# 《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。

---

## 第 5 章讲稿（5.33–5.44，v4 逐页重写）

> 每页先讲清页面上有什么，再展开道理。取消板书。口播 450–700 字，单句 ≤45 字。
>
> 5.43 作概览页处理：只讲分类与选型，每工具一句，细讲留给后面几页；概览页不报版本号。
>
> 5.44 作细讲页，末尾补一句已核实的 JUnit 6 发布记录。

---

## 第 5 章讲稿（5.33–5.44，v4 逐页重写）

> 每页先讲清页面上有什么，再展开道理。口播 450–700 字，单句 ≤45 字。

> 5.43 作概览页处理：只讲分类与选型，每工具一句，细讲留给后面几页；概览页不报版本号。
>
> 5.44 作细讲页，末尾补一句已核实的 JUnit 6 发布记录。

---

## 【5.33】单元测试内容（2.5 分钟）

> 左边一列编号，从 1 数到 6。右边一张流程图，四块图形。最左边绿框写着已知输入。中间橙框写着测试方法。右边绿框写着输出，后面还跟着过或不过。橙框上方一个蓝灰框写着 Class，和橙框之间上下两个箭头。最右边绿框上面还浮着一个橙色气泡，写着期望结果。

>

> 三个框连起来，一条用例就三件事。喂一个已知输入。跑一次被测方法。拿输出跟期望结果比，得出一句过还是不过。气泡挂在输出那一头，意思是比较的标准得先写下来。没有期望值的测试，跑完只能看它崩不崩，看不出它算得对不对。Class 和测试方法之间那两个箭头，是说测试方法长在测试类里。测试类和被测类不是一回事，这个区分后面写 JUnit 会一直用到。

>

> 左边六条，就是接着要走的六步。第一条问为何要测。第二条定目标和要测到什么程度。第三条讲驱动程序和桩程序。第四条是单元测试工具 JUnit 及其实践。第五条是 Mock 技术与框架，括号里举了 Mockito。第六条是测试覆盖率工具 JaCoCo。

>

> 六条按顺序读，是一条作业链。先说服自己值。再定测到什么程度算完。再解决上下游不在的问题。再动手写。再处理依赖。最后量一量够不够。六条里三条偏想，三条偏做。第二条和第六条要能背、能解释。第四条、第五条是实验要交代码的部分。第三条只有两个词，年年考方位。六步走完，你手上会有两样东西。一份能跑的测试代码，一份说明为什么这么测的报告。

**提问：**六条里，哪一条最容易被当成"跑完看一片绿"，其实绿色什么也说明不了？

> **预设回答：**

- 答"第六条覆盖率"——对。绿的是报告，不是正确性。跑了没断言，照样绿。

- 答"第二条目标和要求"——也对，而且更靠前。连测到什么程度算完都没定，后面全凭感觉。

- 答"第一条为什么要测"——看着务虚，其实是唯一能回答"这活值不值"的一条。

- 收口：第二条定标准，第六条量结果。中间四条都围着这两个转。

> **易错点 / 考点：**简答常问"单元测试包括哪些内容"，六条要数全。最容易丢的是第二条和第六条，它们听着像方法，不像内容。另一个错法：把驱动程序和桩程序当成两个软件名，它们是被测代码周围的两个角色。

> **过渡：**六条从第一条开始。先把"值不值"这件事算清楚，后面几条才有底气。

---

## 【5.34】为何要进行单元测试？（3 分钟）

> 左边四个方框要点，右边一张照片，底下两个红色算式。照片是一辆红色两厢车，被拆成几百个零件，平铺在一张白桌上。门板、座椅、轮毂、线束、仪表台，一样样摆开。

>

> 这张照片在说一件事：整车是零件堆出来的。装的时候看不出哪颗螺丝歪了，跑起来才知道。软件也一样。编译通过只说明零件都在，不说明装得对。

>

> 第一条约：编程与编译运行结束后，每 1000 行代码中大约残留 2 到 6 个缺陷。这不是新手代码，是行业常态。五万行的系统，天生带着一两百个缺陷。

>

> 第二条约：寻找与修改程序错误的代价，占总体开发投资的三成到六成。开发预算里有一大半，花在找错和改错上。测试不是额外开支，它就是开发成本的大头。区别只在你把它花在便宜的环节，还是贵的环节。

>

> 第三条约：尽早发现错误，成本越低。下面挂了两小条。发现问题比较容易。修正问题更容易。同一个缺陷，单元阶段改一行代码。系统测试阶段要重新部署、重造数据、重跑回归。上线以后要发版、要道歉，可能还要赔钱。

>

> 第四条约：单元质量是系统质量的基石，单元测试可以比系统测试测得更彻底。系统测试受场景限制，方法内部那些分支组合你摆不出来。单元测试可以一条分支一条分支地喂数据。

>

> 底下两个红算式，是这一条的算术版本。0.8 的 30 次方，约等于 0.00123。0.99 的 30 次方，等于 0.740。
>
> 三十个单元串成一条链，每个八成把握，整机一次跑通只有千分之一。每个九成九，整机也才七成四。

> **提问：**每个单元都做到 99%，整机只有 74%。那"每个单元 99%"这个目标，算不算高？

> **预设回答：**

- 答"99% 已经很高了"——单看一个单元确实高。串起来掉到七成四，说明高不高要看整机要求。

- 答"不够，得靠集成测试补"——能兜一段，兜不住组合爆炸。正确做法是两头一起做。

- 追问：那 100% 呢？——100% 覆盖不等于 100% 正确，讲到覆盖率那一页再说。

- 收口：目标不是数字好看，是风险可控。先按整机要求倒推单元目标。

> **易错点 / 考点：**两个数字必背：每 1000 行残留 2 到 6 个缺陷，找错改错占总投入三到六成。两个算式要会解释：0.8 的 30 次方约 0.00123，0.99 的 30 次方约 0.740。易错点：把 0.8 的 30 次方说成"缺陷率"。它算的是三十个环节全过的联合概率，不是某个单元的缺陷比例。另一个易错：把 0.8 写成 0.83。

> **过渡：**值的账算完了。可如果单元测试只干"抓缺陷"这一件事，凭什么每个函数都要配一条用例？下面七条价值回答它。

---

## 【5.35】单元测试的其它价值（3 分钟）

> 屏幕上一列方框，一共七条，从上排到下，没有配图。每条都是四五个字的短语，加一句解释。前两条名字没加粗，后面五条加粗了。加粗这五条，是出报告时希望你能点到的。

>

> 第一条，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，单元测试就是基础。翻成团队里的话：没有测试，没人敢接别人的代码。代码被覆盖、回归缺陷、组与组之间推责任，都从这儿长出来。

>

> 第二条，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。这句底下藏着一个手法。你要给一个方法写测试，就得把它单独拎出来跑。一拎，三种病现形。方法有太多参数。巨型方法。依赖过多。为了测得动，你会去拆。拆完，设计就好了。

>

> 第三条，支持变化。任何人修改系统的一部分，就必须了解别的哪些部分需要运行验证。加粗那句说，单元测试提供了所需要的安全网。没有这张网，改代码就是赌自己的记性。

>

> 第四条，避免回归缺陷。系统集成为一个庞大系统以后，靠系统测试穷尽全部回归，几乎不可能。单元测试几秒就能跑到每个分支。

>

> 第五条，保证工作的平稳步调。代码编写跟单元测试配合，很少引起意外，也很少出现最后一分钟问题。

>

> 第六条，节省测试时间。边界值验证、输入验证、程序主逻辑路径调用，用单元测试做最简单、最快、最廉价。

>

> 第七条，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。跑不通就红，这份说明书不会过期。

**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。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"，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。先用测试把现有行为钉住，再改。

- 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叫重构，叫赌。

- 收口：三条是一条链。读懂了才写得出测试，有测试才敢重构，重构干净了才好加功能。

> **易错点 / 考点：**多选题问"单元测试的价值"，七条尽量记全。最容易漏的是"引导更好的设计"和"规范行为和记录代码"，因为听着不像测试的活。判断题"单元测试只能发现功能缺陷"是错的。口诀：扩展、设计、变化、回归、步调、省时、文档。

> **过渡：**七条价值讲起来容易空，落到目标就具体了。下面四条目标，第四条那个百分比最容易被拿来交差。

## 【5.36】单元测试的目标（3 分钟）

> 四条方框，从功能说到覆盖率，一条比一条抽象。第四条那个百分比，最容易被拿来交差。

>

> 第一条，单元模块被正确实现。括号里写明，主要指编码，包括功能、性能、安全性等。后面紧跟一句：但一般主要介绍单元功能测试。这句是边界，也是分寸。功能是主战场。性能受调用方式影响，安全要看整个数据流，单元层面说不完整。说不完整，不等于不用看。一个正则写错导致回溯爆炸，一句查询没参数化，都是单元层就埋下的。

>

> 第二条，参数输入是否正确传递、是否得到保护，括号里写着容错，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，有没有被挡住。线上大量事故，根子不在算错，在根本没检查输入。

>

> 第三条，内部数据能否保持完整性。后面跟了三个具体项。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，影响最低。这三项是单元测试独有的地盘。系统测试从外面只看得见输入输出，看不见这个对象到底放没放掉。

>

> 第四条，代码行、分支覆盖或判定条件覆盖达到要求，比如高于八成或九成五。行覆盖看这一行跑到没有。分支覆盖看判定为真为假两条路都走过没有。再往上那一档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它。八成和九成五不是同一条及格线，是两个完整性等级。

>

> 四条合起来才叫测完。覆盖率是必要不充分条件，缺了它不行，只有它也不够。

> **提问：**一个模块行覆盖 100%、分支覆盖 100%，能不能说它测充分了？给个反例。

> **预设回答：**

- 答"可以，都 100% 了"——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，可能一句断言都没写"——对。跑到了没断言的代码，等于走过场。

- 答"不能，输入可能全用正常值"——更狠。异常分支靠抛异常顺便覆盖过去，报告好看，缺陷还在。

- 收口：覆盖率要配着断言密度和输入设计一起看，三样齐了才算充分。

> **易错点 / 考点：**选择题爱问"单元测试的目标包括哪些"，四条都要选，包括"内部数据能否保持完整性"这条抽象表述。高频陷阱是把判定条件覆盖说成条件组合覆盖。它比判定条件覆盖强，比条件组合覆盖弱，核心是每个条件能独立改变判定结果。

> **过渡：**目标讲完了，马上动手。下面这段只有十一行，四个人里有三个第一次看不出来。

---

## 【5.37】一个小练习（2 分钟）

> 上面一段黑字是需求，后面两句红字是两个问句。一句问错误在哪里。一句问通过怎样的测试数据可以发现。下面一个米黄色框，装的是一段 Java 代码，十一行，从包名到最后一个右括号。

>

> 需求先读一遍。实现两个正整数或零的加法。只要有一个数小于或等于 0，就返回 0。注意"或零"和"小于或等于 0"，两处都带等号。

>

> 代码再读一遍。第一行是包名。第二行声明一个公开类，类名是计算器。第三行是方法声明，两个整型参数，一个叫 num1，一个叫 num2。第四行是判定，两个条件用与号连着。左边写 num1 大于等于 0，带等号。右边写 num2 大于 0，不带等号。成立就返回两个数相加。否则走 else，返回 0。

>

> 判定里两个条件不对称。需求说零是合法输入。所以 num2 传 0，判定为假，程序直接返回 0。5 加 0 得 5，它给你 0。错就错在右边这一个等号上。

>

> 这个错为什么难发现？平时测函数都用正常值。2 加 3 得 5。10 加 20 得 30。-1 加 5 得 0。三条都对，全绿，代码就提交了。要暴露它得用边界值。5 加 0，期望 5，实际 0。零不是随便一个数，零是两个合法区间的交界。

>

> 再看覆盖。让判定真假两条都跑到，两组数据就够。5 加 3 走真，-1 加 5 走假。分支覆盖看着 100%，这个缺陷还活着。因为 num2 等于 0 这一格，被 -1 加 5 那个假代表了。要补 5 加 0，让 num1 取真、num2 取假，条件覆盖才齐。

>

> 修法不是把右边那个大于号改成大于等于就完事。先在方法开头挡掉负数，再放心相加。边界写在开头，比藏在复合判定里安全。

**提问：** 5 加 0 这条用例，同时完成了哪种覆盖？还差哪种？

**预设回答：**

- 答"完成了分支覆盖"——只对一半。它让判定取到假，取真还得靠 5 加 3，两条合起来才叫分支覆盖。

- 答"完成了条件覆盖"——对。num1 大于等于 0 取真，num2 大于 0 取假，两个条件都取到了真假中的一个。

- 答"没完成路径组合覆盖"——也对。四种条件组合只走到一部分，要全覆盖得再补 0 加 0、0 加 3 这类数据。

- 收口：这条用例的价值，是它同时踩在边界和条件上。边界值天生比中间值能打。

**易错点 / 考点：** 这道题几乎年年考。两个坑：答"没发现错误"，或者答"num1 要写成大于 0"，把对称当正确，方向反了。答题模板固定三句：错误位置在 num2 大于 0 的边界，暴露数据是 num2 等于 0，预期 5 实际 0。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。真实函数上面有人调，下面它还调别人。假邻居怎么搭，看下面两个词。

---

## 【5.38】驱动程序与桩程序（2 分钟）

深蓝横幅上七个字，驱动程序与桩程序。横幅下面这张小图，只有两个灰框和四条箭头，箭头都带编号。左边框写"被测对象"。右边框写"合作者"。箭头从左边进场，绕一圈，再从左边出去。

>

> 编号 1，调用，从外面指向被测对象。这是有人来喊它干活。编号 2，调用，从被测对象指向合作者。这是它自己再去喊别人。编号 3，数值，从合作者回到被测对象。别人把结果交回来。编号 4，数值，从被测对

> 象回到左边那一头。它把结果交出去。

>

> 四条箭头，就是一次单元测试的完整走位。可两个框里，只有一个是真的。被测对象是真的，要测它。合作者是假的，测试环境里根本没有那个数据库、那个支付接口、那个消息队列。

>

> 假的那个框，按位置分两个名字。站在左边、负责发起调用的假角色，叫驱动程序。站在右边、被调用的假角色，叫桩程序。驱动是上游的替身，桩是下游的替身。一个把数据喂进来，一个把结果顶出去。

>

> 这两个词的方位，是这一节最爱设陷阱的地方。判断只看一件事：谁调用谁。它调用被测对象，它是驱动。被测对象调用它，它是桩。你写的那个测试方法，本身就是驱动。桩得你自己写，因为只有你知道下游要返回什么。

>

> 还有一句别忘了。两个角色都不在被测代码里，都在测试代码里。笔记上可以左边写驱动，右边写桩。课程设计里要交的，除了被测类，还有这两个假邻居。

> **提问：**你写了一个测试方法，里面新建一个被测类、传两个参数、断言返回值。哪部分是驱动，哪部分是桩？

> **预设回答：**

- 答"整个测试方法就是驱动"——对。JUnit 的运行器加上你的方法，一起构成驱动。

- 答"这里没有桩，因为这个类没有下游依赖"——正确。桩不是必需品，只有真调用了外部协作者才要。

- 答"传进去的两个参数就是桩"——概念串了。参数是数据，桩是一个实现了相同接口的替身对象。

- 收口：看被测方法体里有没有别的对象被调用。没有，就不用桩。

> **易错点 / 考点：**名词方位必考。三步判断法：看它调用谁。它调用被测单元，是驱动；被测单元调用它，是桩。第二个高频错答是把桩和 Mock 当同义词，两者机制不同，后面专门讲。第三个错法是把驱动和桩当成工具名。

> **过渡：**光说方位还是抽象。下面把这两个词塞进一段真代码，再配一张类图。

---

## 【5.39】驱动程序与桩程序（2 分钟）

左半边是一段带行号的代码，从第 1 行排到第 31 行，里面装了两个程序。右半边是一张类图，标题写着实例化被测类。

>

先看左边。红虚线从第 4 行拉出来，标着驱动。那一行是主函数声明。主函数负责起头、准备变量、循环调用、最后返回 0。它就是驱动。第 6 到第 9 行声明了三个变量，一个双精度数、一个整数、一个字符。第 10 行起一个循环。第 18 行打印"再来一次"，接着问是 y 还是 n。第 22 行的循环条件看那个字符是不是小写 y 或大写 Y。这套东西就是典型的驱动。

>

橙虚线从第 11 行拉出来，标着被测函数。那一行调用了一个取输入的函数，传进去两个变量。被测的就是它，输入两个数，把结果带回来。

>

第 25 行有个注释，写着第二个程序。下面另起一个函数，函数据说就是被测函数本身，两个引用参数。它在第 28 行写了一个赋值，右边算的是单价。

>

单价那个函数在第 31 行，蓝色高亮。函数体就一句，返回 10.00。红虚线标着桩。这就是桩。被测函数要算钱，测试时不许它真去查价，写死 10 块。

>

再看右边类图。紫色框是被测类，在左。绿色框是测试夹具类，在右。中间四条箭头。两条绿箭头指向左边，写调用方法，测试类去调被测类。两条紫箭头指向右边，写检查结果，结果回来给测试类看。绿色框下面一个向下箭头，通到黄色框报告。

>

这张图说的是两个角色在代码里的位置。测试类既是驱动，也是裁判。喊人干活，收结果，写报告。被测类自己不知道在被测。

>

> 为什么非要这么搭？三个理由。隔离，失败时能立刻确定问题在它自己身上。可控，真实下游今天成功明天超时，你的测试就时红时绿。速度，连库一条几百毫秒，用桩是微秒级。

> **提问：**被测方法里调了一次取当前时间的系统调用。要不要为它做桩？

> **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、闰秒那天莫名其妙地红，还没人查得出原因。

- 答"要，因为时间不可重复"——正确。做法是把取时间抽成一个可注入的时钟，测试时注入固定值。

- 追问：这又回到第二条价值？——对。为了可测，你把时间依赖抽出来了，代码反而更干净。

- 收口：跨进程、不可重复、有副作用，三条中一条就别硬连真的。

> **易错点 / 考点：**图上的箭头方向是考点：驱动调被测单元，被测单元调桩，一条链。画成"桩调被测单元调驱动"就是方位全反。另常考"驱动和桩由谁写"：驱动多由测试框架提供，也可手写；桩必须测试人员自己写。还有一条：第 31 行那句返回 10.00 是硬编码，下一页就说它哪里不好。

> **过渡：**写死 10 块能用一次。第二次要 20 块，你就得再写一个类。这个毛病，下面这段 C 井号代码给了改法。

---

## 【5.40】桩程序的灵活性（2 分钟）

> 左边一大块米黄色代码，右边一段话，两个词标成红色。

>

> 代码从上往下读。第一行声明一个类，类名直译就是参数化的桩，冒号后面是它实现的协作接口。里面一个私有整数字段。构造函数收一个整数，把值存进字段。下面一个公开方法，名字叫算完返回，返回整数。方法体两件事。字段值小于 10，抛一个非法操作异常。否则返回字段值。

>

> 关键在构造函数那个参数上。这个桩返回什么，由新建对象时传进去的数决定。传 100，返回 100。传 0，返回 0。传 5，抛异常。一个类，三种行为。

>

> 右边那段话，红字标着"硬编码数值"，还有一句"最简单，但也是最不聪明的做法"。原话是：不会像上一页返回一个硬编码数值。让一个桩返回单个数值，是最简单、也最不聪明的做法。理由是别的测试迟早都要一个不同的数值。

>

> 为什么不聪明，算一下就明白。第一个用例要 100，你写个返回 100 的桩。第二个要 0，再写一个类。第三个要抛异常，第三个类。三个用例三个桩，代码量比被测代码还多，大半还是重复的。接口一改，三处一起改。

>

> 这段代码换成 Java 写也成立，思路一样：把要返回的值从方法体搬到构造参数里。参数化以后，变化的部分从类的个数，变成了构造参数。这就是灵活性三个字的意思。桩的价值不在能返回，在能按需返回。写桩之前先问一句：下一个用例会不会要一个不同的值？会，就别写死。

**提问：**桩里能不能写业务判断，比如"金额大于 1000 就打九折"？

**预设回答：**

- 答"不能，桩越简单越好"——方向对，但太严。桩里可以有逻辑，前提是这段逻辑只为测试服务。

- 答"能，只要能造出想要的场景"——可以接受，但桩一复杂，你就在测桩，不在测被测代码。

- 追问：判断标准是什么？——看这段逻辑会不会影响断言的正确性。会，就说明它得搬到构造参数里。

- 收口：这段代码里"小于 10 就抛异常"，就是故意的假业务，只为造失败路径。

**易错点 / 考点：**常考"硬编码桩的问题"，标准答案是用例一多就要写多个桩类、复用性差、维护成本高。另一个易错：把参数化桩和 Mock 框架的配置混成一件事。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。

**过渡：**为什么测、测什么、怎么搭台子，三条都齐了。工具这一节，从一份名单开始对号。

---

## 【5.41】单元测试工具及其应用（3 分钟）

深蓝横幅上七个字，单元测试工具及其应用。横幅下面并排两个东西。一个红绿两色的圆形数字 5，一个灰色的 JUnit 5 字样。这是 JUnit 5 的官方标志，也是这一节要落的那个点。

>

> 前面几页答的都是"想清楚"。从这里开始要动手。为什么单元测试离不开工具，先算笔账。一个班三十人，每人写五个类，每个类五个方法，七百五十个方法。手工点，点不过来。系统测试还能靠人点界面，也能靠脚本点。单元测试对着的是函数，没有界面可点。数量一上去，人就跟不上。

>

> 还有一个更硬的理由。跑一次单元测试要干四件事：找出用例、挨个跑起来、抓住抛出来的异常、把结果汇总成报告。这四件事自己凑，凑出来的仍是一套框架，只是自己写的、没人维护。所以在各类测试里，单元测试最离不开工具。

>

> 这一节的路子，是从名单走到分类。先看一张网页截图上的长表，知道这个生态有多密。再把最常听到的十个名字归成四类：框架、用例自动生成、Mock、覆盖率。最后落到 Java，看 JUnit 4 那张类图。

>

> 类图里四个角色要对上。运行器负责跑。测试套件负责收集。测试用例负责执行和断言。结果对象负责存。四个角色谁管什么，下一节逐个指。四个对不上，那张类图就看成一堆框。

>

> 实验课用 JUnit 5。这一节先讲 JUnit 4 的骨架。理由有两条。存量代码和面试题还在用它。JUnit 5 本身也是从 4 上拆出来的。先懂 4 里谁管什么，才看得懂 5 为什么分成三层。

**提问：**如果团队只许你引入一个单元测试相关的工具，你选哪一个？

> **预设回答：**

- 答"覆盖率工具"——想先看现状，可以理解。没有用例可跑，覆盖率会是 0，看不成东西。

- 答"测试框架，比如 JUnit"——多数时候就选它。另外三类都挂在它上面。

- 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，错也一起焊进去。

- 收口：先有框架，再谈替身和覆盖率。顺序反了，工具都装不上。

> **易错点 / 考点：**考点是工具分类，不是装哪个。用例自动生成那一类不是 Mock，也不是覆盖率。覆盖率库靠虚拟机的探针插桩，自己不是测试框架。这些混答是最高频的失分点。

> **过渡：**名字太多，先从这张表的出处看起。一张维基百科的截图，正好把"能力可以横着比"这件事摆出来。

---

## 【5.42】单元测试工具（2 分钟）

> 一张网页截图。标题是 C 加加，后面跟一个方括号的编辑链接，说明这是维基百科里 C 加加那一节。上面一行蓝色链接，是单元测试框架列表那个页面的地址。下面一张很宽的对照表，绿格子写有，红格子写无，一眼看过去红绿相间。右下角页码 42。

>

> 先数列。表头十三列。第一列是工具名。第二列是许可证，能看见通用公共许可、BSD、MIT、Boost、专有这些字样。中间九列全是能力格，横着排一排小字。倒数第二列是维基的引用编号。最后一列是备注，一大段英文说明。

>

> 怎么读？横着读一行，是一个工具的能力清单。竖着读一列，是这项能力有几家支持。第一列往下能看到十来行工具名。Aeryn、Bandit、Boost、Catch2、doctest 都在里面。还有商业产品 Cantata。一行里红多绿少，说明这个工具短板明显。第一行那个工具，能力格几乎全红。

>

> 竖着读也有用。挑一列能力格从上往下数。绿色越多，说明这项能力支持越广。备注列写的是擅长什么、限制在哪。选型的时候，备注比打勾更有用。

>

> 有两个格子值得点一下。Boost 那一行，生成器那格写的不是干脆的有。它写的是"另配一个库"，意思是另装一个库才有这能力。Bandit 那一行，能力格第一列是红的无，括号里写了另一套写法。能力有，名字不叫那个。

>

> 这张表不要求背。它告诉你三件事。第一，框架很多。第二，能力可以横着比。第三，许可证会影响能不能商用。做选型时先按许可证筛，再比能力。

> **提问：**这张表里许可证那一列，为什么值得看？直接挑能力全的不好吗？

> **预设回答：**

- 答"能力全就行"——不行。专有那行是商业授权，学生项目能试用，公司上线要买。

- 答"要看能不能商用"——对。许可证决定这条工具能不能进产品，不只决定它好不好用。

- 追问：那学生做课设呢？——开源那几档随便挑。但报告里写一句许可证，说明你看过。

- 收口：先看许可证过不过关，再看能力够不够。顺序反了容易白折腾。

> **易错点 / 考点：**这一页不考名单。考的是会不会读对照表：横比能力、竖比工具、许可证单独看。简答常问"选择单元测试框架看哪几个维度"，答生态、集成开发环境与构建工具支持、覆盖率报告、团队熟悉度、许可证。

> **过渡：**表里几十个名字，课上只留最常听到的。下面这十个，按四类归好。

---

## 【5.43】最常用的单元测试工具（3 分钟）

> 顶上一句说明。它说的是 Java、C 加加和 Python 的单元测试里，受欢迎的工具分四类。单元测试框架一类。两个智能化的用例自动生成工具一类。Mock 工具一类。代码覆盖率工具一类。这一句就是整页的架子。

>

> 左边一列编号，① 到 ⑦，七个名字。第一组是 JUnit 和 TestNG。第二组是 GoogleTest。第三组是 pytest。第四组是 unittest。第五组是 Coverage.py。第六组是 EvoSuite。第七组是 Diffblue Cover。右边另起一套编号，三个。第一个是 JMockit。第二个是 JaCoCo。第三个是 gcov、lcov 和 gcovr 三个采集与报告工具。右下角一个二维码，下面一行字，详见公众号文章。

>

> 两套编号为什么分开？因为归类不同。左列混着三类。前五个是框架。中间两个是用例自动生成。Coverage.py 是覆盖率。右列补的是 Mock 和覆盖率那一头。看名单别按行看，按类别看，才不会把自动生成工具当成 Mock。

>

> 四类各自管什么，一句话一条。框架管组织和执行，用例靠它跑起来，断言和报告也归它。自动生成管速度，替你写出一批用例。Mock 管依赖，被测代码要调别人时拿它顶上。覆盖率管度量，回答哪一行被跑到、哪

> 条分支没走到。

>

> 选型怎么定？先看这门语言的生态里，那一个是现成的。再看集成开发环境和构建工具认不认。再看覆盖率报告出不出得来。最后看团队熟不熟。前三项是硬门槛，最后一项是成本。四项都过，名字就不用纠结。

>

> 名单先摆在这儿。接着的几页挑 JUnit 细讲，把一条用例怎么跑起来拆开。再往后是 Mockito 和 JaCoCo，各占一节。这一页只要记住四类，别把名字和类别对错。

> **提问：**四类里如果只能先装一类，你装哪一类？为什么不是别的三类？

> **预设回答：**

- 答"框架，别的都挂在它上面"——对。没有框架就没有用例，替身和覆盖率都没有东西可跑。

- 答"覆盖率，先看现状"——听上去务实。没有用例，报告是 0，看不出任何东西。

- 答"自动生成，一步到位"——顺序反了。没有框架接住，生成的用例跑不起来。

- 收口：顺序是框架、替身、覆盖率、生成。这一页只是把四类摆开，细节后面各讲一节。

> **易错点 / 考点：**最爱考归类。EvoSuite 和 Diffblue Cover 是用例自动生成，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。另有一个拼写坑：写报告时 SonarLint、SonarQube 不要写成 Solar。

> **过渡：**名单对上了，回到 Java。实验课用 JUnit 5，可先得看懂 JUnit 4 那张类图，四个角色才立得住。

---

## 【5.44】JUnit 4 工作原理（2 分钟）

> 一张统一建模语言的类图，两大块。上面大的一块，边框左上角标着框架包名。下面窄的一条，左上角标着运行器包名。两块之间的连线在左下角那一块上。

>

> 上面那块从右上角看起。一个矩形框，上面一行标着接口，框里写 Test。它挂两个方法，run 和 countTestCases。后者回答的是这一套里有几条用例。

>

> 它左边是 TestResult，一个长方形框，站在中间。它负责存结果。

>

> 它下面靠左是 TestCase。框里一栏字段 fName，一栏方法。方法有四个：setUp、runTest、tearDown、run。它和 Test 之间是虚线，表示实现关系。也就是说，用例类把接口里那两个方法都写了出来。框里还有两个适配类，一个叫适配器，一个叫门面。

>

> 右边是 TestSuite。字段 fName，方法两个：run 和 addTest。它和 Test 之间是空心菱形加连线，聚合关系。意思是套件里装着用例，也装着别的套件。所以套件可以嵌套，按包跑、按类跑都靠它。

>

> 中间上方有 Assert，立在用例类头顶，断言就归它。它下面空着，没有方法，就一层壳。左上角两个异常类，上面那个继承自下面那个，一个空心三角指上去。失败就先落到这两个类上。左边还有一个监听接口，挂四个方法，两个记错误，两个标记开始和结束。

>

> 下面那块运行器包里，BaseTestRunner 有一条实心三角箭头，指向上面的监听接口。旁边还有一个运行器监听接口和一个版本类。前者听运行器自己的事件，后者记版本号。

>

> 整张图读成一句话。一个运行器跑一个套件，套件里装一个或多个用例。结果由结果类收，再由运行器报出去。

> **提问：**为什么每个测试方法都要新建一次测试类实例，不能让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，只是会引入顺序依赖。

- 追问：某个对象很贵，每个方法都要用怎么办？——提到类级别那一层，用静态资源，或者交给依赖注入。

- 收口：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 与 JUnit 5 的注解对应。Before 对 BeforeEach，After 对 AfterEach，BeforeClass 对 BeforeAll，AfterClass 对 AfterAll，Ignore 对 Disabled。另一考点是四个角色的职责：用例类负责跑和断言，套件类负责收集，结果类负责存，运行器负责报。据 GitHub 官方仓库 2026 年 11 月的发布记录，JUnit 已经发布 JUnit 6，最新是 r6.1.3，2026 年 8 月 7 日。

> **过渡：**四个角色已经就位，但报告上失败和错误分成两列。这两列为什么不能混，下一节接着拆。

# 第 5 章讲稿（5.45–5.56，v4）

> 逐页重写。每页先落到图上的实体：几个方框、几行代码、表格几列。取消板书。口播四拍：起 → 推 → 收。

---

### 【5.45】JUnit 工作原理 – 续（3 分钟）

> 图的上半部是一个算式，四个方框、两个加号、一个等号。方框里是类名，不是流程框。左边三个加数：TestCase、TestSuite、BaseTestRunner。等号右边是 TestResult。

>

> 一个 TestRunner 运行一个 TestSuite。套件由一个或多个 TestCase 组成。括号里还补了一句，套件也能由别的套件组成。结果由 TestResult 收集。再由 TestRunner 报出来。

>

> 先读等号。右边是产出，只有一项。左边三项都是投进去的。

>

> TestCase 是最小的一份卷子。一个 @Test 方法就是一道题。它负责跑，也负责判。判靠断言，下一页再说。

>

> TestSuite 是装卷子的袋子。袋子里能装袋子。所以按包、按类、按标签都能跑。

>

> BaseTestRunner 是方框上印的全名。正文写的是 TestRunner。Base 只是那个基类，真跑的是它的子类。答题写 TestRunner 不算错。它干两件事：启动流程，报出结果。控制台那串点、IDE 左边那棵树，都出在

> 它手上。

>

> 收集和报告要分开。TestResult 只往里记。报告是 TestRunner 的活。问谁收集，答 TestResult。问谁显示，答 TestRunner。

>

> 记进去的结论有三类：成功、失败、错误。失败是断言没成立，程序跑完了，值不对。错误是跑到一半抛了异常，没走到断言。报告上 Failures 和 Errors 分两列，就是这两类。看到 Errors，先看异常栈。别急着改业务代码。

>

> JUnit 5 把这几个名字换掉了。发现交给 TestEngine。结构变成一棵 TestDescriptor 树。启动交给 Launcher。四件事还在：组织、执行、收集、报告。

**提问：**报告上写着 `Failures: 0, Errors: 3`。你第一件事做什么？直接改业务代码，还是先看栈？

**预设回答：**

- 答"三个用例都失败了，去看业务逻辑"——粗。Errors 说明没走到断言，先分清异常是谁抛的。

- 答"先看异常栈，区分是测试代码抛的还是被测代码抛的"——对。桩没配对、参数给错、依赖没注入，异常先在测试这一侧冒出来。

- 答"Errors 一定是测试自己写错了"——也不对。被测方法里的空指针同样报 Error。看栈顶的类名归谁。

- 追问：那 Failures 那列呢？——断言判过了，值不对，方向更清楚。

**易错点 / 考点：**failure 和 error 必须分开。failure 是断言不成立，程序跑完。error 是异常打断执行。第二个考点：套件能装套件，靠的是单个用例和套件实现同一个 Test 接口。第三个错答是"TestResult 负责打印报告"——收集归 TestResult，显示归 TestRunner。

**过渡：**四个类在这个算式里接成一条链。到了 JUnit 5，这条链被拆成三块子项目，下一页看拆法。

---

## 【5.46】JUnit 5 架构（3 分钟）

> 右半边那张图从上往下有五层。最上面一条灰带，写着 Your tests。往下三个带尖角的六边形。橙色那个写 junit-4.12。绿色那个写 junit-jupiter-api。粉色那个写 testing-api。再往下一排三个长条，是引擎。再往下是一个很宽的菱形，写 platform-engine。再往下一条蓝条，是 launcher。最底灰带列着 IDE 和构建工具。左半边三段带勾的文字，一段对一个子项目。

>

> 第一段讲 Platform。它提供跑测试的底子，还有引擎接口。它自带控制台启动器，命令行就能拉起来。构建插件也接在这儿，所以 mvn test 能跑，也能进流水线。

>

> 第二段讲 Jupiter。写测试的新编程模型，加扩展模型。@Test、@BeforeEach、assertAll 都从这儿来。它自带引擎，在平台上跑自己的测试。

>

> 第三段讲 Vintage。它也给一个引擎，跑 JUnit 3 和 4 的老用例。它是可选依赖，新项目可以不引。

>

> 再看箭头。橙色和绿色各有一条，指向自己那边的引擎。旁边一行小字，意思是发现并运行匹配的测试。三个引擎都指向蓝色菱形，旁边写 Implements。菱形下方两行小字，说通过 ServiceLoader 找实现，再说编排执行。平台只认引擎这层接口。粉色那列标了一句，说那是自己编的例子。自己写一个引擎照样能挂上来。

>

> 这句要记准。JUnit 5 是三个子项目一起发布，不是三个版本。旧用例只引 Platform 加 Jupiter，不会报红。它只是没被发现。用例数对不上，就是没跑。据 GitHub 官方仓库 2026 年 11 月的发布记录，JUnit 已到 6.1.3。仓库改名叫 junit-framework。这三块结构没变。

> **提问：**项目里有 JUnit 4 的老测试，也有 JUnit 5 的新测试。只引 Platform 加 Jupiter，会发生什么？

> **预设回答：**

- 答"老用例会跑失败"——不是。它压根没被发现，报告里少一截。

- 答"三块都要引，Vintage 负责老用例"——对。Jupiter 跑新的，Vintage 跑旧的，Platform 负责调度。

- 答"没被发现也不算大事"——麻烦就在这儿。红是响的，静悄悄少掉的那些最容易被忽略。

- 追问：怎么确认老用例真在跑？——数报告里的用例总数，对一下类名。

> **易错点 / 考点：**三块的分工必考。三个高频错法：把 JUnit 5 说成就是 Jupiter；把 Vintage 只说成"旧包"，说不出它提供引擎；把 Platform 说成 IDE 插件。口诀：平台管跑，Jupiter 管写，Vintage 管旧用例还能跑。

> **过渡：**架构是给人看的。每天都坐在电脑前写代码，手边只有三样东西。

---

## 【5.47】需要了解 JUnit 的关键组件（2.5 分钟）

> 这里没有图，也没有表，就是三组文字，竖着排。每组一个加粗小标题，下面跟一句解释。第一组 Assert 类。第二组 Annotation，注释。第三组 Runner，运行器。三组是并列的，没有先后。

>

> 第一组说，Assert 类是 JUnit 里很重要的类。它判断返回的实际结果和预期结果是否相符。前面造数据、调方法，都在凑材料。断言才决定过还是不过。一条测试没有断言，它就不是测试。它只是一段能跑通的演示。框架没有"什么叫错"的出口，自然给你绿灯。

>

> 第二组说，JUnit 用 @ 引导，标出测试类里各个方法的功能。JUnit 3 的时候，方法名要以 test 开头才会被执行。有了 @Test，名字就能写意图。should_return_zero 比 testAdd2 有用。代价是你不标，框架就找不到它。名字取得再好也不跑。

>

> 第三组说，JUnit 通过运行器显示测试执行结果。这句里有两个动作：跑，和显示。JUnit 5 里跑这件事改由平台的引擎做，概念没变。总得有个东西，把注解标出来的方法找出来，再驱动它们。

>

> 三组连成一圈。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。查"测试没生效"也按这个顺序。先看注解加了没有，是不是被标成禁用。再看运行器扫到没有，看包路径和命名过滤。最后看断言写了没有。

>

> 顺序反过来也成立。报告全绿，先别高兴。打开方法体，数里面有几条 assert。零条的话，这片绿什么都没证明。

> **提问：**一条测试加了 `@Test`，能跑，报告是绿的，但里面一个断言都没有。你怎么在一分钟内发现？

> **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。

- 答"看覆盖率"——不够。空壳也能把行覆盖跑满。

- 追问：空壳测试为什么危险？——它给的是"已验证"的错觉。改动之后该红的地方不会红。

> **易错点 / 考点：**三个组件各干什么。Assert 判断实际与预期是否相符。Annotation 以 @ 引导，标出方法角色。Runner 负责运行并显示结果。常见错答是把 Assert 说成"抓异常的工具"。抓异常有另外的写法，Assert 只做判断。

> **过渡：**三样里，注解写得最多，也最容易写错。

---

## 【5.48】注释 Annotation（2.5 分钟）

> 左边两行文字。第一行说，注解把逻辑结构和功能讲清楚。第二行是一长串名字，全以 @ 开头，从 @Test 排到 @TempDir。右边是一段 36 行的代码，行号在左边缘。类名叫 StandardTests。第 1 到 9 行是 import。第 12 行挂 @BeforeAll。第 16 行挂 @BeforeEach。第 21 到 25 行是一个测试方法，头上压着 @Disabled。第 28 行挂 @AfterEach。第 33 行挂 @AfterAll。第 36 行收右花括号。

>

> 左边那串名字别硬背，按用途切四组。

>

> 第一组管用例。@Test 是底子。@ParameterizedTest 让一条用例跑多组数据，省掉一大段复制粘贴。它要配数据来源。@ValueSource、@CsvSource、@MethodSource，至少给一个。

>

> 第二组管生命周期。@BeforeEach 和 @AfterEach，每个方法前后各一次。@BeforeAll 和 @AfterAll，整个类只一次。看右边代码，第 13 行和第 34 行都写了 static。默认每个方法一个新实例。静态方法才活得比实例长。

>

> 第三组管组织和展示。@DisplayName 起一句人话。@Nested 做分组。@Tag 打标签。@TestClassOrder 和 @TestMethodOrder 管顺序。顺序默认没有。靠上一条喂下一条的写法，换个环境就会红。

>

> 第四组管控制和扩展。@Disabled 先停用。@Timeout 超时就判失败。@TempDir 建临时目录，测完自动清。

>

> 还有两处细节。左边把 @BeforeEach 印成了小写的 each，代码里是驼峰。第 22 行那个 @Disabled 带了一句说明，写清了为什么停用。

**提问：**`@BeforeEach` 和 `@BeforeAll` 各放什么？`@BeforeAll` 标的方法不是 static，会怎样？

**预设回答：**

- 答"每条用例都要的放 @BeforeEach，贵重的共享资源放 @BeforeAll"——对。连接池、容器启动放后者。

- 答"不是 static 会报错"——对。除非加 @TestInstance，把生命周期改成 PER_CLASS。

- 答"全塞进 @BeforeAll 省事"——省的是写的时间，赔的是排错的时间。共享状态会让用例互相踩。

- 追问：第 3 条红了怎么办？——先看第 1 条改过什么。这种红最难查。

**易错点 / 考点：**两组必分。每个方法一次的 @BeforeEach，整个类一次的 @BeforeAll。@ParameterizedTest 必须配数据来源。另外记一句：@Disabled 的用例不进统计，它和被断言判失败的用例不是一回事。

**过渡：**注解管怎么组织、怎么跑。过不过，得看断言。

---

## 【5.49】Assert 类（2.5 分钟）

> 右半边是整块代码，从 import 开始。前九行都是静态导入的断言。能认出 assertEquals、assertTrue、assertThrows。类叫 AssertionsDemo。两个 final 字段，一个 calculator，一个 person。

>

> 第一个测试方法叫 standardAssertions，里面三条断言。第一条 assertEquals，参数是 2 和 add(1, 1)。第二条 assertEquals，参数是 4 和 multiply(2, 2)。第二条后面还挂了一句说明文字。第三条 assertTrue，判 'a' 小于 'b'，后面跟一个 lambda。代码里一行红注释，点明说明文字现在是最后一个参数。

>

> 第二个方法叫 groupedAssertions，一句 assertAll。括号里先给个 person，再跟两个 lambda，一个比名字，一个比姓。分组断言的好处是：一条红了，后面的照跑，几处错一次看全。普通断言第一条红就停。

>

> 左边两段文字给的是机制。检查并判断运行结果是测试里的一件重要工作。测试程序里必然有断言条件。运行时不满足，就说明状态或结果与期望不一致，程序里有缺陷。JUnit 用 Assert 类提供一组断言方法。教材说这个类在 junit.framework 包。参数分期望和实际。相等就通过，不等就失败。

>

> 包名分两套。junit.framework 是 JUnit 3 和 4 的路径，答题按教材写它。右边代码用的是 JUnit 5 的 Assertions。两个都对，别混着答。

>

> 参数顺序要盯住。assertEquals 第一个写期望，第二个写实际。写反了，编译器不拦。通过时看不出问题。一失败，报告把方向说反，人跟着跑偏。

> **提问：** `assertEquals(3, result)` 和 `assertEquals(result, 3)` 都能编译。为什么后一种必须避开？

> **预设回答：**

- 答"没差别，相等就行"——通过时确实没差别。失败时报告会反着说，人判错方向。

- 答"失败信息会颠倒，排查方向错"——对。这是最实在的害处。

- 答"加一句说明文字就补上了"——说明是辅助。顺序是约定，不能靠一句话兜。

- 追问：断言写得少，覆盖率还能很高，为什么？——覆盖看的是执行到没到。断言写得粗，错也照绿。

> **易错点 / 考点：**三点。assertEquals 先期望后实际。Assert 的包名两套口径，教材写 junit.framework，代码用 Assertions。相等通过，不等失败。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，这句通常判错。

> **过渡：**断言不是孤零零一个类。到了 JUnit 3 那套设计里，它被测试类继承下去了。

---

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

> 右半边是一张表，三列：返回类型、方法名、说明。竖着十一行，从 countTestCases 排到 toString。第一列有几格是空的，那是不返回值的 void。有两行写着 protected。方法名那一列用了红黑两种字。表头右边被切掉一点，说明这表是从文档里截的。

>

> 表里这几行要认。run 是跑这条用例，顺手收进一个默认的结果对象。带参数的 run 是跑，结果记进你给的那个。runBare 是裸跑，不带夹具。runTest 是 protected 的，重写它，在里面写断言。

>

> 还有两行要停一下：setUp 和 tearDown。说明栏一行写打开一条网络连接，另一行写关闭连接。这两个就是今天 @BeforeEach 和 @AfterEach 的前身。JUnit 3 靠重写方法做夹具，JUnit 5 靠注解。

>

> 左边三段文字把三个类对上。TestCase 是给测试用例提供服务的核心类。它在 junit.framework 包里，并且继承自 Assert。所以断言方法能直接用，不用带类名。好处是随手能写。代价是测试类被绑在框架的父类上。JUnit 4 改成注解，不再强制继承。JUnit 5 改成静态导入。一路都在松开这层耦合。

>

> TestSuite 实现了 JUnit 的 Test 接口，用来管理每一个测试用例。关键在实现接口这四个字。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。

>

> TestResult 收集并记录所有用例的结果，分成功、失败、错误三类。看报告先看错误那一列。错误常常意味着这条用例压根没验到东西。

>

> 四个动作串起来：组织、执行、记录、呈现。组织是套件，执行是用例，记录是结果对象，呈现是运行器。答串了，整题就散。

**提问：**`TestCase` 继承 `Assert`，这个设计图什么方便，又留下什么麻烦？

**预设回答：**

- 答"方便，测试类里直接写 assertEquals"——对。

- 答"麻烦，测试类被框架的父类占住了"——对。Java 只能单继承，占用就收不回。

- 答"所以后来改成注解了"——方向对。注解之后又走到静态导入，测试类不再需要父类。

- 追问：那表里的 runTest 干什么？——重写它，断言写在里面。今天写在 @Test 方法里，位置换了吗？没换。

**易错点 / 考点：**三句定义要说全。TestCase 继承 Assert，最容易漏。TestSuite 实现 Test 接口，没这个接口就嵌不起来。TestResult 收集，Runner 报告。还可以加一条：setUp 和 tearDown 对应今天的 @BeforeEach 和 @AfterEach。

**过渡：**类的关系对齐了。看一段能跑的 JUnit 5 脚本，把这些东西用上。

---

## 【5.51】单元测试脚本（以 Junit 5 为例）（2 分钟）

这里是三块代码卡片。左边一块压在中间一块上。中间那块又压住右边一块。三块各讲一件事。

>

> 左边那块类叫 StandardTests，从上到下一串方法。先 @BeforeAll 标 initAll。再 @BeforeEach 标 init。然后四个 @Test。第一个 succeedingTest，方法体是空的。第二个 failingTest，里面只有一句 fail。第三个 skippedTest，头上压着 @Disabled。第四个 abortedTest，先 assumeTrue，再 fail。最后 @AfterEach 和 @AfterAll 各一个。四个方法摆出四种结果：成功、失败、跳过、中止。中止和失败是两码事。假设不成立就停，不算缺陷。

>

> 中间那块是顺序。类上挂 @TestClassOrder。里面两个 @Nested 类。@Order(1) 的 PrimaryTests 放 test1。@Order(2) 的 SecondaryTests 放 test2。分组靠嵌套，先后靠 Order。少了 Order 那句，两个组的执行顺序就不保证。

>

> 右边那块是参数化。@ParameterizedTest 配 @MethodSource。方法收三个参数：字符串、整数、列表。三条断言。比长度是 5，比范围在 1 到 2 之间，比列表大小是 2。下面那个静态方法返回 Stream，给了两组数据。apple 配 1 和 a、b。lemon 配 2 和 x、y。一条方法跑两遍，换数据不用改方法。

>

> 三块合起来是一段完整脚本。夹具在前后，用例在中间，断言在末尾，数据从外面来。写实验就按这个骨架。一条测试只围一件事，方法名写意图。

**提问：** `abortedTest()` 里那个 `assumeTrue` 判失败了，报告里算失败吗？

**预设回答：**

- 答"算失败"——不算。假设不成立叫中止，单独一类。

- 答"它连跑都没跑，因为头上有 @Disabled"——看得很细。这段照抄官方示例，停用是演示用的。把 @Disabled 摘掉，才是中止。

- 追问：什么时候该用假设？——前提不满足就没意义的时候。比如只在某个系统上才有意义的那条。

- 收口：四种结果分得清，报告才读得懂。

**易错点 / 考点：** AAA 三段必须说全：准备数据、调用被测方法、断言结果。参数化注解必须配数据来源，只写一个跑不起来。@BeforeAll 的方法默认要 static。别把 @BeforeEach 说成每类一次。

**过渡：** 一个类好办。被测代码一调外部服务，这些招都不够了。

---

## 【5.52】Mock 技术（2 分钟）

> 顶部一条深蓝横幅写着"Mock 技术"。下面一张图是 mockito 的商标：绿色的 mockito 单词，右边一只玻璃杯，杯里是莫吉托。据 GitHub 官方仓库 2026 年 11 月的发布记录，Mockito 已经到 5.25.0。看到它，说明从这里换话题。前面讲怎么把用例写对，接下来讲怎么把依赖换掉。

>

> 为什么非换不可。想一个下单方法。它查库拿订单。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。风控在灰度。消息队列没起来。支付接口按次收费。你的测试就卡在门口。

>

> 路只有三条。搭一整套真环境，贵、慢，还会因为别人的服务抖动而时红时绿。干脆不测，缺陷欠着。或者把下游换成你能控制的对象。第三条是这一节的路，统称测试替身。

>

> 换成替身，马上要分六个词：Fake、Dummy、Double、Spy、Stub、Mock。这六个词在卷面上很爱考，因为长得像。分不清，写出来的测试会跑偏。该用简化实现的地方塞了个桩，行为对不上。该只看返回值的地方用了 Mock，一重构就全红。

>

> 所以这一节的目标不是背六个定义。是知道这一次该用哪一个。判断的线索有三条。下游要不要真的做事。你要不要核对它被调了几次。这套替身会不会被很多用例复用。

**提问：**被测方法要调第三方短信接口。你搭真环境，还是用替身？凭什么定？

> **预设回答：**

- 答"用替身，短信发不出去还花钱"——对。有副作用的外部依赖，先隔离。

- 答"真环境更真实"——真实，代价也高。限流和延迟抖动会让结果没法重复。

- 答"那就都别测了"——缺口不会自己消失，只会挪到上线那天。

- 追问：真实性怎么补？——分两层。单元测试用替身钉逻辑，集成测试连真环境看接线。

**易错点 / 考点：**六个词的定义和分工，重点在 Stub 和 Mock。只给返回值、不去核对调用的，是 Stub。会核对次数、顺序、参数的，是 Mock。Test Double 是总称，别把它当成第六种新东西。

> **过渡：**六个词先钉四个。图上左边是定义，右边是一段跑得起来的代码。

---

## 【5.53】Fake、Dummy、Double 等（3 分钟）

> 左半边四组定义，每组开头一个方括号符号。标题下面还有一行蓝字：Dummy Object、Test Spy 可归为 Stub。右半边一段 27 行的代码，两个类。

>

> 先看左边。第一组 Fake，伪对象，括号里写着伪造、假货，反义词 real。下面两行说，它几乎和产品代码做同样的工作。为了满足测试要求，用更简单的实现。关键词是更简单的实现，不是返回一句假数据。

>

> 第二组 Dummy，哑对象，括号里写仿制品、哑巴、傀儡。下面三行说，它是对被测对象的模拟。缺原有的特性和功能，做欺骗性的替换。它完全听命行事，在具体测试里不起作用。方法签名要一个 Logger，这条用例却不写日志，传个空实现。那个空实现就是 Dummy。

>

> 第三组 Double，测试替身，有相同的行为和响应。它是总称。上面五个都算它的一种。

>

> 第四组 Spy，测试间谍，能接收参数并返回值。它比这句多一件事。它把调用记下来：谁调的、几次、参数是什么，事后供你核对。

>

> 右边代码把 Spy 演了一遍。TestSubject 里 methodA 直接抛运行时异常。methodB 先打印 begin，再调 methodA，再调 methodC，最后打印 end。测试方法里先 new 一个真对象。再用 Mockito.spy 包一层。然后 doNothing 把那次必抛异常的调用压掉。最后调 methodB。这样 methodB 能跑完，你能单独看它的行为。

>

> 代码有两处手误，抄的时候注意。注释里把 methodB 拼成了 mehtoB。最后一行对象名多打一个 t。

> **提问：**方法签名要传 `UserRepository`，这条用例根本不查库。你传什么？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"——不是。Fake 里面还在替原对象干活，空实现不干活。判据是有没有在替它做事。

- 答"传 null 最省事"——省事，也埋雷。真被调用就是空指针，报错位置离原因很远。

- 追问：怎么防它被误调用？——让不该被调的方法直接抛异常。这条用例就多了一层"它不该被叫到"。

> **易错点 / 考点：**四词定义，加那句归类：Dummy Object 和 Test Spy 可归为 Stub。三个错法：把 Fake 说成假数据；把 Dummy 说成返回默认值的桩；把 Double 当成某一个具体替身。

> **过渡：**四个词里，Fake 最容易听成"返回一句假数据"。下一页给它一个正经定义。

---

## 【5.54】伪对象（2.5 分钟）

> 左半边只有一条论点，短短的六行字。插桩在某些情况下不够用。被桩替换掉的行为，正是被测对象所需要的那部分。这时候伪对象是一种合理的折衷。最后一句给定义：伪对象是协作者的轻量级实现。右半边是一个米黄色的代码块，二十来行。

>

> 方法签名叫 MakePurchase，收三个参数：客户、商品、折扣。代码里五步。第一步建单，拿到一个 purchase。第二步 AddProduct，把商品挂上去。第三步 CreateInvoice，开票。第四步判折扣不为空，成立就 ApplyDiscount。第五步返回发票。

>

> 注意前三步都走 purchaseFacade 这个门面。每一步都会让数据以某种方式落下来。这就是桩撑不住的地方。给每一步配一个桩，各返回一个你编的值。编到第四步，折扣算得对不对，你自己都不信。因为前一步的结果没有真的进到后一步。桩只回答这一次返回什么，它不记事。

>

> 换成伪对象的门面，事情就顺了。前三步在内存里真的建出对象。第四步的折扣真的能读到数据，真的算在票上。行为是真的，只是没落库，也没把一整套老业务规则搬过来。

>

> 什么时候从桩换到伪对象，看三条。被测代码先写再读。它依赖下游的状态变化，比如从没支付到已支付。同一个协作者要被很多用例反复用。对上两条，就值得写一个。

>

> 反过来，伪对象不是白拿的。它自己有一套简化逻辑，这套逻辑也会有错。内存库和真库的语句也不一定通。只用在复用多、又值当的地方。

> **提问：**用内存库代替真库，会不会把"语句写错了"这类缺陷盖住？怎么办？

> **预设回答：**

- 答"会盖住，两边方言不一样"——对。一边能跑的写法，另一边未必。

- 答"所以还得有一层连真库的测试"——对。单元层保逻辑，集成层保语句。

- 答"那就干脆别用伪对象"——代价太大。大多数用例会慢到没人愿意跑。

- 追问：那这段下单代码该测什么？——测折扣算得对不对，不测持久化。持久化是另一层的事。

> **易错点 / 考点：**Stub 和 Fake 的差别。Stub 只给预设的返回值。Fake 提供简化但真实的行为。判据是被测对象依赖"怎么做"，还是只要"返回什么"。别把 Fake 说成造出来的测试数据，那是夹具。

> **过渡：**概念说完了。图上把 SUT 和 DOC 摆成两张对照图，位置一看就明白。

---

## 【5.55】Fake 示意图（2 分钟）

> 图分左右两组，中间没有箭头连着，是对照关系。左边一组三块。最上面是 RealComponentTest 的卡片。卡片里两个椭圆，分别叫 testMethod_1 和 testMethod_2。它往下一条粗箭头，指向 SUT。SUT 再往下一条箭头，指向 DOC。右边一组结构一样。测试类换成 FakeComponentTest。下面那块换成 Fake Object。Fake Object 背后还压着一张灰底的 DOC。

>

> 底部有一行图例。SUT 是被测系统。DOC 是被依赖的组件。这句先记住，再看位置。替身站的永远是 DOC 那一格。把 DOC 答成被测对象，整题就反了。

>

> 左边那张图上有两处蓝字。一处标在 SUT 和 DOC 之间，写 round-trip test。往返测试，一次测试里既写进去又读出来再比对。另一处标在最下面，写 public interface，下面加一句 front door。意思是测试从公开接口走进去，直接调对外的方法。别绕过接口去改内部字段。从接口进去，测的才是别人真实的用法。

>

> 右边多出来的东西是两个词。一条虚线从测试类绕下去，指到 Fake Object。旁边写 Installation，装配。意思是替身得有人装上，不是自己变出来的。通常就在测试的初始化里塞进去。另一处是橙色小字，标在替身指回 SUT 的箭头上，写直接输入。数据从这个替身喂进去。

>

> 右边那张灰底的 DOC 别漏看。它还在，只是被假对象挡住了。真组件没有消失，是这一条用例不碰它。箭头底下还有一个前提。SUT 只认接口，不认实现。被测代码自己 new 一个数据库连接，你连替换的机会都没有。

**提问：**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，能测就行"——眼前方便。测试和实现就绑在一起了，一重构就红。

- 答"public 一开就收不回来，别人会依赖它"——对。接口是承诺。

- 追问：私有方法里的复杂逻辑怎么测？——从公开接口把分支走到；走不到，就说明该抽成独立的类。

- 收口：测不下去的时候，先怀疑依赖是不是从外面传进来的。

**易错点 / 考点：**四个词对中文。SUT 被测系统，DOC 被依赖的组件，front door 是公开接口，round-trip test 是写进去读出来再比对，不是"把测试跑两遍"。图上还有一个词常被漏：Installation，替身要靠装配塞进去。

**过渡：**图看懂了，看一个自己动手写的替身长什么样。

---

## 【5.56】Fake 示例 1（2 分钟）

> 图分左右两组，中间一个大蓝箭头，从左指到右。每组四张小卡片，摆成田字。上面是 Unit Test 和 Parent。下面是 Child Test 和 Child。左右两张图结构一样，差别在两处。左边 Unit Test 里三条 itShould，后面的波浪线是红的。右边同样三条变成了绿的。右边还多出一个绿框，写着 Fake Child，框里只有一行 do。

>

> 看左边那条链。Parent 里 doStuff 的最后一行是 child.do。说明 Parent 干活要调 Child。Parent 的单元测试写在这个依赖之上。Child 还没写稳，或者行为不由你控制。那三条 itShould 就只能红着。这就是上一页说的困境。被测对象离不开协作者，协作者又不在你手里。

>

> 右边箭头之后，做法换了。写一个只实现 do 的假 Child。把它装到 Parent 脚下。Parent 那条 child.do 的箭头，直接指进绿框。Parent 的三条 itShould 一起变绿。Parent 关心什么？它只关心"我叫了 do"。do 里面怎么写，不归它管。既然只关心调用，那就不必等真的 Child。

>

> 右下角那张灰掉的 Child 别漏看。它还在，颜色淡了。真组件没被删掉，只是这一条用例不碰它。

>

> 左下角 Child Test 里两行要读清。itShould 是黑的，itShouldInstead 是绿的。Child 自己的行为换了写法，所以第一条让位，第二条接手。这一格提示一件事。替换掉的只是 Parent 那条测试里的依赖。Child 自己的测试，一条都不能少。替身不替你验证真组件。

> **提问：** `Fake Child` 里的 `do()` 如果写成空方法，Parent 那三条测试还有意义吗？

> **预设回答：**

- 答"有意义，验证了 Parent 有没有调它"——对。这三条测的是调用，不是调用结果。

- 答"没意义，什么都没发生"——要看断言写在哪。断言要是落在返回值上，空方法确实测不出东西。

- 追问：那空方法够不够？——不够。想核对调用，还得记下来：调了几次、参数是什么。这就是 Spy 的活。

- 收口：替身里写多少，取决于你要证什么。

> **易错点 / 考点：** 左边红、右边绿这组对照，说的是同一套测试在换掉依赖前后的两种状态，不是两套不同的用例。第二点，替身只顶替协作者，被测对象本身照测。第三点，假 Child 只实现被用到的那一个方法，别把整个 Child 抄一遍。

> **过渡：** 手工写假 Child 是一种。下一页换成另一种：直接拿内存里的东西当数据库。

# 第 5 章讲稿（5.57–5.68，v4）

> 逐页重写。每页先落到图上的实体：几个方框、几行代码、几条条目。取消板书。口播四拍：起 → 推 → 收。5.61 与 5.68 是概览页，只讲分工，不展开用法。

---

## 【5.57】Fake 示例 2（2.5 分钟）

> 右边那段代码是一个 DAO 测试类，类名 UserDaoJdbcImplTest。开头三个 static 字段：被测的 dao，一个 Connection，一个 HsqldbConnection。第一段方法带 @BeforeClass，名字是 setupDatabase。它先 forName 载入 hsqldb 的驱动类。再 getConnection，连接串里写着 jdbc 加 hsqldb、mem，库名是 my-project-test，末尾还带一句 shutdown 等于 true。串里两处 hsqldb 被高亮标出来。接着把连接包成 HsqldbConnection，交给 DBUnit 用。最后 dao.setConnection，再 dao.createTables。第二段带 @AfterClass，名字是 closeDatabase。两个 if 先判空，再关连接，关完把字段置 null。

>

> 左边三行字配这段代码。第一行说，要验证 DAO 能正常工作，就构造 fake object 代替数据库。第二行说，HSQLD 是流行的 in-memory database。这里少印了一个 B，指的是 HSQLDB。第三行只有 DBUnit 一个词。顶部那行小标题才是结论：fake object 等于 enhanced stub。

>

> mem 这个前缀是重点。库建在内存里，跟着 JVM 起落。每跑一次就是一份干净的库。shutdown 那句让连接关掉时把库一起收掉。DAO 测的 SQL 是真跑的，走的是真 JDBC，真类型映射。所以它比全用替身更接近线上。把 JdbcTemplate 换成替身，你只能证明调用发生过。SQL 写错、字段映射错，替身一句都不提。

>

> DBUnit 那一行是另一个动作。它不造库，只往库里灌数据。一条用例开始前，按 XML 或 CSV 把表恢复成同一份初始状态。测完再重建。数据是固定的，结果才能比。

>

> @BeforeClass 也要留意。它一个类只跑一次，表也只建一次。所有测试方法共用同一份库。前一条用例插

> 进去的行，后一条还看得见。用例之间的脏数据，就出在这里。要么每条用例自己收拾，要么改成每条重建。

>

> 还有一层差别别放过。内存库和真库的 SQL 方言不完全一样。它替代不了上线前在真库上跑的那一轮。

> **提问：**测一个 DAO，用 Mockito 把 JDBC 全 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 看你传进去的 SQL 和参数。内存库看这段 SQL 是不是真的返回了那几行。

- 答"Mock 快，内存库更接近真库"——对，这就是取舍。

- 答"两个都要"——方向对。逻辑层用替身跑得快，DAO 这一层用内存库把 SQL 和映射跑真。

- 追问：到底选哪个？——DAO 这一层的毛病多在 SQL 和映射上，所以内存库更值钱。

> **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，真行为加简化实现。别把 DBUnit 说成数据库，它只准备测试数据。别把 HSQLDB 说成 Mock 框架，它是真数据库，只是驻在内存里。也常被问成"@BeforeClass 和 @BeforeEach 差在哪"。

> **过渡：**自己写一个内存库是一种。也有人手写替身类，直接盯交互。

---

## 【5.58】模拟对象（2.5 分钟）

> 左边一大段话，三个词染成蓝色：状态、行为、交互。整段只讲一件事。Mock 把测试的焦点从状态换成了行为。右边一个浅黄底的代码框，七行。上面一个 @Test，方法名 useLenientMock。第一行造替身，new 一个 LenientMock，变量叫 campaignMock，这个名字也是蓝的。第二行把它传给 PurchaseWorkflow 的构造器。后面三行是链式调用：addItem 加一本书，书名由 getBookByTitle 取，参数是 Developer Testing；usingExistingCustomer 传客户号 1234567；enterDiscountCode 传折扣码 DEAL。最后一句 campaignMock.verify()，然后收花括号。

>

> 这段里没有 assertEquals。整条测试过不过，就看最后那一句。这就是它和前面那些例子的分界。fake 那

> 条以断言收尾，断言查返回值，或者去查属性。这里一条断言都没有，只问一件事：这个替身被怎么对待了。

>

> 再往下想一步，为什么值得为这种写法单独开一段。返回值错了，断言一眼看得出来。可有些事根本不在返回值上：报备发了没发，缓存清了没清，日志写没写。状态验证在这里没有抓手。行为验证有。报备那条可以写成"风控客户端必须被调一次"。返回值完全不用管。

>

> 链式那三行在读法上也有意思。它本身就是一段业务动作：加商品、认客户、用折扣码。三行连着写，等于把调用顺序写进了测试。将来流程改成两步，或者换掉一个方法名，这条测试就会红。业务结果一点没变，它也红。

>

> 名字里的 Lenient 是宽松的意思。多出来的调用，它不算失败。严格口径下，多调一次也要报。宽松有好处，测试不容易被无关调用打红。坏处是它挡不住多调。

>

> 最后那句 verify 不带参数。它核的是这个替身身上所有约定，一次结清。后面 Mockito 的写法是 verify 加方法和参数，一次核一条。两种口径，别混着记。

>

> 所以这条测试证明的是过程，不是结果。返回值看得见的计算，用断言就够，不必上 Mock。

> **提问：**一个方法"发送一封欢迎邮件"，返回值是 void。怎么验证它真发了？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"——先问业务上要不要这个返回值。不要，就别为了测试改产品接口。

- 追问：这类测试最脆的地方在哪？——把"怎么调"写死了。改个名、拆一次调用，它就红。

> **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"，这里的行为指交互。另一个错答是把 verify() 说成验证

返回值。

> **过渡：**状态和行为分开了。再把 Mock 和 Stub 并排放，看替身站在哪一侧。

---

## 【5.59】Mock vs. Stub（2 分钟）

图上左右各一块，结构一样，只有下半个方框换了名字。先看左块。最上面一个框，写 IndirectInputTest，里面两个胶囊形，testMethod_1 和 testMethod_2。中间一个大方框，写 SUT。下面一个方框，写 Test Stub，它下面压着一层灰影，写 DOC。

>

> 左块有三条线。第一条实线从测试框左侧下来，标注 Creation and Configuration，箭头进 Test Stub。第二条是虚线，从测试框下来，标注 Installation，进 SUT。第三条从 Test Stub 弯上去进 SUT，标注 Indirect Inputs，箭头朝上。

>

> 右块换了两处。上面那个框叫 IndirectOutputTest。下面那个方框叫 Mock Object，框里还画了一个灯泡形状的小图标。方向全反了：测试框一条实箭头直下，进 SUT；Creation and Configuration 那条竖线还是进下面的方框；SUT 反而有一条箭头向下，进 Mock Object，标注 Indirect Outputs。虚线 Installation 两边都有，都是把替身装进 SUT。

>

> 两张图并起来读，区别就一句。左边替身站在上游，往 SUT 里喂值，喂进去的叫间接输入。右边替身站在下游，SUT 往它身上写，写出来的叫间接输出。

>

> 这正好对上前面那两种验证。喂值的替身，你只能断言结果，这是 Stub 的活。被写的替身，它自己记下了这次交互，测试回头来查，这才是 Mock 的活。名字换了，箭头方向换了，验证的方向也跟着换。

>

> 框里那两个胶囊形也别略过。它说明同一套结构对多条用例都成立，每条用例各写一份期望。图里画两个，是因为两块替身分别要应对两套期望。

>

> DOC 那层灰影别忽略。它写的是 depended-on component，被测代码依赖的那个东西。两个替身框都压在它上面，意思是替身站的就是它的位置。

>

> 所以判据很硬。箭头朝 SUT，是 Stub。箭头从 SUT 出来，是 Mock。看懂这张图，就分得清谁在供数据，谁在收数据。

> **提问：**同一个被测类，一次改动把它的输出从"返回一个值"变成"回调一个监听器"。你的测试要跟着改什么？

> **预设回答：**

- 答"改成核监听器被调用"——对。输出从返回值挪到了回调上，观察点就得跟着挪到那个替身身上。

- 答"还是断言返回值"——返回 null 了，断言什么也说明不了。

- 答"这时候替身从 Stub 变成了 Mock"——说清了一半。位置没变，是数据流向变了，验证方式跟着变。

- 追问：两种写法哪个更脆？——核交互那条更脆，它绑住了调用方式。

> **易错点 / 考点：**图上的关键符号要认得出：Indirect Inputs 指向 SUT，Indirect Outputs 从 SUT 指出；Installation 是虚线，Creation and Configuration 是实线。高频错答是把 Mock 说成"更高级的 Stub"。不是层级关系，是关心的方向不同。

> **过渡：**两张图放一起，强弱也就排得出来了。下面四条把这条梯子收一遍。

---

## 【5.60】Stub 和 mock 小结（3 分钟）

> 四条带空心方块的条目，没有图，也没有代码。第三条和第四条各占两行，前面两条一行。四条从头排到尾，不是并列的四类，是一条从弱到强的梯子。

>

> 第一条最短。Stub 最弱，只是参与测试，为了让测试跑起来。它只提供输入，不参与判断。弱不是缺点。你改了实现，它不跟着红。

>

> 第二条最长。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。例子给的是 Dummy server：给它不同的输出，它给不同的响应。内部没有实质计算，是从接口或基类派生出来的。这一句把它和 Fake 分开了。Fake 有简化过的计算，Dummy 没有。前面把 Spy 归到 Stub 那一侧，是因为它不强制核对交互。这里按强弱排，Spy 多看一眼结果。两处说法不冲突。

>

> 第三条。Mock 更强，要求你说清它如何参与，又比较灵活，视情况而定。它的行为可能像 Dummy，像 Stub，也可能像 Spy。配一配，这几个角色它都演得了。演得了，不等于每次都要上最强的。

>

> 第四条最要紧。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，开始标上禁用注解。禁用一多，测试文件还在，保护没了。

>

> 梯子记住，判据也记住。问一句：这件事能不能只用一个返回值说清。能，就停在 Stub。观察不到结果，比如返回 void、发消息、写日志，才往上升。还有一种要上升：顺序本身就是业务要求，先扣库存再扣款，倒过来就是事故。

>

> 所以这四条不是在比谁先进。它在教你用最轻的那一种把话说清楚。

**提问：**组里有人每个测试都写三条 verify。你拿什么说服他往下退一步？

**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"做一次能重复的对比"——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。拿这个结果说。

- 收口：说它好或不好，都要有一次别人能自己跑一遍的对比。

**易错点 / 考点：**四句要能说全：Stub 最弱；Dummy 和 Spy 强一些；Mock 更强且能灵活扮演；够用就行。

判断题常出"Mock 比 Stub 好，应优先使用"，错。另一个点：Dummy 内部没有实质计算。

> **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）

左边四个方块条目，竖着排：JMock、EasyMock、Mockito，第四个 Moco 后面跟着括号，写着 Java API、独立服务器。右边是一块浅灰底的代码截图，五段，每段前面一行绿色注释：Fixture setup、Mock configuration、Mock installation、Exercise、Verify。

>

先说名字的分工。前三个是同一个 JVM 里的 Mock 框架，在测试代码里造替身，管返回值，记调用。它们还有先后：JMock 最早，接着 EasyMock，再是 Mockito，一代比一代轻。第四个 Moco 不属于这一支。它是服务桩，自己起一个 HTTP 服务器，按配置回响应，被测代码要走网络才碰得到它。括号那句就是提醒这件事：它有 Java API，也能当独立服务器跑。

>

判据很具体。这个协作者能不能在测试方法里直接 new 出来。能，就归前三个管。要走网络、要占端口，才轮到 Moco。名字先摆到这里，下面几页细讲 EasyMock 和 Mockito。

>

右边那段代码值得单独看一眼。方法名 testRemoveFlight_Mock，测的是一个订票门面。第一段准备数据，造一个航班对象。第二段是重点，红色字占了大半：先 new 一个 ConfigurableMockAuditLog，再 setExpectedLogMessage，一次传四个参数，今天日期、测试用户名、删除动作码、航班号。接着 setExpectedNumberCalls，约定只调一次。第三段把它塞进门面。第四段真删一次。第五段先断言航班没了，再调替身的 verify。

>

这五段不是为了讲某个框架的 API。它展示的是手写替身要怎么摆：期望先写，装进去，跑一遍，最后结账。名字带 Configurable，说明它是自己写的一个可配置替身类。四个框架帮你省掉的，正是中间那段配期望的代码。

>

> 所以选型先分场合。同进程的协作者，用前三个。要走网络的 HTTP 依赖，用 Moco 这类服务桩。

> **提问：**被测代码要调一个第三方支付接口，走的是 HTTP。用 Mockito 还是 Moco？Mockito 在这里差在哪？

> **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，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测不到。

- 答"两层都做"——内行。逻辑层 mock 客户端，契约层用 Moco 走一遍真 HTTP。

- 追问：这四个里为什么只细讲两个？——EasyMock 和 Mockito 是课上实验要用的那一档。

> **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。易错有两个：把 Moco 说成又一个 Java Mock 框架；把 JMockit 和 JMock 当成同一个。

> **过渡：**三个 Java 框架里先看 EasyMock。它把录制、回放、检查三步写在明面上。

---

## 【5.62】EasyMock（2.5 分钟）

> 橙色那段话是定义，占七行。它说 EasyMock 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、检查三步走完。它能核对调用的种类、次数、顺序，也能让替身返回指定的值，或者抛出指定的异常。

>

> 下面浅蓝底那块是四行代码。第一行 EasyMock.createControl，交回一个 IMocksControl，变量名 control。后三行都调 control.createMock，参数分别是 Connection、Statement、ResultSet 三个类。左边变量的类型写的是全限定名 java.sql 打头，说明没有 import，直接按接口造。

>

> 这四行里有两个点要读出来。

>

> 第一个，三个替身共用一个 control。它们不是各管一段，而是同一份剧本下的三个角色。所以核对可以跨对象：先要 Connection，再拿 Statement，再执行，最后取 ResultSet。顺序错了，同一个 control 就能报出来。

>

> 第二个，参数全是接口。Connection、Statement、ResultSet 都是 JDBC 的接口，不是实现类。定义里那句"对接口的模拟"，指的就是这件事。要造一个具体类的替身，这条路当年走不通。这是它后来被 Mockito 比下去的一处。

>

> 三层为什么正好对上。一次查询就是三步：拿连接、开语句、读结果。后一步要用前一步的对象，所以替身也得按这个层次造，一次造三个。只造一个 Connection，后面两步还得真跑。

>

> 为什么必须先录。替身一放进被测代码，就在被动应答，中途改不了剧本。所以期望要写在运行之前。回放是把这套期望生效，检查是跑完对一遍账。约好三次，只来两次，检查那一步就报。

>

> 录制还没开始。这四行只是把替身造出来，期望得接着写。据 GitHub 官方仓库 2026 年 8 月的发布记录，EasyMock 已更新到 5.7.0，仍在活跃维护。

**提问：**三个替身为什么共用一个 control，而不是各造各的？

**预设回答：**

- 答"好统一管理"——太虚。具体管的是期望和检查。

- 答"因为要核跨对象的顺序和次数"——对。先 createStatement 再 executeQuery，顺序错了要能报。

- 答"因为回放要一起放"——也是。同一个 control 上写好的期望，回放时一起生效，检查时一起结清。

- 追问：那 Mockito 为什么不需要这一步？——它不预录剧本，跑的时候记日志，事后 verify 去查。

**易错点 / 考点：**录制、回放、检查三步必考，还要能说清可验证的是调用的种类、次数、顺序。易错是把 verify 说成"验证返回值"，返回值归录制阶段的设定管。第二个易错：把 IMocksControl 说成被测对象。

**过渡：**EasyMock 先把剧本录好。Mockito 换了顺序，跑完再查。

## 【5.63】Mockito（2 分钟）

> 三个深色代码块，配三个蓝色标签，摆成斜的三角。最上面一块最短，只有两行构建脚本，左边标签写"设置依赖性"。内容是 repositories 加 mavenCentral，再一行 dependencies，引进 mockito-core，版本写成 3 加点加号。这个写法每次取最新的 3 系列，课件写的时候就是这么写的，记住它，后面好对比。

>

> 中间那块最大，标签"验证交互性操作"压在它下面。开头一行静态导入，把 Mockito 那个类下的方法全引进来。然后注释一行 mock creation，接着 mock 一个 List。再两行调用，add 一个字符串 one，然后 clear，注释写着它不会抛 unexpected interaction 异常。往下注释写着 selective、explicit、highly readable verification，两行 verify，一次核 add，一次核 clear。

>

> 右下那块压在中间块上，标签"Stub 方法调用"在它下面。第一行注释说，具体类也能 mock，不只接口，代码里就是 mock 一个 LinkedList。第二行注释说，打桩要写在执行之前。然后 when 里面取第 0 个元素，thenReturn 交回 first。下一行打印第 0 个，输出 first。最后一行打印第 999 个，注释写着输出 null，因为没打桩。

>

> 三块合起来就是三个动作：造替身、打桩、验交互。中间那块一行打桩语句都没有，它只证明两件事：mock 出来的对象调用不报错，verify 能事后核对。右下那块专管返回值，第 999 个那句是重点。没打桩的方法，它交回 null，不报错。这就是全假替身和 spy 的分别。

>

> unexpected interaction 那个词针对的是 EasyMock 那条路。那边多调一次就失败，这边默认不管。你不写 verify，它就不查。宽松是好事，也是坑：以为验过了，实际一句 verify 都没写。

>

> 三段还有个顺序不能倒。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。据官方仓库 2026 年 11 月的发布记录，Mockito 已到 5.25.0。

> **提问：**页面上的依赖写的是 3 加点加号，现在主线已经到 5.x。同一个 mock 一个 List 的写法会变吗?

> **预设回答：**

- 答"会变，要重写"——不会。三段结构没动，静态导入的路径也没动。

- 答"结构不变，能 mock 的范围变宽了"——对。早期不能 mock 具体类，后来能 mock final 类。

- 答"这种浮动版本号不能用在生产"——对了一半。它每次都取最新小版本，构建不可重复。锁具体版本更稳。

- 追问：升级时最先看什么？——看有没有用被移除的老 API，比如旧的验证模式。

> **易错点 / 考点：**三个标签要对上三个动作：设置依赖性、Stub 方法调用、验证交互性操作。易错有三个：把"设置依赖性"理解成配 Spring；以为 verify 会验返回值；以为 mock 出来的对象调用未打桩的方法会抛异常。它不抛，交回 null 或 0。

> **过渡：**三个动作拆开，落到页面上就是那份清单。

---

## 【5.64】Mockito 主要功能（3 分钟）

> 纯文字，四个圆点条目，第一和第四条各带两三行缩进说明，没有图，也没有代码。四条正好是上一页三个动作的展开。

>

> 第一条，mock 和 @Mock 用来创建 mock。下面第一行说，可以用 Answer 或 MockSettings 指定它的行为方式。第二行说，when 和 given 指定它如何行动。第三行说，如果给的返回不合需求，可以自己写一个返回接口的扩展。第三行讲的是自定义 Answer，把"被调用时跑哪段代码"交给你写。

>

> 第二条，spy 和 @Spy，部分模拟。它调用真实的方法，但仍然可以被验证。半真半假，只改你想钉住的那一个。

>

> 第三条，@InjectMocks，自动注入带 @Spy 或 @Mock 注释的字段。被测对象里那几个协作者，不用你手写 setter。

>

> 第四条，verify，检查方法是否用给定的参数调用过。下面两行讲参数怎么给。一行说可以用灵活的参数匹配，比如 any 这种任意表达式。另一行说可以用 @Captor 把实际传进去的参数抓出来。

>

> 四条的顺序要顺过来。第一条造，第二条是第一条的特殊版，第三条负责塞进被测对象，第四条结账。少了第三条，替身造好了却进不去，被测对象里全是 null。这种报错很常见，看着像框架没生效，实际是没注入。命中了这个问题，先查注入那一头，别急着怀疑版本。

>

> 参数那一行最容易出事。同一个方法里，匹配器不能和具体值混着写。一个参数写 any，另一个写死字符串，会直接抛异常。要么都写匹配器，要么都写具体值。这条规矩没有例外。

>

> @Captor 的用场值得记。有些参数没法事先写死，比如一个自动生成的流水号。抓出来再断言它的格式和长度，比放宽成 any 有力得多。

>

> 四条先用熟三条：mock、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，剩下的走真实逻辑。用 @Mock 还是 @Spy?

> **预设回答：**

- 答"用 @Spy"——对，这就是部分模拟。

- 答"用 @Mock，只给 getDiscount 打桩"——另外九个变空，返回 null 或 0。被测逻辑会在不相干的地方失败。

- 答"看真实逻辑是在帮忙还是添乱"——判据抓住了。它会去连库，就用 @Mock。

- 追问：@Spy 有什么坑？——它跑真方法，真方法里的副作用也一起跑。

> **易错点 / 考点：**六个 API 的作用要能对上。三个高频错：verify 不写次数就是一次，调两次也红；匹配器和具体值混用会抛异常；@Spy 和 @Mock 弄反。判断题还爱出"Mockito 只能 mock 接口"，错。

> **过渡：**清单里 @Mock 被单独拎出来讲了一页。一个注解为什么要占那么长？

## 【5.65】Annotation Type Mock（2.5 分钟）

深色代码面板占了多半幅，右下角压着一块蓝底说明框，五行字。

>

先说面板。上面一个类 ArticleManagerTest，继承 SampleBaseTestCase。类里连着六个 @Mock 字段，每个带不同参数。第一个是光秃秃的，私有字段 ArticleCalculator calculator。第二个带 name 参数，值写 database，字段叫 dbMock。第三个带 answer 参数，值是 RETURNS_MOCKS，字段是 userProvider。第四个带 extraInterfaces，传 Queue 和 Observer 两个接口。第五个带 strictness 参数，值是 LENIENT。第六个带 stubOnly，值为真。往下 setup 方法里，new 一个 ArticleManager，把这几个替身当构造参数一次传进去。

>

面板下半幅是基类 SampleBaseTestCase。一个 AutoCloseable 字段，@Before 里调 openMocks 把 this 传进去，@After 里调 closeable 的 close。生命周期的头和尾都在这儿。

>

蓝框五条讲注解的用处。第一条，把一个字段标记为 Mock。第二条，允许速记创建。第三条，尽量减少重复的 Mock 创建代码。第四条，让测试类和验证错误更好读。第五条，自动检测 MockedStatic 类型的静态 Mock。

>

六个参数正好对上 mock 方法的那组重载。name 让报错带上你给的名字，不是一串哈希。answer 那一项让没打桩的方法交回一个空对象，不是 null。extraInterfaces 让替身同时是 Queue 和 Observer。LENIENT 是放松严格校验，多调不算失败。stubOnly 是不记调用，只当 Stub 用。

>

所以六个字段是同一件事的六种配置。注解把创建搬到了字段上，参数把 mock 的重载搬到了注解里。读这半幅的时候，别把它当六个孤立的例子，它是一张参数对照表。

>

> 读的时候有个细节：构造器里传的是 database，字段名却是 dbMock。摘录时改过名，照注解那几行读，别把它当能直接编译的代码。

>

> 还有一个坑。注解写上了，字段却是 null，因为没走 openMocks 或对应的扩展。省代码是一层，管生命周期是另一层。

**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，忘了会污染别的用例"——对，它在整个 JVM 里生效。

- 答"注解会自己开关"——对。创建和释放都交给框架，比手写 try finally 稳。

- 答"openMocks 返回的那个对象就是收尾的把手"——正是面板下半幅那三行。

- 追问：注解一般在省什么？——成对出现的动作。人容易忘的总是后一半。

**易错点 / 考点：**用 @Mock 之前要启用 Mockito 扩展，或者调 openMocks。易错是以为加了 @Mock 就会自动注入被测对象，注入归 @InjectMocks。还要认得出 name、answer、extraInterfaces、strictness、stubOnly 这五个参数各管什么。

**过渡：**given 这种写法前面一闪而过。它是 BDD 那一套的入口。

---

## 【5.66】支持 BDD（2.5 分钟）

两块深色代码面板斜着叠在一起。左上那块长，右下那块短，压在它上面。

>

> 左上开头一行静态导入，把 BDDMockito 那个类下的方法全引进来。接着两行准备：mock 一个 Seller，再 new 一个 Shop，把卖家塞进去。然后方法 shouldBuyBread，里面三段注释：given、when、then。given 那段是 given 里面调卖家问面包，然后 willReturn 一个 new 出来的 Bread。when 那段只有一行，把 buyBread 的结果接住。then 那段是 assertThat，判定拿到的东西是面包。

>

> 右下那块是另一个例子。第一行把 person 包成 InOrder。中间三行是真实调用：开车、骑车、又骑一次车。最后两行是核对：then 加 should，带 inOrder，核开车；再来一条，带 inOrder 和 times 二，核骑车。

>

> BDD 本身是三句话的写法：给定什么，当发生什么，那么结果是什么。左上的三段注释正好对上这三句。业务方读得懂，代码也照着写。它解决的是沟通，不是技术难度。要不要这么写，判据不在语法上，而在读的人是不是业务方。

>

> 两块的语法是一套。given 顶替 when，willReturn 顶替 thenReturn，then 加 should 顶替 verify。语序从"当它被调用就返回"换成"给定这次调用，将返回"。

>

> 右下那块还多带了两件东西，别忽略。它用了 inOrder，所以先开车后骑车这个顺序被钉住了。它还用了 times 二，所以骑车必须是两回。这两条都是行为上的硬约束，业务上说得清：先开车，再骑车，只骑两回。

>

> 所以 BDD 那套写法不是花架子。它能表达的东西和 when、verify 一样多，只是句子顺着人读。代价在一个类里混着两种风格，读的人要在两套语序间来回切。定一种，别混。

>

> 它也不让断言更正确。方法名照着 BDD 写成 shouldBuyBread，报告拿出来业务方看得懂，这才是它值的地方。

**提问：**inOrder 这种核对，什么业务上非查不可？

> **预设回答：**

- 答"有先后依赖的流程"——对。先扣库存再扣款，倒过来就是资金事故。

- 答"随便，反正结果对就行"——结果对不代表顺序对。顺序错的时候，往往是中间态出了问题。

- 答"inOrder 会拖慢测试"——会多花点核对成本，但比线上错序便宜。

- 追问：什么样的顺序不值得查？——纯查询、无副作用的那种。查了只会让测试更脆。

**易错点 / 考点：**BDD 支持由 BDDMockito 提供，given、willReturn、then 加 should 分别等价于

when、thenReturn、verify。两个易错：把 BDD 说成一种测试层级，它是写需求和协作的方法；把 then 加 should 说成在断言返回值，它验的是交互。

**过渡：**交互能写清楚了。下一个问题换了：我怎么知道测够了。

---

## 【5.67】测试覆盖率工具（2.5 分钟）

题目换了。前面讲怎么造替身、怎么核对交互。接下来问另一件事：我怎么知道测够了。

>

一条深蓝横带横过页面，上面四个白字，测试覆盖率工具。带子下面是一个深红衬线的大标志，一个字母一个字母拼成 JaCoCo，底下压着一行小字，Java Code Coverage。标志先到，说明这一段的主角是谁。

>

为什么非得换这个问题。用例全绿，你仍可能一个类里的代码一行都没碰到。通过只说明跑过的那些路径是对的。没跑过的，绿灯一个字都不提。这种漏法最安静，也最容易在评审时被放过。

>

覆盖率工具补的就是这块。它不判对错，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，还得靠你补用例、补断言。工具不会替你把错改掉，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。

>

所以对它的期待要先摆正。它是下限，不是质量证明。这句话后面几页会反复出现。

>

这一段怎么走。先把工具按语言分一遍，看各家用谁插桩。再落回 JaCoCo，看它给你哪几个数字、插桩插在哪、报告长什么样。名字不用背全，链条要留住：插桩、记录、汇总、出报告。

>

最后一句提醒。覆盖率不进每次构建，数字就只是当天的快照。新类加上去没人盯，过几个月就掉下来。它是个会过期的指标。要让这个数字有用，就得让它每次构建都自动跑、掉了能告警。

> **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，未知变成了一份缺口清单。

- 答"覆盖率很高还出事，正常吗"——正常。那部分断言不够，或者缺陷在没设计到的输入上。

- 追问：那它为什么还要用？——它能把"我以为测过了"变成一串能被查的数字。

> **易错点 / 考点：**先记住这一段的定位：覆盖率是必要不充分条件。判断题"覆盖率高等于测试充分"，答案始终是错。第二个易错是把工具当成"能把覆盖率提上去"的手段，它只测量，往上提靠写用例。

> **过渡：**名字现在摊开。先看各语言分别用谁来插桩。

---

## 【5.68】代码覆盖率分析工具（2 分钟）

> 三条带绿色空心方块的条目，长短差很多。第一条两行，第二条两行、名字带下划线，第三条一行半，最后收在一串省略号上。

>

> 第一条是 JaCoCo。它讲的是机制：对编译后的 Java class 文件做插桩，统计代码的执行情况，收集覆盖率，再出报告。关键词是编译后。它不动你的源文件，源码只是后来对照着看。

>

> 第二条是 BullseyeCoverage。名字带下划线，像网页上的链接。内容说它是获取 C 和 C++ 代码覆盖率的工具，支持分支覆盖率，英文 Branch Coverage 写在括号里。

>

> 第三条是一串名字：Testwell CTC++、Parasoft 的 Jtest 和 C++Test、CoverageMeter、Jcov、CodeCover、Coverage.py，后面跟省略号。省略号的意思是还有，别把它当全部。

>

> 按语言分工，一口能说完。Java 这边是 JaCoCo，还有 Jcov、CodeCover 这些同类。C 和 C++ 那边是 BullseyeCoverage 和 Testwell CTC++，两个 C++ 测试工具也在这一侧。名字里带 py 的 Coverage.py，方向是 Python。剩下几个别照着名字猜语言，回去查文档。

>

> 为什么这里只给名字。选型先要的是谱系，不是装法。每个名字只说一句它管什么，版本、配置、参数都不在这儿。后面几页只细讲一个，就是第一条那个 JaCoCo。理由很实际：这门课的实验是 Java，报告要从构建工具里出来。

>

> 还有一层分工要看清。BullseyeCoverage 那条专门点了分支覆盖，说明覆盖率不止"这行走过没有"。一个 if 的两个方向、一个 switch 的每个分支，都可能没走到。语句覆盖高、分支覆盖低，缺口就在这些地方。

>

> 最后记一句判断。列出工具名不难，难在说清插桩插在哪。JaCoCo 插在编译后的 class 文件上。工具不同，链条一样：插桩、记录、汇总、出报告。

**提问：**同一个项目里既有 Java 代码，又有 C++ 的本地库，覆盖率要怎么算？

**预设回答：**

- 答"用一个工具全搞定"——做不到。JaCoCo 只管 Java 字节码，碰不到 C++。
- 答"两边各用一套，分别报"——对。Java 用 JaCoCo，C++ 用 BullseyeCoverage 这类，按模块各自给数。
- 答"合起来报一个总分"——危险。两种工具的分母不一样，加起来的数没有意义。
- 追问：那报告怎么给人看？——分语言列，各自给分母，别混成一个百分比。

**易错点 / 考点：**工具和语言要对上。JaCoCo 是 Java，插桩对象是编译后的 class 文件。BullseyeCoverage 是 C 和 C++，主打分支覆盖。Testwell CTC++ 也是 C 和 C++。Coverage.py 是 Python。三个高频错：把 JaCoCo 说成测试框架；把 BullseyeCoverage 说成 Java 工具；以为覆盖率工具能把覆盖率提上去。

**过渡：**谱系看完，落回第一个名字。它到底给你哪几个数字。

# 第 5 章讲稿（5.69–5.80，v4）

> 逐页重写。每页先落到图上的实体：八个编号条目、两段 premain、三栏类池、两张报告表、一棵模块树、一张服务框图。取消板书。口播四拍：起 → 推 → 收。

---

## 【5.69】实现覆盖率度量：Jacoco（3 分钟）

> 上一屏把工具按语言排了一排，名字记了一堆。落到 Java 这边，要动手的是 JaCoCo。它凭什么算覆盖率？标题右边挂着一行网址，www.eclemma.org/jacoco，那是项目主页。下面八条编号，从①排到⑧，是它的能力清单。

>

> ① 给的是分析维度，六个：指令、分支、行、方法、类、圈复杂度。前五个才是覆盖率，第六个不是。

>

> 为什么这么分？粒度不一样。指令最细，数的是字节码指令走过了多少条。分支看每个判定的两个出口，if 只走 true 没走 false，叫部分覆盖。行按源码行算。方法更粗，方法里只要有一条指令走过，这个方法就算覆盖。类最粗，类里任何一个方法跑过，这一类就算过。

>

> 圈复杂度是另一码事。它量的是复杂度，不是覆盖。McCabe 那个式子是边数减点数加二，JaCoCo 按方法算。它回答的是：路径要走全，最少得几条用例。所以它得跟覆盖率并排看，别当成第六种覆盖率。一个方法圈复杂度高、覆盖又低，那是第一顺位要补的。

>

> ② 说的是统计口径：基于 java 二进制码，不需要源文件。报告里能高亮到某一行，是因为 class 里带着行号调试信息。关掉调试信息再编译，就只剩方法和类两级。

>

> ③ 是插桩方式，on-the-fly，靠 java agent 挂上去。后面三屏都讲它。④ 讲能挂哪儿：OSGi 框架、web 容器、EJB 服务器，凡是跑在 Java 虚拟机上的都能接。⑤ 说兼容所有 class 文件版本，也支持别的 JVM 语言，

> Kotlin、Scala 编出来都是 class。⑥ 是三种报告：HTML 给人看，CSV 和 XML 给流水线做门禁。⑦ 说执行数据随时能取，走远程协议或者 JMX，不必等整轮跑完。⑧ 是构建侧入口：ANT task 和 Maven 插件，收集数据再出报告。

>

> 据 GitHub 官方仓库 2026 年 11 月的发布记录，JaCoCo 已经发布 0.8.16，仍在维护。名字有两种写法，Jacoco 和 JaCoCo，同一个东西。

**提问：**一个方法行覆盖满了，分支覆盖只有一半。问题出在哪儿，该补什么？

> **预设回答：**

- 答"代码有 bug"——推不出来。覆盖低只说明没测到，不说明测错了。

- 答"有分支没走，补让判定走反方向的用例"——对。只测了正数，负数那一支从没进去过。

- 答"行覆盖满了就够了"——短路右边、空的 else，照样可以一次都没测。

- 收口：先看缺哪个方向，再看圈复杂度还差几条路径。

**易错点 / 考点：**六个维度要全答；问"哪些是覆盖率"要把圈复杂度摘出去，这是最常设的陷阱。"基于二进制码、无需源文件"常被改成"必须有源码"。on-the-fly 是运行时插桩，磁盘上的 class 不动。报告三种记全：HTML、CSV、XML。口诀"指、分、行、方、类、圈"，前五个从细到粗，第六个不是覆盖。

> **过渡：**六个数字从哪来？先看 agent 启动那一下。

---

## 【5.70】on-the-fly instrumentation（2.5 分钟）

> 启动命令里多出一串 -javaagent，后面跟着 jar，等号后面还有键值对。你的 main 还没跑，JVM 先读这个 jar 的清单，找到 Premain-Class，去调 premain。

>

> 左边那段代码就是入口。包名 org.jacoco.agent.rt.internal，类名 PreMain，方法 premain 收两个参数：一个是 options 字符串，一个是 Instrumentation。方法体只有五行，干三件事。

>

> 第一件，拆参数。new AgentOptions(options)。等号后面那串键值对在这行被分开：排除哪些包、数据文件写哪儿、端口多少，都从这儿进来。

>

> 第二件，把通道建起来。Agent.getInstance 拿到代理，createRuntime 造出运行时，再 startup，参数是 agent.getData()。收数据和交数据的口，从这儿打开。第⑦条说的远程协议和 JMX，露的就是这个口。

>

> 第三件，注册转换器。inst.addTransformer，塞进去一个 CoverageTransformer，构造参数里带 runtime、agentOptions，还有一个 IExceptionLogger.SYSTEM_ERR。最后这个参数留了条退路：哪个类分析失败，记到标准错误，然后原样放行。第三方 jar 的字节码太老转不了，就这一类不统计，别的不受影响。覆盖率不能把应用搞崩。

>

> 右边那棵树在给这些做法分类。根上写着 Code Coverage，往下分两支。左边一支叫 Runtime Profiling，底下两个叶子 JVMPI 和 JVMTI，那是虚拟机给的调试接口，用来旁听，不动字节码。右边一支叫 Instrumentation，先分 Source 和 Byte Code：一个改源码，一个改字节码。Byte Code 再分两路，Offline 底下是 Replace 和 Inject，先插好再跑；On-The-Fly 底下是 Class Loader 和 Java Agent。

>

> 染成橙色的那条路是 Instrumentation 到 Byte Code、再到 On-The-Fly、最后 Java Agent，一共四个框。左边那段代码就站在这条路上。为什么挑这一支？源码不用动，编译产物也不用提前处理，挂上去就能统计。

>

> premain 自己一行字节码都不改。它只把钩子装上。改字节码发生在每一次类加载。

>

> 那棵树还有一层用处。以后面试问"覆盖率工具怎么拿到数据"，你能答出两条完全不同的路。一条是旁听，靠虚拟机上报；一条是改造，往代码里加东西。前者不改产物，能测的东西受接口限制。后者精确到每一行，代价是要能拿到 class。JaCoCo 选了后一条，也顺手选了它带来的那个麻烦：你改过的东西，解码器不认怎么办。所以才有 SYSTEM_ERR 那一手。

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

> **预设回答：**

- 答"整个应用起不来"——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，别的照常"——对。覆盖率不能变成应用的故障源。

- 追问：这算什么原则？——旁路设施要能降级。埋点、日志、监控出问题，主流程还得走。

- 收口：工具是来旁听的，不许抢方向盘。

> **易错点 / 考点：**premain 是 agent 入口，不是插桩本身。问"哪一步改字节码"，答类加载时的 Transformer。启动参数写法 -javaagent:xxx.jar=key=value，别和 -agentlib 混。addTransformer 只管之后加载的类，启动前就在虚拟机里的核心类不在内。树里 Offline 是 Replace 加 Inject 两个叶子，别答成一个。

> **过渡：**钩子挂上了，插桩这一下交给 ASM。

---

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

> 类加载器把 class 的字节流递进来，出去的时候已经不是原样。左边这个方法签名写得直白：进来一个 ClassReader，出去一个 byte 数组。它是一条加工线。

>

> 第一步读。ClassReader 把字节流解析成能一路遍历的结构。

>

> 第二步准备写出器。new ClassWriter，第一个参数把 reader 传进去，为的是复用常量池，输出小一点。第二个参数是 0。

>

> 第三步最容易被跳过：重写 getCommonSuperClass，方法体直接抛 IllegalStateException。为什么？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，干脆拒绝。

>

> 第四步，探针得有地方记"走没走过"。ProbeArrayStrategyFactory.createFor，按类的形态挑策略：放静

> 态字段、每次新建、还是从外面注入，工厂替你定。

>

> 第五步，把访问者串成一链。外层 ClassProbesAdapter，套着 ClassInstrumenter，再套到 writer 上。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。

>

> 第六步 reader.accept，第二个参数是 EXPAND_FRAMES。栈映射帧被展开，插的位置才算得准。最后 writer.toByteArray，交出新的字节。

>

> 右边三栏是换一个角度看同一件事。第一栏 ClassPool，里面码着几个 CtClass，一个类池管一堆类。第二栏 CtClass，装着字段、常量、方法三种成员。第三栏 CtMethod，列了四个插点：insertBefore、catchEx、insertAt、insertAfter，分别是方法开头、异常出口、指定位置、结尾。绿箭头从左往右，一层比一层细。

>

> 右边这套是高层接口，一句"在方法开头插一行"就够了。左边那段是 ASM，一条指令一条指令地摆。JaCoCo 走的是后者，探针要落在精确的偏移上。

>

> 一路下来改的都是字节码，源文件一个字节没动。

**提问：**插桩的时候为什么这么怕顺带加载别的类？

> **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"怕拿到错的版本"——对。不同加载器下的同名类，是两个类。

- 追问：别处见过这条约束吗？——APM 探针、热部署、字节码增强框架，都工作在类加载期，都得守。

- 收口：谁站在类加载这一环里，谁就不能再触发类加载。

**易错点 / 考点：**改字节码的库是 ASM。ClassWriter 传 reader 是为复用常量池，不是读源码。EXPAND_FRAMES 是遍历选项，不是插桩策略。"抛异常那一步"常被漏答，它防的是类加载期的死锁。右边

> 三栏的接口名和左边 ASM 不是一套，别混着答。

> **过渡：**类这一层过了，方法这一层怎么下针？

---

## 【5.72】on-the-fly instrumentation –续（2.5 分钟）

> 遍历走到某个方法，visitMethod 被回调。它这时还不动手。方法的全貌没走完，插不了。

>

> 左边这半段先做两件事。一是往下问一句：下游接不接这个方法？cv.visitMethod，参数是 access、name、desc、signature、exceptions。二是看回答。mv 是空，说明下游不接，拿 EMPTY_METHOD_PROBES_VISITOR 顶上，一个空实现，什么都不记。mv 不空，就存进 methodProbes。

>

> return 那一句返回的是 MethodSanitizer 的匿名子类。名字就叫清理器。有的编译器吐出来的字节码不规整，先收拾一遍，否则过不了后面的校验。

>

> 下针在右边 visitEnd。走到这儿，方法的全貌齐了，按顺序四件事。

>

> 第一件，LabelFlowAnalyzer.markLabels。把跳转标签标出来，看清哪条指令跳到哪。控制流的汇合点就在这些标签上。

>

> 第二件，new MethodProbesAdapter，把探针和标签对上，谁跟谁是同一对出口。

>

> 第三件，trackFrames 为真时再套一层 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。

>

> 第四件，accept。为真走 methodProbes.accept，注释写的是"注入数据分析"。为假就少套一层，直接 accept。

>

> 这里有个坑，很多人以为探针插得越密越准。不是。探针要挂在分支分开又汇合的地方。每条指令前面都插，类的体积和运行耗时都会爆，分支还会被重复计数。挂对地方，一个判定一个探针就够。

>

> 三段连起来是一条线：premain 装钩子，instrument 改类，visitMethod 和 visitEnd 改方法。六个维度全从这次遍历里带出来。指令和行，看探针落在哪。分支，看标签怎么配对。圈复杂度，看控制流图。工具没有魔法，它只是把一次遍历记下来的东西加起来。

>

> 读懂这一段，报告上那个黄格子就不神秘了。它背后是一对标签，只有一边被走过。

> **提问：**探针为什么非要插在控制流的汇合点上？

> **预设回答：**

- 答"每条指令都插更准"——方向对，代价太大。一个类能被插进成千上万个探针。

- 答"汇合点少，计数还对得上"——对。分支覆盖只问这个判定走没走两边。

- 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。

- 收口：能少插就少插，插在该插的地方。

> **易错点 / 考点：**三个层次要对上号。premain 是入口，instrument 是类这一级，visitMethod 与 visitEnd 是方法这一级。AnalyzerAdapter 与栈映射帧是为了过校验，不是在分析业务。MethodSanitizer 的活是清理不规整字节码。常见错答是"JaCoCo 往源文件里插统计代码"，它碰的一直是 class。

> **过渡：**针插进去了，报告长什么样？看一份真跑出来的。

---

## 【5.73】示例（2.5 分钟）

> 一份真跑出来的报告截图。上半部两张表，下半部一段染了色的源码。三处一起读，报告就读懂了。

>

> 第一张表，表头写着某个源文件的覆盖率汇总，文件名 EmailTemplateForm.java。五列：name、class %、method %、block %、line %。这一行数据是：类 100%，一个占一个；方法 95%，括号里 19 比 20；块 96%，64 比 67；行 97%，28 比 29。括号里不是百分比，是覆盖数比总数。读数先读括号，才知道差了几个。

>

> 第二张表叫按类和方法的覆盖率明细。列头还是那四组，下面把类展开成方法清单：getAt、getConfID、getKey、getMessage、getSiteUrl、getSubject、getTemplateName，后面还有两个构造方法。每个方法名前面一小截灰条，长度就是它的覆盖比例。

>

> 下半部右边是源码窗格，行号从 24 排到 45。底色三种。24 行浅黄，28 行绿，34 到 36 行绿，43 到 45 行整段红。黄色是部分覆盖，绿的是走全了，红色一次没跑。

>

> 四列里最该盯哪一列？方法那一列。95%，括号里 19 比 20，二十个方法差一个。往下翻就找到是谁：43 行那个带参数的重载构造函数，整段红。行覆盖 97%，差一行，差的也是这一处。两个数指着同一个地方，这就对上了。报告还能按覆盖率排序，排在最前的类，就是最该补的那个。

>

> 类那一列是 100%，别被它骗。类这一级最粗，类里任何一个方法跑过，这一类就算过。看到 100% 就当没事，那是最容易放过问题的地方。

>

> 还有一处要看仔细。上面表头写的是 EmailTemplateForm.java，下面源码窗格是另一个类 MailResourceLoader。看报告先对文件名，别把别处的源码当这一份。

>

> 据 GitHub 官方仓库 2026 年 11 月的发布记录，这套报告的列到 0.8.16 还是这个样子，HTML 里就是这几列。

**提问：**红格子一堆，全是 getter 和自动生成的代码；黄格子只有三个，都在核心业务方法里。先动哪边？

> **预设回答：**

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

- 答"先看黄的，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。

- 追问：怎么排除？——agent 的 excludes，或者在构建里配规则，把生成目录划出去。

- 收口：先划范围，再看数字。范围不划，两个项目没法比。

> **易错点 / 考点：**黄色是部分覆盖，通常是分支只走了一边，不是"跑得慢"。括号里是覆盖数比总数，说反就全错。类覆盖率最粗，容易虚高。读报告的顺序：先看方法列的缺口，再顺红行补用例。案例题问下一步，答三条：找出分支没覆盖的方法、排除生成代码、补走反方向的用例。

> **过渡：**命令行跑完才知道结果，太慢。编辑器里能不能直接看？

---

## 【5.74】代码覆盖率度量（2 分钟）

> 左边一栏红字 EclEmma，左下角一行蓝字把它和 JaCoCo 的关系说清了：EclEmma 是基于 JaCoCo 的 Eclipse 插件。引擎是 JaCoCo，外壳是这个插件。方向别记反。

>

> 中间是 Eclipse 的编辑器，Calculator.java，二十几行。行号左边一条细窄的彩条。7 行那条 if 是黄的，10 行、15 行、19 行、22 行是绿的。右边有人加了三条批注：黄色部分执行，红色没有执行，绿色全部执行。

>

> 15 行那句要停下来看。它在 substract 方法里，代码写的是 result 减 1，可它该减的是 n。旁边注释标着这里有 bug。这一行是绿的，跑过了，可它是错的。覆盖率回答"跑没跑过"，不回答"对不对"。判断对错靠断言。

>

> 左边是项目树：TestingAndEclEmmaDemo，下面 src 分两组，Tester 里是 CalculatorTest.java，TestTarget 里是 Calculator.java。再往下是 JRE System Library，版本 1.8，还有 TestNG 和一个 testing.xml。

>

> 底部是覆盖率视图。表头五列：Element、Coverage、Covered Instructions、Missed Instructions、Total Instructions。四行数：整个项目 86.2%，覆盖 50，漏 8，共 58；src 一样；Tester 84.8%，28 比 33；TestTarget 88.0%，22 比 25。最细一行是 Calculator.java。

>

> 右上角那个下拉菜单值得点开。里面一列是计数器：指令、分支、行、方法、类型、复杂度。跟前面那六个维度一一对上。切一下，整张表换一种算法。

>

> 用法分两头。写代码的时候，本地拿它跑一次，几秒就看见颜色，改起来快。要出正式报告、要做门禁，还是走构建里的 JaCoCo 插件，那份才代表整个团队。据 GitHub 官方仓库 2026 年 11 月的发布记录，引擎那边已经到 0.8.16。

> **提问：**流水线上已经有 JaCoCo 报告了，开发还在本地装这个插件图什么？

> **预设回答：**

- 答"没必要，重复了"——漏了等待。报告出来时，人往往已经在做下一件事。

- 答"有必要，本地改得快"——对。补一句：本地这张图不能拿去汇报。

- 追问：为什么？——它只反映你这台机器上这一次跑的用例，范围不一样。

- 收口：本地自检，流水线验收，一个引擎两套用法。

> **易错点 / 考点：**EclEmma 是基于 JaCoCo 的 Eclipse 插件，方向和年代都别答反，它早期挂过别的引擎，现在挂的是 JaCoCo。"本地覆盖率等于项目覆盖率"是错的。覆盖视图那五列要能说清哪列是漏的。还有一条：绿色不等于对，15 行那个减 1 就是证据。

> **过渡：**单个方法能测、覆盖率能看了。可五个人一起提交，谁保证拼起来还成？

---

## 【5.75】5.3 持续集成测试（1 分钟）

> 前面几节都在一台机器上。你写测试，你跑，你看覆盖率。换到团队里，问题会变样：五个人一天提交二十次，谁保证每次合进去以后，大家的代码还认得彼此？

>

> 这一节换三个问句。什么时候拼？从"发布前拼一次"改成"每次提交都拼"。谁来拼？机器，不靠人记得。拼完算什么？算一道门，过不去就不许合。

>

> 顺序也有讲究。先看单体架构怎么拼，自顶向下、自底向上、混着来，各自的桩和驱动怎么摆。再看微服务，那边拼的不是函数，是进程和接口。然后把拼这件事交给流水线，让它每次都自动跑。最后看一条完整的 CI/CD 流水线长什么样。

>

> 有一件事得先说清。单元测完只叫局部正确，集成测完才叫装配正确。这两件事互相替代不了。单元测试写得再漂亮，两个模块的字段对不上，联调照样崩。

>

> 这一节的出口是一句话：说清一条流水线该有哪几道关，每道关拦什么。说不清，工具装得再多也只是在刷绿灯。

**提问：**如果只能给流水线留一道关，你留编译、留单测、还是留集成？

**预设回答：**

- 答"编译"——它只证明语法和类型对，证明不了行为。

- 答"单测"——挡得住方法级的错，挡不住接口对不上。

- 答"集成"——最接近真实，可它要环境、跑得慢。

- 收口：三道都要，从快到慢排，越快越先拦。

**易错点 / 考点：**持续集成不是"装一个服务器"。核心是每次变更过同一套自动检查，不过就不能合。持续集成和持续交付不是一件事：前者保证随时可集成，后者保证随时可发布。

**过渡：**这一节四块内容，先看它们怎么排。

---

## 【5.76】持续集成测试内容（1.5 分钟）

左边四条编号，右边一个环。四条是路，环是节奏。

>

> 第一条，单体架构的集成测试。第二条，微服务架构的集成测试。这两条按架构分，同一个词，做法差得很远。第三条，持续集成及其测试。第四条，CI/CD 流水线。后两条按节奏和载体分，跟前面不是一类。

>

> 再看右边那个环。中间是 Continuous Integration 的标志，一圈深蓝，六个节点沿环排开。上方 Test，一个对勾；右侧 Build，一把扳手；左下 Coding，一副代码括号；下沿 Merge，一个分支汇合的图标；左侧 Report，一个饼图。走完一圈回到 Coding。环的下沿伸出一支浅蓝箭头，指向 Release，一个拇指图标。

>

> 顺序就在这张图上。写完代码先提交合并，再编译，再跑测试，再出报告，报告回到写代码的人手上。Release 在环外，它是环转顺了以后才发生的事。

>

> 容易看错的地方在这儿：Report 不是终点。报告不回到开发手里，这一圈就白转了。测试跑红、报告没人点开，流水线只是一个很贵的闹钟。

>

> 四条里第三条和第四条也不是并列。持续集成是做法，流水线是这套做法的载体。

**提问：**环上六个节点，哪个最容易被做成形式？

**预设回答：**

- 答"Test"——有可能。用例少、跑得慢，最后就被跳过了。

- 答"Report"——对。报告发出去没人看，红灯就变成背景噪音。

- 追问：怎么让它不形式？——把结果推给提交的人，别放在一个没人访问的页面里。

- 收口：一圈里任何一环没人接，整圈都停在那一环。

**易错点 / 考点：**四条的顺序要记：先单体、再微服务、再持续集成、最后流水线。Release 在环外，别答成环上第七个节点。CI 与 CD 不是一回事。

**过渡：**从第一条开始。单体架构怎么拼？

---

## 【5.77】单体架构的集成测试（2.5 分钟）

两棵树并排，中间一段小字，两边各一根橙色箭头，方向相反。左边那根朝下，底下红字：自顶向下。右边那根朝上，底下红字：自底向上。

>

先看左树。顶端 M1，往下三个孩子：M2、M3 是实线框，最右那个 S4 是虚线框。M2 往下 M5、M6，M5 下面再挂 M8。M3 往下只有一个虚线框 S7。

>

实线是已经写好的模块，虚线是临时补的桩。S4 和 S7 就是 Stub：上层要调下层，下层还没写，先拿一个假的顶在那儿，只返回约定的值。自顶向下从 M1 开始，一路往下，遇到没写的就摆桩。

>

树下面两行是遍历顺序。深度优先：M1 到 M2 到 M5 到 M8，折回 M6，再 M3、S7、S4。宽度优先：M1、M2、M3、S4，再下一层 M5、M6、S7，最后 M8。差别就一句：一条道走到黑，还是一层一层扫。深度优先能早点把一条链路验通，宽度优先能早点看见整体骨架。

>

再看右树。顶端 Mc，下面 Ma、Mb。Ma 下面挂两个虚线框 D1、D2，Mb 下面挂 D3。再往下是几排小方框，左边一个大括号标着族 1，中间一个大括号标着族 2。

>

这里的虚线换了身份，D 是驱动模块，Driver。自底向上从最底层开始，可最底层没人调用它，你就得自己造一个调用者，喂参数、收结果、比对。方向反过来，缺的东西也反过来：自顶向下缺下层，所以要桩；自底向上缺上层，所以要驱动。

>

两根箭头就是这两句话的图形版。往下走，桩在上面等人；往上走，驱动在下面推。

**提问：**自顶向下要写一堆桩，自底向上要写一堆驱动。哪一种更省事？

**预设回答：**

- 答"自底向上省，桩难写"——桩要模拟返回值，写不好就测出一个假结论。

- 答"看项目"——对。取决于上层接口稳不稳、下层能不能先跑。

- 追问：上层接口还没定怎么办？——那就先自底向上，把能定的底层锁住。

- 收口：桩和驱动都不是被测对象，别把它们当成果交。

> **易错点 / 考点：**桩 Stub 顶替被调用的下层，驱动 Driver 调用被测的上层。方向记牢：自顶向下配桩，自底向上配驱动，这里最爱出题，也最常答反。两种遍历顺序要能默写出来。

> **过渡：**两边各缺一样东西，能不能把两条路接起来走？

---

## 【5.78】单体架构的集成测试 –混合策略（2 分钟）

> 左边一棵树，A 在顶，下面 B、C、D 三个孩子，B 下面 E、F，D 下面 G。全实线，一个虚线框都没有。这是前提：模块都在，不用拿替身糊。

>

> 右边是一串圆圈，每个圈是一次测试。上面 Test E 和 Test F 两个圈，箭头汇进 Test B,E,F。中间 Test G 一个圈，箭头进 Test D,G。最下面 Test A 一个圈，箭头直接汇进最右那个大圈。大圈上写着 Test A,B,C,D,E,F,G，它收三条线：来自 B、E、F 那一组，来自 D、G 那一组，还有 Test A 那一条。

>

> 中间的竖向虚线把两边隔开，两个空心箭头夹在树上，上面那个朝下，下面那个朝上。方向在中间会合，这就是混合。

>

> 具体怎么走？底层 E、F、G 先自底向上。它们下面没有东西了，只能靠驱动往上推，所以先有 Test E、Test F、Test G。这一步验的是叶子自己的行为。

>

> 然后到 B 和 D 这一层，把 E、F 拼进 B，把 G 拼进 D，得到中间那两个圈。顶层 A 呢，从顶往下走，先单独跑一个 Test A，下面用桩顶着。

>

> 最后三个圈一起汇进大圈，做一次整体回归，A 到 G 全部合起来跑。

>

> 为什么不一路自顶向下？上层能早早验证，可下层全靠桩。桩一多，测出来的东西越来越像桩的行为。为什么不一路自底向上？桩是省了，可顶层那个总调度迟迟没人验，风险全压到最后。

>

> 混着来取两头的好处：最深那几层自底向上，桩少；最上面那层自顶向下，早暴露；中间在 B、D 这一层接上。接缝选在哪儿？选接口最少、最稳定那一层。

>

> C 只有一层，没有子模块，单独测完直接并进大圈。图里没给它画圈，因为它不需要额外拼一次。

> **提问：**混合策略里，那个"接缝"该选在哪一层？

> **预设回答：**

- 答"选在中间就行"——太粗。中间有很多层。

- 答"选接口最少、变化最少的那一层"——对。两边改动少，拼一次能管很久。

- 追问：选错了会怎样？——接缝天天动，等于每层都在重测，混合就退化成全量。

- 收口：先定接缝，再排顺序。

> **易错点 / 考点：**混合不是"两种都做一遍"，是在不同层用不同方向，中间会合。左边那棵树全实线，说明这一轮不用桩。最右那个大圈是整体回归，不是又一种策略。C 不画圈是有意的，单层模块不产生新的拼接点。

> **过渡：**单体的树画得出来，是因为模块都在一个进程里。换成微服务，树就没了。

---

## 【5.79】微服务架构（2 分钟）

> 这张框图从上往下一路走，就是一次请求的路。最上面 Client，两个图标，手机和笔记本。箭头往下一格，API Gateway，里面一个蓝标签写着 ZUUL。外部请求先到这儿，不直连服务。

>

> 网关往下分三根箭头，落到三个服务：Appointment Service、Patient Service、Prescription Service。

> 第一个框叠了两层，那是同一个服务起了多个实例。

>

> 每个服务框里都压着两个标签：RIBBON 和 HYSTRIX。RIBBON 管选谁：一个服务有五个实例，这次调哪个，它来挑。HYSTRIX 管兜底：下游卡住或者连续失败，把这条调用断开，别让上游跟着一起死。这两个不在网关里，在每个调用方身上。

>

> 三个服务往下有几根细线汇到一起，线上标着 register，箭头指向 Service Discovery，红标签是 EUREKA。服务启动先来这儿登记：我叫什么、我在哪个地址。左边还有一条长线从那块绕回网关，意思是网关问它要实例列表，拿到再去调。

>

> 右上角是另一条链。Auth Service，标签 KEYCLOAK，虚线箭头朝上连到 OAuth2 Service，那一格排着一串第三方图标。登录校验单独做成一个服务，第三方账号登录也走它。

>

> 最下面一排三个方框：Monitor Dashboard 配 GRAFANA，Log Analysis 配 ELK STACK，Config Service 配 SPRING CLOUD CONFIG。一个看指标，一个收日志，一个发配置。

>

> 有一件事图上没画：数据库。库被藏到每个服务后面了。这不是省事，是规则。服务之间要数据只能调接口，不能直连别人的库。图上不出现库，说的就是这件事。

**提问：**网关上挂的三个服务，其中一个慢下来，为什么最后整站都慢？

> **预设回答：**

- 答"网关是单点"——网关确实关键，但这张图上还有更直接的原因。

- 答"调用方线程被占满，一路往上堵"——对。这就是 HYSTRIX 要断开的理由。

- 追问：RIBBON 能救吗？——它只负责挑实例，挑到一个慢的，照样慢。

- 收口：没有超时和熔断，一个慢服务能拖垮一串。

**易错点 / 考点：**ZUUL 是网关，EUREKA 是注册中心，RIBBON 做客户端负载均衡，HYSTRIX 做熔断，

KEYCLOAK 管认证，五个标签别对错号。图上没有数据库节点，库在服务内部，跨服务取数只能走接口。GRAFANA、ELK、Spring Cloud Config 是三个辅助件，不是业务服务。

**过渡：**节点这么多，测什么才算测到要害？

---

## 【5.80】微服务架构的集成测试（2.5 分钟）

左边一个分层框图，右边两条编号，紫色箭头把两边连起来。整个框外围一圈虚线，斜纹描边，意思是这是一份部署单元。

>

从上往下读。Resources 绿色，最上面那层，对外暴露的入口。Service Layer 和 Domain 浅蓝，业务编排和领域逻辑都在自家进程里。Repositories 再下一层，管数据怎么取。最底 Data Mappers / ORM 粉色，箭头往下指进 External Datastore，也是粉色。中间灰字写着 ORM 的全称：Object Relational Mapping。

>

右侧还侧着两块紫色竖条：Gateways 和 HTTP Client。一条横线从它们伸出去，接到右边那个大紫块 External Service。

>

黄色虚线框圈了两处，正好是两条编号指的地方。上面那一处圈住 Gateways、HTTP Client 和 External Service。下面那一处圈住 Data Mappers / ORM 和 External Datastore。

>

① 说的是上面这处。测外部通信，先确认通不通，核心功能过一遍就行。它专抓协议层的错：HTTP 报头丢了、SSL 用错了、请求和响应对不上。这类错单元测试根本碰不到，因为它压根不出进程。

>

② 说的是下面这处。数据库访问测试，三件事。一是服务里用的数据结构跟库里的表对不对得上；二是 ORM 里配的映射关系对不对；三是网络抖一下，访问模块能不能兜住，别把异常直接甩给用户。

>

为什么偏偏圈这两处？中间那几层都在同一个进程里，单元测试和组件测试管得住。跨进程、跨库这两处才

> 是新长出来的失败面。清单不用长，但边界上这两处必须覆盖。

>

> 框外围那圈斜纹还有一层意思：这两处一跨出去，环境就不由你控制了。对端什么时候升级、网络什么时候抖，都不在你手上。

> **提问：**微服务集成测试里，为什么不必把每个服务内部都重测一遍？

> **预设回答：**

- 答"因为已经有单元测试"——对。内部逻辑在单元和组件那层就该钉住。

- 答"重测也测不完，成本不划算"——现实理由。服务一多，全量组合跑不动。

- 追问：省下来的力气花在哪？——花在边界：接口契约、协议细节、数据库映射。

- 收口：测新长出来的失败面，不测已经测过的。

> **易错点 / 考点：**① 是通信测试，抓协议层错误，三个例子要记全：报头丢失、SSL 用错、请求响应不匹配。② 是数据库访问测试，三件事：结构相符、ORM 映射、网络出错的处理。两条别答成"性能测试"和"安全测试"。

> **过渡：**架构看完了，回到那个问题：这些检查怎么每次都自动跑一遍。

# 第 5 章讲稿（5.81–5.90，v4 逐页重写）

> 每页先落到画面实体，再展开。无板书。口播 450–700 字，单句 ≤45 字。

---

### 【5.81】微服务架构的集成测试 – 续（2 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。错在中间那份没有写下来的约定。

>

> 标题下面一行橙色字，另一种观点：用契约测试或协议测试来做集成测试。左边接着讲这种测试叫什么。消费者驱动的契约测试，英文缩写 CDC。全称 Consumer Driven Contract Testing。

>

> 契约规定的是什么？左边说得很直接。接口的调用方和被调用方之间，约定的请求与响应格式。调用方叫 Consumer，消费者。被调用方叫 Provider，提供者。锁的是数据怎么交互，不是业务算得对不对，也不是快慢。

>

> 右边那段 Java 代码，就是一份契约长什么样。注解上先标消费者名字。方法里给一个场景起名，再给这次交互起名。接着写请求，路径 /pact，方法 GET。再写对方该回什么，状态 200，头里声明 json。身体里两个字段，condition 为真，name 是 tom。右上角蓝字四个字，一份契约，指的就是这段代码跑出来的东西。

>

> 留意这段代码里没有的东西。没有业务断言，没有金额怎么算，没有性能指标。它只写一件事，我发什么，你回什么。

>

> 最下面一行小字把顺序钉住了。从消费者的需求出发设计用例，产生契约，再去验证提供者。顺序反过来就不成立。提供者先宣布我支持这些字段，消费者照着写，那叫文档服从实现，不叫契约。

>

> 格式错为什么会变成空指针？字段名一改，消费者读不到，拿到空。崩的地方离原因很远，查起来要翻三层日志。这就是它值得单独写一份文件的原因。

>

> 还有两个词要分清。协议测试盯通信通不通，报头、SSL、状态码。契约测试盯约定的那一小片字段。前者管链路，后者管内容。

> **提问：**契约里只写了两个字段，condition 和 name。提供者实际回了八个字段。这份契约算不算写得太窄？

> **预设回答：**

- 答"太窄，应该写全"——写全就变成抄文档。抄得越多，对方一改你就红。

- 答"不窄，只用这两个就写两个"——对。契约锁的是依赖，不是全集。

- 追问：万一以后要用第三个字段呢？——那时改契约、跑验证，影响面查得出来。这就是它的用处。

- 收口：契约是消费者依赖的那一片，不是提供者能提供的全部。

> **易错点 / 考点：**CDC 的中英文都要能写全，Consumer Driven Contract Testing。契约锁请求与响应的格式，不锁业务逻辑。顺序是消费者需求、产生契约、再验证提供者。判断题"契约测试能发现业务计算错误"——错。

> **过渡：**这一份契约能干什么活，图上给了三条好处和一行工具名。

---

## 【5.82】契约测试及其优势、工具（3 分钟）

> 上一页把契约锁什么讲清了。现在回答两个问题，它能带来什么，谁来干这件事。

>

> 上面四条，前三条是好处，第四条是名单。第一条，降低服务集成的难度，把它分解成单元测试和接口测试。这句话要读慢一点。原来五个服务互相调，联调是五次乘五次的事。拆开以后，每个消费者对自己那份约定测一次，每个提供者验一次，变成加法。

>

> 第二条，团队能离线完成验证，联调成本几乎降到零。零的不是写代码，是约时间、占环境、等人回消息。第三条，接口的变更有迹可循。契约进了版本库，谁改了什么，翻提交记录就能看。

>

> 第四条只有一行字。常用的框架，Janus、Pact、Pacto 和 Spring Cloud Contract。注意这一行是并列点名。图上没有给这四家各画一个格子，也没有标谁强谁弱。分工不在这行字里，分工在下面那张图上。

>

> 看中间那条虚线。虚线左边是消费者这一侧。蓝色 Consumer，旁边一个灰色小方块，写 Test。请求箭头从它发出，打进绿色的 Provider Mock，这一块左边标着 API。回来的箭头写着 expected response。

>

> 虚线右边是提供者那一侧。绿色 Contract Verifier，旁边也标 Test。请求箭头写 expected request，打进蓝色 Provider，同样标着 API。回来的箭头写 response。

>

> 两半靠什么连起来？靠最上面那份 Contract 文档。两个向上箭头，一头从 Provider Mock 出来，一头从 Contract Verifier 出来，都指向它。左边产出它，右边消费它。图里没有第三根线，说明两边不直接通话。

>

> 两个灰方块都写 Test，位置不一样。左边那个测消费者自己的代码，对面是假的。右边那个测提供者真实的服务，请求由契约生成。同一次约定，两头各跑一次，谁都不用等谁上线。

>

> 四个名字怎么记？Pact 是后面几页要拆开的主线，文档在 docs.pact.io。Spring Cloud Contract 用契约文件生成验证测试，八十五页下方那个 contract.groovy 是它的写法。Janus 和 Pacto 与 Pact 同一族，课件只点名，不展开。

>

> 名单会换，动作不换。写约定，生成桩，两头各验一遍。

**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。契约测试拦得住吗？

**预设回答：**

- 答"拦得住，字段名在契约里"——对。改名就是破坏约定，右边那个验证器会红。

- 答"拦不住，值还一样"——要改口。对消费者，字段名就是接口。读不到就是空。

- 追问：怎么安全改名？——先加新字段，两边并行一段，消费者迁完再删旧的。

- 收口：图右半那个验证器，拦的就是这种"功能没变"的破坏。

**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，产出契约的是消费者侧，验证的是提供者侧。Provider Mock 在左，Contract Verifier 在右，两个灰方块别对调。

**【思政落点 4】**：契约测试把"我答应你"变成了一份能自己跑的检查。消费者写下我要什么，提供者拿真服务去对，两头都不用在群里喊一句应该没问题。工程协作里最贵的成本，是彼此以为对方知道。把口头承诺写成明文，再让机器替你守，这是职业信用最实在的样子，也是对同事时间的尊重。

**过渡：**道理讲完了，看真实工具链怎么把这几块接起来。

---

## 【5.83】示例：CDC 测试框架（2 分钟）

> 标题右边那个黑色标志就是 Pact。左下角一行地址，docs.pact.io。整张图要看的是几条箭头，和它们各自的方向。

>

> 先看最上面那个圆柱，写着 maven repository。旁边一个蓝色文件，Stubs.jar。这一版把契约仓库落在你们熟悉的制品库上，不是另起一个服务。好处很直接，权限、版本号、谁能拉、拉了哪一版，都是现成的。

>

> 左下白色框，subscription-service，括号 consumer。它右边灰色小方块 tests，双向箭头，连着绿色 stub server。这一段说的是，消费者跑测试时，对面是桩服务端，不是真服务。

>

> 右边绿色方块 contract verifier tests，箭头打到灰色 API，再进白色框 account-service，括号 producer。这一段是真服务在应考。

>

> 三条长箭头，方向都不重复。resolve 从上面那个 jar 往左下，指向 stub server，意思是把桩取下来用。publish 从 contract verifier tests 往右上，回到那个 jar，意思是验证产物发布出去。auto-generate 从最下方黄色文件 contract.groovy 往上，指向验证测试。

>

> 连起来读一遍。契约文件自动生成验证测试。验证和桩制品发布进仓库。下一轮谁要，再从仓库取。仓库在中间，两头都不用直接对接。

>

> 那个黄色文件为什么单独画出来？因为它不是代码里随手写的期望。它是一份单独的契约描述，改它等于改约定，要走评审。

>

> 右边一列九行图标，写着各种语言。Node 与 JavaScript。Java 与 JVM 语言。.NET。Golang。Python。Swift 与 Objective-C。Ruby。PHP。C++。这一列想说的只有一件事，一

> 份契约，各种语言都能来验。桩往往每种语言各写一套，契约只有一份。

>

> 一个用词要对齐。图里 producer 和前几页的 provider 是一回事，两个词换着用。别当成两个角色。

>

> 还有一处细节值得停一下。仓库在中间，就多了一道版本对齐的活。消费者的桩和提供者的验证，必须取同一版契约。取错了，两边都绿，结论却是错的。所以版本号不是装饰。

**提问：**仓库里那份 Stubs.jar 一直没更新，消费者的测试却全绿。问题可能出在哪一环？

**预设回答：**

- 答"桩是旧的，测试对着旧桩跑"——最可能。契约改了，jar 没重新发布。

- 答"publish 那一步没接进流水线"——也对。手动的步骤一定会被忘。

- 追问：怎么发现？——提供者验证要跑在同一个契约版本上。版本对不上，就该红。

- 收口：图上三条箭头，断哪一条都会变成假绿。

**易错点 / 考点：**三段要能复述，契约自动生成验证测试、桩与验证产物发布进仓库、消费者用桩测自己。三条箭头名字要能对应方向，resolve、publish、auto-generate。producer 与 provider 同义。跨语言是契约的天生优势，不是额外配置。

**过渡：**链看全了，放大左半那一段，看消费者这边四个动作的顺序。

---

## 【5.84】Consumer 测试（2 分钟）

> 上一页把整条链画全了。现在只看左边那一半，因为这里最容易看错。

>

> 图里三层。最左橙色大块，标着 Consumer。里面一条橙色竖条，写 Unit test，上面圈 2，下面圈 4。中间蓝色大块，顶上一行 Pact Mock Provider，中间圈 3。最右一个虚线框，蓝色标题条写 Interaction，里头上下两条，Expected request 和 Minimal expected response，旁边圈 1。最上面横着一个黄标签，Stub server，罩住中间和右边两块。

>

> 数字就是顺序，按它走一遍。圈 1，先声明期望。这次交互，请求该长什么样，回什么。圈 2，消费者自己的单元测试发出真请求。实际请求那一块标着 Actual request，打进 Mock Provider。圈 3，Mock Provider 拿实际请求去比期望请求，蓝色小标签写 Compare。圈 4，按最小期望应答回给消费者，代码拿到响应再断言。

>

> 所以测的是消费者，不是提供者。对面那个服务从头到尾不存在。

>

> 那个 Compare 为什么在 Mock Provider 里？因为比对是工具干的活，不是你的断言。你的断言看的是业务结果，比的是我自己拿到的响应能不能用。两件事分开。

>

> 下面那段代码是 Ruby。前半段描述交互。收到一个带事件的 POST 请求，路径斜杠 events，头里 json，身体是事先备好的 event_json，回 200。后半段把客户端代码真的跑一遍，断言 save_event 返回真。

>

> 两个词要抠。Stub server 是真起一个 HTTP 服务，占一个端口，按你声明的规则应答。它不是把某个方法替换掉。走真 HTTP，序列化、报头、状态码才有机会露出来。另一个词是最小期望应答，只写你真正会读的字段。

>

> 这一侧有个天生的盲区。期望写错了，它照样绿。因为它回的就是你写的。

> **提问：**你把期望里的状态码写成 200，真实接口其实回 201。这一侧的测试会不会红？

> **预设回答：**

- 答"会红，和真实不符"——不会。假服务端只按你写的回。

- 答"不会红，这就是盲区"——对。它只证明你能处理你以为的响应。

- 追问：谁来补这个洞？——提供者那一侧，真实响应和契约不符就红。

- 收口：两头都跑，才叫闭环。只跑一头，是自说自话。

> **易错点 / 考点：** Stub server 起的是真 HTTP 服务，不是方法级 Mock。消费者测试测的是消费者。契约由测试跑出来，不是手写文档。四个圈对应四步，声明、发请求、比对、应答断言，顺序不能答反。

> **过渡：** 这一侧低着头测自己。对面那个真服务，下一张图才上场。

---

## 【5.85】Provider 测试（2 分钟）

> 把镜子翻过来。这一侧要拿真服务去对契约。

>

> 图基本是上一页的镜像，但有一处不同。最左还是那个虚线框 Interaction，里头仍是 Expected request 和 Minimal expected response，中间竖着标 Auto-generate。左下角黄色字 Contract.groovy，一条虚线箭头指进这个框。意思是这些期望由契约生成，用例不手写。

>

> 中间蓝色大块换了名字，Pact Mock Consumer。顶上一行浅蓝字，Contract verifier tests。看名字就明白，替身换到对面，这回扮的是假消费者。上一张里是假提供者，这一张里是假消费者。

>

> 最右橙色大块，Provider，真的服务。

>

> 箭头方向也跟着反。期望请求从左边出发，蓝色 Send，进 Mock Consumer，再打给 Provider。Provider 被调用，旁边写 Call API，回来的是橙色 Actual response。最后在 Mock Consumer 里 Compare，拿真实响应去对最小期望应答。

>

> 为什么替身要扮消费者？因为这次要让真服务收到一个请求。请求从哪来不重要，重要的是它和你约定的那个一模一样。所以由契约生成，不靠人手敲。

>

> 代码是 JavaScript。引一个 Verifier，配好选项，调 verifyProvider，回来再做后续处理。选项里填服务地址和契约来源。地址在画面最下面，pact-foundation 那个 pact-provider-verifier 仓库。

>

> 这一侧不用知道消费者是谁，也不用知道对方用什么语言。它读契约，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。

>

> 失败了会说什么？哪一条契约、哪个字段、期望什么、实际什么。这四样信息比一句失败有用得多。

>

> 一句必须钉住。验证要跑在你准备发布的那个构建上。数据库和配置用真的。再替一层，验的还是假的，等于白跑。

>

> 和上一侧对照着记最省力。左边测自己能不能用，右边测自己有没有变。左边怕写错期望，右边怕改坏行为。两句话，把两个角色分开了。

**提问：**提供者验证红了。是提供者改坏了，还是消费者把契约写错了？

**预设回答：**

- 答"提供者改坏了"——多数是。行为变了，没通知人。

- 答"也可能是消费者照想象写的"——存在。真实接口从来不是那样。

- 追问：怎么判？——看谁的需求更合理，然后把契约当接口文档评一次。

- 收口：红了先别改代码，先看契约里那一条是不是真需求。

**易错点 / 考点：**消费者侧用 Stub server 产出契约，提供者侧用 verifier 验证契约。提供者的用例是自动生成的，Auto-generate 就是这意思。不需要把所有消费者启动起来。不要求两端同一种语言。Pact Mock Consumer 站在提供者这一侧，别和 Provider Mock 混。

**过渡：**两头分开看完，放回一份真实代码里对一下位置。

---

## 【5.86】示例（2 分钟）

画面中间一条绿色竖线，把屏幕分成两栏。左边那栏浅灰底，字偏淡，是自动生成的测试类。类名

ContractVerifierTest，继承一个基类。

>

> 方法名一长串，validate 后面跟 shouldReturnEvenWhenRequest，读起来是，请求某种输入时应该返回偶数。

>

> 方法体里四个注释块，given、when、then、and。given 里准备一次模拟请求。when 里带上参数 number 等于 2，发 GET 到斜杠 validate 斜杠 prime-number。then 断言状态码。and 取出响应体，断言它等于 Even。四段正好是准备、动作、结果、细节。

>

> 记住这一段不是人写的。它由契约生成。你改契约，它跟着改；你不改，它就一直是老样子。

>

> 右边那栏是一个控制器。类名 BasicMathController，头顶两个注解。里面注入一个 RestTemplate，暴露一个 /calculate 的 GET 接口，收一个 number 参数。它做的事是，拿这个 number 拼一条地址，去调本机 8090 上的那个校验接口，把对方回的字符串原样交回去。

>

> 注意 exchange 那几个参数。地址、GET、一个带头的实体、还要声明返回类型是字符串。这种写法最容易在头的类型上出错，写成 text 就解析不出 json。

>

> 底下两组标注。左边写 Server side 冒号 service provider，提供者这一侧。右边写 Client side 冒号 service consumer，消费者这一侧。consumer 那里课件少打了一个字母，答题按正确拼法写。

>

> 读法是这样。右边这个控制器在调别人，它是消费者。左边那段自动生成的测试，站在提供者这边验。中间那条绿线，就是两套代码的边界。

>

> 一条边界上只有一份约定。谁要变，先动那份契约，再让两边的测试说话。

>

> 还有一层读法。左边那四个注释块，其实是契约里的四样东西落到了代码里。场景、请求、状态、身体。看到 given、when、then、and，就该想到契约里那几行。

>

> 两栏摆在一起看，还有一件好事。这份测试永远不会和接口漂开，因为它就是从接口描述长出来的。手写的那一类，写着写着就旧了。

> **提问：**右边这个控制器调的是本机 8090。这样的代码，消费者测试该怎么写？

> **预设回答：**

- 答"直接起一个真服务在 8090"——能跑，但依赖别人的环境和数据。

- 答"起桩，声明 8090 上该回什么"——对。这就是 5.84 那四个圈。

- 追问：桩里那个 Even 从哪来？——从契约来。不是随便写的字符串。

- 收口：地址写在代码里，本来就该由约定来定它回什么。

> **易错点 / 考点：**左边是自动生成的验证测试，四个注释块 given、when、then、and 是它的骨架。右边控制器是消费者。Server side 对应 provider，Client side 对应 consumer。课件把 consumer 拼成 concumer，答题写对。

> **过渡：**契约里的字段名，不能靠回忆。下一张图给出正经来源。

---

## 【5.87】如何获取接口信息（2 分钟）

> 字段名从群里的截图抄，桩就会停在旧接口上。看一个正经的来源。

>

> 这是一整屏浏览器。地址栏里是 localhost 加 9095，后面跟着 swagger-ui 加一个锚点，机具管理。

>

> 左边主体是一份接口目录。大标题，系统 API 服务，旁边一个小徽标。底下写 Base URL，localhost 9095。再一行链接，v2 斜杠 api-docs。那一行才是给机器读的结构化描述，界面是照它渲染出来的。下面

> 一个下拉，Schemes，选的是 HTTP。

>

> 目录分两组。第一组，机具管理，机具管理业务接口类。五条，每条左边一个色块标方法。POST 的 bind，请求绑定网点接口。GET 的 join，入网报备查询接口。POST 的 join，请求入网接口。POST 的 report，报告位置接口。POST 的 version，APP 版本更新检查。第二组，登录，用户请求登录获取 Token，一条 POST login。

>

> 右边浮出两块展开的面板，两条橙色箭头从目录指过去。上面那块是某条接口的详情，参数表两列，名字和说明。头里的 Content-Type，查询参数里的 loginToken，还有请求体，带一段示例 JSON。下面那块是响应表，两列，Code 和 Description。200 成功，401 权限校验失败，404 接口不存在，每行还配示例。

>

> 那两张表要会读。参数表告诉你，这个接口除了地址还要什么。响应表告诉你，它可能回哪些码，你该为哪几个写用例。这就是契约初稿的骨架。

>

> 要带走的不是界面。是这份描述从代码里生成。控制器上写了注解，工具扫一遍就出文档。代码改了，文档跟着改，不会出现文档写旧版、代码跑新版。

>

> 文档和契约还不是一回事。文档声明我准备提供什么，契约规定你依赖什么。文档是原料，契约是从原料里切出来的那一小片。

> **提问：**有了这份自动生成的文档，是不是就不用写契约了？

> **预设回答：**

- 答"不用了，文档很全"——文档只声明，不验证。对方改坏了，它不会红。

- 答"要写，文档只是原料"——对。而且只摘自己真正会读的字段。

- 追问：摘多少？——用到几个写几个。多摘一项，维护成本先把这个实践拖垮。

- 收口：文档管有什么，契约管我靠什么活着。

**易错点 / 考点：**文档由代码生成，不是另维护的手写说明。OpenAPI 是规范名，Swagger 是这套工具链的名字，别混。文档不等于契约，一个声明，一个验证。契约粒度只写用到的字段。图上那个 api-docs 链接就是结构化描述本身。

**过渡：**文档和契约都还没有的时候，今天要测的东西还得测。

---

## 【5.88】利用 Mock 技术解除微服务之间的依赖（3 分钟）

B 归别的组，C 还没写完，A 今天就得测。等它们都在线，这组测试永远起不来。看这张图怎么摆。

>

左边红字斜体，API Test，一对方箭头指向橙色圆角框，微服务 A，括号 SUT，被测系统。测试从外面打进来，打在 A 的边界上。

>

A 向两个方向分叉。右上一条灰箭头，打进绿色平行四边形，Mock Service B。右下一条，打进蓝紫平行四边形，Mock Service C。这两块是替身，顶在前面。

>

再往右看，两个灰框。上面写微服务 B，下面写微服务 C。它们和那两个替身在同一条竖线上，只是被挡在后面。意思是真服务还在，这次不叫它。

>

右边一段深色 JSON，标题写着 Mock 请求和响应信息示例。外面一对 request 和 response。request 里两个字段，method 是 GET，urlPathPattern 是斜杠 api 斜杠 users。response 里 status 二百，body 里 success 为真，msg 是查询列表成功，data 两条，David 三十岁，Susan 二十八岁。

>

这段 JSON 只看两件事。请求怎么匹配，回什么内容。多了没用，少了不够。它和契约里一次交互是同一个粒度。

>

替身顶在 A 控制不了的那一端。B 的人不归你管，C 可能明天才提测。A 自己的库、缓存、消息队列，别一

> 起替掉。替掉那一层，测的就不是集成，退回单元测试了。

>

> 三条收益要能说出口。可控，B 回 500、回超时、回脏数据都行。快，不起 B 不起 C，不连它们的库。进度解开，B 没做完，A 照样能测。

>

> 还有一条风险。替身的报文是照当时的 B 写的。B 后来改了，替身不知道，测试继续绿。这个洞只有一个补法，把报文收成契约，让 B 那一侧去验。

>

> 两条风险合起来就是一句话。替身越多，离真实越远。所以替身要少，边界要清楚，报文要有人对。

> **提问：**外部依赖全替掉了，测试天天绿，一上线就出问题。根子在哪？

> **预设回答：**

- 答"替身写错了"——有可能。更常见的是它从没和真服务对过。

- 答"没有一环在验真实接口"——这是根子。缺的不是替身，是提供者那一侧的验证。

- 答"自家数据库也替掉了"——那样测的就不是集成。价值没了。

- 收口：问三句。报文是不是契约，契约有没有在对方验，自家依赖是不是真的。

> **易错点 / 考点：**SUT 是 System Under Test，被测系统。解除微服务依赖靠测试替身，粒度是请求和响应这一对报文。替身只放在你控制不了的远端。三条收益，可控、快、进度解开。两条风险，漂移和失真。

> **【思政落点 5】**：让测试变绿很容易，让绿变成真话很难。把上下游全替掉，屏幕上天天通过，自己心里最清楚那是安慰。工程里最贵的不是不会，是拿一个漂亮的数字骗自己。给自己留一句追问：这一条绿，是谁替我验的？答不上来，就还没验。求真比求绿难，也才值钱。

> **过渡：**依赖解开了，这些测试什么时候跑，不能靠谁想起来。

---

## 【5.89】CI/CD 与持续测试（1 分钟）

> 微服务这一段到这里收口。契约、桩、替身、接口文档，手里都有了。接下来换一个问题。

>

> 屏幕中间一条深蓝横带，白字一行，CI、CD 与持续测试。这是这一段的入口，不是结论。右上角是校徽。

>

> 上面那些内容回答的是，一项一项怎么测。从这里开始问的是，这些测试什么时候跑，跑不过能不能拦住人。

>

> 为什么要换问法？前面那些做法，你们组里多半都听过。不写，常常不是不会写。是没人逼着每次都写。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。

>

> 所以这一段解决的是机制。提交之后自动发生什么，哪一道门能让人过不去。还会把几个以持续开头的词分开，因为它们分不清，门就会对错。持续交付和持续部署就差一步，答错了整道题都歪。

>

> 最后落到持续测试要的几个特点，还有一张完整的流水线。学完要能把这一章的技术排到一条时间上，每项在哪一段跑，跑完卡什么。答辩时问你们质量怎么保证，你要能指着某一格说话。

>

> 再往下会看到一张图，五个圆排在一条线上，箭头全朝左。那张图回答的是压力从哪来。还会看到流水线的完整环节，门禁在哪一格。

>

> 课前先想一件事。你们组从改完一行代码到能给别人演示，中间有几步是自己动手点的。手动点得越多，越需要这一段。

>

> 说清一句，这一段不考运维命令。考的是测试怎么才真的发生。

> **提问：**单元测试写得很规范，只在发版前跑一次。算不算落地了？

> **预设回答：**

- 答"写了就算"——不算。跑得越晚，问题越贵。

- 答"得进流水线，每次提交都跑"——对。这就是这一段要解决的事。

- 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。

- 收口：写下来不等于会发生。能让它发生的是机制。

> **易错点 / 考点：**这一段在课里的定位是测试的执行与门禁，不是运维专题。四个以持续开头的词要能分开。判断题"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。

> **过渡：**先把这几个词摆到一条线上，看压力从哪头来。

---

## 【5.90】CD 倒逼持续测试（3 分钟）

> 五个圆排在一条横线上，箭头全朝左。方向是这张图最要紧的地方。

>

> 最右深绿，持续交付。往左一个，蓝色持续部署。再往左，中间红色，持续测试，外面套一个绿色粗圆环，环上写 Continuous Testing。继续往左，蓝色持续集成。最左，蓝色持续构建。

>

> 箭头从右往左指。压力从交付那头进来，一路压到构建。这就是倒逼。名字虽然叫持续测试，图上它站在中间，被两头夹着。

>

> 绿环上有六个点，每个点外挂一行英文。右侧，需求与验收标准。右下，把功能映射成用例并准备数据。下方，设计并开发自动化用例。左下，执行自动化用例。左上，反馈回开发循环。上方，缺陷处理与分派。

>

> 六个点是转的，不是排的。它说明持续测试不是一个关卡，是一个循环。验收标准想不清楚，用例就写不出来。用例不能自动跑，就进不了流水线。跑完没人接缺陷，下一轮还是同样的问题。

>

> 四个词要分开。持续集成，是频繁把代码合进主干，每次合都编译都测。持续构建，是每次代码一变，就能出可运行的制品。持续交付，是随时有一份能发布的制品，发不发由人定。持续部署，连最后那一下也交给

> 机器。

>

> 交付和部署只差一条，上生产是不是自动的。这条辨析每年都考。

>

> 还有一句。组装了流水线工具，测试形态没变，人还是发版前跑一遍。工具只把结果摆出来。快不快、准不准，是测试自己的事。

>

> 你们课设现在最多三个人。不必搭一条真的流水线，但可以有一条最小的：合并前自动跑一遍单元测试，红了不许合。这一格守住了，后面才好加门。

> **提问：**已经做到持续部署，合并过了门禁就自动上生产。测试得满足什么？

> **预设回答：**

- 答"必须全自动"——必要，不够。全自动却跑两小时，还是跟不上。

- 答"要快、要准、要拦得住"——完整些。分钟级反馈，只跑受影响的，门禁不过不能上。

- 答"还得能回滚"——对。自动部署要配自动回滚和监控。

- 收口：倒逼的意思不是多按一次发布，是整条链都得跟上那个节奏。

> **易错点 / 考点：**持续交付与持续部署的差别只在最后一步是否自动。五个词要能各自说清。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。环上六点常考顺序，验收标准在最前，缺陷处理接回开发。CI 不等于装一个流水线工具。

> **过渡：**词摆到了线上，下一步看这些门按什么顺序排，才能让人早一点看见错。

# 第 5 章讲稿（5.91–5.98，v4）

> 逐页重写。每页先落到图上的实体：几节管子、几个方框、几条条目。取消板书。口播四拍：起 → 推 → 收。

---

## 【5.91】CI/CD 流水线：七个阶段与一条回流（3.5 分钟）

上一页那五个词排成一行，箭头都朝左。词认清了，换个问题：这套东西真跑起来，长什么样？

>

课件给了一根横着的管子，分七节，节与节之间是一圈法兰。从左数。Source Stage，源码。Build Stage，构建。Team Test Stage，团队测试。Deploy for Business Test Stage，为业务测试部署。Business Test Stage，业务测试。Deploy for Production Stage，为生产部署。Production，生产。

>

七个名字别硬背，看它们怎么分堆。管子上头有两条蓝色箭头带。左边写 CI，括号里是 Build Pipeline。右边写 CD，括号里是 Release Pipeline。两条带在管子中间偏左接上，接缝正好落在第三节和第四节之间。所以前三个归 CI，后四个归 CD。三比四，分界在团队测试和业务测试之间。

>

再看最上面那条深紫色带子。它写 Continuous Testing，两头带尖，宽度几乎盖满底下两条蓝带。位置很讲究。它不插在七节里的任何一格，它铺在最上层，压住整根管子。测试不在某一段，测试横跨所有段。Team Test 在 CI 那半边，Business Test 在 CD 那半边，两头都有测试。

>

管子底下还有一根。七节每一节都伸出一个向下的小箭头，接到下面那根管子上。这根管上写 Continuous Monitoring / Feedback，箭头朝左，一路退回起点。两端各有一个弯头，把回流接回入口。发布之后的监控数据，从这里回到下一轮。

>

那条回流不是装饰。没有它，前面的门只能凭老经验定：跑什么、跑多深，全靠猜。有了它，线上真出过的那类问题，下一轮才写得进用例。

>

那这根管子能当部署图看吗？不能。它标的是检查点。每一节都在回答一句：走到这里，必须验什么才算过。

**提问：**七个阶段里，生产环境是终点吗？

**预设回答：**

- 答"是终点，发上去就完了"——管子上它确实是最后一节。可底下那根回流管，箭头是从生产往回走的。

- 答"不是，后面还有监控和反馈"——对。回流管上写着 Continuous Monitoring / Feedback，箭头朝左，一路退到入口。

- 答"那回流算第几节"——它不属于任何一节。七个小箭头都往它身上收，它接在每一节底下。

- 收口：七节管"往哪走"，回流管"往回带什么"。少了后半根，前面只能凭经验。

> **易错点 / 考点：**七节的英文名要能对上中文。最关键的一句是分界：CI 到 Team Test 为止，CD 从 Deploy for Business Test 起。第二个考点：持续测试铺在最上层，不是七节里的某一格。第三个：回流管的名字，Continuous Monitoring / Feedback，方向朝左。

> **过渡：**管子是示意。真跑起来是一棵分叉的树，还多出一格管子没画的：质量门禁。

---

## 【5.92】CI/CD 流水线：触发、分叉与质量门禁（4 分钟）

> 真跑起来的流水线不长成直管。它从左往右推，中途分叉，再合回来。

>

> 左端竖着一个长条，写排队限流。它右边四个小方框上下排开：代码提交、定时触发、事件触发、人工触发。四条线并成一条，进"编译"。四个入口就是四种"什么时候跑"。提交最常见，一推就跑。另外三种补的是提交管不到的情况：环境变了、依赖换了、要单独重跑某一次。只配第一种，另外三种会漏。

>

> 编译是一个虚线圆。它上方挂两个方框，虚线连着：编译结果预推送、代码跨数据中心同步。两样都不在主线上。多地一起构建，拿到的得是同一份代码。产物提前送到环境，后面就少等一段。

>

> 编译往右是"测试"，也是虚线圆。它往下拉出一条竖线：单元测试，再往下自动化测试。它往右分成两支，一支上行到静态扫描，一支下行到安全扫描，两支在下游又合成一条实线。这里有个坑：扫描排在测试之后，不在编译之后。编译只把代码变成能跑的东西。

>

> 看安全扫描。它下面挂一个方框，虚线连下来，写 Sonar 增量扫描。为什么是增量？全量扫一遍常常几十

> 分钟，每次提交都扫撑不住。只看这次改动，才塞得进这条线。

>

> 合回来之后是三个圆，实线串着：部署 STG、质量门禁、发布。质量门禁夹在中间，是整张图上最要紧的一格，因为它是唯一一个"汇总之后再判断"的位置。编译、扫描、单测的结论都收到这里，够了才往下走。单个检查失败会自己停，不用门禁开口。门禁管的是另一句：都过了，够不够放行。

>

> 发布后面换成浅色。发布、灰度环境、生产环境，三个圆竖着往下走。底部方框写制品预推送，虚线指向生产环境。右边缘竖着一条长框写动态扩容，虚线回到上方，构建高峰时临时加机器。

>

> 主干记一条：触发 → 编译 → 测试 → 部署 STG → 质量门禁 → 发布 → 灰度 → 生产。

> **提问：**质量门禁为什么排在部署 STG 后面，不排在单元测试后面？

> **预设回答：**

- 答"因为部署更重更要紧"——不是轻重的问题。门禁不做单项判断。

- 答"因为门禁要等结论齐，STG 里也在验"——对。它是一次总判断，总判断得等证据到齐。

- 答"那单元测试失败谁来拦"——那一格自己就是门，红了就停，不需要门禁开口。

- 收口：单项各有各的门，门禁管"齐了以后够不够放行"。两件事别混。

> **易错点 / 考点：**主干顺序必考：四种触发 → 排队限流 → 编译 → 测试（静态扫描、安全扫描、单元测试、自动化测试）→ 部署 STG → 质量门禁 → 发布 → 灰度 → 生产。两个高频错：把静态扫描与安全扫描放到编译之后（它们在测试之后分两支）；漏掉挂在安全扫描下面的 Sonar 增量扫描。四个旁支也别漏：编译结果预推送、代码跨数据中心同步、制品预推送、动态扩容。

> **【思政落点 6】**：门禁写进配置只要一行，红了敢不敢停才难。赶演示那天，阈值往下调一格，全组都松一口气，用户不会知道。质量这件事没人替你验，守住的往往是没人看见的那一次。绿灯要自己跑出来，不能靠放宽标准换。

> **过渡：**门排好了。持续测试要算达标，课件给了四个说法。

---

## 【5.93】持续测试的四个特点（3 分钟）

> 四个标题并排，字号一样大，前面各一个灰圆点。并排就说明一件事：这四条平级，不是一级一级往下走的顺序。

>

> 挨个看。第一条，足够快。底下正文说的是整个测试过程要快，一头靠高度自动化，另一头靠业务端到端的探索式测试。两头都要。全自动不等于够用，脚本盖不住的路径还得人去看。

>

> 第二条，准确且有效。正文说被测系统往往很复杂，全回归做不了，要推精准测试。为什么做不了？不是懒。系统一大，用例涨得比代码快，发布还越来越密。硬跑全量，最后一定退化成发版前跑一次。

>

> 第三条，平滑有序。正文一句打通整个测试过程，后面跟三对：测试左移到测试右移、单元测试到系统测试、静态测试到动态测试。用词是打通，不是覆盖。三对是三条缝，缝两边原本各干各的。

>

> 第四条，与 CI/CD 集成。正文四行，跟研发的持续构建、持续集成、持续部署、运维这些环境接上。四个名字都在：构建、集成、部署、运维。

>

> 四条里，前三条讲测试自己怎么干，第四条讲测试站在哪儿。位置这条最容易漏，也最容易白干。测试写得再快再准，不挂进构建，就没人替你跑。它不进流水线，等于不存在。

>

> 还有一处排版能说明问题。第四条标题在右边被挤成两行，"与 CI/CD 集成"最后一个字单独占一行，前面还断在 CI/CD 中间。格子等宽，字不够放就折。它们没有谁包含谁，是四个并列的格子。

> **提问：**四条只能留一条，你留哪条？

> **预设回答：**

- 答"足够快"——快是前提。慢的测试会被绕过，绕两次就没人再等。

- 答"与 CI/CD 集成"——我认这条。它决定前三条会不会真的发生。

- 答"准确且有效"——精准要有依据。说不出这次为什么只跑这些，精准就是全回归换了个名字。

- 收口：能力三条都在测试这边，位置那条在测试外面。位置丢了，能力没有出场机会。

> **易错点 / 考点：**四条名称要能写全：足够快、准确且有效、平滑有序、与 CI/CD 集成。平滑有序底下三对别串：左移与右移、单元与系统、静态与动态。不要把精准测试答成"少写几条用例"，要答"按改动的影响范围来选"。

> **过渡：**四条是目标。真做起来，架子上摆着三层东西。

---

## 【5.94】持续测试实践框架：三层八个框（2.5 分钟）

> 三条横带，从上到下是橙、蓝、绿。每条带左边立着一个名字。橙带写技术手段/方法，蓝带写中间过程，绿带写实现基础。名字靠左，白框靠右，一条带一层。

>

> 数框。橙带里三个白框：探索式测试、测试自动化、测试设计方法。蓝带里两个：基于风险的测试分析、测试影响分析。绿带里三个：测试数据管理、DevOps 工具链集成、服务虚拟化。三、二、三，一共八个。

>

> 中间那层只有两个框，是最容易被跳过的一层。上下两头都在讲手段和基础，中间这层回答一句更硬的话：这一轮要测哪些。基于风险，是照风险高低排优先级。影响分析，是照这次改动圈出范围。少了这层，自动化就成了把所有用例再跑一遍。

>

> 箭头值得停一下。三条带之间不是一根箭头往下指，是双向箭头，上下两头都有尖。橙带和蓝带之间两根，蓝带和绿带之间三根。双向的意思：上层的方法要靠下层撑着，下层建什么又由上层要什么拉出来。按单向读，会读成"先建基础，再谈方法"，那就把顺序定死了。

>

> 绿带写在最下面，名字还叫实现基础。因为这一层最不显眼，缺了最疼。测试数据管理没有，用例只能手搓，数据一改全红。DevOps 工具链不集成，上一页那条流水线就停在概念上。服务虚拟化没有，第三方接口一

> 停，整批用例跟着停。

>

> 右边一个二维码，下面两行字：扫码下载《持续测试白皮书》。

>

> 这张图合上以后可以当自检表。三层各问一句：我们的手段是什么？中间的分析谁在做？基础那三样齐不齐？三句里有一句答不上来，就是缺口。

**提问：**三层里，通常哪一层最先被砍？

**预设回答：**

- 答"绿带，基础的三个"——最常见。它不出现在任何一次演示里，砍了当天没人喊疼。

- 答"蓝带，中间过程"——也常见。分析这层没有交付物，看起来像开会。

- 答"橙带最贵，先砍它"——砍了手段，测试直接停，这刀没人敢下。

- 收口：被砍的总是"当下看不见"的那层，而三层缺一层，另外两层跟着退化。

**易错点 / 考点：**三层与框要配得上：技术手段（探索式测试、测试自动化、测试设计方法）、中间过程（基于风险的测试分析、测试影响分析）、实现基础（测试数据管理、DevOps 工具链集成、服务虚拟化）。箭头是双向，不是单向流程。白皮书是参考材料，别说成强制标准。

**过渡：**三层摆完，回全章收拾一遍。

---

## 【5.95】本章小结：五条与一条线（3.5 分钟）

开始收拾。五个条目，每条前面一个蓝色空心方框，一条占两行。数是五条，不是六条。

>

> 五条的写法有个共同点：前半截说"理解什么"，后半截说"做什么"。第一条，理解代码规范和评审的价值，然后一直做下去，方式是互为评审、集体评审，团队有多方面收益。第二条，分析工具今天很流行，两个落点写在一句里：开发本地扫代码，和与 CI/CD 流水线集成。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，理解单元测试的价值和 xUnit 的实现机制，重视单元测试，掌握 JUnit 这类工具和

> mock 的技巧。机制清楚了，换一门语言的 xUnit，上手快得多。第四条，单元测试够不够，看语句覆盖和分支覆盖，一般交给覆盖率工具自动算。第五条两个半句：持续测试促持续交付；契约测试值得关注，从开发自测开始。

>

> 五条这样背会散。收成一条线：评审、单元测试、Mock、覆盖率、契约、流水线，一环扣一环。要紧的是每一环各挡什么。

>

> 评审挡"人能不能读懂、意图会不会被读错"。它不运行程序。单元测试挡"这个方法自己跑对不对"。Mock 挡的不是缺陷，是"依赖不在的时候还能不能测"。覆盖率挡"有没有代码从没被走到"，它管遗漏，不管断言写得软不软。契约挡"两个服务对同一个字段的理解是不是一致"。流水线挡"改坏了没人知道"，它自己不做检查，它让前面几项每次提交都发生。

>

> 每一环只能挡一类。用错环，等于没挡。拿覆盖率证明代码是对的，拿 Mock 证明集成没问题，都是把一环当成了全章。

**提问：**六环里，哪一环最容易被当成"已经做过了"？

**预设回答：**

- 答"覆盖率"——最像。报告一出来有百分比，人就想拿它交差。它只管走没走到。

- 答"Mock"——也常被误认。替身写得漂亮，拼起来照样对不上字段。

- 答"评审"——最容易走过场。开过会不等于看过代码。

- 收口：越容易出数字的那环，越容易被当成结论。数字是线索，不是结论。

**易错点 / 考点：**五条要能写全，别答成六条。第四条两个限定别丢：语句与分支、自动分析。论述题按"评审 → 单元测试 → Mock → 覆盖率 → 契约 → 流水线"组织，每环说清挡什么。不要拿覆盖率当正确性证明，也不要用 Mock 替代集成验证。

**【思政落点 7】**：这六环是一条接力。前一环松一寸，后一环要多跑十倍。人常想找一个最强的工具，把质量一次解决掉。现实是分工：谁挡哪一类，说清楚，别越位，也别缺位。守住自己那一环，就是最实在的专业。

> **过渡：**收拾完了，往下是动手。实验 2 只交测试代码。

---

## 【5.96】实验 2：单元测试（3.5 分钟）

> 任务只有一句话，横在标题下面，占两行。用 JUnit，针对 Spring Unit Testing 里的 ItemController 类写测试类，最后交测试代码。三个要件：JUnit、ItemController、交测试代码。

>

> 下面是一张 IntelliJ IDEA 截图，深色主题，版本 2022.2.3。左边项目树，中间代码，右边 Gradle 和通知栏。先说清一件事：图里打开的类是 RxJava 项目里的 SchedulerTest.java，不是你们的 ItemController。它给的是一个合格测试类长什么样。

>

> 看左边。src/main/java 和 src/test/java 上下并列，包名一样，都是 io.reactivex.rxjava3.core。测试代码和被测代码同包不同目录，这是标准位置。test 目录下面挂一长串 XxxTest，和 main 那边的类大致对得上。

>

> 看中间。类名叫 SchedulerTest，文件名和类名一致，命名规则是被测类加 Test。第一个方法上面标着 @After，方法体两行：把静态开关改回 false，注释写的是为了不影响别的测试。这就是清场。第二个方法标着 @Test，名字念一遍就知道在验什么：默认没设成纳秒计时。里面两条断言，一条查静态字段初值，一条查那个系统属性没被设过。

>

> 四样可以直接抄：位置放 src/test/java；类名加 Test；@After 复原共享状态；方法名写出意图。

>

> 提交物：页面上明写的只有一样，测试代码。课上再加两样，一张分支覆盖截图，一页短说明，写清测了什么、没测什么、为什么。

>

> 测什么也说清。别去测 Service 的业务算法。控制器只看三件：参数绑不绑得上，Service 回什么它就回什么，异常和错误码对不对。查询要有数据也要无数据；新增要合法、缺必填、类型不合法；改和删要有存在

> 和不存在的对象。断言不能只写不是空，要有状态码、有关键字段，再用 verify 看 Service 被调了几次。不许改被测代码换绿灯。

> **提问：**图里那个 @After 为什么值得抄？

> **预设回答：**

- 答"为了格式好看"——不是格式。它管的是用例之间的独立性。

- 答"防止一个用例改了全局状态，后面全红"——对。图里那个静态开关就是这类东西。

- 答"我每个用例里自己复原，一样的"——行，但漏一次就是一个说不清的红。

- 收口：共享状态要么别用，要么每个用例前后都复原。测试之间互相污染，红绿灯都会说谎。

> **易错点 / 考点：**测试类放 src/test/java，命名 XxxTest。最常见失分是跑去测 Service 的业务算法。断言太软（只判非空、只判没抛异常）也算没做完。用 WebMvcTest 时它不加载 Service，忘了 MockBean 会启动失败。为了让红变绿去改被测代码，这是纪律问题，也影响评分。

> **过渡：**动手会卡住。资料放在课后，按用途取。

---

## 【5.97】学习资源推荐（1.5 分钟）

> 五行，前三是书，后两条是网址。颜色分开：三行书名是深灰，两行网址是蓝色带下划线。一眼能看出哪些要买，哪些要打开。

>

> 三本书按"读完能解决什么"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。讲的是开发人员自己怎么写测试，和这学期的实验对得上，现在就能读。第二本，《敏捷测试》，人民邮电出版社，2021 年 8 月。讲团队怎么在短迭代里守住质量，接的是这一章后半的持续测试，实验做完再读更顺。第三本，《Java 微服务测试》，电子工业出版社，2019 年 7 月。契约、消费者驱动、测试金字塔在微服务里怎么变形，都在这本的范围里。做微服务方向的课设再细读。

>

> 两个站点比书更常打开。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。

>

> 补一句最新的。据 GitHub 官方仓库 2026 年 11 月的发布记录，JUnit 已经发到 6，最新一个版本是 r6.1.3，2026 年 8 月 7 日，仓库也改名为 junit-team/junit-framework。课件这条链接指向的仍是 JUnit 5 的文档。写实验用 5 没问题，读的时候心里有个数：主线已经在往 6 走。

>

> 顺序建议先文档、后书。文档更新快，默认你知道要解决什么。书给的是顺序和背景。微服务那本放最后，单元测试和契约没底子，读起来会空。

**提问：**只有一周，三本书两个站点怎么分？

> **预设回答：**

- 答"先开文档，边做实验边查"——最实用。带着一个红的用例去查，记得住。

- 答"读第一本书的前几章"——可以，要配着手写。只读不写，一周后什么都不剩。

- 答"直接上微服务那本"——底子还没立。契约和单测不清楚，书里的做法对不上你的代码。

- 收口：卡在哪就查哪，别按顺序通读。文档解决当下的错，书解决"先问什么"。

**易错点 / 考点：**两个站点要记得：junit.org/junit5，docs.pact.io。三本书按"读完能解决什么"记，比记出版年份有用。补充的版本号是课外信息，答题时别拿它替换课件说法。

> **过渡：**材料留在课后。这一章收尾。

---

## 【5.98】感 谢 聆 听（1 分钟）

> 这一章从一行代码走到了整条流水线。评审让人读懂。静态分析按规则扫。单元测试把方法钉住。Mock 把控制不了的依赖挪开。覆盖率标出没人走过的分支。契约把服务之间的约定写成文件。流水线让这些每次提交都跑一遍。

>

> 这一路没有一步靠天分。评审要的是第二双眼睛，单测要的是敢写一个会红的用例，门禁要的是红了之后不找借口。都是能练出来的习惯，练的是每次都做，不是偶尔做对。

>

> 交课设之前问自己一句：这段代码，除了我，还有谁能讲清它为什么这么写。答得出来，这一章就没白学。

>

> 以上是本节课的内容。这一章讲到这里。下一章换到系统那一层，看性能和安全在压力下还对不对。

>

> 谢谢大家。

> **提问：**这一章里，哪一样最可能在你第一个项目的第一周就用上？

> **预设回答：**

- 答"单元测试和 Mock"——最直接。进项目第一周就在写。

- 答"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"——微服务团队里很值钱，但要组里一起做，一个人写不起来。

- 收口：只能先练一样，练单元测试。后面几样都站在它上面。

> **易错点 / 考点：**这里不设名词考点。作用是把全章收住，指向下一章的系统测试，并提醒课下交实验 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 7 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 | 5.82 | 契约测试把"我答应你"变成了一份能自己跑的检查。消费者写下我要什么，提供者拿真服 |

| | | |
|---|---|---|
| 5 | 5.88 | 让测试变绿很容易，让绿变成真话很难。把上下游全替掉，屏幕上天天通过，自己心里最清 |
| 6 | 5.92 | 门禁写进配置只要一行，红了敢不敢停才难。赶演示那天，阈值往下调一格，全组都松一口 |
| 7 | 5.95 | 这六环是一条接力。前一环松一寸，后一环要多跑十倍。人常想找一个最强的工具，把质量 |