《软件测试课程设计》讲稿（第 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 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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

> 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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

> 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）

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）

> 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）

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）

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）

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）

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）

> 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）

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）

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）

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、T	a>0 AND	M＝.T.　N

		4		b>0	=.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）

> 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）

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）

> 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）

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）

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）

> 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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

> 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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

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）

> 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）

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）

> 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）

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）

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）

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）

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）

> 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）

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）

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）

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）

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）

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 →	账号 zhangsan／密码 Abc123；商品	订单生成成功，库存扣减 1，

加入购物车 → 提交订单 → 支付成功 → 订单生成	编号 SKU-1001；数量 1；支付方式余额充足	支付流水一条，页面跳转订单详情
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）

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）

> 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）

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）

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）

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）

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）

> 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）

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）

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）

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）

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