第 23 章

Software Engineering: A Practitioner's Approach, Eighth Edition

测试传统的应用软件

要点浏览

概念：一旦生成了源代码，就必须对软件进行测试，以便在交付给客户之前尽可能多地发现（和改正）错误。我们的目标是设计一系列极有可能发现错误的测试用例。但是，如何做呢？这就是软件测试技术发挥作用的地方。这些技术为设计测试提供系统化的指导：（1）执行每个软件构件的内部逻辑和接口；（2）测试程序的输入和输出域以发现程序功能、行为和性能方面的错误。

人员：在测试的早期阶段，软件工程师完成所有的测试。然而，随着测试过程的进行，测试专家可能介入。

重要性：评审及其他软件质量保证活动可以且确实能够发现错误，但只有这些做法是远远不够的。每次执行程序时，用户都在测试它。因此，在程序交付给客户之前，就必须以发现并消除错误为目的来执行它。为了尽可能多地发现错误，必须系统化地执行测试，而且必须利用严格的技术来设计测试用例。

步骤：对于传统的应用软件，可从两个不同的视角测试软件：（1）利用“白盒”测试用例设计技术执行程序内部逻辑；（2）利用“黑盒”测试用例设计技术确认软件需求。用例可辅助测试的设计，在软件确认的层面发现错误。在每种情况下，其基本意图都是以最少的工作量和最少的时间来发现最大数量的错误。

工作产品：设计一组测试用例使其不仅测试内部逻辑、接口、构件协作，还测试外部需求，并形成文档。定义期望结果，并记录实际结果。

质量保证措施：当开始测试时，改变视角，努力去“破坏”软件！规范化地设计测试用例，并对测试用例进行周密的评审。另外，评估测试覆盖率并追踪错误检测活动。

关键概念

基本路径测试
黑盒测试
边界值分析
控制结构测试
环路复杂性
等价类划分
流图
图矩阵
基于图的测试方法

对于本质上具有建设性的软件工程师来说，测试展示出的是有趣的异常现象。测试要求开发者首先抛弃“刚开发的软件是正确的”这一先入为主的观念，然后努力去构造测试用例来“破坏”软件。Beizer[Bei90] 有效地描述了这种情况：

> 有这样一个神话：若我们确实擅长编程，就应当不会有错误。只要我们确实很专注，只要每个人都使用结构化编程，采用自顶向下的设计方法……那么就不应该有错误。所以才有了这样的神话。神话中讲道：由于我们并不擅长所做的事，因此有错误存在。若不擅长，就应当感到内疚。因此，测试和测试用例的设计是对失败的承认，也是失败的一剂良药。测试的枯燥是对我们犯下的错误的处罚。为什么被罚？由于我们是人类？为

什么内疚？由于没能达到非人的完美境界？由于没能区分另一个程序员所想的和所说的之间存在的差异？由于没有心灵感应？由于没有解决交流问题？……由于人类四千年历史的缘故？

测试应该灌输内疚感吗？测试真的是摧毁性的吗？这些问题的回答是“不”！

关键概念
基于模型的测试
正交数组测试
模式
白盒测试

本章针对传统的应用软件讨论软件测试用例设计技术。测试用例设计关注创建测试用例的一系列技术，这些测试用例的设计符合总体测试目标及第 22 章所述的测试策略。

23.1　软件测试基础

测试的目标是发现错误，并且好的测试发现错误的可能性较大。因此，软件工程师在设计与实现基于计算机的系统或产品时，应该想着可测试性。同时，测试本身必须展示一系列特征，达到以最少工作量发现最多错误的目标。

引述　每个程序都做对某件事，可是那恰恰不一定是我们想让它做的事情！
作者不详

可测试性。James Bach[⊖]为可测试性提供了下述定义：“软件可测试性就是（计算机程序）能够被测试的容易程度。”可测试的软件应具有下述特征。

提问　可测试性的特征是什么？

可操作性。“运行得越好，越能有效地测试。”若设计和实现系统时具有质量意识，那么妨碍测试执行的错误将很少，从而使测试顺利进行。

可观察性。“你所看见的就是你所测试的。”作为测试的一部分所提供的输入会产生清楚的输出。测试执行期间系统状态和变量是可见的或可查询的，不正确的输出易于识别，内部错误会被自动检测和报告，源代码是可访问的。

可控制性。“对软件控制得越好，测试越能被自动执行和优化。”通过输入的某些组合可以产生所有可能的输出，并且输入 / 输出格式是一致的和结构化的。通过输入的组合，所有代码都可以执行到。测试工程师能够控制软硬件的状态和变量，能够方便地对测试进行说明、自动化执行和再现。 497

可分解性。“通过控制测试范围，能够更快地孤立问题，完成更灵巧的再测试。”软件由能够进行单独测试的独立模块组成。

简单性。“需要测试的内容越少，测试的速度越快。”程序应该展示功能简单性（例如，程序特性集是满足需求的最低要求）、结构简单性（例如，将体系结构模块化以限制错误的传播）以及代码简单性（例如，采用编码标准以使代码易于审查和维护）。

引述　软件中的错误比其他技术中的错误更普通、普遍且更烦人。
David Parnas

稳定性。“变更越少，对测试的破坏越小。”软件的变更不经常发生，变更发生时是可以控制的，且不影响已有的测试，软件失效后得到良好恢复。

易理解性。“得到的信息越多，进行的测试越灵巧。”体系结构设计以及内部构件、外部构件和共享构件之间的依赖关系能被较好地理解。技术文档可随时获取、组织合理、具体、详细且准确。设计的变更要通知测试人员。

可以使用 Batch 所建议的属性来开发易于测试的软件工作产品。

⊖ 后面几段取得了 James Bach（copyright 1994）的使用许可，并对最初出现在新闻组 comp.software-eng 的资料进行了改编。

测试特征。关于测试本身有哪些特征呢？ Kaner、Falk 和 Nguyen[Kan93] 提出“好”的测试具有以下属性。

提问　什么才是“好”的测试？

好的测试具有较高的发现错误的可能性。为达到这个目标，测试人员必须理解软件并尝试设想软件怎样才能失败。

好的测试是不冗余的。测试时间和资源是有限的，执行与另一个测试有同样目标的测试是没有意义的。每个测试都应该有不同的目标（即使是细微的差别）。

好的测试应该是“最佳品种”[Kan93]。在一组具有类似目的的测试中，时间和资源的有限性会迫使只运行最有可能发现所有类别错误的测试。

好的测试应该既不太简单也不太复杂。尽管将一系列测试连接为一个测试用例有时是可
498 能的，但潜在的副作用会掩盖错误。通常情况下，应该独立执行每个测试。

SafeHome　设计独特的测试

[场景] Vinod 的工作间。

[人物] Vinod 与 Ed，SafeHome 软件工程团队成员。

[对话]

Vinod：这些是你打算用于测试操作 passwordValidation 的测试用例吗？

Ed：是的，它们应该能覆盖用户进入时所有可能输入的密码。

Vinod：让我看看……你提到正确的密码是 8080，对吗？

Ed：嗯。

Vinod：你指定密码 1234 和 6789 是要测试在识别无效密码方面的错误？

Ed：对，我也测试与正确密码相接近的密码，如 8081 和 8180。

Vinod：那是可行的。但是我并不认为运行 1234 和 6789 两个输入有多大意义。这两个输入是冗余的……它们在测试同样的事情，不是吗？

Vinod：确实是这样。倘若输入 1234 不能发现错误，换句话说，操作 passwordValidation 指出它是无效密码，那么输入 6789 也不可能显示任何新的东西。

Ed：我明白你的意思。

Vinod：我不是吹毛求疵，只是我们做测试的时间有限，因此，好的方法是运行最有可能发现新错误的测试。

Ed：没问题……我再想想。

23.2　测试的内部视角和外部视角

引述　在设计测试用例中只有一条规则，那就是覆盖所有特征，但并不创建太多的测试用例。

Tsuneo Yamaura

任何工程化的产品（以及大多数其他东西）都可以采用以下两种方式之一进行测试：（1）了解已设计的产品要完成的指定功能，可以执行测试以显示每个功能是可操作的，同时，查找在每个功能中的错误；（2）了解产品的内部工作情况，可以执行测试以确保“所有的齿轮吻合”——即内部操作依据规格说明执行，而且对所有的内部构件已进行了充分测试。第一种测试方法采用外部视角，也称为黑盒测试；第二种方法采用内部视角，也称为白盒测试。⊖

黑盒测试暗指在软件接口处执行测试。黑盒测试检查系统的功能方面，而不考虑软件的内部结构。软件的白盒测试是基于过程细节的封闭检查。通过提供检

⊖ 术语功能测试和结构测试有时分别用于代替黑盒测试和白盒测试。

查特定条件集或循环的测试用例，测试将贯穿软件的逻辑路径和构件间的协作。

关键点　只有在构件级设计（或源代码）存在之后，才设计白盒测试。此时，一定要获得程序的逻辑细节。

乍一看，好像是全面的白盒测试将获得“100% 正确的程序”。我们需要做的只是识别所有的逻辑路径，开发相应的测试用、例执行测试用例并评估结果；即生成测试用例，彻底地测试程序逻辑。遗憾的是，穷举测试存在某种逻辑问题，即使对于小程序，可能的逻辑路径的数量也可能非常大。然而，不应该觉得白盒测试不切实际而抛弃这种方法。可以选择并测试有限数量的重要逻辑路径，检测重要数据结构的有效性。

499

信息栏　穷举测试

考虑 100 行的 C 语言程序。一些基本的数据声明之后，程序包含两个嵌套循环，依靠输入指定的条件，每个从 1 到 20 次进行循环。在内部循环中，需要 4 个 if-then-else 结构。这个程序中大约有 1014 个可能的执行路径！

为了说明这个数字代表的含义，我们假设已经开发了一个神奇的测试处理器（“神奇”意味着没有这样的处理器存在）来做穷举测试。在 1 毫秒内，处理器可以开发一个测试用例、执行测试用例并评估测试结果。若处理器每天工作 24 小时，每年工作 365 天，要对这个程序做完穷举测试，需要工作 3170 年。不可否认，这将对大多数的开发进度造成巨大障碍。

因此，可以肯定地说，对于大型软件系统，穷举测试是不可能的。

23.3　白盒测试

引述　bug 潜伏在角落并在边界处聚集。
Boris Beizer

白盒测试有时也称为玻璃盒测试或结构化测试，是一种测试用例设计方法，它利用作为构件级设计的一部分所描述的控制结构来生成测试用例。利用白盒测试方法导出的测试用例可以：（1）保证一个模块中的所有独立路径至少被执行一次；（2）对所有的逻辑判定均需测试取真（true）和取假（false）两个方面；（3）在上下边界及可操作的范围内执行所有循环；（4）检验内部数据结构以确保其有效性。

23.4　基本路径测试

基本路径测试是由 Tom McCabe[McC76] 首先提出的一种白盒测试技术。基本路径测试方法允许测试用例设计者计算出过程设计的逻辑复杂性测量，并以这种测量为指导来定义执行路径的基本集。执行该基本集导出的测试用例保证程序中的每一条语句至少执行一次。

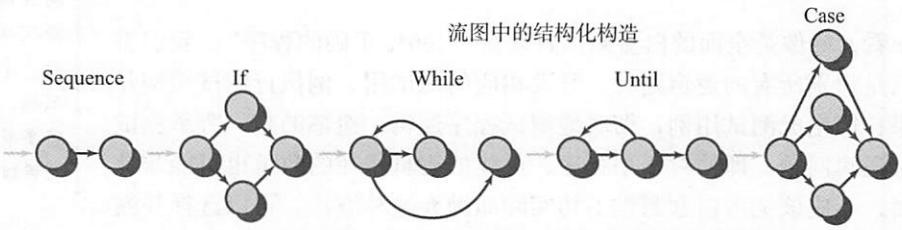
23.4.1　流图表示

在介绍基本路径方法之前，必须介绍一种简单的控制流表示方法，称为流图（或程序图）[⊖]。流图利用图 23-1 所示的表示描述逻辑控制流。每种结构化构造（第 14 章）都有相应

500

⊖ 事实上，不使用流图也可以执行基本路径测试方法，但是，流图是用于理解控制流和解释方法的一种有用表示。

的流图符号。



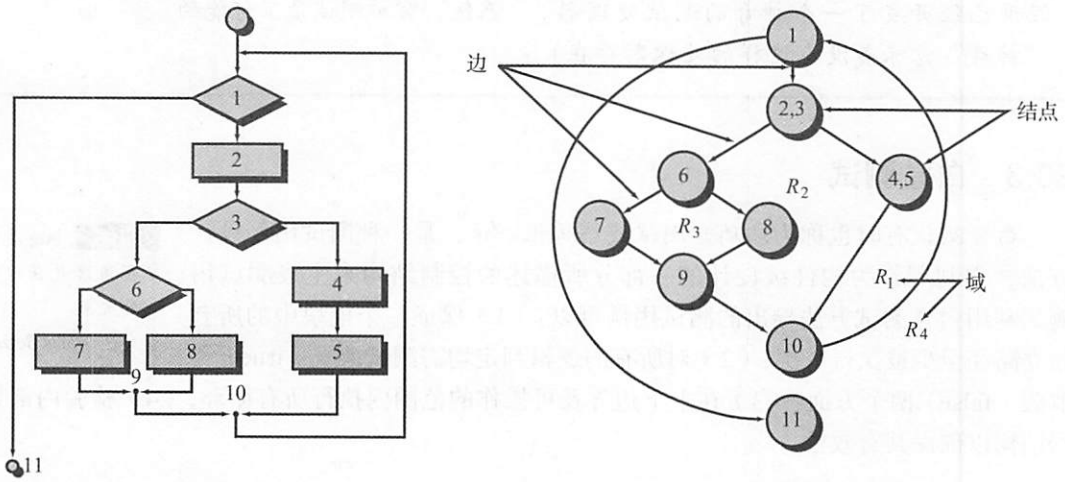
图中每个圆圈表示一个或多个无分支的 PDL 语句或源代码语句

图 23-1　流图表示

为了说明流图的使用，考虑图 23-2a 所示的过程设计表示。这里，流程图用于描述程序的控制结构。图 23-2b 将这个流程图映射为相应的流图（假设流程图的菱形判定框中不包含复合条件）。在图 23-2b 中，圆称为流图结点（flow graph node），表示一个或多个过程语句。处理框序列和一个菱形判定框可以映射为单个结点。流图中的箭头称为边或连接，表示控制流，类似于流程图中的箭头。一条边必须终止于一个结点，即使该结点并不代表任何过程语句（例如表示 if-then-else 结构的流图符号）。由边和结点限定的区域称为域。计算域时，将图的外部作为一个域[⊖]。

建议　仅在构件的逻辑结构复杂的情况下，才应该画流图。使用流图可以更轻易地追踪程序路径。

501



a) 流程图　　b) 流图

图 23-2　流程图和流图

在过程设计中遇到复合条件时，流图的生成会变得稍微复杂一些。当一个条件语句中存在一个或多个布尔运算符（逻辑 OR、AND、NAND、NOR）时，复合条件就出现了。图 23-3 给出了一段程序设计语言（PDL）程序及其对应的流图。注意，分别为条件语句“IF a OR b”的每个条件（a 和 b）创建不同的结点。包含条件的结点称为判定结点，其特征是由它发射出两条或多条边。

⊖ 23.6.1 节将更详细地讨论图及其使用。

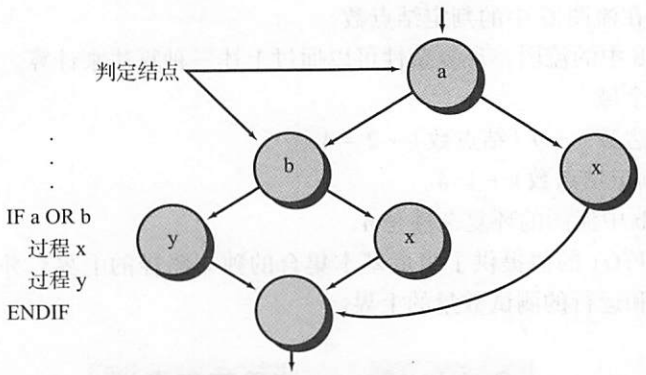


图 23-3　复合逻辑

23.4.2　独立程序路径

独立路径是任何贯穿程序的、至少引入一组新处理语句或一个新条件的路径。当按照流图进行描述时，独立路径必须沿着至少一条边移动。这条边在定义该路径之前未被遍历。例如，图 23-2b 所示流图的一组独立路径如下：

路径 1: 1-11

路径 2: 1-2-3-4-5-10-1-11

路径 3: 1-2-3-6-8-9-10-1-11

路径 4: 1-2-3-6-7-9-10-1-11

注意，每条新的路径引入一条新边，路径

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

502

不是一条独立路径，因为它不过是已提到路径的简单连接，而没有引入任何新边。

建议　在预见易于出错的模块方面，环复杂性是一种有用的度量，可以用于做测试计划以及测试用例设计。

路径 1、2、3 和 4 构成图 23-2b 所示流图的基本集合。也就是说，若设计测试以强迫执行这些路径（基本集合），则可以保证程序中的每条语句至少执行一次，且每个条件的取真和取假都被执行。应该注意到，基本集合不是唯一的。事实上，对给定的过程设计，可以导出很多不同的基本集合。

如何知道要找出多少路径？环复杂性的计算提供了答案。环复杂性是一种软件度量，它为程序的逻辑复杂度提供了一个量化的测度。用在基本路径测试方法的环境下时，环复杂性的值定义了程序基本集合中的独立路径数，并提供了保证所有语句至少执行一次所需测试数量的上限。

环复杂性以图论为基础，并提供了非常有用的软件度量。可以通过以下三种方法之一来计算环复杂性。

提问　如何计算环复杂性？

1. 流图中域的数量与环复杂性相对应。

2. 对于流图 G，环复杂性 $V(G)$ 定义如下：

$$V(G) = E - N + 2$$

其中 E 为流图的边数，N 为流图的结点数。

3. 对于流图 G，环复杂性 $V(G)$ 也可以定义如下：

$$V(G) = P + 1$$

其中 P 为包含在流图 G 中的判定结点数。

关键点　环复杂性提供保证程序中每条语句至少执行一次所需测试用例数的上界。

再回到图 23-2b 中的流图，环复杂性可以通过上述三种算法来计算。

1. 该流图有 4 个域。

2. $V(G)$ = 11（边数）− 9（结点数）+ 2 = 4。

3. $V(G)$ = 3（判定结点数）+ 1=4。

因此，图 23-2b 中流图的环复杂性是 4。

更重要的是，$V(G)$ 的值提供了组成基本集合的独立路径的上界，并由此得出覆盖所有
503 程序语句所需设计和运行的测试数量的上界。

SafeHome　使用环复杂性

[场景] Shakira 的工作间。

[人物] Vinod 和 Shakira，SafeHome 软件工程团队成员，他们正在为安全功能准备测试计划。

[对话]

Shakira：看，我知道应该对安全功能的所有构件进行单元测试，但是，如果考虑所有必须测试的操作的数量，工作量就太大了，我不知道……可能我们应该放弃白盒测试，将所有的构件集成在一起，开始执行黑盒测试。

Vinod：你估计我们没有足够的时间做构件测试、检查操作，然后集成，是不是？

Shakira：第一次增量测试的最后期限离我们很近了……是的，我有点担心。

Vinod：你为什么不对最有可能出错的操作执行白盒测试呢？

Shakira（愤怒地）：我怎么能够准确地知道哪个是最易出错的呢？

Vinod：环复杂性。

Shakira：嗯？

Vinod：环复杂性。只要计算每个构件中每个操作的环复杂性。看看哪些操作的 $V(G)$ 具有最高值。那些操作就是最有可能出错的操作。

Shakira：怎么计算 $V(G)$ 呢？

Vinod：那相当容易。这里有本书说明了怎么计算。

Shakira（翻看那几页）：好了，这计算看上去并不难。我试一试。具有最高 $V(G)$ 值的就是要做白盒测试的候选操作。

Vinod：但还要记住，这并不是绝对的，那些 $V(G)$ 值低的构件还是可能有错的。

Shakira：好吧。但这至少降低了必须进行白盒测试的构件数。

23.4.3　生成测试用例

基本路径测试方法可以应用于过程设计或源代码。在本节中，我们将基本路径测试描述为一系列步骤。以图 23-4 中用 PDL 描述的过程 average 为例，说明测试用例设计方法中的各个步骤。注意，尽管过程 average 是一个非常简单的算法，但却包含了复合条件与循环。下列步骤可用于生成基本测试用例集。

引述　犯错误的是人，发现错误的是神。
Robert Dunn

1. **以设计或源代码为基础，画出相应的流图**。利用 23.4.1 节给出的符号和构造规则创建流图。参见图 23-4 中过程 average 的 PDL 描述，将那些 PDL 语句进行编号，并映射到相应的流图结点，以此来创建流图。图 23-5 给出了相应的流图。

504

> **引述**　只是由于在将64位浮点值转换为16位整数的操作中包含了一个软件缺陷（代码错误），Ariane 5型火箭在升空时发生爆炸。这枚火箭和它的4颗卫星都没有投保，它们价值5亿美元。如果进行路径测试是可以发现这个错误的，但由于预算原因被否决了。
>
> *一篇新闻报道*

2. **确定所得流图的环复杂性**。通过运用 23.4.2 节描述的算法来确定环复杂性 $V(G)$ 的值。应该注意到，不建立流图也可以确定 $V(G)$，方法是通过计算 PDL 中条件语句的数量（过程 average 的复合条件语句计数为 2），然后加 1。在图 23-5 中：

   $V(G)$ = 6(域数)

   $V(G)$ = 17(边数)−13(结点数) + 2 = 6

   $V(G)$ = 5(判定结点数) + 1 = 6

3. **确定线性独立路径的基本集合**。$V(G)$ 的值提供了程序控制结构中线性独立路径的数量。在过程 average 中，我们指定了 6 条路径：

   路径 1: 1-2-10-11-13

   路径 2: 1-2-10-12-13

   路径 3: 1-2-3-10-11-13

   路径 4: 1-2-3-4-5-8-9-2-...

   路径 5: 1-2-3-4-5-6-8-9-2-...

   路径 6: 1-2-3-4-5-6-7-8-9-2-...

路径 4、5、6 后面的省略号（...）表示可加上控制结构其余部分的任意路径。在设计测试用例的过程中，经常通过识别判定结点作为导出测试用例的辅助手段。本例中，结点 2、3、5、6 和 10 为判定结点。

```
PROCEDURE average;

  *  This procedure computes the average of 100 or fewer
     numbers that lie between bounding values; it also computes the
     sum and the total number valid.

  INTERFACE RETURNS average, total.input, total.valid;
  INTERFACE ACCEPTS value, minimum, maximum;

  TYPE value[1:100] IS SCALAR ARRAY;
  TYPE average, total.input, total.valid;
     minimum, maximum, sum IS SCALAR;
  TYPE i IS INTEGER;
1 i = 1;
  total.input = total.valid = 0;
  sum = 0;
  DO WHILE value[i] <> –999 AND total.input < 100      3
4   increment total.input by 1;
    IF value[i] > = minimum AND value[i] < = maximum    6
5     THEN increment total.valid by 1;
7           sum = s sum + value[i]
      ELSE skip
8   ENDIF
    increment i by 1;
9 ENDDO
  IF total.valid > 0      10
11  THEN average = sum / total.valid;
12    ELSE average = –999;
13 ENDIF
END average
```

图 23-4　已标识结点的 PDL

4. **准备测试用例，强制执行基本集合中的每条路径**。测试人员应该选择测试数据，以便在测试每条路径时适当地设置判定结点的条件。执行每个测试用例并将结果与期望值进行比较。一旦完成了所有的测试用例，测试人员就可以确信程序中所有的语句至少已被执行一次。

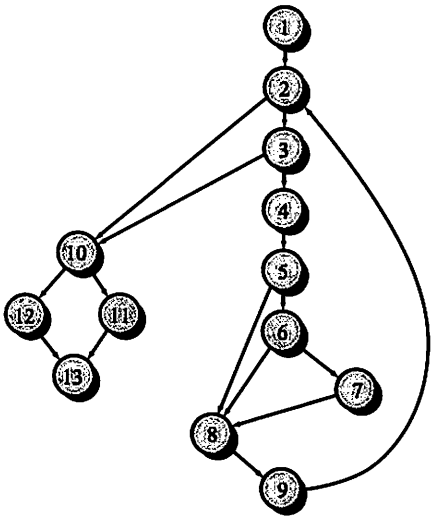


图 23-5　过程 average 的流图

注意，某些独立路径（本例中的路径 1）不能单独进行测试。也就是说，遍历路径所需的数据组合不能形成程序的正常流。在这种情况下，将这些路径作为另一个路径的一部分进行测试。

23.4.4　图矩阵

导出流图甚至确定基本路径集合的过程都可以机械化。一种称为图矩阵（graph matrix）的数据结构对于开发辅助基本路径测试的软件工具相当有用。

图矩阵是一种方阵，其大小（即行与列的数量）等于流图的结点数。每行和每列都对应于已标识的结点，矩阵中的项对应于结点间的连接（边）。图 23-6 给出了一个简单流图及相应的图矩阵 [Bei90]。

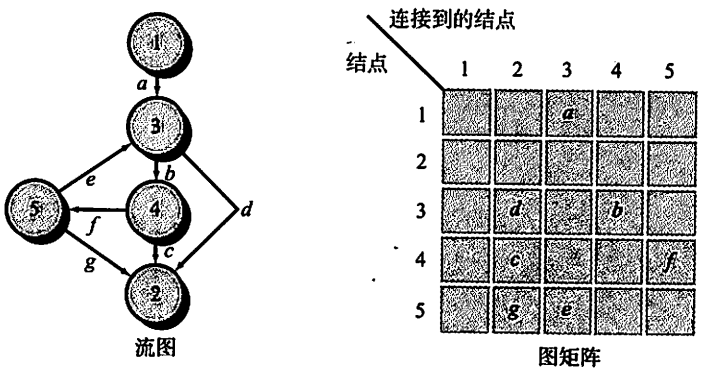


图 23-6　图矩阵

如图 23-6 所示，流图的每个结点用数字标识，而每条边用字母标识。矩阵中的每个字母对应于纵横方向两结点间的连接，例如，边 *b* 连接结点 3 和结点 4。

从这种意义上讲，图矩阵只是流图的表格表示。然而，通过为每个矩

提问　什么是图矩阵？如何对其进行扩展以用于测试？

阵项加入一个*连接权值*，图矩阵将成为测试期间评估程序控制结构的一个强有力的工具。连接权值提供了有关控制流的附加信息。最简单的情况下，连接权值是 1（连接存在）或 0（连接不存在），但是，可以赋予连接权值其他更有意义的特征：

505 ~ 506

- 执行连接（边）的概率。
- 遍历连接的处理时间。
- 遍历连接时所需要的内存。
- 遍历连接时所需要的资源。

Beizer[Bei90] 提供了可用于图矩阵的其他数学算法的全面讨论。利用这些技术，设计测试用例时所需进行的分析可以部分或完全自动化。

23.5　控制结构测试

引述　将执行测试看得比设计测试更重要是一个典型的错误。
Brian Marick

23.4 节所描述的基本路径测试是控制结构测试技术之一。虽然基本路径测试简单且高效，但其本身并不充分。本节简单讨论控制结构测试的其他变体，这些技术拓宽了测试的覆盖率并提高了白盒测试的质量。

条件测试 [Tai89] 通过检查程序模块中包含的逻辑条件进行测试用例设计。*数据流测试* [Fra93] 根据程序中变量的定义和使用位置来选择程序的测试路径。

*循环测试*是一种白盒测试技术，完全侧重于循环构建的有效性。可以定义 4 种不同的循环 [Bei90]：简单循环、串接循环、嵌套循环和非结构化循环（图 23-7）。

507

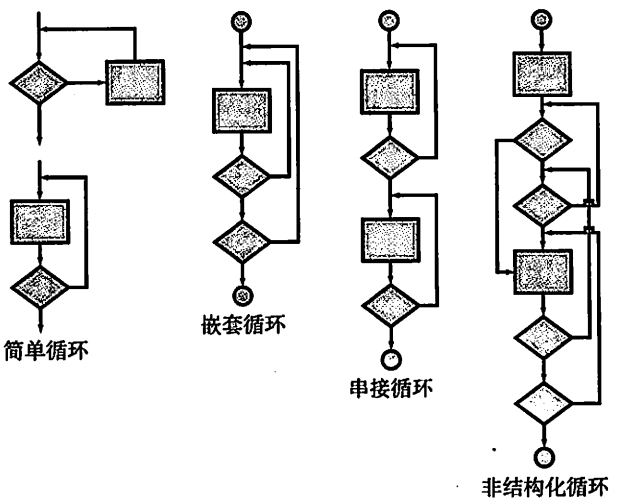


图 23-7　循环的类别

引述　优秀的测试人员是注意到“奇怪的事情”就会对它采取行动的大师。
Brian Marick

简单循环。下列测试集可用于简单循环，其中，n 是允许通过循环的最大次数。

1. 跳过整个循环。
2. 只有一次通过循环。
3. 两次通过循环。
4. m 次通过循环，其中 $m < n$。
5. $n-1$、n、$n+1$ 次通过循环。

嵌套循环。若将简单循环的测试方法扩展应用于嵌套循环，则可能的测试数将随着嵌套层次的增加而成几何级数增长。这将导致不切实际的测试数量。Beizer[Bei90] 提出了一种有助于减少测试数的方法。

1. 从最内层循环开始，将其他循环设置为最小值。
2. 对最内层循环执行简单循环测试，而使外层循环的迭代参数（例如循环计数）值最小，并对范围以外或不包括在内的值增加其他测试。
3. 由内向外构造下一个循环的测试，但使其他外层循环具有最小值，并使其他嵌套循环为“典型”值。
4. 继续上述过程，直到测试完所有的循环。

串接循环。若串接循环的每个循环彼此独立，则可以使用简单循环测试方法。然而，若两个循环串接起来，且第一个循环的循环计数为第二个循环的初始值，则这两个循环并不独立。若循环不独立，则建议使用嵌套循环的测试方法。

508

建议　不能对非结构化循环进行有效测试，需要对它们进行重新设计。

非结构化循环。若有可能，应该重新设计这类循环以反映结构化程序结构的使用（第 14 章）。

23.6　黑盒测试

黑盒测试也称行为测试或功能测试，侧重于软件的功能需求。黑盒测试使软件工程师能设计出可以测试程序所有功能需求的输入条件集。黑盒测试并不是白盒测试的替代品，而是作为发现其他类型错误的辅助方法。

黑盒测试试图发现以下类型的错误：（1）不正确或遗漏的功能；（2）接口错误；（3）数据结构或外部数据库访问错误；（4）行为或性能错误；（5）初始化和终止错误。

与白盒测试不同，白盒测试在测试过程的早期执行，而黑盒测试倾向于应用在测试的后期阶段（第 22 章）。黑盒测试故意不考虑控制结构，而是侧重于信息域。设计黑盒测试要回答下述问题：

提问　什么是黑盒测试必须回答的问题？

- 如何测试功能的有效性？
- 如何测试系统的行为和性能？
- 哪种类型的输入会产生好的测试用例？
- 系统是否对特定的输入值特别敏感？
- 如何分离数据类的边界？
- 系统能承受什么样的数据速率和数据量？
- 特定类型的数据组合会对系统运行产生什么样的影响？

通过运用黑盒测试技术，可以生成满足下述准则的测试用例集 [Mye79]：能够减少达到合理测试所需的附加测试用例数，并且能够告知某些错误类型是否存在，而不是仅仅知道与特定测试相关的错误。

23.6.1　基于图的测试方法

黑盒测试的第一步是理解软件中建模的对象[⊖]及这些对象间的关系。这一步一旦完成，

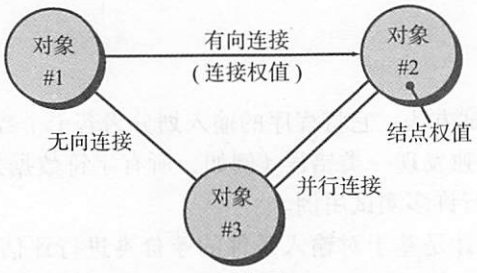
⊖ 这里，我们在最广泛的环境中考虑术语“对象”。它包括数据对象、传统的构件（模块）以及计算机软件的面向对象元素。

下一步就是定义一系列验证“所有对象之间具有预期关系”的测试 [Bei95]。换言之，软件测试首先是创建重要对象及其关系图，然后设计覆盖图的一系列测试用例，使得图中的每个对象和关系都测试到，并发现错误。 509

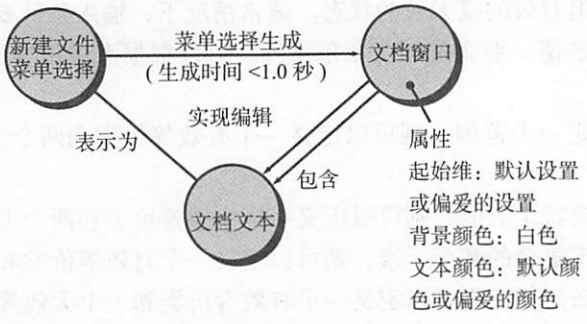
为完成这些步骤，软件工程师首先要创建图，其中结点表示对象，连接表示对象间的关系，结点权值描述结点的属性（例如，具体的数据值或状态行为），连接权值描述连接的某些特征。

关键点　图表示数据对象与程序对象间的关系，它使我们能够设计测试用例，查找与这些关系有关的错误。

图的符号表示如图 23-8a 所示。结点用圆表示。而连接有几种形式，有向连接（用箭头表示）表示这种关系只在一个方向存在，双向连接（也称对称连接）表示关系适用于两个方向，并行连接表示图结点间有几种不同的关系。



a)　图表示



b)　一个简单的例子

图 23-8　图符号表示及示例

考虑一个简单的例子，字处理应用中图的一部分，如图 23-8b 所示，其中

对象 #1 = 新建文件（菜单选择）
对象 #2 = 文档窗口
对象 #3 = 文档文本

该图中，选择菜单“新建文件”生成一个“文档窗口”。“文档窗口”的结点权值提供窗口生成时预期的属性集。连接权值表明必须在 1.0 秒之内生成。一条无向连接在“新建文件”菜单选择和“文档文本”之间建立对称关系。并行连接显示“文档窗口”与“文档文本”间的关系。事实上，设计测试用例还需要更详细的图描述。然后软件工程师通过遍历图并覆盖图中所示的关系来设计测试用例。这些测试用例用于发现各种关系中的错误。Beizer[Bei95] 描述了下面几种使用图的行为测试方法。 510

事务流建模。结点表示事务的步骤（例如，利用联机服务预订机票所需的步骤）。连接

表示这些步骤间的逻辑连接。例如，数据对象“航班信息输入”的后面跟着“确认有效性的处理”操作。

有限状态建模。结点表示用户可见的不同软件状态（例如，订票人员处理电话订票时的各个屏幕），连接表示状态间的转换（例如，在“库存有效性检查”期间，“订单信息”会得到验证，之后会输入“客户账单信息”）。状态图（第 11 章）可用于辅助创建这种图。

数据流建模。结点表示数据对象，而连接为一个数据对象转换为其他数据对象时发生的变换。例如，结点**扣缴税款**（FTW）由总工资（GW）利用关系 $FTW = 0.62 \times GW$ 计算出来。

时间建模。结点为程序对象，连接是对象间的顺序连接。连接权值用于指定程序执行时所需的执行时间。

基于图的测试方法的详细讨论超出了本书的范围。感兴趣的读者参看 [Bei95]，可以对其有全面的了解。

23.6.2　等价类划分

511

等价类划分是一种黑盒测试方法，它将程序的输入划分为若干个数据类，从中生成测试用例。理想的测试用例可以单独发现一类错误（例如，所有字符数据处理不正确），否则在观察到一般的错误之前需要运行许多测试用例。

等价类划分的测试用例设计是基于对输入条件的等价类进行评估。利用上节引入的概念，若对象可以由具有对称性、传递性和自反性的关系连接，则存在等价类 [Bei95]。等价类表示输入条件的一组有效的或无效的状态。通常情况下，输入条件要么是一个特定值、一个数据域、一组相关的值，要么是一个布尔条件。可以根据下述指导原则定义等价类。

提问　如何为测试定义等价类？

1. 若输入条件指定一个范围，则可以定义一个有效等价类和两个无效等价类。
2. 若输入条件需要特定的值，则可以定义一个有效等价类和两个无效等价类。
3. 若输入条件指定集合的某个元素，则可以定义一个有效等价类和一个无效等价类。
4. 若输入条件为布尔值，则可以定义一个有效等价类和一个无效等价类。

通过运用设计等价类的指导原则，可以为每个输入域数据对象设计测试用例并执行。选择测试用例以便一次测试一个等价类的尽可能多的属性。

23.6.3　边界值分析

大量错误发生在输入域的边界处，而不是发生在输入域的“中间”。这是将*边界值分析*（Boundary Value Analysis，BVA）作为一种测试技术的原因。边界值分析选择一组测试用例检查边界值。

引述　测试代码的一种有效方式是在其自然边界处运行它。
Brian Kernighan

边界值分析是一种测试用例设计技术，是对“等价划分”的补充。BVA 不是选择等价类的任何元素，而是在等价类“边缘”上选择测试用例。BVA 不是仅仅侧重于输入条件，它也从输出域中导出测试用例 [Mye79]。

关键点　通过侧重考虑一个等价类“边界”处的数据，BVA 扩展了等价类划分。

BVA 的指导原则在很多方面类似于等价划分的原则。

1. 若输入条件指定为以 a 和 b 为边界的范围，则测试用例应该包括 a 和 b，略大于和略小于 a 和 b。
2. 若输入条件指定为一组值，则测试用例应当执行其中的最大值和最

小值，以及略大于和略小于最大值和最小值的值。

3. 指导原则 1 和 2 也适用于输出条件。例如，工程分析程序要求输出温度和压强的对照表，应该设计测试用例创建输出报告，输出报告可生成所允许的最大（和最小）数目的表项。 512

4. 若内部程序数据结构有预定义的边界值（例如，表具有 100 项的定义限制），则一定要设计测试用例，在其边界处测试数据结构。

大多数软件工程师会在某种程度上凭直觉完成 BVA。通过运用这些指导原则，边界测试会更加完全，从而更有可能发现错误。

23.6.4　正交数组测试

关键点　正交数组测试使得软件工程师设计的测试用例能够以合理的数量提供最大的测试覆盖。

许多应用程序的输入域是相对有限的。也就是说，输入参数的数量不多，且每个参数可取的值有明确的界定。当这些数量非常小时（例如，3 个输入参数，取值分别为 3 个离散值），则有可能考虑每个输入排列，并对所有的输入域进行测试。然而，随着输入值数量的增加及每个数据项的离散值数量的增加，穷举测试将是不切实际或不可能的。

正交数组测试（orthogonal array testing）可以应用于输入域相对较小但对穷举测试而言又过大的问题。正交数组测试方法对于发现区域错误（region fault）（有关软件构件内部错误逻辑的一类错误）尤其有效。

为说明正交数组测试与更传统的“一次一个输入项”方法之间的区别，考虑有 3 个输入项 X、Y 和 Z 的系统。每个输入项有 3 个不同的离散值。这样可能有 $3^3 = 27$ 个测试用例。Phadke[Pha97] 提出了一种几何观点，用于组织与 X、Y 和 Z 相关的测试用例。如图 23-9 所示，一个输入项一次可能沿着某个输入轴在顺序上有变化。这导致了相对有限的输入域覆盖率（图 23-9 中左图立方体所示）。 513

使用正交数组测试时，创建测试用例的一个 L9 正交数组。L9 正交数组具有“平衡特性”[Pha97]，即测试用例（图中表示为黑点）“均匀地分散在整个测试域中”，如图 23-9 中右图立方体所示。这样整个输入域的测试覆盖会更完全。

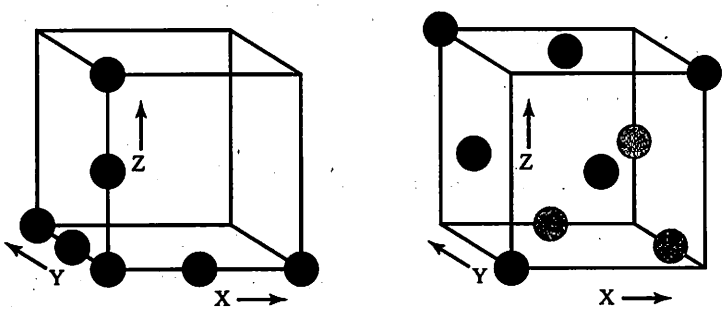


图 23-9　测试用例的几何视图 [Pha97]

为了说明 L9 正交数组的使用，考虑传真应用中的 send 函数。向函数 send 传递 4 个参数 P1、P2、P3 和 P4。其中每个参数取 3 个不同的值。例如，P1 的取值：

P1= 1，现在发送

P1= 2，一小时后发送

P1=3，半夜 12 点后发送

P2、P3 和 P4 也分别取值 1、2 和 3，表示其他发送功能。

如果选择“一次一个输入项”的测试策略，则测试（P1, P2, P3, P4）的测试序列如下：（1,1,1,1），（2,1,1,1），（3,1,1,1），（1,2,1,1），（1,3,1,1），（1,1,2,1），（1,1,3,1），（1,1,1,2），（1,1,1,3）。但是这些测试数据只会揭示单模式错误 [Pha97]，也就是说，这些错误由一个参数触发。

给定相对少量的输入参数和离散值，穷举测试是可能的。所需要的测试数为 $3^4 = 81$，虽比较大，但还是能够做到的。可以发现所有与数据项排列相关的错误，但所需的工作量较大。

正交数组测试方法使我们可以提供较好的测试覆盖，而测试用例比穷举测试少得多。send 函数的 514 L9 正交数组如图 23-10 所示。

测试用例	测试参数			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

图 23-10　L9 正交数组

Phadke[Pha97] 对利用 L9 正交数组测试方法的测试结果评价如下。

检测和分离所有单模式错误。单模式错误是任意单个参数在任意级别上的一致性问题。例如，若因子 P1 = 1 的所有测试用例产生一个错误条件，则它就是一个单模式错误。在这个例子中，测试 1、2 和 3（图 23-10）将显示错误。通过分析哪些测试显示了错误的信息，可以识别出是哪个参数值产生了错误。在这个例子中，注意到测试 1、2 和 3 产生错误，因而可以将其分离（有关“现在发送（P1 = 1）”的逻辑处理）为错误源。这样的错误分离对于修改错误是很重要的。

检测所有双模式错误。若当两个参数的特定级别一起出现时存在一致性问题，则称之为双模式错误。实际上，双模式错误表示成对不相容问题或两个测试参数间的有害干扰问题。

多模式错误。（所显示类型的）正交数组仅可以保证单模式和双模式错误的检测。然而，有些多模式错误也可以通过这些测试检测出来。

正交数组测试的详细讨论见 [Pha89]。

软件工具　测试用例设计

[目标] 辅助软件团队设计完整的黑盒测试和白盒测试用例集。

[机制] 这些工具可分为静态测试与动态测试两大类。在产业界，有三种不同类型的静态测试工具：基于代码的测试工具、基于专用测试语言的测试工具和基于需求的测试工具。基于代码的测试工具所接收的输入为源代码，完成一系列分析，最后生成测试用例。专用测试语言（例如 ATLAS）使软件工程师能够书写详细的测试规格说明，在规格说明中，描述每个测试用例及其执行逻辑。基于需求的测试工具将特定用户需求进行分离，为检查需求的测试用例（或测试类）的设计提出建议。动态测试工具与执行程序进行交互，检查路径覆盖，测试特定变量值的断言，或监

测程序的执行流。

[代表性工具]⊖

- McCabe Test。由 McCabe & Associates（www.mccabe.com）开发，它实现了一些从环复杂性评估和其他软件度量中派生的路径测试技术。
- TestWorks。由 Software Research（http://www.testworks.com/stwhome.html）开发，它是一套完整的自动化测试工具，有助于开发 C、C++ 和 Java 软件的测试用例设计，并为回归测试提供支持。
- T-VEC Test Generation System。由 T-VEC Technologies（www.t-vec.com）开发，它是支持单元测试、集成测试和确认测试的工具集。通过使用面向对象需求规格说明中的信息辅助测试用例的设计。
- e-Test Suite。由 Empirix（www.empirix.com）开发，拥有测试 WebApp 的完整工具集，包括辅助测试用例设计工具和测试计划工具。

515

23.7　基于模型的测试

引述　当你在代码中寻找错误时，很难发现它；当你认为自己的代码没有错误时，就更难发现它。
Steve McConnell

基于模型的测试（Model-based Testing，MBT）是一种黑盒测试技术，它使用需求模型中的信息作为生成测试用例的基础 [DAC03]。在很多情况下，基于模型的测试技术使用 UML 状态图——一种行为模型（第 11 章）作为测试用例设计的基础⊖。MBT 技术需要以下 5 个步骤。

1. **分析软件的已有行为模型或创建一个行为模型。**回忆一下，行为模型指明软件是如何响应外部事件或刺激的。为了创建行为模型，我们需要执行第 11 章所讨论的步骤：（1）评价所有的用例，以完全理解系统内的交互顺序；（2）标识驱动交互顺序的事件，并理解这些事件如何与特定的对象相关；（3）为每个用例创建交互顺序；（4）构造系统的 UML 状态图（例如，见图 11-1）；（5）评审行为模型，验证其精确性和一致性。
2. **遍历行为模型，并标明促使软件在状态之间进行转换的输入。**输入将触发事件，使转换发生。
3. **评估行为模型，并标注当软件在状态之间转换时所期望的输出。**回想一下，每个转换都由一个事件触发，作为转换的结果，某些方法会被调用并产生输出。对于步骤 2 所指定的每个输入（用例）集合，指定所期望的输出，以说明它们在行为模型中的特点。
4. **运行测试用例。**可以手工执行测试，也可以创建测试脚本并使用测试工具执行测试。
5. **比较实际结果和期望结果，并根据需要进行调整。**

MBT 可帮助我们发现软件行为中的错误，因此，它在测试事件驱动的应用时也非常有用。

23.8　文档测试和帮助设施测试

软件测试一词造成一种假象：大量的测试用例是为检查计算机程序和它们所管理的数据做准备的。但是，帮助设施或文档中的错误与数据或源代码中的错误一样，它们都会影响程序的验收。完全按照用户指南或在线帮助进行操作，但得到的结果或行为却与文档的描述不

516

⊖ 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。

⊖ 当软件需求是用决策表、语法或 Markov 链表示时，也可以使用基于模型的测试 [DAC03]。

符，没有什么比这种情况更让人沮丧了。因此，文档测试应该是所有软件测试计划中有意义的一部分。

文档测试可分为两个阶段进行。第一阶段为技术评审（第 20 章），检查文档编辑的清晰性；第二阶段是现场测试（live test），结合实际程序使用文档。

令人意外的是，对文档的现场测试竟可以采用与前面讨论的许多黑盒测试方法相似的技术，包括：基于图的测试可用于描述程序的使用；等价类划分和边界值分析方法可用于定义各种输入类和相关的交互操作；MBT 则可用于确保文档规定的行为和实际行为的吻合。因而程序的用法可以贯穿全部文档而得到追踪。

信息栏　文档测试

在测试文档和帮助设施时，应该回答下列问题：

- 该文档准确地描述了如何完成每种使用模式吗？
- 每种交互序列的描述是否准确？
- 实例准确吗？
- 术语、菜单描述以及系统响应与实际程序一致吗？
- 在文档中能够比较容易地得到指导吗？
- 利用该文档可以容易地完成疑难解答吗？
- 该文档的目录和索引是否健壮、准确和完整？
- 该文档的设计（布局、字体、缩进、图表）有助于信息的理解与快速吸收吗？
- 所有显示给用户的软件错误信息在该文档中有更详细的描述吗？对看到错误信息后所采取的行动有明确的描述吗？
- 如果提供超文本链接，链接是否准确和完整？
- 如果提供超文本链接，导航设计是否适合信息获取？

回答这些问题唯一可行的方法是让独立的第三方（如选定的用户）在程序使用的环境下测试该文档。应该记录所有的差异，确定模糊或薄弱的地方，以方便可能的重写。

23.9　实时系统的测试

许多实时应用的时间依赖性和异步特征给测试带来了新的困难——时间。测试用例设计者不仅必须考虑传统的测试用例，而且要考虑事件处理（即中断处理）、数据的定时以及处
517 理数据的任务（进程）的并行性。在许多情况下，实时系统在一种状态下提供的测试数据可以正常处理，而在另一种状态下提供同样的数据将会出现错误。

例如，控制复印机的实时软件在机器处于复印状态时，接收操作员的中断（即机器操作员按控制键，如 RESET 或 DARKEN）不会产生错误。若同一操作员中断出现在机器处于卡纸状态时，则会显示诊断代码，指明卡纸的位置将丢失（一个错误）。

此外，实时系统的软件和硬件环境之间的密切关系也会导致测试问题。软件测试必须考虑硬件故障对软件处理的影响。这种故障很难实时模拟。

对于实时软件的测试，可以提出以下 4 个步骤的策略。

提问　测试实时系统的有效策略是什么？

任务测试。单独测试每个任务。也就是说，对每个任务设计并执行传统的测试。在测试期间，每个任务单独执行。任务测试可以发现逻辑和功能错误，但不能发现时间或行为错误。

行为测试。利用通过自动化工具创建的系统模型，是可以模拟实时系统的行为并按照外部事件序列检查其行为的。这些分析活动可以作为测试用例设计的基础。当实时软件建成时，执行这些测试用例。使用类似等价划分的技术（23.6.2 节），对事件（如中断、控制信号）进行分类测试。例如，复印机的事件可能是用户中断（如重置计数器）、机械中断（如卡纸）、系统中断（如碳粉低）及失效模式（如滚筒过热）。这些事件的每一个都要单独测试，并检查可执行系统的行为，以检测与这些事件有关的处理错误。

任务间测试。一旦单个任务和系统行为中的错误已经分离出来，测试就要转向与时间相关的错误。用不同的数据速率和处理负载来测试任务间的异步通信，以确定任务间是否发生同步错误。另外，通过消息队列和数据存储进行通信任务的测试，以发现这些数据存储区域大小方面的错误。

系统测试。集成软件与硬件并进行全范围的系统测试以发现软件 / 硬件接口处的错误。多数实时系统都能处理中断，因此，测试布尔事件的处理尤其重要。利用状态图（第 11 章），测试人员开发所有可能的中断和中断处理列表，然后设计测试以评估下列系统特征：

518

- 是否正确赋予和处理中断优先级？
- 每个中断的处理是否正确？
- 中断处理过程的性能（如处理时间）是否符合需求？
- 关键时刻若出现大量中断，是否会导致功能和性能上的问题？

另外，作为中断处理的一部分并用于传输信息的全局数据区域也应该测试，以评估产生副作用的可能性。

23.10　软件测试模式

模式作为描述特定设计问题解决方案的一种机制，其使用已经在第 16 章讨论过了。但模式也可以用于提出其他软件工程解决方案——此处为软件测试。测试模式描述常见的测试问题和解决方案，可以辅助软件工程师处理这些问题。

网络资源　软件测试模式目录可在 http://c2.com/cgi-bin/ 找到。

在过去 10 年间，大多数软件测试已是一项专门的活动。如果测试模式有助于软件测试团队对软件测试进行更有效的交流、理解采用特定测试方法的动机，以及将测试用例的设计作为一种进化活动，以使每次迭代都产生更完整的测试用例，那么测试模式就达到了预期的目的。

测试模式可以采用与设计模式（第 16 章）同样的方式进行描述。文献（例如 [BIN99]、[Mar02]）中已提出了几十种测试模式。下面的三种测试模式（仅以摘要的形式给出）是较有代表性的例子。

关键点　测试模式有助于软件团队对测试进行更有效的交流，并对采用特定测试方法的影响力有更好的理解。

模式名称：结对测试

摘要：一种面向过程的模式，结对测试描述了一种与结对编程（第 5 章）类似的技术。在这种测试模式中，两个测试人员一起设计并执行一系列测试，可以应用于单元测试、集成测试或确认测试活动中。

模式名称：独立测试接口

摘要：在面向对象系统中需要对每个类进行测试，包括“内部类”(不向使用它们的外部构件暴露任何接口的类)。独立测试接口模式描述如何创建“一个测试接口，该测试接口可用于描述一些类（这些类仅对某个内部构件可见）的特定测试”[Lan01]。

519

模式名称：场景测试

摘要：一旦已经执行了单元测试与集成测试，就需要确定软件是否能够以让用户满意的方式执行。场景测试描述一种从用户的角度测试软件的技术。在这个层次上的失败表明软件不能满足用户的可见需求 [Kan01]。

对测试模式的全面讨论超出了本书的范围。对于这个重要主题的其他信息，有兴趣的读者可以参看 [Bin99]、[Mar02] 和 [Tho04]。

23.11　小结

测试用例设计的主要目标是设计最有可能发现软件错误的测试用例集。为达到这个目标，可采用两种不同的测试用例设计技术：白盒测试和黑盒测试。

白盒测试侧重于程序控制结构。设计测试用例以保证测试期间程序中所有的语句至少被执行一次，且所有的逻辑条件都得到检查。基本路径测试是一种白盒测试技术，利用程序图（或图矩阵）生成保证覆盖率的线性无关的测试集。条件和数据流测试进一步检查程序逻辑，循环测试作为白盒测试技术的补充，检查不同复杂度的循环。

Hetzel[Het84] 将白盒测试描述为“小型测试”。他的意思是，本章所考虑的白盒测试一般应用于小的程序构件（例如模块或一小组模块）。而黑盒测试放宽了测试的焦点，可以将其称为“大型测试”。

黑盒测试用来确认功能需求，而不考虑程序的内部结构。黑盒测试技术侧重于软件的信息域，通过划分程序的输入域和输出域来设计测试用例，以提供完全的测试覆盖。等价划分将输入域划分为有可能检查软件特定功能的数据类。边界值分析则检查程序在可接受的限度内处理边界数据的能力。正交数组测试提供了一种高效的、系统的、使用少量的输入参数的测试方法。基于模型的测试使用需求模型的元素测试应用的行为。

520 有经验的软件开发人员经常说：“测试永无止境，它只不过是从软件工程师转移到用户。客户每次使用程序时都是一次测试。”通过运用测试用例设计，软件工程师可以取得更完全的测试，因此可以在“客户的测试”开始之前，发现和改正尽可能多的错误。

习题与思考题

23.1　Myers[Mye79] 用以下程序作为对测试能力的自我评估：某程序读入 3 个整数值，这 3 个整数值表示三角形的 3 条边。该程序打印信息以表明三角形是不规则的、等腰的或等边的。开发一组测试用例测试该程序。

23.2　设计并实现习题 23.1 描述的程序（适当时使用错误处理）。从该程序中导出流图并用基本路径测试方法设计测试，以保证程序中的所有语句都被测试到。执行测试用例并显示结果。

23.3　你能够想出 23.1.1 节中没有讨论的其他测试目标吗？

23.4　选择一个你最近设计和实现的构件。设计一组测试用例，保证利用基本路径测试执行所有的语句。

23.5　说明、设计和实现一个软件工具，使其能够对你所选的程序设计语言计算环复杂性。在你的设计中，利用图矩阵作为有效的数据结构。

23.6　阅读 Beizer[Bei95] 或相关的网络资源（例如，www.laynetworks. com/ Discrete%20 Mathematics_1g.htm），并确定如何扩展习题 23.5 所开发的程序以适应各种连接权值。扩展你的工具以处理执行概率或连接处理时间。

23.7　设计一个自动化测试工具，使其能够识别循环并按照 23.5.3 节中的方法分类。

23.8　扩展习题 23.7 中描述的工具，为曾经遇到的每个循环类生成测试用例。与测试人员交互地完成

这个功能是有必要的。

23.9 至少给出 3 个例子，在这些例子中，黑盒测试能够给人“一切正常”的印象，而白盒测试可能发现错误。再至少给出 3 个例子，在这些例子中白盒测试可能给人“一切正常”的印象，而黑盒测试可能发现错误。

23.10 穷举测试（即便对非常小的程序）是否能够保证程序 100% 正确？

23.11 测试你经常使用的某个应用软件的用户手册（或帮助设施）。在文档中至少找到一个错误。

扩展阅读与信息资源

实际上，所有软件测试方面的书籍都同时考虑测试策略和测试技术。因此，第 22 章的推荐读物同样适用于本章。有许多讨论测试原理、概念、策略和方法的书籍，下面的书籍只是其中的一小部分：Burnstein（《Practical Software Testing》，Springer, 2010）、Crispin 和 Gregory（《Agile Testing: A Practical Guide for Testers and Agile Teams》，Addison-Wesley，2009）、Lewis（《Software Testing and Continuous Quality Improvement》，3rd ed.，Auerbach，2008）、Ammann 和 Offutt（《Introduction to Software Testing》，Cambridge University Press，2008）、Everett 和 McCleod（《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，2006）、Loveland 和他的同事（《Software Testing Techniques》，Charles River Media，2004）、Dustin（《Effective Software Testing》，Addison-Wesley，2002）、Craig 和 Kaskiel（《Systematic Software Testing》，Artech House，2002）、Tamres（《Introducing Software Testing》，Addison-Wesley，2002）以及 Whittaker（《Exploratory Software Testing: Tips, Tricks, and Techniques to Guide Test Design》，Addison-Wesley, 2009 和《How to Break Software》，Addison-Wesley, 2002）。 521

Myers[Mye79] 经典书籍的第 3 版（《The Art of Software Testing，3rd ed.，Wiley, 2011》）由 Myers 及其同事编写，非常详细地讲述了测试用例的设计技术。Black（《Managing the Testing Process》，3rd ed., Wiley，2009）、Jorgensen（《Software Testing: A Craftsman's Approach》，3rd ed., CRC Press, 2008）、Pezze 和 Young（《Software Testing and Analysis》，Wiley, 2007）、Perry（《Effective Methods for Software Testing》，3rd ed., Wiley, 2006）、Copeland（《A Practitioner's Guide to Software Test Design》，Artech, 2003）以及 Hutcheson（《Software Testing Fundamentals》，Wiley, 2003）都提供了测试用例设计方法和技术的有用介绍。Beizer[Bei90] 的经典文本全面介绍了白盒测试技术，引入了数学级别上的严格性，这在其他测试方面的论述中一般是不具备的。他后来的书籍 [Bei95] 对重要方法作了简明介绍。

软件测试是一种资源密集型的活动。为此，许多组织为部分测试过程提供了自动化支持。Graham 和她的同事（《Experiences of Test Automation: Case Studies of Software Test Automation》，Addison-Wesley, 2012 和《Software Test Automation》，Addison-Wesley, 1999）、Li 和 Wu（《Effective Software Test Automation》，Sybex, 2004）、Mosely 和 Posey（《Just Enough Software Test Automation》，Prentice Hall, 2002）、Poston（《Automating Specification-Based Software Testing》，IEEE Computer Society, 1996）以及 Dustin、Rashka 和 Poston（《Automated Software Testing: Introduction, Management, and Performance》，Addison-Wesley, 1999）讨论了自动化测试的工具、策略和方法。Nquyen 和他的同事（《Happy About Global Software Test Automation》，Happy About Press, 2006）介绍了测试自动化的执行视图。

Meszaros（《Unit Test Patterns: Refactoring Test Code》，Addison-Wesley, 2007）、Thomas 和他的同事（《Java Testing Patterns》，Wiley, 2004）以及 Binder[Bin99] 描述了测试模式，包括方法测试、类 / 簇测试、子系统测试、可复用构件测试、框架测试、系统测试、测试自动化及特定数据库测试。

从网上可以获得大量的有关测试用例设计方法的信息。有关测试技术的最新参考文献可在 SEPA 网站 www.mhhe.com/ pressman 找到。 522