第二个需要考虑的因素是，应用程序是运行在单处理器上还是多处理器上。全对测试技术不能保证输入数据对在多处理器上都是合适的。竞争条件、事件实时发生的持续时间、异步输入顺序等因素在多处理器应用中是很常见的，而全对测试技术却难以满足这些条件。因此对动态应用来说，不论是在单处理器上还是多处理器上，都不适合采用全对测试技术。

对剩下的这个应用领域，即多处理器环境中的静态应用，是否适合于用这项技术还不是很明确。因为此类系统一般都是计算密集型的（因此需要并行处理）。若将它们放在一个处理器上执行时系统确实是静态的，那么全对测试技术还是适合的。

24.4　对全对测试的建议

全对测试仅仅是一种捷径。当用于测试的时间被不断缩减时——实践中经常是这样的，那么就会选择走捷径，这既充满诱惑也充满风险。如果对下面这些问题都能够给出肯定的回答，那么采用全对测试技术的风险就会大大降低。

- 输入是单纯的数据吗（而不是数据或事件的混合）？
- 都是逻辑变量吗（而不是物理变量）？
- 变量间是相互独立的吗？
- 变量具备有效的等价类吗？
- 输入是与顺序无关吗（特别是，应用程序是静态的、单处理器的）？

因为全对算法只能够产生测试用例的输入部分，所有最后还有一个问题：

- 能确切地给出全对测试的期望输出吗？

参考文献

Bach, J. and Schroeder, p.J., Pairwise Testing: A Best Practice That Isn't, presented at STAR West, 2003.

Berger, B., Efficient Testing with All-Pairs, presented at STAR East, 2003.

Cohen, D.M., Dalal, S.R., Kajla, A., and Patton, G.C., The Automatic Efficient Test Generator (AETG) System, in *Proceedings of the 5th International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, 1994, pp. 303-309.

Mandl, R., Orthogonal latin squares: an application of experiment design to compiler testing, *Communications of the ACM*, Vol. 28, No. 10, 1985, pp. 1054-1058.

Wallace, D.R. and Kuhn, D.R., Converting System Failure Histories into Future Win Situations, 2000, available online at http://hissa.nist.gov/effProject/handbook/failure/hase99.pdf.

Wallace, D.R. and Kuhn, D.R., Failure modes in medical device software: an analysis of 15 years of recall data, *International Journal of Reliability, Quality, and Safety Engineering*, Vol. 8, No. 4,2001, pp. 351-371.

第25章

尾声：软件测试精益求精

要结束一本书几乎和开始一本书的写作一样困难。那些无处不在的诱惑总是怂恿你回到已经写好的章节，再添一点儿什么内容，再改动点儿什么地方，或者干脆删掉一部分。这也是写书和软件开发过程相类似的地方，并且两者都在临近最后期限时使人变得有点焦躁不安。

编写本书最初的想法是与 Myers 的 *The Art of Software Testing* 一书相呼应；因此这本书最早的名字实际上是 *The Craft of Software Testing*，遗憾的是 Brian Marrick 先出版了一本同名的书。从 1978 年（Myers 的书出版）至 1995 年（本书的第 1 版），软件测试的技术与工具已经成熟起来，可以称得上是一种技艺了。

我们可以想象：这个以艺术为开端，创新发展到技艺，探索发展到科学，持续发展成为一项工程为目标的发展过程。软件测试应该属于这个过程的哪一个阶段呢？测试工具厂商当然愿意把它定位在工程阶段，并声称只要利用他们的工具，就不需要在其他阶段必需进行的思考。而强调过程的群体则试图把它定位为一门科学，声称只要遵从一种定义良好的“软件测试过程”一切就万事大吉了。上下文驱动学派可能会将软件测试视为一门艺术，因为其中需要创造性和个人天赋。就我个人而言，我始终认为软件测试是一种技艺。无论对它如何定位，软件测试所追求的永远都是“精益求精”。

25.1 软件测试是一种技艺

首先，我放弃了一个努力：因为想用一个词来确切表达清楚这其中的含义实在不是件容易事儿。这里，我们就采用“工匠”这个词的“从事某个行当的人”这个普通含义来代表“精通一种技艺的手艺人”吧。

什么东西能使一个人成为工匠？我的祖辈中有一个是在丹麦制作家具的，这个层次的木工活儿显然是一种技艺。我父亲是制作工具和模具的——这是另外一种有着非常严格标准的技艺。他们及其他被称为工匠的人都有什么共同点呢？下面列出的这些就很说明问题：

- 对用料了如指掌；
- 对工具了如指掌；
- 对技术了如指掌；
- 具有选择适合的工具和技术的能力；
- 对用料经验丰富；
- 曾经用这种材料完成过一系列高质量作品。

自从 Juran 和 Deming 的时代以来，部分软件开发社区已经将目光集中在软件质量方面。人们自然是要追求软件的高质量，但问题是软件的质量不仅难以定义，而且对软件质量的测量更

是难上加难。简单的质量属性，如简单性、可扩展性、可靠性、易测性、可维护性等都存在上述这两个问题。所有这些属性差不多都同样难以定义和测量。注重过程的群体认为：好的过程就会得到高质量的软件，但是这一点同样难以证实。高质量软件能不能以一种“特别的过程”进行开发？答案是有可能，并且敏捷社区很相信这一点。“标准”能保障软件的质量吗？这看起来也有问题。可以想象：一个程序遵守了一系列定义好的标准，但是质量却很低下。那么，人们应该如何定义软件质量呢？我认为：应用“技艺”这个概念来回答这个问题，将得到很好的答案，这也正是杰出的作品诞生的地方。真正的巨匠对自己的作品会感到骄傲——在创作中，他知道什么时候他完成了一个最好的作品，并且自豪感油然而生。对作品感到骄傲的同时也挑战着传统的行业标准，但是每个对自己诚实的人都知道什么时候他真正完成了一项漂亮的工作。因此拥有了技能、自豪感和杰作的“技艺”——我们是认同的，但是难以定义，也就更难以衡量——所有这些都是与最佳实践紧密联系在一起的。

25.2　软件测试的最佳实践

任何声称的最佳实践都有主观性，并且总会受到批判。下面给出了最佳实践应具备的一些特征。

- 通常是由专业人员定义的。
- 经过尝试并证明是正确的。
- 取决于具体的问题。
- 获得过巨大的成功。

软件开发界对如何解决软件开发中的难题已经进行了很长时间的研究。Fred Brooks 在他1986 年发表的著名论文中写道“没有灵丹妙药”，他认为软件业界永远也不可能找到一个技术来解决软件开发中的所有难题。下面列出了一部分最佳实践，其中的每一个都曾经被奉为“灵丹妙药”。各项是按照时间的大致顺序进行排列的。

- 高级编程语言（Fortran 和 COBOL）。
- 结构化程序设计。
- 第三代编程语言。
- 软件评审和审查。
- 软件开发生命周期的瀑布模型。
- 第四代编程语言（针对领域的）。
- 面向对象范型。
- 瀑布模型的各种替代模型。
- 快速原型法。
- 软件度量。
- CASE（辅助计算机软件工程）工具。
- 进行项目、变更和配置管理的商用工具。
- 集成开发环境。
- 软件过程成熟度（及其评估）。

- 软件过程改进。
- 可执行的规格说明。
- 自动代码生成。
- UML（及其变种）。
- 模型驱动开发。
- 极限编程（缩写为XP）。
- 敏捷编程。
- 测试驱动开发。
- 自动测试框架。

很不错的一个列表，不是吗？这里可能会丢失一些条目，但关键是已经说明了软件开发目前还是一项复杂艰巨的工作，并且献身于软件开发的工作者将会永不停歇地改进现有方法和探索全新的方法。

25.3 让软件测试更出色的10项最佳实践

由软件测试专家来进行软件测试，这是实现最佳测试实践的最基本假设。根据前面的讨论，这就意味着测试人员必须具有丰富的知识，而且有时间和工具来出色地完成测试任务。2006年10月在西班牙毕尔巴鄂（Bilbao）举行的QA & TEST会议上，与会人员对测试员是否应该是优秀的程序员展开了有趣的讨论。最后大家给出了一致的、肯定的回答。（对于一名测试人员来说，编程显然是应该掌握的一部分技能。）此外还需要的品质包括创造性、灵活性、好奇心、纪律性、批判性以及某些挑战性地“推翻设计”等。下面简单总结一下我认为最好的10项最佳实践，它们中的大多数都已在相应的章节中做了详细介绍。

25.3.1 模型驱动开发

模型驱动开发（MDD）对于大型的复杂应用程序来说是很好的选择。在大量商业产品中，有些产品能够生成源代码，但大多数仅能生成代码框架。对开发人员来说，不论哪种方式都是开发与标准紧密关联系统的良好开端。这里需要遵守的一个原则——更改代码就要相应地更新基本模型，否则会弄巧成拙。MDD的一个主要优势在于，如果很好使用，能够发现一些可能被忽略的细节问题。此外模型在应用程序的有效生命周期中，对维护也是非常有用的。在理想的情况下，根据Petri网或状态图思想，MDD应该包含一个能够支持可执行规格说明概念的产品。

25.3.2 慎重地定义与划分测试的层次

对任何应用程序（除非它非常小）都至少应该进行两个层次的测试——单元测试和系统测试。对更大的程序一般最好还要加上集成测试。有效地控制这些层次上的测试是至关重要的。每一层测试都有其明确的目标，并且这些目标应该是可以观察的。应用系统测试用例进行单元测试不仅荒谬可笑而且还浪费了宝贵的测试时间。

25.3.3　基于模型的系统级测试

如果模型中使用了可执行的规格说明，那么就能够自动生成大量的系统级测试用例。这样可以大大地消减用于创建可执行模型所带来的额外工作量。此外，这还能够根据需求模型直接对系统测试进行追踪。由于可执行规格说明可以引发具有争议用例的特点，所以自动生成的系统测试用例能够涵盖其他方法所不能详尽的问题。

25.3.4　系统测试的扩展

对于复杂的任务优先的应用程序来讲，单线索测试是必要的，但却是不够的，最低限度还需要进行线索间交互测试（第15章中介绍过）。尤其是在复杂系统中，线索间相互不仅非常重要而且难以识别。压力测试是确认线索交互的一种蛮力方法。很多时候只有通过真正的海量压力测试才能够发现以前无法发现的故障（Hill, 2006）。Hill指出，压力测试的注意力集中在软件中已知的（或怀疑的）弱点上，而且与其他常规测试方法相比，所给出的“测试是否通过”判断要更加主观一些。基于风险的测试可能是线索交互测试中一条必要的捷径。基于风险的测试是第14章中所讨论的操作剖面方法的延伸。它不同于仅仅测试最常用（高概率）的线索，基于风险的测试用失败开销（惩罚）来修正线索概率。当测试时间十分有限时，根据风险而不是简单的概率来测试线索。

25.3.5　利用关联矩阵指导回归测试

传统的和面向对象的软件项目的开发都受益于关联矩阵。对于过程性软件，其基本功能（有时候称为特性）和过程实现之间的关联关系会被记录在矩阵中。因此，对于软件的一个具体功能来说，能够很容易地识别出支持它所必要的一组过程。面向对象软件的情况与此类似，用例和类之间的关联关系会被记录下来。这两种情形下，关联矩阵中的信息都可以用来：

- 决定构建（或增加）的顺序和内容；
- 当发现故障时（或有故障报告）时，帮助隔离故障；
- 指导回归测试。

25.3.6　利用MM路径实现集成测试

在第13章中所给出的3种基本集成测试方法中，MM路径被证明是最好的一种。MM路径也可以与关联矩阵方法相结合，共同进行系统级测试。

25.3.7　把基于规格说明的测试和基于代码的单元级测试有机地结合起来

无论是基于规格说明的测试还是基于代码的单元测试，每一种独立而言都是不充分的，但是结合起来却能取长补短。最好的方法是根据单元的属性选择一种基于规格说明的方法（见第8章），使用某种测试工具来运行测试用例并得出测试覆盖指标，然后根据覆盖指标报告去掉冗余的测试用例，并新增测试用例以达到覆盖指标的要求。

25.3.8 基于单个单元特性的代码覆盖指标

没有“放之四海而皆准”的测试覆盖指标。最好的方法是根据源代码的特点来选择相应的覆盖指标。

25.3.9 维护阶段的探索式测试

当被测试代码不是由测试人员所编写时，探索式测试是一种强有力的方法。特别是在遗留代码的维护方面更是如此。

25.3.10 测试驱动开发

敏捷编程社区已证明：在适用敏捷方法的应用中采用测试驱动开发（TDD）方法是成功的，因为具有出色的故障隔离能力是 TDD 的最大优势。

25.4 针对不同项目实现最佳实践

最佳实践必定是依赖于项目的。管理 NASA 空间任务的软件显然与用来给某些主管领导提供所需信息的简单粗陋的程序完全不同。本节给出 3 类不同的项目类型，简单介绍每类项目的特点之后，表 25-1 给出了前面 10 项最佳实践分别适用于哪一类项目。

表25-1 适合于各种不同项目的最佳测试实践

最佳实践	任务关键型	时间关键型	遗留代码
模型驱动开发	×		
慎重定义和划分测试的层次	×	×	×
基于模型的系统级测试	×		
系统测试的扩展	×		
用关联矩阵指导回归测试	×		×
用MM路径实现集成测试	×		
将基于规格说明和基于代码的单元级测试有机结合	×		×
基于不同单元特性的代码覆盖指标	×		
维护阶段的探索式测试			×
测试驱动开发		×	

25.4.1 任务关键型项目

任务关键型项目对系统的稳定性和性能都有很严格的要求，并且通常都是很复杂的软件。项目的规模一般都很大，所以一个人不可能彻底理解整个系统以及其内部的所有相互关系。

25.4.2　时间关键型项目

尽管任务关键型项目也可能对时间有严格要求，但这里，时间关键型是指那些要求在短时间里必须快速完成的项目。尽快推向市场和避免失去市场份额是此类项目的着眼点。

25.4.3　对遗留代码的纠错维护

对现有软件的纠错维护是软件维护的一种最普通形式。纠错维护为的是处理在现有软件中发现的故障。在大多数机构中，软件维护的工作量通常会占到总编程工作量的 3/4；由于完成修改与维护工作的人员通常不是被修改代码的开发人员，因此这部分工作所占的比重还在进一步扩大。