第 1 章 软件测试入门

随着我国软件产业的蓬勃发展和软件系统的规模与复杂性的急剧增加，软件开发成本及由于软件故障造成的经济损失在不断增加，因此软件质量问题越来越被企业所重视。软件测试是保证软件质量的主要手段。近几年来，社会对软件测试人员的需求迅速增长。

1.1 软件、软件生命周期与软件缺陷

在进行软件测试之前，首先介绍几个相关的概念。

1.1.1 软件和软件生命周期

什么是软件？软件是计算机程序、程序所用的数据及有关文档资料的集合。其各组成部分可具体描述为：

（1）程序是能够完成预定功能和性能的、可执行的指令集。

（2）数据是使程序能够正确运行的数据结构。

（3）文档是程序研制过程和使用方法的描述。

像我们经常使用的 Windows、Android、Office、QQ、PhotoShop、微信、滴滴打车、百度地图等都是软件产品。

同任何事物一样，一个软件产品或软件系统也要经历孕育、诞生、成长、成熟、衰亡等阶段，这些阶段统称为软件生命周期。把整个软件生存周期划分为若干阶段，并为每个阶段规定明确的任务，可以使规模庞大、结构复杂、管理困难的软件的开发变得容易控制。通常，软件生命周期包括：

（1）问题定义。要求系统分析员与用户进行交流，弄清“用户需要计算机解决什么问题”，然后提出关于“系统目标与范围的说明”，提交用户审查和确认。

（2）可行性研究。一方面在于把待开发的系统的目标以明确的语言描述出来，另一方面从经济、技术、法律等多方面进行可行性分析。

（3）需求分析。弄清用户对软件系统的全部需求，编写需求规格说明书和初步的用户手册，提交评审。

（4）软件开发。软件开发由以下三个阶段组成。

- 软件设计：此阶段主要根据需求分析的结果，对整个软件系统进行设计，如系统框架设计、数据库设计等。软件设计一般分为总体设计和详细设计。好的软件设计将为软件程序编写打下良好的基础。
- 软件实现：此阶段是根据选定的程序设计语言，将软件设计的结果转换成计算机可运行的程序代码。在程序编码中必须制定统一、符合标准的编写规范，以保证程序的可读性、易维护性，提高程序的运行效率。
- 软件测试：在软件开发过程中和开发完成后要经过严密的测试，以发现软件中存在的问题并加以纠正。整个测试过程包括单元测试、集成测试、系统测试三个阶段。测试的方法主要有黑盒测试和白盒测试两种。在测试过程中需要建立详细的测试计划并严格按照测试计划进行测试，以减少测试的随意性。

（5）软件维护。软件维护包括以下四个方面。

- 改正性维护：在软件交付使用后，由于开发测试时的不彻底、不完全，必然会有一部分隐藏的错误被带到运行阶段，这些隐藏的错误在某些特定的使用环境下就会暴露。
- 适应性维护：是为适应环境的变化而修改软件的活动。
- 完善性维护：是根据用户在使用过程中提出的一些建设性意见而进行的维护活动。
- 预防性维护：是为了进一步改善软件系统的可维护性和可靠性，并为以后的改进奠定基础的维护活动。

从广义的角度看，软件测试是软件开发的一个子过程。从狭义的角度看，软件开发是生产制造软件，软件测试是验证开发出来软件的质量，其关系是：

（1）没有软件开发就没有测试，软件开发提供软件测试的对象。

（2）软件开发和软件测试都是软件生命周期中的重要组成部分。

（3）软件测试是保证软件开发产品质量的重要手段。

由于越早发现软件存在的问题，修正的成本越低，所以软件测试是伴随整个软件生命周期的，且其不仅要发现问题，还要纠正发现的问题。

1.1.2 软件缺陷

对于软件来讲，不论采用什么样的技术和方法，软件中都会存在缺陷。标准商业软件里也存在缺陷，只是严重的程序不同而已。虽然采用新的编程语言、先进敏捷的开发方式、完善的流程管理可以很大程度地减少缺陷的引入，但最终还是难以杜绝。正是因为软件缺陷的存在，才有了软件测试的必要性。为了更好地理解和完成软件测试，下面对软件缺陷做一介绍。

1．软件缺陷的定义

软件缺陷是指计算机系统或者程序中存在的任何一种破坏正常运行能力的问题、错误，

或者隐藏的功能缺陷、瑕疵，常用“bug”表示。缺陷会导致软件产品在某种程度上不能满足用户的需要。这里，我们介绍几个典型和最近出现的软件缺陷带来的灾难性后果。

（1）千年虫。

20 世纪 70 年代，程序员为了节约非常宝贵的内存资源和硬盘空间，在存储日期时，只保留年份的后 2 位，如“1980”被存为“80”。但是，这些程序员万万没有想到他们的程序会一直被用到 2000 年，当 2000 年到来的时候，问题出现了。比如银行的计算机可能将 2000 年解释为 1900 年，引起利息计算上的混乱，甚至自动将所有的记录消除。所以，当 2000 年快要来到的时候，为了这样一个简单的设计缺陷，全世界付出了几十亿美元的代价。

（2）“冲击波”计算机病毒。

2003 年 8 月 11 日，“冲击波”计算机病毒首先在美国发作，使美国的政府机关、企业及个人用户的成千上万台计算机受到攻击。随后，冲击波蠕虫很快在 Internet（因特网）上广泛传播，中国、日本和欧洲等国家也相继受到不断的攻击，结果使十几万台邮件服务器瘫痪，给整个世界范围内的 Internet 通信带来惨重损失。

制造冲击波蠕虫的黑客仅仅用了 3 周时间就制造了这个恶毒的程序，“冲击波”计算机病毒仅仅是利用微软 Messenger Service 中的一个缺陷，攻破计算机安全屏障，就使基于 Windows 操作系统的计算机崩溃。

（3）爱国者导弹防御系统。

美国爱国者导弹防御系统是主动战略防御系统的简化版本，它首次被用在第一次“海湾战争”中对抗伊拉克飞毛腿导弹的防御作战中，总体上看效果不错，赢得各种的赞誉。但它还是有几次未能成功拦截伊拉克飞毛腿导弹，其中一枚在多哈的飞毛腿导弹造成 28 名美国士兵死亡。分析专家发现，拦截失败的症结在于一个软件缺陷，当爱国者导弹防御系统的时钟累计运行超过 14h 后，系统的跟踪系统就不准确。在多哈袭击战中，爱国者导弹防御系统运行时间已经累计超过 100 多个小时，显然那时的跟踪系统已经很不准确，从而造成这种结果。

（4）放射性机器系统。

由于放射性治疗仪 Therac-25 中的软件存在缺陷，导致几个癌症病人受到非常严重的放射性治疗，其中 4 个人因此死亡。

（5）美国联航系统免费发放机票。

2013 年 9 月 12 日，美联航售票网站一度出现问题，售出票面价格为 0～10 美元的超低价机票，引发乘客抢购。大约 15min 后，美联航发现错误，关闭售票网站并声称正在进行维护。大约两个多小时后，该公司购票网站恢复正常，并且承认已卖出的票有效。

但是事情并没有结束，一个月后，注册常旅卡的用户在取消过程中，只需花几美元即可购买实际价值为几千美元的机票。美联航发飙了，指责发现该 bug 的用户，认为有人“有意”操作网站，因此不承认这些机票。通常，软件公司会对发现重大 bug 的用户给予一定

的奖励，但这样的事情并未在美联航身上发生。

（6）Dropbox 宕机 1h。

当你在把数据上传到公共云时，你是否担心过数据会被黑客攻击，有一天，你无法访问这些数据，虽然很恐怖，但噩梦还是变成了现实。

云端存储解决服务提供商 Dropbox 在 2013 年 5 月发生了一次宕机事件，事件持续 1h，用户无法使用 Dropbox，在页面上显示无法链接服务器。而在 2013 年，Dropbox 共发生过两次宕机事件，虽然官方回应并非遭到黑客攻击，但仍然引发不少用户的担心。

（7）CBOE 事件。

CBOE（Chicago Board Options Exchange）是美国最大的期权交易所。2013 年 4 月，CBOE 因软件故障引起延迟开盘事件，事故从早上 8:30 开始，直到中午 12 点才全部开盘。导致此事件的缺陷主要源于一个产品维护功能，是由于该功能中针对一个期权类进行标识符号改变而引起的。在事件结束后，CBOE 因监管失败被罚款 600 万美元。

（8）FSSA 信息泄露事件。

2013 年 5 月，印第安纳州家庭和社会服务管理局（FSSA）泄露了用户的私人信息，其中包括社会安全号码以及错误的收件人信息，其中大约有 188000 人的信息被公开。

FSSA 花了一个月的时间来修复这些错误，并且被公开信息的用户也的确因此受到了影响。现在，不仅用户在使用该系统时会陷入困境，而且他们更担心个人信息会被陌生人利用或者操作来攻击他们。

对于软件缺陷的准确定义，通常有以下 5 条描述。

（1）软件未实现产品说明书要求的功能。

（2）软件出现了产品说明书指明不会出现的错误。

（3）软件实现了产品说明书未提到的功能。

（4）软件未达到产品说明书虽未明确指出但应该实现的目标。

（5）软件难以理解、不易使用、运行缓慢或者终端用户认为不好。

为了更好地理解每一条规则，我们以计算器为例进行说明。

计算器的产品说明书声称它能够无误地进行加、减、乘、除运算。当你拿到计算器后按下“+”键，结果什么反应也没有，根据第（1）条规则，这是一个软件缺陷。

若产品说明书声称计算器永远不会崩溃、锁死或者停止反应，而当你任意按键时，计算器停止接收输入，根据第（2）条规则，这是一个软件缺陷。

若对计算器进行测试发现除了加、减、乘、除之外，它还可以求平方根，而说明书中没提到这一功能，根据第（3）条规则，这是一个软件缺陷。

若在测试计算器时发现电池没电会导致计算不准确，但产品说明书未提出这个问题，根据第（4）条规则，这是一个软件缺陷。

如果软件测试员发现“=”键的布置位置不易于使用，或者在明亮光下显示屏难以看清，则根据第（5）条规则，这是一个软件缺陷。

2．软件缺陷产生的原因

在软件生命周期的各个阶段都可能引入缺陷。软件缺陷的产生，首先是不可避免的。那么，造成软件缺陷的原因有哪些呢？从软件本身、团队工作、技术问题和项目管理方面对软件缺陷的原因归纳如下。

（1）软件本身问题。

- 软件需求等文档错误、内容不正确或拼写错误，导致设计目标偏离客户的需求，从而引起功能或产品特征上的缺陷。
- 系统结构复杂，使得层次或组件结构设计考虑不够周全，引起强度或负载问题，结果导致意想不到的问题或系统维护、扩充上的困难。
- 对程序逻辑路径或数据范围的边界考虑不够周全，漏掉某些边界条件，造成容量或边界错误。
- 对一些实时应用系统，由于没有进行精心设计和技术处理来保证精确的时间同步，容易引起时间上不协调、不一致的问题。
- 没有考虑系统崩溃后在系统安全性、可靠性方面的隐患。
- 系统运行环境复杂，容易引起一些特定用户环境下的问题。
- 由于通信端口多、存取和加密手段的矛盾性等，造成系统的安全性与适用性缺陷。
- 事先没有考虑到新技术的采用涉及的技术或系统兼容的问题。

（2）团队工作问题。

- 系统需求分析时，对客户的需求理解不清楚，或者和用户的沟通存在一些困难。
- 不同阶段的开发人员相互理解不一致，如软件设计人员对需求分析的理解有偏差，编程人员对系统设计规格说明书中某些内容重视不够或存在误解。
- 对于设计或编程上的一些假定或依赖性，没有得到充分的沟通。
- 项目组成员的技术水平参差不齐，新员工较多，或培训不够。

（3）技术问题。

- 算法错误：在给定条件下，没能给出正确或准确的结果。
- 语法错误：对于编译性语言程序，编译器可以发现这类问题，但对于解释性语言程序，只能在测试运行时发现。
- 计算和精度问题：计算的结果没有满足所需要的精度。
- 系统结构不合理，造成系统性能低下。
- 接口参数传递不匹配导致的问题。

（4）项目管理问题。

- 缺乏质量文化，不重视质量计划，对质量、资源、任务、成本等的平衡性把握不好，容易挤掉需求分析、评审、测试等时间，遗留较多的缺陷。
- 开发周期短，需求分析、设计、编程、测试等各项工作不能完全按照定义好的流程来进行，工作不够充分，导致错误较多，同时，周期短还给各级开发人员造成太大的压

力，引起一些人为的错误。

- 开发流程不够完善，存在太多的随机性和缺乏严谨的内审或评审机制，容易产生问题。
- 文档不完善，风险估计不足。

3．软件缺陷的组成

由于造成软件缺陷的原因较多，所以软件缺陷的组成结果较复杂。按需求分析结果来划分，软件缺陷包括规格说明书缺陷、系统设计结果缺陷、代码编写缺陷；按软件缺陷的级别来划分，可分为微小的、一般的、严重的、致使的四种软件缺陷。微小的软件缺陷对功能几乎没有影响，如错别字、排版不整齐等就属于此类缺陷；一般的软件缺陷是不太严重的错误，如次要功能缺失、提示信息不准确、用户交互友好性差等；严重的软件缺陷包括主要功能部分丧失、次要功能全部丧失或致命的错误声明；致命的软件缺陷是造成系统崩溃、死机或数据丢失的缺陷。

1.2 软件测试概述

随着软件应用领域越来越广泛，其质量的优劣也日益受到人们的重视。质量保证能力的强弱直接影响着软件业的生存与发展。软件测试是一个成熟软件企业的重要组成部分，它是软件生命周期中一项非常重要且复杂的工作，它贯穿整个软件开发生命周期，是对软件产品（包括阶段性产品）进行验证和确认的活动过程。通过尽快尽早地发现在软件产品中所存在的各种问题——与用户需求、预先定义的不一致性，检查软件产品的错误，写成测试报告，交给开发人员修改，对保证软件可靠性具有极其重要的意义。

除此之外，由于软件缺陷所带来的高额修复费用，使得人们不得不将项目开发的 30%～50%的精力用于测试。据不完全统计，一些涉及生命科学领域的大型软件在测试上所用的时间往往是其他软件工程活动时间的 3～5 倍。软件测试好比工厂流水线中的质量检验部门，对软件产品的阶段性和整体性的质量进行检测和缺陷排查，并修正缺陷，从而达到保证产品质量的目的。由此可见，软件测试在产品的生命周期中的作用举足轻重。

1.2.1 软件测试概念

为了更好地理解软件测试的概念，下面分别介绍软件测试的定义、目的、原则和质量保证。

1．软件测试的定义

对于软件测试的定义，目前混杂，没有统一的标准，人们在很长一段时间里有着不同的认识。

1979 年，G. J. Myers 在他的经典著作《软件测试之艺术》中给出了软件测试的定义：程序测试是为了发现错误而执行程序的过程。

1983 年，在 IEEE（国际电子电气工程师协会）提出的软件工程标准术语中给软件测

试下的定义是：使用人工或自动手段来运行或测定某个系统的过程，其目的在于检验它是否满足规定的需求或弄清楚预期结果与实际结果之间的差别。

1983 年，软件测试的先驱 Dr. Bill Hetzel 对软件测试做了如下定义：评价一个程序和系统的特性或能力，并确定它是否达到预期的结果。

综上所述可以看出，人们对于软件测试的理解是不断深入的，且是从不同的角度加以诠释的。但总的可以理解为：软件测试是在规定的条件下对程序进行操作，以发现错误，对软件质量进行评估的一个过程。

2．软件测试的目的

软件测试是程序的一种执行过程，目的是尽可能发现并改正被测试软件中的错误，提高软件的可靠性。在谈到软件测试的目的时，许多人都引用 G. J. Myers 在《软件测试之艺术》一书中的观点：

（1）软件测试是为了发现错误而执行程序的过程。

（2）测试是为了证明程序有错，而不是证明程序无错误。

（3）一个成功的测试是发现了至今未发现的错误的测试。

从上述观点可以看出，测试是以查找错误为中心的，而不是为了演示软件的正确功能。但是仅凭字面意思理解这一观点，可能会产生误导，认为发现错误是软件测试的唯一目的，查找不出错误的测试就是没有价值的，但事实并非如此。

首先，测试并不仅仅是为了找出错误。通过分析错误产生的原因和错误的分布特征，可以帮助项目管理者发现当前所采用的软件过程的缺陷，以便改进。同时，这种分析也能帮助我们设计出有针对性的检测方法，改善测试的有效性。

其次，没有发现错误的测试也是有价值的，完整的测试是评定测试质量的一种方法。详细而严谨的可靠性增长模型可以证明这一点。

总之，测试的目的是要证明程序中有错误存在，并且尽最大努力、尽早地找出最多的错误。测试不是为了显示程序是正确的，而是从软件含有缺陷和故障这个假定进行测试活动，并从中尽可能多地发现问题。当然，有些测试是为了给最终用户提供具有一定可信度的质量评价，此时的测试就应该直接针对在实际应用中经常用到的商业假设。

从用户的角度考虑，借助软件测试充分暴露软件中存在的缺陷，从而考虑是否接受该产品；从开发者的角度考虑，软件测试能表明软件已经正确地实现了用户的需求，达到软件正式发布的规格要求。

3．软件测试的原则

软件测试的对象不仅仅包括对源程序的测试，开发阶段的文档（如用户需求规格说明书、概要设计说明书、详细设计说明书等）都是软件测试的重要对象。在整个的软件测试过程中，应该努力遵循以下原则。

（1）尽可能早地开展预防性测试。测试工作进行得越早，就越有利于软件产品的质量提升和成本的降低。由于软件的复杂性和抽象性，在软件生命周期的各阶段都有可能产生错误，所以，软件测试不应是独立于开发阶段之外的，而应该是贯穿到软件开发的各个阶

段之中。确切地说，在需求分析和设计阶段中，就应该开始进行测试工作了。只有这样才能充分保证尽可能多且尽可能早地发现缺陷并及时修正，以避免缺陷或者错误遗留到下一个阶段，从而提高软件质量。

（2）可追溯性。所有的测试都应该追溯到用户需求。软件测试提示软件的缺陷，一旦修复这些缺陷就能更好地满足用户需求。如果软件实现的功能不是用户所期望的，将导致软件测试和软件开发做了无用功，而这种情况在具体的工程实现中确实时有发生。

（3）投入/产出原则。根据软件测试的经济成本观点，在有限的时间和资源下进行完全的测试即找出软件所有的错误和缺陷是不可能的，也是软件开发成本所不允许的，因此，软件测试不能无限制地进行下去，应适时终止，即不充分的测试是不负责任的。过分的测试却是一种资源的过度浪费，同样是一种不负责任的表现。所以，在满足软件的质量标准的同时，应确定质量的投入产出比是多少。

（4）80/20 原则。测试实践表明：系统中 80%左右的缺陷主要来自 20%左右的模块和子系统中，所以，应当用较多的时间和精力测试那些具有更多缺陷数目的程序模块和子系统。

（5）注重回归测试。由于修改了原来的缺陷，将可能导致更多的缺陷产生，因此，修改缺陷后，应集中对软件的可能受影响的模块和子系统进行回归测试，以确保修改缺陷后不引入新的软件缺陷。

（6）引入独立的软件测试机构或委托第三方测试。由于开发人员思维定式和心理因素等原因，开发工程师难以发现自己的错误，同时揭露自己程序中的错误也是件非常困难的事情。因此，软件测试除了需要软件开发工程师的积极参与外，还需要由独立的测试部门或第三方机构进行。

4．软件测试与质量保证

在讨论软件测试时，经常会提到质量保证。下面说明两者的关系。

质量保证，即软件质量保证（Software Quality Assurance，SQA），是建立一套有计划的系统方法，来向管理层保证拟定出的标准、步骤、实践和方法能够正确地被所有项目所采用。进一步地说，软件的质量保证活动是确保软件产品从诞生到消亡为止的所有阶段的质量的活动。

质量保证的主要工作范围如下。

（1）指导并监督项目按照过程实施。

（2）对项目进行度量、分析，增加项目的可视性。

（3）审核工作产品，评价工作产品和过程质量目标的符合度。

（4）进行缺陷分析、缺陷预防活动，发现过程的缺陷，提供决策参考，促进过程改进。

质量保证和软件测试都是贯穿于整个软件开发生命周期的。但是，SQA 侧重于对流程中各过程的管理与控制，是一项管理工作，侧重于流程和方法；而软件测试是对流程中各过程管理与控制策略的具体执行和实施，是一项技术性工作，其对象是软件产品（包括阶段性的产品），即 SQA 是从流程方面保证软件的质量，软件测试是从技术方面保证软件的质量。有了 SQA，软件测试工作就可以被客观地检查和评价，同时也可以协助测试流程的

改进。而软件测试为 SQA 提供了数据和依据，可以帮助 SQA 更好地了解质量计划的执行情况。软件测试，常常被认为是质量控制的最主要手段。

1.2.2　软件测试的重要性

软件测试是软件工程的重要部分，是保证软件质量的重要手段。在这一节中，通过几个成功的软件测试带来的好处和不完整的软件测试带来的教训，说明软件测试的重要性。

（1）IE 和 Netscape。

在 IE 4.0 的开发期间，微软为了打败 Netscape 而汇集了一流的开发人员和测试人员。测试人员搭建起测试环境，让 IE 在数台计算机上持续运行一个星期，而且要保障 IE 在几秒钟以内可以访问数千个网站，在无数次的试验以后，测试人员证明了 IE 在多次运行以后依然可以保障它的运行速度。而且，为了快速完成 IE 4.0 的开发，测试人员每天都要对新版本进行测试，不仅要发现问题，而且要找到问题是哪一行代码造成的，让开发人员专心于代码的编写和修改，最终 IE 取得了很大的成功。

（2）360 存在严重后果缺陷导致系统崩溃。

计算机中了木马，使用 360 安全卫士查出一个名为 Backdoor/Win32.Agent.cgg 的木马，文件位置为 C:\Windows\system32\shdocvw.dll。进行清理后看不到 Windows 任务栏和桌面图标，根本进不去桌面，手工运行 Explorer.exe 也是一闪就关，后来查明是由于 360 在处理此木马时存在严重缺陷。360 安全卫士只是简单地删除了木马文件，没有进行相关的善后处理工作，致使系统关键进程 Explorer.exe 无法加载。

（3）2009 年 2 月，Google 的 Gmail 故障。

2009 年 2 月，Google 的 Gmail 发生故障，Gmail 用户几小时不能访问邮箱。据 Google 后称，那次故障是因数据中心之间的负载均衡软件的 bug 引发的。

再看看下面的例子就会发现，360 和 Gmail 的问题是“小巫见大巫”了。

（4）2011 年温州“7·23”动车事故。

2011 年 7 月 23 日 20 时 30 分 05 秒，甬温线浙江省温州市境内，由北京南站开往福州站的 D301 次列车与杭州站开往福州南站的 D3115 次列车发生动车组列车追尾事故，造成 40 人死亡、172 人受伤，中断行车 32 小时 35 分，直接经济损失达 19371.65 万元。

事后，调查分析，“7·23”动车事故是由于温州南站信号设备在设计上存在严重缺陷，遭雷击发生故障后，导致本应显示为红灯的区间信号机错误显示为绿灯。

（5）火星登录事故。

由于两个测试小组单独进行测试，没有进行很好沟通，缺少一个集成测试的阶段，导致 1999 年美国宇航局的火星基地登录飞船在试图登录火星表面时突然坠毁失踪。质量管理小组观测到故障，并认定出错误动作的原因极可能是某一个数据位被意外更改，什么情况

下这个数据位被修改了？又为什么没有在内部测试时发现呢？

从理论上看，登录计划是这样的：在飞船降落到火星的过程中，降落伞将被打开，减缓飞船的下落速度。在降落伞打开后的几秒钟内，飞船的 3 条腿将迅速撑开，并在预定地点着陆。当飞船离地面 1800m 时，它将丢弃降落伞，点燃登录推进器，在余下的高度缓缓降落地面。

美国宇航局为了省钱，简化了确定何时关闭推进器的装置。为了替代其他太空船上使用的贵重雷达，在飞船的脚上装了一个廉价的触点开关，在计算机中设置一个数据位来关掉燃料。很简单，飞船的脚不“着地”，引擎就会点火。不幸的是，质量管理小组在事后的测试中发现，当飞船的脚迅速摆开准备着陆时，机械震动在大多数情况下也会触发着地开关，设置错误的数据位。设想飞船开始着陆时，计算机极有可能关闭推进器，而火星登录飞船下坠 1800m 之后冲向地面，必然会撞成碎片。

为什么会出现这样的结果？原因很简单，登录飞船经过了多个小组测试，其中一个小组测试飞船的脚落地过程，但从没有检查那个关键的数据位，因为那不是由这个小组负责的范围；另一个小组测试着陆过程的其他部分，但这个小组总是在开始测试之前重置计算机、清除数据位。双方本身的工作都没什么问题，就是没有合在一起测试，其接口没有被测，后一个小组没有注意到数据位已经被错误设定。

1.3 软件测试模型

软件测试模型是软件测试工作的框架，它描述了软件测试过程中所包含的主要活动以及这些活动之间的相互关系。通过测试模型，软件测试工程师及相关人员可以了解到测试何时开始、何时结束，测试过程中主要包含哪些活动以及需要哪些资源等。下面介绍常用的软件测试模型。

1.3.1 V 模型

在软件测试方面，V 模型是最广为人知的模型，它是软件开发中瀑布模型的变种。V 模型是在 RAD（Rap Application Development，快速应用开发）模型的基础上演变而来的，由于整个开发过程构成一个 V 字形而得名，是属于线性顺序一类的软件开发模型。

由于 RAD 通过使用基于构件的开发方法来缩短产品开发的周期，提高开发的速度，所以，软件测试 V 模型实现的前提是能做好需求分析，并且项目范围明确。RAD 开发模型包含如下几个阶段。

（1）业务建模。业务活动中的信息流被模型化，通过回答以下问题来实现：什么信息驱动业务流程？生成什么信息？谁生成该信息？该信息流往何处？谁处理它？

（2）数据建模。业务建模阶段定义的一部分信息流被细化，形成一系列支持该业务所需的数据对象，标识出每个对象的属性，并定义这些对象间的关系。

（3）处理建模。数据建模阶段定义的数据对象变换成要完成一个业务功能所需的信息

流，创建处理描述以便增加、修改、删除或获取某个数据对象。

（4）应用生成。RAD 过程不是采用传统的第三代程序设计语言来创建软件，而是使用 4GL 技术或软件自动化生成辅助工具，复用已有的程序构件（如果可能的话）或创建可复用的构件（如果需要的话）。

（5）测试及反复。因为 RAD 过程强调复用，许多程序构件已经是测试过的，这缩短了测试时间，但对新构件必须测试，也必须测到所有接口。

软件测试 V 模型如图 1.1 所示。V 模型的左边，从上到下描述了基本的开发过程，V 模型的右边，从下到上描述了基本的测试行为。

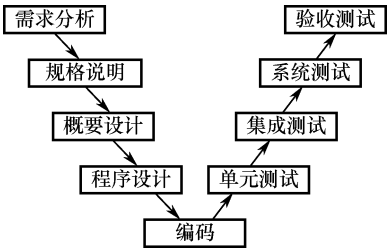

图 1.1　软件测试 V 模型示意图

除此以外，软件测试 V 模型还有一种改进型，将“编码”从 V 字形的顶点移动左侧，和单元测试对应，从而构成水平的对应关系，如图 1.2 所示。下面通过水平和垂直对应关系的比较，使用户能更清楚、全面地了解软件开发过程的特性。

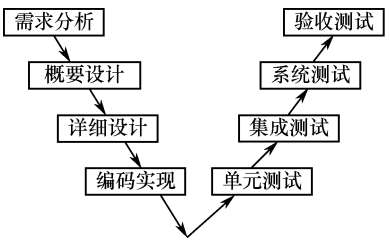

图 1.2　改进的软件测试 V 模型示意图

（1）从水平对应关系看，左边是设计和分析，右边是验证和测试。右边是左边结果的检验，即对设计和分析的结果进行测试，以确认是否满足用户的需求。具体来说：

- 需求分析对应验收测试，说明在做需求分析、产品功能设计的同时，测试人员就可以阅读、审查需求分析的结果，从而了解产品的设计特性、用户的真正需求；
- 当系统人员在做概要设计时，测试人员可以了解系统是如何实现的，基于什么样的平台，这样可以事先准备系统的测试环境，包括硬件和第三方软件的采购；
- 当程序设计师在做详细设计时，测试人就可以考虑准备测试用例；
- 程序员一面编程，一面进行单元测试，是一种很有效的办法，可以尽快地找出程序中的错误，充分的单元测试可以大幅度提高程序质量、减少成本。

（2）从垂直方向看，需求分析和验收测试是面向用户的，需要和用户进行充分的沟通和交流，或者和用户一起完成。其他工作都是技术工作，在开发组织内部进行，由工程师

完成。

在软件测试 V 模型中，项目启动，软件测试的工作也就启动了。它具有如下优缺点。

（1）优点：

①强调软件开发的协作和速度，反映测试活动和分析设计关系，将软件实现和验证有机结合起来。

②明确界定了测试过程存在不同的级别。

③明确了不同的测试阶段和研发过程中的各个阶段的对应关系。

（2）缺点：

①仅仅把测试过程作为在需求分析、系统设计及编码之后的一个阶段，忽视了测试对需求分析。

②系统设计的验证，一直到后期的验收测试才被发现。

③没有明确地说明早期的测试，不能体现“尽早地、不断地进行软件测试”的原则。

1.3.2 W 模型

针对 V 模型没有明确地说明早期的测试，无法体现“尽早地、不断地进行软件测试”的原则的局限性。在 V 模型中增加软件各开发阶段应同步进行的测试，演化为 W 模型，如图 1.3 所示。从图 1.3 可看出，开发是“V”，测试是与此并行的“V”。基于“尽早地、不断地进行软件测试”的原则，在软件的需求和设计阶段的测试活动应遵循 IEEE 1012—1998《软件验证与确认（V&V）》的原则。

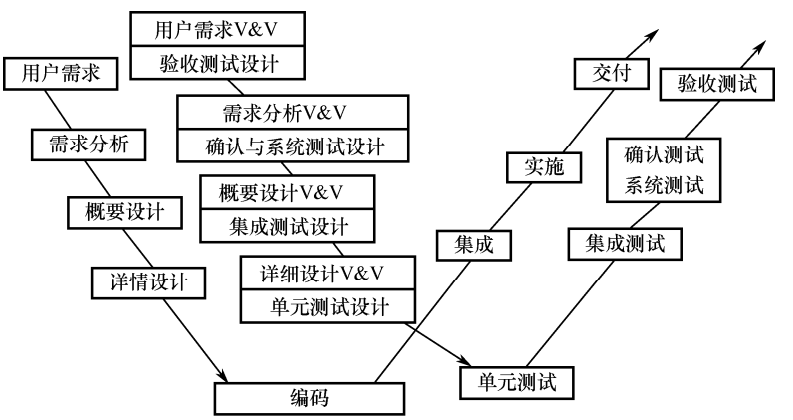

图 1.3　软件测试 W 模型示意图

W 模型由 Evolutif 公司提出，相对于 V 模型，W 模型更科学。W 模型是 V 模型的发展，强调的是测试伴随着整个软件开发周期，而且测试的对象不仅仅是程序，对需求、功能和设计也要测试。比如在进行需求分析、软件功能规格说明书评审、软件功能规格说明书基线化后，系统测试计划、方案、用例也设计完毕，接着是概要设计与集成测试设计，详细设计与单元测试设计，直到编码完成后，进行代码审查，继续执行单元测试、集成测试、系统测试。所以，W 模型，也就是双 V 模型，并不是在 V 模型上又搞出一个来，而是

开发阶段与测试设计阶段同步进行。

W 模型也有局限性。W 模型和 V 模型都把软件的开发视为需求、设计、编码等一系列串行的活动，无法支持迭代、自发性以及变更调整。所以，下面再简单介绍 X 模型和 H 模型。

X 模型也是对 V 模型的改进，X 模型提出针对单独的程序片段进行相互分离的编码和测试，此后通过频繁的交接，通过集成最终合成为可执行的程序，如图 1.4 所示。

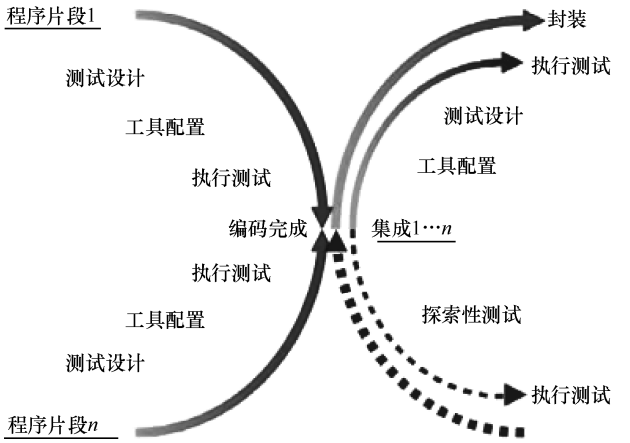

图 1.4　软件测试 X 模型示意图

图 1.4 左边描述的是针对单独程序片段所进行的相互分离的编码和测试，此后将进行频繁的交接，通过集成最终成为可执行的程序，然后再对这些可执行程序进行测试。对已通过集成测试的成品可以进行封装并提交给用户，也可以作为更大规模和范围内集成的一部分。多根并行的曲线表示变更可以在各个部分发生。由图中可见，X 模型还定位了探索性测试，这是不进行事先计划的特殊类型的测试，这一方式往往能帮助有经验的测试人员在测试计划之外发现更多的软件错误。但这样可能会对测试造成人力、物力和财力的浪费，对测试员的熟练程度要求比较高。

在 H 模型中，软件测试过程活动完全独立，贯穿于整个产品的周期，与其他流程并发地进行，某个测试点准备就绪时，就可以从测试准备阶段进行到测试执行阶段。软件测试可以尽早地进行，并且可以根据被测物的不同而分层次进行，如图 1.5 所示。

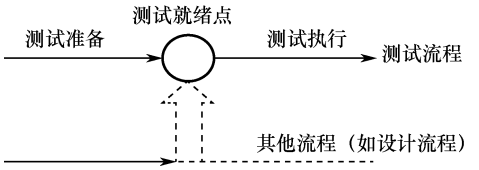

图 1.5　软件测试 H 模型示意图

图 1.5 演示了在整个生产周期中，某个层次上的一次测试“微循环”。图中标注的其他流程可以是任意的开发流程，例如设计流程或者编码流程。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了。

H 模型揭示了一个原理：软件测试是一个独立的流程，贯穿于产品整个生命周期，与其他流程并发地进行。H 模型指出软件测试要尽早准备，尽早执行。不同的测试活动可以是按照某个次序先后进行的，但也可能是反复的，只要某个测试达到准备就绪点，测试执行活动就可以开展。

1.4 软件测试用例

软件测试用例，简称测试用例，是为了实现测试有效性的一种常用工具，不进行良好的测试用例设计，测试过程将变得混乱而没有目标。一个成功的测试，将在很大程度上依赖于所设计和使用的测试用例。一个好的测试用例可以发现尽可能多的软件缺陷，而一个成功的测试用例会发现从未发现的软件缺陷。

在对不同的软件进行测试时，需要开发不同的测试用例。很显然，开发良好的测试用例，需要对被测试软件有充分的了解，同时也需要有丰富的测试知识。在为不同的软件开发测试用例时，需要掌握一些共性知识。下面介绍在开发测试用例时需要掌握的一些共性知识。

1.4.1 测试用例的基本概念

软件测试是一个分析或操作软件的过程，其设计目标是在一个软件系统中，通过一组操作，得到一个或多个预期的结果。如果得到了所有预期的结果，那么测试就通过了；如果实际结果与预期的不同，那么测试就失败了。软件测试由三个部分组成：一组操作、一组预期的结果和一组实际的结果。为了使测试有效，对操作和预期结果都需要进行清楚而无歧义的定义，即测试用例设计。

测试用例是为了某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个程序路径是否正确或核实某个功能是否满足特定需求。换句话说，一个测试用例就是一个文档，描述输入、动作或者时间和一个期望的结果，其目的是确定应用程序的某个特性是否正常工作。测试用例是测试方案、方法、技术和策略的综合体现，其内容包括测试目标、测试环境、输入数据、测试步骤、预期结果、测试脚本等。

在实际工程中，测试用例通常包括如下一些内容。

（1）用例 ID（test case ID）。

（2）用例名称（test case name）。

（3）测试目标（test target）。

（4）测试级别（test level（test phase，ST，SIT，UAT…））。

（5）测试对象（test object）。

（6）测试环境（test environment）。

（7）前提条件（prerequisites/dependencies/assumptions）。

（8）测试步骤（test steps/test script）。

（9）预期结果（expected result）。

（10）设计人员（designer）。

（11）执行人员（tester）。

（12）实际的结果/测试的结果（actual result/test result）。

（13）相关的需求和功能描述、需求描述（requirement description）。

（14）测试数据（test data）。

（15）测试结果的状态（test case status（passed，failed，hold，attention））。

其中，测试目标、测试对象、测试环境、前提条件、测试数据、测试步骤和预期结果是必需的。

不同类别的软件，其测试用例是不同的。测试用例是针对软件产品的功能、业务规则和业务处理所设计的测试方案。对软件的每个特定功能或运行操作路径的测试构成了一个测试用例。

1.4.2　测试用例的作用

测试用例是最小的测试单元，构成了设计和制定测试过程的基础。测试用例的作用如下。

（1）指导测试的实施。测试用例主要适用于集成测试、系统测试和回归测试。在实施测试时，测试用例作为测试的标准，测试人员一定要严格按照测试用例对项目实施测试，并将测试情况记录在测试用例管理软件中，以便自动生成测试结果文档。根据测试用例的测试等级，集成测试应测试哪些用例，系统测试和回归测试又该测试哪些用例，在设计测试用例时都已做明确规定，实施测试时测试人员不能随意做变动。

（2）规划测试数据的准备。在我们的实践中，测试数据与测试用例是分离的，按照测试用例，可以配套准备一组或若干组测试原始数据，以及标准测试结果。尤其是对于测试报表之类数据集的正确性，按照测试用例规划准备测试数据是十分必需的。除正常数据之外，还必须根据测试用例设计大量边缘数据和错误数据。

（3）编写测试脚本的“设计规格说明书”。为提高测试效率，软件测试已大力发展自动测试。自动测试的中心任务是编写测试脚本，如果说软件工程中软件编程必须有设计规格说明书，那么测试脚本的设计规格说明书就是测试用例。

（4）评估测试结果的度量基准。完成测试实施后，需要对测试结果进行评估，并且编制测试报告，判断软件测试是否完成和衡量测试质量需要一些量化的结果。例如：测试覆盖率是多少、测试合格率是多少、重要测试合格率是多少，等等。以前统计基准是软件模块或功能点，显得过于粗糙，采用测试用例做度量基准更加准确、有效。

（5）分析缺陷的标准。通过收集缺陷，对比测试用例和缺陷数据库，分析确定是漏测还是缺陷复现。漏测反映了测试用例的不完善，应立即补充相应测试用例，最终达到逐步完善软件质量。如果已有相应测试用例，则反映实施测试或变更处理存在问题。

测试工作量与测试用例的数据成比例。测试设计和开发的类型以及所需的资源主要受控于测试用例。测试用例通常根据它们所关联关系的测试类型或测试需求来分类，而且将随类型和需要进行相应的改变。最佳方案是为每个测试需求至少编制两个测试用例：一个

测试用例用于证明该需求已经满足，通常称为正面测试用例；另一个测试用例反映某个无法接受、反常或意外的条件或数据，用于评论只有在所需条件下才能够满足该需求，通常称为负面测试用例。

同时，测试的“深度”也与测试用例的数量成比例。由于每个测试用例反映不同的场景、条件或经由产品的事件流，因此随着测试用例数量的增加，对软件产品质量和测试流程也就越有信心。

1.4.3 测试用例的设计原则

设计测试用例的时候，需要有清晰的测试思路，对要测试什么，按照什么顺序测试，覆盖哪些需求做到心中有数。测试用例编写者不仅要掌握软件测试的技术和流程，而且要对被测试软件的设计、功能规格说明、用户试用场景以及程序/模块的结构都有比较透彻的理解。测试用例设计一般包括以下几个步骤。

（1）测试需求分析。从软件需求文档中，找出待测软件/模块的需求，通过自己的分析、理解，整理成为测试需求，清楚被测试对象具有哪些功能。测试需求的特点是：包含软件需求，具有可测试性。

测试需求应该在软件需求基础上进行归纳、分类或细分，方便测试用例设计。测试用例中的测试集与测试需求的关系是多对一的关系，即一个或多个测试用例集对应一个测试需求。

（2）业务流程分析。软件测试，不单纯是基于功能的黑盒测试，而且还需要对软件的内部处理逻辑进行测试。为了不遗漏测试点，需要清楚地了解软件产品的业务流程。建议在做复杂的测试用例设计前，先画出软件的业务流程。如果设计文档中已经有业务流程设计，可以从测试角度对现有流程进行补充。如果无法从设计中得到业务流程，测试工程师应通过阅读设计文档，与开发人员交流，最终画出业务流程图。业务流程图可以帮助理解软件的处理逻辑和数据流向，从而指导测试用例的设计。

（3）测试用例设计。完成了测试需求分析和软件流程分析后，开始着手设计测试用例。测试用例设计的类型包括功能测试、边界测试、异常测试、性能测试、压力测试等。在用例设计中，除了功能测试用例外，应尽量考虑边界、异常、性能的情况，以便发现更多的隐藏问题。

（4）测试用例评审。测试用例设计完成后，为了确认测试过程和方法是否正确，是否有遗漏的测试点，需要进行测试用例的评审。测试用例评审一般由测试经理安排，参加的人员包括测试用例设计者、测试经理、项目经理、开发工程师、其他相关开发测试工程师。测试用例评审完毕，测试工程师根据评审结果，对测试用例进行修改，并记录修改日志。

（5）测试用例更新完善。测试用例编写完成之后需要不断完善，软件产品新增功能或更新需求后，测试用例必须配套修改更新。在测试过程中发现测试用例设计考虑不周时，需要对测试用例进行修改完善；在软件交付使用后，客户反馈了软件缺陷，而缺陷又是因测试用例存在漏洞造成的，也需要对测试用例进行完善。一般小的修改完善可在原测试用例文档上修改，但文档要有更新记录。在软件的版本升级更新时，测试用例一般也应随之

编制升级更新版本。

一般来说，测试用例设计中应尽可能遵守下列原则。

（1）测试用例的代表性：能够代表并覆盖各种合理的和不合理的、合法的和非法的、边界的和越界的，以及极限的输入数据、操作和环境设置等。

（2）测试结果的可判定性：测试执行结果的正确性是可判定的，每一个测试用例都应有相应的期望结果。

（3）测试结果的可再现性：对同样的测试用例，系统的执行结果应当是相同的。

满足了上述原则设计出来的测试用例在理论上就是好的测试用例。但在实际工程中还远远不是，因为在理论上不需要考虑的东西，在实际工程中却是不得不考虑的——成本。由于成本因素的介入，我们在设计测试用例时，还需要考虑以下四条原则。

（1）单个用例覆盖最小化原则。

这条原则是所有这四条原则中的“老大”，也是在工程中最容易被忘记和忽略的，它或多或少地都影响到其他几条原则。下面举个例子，假如要测试一个功能 A，它有三个子功能点（A1、A2 和 A3），可以采用下面两种方法来设计测试用例。

方法 1：用一个测试用例覆盖三个子功能，即 Test_A1_A2_A3。

方法 2：用三个单独的用例分别来覆盖三个子功能采用，即 Test_A1，Test_A2，Test_A3。

方法 1 适用于规模较小的工程，但凡是稍微有点规模和质量要求的项目，方法 2 则是更好的选择，因为它具有如下优点：

- 测试用例的覆盖边界定义更清晰。
- 测试结果对产品问题的指向性更强。
- 测试用例间的耦合度最低，彼此之间的干扰也就越低。

上述这些优点所能带来直接好处是，测试用例的调试、分析和维护成本最低。每个测试用例应该尽可能简单，只验证你所要验证的内容。David Astels 在他的著作《Test Driven Development：A Practical Guide》曾这样描述：“最好一个测试用例只有一个 Assert 语句。”此外，覆盖功能点简单明确的测试用例，便于组合生成新的测试，很多测试工具都提供了类似组合已有测试用例的功能，例如 Visual Studio 中就引入了 Ordered Test 的概念。

（2）测试用例替代产品文档功能原则。

通常，我们会在开发的初期用文档记录产品的需求、功能描述，以及当前所能确定的任何细节等信息，勾勒将要实现功能的轮廓，便于团队进行交流和细化，并在团队内达成对将要实现的产品功能共识。假设我们在此时达成共识后，描述出来的功能为 A，随着产品开发深入，经过不断地迭代之后，团队会对产品的功能有更新的认识，产品功能也会被更具体细化，最终实现的功能很可能是 A+。如此往复，在不断倾听和吸收用户的反馈，修改产品功能，多个迭代过后，原本被描述为 A 的功能很可能最终变为了 Z。这时候再去看最初的文档，却仍然记录的是 A。之所以会这样，是因为很少有人会去（以及能够去）不断更新那些文档，以准确反映出产品功能当前的准确状态。不是不想去做，而是实在很难！

但是测试需要实时地反映产品的功能，否则的话，测试用例就会执行失败。因此，对测试用例的理解应该再上升到另一个高度，它应该是能够扮演产品描述文档的功能。这就要求我们编写的测试用例足够详细，测试用例的组织要有条理、分主次，单靠文档编辑工

具是远远无法完成的，需要更多专用的测试用例管理工具来辅助，例如 Visual Studio 2010 引入的 Microsoft Test Manager。

（3）单次投入成本和多次投入成本原则。

与其说这是一条评判测试用例的原则，不如说它是一个思考问题的角度。成本永远是任何项目进行决策时所要考虑的首要因素，软件项目中的开发需要考虑成本，测试工作同样如此。对成本的考虑应该客观和全面地体现在测试的设计、执行和维护的各个阶段。当你在测试中遇到一些左右为难的问题需要决策时，尝试着从成本角度去分析一下，也许会给你的决策带来一些新的启发和灵感。

测试的成本按其时间跨度和频率可以分为：单次投入成本和多次投入成本。例如：编写测试用例可以看成单次投入成本，因为编写测试用例一般是在测试的计划阶段进行的，虽然后期会有不断的调整，但绝大多数是在一开始的设计阶段就基本上成型了；自动化测试用例也是如此，它也属于是一次性投入；测试用例的执行则是多次投入成本，因为每出一个新版本时，都要执行所有的测试用例、分析测试结果、调试失败测试用例、确定测试用例的失败原因，以验证该版本整体质量是否达到了指定的标准。

当我们意识到了，测试用例的设计和自动化属于一次性投入，而测试用例的执行则是反复多次的投入时，就应该积极思考如何能够提高需要反复投入的测试执行的效率，在一次投入和需要多次活动需要平衡时，优先考虑多次投入活动的效率。例如：单个用例覆盖最小化原则中的例子，测试 A 功能的 3 个功能点（A1、A2 和 A3），从表面上看用 Test_A1_A2_A3 这一个用例在设计和自动化实现时最简单的，但它在反复执行阶段会带来很多的问题：首先，这样的用例的失败分析相对复杂，你需要确认到底是哪一个功能点造成了测试失败；其次，自动化用例的调试更为复杂，如果是 A3 功能点的问题，你仍需要不断地走过 A1 和 A2，然后才能到达 A3，这增加了调试时间和复杂度；再次，步骤多的手工测试用例增加了手工执行的不确定性，步骤多的自动化用例增加了其自动执行的失败可能性，特别是那些基于 UI 自动化技术的用例；最后，将不相关功能点耦合到一起，降低了尽早发现产品回归缺陷的可能性，这是测试工作的大忌。

（4）使测试结果分析和调试最简单化原则。

在编写自动化测试代码时，要重点考虑如何使得测试结果分析和测试调试更简单，包括用例日志、调试辅助信息输出等。往往在测试项目中，测试用例的编写人和最终的执行者是不同的团队的成员，甚至有个能测试的执行工作被采用外包的方式交给第三的团队去进行，这在当下也是非常流行的。因为测试用例的执行属于多次投入，测试人员要经常地去分析测试结果、调试测试用例，在这部分活动上的投入是相当可观的。

1.4.4 测试用例设计实例

登录功能是一个大家熟悉得不能再熟悉的功能了。但是，往往这类看似简单但却不简单的功能，在设计测试用例时却漏洞百出。下面通过 Google 邮箱的登录窗口实例进一步了解测试用例的设计。Google 邮箱登录界面截图如图 1.6 所示。