# 《软件测试课程设计》讲稿（第 5 章）

第 6 周 · 2 学时（90 分钟）

广东金融学院 计算机学院 | 周宇文

讲稿可照读版 | 由课件逐页撰写

# 第 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）】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为时间紧张时可压缩页。

---

## 一、教学目标

| 维度 | 目标（课后可检验） |
| --- | --- |
| **知识** | ① 能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）＋板书＋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**；其后是板书、提问与预设回答、易错点/考点、过渡语。板书建议边讲边写，不要一次写完。
>
> 页码锚点与课件页一一对应（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）

同学们，我们翻到新的一篇了——**第 2 篇，软件测试技术**。

上一篇讲的是概念、流程和规范，它回答的是"测试这件事该不该做、按什么规矩做"。可真坐到工位上、打开一个项目，你遇到的第一个问题不是流程，而是：**我手上这段代码，究竟怎么测？**

这一篇就来回答这个问题。它一共五章：**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。

你把五章连起来看，就是一条从"最小零件"到"整条流水线"的路：先把一个函数测住，再把模块拼起来测，然后测整个系统的功能与性能安全，最后用自动化把这些活交给机器。

今天上的第 5 章，是这条路的第一段。第一段没打好，后面全白费。

> **板书：**竖写"第 2 篇 技术篇"→ 下方五章：单元与集成（5）→ 系统功能（6）→ 专项（7）→ 国际化本地化（8）→ 自动化框架（9）；左端标"零件"，右端标"流水线"。

> **提问：**这五章里，你觉得哪一章最"看得见成果"、哪一章最"看不见但最要命"？

> **预设回答：**

- 答"第 9 章自动化最看得见"——确实，自动化跑起来满屏绿灯很直观；但它测什么，取决于前几章设计的用例。
- 答"第 5 章最看不见"——对，单元测试写完谁也不夸你；可它是后面所有层次的底座。
- 答"专项测试最要命"——性能和安全出事最贵。但它依赖单元和集成阶段留下的可测性。
- 收口：**看得见的是结果，看不见的是地基**，这一章讲地基。

> **易错点 / 考点：**本篇五章的顺序不是随便排的，考试常问"哪一章讲集成测试""哪一章讲自动化框架"。记法：**从零件到整机到特种检验到出口到无人车间**。

> **过渡：**既然这一章讲地基，我们先把这一章的名字记住——单元测试与集成测试。

---

## 【5.2】软件测试方法和技术（1 分钟）

这一页是本章的封面，我借它说三件事：这门课、这一章、以及这一章要替你解决的那个"卡点"。

课程是《软件测试课程设计》，我们用的是朱少民老师《软件测试方法和技术》第 4 版。**第 5 章 单元测试与集成测试**——请注意这个名字是两段："单元"是把一个函数、一个类单独拎出来钉死；"集成"是把这些零件装到一起，看它们还认不认对方。

为什么要单开一章讲这两段？因为现实中出问题最多的，恰恰是这两段的中间地带：单个函数都对，一拼起来就错；或者单元层次全是漏洞，一集成全线崩。

这一章学完，你应该能独立做一件事：**拿到一段陌生代码，写出第一批能跑起来的测试，并且说清"我为什么测这几组数据"。**

这句话就是你这一章的学习目标，请你抄在笔记第一行。

> **板书：**中间写"第 5 章 单元测试与集成测试"；上写"单元：先钉死零件"；下写"集成：再拼成整机"；右侧写本章出口一句话"给定代码 → 写出测试 → 说清理由"。

**提问：**如果只能选一件事作为本章的"毕业标准"，你会选"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"——能跑是底线，工具一周就会，但会跑不等于会测。
- 答"说清理由"——这才是我想听到的。会跑的人多，能说清"为什么测这组数据、为什么漏了那组"的人少。
- 追问：那你觉得"说清理由"需要哪些知识？——用例设计、覆盖率、以及对缺陷类型的敏感度，本章都会给。

**易错点 / 考点：**章名里的"单元"不是"一个文件"，而是**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"，而是**把单元按设计装配起来并测试装配结果**。这两个概念每年都有人答错。

> **过渡：**先把教材的这一章摊开看一遍，你就知道我们要走多少路。

---

## 【5.3】教材内容（2 分钟）

这一页是教材第 5 章的目录，八节。我们先按顺序念一遍，然后我告诉你哪几节是一组。

第一节**代码静态测试**、第二节**代码评审案例分析**、第三节**代码静态检测工具**——这三节是一组，共同点是：

**它们都不运行代码**。靠人看，靠工具扫，在程序跑起来之前就把毛病挑出来。

第四节**单元测试的目标和任务**、第五节**分层单元测试**、第六节**单元测试工具**——这三节是第二组，共同点是：**它们要运行代码**，把最小单位拉出来单独跑一遍，看它答不答得对。

第七节**系统集成的模式与方法**、第八节**持续集成及其测试**——这两节是第三组，共同点：**它们关心的是"拼"**，而且是"拼了还要天天拼、每次改动都拼一次"。

所以八节看着多，其实就是三块：**不跑代码先挑错、跑了代码钉零件、拼起来还得一直拼**。

你把这三块的分界线记住了，整章的骨架就立住了。教材是按八节写的，我们上课会按三块讲，一会儿那页会讲怎么对上。

再补一句读教材的方法：**教材的节标题，往往就是这一节的结论**。比如"分层单元测试"这五个字，已经告诉你单元测试不是一层，是分层做的；"持续集成**及其**测试"这个"及其"，暗示了持续集成落到测试上有自己的一套做法。你带着这个习惯去读，八节读起来就快得多。

还有一点要提醒：教材写的是"静态测试"和"静态检测工具"，不是"静态的测试"，这两个词别念混——**静态的是被测对象的状态（没运行），不是测试的手段**。

> **板书：**左侧竖写教材八节编号 5.1–5.8；右侧用三条横线切成三段：①静态（5.1–5.3）②单元（5.4–5.6）③集成（5.7–5.8）；每段末尾写一个字："看""跑""拼"。

**提问：**为什么教材把"代码评审"这种看起来"很土"的人工活动，放在"静态检测工具"前面讲？

**预设回答：**

- 答"因为它更早出现"——时间上是，但顺序的真正理由是**逻辑顺序**：先知道人会看什么、看哪些维度，才知道该让工具去自动化哪些维度。

- 答"因为工具是评审的自动化"——很好，这句话是本章的一条暗线。工具检查的东西，几乎都能追溯到评审关注点。

- 追问：那反过来，工具能替代评审吗？——不能，工具只认规则，评审还能问"这段代码为什么存在"。

**易错点 / 考点：**这八节的名字常被考成填空或匹配题。最易错的是把"5.5 分层单元测试"当成集成测试的内容——它讲的仍然是**单元层次内部的分层**（比如先测最底层算法、再测上层的服务类），不是模块之间的集成。

**过渡：**教材八节，我们只有九十分钟。所以课件做了整合，怎么整的，下一页说清楚。

## 【5.4】本章结构：三部分五环节（2.5 分钟）

这一页是我们这一章的**行程表**。我把教材的八节压成三部分，再补上小结和练习。

第一部分，**5.1 代码评审与分析**——对应教材前三节。学完你要能做到两件事：知道评审该盯哪些点（规范风格、圈复杂度、信息流复杂度、耦合度），以及会用一组工具把能自动查的部分查掉。

第二部分，**5.2 单元测试**——对应教材四到六节。这是本章的重心，也是你实验要动手的部分：目标是什么、驱动程序和桩程序怎么用、JUnit 5 怎么写出第一条用例、Mock 什么时候该上、覆盖率报告怎么看。

第三部分，**5.3 持续集成测试**——对应教材最后两节。它把"集成"从一次性动作变成一条流水线：代码一提交就编译、就跑测试、就过门禁。

后面还有两块收尾：**小结**帮你把重点难点归位，**练习**就是基于 JUnit 的单元测试实验——那是要交代码的，不是交感想。

为什么敢这么压？因为这三部分各自对应一个问句：**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？** 三个问句答完，这一章就学完了。

再说时间怎么分。第一部分我给你们十二分钟左右，因为评审的关键点必须当堂讲透，尤其"互查那几条硬指标"——每次看多少行、用什么速度看、看多久，这些是要你以后真照着做的。第二部分最长，二十分钟上下，因为它是实验的直接依据：JUnit 怎么写、Mock 什么时候用，实验课上没人再给你从头讲。第三部分十分钟，重点是"策略选哪一种"和"契约测试解决什么问题"，工具配置可以自己看。最后留几分钟讲小结和实验要求。

也就是说：**今天听不懂的地方，你要当场问；今天听得懂但没动手的地方，实验课上必须补回来。**

> **板书：**横排三格：①代码评审与分析（看）→ ②单元测试（跑）→ ③持续集成测试（拼）；下方两行："小结：重点难点归位"、"练习：JUnit 单元测试实验（交代码）"；右侧竖写三个问句。

> **提问：**三部分里，哪一部分你以前在别的课里接触过、哪一部分完全没碰过？说说你印象里"单元测试"是干什么的。

> **预设回答：**

- 答"没碰过单元测试，以为是测试人员的事"——很典型。单元测试的第一责任人恰恰是**写这段代码的开发人员**，测试人员更多是定标准、看覆盖率、补关键用例。

- 答"写过 JUnit，就是 assert 一下"——入门了，但"assert 一下"背后有 AAA 结构、有边界数据选择、有替身技术，本章会把它们补齐。

- 答"持续集成听过，就是用 Jenkins"——工具只是外壳，内核是"每次变更都必须过同一套自动检查"。

- 收口：这三块里，第二块你要动手，第一块你要养成习惯，第三块你要理解它在团队里怎么落地。

> **易错点 / 考点：**本章的"三部分"是**课件的教学整合**，教材仍是八节。答题时如果题目问"教材第 5 章包括哪八节"，按八节答；问"本章分几部分讲"，按三部分答。两套口径不要混。

> **过渡：**好，从第一部分开始——代码评审与分析。

---

## 【5.5】5.1 代码评审与分析（1 分钟）

> 我们进入第一部分，**代码评审与分析**。这一部分有一个共同的前提：**代码一行都不用跑**。
>
> 你别小看这个前提。程序不跑，意味着你还没有环境、没有数据、没有依赖——也就是说，**这是你能最早介入的时刻**。代码刚写完，甚至刚提交，你就能看。
>
> 这一部分有三个关键词：**评审**，是人看，同行坐下来读你的代码；**分析**，是工具看，静态扫描器按规则把可疑模式挑出来；**规范**，是照着看，先约定"什么算好代码"，评审才有共同语言。
>
> 记住一句话：**评审解决"人能不能看懂、会不会想错"，分析解决"规则能不能自动兜住"。** 两者不是替代关系，是前后脚关系。
>
> 接下来几页，我们先把"评审值不值"这个问题答掉——因为很多同学心里其实是怀疑的：花时间看别人代码，图什么？

> **板书：**写"5.1 代码评审与分析"，下画三个圈：人看（评审）／工具看（分析）／照着看（规范）；右下角写"前提：不运行代码 → 介入最早"。

> **提问：**不运行代码就能找缺陷，你觉得这件事靠谱吗？凭什么？

> **预设回答：**

- 答"不靠谱，不跑怎么知道错"——这是最常见的直觉。但缺陷分两类：一类是"跑出来才知道"（性能、并发），一类是"读一遍就知道"（空指针没判、资源没关、日志打了密码）。后一类占比很大。

- 答"靠谱，因为很多错误看就能看出"——对，而且越早看越便宜。

- 追问：那评审能替代测试吗？——不能。评审只能证明"我没看出问题"，不能证明"它是对的"。这句话后面还会用到。

> **易错点 / 考点：**评审属于**静态测试**，是测试活动的一部分，不是"开发内部的事"。考试常出判断题："代码评审不属于软件测试"——错。

> **过渡：**既然评审属于测试，那它到底能带来什么？我们看它列出的价值。

---

## 【5.6】代码评审的价值（2.5 分钟）

> 这一页讲"评审到底图什么"，一共列了六条价值，我一条一条讲，但请你注意它们的层次不一样。
>
> 第一条，**改善代码质量**。这是最直接的：别人读你的代码，最容易发现"你没想过的输入"和"你写顺手了的坏习惯"。
>
> 第二条，**跨团队共享知识**。这条容易被忽略。一个模块只有一个人懂，是项目最大的风险；评审逼着第二个人把这段代码读进去，风险就摊薄了。
>
> 第三条，**辅导缺少经验的程序员**。新手最有效的成长方式不是听课，是有人在他代码上写具体的评论："这里为什么要判空？如果传进来的是 null 会怎样？"
>
> 第四条，**坚持遵守代码规范**。规范写在文档里没人看，写在评审意见里就必须改。
>
> 第五条，**增强协作**。评审是一种低成本的、以代码为中心的技术交流，比开会有效。
>
> 第六条，**减少项目代价**。这是总账：缺陷在这一层被拿掉，就不必留到测试、上线、甚至用户投诉那一层去付账。
>
> 六条合起来一句话：**评审买到的不只是"少几个 Bug"，还买到了"团队对这段代码的共同理解"。** 后者的价值，往往比前者大。
>
> 我再讲一个反面情形，你就知道为什么值得。一个团队不评审，会发生什么？代码写完直接提测，测试人员拿到一个黑盒，只能按需求文档试；代码里那些"我看一眼就知道有问题"的东西——比如一个异常被 `catch` 之后什么都没做、比如一个循环里每次都去查数据库——测试用例根本碰不到，它们会一路带到线上。等到线上出了事再回头定位，你得先把这段没人读过的代码重新读懂一遍。

所以评审的本质是**把"读代码"这件事提前、并且让第二个人读**。它便宜，是因为读代码比调试代码便宜得多。

> **板书：**中间写"代码评审"；六个分支：质量／知识共享／带新人／守规范／协作／降代价；在"降代价"旁标"← 总账"，在"知识共享"旁标"← 最易忽略"。

**提问：**六条里，哪一条是"老板最不在乎、但对团队长期最重要"的？

**预设回答：**

- 答"知识共享"——我同意。老板看的是这次交付，知识共享的收益在半年后才显形，可它决定了"这个人走了项目会不会瘫"。

- 答"辅导新人"——也很重要，但要注意：评审不是培训课的替代品，它是"在真实代码上做辅导"。

- 答"减少代价"——这是最容易被接受的理由，也是最容易量化的理由。

- 收口：**对外讲"降代价"，对内讲"知识共享"**——跟不同的人沟通，用不同的价值主张，这也是工程沟通的一部分。

**易错点 / 考点：**常考多选题"代码评审的价值包括哪些"，六条尽量记全。易错点是把"发现缺陷"当成唯一价值——只答这一条会丢分，因为评审在**知识传递和规范落地**上的作用同样被教材强调。

**【思政落点 1】**：代码评审最珍贵的地方，是它把"我的代码"变成"我们的代码"。写代码的人天然有防御心理——被人指出问题，第一反应是解释而不是修改。可工程上的事实是：**没有人能独自保证正确**。愿意让别人看、愿意承认自己没想到、愿意把评论当成帮助而不是挑刺，这种开放心态，是技术人最难练也最值钱的品格。团队协作不是一句口号，它就落在你收到评审意见时的那一次选择上：先改，再辩。

**过渡：**既然要评，评什么？总不能凭感觉说"我觉得这段不好"。下一页给出四个可操作的关注点。

---

## 【5.7】关注代码评审（3 分钟）

评审要看什么？这一页给了四个关注点。我一个一个说，前两个是重点，后两个理解思想就行。

第一个，**是否符合代码规范、代码风格**。这一条最基础，也最容易自动化。命名是不是有意义、一个方法有没有写成长篇大论、缩进和括号是不是统一。风格问题不影响程序正确性，但它影响**可读性**，而可读性直接影响后面所有的维护成本。

第二个，**圈复杂度**，公式是 $e - n + 2p$。e 是控制流图的边数，n 是节点数，p 是连通分量数，一般一个方法

就是 1。算出来的这个数，含义非常实在：**它等于这个方法的独立路径条数，也就是"最少要写几条测试用例才能把每条路走一遍"**。我给你一个手算的例子：一个方法画出来 9 个节点、10 条边，那么 10 − 9 + 2 = 3，说明它有三条独立路径，你最少的用例数就是三。行业里常用的经验阈值是**一个方法的圈复杂度不超过 10**，超了就该拆。

第三个，**信息流复杂度**，课件给的式子是**扇入乘扇出再平方**。扇入是"有多少个上层模块调用我"，扇出是"我调用了多少个下层模块"。为什么要平方？因为上下两头的复杂度是互相放大的：一个被二十处调用的方法，只要它自己还依赖五六个服务，改它一处，风险就横跨一大片。**接口越宽，改动越贵。**

第四个，**模块耦合性**——看功能上、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。

四个关注点其实在说同一件事：**评审不只看"这句写得对不对"，还要看"这个结构以后好不好活"。**

> **板书：**四条竖列：规范风格／圈复杂度 e−n+2p／信息流复杂度（扇入×扇出）²／耦合性；圈复杂度下面写手算例"e=10, n=9, p=1 → 3 条路径"；最右侧写一句话"结构决定维护成本"。

**提问：**一个方法圈复杂度等于 1，说明什么？它是不是就"没问题"了？

**预设回答：**

- 答"只有一条路径，不需要测试"——**这是最危险的误解**。圈复杂度只数**判定分支**，它不管数据。一个没有任何 if 的方法，照样可能因为除零、空指针、类型溢出出错。

- 答"结构简单，但仍要测数据和边界"——完全正确。圈复杂度低只说明"路径少"，不说明"数据安全"。

- 答"圈复杂度等于 1 就是没有 if"——基本对，但循环、`case`、`&&`／`||` 的短路也算分支，会把数字抬上去。

- 收口：所以圈复杂度是**用例数量的下限估计**，不是质量证明。

**易错点 / 考点：**① 公式必须记准：**e − n + 2p**，考试常直接把 e、n、p 给你让算，最常错的是漏掉 `2p` 或把 p 当成 0；② p 是**连通分量数**，单个方法通常取 1，所以很多教材写成 e − n + 2；③ 判断题"圈复杂度越高程序越复杂、越难测试"——对；"圈复杂度低就一定没有缺陷"——错。

**过渡：**四个关注点里，第一个最好落地，因为已经有现成的规范可以照抄。下一页就是两份。

---

## 【5.8】首推代码规范（2 分钟）

这一页只给两个链接，但它的意思比链接重要：**规范不要自己发明，先用业界成熟的。**

课件首推的是 Google 的代码规范，给了 C++ 和 Java 两份：C++ 那份在 google.github.io/styleguide/cppguide.html，Java 那份是同一站点下的 javaguide.html。为什么首推它？三个原因。

第一，它**足够细**。细到什么程度？命名怎么起、一个函数多长、注释写在哪、异常怎么用、甚至哪些 C++ 特性禁用，它都写了。细，才可执行；"代码要优雅"这种话是没法执行的。

第二，它**足够权威且公开**。公开意味着你的团队可以直接引用："我们按 Google Java 规范"，不用再花两个月吵命名该用驼峰还是下划线。

第三，它**天然适配工具**。规范一旦写死，就能变成 Checkstyle 的配置文件、变成 IDE 的格式化模板——这一点后面讲 Checkstyle 时会接上。

我的建议是：**团队规范 = 业界规范 + 少量项目约定**。业界规范解决 90% 的争议，剩下 10% 你们自己补一条"本项目额外约定"就够了。别一上来就写一份三十页的部门规范，写完没人看，等于没有。

> **板书：**写"规范三原则：够细 → 可执行；公开权威 → 不吵架；可配置 → 能进工具"；下行写"团队规范 = 业界规范（Google C++／Java）＋ 项目补充"。

**提问：**如果你们团队已经在用另一套规范（比如阿里的 Java 规范），还要不要换成 Google 的？

**预设回答：**

- 答"要换，Google 更权威"——没必要。规范的价值在于**统一**，不在于"哪家更正确"。已经统一了就别折腾。

- 答"不用换，能用就行"——正确方向，但要检查一件事：这套规范**有没有配套的自动检查工具**。没有工具支撑的规范，落地率极低。

- 追问：那如果一个项目里一半人按 A 规范、一半人按 B 规范呢？——这才是真正的问题，评审会变成风格吵架。先统一，再谈优劣。

**易错点 / 考点：**考试可能问"代码规范的作用"，标准答法要落到**可读性、可维护性、团队一致性、可自动化检查**四点，不能只答"好看"。另一个易错点：把"代码规范"和"编码标准/安全编码标准"混为一谈——前者管风格与结构，后者管"哪些写法会引发漏洞"。

> **过渡：**规范有了，评审怎么组织才有效？这一页的四条硬指标，是本章最该背下来的操作细节。

---

## 【5.9】代码互查的优秀实践（3 分钟）

这一页是本章最"能直接用"的一页，标题叫"代码互查的优秀实践"，说的是**同行互查怎么做才不流于形式**。我按课件给的几条讲，每条都给你翻成一句人话。

第一条，**一次检查少于 200 到 400 行代码**。为什么？因为人的注意力是有限的，超过这个量，后面看的都是"扫一眼就放过"。你要是拿着一千行 diff 说"帮我看看"，对方只会回你一句"看着挺好的"。

第二条，**努力达到一个合适的检查速度：每小时 300 到 500 行**。注意这个数字很反直觉——它意味着看四百行代码要花将近一个小时。**评审不是浏览，是逐行推演**：这个变量此时可能是什么值、这个分支进得去吗、这个异常谁接。

第三条，**有足够的时间、以适当的速度、仔细地检查，但不宜超过 60 到 90 分钟**。超过一个半小时，人的缺陷发现率会明显下降。所以正确做法是**分多次**，而不是"今天下午我全看完"。

第四条，**复审前，代码作者应该先给代码加注释**。这一条常被跳过。作者先在关键处写清"这里为什么这么处理"，评审者才不用 reverse engineering 你的思路。

第五条，**使用检查表（checklist）**。课件说得很明确：检查表肯定能改进双方的结果。因为清单把"凭记忆"变成"逐项确认"——空指针判了吗？资源关了吗？日志里有敏感信息吗？

第六条，**验证缺陷是否真正被修复**。这一条最容易被忽略：评审意见提了，作者说"改了"，然后没人回头看。**没验证的修复等于没修复。**

六条合起来其实是一个意思：**评审是一件有节奏、有清单、有闭环的工程活动**，不是"有空帮我瞄一眼"。

> **板书：**六条压成一行口诀："**少（<200–400 行）· 慢（300–500 LOC/h）· 短（≤60–90 分钟）· 先注释 · 用清单 · 必回验**"。

> **提问：**你们小组做课程设计时，如果只允许你落实其中一条，你选哪条？为什么？

> **预设回答：**

- 答"用检查表"——很务实，成本最低、收益最稳。清单还能沉淀成团队资产。

- 答"少看一点"——也对，这条最容易执行：把大改动拆成几次提交、几次评审。

- 答"必回验"——如果只能选一条，我会选这条。因为**前五条决定"能不能发现问题"，第六条决定"问题有没有被解决"**。

- 收口：现实里最容易崩的是"速度"——大家习惯把评审当浏览，所以第 2、3 条最需要刻意练习。

> **易错点 / 考点：**三个数字必须记准，考试很可能直接考数字：**200–400 行／次、300–500 LOC/h、60–90 分钟／次**。常见错法是把"每小时 300–500 行"记成"每天"或"每分钟"，或者把"单次不超过 60–90 分钟"记成"每天不超过"。判断口诀：**量小、速慢、时短**——这三个字方向一致，都是"少一点、慢一点"。

> **【思政落点 2】**：这几条指标背后是一种很朴素的职业态度：**认真是有成本的，所以要给它留出成本**。"帮我瞄一眼"式评审之所以普遍，不是人懒，是没人给它排时间。请你们以后带项目时记住：给评审排一小时，比事后加班查一个线上 Bug 便宜得多。同时，检查表这件事本身就是一种可贵的习惯——**把靠记性的事变成靠清单的事**。飞行员起飞前要念检查单，不是因为不专业，恰恰因为专业的人也会漏。

> **过渡：**方法有了，工具也得跟上。评审在今天的团队里，多数不是坐在会议室念代码，而是在版本平台上点开一个合并请求。

---

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

> 这一页讲工具怎么承载评审。课件给的入口是 GitHub 的 pull request 协作文档，也就是我们常说的 **PR**。
>
> PR 是什么？说白了就是"我改完了，请你们看，看完同意了我再合进主干"。它把评审需要的四样东西放在同一个地方：**改动的差异、逐行的评论、讨论的来龙去脉、以及"同意／要求修改"的结论**。
>
> 这四样东西为什么重要？我讲三个实际好处。
>
> 第一，**评审对象是"变更"，不是"整个仓库"**。你不是把三万行代码重读一遍，你只读这一次改了什么、为什么改。这就是为什么前面那条"一次 200 到 400 行"能落地——PR 天然把改动切小了。
>
> 第二，**留痕可追溯**。半年后有人问"这行为什么这么写"，你翻到那条评论，理由还在。文档会过期，PR 记录不会。
>
> 第三，**评审可以和自动检查绑在一起**。CI 一跑，静态检查、单元测试的结果直接挂在 PR 上，人只看"机器查不出来的部分"。这就是本章第三部分要讲的持续集成。

所以我的建议是：**哪怕课程设计只有三个人，也走 PR**。不是为了流程好看，是为了让你体验"我的代码要被人看"这种压力，这个压力会改变你写代码的方式。

> **板书：**中间写"PR = 差异 + 行级评论 + 讨论 + 结论"；左侧写"评审对象 = 变更"；右侧写"CI 结果挂在 PR 上"；下方写"三个好处：切小、留痕、自动化"。

**提问：**如果团队不用 GitHub，用 GitLab、Gitee 或者公司自建平台，这套做法还成立吗？

**预设回答：**

- 答"不成立，得用 GitHub"——名字不重要。GitLab 叫 Merge Request，本质一模一样；Gitee、Coding 也都有。

- 答"成立，换个名字"——对。要抓的是**四要素**（差异、评论、讨论、结论），平台只是载体。

- 追问：如果团队连版本管理都没有，只有共享文件夹呢？——那先解决版本管理，评审这件事在这个阶段做不了，因为**你根本说不清"改了哪些行"**。

**易错点 / 考点：**常考"代码评审的工具支撑方式"，答案要包含**版本控制平台的合并请求、行级评论、与 CI 集成**。易错点：把"PR"和"commit"混为一谈——commit 是一次提交，PR 是**一次待评审的变更集合**，可能包含多个 commit。

**过渡：**光说流程太抽象，我们看一个真实的变更评审长什么样。

---

## 【5.11】示例：change review（2 分钟）

这一页是一次真实的变更评审示例。你不要把它当成"界面截图"看，要把它当成**一次评审的现场记录**，从中读出四件事。

第一件，**看边界**。这次改动动了哪些文件、每个文件改了几行、是新增还是修改。评审的第一分钟就该弄清"改动范围"，因为范围之外的东西你不需要看。

第二件，**看上下文**。好的评审界面不只显示改动的那一行，还显示它前后几行。原因很简单：**一行代码对不对，取决于它周围在做什么**。同一句赋值，放在事务里和放在事务外，风险完全不同。

第三件，**看评论的落点**。评论是挂在具体某一行上的，不是笼统地说"这段写得不好"。挂到行上，作者才知道该改哪；这也是我们训练表达的地方——**把模糊的不满，说成具体的建议**。

第四件，**看结论**。评审最后必须有一个明确的状态：同意合并、要求修改、还是先讨论。最怕的是"看完了，没说话"——那不是通过，那是漏审。

这张示例图要给你们的印象是：**评审是一个结构化的动作，有范围、有上下文、有落点、有结论**。缺一样，它就退化成聊天。

**板书：**四格：范围（改了什么）／上下文（前后在干什么）／落点（评论挂在哪一行）／结论（同意・改・再议）。

**提问：**一次评审看完，作者只收到一句"整体还行，建议优化一下"，你作为作者会怎么办？

**预设回答：**

- 答"那就先不改了"——这正是模糊评论的后果：它不构成可执行的要求，于是被忽略。

- 答"回去问清楚哪里要优化"——对，这是作者该做的。同时也说明评审者没尽到责任。

- 追问：怎么把这句话变有用？——"第 42 行这个查询在循环里，改成批量查一次"，**具体到行、具体到改法**，这才叫评审意见。

**易错点 / 考点：**这页考的是"评审意见的质量标准"：**可定位、可执行、有理由**。易错点是把"评审"理解成"打分"或"表态"——评审的产出是**可执行的修改项**，不是一个态度。

**过渡：**评审最该盯的是什么类型的缺陷？下一页给了一个特别典型、也特别常见的：空指针保护。

---

## 【5.12】缺陷示例：空指针保护案例分析（2 分钟）

这一页是一个缺陷案例，主题是**空指针保护**。它是评审里最值得练的一类缺陷，因为它有个特点：**不运行也能看出来，运行起来却常常不报错**——只是结果不对，或者干脆悄悄吞掉。

我们看这类缺陷的几种典型形态。

第一种，**该判的没判**。方法拿到一个对象就直接调它的方法，注释里声称"调用方保证非空"。可调用方有三个，其中两个不保证。这就是典型的"契约只写在注释里，没写在代码里"。

第二种，**判错了对象**。判了 A 不为空，用的却是 B；或者判了外层对象不为空，里面那个字段还是空的。这是评审时最容易看走眼的一种，因为"看起来有判空"。

第三种，**判了之后又拿来用**。比如判空之后走了一个分支，分支里又从别处取了个可能为空的值，直接使用。保护只保护了一半。

第四种，**用异常吞掉问题**。捕获了空指针异常，然后 `return null` 或者只打一行日志。这段代码从此"永不报错"，但它把一个确定性错误变成了一个随机错误——**错误被推迟到更远、更难查的地方**。

这四种形态说明一件事：**空指针不是一个"记得判一下"的问题，而是一个"要想清楚谁能保证非空"的问题**。所以评审到这里，问的不该是"这里判空了吗"，而是"**这个参数的为空可能性，由谁负责**"。

**板书：**四种形态：漏判／判错对象／判一半／异常吞掉；右侧写一行"评审要问的问题：这个值由谁保证非空？"；底写"注释里的契约 ≠ 代码里的契约"。

**提问：**一个方法在入口处对所有参数都做了判空，抛 `IllegalArgumentException`，这算不算做到了空指针保护？

**预设回答：**

- 答"算，很规范"——形式上规范，但要追问：**调用方是谁？** 如果调用方全在同一层，这个异常会一路抛到用户面前；如果是给外部用的接口，快速失败是对的。

- 答"不算，得看业务上能不能为空"——这更接近本质。有些参数"可以为空"是合法输入，直接抛异常反而是错的，应该走默认分支。

- 追问：所以判空的正确姿势是什么？——**先定义契约**（可空／不可空、为空时怎么处理），再让代码和契约一致。

**易错点 / 考点：**常考的判断题是"加了判空就没有空指针风险了"——错，判空只处理了你想到的路径。另一个易错点：`catch (NullPointerException e) {}` 这种写法在评审里属于**必须修改项**，因为它掩盖缺陷；考试若问"捕获空指针异常的坏处"，答"隐藏真实缺陷、把确定性错误变随机错误"。

**过渡：**除了互查和工具，还有一些做法能让评审真正变成团队习惯。下一页列了五条。

---

## 【5.13】开发人员走上台介绍其代码（2 分钟）

这一页讲一种很朴素、但效果出奇好的做法：**让开发人员走上台，介绍自己写的代码**。课件给了三条规则：每个人都有机会、随机抽取、现场展示。

我先把这三条拆开说。

**每个人都有机会**——意思是这不是"高手表演"，是全员参与。新人也要讲，讲得不好也要讲，因为讲的过程就是逼他把自己的设计想清楚。

**随机抽取**——这一条是关键。如果提前指定，被点到的人会准备一份"演示版代码"；随机抽，你写的每一行都可能被摊开，这会实实在在改变平时的写法。

**现场展示**——不是念 PPT，是打开代码，从入口一路走下来：这段为什么这么拆、这个分支为什么存在、这里为什么加了缓存。

为什么有效？因为**写代码时你可以糊弄自己，讲代码时糊弄不过去**。讲到某一行你卡住了，说不出它为什么在这儿，那这行八成就该重构。这叫"**用讲解来验证理解**"。

它还顺带解决一个问题：团队里只有一个人懂某块代码，是最大的隐性风险。上台讲一次，第二个人、第三个人就有了基本概念，将来出问题不至于只有一个人能救。

你们课程设计只有三四周，我建议至少做一次：**每人讲五分钟自己的核心类**，不讲界面，讲逻辑。

> **板书：**写"上台讲代码"；下三条：人人有机会／随机抽取／现场展示；旁边写"讲到卡住的那一行 = 该重构的那一行"；再加一句"知识分散 = 风险分散"。

**提问：**如果你被随机抽到，讲自己上周写的一段代码，你觉得最可能在哪一句上卡住？

**预设回答：**

- 答"讲不清为什么用这个设计"——非常真实。很多设计是"当时顺手就这么写了"，讲不出理由，就说明这里没有真正的决策。

- 答"讲不清某个分支什么时候会进"——那这段代码要么是死代码，要么是没人测过的路径，两种情况都该处理。

- 答"能讲清，但怕被人挑毛病"——有压力是正常的。我提醒一句：**被挑毛病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。

**易错点 / 考点：**这一页常被当成"软技能、不考"。其实考点很明确：**代码评审的形式包括正式会议评审、同行互查、走查（walkthrough）、上台讲解等**，判断题若说"代码评审只能是书面检查"就是错的。另外记住"随机抽取"的作用是**保持压力、避免表演**。

**过渡：**互查、上台讲，是"怎么评"。还有几条是关于"评审怎么变成组织习惯"的，我们一起收一下。

## 【5.14】其它优秀实践（2.5 分钟）

这一页是第一部分关于评审的最后一块拼图，五条实践，条条都指向同一件事：**让评审从"个人美德"变成"组织习惯"**。

第一条，**关键代码的集体评审**。不是所有代码都值得全员坐下来看，要挑：涉及资金的、涉及权限的、涉及核心算法的、并发和事务相关的。这类代码出错代价最大，评审的投入产出比最高。**评审要有优先级，不能平均用力。**

第二条，**代码评审作为一种文化**。文化这个词听着虚，其实有很具体的判断标准：新人提交代码，会不会有人主动看？指出问题时，语气是"这里可能有个坑"还是"你怎么连这个都不懂"？**前者是文化，后者是内耗。**

第三条，**借助工具更有效**。这一条前面已经讲过了：PR、静态检查、CI 报告。工具的价值是**把人从能自动查的事情里解放出来**，让评审的注意力集中在设计和逻辑上。

第四条，**流程定义**。要写清楚：谁必须评审、多长时间内响应、什么级别的改动需要几个人同意、评审不通过怎么处理。没定义的流程，全靠"看谁好意思催"。

第五条，**培训**。让评审者知道该找什么。这一条和前面的检查表是一对：**检查表告诉你看什么，培训告诉你为什么看。**

五条连起来看，它是从"挑重点（集体评审）→ 立氛围（文化）→ 上工具 → 定规矩（流程）→ 提能力（培训）"的一条链。缺哪一环，评审都会退化成形式。

> **板书：**五行链条：挑重点（关键代码集体评审）→ 立氛围（文化）→ 上工具 → 定规矩（流程定义）→ 提能力（培训）；下方标一句"评审 = 制度，不是美德"。

> **提问：**这五条里，如果你们团队只能补一条，你会补哪条？说说理由。

> **预设回答：**

- 答"流程定义"——务实，先把"谁看、多久回、几个人同意"写下来，评审才有约束力。
- 答"文化"——最根本，但最难，因为它改的是人的态度。
- 答"关键代码集体评审"——投入产出比最高，尤其适合人少的团队。

- 收口：我的建议是先立"关键代码必集体评"这条最小规则，跑顺了再补流程和培训。**先有,再好。**

> **易错点 / 考点：**这一页常考"代码评审的组织形式与实践"，要能说出**集体评审、评审文化、工具支撑、流程定义、培训**五项。易错点是把"集体评审"理解成"所有代码都要集体评审"——恰恰相反，它的前提是**挑关键代码**。

> **过渡：**评审是人看。人看得慢、会累、也会漏。所以接下来我们要给评审配一套自动化工具——代码静态分析工具。

---

## 【5.15】代码静态分析工具（1 分钟）

> 我们换一个角色——从"人看代码"换成"工具看代码"，这一块叫**代码静态分析工具**。
>
> 先把这个词拆开：**静态**，指不运行程序；**分析**，指按规则去理解代码结构；**工具**，指它能自动跑。三样合起来，它干的事就是：**把源码当成数据，用规则去查里面有没有可疑模式。**
>
> 它和评审什么关系？一句话：**评审负责"为什么这么写"，工具负责"这么写有没有踩规则"。** 人擅长判断意图，机器擅长不知疲倦地重复检查。这两件事不要互相替代。
>
> 这一块内容看着全是工具名，但考点其实不在"你会不会装插件"，而在**哪一类工具查哪一类问题**。我们一会儿会把工具谱系摊开，你重点记"归属"，别记按钮。
>
> 还有一句提醒：这些工具**会误报，也会漏报**。所以你看到它报了一堆，不要慌；看到它一个都没报，更不要放心。

> **板书：**写"静态分析 = 不运行 + 按规则 + 自动化"；左右各一行："评审管意图／工具管规则"；下写"会误报、会漏报 → 结果是线索，不是判决"。

> **提问：**一个工具跑完，报告里零缺陷，能不能据此说这段代码是安全的？

> **预设回答：**

- 答"能，零缺陷就是没问题"——这正是最危险的想法。零缺陷只说明**没有命中它的规则**。

- 答"不能，规则有限"——对。规则的覆盖面永远是代码可能性的一小部分。

- 追问：那工具报告还有什么用？——它是**线索清单**，帮你把注意力放到高风险的地方，比人眼全量扫描高效得多。

**易错点 / 考点：**静态分析属于**静态测试手段**，"静态"指的是**不执行被测程序**，不是"代码不动"。判断题"静态分析可以发现运行时性能问题"——错，运行时问题必须动态测试。

**过渡：**那么这一块到底有哪些工具？我们按语言和用途把谱系排一遍。

---

## 【5.16】代码静态分析工具（2.5 分钟）

这一页把静态分析工具的谱系排开了。我们不逐个讲用法，只讲"谁管哪一块"，这是考点，也是你以后选工具的依据。

先看通用型这一类：**SonarLint**。它不绑定某一种语言，Java、C、Python、JS 都能查，还能装进各种 IDE。它是这一节的"首推"，下一页单独讲。

然后是**按语言分的**。Java 这边三个老牌选手：**FindBugs** 查字节码层面的缺陷模式、**PMD** 查源码里的坏味道和可疑写法、**Checkstyle** 只管风格规范。请注意这三个的分工：**一个管"可能出错"，一个管"写得不好"，一个管"不符合规范"**——很多同学把它们混成一个东西，其实它们查的不是一回事。

C/C++ 这边，课件列的是 **Clint、Oclint、Cpplint** 和 **TestC++** 这一类，多是从 Google 内部或开源社区长出来的风格与缺陷检查器。

Python 这边是 **Pylint** 和 **PySonar2**——Pylint 管规范和可疑写法，PySonar2 更偏类型推断和语义分析。

最后是**安全类**的大块头：**Coverity、Fortify**。它们面向的是"能被攻击的缺陷"，做的是深度数据流分析，代价是慢、贵，一般放在 nightly 或发布前跑。

课件还给了一个总入口：OWASP 的源码分析工具清单。它的价值是**帮你横向对照**——同一类问题，有哪几个工具能查、各自支持什么语言。

这一页请记住一条选型逻辑：**开发机上装轻的（SonarLint、Checkstyle），流水线上装重的（Coverity、Fortify）。**

**板书：**三层：通用（SonarLint）／分语言（Java：FindBugs・PMD・Checkstyle；C/C++：Clint・Oclint・Cpplint・TestC++；Python：Pylint・PySonar2）／安全（Coverity・Fortify）；旁注"轻的上开发机，重的上流水线"。

**提问：**同样是 Java，FindBugs 和 Checkstyle 能不能互相替代？为什么？

**预设回答：**

- 答"能，都是查 Java 的"——不能。它们查的对象不同：Checkstyle 只看**源码的书写形式**，FindBugs 看**编译后的字节码里的可疑模式**。

- 答"不能，一个管风格一个管缺陷"——对。风格问题不影响运行，缺陷模式可能导致运行出错，严重程度不同，处理优先级也不同。

- 追问：那 PMD 放哪？——PMD 更偏源码级的"坏味道"：没用的变量、空的 catch、复杂的条件。三者是**互补**关系。

**易错点 / 考点：**① 三个 Java 工具的定位必须分清：**Checkstyle＝风格规范，PMD＝源码坏味道，FindBugs＝字节码缺陷模式**；② 安全类工具（Coverity／Fortify）属于**深度静态分析**，常考"它和轻量检查的区别"；③ 易错：以为 SonarLint 只支持 Java——它几乎是全语言。

**过渡：**谱系里排第一的那个，我们单独看一眼它凭什么被首推。

---

## 【5.17】首推：SonarLint（2.5 分钟）

这一页讲为什么在这么多工具里，课件首推 **SonarLint**。它给了三件事：**发现缺陷、修正缺陷、学习**。我按这个顺序讲，但重点在第三件。

**发现缺陷**——这是基本功能。你在 IDE 里写代码，它实时在可疑的地方标出来。注意"实时"这两个字很关键：它把检查从"提交后"提前到"书写中"，你刚写完就知道，改起来几乎零成本。

**修正缺陷**——它不只告诉你"这里有错"，还会给出**具体怎么改**，很多时候能一键修。这一点对新手特别友好：它教的不是"你错了"，而是"应该这样写"。

**学习**——这是我认为最值钱的一条。每一条规则背后都有解释：为什么这么写有风险、什么场景下例外。你用三个月 SonarLint，等于跟着一本会挑你毛病的教材在学。**工具的最高价值不是替你干活，是让你少犯同样的错。**

课件还强调它**几乎支持所有流行的语言和 IDE**。这一条的现实意义是：你不用为了换语言换工具，团队里用 Eclipse、用 IntelliJ、用 VS Code 的人，规则口径能保持一致。

最后提醒一句：SonarLint 装上之后，第一周你会觉得满屏都是问题，很想把它关掉。**别关。** 先把严重级别

调到你关心的那一档，逐个消，你会发现自己写代码的习惯在两三周内明显变化。

> **板书：**三个圈写"发现 → 修正 → 学习"，在"学习"上打星；下写"实时 = 把检查提前到书写中"；再写"几乎全语言、全 IDE → 团队口径一致"。

**提问：**一个工具既能发现问题又能教你怎么改，那它是不是可以替代老师、替代代码评审？

**预设回答：**

- 答"差不多可以"——差得远。工具只认规则，它不会问"这个需求该不该这么实现"。

- 答"不能，规则有边界"——对。它能查"这里可能空指针"，查不了"这个业务逻辑本身错了"。

- 追问：那它能替代哪部分工作？——**能替代"机械的、有明确规则的检查"**，正好把评审者的时间省下来看设计。

**易错点 / 考点：**SonarLint 的三项能力（发现／修正／学习）常考。易错点有两个：① 把它和 **SonarQube** 混为一谈——SonarLint 是 **IDE 里的本地检查器**，SonarQube 是**服务端的质量平台**，下一节会讲两者怎么配合；② 误以为它只在保存或提交时检查——它是**边写边查**的。

**过渡：**说它"能装进几乎所有 IDE"，我们就在最常见的 IntelliJ 里看一眼它长什么样。

---

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

这一页是在 IntelliJ 里装 SonarLint 的示例。安装动作本身很简单——在插件市场搜 SonarLint、装上、重启，课件给的插件页地址是 JetBrains 插件市场的 7973 号条目。我要讲的不是点哪里，而是**装完以后你应该怎么用它**。

第一，**看提示的层级**。它标出来的问题分严重级别，从一个文件的窗口里能按级别筛。你可以先只看最严重的那一档，把"会出事的"清掉，再处理"不好看的"。

第二，**看规则的说明**。每条提示点开都有解释：这条规则在防什么、什么情况下可以忽略。**忽略也是一门学问**——你要能说清"我为什么认为这里不适用"，而不是"我觉得它烦"。

第三，**注意它查不到的地方**。IDE 插件做的是单文件、轻量、快速的分析，它不做跨模块的深度数据流追踪。所以它报"没问题"，只代表这个文件在本文件范围内看起来没问题。

第四，**别把提示当成必须全清的清单**。全清意味着你要为一个老项目改几千处风格，那不现实。正确做法是：

**新代码零新增问题，老代码逐步收敛。** 这条原则在团队落地时特别重要。

> **板书：** 写"装插件 → 分级看 → 读规则说明 → 知道它查不到什么"；右下写团队落地原则："新代码零新增，老代码逐步收敛"。

**提问：** 一个已有十万行的老项目，SonarLint 一扫报了两千个问题，你要不要一次性全改？

**预设回答：**

- 答"要，一次清干净"——风险很大：两千处改动，回归测试根本覆盖不住，很可能改出新的 Bug。

- 答"不改，反正能跑"——那新代码会继续往这个坑里掉。

- 答"分两步：先卡住新增，再慢慢清老的"——这是正确做法，也是业界通行的"棘轮"策略。

- 追问：怎么"卡住新增"？——**只对本次变更的代码做检查**，这就是后面要讲的 CI 门禁。

**易错点 / 考点：** 这一页考的是"工具落地的策略"，而不是安装步骤。标准答法：**分级处理、差异检查、新代码零新增**。易错点是把"清空所有告警"当成目标——那会导致大量无效改动与技术债的假性清偿。

**过渡：** 本地查完，团队怎么保证"每个人的标准一样"？靠服务端。SonarLint 可以和 SonarQube 连起来用。

---

## 【5.19】可以和 SonarQube 集成使用（2 分钟）

这一页讲 SonarLint 和 **SonarQube** 的配合。这两个名字太像了，我先把它们分清楚：**SonarLint 在开发者的 IDE 里，是个本地检查器；SonarQube 在服务器上，是个团队级的质量管理平台。**

它们为什么要连起来？因为不连会出两个问题。

第一个问题，**规则不一致**。你本地用默认规则集，觉得挺干净；服务端用团队配置的规则集，一扫一大片。开发者和流水线互相不认账，最后谁也不信谁。连接之后，本地会**同步服务端的质量配置**，本地和服务端用同一套标准。

第二个问题，**没有全局视角**。本地只能看你打开的这几个文件；服务端能看整个项目、整个版本、甚至多个版本的趋势——**技术债是在涨还是在降，重复代码有多少，覆盖率是多少**，这些都得靠平台。

所以这套组合的分工是：**SonarLint 负责"我正在写的这段代码"，SonarQube 负责"我们整个项目现在什么水平"。** 一个管即时反馈，一个管长期趋势。

落到团队实践上，通常是这样：本地装了 SonarLint 并且连上服务端，提交前该清的就清了；CI 上再跑一遍 SonarQube，用**质量门禁**卡住合并。两道关，一道防"写进去"，一道防"合进去"。

**板书：**左写"SonarLint：IDE 内、单文件、即时"；右写"SonarQube：服务端、全项目、趋势＋门禁"；中间写"连接 → 规则同步"；下写"两道关：写了就查、合了就卡"。

**提问：**如果团队只允许保留一个，你留 SonarLint 还是 SonarQube?

**预设回答：**

- 答"留 SonarQube"——它覆盖面更广、能做门禁；但少了本地即时反馈，开发者要等到提交才知道问题。

- 答"留 SonarLint"——反馈快，但没有统一标准和趋势，团队层面管不住。

- 答"看团队规模：小团队留 SonarLint 够用，多人协作要 SonarQube"——这个判断很实用。**工具的选择跟协作规模走。**

- 追问：那如果两个都留，冲突怎么处理？——让服务端当唯一权威，本地跟随服务端配置。

**易错点 / 考点：**SonarLint 与 SonarQube 的区别是高频考点：**位置（IDE／服务端）、范围（单文件／全项目）、时效（即时／周期性）、职责（个人反馈／团队门禁）**。另注意课件里 `SolarLint`／`SolarQube` 是笔误，正确拼写是 **Sonar**Lint／**Sonar**Qube，答题时写对拼写。

**过渡：**接下来看一个"曾经很红、现在基本退场"的工具，它的兴衰本身就是一课。

---

## 【5.20】来自大学的 FindBugs（2 分钟）

这一页讲 **FindBugs**。它的出身很有意思：**不是商业公司做的，是高校研究团队做出来的**，项目主页在 findbugs.sourceforge.net。

它的技术路线和前面那些不一样：**它不看源码，看编译后的字节码**。为什么这么做有道理？因为源码里有很多写法是"看起来像缺陷"的，而字节码反映的是编译器真正生成的东西——它直接对应"程序运行时会怎么做"。所以 FindBugs 抓的是**缺陷模式**：空引用可能发生、锁的释放路径不完整、equals 和 hashCode 不配对、一个值被算了但没用上。这些东西有个共同特点：**它们不违法语法，编译器不报错，但运行时可能出事。**

它的价值在哪？在于**它把"老程序员的经验"变成了可自动执行的规则**。一条规则背后，往往是一个真实事故。

但这一页真正要讲的是后面那半句：**2015 年 3 月，FindBugs 3.0.1 发布之后，就没有更新了。** 一个曾经被

> 广泛使用的工具，停更了。
>
> 这件事给我们两点提醒。第一，**工具是有寿命的**。它依赖的 JDK 在升级、字节码格式在变，停更意味着慢慢失效。第二，**但缺陷模式不会过期**。空引用、锁未释放、equals 不配对，这些坑十年后还在。所以学工具要学它**在查什么**，别只学它怎么点。

> **板书：**写"FindBugs：高校出身 → 查字节码 → 缺陷模式"；右侧写时间线"2015.03 发布 3.0.1 → 之后停更"；下方写一句"工具会停更，缺陷模式不会过期"。

> **提问：**一个工具已经停止维护了，你还敢在项目里用它吗？

> **预设回答：**

- 答"不敢，风险太大"——合理。停更的工具有两个现实问题：新语法可能解析不了，新 JDK 可能跑不起来。

- 答"能用，规则还是那些"——部分正确，但要限定条件：**只作为辅助参考，不作为门禁依据**。

- 追问：如果它报了一个你觉得很有价值的缺陷，怎么办？——把这条规则的理解搬到仍在维护的工具里去实现，**人要带走知识，不要抱着工具**。

> **易错点 / 考点：**FindBugs 的两个特征常考：**基于字节码分析、2015 年 3 月 3.0.1 之后停止更新**。易错点：把它归为"代码风格检查工具"——它不是，它查缺陷模式；风格检查是 Checkstyle 的活。

> **过渡：**它停更之后，位置被谁占了？答案有点出人意料——被 IDE 自己。

---

## 【5.21】FindBugs @ IntelliJ（2 分钟）

> 这一页接着讲 FindBugs 在 IntelliJ 里的命运。课件给了一句很直白的话：**IntelliJ IDEA 2021.2 发布之后，就基本把 FindBugs 干掉了。**
>
> 为什么会被"干掉"？因为 IDE 厂商把这类检查**内建**了。你想想开发者的选择逻辑：装一个插件要配版本、要跟着 JDK 升级、还会和 IDE 的检查重复报错；而 IDE 自带的检查开箱即用、和编辑器深度集成、跟着 IDE 一起升级。**同等效果下，内建永远赢过插件。**
>
> 这件事在教学上很有价值，我给你三层意思。
>
> 第一层，**工具会往平台里收**。单点工具活得好不好，很大程度取决于平台厂商是否愿意自己做。这条规律在测试领域反复出现：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。

第二层，**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。

第三层，也是最实用的：**你学的应该是"缺陷长什么样"，而不是"某个插件在第几个菜单"**。菜单会变，缺陷模式十年不变。这也是为什么这一章我反复强调"工具会换，模式不会换"。

**板书：**写"FindBugs（插件）→ IntelliJ IDEA 2021.2 之后基本被内建 inspection 取代"；下三层：工具往平台收／能力换入口／学模式不学菜单。

**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。更重要的是**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：所以工具的真正战场在哪？——**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：**这一页的考点是"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 = 被淘汰 = 没用"，其实被替代的是**形态**，不是**能力**。

**过渡：**IDE 自带的检查到底能查出什么？下一页用一张图说清楚它的能力边界。

---

## 【5.22】IntelliJ 分析功能代替了 FindBugs（2 分钟）

这一页是用 IDE 自带的分析功能来替代 FindBugs 的示意。我讲这一页的目的不是让你记住某个菜单，而是让你建立"**IDE 检查的四个维度**"这个框架。

第一个维度，**语法与语义层**：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。这类问题 IDE 靠编译器前端就能发现，几乎零误报。

第二个维度，**空值与可空性层**：可能为 null 的引用被直接使用、`@Nullable` 标注与实现不一致。这是日常最高频的一类。

第三个维度，**资源与并发层**：流没关、锁在异常路径上没释放、在同步块里调外部服务。这类问题人眼很难

稳定发现，工具却擅长。

第四个维度，**风格与规范层**：命名、格式、注释缺失。注意这一层**严格来说不算缺陷**，但它影响可读性，所以也放进来了。

四个维度里，前三个是"会出事"，第四个是"不好看"。你在 IDE 里看到的提示，绝大多数都能归到这四个里。**归好类，你就知道哪些必须当场改、哪些可以排队改。**

另外提醒一句：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。所以它的"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 的限制是一样的。

> **板书：**四行写四层：语法语义／空值与可空性／资源与并发／风格规范；右侧标"前三条＝会出事，第四条＝不好看"；底写"本地・增量・快 → 不做全程序分析"。

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

**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"的情况。

- 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 的检查是"提高下限"，但如果要把标准变成**团队的、可执行的门禁**，得靠专门的工具。下一页是其中最典型的一个：Checkstyle。

---

## 【5.23】代码风格检查 CheckStyle（3 分钟）

这一页讲 **Checkstyle**，主页是 checkstyle.org。它的定位特别清楚：**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。它最大的价值是**把规范从文档变成可执行的门禁**。课件给的集成方式很具体：在 Maven 里跑 `mvn checkstyle:checkstyle`；在 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`。

你把这串东西连起来看，它其实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 → 输出机器可读的报告 → 报告进 CI 门禁**。这条链路不是 Checkstyle 独有的，几乎所有静态检查工具都是这个结构，你以后学别的工具只要换个文件名。

为什么强调"进 CI"？因为只在本地跑，结果就是"我今天没空跑"。进了流水线，它才是**所有人必须通过同一道关**。这也是本章第三部分要反复讲的一句话：**规范一旦可执行、可强制，它就真了。**

我再补一个实操建议：**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。一上来开一百条规则，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**。

> **板书：**横链写"checkstyle.xml（规则）→ mvn checkstyle:checkstyle ／ apply from: 'checkstyle.gradle'（调用）→ ./gradlew check（执行）→ build/reports/checkstyle/main.html（报告）→ CI 门禁"；右侧写"只管风格，不管对错"。

**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。**风格检查是地基，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）

这一页是 Checkstyle 的规则示意。规则列表很长，但你不用背，你需要建立的是**规则的分类感**——知道它能在哪些维度上约束你。

我把它归成五类。

第一类，**命名**：类名、方法名、变量名、常量名的书写形式。这一类的争议最小，收益也最直接——名字统一了，读代码的人不用每次重新适应。

第二类，**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"，但也最适合机器管，因为它完全客观。

第三类，**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。这一类的价值在于**逼你写下"这个方法干什么、参数什么含义"**。

第四类，**结构与语句**：`if` 必须带花括号、`switch` 必须有 `default`、不允许 `import *`、不允许空语句块。这一类的部分规则已经接近"潜在缺陷"了——比如漏了 `default`，可能就是你没想到的情况。

第五类，**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**——你还记得圈复杂度 e−n+2p 吧？Checkstyle 可以直接把超过阈值的类和方法列出来。

五类看下来你会发现：**Checkstyle 的规则其实就是"团队约定"的可执行版本**。你写一条规范，它就能查一条。

**板书：**五类竖排：命名／格式排版／注释文档／结构语句／度量（长度・参数数・圈复杂度・嵌套深度）；旁边一行"规范的可执行版本"。

**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：**这一页的考点在"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）

这一页是 Checkstyle 的检查结果。报告怎么读？我给你一个四步顺序。

第一步，**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**"。因为修一条高频规则，往往能一次消掉几百个问题——这是投入产出比最高的动作。

第二步，**看严重级别**。哪些是 error，哪些是 warning。团队应该约定：**error 必须修，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。这里有个使用技巧：**优先修"新改动涉及的"**。老代码里的历史违规先记账，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。

我还想强调一句：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。**

**板书：**四步：看分布（哪条规则最多）→ 分级别（error 必修／warning 排队）→ 定位（文件・行号・规则名）→ 反推（改规范或补培训）；右侧写"看趋势，不看清零"。

**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：**这一页常考"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。要真正看清复杂度，得用专门的度量工具。

---

## 【5.26】SourceMonitor 检测代码复杂度（2 分钟）

这一页讲 **SourceMonitor**，一个专门做**代码度量**的工具。它和前面几个的区别是：它主要不是"报错"，而是**算数、画图、看趋势**。

它算哪些数？我们看几类最常用的。

**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，那这段代码基本没人敢改。

**注释比例**和**方法、类的数量**——用来看结构和文档状况。

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。

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"。**

> **板书：**指标列表：代码行／语句数／复杂度／最大块深度／注释比例／方法与类数量；下方画一条趋势线，标"突然跳高 = 重构热点"；右侧写"定位热点，不给分"。

**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。那部分的风险，比你想的大得多。

---

## 【5.27】软件第三方库风险巨大（3 分钟）

> 这一页是这一部分里最重要的一页，讲**第三方库的风险**。我先念课件给的三个数字，你感受一下量级。
>
> 第一个，**98% 的正在使用开源第三方库的公司，并不了解它们**。注意这句话的措辞——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。
>
> 第二个，**每年有 4000 个新增的开源第三方库漏洞被报告**。这个数字说明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。
>
> 第三个，课件用一条时间轴给出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%。十年九倍。这意味着什么？意味着**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。**
>
> 那问题就来了：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**。它不是"某个库有 Bug"这么简单，它是**风险的传导链**：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。
>
> 所以这一页要建立的意识是：**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。

> **板书：**三个数字：98%（不了解在用）／4000（每年新增漏洞）／10%→30%→90%（2010→2015→2020

使用比例）；下方写一句"你的系统边界 = 依赖边界"，再写"供应链风险 = 传导链风险"。

**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：**既然风险来自两个方向——自己写的代码和引用的库，那安全分析自然也要分两条线。下一页把它们摆清楚。

---

## 【5.28】代码级安全分析对象（2.5 分钟）

这一页把代码级的安全分析分成两条线，一张图讲清，我把它翻成一句话：**一条线查"我自己写错的"，一条线查"我引错版本的"。**

第一条线是 **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，是指"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。

第二条线是 **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**。

这两条线的差别，本质是"**未知 vs 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数字，20%、35%、45%，用来说明这两类问题在整体安全风险里的分布——具体口径以课件为准，你记住"两类都占相当比重、都不能偏废"这个结论就够了。

落到实践上：**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。

**板书：**两列对照表：SAST｜SCA；自己写的代码｜引用的组件；未知漏洞｜已知漏洞；CWE｜CVE；提交与 CI｜依赖变更与定期扫描。旁边写"未知 vs 已知"。

**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？下一页看它的工具形态。

---

## 【5.29】SCA 工具（2 分钟）

这一页是 SCA 工具的示意。工具的具体界面不重要，重要的是它的**工作流程**，一共四步，你把这四步记住，用哪家产品都一样。

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：这个组件是不是真的被打进运行包？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？这一步必须人工参与，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——因为漏洞库天天在更新，今天没问题不代表明天没问题，所以**定期重扫是必须的**。

我特别强调一下第一步的价值：**很多团队连"我用了什么"都答不出来。** 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，能立刻答出来的团队不多。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**。

**板书：**四步链：依赖清单（SBOM）→ 比对漏洞库（CVE）→ 评估影响（可达性）→ 修复与跟踪（定期重扫）；旁边写"第一步的价值 = 可见性"。

**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 + 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，那这些工具在工程里到底怎么用、什么时候用？下一页给了三种典型场景。

---

## 【5.30】代码分析工具常见的使用场景（2.5 分钟）

这一页把代码分析工具的使用场景归成三类，还给了配套的检测项和耗时。这是一页"决策页"——它告诉你**什么场合该用什么强度的检查**。

场景一，**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。它的目的是**为开发者快速返回检查结果**。

场景二，**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。它的目的是**在代码进入主干之前拦住问题**，这是性价比最高的一道关。

场景三，**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**。它的目的是**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**。这个"三档"结构你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**。注意这里既有代码问题（注入、资源泄露），也有配置问题（硬编码凭据、配置审查）——说明现代代码分析不只看 `.java`，还看配置文件。

最后是那组关键词：**规则集、检测项、检测耗时**。三者是互相制约的：**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。

> **板书：**三档表：IDE 实时（几秒・有限检测项・快速反馈）｜每次提交（中等・拦截入库）｜CI/CD 集成（几分钟・全量・守发布口）；下列检测项：SQL 注入／XSS／资源泄露／硬编码凭据／配置审查；底写"规则↑ → 耗时↑ → 位置↓（越靠后）"。

**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：**最后我们掀开盖子看一眼：这些 SAST 工具内部到底是怎么"看懂"你的代码的？下一页是它的通用框架。

## 【5.31】SAST 工具通用框架（3 分钟）

这一页是 SAST 工具的**通用框架**，信息密度很高，但结构其实很清楚。我按数据流向讲，你跟着箭头走一遍。

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**。课件列了一长串，我挑几个解释，你理解它们的共同目的就行：**函数调用**关系、**SSA**（静态单赋值，把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**（这个指针/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。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，才算真的可能发生**。

第五步，**检测结果**。到这里它才给你报一条。

这个流程解释了一个你一定会遇到的现象：**误报**。因为中间每一步都在做近似——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。所以 SAST 的结果永远是"**疑似**"，需要人去确认。这也是为什么它不能替代评审和测试。

> **板书：**竖链：源程序 → 解析（AST）→ 基本分析（函数调用・SSA・指向・前后支配・Mod-effect・多态・全局变量・抽象解释・符号执行・图可达）→ 缺陷模式（OWASP／CERT／CWE 的逻辑表达式）→ 求解（SMT）→ 检测结果；右侧写"每步都在近似 → 必然有误报"。

**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：**这一页的考点是**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/路径爆炸被截断**。易错点：把 SMT 求解当成"必须的一步"——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——我们知道了怎么在不运行代码的情况下抓缺陷。可有些东西，不跑起来是真的看不出来。所以下面进入第二部分：单元测试。

---

## 【5.32】5.2 单元测试（1 分钟）

我们进入第二部分，**单元测试**。这是本章重心，也是你实验课要动手的部分。

前面那部分我们一直在"看代码"，这一段开始**跑代码**。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？一句话：**它是唯一能在"缺陷还没跑远"的时候把它按住的测试层次。** 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。

这一部分我按六个问题讲：为什么要做？要做到什么程度？上下游没有怎么办（驱动和桩）？用什么工具写？依赖对象怎么处理（Mock）？怎么知道测得够不够（覆盖率）？

这六个问题，其实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。**

> **板书：**写"5.2 单元测试：跑代码"；下方六个问号：为什么做／做到什么程度／上下游怎么办／用什么写／依赖怎么处理／够不够；右侧写"对象最小、反馈最快、定位最准"。

> **提问：**单元测试和生产代码，谁应该先写？

> **预设回答：**

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

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

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

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

> **过渡：**这一部分具体包括哪几块？我们看它的内容清单。

---

## 【5.33】单元测试内容（2.5 分钟）

这一页是第二部分的**内容清单**，六项。它其实也是你这门课的六个知识点，我一项一项交代，顺便告诉你哪项会考、哪项要动手。

第一项，**为何要进行单元测试**——讲价值与成本。这一项看着像"务虚"，其实是最容易被问到"到底值不值"的地方。下一页专门讲。

第二项，**单元测试的目标和要求**——讲"测到什么程度算完"。这一项要背：功能、输入输出的容错、内部数据完整性、覆盖率指标。

第三项，**驱动程序和桩程序**——讲"被测的那个方法，上下游还没有怎么办"。这一项是本章的技术门槛，**驱动**是"我来调你"，**桩**是"你调别人，别人先用假的顶上"。理解这两个词，后面 Mock 就一通百通。

第四项，**单元测试工具 JUnit 及其实践**——这是你要动手的部分。JUnit 5 的架构、注解、断言、执行器，一条条都要会用。

第五项，**Mock 技术与框架（如 Mockito）**——讲"依赖的对象怎么造"。这是本章的难点，也是面试高频题。

第六项，**测试覆盖率工具 JaCoCo**——讲"我怎么知道测得够不够"。这一项的关键不是会点按钮，是**看得懂报告、知道行覆盖和分支覆盖的差别**。

六项合起来是一条完整的作业链：**想清楚为什么测 → 定标准 → 解决上下游 → 写出来 → 处理依赖 → 度量效果**。你实验交的那份测试代码，就是这条链的产物。

> **板书：**六项竖排：为何测／目标与要求／驱动与桩／JUnit 实践／Mock 与 Mockito／覆盖率 JaCoCo；右侧标"要背：②④"、"要动手：④⑤⑥"、"难点：③⑤"。

> **提问：**六项里，你觉得哪一项最容易被"跳过"？

**预设回答：**

- 答"覆盖率"——不少团队跑完不看报告，或者只看一个总数。

- 答"驱动和桩"——很多同学一上手就卡在这：被测方法要连数据库，怎么办？

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

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

**【思政落点 4】**：这六项排下来，你会发现单元测试是件"不显眼"的活：没有界面、没有演示效果、写完了也没人鼓掌。可它恰恰是**把地基做扎实**的那部分工作。我们这门课讲"课程设计"，很多同学愿意花两周做界面、花两天写测试。我想说的是：**决定一个系统能活多久的，往往不是它第一天多好看，而是它的每个零件经不经得起推敲。**愿意在看不见的地方下功夫，这就是工匠精神最朴素的样子——不是把活干得漂亮，是把活干得住。

**过渡：**第一项我们先讲透——为什么非得在单元这一层花这个力气？

---

## 【5.34】为何要进行单元测试？（3 分钟）

这一页回答"值不值"。课件给了四组论据，我逐条讲，最后给你一个能说服人的算法。

第一条，**编程与编译运行结束后，每 1000 行代码中大约残留有 2 到 6 个 Bug**。注意语境：它不是"新手代码"，是行业常态。一个五万行的系统，天生就带着一两百个缺陷等着你。**缺陷不是意外，是常态。**

第二条，**寻找与修改程序错误的代价占总体开发投资的 30% 到 60%**。这条更扎心：开发投入里有三到六成其实花在"找错和改错"上。**测试不是额外成本，它就是开发成本的主要组成**——区别只在你在便宜的环节花，还是在贵的环节花。

第三条，**尽早发现错误，成本越低**。这条是前面两条的推论，也是最重要的行动指南：同一个缺陷，在单元阶段发现，改一行代码；在系统测试阶段发现，要重新部署、重造数据、重新回归；上线后发现，要发版、要道歉、可能还要赔钱。

第四条，**发现问题比较容易、修正问题更容易、单元质量是系统质量的基石**。为什么容易？单元小、边界清

楚、能直接构造输入。为什么是"基石"？课件还补了一句关键的话：**单元测试可以比系统测试测得更彻底**——系统测试受限于场景，方法内部的分支组合你构造不出来，单元测试可以逐个分支喂数据。

最后看这两个算式，这是本页精华。**0.83 的 30 次方，约等于 0.00123**；**0.99 的 30 次方，约等于 0.74**。什么意思？一个系统由 30 个单元串起来，每个单元"通过率" 83%，整体一次跑通的概率只有千分之一左右；每个单元做到 99%，整体也才 74%。

这两个数字说明：**单元层面的缺陷率，会被集成按指数放大。** 指望系统测试阶段"整体调通"，等于指望 30 个不靠谱零件拼出一台靠谱机器。**所以质量必须在单元这一层做上去——它是唯一能让后面几层站得住的地方。**

**板书：**四行论据：每千行残留 2–6 Bug／找错改错占总投入 30%–60%／越早越便宜／单元可测得更彻底；右侧写两个算式：$0.83^{30}\approx0.00123$、$0.99^{30}\approx0.74$；底部一行大字"单元缺陷率被集成指数放大"。

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

**预设回答：**

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

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

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

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

**【思政落点 5】**：这两个算式其实是**成本意识的数学表达**。工程上最贵的一句话是"先这样，回头再说"——因为"回头"意味着这个缺陷已经流到了更贵的环节。请你们记住：**质量不是免费的，但劣质更贵**。花时间在单元测试上，看起来是"慢"，实际上是把后面几轮返工的钱提前省下来。这种"算总账而不是算眼前账"的习惯，将来你们做技术决策、做预算、甚至做人生选择，都用得上。

**过渡：**知道了为什么必须测，下一个问题就来了——测到什么程度算够？下一页给四条目标。

## 【5.35】单元测试的其它价值（3 分钟）

上一页讲"为什么做单元测试"——早发现、好定位、修得便宜，那是算成本账。这一页回答另一个问题：如果单元测试只是抓 bug，凭什么要求每个函数都配一条用例？课件列了七条价值，我挑四条讲透。

第一条，**支持扩展**。课件原话：没有各种类型测试的支持，软件开发无法支持扩展，单元测试就是它的基础。假设你手里是电商结算模块，一开始只有"单品下单"，半年后要加团购、预售、秒杀。有一套跑得起来的单元测试，你改完跑一遍、绿了就敢提测；没有这套测试，团队第一反应一定是"别动那块代码，谁动谁背锅"。**没有测试的系统，扩展能力是被冻住的**——不是不能加功能，是没人敢加。

第二条，**引导更好的设计**。课件说得硬气：能够支持单元测试的代码总是不会被人遗弃。你要给一个方法写测试，就必须能把它单独拿出来跑，这一"拿"就照出三种病：参数太多、Monster Method 巨型方法、依赖过多。一个八百行的 `processOrder`，光构造入参就要 new 十几个对象，你有两条路——放弃测试，或者拆成六个职责单一的小方法；一拆，设计就好了。业内叫**可测性倒逼设计**。

第三条，**避免回归缺陷**。课件说：集成为庞大系统后，靠系统测试穷尽回归几乎不可能。三十万行的系统改一行工具类，系统测试跑一遍六小时只能抽样；单元测试两千条用例八秒跑完，还能跑到每个分支。**它不保证你没写错，它保证你写错了马上知道**。

第四条，**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，也是一个例子，解释它怎么工作、怎么实现某条业务逻辑。新人接手三年前的模块，注释可能过期、设计文档可能没写，但测试用例是真的——输入 5 和 0 期望返回 5。**这是一份可执行的需求文档，而且是唯一不会过期的文档，因为它跑不通就会红**。

剩下三条一句带过：**支持变化**——谁改代码都得知道哪些部分还要验证，测试给了这份名单；**平稳步调**——代码和测试配合着写，很少有最后一分钟的意外；**节省时间**——边界值、输入校验、主逻辑路径，用单元测试做最廉价。课件还引了一句话收尾：人们总愿意写"工作完全正确"的代码，可证明程序完全正确几乎不可能，所以单元测试的目的不是证明正确，是**持续提供证据**。

> **板书：**中间写"单元测试 ≠ 抓 bug"，四周四条：① 支撑扩展（敢改）→ ② 倒逼设计（可测性 = 好设计）→ ③ 回归安全网（八秒两千条）→ ④ 可执行的行为文档（不会过期）；右下角小字"＋变化・步调・省时"。

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

> **预设回答：**

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

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

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

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

> **易错点 / 考点：**多选题考"单元测试的价值"，学生最容易漏掉"引导更好的设计"和"规范行为和记录代码"这两条，因为它们听起来不像"测试"的活。判断题"单元测试只能发现功能缺陷"是错的——它同样能暴露设计与可测性问题。记忆口诀：**扩展、设计、回归、文档**，四条一组背。

> **过渡：**价值讲起来容易空，落到"目标"就具体了。下一页把单元测试要达成的目标列成四条，其中第四条是一个数字，也是本课程最容易被误用的一条。

---

## 【5.36】单元测试的目标（3 分钟）

这一页是考点密集页，四条目标请记准，尤其第四条那个数字。

第一条，**单元模块被正确实现**，主要指编码，包括功能、性能、安全性，课件补一句"一般主要介绍单元功能测试"。为什么？性能受调用方式影响，安全要看整个数据流，单元层面难以完整验证。但"一般不测"不等于"不该测"：一个正则写错导致回溯爆炸、一处 SQL 拼接没参数化，都是单元层面爆的。所以准确说法是**功能是主战场，性能和安全做基础检查**。

第二条，**参数输入是否正确传递并得到保护，也就是容错；输出是否正常**。这条最容易被做成"只测好数据"。什么叫得到保护？空值、超长字符串、类型边界、非法格式进来时有没有被挡住，还是直接把脏数据带给下游。线上大量事故，根子不在算错，而在**根本没检查输入**。

第三条，**内部数据能否保持完整性**，课件给三个点：变量的正确定义与引用、内存及时释放、全局变量的正确处理和影响最低。这是单元测试独有的地盘——系统测试从外面只能看输入输出，看不到"这个对象释放了没有"。举个 Java 例子：`static` 的 SimpleDateFormat 被两个线程同时用，格式化出来的日期会串；这种

bug 黑盒测一百遍可能都不出来，一条并发单元测试就现形。

第四条，**覆盖率达标**：代码行覆盖、分支覆盖或 MC/DC 达到要求，比如高于 80% 或 95%。三个词要分清：行覆盖看这一行执行过没有；分支覆盖看判定为真为假两条路都走过没有；MC/DC 是修正条件/判定覆盖，要求每个条件都能独立影响判定结果，航空轨交这类高安全领域常用。80% 和 95% 不是及格线，而是不同完整性等级的要求——一般业务系统 80% 行覆盖是基本盘，安全关键模块要 95% 以上甚至要求 MC/DC。

但比数字更重要的是这句：**覆盖率是必要不充分条件**。100% 只说明每行都跑过，不说明每行都断言过。一段代码跑了、你没写 assertEquals，它错了你照样绿。所以决定质量的不是覆盖率，是**你断言了什么**。

**板书：**竖排四条目标：① 正确实现（功能为主，性能/安全做基础）→ ② 输入正确传递＋容错，输出正常 → ③ 内部数据完整性（变量/内存/全局）→ ④ 覆盖率（行・分支・MC/DC，80% 或 95%）；旁注一行红字"覆盖率高 ≠ 测得好，断言才算数"。

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

**预设回答：**

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

- 答"不能，可能没写断言"——对。执行到但没有断言的代码，等于走过场。

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

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

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

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

**过渡：**目标讲完了，我们马上动手。下一页这个小练习只有十几行代码，但它是本课程最经典的一道题——四个人里有三个第一次看不出来。

## 【5.37】一个小练习（2 分钟）

> 看这段代码。需求原文：实现两个**正整数或零**的加法，如果其中有 1 个数小于或等于 0，则返回 0。请找出错误，并说出什么测试数据能暴露它。
>
> 先读判定：`if(num1 >= 0 && num2 > 0)`。`num1` 是大于等于 0，`num2` 是**大于** 0。左边带等号，右边没带。而需求说零是合法的。所以 `num2 = 0` 时条件为假，程序走进 else 返回 0；正确答案应该是 `5 + 0 = 5`。**错误就在这一个等号上**。
>
> 这个 bug 为什么难发现？因为平时测函数下意识都用"正常值"：`add(2,3)` 得 5、`add(10,20)` 得 30、`add(-1,5)` 得 0，三个都对，测试通过，代码提交。要暴露它必须用**边界值**：`add(5,0)` 期望 5、实际 0，红了。零不是随便一个数，**零是两个合法区间的交界点**。
>
> 再深一层看覆盖。要让判定两条分支都跑到，最少两组数据：`(5,3)` 为真、`(-1,5)` 为假。可**判定覆盖做到了，这个 bug 依然活着**，因为 `num2=0` 这一格在"假"那一大类里被 `(-1,5)` 代表了。要让每个**条件**都取到真和假，得补 `(5,0)`——`num1` 真、`num2` 假。这就是条件覆盖比判定覆盖更强的地方，也解释了为什么"分支覆盖率 100%"的代码照样有 bug。
>
> 工程教训：修法不是把 `>` 改成 `>=` 就完事，而是**把判定写清楚**——先做参数校验 `if (num1 < 0 || num2 < 0) return 0;` 再放心相加。**边界条件写在开头，比藏在复合判定里安全**。
>
> 请当堂在纸上写三件事：错误在哪一行、暴露它的数据是什么、修好之后补哪几条用例。三分钟，我们对答案。

> **板书：**抄判定 `if(num1>=0 && num2>0)`，在 `num2>0` 上画圈打叉，旁边写 `>=`；下方列三行数据：`(5,3)→8 ✅`、`(-1,5)→0 ✅`、`(5,0)→0 ❌ 应为 5`；右侧写"判定覆盖 ≠ 条件覆盖"。

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

> **预设回答：**

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

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

(-1,5)` 就齐了。

- 答"没完成路径覆盖/组合覆盖"——也对，四种条件组合只走到三种组合里的一部分，要全覆盖得再加 `(0,0)`、`(0,3)` 这类数据。

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

> **过渡：**单个函数好测，因为它没有"邻居"。可真实的函数总要调用别人、也被别人调用。下一页开始讲：怎么给一个函数搭出"假邻居"，让它在孤岛上被测。

---

## 【5.38】驱动程序与桩程序（2 分钟）

> 这一页是标题页，但它是本节最关键的入口：单元测试的两件"脚手架"——**驱动程序（Driver）**和**桩程序（Stub）**。概念我先给一句话版，下一页看图。
>
> 先说为什么需要它们。单元测试的对象是一个单元，可能是一个函数、一个类。可现实中的单元不是孤岛：上面有调用者，下面有被调用者——要查数据库、发消息、调支付接口。做单元测试时，**上下两头都不在**：调用者还没写好，或者我不想连带它一起测；被调用者可能是别人的模块、可能是生产库、可能是收费的第三方接口。
>
> 怎么办？**上面缺一个"假调用者"，下面缺一个"假被调用者"**。假调用者就是驱动程序：准备数据、调用被测单元、拿回结果。假被调用者就是桩程序：顶替真实的下游，按我需要的样子返回结果。
>
> 打个比方：你要考一个学生的计算能力，又不想让其他环节干扰他，于是找个人专门出题收卷，这是驱动；再把"查表""计算器"换成一份固定答案表，让他不用真去查，这是桩。**驱动是上游的替身，桩是下游的替身**，一个喂进来，一个顶出去。
>
> 这两个词的方位，考试最爱设陷阱：**驱动力在被测单元之上**，它扮演"主程序"，通常由测试框架提供——你写的 `@Test` 方法本身就是驱动；**桩在被测单元之下**，它扮演"被依赖者"，需要你自己写小类去冒充。

> **板书：**中间画方框写"被测单元 Function under test"；上方箭头写"Driver（假调用者：造数据、发调用、收结果）"，下方箭头写"Stub（假被调用者：按需返回）"；右侧竖排一句"上一个、下一个，都不是真的"。

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

> **预设回答：**

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

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

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

> **易错点 / 考点：**名词方位必考。判断三步法：**看它调用谁**——它调用被测单元，是驱动；被测单元调用它，是桩。另一个高频错答是把"桩"和"Mock"当成同义词，这个差别我们在 5.59、5.60 两页专门讲。

> **过渡：**光说方位还是抽象，我们看那张三层结构图，一眼就明白。

---

## 【5.39】驱动程序与桩程序（2 分钟）

这张图是单元测试的标准骨架，三层，从中间说起。

中间一层是**被测单元**（Function under test），我们要真正测的那个函数或类，它是舞台中央的主角，不动。

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

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

为什么一定要这么搭？三条理由。**隔离**：一个单元测失败，你要能立刻确定问题在它自己身上，而不是"可能数据库挂了"。**可控**：真实下游你控制不了，它今天成功明天超时，你的测试就时红时绿，这种测试没人信；桩把"下游返回什么"变成一行代码。**速度**：连库的一条用例几百毫秒，用桩是微秒级，几千条用例就是半小时和几秒的差距。

补一个工程判据：**什么时候必须上桩？** 满足三条之一——跨进程（网络、数据库、消息）、不可重复（时间、随机数、流水号）、有副作用（扣款、发短信、写外部日志）。三条都不满足的，老实调真的，别为了"纯"而纯。

**板书：**三层堆叠图：上"Driver（准备数据→调用→判断输出）"、中"被测单元 Function under test"、下"Stub（固定返回值替代真实下游）"；右侧三条"必须上桩"：跨进程／不可重复／有副作用。

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

**预设回答：**

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

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

- 追问：这又回到 5.35 那条"引导更好的设计"——**为了可测，你把时间依赖抽出来了，代码反而更干净**。测试逼设计，就是这个意思。

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

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

**过渡：**桩写起来容易，写好很难。最常见的写法是"返回一个写死的 5"，下一页告诉你为什么这是最不聪明的做法。

---

## 【5.40】桩程序的灵活性（2 分钟）

这一页是 C# 代码，`ParameterizedStub`——参数化的桩。我们看它比"硬编码的桩"强在哪。

它长这样：类实现了协作接口 `ICollaborator`，构造函数收一个 `value` 存起来，然后在 `ComputeAndReturnValue()` 里判断——`value` 小于 10 就抛 `InvalidOperationException`，否则返回 `value`。关键在于：**返回什么由构造时传进来的参数决定**，不是写死在方法体里。

课件底下那句话很直接：不会像上一页返回一个硬编码数值；**让一个桩对象返回单个数值是最简单、但也是最不聪明的做法**，因为其它测试迟早都需要返回一个不同的数值。

为什么不聪明？第一个测试要返回 100，你写个 `return 100` 的桩；第二个要返回 0，再写一个类；第三个要抛异常，第三个类。三个测试三个类，代码量比被测代码还多，而且 90% 是一样的。这叫**测试代码的复制粘贴**——接口一改，你要改三处。

参数化之后一个类搞定：`new ParameterizedStub(100)`、`new ParameterizedStub(0)`，要异常就传 5。**变化的部分从"类的个数"变成了"构造参数"**，这就是灵活性。

这段代码还有两个细节值得学：第一，它把**异常路径**也做进了桩——真实下游会失败，你的桩也要能失败，否则容错代码永远测不到；第二，它的判断逻辑是"假业务"，但这是刻意的——桩只需要满足测试需要的行为，不必还原真实逻辑。

一句总结：**桩的价值不在"能返回"，而在"能按需返回"**。写桩前先问自己：下一个测试会不会要一个不同的值？会，就别硬编码。

**板书：**左侧写"硬编码桩：`return 100` → 每个用例一个类 → 复制粘贴"；右侧写"参数化桩：`new ParameterizedStub(100/0/5)` → 一个类覆盖正常・边界・异常"；中间一句"变化点从『类的个数』移到『构造参数』"。

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

**预设回答：**

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

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

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

**易错点 / 考点：**常考"硬编码桩的问题"——标准答案是"复用性差、用例一多就要写多个桩类、维护成本高"。

另一个易错点是把"参数化桩"和"Mock 的 when 配置"混为一谈：前者是你自己写的类，后者是框架动态生成的代理对象，机制不同、效果相近。

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

---

## 【5.41】单元测试工具及其应用（3 分钟）

这一页是本节的分隔页，任务只有一个：把"工具"的地图铺开。

为什么单独立一节讲工具？因为单元测试是**所有测试类型里最依赖工具的一种**。系统测试还能靠人点、靠脚本点；单元测试面对几百上千个函数，手工根本不可能。业内说法是：**没有框架就没有单元测试**——你不可能自己造一套"发现用例、执行、捕获异常、汇总报告"的机制，那本身就是个框架。

这一节带你过四类工具，后面每页都能归位：

第一类**单元测试框架**——JUnit、TestNG、GoogleTest、pytest，负责组织用例、执行、断言、报告，这是地基。第二类**测试替身与 Mock 框架**——JMockit、Mockito、EasyMock，负责造假协作者，把上一页我们手写桩的活自动化。第三类**用例自动生成工具**——EvoSuite、Diffblue Cover，基于遗传算法或符号执行自动生成能提高覆盖率的测试，注意它们是**起点不是终点**。第四类**覆盖率工具**——JaCoCo、gcov/lcov、Coverage.py，回答"我到底测到哪儿了"。

四类的关系一句话串起来：**框架让你写得出，替身让你写得下去，生成工具让你写得快，覆盖率工具让你知道写够了没有**。四件套缺一件，工程上都跑不起来——只有框架没覆盖率，你不知道测够没有；只有覆盖率没替身，稍微复杂的类根本测不动。

顺带说本课程的实操路线：课堂和实验用 Java 生态，**JUnit 5 + Mockito + JaCoCo**。这三样在 Java 项目里占有率最高，学一套够用很久。

**板书：**四行分类：① 框架（JUnit／TestNG／GoogleTest／pytest）② 替身与 Mock（Mockito／JMockit／EasyMock）③ 自动生成（EvoSuite／Diffblue Cover）④ 覆盖率（JaCoCo／gcov／Coverage.py）；右侧一句话"写得出・写得下去・写得快・知道够没够"。

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

**预设回答：**

- 答"覆盖率工具"——理由通常是想先量化现状。可行，但没有框架就没有用例可跑，覆盖率会是 0，量化的意义不大。

- 答"测试框架，比如 JUnit"——多数情况下的正解。框架是地基，其它三类都是它的插件或配套。

- 答"自动生成工具，一步到位"——听着高效，实际风险最大：生成的用例断言往往是"照抄当前行为"，**它会把 bug 一起固化下来**。

**易错点 / 考点：**这一页的考点是"工具分类"。最容易错的是把 **EvoSuite／Diffblue Cover** 归到"Mock 工具"里——它们是用例生成工具。另一个常见错答是把 JaCoCo 说成"测试框架"，它是覆盖率库，靠 JVM agent 插桩工作。

**过渡：**分类讲完了，下一页把这几类的具体名字摆出来，出处是维基百科那份单元测试框架清单。

---

## 【5.42】单元测试工具（2 分钟）

这一页只给了一个出处——维基百科的"单元测试框架列表"。课件把它当入口，不是当要背的清单。我借它讲三件事。

第一，**"框架列表"本身就说明行业成熟度**。一个语言生态成不成熟，看它有没有成体系的单元测试框架就够了：Java 有 JUnit、TestNG；C++ 有 GoogleTest、Catch2；Python 有 unittest、pytest；JavaScript 有 Jest、Mocha；Go 更狠，`go test` 直接做进语言工具链。**越主流的语言，测试框架越像空气——你不用特意选，它就在那儿**。

第二，**选框架看三件事，不看功能多少**：① 生态——IDE 支不支持、Maven／Gradle 能不能一键跑、能不能出覆盖率报告；② 团队熟悉度——团队用 pytest 用得顺，你非换 JUnit 就是纯增成本；③ 与 CI 的集成成本——能不能只跑失败的用例、能不能产出标准报告格式。这三条比"哪个断言 API 更优雅"重要一百倍。

第三，**这份清单是历史地图**。很多框架名字带 xUnit——JUnit、NUnit、xUnit.net、PHPUnit，都源自 Smalltalk 的 SUnit 那套设计：测试类、测试方法、断言、前后置处理、测试套件、结果汇总。**这套设计活三十年没被推翻**，说明它抓住了本质：测试就是"准备—执行—断言—清理"四步，任何框架都是给这四步换皮。

所以这一页当**坐标系**用：以后换语言、换公司、换项目，先找"这个生态的 JUnit 是谁"，剩下可以平移。

> **板书：**写"xUnit 家族"→ 下面拉四条：JUnit（Java）／NUnit・xUnit.net（.NET）／GoogleTest（C++）／pytest・unittest（Python）；右侧写"选框架看三条：生态・熟悉度・CI 集成"，下方一行"四步不变：准备→执行→断言→清理"。

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

**预设回答：**

- 答"那就别用这门语言"——结论可能对，但理由要更细：没有框架意味着团队要自己搭测试基础设施，成本会翻几倍。

- 答"风险高，因为没法做自动化测试"——对，而且后果是**回归只能靠人**，人一累就漏，漏了就在线上见。

- 追问：所以选技术栈时，"有没有成熟的测试生态"应该和"性能好不好""社区活不活跃"放在同一张评估表里——这也是很多团队踩过坑才学会的。

**易错点 / 考点：**这一页不考清单，考的是"框架的共同结构"。常考简答："xUnit 系列框架一般包含哪些要素？"标准答案：测试类与测试方法、断言机制、前置/后置处理（setUp/tearDown）、测试套件组织、结果报告。别答成"JUnit 有 @Test"这种具体到某一家的细节。

**过渡：**清单太宽，我们把镜头收窄到 Java 和 Python 上最常用的那几个名字，看它们的定位差别。

---

## 【5.43】最常用的单元测试工具（3 分钟）

这一页是干货清单，我按四类工具给你归位，说清每个工具**解决什么问题**。

第一类**框架**：JUnit、TestNG 是 Java 两大主力，GoogleTest 是 C++ 标杆，pytest 和 unittest 是 Python 两条路——unittest 是标准库自带、风格像 JUnit，pytest 更 Pythonic、直接写 `assert`。职责都是组织、执行、报结果。

第二类**智能化用例自动生成**，课件点了两个：**EvoSuite** 用遗传算法搜索能提高覆盖率的输入，最后生成 JUnit 用例；**Diffblue Cover** 是商业产品，主打给存量 Java 代码自动补测试。定位我说重一点：**它们是起点工具，不是终点工具**。因为它们生成的断言，本质是从当前代码行为反推的——**代码错了，它就把错误行为写成期望值**。所以生成完必须由人审断言，尤其业务规则那几条。

第三类 **Mock 工具**，课件写的是 **JMockit**：能力最强，能 mock 静态方法、构造器、私有方法，但和 JVM 版

本耦合最紧。后面重点讲更主流更稳的 Mockito。

第四类**覆盖率工具**：Java 是 **JaCoCo**；C/C++ 是 **gcov**，配 **lcov** 出 HTML、**gcovr** 做汇总和 XML；Python 是 **Coverage.py**。原理都是"埋点统计哪些行被执行过"，具体机制后面讲。

课件那句要划重点的话：Java、C++、Python 里受欢迎的就是这四类——**框架、用例自动生成、Mock、覆盖率**。各认一个主力，工具链就齐了：JUnit + Mockito + JaCoCo，也是我们实验用的组合。

顺手纠一个笔误：前面第 18、19 页把 SonarLint 和 SonarQube 写成了 `SolarLint`、`SolarQube`，那是拼错了，正确是 **SonarLint**（IDE 里的静态检查插件）和 **SonarQube**（服务端代码质量平台），答题写报告按正确拼写来。

**板书：**四栏对照表：框架（JUnit／TestNG／GoogleTest／pytest／unittest）｜自动生成（EvoSuite／Diffblue Cover）｜Mock（JMockit）｜覆盖率（JaCoCo／gcov・lcov・gcovr／Coverage.py）；底部一行"实验组合：JUnit 5 + Mockito + JaCoCo"；右上角订正框"SonarLint／SonarQube（课件误拼 SolarLint／SolarQube）"。

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

**预设回答：**

- 答"能，覆盖率涨了就是好事"——风险最大。自动生成的断言固化的是**现状**，不是**需求**；如果现状本身有 bug，你就把 bug 写成了"期望行为"。

- 答"不能，要人工审断言"——正解。至少要抽查业务规则相关的用例，确认期望值是从需求来的。

- 答"可以先合并，再慢慢审"——不推荐。一旦进了主干，后面所有人都以为这是"正确行为"，改起来阻力更大。宁可分批合，也要先审。

**易错点 / 考点：**最爱考的是**工具归类**：EvoSuite／Diffblue Cover 属于"用例自动生成"，不是"Mock 工具"也不是"覆盖率工具"；JMockit 是 Mock 工具，不是框架。第二个考点是 **gcov 家族的分工**：gcov 采集、lcov 出报告、gcovr 出汇总——三者别混答。

**过渡：**工具认完了，回到 Java 的主线。下一页开始讲 JUnit，先从它的老版本 JUnit 4 的工作原理说起，因为那几个类名到现在还在面试题里出现。

---

## 【5.44】JUnit 4 工作原理（2 分钟）

> 这一页是标题页，我先给总纲，下一页看那四个类的分工。
>
> 为什么讲 JUnit 4 而不是直接讲 5？两个原因。第一，**JUnit 5 是 2017 年才发布的，绝大多数存量项目和面试题还是 JUnit 4 的口径**——你以后接手老代码，看到的还是 `org.junit.Test`、还是 `@Before`／`@After`／`@RunWith`。第二，JUnit 5 的架构是在 4 的基础上拆出来的，**你懂了 4 的"四件套"，才看得懂 5 的三层架构为什么这么分**。
>
> JUnit 4 的核心设计一句话：**用注解标记测试，用运行器驱动执行，用结果对象收集结论**。注解是 `@Test`、`@Before`、`@After`、`@BeforeClass`、`@AfterClass`、`@Ignore`；运行器是 `Runner` 及其子类；结果是 `Result` 和 `RunListener`。
>
> 这里有个设计上的小心思：**一个测试方法失败，不能影响其它方法继续跑**。所以 JUnit 给每个方法套了一层隔离——新建实例、捕获异常、记录下来、继续下一个。你在 `@Test` 里抛异常，框架不会崩，它只是把这个方法标成"错误"。这就是"框架"和你自己写个 main 循环调用最大的差别。
>
> 下一页把参与这件事的四个类摆在一起：`TestRunner`、`TestSuite`、`TestCase`、`TestResult`。课件管它们叫"成员三重唱"，实际上是四个角色。

> **板书：**写"JUnit 4 = 注解 + 运行器 + 结果"；下方三行：`@Test`／`@Before`／`@After`／`@BeforeClass`／`@AfterClass`／`@Ignore`；`Runner`、`Result`、`RunListener`；右侧一句"一个方法失败，不影响其它方法"。

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

> **预设回答：**

- 答"为了隔离，避免上一个方法改了字段影响下一个"——正确，这是最核心的理由，测试方法之间不能有隐式依赖。

- 答"因为 Java 只能这么写"——不是，共用实例技术上完全可行（JUnit 4 里用 `@FixMethodOrder` 也只是控制顺序，不共享状态）。

- 追问：那如果某个对象**很贵**、所有方法都要用，怎么办？答案是 `@BeforeClass` 级别的静态资源，或者干脆用依赖注入把共享部分管起来——**共享要显式写出来，不能靠"反正上一个方法已经初始化了"**。

> **易错点 / 考点：**常考"JUnit 4 与 JUnit 5 的注解对应关系"：

`@Before`→`@BeforeEach`、`@After`→`@AfterEach`、`@BeforeClass`→`@BeforeAll`、`@AfterClass`→`@AfterAll`、`@Ignore`→`@Disabled`。这个对照表必须背下来，选择题年年出。

> **过渡：**四个角色怎么串成一条流水线，看下一页。

---

## 【5.45】JUnit 工作原理 – 续（3 分钟）

这一页是 JUnit 4 的核心机制。课件原话：**一个 `TestRunner` 运行一个 `TestSuite`，这个 `TestSuite` 可以由一个或多个 `TestCase`（或由其它 `TestSuite`）组成；运行结果由 `TestResult` 收集，由 `TestRunner` 报告。**四个角色，我用考试场景对齐，一次记住。

**`TestCase` 是考生**。一个测试类，里面每个 `@Test` 方法是一道题。它知道怎么答题（执行方法）、怎么自评（断言）。

**`TestSuite` 是考场**。它把考生组织起来，而且可以嵌套——套件里装套件，就像"上午场"里分"数学考场""语文考场"。这就是你能按包、按类、按标签筛选用例的原因。

**`TestResult` 是成绩单**。它只干一件事：收集。收集每道题的结论，结论分三类：**成功、失败（failure）、错误（error）**。

这里有个必考区分：**failure 和 error 不是一回事**。failure 是断言不成立——程序跑完了，结果与期望不符。error 是执行中抛了异常——程序根本没跑到断言，这可能是被测代码的问题，**也可能是测试代码自己写错**。所以 `Failures: 2, Errors: 1` 是两种不同性质的信号。

**`TestRunner` 是监考老师兼广播员**：启动流程，然后报告结果——控制台那一条条绿点红叉、IDE 左边那棵树、CI 上那份报告，都是它报的。

这里还藏着一个架构洞察：`TestSuite` 用**组合模式**把"单个用例"和"一组用例"统一成同一个 `Test` 接口。所以跑整个项目、跑一个包、跑一个类、跑一个方法，对上层是同一种操作。**这个设计被 JUnit 5 继承下来，变成了 `TestEngine` 发现并执行 `TestDescriptor` 树**——你现在理解的这套结构，换个名字还能用十年。

> **板书：**四个角色对齐：`TestCase`＝考生（一个测试类／一个 @Test 一道题）→ `TestSuite`＝考场（可嵌套）→ `TestResult`＝成绩单（成功／failure／error）→ `TestRunner`＝监考＋广播（启动＋报告）；右侧一句"组合模式：单个和一组，都是 Test"。

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

> **预设回答：**

- 答"三个都失败了，去改代码"——太粗。Errors 表示压根没跑到断言，先要看异常栈。

- 答"Errors 是抛异常，可能是测试代码自己写错了"——很准。常见的是桩没配对、构造参数给错、依赖没注入，这类问题先看"是不是测试环境的问题"。

- 答"Failures 才是真失败，Errors 是环境问题"——不完全对。Errors 也可能是被测代码的空指针，那就是真缺陷。所以正确姿势是**先看异常类型和栈顶**，再判断归属。

> **易错点 / 考点：**必考区分 —— **failure（断言失败）vs error（异常中断）**；以及 **TestSuite 可以包含 TestSuite**（组合模式）。还有个常见错答："TestResult 负责打印报告"——不对，**收集是 TestResult，报告是 TestRunner**，这两个职责在 JUnit 5 里被拆得更清楚。

> **过渡：**JUnit 4 是一整块，JUnit 5 把它拆成了三层。这个拆分是理解现代测试框架的钥匙。

---

## 【5.46】JUnit 5 架构（3 分钟）

> 这一页是架构核心，一句话必须背下来：**JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**。这不是三个版本，是**三个职责不同的子项目**，今天拼在一起发布。
>
> 第一块，**JUnit Platform**，基于 JVM 的执行测试的基础框架及其 Test-Engine API。翻成人话：它是**插座和电网**，不管你怎么写测试，只负责发现测试、调度执行、把结果吐出来。它还提供两样：一个**控制台启动器**（命令行就能启动平台），以及给 **Gradle 和 Maven 构建插件的支持**——这就是 `mvn test` 能跑、也能接进 CI/CD 的原因。还有：Platform 自带一个**基于 JUnit 4 的 Runner**，旧测试不用改就能跑。
>
> 第二块，**JUnit Jupiter**，是 JUnit 5 里编写测试和扩展的新编程模型与扩展模型的组合，并提供 Test-Engine 让基于 Jupiter 写的测试在平台上运行。翻成人话：**新国标插头**——你天天写的 `@Test`、`@BeforeEach`、`Assertions.assertEquals`、`@ParameterizedTest` 全来自它，是你最常打交道的一块。
>
> 第三块，**JUnit Vintage**，提供 TestEngine 让平台能跑**基于 JUnit 3 和 JUnit 4 的测试**。翻成人话：**转接头**。公司有十万条存量用例，不可能一夜重写，Vintage 就是那座桥。

为什么这个拆分是神来之笔？它把**执行**和**编写**彻底解耦了。Platform 只认 Test-Engine 这个接口——谁实现它谁就能上平台，于是 TestNG、Spock 都能挂上来。**这是"面向接口而不是面向实现"的一次设计胜利**。两个实践点：Vintage 是**可选依赖**，纯新项目可以不引；从 4 迁 5 的标准做法是"先引 Vintage 保证存量能跑，再逐模块改写，最后摘掉"——**不要停机大改**。

**板书：**三层结构图：上"JUnit Jupiter（新编程模型：@Test／@BeforeEach／Assertions）"、中"JUnit Platform（Test-Engine API＋控制台启动器＋Gradle/Maven 插件＋JUnit4 Runner）"、下"JUnit Vintage（跑 JUnit 3／4 的转接头）"；右侧一句"执行与编写解耦：谁实现 Test-Engine，谁就能上平台"。

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

**预设回答：**

- 答"三块都引"——正确。Jupiter 跑新的，Vintage 跑旧的，Platform 调度。

- 答"只引 Platform＋Jupiter 也行，JUnit 4 的用例会被忽略"——现象描述对，后果要说清：**旧用例不是"跑失败"，是"根本没被发现"**，覆盖率会莫名掉一大截，这种"静默丢失"最危险。

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

**易错点 / 考点：**必考"JUnit 5 由哪三部分组成及各自职责"。三个高频错答：① 说"JUnit 5 就是 Jupiter"（漏了 Platform 和 Vintage）；② 把 Vintage 说成"旧版本兼容包"就完了——要说出它是"提供 TestEngine 让平台跑 JUnit 3/4 测试"；③ 把 Platform 说成"IDE 插件"。记忆口诀：**平台管跑、朱庇特管写、复古管兼容**。

**过渡：**架构是骨头，接下来是肉——你每天真正会用到的那三个组件。

---

## 【5.47】需要了解 JUnit 的关键组件（2.5 分钟）

这一页把 JUnit 拆成三个你每天真正会碰的组件：**Assert 类、Annotation、Runner**。

**Assert 类**。课件定义：它是 JUnit 框架中非常重要的类，用于判断所返回的**实际结果**与**预期结果**是否相符。注意主语——它判断的是"结果"，不是"过程"。所以断言是整个测试的**判决环节**：前面准备数据、调用方法都是取证，断言才是宣判。**没有断言的测试不是测试，是"能跑通就算过"的演示代码**。

**Annotation**。课件定义：JUnit 中以 `@` 引导，用来标注测试类中不同方法的功能。一句话说清它的价值：**它把"这个方法是什么角色"从命名约定变成了显式声明**。JUnit 3 时代方法名必须以 `test` 开头才会被执行；JUnit 4 之后你写 `@Test` 就行，方法名叫 `should_return_zero_when_num2_is_zero` 也没问题——名字终于能用来表达意图，而不是讨好框架。

**Runner**。课件定义：JUnit 通过运行器 Runner 来运行和显示测试执行结果。它是"发动机＋仪表盘"，负责启动，也负责呈现。JUnit 5 里它的职责被 Platform 的 Test-Engine 接管了，但概念没变——总有一个东西在发现并驱动你的测试。

三者是一个闭环：**Annotation 告诉框架有哪些测试，Runner 把它们跑起来，Assert 决定哪条通过**。缺一不可——没注解框架找不到你，没运行器跑不起来，没断言永远绿。

这也是排查"测试没生效"的三步法：先看注解加没加（是不是被误标 `@Disabled`）、再看运行器扫到没（包路径、命名过滤）、最后看断言写没写。

> **板书：**三角关系图：`Annotation`（有哪些测试）→ `Runner`（跑起来＋报结果）→ `Assert`（判通过）；下方写排查三步："注解加没加 → 运行器扫到没 → 断言写没写"。

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

> **预设回答：**

- 答"看代码里有没有 assert"——最快最直接。

- 答"故意把断言改成错的，看它红不红"——更工程化的做法，叫**变异测试**思路：把期望值改错，测试还是绿，就说明这条用例是空壳。

- 答"看覆盖率"——不够。空壳用例照样能把行覆盖率跑满，这正是 5.36 讲的"覆盖率骗人"。

> **易错点 / 考点：**常考"JUnit 三个关键组件及各自作用"，注意 **Assert 判断的是实际与预期是否相符**、**Annotation 以 @ 引导**、**Runner 负责运行并显示结果**这三个标准表述。易错点是把 Assert 说成"捕获异常的工具"——捕获异常是运行器的事，Assert 只做判断。

> **过渡：**三个组件里，注解是你写得最多、也最容易用错的一个。我们把它逐个过一遍。

---

## 【5.48】注释 Annotation（2.5 分钟）

这一页是注解清单，看着像背单词，其实有结构。我先给结构，再点必会的。

课件第一句：Annotation 清晰地表达测试程序的**逻辑结构和功能**。"逻辑结构"这四个字是关键——注解不只标记"这是个测试"，它还在描述这个测试在类里的位置和生命周期。按四组记：

**用例本身**：`@Test` 是基础；`@ParameterizedTest` 参数化，一条用例跑多组数据，是**减少重复代码最有效的手段**；还有 `@RepeatedTest`、`@TestFactory`、`@TestTemplate`。

**生命周期**：`@BeforeEach`／`@AfterEach` 每个方法前后各跑一次；`@BeforeAll`／`@AfterAll` 整个类只跑一次，默认必须是静态方法。这四个的差别是本节最常考的点——**"每个方法一次"还是"整个类一次"**。

**组织与展示**：`@DisplayName` 给用例起人话名字；`@Nested` 嵌套分组；`@Tag` 打标签筛选（CI 里排除 `slow`）；`@TestMethodOrder`／`@TestClassOrder` 控制顺序。

**控制与扩展**：`@Disabled` 禁用；`@Timeout` 超时即失败——很多人不知道，但它是**防测试挂死的关键**；`@ExtendWith`／`@RegisterExtension` 注册扩展；`@TempDir` 自动建临时目录并在测试后清理，测文件读写必备。

一个使用建议：**注解要"少而准"**。一个类里 `@Disabled` 一大堆、`@Timeout` 一个没有、`@DisplayName` 全不写，那份报告对别人就没有可读性。注解成本极低，收益全在"别人看得懂"和"以后改得动"。

> **板书：**四组分类竖排：① 用例（@Test・@ParameterizedTest・@RepeatedTest・@TestFactory・@TestTemplate）② 生命周期（@BeforeEach・@AfterEach・@BeforeAll・@AfterAll）③ 组织展示（@DisplayName・@Nested・@Tag・@Order）④ 控制扩展（@Disabled・@Timeout・@ExtendWith・@TempDir）；右下角一行"每方法一次 vs 每类一次"。

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

> **预设回答：**

- 答"`@BeforeEach` 放每个用例都要的初始化，`@BeforeAll` 放贵重的共享资源"——标准答案，比如数据库连接池、容器启动放 `@BeforeAll`，测试数据构造放 `@BeforeEach`。

- 答"非 static 会报错"——对，JUnit 5 默认生命周期是"每个方法一个新实例"，静态方法才活得比实例长；除

非加 `@TestInstance(Lifecycle.PER_CLASS)`。

- 追问：那为什么不干脆全用 `@BeforeAll` 图省事？因为**共享状态会带来用例间的隐式依赖**——第 3 条用例失败可能因为第 1 条改了数据，这种 bug 排查起来极度痛苦。

> **易错点 / 考点：**必背两组对照：JUnit 4 → 5 的注解映射（见 5.44）；以及 **`@BeforeEach`（每方法）与 `@BeforeAll`（每类）的区别**。另一个高频错点：`@ParameterizedTest` 要配合 `@ValueSource`／`@CsvSource`／`@MethodSource` 提供数据，**单独写 `@ParameterizedTest` 是跑不起来的**。

> **【思政落点 8】**：注解看起来只是几个 `@` 符号，但它体现的是**规范意识**。`@Test` 让框架找得到你的测试，`@DisplayName` 让同事看得懂你的意图，`@Timeout` 让一条挂死的用例不至于拖垮整条流水线——这些都不是"技术难度"，而是"愿不愿意按约定把事做完整"。工程里最贵的成本从来不是写不出高深算法，而是**别人接手你的代码时要猜**。请把"让别人少猜一次"当成写代码的默认标准，这就是职业素养的日常形态。

> **过渡：**注解管"怎么组织和跑"，断言管"算不算过"。下一页是断言的老家：Assert 类。

---

## 【5.49】Assert 类（2.5 分钟）

> 这一页讲断言，先把课件那两句核心讲透。
>
> 第一句：**检查并判断程序的运行结果是软件测试中的一项重要工作**。测试程序必然包含断言条件，运行时不满足，就表明结果与期望不一致，程序中存在缺陷。这句话的分量在于——**断言是缺陷判定的唯一出口**，你不写断言，框架就不知道什么叫错。
>
> 第二句是机制：JUnit 通过 Assert 类提供一系列断言方法判断运行结果，该类位于 `junit.framework` 包中；参数包括**期望变量 `expected`** 和**实际变量 `actual`**，两者相等则相符，否则用例失败。
>
> 注意版本口径：`junit.framework` 是 JUnit 3/4 的包路径，JUnit 5 对应 `org.junit.jupiter.api.Assertions`，方法名基本沿用。**答题按教材写 `junit.framework`，写代码用 `Assertions`**，两个都对。
>
> 实战最容易踩的坑：**`assertEquals(expected, actual)` 的参数顺序**。写反了不会报错——通过时结果一样，只有**失败时**你才会看到一句完全相反的报错："expected: 5, but was: 3"，而你脑子里想的是"应该是 3 啊"，排查方向彻底跑偏。给你一条硬规矩：**第一个参数永远是"我期望的"，第二个永远是"实际跑出来的"**。

再补三个常用断言：`assertTrue`／`assertFalse`判真假；`assertNull`／`assertNotNull`判空；`assertThrows`判"这段代码应该抛某个异常"——它让异常路径也能被断言。JUnit 5 的`assertAll`能把多条断言打包，**一次报出全部失败项**。

最后一个观念：**断言要具体**。`assertNotNull(result)`只说明有东西返回，`assertEquals(5, result)`才说明返回对了。**覆盖靠执行，质量靠断言**。

> **板书：**写`assertEquals(expected, actual)`并用箭头标"① 期望 ② 实际"；下方列：`assertTrue/False`、`assertNull/NotNull`、`assertThrows`、`assertAll`；右侧两行红字"参数别写反""断言要具体：`assertNotNull` ＜ `assertEquals`"。

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

> **预设回答：**

- 答"没什么区别，相等就行"——只在通过时没区别。失败时报告会反着说，让人误判。

- 答"失败信息会颠倒，排查方向错"——正确。这是最实际的危害。

- 答"JUnit 5 有重载能传 message，写清楚就行"——补得好，但 message 是辅助，**参数顺序是契约**，不能靠 message 兜底。

> **易错点 / 考点：**三个必考点：① `assertEquals(expected, actual)`的**参数顺序**；② Assert 类的**包路径**（教材口径`junit.framework`，JUnit 5 为`org.junit.jupiter.api.Assertions`）；③ 断言方法"相等则通过、不等则失败"的判定逻辑。判断题常出："断言失败就说明被测程序一定有缺陷"——严格说还要排除"测试代码本身写错"的可能，所以这句话在考试里通常判错。

> **过渡：**断言在 JUnit 4 里不是一个孤立的类，它被 TestCase 继承了——这就把断言和用例绑在了一起。下一页看这三个类的关系。

---

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

这一页把 5.45 讲的那三个角色落成三个具体的类，我逐条讲定义，并补一句"为什么这样设计"。

**`TestCase`类**：JUnit 框架中为测试用例提供服务的核心类，位于`junit.framework`包中，并且**继承自**

**Assert 类**——所以它可以直接使用 Assert 里的断言方法。更关键的是：编写的测试类**都需直接或间接继承于 `TestCase`**，依靠它提供的办法来实现测试方法的运行与判断。

注意"继承自 Assert"的意义：**它把断言能力变成了测试类的天然属性**。写 `extends TestCase` 就能直接 `assertEquals(...)`，不用带类名。JUnit 3 就是这么用的——必须继承、方法名必须 `test` 开头。JUnit 4 之后改注解、不再强制继承，但**设计思路留了下来**：测试类＋断言库这个组合没变。

**`TestSuite` 类**：实现了 JUnit 的 `Test` 接口，用于管理每一个测试用例。关键词是"实现 `Test` 接口"——这就是**组合模式**：两者实现同一个接口，所以套件能装套件，你能一次跑一个方法、一个类、一个包、一个项目。

**`TestResult` 类**：收集并记录所有测试用例的运行结果，课件明确分三类：**成功、失败、错误**。实践建议：**看报告先看 Errors 再看 Failures**，因为 error 往往意味着这条用例根本没验到东西。

三个类连起来是一条链：`TestSuite` 组织 `TestCase`，`TestCase` 执行时把结论写进 `TestResult`，最后由 `TestRunner` 读 `TestResult` 报告。**组织—执行—记录—呈现**，四个动作四个对象。

最后提醒概念迁移：JUnit 5 里这三个类基本不直接用了，分别被注解、引擎发现机制、`TestExecutionSummary` 取代。但它们**在结构上仍然存在**，所以考题还在问。

> **板书：**三行三定义：`TestCase`＝测试用例核心类・继承 Assert・测试类需直接或间接继承它；`TestSuite`＝实现 Test 接口・管理用例・可嵌套；`TestResult`＝收集记录结果・成功／失败／错误；底部一条链"组织→执行→记录→呈现"。

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

**预设回答：**

- 答"好处是随取随用，不用 import 断言类"——对，这是便利性。

- 答"代价是测试类被迫继承一个框架类，耦合了"——很到位。这就是 JUnit 4 改用注解、JUnit 5 改成静态导入的原因：**测试类不应该被框架绑死继承链**。

- 追问：所以从 JUnit 3 到 JUnit 5 的演进主线是什么？——**从"继承框架"到"注解标记"，再到"组合扩展"**，一路在解耦。

**易错点 / 考点：**必背三句定义（尤其 `TestCase` **继承自 Assert** 这一条，最容易被漏）；必考区分 **`TestResult` 收集 vs `TestRunner` 报告**；以及 `TestSuite` **实现了 Test 接口**这个细节，它是"套件能套

套件"的技术前提。

**过渡：**理论讲够了，看一段真实的 JUnit 5 测试脚本，把注解、断言、准备清理全用一遍。

---

## 【5.51】单元测试脚本（以 Junit 5 为例）（2 分钟）

这一页是一段完整的 JUnit 5 测试脚本，我按结构拆，你以后照这个骨架填。

第一层是**类**：通常跟被测类同名加 `Test` 后缀，类上可以用 `@DisplayName` 写人话标题。

第二层是**准备**：`@BeforeEach` 里造数据、建对象。原则是——**每条用例都从同一个干净状态出发**。如果你发现某条用例必须接着上一条的数据才能跑，那不是测试，是连续剧，早晚会崩。

第三层是用例本体，每个 `@Test` 方法内部一定是三段式，业内叫 **AAA：Arrange（准备）— Act（执行）— Assert（断言）**。读代码就找这三段：前面 new 对象设参数是 Arrange，中间那行调用被测方法是 Act，后面 `assertEquals` 是 Assert。

为什么死守 AAA？它是**可读性的骨架**。一条测试挂了，你要在十秒内判断"是数据错了、调用错了、还是结果真错了"，三段分明一眼定位；三段混在一起就得逐行读。

第四层是**清理**：`@AfterEach` 释放资源、删临时文件。更省事的是 `@TempDir`，框架自动建目录并在测试后清掉。

两个实践细节：① **一个测试方法只断言一件事的主题**，围绕同一个行为；一条用例既测加法又测减法，挂了你说不清哪个坏。② **方法名就是文档**：`should_return_zero_when_second_arg_is_zero` 比 `testAdd2` 强一百倍，配 `@DisplayName` 中文描述，报告直接能当验收证据。

一句总结：**注解管结构，AAA 管顺序，断言管判定，命名管可读**。

**板书：**抄骨架四层：类（`@DisplayName`）→ `@BeforeEach`（造数据）→ `@Test`（Arrange・Act・Assert）→ `@AfterEach`／`@TempDir`（清理）；右侧一行"一个方法一个主题；方法名即文档"。

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

**预设回答：**

- 答"太长了，不好读"——表象。真正的问题是：**Arrange 太长说明被测对象的依赖太多**，回到了 5.35 那条"引

导更好的设计"。

- 答"可能该抽成 @BeforeEach 或工厂方法"——对，重复的构造逻辑应该提取。

- 答"没问题，只要断言对就行"——短期能跑，长期没人敢改：改一处构造就要动三十行。

> **易错点 / 考点：**实验和卷面都常考 **AAA 三段式**，答题时要说清每段职责（准备数据／调用被测方法／断言结果）。易错点：把 `@BeforeEach` 当成"每个类执行一次"（那是 `@BeforeAll`）；以及忘记 JUnit 5 里 `@BeforeAll` 标注的方法默认必须是 **static**。

> **过渡：**单个类测起来还算舒服，可一旦被测类要调用别人的服务，麻烦就来了——这就进入 Mock 的世界。

---

## 【5.52】Mock 技术（2 分钟）

> 这一页是分隔页，任务是给一个总印象：**Mock 技术解决的是"依赖"这个麻烦**。
>
> 先把问题摆清楚。一个典型的业务方法：查数据库拿订单、调风控判断能不能放行、调支付网关扣款、发消息通知仓库。你要测它，四个下游全在别人手里——数据库没你要的数据、风控在灰度、支付网关收费、消息队列没起。三条路：**搭全套环境**（贵、慢、脆）、**不测**（欠债）、**用替身**（Mock 技术）。
>
> Mock 技术就是第三条，把下游换成可控的替身对象。前面手写桩是最原始的做法；接下来要讲的是一整套替身的分类学：Fake、Dummy、Double、Spy、Stub、Mock，六个词是本节核心，也最容易混。
>
> 先给一句话地图，听后面几页不迷路：**Dummy 是凑数的、Stub 是给答案的、Spy 是记账的、Fake 是能真跑但简化的、Mock 是既给答案又验交互的**。`Test Double` 是这五个的总称——就像"车辆"是轿车卡车摩托的总称。
>
> 为什么值得花一整节？因为它决定你写测试的**成本和质量**。用一个 Fake 能让几百条用例又快又真；而滥用 Mock 会把测试变成"验证我自己的调用代码"，一重构就全红，业内叫**过度 Mock 的脆性测试**。所以目标不只是会分类，而是**会选**——这是本章的一个难点。

> **板书：**中间写"Test Double（测试替身）"；五个分支：Dummy（凑数）／Stub（给答案）／Spy（记账）／Fake（简化但能跑）／Mock（给答案＋验交互）；右侧写三条路："搭环境（贵）／不测（欠债）／替身（对）"。

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

**预设回答：**

- 答"用替身，短信发不出去还会花钱"——对，这是"有副作用"那一类，必须隔离。

- 答"搭真环境更真实"——真实但代价高，而且**不可重复**（运营商限流、延迟抖动），会让测试时红时绿。

- 追问：那"真实性"怎么补？答案是分层——单元测试用替身保证逻辑正确，集成测试再用真环境验证"接线对不对"。**不同层测不同的事，别让单元测试承担集成测试的责任**。

**易错点 / 考点：**本节最容易考的是**六个替身词的定义与区别**，尤其 Stub 与 Mock。判断口诀先给你：**看它"有没有被验证交互"**——只提供返回值、从不检查"被调用了几次"的，是 Stub；会 `verify` 调用次数／顺序／参数的，是 Mock。

**过渡：**分类从最弱的一个开始。下一页把 Fake、Dummy、Double、Spy 四个词一次讲清。

---

## 【5.53】Fake、Dummy、Double 等（3 分钟）

这一页是定义页，四个词一个一个来，每个给你一条"什么时候用它"的判据。

**Fake，伪对象**——伪造、假货，反义词是 real。课件描述很准：**几乎和产品代码做同样的工作（伪装成跟真的一样），但为了满足测试要求，采用更简单的实现**。关键词是"更简单的实现"，不是"返回假数据"。典型例子：内存数据库代替真库、内存队列代替 Kafka。**它不是骗子，它是精简版**——行为对，性能和省事优先。

**Dummy，哑对象**——仿制品、傀儡。课件说：对被测对象的模拟，特别是**缺乏原有特性或功能而实现欺骗性地替换**，括号里补了句特别生动的话：**完全听命行事，不在具体测试里面起任何作用**。这就是 Dummy 的全部精髓——它只为"把参数位填上"。比如方法必须传一个 `Logger` 才能编译，你传一个什么都不做的空实现，那个 Logger 就是 Dummy。

**Double，测试替身**——课件定义：有相同的行为和响应。这是**总称**，就是"替身演员"：长得像、能站位、能演戏，但不是本人。Fake、Dummy、Stub、Spy、Mock 都是它的子类。

**Spy，测试间谍**——课件说：能够接收参数并返回值。这句太短，我补全：Spy 是**会记账的替身**——既像 Stub 一样返回你设定的值，又把"谁调了我、调了几次、传了什么参数"记下来供你核对。它天然适合验证副作用，比如"提交订单后审计日志接口应被调用一次，参数里必须带用户 ID"。

最后一句归类，必考：**Dummy Object、Test Spy 可归为 Stub**。因为在强弱维度上它们都还属于"提供输入

／记录输出"这一档，**都不强制你验证交互**；Mock 不一样，它天生就让你去 `verify`。

日常选用顺序，从便宜到贵：**能不用就不用 → 需要返回值用 Stub → 要看有没有被调用用 Spy → 需要"没按约定交互就失败"才用 Mock**。

> **板书：**四行定义：`Fake`＝精简版真实现（内存数据库）；`Dummy`＝凑数，不参与任何逻辑（空 Logger）；`Double`＝替身总称；`Spy`＝能给值＋会记账；底部红字"**Dummy Object、Test Spy 可归为 Stub**"；右侧一列"由便宜到贵：不用→Stub→Spy→Mock"。

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

> **预设回答：**

- 答"传个空实现，什么都不返回"——正确做法，这是 Dummy。

- 答"这是 Fake"——错了。Fake 是"能真跑的精简实现"，空实现不跑任何逻辑，是 Dummy。**判据：它内部有没有"替代性的真实计算"**。

- 追问：如果这个空实现的某个方法被误调用了、返回 null 导致下游空指针，你怎么办？——让 Dummy 在"不该被调用"的方法里直接抛异常，这条用例就变成了"验证它没被调用"，非常好用。

> **易错点 / 考点：**这一页是纯概念页，考点就是**四词定义 + "Dummy／Spy 可归为 Stub"**。三个高频错答：① 把 Fake 说成"假数据"（Fake 是假**对象**，是简化实现）；② 把 Dummy 说成"返回默认值的桩"（它不返回有意义的值，它根本不该被调用）；③ 把 Double 当成某个具体替身（它是总称）。记忆口诀：**Fake 像真的、Dummy 是摆设、Double 是统称、Spy 会记账**。

> **过渡：**四个词里最容易被误解的是 Fake——很多人以为 Fake 就是"返回假数据"，其实它可能是你整个测试策略里最有价值的一件东西。下一页看它到底怎么用。

---

## 【5.54】伪对象（2.5 分钟）

这一页讲 Fake 的适用场景。课件第一句就把定位说清楚了：**在某些情况下，插桩不够用。被桩对象替换掉的行为是被测对象所需要的。在这种情况下，伪对象（fake）可能是一种合理的折衷。**

这句话要慢慢嚼。什么叫"被桩替换掉的行为是被测对象所需要的"？意思是：被测代码不只是拿一个返回值

就完事，它需要下游**真的做出某种行为**。最常见的例子——测"下单"流程，它调仓储**保存**订单，然后**查出来**生成发票。你用 Stub 让 `save()` 返回 true，那 `find()` 呢？数据根本没存进去，你只能再写一个桩返回一个假订单——**结果你测的是"桩的剧本"，不是"仓储的行为"**。

这时候 Fake 就对了：内存仓储，`save()` 真存进一个 Map，`find()` 真从 Map 里取。行为是真的，只是没落磁盘。这就是"折衷"两个字的含义——**比桩真，比真库轻**。

课件给的例子是这段 `MakePurchase`：先 `CreatePurchase(customer)`，再 `AddProduct(purchase, product)`，再 `CreateInvoice(purchase)`，有折扣就 `ApplyDiscount(discount)`。它的特征是**一串有状态的协作调用**，后一步依赖前一步的结果。这种代码用 Stub 硬顶，你要为每一步编一个"看起来像真的"返回值，编到第三步你自己都不信。

三条判断标准，什么时候从 Stub 升级到 Fake：① 被测代码有**写后读**依赖（存了再查）；② 依赖下游的**状态机**（未支付→已支付→已发货，桩很难维护）；③ 同一批测试要**反复复用**同一个协作者。

反面也要说清：**Fake 的成本是自己实现一套简化逻辑**，而且这套逻辑本身可能有 bug。所以只用在价值高、复用多的协作者上；一次性场景，老实写 Stub。

> **板书：**左侧"Stub 不够用的信号"：写后读／依赖状态机／多用例复用；右侧"Fake = 行为真、实现简"，例：内存仓储（Map 代替 DB）；中间一句课件原话"被桩替换掉的行为是被测对象所需要的"；底部"Fake 的成本：自己实现＋自己可能有 bug"。

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

**预设回答：**

- 答"会，内存库和真库的 SQL 方言不一样"——非常关键的一点。HSQLDB 能跑的 SQL 到了 MySQL 未必能跑。

- 答"所以还要有集成测试跑真库"——正解。**单元测试用 Fake 保逻辑，集成测试用真库保 SQL**，这就是分层。

- 追问：那 Fake 到底值不值？值——因为它把 90% 的用例从"几百毫秒"压到"几毫秒"，让开发者愿意频繁跑；剩下 10% 交给集成测试兜底。

**易错点 / 考点：**常考"Fake 与 Stub 的适用差别"——标准答案：**Stub 只提供预设返回值；Fake 提供简化但真实的行为实现**，适用于被测对象依赖协作者"行为"而非仅"返回值"的场合。易错点是把 Fake 说成"造假的测试数据"，那是 fixture（测试夹具），不是 Fake。

**过渡：**光说概念还是飘，下一页那张示意图把"SUT、DOC、前门、往返测试"这四个词摆到了一起。

## 【5.55】Fake 示意图（2 分钟）

这张图是 Fake 的结构图，要你看出四件事：被测系统在哪、被依赖组件在哪、从哪个门进去、什么叫往返测试。

两个缩写先认：**SUT 是 System Under Test，被测系统**；**DOC 是 Depend-on Component，被测系统所依赖的组件**。单元测试的基本格局就是"SUT 一个、DOC 若干"，桩、假、Mock 全是围着 DOC 打转。

第一个要点是**前门（front door）**，图上标着 Public interface。意思是：**测试应该从被测对象的公开接口进去**，不要绕过接口去动内部字段。从公开接口进去，测的才是真实使用方式；一旦用反射改私有字段，测试就跟实现细节绑死，实现一改测试就红——那种测试是负担，不是资产。

第二个要点是 **Fake 替换的位置**：它站在 DOC 那个位置上，接口一样、行为简化。注意它替换的是"整个协作者"，不是"某一次调用"——这是它和 Stub 在结构上的差别：**Stub 管"某次调用返回什么"，Fake 管"这个协作者整体上是什么"**。

第三个要点是 **round-trip test，往返测试**：一次测试里既写进去又读出来，验证数据在往返中没丢没错。典型场景：保存订单→重新查询→断言字段一致。这种测试用 Fake 最自然，因为 Fake 真有存储；用 Stub 做往返，等于自己跟自己对台词。

第四个要点藏在箭头里：**SUT 只依赖接口，不依赖实现**。这是整个技术能成立的前提——如果被测代码直接 `new MySQLConnection()`，你连替换的机会都没有。所以写不下去时先回头看这条：**不是测试方法的问题，是依赖注入没做好**。

> **板书：**画图：左"前门 Public interface / front door"→ 中"SUT 被测系统"→ 右"DOC 依赖组件"，DOC 位置上标"Fake 站在这里"；下方写 "round-trip test：写进去→读出来→比对"；底部一行"SUT 只依赖接口，才换得动"。

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

**预设回答：**

- 答"没什么问题，能测就行"——短期方便，长期是**测试与实现耦合**，重构时测试必炸。

- 答"会把实现细节暴露成契约"——很准。public 一开，别人就会依赖它，你就再也改不动了。

- 追问：那私有方法里的复杂逻辑怎么测？——通过公开接口覆盖它的所有分支；如果实在测不到，说明它应该被提取成一个独立的类，然后单独测。**又是"测试逼设计"**。

> **易错点 / 考点：**这一页的考点是**缩写与概念**：SUT、DOC、front door、round-trip test 四个词要能对上中文。易错点是把 DOC 说成"被测对象"（DOC 是**被依赖**的组件）；以及把 round-trip 理解成"跑两遍"，它的意思是"写—读—比对"一个来回。

> **过渡：**示意图之后是两个具体的 Fake 例子，第一个看结构，第二个看它怎么顶替数据库。

---

## 【5.56】Fake 示例 1（2 分钟）

这个示例是 Fake 的一个具体落地，我们从"为什么这么写"看它，而不是逐行念代码。

场景：被测代码需要一个仓储协作对象，负责存和取。真实实现连数据库，测试里不想连。于是写一个 `FakeRepository`，**内部用 Map 或 List 存数据**，对外实现同一套接口——`save` 往 Map 里放，`findById` 从 Map 里取，`count` 返回 Map 大小。

三个设计要点决定它写得好不好。

**第一，接口必须和真实实现一致。** 最稳的做法是**让 Fake 和真实现实现同一个接口**，这样替换谁都不用改被测代码，编译期就能保证签名对得上。

**第二，行为要"够真"。** 什么叫够真？`save` 之后 `findById` 必须查得到（这就是往返）；同一个 ID 重复 `save` 应该覆盖而不是新增；`count` 要跟着变。这些不是额外功能，它们恰恰是**被测代码依赖的语义**。少实现一条，测试就给出错误的绿灯。

**第三，不要实现真实现不需要的东西。** 事务、缓存、分页优化——真实现有，但被测逻辑用不到，写了只是给自己埋 bug。**Fake 的目标是"语义够用"，不是"功能齐全"**。

还有一个工程习惯：**Fake 是生产资产，不是测试脚本里的临时类**。它应该放在 `src/test/java` 下独立的 `fakes` 包里，有名字、有注释，可能还有自己的单元测试。因为它会被几十上百条用例复用，一旦它有 bug，全项目的测试都在骗你。**给测试代码和产品代码同等的待遇**，这是成熟团队和初级团队最明显的差别之一。

> **板书：**三要点：① 与真实现同接口（编译期保签名）② 语义够真（存后能取・同 ID 覆盖・count 同步）③

不多实现用不到的功能；右侧一句"Fake 放 `src/test/.../fakes`，是资产不是草稿"。

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

**预设回答：**

- 答"测新增的用例"——不一定。真正假绿的是**测"更新"和"数量"的用例**：更新后 `count` 应该还是 1，但 Fake 返回 2，而你写的断言如果是"能查到"就发现不了。

- 答"所有用例都会错"——太绝对，但方向对：**依赖语义的用例都会给出错误信号**。

- 追问：所以 Fake 要不要自己也写测试？要。**给 Fake 写测试**，是本课程希望你建立的一个反直觉习惯。

**易错点 / 考点：**这一页不考代码细节，考的是"Fake 与真实现的关系"：**同接口、语义一致、实现更简单**。易错点：学生常把 Fake 写成"另一个完全不同的类，只是名字像"——那样替换时就得改被测代码，等于没做到"可替换"。

**过渡：**第一个例子讲"怎么造一个 Fake 仓储"，第二个例子更进一步：直接用现成的内存数据库当 Fake。

---

## 【5.57】Fake 示例 2（2.5 分钟）

这个例子给了一句非常凝练的定义：**fake object = enhanced stub**——伪对象就是"增强版的桩"。为什么说增强？因为它把"返回一个值"升级成了"表现出一种行为"，但目的仍然是测试，不是生产。

场景是：**验证 DAO（Data Access Object）能正常工作，就构造 fake object 代替数据库**。关键词是"验证 DAO 本身"。前面讲的 Fake 都是当配角、帮别人被测；这一页反过来——**Fake 当背景，让 DAO 成为被测对象**。DAO 的职责就是"把对象翻译成 SQL 再翻译回来"，当然要跟真数据库打交道才能验，可每次连真库太慢、还要清数据。

课件给了两个具体东西：**HSQLDB 是流行的 in-memory database**（内存数据库），**DBUnit** 是配套的数据准备框架。

这两个的组合就是经典的"DAO 单元测试方案"：用 HSQLDB 起一个内存库（进程内、毫秒级、随测试生死），用 DBUnit 在每条用例前把数据灌成一份 XML／CSV 定义的初始状态，测完自动重建。**这样你测的是真 SQL、真 JDBC、真类型映射，只是数据库是"内存版的"**。

这里有个分层观念要说清：**这已经不算纯单元测试了**，业内叫"集成度更高的单元测试"或"轻量集成测试"。

> 它比纯单元测试真（SQL 真跑），比全集成快（不连网络、不起容器），是很好的折衷点，尤其适合 DAO／Repository 这一层——这一层如果全用 Mock 把 JDBC 替掉，你测的只是"我调了 jdbcTemplate.update"，SQL 对不对一点没验。
>
> 最后提醒一个坑：**内存库和真库的 SQL 方言不完全一样**。HSQLDB 有 MySQL 兼容模式，但函数、索引、锁行为还有差异。所以这类测试**不能替代**真库上的集成测试，上线前那一轮还得在真库跑。

> **板书：**顶部写课件原话"fake object = enhanced stub"；中部写方案：HSQLDB（内存库）＋ DBUnit（数据准备）；下方三段定位："纯单元（全替身）→ **DAO 用内存库（本页）** → 全集成（真库真网）"；右下角红字"方言有差异，替代不了真库集成测试"。

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

> **预设回答：**

- 答"Mock 能验调用参数，内存库能验 SQL 结果"——准确。Mock 验"我传了正确的 SQL 和参数"，内存库验"这段 SQL 真的返回了正确的行"。

- 答"Mock 更快、内存库更真"——对，这就是取舍。

- 追问：那到底用哪个？——**看你要防什么缺陷**。防"参数传错"用 Mock；防"SQL 写错、映射错"必须真跑。DAO 这一层的缺陷 80% 在 SQL 和映射上，所以内存库更值。

> **易错点 / 考点：**常考"Fake 的典型实现方式"——答案是内存数据库、内存队列、内存文件系统这类"真实行为的简化实现"。易错点：把 DBUnit 说成"数据库"（它是**测试数据准备**工具），把 HSQLDB 说成"Mock 框架"（它是**真的数据库**，只是驻留内存）。

> **过渡：**到这里，替身家族里"提供状态"的那几个都讲完了。接下来是最容易混、也最重要的一对：Mock 和 Stub。

---

## 【5.58】模拟对象（2.5 分钟）

> 这一页是本节的思想转折点，课件用了很重的措辞：**模拟对象（Mock）在某种程度上是游戏的改变者，因为它们将测试的焦点从"状态"转换到"行为"。**

先说**基于状态的测试**。课件描述：像 fake 会聚焦于状态的测试，以断言结束，该断言会核对返回值，或以某种方式查询被测对象的状态（属性）。翻成人话：**我关心结果对不对**——算出来是 5 就是 5，库存从 100 变成 99 就是 99。这是绝大多数人写测试的默认方式。

再说**基于行为的测试**。课件说它完全不同，能够验证模拟对象和被测代码、或另一协作者之间**特定的交互行为**，会关注被测对象**是否被调用、调用了多少次、是否查询了 Value 属性**。翻成人话：**我关心你有没有按约定去打招呼**。

为什么需要第二种？因为有一大类逻辑，结果根本不体现在返回值上。比如用户下单成功后要给风控系统发一条"交易报备"。这个方法返回的只是一个订单对象，你从返回值上看不出报备发没发。状态验证无从下手；行为验证一句话：`verify(riskClient).report(any())`——**报备必须发生一次**。这就是 Mock 的战场。

课件那段代码就是这个套路：构造一个 `LenientMock`，注入 `PurchaseWorkflow`，走一遍"加商品→用已有客户→输折扣码"，最后 `campaignMock.verify()`。注意这里没有 assertEquals，它的判决完全来自"交互有没有发生"。

最后给一个平衡：**行为验证更强，也更脆**。它把"怎么调"写进了测试，一旦 `report()` 改名成 `submitRisk()`，或一次调用拆成两次，测试就红——即使业务行为完全正确。所以业内经验是：**优先状态验证，只在结果不可观测时才用行为验证**。

> **板书：**左右对照：左"状态验证（state）：断言返回值／对象属性——问『结果对不对』"；右"行为验证（behavior）：verify 调用次数・参数・顺序——问『有没有按约定交互』"；中间一行"结果不可观测时，才用行为验证"；底部写 `campaignMock.verify()` 圈出来标注"这就是断言"。

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

**预设回答：**

- 答"返回值是 void，没法验"——这是没思路的表现。void 恰恰是行为验证的典型场景。

- 答"用行为验证，verify 邮件服务被调用一次，且收件人是新用户"——标准答案，而且要注意"参数也要验"，否则发错人也算通过。

- 答"我改成让它返回发送结果，然后断言"——可以，但要问一句：**为了可测改产品接口，值不值？** 如果这个返回值业务上本来不需要，就别为测试污染设计；如果它本来就有价值（比如用于重试），那改得对。

**易错点 / 考点：**必考"状态验证 vs 行为验证"的定义与适用场景。标准表述：**状态验证检查被测对象最终的状**

**态或返回值；行为验证检查被测对象与协作者之间的交互（是否调用、次数、顺序、参数）**。易错点：把"行为验证"说成"测试行为"（两个"行为"不是一回事，前者指交互，后者指功能）。

**过渡：**概念讲完，两页纸把 Mock 和 Stub 摆在一起对比——这是本节的必考页。

---

## 【5.59】Mock vs. Stub（2 分钟）

这一页是一张对比图，把 Mock 和 Stub 摆在一起。要记住的不是两个名字，而是**三条分界线**。

**第一条，目的不同。** Stub 的目的是让测试跑起来——被测代码要一个值，它给一个值，任务完成。Mock 的目的是让测试做判断——它存在的意义就是被你 `verify`。所以有句话很精辟：**Stub 是被动应招，Mock 是主动待命**。

**第二条，验证什么。** Stub 讲究状态——你给什么输入、它返回什么输出，断言落在被测对象的返回值上。Mock 讲究行为——断言落在"这个协作者被怎么对待了"：调了几次、参数是什么、顺序对不对。同一段代码，用 Stub 你只能问"结果对不对"，用 Mock 才能问"过程守不守规矩"。

**第三条，严格程度。** Mock 默认严格：你声明了期望，它没被满足就失败——即使业务结果完全正确。这就是"过度规范"（over-specification）的风险。Stub 很宽松：它不管你调没调，反正它在那儿。

再补一条机制上的差别，读代码时很有用：**Mock 通常是动态生成的代理对象**（框架运行时造一个实现接口的类），**Stub 通常是你手写的类**。看到 `when(...).thenReturn(...)`，那是框架造的 Mock；看到 `class FakeXxx implements Xxx`，那是手写 Stub。

给你一个判断题模板，考试直接用：**这条测试挂掉，是因为结果错了，还是因为调用方式变了？** 前者说明该用 Stub；后者说明你用了 Mock，它在向你报警。**如果这个变更业务上完全无害，那这条 Mock 测试就是负资产**——它没保护质量，只保护了实现。

**板书：**三行对照表：目的（跑起来 vs 做判断）｜验证（状态 vs 行为）｜严格度（宽松 vs 严格可 over-spec）｜机制（手写类 vs 动态代理）；底部口诀"Stub 被动应招，Mock 主动待命"；右下角写判断句"结果错了用 Stub，调用变了才有 Mock 的份"。

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

**预设回答：**

- 答"Stub 的测试还是绿的，Mock 的红了"——正是这道题的标准答案，它把两条分界线一次说明白了。

- 答"两个都会红"——不对，Stub 不关心你调了几次。

- 追问：那说明什么？——说明 **Mock 测试的成本要算进重构成本里**。所以你写 Mock 的时候要克制：只对"业务上必须发生的交互"做 verify，别对"实现细节"做 verify。

**易错点 / 考点：**本节最核心的考点，几乎必出。答题模板：**Stub 提供预设返回值、用于状态验证；Mock 验证交互行为（调用次数／参数／顺序）、用于行为验证；Mock 更严格，滥用会导致测试脆弱**。易错点：说"Mock 是更高级的 Stub"——不是"更高级"，是"关注点不同"。

**过渡：**这一页是两张图，下一页是课件给的一段总结，它给出了这件事最实用的那句话。

---

## 【5.60】Stub 和 mock 小结（3 分钟）

这一页是本节的收口，四句话，一句比一句重要，最后一句是本章的名言。

第一句：**Stub 最弱，只是参与测试，为了让测试跑起来。** 它提供输入，不参与判断。这不是贬义——**弱有弱的好处，弱就是稳定**，它不会因为你改了实现就炸。

第二句：**Dummy／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，比如 Dummy server，给它不同输出会有不同响应，但内部没有实质计算。** 注意"内部没有实质计算"——这是 Dummy 和 Fake 的分界线。

第三句：**Mock 更强，要求"如何参与"，且比较灵活，视情况而定，其行为可能像 Dummy、Stub 或 Spy。**这句信息量很大：Mock 是万能替身——通过配置，今天可以是 Dummy、明天是 Stub、后天是 Spy。所以**从能力上说，Mock 包含其它**。

第四句，最重要：**够用就行，如果 Stub 够用，就不要用 Mock。**

为什么？因为**替身的强度不是免费的**。用 Mock 就要写 `verify`，就要约定调用次数和参数——这些全是**实现细节**。实现一变测试就红，红了团队就不信任测试，不信任就开始 `@Disabled`，`@Disabled` 一多，测试就名存实亡。**这是测试体系崩塌的典型路径，起点就是"随手多写了一个 verify"**。

反过来，Stub 只承诺"我给这个值"，不管你怎么用，所以它对重构**免疫**。同一场景能用 Stub 表达清楚，就

别升级成 Mock。

那什么时候必须用 Mock？三个条件满足其一：① **结果不可观测**（void 方法、发消息、写日志）；② **交互本身是业务要求**（"必须且只能报备一次"，重复报备是事故）；③ **要验证顺序或次数**（先扣库存再扣款，顺序错了就是资损）。这三条之外，都是 Stub 的活。

再深一层：这句话是一种**工程价值观**——用最简单的手段达成目的，不为炫技买单。技术选型不是"谁强用谁"，而是"谁刚好够用用谁"。

**板书：**强度阶梯：`Stub`（最弱・给值）→ `Dummy／Spy`（参与・记账）→ `Mock`（最强・可扮任意角色）；中间画一条横线写课件名言"**够用就行，Stub 够用就不要用 Mock**"；右下角三条件："结果不可观测／交互即业务要求／需验顺序次数 → 才上 Mock"。

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

**预设回答：**

- 答"Mock 性能差"——不成立，性能差别可以忽略。

- 答"下次重构你就知道了"——情绪化，不是论证。

- 答"我用一次重构实验证明"——最佳答案：拿一条用例，把实现里的方法调两次改成一次（业务等价），跑测试，红了；再换成 Stub 版本，绿了。**用证据说话，比讲道理有效十倍**。

- 收口：这正好引出"技术选型要能被验证"这个习惯——你说 Mock 好或不好，都要有实验。

**易错点 / 考点：**这一页几乎就是标准答案原文，四句话要能背：**Stub 最弱、Dummy／Spy 次之、Mock 最强且灵活、够用就行**。判断题常出"Mock 比 Stub 好，应该优先使用"——错，标准答案就是课件那句"如果 Stub 够用，就不要用 Mock"。另一个考点：Dummy 的"内部没有实质计算"。

**【思政落点 9】**：这句"够用就行"是难得的**务实精神**教科书。技术圈有个通病——**用什么工具代表水平**：写测试必须上 Mock，做架构必须上微服务，选数据库必须分布式。结果系统越来越复杂，问题越来越多。课件这句话给的立场正好相反：**以问题为中心，不以工具为中心**。请把这条带到你们以后的工作里：先问"我要解决什么"，再问"最轻的手段是什么"。**克制，是一种更高级的专业能力**。

**过渡：**到这里"该用哪种替身"想清楚了，剩下的问题是"用手写还是用框架"。下一页进入 Mock 框架。

---

## 【5.61】Mock 框架（2 分钟）

这一页给了四个名字：**JMock、EasyMock、Mockito、Moco**，括号里还注了"Java API、独立服务器"。它们不是同一类东西。

前三个是 **Java 的 Mock 框架**，解决"在测试代码里动态造替身对象"，能力边界是**同一个 JVM 进程内**：给接口生成代理、控制返回值、记录调用。

第四个 **Moco** 不一样，课件标了"独立服务器"——它是**可以独立启动的 HTTP 服务桩**，用来模拟外部 HTTP 接口。给它一份配置，它就监听端口，按匹配规则返回响应。所以 Moco 解决的是**跨进程**的替身。

这个区分考试和实战都常混：**同进程内的协作者用 Mockito；跨进程的 HTTP 依赖用 Moco（或 WireMock、MockServer）**。判据一句话：**能不能在测试方法里 `new` 出来**。能，用 Mockito；不能（要走网络），用服务桩。

再看三个 Java 框架的代际关系，帮你记顺序：**JMock → EasyMock → Mockito**。演进主线是**越来越轻**：JMock 用一套约束对象（`oneOf`、`allowing`）表达期望，读着像英语但写起来啰嗦；EasyMock 用"录制—回放—检查"三步，必须先声明期望再回放；Mockito 砍成 `when/verify` 两步，还允许先调用再验证。**这就是它胜出的原因：更符合人写测试的思路**。

实践建议：**除非维护老项目，新项目直接上 Mockito**。EasyMock／JMock 的价值主要是读懂存量代码，以及理解"为什么 Mockito 的设计更好"。

> **板书：**左边一列"进程内 Mock 框架"：JMock → EasyMock → Mockito（演进主线：越来越轻）；右边单列"Moco＝独立 HTTP 服务桩（跨进程）"；中间一行判据"能在测试里 new 出来 → Mockito；要走网络 → Moco"。

> **提问：**被测代码要调一个第三方支付接口（HTTP），你该用 Mockito 还是 Moco？为什么 Mockito 在这里不管用？

> **预设回答：**

- 答"用 Moco，因为要发 HTTP 请求"——正确，跨进程就得服务桩。

- 答"Mockito 也能 mock，mock 掉那个 HTTP 客户端类就行"——这条**部分成立**，而且很常见：把客户端封装成一个接口再 mock 它。但代价是你测不到"序列化、URL、超时、重试"这些真正容易出错的地方。

- 追问：所以正确姿势是两层都测——**逻辑层 mock 客户端，契约层用 Moco 验真实 HTTP**。分层，永远分层。

**易错点 / 考点：**常考"Mock 框架有哪些"，四个名字要能列全；同时要能说出 **Moco 与前三者的定位差别（独立服务器／HTTP 服务桩）**。易错点：把 Moco 说成"Java Mock 框架"（它也支持 Java API，但核心是独立服务器）；把 JMockit 和 JMock 混成一个（JMockit 能力更强、耦合 JDK 更紧，见 5.43）。

**过渡：**先看最早被广泛使用的那一个，EasyMock，因为它的"三步法"是理解所有 Mock 框架的钥匙。

---

## 【5.62】EasyMock（2.5 分钟）

EasyMock 的定义课件写得很准：**通过简单的方法对于给定的接口生成 Mock 对象的类库。它提供对接口的模拟，能够通过录制、回放、检查来完成测试过程，可验证方法的调用种类、次数、顺序，可令 Mock 对象返回指定的值或抛出指定异常。**

三个关键词必须抓住：**录制（record）、回放（replay）、检查（verify）**，这是它的心脏。

用录像机打比方：**录制**是先跟替身对台词——"待会儿有人问你 `execute()`，你就返回这个 ResultSet"；**回放**是把替身放进被测代码里跑，它按剧本应答；**检查**是跑完看剧本有没有演完——该调三次的调了两次，报错。

课件那段代码就是这个骨架：先 `EasyMock.createControl()` 建控制中心，再用 `control.createMock()` 造出 `Connection`、`Statement`、`ResultSet` 三个替身——JDBC 三层对象，正好是"层层依赖"的典型。

为什么这个设计是"必然的"？因为 **Mock 对象的行为必须在被测代码运行之前就定义好**——它一旦开始跑就在被动应答，没有机会改剧本。所以"先录后放"是 Mock 的**逻辑必然**，EasyMock 只是把它显式化了。

那 Mockito 为什么能"先调用后验证"？因为它把"期望"和"验证"拆开了：`when()` 定义应答（相当于录制），`verify()` 是**事后**检查，而且不预先声明就必须满足——你没 verify 的调用它不管，你 verify 了的它才查。**这是宽松模式对严格模式的胜利**。

留一个知识点：EasyMock 当年只能模拟**接口**，Mockito 能模拟类和 final 类，JMockit 连静态方法、构造器都能模拟。**"能模拟什么"这条谱系，是理解三个框架差别的坐标轴**。

**板书：**三步法大字：**录制（record）→ 回放（replay）→ 检查（verify）**；下方抄骨架：`createControl()` → `createMock(Connection/Statement/ResultSet)`；右侧一行"EasyMock 严格：先声明后回放；Mockito 宽松：when 定义＋事后 verify"。

**提问：**为什么 Mock 对象的行为必须在被测代码运行"之前"就定义好？如果不能，会发生什么？

**预设回答：**

- 答"因为运行时它已经在应答了，改不了"——到位。顺序是结构决定的，不是习惯问题。

- 答"其实可以边跑边改"——技术上部分可行（有些框架支持 Answer 回调里改），但那样测试就变成"跟被测代码赛跑"，可读性极差。

- 追问：那 Mockito 的 `verify` 为什么可以事后？——因为 `verify` 不是"设定剧本"，是"核对发生了没有"，它读的是框架记的调用日志。**一个管"演什么"，一个管"演了没"**。

**易错点 / 考点：**必考 **录制—回放—检查** 三步，以及 EasyMock "可验证方法的调用种类、次数、顺序"。易错点：把 verify 说成"验证返回值"——verify 验的是**调用**，返回值是 when/thenReturn 那段的事。

**过渡：**EasyMock 之后就是今天的主角。Mockito 的三个核心动作，下一页用一张图给你摆开。

---

## 【5.63】Mockito（2 分钟）

这一页是 Mockito 的总览图，把用法分成三个动作：**设置依赖性、Stub 方法调用、验证交互性操作**。三个动作正好对应一条测试的生命周期。

**设置依赖性**，就是"把替身造出来、塞进被测对象"。两种写法：代码式 `mock(Xxx.class)`，或注解式 `@Mock` ＋ `@InjectMocks` 自动注入。这一步解决"被测对象的协作者从哪来"。

**Stub 方法调用**，注意这个说法——它是**用 Mockito 的 API 给 mock 对象打桩**：`when(mock.getPrice()).thenReturn(100)`。容易混的点是：**"mock 对象"是角色，"给它打桩"是动作**，打桩在术语上叫 stubbing，所以**一个 Mock 对象也可以被当作 Stub 用**——这就解释了 5.60 那句"Mock 的行为可能像 Stub"。

**验证交互性操作**，就是 `verify(mock).save(any())`——检查方法有没有被调用、调了几次、参数是什么。这是 Mockito 区别于纯 Stub 工具的地方。

顺序也值得注意：**设置依赖 → 打桩 → 执行 → 验证**。打桩必须在执行之前，因为它改的是应答规则；验证在执行之后，因为它读的是调用记录。

一个工程建议：**这三个动作要能被一眼看出来**。标准做法是 Arrange 段集中 `mock`＋`when`，Act 段一行

调用，Assert 段集中 `verify` + `assert`。**结构清晰比技巧花哨重要**。

补一句定位：Mockito 是 Java 生态里**事实标准**，Spring Boot 的 `spring-boot-starter-test` 默认就带它。你不用选，只要会用。

**板书：**三动作横排：① 设置依赖性（`mock()` / `@Mock` + `@InjectMocks`）→ ② Stub 方法调用（`when(...).thenReturn(...)`，术语 stubbing）→ ③ 验证交互（`verify(...)`）；下面一行时序："设置依赖 → 打桩 → 执行 → 验证"；右侧一句"Mock 对象也能当 Stub 用"。

**提问：** `when(mock.save(any()))` 出现在被测方法调用**之后**，会发生什么？为什么？

**预设回答：**

- 答"没效果，桩打晚了"——基本对，但更准确的说法是：Mockito 会认为你在**执行一次真实调用**（默认返回 null），然后你又对这次调用打了桩，结果既没起到作用，还可能污染调用记录。

- 答"会报错"——某些情况下会（比如对未完成的 stubbing 做检测），但不保证报错，这才是危险的地方：**它可能静默失效**。

- 追问：所以怎么避免？——**把 when 全部写在 Arrange 段**，并且尽量在创建 mock 后立刻打桩，形成肌肉记忆。

**易错点 / 考点：**三个动作的名称要能对上 API（设置依赖＝mock／@Mock；Stub 调用＝when／given；验证交互＝verify）。易错点：把"设置依赖性"理解成"配置依赖注入框架"，它在这里指的是**准备并注入替身对象**。

**过渡：**三个动作展开就是 Mockito 的全部 API。下一页逐条过，这一页是本章的操作手册。

---

## 【5.64】Mockito 主要功能（3 分钟）

这一页是 Mockito 的功能清单，六条，按"你会怎么用"讲，每条给一句实战提醒。

**`mock()` / `@Mock`：创建 mock。** 课件补了重要一句：可以通过 `Answer` / `MockSettings` 指定行为方式。`Answer` 是"方法被调用时执行什么代码"，比 `thenReturn` 灵活，可以返回值依赖入参。**提醒：给 mock 起名字**，出错信息里就能看出是哪个替身。

**`when()` / `given()`：指定 mock 如何行动。** 这是同一件事的两种写法，`when` 是原生风格，`given`

是 BDD 风格。课件还说：返回不符合需求时可以自己写扩展，也就是自定义 `Answer`。

**`spy()`／`@Spy`：部分模拟，调用真实的方法，但仍可被验证。** `mock` 造的是全假替身，所有方法返回默认值；`spy` 是包着真对象的壳，**默认调真方法，你只对需要的方法打桩**。典型场景：一个类十个方法，只让其中一个返回特定值。

**`@InjectMocks`：自动注入带有 `@Spy` 或 `@Mock` 注释的字段。** 它先按类型再按名字注入，构造器、setter、字段注入都试。**注不进去是最高频的报错**，原因通常是类型对不上或用了 final 字段。

**`verify()`：检查方法是否用给定参数调用。** 可配 `times(n)`、`never()`、`atLeastOnce()`、`inOrder()`。**注意 `verify` 默认是 `times(1)`**——很多人以为是"至少一次"，于是调了两次的 bug 反而测不出来。

**参数匹配和 `@Captor`。** 匹配器有 `any()`、`eq()`、`anyString()`；`@Captor` 是参数捕获器——需要检查传进去的对象内容对不对，就捕获下来再断言。

有个坑必须说：**参数匹配器不能混用**，一个方法里要么全用匹配器，要么全用具体值，混着写会抛 `InvalidUseOfMatchersException`，这是新手第一大报错。

学习方法：**默写四条**——`mock`、`when`、`verify`、`@InjectMocks`，能覆盖 90% 日常场景；`spy` 和 `@Captor` 会用就行。

**板书：**六条竖排：① `mock()`／`@Mock`（可配 Answer／MockSettings）② `when()`／`given()` ③ `spy()`／`@Spy`（部分模拟，真方法）④ `@InjectMocks`（自动注入）⑤ `verify()`（次数・参数・顺序）⑥ 参数匹配 `any()`／`@Captor`；右下角红字"匹配器不能混用；verify 默认 times(1)"。

**提问：**一个类有十个方法，你只想让 `getDiscount()` 返回固定值，其余方法要跑真实逻辑。用 `@Mock` 还是 `@Spy`？如果用 `@Mock` 会怎样？

**预设回答：**

- 答"用 `@Spy`"——正确，这正是"部分模拟"的定义。

- 答"用 `@Mock`，给 `getDiscount()` 打桩"——那就出问题了：**其余九个方法全变成空实现**，返回 null／0，被测逻辑会莫名其妙地失败。

- 追问：那什么时候反过来该用 `@Mock`？——当被测对象**完全不该执行真实逻辑**时（比如它要连数据库），用 `@Mock` 隔离干净。**判据：真实逻辑对本条测试是"有帮助"还是"有害"**。

**易错点 / 考点：**必考六个 API 的作用，尤其 `spy()`＝部分模拟、`@InjectMocks`＝自动注入、`verify()`＝检查调用。三个高频错点：① `verify` 默认 `times(1)`；② 匹配器混用报错；③ `@Spy` 与 `@Mock` 的差别（前者调真方法）。还有一条判断题常出："Mockito 只能 mock 接口"——现在能 mock 类和 final 类（需 `mockito-inline`／新版默认支持）。

**【思政落点 10】**：`verify()` 这个 API 的名字值得琢磨——它叫"验证"，不叫"检查过了就行"。它要求你说清楚：**调了几次、参数是什么、顺序对不对**。这背后是一种**证据意识**：判断一件事有没有发生，不能靠"我觉得应该发了"，要靠可复核的记录。工程上大量事故源于"以为做了"——以为发了通知、以为清了缓存、以为关了口子。**把"以为"换成"有记录、可验证"，是责任心最具体的技术形态**。你以后写任何关键逻辑，都问自己一句："这件事怎么才能被证明发生过？"

**过渡：**清单里有一个注解被单独拎出来讲了一页，就是 `@Mock`。为什么它值得单独一页？

---

## 【5.65】Annotation Type Mock（2.5 分钟）

这一页讲 `@Mock` 这个注解的定位，课件给了四条，我把它们翻译成"为什么要有注解，而不是一路 `mock()`"。

第一条：**将一个字段标记为 Mock**——你声明一个字段，加上注解，Mockito 就帮你造好对象。

第二条：**允许速记的 Mock 创建**。对照一下就知道省了什么：不用注解是 `OrderRepository repo = mock(OrderRepository.class);`，用注解就是 `@Mock OrderRepository repo;`，省掉重复的类字面量。

第三条：**尽量减少重复的 Mock 创建代码**。一个测试类常有五六个协作者，每个都写一行 `mock(...)`，光准备段就十行，**把 Act 和 Assert 挤到看不见**；注解把噪音挪到字段声明区，让方法本体只剩业务。

第四条：**使测试类、验证错误更具可读性**——出错信息里带字段名（如 `orderRepository`），比 `mock for OrderRepository, hashCode: 1234567` 好读得多。

第五条：**自动检测 `MockedStatic` 类型的静态 Mock**。这是 mock 静态方法的能力，用注解可以把这个静态 mock 的生命周期纳入框架管理，**自动在合适的时候关闭**，不用手写 `try-with-resources`。**这是注解除了省代码之外的第二个价值：管生命周期**。

所以核心不是记四条，而是理解**注解的本质是声明式**：命令式说"我要做这几步"，声明式说"我声明这些东西是这样，剩下交给框架"。Spring 的 `@Autowired`、JUnit 的 `@Test` 都是同一思路。

最后提醒配套动作：**注解要生效，必须有初始化那一步**——`@ExtendWith(MockitoExtension.class)`，或手动`MockitoAnnotations.openMocks(this)`。**忘了这一步，`@Mock`字段就是 null**，这是新手第二大报错。

**板书：**四条：① 标记字段为 Mock ② 速记创建（vs `mock(X.class)`）③ 减少重复代码 ④ 提高可读性（报错带字段名）＋⑤ 自动管理`MockedStatic`生命周期；底部红字"注解要生效：`@ExtendWith(MockitoExtension.class)`或`openMocks(this)`，否则字段是 null"。

**提问：**为什么说注解除了"省代码"，还解决了"生命周期"问题？请用静态 Mock 举例。

**预设回答：**

- 答"因为静态 Mock 用完必须关，忘了关会影响其它测试"——对。静态方法被 mock 后是**全局生效**的，不关闭会污染整个 JVM 里的其它用例。

- 答"注解会自动开关"——准确。框架知道什么时候创建、什么时候释放，比你手写`try-finally`稳。

- 追问：这告诉我们注解的通用价值是什么？——**把"必须成对出现的操作"（开／关、建／销）交给框架管**，人最容易忘的就是收尾。

**易错点 / 考点：**常考"使用 @Mock 注解需要做什么前置动作"——答案是**启用 Mockito 扩展**（`@ExtendWith(MockitoExtension.class)`）或调用`openMocks`。易错点：以为加了`@Mock`就自动注入到被测对象——**注入是`@InjectMocks`的事**，两个注解解决两个问题，别混。

**过渡：**注解那一页提到了`given()`这种写法，它是 BDD 风格的入口。下一页讲 Mockito 怎么支持 BDD。

---

## 【5.66】支持 BDD（2.5 分钟）

这一页讲 Mockito 对 **BDD（行为驱动开发）** 的支持，核心一件事：**它提供了一套 BDD 风格的 API，让测试代码读起来像一句需求描述。**

先说 BDD 是什么。核心是用业务语言描述行为，典型句式 **Given（给定前提）— When（当发生某事）— Then（那么应该如此）**。比如"给定账户余额 100 元，当我取款 50 元，那么余额应为 50 元"。这句话业务方看得懂，代码也能照着写。

Mockito 的`BDDMockito`三个方法正好对上：**`given(...)`**对应 Given，给定替身的行为，等价于

`when(...)`；**`willReturn(...)`／`willThrow(...)`** 是它的搭档，写法 `given(mock.getPrice()).willReturn(100)`——**语序变了**，"当调用时，就返回"变成"给定调用，将返回"，后者更像英语句子；**`then(...)`** 对应 Then，`then(mock).should().save(any())`，读起来就是"那么它应该保存过"。

为什么值得单独一页？因为它是**可读性的实质提升**，而且把测试和需求的表述对齐了——方法名可以直接写 `given_balance_100_when_withdraw_50_then_balance_50`，报告生成出来能给业务方看。**测试报告变成验收证据**，在金融、医疗这类受监管行业价值很大。

但要给一个冷静提醒：**BDD 风格是语法糖，不是新能力**。它跟 `when/verify` 完全等价，不存在"用 BDD 测得更准"。所以**团队统一一种风格就行，别在一个类里混着写**——混着写的结果是读代码的人要在两套思维间切换，比只用一种更糟。

一句话收口：**BDD 的语法是为了让人读懂，不是为了更正确**。

> **板书：**三行对照：`given(...)` ≈ `when(...)`（给定前提）｜`willReturn／willThrow` ≈ `thenReturn／thenThrow`（将返回）｜`then(mock).should()` ≈ `verify(mock)`（那么应如此）；右侧写 "Given—When—Then ↔ 需求语句"；底部一句"语法糖，不新增能力；一个类只用一种风格"。

**提问：**如果团队一半人用 `when/verify`、一半人用 `given/then`，会不会出问题？出在哪？

**预设回答：**

- 答"不会，功能一样"——功能一样，但**认知成本翻倍**：读代码时要在两套语序间切换，评审时容易漏。

- 答"会有维护问题，风格不统一"——正确，这就是"约定优于配置"的意义。

- 追问：那怎么定？——看团队有没有 BDD 的落地场景（有没有业务方看测试报告）。有，就统一 BDD 风格；没有，`when/verify` 更短更直接。

> **易错点 / 考点：**考点是"Mockito 的 BDD 支持体现在哪"——答案：提供 `given／willReturn／then` 这类 BDD 风格 API（`BDDMockito`），语义与 `when／thenReturn／verify` 等价。易错点：把 BDD 说成"一种测试类型"（它是**开发/协作方法**，不是测试层级）；以及把 `then(...).should()` 说成"断言返回值"（它验证的是**交互**）。

**过渡：**单元测试的"写"讲完了，接下来要回答一个更硬的问题：**我怎么知道测够了？** 这就进入覆盖率工具。

---

## 【5.67】测试覆盖率工具（2.5 分钟）

这一页是覆盖率工具的入口，任务是把"这类工具干什么、有哪几个代表"讲清，原理下一页展开。

先说它要回答什么问题：**你的测试到底跑到了代码的哪些部分？** 它不判断对错，只做一件事——**统计执行痕迹**。所以覆盖率工具从来不提高质量，它只是**照镜子**：照出哪些代码从来没被跑过。

课件后面给了一串名字，我按语言归类。**Java：JaCoCo**，生态的标准答案，通过对编译后的 class 文件插桩统计覆盖情况；还有 **Jcov**、**CodeCover**、**CoverageMeter**，多用于特定场景。**C/C++：BullseyeCoverage**，课件特别点了它"支持分支覆盖率（Branch Coverage）的分析"；还有 **Testwell CTC++**（嵌入式很有名，能在目标板上跑）、**Parasoft C++Test**（测试＋静态分析＋覆盖率）。**Python：Coverage.py**，命令行 `coverage run` ＋ `coverage report`，简单直接。

这么多名字，选型三条：① **跟着语言走**（Java 就 JaCoCo，别折腾）；② **看能不能出标准报告格式**（Cobertura／XML／LCOV），因为 CI 平台和 SonarQube 要消费这份报告；③ **看插桩方式会不会影响运行**（改字节码、源码插桩、硬件探针，各有代价）。

最后补一个观念，本页最重要：**覆盖率必须"无侵入地"接进流程才有价值**。它不是"想起来了跑一次"，而是每次构建自动跑、报告自动上传、覆盖率掉了自动告警。**一次性测量毫无意义**，因为覆盖率是**会腐烂的指标**——今天 85%，明天加个新类掉到 70%，没人盯着，三个月后就是 40%。

> **板书：**三列按语言：Java（JaCoCo・Jcov・CodeCover・CoverageMeter）｜C/C++（BullseyeCoverage・Testwell CTC++・Parasoft C++Test）｜Python（Coverage.py）；底部两条："工具只照镜子，不提质量""覆盖率会腐烂，必须进流水线"。

**提问：**覆盖率工具能不能发现"某段代码逻辑写错了"？如果不能，它的价值到底在哪？

**预设回答：**

- 答"不能，它只看执行没执行行"——正确。它没有"对错"的概念，只有"到没到"的概念。

- 答"它的价值是告诉团队哪里还没测"——对，这是核心价值：**它把"未知"变成"已知的缺口清单"**。

- 追问：所以"覆盖率 90% 但线上出大事故"这种事正常吗？——完全正常，说明那 90% 里**断言不足**，或者**缺陷在输入设计没覆盖的边界上**。覆盖率是**下限保障**，不是质量证明。

**易错点 / 考点：**常考"覆盖率工具与语言对应"：**JaCoCo（Java）、BullseyeCoverage（C/C++，支持分支覆盖）、Testwell CTC++（C/C++）、Coverage.py（Python）、gcov／lcov（C/C++）**。易错点：把

JaCoCo 说成"测试框架"；把 BullseyeCoverage 说成"Java 工具"；以及把覆盖率工具说成"能提高覆盖率"（它只测量，提高靠写用例）。

**过渡：**下一页开始逐个看这些工具的实现机制，先看 JaCoCo 是怎么"不改源码也能统计"的。

---

## 【5.68】代码覆盖率分析工具（2 分钟）

我们先把这一节的问题摆出来。假设你写了三百个单元测试，全部绿灯，那能不能说"这段代码被测过了"？不能。因为用例通过只说明**你跑过的那几条路径**是对的，它完全没告诉你**还有多少代码从来没被执行过**。覆盖率工具就是回答这个问题的——它不判断对错，它只统计"哪些指令被走到过、哪些分支一个方向都没走"。

那这个统计怎么做？靠人眼看代码是做不到的。一次回归可能跑几千个用例，执行路径的分布只有机器能记下来。所以工具的做法是**在代码里埋探针**：程序跑到哪儿，就把哪儿标记一下，跑完把标记汇总成报告。JaCoCo 就是在**编译之后的 Java class 文件**上做这件事，而不是改你的源码——这一点很关键，下一页我展开。

工具不是只有 Java 一家。做 C/C++ 常用的一个是 **BullseyeCoverage**，它特别强调**分支覆盖率**的分析。为什么强调分支？因为 C/C++ 里一个 `if` 的两个方向、`switch` 的每个 case、`&&` 和 `||` 的短路，都可能是缺陷的藏身处，只看"这行执行过没有"远远不够。同一个生态里还有 **Testwell CTC++**、**Parasoft 的 Jtest 和 C++test**、**CoverageMeter**、**Jcov**、**CodeCover**；Python 那边是 **Coverage.py**。

你会发现一个规律：**每个语言生态都有自己的覆盖率工具，但原理是同一套**——插桩、记录、汇总、出报告。所以这一页不要背工具名，要记住那条原理链，工具名只是这条链在不同语言上的实现。

最后留一句提醒：覆盖率是**度量**，不是**目标**。当目标是它的时候，人会去写"能把数字刷上去"的用例；当它是度量的时候，人才会去问"这个分支为什么没被走到"。这句话下一页我给你讲透。

**板书：**覆盖率工具原理链：插桩 → 记录 → 汇总 → 报告；Java：JaCoCo（对编译后 class 插桩）；C/C++：BullseyeCoverage（专看分支覆盖）、Testwell CTC++；其他：Jtest／C++test、CoverageMeter、Jcov、CodeCover、Coverage.py。右下角写一句"覆盖率是度量，不是目标"。

**提问：**某团队覆盖率从 62% 涨到 91%，你能不能由此得出"质量变好了"这个结论？

**预设回答：**

- 答"能，涨了就是好事"——先不否定，但要追问一句：那 9 个百分点是**新增用例**涨的，还是**删掉了难测的代码**涨的？如果是后者，质量并没有变好，只是账变好看了。

- 答"不能，覆盖率只说明执行过，不说明断言对"——这是准确答案。覆盖率高但断言写成 `assertTrue(true)`，等于什么都没验。

- 答"要看分支覆盖涨没涨"——很好，这是内行的判断：语句覆盖涨了、分支覆盖没动，说明只是"多跑了几行直路"。

- 收口：所以看覆盖率要看三件事——**哪个维度涨的、涨在哪些类上、配套的断言有没有跟上**。

> **易错点 / 考点：** ① 最常见的错是"覆盖率高＝测试充分"，判断题里几乎每年出现，正确答案永远是"覆盖率是必要不充分条件"；② 插桩对象是**编译后的 class**，不是源码，选择题常把"修改源文件插入统计代码"当干扰项；③ 工具与语言的对应关系要记牢：BullseyeCoverage 是 **C/C++**，且主打**分支覆盖**；JaCoCo 是 Java；Coverage.py 是 Python。

> **过渡：** 原理说完了，具体到 JaCoCo，它到底给你几个维度的数字？我们一页看全。

---

## 【5.69】实现覆盖率度量：Jacoco（3 分钟）

这一页考得最细，我们拆成四块看：**给什么数字、凭什么能算、怎么装进去、能出什么报告**。

第一块，**六个分析维度**：指令、分支、行、方法、类、圈复杂度。这六个不是同级概念。**指令覆盖**最细，数的是字节码指令被执行的比例；**分支覆盖**数的是每个判定点的两个出口走了几个——`if` 只走 true 没走 false，这叫"部分覆盖"；**行覆盖**按源码行算；**方法覆盖**和**类覆盖**更粗，方法里有一条指令被执行就算覆盖。而**圈复杂度**不是覆盖率，它是**复杂度指标**，McCabe 的 `e − n + 2p`，JaCoCo 按方法算，告诉你"这个方法最少要几条用例才能把路径走全"。所以报告里**圈复杂度高、覆盖率低**的方法，就是最该补用例的地方。

第二块，**基于 Java 二进制码、无需源文件**。它统计的是 class，源码只用来展示。那怎么高亮到具体某一行？因为编译时默认生成了行号调试信息，class 里带着"这条指令对应第几行"的对照表。这也解释了一个现象：关掉调试信息编译，你就只剩方法级、类级的数字。

第三块，**on-the-fly instrumentation 的 java agent 集成**——边跑边插：JVM 启动时挂 agent，类被加载的那一刻改字节码、塞探针，全程不动磁盘上的 class。另一种叫 offline，先插桩再跑。它能集成到 OSGi、Web 容器、EJB 服务器这类基于 Java VM 的应用，兼容所有 class 版本、支持不同 JVM 语言——

Kotlin、Scala 编译出来也是 class。

第四块，**报告与数据**：报告有 HTML、CSV、XML 三种，HTML 给人看，另两种给流水线做门禁；执行数据可通过远程协议和 JMX 随时从 agent 里取；构建侧用 ANT task 或 Maven plugin 收集数据并生成报告。

**板书：**六维度竖排：指令（最细）→ 分支（判定出口）→ 行 → 方法 → 类（最粗）；旁边单独一列写"圈复杂度＝复杂度指标，不是覆盖率（按方法算，e−n+2p）"。第二行写"统计对象＝class；行级高亮靠调试信息"；第三行写"on-the-fly agent：加载时改字节码"；第四行写"报告 HTML／CSV／XML；数据 dump：远程协议・JMX；构建：ANT／Maven"。

**提问：**一个方法行覆盖 100%、分支覆盖只有 50%，问题出在哪？你应该补什么用例？

**预设回答：**

- 答"代码有 bug"——不成立，覆盖率和 bug 不是一回事；覆盖率低只说明**没测到**，不说明**测错了**。

- 答"说明有分支没走，要补让判定取反的用例"——对。典型场景是 `if (x > 0)` 只测了正数，负数那一支从没进过，异常处理分支也没进过。

- 答"把行覆盖刷到 100% 就够了"——要点出这是最危险的想法：行覆盖 100% 时，短路运算的右半边、`else` 的空分支照样可能完全没测。

- 收口：补用例的顺序是**先看分支、再看圈复杂度**——分支告诉你缺哪个方向，圈复杂度告诉你这个方法还差多少条路径。

**易错点 / 考点：**① 六个维度里**圈复杂度不属于覆盖率**，这是最爱考的陷阱，问"JaCoCo 提供哪六个维度"要全答，问"哪些是覆盖率指标"要把圈复杂度摘出去；② "基于二进制码、无需源文件"常被改成"需要源代码"来考判断；③ on-the-fly 与 offline 的区别：前者运行时插桩、不改文件，后者先插桩后运行；④ 报告格式三种要背全：HTML、CSV、XML。

>

**记忆口诀："指分行方类圈"**——前五个是覆盖率的从细到粗，最后一个是复杂度，别混进去。

**【思政落点 11】**：覆盖率最容易被用错的地方，是把它当成**交差的凭证**——数字到 80% 就收工，报告一贴，任务完成。但覆盖率真正的用处是**指路**：它告诉你哪一段代码从来没人走过，那些地方往往就是线上事故的产地。我希望大家养成一个习惯：看到红黄格子，第一反应不是"怎么把它变绿"，而是"这块逻辑为什么没人测、它出错会怎样"。**把度量当镜子而不是当奖状**，这就是工程人员该有的诚实。

> **过渡：**下一页开始看源码。别怕，我不带你逐行读代码，我们只看它"在干什么"——因为这三页合起来，就是覆盖率工具的心脏。

---

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

这是 JaCoCo agent 的入口方法 `premain`。名字已经说明了它的身份——**在 main 之前**。只要在启动参数里加 `-javaagent:jacocoagent.jar`，JVM 就会去这个 jar 的清单里找 `Premain-Class`，然后调用它的 `premain`，**在你的程序跑第一行代码之前先给它一个动手的机会**。

它只干三件事。第一，**读配置**：`new AgentOptions(options)`，把启动参数等号后面那串键值对（排除哪些包、数据文件落哪、端口多少）解析成配置对象。第二，**建通道**：`createRuntime(inst)` 造出运行时，再 `runtime.startup(...)`，这个运行时负责收数据、存数据、对外提供 dump 的口子——前面说的远程协议和 JMX 就是这儿暴露的。第三，`inst.addTransformer(new CoverageTransformer(...))`，往 JVM 注册一个**字节码转换器**。从这一刻起，**每个类被加载都要先过这道转换器**：读进来、插探针、再交给 JVM。

注意一个细节：它传了 `IExceptionLogger.SYSTEM_ERR`，意思是哪个类不兼容、分析失败，**不要把整个应用搞崩**，记到标准错误然后放行。

所以这一页记住一句话：**premain 不是插桩的地方，是装钩子的地方**。钩子装好，插桩发生在每一次类加载的瞬间。

> **板书：**画一条竖线代表 JVM 启动流程：`-javaagent` → 找 Premain-Class → **premain（本页）** → 应用 main。premain 下挂三个动作：① `AgentOptions` 读配置 ② `createRuntime`＋`startup` 建数据通道 ③ `addTransformer(CoverageTransformer)` 装钩子。旁注："插桩不在这一页，在每次类加载时"。

**提问：**如果某个第三方 jar 的字节码格式特别老、转换器分析不了，按这段代码的逻辑，会发生什么？

**预设回答：**

- 答"整个应用启动失败"——不会，因为转换器传了异常记录器，异常被记到 stderr，类原样放行。

- 答"那个类不被统计，其他类照常统计"——对，这是设计意图：**覆盖率工具不能成为应用的故障源**。

- 追问：这体现了一个什么工程原则？答"工具的可降级性"——监控、埋点、日志这类旁路设施，都应该是"出问题不影响主流程"。

**易错点 / 考点：**① `premain` 是 **agent 入口**，不是插桩实现，问"JaCoCo 在哪一步改字节码"要答"类加载时的 Transformer"；② 启动参数形式是 `-javaagent:xxx.jar=key=value`，考试常把 `-agentlib`、`-Xbootclasspath` 混进来当干扰项；③ `addTransformer` 注册后对**之后**加载的类生效，启动前已经在 JVM 里的核心类不在其列。

**过渡：**钩子装好了，那"插"这个动作具体怎么写？下一页是 ASM 的活。

---

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

这一页是 `instrument` 方法：输入一段字节码，输出插好探针的字节码。把它当成一条**字节码加工流水线**：进门是原始 class，出门是带探针的 class。

第一步 **读**：`ClassReader` 把 class 字节流解析成可遍历的结构。第二步 **备写出器**：`new ClassWriter(reader, 0)`，这里重写了 `getCommonSuperClass` 直接抛异常——为什么？因为 ASM 判断两个类型的父子关系时默认会**去加载这两个类**，而我们正处在"类加载"这个环节里，再去触发类加载，轻则死锁重则栈溢出。JaCoCo 的假设是：我只插探针、不做类型变换，栈帧的公共父类不会变，所以这个能力根本不需要。抛异常是一种"声明式拒绝"。

第三步 **决定探针存哪**：`ProbeArrayStrategyFactory.createFor(...)`，探针要有一个数组存"走没走过"，放静态字段、每次新建还是外部注入，得看类的形态，工厂替你选。第四步 **串访问者链**：`ClassProbesAdapter` 套 `ClassInstrumenter`，再套 writer——外层管这个类有几个探针，中层管探针怎么落进字节码，最里层管写出去。

第五步 `reader.accept(visitor, ClassReader.EXPAND_FRAMES)` 开始遍历，`EXPAND_FRAMES` 把栈映射帧展开，方便精确插桩；最后 `toByteArray()` 拿到新字节码。

一句话：它自己实现了"编译器的后半段"——不生成源码，直接在字节码上做等价变换，还要保证结果能过 JVM 校验。

**板书：**流水线框图：`class 字节 → ClassReader →（ClassProbesAdapter → ClassInstrumenter → ClassWriter）→ 带探针的 class`。旁注三点：① 重写 `getCommonSuperClass` 抛异常＝拒绝触发类加载；② `ProbeArrayStrategy` 决定探针数组存哪；③ `EXPAND_FRAMES` 展开栈帧便于插桩。

**提问：**为什么插桩工具要这么小心地"不加载别的类"？你在别的地方见过同类的约束吗？

**预设回答：**

- 答"怕慢"——不完全是，慢是次要的；主要是**类加载本身有锁**，在加载过程中再去加载，很容易死锁。

- 答"类加载器隔离，可能加载错版本"——很到位。同一个类名在不同 ClassLoader 下是两个类，工具贸然加载会拿到错误的类型信息。

- 追问：还有哪儿有类似约束？答：Java agent、热部署、APM 探针、反射框架——**凡是工作在"类加载期"的代码，都要遵守这条纪律**。

**易错点 / 考点：**① 这一段用的库是 **ASM**（字节码操作框架），考试常问"JaCoCo 用什么改字节码"；② `ClassWriter` 构造参数里的 `reader` 是为了复用常量池、减小输出体积，不是"读源码"；③ `EXPAND_FRAMES` 是遍历选项，不是插桩策略，别混。

**过渡：**类这一层过去了，方法这一层怎么插？下一页是最后一段，也是最见功力的一段。

---

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

这一页是 `visitMethod`，访问者走到"一个方法"时的回调。它的任务不是马上插，而是**先包一层壳，等方法全部走完再动手**。

先看结构：问下层"这个方法你接不接"，不接就用空的探测器顶上；接了则返回一个 `MethodSanitizer` 的匿名子类——**名字就是信息：清理器**，有些编译器生成的字节码不规范，得先规整再插，否则过不了校验。

关键在 `visitEnd`，到这一刻才掌握方法的完整结构，于是按序做四件事。一，`markLabels` **标出所有跳转标签**，搞清哪条指令跳到哪，才能找到控制流的**汇合点**——探针必须挂在分支真正分开又真正汇合的位置，挂别处要么重复计数要么漏计。二，`MethodProbesAdapter` 把探针和标签对应起来。三，`trackFrames` 为真时套 `AnalyzerAdapter`，**模拟操作数栈和局部变量表的变化**：你插了指令，栈深度就变了，不重算栈帧，JVM 校验直接给你 `VerifyError`。这是插桩最硬的门槛——**你改的不是文本，是一个必须严格自洽的栈式程序**。四，`methodProbes.accept(...)` 真正注入。

三页连起来：**premain 装钩子 → instrument 改类 → visitMethod 改方法**。回头看那六个维度就有底了：指令和行来自探针位置，分支来自标签配对，圈复杂度来自控制流图的静态计算——**全是这次字节码遍历的副产品**。

**板书：**`visitMethod` → 返回 `MethodSanitizer` 壳 → `visitEnd` 四步：①

> `LabelFlowAnalyzer.markLabels` 找汇合点 ② `MethodProbesAdapter` 绑探针 ③ `AnalyzerAdapter` 重算栈帧（防 VerifyError） ④ `methodProbes.accept` 注入。顶部写一条总线："premain 装钩子 → instrument 改类 → visitMethod 改方法"。

> **提问：**为什么探针一定要插在"控制流汇合点"，插在每条指令前面不行吗？

> **预设回答：**

- 答"每条都插更准"——方向对，但代价是**体积和性能爆炸**：一个类可能被插进成千上万个探针，测试跑起来慢到不可用，而且分支计数会重复。

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

- 追问：这说明了工具设计的什么取舍？答：**度量精度与运行开销之间的取舍**，这也是所有埋点系统的共同命题。

> **易错点 / 考点：**① 三个方法的分工要能对上号：`premain`＝入口、`instrument`＝类级、`visitMethod`＝方法级；② `AnalyzerAdapter`／栈映射帧是**防校验失败**的，不是"分析业务逻辑"；③ 这三页属于"原理了解"层级，考法以**填空、排序、简答**为主，不会让你写 ASM 代码；④ 常见错误说法："JaCoCo 通过修改源文件插入统计代码"——从头到尾它碰的都是 class。

> **过渡：**理论讲完，我们看一眼真实的报告长什么样——这一页是整节最直观的一页。

---

## 【5.73】示例（2 分钟）

> 这是一张 JaCoCo 报告的实图。我不要你记界面，我要你学会**三秒钟读出一份覆盖率报告**。
>
> 第一眼，看**颜色**。绿色是"完全覆盖"，所有分支都走到了；黄色是"部分覆盖"，最常见的情形是一个判定只走了一个方向；红色是"完全没执行"。注意——**真正值钱的是黄色**。红色一眼就知道没测，反而好处理；黄色最容易被忽略，因为它看起来"测过了"，实际上有一半逻辑从没被验证过。
>
> 第二眼，看**层级**。报告是按包 → 类 → 方法 → 行逐层下钻的。上面几层给的是汇总数字，用来定位"哪个包最差"；点进去到方法层，才能看到"哪个方法最差"；再点到行，才知道"哪一行是黄的"。**自上而下找问题，自下而上补用例**——不要一上来就翻源码找红线，那样效率最低。
>
> 第三眼，看**列**。每一层都有 Missed 和 Covered 两列，按六个维度分栏：指令、分支、行、方法、类、复杂

度。判断一个模块该不该补测试，我的经验是看两个组合：**分支覆盖率低、圈复杂度高**——这两个同时出现，说明这块代码逻辑分支多、还没测到，是风险最高区。

还有一个技巧：报告里可以按"覆盖率升序"排。排完排在最前的那几个类，就是这个项目最该补测试的地方。这比"从头到尾看一遍"高效一个数量级。

顺带说一句，为什么这张图能高亮到具体某一行？因为 class 里带着行号调试信息——这就印证了前面那句"基于二进制码，无需源文件"：源码是用来**展示**的，统计口径始终在字节码上。

> **板书：**写三行读图法：① 看颜色（绿＝全覆盖／黄＝部分覆盖＝重点／红＝未执行）② 看层级（包→类→方法→行，自上而下定位、自下而上补用例）③ 看列（Missed／Covered × 六维度）。右下角写判断公式：**分支覆盖率低 ＋ 圈复杂度高 ＝ 最该补测试**。

> **提问：**一份报告里红格子很多，但都是 getter/setter 和自动生成的代码；黄格子只有三个，都在核心业务方法里。你先处理哪边？

> **预设回答：**

- 答"先补红的，数量多"——数字思维，但方向错了：getter/setter 补上去对质量几乎没贡献。

- 答"先看黄的，因为它在核心逻辑里"——对。而且更专业的做法是：**把自动生成的代码从统计里排除**（agent 的 excludes 配置），让报告只反映人写的、有业务含义的代码。

- 追问：这说明覆盖率统计还有一个前置工作是什么？答：**划定统计范围**。范围不划，数字就没有可比性。

> **易错点 / 考点：**① 黄＝部分覆盖（分支只走一边），不是"跑得慢"；② 覆盖率报告的读法是"自上而下定位、自下而上补"，考试若问"如何用覆盖率报告指导补测"，要答出这两个方向；③ 案例题常给一张报告截图问"下一步该做什么"，标准答法是"定位分支未覆盖的方法＋排除自动生成代码＋补判定取反的用例"。

> **过渡：**报告看得懂还不够，工具得能顺手用起来。Eclipse 里那个一键跑覆盖率的插件，就是下一頁。

---

## 【5.74】代码覆盖率度量（2 分钟）

> 这一页讲的是 **EclEmma**，一句话定位：**它是基于 JaCoCo 的 Eclipse 插件**。注意这个关系——JaCoCo 是**引擎**，EclEmma 是**外壳**。引擎负责插桩、统计、算数；外壳负责让你在 IDE 里点一下就能跑、跑完直接在源码上染色。

它在开发者的日常里解决什么问题？解决**反馈距离**的问题。你在写一个方法，写完随手点一下"以覆盖率方式运行"，绿黄红立刻出现在编辑器左边那条竖条上，哪个分支没走，一眼就看到。这个反馈只要几秒钟。而如果只能等流水线出报告，反馈要几十分钟——**反馈延迟越长，人越不愿意改**，这是工程心理学，不是技术问题。

但要注意它的局限，这也是考点。第一，**它只覆盖你本机跑的那一次**，别人写的用例、流水线跑的集成测试都不在内；第二，**它依赖 IDE**，CI 服务器上没有 Eclipse；第三，**它容易被当成"我测过了"的证据**。所以企业的正确做法是**两条腿走路**：本地用 EclEmma 做即时反馈，流水线上用 JaCoCo 的 Maven／Gradle 插件出正式报告、做质量门禁。一个是"照镜子"，一个是"过秤"。

我再补一个工程细节：EclEmma 装好之后，Eclipse 里会多出一个"覆盖率"视图，你可以对**整个项目**跑，也可以对**单个测试类**跑。建议养成对"当前正在改的那个类"跑的习惯——范围越小，反馈越快，改起来越有针对性。

那这一节到此收口：**工具能告诉你哪儿没测，但不能替你决定"测得够不够"**。标准要人来定，工具只负责把事实摆出来。

> **板书：**中间写 `JaCoCo（引擎：插桩・统计・算数）`，外面套一个框写 `EclEmma（外壳：IDE 集成・源码染色）`。下面两行对比：本地 EclEmma＝即时反馈／范围仅本次；流水线 JaCoCo 插件＝正式报告／质量门禁。旁边一句："工具摆事实，人定标准。"

**提问：**既然流水线已经有 JaCoCo 报告了，开发还有必要在本地装 EclEmma 吗？

**预设回答：**

- 答"没必要，重复了"——忽略了反馈延迟这个变量。等流水线报告出来，人已经切到下一个任务了，改的意愿会大幅下降。

- 答"有必要，本地改起来快"——对，但要补一句：本地看到的数字**不能拿去汇报**，因为它只反映你这一台机器上跑过的东西。

- 追问：所以两个工具的定位差别是什么？答：**EclEmma 是开发的自检工具，JaCoCo 流水线报告是团队的验收凭证**。同一个引擎，两种用途。

**易错点 / 考点：**① EclEmma 与 JaCoCo 的关系必考：**EclEmma 是基于 JaCoCo 的 Eclipse 插件**，别答成"两个独立工具"或"JaCoCo 是 EclEmma 的插件"（方向反了）；② EclEmma 历史上曾基于 EMMA，现在用 JaCoCo 引擎——考题若出现"EclEmma 基于 EMMA"要判错；③ 常考判断："IDE 里的覆盖率等于项目的覆

盖率"——错，范围不同。

> **过渡：**覆盖率这一节到这儿收住了。从这里开始，我们换一个更大的问题：**一个个方法都测过了，把它们拼起来还能不能跑？**

---

## 【5.75】5.3 持续集成测试（1 分钟）

我们进入本章的第三块——**持续集成测试**。先把口径说清楚：教材里"系统集成的模式与方法"和"持续集成及其测试"是两节，课件把它们合成了一节叫"持续集成测试"，**内容没少，只是编排合并了**，答题时按教材口径写，听课时按课件顺序走。

这一块要解决什么问题？前面单元测试把每个函数钉死了，可系统出问题，十次有八次不是某个函数算错了，而是**两个都对的东西拼在一起不对**：接口参数顺序对不上、时间格式一个是本地时间一个是 UTC、A 服务以为 B 会重试、B 以为 A 只发一次。这些错误在单元测试里**永远测不出来**，因为单元测试把对方都 Mock 掉了。

这一块分三层递进。**第一层是单体架构**：所有代码在一个进程里，问题是"按什么顺序把模块拼起来测"，答案是三种策略——自顶向下、自底向上、混合。**第二层是微服务架构**：模块变成了独立部署的服务，拼装变成了跨进程调用，问题升级成"我根本没法把别人的服务随时拉起来"，答案就是**契约测试**。**第三层是持续集成与 CI/CD 流水线**：拼装不再是人工动作，而是每次提交都自动发生，这时候测试必须**又快又准又有序**，这就是持续测试。

还有一个学习上的提醒。这一块的内容看起来"软"——没有公式、没有算法，全是策略和做法，于是很容易被当成"听听就行"。但恰恰是这块内容最像真实工作：你以后进项目，八成会先被安排去"把这两块拼起来测通"，而不是先让你设计算法。**集成测试是新人最快接触到全局的地方**，因为拼装的时候，你必须同时理解两边的接口。

请你带着一个问题听这一块：**你们项目上一次联调，卡了几天？卡在谁等谁？**

> **板书：**中间写"5.3 持续集成测试"，下面三行：① 单体：拼装顺序（自顶向下／自底向上／混合）② 微服务：跨进程（契约测试 CDC）③ 流水线：自动拼装（持续测试）。右上角小字备注："教材 5.7／5.8 两节，课件合并为 5.3"。

**提问：**一个函数单元测试 100% 通过的系统，集成的时候最可能先在哪一类问题上翻车？

> **预设回答：**

- 答"接口对不上"——最常见的一类，参数名、类型、空值约定、编码格式，任何一处不一致都会炸。

- 答"环境问题"——也很真实：配置项、依赖版本、数据库字符集，这些在单元测试里全被 Mock 掉了。

- 答"并发和时序"——最深的一类，单线程测不出，两个服务一并行就暴露。

- 收口：这三类问题的共同点是"**它们都发生在边界上**"，而集成测试的全部工作，就是**盯着边界看**。

**易错点 / 考点：** ① 口径陷阱：课件叫"持续集成测试"，教材分"系统集成的模式与方法"和"持续集成及其测试"两节，答题看题干用哪个词；② 集成测试的对象是**接口与交互**，不是"把所有单元测试再跑一遍"；③ 集成测试**不是**系统测试，它不验证业务需求，只验证"拼起来对不对"。

**过渡：** 先看最小的场景——所有代码都在一个进程里的单体架构，按什么顺序拼？

---

## 【5.76】持续集成测试内容（2 分钟）

这一页是这一块的目录，四条：**单体架构的集成测试、微服务架构的集成测试、持续集成及其测试、CI/CD 流水线**。我按"由小到大、由静到动"给你串一遍。

前两条是**架构维度**：单体怎么拼、微服务怎么拼。它们的差别不在技术难度，而在**你能不能控制对方的运行**。单体里所有模块都在你手里，你想先测谁就先测谁；微服务里对方是另一个团队、另一个发布节奏、另一套数据库，你连"把它启动起来"都做不到。所以单体的答案是"策略"，微服务的答案是"契约"。

后两条是**节奏维度**：持续集成讲"多久拼一次"，CI/CD 流水线讲"拼完怎么送到生产"。这两条回答的是同一个问题——**当拼装从"项目末期干一次"变成"每天干几十次"，测试要怎么变**。

为什么集成测试值得单开一节？因为在测试的投入结构里，**它是最容易被压缩、又最贵的一段**。单元测试写起来快、跑起来快，大家愿意写；系统测试有明确的验收标准，不敢省；只有集成测试卡在中间——不写，出了问题查不出来；写全，环境、数据、依赖全是成本。所以这一节的核心不是"要不要做集成测试"，而是"**怎么用最少的成本，把边界上的风险盖住**"。

再补一句学习方法：这四条不是四个并列的知识点，而是**一条递进的线**。你听的时候不要一条一条记，而要问自己"如果架构从单体变成微服务，上一条的答案哪里失效了？如果节奏从每月一发布变成每天十发布，上一条的成本哪里撑不住了？"——**失效的地方，就是下一条要解决的问题**。这样记，四条就变成一条逻辑，不容易忘。

这四条有个共同的关键词：**边界**。你听后面的内容时，只要盯住"这里的边界是什么、谁在边界两边、边界上传递什么"，这一节就不会乱。

**板书：**四行目录：① 单体架构集成测试（策略：自顶向下／自底向上／混合）② 微服务架构集成测试（外部通信・数据库访问・契约 CDC）③ 持续集成及其测试（节奏）④ CI/CD 流水线（交付）。左侧竖写一条主线："架构维度（怎么拼）→ 节奏维度（多久拼一次）"，右侧竖写"边界"。

**提问：**为什么"微服务架构的集成测试"要单独成一条，而不能用单体的三种策略套上去？

**预设回答：**

- 答"因为服务是分开部署的"——这是表层原因，继续追问：分开部署带来什么后果？答：**你无法控制对方何时可用**，Stub 和 Driver 那套"我编个假的顶上"在跨团队、跨语言时成本极高。

- 答"因为技术栈可能不同"——也对，一个 Java 服务调一个 Go 服务，你的 Stub 要按人家的协议造，改一次协议你就要改一次假服务。

- 追问：所以微服务需要的不是"更好的假模块"，而是什么？答：**一份双方都认的、可自动验证的约定**——这就是契约测试。

**易错点 / 考点：**① 四条内容要能默写，常考填空；② 别把"持续集成"当成一种测试类型，它是**实践／节奏**；③ 别把 CI/CD 流水线当成"部署工具"，它在这一节里是**测试的执行载体**。

**过渡：**从最小的场景开始——单体架构，先测谁？

---

## 【5.77】单体架构的集成测试（3 分钟）

单体架构下，所有模块在一个进程里，问题就变成一道很朴素的题：**先拼哪两个模块、按什么顺序拼**。课件给了两种基本策略，我们把它们对照着看。

**第一种，自顶向下**。从最上层的**主控模块**开始，往下逐层拼。问题是：主控模块要调用下层，可下层还没拼进来，怎么办？**造一个假的顶上**——这个假模块叫 **Stub（桩）**。桩不干实事，只按约定返回一个固定结果，比如"查询用户"的桩永远返回一个写死的用户对象。这样你就能在**下层一行代码都没写**的时候，先把上层的控制逻辑跑起来。

它的好处很明确：**主控逻辑、业务流程、界面导航这些"骨架"能在最早的时候被验证**，而这类逻辑一旦错，

返工最大。坏处也很明确：**底层的关键模块被放到最后才测**——数据库访问、算法、第三方对接，这些最容易出问题的地方反而最晚验证；更要命的是，桩返回的是假数据，上层"跑通了"可能只是**假通过**，真实数据一进来就崩。

**第二种，自底向上**。反着来：从最底层开始，一层层往上拼。这次的问题变成——底层模块写好了，**谁来调用它**？没有上层，你得自己写一个调用者，这个临时写出来的调用者叫 **Driver（驱动模块）**，它负责准备输入、调用被测模块、打印结果。好处是**底层核心功能被最充分地测过**，算法、数据访问这些硬骨头先啃掉；坏处是**主控逻辑、整体流程要到很晚才能验证**，而且**驱动模块要写很多**，这些代码是测试脚手架，不进产品，但工作量实打实。

把两种放一起，你就看出这是一道**对称的取舍题**：自顶向下缺下层，所以要 **Stub**；自底向上缺上层，所以要 **Driver**。这句对应关系是本节最高频的考点，务必刻进脑子：**自顶向下需要桩，自底向上需要驱动**。

还有一个判断口诀给你：**想早看流程，选自顶向下；想早验核心，选自底向上**。如果项目是"界面和流程先给客户看"，用前者；如果是"算法准确性决定成败"，用后者。

**板书：**左右对照两栏。左栏"自顶向下"：主控 → 逐层向下 → **缺下层 ⇒ 用 Stub（桩）** → 优点：骨架早验证；缺点：底层晚测、假通过风险。右栏"自底向上"：底层 → 逐层向上 → **缺上层 ⇒ 用 Driver（驱动）** → 优点：核心早验证；缺点：流程晚验证、驱动工作量大。中间一行大字："上缺驱动、下缺桩"。

**提问：**一个项目的核心风险是"利息计算算法必须绝对准确"，界面下个月才定稿。你选自顶向下还是自底向上？为什么？

**预设回答：**

- 答"自顶向下，因为要早做演示"——需求里没提演示，这个理由不成立；而且算法在底层，自顶向下会把它排到最后。

- 答"自底向上，先把算法啃下来"——正确。判据是"**风险最大的模块在哪一层**"，在底层就自底向上。

- 答"两个都要"——这其实就是下一页的混合策略，可以顺着接一句："你说到点子上了，下一页正是它。"

- 收口：选策略的判据只有一条——**哪一层的风险最大，就先到那一层去**。

**易错点 / 考点：**① **Stub 与 Driver 的对应关系**是必考点，选择题、填空题、简答题都出过：**自顶向下→Stub（模拟被调用的下层）**，**自底向上→Driver（模拟调用它的上层）**；② 常见错误：把 Stub 说成"模拟上层"——记住桩是**被调用方**，驱动是**调用方**；③ 优缺点要能各说两条；④ 陷阱题："自顶向下不需要写任何

辅助代码"——错，桩也是要写的，只是比驱动轻。

**过渡：**既然两种各有一半好处，那能不能两头一起上？能——这就是混合策略。

---

## 【5.78】单体架构的集成测试 –混合策略（2 分钟）

混合策略，也叫**三明治策略**。做法很直白：**上层用自顶向下，下层用自底向上，两边往中间推进，最后在中间层汇合**。

为什么这样能省事？因为两种策略的辅助代码是可以省下来的。自顶向下越往下走，桩要造得越多、越复杂；自底向上越往上走，驱动要写得越多。而一旦分成两半，**上层只需要造到某个深度，下层只需要写到某个高度**，中间那一段直接对接——**桩和驱动的总量都下来了**。这是混合策略最实在的收益：省的不是思路，是**脚手架的工作量**。

第二个好处是**风险优先**。分层的时候不是对半切，而是按"哪一层最关键"来切：把**核心模块所在的那一层**作为优先测试的起点，让它尽早在真实代码（而不是桩）上被验证。这就是常说的"关键模块优先"。

第三个好处是**能并行**。上层一个小组、下层一个小组，两边同时推进，最后在中间对接口。对进度紧的项目，这是实打实的收益。

但它有代价，也要说清。第一，**分层要先想清楚**——哪些层归上、哪些层归下、中间那一层谁负责，这个划分如果拍脑袋，最后会在中间打成一团。第二，**中间层压力最大**——它是两边的汇合点，两边的假设都在这儿对撞，所以中间层的接口定义必须**最早定、最稳**。第三，管理成本高一些，两套推进节奏要协调。

最后必须点一句反面教材：还有一种做法叫**一次性集成**，也叫"大爆炸"，就是所有模块写完，一次性拼起来跑。它的问题不是"一定失败"，而是**失败了不知道怎么定位**——几十个模块同时引入，报一个错出来，你连是哪个模块的错都判断不了。所以它只在极小系统里勉强能用，工程上一般不推荐。

**板书：**画一个三层方块图：上层标"自顶向下（Stub 逐渐减少）"，下层标"自底向上（Driver 逐渐减少）"，中间画一条粗线标"**汇合层：接口最早定、最稳**"。右侧三条收益：省脚手架／风险优先／可并行；左侧两条代价：分层要先想清／中间层压力大。最下面一行写"一次性集成（大爆炸）＝不推荐：错了定位不了"。

**提问：**混合策略里，"中间汇合层"应该选在系统的哪一层？选错了会怎样？

**预设回答：**

- 答"选在正中间，上下各一半"——形式上对，但工程上更该按"**接口最稳定、变更最少**"的那一层来切，而不是按层数平分。

- 答"选在业务逻辑层"——常见的合理答案：因为业务层接口相对稳定，上下两边的数据访问和界面都能各自推进。

- 追问：选错的后果是什么？答：汇合层如果是个**天天变的接口**，两边刚对上就又变了，等于白干——所以混合策略的前提是**架构分层清晰、接口相对稳定**。

> **易错点 / 考点：** ① 混合策略又叫**三明治策略**，别名要会；② 它的三个收益（省辅助代码、风险优先、可并行）与两个代价要能成对答出；③ 常考对比题："三种策略怎么选"——答"看风险在哪层、看接口稳不稳、看进度要不要并行"；④ **一次性集成（大爆炸）** 一般作为反面选项出现，判"推荐"即错。

> **过渡：** 单体的问题解决了，可现在的系统很少是单体。服务拆开以后，集成测试的难度不是一个量级——下一页我们看微服务。

---

## 【5.79】微服务架构（2 分钟）

先建立一个基本认识：**微服务不是"把单体拆小"，而是把"部署单元"和"团队边界"对齐**。一个服务对应一个业务能力，自己一个进程、自己一个数据库、自己一条发布流水线。听起来很美，可对测试来说，这是一次**结构性升级**。

升级在哪？在单体里，模块之间是**方法调用**——传的是对象，编译器帮你检查类型，同一个进程，出错了堆栈一路到底。在微服务里，模块之间是**网络调用**——传的是字节流，序列化成 JSON 或 Protobuf，跨进程、跨机器、可能跨语言。**方法调用变成网络调用，出错的方式一下多出好几个量级**：网络会超时、会重试、会乱序；对方会重启、会限流、会版本不一致；序列化会因为一个字段可空性不同而在半夜炸掉。

这带来一个很现实的测试困境：单元测试里，你把依赖都 Mock 掉，测得很舒服，可 **Mock 掉的东西正是最容易出问题的地方**；想端到端测，你得把 A、B、C 三个服务、两个数据库、一个消息队列都拉起来，还得保证它们是"配套的版本"——这个成本，很多团队根本承担不起，于是**联调变成了项目末期的一次大型赌博**。

所以微服务的测试策略要变：**测试金字塔不再往上堆，而是往两边分**——单元测试守住每个服务内部，**接口与契约测试守住服务之间的边界**，端到端只留极少数最关键的场景做冒烟。这一页先建立这个判断，下两页讲具体怎么做：先看"通信与数据"这两类集成测试，再看契约测试。

记住一句话：**微服务的测试难点不在"测得多深"，而在"边界怎么守住"**。

> **板书：**左右对比。左"单体：方法调用"——同进程、对象传参、编译期检查、堆栈连贯；右"微服务：网络调用"——跨进程、序列化、超时重试、版本漂移、部分失败。下面画一个变形的金字塔：底层单元测试（守服务内）→ 中层**契约／接口测试（守边界）**→ 顶层极少量端到端（冒烟）。右下角写"难点不在测多深，在守边界"。

**提问：**单体应用里，"调用一个方法失败"和微服务里"调用一个接口失败"，最大的区别是什么？

**预设回答：**

- 答"微服务会超时"——是其中一种，但更本质的是**部分失败**：单体里一个方法炸了，整个请求就失败了，逻辑很干脆；微服务里 A 成功了、B 超时了、C 其实成功了只是响应丢了，系统处在一个**不确定的中间状态**。

- 答"网络不可靠"——对，而且不可靠会连带出重试、幂等、补偿这一整套设计，这些都是**测试要覆盖的新场景**。

- 追问：那单元测试能不能覆盖这些？答：不能，因为单元测试里网络根本不存在——**只有集成测试能碰到这类问题**。

**易错点 / 考点：**① 微服务集成测试的对象是**通信与数据边界**，不是"每个服务内部的业务逻辑"；② 常考判断："微服务架构下应该加强端到端测试"——错，主流做法是**减少端到端、加强契约与接口测试**；③ "部分失败"是微服务测试的核心概念，答题时能写出这个词，往往就是得分点。

**过渡：**边界有两个面：一个是"调用通不通"，一个是"数据存得对不对"。我们分别看。

---

## 【5.80】微服务架构的集成测试（3 分钟）

这一页给出微服务集成测试的两个主攻方向，课件写得很克制，但每句话背后都有血。

**第一个方向：测试外部的通信，确认通信是否通畅**。注意它的定位——**检查核心功能即可**，它不是让你把业务逻辑再测一遍，而是专盯**协议层次的错误**。课件列了三类典型：**丢失 HTTP 报头、SSL 使用错误、请求／响应不匹配**。

这三类为什么值得单列？因为它们有一个共同特点：**在单元测试里绝对测不出来，在业务测试里又容易被业务错误掩盖**。丢报头，最常见的是认证头、内容类型头、追踪头，开发本地用 Postman 手测时手动带着，一进代码就忘了加；SSL 用错，是证书校验被关掉、或者双向认证只配了一半，本地 HTTP 跑得好好的，一上

HTTPS 全线 500；请求／响应不匹配，是字段名大小写、时间格式、枚举值、可空性——最容易发生在"两个团队各自按自己的理解实现了同一个接口"。

**第二个方向：数据库访问测试**。它的目标也有三条：**确保微服务使用的数据结构与数据库相符、检查 ORM 中设置的映射关系、数据库访问模块能否妥善处理网络出错**。ORM 就是对象关系映射，把表和对象对起来。这三条其实是在说：**代码里的对象模型和数据库里的表结构之间，有一层翻译，翻译错了不会报编译错误，只会在运行时给出错数据**。

举个最容易踩的例子：代码里一个字段是可空的，数据库里那一列却是 NOT NULL；或者枚举在代码里是字符串，在库里存的是整数，某一方加了新枚举值，另一方不认识。这类问题单元测试永远发现不了，因为它用的是内存里的假数据；只有真的连上库、真的走一次 ORM 映射，才会暴露。第三条"网络出错的处理"更微妙——数据库连不上时，你的服务是快速失败、还是把请求挂在那儿把连接池耗光，这属于**韧性**问题，必须靠集成测试来验证。

把两个方向合起来看，你会发现它们的共同逻辑：**集成测试盯的是"翻译层"**——协议翻译（对象 → 字节流）和数据翻译（对象 → 表）。业务逻辑对不对，单元测试管；翻译对不对，集成测试管。

> **板书：**两栏。左"外部通信"：目标＝通信是否通畅、只查核心功能；典型错误＝**丢失 HTTP 报头／SSL 使用错误／请求-响应不匹配**。右"数据库访问"：目标＝数据结构与库相符、**ORM 映射**正确、网络出错能妥善处理。下方一句总结："集成测试盯**翻译层**：协议翻译 + 数据翻译"。

**提问：**一个服务在开发环境一切正常，一上测试环境所有请求都 401。你第一个怀疑什么？

**预设回答：**

- 答"账号密码错了"——先排除，但环境间最常见的其实是**报头没带**：开发环境的网关可能默认注入了认证头，测试环境没配。

- 答"证书或 HTTPS 配置问题"——很可能，尤其是"本地 HTTP、环境 HTTPS"这种差异，SSL 相关错误经常表现为认证失败。

- 答"接口版本不一致"——也对，特别是请求／响应字段不匹配时，有的框架会直接返回 4xx。

- 收口：三种答案恰好对应课件列的三类协议层错误——**这页的三条不是随便列的，是踩出来的**。

**易错点 / 考点：**① 两个方向要能默写并各举一例，案例题高频；② 常见错误：把"数据库访问测试"理解成"测 SQL 性能"——它测的是**映射与容错**；③ ORM 的全称 **Object Relational Mapping** 要会写；④ 陷阱："通信测

试要把业务功能全部重测一遍"——错，课件明确"检查核心功能即可"。

> **过渡：**通信和数据都盯住了，可还有一个更麻烦的问题没解决——**我根本拉不起对方的服务，怎么测？** 下一页的答案，是本章我最想让你记住的一个方法。

---

## 【5.81】微服务架构的集成测试 – 续（3 分钟）

这一页引出一个方法：**消费者驱动的契约测试**，英文缩写 **CDC**，全称 Consumer Driven Contract Testing。我先把"契约"这个词说准。

课件给的定义是：**契约规定的是接口的调用者（Consumer，消费者）和被调用者（Provider，提供者）之间约定的 Request 和 Response 数据交互格式**。请注意它规定的是**格式**，不是行为、不是性能、不是业务规则。它回答的问题非常具体：我发给你什么字段、什么类型、什么必填；你还给我什么字段、什么类型、什么情况下返回什么状态码。**就这些，不多不少**。

那为什么叫"消费者驱动"？因为**契约是从消费者的需求出发写的**。课件写得很清楚：**从 Consumer 的需求出发设计测试用例并产生一份契约，然后验证 provider 端的功能**。这个顺序非常关键——不是提供者说"我提供这些字段"，而是消费者说"我只需要这些字段、我依赖这些字段"。

这个方向为什么重要？因为提供者天然倾向于"多提供"：多给字段、多加参数，反正不影响自己。可对消费者来说，**多给不是问题，少给和改给才是灾难**。所以契约必须由消费者的真实需求来定义——**我只承诺我用到的，你就不能动这些**。

课件还提了"另一种观点：用契约测试或协议测试来做集成测试"。这句话的意思是，在微服务下，**集成测试的重心可以从"把系统拼起来跑"转移到"验证边界上的约定"**。为什么这是一种合理的观点？因为集成测试要发现的问题，八成就在边界上；而边界上的问题，本质是"两边理解不一致"。既然是不一致，那**把两边的理解显式写下来、再各自验证**，就比"拼起来跑一遍看运气"精准得多。

最后说清它不是什么。契约测试**不能替代**单元测试，也**不能替代**端到端测试。它解决的是"服务之间的约定有没有被遵守"，解决不了"业务算得对不对"。它的定位是**金字塔中间那一层**，把最难搞的跨团队协作，变成两份可以自动跑的测试。

> **板书：**中间写"契约 = Consumer ↔ Provider 约定的 **Request / Response 格式**"。下方三步箭头：① Consumer 需求 → 设计用例 ② **产生一份契约** ③ 验证 Provider。左侧写"消费者驱动＝我只承诺我用到的"；

> 右侧写"边界观点：集成重心从'拼起来跑'转到'验证约定'"。底下一行："契约测试≠单元测试，≠端到端测试"。

**提问：**为什么契约要由消费者来写，而不是让提供者把自己支持的字段全列出来？

**预设回答：**

- 答"消费者更清楚自己用什么"——对。提供者列全量字段，看着更"完整"，但没人能保证消费者真的只用这些，也没人能保证提供者以后不改。

- 答"提供者写会更准"——反例很明显：提供者写"我支持 name 字段"，可消费者其实依赖的是`user_name`，两边一跑就炸，而契约测试如果按提供者的说法写，反而会**给出错误的通过**。

- 追问：这体现了一个什么原则？答：**接口的约束应该由使用方定义**——和"需求由用户定义"是同一个道理。

**易错点 / 考点：**① CDC 的中英文全称必背：**Consumer Driven Contract Testing／消费者驱动的契约测试**；② 契约规定的是**格式**（Request／Response），不是业务逻辑，这是最爱考的一句话；③ 顺序不能错：**先消费者需求 → 再生成契约 → 最后验证提供者**；④ 常考简答："契约测试能不能替代集成测试"——答"能替代其中验证接口约定那部分，不能替代端到端业务验证"。

**过渡：**这个方法到底能带来多少好处？课件用三条把它说透了。

---

## 【5.82】契约测试及其优势、工具（3 分钟）

这一页是三条优势加四个工具，我们一条一条落地。

**第一条，降低服务集成的难度，把它分解为单元测试和接口测试**。这句话的分量在于"分解"两个字。原来微服务集成是一个巨大的、不可控的任务：要把 N 个服务拼起来。契约测试把它切成两半——消费者这一侧，用假的服务端跑自己的单元测试；提供者那一侧，用契约回放跑接口测试。**一个 N×M 的联调矩阵，被拆成了 N 加 M 两份独立的活**。这就是它最漂亮的地方：不是让集成变简单了，而是**让集成不再需要"同时在场"**。

**第二条，团队能以一种离线的方式完成验证，把联调的成本几乎降到零**。"离线"是关键词。传统联调要占环境、要约时间、要人等：前端等后端发版，后端等前端确认，一个字段改三天，两边都卡着。契约测试里，消费者拿着契约自己就能验证，提供者拿着契约自己也能验证，**谁都不用等谁**。课件用"几乎降到零"这个说法，指的就是这部分**等待与协调的成本**。你可以回想上一页我提的那个问题——"联调卡了几天、卡在谁等谁"，契约测试正是冲着这个去的。

**第三条，基于契约让接口的变更有迹可循**。以前接口改了，怎么发现影响面？靠人记、靠群里喊一句"我把字段改名了啊"。现在契约是一份**受版本管理的文件**，改了它就得跑验证，跑不过就是有消费者会挂。于是"接口变更"从一个**口头事件**变成了一次**可审计的动作**，影响面是查出来的，不是猜出来的。

工具方面，课件列的是 **Janus、Pact、Pacto、Spring Cloud Contract** 这四个框架（课件这里写的是 CDT，实际指的就是 CDC 工具）。其中 **Pact** 是社区里最流行的一套，文档在 docs.pact.io，后面几页的示例都是它；**Spring Cloud Contract** 是 Spring 生态的方案，测试用 Groovy 或 YAML 写契约，和你后面实验里的 Spring 项目更贴近。工具会换，原理不变：**写契约 → 生成桩 → 双向验证**。

**板书：**三条优势竖排：① **分解**：N×M → N＋M（消费者跑单测、提供者跑接口测试）② **离线**：不用同时在场，联调成本≈0 ③ **留痕**：契约进版本库，接口变更可审计。右侧写工具：Janus、**Pact**（docs.pact.io）、Pacto、Spring Cloud Contract。底部一行口诀："**分解—离线—留痕**"。

**提问：**假设提供者把响应里的 `userId` 改名成 `id`，但功能完全没变、测试也全绿。契约测试能不能拦住这个问题？

**预设回答：**

- 答"能，因为契约里写了字段名"——对，这正是第三条第优势的落点：字段名是契约的一部分，改名就是破坏契约。

- 答"不能，改了名但值一样"——要纠正：对消费者来说，**字段名就是接口**，`userId` 读不到就是 null，后面一连串空指针，功能上根本不是"没变"。

- 追问：那提供者怎么安全地改名？答：**先加新字段、两边并行一段时间、消费者迁完再删旧字段**——这叫契约的兼容演进，也是契约测试能支撑的实践。

**易错点 / 考点：**① 三条优势要能默写，尤其"**分解为单元测试和接口测试**"和"**离线验证**"这两个提法；② 工具名四个要记全，Pact 与 Spring Cloud Contract 最常考；③ 常见错误：认为契约测试是"提供者写契约、消费者遵守"——方向反了；④ 判断题："契约测试能发现业务逻辑错误"——错，它只保证约定被遵守。

**【思政落点 12】**：契约这两个字，在工程上和生活里是同一个意思——**我说到做到，你按我说的准备**。契约被遵守，才有"离线验证"：因为我相信你会按约定返回，我才敢不等你。反过来，接口悄悄改名、字段悄悄变可空，看似只少一行通知，实际是**把成本转移给了所有信任你的人**。写接口时请记住：**你发布的每个字段，都是对别人的承诺**。守约、留痕、变更前通知——这是技术规范，也是职业信用。

**过渡：**说完了道理，我们看它具体怎么跑起来。Pact 的官方文档给了一条很清晰的三段流程。

## 【5.83】示例：CDC 测试框架（2 分钟）

这一页给的是 Pact 的示例，地址是 docs.pact.io。我不逐图念，我们把它抽象成一条**三段流水**，你记住这条线，换任何 CDC 工具都通用。

**第一段，消费者端产出契约**。消费者写测试的时候，不连真实的服务端，而是让 Pact 起一个**本地的假服务端**（mock server），你告诉它"我发这个请求，你应该回这个响应"。测试跑的时候，请求打到假服务端上，消费者代码被完整地跑了一遍；跑完，Pact 把这次交互**记录下来**，写成一份 JSON 文件——**这份 JSON 就是契约**。

**第二段，契约被送到一个中枢**。这份 JSON 被发布到一个叫 **Broker（契约仓库）** 的地方，带上版本号。有了这个中枢，提供者才知道"现在有哪些消费者、依赖我的哪些行为、依赖的是哪个版本"。

**第三段，提供者端验证契约**。提供者从 Broker 拉取契约，启动自己的真实服务，**按照契约里记录的请求真的发一次**，然后拿真实响应和契约里约定的响应比对——字段、类型、状态码，一项一项对。全对，这条契约就是"已验证"；有一项不对，构建就红了，而且是**在提供者的流水线里红**，不是等联调时才发现。

三段跑完之后还有一步很关键：Broker 上有一个"**能不能部署**"的查询接口，部署前问一句"我要上的这个版本，它的契约都验证通过了吗"，通过了才允许上线。这一步把契约测试**接进了发布流程**，也就接上了我们后面要讲的 CI/CD。

顺便说一个术语口径：课件这里写的是"CDT 测试框架"，规范写法是 **CDC（Consumer Driven Contract）**，答题以 CDC 为准。

> **板书：**三段流水横排：① **Consumer 测试**：Pact 起 mock server → 跑消费者代码 → 生成 **契约 JSON** ② **发布到 Broker**（带版本）③ **Provider 验证**：拉契约 → 真发请求 → 比对响应 → 红／绿。第四格写"部署前问 Broker：这个版本能上吗"。右上角注："课件写 CDT，规范为 **CDC**"。

**提问：**如果契约一直是绿的，但上线后还是出了对接问题，最可能是哪一环漏了？

**预设回答：**

- 答"契约写得不全"——最常见。契约只覆盖了消费者**已经实现的**调用，新写的调用还没进契约，自然测不到。

- 答"提供者验证用的不是生产版本"——很关键，验证必须跑在**准备发布的那个构建**上，否则验的是另一个东西。

- 答"契约没进发布门禁"——最要命的一种：契约测试跑是跑了，但没人卡它，红了也照发。

- 收口：所以判断契约测试有没有真落地，看三件事：**契约覆盖了多少调用、验证跑在哪个构建上、红了能不能拦住发布**。

> **易错点 / 考点：** ① 三段流程要能复述（消费者产契约 → Broker 中转 → 提供者验证）；② 术语：Broker＝契约仓库，mock server＝消费者端的假服务端，两者别混；③ 口径提醒：**CDC 不是 CDT**；④ 常考判断："契约测试要求消费者和提供者在同一时间联调"——错，恰恰相反。

> **过渡：** 三段里的第一段最容易被误解——消费者到底在测什么？下一页专门讲。

---

## 【5.84】Consumer 测试（2 分钟）

消费者测试，核心道具是 **Stub server**，也就是桩服务端。

我们要搞清楚它在测谁。**它测的是消费者自己**——我这个客户端代码，在拿到符合约定的响应之后，能不能正确解析、正确转换、正确展示。它**不测**提供者，因为对面是个假的。

那假服务端怎么知道该回什么？靠你在测试里**声明期望**。你写"当我 GET 这个路径、带这个头时，请回 200 和这段 JSON"，Pact 就把这个期望变成行为：**真的起一个 HTTP 服务，监听一个端口，按你声明的规则应答**。所以它叫 Stub server 而不是 Mock 对象——它是**进程内的一个真服务端**，走的是真的 HTTP 协议栈。这一点很重要：因为走了真协议，序列化、报头、状态码这些"协议层的问题"才有机会暴露，而如果只是一个方法级的 Mock，这些全被绕过了。

测试跑起来的顺序是这样：先声明期望 → 起假服务端 → 消费者代码发真请求 → 假服务端按期望应答 → 断言消费者解析结果对不对 → **最后把这次交互写成契约**。

请注意最后这一步的分量：**契约是测试的副产品，不是手写的文档**。这是 CDC 最聪明的地方——文档会过期，测试不会。你改一行消费者代码，跑一次测试，契约自动更新；忘了更新？测试就红。

还有一个实践细节：消费者测试里应该**多造异常场景**。返回 500 怎么办、返回空数组怎么办、返回字段类型和约定不一致怎么办、超时怎么办。因为对面是假服务端，**造这些场景的成本几乎为零**——这在真实联调里是做不到的，你没法让别人的服务专门给你返回一个 503。

> **板书：** 中间画"消费者代码 → 真 HTTP 请求 → **Stub server**（按声明的期望应答）"。下方流程：声明期望

→ 起服务 → 发请求 → 断言解析 → **生成契约**。右侧两条批注：① 测的是消费者自己，不测提供者；② **契约是测试的副产品，不是手写文档**。底部："异常场景在这里最便宜：500／空数组／类型不符／超时"。

**提问：**消费者测试里，如果我把期望写错了——比如实际接口返回的是字符串、我期望成数字——测试会不会红？

**预设回答：**

- 答"不会红，因为假服务端按我写的回"——完全正确，这是这个方法的**固有盲区**：消费者测试只能证明"我能处理我以为的响应"，不能证明"真实响应就是我以为什么样"。

- 答"会红，因为跟真实服务对不上"——那就跑到提供者验证阶段了，消费者这一侧不会红。

- 追问：那这个盲区由谁补上？答：**提供者验证**——它拿真实服务去对契约，一旦真实响应和契约不符，构建就红。**两端互补，才闭环**。

**易错点 / 考点：**① Stub server 是**真起一个 HTTP 服务**，不是方法级 Mock，这个区别常考；② 消费者测试测的是消费者，别答成"测接口对不对"；③ 契约由测试**自动生成**，这是 CDC 与"手写接口文档"的本质差别；④ 盲区问题："消费者测试全绿能说明接口没问题吗"——不能，要靠提供者验证补。

**过渡：**那就看另一端——提供者怎么拿契约反过来验自己。

---

## 【5.85】Provider 测试（2 分钟）

提供者测试，核心道具是 **contract verifier**（契约验证器）。课件给了它的仓库地址，是 pact-foundation 的 pact-provider-verifier。

它干的事和消费者正好相反：**从 Broker 把契约拉下来，然后拿真实的服务去"应考"**。具体流程：把契约里记录的每一条请求，**原样发给真实运行的提供者**；拿到真实响应后，和契约里约定的响应**逐项比对**——状态码、报头、响应体的字段和类型。全对就通过，有一项不对就失败，并且会明确告诉你"是哪条契约、哪个字段、期望什么、实际什么"。

课件这一页还写了几个工程上的关键点。**Contract verifier tests** 指的是"由契约驱动的验证测试"，它不是手写的用例，而是**由契约自动生成**的——这就是那个 **auto-generate**。你在图上还能看到 `Contract.groovy` 和 `Call API`：前者是契约（Spring Cloud Contract 里常用 Groovy DSL 写），后者是验证器真正发出调用的那一步。

这里有个特别值得强调的设计：**提供者不需要知道消费者是谁、用什么语言写的**。它只需要读契约 JSON，发 HTTP 请求，比响应。所以一个 Java 服务可以被一个前端、一个 Python 脚本、一个 Go 服务同时"契约验证"，**跨语言是天然的**——这也是它比"造 Stub"高明的地方，Stub 是每种语言各写一套，契约是**一份文件所有人共用**。

最后提醒一个容易忽略的前提：提供者测试必须跑在**你准备发布的那个构建**上，而且**依赖要真实**（真数据库、真配置）。如果验证时又把它 Mock 了一层，那验的还是假的，等于白跑。

**板书：**流程横排：**Broker 拉契约 → 逐条回放请求 → 打真实 Provider → 比对响应（状态码／报头／字段类型）→ 红／绿**。旁注三个词：contract verifier、auto-generate（契约自动生成用例）、`Contract.groovy`（契约 DSL）＋`Call API`（真发调用）。右下角："跨语言天然：一份 JSON，谁都能验"；"必须跑在待发布构建上、依赖要真"。

**提问：**提供者验证失败了，是提供者的错，还是消费者把契约写错了？

**预设回答：**

- 答"提供者的错"——多数情况是，提供者改了行为没通知。

- 答"可能是消费者写错了"——也存在：消费者按自己的想象声明了期望，而真实接口从来不是那样。这时要判断**谁是对的**：如果契约反映的是合理的消费需求，那提供者要改；如果契约写错了，就改契约。

- 追问：那怎么减少这种争议？答：**契约评审**——把契约当接口文档一样评审，这正是它比口头约定强的地方：**争议有实物可依**。

**易错点 / 考点：**① 两端对应关系必考：**Consumer 端用 Stub server 生成契约，Provider 端用 contract verifier 验证契约**；② 提供者验证的用例是**自动生成**的，不是手写；③ 常见错误："提供者测试需要启动所有消费者"——错，只需要契约；④ 判断题："契约测试要求两端同语言"——错，跨语言是它的优势。

**过渡：**两端都讲完了，我们把它们放在一张图上对照，你就能看清这条闭环。

---

## 【5.86】示例（2 分钟）

这张图把两端放在一起，标了两个身份：**Server side 是 service provider（服务提供者）**，**Client side 是 service consumer（服务消费者）**。我们顺着看这条闭环是怎么合上的。

左侧是客户端。它在自己的测试里，面对的是一个按契约应答的假服务端，测的是自己的解析和业务处理；测试跑完，产出一份契约文件。这份文件被发布出去，进入共享的契约库。

右侧是服务端。它从契约库拿到同一份文件，用验证器把每条请求打到真实服务上，比对响应。通过了，说明"我仍然满足所有消费者的期待"；不通过，说明"我这次改动破坏了某人"。

这张图最重要的信息，不是流程，而是**那份文件在中间**。请注意：**消费者和提供者之间，传递的不是代码、不是环境、也不是人，而是一份可执行的文件**。这就是 CDC 的本质——**把协作从"人对人沟通"变成"文件对文件验证"**。

再对照一下我们前面讲的单体策略，你会有个顿悟：单体里的 Stub 是**每个调用方各写一个假模块**，微服务里的契约是**一份共享的约定**。单体里 N 个模块要拼，就有 N 份桩；微服务里 N 个消费者对 1 个提供者，只需要 N 条契约，而且**提供者一次验证全部搞定**。所以 CDC 不是凭空发明的新东西，**它是把"桩"这个老思路，从"各自造假的"升级成了"共享约定的"**。

这也解释了为什么它能"把联调成本降到零"：因为成本的大头从来不是写假服务，而是**协调**——约时间、对版本、等人回消息。契约把这部分成本直接删掉了。

> **板书：**左右两端：**Client side / Consumer**（Stub server → 生成契约）｜**Server side / Provider**（contract verifier → 回放验证）。中间画一个文件图标写"**契约（JSON）**"，箭头从消费者指向契约、从契约指向提供者。下方对比一行："单体 Stub：各写各的假模块；CDC：一份共享约定，提供者一次验证全部"。

**提问：**为什么说契约测试是把"桩"的思路升级了，而不是发明了一个新东西？

**预设回答：**

- 答"都是拿假的顶替真的"——有一半对。差别在于：Stub 的"假"是**每个调用方私有的一份想象**，契约的"假"是**双方共同确认的一份约定**，而且能被真实服务反向验证。

- 答"因为契约能自动验证"——这是升级的关键动作：**从"假设对方会这样"变成"验证对方确实这样"**。

- 追问：那这个升级解决了 Stub 的什么老毛病？答：**漂移**——Stub 写完之后，对方改了，你的假模块还在按老样子应答，测试全绿、上线全炸。契约一旦漂移，提供者那边立刻红。

**易错点 / 考点：**① 两个角色的英文要会对应：**provider＝提供者＝server side，consumer＝消费者＝client side**，考试常故意对调；② 这张图的标准问法是"简述 CDC 的完整闭环"，答题要点必须包含"契约在中间传

递"；③ 常考对比："Stub 与契约的区别"——答"私有想象 vs 共享约定 + 可被真实服务反向验证"。

**过渡：**契约要写，可字段那么多，总不能全靠手敲。工程上有个现成的来源——接口文档。下一页看它。

---

## 【5.87】如何获取接口信息（2 分钟）

这一页讲的是工具链上的一个衔接点：**接口信息从哪儿来**。课件给的是 **Swagger 生成的动态接口文档**示例。

什么叫"动态"？就是文档不是人写的，是**从代码里生成的**。你在 Controller 的方法上打注解，说明这个接口的路径、参数、返回类型，工具扫描一遍，就产出一份结构化的接口描述——现在通行的标准叫 **OpenAPI**，Swagger 是这套东西最常用的工具链。所以它天然"跟代码同步"：代码改了，文档跟着变，不会出现"文档标 v1、代码跑 v2"这种经典事故。

它和契约是什么关系？我的理解是：**文档描述"我提供什么"，契约规定"你依赖什么"**。这两者不一样。文档是提供者视角的全集，契约是消费者视角的子集。但文档是**写契约最好的原料**——你从文档里把"我这个消费者真正用到的那几个字段"摘出来，就是一份契约的初稿。

反过来，文档还能省掉一大半客户端代码。有了结构化的接口描述，工具可以**自动生成客户端 SDK 和桩服务**：类型定义、序列化、请求方法，全是生成出来的。所以你会看到一个很顺的链路：**注解 → OpenAPI 文档 → 生成客户端／桩 → 写契约 → 两端验证**。这条链上每一步都是自动的，人只负责"确认需求"。

但也要提醒一句：**别把文档当契约用**。文档说"这个字段可能为空"，你的代码按非空处理，文档不会报错，运行时会给你一个空指针。**文档是说明书，契约是考试卷**——说明书可以写得很全，考试卷只考你真用到的那几道题，但考不过就是考不过。

**板书：**链路横排：**代码注解 → Swagger／OpenAPI 文档（动态生成）→ 生成客户端 SDK／桩服务 → 摘出消费者依赖写成契约 → 两端验证**。中间写一句对比："文档＝提供者视角的全集（说明书）；契约＝消费者视角的子集（考试卷）"。

**提问：**有了自动生成的接口文档，是不是就不用写契约了？

**预设回答：**

- 答"不用了，文档已经很全"——常见误解。文档只**声明**，不**验证**；它不会因为提供者改坏了就变红。

- 答"要写，文档只是原料"——对。而且要从文档里只挑**消费者真正依赖的字段**，多挑会让契约过于脆弱，提供

者改任何东西你都红。

- 追问：那契约写多细合适？答：**只写你真正依赖的**——这是"消费者驱动"的精髓，也是控制契约维护成本的关键。

> **易错点 / 考点：**① Swagger 文档是**从代码动态生成**的，不是手写，这是选择题常考点；② OpenAPI 是规范名，Swagger 是工具链名，别混答；③ 核心判断："文档 ≠ 契约"——文档声明、契约验证，这是简答题的得分点；④ 契约粒度原则："只写真正依赖的字段"。

> **过渡：**最后一种情况最常见：你要测的服务 A，依赖 B 和 C，可 B、C 你既拉不起来、又还没写契约。怎么办？

---

## 【5.88】利用 Mock 技术解除微服务之间的依赖（3 分钟）

> 这一页给的是最朴素、也最常用的一招：**用 Mock 把微服务之间的依赖解除掉**。图上标的很清楚：中间是**微服务 A，标注 SUT**，也就是被测系统；它两侧的 **Service B 和 Service C 都被 Mock 掉了**；把这条链打通之后做的，是 **API Test**——在 A 的接口边界上做测试。
>
> 先讲它解决什么问题。你要测 A 的一个接口，A 要调 B 和 C。B 是别的团队维护的，C 还在开发中，两个都不一定在线。如果非要等它们，你的测试就永远跑不起来。Mock 的办法是：**在 A 的外部依赖点上放一个假的**，A 发请求给它，它按预设返回。于是 A 变成了**一个只依赖自己的单元**，随时能跑、随时能重复跑。
>
> 具体 Mock 什么？课件强调"**Mock 请求和响应信息**"——也就是说，你 Mock 的粒度是**协议层的一对报文**：请求长什么样、回什么样。这跟前面契约里的"交互"是同一个粒度。
>
> 这一招的收益有三条。**第一，测试可控**：你可以让 B 返回 500、返回超时、返回一堆脏数据，然后看 A 怎么处理——这在真实环境里根本做不到。**第二，测试快**：不起 B、不起 C，不连它们各自的库，一个测试从秒级降到毫秒级。**第三，解耦进度**：B 没做好，A 照样能测，团队不用互相等。
>
> 但必须讲清它的风险，否则会误用。**最大的风险是"漂移"**：你 Mock 的报文是照着当时 B 的样子写的，B 后来改了，你的 Mock 不知道，测试继续全绿，上线全炸。**第二个风险是"测了假的"**：如果连 A 自己的数据库、消息队列也一起 Mock 掉，那这个测试就退化成单元测试了，集成测试的意义没了。
>
> 所以正确的用法是：**Mock 只用在"我控制不了的远端"，"我自己这一侧的依赖（库、缓存、MQ）要用真的"**。而防漂移的办法，就是我们前面讲的那一套——**把 Mock 的报文固化成契约，让提供者去验证**。你看，绕了

一圈，Mock 和契约其实是配套的：**Mock 让你现在能测，契约让你以后还测得准**。

> **板书：**画图：`Service B（Mock） ／微服务 A（SUT） ／ Service C（Mock）`，A 的两侧箭头标"Mock 请求与响应报文"，A 外面画个框标"**API Test 的边界**"。下方三条收益：可控（能造 500／超时／脏数据）／快（毫秒级）／解耦进度；两条风险：**漂移**（Mock 与真实不一致）、**测了假的**（把自家依赖也 Mock 掉）。底部："Mock 管现在，契约管以后"。

**提问：**一个团队把所有外部依赖都 Mock 了，测试天天绿，可一上线就出问题。问题出在哪？

**预设回答：**

- 答"Mock 写错了"——可能，但更常见的是 **Mock 从没跟真实服务对过**，也就是漂移。

- 答"没有一个环节验证真实接口"——这才是根因。缺的不是 Mock，缺的是**提供者侧的契约验证**。

- 答"把自家数据库也 Mock 了"——这个也要点出来：那样测的就不是集成，是单元了，集成测试的价值全部丢失。

- 收口：所以判断一套微服务测试是否可靠，问三句：**Mock 的报文是不是契约、契约有没有在提供者端验证、自家依赖是不是真的**。

**易错点 / 考点：**① SUT 的含义（System Under Test，被测系统）要会；② 这一页的典型考法是"用什么方法解除微服务依赖"——答 **Mock（测试替身）**，并说明 Mock 的粒度是请求／响应报文；③ 常见错误：认为 Mock 一多就是好事——**Mock 越多，离真实越远**，边界要卡在"我控制不了的远端"；④ 简答题要点：Mock 的收益（可控／快／解耦）与风险（漂移／失真），以及用契约防漂移。

> **过渡：**到这里，集成的"怎么测"讲完了。最后一个问题：这一切在什么时候发生？答案是——**每次提交都发生**。这就是持续集成。

---

## 【5.89】CI/CD 与持续测试（1 分钟）

这是本章最后一块，也是把前面所有内容串起来的一块：**CI/CD 与持续测试**。

先问一个问题：前面讲的单元测试、Mock、覆盖率、契约测试，它们都是"好东西"，可为什么很多团队明明都知道，却还是做不好？原因往往不是不会写，而是**没有一个机制让它们每次都发生**。人是有惰性的：这周赶进度，单元测试先不写了；这个版本急，覆盖率先不看。于是这些实践在文档里存在，在流程里消失。

CI/CD 就是那个**强制机制**。它把"跑测试"从一件靠自觉的事，变成一件**流水线上自动发生、不通过就过不去**的事。所以这一块不是在讲运维，是在讲**怎么让测试真正落地**。

还有一个更根本的理由：测试能力是**会被消耗的**。一个团队的手工测试经验，往往集中在两三个人身上；人一走，经验就断。而流水线上跑的东西是**沉淀下来的资产**：用例、脚本、门禁标准、报告基线，谁都能接手。所以把测试搬进流水线，不只是提速，也是**把个人经验变成团队能力**。

这一块四页：**流水线长什么样**（两次）、**为什么 CD 会倒逼持续测试**、**一个真实的流水线示例**、**持续测试有哪四个特点**，最后一页是实践框架。学完这一块，你应该能把第 5 章所有技术**摆到一条时间轴上**——每项技术在流水线的哪一段跑、跑完卡什么。

带着一个问题听：**你们项目现在，从提交代码到能上线，中间有几步是自动的？**

**板书：**中间写"CI/CD 与持续测试"，下面四行：① 流水线形态 ② CD 倒逼持续测试 ③ Amazon 示例 ④ 持续测试四特点 + 实践框架。左侧竖写一句："实践靠机制落地，不靠自觉"。

**提问：**一个团队单元测试写得很规范，但每次发版前才跑一次。这套实践算不算"落地"了？

**预设回答：**

- 答"算，写了就是落地"——不算。**测试的价值取决于它多久跑一次**：跑得越晚，发现的问题越贵，而且改起来牵扯越多。

- 答"不算，得进流水线每次跑"——对，这正是这一块的主题。

- 追问：那"每次跑"会不会太慢？答：会，所以才有下一页之后要讲的**分层与精准**——不是所有测试都每次都跑全套。

**易错点 / 考点：**① CI/CD 在测试课里的定位是**测试的执行与门禁载体**，不是运维话题；② 常考概念区分：CI（持续集成）／持续构建／CD（持续交付与持续部署）——四个词的区别是本节高频考点；③ 判断题："持续测试就是自动化测试"——错，持续测试强调**在流水线中持续发生**，范围比自动化测试更大。

**过渡：**先看这条流水线整体长什么样，你才有地方安放前面学的每一项技术。

---

## 【5.90】CD 倒逼持续测试（3 分钟）

这一页是一条链，五个词：**持续集成 → 持续构建 → 持续交付 → 持续部署**，最后落到**持续测试**。我们一个一

个说清，因为这是必考的概念辨析。

**持续集成（CI）**：开发人员频繁地把代码合到主干，每次合并都自动触发编译和测试。它解决的是"集成地狱"——以前大家各写几个月，最后合并时冲突成堆。关键词是**频繁**和**自动**。

**持续构建**：每次代码变化都自动构建出可运行的制品。差别在于关注点从"代码能不能合"变成"制品能不能出"。**持续交付**：保证任何时候都有一份**可以随时发布到生产**的制品，但**发布这个动作由人决定**——一键可发，按不按人说了算。**持续部署**：连"按"都不用人，通过所有门禁就自动上生产。所以交付与部署的区别只有一条：**最后那一下是不是自动的**。

现在说"倒逼"。为什么 CD 会**逼出**持续测试？逻辑很硬：当发布频率从"每月一次"变成"每天十次"，**人工回归测试在物理上就不可能完成**。你不可能每天组织十次回归。于是只有两条路：要么放弃质量，要么**让测试自动化、并且快到能跟上发布节奏**。这就是"倒逼"——**不是测试自己想起来要进步，是交付节奏压着它必须进步**。

这个倒逼会连带改变测试的形态。第一，**测试必须分层**：大量快速的单元测试挡住大部分问题，少量慢的端到端测试守住关键路径。第二，**测试必须精准**：不可能每次都全回归，得能算出"这次改动影响了哪些用例"。第三，**测试必须左移**：等代码写完再测，根本来不及，得在编码阶段就把问题挡掉。

反过来看，这也解释了一个常见困惑：为什么有些团队上了 CI 工具，测试还是老样子？因为**他们只是把手工流程自动化了一遍，没有改变测试的形态**。CI 工具不是目的，**它是一面镜子，照出你的测试够不够快、够不够准**。

> **板书：**横排链条：**持续集成（频繁合并＋自动构建测试）→ 持续构建（自动出制品）→ 持续交付（随时可发，人按发布）→ 持续部署（自动上生产）**。链条下方画一个大箭头写"**倒逼**"，指向"持续测试"。再往下三行：分层／精准／左移。右下角辨析："交付 vs 部署 ＝ 最后那一下是否自动"。

**提问：**一个团队做到了持续部署——每次合并通过门禁就自动上生产。那他们的测试必须要满足什么条件？

**预设回答：**

- 答"必须全自动"——必要但不充分。全自动但跑两小时，还是跟不上。

- 答"必须快、必须准、必须能拦住"——完整答案：快（分钟级反馈）、准（只跑受影响的）、拦得住（门禁卡死）。

- 答"还得能回滚"——很好的补充，自动部署必须配自动回滚和监控，这是"测试右移"的部分。

- 收口：所以持续部署不是一个部署动作，它要求**整套测试与监控体系同时到位**——这就是"倒逼"的真实含义。

**易错点 / 考点：**① **持续交付 vs 持续部署**：唯一的区别是"上生产是否自动"，这是最高频的辨析题；② 四个词要能按顺序说清，且能各用一句话定义；③ "倒逼"的逻辑要能讲出来：**发布频率上升 → 人工回归不可行 → 测试必须自动化、分层、精准**；④ 常见错误：把 CI 等同于"用 Jenkins 之类的工具"——工具只是载体。

**过渡：**理论讲完，我们看一条真实的流水线是怎么排的——先看一个业界示例。

---

## 【5.91】CI/CD 流水线（二）环节排布（2 分钟）

这一页又是一张流水线图。前面那张讲的是"形态"，这张讲的是"排布"——**同样的门，摆放顺序不同，效果差很远**。我给你三条排布原则。

**第一条：便宜的放前面，贵的放后面**。编译几秒钟，静态扫描一两分钟，单元测试几分钟，集成测试十几分钟，端到端半小时以上。这个顺序不是随便排的，它符合一个经济学直觉：**越早失败越省钱**。一个编译错误不该等到跑完半小时的端到端测试才被发现。

**第二条：确定性的放前面，不确定的放后面**。什么叫不确定？依赖外部环境的测试就是不确定的——网络抖动、第三方接口超时、测试数据被别人改了。这类测试放在前面，会让流水线"时红时绿"，团队很快就学会"红了再点一次重跑"，**门禁就此失效**。所以不确定的测试必须往后放，并且要单独统计它的稳定率。

**第三条：只在必要的分支上跑全套**。主干和发布分支跑全套门禁，功能分支只跑快速门禁。这不是降低标准，而是**把有限的门禁预算花在最需要的地方**——毕竟门禁的目的是拦住问题进入主干和生产，不是给每个分支都做体检。

最后提一句"失败要快"这个工程原则。它的完整说法是 **fail fast**：不仅流水线要尽早失败，**测试本身也要尽早失败**——一个用例里如果有五个断言，第一个就错了，后面四个不该继续跑；一个测试套件里如果连数据库都连不上，就不该傻等着超时。你在写测试时也要有这个意识：**让错误尽早、尽量明确地暴露**。

**板书：**横向阶梯图，从便宜到贵：`编译（秒） → 静态扫描（分） → 单元测试（分） → 集成／契约（十分钟级） → 端到端（半小时级）`。下面三条原则：① 便宜在前、贵在后（越早失败越省）② 确定性在前、不确定性在后（防"红了重跑"）③ 主干跑全套、分支跑快速。底部写 **fail fast**：用例内早失败、套件内早失败。

**提问：**团队里流行一句话——"红了不要紧，点一下重跑就绿了"。这句话暴露了流水线的什么问题？

**预设回答：**

- 答"测试不稳定"——对，但不稳定只是症状。根因往往是**不确定的测试被放在了前面**，或者测试之间共享了状态。

- 答"门禁形同虚设"——更本质。一旦大家习惯"重跑"，门禁就从"拦截"变成"抽奖"，信任崩塌之后，所有红灯都会被忽略。

- 追问：怎么治？答：**隔离不确定测试、统计不稳定率、超过阈值就修或下线**——宁可少一个用例，不要一条会撒谎的流水线。

> **易错点 / 考点：**① 排布原则要能说三条，尤其"**便宜在前、贵在后**"和"**确定性在前**"；② fail fast 的含义要准确（尽早失败、明确暴露），别答成"快速失败重试"；③ 常考判断："流水线应该把所有测试每次都跑全"——错，成本上不可行，也不必要。

> **过渡：**原则有了，看一个业界的具体例子——课件用的是 Amazon。

---

## 【5.92】CI/CD 流水线（三）完整环节与门禁（3 分钟）

这一页是流水线的完整展开，我按"入口—检查—放行"三段带你看，然后指出三个最容易被忽略的环节。

**第一段，入口——谁触发。**课件列了四种：**代码提交**（最常见，一 push 就跑）、**定时触发**（每天夜里跑一次全量回归）、**事件触发**（比如依赖库升级、镜像更新、上游服务发了新版本）、**人工触发**（指定某次构建、某个分支）。四种触发的意义不同：提交触发保证"每次改动都被检查"；定时触发补上"没改动但环境变了"的情况；事件触发处理"我没改，但世界变了"；人工触发用于特殊验证。**只配提交触发是不够的**，这是很多团队的漏洞。

入口之后立刻是**排队限流**。为什么需要它？因为提交一密集，构建就会挤在一起抢资源，最后大家一起慢。限流的作用是**保证关键分支优先、防止资源被打爆**。

**第二段，检查。**顺序是：**编译 → 静态扫描（Sonar 增量扫描）→ 安全扫描 → 单元测试 → 自动化测试**。这里有个专业细节值得单独讲：**增量扫描**。全量扫一遍代码库可能几十分钟，增量扫描只扫本次改动涉及的部分，把反馈压到分钟级——这就是前面说的"便宜在前"的具体实现。安全检查放在单元测试**之前**，意思是"有高危漏洞的代码，连测试都不必跑"，这体现的是**风险前置拦截**的思路。

**第三段，放行。质量门禁**是一道总闸——所有检查的结论在这里汇总，达标才放行。放行之后是**部署 STG**（预发布环境）、**灰度环境**（小流量验证）、**生产环境**。注意这个顺序的用意：**每一级都是一次风险放大**，

所以在低风险环境里尽可能把问题暴露完。

还有四个环节容易被忽略，我要点出来：**代码跨数据中心同步**（多地构建的一致性）、**编译结果预推送、制品预推送**（把构建产物提前送到各环境，缩短部署等待）、**动态扩容**（构建高峰时临时加机器）。这四个不是"主流程"，但**没有它们，主流程就会卡**。

**板书：**三行流程。入口：**代码提交／定时／事件／人工 → 排队限流**。检查：**编译 → Sonar 增量扫描（静态）→ 安全扫描 → 单元测试 → 自动化测试**。放行：**质量门禁 → 部署 STG → 灰度 → 生产**。侧边小字：跨数据中心同步／编译结果预推送／制品预推送／动态扩容。右上角批注："安全扫描放在单测前＝风险前置"。

**提问：**为什么"安全扫描"要放在单元测试**前面**？按"便宜的在前"原则，单元测试可能更快啊。

**预设回答：**

- 答"因为安全问题更严重"——方向对，但要说得更工程化：**高危漏洞属于"根本不该进入后续流程"的问题**，它的处置成本与测试结果无关，越早拦越省。

- 答"因为扫描快"——不一定，全量扫描很慢，能放前面靠的是**增量扫描**。

- 追问：这说明排布原则不止"便宜在前"一条，还有什么？答：**按"问题的严重性与处置方式"排**——能被立刻判定为"不可接受"的检查，尽量往前放。

**易错点 / 考点：**① 流水线环节的顺序是**填空与排序题的高发区**，务必记住"编译→静态→安全→单测→自动化→门禁→STG→灰度→生产"这条主干；② 四种触发方式要能说全，尤其**事件触发**最容易被漏；③ "Sonar 增量扫描"是关键词，全量扫描撑不起持续集成；④ 三个易忽略环节（预推送、跨中心同步、动态扩容）常作为多选题出现。

**【思政落点 13】**：质量门禁最有意思的地方是**它不商量**：单元测试失败就是失败，覆盖率不达标就是不放行，不会因为"这个版本很重要"就网开一面。这恰恰是它有价值的原因——**规则一旦可以为例外让步，就不再是规则，只是一句建议**。同学们以后一定会遇到"这次特殊，先过了吧"的时刻。希望你能守住那个"不特殊"，因为**破例一次就会有第二次**。守规则不是死板，是对交付负责。

**过渡：**流水线搭好了，那"持续测试"到底要做到什么样才算合格？课件给了四个词。

---

## 【5.93】持续测试的特点（3 分钟）

这一页是本节最该背下来的一页：持续测试的**四个特点**——足够快、准确且有效、平滑有序、与 CI/CD 集成。我一个一个讲，每个都给你一个"判断标准"，这样你答题和干活都能用。

**第一个，足够快**。课件写的是"整个测试过程要快，一方面高度自动化测试，另一方面业务端到端的探索式测试"。这句话有两层，很多人只看到第一层。第一层好理解：能自动的都自动。第二层才是精髓：**自动化解决不了的部分，要靠探索式测试去补**——而探索式测试要快，靠的是**人的经验被复用**（会话式测试、检查表、缺陷模式库），而不是靠多派人。所以"快"不是单一的提速，是**机器管确定性、人管不确定性**。

**第二个，准确且有效**。课件说"被测系统往往很复杂，不可能做全回归测试，而是要推行**精准测试**"。这句话是持续测试的**技术核心**。全回归的成本随系统规模线性上涨，而发布频率也在上涨，两条线一交叉，全回归必然崩。精准测试要解决的是：**这次改动，到底影响哪些用例**——靠代码变更分析、调用链分析、历史缺陷关联，把回归集缩到最小但够用的子集。判断标准很简单：**你能不能说清"这次改动为什么只跑这些用例"**，说不清就是全回归的伪装。

**第三个，平滑有序**。课件把"有序"解释成三个打通：**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**。意思是测试不该是各自为政的孤岛，而应是**一条按成本和风险排好序的链**：左移管预防、右移管监控，单元管内部、系统管整体，静态管不运行就能发现的、动态管必须跑起来才能发现的。判断标准：**能不能把所有测试活动画在一条时间轴上，没有断点、没有重叠浪费**。

**第四个，与 CI/CD 集成**。课件说"与研发的持续构建、持续集成、持续部署、运维等环境的集成"。这一条是前三条的**落地条件**：再快的测试不进流水线就不会被执行；再精准的选择不接构建系统就选不出范围。判断标准：**测试是构建的一部分，而不是构建之后的一个独立动作**。

四个特点其实是一条逻辑链：**快**是前提，**准**是手段，**有序**是结构，**集成**是保障。

**板书：**四行，每行"特点 + 判断标准"：

① **足够快**：自动化＋端到端探索式 → 判据：机器管确定性、人管不确定性

② **准确且有效**：不做全回归，推行**精准测试** → 判据：能说清"为什么只跑这些用例"

③ **平滑有序**：左移↔右移、单元↔系统、静态↔动态三个打通 → 判据：所有测试能画在一条无断点时间轴上

④ **与 CI/CD 集成**：接持续构建／集成／部署／运维 → 判据：测试是构建的一部分

底部一条链：快（前提）→ 准（手段）→ 有序（结构）→ 集成（保障）。

**提问：**"我们每次都跑全量回归，所以我们的测试很充分。"这句话在持续测试的语境下有什么问题？

**预设回答：**

- 答"太慢了，跟不上发布节奏"——对，这是最直接的后果：全回归跑不动，最后一定演变成"发版前跑一次"，持续测试就没了。

- 答"全回归不代表充分"——也对，跑得多不等于测得准，很多缺陷在没被设计的用例里。

- 追问：那精准测试是不是"少测"？答：不是少测，是**把力气放在受影响的地方**——精准的前提是"能证明影响范围"，证明不了就不能缩。

- 收口：所以"全回归"和"精准测试"的差别不在数量，在**有没有依据**。

**易错点 / 考点：**① 四个特点的名称与解释是**简答／论述题的高频考点**，要能各配一句原文式的解释；② "精准测试"是关键词，答"减少测试用例"是不准确的，要答"**基于变更影响分析选择用例**"；③ "平滑有序"的三个打通要能背：左移—右移、单元—系统、静态—动态；④ 常见错误：把"持续测试"理解成"一直不停地跑测试"——它是**在流水线中按节奏、分层、有选择地跑**。

**过渡：**四个特点讲的是"应该做到什么"，那"怎么一步步做到"？业界有一份专门的白皮书。

---

## 【5.94】持续测试实践框架（1 分钟）

这一页指向一份资料：**《持续测试白皮书》**，课件给了扫码下载的入口。我不逐条念它，讲清楚"为什么需要一份框架"更有用。

因为持续测试这件事，最难的从来不是"会不会写自动化脚本"，而是**怎么让一整个组织持续地做下去**。你只讲技术，会得到一个尴尬的结果：几个人写了一批脚本，半年后没人维护，全部失效。所以成熟的做法一定是**多维度同时推进**——通常包括这么几块：**组织与角色**（谁负责测试资产、谁维护流水线）、**流程与规范**（什么改动跑什么测试、门禁标准是什么）、**技术与工具**（分层框架、环境与数据管理）、**度量与改进**（反馈时间、逃逸缺陷率、用例稳定率）。这四块缺一块，另外三块都会退化。

这份框架对你的直接价值是：**给你一张"能力清单"**。你以后进公司，如果想推动持续测试，不要一上来就谈工具，先拿这张清单对一遍——我们现在缺的是哪一块？缺组织，就先定责任；缺度量，就先把反馈时间和逃逸率统计起来。**先能量化，才能改进**。

顺便说一句：这类白皮书的价值不在"读完"，在**当检查表用**。读一遍记不住，但拿着它一条条对现状，能立刻发现缺口。

再说清一个边界：白皮书是**参考框架**，不是强制标准，也不是成熟度评级。它的作用是给你一套共同语言——当你跟别人说"我们的持续测试卡在度量这一块"，对方能立刻明白你在说什么。**有了共同语言，改进才谈得下去**。

**板书：**中间写"持续测试实践框架（《持续测试白皮书》）"，四周四个块：**组织与角色／流程与规范／技术与工具／度量与改进**。下面一行："缺一块，三块退化"。底部："先能量化，才能改进——反馈时间、逃逸缺陷率、用例稳定率"。

**提问：**如果一个团队技术很强、自动化脚本写得很好，但半年后脚本全废了。按这个框架看，最可能缺哪一块？

**预设回答：**

- 答"缺组织，没人负责维护"——最常见。测试资产如果没有明确 owner，就会随人员流动一起消失。

- 答"缺度量，没人知道它失效了"——也很关键：如果没人统计用例稳定率和反馈时间，脚本烂掉的过程是无声的。

- 追问：这告诉我们什么？答：**持续测试是组织能力，不是个人技能**——这也是它叫"持续"的原因。

**易错点 / 考点：**① 这一页考法很轻，通常只考"持续测试实践框架／白皮书的四个维度"这类填空；② 记住"持续测试是**组织能力**"这个判断，论述题可以引用；③ 别把白皮书当成标准（它不是强制规范），表述时说"实践框架／参考模型"更准确。

**过渡：**第 5 章到这里讲完了。我们用一页把整章收口。

---

## 【5.95】本章小结（3 分钟）

这一页是本章的收口，六条，我按"从静态到动态、从零件到系统"的顺序重讲一遍，你对着自己的笔记查漏。

**第一条，代码规范与代码评审的价值**。要深刻理解它，并且**持续做**——方式是**互为评审、集体评审**。为什么强调"互为"？因为单向评审会退化成"资深的人挑新人的错"，而互为评审是**团队共同建立标准**。课件说"团队有多方面的收益"，这个收益不只是缺陷变少，还包括知识共享和标准统一。

**第二条，代码分析工具的使用**。课件说"今天很流行"，两个落点：**开发本地扫描**（写的时候就发现问题，反馈最快）和**与 CI/CD 流水线集成**（保证每次提交都被扫）。这两条要同时有，只有本地扫描靠自觉，只有流水线扫描反馈太慢。

**第三条，深刻理解单元测试（UT）的价值和 xUnit 框架的实现机制**。这里要注意"实现机制"四个字——JUnit 不是一个魔法，它就是**注解 + 反射 + 断言 + 运行器**这套东西。理解了机制，你换任何语言的 xUnit 都能上手。

**第四条，掌握 JUnit 这类工具和 mock 工具的使用技巧**。技巧的关键在**取舍**：什么时候用 Stub、什么时候用 Mock、什么时候干脆不 Mock。记住那句原则——**够用就行，Mock 越多离真实越远**。

**第五条，单元测试是否充分，需要衡量代码语句／分支覆盖率，一般用覆盖率工具自动分析**。注意这句话的两个限定：**语句和分支**（最常用的两个维度）、**自动分析**（人算不出来）。而覆盖率数字要看，但不能只看数字。

**第六条，今天提倡持续测试以促进持续交付；契约测试值得关注，从开发自测开始，比较彻底地保证质量**。这一条把整章的技术和流程接在了一起：**契约测试让你在开发阶段就能验证跨服务约定**，这是"从开发自测开始"的真正含义。

六条合起来是一句话：**先用评审和静态分析在不运行代码时抓缺陷，再用单元测试把每个函数钉死，用覆盖率衡量够不够，用契约测试守住服务边界，最后靠持续集成让这一切每次都发生。**

> **板书：**竖排六条关键词：① 规范与评审（互为／集体，持续做）② 分析工具（本地＋流水线）③ UT 价值与 xUnit 机制 ④ JUnit／Mock 技巧（够用就行）⑤ 覆盖率（语句／分支，自动分析）⑥ 持续测试＋契约测试。底部收口一句："评审→单测→覆盖率→契约→持续集成"。

**提问：**这六条里，如果只能保留三条，你保留哪三条？说明理由。

**预设回答：**

- 答"评审、单元测试、持续集成"——很稳的组合：一个管"不运行就能发现"，一个管"零件正确"，一个管"每次都发生"。

- 答"覆盖率、契约测试、持续集成"——也成立，但要指出：覆盖率是**度量**，它自己不产生质量，得靠单测支撑。

- 追问：为什么"持续集成"几乎所有人都会保留？答：因为它是**机制**——前五条都是"能力"，只有它是"让能力持续发生"的那一环。

**易错点 / 考点：**① 六条要能默写关键词，简答题常要求"简述本章主要内容"；② 覆盖率那条的两个限定（语

句／分支、自动分析）别漏；③ 契约测试在本章的地位要摆对：**它是第 5 章的新增重点**，也是与第 4 章流程衔接的落点；④ 论述题可用"评审→单测→覆盖率→契约→持续集成"这条主线组织答案。

**【思政落点 14】**：这一章讲的东西都有点"笨"：评审要一行行看，单元测试要一个个写，覆盖率要一格一格补，契约要一条条对。它们不像"上个自动化平台"那样有成就感，可**系统的可靠性恰恰是从这些笨功夫里长出来的**。工程质量没有捷径，只有把基础做扎实的耐心。希望同学们愿意在没人看见的地方多写一个边界用例、多补一个异常分支——**这些不被表扬的部分，才是你专业性的证据**。

**过渡：**理论讲完了，接下来是动手。实验 2 要求你用 JUnit 把一个 Spring 控制器测出来，我们把要求说清楚。

---

## 【5.96】实验 2：单元测试（3 分钟）

实验 2 的任务一句话就能说完：**使用 JUnit 工具，针对 Spring Unit Testing 控制器代码中的 `ItemController` 类进行测试，编写对应的测试类完成单元测试，最终提交测试代码**。但我把要求拆细，否则你会发现"能跑"和"合格"差得很远。

**第一，被测对象是谁。**`ItemController` 是一个 Spring MVC 的控制器，它的职责是**接收 HTTP 请求、调用 Service、返回响应**。所以你要测的不是"业务算得对不对"（那是 Service 的事），而是**控制器这一层的三件事**：请求参数能不能正确绑定、Service 返回什么它就返回什么、异常和错误码处理对不对。

**第二，怎么隔离依赖。**控制器依赖 Service，Service 可能又依赖数据库。**不能真连数据库**——那样就不是单元测试了。两种常见做法：一是用 `@WebMvcTest` 只加载 Web 层，把 Service 用 `@MockBean` 换掉；二是不启 Spring 容器，直接 `new ItemController(mockService)`，用 Mockito 造 Service，再直接调用方法。**两种都可以，但你必须在实验报告里说清你选了哪种、为什么**。

**第三，用例要覆盖哪些场景。**我给你一个最小清单，你要在此基础上自己加：**正常查询**（有数据／无数据）、**新增**（合法参数／缺必填参数／参数类型不合法）、**修改与删除**（存在／不存在）、**异常路径**（Service 抛异常时控制器返回什么）。用 AAA 结构写：准备（Arrange）、执行（Act）、断言（Assert），**一个用例只验一个行为**。

**第四，断言要硬。**不要只 `assertNotNull`。要断言**状态码、响应体的关键字段、Service 被调用的次数和参数**（`verify`）。这是"行为验证"，也是本实验的评分点之一。

**第五，提交物。**至少包含：测试类源码（放在 `src/test/java/.../ItemControllerTest.java`）、一份覆盖率报告截图（说明分支覆盖情况）、一份简短的实验说明（测了哪些场景、哪些没测、为什么没测）。

最后提醒三条纪律：**测试方法命名要能读懂意图**（比如 `findById_whenNotFound_returns404`）；**测试之间不能有依赖**，任意顺序都能跑；**不许改被测代码来让测试通过**——如果确实发现被测代码有问题，写进报告，别偷偷改。

**板书：**五行：① 被测＝`ItemController`（参数绑定／响应／异常处理），不测 Service 业务 ② 隔离＝`@WebMvcTest`＋`@MockBean` 或 `new Controller(mockService)` ③ 场景清单：查（有／无）·增（合法／缺参／类型错）·改删（存在／不存在）·异常 ④ 断言：状态码＋响应字段＋`verify` 调用 ⑤ 提交：测试类＋覆盖率截图＋说明。底部三条纪律：命名可读／用例独立／不改被测代码。

**提问：**`ItemController` 里有个方法直接返回了 `null` 而没有返回 404，测试该怎么写才算"测出来了"？

**预设回答：**

- 答"断言返回 null"——那只是**记录现状**，不是测出问题：测试会把错误行为固化下来，以后谁修好了反而测试红。

- 答"断言状态码应该是 404，现在不是，所以测试红"——这才是正确做法：**测试写的是"应该怎样"，不是"现在怎样"**。红了正好证明缺陷存在。

- 追问：那测试红了，实验算失败吗？答：不算——**能用一个失败的用例精准指出缺陷位置，比一堆绿的用例更有价值**，把它写进报告。

**易错点 / 考点：**① 实验最常见的失分是"**测了 Service 的业务逻辑**"，那不是本实验的目标；② 第二常见的是"**断言太软**"（只判非空、只判不抛异常）；③ 用 `@WebMvcTest` 时容易踩的坑：它不加载 `@Service`，忘了加 `@MockBean` 就会启动失败；④ **不许为了通过测试修改被测代码**，这是纪律问题，也会直接影响评分。

**过渡：**想在这条路上继续深入，有几本书和两个官方站点值得你收藏。

---

## 【5.97】学习资源推荐（1 分钟）

这一页是资源清单，我按"读完能解决什么问题"给你排个序，你别一次全买。

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）。这本适合**现在读**——它讲的正是"开发人员自己怎么写测试"，和你这学期做的事完全对口，读完你的单元测试水平会有明显变化。

第二本，**《敏捷测试》**（人民邮电出版社，2021 年 8 月）。这本适合**做完实验之后读**，它讲的是团队层面怎么组织测试、怎么在快速迭代里保证质量，正好接上本章最后的持续测试。

第三本，**《Java 微服务测试》**（电子工业出版社，2019 年 7 月）。这本是**进阶**，讲微服务下怎么测：契约、消费者驱动、测试金字塔在微服务里的变形。你如果要做微服务方向的项目，这本值得细读。

两个站点更重要，因为它们是**活的**：**junit.org/junit5** 是 JUnit 5 的官方文档，你实验里遇到的注解、断言、扩展模型，全在里面，比任何二手教程都准；**docs.pact.io** 是 Pact 的官方文档，契约测试那一套流程和代码示例都在上面。

再说说怎么用这几份材料。**实验卡住的时候查官方文档**，因为报错信息、注解用法、版本差异都在那儿；**想系统补基础的时候读书**，因为书有章节顺序和上下文；**想找方向的时候看白皮书和案例**，因为它们告诉你行业在往哪走。三种材料对应三种需求，别指望一本书解决所有问题。

我给你的建议是：**先读官方文档，再读书**。官方文档更新快、写得准，缺点是它假设你已经知道要解决什么问题——所以配合教材和这几本书打底，效果最好。

> **板书：**两栏。左"书"：《程序开发人员测试指南》→ 现在读；《敏捷测试》→ 实验后读；《Java 微服务测试》→ 进阶。右"站点"：**junit.org/junit5**（注解・断言・扩展）、**docs.pact.io**（契约流程・示例）。底部一句："先读官方文档，再读书。"

**提问：**如果只有一周时间，你在这三本书和两个站点之间怎么分配？

**预设回答：**

- 答"先看 JUnit 文档，边做实验边查"——最实用，实验驱动学习效率最高。

- 答"读第一本书的前几章"——也可以，但要配套动手，光读不写记不住。

- 追问：为什么不建议先读微服务那本？答：**没有单元测试和契约的基础，微服务测试那一套读起来是空中楼阁**——顺序很重要。

**易错点 / 考点：**① 这一页一般只作了解，考法最多是"推荐一本单元测试相关的书"这类开放题；② 书名、出版社、年份不必死记，但**两个官方站点要记住**（实验和后续作业都要用）；③ 学习路径的判断（先文档后书、

先基础后微服务）比书目本身更值得记。

**过渡：**这一章就到这里。最后，我把这 90 分钟想留下的东西，用几句话收一下。

---

## 【5.98】感 谢 聆 听（1 分钟）

这一章我们从"代码这一层怎么测"出发，走完了评审、静态分析、单元测试、Mock、覆盖率、集成策略、契约测试、持续集成这一整条线。如果 90 分钟之后你只记住三句话，我希望是这三句。

第一句：**测试不是编码之后的补救，是编码过程的一部分**。评审、静态扫描、单元测试，这些都能在代码还没跑起来的时候就发现问题，越早越便宜。

第二句：**覆盖率是镜子，不是奖状**。它告诉你哪里没人走过，不告诉你"够了"。真正的标准是人定的，工具只负责把事实摆出来。

第三句：**让好习惯每次自动发生**。会写测试不难，难的是每次提交都跑、每次改动都验。把这件事交给流水线，别交给自觉。

下一章我们会继续往系统层面走，看性能、安全这些"跑起来之后才暴露"的问题。课下请把实验 2 的 `ItemController` 测出来——**写测试这件事，只有动手才算学会**。

谢谢大家。

**板书：**三行大字：① 测试在编码中，不在编码后 ② 覆盖率是镜子，不是奖状 ③ 让好习惯每次自动发生。下方一行小字："实验 2：`ItemController` 单元测试，提交测试代码。"

**提问：**这一章里，哪一个知识点你认为最可能在你的第一个项目里立刻用上？为什么？

**预设回答：**

- 答"单元测试和 Mock"——最直接，进项目第一周就会写。

- 答"代码评审的检查表"——也很实用，评审是最快能参与的团队活动。

- 答"契约测试"——如果进的是微服务团队，这个最值钱，但需要团队配合才能落地。

- 收口：三种答案都对，取决于你的岗位；但**如果只能先练一样，练单元测试**——它是所有其他实践的基础。

**易错点 / 考点：**无考点。这一页的作用是**收束全章、指向下一章、明确课下任务**，讲的时候不要拖，一分钟内

结束，把时间留给学生提问。

**过渡：**本章结束。下一次课进入第 6 章，我们从"代码正确"转向"系统在压力下还正确吗"。

---

## 四、思政落点清单（共 14 处）

| 序号 | 所在页 | 落点主题 |
| --- | --- | --- |
| 1 | 5.6 | 代码评审最珍贵的地方，是它把"我的代码"变成"我们的代码"。写代码的人天然有防御 |
| 2 | 5.9 | 这几条指标背后是一种很朴素的职业态度：认真是有成本的，所以要给它留出成本。"帮我 |
| 3 | 5.28 | 第三方库这条链，给工科生上了一堂很现实的责任课。你引一个包，可能只是图省事；但这 |
| 4 | 5.33 | 这六项排下来，你会发现单元测试是件"不显眼"的活：没有界面、没有演示效果、写完了 |
| 5 | 5.34 | 这两个算式其实是成本意识的数学表达。工程上最贵的一句话是"先这样，回头再说"—— |
| 6 | 5.36 | 覆盖率这个数字，最考验一个人的诚实。因为它太好看了：写一条把整个方法跑一遍的用例 |
| 7 | 5.39 | 驱动和桩，本质是为协作搭桥。被测单元本身没变，是周围的环境被"补"出来了——上游 |
| 8 | 5.48 | 注解看起来只是几个`@`符号，但它体现的是规范意识。`@Test`让框架找得到你 |
| 9 | 5.60 | 这句"够用就行"是难得的务实精神教科书。技术圈有个通病——用什么工具代表水平：写 |
| 10 | 5.64 | `verify()`这个 API 的名字值得琢磨——它叫"验证"，不叫"检查过了就行 |
| 11 | 5.69 | 覆盖率最容易被用错的地方，是把它当成交差的凭证——数字到 80%就收工，报告一贴， |
| 12 | 5.82 | 契约这两个字，在工程上和生活里是同一个意思——我说到做到，你按我说的准备。契约被 |
| 13 | 5.92 | 质量门禁最有意思的地方是它不商量：单元测试失败就是失败，覆盖率不达标就是不放行， |
| 14 | 5.95 | 这一章讲的东西都有点"笨"：评审要一行行看，单元测试要一个个写，覆盖率要一格一格 |