用例连人为植入的小错误都发现不了，凭什么相信它能发现真实缺陷？

- 答"先看看是不是等价变异体"：✅✅ 最专业的回答。点评：先把语义未变的等价变异体识别剔除，再算变异得分，才公平；但要注意，能被称为等价变异体的通常只是少数，不能拿它当万能挡箭牌。

易错点 / 考点：三个高频考点。第一，变异测试的对象是"测试用例集的强度"，不是被测程序本身，也不是找

程序里的缺陷——方向答反必错。第二，变异得分公式的分子分母都要限定在非等价变异体内，漏掉"非等价"三个字要扣分。第三，等价变异体是"改了等于没改"，杀不死不是用例的错，必须识别剔除；而"杀不死的非等价变异体"才是用例的漏洞。记忆三步法：改一处 → 跑两遍 → 看杀没杀死。

【思政落点 20】变异测试本质上是"自己给自己出难题、自己审自己"：没有人强迫你把程序改坏，是你主动植入缺陷，反过来检验自己的用例到底顶不顶用。这种自我加压、求真务实的态度，正是工程诚信的底色——报告上说"用例覆盖充分"，就得拿得出杀死变异体的证据，而不是凭感觉拍胸脯。将来无论做测试还是做审计，都要敢于用最苛刻的标准检验自己交出的成果。

过渡：变异体怎么生成、等价变异体怎么剔除、用例不够时怎么办？下一页用两张图，把变异测试的方法体系和完整流程收口。

【3.83】变异测试的方法与流程（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页两张图把变异测试的方法体系与流程讲清。左边分类树，根为"变异测试"，下分原理、优化、应用三支。

原理支下有三项：基本假设、分析流程、等价变异体检测。它被放在"原理"层，可见是方法正确性的前提。

优化支最大，分两类：选择优化——随机选择法、聚类选择法、变异算子选择法、高阶变异体优化；执行优化——变异体检测优化、变异体编译优化、并行执行，以及面向结构语言、面向对象语言等。变异体一多，每个都要编译、都要跑全部用例，成本极高，所以要"选得准、跑得快"。

应用支有三项：测试用例生成、用例集充分性、回归测试。"用例集充分性"正是上一页"用例够不够狠"的工程落点；回归测试则看改了代码后，原有用例集还能不能守住底线。

右边流程图：从原有程序 p 出发，配变异算子生成变异体；识别并剔除等价变异体，剩下非等价变异体；再拿用例集 T 在变异体上执行，判断是否检测出所有变异体？是，退出；否，就往用例集加新用例，回到执行这步继续杀。

这个闭环就是工作节奏：杀不死的变异体标记用例空白区；补用例、再杀，直到非等价变异体全被干掉。

板书：分类树 ｜ 原理：基本假设 · 分析流程 · 等价变异体检测 ｜ 优化：选择优化（随机／聚类／算子选

> 择／高阶变异体）+ 执行优化（检测／编译／并行／面向结构语言／面向对象语言）｜应用：用例生成 · 用例集充分性 · 回归测试｜流程：p + 变异算子 → 变异体 → 识别等价变异体 → 非等价变异体 → 在变异体上执行用例 T → 全检测出？是→退出；否→添加用例 → 回到执行

> **提问：**流程图中"是否检测出所有变异体"这一步答"否"时，为什么是去"添加测试用例"，而不是去"修改变异体"或"换个变异算子"？

> **预设回答：**

- 答"因为要检验的是用例集，用例杀不死就说明用例有缺口"：✅ 对。点评：变异体是考题，用例是考生；考题本身不合理（等价变异体）属于要剔除的范围，但不能因为考生答不出就把考题改简单。

- 答"换个变异算子重新生成变异体"：❌ 方向错了。点评：变异算子的选择属于方法优化，发生在生成阶段，不能用来回避流程里暴露出来的用例空白。

- 答"也可能是等价变异体没识别干净"：✅✅ 补充得很好。点评：所以正确的顺序是先把等价变异体识别剔除，再进入这个判断，否则会白补一堆用例。

> **易错点 / 考点：**本页最容易考的是流程图的顺序，尤其两个位置：等价变异体的识别发生在"执行测试用例之前"，用例的补充发生在"判断未全部检测出之后"。选择题常把"先执行用例再识别等价变异体"作为干扰项，那是错的。另外记住分类树上"等价变异体检测"挂在"原理"支下，而不是"优化"支下——它关乎方法能不能算得准，不只是效率问题。

> **过渡：**到这里，基于模型的测试方法（MBT）这一节就讲完了。接下来进入本章理论门槛最高的一节：3.7 形式化方法。

【3.84】3.7 形式化方法（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 到这里，3.6 基于模型的测试方法告一段落，接下来进入 3.7 形式化方法，分三个小节：3.7.1 形式化方法、3.7.2 形式化验证、3.7.3 扩展有限状态机方法。
>
> 为什么要有形式化方法？教材讲得清楚：用自然语言描述需求时，语义不够清晰、容易有歧义；分析结果往往依赖参与者的经验和理解，没有严格量化的标准。后续活动都会受影响：对功能特性的验证缺乏客观、定量的依据，具有不确定性，度量也不可靠。

这些问题，一般应用软件可以接受。但有几种系统不能接受：核电站控制软件、航天器的控制系统、导弹防御系统。那里一个需求歧义不是"重做一遍"能补救的，代价可能是生命和环境。所以要换一套以数学和逻辑为基础的表达方式：语义清晰、无歧义，还能借助工具做自动化分析、编码和验证。

板书：3.7 形式化方法 | 3.7.1 形式化方法 · 3.7.2 形式化验证 · 3.7.3 扩展有限状态机方法 | 动因：自然语言 → 语义不清、有歧义、依赖经验、无量化标准 → 关键系统不可接受（核电站控制 · 航天器控制 · 导弹防御） | 出路：数学与逻辑学 + 形式语言 → 语义清晰、可自动化分析/编码/验证

提问：需求文档里写"系统应快速响应用户请求"，这句话为什么在核电站控制软件里是不能接受的？请你指出它缺了什么。

预设回答：

- 答"没说多快，也没说响应什么"：✅ 对。追问：把它形式化，要补上什么？——补上可度量的量（时长上限）、明确的对象（哪个接口、哪类请求）和判定条件。

- 答"这句话没问题，开发看得懂就行"：❌ 典型误区。点评："看得懂"是人的主观判断，不同人理解不同；安全关键系统要的是任何人、任何工具读出来的含义都一样。

- 答"这是需求问题，不是测试问题"：⚠ 只说对了一半。点评：需求含糊，后面的验证就没有客观依据，测试也就无法证明"满足了这条需求"，这正是本章要引入形式化方法的原因。

易错点 / 考点：考点是"形式化方法的动因与适用对象"这组对应：动因＝自然语言描述的歧义性与主观性，适用对象＝高可靠性、安全关键的系统。选择题常把"所有软件都必须用形式化方法"设为干扰项，那是错的——课件明确说形式化方法不是万能灵药，一般应用软件的歧义是可以接受的。另外记住本节三个小节的顺序与名称，3.7.3 是"扩展有限状态机方法"，不是"有限状态机方法"。

过渡：先看 3.7.1，什么叫形式化方法，它靠什么语言、分哪几类、代价在哪里。

【3.85】3.7.1 形式化方法（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

先看定义。课件上说：形式化方法是基于数学的方法（数学表示、精确数学语义）来描述目标软件系统属性的一种技术。判断标准很干脆——凡是采用严格的数学语言、具有精确的数学语义的方法，都称为形式化方法。形式化的核心不是"图画得漂亮"，而是"每个符号都有确定含义，谁读出来都是同一个意思"。

形式化靠形式化规范说明语言落地。它由三个主要成分构成：语法，定义用于表示规约的特定符号；语义，帮助定义用于描述系统的对象及其属性；一组关系，定义如何确定某个对象是否满足规约。可记成三句话：用什么符号写、什么意思、怎么判断满足不满足。

它的作用范围在软件规格和验证之上，包括软件系统的精确建模和规格特性描述，因此可看作两类：面向模型的形式化方法和面向属性的形式化方法。

具体方法有五类。基于模型的方法，明确定义状态和操作，让系统在状态间转换，能表示时间这类非功能性需求，但并发性弱；代数方法，通过操作间的行为关系隐式定义操作，不定义状态；过程代数方法，给出并发过程的显式模型，用过程间通信表示系统行为；基于逻辑的方法，用逻辑描述预期性能，再用公理系统证明；基于网络的方法，用带形式语义的图形语言给出并发模型。具体语言下一页专门讲。

最后必须泼一盆冷水，课件专门强调了：形式化方法不是解决开发问题的万能灵药，它也有缺点，也不能保证不出现错误。第一，非形式化的客观需求与形式化规范的关系难以处理好——用户那句话怎么"翻译"成公式，最容易出错；第二，理论上对规范深入分析可以证明所需性质，但实践比较复杂；第三，强有力的支持工具仍然缺乏。另外，UML 只能算半形式化的方法，有较清晰定义的形式和部分语义定义，并支持自动生成代码和测试用例。

板书：形式化方法＝基于数学（数学表示 + 精确数学语义）描述目标软件系统属性 ｜ 规范语言三要素：语法（符号）· 语义（对象与属性）· 一组关系（是否满足规约） ｜ 分类：基于模型（Z、B）· 代数（OBJ、CLEAR、ASL、ACT）· 过程代数（CSP、CCS、ACP、LOTOS、TPCCS）· 基于逻辑（区间时序逻辑、Hoare 逻辑、模态逻辑、时序逻辑、时序代理模型）· 基于网络（Petri 网、谓词变换网） ｜ 代价：需求与规范难对应 · 证明复杂 · 工具缺乏 ｜ UML＝半形式化

提问：UML 为什么只能叫"半形式化"方法？如果项目组已经画了完整的 UML 图，还需要形式化方法吗？

预设回答：

- 答"UML 有较清晰定义的形式和部分语义定义，还能自动生成代码和测试用例，但语义没有完全形式化"：✅ 对，这几乎是课件的原文口径。

- 答"UML 就是形式化的，画了图就不用别的了"：❌ 误区。点评：部分语义定义意味着存在解释空间；对安全关键系统，解释空间就是风险。

- 答"看系统风险等级决定"：✅✅ 最实务。点评：一般系统用 UML 足够；核电站控制、航天器控制这类系统，才值得付出形式化的高成本。

易错点 / 考点：三个必记点。第一，形式化规范说明语言的三要素——语法、语义、一组关系，多选题常考，别把"一组关系"漏掉。第二，五类方法的名称与其代表语言的对应，尤其是"基于模型 = Z／B""基于网络 = Petri 网"。第三，课件对形式化方法的态度是辩证的：有优势，但不是万能灵药，也不能保证不出现错误——考题若说"形式化方法可保证软件无错误"，判错。

【思政落点 21】形式化方法为什么最常用在核电站控制、航天器控制、导弹防御这类系统上？因为那里的每一行需求都连着人的安全。愿意花高成本把需求写成精确的数学语义、再逐条证明属性，本质上是对生命的敬畏、对工程质量的精益求精。我们做测试也一样：越是关系重大的功能，越不能靠"我觉得没问题"，而要靠可检验的证据和严格的标准——这就是工程人应有的责任担当。

过渡：形式化规范到底长什么样？下一页用最经典的巴科斯范式，给大家一个看得见摸得着的例子。

【3.86】示例（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页给一个最朴素的形式化规范语言例子：巴科斯范式，Backus–Naur Form，简称 BNF。别看它朴素，编程语言的语法定义大多用它，测试数据生成也离不开它。

先认识几个符号。尖括号里的名字，比如 postal-address，叫非终结符，意思是"还要继续展开"；双引号里的内容，比如"Sr."，是终结符，就是字面上要出现的字符串；中间的 ::= 读作"定义为"；竖线 | 读作"或者"，表示可以取其中任意一条产生式。

我们看第一行：<postal-address> ::= <name-part> <street-address> <zip-part>。意思是，一个邮寄地址由三部分按顺序拼成——姓名部分、街道地址部分、邮编部分。再看 <name-part>，它有两条产生式，用竖线分开：一条是 <personal-part> <last-name> <opt-jr-part> <EOL>，另一条是 <personal-part> <name-part> <EOL>。请特别注意第二条：<name-part> 出现在自己产生式的右边，这叫递归，正是它让我们能写出"姓 名 名 名"这种多段姓名。还有 <opt-jr-part>，它的最后一个选项是空串，表示这个位置可以什么都没有——这就是 BNF 表达"可选"的方式。

这一页的重点不是背下这几行规则，而是体会"精确"两个字：自然语言里含糊的"地址"概念，被拆成了一条条没有歧义的规则，有了这套文法，工具就能顺着它自动生成语法分析器，也能顺着它自动造出符合格式的测试数据——大家回想上一节讲的基于语法的模糊，它要指定的"输入语法"，正是这种文法。

> **板书：**示例：巴科斯范式 BNF ｜ 非终结符 <xxx>（还要展开） ｜ 终结符"xxx"（字面字符） ｜ ::= 定义为 ｜ | 或者 ｜ <postal-address> ::= <name-part> <street-address> <zip-part> ｜ 递归：<name-part> 出现在自己右侧 ｜ 空串＝可选（<opt-jr-part>） ｜ 用途：精确无歧义 → 自动生成解析器与测试数据

> **提问：**在 <name-part> 的两条产生式里，为什么必须保留带递归的那一条？如果只留第一条 <personal-part> <last-name> <opt-jr-part> <EOL> 会有什么后果？

> **预设回答：**

- 答"就只能表示一个名加一个姓，中间名写不出来"：✅ 对。追问：加了递归能表示什么？——可以表示任意多个名字段，规则长度却不变。

- 答"两条都可以，随便留一条"：❌ 典型错误。点评：删掉递归那条，语言被削弱；删掉非递归那条，就没有递归出口，规则永远展开不完，文法无法终止。

- 答"可以用重复符号代替递归"：⚠ 在其他记法里可以，但在本页这套 BNF 里没有重复记号，只能靠递归表达"若干个"。

> **易错点 / 考点：**三个易错点。第一，符号读法：::= 是"定义为"，不是"等于"，| 是"或"，两个符号在选择题里经常互换来设陷阱。第二，递归必须同时有递归分支和终止分支，缺一个文法就不成立——这是常考的判断题。第三，空串选项表达"可选"，不要把它当成"缺失"或"错误"。判断口诀：尖括号待展开、引号内即字面、竖线挑一条、递归要出口。

> **过渡：**看了一个小例子，我们退回来问一个更大的问题：形式化方法在整个开发流程里处在哪几步？下一页的"形式化三部曲"来回答。

【3.87】形式化三部曲（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页讲形式化方法在流程上的三个环节，课件称之为"形式化三部曲"：形式化描述、形式化开发、形式化验证。
>
> 形式化描述，是用形式化的规范语言把需求或设计写下来，得到一份语义精确、没有歧义的规格说明；形式化开发，是在规格基础上做细化，一步步把抽象规格变成可执行的东西，每一步细化都可被检查；形式化验证，是拿形式规范或属性，用形式逻辑方法证明正确性或者非正确性。三个词连起来，就是形式化方法的核

心思想：先写清楚，再往下做，边做边证。

右边这张图来自产品线工程。中间是一个统一底座：Product Line Models，用 HATS ABS 表达，带有统一的形式语义。上下两圈是围绕它展开的活动：上面是产品线层面，有一致性分析、复用正确性、产品线可视化、测试用例生成、确认与验证、产品线演进；下面是应用层面，有快速原型、代码生成、产品可视化、测试用例生成、确认与验证、产品演进。

请特别注意两点。第一，所有这些分析和生成活动，都向中间那个"统一形式语义"汇聚，又都从它出发——一份精确的模型，同时支撑一致性分析、代码生成、测试用例生成和验证，这就是"一份规格、多处复用"的威力。第二，测试用例生成和验证在上下两圈都出现，说明在形式化的路子里，测试不是事后补的，而是和模型一起被"生成"和"证明"出来的。

板书：形式化三部曲：形式化描述 → 形式化开发 → 形式化验证 ｜ 核心：统一形式语义（Product Line Models，HATS ABS） ｜ Family Engineering：一致性分析 · 复用正确性 · 产品线可视化 · 测试用例生成 · 确认与验证 · 产品线演进 ｜ Application Engineering：快速原型 · 代码生成 · 产品可视化 · 测试用例生成 · 确认与验证 · 产品演进 ｜ 一份规格 → 同时支撑分析、生成、验证

提问：为什么在图里，"test case generation（测试用例生成）"和"validation, verification（确认与验证）"在产品线和应用两个层面都出现了？这说明了什么？

预设回答：

- 答"因为两个层面都要保证质量，都要生成用例、都要验证"：✅ 对。追问：它们的依据是什么？——同一份带统一形式语义的模型。

- 答"图画重复了"：❌ 错误。点评：产品线层面验的是"这条产品线的共性规格"，应用层面验的是"具体产品的规格"，对象不同，不能省。

- 答"说明在形式化方法里测试是被模型自动推导出来的"：✅✅ 更进一步。点评：这正是形式化方法能把测试左移、并让测试依据客观化的关键。

易错点 / 考点：第一，三部曲的顺序与名称必须记准：形式化描述、形式化开发、形式化验证——判断或填空题常把"形式化测试"塞进来当干扰项，那是错的。第二，理解"统一的形式语义"是整个体系的枢纽：它是所有分析、生成、验证活动的公共依据。第三，记住形式化方法里测试用例是可以由模型生成的，这与后面有限状态机、EFSM 的图覆盖准则直接衔接。

过渡：三部曲讲的是流程，那具体用哪些形式化语言和方法来落地？下一页给出五类方法的名录。

【3.88】形式化的具体方法（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

上一页讲流程，这一页把"具体有哪些形式化方法"列清楚。课件分五类，每类记一两个代表就够用了。

第一类，基于模型的方法，比如 Z 语言、B 语言。它的特点是明确定义状态和操作，让系统从一个状态转换到另一个状态；可以表示时间这类非功能性需求，但对并发性表示得不好。

第二类，代数方法，如 OBJ、CLEAR、ASL、ACT。它通过联系不同操作间的行为关系来给出操作的隐式定义，而不定义状态——也就是说，它不画状态机，它写等式，同样也没有并发的显式表示。

第三类，过程代数方法，如 CSP、CCS、ACP、LOTOS、TPCCS。它给出并发过程的一个显式模型，并通过限制所有容许的、可观察的过程间通信来表示系统行为。要测并发、要验通信协议，就找这一路。

第四类，基于逻辑的方法，如区间时序逻辑、Hoare 逻辑、模态逻辑、时序逻辑、时序代理模型。它用逻辑描述系统预期的性能，包括底层规约、时序和可能性行为，然后采用与所选逻辑相关的公理系统证明系统具有这个性能。这类方法还有一个特点：可以用具体的编程构造扩充逻辑，得到一种广谱的形式化方法，通过正确的细化步骤集来开发系统。

第五类，基于网络的方法。它根据网络中的数据流显式地给出系统的并发模型，包括数据在网络中从一个节点流向另一个节点的条件；采用的是具有形式语义的图形语言，因而易理解，典型代表是 Petri 网、谓词变换网。

这一页的清单不难，难在选型：如果要表达状态转换，选基于模型；如果要刻画操作之间的代数关系，选代数方法；如果要描述并发与通信，选过程代数或基于网络；如果要证明时序性质，选基于逻辑。记忆口诀：模型看状态，代数写等式，过程代数讲通信，逻辑做证明，网络画数据流。

板书：五类形式化方法 ｜ ① 基于模型：Z、B（状态 + 操作；可表时间，弱并发） ｜ ② 代数：OBJ、CLEAR、ASL、ACT（隐式定义操作，不定义状态） ｜ ③ 过程代数：CSP、CCS、ACP、LOTOS、TPCCS（并发显式模型，通信刻画行为） ｜ ④ 基于逻辑：区间时序逻辑、Hoare 逻辑、模态逻辑、时序逻辑、时序代理模型（公理系统证明） ｜ ⑤ 基于网络：Petri 网、谓词变换网（数据流 + 形式语义图形语言） ｜ 口诀：模型看状态／代数写等式／过程代数讲通信／逻辑做证明／网络画数据流

> **提问：**要为一个"并发通信协议"写形式化规格，你会优先选哪一类方法？请说出理由，并指出哪一类方法在这件事上先天不足。

> **预设回答：**

- 答"过程代数方法，比如 CSP、LOTOS"：✅ 对。点评：它给出并发过程的显式模型，用可观察的过程间通信刻画行为，正对协议这件事；基于网络的方法（Petri 网）同样可用。

- 答"基于模型的方法，用 Z 或 B"：⚠ 不是不可以，但有短板。点评：课件明确说基于模型的方法不能很好地表示并发性，它会写得很吃力。

- 答"代数方法，写两个等式就完了"：❌ 方向错。点评：代数方法不定义状态、也没有并发的显式表示，用它描述协议会丢掉时序关系。

> **易错点 / 考点：**最常考"方法—代表语言—优缺点"的对应关系。三个易错细节：第一，基于模型的方法强在状态与操作、能表示时间这类非功能性需求，弱在并发；第二，代数方法"不定义状态"，很多人误以为它和基于模型是一回事；第三，基于网络的方法的关键词是"数据流"和"图形语言（Petri 网）"，不要和过程代数混起来。记"强弱两句"：模型能表时间弱并发；代数不定义状态且无显式并发。

> **过渡：**规范写好了，接下来最关键的一步是证明。下一页进入 3.7.2 形式化验证。

【3.89】3.7.2 形式化验证（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 进入 3.7.2 形式化验证。定义是：根据某些形式规范或属性，使用形式逻辑方法证明其正确性或非正确性。请注意后半句——"或非正确性"，形式化验证不只确认"对"，也能证明"不对"，也就是把反例找出来。
>
> 它的做法是：通过形式化规范进行分析和推理，研究它的各种静态和动态性质，验证是否一致、是否完整，从而找出错误和缺陷。它首先被用于生成软件规格说明书，作为软件开发的基础和软件测试验证的依据；因为描述基于严格定义的规范语言，就可以借助工具完成验证。
>
> 那为什么不能用老办法？传统验证方法包括模拟和测试，都是通过实验的方法对系统查错，分别在系统的抽象模型和实际系统上进行：在系统的某点给予输入，观察它的输出，要完成大量数据输入和输出结果的检查。问题在于——实验所能涵盖的输入有限，很难找出所有潜在的错误。早期的形式验证，主要研究用数学方法证明程序正确性。

接下来是这页最要命的一句话，课件单独列了出来：无法证明某个系统没有缺陷，因为不能定义"没有缺陷"。我们只能证明一个系统不存在"我们可以想得到的缺陷"，以及验证它满足系统质量要求的属性。这句话和第二章的测试原则一致——测试无法证明程序没有错误，形式化验证也不能。区别在于：传统测试靠有限实验，覆盖的输入有限；形式化验证靠逻辑推理，能把"想得到的缺陷"一次性排除干净。所以说，形式化方法大幅提高了可信度，但它给出的仍是"相对于某个规范的正确性"，而不是"绝对无缺陷"。

因此判断这类题有现成三步法：第一，看它证明的对象是"属性"还是"无缺陷"；第二，若说"形式化验证能保证系统完全没有缺陷"，一定判错；第三，正确性永远相对于规范——规范本身写错了，验证再严密也救不回来。

板书： 3.7.2 形式化验证＝依据形式规范或属性，用形式逻辑方法证明正确性"或非正确性" | 做法：分析推理规范 → 研究静态/动态性质 → 验一致性、完整性 → 找出错误与缺陷 | 传统验证：模拟 + 测试（实验法，输入有限，难穷尽） | 关键论断：无法证明系统"没有缺陷"，因为不能定义"没有缺陷"；只能证明不存在我们想得到的缺陷 + 满足质量要求的属性 | 结论：可信度更高，但仍是"相对于规范"的正确性

提问： 有人说"既然形式化验证也不能证明系统没有缺陷，那它和普通测试就没区别了"。这个说法错在哪里？

预设回答：

- 答"区别在排除缺陷的方式和范围：测试靠有限实验，形式化验证靠逻辑推理，能把想得到的一类缺陷一次性排除"：✅ 对，这正是本页的核心对比。

- 答"没区别，反正都不能保证没有缺陷"：❌ 典型错误。点评：两者都不能保证绝对无缺陷是共同点，但手段、成本、可信度完全不同；把"都不能绝对保证"等同于"没区别"，是把结论当成了全部。

- 答"区别是形式化验证能生成规格说明书，测试不能"：⚠ 沾边但没抓到重点。点评：本页确实说形式化验证首先被用于生成软件规格说明书，但更本质的差别在"实验穷举"与"逻辑证明"。

易错点 / 考点： 本页是高频判断题来源。三句话必须背准：①形式化验证证明的是正确性或非正确性；②不能证明系统"没有缺陷"，因为"没有缺陷"无法形式化定义；③只能证明不存在"我们想得到的缺陷"，以及满足质量要求的属性。常见错误选项："形式化验证可以证明软件完全无缺陷""形式化验证属于动态测试"。另外记住传统验证方法是模拟和测试，并且它们在系统抽象模型和实际系统两个层面上分别进行。

过渡： 那么，形式化验证在工程上到底有哪些可用的方法？下一页把名单列出来。

【3.90】形式化验证的一些具体方法（1 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页列的是形式化验证的一些具体方法。先记一句总纲：这些方法多数是基于模型的，也可以归为 MBT 方法——也就是说，它们和我们讲的基于模型的测试是一家人，只是把模型换成了带精确语义的形式模型。

第一项，有限状态机 FSM 或扩展有限状态机 EFSM，用来做模型检验，这也是本章 3.7.3 节的主角。

第二项，采用 SPIN 和线性时序语言，验证系统的相关属性。SPIN 是并发系统模型检验的经典工具，线性时序语言负责把"性质"写成公式。

第三项，UML 语义转换形式化验证。也就是把半形式化的 UML 模型转换成更严格的形式模型再验证。

接下来两项和访问控制有关。第四项是标准 RBAC 模型，它包括四个部件模型：基本模型 RBAC0，也就是 Core RBAC；层次模型 RBAC1，Hierarchal RBAC；角色限制模型 RBAC2，Constraint RBAC；以及组合模型 RBAC3，Combines RBAC。第五项是扩展的 RBAC 模型和基于粒计算的 RBAC 模型（G-RBAC）。为什么要专门验证这类模型？因为它管的是"谁能访问什么"，权限模型一旦有漏洞，往往就是最严重的安全问题。

第六项，符号模型检验（Symbolic Model Checking），把问题形式化为特定时序逻辑下的问题来求解，用到的技术包括 BDD 二叉决策图、SAT 可满足性求解器、ATPG 自动测试模式发生器，以及定理证明器。第七项，BAN 逻辑模型，用于安全协议的验证。

这一页名字多、信息量大，大家先不要死抠工具名，记住三条脉络就够：状态模型、逻辑与时序、访问控制与安全协议。判断题爱考的是"这些方法多数基于模型，因此也可以归入 MBT 方法"这一句。

板书：具体方法 ｜ ① FSM／EFSM 模型检验 ② SPIN + 线性时序语言 ③ UML 语义转换 ④ 标准 RBAC：RBAC0 基本 · RBAC1 层次 · RBAC2 角色限制 · RBAC3 组合 ⑤ 扩展 RBAC 与粒计算 RBAC（G-RBAC） ⑥ 符号模型检验：BDD · SAT · ATPG · 定理证明器 ⑦ BAN 逻辑（安全协议验证） ｜ 总纲：多数基于模型 → 可归入 MBT

提问：RBAC 这一类模型为什么要专门做形式化验证？它和 FSM、SPIN 这类方法关注的东西有什么不同？

预设回答：

- 答"RBAC 管权限分配，出错就是安全事故，所以要证明它没有越权路径"：✅ 对。追问：证明的目标是什么？——验证权限模型的一致性、完整性，确认不存在违反安全策略的访问路径。

- 答"都是验证，没区别"：❌ 太粗。点评：FSM／SPIN 关注系统的状态迁移与时序性质，RBAC 关注访问控制策略的结构与约束，验证对象不同。

- 答"因为有 RBAC0 到 RBAC3 四层，层次多所以要验证"：⚠ 沾边。点评：层次化确实增加复杂度，但根本原因还是安全属性必须被证明而不只是被测试。

> **易错点 / 考点：**三个易错点。第一，把"符号模型检验"和"BAN 逻辑模型"弄混：前者关键词是 BDD／SAT／ATPG／定理证明器，后者专门用于安全协议验证。第二，RBAC 四层模型的名称与缩写必须一一对应（RBAC0 基本、RBAC1 层次、RBAC2 角色限制、RBAC3 组合），多选题常考。第三，忘记"多数基于模型、可归为 MBT 方法"这个总纲——它是把形式化验证接回本章知识框架的关键一句话。

> **过渡：**名单里排第一的 FSM／EFSM 是本节的落点，也是本章唯一动手建模的部分。下一页先讲清楚有限状态机到底是什么。

【3.91】有限状态机（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页讲有限状态机。课件的定义是：有限状态机（Finite State Machine，FSM）是对象行为建模的工具，用来描述对象在其生命周期内所经历的状态序列，以及如何响应来自外界的各种事件。

这里把定义拆成三层。第一，它建模的是"对象"的行为：软件系统里一个对象无论简单还是复杂，都必然经历从开始创建到最终消亡的完整过程，这就是对象的生命周期。第二，它描述的是"状态序列"——对象在生命周期内会停在哪些状态上、按什么顺序走。第三，它描述"如何响应外界的各种事件"，也就是什么事件会触发什么状态迁移。

那什么样的对象值得花力气去建模？课件点出了关键：系统都必须维护一两个非常关键的对象，它们通常具有非常复杂的状态转换关系，要对外部的各种异步事件进行响应。教材举的例子是 VoIP 电话系统里的电话类 Telephone：它要响应来自对方的随机呼叫、来自用户的按键事件，以及来自网络的信令。这种"行为完全依赖当前状态、又被异步事件驱动"的对象，用状态机描述最合适。

右边这张图是 FSM 对象在实现层面的构造：中心是 FSM 对象构造器，一边与 FSM 初始化交互（创建表元素数组、表变量数组），一边向下构造出状态转换对象数组和状态变量数组；状态变量数组再连到"表对象"，表对象创建状态数组，状态数组又连回状态转换对象数组。说白了，状态机在代码里就是"表加数组"。

最后交代它和测试的关系：FSM 模型包含输入符号、输出符号、状态集合等要素，基于它可以做模型检验，也可以做基于模型的测试——遍历状态、遍历迁移、遍历路径，就得到不同的覆盖准则；把动作和转移条件也加进去，FSM 就扩展成 EFSM，能进一步处理系统的数据流问题。

板书：有限状态机 FSM＝对象行为建模工具 ｜ 三层含义：对象生命周期 · 状态序列 · 响应外界事件 ｜ 值得建模的对象：关键对象 + 复杂状态转换 + 响应异步事件（例：VoIP 的 Telephone） ｜ 实现构造：FSM 对象构造器 ←→ FSM 初始化（表元素数组／表变量数组）→ 状态转换对象数组 + 状态变量数组 → 表对象 → 创建状态数组 ｜ 用途：模型检验 + 基于模型的测试（状态／迁移／路径覆盖） ｜ 加动作与转移条件 → EFSM（处理数据流）

提问：为什么说电话类 Telephone 最适合用状态机来建模？请从"行为依赖当前状态"这一点解释。

预设回答：

- 答"因为电话的行为完全取决于当前状态，同一按键在不同状态下含义不同"：✅ 对。追问：举一个例子？——通话中按挂断是结束通话，待机时按同样键可能毫无意义，行为由状态决定。

- 答"因为电话状态少，画状态机方便"：⚠ 只说对了一半。点评：状态少确实便于建模，但决定性理由是行为与状态的强绑定关系。

- 答"用普通函数或流程图也能写"：❌ 忽视要点。点评：流程图擅长表达顺序逻辑，很难清晰表达"被异步事件打断并在多个状态间跳转"的行为，容易漏掉状态组合。

易错点 / 考点：三个考点。第一，FSM 定义的三个关键词——状态序列、生命周期、响应外界事件，简答题按这三点写就不会漏。第二，FSM 与 EFSM 的区别：EFSM 在 FSM 基础上增加了动作和转移条件（并增加了初始状态，把原来的"输出函数"变为"变量集合和转移集合"），从而能处理系统的数据流问题；这一句几乎每年都考。第三，FSM 的测试可以直接借用图覆盖准则（状态覆盖、迁移覆盖、路径覆盖），这与前面 3.6.1 节的功能图、状态迁移测试是一套东西，不要当成新知识死记。

过渡：FSM 讲完，接下来看它的扩展形式 EFSM：多出的动作、转移条件和初始状态到底解决了什么问题，以及怎样基于它自动选取测试数据。

【3.92】示例：状态图（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

上一页我们讲了有限状态机，也就是 FSM，说它是"描述对象在其生命周期内所经历的状态序列，以及如何响应外界事件"的工具。这一页就来看一台活生生的状态机：一个堆栈的状态图。

先教大家怎么看这张图，看五样东西。第一，左边那个实心黑圆，是初始状态；第二，下面那个双圈，是终止状态；第三，中间三个圆角矩形是状态，分别叫 empty、filled、full，翻译过来就是空、非空、满；第四，箭头是迁移，箭头旁边写的是"事件"；第五，很多箭头上还有一对方括号，里面是守卫条件，也就是这件事在什么条件下才允许发生。

现在把迁移一条条读过去。从初始状态出发，init 这个迁移把系统带进 empty。在 empty 上，push 走到 filled，delete 直接走到终止状态——也就是把空栈销毁掉，这是合法结局。到了 filled 这一层条件就出来了：push 且 height 小于 max 减 1，留在 filled；pop 且 height 大于 1，也留在 filled；而 push 且 height 等于 max 减 1，就进入 full。full 上 pop 回到 filled，top 留在 full；filled 上 pop 且 height 等于 1，就回到 empty。

这里有两个记号要特别说。一个是 top，它是读栈顶元素，读完栈没变，所以在图上画成自己指向自己的自环；另一个是 full 上面的"push 星号"，这个星号的意思是"这个操作在当前状态下对状态没有影响"，也就是栈满了还压，系统把它忽略掉——它不是没发生，是被吸收了。

为什么一定要写守卫条件？因为"能不能压、能不能弹"根本不取决于操作本身，而取决于栈里现在有几个元素。不把 height 写清楚，这张图就说不清任何事。这个道理放到别的系统上一样成立：电梯的"上行"按钮在顶楼按了没用，自动售货机的"退币"在没投币时无币可退，都是同一个逻辑。

最后提一句这张图的来历：课件标注它来自 nach Spillner、Linz 的《Basiswissen Softwaretest》2005 年那本书，是德国软件测试的经典教材。请记住一点——这张图不是画着好看的，它就是**模型**，测试用例要从它长出来。

板书：状态图五要素 → 初始态（实心圆）· 终止态（双圈）· 状态（empty｜filled｜full）· 迁移（事件〔守卫条件〕）· 动作；迁移清单：init／push／pop／top／delete；守卫：push〔height＜max−1〕· push〔height＝max−1〕· pop〔height＞1〕· pop〔height＝1〕；push ✳ ＝ 对状态无影响（忽略）｜top ＝ 只读不自环变化

提问：假设 max 是 5，栈里现在有 3 个元素，处在 filled 状态。此时我分别按 push、pop、top，图上的迁移分别走哪一条、状态变成什么？

预设回答：

- 答"push 走 filled 上的自环，还是 filled；pop 也还是 filled；top 也还是 filled"：✅ 对，因为 3 既不等于 max−1，也不等于 1，三个操作都在守卫条件允许的自环里。追问：那什么时候才跳出 filled?

- 答"push 就进 full"：❌ 典型错误，忘了守卫条件。点评：只有 height＝max−1 时 push 才进 full，也就是 4 个元素时再压一个。这说明读状态图最容易漏掉的就是方括号。

- 答"top 会把栈顶弹掉"：❌ 概念混淆。点评：top 是读、pop 是取，读操作在图上表现为自环，这不是"没画"，而是"状态确实没变"。

> **易错点 / 考点：**三个高频混淆——①把"状态"和"事件"搞反（empty 是状态，push 是事件）；②漏看守卫条件，以为 empty 上什么都能干（空栈 pop、空栈 top 在下一页的树里都通向 ERROR）；③不理解星号 ✳ 的含义，把它当成未知或错误。考试常见形式：给一张状态图，数状态数与迁移数；或给一条操作序列，判断最终状态、判断是否合法。判断三步法：**先看当前状态 → 再看事件 → 最后看守卫条件**，三步都过了才是合法迁移。

> **过渡：**状态图是"有环"的图，有环就意味着路径有无穷多条，穷举测试在数学上就不可能。要把无限变成有限，就得把环掰开——下一页讲状态图怎么转化成树结构，并据此生成测试用例。

【3.93】状态图转化为树结构、生成测试用例（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 这一页是本片第一个要讲透的技术点：状态图怎么变成树，再从树生成测试用例。请大家跟着我一步一步走。
>
> 第一步，定根。树根就是初始状态 initial，也就是上一页那个实心黑圆。
>
> 第二步，展开。从根出发，把这个节点上所有可能的事件分支都画成子节点，子节点的名字就是这条迁移**到达的状态**。所以 empty 下面会长出多个孩子。
>
> 第三步，剪枝，这是最关键的一步。当一条迁移指回到一个已经出现过的状态时，就在这个位置停住，把它当成叶子，不再往下展开。因为如果继续展开，就会得到无限的路径，永远画不完。图上 filled 出现了三次、full 出现了三次，就是这么来的——它们不是重复画出错，而是"展开到回边就停"的结果。
>
> 第四步，保留 ERROR 出口。大家看这棵树上有一批灰色的 ERROR 叶子：empty 上 pop 到 ERROR，empty 上 top 到 ERROR，filled 上 delete 到 ERROR，full 上 delete 到 ERROR。翻译成业务语言就是：空栈不能弹、不能读顶，非空或满的栈不能直接删除。这些路径是模型明确告诉我们"不该被允许"的，恰恰是缺陷最集中

的地方，绝不能因为它们是"错误路径"就不写用例。

第五步，读路径。树上每一条从根到叶子的路径，就是一条测试路径，路径上迁移的序列就是操作序列。比如 init→empty→delete 是一条；init→empty→push→filled 是一条；init→empty→push→filled→push→full→push ✳→full 又是一条。请数一下这棵树上一共有多少条根到叶的路径——这个条数就是"完全展开"下的用例条数。

第六步，补数据与预期。光有迁移序列不叫测试用例，要给每一个迁移补上输入数据和预期结果：max 取值多少、栈里原有几个元素、push 进去什么值，以及每一步之后栈的状态与返回值是什么。

第七步，按覆盖准则剪枝。完全展开的路径条数随着状态数会迅速膨胀，教材在讲图覆盖准则时明确给了三档：节点覆盖、边覆盖、路径覆盖，而完全路径覆盖往往产生无限的路径、是无法满足的，一般选主路径覆盖。所以工程做法是：先按覆盖准则挑出一组路径，再逐条补数据。这七步，就是"模型驱动测试用例"的手工版本。

板书：状态图 → 树 → 用例 七步：① 定根（initial）② 逐条迁移展开（子节点＝到达状态）③ 遇"指回已出现状态"的边 → 当叶子停（剪环）④ 保留 ERROR 出口 ⑤ 根→叶 ＝ 一条测试路径 ⑥ 每条路径补输入数据＋预期结果 ⑦ 按覆盖准则（节点／边／主路径）剪枝｜示例路径：init→empty→delete；init→empty→push→filled；init→empty→pop ＝ ERROR

提问：这棵树里 filled 出现了三次，看起来像画重复了。为什么必须允许状态在树里重复出现？如果不允许，会出什么问题？

预设回答：

- 答"因为要表示不同的到达路径"：✅ 对，第一次 filled 是"压了一个"，第二次是"压了又弹"，第三次是"压满又弹回"，到达同一个状态的历史不同，后续可走的分支和风险也不同。

- 答"重复了，应该只画一次"：❌ 那就不叫树了。点评：树和图的分工就在这里——图允许环、表达"系统能怎么走"；树不允许环、表达"测试可以怎么走一条不重复的路"。

- 答"重复是为了把环剪开"：✅✅ 最准确。请他们注意：树里的每个叶子要么是终止状态，要么是 ERROR，要么是"回边停住"的位置。

易错点 / 考点：①把"转化"当成"复制"——以为树上每个状态只该出现一次；②不剪枝，试图展开所有路径（结果就是无限，教材讲得很清楚，完全路径覆盖无法满足）；③只写正常路径，丢掉 ERROR 叶子，而非法操作才是缺陷高发的区域；④路径写完不配预期结果——没有预期的不叫测试用例，只叫操作记录。考试常

见形式：给一张状态图，要求画出转化后的树并写出若干条测试用例，或者直接比较"节点覆盖、边覆盖、主路径覆盖各需要几条用例"。记忆口诀：**图看怎么走，树看怎么测；遇环就停，见 ERROR 必测。**

> **过渡：**手工画树、数状态、算路径，状态一多就顶不住了。工业界的做法是把模型直接喂给工具——下一页看微软研究员的利器 Spec Explorer。

【3.94】状态图工具：Spec Explorer（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是工具页，pace 快一点，但两个要点必须留下印象。教材里介绍得很简短：微软研究院用 C# 开发了一个测试建模工具叫 Spec Explorer，它还能与 Visual Studio 集成；课件配的正是它的主界面截图。

看截图左半边：这是一个 Visual Studio 的窗口，左边栏是规格文件 SimpleSlice.seexpl 和它的状态图视图，节点编号是 S0、S2、S6、S8、S10 这样的机器编号，边上写的是 CreateAccessControl、RequestAccess、Connect、Disconnect，以及 Calculator 那一组 NewNumber、Add、Subtract、Multiply、Divide、Clear、Equals 的操作。请注意状态栏的数字：据课件截图数据， States: 20/42、Steps: 26/48，后面还有 Requirements、Bounds、Errors 三项计数全是 0——斜杠前是当前已经探索或已生成的数量，斜杠后是这个探索的上限。

再看右半边：这是从同一份规格派生出来的测试套件视图 TestSuite.seexpl，据课件截图数据显示 States: 146/218、Steps: 165/237，也就是说，从一个几十个状态的规格自动生成了一百多个步骤的测试套件。

教材里还给了它的官方网站和团队博客地址，说明这是一个持续维护的研究型项目。顺便说一句，截图底栏里那一项 Errors: 0，报的是模型自身的不一致或约束冲突，不是测试发现的缺陷——工具先要求模型自洽，然后才谈生成用例，这个顺序值得记住。

这两张图放在一起，就把 MBT 的核心价值讲清楚了：**模型是源，用例是流**。模型改一处，测试套件重新生成一次，人不用手抄用例、不用手工维护用例与需求的对应关系。但请记住它的边界——工具只能忠实地展开你给它的模型。模型里漏了"空栈 pop 应该报错"这条迁移，工具生成一万条用例也变不出这条规则来。所以说，工具放大的是模型的正确性，也放大模型的错误。

> **板书：**Spec Explorer ＝ 微软研究院・C# 开发・可与 Visual Studio 集成｜左：规格 `SimpleSlice.seexpl` ＋ 状态图（S0→S2→S6→S8→S10，CreateAccessControl／RequestAccess／Connect／Disconnect）

> ｜底栏：States／Steps／Requirements／Bounds／Errors 计数（据课件截图）｜右：`TestSuite.seexpl` 自动生成的测试套件｜一句话：模型是源，用例是流；模型错，用例全错

> **提问：**如果建模的人漏画了一条业务规则，比如漏了"空栈不许弹栈"，Spec Explorer 会不会帮我们把这条用例补出来？

> **预设回答：**

- 答"会，工具很智能"：❌ 纠正：工具不会凭空创造模型里没有的行为。它只能在给定迁移上枚举序列。

- 答"不会，工具只展开你给的模型"：✅ 对。追问：那怎么防这种错？——靠模型评审，也就是第 2 章讲过的静态测试，人对模型的评审。

- 答"可能会报错提示"：⚠ 部分正确——工具能报出状态空间上界、约束冲突这类问题，但业务规则缺漏它无从知道。

> **易错点 / 考点：**①把 Spec Explorer 当成"测试执行工具"，它做的是建模与用例生成，执行还要接测试框架；②以为有了 MBT 工具就不需要人做需求判断；③记不住它属于哪一类方法——它属于"基于模型的测试（MBT）"，也与 3.7 节形式化方法一脉相承。考点形式多为选择、判断（"MBT 工具可以自动发现需求遗漏"——错）。记忆口诀：**模型是源，用例是流；工具不背需求的锅。**

> **过渡：**Spec Explorer 是研究背景的工具，我们再看一个开源的应用建模工具——ATS4_AppModel。它做的事情很有意思：把"用例"和"状态模型"放进同一个工程里管理。

【3.95】示例：ATS4_AppModel（1 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这是一个开源工具的示例，网址课件给的是 ats4appmodel.sourceforge.net，工程文件名据课件截图是 N78.xml。
>
> 看这张截图的三块内容。先看"System model"系统模型树：Contacts 下面挂着 GroupContacts、Edit contact，Gallery 下面挂着 Images、Video clips、Songs、Sound clips、Streaming links，还有 Song is playing 这样的状态项。另一棵是"Main Use Cases"主要用例树：Contacts 下面是 ContactDelete、ContactGroupAdd、ContactGroupDelete，Gallery 下面是 Delete song、Delete sound clip、Edit Streaming Link 等等，每一项前面还有一个像执行按钮一样的小图标。

中间是状态模型视图：圆角矩形是状态，有 IDLE、Menu、Media、Contacts 等，从 IDLE 出去一条 U2 进 Menu，从 Menu 出去 U3.1、U3.2、U3.3 分别到 Media 和 Contacts。注意它还有"分支/合并"的菱形符号，说明这个工具的模型表达能力比一张纯状态图更接近真实的软件结构。

下面是 Description 栏，用来给选中的节点写文字说明。

顺带说一句：左边用例列表里每一项前面都有一个小图标，说明每一项都是可以被调用的操作入口。把这些入口和状态模型挂在一起，"用户能做什么"与"系统会怎么变"就一一对应上了——这正是后面几页写场景时要用的思维方式。

这张图我最想让大家记住的是两点。第一，**模型、用例、说明三样东西在同一个工程里**，它们不会各说各话——用例挂在哪块功能上是明确的，改动可追溯。第二，请留意它用 U2、U3.1 这样的编号来标记迁移。在后面几页要讲的 ATM 事件流图里，我们会看到同一套记法：**U 打头的是用户动作，S 打头的是系统响应**，两套编号交替记录，一条场景就能写成"用户做了什么 → 系统返回什么"的序列，正好就是测试用例的输入与预期输出。

板书：ATS4_AppModel（开源，http://ats4appmodel.sourceforge.net/；示例工程 N78.xml）｜左：System model（Contacts／Gallery：Images・Video clips・Songs・Streaming links 等）＋ Main Use Cases（ContactDelete・Delete song・Delete sound clip…）｜中：状态模型（IDLE --U2--> Menu --U3.1／U3.2／U3.3--> Media／Contacts）｜下：Description 逐节点说明｜要点：模型＋用例＋说明同工程管理；U＝用户动作・S＝系统响应

提问：为什么要把"用例"和"状态模型"放进同一个工程，而不是各写一份文档?

预设回答：

- 答"方便一起改"：✅ 方向对，追问：放在一起改，能防住什么错？——能防住"用例描述的操作在模型里根本不存在"这类不一致。

- 答"因为工具要自动生成用例"：✅✅ 更到位。用例是模型的入口，模型是用例的展开，两者同源才能自动派生测试序列。

- 答"这只是界面设计，没什么讲究"：❌ 点评：这正是建模工具与画图工具的分水岭——画图工具产出图片，建模工具产出可执行的规格。

易错点 / 考点：①把建模工具当画图工具用（价值在于"模型是唯一事实来源"）；②建模粒度失控，把界面上每个控件都建成一个状态，模型立刻爆炸——粒度以"业务上可区分的状态"为准；③记不住两个工具名与定

位：Spec Explorer 属微软研究院的研究型工具，ATS4_AppModel 是开源应用建模工具；④考点：MBT／形式化工具的归属判断（属于基于模型的方法），以及"U 与 S 各表示什么"。

过渡：到这里，状态图这条线索收口了——从状态图，到树，到用例，到工具。接下来我们换一副眼镜看同一个系统：不从"系统有哪些状态"出发，而从"用户想干成什么事"出发。这就是课件专门补充进来的一节：基于场景的测试方法。

【3.96】补充：基于场景的测试方法（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是本片第二个重点，也是课件里明确标为"补充"的一节。为什么补充？因为它跟你们马上要做的课程设计最贴——需求只有一个用户故事，测试用例从哪儿来，这一页给了答案。

先记住这一页给出的等式：**测试用例 = 用户故事（行为）+ 场景 + 测试数据**。这是乘法关系，不是并列关系，缺任何一项用例都落不了地。

第一个乘数，用户故事，写的是**行为**。图上那张黄色便利贴上写的是它的标准句式：As who, I want what, so that why——作为谁，我想要什么，以便达成什么。这是敏捷开发里最常用的需求表达方式。第 2 章我们讲过四对测试方式，也讲过敏捷团队会把单元、集成、系统、验收压缩在几天内完成；但压缩的是时间，不是内容，该问的问题一个都不能少。基于场景的测试，就是把这些问题的答案直接拿来当测试的起点。

第二个乘数，**场景**。图上给的定义是：条件、前提、运行环境等。具体说就是：首次安装还是老用户、中午十二点的高峰还是晚上八点、预定成功后还是取消订单后、网络通还是断。这些"条件与前提"，决定了同一句用户故事会走出完全不同的路径。

第三个乘数，**数据**。让这个场景真正跑起来的具体输入：账号、商品编号、金额、库存数量。没有数据的场景只是一句愿望，跑不起来，也判不了对错。

为什么只有用户故事不够？因为它只说"要什么"，没说"在什么情况下要、拿什么数据要"。所以要把故事乘上场景、乘上数据，才能变成一条能执行的用例。

再把它的位置摆清楚：等价类划分和边界值分析从**输入域**出发，逻辑覆盖从**代码**出发，状态图从**系统内部状态**出发，而基于场景的测试从**用户的使用路径**出发。它是黑盒的、面向业务流的，天然能覆盖那种"单个功能都对、一串起来就出问题"的缺陷——这正是别的方法最容易漏掉的一类。

最后回应一下敏捷。敏捷的验收标准常常就写成场景（Given／When／Then 这类写法），基于场景的测试直接复用这套语言，不需要二次翻译，需求和用例天然同源。本章最后的思考题里有一问："为什么敏捷测试中适合使用基于场景的测试设计方法？"答案就在这里。

板书：测试用例 = 用户故事（行为）+ 场景 + 测试数据｜用户故事句式：As who（作为谁）· I want what（我要什么）· so that why（以便什么）｜场景 = 条件·前提·运行环境（首次安装／高峰时段／预定成功后／网络状态）｜数据 = 账号·编码·数量·金额｜出发点：用户使用路径（黑盒·业务流）｜对照：输入域→等价类／代码→逻辑覆盖／状态→状态图／用户→场景

提问：同一个用户故事"我要预订酒店"，一个刚装上 App 的入门级用户和一个用了三年的熟练用户，测试用例能一样吗？差在哪里？

预设回答：

- 答"一样，都是订酒店"：❌追问：新用户面对的是空账号、没有历史地址、可能还没授权定位；老用户有历史订单、有默认支付方式、有会员等级。路径、入口、风险点全都不同。

- 答"不一样，因为场景不同"：✅对。场景就是把这些差异显式写下来，逼着你想清楚"还有哪些用户、在什么条件下走这条故事"。

- 答"数据也不一样，而且预期结果也可能不一样"：✅✅最完整。场景的差异最终必须落到数据与预期结果上，否则这条用例还是不可执行。

易错点 / 考点：①只写故事不写场景，用例退化成"把功能点一遍看看"；②把场景写成"操作步骤清单"——场景关注条件与前提，步骤是场景确定之后才派生出来的；③以为基于场景的测试只能测正常流程，恰恰相反，它的价值在备选流（异常与失败路径）；④与等价类的关系要说清：场景法决定"走哪条业务路径"，等价类决定"这条路径上用哪组数据"，两者互补而不是互相替代。考点形式：给一个用户故事，要求补齐场景与数据；或判断题"基于场景的测试属于白盒方法"（错，属黑盒／基于需求的测试方法）。

【思政落点 22】：这一页要求我们先问"谁在用、在什么条件下用、想办成什么事"，而不是"代码里有哪些分支"。这就是"以人民为中心"在软件工程里的样子：把用户当成活生生的人，而不是一组输入数据。政务、医疗、金融系统的适老化与无障碍改造，靠的正是把老年人、残障人士当作真实场景主体去建模、去测试。设计用例时多站到用户那一侧看一眼，很多"想当然"的缺陷动手之前就会被发现。

过渡：概念讲完了，我们来看一张只有三列的表：用户、上下文（场景）、目标。看看别人是怎么把这三样东西摊在一张纸上、再连出场景来的。

【3.97】示例：基于场景的测试（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这是某酒店预订 App 的场景梳理表，只有三列：用户、上下文（场景）、目标，中间用虚线把相关的格子连起来。

第一列，用户，列了两种：入门级用户、熟练用户。

第二列，上下文，也就是场景，列了五种：首次安装使用；特定时间打开 App，括号里特意写了 12:00 和 20:00；预定成功后；取消订单后；退出 App 后。

第三列，目标，列了五种：随便看看；查找符合自己要求的酒店；预定酒店；入住酒店；离开酒店。

这张表怎么读？一条连线就是一个候选场景。比如"入门级用户 × 首次安装使用 × 随便看看"是一条；"熟练用户 × 取消订单后 × 预定酒店"，又是一条。三列不是三份互不相干的清单，而是一个组合空间——两乘五乘五，光是这三个清单全组合就是五十种，据课件数据列出的就是这些项。

但图上画出来的线是有限的，为什么？因为有些组合在业务上根本不成立或者风险极低。比如"首次安装使用"这个场景，和"离开酒店"这个目标连起来就没有什么意义——刚装上 App 怎么谈离开酒店。所以这张图真正的功夫在**筛选**：先把三列拉满，再按业务风险连线，连出来的线才是要测的场景。这就是场景法避免组合爆炸的办法——不穷举，而是有依据地挑。

还有两处细节值得停一下。第一处，第二列里写了"特定时间打开 App（12:00／20:00）"。这说明场景里包含时间这种运行环境条件：中午和晚上是下单高峰，可用库存、并发压力、网络质量都不一样。这条要求在等价类划分里几乎没法表达，但在场景里一句话就说清楚了。第二处，"预定成功后""取消订单后""退出 App 后"这三项，本质上都是**前置状态或前序动作**。入门级用户在"预定成功后"再操作，和一开始就操作，能踩到的坑完全不同。所以我要给大家一个判断：**看一个测试人员会不会做场景测试，就看他有没有把"前提"当回事。**

板书：三列场景表｜用户：入门级用户・熟练用户｜上下文（场景）：首次安装使用・特定时间打开 App（12:00／20:00）・预定成功后・取消订单后・退出 App 后｜目标：随便看看・查找符合要求的酒店・预定酒店・入住酒店・离开酒店｜读法：一条连线 = 一个候选场景｜做法：三列拉满 → 按业务风险连线（不穷举）｜注意：时间／环境也是场景；"…后"是前置状态

> **提问：**如果只允许你先测三条连线，你选哪三条？依据是什么？

> **预设回答：**

- 答"选预定酒店相关的"：✅ 方向对。追问：为什么？——涉及钱，高频且高损失，出问题的代价最大。

- 答"选入门级用户的首次安装使用"：✔️ 很有价值，新用户路径最长、依赖授权与引导，是最容易流失也最容易报错的一段。

- 答"五十种全测一遍才放心"：❌ 点评：一个迭代里没有这个时间，而且大量组合的风险几乎为零。场景法的方法论恰恰是"有依据地不测"——把有限的测试时间投到高频与高损失的路径上。

> **易错点 / 考点：**①把这张三列表直接当成测试用例清单——它还缺数据与预期结果；②只按"目标"列设计用例，忽略"上下文"会改变同一目标下的系统行为；③漏掉时间、网络、权限这类非功能条件；④误以为连线越多越好。考点形式：给一张三列表，要求指出哪两条不能连（业务上不成立）并说明理由；或要求为某条连线补出测试数据与预期结果。

> **过渡：**现在轮到你们动手了。下一页是一道当堂练习：给你一个用户故事，请你把场景和数据列出来。我先把做题的四步和时间安排讲清楚。

【3.98】练习：列出测试场景和数据（5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

> 这一页是当堂练习，任务写在图上：**针对某典型用例列出测试场景和测试数据**；课件配图是大家最熟的社交信息流界面，你们可以把它直接当靶子。这一页的讲者备注还加了一条要求：**针对刚才完成的用户故事，拿出 3 个不同类型的用户故事进行切分**。
>
> 时间这样安排：**前 2 分钟**每个人自己在纸上写，不许讨论；**中间 1 分钟**同桌互换，只评一句话——"基本流找全了吗？备选流漏了哪条？"；**最后 2 分钟**我们一起对答案。
>
> 做题按四步走，这四步也是我要求你们写在纸上的顺序。第一步，**写基本流**：把"一切顺利"的主路径按先后顺序写出来，一条不漏，这是主干。第二步，**枚举备选流**：主路径上任何一个环节可能出岔子的地方，都单列一条，包括分支、异常、失败回退，还有"用户中途改主意"这种非异常的岔路。第三步，**组装场景**：一条场景 = 基本流 + 一条或几条备选流。第四步，**补数据与预期结果**：给每条场景配上具体输入和可判定的预期。请记住一句话——没有数据和预期结果的场景，只是一句愿望，不是测试用例。

场景表就列三列：第一列写场景（要写清它走了哪条基本流、哪条备选流），第二列写输入数据，第三列写预期结果；如果还有空，就在备注里标明这条场景覆盖了哪条业务规则，这一列将来能让你的用例库被人看懂、被人复用。

我给大家一个可以照着套的样例，用的是"登录—检索商品—下单支付"这条业务。基本流是：登录成功 → 检索到商品 → 加入购物车 → 提交订单 → 支付成功 → 订单生成。备选流我举四条：账号或密码错误；商品库存不足；支付超时或余额不足；提交过程中网络中断。然后组装场景、补数据——具体的一条我写在板书右侧的表里。

最后说讲评要点。场景法最大的好处是贴近用户真实使用路径，能抓到"多个功能连起来才出问题"的缺陷，比如下单成功了但库存没扣减。它的局限也要说清楚：场景数量会随着备选流的组合迅速增长，所以必须按业务风险排序，优先覆盖高频和高损失的路径，其余的安排在后面几轮迭代里补。

> **板书：**四步：① 写基本流（顺利主路径）② 枚举备选流（分支・异常・失败回退・用户改主意）③ 组装场景（基本流＋备选流）④ 补数据与预期结果｜场景表三列：**场景（含所走基本流／备选流）**｜**输入数据**｜**预期结果**（＋备注：覆盖的业务规则）｜时间：2 分钟独立写 → 1 分钟同桌互评 → 2 分钟讲评｜好处：贴近真实路径、抓跨功能缺陷｜局限：场景组合膨胀，须按风险排序

提问：如果时间只够写三条场景，你的三条会怎么排？排序依据是什么？

预设回答：

- 答"基本流一条、最可能出的错一条、后果最严重的一条"：✅✅ 这是非常成熟的思路——主干保底线，高频抓概率，高损失兜风险。

- 答"三条都写基本流的不同数据"：❌ 点评：那不是三条场景，是同一条场景的三组数据，等价类划分就能解决，用不着场景法。

- 答"随便挑三条备选流"：⚠ 能写出来但缺少依据。追问：你怎么知道这三条是最该测的？——要能说出风险来源（投诉量、资金、合规）。

预设回答（讲评参考答案，按 `_answers.json` 口径）：

场景（所走基本流／备选流）	输入数据	预期结果
基本流：登录成功 → 检索到商品 → 加入购物车 → 提交订单 → 支付成功	账号 zhangsan／密码 Abc123；商品编号 SKU-1001；数量 1；支付方式余	订单生成成功，库存扣减 1，支付流水一条，页面跳转订单

→ 订单生成	额充足	详情
备选流一：账号或密码错误（登录失败并提示）	账号 zhangsan／密码错误一次	提示账号或密码错误，停留在登录页，不生成会话，不记入订单
备选流二：商品库存不足（禁止下单）	登录 zhangsan／Abc123；商品编号 SKU-1001；库存 0；购买数量 1	提示库存不足且**不生成订单**，购物车状态可恢复
备选流三：支付超时或余额不足（订单转待支付）	已提交订单；支付超时／余额不足	订单状态转为"待支付"，库存不被永久占用，可在时限内重试支付
备选流四：提交过程中网络中断（可重试且不重复扣款）	提交订单时断网后恢复	提示网络异常，可重试；重试后**只有一笔扣款、一个订单**

易错点 / 考点：①把基本流写成"测试步骤"而不是"用户与系统的交互序列"——步骤是执行动作，基本流是业务过程；②备选流只写"失败了"，不写失败之后系统怎么回退、怎么恢复（能不能重试、会不会重复扣款、订单停在什么状态），这类"失败后的状态"才是缺陷重灾区；③预期结果写成"正常"两个字，不可判定，等于没写；④把场景表和用例编号混在一起，忘了标注覆盖的业务规则，导致用例库不可复用。考点形式：大题给一段业务描述，要求写基本流、备选流，并设计三条场景及其数据与预期结果——最常见的失分点就是备选流只写了一条，或者预期结果不可判定。

【思政落点 23】：练习里最容易走偏的，是为了让作业"看起来完整"而编造预期结果，或者把没测到的分支含糊地写成"正常"。诚实是测试人员的职业底线：覆盖到了就写覆盖到了，没覆盖就写明未覆盖；结果是"未验证"就写未验证，不臆造、不含糊。缺陷报告也一样，如实记录复现步骤和当时的假设，不夸大、不隐瞒，团队才能基于真实信息做判断。软件质量的第一道防线，是人肯说真话。

过渡：场景法从"用户目标"出发很好用，但业务规则一多，目标之间就会互相干扰。这时我们换一种画法：把主事件流画成一条主干，把每个判断条件画成从主干分出去的支流。这就是下一页的事件流图。

【3.99】事件流图（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

事件流图，英文是 event flow diagram，是场景法最常用的图形化表达。结构很简单：中间一条主干，代表业务的主事件流；主干上每个判断条件引出一条支流，支流要么绕回主干，要么走到一个终止点结束。

看课件这张图。这一页开头那句话就是业务目标，也是标准的用户故事写法：**作为承兑人，根据委托收款请求，签发承兑**——作为谁、做什么，一目了然。这就是主干要完成的事。

主干往下走，会遇到一组判断条件。图上把这六个问题一次性列了出来：无条件支付？不得转让？已转让了吗？收款人清楚吗？收款人是否破产？金额超过上限？请大家注意，这六个带问号的句子，就是这张图里全部的**业务规则清单**——每个问号都是一个分支点，答案决定业务往哪走。

再看从主干上分出去的分支：金额"超过上限"，先走到"有条件支付"，如果"不符合条件"，就终止；票据"已转让"，若"不得转让"，终止；"收款人破产"，终止。到这儿，一条主干、六个判断、三个终止点，这张图就画完了。

读图三步法。第一步，**找主干**：哪条线是"一切顺利、业务办成"的那条。第二步，**圈判断**：主干上每个判断条件都圈出来，挨个问"它成立走哪、不成立走哪"，注意有的分支不止两条出路。第三步，**标终止**：每个终止点都要写一句业务结论，比如"不符合条件不予承兑"，不能只写"结束"。

为什么票据业务非要用这种图？课件的讲者备注说得很明白：票据的自助式受理要完成票据的**存入、存储、记账**，功能上还要考虑**身份认证、防伪识别**。这类业务规则密密麻麻又互相制约——能不能转让、要不要无条件支付、金额有没有上限，这些条件纠缠在一起。用一条主干加若干分支来梳理，比写一大段文字规则清楚得多，评审会上更容易被人指着某条支流问"这条你测了吗"。

最后和状态图对比一下：状态图问"系统现在什么状态、这个事件把它带到什么状态"，是**系统视角**；事件流图问"用户要把这件事办成，路上有哪些岔路、哪些岔路直接结束"，是**业务视角**。事件流图还有个直接的好处——**每一条从主干起点走到某个终止点的完整路径，就是一条测试场景**。

> **板书：**事件流图 = 主干（主事件流）+ 分支（判断条件）+ 终止点（业务结论）｜本例主干：作为承兑人，根据委托收款请求，签发承兑｜判断条件：无条件支付？·不得转让？·已转让？·收款人清楚？·收款人破产？·金额超过上限？｜分支：超过上限 → 有条件支付 → 不符合条件（终止）｜已转让 → 不得转让（终止）｜收款人破产（终止）｜读图三步：① 找主干 ② 圈判断（问成立／不成立各走哪）③ 标终止并写业务结论

> **提问：**这张图里"金额超过上限"并没有直接终止，而是先走到"有条件支付"，再判断"是否符合条件"才终止。为什么要多走这一层，而不直接终止？

预设回答：

- 答"因为超过上限不一定要拒绝，可以转成有条件支付"：✅ 对。业务上超过限额往往还有补救路径，比如需要额外审批或担保，所以必须再判一次。

- 答"图上画错了"：❌ 点评：这不是笔误，而是业务规则的真实层次——"不满足某个条件"和"业务上必须拒绝"是两件事，中间还有补救空间。测试时这一层必须单独测。

- 答"有条件支付成立和不成立要各测一条"：✅✅ 追加：这就是两条不同的场景，各自要有数据与预期结果。

易错点 / 考点：①把事件流图和程序流程图混为一谈——流程图描述"程序按什么顺序做"，事件流图描述"业务上有哪些岔路与结论"，它不关心处理顺序，只关心条件与结果；②每个判断只画"是／否"两路，忘了现实里还有第三种情况（比如"有条件支付"）；③终止点不写业务结论，只写"结束"，评审时看不出业务后果，也无法判断预期结果；④把主干画成只有一条直线，忘了主干上也有"环节"，后面 ATM 那个例子里主干会被拆成 U1、S1.1、U2、S2.1 这样一连串。考点形式：给一段业务规则文字，要求画出事件流图或列出判断条件与终止点清单。

过渡：事件流图长什么样知道了，接下来两页用一个人人都用过的系统，把"场景"真正落到编号和数据上——ATM 取款。

【3.100】基于场景的测试方法 – 示例（2 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

这一页是 ATM 取款的场景清单，一个正常场景，加十条可选场景——注意课件上写的是"可选场景"，其实就是我们说的备选场景。

正常场景只有一句话：**从账户中成功取款**。别小看这一句，它就是主干，是后面所有分支都要挂上去的那条线。

备选场景的标题是"无法取款，因为"，下面列了十条原因：① 用户的银行卡被拒绝，因为银行卡无法被 ATM 机识别；② 用户三次以上都没能正确地输入密码；③ 用户三次或三次以上都没能正确地输入密码，ATM 机发生吞卡；④ 用户选择了存款或转账功能，而不是取款功能；⑤ 用户在插入的卡上选择了一个非正确的账户；⑥ 用户的取款数额不正确；⑦ ATM 机的现金不足；⑧ 用户输入了一个不存在的数额；⑨ 用户输入的数额超出了每日的取款限额；⑩ 用户银行卡账户的余额不足。

这十条不是拍脑袋想出来的，它们是顺着正常场景的主干，一个环节一个环节问"这里会不会出岔子"问出来的：插卡环节 → 卡片识别失败；密码环节 → 输错、连续错三次直到吞卡；功能选择环节 → 用户改选了存款或转账；账户选择环节 → 选错了账户；金额环节 → 数额不正确、不存在、超限、余额不足；出钞环节 → 机器里没钱。你们把这个顺序记下来，就等于掌握了一套"生成备选场景"的检查表。

这里我要专门点一处：②和③看起来像重复，其实必须拆成两条。②说的是"密码不正确"，系统大概率是提示后让你重试；③说的是"错误次数达到三次并把卡吞掉"——预期结果完全不同，一条是"退卡重来"，一条是"卡片被机器收走"。**拆不拆的判据只有一条：预期结果是否相同。**

还有第④条也值得琢磨："用户选择了存款或转账功能"严格说不是失败，而是用户改主意、走到了另一条业务流。它提醒我们：备选场景不全是异常，还包括"中途换了目标"这类正常但分岔的行为。

最后一句——每一条场景后面都还缺着东西：具体的数据和可判定的预期结果。下一页和再下一页，我们把它们落到流程图上。

板书：正常场景：从账户中成功取款（＝主干）｜备选场景十条，按主干环节归类：插卡 → ① 卡无法识别｜密码 → ② 密码错误 · ③ 错三次吞卡｜功能 → ④ 改选存款／转账（走另一条业务流）｜账户 → ⑤ 选错账户｜金额 → ⑥ 数额不正确 · ⑧ 数额不存在 · ⑨ 超日限额 · ⑩ 余额不足｜出钞 → ⑦ 现金不足｜拆分判据：**预期结果不同就拆**

提问：第②条"三次以上都没能正确输入密码"和第③条"三次或三次以上都没能正确输入密码，ATM 机发生吞卡"，到底要不要拆成两条场景？

预设回答：

- 答"重复了，合并成一条"：❌ 追问：合并以后，"提示你再试一次然后把卡退给你"和"直接把卡吞掉"这两种结果，你的预期结果栏只能写一个，另一个就漏测了。

- 答"要拆，因为预期结果不同"：✅ 对，这就是唯一正确的判据。

- 答"还应该再拆出密码错一次、错两次"：✔️ 可以讨论。追问：系统对第 1 次和第 2 次的行为一样吗？如果完全一样，它们属于同一个等价类，测一次就够——这说明场景法与等价类划分是配合使用的。

易错点 / 考点：①把备选场景当成"错误提示文案清单"，忘了预期结果里还有系统状态与账务变化（比如"不得生成订单""不得重复扣款"）；②漏掉"用户改主意"这类非异常备选流；③场景一多就放弃排序、想全测，实务上要按"发生概率 × 损失大小"排序；④同一环节的多档严重程度只写一条——这是这类题最集中的失分点。考点形式：给一段业务描述，要求补全备选场景清单，并挑一条写出数据与预期结果。

> **过渡：**清单有了，可它还悬在空中——插卡、密码、账户、金额这些环节之间的先后顺序和分支关系，还没有画出来。看下一页的 ATM 事件流图，我们把十条场景一条条挂上去。

【3.101】基于场景的测试方法 – 示例（续）（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这张图就是把上一页那十条备选场景画成了一张事件流图。看图之前先记住它的编号规则，这是整页的钥匙：**U 打头的是用户动作，S 打头的是系统响应**。

先走主干。U1 用户插入卡 → S1.1 系统接受卡 → U2 输入密码 → S2.1 密码正确 → U3.1 按取款键 → U4 选择账户 → S3.1 正确的账户 → U5 输入取款金额 → S4.1 金额适当 → S5 分发现金 → S6 计算余额 → S7 打印凭条 → S8 退卡 → U6 用户取卡，结束。从插卡到取到钱，这一串就是"成功取款"那条主干。

再看分支，一共四处。**插卡处**：S1.2 系统拒绝卡，然后 S8 退卡、U6 用户取卡——卡不认，退给你。**密码处**：S2.2 密码不正确但少于 3 次，箭头绕回 U2，让你重输；S2.3 密码不正确等于 3 次，走 S9 吞卡，然后 U6 用户取卡。**功能处**：U3.2 按存款／转账键，走 S10 调用存款／转账模块，结束——这就是上一页第④条"用户改主意"应有的样子。**账户处**：S3.2 非正确的账户，箭头绕回 U4 重选。**金额处**：S4.2 金额不正确、S4.3 超出每日取款上限、S4.4 账户余额不足、S4.5 ATM 机余额不足——四条异常响应，都从 U5 这里分出去。

这张图有三个读图要点，请大家记下来。第一，**每个 S 都是断言点**。用户做了 U，测试真正要判的是系统的响应是不是等于预期的那个 S——只写用户做了什么，用例是判不了通过失败的。第二，**绕回上游的箭头意味着重试**。S2.2 回到 U2、S3.2 回到 U4、S4.2 回到 U5，这些回边让场景有了"重试"的味道，而重试最经典的缺陷就是次数记不准、计数不重置——密码连错三次这条逻辑，是这一页最值钱的测试点。第三，**每个终点都要写成业务结论**：退卡、吞卡、改走存款转账模块、取现成功，各有各的业务含义。

有了这张图，"场景"就不再是一句文字清单，而是一条能逐号追溯的路径——一条路径就是一串 U／S 编号序列。下一页我们看课件怎么用这些编号，把每一条场景"记账"下来。

> **板书：**编号规则：**U ＝ 用户动作｜S ＝ 系统响应**｜主干：
> U1→S1.1→U2→S2.1→U3.1→U4→S3.1→U5→S4.1→S5→S6→S7→S8→U6｜分支：S1.2 拒绝卡（→S8→U6）｜S2.2 密码错＜3 次（回 U2）・S2.3 密码错＝3 次（→S9 吞卡）｜U3.2 存款／转账（→S10，结束）｜S3.2 非正确账户（回 U4）｜S4.2 金额不正确・S4.3 超日限・S4.4 余额不足・S4.5

ATM 余额不足｜三要点：S 是断言点・回边＝重试要测计数・终点写业务结论

提问：为什么这张图要把每个环节都写成"一对 U 和 S"，而不是只记用户动作?

预设回答：

- 答"因为要断言系统响应"：✅ 对，用户动作只是输入，测试判的是系统返回对不对。

- 答"这样画比较整齐"：❌ 追问：只写用户动作，你拿什么判定这条场景通过还是失败?

- 答"U 是输入、S 是预期输出，正好对应测试用例的输入栏和预期结果栏"：✅✅ 最到位——事件流图天然就是用例的骨架：路径上每个 U 是输入数据的位置，每个 S 是预期结果的断言点。

易错点 / 考点：① U／S 编号混用，把系统响应写成用户动作（比如把"退卡"记为 U）；②忽略回边上的重试计数逻辑——密码重试三次是最典型的缺陷点，要专门验证第 3 次之后的行为与计数重置；③把 S5"分发"这种复合环节当成一个黑盒，忘了它下面还有更细的分支（下一页的覆盖串里会出现 S5.1 到 S5.6 这样的编号，据课件数据）；④断言点只写"页面正常"。考点形式：给一张事件流图，要求写出一条场景的完整 U／S 覆盖序列，或指出某条分支的预期结果。

过渡：图有了，接下来看课件怎么用 U／S 序列给每一条场景"记账"——这一页的清单，就是把场景和覆盖编号对应起来的台账。

【3.102】基于场景的测试方法 – 示例（续）（2 分钟）

☐ 已讲 ☐ 需回看 课堂记录：_____

这一页是一张覆盖清单：每一条场景后面都用括号注明了它走过哪些 U／S 编号。这就是"场景—覆盖"的可追溯记录，也是你们写课程设计报告时该有的样子。

我把主要的几条念一遍。① **成功地从账户中提取现金**，覆盖 U1、S1.1、U2、S2.1、U3.1、U4、S4.1、U5、S5.1、S6、S7、S8、S9、U6——这就是主干。② **用户银行卡无法被 ATM 机识别**，覆盖 U1、S1.2、S9。③ **用户输入密码不正确少于 3 次**，覆盖 U1、S1.1、U2、S2.2。④ **三次输入密码不正确**，覆盖 U1、S1.1、U2、S2.2、U2、S2.3、S3。⑤ **ATM 机中现金不足**，覆盖到 U5、S5.3。⑥ **用户账户余额不足**，覆盖到 U5、S5.6。⑦ **用户输入的取款金额不正确**，覆盖到 U5、S5.2。⑧ **用户输入的密码不正确超过 3 次**，覆盖到 S2.3、S3。⑨ **用户输入一个不存在的数额**，覆盖到 U5、S5.4。⑩ **用户输入的金额超出了日取款限制**，覆盖到 U5、S5.5。最后还有一条省略号后面的：**用**

户选择存款或转账，覆盖 U1、S1.1、U2、S2.1、U3.2、S10。

这张清单有三个用法，请记牢。第一，**查漏**：把清单里出现过的编号在图上一个个点亮，哪些 S 一次都没被点亮，那就是测试盲区。第二，**反查**：一条用例失败了，从它的覆盖串可以立刻回溯到它命中或违反了哪条业务规则，写缺陷报告时能说清楚，而不是只写"取款失败"。第三，**去重**：④和⑧都在讲密码错三次，你要能说出一条比另一条多了什么、或者干脆合并——重复用例是测试集里最隐形的浪费。

还有一件事值得说：注意①号场景的覆盖串末尾出现了 S9、⑧号里出现了 S3、②号里出现了 S9，这些编号在图上都属于"吞卡"那条支线。当覆盖串和你在图上看到的路径对不上时，不要跳过——**回图核对**。核对的结果无非两种：要么发现是抄写时串了行，要么发现图外还有一条你没画出来的规则。两种结果都有价值，但前提是你真的核了，并且把核对的结果和你的假设**如实写下来**。这就是我一直强调的"覆盖要能追溯"。

板书：场景 → 覆盖序列（可追溯台账）｜① 成功取现：U1,S1.1,U2,S2.1,U3.1,U4,S4.1,U5,S5.1,S6,S7,S8,S9,U6｜② 卡无法识别：U1,S1.2,S9｜③ 密码错＜3 次：U1,S1.1,U2,S2.2｜④ 错 3 次：U1,S1.1,U2,S2.2,U2,S2.3,S3｜⑤ 现金不足：…U5,S5.3｜⑥ 余额不足：…U5,S5.6｜⑦ 金额不正确：…U5,S5.2｜⑧ 密码错＞3 次：…S2.3,S3｜⑨ 数额不存在：…U5,S5.4｜⑩ 超日限：…U5,S5.5｜⑪ 存款／转账：U1,S1.1,U2,S2.1,U3.2,S10｜三个用法：点灯查漏 → 反查规则 → 找出重复｜纪律：覆盖串与图对不上就回图核对，并把结论与假设如实写下来

提问：拿到这样一张覆盖清单，你怎么在图上找出"测试盲区"？

预设回答：

- 答"看哪些编号一次都没出现过"：✅ 对，就是"点灯"——把清单里的编号在图上点亮，没亮的 S 就是没测到的地方。

- 答"数一数有几条用例，够不够多"：❌ 点评：条数不等于覆盖。十一条场景照样可能留下十个盲点，覆盖率要看编号覆盖，不是用例数量。

- 答"把它当成回归清单"：✅✅ 很有价值。下个版本改了哪条业务规则，就重跑覆盖这条规则的那几条场景，这就是基于场景的测试在回归阶段的用法。

易错点 / 考点：①把覆盖串当成"用例的全部内容"——它只是路径记录，还缺具体数据与预期结果；②看到覆盖串里 U2、U2 重复出现就以为写错了，其实那是"重试"场景的正常记法（同一动作走两次）；③不核对覆盖串与图是否对得上，直接把课件上的编号抄进报告；④把覆盖串的编号写成中文描述（"插卡成功"），失去可追溯性。考点形式：给出覆盖清单与事件流图，问哪条业务规则没有被覆盖、或哪两条场景重复。

> **过渡：**基于场景的这条线索到这里收口。接下来我们把镜头拉回代码层——最后两页用一个人寿保险投保的案例，先把语句覆盖和判定覆盖做完，再当堂完成条件覆盖设计。

【3.103】案例：语句覆盖与判定覆盖（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页是实验教程里的经典案例：投保业务的控制流图配上两组用例表——一组是语句覆盖的、一组是判定覆盖的，我们对照着看两者的差别。我们先把图读明白，再读表。

图的读法：开始之后遇到判定 A，条件是"16 岁＜年龄＜59 岁的男性，或 16 岁＜年龄＜54 岁的女性"。这个判定有两条出路：成立，也就是 Y，往下走判定 B；不成立，走 D，直接到结束——也就是不予受理。判定 B 的条件是"被保险人为健康体，或有医疗保险"：成立走 C，可投保本产品 A、B1、B2 一份；不成立走 E，可投保本产品 C 款 1 份。C 和 E 最后都汇合到结束。

所以这张图一共有三条基本路径：A→B→C，我们叫它路径 ABC；A→B→E，叫路径 ABE；A 不成立走 D，叫路径 AD。

现在看语句覆盖表。CASE1：年龄 20、男性、健康体、有医疗保险，走路径 ABC。CASE2：年龄 20、男性、非健康体且没有基本医疗，走路径 ABE。两条用例把所有语句——也就是 A、B、C、E 这几个节点——都执行到了，语句覆盖满足。请注意这两条用例的共同点：判定 A 的结果都是"真"。

再看判定覆盖表。判定覆盖的要求是：判定 A 和判定 B 各自都要取到"真"和"假"。CASE3：年龄 20、男性、健康体、有医疗保险，判定 A＝真、判定 B＝真；CASE4：年龄 15、男性，判定 A＝假；CASE5：年龄 20、男性、健康体、没有医疗保险，判定 A＝真、判定 B＝假。三条用例合起来，A 的真假有了，B 的真假也有了，判定覆盖满足。

这里有一个必须点破的直觉，也是这一页最值钱的地方：**语句覆盖用了 2 条用例，判定覆盖用了 3 条。**判定覆盖比语句覆盖强，但它并不必然用更少的用例！说出"覆盖能力更强"和"用例条数更少"，是两件事。另外请注意，CASE1 和 CASE3 描述的业务场景几乎一样——同一组数据可以同时为两种覆盖目标服务，实务上用例只写一次，在覆盖率台账里标注它服务哪些准则就行，不要抄两遍。

最后再点一个细节：CASE4 年龄 15，判定 A 为假，流程根本走不到 B，所以判定 B 的真假必须由 CASE3 和 CASE5 来承担。这就是"判定覆盖要求每个判定的两个出口都走到"的具体含义，也是后面条件覆盖那一页的

伏笔。

> **板书：**判定 A ＝ 16＜年龄＜59 的男性 ∨ 16＜年龄＜54 的女性｜判定 B ＝ 健康体 ∨ 有医疗保险｜三条路径：A→B→C（ABC）・A→B→E（ABE）・A→D（AD）｜语句覆盖：CASE1 ABC ／ CASE2 ABE（A 均为真）｜判定覆盖：CASE3 A 真 B 真 ／ CASE4 A 假 ／ CASE5 A 真 B 假｜要点：判定覆盖 ≥ 语句覆盖，**但用例条数不一定更少**；同一组数据可服务多种准则

提问：CASE3 和 CASE1 描述的业务场景几乎一样，为什么判定覆盖表里还要保留 CASE3？

预设回答：

- 答"重复了，可以删"：❌ 追问：判定覆盖要的是"A 真、B 真"这组判定取值，删掉 CASE3，B 取"真"由谁来保证？

- 答"因为它承载了判定取值的信息"：✅ 对。表里最后一列"判定 A＝真、判定 B＝真"才是判定覆盖的判据，用例描述只是让人看懂。

- 答"可以合并成一条，同时标注两种覆盖"：✅✅ 实务上就是这么做的——用例写一次，台账里标注它服务哪些准则。

> **易错点 / 考点：**①把"测试路径"和"测试用例"混为一谈——路径是判定的走向，用例是数据加预期；②以为判定覆盖一定比语句覆盖用的用例少；③漏掉 A 为假这条路径（年龄不符直接不予受理），它不产生 C、E 语句，但对判定覆盖是必需的；④边界值提醒：判定 A 的两个区间都是开区间，16 岁、59 岁、54 岁都取不到，用边界值分析要为 15／16／17、53／54／55、58／59／60 这几组年龄分别设计数据（据课件给出的区间）。考点形式：给控制流图和用例表，判断满足的是哪一级覆盖、还差哪条判定取值。

> **过渡：**语句覆盖和判定覆盖做完了，最后我们把这张表再往下推一格——条件覆盖。下一页右边那张表是空白的，我们当堂一起把它填完。

【3.104】完成条件覆盖设计（4 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这是本片最后一个练习页，也是课件留给你们的动手题。还是那张投保控制流图，任务是把"条件覆盖"表填完：表头三列——测试用例名称、测试用例描述、测试条件，CASE1 到 CASE3 三行留给你们填。我们一起把它填上。

先回忆条件覆盖的定义：它不看判定整体，只看判定里面的**每一个原子条件**——每个原子条件都要取到"真"和"假"至少一次。

那么这张图里有几个原子条件？判定 A 是两个条件"或"起来的：c1 是"16 岁＜年龄＜59 岁的男性"，c2 是"16 岁＜年龄＜54 岁的女性"。判定 B 也是两个条件"或"起来：c3 是"被保险人为健康体"，c4 是"有医疗保险"。一共四个原子条件，条件覆盖就要求 c1、c2、c3、c4 各取真、各取假，一共是八个取值要求。

我们开始做。CASE1：年龄 20、男性、健康体、有医疗保险。c1 真、c2 假、c3 真、c4 真；判定 A 因为 c1 为真而整体为真，判定 B 为真，走路径 ABC。

CASE2：年龄 20、女性、非健康体、没有医疗保险。c1 假、c2 真、c3 假、c4 假；判定 A 因为 c2 为真而整体为真，判定 B 为假，走路径 ABE。

到这里，八个取值全齐了：c1 有真（CASE1）有假（CASE2）；c2 有假（CASE1）有真（CASE2）；c3 有真（CASE1）有假（CASE2）；c4 有真（CASE1）有假（CASE2）。**条件覆盖已经满足了。**

但是——请大家盯着判定 A 看：CASE1 里 A 是真，CASE2 里 A 还是真。也就是说，条件覆盖满足了，判定 A 却从头到尾没有取到过"假"！这就是条件覆盖最大的坑：它把注意力打散到每一个原子条件上，反而可能放过判定整体的出口。所以我们要加第三行。

CASE3：年龄 15、男性。c1 假、c2 假，判定 A 整体为假，走路径 AD，到此结束——B 根本不会被求值。加上这一条，判定 A 的真假都出现了，A 的假出口也被覆盖。

请大家把这个结论记牢：**条件覆盖比判定覆盖看得更细，但它不保证判定覆盖。**想要两者兼得，就要用判定-条件覆盖或者条件组合覆盖——那是更上一级的准则。这也解释了为什么航空航天这类安全关键系统最后选 MC/DC：它既要求每个条件取真取假，又要求每个条件能够**独立影响**判定结果，把两种准则的长处合在一起。

这张表请你们带回去，收进课程设计的用例库。它不只是一次课堂练习的答案，更是你们报告里"为什么这么设计覆盖用例"的直接证据。

板书：四个原子条件：c1 ＝ 16＜年龄＜59 的男性｜c2 ＝ 16＜年龄＜54 的女性｜c3 ＝ 被保险人为健康体｜c4 ＝ 有医疗保险｜CASE1：20 岁・男・健康体・有医保 → c1 真 c2 假 c3 真 c4 真，A 真 B 真 → ABC｜CASE2：20 岁・女・非健康体・无医保 → c1 假 c2 真 c3 假 c4 假，A 真 B 假 → ABE｜CASE3：15 岁・男 → c1 假 c2 假，A 假 → AD｜结论：**条件覆盖 ≠ 判定覆盖**（本例中 A 只取真，须补 CASE3）

提问：CASE1 加 CASE2 已经让四个原子条件都取到了真假，条件覆盖已经满足了。为什么还非要加上

CASE3?

预设回答：

- 答"因为要覆盖 A 为假的情况"：✅ 对，A 的假出口在前两条里一次都没走。

- 答"多余，条件覆盖已经满足了"：❌ 这正是本页要破的直觉——满足条件覆盖不等于覆盖了判定的所有出口。

- 答"为了覆盖 D 这条路径，也就是拒绝投保这条业务"：✅✅ 追加一句：从业务上讲，年龄不符直接不予受理，是最需要确认的规则之一——覆盖率指标和业务风险在这里指向了同一个答案。

易错点 / 考点：①把条件覆盖理解成"每个判定取真取假"——那是判定覆盖；②把复合条件当成一个条件（把 c1∨c2 整体当成一个 c，就又退回判定覆盖了）；③误以为条件覆盖一定包含判定覆盖；④条件组合覆盖要求每个判定内原子条件的所有组合，用例数上升很快，实务上要配合约束与风险筛选；⑤三个准则的记忆口诀：**语句覆盖看语句，判定覆盖看出口，条件覆盖看原子条件**。考点形式为高频题型：给出判定表达式与用例表，判断满足的是哪一级覆盖；或反过来，给定原子条件，要求设计满足条件覆盖的用例表。

【思政落点 24】：这张条件覆盖表不是交完就算的一次性作业。把判定、原子条件、用例数据、预期结果整理成这样的表格，它就变成了团队资产：同事接手能看懂，下个版本改动了哪条业务规则，直接重跑对应行即可。反过来，只写在自己笔记本上的用例，人一走就散了。把测试用例写成团队可复用的资产，背后是协作与共享的工程文化，也是这门课要求你们建立用例库的原因。

过渡：到这里，第 3 章的方法部分就全部走完了——从等价类划分、边界值分析，到判定表与因果图，到逻辑覆盖与基本路径，到基于模型的测试和形式化方法，最后补上基于场景的测试。下一页开始是本章小结和思考题，其中一问正是"为什么敏捷测试中适合使用基于场景的测试设计方法"，请大家带着 3.96 那一页的答案去回答它。

【3.105】本章小结（3 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

好，第 3 章的方法讲到这里，我们花三分钟把它收成一张地图，因为这一章最容易出现的状况就是"方法学了一堆，换个题就不知道该用哪个"。

先看这张图的三条主线。第一条是**方法论层面**的三类方法：**黑盒方法**，也就是基于需求的测试方法，从规格说明出发；**结构化方法**，包括逻辑覆盖和基本路径覆盖，从代码结构出发；还有**基于场景的方法**，从用户的使用流程出发。第二条是**覆盖的两条通道**：**数据流覆盖**关注数据流程图、代码中变量的定义与引用——一个

变量在哪里被定义、在哪里被使用、有没有"定义了从没用过"或者"用了还没定义"；**控制流覆盖**关注业务流程的基本路径覆盖和代码中的逻辑覆盖。第三条是**层次**：代码层次、功能与非功能层次、业务层次——回答的是"测哪一层"。

那么本章讲了十几套方法，真正常用的是哪几个？请记住这四个：**等价类划分、边界值分析、决策表（判定表）、Pairwise**。它们性价比最高，绝大多数项目里都跑不掉。**最常用的覆盖标准**是三个：**语句覆盖、分支覆盖和 MC/DC 覆盖**。

最后给一个可操作的"方法选择三问"。第一问：**需求形态是什么**——需求完整清晰，走黑盒方法；需求模糊、时间紧，先用错误猜测和探索式测试摸风险。第二问：**风险等级有多高**——安全关键、钱和命相关，上 MC/DC 甚至形式化；普通业务模块，场景法加自动化回归就够。第三问：**能拿到什么模型**——有状态迁移图就用状态测试，有判定逻辑就用判定表或因果图，什么模型都没有就只能靠经验和场景。三问的答案一交叉，方法基本就定下来了。

回扣一下本章的教学目标：**知识上**，你现在应该能按依据说出方法类别，能说明等价类与边界值的取点规则，能讲清覆盖强度的递进关系与圈复杂度的算法；**能力上**，你应该能对给定规格划分等价类、列边界值用例、为复合条件设计满足 MC/DC 的用例、算圈复杂度导出独立路径集，并且能说明你为什么选这套方法。这两条就是本章的验收标准。

板书：方法地图：黑盒（需求）｜结构化（逻辑覆盖・基本路径）｜场景（用户流程）｜补充：数据流覆盖（定义-引用）・控制流覆盖｜最常用方法：等价类 ・ 边界值 ・ 判定表 ・ Pairwise｜常用覆盖标准：语句 ・ 分支 ・ MC/DC｜选择三问：需求形态？风险等级？模型可得？

提问：现在给你一个"手机银行转账"模块——需求文档完整、有状态迁移图、上线风险高、团队只有你和另一位同学、两小时内要交出第一批用例。你选哪两种方法组合？为什么？

预设回答：

- 答"等价类＋边界值"：✅ 起点正确，输入域方法性价比最高。但要追问：转账最容易出事的到底是输入还是状态？——是状态（重复提交、超时未回滚、并发扣款），光靠输入域方法抓不到，所以还得补状态或场景。

- 答"基于场景＋状态迁移"：✅✅ 更贴合。业务流加状态机一起覆盖，正好能抓住"重复扣款""超时未回滚"这类高危缺陷，与高风险、需求完整的条件匹配。

- 答"先跑一遍基本路径覆盖（白盒）"：❌ 典型的错配。白盒覆盖回答的是"代码被测到了吗"，而此刻你连第一批用例都要交，且未必有源码、插桩与工具链。点评：方法选择的第一步不是"我会什么"，而是"这一阶段最

怕什么"。

> **易错点 / 考点：**最高频的混淆是把**覆盖层次**和**覆盖标准**当成一件事。层次回答"测哪一层"——代码、功能/非功能、业务；标准回答"测到什么程度"——语句、分支、条件、判定-条件、条件组合、MC/DC、路径。考试常见问法"下列属于最常用覆盖标准的是"，答案落在**语句覆盖、分支覆盖、MC/DC 覆盖**。再记一句口诀：**层次看对象，标准看强度；黑盒对需求，白盒对代码，场景对用户，模型对状态。**

> **【思政落点 25】**方法没有最好，只有最合适——这和做任何工作的道理一样，实事求是、量力而行。我们用了两次课讲十几套方法，但真正的高手不是"会最多方法的人"，而是"能根据风险与成本挑对方法、并且说得出理由的人"。希望大家把这种"看条件做选择"的思维带到课程设计作品里，也带到今后的工程实践中：不迷信工具、不堆砌指标，把有限的时间投到最可能出事的地方——这既是对项目负责，也是对用户负责。

> **过渡：**方法讲完了，下面用两道思考题把这一章的"为什么"再收一遍。

【3.106】思考题（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

这一页两道思考题，请大家当成复习提纲，答案我在这里讲清楚，课后自己再写一遍。

第一题：为何在航空航天应用软件中会使用 MC/DC 代码覆盖标准？ 先明确 MC/DC 是什么——修正条件／判定覆盖，它要求每个原子条件都能**独立影响**判定结果，并且每个判定的真、假分支至少要各取一次。它的强度接近多条件覆盖，但用例数量只需 O(2n) 量级，而多条件覆盖要 2^n，工程上做得到。为什么航空航天要用它？因为这类软件属于**安全关键系统**，一旦失效可能直接危及生命，缺陷代价不可承受，而且**上天之后无法回滚**，只能在事前把正确性做到极致；同时航空机载软件的标准（DO-178C 对最高安全等级）就要求取得这种强度的结构覆盖证据。MC/DC 还有一个不可替代的价值：能逼出"短路求值掩盖的条件缺陷"——如果某个条件从来没有真正决定过结果，它的错误会被别的条件掩盖，只做判定覆盖根本发现不了。

第二题：为什么敏捷测试中适合使用基于场景的测试设计方法？ 敏捷的三个前提是：需求用用户故事表达、迭代快速、文档轻。逐条规格验证在这种节奏下成本太高、也跟不上变化。场景法直接把"用户怎么用"画成流程——基本流加各种备选流，一条场景就是一条端到端的真实路径，天然对齐用户故事。它有三个好处：用例能随迭代增量维护，只补变动的那条备选流；业务人员看得懂，方便一起评审；按风险排序后，时间不够就优先跑高价值路径，配合自动化回归，每次迭代都能快速回答"关键路径有没有被打断"。

把两题合起来看，就是本章的一个总结论：**结构覆盖回答"代码有没有被测到"，场景法回答"用户要做的事能不能走通"**——安全关键系统靠前者取证，业务系统靠后者保价值。

板书： MC/DC = 每个条件独立影响判定 + 真假分支各一次；强度：组合 ⊃ MC/DC ⊃ 判定-条件 ⊃ 判定；用例 O(2n)｜DO-178C 安全关键=不可回滚 → 事前做到极致｜场景法 = 基本流+备选流，对齐用户故事、随迭代增量维护、按风险排序

提问： 既然 MC/DC 这么强，为什么互联网业务系统不全面采用它？

预设回答：

- 答"因为贵"：✅ 对了一半。追问：贵在哪？——插桩与用例设计成本、需要源码与完整工具链、需求变化快时用例维护成本高。

- 答"因为互联网不怕出错"：❌ 要纠正。互联网也怕出错，只是它用别的手段兜底——灰度发布、监控告警、快速回滚、A/B 实验，把"事后快速恢复"当成质量策略；而航空软件一旦上天就无法回滚。这本质上是**修复成本与可恢复性**决定的策略差异。

- 答"两者不冲突，按风险分级用"：✅✅ 最接近工程实践——安全关键模块上 MC/DC 甚至形式化，普通业务模块用场景法加自动化回归。

易错点 / 考点： 常错在把 MC/DC 说成"比条件组合覆盖更强"。条件组合覆盖要求所有条件取值的**全部组合**；MC/DC 只要求每个条件**独立影响**结果并覆盖真假分支，强度低于全组合、远高于判定覆盖，是用例数量与覆盖强度之间的折中。记住这条链：**组合覆盖 ⊃ MC/DC ⊃ 判定-条件覆盖 ⊃ 判定覆盖；而条件覆盖与判定覆盖互不包含。** 另外判断题高频："MC/DC 要求每个条件取遍真值表中的所有组合"——**错**，那是条件组合覆盖。

过渡： 方法落到手上就是作业，下面两道作业请把提交要求记清楚。

【3.107】作业一（1.5 分钟）

☐ 已讲 ☐ 需回看　　课堂记录：_____

作业一是覆盖率实操。任务是这样：**找一段合适的函数代码，选一个覆盖率工具，完成三种覆盖率的测试**——DC（判定覆盖）、CC（条件覆盖）和 BPC（基本路径覆盖），另外 MC/DC 也一并给出。工具页上列了十个可以选：Testwell CTC++、CoverageMeter、BullseyeCoverage、GCT、CppUnit、Dynamic Code Coverage、TCAT C/C++、COVTOOL、gocv、xCover，选一个你装得上的就行。

做法分六步。第一步，选代码——建议挑**含两到三个判定、并且判定里出现 && 或 ||** 的函数，这样才看得出 DC、CC 与 MC/DC 的差别；只有一个 if 的函数做不出层次感。第二步，准备可编译工程和单元测试框架。第三步，插桩后用测试跑出 DC。第四步，看 CC 的缺口，**为每个条件补出让它单独取真、取假的用例**——这一步其实就是 MC/DC 的做法。第五步，画控制流图、算圈复杂度 V(G)＝E−N+2（也可以直接数判定数加一），导出独立路径集，为每条路径补用例把 BPC 拉到 100%。第六步，出报告。

报告里必须有三样东西：**控制流图与 V(G) 的算法过程、三种覆盖率的报告截图**、以及**未能覆盖的分支及其原因**。第三样最能区分认真和应付——不可达代码、异常保护分支、编译条件分支都要写清楚为什么覆盖不到，不能一句"无法覆盖"了事。

板书： DC＝判定的真假分支各≥1 ｜ CC＝每个原子条件的真假各≥1 ｜ BPC＝按 V(G)=E−N+2 导出独立路径集并各覆盖一次｜步骤：选代码（含 &&/||）→ 插桩 → 跑 DC → 补 CC → 补 BPC → 出报告（图＋V(G)＋截图＋未覆盖原因）

提问： 你的报告里 DC 是 100%，而 CC 只有 60%，这说明了什么？下一步该怎么做？

预设回答：

- 答"说明代码质量好"：❌ 典型误判。DC 100% 恰恰不保证条件被验证过，只说明每个判定的两个结果都出现过，有些条件可能从未单独起过作用。

- 答"说明用例不够，要补条件级用例"：✅ 追问：怎么补？——按 CC 的缺口，为每个条件构造"让它取反、判定结果随之改变"的用例，这正是 MC/DC 的思路。

- 答"说明存在不可达代码"：✅ 有可能，但要在报告里给出理由（异常保护、编译分支等），不能拿它当借口不做分析。

易错点 / 考点： 必须记牢 **DC 与 CC 互不蕴含**——满足了判定覆盖不一定满足条件覆盖，反之亦然，所以只报一个指标是不充分的。评分点三条：控制流图与 V(G) 是否算对；用例是否接近最少（不是越多越好）；**未覆盖分支是否给出原因**。另外注意工具差异：有些工具只报语句/分支覆盖，条件级要手动导出条件取值表。

【思政落点 26】 这份作业表面上是跑覆盖率，实质是练"证据意识与诚信"。覆盖率跑出来不理想，**不要改代码去凑数，也不要把未覆盖的分支悄悄从报告里删掉**——如实写清"没覆盖什么、为什么没覆盖"，这才是工程上真正有价值的信息，也是评审者最看重的部分。将来走上岗位，你交的每一张测试报告都是要签字的：数字真不真，就是职业底线。

过渡： 第二道作业把组合测试落到工具上。

【3.108】作业二（2 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

作业二是组合测试实操，用工具 **ACTS** 完成不同强度的组合测试。题目给了七个因素：**驾驶记录**五个水平、**汽车型号**四个水平、**使用汽车的方式**三个水平、**所住的地区**四个水平、**受保的项目**三个水平、**司机的驾龄**五个水平、**保险方式**三个水平。先算规模：5×4×3×4×3×5×3，全组合是 **10800 条**——这个数字请一定自己算一遍，很多同学会把水平数当成因素数，一算就错。

10800 条显然不可能全测，所以要用组合强度来裁剪。**1-way**：每个因素的每个水平至少出现一次，用例数大约取水平数最多的那一档，十几条；**2-way（pairwise）**：任意两个因素的取值组合至少覆盖一次，通常十几到二十几条；**3-way、4-way** 依次递增。你要分别生成这三档，对比用例数，并说明你最终推荐哪一档、为什么。

关键一步是**约束条件必须显式声明**，否则 ACTS 会生成现实中不存在的组合。题目里至少有这三组互斥或蕴含关系：**驾龄"≤1 年"与"高档进口车（≥100 万）"通常互斥**；**保险方式"首次参保"与"连续受保 ≥3 次"互斥**；**驾驶记录"过去 5 年内没有违规"与"过去 1 年内违规 3 次或以上"互斥**。把这些用约束表达式写进 ACTS，再生成测试集。

提交要求：ACTS 导出的用例表（因素列＋用例编号）、三档强度的用例数统计、与全组合 10800 条的压缩比，以及一段说明"为什么选这个强度"。

> **板书：** 7 因素｜水平数 5・4・3・4・3・5・3 → 全组合 10800｜强度：1-way（十几条）・2-way/pairwise（十几~二十几条）・3-way、4-way 递增｜约束示例：驾龄≤1 年 × 高档进口车 ・ 首次参保 × 连续受保≥3 次 ・ 5 年无违规 × 1 年内违规≥3 次｜提交：用例表＋用例数对比＋压缩比＋强度选择理由

提问：为什么两两组合（pairwise）通常就够了？如果只做 1-way，会漏掉什么？

预设回答：

- 答"因为大部分缺陷是两个因素交互引起的"：✅ 追问：举个例子？——"首次参保＋高档进口车"这种组合才可能触发核保规则异常，而单看任何一个因素都完全正常。

- 答"1-way 也够，省事"：❌ 要纠正。1-way 只保证每个因素的每个水平都出现过，两个因素之间的搭配可能

大量缺失——比如"农村＋出租车"这条组合从没出现过，恰恰可能漏掉一条业务规则。

- 答"强度越高越好，直接上 4-way"：❌成本失控。追问：10800 条和二十几条的差别意味着什么？——执行时间、用例维护成本、以及"跑不完等于没测"。工程上就是在覆盖强度与可执行性之间取舍——这正是本章反复强调的"以可接受的成本发现更多、更严重的缺陷"。

> **易错点 / 考点：**三个易错点——① **忘记声明约束**，生成"首次参保且连续受保≥3 次"这类不可能组合，用例白跑；② **把水平数当因素数**算错全组合；③ 报告只贴用例表，不写"为什么选这个强度"。评分点：约束表达是否正确、三档强度的用例数对比、压缩比是否算对、强度选择的理由是否站得住。

> **过渡：**好，这一章就讲到这里。

【3.109】感 谢 聆 听（0.5 分钟）

☐ 已讲　☐ 需回看　　课堂记录：_____

> 以上是本节课的内容，第 3 章到此结束。我们用了两次课，把软件测试的方法从"凭直觉和经验"一路走到"基于模型与形式化"：错误猜测与探索式测试解决需求不清、时间紧的开局；等价类划分与边界值分析守住输入域；判定表、因果图、Pairwise、正交实验法处理组合与约束；逻辑覆盖与基本路径覆盖回答"代码测到什么程度"；缺陷模式、基于模型的测试、模糊测试与变异测试把经验和模型变成可复用的武器；最后形式化方法与基于场景的测试，一个面向安全关键的高可信，一个面向业务与敏捷。请回去把本章小结那张方法地图和两道思考题结合起来复习，两次作业按要求提交。
>
> 我是广东金融学院的周宇文，课件和资料会发到课程群，有问题随时找我。下一章我们讲**软件测试流程和规范**，开始动手做课程设计作品。感谢聆听。

> **板书：**方法地图（经验 → 输入域 → 组合 → 逻辑覆盖 → 缺陷模式 → 模型 → 形式化 → 场景）｜下次课预告：第 4 章 软件测试流程和规范｜两次作业提交要求 · 联系方式

> **提问：**请用一句话说出本章你最有收获的一个方法，以及你打算把它用在课程设计作品的哪一步。

> **预设回答：**①"边界值分析，我打算用在输入校验的用例设计上"——落地得很具体，好。②"MC/DC，我想试试给自己的代码跑覆盖率"——动手意识好，注意先选含 && 或 || 的函数才有层次感。③"基于场景的测试，我想先画基本流和备选流"——这个思路直接对接课程设计的需求分析与用例设计，方向对。

> **易错点 / 考点：**收尾前再点一遍本章三个高频错点——探索式测试不等于随意乱点（要有章程与记录）；DC 与 CC 互不蕴含，只看一个指标不充分；覆盖率是证据不是目标，不可达分支要说明原因而不是删掉。判断三步法：先问"测哪一层"，再问"测到什么程度"，最后问"成本与风险是否匹配"。

> **过渡：**第 3 章结束。下次课进入第 4 章"软件测试流程和规范"，我们开始把方法装进流程里，为课程设计作品打基础。

四、思政落点清单（共 26 处）

序号	所在页	落点主题
1	3.8	讲一个关于知识体系与话语权的故事：SWEBOK 是国际上通行的软件工程知识体系，我
2	3.11	请大家注意课件里的第二条来源——"产品的以前版本曾经发现的错误"。这句话的分量在
3	3.13	刚才那笔账——2 的 64 次方组数据、5 亿年——把一个工程现实摆在所有人面前：资源永
4	3.14	等价类划分的完备性要求"所有可能的输入都必须归入某一类，一个都不能漏"，互斥性要
5	3.16	边界值分析讲的就是"刚刚好"和"差一点点"——多一个字符、少一分钱、差一天。真实
6	3.24	判定表最可贵的，是把口口相传的业务规则变成一张谁都看得懂的表格：条件、取值、动作
7	3.29	六个符号训练的是"把话说到没有歧义"的能力。工程上最贵的错误往往不是写错代码，而
8	3.32	Pairwise 背后是一种朴素又高级的做事方法：资源有限，不能平均用力，要抓住主
9	3.38	ACTS 界面上默认选中的 IPOG 和一旁的 Recommended，背后是几代研究者
10	3.40	正交实验法最能体现科学精神的，是它敢给自己划一条可验证的红线：9 个用例不是"差不
11	3.42	这张小结图值得多说一句：同一个系统，从"输入域"看、从"组合"看、从"状态"看，
12	3.43	覆盖率是"证据"，不是"目标"，这句话请写进笔记。行业里真实存在的坏做法是：为了
13	3.56	航空软件质量标准 DO-178C 明确规定，影响飞机起飞与降落安全的 A 等级软件必须满
14	3.63	基本集导出的用例，是要拿来说"这段代码我确实测到了"的证据。这里有一条职业底线值
15	3.65	从语句覆盖到判定覆盖、条件覆盖，再到 MC/DC 和基本路径覆盖，这一族方法是典型的

16 3.67 缺陷预防优于事后返工。怎么联系知识点：DPBT 的做法是把历史缺陷抽象成模式，再拿

17 3.68 讲什么：把个人踩过的坑写成缺陷模式、写进检查单，而不是留在自己脑子里。怎么联系知

18 3.73 讲什么：MBT 的需求追溯矩阵与模型覆盖率是自动产出的。怎么联系知识点：它要求需求

19 3.81 AFL 把人从"一条条手写畸形数据"的重复劳动里解放出来：机器负责海量试错与状态记

20 3.82 变异测试本质上是"自己给自己出难题、自己审自己"：没有人强迫你把程序改坏，是你主

21 3.85 形式化方法为什么最常用在核电站控制、航天器控制、导弹防御这类系统上？因为那里的每

22 3.96 这一页要求我们先问"谁在用、在什么条件下用、想办成什么事"，而不是"代码里有哪些

23 3.98 练习里最容易走偏的，是为了让作业"看起来完整"而编造预期结果，或者把没测到的分支

24 3.104 这张条件覆盖表不是交完就算的一次性作业。把判定、原子条件、用例数据、预期结果整理

25 3.105 方法没有最好，只有最合适——这和做任何工作的道理一样，实事求是、量力而行。我们用

26 3.107 这份作业表面上是跑覆盖率，实质是练"证据意识与诚信"。覆盖率跑出来不理想，不要改