第3章

软件测试过程与管理

本章介绍了软件测试的过程，详细描述了测试执行的5个阶段，最后对软件过程管理、软件需求管理、软件配置管理和缺陷管理相关内容进行了介绍。

3.1 软件测试过程概述

3.1.1 软件测试阶段

软件测试贯穿于软件的整个生命周期，与软件开发各阶段的活动息息相关。与软件开发过程相对应，软件测试的过程包括测试计划、测试设计、测试执行、测试评估以及测试管理5个阶段，如图3.1所示。

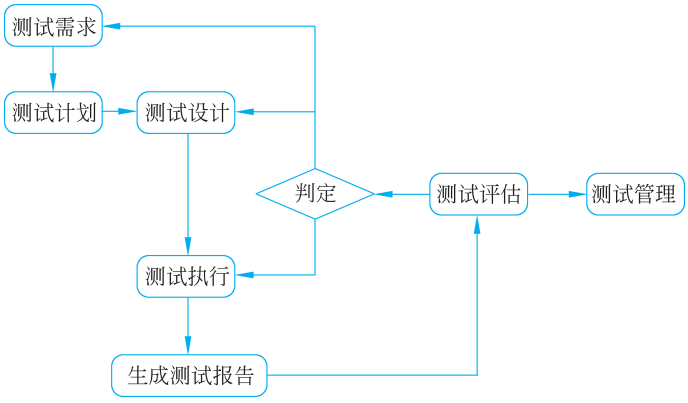

图3.1 软件测试过程

1. 测试计划

测试计划是规划所进行测试活动的范围、方法、资源和进度的文档。通过确定测试对象、测试特性、测试任务、对异常情况的处理方法等，有效预防软件测试的风险，保障计划的顺利实施。

2. 测试设计

测试设计文档是由测试人员负责编写的，将测试需求分析细化为若干可执

行测试的过程，同时为每个测试过程选择适当的测试用例，来保证测试结果的有效性。测试设计用于确定各个阶段的目标，并明确每个阶段要完成的测试活动，此外还包括评估完成活动所需的时间和资源，进行活动安排和资源分配等过程。它的主要依据是软件需求文档、软件需求规格说明书、软件设计文档等。在制定测试计划的时候还要明确测试背景，包括项目的简单介绍、参与测试人员的介绍；明确测试资源，包括设备需求、人员需求、环境需求；明确测试策略，包括搭建测试环境、选取测试工具、对测试人员进行培训等。

3. 测试执行

测试执行过程一般分为 5 个阶段：单元测试、集成测试、确认测试、系统测试及验收测试，其具体执行步骤如图 3.2 所示。

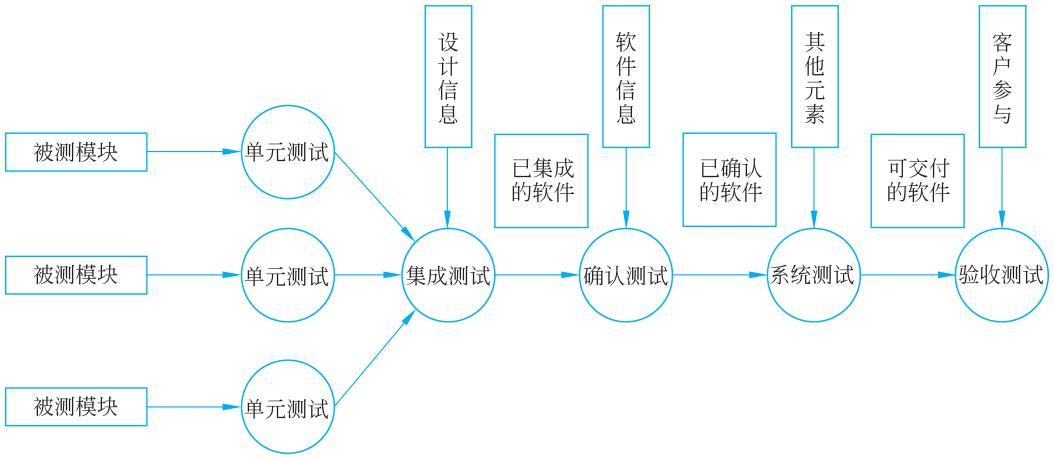

图 3.2　测试执行步骤

单元测试：又称模块测试，是针对软件设计的最小单位（程序模块）进行的针对正确性检查的测试工作，单元测试需要从程序内部结构出发来设计测试用例，所以多个模块可以并行独立地进行单元测试。

集成测试：又称组装测试或联合测试，在单元测试的基础上，将所有模块按照概要设计规格说明书和详细设计规格说明书的要求组装起来并对模块的接口进行测试。

确认测试：用于验证软件的功能、性能以及其他特性是否与用户的要求一致。确认测试一般包括有效性测试和软件配置复查，通常由第三方测试机构进行。

系统测试：测试软件系统能否投入使用，系统性能及表现是否和计算机硬件设备、外围设备兼容等。

验收测试：是测试执行的最后一步，主要是让用户对软件进行测试，确保用户的需求都被满足，并且没有新的错误产生。

4. 测试评估

测试评估是生成具有测试缺陷以及测试缺陷跟踪信息的报告，对应用软件的质量和开发团队的工作进度以及工作效率进行综合分析。

5. 测试管理

测试管理就是对每种具体测试任务、流程、体系、结果、工具等进行监督和管理，以此来促进和达到软件测试目标。

3.1.2　软件测试模型

软件测试的过程是一种抽象的模型，用于定义软件测试的流程和方法。开发过程的质量决定了软件的质量，测试过程的质量将直接影响测试结果的准确性和有效性。软件测试的过程和软件开发的过程一样，都需要遵循软件工程的原理。

随着测试过程不断规范化发展，软件测试专家通过实践总结出了一批测试过程模型，如 V 模型、W 模型、H 模型等。这些模型将测试过程抽象成为具体的流程，同时与开发过程的各个阶段进行融合，逐渐成为测试过程管理的重要依据。

1. V 模型

如图 3.3 所示，V 模型明确地标注了测试过程中存在着哪些测试活动阶段，并且清楚地表达了这些测试阶段和开发过程各阶段的对应关系。从图 3.3 中可以看到，单元测试和集成测试分别对应详细设计和概要设计，故这两个阶段应当检测程序的执行是否满足软件设计的要求；系统测试对应需求分析与系统分析，应检测系统功能、性能的质量特性是否达到系统要求的指标；验收测试对应用户需求阶段，其需要确定软件的实现是否满足用户的需要或者合同的要求。

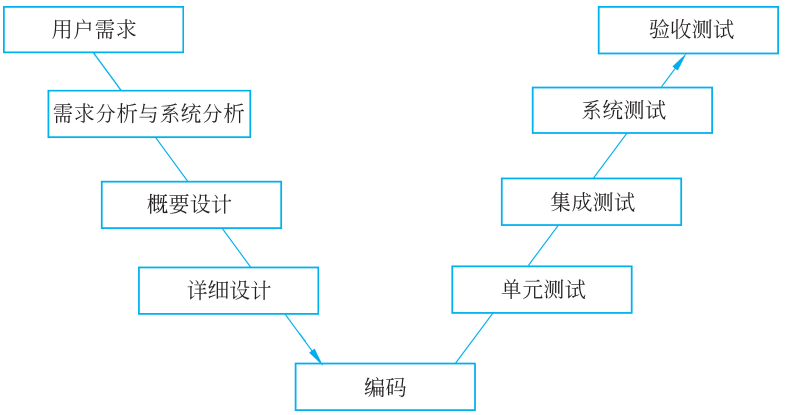

图 3.3　V 模型

但 V 模型也存在一定的局限性，它仅仅把测试作为编码之后的一个阶段，是针对程序进行的一种寻找错误的活动，并没有在需求开发阶段就开始测试，忽视了测试活动对需求分析、系统设计等活动的验证和确认功能。

2. W 模型

如图 3.4 所示，W 模型由两个 V 模型组成，其分别代表开发与测试过程。应用此模

型的测试活动与软件开发同步进行，测试对象不仅有程序，还包括软件的需求和设计，尽早发现软件缺陷可降低软件开发的成本。

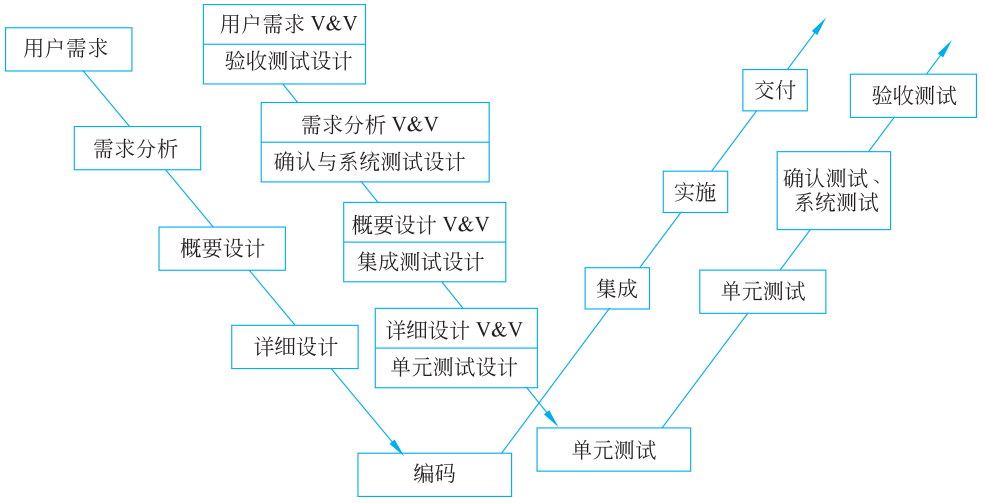

图 3.4 W 模型

在该模型中，需求、设计、编码等活动是串行的。同时，对于实际开发和测试活动而言，它们之间是线性关系，上一个阶段的活动结束，下一个阶段的活动才能开始，本身并不支持迭代。

V 模型、W 模型均存在一些缺陷。首先，它们都把软件的开发视为需求、设计、编码等一系列串行的活动。事实上，这些活动之间相互牵制，彼此交叉进行，相应的测试之间并不存在严格的次序关系，各层次之间的测试也存在反复触发、迭代和增量的关系。其次，V 模型、W 模型都没有很好地体现测试流程的完整性。

3. H 模型

如图 3.5 所示，H 模型仅演示了在整个生命周期中某个层次上的一次测试“微循环”。图 3.5 中的其他流程可以是任意开发流程，如设计流程和编码流程等，也可以是其他非开发流程，如测试流程自身。也就是说，只要测试条件成熟了，测试准备活动完成了，测试执行活动就可以进行。H 模型的测试流程如下。

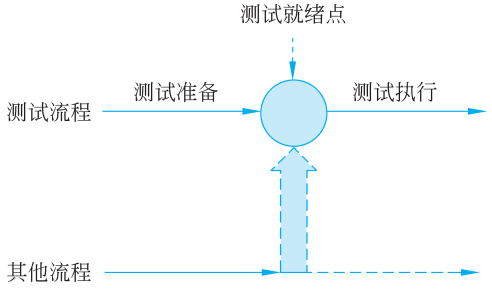

图 3.5 H 模型

测试准备：判断所有测试活动的准备是否到测试就绪点。

测试就绪点：测试准入准则，即是否可以开始执行测试的条件。

测试执行：具体的执行测试的程序。

H 模型揭示了以下内容。

(1) 软件测试不仅指测试的执行，还包括很多其他的活动。

(2) 软件测试是一个独立的流程，其贯穿产品整个生命周期，与其他流程并发地进行。当某个测试时间点就绪时，软件测试即从测试准备阶段进入测试执行阶段。

(3) 软件测试要尽早准备，尽早执行。

(4) 软件测试可以根据被测对象的不同而分层次、分阶段、分次序地执行，同时也是可以被迭代的。

3.2　单 元 测 试

测试执行的第一步就是进行单元测试，它的具体工作就是对程序员的代码进行检查，负责检查的人最好就是程序员自己，在此阶段务必检查出每个小单元的错误。完成单元测试后，测试工作才可进行至下一阶段。

3.2.1　单元测试的定义

单元测试，又称模块测试，其是对程序中的单个模块、子程序、某一函数或者类进行正确性检验的测试工作。单元测试中的单元，一般来说要根据实际情况判定其具体含义，如 C 语言中的单元一般指一个函数，Java 里的单元通常指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元测试并不是对整个程序进行测试，而是首先将注意力集中在对构成程序的较小模块进行测试。

这样做的理由有以下 3 点：首先，由于单元测试的注意力一开始就集中在程序的较小单元上，因此它可以作为一种管理组合的测试元素手段；其次，单元测试减轻了调试(即准确定位并纠正某个已知错误过程)的难度；最后，单元测试通过提供同时测试多个软件模块的可能，将并行工程引入软件测试中。

3.2.2　单元测试的思路

在进行单元测试时，需要针对单元测试的程度、时机及对象这 3 个问题进行思考。

1. 单元测试的程度

极限编程、测试驱动开发、单元测试及 JUnit 的创造者 Kent Beck 曾经说过："单元测试不是越多越好，而是越有效越好！"因此单元测试的程度并不是越细化越好，而是应关注哪些代码需要有单元测试的覆盖。下面给出一些示例。

(1) 逻辑复杂的代码。

(2) 容易出错的代码。

(3) 不易理解的代码，单元测试有助于理解代码的功能和需求。

（4）公共代码，如自定义的所有超文本传送协议（Hypertext Transfer Protocol，HTTP）请求都会经过的拦截器、工具类等。

（5）核心业务代码，一个产品里最核心最有业务价值的代码应该要有较高的单元测试覆盖率。

2. 单元测试的时机

单元测试的时机包括以下 3 种情况。

（1）在具体实现代码之前，这是测试驱动开发所提倡的。

（2）与具体实现代码同步进行，先写少量功能代码，紧接着写单元测试，重复这两个过程，直到完成功能代码开发。基本上功能代码开发完，单元测试也基本完成了。

（3）编写完功能代码再写单元测试。一般来说，事后编写的单元测试粒度都比较粗。对同样的功能代码，采取前两种方案的结果可能是用 10 个较小的单元测试来覆盖，每个单元测试都比较简单易懂，具有良好的可读性和可维护性，在重构时单元测试的改动不大；而第三种方案写的单元测试，往往是用一个较大的单元测试来覆盖，这个单元测试的逻辑往往比较复杂，测试的内容较多，可读性和可维护性也会比较差。

在实际测试过程中，可以将单元测试与具体实现代码同步进行。只有对需求具备一定的理解，才能保证代码的正确性，才能写出有效的单元测试来验证程序的正确性。

3. 单元测试的对象

在结构化程序时代，单元测试所说的单元是指函数，在当今面向对象的时代，单元测试所说的单元是指类。但以类作为测试单位，复杂度高，可操作性较差，因此笔者仍然主张以函数作为单元测试的测试单位，但可以用一个测试类来组织某个类的所有测试函数。单元测试不应过分强调面向对象，因为局部代码依然是结构化的。

3.2.3 单元测试的实施者

1. 单元测试的实施

单元测试大部分使用白盒测试法，对测试用例的编写人员来说，被测试对象的内部结构和运行机制是完全透明的。因此，唯有开发人员（即类的实现者本人）最清楚类对象的结构、机制及局限性，并且明白在各种测试条件下测试用例的输出结果应该是怎么样的。毫无疑问，只有代码的编写人员（即开发者）才是进行单元测试最合适的人选。测试人员既不了解也不注重类的结构和机制，甚至不关心是否有这样一个类。测试人员唯一注重的是系统的功能和质量是否满足软件需求规格说明书或者用户的要求，因此并不是所有的测试都应该由测试人员来实现。

在开发人员编写单元测试用例的过程中，面对一些相似度较高的代码，如果每段代码都去编写单元测试用例，虽然这保证了测试用例的覆盖率，但其实对开发者来说也造成了额外的负担。这就要求开发者重新审视自己的代码，将一些重复的代码进行重构，把代码改得简洁明了。这样既提高了覆盖率，又减少了测试工作。

另外，由于单元测试中需要尝试覆盖一些异常分支，这通常是系统测试走不到的地方，那么就需要开发者仔细地思考，这个异常分支是否真的需要？是否真的会发生？对于一些实际上绝对不会出错的函数，进行异常分支处理是完全没有必要的。

2. 单元测试的标准

1）基础

单元测试的对象是程序中最基本的单元，在此基础上，可以测试系统中一些最基本的功能点，这些功能点往往由几个基本类组成。从面向对象的设计原理出发，系统中最基本的功能点也应该由一个类及其方法来表现。单元测试要测试应用程序接口中的每个方法及参数。

2）自动化

单元测试应该集成到自动测试的框架中，保证单元测试能够自动进行，无论是参数的输入输出还是检查结果的判定都不需要人工干预。

3）彻底

一个好的单元测试应该覆盖被测单元的每条分支语句，甚至每条可能抛出的异常。如果其中某个错误的处理路径很难达到，那么就要重新思考，软件是否需要处理这个错误，要注意的一点是 100％的代码覆盖率不等于 100％的正确率。

4）独立

一般来说，系统中的各个模块都是互相依赖的，这个模块需要上一个模块的输出作为输入等。但是单元测试与单元测试之间应该是独立的、不存在顺序依赖关系的。对于每个单元测试来讲，其所需要的预设条件都应该包含在测试中。

5）可重复

单元测试必须是可重复的，并且不应受环境影响，也不应受上次测试结果的影响。这就要求测试执行时不能使用共享资源或者随机数据，如共享数据库等。一旦别人进行了别的操作，则势必会影响测试结果。

3.2.4　单元测试的内容

单元测试一般包括 5 方面的内容，分别是模块接口测试、局部数据结构测试、边界条件测试、模块中所有独立路径测试和各种错误处理测试，如图 3.6 所示。

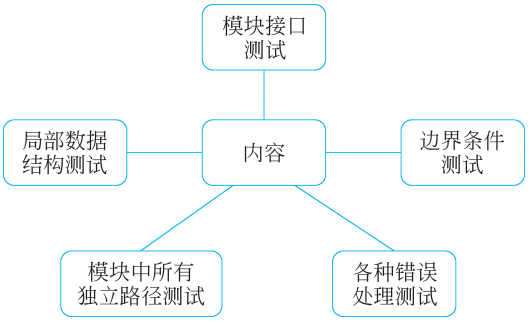

图 3.6　单元测试内容

1. 模块接口测试

模块接口测试是单元测试的基础，只有在数据能正确流入、流出的前提下，其他测试才有意义。模块接口测试同时也是集成测试的重点，其可以为后面进行的测试过程打好基础。判断测试接口是否正确应该考虑下列因素。

（1）输入的实际参数与形式参数的个数是否相同。

（2）输入的实际参数与形式参数的属性是否匹配。

（3）输入的实际参数与形式参数的量纲是否一致。

（4）调用其他模块时所给实际参数的个数是否与被调模块的形式参数个数相同。

（5）调用其他模块时所给实际参数的属性是否与被调模块的形式参数属性匹配。

（6）调用其他模块时所给实际参数的量纲是否与被调模块的形式参数量纲一致。

（7）调用预定义函数时所用参数的个数、属性和次序是否正确。

（8）是否存在与当前入口点无关的参数引用。

（9）是否修改了只读型参数。

（10）各模块对全局变量的定义是否一致。

（11）是否曾把某些约束作为参数传递。

如果模块功能包括外部输入输出，那么还应该考虑下列因素。

（1）文件属性是否正确。

（2）open/close 语句是否正确。

（3）格式说明与输入输出语句是否匹配。

（4）缓冲区大小与记录长度是否匹配。

（5）文件使用前是否已经被打开。

（6）是否处理了文件尾。

（7）是否处理了输入输出错误。

（8）输出信息中是否有文字性错误。

2. 局部数据结构测试

检查局部数据结构是为了保证临时存储在模块内的数据在程序执行过程中的完整性、正确性。局部功能是整个功能运行的基础，重点是一些函数是否被正确执行，内部是否运行正确。局部数据结构往往是错误的根源，所以应仔细设计测试用例，重点监测下面 5 类错误。

（1）不合适或不相容的类型说明。

（2）变量无初始值。

（3）变量初始化或默认值有错。

（4）不正确的变量名，如拼错或被不正确地截断等。

（5）出现上溢、下溢和地址异常。

3. 边界条件测试

边界条件测试是单元测试中最重要的一项任务。众所周知，软件经常在边界上失效，

采用边界值分析技术，针对边界值及其左、右设计测试用例，很有可能会发现新的错误。边界条件测试是一项基础测试，也是后面系统测试中功能测试的重点，边界条件测试执行得较好，可以大大提高程序的稳健性。

4. 模块中所有独立路径测试

在模块中应对每条独立执行路径进行测试，单元测试的基本任务是保证模块中每条语句至少被执行一次。测试目的主要是发现因错误计算、不正确的比较和不适当的控制流所造成的错误。具体做法就是程序员逐条调试语句，同时进行比较判断。常见的错误包括以下 9 种：

(1) 不同数据类型的对象之间进行了比较运算。
(2) 错误地使用逻辑运算符或优先级错误。
(3) 变量初始值错误。
(4) 表达式符号错误。
(5) 因计算机表示的局限性，将理论上相等而实际上不相等的两个量视为相等。
(6) 比较运算符或变量出错。
(7) 循环终止条件或不可能成立。
(8) 迭代发散时不能退出。
(9) 错误地修改了循环变量。

5. 各种错误处理测试

程序在遇到异常情况时不应该退出，好的程序应能预见各种出错条件，并预设各种出错处理通路。如果用户不按照正常流程操作，程序就退出或者停止工作，这实际上也是一种缺陷，因此单元测试要测试各种错误处理路径。一般这种测试着重检查下列问题。

(1) 输出的出错信息难以被用户理解。
(2) 记录的错误与实际遇到的错误不相符。
(3) 在程序自定义的出错处理段运行之前，系统已介入并处理了错误(如崩溃)。
(4) 异常处理不当。
(5) 错误陈述中未能提供足够的可用于定位出错的信息。

3.2.5　单元测试的特点

单元测试具有以下特点。

1. 独立性

单元测试是针对代码单元的独立测试。因为其独立性，所以可以做到针对代码单元设计完整的测试数据，覆盖代码单元的所有功能逻辑，从而在根本上保证代码单元的质量。

正因为它的独立性，所以每个程序员都可以在编写或修改代码的同时进行单元测试。即使项目刚刚开始编码，或者当前代码所依赖的其他代码还不存在，甚至项目所依赖的软

硬件环境都不完整时，仍然可以进行单元测试。

2. 查找低级错误

基本的单元测试可以在系统测试之前把大部分比较低级的错误都排除，减少系统测试过程中的问题，同时也减少了在系统测试中定位和解决问题的时间成本。

3. 找出潜在的缺陷

一些缺陷靠系统测试是很难找到的，例如，一些代码分支平时 99％的场景基本上都运行不到，但一旦运行到了，如果没有提前测试好，那么可能就是一个灾难。

4. 提供验证的功能

程序中的每项功能都需要测试来验证它的正确性，为以后的开发提供支持，使项目即使进行到开发后期，也可以轻松地增加功能或更改程序结构，而不用担心这个过程中会破坏重要内容，这便需要单元测试为代码的重构提供保障。在此基础上，开发人员可以更自由地对程序进行改进。

5. 设计文档

编写单元测试将要求测试人员从调用者的角度观察、思考。特别是先写测试用例，将迫使开发者把程序设计成易于调用和可测试的，即迫使开发者解除软件中的耦合。同时，单元测试是一种无价的文档，是展示函数或类如何使用的最佳文档。这份文档是可编译、可运行的，并且它保持最新，永远与代码同步。

6. 具有可回归性

自动化的单元测试避免了代码出现回归，在编写完成之后，可以随时随地快速地运行测试。

3.3 集成测试

在完成每个模块的单元测试之后，接下来的集成测试就将各个模块结合起来，初步地将之组成一个完整的系统，在全局的基础上对整个系统进行下一步的测试。

3.3.1 集成测试的定义

集成测试也称组装测试或联合测试。在单元测试的基础上，将所有模块按照设计要求（如根据结构图）组装成为子系统或系统进行集成测试。集成测试最简单的形式是把两个已经测试过的单元组合成一个组件，测试它们之间的接口。从这一层意义上讲，组件是指多个单元的集成聚合。在现实方案中，许多单元组合成组件，而这些组件又聚合为程序的更大部分。集成测试的基本方法是测试片段的组合，并最终扩展成进程，将模块与其他组的模块一起测试。最后，将构成进程的所有模块一起测试。此外，如果程序由多个进程

组成，那么应该成对地测试它们，而不是同时测试所有进程。

实践表明，一些模块虽然能够单独工作，但并不能保证它们连接起来也能正常工作。一些局部反映不出来的问题，很可能会暴露在全局中。

3.3.2　集成测试和单元测试的关系

单元测试用例通常是由开发者编写的，目的是验证一段相对较小的代码是做什么的，它们的范围狭窄，易于编写和执行，其有效性取决于开发者认为有用的东西。集成测试可以覆盖整个应用程序，将各个模块组合在一起，证明系统的不同部分可以协同工作。与一组单元测试相比，在集成测试环境的范围内，如生产环境中，集成测试在证明系统工作上更具说服力。实际上，集成测试可以用于各种各样的环境，从针对类似生产环境进行的全面系统测试，到使用数据库或队列的任何测试。集成测试既可以是一个 JUnit 测试，针对内存数据库对存储库进行测试；也可以是一个系统测试，验证应用程序可以交换消息。

单元测试具有集成测试没有的独立性优势。与此同时，单元测试是针对彼此分开的模块单元单独进行测试，这样会难于测试与其他代码和依赖环境之间的相互关系。单元测试与集成测试形成了互补关系。不同阶段进行的测试使得软件系统更加趋于完善，符合用户的需求。

表 3.1 中给出了单元测试和集成测试的具体区别。

表 3.1　单元测试和集成测试区别

项　　目	单 元 测 试	集 成 测 试
测试对象	模块	单元
测试时间	开发开始	开发过程
测试方法	黑盒、白盒	黑盒、白盒、灰盒
测试内容	具体模块	模块间接口
测试目的	检查代码错误	发现接口错误
测试角度	开发人员	测试人员

3.3.3　集成测试的目标

集成测试的目标是确保各模块组装在一起后能够正常地协作运行。一个或者多个模块运行良好并不足以保证整个系统的质量，因为有许多隐蔽的失效是在高质量模块间发生非预期交互而产生的。集成测试主要有两个目的。

1. 验证接口是否与设计相符

（1）在把各个模块连接起来的时候，验证穿越模块接口的数据是否会丢失。

（2）验证全局数据结构是否有问题，会不会被异常修改，还需要使用黑盒测试技术针对被测模块的接口规格说明进行测试。

2. 验证集成后的功能

(1) 将各个子功能组合起来,验证其能否达到预期要求的父功能。

(2) 验证一个模块的功能是否会对另一个模块的功能产生不利的影响。

(3) 验证单个模块的误差积累起来是否会被放大,从而达到不可接受的程度。

集成测试的必要性在于一些模块虽然能够单独工作,但并不能保证其在该连接起来也能正常工作。程序在进行单元测试时表现良好,并不代表在进行集成测试时一定能够全部运行通过,也许其可能暴露出更多的问题。此外,在某些开发模式中(如迭代式开发),设计和实现是迭代进行的,在这种情况下,集成测试的意义还在于它能间接地验证概要设计是否具有可行性。

3.3.4 集成测试的方法

进行集成测试通常有以下两种方法:非渐增式集成和渐增式集成。非渐增式集成测试是先分别测试每个模块,再把所有模块按照设计要求放在一起结合成所需要的程序并予以测试的方法。渐增式集成测试则是把下一个要测试的模块和已经测试好的模块结合起来进行测试,同时完成单元测试和集成测试。

1. 集成测试的辅助模块

要想进行集成测试,必须知道辅助模块的概念。辅助模块分为驱动模块和桩模块,表3.2中给出了驱动模块和桩模块的概念和功能。

表 3.2 辅助模块

名　称	概　念	功　能
驱动模块	用以模拟待测模块的上级模块	在集成测试中接受测试数据,把相关的数据传送给待测模块,启动待测模块,并输出相应的结果
桩模块	存根程序,用以模拟待测模块工作过程中所调用的模块	由待测模块调用,它们一般只进行很少的数据处理,以便于检验待测模块与其下级模块的接口

2. 非渐增式集成

非渐增式集成又称一次性集成,其首先对每个子模块进行单元测试,然后将所有的模块全部集成起来进行一次性测试。图3.7是非渐增式集成测试的过程,图3.8是非渐增式集成测试的程序结构图。

非渐增式集成测试可以并行地测试所有的模块,需要的测试用例少,并且测试的方法简单易行。但是由于其不可避免地存在模块间接口、全局数据结构等方面的问题,因此一次运行成功的可能性不大,即使集成测试通过,其也会出现很多的遗漏。如果一次集成的模块数量过多,集成后可能会出现大量的错误。另外,修改了一处错误后很有可能会新增更多的错误,新旧错误混杂,会给程序带来更大的麻烦。

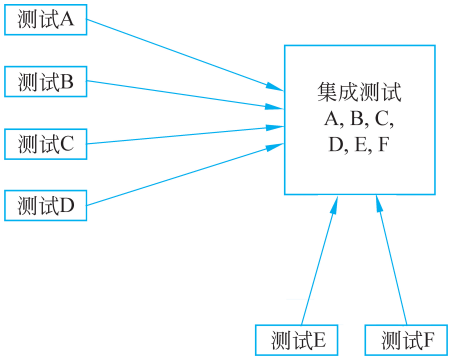

图 3.7　非渐增式集成测试的过程

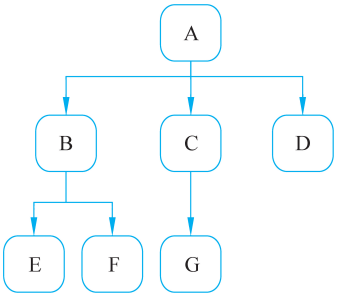

图 3.8　非渐增式集成测试的程序结构图

大爆炸集成方式是一种典型的非渐增式集成测试，也称一次性组装或者整体拼装。该集成正是把所有系统组件一次性集合到被测系统中的集成方式，其并不考虑组件之间的相互依赖性以及可能存在的风险，目的就是在最短的时间内把系统组装起来，并且通过最少的测试用例来验证整个系统。在有利的情况下，大爆炸集成可以迅速地完成集成测试，并且只需要极少数的驱动模块和桩模块以及最少的测试用例，同时其可以并行开展测试，对人力、物力的资源利用率较高。

这种在单元测试的基础上将所有组件一次性进行组装，不考虑组件之间的依赖性的集成方式虽然简单，但是在系统规模较大、模块间接口复杂的情况下会造成错误集中爆发，同时在发现错误时问题定位和修改都比较困难。即使被测系统能够被一次性集成，但仍会存在一部分接口问题可以躲过集成测试而进入系统测试。

3. 渐增式集成

渐增式集成测试一般包括自底向上集成测试、自顶向下集成测试、三明治集成测试等，下面进行简单介绍。

1）自底向上集成测试

自底向上集成测试是使用最广泛的集成方法。该方法是从软件程序结构中最底层、最基础的模块开始组装和测试，如图 3.9 所示。正是由于这种组装方式是由底向上进行的，所以对于一定高度和层次的模块来说，其下层模块（即子模块）包括子模块的下层模块已经经过组装和测试，因此其不再需要桩模块（即模拟被测试的模块所调用的模块）了，仅需要调用这些模块的驱动模块。

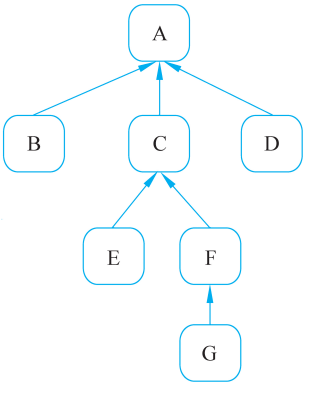

图 3.9　自底向上集成测试的程序结构图

自底向上集成测试的步骤大致如下。

（1）按照概要设计规格说明书进行规划，明确有哪些被测模块。在熟悉被测模块性质的基础上对被测模块进行分层，然后排列出测试活动的先后关系，制订测试的进度计划。根据各项测试工作之间的时间序列关系可以发现，处

于同一层次的测试工作可以同时进行，而不会相互影响。

（2）按时间线序关系，将软件单元集成为模块，并在集成过程中测试出现的问题。这里，可能需要测试人员开发一些驱动模块来驱动集成活动中形成的被测模块。对于比较大的模块，可以先将其中的几个软件单元集成为子模块，然后再将之集成为一个较大的模块。

（3）将各软件模块集成为子系统。检测各自子系统是否能正常工作。这同样可能需要测试人员开发少量的驱动模块来驱动被测子系统。

（4）将各子系统集成为最终用户系统，测试各分系统能否在最终用户系统中正常工作。

如图 3.10 所示，d1、d2、d3、d4、d5、d6 为桩模块，其可被用于从每层最底部模块进行测试，然后一步一步向上集成，直至完成测试。

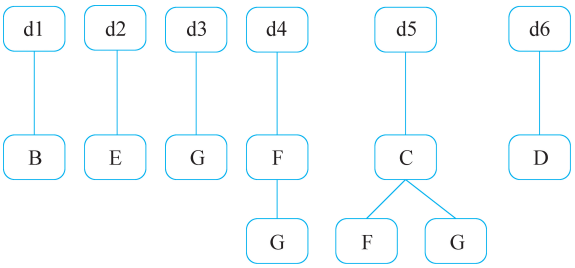

图 3.10　自底向上集成测试流程

自底向上集成支持对底层模块的早期验证，可在任何一个叶节点就绪的情况下进行集成测试。其可以并行集成，对被测模块可测性要求比自顶向下集成略低，减少了开发桩模块的工作量，同时支持故障隔离。自底向上集成的缺点是驱动模块开发量大，其针对高层的测试被推迟到最后，整体设计的错误被发现得比较晚，集成到顶层时将变得越来越复杂。

2）自顶向下集成测试

自顶向下集成测试是一个递增的组装软件结构的方法。按照控制的结构来说，其将从主要控制模块（简称主控模块）开始沿控制层向下移动，把模块一一组合起来，如图 3.11 所示。在主控模块的附属子模块进行集成时主要有深度优先和广度优先两种集成方法。

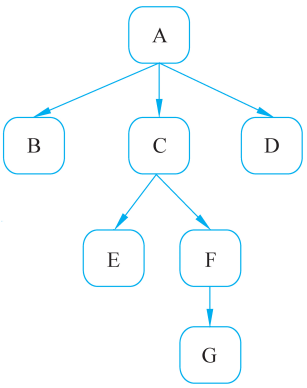

图 3.11　自顶向下集成测试的程序结构图

（1）深度优先是指先将程序结构中一条主要控制路径上的所有模块一层一层地向下集成起来，类似于树状数据结构的深度优先搜索，之后再将其他路径的模块集成起来，进行连接，路径的选取与系统应用的特性有关。

（2）广度优先即从结构的顶层开始，将这一层的所有模块集成起来，之后再往下一层继续进行集成，类似于树状数据结构的广度优先搜索，这样逐层集成直至结束。

自顶向下集成方式的组装过程分以下 5 个步骤。

（1）用主控模块作为测试驱动程序，其直接下属模块

用承接模块来代替。

(2) 根据所选择的集成测试方法,每次用实际模块代替下属的承接模块。

(3) 在组合每个实际模块时都要进行测试。

(4) 完成一组测试后再用一个实际模块代替另一个承接模块。

(5) 可以进行回归测试,即重新再进行所有的或者部分已做过的测试,以保证不引入新的错误。

例 3.1　分别以广度优先方法和深度优先方法实现自顶向下集成测试,程序结构如图 3.11 所示。

(1) 广度优先方法。

特点:从上到下分层,从左到右排序。

首先对图 3.11 分层并且从上到下排序。

第 1 层:A。

第 2 层:B,C,D。

第 3 层:E,F。

第 4 层:G。

其次对每层进行细分,从左到右排序。

第 1 层排序后:A。

第 2 层排序后:B,C,D。

第 3 层排序后:E,F。

第 4 层排序后:G。

最后经过整合,得出以广度优先方的自顶向下集成测试方法的测试次序,其先从主控模块 A 开始,沿着最后的排序结果(即 A,B,C,D,E,F,G 的次序)向下移动,逐步把各个模块组合起来进行测试。

(2) 深度优先方法。

特点:从左到右分支,从上到下排序。

首先对图 3.11 进行从左到右分支。

第 1 分支:A→B 分支。

第 2 分支:A→C→E 分支。

第 3 分支:A→C→F→G 分支。

第 4 分支:A→D 分支。

其次对各分支进行从上到下排序。

第 1 分支排序后:A,B,E。

第 2 分支排序后:A,B,F。

第 3 分支排序后:A,C,G。

第 4 分支排序后:A,D。

最后经过整合,遵循先左右,后上下的原则,最终的测试顺序为 A,B,E,F,C,G,D。

自顶向下集成测试方法能够在测试的早期对主控模块进行检验,深度优先的结合策略可以在早期实现验证软件的一个完整功能。其缺点是没有底层返回来的真实数据流,

需要推迟许多需要真实数据支持的测试。

3）三明治集成测试

三明治集成也称混合式集成，是把系统划分为 3 层，中间层为目标层。测试时对目标层的上一层使用自顶向下的集成策略，对目标层的下一层使用自底向上的集成策略，最后测试在目标层会合。图 3.12 展示了三明治集成测试的测试步骤，该测试的程序结构图如图 3.13 所示。

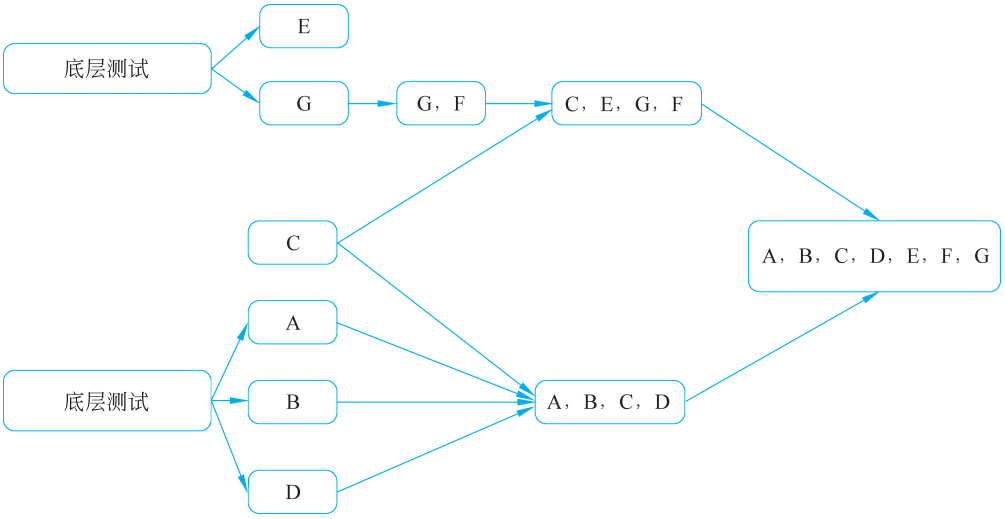

图 3.12　三明治集成测试的测试步骤

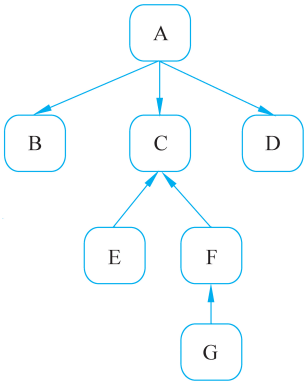

图 3.13　三明治集成测试集成的程序结构图

首先选择分界模块，在此选择以 C 模块为界，对模块 C 层（即 BCD 层）上面使用自顶向下集成测试，模块 C 层下面则使用自底向上集成测试，对 C 层使用独立测试。

这种集成方式综合了自顶向下集成测试和自底向上集成测试的优点。但是，其存在的问题是中间层选择困难，如果中间层在被集成前测试不充分，则其将影响最终的测试结果。但目前这种集成测试方法仍然适用于大部分软件开发项目。

3 种集成测试方式的比较结果如表 3.3 所示。

表 3.3　3 种集成测试方法的比较

项　　目	自顶向下集成测试	自底向上集成测试	三明治集成测试
主要程序工作时间	早	晚	早
是否需要驱动模块	否	是	是
是否需要桩模块	是	否	是
工作并行性	差	良好	良好
特殊路径测试	难	容易	中等
计划和控制	难	容易	难

3.3.5　集成测试的过程

根据 IEEE 标准，集成测试可被划分为 5 个阶段，即计划阶段、设计阶段、实施阶段、执行阶段和评估阶段。集成测试的过程如图 3.14 所示。

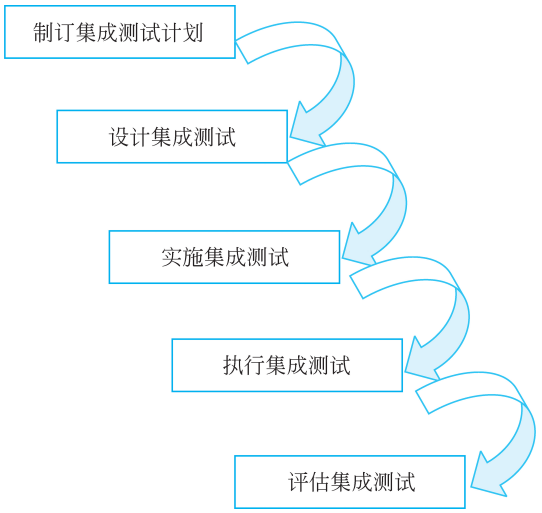

图 3.14　集成测试的过程

1. 计划阶段

（1）时间安排：概要设计完成评审后大约一个星期。

（2）输入：软件需求规格说明书、概要设计规格说明书、产品开发计划路标。

（3）实施条件：概要设计规格说明书已经通过评审。

（4）活动步骤：确定被测试对象和测试范围；评估集成测试中被测试对象的数量及难度，即工作量；确定角色分工和任务；标识测试各阶段的时间、任务和约束等条件；考虑一定的风险分析和应急计划；考虑和准备集成测试需要的测试工具、测试仪器和测试环境等资源；考虑外部技术支援的力度和深度，以及相关培训安排；定义测试完成的判定标准。

（5）输出：集成测试计划。

（6）完成条件：集成测试计划通过概要设计阶段基线评审。

2. 设计阶段

（1）时间安排：从详细设计阶段开始。

（2）输入：软件需求规格说明书、概要设计规格说明书以及集成测试计划。

（3）实施条件：概要设计阶段基线通过评审。

（4）活动步骤：分析被测对象的结构；分析集成测试模块；分析集成测试接口；分析集成测试策略；分析集成测试工具；分析集成测试环境；评估和安排集成测试的工作量。

（5）输出：集成测试设计。

（6）完成条件：集成测试设计通过详细设计阶段基线评审。

3. 实施阶段

（1）时间安排：在编码阶段开始后进行。

（2）输入：软件需求规格说明书、概要设计规格说明书、集成测试计划以及集成测试设计。

（3）实施条件：详细设计阶段。

（4）活动步骤：设计集成测试用例；设计集成测试代码、脚本；设计集成测试工具。

（5）输出：集成测试用例、集成测试规程、集成测试代码、集成测试脚本和集成测试工具。

（6）完成条件：测试用例和测试规程通过编码阶段基线评审。

4. 执行阶段

（1）时间安排：单元测试完成后就可以开始执行集成测试了。

（2）输入：软件需求规格说明书、概要设计规格说明书、集成测试计划、集成测试用例、集成测试规程、集成测试代码、集成测试脚本、集成测试工具、详细设计代码以及单元测试报告。

（3）实施条件：单元测试阶段已经通过基线评审。

（4）活动步骤：执行集成测试用例，回归集成测试用例，进行集成测试报告的撰写。

（5）输出：集成测试报告。

（6）完成条件：集成测试报告通过集成测试阶段基线评审。

5. 评估阶段

（1）时间安排：集成测试计划测试结果。

（2）阶段条件：集成测试计划测试结果等。

（3）行动指南：相关人员包括测试设计人员、编码人员、系统设计人员等对测试结果进行评估，确定是否通过测试。

（4）阶段成果：测试评估摘要。

3.4　确 认 测 试

集成测试完成以后，分散开发的模块已经按照设计要求组装成了一个完整的软件系统，各模块之间存在的种种问题都已基本排除。为进一步验证软件的有效性，对其在功能、性能、接口以及限制条件等方面做出更切实的评价，需要进行确认测试。

3.4.1　确认测试的定义

确认测试是检验组合测试的软件是否严格遵循有关标准的一种符合性测试，其需要确定软件产品是否满足所规定的要求，确保这些软件能够正常运行在系统目标设备的介质上。如果能够达到这一要求，则可以认为开发的软件是合格的。因而可将确认测试称为合格性测试或者有效性测试。

测试工作要从用户观点出发，由一个独立的组织执行。图 3.15 给出了确认测试的体系结构。

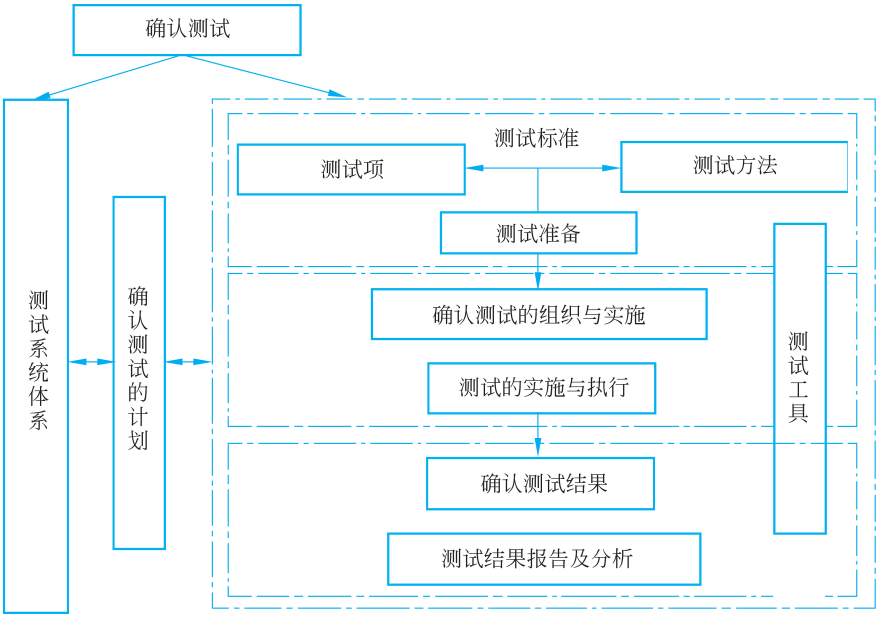

图 3.15　确认测试的体系结构

从阶段上来说，确认测试是在完成集成测试之后，即分散开发的模块被连接起来，已经构成完整的程序，其中各模块之间接口存在的问题都已消除后进行的。此时需要根据确认测试准则，针对软件需求规格说明来进行测试，以确认所开发的软件系统满足规定的功能和性能需求。

软件确认测试通常采用黑盒测试法，其测试内容主要包括功能测试、性能测试、可靠性测试等。同时由于确认测试针对的是用户的直接需求，所以其应该在尽可能真实的环境中进行，目前大部分确认测试都是在开发场景下进行的。

3.4.2 确认测试基本方法

实现软件确认测试要通过一系列黑盒测试，验证被测软件是否满足软件需求规格说明书列出的要求。常用的方法有以下 6 种。

（1）等价类划分：把程序的输入域划分成数据类，据此可以导出测试用例，一个理想的测试用例应该能独自发现一类错误。

（2）边界值分析：确定程序处理的边界情况，设计使程序运行在边界情况附近的测试方案，如下标、纯量、数据结构和循环等边界附近。

（3）错误猜测：在很大程度上靠直觉运行，其基本想法是列举出程序中可能有的错误和容易发生错误的特殊情况，并据此选择测试用例。

（4）因果图：是一种通过基于多种条件组合的图形来产生相应测试用例的方法。

（5）状态转换测试：是利用状态转换图来描述系统设计，指导测试人员设计测试用例的方法。其特点是先前输入的结果在接收当前输入的情况下，发生状态转换。

（6）判定表：列举所有可能的条件（即输入）和所有可能的活动（即输出）的表格。将表中每个条件组合都视为一个规则，通过对规则的查找，最终定位到规则列所指示的活动，此活动就是所需实施的行为。

3.4.3 确认测试的内容

常规确认测试一般需要测试以下测试项。

（1）功能度：测试目标软件是否实现了软件需求规格说明书中规定的一切功能，并找出其尚未实现的功能需求。

（2）性能测试：主要表现在效率和资源占用方面，例如，测试软件运行效率、数据处理的响应时间，在软件安装卸载前后以及运行期间对计算机资源的占用情况，包括软件对内存的占用率、对 CPU 的占用率以及占用的硬盘空间等指标。实时系统和嵌入式系统尤其要进行性能测试。另外，性能测试有时需要与强度测试相结合，经常需要其他软硬件的配套支持。

（3）安全性：测试软件系统在安全管理、数据安全、权限管理以及防止对程序和数据的非授权，故意或意外访问的能力等有关的软件属性。

（4）配置复审：保证软件配置的所有成分都齐全，各方面的质量都符合要求，具有维护阶段所必需的细节，而且已经编排好分类的目录。

（5）其他方面：测试软件的兼容性、可移植性、易用性、可扩充性和用户文档完备性等。具体考察系统在软件、硬件及数据格式的兼容性方面的应用表现；考察移植程序至另一个硬件配置或软件系统环境需要付出的成本；考察评定软件的易学易用性，各个功能是否易于完成，软件界面是否友好等；测试软件产品是否留有与异种数据的接口，是否满足业务模块的扩充需求；考察软件用户文档是否齐全，着重测试软件相关文档的文字描述与软件实际功能的一致性程度和文档的易理解程度等。

3.4.4 确认测试过程

确认测试的具体工作和步骤如图 3.16 所示，其包括有效性测试以及软件配置复审两