《软件测试课程设计》讲稿（第 11 章）

第 13 周 · 2 学时

广东金融学院 计算机学院 | 周宇文

讲稿可照读版 | 由课件逐页撰写

第 11 章　设计和维护测试用例

第 13 周 · 2 学时（90 分钟）

《软件测试课程设计》第 11 章 讲稿（第 13 周 · 2 学时 90 分钟）

> 课程：软件测试课程设计（154442008） | 广东金融学院 计算机学院 | 专业选修
>
> 章节：第 11 章 设计和维护测试用例 —— 用例是团队的资产，不是一次性的草稿
>
> 学时：**2 学时（90 分钟）**
>
> 教材：朱少民《软件测试方法和技术（第 4 版）》第 3 篇 软件测试项目实践
>
> 配套课件：逐页还原版 **40 页**（`lecture11/`，云端 http://111.231.0.226:2345/lecture11/index.html）
>
> 教案对应：第 13 次课「设计和维护测试用例」
>
> 本讲重点：用例构成要素与书写标准；用例的作用与价值；粒度与组织；设计方法的组合；用例评审；维护与复用；与需求的追溯；用例度量
>
> 本讲难点：用例粒度的把握（太粗漏测、太细维护不动）；需求变更时用例库怎么跟着走；把用例从"文档"变成"资产"
>
> **本讲稿为可照读完整版**：每页含【口播】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为可压

> 缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出测试用例的八个构成要素；② 能说明用例的四项作用（可复用、可度量、可追溯、防遗漏）；③ 能描述用例书写标准的关键要求；④ 能说出用例维护的时机与方式；⑤ 能解释用例有效性、覆盖率、缺陷发现率三个度量
能力	① 能为一个功能点写出 5 条符合标准的用例（含前置条件、步骤、数据、预期、优先级、需求追溯）；② 能对别人的用例做评审并给出修改意见；③ 能设计一份用例库的版本与基线管理方案
素质（思政）	① 用例作为知识资产＝为团队留下东西；② 书写规范＝严谨表达；③ 维护与复用＝对变更负责、不糊弄

本讲要建立的一个总印象：**好的用例，换一个人拿着它也能把同一个场景测出来**；做不到这一点，写的就是"给自己看的笔记"，不是团队资产。

二、时间分配总表

> **容量提示**：本章课件 **40 页**，讲稿为素材全量版（口播约 1.9 万字，≈80 分钟），**90 分钟可以按页讲完**。

时段	页码	内容	分钟	备注
第 1 节	11.1–11.12	本章问题、用例构成、重要性、书写标准	24	**构成要素与书写标准必讲**
第 1 节	11.13–11.26	粒度与组织、设计方法组合运用	22	当堂写 5 条用例
——	——	**课间休息 10 分钟**	——	——

第 2 节	11.27–11.36	用例评审、维护与复用、追溯矩阵、度量	26	**维护与复用必讲**
第 2 节	11.37–11.40	小结、作业、结束	8	——

> **压缩优先级**：先压缩示例罗列页 → 再压缩单一方法的细节（方法在第 3 章已讲）→ 最后才压缩构成要素、书写标准、维护复用、度量。

三、逐页讲稿

> 每页第一段是可直接朗读的口播；其后是提问与预设回答、易错点/考点、过渡语。
>
> 页码锚点与课件页一一对应（11.1–11.40）：http://111.231.0.226:2345/lecture11/index.html
>
> **口径提醒**：素材第 6 页沿用了一个旧标题"4.1 传统的软件测试过程"，属上一版残留；讲稿按该页实际内容讲，标题用内容概括，不照抄错标题。

【11.1】软件测试方法和技术·第 11 章 设计和维护测试用例（1 分钟）

> 三年后你离开这个组，新人接手这个模块，他凭什么知道哪里测过、哪里一碰就出事？答不上来，问题不在他，在这个团队没有一份能看的用例库。

>

> 前面十章，我们一直在回答"怎么想"。等价类、边界值、判定表、路径覆盖，这些方法都在你手上。可方法只解决想不想得到，它不管你想出来的东西写不写得下来、别人接不接得住。

>

> 中间这段空白，就是第 11 章补的。章名两个动词。设计，把脑子里那点测试行为变成一份能交出去的文档。维护，让这份文档跟着软件一起活，版本走到哪，它跟到哪。

>

> 为什么这两件事要放一章？因为写下来只是开头。写了不维护，两年后它比没有更糟，它会骗人。库里写着"已通过"，那条规则早改过了。新人照着跑，跑出问题就怪软件。

>

> 课程设计里你们马上会撞上。答辩时老师问"这个模块怎么测的"，你答"我点了一遍"。"点了一遍"没法复现，也没法交给下一个人。你要交的是一份东西：别人拿着它，不问你，能跑出跟你一样的结论。

>

> 那套方法，一章章学下来，很多人以为学完就能上项目。真上了才发现，卡住的地方是写下来这一关。设计是让想法落地，维护是让落地的东西不过期。落不了地，方法白学；过不了期，落地也白落。课程设计里最常见的失分点不是不会挑用例，是用例写了只有自己看得懂，答辩时只能靠嘴补。

>

> 这条线就是整章的验收标准。后面每一段，都在往它上面靠。

> **提问：**同一段功能，让两位测试工程师各自测一遍，结果会一样吗？为什么？

> **预设回答：**

- 答"应该一样，功能就那几项"（典型错答）——现实里常不一样。两个人脑子里的边界、数据、判定标准都不同，没有书面依据就没有可比性。

- 答"不一样，看经验"——这恰好是不写用例的代价：质量取决于谁值班，不取决于流程。

- 答"写成用例就一样了"——方向对，还得补一句：写成谁都读得懂的用例才会一样。写得含糊，两个人执行还是两个结果。

- 收口：这一章治的就是这件事。把"我会"变成"纸会"。

> **易错点 / 考点：**① 章名两个动词要能分开说——设计是把想法变文档，维护是让文档跟着软件走；② 判断陷阱："用例写完就完事了"——错，用例是有生命周期的资产；③ 别把本章当第 3 章的重复：第 3 章讲用什么方法设计，本章讲设计出来的东西长什么样、怎么组织起来。

> **过渡：**先把这一章要回答的问题摆出来，你才知道后面每一段在解哪道题。

【11.2】本章要解决的问题（1.5 分钟）

> 五个问题，先别找答案。你对着自己做过的项目数一数，能答几个。

>

> 第一问，为什么要用测试用例。关键词是"使用"，不是"写"。写完锁进硬盘，等于零。它的价值在被反复执行、被反复参照的时候才发生。这一问后面用一整段讲六项作用。

>

> 第二问，用例由哪些基本元素组成。前置条件、测试数据、测试环境、操作步骤、期望结果，这五样是底子。往上还有编号、标题、优先级、层次、估计用时、追溯需求这些扩展属性。缺了基本元素，用例不成立；缺了扩展属性，用例不好管。

>

> 第三问，编写和设计要守哪些原则。这一问是红线：不写含糊的、不写冗长的、不写重复的、一个用例一个验证点。

>

> 第四问，结构和设计过程是什么样。一条用例、一套用例、一个套件、一份测试计划，这是一棵树，从叶子往上长。长的时候先长哪根枝，后面会讲。

>

> 第五问，怎么跟踪和维护。用例会过期、会失效、需求一变它得跟着改。谁改、什么时候改、怎么留痕，这一问最容易被跳过，工程上却最贵。

>

> 五问其实是一条线上的五个结。先决定写不写，再决定写成什么形状，再决定想多深，最后决定怎么让它活下来。结解不开，后面全卡着。考题常按这个顺序出简答，笔记照这个骨架记，答题就不会漏块。

>

> 课件底下还有一句：具体的设计方法，第 3 章已经讲过。所以这一章不教你等价类怎么划、边界值怎么取。它管的是，那些方法产出的结果写成什么样子、怎么组织、怎么活下去。

提问：这五问里，你认为实际项目中最容易做砸的是哪一问？

预设回答：

- 答"第一问，很多团队根本不写"（典型错答）——确实普遍。可要追问一句：不写是因为不会写，还是因为写

了没人用？两种病，药方完全不同。

- 答"第三问，写得太啰嗦"——很具体。冗长会让执行变慢、维护变贵，还容易漏判，第 19 段给"步骤不超过 7 步"这种硬口径。

- 答"第五问，写完就没人管了"——最接近真相。行业里用例库死亡的方式几乎只有一种：需求改了，用例没改，跑两轮对不上，最后没人信它。

- 收口：前四问答得好叫会写，第五问答得好才叫会管。

> **易错点 / 考点：** ① 五问顺序要能背，简答题常按这个框架给分；② 常考判断："测试用例设计方法属于本章内容"——错，方法在第 3 章；③ "使用"和"编写"要分开，答"用例的价值在执行与复用阶段实现"比答"用例能发现缺陷"更准。

> **过渡：** 五问落到两节，我们先看这两节各管什么。

【11.3】11.1 测试用例构成及其设计 / 11.2 测试用例的组织和跟踪（1 分钟）

> 一条用例和一库用例，是两门手艺，别当一件事。

>

> 第一节，11.1 测试用例构成及其设计，管的是"一条"。它由什么构成、为什么非写不可、书写标准是什么、设计时要考虑哪些因素、要守哪些原则。四个小节是递进的：先说服装你写，再规定格式，再提示你该想什么，最后给底线。

>

> 第二节，11.2 测试用例的组织和跟踪，管的是"一堆"。给用例挂上可管理的属性，用测试套件把它们组织起来，执行过程中跟踪进度，跟着版本做维护，最后用覆盖率检验这套东西够不够。

>

> 为什么非要分两节？一条写得再好，也抵不过一库乱。反过来，库管得再整齐，条目本身含糊也白搭。所以表达力和管理力都得练，缺一边都撑不住。

>

> 现实里很多团队第一节做得还行，第二节完全空白。结果就是用例越写越多、越写越乱，最后没人打开。不

> 是用例没用，是没人管得住它。

>

> 分节还有个好处，是考法清楚。问用例管理的两个层面，你答 11.1 和 11.2 各自管什么；问一条用例怎么算写好，只在 11.1 里找答案。两节混着答，阅卷的人一眼看出你没把书读成结构。还有一条常错：覆盖率归 11.2，不归 11.1，它检验整套用例够不够用，不是某一条写得对不对。两节名字里都有测试用例四个字，剩下的才是区别：构成、设计、组织、跟踪。备考时按节画一张表，比通读省时间。

>

> 一句话分开：第一节决定你写的东西有没有用，第二节决定它能不能活过三个版本。

> **提问：**如果只能先做好一节，你先做哪一节？为什么？

> **预设回答：**

- 答"先做 11.1，用例都写不规范谈什么管理"——合理。但要补一句：规范能靠模板一次性解决，管理不行。

- 答"先做 11.2，先有库再谈质量"——也成立。很多团队就是先建库、边用边改。

- 答"两节是一回事"（典型错答）——要纠。11.1 的目标是这条用例别人能照着跑出同一个结论；11.2 的目标是这个库别人找得到、跟得上、改得动。

- 收口：先做哪节看团队现状。考"用例管理的两个层面"，两节的名字都得点到。

> **易错点 / 考点：**① 两节名称与各自小节要能对上号，别把覆盖率错记到 11.1；② 判断陷阱："用例组织和跟踪属于测试执行阶段的工作"——错，它是设计之后的持续性工作，跨阶段存在；③ 11.2 里有"属性"这一节，最容易被漏。

> **过渡：**进 11.1 之前，先答一个更朴素的问题：我们到底在描述什么？

【11.4】如何描述测试行为？（1.5 分钟）

> 你脑子里其实很清楚要测什么。麻烦出在说给别人听，那句话一出口就开始漏。

>

> 先看清那团东西是什么。它是一团混合物：我想验证哪条规则、我准备给它什么输入、我怎么判断它对了、

> 我在什么环境里做。这四样在你脑子里同时存在，所以你觉得自己"很清楚"。

>

> 可一旦要交给别人，这团东西必须摊成一条路径：从什么状态出发，做什么动作，得到什么，跟什么比。摊不开，别人只能猜。猜出来的结果，跟你跑出来的不是一回事。

>

> 所以我给"描述测试行为"下一个工作定义：去歧义。不是把话说漂亮，是把所有可能被误解的地方堵死。

>

> 举个例子。你写"输入一个非法密码"，别人要追问四句：多长算非法、几位、含什么字符、大小写算不算。每问一句，都是你的描述漏了一处。四句问完，这句话基本得重写。

>

> 生活里最像的东西是菜谱。"盐少许"靠师傅的手感，"盐 3 克"靠纸。测试用例要的是后者，因为执行的人不一定有你那双手。

>

> 再举个更近的例子。你写输入手机号，同事照样要问：11 位还是能带 +86，中间带空格算不算错，空号能不能过，测试该用哪几个号。这些没人替你想，全得写进去。所以描述这件事，是替别人把他会问的问题先答掉。答得掉，用例站得住；答不掉，它还只是你脑子里的话。写得清楚不是为了好看，是为了让别人不必来问你。这一条放到答辩上也成立：老师听你说测过了，一定追问怎么测的。

提问：你写"输入一个非法密码"，同事会追问你几个问题？请具体说出三个。

> **预设回答：**

- 答"长度多少、有没有特殊字符、是不是必填"——三个都问在点上。一句话里至少藏了三处歧义。

- 答"不用问，一看就懂"——最典型的错答。你懂，是因为你脑子里有上下文；读者没有你的上下文。

- 答"那我把所有情况都写上"——另一个极端。写成一整段，读者反而找不到判定点，这就是后面要讲的冗长。

- 收口：描述的目标不是写得多，是歧义为零、判定唯一。

易错点 / 考点：① 描述的四个落点：验证什么、输入什么、什么环境、怎么判定；② 判断陷阱："用例写得越长越完整"——错，完整指信息齐全，不指篇幅长；③ 简答常问"为什么测试行为必须文档化"，答题要点是可

> 执行、可判定、可交接、可复用。

> **过渡：**抽象说完了，看一条具体的描述长什么样。

【11.5】示例（1.5 分钟）

> 先看一条写得像样的用例。别急着挑毛病，先学会它怎么读。

>

> 读的顺序建议反过来：先看判定，再看前提，最后看步骤。

>

> 为什么先看判定？一条用例存在的理由就是能判对错。判定含糊，后面写得再细也全废。判定清楚，你才知道这条用例到底要证什么。

>

> 再看前提。执行者得知道从哪个状态起跑。前置条件不同，同样的输入会有不同结果。这时候结果对不上，不是缺陷，是用例没写清。

>

> 最后看步骤。步骤是路径，路径错了，判定也就落不到实处。

>

> 这三步的顺序不是随手排的。判定决定这条用例值不值得留，前提决定它能不能被复现，步骤决定它跑起来顺不顺。三样都过关，才算一条能用的用例。刚上手的人爱从步骤读起，因为它最长。读到末尾常会发现，判定的那一句根本判不了。示例给的是格式的下限，不是上限。看懂下限，你写出来的东西至少能交差。

>

> 拿一句常见的话对照："登录功能测一下"。这句话里，验证什么不清楚，输入不清楚，环境不清楚，判定更不清楚。它不是用例，是任务。任务靠人补全，用例靠纸补全。

>

> 落到用法上：示例就是给你一个能照着改的模板。课程设计最省力的起点不是打开空白文档从零想，而是找

> 一条同类用例，把编号、前提、数据、步骤、判定五处逐一替换。改十条，格式的规律就摸到了。

>

> 还有一层：示例里的数据是示意，不是标准答案。数据要能反映你要验证的那条规则，别照抄。

> **提问：**一条用例的期望结果写成"系统正常"，你能不能执行？为什么？

> **预设回答：**

- 答"能执行，但不一定能判定"——抓住了关键。动作能做完，"正常"没法判。

- 答"不能，正常是什么没说"——完全正确。要改成"提示密码错误、停留在登录页、账号未被锁定"这种看得见的句子。

- 答"正常就是没报错"——最典型的错答。不报错不等于做对了，很多缺陷的表现恰恰是静默地做错。

- 收口：判定的标准是可观测——界面看得见、日志查得到、数据能比对，三者占一个。

> **易错点 / 考点：**① 判定要可观测，禁止"正常""正确""符合预期"这类自我指涉的词；② 前置条件是执行起点，不是背景介绍，写多了没用，写漏了跑偏；③ 案例题常给一条含糊用例让你改，改法是逐处补"谁、在什么状态、给什么、得到什么"。

> **过渡：**那条用例由哪些零件拼起来？进 11.1 的第一小节。

【11.6】用例设计的基本流程：11.1 的四个小节（1.5 分钟）

> 一条用例要过四道关才算能交出去。这四关就是 11.1 的四个小节。

>

> 第一关，11.1.1 测试用例的重要性，回答值不值得写。这一关我不讲情怀，讲代价：不写会出什么事——进度不可测、覆盖不可证、人一走经验清零。这三笔代价算清楚，写用例就不用靠自觉了。

>

> 第二关，11.1.2 测试用例设计书写标准，回答一条用例由哪些部分构成、每部分填什么。它看着最像格式要求，实质是一份契约：写的人和读的人，对同一条用例必须得出同一个理解。契约不清，后面全是扯皮。

>

> 第三关，11.1.3 测试用例设计考虑因素，回答设计时脑子里要同时想哪几件事——要有代表性，要盯设计弱点，正输入也要异常输入，还得想用户真实会怎么用。

>

> 第四关，11.1.4 测试用例设计的基本原则，给红线：不含糊、不冗长、不重复、一个用例一个验证点。

>

> 四关的顺序是：要不要写 → 写成什么形状 → 该想什么 → 不许怎么写。按这个顺序记，比按编号背牢。

>

> 这四关也解释了一个常见困惑：有的团队用例写得整整齐齐，还是没人用。整齐只过了第二关，第一关和第三关都没过——不值得写的也写了，该碰的弱点一个没碰。用例的价值不在整齐，在能挡住事。整齐是给别人看的，挡住事是给上线那天用的。四关里最常被跳过的是第一关，因为它问的是值不值得，答起来最费劲。

>

> 再补一句：四关不是走一次就完，是每条用例都要反复过的四问。值不值得存在、格式齐不齐、异常想到没、有没有含糊和冗余。

提问：为什么"书写标准"要放在"考虑因素"前面？换个顺序行不行？

预设回答：

- 答"行，先想再写更自然"——从个人习惯说没错。从协作看不行：没有统一形状，各人想出来的东西没法互相比较，评审也无从谈起。

- 答"先定格式再谈内容"——对。格式是公共语言，内容是个人判断。公共语言先立，个人判断才谈得上交流。

- 答"顺序无所谓"（典型错答）——顺序体现的是教学逻辑。考"概述用例设计的过程"，按四步答才拿得全分。

- 收口：格式先行不是形式主义，是为了让"想"这件事能被别人检查和质疑。

易错点 / 考点：① 四个小节名称与顺序必须记准；② 课件顶上的标题还留着上一版的字样，按内容它就是 11.1 的四个小节，笔记按"重要性—书写标准—考虑因素—基本原则"记；③ 判断陷阱："本章会讲等价类、边界值的设计方法"——错，方法在第 3 章。

过渡：从"值不值得写"开始。

【11.7】什么是测试用例？（2 分钟）

同一件东西，教材给了三种说法。你信哪个？三个都得信，它们是同一件事的三个切面。

>

先叠成一句：测试用例是把一次测试行为，写成谁都照着能做、能判的文档。

>

第一种说法，可以独立执行的最小单元。"独立"和"最小"是两个关键词。独立，是不依赖别的用例的结论，拿出来就能跑。最小，是不能再拆，再拆就判不出结论。这两个词定的是粒度。

>

第二种说法，一系列情景，加每个情景必须依靠的输入和输出，用来判断软件的正确性。这句长，结构却清楚：情景、输入、输出，三样合起来才是判断的依据。"必须依靠"四个字要留意——输入输出不是随手记的，是判定必需的那部分。

>

第三种说法，把测试的行为活动转成规范化的文档。这句最提纲挈领。行为是活动，活动会消失；文档会留下。所以用例的本质是一次转化：把只有你会做的动作，变成谁都能做的说明。

>

三层叠起来就明白了：它既是动作的说明书，也是判定的依据，还是交接的凭据。

>

三种说法来自不同版本的教材，措辞不一样，指向的是同一件事。答题按题目给的版本答，别把三句揉成一句四不像。真要背定义，背第三种最省力。前两种适合答简答：一个说粒度，一个说判定。

>

还有个推论容易漏。既然是文档，它就有读者：写的、执行的、评审的、三个月后来接手的。判断一条用例好坏，最省事的办法是找个没参与过的人读——他十分钟内能不能跑出跟你一样的结论。

提问：一条用例测完，能不能紧接着执行下一条而不做任何准备？这和"独立"是什么关系？

预设回答：

- 答"能，用例之间本来就该独立"——理想状态是这样。代价是每条都要自带前置条件，执行成本上升。

- 答"不能，上一条把数据改了"——很实际。这就引出独立的真实含义：不依赖别的用例的结论，但可以依赖前置条件写明的状态。

- 答"独立就是互不相关"（典型错答）——偏了。互不相关做不到，也没必要。工程上要的是任何一条失败不影响别的条目判定。

- 收口：独立是判定上的独立，不是环境上的隔离。两件事分开，你才会认真写前置条件。

> **易错点 / 考点：** ① 三种定义各能说一句，简答按"最小独立单元""情景＋输入输出""行为转文档"三点各给分；② 别把测试用例和测试脚本混了：脚本是可执行代码，用例是描述，脚本可以由用例生成；③ 判断陷阱："用例必须包含全部测试数据"——错，包含的是判定必需的数据。

> **过渡：** 既然是把行为变文档，这份文档到底解决什么问题？

【11.8】11.1.1 测试用例的重要性（2 分钟）

> 老板站在你工位边上问"这个版本测得怎么样"，你张嘴第一句是什么？这句话就见出有没有用例。

>

> 课件给三层理由，从商业目标一路落到质量保障。

>

> 第一层是商业目标：怎么用最少的人力、最少的资源，在最短时间内完成测试、发现缺陷、保证品质。听着像口号，其实是个硬约束——测试资源永远有限，你不可能无限测。资源有限，就必须回答先测什么、测到什么程度算够。这两个答案只能靠用例承载。没有用例，"测够了"就是一句主观感受。

>

> 第二层是可管理性：测试有组织性、步骤性、计划性。要把测试行为转成可管理的、具体量化的模式，就得创建和维护用例。"量化"是关键词。进度是执行了多少条、通过率多少、还剩多少条没跑，这些全是数字。没有用例，你只能说"差不多了"，说不出"还差 37 条，12 条卡着跑不动"。

>

> 举个具体的。一轮回归 200 条，跑完 160 条通过，30 条失败，10 条没跑。这组数字能撑起一句话：这个版

> 本可以发，但还有 10 条关键路径没盖。没有用例，这句话说不出来，只能说感觉还行。分母这个词很关键。没有分母的百分比，都是自己骗自己。

>

> 第三层是准则与保障：用例是测试工作的指导，是必须遵守的准则，更是质量稳定的根本保障。

>

> 三层连成一条链：资源有限 → 必须量化 → 量化靠用例 → 用例稳，质量才稳。

>

> 反过来看代价：不写用例的团队，进度靠感觉、覆盖靠回忆、人一走靠运气。三样凑一起，质量一定是波动的。

> **提问：**老板问你"这个版本测得怎么样了"，手上没有用例，你能给出的最诚实回答是什么？

> **预设回答：**

- 答"测完了，没问题"（典型错答）——最危险的一句。它既不可证也不可追，出了事没法复盘是漏测还是没测。

- 答"大概测了七成"——听着像量化。可七成按什么算？没有分母就没有百分比。

- 答"我不知道，得看还剩哪些没测"——这就是正确答案的雏形：承认需要一个分母，分母就是用例集。

- 收口：用例的第一个作用不是防缺陷，是让测试工作可被度量。度量立住，防缺陷才有抓手。

> **易错点 / 考点：**① 三层理由的顺序与关键词（最少投入、可量化、准则保障）；② 常考简答"没有测试用例会带来哪些问题"——进度不可测、覆盖不可证、经验不可传、重复劳动；③ 判断陷阱："用例的重要性主要体现在发现缺陷的数量上"——错，主要体现在可管理性与质量稳定性上。

> **过渡：**重要性讲了三层，作用就一条条列清了。

【11.9】测试用例的作用（2 分钟）

> 六项作用，年年考，年年有人只答得出两个。我一项一项讲，每项给一个"不这样会怎样"。

>

> 第一项，有效性。用例是围绕测试目标设计的，每次执行都指向一个明确的验证点。没有它，测试员点来点

> 去，看着也没闲着，可你说不清他到底验证了什么。

>

> 第二项，可复用性。同一批用例，这个版本跑、下个版本还跑、换个人还跑、以后还能交给自动化跑。写一次，几十次回归都在吃这份红利。

>

> 第三项，易组织性。用例天生带属性，能按模块分、按类型分、按优先级分。于是"今天下午测哪一块"这种问题，靠筛选就能回答。

>

> 第四项，客观性。这项最容易被低估。用例写了什么就测什么，判定标准事先定好，执行完只有通过和不通过两种结论。质量结论不再取决于某个人的心情和立场。测试报告要能顶住压力，靠的就是这种客观性。

>

> 第五项，可评估性和可管理性。用例数、执行数、通过率、缺陷密度，都是可统计的量。有了量，才谈得上进度、风险和资源。

>

> 第六项，知识传递。新人入职，看文档不如看用例：业务规则怎么组合、哪些边界特别容易错、历史上踩过什么坑，全沉在前置条件和判定里。

>

> 这六项不是并列的六个优点，它们有先后。先得写对，才谈客观；先能度量，才谈复用。前五项是给这个项目用的，第六项是给下一批人用的，前者保住这一次交付，后者保住下一次开工。

>

> 六项收到课件那句话上：重要参考依据，提高测试质量。

提问：六项里，哪一项在"人走了、新人接手"的场景下最值钱？

预设回答：

- 答"可复用性"（典型错答）——算得对。可复用省的主要是执行成本，人走了以后最先崩的往往不是成本。

- 答"知识传递"——对。新人最缺的不是体力，是"这块业务哪里会出问题"的判断力，那判断力就写在用例的边界和异常场景里。

- 答"客观性"——也有道理。还要说清：客观性靠判定标准事先写死，不靠某个人公正。

- 收口：六项是一条链。写得好才有客观性，有客观性才可度量，可度量才可复用，可复用才沉淀成知识。

> **易错点 / 考点：** ① 六项要能默写，简答题几乎年年出；② 最容易漏的是客观性和知识传递，补上这两条常能拉开分差；③ 判断陷阱："用例的作用就是发现缺陷"——错，发现缺陷只是有效性的一部分，其余五项指向管理与传承。

> **【思政落点 1】**：把用例当资产经营，还是当交差材料，是职业成熟度的分水岭。前者每版更新，每出一个缺陷就回填一条，三年后新人翻开库就知道坑在哪；后者评审完就锁进硬盘，人一走全废。同样的工作量，一个归零，一个增值。差别不在技术，在你有没有把"给团队留下东西"当成自己的责任。

> **过渡：** 作用清楚了，接下来是"写成什么样"。

【11.10】11.1.2 测试用例设计书写标准（1.5 分钟）

> 动笔之前，先回答四句话。四句答不上来，先别写。

>

> 第一句，目的。这条用例为了验证什么？是验证一条业务规则、一个边界、一种异常分支，还是验证性能、兼容性？目的不清，后面全是无效劳动。

>

> 第二句，输入数据。给它什么？要具体到值：用户名多长、密码几位、金额是正数还是负数、日期是不是闰年二月三十。写成"输入合法数据"，等于没写。

>

> 第三句，操作步骤。按什么顺序做？步骤要能让执行者复现，包括从哪个界面进、点哪个按钮、要不要先登出再登录。

>

> 第四句，期望结果。做完该看到什么？要可观测，要唯一，不能出现"正常""符合要求"这种自我指涉的说法。

>

> 课件最后留了一句"还有呢"，这才是真正的考点。除了这四样还缺什么？缺前置条件，执行前系统得处于什么状态；缺测试环境，什么版本、什么配置、什么数据基线；再往外，还缺编号、标题、优先级、层次、追溯需求这些管理属性。基本元素让你能执行，管理属性让它能被组织。

>

> 四句为什么按这个顺序问？因为它对应一条用例被理解的顺序。先知道要证什么，才知道给什么数据；知道给什么数据，才知道步骤怎么走；走完才知道该看到什么。

>

> 书写标准这个词听着像行政要求，实质是团队内部的接口规范：字段名统一、填写粒度统一、判定写法统一。统一之后两件事才可能——评审能逐字段对照，不必从头读；工具能按字段抽取，自动生成执行清单和统计报表。

提问：如果把"前置条件"漏掉，会出现什么具体后果？举一个你自己遇到过的例子。

预设回答：

- 答"结果对不上，别人跑和我跑不一样"——具体。前置条件不同，同样的输入本来就会出不同输出，这不是缺陷，是用例没写清。

- 答"前置条件不重要，从登录开始就行"（典型错答）——很多用例的起点是账号已被锁定、购物车已有三件商品、库里已有历史订单，从登录开始根本造不出这个状态。

- 答"前置条件就是环境"——要纠。前置条件讲数据与业务状态，环境讲版本、配置、平台，两者不能互相替代。

- 收口：前置条件是执行者最容易摔的地方，也是评审最常打回的一项。

易错点 / 考点：① 四问加"还有呢"共五项，考试常以填空或简答出现；② 输入数据必须具体到值，写"合法/非法"不得分；③ 判断陷阱："期望结果可以写与需求一致"——错，要写出可观测的具体表现。

过渡：把"还有呢"补齐，就是最基本的内容。

【11.11】测试用例最基本的内容（2 分钟）

五项内容，缺一样这条用例就废一半。课件还指了个地方，完整模板见教材第 4 版的附录 C。

>

> 前置条件，执行之前系统得处于什么状态。账号注册过没有、余额是不是 0、订单付过款没有、缓存清没清。它的作用是把起跑线画齐。起跑线不齐，同一句话在两个人手里能跑出两种结果。

>

> 测试数据，具体到值的输入，加必要的初始数据准备。这项最容易偷懒，也最容易被评审打回。"输入一个超长用户名"不是数据，"输入 65 个字符的用户名，上限 64"才是。

>

> 测试环境，操作系统、浏览器或机型、软件版本、配置参数、依赖服务的状态。同一条用例在测试环境和生产环境表现不同，是行业里最常见的争议来源。所以环境得写清，不能默认别人知道。

>

> 操作步骤，从起点到判定点之间的动作序列。要能照着做，要包含必要的等待和中间确认，不能跳步。

>

> 期望结果，做完该观测到什么。这是判据。缺了它，用例就退化成一份操作手册，测完了也不知道算不算过。

>

> 五项的关系这样记：前置条件定起点，环境加数据定条件，步骤定路径，期望结果定终点。四样齐，一条用例才闭合。

>

> 缺一项会怎样？缺前置条件，两个人跑出两个结果；缺数据，执行者自己编；缺环境，缺陷复现不了；缺步骤，判定点找不到；缺期望结果，测完了不知道算不算过。五项里最容易被省的是环境，最容易被乱写的是数据。

>

> 颗粒度还得说一句。字段一样，写多细按需。界面文案用例，环境写"默认 Windows 浏览器"就够；数据迁移用例，环境要写到数据库版本、字符集、数据量级。

提问：五项里，如果只能保留四项，你会删哪一项？删了以后风险是什么？

预设回答：

- 答"删环境，反正都在同一台机器上测"（典型错答）——风险很实在。缺陷在别的环境复现不了，你连当时是什么环境都说不清，问题会拖成悬案。

- 答"删前置条件，从登录开始重造状态"——代价是执行时间暴涨，而且有些状态（历史数据累积）根本造不出来。

- 答"一项都不能删"——最稳。非删不可时，工程上通常把环境写进版本说明统一约定，不是彻底不写。

- 收口：这五项不是五个格子，是四条边界。少一条边界，判定就漏风。

> **易错点 / 考点：** ① 五项名称与"起点—条件—路径—终点"的对应关系；② 常考填空"测试用例最基本的五项内容是……"；③ 判断陷阱："测试数据可以只写数据类型"——错，必须写到值，或写到值的构造方法。

> **过渡：** 最基本的是五项，工程上还会挂一批扩展属性。

【11.12】测试用例的元素（1.5 分钟）

> 一张元素图摊开，你先看到的应当是分层，不是一串名词。

>

> 最里面一层，判定必需：输入、步骤、期望结果。少任何一样，用例就不成立。

>

> 中间一层，执行必需：前置条件、测试环境、测试数据准备。它们不参与判定，但决定这条用例能不能被别人跑出来。

>

> 最外面一层，管理必需：编号、标题、测试目的与类型、优先级、层次（父、子、孙）、估计用时、追溯的需求或功能点。这一层不影响单条用例的对错，却决定整个库能不能被组织、被检索、被度量。

>

> 为什么分三层？因为维护成本完全不同。里面那层改一次要重跑，中间那层改一次要重配环境，外面那层改一次只是改属性。三层混着写，就会在期望结果里塞业务背景、在步骤里塞优先级。

>

> 举个混写的例子。有人在期望结果里写这是中心客户、能享受折扣，那是业务背景，不是判定；有人在步骤里写优先级高，那是属性，不是动作。评审看到这种错位，基本能判断这条用例没按层想过。三层的边界其实好记：里层管对不对，中层管跑不跑得起来，外层管找不找得到。分不清这三层，评审时就只能给一句写得不太清楚。

>

> 还有个实用判断：评审时先看最外层齐不齐，再看中间层全不全，最后看最里层准不准。从粗到细筛，比逐字读快得多。

>

> 编号这项看着最没用，工程上却非要给。因为它是唯一不变的字段。别的都能改，锚点不能改。

> **提问：**编号看起来最没用，为什么工程上非要给每条用例一个编号？

> **预设回答：**

- 答"方便排序"——作用太浅，排序用标题也行。编号真正的价值是唯一标识。

- 答"方便引用，缺陷单里能写对应用例 TC-1024"——这就是关键。编号让用例、缺陷、需求三者互相指向，追溯才有落点。

- 答"编号可以不要，标题就够了"——错答。标题会改，改完就对不上；编号一旦分配就不再变。

- 收口：编号是不变的那个字段，别的字段都能改，锚点不能改。

> **易错点 / 考点：**① 三层划分（判定必需、执行必需、管理必需）是理解这张图的关键；② 常考简答"测试用例的扩展属性有哪些"——编号、标题、目的、类型、优先级、层次、估计用时、追溯；③ 判断陷阱："优先级属于基本元素"——错，属管理属性。

> **过渡：**元素摆齐了，看一条按这套元素填满的用例。

【11.13】示例（1.5 分钟）

> 元素是骨架，示例是长肉的样子。这一条填满的用例，我教你四眼看过去。

>

> 第一眼看判定点。这条用例到底要证明什么？判定点通常一句话，写在期望结果里。判定点含糊，别的字段写得再满也是废的。

>

> 第二眼看起点。前置条件有没有把状态说死？只写"系统已启动"，那这条用例对数据的依赖就被藏起来了。执行者各自造数据，结果自然各不相同。

>

> 第三眼看路径。步骤有没有跳步，有没有隐含的中间确认？很多用例失败不是软件的问题，是步骤里少了一句"等页面加载完"。

>

> 第四眼看属性。优先级是高还是中、挂在哪个父用例下、估计用时写没写。这三项平时不显眼，一进排期和统计，缺一项就得靠人拍脑袋。

>

> 四眼看下来，你其实在做一次快速评审。判定点不清的，退回去重写；起点含糊的，补前置条件；步骤跳步的，补中间确认；属性缺的，补齐。四眼花不了一分钟，比逐字读省得多，也比凭感觉说一句好像不对有用。示例的用处就在这儿：它把好具体成一个能对照的样本，不是一句评语。照着对四眼，改一条用例用不了十分钟。

>

> 还有一件事想让你留意：一条用例只回答一个问题。期望结果里同时写着"提示错误""账号未锁定""日志有记录"，那其实是三条捆成了一根。捆一起的后果是，只要一项不符整条判失败，你不知道到底哪项坏了。

>

> 拆的时候有个判据：验证点之间能不能各自归因。能，就拆；不能，就留着。

提问：一条用例的期望结果里塞了三个验证点，会造成什么具体麻烦？

> **预设回答：**

- 答"判失败时不知道哪一项坏了"——正中要害。定位成本上去，返工时间白花。

- 答"没关系，反正都要验"——错答。捆一起，失败率会虚高，统计出来的通过率也失去意义。

- 答"拆成三条，成本不是更高"——成本确实上升，换来的是可定位。工程上按"验证点能否独立可判"来拆，不是一律拆到最细。

- 收口：拆分的目的不是让用例变多，是让失败能归因。

> **易错点 / 考点：** ① 拆分判据是验证点能否独立可判，不是数量越少越好；② 案例题常给一条多验证点用例让你拆，答题时逐点列出各自的期望结果；③ 层次属性上，子用例挂在父用例下，父用例通常不直接执行。

> **过渡：** 会读示例，接下来判断什么叫"好"。

【11.14】良好测试用例的特征（2 分钟）

> 八条特征，我压成三组：抓得到、判得清、管得住。

>

> 第一组，抓缺陷的能力。可以最大程度地找出软件隐藏的缺陷，可以最高效率地找出缺陷，可以最大程度地满足测试覆盖要求。这三条要连起来看：既要有深度，藏得深的也能挖出来；也要有性价比，不能一条用例跑一天才挖出一个。覆盖是第 3 章讲过的概念，这里的要求是该覆盖的都覆盖到，不多不少。

>

> 第二组，可判读。既不过分复杂，也不能过分简单。太复杂，执行容易出错、维护成本高；太简单，验证不到东西，等于没测。缺陷的表现要能清楚判定，露出来时你一眼知道错在哪，不是"页面好像有点不对"。待查的输出结果或文件要尽量简单明了——一条用例的期望结果是一份几百行的报表，你根本没法逐项核对，改成只核关键字段。

>

> 第三组，可维护。不含重复的用例，内容清晰、格式一致、分类组织。重复用例是维护灾难，改一处要改十处，还容易改漏；格式一致让评审变快，分类组织让检索变快。

>

> 有个误区得点一下：好用例不是覆盖分支最多那条。分支多不是标准，能明确判定、能稳定复现才是。

>

> 八条之间还有轻重。抓缺陷和可判读是硬要求，缺了这条用例基本白写；可维护那四条是长期要求，缺了它，

> 库会越用越沉。短期看前五条，长期看后三条。多选题爱考这两端：问哪些属于抓缺陷的能力，答成可维护就算错位。八条背不全不要紧，三组别混。

提问：为什么"重复的测试用例"被单独列成一条要避免的特征？

预设回答：

- 答"重复浪费时间"——只说对一半。更贵的是维护成本：规则一变，同一套逻辑要在多条里改，漏一条就留下一条会误报的用例。

- 答"重复可以多测一遍，更保险"——错答。要多测的是不同路径和不同数据组合，机械重复不增加覆盖，只增加噪音。

- 答"重复说明设计时没查重"——很到位。查重该发生在设计阶段，靠库检索和评审，不是执行完才发现。

- 收口：重复用例最大的危害不是多跑几遍，是让通过率这个数字变得不可信。

易错点 / 考点：① 八条要能分三组复述；② "既不过分复杂、也不能过分简单"最常被考，要能说出两端各有什么代价；③ 判断陷阱："输出结果越详细越好"——错，课件要求尽量简单明了，判读成本也是成本。

过渡：八条是"好"的标准，设计时脑子还要盯几件事。

【11.15】11.1.3 测试用例设计考虑因素（2 分钟）

设计的时候，脑子要同时转五件事。转漏一件，用例就偏。

>

第一件，具有代表性、典型性。用例不是越多越好，是要能代表一类情况。测了密码 6 位和密码 20 位，长度这一维的两端就代表了；再测 7 位、8 位，边际收益很低。这条其实在替你做减法，用最少的用例代表最大的输入空间。

>

第二件，寻求系统设计、功能设计的弱点。这条最考验功力。弱点在哪？在状态切换、在并发、在权限交叉、在异常回滚、在边界条件。用例不能只顺正常流程走一遍，得专门往设计者最没把握的地方捅。你可以自问一句：我是开发者，我最怕别人测哪里？

>

> 第三件，既考虑正确的输入，也考虑错误的或者异常的输入。课件把它单列，因为初学者几乎都偏科，只测对的。真实世界里，用户输错的频率远高于输对，系统在错误输入下的表现才是它的真实质量。

>

> 第四件，分析这些错误或异常怎么才能发生。断网、超时、重复提交、越权访问、磁盘满、时间跨零点。异常不是等来的，是造出来的。

>

> 第五件，考虑用户实际的使用场景。真实用户不会按你的步骤走，他会中途退出、会连点、会开两个窗口、会在弱网下操作。

>

> 五件事里最难的是第二件。前四件都有清单可对，第二件要靠你对系统的理解：哪里写得含糊，哪里临时补了个判断，哪里没人敢动。这些不在文档里，在你读代码和问开发的过程里。

> **提问：**要给"用户登录"设计异常用例，你能造出哪几种异常？至少说四种。

> **预设回答：**

- 答"密码错、账号不存在"——只覆盖了数据异常这一维，还不够。

- 答"连错五次锁定、重复提交、断网中途提交、验证码过期"——这就到状态异常和时序异常了，质量明显更高。

- 答"异常就是输入非法数据"（典型错答）——异常包括数据、状态、时序、环境四类，只测数据会漏一大半。

- 收口：造异常的能力，直接等于你挖缺陷的能力。

> **易错点 / 考点：**① 五条考虑因素与 11.1.4 的原则别混，前者是想什么，后者是不许怎么写；② 异常四类（数据、状态、时序、环境）建议背下来，案例题按类展开；③ 判断陷阱："用例只需覆盖正常业务流"——错，这是初学者最常见的错。

> **过渡：**内容之外，还有一批属性要填，这是进管理体系的门票。

【11.16】测试用例其它重要属性（2 分钟）

> 用例还有一批字段要填。填不对，一次排期就能乱掉。

>

> 先说测试目的和测试类型。目的回答这条用例为谁服务，类型回答它属于哪一类测试——功能、性能、易用性、兼容性、安全性。为什么必须标？不同类型的执行条件完全不同：功能用例在普通环境跑，性能用例要加压，兼容性用例要换平台。类型不标，排期就错配。

>

> 再说优先级，通常分高、中、低。优先级的真实用途是时间不够时决定放弃谁。项目延期是常态，延期一定要砍用例，砍哪些？按优先级砍。所以它不能凭感觉填，得有判据：核心业务、高频路径、不可恢复的损失，这些算高；边缘功能、提示文案，能降到中低。

>

> 第三是层次，父、子、孙这样的树。层次解决两件事，组织和复用。一个父用例统摄一组子用例，执行时按树遍历，统计时按树汇总。它还决定用例在套件里挂在哪。

>

> 第四是估计用时。一条大概跑多久，五分钟还是半小时。用途有两个：一是排期，你能算出这轮回归要多少人天；二是探异常，某条估 5 分钟实际跑了 40 分钟，要么用例写得不清楚，要么系统真的变慢了，两种都值得查。

>

> 四类属性填全了，这批用例才谈得上被工具处理。排期工具按估计用时算人天，看板按优先级排序，报表按层次汇总。少一个字段，工具就替你猜，猜出来的数没人敢拿去汇报。估计用时这项还常被当成形式，其实它是最早暴露用例写不清楚的地方。写不出时间，多半是步骤没想清。

提问：项目只剩三天，还有 400 条用例没跑，你怎么决定跑哪些？

预设回答：

- 答"随机挑，反正跑不完"（典型错答）——最糟的答案，等于把风险交给运气。

- 答"按优先级跑高的"——方向对。还得补两步：高优先级里按本次改动影响到的模块再筛一遍，最后按历史缺陷密度排序。

- 答"全都跑，加班"——短期可行，长期会把团队拖垮。疲劳执行带来误判，跑了等于没跑。

- 收口：优先级要在设计阶段就填，因为砍用例的决定往往只有十分钟，那时没人有空重新评估。

> **易错点 / 考点：** ① 四类属性要能列举；② 优先级的判据（核心业务、高频、不可恢复的损失）是简答高频点；③ 判断陷阱："优先级由测试人员执行时临时决定"——错，应在设计阶段确定，并由评审确认。

> **过渡：** 属性是纸上填的，看两份填全的示例，再比差别。

【11.17】示例一（1.5 分钟）

> 示例要当一组看，别当一条看。

>

> 一条用例只能告诉你格式长什么样，一组用例才能告诉你设计思路长什么样。同一功能的一组摆在一起，你要看的是它们之间的差异：哪几个字段变了，哪几个没变。变的那几处，就是这个功能的关键维度；没变的那些，是可以抽出来公用的骨架。

>

> 比如一组输入校验的用例，往往只有测试数据在变，前置条件、步骤、期望结果的结构基本不动。这说明什么？说明这个功能的测试就是同一动作、不同数据。看清这一点，你才知道该用数据驱动去组织它，而不是复制十遍、每遍改一个数。

>

> 第二件要看的是属性列怎么填。优先级怎么分档、层次怎么挂、估计用时怎么写。这些是格式之外的经验，教材讲原理，示例给示范。

>

> 还有一层：一组用例的相似度，能反过来检验设计。相似度高的那几条，本来能合成一条加一张数据表；差异特别大的那几条，往往说明这个功能里有几个互不相干的分支，得分开对待。看完差异再看数量：一组十来条，通常比散着写三十条好维护。示例不是让你抄，是让你找到能抄的骨架。

>

> 第三件是个小练习：把示例里的业务名词换成你课程设计里的模块，看哪些字段能直接沿用、哪些必须重写。能沿用的部分，就是你以后的模板骨架；必须重写的部分，就是这个模块的特殊性。做完这个替换你会发现，

> 写用例不是从零开始想，是填空加判断。

> **提问：**一组用例里只有"测试数据"这一列在变，其余几乎一样。这说明什么？该怎么优化？

> **预设回答：**

- 答"说明用例重复了，应该合并"——方向对。注意别把判定合掉：合并的前提是期望结果的结构相同，不是内容相同。

- 答"说明可以做成数据驱动"——正中要害。骨架一条，数据一张表，执行时逐行喂进去，维护成本立刻降下来。

- 答"说明这些用例太简单，没有价值"——错答。简单不等于没价值，输入校验本来就是一个动作配大量数据，价值在覆盖面。

- 收口：看见只有数据在变，就想数据驱动；看见只有步骤在变，就想参数化流程。

> **易错点 / 考点：**① 示例的用法是看差异、看属性、做替换，不是背；② 数据驱动的前提是骨架一致、判定结构一致；③ 判断陷阱："用例之间越不像越好"——错，同一功能的用例应当高度同构，同构才好维护。

> **过渡：**示例一是一组，示例二再给一组不同形态的。

【11.18】示例二（1.5 分钟）

> 两组示例摆一起，差别比单看一组更清楚。重点在对照。

>

> 第一个对照点是判定点的数量。示例二里如果一条仍然只对一个验证点，说明作者守住了"一个用例一个验证点"。要是出现一条挂多个期望结果，你就得判断它该不该拆。判据还是那句：验证点之间能不能各自归因。

>

> 第二个对照点是前置条件的写法。有的示例写成业务状态，比如"用户已完成实名认证"；有的写成数据状态，比如"库里存在一条已支付订单"。前者好读，后者好造。工程上通常两种都写：先用业务语言说清，再用数据语言落实。这也是评审时最容易被追问的地方。

>

> 第三个对照点是步骤的粒度。写到"点击提交"就够，还是要写到"在弹窗里点确定，等列表刷新"？判据是执行者会不会在这里产生歧义。会，就写细；不会，别啰嗦。这也是后面讲"避免冗长"的实操边界。

>

> 三个对照点合起来，其实在问一件事：这两组用例，作者是不是用同一套判断写出来的。同一套判断，字段之间能对上；两套判断混着写，读的人得先猜他当时怎么想。对照的时候别只挑毛病。作者守住的那几条，往往正是你以后要照做的部分。两组摆一起，也能看出教材想让你往哪个方向写。

>

> 最后提一句：示例不是模板的终点。教材附录 C 给了更完整的字段，示例给的是常用字段的填法示范。字段取全集，填法照示例。

提问：前置条件写"用户已登录"够不够？为什么？

> **预设回答：**

- 答"够了，登录了就能测"——不够。登录的是哪个角色？有没有未完成的订单？账号状态正常吗？这些都会改结果。

- 答"不够，还缺角色和账号状态"——正确方向。完整写法是谁、什么状态、已有数据。

- 答"前置条件写越细越好"（典型错答）——另一个极端。写细要付执行成本，判据是判定是否依赖它，不依赖的别写。

- 收口：前置条件不是背景介绍，是判定所依赖的初始状态，只写依赖的那部分。

易错点 / 考点：① 前置条件三要素：身份、状态、已有数据；② 步骤粒度的判据是会不会产生歧义，不是越细越好；③ 案例题常问"这两条用例的差异说明了什么设计问题"，答题要落到是否同构、是否可复用。

> **过渡：**示例看完，收回来看 11.1 最后一个小节，三条红线。

【11.19】11.1.4 测试用例设计的基本原则（2 分钟）

> 三条红线，碰了就是不合格用例，没有例外。

>

> 第一条，避免含糊的测试用例。课件括号里那句英文很直白：Pass/Failed is clear。通过还是不通过，必须一眼能判。怎么做？课件点了两个抓手：操作环境写清，操作步骤写清。含糊通常就出在这两处。环境没交代，变成"我这能过、你那不过"；步骤没交代，变成"我点这个按钮、你点那个"。养成的习惯是：每写完一条，回头问自己有没有第二种理解方式，有就补。

>

> 第二条，把功能相似的用例抽象并归类，落到做法上就是数据驱动。同一套步骤、不同数据的一批用例，不该复制粘贴十份，该抽成一个骨架加一张数据表。抽象这一步看着费事，收益在每次回归里：改一次骨架，十条全跟着改。

>

> 数据驱动还有一层好处，是复现快。改一个数重跑一行，比打开一条用例改三步再跑，快得多。回归次数一多，这点差别会拉开。抽象归类这一步，难看的是头一次，好用的是后面每一次。考选择题时，7 步这个数和"一个验证点"一起出现，基本就是标准答案。

>

> 第三条，避免冗长和复杂。课件给了两个量化口径：操作步骤不超过 7 步，一个用例一个验证点。为什么是 7 步？步骤一多，执行者的注意力会衰减，漏做、做错、跳过中间确认的概率直线上升。为什么一个验证点？多验证点捆一起，失败没法归因。

>

> 三条治的是同一种病：含糊带来的不可控。压成一句就是说得准、归得类、留得短。

提问：一条 15 步的用例，除了执行慢，还有什么更严重的后果？

预设回答：

- 答"容易漏步"——对。而且漏了往往不自知，最后判"通过"，其实根本没跑到判定点。

- 答"失败了不知道错在哪一步"——更严重。步骤越长，定位成本越高，返工时间被成倍放大。

- 答"没关系，写详细点总没错"（典型错答）——冗长本身就是缺陷来源。步骤多了，环境依赖多了，用例的稳定性就下来了。

- 收口：步骤长度不是文风问题，是可靠性问题。超过 7 步就该考虑拆。

易错点 / 考点： ① 三条原则要能复述，尤其"步骤不超过 7 步""一个用例一个验证点"这两个量化口径，选择题高频；② 数据驱动是抽象归类的落地做法，别答成两件事；③ 判断陷阱："用例写得越细越好"——错，细到产生维护负担就是冗长。

【思政落点 2】：写用例最见功夫的地方不是工具会不会用，是表达够不够严谨。一句"输入非法密码"看着省事，等于把判断责任推给下一个执行的人；写成"输入 65 位密码，上限 64 位"，责任才收回自己身上。工程文书里抠字眼不是小气，是对同事的时间负责，对结论的可信度负责。

过渡：原则是底线，往上还有一层质量要求。

【11.20】单个测试用例的质量要求（2 分钟）

新人拿到你的用例，不问你一句，能不能跑完并给出结论？这一问就是这一段的验收线。

>

> 课件给七条。第一条，具有可操作性。执行者拿到它能不能直接动手，需不需要再问作者。需要问，就说明不可操作。

>

> 第二条，具备所需的各项信息。账号、数据、地址、权限、依赖服务，一样不能少。信息不全的用例，执行时变成边测边猜，猜出来的结果没有复现价值。

>

> 第三条，各项信息描述准确、清楚。准确指不含歧义，清楚指结构分明。这两点常常一起失守：一句长句塞了三个条件，读的人要拆半天。

>

> 第四条，测试目标针对性强。一条用例只打一个点，别顺手把别的东西也验一下。针对性强的用例，失败时给出的信息量最大。

>

> 第五条，验证点完备，而且没有太多的验证点。这句是双向的：漏了叫不完备，多了叫冗余。完备是对这条用例的目标而言，不是对整个功能而言。

>

> 第六条，没有太多的操作步骤，还是那个 7 步的口径。

>

> 第七条，符合正常业务惯例。这条最容易被忽略。操作顺序要符合真实业务逻辑，不能为了测而造出现实里不可能出现的操作序列，那样测出的缺陷没有修复价值。

>

> 七条里有三条是硬指标：信息齐、描述清、步骤少。另外四条靠判断力：可操作、目标准、验证点不多不少、场景可信。硬指标模板能兜住，判断力只能一条条写出来。

>

> 七条压成一句：可操作性是唯一的验收标准。别的做得好不好，都用它来检验。

> **提问：**"符合正常业务惯例"这条，为什么也算质量要求？不符合会怎样？

> **预设回答：**

- 答"不符合就是瞎测"——话糙理对。造出真实业务里不会出现的操作序列，测出的"缺陷"开发可能直接判无效。

- 答"按惯例走会漏掉异常"——这个反问好。两者不冲突：异常用例也要在合理的业务前提下构造。"重复提交"是真实的，"先退款再下单再删账号"这种顺序往往不是。

- 答"惯例是业务定的，测试不该管"——错答。用例的业务合理性正是测试人员要守的，否则缺陷单会被大量打回。

- 收口：用例要有业务可信度。不可信的场景，即使测出问题，也换不来一次修复。

> **易错点 / 考点：**① 七条要能列举，最后两条（步骤不多、符合业务惯例）常被合起来考；② "验证点完备而且没有太多"这种双向表述常出判断题；③ 判断陷阱："信息越多越好"——错，是所需信息齐备，多余的描述反而干扰判读。

> **过渡：**单条合格了，接下来是一个绕不开的取舍——一条用例该写多细？

【11.21】测试用例的颗粒度（2 分钟）

同一个登录功能，有人写了三条用例，有人写了三十条。谁对？

>

先说结论：看你要拿它干什么。课件给的例子很朴素——输入不同的用户名和口令，结果要满足设定的要求。粗的那种，一条用例塞五组数据、五个结果。条数少、跑得快，数据变了改一处，维护便宜。坏处也实在：第三组不对，整条判失败，你得手工再跑一遍，才知道坏在哪一组。执行的人自由度太大，还能自己解释结果，跑偏了也看不出来。

>

细的那种，一组数据一个结果。用户名对、口令错，期望提示"密码错误"。判定唯一，失败能归因，统计也准。麻烦在数量上：五组变五条，五十组变五十条，库很快就胀起来。条数一多，一次回归的执行窗口装不下，你就得开始挑着跑，挑的过程又容易漏。

>

那怎么选？三步。第一步，看判定能不能独立。多组数据的期望结构一样、只是取值不同，可以粗。有的成功、有的报错、有的账户锁定，必须细。第二步，看执行成本。要造历史数据才跑得动的，尽量并成一条；界面上点几下就完的，拆开更省事。第三步，看以后还要不要单独挑出来跑回归。要，就拆细，细的才能被单独调度。三条里最硬的是第一条。判定不独立，细也细不到点子上。

>

工程里用得最多的是中间路线：骨架粗、数据细。一条用例配一张数据表，判定写成每行数据对各自的期望结果。这就是数据驱动。维护的便宜和定位的准确，两边都拿到。

提问：一条用例里放五组数据，执行到第三组失败了，你接下来做什么？

预设回答：

- 答"提缺陷"——太快。你还没确认是哪一组、错在哪个字段，缺陷单会被打回。

- 答"手工重跑五组，逐组确认"——对。这就是粗颗粒度的隐性成本，一次失败要再付一次定位的钱。

- 答"整条判失败就行了"——错答。四组明明是对的，也被记成失败，通过率立刻失真。

- 收口：颗粒度是在维护成本和定位成本之间，选一个你更能承受的。

易错点 / 考点： ① 粗与细的利弊要成对答，只答一头不得分；② 三条判断依据要背：判定独立性、执行成本、复用需求；③ 陷阱题"颗粒度越细质量越高"——错，过细会让数量失控，课件在整体质量要求里写的是粒度适中。

过渡： 单条看完了，把目光抬到一整套库，还有四项要求。

【11.22】整体测试用例的质量要求（2 分钟）

想象你接手一个两千条的用例库，打开第一屏，名字全叫"用例 1""用例 2"。这种库的毛病不在条数，在质量。整体质量看四项。

>

第一项，覆盖率：依据特定的测试目标，尽可能覆盖所有的测试范围、功能特性和代码。这里有半句限定别读漏——依据特定的测试目标。目标是把支付主流程测透，和目标是测全所有异常分支，算法完全不是一回事。所以问覆盖率之前，先问一句"对什么目标"。

>

第二项，易用性：设计思路清晰、组织结构层次合理、操作连贯、执行顺畅。判断办法很土但管用：找个没参与设计的人，让他按库里的组织找三条用例跑一遍。他卡在哪，易用性的问题就在哪。找不到入口，多半是树和命名的问题，不是他不熟。

>

第三项，易维护性：用较少的时间完成用例的维护，包括易读性、一致性。这一项不产生即时收益，只在需求变更时一次性爆发。好的库改一条需求动三五条，差的库要翻两百条，还得担心改漏。

>

第四项，粒度适中。既能覆盖各个特定场景、保证覆盖率，又能处理好不同的测试数据、测试条件。这就是数据驱动，可维护性跟着上来。这一项是前三项的平衡点：太粗覆盖不住场景，太细维护不起。

>

四项连起来看：覆盖率管够不够，易用性管顺不顺，易维护性管养不养得起，粒度管平不平衡。一个库四项都立得住，才算真的可用。课程设计交库时，这四项都能让人当场验一遍。

> **提问：**一个库覆盖率很高，可没人愿意用。这算合格吗？

> **预设回答：**

- 答"算，覆盖率高就是好"——错答。没人用的库，覆盖率只是纸面数字，执行时大家还是凭经验。

- 答"不算，易用性是硬要求"——对。库是给团队用的工具，不是给自己看的作品。

- 答"说明覆盖率算错了"——不一定。更常见的原因是组织混乱、命名随意、找不到入口。

- 收口：四项里塌一项，另外三项的价值都会被拖下水。

> **易错点 / 考点：**① 四项名称与各自判据；② 覆盖率要强调"相对测试目标"，这是简答的加分点；③ 陷阱题"覆盖率是唯一硬指标"——错，课件把易用性、易维护性、粒度并列。

> **过渡：**一条怎么写讲过了，接下来是一堆怎么管。

【11.23】11.2 测试用例的组织和跟踪（1 分钟）

> 从这里开始，问题换了。不只是这条写得对不对，还要问这一堆放在哪、怎么管、跑没跑完。

>

> 举个具体的场面。三个人一起做课程设计，各写各的用例。回归那天你问一句"支付这块跑过没有"，三个人答得都不一样。不是谁偷懒，是这些用例没人给它们挂过标签，也没人记过跑没跑。

>

> 这一节五个小节，顺序有讲究。第一站，测试用例的属性。给用例挂上能管的标签：目标性、所属范围、关联性、阶段性、时效性。属性不全，后面套件分不出来，跟踪也统计不出维度。第二站，测试套件及其构成方法。把用例按目标组合成套件，回答这一轮要跑哪些。套件是这一节的主干，后面六个场合都从它长出来。第三站，跟踪测试用例。执行过程里盯进度：跑了多少、通过多少、卡住多少、为什么卡。第四站，维护测试用例。需求变了、版本换了，用例跟着走。这一站决定整套东西能不能活过三年，也是大多数团队实际塌方的地方。版本一多，用例和需求就脱钩，脱钩的库没人敢信。第五站，测试用例的覆盖率。用三类覆盖率检验这套东西够不够，回答测够了没有。

>

> 很多人第一反应是：管用例是测试经理的活。不对。属性是写用例的人填的，跟踪的数据源也是执行的人填

> 的。经理手里那张表能不能切，全看每个人填得规矩不规矩。

>

> 五个词串成一条线：贴标签、组套件、盯执行、跟着改、量出来。这就是一个用例库的完整生命周期。缺一站，链条还在转，转到某一环就断：属性不全，套件分不出来；套件不切，进度统计不出维度；不维护，过两个版本整库没人敢用。你现在用表格也能搭这五步。先把属性列全，再谈树怎么画，别反过来。

> **提问：**这五站里，哪一站最容易被跳过？跳过的后果是什么？

> **预设回答：**

- 答"属性，大家嫌填表麻烦"——很常见。属性一缺，套件只能按目录硬分，跟踪时统计不出维度。

- 答"维护，版本一多就没人管"——这才是最普遍的塌方点，也是后面要重点讲的一站。

- 答"覆盖率，反正也测不完"——错答。跳过它，你就回答不了"测够了没有"，验收只能靠感觉。

- 收口：五站是链条，跳一站链条还在转，但转到某一环就会断。

> **易错点 / 考点：**① 五个小节的名称与顺序；② 属性是组织的前提，常以"为什么用例需要属性"来问；③ 陷阱题"跟踪和维护是测试经理的事"——错，执行者填的状态就是跟踪的数据源。

> **过渡：**从第一站开始，先看用例要挂哪些标签。

【11.24】11.2.1 测试用例的属性（1.5 分钟）

> 几千条用例平铺在一张表里，会变成一堆沙子。属性就是给每粒沙子贴标签，让它能被抓起来。

>

> 管理属性解决三类问题。第一类，分类：这条用例属于哪个模块、哪个功能、哪一类测试。没有分类，几千条就是散的，谁也不敢删，谁也不敢改。第二类，筛选：这一轮回归要跑哪些？答案只能来自属性筛选，按模块、按优先级、按类型、按阶段。填得越细，筛得越准。填得不对，筛出来的是错的那批。第三类，关联：这条用例对应哪条需求、哪个设计、哪个历史缺陷。关联立起来，追溯才成立，改动影响分析才有依据。

>

> 关联这一项举个例。你这条用例对应需求单 3.2 里的退款校验。三个月后 3.2 改了，退款要往下传三笔。你

> 靠什么找到要改的用例？只能靠这条关联。

>

> 那属性要不要加？有个很实用的判据：你将来会不会用它来筛选或统计。会，就必须加，而且值域要规范。优先级只填高、中、低，不能一会儿"高"、一会儿"紧急"、一会儿"P1"。同一件事三种写法，等于没填。不会，就别加。字段越多，填得越糊，最后整张表没人维护。一条用例挂十个字段、八个空着，比挂四个字段全填满更糟。

>

> 有人把属性当备注栏，写一句"这条挺重要，别忘了"。工具读不懂这句话，筛选也筛不出来。要写就写成能被算的值：优先级高，模块是支付，阶段是系统测试。

>

> 前面讲元素，是站在一条用例内部看它由什么组成；这里站在管理的位置上，看它要挂什么标签。角度不同，目的也不同。填之前先想清楚谁来读这一行：作者、执行的人，还是做统计的人。想清楚之后，字段数就定了，不用再开会争。

提问：为什么属性的值域必须统一，比如优先级只能填高、中、低？

预设回答：

- 答"为了好看"——不是。为了能被机器统计：高、紧急、P1 混着填，筛选和汇总全废。

- 答"为了统一口径"——对。属性是团队共用的语言，语言不统一，沟通成本成倍往上翻。

- 答"随便填，反正自己看得懂"——错答。属性是给别人和给工具看的，自己看得懂没有意义。

- 收口：属性不是备注栏，是能被算的字段；算不出来的属性，等于没填。

易错点 / 考点：① 三类用途（分类、筛选、关联）是理解这一节的关键；② 值域规范化是简答常考点；③ 陷阱题"属性越多越专业"——错，属性要服务于筛选与统计，否则是负担。

过渡：结构立住，再逐个看最要紧的五个属性。

【11.25】一些属性说明（2 分钟）

> 五个属性里，有一个最像废话，也最狠——时效性。先把五个过一遍。

>

> 目标性，包括功能性、性能、容错性、数据迁移各个方面。它回答的是这条用例验证哪一类质量特性。为什么重要？不同目标的用例，执行条件差得远。功能用例在普通环境点一点，性能用例要加压要监控，容错用例要人为造故障，数据迁移要准备两套库。标不清，排期会算错，环境会白搭。

>

> 所属的范围，就是属于哪一个组件或模块。这一项最常用：组套件靠它，出了缺陷划范围也靠它。一看这个模块有多少条用例、跑了多少条，就知道这块测得够不够。五个里前四个管怎么找、怎么跑，最后一个管还能不能用。

>

> 关联性，和软件产品特性相联系。关联立起来，某个特性要改，受影响的用例一查就出来，不必靠人回忆。

>

> 阶段性，属于单元测试、集成测试、系统测试、验收测试里的某一个。标清阶段，能躲开一种常见混乱：拿系统测试的用例去单元阶段跑，跑不起来，还以为是环境问题。不同阶段的用例，关注点、环境、执行的人都不同。

>

> 最后是时效性：不同版本适用的用例可能不相同。这条承认一件很多人不愿承认的事——用例有寿命。某个版本删掉的功能，它的用例要下线；新增的开关，要有新用例。不标会怎样？执行的人拿过期用例去跑，跑不过就报缺陷，查半天发现功能早删了。这五条不是表格里凑数的列。缺一条，排期或回归就会错一次。

> **提问：**"时效性"为什么必须作为属性存在？不标会出什么问题？

> **预设回答：**

- 答"不标就不知道这条还能不能用"——正中要害。执行者会拿过期用例去跑，跑不过就报缺陷。

- 答"版本多了自然乱，标了也没用"——错答。正因为版本会多才要标；不标是听任它烂。

- 答"删掉旧用例就行，不用标"——太粗暴。有些功能暂时下线、下版回来，删了得重写。

- 收口：时效性是库的保质期标签，没有它，你判断不了哪些还能用。

> **易错点 / 考点：** ① 五个属性名称与含义要能对上，阶段性和时效性最容易混；② 阶段性答四个级别（单元、集成、系统、验收）；③ 陷阱题"一条用例可以同时属于多个阶段"——主要适用阶段应唯一，跨阶段复用要重新确认判定。

> **过渡：** 属性齐了，把它们组合起来就是套件。

【11.26】11.2.2 测试套件及其构成方法（2 分钟）

> 先报个数：一个中型项目，几千条用例是常态，上万条也不稀奇。几千条平铺着，你回答不了这一轮跑哪些，只能靠搜索和记忆。搜一次漏一条，漏的那条往往就是出问题的那条。三千条对一个人是查不完的，对一棵分好的树是查得完的。

>

> 课件给的理由有三层。第一层是"众多"：数量摆在这儿，不组织就没法用。第二层是"可扩展"：框架不是为今天这一版画的，是为第一百版画的。判一个组织方式好不好，有个很简单的测试——新增一个模块，你要改多少东西？只加一个分支、别处不动，说明撑得住；每加一块都要重排整棵树，它活不过三个版本。第三个版本之后，没人敢动这棵树。第三层是"层次结构"：项目、模块、功能、用例，逐层收敛，把多变成能管。层次还有个顺带的好处，统计能按层汇总。想知道支付模块的通过率，按层聚合就行，不用手工筛。按人分也省事，谁负责哪块一目了然。

>

> 可扩展这条可以当练习做。拿自己组的用例树，假装明天要加一个"优惠券"模块。数一数改了几处：只加一个节点、别处没动，这棵树撑得住；要重排分支、改一串命名，它撑不过下一版。

>

> 课件那个等式也点一下：Test Case → Test Suite。用例是叶子，套件是枝干。这说明套件不是新东西，是用例的组合方式；也说明套件能嵌套，套件里还可以有子套件。叶子挂在枝上，枝挂在干上，谁是父谁是子，一眼看得清。

>

> 还有一层要算清。库的成本大头不在第一次编写，在第 N 次改动。框架好，第 N 次只是加个节点、改个字段。框架差，第 N 次要重排整棵树，还得逐条核着改。所以判框架，判的是将来改动的代价。

> **提问：**判断一个用例组织框架好不好，你会用什么方法检验？

> **预设回答：**

- 答"看树画得漂不漂亮"——错答。漂亮不等于好用，很多漂亮的树是按组织架构画的，换个人就全乱。

- 答"看新增一个模块要改多少地方"——好判据，这就是可扩展性的实操检验。

- 答"看能不能一眼找到某条用例"——也对，这是检索性；但检索靠属性和命名，不完全靠层级。

- 收口：两条硬指标，加东西不用重排，找东西不用翻遍。

> **易错点 / 考点：**① 套件的三条存在理由（数量、可扩展、层次）；② Test Case → Test Suite 的从属关系常出填空；③ 陷阱题"套件只能有一层"——错，套件可以嵌套。

> **过渡：**层次到底怎么画，看一个具体的组织示例。

【11.27】示例（1.5 分钟）

> 打开一个用例库，先看它的树，基本能判断这库还能活多久。看三件事。

>

> 第一件，层次按什么分。常见是按项目、模块、功能、用例四级往下走，也有按版本、测试类型、模块走的。两种都能用，判据只有一句：你最常按哪个维度问"跑哪些"，那个维度就放上层。反过来，按组织架构画树最省事，谁负责谁一块，看着整齐。可换个人接手，整棵树就没人认。

>

> 举个对比。一个库打开，一级是登录、下单、支付，二级是正常、异常、边界，新人不用问人就知道往哪放。换个库，一级是"我的""老张的""待整理"，这棵树得重画。

>

> 第二件，节点怎么命名。名字要能自解释，不许出现"杂项""临时""待分"这种垃圾桶节点。垃圾桶是所有库烂掉的起点：一旦有了它，不好归类的全往里扔，半年后没人敢打开。套件名带上版本和目标，节点名带上

> 模块，光这两条就省掉一半问人的时间。

>

> 第三件，叶子挂在哪。一条用例只给一个主归属位置，别的维度用属性表达。为什么要这么死？多处挂载会让人把同一条改两次、统计两次，数字立刻失真。失真一次，后面所有结论都不作数。多挂一处，两边的数字对不上，对不上的数字没人敢用。

>

> 主归属举个例。"支付超时"这条，主归属挂支付模块，属性里标一次"涉及订单模块"。订单改动时属性一筛能看见它，统计支付模块也不会把它算两次。

>

> 树和属性还是搭配关系：树回答我在哪，属性回答我是什么。只用树，筛选维度不够；只用属性，人找不到入口。课程设计里建议先画这棵树，再写用例。顺序反了，你会写完两百条才想起没分类，然后被迫大改。

> **提问：**为什么"杂项"这样的节点是危险的？

> **预设回答：**

- 答"不好看"——不是重点。重点是它会持续吸收一切不好归类的东西，变成整个库的沉积层。

- 答"以后找东西找不到"——对。更糟的是它让分类失去约束：既然有"杂项"，设计时就不必想清楚。

- 答"先扔进去，以后再说"——这就是多数烂库的形成过程，"以后"从来不会来。

- 收口：宁可多建一个正经节点，也不要留一个垃圾桶。

> **易错点 / 考点：**① 层次划分依据是"最常用的筛选维度"，这句可以直接当简答答案；② 一个主归属加属性表达别的维度，是避免重复统计的关键；③ 陷阱题"用例可以同时挂在多个节点下"——错，会造成维护与统计的双重失真。

> **过渡：**结构看完了，回到定义，套件到底是什么。

【11.28】测试集/测试套件（2 分钟）

> 把套件当成"用例的文件夹"，是常见的一处偏差。定义里有两个成分：一系列测试用例，以及与之关联的测

试环境。很多同学只记住前半句。同一批用例换个环境跑，本来就是两件事。环境不写进套件，结论就复现不了。环境要写到能复现的程度：数据库版本、中间件、账号权限，缺一样结论就不算数。

>

环境那一半最容易被省。同一条下单用例，在测试库和预发库跑，库存数不一样，支付通道也不一样，结论自然不同。套件里不写清哪个环境，明天换个人跑，两边对不上。

>

定义后半句更硬：把服务于同一个测试目标、特定测试任务或某一运行环境下的一系列用例组合起来。三个"同一"就是边界判据。一条用例和这个套件的目标不同，就不能进来，哪怕它属于同一个模块。

>

构成方法三种。按程序功能模块组织，登录一套、下单一套、支付一套。直观、好找、和开发的分工对得上；跨模块的场景不好归，比如下单后支付失败再退款。按类型组织，功能、性能、兼容性、安全各一套。同类执行条件一致，便于批量备环境；按业务切块交付时不好凑。按优先级组织，高的一套、中低各一套。时间不够时最管用，先跑高的那套，跑完就有个能交出去的结论。三种切法都能用，选哪个取决于这一轮要回答什么问题。

>

三种不互斥，实际都是组合：模块建大树，模块内按优先级切，跨模块场景单独建类型套件。这也解释了属性为什么必须填全，三种切法全靠属性筛。切法换了，套件名字也得换，否则执行的人会按老名字找。名字和切法对齐，交接才不靠口头解释。

提问：按功能模块组织的套件，遇到跨模块的业务场景，怎么处理？

预设回答：

- 答"塞进最相关的那个模块"——可行，但要标出关联的另外几个模块，否则影响分析会漏。

- 答"单独建一个跨模块场景套件"——很实用，代价是要维护一条清楚的命名规则。

- 答"跨模块的就不管了"——错答。跨模块场景恰恰是缺陷最密的地方，漏了它等于漏掉最值钱的一块。

- 收口：组织的目标不是每条都有归属，而是每条都能被找到、被执行、被统计。

易错点 / 考点：① 定义里"用例加环境"两个成分都要答到，只答用例会丢分；② 三种组织方式的优缺点要成

对说；③ 陷阱题"三种方式只能选一种"——错，工程里组合使用是常态。

过渡：三种组织落到图上，就是套件和测试计划的关系。

【11.29】测试套件（Test suite）（1.5 分钟）

一个计划配四组套件，还是把四组并成一组？先看层级怎么搭。三层：测试计划定义这一轮要达成什么，向下拆成若干测试目标，每个目标再落地成对应的测试套件。计划这一层要写清范围和时间，不然目标也没法落地。

>

课件列的目标有四个：集成测试目标、新功能测试目标、系统性能测试目标、功能回归测试目标。四个目标，各自一套。为什么这么排？因为套件不是按目录长的，是按目标长的。这一轮想达成什么，决定要组哪套。目标换了，套件就要重组，不是建好一套年年用。目标不写清，套件就退化成按人分、按目录分。

>

目标还会变。这一版只动了登录页的校验，回归目标就缩成登录这一小片。你还抱着上个版本的全量套件跑，一天跑不完，交上去的结论还是"没跑完"。

>

举个具体场面。上线前四个目标摆在一起。集成阶段要验模块之间的接口。新功能阶段要验这版新增的三个功能。性能阶段要验下单链路的承载。回归阶段要验老功能没被改坏。四组套件，执行顺序、环境准备、判定标准都不一样。只有一套全集套件，就只能一股脑全跑，跑完还不知道每个目标达成没有。四个目标达成没达成，交出去的结论才分得清是谁要的。

>

上一层定义要什么，下一层回答怎么给。命名也顺手带上目标，比如"V2.3 回归-支付"。执行的人一看就知道它为谁服务、什么时候跑。名字里带目标，还有个好处：复盘时能看出哪一组一直没跑。四组套件跑完，一张表就能说清这一轮过了几个目标。

>

三层各叫什么叫清楚。计划和目标混着叫，套件就没人归得清，统计也归不了口。

提问：如果只有一个"全集套件"，会带来什么实际问题？

预设回答：

- 答"跑得太久"——是问题，但不是最严重的，跑得久可以分批跑。

- 答"跑完不知道每个目标达成没有"——正中要害。目标是判定单位的来源，没切分，结论交不出去。

- 答"有全集就够了，需要时手工挑"——错答。手工挑意味着每次执行都不一样，回归没法比。

- 收口：套件的价值不在装得多，在切得准；切得准，结论才交得出去。

易错点 / 考点：① 三层结构（计划、目标、套件）与四个目标示例要能复述；② "套件按目标组织"是核心结论；③ 陷阱题"一个计划只能有一个套件"——错，按目标可以有多个。

过渡：目标之外还有一个更细的切法，按测试类型。

【11.30】测试类型与测试用例设计（2 分钟）

十四个测试类型摆在眼前，硬背是会崩的。先分组，再记。

>

第一组管质量特性：功能测试、易用性测试、压力测试、界面测试。它们问的是这个软件在某一类质量上够不够。第二组管环境与配置：配置测试、国际化测试。换一套环境还对不对，是它们的事。第三组管生命周期：安装卸载测试、软件更新测试、联机注册测试。软件从装到卸这条路上，每个动作都算。第四组管配套内容：文档测试、联机帮助测试、文件操作测试、数据备份测试。这组最容易被漏，也最容易出问题——不少差评不是功能错，是帮助文档和实际行为对不上。第五组是回归测试。它不是一个类型，是一种重新执行的组织方式，这个区别要分清。五组记法的好处是漏了哪组一眼能看出来，比按字母顺序背强。

>

文档那组举个例。帮助里写退款三个工作日办完，代码走的是七个工作日。功能全对，用户照样投诉。这种缺陷工具扫不出来，只能有人对着文档点一遍。

>

两种切法各有亏。按类型切，同类执行条件一致，批量备环境省事；同一个模块的用例被拆散在几个类型里，按模块交付时不好凑。按模块切，直观、对得上分工；同一类条件要在多个模块里重复准备，比如国际化环

> 境得备三遍。

>

> 工程上的解法是二维：树按模块长，属性按类型标，需要时筛出一个类型视图。一条用例只有一个主归属，却能出现在多个视图里，既不重复维护，也不丢维度。课程设计里，模块树先搭，类型属性后补，补的时候别只补两种。五组里最常漏的是配套内容和生命周期这两组。

> **提问：**"回归测试"为什么不算一种测试类型，而算一种组织方式?

> **预设回答：**

- 答"因为它就是把老用例再跑一遍"——这就是关键。它不针对某一类质量特性，针对的是改动有没有破坏已有功能。

- 答"回归也是一种类型"——错答。回归用例的内容来自功能、界面、配置各类，它没有独立的判定标准。

- 答"回归是阶段，不是类型"——接近了。更准确说，它是由变更触发的重新执行策略。

- 收口：分类要分清"测什么"和"为什么现在测"，回归属于后者。

> **易错点 / 考点：**① 课件的类型名称要能列举，按五组归类比逐条硬背稳；② 回归的性质是判断题高频点；③ 陷阱题"按类型和按模块只能选一个"——错，用属性做二维视图可以兼顾。

> **过渡：**组织方式定了，还要和测试过程对齐，否则库和流程是两张皮。

【11.31】测试用例的组织和测试过程的关系（1.5 分钟）

> 有些团队用例写得不错，流程也跑得顺，可两边从来不互相喂。库是库，流程是流程，这就是两张皮。五环里任何一环不用库，库就退成一份存档。存档没人查，下一轮还得从头问一遍。

>

> 把测试过程的主干摆出来：计划、设计、执行、缺陷、回归。用例组织要在每一环都用得上。计划环节，库提供分母：这一轮跑多少条、要几个人天，靠套件和估计用时算出来。没有库，计划只能拍脑袋。拍脑袋的排期，最后都靠加班补。设计环节，库提供基线：新功能照着同类老用例的骨架写，省一半时间，还继承历史踩过的坑。老用例的骨架，是上一轮人替你踩出来的路。执行环节，库提供清单：按套件取用例，逐条记状态，进度就是已执行除总数。清单在，谁跑哪条不用现问。缺陷环节，库提供坐标：缺陷单挂上用例编号。

> 一看就知道哪条用例发现的、当时什么条件。这条用例将来就是回归的必跑项。断开的缺陷单，只能靠人回忆当时的操作。回归环节，库提供现成的范围：受影响的模块筛出来，历史出过缺陷的优先跑。范围一筛，回归就不必全量。

>

> 计划那一环举个例。领导问这轮回归要几天，你答"两三天"。有库的人答得出来：回归套件八百条，一人一天六十条，两个人两天。分母是数出来的。

>

> 五环串起来有个规律：用例库在流程里既是输入，也是输出。设计时它输出用例，执行后它吸收结果，缺陷修完又长出新的用例。它不是流程旁边的一份文档，是流程里的一个节点。课程设计里，至少要让人能从一张缺陷单反查到一条用例。

>

> 还有一层：库不是给自己留的备忘。它是流程里每个环节都要伸手去拿的东西，谁都拿得动，它才算活着。

提问：如果缺陷单里不关联用例编号，会损失什么？

预设回答：

- 答"少个链接而已"——损失不小。断开之后，你答不出这条缺陷当时怎么测出来的，也无法精准回归。

- 答"回归时不知道要跑哪条"——对，这是最直接的损失，验证修复只能靠人回忆当时的操作。

- 答"用例库和缺陷库本来就应分开"——错答。分开管可以，关联必须建，否则两套数据互相证明不了。

- 收口：关联不是形式，是把一次偶发变成可复现资产的手段。

易错点 / 考点：① 五个环节与库的对应关系（分母、基线、清单、坐标、必跑项）建议背；② "既是输入也是输出"是这一节的核心判断；③ 陷阱题"用例只在执行阶段有用"——错，计划、设计、缺陷、回归都在用。

过渡：咬合关系讲清了，最后给六个具体场合。

【11.32】测试套件应用场合（2 分钟）

时间永远不够。套件这套东西的用处，就在只给我两小时的时候显出来。六个场合，一条一条说。

>

> 第一，只有部分功能模块变了，就用这些改动模块的用例组一套。这是最日常的用法，前提是用例按模块组织得好，否则挑不出来。第二，改动模块里也不必全跑，按优先级再切一套。改动模块里也有核心路径和边缘路径，先跑高优先级那套，核心风险先兜住。第三，执行第一阶段可以建一个特定平台上的套件。第一阶段通常是冒烟或基础验证，要的就是这台机器上跑不跑得通。第四，自动化测试和手工测试分开建套件。自动化要无人值守、可重复、判定可机读；手工可以有主观判断和探索。混在一起，自动化跑不动，手工也跑不顺。第五，可以按测试人员建，按平台或模块建。人手分工明确的团队里很有效，各人打开自己的套件就知道今天干什么。但它是执行视图，不是资产归属，人走了东西还在库里。第六，回归时先跑曾经发现过缺陷的用例，再跑从没发现过的。历史出过问题的地方，再出的概率明显更高，这是最便宜的风险信号。六条里前三条管范围，第四条管分工，第五条管人，第六条管顺序。

>

> 第四条值得多说一句。自动化那套要能夜里无人值守跑完，判定必须机读，一条用例只许一个结论。手工那套可以让人看着界面说"这个按钮位置有点怪"。两套混在一起，两边都不好用。

>

> 六条合起来看，套件排的是顺序。先跑哪套、后跑哪套，排的都是风险。排完还留一条给新功能，别把时间全给老地方。风险高的先跑，跑完心里才有底。顺序定不下来，两小时只够跑一半。

提问：第六条"先跑出过缺陷的用例"，依据是什么？有没有风险？

预设回答：

- 答"依据是缺陷会扎堆"——对。同一个地方反复出问题，通常是设计复杂或改动频繁。

- 答"没依据，就是习惯"——错答。这是基于风险排序的做法，缺陷历史是最便宜的风险指标。

- 答"风险是可能漏掉新问题"——很好。所以它是先跑，不是只跑；新功能的用例仍然要跑，只是排在后面。

- 收口：套件的排序排的是风险，缺陷历史是最廉价可靠的那个信号。

易错点 / 考点：① 六个场合各说一句，简答按点给分；② 第四条和第六条最实用，答题优先写；③ 陷阱题"按人员建套件就是把用例归个人所有"——错，那是执行分工视图。

过渡：套件解决跑哪些，下一个问题更硬：跑到哪了。

【11.33】11.2.3 跟踪测试用例（2 分钟）

> 交付前三天，负责人问一句测到哪了，听到的回答是差不多了。这句话背后往往什么都没有。

>

> 执行跟踪看四个问句。第一问，跟上进度没有？进度不是感觉，是已执行数除以计划执行数，每天都要更新。等到交付前三天才发现只跑了一半，那时候做什么都晚。更新不及时，曲线就是编的。第二问，一个人每天能执行多少条？这个数有两个用处：排期时算要几个人几天；出异常时当温度计。某人执行量突然腰斩，要么用例写得太粗、跑一条卡一次，要么环境不稳、反复重试。这个数不用精确，但要每天差不多。第三问，通过、未通过、未测试各占多少？三个数必须分开，不能揉成一个完成率。通过是已验证，未通过是有缺陷待修，未测试是风险还在。一个完成 80% 里未通过占了三成，这个 80% 一点都不值得高兴。三个数分开，才知道下一个小时先修缺陷还是先跑用例。第四问，不能执行的原因是什么？常见四类：环境没准备好、前置数据造不出来、相关功能没提测、用例本身已失效。前三类找环境或开发，第四类得改用例。不分类，你只会得到一句跑不动。四类原因分开登记，责任才落得下去。

>

> 课件那条曲线，横轴是时间，纵轴是需要执行的用例数，目标是爬到 100%。真实曲线开头慢、中段快、尾段又慢下来。看曲线看尾段，尾段收不上去，说明阻塞没被及时清掉。曲线画不出来，说明每天没人记状态。

> **提问：**"完成率 80%"，什么情况下不能采信？

> **预设回答：**

- 答"未测试的还很多的时候"——对。分母只算了已排期的用例，实际风险就被低估。

- 答"未通过的占很大比例的时候"——也对。未通过意味着缺陷未修，功能不可用，不能算完成。

- 答"完成率就是完成率"——错答。不区分状态、不统计阻塞原因的完成率，只是自我安慰。

- 收口：进度必须带状态分解才有意义，只有一个总数，等于没有数据。

> **易错点 / 考点：**① 四个问句要能列举，"不能执行的原因"最常被漏；② 三个状态（通过、未通过、未测试）必须分开统计；③ 陷阱题"执行进度就是看已执行数"——错，还要看通过率与阻塞情况。

> **过渡：**跟踪看的是现在，接下来是一件更长远的事。

【11.34】11.2.4 维护测试用例（2 分钟）

> 有一类活，做好了没人看见，做坏了半年后才炸——维护用例。课件把它压成一句：测试用例的维护是持续改进的过程。这句话短，拆开是四件事。四件事缺一件，持续改进就退成一句口号。

>

> 第一件，什么时候要动。触发点五类：需求变更、发现新缺陷、用例评审意见、环境或平台升级、用例本身失效。维护不是有空再做，是有触发就必须动。触发点写下来，才叫机制；记在脑子里，只是习惯。没有触发机制的团队，库是慢慢死掉的：先是几条用例跑不过，没人管；然后一批，开始被跳过；最后整库没人信。反过来，触发一堆、没人认领，库一样会烂。

>

> 触发点举个例。产品说这版加了短信登录。这一句就是触发：登录模块要新增用例，原有用例还得补一条"未绑手机号"的分支。没人接这一句，下版测登录还是老那几条。

>

> 第二件，做什么动作。不只是改。更新，判定或数据变了；新增，新需求和新缺陷回填；废弃，功能删了，下线但不删记录；归档，旧版本留档备查。废弃和归档要分开。直接删最糟，你连这里曾经测过什么都丢了。

>

> 第三件，成果怎么固定。靠基线：在里程碑上把库冻成一个版本，记版本号和变更说明，下一轮在这份上改。改坏了能回滚，当初怎么测的能追溯。

>

> 基线的意思不难懂。就像每学期期末把教案定一版，写上日期和改了什么，下学期在这版上改。改坏了，回上一版。

>

> 第四件，谁负责。作者管自己写的，模块负责人管本模块，测试负责人管基线与评审。责任要落到这一块归谁。否则大家一起维护，等于没人维护。责任人写进库里，交接时才有人接得住。

> **提问：**功能被删掉了，对应用例怎么处理？直接删行不行？

> **预设回答：**

- 答"删掉，留着占地方"——错答。删了就丢了历史，功能将来回来要重写，出问题也查不到当初怎么测。

- 答"标记失效，下线但保留记录"——正确做法。保留的是曾经测过什么、为什么下线，这是资产。

- 答"先不改，反正不跑了"——最危险。库里留着一条永远跑不过的用例，通过率长期失真。

- 收口：维护的第一原则是不留模糊状态，能用、不能用、为什么不能用，得有一个明确答案。

> **易错点 / 考点：** ① 四种动作（更新、新增、废弃、归档）容易只答更新；② 基线的概念与作用（冻结、可回滚、可追溯）是加分点；③ 陷阱题"用例维护就是改错别字"——错，维护的核心是跟着需求与缺陷演进。

> **【思政落点 3】**：维护用例是最没有掌声的那部分工作：改十条没人看见，不改却在半年后以漏测的形式爆出来。它对应的知识点是触发机制与责任人——有触发就必须动，动了还要留痕。落到价值观上，是对变更负责、不糊弄：需求变了就把用例改到位，改不动就如实标失效。明知不对还含糊留着，比能力不足更伤人。

> **过渡：** 机制讲完了，看一条把机制画成流程的线。

【11.35】测试用例维护流程示例（1.5 分钟）

> 一条用例改完了，凭什么说它改对了？流程就是干这个的。顺着走一遍，看每一步吃什么、出什么。

>

> 起点是触发：需求变更单、新提交的缺陷、评审意见、环境升级通知。触发源要明确，否则流程启动不了。很多团队卡在知道要改，就是没有一个正式入口。入口写在哪，就决定这件事会不会被忘掉。第二步，影响分析：这次变更影响哪些功能、哪些用例。靠追溯关系和模块属性找。这一步做得准不准，决定维护成本是改五条还是改五十条。分析不准，后面五步都在改错的地方。第三步，动作执行：更新、新增、废弃，选一个或几个。这里要同步：用例改了，套件里的引用、缺陷单里的关联、需求矩阵里的映射都得跟着动。只改用例不改关联，等于留了个断点。第四步，评审：改完得有人看。看的不是文字，是判定还成立吗。需求变了，原来那条判定可能已经不成立，这得懂业务的人确认。评审只看文字，改完还是跑不过。第五步，入库打版本：形成新基线，写清变更说明。这一步最常被省，也最不能省。第六步，回归验证：改完的用例实际跑一遍，确认还能执行、判定还准。改了没跑，等于没验过。

>

> 最后是度量反馈：这轮改了多少、新增多少、废弃多少，数字回到覆盖率里去。流程在这儿闭环，也给下一轮触发攒依据。数字不回填，下一轮又从头猜。猜出来的数字，汇报时自己都不敢念。

> **提问：**这条流程里，哪一步最容易被省略？后果是什么？

> **预设回答：**

- 答"评审，自己改的觉得没问题"——很常见。后果是判定悄悄失真，执行时才发现用例和需求对不上。

- 答"回归验证，改完就当改好了"——同样常见，还更危险：错用例会一直错下去，还会误导统计。

- 答"入库打版本，反正文件在那儿"——错答。这一省，回滚能力就没了，将来谁也说不清何时改的、为什么改。

- 收口：三步都不能省，它们分别守住正确性、有效性、可追溯性。

> **易错点 / 考点：**① 七个环节的顺序；② "影响分析靠追溯"是简答要点；③ 陷阱题"用例改完不需要再执行"——错，改完必须回归验证。

> **过渡：**机制和流程都有了，接下来回答一个更硬的问题：这套东西够不够。

【11.36】11.2.5 测试用例的覆盖率（2 分钟）

> 覆盖率这三个字被问得最多，也最容易被糊弄过去。三个量法，量的不是一件事。

>

> 第一个，用例本身：发现缺陷后补充的测试用例数，除以总的测试用例数。举个具体的数。这一轮发现 20 个缺陷，有 12 个是原有用例没覆盖到的，于是补了 12 条。总用例 600 条，比值就是 2%。这个数不是越高越好。它测的是库的学习能力，不是勤奋程度。持续偏高，说明前期设计太粗，缺陷在替你做设计。补得越多，说明前面的设计越靠猜。长期为零，更让人怀疑是不是没人认真回填。回填这件事，得有流程逼着做。

>

> 第二个，需求和功能点覆盖率：每个功能点都要有测试用例。课件给的经验口径是平均大于 4 个每功能点。为什么是这个量级？一个功能点只有一条用例，基本只覆盖正常流程。边界、异常、组合都要有，四条左右才勉强够。这个数一眼就能看出漏测区：某个功能点下一条用例都没有，风险就摆在那儿。功能点多、用例少的模块，就是下一轮要补的地方。

>

> 需求覆盖那块举个例。一个"优惠券"功能点下面一条用例都没有，你在覆盖率表上看不出来，因为分母里根本没有它。所以需求覆盖得对着需求清单一条条数。

>

> 第三个，代码覆盖率：借工具跟踪执行过程，看代码行、类、方法的覆盖情况。它最客观，不看你怎么写用例，只看程序实际跑了哪些代码。但它有个明显的毛病：覆盖率高的测试不一定有效。全部代码都执行过，一个断言都不做，缺陷照样一个不抓。所以这个数要跟断言、跟缺陷发现情况一起看。三个数分开看，才知道是设计漏，还是执行漏。

> **提问：**代码覆盖率达到 90%，能不能说明测试做得不错?

> **预设回答：**

- 答"能，90% 已经很高"——错答。覆盖率只证明代码被执行过，不证明结果被验证过。

- 答"不能，还得看有没有断言"——正中要害。执行不等于验证，没断言的执行只是跑过去了。

- 答"还要看需求覆盖和缺陷发现情况"——很完整。三个维度互补，单看一个都会误判。

- 收口：覆盖率回答有没有走到，断言回答有没有验到；走到没验到，是最常见的假安全感。

> **易错点 / 考点：**① 三类覆盖率的名称与算法，第一类的分子分母别记反；② "平均大于 4 个每功能点"是课件给的经验口径，选择题常考；③ 陷阱题"代码覆盖率越高测试质量越好"——错，覆盖与有效不是一回事。

> **过渡：**结果量完了，最后看支撑这些结果的那个框架由什么构成。

【11.37】测试用例框架的构成（1.5 分钟）

> 前面讲了七八样东西，现在把它们装到一起。框架的目的很直接：更有效、有序地组织测试。装成四层，外加一条隐线。四层加一条隐线，少一层都撑不久。

>

> 第一层，骨架是层次结构：项目、模块、功能、用例，或者版本、目标、套件。骨架决定东西放在哪，也决定统计怎么汇总。层级一变，历史数据就对不上。第二层，血肉是属性体系：目标性、范围、关联性、阶段性、时效性，再加优先级、类型、估计用时。属性决定东西怎么被筛出来。字段填不全，视图就筛不干净。骨架和属性是一对：骨架管位置，属性管标签。只有骨架，筛不出东西；只有属性，人找不到入口。第三层，规则是模板与命名：模板规定字段，命名规定格式，编号怎么排、标题怎么写、套件怎么叫。规则的价值只有两个字，一致。一致才能被评审、被检索、被工具处理。格式一乱，评审先花半小时对格式。第四层，承

> 载是工具与追溯：用例放在哪个系统，和需求怎么连、和缺陷怎么连、变更怎么留痕。几千条靠文档管，维护成本会高到没人愿意碰。工具不在多，能关联需求就够了。

>

> 模板举个例。字段固定成那几项：编号、标题、前置、数据、步骤、期望、结果、追溯需求。少一项，评审就得追着人问；多一项自由填写的备注，三个月后全是只有作者看得懂的话。

>

> 隐线是维护规则：谁改、什么时候改、怎么评审、怎么打基线。它不写在图上，却决定这套东西能活多久。隐线断了，前面四层慢慢退成一张旧图。按这五样搭课程设计的库，结构站得住。顺序也别反：先想结构，再挑工具。结构不对，工具只会把乱放的东西摆得更整齐。

提问：只有目录树、没有属性体系，会在哪个环节最先出问题？

预设回答：

- 答"找用例的时候"——确实最先暴露，但通常还能靠命名硬找。

- 答"统计和筛套件的时候"——更准确。没有属性，生成不了高优先级回归套件，也统计不了类型覆盖。

- 答"写用例的时候"——也有道理，没有属性约束，作者不知道填什么，形式立刻散掉。

- 收口：骨架解决找得到，属性解决筛得出；只找不筛，测试就只能全量跑。

易错点 / 考点：① 框架的四个组成部分加维护规则；② "骨架管位置、属性管标签"是简答要点；③ 陷阱题"框架就是目录结构"——错，目录只是骨架那一层。

【思政落点 4】：框架里最不起眼的一层是模板与命名，它管的是一致。字段统一、编号统一、套件名字统一，后来的人才能检索、评审、交给工具处理。这件事对应的是把个人习惯变成团队规矩。落到价值观上，是敬业与协作：你多写一行规范的说明，下一个接手的人就少问十句。手艺人的规矩，往往都花在别人看不见的地方。

过渡：全章讲完，用两页收口。

【11.38】本章小结（2 分钟）

盖上课件，你能复述出几条？四条小结，按一条、一套、一棵树、一条命来记。记不住四条，就记这四个词。

>

> 第一条，基本属性和扩展属性。基本属性解决这条用例成不成立：前置条件、测试数据、测试环境、操作步骤、期望结果。扩展属性解决好不好管：编号、标题、目的、类型、优先级、层次、估计用时、追溯关系。两层都要会。只答基本属性，是半张答卷。两层说的是两个问题：写得对不对，管不管得住。

>

> 举个最简单的例子。"输入正确的用户名和口令，点登录，进首页"，五样基本属性齐了。这条用例叫什么、归哪个模块、优先级多高、跑一次几分钟，是扩展属性。少前者跑不起来，少后者管不起来。

>

> 第二条，为了测试子目标构建 Test Suite。注意"子目标"这个词。套件不是按目录长的，是按目标长的。一个计划下面有几个子目标，就有几组套件。子目标写得虚，套件就是按目录切。

>

> 第三条，框架里的层次关系和依赖关系。层次解决放在哪，依赖解决先跑谁。依赖最容易漏。有些用例必须先 A 后 B，比如先下单再支付，这个顺序得显式标出来。不标，套件一乱序执行，就是一片假失败。团队会花几天去追一个根本不存在的缺陷。依赖标不全，跑出来的失败一半是顺序造成的。

>

> 第四条，评审、维护和持续改进。用例不是写完就完，要评审、要跟版本走、要在缺陷里学习。评审和复审分开做，前者看新写的，后者看还活着的。四条收在一句：一条要合格，一套要成目标，一棵树要有层次和依赖，一条命靠评审和维护续着。考综合题时，这四句能当四个小标题。答题时四个小标题下面各写一句，分就稳了。

> **提问：**如果只允许你从这一章带走一句话，你会带哪一句？

> **预设回答：**

- 答"用例要有编号和优先级"——太细，是操作层面的东西，带不走整章。

- 答"用例是可复用、可度量、可追溯的资产"——概括得好，用途和性质都说到了。

- 答"写完还要维护"——朴素但很准。多数库不是死于写得差，是死于没人维护。

- 收口：我的版本是，用例写给别人用，库建给下一版用。

易错点 / 考点： ① 四条小结要与正文对应，常以"本章学了什么"的形式考；② 依赖关系最容易漏，补上能加分；③ 陷阱题"用例评审只在新写时做"——错，变更与定期复审同样要评审。

过渡： 最后三个思考题，正好检验这章有没有真用起来。

【11.39】思考题（2 分钟）

这三问丢分，多半不是不会，是答得散。答题的抓手先说清。先把评分点在纸上列成三行，再动笔。

>

第一问，构造测试套件里哪种方法最常用。抓手是三种组织方式各自的适用条件。按程序功能模块组织最常用，因为它直观，而且和开发的分工对得上。实践中以它为主干，再叠加按优先级切分、按类型筛选。光写一个名字拿不到全分。要说清为什么最常用，另外两种什么时候用。答题时把为什么写在前面，名字写在后面。

>

第二问，怎么有效维护用例。抓手是四件事：触发机制、动作分类（更新、新增、废弃、归档）、基线与版本、责任人。再加两条：把缺陷回填成用例，让库从缺陷里学习；定期复审，主动清过期用例，别等它跑不过才被发现。两条加分项能写就写，简答按点给分。

>

第三问，用什么工具度量覆盖率。这问要分两类答。代码覆盖率工具一类：Java 有 JaCoCo、Cobertura。C 和 C++ 有 gcov、lcov。前端 JS 有 Istanbul、nyc。它们给的是语句、分支、方法、类的覆盖。另一类是用例与需求管理工具。它们统计需求到用例的映射率、每个功能点的用例数、执行状态分布，用来算需求覆盖和进度。两类混着答，阅卷的人找不到点。工具名写不全没关系，两类分清楚就拿分。

>

三问连着看，就是三条主线：怎么组织、怎么维护、怎么度量。方法类简答看要点齐不齐。先列点，再各补一句解释。三问答顺了，这一章的方法线就通了。

提问： 为什么"按功能模块组织"会成为最常用的方式？它赢在哪？

预设回答：

- 答"因为开发就是按模块分工的"——最实在的原因：用例组织和人的分工对齐，沟通成本最低。

- 答"因为它最直观，新人一看就懂"——也对，可理解性直接影响使用率。

- 答"因为它覆盖最全"——错答。组织方式不决定覆盖，覆盖靠设计；模块切法甚至会漏掉跨模块场景。

- 收口：最常用不等于最好；模块切法是主干，跨模块和类型维度要靠属性与专门套件补。

> **易错点 / 考点：** ① 第三问必须区分代码覆盖率工具和用例管理工具，只答一类会失分；② 第一问答"按功能模块"要给理由；③ 陷阱题"覆盖率只能靠工具算"——需求覆盖率很大程度靠映射关系与人工确认。

> **过渡：** 这一章到这儿，最后把要带走的三件事收一下。

【11.40】感 谢 聆 听（1 分钟）

课讲到这里，带走三个动作，出教室就能用。

>

> 第一个，把一条用例写清。挑课程设计里你最熟的一个功能，按前置条件、测试数据、测试环境、操作步骤、期望结果五样写一条。然后找个没参与的同学照着跑。他跑不出你的结论，就回去改。这一条练的是表达。

>

> 第二个，把一组用例组织起来。编号、标优先级、挂层次，再按一个目标组一个套件。然后问自己：明天只给我两小时，我跑哪套？答得出来，说明组织有效。这一条练的是管理。

>

> 第三个，给它留一条命。写下什么情况下这条用例要改，下次需求变更时真的去改。这一条练的是责任。

>

> 测试这行有句话：缺陷会说话，但只有留下证据的人听得见。用例就是那份证据。证据留清楚了，三年后接手的人不必感谢你，他只要能干活就行。谢谢大家。

> **提问：** 课后回答一句：你手上这个项目，三年后新人接手时，最先感谢你留下的是什么？

> **预设回答：**

- 答"代码注释"——有用，但注释解释怎么做的，不解释哪里容易错。

- 答"用例库"——这正是本章的答案：它记录的不只是怎么测，还有哪些地方历史上出过问题。

- 答"文档写得全"——方向对，但要具体。全而散的文档没人读，有结构、可检索、有判定的用例才有人用。

- 收口：能被下一个人用起来的，才叫资产；用不起来的，只能叫记录。

> **易错点 / 考点：** ① 收口三动作（写清、组织、维护）可当课程设计自检清单；② 综合题里"用例是资产"要用可复用、可度量、可追溯、可传承四个词撑住；③ 陷阱题"用例写完交给文档管理员就结束了"——错，用例的寿命等于产品的寿命。

> **过渡：** 下一章进入缺陷管理与测试报告，把这一章的用例和执行数据变成能交出去的质量结论。

四、思政落点清单（共 4 处）

序号	所在页	落点主题
1	11.9	把用例当资产经营，还是当交差材料，是职业成熟度的分水岭。前者每版更新，每出一个缺
2	11.19	写用例最见功夫的地方不是工具会不会用，是表达够不够严谨。一句"输入非法密码"看着
3	11.34	维护用例是最没有掌声的那部分工作：改十条没人看见，不改却在半年后以漏测的形式爆出
4	11.37	框架里最不起眼的一层是模板与命名，它管的是一致。字段统一、编号统一、套件名字统一