《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】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的

> 页为可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出测试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；其后是板书、提问与预设回答、易错点/考点、过渡语。
>
> 页码锚点与课件页一一对应（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）

> 各位同学，我们进入第 11 章——**设计和维护测试用例**。
>
> 前面十章，我们把"测什么、怎么测、什么时候测"这条主线走完了，你手上已经攒了一批方法：等价类、边界值、判定表、路径覆盖。可方法只解决"怎么想"，它不解决另一个问题——**想出来的东西写不写得下来、别人接不接得住**。
>
> 这一章补的就是这一环：先把测试行为变成**可以交付的文档**，再把文档变成**可以长期维护的资产**。前者叫设计，后者叫维护，就是章名里的两个动词。
>
> 我带一个问题进来，请你先记着：三年以后你离开这个团队，新人接手这个模块，他凭什么知道哪些地方已经测过、哪些地方一动就会出事？答不上来，说明这个团队缺的不是人，是一份像样的**用例库**。

> **板书：**中间写"第 11 章 设计和维护测试用例"；左侧写"前面十章：怎么想（方法・流程・各级测试）"；右侧写"本章：怎么落纸、怎么养（设计・组织・跟踪・维护）"。

> **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，功能就那些"（典型错答）——现实里常常不一样，因为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，而不取决于流程。

- 答"写成用例就一样了"——方向对，但要补一句：**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，这正是本章要解决的核心。

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，你才知道后面每一页在解哪一道题。

【11.2】本章要解决的问题（1.5 分钟）

这一页把全章的五个问题摆出来，我逐个点，你对着自己的经验判断一下能答几个。

第一问，**为什么我们要使用测试用例**？注意是"使用"，不是"写"。也就是说，用例的价值不在写完那一刻，而在被别人反复执行、反复参照的时候。这一问后面用一页专门讲它的六项作用。

第二问，**测试用例有哪些基本元素组成**？前置条件、测试数据、测试环境、操作步骤、期望结果，这五样是最基本的，再往上还有编号、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，用例就不成立；缺了扩展属性，用例就不好管。

第三问，**编写和设计时需要遵循哪些基本原则**？这一问是本章的"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，**测试用例的结构和设计过程**是什么样的？从一条用例到一套用例，从用例到套件再到测试计划，这是一棵树。

第五问，**怎么跟踪和维护测试用例**？用例会过期、会失效、会随着需求变更而必须改，谁改、什么时候改、怎么留痕，这一问最容易被学生忽略，却是工程上最贵的一块。

课件还补了一句很重要的话：**测试用例设计的具体方法，第 3 章已经介绍过了**。所以这一章不讲等价类怎么

划、边界值怎么取，那些是你的基本功；这一章讲的是把这些方法产出的结果**写成什么样子、怎么组织、怎么活下去**。

板书：竖排五问：① 为什么要用 → ② 由哪些元素构成 → ③ 编写原则 → ④ 结构与设计过程 → ⑤ 跟踪与维护。右侧一行："方法在第 3 章，本章管'落纸'与'养'。"

提问：这五问里，你认为在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，但要追问一句：不写是因为不会写，还是因为写了没人用？这两种病因，药方完全不同。

- 答"第三问，写得太啰嗦"——很具体。冗长的用例执行慢、维护贵、还容易漏判，第 19 页会给"步骤不超过 7 步"这种硬约束。

- 答"第五问，写完就没人管了"——这是最接近真相的答案。行业里用例库死亡的方式几乎都是同一种：需求改了，用例没改，跑两轮对不上，最后没人信它。

- 收口：五问里前四问答得好叫"会写"，第五问答得好才叫"会管"。

易错点 / 考点：① 五问的顺序要能背下来，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，本章只讲构成要素与规范；③ 注意"使用"和"编写"的区分，考简答时答"用例的价值在执行与复用阶段实现"比答"用例能发现缺陷"更准确。

过渡：五问对应两节，我们先看这两节各自管什么。

【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）

这一页是本章的分节，两节的边界很清楚，我用一分钟划开。

第一节 **11.1 测试用例构成及其设计**，解决的是"**一条**用例"的问题：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**，解决的是"**一堆**用例"的问题：给用例挂上可管理的属性，用测试套件把它们组织起来，在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。

一句话区分：第一节练**表达能力**，第二节练**管理能力**。前者决定你写的东西有没有用，后者决定它能不能活过三个版本。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、最后没人看。

板书：横分两栏：11.1 一条用例（构成・重要性・书写标准・考虑因素・基本原则）｜11.2 一堆用例（属性・测试套件・跟踪・维护・覆盖率）。底部："表达力 → 管理力"。

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，但要注意：规范是可以靠模板一次性解决的，管理不行。

- 答"先做 11.2，先有库再谈质量"——也成立，很多团队确实是先建库、再逐步规范，边用边改。

- 答"两节是一回事"（典型错答）——这句要纠一下：它们的目标不同。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"这个库别人能找得到、跟得上、改得动"。

- 收口：先做哪节看团队现状；但考试问你"用例管理的两个层面"，答案必须把两节的名字都点到。

易错点 / 考点：① 两节的名称与各自五个小节要能对上号，别把"覆盖率"错记到 11.1；② 判断题陷阱："用例组织和跟踪属于测试执行阶段的工作"——错，它是测试设计之后的持续性工作，跨阶段存在；③ 记住 11.2 里有"属性"这一节，很多学生会漏。

过渡：进入 11.1 之前，先回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）

这一页只给了一个问题，没有给答案，这是故意的。答案是整节的内容，问题本身才是这一页要你带走的东西。

我们先想清楚：所谓"测试行为"，在你脑子里是什么形态？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、我要在什么环境里做。这四样东西在你脑子里是**同时存在**的，所以你觉得自己"很清楚"。

可一旦要交给别人，这团东西就必须被**摊开成一条路径**：从什么状态出发，做什么动作，得到什么，和什么比。摊不开，别人就只能猜。

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

> 你写"输入一个非法密码"，别人会问：多长算非法？几位？含什么字符？大小写？——每问一句，都是你的描述漏了一处。
>
> 生活里最像的东西是菜谱。"盐少许"和"盐 3 克"，一个靠师傅，一个靠纸。测试用例要的是后者。
>
> 所以看这一页，你要看出的是：**从"我知道怎么测"到"别人照着也能测出同一个结论"，中间隔着一整套描述规范。**

> **板书：**中间写"测试行为 → 描述"；四要素："验证什么・给什么输入・在什么环境・怎么判定"；下方一行："描述的本质＝去歧义（菜谱：盐少许 vs 盐 3 克）"。

> **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，这说明一句话里至少藏了三处歧义。

- 答"不用问，一看就懂"——这是最典型的错答。你自己懂，是因为你脑子里有上下文；用例的读者没有你的上下文，这就是为什么必须写死。

- 答"那我把所有情况都写上"——另一个极端：写成一大段，读者反而找不到判定点，这就是第 19 页要讲的"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 记住描述的四个落点：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）

> 这一页给的是实物。我不逐字念，我教你**怎么读一条用例**。
>
> 读的顺序建议反过来：**先看判定，再看前提，最后看步骤**。先看判定，是因为一条用例存在的理由就是"能判对错"；如果判定含糊，后面写得再细也没用。再看前提，因为执行者要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。最后看步骤，步骤是路径，路径错了，判定也就无从谈起。

反过来看一句常见的话："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。

示例页的价值在于给你一个**可以照着改的模板**。你们做课程设计时，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，你就摸到格式的规律了。

还有一层要提醒：示例里的数据是**示意**，不是标准答案。你写自己的用例时，数据要能反映你要验证的那条规则，别照抄。

板书：读用例三步：① 先看期望结果（能不能判）→ ② 再看前置条件（从哪起跑）→ ③ 最后看步骤（路径对不对）。右侧对照："登录功能测一下"＝任务；五要素齐全＝用例。

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"却没法判。

- 答"不能，正常是什么没说"——完全正确。要改成"提示'密码错误'且停留在登录页、账号未被锁定"这种可观测的句子。

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

- 收口：判定的标准是**可观测**——界面能看到、日志能查到、数据能比对，三者至少占一个。

易错点 / 考点：① 判定必须可观测，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、写漏了会跑偏；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：那条用例到底由哪些零件拼起来？我们进 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）

这一页是 11.1 的小节导航，四个小节串起来正好是"一条用例从要不要写、到合格"的流程，我按这个逻辑讲一遍。

第一步，**11.1.1 测试用例的重要性**。这一步回答"值不值得写"。我不打算讲情怀，我讲代价：不写会出现什么——进度不可测、覆盖不可证、人员一走经验清零。你把这三笔账算清楚，写用例这件事就不需要靠自觉

了。

第二步，**11.1.2 测试用例设计书写标准**。这一步回答"一条用例由哪些部分构成，每部分填什么"。它看起来最像格式要求，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。

第三步，**11.1.3 测试用例设计考虑因素**。这一步回答"设计的时候，脑子里要同时想哪几件事"——要有代表性，要盯着设计的弱点，既要正输入也要异常输入，还要想用户真实会怎么用。

第四步，**11.1.4 测试用例设计的基本原则**。这一步给红线：不写含糊的、不写冗长的、不写重复的、一个用例一个验证点。

四步的逻辑是：**要不要写 → 写成什么形状 → 该想什么 → 不许怎么写**。你按这个顺序记，比按编号背要牢。

再补一句学习方法：这四步不是四次性的工作，而是**一道用例要反复过的四道关**。你写完一条用例，先问自己"这条值不值得存在"，再问"格式齐不齐"，再问"该想到的异常想到了吗"，最后问"有没有含糊和冗余"。四问都过，这条用例才算能交给别人。

> **板书：**四步箭头：值不值得写（重要性）→ 由哪些部分构成（书写标准）→ 脑子里想什么（考虑因素）→ 不许怎么写（基本原则）。右侧："契约：写的人与读的人理解一致。"

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，但从协作看不行：没有统一形状，每个人想出来的东西都没法互相比较，评审也就无从谈起。

- 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，都是要讲的"（典型错答）——这句要纠：顺序体现的是教学逻辑，考试让你"概述用例设计的过程"，按四步作答才拿得全分。

- 收口：格式先行不是形式主义，是为了让"想"这件事可以被别人检查和质疑。

易错点 / 考点：① 四个小节名称与顺序必须记准；② 课件这一页的小节编号沿用了上一版的排版，按内容它就是 11.1 的四个小节，你做笔记按"重要性—书写标准—考虑因素—基本原则"记，别照旧编号抄；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。先看什么是一条用例。

【11.7】什么是测试用例？（2 分钟）

这一页给的是定义，课件给了三种说法，我用一句话把它们叠起来：**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：**测试用例可以独立进行测试执行的最小单元**。"独立"和"最小"是两个关键词。独立，意味着它不依赖另一条用例的执行结果，拿出来就能跑；最小，意味着它不能再拆——再拆就判不出结论了。这一条决定了用例的**粒度**，第 21 页专门讲。

第二种说法：**测试内容的一系列情景，以及每个情景中必须依靠的输入和输出，用来判断软件正确性的测试文档**。这句长，但结构清楚：情景（在什么场景下）、输入（给什么）、输出（应该得到什么），三者合起来才是"判断正确性"的依据。注意"必须依靠"四个字——输入输出不是随手记的，是**判定所必需**的那部分。

第三种说法：**测试用例就是将软件测试的行为活动转化为规范化的文档**。这句最提纲挈领。测试行为是活动，活动会消失；文档会留下来。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，你就明白为什么教材说用例是"测试工作的指导"：它既是动作的说明书，也是判定的尺子，还是交接的凭据。

这里还有个容易被忽略的推论：既然用例是"文档"，它就有**读者**。写的人、执行的人、评审的人、三个月后来接手的人，都是读者。判断一条用例写得好不好，最省事的办法是换一个没参与过的人来读——他能不能在十分钟内把这条用例跑出和你一样的结论。跑不出，问题不在他，在文档。

> **板书：**中间"测试用例"；三支箭头：最小独立执行单元（粒度）·情景＋输入＋输出（判定）·行为活动→规范化文档（转化）。底部："一次测试行为，写成谁都照着能做、能判的文档。"

> **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，执行成本会上升。

- 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：**不依赖其他用例的结论**，但可以依赖前置条件所描述的状态。

- 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，不影响其他条目的判定"。

- 收口：独立是**判定上的独立**，不是环境上的隔离；把这两件事分开，你才会写前置条件。

> **易错点 / 考点：** ① 三种定义要能各说一句，考简答时"最小独立单元""情景＋输入输出""行为转文档"三个点各占分；② 别把"测试用例"和"测试脚本"混为一谈：脚本是可执行的代码，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，包含的是判定所必需的数据。

> **过渡：** 既然是把行为变文档，那这份文档到底解决了什么问题？我们看它的重要性。

【11.8】11.1.1 测试用例的重要性（2 分钟）

这一页讲"为什么非写不可"。课件给了三层理由，从商业目标一路落到质量保障，我逐层拆。

第一层是**商业目标**：如何以最少的人力、资源投入，在最短时间内完成测试、发现缺陷、保证品质，这是软件公司一直在探索和追求的目标。这句话听着像口号，其实是个很硬的约束：测试资源永远是有限的，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"——而这两个问题的答案，只能靠用例来承载。没有用例，"测够了"就是一句主观感受。

第二层是**可管理性**：软件测试是有组织性、步骤性和计划性的；为了把测试行为转换为**可管理的、具体量化的**模式，需要创建和维护测试用例。注意"量化"这个词——进度是执行了多少条、通过率是多少、还剩多少条没跑，这些都是数字。没有用例，你只能回答"差不多了"，没法回答"还差 37 条，其中 12 条是阻塞的"。

第三层是**准则与保障**：测试用例是测试工作的指导，是软件测试必须遵守的准则，更是测试质量稳定的根本保障。三层连起来是一条因果链：资源有限 → 必须量化 → 量化要靠用例 → 用例稳定，质量才稳定。

反过来看代价，你就知道为什么这一节值得讲两分钟：不写用例的团队，进度靠感觉、覆盖率靠回忆、人员流动靠运气。这三样凑在一起，质量一定是波动的。

这层因果链还有个隐含结论：**用例的稳定，比用例的数量重要**。一个团队有 800 条老用例、结构清楚、每版都在跑，比有 3000 条堆在一起、没人敢动的用例库，质量保障能力强得多。数量是虚荣指标，稳定性才是能力。

> **板书：** 三层：① 商业目标（最少投入・最短时间・保证品质）→ ② 可管理可量化（组织性・步骤性・计划

性）→ ③ 准则与保障（指导・必须遵守・质量稳定）。右侧写："资源有限 → 必须量化 → 量化靠用例。"

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

预设回答：

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

- 答"大概测了七成"——听着像量化，其实"七成"是按什么算的？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，而这个分母就是用例集。

- 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**；度量立住了，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"——答进度不可测、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，接下来把它的作用一条条列清楚，这是本章出现频率最高的一个考点。

【11.9】测试用例的作用（2 分钟）

这一页是本章最像"考点清单"的一页，课件列了六项作用，最后收在一句话上：测试用例是重要参考依据，用来提高测试质量。我一项一项讲，每一项都给你一个"不这样会怎样"。

第一项，**有效性**。用例是围绕测试目标设计的，它保证每一次执行都指向一个明确的验证点。没有它，测试员点来点去，看起来也没闲着，但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、将来还能交给自动化去跑。可复用是用例最实在的一笔经济账：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，可以按模块分、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**。这一项最容易被低估。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。这样质量结论就不依赖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，这些都是可统计的量；有了量，才能谈进度、谈风险、谈资源。管理不是靠喊，是靠数字。

第六项，**知识传递**。这一点我要单独说。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、历史上踩过哪些坑，全沉淀在用例的前置条件和判定里。用例库其实是一个团队对这块业务的**理解底稿**。

板书：六项竖排：有效性・可复用性・易组织性・客观性・可评估性与可管理性・知识传递；收口一句："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）——账算得对，但复用主要省的是执行成本，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，而这个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，客观性让结论不随人变；但要说明：客观性靠的是判定标准事先写死，不是靠某个人公正。

- 收口：这六项其实是一条链——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。

【思政落点 1】：把测试用例当"知识资产"看，是职业成熟度的分水岭。有的团队把用例当交差材料，评审完就锁进硬盘，人一走全废；有的团队把它当资产经营，每版更新、每次缺陷都回填一条用例，三年后新人接手，翻开库就知道这块业务的坑在哪。同样的工作量，一个归零，一个增值——差别不在技术，在于有没有把"给团队留下东西"当成自己的责任。

易错点 / 考点：① 六项作用要能默写，简答题几乎年年出；② 最容易漏的是"客观性"和"知识传递"，答题时把这两条补上往往能拉开分差；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，其余五项都指向管理与传承。

过渡：作用讲清了，接下来是"写成什么样"。先看书写标准该回答哪几个问题。

【11.10】11.1.2 测试用例设计书写标准（1.5 分钟）

这一页课件用四个问句加一个"还有呢"，把书写标准逼出来。我按这个思路走一遍。

第一问，**目的**？这条用例是为了验证什么？是验证一条业务规则、一个边界、一种异常分支，还是验证性能或兼容性？目的不清，后面全是无效劳动。

第二问，**输入数据**？给它什么？这里要具体到值：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。你写成"输入合法数据"，等于没写。

第三问，**操作步骤**？按什么顺序做？步骤要能被执行者复现，包括从哪个界面进、点哪个按钮、是否要先登出再登录。

第四问，**期望结果**？做完应该看到什么？要可观测，要唯一，不能出现"正常""符合要求"这种自我指涉的表述。

最后那句"**还有呢**"，是这一页的真正考点。除了这四样，还缺什么？缺**前置条件**——执行之前系统必须处于什么状态；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。这就是下一页要展开的内容。

再补一层意思：这四个问句为什么按这个顺序问？因为它们对应一条用例被理解的顺序——先知道要证什么，才知道该给什么数据；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。顺序反了，你会先写步骤再想目的，结果就是"为了点按钮而点按钮"的用例，这是初学者最普遍的通病。

还要说清"书写标准"这个词的分量。它听起来像行政要求，实质是**团队内部的接口规范**：字段名称统一、填写粒度统一、判定写法统一。统一之后会发生两件好事——评审可以逐字段对照，不必从头读；工具可以按字段抽取，自动生成执行清单和统计报表。你的用例如果每个人一种写法，这两件事都做不成。

> **板书：**四问一行：目的？输入数据？操作步骤？期望结果？——下方补："＋前置条件 ＋测试环境 ＋（编号・标题・优先级・层次・追溯）"。

> **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产生不同输出，这不是缺陷，是用例没写清。

- 答"前置条件不重要，反正从登录开始就行"——这是典型错答：很多用例的起点是"账号已被锁定""购物车已有三件商品""数据库里已有历史订单"，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：前置条件讲的是**数据与业务状态**，环境讲的是**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大的坑，也是评审时最常被打回的一项。

易错点 / 考点： ① 四问加"还有呢"的五项，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，期望结果要写出可观测的具体表现。

过渡： 把"还有呢"补齐，就是测试用例最基本的内容。

【11.11】测试用例最基本的内容（2 分钟）

这一页给的是五项最基本的内容：**前置条件、测试数据、测试环境、操作步骤、期望结果**。课件还提示，完整模板见教材第 4 版的附录 C。我逐项讲"为什么它是最基本的"。

前置条件——执行这条用例之前，系统必须处于什么状态。账号是否已注册、余额是否为 0、订单是否已支付、缓存是否已清。前置条件的作用是**把起跑线画齐**，起跑线不齐，同一句话在不同人手里会跑出不同结果。

测试数据——具体到值的输入，以及必要的初始数据准备。这一项最容易偷懒，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，所以环境必须写清，而不是"默认你知道"。

操作步骤——从起点到判定点之间的动作序列。步骤要能被执行者照着做，要包含必要的等待与中间确认，不能跳步。

期望结果——做完应该观测到什么。这是用例的"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。

这五项的关系可以这样记：**前置条件定起点，测试环境和数据定条件，步骤定路径，期望结果定终点**。起点、条件、路径、终点四样齐全，一条用例才闭合。任何一项缺失，这条用例就变成一份"需要打电话问作者"的文档，而那份电话费，会在每一次回归里重复支付。

还有一个实践提醒：模板字段是全的，但不是每条都要写得一样长。判定依赖的写细，不依赖的写一句就够。比如一条界面文案用例，环境可以写"默认 Windows 浏览器环境"，没必要列配置清单；而一条数据迁移用例，环境必须写到数据库版本、字符集、数据量级。**字段一样，颗粒度按需**，这是写用例的老手感。

> **板书：**五框：前置条件（起点）→ 测试环境＋测试数据（条件）→ 操作步骤（路径）→ 期望结果（终点）；旁注"完整模板见教材附录 C"。

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：一旦缺陷在别的环境复现不了，你连"当时是什么环境"都说不清，问题会被拖成悬案。

- 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（比如历史数据累积）根本造不出来。

- 答"一项都不能删"——最稳的答案。如果非删不可，工程上通常是**把环境写进版本说明统一约定**，而不是彻底不写。

- 收口：这五项不是五个格子，是四条边界；少一条边界，判定就会漏风。

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，必须写到值或值的构造方法。

过渡：最基本的是五项，工程上还会挂上一批扩展属性，我们看一张完整的元素图。

【11.12】测试用例的元素（1.5 分钟）

这一页把用例的元素摊成一张全景图，你要看出的是**分层**，不是一个个名词。

最里面一层是**判定必需的元素**：输入、步骤、期望结果。少了任何一样，用例就不成立。这是第 11 页讲过的五项里的核心。

中间一层是**执行必需的元素**：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。

最外面一层是**管理必需的元素**：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正确性，却决定整个用例库能不能被组织、被检索、被度量。

为什么要分三层？因为三层的维护成本完全不同。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。你把三层混在一起写，就会出现在"期望结果"里塞业务背景、在"步骤"里塞优先级这种糊涂账。

还有一个实用判断：**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**。这个顺序能从粗到细筛掉大部分问题，比逐字读快得多。

板书：三个同心圈：内层（输入・步骤・期望结果＝判定必需）｜中层（前置条件・环境・数据准备＝执行必需）｜外层（编号・标题・目的・类型・优先级・层次・用时・追溯＝管理必需）。

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，编号的真正价值是**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、需求三者能互相指向，追溯才有落点。

- 答"编号可以不要，标题就够了"——错答。标题会改，改完就对不上了；编号一旦分配就不再变，这是它能当锚点的原因。

- 收口：编号是**不变的那一个字段**，其他字段都可以改，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）是理解这一页的钥匙；② 常考简答："测试用例的扩展属性有哪些"——答编号、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是什么样子。

【11.13】示例（1.5 分钟）

这一页是把上一页的元素落到一条具体用例上，我教你怎么"对照着读"。

第一眼看**判定点**：这条用例到底要证明什么？判定点通常就一句话，写在期望结果里。判定点含糊的用例，其余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？如果前置条件只写"系统已启动"，那这条用例对数据的依赖就被藏起来了，执行者会各自造数据，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"等待页面加载完成"。

第四眼看**属性**：优先级是高还是中？层次挂在哪个父用例下？估计用时写了没有？这三项平时不显眼，一旦进入排期和统计，缺一项就得靠人拍脑袋。

我特别想让你注意一件事：**一条用例只回答一个问题**。如果一条用例的期望结果里同时写了"提示错误"和"账号未锁定"和"日志有记录"，那它其实是三条用例捆在一起。捆在一起的结果是：只要有一项不符，整条判失败，你不知道到底哪一项坏了。

板书：读示例四眼：判定点（要证什么）→ 起点（前置条件说死没）→ 路径（步骤跳没跳）→ 属性（优先级・层次・用时）。右侧："一条用例只回答一个问题。"

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"判失败时不知道是哪一项坏了"——正中要害。定位成本上升，返工时间被白白消耗。

- 答"没关系，反正都要验"——错答。捆在一起，失败率会虚高，统计出来的"通过率"也就失去意义。

- 答"那拆成三条，成本不是更高"——成本确实上升，但换来的是可定位；工程上通常按"验证点是否独立可判"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**可归因**。

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

过渡：有了元素、会读示例，接下来判断什么叫"好"。课件给了八条特征。

【11.14】良好测试用例的特征（2 分钟）

这一页给了八条特征，是本章的标准答案区，我按"能不能抓缺陷、能不能判、好不好管"三组来讲。

第一组讲**抓缺陷的能力**。第一条，可以最大程度地找出软件隐藏的缺陷；第二条，可以最高效率地找出缺陷。这两条要连起来看：既要有深度（藏得深的也能挖出来），又要有性价比（不能一条用例跑一天才挖一个）。第三条，可以最大程度地满足测试覆盖要求——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"。

第二组讲**可判读性**。第四条，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。第五条，使软件缺陷的表现可以清楚判定——缺陷暴露出来时，你要能一眼看出它错在哪，而不是"页面好像有点不对"。第六条，待查的输出结果或文件必须尽量简单明了——这条特别实用：如果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，应该改成只核对关键字段。

第三组讲**可维护性**。第七条，不包含重复的测试用例——重复用例是维护灾难，改一处要改十处，还容易改漏。第八条，内容清晰、格式一致、分类组织——格式一致让评审变快，分类组织让检索变快。

这八条你可以压成一句口诀：**抓得到、判得清、管得住**。答题时按这三组展开，比死记八条更容易写全。

再提醒一个常见误区：很多同学把"好用例"理解成"覆盖最多分支的那条"。分支多不是标准，**能明确判定、能稳定复现**才是标准。一条只验一个点、判定干脆的用例，比一条想兼顾五个点、判定含糊的用例，价值高得多。

> **板书：**三组八条：抓缺陷（找隐藏・高效率・满足覆盖）｜可判读（不过繁不过简・缺陷表现可判・输出简明）｜可维护（无重复・清晰一致分类）。

提问：为什么"重复的测试用例"被单独列为一条要避免的特征？

预设回答：

- 答"重复浪费时间"——只说对了一半。更贵的是**维护成本**：规则一变，同一逻辑要在多条用例里改，漏改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。真正需要多测的是**不同路径与不同数据组合**，机械重复不增加覆盖，只增加噪音。

- 答"重复说明我们设计时没查重"——很到位。查重应该发生在设计阶段，靠用例库检索和评审，而不是执行完才发现。

- 收口：重复用例的最大危害不是多跑几遍，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"是最常被考的一句，要能解释

两端各有什么代价；③ 判断陷阱："输出结果越详细越好"——错，课件明确要求"尽量简单明了"，判读成本也是成本。

过渡：八条是"好"的标准。设计的时候，脑子还要同时盯住几件事。

【11.15】11.1.3 测试用例设计考虑因素（2 分钟）

这一页讲设计时脑子里的五个"同时"。它和下一页的"原则"不同：原则是底线，考虑因素是**思路**。

第一条，**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。你测了"密码 6 位"和"密码 20 位"，就代表了长度这一维的两端；再测 7 位、8 位，边际收益很低。这条其实是在替你做减法：用最少的用例代表最大的输入空间。

第二条，**寻求系统设计、功能设计的弱点**。这一条最考验功力。设计弱点在哪？在状态切换、在并发、在权限交叉、在异常回滚、在边界条件。用例不能只顺着"正常流程"走一遍，要专门往设计者最没把握的地方捅。你可以这样想：如果我是开发者，我最怕别人测哪里？

第三条，**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，是因为初学者几乎都偏科：只测对的，不测错的。可真实世界里，用户输入错数据的频率远高于输入对的，系统在错误输入下的表现才是它的真实质量。

第四条，**分析怎样使得这样的错误或者异常能够发生**。这一条是把"要测异常"落到"怎么造出异常"：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。

第五条，**考虑用户实际的诸多使用场景**。真实用户不会按你的步骤走：他会中途退出、会连点、会同时开两个窗口、会在弱网下操作。用例里加几条这样的"不守规矩"的场景，往往能挖到最深的那批缺陷。

板书：五条：代表性典型性｜盯设计弱点（状态・并发・权限・异常・边界）｜正输入＋异常输入｜怎么造出异常｜真实使用场景。

提问：如果要给"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"连续输错五次锁定、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。异常包括数据异常、状态异常、时序异常、环境异常（网络、磁盘、时间），只测数据异常会漏掉一大半。

- 收口：造异常的能力，直接等于你挖缺陷的能力；这一条练的是想象力加工程手段。

> **易错点 / 考点：** ① 五条要考虑因素与 11.1.4 三条原则别混，前者是想什么、后者是不许怎么写；② 异常的四类（数据・状态・时序・环境）建议背下来，案例题答题时按类展开；③ 判断陷阱："用例只需覆盖正常业务流"——错，且是初学者最常见的错误。

> **过渡：** 除了内容，用例还有一批属性要填，这是它进入管理体系的门票。

【11.16】测试用例其它重要属性（2 分钟）

这一页给四类属性：**测试目的与测试类型、优先级、层次、估计用时**。它们不影响单条用例的正确性，但决定了这批用例能不能被排期、被筛选、被统计。

先说**测试目的、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？因为不同类型的用例执行条件完全不同：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，排期就会错配。

再说**优先级**，通常分高、中、低。优先级的真实用途是**在时间不够时决定放弃谁**。项目延期是常态，延期时一定要砍用例，砍哪些？按优先级砍。所以优先级不能凭感觉填，要有判据：核心业务、高频路径、不可恢复的损失，这些是高；边缘功能、提示文案这类可以降为中低。

第三是**层次**，父、子、孙这样的树形结构。层次解决的是"组织"和"复用"：一个父用例可以统摄一组子用例，执行时按树遍历，统计时按树汇总。层次还决定了用例在套件里挂在哪，第 26 页会接着讲。

第四是**估计用时**。一条用例大概要跑多久，五分钟还是半小时。它的用途有两个：一是排期，你能算出这一轮回归需要多少人天；二是发现异常——如果某条用例估计五分钟、实际跑了四十分钟，那要么是用例写得不清楚，要么是系统真的变慢了，两种都值得查。

> **板书：** 四属性：目的・类型（决定执行条件）｜优先级（决定砍谁）｜层次（决定挂哪、怎么汇总）｜估计用时（决定排期、暴露异常）。

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）——这是最糟的答案，等于把风险交给运气。

- 答"按优先级跑高优先级的"——正确方向。但还要补两步：高优先级里再按"本次改动影响到的模块"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，而且疲劳执行会带来误判，跑了等于没跑。

- 收口：优先级之所以要在设计阶段就填，就是因为砍用例的决定往往只有十分钟做，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，优先级应在设计阶段确定，并由评审确认。

过渡：属性是纸上填的，我们看两份把属性填全的示例，再对比它们的差别。

【11.17】示例一（1.5 分钟）

这一页是示例，我给你的第一条建议是：**别把示例当一条看，当一组看**。

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"。同一功能的一组用例摆在一起，你要看的是它们之间的**差异**：哪几个字段变了、哪几个字段没变。变的那几个，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。这说明了什么？说明这个功能的测试本质是**同一动作、不同数据**。看清这一点，你就知道该用数据驱动的方式组织它，而不是复制十遍、每遍改一个数。

第二件要看的是**属性列的填法**。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，示例给你示范。

第三件事，请你做一个小练习：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，就是你以后写用例的**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，你会发现写用例不再是"从零开始想"，而是"填空加判断"。

板书：读示例三条：① 看差异（哪个字段在变＝关键维度）② 看属性（优先级・层次・用时怎么填）③ 做

替换（沿用骨架，重写业务）。

提问：一组用例里，只有"测试数据"这一列在变，其余几列几乎一样。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"——方向上是对的，但要注意别合并掉判定：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，输入校验类用例本来就是一个动作配大量数据，价值在数据覆盖面。

- 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。这两种直觉，是从示例里能学到的最实用的东西。

易错点 / 考点：① 示例的用法是"看差异、看属性、做替换"，不是背诵；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，同一功能的用例应该高度同构，同构才好维护。

过渡：示例一是一组，示例二再给你一组不同形态的，我们对照着看。

【11.18】示例二（1.5 分钟）

这一页还是示例，重点变成**对照**：和示例一比，它在结构上有什么不同。

我提示三个对照点。第一，**判定点的数量**。示例二里如果一条用例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"的原则；如果出现一条用例挂多个期望结果，你就要判断它是不是该拆。判断标准还是那句：验证点之间能不能独立归因。

第二，**前置条件的写法**。有的示例把前置条件写成业务状态（"用户已完成实名认证"），有的写成数据状态（"数据库中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。这就是评审时最容易被追问的地方。

第三，**步骤的粒度**。步骤写到"点击提交"就够，还是要写到"在弹窗中点确定，等待列表刷新"？判据是：执行者会不会在这里产生歧义。会，就写细；不会，就别啰嗦。这也是下一页要讲的"避免冗长"的实操边界。

最后提醒一句：示例不是模板的终点。教材附录 C 给了更完整的模板字段，示例给的是**常用字段的填法示范**。

> 你写课程设计时，字段取模板的全集，填法参照示例的写法。

> **板书：**三个对照点：判定点数量（能否独立归因）｜前置条件（业务语言＋数据语言）｜步骤粒度（以"会不会歧义"为界）。右侧："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？这些都会改变结果。

- 答"不够，还缺角色和账号状态"——正确方向。前置条件的完整写法是"谁＋什么状态＋有什么数据"。

- 答"前置条件写越细越好"（典型错答）——另一个极端。写细要付出执行成本，判据是"这条用例的判定是否依赖它"，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 前置条件的三要素：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：示例看完了，我们收回来看 11.1 的最后一个小节——三条不能碰的红线。

【11.19】11.1.4 测试用例设计的基本原则（2 分钟）

这一页是本章的红线，课件给了三条，每条后面都跟了一句括号里的操作要求，我按"问题—判据—做法"讲。

第一条，**避免含糊的测试用例**。括号里那句英文说得很直白：**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**要写清，**操作步骤**要写清。含糊通常出在这两处：环境没交代，导致"我这能过、你那不过"；步骤没交代，导致"我点的是这个按钮、你点的是那个"。你要养成的习惯是：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，**将具有相类似功能的测试用例抽象并归类**，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，而该抽成"一个骨架＋一张数据表"。抽象这一步看着费事，收益在后面每一次回归里：改一次骨架，十条用例全跟着改。

第三条，**避免冗长和复杂的测试用例**。课件给了两个具体的量化建议：**操作步骤不超过 7 步**；**一个测试用例一个验证点**。为什么是 7 步左右？因为步骤一多，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，用例的可靠性就没了。为什么一个验证点？因为多验证点捆在一起，失败无法归因。

这三条其实在治同一种病：**含糊带来的不可控**。含糊在表达上，就写成冗长；含糊在结构上，就写成重复。所以三条原则可以压成一句：**说得准、归得类、留得短**。

板书：三原则：① 不含糊（Pass/Failed 清楚・环境・步骤）② 相似功能抽象归类（→ 数据驱动）③ 不冗长（步骤 ≤7・一个用例一个验证点）。底部："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，最终判"通过"其实是没跑到判定点。

- 答"失败了不知道错在哪一步"——更严重。步骤越长，缺陷定位的成本越高，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，用例的稳定性就下降了。

- 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。

【思政落点 2】：写用例最见功夫的地方不是会不会用工具，而是表达够不够严谨。一句"输入非法密码"看着省事，实际把判断责任推给了下一个执行的人；写成"输入 65 位字符的密码（上限 64 位）"，才把责任收回到自己身上。工程文书里的严谨不是抠字眼，是对同事的时间负责、对结论的可信度负责。你把每一处歧义都堵死，这份文档才配叫"标准"，而不是"草稿"。

易错点 / 考点：① 三条原则的表述要能复述，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，是选择题高频点；② 数据驱动是"抽象归类"的落地做法，别把两者答成两件事；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。

过渡：原则是底线，往上还有一层"质量要求"。先看单条用例要达到什么水准。

【11.20】单个测试用例的质量要求（2 分钟）

这一页给的是"一条用例合格证"上的六项，我按执行者视角讲，因为用例是写给别人用的。

第一项，**具有可操作性**。执行者拿到这条用例，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。可操作的最低标准是：一个新人在没有你在场的情况下能独立跑完并给出结论。

第二项，**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，执行时会变成"边测边猜"，猜出来的结果没有复现价值。

第三项，**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：一句长句里塞了三个条件，读的人要拆半天。

第四项，**测试目标针对性强**。一条用例只打一个点，不要顺手把别的东西也验一下。针对性强的用例，失败时的信息量最大。

第五项，**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。完备是针对这条用例的目标而言的，不是针对整个功能。

第六项，**没有太多的操作步骤**，与上一条呼应的 7 步口径；再一条，**符合正常业务惯例**——这条容易被忽略：用例的操作顺序要符合真实业务逻辑，不能为了测而测出一些现实中不可能的操作序列，那样的缺陷没有修复价值。

六项里我建议你特别记住"可操作性"这三个字，因为它是**唯一的验收标准**：其他五项做得好不好，都可以用它来检验——执行者能不能不问你、直接跑、给出结论。能，就合格；不能，就还有一项没做到。这相当于给了你一把尺子，而不是六条口号。

> **板书：**六项：可操作性｜信息齐备｜准确清楚｜目标针对性强｜验证点完备而不多｜步骤不多＋符合业务惯例。

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"开发可能直接判为无效。

- 答"按惯例走会漏掉异常"——这个反问很好，但两者不冲突：异常用例也要在合理的业务前提下构造，比如"重复提交"是真实的，而"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，也换不来修复。

> **易错点 / 考点：** ① 六项要能列举，尤其"验证点完备而且没有太多"这种双向表述常出判断题；② "符合正常业务惯例"是最容易漏答的一条，答简答时补上能加分；③ 判断陷阱："信息越多越好"——错，是"所需的各项信息"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）

> 这一页讲颗粒度，也就是用例的粗细。课件用"输入不同的用户名和口令，结果要满足设定的要求"这个例子，把粗和细摆在一起对比，我把它讲透。
>
> 先说**粗颗粒度**。一条用例覆盖多组数据、多个结果：比如"用五组用户名口令组合登录，分别得到对应的五种结果"。好处很明显：用例条数少、执行快、维护成本低——数据变了改一处。坏处也明显：只要其中一组不对，整条判失败，你得手工再跑一遍才知道是哪一组坏了，定位成本高；而且执行者的自由度过大，容易跑偏。
>
> 再说**细颗粒度**。一条用例一组数据一个结果：对应用户名正确密码错误，期望提示"密码错误"。好处是判定唯一、失败可归因、统计精确。坏处是数量爆炸：五组数据变五条用例，五十组变五十条，用例库迅速膨胀，维护成本直线上升。
>
> 那怎么选？我给一个三步判断法。**第一步看判定能不能独立**：如果多组数据的期望结果结构相同、只是取值不同，可以粗；如果期望结果完全不同（有的成功、有的报错、有的锁定），必须细。**第二步看执行成本**：需要长时间环境准备的（比如造历史数据），尽量合并到一条；纯界面点击的，可以拆细。**第三步看复用需求**：这条用例将来要不要单独挑出来跑回归？要，就拆细，因为细的才能被单独调度。
>
> 现实中用得最多的方案是中间路线：**骨架粗、数据细**——一条用例配一张数据表，判定写成"每行数据对应其期望结果"。这就是数据驱动，它同时拿到了粗的维护性和细的可归因性。

> **板书：** 粗（多数据多结果：条数少・维护省・定位难）vs 细（一数据一结果：判定唯一・可归因・数量爆炸）。三步判断：判定能否独立 → 执行成本 → 复用需求。折中："骨架粗、数据细（数据驱动）"。

> **提问：** 一条用例里放五组数据，执行时第三组失败了，你接下来要做什么？

预设回答：

- 答"提缺陷"——太快了。你连是哪一组、错在哪个字段都还没确认，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：一次失败要额外付一次定位成本。

- 答"整条判失败就行了"——错答。整条判失败会让通过率统计失真：明明四组是对的，也被记成失败。

- 收口：颗粒度选择的本质，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 粗细的利弊要成对答，只答一头不得分；② 判断依据三条（判定独立性・执行成本・复用需求）建议背下来；③ 判断陷阱："颗粒度越细质量越高"——错，过细会导致数量失控与维护灾难，课件在整体质量要求里明确要"粒度适中"。

过渡： 单条看颗粒度，整体看四件事，下一页是这一节的收口。

【11.22】整体测试用例的质量要求（2 分钟）

这一页从"一条"跳到"一套"，给四项整体要求：**覆盖率、易用性、易维护性、粒度适中**。这四项是评价一个用例库的核心指标，我逐项讲清判据。

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。注意"依据特定的测试目标"这个限定——覆盖率不是一个绝对数，是相对于目标而言的。目标是把支付主流程测透，和目标是测全所有异常分支，两者的覆盖率算法不同。所以谈覆盖率之前，先问一句"针对什么目标"。

第二项，**易用性**：设计思路清晰、组织结构层次合理，用例操作的连贯性好、执行顺畅。说白了就是"别人用得顺不顺"。判断方法很土但有效：找个没参与设计的人，让他按库里的组织找三条用例并执行，卡在哪，就是易用性的问题在哪。

第三项，**易维护性**：以较少的时间完成用例的维护工作，包括易读性、一致性等。这一项最容易被忽略，因为它不产生即时收益，只在需求变更时一次性爆发。易维护性好的库，改一条需求只动三五条用例；易维护性差的库，改一条需求要翻遍两百条。

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（数据驱动），提高可维护性。这一项其实是前三项的平衡点：太粗覆盖不住场景，太细维护不起。

四项连起来看很有意思：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"。一个库四项都好，才算真的可用。

> **板书：**四项：覆盖率（相对目标而言）｜易用性（找得到·跑得顺）｜易维护性（改一处只动几处）｜粒度适中（覆盖与维护的平衡）。

提问：一个用例库覆盖率很高，但没人愿意用。这算合格吗？为什么？

预设回答：

- 答"算，覆盖率高就是好"——错答。没人用的库，覆盖率再高也是纸面数字，实际执行时大家还是凭经验。

- 答"不算，易用性是硬要求"——对。用例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，这属于易用性和易维护性问题。

- 收口：四项里任何一项塌了，另外三项的价值都会被拖下水；这就是"整体质量"的含义。

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 判断陷阱："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：11.1 讲完"一条用例怎么写"，11.2 开始讲"一堆用例怎么管"。

【11.23】11.2 测试用例的组织和跟踪（1 分钟）

这一页是第二节的开头，先给你路线，因为这一节五个小节的逻辑比第一节更绕。

第一站，**11.2.1 测试用例的属性**。给用例挂上可管理的标签：目标性、所属范围、关联性、阶段性、时效性。属性是后面一切组织工作的地基，属性不全，套件分不出来，跟踪也统计不出来。

第二站，**11.2.2 测试套件及其构成方法**。把用例按目标组合成套件，解决"这一轮该跑哪些"的问题。

第三站，**11.2.3 跟踪测试用例**。执行过程中盯进度：跑了多少、通过多少、卡住多少、为什么卡住。

第四站，**11.2.4 维护测试用例**。需求变了、版本换了，用例怎么跟着走。这一站是整套体系能不能活过三年的关键。

第五站，**11.2.5 测试用例的覆盖率**。用三类覆盖率来检验这套东西够不够。

一句话概括这条路线：**贴标签 → 组套件 → 盯执行 → 跟着改 → 量出来**。它其实就是一个用例库的完整生命周

期，你按这五步去搭自己的课程设计用例库，结构天然就是对的。

> **板书：**五站箭头：属性（贴标签）→ 测试套件（组起来）→ 跟踪（盯执行）→ 维护（跟着改）→ 覆盖率（量出来）。

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。属性不全的直接后果是套件只能按目录硬分，跟踪时统计不出维度。

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

- 答"覆盖率，反正也测不完"（典型错答）——跳过覆盖率，你就无法回答"测够了没有"，验收时只能靠感觉。

- 收口：这五站是链条，跳过任何一站，链条都还在转，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 注意"属性"是组织的前提，考试常以"为什么用例需要属性"来问；③ 判断陷阱："跟踪和维护是测试经理的事，与执行人员无关"——错，执行者填写的执行状态就是跟踪的数据源。

过渡：从地基开始，先看用例该带哪些属性。

【11.24】11.2.1 测试用例的属性（1.5 分钟）

这一页讲属性体系。前面第 12 页我们讲过"元素"，那里是站在一条用例内部看它由什么组成；这一页是站在**管理**的角度看它该挂什么标签。两者角度不同，内容有交叠，但目的不一样。

管理属性要解决三类问题。第一类是**分类问题**：这条用例属于哪个模块、哪个功能、哪一类测试？没有分类，几千条用例就是一堆沙子。

第二类是**筛选问题**：这一轮回归我要跑哪些？答案来自属性筛选——按模块筛、按优先级筛、按类型筛、按阶段筛。筛选能力完全取决于属性填得多细。

第三类是**关联问题**：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。

所以判断属性该不该加，有个很实用的标准：**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（比如优先级只能是高、中、低，不能一会儿写"高"一会儿写"紧急"）。不会，就别加，字段越多填得越糊。

这一页和下一页是一组：这一页给结构，下一页逐个解释其中最要紧的五个属性。

板书：属性三类用途：分类（模块・功能・类型）｜筛选（回归选谁）｜关联（需求・设计・缺陷）。

提问：为什么属性的值域必须统一规范，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**：值域不统一，"高""紧急""P1"混着填，筛选和汇总就全废了。

- 答"为了统一口径，避免各写各的"——对。属性是团队共用的语言，语言不统一，沟通成本会成倍上升。

- 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**可计算的字段**；不可计算的属性，等于没填。

易错点 / 考点：① 属性的三类用途（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 判断陷阱："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：结构立起来之后，看其中最要紧的五个属性怎么解释。

【11.25】一些属性说明（2 分钟）

这一页把五个最要紧的属性逐个解释，我用"它回答什么问题"的方式讲，你记起来会快很多。

第一，**目标性**。课件说得很具体：包括功能性、性能、容错性、数据迁移等各方面的测试用例。这个属性回答的是"这条用例要验证哪一类质量特性"。为什么重要？因为不同目标性的用例，执行条件天差地别：功能用例在普通环境点一点就行，性能用例要加压要监控，容错用例要人为制造故障，数据迁移用例要准备两套库。目标性不标，排期会算错，环境会白搭。

第二，**所属的范围**，也就是它属于哪一个组件或模块。这一项是组织套件时最常用的筛选维度，也是缺陷定位时最快的范围划定——出了缺陷，先看这个模块有多少条用例，就知道这块测得够不够。

第三，**关联性**，和软件产品特性相联系。这条用例验证的是产品的哪个特性？关联立起来，你才能做影响分析：某个特性要改，受影响的用例一查就出来。

第四，**阶段性**，属于单元测试、集成测试、系统测试、验收测试中的某一个阶段。属性写清阶段，能避免一种很常见的混乱：拿系统测试用例去单元测试阶段跑，跑不起来，还以为是环境问题。不同阶段的用例，关注点、环境、执行者都不同。

第五，**时效性**，不同的版本所适用的测试用例可能不相同。这一项最容易被当成废话，其实它最狠：它承认用例是有寿命的。某个版本删掉的功能，它的用例就该下线；某个版本新增的开关，就该有新用例。时效性属性写不清，用例库就会越积越脏。

板书：五属性：目标性（功能・性能・容错・数据迁移）｜所属范围（组件/模块）｜关联性（产品特性）｜阶段性（单元・集成・系统・验收）｜时效性（适用版本）。

提问："时效性"为什么必须作为属性存在？不标会有什么具体问题？

预设回答：

- 答"不标就不知道这条用例还能不能用"——正中要害。执行者会拿过期用例去跑，跑不过就报缺陷，最后发现功能早就删了。

- 答"版本多了自然会乱，标了也没用"——错答。正因为版本会多，才需要标；不标是听任它烂，标了至少能筛。

- 答"删掉旧用例就行了，不用标时效"——方向可行但太粗暴：有些功能是暂时下线、下个版本又回来，删了就得重写。标时效比删更经济。

- 收口：时效性属性是用例库的"保质期标签"，没有它，你无法判断库里哪些还能吃。

易错点 / 考点：① 五个属性名称与各自的含义要能对上，尤其"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 判断陷阱："一条用例可以同时属于多个测试阶段"——严格说，一条用例的主要适用阶段应唯一，跨阶段复用要重新确认判定标准。

过渡：属性齐了，接下来把它们组合起来，这就是测试套件。

【11.26】11.2.2 测试套件及其构成方法（2 分钟）

这一页开场就给了套件存在的理由：**建立合适的、可扩展的测试用例框架，有效地组织众多的测试用例，包括对测试用例的分类、清晰的层次结构等。**这句话里有两个词要点出来——"可扩展"和"层次结构"。

先说"众多"。为什么一定要组织？因为数量。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

铺在一张表里，你没法回答"这一轮跑哪些"，只能靠搜索和记忆，效率极低、还容易漏。

再说"可扩展"。框架不是为今天这一版设计的，是为将来第一百版设计的。判断一个组织方式好不好，有个简单测试：**新增一个模块，你要改多少东西？** 如果只是新增一个分支、其余不动，这个框架是可扩展的；如果每加一块都要重排整棵树，那它撑不过三个版本。

再说"层次结构"。层次是把"多"变成"可管理"的核心手段：项目→模块→功能→用例，逐层收敛。层次还有一个隐含好处——**统计可以按层汇总**。你想知道支付模块的通过率，按层聚合就行，不用手工筛。

最后是课件的那个等式：**Test Case → Test Suite**。用例是叶子，套件是枝干。这个等式说明套件不是新东西，它是用例的组合方式；也说明套件是可以嵌套的：套件里还可以有子套件。

补一句为什么"可扩展"这么关键。用例库的成本大头不在第一次编写，而在第 N 次改动。框架设计得好，第 N 次改动只是加一个节点、改一个字段；设计得差，第 N 次改动要重排整棵树，还得逐条核对着改。所以判框架好坏，本质上是在判"**未来改动的代价**"，而不是判今天画得整不整齐。

> **板书：**理由："众多"（数量）＋"可扩展"（撑得住版本）＋"层次结构"（可汇总统计）。等式：Test Case → Test Suite（叶子 → 枝干）。

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"（典型错答）——不是。漂亮不等于好用，很多"漂亮"的树是按组织架构画的，谁负责谁一块，换个人就全乱。

- 答"看新增一个模块要改多少地方"——很好的判据，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；但要注意，检索靠属性和命名，不完全靠层级。

- 收口：好框架的两个硬指标：**加东西不用重排，找东西不用翻遍**。

易错点 / 考点：① 套件的存在理由三条（数量・可扩展・层次）要能复述；② Test Case → Test Suite 的从属关系常出填空；③ 判断陷阱："套件只能有一层"——错，套件可嵌套，套件里可以含子套件。

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）

这一页是组织结构的示例，你要从里面看出三件事：**层次怎么分、命名怎么定、叶子挂在哪**。

第一，层次的划分依据。常见做法是按"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，区别在于你更常按哪个维度筛选。判据很简单：**你最常按哪个维度问"跑哪些"，那个维度就该在上层**。

第二，命名规则。节点的名字要能自解释，不要出现"其他""杂项""临时"这种垃圾桶节点。垃圾桶节点是所有用例库腐烂的起点：一旦有了它，所有不好归类的都会进去，半年后没人敢打开。

第三，叶子挂在哪里。一条用例只能有一个主归属位置，其余维度靠属性表达。为什么强调"一个主归属"？因为多处挂载会导致同一条用例被改两次、被统计两次，数字立刻失真。

还有一层：这一页的组织结构会和属性配合使用。树负责"我在哪"，属性负责"我是什么"。只用树，筛选维度不够；只用属性，人找不到入口。两者配合，才是完整的组织方案。

你们做课程设计时，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来没分类，然后被迫大改。

> **板书：**三看：层次依据（最常用的筛选维度放上层）｜命名（不许"其他/杂项/临时"垃圾桶）｜叶子（一个主归属＋属性表达其余维度）。

提问：为什么"其他"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，成为整个库的沉积层。

- 答"以后找东西找不到"——对。而且更糟的是，它让分类工作失去了约束：既然有"其他"，设计时就不必想清楚该归哪。

- 答"有了它更方便，先扔进去以后再说"——这就是典型错答，也是现实中绝大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；分类的纪律一旦破，就再也立不起来。

易错点 / 考点：① 层次划分依据是"最常用的筛选维度"，这句话可以直接当简答答案；② 一个主归属＋属性表达其余维度，是避免重复统计的关键；③ 判断陷阱："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，有几种构成方法。

【11.28】测试集/测试套件（2 分钟）

这一页给定义加三种组织方式，是 11.2.2 的核心页，我讲细一点。

先看定义：**测试套件是由一系列测试用例，并与之关联的测试环境组合而构成的集合，以满足测试执行的特定要求。**这个定义里有两个成分不能漏：一是**一系列测试用例**，二是**与之关联的测试环境**。很多同学只记住了前半句，把套件当成"用例的文件夹"，这是不准确的——套件是"用例＋环境"的打包，因为同一批用例在不同环境下执行，本来就是两件事。

定义的后半句更重要：**通过测试套件，将服务于同一个测试目标、特定的测试任务或某一运行环境下的一系列测试用例有机地组合起来。**注意三个"同一"：同一目标、同一任务、同一环境。这三个"同一"就是套件的边界判据——一条用例如果和这个套件的目标不同，它就不该进来，哪怕它属于同一个模块。

然后是三种构成方法。**第一种，按程序功能模块组织**：登录、下单、支付各成一套。优点是直观、好找、和开发的分工对得上；缺点是跨模块的场景（比如"下单后支付失败再退款"）不好归。

第二种，按测试用例的类型组织：功能、性能、兼容性、安全性各成一套。优点是同类用例的执行条件一致，便于批量准备环境；缺点是不便于按业务切块交付。

第三种，按测试用例的优先级组织：高优先级一套、中低各一套。优点是应对"时间不够"极其高效——先跑高优先级那一套，跑完就有个可交付的结论。

这三种不是互斥的，实际项目里通常是**组合使用**：先按模块建大树，再在模块内按优先级切套件，跨模块的场景单独建类型套件。这也解释了为什么属性必须填全——三种切法都靠属性筛。

板书：定义两成分：用例集 ＋ 关联环境；边界判据：同一目标・同一任务・同一环境。三种组织：① 按功能模块 ② 按用例类型 ③ 按优先级（可组合）。

提问：按功能模块组织的套件，遇到跨模块的业务场景，你会怎么处理？

预设回答：

- 答"塞进最相关的那个模块"——可行，但要标注关联的其他模块，否则影响分析会漏。

- 答"单独建一个'跨模块场景'套件"——很实用的做法，实践中很常见，代价是要维护一条清晰的命名规则。

- 答"跨模块的就不管了"——错答。跨模块场景恰恰是缺陷密度最高的地方，漏掉它等于漏掉最值钱的一块。

- 收口：组织的目标不是"每一条都有归属"，而是"每一条都能被找到、被执行、被统计"。

易错点 / 考点： ① 定义里"用例＋环境"两个成分必须都答到，只答用例会丢分；② 三种组织方式的优缺点要成对说；③ 判断陷阱："三种组织方式只能选一种"——错，实际工程中组合使用是常态。

过渡： 三种组织方式落到一张图上，就是套件与测试计划的关系。

【11.29】测试套件（Test suite）（1.5 分钟）

这一页把套件放进整个测试计划的层级里，你要看出的是**目标驱动**这四个字。

课件这一页给的是三层从属关系：**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**。列出的目标有四个：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，各有自己的套件。

这个层级为什么这么排？因为**套件不是按目录长的，是按目标长的**。你这一轮想达成什么，决定你该组哪个套件。目标换了，套件就该重新组；而不是"套件建好了，永远就那一套"。

举个具体场景：这一版上线前，你的测试计划里同时有四个目标——集成阶段要验模块之间的接口，新功能阶段要验这一版新增的三个功能，性能阶段要验下单链路的承载能力，回归阶段要验老功能没被改坏。四个目标对应四组套件，执行顺序、环境准备、判定标准都不一样。你要是只有一套"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。

这就是为什么教材要把"测试计划—目标—套件"画成三层：**上一层定义要什么，下一层回答怎么给。** 套件是目标的落地形态。

顺带说一个实操建议：套件的命名里最好带上目标，比如"V2.3 回归-支付"，这样执行者一看就知道这个套件为谁服务、什么时候该跑。

板书： 三层：测试计划 → 测试目标（集成・新功能・系统性能・功能回归）→ 测试套件。旁边一行："套件按目标长，不按目录长。"

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"——是问题之一，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，没有目标切分，结论就无法交付给不同的人。

- 答"有全集就够了，需要的时候手工挑"——错答。手工挑就意味着每次执行都不一样，回归的可比性就没了。

- 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"是本页最核心的结论；③ 判断陷阱："一个测试计划只能有一个测试套件"——错，一个计划下按目标可以有多个套件。

过渡：目标之外，还有一个更细的切法——按测试类型切，这是下一页的内容。

【11.30】测试类型与测试用例设计（2 分钟）

这一页给的是两种切法的对照：**根据测试类型设计**，和**根据程序功能模块设计**。课件列了一长串测试类型，我先把它们分成几组，你才好记。

第一组是**质量特性类**：功能测试、易用性测试、性能类（课件列了压力测试）、界面测试。它们关注"这个软件在某一类质量上够不够"。

第二组是**环境与配置类**：配置测试、国际化测试。它们关注"换一套环境还对不对"。

第三组是**生命周期类**：安装／卸载测试、软件更新测试、联机注册测试。它们关注"软件从装到卸这条路上的每个动作"。

第四组是**配套内容类**：文档测试、联机帮助测试、文件操作测试、数据备份测试。它们关注"软件周边的东西对不对"，这一组最容易被漏，也最容易出问题——很多产品的差评不是功能错，是帮助文档和实际行为不一致。

第五组是**回归测试**，它本身不是一个"类型"，而是一种"重新执行"的组织方式，这个区别要分清。

现在说两种切法的关系。按类型切，好处是同类用例执行条件一致、便于批量准备；坏处是同一个模块的用例被拆散在多个类型里，按模块交付时不好凑。按模块切，好处是直观、和开发分工一致；坏处是同一类测试条件（比如都要国际化环境）要在多个模块里重复准备。

> 工程上的解法是**二维矩阵**：树按模块长，属性按类型标，需要时用筛选生成"类型视图"。这样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。

> **板书：**五组类型：质量特性（功能・易用・性能/压力・界面）｜环境配置（配置・国际化）｜生命周期（安装卸载・更新・联机注册）｜配套内容（文档・联机帮助・文件操作・数据备份）｜回归（是方式不是类型）。二维矩阵：树按模块，属性按类型。

> **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，它针对的是"改动有没有破坏已有功能"。

- 答"回归也是一种类型，和其他并列"——错答。回归用例的内容来自功能、界面、配置等各类，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：分类维度要分清"测什么"和"为什么现在测"，回归属于后者。

> **易错点 / 考点：**① 课件列出的类型名称要能列举，答题时按五组归类比逐条硬背更稳；② 回归的性质（组织方式而非类型）是判断题高频点；③ 判断陷阱："按类型组织和按模块组织只能取其一"——错，用属性做二维视图可以兼顾。

> **过渡：**组织方式定下来之后，还要和测试过程对齐，否则库和流程会两张皮。

【11.31】测试用例的组织和测试过程的关系（1.5 分钟）

> 这一页讲的是"库"和"流程"的咬合，这是很多团队做不好的地方：用例写得不错，流程也跑得挺顺，但两者是两张皮，谁也不喂谁。
>
> 先把测试过程的主干摆出来：计划 → 设计 → 执行 → 缺陷 → 回归。用例组织要在每个环节都能被用到，否则它就是个摆设。
>
> 在**计划**环节，用例库提供分母：这一轮要跑多少条、需要多少人天，靠套件和估计用时算出来。没有库，计划只能拍脑袋。

在**设计**环节，用例库提供基线：新功能照着同类老用例的骨架写，省一半时间，还能继承历史踩过的坑。

在**执行**环节，用例库提供清单：按套件取用例，逐条记录状态，进度就是"已执行/总数"。

在**缺陷**环节，用例库提供坐标：缺陷单关联用例编号，一看就知道是哪条用例发现的、当时什么条件。将来这条用例就是回归的必跑项。

在**回归**环节，用例库提供弹药：改动影响到的模块筛出来，历史出过缺陷的用例优先跑。

五个环节串起来，你会看到一个规律：**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，缺陷修复后又长出新的用例。它不是流程旁边的一个文档，它是流程里的一个节点。这一页的图要你看出的，就是这个"节点"关系。

板书：流程五环：计划（分母）→ 设计（基线）→ 执行（清单）→ 缺陷（坐标）→ 回归（弹药）。右侧："用例库既是输入，也是输出。"

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。缺陷和用例断开，你就无法回答"这条缺陷当时是怎么测出来的"，也无法精准回归。

- 答"回归时不知道该跑哪条"——对，这是最直接的损失：修复验证只能靠人回忆当时的操作。

- 答"用例库和缺陷库本来就该分开"——错答。分开管理可以，但关联必须建立，否则两套数据各自为政，谁也证明不了谁。

- 收口：关联不是形式，是把"一次偶发"变成"可复现资产"的唯一手段。

易错点 / 考点：① 五个环节与用例库的对应关系（分母・基线・清单・坐标・弹药）建议背；② "用例库既是输入也是输出"是本页的核心判断；③ 判断陷阱："用例只在执行阶段有用"——错，计划、设计、缺陷、回归各环节都在用。

过渡：咬合关系讲清了，最后给六个具体应用场合，这是套件最实用的部分。

【11.32】测试套件应用场合（2 分钟）

这一页给六个场合，全是工程里的真实场景，我一条一条讲"什么时候用、为什么这么用"。

第一条，**只是部分功能模块发生了变化，就可创建由这些改动模块的测试用例构成的测试套件**。这是套件最日常的用法：改动影响分析之后，把受影响的用例挑出来组一套，跑完就有结论。它的前提是你的用例按模块组织得好，否则挑不出来。

第二条，**在修改的模块中，也不需要选择所有的测试用例，针对不同的优先级创建不同的测试套件**。第一层按模块筛完，第二层按优先级再筛。为什么？因为时间永远不够，改动模块里也分核心路径和边缘路径，先跑高优先级那套，至少核心风险被兜住了。

第三条，**测试执行的第一阶段可以创建一个特定平台上的测试套件**。第一阶段通常是冒烟或基础验证，这时候要的是"这台机器上能不能跑通"，所以按平台组套件最合手。

第四条，**有必要为自动化测试、手工测试分别建立测试套件**。这一条非常实用：自动化套件要能无人值守、可重复、判定可机读；手工套件可以有主观判断和探索性内容。混在一起，自动化跑不动，手工也跑不顺。

第五条，**可建立和测试人员相对应的、不同平台或模块的测试套件**。按人分组有争议，但在人手分工明确的团队里很有效——每个人打开自己的套件，就知道今天干什么。要注意的是：按人分组是执行视图，不是资产归属视图，人走了资产还在库里。

第六条，**回归测试中，可以先运行曾经发现缺陷的测试用例，然后再运行从来没有发现过缺陷的测试用例**。这条最有含金量：历史出过缺陷的地方，再次出缺陷的概率显著更高，所以先跑它们。这是把"缺陷历史"当成了排序依据。

> **板书：**六场合：① 改动模块成套 ② 改动模块内再按优先级切 ③ 第一阶段按平台建 ④ 自动化/手工分开 ⑤ 按人员/平台/模块建（执行视图）⑥ 回归先跑历史有缺陷的，再跑没缺陷的。

提问：第六条"先跑出过缺陷的用例"，依据是什么？这样做有没有风险？

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，风险天然更高。

- 答"没依据，就是习惯"——错答。这是基于风险排序的典型做法，缺陷历史就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、新链路的用例仍然要跑，只是

排序靠后。

- 收口：套件的排序本质是**风险排序**，而缺陷历史是最廉价可靠的风险信号。

> **易错点 / 考点：**① 六个场合要能各说一句，简答题按点给分；② 第四条（自动化与手工分开）和第六条（回归排序）是最实用的两条，答题时优先写；③ 判断陷阱："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，资产归属应在库里。

> **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）

> 这一页讲执行阶段的度量，课件给了四个问句，我认为这四个问句就是一个测试负责人的日常。
>
> 第一问，**跟上进度了吗**？进度不是感觉，是"已执行数 ÷ 计划执行数"。这个数每天都要更新，否则等到交付前三天你才发现只跑了一半，那时做什么都晚了。
>
> 第二问，**测试人员每天能执行多少个测试用例**？这个数字叫日执行率。它的用途有两个：一是用来做计划，知道这一轮需要几个人几天；二是用来发现异常——如果某人一天的执行量突然腰斩，要么是用例写得太粗、执行时不断卡壳，要么是环境不稳、反复重试。日执行率是一个很灵敏的"体感温度计"。
>
> 第三问，**"通过、未通过以及未测试的"各占多少**？这三个数必须分开统计，不能混成一个"完成率"。因为它们的含义完全不同：通过代表已验证，未通过代表有缺陷待修，未测试代表风险还在。一个"完成 80%"的进度里，如果未通过的占 30%，那这个 80% 一点都不值得高兴。
>
> 第四问，**不能被执行的原因是什么**？这一问最见功力。不能执行的原因通常有四类：环境没准备好、前置数据造不出来、相关功能还没提测、用例本身已失效。四类原因的处理责任完全不同——前三类找环境或开发，第四类要改用例。你不分类统计，就只会得到一句"跑不动"，然后没有人负责。
>
> 再说课件那张曲线：横轴是时间，纵轴是工作量，也就是需要执行的用例数，理论目标是一条爬升到 100% 的线。真实的执行曲线很少是直线：开头慢，因为环境在搭、数据在造；中段快，进入状态；末段又慢下来，因为剩下的都是难的、阻塞的、依赖别人的。**看曲线要看尾段**，尾段收不上去，说明阻塞问题没有被及时清掉。

> **板书：**四问：进度（已执行/计划）｜日执行率（排期＋异常预警）｜通过/未通过/未测试（分开统计）｜

> 不可执行原因（环境・数据・未提测・用例失效）。曲线：纵轴工作量、横轴时间，目标 100%，尾段最要盯。

> **提问：**"完成率 80%"这个数字，什么情况下你不能采信？

> **预设回答：**

- 答"未测试的还很多的时候"——对，完成率的分母如果只算了已排期的用例，实际风险可能被低估。

- 答"未通过的占了很大比例的时候"——也对。未通过意味着缺陷未修，功能不可用，这个进度不能算"完成"。

- 答"完成率就是完成率，怎么会不能采信"——典型错答。一个不区分状态、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，尤其"不能执行的原因"这一问最常被漏；② 三个状态（通过・未通过・未测试）必须分开统计；③ 判断陷阱："执行进度就是看已执行用例数"——错，还要看通过率与阻塞情况。

> **过渡：**跟踪解决"现在怎么样"，下一个问题是更长线的事——用例怎么跟着软件一起变。

【11.34】11.2.4 维护测试用例（2 分钟）

> 这一页只有一句话：**测试用例的维护是持续改进的过程**。这句话短，但它是整章最重的一句，我拆成四件事讲。
>
> 第一件，**什么时候要维护**？触发点有五类：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。你要建立一个意识：维护不是"有空再做"，而是有触发就必须动。没有触发机制的团队，用例库的死亡方式是缓慢而确定的——先是几条例外用不了，然后是一批，最后整库没人信。
>
> 第二件，**维护要做哪几种动作**？不只是"改"。至少四种：**更新**（判定或数据变了）、**新增**（新需求、新发现的缺陷回填）、**废弃**（功能删了，用例下线但不删记录）、**归档**（旧版本用例留档，备查）。这四种动作要分清，尤其"废弃"和"归档"——直接删掉是最糟的，因为你失去了"这里曾经测过什么"的历史。
>
> 第三件，**维护的成果怎么固定下来**？靠**基线**。在一个里程碑节点，把当前用例库冻结成一个版本，记下版本号和变更说明；下一轮在这份基线上改。基线的作用是让"改坏了"可以回滚，让"当初是怎么测的"可以被追溯。

第四件，**谁来维护**？常见做法是：用例作者负责自己写的，模块负责人负责本模块整体，测试负责人负责基线与评审。责任要落到人，落到"这一块归谁"，否则"大家一起维护"等于"没人维护"。

你把四件事凑起来看，"持续改进"就不是一句口号了：它是有触发、有动作、有基线、有责任人的一套机制。

板书：四件事：何时（需求・缺陷・评审・环境・失效）→ 做什么（更新・新增・废弃・归档）→ 怎么固定（基线＋版本号＋变更说明）→ 谁负责（作者・模块负责人・测试负责人）。

【思政落点 3】：维护测试用例是最没有掌声的工作：改十条用例没人看得见，不改却在半年后以"漏测"的形式爆出来，而那时已经找不到责任人。这一节要讲的态度很朴素——**对变更负责**。需求变了，你把用例改到位，是对同事和用户负责；改不动就如实标注"此条已失效"，也比含糊留着强。工程里最伤人的不是能力不足，是明知不对却糊弄过去。

提问：功能被删掉了，对应用例该怎么处理？直接删掉行不行？

预设回答：

- 答"删掉，留着占地方"——错答。删掉就丢了历史：将来功能回来要重写，出问题也查不到当初怎么测的。

- 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"——最危险的一种。库里留着一条永远跑不过的用例，会让通过率长期失真，最后没人相信统计数字。

- 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，三者必须有一个明确答案。

易错点 / 考点：① 维护的四种动作（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）是简答加分点；③ 判断陷阱："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。

过渡：机制讲完了，看一张把机制画成流程的图。

【11.35】测试用例维护流程示例（1.5 分钟）

这一页把上一页的机制画成了一条流程，我们顺着走一遍，重点看每一步的**输入和产出**。

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。触发源必须明确，否则流程永远启动不了——很多团队的维护工作停在"知道该改"，就是因为没有一个正式的触发入口。

第二步是**影响分析**：这次变更影响哪些功能、哪些用例？靠什么找？靠追溯关系和模块属性。这一步做得好不好，直接决定维护成本是"改五条"还是"改五十条"。

第三步是**动作执行**：更新、新增、废弃，三选一或组合。这一步要注意同步：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，都要跟着动。只改用例不改关联，等于留了个断点。

第四步是**评审**：改完要有人看。评审的重点不是文字，是"判定还成立吗"——需求变了，原来那条判定可能已经不成立了，这必须由懂业务的人确认。

第五步是**入库与打版本**：把改动固化，形成新的基线，写清变更说明。这一步是很多人省略的，也是最不该省的。

第六步是**回归验证**：改动后的用例要实际跑一遍，确认它还能执行、判定还准。改了没跑的用例，等于没验证过的代码。

最后一步是**度量反馈**：这一轮改了多少条、新增多少条、废弃多少条，这些数字要回到覆盖率指标里去。流程在这里闭环，也在为下一轮触发积累依据。

> **板书：**七步：触发 → 影响分析（追溯・模块属性）→ 动作（更新・新增・废弃）→ 评审（判定是否仍成立）→ 入库打版本（基线＋变更说明）→ 回归验证 → 度量反馈 →（闭环）。

> **提问：**这条流程里，哪一步最容易被省略？省略的后果是什么？

> **预设回答：**

- 答"评审，大家觉得自己改的没问题"——很常见。后果是判定悄悄失真，等到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，而且更危险：改错了的用例会一直错下去，还会误导统计。

- 答"入库打版本，反正文件在那儿"（典型错答）——这一省，回滚能力就没了，将来谁也说不清某条用例是什么时候、为什么改的。

- 收口：三步都不能省，因为它们分别守住了**正确性、有效性、可追溯性**。

> **易错点 / 考点：**① 流程七个环节的顺序；② "影响分析靠追溯"这句是简答要点；③ 判断陷阱："用例改完不需要再执行"——错，改动后的用例必须回归验证。

> **过渡：**机制和流程都讲了，接下来要回答一个更硬的问题——这套东西到底够不够？

【11.36】11.2.5 测试用例的覆盖率（2 分钟）

这一页给三类覆盖率，它们是检验整套体系的三把尺子，我按"看自己、看需求、看代码"来讲。

第一把尺子，**测试用例本身**：**发现缺陷后补充的测试用例数 ÷ 总的测试用例数**。这个指标衡量的是"用例库有没有从缺陷中学习"。举个直观的例子：这一轮发现 20 个缺陷，其中有 12 个是原有用例没覆盖到的，于是你补了 12 条用例；如果总用例数是 600，这个比值就是 2%。这个数字不是越高越好，也不是越低越好——它反映的是**用例设计的盲区有多大**。持续偏高，说明前期设计太粗；持续为零，反而要怀疑：是不是没人认真回填？

第二把尺子，**需求、功能点覆盖率**：每个功能点都要有测试用例，课件给了一个经验口径：**平均大于 4 个每功能点**。为什么要有这个量？因为一个功能点只有一条用例，基本只能覆盖正常流程；要覆盖边界、异常、组合，四条左右才勉强够。这个指标的价值在于它能一眼暴露"漏测区域"：某个功能点下一条用例都没有，那就是明摆着的风险。

第三把尺子，**代码覆盖率**：借助工具跟踪执行过程，了解代码行、类、方法等覆盖率情况。这一把尺子最客观，因为它不看你怎么写用例，只看程序实际跑了哪些代码。但它有个陷阱：**覆盖率高的测试不一定有效**。你可以把所有代码都执行一遍，却一个断言都不做——代码覆盖了，缺陷一个没抓到。所以代码覆盖率要结合断言数量、缺陷发现情况一起看。

三把尺子合起来才是完整判断：用例自身看学习能力，需求覆盖看有没有漏块，代码覆盖看有没有漏路。三者互补，单独看任何一把都会误判。

> **板书：**三把尺：① 用例自身＝缺陷后补用例数/总用例数（学习能力・盲区大小）② 需求功能点覆盖＝每功能点都要有用例，平均 >4 个/FP ③ 代码覆盖＝工具跟踪行/类/方法（客观，但需配合断言）。

提问：代码覆盖率达到 90%，能不能说明测试做得不错？为什么？

预设回答：

- 答"能，90% 已经很高了"——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，没有断言的执行只是"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，尤其第一类的分子分母别记反；② "平均 >4 个每功能点"是课件给的经验口径，选择题常考；③ 判断陷阱："代码覆盖率越高测试质量越好"——错，覆盖率与有效性不是一回事。

过渡：三把尺子量的是结果，最后我们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）

这一页把前面所有零件装成一个整体，课件给的目的很明确：**更有效、有序地组织测试**。我按"骨架、血肉、规则、承载"四层讲。

骨架是层次结构：项目—模块—功能—用例，或者按版本—目标—套件。骨架决定"东西放在哪"，也决定统计怎么汇总。

血肉是属性体系：目标性、范围、关联性、阶段性、时效性，加上优先级、类型、估计用时。属性决定"东西怎么被筛出来"。骨架和属性是一对：骨架管位置，属性管标签；只有骨架，筛选维度不够；只有属性，人找不到入口。

规则是模板与命名：模板规定字段，命名规定格式（编号规则、标题写法、套件命名）。规则的价值在于**一致性**——一致才能被评审、被检索、被工具处理。

承载是工具与追溯关系：用例放在哪个系统里、和需求怎么关联、和缺陷怎么关联、变更怎么留痕。没有工具承载，几千条用例靠文档管理，维护成本会高到没人愿意碰。

四层之外还有一条隐线：**维护规则**。谁改、什么时候改、怎么评审、怎么打基线。这条隐线不写在框架图上，却决定框架能活多久。

一句话总结这一页：框架不是一张目录树，是"骨架＋属性＋规则＋承载＋维护"的组合。你在课程设计里搭自己的用例库，按这五样搭，结构天然站得住。

板书：四层＋一隐线：层次结构（骨架）｜属性体系（血肉）｜模板与命名（规则）｜工具与追溯（承载）｜维护规则（隐线）。目的："更有效、有序地组织测试。"

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，你无法生成"高优先级回归套件"，也无法统计某个类型的覆盖情况。

- 答"写用例的时候"（典型错答）——也有道理：没有属性约束，作者不知道该填什么，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＋维护规则；② "骨架管位置、属性管标签"这句是简答要点；③ 判断陷阱："框架就是目录结构"——错，目录只是骨架一层。

过渡：整章讲完了，我们用两页收口：先小结，再思考。

【11.38】本章小结（2 分钟）

这一页四条小结，我按"一条—一套—一棵树—一条命"的顺序串，你跟着这个骨架回忆全章。

第一条，**软件测试用例的基本属性、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"这条用例好不好管"：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。

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。一个测试计划下面有若干子目标，每个子目标对应一组套件。这条小结把第 26 到 32 页全串起来了。

第三条，**测试用例框架、结构：层次关系、依赖关系等**。层次关系解决"放在哪"，依赖关系解决"先跑谁"。依赖关系最容易被忽略：有些用例必须先执行 A 再执行 B，比如"先下单再支付"，这个顺序必须显式标注，否则套件乱序执行就会大面积假失败。

第四条，**测试用例评审、维护和持续改进**。这一条是本章的价值落点：用例不是写完就完，它要评审、要跟着版本走、要在缺陷中学习。你把这四条记住，本章的骨架就立住了：**一条要合格，一套要成目标，一棵树要有层次与依赖，一条命要靠评审与维护续上。**

最后再点一次那个容易漏的概念——**依赖关系**。层次关系管的是"放在哪"，依赖关系管的是"先跑谁"。现实里大量用例是**有顺序的**：先注册再登录、先下单再支付、先造历史订单再测退款。这类顺序必须显式写进框架，否则套件一乱序执行，就会得到一片"假失败"，团队会花几天时间去追一个根本不存在的缺陷。

> **板书：**四条：基本属性＋扩展属性（一条合格）｜按子目标建 Test Suite（一套成目标）｜框架：层次关系＋依赖关系（一棵树）｜评审・维护・持续改进（一条命）。

提问：如果只允许你从本章带走一句话，你会带哪一句？

预设回答：

- 答"测试用例要有编号和优先级"（典型错答）——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"——这句概括得好，把用途和性质都说到了。

- 答"写完还要维护"——朴素但很准。行业里绝大多数用例库不是死于写得差，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**——两句话，把设计和维护都覆盖了。

易错点 / 考点：① 四条小结要与正文对应，考试常以"本章学了什么"的形式考；② "依赖关系"是最容易漏的一条，答题时补上能加分；③ 判断陷阱："用例评审只在新写时做"——错，变更与定期复审同样需要评审。

过渡：最后三个思考题，正好检验你是不是真的把这章用起来了。

【11.39】思考题（2 分钟）

三个思考题，我先不急着给答案，先把答题的抓手说清，你们课后按抓手展开。

第一问，**在构造测试套件中，哪种方法是最常用的**？抓手是三种组织方式各自的适用条件。按功能模块组织是最常用的一种，因为它的直观性和与开发分工的匹配度最高；实践中通常以它为主干，再叠加按优先级切分和按类型筛选。答题时别只答一个名字，要说明"为什么最常用"和"其他两种什么时候用"，这才完整。

第二问，**如何有效地维护测试用例**？抓手就是我刚讲过的四件事：触发机制、动作分类（更新・新增・废弃・归档）、基线与版本、责任人。再加两条加分项：一是把缺陷回填成用例，让库从缺陷中学习；二是定期复审，主动清理过期用例，而不是等它跑不过了才发现。

第三问，**有什么工具可以来度量测试用例的覆盖率**？这个问题要分两类答。第一类是**代码覆盖率工具**：Java 生态常见的 JaCoCo、Cobertura，C/C++ 有 gcov／lcov 一类，前端 JS 有 Istanbul／nyc，这类工具能给出语句、分支、方法、类的覆盖情况。第二类是**用例与需求管理工具**：它们能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，用来算需求覆盖率和进度。答题时要把两类分开说，别只答一类。

三个问题连起来，其实是本章的三条主线：怎么组织、怎么维护、怎么度量。你把它们答顺了，这一章就是

真会了。

再给一个答题技巧：这三问都属于"方法类"简答，评分看的是**要点齐不齐**，不是文采好不好。所以作答时先列点、再各补一句解释，别写成一大段叙述——阅卷的人找要点，找的是结构。

板书：三问三抓手：① 最常用＝按功能模块（主干）＋优先级／类型叠加 ② 维护＝触发・动作・基线・责任人＋缺陷回填 ③ 覆盖率工具＝代码类（JaCoCo・Cobertura・gcov・Istanbul）＋用例需求管理类。

提问：为什么"按功能模块组织"会成为最常用的方式？它赢在哪里？

预设回答：

- 答"因为开发就是按模块分工的"——这是最实在的原因：用例的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，跨模块场景和类型维度要靠属性与专门套件补上。

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"两类，只答一类会失分；② 第一问答"按功能模块"要给理由，光写名字拿不到全分；③ 判断陷阱："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：这一章就到这里，最后我把带走的三件事收一下。

【11.40】感 谢 聆 听（1 分钟）

第 11 章讲完了。我不做总结陈词，只留三个动作给你，出了这门课就能用。

第一个动作：**把一条用例写清**。挑你课程设计里最熟的一个功能，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，然后找一个没参与的同学照着跑。他跑不出你的结论，就回去改——改到他能跑出来为止。这一条练的是表达。

第二个动作：**把一组用例组织起来**。给这组用例编号、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，我跑哪一套？答得出来，说明组织是有效的。这一条练的是管理。

第三个动作：**给它留一条命**。写下"什么情况下这条用例要改"，并在下一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。你把证据留清楚，三年后接手的人不必感谢你，他只要能干活就行——这就是我们这一章想要的职业水准。
>
> 谢谢大家。

> **板书：**三个动作：写清一条（表达）→ 组织一组（管理）→ 留一条命（责任）。底部一句："用例是证据；留下证据的人，才听得见缺陷说话。"

提问：课后请回答一句：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"（典型错答）——方向对，但要具体：全而散的文档没人读，结构化、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东西，才叫资产；用不起来的，只能叫记录。

易错点 / 考点：① 本章的收口三动作（写清・组织・维护）可作为课程设计自检清单；② 期末若考综合题，"用例是资产"这个论点要能用"可复用、可度量、可追溯、可传承"四个词支撑；③ 判断陷阱："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：本章结束。下一章进入测试的收尾环节——缺陷管理与测试报告，把这一章积累的用例与执行数据真正变成可交付的质量结论。

四、思政落点清单（共 3 处）

序号	所在页	落点主题
1	11.9	把测试用例当"知识资产"看，是职业成熟度的分水岭。有的团队把用例当交差材料，评审
2	11.19	写用例最见功夫的地方不是会不会用工具，而是表达够不够严谨。一句"输入非法密码"看

3 11.34 维护测试用例是最没有掌声的工作：改十条用例没人看得见，不改却在半年后以"漏测"的