第 22 章

Software Engineering: A Practitioner's Approach, Eighth Edition

软件测试策略

要点浏览

概念：软件测试的目的是为了发现软件设计和实现过程中因疏忽所造成的错误。但是，如何进行测试？是否应该制定正式的测试计划？应该将整个程序作为一个整体来测试，还是应该只测试其中的一小部分？当向一个大型系统加入新的构件时，对于已经做过的测试，是否还要重新测试？什么时候需要客户参与测试工作？在制定测试策略时，就需要回答上述问题以及一些其他问题。

人员：软件测试策略由项目经理、软件工程师及测试专家来制定。

重要性：测试所花费的工作量经常比其他任何软件工程活动都多。若测试是无计划进行的，则既浪费时间，又浪费不必要的劳动。甚至更糟的是，错误未被测出，因而隐蔽下来。因此，为测试软件建立系统化的测试策略是合情合理的。

步骤：测试从“小范围”开始，并逐步过渡到“软件整体”。这意味着，早期的测试关注单个构件或相关的一小组构件，利用测试发现封装在构件中的数据错误和处理逻辑错误。当完成单个构件的测试后，需要将构件集成起来，直到建成整个系统。这时，执行一系列的高阶测试以发现不满足客户需求的错误。随着错误的发现，必须利用调试过程对错误进行诊断和纠正。

工作产品：测试规格说明是将软件测试团队的具体测试方法文档化。这主要包括制定描述整体策略的计划，定义特定测试步骤的规程以及将要进行测试的类型。

质量保证措施：在测试进行之前通过对测试规格说明进行评审，可以评估测试用例及测试任务的完整性。有效的测试计划和规程可以引导团队有秩序地构建软件，并且在构建过程中能够发现各个阶段引入的错误。

软件测试策略提供了一张路线图：描述将要进行的测试步骤，包括这些步骤的计划和执行时机，以及需要的工作量、时间和资源。因此，任何测试策略都必须包含测试计划、测试用例设计、测试执行以及测试结果数据的收集与评估。

软件测试策略应该具有足够的灵活性，以促进测试方法的定制；同时又必须足够严格，以支持在项目进行过程中对项目进行合理策划和追踪管理。Shooman[SHO83] 对这些问题进行了讨论：

从许多方面来看，测试都是一个独立的过程，并且同软件开发方法一样，测试类型也有很多种。多年以来，对付程序出错的唯一武器就是谨慎的设计和程序员的个人智慧。目前，有很多现代设计技术（和正式技术评

关键概念

α 测试
β 测试
自底向上集成
类测试
簇测试
完成
配置评审
调试
部署测试
驱动
独立测试组
集成测试

审）可以帮助我们减少代码中内在的初始错误。类似地，不同的测试方法正在逐渐聚合成几种不同的途径和思想。

这些途径和思想就是我们所谓的策略。本章讨论软件测试策略，本书的第 23 ～ 26 章介绍实施测试策略的测试方法和测试技术。

关键概念
面向对象软件
性能测试
恢复测试
回归测试
安全测试
冒烟测试
压力测试
桩
系统测试
移动 App 的测试策略
WebApp 的测试策略
基于线程的测试
自顶向下集成
单元测试确认
确认测试
验证

22.1　软件测试的策略性方法

测试是可以事先计划并可以系统地进行的一系列活动。因此，应该为软件过程定义软件测试模板，即将特定的测试用例设计技术和测试方法放
466 到一系列的测试步骤中去。

文献中已经提出了许多软件测试策略。这些策略为软件开发人员提供了测试模板，且具备下述一般特征：

- 为完成有效的测试，应该进行有效的、正式的技术评审（第 20 章）。通过评审，许多错误可以在测试开始之前排除。
- 测试开始于构件层，然后向外“延伸”到整个基于计算机系统的集成中。
- 不同的测试技术适用于不同的软件工程方法和不同的时间点。
- 测试由软件开发人员和（对大型项目而言）独立的测试组执行。
- 测试和调试是不同的活动，但任何测试策略都必须包括调试。

网络资源 有关软件测试的有用资料可在 www.mtsu.edu/~storm/ 找到。

软件测试策略必须提供必要的低级测试，可以验证小段源代码是否正确实现，也要提供高级测试，用来确认系统的主要功能是否满足用户需求。软件测试策略必须为专业人员提供工作指南，同时，为管理者提供一系列的里程碑。由于测试策略的步骤往往是在软件完成的最后期限的压力开始呈现时才刚刚进行的，因此，测试的进度必须是可测量的，并且应该
467 让问题尽可能早地暴露。

22.1.1　验证与确认

软件测试是通常所讲的更为广泛的主题——验证与确认（Verification and Validation，V&V）的一部分。验证是指确保软件正确地实现某一特定功能的一系列活动，而确认指的是确保开发的软件可追溯到客户需求的另外一系列活动。Boehm[BOE81] 用另一种方式说明了这两者的区别：

验证：“我们在正确地构建产品吗？”

确认：“我们在构建正确的产品吗？”

验证与确认的定义包含很多软件质量保证活动（第 21 章）⊖。

引述 测试是开发软件系统过程中每项可靠的工作都不可避免的部分。
William Howden

验证与确认包含广泛的 SQA 活动：正式技术评审、质量和配置审核、性能监控、仿真、可行性研究、文档评审、数据库评审、算法分析、开发测试、易用性测试、合格性测试、验收测试和安装测试。虽然测试在验证

⊖ 应该注意到，哪些类型的测试构成“确认”，然而人们对此观点存在极大的分歧。一些人认为所有的测试都是验证，而确认是在对需求进行评审和认可时进行的，也许更晚一些——是在系统投入运行时由用户进行的。另外一些人将单元测试和集成测试（22.3.1 节和 22.3.2 节）看成验证，而将高阶测试（22.6 节和 22.7 节）看成确认。

与确认中起到了非常重要的作用，但很多其他活动也是必不可少的。

测试确实为软件质量的评估（更实际地说是错误的发现）提供了最后的堡垒。但是，测试不应当被看作安全网。正如人们所说的那样："你不能测试质量。如果开始测试之前质量不佳，那么当你完成测试时质量仍然不佳。"在软件工程的整个过程中，质量已经被包含在软件之中了。方法和工具的正确运用、有效的正式技术评审、坚持不懈的管理与测量，这些都形成了在测试过程中所确认的质量。

建议 不要轻易地将测试看成是一个安全网，认为它能捕捉由不良的软件工程实践引发的所有错误。应该在整个软件过程中注重质量和错误检测。

Miller[MIL77] 将软件测试和质量保证联系在一起，他认为："无论是大规模系统还是小规模系统，程序测试的根本动机都是使用经济且有效的方法来确认软件质量。"

22.1.2 软件测试组织

对每个软件项目而言，在测试开始时就会存在固有的利害关系冲突。要求开发软件的人员对该软件进行测试，这本身似乎是没有恶意的！毕竟，谁能比开发者本人更了解程序呢？遗憾的是，这些开发人员感兴趣的是急于显示他们所开发的程序是无错误的，是按照客户的需求开发的，而且能按照预定的进度和预算完成。这些利害关系会影响软件的充分测试。

引述 乐观主义是编程的职业障碍；测试是治疗良方。
Kent Beck

468

从心理学的观点来看，软件分析和设计（连同编码）是建设性的任务。软件工程师分析、建模，然后编写计算机程序及其文档。与其他任何建设者一样，软件工程师也为自己的"大厦"感到骄傲，而蔑视企图拆掉大厦的任何人。当测试开始时，有一种微妙的但确实存在的企图，即试图摧毁软件工程师所建造的大厦。以开发者的观点来看，可以认为（心理学上）测试是破坏性的。因此，开发者精心地设计和执行测试，试图证明其程序的正确性，而不是注意发现错误。遗憾的是，错误仍然是存在的，而且，即使软件工程师没有找到错误，客户也会发现它们。

上述讨论通常会使人们产生下面的误解：（1）软件开发人员根本不应该做测试；（2）应当让那些无情地爱挑毛病的陌生人做软件测试；（3）测试人员仅在测试步骤即将开始时参与项目。这些想法都是不正确的。

软件开发人员总是要负责程序各个单元（构件）的测试，确保每个单元完成其功能或展示所设计的行为。在多数情况下，开发者也进行集成测试。集成测试是一个测试步骤，它将给出整个软件体系结构的构建（和测试）。只有在软件体系结构完成后，独立测试组才开始介入。

独立测试组（Independent Test Group，ITG）的作用是为了避免开发人员进行测试所引发的固有问题。独立测试可以消除利益冲突。独立测试组的成员毕竟是依靠找错误来获得报酬的。

关键点 独立测试组没有软件开发者可能经历的"利益冲突"。

然而，软件开发人员并不是将程序交给独立测试组就可以一走了之。在整个软件项目中，开发人员和测试组要密切配合，以确保进行充分的测试。在测试进行的过程中，必须随时可以找到开发人员，以便及时修改发现的错误。

引述 人们犯的第一个错误是认为测试团队负责保证质量。
Brian Marick

从分析与设计到计划和制定测试规程，ITG 参与整个项目过程。从这种意义上讲，ITG 是软件开发项目团队的一部分。然而，在很多情况下，

ITG 直接向软件质量保证组织报告，由此获得一定程度的独立性。如果 ITG 是软件工程团队的一部分，那么这种独立性将是不可能获得的。

22.1.3　软件测试策略——宏观

可以将软件过程看作图 22-1 所示的螺旋。开始时系统工程定义软件的角色，从而引出软件需求分析，在需求分析中建立了软件的信息域、功能、行为、性能、约束和确认标准。
469 沿着螺旋向内，经过设计阶段，最后到达编码阶段。为开发计算机软件，沿着流线螺旋前进，每走一圈都会降低软件的抽象层次。

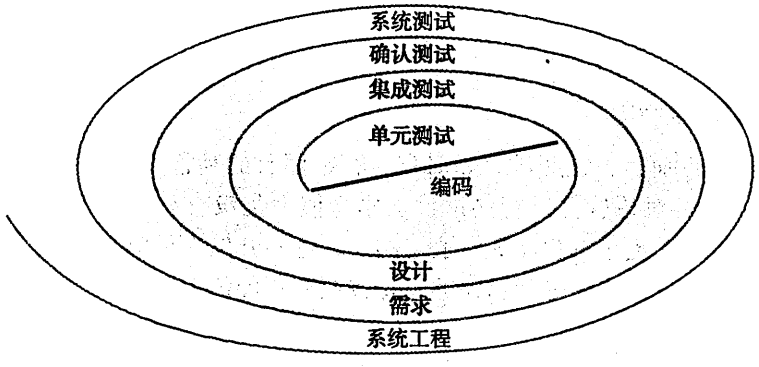

图 22-1　测试策略

软件测试策略也可以放在螺旋模型中来考虑（图 22-1）。*单元测试*起始于螺旋的旋涡中心，侧重于以源代码形式实现的每个单元（例如，构件、类或 WebApp 内容对象）。沿着螺旋向外就是*集成测试*，这时的测试重点在于软件体系结构的设计和构建。沿着螺旋向外再走一圈就是*确认测试*，在这个阶段，依据已经建立的软件，对需求（作为软件需求建模的一部分而建立）进行确认。最后到达*系统测试*阶段，将软件与系统的其他成分作为一个整体来测试。为了测试计算机软件，沿着流线向外螺旋前进，每转一圈都拓宽了测试范围。

提问　什么是软件测试的总体策略？

以过程的观点考虑整个测试过程，软件工程环境中的测试实际上就是按顺序实现四个步骤，如图 22-2 所示。最初，测试侧重于单个构件，确保它起到了单元的作用，因此称之为*单元测试*。单元测试充分利用测试技术，运行构件中每个控制结构的特定路径，以确保路径的完全覆盖，并最大可能地发现错误。接下来，组装或集成各个构件以形成完整的软件包。*集成测试*处理并验证与程序构建相关的问题。在集成过程中，普遍使用关注输入和输出的测试用例设计技术（尽管也使用检验特定程序路径的测试用例设计技术来保证主要控制路径的覆盖）。在软件集成（构建）完成之后，要执行一系列的高阶测试。必须评估确认准则（需求
470 分析阶段建立的）。*确认测试*为软件满足所有的功能、行为和性能需求提供最终保证。

网络资源　有关软件测试人员的有用资源可在站点 www.SQAtster.com 中找到。

最后的高阶测试步骤已经超出软件工程的边界，属于更为广泛的计算机系统工程范围。软件一旦确认，就必须与其他系统成分（如硬件、人、数据库）结合在一起。系统测试验证所有成分都能很好地结合在一起，且能满足整个系统的功能或性能需求。

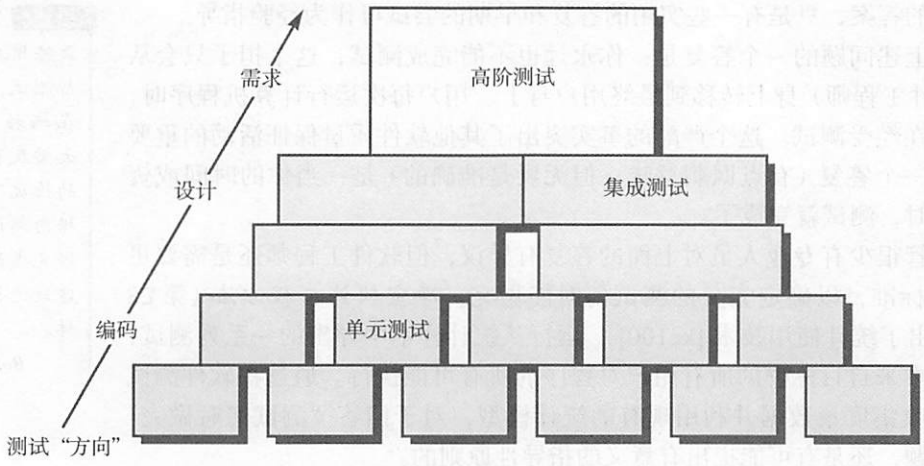

图 22-2　软件测试步骤

SafeHome　准备测试

[场景] Doug Miller 的办公室，继续构件级设计，并开始特定构件的构建。

[人物] Doug Miller，软件工程经理；Vinod、Jamie、Ed 和 Shakira，SafeHome 软件工程团队成员。

[对话]

Doug: 在我看来，我们似乎是没有花费太多的时间讨论测试。

Vinod：对，但我们都有点忙。另外，我们一直在考虑这个问题……实际上，远不止考虑。

Doug（微笑）: 我知道，大家都在超负荷地工作，不过我们还得全面考虑。

Shakira：在对构件开始编码之前，我喜欢设计单元测试，因此，那是我一直尽力去做的。我有一个相当大的测试文件，一旦完成了构件编码工作，就运行这个测试文件。

Doug: 那是极限编程（敏捷软件开发过程，见第 5 章）概念，不是吗？

Ed：是的。尽管我没有亲自使用极限编程，但可以肯定，在建立构件之前设计单元测试是个好主意，这种单元测试的设计会给我们提供所需要的所有信息。

Jamie：我一直在做这件事情。

Vinod：我负责集成，因此，每当别人将构件传给我，我就将其集成到部分已集成的程序中，并运行一系列的集成测试。我一直忙于为系统中的每个功能设计适当的测试集。

Doug（对 Vinod）：你多长时间运行一次测试？

Vinod: 每天……直到系统被集成……嗯，直到我们计划交付的软件增量被集成。

Doug：你们已经走在我前面了。

Vinod（大笑）：在软件业务中，抢先就是一切，老板。

471

22.1.4　测试完成的标准

提问　我们什么时候完成测试？

每当讨论软件测试时，就会引出一个典型的问题：“测试什么时候才算做完？怎么知道我们已做了足够的测试？”非常遗憾的是，这个问题没

有确定的答案，只是有一些实用的答复和早期的尝试可作为经验指导。

对上述问题的一个答复是：你永远也不能完成测试，这个担子只会从你（软件工程师）身上转移到最终用户身上。用户每次运行计算机程序时，程序就在经受测试。这个严酷的事实突出了其他软件质量保证活动的重要性。另一个答复（有点讽刺意味，但无疑是准确的）是：当你的时间或资金耗尽时，测试就完成了。

尽管很少有专业人员对上面的答复有异议，但软件工程师还是需要更严格的标准，以确定充分的测试何时能做完。*净室软件工程方法*（第28章）提出了统计使用技术 [ke100]：运行从统计样本中导出的一系列测试，统计样本来自目标群的所有用户对程序的所有可能执行。通过在软件测试过程中收集度量数据并利用现有的统计模型，对于回答“测试何时做完”这种问题，还是有可能提出有意义的指导性原则的。

引述 仅仅针对最终用户需求进行测试，就如同在牺牲了地基、大梁及管道工程的情况下，只根据内部设计师已经完成的工作对建筑进行检查一样。

Boris Beizer

22.2 策略问题

本章的后面几节介绍系统化的软件测试策略。然而，如果忽视了一些重要问题，即使最好的策略也会失败。Tom Gilb[GIL95] 提出，只有软件测试人员解决了下述问题，软件测试策略才会获得成功：（1）早在开始测试之前，就要以量化的方式规定产品需求；（2）明确地陈述测试目标；（3）了解软件的用户并为每类用户建立用户描述；（4）制定强调“快速周期测试”的测试计划；[⊖]（5）建立能够测试自身的“健壮”软件（防错技术在22.3.1节讨论）；（6）测试之前，利用有效的正式技术评审作为过滤器；（7）实施正式技术评审以评估测试策略和测试用例本身；（8）为测试过程建立一种持续的改进方法（第37章）。

472

提问 什么样的指导原则使软件测试策略获得成功？

网络资源 相当好的测试资源列表可在 www.SQAtester.com 找到。

22.3 传统软件的测试策略[⊜]

许多策略可用于测试软件。其中的一个极端是，软件团队等到系统完全建成后再对整个系统执行测试，以期望发现错误。虽然这种方法很有吸引力，但效果不好，可能得到的是有许多缺陷的软件，致使所有的利益相关者感到失望。另一个极端是，无论系统的任何一部分在何时建成，软件工程师每天都在进行测试。

多数软件团队选择介于这两者之间的测试策略。这种策略以渐进的观点对待测试，以个别程序单元的测试为起点，逐步转移到方便于单元集成的测试（有的时候每天都进行测试），最后以实施整个系统的测试而告终。下面几节将对这几种不同的测试进行描述。

建议 在为构件开发代码之前就设计单元测试用例是个不错的想法，有助于确保开发的代码能够通过测试。

22.3.1 单元测试

单元测试侧重于软件设计的最小单元（软件构件或模块）的验证工作。利用构件级设计

⊖ Gilb[Gil95] 建议软件团队学习对客户可用性进行快速周期测试（项目工作的2%），至少在“可试验性”方面增强功能性或提高质量。从这些快速周期测试得到的反馈可以用于控制质量级别及相应的测试策略。

⊜ 本书使用术语常规软件和传统软件来表示在多种应用领域中经常碰到的普通分层软件体系结构或调用－返回软件体系结构。传统的软件体系结构不是面向对象的，也不包括 WebApp 或移动 App。

描述作为指南，测试重要的控制路径以发现模块内的错误。测试的相对复杂度和这类测试发现的错误受到单元测试约束范围的限制。单元测试侧重于构件的内部处理逻辑和数据结构。这种类型的测试可以对多个构件并行执行。

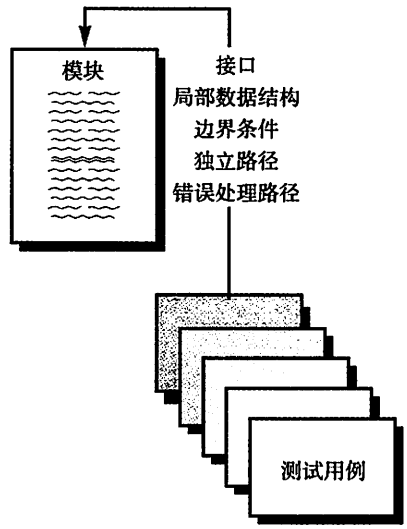

图 22-3　单元测试

单元测试问题。图 22-3 对单元测试进行了概要描述。测试模块的接口是为了保证被测程序单元的信息能够正常地流入和流出；检查局部数据结构以确保临时存储的数据在算法的整个执行过程中能维持其完整性；执行控制结构中的所有独立路径（基本路径）以确保模块中的所有语句至少执行一次；测试边界条件确保模块在到达边界值的极限或受限处理的情形下仍能正确执行。最后，要对所有的错误处理路径进行测试。

对穿越模块接口的数据流的测试要在任何其他测试开始之前进行。若数据不能正确地输入 / 输出，则其他测试都是没有意义的。另外，应当测试局部数据结构，可能的话，在单元测试期间确定对全局数据的局部影响。

473

在单元测试期间，选择测试的执行路径是最基本的任务。设计测试用例是为了发现因错误计算、不正确的比较或不适当的控制流而引起的错误。

提问　单元测试期间常发现的错误是什么？

边界测试是最重要的单元测试任务之一。软件通常在边界处出错，也就是说，错误行为往往出现在处理 n 维数组的第 n 个元素，或者 i 次循环的第 i 次调用，或者遇到允许出现的最大、最小数值时。使用刚好小于、等于或大于最大值和最小值的数据结构、控制流和数值作为测试用例就很有可能发现错误。

建议　确信已经设计了执行每个异常处理路径的测试。若没有，当执行这样的路径时就可能失败，从而加重了危险的形势。

好的设计要求能够预置出错条件并设置异常处理路径，以便当错误确实出现时重新确定路径或彻底中断处理。Yourdon[YOU75] 称这种方法为*防错法*（antibugging）。遗憾的是，存在的一种趋势是在软件中引入异常处理，然而却从未对其进行测试。如果已经实现了错误处理路径，就一定要对其进行测试。

在评估异常处理时，应能测试下述的潜在错误：（1）错误描述难以理解；（2）记录的错误与真正遇到的错误不一致；（3）在异常处理之前，错误条件就引起了操作系统的干预；（4）异常条件处理不正确；（5）错误描述没有提供足够的信息，对确定错误产生原因没有帮助。

单元测试过程。单元测试通常被认为是编码阶段的附属工作。可以在编码开始之前或源代码生成之后进行单元测试的设计。设计信息的评审可以指导建立测试用例，发现前面所讨论的各类错误，每个测试用例都应与一组预期结果联系在一起。

474

建议　在没有资源做全面测试的情况下，只选择关键模块和高复杂性模块做单元测试。

由于构件并不是独立的程序，因此，必须为每个测试单元开发驱动程序和桩程序。单元测试环境如图 22-4 所示。在大多数应用中，驱动程序只是一个“主程序”，它接收测试用例数据，将这些数据传递给（将要测试的）构件，并打印相关结果。桩程序的作用是替换那些从属于被测构件（或被其调用）的模块。桩程序或“伪程序”使用从属模块的接口，可能做少量

的数据操作，提供入口的验证，并将控制返回到被测模块。

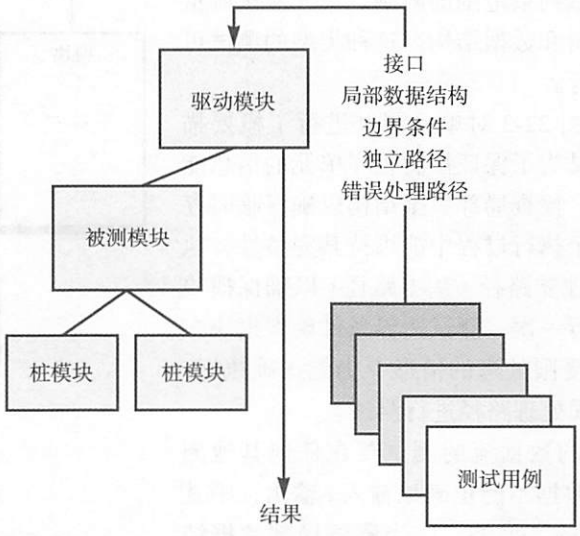

图 22-4　单元测试环境

驱动程序和桩程序都意味着测试开销。也就是说，两者都必须编写代码（通常并没有使用正式的设计），但并不与最终的软件产品一起交付。若驱动程序和桩程序保持简单，实际开销就会比较低。遗憾的是，在只使用“简单”的驱动程序和桩程序的情况下，许多构件是不能完成充分的单元测试的，因此，完整的测试可以延迟到集成测试这一步（这里也要使用驱动程序和桩程序）。

22.3.2　集成测试

软件界的初学者一旦完成所有模块的单元测试之后，可能会问一个似乎很合理的问题：如果每个模块都能单独工作得很好，那么为什么要怀疑将它们放在一起时的工作情况呢？当然，这个问题涉及“将它们放在一起”的接口连接。数据可能在穿过接口时丢失；一个模块可能对另一个模块产生负面影响；子功能联合在一起并不能达到预期的功能；单个模块中可以接受的不精确性在连接起来之后可能会扩大到无法接受的程度；全局数据结构可能产生问题。遗憾的是，问题还远不止这些。

建议　采取一步到位的集成方法是一种懒惰的策略，注定会失败。在进行测试时，应该采用增量集成。

475

集成测试是构建软件体系结构的系统化技术，同时也是进行一些旨在发现与接口相关的错误的测试。其目标是利用已通过单元测试的构件建立设计中描述的程序结构。

常常存在一种非增量集成的倾向，即利用“一步到位”的方式来构造程序。所有的构件都事先连接在一起，全部程序作为一个整体进行测试。结果往往是一片混乱！会出现一大堆错误。由于在整个程序的广阔区域中分离出错的原因是非常复杂的，因此改正错误也会比较困难。

增量集成与“一步到位”的集成方法相反。程序以小增量的方式逐步进行构建和测试，这样错误易于分离和纠正，更易于对接口进行彻底测试，而且可以运用系统化的测试方法。下面将讨论一些不同的增量集成策略。

自顶向下集成。自顶向下集成测试是一种构建软件体系结构的增量方法。模块的集成顺

序为从主控模块（主程序）开始，沿着控制层次逐步向下，以深度优先或广度优先的方式将从属于（和间接从属于）主控模块的模块集成到结构中去。

建议　制定项目进度时，必须考虑将要采取的集成方式，使得构件在需要时是可用的。

参见图 22-5，深度优先集成是首先集成位于程序结构中主控路径上的所有构件。主控路径的选择有一点武断，也可以根据特定应用的特征进行选择。例如，选择最左边的路径，首先集成构件 M_1、M_2 和 M_5。其次，集成 M_8 或 M_6（若 M_2 的正常运行是必需的），然后集成中间和右边控制路径上的构件。广度优先集成首先沿着水平方向，将属于同一层的构件集成起来。如图 22-5 中，首先将构件 M_2、M_3 和 M_4 集成起来，其次是下一个控制层 M_5、M_6，依此类推。集成过程可以通过下列 5 个步骤完成：

提问　自顶向下集成的步骤是什么？

1. 主控模块用作测试驱动模块，用直接从属于主控模块的所有模块代替桩模块。
2. 依靠所选择的集成方法（深度优先或广度优先），每次用实际模块替换一个从属桩模块。
3. 集成每个模块后都进行测试。
4. 在完成每个测试集之后，用实际模块替换另一个桩模块。
5. 可以执行回归测试（在本节的后面讨论）以确保没有引入新的错误。

回到第 2 步继续执行此过程，直到完成了整个程序结构的构建。

476

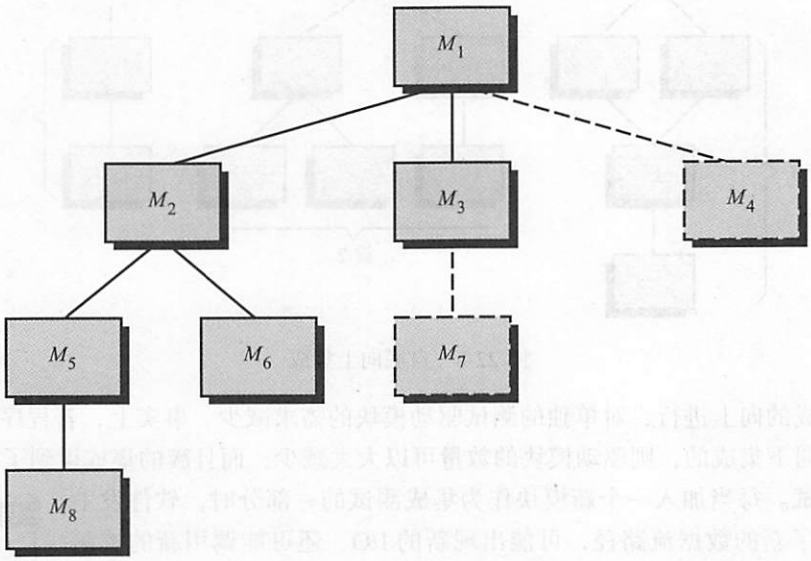

图 22-5　自顶向下集成

自顶向下集成策略是在测试过程的早期验证主要控制点或决策点。在能够很好分解的程序结构中，决策发生在层次结构的较高层，因此会首先遇到。如果主控问题确实存在，尽早地发现是有必要的。若选择了深度优先集成方法，可以实现和展示软件的某个完整功能。较早的功能展示可以增强所有开发者、投资者及用户的信心。

提问　选择自顶向下集成方法时，可能会遇到什么问题？

自底向上集成测试。自底向上集成测试，顾名思义，就是从原子模块（程序结构的最底层构件）开始进行构建和测试。由于构件是自底向上集成的，在处理时所需要的从属于给定层次的模块总是存在的，因此，没有必

要使用桩模块。自底向上集成策略可以利用以下步骤来实现：

提问　自底向上的集成步骤是什么？

1. 连接低层构件以构成完成特定子功能的簇（有时称为构造）。
2. 编写驱动模块（测试的控制程序）以协调测试用例的输入和输出。
3. 测试簇。

477　4. 去掉驱动程序，沿着程序结构向上逐步连接簇。

关键点　自底向上集成排除了对复杂桩的需要。

遵循这种模式的集成如图 22-6 所示。连接相应的构件形成簇 1、簇 2 和簇 3，利用驱动模块（图中的虚线框）对每个簇进行测试。簇 1 和簇 2 中的构件从属于模块 M_a，去掉驱动模块 D_1 和 D_2，将这两个簇直接与 M_a 相连。与之相类似，在簇 3 与 M_b 连接之前去掉驱动模块 D_3。最后将 M_a 和 M_b 与构件 M_c 连接在一起，依此类推。

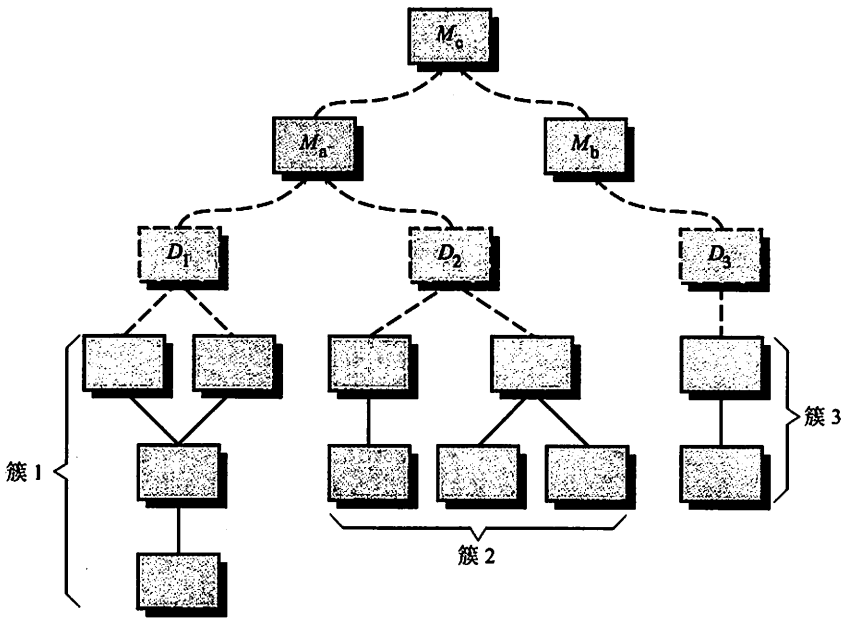

图 22-6　自底向上集成

随着集成的向上进行，对单独的测试驱动模块的需求减少。事实上，若程序结构的最上两层是自顶向下集成的，则驱动模块的数量可以大大减少，而且簇的集成得到了明显简化。

回归测试。每当加入一个新模块作为集成测试的一部分时，软件发生变更，建立了新的数据流路径，可能出现新的 I/O，还可能调用新的控制逻辑。这些变更所带来的副作用可能会使原来可以正常工作的功能产生问题。在集成测试策略的环境下，*回归测试*是重新执行已测试过的某些子集，以确保变更没有传播不期望的副作用。回归测试有助于保证变更（由于测试或其他原因）不引入无意识行为或额外的错误。

建议　回归测试是减少“副效应”的重要方法。每次对软件做重要变更时（包括新构件的集成），都要进行回归测试。

回归测试可以手工进行，方法是重新执行所有测试用例的子集，或者

478　利用捕捉 / 回放工具自动进行。*捕捉 / 回放工具*使软件工程师能够为后续的回放与比较捕捉测试用例和测试结果。*回归测试套件*（将要执行的测试子集）包含以下三种测试用例：

- 能够测试软件所有功能的具有代表性的测试样本。
- 额外测试，侧重于可能会受变更影响的软件功能。

- 侧重于已发生变更的软件构件测试。

随着集成测试的进行，回归测试的数量可能变得相当庞大，因此，应将回归测试套件设计成只包括涉及每个主要程序功能的一个或多个错误类的测试。

关键点 冒烟测试被称为是一种滚动的集成测试方法。每天对软件进行重构（加入新的构件），并进行冒烟测试。

冒烟测试。开发软件产品时，冒烟测试是一种常用的集成测试方法，是时间关键项目的决定性机制，允许软件团队频繁地对项目进行评估。大体上，冒烟测试方法包括下列活动。

1. 将已经转换为代码的软件构件集成到构造（build）中。一个构造包括所有的数据文件、库、可复用的模块以及实现一个或多个产品功能所需的工程化构件。
2. 设计一系列测试以暴露影响构造正确地完成其功能的错误。其目的是发现极有可能造成项目延迟的业务阻塞（show stopper）错误。
3. 每天将该构造与其他构造及整个软件产品（以其当前的形式）集成起来进行冒烟测试。这种集成方法可以是自顶向下的，也可以自底向上的。

引述 将每日构造当成项目的心跳。如果没有了心跳，项目就死了。

Jim McCarthy

提问 从冒烟测试中可以得到什么好处？

每天频繁的测试让管理者和专业人员都能够对集成测试的进展做出实际的评估。McConnell[MCO96] 是这样描述冒烟测试的：

> 冒烟测试应该对整个系统进行彻底的测试。它不一定是穷举的，但应能暴露主要问题。冒烟测试应该足够彻底，以使得若构造通过测试，则可以假定它足够稳定以致能经受更彻底的测试。

当应用于复杂的、时间关键的软件工程项目时，冒烟测试提供了下列好处：

- 降低了集成风险。冒烟测试是每天进行的，能较早地发现不相容性和业务阻塞错误，从而降低了因发现错误而对项目进度造成严重影响的可能性。
- 提高最终产品的质量。由于这种方法是面向构建（集成）的，因此，冒烟方法既有可能发现功能性错误，也有可能发现体系结构和构件级设计错误。若较早地改正了这些错误，产品的质量就会更好。
- 简化错误的诊断和修正。与所有的集成测试方法一样，冒烟测试期间所发现的错误可能与新的软件增量有关，也就是说，新发现的错误可能来自刚加入到构造中的软件。
- 易于评估进展状况。随着时间的推移，更多的软件被集成，也更多地展示出软件的工作状况。这就提高了团队的士气，并使管理者对项目进展有较好的把握。

479

集成测试工作产品。软件集成的总体计划和特定的测试描述应该在测试规格说明中文档化。这项工作产品包含测试计划和测试规程，并成为软件配置的一部分。测试可以分为若干个阶段和处理软件特定功能及行为特征的若干个构造来实施。例如，SafeHome 安全系统的集成测试可以划分为以下测试阶段：用户交互，传感器处理，通信功能及警报处理。

每个集成测试阶段都刻画了软件内部广泛的功能类别，而且通常与软件体系结构中特定的领域相关，因此，对应于每个阶段建立了相应的程序构造（模块集）。

集成的进度、附加的开发以及相关问题也在测试计划中讨论。确定每个阶段的开始和结束时间，定义单元测试模块的“可用性窗口”。附加软件（桩模块及驱动模块）的简要描述侧重于可能需要特殊工作的特征。最后，描述测试环境和资源。特殊的硬件配置、特殊的仿

真器和专门的测试工具或技术也是需要讨论的问题。

紧接着需要描述的是实现测试计划所必需的详细测试规程。描述集成的顺序以及每个集成步骤中对应的测试，其中也包括所有的测试用例（带注释以便后续工作参考）和期望的结果列表。

实际测试结果、问题或特例的历史要记录在测试报告中，若需要的话可附在测试规格说
480 明后面。这部分包含的信息在软件维护期间很重要。也要给出适当的参考文献和附录。

22.4　面向对象软件的测试策略[⊖]

简单地说，测试的目标就是在现实的时间范围内利用可控的工作量找到尽可能多的错误。对于面向对象软件，尽管这个基本目标是不变的，但面向对象软件的本质特征改变了测试策略和测试战术（第 24 章）。

22.4.1　面向对象环境中的单元测试

在考虑面向对象的软件时，单元的概念发生了变化。封装导出了类和对象的定义。这意味着每个类和类的实例包装有属性（数据）和处理这些数据的操作。封装的类常是单元测试的重点，然而，类中包含的操作（方法）是最小的可测试单元。由于类中可以包含很多不同的操作，且特殊的操作可以作为不同类的一部分存在，因此，必须改变单元测试的战术。

我们不再孤立地对单个操作进行测试（传统的单元测试观点），而是将其作为类的一部分。为便于说明，考虑一个类层次结构，在此结构内对超类定义某操作 X，并且一些子类继承了操作 X。每个子类使用操作 X，但它应用于为每个子类定义的私有属性和操作的环境内。由于操作 X 应用的环境有细微的差别，因此有必要在每个子类的环境中测试操作 X。这意味着在面向对象环境中，以独立的方式测试操作 X（传统的单元测试方法）往往是无效的。

关键点　面向对象软件的类测试与传统软件的模块测试相似。对操作进行孤立测试是不可取的。

面向对象软件的类测试等同于传统软件的单元测试。不同的是传统软件的单元测试侧重于模块的算法细节和穿过模块接口的数据，而面向对象软件的类测试是由封装在该类中的操作和类的状态行为驱动的。

22.4.2　面向对象环境中的集成测试

由于面向对象软件没有明显的层次控制结构，因此，传统的自顶向下和自底向上集成策略（22.3.2 节）已没有太大意义。另外，由于类的成分间直接或间接的相互作用，因此每次将一个操作集成到类中（传统的增量
481 集成方法）往往是不可能的 [Ber93]。

面向对象系统的集成测试有两种不同的策略 [Bin94b]。一种策略是基于线程的测试（thread-based testing），对响应系统的一个输入或事件所需的一组类进行集成。每个线程单独地集成和测试。应用回归测试以确保没有产生副作用。另一种方法是基于使用的测试（use-based testing），通过测试很少使用服务类（如果有的话）的那些类（称为独立类）开始系统的构

关键点　面向对象软件集成测试的一个重要策略是基于线程的测试。线程是对一个输入或事件做出反应的类集合。基于使用的测试侧重于那些不与其他类进行频繁协作的类。

⊖ 基本的面向对象概念在附录 2 中介绍。

建。独立类测试完成后，利用独立类测试下一层次的类（称为依赖类）。继续依赖类的测试直到完成整个系统。

在进行面向对象系统的集成测试时，驱动模块和桩模块的使用也发生了变化。驱动模块可用于低层操作的测试和整组类的测试。驱动模块也可用于代替用户界面，以便在界面实现之前就可以进行系统功能的测试。桩模块可用于类间需要协作但其中的一个或多个协作类还未完全实现的情况。

簇测试（cluster testing）是面向对象软件集成测试中的一个步骤。这里，借助试图发现协作错误的测试用例来测试（通过检查 CRC 和对象关系模型所确定的）协作的类簇。

22.5 WebApp 的测试策略

WebApp 测试策略采用所有软件测试的基本原理，并使用面向对象系统所使用的策略和战术。此方法可概括为：

1. 对 WebApp 的内容模型进行评审，以发现错误。
2. 对接口模型进行评审，保证适合所有的用例。
3. 评审 WebApp 的设计模型，发现导航错误。
4. 测试用户界面，发现表现机制和导航机制中的错误。
5. 对每个功能构件进行单元测试。
6. 对贯穿体系结构的导航进行测试。
7. 在各种不同的环境配置下实现 WebApp，并测试 WebApp 对于每一种配置的兼容性。
8. 进行安全性测试，试图攻击 WebApp 或其所处环境的弱点。
9. 进行性能测试。
10. 通过可监控的最终用户群对 WebApp 进行测试，对他们与系统的交互结果进行错误评估。

关键点 WebApp 测试的总体策略在这里可以总结为 10 个步骤。

网络资源 WebApp 测试方面的优秀文章可以在 www.sticky-minds.com/test-ing.asp 找到。

482

因为很多 WebApp 在不断进化，所以 WebApp 测试是 Web 支持人员所从事的一项持续活动，他们使用回归测试，这些测试是从首次开发 WebApp 时所开发的测试中导出的。WebApp 测试方法将在第 25 章讨论。

22.6 移动 App 的测试策略

移动 App 的测试策略采用所有软件测试的基本原则。然而，移动 App 的独特性质要求考虑一些特殊的测试方法：

- *用户体验测试*。用户在开发过程的早期就介入，以确保移动 App 在所有被支持的设备上实现其可用性及利益相关者的最高期望。
- *设备兼容性测试*。测试人员要验证移动 App 在所有要求的硬件设备和软件组合上都能够正确地工作。
- *性能测试*。测试人员要专门针对移动设备检查非功能性需求（例如，下载时间、处理器速度、存储容量、供电）。
- *连接性测试*。测试人员要确保移动 App 能够访问任何所需要的网络或者 Web 服务，并且可以容忍微弱或者中断的网络访问。

- 安全性测试。测试人员要确保移动 App 没有违背用户的私密性及安全性需求。
- 在现实环境中测试。现实条件下，在实际用户的设备上及全球各种网络环境中对 App 进行测试。
- 认证测试。测试人员要确保移动 App 满足发布它的应用程序商店所建立的标准。

移动 App 的测试方法将在第 26 章讨论。

22.7　确认测试

确认测试始于集成测试的结束，那时已测试完单个构件，软件已组装
483 成完整的软件包，且接口错误已被发现和改正。在进行确认测试或系统级测试时，不同类型软件之间的差别已经消失，测试便集中于用户可见的动作和用户可识别的系统输出。

确认可用几种方式进行定义，但是，其中一个简单（尽管粗糙）的定义是当软件可以按照客户合理的预期方式工作时，确认就算成功。在这一点上，喜欢吹毛求疵的软件开发人员可能会提出异议："谁或者什么是合理预期的裁决者呢?"如果已经开发了*软件需求规格说明*文档，那么此文档就描述了所有用户可见的软件属性，并包含*确认准则*部分，确认准则部分就形成了确认测试方法的基础。

关键点 与所有其他测试步骤相类似，确认测试尽力发现错误，但是它侧重于需求级的错误，即那些对最终用户而言显而易见的错误。

22.7.1　确认测试准则

软件确认是通过一系列表明软件功能与软件需求相符合的测试而获得的。测试计划列出将要执行的测试类，测试规程定义了特定的测试用例，设计的特定测试用例用于确保软件满足所有的功能需求，具有所有的行为特征，所有内容都准确无误且正确显示，达到所有的性能需求，文档是正确的、可用的，且满足其他需求（如可移植性、兼容性、错误恢复和可维护性）。如果发现了与规格说明的偏差，则要创建缺陷列表。并且必须确定（利益相关者可以接受的）解决缺陷的方法。

22.7.2　配置评审

确认过程的一个重要成分是配置评审。评审的目的是确保所有的软件配置元素已正确开发、编目，且具有改善支持活动的必要细节。有时将配置评审称为审核（audit），这将在第 29 章详细讨论。

22.7.3　α 测试和 β 测试

对软件开发者而言，预见用户如何实际使用程序几乎是不可能的。软件使用指南（使用手册）可能会被错误理解；可能会使用令用户感到奇怪的数据组合；测试者看起来很明显的输出对于工作现场的用户却是难以理解的。

为客户开发定制软件时，执行一系列验收测试能使客户确认所有的需求。验收测试是由最终用户而不是软件工程师进行的，它的范围从非正式的"测试驱动"直到有计划地、系统地进行一系列测试。实际上，验收测试的执行可能超过几个星期甚至几个月，因此，可以发现长时间以来影响

引述 只要给予足够的关注，所有 bug 都是容易找到的（例如：给予足够多的 b 测试人员和相关的开发人员，几乎每个问题都很快捕获，且容易修改）。

E. Raymond

系统的累积错误。

若将软件开发为产品，由多个用户使用，让每个用户都进行正式的验收测试，这当然是不切实际的。多数软件开发者使用称为 α 测试与 β 测试的过程，以期查找到似乎只有最终用户才能发现的错误。 484

α 测试是由有代表性的最终用户在开发者的场所进行。软件在自然设置下使用，开发者站在用户的后面观看，并记录错误和使用问题。α 测试在受控的环境下进行。

提问　α 测试和 β 测试之间的区别是什么？

β 测试在一个或多个最终用户场所进行。与 α 测试不同，开发者通常不在场，因此，β 测试是在不为开发者控制的环境下“现场”应用软件。最终用户记录测试过程中遇见的所有问题（现实存在的或想象的），并定期报告给开发者。接到 β 测试的问题报告之后，开发人员对软件进行修改，然后准备向最终用户发布软件产品。

β 测试的一种变体称为客户验收测试，有时是按照合同交付给客户时进行的。客户执行一系列的特定测试，试图在从开发者那里接收软件之前发现错误。在某些情况下（例如，大公司或政府系统），验收测试可能是非常正式的，会持续很多天甚至好几个星期。

SafeHome　准备确认

[场景] Doug Miller 的办公室，构件级设计及这些构件的构建工作正继续进行。

[人物] Doug Miller，软件工程经理；Vinod、Jamie、Ed 和 Shakira，SafeHome 软件工程团队成员。

[对话]

Doug：我们将在三个星期内准备好第一个增量的确认，怎么样？

Vinod：大概可以吧。集成进展得不错。我们每天执行冒烟测试，找到了一些 bug，但还没有我们处理不了的事情。到目前为止，一切都很好。

Doug：跟我谈谈确认。

Shakira：可以。我们将使用所有的用例作为测试设计的基础。目前我还没有开始，但我将为我负责的所有用例开发测试。

Ed.：我这里也一样。

Jamie：我也一样。但是我们已经将确认测试与 α 测试和 β 测试一起考虑了，不是吗？

Doug：是，事实上我一直考虑请外包商帮我们做确认测试。在预算中我们有这笔钱……它将给我们新的思路。

Vinod：我认为确认测试已经在我们的控制之中了。

Doug：我确信是这样，但 ITG（独立测试组）能用另一种眼光来看这个软件。

Jamie：我们的时间很紧了，Doug，我没有时间培训新人来做这项工作。

Doug：我知道，我知道。但 ITG 仅根据需求和用例来工作，并不需要太多的培训。

Vinod：我仍然认为确认测试已经在我们的控制之中了。

Doug：我知道，Vinod，但在这方面我将强制执行。计划这周的后几天与 ITG 见面。让他们开始工作并看他们有什么意见。

Vinod：好的，或许这样做可以减轻工作负荷。 485

22.8 系统测试

引述 与死亡和税收一样，测试既是令人不愉快的，也是不可避免的。
Ed Yourdon

在本书的开始，我们就强调过，软件只是基于计算机大系统的一部分。最终，软件要与其他系统成分（如硬件、人和信息）相结合，并执行一系列集成测试和确认测试。这些测试已超出软件过程的范围，而且不仅仅由软件工程师执行。然而，软件设计和测试期间所采取的步骤可以大大提高在大系统中成功地集成软件的可能性。

一个传统的系统测试问题是“相互指责”。这种情况出现在发现错误时，每个系统成分的开发人员都因为这个问题抱怨别人。其实大家都不应该陷入这种无谓的争论之中，软件工程师应该预见潜在的接口问题，以及：（1）设计出错处理路径，用以测试来自系统其他成分的所有信息；（2）在软件接口处执行一系列模拟不良数据或其他潜在错误的测试；（3）记录测试结果，这些可作为出现“相互指责”时的“证据”；（4）参与系统测试的计划和设计，以保证软件得到充分的测试。

22.8.1 恢复测试

多数基于计算机的系统必须从错误中恢复并在一定的时间内重新运行。在有些情况下，系统必须是容错的，也就是说，处理错误绝不能使整个系统功能都停止。而在有些情况下，系统的错误必须在特定的时间内或严重的经济危害发生之前得到改正。

*恢复测试*是一种系统测试，通过各种方式强制地让系统发生故障，并验证其能适当恢复。若恢复是自动的（由系统自身完成），则对重新初始化、检查点机制、数据恢复和重新启动都要进行正确性评估。若恢复需要人工干预，则应计算平均恢复时间（Mean-Time-To-Repair, MTTR）以确定其值是否在可接受的范围之内。

22.8.2 安全测试

任何管理敏感信息或能够对个人造成不正当伤害（或带来好处）的计算机系统都是非礼或非法入侵的目标。入侵包括广泛的活动：黑客为了娱乐而试图入侵系统，不满的雇员为了报复而试图破坏系统，不良分子在非法利益驱使下试图入侵系统。

*安全测试*验证建立在系统内的保护机制是否能够实际保护系统不受非法入侵。引用Beizer[Bei84]的话来说：“系统的安全必须经受住正面的攻击，但是也必须能够经受住侧面和背后的攻击。”

486 只要有足够的时间和资源，好的安全测试最终将能够入侵系统。系统设计人员的作用是使攻破系统所付出的代价大于攻破系统之后获取信息的价值。安全测试和安全工程在第27章详细讨论。

22.8.3 压力测试

本章前面所讨论的软件测试步骤能够对正常的程序功能和性能进行彻底的评估。压力测试的目的是使软件面对非正常的情形。本质上，进行压力测试的测试人员会问：“在系统失效之前，能将系统的运行能力提高到什么程度？”

压力测试要求以一种非正常的数量、频率或容量的方式执行系统。例如：（1）在平均每秒出现1～2次中断的情形下，可以设计每秒产生10次中断的测试用例；（2）将输入数据

的量提高一个数量级以确定输入功能将如何反应；（3）执行需要最大内存或其他资源的测试用例；（4）设计可能在实际的运行系统中产生惨败的测试用例；（5）创建可能会过多查找磁盘驻留数据的测试用例。从本质上来说，压力测试者将试图破坏程序。

压力测试的一个变体称为敏感性测试。在一些情况下（最常见的是在数学算法中），包含在有效数据界限之内的一小部分数据可能会引起极端处理情况，甚至是错误处理或性能的急剧下降。敏感性测试试图在有效输入类中发现可能会引发系统不稳定或者错误处理的数据组合。

> **引述**　若你正在尽力查找实际系统的bug，且没有为你的软件提供实际的压力测试，那么现在应该是你立即开始的时候了。
>
> *Boris Beizer*

22.8.4　性能测试

对于实时和嵌入式系统，提供所需功能但不符合性能需求的软件是不能接受的。性能测试用来测试软件在集成环境中的运行性能。在测试过程的任何步骤中都可以进行性能测试。即使是在单元级，也可以在执行测试时评估单个模块的性能。然而，只有当整个系统的所有成分完全集成之后，才能确定系统的真实性能。

性能测试经常与压力测试一起进行，且常需要硬件和软件工具。也就是说，以严格的方式测量资源（例如处理器周期）的利用往往是必要的。当有运行间歇或事件（例如中断）发生时，外部工具可以监测到，并可定期监测采样机的状态。通过检测系统，测试人员可以发现导致效率降低和系统故障的情形。

22.8.5　部署测试

在很多情况下，软件必须在多种平台及操作系统环境中运行。有时也将部署测试称为配置测试，即在软件将要运行其中的每一种环境中测试软件。另外，部署测试检查客户将要使用的所有安装程序及专业安装软件（例如“安装程序”），并检查用于向最终用户介绍软件的所有文档。

487

软件工具　测试计划与管理

[目标] 这些工具辅助软件团队根据所选择的测试策略制订计划，并进行测试过程的管理。

[机制] 这类工具提供测试计划、测试存储、管理与控制、需求追踪、集成、错误追踪和报告生成。项目经理用这些工具作为项目策划工具的补充；测试人员利用这些工具计划测试活动，以及在测试进行时控制信息的流动。

[代表性工具][⊖]

- QaTraq 测试用例管理工具由 Traq Software（www.testmanagement.com）开发，“鼓励以结构化方法进行测试管理”。
- QAComplete 由 SmartBear（http://SmartBear.com/products/qa-tools/test-management）开发，为管理敏捷测试过程的各个阶段提供单点控制。
- TestWorks 由 Software Research, Inc.（http://www.testworks.com/）开发，包含一个完整的、集成的成套测试工具，包括测试管理和报告工具。
- OpensourceTesting.org（www.opensou-

⊖ 这里提到的工具只是此类工具的例子，并不代表本书支持采用这些工具。在大多数情况下，工具名称被各自的开发者注册为商标。

rcetesting.org/testmgt.php）列出了各种开源测试管理和计划工具。

- OpensourceManagement.com（http://www.opensourcetestmanagement.com/）列出了各种开源测试管理和计划工具。

22.9　调试技巧

软件测试是一种能够系统地加以计划和说明的过程，可以进行测试用例设计，定义测试策略，根据预期的结果评估测试结果。

调试（debugging）出现在成功的测试之后。也就是说，当测试用例发现错误时，调试是使错误消除的过程。尽管调试可以是也应该是一个有序的过程，但它仍然需要很多技巧。当评估测试结果时，软件工程师经常面对的是软件问题表现出的“症状”，即错误的外部表现与其内在原因之间可能并没有明显的关系。调试就是探究这一关系的智力过程。

引述　一旦我们开始编程，就会惊奇地发现，程序并不是像我们想象的那样容易正确。不得不去发现错误。我能记起那一刻，意识到从那时起我将花费大部分精力去查找自己程序中的错误。

Maurice Wilkes, discovers debugging, 1949

22.9.1　调试过程

488

调试并不是测试，但总是发生在测试之后[⊖]。参看图 22-7，执行测试用例，对测试结果进行评估，而且期望的表现与实际表现不一致时，调试过程就开始了。在很多情况下，这种不一致的数据是隐藏在背后的某种原因所表现出来的症状。调试试图找到隐藏在症状背后的原因，从而使错误得到修正。

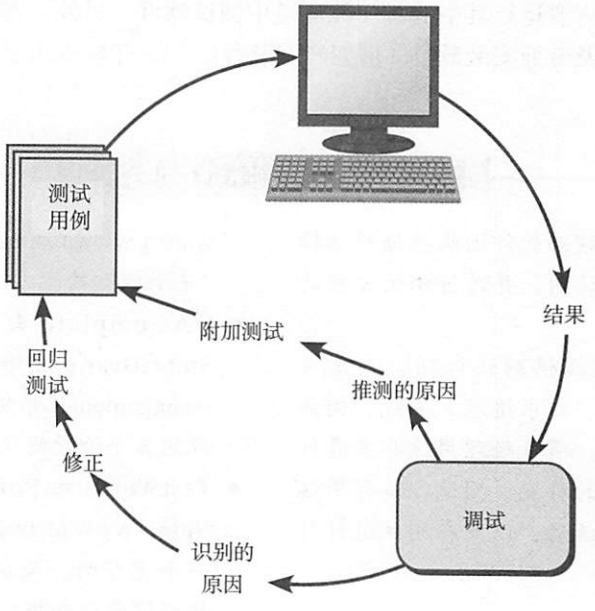

图 22-7　调试过程

⊖ 为了对此进行说明，我们采用最广泛的可能测试视图。在软件发布之前，不仅开发者要测试软件，客户 / 用户在每次使用软件之前也要对其进行测试。

调试过程通常得到以下两种结果之一：（1）发现问题的原因并将其改正；（2）未能找到问题的原因。在后一种情况下，调试人员可以假设一个原因，设计一个或多个测试用例来帮助验证这个假设，重复此过程直到改正错误。

为什么调试如此困难？在很大程度上，人类心理（22.9.2 节）与这个问题的答案间的关系比软件技术更密切。然而，软件 bug 的以下特征为我们提供了一些线索。

提问　为什么调试如此困难？

1. 症状与原因出现的地方可能相隔很远。也就是说，症状可能在程序的一个地方出现，而原因实际上可能在很远的另一个地方。高度耦合的构件（第 12 章）加剧了这种情况的发生；
2. 症状可能在另一个错误被改正时（暂时）消失；
3. 症状实际上可能是由非错误因素（例如舍入误差）引起的；
4. 症状可能是由不易追踪的人为错误引起的；
5. 症状可能是由计时问题而不是处理问题引起的；
6. 重新产生完全一样的输入条件是困难的（如输入顺序不确定的实时应用）；
7. 症状可能时有时无，这在软硬件耦合的嵌入式系统中尤为常见；
8. 症状可能是由分布运行在不同处理器上的很多任务引起的。

引述　每个人都知道调试的难度是首次写程序的两倍。因此，如果你像写它时一样聪明，那么将如何对它进行调试呢？

Brian Kernighan

489

在调试过程中，我们遇到错误的范围从恼人的小错误（如不正确的输出格式）到灾难性故障（如系统失效，造成严重的经济或物质损失）。错误越严重，查找错误原因的压力也就越大。通常情况下，这种压力会使软件开发人员在修改一个错误的同时引入两个甚至更多的错误。

22.9.2　心理因素

建议　设置一个时限，比如两个小时。在这个时间限制内，尽力独自调试程序。然后，求助。

遗憾的是，有证据表明，调试本领属于一种个人天赋。一些人精于此道，而另一些人则不然。尽管有关调试的实验证据可以有多种解释，但对于具有相同教育和经验背景的程序员来说，他们的调试能力是有很大差别的。尽管学会调试可能比较困难，但仍然可以提出一些解决问题的方法。这些方法将在下一节讨论。

SafeHome　调试

[场景] Ed 的工作间，进行编码和单元测试。

[人物] Ed 和 Shakira，SafeHome 软件工程团队的成员。

[对话]

Shakira（经过工作间门口时向里张望）：嘿，午饭时你在哪儿？

Ed：就在这里……工作。

Shakira：你看上去很沮丧，怎么回事？

Ed（轻声地叹息）：我一直忙于解决这个 bug，从今天早晨 9:30 发现它之后，现在已下午 2:40 了，我还没有线索。

Shakira：我想大家都同意在调试我们自己的东西时花费的时间不应该超过一小时，我们请求帮助，怎么样？

Ed：好，但是……

Shakira（走进工作间）：什么问题？

Ed：很复杂。而且，我查看这个问题已有 5 个小时，你不可能在 5 分钟内找到原因。

Shakira：真让我兴奋，什么问题？

（Ed向Shakira解释问题，Shakira看了大约30秒没有说什么，然后……）

Shakira（笑了）：哦，就是那个地方，在
490 循环开始之前，变量setAlarmCondition是不是不应该设置为“false”？

（Ed不相信地盯着屏幕，向前躬着腰，开始对着监视器轻轻地敲自己的头。Shakira开怀大笑，起身走出去。）

22.9.3　调试策略

不论使用什么方法，调试有一个基本目标：查找造成软件错误或缺陷的原因并改正。通过系统评估、直觉和运气相结合可以实现这个目标。

总的来说，有三种调试方法[Mye79]：蛮干法、回溯法及原因排除法。这三种调试方法都可以手工执行，但现代的调试工具可以使调试过程更有效。

调试方法。蛮干法可能是查找软件错误原因最常用但最低效的方法。在所有其他方法都失败的情况下，我们才使用这种方法。利用“让计算机自己找错误”的思想，进行内存转储，实施运行时跟踪，以及在程序中添加一些输出语句。希望在所产生的大量信息里可以让我们找到错误原因的线索。尽管产生的大量信息可能最终带来成功，但更多的情况下，这样做只是浪费精力和时间，它将率先耗尽我们的想法！

> **引述**　修改一个已坏的程序时，第一步是让它重复失败（尽可能是在最简单的例子上）。
>
> *T. Duff*

回溯法是比较常用的调试方法，可以成功地应用于小程序中。从发现症状的地方开始，向后追踪（手工）源代码，直到发现错误的原因。遗憾的是，随着源代码行数的增加，潜在的回溯路径的数量可能会变得难以控制。

第三种调试方法——原因排除法——是通过演绎或归纳并引入二分法的概念来实现。对与错误出现相关的数据加以组织，以分离出潜在的错误原因。假设一个错误原因，利用前面提到的数据证明或反对这个假设。或者，先列出所有可能的错误原因，再执行测试逐个进行排除。若最初的测试显示出某个原因假设可能成立的话，则要对数据进行细化以定位错误。

自动调试。以上调试方法都可以使用辅助调试工具。在尝试调试策略时，调试工具为软件工程师提供半自动化的支持。Hailpern与Santhanam[Hai02]总结这些工具的状况时写道：“人们已提出许多新的调试方法，而且许多商业调试环境也已经具备。集成开发环境（IDE）提供了一种方法，无需编译就可以捕捉特定语言的预置错误（例如，语句结束符的丢失、变量未定义等）。”可用的工具包括各种调试编译器、动态调试辅助工具（跟踪工具）、测试用例自动生成器和交互引用映射工具。然而，工具不能替代基于完整设计模型和清晰源代码的
491 仔细评估。

软件工具　调试

[目标]这些工具为那些调试软件问题的人提供自动化的帮助，目的是洞察那些用手工调试可能难以捕捉的问题。

[机制]大多数调试工具是针对特定编程语言和环境的。

[代表性工具]⊖

- Borland Silkt。由Borland（http://www.borland.com/products/）开发，辅助测试

⊖ 这里提到的工具只是此类工具的例子，并不代表本书支持采用这些工具。在大多数情况下，工具名称被各自的开发者注册为商标。

和调试。

- Coverty Development Testing Platform。由 Coverty（http://www.coverty.com/products/）开发，该工具将质量测试和安全性测试引入早期开发过程。
- C++Test。由 Parasoft（www.parasoft.com）开发，是一个单元测试工具，对 C 和 C++ 代码的测试提供完全的支持。调试功能有助于已发现错误的诊断。
- CodeMedic。由 NewPlanet Software(www.newplanetsoftware.com/medic/）开发，为标准的 Unix 调试器 gdb 提供图形界面，且实现了它的最重要特征。gdb 目前支持 C/C++、Java、PalmOS、各种嵌入式操作系统、汇编语言、FORTRAN 和 Modula-2。
- GNATS。一个免费应用软件（www.gnu.org/software/gnats/），是一组用于追踪 bug 报告的工具。

人为因素。若不提到强有力的助手——其他人，那么有关调试方法和调试工具的任何讨论都是不完整的。有一个新颖的观点：每个人都可能有为某个错误一直头痛的经历[⊖]。因此，调试的最终箴言应该是："若所有方法都失败了，就该寻求帮助！"

22.9.4 纠正错误

引述 最好的测试人员不是发现错误最多的人，而是纠正错误最多的人。
Cem Kaner et al.

一旦找到错误，就必须纠正。但是，我们已提到过，修改一个错误可能会引入其他错误，因此，不当修改造成的危害会超过带来的益处。Van Vleck[Van89] 提出，在进行消除错误原因的"修改"之前，每个软件工程师应该问以下三个问题：

1. 这个错误的原因在程序的另一部分也产生过吗？在多数情况下，程序的错误是由错误的逻辑模式引起的，这种逻辑模式可能会在别的地方出现。仔细考虑这种逻辑模式可能有助于发现其他错误。
2. 进行修改可能引发的"下一个错误"是什么？在改正错误之前，应该仔细考虑源代码（最好包括设计）以评估逻辑与数据结构之间的耦合。若要修改高度耦合的程序段，则应格外小心。 492
3. 为避免这个错误，我们首先应当做什么呢？这个问题是建立统计软件质量保证方法的第一步（第 21 章）。若我们不仅修改了过程，还修改了产品，则不仅可以排除现在的程序错误，还可以避免程序今后可能出现的错误。

22.10 小结

软件测试在软件过程中所占的技术工作量比例最大。不考虑所构建软件的类型，系统测试计划、运行和控制策略从考虑软件的小元素开始，逐渐面向整个软件。

软件测试的目标是发现错误。对于传统软件，这个目标是通过一系列测试步骤达到的。单元测试和集成测试侧重于验证模块的功能以及将模块集成到程序结构中；确认测试验证软件需求的可追溯性；系统测试在软件集成为较大的系统时对软件进行确认。每个测试步骤都是通过有助于测试用例设计的一系列系统化测试技术来完成的。在每一步测试中，用于考虑

⊖ 在设计软件和编码的过程中，结对编程（第 5 章中所讨论的极限编程模型的一部分）提供了一种"排错"机制。

软件的抽象层次都得到了扩展。

面向对象软件的测试策略开始于类中操作的执行，然后转到以集成为目的的基于线程的测试。线程是响应输入或事件的一组类。基于使用的测试关注那些不与其他类过多协作的类。

对 WebApp 及移动 App 的测试方法与面向对象系统是一样的。然而，所设计的测试用于检查内容、功能性、界面、导航以及应用的性能和安全性方面。移动 App 需要特殊的测试方法，重点是在多种设备及实际网络环境中对应用程序进行测试。

与测试（测试是一种系统的、有计划的活动）不同的是，调试必须被看作一种技术。从问题的症状显示开始，调试活动要去追踪错误的原因。在调试过程可以利用的众多资源中，最有价值的是其他软件工程师的建议。

习题与思考题

493

22.1　用自己的话描述验证与确认的区别。两者都要使用测试用例设计方法和测试策略吗？

22.2　列出一些可能与独立测试组（ITG）的创建相关的问题。ITG 与 SQA 小组由相同的人员组成吗？

22.3　使用 22.1.3 节中描述的测试步骤来建立测试软件的策略总是可能的吗？对于嵌入式系统，会出现哪些可能的复杂情况？

22.4　为什么具有较高耦合度的模块难以进行单元测试？

22.5　“防错法”（antibugging，22.3.1 节）的概念是一个非常有效的方法。当发现错误时，它提供了内置调试帮助：

a. 为防错法开发一组指导原则。

b. 讨论利用这种技术的优点。

c. 讨论利用这种技术的缺点。

22.6　项目的进度安排是如何影响集成测试的？

22.7　在所有情况下，单元测试都是可能的或是值得做的吗？提供实例来说明你的理由。

22.8　谁应该完成确认测试——是软件开发人员还是软件使用者？说明你的理由。

22.9　为本书讨论的 SafeHome 系统开发一个完整的测试策略，并以测试规格说明的方式形成文档。

22.10　作为一个班级项目，为你的安装开发调试指南。这个指南应该提供面向语言和面向系统的建议。这些建议是通过总结学校学习过程中所遇到的挫折得到的。从一个经过全班和老师评审过的大纲开始，并在你的局部范围内将这个指南发布给其他人。

扩展阅读与信息资源

实际上，每本软件测试的书都讨论测试策略和测试用例设计方法。Everett 和 Raymond（《Software Testing》，Wiley-IEEE Computer Society Press, 2007）、Black（《Pragmatic Software Testing》，Wiley, 2007）、Spiller 和他的同事（《Software Testing Process: Test Management》，Rocky Nook, 2007）、Perry（《Effective methods for Software Testing》，3rd ed., Wiley, 2005）、Lewis（《Software Testing and Continuous Quality Improvement》，2nd ed., Auerbach, 2004）、Loveland 和他的同事（《Software Testing Techniques》，Charles River Media, 2004）、Burnstein（《Practical Software Testing Techniques》，Springer, 2003）、Dustin（《Effective Software Testing》，Addison-Wesley, 2002）以及 Kaner 和他的同事（《Lessons learned in Software Testing》，Wiley, 2001）所写的书只是讨论测试原理、概念、策略和方法的众多书籍中的一小部分。

对于敏捷软件开发方法有兴趣的读者，Gartner（《ATDD by Example: A Practical Guide to Acceptance Test-Driven Development》，Addison-Wesley, 2012）、Crispin 和 Gregory（《Agile Testing: A Practical Guide

for Testers and Teams》, Addison-Wesley, 2009)、Crispin 和 House(《Testing Extreme Programming》, Addison-Wesley, 2002)以及 Beck(《Test Driven Development: By Example》, Addison-Wesley, 2002)针对极限编程技术描述了测试策略与战术。Kamer 和他的同事(《Lessons Learned in Software Testing》, Wiley, 2001)描述了每个测试人员应该学习的 300 多条实用的“教训”(指导原则)。Watkins(《Testing IT: An Off-the Shelf Testing Process》(2nd ed.), Cambridge University Press, 2010)为所有类型的软件(开发的 494
和获取的)建立了有效的测试框架。Manages 和 O'Brien(《Agile Testing with Ruby and Rails》, Apress, 2008)描述了针对 Ruby 编程语言和 Web 框架的测试策略和技术。

Bashir 和 Goel(《Testing Object-Oriented Software》, Springer-Verlag, 2012)、Sykes 和 McGregor(《Practical Guide to Testing Object-Oriented Software》, Addison-Wesley, 2001)、Binder(《Testing Object-Oriented Systems》, Addison-Wesley, 1999)、Kung 和他的同事(《Testing Object-Oriented Software》, IEEE Computer Society Press, 1998)以及 Marick(《The Craft of Software Testing》, Prentice-Hall, 1997)描述了测试面向对象系统的策略与方法。

Grotker 和他的同事(《The Developer's Guide to Debugging》(2nd ed.), CreateSpace Independent Publishing, 2012)、Whittaker(《Exploratory Testing》, Addison-Wesley, 2009)、Zeller(《Why Programs Fail: A Guide to Systematic Debugging》(2nd ed.), Morgan Kaufmann, 2009)、Butcher(《Debug It!》, Pragmatic Bookshelf, 2009)、Agans(《Debugging》, Amacon, 2006)以及 Tells 和 Heieh(《The Science of Debugging》, The Coreolis Group, 2001)所编写的书中包括调试指南。Kaspersky(《Hacker Debugging Uncovered》, A-list Publishing, 2005)讲述了调试工具的技术。Younessi(《Object-Oriented Defect Management of Software》, Prentice-Hall, 2002)描述了面向对象系统的缺陷管理技术。Beizer[Bei84]描述了有趣的“bug 分类”,这种分类引领了很多制定测试计划的有效方法。

Graham 和 Fewster(《Experience of Test Automation》, Addison-Wesley, 2012)以及 Dustin 和他的同事(《Implementing Automated Software Testing》, Addison-Wesley, 2009)所编写的书讨论了自动测试。Hunt 和 John(《Java Performance》, Addison-Wesley, 2011)、Hewardt 和他的同事(《Advanced .NET Debugging》, Addison-Wesley, 2009)、Matloff 和他的同事(《The Art of Debugging with GDB, DDD, and Eclipse》, No Starch Press, 2008)、Madisetti 和 Akgul(《Debugging Embeded Systems》, Springer, 2007)、Robbins(《Debugging Microsoft .NET 2.0 Applications》, Microsoft Press, 2005)、Best(《Linux Debugging and Performance Tuning》, Prentice Hall, 2005)、Ford 和 Teorey(《Practical Debugging in C++》, Prentice Hall, 2002)以及 Brown(《Debugging Perl》, McGraw-Hill, 2000)、Mitchell(《Debugging Java》, McGraw-Hill, 2000)都针对书名所指的环境,讲述了调试的特殊性质。

从网上可以获得大量有关软件测试策略的信息资源。与软件测试策略有关的最新的参考文献可在 SEPA 网站 www.mhhe.com/ pressman 找到。 495