《软件测试课程设计》讲稿（第 5 章）

第 6 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 5 章　单元测试与集成测试

第 6 周 · 2 学时（90 分钟） | 教材：朱少民《软件测试方法和技术（第 4 版）》

《软件测试课程设计》第 5 章 讲稿（第 6 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 5 章 单元测试与集成测试 —— 从"写第一行测试代码"到"把系统拼起来还能跑"
>
> 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 2 篇 软件测试技术（本章 = 书页 100 起）
>
> 配套课件：逐页还原版 **100 页**（`lecture5/`，云端 http://111.231.0.226:2345/lecture5/index.html）
>
> 教案对应：第 6 次课「单元测试与集成测试」
>
> 本讲重点：代码评审的价值与优秀实践（互查、检查表、速度与规模控制）、静态分析工具链（SonarLint／FindBugs／Checkstyle／SourceMonitor）、单元测试的目标与"驱动程序＋桩程序"、JUnit 5 的组成与关键组件（Annotation／Assert／Runner）、Mock 家族（Stub／Dummy／Spy／Fake／Mock）与 Mockito、覆盖率工具 JaCoCo 的原理、集成策略（自顶向下／自底向上／混合）、微服务下的契约测试（CDC）与 CI/CD 流水线中的持续测试

> 本讲难点：Mock 与 Stub 的取舍（"够用就行"）；契约测试为什么能把联调成本降到零；覆盖率数字与测试充分性的关系；CI/CD 流水线中各道质量门禁的先后与作用
>
> **本讲稿为可照读完整版**：每页含【口播（可直接朗读）】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为时间紧张时可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出代码评审的四类关注点（规范与风格、圈复杂度 e−n+2p、信息流复杂度、耦合度）与互查的三条硬指标（每次 <200–400 行、300–500 LOC/h、单次 ≤60–90 分钟）；② 能列举静态分析工具及其定位（SonarLint／FindBugs／Checkstyle／SourceMonitor／Coverity）；③ 能说明单元测试的四条目标与"驱动一桩"的作用；④ 能说出 JUnit 5 = Platform + Jupiter + Vintage 三部分，并解释 @Test／@BeforeEach／Assert／Runner 各自的角色；⑤ 能区分 Dummy／Stub／Spy／Fake／Mock 五种测试替身，并说明 Mock 与 Stub 的核心差别（状态验证 vs 行为验证）；⑥ 能说出 JaCoCo 的六个覆盖率维度与 on-the-fly 插桩原理；⑦ 能说明自顶向下／自底向上／混合三种集成策略的优缺点，以及契约测试（CDC）的两端（Consumer／Provider）；⑧ 能画出 CI/CD 流水线的主要环节与质量门禁
能力	① 能对一个类写出符合 AAA（准备—执行—断言）结构的 JUnit 5 测试；② 能用 Mockito 的 mock／when／verify 写出一条"行为验证"用例；③ 能读 JaCoCo 报告，指出"覆盖率高但分支没测到"的具体位置；④ 能为一个微服务接口写一份最小契约（请求/响应格式），并说明消费者端与提供者端各测什么；⑤ 能在 CI 流水线里定位"测试失败该看哪一段日志"
素质（思政）	① 由代码评审"互为评审、集体评审"体会团队协作与开放心态；② 由"每 1000 行残留 2–6 个 Bug、找 bug 占总投入 30%–60%"体会质量成本意识与精益求精；③ 由第三方开源库漏洞数据（98% 的公司不了解自己在用什么）体会**供应链安全意识与合规责任**；④ 由"单元测试是系统质量的基石"体会脚踏实地、把基础做扎实的工匠精神；⑤ 由持续集成"门禁不通过就发不出去"体会规则意识与对交付负责

本讲要建立的一个总印象：第 4 章讲的是"流程怎么排"，这一章开始讲"**代码这一层到底怎么测**"——先用**评审和静态分析**在不运行代码的情况下抓缺陷，再用**单元测试**把每个函数钉死，最后靠**集成测试与持续集成**把零件拼成能跑的系统。

二、时间分配总表

> ⚠ **容量提示（务必先读）**：本章课件 **100 页**，本讲稿是**素材全量版**（口播约 5 万字，≈200 分钟纯朗读），课时只有 **90 分钟**。请按下表取用：
>
> 1. **重点页全文慢讲（2–3 分钟/页）**：
> 5.6、5.9、5.16、5.34、5.36、5.37、5.39、5.46、5.48、5.49、5.53、5.60、5.64、5.69、5.77、5.81、5.82、5.90、5.92、5.93、5.95。
>
> 2. **常规页快速带过（0.8–1.5 分钟/页）**：只讲口播第一段（结论）＋1 个提问。
>
> 3. **可课后阅读页（★）**：★5.10–5.15、★5.17–5.28、★5.30–5.31、★5.40、★5.42–5.45、★5.50–5.52、★5.55–5.59、★5.61–5.63、★5.65–5.68、★5.70–5.74、★5.78–5.80、★5.83–5.89、★5.90–5.93、★5.94、★5.96–5.97。
>
> 4. **必须当堂动手页**：5.37（那个 `add` 方法的缺陷，现场找）、5.49（断言写一条）、5.60（Stub/Mock 取舍判断）、5.77（画集成顺序）。
>
> 5. **实验 2（5.96）**务必用 3 分钟交代清楚：用 JUnit 测 `ItemController`，提交测试代码；**5.97 学习资源**可略。

90 分钟课堂推进表

时段	页码	内容	分钟	备注
第 1 节	5.1–5.4	开场、本篇地图、本章三部分	5	讲清"第 2 篇开始讲技术"
第 1 节	5.5–5.14	5.1 代码评审：价值、关注点、代码规范、互查实践	12	**5.6、5.9 必讲**（互查三条硬指标）
第 1 节	5.15–5.31	静态分析工具链与 SAST/SCA、供应链风险	10	**5.16、5.28 必讲**；工具页可略

——	——	**课间休息 10 分钟**	——	——
第 2 节	5.32–5.40	5.2 单元测试：价值、目标、驱动与桩	10	**5.34、5.36、5.37、5.39 必讲**
第 2 节	5.41–5.51	JUnit 5：架构、Annotation、Assert、Runner	9	**5.46、5.48、5.49 必讲**
第 2 节	5.52–5.66	Mock 家族与 Mockito	10	**5.53、5.60、5.64 必讲**
第 2 节	5.67–5.74	覆盖率：JaCoCo 与插桩原理	6	**5.69 必讲**；插桩源码页可略
第 2 节	5.75–5.88	5.3 集成测试：单体策略、微服务、契约测试	10	**5.77、5.81、5.82 必讲**
第 2 节	5.89–5.95	CI/CD 与持续测试、本章小结	6	**5.92、5.93 必讲**
第 2 节	5.96–5.98	实验 2 交代、学习资源、结束	3	实验要求说清

> **压缩优先级**：先压缩 ★ 标记的工具截图页与框架源码页 → 再压缩 JUnit 内部类（TestCase／TestSuite／TestResult）与 EasyMock／JMock 细节 → 最后才压缩
>
> 5.9、5.34、5.37、5.46、5.53、5.60、5.69、5.77、5.81、5.82、5.92。**这十一页是考点密集页，不要动。**

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（5.1–5.98），可逐页对照：http://111.231.0.226:2345/lecture5/index.html
>
> **口径提醒**：教材第 5 章标题为"单元测试与集成测试"，课件 5.3 页把教材的 5.7／5.8 两节（系统集成的模式与方法、持续集成及其测试）合并讲成"5.3 持续集成测试"——**按课件顺序讲，按教材口径记**。课件里出现的 `SolarLint`／`SolarQube` 是 `SonarLint`／`SonarQube` 的笔误，以正确拼写为准。

【5.1】第 2 篇 软件测试技术（1 分钟）

> 前四章讲完，问题会换。前四章解决的是要不要测、缺陷怎么分类、用例怎么设计、流程怎么排。那些都还要用。可你一旦打开真实项目，第一个卡住你的会变成：手上这段代码，究竟怎么测？

>

> 第 2 篇用五章回答它。第 5 章，单元测试与集成测试。第 6 章，系统功能测试。第 7 章，专项测试。第 8 章，国际化和本地化测试。第 9 章，软件测试自动化及其框架。五章按这个顺序走。先把最小的那一块钉住，再把它们拼成系统。然后做性能、安全这类专项。再处理跨语言、跨地区的交付。再把重复劳动交给自动化。顺序不能倒。倒了，专项测试会站在没钉死的代码上。

>

> 今天只走第一段。第一段松了，后面的系统测试会站在松的东西上。自动化只会让这个松的东西更快地重复出错。

>

> 单元测试看的是最小那一块还对不对。集成测试看的是拼上之后它们还认不认对方。页面好不好看、接口好不好用，是后面几章的事。

>

> 这条线请进笔记：从零件，到整机，到特种检验，到跨地区交付，到无人车间。五章各管哪一段搞混了，做实验时就会把单元测试和系统测试搅成一件事。今天的出口更具体。拿到一段陌生代码，写出第一批能跑的测试，并且说得清为什么是这几组数据。

提问：这五章里，哪一章的成果最显眼，哪一章最不显眼、但后面都靠它？

预设回答：

- 答"第 9 章自动化最显眼"——绿灯确实醒目。它跑的是前几章设计出来的检查，不会自己发明该测什么。

- 答"第 5 章最不显眼"——对。单元测试写完很少有人鼓掌，可集成和系统测试都站在它上面。

- 答"专项测试最贵"——性能和安全出事代价大，但它依赖前面留下的可测性。

- 收口：显眼的是结果，不显眼的是底下那一层。这一章就讲这个。

易错点 / 考点：五章顺序常被考成"集成测试在哪一章、自动化框架在哪一章"。口诀：零件 → 整机 → 特种检

验 → 跨地区交付 → 无人车间。第 8 章是国际化和本地化，不是"只改翻译"。

过渡：这一章名字里有两个词，中间那个"与"最容易出事。我们先把名字拆开。

【5.2】软件测试方法和技术（1 分钟）

课设里那种场面你们见过。计算模块绿灯，接口绿灯，演示前一夜一对字段，空指针。两边都说"我这边测过了"。测过的是各自那一块，没测的是拼上那一下。

>

章名里两个词，中间那个"与"才是事故高发区。"单元"是把一个方法或一个类单独拎出来钉死。"集成"是把这些零件按设计装到一起，看它们还认不认对方。最常见的事故就发生在这个地方：每个函数单独看都对，一拼就错。或者单元层全是洞，集成时一起塌。这一章就是把这两段分开钉、再合起来验。

>

你们很多人会进银行科技或金融科技团队。2024 年 1 月 15 日，人民银行发布并实施 JR/T 0290—2024。标准名是《金融业开源软件应用管理指南》。它要求对开源软件做全流程管理，而不是靠人记得用过什么库。2026 年 9 月 19 日，银发〔2026〕180 号又发了三项推荐性标准。JR/T 0362 是测试术语，0363 是测试数据设计，0364 是验收测试质量控制框架。它们不代替你写 JUnit。但说明一件事：金融系统里，"测过"必须说得清。这是行业标准文本，不是课堂编的案例。

>

所以毕业标准请写在笔记第一行。拿到一段陌生代码，写出第一批能跑起来的测试。并且说清为什么测这几组数据。会把框架跑起来只是底线。说不清理由，就还没学完。

提问：如果本章只留一个毕业标准，你选"会把 JUnit 跑起来"，还是"能说清为什么这么测"？

预设回答：

- 答"跑起来"——这是底线。工具一周就能熟悉，会跑不等于会测。

- 答"说清理由"——这是我想听到的。能讲清"为什么是这组、为什么漏了那组"的人少得多。

- 追问：说清理由要靠什么？——用例设计、覆盖的含义、对缺陷类型的敏感。这三样本章都会给。

易错点 / 考点："单元"不是"一个文件"，是可独立测试的最小单位，通常是一个方法或一个类。"集成"不

是"部署上线"，是按设计把单元装配起来并测试装配结果。这两个词每年都有人答反。

过渡：名字清楚了，先看教材这一章原来分了多少节，我们再决定课上怎么走。

【5.3】教材内容（2 分钟）

假设课设做到第三周。甲写了计算模块，乙写了接口，丙负责把两边接上。演示前一天，单独跑都绿，一联调就空指针。甲说"我的方法测过了"，乙说"文档上就是这个字段"，丙说"我又不是开发"。这个场面，就是教材第 5 章八节要拆开的问题。

>

八节看着多，课上按三块走。第一块：代码静态测试、代码评审案例分析、代码静态检测工具。这三节的共同点是不运行程序。人读，工具按规则扫。环境、数据和依赖还没齐，能看出来的毛病先挑掉。

>

第二块：单元测试的目标和任务、分层单元测试、单元测试工具。这三节要运行，而且只运行最小单位。刚才那个空指针，如果甲的方法有断言、有传入 null 的用例，演示前一天就该红。不该等到联调才红。

>

第三块：系统集成的模式与方法、持续集成及其测试。这两节关心的是拼，而且是每次改动都再拼一次。丙的痛苦多半是拼完没有一条自动的回归。

>

三块对应三个字：看、跑、拼。教材按八节写，我们按三块讲。读的时候请看节标题里的判断。"分层单元测试"已经告诉你，单元内部也有层次。"持续集成及其测试"里的"及其"，是说集成落到测试上有自己的做法。还有一个用词：静态测试里的"静态"，指的是被测程序没有运行，不是"测试手段很死板"。

提问：教材把人工的代码评审放在静态检测工具前面。若只许留一个理由，你留哪个？

预设回答：

- 答"因为评审出现得更早"——时间上对，但顺序的理由是逻辑：人先知道要看哪些维度，工具才知道该把哪些维度自动化。

- 答"工具是评审里可规则化的那一部分"——这句是本章暗线。工具几乎都查得到的东西，都能追溯到人的关注点。

- 追问：工具能替代评审吗？——不能。工具认规则，评审还要问"这段代码为什么存在"。

> **易错点 / 考点：**八节名称常考填空或匹配。最容易错的是把"5.5 分层单元测试"答成集成测试。它仍是单元内部的分层，例如先测底层算法、再测上层服务类，不是模块之间的装配。

> **过渡：**八节我们压成三部分来讲。压法本身也是考点，下面把路线对齐。

【5.4】本章结构：三部分五环节（2.5 分钟）

> 教材写了八节，看着很多。课上按三部分走就行。课件把八节收成三部分，后面再留小结和练习。第 4 版相对前一版，这一块的骨架变化不大。所以笔记可以按这三部分记。答题时再看题目问的是"教材八节"还是"课件三部分"，切换口径。

>

> 第一部分，5.1 代码评审与分析，对应教材前三节。学完要能做两件事。评审时盯得住规范与风格、圈复杂度、信息流复杂度、耦合。也知道哪些检查可以交给工具。

>

> 第二部分，5.2 单元测试，对应教材第四到第六节。这是今天的重心，也是实验要交代码的部分。目标、驱动程序和桩程序、JUnit 5、什么时候用 Mock、覆盖率报告怎么读，都在这里。

>

> 第三部分，5.3 持续集成测试，对应教材最后两节。集成从"发布前拼一次"变成"每次提交都拼"。代码一提交，就编译、就跑测试、就过门禁。

>

> 后面两块是收尾。小结把重点难点归位。练习是基于 JUnit 的单元测试实验，交的是测试代码，不是一篇感想。三部分各对应一个问句：不跑代码能查出什么？跑起来怎么测一个单元？拼起来以后靠什么保证它一直是对的？行业标准定的是术语、测试数据和验收质量的口径，不代替这三问。你的课程设计仍然要交：能跑的测试，加上你为什么选这组数据。

> **提问：**三部分里，你以前接触过哪一块？你印象中的"单元测试"是谁的活？

> **预设回答：**

- 答"没写过，以为是测试人员的事"——很典型。单元测试的第一责任人是写这段代码的人。测试人员更多是定标准、看覆盖、补关键用例。

- 答"写过，就是 assert 一下"——入了门。assert 后面还有数据怎么选、替身怎么用、失败了说明什么。本章把这些补上。

- 答"持续集成就是装一个 Jenkins"——工具是外壳。内核是：每次变更过同一套自动检查，不过就不能合。

- 收口：第二块要动手，第一块要养成习惯，第三块要看懂它在团队里怎么落地。

> **易错点 / 考点：**"三部分"是课件的教学整合，教材仍是八节。题目问教材结构，按八节答；问本章怎么讲，按三部分答。两套口径不要混在同一道题里。

> **过渡：**从第一部分开始。先建立一个前提：代码可以一行都不跑，就已经在做测试。

【5.5】5.1 代码评审与分析（1 分钟）

> 程序一行都不用跑，测试就已经开始了。不跑，意味着你还没有齐环境、没有造数据、没有把依赖拉起来。这也是你能最早插手的时刻。代码刚提交，就可以看。

>

> 三个词请分开记。评审是人看，同行读你的变更。分析是工具看，扫描器按规则挑可疑模式。规范是大家事先约定什么叫写得清楚，否则评审会变成口味之争。

>

> 两者的关系可以点成一句：评审解决"人能不能看懂、会不会把意图想错"；分析解决"已经写成规则的坑，能不能自动兜住"。不是谁替代谁。人会累、会放过第 400 行以后的代码。工具不会问"这个需求该不该存在"。

>

> 很多同学心里其实不信这件事：不跑怎么知道错？缺陷分两类。一类要跑起来才暴露，比如并发、性能。另一类读一遍就知道，比如该判空的没判、资源没关、日志里打了口令。后一类占比不小，而且越晚发现越贵。

>

> 还有一句界限：评审只能说明"我没看出问题"，不能证明"程序是对的"。证明对不对，还要靠后面的运行测试。课程设计里，请不要用"我们开过评审会"代替测试代码。会可以开，结论要落到修改项，修改项要有人回头验证。提交之前，至少有第二双眼睛，或者有一套规则扫过。

> **提问：**不运行程序就找缺陷，你觉得靠谱吗？它靠谱的边界在哪里？

> **预设回答：**

- 答"不靠谱，不跑怎么知道错"——这是最常见的直觉。把它改成：有一类错误读一遍就知道，另一类必须跑。

- 答"靠谱，很多错看就能看出"——对，而且越早看越便宜。

- 追问：评审能替代测试吗？——不能。没看出问题，不等于没有问题。

> **易错点 / 考点：**评审属于静态测试，是测试活动，不是"开发组内部的私事"。判断题"代码评审不属于软件测试"——错。另一题："静态分析能发现运行时的性能问题"——也错，那要动态测试。

> **过渡：**既然它算测试，它到底图什么？下面六条价值，层次并不一样。

【5.6】代码评审的价值（2.5 分钟）

> 先说个你们组里肯定出现过的情况。核心那个类，只有一个人能讲清楚。他一请假，别人不敢改。不是懒，是看不懂，怕改坏。

>

> 评审干的就是这件事：逼着第二个人把代码读进去。他读的时候一般会问你两个问题。这块为什么判空？这个字段传空进来会怎么样？你答不上来的地方，就是 bug 待的地方。所以评审最值钱的地方，是让第二个人也懂这段代码。人走了，项目别瘫。

>

> 课件列了六条价值，其实都从这一件事伸出来。改善质量最直接，别人看得见你没看见的输入。带新人最实在，你在具体某一行上问他一句，比听十节课管用。规范写在文档里没人看，写成评审意见他就得改。围绕代码的评论，比周会便宜，这叫增强协作。减少项目代价也跟上来。缺陷在这一层拿掉，就不用留到线上再付。

>

> 最容易被省掉的是后面几条。改善质量当场见效，知识共享要到半年后才显形，所以它总是第一个被砍。跟领导汇报，你讲省了多少钱。跟组里推动，你讲"人走了项目别瘫"。同样一件事，两种说法。这不是滑头，是让对的人听得进去。

>

> 现在还有个新变化。2026 年 11 月 CodeRabbit 转述了一份它赞助的 LeadDev 调查。约一半受访者说，用上 AI 编码后每人提交的变更量至少多了四分之一。45% 的组织把代码评审列成交付中最大的瓶颈。82% 的团队仍然在合并前人工看代码。只有 14% 的组织有一套说得清、也真在执行的评审标准。这是厂商赞助的调查，不是独立实验。但方向挺明确：代码变多以后，评审不会消失，没有标准的评审会先崩。

>

> 课程设计里请你做到一件具体的事。让你的核心类被另一个人读过。并且他能在答辩时讲清你那段逻辑为什么这么写。

提问：六条里，哪一条老板最不当回事，可半年后最要命？

预设回答：

- 答"知识共享"——对。它不在这次迭代的燃尽图上，所以第一个被砍。
- 答"辅导新人"——重要，但评审不是培训的替代品，它是在真实代码上做辅导。
- 答"减少代价"——最容易被接受，也算得清，适合对外讲。
- 收口：跟领导汇报就讲省了多少钱，跟组里讲就讲人走了别瘫。

易错点 / 考点：多选题问"代码评审的价值包括哪些"，六条尽量记全。只答"发现缺陷"会丢分——教材同时强调知识传递和规范落地。

【思政落点 1】：评审把"我的代码"变成"我们的代码"。被人指出问题，第一反应常常是解释。工程事实是，没有人能独自保证正确。收到意见时先改、再辩，不是没骨气，是把损失留在没有用户受害的时候。开放心态就落在这一次选择上。

过渡：价值清楚了，接下来得知道眼睛往哪儿看。

【5.7】关注代码评审（3 分钟）

> 评审如果只甩一句"感觉不好"，作者没法改。眼睛得落到能指认的点上。课件给四个关注点，问的是同一件事：这句现在对不对，以及这个结构以后还活不活。

>

> 第一个就接住那句问：是否符合代码规范和代码风格？要看。命名有没有意义、方法是不是长篇大论、格式是不是统一。风格不直接决定对错，它决定别人能不能读。读不懂的代码，后面的测试和维护都会变贵。这一条也最好自动化，所以后面会接到规范文档和 Checkstyle。

>

> 第二个，圈复杂度，公式是 e − n + 2p。e 是控制流图的边，n 是节点，p 是连通分量。一个方法通常 p 取 1，于是常写成 e − n + 2。这个数的实在含义是：独立路径有几条，也就大约是"每条路都走到"所需用例数的下限。手算一个例子。9 个节点，10 条边，p 为 1。10 − 9 + 2 = 3。至少准备三条路径用例。经验上，一个方法超过 10 就该考虑拆开。下限不是证明：圈复杂度等于 1，只说明判定分支少，除零、空指针、溢出照样会发生。循环、switch、短路的与或，都会把数字抬上去。

>

> 第三个，信息流复杂度，课件的式子是（扇入 × 扇出）的平方。扇入是有多少上层调用我，扇出是我依赖多少下层。平方是在说两头互相放大：一个被很多地方调用的方法，自己再抓住五六个服务，改一处就横跨一片。接口越宽，改动越贵。

>

> 第四个，模块耦合性，看功能和数据上相关联的模块占多大比例。关联越多，越难单独测、单独改。四个点收到这儿：评审不只判"这句对不对"，还判"这个结构以后的修改成本"。课程设计评审时，请至少报出核心方法的圈复杂度，并说明你准备了几条路径。报不出来，就还没看结构。

提问：一个方法圈复杂度等于 1，能不能说它不用测？

预设回答：

- 答"只有一条路径，不用测"——这是最危险的误解。圈复杂度只数判定，不管数据。

- 答"结构简单，但仍要测边界和非法数据"——对。低复杂度只说明路径少。

- 答"等于 1 就是没有 if"——大致对，但循环和短路表达式也算分支。

- 收口：它是用例数量的下限估计，不是"没有缺陷"的证明。

> **易错点 / 考点：**公式必须记准 e − n + 2p。最常漏的是 `2p`，或把 p 当成 0。单个方法 p 通常为 1。判断题"圈复杂度高则更难测"——对；"圈复杂度低就没有缺陷"——错。

> **过渡：**四个关注点里，风格最容易先落地，因为已经有公开规范可以引用，不必从零吵。

【5.8】首推代码规范（2 分钟）

> 团队规范最容易走的弯路，是从一张白纸开始发明。正确的决策是：先引用一份够细、公开、能进工具的业界规范。

>

> 课件首推 Google 的两份。C++ 看 cppguide.html，Java 看 javaguide.html。都在 google.github.io/styleguide。选它们，有三个能执行的理由。

>

> 够细。命名、函数长度、注释放哪、异常怎么用，C++ 里哪些特性禁用，都写了。"代码要优雅"无法执行，"一个函数不要承担两件不相干的事"可以争论，也可以检查。细，才不会每次评审都重新开会。

>

> 公开，而且大家认这个出处。团队可以说"我们以 Google Java 规范为底"，就不必花两个月争论驼峰还是下划线。权威在这里的意思不是"不许有例外"，而是"例外要写出来，不能靠口头习惯"。

>

> 能变成工具配置。规范一旦稳定，就能进 Checkstyle、进 IDE 的格式化。人去看设计和意图，机器去看已经写成规则的风格。这个接口，后面讲 Checkstyle 时会接上。

>

> 落地就按这个写：团队规范 = 一份业界规范 + 少量项目约定。业界规范解决大部分争议。项目约定只写项目里特殊的那几条，比如金额用哪个类型。已经在用阿里 Java 规范并且全组一致的，不必为了"更权威"改投 Google。最容易出事的，是一半人按 A、一半人按 B，评审变成风格吵架。课程设计从第一次提交就选定一

> 份，写进仓库里的说明，不要等联调前再统一。

> **提问：**组里已经统一用另一套 Java 规范了，要不要换成课件首推的 Google 这份？

> **预设回答：**

- 答"要换，课件首推"——没必要。规范的价值在统一，不在品牌。

- 答"不用换"——方向对，再检查一件事：有没有自动检查。没有工具的规范，落地率很低。

- 追问：一半人按 A、一半人按 B 呢？——先统一，再谈哪份更好。评审时间不该花在括号上。

> **易错点 / 考点：**代码规范的作用要答到可读性、可维护性、团队一致、可自动化检查，不能只答"好看"。不要把风格规范和安全编码标准混成一件事：前者管结构和写法，后者管哪些写法会变成漏洞。

> **过渡：**规范有了，评审怎么组织才不像"帮我瞄一眼"？下面这几个数字，课设里就按这个来。

【5.9】代码互查的优秀实践（3 分钟）

> 人不是扫描仪。一次看太多、看得太快、看得太久，后面的行都会变成"看着挺好"。课件列的是代码互查的优秀实践。出处是 Best Practices for Peer Code Review。三条硬指标，外加三条闭环做法。

>

> 三条硬指标方向一致，都是"少一点、慢一点、短一点"。一次检查少于 200 到 400 行代码。你把一千行差异扔给同学，对方只能浏览。检查速度努力落在每小时 300 到 500 行。四百行大约要一小时。评审是在推演：这个变量此刻可能是什么、这个分支进得去吗、异常谁接。单次不宜超过 60 到 90 分钟。超过这个长度，发现率会掉下来。正确做法是拆成几次，别想着今天下午全看完。这三条数字请记住。

>

> 2025 到 2026 年，Graphite 的公开产品资料用堆叠拉取请求把大改动拆小。好让每次评审仍落在人读得完的尺寸里。2025 年 12 月有公开报道称 Cursor 宣布收购它。你不必去装。课程设计的动作只有一个：一次合并请求不超过这条上限，超过就拆。拆不开，往往是设计耦在一起了。

>

> 另外三条是闭环。复审前，作者先把关键处的注释写上，说明为什么这样处理，评审者不必先考古你的思路。使用检查表。课件的判断很硬：检查表肯定能改进作者和评审者双方的结果。空指针判了没有、资源关了没

> 有、日志里有没有敏感信息，靠记忆一定会漏。验证缺陷是否真正被修复。意见提了、作者说改了、没有人回头看，等于没改。

>

> 六条收到这儿：少、慢、短、先注释、用清单、必回验。前五条决定能不能发现问题，回验决定问题有没有被解决。小组如果只先落实一条，先落实回验：每条意见有"已改"或"不改的理由"。

提问：小组只能先落实一条，你选哪条？说一个会因此少掉的具体事故。

预设回答：

- 答"用检查表"——成本最低，而且能留给下一组人。

- 答"一次少看一点"——最容易执行：大改动拆开再请人看。

- 答"必回验"——前五条负责发现，这一条负责关闭。没有关闭，评审是聊天。

- 收口：最容易崩的是速度。大家习惯把评审当浏览，所以"慢"和"短"要刻意排进时间表。

易错点 / 考点：三个数字必须记准：200–400 行/次，300–500 行/小时，60–90 分钟/次。常见错法是把"每小时"记成"每天"，或把"单次时长"记成"每天时长"。口诀方向一致：量小、速慢、时短。

【思政落点 2】：认真是有成本的，所以要给它排时间。"帮我瞄一眼"之所以普遍，常常是因为计划里根本没有这一小时。给评审留一小时，比事后通宵追一个已经合进去的错误便宜。检查表也是同一态度：专业的人也会漏，所以把靠记性的事变成靠清单的事。

过渡：这些动作在今天大多不发生在会议室里，而发生在一次合并请求上。

【5.10】工具支持：Collaborating with pull requests @GitHub（2.5 分钟）

> 今天的评审，多半不发生在会议室。课件给的入口是 GitHub 那份文档。标题是 Collaborating with pull requests。地址在 docs.github.com 的 pull-requests。别盯菜单。合并请求把四样东西放在一起。这次改了什么、评论落在哪一行、讨论的来龙去脉、同意或要求修改的结论。

>

> 评审对象是变更，不是整个仓库。你不必重读三万行，只读这一次为什么改。上一页"200 到 400 行"能落地，就是因为变更被切小了。一次合并请求可以包含多个提交。提交是保存点，合并请求才是待评审的那一包。

>

> 留痕。半年后有人问这行为什么这样写，评论还在。文档会过期，当时讨论的记录更接近决策。

>

> 自动检查可以挂在同一次请求上。静态分析、单元测试的结果先出来，人把时间留给机器判不了的设计和意图。这就是后面持续集成要接上的钩子。

>

> 2026 年 GitHub 官方更新日志把 Copilot 代码评审推了三步。10 月 2 日可经接口发起，默认力度改为 Balanced。10 月 28 日，Max 进入公开预览，模型的批准可计入仓库要求的批准数。11 月 17 日，按风险选力度的 Adaptive 和密钥检测进入公开预览。公开技术解读还提到，依赖清单可能不在评审范围内。"加了一个库"时模型可能不说话。那一刀留给后面的成分分析。

>

> 课程设计哪怕只有三个人，也走合并请求。模型可以提意见，不能算你们的评审人。同意合并的那一下，必须是一个能对这段代码负责的人。平台换成 GitLab 或 Gitee 也成立，四要素一样。连版本管理都没有，先解决"说得清改了哪些行"。否则评审无从开始。

> **提问：**团队用的是 GitLab 或学校的仓库，不是 GitHub。这套做法还成立吗？

> **预设回答：**

- 答"不成立，课件写的是 GitHub"——名字不重要。要抓住的是四要素。

- 答"成立，换个名字"——对。

- 追问：如果只有共享文件夹呢？——先做版本管理。没有差异，就没有评审对象。

> **易错点 / 考点：**工具支撑要答到合并请求、行级评论、与自动检查集成。不要把提交和合并请求混为一谈。另一个新错法：把模型的"批准"写成测试通过。批准表示某一方认为可以合，不是用例全绿。

> **过渡：**流程还是抽象的。下面把一次变更评审拆成你要读出的四件事。

【5.11】示例：change review（2 分钟）

> 你打开一个合并请求，第一眼看什么？别急着评好不好。一次变更摊开，你要读出四件事。缺一个，评审就退化成聊天。

>

> 第一，范围。这次动了哪些文件，是新增还是修改，和你要看的问题有没有关系。范围之外的代码，这次不必重读。第一分钟就该划出边界，否则你会在无关文件上花掉那 60 分钟。

>

> 第二，上下文。只看改动的那一行不够。同一句赋值，放在事务里和放在事务外，风险完全不同。好的评审会让你看见前后在做什么，你要主动问：调用方是谁，失败了谁来接。

>

> 第三，评论的落点。意见要挂在具体的行上，并写成可执行的改法。"整体还行，建议优化一下"不是评审意见。作者无法改，于是只能忽略。把模糊的不满说成"循环里的查询改成先收集再查一次"，才叫意见。

>

> 第四，结论。看完要有状态：同意合并、要求修改、或先讨论。沉默就是漏审。2026 年 11 月 17 日，GitHub 官方更新日志增加了仓库 Insights。用来看 Copilot 评审评论有没有被处理、高严重度评论处理了多少。启发不在于去开那个页面，而在于：评论要有下落。课程设计的每次请求，请留下"已改"或"不改，因为……"。没有下落的评论，下周不会有人记得。

> **提问：**作者只收到一句"整体还行，建议优化一下"。他下一步怎么做才对？评审者错在哪？

> **预设回答：**

- 答"先不改"——模糊意见确实无法执行，但作者就此停下，问题还在。

- 答"问清楚要优化哪一行、改成什么样"——对。同时也说明评审者没把意见写成可执行的话。

- 追问：怎样才算一条合格意见？——定位到行、给出改法、说出理由。三条缺一，就还是态度，不是评审。

> **易错点 / 考点：**评审意见的质量标准是可定位、可执行、有理由。评审的产出是修改项，不是一个分数，也不是一句鼓励。

> **过渡：**在这些意见里，最值得反复练的一类缺陷，是空指针保护。它常常不运行也看得见。

【5.12】缺陷示例：空指针保护案例分析（2 分钟）

> 空指针有个讨厌的脾气：不跑也能看见，跑起来却常常不立刻崩。只是结果不对，或者被悄悄吞掉。我们不编一个公司事故，就看四种在课堂和课设里反复出现的形态。

>

> 第一种，该判的没判。方法拿到对象就调用，注释里声称"调用方保证非空"。调用方有三个，其中两个并不保证。契约写在注释里，没有写进代码，也没有写进测试。

>

> 第二种，判错了对象。判了 A 不为空，用的却是 B；或者外层对象判过了，里面的字段仍可能是空。代码里"看起来有判空"，评审最容易在这里放行。

>

> 第三种，只保护了一半。入口判过了，后面某个分支又取了一个可能为空的值，直接用。保护停在你记得的那条路上。

>

> 第四种，用异常把问题吞掉。捕获空指针，然后返回空，或者只打一行日志。程序从此"不报错"，错误被推迟到更远、更难查的地方。这种写法在评审里应是必须修改项。

>

> 四种形态问的是同一个问题：这个值由谁保证非空？不是"这里有没有一个 if"。2026 年 11 月 17 日，GitHub 官方更新日志做了密钥检测。它放进 Copilot 代码评审的公开预览。规则和模型一起指出暴露的口令，希望在合并前删掉。它和空指针是同一类：不跑程序也看得见。但公开预览不是你的检查表。预览会变，模型会漏。课程设计的检查表请加两条：谁保证这个参数非空；这次差异里有没有口令、密钥、连接串。两条都要人确认，不能只等工具变绿。

> **提问：**方法入口对所有参数判空，为空就抛 IllegalArgumentException。这算不算做好了空指针保护？

> **预设回答：**

- 答"算，很规范"——形式完整，还要问调用方是谁。对外接口上快速失败通常是对的；如果空是合法业务输入，

直接抛就是错的。

- 答"要看业务上能不能为空"——更接近要问的那件事。先定契约，再让代码和测试跟契约一致。

- 追问：正确的顺序是什么？——可空还是不可空、为空时返回、抛出还是走默认值，先写清，再写 if。

> **易错点 / 考点：**判断题"加了判空就没有空指针风险"——错。`catch (NullPointerException)` 然后空处理，坏处是隐藏缺陷，把确定的错误变成更远处的随机错误。

> **过渡：**互查和检查表之外，还有一种很笨、但很管用的做法：让作者当着大家把代码讲出来。

【5.13】开发人员走上台介绍其代码（2 分钟）

> 写的时候你可以糊弄自己，讲的时候糊弄不过去。课件给的做法很朴素：开发人员走上台，介绍自己写的代码。规则只有三条。每个人都有机会，这不是高手表演，新人也要讲。随机抽取，不提前指定。提前指定，人会准备一份演示版。随机抽，你平时写的每一行都可能被摊开。现场展示，打开代码从入口走下来：为什么这样拆、这个分支什么时候进、这里的缓存为什么存在。不是念稿。

>

> 它为什么管用？讲到某个语句卡住，说不出它为什么在这儿，这个语句多半该重构，或者根本没人测过。这就叫用讲解来验证理解。

>

> 还有一层和现在的写法有关。前面那份 2026 年的赞助调查里有一个刺耳的数字。大约七成工程师承认，自己批准过并没有完全看懂的 AI 生成代码。调查由代码评审厂商赞助，数字不要背成定律，但现象值得防。模型可以在你讲不清的地方写得很流畅。上台讲，就是不许用"它能跑"代替"我知道它在做什么"。

>

> 它还顺手降低一个风险：只有一个人懂某块代码。讲一次，第二个人至少有了地图，出问题不至于只有作者能救。课程设计时间短，也请做一次：每人五分钟，不讲界面，讲一个核心类。讲到卡住的那一行，记下来，要么补测试，要么改设计。被挑出问题，是这次活动最便宜的部分，因为它发生在没有用户受损的时候。

> **提问：**如果你被随机抽到，讲上周写的一个类，最可能在哪一句上卡住？

> **预设回答：**

- 答"讲不清为什么这样设计"——很多决定其实是顺手写的。讲不出理由，就还没有决策。

- 答"讲不清某个分支何时进入"——要么是死代码，要么是没测过的路径。两种都要处理。

- 答"能讲清，但怕被挑错"——怕是正常的。被挑错的成本，远低于合进主干之后再改。

> **易错点 / 考点：**评审形式包括正式会议、同行互查、走查、上台讲解。判断题"代码评审只能是书面检查"——错。"随机抽取"的作用是避免表演、让日常代码保持可讲。

> **过渡：**怎么评，讲过了。还要让它变成习惯，而不是靠某一个人记得提醒。

【5.14】其它优秀实践（2.5 分钟）

> 组里换了一茬人，新来的还在看别人的代码。这个东西就叫习惯，不叫某个人人好。课件给的五条，就是把这件事做成习惯的顺序。

>

> 关键代码做集体评审。不是所有代码都配得上全员坐下来。挑涉及金额、权限、核心计算、并发和事务的部分。这类代码出错的代价最大，评审的投入才划算。平均用力，结果是关键处反而没人看。

>

> 把评审当成文化。判断标准很具体：新人提交后，有没有人看？指出问题时，是"这儿可能有问题"，还是"你怎么连这个都不懂"？前者让人下次还愿意打开评论，后者让人学会把请求拆得谁也不想看。

>

> 借助工具。合并请求、静态检查、自动测试的结果，把人从能规则化的事情里解放出来。工具的价值是省下注意力，不是代替判断。

>

> 把流程写下来。谁必须看、多久之内响应、什么级别的改动要几个人同意、不通过怎么处理。没写下来的流程，全靠谁好意思催。这里要补一条 2026 年已经出现的规则。GitHub 允许把 Copilot 的批准计入必需批准数。那是平台能力，见官方更新日志，不是你们的默认设置。课程设计请写明：模型评论必须由人关闭；涉及金额和权限的改动，不允许只靠模型点同意。

>

> 培训。检查表告诉评审者看什么，培训告诉他为什么看。没人讲过空指针的四种形态，清单上的那一行会被勾成"已看"，其实没看懂。五条是一条链：挑重点，立氛围，上工具，定规矩，提能力。人少的组，先立最小规矩：关键类必须有第二个人看过，并留下结论。先有，再谈好。

> **提问：**五条里小组只能先补一条，你补哪条？补上之后，下周哪件具体的事会不一样？

> **预设回答：**

- 答"先写流程"——务实。谁看、多久回、几个人同意，写下来才有约束。

- 答"先做关键代码集体评审"——人少时投入产出比最高。

- 答"先讲文化"——最根本，也最慢。没有一条最小规矩，文化无从长出来。

- 收口：先规定"关键类必须第二个人看过"。跑顺了，再补培训和工具。

> **易错点 / 考点：**五项是集体评审、评审文化、工具支撑、流程定义、培训。易错点是把集体评审理解成"所有代码都要开会评"。前提恰恰是挑选关键代码。

> **过渡：**人看得慢，也会漏。接下来把能写成规则的那一部分交给工具。

【5.15】代码静态分析工具（1 分钟）

> 人看得慢，也会漏。从这里起，角色换成工具读代码。名字拆开就是三件事。静态：不运行被测程序。分析：按规则理解代码的结构。工具：这件事可以自动、重复地做。合起来，它把源码或字节码当成数据，查找可疑模式。

>

> 它和评审的分工请固定成一句。评审负责"为什么这样写"。工具负责"这样写有没有踩已经明文化的规则"。人判断意图，机器不知疲倦地重复。两边不要互相替代，也不要互相甩锅。

>

> 这一块看着全是工具名。考点不在你会不会点安装，而在哪一类工具查哪一类问题。装错类，风格工具去查空指针，查了个空。

>

> 还有一句先听进去：这些工具会误报，也会漏报。报告里有一堆问题，不要慌，那是线索。报告里一个问题都没有，更不要放心，那只说明没有命中它携带的规则。规则永远只覆盖代码可能性的一小部分。

>

> 课程设计里请这样用：提交前跑一回轻量检查，把这次改动新引入的、你能解释的问题处理掉。不要把"工具零告警"写进实验报告，当成程序正确的证据。零告警只是"这套规则没说话"。对不对，仍要靠你写进断言里的预期。课设报告若只贴一张全绿扫描结果，却说不清处理了哪几条、为何忽略其余，这份静态分析就不算做完。

> **提问：**工具报告零缺陷。能不能据此说这段代码是安全的？

> **预设回答：**

- 答"能"——这是最危险的推论。零缺陷只表示没有命中规则。

- 答"不能，规则有限"——对。

- 追问：那报告还有什么用？——它是线索清单，帮你把注意力放到高风险的模式上，比人眼从头扫到尾便宜。

> **易错点 / 考点：**静态分析是静态测试手段。"静态"指不执行被测程序，不是"代码冻结"。判断题"静态分析可以发现运行时性能问题"——错。

> **过渡：**谱系很长。我们不背菜单，先看同一段代码会分别被谁拦下来。

【5.16】代码静态分析工具（2.5 分钟）

> 同一段 Java 代码，不同工具会说不同的话。想象一个方法：命名违规，catch 是空的，循环里每次都查库。某条路上还可能对空引用调方法。别背点名册。

>

> Checkstyle 管风格和规范。它看源码的写法像不像约定的那样，不管运行时会不会错。PMD 更偏源码上的坏味道：没用的变量、空的 catch、过于复杂的条件。它说的是"写得可疑"。FindBugs 走另一条路：看编译后的字节码，找缺陷模式。比如空引用、锁没按路径释放、equals 和 hashCode 不配对。语法合法，编译器放行，运行时可能出事。三者不能互相替代。2015 年 FindBugs 停更，社区继任者 SpotBugs 的仓库自述就是它的继任。Maven Central 上 2026 年 10 月仍能看到 4.10 系列。新项目别把停更的 FindBugs 当

> 门禁。

>

> 跨语言、坐在 IDE 里的，课件首推 SonarLint，不限 Java。C 和 C++ 一侧，课件列了 Clint、Oclint、Cpplint、TestC++。Oclint 更常见的写法是 OCLint。Python 一侧是 Pylint 和 PySonar2，前者偏规范，后者偏类型与语义。Coverity、Fortify 做更重的数据流分析，慢也贵，放夜间构建或发布前。对照看 OWASP 的源码分析工具清单。地址在 owasp.org 的 Source_Code_Analysis_Tools。

>

> 口诀：开发机上放轻的，流水线上放重的。课设用 SonarLint 或 Checkstyle 卡住本次变更。不必为名词去装 Coverity。

> **提问：**都是查 Java，FindBugs 和 Checkstyle 能不能互相替代?

> **预设回答：**

- 答"能，都是 Java 工具"——不能。一个看字节码里的缺陷模式，一个看源码写得规不规范。

- 答"不能，一个管缺陷模式，一个管风格"——对。严重程度和处理顺序都不同。

- 追问：PMD 放哪？——源码级坏味道，和前两者互补，不互相覆盖。

> **易错点 / 考点：**Java 三个工具的定位：Checkstyle 管风格，PMD 管源码坏味道，FindBugs 管字节码缺陷模式。SpotBugs 是后继，不是第四种新理论。SonarLint 不是只支持 Java。安全类工具是深度静态分析，不替代 IDE 里的轻量检查。

> **过渡：**谱系里被课件单独拿出来首推的，是 SonarLint。它到底多做了哪一件事?

【5.17】首推：SonarLint（2.5 分钟）

> 工具一多，"首推"听起来像广告。课件首推 SonarLint。产品页在 sonarsource.com/products/sonarlint。它给了三件事：发现缺陷、修正缺陷、学习。第三件才是它被课件首推的理由。

>

> 发现，而且是在你写的时候发现。检查从"提交之后"挪到"这句话还没写完"。改动还热着，成本几乎是零。等合并请求红了再改，你已经切换到下一个任务，上下文要重新装回脑子。

>

> 修正，是它经常能给出具体改法，而不只是一个红色标记。对新手，这比一句"你错了"有用。你要做的是读那条改法为什么成立，而不是无脑点接受。无脑接受，就和批准看不懂的生成代码是同一种习惯。

>

> 学习，是每条规则背后的解释：它在防什么，什么场景可以例外。用上几周，你等于有一个会针对你刚写的代码说话的规则教材。工具最值钱的地方，是让你少犯同一类错。例外也可以有，但你要说得出"为什么这里不适用"，不能说"它太烦"。

>

> 课件还强调，它几乎覆盖流行的语言和 IDE。团队里有人用 IntelliJ，有人用 VS Code，规则口径仍能对齐。这比"每台机器装一个只支持一种语言的插件"更接近评审要的共同语言。装上的第一周，屏幕会很吵。不要关掉。先把级别收到你最在意的严重问题，逐个处理。课程设计的约定可以是：本次改动不引入新的严重问题。老项目里的历史告警，不要求一夜清完。

提问：它既能发现问题，又能告诉你怎么改。能不能因此取消代码评审？

> **预设回答：**

- 答"差不多可以"——不行。它不会问这个需求该不该这样实现。

- 答"不能，规则有边界"——对。它能说"这里可能空指针"，不能说"利率公式本身错了"。

- 追问：它替代得了哪一部分？——替代机械的、已经写成规则的检查，把人的时间留给设计。

易错点 / 考点：三项能力是发现、修正、学习。不要和 SonarQube 混：SonarLint 在 IDE 里做本地、即时的检查；SonarQube 在服务端看整个项目。另一错法是以为它只在提交时才跑。它是边写边提示。

过渡：说"能进常见的 IDE"，我们就以 IntelliJ 为例，看装上之后你应该建立什么习惯。

【5.18】示例：在 IntelliJ 应用 SonarLint 插件（2 分钟）

> 插件装上了，屏幕开始画波浪线。你怎么对待这些提示，比你会不会点安装重要。课件在 IntelliJ 里演示了 SonarLint 的用法。插件市场条目是 7973。地址在 plugins.jetbrains.com 下面那条 7973-sonarlint。课件标题里的 SolarLint 是拼写错误，正确是 SonarLint。安装本身不是考点。考点是装完以后你怎么对待那些提示。

>

> 按严重级别看。先处理会在运行时出事的，再处理不好读的。把风格问题和空指针放在同一优先级，人会先被风格淹没，真问题留到合并之后。

>

> 点开规则说明再决定忽略。忽略是允许的，但要留一句理由。没有理由的忽略，下周会变成一百条，工具等于没装。

>

> 知道它看不到什么。IDE 里的插件做的是快而轻的分析，通常强在你正在编辑的范围，不强在跨模块的深度数据流。它说这个文件没问题，只表示在它的规则和它的视野里没说话。Coverity 那一类才把更重的分析放到流水线。两套视野不要互相代替。

>

> 不要把清空历史告警当成目标。一个老项目一扫几千条，一次改完，回归根本盖不住，你会修出新的缺陷。通行的做法是棘轮：新代码不新增问题，老代码碰到再收敛。怎么卡住新增？只对本次变更做检查，并把它放进合并前的自动步骤。这就是后面质量门禁的雏形。课程设计的仓库多半是新的，没有历史债。那就把棘轮设严一点：严重问题清到零再请同学评审。请同学看的，应该是设计和用例，不是一堆本可以自动消掉的风格问题。

提问：老项目一扫两千个问题。要不要一次改完？

预设回答：

- 答"一次改完才干净"——两千处改动，测试盖不住，很容易改出新错。

- 答"不改，反正能跑"——新代码会继续掉进同一个坑。

- 答"先卡住新增，再慢慢清旧的"——这是棘轮。对的。

- 追问：怎么卡住新增？——只检查这次变更，不过就不能合。

易错点 / 考点：这里考落地策略：分级、只对变更、新代码不新增。不考插件按钮。"清空所有告警"不是目标，那会造成大量与行为无关的改动。

过渡：每个人本地一套规则，团队仍会吵架。下面把本地检查和服务端的质量平台接上。

【5.19】可以和 SonarQube 集成使用（2 分钟）

> 两个名字差一个字母，职责差一层楼。课件写的 SolarQube 同样是笔误，正确是 SonarQube。分工先说清：SonarLint 在你的 IDE 里，看你正在写的代码；SonarQube 在服务端，看整个项目、各个版本的趋势，并可以当合并前的门禁。

>

> 不连起来，会出两个具体的亏。规则不一致。你本地用默认规则，觉得很干净；流水线用团队的规则集，一扫一片红。于是谁也不信谁。连上之后，本地跟随服务端的质量配置，写的时候和合的时候是同一套标准。服务端是权威，本地不要私自放宽。

>

> 没有全局视角。本地只能看见你打开的文件。服务端能看见技术债在涨还是在降、重复代码有多少、覆盖率是多少。这些趋势是为了决定下周还债还是继续加功能。

>

> 所以组合是两道关。第一道在写的时候：SonarLint 即时提醒。第二道在合的时候：SonarQube 按质量门禁决定这次变更能不能进主干。门禁要卡的是这次新增的问题，不是强迫把三年的历史告警一夜改完。那会制造巨大的无关差异，评审和回归都失真。覆盖率数字也可以挂在这道门上，但这儿先留一个缺口：行覆盖高，不代表断言抓得住错误。怎样用变异去检查断言的强度，是后半章的事。这一小节先抓住一句：门禁卡新增。

>

> 如果团队小到只能留一个：人少、仓库新，SonarLint 往往够用；多人、规则必须统一，才需要服务端。课程设计三人小组，本地插件加上合并请求里的自动检查，就已经够用。两道关的思想落到了你做得到的程度。不必为了名词去搭一整套服务器。

> **提问：**只能留一个，你留本地的 SonarLint，还是服务端的 SonarQube？依据是什么？

> **预设回答：**

- 答"留服务端"——覆盖面和门禁更完整，但作者要等到提交才知道问题。

- 答"留本地"——反馈快，团队口径和趋势管不住。

- 答"看人数：人少留本地，多人要服务端"——实用。选择跟着协作规模走。

- 追问：两个都留，谁说了算？——服务端配置是权威，本地跟随。

> **易错点 / 考点：**区别在四个维度：位置（IDE／服务端）、范围（正在写的代码／全项目）、时效（即时／趋势）、职责（个人反馈／团队门禁）。拼写是 Sonar，不是 Solar。

> **过渡：**下一个工具曾经很红，后来停了。它的停更，比它的菜单更值得讲。

【5.20】来自大学的 FindBugs（2 分钟）

> 这个工具是高校研究团队做出来的。项目页在 findbugs.sourceforge.net。技术路线也和 Checkstyle 不同。它不拿源码的格式做文章，它看编译之后的字节码，找的是缺陷模式。

>

> 为什么看字节码有道理？源码里有些写法"看起来像问题"，字节码更接近编译器交给虚拟机的东西。它抓的典型问题包括可能的空引用、锁在某些路径上没有释放。还有 equals 和 hashCode 不配对、算了一个值却没用。共同特点是：不违反语法，编译器放行，运行时可能出事。一条规则背后，往往是一类反复出现的事故。它把有经验的人的警惕，变成了可以重复执行的检查。

>

> 后半句更重要。2015 年 3 月，FindBugs 3.0.1 发布之后，就没有再更新。工具是有寿命的。JDK 在升级，字节码在变。停更意味着新的语言特性慢慢解析不了。新的缺陷模式也不会再补进来。

>

> 但缺陷模式不会跟着版本号过期。空引用、锁未释放、equals 不配对，现在仍在。社区把这条线接了下去，名字是 SpotBugs。仓库自述它是 FindBugs 的继任，而且仍在发版。停更给你的教训是：人要带走知识，不要抱着一个不再更新的程序。课程设计里如果还看到旧文档让你装 FindBugs，请改用仍在维护的 SpotBugs。也可以用 IDE、Sonar 系里同类的规则。说明里写一句：查的是缺陷模式，不是风格。只把它的报告当参考可以，不要把一个 2015 年停更的工具设成合并门禁。

> **提问：**一个工具已经停更十年，项目里还能不能用它当合并门禁？

> **预设回答：**

- 答"不能，新 JDK 可能跑不起来"——这是现实风险。门禁工具必须还在维护。

- 答"规则还在，可以当参考"——可以，但限定为辅助。发现一条仍有价值的规则，就把对这条规则的理解迁到仍在维护的工具里。

- 追问：该留下什么？——留下"缺陷长什么样"，不要留下"旧插件在哪个菜单"。

> **易错点 / 考点：**两个事实常一起考：基于字节码，以及 2015 年 3 月 3.0.1 之后停止更新。不要把它答成风格检查工具。风格是 Checkstyle 的事。SpotBugs 是继任者，不是 FindBugs 的新版本号。

> **过渡：**它在 IDE 里的位置，后来被 IDE 自己的检查收进去了。下面看这个"被收编"说明什么。

【5.21】FindBugs @ IntelliJ（2 分钟）

> 组里还有人在插件市场搜 FindBugs。搜得到，装完提示不兼容。IntelliJ IDEA 2021.2 发布之后，就基本把 FindBugs 干掉了。

>

> 为什么会被干掉？IDE 把这类检查内建了。你装一个插件，要配版本、跟着 JDK 升级，还会和 IDE 的检查重复报错。IDE 自带的检查开箱就能用，和编辑器绑在一起，跟着 IDE 一起升级。同等效果下，内建会赢过插件。

>

> 这件事有三层意思。工具会往平台里收。单点工具活得好不好，常常取决于平台愿不愿意自己做。测试领域反复出现同一件事。IDE 收了静态检查，CI 平台收了测试执行，代码托管平台收了评审。

>

> 能力没有消失，只是换了入口。字节码级的缺陷检查思路并没有死。它变成了 IDE 的 inspection，也变成了 SonarLint 这类工具里的规则。空引用、锁没按路径释放、equals 和 hashCode 不配对，这些模式还在。换的是入口，不是问题本身。

>

> 前面刚说 FindBugs 2015 年停更，社区用 SpotBugs 接着做。在 IntelliJ 里，连插件这条路也被 IDE 自己收

> 了。课设文档如果还让你装 FindBugs，请改用 IDE 自带的 inspection。也可以改用仍在维护的 SpotBugs。你要学的是缺陷长什么样，不是某个插件在第几个菜单。菜单会变，缺陷模式十年不变。

> **提问：**既然 IDE 自带检查就够了，我们为什么还要单独学 Checkstyle、SonarLint 这些工具？

> **预设回答：**

- 答"因为它们查得更全"——部分对。更要紧的是它们能在 IDE 之外跑：CI 上、构建脚本里，这些地方 IDE 不在场。

- 答"因为团队要统一标准"——很关键。IDE 自带的检查，每个人的配置可以不一样；CI 上的检查，全团队一个标准。

- 追问：所以工具的战场在哪？——在离开开发者电脑之后。这一句直接指向后面的持续集成。

> **易错点 / 考点：**考点是工具演进的原因。标准答法是 IDE 内建能力替代第三方插件、平台收敛单点工具。易错点：以为被替代就是被淘汰、没用了。被替代的是形态，不是能力。

> **过渡：**IDE 自带的检查到底能查出什么？下面用四个维度把边界划清。

【5.22】IntelliJ 分析功能代替了 FindBugs（2 分钟）

> IDE 自己会画波浪线。先别关，问一句：它到底能看见哪几类错？

>

> 第一类，语法与语义。变量声明了没用、方法返回值被忽略、不可能成立的条件、switch 少了分支。这类靠编译器前端就能发现，误报很少。

>

> 第二类，空值与可空性。可能为 null 的引用被直接使用，`@Nullable` 标注和实现不一致。这是日常最高频的一类。

>

> 第三类，资源与并发。流没关、锁在异常路径上没释放、在同步块里调外部服务。人眼很难稳定发现，工具却擅长。

>

> 第四类，风格与规范。命名、格式、注释缺失。这一层严格说不算缺陷，它影响可读性，所以也放进来了。

>

> 前三类会出事，第四类不好读。你在 IDE 里看到的提示，绝大多数都能归到这四个里。归好类，你就知道哪些要当场改，哪些可以排队改。把风格问题和空指针放在同一优先级，人会先被风格淹没，真问题留到合并之后。

>

> 还有一句边界。IDE 的分析是本地的、增量的、快的，它不做跨模块的全程序分析。它没报，只代表这一段代码在它能看到的范围内没问题。这一点和前面 SonarLint 的限制一样。它也看不见性能瓶颈、并发时序、业务逻辑错误、界面问题。那些必须跑起来才暴露。所以 IDE 检查是提高下限，不是证明程序对。

> **提问：**这四层里，哪一类问题最容易被"我本地跑得好好的"这句话盖过去？

> **预设回答：**

- 答"资源与并发"——很准。资源泄漏和竞态在单次运行、数据量小的时候几乎不显形。

- 答"空值"——也对，尤其是这次传进来的恰好不是 null。

- 追问：所以这类问题的检查应该放在哪？——放在流水线上做，不依赖运气。这就是持续集成的价值。

> **易错点 / 考点：**常考"静态分析能发现哪类缺陷、不能发现哪类"。能：空引用、资源未释放、未使用变量、可疑逻辑。不能：性能瓶颈、并发时序、业务逻辑错误、界面问题，这些必须动态测。答这类题要同时给出"能"和"不能"，只答一边会被扣分。

> **过渡：**IDE 检查是在提高下限。要把标准变成全组过同一道关，得靠专门的工具。

【5.23】代码风格检查 CheckStyle（3 分钟）

> 规范写在 wiki 里，没人看。写成 `checkstyle.xml`，合不进去。Checkstyle 的主页是 checkstyle.org。定位特别清楚：它只管代码风格与规范，不管程序会不会出错。

>

> 别因为只管风格就小看它。它最大的用处，是把规范从文档变成能执行的门禁。课件给的集成方式很具体。Maven 里跑 `mvn checkstyle:checkstyle`。Gradle 里 `apply from: 'checkstyle.gradle'`。规则写在 `config/checkstyle/checkstyle.xml`。跑 `./gradlew check` 就会执行检查。结果输出到 `build/reports/checkstyle/main.html`。还能接到 CI/CD。

>

> 你把这串连起来看，是一条很典型的工程链路。规则写成配置文件，构建工具调用，输出报告，报告进 CI 门禁。这条链路不是 Checkstyle 独有的。以后学别的静态检查，多半只是换个文件名。

>

> 为什么强调进 CI？只在本地跑，结果就是"我今天没空跑"。进了流水线，才是所有人过同一道关。规范一旦能执行、能强制，它就真了。

>

> 第一版规则别开太多。先开命名、行长、Javadoc、未使用导入这几类低争议的，跑顺了再逐步加严。一上来开一百条，团队第一天就会集体绕过它。能被执行的严格，比纸面上的完美重要。

提问：一个团队用了 Checkstyle 之后，代码风格统一了，是不是代码质量就提升了？

预设回答：

- 答"提升了"——要打个折。风格统一提升的是可读性和可维护性，跟有没有 Bug 没有直接关系。

- 答"没提升，只是整齐"——也不完全对。整齐带来的好处是真实的：评审时不用再讨论格式，注意力能放到逻辑上。

- 追问：那质量该怎么提？——靠评审抓意图、靠静态分析抓缺陷模式、靠测试抓行为。风格检查是起点，不是全部。

易错点 / 考点：Checkstyle 的定位是高频考点：只查风格规范，不查缺陷。常见的混淆是把 Checkstyle 和 PMD 搞反。PMD 会报空的 catch 块、没用的变量这类坏味道。Checkstyle 报的是行长超过 120、类名首字母没大写。另外，`./gradlew check` 与 `checkstyle:checkstyle` 的区别要能说清：一个是任务聚合，一个是直接生成报告。

过渡：规则那么多，具体都能查些什么？看一个规则示例，你就知道它的粒度有多细。

【5.24】CheckStyle 规则（2 分钟）

> 规则列表很长，别背。你要有的是分类感：它能在哪几个方向上卡你。

>

> 命名。类名、方法名、变量名、常量名的书写形式。这一类争议最小，收益也最直接。名字统一了，读代码的人不用每次重新适应。

>

> 格式与排版。缩进宽度、行长度上限、左花括号的位置、空行的使用、运算符两侧的空格。这一类最琐碎，也最适合机器管，因为它完全客观。

>

> 注释与文档。公共方法和公共类要不要写 Javadoc，注释的格式对不对。这一类会逼你写下这个方法干什么、参数什么含义。

>

> 结构与语句。if 要带花括号，switch 要有 default，不允许 `import *`，不允许空语句块。部分规则已经接近潜在缺陷。漏了 default，可能就是你没想到的情况。

>

> 度量。方法长度、参数个数、圈复杂度、嵌套深度。这一类是把前面讲的评审关注点变成自动检查。圈复杂度那个 e−n+2p，Checkstyle 可以直接把超过阈值的类和方法列出来。方法太长、太复杂、嵌套太深，本身就是缺陷温床。

>

> 五类看下来：Checkstyle 的规则，就是团队约定的可执行版本。你写一条规范，它就能查一条。命名和格式争议小，适合机器先管。结构语句开始靠近缺陷。度量直接对接前面讲的圈复杂度。评审时人去看设计和意图，机器去看已经写成规则的那些。规则不是越严越好。过度严格会诱发大量规避行为，要有取舍。

> **提问：**这五类里，哪一类最可能"查出来但你不服"？为什么？

> **预设回答：**

- 答"格式排版"——常见。比如行长度不超过 100，你写一个长正则就会被卡。

- 答"参数个数"——真实痛点。有时候参数多是业务决定的，硬拆反而更糟。

- 追问：遇到这种情况正确做法是什么？——要么在配置里给这一条加例外并写明理由，要么改设计。不能做的是悄悄在注释里加个忽略标记，谁也不知道为什么。

> **易错点 / 考点：**考点在规则的层次感。第五类度量规则和代码质量最相关：方法太长、太复杂、嵌套太深，本身就是缺陷温床。易错点：把 Checkstyle 的规则当成越严越好。过度严格会诱发大量规避行为，规则要有取舍。

> **过渡：**规则跑完会出报告，我们看一眼报告该怎么读。

【5.25】checkstyle 结果显示（2 分钟）

> 报告出来八百条红字，很多人第一反应是关掉。先别关，按四步读。

>

> 先看总数和分布。报告会按规则统计违规条数。你要问的第一个问题不是第一条是什么，而是哪条规则报得最多。修一条高频规则，往往能一次消掉几百个问题。这是投入产出比最高的动作。

>

> 再看严重级别。哪些是 error，哪些是 warning。团队应该约定：error 要修，warning 可以排队。没有这个约定，报告就是一堆没人看的红字。

>

> 第三步，定位到文件和行号。报告会给出具体的文件、行、规则名和说明。优先修新改动涉及的。老代码里的历史违规先记下来，不急着还。这就是前面说的新代码零新增。

>

> 第四步，把高频违规反推成规范问题。如果一条规则反复被踩，可能不是人的问题，而是规范本身不合理，或者没人在新人入职时讲过。这时候该改的是规范和培训，不是骂人。

>

> 报告的意义不在清零，在趋势。这个版本 800 条、下个版本 700 条，说明在收敛。这个版本 800、下个 1100，说明在恶化。看趋势比看绝对值有用。还有一句：违规数下降不等于质量提升。也可能是规则被关

> 掉了。所以看趋势的同时，要看规则集有没有被动过。新代码零新增，老代码碰到再收敛。报告是线索，不是分数。

> **提问：**报告显示某条规则报了 300 次，你第一反应是改代码还是改配置？

> **预设回答：**

- 答"改代码，规范就是规范"——如果这条规则合理，那就该改；但 300 次说明这是系统性问题，改起来要排期。

- 答"改配置，把它关掉"——要看情况。如果这条规则在本项目确实不适用，比如自动生成的代码，关掉是对的；如果只是嫌烦，那是自欺。

- 追问：怎么判断？——抽十条看。如果是同一类真实问题，改代码；如果是场景不适用，改配置并写清理由。

> **易错点 / 考点：**常考静态检查报告的使用方法，要点是分级、抓高频、看差异、看趋势。易错点：把违规数下降直接等同于质量提升——也可能是规则被关掉了。

> **过渡：**Checkstyle 管风格和度量，但度量里的复杂度它只能给个阈值判断。要看清复杂度，得用专门的度量工具。

【5.26】SourceMonitor 检测代码复杂度（2 分钟）

> 风格工具报的是不合规矩。还有一类工具不报错，只算数、画图、看趋势。SourceMonitor 就是做代码度量的。

>

> 它算哪些数？代码行数，包括总行数、有效代码行、注释行。有效代码行和注释行的比例，能看出这个模块好不好读。

>

> 语句数。比行数更能反映逻辑量。因为一行可以写三条语句，也可以三条语句拆成十行。

>

> 复杂度。每个方法、每个类的复杂度值，这是它的核心指标。

>

> 最大块深度。嵌套最多嵌套了几层。一个方法里 if 套 for 套 if 套 try，深度到五层，这段代码基本没人敢改。

>

> 注释比例，还有方法、类的数量。用来看结构和文档状况。注释比例不是越高越好。注释过多有时说明代码本身不清晰。重点看有效注释：解释为什么，而不是复述代码在做什么。

>

> 这些数字单独看没什么意思。关键是横向比和纵向比。横向比是同一个项目里哪个模块最重。纵向比是同一个模块这三个版本在变好还是变坏。高而稳定的模块，可以排期重构。每次发布都在涨的模块，越早拦成本越低。

>

> 所以这类工具的典型用法是：定期跑一次，看趋势图，找出复杂度突然跳高的那个模块。突然跳高通常意味着有人在一个地方堆了太多逻辑。那就是下一次评审、下一次重构的重点。度量工具的价值是定位热点，不是给分。

> **提问：**一个模块的复杂度一直很高但很稳定，另一个模块复杂度不高但每次发布都在涨，你先管哪个？

> **预设回答：**

- 答"先管高的"——直觉答案，但高而稳定的模块可能已经被测试覆盖得很扎实。

- 答"先管在涨的"——我倾向这个。上涨意味着失控趋势，越早拦成本越低。

- 追问：那高而稳定的就放着不管？——不是不管，是排期重构，而不是紧急处理。这就是用趋势做优先级判断。

> **易错点 / 考点：**常考代码度量指标有哪些，要点：行数、语句数、复杂度、嵌套深度、注释比例。易错点：把"注释比例越高越好"当成结论。注释过多有时说明代码本身不清晰。教材口径是适度，重点看有效注释：解释为什么，而不是复述代码在做什么。

> **过渡：**到目前为止，我们查的都是自己写的代码。可现代软件里，大部分代码不是你写的，是第三方库。

【5.27】软件第三方库风险巨大（3 分钟）

> 课设 pom 里那一长串依赖，有几条你说得出谁维护、有没有已知漏洞？

>

> 课件给了三个数字，量级先感受一下。98% 的正在使用开源第三方库的公司，并不了解它们。不是没用，是不了解。用了，但不知道用了什么、什么版本、谁在维护、有没有已知漏洞。

>

> 每年有 4000 个新增的开源第三方库漏洞被报告。漏洞不是一次性事件，是持续流入的。你今天扫干净了，下周新披露一个 CVE，你又中招了。

>

> 开源第三方库的使用比例在急速上升。2010 年是 10%，2015 年是 30%，2020 年是 90%。十年从一成到九成。今天一个系统里，你自己写的代码可能只占一成，九成是别人的。

>

> 你把九成的信任交给了一群你不认识、也没签合同的人。他们修不修、什么时候修、修了你要不要跟着升，这些你都不控制。

>

> 这就是供应链风险。不是某个库有 bug 这么简单。上游一个包被投毒或被披露高危漏洞，下游成千上万个系统同时中招，而你连自己用没用它都说不清。

>

> 用了一个方法，不等于只引入了一点风险。漏洞不看你怎么用，看它在不在依赖树里。你的系统边界，不是你代码的边界，是你依赖的边界。后面讲的 SCA 工具，就是为了把这条边界画出来。

> **提问：**你的项目引入了一个开源库，只用了一个工具类里的一个方法。这算不算引入了这个库的全部风险？

> **预设回答：**

- 答"不算，我只用了很小一部分"——这是最常见的误解。漏洞不看你怎么用，看它在不在你的依赖树里。它带的传递依赖、它的构建脚本、它的许可证，全都在你的交付物里。

- 答"算，依赖是整体的"——正确。你甚至可能没直接用到出问题的那段代码，但只要它被打进包里，攻击面就在。

- 追问：那怎么办？——要么升级到修复版本，要么换实现，要么明确接受并记录这个风险。三种都行，唯独

不能"不知道"。

> **易错点 / 考点：**这三个数字是硬考点，尤其"98% 的公司不了解自己在用的开源库"和"每年 4000 个新增漏洞"。判断题"只要不调用有漏洞的方法就没有风险"——错。另一个易错点：把第三方库风险和"开源不安全"划等号。问题不在开源，在缺乏可见性与持续跟踪。

> **过渡：**风险来自两个方向，自己写的代码和引用的库。安全分析也要分两条线。

【5.28】代码级安全分析对象（2.5 分钟）

> 安全问题从两个方向来。自己写错的，和引错版本的。

>

> 一条线是 SAST，静态应用安全测试。它分析的是你自己写的源代码，找的是未知漏洞。所谓未知，是这个漏洞还没有被公开编号，是你的代码里特有的写法问题。它依据的标准是 CWE，也就是通用缺陷枚举。比如注入、越界、路径穿越、不安全的反序列化。

>

> 另一条线是 SCA，软件成分分析。它分析的不是你的代码，是你引用的组件，找的是已知漏洞。已经有 CVE 编号、已经被公开披露、可能已经有修复版本的那些。它依据的标准是 CVE。

>

> 两条线的差别是未知和已知。你自己的代码写错了，世界上没有第二个人知道，只能靠分析。你引的库出问题了，全世界都知道，只差你有没有去比对。

>

> 课件还给了三个比例，20%、35%、45%。具体哪块对应哪个数字，看课件上的图。带走的结论是：两类都占相当比重，都不能偏废。课件上那张图把对象分成两块。一块是源码，一块是第三方组件。考试就考这组对应：SAST 对 CWE，SCA 对 CVE。

>

> 落到做法上：SAST 要接在提交和 CI 上，SCA 要接在依赖变更和定期扫描上。两条线都跑，才叫完整的代码级安全分析。只扫自己的代码，看不见库里的洞。只扫依赖，看不见你自己写出来的注入。

> **提问：**一个项目 SAST 扫出来零问题，SCA 扫出来十七个高危，你怎么判断该先做什么？

> **预设回答：**

- 答"先修 SCA，因为有编号说明很明确"——合理，已知漏洞有现成修复版本，性价比最高。

- 答"先修 SAST，自己的代码更可控"——要看具体漏洞：能被远程触发的注入，比一个只影响内部工具的库漏洞更急。

- 追问：判断依据是什么？——可达性与暴露面：这个漏洞在不在攻击者能碰到的路径上。有编号不代表一定危险，没编号也不代表一定安全。

> **易错点 / 考点：**SAST ↔ 未知漏洞 ↔ CWE、SCA ↔ 已知漏洞 ↔ CVE，这个对应关系是必考内容，四条要能一一对上。易错点：把 SCA 当成代码扫描。它不读你的代码逻辑，它读的是依赖清单与版本号。

> **【思政落点 3】**：第三方库这条链，给工科生上了一堂很现实的责任课。你引一个包，可能只是图省事。但这个包会被打进交付物，进入客户的机房、进入别人的账户体系。98% 的公司不知道自己用了什么。这句话背后不是技术问题，是知情意识的缺失。将来你们在银行、券商做系统，一份依赖清单就是一份责任清单。对自己交付的每一个组件说得出它是什么、谁维护、有什么风险，这叫职业底线，也叫对用户负责。

> **过渡：**SCA 这条线具体怎么做？下面看它的工作流程。

【5.29】SCA 工具（2 分钟）

> 你问一个上线三年的系统用了哪些开源组件、分别什么版本。能立刻答出来的组不多。

>

> SCA 工具的界面不重要，重要的是工作流程，一共四步。换哪家产品都一样。

>

> 第一步，生成依赖清单。工具先要弄清你到底用了什么：直接依赖、传递依赖、各自的版本号。这一步的产物现在有个正式名字叫 SBOM，软件物料清单。你可以把它理解成食品包装上的配料表。

>

> 第二步，比对漏洞库。拿着这份清单去和 CVE 库、厂商公告、开源社区的安全通告比对，看哪些版本命中了已知漏洞。

>

> 第三步，评估影响。命中了不代表一定可利用。这个组件是不是被打进运行包？出问题的那段代码在不在调用路径上？有没有其它防护措施？这一步要人来看，工具给的是候选清单。

>

> 第四步，给出修复建议并跟踪。通常是升级到某版本、换用替代库、加临时缓解措施。然后还要跟踪。漏洞库天天在更新，今天没问题不代表明天没问题，所以定期重扫少不了。

>

> 很多团队连"我用了什么"都答不出来。这不是技术能力问题，是可见性问题。SCA 的第一个贡献不是修漏洞，是让你看得见。传递依赖也算在你头上。你没有直接调用，它仍可能被打进包里。定期重扫，是因为漏洞库天天在更新。今天没问题，不代表下周没问题。

提问： SCA 报出某个传递依赖有高危漏洞，但你从来没有直接调用过它，该怎么处理？

> **预设回答：**

- 答"忽略，我又没用"——风险在于它可能被别的依赖调用，也可能在特定输入下被触发。

- 答"升级父依赖，让传递依赖跟着升"——这通常是正确路径，因为传递依赖你不能直接改版本，得从上游解。

- 追问：如果上游几年不更新呢？——那就是要评估替代方案，或记录并接受风险，把决定写进文档，而不是让它在清单里躺着没人管。

易错点 / 考点： SCA 的产出物是依赖清单加已知漏洞匹配结果，不是代码缺陷列表。常考的另一个点：传递依赖也算在你头上。很多人以为只要不直接引用就没事。

> **过渡：** SAST 和 SCA 都讲完了。这些工具在工程里到底什么时候用、用多重？

【5.30】代码分析工具常见的使用场景（2.5 分钟）

> 同一套分析，放在写的时候、提交的时候、发布前，强度完全不一样。课件把使用方式归成三类，还配了检测项和耗时。它告诉你什么场合用什么强度的检查。

>

> 在 IDE 中实时分析。你写代码的时候它就查了。特点是快，几秒钟。规则集可选有限个检测项，因为它要保证不卡你的编辑器。目的是为开发者快速返回检查结果，及时消除常见的安全问题。

>

> 代码每次提交时分析。提交或合并请求触发一次检查。速度比 IDE 慢，规则可以多一些。目的是在代码进入主干之前拦住问题。这是性价比最高的一道关。

>

> 与 CI/CD 流水线集成。每次构建、每晚定时跑全量。规则最全、分析最深，耗时可能就是几分钟。目的是守住发布口。

>

> 三类场景对应三种强度：实时查轻的，提交查中的，发布前查重的。三种使用方式，对应三种使用目的。写的时候快速反馈，提交时拦住入库，流水线上守住发布口。这个三档以后设计任何质量体系都能套用。

>

> 再看检测项举例：SQL 注入、跨站脚本攻击、资源泄露、硬编码凭据、配置审查。这里既有代码问题，也有配置问题。现代代码分析不只看 `.java`，还看配置文件。

>

> 规则集、检测项、检测耗时，三者互相制约。规则越多，耗时越长，能放的位置就越靠后。这就是为什么重型工具只能放在流水线末尾。

提问：如果只在 CI 上做全量分析，不在提交时做，会有什么后果？

预设回答：

- 答"没后果，反正最后都会查"——有后果：问题会堆积。等到发布前才发现两百个问题，你只能选择延期或者忽略，两种都很糟。

- 答"反馈太慢，改起来成本高"——对。提交时发现问题，改动只要几分钟；发布前发现，要重新走一遍测试。

- 追问：那反过来，只在 IDE 里查呢？——更糟，因为 IDE 里的规则是个人配置，谁都可以关掉，没有约束力。

易错点 / 考点：三档场景与"耗时—规则数—位置"的制约关系是核心考点。易错点：以为检测项越多越好。实际工程里要按场景裁剪规则集，否则要么卡死开发机，要么拖垮流水线。

> **过渡：**这些 SAST 工具内部到底怎么看懂你的代码？下面按数据流向走一遍。

【5.31】SAST 工具通用框架（3 分钟）

工具怎么看懂你的代码？跟着箭头走一遍就清楚。

>

> 起点是被测源程序。第一步是解析：把源代码变成抽象语法树，也就是把文本变成结构。到这一步为止，工具只知道语法，不知道语义。

>

> 第二步是基本分析方法，包括数据流分析。共同目的是压缩程序状态空间。函数调用关系。SSA，全称 Static Single-Assignment，静态单赋值。每个变量赋值都变成唯一版本，方便追踪数据从哪来到哪去。指向分析，这个引用可能指向哪些对象。前后支配，哪条语句一定在另一条之前执行。Mod-effect，一个函数改了哪些外部状态。多态分析。全局变量分析。抽象解释，用抽象值概括所有可能的运行状态。符号执行，把输入当成符号，推演路径条件。图可达，从可控输入到危险操作之间有没有路径。

>

> 第三步，缺陷模式。工具不是凭感觉报错。它有一套基于缺陷模式的逻辑表达式，来源是 OWASP、CERT、CWE 这些标准。CWE 全称 Common Weakness Enumeration。比如外部可控的字符串未经校验就进入 SQL 拼接，这就是一条模式。

>

> 第四步，逻辑表达式求解，常用 SMT 求解器。把这条路径的条件和缺陷模式的条件放在一起解。能解出满足条件的输入，才算可能发生。

>

> 第五步，检测结果。到这里它才给你报一条。中间每一步都在做近似，所以结果永远是疑似，需要人去确认。指向分析可能算多了，抽象解释可能放宽了，路径爆炸时只能截断。这也是为什么它不能替代评审和测试。

提问：既然每一步都是近似、必然有误报，那为什么还要用它？

> **预设回答：**

- 答"因为人更不靠谱"——有点糙，但接近事实：人看不完几十万行，工具能。

- 答"因为它不漏那些规则覆盖的路径"——对，工具的价值是稳定与覆盖，不是精确。

- 追问：那怎么用好一个会误报的工具？——降低噪音：调规则集、按可达性排序、只对变更做门禁。工具的价值取决于你怎么用它，而不是它出厂什么样。

> **易错点 / 考点：**考点是流程顺序和关键术语：AST、SSA（Static Single-Assignment）、CWE（Common Weakness Enumeration）、SMT。常考"SAST 的第一步是什么"——解析成抽象语法树；"误报产生的原因"——分析过程做了近似，或路径爆炸被截断。易错点：把 SMT 求解当成少不了的一步。某些轻量工具跳过它，代价是误报更高。

> **过渡：**第一部分讲完了。有些东西不跑起来是真看不出来。下面进入第二部分：单元测试。

【5.32】5.2 单元测试（1 分钟）

> 前面一直在看代码。从这儿起，开始跑。跑的对象很小，一个方法、一个类。但要求很硬：你要能明确说出它应该是什么行为，然后用一段代码去证明。

>

> 为什么它重要？它能在缺陷还没跑远的时候把它按住。到了集成、到了系统，一个缺陷混在一堆模块里，定位成本是几十倍。对象最小，反馈最快，定位最准。

>

> 这一部分按六个问题讲。为什么要做？要做到什么程度？上下游没有怎么办，也就是驱动和桩。用什么工具写？依赖对象怎么处理，也就是 Mock。怎么知道测得够不够，覆盖率。

>

> 这六个问题，就是你写第一个测试时会依次卡住的地方。卡住很正常，我们一个个解。为什么做，决定你肯不肯花这个时间。做到什么程度，决定你何时停手。上下游没有，决定你测不测得动。

>

> 单元测试看的是最小那一块还对不对。页面好不好看、接口好不好用，是后面几章的事。今天的出口更具体：拿到一个方法，写出第一批能跑的测试，并且说得清为什么是这几组数据。

>

> 单元测试和生产代码谁先写，后面会碰到。常见做法是先写实现再补测试。风险是测试变成照着实现写，验证的是代码现在这么干，不是它应该这么干。先写测试也行，好处是你先在脑子里把什么叫对想清楚。两种都能用。关键是测试和实现不要同一个人照着同一份理解抄一遍。

提问：单元测试和生产代码，谁应该先写？

预设回答：

- 答"当然先写代码"——最常见。先写实现再补测试，测试往往变成照着实现写，验证的是代码现在这么干，而不是它应该这么干。

- 答"先写测试"——这就是 TDD 的思路，好处是你先在脑子里把"什么叫对"想清楚。

- 追问：两种都能用吗？——能。关键是测试和实现不要同一个人照着同一份理解抄一遍，否则你只是把代码翻译了一次。

易错点 / 考点：单元测试的"单元"指可独立测试的最小功能单位，通常是一个方法或一个类，不是一个模块、一个服务。判断题"单元测试由测试人员独立完成"——错，主责是开发人员。

过渡：这一部分具体包括哪几块？我们看它的内容清单。

【5.33】单元测试内容（2.5 分钟）

> 写第一个单元测试，你会依次卡在六个地方。这六项就是这一部分要解的题。哪项会考、哪项要动手，我一并交代。

>

> 为什么要做单元测试。讲价值与成本。看着像务虚，其实最容易被问到到底值不值。下面专门讲。

>

> 单元测试的目标和要求。讲测到什么程度算完。这一项要能背：功能、输入输出的容错、内部数据完整性、覆盖率指标。

>

> 驱动程序和桩程序。被测的那个方法，上下游还没有怎么办。驱动是我来调你，桩是你调别人、别人先用假的顶上。理解这两个词，后面 Mock 就顺了。

>

> 单元测试工具 JUnit 及其实践。这是你要动手的部分。JUnit 5 的架构、注解、断言、执行器，一条条都要会用。

>

> Mock 技术与框架，如 Mockito。依赖的对象怎么造。这是本章的难点，也是面试高频题。

>

> 测试覆盖率工具 JaCoCo。我怎么知道测得够不够。关键不是会点按钮，是看得懂报告、知道行覆盖和分支覆盖的差别。

>

> 六项合起来是一条完整的作业链。想清楚为什么测，定标准，解决上下游，写出来，处理依赖，度量效果。目标和 JUnit 要能背、能写。驱动与桩、Mock 是难点。覆盖率是度量。你实验交的那份测试代码，就是这条链的产物。

提问：六项里，你觉得哪一项最容易被跳过？

> **预设回答：**

- 答"覆盖率"——不少团队跑完不看报告，或者只看一个总数。

- 答"驱动和桩"——很多同学一上手就卡在这：被测方法要连数据库，怎么办？

- 追问：跳过哪一项后果最严重？——跳过"目标与要求"最严重，因为你连测到什么程度算完都没定义，后面全是凭感觉。

易错点 / 考点：这六项是本章的目录级考点，简答题常问"单元测试包括哪些内容"。易错点：把驱动程序和桩程序当成工具名。它们是测试代码的角色，不是软件产品。

【思政落点 4】：这六项排下来，你会发现单元测试是件不显眼的活。没有界面、没有演示效果、写完了也没人鼓掌。可它恰恰是把每个零件钉牢的那部分工作。我们这门课讲课程设计，很多同学愿意花两周做界面、花两天写测试。决定一个系统能活多久的，往往不是它第一天多好看，而是它的每个零件经不经得起推敲。

> 愿意在看不见的地方下功夫，这就是工匠精神最朴素的样子。不是把活干得漂亮，是把活干得住。

> **过渡：**第一项我们先讲透。为什么非得在单元这一层花这个力气？

【5.34】为何要进行单元测试？（3 分钟）

> 编译过了，以为能交差。行业常态是：编程与编译运行结束后，每 1000 行代码中大约残留有 2 到 6 个 Bug。不是新手代码，是常态。一个五万行的系统，天生就带着一两百个缺陷等着你。缺陷不是意外，是常态。

>

> 寻找与修改程序错误的代价，占总体开发投资的 30% 到 60%。开发投入里有三到六成其实花在找错和改错上。测试不是额外成本，它就是开发成本的主要组成。区别只在你在便宜的环节花，还是在贵的环节花。

>

> 尽早发现错误，成本越低。同一个缺陷，在单元阶段发现，改一行代码。在系统测试阶段发现，要重新部署、重造数据、重新回归。上线后发现，要发版、要道歉，可能还要赔钱。

>

> 发现问题比较容易，修正问题更容易。单元质量是系统质量的基石。为什么容易？单元小、边界清楚、能直接构造输入。课件还补了一句：单元测试可以比系统测试测得更彻底。系统测试受限于场景，方法内部的分支组合你构造不出来，单元测试可以逐个分支喂数据。

>

> 再看这两个算式。0.83 的 30 次方，约等于 0.00123。0.99 的 30 次方，约等于 0.74。一个系统由 30 个单元串起来，每个单元通过率 83%，整体一次跑通的概率只有千分之一左右。每个单元做到 99%，整体也才 74%。

>

> 单元层面的缺陷率，会被集成按指数放大。指望系统测试阶段整体调通，等于指望 30 个不靠谱零件拼出一台靠谱机器。质量得在单元这一层做上去。它是能让后面几层站得住的地方。

> **提问：**如果每个单元的通过率都是 99%，30 个单元串起来整体只有 74%。那你觉得"每个单元 99%"这个目标够不够？

> **预设回答：**

- 答"够了，99% 已经很高"——单看一个单元很高，串起来就掉到七成。这说明"高"是相对的，要按整体目标倒推单元目标。

- 答"不够，得靠集成测试补"——集成测试能兜一部分，但它兜不住组合爆炸。正确做法是两头一起做：单元抬高，集成专项覆盖接口。

- 追问：那 100% 呢？——100% 覆盖率不等于 100% 正确，这一点后面讲覆盖率时会专门说。目标不是数字好看，是风险可控。

> **易错点 / 考点：**两个数字必背：每 1000 行残留 2–6 个 Bug、找错改错占总投入 30%–60%；两个算式要会解释：$0.83^{30}\approx0.00123$、$0.99^{30}\approx0.74$。易错点：把 0.83^{30} 理解成缺陷率。它算的是整体一次通过的联合概率，含义是每个环节都成的概率相乘。

> **【思政落点 5】**：这两个算式其实是成本意识的数学表达。工程上最贵的一句话是"先这样，回头再说"。因为回头意味着这个缺陷已经流到了更贵的环节。质量不是免费的，但劣质更贵。花时间在单元测试上，看起来是慢，实际上是把后面几轮返工的钱提前省下来。这种算后面几轮、不只算眼前一小时的习惯，将来你们做技术决策、做预算，都用得上。

> **过渡：**知道了为什么必须测，下一个问题就来了。如果它只是抓 bug，凭什么每个函数都配一条用例？

【5.35】单元测试的其它价值（3 分钟）

> 抓 bug 只是第一条。课件还列了七条价值，很多人只会背前面那条。

>

> 支持扩展。没有各种类型测试的支持，软件开发无法支持扩展，单元测试就是它的基础。有一套能跑的单元测试，改完跑绿了就敢提测。没有的话，没人敢动那块代码。

>

> 引导更好的设计。能够支持单元测试的代码总是不会被人遗弃。你要单独拿一个方法来跑，就会照出三种病。参数太多。Monster Methods，巨型方法。依赖过多。一拆，设计就好了。

>

> 支持变化。谁改系统的一部分，都得知道其它哪些部分还要跑。单元测试给了这份名单。我们改代码的时候，不必担心无法预料的破坏。

>

> 避免回归缺陷。集成为一个庞大系统后，靠系统测试穷尽所有的回归测试几乎是不可能的。单元测试几秒就能跑到每个分支。

>

> 保证工作的平稳步调。代码和测试配合着写，很少有最后一分钟的意外。节省测试时间。边界值、输入校验、主逻辑路径，用单元测试做最简单、最快、最廉价。

>

> 规范行为和记录代码。理想状态下，单元测试就是被测代码的行为描述。它也是一个例子，解释代码怎么工作、怎么实现某条业务逻辑。跑不通就会红，这份说明不会过期。

>

> 课件还引了一句。人们总愿意写"工作完全正确"的代码，而不是写"按照预期的设想工作"的代码。可证明程序完全正确几乎不可能，除非是大学教材里有关形式化方法的非常简单的代码片段。单元测试的目的不是证明正确，是持续提供证据。

提问：你接手一个跑了五年、一行测试都没有的模块，领导要你两周内加一个新功能，你的第一步做什么？

> **预设回答：**

- 答"先读代码，读懂了再改"——必要，但不够。读懂的判据不是"我觉得懂了"，而是"我能给它写出一条通过的测试"。

- 答"先补几条测试再动手"——这是正解。业内叫特征测试：先把现有行为钉住，再改。等于给旧代码穿一件防护服。

- 答"先重构，重构完再加功能"——顺序危险。没有测试保护的重构不是重构，是赌博，赌你还记得每一处隐含依赖。

- 收口：这三条其实是一条链。读懂了才写得出测试，有了测试才敢重构，重构干净了才好加功能。

易错点 / 考点：多选题考"单元测试的价值"，学生最容易漏掉"引导更好的设计"和"规范行为和记录代码"这两

条，因为它们听起来不像测试的活。判断题"单元测试只能发现功能缺陷"是错的，它同样能暴露设计与可测性问题。记忆口诀：扩展、设计、变化、回归、步调、省时、文档。

过渡：价值讲起来容易空，落到目标就具体了。下面四条，第四条那个数字最容易被误用。

【5.36】单元测试的目标（3 分钟）

测到什么程度算完？四条目标，第四条那个数字最容易被误用。

>

单元模块被正确实现，主要指编码，包括功能、性能、安全性。课件补一句：一般主要介绍单元功能测试。性能受调用方式影响，安全要看整个数据流，单元层面难以完整验证。但一般不测不等于不该测。一个正则写错导致回溯爆炸，一处 SQL 拼接没参数化，都是单元层面爆的。功能是主战场，性能和安全做基础检查。

>

参数输入是否正确传递和得到保护，也就是容错；输出是否正常。这条最容易被做成只测好数据。空值、超长字符串、类型边界、非法格式进来时有没有被挡住，还是直接把脏数据带给下游。线上大量事故，根子不在算错，而在根本没检查输入。

>

内部数据能否保持完整性。包括变量的正确定义与引用、内存及时释放、全局变量的正确处理和影响最低。这是单元测试独有的地盘。系统测试从外面只能看输入输出，看不到这个对象释放了没有。

>

覆盖率达标。代码行覆盖、分支覆盖或 MC/DC 达到要求，比如高于 80% 或 95%。行覆盖看这一行执行过没有。分支覆盖看判定为真为假两条路都走过没有。MC/DC 是修正条件/判定覆盖。它要求每个条件都能独立影响判定结果。航空、轨交这类高安全领域常用。80% 和 95% 不是同一条及格线，而是不同完整性等级的要求。一般业务系统 80% 行覆盖是基本盘，安全关键模块要 95% 以上，甚至要求 MC/DC。

>

覆盖率是必要不充分条件。100% 只说明每行都跑过，不说明每行都断言过。一段代码跑了、你没写断言，它错了你照样绿。决定质量的不是覆盖率数字，是你断言了什么。

> **提问：**一个模块行覆盖率 100%、分支覆盖率 100%，能不能说它测试充分了？请给一个反例。

> **预设回答：**

- 答"可以，都 100% 了"——这是最典型的错答。覆盖率只记录执行路径，不记录判断依据。

- 答"不能，可能没写断言"——对。执行到但没有断言的代码，等于走过场。

- 答"不能，可能用例的输入选得不对"——也对，而且更狠：输入全用正常值，异常分支靠抛异常顺便覆盖过去，覆盖面好看，缺陷还在。

- 收口：所以覆盖率要配着断言密度和输入设计一起看，这三件套齐了才算测试充分。

> **易错点 / 考点：**选择题最爱考"单元测试的目标包括哪些"，正确答案四条都要选上，包括"内部数据完整性"这条抽象表述。另一个高频陷阱是把 MC/DC 说成多条件覆盖。它的全称是 Modified Condition/Decision Coverage，核心是每个条件能独立改变判定结果，比条件组合覆盖弱、比判定/条件覆盖强。

> **【思政落点 6】**：覆盖率这个数字，最考验一个人的诚实。因为它太好看了。写一条把整个方法跑一遍的用例、不写一句断言，覆盖率立刻 90%，报告很漂亮，缺陷一个没抓。工程上这叫用指标交差，代价是把风险留给了用户。指标是给管理看的，断言才是给质量看的。做技术的人，宁可报告上写 72%，也不要拿 100% 去骗自己和团队。这种对数据负责、不糊弄的态度，是工程师最基础的职业底线。

> **过渡：**目标讲完了，我们马上动手。下面这段只有十几行，四个人里有三个第一次看不出来。

【5.37】一个小练习（2 分钟）

> 先看这段只有十几行的代码。类名 `Calculator`，方法 `add`。需求原文：实现两个正整数或零的加法，如果其中有 1 个数小于或等于 0，则返回 0。你能一眼看出它错在哪吗？多半不能。

>

> 先读判定：`if(num1 >= 0 && num2 > 0)`。`num1` 是大于等于 0，`num2` 是大于 0。左边带等号，右边没带。而需求说零是合法的。所以 `num2 = 0` 时条件为假，程序走进 else 返回 0。正确答案应该是 `5 + 0 = 5`。错误就在这一个等号上。

>

> 这个 bug 为什么难发现？因为平时测函数下意识都用正常值。`add(2,3)` 得 5。`add(10,20)` 得

> 30。`add(-1,5)` 得 0。三个都对，测试通过，代码提交。要暴露它得用边界值：`add(5,0)` 期望 5、实际 0，红了。零不是随便一个数，零是两个合法区间的交界点。

>

> 再看覆盖。要让判定两条分支都跑到，最少两组数据：`(5,3)` 为真、`(-1,5)` 为假。可判定覆盖做到了，这个 bug 依然活着。因为 `num2=0` 这一格在"假"那一大类里被 `(-1,5)` 代表了。要让每个条件都取到真和假，得补 `(5,0)`：`num1` 真、`num2` 假。这就是条件覆盖比判定覆盖更强的地方。也解释了为什么分支覆盖率 100% 的代码照样有 bug。

>

> 修法不是把 `>` 改成 `>=` 就完事。先做参数校验 `if (num1 < 0 || num2 < 0) return 0;` 再放心相加。边界条件写在开头，比藏在复合判定里安全。请当堂写三件事：错误在哪一行、暴露它的数据是什么、修好之后补哪几条用例。

提问： `add(5, 0)` 这条用例，它同时完成了哪两种覆盖？它没完成哪一种？

预设回答：

- 答"完成了判定覆盖"——只对了一半。它让判定取到"假"，但"真"还得靠 `(5,3)` 那一条，两条合起来才叫判定覆盖。

- 答"完成了条件覆盖"——对。`num1>=0` 取真、`num2>0` 取假，两个条件都取到了真假中的一个；配合 `(-1,5)` 就齐了。

- 答"没完成路径覆盖/组合覆盖"——也对，四种条件组合只走到一部分，要全覆盖得再加 `(0,0)`、`(0,3)` 这类数据。

易错点 / 考点： 这道题几乎年年考，两个坑：一是答"没发现错误"，二是答"错误是 num1 应该写成 >0"，把对称当正确，方向搞反。答题模板固定三句：错误位置（`num2>0` 的边界）→ 暴露数据（`num2=0`，如 `add(5,0)`）→ 预期与实际（预期 5，实际 0）。判断口诀：看边界，不看中间；等号两边，必须对齐需求。

过渡： 单个函数好测，因为它没有邻居。真实函数总要调用别人、也被别人调用。下面开始搭假邻居。

【5.38】驱动程序与桩程序（2 分钟）

> 单个函数好测，因为它没有邻居。真实函数上面有人调，下面它调别人。

>

> 单元测试的对象是一个单元，可能是一个函数、一个类。可现实中的单元不是孤岛。上面有调用者，下面有被调用者，要查数据库、发消息、调支付接口。做单元测试时，上下两头都不在。调用者还没写好，或者我不想连带它一起测。被调用者可能是别人的模块、可能是生产库、可能是收费的第三方接口。

>

> 怎么办？上面缺一个假调用者，下面缺一个假被调用者。假调用者就是驱动程序 Driver：准备数据、调用被测单元、拿回结果。假被调用者就是桩程序 Stub：顶替真实的下游，按我需要的样子返回结果。

>

> 你要考一个学生的计算能力，又不想让其他环节干扰他。找个人专门出题收卷，这是驱动。再把查表、计算器换成一份固定答案表，让他不用真去查，这是桩。驱动是上游的替身，桩是下游的替身。一个喂进来，一个顶出去。

>

> 这两个词的方位，考试最爱设陷阱。驱动在被测单元之上，它扮演主程序，通常由测试框架提供。你写的 `@Test` 方法本身就是驱动。桩在被测单元之下，它扮演被依赖者，需要你自己写小类去冒充。这两个脚手架不在被测代码里，在测试代码里。

> **提问：**你写了一个 `@Test` 方法，里面 new 了一个被测类、传了两个参数、断言返回值。这里面哪些部分相当于驱动，哪些相当于桩？

> **预设回答：**

- 答"整个测试方法就是驱动"——对，JUnit 的 Runner 加上你的测试方法，共同构成驱动。

- 答"没用到桩，因为这个类没有下游依赖"——正确。桩不是必需品，只有被测单元真的调用了外部协作者时才需要。

- 答"传进去的两个参数就是桩"——概念串了。参数是数据，桩是对象，一个实现了相同接口的替身类。这个区分后面讲 Mock 时还会用到。

> **易错点 / 考点：**名词方位必考。判断三步法：看它调用谁。它调用被测单元，是驱动；被测单元调用它，是桩。另一个高频错答是把桩和 Mock 当成同义词，这个差别我们在后面专门讲。

> **过渡：**光说方位还是抽象，我们看那张三层结构图，一眼就明白。

【5.39】驱动程序与桩程序（2 分钟）

三层结构，从中间说起。中间那个才是你要测的，上下两层都是假的。

>

> 中间一层是被测单元，Function under test。我们要测的那个函数或类。它是舞台中央的主角，不动。

>

> 上面一层是 Driver，驱动程序。职责三件：准备输入数据、调用被测单元、接收并判断输出。所以你写的 `@Test` 方法、那个 `main` 函数、测试框架的 Runner，都是驱动。驱动是可以通用的。框架把调用、捕获、比对、报告都封装好了，你只管填数据和期望值。

>

> 下面一层是 Stub，桩程序。被测单元运行时要去调用别人。那些别人在单元测试环境里通常不可用，或不该用。数据库没测试数据。支付接口不能真扣钱。消息队列没起。桩就顶在那儿，用固定返回值替代真实调用。图上的箭头方向很关键：被测单元指向桩，说明是被测单元去调用桩。

>

> 为什么一定要这么搭？隔离：一个单元测失败，你要能立刻确定问题在它自己身上，而不是可能数据库挂了。可控：真实下游你控制不了，它今天成功明天超时，你的测试就时红时绿。桩把下游返回什么变成一行代码。速度：连库的一条用例几百毫秒，用桩是微秒级。几千条用例就是半小时和几秒的差距。

>

> 什么时候要上桩？满足三条之一。跨进程：网络、数据库、消息。不可重复：时间、随机数、流水号。有副作用：扣款、发短信、写外部日志。三条都不满足的，老实调真的。

> **提问：**如果一个被测方法里调用了 `System.currentTimeMillis()` 取当前时间，你要不要为它做桩？为什么？

> **预设回答：**

- 答"不用，取个时间而已"——那就踩坑了。这类测试会在跨月、跨年、闰秒、夏令时那天莫名其妙地红，而且没人查得出原因。

- 答"要，因为时间不可重复"——正确，这属于不可重复那一类。做法是把取时间抽成一个可注入的 Clock 接口，测试时注入固定时间。

- 追问：这又回到前面那条"引导更好的设计"。为了可测，你把时间依赖抽出来了，代码反而更干净。测试逼设计，就是这个意思。

> **易错点 / 考点：**图上的箭头方向是考点：驱动 → 被测单元 → 桩，一条链。如果考生画成"桩 → 被测单元 → 驱动"，说明方位全反了。另常考"驱动和桩由谁来写"：驱动多由测试框架提供，也可手写；桩必须由测试人员自己写，因为只有你知道下游该返回什么。

> **【思政落点 7】**：驱动和桩，做的是为协作搭桥。被测单元本身没变，是周围的环境被补出来了。上游有人喂数据，下游有人应答。这跟我们做团队项目一模一样：一件事推不动，往往不是这个人不行，而是上下游没接上。愿意去写那个假的下游的人，其实是在替整个团队承担衔接成本。这种角色不一定出现在架构图里，但没有他，谁都跑不起来。这种我来补位的意识，就是协作精神最实的样子。

> **过渡：**桩写起来容易，写好很难。最常见的写法是返回一个写死的 5。下面看为什么这不聪明。

【5.40】桩程序的灵活性（2 分钟）

> 最常见的桩，是 return 一个写死的 5。课件说这是最简单、也最不聪明的做法。

>

> 看这段 C# 代码，类名是 `ParameterizedStub`，参数化的桩。它实现了协作接口 `ICollaborator`。构造函数收一个 `value` 存起来。方法名是 `ComputeAndReturnValue()`。`value` 小于 10 就抛 `InvalidOperationException`。否则返回 `value`。返回什么由构造时传进来的参数决定，不是写死在方法体里。

>

> 课件底下那句话很直接。不会像上一页返回一个硬编码数值。让一个桩对象返回单个数值，是最简单、也最

> 不聪明的做法。其它测试迟早都需要返回一个不同的数值。

>

> 为什么不聪明？第一个测试要返回 100，你写个 `return 100` 的桩。第二个要返回 0，再写一个类。第三个要抛异常，第三个类。三个测试三个类，代码量比被测代码还多，而且大半是一样的。这叫测试代码的复制粘贴。接口一改，你要改三处。

>

> 参数化之后一个类搞定。`new ParameterizedStub(100)` 返回 100。传 0 返回 0。传 5 就抛异常。变化的部分从类的个数变成了构造参数。这就是灵活性。

>

> 这段代码还有两个细节。它把异常路径也做进了桩。真实下游会失败，你的桩也要能失败，否则容错代码永远测不到。它的判断逻辑是假业务，但这是刻意的。桩只需要满足测试需要的行为，不必还原真实逻辑。桩的价值不在能返回，而在能按需返回。写桩前先问自己：下一个测试会不会要一个不同的值？会，就别硬编码。

提问：桩里能不能写业务判断逻辑，比如"金额大于 1000 就打九折"？写了会不会让测试变假？

预设回答：

- 答"不能，桩要越简单越好"——方向对，但过严。桩里可以写逻辑，前提是这段逻辑只为测试服务、且你能说清它模拟的是什么。

- 答"能，只要能造出想要的场景"——可以接受，但要当心：桩里的逻辑一旦复杂，你就在测桩而不是测被测代码，缺陷会被桩掩盖。

- 收口：判断标准是这段逻辑会不会影响断言的正确性。会，就说明它该搬到参数里；不会，写一点无妨。

易错点 / 考点：常考"硬编码桩的问题"。标准答案是复用性差、用例一多就要写多个桩类、维护成本高。另一个易错点是把参数化桩和 Mock 的 when 配置混为一谈：前者是你自己写的类，后者是框架动态生成的代理对象，机制不同、效果相近。

过渡：到这里，单元测试的为什么测、测什么、怎么搭台子都齐了。接下来进入工具，从一份工具清单开始。

【5.41】单元测试工具及其应用（3 分钟）

> 几百个函数摆在面前。你一个个手工点，点不完。系统测试还能靠人点，也能靠脚本点。单元测试对着的是函数。数量一上去，人就跟不上。

>

> 业内有个说法。没有框架，就没有单元测试。你得找出用例，跑起来，抓住异常，再汇总报告。这四件自己凑齐，凑出来的仍是一套框架。所以在各类测试里，单元测试最离不开工具。

>

> 后面四类会反复出现。先对上号。第一类是单元测试框架。Java 用 JUnit，也用 TestNG。C++ 用 GoogleTest。Python 用 pytest。它们组织用例，执行，做断言，出报告。没有这一层，后面三类没处挂。

>

> 第二类是测试替身和 Mock 框架。名字有 JMockit、Mockito、EasyMock。被测代码要调别人，你造一个假的协作者顶上。手写桩能做这件事。框架是把桩做成现成的。没有替身，带依赖的类往往跑不动。

>

> 第三类是用例自动生成。EvoSuite 和 Diffblue Cover 在这一类。有的靠遗传算法搜输入。有的靠符号执行。它们能把覆盖率拉高。可断言常常照着当前行为抄。代码里原有的错，会被写成期望。所以它们是起点，不是终点。人还得审。

>

> 第四类是覆盖率工具。JaCoCo，gcov 配上 lcov，还有 Coverage.py。它们回答哪一行被跑到了。只有框架、没有覆盖率，你不知道测够没有。只有覆盖率、没有替身，复杂一点的类照样测不动。

>

> 四类可以串起来。框架让你写得出。替身让你写得下去。生成工具让你写得快。覆盖率告诉你够不够。课上和实验走 Java。JUnit 5，加上 Mockito，加上 JaCoCo。先把这一套用熟。

> **提问：**如果团队只允许你引入一个单元测试相关的工具，你选哪一个？为什么？

> **预设回答：**

- 答"覆盖率工具"——想先看现状，可以理解。没有框架就没有用例可跑，覆盖率会是 0。

- 答"测试框架，比如 JUnit"——多数时候就选它。别的三类都挂在它上面。

- 答"自动生成，一步到位"——风险最大。生成的断言常把当前行为抄成期望，错也会被焊进去。

> **易错点 / 考点：**考点是工具分类。EvoSuite 和 Diffblue Cover 是用例生成工具，不是 Mock。JaCoCo 是覆盖率库，靠 JVM 的 agent 插桩，不是测试框架。

> **过渡：**名字太多，先从这份清单的出处看起。

【5.42】单元测试工具（2 分钟）

> 维基百科有一份单元测试框架列表。名字多，不用背。课件拿它当入口，不是当默写题。顺着名单，能看清三件事。入口就在那份 List of unit testing frameworks。

>

> 一件事看生态熟不熟。Java 有 JUnit，也有 TestNG。C++ 有 GoogleTest，也有 Catch2。Python 有 unittest，也有 pytest。JavaScript 有 Jest，也有 Mocha。Go 更直接。go test 做进了语言自带的工具链。常用的语言，你往往不用特意去找框架。项目一打开，它就在。

>

> 另一件事是怎么选。别比谁的功能多。看生态。IDE 支不支持。Maven 或 Gradle 能不能一键跑。覆盖率报告出不出。看团队熟不熟。大家 pytest 用顺了，你换成别的，就是加成本。看跟 CI 接得贵不贵。能不能只跑失败的。能不能出标准报告。这三样，比断言的写法好不好看要紧。

>

> 还有一件事在家族。很多名字带 xUnit。JUnit、NUnit、xUnit.net 都算这一家。PHPUnit 也算。源头是 Smalltalk 的 SUnit。共同的零件是这些。测试类，测试方法，断言。前后置处理，测试套件，结果汇总。这套设计过了三十年还在用。因为它抓住的是四步。准备，执行，断言，清理。换一个框架，多半是给这四步换一层皮。

>

> 以后换语言、换项目，先问一句。这个生态里，相当于 JUnit 的是谁。找着了，剩下的用法可以平移。

> **提问：**如果一门语言连一个像样的单元测试框架都没有，你会怎么判断这个技术选型的风险？

> **预设回答：**

- 答"那就别用这门语言"——结论也许对。理由要落到成本：团队得自己搭测试的底子，人会多耗好几倍。

- 答"风险高，因为没法做自动化测试"——对。回归就只能靠人。人一累就漏，漏了就到线上见。

- 追问：选技术栈时，测试生态该和什么放在一起看？——和性能、和社区还活不活，放在同一张表上。

> **易错点 / 考点：**这一页不考名单。考的是框架的共同结构。简答常问 xUnit 系列一般有哪些要素。答测试类和测试方法、断言、前置和后置、测试套件、结果报告。别只答成某一家的 `@Test`。

> **过渡：**清单收一收，只留 Java、C++、Python 里最常听到的。

【5.43】最常用的单元测试工具（3 分钟）

> 同样叫测试工具，JUnit 和 JaCoCo 不是一件事。Java、C++、Python 里常听到的，就是四类。框架。两个能自动生成用例的。Mock。覆盖率。更全的名单在公众号那篇文章里。

>

> 框架这一类，JUnit 和 TestNG 是 Java 的两支。GoogleTest 是 C++ 的标杆。Python 有两条路。unittest 在标准库里，写法靠近 JUnit。pytest 更像日常的 Python。断言直接写 assert。活一样。组织用例，跑完，报结果。

>

> 自动生成，课件点了两个名字。EvoSuite 用遗传算法去搜输入，再吐出 JUnit 用例。Diffblue Cover 是商业产品。主打给存量的 Java 代码补测试。断言是从当前代码的行为反推的。代码错了，错误行为会变成期望值。所以生成完，人必须看断言。特别是业务规则那几条。不看就合并，后来的人会把现状当成对的。

>

> Mock，课件写的是 JMockit。静态方法、构造器、私有方法，它都能 mock。能力强，也和 JVM 版本绑得紧。实验里更多用 Mockito。

>

> 覆盖率按语言分。Java 用 JaCoCo。它靠 JVM 的 agent 插桩，自己不是测试框架。C 和 C++ 用 gcov 采集。lcov 出 HTML。gcovr 做汇总，也能出 XML。Python 用 Coverage.py。原理都是埋点，数哪些行被执行过。

> 实验组合是 JUnit、Mockito、JaCoCo。

>

> 顺手订一个拼写。前面把 SonarLint、SonarQube 写成了 SolarLint、SolarQube。SonarLint 是 IDE 里的静态检查插件。SonarQube 是服务端的代码质量平台。写报告按正确的拼。

> **提问：**Diffblue Cover 给一个老模块自动生成了一千条用例，覆盖率从 20% 冲到 85%，能不能直接合并进主干？

> **预设回答：**

- 答"能，覆盖率涨了就是好事"——风险最大。自动生成的断言固化的是现状，不是需求。现状里有错，错就被写成期望。

- 答"不能，要人工审断言"——对。至少抽查跟业务规则有关的用例，看期望值是不是从需求来的。

- 答"先合并，再慢慢审"——不合适。进了主干，大家会以为这就是正确行为。宁可分批合，也要先审。

> **易错点 / 考点：**最爱考归类。EvoSuite 和 Diffblue Cover 属于用例自动生成，不是 Mock，也不是覆盖率。JMockit 是 Mock 工具，不是框架。gcov 负责采集，lcov 出报告，gcovr 出汇总。三个别混答。

> **过渡：**名字对上了。回到 Java，先看老版本怎么把一次运行串起来。

【5.44】JUnit 4 工作原理（2 分钟）

> 实验课打开的是 JUnit 5。面试题里还在问 `@Before`，还在问 `@RunWith`。所以 JUnit 4 的骨架还得看懂。

>

> JUnit 5 是 2017 年才发布的。存量项目和不少面试题，用的还是 JUnit 4 的说法。你接手老代码，会看到 `org.junit.Test`。不认识它，老项目就读不下去。

>

> 还有一个原因。JUnit 5 是从 4 上拆出来的。你先懂 4 里谁负责什么，才看得懂 5 为什么要拆成三层。

>

> JUnit 4 可以收成这句。用注解标出测试。用运行器去跑。用结果对象把结论收下来。注解有 `@Test`。还有 `@Before` 和 `@After`。类级别是 `@BeforeClass` 和 `@AfterClass`。暂时不跑，用 `@Ignore`。运行器是 Runner，以及它的子类。结果落在 Result 上。RunListener 负责听这些结果。

>

> 一个方法失败，不能把后面的方法一起停掉。上一个方法改过的字段，也不能漏到下一个。所以每个方法都新建一次实例。异常抓住，记下来，再跑下一个。你在 `@Test` 里抛了异常，框架不会崩。它只把这个方法标成错误。你自己写个 main 去循环调用，一个异常往往后面全没了。

>

> 注解名字到了 5 会变。`@Before` 对应 `@BeforeEach`。`@After` 对应 `@AfterEach`。`@BeforeClass` 对应 `@BeforeAll`。`@AfterClass` 对应 `@AfterAll`。`@Ignore` 对应 `@Disabled`。四个类是 TestRunner 和 TestSuite。再加上 TestCase 和 TestResult。

提问：为什么 JUnit 要给每个测试方法都新建一次测试类的实例，而不是所有方法共用一个对象？

预设回答：

- 答"为了隔离，避免上一个方法改了字段，影响到下一个"——对。用例之间不能有看不见的依赖。

- 答"因为 Java 只能这么写"——不是。共用一个实例，技术上做得到。`@FixMethodOrder` 只控制顺序，不表示大家共用一份状态。

- 追问：某个对象很贵，每个方法都要用，怎么办？——放到 `@BeforeClass` 那一层，用静态资源。或者用依赖注入把共享部分管起来。共享要写明，不能靠"上一个方法已经准备好了"。

易错点 / 考点：常考 JUnit 4 和 JUnit 5 的注解对应。`@Before` 对 `@BeforeEach`，`@After` 对 `@AfterEach`，`@BeforeClass` 对 `@BeforeAll`，`@AfterClass` 对 `@AfterAll`，`@Ignore` 对 `@Disabled`。这个对照选择题里常见。

过渡：四个类怎么接成一条线，接着拆。

【5.45】JUnit 工作原理 – 续（3 分钟）

报告上 Failures 和 Errors 分成两列。很多人把它们当成"都是没过"。两列的查法不一样。

>

> 课件把产生结果的关系写成一句。一个 TestRunner 运行一个 TestSuite。这个 TestSuite 可以由一个或多个 TestCase 组成。也可以由别的 TestSuite 组成。运行结果由 TestResult 收集。再由 TestRunner 来报告。标题写着成员三重唱。你要对上的名字，按这句里的四个来。少记一个，收集和报告就会混。

>

> TestCase 是答卷的那个。每个 `@Test` 是一道题。它负责跑，也负责用断言打分。

>

> TestSuite 把卷子收成一场。它能嵌套。套件里面再装套件。能按包、按类、按标签去筛，是因为套件能嵌。只一层的话，按包跑不了。

>

> TestResult 只做收集。每道题的结论分三类。成功，失败，错误。失败叫 failure。断言没成立。程序跑完了，结果和期望不符。错误叫 error。执行中途抛了异常，没走到断言。这可能是被测代码的问题。也可能是测试自己写错了。桩没配对，构造参数给错，都算后一种。所以 Failures 是 2、Errors 是 1，这是两种信号。不能合成"三个都没过"。

>

> TestRunner 把流程启动，再把结果报出来。控制台、IDE 里的树、CI 的报告，都是它在报。收集归 TestResult。报告归 TestRunner。两个活不要答成一个类。

>

> TestSuite 用的是组合。单个用例和一组用例，都实现同一个 Test 接口。所以上层跑项目、跑包、跑类、跑方法，都是同一种调用。JUnit 5 把这个结构留下了。换成 TestEngine 去发现，再执行一棵 TestDescriptor 树。名字换了，这层关系还在。

> **提问：**报告上显示 `Failures: 0, Errors: 3`，你的第一反应应该是什么？这和 `Failures: 3, Errors: 0` 有什么不同的排查路径？

> **预设回答：**

- 答"三个都失败了，去改产品代码"——太粗。Errors 表示没走到断言。先看异常栈。

- 答"Errors 是抛了异常，可能是测试代码自己写错了"——方向对。桩没配对、参数给错、依赖没注入，先怀疑测试环境。

- 答"Failures 才是真失败，Errors 都是环境问题"——不全对。空指针也可能出在被测代码里，那就是缺陷。先看异常类型和栈顶，再判断归谁。

> **易错点 / 考点：**必考 failure 和 error。failure 是断言失败。error 是异常把执行打断。TestSuite 里面还可以再放 TestSuite，这是组合。另一个错答是"TestResult 负责打印报告"。收集是 TestResult。报告是 TestRunner。

> **过渡：**JUnit 4 是一整块。JUnit 5 把它拆开了。

【5.46】JUnit 5 架构（3 分钟）

构建文件里，JUnit 4 和 JUnit 5 的用例会写在一起。少引一个依赖，旧的不会报红。它只是从报告里消失。

>

> 关系就这一句。JUnit 5 等于三块。Platform，Jupiter，再加 Vintage。Vintage 就是复古。不是三个版本。是三个子项目一起发布。细则在 junit.org 的用户指南。

>

> Platform 是基于 JVM 的执行底子，加上 Test-Engine API。它负责发现测试，调度执行，把结果交出去。不关心你用哪一代写法。它带控制台启动器。命令行就能把平台拉起来。Gradle 和 Maven 的插件也接在这儿。所以 `mvn test` 能跑，也能进 CI/CD。它还提供一个基于 JUnit 4 的 Runner。有些旧测试不用改写法就能挂上。

>

> Jupiter 是写测试、写扩展的新模型。编程模型和扩展模型合在一起。`@Test` 从这儿来。`@BeforeEach` 从这儿来。`Assertions.assertEquals` 也从这儿来。`@ParameterizedTest` 同样是。Jupiter 自己提供 Test-Engine。这些测试靠它在平台上跑。你最常碰到的是这一块。

>

> Vintage 也提供 TestEngine。平台因此能跑 JUnit 3，也能跑 JUnit 4。旧用例不可能一夜重写。它是可选

> 依赖。纯新项目可以不引。从 4 迁到 5，先引它，旧的先能跑。再按模块改。改完再摘掉。别停下来大改。

>

> 这个拆法把怎么跑和怎么写分开了。Platform 只认 Test-Engine 这个接口。谁实现它，谁就能上平台。TestNG 能挂。Spock 也能挂。只引 Platform 和 Jupiter，旧用例不是跑失败。是没被发现。用例数对不上，就是没跑。

> **提问：**一个项目同时有 JUnit 4 的老测试和 JUnit 5 的新测试，需要引哪几块？如果只引 Platform 加 Jupiter，会发生什么？

> **预设回答：**

- 答"三块都引"——对。Jupiter 跑新的。Vintage 跑旧的。Platform 负责调度。

- 答"只引 Platform 和 Jupiter 也行，旧用例会被忽略"——现象对。后果要说清。旧用例不是跑失败，是没被发现。覆盖率会无缘无故掉一截。这种静静丢掉最麻烦。

- 追问：怎么确认旧用例真的在跑？——看报告里的用例总数和类名。数量对不上，就是没跑。也可以给 Vintage 单独跑一次。

> **易错点 / 考点：**必考 JUnit 5 的三块和各自的活。三个高频错法：说 JUnit 5 就是 Jupiter，漏了另外两块。把 Vintage 只说成"旧包"，却说不出它提供 TestEngine，让平台跑 JUnit 3 和 4。把 Platform 说成 IDE 插件。口诀是：平台管跑，Jupiter 管写，Vintage 管旧测试还能跑。

> **过渡：**架构有了。每天写代码时碰到的是三个组件。

【5.47】需要了解 JUnit 的关键组件（2.5 分钟）

> 一条测试是绿的。你先别当它验过了什么。打开方法体，里面可能一个断言都没有。

>

> 写代码时会反复碰到三个组件。Assert 类。注解，也叫 Annotation。运行器 Runner。

>

> Assert 类，课件说它非常重要。它用来判断返回的实际结果和预期结果是否相符。主语是结果，不是过程。前面准备数据、调用方法，都是在凑材料。断言才决定过还是不过。没有断言的测试，不是测试。它只是一

> 段能跑通的演示。框架无从知道什么叫错。

>

> 注解以 `@` 开头。用来标明测试类里不同方法是干什么的。JUnit 3 的时候，方法名要以 test 开头才会被执行。有了 `@Test`，方法名就可以写意图。比如 `should_return_zero_when_num2_is_zero`。不用再把名字写成框架喜欢的样子。不用再把名字写成框架喜欢的样子。你不标，框架就找不到它。

>

> Runner 负责运行，也负责把结果显示出来。课件的说法是：通过运行器来运行，并显示执行结果。JUnit 5 里，这件事改由 Platform 的 Test-Engine 来做。概念没变。总有一个东西在发现测试，再把它们驱动起来。没有运行器，注解标了也跑不起来。

>

> 三个连成一圈。注解告诉框架有哪些测试。Runner 把它们跑起来。Assert 决定哪条通过。排查"测试没生效"，也按这个顺序。注解加了没有，是不是被标成禁用。运行器扫到没有，看包路径和命名过滤。最后看断言写了没有。三条对一下，才知道这片绿验过什么。

> **提问：**一个测试方法加了 `@Test`、能跑、报告是绿的，但它其实什么都没验证。你怎么用一分钟发现这件事？

> **预设回答：**

- 答"看代码里有没有 assert"——最快。

- 答"故意把期望值改错，看它红不红"——更稳。还是绿，就说明这条是空壳。这是变异测试的思路。

- 答"看覆盖率"——不够。空壳也能把行覆盖跑满。覆盖高，不代表断言抓得住错。

> **易错点 / 考点：**常考三个组件各自干什么。Assert 判断实际和预期是否相符。Annotation 以 `@` 引导。Runner 负责运行并显示结果。别把 Assert 说成"抓异常的工具"。抓异常是运行器的事。Assert 只做判断。

> **过渡：**三个里，注解写得最多，也最容易写错。

【5.48】注释 Annotation（2.5 分钟）

> `@` 后面那一长串，第一次看会眼晕。它们不是单词表。课件说，注解把测试程序的逻辑结构和功能写清楚。

> 它标明方法是什么角色，也标明什么时候跑。

>

> 跟用例有关的，`@Test` 是基础。`@ParameterizedTest` 让一条用例跑多组数据。不这样，同一段断言得复制很多遍。改期望值就容易漏。它还得配数据来源。来源可以是 `@ValueSource`。也可以是 `@CsvSource`，或 `@MethodSource`。只写参数化注解，跑不起来。同组还有 `@RepeatedTest`。还有 `@TestFactory`。还有 `@TestTemplate`。

>

> 生命周期是四个。`@BeforeEach` 和 `@AfterEach`，每个方法前后各一次。`@BeforeAll` 和 `@AfterAll`，整个类只一次。后两个默认要写成静态方法。因为每个方法默认一个新实例。静态方法才比实例活得长。要用实例方法，加 `@TestInstance`。生命周期设成 PER_CLASS。放成整个类一次，用例会互相改数据。那种红很难查。

>

> 组织和展示是这些。`@DisplayName` 起一句人话。`@Nested` 做嵌套分组。`@Tag` 打标签。CI 可以把标成 slow 的排除掉。`@TestClassOrder` 管类的顺序。`@TestMethodOrder` 管方法的顺序。顺序不是默认就有的。靠上一条喂下一条，换个环境就可能红。

>

> 控制和扩展还有一组。`@Disabled` 先禁用。`@Timeout` 一超时就失败。卡住的用例才不会拖死整次运行。`@ExtendWith` 和 `@RegisterExtension` 都是注册扩展，挂法不同。`@TempDir` 建临时目录，测完再清。写文件的测试用得上。全文在 junit.org 用户指南的 writing-tests。课件把 BeforeEach 的大小写印拧了。代码里按驼峰写。

>

> 禁用一大片，超时一个没有，展示名也不写。报告别人就读不下去。

> **提问：**`@BeforeEach` 和 `@BeforeAll` 各适合放什么？如果一个方法标了 `@BeforeAll`，但方法不是 static，会发生什么？

> **预设回答：**

- 答"`@BeforeEach` 放每条用例都要的初始化，`@BeforeAll` 放贵的共享资源"——对。连接池、容器启动

放后面那个。测试数据放前面那个。

- 答"不是 static 会报错"——对。默认每个方法一个新实例。静态方法才活得比实例长。除非加上 `@TestInstance`，把生命周期改成 PER_CLASS。

- 追问：为什么不图省事，全放进 `@BeforeAll`？——共享状态会让用例互相踩。第 3 条失败，可能是第 1 条改了数据。这种红很磨人。

> **易错点 / 考点：**两组要分开。一组是每个方法一次的 `@BeforeEach`，和整个类一次的 `@BeforeAll`。另一组是 `@ParameterizedTest` 必须配数据来源，`@ValueSource`、`@CsvSource`、`@MethodSource` 至少有一个。JUnit 4 到 5 的注解对照，前面那一页已经对齐过。

> **【思政落点 8】**：注解看起来只是几个 `@`。它要求的是把角色写明白。`@Test` 让框架找得到。`@DisplayName` 让同事读得懂。`@Timeout` 让一条卡住的用例不拖垮整条流水线。这些不难写。难的是别人接手时还要猜。少让人猜一次，就是写代码时的默认做法。

> **过渡：**注解管怎么组织、怎么跑。过不过，要看断言。

【5.49】Assert 类（2.5 分钟）

> `assertEquals` 的两个参数写反，编译器不会拦你。通过的时候两边相等，你看不出问题。一失败，报告就反着说。

>

> 课件有两句要吃进去。检查并判断程序的运行结果，是软件测试里的一项重要工作。测试程序里一定会有断言条件。运行时条件不满足，就说明状态或结果和期望不一致。程序里有缺陷。你不写断言，框架就没有"什么叫错"的出口。

>

> 机制是这样。JUnit 用 Assert 类提供一组断言方法。这个类在 `junit.framework` 包里。参数里有期望变量 expected。也有实际变量 actual。两个值相等，结果和期望相符，用例通过。不相等，用例失败。

>

> 包名要分口径。`junit.framework` 是 JUnit 3 和 4 的路径。JUnit 5 写成 `org.junit.jupiter.api.Assertions`。答题按教材写前一个包。写代码用 Assertions。两个都对，别混着答。

>

> 参数顺序要注意。`assertEquals` 第一个是期望，第二个是实际。写成先实际、后期望，失败报告会把方向说反。排查就跑偏。先写你期望的。再写跑出来的。

>

> 常用的还有几组。`assertTrue` 和 `assertFalse` 判真假。`assertNull` 和 `assertNotNull` 判空。`assertThrows` 用来检查这段应该抛某个异常。异常路径也能被断言。JUnit 5 的 `assertAll` 把多条断言打成一包。失败可以一次看全。

>

> `assertNotNull` 只说明有东西返回。`assertEquals` 才说明返回的就是那个值。覆盖靠的是执行到了。质量靠的是断言写得具体。

> **提问：** `assertEquals(3, result)` 和 `assertEquals(result, 3)` 都能编译通过，为什么后者是个必须避开的习惯？

> **预设回答：**

- 答"没什么差别，相等就行"——只在通过时没差别。失败时报告会反着说，人会判错方向。

- 答"失败信息会颠倒，排查方向错"——对。这是最实在的害处。

- 答"JUnit 5 可以再传一句说明，写清楚就行"——说明是辅助。参数顺序是约定。不能靠那句说明来兜。

> **易错点 / 考点：** 三个点。`assertEquals` 先期望、后实际。Assert 的包，教材口径是 `junit.framework`，JUnit 5 是 `org.junit.jupiter.api.Assertions`。相等就通过，不等就失败。判断题常出"断言失败就说明被测程序一定有缺陷"。还要排除测试代码自己写错。这句在考试里通常判错。

> **过渡：** 断言不是孤零零一个类。TestCase 把它继承下来了。

【5.50】TestCase、TestSuite 和 TestResult 类（2.5 分钟）

> 测试类里能直接写 `assertEquals`，不用再带类名。这个方便是继承来的。

>

> TestCase 是给测试用例提供服务的核心类。它同样在 `junit.framework` 包里。它继承 Assert。所以断言方法可以直接用。你写的测试类，都要直接继承它，或者间接继承它。靠它提供的方法来跑 `@Test`。也靠它来判断。JUnit 3 就是这么用的。必须继承。方法名还要以 test 开头。

>

> 继承 Assert，好处是随手能断言。代价也在。测试类被绑在框架的继承链上。JUnit 4 改成注解，不再强制继承。JUnit 5 改成静态导入 Assertions。组合还是测试类加上断言库。只是不再占用唯一的父类。教材这一页按继承来讲。题目要看说的是哪一版。

>

> TestSuite 实现了 JUnit 的 Test 接口。用来管理每一个测试用例。关键词是实现了 Test 接口。单个用例和套件是同一种类型，套件才能再装套件。你才能一次跑一个方法，一个类，一个包，或者整个项目。接口不统一，嵌套就做不到。

>

> TestResult 收集并记录所有用例的运行结果。课件分成三类。成功，失败，错误。看报告先看错误。错误常常表示这条用例没验到东西。

>

> 三个类接成一条链。TestSuite 把 TestCase 组织起来。TestCase 跑的时候，把结论写进 TestResult。TestRunner 再读 TestResult，去报告。组织，执行，记录，呈现。四个动作，别答串。
> JUnit 5 里这三个类已经很少直接用了。分别被注解、引擎的发现机制、TestExecutionSummary 接过去。结构还在。所以题还在考。

提问： `TestCase` 继承 `Assert`，这个设计有什么好处，又有什么代价？

预设回答：

- 答"好处是随手能用，不用再单独准备断言类"——对。这是方便。

- 答"代价是测试类被绑死在框架的父类上"——对。所以后来改成注解，再改成静态导入。测试类不该被父类占住。

- 追问：从 JUnit 3 到 JUnit 5，主线是什么？——从继承框架，到用注解标记，再到用组合做扩展。一路都在

松开耦合。

> **易错点 / 考点：**三句定义要能说全。特别是 TestCase 继承 Assert，最容易漏。TestResult 负责收集，TestRunner 负责报告。TestSuite 实现了 Test 接口。没有这个接口，套件就不能再装套件。

> **过渡：**类的关系对齐了。看一段 JUnit 5 的脚本怎么把它们用上。

【5.51】单元测试脚本（以 Junit 5 为例）（2 分钟）

> 十条测试红了三条。你看方法名，分不清是数据错了，调用错了，还是结果真错了。脚本如果没骨架，就只能逐行读。

>

> 以 JUnit 5 为例，一段完整脚本分四层。类这一层，通常跟被测类同名，再加 Test。类上可以用 `@DisplayName` 写一句人话。报告里就不是一串看不懂的方法名。

>

> 准备放在 `@BeforeEach`。在这里造数据，建对象。每条用例都从同一份干净状态出发。你要是发现某条还得接着上一条的数据才能跑，那不是测试。那是连续剧。上一条一改，后面跟着红。红的原因还对不上。

>

> 用例本体是三段。业内叫 AAA。Arrange 是准备。前面 new 对象，设参数。Act 是执行。中间那一次调用被测方法。Assert 是断言。后面的 `assertEquals`。为什么这个顺序不要打乱？挂了的时候，你要很快判断是数据，是调用，还是结果。三段分开，一眼能定位。混在一起，就得重读整段。

>

> 清理放在 `@AfterEach`。释放资源，删临时文件。更省事的是 `@TempDir`。框架建目录，测试结束再清。你自己忘了删，下一条可能读到上一条留下的文件。

>

> 再加两条写法。一个测试方法只围着一件事。又测加法又测减法，红了你说不清哪件坏了。方法名就当文档。`should_return_zero_when_second_arg_is_zero`。它比 `testAdd2` 清楚。配上 `@DisplayName`，报告可以直接拿去对验收。注解管结构。AAA 管顺序。断言管判定。名字管能不能读。

> **提问：**一条 `@Test` 方法里，Arrange 写了 30 行，Act 1 行，Assert 1 行。这条测试有什么问题？

> **预设回答：**

- 答"太长了，不好读"——这是表面。问题在准备太长，被测对象的依赖太多。这会把你推回去想设计。

- 答"重复的构造应该抽到 `@BeforeEach`，或抽成工厂方法"——对。

- 答"没问题，断言对就行"——眼前能跑。改一处构造就要动几十行，以后没人敢动。

> **易错点 / 考点：**实验和卷面都常考 AAA。准备数据，调用被测方法，断言结果。三句话要说全。别把 `@BeforeEach` 记成"每个类一次"，那是 `@BeforeAll`。JUnit 5 里 `@BeforeAll` 标的方法，默认必须是 static。

> **过渡：**单个类还好办。被测代码一调外部服务，就得换办法。

【5.52】Mock 技术（2 分钟）

> 被测方法要调支付网关。网关不在你手里。调用还可能按次花钱。数据库里没有你要的订单。风控在灰度。消息队列也没起来。

>

> 这种业务方法很常见。查库拿订单。调风控，看能不能放行。调支付扣款。再发消息通知仓库。四个下游都在别人手里。路有三条。搭一整套环境，贵，慢，还容易因为别人的服务抖动而红。干脆不测，缺陷就欠着。换成替身，把下游换成你能控制的对象。Mock 技术走的是第三条。

>

> 手写桩是最原始的替身。后面要分的词有六个。Fake，Dummy，Double，Spy，Stub，Mock。先给一张地图，听的时候别混。Dummy 用来把参数位填上。Stub 用来给一个你指定的返回值。Spy 把调用记下来。谁调的，几次，参数是什么。Fake 能真的跑起来，只是实现更简单。Mock 既给返回值，又让你去核对交互。

>

> Double 这个词要单独放。Test Double 是总称。上面那五个都算它的一种。看到 Double，别再把它当成第六个新品。

>

> 为什么要花时间分？选错了，成本和结果都会偏。一个合适的 Fake，能让很多用例又快，又接近真的行为。滥用 Mock，测试就变成核对你自己怎么调用。一重构，调用方式变了，测试全红。业务却没变。这种一改就碎的测试，后面会反复碰到。所以这一节的难点不是背六个定义。是知道这一次用哪一个。

> **提问：**如果被测方法要调一个第三方短信接口，你会搭一个真的测试环境，还是用替身？说说你的取舍依据。

> **预设回答：**

- 答"用替身。短信发不出去，还可能花钱"——对。有副作用的依赖，要隔离开。

- 答"搭真环境更真实"——真实，代价也高。运营商限流、延迟抖动，会让测试时红时绿。这种结果不好重复。

- 追问：真实性怎么补？——分开层。单元测试用替身，把逻辑钉住。集成测试再用真环境，看接线对不对。别让单元测试去扛集成测试的活。

> **易错点 / 考点：**六个词的定义和差别，特别是 Stub 和 Mock。只提供返回值、不去核对"被调用了几次"的，是 Stub。会核对次数、顺序、参数的，是 Mock。

> **过渡：**六个词里，先把 Fake、Dummy、Double、Spy 钉住。

【5.53】Fake、Dummy、Double 等（3 分钟）

> Fake、Dummy、Double、Spy，四个词常被当成近义词。课件给的定义，差得很远。

>

> Fake 是伪对象。也叫伪造，叫假货。反义词是 real。课件说，它几乎和产品代码做同样的工作。伪装成跟真的一样。但为了测试，实现更简单。关键词是更简单的实现。不是返回一句假数据。内存数据库代替真库。内存队列代替消息中间件。行为要对，只是更轻。它是精简版，不是在骗人。

>

> Dummy 是哑对象。课件用了仿制品、哑巴、傀儡这几个说法。它是对被测对象的一种替换。特别是缺乏原有的特性或功能，用来做欺骗性的替换。课件还有一句。完全听命行事。不在具体测试里面起任何作用。方法签名要求传一个 Logger，这条用例却不写日志。你传一个什么都不做的空实现。那个 Logger 就是 Dummy。它只是把参数位填上。

>

> Double 是测试替身。课件的定义是：有相同的行为和响应。它是总称。
>
> Fake、Dummy、Stub、Spy、Mock，都算它的一种。看到 Double，别把它当成某一个具体类。

>

> Spy 是测试间谍。课件说，它能够接收参数，并返回值。再补一句。它既按你设定的值返回。又把调用记下来。谁调的，几次，参数是什么。事后供你核对。它适合看副作用。提交订单之后，审计接口应被调用一次。参数里要带用户标识。只看返回值，这个副作用看不见。

>

> 归类这句要能说出来。Dummy Object 和 Test Spy，可以归为 Stub。它们提供输入，或把输出记下来。并不强制你去 verify。Mock 不一样。它就是等着你核对交互。选用时从轻到重。能不替就不替。要返回值，用 Stub。要看有没有被调用，用 Spy。只有"没按约定交互就算失败"，才用 Mock。

> **提问：**一个方法签名要求传 `UserRepository`，但这条用例根本不查库。你传什么？这算 Fake 还是 Dummy?

> **预设回答：**

- 答"传一个什么都不做的空实现"——对。这是 Dummy。

- 答"这是 Fake"——不对。Fake 里面还有替代性的真计算。空实现不算。判据是：里面有没有在替原对象干活。

- 追问：这个空实现要是被误调用了，返回空，下游空指针，怎么办？——让不该被调用的方法直接抛异常。这条用例就变成了"它不该被叫到"。

> **易错点 / 考点：**四词定义，加上"Dummy 和 Test Spy 可归为 Stub"。三个错法：把 Fake 说成假数据，它是简化过的对象。把 Dummy 说成"返回默认值的桩"，它不该提供有意义的值，也不该被当成被测行为。把 Double 当成某一个具体替身，它是总称。

> **过渡：**四个词里，Fake 最容易听成"返回一句假数据"。

【5.54】伪对象（2.5 分钟）

`save` 用桩返回 true。下一步 `find` 却什么都查不到。订单根本没存进去。你再写一个桩，让 `find` 返回

一条假订单。测下来，你对的是桩的剧本。不是仓储的行为。

>

> 课件第一句就是这个场合。在某些情况下，插桩不够用。被桩替换掉的行为，正是被测对象所需要的。这时候，伪对象可能是一种合理的折衷。伪对象是协作者的轻量级实现。

>

> 什么叫"换掉的行为，被测对象还需要"？被测代码不是拿一个返回值就结束。它需要下游真的做出某件事。下单要先保存，再查出来，再开发票。Stub 只能逐个调用编返回值。编到后面，前后已经对不上。

>

> Fake 就补在这里。做一个内存仓储。`save` 真的放进一个 Map。`find` 真的从 Map 里取。行为是真的。只是没写到磁盘。比桩真。比真库轻。折衷就是这个意思。

>

> 课件的 `MakePurchase` 就是一串有状态的协作。先 `CreatePurchase`，传入 customer。再 `AddProduct`，把 product 加到这个 purchase 上。再 `CreateInvoice`。discount 不是 null，就 `ApplyDiscount`。最后返回 invoice。后一步依赖前一步的结果。你要是给每一步都配一个互不相干的桩，折扣算没算对，你自己都不信。

>

> 什么时候从 Stub 换成 Fake，看三条。被测代码先写再读。它依赖下游的状态变化，比如从未支付到已支付。同一批测试要反复用同一个协作者。反过来，Fake 得自己写一套简化逻辑。这套逻辑自己也会有错。内存库和真库的 SQL，也不一定一样。只用在复用多、又值得的协作者上。一次性的场景，写 Stub 就够。

提问：用内存数据库代替真库，会不会掩盖"SQL 写错了"这类缺陷？如果会，你怎么补？

> **预设回答：**

- 答"会。内存库和真库的 SQL 方言不一样"——对。一边能跑的语句，另一边未必能跑。

- 答"所以还要有集成测试跑真库"——对。单元测试用 Fake 保逻辑。集成测试用真库保 SQL。

- 追问：那 Fake 还值不值？——值。它让大多数用例变得很快，人愿意反复跑。剩下那一小部分，交给真库兜。

易错点 / 考点：常考 Fake 和 Stub 的差别。Stub 只提供预设的返回值。Fake 提供简化、但真实的行为。用

在被测对象依赖协作者"怎么做"、不只是"返回什么"的时候。别把 Fake 说成造出来的测试数据。那是夹具，不是 Fake。

过渡：概念有了。图上把 SUT 和 DOC 的位置画出来了。

【5.55】Fake 示意图（2 分钟）

图左边标着 Public interface。右边的缩写是 DOC。这两个位置先认清。Fake 站错边，整张图就反了。

>

SUT 是 System Under Test。被测系统。DOC 是 Depend-on Component。被测系统所依赖的组件。单元测试的基本格局，就是一个 SUT，旁边若干 DOC。桩、Fake、Mock，都是在替换 DOC。不是去替换 SUT。把 DOC 答成被测对象，整题就反了。

>

前门，图上写的是 front door。也就是 Public interface。旁边还有四个字，直接输入。意思是测试从公开接口走进去。直接调用它对外的方法。不要绕过接口去改内部字段。从公开接口进去，测的才是别人真实的用法。一旦用反射改私有字段，测试就和实现绑死。实现一改，测试就红。那种测试以后会变成负担。

>

私有方法改成 public 再测，别人会依赖它，你就改不动了。

>

Fake 站的位置是 DOC 那一格。接口一样。行为简化。它替换的是整个协作者。不是某一次调用的返回值。Stub 管的是"这一次返回什么"。Fake 管的是"这个协作者整体上怎么做事"。位置画错，你会用桩的要求去套 Fake。

>

图上还有 round-trip test。往返测试。一次测试里既写进去，又读出来，再比对。看数据在一来一回里有没有丢，有没有错。保存订单，重新查询，断言字段一致。就是这种。Fake 自己有存储，往返很自然。用 Stub 做往返，存进去的值是你编的，读出来的也是你编的。

>

> 箭头里还有一个前提。SUT 只依赖接口，不依赖具体实现。被测代码如果直接 new 一个数据库连接，你连把 DOC 换掉的机会都没有。测试写不下去，先看依赖是不是从外面传进来的。

提问：为什么强调从公开接口进入？如果为了测试方便，把某个私有方法改成 public 再测，有什么问题？

> **预设回答：**

- 答"没什么问题，能测就行"——眼前方便。测试会和实现绑在一起。一重构就红。

- 答"会把实现细节变成别人可以依赖的约定"——对。public 一开，你就收不回来了。

- 追问：私有方法里的复杂逻辑怎么测？——从公开接口把分支覆盖到。盖不住，就抽成独立的类再测。测试在逼你把设计拆开。

> **易错点 / 考点：**四个词要能对上中文。SUT 是被测系统。DOC 是被依赖的组件。front door 是公开接口，也就是直接输入的那扇门。round-trip test 是写进去、读出来、再比对。不是"把测试跑两遍"。DOC 不是被测对象。

> **过渡：**图看完，看一个自己写的仓储替身长什么样。

【5.56】Fake 示例 1（2 分钟）

> 只用一个 Map，也能把仓储的存和取做完。真库那一套，测试里先别连。

>

> 被测代码需要一个仓储。负责存，也负责取。真实实现连着数据库。测试不想连。于是写一个 FakeRepository。里面用 Map，或者用 List。对外仍是同一套接口。`save` 往里放。`findById` 按标识取。`count` 返回里面有多少条。能不能用，取决于三件事。

>

> 第一，接口要和真实现一致。稳妥的做法，是让 Fake 和真实现实现同一个接口。替换的时候不用改被测代码。编译期就能保证方法签名对得上。各写各的方法名，替换时就得改产品代码。那就不叫替换得动。

>

> 第二，行为要够真。`save` 之后，`findById` 必须查得到。这就是上一页的往返。同一个标识再 `save` 一次，应该覆盖，不是再追加一条。`count` 要跟着变。这些不是额外功能。被测代码依赖的就是这些语

> 义。少做一条，测试会给出错误的绿灯。你的断言如果只写"能查到"，一条被存成两条，你也发现不了。

>

> 第三，别实现这条逻辑用不到的东西。事务、缓存、分页，真库有。被测代码用不到。写进去，只是给 Fake 自己增加出错的机会。目标是语义够用。不是功能齐全。

>

> Fake 会被很多用例复用。它有错，全项目的测试都在给错误信号。所以把它放在测试源码里单独的 fakes 包。有名字，有说明。需要的话，给 Fake 自己也写测试。它是测试资产。不是随手塞在某个测试方法里的临时类。

> **提问：**如果 Fake 的 `save` 忘了做"同 ID 覆盖"，而是在 List 里又加了一条，哪类测试会假绿?

> **预设回答：**

- 答"测新增的用例"——不一定。更容易假绿的是测更新和测数量的用例。更新之后数量应该还是 1。Fake 却返回 2。你的断言如果只写"能查到"，就发现不了。

- 答"依赖这些语义的用例，都会给出错误信号"——方向对。不是每一条都会红，但该红的可能还绿着。

- 追问：Fake 要不要自己也写测试？——要。替身自己错了，上面一大片用例都不可信。

> **易错点 / 考点：**不考某一行代码。考 Fake 和真实现的关系。同一个接口。语义一致。实现更简单。常见的错是写成一个名字像、方法却对不上的类。替换时还得改被测代码，那就没做到可替换。

> **过渡：**自己用 Map 是一种。也有人直接拿内存数据库当替身。

【5.57】Fake 示例 2（2.5 分钟）

> 有人把伪对象写成一个等式。fake object 等于 enhanced stub。伪对象就是增强了的桩。

>

> 增强在哪里？桩只负责返回一个值。伪对象要表现出一种行为。目的仍是测试。不是拿去当生产实现。等式用来记关系。别记成"Fake 就是一份假数据"。

>

> 这一页的场景反过来了。前面的 Fake 是配角，帮别的类被测。这里要验证的是 DAO 本身。DAO 是 Data Access Object。它把对象翻译成 SQL，再翻译回来。要验它能正常工作，就构造 fake object 代替数据库。每次连真库太慢，还要清数据。不代替，这条测试就很难反复跑。

>

> 课件点了两个具体东西。HSQLDB 是流行的内存数据库。英文是 in-memory database。DBUnit 用来准备测试数据。两个合在一起，是一套常见的 DAO 测试做法。HSQLDB 在进程里把库拉起来。随测试创建，随测试结束。DBUnit 在每条用例前，按 XML 或 CSV 把数据灌成一份初始状态。测完再重建。你跑的是真 SQL，真 JDBC，真的类型映射。只是库在内存里。

>

> 这已经不是最纯的单元测试。有人叫它集成度更高的单元测试。也有人叫轻量集成测试。它比全用替身更真。因为 SQL 真的执行了。它又比连上网络、拉起容器更快。DAO 这一层如果把 JDBC 全 mock 掉，你只能证明调用发生过。SQL 对不对，一点没验到。

>

> 还有一个差别别放过。内存库和真库的 SQL 方言不完全一样。HSQLDB 有兼容模式。函数、索引、锁的行为仍会有差别。所以它替代不了上线前在真库上的那一轮。逻辑用内存库兜。SQL 落到真库再确认一次。

> **提问：**测一个 DAO，用 Mockito 把 `JdbcTemplate` mock 掉，和用 HSQLDB 真跑，两种做法各能发现什么、各漏什么？

> **预设回答：**

- 答"Mock 能核对调用参数。内存库能核对 SQL 的结果"——对。Mock 看你传入的 SQL 和参数。内存库看这段 SQL 是不是真的返回了那些行。

- 答"Mock 更快，内存库更接近真库"——对。这就是取舍。

- 追问：到底用哪个？——看你要防什么。防参数传错，用 Mock。防 SQL 写错、映射错，得真跑。DAO 这一层的问题，多半在 SQL 和映射上。所以内存库更值。

> **易错点 / 考点：**Fake 的典型做法，是内存数据库、内存队列、内存文件系统。真实行为，简化实现。别把 DBUnit 说成数据库。它是准备测试数据的工具。别把 HSQLDB 说成 Mock 框架。它是真的数据库，只是驻在内存里。

> **过渡：**负责给出状态的几个替身，到这儿见过了。

【5.58】模拟对象（2.5 分钟）

下单方法返回的订单对象完全正确。风控报备却没发出去。你盯着返回值，这个错看不见。

>

> 课件对 Mock 用了很重的说法。它把 Mock 叫做游戏的改变者。因为测试的焦点从状态转到了行为。Fake 这类看状态。测试以断言结束。断言核对返回值，或去查对象的属性。你关心的是结果对不对。多数测试默认就是这样。

>

> 基于行为的测试不一样。它核对模拟对象和被测代码之间的交互。也可以是和另一个协作者的交互。关心有没有被调用，调用了几次，有没有去查 Value 属性。结果不在返回值里，状态验证就无从下手。

>

> 报备就是这种。方法返回订单。返回值上看不出报备发过没有。行为验证可以写成核对。风控那个客户端的 report，必须发生一次。这就是 Mock 的用场。返回值看得见的计算，不必上 Mock。

>

> 课件那段代码就是这个套路。测试方法叫 `useLenientMock`。先 new 一个 LenientMock。变量名是 campaignMock。再 new 一个 PurchaseWorkflow。把这个 mock 传进去。然后链式调用。`addItem`，书名用 `getBookByTitle` 取。标题是 Developer Testing。`usingExistingCustomer`，客户号是 1234567。`enterDiscountCode`，折扣码是 DEAL。最后是 `campaignMock.verify()`。这里没有 `assertEquals`。过不过，看交互有没有发生。

>

> 行为验证更强，也更脆。你把"怎么调"写进了测试。方法改个名，或者一次调用拆成两次。业务结果没变，测试也会红。所以先做状态验证。只有结果观察不到的时候，才改用行为验证。

提问：一个方法"发送一封欢迎邮件"，返回值是 void。你要怎么验证它真的发了？用状态验证还是行为验证？

预设回答：

- 答"返回值是 void，没法验"——void 正是行为验证的典型场合。

- 答"用行为验证。核对邮件服务被调用一次，收件人是新用户"——对。参数也要核。发错了人，不能算通过。

- 答"我改成让它返回发送结果，再断言"——可以想。先问这个返回值业务上要不要。业务不需要，就别为了测试去改产品接口。业务本来就要这个结果，比如用来重试，那改得对。

> **易错点 / 考点：**状态验证看最终状态或返回值。行为验证看和协作者的交互：有没有调用，几次，什么顺序，参数是什么。别把"行为验证"理解成"把功能测一遍"。这里的行为，指的是交互。

> **过渡：**状态和行为分开了。再把 Mock 和 Stub 并排放。

【5.59】Mock vs. Stub（2 分钟）

同一次重构，Stub 写的测试还是绿的。Mock 写的测试红了。业务结果一点没变。

>

> 红的那条并没有发现缺陷。它发现的是调用方式变了。Mock 和 Stub 的差别，就卡在这种地方。并排看的时候，抓住三条线。

>

> 目的不一样。Stub 是为了让测试跑起来。被测代码要一个值，它给一个值。事情就完了。Mock 是为了让测试做判断。它待在那儿，就是等你 verify。Stub 不关心你最后核什么。Mock 在，就是为了被核对。

>

> 验证的对象不一样。Stub 对着状态。给什么输入，返回什么。断言落在被测对象的返回值上。Mock 对着行为。这个协作者被怎么对待了。调了几次。参数是什么。顺序对不对。同一段代码，Stub 只能问结果对不对。Mock 才能问过程守不守你写下来的约定。

>

> 严不严也不一样。Mock 默认较严。你声明了期望，没满足就失败。哪怕业务结果是对的。这就是过度规范。英文常叫 over-specification。Stub 宽松。你调不调，它都在。它不因此失败。所以 Mock 把实现细节写进测试之后，无害的重构也会红。那条测试没在保护行为。它只是保护了旧的调用方式。

>

> 读代码还能看机制。Mock 多半是框架在运行时造出来的代理。临时实现一个接口。Stub 多半是你手写的类。看到 when，再看到 thenReturn，那是框架造的 Mock。看到一个类实现某个接口、自己把方法填上，那是手写的 Stub。分开的办法是问一句。失败是因为结果错了，还是因为调用方式变了。前者用 Stub 就能表达。后者才是 Mock 在报警。无害的改动把测试打红，这条 Mock 就要收。

> **提问：**重构时把一次 `save()` 拆成 `saveDraft()` 和 `confirm()` 两步，业务结果完全一样。用 Stub 写的测试和用 Mock 写的测试，分别会怎样？

> **预设回答：**

- 答"Stub 的还是绿的，Mock 的红了"——对。一次把两条线说清了。

- 答"两个都会红"——不对。Stub 不关心你调了几次。

- 追问：这说明什么？——Mock 的成本要算进重构成本里。只对业务上必须发生的交互做 verify。别对实现细节做 verify。

> **易错点 / 考点：**答题可以按这三句。Stub 提供预设返回值，用于状态验证。Mock 核对交互，次数、参数、顺序，用于行为验证。Mock 更严。用多了，测试会变脆。别说"Mock 是更高级的 Stub"。不是更高一级。是关心的东西不同。

> **过渡：**课件把这事收成了四句，可以直接拿去答题。

【5.60】Stub 和 mock 小结（3 分钟）

> Stub 已经够用的地方，还是有人上手就 Mock。每次再写两三条 verify。下次一重构，全红。

>

> 课件这页四句，一句比一句要紧。第一句。Stub 最弱。它只是参与测试，为了让测试跑起来。它提供输入。不参与判断。弱不是缺点。你改了实现，它不跟着失败。

>

> 第二句。Dummy 和 Spy 强一些。Test Spy 参与测试，还要验证产生的某种结果。Dummy 是简单、原始的测试替身。例子是 Dummy server。你给它不同的输出，它就有不同的响应。内部没有实质计算。它是从接口或基类派生出来的。这就是和 Fake 的分界。Fake 有简化过的计算。Dummy 没有。前面把 Spy 归进

> Stub，是因为都不强制核对交互。这儿按强弱排，Spy 还多看一项结果。两句不打架。

>

> 第三句。Mock 更强。它要求你规定它如何参与。又比较灵活，看情况定。它的行为有时像 Dummy。有时像 Stub。有时像 Spy。配一配，它可以扮演这几个角色。扮演得了，不代表每次都要上最强的。

>

> 第四句最要紧。够用就行。Stub 够用，就不要用 Mock。用 Mock 就要写 verify，还要约定次数和参数。这些都是实现细节。实现一变，测试就红。红多了，人就不信，开始标 `@Disabled`。禁用一多，测试还在，保护没了。

>

> Stub 只承诺给一个值。重构时不容易误伤。能用 Stub 说清楚，就别升成 Mock。要用 Mock，通常是这三种里的一种。结果观察不到。比如返回 void，发消息，写日志。交互本身就是业务要求。比如报备只能一次，多了就是事故。或者你要核对顺序和次数。先扣库存，再扣款。顺序错了，会出资金损失。这三种之外，用 Stub。

> **提问：**团队里有个同事特别喜欢用 Mock，每个测试都写三条 `verify`。你会怎么说服他改？请给一个基于事实的论证。

> **预设回答：**

- 答"Mock 更慢"——说不服人。这点时间差，日常可以忽略。

- 答"下次重构你就知道了"——这是情绪，不是论证。

- 答"拿一条用例做个小实验"——对。把一次调用拆成两次，业务结果不变。Mock 那条红了。换成 Stub，还是绿的。拿这个结果跟他说。

- 收口：你说 Mock 好，或说它不好，都要有一次能重复的对比。

> **易错点 / 考点：**四句要能说全。Stub 最弱。Dummy 和 Spy 强一些。Mock 更强，也能灵活扮演。够用就行，Stub 够用就不要用 Mock。判断题常出"Mock 比 Stub 好，应该优先使用"。错。标准答案就是课件那句。另一个点：Dummy 内部没有实质计算。

> **【思政落点 9】**：这句"够用就行"，拦的是一种习惯。工具越重，越像有水平。先问要解决的是什么。再问最轻的办法是哪一种。Stub 能把事情说清楚，就不要为了 verify 把调用细节写进测试。别给问题加上它不需要

的复杂度。

> **过渡：**替身选哪一种，想清楚了。剩下是手写，还是交给框架。

【5.61】Mock 框架（2 分钟）

> 四个名字常被排成一排：JMock、EasyMock、Mockito，再加上 Moco。括号里写着 Java API、独立服务器。前三个和第四个，不是同一类东西。

>

> 前三个是 Java 的 Mock 框架。它们在测试代码里造替身。活动范围是同一个 JVM。给接口生成代理，控制返回值，把调用记下来。判据很具体：这个协作者能不能在测试方法里 new 出来。能，就用它们。

>

> Moco 被标成独立服务器。它自己起一个 HTTP 服务。你给一份配置，它监听端口，按规则回响应。被测代码要走网络才碰得到它。跨进程的 HTTP 依赖，用 Moco。WireMock、MockServer 也是这一类服务桩。

>

> 三个 Java 框架还有一条先后。JMock，然后 EasyMock，然后 Mockito。一代比一代轻。JMock 用约束对象写期望。oneOf、allowing 读着像英语，写起来啰嗦。EasyMock 分成录制、回放、检查。得先声明，再放行。Mockito 收成 when 和 verify。可以先跑，再查有没有被调用。新项目直接用 Mockito。前两个主要用来读老代码，也用来看清 Mockito 为什么更顺手。

>

> 所以先分清场合。同进程里的协作者，用 Mockito。要走网络的 HTTP 依赖，用 Moco 这种服务桩。

> **提问：**被测代码要调一个第三方支付接口，走的是 HTTP。你用 Mockito 还是 Moco？Mockito 在这里为什么不够？

> **预设回答：**

- 答"用 Moco，因为要发 HTTP"——对。跨进程就得用服务桩。

- 答"把 HTTP 客户端封成接口，再 mock 掉"——常见，也只对了一半。逻辑层可以这么隔离。序列化、URL、

超时、重试，这些容易错的地方你测不到。

- 追问：那怎么补？——逻辑层 mock 客户端。契约层用 Moco 走一遍真实 HTTP。两层各管一段。

> **易错点 / 考点：**四个名字要能列全。还要说出 Moco 和前三个的差别：它是独立服务器，用来做 HTTP 服务桩。易错：把 Moco 说成又一个 Java Mock 框架。它也有 Java API，核心仍是独立服务。另一个易错：把 JMockit 和 JMock 当成同一个。JMockit 能模拟的范围更宽，和 JDK 耦合也更紧。

> **过渡：**三个里先看 EasyMock。它把录制、回放、检查写在明面上。

【5.62】EasyMock（2.5 分钟）

> 按 Mockito 的习惯去读 EasyMock，会觉得它多了一步。课件把这步写进定义里了。

>

> 它是一个类库。用简单的方法，给指定的接口生成 Mock 对象。它提供的是对接口的模拟。测试靠录制、回放、检查走完。它能核对调用的种类、次数、顺序。也能让 Mock 返回指定的值，或抛出指定的异常。

>

> 三步分开看。录制，是先跟替身对词。待会儿有人调 execute，你就交回这个 ResultSet。回放，是把替身放进被测代码，按词应答。检查，是跑完看剧本演完没有。约好三次，只来了两次，就报错。

>

> 课件上的代码就是这个架子。先 EasyMock.createControl，得到一个 IMocksControl。再用 control.createMock。造出 Connection、Statement、ResultSet 三个替身。JDBC 这三层，正好一层套一层。

>

> 为什么得先录？替身一跑起来就在被动应答，中间改不了剧本。Mockito 看起来能事后再查，是因为拆开了。when 管它怎么应答。verify 事后去对调用日志。你没写 verify 的调用，它就不管。一个管演什么，一个管演了没有。

>

> 还有一条谱系。EasyMock 当年主要模拟接口。Mockito 能模拟普通类，也能模拟 final 类。JMockit 连静

> 态方法、构造器都能模拟。能模拟什么，就是三个框架的差别轴。

> **提问：**Mock 的行为，为什么要在被测代码运行之前就定好？不定，会怎样？

> **预设回答：**

- 答"跑起来它就在应答，改不了"——对。这个顺序是结构决定的。

- 答"可以边跑边改"——有的框架能在 Answer 回调里改。测试会变成跟被测代码赛跑，很难读。

- 追问：Mockito 的 verify 为什么可以事后？——它不改剧本。它只核对调用日志里发生过没有。

> **易错点 / 考点：**录制、回放、检查三步必考。还要能说：可验证调用的种类、次数、顺序。易错是把 verify 说成"验证返回值"。verify 验的是调用。返回值归 when、thenReturn 那段。

> **过渡：**再看现在用得最多的 Mockito。三个动作先分开。

【5.63】Mockito（2 分钟）

> 一条测试里，造替身、定返回值、查调用，三件事常常搅在一起。Mockito 把它们拆成三个动作：设置依赖性，Stub 方法调用，验证交互性操作。

>

> 设置依赖性，是把替身造出来，塞进被测对象。两种写法。代码里 mock 一个类。或者字段上标 @Mock，再用 @InjectMocks 注进去。这一步回答协作者从哪来。别理解成去配一套依赖注入框架。

>

> Stub 方法调用，是给这个 mock 打桩。写成 when(mock.getPrice()).thenReturn(100)。mock 是角色。打桩是动作，术语叫 stubbing。所以一个 Mock 也可以只当 Stub 用。前面说过，Mock 的行为可能像 Stub。就是这个意思。

>

> 验证交互性操作，就是 verify。看方法有没有被调，调了几次，参数是什么。纯 Stub 工具不干这件事。Mockito 和它们的差别就在这儿。

>

> 顺序别倒。先设置依赖，再打桩，再执行，再验证。桩改的是应答规则，得写在执行前面。验证读的是调用记录，只能放在执行后面。

>

> 写的时候按三段放。准备段集中 mock 和 when。执行段一行调用。断言段集中 verify 和 assert。测试起步依赖的名字是 spring-boot-starter-test。它默认就带 Mockito。先把这三步写顺，比先纠结选型有用。

> **提问：**when(mock.save(any())) 写在被测方法调用之后，会发生什么？

> **预设回答：**

- 答"桩打晚了，没效果"——方向对。更准确一点：Mockito 会把那次调用当成一次真实调用，默认交回 null。你事后再打桩，既没改变刚才那次，还可能弄脏调用记录。

- 答"一定会报错"——不保证。有时会查未完成的 stubbing。有时静静地失效。静静失效更危险。

- 追问：怎么避开？——when 全部放在准备段。mock 造出来就打桩。

> **易错点 / 考点：**三个动作要对上 API。设置依赖是 mock 和 @Mock。Stub 调用是 when 和 given。验证交互是 verify。易错：把"设置依赖性"理解成配置 Spring。这里只是准备替身，再注进被测对象。

> **过渡：**三个动作拆开，就是下面这些 API。

【5.64】Mockito 主要功能（3 分钟）

> 文档一长，人就想跳。Mockito 日常就靠几条 API 撑着。

>

> mock 和 @Mock 用来创建 mock。Answer 规定被调用时跑哪段代码，返回值可以看入参。MockSettings 用来指定行为方式。返回对不上需求，就自己写一个返回接口的扩展。也就是自定义 Answer。

>

> when 和 given 指定 mock 怎么行动。前者是原写法。后者是 BDD 写法。

>

> spy 和 @Spy 做部分模拟。默认去调真实方法，调用仍然可以被验证。mock 是全假的，没打桩的方法交回

> 0 或 null。spy 包着真对象。你只改需要的那一个。十个方法里只想钉住 getDiscount，用 spy。用成 mock，其余九个全空，被测逻辑会莫名其妙地失败。

>

> @InjectMocks 把标了 @Spy 或 @Mock 的字段自动注入。先按类型找，再按名字找。构造器、setter、字段注入都会试。注不进去，多半是类型对不上，或字段是 final。

>

> verify 检查方法是不是用给定的参数被调用过。可以配 times、never、atLeastOnce，也可以用 inOrder 查顺序。注意，不写次数时，verify 就是 times(1)。调了两次，它也会红。把它当成"至少一次"，重复调用就漏过去了。

>

> 参数不必写死。any 能匹配任意表达式。还有 eq、anyString。要看传进去的对象长什么样，用 @Captor 抓住，再做断言。同一个方法里，匹配器和具体值不能混着写。混了会抛 InvalidUseOfMatchersException。

>

> 先把四条用熟：mock、when、verify、@InjectMocks。spy 和 @Captor 会用就行。

> **提问：**一个类有十个方法。你只想让 getDiscount 返回固定值，其余走真实逻辑。用 @Mock 还是 @Spy？用反了会怎样？

> **预设回答：**

- 答"用 @Spy"——对。这就是部分模拟。

- 答"用 @Mock，只给 getDiscount 打桩"——其余九个方法变成空的。返回 null 或 0。被测逻辑会在不相干的地方失败。

- 追问：什么时候反过来用 @Mock？——真实逻辑对本条测试有害的时候。比如它会去连数据库。判据就一条：真实逻辑是在帮忙，还是在添乱。

> **易错点 / 考点：**六个 API 的作用要能对上。尤其 spy 是部分模拟，会调真实方法。@InjectMocks 是自动注入。verify 是检查调用。三个高频错：verify 默认 times(1)；匹配器混用会报错；@Spy 和 @Mock 弄反。判断题还爱出"Mockito 只能 mock 接口"。现在能 mock 类，也能 mock final 类。老项目要 mockito-inline，新版

默认就支持。

【思政落点 10】：verify 要求你说清三件事：调了几次，参数是什么，顺序对不对。判断一件事有没有发生，不能靠"我觉得发过了"。通知、清缓存、关口子，都一样。写关键逻辑时问一句：这件事怎样才能被证明发生过？

过渡：清单里 @Mock 被单独拿出来了。一个注解为什么要占这么长？

【5.65】Annotation Type Mock（2.5 分钟）

注解写上了，字段却是 null。这种报错很常见。先看 @Mock 比一路写 mock() 多解决了什么。课件列了五条。

>

第一条，把一个字段标记为 Mock。你声明字段，加上注解，框架帮你把对象造出来。

>

第二条，允许速记创建。手写要把类名写两遍。左边写下 OrderRepository repo。右边再写一次 mock，参数还是这个类。注解只留一行。@Mock，后面跟上 OrderRepository repo。

>

第三条，尽量减少重复的 Mock 创建代码。一个测试类常有五六个协作者。每行一个 mock，准备段就把执行和断言挤没了。挪到字段上，方法里只剩业务。

>

第四条，让测试类和验证错误更好读。报错里带着字段名，比如 orderRepository。比"某个 OrderRepository 的 mock"再加一串 hashCode 好认。

>

第五条，自动检测 MockedStatic 类型的静态 Mock。静态方法被替掉之后，整个 JVM 里都生效。忘了关，别的用例会被污染。注解把创建和关闭交给框架，不用自己写 try-with-resources。省代码是一层。管住生命周期是另一层。人最容易忘的就是收尾。

>

> 注解要生效，还得初始化。类上加 @ExtendWith(MockitoExtension.class)。或者调 MockitoAnnotations.openMocks(this)。少了这一步，@Mock 字段就是 null。把替身注进被测对象，是 @InjectMocks 的事。两个注解，两件事，别并成一件。

> **提问：**为什么说注解除了省代码，还管生命周期？用静态 Mock 举个例子。

> **预设回答：**

- 答"静态 Mock 用完必须关，忘了会污染别的测试"——对。它是全局生效的。

- 答"注解会自己开关"——对。框架知道何时创建、何时释放。比手写 try-finally 稳。

- 追问：这告诉你注解一般在省什么？——成对出现的动作，打开和关闭、创建和释放。人容易忘的是后一半。

> **易错点 / 考点：**使用 @Mock 之前要做什么？启用 Mockito 扩展，也就是 @ExtendWith(MockitoExtension.class)，或者调用 openMocks。易错：以为加了 @Mock 就会注入被测对象。注入是 @InjectMocks 的事。

> **过渡：**刚才 given 这种写法一闪而过。它是 BDD 的入口。

【5.66】支持 BDD（2.5 分钟）

> given(mock.getPrice()).willReturn(100)。这句读起来像在说需求。Mockito 对 BDD 的支持，就是把测试写成这种句子。

>

> BDD 是行为驱动开发。它用业务语言描述行为。句式是 Given、When、Then。给定账户余额 100 元。当取出 50 元。那么余额应为 50 元。业务方看得懂。代码也能照着写。

>

> BDDMockito 里三个方法和这三句对上。given 对应 Given。给定替身怎么行动，和 when 等价。willReturn、willThrow 是它的搭档。语序从"调用了就返回"换成"给定这次调用，将返回"。then(mock).should() 对应 Then。写法是 then(mock).should().save(any())。读出来就是：那么它应该保存过。它验的是交互，不是返回值。

>

> 方法名也可以写成 given、when、then 那一长串。报告拿出来，业务方能看。在金融、医疗这类要留验收痕迹的地方，这份报告能当证据。

>

> 别把它当成新能力。它和 when、thenReturn、verify 完全等价。测得不会更准。一个类里只留一种风格。混着写，读的人要在两套语序里来回切，比只用一种更累。

>

> 语法是为了让人读懂。它不让断言变得更正确。

> **提问：**组里一半人写 when 和 verify，一半人写 given 和 then。功能一样，会不会出问题？出在哪？

> **预设回答：**

- 答"不会，功能一样"——结果一样。读代码和评审时要切换两套语序，容易漏。

- 答"风格不统一，维护会疼"——对。约定一旦定下，就别在一个类里混。

- 追问：怎么定？——有业务方要看测试报告，就统一成 BDD。没有，when 和 verify 更短。

> **易错点 / 考点：**Mockito 的 BDD 支持，是 BDDMockito 提供的 given、willReturn、then。语义和 when、thenReturn、verify 等价。易错有两个。把 BDD 说成一种测试层级。它是写需求和协作的方法。把 then(mock).should() 说成在断言返回值。它验证的是交互。

> **过渡：**用例会写了。接下来要问：我怎么知道测够了？

【5.67】测试覆盖率工具（2.5 分钟）

> 用例全绿，你仍可能没碰到半个类里的代码。通过只说明你跑过的路径是对的。没跑过的，绿灯一个字都不提。

>

> 覆盖率工具回答的就是这块缺口。它不判对错。它只统计执行痕迹：哪些指令走过，哪些分支一个方向都没走。它照出从来没被跑过的代码。质量要上去，还是靠你补用例、补断言。工具不会替你把错改掉。它也不告诉你断言写得对不对。走过的行，断言仍可以是一句恒真。

>

> 每种语言都有自己的实现，名字下一页点。选型先看三条，别看谁的界面花。跟着语言走。Java 用这个生态里的主力，别换着玩。报告要能被别人消费。XML 这一类格式，CI 和 SonarQube 才吃得下。Cobertura、LCOV 也是常见口径。插桩方式会改字节码，会改源码，或者另挂探针。运行时代价不一样，得问一句能不能接受。

>

> 还有一件事比名字重要。覆盖率不进每次构建，数字就是当天的快照。新类加上去，没人盯，过几个月就掉下来。它是会过期的指标。掉了要能告警，才算接进了流程。想起来才跑一次，看完就忘。组里常见的做法，是期末前跑一次，截个图放进报告。下周有人加了类，数字掉下去，没人知道。

>

> 所以先把期待放准。它告诉你哪里还没测。它不证明测过的地方是对的。

> **提问：**覆盖率工具能不能发现"这段逻辑写错了"？不能的话，它的价值在哪？

> **预设回答：**

- 答"不能，它只看执行没执行"——对。它没有对错，只有到没到。

- 答"价值是告诉团队哪里还没测"——对。未知变成一份缺口清单。

- 追问：覆盖率很高，线上还是出大事，正常吗？——正常。那一部分里断言不够，或者缺陷在没设计到的输入上。它是下限，不是质量证明。

> **易错点 / 考点：**工具和语言要对上。JaCoCo 是 Java。BullseyeCoverage 是 C 和 C++，支持分支覆盖。Testwell CTC++ 也是 C 和 C++。Coverage.py 是 Python。gcov、lcov 同样是 C 和 C++。易错：把 JaCoCo 说成测试框架。把 BullseyeCoverage 说成 Java 工具。把覆盖率工具说成"能把覆盖率提上去"。它只测量。往上提，靠写用例。

> **过渡：**名字现在摊开。先看各语言用谁来插桩。

【5.68】代码覆盖率分析工具（2 分钟）

> 人眼数不过来哪条指令走过。一次回归几千条用例，只有机器记得。做法是在程序里埋探针。跑到哪儿，就

> 把哪儿标上。跑完汇总成报告。

>

> Java 这边的主力是 JaCoCo。它对着编译后的 class 文件做插桩。然后收集覆盖率，再出报告。它不改你的源码。源码是后来对照着看的。

>

> C 和 C++ 常用 BullseyeCoverage。课件特别写了：它支持分支覆盖率，英文是 Branch Coverage。一个 if 的两个方向，switch 的每个 case，还有短路的与、或。只看"这行走过没有"，这些藏身处都漏。同一类里还有 Testwell CTC++。嵌入式里很常见，能在目标板上跑。还有 Parasoft 的 Jtest 和 C++Test，以及 CoverageMeter。

>

> 名单后半段还有 Jcov、CodeCover。Python 那边是 Coverage.py。

>

> 语言不同，链条一样。插桩，记录，汇总，出报告。工具名是这条链在各语言上的样子。别只背名字，把链条留住。

>

> 覆盖率拿来度量，别拿来当交卷的目标。目标一旦是数字，人会写只为刷绿的用例。把它当度量，才会问这个分支为什么没走到。

> **提问：**覆盖率从六成多涨到九成多。能不能由此说质量变好了？

> **预设回答：**

- 答"涨了就是好事"——先追问那一段是怎么涨的。新增了用例，还是删掉了难测的代码？后者只是数字好看。

- 答"不能。走过不等于断言对"——这是准确答案。断言写成恒真，覆盖率再高也没验到东西。

- 答"要看分支覆盖动了没有"——内行。语句覆盖涨了、分支没动，多半只是多跑了几行直路。

- 收口：看三件事。哪个维度在涨，涨在哪些类上，断言有没有跟上。

> **易错点 / 考点：**判断题爱出"覆盖率高等于测试充分"。答案始终是：它是必要不充分条件。插桩对象是编译后的 class，不是往源文件里插入统计代码。工具和语言别配错。BullseyeCoverage 是 C 和 C++，主打分支覆

盖。JaCoCo 是 Java。Coverage.py 是 Python。

过渡：落到 JaCoCo，它到底给你哪几个数字？

【5.69】实现覆盖率度量：Jacoco（3 分钟）

报告上一排数字，圈复杂度经常被当成又一种覆盖率。JaCoCo 的分析维度有六个。指令、分支、行、方法、类、圈复杂度。前五个才是覆盖。

>

指令最细。数的是字节码指令走过多少。分支看每个判定的出口。if 只走了 true，没走 false，叫部分覆盖。行按源码行算。方法和类更粗。方法里只要一条指令走过，这个方法就算覆盖。圈复杂度不是覆盖率。它是复杂度。McCabe 的式子是 e − n + 2p。JaCoCo 按方法算。它告诉你，路径要走全，至少得有几条用例。圈复杂度高、覆盖又低的方法，最该先补。

>

统计基于 Java 二进制码，不需要源文件。行能高亮，是因为 class 里带着调试信息。指令能对上源码的某一行。关掉调试信息再编译，就只剩方法和类。

>

插进去的方式叫 on-the-fly instrumentation。JVM 启动时挂上 java agent。类被加载的那一刻改字节码、塞探针。磁盘上的 class 不动。另一种叫 offline，先插好再跑。它能接到基于 Java VM 的框架上。课件点了 OSGi、Web 容器、EJB 服务器。所有 class 文件版本都兼容。Kotlin、Scala 编译出来也是 class，所以不同的 JVM 语言都能统计。

>

报告有三种：HTML、CSV、XML。HTML 给人看。另两种给流水线做门禁。执行数据可以随时从覆盖率代理里取。口子是远程协议，也可以走 JMX。构建侧用 ANT task，或 Maven 插件，收集数据并生成报告。项目页在 https://www.eclemma.org/jacoco/。

>

读报告时把圈复杂度摘出去。它指路，它不是第六种覆盖率。

提问：一个方法行覆盖是满的，分支覆盖只有一半。问题在哪？你该补什么用例？

预设回答：

- 答"代码有 bug"——推不出。覆盖低只说明没测到，不说明测错了。

- 答"有分支没走，要补让判定走反方向的用例"——对。if (x > 0) 只测了正数，负数那一支没进过。异常分支也常这样。

- 答"行覆盖满了就够"——短路右边、空的 else，照样可以完全没测。

- 收口：先看分支缺了哪个方向。再用圈复杂度看这个方法还差几条路径。

易错点 / 考点：问"JaCoCo 六个维度"要全答。问"哪些是覆盖率"要把圈复杂度摘出去。这是最爱考的陷阱。"基于二进制码、无需源文件"常被改成"必须有源代码"。on-the-fly 是运行时插桩，不改文件。offline 是先插桩再运行。报告三种要记全：HTML、CSV、XML。口诀可以用"指、分、行、方、类、圈"。前五个从细到粗，最后一个是复杂度。

【思政落点 11】：覆盖率容易被当成交差的凭证。数字过了线就收工，报告一贴，任务就算完。它更有用的用法是指路：哪一段从来没人走过。看到红格子、黄格子，先问这块为什么没人测，出错会怎样。别先问怎么把它刷绿。

过渡：这些数字从哪来？先看 agent 启动时干了什么。

【5.70】on-the-fly instrumentation（2 分钟）

启动参数里多了 -javaagent:jacocoagent.jar。你的 main 还没跑。JVM 先读这个 jar 的清单，找到 Premain-Class，调用 premain。类在 org.jacoco.agent.rt.internal，名字就叫 PreMain。

>

方法是 premain。参数一个是 options，一个是 Instrumentation。它只做三件事。

>

先读配置。new AgentOptions(options)。等号后面那串键值对在这里拆开。排除哪些包，数据文件落在哪，端口是多少。

>

> 再把通道建起来。Agent.getInstance 拿到代理。createRuntime(inst) 造出运行时。接着 runtime.startup，参数是 agent.getData()。收数据和交数据的口，就此打开。前面说的远程协议和 JMX，就是从这里露出去的。

>

> 然后注册转换器。inst.addTransformer，放进 CoverageTransformer。构造参数里有 runtime，也有 agentOptions。最后一个是 IExceptionLogger.SYSTEM_ERR。从此每个后加载的类都先过它。读进来，插探针，再交给 JVM。

>

> SYSTEM_ERR 是在说：某个类分析失败，记到标准错误，然后放行。别让覆盖率把应用搞崩。第三方 jar 的字节码太老、转不了，就这一类不被统计。别的类照常。

>

> premain 自己不改字节码。它只把钩子装上。插桩发生在每一次类加载。启动前已经躺在 JVM 里的核心类，不在这批之列。

提问：某个第三方 jar 的字节码很老，转换器分析不了。按这段代码，会发生什么？

预设回答：

- 答"整个应用启动失败"——不会。异常交给记录器，记到标准错误，类原样放行。

- 答"那个类不统计，其他类照常"——对。覆盖率不能变成应用的故障源。

- 追问：这是什么原则？——旁路设施要能降级。监控、埋点、日志出了问题，主流程还得走。

易错点 / 考点：premain 是 agent 入口，不是插桩实现。问"JaCoCo 在哪一步改字节码"，答类加载时的 Transformer。启动参数是 -javaagent:xxx.jar=key=value。别和 -agentlib、-Xbootclasspath 混。addTransformer 只对之后加载的类生效。

过渡：钩子装上了。插桩这一下，是 ASM 在做。

【5.71】on-the-fly instrumentation –续（2 分钟）

class 的字节流走进 instrument，出来时已经不是原样。把它看成一条加工线。进门是原始 class。出门是带

探针的 class。用的库是 ASM。

>

先读。ClassReader 把字节流解析成能遍历的结构。

>

再备写出器。new ClassWriter，把 reader 传进去，标志是 0。这里重写了 getCommonSuperClass。方法体直接抛 IllegalStateException。ASM 判断两个类型谁是父类时，默认会去加载它们。我们正站在类加载这一环里。再触发加载，轻则死锁，重则栈溢出。JaCoCo 只插探针，不做类型变换，这个能力用不上。抛异常就是拒绝它。传入 reader，是为了复用常量池，让输出小一点。不是去读源码。

>

探针得有地方记"走没走过"。ProbeArrayStrategyFactory.createFor 按类的形态选。放静态字段，每次新建，还是从外面注入。工厂替你定。

>

访问者串成一链。外层是 ClassProbesAdapter。它套着 ClassInstrumenter，再套到 writer。外层管这个类有几个探针。中层管探针怎么落进字节码。里层负责写出去。

>

reader.accept 开始遍历。第二个参数是 ClassReader.EXPAND_FRAMES。栈映射帧被展开，插的位置才准。然后 writer.toByteArray，交出新的字节码。结果还得过 JVM 校验。改的是字节码，不是源文件。

提问：插桩时为什么要这么小心地不加载别的类？你在别处见过同类约束吗？

预设回答：

- 答"怕慢"——慢是次要的。类加载有锁。加载过程中再加载，容易死锁。

- 答"类加载器一隔离，可能拿到错的版本"——对。同名类在不同加载器下是两个类。

- 追问：还有哪儿一样？——Java agent、热部署、APM 探针。凡是工作在类加载期的代码，都得守这条。

易错点 / 考点：改字节码用的库是 ASM。ClassWriter 构造时传入 reader，是为了复用常量池，不是读源码。EXPAND_FRAMES 是遍历选项，不是插桩策略。别和探针放哪混在一起。

过渡：类这一层过了。方法这一层怎么插？

【5.72】on-the-fly instrumentation –续（2 分钟）

> 走到某个方法，visitMethod 被回调。它先不插。等方法的结构走完，再动手。

>

> 它先问下一层接不接这个方法。参数是 access、name、desc、signature、exceptions。不接，mv 是 null，就用空的 MethodProbesVisitor 顶上。接了，就交回 MethodSanitizer 的一个匿名子类。名字就是清理器。有的编译器吐出的字节码不规整。先收拾，否则过不了校验。

>

> 动手在 visitEnd。到这儿，方法的全貌才齐。按顺序做四件事。

>

> 先 markLabels。LabelFlowAnalyzer 标出跳转标签，看清哪条指令跳到哪。控制流的汇合点就在这些标签上。探针要挂在分支分开、又重新汇合的地方。挂在每条指令前面，体积和耗时都会爆。分支计数还会重复。

>

> 再 new 一个 MethodProbesAdapter，把探针和标签对上。

>

> trackFrames 为真时，再套 AnalyzerAdapter。它模拟操作数栈和局部变量表。你多插了指令，栈深度就变了。不重算栈帧，JVM 校验直接抛 VerifyError。注释里写的"注入数据分析"，就在 methodProbes.accept 这一下。为假时少套分析器，直接 accept。

>

> 三段连起来看。premain 装钩子。instrument 改类。visitMethod 改方法。指令和行，来自探针落在哪。分支来自标签怎么配对。圈复杂度来自控制流图。六个维度都是这次遍历带出来的。

> **提问：**探针为什么要插在控制流的汇合点？插在每条指令前面不行吗？

> **预设回答：**

- 答"每条都插更准"——方向对，代价是体积和耗时爆掉。一个类可能被插进成千上万个探针。分支还会被重复

计数。

- 答"汇合点少，计数还能对上"——对。分支覆盖要的是这个判定走没走两个方向。一个判定一个探针就够。

- 追问：这是在取舍什么？——精度和运行开销。埋点系统都要做这个取舍。

> **易错点 / 考点：**三个方法的分工要能对上。premain 是入口。instrument 是类这一级。visitMethod 是方法这一级。AnalyzerAdapter 和栈映射帧是为了防止校验失败，不是在分析业务。考法多是填空、排序、简答，不会让你写 ASM。错误说法："JaCoCo 修改源文件来插入统计代码。"它从头到尾碰的都是 class。

> **过渡：**探针插完，报告长什么样？看一份真实的。

【5.73】示例（2 分钟）

> 绿格子让人安心，红格子让人去补。黄格子最容易被放过。一份 JaCoCo 报告，三眼就能读出个大概。

>

> 先看颜色。绿色是完全覆盖，分支都走到了。黄色是部分覆盖。常见情况是一个判定只走了一边。红色是完全没执行。红色好办，没测就是没测。黄色看着像测过。有一半逻辑，断言其实没碰到。

>

> 再看层级。从包到类，到方法，再到行。上面几层是汇总，用来找哪个包最差。点进方法，才知道哪个方法最差。点到行，才看见哪一行是黄的。先从上面定位，再从下面补用例。一上来翻源码找红线，最慢。

>

> 然后看列。每一层都有 Missed 和 Covered。栏位就是六个维度：指令、分支、行、方法、类、复杂度。一个模块该不该补，看两个数是不是一起出现。分支覆盖低，圈复杂度高。分支多，又没测到，风险最高。

>

> 还有一处容易看反。Missed 是没覆盖到的数量。Covered 是已经覆盖的数量。只看百分数，会把很小的类和很大的核心类放在同一档。先看 Missed 有多少，再看百分比。

>

> 报告可以按覆盖率从低到高排。排在最前的类，就是最该补的地方。能高亮到某一行，还是因为 class 里有

> 行号调试信息。统计口径在字节码上。源码是拿来展示的。

> **提问：**红格子很多，可都是 getter、setter 和自动生成的代码。黄格子只有三个，都在核心业务方法里。你先处理哪边？

> **预设回答：**

- 答"先补红的，数量多"——数字思维。getter 和 setter 补上去，质量几乎不动。

- 答"先看黄的，因为它在核心逻辑里"——对。更稳的做法是把自动生成的代码排除出统计。agent 的 excludes 就是干这个的。报告只反映人写的、有业务含义的代码。

- 追问：这说明统计之前还要做什么？——先划定范围。范围不划，数字没法比。

> **易错点 / 考点：**黄是部分覆盖，通常是分支只走了一边。不是"跑得慢"。用报告指导补测，要答出两个方向：从上往下定位，从下往上补。案例若问下一步，标准答法是三条。找到分支没覆盖的方法。排除自动生成的代码。补上让判定走反方向的用例。

> **过渡：**报告能读了。Eclipse 里怎么随手跑？

【5.74】代码覆盖率度量（2 分钟）

> 编辑器左边突然染上绿黄红。Eclipse 里干这件事的插件叫 EclEmma。课件的定位就一句：它基于 JaCoCo。

>

> JaCoCo 是引擎。插桩、统计、算数，都是它。EclEmma 是外壳。你在 IDE 里点一下，跑完在源码上染色。别把方向说反。不是 JaCoCo 去给 EclEmma 当插件。

>

> 它缩短的是你等到结果的时间。刚写完一个方法，用覆盖率方式跑一次。绿黄红出现在编辑器边上，几秒就有。等流水线出报告，常常要几十分钟。拖得越久，人越不想回头改。

>

> 局限有三条。它只覆盖你这台机器上这一次跑的用例。别人写的、流水线上的集成测试，都不在里面。它离不开 Eclipse。CI 服务器上没有这个 IDE。本地这张图，也不能拿去当"我测过了"的汇报。范围不一样。

>

> 两边一起用。本地用 EclEmma 做即时自检。流水线上用 JaCoCo 的 Maven 或 Gradle 插件，出正式报告，做质量门禁。同一个引擎。一个给开发自己看，一个给团队验收。

>

> 装好以后，Eclipse 里会多一个覆盖率视图。可以对整个项目跑，也可以只对一个测试类跑。范围收在你正在改的那个类上，反馈更快，改起来也更准。

>

> 工具把没测到的地方摆出来。够不够，标准得人来定。

> **提问：**流水线上已经有 JaCoCo 报告了。开发还有必要在本地装 EclEmma 吗?

> **预设回答：**

- 答"没必要，重复了"——漏了等待时间。报告出来时，人往往已经去做下一件事了。

- 答"有必要，本地改起来快"——对。补一句：本地看到的数字不能拿去汇报。它只反映你这台机器上跑过的东西。

- 追问：两个的差别是什么？——EclEmma 是开发自检。流水线上的 JaCoCo 报告是团队的验收凭证。引擎是同一个。

> **易错点 / 考点：**EclEmma 是基于 JaCoCo 的 Eclipse 插件。别答成两个独立工具，也别把方向说反。历史上它曾基于 EMMA，现在的引擎是 JaCoCo。题目若说"EclEmma 基于 EMMA"，按现在的关系判错。另一题："IDE 里的覆盖率等于项目的覆盖率。"错。范围不同。

> **过渡：**单个方法测过了。拼起来还认不认对方?

【5.75】5.3 持续集成测试（1 分钟）

> 函数单独跑都对，一拼就错。这种错不在函数体内。参数顺序对不上。时间一边是本地，一边是 UTC。A 以为 B 会重试，B 以为 A 只发一次。单元测试把对方都换成了替身。这些错出不来。

>

> 从这里起是持续集成测试。教材分成两节。一节是系统集成的模式与方法。一节是持续集成及其测试。课件合成了一节，名字就叫持续集成测试。内容没少，编排并在一起了。答题时看题干用哪套词。听课按课件的顺序走。

>

> 往下分三层。单体架构里，代码在一个进程中。问题是按什么顺序把模块拼起来。三条策略：自顶向下，自底向上，混合。微服务里，模块变成独立部署的服务。你常常拉不起别人的服务。答案转到契约测试。再往上，持续集成和 CI/CD 流水线把拼装变成每次提交都发生。测试得快，得准，还得排得上顺序。

>

> 这一块没有公式，看着像听听就行。进了项目，很多人先碰到的是把两块拼通，不是先去设计算法。拼的时候，你得同时看懂两边的接口。集成测试看的是接口和交互。不是把单元测试再跑一遍，也不是去验收整份业务需求。拼错的代价，往往比单个函数算错更大。因为你要两边一起查。

>

> 带着一个问题往下听。你们上一次联调，卡了几天，卡在谁等谁？

> **提问：**函数的单元测试全部通过。集成时最可能先在哪一类问题上翻车？

> **预设回答：**

- 答"接口对不上"——最常见。参数名、类型、空值约定、编码，有一处不一致就炸。

- 答"环境"——也很真实。配置、依赖版本、数据库字符集，单元测试里全被替身换掉了。

- 答"并发和时序"——单线程测不出。两个服务一并行就露出来。

- 收口：三类都发生在边界上。集成测试就是盯着边界看。

> **易错点 / 考点：**课件叫持续集成测试。教材分成"系统集成的模式与方法"和"持续集成及其测试"。看题干用哪个词。集成测试的对象是接口和交互，不是把所有单元测试再跑一遍。它也不是系统测试。它不验收业务需求，只看拼起来对不对。

> **过渡：**这一块的目录就四条。先把关系理顺。

【5.76】持续集成测试内容（2 分钟）

目录有四条。先别一条一条背。先分清哪两条在讲架构，哪两条在讲节奏。

>

> 单体架构的集成测试，微服务架构的集成测试，问的是怎么拼。差别不在谁更难写。在你能不能控制对方的运行。单体里模块都在你手里。想先测谁，就先测谁。微服务里对方是另一个团队，另一套发布节奏，另一套数据库。你连把它拉起来都做不到。所以单体给你策略。微服务给你契约。

>

> 后两条问节奏。持续集成及其测试，讲多久拼一次。CI/CD 流水线，讲拼完怎么送到生产。拼装若从项目末期干一次，变成每天干很多次，测试就得变。变慢了，门禁会先被绕过去。

>

> 集成测试夹在中间，最容易被挤掉。单元测试写得快，跑得快，大家愿意写。系统测试有验收标准，不敢省。集成测试不写，边界上的错查不出来。写全了，环境、数据、依赖都是成本。这一节要的不是"做不做"。是用较少的成本，把边界上的风险盖住。

>

> 四条是一条线。架构从单体换成微服务，上一条的答案哪里失效，下一条就补哪里。发布从每月一次变成每天很多次，上一条的成本哪里撑不住，下一条就改哪里。听的时候盯住边界。边界是什么，两边是谁，传过去的是什么。

提问：微服务的集成测试，为什么不能套用单体的三种策略？

预设回答：

- 答"因为服务是分开部署的"——这是表层。分开之后，你控制不了对方何时可用。跨团队、跨语言去造桩和驱动，成本很高。

- 答"技术栈可能不同"——也对。Java 调 Go，桩得按对方的协议造。协议一改，假服务就得改。

- 追问：所以缺的不是一个更好的假模块，是什么？——一份双方都认、又能自动核对的约定。这就是契约测试。

易错点 / 考点：四条要能默写。单体架构的集成测试，微服务架构的集成测试，持续集成及其测试，CI/CD 流

水线。别把持续集成当成一种测试类型。它是节奏，是实践。也别把流水线只当成部署工具。在这一节里，它是测试跑在上面的载体。

过渡：从最小的场景看起。单体里，先测谁？

【5.77】单体架构的集成测试（3 分钟）

上层要先跑，下层一行都还没写。拿什么顶？单体里模块都在一个进程中。先拼谁，决定你要造哪一种假模块。课件给了两种。

>

自顶向下，从最上层的主控模块开始，往下一层一层拼。下层还没进来，就造一个 Stub，也叫桩。桩不干实事。它按约定交回固定结果。查询用户的桩，永远交回一个写死的用户。这样，下层还没写，上层的控制逻辑就能跑。骨架、业务流程、界面导航，可以最早被看到。这类逻辑错了，返工最大。坏处是底层被放到后面。数据库、算法、对接第三方，最容易出错，反而最晚测。桩给的是假数据。上层跑通，可能只是假通过。真实数据一来就崩。

>

自底向上，从最底层往上拼。底层写好了，没有上层来调它。你得临时写一个调用者，叫 Driver，驱动模块。它准备输入，调用被测模块，把结果打出来。好处是底层核心先被测透。坏处是主控和整体流程要很晚才看得到。驱动要写很多。这些代码不进产品，工作量却是真的。

>

两边是对称的。自顶向下缺的是下层，所以要桩。自底向上缺的是上层，所以要驱动。桩是被调用的那一方。驱动是发出调用的那一方。说反了，这道题就没了。

>

想早看到流程，选自顶向下。想早验核心，选自底向上。界面和流程要先给客户看，用前者。算法对不对决定成败，用后者。风险最大的那一层在哪，就先到哪一层去。

提问：核心风险是利息算法必须算准。界面下个月才定稿。你选自顶向下，还是自底向上？为什么？

预设回答：

- 答"自顶向下，好早做演示"——题目没要求演示。算法在底层，自顶向下会把它排到后面。

- 答"自底向上，先把算法测透"——对。判据是风险最大的模块在哪一层。在底层，就自底向上。

- 答"两个都要"——这就是下一页的混合策略。

- 收口：哪一层风险最大，就先到那一层。

> **易错点 / 考点：**对应关系必考。自顶向下用 Stub，模拟被调用的下层。自底向上用 Driver，模拟调用它的上层。桩是被调用方，驱动是调用方。优缺点要各能说两条。陷阱题："自顶向下不用写任何辅助代码。"错。桩也得写，只是通常比驱动轻。

> **过渡：**两种各有一半好处。两头一起上，就是混合策略。

【5.78】单体架构的集成测试 –混合策略（2 分钟）

> 桩越往下越厚，驱动越往上越多。两头往中间挤，中间就能少造一点。这就是混合策略，也叫三明治策略。

>

> 上层走自顶向下。下层走自底向上。两边往中间推进，在中间那一层汇合。上层只造到某个深度。下层的驱动只写到某个高度。中间直接对接。桩和驱动的总量都会下来。省的不是思路，是脚手架的工作量。

>

> 第二件好处是风险优先。分层不是按层数对半切。核心模块在哪一层，就把那一层当成先测的起点。让它尽早跑在真实代码上，而不是跑在桩上。常说的关键模块优先，就是这个切法。

>

> 第三件是能并行。上层一个小组，下层一个小组，同时往前走，到中间对接口。进度紧的项目，这是实在的收益。

>

> 代价也要说清。哪些层归上，哪些层归下，中间那一层谁负责，得先想清楚。拍脑袋切，两边会在中间打架。中间层是汇合点。两边的假设在这儿碰上。这一层的接口要最早定下来，而且尽量稳住。两套节奏还得有人协调，管理成本会高一些。

>

> 反面是一次性集成，也叫大爆炸。模块全写完，一次拼上跑。它不一定马上失败。失败了往往不知道是哪一块引进来的。几十个模块同时在场，一个错抛出来，你分不清出处。极小的系统或许还能这么干。稍微大一点，一般不推荐。

> **提问：**混合策略里，中间那层汇合点应该选在哪？选错了会怎样？

> **预设回答：**

- 答"选在正中间，上下各一半"——形式上说得通。工程上更该选接口最稳、变更最少的那一层。

- 答"选在业务逻辑层"——常见的合理答案。业务层的接口相对稳。界面和数据访问可以各自往前走。

- 追问：选错的后果？——汇合层天天在变，两边刚对上又得改，等于白干。前提是分层清楚，接口相对稳。

> **易错点 / 考点：**混合策略又叫三明治策略，别名要会。收益能说三条：少写桩和驱动，风险优先，可以并行。代价能说两条：分层得先想清，中间层压力大。三种策略怎么选：看风险在哪一层，看接口稳不稳，看进度要不要并行。一次性集成，也叫大爆炸，多当反面出现。判成"推荐"就错。

> **过渡：**单体的顺序定了。服务拆开之后，拼法要换。

【5.79】微服务架构（2 分钟）

> 方法调用失败，堆栈能追到底。换成网络调用，事情就没这么干脆。先把微服务看成什么。不是把单体切成小块就完了。一个服务对上一个业务能力。自己一个进程，自己一个数据库，自己一条发布流水线。部署单元和团队边界对齐了。对测试来说，结构变了。

>

> 单体里，模块之间是方法调用。传的是对象。编译器帮你查类型。同一个进程，出错了堆栈一路到底。微服务里是网络调用。对象先变成字节流。JSON，或者 Protobuf。跨进程，跨机器，还可能跨语言。网络会超时，会重试，会乱序。对方会重启，会限流，版本也会对不上。一个字段能不能为空，两边理解不同，序列化会在运行时炸。出错的方式比方法调用多出一截。

>

> 单元测试把依赖都换成替身，测得很顺。被换掉的，往往正是容易出错的边界。想端到端测，得把多个服务、

> 数据库、消息队列按配套版本拉起来。很多团队付不起。联调就堆到项目末期，一次赌上。

>

> 策略要改。测试金字塔不要只往上堆端到端。单元测试守住每个服务内部。接口测试和契约测试守住服务之间。端到端只留极少的关键场景，做冒烟。难点不在测得多深。在边界怎么守住。

> **提问：**单体里"调一个方法失败"，和微服务里"调一个接口失败"，最大的差别是什么？

> **预设回答：**

- 答"微服务会超时"——这是其中一种。更关键的是部分失败。单体里一个方法炸了，这次请求就失败了。微服务里 A 成功了，B 超时了，C 其实成功了，只是响应丢了。系统停在一个说不清的中间状态。

- 答"网络不可靠"——对。不可靠会带出重试、幂等、补偿。这些都是要补的新场景。

- 追问：单元测试能覆盖这些吗？——不能。单元测试里根本没有网络。只有集成测试碰得到。

> **易错点 / 考点：**微服务集成测试看的是通信和数据的边界，不是每个服务内部的业务逻辑。判断题："微服务下应该加强端到端测试。"错。常见做法是减少端到端，加强契约测试和接口测试。答题时能写出"部分失败"，往往就是得分点。

> **过渡：**边界有两面。调用通不通，数据存得对不对。分开看。

【5.80】微服务架构的集成测试（3 分钟）

> 开发环境一切正常，一到测试环境请求全是 401。先别改业务代码。微服务的集成测试，课件给了两个方向。

>

> 第一个，测试外部的通信，确认通信是否通畅。检查核心功能即可。不是把业务逻辑再测一遍。它用来发现协议层的错。课件列了三类。丢失 HTTP 报头。SSL 使用错误。请求和响应不匹配。

>

> 这三类，单元测试里出不来。业务测试里又容易被业务错误盖住。丢报头，常见的是认证头、内容类型、追踪头。用 Postman 手测时，人会手动带上。写进代码就忘了。SSL 用错，常见是证书校验被关掉，或者双向认证只配了一半。本地走 HTTP 好好的，一上 HTTPS 就成片返回 500。请求和响应不匹配，是字段名的大小写、时间格式、枚举值、能不能为空。两个团队按各自的理解去实现同一个接口，最容易出这种错。

>

> 第二个方向是数据库访问测试。目标有三条。微服务用的数据结构，要和数据库相符。ORM 里设置的映射关系，要查一遍。数据库访问模块遇到网络出错，要能妥善处理。ORM 的全称是 Object Relational Mapping，对象关系映射。它把表和对象对起来。这层翻译错了，编译期不会报。运行时才会交出错误的数据。

>

> 举两个最容易踩的样子。代码里字段可空，库里那一列却是 NOT NULL。枚举在代码里是字符串，库里存的是整数。一方加了新值，另一方不认识。单元测试用的是内存里的假数据，发现不了。真的连上库，走一次映射，才会露出来。库连不上的时候，服务是马上失败，还是把请求挂住、把连接池耗光。这是韧性问题。也得靠集成测试看。

>

> 两个方向盯的是同一类翻译。协议把对象变成字节流。ORM 把对象变成表里的行。业务对不对，单元测试管。翻译对不对，集成测试管。

> **提问：**一个服务在开发环境一切正常，一上测试环境全是 401。你第一个怀疑什么？

> **预设回答：**

- 答"账号密码错了"——先排除。环境之间更常见的是报头没带。开发环境的网关可能默认塞了认证头，测试环境没配。

- 答"证书或 HTTPS"——很可能。本地 HTTP、环境里 HTTPS，SSL 的错经常表现成认证失败。

- 答"接口版本不一致"——也对。请求和响应的字段对不上，有的框架直接回 4xx。

- 收口：三种答案正好对上课件列的三类协议层错误。

> **易错点 / 考点：**两个方向要能默写，各举一例。数据库访问测试不是在测 SQL 快慢。它测映射，也测出错时怎么处理。ORM 要会写全称：Object Relational Mapping。陷阱："通信测试要把业务功能全部重测一遍。"错。课件写的是检查核心功能即可。

> **过渡：**通信和数据都盯住了。还有一个麻烦：对方的服务你根本拉不起来，怎么测？

【5.81】微服务架构的集成测试 – 续（3 分钟）

> 联调群里最常出现的一句是："我字段没改，你怎么取到空了。"两边都觉得自己没错。错在中间那份没写下来的约定。

>

> 这份约定叫消费者驱动的契约测试。英文全称是 Consumer Driven Contract Testing。缩写 CDC。契约规定的是调用方和被调用方之间的 Request 与 Response。调用方是 Consumer，也就是消费者。被调用方是 Provider，也就是提供者。它锁的是数据怎么交互。不是业务对不对，也不是快慢。

>

> 你发哪些字段、什么类型、哪些必填。对方回哪些字段、什么类型、哪种情况回哪个状态码。格式就这些。

>

> 为什么要消费者来驱动？契约从消费者的需求长出来。先按这个需求设计用例，再产生一份契约。然后拿这份契约去验证提供者。顺序不能倒。不是提供者宣布"我支持这些字段"。是消费者说"我依赖的就是这些"。

>

> 提供者往往想多给。多一个字段，自己这边常常还能跑。对消费者，多给通常没事。少给、改名，才会炸。所以契约只锁住用到的部分。这些一动，验证就会失败。

>

> 还有一种观点：用契约测试，或用协议测试，来做集成测试。微服务里，集成要抓的问题多半在边界。边界上的问题，常常是两边对格式的理解不一样。把理解写成契约，各自去验。比把整条链路拼起来碰运气，更知道错在哪一侧。

>

> 它替代不了单元测试。也替代不了端到端。它只回答约定有没有被遵守。业务算得对不对，它不管。位置在金字塔中间。跨团队那点协作，变成两份可以自动跑的测试。

> **提问：**为什么契约要由消费者来写，而不是让提供者把支持的字段全列出来？

> **预设回答：**

- 答"消费者更清楚自己用什么"——对。全量字段看起来完整，却没人保证你以后不改其中某一项。

- 答"提供者写会更准"——反例很直白。提供者写了 name，消费者依赖的是 user_name。按提供者的说法来验，会给出错误的通过。

- 追问：这和哪条原则是一回事？——接口的约束由使用方定义。和"需求由用户提"是同一件事。

> **易错点 / 考点：**CDC 的中英文要能写全：Consumer Driven Contract Testing，消费者驱动的契约测试。契约锁的是 Request 和 Response 的格式，不是业务逻辑。顺序是消费者需求、产生契约、再验证提供者。问能不能替代集成测试：能替代"验接口约定"那一段，不能替代端到端的业务验证。

> **过渡：**好处具体有哪几条，下面拆开讲。

【5.82】契约测试及其优势、工具（3 分钟）

> 两个服务各自都是绿灯，一对接口却能拖上三天。拖的不是代码，是人在等。

>

> 第一条好处，是把服务集成的难度降下来。原来要把 N 个服务拼成一个大联调。契约把它拆开。消费者这边，对着假的服务端跑自己的测试。提供者那边，按契约回放，做接口测试。课件的说法是：分解为单元测试和接口测试。一个 N 乘 M 的联调，变成 N 加 M 两份可以分开做的活。人不用同时在场。

>

> 第二条，团队能离线把验证做完。联调的成本几乎降到零。零的是等待。以前要占环境、约时间、等人回消息。一个字段改三天，两边都卡着。现在消费者拿着契约就能验自己。提供者拿着同一份也能验自己。谁都不用等谁。

>

> 第三条，接口一改，有迹可循。以前改名，靠群里喊一声。喊晚了，消费者第二天才炸。契约是进版本库的文件。改了它就要跑验证。跑不过，就是有消费者会挂。影响面是查出来的，不是猜的。

>

> 工具课件列了四个。Janus、Pact、Pacto，再加上 Spring Cloud Contract。课件写成 CDT 框架。规范说法是 CDC。答题写 CDC。Pact 是后面示例的主线，文档在 docs.pact.io。Spring Cloud Contract 用 Groovy 或 YAML 写契约。它和 Spring 项目更近。

>

> 2026 年 7 月 6 日，Spring 官方博客宣布一件变动。Spring Cloud Contract 退出后续的发布列车。维护交给原作者 Marcin Grzejszczak。项目以 Stubborn Contract 继续。作者同日说明，这不是另写一套的分叉。这是官方博客和作者说明，不是社区传闻。课堂主线仍是课件上的 Pact。

>

> 工具会换。动作不换：写契约，生成桩，两边都验。

> **提问：**提供者把响应里的 userId 改名成 id。功能没变，原有测试也全绿。契约测试能不能拦住？

> **预设回答：**

- 答"能，字段名写在契约里"——对。改名就是破坏契约。这正是"变更有迹可循"要拦住的事。

- 答"不能，值还一样"——要改口。对消费者，字段名就是接口。userId 读不到就是空。后面一串空指针，功能并没有"没变"。

- 追问：那怎么安全改名？——先加新字段，两边并行一段时间。消费者迁完，再删旧字段。

> **易错点 / 考点：**三条好处要能说全：拆成单元测试和接口测试、离线验证、变更留痕。四个工具名都要在。方向不要答反：不是提供者写契约、消费者只准遵守。判断题若说"契约测试能发现业务逻辑错误"，这是错的。它只保证约定被遵守。

> **【思政落点 12】**：契约测试最值得学的地方，是它把"承诺"变成了可验证的东西。消费者说我要这三个字段、这个格式，提供者说我保证给你——然后双方各自跑测试，不用见面就能确认对不对。工程协作里最怕的就是"我以为你会给"，契约把这种模糊的期待写成明文。写清约定、守住约定，是职业信用最具体的样子。

> **过渡：**道理有了。看 Pact 把这三步怎么跑起来。

【5.83】示例：CDC 测试框架（2 分钟）

> 约定如果只躺在 wiki 里，它就不是考卷。没人拿它去对真实响应，绿不绿全看运气。

>

> 课件给的示例是 Pact。地址是 docs.pact.io。换别的 CDC 工具，骨架也是这三段。

>

> 第一段在消费者。测试不连真实服务端。Pact 在本地起一个假服务端。你声明：我发这个请求，你回这个响应。请求打到假服务端上，消费者的代码被完整跑一遍。跑完，这次交互被记成一份 JSON。这份 JSON 就是契约。

>

> 第二段，把 JSON 交到一个中枢。这个中枢叫 Broker，可以当成契约仓库。带上版本号。提供者由此知道：现在有哪些消费者，依赖我的哪些行为，依赖的是哪个版本。

>

> 第三段在提供者。从 Broker 拉下契约，启动自己的真实服务。按契约里的请求真的发一次。拿真实响应和约定逐项比。字段、类型、状态码。全对，这条契约算验证过。有一项不对，构建就红。红在提供者自己的流水线里，不等到联调那天才看见。

>

> 三段之后还有一问。部署前问 Broker：我要上的这个版本，契约都验证过了吗？过了才允许上。契约测试就接到了发布上。后面的 CI/CD，接的就是这一问。

>

> 术语再对一下。规范写法是 CDC，Consumer Driven Contract。不要写成 CDT。

> **提问：**契约一直是绿的，上线后还是对接错了。最可能漏了哪一环？

> **预设回答：**

- 答"契约没覆盖新调用"——最常见。契约只记录已经写进测试的调用。新调用没进契约，就测不到。

- 答"验证没跑在准备发布的那个构建上"——也很关键。验的是另一个包，等于没验。

- 答"红了也没拦住发布"——测试跑过，却没人拿它当门。红了照样上。

- 收口：看三件事。覆盖了多少调用，验证跑在哪个构建上，红了能不能拦住发布。

> **易错点 / 考点：**三段要能复述：消费者产出契约，Broker 中转，提供者验证。Broker 是契约仓库。假服务端在消费者这一侧。两者别混。判断题若说"契约测试要求两边同时联调"，错。它恰恰是为了不用同时在场。

> **过渡：**三段里最容易看错的是第一段。消费者到底在测谁。

【5.84】Consumer 测试（2 分钟）

> 消费者这边跑测试，对面可以根本不是真服务。测的是你自己拿到响应之后，会不会解析错。

>

> 核心道具是 Stub server，桩服务端。它测的是消费者自己。我这个客户端，拿到符合约定的响应，能不能正确解析、转换、往下用。它不测提供者。对面是假的。

>

> 假服务端怎么知道回什么？靠你在测试里声明期望。你写：GET 这个路径，带这个头，请回 200 和这段 JSON。Pact 就按这个期望做事。它真的起一个 HTTP 服务，听一个端口，按你声明的规则应答。所以它叫 Stub server，不是一个方法级的替身。走的是真的 HTTP。序列化、报头、状态码，才有机会露出来。只 Mock 一个方法，这些全被绕过去了。

>

> 顺序是这样。先声明期望。再起假服务端。消费者发出真请求。假服务端按期望应答。你断言解析结果对不对。跑完，把这次交互写成契约。

>

> 这一步要看清。契约是测试跑出来的副产品，不是手写的文档。文档会过期。测试一改，契约跟着更新。忘了更新，测试就红。

>

> 异常场景请在这里多造。返回 500 怎么办。返回空数组怎么办。字段类型和约定不一致怎么办。超时怎么办。对面是假的，造这些几乎不花钱。真实联调里，你没法让别人的服务专门给你回一个 503。

> **提问：**期望写错了。真实接口回的是字符串，我期望成了数字。消费者这边的测试会不会红？

> **预设回答：**

- 答"不会红，假服务端按我写的回"——对。这是这个方法的盲区。它只证明"我能处理我以为的响应"。它不能证明真实响应就是那样。

- 答"会红，因为和真实服务对不上"——那是提供者验证阶段的事。消费者这一侧不会红。

- 追问：盲区谁来补？——提供者验证。真实响应和契约不符，构建就红。两端凑在一起，才闭环。

> **易错点 / 考点：**Stub server 是真起一个 HTTP 服务，不是方法级 Mock。消费者测试测的是消费者，不是"接口对不对"。契约由测试生成，这是它和手写接口文档的差别。全绿不能说明真实接口没问题。

> **过渡：**这一侧测的是自己。对面那个真服务，下面才上场。

【5.85】Provider 测试（2 分钟）

> 契约文件已经生成了，提供者还没拿真服务去对。文件再完整，也不等于线上还是这个样子。

>

> 这边的道具是 contract verifier，契约验证器。课件给的仓库可以拆开记。前半是 github.com/pact-foundation。后半是 pact-provider-verifier。合起来就是 pact-foundation 那个验证器仓库。

>

> 它做的事和消费者相反。从 Broker 把契约拉下来，让真实服务来应考。契约里的每一条请求，原样发给正在运行的提供者。拿到真实响应，和约定逐项比。状态码、报头、响应体的字段和类型。全对就过。有一项不对就失败。失败信息会告诉你：哪条契约、哪个字段、期望什么、实际什么。

>

> 课件上的 Contract verifier tests，指的是由契约驱动的验证测试。用例不是手写的。它由契约自动生成。这就是图上的 Auto-generate。你还会看到 Contract.groovy 和 Call API。前者是契约，Spring Cloud Contract 里常用 Groovy 来写。后者是验证器真的把调用发出去的那一步。

>

> 提供者不用知道消费者是谁，也不用知道对方用什么语言。它只读契约，发 HTTP，比响应。一个 Java 服务，可以被前端、Python 脚本、Go 服务同时来验。跨语言是天生的。桩往往每种语言各写一套。契约是一份文件，大家共用。

>

> 还有一个前提。验证必须跑在你准备发布的那个构建上。数据库和配置要用真的。再 Mock 一层，验的还是

> 假的，等于白跑。

> **提问：**提供者验证失败了。是提供者的错，还是消费者把契约写错了？

> **预设回答：**

- 答"提供者的错"——多数是。行为改了，没通知别人。

- 答"也可能是消费者按想象写的"——也存在。真实接口从来不是那样。这时要判谁对。契约反映的是合理需求，提供者改。契约写错了，就改契约。

- 追问：怎么少吵一点？——把契约当接口文档来评审。争议有一份文件可对，不靠聊天记录。

> **易错点 / 考点：**消费者用 Stub server 生成契约。提供者用 contract verifier 验证契约。提供者的用例是自动生成的，不是手写的。不需要把所有消费者都启动起来，只需要契约。也不要求两端同一种语言。

> **过渡：**两端分开看完了。放回一张图里对一下。

【5.86】示例（2 分钟）

> 左右对不上的时候，中间缺的常常是同一份文件。人在群里对过字段，文件里却没有。

>

> 图上两个身份要分清。Server side 是 service provider，服务提供者。Client side 是 service consumer，服务消费者。课件把 consumer 拼成了 concumer，少了一个 s。答题按 consumer 写。

>

> 客户端这一侧，测试面对的是按契约应答的假服务端。测的是自己的解析和处理。跑完，产出契约文件，交进共享的契约库。

>

> 服务端那一侧，从契约库拿出同一份文件。验证器把每条请求打到真实服务上，比对响应。通过了，说明我仍满足这些消费者的期待。不通过，说明这次改动破坏了某人。

>

> 两边之间传递的不是代码，不是环境，也不是一句口头承诺。是一份可以执行的文件。协作从"人对人解释"，

> 变成"文件对文件验证"。

>

> 再和单体里的桩比一下。单体里，每个调用方各写一个假模块。微服务里，契约是一份共享的约定。N 个消费者对一个提供者，是 N 条契约。提供者验证一次，把这些一起对完。它不是凭空的新发明。它把"各自造一个假的"，换成了"共享一份约定，还能拿真服务反着验"。

>

> 联调成本能降下来，是因为大头本来就不是写假服务。大头是约时间、对版本、等人回消息。契约把这截协调拿掉了。

> **提问：**为什么说契约测试是把"桩"升级了，而不是发明了一个全新的东西？

> **预设回答：**

- 答"都是拿假的顶替真的"——有一半对。差别在于：桩的"假"是调用方私有的想象。契约的"假"是双方确认过的约定，而且能被真实服务反着验证。

- 答"因为能自动验证"——这是升级的那一下。从"假设对方会这样"，变成"验证对方确实这样"。

- 追问：桩的老毛病是什么？——漂移。对方改了，你的假模块还按老样子应答。测试全绿，上线全炸。契约一漂移，提供者那边就会红。

> **易错点 / 考点：**provider 是提供者，也是 server side。consumer 是消费者，也是 client side。考试爱把两边对调。简述闭环时，必须说到契约在中间传递。桩和契约的差别：私有想象，对上共享约定，再加上真实服务能反着验。

> **过渡：**契约里的字段从哪来，别靠记忆。

【5.87】如何获取接口信息（2 分钟）

> 字段名要是从群截图里抄，桩会停在旧接口上。下周代码改了，截图还在，测试照样绿。

>

> 课件给的来源，是 Swagger 生成的动态接口文档。动态的意思是：文档不是人另写的一份，是从代码里生成的。你在控制器方法上写注解，标明路径、参数、返回类型。工具扫一遍，产出一份结构化描述。现在通

> 行的规范名是 OpenAPI。Swagger 是这套东西最常用的工具链。代码改了，文档跟着变。不容易出现文档写着旧版、代码跑着新版。

>

> 文档和契约不是一回事。文档描述"我准备提供什么"，多半是提供者视角的全集。契约规定"你依赖什么"，是消费者视角里用到的那一部分。文档是写契约最好的原料。你从文档里摘出自己会读到的字段，就是契约的初稿。

>

> 结构化描述还能省掉一大半客户端代码。工具可以生成客户端 SDK，也可以生成桩。类型、序列化、请求方法，都是生成的。链是：注解，到 OpenAPI 文档，到生成的客户端或桩，再到契约，再到两端验证。人负责确认"我依赖哪些"。

>

> 不要把文档直接当成契约。文档写"这个字段可能为空"，你的代码按非空来用。文档不会红。运行时会给你一个空指针。文档负责声明。契约负责验证。声明写得很全，也不等于考得过。

>

> 契约别写太肥。只写你实际依赖的字段。多写一项，提供者一改你就红，维护成本会先把这个实践拖垮。

> **提问：**有了自动生成的接口文档，是不是就不用写契约了？

> **预设回答：**

- 答"不用了，文档已经很全"——这是常见误解。文档只声明，不验证。提供者改坏了，文档不会因此变红。

- 答"要写，文档只是原料"——对。而且只摘消费者实际依赖的字段。多摘，契约会很脆。

- 追问：写多细合适？——只写你用到的。这就是消费者驱动，也是控制维护成本的办法。

> **易错点 / 考点：**Swagger 文档是从代码生成的，不是另维护的一份手写说明。OpenAPI 是规范名，Swagger 是工具链名，别混。文档不等于契约：一个声明，一个验证。契约粒度是"只写用到的字段"。

> **过渡：**文档和契约都还没有的时候，今天怎么测。

【5.88】利用 Mock 技术解除微服务之间的依赖（3 分钟）

服务 B 归别的组，服务 C 还在写，服务 A 今天却得测。你要是非等它们都在线，这组测试就永远起不来。

>

> 办法是用 Mock 把微服务之间的依赖解开。图上被测的是微服务 A，标了 SUT，也就是被测系统。它要调的 Service B 和 Service C，都用 Mock 顶上。旁边还能看到微服务 B、微服务 C。打在 A 的边界上的，是 API Test。Mock 的内容，课件写的是请求和响应信息。也就是协议层的一对报文：请求长什么样，响应长什么样。这和契约里的一次交互，是同一个粒度。

>

> 你要测 A 的一个接口，A 却要调 B 和 C。B 不归你管，C 可能还没写完。Mock 放在 A 控制不了的那一端。A 把请求发出去，Mock 按预设回。于是 A 可以单独跑，也可以重复跑。

>

> 好处有三条。测试可控。你可以让 B 回 500、回超时、回一堆脏数据，再看 A 怎么处理。真实环境里，别人不会专门给你造这些。测试快。不起 B，不起 C，也不连它们的库。解耦进度。B 没做好，A 照样能测，组与组不用互相干等。

>

> 风险也有两条，不讲就会用歪。最大的是漂移。Mock 的报文照着当时的 B 写。B 后来改了，Mock 不知道。测试继续全绿，上线全炸。第二条是测了假的。连 A 自己的数据库、消息队列也一起 Mock 掉，这个测试就退回单元测试。集成要看的那一层没了。

>

> 所以 Mock 只用在你控制不了的远端。你自己这一侧的库、缓存、消息，用真的。防漂移的办法，就是把这份报文收成契约，让提供者去验证。Mock 让你现在能测。契约让你以后还测得准。

提问：外部依赖全 Mock 了，测试天天绿，一上线就出问题。问题出在哪？

预设回答：

- 答"Mock 写错了"——有可能。更常见的是 Mock 从没和真实服务对过，也就是漂移。

- 答"没有一环在验证真实接口"——这是根子。缺的不是 Mock，缺的是提供者那边的契约验证。

- 答"把自家数据库也 Mock 了"——那样测的就不是集成。集成测试的价值没了。

- 收口：问三句。报文是不是契约，契约有没有在提供者端验证，自家依赖是不是真的。

> **易错点 / 考点：** SUT 是 System Under Test，被测系统。解除微服务依赖，答 Mock，也就是测试替身。粒度是请求和响应报文。Mock 越多，离真实越远。边界卡在你控制不了的远端。收益是可控、快、进度解开。风险是漂移和失真。

> **过渡：** 依赖解开了。这些测试什么时候跑，不能靠想起来。

【5.89】CI/CD 与持续测试（1 分钟）

> 仓库里测试并不少，可发版前才有人想起来跑。跑晚了，改一处要牵一片。

>

> 单元测试、Mock、覆盖率、契约，这些做法很多组都知道。做不好，常常不是不会写。是没有一个机制让它们每次都发生。这周赶进度，单元测试先不写。这个版本急，覆盖率先不看。文档里还有，流程里没了。

>

> CI/CD 就是那个机制。它把"跑测试"从靠自觉，变成流水线上自动发生的事。不过，就过不去。提交之后，不能靠某个人记得去点运行。点忘了，这套测试就等于没写。你们课设也一样。测试代码交了，却只在答辩前跑一次，和发版前才跑是同一类问题。这一块不是在讲怎么运维机器。是在讲测试怎样才真的落地。

>

> 还有一层。手工测试的经验，往往集中在两三个人身上。人一走，经验就断。流水线上留下的是资产：用例、脚本、门禁标准、报告基线。谁都能接着用。把测试放进流水线，不只是为了快。也是让能力不要跟着某一个人走。

>

> 后面你会看到流水线长什么样，为什么持续交付会逼出持续测试，以及持续测试要哪几个特点。课件里还有一份实践框架。学完，你要能把这一章的技术放到一条时间上：每项在哪一段跑，跑完卡什么。

>

> 先问自己一句。你们现在的项目，从提交到能上线，中间有几步是自动的？

> **提问：**单元测试写得很规范，可每次发版前才跑一次。这算不算落地了？

> **预设回答：**

- 答"写了就算落地"——不算。跑得越晚，问题越贵，改起来牵涉也越多。

- 答"得进流水线，每次提交都跑"——对。这就是这一块要解决的事。

- 追问：每次都跑全套，会不会太慢？——会。所以后面要讲分层和精准。不是所有测试每次都跑全。

> **易错点 / 考点：**在这门课里，CI/CD 是测试的执行和门禁，不是运维专题。持续集成、持续构建、持续交付、持续部署，四个词要能分开。判断题若说"持续测试就是自动化测试"，错。持续测试强调它在流水线里按节奏发生，范围比"有一批自动化脚本"更大。

> **过渡：**先把几个"持续"分清。分不清，后面的门会对错。

【5.90】CD 倒逼持续测试（3 分钟）

> 一个月只发一次的时候，全组还能挤出两天做回归。一天发好几次，人就排不开了。

>

> 五个词请分开。持续集成，是开发人员频繁把代码合到主干。每次合并都自动编译、自动测试。它治的是集成地狱。以前各写几个月，最后一合，冲突成堆。关键词是频繁，和自动。

>

> 持续构建，是每次代码一变，就自动做出可运行的制品。关注点从"代码能不能合"，转到"制品能不能出"。

>

> 持续交付，是任何时候都有一份可以发布到生产的制品。发布这个动作由人决定。一键能发，按不按，人说了算。持续部署连这一下也交给机器。门禁过了，就自动上生产。交付和部署只差一条：最后上生产，是不是自动的。

>

> 持续测试被这条链逼出来。假如发布从一个月一次，变成一天十次。人工回归在时间上就做不完。你不可能一天组织十次回归。只剩两条路。要么放开质量。要么让测试自动，并且快到跟得上发布。不是测试自己忽

> 然想进步。是交付的节奏压着它必须进步。

>

> 形态会跟着变。测试要分层。大量很快的单元测试挡住大多数问题。少量慢的端到端，守住关键路径。测试要精准。不可能每次全回归，得能算出这次改动影响了哪些用例。测试要左移。等代码写完再测，来不及。编码时就要把问题挡住。

>

> Google Cloud 在 2024 年 10 月 22 日的博客里，发布了 DORA 2024 报告。其中有一个研究估计。AI 采用上升时，交付稳定性估计下降约 7.2%。这是报告里的估计，不是这门课的实验。方向和刚才一样。代码写得更快，若没有小变更，也没有跟得上的测试，交付不会自动变稳。

>

> 有的组装了 CI 工具，测试还是老样子。他们只是把手工步骤自动跑了一遍，测试的形态没变。工具只把结果摆出来。快不快、准不准，还是测试自己的事。

提问：已经做到持续部署，合并过了门禁就自动上生产。测试必须满足什么？

> **预设回答：**

- 答"必须全自动"——必要，但不够。全自动却要跑两小时，还是跟不上。

- 答"要快、要准、要拦得住"——完整一些。快到分钟级反馈。准到只跑受影响的。门禁不过就不能上。

- 答"还得能回滚"——自动部署要配上自动回滚和监控。这是测试右移的部分。

- 收口：持续部署不是多按一次发布。测试和监控得一起到位。这才是"倒逼"。

> **易错点 / 考点：**持续交付和持续部署，差别只在"上生产是否自动"。这是最高频的辨析。五个词要能按自己的定义说清：持续集成、持续构建、持续交付、持续部署、持续测试。倒逼的逻辑是：发布变密，人工回归做不完，测试就得自动、分层、精准。CI 不等于"装了一个 Jenkins"。工具只是载体。

> **过渡：**词分清了。同样几道门，前后一换，反馈就差一截。

【5.91】CI/CD 流水线（二）环节排布（2 分钟）

> 编译错误要是排在半小时的端到端后面，机器就在空转。学生等的也是这半小时。

>

> 同样的门，摆放顺序不同，效果差很远。三条原则够用。

>

> 便宜的放前面，贵的放后面。按常见量级：编译是几秒。静态扫描一两分钟。单元测试几分钟。集成或契约，到十几分钟。端到端可能半小时以上。越早失败越省。一个编译错误，不该等端到端跑完才被看见。

>

> 确定的放前面，不确定的放后面。不确定，是指依赖外部环境。网络抖一下，第三方接口超时，测试数据被别人改了。这类测试放在前面，流水线会时红时绿。组里很快学会：红了再点一次重跑。门禁就废了。不确定的测试往后放，并且单独看它的稳定率。

>

> 只在必要的分支上跑全套。主干和发布分支跑全套门禁。功能分支跑快速的那一套。这不是降低标准。门禁是为了拦住问题进主干、进生产。不是给每个分支都做一遍全身体检。

>

> 失败要快。英文是 fail fast。流水线要尽早失败。测试自己也要尽早失败。一个用例里五个断言，第一个就错了，后面四个不该继续装成"还在测"。套件连数据库都连不上，就不该傻等超时。让错误早一点、明确一点露出来。

>

> 课件后面有一页业界例子，标题是 Amazon 的 CI/CD。图上没有可引用的次数和耗时，这里不编。能带走的是门怎么排，不是厂家的口号。

> **提问：**组里有句话："红了不要紧，点一下重跑就绿了。"这句话暴露了什么？

> **预设回答：**

- 答"测试不稳定"——对，但这是症状。根子常常是不确定的测试放得太靠前，或者用例之间共享了状态。

- 答"门禁形同虚设"——更要命。大家习惯重跑之后，红灯会从拦截变成抽奖。信任一掉，所有红灯都会被忽略。

- 追问：怎么治？——把不确定的测试隔开，统计不稳定率。过了阈值就修，或先下线。宁可少一个用例，也不要一条会撒谎的流水线。

> **易错点 / 考点：**排布要能说三条。便宜在前、贵在后。确定的在前、不确定的在后。主干跑全套，分支跑快速的。fail fast 是尽早失败、把错误说清楚。不要答成"失败了就赶快重试"。也不是每次把所有测试都跑全。成本上做不到，也不必要。

> **过渡：**原则先放着。课件上能读出来的环节，按名字过一遍。

【5.92】CI/CD 流水线（三）完整环节与门禁（3 分钟）

> 一次提交进流水线，后面会连续过十几道门。门的顺序，决定你多快看见错。

>

> 入口有四种。代码提交最常见，一推送就跑。定时触发补的是代码没动、环境变了，比如夜里跑一次更全的回归。事件触发处理"我没改，依赖库、镜像或上游版本变了"。人工触发用来指定某次构建、某个分支。只配代码提交，后三种会漏。

>

> 入口后面是排队限流。提交一密，构建会抢机器，大家一起变慢。限流让关键分支先跑，也免得机器被打满。

>

> 检查这一段，课件上有个总名，就叫测试。里面再分开。先编译。编译不过，后面都不用跑。再做静态扫描。Sonar 增量扫描只看这次改动。全量扫一遍，往往几十分钟，撑不住每次提交。安全扫描放在单元测试前面。高危漏洞不该再进入后面的测试。然后是单元测试，再是自动化测试。

>

> 放行靠质量门禁。各项结论在这里汇总，达标才往下。之后才是发布。先部署 STG，再进灰度环境，最后才到生产环境。每一级都在放大风险。问题尽量留在流量小的地方。

>

> 还有几项不在主箭头上，缺了主流程会卡。还有代码跨数据中心同步，多地拿到的才是同一份。编译结果预

> 推送，制品预推送，把产物提前送到环境里，缩短等待。动态扩容，在构建高峰临时加机器。

>

> 课件有一页例子，标题就是 Amazon 的 CI/CD。截图上没有可引用的次数，不编。能核对的是这些环节名。

>

> dora.dev 现在的公开说明，把交付指标写成五项。吞吐这边有变更前置时间、部署频率。失败部署要多久恢复，也算在吞吐里。不稳定这边，除了变更失败率，又加了部署返工率。返工率看的是为了线上事故临时补上去的发布。这是指南上的口径，不是我们的实验数据。门禁若只盯快、不盯返工，绿灯也可能是补丁换来的。

> **提问：**安全扫描为什么放在单元测试前面？单元测试有时更快。

> **预设回答：**

- 答"因为安全问题更严重"——方向对。说得再具体一点：高危漏洞属于不该进入后续流程的问题。处置不取决于后面的测试结果。越早停越省。

- 答"因为扫描很快"——不一定。全量扫描可以很慢。能放前面，靠的是增量扫描。

- 追问：所以排布不止"便宜在前"？——还有一条。能立刻判定为不可接受的检查，尽量往前放。

> **易错点 / 考点：**排序题爱考这条主干：编译，静态扫描，安全扫描，单元测试，自动化测试，质量门禁，部署 STG，灰度，生产。四种触发都要能说，事件触发最容易漏。Sonar 增量扫描是关键词，全量扫描撑不起每次提交。预推送、跨数据中心同步、动态扩容，常出现在多选题里。

> **过渡：**门搭好了。持续测试要达标，课件给了四个说法。

【5.93】持续测试的特点（3 分钟）

> 全量回归还跑得动的组，多半仍是一个月发一次。发布一密，这套跑法会先崩。

>

> 四个特点要连着记。足够快。准确且有效。平滑有序。与 CI/CD 集成。

>

> 足够快。课件写的是：整个测试过程要快。一方面高度自动化。另一方面，业务端到端还要做探索式测试。能自动的交给脚本。脚本盖不住的路径，靠人按检查表做探索。

>

> 准确且有效。课件说，被测系统往往很复杂，不可能做全回归，要推行精准测试。全回归的成本随系统变大，发布也在变密，全回归就跑不动。精准测试要回答：这次改动，影响哪些用例。靠代码变更、调用链、历史上的缺陷，把回归缩到够用的那一截。你要能说清"为什么只跑这些"。说不清，就是全回归换了个名字。

>

> 精准要能落地，得有覆盖映射。Datadog 的文档把这件事叫 Test Impact Analysis。它按每条用例覆盖的文件，决定能不能跳过。没有映射，就不能跳。主干默认不跳过，免得漏到发布前。依赖库、编译选项、外部服务变了，默认配置可能看不出来。数据文件变了，也一样。这是厂商文档。前身叫 Intelligent Test Runner。

>

> 平滑有序。课件把它说成三个打通。从测试左移到测试右移。从单元测试到系统测试。从静态测试到动态测试。左移管预防，右移管上线后的监控。单元管内部，系统管整体。静态管不运行就能看见的。动态管必须跑起来才露出来。这些活动要能排到一条时间上。

>

> 与 CI/CD 集成。课件写的是：和研发的持续构建、持续集成、持续部署、运维这些环境接上。再快的测试，不进流水线，就不会被执行。再准的选择，不接构建，也选不出范围。测试是构建的一部分，不是构建完了再另起一次。

>

> 四个特点要凑齐。少了和流水线的集成，前三条不会自己发生。

> **提问：**"我们每次都跑全量回归，所以测试很充分。"这句话在持续测试里有什么问题？

> **预设回答：**

- 答"太慢，跟不上发布"——最直接。全回归跑不动，最后会变成发版前跑一次。持续就没了。

- 答"跑得多不等于测得准"——也对。很多缺陷在没被设计进去的用例里。

- 追问：精准是不是少测？——不是少测。是把力气放在受影响的地方。影响范围证明不了，就不能缩。

- 收口：差别不在用例条数。在有没有依据。

> **易错点 / 考点：**四个特点的名字和解释，简答爱考。精准测试不要答成"少写几条用例"。要答：按变更的影响来选用例。平滑有序的三个打通：左移和右移，单元和系统，静态和动态。持续测试不是一刻不停地跑。它是在流水线里按节奏、分层、有选择地跑。

> **过渡：**四个说法是目标。一个组怎样做下去，课件指向一份白皮书。

【5.94】持续测试实践框架（1 分钟）

> 自动化脚本写得很整齐，过了半年却没人再维护。技术没丢，人散了，脚本就废了。

>

> 课件给的是《持续测试白皮书》，可以扫码下载。扫之前先问一句：组里现在谁负责这些脚本。答不上来，就是缺口。难的从来不是会不会写脚本。难的是让一个组织持续做下去。只讲技术，常见结局就是：几个人写了一批脚本，半年后没人管。

>

> 课上用四块来对现状。这是读这份材料时的检查角度，不是说白皮书的目录一定印成这四个标题。组织与角色：谁负责测试资产，谁维护流水线。流程与规范：什么改动跑什么测试，门禁标准是什么。技术与工具：分层、环境、数据。度量与改进：反馈时间、逃逸缺陷率、用例稳定率。缺一块，另外三块会跟着退化。

>

> 你以后想推动这件事，先别开口就谈工具。拿这四块对一遍。缺人负责，就先定责任。没人知道脚本是不是还绿，就先把反馈时间和逃逸率记下来。没有数字，坏是无声的。

>

> 白皮书的用法像检查表。读一遍记不住。拿着它一条条对现状，缺口会现形。它是参考框架，不是强制标准，也不是一张成熟度奖状。它给你一套能一起说话的词。你说"我们卡在度量"，对方知道你在说哪一块。话能对上，改进才谈得下去。

> **提问：**脚本写得很好，半年后全废了。按这四块看，最可能缺哪一块？

> **预设回答：**

- 答"缺组织，没人负责维护"——最常见。测试资产没有明确的人管，会跟着人员流动一起消失。

- 答"缺度量，没人知道它失效了"——也很关键。没人看稳定率和反馈时间，烂掉的过程是安静的。

- 追问：这说明什么？——持续测试是组织能力，不是某个人的手艺。它叫"持续"，就是这个原因。

> **易错点 / 考点：**考法通常很轻，多是"持续测试实践框架"或白皮书这个入口。四块可以当论述的架子：组织、流程、技术、度量。不要把白皮书说成强制规范。说"实践框架"或"参考"更准确。

> **过渡：**章内的技术讲完了。用六条把整章归位。

【5.95】本章小结（3 分钟）

> 六条如果只剩关键词，答题时会漏掉限定词。限定词往往就是得分点。

>

> 第一条，代码规范和代码评审。要理解它值在哪里，并且持续做。方式是互为评审、集体评审。只由资深的人挑新人，标准会变成一个人的口味。互为评审，是组里一起把标准立起来。课件说团队有多方面的收益。不只是缺陷变少，还有知识留下来，规范变成改得动的意见。

>

> 第二条，代码分析工具。课件说今天很流行。两个落点要同时有。开发时在本地扫描，写的时候就看见，反馈最快。再和 CI/CD 流水线接上，保证每次提交都被扫。只有本地，靠自觉。只有流水线，作者要等到提交才知道。

>

> 第三条，单元测试，也就是 UT。要理解它的价值，也要理解 xUnit 的实现机制。JUnit 不是魔法。它就是注解、反射、断言、运行器。机制清楚了，换一种语言的 xUnit，上手会快很多。

>

> 第四条，掌握 JUnit 这类工具，也掌握 mock 工具怎么用。关键是取舍。什么时候用桩，什么时候用 Mock，什么时候干脆不替身。够用就行。替身越多，离真实越远。

>

> 第五条，单元测试是否充分。要看代码的语句覆盖和分支覆盖。一般用覆盖率工具自动分析。两个限定都在：语句和分支，以及自动分析。人算不出那张表。数字要看，不能只看数字。

>

> 第六条，今天提倡持续测试，用来促进持续交付。契约测试值得关注。从开发自测开始，把跨服务的约定尽早验掉。这比等到联调再发现字段对不上，要彻底一些。

>

> 六条可以收成一条线。评审和静态分析，在不运行时抓问题。单元测试把函数钉住。覆盖率看语句和分支够不够。契约守住服务边界。持续集成让这些每次都发生。

> **提问：**六条里只能留三条，你留哪三条？理由是什么？

> **预设回答：**

- 答"评审、单元测试、持续集成"——稳。一个管不运行就能看见的问题。一个管零件对不对。一个管每次都发生。

- 答"覆盖率、契约、持续集成"——也能成立。但覆盖率是度量。它自己不产生质量，得有单元测试撑着。

- 追问：为什么持续集成几乎总会被留下？——前几条是能力。它是让能力每次都发生的那一环。

> **易错点 / 考点：**简述本章，六条关键词要能写出来。覆盖率那条的两个限定别漏：语句和分支，自动分析。契约测试是这一章后半的重点，也是和集成、持续交付接上的那一环。论述可以按这条线组织：评审，单元测试，覆盖率，契约，持续集成。

> **【思政落点 13】**：这一章从代码评审讲到持续集成，中间隔了单元测试、Mock、覆盖率、契约测试。看起来是六个不相干的技术点，其实是一条线：**每一层都在替下一层挡住一类问题**。评审挡"看不懂"，单元测试挡"单个跑不对"，契约测试挡"拼起来对不上"，流水线门禁挡"改坏了没人发现"。做工程的人要有这种分层意识——不是把某个工具用到极致，而是知道每一层该管什么、不该管什么。

> **过渡：**归位之后是动手。实验 2 只交测试代码。

【5.96】实验 2： 单元测试（3 分钟）

> 实验说明书写着测控制器，交上来的却是 Service 里的加法。能跑，也不算这道实验。

>

> 任务写在课件上。用 JUnit，测 Spring Unit Testing 里的 ItemController。写出测试类，提交测试代码。

>

> 它是 Spring MVC 的控制器。收 HTTP 请求，调 Service，再回响应。业务算得对不对，是 Service 的事。控制器看三件。参数能不能绑上。Service 回什么，它就回什么。异常和错误码对不对。

>

> 不要真连数据库。真连了，就不是单元测试。两种做法都行。一种用 WebMvcTest，只加载 Web 层。Service 用 MockBean 换掉。另一种不起容器。new 一个 ItemController，参数里放 Mockito 做的 Service。报告里写清你选了哪种。

>

> 查询要有数据，也要无数据。新增要有合法、缺必填、类型不合法。改和删要有存在，也要不存在。Service 抛异常时，控制器回什么。按准备、执行、断言来写。一个用例只验一件事。

>

> 断言别只写不是空。要有状态码，要有响应里的关键字段，还要有 verify。verify 看 Service 被调了几次，参数是什么。

>

> 提交三样。测试类放在 src/test/java 下。一张分支覆盖的截图。一份短说明：测了什么，没测什么，为什么。

>

> 方法名要能读出意图。找不到就回 404，写进名字里。用例换顺序也能跑。不许改被测代码来换绿灯。

>

> 测试数据要说清来源，也要说清用完怎么处理。标准号是 JR/T 0363—2026。名称是《金融业应用软件测试数据设计指南》。2026 年 9 月 19 日起实施。这是推荐性行业标准，不是实验步骤。模型可以起草稿。交上

> 去的断言，必须你自己讲得清。

> **提问：**ItemController 里有个方法直接回了 null，没有回 404。测试怎么写，才算把问题测出来了？

> **预设回答：**

- 答"断言它返回 null"——那只是记下现状。错误行为被焊进测试。以后有人修好了，测试反而红。

- 答"断言状态码应该是 404，现在不是，所以测试红"——对。测试写的是应该怎样，不是现在怎样。红，正好指出缺陷在哪。

- 追问：测试红了，实验算失败吗？——不算。一个失败用例能指出缺陷位置，比一堆说不清的绿灯更有用。写进报告。

> **易错点 / 考点：**测试类放在 src/test/java，类名常用 ItemControllerTest。最常见的失分，是去测 Service 的业务，而不是控制器。另一类是断言太软，只判非空，或只判没抛异常。用 WebMvcTest 时，它不加载 Service。忘了 MockBean，启动会失败。为了让测试通过去改被测代码，这是纪律问题，也会影响评分。

> **过渡：**做的时候会卡住。三本书、两个站点，按用途拿。

【5.97】学习资源推荐（1 分钟）

> 实验一报错，二手博客和官方文档经常各说一套。先看哪一份，决定你要绕多少弯。

>

> 三本书不要一次全买。按"读完能解决什么"来排。

>

> 第一本，《程序开发人员测试指南：构建高质量的软件》。人民邮电出版社，2018 年 5 月。适合现在读。它讲的就是开发人员自己怎么写测试，和这学期的实验对得上。

>

> 第二本，《敏捷测试》。人民邮电出版社，2021 年 8 月。适合实验做完再读。它讲团队怎么组织测试，怎么在短迭代里守住质量。接的是这一章后半的持续测试。

>

> 第三本，《Java 微服务测试》。电子工业出版社，2019 年 7 月。这是进阶。契约、消费者驱动、测试金字塔在微服务里怎么变形，都在这本书的范围里。你要做微服务方向的课设，再细读。

>

> 两个站点比书更要常打开。junit.org/junit5 是 JUnit 5 的官方文档。实验里的注解、断言、扩展，以这里为准。docs.pact.io 是 Pact 的官方文档。契约那套流程和示例在上面。

>

> 三种材料对应三种需求。实验卡住，查官方文档。报错、注解、版本差异，都在那里。想把基础串起来，读书。书有章节顺序。想看行业把持续测试放在哪，再看白皮书和案例。别指望一本书解决所有问题。

>

> 顺序建议是先读官方文档，再读书。文档更新快。它默认你已经知道要解决什么。教材和这三本书，用来把问题问清楚。没有单元测试和契约的底子，微服务那本会读得很虚。

提问：只有一周，三本书和两个站点你怎么分配？

预设回答：

- 答"先看 JUnit 文档，边做实验边查"——最实用。带着失败的测试去查，记得住。

- 答"读第一本书的前几章"——可以。要配着手写。只读不写，记不住。

- 追问：为什么不建议先读微服务那本？——底子还没立住。契约和单元测试不清楚，那本书里的做法对不上你的代码。

易错点 / 考点：这一处多半是了解。开放题可能问你推荐一本和单元测试有关的书。书名、出版社、年份不必死背。两个官方站点要记得：junit.org/junit5，docs.pact.io。学习顺序比书目更值得留下：先文档，后书；先基础，后微服务。

过渡：材料放在课后。这一章如果只留三句，留下面这三句。

【5.98】感 谢 聆 听（1 分钟）

工具名单很容易带走，习惯却很容易留下。名单会过时。习惯会跟着你进第一个项目。

>

> 这一章从代码这一层怎么测走下来。评审看人能不能读懂。静态分析按规则扫。单元测试把函数钉住。Mock 解开你控制不了的依赖。覆盖率看语句和分支有没有走到。集成看拼上之后还认不认。契约把服务之间的格式写成文件。持续集成让这些每次提交都发生。

>

> 如果课后只留三句，留这三句。

>

> 测试写在编码过程里，不是等代码写完再补。评审、静态扫描、单元测试，都能在程序还没对外跑起来的时候发现问题。留到联调再改，牵涉的人更多。

>

> 覆盖率标出没人走过的语句和分支。它不宣布已经够了。一条用例把方法跑通，断言却很软，数字照样好看。够不够由人定。工具只把走过的地方标出来。

>

> 会写测试不算完。每次提交都跑，每次改动都验，才算养成。这件事交给流水线。靠想起来，总会有一周被进度挤掉。

>

> 下一章往系统层面走。看性能、安全这些跑起来之后才暴露的问题。那一层默认你已经会写这一章的测试。课下把实验 2 的 ItemController 测出来。答辩时请另一个人讲清：断言为什么是这组数据。讲不清，测试还停在能跑。写过，才算学过。

>

> 谢谢大家。

> **提问：**这一章里，哪一块最可能在你的第一个项目里马上用上？为什么？

> **预设回答：**

- 答"单元测试和 Mock"——最直接。进项目第一周就会写。

- 答"代码评审的检查表"——也很实用。评审是你最快能参加的团队活动。

- 答"契约测试"——如果进的是微服务团队，它很值钱。但要组里一起做，一个人写不起来。

- 收口：三种都成立，看你的岗位。如果只能先练一样，练单元测试。后面的实践都站在它上面。

> **易错点 / 考点：**这里不设名词考点。作用是把全章收住，指向下一章的性能和安全，并提醒课下提交实验 2 的测试代码。讲的时候别再展开新工具。

> **过渡：**本章到这里。下一次课看系统在压力下还对不对。

四、思政落点清单（共 13 处）

序号	所在页	落点主题
1	5.6	评审把"我的代码"变成"我们的代码"。被人指出问题，第一反应常常是解释。工程事实
2	5.9	认真是有成本的，所以要给它排时间。"帮我瞄一眼"之所以普遍，常常是因为计划里根本
3	5.28	第三方库这条链，给工科生上了一堂很现实的责任课。你引一个包，可能只是图省事。但这
4	5.33	这六项排下来，你会发现单元测试是件不显眼的活。没有界面、没有演示效果、写完了也没
5	5.34	这两个算式其实是成本意识的数学表达。工程上最贵的一句话是"先这样，回头再说"。因
6	5.36	覆盖率这个数字，最考验一个人的诚实。因为它太好看了。写一条把整个方法跑一遍的用例
7	5.39	驱动和桩，做的是为协作搭桥。被测单元本身没变，是周围的环境被补出来了。上游有人喂
8	5.48	注解看起来只是几个`@`。它要求的是把角色写明白。`@Test`让框架找得到。`
9	5.60	这句"够用就行"，拦的是一种习惯。工具越重，越像有水平。先问要解决的是什么。再问
10	5.64	verify 要求你说清三件事：调了几次，参数是什么，顺序对不对。判断一件事有没有
11	5.69	覆盖率容易被当成交差的凭证。数字过了线就收工，报告一贴，任务就算完。它更有用的用
12	5.82	契约测试最值得学的地方，是它把"承诺"变成了可验证的东西。消费者说我要这三个字段
13	5.95	这一章从代码评审讲到持续集成，中间隔了单元测试、Mock、覆盖率、契约测试。看起