普通高等教育“十二五”规划教材

软件测试程序设计技术

孙　晶　杨　波　主编

赵会群　主审

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书从软件测试的基本理论出发，围绕 TTCN-3 核心语言国际测试标准，并结合大量的实际测试案例，对软件测试的相关方法与技术进行了详细的介绍，使读者能够更贴近实际地去了解软件测试。全书共 10 章，主要内容包括软件测试概述，软件测试基础，TTCN 树表描述语言程序设计，TTCN-3 基本语言元素，类型声明，语句、函数、可选步与通信，TTCN-3 核心语言程序设计，测试描述与测试控制，系统测试及测试工具，基于 TTCN-3 的软件测试案例。

本书内容全面、实例丰富、可操作性强，做到了理论与实践的有机结合。

本书适合作为计算机专业高年级本科生和研究生教材或教学参考书，也适合作为软件测试和软件开发相关人员的技术参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

软件测试程序设计技术 / 孙晶，杨波主编．— 北京：电子工业出版社，2015.10
（普通高等教育“十二五”规划教材）
ISBN 978-7-121-27337-7

I．①软… II．①孙… ②杨… III．①软件－测试 IV．①TP311.5

中国版本图书馆 CIP 数据核字（2015）第 234529 号

策划编辑：袁 玺
责任编辑：郝黎明　　特约编辑：张燕虹
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　印张：19　字数：536 千字
版　　次：2015 年 10 月第 1 版
印　　次：2015 年 10 月第 1 次印刷
定　　价：39.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010)88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010)88258888。

前　言

计算机技术已经越来越广泛地应用于国民经济和国防建设的各个部门，以不可阻挡之势渗透到人们工作和生活的各个领域，尤其在航天、航空、核能、通信、交通、金融等一些关键领域中，计算机的作用更加至关重要。同时，它们对计算机软件的可靠性和安全性也有严格的要求。近年来，由于软件错误而造成经济损失、导致严重后果的事件屡见不鲜，因此，如何保证软件产品的质量和可靠性就成为人们必须解决的一个重要问题，而软件测试便是保证软件质量的一个重要手段。据统计，国外在软件开发中，开发费用的近一半甚至更多要用于软件测试，由此也可以看出软件测试在软件开发中的重要地位。

本书是为希望了解软件测试的相关技术，尤其是希望了解 TTCN-3 的读者而编写的。依托 TTCN-3，本书给出了丰富且易于实践的案例，让读者能够做到理论与实践相结合，从而能够更加充分地理解和掌握软件测试的相关方法和技术。

本书共 10 章，第 1 章是软件测试概述，第 2 章是软件测试基础，第 3 章是 TTCN 树表描述语言程序设计，第 4 章是 TTCN-3 基本语言元素，第 5 章是类型声明，第 6 章是语句、函数、可选步与通信，第 7 章是 TTCN-3 核心语言程序设计，第 8 章是测试描述与测试控制，第 9 章是系统测试及测试工具，第 10 章是基于 TTCN-3 的软件测试案例。

本书主要介绍如下内容。

(1)介绍软件测试的基本方法，重点讨论软件测试黑盒法和白盒法。对软件测试的过程进行讨论，重点讨论软件测试的设计和测试文档的使用。

(2)介绍 TTCN-3 核心语言的基本概念、语法结构和测试系统结构。通过学习可以掌握 TTCN-3 核心语言测试系统的设计与实现方法，能够设计小型协议软件、应用软件、嵌入式软件的测试系统，为实际从事测试工作奠定理论基础。

(3)介绍如何利用 TTCN-3 进行实际的软件测试，结合多个测试案例进行详细的讲解。

本书详细地讲述了软件测试的有关概念、方法、过程等方面的基础知识，也用大量篇幅讲解了 TTCN-3 的语法知识，并给出了丰富的案例，目的是使读者对软件测试有一个比较全面的了解，并为进一步研究软件测试技术奠定基础。

本书中有大量的算法语句、程序语句及计算公式等，对于其中的变量，为了方便读者阅读，避免歧义，不再区分正、斜体，而是统一采用正体，特此说明。

本书由孙晶、杨波主编，赵会群主审。

由于编者水平有限，书中难免有疏漏和不当之处，敬请广大读者不吝指正。

作　者

目　录

第 1 章　软件测试概述

本章概述软件测试的有关概念、方法和过程等方面的基础知识。使读者对软件测试有一个比较全面的了解，并为进一步讨论软件测试技术奠定基础。

1.1　软件故障与软件测试

在计算机故障中，有相当一部分是软件故障。下面让我们看两个例子。

例 1：英特尔奔腾浮点除法软件故障

在计算机的“计算器”程序中输入以下算式：

(4195835/3145727) *3145727-4195835

如果答案是 0，则说明计算机没有问题；如果得出的结果不是 0，则说明计算机的工作不正常。看起来这不应该是个问题，可实际上就出现了问题。

1994 年 12 月 30 日，美国 Lynchburg 大学的 Thomas R.Nicely 博士在一台奔腾 PC 上做除法运算时发现，上面的算式不等于 0。后来，他把这一个惊人的发现在 Internet 上发布出去，引起了一场风暴，成千上万的人都发现了同样的问题。那么是什么原因造成这样的算式计算错误呢？这由固化在奔腾 CPU 上的运算器芯片中的软件故障所致。

例 2：千年虫(Y2K)问题

首先介绍一个传说：20 世纪 70 年代一个名叫 Dave 的程序员，负责其公司的工资系统。他使用的计算机存储空间很小，迫使其尽量节省每一个字节。Dave 自豪地将自己的程序压缩得比其他人的更小。他使用的其中一个方法是把 4 位数日期缩减为 2 位，例如将 1973 年缩为 73。因为工资系统极度依赖数据处理，Dave 节省了可观的存储空间。Dave 并没有想到这是个很大的问题，他认为只有在 2000 年时程序计算 00 或 01 这样的年份时才会出现错误。他知道那时会出问题，但是在 25 年之内程序肯定会更改或升级，而且眼前的任务比未来更加重要。这一天毕竟是要来到的。1995 年，Dave 的程序仍然在使用，而 Dave 退休了，谁也不会想到进入程序检查 2000 年的兼容性问题，更不用说去修改了。

关于 Y2K 问题的说法不一，但根本的问题是用 2 位表示年份的问题。这是一个十分典型的软件设计缺陷。Y2K 问题涉及 4 个方面：硬件、操作系统、应用软件及数据。

有关千年的例子很多，给计算机产业带来一次震惊和恐慌。许多国家和大型计算机公司都动用了大量的人力和物力，解决千年虫问题，尤其是解决关系到国家安全、国家支柱产业正常运转和与百姓生活息息相关领域的计算机系统的千年虫问题。

微软作为全球最大的软件供应商，其产品涵盖了操作系统、应用软件及数据等领域，而在 PC 平台上形成最为广泛的应用。关于这一问题，微软对其产品进行了全面的兼容性测试。

微软自 1996 年起开始涉及有关 Y2K 问题的研究，对此，微软采取的是完全对外公开的策略，其中包括产品及其他任何有关 Y2K 的信息。作为一家既面向企业用户也针对广大个人用户的软件产品供应商，微软认为解决 Y2K 的首要问题是对产品进行全面深入的 2000 年兼容性测试。

首先，需要找到一种统一的方法来对不同的产品进行 2000 年兼容性测试；同时，由于微软的产品在全球得到了极为广泛的应用，因此要对产品进行不同语言的测试。至 1999 年年底，微软共

测试了 4052 种产品，这可谓迄今为止历史上最大的软件测试工程之一；其中，97%达到 2000 年兼容，3%不兼容。而不兼容的产品基本都是老产品，如 DOS 版本的 Word 5.0。同时，在整个测试过程中，并未做任何推断性测试，即不能在未对某种语言进行实际性测试的前提下，而从其他语言的同种产品的测试结果进行推断(如假设简体中文的测试结果没问题,则简单推断韩文也不存在任何问题)。英文及其他欧洲语言中只有 5%，而 20%以上的双字节语言(如大多数亚洲国家的语言)采用 UNICODE；同时，世界各地不同的民族采用不同的历法，这些都是测试需要考虑的问题。测试时尽力确保最新产品和最大量使用的产品以及采用关键技术，如 ODBC 的产品的 2000 年兼容测试，然后再来测试客户有特别需求的、采用某项专有技术的产品。

从上面的两个例子中，我们可以看出软件缺陷是造成软件故障的主要问题，也是软件测试的主要对象。那么什么是软件缺陷故障？什么是软件故障？软件测试定义是什么？它们之间的关系是什么？下面先给出一组有关软件测试的相关术语，明确它们之间的关系，进而给出软件测试的概念。

缺陷(bug)	偏差(variance)
缺点(defect)	失败(failure)
问题(problem)	矛盾(inconsistency)
错误(error)	事故(incident)
异常(anomaly)	谬误(fault)

在上述名词中，有一些名词的含义相近，属于一个范畴。第一类是缺陷、缺点和偏差，它们是一类含义相近的概念，我们不妨把它们统称为缺陷。它们都是软件开发过程潜在的缺陷，这些缺陷可能在软件投入运行后出现，使得软件的性能和可靠性等方面与系统的设计需求不符。有时，这些问题可能不出现，软件的性能和可靠性并不会因为它们的存在而受到影响。第二类是错误、谬误、问题、异常和矛盾等，我们把这一类问题统称为错误。这类错误与软件运行状态有关，它们是在软件运行过程中可观测到的软件错误。这些问题出现的原因是软件缺陷所致。第三类是失败、事故或灾难等，我们把这类统称为失败。这是软件运行给用户造成的损失的一类软件故障，它强调软件失败的结果。失败的直接原因是软件系统存在软件错误。并不是所有的软件错误都会导致软件失败，如果对软件错误加以适当的控制，软件错误可以导致安全。

综上所述，软件故障大体上可分为三种类型，每一类对应软件生命周期的不同阶段，贯穿整个软件开发和使用的全部过程。其中，第一类缺陷是软件故障的根源，后两类故障是软件缺陷的直接后果。所以，在软件开发过程中，发现和排除软件故障是一项长期艰苦的工作。而这一项工作的基础是加强软件设计时设计缺陷的检测。

那么什么是软件测试呢？所谓软件测试是为了评价一个软件系统的质量和发现错误而从事的一种工作过程。从软件测试作为软件的执行过程来看，可分为局部软件的局部运行和全部运行；从运行的环境来看，可有仿真运行和实际运行。这就存在一个软件测试中的方式和方法的问题。而方法又与采用的技术相关，技术不同，方法也不同。所以，软件测试技术是测试的关键。

从软件故障的分类中，可以看到软件的故障分布在软件开发的全过程，所以软件测试也就伴随软件开发和使用的整个过程中。在下一节中，我们将分析软件测试与软件生命周期的关系。把握软件测试与开发过程的阶段关系，为有针对性地开展软件测试奠定基础。

1.2 软件测试与软件开发过程

软件开发过程中的各种活动构成软件开发的生命周期,随着这些活动的组织方式和方法不同，就构成不同的软件开发生命周期模型。然而，无论是什么样的生命周期模型，软件开发无一例外

地要经历从软件需求分析到软件测试这样一个过程。也就是说，虽然软件开发的生命周期模型有所不同，但软件开发的阶段性始点和终点是相同的，而且软件测试是不可缺少的一项工作。

软件开发的生命周期并不是独立存在的，它是包含该软件的产品生命周期的一部分。在产品的生命周期内，软件被维护和纠错。当产品就是软件本身时，软件的维护和测试也是相当复杂的一项工作。不同的软件模块被组合成一个大的软件系统，给软件测试工作带来一定的难度。

有许多不同的软件生命周期模型，它们都需要进行测试。本节讨论各种软件开发生命周期模型与软件测试的关系，从而进一步明确软件测试在软件开发中的重要作用；为在采用不同软件开发方法的情况下，灵活运用软件测试的方法和技术奠定基础。

1.2.1　顺序生命周期模型(Sequential Lifecycle Models)

所谓的顺序生命周期模型(简称顺序模型)，是指把软件开发的整个过程定义成有序的开发活动序列，随着开发工作的进展，软件生命周期的状态也在迁移。如图 1-1、图 1-2 所示，顺序模型也称 V 模型或瀑布模型。

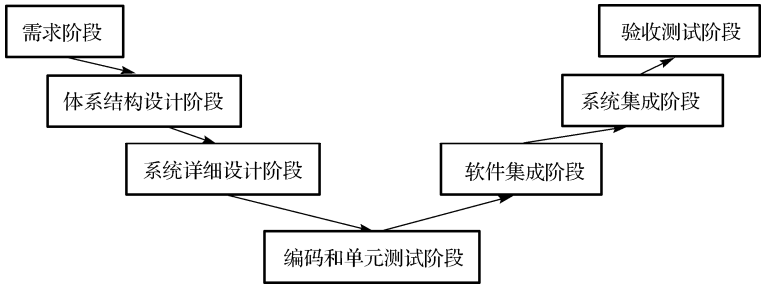


图 1-1　顺序生命周期模型

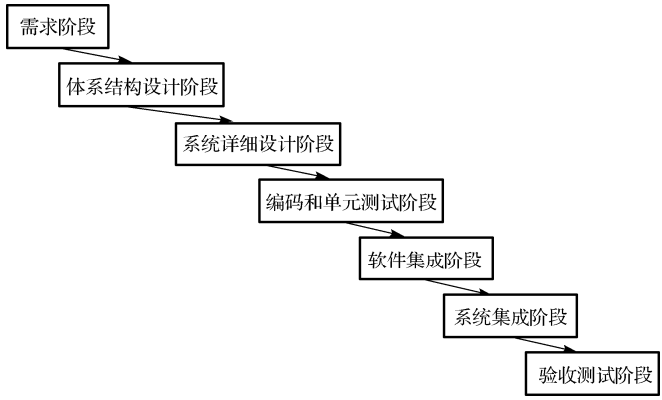


图 1-2　瀑布生命周期模型

瀑布模型也有许多变形，这些模型可能随着软件开发需求的不同，增加新的状态，这些状态有着不同的边界，下面给出一组典型的开发状态描述。

- **需求阶段(Requirements Phase)**：在需求阶段需要对用户需求进行分析，对要开发的软件进行严格的定义，这种定义应该是非二义性的。
- **体系结构设计阶段(Architectural Design Phase)**：在该阶段对软件系统的体系结构进行分析、设计和定义，进而说明体系结构中组件和组件之间的联系。

- **系统详细设计阶段(Detailed Design Phase)**：在该阶段对构成系统的各组件给出详细的设计和说明。
- **编码和单元测试阶段(Code and Unit Test Phase)**：该阶段对设计的组件进行编码，并验证组件代码与详细设计阶段定义的组件细节的正确性。
- **软件集成阶段(Software Integration Phase)**：在该阶段把已经测试过的组件组装到一起，并对集成系统进行测试，直到由组件所构成的软件满足设计要求。
- **系统集成阶段(System Integration Phase)**：在该阶段对软件与其他系统部件进行集成，并进行系统测试，直到系统正常工作为止。
- **验收测试阶段(Acceptance Test Phase)**：在该阶段将按照系统分析、设计定义的各个方法对系统进行测试，从而检验系统开发的正确性。

在上述系统生命周期中，前三个阶段是系统的定义阶段，后面四个阶段都需要对阶段成果进行测试。测试工作同样需要设计和说明，在生命周期的各个层面中，均需要定义该层面的测试点。

1.2.2 渐进(Progressive Development)生命周期模型

顺序生命周期模型是一个理性化的软件开发生命周期模型。在实际开发过程中，可能情况要复杂得多。这样根据特殊情况设计具体的软件开发生命周期模型是常见的。比如软件需求往往随着开发工作的进行而不断地扩充，或者为了避免开发周期过长，而先开发一个中间系统投入运行等，这些特殊情况改变了原有生命周期的各个阶段的分配。

下面介绍另一个软件开发生命周期模型，这个模型称为渐进生命周期模型(如图 1-3 所示)或阶段生命周期模型。

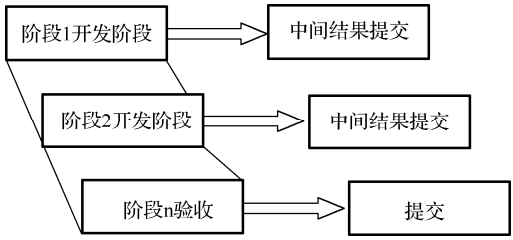


图 1-3 渐进生命周期模型

软件开发过程中的一个矛盾是，开发时间过长，而工期又往往要求很短。解决此问题的一个方法是在两者之间折中处理，首先用较短的时间开发一个中间系统，而功能相对比设计要求少一些。这个中间系统还需要进一步扩展，直到满足所有的功能需求。中间软件的开发可以有效地减少软件开发风险。我们把这种开发方法所建立的软件开发生命周期称为渐进生命周期模型(简称渐进模型)或阶段实现(Phased Implementation)生命周期模型。

在渐进模型中，每一个开发阶段仍然服从顺序模型中的规范，而整个生命周期的阶段可能有所增减，它的实际阶段取决于实际开发过程。

每一个提交的软件都要经过测试，以检验软件是否满足设计要求。测试工作分布在软件开发的各个阶段，不但要对各个阶段的成果进行测试，而且还要对各个阶段的集成成果进行检验。如果这种测试工作能够在控制范围内完成，则该工作将不会影响整个生命周期。然而，测试中发现的问题是严重的，有时需要重新开发软件。这是最坏的情况，但在这种情况下，整个软件的生命周期都将被打乱。无论是哪种情况，测试都需要花费很多时间和精力。所以我们应该尽可能及早地发现问题，减少不必要的开发代价。其中一个解决办法就是利用原型法构造系统的原型。系统

的原型是提供快速实现系统设计功能的方法，开发人员和用户可以在系统原型上测试所开发的系统是否满足设计要求。一个原型系统最终要进一步扩展成一个实际的系统，但在从原型系统到实际系统的过渡过程中，仍然需要测试。

1.2.3　迭代生命周期模型(Iterative Lifecycle Model)

在迭代生命周期模型中，开发工作的最初并不要求对软件需求进行详细的说明，而是随着软件开发工作的进行，逐渐辨别所开发软件的需求，并加以说明。上述过程可能要重复多次，产生新的软件版本。迭代模型就是这样一种开发模型，其生命周期表现为四个不断迭代的阶段，如图 1-4 所示。

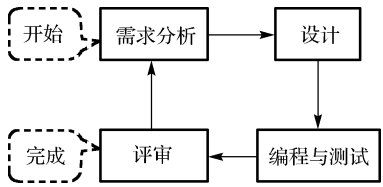


图 1-4　迭代生命周期模型

- **需求分析阶段：**该阶段对软件的需求进行收集并分析。在此基础上，不断的迭代将会得到最终的需求定义。
- **设计阶段：**该阶段根据需求进行设计，设计可能是新的设计，也可能是对早期设计的扩展。
- **编程与测试阶段：**该阶段的中心任务是对软件进行编码、集成和测试。
- **评审阶段：**在该阶段要对软件进行评估，回顾软件的需求，改变或增加新的需求。

对迭代中的每一个周期循环，必须决定软件中哪些部分或全部组件需要在下一次周期循环中被保留，或是被摒弃掉。最终将达到一个目标就是软件需求被满足，这时软件被提交。或者该软件不可能达到设计要求，而不得不从头开始。

迭代模型可以形象地比喻为通过连续的逼近方法来开发软件。这有一些像数学中的逼近法，它通过不断的逼近最终使问题求解。然而，在数学中逼近有时没有解，每一次迭代都在可行解的左右摆动，甚至是发散的。迭代的次数可能会很大，甚至是不可能的。那么在软件开发中，是不是这样呢？情况十分相似。软件中的错误会使得软件生命周期没有尽头，这也是一种发散。

在迭代模型中，成功的关键是在周期内的每一个阶段和循环中都要对结果进行严格的测试。模型的前三个阶段是顺序模型的浓缩，每一个循环中都要对软件进行单元测试，随着生命周期的延续，测试形影不离。

1.3　软件测试方法与测试内容

软件测试的种类很多，大体上可以从以下几个方面来进行划分。

- 从是否需要执行被测软件的角度，可分为静态测试和动态测试。
- 从测试是否针对系统的内部结构和具体实现算法的角度，可分为白盒测试和黑盒测试。
- 从测试范围角度，可分为单元测试、系统测试、集成测试等。
- 从测试目标角度，可分为性能测试、功能测试、可靠性测试等。
- 从测试采用的工具角度，可分自动测试和手工测试等。

软件测试的种类多种多样，测试所采用的测试的方法和技术也是多种多样的。有些测试方法是针对测试对象提出的，有些测试方法是根据测试软件的组织结构而提出的，而有些是从测试工具角度提出的。但无论是哪种方法，其核心都是比较设计需求与软件的实际执行结果的一致性、兼容性。

下面逐一介绍目前我们所听到的各种软件测试方法，通过这些方法的介绍了解软件测试的测试内容。

1.3.1 黑盒测试

黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能的情况下，通过测试来检测每个功能是否都能正常使用。在测试时，把程序看成一个不能打开的黑盆子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确地输出信息，并且保持外部信息(如数据库或文件)的完整性。黑盒测试方法主要有等价类划分、边值分析、因—果图、错误推测等，主要用于软件确认测试。“黑盒”法着眼于程序外部结构、不考虑内部逻辑结构、针对软件界面和软件功能进行测试。“黑盒”法是穷举输入测试，只有把所有可能的输入都作为测试情况使用，才能以这种方法查出程序中所有的错误。实际上测试情况有无穷多个，人们不仅要测试所有合法的输入，而且还要对那些不合法但是可能的输入进行测试。

1.3.2 白盒测试

白盒测试也称结构测试或逻辑驱动测试，它是在知道它产品内部工作过程的前提下，可通过测试来检测产品内部动作是否按照规格说明书的规定正常进行。按照程序内部的结构测试程序，检验程序中的每条通路是否都有能按预定要求正确工作，而不顾它的功能，白盒测试的主要方法有逻辑驱动、基路测试等，主要用于软件验证。

“白盒”法要求全面了解程序内部逻辑结构、对所有逻辑路径进行测试。“白盒”法是穷举路径测试。在使用这一方案时，测试者必须从检查程序的逻辑着手，检查程序的内部结构，得出测试数据。而贯穿程序的独立路径数是天文数字。但由于下面的原因即使每条路径都测试过，仍然可能存在错误。第一，穷举路径测试决不能查出程序违反了设计规范，即程序本身是个错误的程序。第二，穷举路径测试不可能查出程序中因遗漏路径而出现错误。第三，穷举路径测试可能发现不了一些与数据相关的错误。

1.3.3 ALAC(Act-like-a-customer)测试

ALAC 测试是一种基于客户使用产品的知识开发出来的测试方法。ALAC 测试是基于复杂的软件产品有许多错误的原则。最大的受益者是用户，缺陷查找和改正将针对哪些客户最容易遇到的错误。

1.3.4 单元测试

单元测试的对象是软件设计的最小单位——模块。单元测试的依据是详细设计描述，单元测试应对模块内所有重要的控制路径设计测试用例，以便发现模块内部的错误。单元测试多采用白盒测试技术，系统内多个模块可以并行地进行测试。

1.3.5 综合测试

时常有这样的情况发生：每个模块都能单独工作，但这些模块集成在一起之后却不能正常工作。发生这种情况的主要原因是，模块相互调用时接口会引入许多新问题。例如，数据经过接口可能丢失、一个模块对另一模块可能造成不应有的影响、几个子功能组合起来不能实现主功能、误差不断积累达到不可接受的程度以及全局数据结构出现错误，等等。综合测试是组装软件的系统测试技术，按设计要求把通过单元测试的各个模块组装在一起之后，进行综合测试以便发现与接口有关的各种错误。

1.3.6　确认测试

通过综合测试之后，软件已完全组装起来，接口方面的错误也已排除，软件测试的最后一步——确认测试即可开始。确认测试应检查软件能否按合同要求进行工作，即是否满足软件需求说明书中的确认标准。

1．确认测试标准

实现软件确认要通过一系列黑盒测试。确认测试同样需要制订测试计划和过程，测试计划应规定测试的种类和测试进度，测试过程则定义一些特殊的测试用例，旨在说明软件与需求是否一致。无论是计划还是过程，都应该着重考虑软件是否满足合同规定的所有功能和性能，文档资料是否完整、准确人机界面和其他方面(例如可移植性、兼容性、错误恢复能力和可维护性等)是否令用户满意。

确认测试的结果有两种可能：一种是功能和性能指标满足软件需求说明的要求，用户可以接受；另一种是软件不满足软件需求说明的要求，用户无法接受。项目进行到这个阶段才发现严重错误和偏差一般很难在预定的工期内改正，因此必须与用户协商，寻求一个妥善解决问题的方法。

2．配置复审

确认测试的另一个重要环节是配置复审。复审的目的在于保证软件配置齐全、分类有序，并且包括软件维护所必需的细节。

1.3.7　α、β 测试

事实上，软件开发人员不可能完全预见用户实际使用程序的情况。例如，用户可能错误地理解命令，或提供一些奇怪的数据组合，也可能对设计者自认明了的输出信息迷惑不解，等等。因此，软件是否真正满足最终用户的要求，应由用户进行一系列验收测试。验收测试既可以是非正式的测试，也可以是有计划、系统的测试。有时，验收测试长达数周甚至数月，不断暴露错误，导致开发延期。一个软件产品，可能拥有众多用户，不可能由每个用户验收，此时多采用称为 α、β 测试的过程，以期发现那些似乎只有最终用户才能发现的问题。

α 测试是指软件开发公司组织内部人员模拟各类用户对即将面市软件产品(称为 α 版本)进行测试，试图发现错误并修正。α 测试的关键在于尽可能逼真地模拟实际运行环境和用户对软件产品的操作并尽最大努力涵盖所有可能的用户操作方式。经过 α 测试调整的软件产品称为 β 版本。紧随其后的 β 测试是指软件开发公司组织各方面的典型用户在日常工作中实际使用 β 版本，并要求用户报告异常情况、提出批评意见。然后，软件开发公司再对 β 版本进行改错和完善。

1.3.8　系统测试

计算机软件是基于计算机系统的一个重要组成部分，软件开发完毕后会与系统中其他成分集成在一起，此时需要进行一系列系统集成和确认测试。对这些测试的详细讨论已超出软件测试的范围，这些测试也不可能仅由软件开发人员完成。在系统测试之前，软件工程师应完成下列工作。

(1) 为测试软件系统的输入信息设计出错处理。

(2) 设计测试用例，模拟错误数据和软件界面可能发生的错误，记录测试结果，为系统测试提供经验和帮助。

(3) 参与系统测试的规划和设计，保证软件测试的合理性。

系统测试应该由若干个不同测试组成，目的是充分运行系统，验证系统各部件是否都能正常工作并完成所赋予的任务。系统测试中又包括恢复测试、安全测试、强度测试、性能测试、可用性测试、可靠性测试。

(1)恢复测试：恢复测试主要检查系统的容错能力，即当系统出错时，能否在指定时间间隔内修正错误并重新启动系统。恢复测试首先要采用各种办法强迫系统失败，然后验证系统是否能尽快恢复。对于自动恢复需验证重新初始化(Reinitialization)、检查点(Checkpointing Mechanisms)、数据恢复(Data Recovery)和重新启动等机制的正确性；对于人工干预的恢复系统，还需估测平均修复时间，确定其是否在可接受的范围内。

(2)安全测试：安全测试检查系统对非法侵入的防范能力。安全测试期间，测试人员假扮非法入侵者，采用各种办法试图突破防线。例如，①想方设法截取或破译口令；②专门定做软件破坏系统的保护机制；③故意导致系统失败，企图趁恢复之机非法进入；④试图通过浏览非保密数据，推导所需信息，等等。理论上讲，只要有足够的时间和资源，没有不可进入的系统。因此系统安全设计的准则是，使非法侵入的代价超过被保护信息的价值。此时，非法侵入者已无利可图。

(3)强度测试：强度测试检查程序对异常情况的抵抗能力。强度测试总是迫使系统在异常的资源配置下运行。例如，①当中断的正常频率为 1～2h/s，运行每秒产生 10 个中断的测试用例；②定量地增加数据输入率，检查输入子功能的反应能力；③运行需要最大存储空间(或其他资源)的测试用例；④运行可能导致虚存操作系统崩溃或磁盘数据剧烈抖动的测试用例，等等。

(4)性能测试：对于那些实时和嵌入式系统，软件部分即使满足功能要求，也未必能够满足性能要求，虽然从单元测试起，每一测试步骤都包含性能测试，但只有当系统真正集成之后，在真实环境中才能全面、可靠地测试运行性能，系统性能测试是为了完成这一测试任务的。性能测试有时与强度测试相结合，经常需要其他软硬件的配套支持。

(5)可用性测试：对“用户友好性”的测试。显然这是主观的，且将取决于目标最终用户或客户。用户面谈、调查、用户对话的录像和其他一些技术都可使用。程序员和测试员通常都不宜作为可用性测试员。

(6)可靠性测试：可靠性测试是为了检验系统软件系统运行是否可靠而进行的一种测试。这类软件系统的失败往往导致不可预料的结果，如航空、航天领域中运行的软件，铁路系统中运行的软件等。可靠性测试的方法关注的是，一旦软件系统出现故障，其系统是否导向安全，所以可靠性测试与安全测试紧密相关。可靠性测试通常采用黑盒测试法。

1.3.9 面向对象的软件测试

面向对象的软件测试(OO Test)是根据面向对象的软件开发方法所设计的软件系统所提出的软件测试方法。OO Test 又分为面向对象分析的测试(OOA Test)、面向对象设计的测试(OOD Test)和面向对象的程序测试(OOP Test)。它们是对分析结果和设计结果的测试，主要是对分析设计产生的文本进行，是软件开发前期的关键性测试。OOP Test 主要针对编程风格和程序代码实现进行测试，其主要的测试内容在面向对象单元测试和面向对象集成测试中体现。面向对象单元测试是对程序内部具体单一的功能模块的测试，如果程序是用 C++语言实现，主要就是对类成员函数的测试。面向对象单元测试是进行面向对象集成测试的基础。面向对象集成测试主要对系统内部的相互服务进行测试，如成员函数间的相互作用、类间的消息传递等。面向对象集成测试不但要基于面向对象单元测试，更要参见 OOD 或 OOD Test 结果。面向对象系统测试是基于面向对象集成测试的最后阶段的测试，主要以用户需求为测试标准，需要借鉴 OOA 或 OOA Test 结果。

1.3.10　协议软件测试

在计算机网络的发展历程中，协议一直处于核心地位，它是计算机网络和分布式系统中各种通信实体之间相互交换信息所必须遵守的一组规则。1984 年，国际标准化组织 ISO 提出了开放式系统互连 ISO/OSI 参考模型。1993 年 1 月 1 日，TCP/IP 被宣布为 Internet 上唯一正式的协议，为 Internet 的发展铺平了道路。

协议软件作为软件的一种特殊形式，自 20 世纪 80 年代以来，其开发和检测方法已经得到快速的发展，已经形成了一个崭新的学科——协议工程学。它的研究范围包括协议说明(Protocol Specification)、协议证实(Protocol Validation)、协议验证(Protocol Verification)、协议综合(Protocol Synthesis)、协议转换(Protocol Conversion)、协议性能分析(Protocol Performance Analysis)、协议自动实现(Protocol Automatic Implementation)和协议测试(Protocol Testing)。

目前的网络协议多是以自然语言描述的文本，实现者对于协议文本的不同理解以及实现过程中的非形式化因素都会导致不同的协议实现，有时甚至是错误的协议实现。即便协议实现正确，也不能保证不同的实现彼此之间能够准确无误地通信，而且同一协议的不同实现其性能也有差别。在这种情况下，需要一种有效的方法对协议实现进行评价，这就是“协议测试”。

伴随着计算机网络的普及和网络需求的增多，计算机网络协议越来越复杂庞大，协议实现不仅仅要求功能正确完善、能够互通，而且要求具有良好的性能，因此协议的实现和开发越来越复杂。为了保证质量，协议测试是一个必需且十分重要的手段。目前的协议测试已经不仅仅是产品开发研制过程中一个简单的检测支持过程，而是发展成为计算机网络技术的一个重要分支。对协议测试技术的研究将直接影响到计算机网络技术的进步和世界网络市场的竞争与发展。所以，很多国家都投入了大量的人力、物力从事协议测试的研究工作，例如英国的国家物理实验室 NPL、法国国家通信研究中心、德国国家通信研究局 GMD、美国国家标准化研究局、美国新罕布什尔大学互操作研究实验室、中国清华大学计算机科学与技术系的计算机网络与协议测试实验室等单位都在这个领域投入了大量的研究力量。

协议测试是在软件测试的基础上发展起来的。根据对被测软件的控制观察方式，软件测试方法分为三种：白盒测试、黑盒测试和灰盒测试。协议测试是一种黑盒测试，它按照协议标准，通过控制观察被测协议实现的外部行为对其进行评价。目前，协议测试分成三个方面进行研究：一致性测试(Conformance Testing)、互操作性测试(Interoperability Testing)和性能测试(Performance Testing)。一致性测试主要测试协议实现是否严格遵循相应的协议描述；互操作性测试关注的是对于同一个协议标准，不同协议实现之间的连通问题。性能测试是用实验的方法来观测被测协议实现的各种性能参数，如吞吐量和传输延迟等，其结果往往与输入负载有关。

在上述三个方面，一致性测试开展最早，也形成了很多有价值的成果。1991 年，国际标准化组织 ISO 制定的国际标准 ISO9646——“OSI 协议一致性测试的方法和框架”，用自然语言描述了基于 OSI 七层参考模型的协议测试过程、概念和方法。但是，随着计算机网络技术的不断发展，新的协议越来越复杂，协议一致性测试工作遇到了很多困难。在这个过程中，大量形式化方法被引进到协议测试研究领域。1995 年，ISO 推出了“一致性测试中的形式化方法”国际标准，对协议一致性测试过程各个阶段使用的形式化方法进行了说明。但由于协议一致性测试本身的复杂性，使得该标准一直停留在草案阶段。对于互操作测试的研究技术基本上是从一致性测试继承过来的。由于对网络应用的需求急剧增长，网络性能已经变得与功能同等重要了。协议实现性能测试的研究工作也正在进行之中。在进行大量的测试实践的同时，理论研究也正在起步。

目前，国际协议测试研究领域已经取得了以下两点共识：

(1)理顺了协议一致性测试的过程。

(2)将形式化技术引入了协议测试领域，力图用严格的数学语言清晰、无二义性地研究协议测试的概念和方法。

但是，也发现这种方法存在着很多不足，其中最明显的就是这些理论研究与实际应用之间还存在着巨大的差距。

1.4 软件测试原则与特点

在明确软件测试的基本概念和方法后，接下来就是如何开展软件开发工作。本节讨论软件测试的一般性原则，以及软件测试工作的特点。

1.4.1 软件测试的原则

软件测试，从不同的角度出发会派生出两种不同的测试原则。从用户的角度出发，就是希望通过软件测试能充分暴露软件中存在的问题和缺陷；从开发者的角度出发，就是希望测试能表明软件产品不存在错误，已经正确地实现了用户的需求。

无论是站在用户角度，还是站在开发者角度，软件测试都应该从公正、中立和客观的角度开展软件测试工作，这也是软件测试的基本原则。为此应注意以下几点。

(1)应当把“尽早和不断地测试”作为开发者的座右铭。

(2)程序员应该避免检查自己的程序，测试工作应该由独立的专业的软件测试机构来完成。

(3)设计测试用例时，应该考虑到合法的输入和不合法的输入，以及各种边界条件，特殊情况下要制造极端状态和意外状态，比如网络异常中断、电源断电等情况。

(4)一定要注意测试中的错误集中发生现象，这和程序员的编程水平和习惯有很大的关系。

(5)对测试错误结果一定要有一个确认的过程。一般有 A 测试出来的错误，一定要有一个 B 来确认，严重的错误可以召开评审会进行讨论和分析。

(6)制订严格的测试计划，并把测试时间安排得尽量宽松，不要希望在极短的时间内完成一个高水平的测试。

(7)重复测试的关联性一定要引起充分的注意，修改一个错误而引起更多错误出现的现象并不少见。

(8)妥善保存一切测试过程文档，意义是不言而喻的，测试的重现性往往要靠测试文档。在把握上述原则的同时，还应该根据软件测试工作的特点开展测试工作。下面讨论软件测试的特点。

1.4.2 软件测试特点

1. 完全测试程序是不可能的

初涉软件测试的人可能认为拿到软件后就可以进行完全测试，找出所有软件的缺陷，并使得软检达到完美。遗憾的是，这是不可能的，即使最简单的程序也不行，主要有如下 4 个原因。

(1)输入量太大。

(2)输出结果太多。

(3)软件实现途径太多。

(4)软件说明书没有客观标准。

以上这些因素使得测试条件难以确定。

例如，图 1-5 是一个含有 4 个分支和一个至少执行 20 次的循环的程序结构图。如果要对这个程序进行穷举测试，就要把从 A 到 B 的各种路径全都测试一遍。不难看出，从 A 到 B 的不同路径共有 $5^1+5^2+5^3+\cdots+5^{20}\approx10^{40}$。这是一个相当大的计算量。

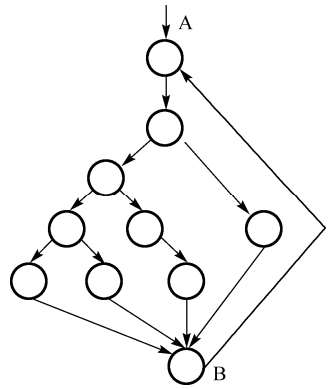


图 1-5　一个程序结构图

2．软件测试是有风险的行为

软件如果不测试就会有风险，是一个公认的事实。例如，千年虫问题，如果不及时发现和排除，就会给计算机关键领域带来灾难。然而，如果说对软件进行测试同样也会带来风险，就不是所有人都能理解的。为什么会同样有风险呢？

软件是个复杂系统，其复杂性体现在软件实现的内容复杂性、开发过程复杂性和组织工作复杂性上。而软件测试的目的是为了发现故障，并加以排除。对一个复杂的软件系统，故障的排除往往可能又会带来新的软件缺陷。所以，软件测试又会带来一定的风险。

3．测试无法显示潜在的软件缺陷

有时，软件中存在的缺陷是无法全部清查出来的，即使知道肯定会存在软件缺陷。例如，Y2K 问题，我们知道软件中存在设计缺陷，这些缺陷并不一定总是发生，但你还是要进行测试。对软件测试工作没有任何捷径，只有认真地测试下去。

4．发现的缺陷越多，说明软件缺陷越多

在软件测试中发现的缺陷越多，就说明软件中存在的缺陷越多，这是一个显然的事情，一个好的软件测试是能够发现足够多的软件缺陷和故障的。所以，发现的缺陷越多，说明软件中的缺陷越多，同时也说明软件测试工作或方法越好。

发现的缺陷越多，软件缺陷越多，但反之不一定成立。也就是说，软件缺陷多，并不一定能够都检查出来。如果一个软件的缺陷很多，而我们没有发现足够多的缺陷，则只能说明我们选择的测试方法和手段不好。

思考题

1．简述软件缺陷的含义。
2．说明软件缺陷、软件错误和软件失败的关系。
3．试评价一个软件测试的优劣的最主要指标。
4．“千年虫是不能被彻底清除的”这种说法是否正确？试说明原因。
5．黑盒测试是一种穷举测试，试说明穷举测试的含义。
6．简述 ALAC 测试的含义。
7．试比较渐进生命周期的测试工作和顺序生命周期的测试工作的共同点。
8．试从软件测试的范围角度，给出软件测试的分类。
9．简述单元测试的依据。
10．简述确认测试的测试目标。
11．给出 α 测试的执行者，并说明原因。
12．简述面向对象的软件测试的对象及采用的基本测试方法。
13．简述协议测试的内容以及采用的基本测试方法。
14．软件测试是有风险的工作，试解释这种说法的含义。

第 2 章　软件测试基础

本章讨论软件测试的基本方法，重点讨论软件测试黑盒法和白盒法；并对软件测试的过程进行讨论，重点讨论软件测试的设计和测试文档的使用。通过本章的学习，掌握软件测试的基本方法，并对软件测试作为一个工程组织所必要工程化方法有全面的了解。

按测试的范围和级别，软件测试可分为单元测试、综合测试、高级测试。无论是哪一级测试，所采用的方法基本有两种：白盒法(结构测试)和黑盒法(功能测试)。前者是针对系统内部实现的测试，而后者侧重于系统的外部功能和特性。

2.1　软件测试白盒法

白盒法是以程序的内部逻辑为依据。合理的白盒测试，就是要选取足够的测试用例，对源代码进行比较充分的覆盖，以便尽可能多地发现程序中的错误。

2.1.1　逻辑覆盖法

逻辑覆盖是一组覆盖方法的总称。按照由低到高对程序逻辑的覆盖程度，又可区分为以下几种覆盖。

- 语句覆盖：使被测试程序的每条语句至少执行一次。
- 判断覆盖：使被测试程的每一分支至少执行一次，故又称分支覆盖。
- 条件覆盖：要求判断中的每一个条件按“真”、“假”两种结果至少执行一次。
- 条件组合覆盖：这是覆盖程度比前 3 种都强的一种覆盖，它与条件覆盖的区别是它不是简单地要求每个条件都出现“真”与“假”两种结果，而是要求让这些条件的所有可能组合都至少出现一次。
- 判断/条件覆盖：它使判断中的每个条件取得各种可能值，并使每个判断也取得“真”与“假”的结果。

下面通过一个实例说明上述各种覆盖方法的含义及测试用例的设计。

已知一个被测模块的程序结构如图 2-1 所示，设符号“∧”表示“and”运算，“∨”表示“or”运算，上画线“¯”表示“非”运算。

由图 2-1 可知，该程序共有 4 条不同的路径，L_1(a→c→e)、L_2(a→b→d)、L_3(a→b→e)、L_4(a→c→d)，也可以简写成 ace、abd、abe、和 acd。其推导过程如下：

L_1(a→c→e)

=｛(A>1) and (B=0)｝and ｛(A=2) or (X/A>1)｝

=(A>1) and (B=0) and (A=2) or (A>1) and (B=0) and (X/A>1)

=(A=2) and (B=0) or (A>1) and B=0 and (X/A>1)

L_2(a→b→d)

=｛$\overline{(A>1)}$ and $\overline{(B=0)}$｝and ｛$\overline{(A=2)}$ or $\overline{(X>1)}$｝

=｛(A>1) or (B=0)｝and ｛(A=2) and (X>1)｝

=(A>1) and (A=2) and (X>1) or (B=0) and (A=2) and (X>1)

=(A<=1) and (X<=1) or (B<>0) and (A<>2) and (X<=1)

L_3(a→b→e)

={(A>1) and (B=0)} and {(A=2) or (X>1)}

={(A>1) or (B=0)} and {(A=2) or (X>1)}

=(A<=1) and (X>1) or (B<>0) and (A=2) or (B<>0) and (X>1)

L_4(a→c→d)

={(A>1) and (B=0)} and {(A=2) or (X/A>1)}

=(A>1) and (B=0) and (A<>2) and (X/A<=1)

在以上的逻辑式中，由符号“and”连接起来的断言就是为了指明在遍历该路径时各个输入变量应取值的范围；符号“or”则划分了几组可选的取值。依据上述导出结果即可设计满足要求的测试用例。

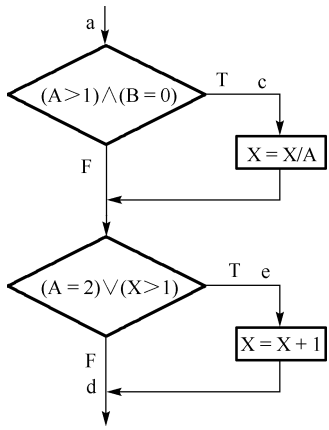


图 2-1　一个被测模块的程序结构

1．语句覆盖

语句覆盖的含义是指在测试的过程中，软件测试者应选择足够多的测试用例，使被测试程序中每个语句至少执行一次。

例如，在如图 2-1 所示的流程图中，正好所有的可执行语句都在路径 L_1 上，故选择路径 L_1 设计测试用例，就可以覆盖所有的可执行语句。

满足本例的测试用例是：[(2，0，4)，(2，0，3)]覆盖 ace[L_1]。

本测试用例实际上只测试了条件为真的情况，如果条件为假，则使用本测试用例显然不能发现问题。此外，当第一个判断中的逻辑符“∧”写成“∨”，或者第二个判断中的逻辑符号“∨”写成“∧”时，本测试用例也不能查出上述错误。所以，语句覆盖是最弱的逻辑覆盖准则。

2．判断覆盖

判断覆盖的含义是指在测试的过程中，软件测试者应设计若干测试用例，并运行所测程序，使被测试程序中每个判断的真分支和假分支至少经历一次。

例如，在如图 2-1 所示的流程图中，如果选择路径 L_1 和 L_2，则可满足判断覆盖，其测试用例如下：

[(2，0，4)，(2，0，3)]覆盖 ace[L_1]

[(1，1，1)，(1，1，1)]覆盖 abd[L_2]。

如果选择路径 L_3 和 L_4，则可得另一组测试用例：

[(2，1，1)，(2，1，2)]覆盖 abe[L_3]

[(3，0，3)，(3，1，1)]覆盖 acd[L_4]。

由此看来，测试用例的取法并不是唯一的。此外，若把如图 2-1 所示的流程中的第二个判断中的条件 X>1 错写成 X<1，那么利用上面两组测试用例，仍能得到同样的结果。这表明：只是判断覆盖不能确保一定能查出在判断的条件中存在的错误。

以上只讨论了两个出口的判断，还应将判定覆盖推广到多出口判断，如用 case 语句可进行多出口判断。

3．条件覆盖

用条件覆盖所设计的测试用例可使得程序中的每一个判断的每一个条件的可能取值至少执行一次。例如，在如图 2-1 所示的流程中，事先可对所有条件的取值加以标注，比如：

- 对第一个判断，若条件 A>1 成立，则取真值为 T_1，反之，取假值为～T_1；若条件 B=0 成立，则取真值为 T_2，反之，取假值为～T_2。
- 对第二个判断，若条件 A=2 成立，则取真值为 T_3，反之，取假值为～T_3；若条件 X>1 成立，则取真值为 T_4，反之，取假值为～T_4，可选测试用例如表 2-1 所示。

表 2-1　条件覆盖测试用例

测试用例	通过路径	条件取值	覆盖分支
[(2,0,4)(2,0,3)]	ace(L_1)	T_1 T_2 T_3 T_4	c,e
[(1,0,1)(1,0,1)]	abd(L_2)	～T_1 T_2～T_3～T_4	b,d
[(2,1,1)(2,1,2)]	abe(L_3)	T_1～T_2 T_3～T_4	b,e

或

测试用例	通过路径	条件取值	覆盖分支
[(1,0,3)(1,0,4)]	abe(L_3)	～T_1 T_2～T_3 T_4	b,e
[(2,1,1)(2,1,2)]	abe(L_3)	T_1～T_2 T_3～T_4	b,e

比较这两组测试用例可以发现，第一组测试用例不仅覆盖了所有判断的取真分支和取假分支，而且覆盖了判断中条件的可能取值；第二组测试用例虽然满足了条件覆盖，但由于只覆盖了第一个判断的取假分支和第二个判断的取真分支，不满足判断覆盖的要求。为此，必须引入更强的覆盖，即判断—条件覆盖。

4．判断—条件覆盖

用判断—条件覆盖所设计的测试用例能够使得判断中每一个条件的所有可能取值至少执行一次，同时每个判断的所有可能判断结果至少执行一次。

例如，对于如图 2-1 所示的流程中的各个判断，若 $T_1T_2T_3T_4$ 及～T_1～T_2～T_3～T_4 的含义如前所述，则只需设计以下两个测试用例便可覆盖图 2-1 的 8 个条件取值、4 个判断分支。

判断—条件覆盖表如表 2-2 所示。

表 2-2　判断—条件覆盖表

测试用例	通过路径	条件取值	覆盖分支
[(2,0,4)(2,0,3)]	ace(L_1)	T_1 T_2 T_3 T_4	c,e
[(1,1,1)(1,1,1)]	abd(L_2)	～T_1～T_2～T_3～T_4	b,d

判断—条件覆盖也有缺陷。从表面上看，它测试了所有条件的取值，但事实并非如此。这是由于某些条件覆盖了另一些条件所致。比如，对于条件表达式(A>1)and (B=0)来说，若(A>1)的测试结果为真，则还要测试(B=0)，才能决定表达式的值；而若(A>1)的测试结果为假，可以立刻确定表达式的结果为假。这时，往往就不再测试(B=0)的取值了，因此，条件(B=0)就没有检查。同样，对于条件表达式(A=2)or (X>1)来说，若(A=2)的测试结果为真，就可以立刻确定表达式的

结果为真。这时，条件(X>1)就没有检查。因此，采用判断—条件覆盖，也不一定能查出逻辑表达式中的错误。

为彻底地检查所有条件的取值，可以将图 2-1 给出的多重条件判定分解，形成图 2-2 表示的由多个基本判断组成的流程图，这样就可以有效地检查所有的条件了。

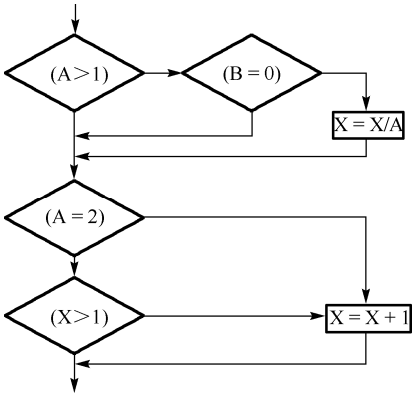


图 2-2　分解为基本判断的例子

5．条件组合覆盖

用条件组合覆盖所设计的测试用例能够使得每个判断的所有可能的条件取值组合至少执行一次。

现在仍来考察图 2-1 所给出的例子，先对各个判断的条件取值组合加以标记。例如，记：

(1) A>1，B=0 为 T_1 T_2，是第一个判断取真值的分支；

(2) A>1，B<>0 为 T_1～T_2，是第一个判断取假值的分支；

(3) A<=1，B=0 为～T_1 T_2，是第一个判断取假值的分支；

(4) A<=1，B<>0 为～T_1～T_2，是第一个判断取假值的分支；

(5) A=2，X>1 为 T_3 T_4，是第二个判断取真值的分支；

(6) A=2，X<=1 为 T_3～T_4，是第二个判断取真值的分支；

(7) A<>2，X>1 为～T_3 T_4，是第二个判断取真值的分支；

(8) A<>2，X<=1 为～T_3～T_4，是第二个判断取假值的分支。

对于每个判断，要求所有可能的条件取值的组合都必须取到，在如图 2-1 所示的流程中的每个判断各有两个条件，各有 4 个条件取值的组合。取如表 2-3 所示的 4 个测试用例，就可以覆盖上述 8 种条件取值的组合。这组测试用例覆盖了所有条件的可能取值的组合，覆盖了所有判断的可取分支，但路径漏掉了 L_4。测试还不完全。

表 2-3　条件覆盖测试用例

测试用例	通过路径	覆盖条件	覆盖组合号
[(2,0,4) (2,0,3)]	ace(L_1)	T_1 T_2 T_3 T_4	(1) (5)
[(2,1,1) (2,1,2)]	abe(L_2)	T_1～T_2 T_3～T_4	(2) (6)
[(1,0,3) (1,0,4)]	abe(L_3)	～T_1 T_2～T_3 T_4	(3) (7)
[(1,1,1) (1,1,1)]	abd(L_2)	～T_1～T_2～T_3～T_4	(4) (8)

说明：这里并未要求第一个判断的 4 个组和与第二个判断的 4 个组和再进行组合。要是那样的话，就需要 2^4=16 个测试用例。

6．路径测试

路径测试就是设计足够的测试用例，覆盖程序中所有可能的路径。若还是以图 2-1 为例，则可以选择如表 2-4 所示的一组测试用例来覆盖该程序段的全部路径。

表 2-4　路径测试的测试用例

测试用例	通过路径	覆盖条件
[(2,0,4) (2,0,3)]	ace(L_1)	T_1 T_2 T_3 T_4
[(1,1,1) (1,1,1)]	abd(L_2)	～T_1～T_2～T_3～T_4
[(1,1,2) (1,1,3)]	abe(L_3)	～T_1～T_2～T_3 T_4
[(3,0,3) (3,0,1)]	acd(L_4)	T_1 T_2～T_3～T_4

2.1.2 基本路径测试法

基本路径测试是由 TOM McCabe[MCC76]首先提出的一种白盒测试技术，基本路径测试法允许测试用例设计者导出一个过程设计的逻辑复杂性测度，并使用该测度作为指南来定义执行路径的基本集。从该基本集导出的测试用例保证对程序中的每一条语句至少执行一次。

1．流图符号

在介绍基本路径测试法之前，必须先介绍一种简单的控制流表示方法，即流图或程序图。流图使用图 2-3 中的符号描述逻辑控制流，每一种结构化构成元素有一个相应的流图符号。

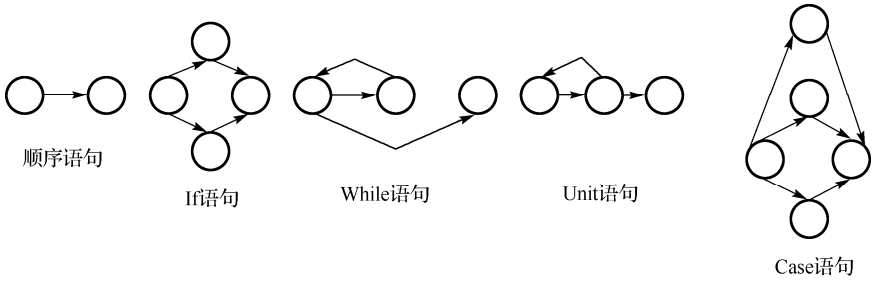


图 2-3 流图符号

为了说明流图的用法，我们采用图 2-4(a)中的过程设计表示法。其中，流程图用来描述程序控制结构。图 2-4(b)将流程图映射到一个相应的流图(假设流程图的菱形决策框中不包含复合条件)。在图 2-4(b)中，每一个圆称为流图的节点，代表一个或多个语句。一个处理方框序列和一个菱形决策框可被映射为一个节点。流图中的箭头，称为边或连接，代表控制流，类似于流程图中的箭头。一条边必须终止于一个节点，即使该节点并不代表任何语句。由边和节点限定的范围称为区域。计算区域时应包括图外部的范围。

任何过程设计表示法都可以被翻译成流图，图 2-5 显示了一个程序设计语言(Program Design Language，PDL)片断及其对应的流图。注意，对 PDL 语句进行了编号，并将相应的编号用于流图中。

程序设计中遇到复合条件时，生成的流图变得更为复杂。当条件语句中用到一个或多个布尔运算符(逻辑 OR，AND，NAND，NOR)时，就出现了复合条件。在图 2-6 中，例如将一个包含复合条件的 PDL 片段翻译为流图。注意，为语句 IF a or b 中的每一个 a 和 b 创建了一个独立的节点，包含条件的节点被称为判定节点，从每一个判定节点发出两条或多条边。

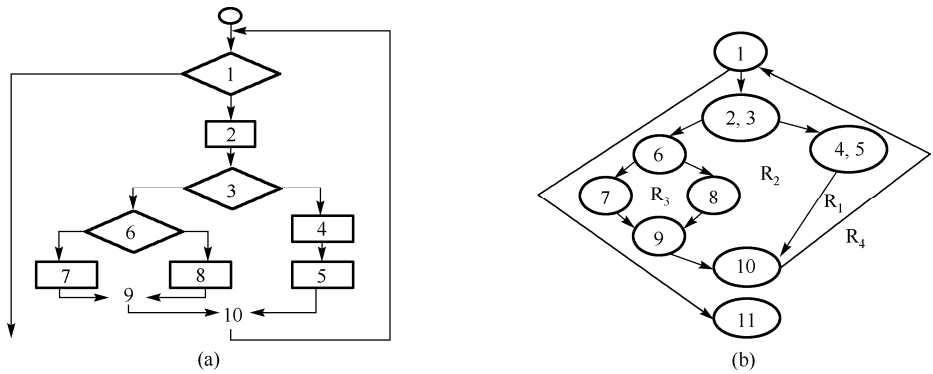


图 2-4 程序流程图

2．环形复杂性

环形复杂性是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于基本路径法，计算所得的值定义了程序基本集的独立路径数量，并为我们提供了确保所有路径至少执行一次的测试数量的上界。

独立路径是指程序中至少引进一个新的处理语句集合或一个新条件的任一路径。采用流图的术语，即独立路径必须至少包含一条在定义路径之前不曾用到的边。例如，图 2-4(b)中流图的一个独立路径集合如下。

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，每一条新的路径都包含了一条新边。路径 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 不是独立路径，意味它只是已有路径的简单合并，并未包含任何新边。

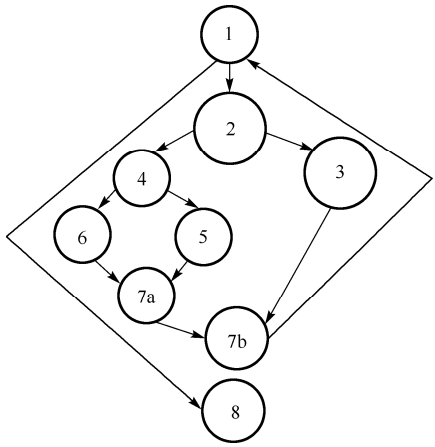


图 2-5　将 PDL 翻译成流图

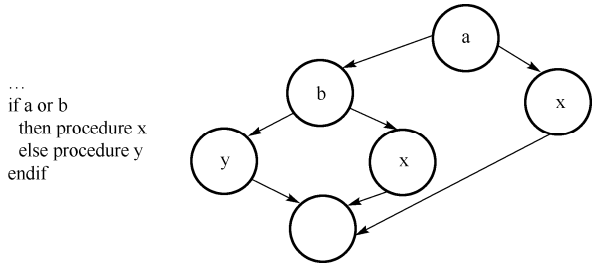


图 2-6　复合逻辑

上面定义的路径 1，2，3 和 4 包含了如图 2-4(b)所示的流图的一个基本集。简而言之，如果能将测试设计为强迫运行这些路径(基本集)，那么程序中的每一条语句将至少被执行一次。每一个条件执行时都将分别取 true 和 false。应该注意到基本集并不唯一。实际上，给定的过程设计可派生出任意数量的不同基本集。

如何才能知道需要寻找多少路径呢？对环形复杂性的计算提供了这个问题的答案。环形复杂性以图论为基础，为我们提供了非常有用的软件度量。可用以下三种方法之一来计算复杂性。

- 流图中区域的数量对应于环形的复杂性。
- 给定流图 G 的环形复杂性 V(G)，定义为 V(G)=E–N+2，E 是流图中边的数量，N 是流图节点数量。
- 给定流图 G 的环形复杂性 V(G)，也可定义为 V(G)=P+1，P 是流图 G 中判定节点的数量。

再回到图 2-4。可采用上述任意一种算法来计算环形复杂性。

- 流图有 4 个区域。
- V(G)=11 条边–9 个节点+2=4。
- V(G)=3 个判定节点+1=4。

因此，图 2-4(b)的环形复杂性是 4。

环形复杂性 V(G)的值提供了组成基本集的独立路径的上界，由此得出覆盖所有程序语句所需的最少测试用例数量的上界。

3．导出测试用例

基本路径测试方法可用于过程设计或源代码生产。下面给出基本路径测试法的步骤，图 2-7 中的 PDL 所描述的过程“求平均值”将被用于阐明测试用例设计方法中的各个步骤。注意，“求平均值”虽然是一个简单的算法，但是仍然包含了复合条件和循环。

```
    Procedure average;//计算不超过 100 个数字的平均值;同时计算总和与有效数字个数
    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, maxImun, sum IS SCALAR
    TYPE I IS INTEGER;
    I=1;Sum=0;
1   Total.input=total.valid=0;          (2)          (3)
    DO WHILE value[i]<>-999 and total.input<100
4    Increment total.input by 1;                     (6)
5    IF value[i]>=minimum AND value[i] <=maximum
     7  THEN  increment total.valid by1;
              Sum=sum+valid[i]
        ELSE skip
8   ENDIF
    Increment I by 1;
9    ENDDO                  (10)
    IF total.valid>0
         11  THEN average=sum/total.valid;
         12  ELSE average=-999;
    13 ENDIF
    END average
```

图 2-7　测试用例设计的 PDL 的节点已经标识

步骤 1：以设计或代码为基础，画出相应的流图。创建一个流图，参考图 2-7 中“求平均值”的 PDL。创建流图时，要对将被映射为流图节点的 PDL 语句进行标号，图 2-8 显示了对应的流图。

步骤 2：确定结果流图的环形复杂性。可采用上一节的任意一种算法来计算环形复杂性 V(G)。

应该注意到，计算 V(G)并不一定要画出流图，计算 PDL 中的所有条件语句数量(过程求平均值中复合条件语句计数为 2)，然后加 1 即可得到环形复杂性。在图 2-8 中，

V(G)=6 个区域

V(G)=18 条边–14 个节点+2=6。

V(G)=5 个判定节点+1=6。

步骤 3：确定线性独立的路径的一个基本集。V(G)的值提供了程序控制结构中线性独立的路径的数量，在过程求平均值中，我们指定以下 6 条路径。

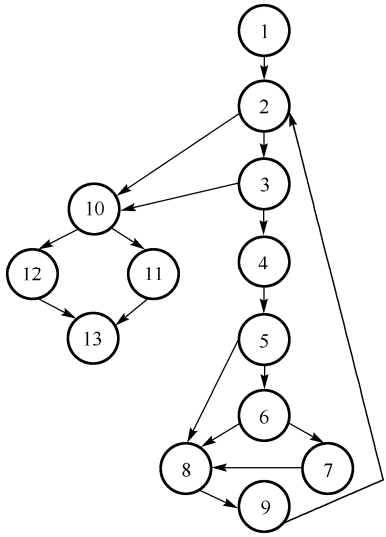


图 2-8　过程求平均值的流图

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 和 6 后面的省略号(…)表示可以加上控制结构其余部分的任意路径。通常在导出测试用例时，识别判定节点是十分必要的。本例中，节点 2、3、5、6 和 10 是判定节点。

步骤 4：准备测试用例，强制执行基本集中每条路径。测试人员可选择数据，以便在测试每条路径时适当设置判定节点的条件。满足上述基本集的测试用例如下。

路径 1 测试用例：

value(k) = 有效输入，其中 k < i。

value(i) = –999，其中 2≤i≤100。

期望结果：基于 k 的正确平均值和总数。

注意：路径 1 无法独立测试，必须作为路径 4、5 和 6 测试的一部分。

路径 2 测试用例：

value(1) =–999

期望结果：平均值=–999；其他按初值汇总。

路径 3 测试用例：

试图处理 101 或更大的值。

前 100 个数值应该有效。

期望结果：与测试用例 1 相同。

路径 4 测试用例：

value(k) <最小值，其中 k < i。

value(i) =有效输入，其中 i≤100。

期望结果：基于 k 的正确平均值和总数。

路径 5 测试用例：

value(k) >最大值，其中 k ≥ i。

value(i) = 有效输入，其中 I<100。

期望结果：基于 k 的正确平均值和总数。

路径 6 测试用例：

value(i) =有效输入，其中 I<100。

期望结果：基于 k 的正确平均值和总数。

执行每个测试用例，并和期望值比较，一旦完成所有测试用例，测试者可以确定在程序中的所有语句至少被执行一次。

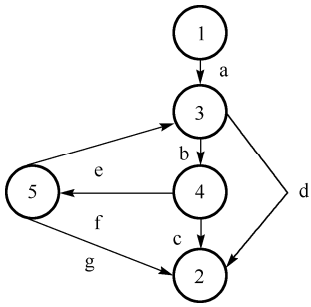
重要的是，要注意某些独立路径(如例子中的路径 1)不能以独立的方式被测试，即穿越路径所需的数据组合不能形成程序的正常流。在这种情况下，这些路径必须作为另一个路径测试的一部分来进行测试。

4．图矩阵

导出流图和决定基本测试路径的过程均需要机械化，为了开发辅助基本路径测试的软件工具，称为图矩阵(Graph Matrix)的数据结构很有用。

图矩阵是一个正方形矩阵，其大小(即行数和列数)等于流图的节点数。每列和每行都对应于标识的节点，矩阵项对应于节点间的连接(边)，图 2-9 显示了一个简单的流图及其对应的图矩阵。

在该图中，流图的节点以数字标识，边以字母标识，矩阵中的字母项对应于节点间的连接，例如边 b 连接节点 3 和 4。



		a		
	d		b	
	c			F
	g	e		

图矩阵

图 2-9　连接矩阵

这里，图矩阵只是流图的表格表示。然而，对每一个矩阵项加入连接权值，图矩阵就可以用于在测试中评估程序的控制结构，连接权值为控制流提供了另外的信息。在最简单情况下，连接权值是 1(存在连接)或 0(不存在连接)。但是，连接权值可以赋予更有趣的属性：

- 穿越连接的处理时间。
- 穿越连接时所需的内存。
- 执行连接的概率。
- 穿越连接时所需的资源。

举例来说，我们用最简单的权值(0 或 1)来标示连接，将图 2-9 的图矩阵重画为图 2-10。字母替换为 1，表示存在边(为清晰起见，没有画出 0)，这种形式的图矩阵称为连接矩阵。图 2-10 中，含两个或两个以上的行表示判定节点。所以，右边所示的算术计算就提供了另一种环形复杂性计算的方法。

1	2	3	4	5	连接
		1			1−1=0
	1		1		2−1=1
	1			1	2−1=1
	1	1			2−1=1

图 2-10　连接矩阵

2.2　软件测试黑盒法

黑盒法以程序的功能作为测试依据。黑盒测试并不是白盒测试的替代品，而是用于辅助白盒测试，从而发现其他类型的错误。白盒测试在测试的早期执行，而黑盒测试则主要用于测试的后期。

黑盒测试用于发现以下类型的错误：

- 功能不符合要求或遗漏。
- 界面错误。
- 数据结构或外部数据库访问错误。
- 性能偏差。
- 初始化或终止错误。

测试用例回答下列问题：

- 如何测试功能的有效性？
- 何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敏感？
- 如何分隔数据类的边界？
- 系统能够承受何种数据率和数据量？
- 特定类型的数据组合会对系统产生何种影响？

运用黑盒测试，要导出满足以下标准的测试用例集：

- 所设计的测试用例能够减少达到合理测试所需的附加测试用例数。
- 所设计的测试用例能够告知某些类型错误的存在与不存在，而不仅仅是告知与特定测试相关的错误。

2.2.1　等价类划分法

等价类划分是一种黑盒测试方法。它是将程序的输入域划分为数据类，以便导出测试用例。理想的测试用例是能用一个用例发现一类错误，等价类划分试图定义一个测试用例，以发现各类错误，从而减少测试用例数。

如果对象由具有对称性、传递性或自反性的关系连接，就意味着存在等价类。因此，等价类是指某个输入域的子集合。在该集合中，各输入数对揭露程序中的错误都是等效的。如果将某个等价类的一个输入条件作为测试数据进行测试并查出了错误，那么使用这一等价类中的一个输入条件进行测试也会查出同样的错误；反之，若使用某个等价类中的一个输入条件作为测试数据进行测试没有查出错误，则使用这个等价类中的其他输入条件也同样查不出错误。因此，把全部输入数据合理地划分为若干等价类，在每一个等价类中取一个数据作为测试的输入条件，就可以使用少量代表性数据，取得较好的测试效果。

等价类划分要考虑以下两种情况：

- 有效等价类。对于程序的规格说明来说，有效等价类是合理的、有意义的输入数据构成的集合。利用它可以检验程序是否实现了规格说明预先规定的功能和性能。
- 无效等价类。对于程序的规格说明来说，无效等价类是不合理的、无意义的输入数据构成的集合。这一类测试用例主要用于检测程序中的功能和性能是否有不符合规格说明要求。

在设计测试用例时，必须同时考虑有效等价类和无效等价类的设计，只有经过这样测试的软件才能达到较高的可靠性。

1．确定有效等价类

划分等价类是使用等价类的关键。以下结合具体实例给出几条确定等价类的原则。

- 如果输入条件中规定了取值范围或值的个数，则可以确立一个有效等价类和两个无效等价类。例如，在程序的规格说明中，若对输入条件有一个规定：“…项数可以从 1 到 999…”则有效等价类是“1<=项数<=999”，两个无效等价类是“项数<1”或“项数>999”。
- 如果输入条件中规定了输入值的集合，或者是规定了“必须如何…”的条件，则可确立一个有效等价类和一个无效等价类。例如，若在 Pascal 语言中对变量标识符规定为“以字母打头的…串”，那么所有以字母打头的构成有效等价类，而不以字母打头的归于无效等价类。
- 如果输入条件是一个布尔量，则可以确定一个有效等价类和一个无效等价类。
- 如果规定了输入数据的一组值，而且程序要对每个输入值分别进行处理，则可以每个输入值确定一个有效等价类，并针对这组值确立一个无效等价类，即所有不允许的输入值的集合。
- 如果给定了输入数据必须遵守的规则，则可以确立一个有效等价类(符合规则)和若干个无效等价类(从不同角度违反规则)。例如，Pascal 语言规定“一个语句必须以分号‘;’结束”，据此就可以确定一个有效等价类“以‘;’结束”，若干个无效等价类“以‘;’结束”、“以‘,’结束”、“以‘ ’结束”、“以 LF 结束”，等等。
- 如果确知已经划分的等价类中各元素在程序中的处理方式不同，则应将此等价类进一步划分成更小的等价类。

2．确立测试用例

在确立了等价类之后，建立等价类表，列出所有划分出的等价类，再从划分出的等价类中按以下原则选择测试用例。

- 为每一个等价类规定一个唯一的编号。
- 设计一个新的测试用例，使其尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步，直到所有的有效等价类都被覆盖为止。
- 设计一个新的测试用例，使其仅覆盖一个尚未被覆盖的无效等价类，重复这一步，直到所有的无效等价类都被覆盖为止。之所以要这样做，是因为在某些程序中对某一输入错误的检查往往会屏蔽对其他输入错误的检查。因此必须针对每一个无效等价类，分别设计测试用例。

3．用等价类划分法设计测试用例

下面用一个例子说明如何用等价类划分法设计测试用例。某一 Pascal 语言规定：“标识符是由字母开头、后接字母或数字的任意组合而成的，有效字符数为 8 个，最大字符数为 80 个。”并且规定：“标识符必须先说明、再使用。”“在同一说明语句中，标识符至少必须有一个。”

遵循划分等价类的原则，建立满足上述要求的等价类表，如表 2-5 所示。

表 2-5　等价类表

输入条件	有效等价类	无效等价类
标识符个数	1 个(1)，多个(2)	0 个(3)
标识符字符数	1-8 个(4)	0 个(5)，〉8 个(6)，80 个(7)
标识符组成	字母(8)，数字(9)	非字母数字字符(10)，保留字(11)
第一个字符	字母(12)	非字母(13)
标识符使用	先说明后使用(14)	未说明已经使用(15)

从中选择 9 个测试用例，它们覆盖了所有的等价类：

1) VAR x,T1234567:REAL; }(1)，(2)，(4)，(8)，(9)，(12)，(14)
　BEGIN x:=3.414;T1234567:=2.732;…
2) VAR: REAL; }(3)
3) VAR x.: REAL; }(5)
4) VAR T12345678: REAL; }(6)
5) VAR T12345…REAL; }(7)　多于 80 个字符
6) VAR T$:CHAR;}(10)
7) VAR GOTO:INTEGER:}(11)
8) VAR 2T:REAL;}(13)
9) VAR APR:REAL;}(15)
　BEGIN…
　PAP:=SIN(3.14*0.8)/6;

2.2.2　边界值分析

边界值分析方法是对等价类划分法的补充。

人们从长期的测试工作中总结出经验：大量的错误发生在输入或输出范围的边界上，而不是在输入范围的区间内部。因此，针对各种边界情况设计测试用例，可以查出更多的错误。

使用边界分析方法设计测试用例，首先应确定边界情况。通常,输入等价类与输出等价类的边界，就是应着重测试的边界情况。应当选择正好等于、刚刚大于或刚刚小于边界的值作为测试数据，而不是选择等价类中的典型值或任意值作为测试数据。

为此，边界值分析的测试用例选择有以下原则。

(1) 如果输入条件规定了值的范围，则应取刚达到这个范围的边界的值，以及刚刚超过这个范围边界的值作为测试输入数据。例如，若输入值的范围是“–0.1～1.0”，则可选取“–0.101”，“0.009”，“–0.009”，“1.001”作为测试输入数据。

(2) 如果输入条件规定了值的个数，则用最大个数、最小个数、比最大个数多 1、比最小个数少 1 的数作为测试数据。例如，若一个输入文件可有 1～255 个记录，则可以分别设计有一个记录、255 个记录以及 0 个记录和 256 个记录的输入文件。

(3) 根据规格说明的每个输出条件，使用原则(1)。例如，某程序的功能是计算折扣量，若最低折扣量是 0 元，最高折扣量是 1050 元，则设计一些测试用例，使它们恰好产生 0～1050 元的结果。此外，还可以考虑设计结果为负值或大于 1050 元的测试用例。由于输入值的边界不与输出值的边界项对应，所以要检查输出值的边界不一定可行，要产生超出输出值值域之外的结果也不一定可行。尽管如此，在必要时还是可以尝试一下。

(4) 根据规格说明的每个输出条件，使用原则(2)。例如，一个信息检索系统根据用户输入的命令，显示有关文献的摘要，但最多只显示 4 篇摘要。这时可以设计一些测试用例，使得程序分别显示 1 篇、4 篇、0 篇摘要，并设计一个有可能使程序错误地显示 5 篇摘要的测试用例。

(5) 如果程序的规格说明给出的输入域或输出域是有序集合(如有序表、顺序文件等)的，则应选取集合的第一个元素和最后一个元素作为测试用例。

(6) 如果程序中使用了一个内部数据结构，则应当选择这个内部数据结构的边界上的值作为测试用例。例如，如果程序中定义了一个数组，其元素下标的下界是 0，上界是 100，那么应该选择达到这个数组下标边界的值，如 0 与 100，作为测试用例。

(7) 分析规格说明书，找出其他可能的边界条件。

2.3 小结

在上面所述传统软件测试方法中，黑盒法是以程序的功能作为测试的内容，测试用例是根据程序的输入变量来设计的。黑盒法通常在系统投入使用前，在综合测试以及高级测试中使用。黑盒法的不足是，测试用例的选择只考虑了程序的输入，以及在该情况下的输出，并没有考虑程序的内部结构。因此，程序内部结构是否规范、结构化程度的好坏、系统的性能的高低等都得不到测试。白盒法以程序的内部逻辑为依据，测试用例根据程序的流程图或程序图(路径法)来确定，完全的白盒测试应该使选取的测试用例覆盖所有的路径。然而，完全的白盒法是不可能的，这也是白盒法的不足之处之一。白盒法的另一个不足是，它没有测试程序的外部功能。

思考题

1．试按逻辑覆盖程度给出语句覆盖、条件覆盖、判断覆盖、条件组合覆盖、判断/条件的由低到高的排列次序。

2．条件覆盖是否高于判断覆盖的逻辑覆盖程度？如果不是，则给出反例加以说明。

3．条件覆盖能否把所有的条件都覆盖？如果不能，则给出反例加以说明。

4．已知某种计算机程序设计语言的标识符语法规则规定“标识符由非数字开头的，有效字符数为 32 个，最大字符数为 128 个的任意符号串”。试给出用等价类划分法设计的测试用例。

5．已知一个变量的类型为 double，它的取值范围为“–0.2～2.2”，试给出边界值法的测试用例。

6．已知程序框图如图 2-11 所示，试分别给出(a)语句覆盖，(b)条件覆盖，(c)判断覆盖，(d)条件组合覆盖，(e)判断/条件的最小测试用例和路径。

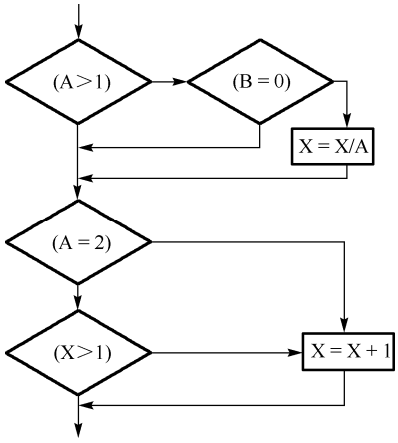


图 2-11 分解为基本判断的例子

7．已知程序如图 2-12 所示，根据该流程图试(a)给出对应的流图，(b)计算环形复杂性 V(G)，(c)求独立路径构成的基本集合，(d)给出每条路径执行的期望结果，(e)给出每一条路径的测试用例。

```
main() //a c code about finding out the large one in tree input number
{int num1,num2,num3,max;
  printf("Please input three numbers:");
  scanf("%d,%d,%d",&num1,&num2,&num3);
  if (num1>num2)
       max=num1;
  else
       max=num2;
  if (num3>max)
       max=num3;
  printf("The three numbers are:%d,%d,%d\n",num1,num2,num3);
  printf("max=%d\n",max);
}
```

图 2-12 一个计算三个数中最大数的程序

第 3 章　TTCN 树表描述语言程序设计

OSI/ITU 组织颁布的协议一致性测试基本框架和方法标准(ISO/IEC 9646（ITU X.290 series)由五大部分构成，树表描述语言(Tree Tabular Combine Notation)是其中的第三部分，即 ISO/IEC 9646-3。该标准的颁布为通信协议的一致性测试提供了准则，所以 TTCN 得到了广泛的应用。下面分别介绍 TTCN 的基本语法、符号的语义及 TTCN 的应用。

3.1　协议一致性测试基础框架

3.1.1　协议一致性测试系统结构

在一致性测试中一个被测试部分(Implement Under Test，IUT)是一个 OSI 协议实体，IUT 所在的系统称为被测试系统(System Under Test，SUT)。一个概念上的一致性测试系统结构如图 3-1 所示。

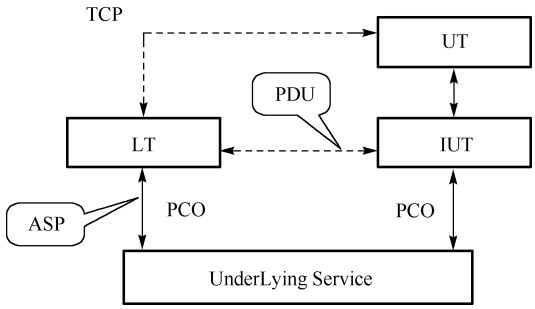


图 3-1　一致性测试系统结构 CTMF

IUT 有一个上层测试(Upper Test)接口和下层测试(Low Test)接口，UT 和 LT 通过控制观察点(Points of Control and Observation，PCO)对系统进行测试。通常，LT 是远程可访问接口，因此 IUT 定义一个远端的 PCO，即底层接口被设置在远端。通信被认为是异步通信，所以在每一个 PCO 都对应两个队列(FIFO)，一个是输入，另一个是输出。在一致性测试方法框架(Conformance Testing Methodology Framework，CTMF)中，严格区分上层测试和下层测试功能，IUT 的上层测试由 UT 控制，下层测试由 LT 控制。在测试过程中，UT 扮演一个用户来使用 IUT 提供的功能，而 LT 模仿一个 IUT 下层的通信实体。也就说，UT 与 IUT 的交互是通过 LT 来实现的。

IUT 和 UT 之间通过抽象服务原语(Abstract Service Primitives，ASP)进行通信。从概念角度，IUT 和 LT 通过协议数据单元(Protocol Data Unit，PDU)交换数据；从实际角度，PDU 采用 ASP 对基本服务动作进行编码，即 PDU 不是直接进行交互，而是 CTMF 允许根据 PDU 的编码进行交互，即在一个抽象的测试中使用 PDU 进行交换，所以 ASP 与 PDU 不再加以区分。

正如图 3-1 所示，测试协调过程(Test Coordination Procedures，TCP)来协调 LT 和 UT 的动作，这在 LT 和 UT 两个独立的过程中是十分必要的。图 3-1 仅表现了一致性测试系统(CTMF)结构，实际中的测试系统可根据采用的测试方法的不同做相应的变化。在 CTMF 中，测试方法可分为局

部的、分布的、协调的和远程的。它们的主要不同是对 LT 和 UT 的协调以及对它们的控制与观察程度的不同。

图 3-2 是一个基于 CTMF 的测试过程。一个 IUT 首先由测试用例的触发条件激活，并从稳定状态进入被测试状态；经过测试用例在测试体中运行，进入结束状态；如果执行的结果不唯一，则需要进一步检查分析结果中存在的问题，从而进入 End State（Verification）状态；根据检查结果提出反馈，进入下一次的测试阶段。在上面的测试过程中，如果测试例的结束状态相同，则直接进入到下一次测试过程。

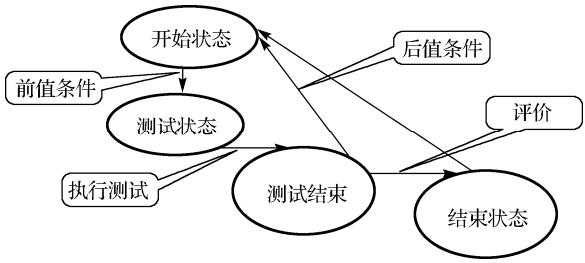


图 3-2　测试例方案

3.1.2　X-协议一致性测试

为了解释 TTCN 一致性测试的过程，本节给出一个假想的协议，称为 X-协议；并给出协议一致性测试的要求。

结合 CTMF 的 X-协议执行脚本如下。

- MTC（Master Tester Component）首先通过产生 PTC（Parallel Tester Component）对测试系统进行初始化。对于 X-协议产生一个低端 PTC 和一个高端 PTC。
- 通过 IUT，低端 PTC 建立一个与高端 PTC 的一个 X-连接。出于简单考虑，我们假定一个 N 网络连接已经建立，即不会出现一个 X_CONNECTrequest 被拒绝（该假定是为了解释 TTCN 特性）。
- 低端 PTC 发送一个数据包，该数据包将通过 IUT 在高端 PTC 返回，这个数据包将在一个指定的时间间隔内返回，该过程重复多次。
- 在完成数据传递后，低端 PTC 断开，并发送它的最初结果给 MTC 后，计算最终结论并终止测试。

为了对 X-协议进行测试，需要提供以下配置和相关描述。

X-协议一致性测试基于 CTMF，其中 IUT 是一个 X-协议的实现。我们也假设有一个网络服务提供商提供网络服务（N），X-协议测试在此基础进行。因此有以下测试例。

- LT 将用 N-SERVICE 元语和 X-PDU 加以说明。
- 分别用 N_DATArequest 和 CR_PDU 进行说明。
- UT 将用 X-SERVICE 元语加以说明。
- 使用 X_CONNECTrequest。

一般来说，ISO 一致性标准需要对 N–1 层的 ASP、N 层 ASP 和 N 层 PDU 进行说明。为此，TTCN 提供了一个最小的功能集合。

- 提供能够通过测试系统发送和/或接收 ASP 的能力。
- 提供嵌在 ASP 中 PDU 的描述能力。

- 说明 ASP 在 PCO 被发送和/或被接收的次序。

TTCN 采用以下方法提供上述功能。

- 声明 ASP 和 PDU 的类型。
- 声明 PCO。
- 说明实际的 ASP 和 PDU。
- 说明行为实例。

在以后的章节中，我们将深入学习以上内容。

本测试例的使用有以下两种意义：

- IUT 在限定的时间内通过 X-协议，接收并返回指定数量的数据包。
- 每一个测试目标将用不同的测试例表达。

我们将在以后的章节中，逐步建立该协议一致性测试例的测试套，X-协议测试的完整说明将在本章的后面“完整实例”一节中给出。

3.2　测试系统行为描述

为了测试 IUT，我们需要建立一个仿真测试事件集合或交互行动序列。这个用于描述测试任务的事件或行动的序列称为测试例(Test Case)，一个特定协议的测试例集合称为测试套。

TTCN 就是一种用于说明测试例的符号集，它可以建立一个实际被测系统的抽象模型，并说明测试例的执行过程。抽象的测试例包括所有的 IUT 所支持的被测目标，但它不包括测试系统的信息。然而，说 TTCN 是一个符号集，并不意味着 TTCN 本身是抽象的，现在 TTCN 已经逐渐进化成具有可操作的语法和语义的形式化描述语言，用 TTCN 描述协议测试例如同编写程序一样。

在 ISO/IEC 9646-3 中定义了两种 TTCN 的图表：一种是图形符号，另一种是语义符号。图形符号采用表格的形式描述测试例的内容。这种表现形式比较直观，所以称为 TTCN-GR，它适合于测试例的分析与设计。语义符号采用巴氏克范式 Backus-Naur Form(简称 BNF)形式说明 TTCN 测试例，所以它更适合用计算机处理，所以把这种形式的表示称为 TTCN-MP(TTCN-machine processable)。在本教材中主要采用 TTCN-GR 来描述测试用例。

3.2.1　行为树

在我们深入讨论 TTCN 语言之前，首先讨论 TTCN 是如何描述一个测试组件的行为的。在 TTCN 中，许多标准服务定义和协议说明都采用图或表来描述，并在此基础上产生测试例。然而，一致性测试只关心控制与观察点上的交互，所以系统行为用一棵树来表示比较自然，这棵树就称为行为树。在行为树中，每一个树枝表示两个协议状态之间可能发生交互。在 TTCN 中，为了与表格形式一致，把行为树的分支随着时间逐层横向缩排，并写在一个表框内。

在行为树中，处在同一层的节点不一定是姊妹关系，如 F、G 与 I、J 就是如此。

在行为树中，一个节点称为行为行。一个行为行由以下部分构成。

(1)行号。

(2)标签。

(3)声明行。

(4)约束。

(5)结论。

(6)行为行注释。

在上面的构成成员中，哪些被使用可随具体情况而定。例如，行号和注释是必须体现的，而约束和结论仅在需要时使用。

TTCN 行为树如图 3-3 所示。

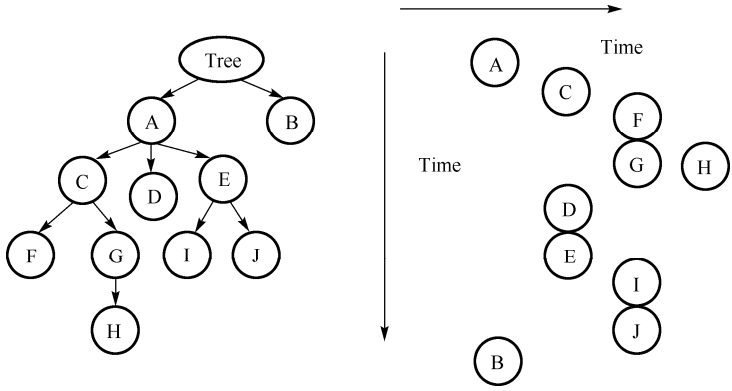


图 3-3　TTCN 行为树

3.2.2　TTCN 行为描述

在 TTCN 中，所有的行为行用动态行为表来说明。有三种类型的行为表，它们的表头和表体有所不同。

- 测试例动态行为表。
- 测试步动态行为表。
- 缺省动态行为表。

虽然它们的表体基本相同，但从表面上看它们有不同的表头，而更实质上的区别是使用方法不同。

在动态行为表（见表 3-1）中，各列分别是行号、标签、声明、约束、结论和注释。淡灰底纹的行表示一个行为行的范围，而深灰底纹的行表示一个声明行。

表 3-1　动态行为表

行号	标签	声明（行为描述）	约束	结论	注释
1		A			
2		C			
3		F			
4		G			
5		H			
6		D			
7		E			
8		I			
9		J			
10		B			

1．声明行与声明

在一个动态行为表中，声明行在“行为”描述栏中定义，它用于说明事件发生顺序。

在声明行中描述测试系统的行为，如发送和接收 ASP 等。一个声明又可分成以下三种不同的类型。

（1）事件。

(2) 行动。

(3) 条件。

1) 事件

声明是否成立，取决于当前所发生的事件。声明成立也称为事件匹配。有两类事件，分别是输入事件(Input Events)和时间事件(Timer Events)。一个输入事件是一个到达指定 PCO 的 ASP，或者是指定 CP(协同点 Coordination Point)的消息。一个时间事件是一个协议时间结束时刻。TTCN 声明中的事件有：

(1) RECEIVE。

(2) OTHERWISE。

(3) TIMEOUT。

2) 行动

有时，声明总是成立的，也就是说它是可执行的，我们把这种声明称为活动。虽然在 ISO/IEC 9646-3 标准中没有行动这个词，但实际上一个测试系统有许多行动被执行。在 TTCN 中假定它们都能够成功地执行。TTCN 中的行动有：

(1) SEND。

(2) IMPLICIT_SEND。

(3) ASSIGNMENT_LIST。

(4) TIMER_OPERATION。

(5) GOTO。

3) 条件

声明行也可以包括条件声明，即布尔表达式。我们把这个声明行称为条件声明行。如果没有事件匹配，也就没有行动被执行，除非声明行中的条件的值为 True。如果一个声明行不包括条件，则为非条件声明行。

一个 TTCN 条件就是布尔表达式：

BOOLEAN_EXPRESSION

4) 事件、行动和条件的结合

事件、行动和条件组合可以通过 TTCN-MP 来定义。在下面的章节中，我们将根据具体内容介绍基于 TTCN-MP 的事件、行为和条件组合的应用。

2．执行和匹配

下面讨论一个行为树如何被执行或遍例。

1) 替换

在同一缩排的声明行的集合中，有相同父节点的节点称为可替换声明行，简称替换。例如，在图 3-3 中(A,B)、(C, D, E)、(F, G)、(I, J)和(H)是所有的替换。

由于一个替换集合中不同替换的前后次序是十分重要的，所以必须把所有事件和条件在一个没有激活的行动之前声明。

2) 行为树的执行

行为树的执行从树根开始。首先第一个替换集合被循环执行，每一个替换均以它们在集合中出现的先后次序被赋值。如果一个替换不成功，则执行下一个替换；如果替换成功，则执行该替换的下一级替换集。在所有的替换都被执行后，该替换集合的循环停止。这时，将得出结论。

图 3-3 是一个行为树执行的例子。首先被执行的是替换集合(A,B)。如果 B 成功，则执行终

止。如果 A 成功，则下一个替换集合(C,D,E)被激活。现在假定 E 是成功的，则下一个替换集合是(I,J)。如果 I 或 J 有一个是成功的，则执行终止。

注意：如果在任意一个替换集合中没有声明是成功的，则执行将“死锁”。

3.3 TTCN 数据类型和取值

本节将讨论 TTCN 的数据类型和它们的取值，TTCN 所包含的数据类型用来说明行为描述中所涉及的数据，如在 ASP 和 PDU 中的数据类型说明等。

TTCN 所定义的数据类型来自于 ASN.1(Abstract Syntax Notation One)定义的数据类型，而且在 TTCN 与 ASN.1 之间无明显的界线，只是构造类型的定义有所不同。在说明测试套时，可以不使用 ASN.1，例如在 ISO 网络模型中的底层协议就没有使用 ASN.1。

TTCN 包括众多的预定义数据类型，也允许用户根据预定义数据类型自己定义构造类型。在定义数据类型时使用下列 TTCN 表。

(1)简单类型定义表。

(2)构造类型定义表(一个类型定义使用一个表)。

3.3.1 预定义数据类型

TTCN 有丰富的预定义数据类型。除 HEXSTRING 之外，TTCN 的预定义数据类型是 ASN.1 的子集，而且两者兼容。ASN.1 中的数据类型都可以在 TTCN 中不加说明地使用。HEXSTRING 类型在 ASN.1 中不存在。TTCN 的数据类型如下。

(1)HEX STRING//十六进制位串，如'0F'。

(2)BOOLEN。

(3)INTEGER。

(4)BIT STRING//位串，如'1001'B。

(5)OCTET STRING//ASN.1 十六进制串，如'0F'O。

(6)Character String//IA5String。

(7)ENUMERATED//枚举类型。

(8)OBJECT IDENTIFIER//对象标识符，由标准化组织定义的整数序列，如 ttcn-standard OBJECT IDENTIFIER::= { iso (1) standard (0) 9646 3 }。

(9)REAL//用科学计数法表示的实数，如 $10*2^{-2}$。

(10)NULL//空。

(11)其他 ASN.1 类型。

3.3.2 取值

TTCN 预定义数据类型的值与 ASN.1 数据类型的值具有相同的取值范围，参见 ASN.1。

3.3.3 简单用户定义类型

TTCN 允许用户不使用 ASN.1 的语法，定义简单的用户自定义数据类型。TTCN 的简单用户定义类型采用简单类型定义表对数据类型进行描述，并可以在测试套中的任何地方使用。简单用户类型的定义，通常是对预定义数据类型或前面已经定义的用户定义类型的进一步说明，这些说明往往是施加一些约束。例如：

- 列表：列表中的数据是字符的枚举。
- 范围：对整数的约束。
- 长度：用于字符串的约束。

注意：TTCN 语法允许在简单类型定义表中使用 ASN.1 数据类型，我们推荐 ASN.1 的数据类型采用专用的表说明 ASN.1 数据。

3.3.4　构造类型

TTCN 有专有的表来定义构造类型数据。构造数据如同简单数据类型一样可以在测试例的任何一个地方使用。但它们主要在 ASP 和 PDU 的子构造中使用。具体细节参见 ASP、PDU 和 CM 取值。

3.4　PCO 和 CP

TTCN 支持异步通信模型，在测试组件和被测软件之间的通信是通过控制和观察点(PCO)实现的，而不同测试组件之间是通过测试协作点(Coordination Point，CP)进行交互。下面给出一个具有普遍意义的通信模型。

3.4.1　通信模型

在通信模型中，我们将使用队列模型描述 PCO 和 CP：

每一个 PCO/CP 有两个先进先出的队列，与程序中的队列不同，这里的队列没有边界。

- 一个队列用于存放发送(SEND)ASP 的对列。
- 另一个用于存放接收(RECEIVE)ASP 信息。
- 两个队列在一个 PCO 或 CP 进行连接。
- 一个用于 PCO/CP 的输入，另一个用于 PCO/CP 的输出。

3.4.2　发送一个 ASP

发送一个 ASP，是通过 SEND 行为把一个 ASP 追加到 PCO 队列中去。PCO 的 SEND 队列是无约束长度的队列，所以它总可以接收来自 LT 或 UT 的消息。

3.4.3　接收一个 ASP

一个有效的接收从 RECEIVE 队列中取出 ASP，一般接收有以下两个步骤。

- 接收 ASP。
- 检查其内容。

3.4.4　声明 PCO 类型

测试套中使用的 PCO 类型必须在 PCO 类型声明中进行。每一个 PCO 类型声明需要下列信息：

- PCO 类型名。
- 与 PCO 通信的对象 LT 或 UT。

PCO 类型 N_SAP 和 X_SAP 声明如表 3-2 所示。

表 3-2　PCO 类型 N_SAP 和 X_SAP 声明

Pco-name	Pco-type	Role	Comments
L_1	N-ASP	LT	N serve access point at lower tester

续表

Pco-name	Pco-type	Role	Comments
L_2	N-ASP	LT	
U_1	X-ASP	UT	X serve access point at upper tester
U_2	X-ASP	UT	
Detail Comments:			

3.4.5 使用 PCO 和 CP

如果测试套仅使用 PCO，则 PCO 在 TTCN 声明中可以省略。如果有多个 PCO 和 CP 被使用，则 PCO 和 CP 必须在 TTCN 声明中加以说明。

3.4.6 PCO 和 CP 快照

我们已经描述了一个行为树的执行过程，我们知道行为树循环执行替换集中的替换，直到一个声明行成功。在每一次循环的最初，从输入队列中取出 PCO 或 CP 的当前值的一个快照。每一个声明的值取决于这个快照，而不是 PCO 或 CP 队列的实际值。这样替换被执行的时间将会冻结，即阻止两个快照之间的事件发生。通过这种方法，一个 ASP、PDU 或 CM 的到达只有在快照更新后才能被登记。

3.4.7 声明 CP

所有在测试套中使用的 CPs 均应该在 CP 声明表中被声明。每一个 CP 需要下列信息：

- CP 名。
- CP 角色。

两个测试组件通过 CP 进行通信。

协同点 CP1 和 CP2 的声明表如表 3-3 所示。

表 3-3 协同点 CP1 和 CP2 的声明表

Cp name	Comments
Cp_1	Coordination between the MTC and PTC of the lower tester
Cp_2	
Detail Comments:	

3.5 发送语句

TTCN 行为树中的主要行为之一是通过 PCO/CP 向 IUT 发送 ASP 或 PDU。下面讨论发送(SEND)行为的描述和使用。

3.5.1 发送 ASP

发送行为声明允许一个测试套说明一个通过 PCO 发送的 ASP 类型。发送语句格式如下：

```
PCO_Identifier ! ASP_Identifier
```

SEND 语句被激发后，接下来是一个 ASSIGNMENT_LIST 和/或 TIMER_OPERATION 语句。它们的先后次序是固定的，例如：

```
SEND[3] [QUALIFIER][1] [ASSIGNMENT_LIST][2] [TIMER_OPERATION][4]
```

方括号中的内容是可选择的。

3.5.2　执行发送语句

语句的行号具有时间性质，它体现了语句执行的先后次序。通常一个发送语句的执行次序为：如果一个条件 QUALIFIER 是 FALSE，则过程停止，并且发送语句为假；如果 QUALIFIER 是 TRUE，则 ASSIGNMENT_LIST 被执行。最后，TIMER_OPERATION 被执行。

例如：

```
L! N_DATArequest
```

发送网络数据请求服务原语给名为 L 的 PCO：

```
L! N_DATArequest [B=1]
```

如果 B=1，则执行 SEND。

```
L! N_DATArequest [B=1] (X:=3)
```

如果 B=1，则把 X 赋值为 3，并且执行 SEND。

3.5.3　发送一个 PDU

通常一个 PDU 数据被包含在 ASP 中，在声明 SEND 语句时并不显式说明。然而，并不是所有的协议有服务定义(如 X.25)，所以 TTCN 提供了显式声明 PDU 的语句格式，而不用 ASP。一个发送 PDU 的语句如下：

```
PCO_Identifier ! PDU_Identifier
```

该语句中还可以包含其他辅助信息，这些信息与 PDC 数据一同被处理，它们的说明格式与标准 SEND 语句相同。

3.5.4　发送协同信息

发送语句也可以在 CP 上发送一条协同信息(Coordination Message，CM)。格式如下：

```
CP_Identifier ! CM_Identifier
```

该语句中还可以包含其他辅助信息，这些信息与 CM 数据一同被处理，它们的说明格式与标准 SEND 语句相同。

3.6　接收语句

接收语句是用于描述一个测试组件从测试系统 IUT 接收消息事件的基本语句。下面介绍接收语句的语法和使用。

3.6.1　接收 ASP

接收语句允许一个测试套说明一个 PCO 所接收的 ASP 类型，接收语句的语法如下：

```
PCO_Identifier ? ASP_Identifier
```

接收语句也可以是有条件的，在语句中也可以有 ASSIGNMENT_LIST 和/或 TIMER_OPERATION 短语。这些短语在接收语句中的出现次序是有一定次序的，而方括号中的内容是可选的。语法如下：

```
RECEIVE[1] [QUALIFIER][2] (ASSIGNMENT_LIST)[3] [TIMER_OPERATION][4]
```

3.6.2 执行接收语句

语句的行号具有时间性质，它体现了语句执行的先后次序。通常一个接收语句的执行次序为：如果一个条件QUALIFIER是FALSE，则过程停止，并且发送语句为假；如果QUALIFIER是TRUE，则 ASSIGNMENT_LIST 被执行。最后，TIMER_OPERATION 被执行。

例如：

```
L? N_DATArequest
```

一个网络数据请求服务原语在名为 L 的 PCO 匹配：

```
L? N_DATArequest [B=1]
```

一个网络数据请求服务原语在名为 L 的 PCO 匹配，并且 B=1：

```
L! N_DATArequest [B=1] (X:=3)
```

一个网络数据请求服务原语在名为 L 的 PCO 匹配，并且 B=1 时，ASSIGNMENT_LIST 被执行。

3.6.3 接收 PDU

通常一个 PDU 数据被包含在 ASP 中，在声明 RECEIVE 语句时并不显式说明。然而，并不是所有的协议有服务定义(如 X.25)，所以 TTCN 提供了显式声明 PDU 的语句格式，而不用 ASP。一个接收 PDU 的语句如下：

```
PCO_Identifier ? PDU_Identifier
```

该语句中还可以包含其他辅助信息，这些信息与 PDC 数据一同被处理，它们的说明格式与标准 RECEIVE 语句相同。

```
PCO_Identifier ? PDU_Identifier
```

3.6.4 接收协同信息

接收语句也可以在 CP 上发送一条协同信息(Coordination Message，CM)。语句格式如下：

```
CP_Identifier ?CM_Identifier
```

该语句中还可以包含其他辅助信息，这些信息与 CM 数据一同被处理，它们的说明格式与标准 RECEIVE 语句相同。

3.6.5 OTHERWISE 语句

OTHERWISE 语句是描述一个测试组件在 PCO 接收任何类型 ASP 或 PDU 的接收语句。该语句强调任意 ASP 或 PDU 的类型，而并不是标准的类型。这是因为有时 IUT 并不是正常工作，所以对这种情况的处理都归结为 OTHERWISE 语句处理。下面介绍 OTHERWISE 语句的语法。

```
PCO_Identifier ? OTHERWISE
```

需要注意的是，OTHERWISE 语句不能在 CP 上使用。

在实际使用过程中，OTHERWISE 常作为一个可替换的接收事件。

3.7　定义 ASP、PDU 和 CM 类型

ASP 是分布式系统中的第 N 层与第 N–1 层服务的标准化定义，所以要测试这种服务的一致性，首先要在测试套中定义相应的 ASP。

PDU 是网络层次结构中，层与层之间交互的格式说明，它与相关的协议说明有关。所以在测试套中也应该定义相应的 PDU。从一致性测试的角度，PDU 的定义应考虑非常规的 PDU 数据格式，而不是标准的交互格式。

另一个需要定义的是协同消息。TTCN 中提供了上述三种数据的定义表格，也可以使用 ASN.1 形式定义。

3.7.1　TTCN 复合类型

TTCN 提供了声明下列复合类型的表格：

- ASP 类型定义。
- PDU 类型定义。
- Structured 类型定义。
- CM 类型定义。

使用上述复合类型，我们可以定义任何复合的 ASP 和 PDU 类型。实质上，TTCN 中的 ASP、PDU 或 structured 类型没有本质上的区别。

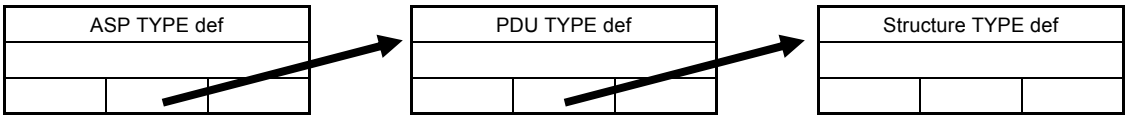
- 一个 ASP 可以有参数，而参数可以是任何非 ASP 类型。
- 一个 PDU 可以有不同的域，而域的类型可以是任何非 ASP 类型。
- 一个 structured 类型有构成元素，而构成元素的类型可以是任何非 ASP 类型。

3.7.2　类型链 Chaining

通常，ASP 参数、PDU 域和 structure 元素是预定义类型或称简单类型(包含使用 ASN.1 中类型)。然而，就像我们在前面提到的那样，参数、域和元素又可以是 PDU 或 structure 类型，从而构成一个类型链。

类型链如表 3-4 所示。

表 3-4　类型链



3.7.3　ASN.1 复合类型

在 ASN.1 中，构造器 SEQUENCE 和 SET 可以用来构造任意的复合类型。ASN.1 复合类型可以使用下面的表格来定义：

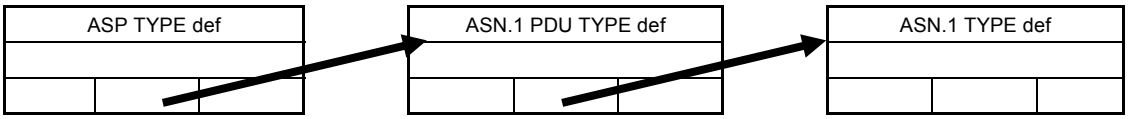
- ASN.1 类型定义。
- ASN.1 ASP 类型定义。
- ASN.1 PDU 类型定义。
- ASN.1 CM 类型定义。

在 TTCN 中，用表格的形式定义数据类型时完全可以使用 ASN.1 来定义子类型，但反之亦然。

这两种格式可以相互结合。例如在 TTCN 中定义 ASP 时使用 ASN.1 定义 PDU 类型。

使用 ASN.1 的 ASP 和 PDU 结构如表 3-5 所示。

表 3-5 使用 ASN.1 的 ASP 和 PDU 结构



3.7.4 局部类型定义

在 ASN.1 表中，至少有一个类型定义存在，这就是主类型定义，它在表的头部定义。然而，ASN.1 表也可以包括一些局部定义，这些称为局部定义的类型只在该表的内部有效，也就是说，ASN.1 表中包括主定义和局部定义。

局部定义用 typereference::=开始，并且它不能作为主定义在表头部使用。

3.7.5 通过引用定义类型

为了省略重复出现的 PDU 定义和已经在其他表中定义过的类型，TTCN 允许使用引用定义，而不是把每一个数据类型重复定义。可以使用引用定义的类型有：

- ASN.1 ASP 定义。
- ASN.1 PDU 定义。
- ASN.1 Type 定义。

对所有的引用可以使用一个表加以说明。引用表为：

- ASN.1 ASP 引用定义。
- ASN.1 PDU 引用定义。
- ASN.1 Type 引用定义。

注意：按照 ASN.1 的要求，Type Reference 列和 Module Identifier 列应该简明扼要。另外，module identifier 后可以使用可选的对象标识。

使用引用来定义 ASN.1 PDU 类型如表 3-6 所示。

表 3-6 使用引用来定义 ASN.1 PDU 类型

PDU Type Definition by reference						
PDU name	PCO type	Type reference	Module identifier	Enc rule	Enc variation	Comment
CR-PDU	N-SAP	CR-PDU	pdu 1.0 123.4			连接请求
CC-PDU	N-SAP	CC-PDU	pdus 1.0 1234			连接确认
Detail Comments:						

3.7.6 定义 ASP

OSI 中，服务原语的定义通常采用元组形式，例如，一个原语后接一个参数列表。每一个参数使用自然语言来描述控制信息或用户数据。一些参数是不可缺省的，而有些是可选择的或在一些情况下可以省略。

在 TTCN 中，服务原语被称为抽象服务原语(Abstract Service Primitives，ASP)，并且在 ASP 类型定义表中定义。

ASP 类型定义如表 3-7 所示。

表 3-7　ASP 类型定义

ASP Type Definition		
ASP Name: N-DATArequest PCOTYPE: N-SAP Comments: This is the type definition of the N-DATArequest ASP. It has a single parameter used to carry user data		
Parameter Name	Parameter Type	Comments
User-data	PDU	
Detailed Comments:		

内嵌(metatype)PDU 类型

在上面的例子中使用了 PDU 内嵌类型。也就是说，在 ASP 的定义中可以使用任意的 PDU 类型。

3.7.7　定义 PDU

在大多数的 OSI 标准文本中，PDU 定义一般采用以下两种形式。

- 简单的表格，在表格中配以文字解释。
- 使用 ASN.1 格式，并使用非规范的文字解释。

第一种形式的定义并不是十分严谨，表现在 PDU 域的类型和 PDU 子结构的元素类型没有显示定义。在使用 TTCN 定义一个测试套时，必须突出上述内容。

例如，一个协议标准可能描述一个特殊的 8-bit 长度的域，这隐含说明域的类型为 BITSTRING。如果一个实际的域中没有这个 8-bit 长度的域，则就会与协议不一致，这也是测试的目标所在。在这种情况下，可以把该域的类型定义为 OCTETSTRING 类型。

在 ASN.1 中，类型、PDU 结构和它们的域的定义通常是完备的。它们可以直接从其他标准中引用过来或从其他引用中复制而来。

PDU 类型定义如表 3-8 所示。

表 3-8　PDU 类型定义

PDU Type Definition			
PDU Name: CR-PDU PCO Type: N-SAP Encoding Rule Name: Encoding Variation: Comments: This is the type definition of the CR-PDU			
Filed Name	Filed Type	Filed Encoding	Comments
Type	OCTESTRING[1]		
Dst-ref	BITSTRING[4]		
Src-ref	BITSTRING[4]		
Variable-part	VARIABLE-PART		Reference to structured type
User-data	IA5STRING[0…32]		
Detail Comments:			

ASN.1 PDU 类型定义如表 3-9 所示。

表 3-9　ASN.1 PDU 类型定义

ASN.1 PDU Type Definition
PDU Name: CR-PDU PCO Type: N-SAP Encoding Rule Name: Encoding Variation: Comments: This is the type definition of the CR-PDU

续表

Type definition
SEQUENCE {Type OCTESTRING（SIZE（1..1）） Dst-ref BITSTRING（SIZE（4..4）） Src-ref BITSTRING（SIZE（4..4）） Variable-part VARIABLE-PART-- Reference to structured type User-data IA5STRING（SIZE（0…32））OPTIONAL }
Detail Comments:

构造类型定义如表 3-10 所示。

表 3-10 构造类型定义

Structure Type Definition			
Type Name:Varable-Part Encoding Variable: Comments: This is the type definition of the variable part of the CR-PDU and the CC-PDU			
Element Name	Type Definition		Comments
ParamA-id	BITSTRING[2]		Parameter identifier
ParamA	OCTESTRING[2..4]		Optional parameterA
ParamB-id	BITSTRING[2]		Parameter identifier
ParamB	BOOLEN		Optional parameterB
Detail Comments:			

3.7.8 构造 ASP 和 PDU 的子集

TTCN 的构造类型（structured types 有时简称构造 structures）仅用在对 ASP、PDU、CM 以及其他构造类型的子集的构造，所构造的子集称为子构造类型（substructure）。在 ASN.1 中，不仅在定义表中可以使用子构造定义 ASP 和 PDU 的子集，也可以在测试套中使用子构造来定义一般的类型。

ASN.1 构造类型定义如表 3-11 所示。

表 3-11 ASN.1 构造类型定义

ASN.1 Structure Type Definition
Type Name: Varable-part Encoding Varable: Comments: This is the type definition of the variable part of the CR-PDU and the CC-PDU
Type definition
SEQUENCE {ParamA-id BITSTRING（SIZE（2..2））Optional ParamA OCTESTRING（SIZE（2..4））Optional ParamB-id BITSTRING（SIZE（2..2））Optional ParamB BOOLEN Optional }
Detail Comments:

3.7.9 定义 CM 类型

CM 在测试套定义中比较特殊， 它可以通过测试套说明器描述，可以用表格形式，也可以使用 ASN.1 形式定义。

CM 类型定义如表 3-12 所示。

表 3-12　CM 类型定义

CM Type Definition		
CM Name: ptc-RESULT Comments: Coordination message to transfer preliminary result from the lower		
Parameter Name	Parameter Type	Comments
Result	RESULT-TYPE	User definition type
Detail Comments:		

ASN.1 CM 类型定义如表 3-13 所示。

表 3-13　ASN.1 CM 类型定义

ASN.1 CM Type Definition
CM Name: ptc-RESULT Comments: Coordination message to transfer preliminary result from the lower
SEQUENCE {result RESULT-TYPE-user definition type}
Detail Comments

3.7.10　在行为树中使用 ASP 和 PDU

下面给出 ASP 和 PDU 在网络连接中的使用实例，以便在学习 ASP 和 PDU 定义之后，对它们的使用有进一步的了解。更深入的学习将在以后的章节中展开。

例 3.1　一个简单网络连接过程描述。一个网络连接的一致性测试模型如图 3-4 所示。测试步骤为：

(1) 测试组件在低端经过控制观察点 L 向 IUT 发出一个连接请求 N-DATA request。

(2) 在高端，测试组件 UT 通过控制观察点，接收到一个连接指示 X-Connect indication 或其他信息。

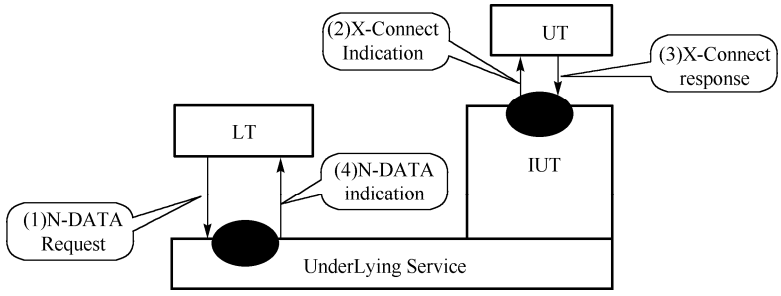


图 3-4　ASP 和 PDU 使用实例

(3) UT 发出建立连接的应答 X-Connect response。

(4) LT 接收一个连接指示 N-DATA indication 或其他信息。

上述描述对应的行为树可以用表 3-14 表示。

表 3-14　一个简单网路连接过程描述的 TTCN 行为树

序号	标签	行为描述	约束	结论	注释
1		L!N-DATA			
2		L?N-DATAindication			
3		L?OTHERWISE			

序号	标签	行为描述	约束	结论	注释
1		U?X-Connect indication			
2		U!X-Connect response			
3		U?OTHERWISE			

3.8 TTCN 表达式

在 TTCN 中，一个表达式可以由值(值是最简单的表达式)和运算符按一定的运算律构成。TTCN 的表达式有：

- 文本。
- 变量和常量标识符。
- 形式化参数标识符。
- ASP 参数。
- PDU 或 CM 域。
- 构造元素。
- 预定义的和用户定义的操作符。
- 由上述表达式经过组合而成是 TTCN 表达式。

在 TTCN 表达式中，变量的使用取决于使用表达式的上下文。有关变量的使用将在其他章节中讨论。

3.8.1 TTCN 运算符

TTCN 表达式中支持下列运算符：

- 算术运算符。
- 关系运算符。
- 逻辑运算符。

算术运算

TTCN 仅支持可以对整数或整数的导出类型运算的运算符，这些运算符有：

+、-、*、 /和 MOD

由算术表达式构成的表达式称为算术表达式。在 TTCN 中的算术表达式也可以写成普通的算术表达式的形式，如 3*(Z+9)。

相等运算

相等“=”和不等“<>”运算符可以用于任何一种数据类型。

由等号和不等号构成的表达式的值为 BOOLEAN 类型。

例如：

B_string = '01'B

H_string <> 'FF'H

其他关系运算

TTCN 支持可以对整数或整数的导出类型运算的关系运算符，它们是<、>、>=和<=，例如 X <= 3*Y。与等号运算符的运算结构类型相同，上述关系表达式的结果也为 BOOLEAN 类型。

逻辑运算

TTCN 支持对布尔类型或布尔的导出类型运算的逻辑运算符，它们是 AND、OR 和 NOT。例如，逻辑运算表达式 A AND NOT (B OR C)，其中 A、B 和 C 均为布尔变量。

条件

条件是包含在括号内的表达式：

```
[ expression ]
```

这个表达式的值为布尔值。例如，一个包含条件的表达式如下：

```
[ X < 6 AND H_string <> 'FF'H ]
```

赋值运算

一个 TTCN 语句可以是一个 ASSIGNMENT_LIST，即一个赋值运算列表，表中的赋值运算用省略号分开，赋值运算被放到一个小括弧内。

```
( assignment1, . . . ., assignmentn )
```

括弧内左边必须是一个变量的解，在 SEND 和 RECEIVE 语句中，它可以是一个 ASP 参数引用，一个 PDU 域引用或一个结构元素的引用。而赋值运算右边必须是一个与左边类型匹配的 TTCN 表达式。

例如：

```
(X:= 3, A:= "a string", Y:= 3*(Z+9), H:= 'FF'H)
```

3.8.2　TTCN 函数

TTCN 支持众多的预定义的函数，也提供了用户定义函数的机制。函数可以作为运算在一个表达式中出现。

预定义函数

TTCN 支持众多的预定义函数，而且还在不断地被扩展。现在，TTCN 中包含的函数有：

- HEX_TO_INT (data_object_reference)

HEXSTRING 到 INTEGER 的转换函数。

- BIT_TO_INT (data_object_reference)

位串到整数转换的函数。

- INT_TO_HEX (data_object_reference)

INTEGER 到 HEXSTRING 的转换函数。

- INT_TO_BIT (data_object_reference)

INTEGER 到 BITSTRING 的转换函数。

- LENGTH_OF (data_object_reference)

返回串类型数据对象的长度。

- NUMBER_OF_ELEMENTS (data_object_reference)

返回 SEQUENCE 或 SET 类型的元素数量。

- IS_PRESENT (data_object_reference)

如果 OPTIONAL 或 DEFAULT 数据对象在接收端的 PDU 出现，则为真；否则为假。

- IS_CHOSEN（data_object_reference）

该函数用于从 CHOICE 中接收一个指定元素。如果一个数据对象（如 PDU 域）与接收值匹配，则返回真。

用户定义函数

TTCN 允许用户定义自己的函数。一个可行的方式是，使用程序设计语言来定义一个函数。与预定义的函数一样，用户定义的函数被用于行为树中作为对变量的处理。

TTCN 中的函数定义使用测试套的函数表。如果一个运算没有声明参数，则参数默认为空，例如：DATE（）。

测试操作定义如表 3-15 所示。

表 3-15 测试操作定义

Test Suit Operation Definition
Operation Name: INC（integer） Result Type: integer Comments:
Description

Int INC（i）
Int temp
{
return（temp+1）:/*return the incremented value of i note that i self not change
}

Detail Comments:

3.9 说明 ASP、PDU 和 CM 值

在前面的章节中已经介绍了 ASP、PDU 和 CM 类型的定义。在实际测试过程中，当发送或接收一个数据时，这些数据类型定义还应指定一个具体的值，我们把这些值称为变量的约束。

ASP、PDU 和 CM 约束定义由专用的表格来定义，一个 ASP、PDU 和 CM 约束定义至少应该与一个 ASP、PDU 和 CM 类型定义相对应。

TTCN 的约束定义表有：

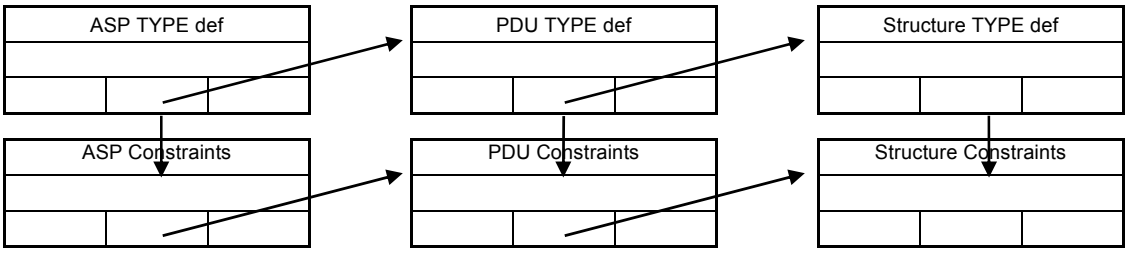
- ASP 约束定义。
- PDU 约束定义。
- structured 类型约束声明。
- CM 约束声明。

3.9.1 Static 和 Dynamic 链

可以把 ASP、PDU 和 structured 类型连接起来，构造复合 ASP 和 PDU 类型数据。这种连接有两种形式。(1)静态连接(Static chaining)：这种连接是把 PDU 约束或 structure 约束作为 ASP 的参数、PDU 域或 structure 的元素。(2)动态连接(Dynamic chaining)与静态连接不同，它只发生在实际约束作为参数传递给约束引用时。

构造 ASP 和 PDU 类型和它们的约束的关系如表 3-16 所示。

表 3-16　构造 ASP 和 PDU 类型和它们的约束的关系



3.9.2　复合 ASN.1 值

无论是静态链，还是动态链，它们都是由 ASN.1 数据类型构成的，虽然在链表中没有使用相关的术语，但其本质仍然是使用类型的引用构造链表所定义的数据。如果一个引用采用链的形式，那么就有相应的约束。

3.9.3　ASP 约束

通常，每一个 ASP 类型定义至少有一个 ASP 约束的定义。然而，在一些服务原语中定义的服务没有参数。在这种情况下，约束是没有必要的，对 CM 也是如此。然而，对于 PDUs 就不是这样的情况，一个没有域的 PDU 是没有意义的。

ASP 约束与 PDU 约束十分相似，有关内容将在下面的章节中介绍。应用于 PDU 约束的规则同样可以应用于 ASP 约束。

3.9.4　PDU 的约束

通常，在 PDU 域的类型定义中，总有一个相应的域约束与其对应，而且它们的类型事件是一样的。我们将在以后的章节中看到 PDU 域的缺省或替换的情况，以及如何使用导出路径和约束值的匹配。

使用 TTCN 表定义 X_PDU 约束如表 3-17 所示。

表 3-17　使用 TTCN 表定义 X_PDU 约束

PDU Constraint Declaration			
Constraint Name:CR1 PDU Type: CR-PDU Derivation Path: Encoding Rule Name: Encoding Variation: Comments A Constraint on the CR-PDU			
Field Name	Field value	Field Encoding	Comments
Type	'F1'o		
Dst-ref	'0001'B		
Src-ref	'0001'B		
Variable-part	Variable part-CR1		
User-data	'Hello'		
Detail Comments:			

使用 ASN.1 定义 X_PDU 约束如表 3-18 所示。

表 3-18 使用 ASN.1 定义 X_PDU 约束

PDU Constraint Declaration
Constraint Name:CR1 PDU Type: CR-PDU Derivation Path: Encoding Rule Name: Encoding Variation: Comments A Constraint on the CR-PDU using ASN.1 value notation
Constraint Value
{Type 'F1'o Dst-ref '0001'B Src-ref 0001'B Variable-part Variable part-CR1 User-data 'Hello' }
Detail Comments:

3.9.5 构造类型的约束

与 ASP 和 PDU 中类型定义和约束定义一样，构造类型的约束定义与 ASN.1 下的约束定义采用相同的方法。

当使用 TTCN 表定义构造类型约束时，所采用的格式与类型定义采用的格式是一致的。也就是说，当一个 PDU 的域使用了一个构造类型定义，则就应该有一个 PDU 的约束定义，同时也应该有一个构造类型约束的定义。

而在 ASN.1 中，构造类型定义与约束定义必须是一致的，但并不是要求一定要一一对应。

构造类型约束声明如表 3-19 所示。

表 3-19 构造类型约束声明

Structure Constraint Declaration			
Constraint Name: Variable part-CR1 Structure Type: VARIABLE-PART Derivation Path: Encoding Rule Name: Encoding Variation: Comments A Constraint on structure type VARIABLE-PART for the CR-PDU			
Element Name	Element value	Element Encoding	Comments
ParamA-id			
ParamA			
ParamB-id	'01'B		
ParamB	True		
Detail Comments:			

ASN.1 中构造类型约束声明如表 3-20 所示。

表 3-20 ASN.1 中构造类型约束声明

Structure Constraint Declaration in ASN.1
Constraint Name: Variable part-CR1 Structure Type: VARIABLE-PART Derivation Path: Encoding Rule Name: Encoding Variation: Comments: how the first tow parameters are omitted in ASN.1

续表

Constraint Value
{ParamB-id　‘01’B 　ParamB　ture }
Detail Comments:

3.9.6　CM 约束

协同信息的约束与 PDU 约束的定义相似。

CM 约束定义如表 3-21 所示。

表 3-21　CM 约束定义

CM Constraint Declaration		
Constraint Name: PTC_RES{actual result: RESULT_TYPE}		
Derivation Path: PTC_RESULT		
Comments：A constraint on the PTC_RESULT coordination message		
Parameter Name	Parameter value	Comments
result	Actual result	Actual result is passed as to parameter to constraint
Detail Comments:		

ASN.1 CM 约束定义如表 3-22 所示。

表 3-22　ASN.1 CM 约束定义

ASN.1 CM Constraint Declaration
Constraint Name: PTC_RES{actual result: RESULT_TYPE}
Derivation Path: PTC_RESULT
Comments：A constraint on the PTC_RESULT coordination message
Constraint Value
Result Actual result---Actual result is passed as to parameter to constraint
Detail Comments:

3.10　约束引用

TTCN 的 SEND 和 RECEIVE 语句明确规定，只有 ASP 或 PDU 类型才能被发送和接收。在动态的行为表中，约束被用来明确定义什么样的 ASP 或 PDU 值被发送或被接收。也就是说，SEND 或 RECEIVE 语句必须有一个约束引用。

在行为行中使用引用如表 3-23 所示。

表 3-23　在行为行中使用引用

序号	标签	行为描述	约束	结论	注释
1		L!N-DATA	NDr		
2		L?N-DATAindication	NDi		
3		L?OTHERWISE			

序号	标签	行为描述	约束	结论	注释
1		U?X-Connect indication	CONind		
2		U!X-Connect response	CONres		
3		U?OTHERWISE			

3.10.1 参数化的约束

约束可以被参数化。具体的方法是在一个约束后面接一个可选择的参数列表。这些形式化的参数可以作为约束值在约束列定义。

参数化的约束如表 3-24 所示。

表 3-24 参数化的约束

PDU Constraint Declaration			
Constraint Name:DT1{actual-data:iA5string} PDU Type: DT-PDU Derivation Path: Encoding Rule Name: Encoding Variation: Comments This is the type definition of the DT-PDU			
Filed Name	Filed value	Field Encoding	Comments
Type	'F3'O		
User-data	'actual-data'		The actual data passed as a parameter to the constraint
Detail Comments:			

当行为描述中的约束被调用时，实参将替换约束。

参数化的调用如表 3-25 所示。

表 3-25 参数化的调用

序号	标签	行为描述	约束	结论	注释
1 2 3		… L?DT-PDU …	… DT1（“A String”） …		

实参必须是一个指定的值。在发送语句中，约束最后被编码并且被发送。而在接收语句中，实参同样也是一个具体的值，该值与所接收的值匹配。在接收语句中，所接收到的值不一定与实参相等。如果希望它们相等，则必须在行为描述中用赋值语句明确定义。

动态链

通常，ASP、PDU 和 structures 类型的参数化约束是动态的，而不是静态的。当一个约束引用的参数用实际约束替换时，它们之间就构成一个动态的链。

ASP 中参数化的 PDU 动态链如表 3-26 所示。

表 3-26 ASP 中参数化的 PDU 动态链

序号	标签	行为描述	约束	结论	注释
1 2 3		… L?N_DATArequest …	… DDr（DT1）		

一个 N_DATArequest 用于传递 DT_PDU。

3.10.2 发送和接收约束

发送一个约束与接收一个约束的规则有些不同，下面将全面比较两者之间的不同。

约束和发送语句

一个发送语句中指明的约束是将要通过网络发送的数据(这里我们暂时不考虑编码的情况)。在 TTCN 中，被发送的对象被称为发送对象(Send Object)，它由相关的约束信息构成。注意，赋值语句可以使变量给发送对象赋予新值，这就是在 ASSIGNMENT_LIST 之前是 BUILD 短语的原因。

```
SEND[3] BUILD[2] [QUALIFIER][1] [ASSIGNMENT_LIST][3] [TIMER_OPERATION][4]
```

约束值和发送

在 SEND 语句中，我们将使用被接收的约束值(Received Constraint Value)表示测试组件希望发送的值，该值是一个 ASP 参数值、ASP、PDU 或 CM 约束中 PDU 或 CM 域。它们都已经在相关的 ASP、PDU 或 CM 定义中声明过。

发送对象的约束值在发送时刻都必须说明。

约束和 RECEIVE 语句

接收一个 ASP、PDU 或 CM 比接收一个 ASP、PDU 或 CM 类型更复杂。这是因为要接收一个 ASP、PDU 或 CM 都要进行正确性检验。这项工作应该通过在 TTCN 中说明必须匹配的约束值来加以确认。一个 RECEIVE 事件仅当所有的条件被满足时，才认为成功。为此，一个扩展的 RECEIVE 语句为：

```
RECEIVE[1] MATCH[2] [QUALIFIER][3] [ASSIGNMENT_LIST][4] [TIMER_OPERATION][5]
```

接收对象

在 TTCN 中使用接收对象(Received Object)一词表示 PCO 或 CP 队列顶端的 ASP、PDU 或 CM 接收值，并且在 RECEIVE 语句执行时被检验。

执行一个对 SEND 约束的替换如图 3-5 所示。

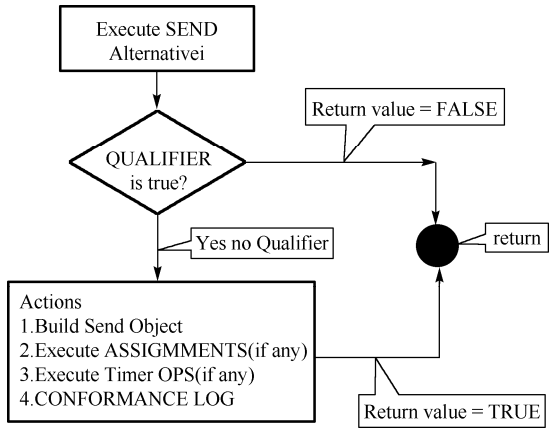


图 3-5　执行一个对 SEND 约束的替换

约束值和接收

在 RECEIVE 语句中，我们将使用被接收的约束值(Received Constraint Value)表示测试组件希望接收的值，该值是一个 ASP 参数值、ASP、PDU 或 CM 约束中 PDU 或 CM 域。它们都已经在相关的 ASP、PDU 或 CM 定义中声明过。

接收值

我们将使用接收值(Received Value)表示一个接收对象元素的值。一个接收值是一个与相关类型定义相一致的值，如 ASP、PDU 或 CM 定义等。

3.10.3　约束与 OTHERWISE 语句

在 OTHERWISE 语句中不能使用约束。切记 OTHERWISE 语句总是匹配非空的 PCO 输入队列，并不进行任何正确性检测。

执行一个有约束 RECEIVE 的替换如图 3-6 所示。执行一个约束的 THERWISE 的替换如图 3-7 所示。

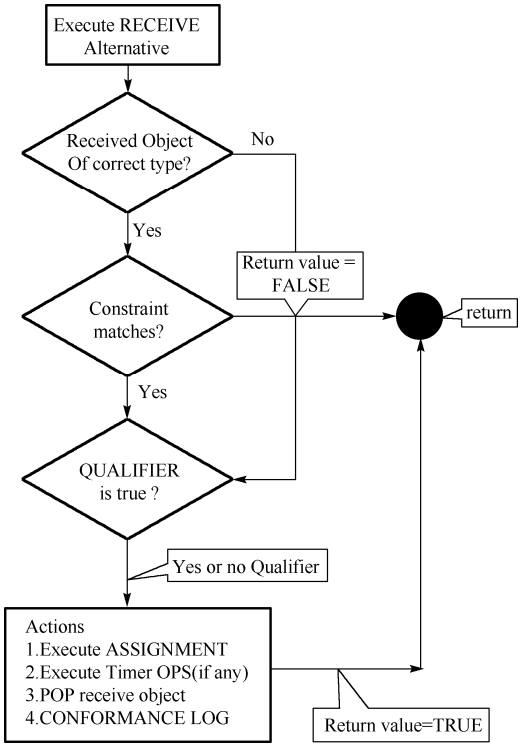


图 3-6 执行一个有约束 RECEIVE 的替换

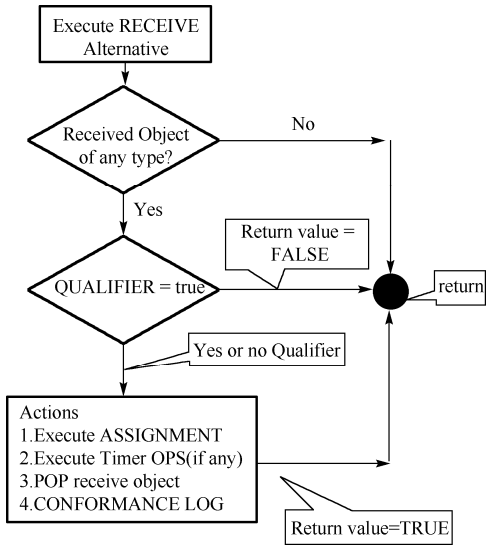


图 3-7 执行一个约束的 THERWISE 的替换

3.11 接收约束值匹配

在这一节中将仔细分析 RECEIVE 语句的执行过程，并分析如何检测一个接收值与指定的约束值匹配等问题。

3.11.1 指定值(Specific Value)

在大多数情况下，一个约束值就是一个指定值，该值并不一定是一个文字(literal)值。在 TTCN 中，一个指定值是一个与 ASP、PDU 或 CM 定义中的类型兼容的表达式的值。TTCN 允许在表达式中使用的运算符如下：

- 文字。
- 变量和常量标识符。
- 形式化参数标识符。
- 预定义的和用户定义的操作符。
- 由上述表达式经过组合而成的 TTCN 表达式。

当一个指定值作为约束值时，一个成功的匹配意味着所接收的值与约束表达式的值完全相等。指定值也可以用于说明任何类型的约束值。

注意：这里讨论的接收值与约束值匹配，并不意味着所有的约束值匹配。

省略值(Omitting Values)

在许多情况下，需要对 ASP 参数或 PDU 域省略。在 TTCN 表中，所有的参数或域都是可选择的，或者是可省略的。省略的方法是使用一个“-”符号来代替实际值。

构造类型约束声明如表 3-27 所示。

表 3-27　构造类型约束声明

Structure Constraint Declaration			
Constraint Name: Variable part-CR1 Structure Type: VARIABLE-PART Derivation Path: Encoding Rule Name: Encoding Variation: Comments A Constraint on structure type VARIABLE-PART for the CR-PDU			
Element Name	Element value	Element Encoding	Comments
ParamA-id	-		Omit the field
ParamA	-		Omit the field
ParamB-id	'01'B		
ParamB	True		
Detail Comments:			

在 ASN.1 中，仅有可选择的或缺省的参数或域的定义才能被省略。可以通过直接使用 OMIT 关键字表示省略，也可以在参数或域的约束中不包括需要省略的值。

替换值（Replacing Values）

在 ASN.1 中，约束可以根据前面定义的约束值来构造，方法是使用关键字 REPLACE。

ASN.1 中构造类型约束声明如表 3-28 所示。

表 3-28　ASN.1 中构造类型约束声明

Structure Constraint Declaration in ASN.1
Constraint Name: Variable part-CR1 Structure Type: VARIABLE-PART Derivation Path: Encoding Rule Name: Encoding Variation: Comments: how the first tow parameters are omitted in ASN.1
Constraint Value
{ParamB-id　'01'B 　ParamB　true }
Detail Comments: 　　{ParamA-id　Omit the field 　　ParamA　　Omit the field 　　ParamB-id　'01'B 　　ParamB　　True 　　}

表 3-29 指出 variable_part_CR2 与 variable_part_CR1 基本相同，只是 paramB 的值是 FALSE。

表 3-29　该表指出 variable_part_CR2 与 variable_part_CR1 基本相同，只是 paramB 的值是 FALSE

Structure Constraint Declaration in ASN.1
Constraint Name: Variable part-CR2 Structure Type: VARIABLE-PART Derivation Path: Variable part-CR1 Encoding Rule Name: Encoding Variation:

续表

Constraint Value
REPLACE paramB with FALSE
Detail Comments:

3.11.2 匹配机制(Matching Mechanisms)

在大多数情况下，一个接收到的 PDU 值和一个指定值完全相同是不可能的。而一个接收到的 PDU 值落入一个值区间或满足一定条件是普遍的现象。

TTCN 支持众多的值匹配机制，这些机制有符号匹配(matching symbols)、操作匹配(matching operations)和属性匹配(matching attributes)。这些匹配机制允许一个测试器表达匹配条件，而不是指定具体的值。具体的匹配机制有：

- 值列表(lists of values)。
- 补值列表(complemented lists of values)。
- 整数范围(ranges of INTEGER values)。
- 任意值(any value)。
- 任意值或省略值(any value or omit value)。
- 通配符(wildcards)。
- 条件属性(if_present attribute)。
- 长度属性(length attributes)。

匹配一个值列表(Matching Value Lists)

一个约束值可以是一个或多个指定值的列表(指定值可以是一个表达式,所以列表中的元素可能十分复杂)。列表匹配是，如果一个接收到的值与列表中的任意一个元素匹配，则匹配成立；否则匹配失败。例如，接收到的值'00'B 或'11'B，这与列表('00'B, '11'B)匹配。

补值列表匹配(Complementing Value Lists)

补值列表匹配是，如果接收到的值与列表中的任意一个值都不相等，则匹配成功；否则匹配失败。补值列表是在一个列表前加上一个关键字 COMPLEMENT。与列表一样，列表中可以使用任何数据类型的值。例如，接收到的值'01'B 或'10'B，与 COMPLEMENT ('00'B,'11'B)匹配。这里的 COMPLEMENT ('00'B, '11'B)与列表(NOT '00'B, NOT '11B')等价。

范围匹配(Matching Ranges)

范围匹配仅与 INTEGER 兼容类型的数据匹配。使用关键字 INFINITY 和-INFINITY 表示正数和负数方向的范围。如果接收到的值落入一个指定的范围之内，则匹配成功，否则失败。例如一个范围(8…INFINITY)与任意大于 7 的整数值匹配。

匹配任意值(Matching Any Value)

在许多情况下，测试套只接收一个域的任意一个单值，并且该值与定义的相对应的值兼容。任意值匹配是基于这种情况下的匹配，当所接收到的值与所希望的值兼容，则匹配成功；否则失败。任意值使用?表示。例如，'00'B、'01'B、'10'B 和'11'B 与定义的长度为 2 的 BITSTRING 串匹配。

匹配任意值或完全省略(Matching Any Value, or Omitting It Altogether)

匹配任意值或完全省略(AnyOrOmit)用“*”表示。它与匹配任意值十分相似，只是所要匹配

的值可以省略。如果一个值出现，并且所接收的值与指定的类型兼容，则匹配成功；否则该值被省略。匹配任意值和完全省略仅用于可选域的匹配。

例如，假设已经声明了一个长度为 2 的 BITSTRING 类型的域，则接收到的'00'B、'01'B、'10'B、'11'B 和空与该域匹配。再如，我们已经声明了一个 SEQUENCE OF INTEGER 类型的域，则 SEQUENCE OF INTEGER 和空序列与其匹配。

值内通配符(Wildcards Within Values)

值内通配符有两种：

- 任意一个(AnyOne)匹配。
- 任意一个或无(AnyOrNone)匹配。

任意一个匹配用符号“?”表示，它表示数据中的？可以用任意同类型的单一元素替换。数据类型可以是 String、SEQUENCE、SEQUENCE OF、SET 和 SET OF 类型。然而，要匹配的元素不能被省略。

例如：

'?0'B 与'00'B 或'10'B 匹配。

"ab?z"将与任何长度为 4 的，并且以 ab 开头，以 z 结尾的字符串匹配。

一个 SEQUENCE OF INTEGER 类型，如{1, 2, ?, 3}与任何第 3 个元素为任何整数的 4 元素元组匹配。

注意：AnyOne 与 AnyValue 的符号表示是相同的，但符号的语义不同。

任意一个或无匹配用符号“*”表示，它用于实现单一元素和一个连续元素替换的匹配，匹配元素还可以省略。可以进行该匹配的数据类型有：String、SEQUENCE、SEQUENCE OF、SET 和 SET OF 类型。

例如：

'*0'B 将与任何以 0 结尾的 BITSTRING 类型值匹配。

"ab*z"将与任何以 ab 开头、以 z 结尾的任意字符串(包括"abz")匹配。

一个 SEQUENCE OF INTEGER 类型的值，如{1, 2, *, 3}与任何 SEQUENCE OF INTEGER 类型的以 1, 2 为第一、二元素，3 为第四元素的元组匹配，其中，第三元素可以为空，因此也可与{1,2,3}匹配。

If_Present 属性匹配

If_Present 属性匹配是专门为可选择域匹配所设计的一种类型的匹配。这是一种常见的情况，一个测试套事先不知道一个 IUT 是否包含一个可选择值，或者协议是否包含一个特殊的 PDU，所以测试套必须说明一个可选择的值，并检测该值。

例如，3 IF_PRESENT 表示如果整数值 3 出现则被接收。

注意：在 TTCN 表中，所有的域都被认为是可选择的，这意味着所接收的任意一个指定的类型值都将匹配。在 ASN.1 中就不是这样，它必须严格地加以说明。

长度限制(Length Restrictions)

长度限制可以应用于下列类型：

- BITSTRING。
- HEXSTRING。
- OCTETSTRING。
- CharacterString。

- SEQUENCE OF。
- SET OF。

实质上，长度限制可以对上述类型限值一个具体的范围,从而精简数据内容。

例如：HEXSTRING [8]，HEXSTRING [4 … 8]等。

一个在 PDU 中使用的匹配机制如表 3-30 所示。

表 3-30　一个在 PDU 中使用的匹配机制

PDU Constraint Declaration			
Constraint Name: A-constraint（PAR:BITSTRING） PDU Type: A-TTCN-PDU Derivation Path: Encoding Rule Name: Encoding Variation: Comments: this a constraint on incoming PDU			
Field Name	Field value	Field Encoding	Comments
FieldA	'00'B, '11'B		
FieldB	COMPLEMENT ('00'B, '11'B)		
FieldC	（8…INFINITY）		
FieldD	'?0B'		
FieldE	'*0B'		
FieldF	PAR[4...8]		
Detail Comments:			

3.12　编码

原 TTCN 标准中不涉及网络传送信息的编码。改版的 TTCN 增加了编码功能。它允许 TTCN 用户在下面的应用中说明编码。

- 所有的 ASP 和/或 PDU。
- 单独的 ASP 类型和/或 PDU 类型。
- 单独的 ASP 参数和/或 PDU 域。
- 单独的 ASP 约束和/或 PDU 约束。
- 单独的 ASP 约束参数和/或 PDU 约束域。

ASP 编码

TTCN 作为一个标准并不关心如何说明 ASP 参数的构成，对 ASP 参数的描述是实现范畴的内容，不在标准讨论的范围之内。TTCN 作为一个标准并不考虑 ASP 类型的绑定，即在测试套中给出一致的表示。也就是说，检查 ASP 参数与 ASP 实现的一致性目标，而不是 TTCN 的说明。

PDU 编码

与 ASP 不同，PDU 域在相关的协议标准中有类型定义。这些类型是 ETS 中正确实现的基础。就编码而言，TTCN 服从与 PDU 有关的标准。例如，若使用 ASN.1，那么 ASN.1 的编码规则（BER）就可以使用。当然，也可以不采用。如果采用该规则，则其他工作就应该服从该编码规则中的条款。

编码的操作

在一些测试过程中，需要处理一些值的编码，例如，在表述层的测试。

3.13　引用复合类型元素

TTCN 允许在表达式中把单个复合类型组件作为操作数使用，或者作为赋值语句的值。这些组件有：

- 一个 ASP 参数。
- 一个 PDU 域。
- 一个 structure 的元素。
- 一个 CM 域 。

在 ASN.1 中，引用可以访问下列元素：

- 在一个 BITSTRING 中的单个 BIT。
- 在一个 SEQUENCE 或 SEQUENCE OF 中的元素。
- 在一个 SET 或 SET OF 中的元素。
- 在一个 CHOICE 中的选择。

这些引用可以产生在：

- SEND 或 RECEIVE 语句。
- 捕获一个接收到的为后来所应用的 ASP 或 PDU。

3.13.1　在 SEND 和 RECEIVE 语句的上下文中引用

SEND 和 RECEIVE 语句中存在对 ASP 参数、PDU 域或构造类型的元素的引用。不仅如此，这些引用与约束相关。

这些引用可以用以下简单的符号表示：

```
ASP_Identifier . ParameterIdentifier
PDU_Identifer . FieldIdentifier
CM_Identifer . FieldIdentifier
StructuredTypeIdentifier . ElementIdentifier
```

假设一个子构造 PDU 被连接到一个 ASP，要在一个语句行中引用第 k 个元素，可以写成：

```
ASP_Identifier . Parameter_i . PDU_Identifier . field_j . StructureIdentifier .
element_k
```

然而，由于 ASP、PDU 和 structure 的标示符在测试套中是单一的，所以它可以化简成：

```
StructureIdentifier . element_k
```

例如，如果我们希望保存一个接收到的嵌入在 N_DATAindication 中的 DT_PDU.user_data 域的值，可以这样表示：

```
A:= N_DATAindication . user_data . DT_PDU . user_data
```

这是一个较烦琐的书写形式，但由于 PDU 的标示符是单一的，所以我们完全可以把上面的语句写成：

```
A:= DT_PDU . user_data
```

也就是说，点路径(dotted path)仅给出一个完全单一的引用。

3.13.2 引用 ASN.1 元素

相同的机制可以应用在 ASN.1 约束的引用中，这些引用约束包括使用 SEQUENCE、SEQUENCE OF 等类型。

假设我们已经定义了一个 PDU(如表 3-31 所示)。

表 3-31 一个 TTCN PDU 类型

ASN.1 PDU Type Definition
PDU Name: A-PDU PDU Type: N-SAP Encoding Rule Name: Encoding Variation: Comments: This is the type definition of the A-PDU
Type definition
SEQUENCE {Field1 BITSTRING; Filed2 SEQUENCE OF INTEGER; Field3 BOOLEN }
Detail Comments:

一个 PDU 约束如表 3-32 所示。

图 3-32 一个 TTCN PDU 约束

ASN.1 PDU Constraint Definition
PDU Name: a-PDU PDU Type: A-SAP Encoding Rule Name: Encoding Variation: Comments: This is the type definition of the A-PDU
Type definition
{Field1 '000'B; Filed2 21,22,23; Field3 true}
Detail Comments:

然后，可以有表 3-33。

表 3-33 一个 SEND 语句

序号	标签	行为描述	约束	结论	注释
1		…	…		
2		L！A_PDU(A.PDU.Field3=FOLSE)	a_pdu		
3		…	…		

表 3-33 意味着约束的第三个域被重载，并且发送对象 a_pdu 以第三个域为 FALSE 值被发送。

如果元素没有指定具体值，例如在前面的例子中的 $field_2$ 是 SEQUENCE OF INTEGER，则可以通过位置引用。例如，如果我们想重载 $field_2$ 的值为 22，则可以简单地表示为如表 3-34 所示。

BIT STRING 可以按位访问，如果想把 $field_1$ 中的第 3 位改成 1，则可以表示成如表 3-35 所示的形式。

表 3-34　一个 SEND 语句

序号	标签	行为描述	约束	结论	注释
1		…	…		
2		L！A_PDU（A.PDU.Field2（2）=22）	a_pdu		
3		…	…		

表 3-35　一个 SEND 语句

序号	标签	行为描述	约束	结论	注释
1		…	…		
2		L！A_PDU（A.PDU.Field1[3]=1）	a_pdu		
3		…	…		

上述用法只限于 BIT STRING 类型。

3.13.3　捕获接收到的 ASP 和 PDU

一个接收到的 ASP 或 PDU 仅在 RECEIVE 语句期间被保存，即被接收的对象不能被后来的接收事件所访问。

然而，在实际中把 ASP、PDU 或 structure type 变量作为接收对象。假设一个 A_PDU 类型的变量 temp_pdu，如表 3-36 所示。

表 3-36　一个 RECEIVE 语句

序号	标签	行为描述	约束	结论	注释
1		…	…		
2		L?A_PDUtemp_PDU:=A_PDU	a_pdu		
3		…	…		

我们现在访问 a_pdu，并正好不在包含 RECEIVE 语句行的语句中访问 a_pdu，如表 3-37 所示。

表 3-37　一个 QUALIFIER 语句

序号	标签	行为描述	约束	结论	注释
1		…	…		
2		[temp_PDUfield3]	a_pdu		
3		…	…		

3.14　裁决（Verdicts）

对测试例 TTCN 提供了如下两种裁决机制。

- 初步结果。
- 最终结果。

一个初步结果或最终结果都通过 TTCN 语句给出，但以下语句除外。

- IMPLICIT SEND。
- ATTACH。
- GOTO。
- REPEAT。

3.14.1 结果变量(Result Variable)

TTCN 有一个预定义的测试例变量，称为结果变量(result variable)，简称 R。该变量既可以在表达式中使用，又可以在行为树的裁决列中使用。结果变量用来存放初步的结果，它有如下特性。

- 一个初步裁决(结果)不能终止测试例的执行。
- 它可以作为一个只读变量在表达式中出现，即它不能出现在赋值语句的左边。
- 它仅能取 pass、fail、inconc 或类型定义中的一个值。这些值都是预定义的标识符，并且严格区分大小写。
- 它的值可以在裁决列中被改变。
- 在测试例开头，R 与一个类型定义绑定。

3.14.2 初步结果

裁决列中 R 的值由记录更新为一个初步结果。一个初步结果可能是下列之一。

- (P)或(PASS)，表明测试目的在一些方面已经达到。
- (I)或(INCONC)，表明一些测试目的之外的现象在测试例中出现。
- (F)或(FAIL)，表明协议有错误或测试目的没有达到。

例如，在裁决列中添入(FAIL)表明 R 的值为 FAIL。

初步结果有一个优先次序。例如，如果 R 的值为 FAIL 并且初步结果 PASS 在裁决列中，则 R 不能改变成 PASS，它还是 FAIL。反之，如果 R 的值是 PASS 并且一个初步结果 FAIL 在裁决列中出现，则 R 的值是 FAIL。表 3-38 给出 R 按优先级改变的情况。

表 3-38 初步结果变量 R 的计算

	(PASS)	(INCONC)	(FAIL)
none	pass	Inconc	Fail
pass	pass	Inconc	Fail
inconc	inconc	Inconc	Fail
fail	fail	Fail	Fail

3.14.3 最终结果(Final Verdicts)

一个测试例在以下几种情况下被终止。

- 一个测试例的行为树到达叶节点。
- 在一个行为行(裁决列)中有一个明确的最终裁决(fail or inconc)。

一个最终的裁决可能是下面几种情况。

- P 或 PASS，表明一个 pass 结果被记录。
- I 或 INCONC，表明一个例外结果被记录。
- F 或 FAIL，表明一个 fail 结果被记录。
- 预定义变量 R 表示最终结果，除非一个测试例被错误地记录。这时，R 的值是 none，而不是最终结果。

如果一个最终结果不能到达，则最终结果是 R 的值；如果 R 的值还是 none，则该测试例有错误。

最终结果一定与 R 值一致。例如，如果 R 的值是 fail 并且最终结果 pass 在裁决列中出现，则最终结果 fail 和 not pass 将被记录。反之，如果 R 的值是 pass 并且一个最终裁决 FAIL 在裁决列中出现，则一个最终结果 fail 将被记录。

表 3-39 展示出一个最终裁决将按照 R 值被记录。

表 3-39　最终裁决

	(PASS)	(INCONC)	(FAIL)	R
none	Pass	Inconc	Fail	*error*
pass	Pass	Inconc	Fail	pass
inconc	*error*	Inconc	Fail	inconc
fail	*error*	*error*	Fail	fail

3.15　GOTO 语句

为了用简单的形式表达循环过程，TTCN 允许在语句前加标号，并使用 GOTO 语句实现过程转移。GOTO 语句格式如下：

```
    -> LabelIdentifier
```

或

```
    GOTO LabelIdentifier
```

使用 GOTO 语句时应该避免无限循环，可以通过条件判断和事件发生来确定是否进入循环。使用 GOTO 语句的行为树如表 3-40 所示。

表 3-40　使用 GOTO 语句的行为树

序号	标签	行为描述	约束	结论	注释
1		…			
2	LAB	L!N_DATArequest(count=count+1)	…		
3		Count<=Max	NDr		
			…		
7		->LAB			

使用 GOTO 时应遵守以下规则：

- 一个 GOTO 只允许在行为描述中的一棵树中出现，即在一个替换子集中使用。
- 在一个行为描述中的一个标号必须是唯一的。
- 行号不应该作为标号。
- 一个标号必须是一个替换集的第一个语句，也就是说，GOTO 语句实现的转移不应该是替换集的中间语句。
- 一个 GOTO 语句不能转移到最上一层的替换(测试步的根节点)语句。
- 在行为树中，一个 GOTO 语句只可以转移到它的祖先节点，即仅能转移到已经执行过的节点。
- 在 GOTO 语句中不能使用其他语句。

3.16　定时器语句

TTCN 定时器被用来测试 IUT 中的定时器事件。一个定时器语句通常用于对 IUT 应答的时间设置，并发出 START 命令和 TIMEOUT 事件。另一个与定时器相关的命令是 CANCEL，它用来终止一个定时器。所有的定时器必须在定时器描述表中定义。Duration 是从定时器启动时刻到定时器停止的时间间隔，该间隔可以使用以下单位计量。

- ps(皮秒)。
- ns(纳秒)。
- μs(微秒)。
- ms(毫秒)。
- s(秒)。
- min(分)。

Timers 声明如表 3-41 所示。

表 3-41 Timers 声明

Timer Declarations			
Time Name	Duration	Unit	Comment
Timer	1	S	Maximum response time
Detail Comment:			

超时列表

TTCN 维持一个超时列表。如果一个定时器期满，则该定时器被加入到列表中。有三种方法可以把定时器从表移出：

- 一个成功的 TIMEOUT 语句。
- 使用定时器启动命令 START。
- 使用定时器终止命令 CANCEL。

TIMEOUT 语句

在 TTCN 测试套中可以定义一个定时器，用于检查一个应答是否超时。语句格式如下：

```
?TIMEOUT TimerIdentifier
```

当执行到该语句时，测试套将检查该定时器是否在超时列表中，如果与超时列表中的一个定时器匹配，则 TIMEOUT 发生，否则 TIMEOUT 不发生。

该语句的另一种格式是：

```
?TIMEOUT
```

即没有指定 timerIdentifier。在这种情况下，只要超时列表不空，就认为 TIMEOUT 匹配成功。

TIMEOUT 可以与条件组成带条件的 TIMEOUT 语句，该语句可以后接 ASSIGNMENT_LIST、TIMER_OPERATION 等命令，格式如下：

```
TIMEOUT[2] [QUALIFIER][1] [ASSIGNMENT_LIST][3] [TIMER_OPERATION][4]
```

其中，方括号内为可选择项。该语句中的各分量的执行顺序一定的。

注：TIMEOUT 不能被用于阻止一个 IUT 部件发送应答。

定时器快照

前面已经提到过，在每一次执行替换集中的动作时都要先检查 PCO 队列的当前状态，瞬间的状态称为快照(snapshot)。然后，替换集中的动作与状态比较。对于超时列表也有同样的概念。在每一次开始执行替换集循环时，当执行到一个 TIMEOUT 动作时，该超时快照(时刻)被检查，这里并不是实际的超时列表值。这意味着，在整个替换集执行期间，直到定时器快照被更新时，定时器才注册。

定时器启动命令 START

一个定时器通过 START 启动。格式如下：

```
START TimerIdentifier
```

在定时器被启动后，可以显示指明定时器的时间区间。如果指定，则该区间将覆盖原有的定时器时间区间。

显示声明时间区间的语句格式如下：

```
START TimerIdentifier ( Duration )
```

如果该定时器正在运行时，START 被激活，则定时器被取消并重新开始；如果该定时器已经到期，则在重新启动之前被移出表格。START_TIMER 语句也可以是有条件的，一个条件 START_TIMER 语句格式如下：

```
[QUALIFIER]1 [ASSIGNMENT_LIST]2 [START_TIMER]3
```

在行为树中使用 START 和 TIMEOUT 如表 3-42 所示。

表 3-42　在行为树中使用 START 和 TIMEOUT

序号	标签	行为描述	约束	结论	注释
1		…	…		
2		START a_timer			
3		?TIMEOUT a_timer	…		

CANCEL 命令

一个计时器可以使用 CANCEL 命令取消。CANCEL 命令的格式如下：

```
CANCEL TimerIdentifier
```

另一个 CANCEL 格式为：

```
CANCEL
```

即没有 TimerIdentifier。在这种情况下，所有的定时器被取消，并且清除超时列表后重新启动。

当取消一个到期定时器时，该定时器重新启动并从超时列表中移出。取消定时器语句也可以被加上条件，格式如下：

```
[QUALIFIER]1 [ASSIGNMENT_LIST]2 [CANCEL_TIMER]3
```

3.17　常量与变量

TTCN 也有变量和常量之分，TTCN 有以下两种类型的常量。

- 测试套参数。
- 测试套常量。

有以下两种类型的变量：

- 测试套变量。
- 测试例变量。

这些变量和常量通过以下表格来定义。

- 测试套常量声明。
- 测试套参数声明。
- 测试套变量声明。
- 测试例变量声明。

执行包含一个条件的替换集如图 3-8 所示。

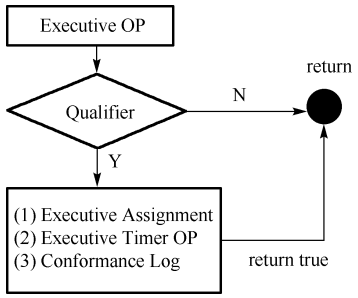


图 3-8 执行包含一个条件的替换集

测试套常量和测试套参数

测试套常量是一种全局常量，该类型的常量可以在测试套的任何一个地方使用，也包括约束部分。在测试套中一个常量一旦被声明就不会改变。测试套常量声明如表 3-43 所示。

表 3-43 测试套常量声明

Test suite constant declarations			
Constant name	Type	value	Comment
Max	Integer	5	
Data_string	IA5String	This is a string	
Detail comment:			

测试套参数

测试套参数也是常量，但对抽象测试套描述它的实际值是未知的。它们的值由 IUT 来确定，并且很可能依赖系统本身。从这种意义上讲，不同的 IUT 它们的测试套参数是不同的，但是在测试一个 IUT 期间，它们是常量。测试套参数声明如表 3-44 所示。

表 3-44 测试套参数声明

Test suite parameter declarations			
parameter name	Type	PICS/PIXIT ref	Comment
LT_address	IA5String	PIXIT question xx	Lower tester address
UT_address	IA5String	PIXIT question yy	Upper tester address
Detail comment:			

测试套参数值是从协议实现一致性声明(Protocol Implement Conformance Statement，PICS)和协议实现附加测试信息(Protocol Implement eXtra Information for Test，PIXIT)中导出的。这些文本就像一个按照 IUT 特性建立的内嵌检查表。

在执行一个测试套之前，PICS 和 PIXIT 与测试套参数绑定，这一过程称为测试套的参数化。

测试套与测试例变量

测试套和测试例变量也是全局变量，它们可以在测试套的任何一个地方使用。每一个测试套或测试例变量都可以有一个缺省值。如果没有设定缺省值，则称该变量是非绑定。一个变量在使用之前应该被绑定，除非该变量出现在等号的左边。

测试例变量声明如表 3-45 所示。

表 3-45 测试例变量声明

Test Case Variable Declaration			
Variable type	Type	Value	Comment
Count	Integer	0	The test case variable is used to count the number of sent and receive
Detail Comment:			

重新定义缺省值

测试套变量和测试例变量的不同在于它们的缺省值重新设置(如果没有缺省值,则意味着变量为非绑定的):

- 测试套变量在测试套的结束部分被重新设置,这意味着该变量的信息在测试例执行之间被保留。
- 测试例变量在测试例的结束部分被重新设置,这意味着在每一个测试例开始时测试例变量与缺省绑定。

并发 TTCN 中的变量

当多个测试组件被执行时，被称为并发 TTCN。在并发执行时，每一个测试组件都有一个测试例变量的副本。

在例子中，我们声明了测试例变量 count，该变量可以分别在低端和高端测试器中使用，也就是说，在低端的测试例中改变了变量的值后，不会影响该变量在高端的值，反之亦然。测试套变量在并发 TTCN 中的使用与串行 TTCN 的情况是相同。

3.18　动态行为描述

TTCN 中有以下三种描述行为的表格。

- 测试例动态行为表。
- 测试步动态行为表。
- 缺省的动态行为表。

这些表格从表头可以明显地加以区分，而不是通过表的内容来区分。

一个测试例行为表的框架的表头部分如表 3-46 所示。

表 3-46　一个测试例行为表的框架的表头部分

Test Case Dynamic Behaviour					
Test Case Name: MP_DATA_TRANSFER Group: MULT/DATA/ Purpose: IUT shall receive and send with timeout limit, a given number of times over of two simultaneous Configuration: Multi_Party Default: T_Default Comments: This test case create the other PICs in the configuration necessary for a Two_connection configuration. Selection Ref: Description: Data transfer multi_connetion					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		Create(LOWER_TEST:LTS(L1,CP1) LOWER_TEST: LTS(L2,CP2) UPPER_TEST: UTS(U1) UPPER_TEST: UTS(U2))			
2		CP1?PTC_RESULT	PTC_RES(pass)		
3		CP2?PTC_RESULT	PTC_RES(pass)	PASS	
4		CP2?PTC_RESULT	PTC_RES(fail)		
5		CP1?PTC_RESULT	PTC_RES(fail)		
6		CP2?PTC_RESULT	PTC_RES(pass)	FAIL	
7		CP2?PTC_RESULT	PTC_RES(fail)	FAIL	

测试例标识和测试组引用

与其他 TTCN 标识的使用方法相同，测试例的标识符放在测试例定义的前面，而且一个测试例的标识符在整个测试套中是唯一的。在测试例标识符的后面是测试例引用，它是一个用路径名来指出的该测试例在测试套中的位置。

当该路径在测试步中引用时，该路径指出的是该测试步在测试步库中的位置；如果没有路径，则表示测试步在缺省库中的缺省位置。引用有以下一般形式：

$$\text{SuiteIdentifier / GroupIdentifier}_1 \text{ / . . . / GroupIdentifier}_n \text{ /}$$

注意：路径是用反斜线作为分割符，最后一个分割符指明了最后一个测试组的名称。一个路径可以是测试组标识符，即测试套标识符是可选的，如果测试套不是层次结构，则引用是空。

测试目的和对象

在测试例表中的第三行用于说明测试目的。与之相对应的测试步和缺省的动态行为表中称为测试对象。

配置

测试例的行为描述通过引入并发 TTCN 的配置实体来加以说明。在测试步中和缺省中不明确地说明测试配置。

缺省行为

如果存在，缺省的实体被用于说明将被使用的缺省行为。

测试步态行为如表 3-47 所示。

表 3-47 测试步态行为

Test Step Dynamic Behaviour					
Test Step Name: LT_DATA_TRANSFER(L:N_SAP;CPP:CP) Group: TEST_STEP_LIB/LOWER/ Objective: Test data transfer Default: LT_DEFAULT(L) Comments: This step implements the body of our example test case on the lower tester side Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
	LAB	L!DATAout(count:=INC(count))	NDr(DT1(data_string))		
		[count<=max] START Timer			
		L?DATAin	NDr(DT1(data_string))	Pass	
		->LAB			
		?TIMEOUT timer		fail	
		CPP!PTC_RESULT	PTC_RES(fail)		
		CPP!PTC_RESULT	PTC_RES(pass)		
Detail Comments:					

3.19 使用别名

TTCN 的主要目标之一是提供一种易读易懂的形式化描述，从而方便测试工作。这就要求在 N 和 N–1 层的 ASPs 之间提供标准的行为描述。然而，由各层的数据需求和表述构成的行为树提供的描述信息并不是很多。服务原语中的哪一个 PDUs 是重要的？如果仅使用静态链，没有数据的约束，读者就不能在测试过程了解 PDU 之间的交互。

TTCN 引入别名的机制，允许 ASPs(包括 PDUs)被重新命名来反映 ASPs 中的 PDUs 的不同。N–1 层数据需求和表述的别名使用取决于 N 层中的 PDU。

别名声明如表 3-48 所示。

表 3-48　别名声明

Alias Definition		
Alias Name	Expansion	Comments
CR	N_DATArequest	Alias for N_DATArequest service primitive used to carry a CR_PDU
DATAout	N_DATArequest	Alias for N_DATArequest service primitive used to carry a outgoing DT_PDU
CC	N_DATAindication	Alias for N_DATAindication service primitive used to carry a CC_PDU
DATAin	N_DATAindication	Alias for N_DATAindication service primitive used to carry a incoming DT_PDU
DR	N_DATArequest	Alias for N_DATArequest service primitive used to carry a outgoing DR_PDU

在 ISO/IEC 9646-3 中，别名被定义成文本的扩展。然而，在 SEND 和 RECEIVE 语句中可以把别名用来代替 ASP 或 PDU 的标识符，下面的语句是有意义的。

- PCO_Identifier ! AliasIdentifier。
- PCO_Identifier ? AliasIdentifier。

测试步动态行为中使用别名如表 3-49 所示。

表 3-49　测试步动态行为中使用别名

Test Step Dynamic Behaviour					
Test Step Name:　LT_DATA_TRANSFER(L:N_SAP;CPP:CP) Group: TEST_STEP_LIB/LOWER/ Objective: Test data transfer Default: LT_DEFAULT(L) Comments: This step implements the body of our example test case on the lower tester side Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1	LAB	L!DATAout(count:=INC(count))	NDr(DT1(data_string))		
2		[count<=max] START Timer			
3		L?DATAin	Ndi(DT1(data_string))	Pass	
4		->LAB			
5		?TIMEOUT timer		fail	
6		CPP!PTC_RESULT	PTC_RES(fail)		
7		CPP!PTC_RESULT	PTC_RES(pass)		
Detail Comments:					

3.20　测试例模块化

一个测试例可能会很复杂，所以 TTCN 提供了两种测试例模块化的方法：一是测试步；二是缺省。

3.20.1　测试步

行为树可以通过划分子树的方法实现模块化，这些子树称为测试步。测试步或者是本地描述，或者是存放在测试步库中。

测试步可以带有参数，即一个行为树可以把 PCOs、变量、文本值和约束等作为参数传递一个测试步。

本地测试步

一个本地测试步是仅出现该行为时的一种描述，如图 3-9 所示。

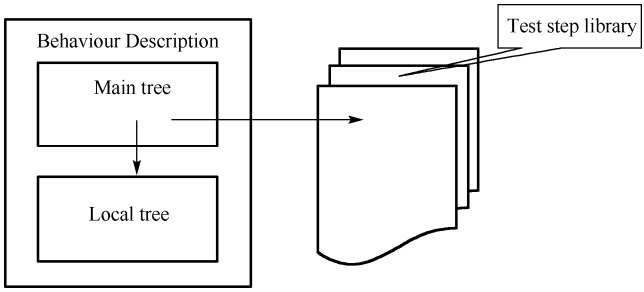


图 3-9 局部测试树和测试步解释

测试步库

测试步库中的测试步在测试步行为表中说明。这些测试步可以被任何测试例、测试步或缺省调用。

ATTACH 语句

ATTACH 语句被用于激发一个测试步，其语句格式如下。

- + TreeIdentifier ActualParameterList 用于本地的测试步。
- + TestStepIdentifier ActualParameterList 用于测试步例库中的测试步。

如果测试步有形参列表，则上面两种类型的测试步一定有实参。这里的参数可以是 PCO 或 CP。

行为树的隶属作为子程序调用

TTCN 定义树的隶属作为被调用树的扩充，即把测试步作为一个子程序，它可以被一个测试例或测试步调用。当用 TTCN 能够恰当地描述这一方式的时候，则把测试步作为子程序是有实际意义的。同时，有过编程经验的人都容易理解。

可以采用以下几种不同的风格来处理测试子程序：

- 当一个替换集合的执行循环执行到一个隶属语句时，其控制被转到隶属的测试步。
- 如果在替换集执行期间，测试步中的替换执行循环中没有成功的测试步，则从测试步返回到调用树，并且继续执行后面的替换；如果在同一个替换集中有隶属，则该测试步可以作为一个调用树。
- 如果测试步的第一层的替换是成功的，则继续在此测试步中的树中执行。
- 如果测试步到达最终的结论，则测试步挂起，控制不返回到用树。
- 如果没有到达最终结论，在没有到达测试步的叶节点时，控制返回到调用树，继续执行替换集中的下一个替换。

测试步动态行为(LTS)如表 3-50 所示。

表 3-50 测试步动态行为(LTS)

Test Step Dynamic Behaviour					
Test Step Name: LTS(L:N_SAP;CPP:CP) Group: TEST_STEP_LIB/LOWER/ Objective: IUT shall receive and send a data within time limit Default: Comments: Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
	LAB	+ESTABLISH_CONNECTION(L)			

续表

Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
		+LT_DATA_TRANSFER(L,CCP)			
		+CLOSE_CONNECTION(L)			
Detail Comments:					

测试步动态行为(LT)如表 3-51 所示。

表 3-51　测试步动态行为(LT)

Test Step Dynamic Behaviour					
Test Step Name:　LTS(L:N_SAP;CPP:CP) Group: TEST_STEP_LIB/LOWER/ Objective: IUT shall receive and send a data within time limit Default: LT_DEFAULT(L) Comments: Description:					
Nr	Label	Behaviour Description	Constraint Ref	V	C
1		+ESTABLISH_CONNECTION(L)			
2	LAB	L!DATAout(count:=INC(count))	NDr(DT1(data_string))		
3		[count<=max] START Timer			
4		L?DATAin	NDr(DT1(data_string))	Pass	
5		->LAB			
6		+CLOSE_CONNECTION(L)			
7		?TIMEOUT timer		fail	
8		CPP!PTC_RESULT	PTC_RES(fail)		
9		+CLOSE_CONNECTION(L)			
10		CPP!PTC_RESULT	PTC_RES(pass)		
11		+CLOSE_CONNECTION(L)			
Detail Comments:					

3.20.2　缺省行为

作为一致性标准需要 TTCN 的测试例对行为的描述是完备的。这意味着在任一时刻，测试组件都准备接收可能的 ASPs 或 PDUs。其中，包括任何协议中规定的 ASPs 或 PDUs，也包括其他类型的 IUT 或服务提供者发出的 ASPs。

一个最简单的方法是使用 OTHERWISE 语句。然而，TTCN 使用 OTHERWISE 语句导向一个失败的结论，这不是我们所希望的。

例如，一个网络服务在某时刻发出一个 N_DISCONNECTindication 是绝对合法的 ASP，但这时该原语被 OTHERWISE 处理就会得到一个完全错误的结论。此时，一个唯一的结论应该是 INCONCLUISIVE，即提醒注意。

在行为树中描述各种行为的可能组合，会忽略主要的行为，从而使测试例的可读性降低。使用缺省方法来说明非主要的行为，可以使描述更清晰。在 TTCN 中，可以使用 OTHERWISE 语句来对缺省的 ASPs 或 PDUs 进行处理，而这些 ASPs 或 PDUs 通常不是测试目标的一部分。TIMEOUT 作为缺省的情况是比较常见的。

缺省行为建模

缺省行为可以作为一个树的隶属，该隶属在每一个替换集合的最后被执行。

缺省引用

一个测试例或测试步在一个缺省实体的头部引用缺省行为，如果该实体是空的，则没有可以用的缺省。

缺省行为描述如表 3-52 所示。

表 3-52 缺省行为描述

Test Step Dynamic Behaviour					
Test Step Name: LT_DEFAULT(L:N_SAP)					
Group: DEFAULT_LIB					
Objective: General catch all for lower tester					
Default:					
Comments:					
Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		L?OTHERWISE		FAIL	The test stop immediately
Detail Comments:					

带有缺省的测试步如表 3-53 所示。

表 3-53 带有缺省的测试步

Test Step Dynamic Behaviour					
Test Step Name: LTS(L:N_SAP;CPP:CP)					
Group: TEST_STEP_LIB/LOWER/					
Objective: IUT shall receive and send a data within time limit					
Default: LT_DEFAULT(L)					
Comments:					
Description:					
Nr	Label	Behaviour Description	Constraint Ref	Vs	Cs
1		+ESTABLISH_CONNECTION(L)			
2	LAB	L!DATAout(count:=INC(count))	NDr(DT1(data_string))		
3		[count<=max] START Timer			
4		L?DATAin	Ndi(DT1(data_string))	Pass	
5		->LAB			
6		+CLOSE_CONNECTION(L)			
7		?TIMEOUT Timer		fail	
8		CPP!PTC_RESULT	PTC_RES(fail)		
9		+CLOSE_CONNECTION(L)			
10		CPP!PTC_RESULT	PTC_RES(fail)		
11		+CLOSE_CONNECTION(L)			
Detail Comments:					

不带有缺省的测试步如表 3-54 所示。

表 3-54 不带有缺省的测试步

Test Step Dynamic Behaviour
Test Step Name: LTS(L:N_SAP;CPP:CP)
Group: TEST_STEP_LIB/LOWER/
Objective: IUT shall receive and send a data within time limit
Default: LT_DEFAULT(L)
Comments:
Description:

续表

Nr	Label	Behaviour Description	Constraint Ref	Vs	C
1		+ESTABLISH_CONNECTION(L)			
2	LAB	L!DATAout(count:=INC(count))	NDr(DT1(data_string))		
3		[count<=max] START Timer			
4		L?DATAin	Ndi(DT1(data_string))	Pass	
5		->LAB			
6		+CLOSE_CONNECTION(L)			
7		L?OTHERWISE		FAIL	
8		L?OTHERWISE		FAIL	
9		L?OTHERWISE		FAIL	
10		L?OTHERWISE		FAIL	
11		?TIMEOUTtimer			
12		CPP!PTC_RESULT	PTC_RES(fail)		
13		L?OTHERWISE		FAIL	
14		+CLOSE_CONNECTION(L)			
15		L?OTHERWISE		FAIL	
16		L?OTHERWISE		FAIL	
17		CPP!PTC_RESULT	PTC_RES(fail)		
18		+CLOSE_CONNECTION(L)			
19		L?OTHERWISE		FAIL	
20		L?OTHERWISE		FAIL	
21		L?OTHERWISE		FAIL	
22		L?OTHERWISE		FAIL	
Detail Comments:					

3.21　TTCN 中的参数列表

下列 TTCN 对象可以作为参数：

- 测试套。
- 约束。
- 测试步。
- 缺省。

形参列表

TTCN 的形参列表使用下面格式：

- an_identifier（fpar1, fpar2:INTEGER, fpar3:HEXSTRING）

其中，an_identifier 是 TTCN 对象，括号内为形参。

实参列表

一个参数化的对象被一个实参列表取代。例如：

- an_identifier(1, 2, FALSE)

在使用参数时有以下规则：

- 实参与形参必须在数量上严格的一一对应。
- 实参的数据类型必须与形参列表中的数据类型兼容。
- 所有的实参必须在一个测试套操作、测试步、约束或缺省时与形参进行绑定。
- 所有的实参必须有明确的值。

传址调用

在测试步或缺省中，TTCN 使用源值取代定义实参的传递。一个描述这种参数传递的更直接的方法是使用传址调用。在传址调用中，直接访问实参的地址而不是实参的一个拷贝，所以在子程序(测试步或缺省)中，变量值的变化直接影响到原来变量的值。

传值调用

TTCN 标准规定，任何用户定义操作和约束都不能被参数值改变，即不能由参数的引用而产生源值的变化。所以 TTCN 提供子程序参数的传值调用，在传值调用中，只拷贝变量的值，而不是变量的地址，所以不会影响变量原来的值。

3.22 测试例选择

一个测试套中可能包括数以百计或数以千计的测试例。在大多数情况下，一个测试要在众多测试例中选择部分运行，这种选择过程称为测试例选择。根据 PICS 和 PIXIT 的具体值和要求，测试套中仅有一个子集被执行。

选择表达式

TTCN 允许每一个测试例配置一个选择表达式。这些表达式是一个有真假值的断言，其真假值根据 PICS 和 PIXIT 来确定。如果无选择断言，则该表达式总是被选择的。

表达式断言在测试例选择表达式定义表中定义，并且在测试例索引中被引用。

在测试套结构表中，可以对选择表达式创建引用的方法选择一组测试例。

3.23 TTCN 测试套结构

在测试套层次结构中，每一个 TTCN 对象都有一个明确的位置。

测试套构成

不同的测试套组件在测试套中出现的次序是不同的，一个 TTCN 测试套由四个部分组成。

- 测试套头部。
- 声明部分。
- 约束部分。
- 行为部分。

上面的每一部分都包含许多 TTCN 表。这些表出现的次序在下面给出。顺序中，每个项目编号代表一个 TTCN 表，每一个有数字下标代表一个单一的 TTCN 对象，例如 PDUs 和测试例。没有数字下标的表是多重 TTCN 对象表，例如简单类型定义或测试套变量。

有些表采用压缩形式，而斜体字的表在 TTCN 扩展中定义。

测试套头部

测试套头部由四个部分构成：

- 测试套结构。
- 测试例索引。
- 测试步索引。
- 缺省索引。

声明部分

声明部分用于定义新的数据类型、新的操作和声明所有的测试套组件。

- 测试组件声明。
- 测试组件配置声明。
- 简单类型定义。
- 结构化类型定义 1。
- …
- ASN.1 类型定义 1。
- …
- ASN.1 类型定义通过引用。
- 测试套操作定义 1
- …
- 测试套参数声明。
- 测试例选择表达式定义。
- 测试套常量声明。
- 测试套变量声明。
- 测试例变量声明。
- PCO 声明。
- CP 声明。
- Timer 声明。
- ASP 声明 1。
- …
- ASN.1 ASP 类型定义 1。
- …
- ASN.1 ASP 使用引用的类型定义。
- PDU 类型定义 1。
- …
- ASN.1 PDU 类型定义 1。
- …
- ASN.1 PDU 使用引用的类型定义。
- TTCN CM 类型定义 1。
- …
- ASN.1 CM 类型定义 1。
- …
- 别名声明。

约束部分

约束部分包括所有的 ASP、PDU、structure 和 CM 约束，它们使用 TTCN 表和 ASN.1 两种表形式。

- ASP 约束声明 1。

  …

  注意：ASP 约束可以使用压缩形式显示。

- ASN.1 ASP 约束声明 1。

  …

  注意：ASN.1 ASP 约束可以使用压缩形式显示。
- PDU 约束声明 1。

  …

  注意：PDU 约束可以使用压缩形式显示。
- ASN.1 PDU 约束声明 1。

  …

  注意：ASN.1 PDU 约束可以使用压缩形式显示。
- 构造类型约束声明 1。

  …

  注意：构造类型约束可以使用压缩形式显示。
- ASN.1 类型约束声明 1。

  …

  注意：ASN.1 类型约束可以使用压缩形式显示。
- CM 约束声明 1。

  …
- ASN.1 CM 约束声明 1。

  …

动态部分

动态部分包括所有的测试例、所有测试步库中的测试步和所有缺省库中的缺省定义。

- 测试例动态行为 1。

  …

  注意：测试组，即可以采用压缩形式显示。
- 测试步动态行为 1。

  …

  注意：测试步组不在此处定义
- 缺省动态形为 1。

  …

  缺省组不在此定义。

1．测试套总揽部分

测试套结构如表 3-55 所示。

表 3-55 测试套结构

Test Suite Structure
Suite Name: TTCN_TUTORIAL
Standards Ref: ISO/IEC XXXX
PICS Ref: ISO/IEC aaaa
PIXIT Ref: ISO/IEC bbbb
Test Method(s):Distributed single layer(DSE)
Comments:

续表

Test Group Reference	Selection Ref	Test group Objective	Page Nr
SINGLE/		Tests run over single connection	
SINGLE/DATA/			
MULT/		Tests run over multiple connection	
MULT/DATA			
Detailed Comments:			

测试步索引如表 3-56 所示。

表 3-56　测试步索引

Test Step Index			
Test Step Group Reference	Test Step Id	Description	Page Nr
Test_step_lib/lower/	LTS		
Test_step_lib/lower/	ESTABLISH_CONNECT		
Test_step_lib/lower/	LT_DATA_TRANSFER		
Test_step_lib/lower/	CLOSE_CONNECTION		
Test_step_lib/upper/	UTS		
Test_step_lib/upper/	ACCEPT_CCNNECTION		
Test_step_lib/upper/	UT_DATA_TRANSFER		
Detailed Comments:			

缺省索引如表 3-57 所示。

表 3-57　缺省索引

Default Index			
Default Group Reference	Default id	Description	Page Nr
DEFAULT_LIB	LT_DEFAULT		
DEFAULT_LIB	UT_DEFAULT		
DEFAULT_LIB	T_DEFAULT		
Detailed Comments:			

2．声明部分

简单类型定义如表 3-58 所示。

表 3-58　简单类型定义

Simple Type Definitions			
Type Name	Type Definition	Type Encoding	Comments
RESULT_TYPE	R_Type		
Detailed Comments:			

构造类型定义如表 3-59 所示。

表 3-59　构造类型定义

Structure Type Definition			
Type Name: VARIABLE_PART Encoding variation: Comments: This is the type definition of the variable part of the CR_PDU and the CC_PDU			
Elements Name	Type Definition	Field Encoding	Comments
ParamA_id	BITSTRING[2]		Parameter identifier
ParamA	OCTETSTRING[2..4]		Optional parameterA
ParamB_id	BITSTRING[2]		Parameter identifier

续表

Elements Name	Type Definition	Field Encoding	Comments
ParamB	BOOLEAN		Optional parameterB
Detailed Comments:			

测试套操作定义如表 3-60 所示。

表 3-60 测试套操作定义

Test Suite Operation Definition
Operation Name: INC（I:INTEGER） Result Type: Integer Comments: The INCrement operation.
Description

Int INC（i）
Int temp;
{
return（temp+1）:/*return the incremented value of i note that i self not change
}

Detailed Comments:

测试套参数声明如表 3-61 所示。

表 3-61 测试套参数声明

Test Suite Parameter Declarations			
Parameter Name	Type	PICS/PIXIT Ref	Comments
IT_ADDRESS	IA5STRING	PIXIT question xx	Lower tester address
UT_address	IA5STRING	PIXIT question yy	Upper tester address
Detailed Comments:			

测试套常量声明如表 3-62 所示。

表 3-62 测试套常量声明

Test Suite Constant Declarations			
Constant Name	Type	Value	Comments
max	INTEGER	5	
Data_string	IA5String	'This is a string'	
Detailed Comments:			

测试套变量声明如表 3-63 所示。

表 3-63 测试套变量声明

Test Case Variable Declarations			
Variable Name	Type	Value	Comments
Count	INTEGER	0	This test case variable is used to count the number of PDU sent and received

PCO 类型声明如表 3-64 所示。

表 3-64　PCO 类型声明

PCO Declarations			
Pco name	Pco type	Role	Comments
L1	N-ASP	LT	N serve access point at lower tester
L2	N-ASP	LT	
U1	X-ASP	UT	X serve access point at upper tester
U2	X-ASP	UT	
Detail Comments:			

协同点声明如表 3-65 所示。

表 3-65　协同点声明

Coordination Declarations	
Cp name	Comments
Cp_1	Coordination between the MTC and PTC of the lower tester
Cp_2	
Detail Comments:	

测试组件声明如表 3-66 所示。

表 3-66　测试组件声明

Test Component Declarations				
Component Name	Component Role	Nr PCOs	Nr CPs	Comments
MASTER_LOWER_TESTER	MTC	0	2	Parallel Test Comments
LOWER_TESTER1	PTC	1	1	Parallel Test Comments
LOWER_TESTER2	PTC	1	1	Parallel Test Comments
UPPER_TESTER1	PTC	1	0	Parallel Test Comments
UPPER_TESTER2	PTC	1	0	Parallel Test Comments
Detailed Comments:				

测试组件配置声明如表 3-67 所示。

表 3-67　测试组件配置声明

Test Components Configuration Declaration			
Configuration Name: SINGLE_PARTY Comments: Configuration to test a single connection			
Components Used	PCOs used	CPs Used	Comments
MASTER_LOWER_TESTER		CP1	MTC
LOWER_TESTER1	L1	CP1	Lower PTC
UPPER_TESTER2	U1		Upper PTC
Detailed Comments:			

多重测试组件配置声明如表 3-68 所示。

表 3-68　多重测试组件配置声明

Test Components Configuration Declaration
Configuration Name: MULTI_PARTY Comments: Configuration to test a single connection

续表

Components Used	PCOs used	CPs Used	Comments
MASTER_LOWER_TESTER		CP1,CP2	MTC
LOWER_TESTER1	L1	CP1	Lower PTC
UPPER_TESTER1	U1		Upper PTC
LOWER_TESTER2	L2	CP2	Lower PTC
UPPER_TESTER2	U2		Upper PTC
Detailed Comments:			

ASP 类型声明如表 3-69 所示。

表 3-69 ASP 类型声明

ASP TYPE Definition		
ASP Name: N-DATArequest PCOTYPE: N-SAP Comments: This is the type definition of the N-DATArequest ASP. It has a single parameter used to carry user data		
Parameter Name	Parameter Type	Comments
User-data	PDU	The PDU meta type is sued to indicate that UT going（i.e from LT）PDUs are embedded in this Network ASP
Detailed Comments:		

ASP 类型声明如表 3-70 所示。

表 3-70 ASP 类型声明

ASP TYPE Definition		
ASP Name: N-DATAindication PCOTYPE: N-SAP Comments: This is the type definition of the N-DATAindication ASP. It has a single parameter used to carry user data		
Parameter Name	Parameter Type	Comments
User-data	PDU	The PDU meta type is sued to indicate that UT going（i.e from LT）PDUs are embedded in this Network ASP
Detailed Comments:		

X-CONNECTidication 类型声明如表 3-71 所示。

表 3-71 X-CONNECTidication 类型声明

ASP TYPE Definition		
ASP Name: X-CONNECTidication PCO TYPE: X-SAP Comments: This is the type definition of the X-CONNECTidication ASP. It is issued by IUT to UT		
Parameter Name	Parameter Type	Comments
Calling address	IA5string	
Calling address	IA5string	
User-data	IA5string[0…32]	
Detailed Comments:		

X-CONNECTresponse 类型声明如表 3-72 所示。

表 3-72 X-CONNECTresponse 类型声明

ASP TYPE Definition
ASP Name: X-CONNECTresponse PCO TYPE: X-SAP Comments: This is the type definition of the X-CONNECTresponse ASP. It is issued by UT to IUT

续表

Parameter Name	Parameter Type	Comments
Calling address	IA5string	
Calling address	IA5string	
User-data	IA5string[0…32]	
Detailed Comments:		

ASP 类型声明如表 3-73 所示。

表 3-73　ASP 类型声明

ASP TYPE Definition		
ASP Name: X-DATArequest PCOTYPE: X-SAP Comments: This is the type definition of the X-DATArequest ASP. It is issued by UT to IUT		
Parameter Name	Parameter Type	Comments
User-data	IA5string[0…32]	
Detailed Comments:		

X-DATAindication 类型声明如表 3-74 所示。

表 3-74　X-DATAindication 类型声明

ASP TYPE Definition		
ASP Name: X-DATAindication PCOTYPE: X-SAP Comments: This is the type definition of the X-request ASP. It is issued by IUT to UT		
Parameter Name	Parameter Type	Comments
User-data	IA5string[0…32]	
Detailed Comments:		

CR-PDU 类型声明如表 3-75 所示。

表 3-75　CR-PDU 类型声明

PDU Type Definition			
PDU Name: CR-PDU PCO Type: N-SAP Encoding Rule Name: Encoding Variation: Comments: This is the type definition of the CR-PDU			
Field Name	Field Type	Field Encoding	Comments
Type	OCTESTRING[1]		
Dst-ref	BITSTRING[4]		
Src-ref	BITSTRING[4]		
Variable-part	VARIABLE-PART		Reference to structured type
User-data	IA5STRING[0…32]		
Detail Comments:			

CC-PDU 类型声明如表 3-76 所示。

表 3-76 CC-PDU 类型声明

PDU Type Definition			
PDU Name: CC-PDU PCO Type: N-SAP Encoding Rule Name: Encoding Variation: Comments: This is the type definition of the CC-PDU			
Field Name	Field Type	Field Encoding	Comments
Type	OCTESTRING[1]		
Dst-ref	BITSTRING[4]		
Src-ref	BITSTRING[4]		
Variable-part	VARIABLE-PART		Reference to structured type
User-data	IA5STRING[0…32]		
Detail Comments:			

DT-PDU 类型声明如表 3-77 所示。

表 3-77 DT-PDU 类型声明

PDU Type Definition			
PDU Name: DT-PDU PCO Type: N-SAP Encoding Rule Name: Encoding Variation: Comments: This is the type definition of the DT-PDU			
Field Name	Field Type	Field Encoding	Comments
Type	OCTESTRING[1]		
User-data	IA5string		
Detail Comments:			

CM 类型声明如表 3-78 所示。

表 3-78 CM 类型声明

CM Type Definition		
CM Name: PTC-RESULT Comments: Coordination message to transfer preliminary result from the lower		
Parameter Name	Parameter Type	Comments
Result	RESULT-TYPE	User definition type
Detail Comments:		

别名类型声明如表 3-79 所示。

表 3-79 别名类型声明

Alias Definitions		
Alias Name	Expansion	Comments
CR	N_DATArequest	Alias for the N_DATArequest service primitive used to carry a CR_PDU
DATAout	N_DATArequest	Alias for the N_DATArequest service primitive used to carry an out going DT_PDU
CC	N_DATAindication	Alias for the N_DATArequest service primitive used to carry an out going CC_PDU
DATAin	N_DATAindication	Alias for the N_DATAindication service primitive used to carry an out going DT_PDU
DR	N_DATArequest	Alias for the N_DATArequest service primitive used to carry an out going DR_PDU
Detailed Comments:		

3．约束部分

Variable part-CR1 声明如表 3-80 所示。

表 3-80　Variable part-CR1 声明

Structure Constraint Declaration			
Constraint Name: Variable part-CR1 Structure Type: VARIABLE-PART Derivation Path: Encoding Rule Name: Encoding Variation: Comments A Constraint on structure type VARIABLE-PART for the CR-PDU			
Element Name	Element value	Element Encoding	Comments
ParamA-id	-		
ParamA	-		
ParamB-id	'01'B		
ParamB	True		
Detail Comments:			

Variable part-CR2 约束声明如表 3-81 所示。

表 3-81　Variable part-CR2 约束声明

Structure Constraint Declaration			
Constraint Name: Variable part-CR2 Structure Type: VARIABLE-PART Derivation Path: Encoding Rule Name: Encoding Variation: Comments A Constraint on structure type VARIABLE-PART for the CC-PDU			
Element Name	Element value	Element Encoding	Comments
ParamA-id	'01'B if present		Accept if present
ParamA	*		Any value or none
ParamB-id	'01'B if present		Accept if present
ParamB	*		Any value or none
Detail Comments:			

NDr 约束声明如表 3-82 所示。

表 3-82　NDr 约束声明

ASP Constraint Declaration		
Constraint Name: NDr（any_pdu:PDU） ASP Type: N_DATArequest Derivation Path: Comments: A constraint on the N_DATArequest ASP.		
Parameter Name	Parameter Value	Comments
User_data	Any_pdu	The actual PDU that is carried in the ASP is dynamically chained from the constraints reference
Detailed Comments:		

NDi 约束声明如表 3-83 所示。

表 3-83 NDi 约束声明

ASP Constraint Declaration		
Constraint Name: NDi（any_pdu:PDU）		
ASP Type: N_DATAindication		
Derivation Path:		
Comments: A constraint on the N_DATAindication ASP it has a single parameter used to carry user data.		
Parameter Name	Parameter Value	Comments
User_data	Any_pdu	The actual PDU that is carried in the ASP is dynamically chained from the constraints reference
Detailed Comments:		

CONind 约束声明如表 3-84 所示。

表 3-84 CONind 约束声明

ASP Constraint Declaration		
Constraint Name: CONind		
ASP Type: X_CONNECTindication		
Derivation Path:		
Comments: A constraint on the X_CONNECTindication ASP.		
Parameter Name	Parameter Value	Comments
Called_address	Ut_address	From test suite parameters
Calling_address	Lt_address	From test suite parameters
User_data	*	Accept any value or none
Detailed Comments:		

CONrep 约束声明如表 3-85 所示。

表 3-85 CONrep 约束声明

ASP Constraint Declaration		
Constraint Name: CONrsp		
ASP Type: X_CONNECTresponse		
Derivation Path:		
Comments: A constraint on the X_CONNECTresponse ASP.		
Parameter Name	Parameter Value	Comments
Called_address	Ut_address	From test suite parameters
Calling_address	Lt_address	From test suite parameters
User_data	*	Omit Optional user data
Detailed Comments:		

DATreq 约束声明如表 3-86 所示。

表 3-86 DATreq 约束声明

ASP Constraint Declaration		
Constraint Name: DATreq（any_pdu:PDU）		
ASP Type: X_DATArequest		
Derivation Path:		
Comments: A constraint on the X_DATArequest ASP.		
Parameter Name	Parameter Value	Comments
User-data	Any_ pdu	
Detailed Comments:		

DATind 约束声明如表 3-87 所示。

表 3-87　DATind 约束声明

ASP Constraint Declaration		
Constraint Name: DATind（any_pdu:PDU）		
ASP Type: X_DATAindication		
Derivation Path:		
Comments: A constraint on the X_DATAindication ASP.		
Parameter Name	Parameter Value	Comments
User-data	Any_pdu	
Detailed Comments:		

CR1 约束声明如表 3-88 所示。

表 3-88　CR1 约束声明

PDU Constraint Declaration			
Constraint Name:CR1			
PDU Type: CR-PDU			
Derivation Path:			
Encoding Rule Name:			
Encoding Variation:			
Comments A Constraint on the CR-PDU			
Field Name	Field value	Field Encoding	Comments
Type	'F1'o		
Dst-ref	'0001'B		
Src-ref	'0001'B		
Variable-part	Variable part-CR1		Reference to a structured constraint
User-data	'Hello'		
Detail Comments:			

CC1 约束声明如表 3-89 所示。

表 3-89　CC1 约束声明

PDU Constraint Declaration			
Constraint Name:CC1			
PDU Type: CC-PDU			
Derivation Path:			
Encoding Rule Name:			
Encoding Variation:			
Comments A Constraint on the CC-PDU			
Field Name	Filed value	Field Encoding	Comments
Type	'F2'o		
Dst-ref	'0001'B		
Src-ref	'0001'B		
Variable-part	Variable part-CR2		Reference to a structured constraint
User-data	*		
Detail Comments:			

DT1 约束声明如表 3-90 所示。

表 3-90 DT1 约束声明

PDU Constraint Declaration			
Constraint Name: DT1 PDU Type: DT-PDU Derivation Path: Encoding Rule Name: Encoding Variation: Comments A Constraint on the DT-PDU			
Field Name	Field value	Field Encoding	Comments
Type	'F3'o		
User-data	Actual_data		The actual data is passed as a parameter to the constraint
Detail Comments:			

CM 约束定义如表 3-91 所示。

表 3-91 CM 约束定义

CM Constraint Declaration		
Constraint Name: PTC_RES{actual result: RESULT_TYPE} Derivation Path: PTC_RESULT Comments: A constraint on the PTC_RESULT coordination message		
Parameter Name	Parameter value	Comments
result	Actual result	Actual result is passed as to parameter to constraint
Detail Comments:		

4．动态部分

SP_DATA_TRANSFER 定义如表 3-92 所示。

表 3-92 SP_DATA_TRANSFER 定义

Test Case Dynamic Behaviour					
Test Case Name: SP_DATA_TRANSFER Group: SINGLE/DATA/ Purpose: IUT shall receive and send with time out limit, a given number of times over of two simultaneous Configuration: Single_Party Default: T_Default Comments: This test case create the other PTCs in the configuration necessary for a Single_connection configuration. Selection Ref: Description: Data transfer single_connetion					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		Create（LOWER_TEST:LTS（L1,CP1） UPPER_TEST: UTS（U1）			1）
2		CP1?PTC_RESULT	PTC_RES（pass）		2）3）
3		CP1?PTC_RESULT	PTC_RES（fail）		2）3）
Detail Recomments: 1）The create command buid a pair of Tester LT and UP 2）The preliminary result from lower PTC are picked up here 3）fail verdict					

MP_DATA_TRANSFER 定义如表 3-93 所示。

LTS 测试步定义如表 3-94 所示。

ESTABLISH_CONNECTION 测试步定义如表 3-95 所示。

表 3-93　MP_DATA_TRANSFER 定义

Test Case Dynamic Behaviour					
Test Case Name: MP_DATA_TRANSFER Group: MULT/DATA/ Purpose: IUT shall receive and send with time out limit, a given number of times over of two simultaneous Configuration: Multi_Party Default: T_Default Comments: This test case create the other PTCs in the configuration necessary for a Two_connection configuration. Selection Ref: Description: Data transfer multi_connetion					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		Create（LOWER_TEST:LTS（L1,CP1） LOWER_TEST: UTS（L2,CP2） UPPER_TEST: UTS（U1） UPPER_TEST: UTS（U2））			1）
2		CP1?PTC_RESULT	PTC_RES（pass）		2）
3		CP2?PTC_RESULT	PTC_RES（pass）	PASS	2）3）
4		CP2?PTC_RESULT	PTC_RES（fail）	FAIL	2）3）
5		CP1?PTC_RESULT	PTC_RES（fail）		2）
6		CP2?PTC_RESULT	PTC_RES（pass）	FAIL	2）3）
7		CP2?PTC_RESULT	PTC_RES（fail）	FAIL	2）3）
Detail Recomments: 1）The create command buid two PTCs 2）The preliminary result from lower PTC are picked up here 3）fail verdict					

表 3-94　LTS 测试步定义

Test Step Dynamic Behaviour					
Test Step Name:　LTS（L:N_SAP;CPP:CP） Group: TEST_STEP_LIB/LOWER/ Objective: IUT shall receive and send a data within time limit Default: Comments: Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
	LAB	+ESTABLISH_CONNECTION（L）			
		+LT_DATA_TRANSFER（L,CCP）			
		+CLOSE_CONNECTION（L）			
Detail Comments:					

表 3-95　ESTABLISH_CONNECTION 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: ESTABLISH_CONNECTION（L:N_SAP） Group: TEST_STEP_LIB/LOWER/ Objective: To establish a connection. Default: LT_DEFAULT（L） Comments: This is apreamble test step used by the lower tester（s） to set up a connection between the lower tester and the upper tester. For the sake of simplicity we shall assume that the connection cannot be refused. Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		L!CR	NDr（CR1）		
2		L?CC	Ndi（CC1）		
Detail Comments:					

LT_DATA_TRANSFER 测试步定义如表 3-96 所示。

表 3-96 LT_DATA_TRANSFER 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: LT_DATA_TRANSFER(L:N_SAP;CPP:CP) Group: TEST_STEP_LIB/LOWER/ Objective: Test data transfer Default: LT_DEFAULT(L) Comments: This test step implement test body of our example test case on the lower tester side. Description:					
Nr	Label	Behaviour Description	Constraint Ref	V	C
1	LAB	L!DATAout(count:=INC(count))	NDr(DT1(data_string))		
2		[count<=max] START Timer			
3		L?DATAin	Ndi(DT1(data_string))	Pass	
4		->LAB			
5		?TIMEOUTtimer		fail	
6		CCP!PTC_RESULT	PTC_RES(pass)		
7		CPP!PTC_RESULT	PTC_RES(fail)		
Detail Comments					

CLOSE_CONNECTION 测试步定义如表 3-97 所示。

表 3-97 CLOSE_CONNECTION 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: CLOSE_CONNECTION(L:N_SAP) Group: TEST_STEP_LIB/LOWER/ Objective: Close the connection to the IUT. Default: Comments: This is Postamble test that close a connection between the lower tester and the upper tester. Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		L!DR	NDr(CR1)		
Detail Comments:					

UTS 测试步定义如表 3-98 所示。

表 3-98 UTS 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: UTS(U:X_SAP) Group: TEST_STEP_LIB/UPPER/ Objective: Accept connection and receive/send DATA Acertain number of times. Default: Comments: Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		+ACCEPT_CONNECTION(U)			
2		+UT_DATA_TRANSFER(U)			
Detail Comments:					

ACCEPT_CONNECTION 测试步定义如表 3-99 所示。

UT_DATA_TRANSFER 测试步定义如表 3-100 所示。

UT_DEFAULT 测试步定义如表 3-102 所示。

LT_DEFAULT 测试步定义如表 3-101 所示。

表 3-99　ACCEPT_CONNECTION 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: ACCEPT_CONNECTION（U:X_SAP） Group: TEST_STEP_LIB/UPPER/ Objective: Accept a X_CNNECTindication from lower tester. Default: Comments: This is a preamble test step used by upper tester to accept an incoming connection request from lower test. Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		U?X_CONNECTindication	CONind		
2		U!X_CONNECTresponse	CONres		
Detail Comments:					

表 3-100　UT_DATA_TRANSFER 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: UT_DATA_TRANSFER（U:X_SAP） Group: TEST_STEP_LIB/UPPER/ Objective: Respond to incoming data. Default: UT_DEFAULT（U） Comments: This is test step implements the body of our example test case on the upper tester side. Description:					
Nr	Label	Behaviour Description	Constraint Ref	V	C
1	LAB	U?X_DATAindication（count:= INC（count））	DATind（DT1（data_string））		
2		U!X_DATArequest[count:=max]	DATreq（DT1（data_string））		
3		->LAB			
Detail Comments:					

表 3-101　LT_DEFAULT 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: LT_DEFAULT（L:N_SAP） Group: DEFAULT_LIB Objective: General catch all for lowser tester Default: Comments: Description:					
Nr	Label	Behaviour Description	Constraint Ref	V	Comments
1		L?OTHERWISE		FAIL	The test stop immediately
Detail Comments:					

表 3-102　UT_DEFAULT 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: UT_DEFAULT（U:X_SAP） Group: DEFAULT_LIB Objective: General catch all for upper tester Default: Comments: Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		U?OTHERWISE		FAIL	The test stop immediately
Detail Comments:					

T_DEFAULT 测试步定义如表 3-103 所示。

表 3-103 T_DEFAULT 测试步定义

Test Step Dynamic Behaviour					
Test Step Name: T_DEFAULT Group: DEFAULT_LIB Objective: General catch all Default: Comments: Description:					
Nr	Label	Behaviour Description	Constraint Ref	Verdict	Comments
1		L1?OTHERWISE		FAIL	
2		L2 OTHERWISE		FAIL	
3		U1 OTHERWISE		FAIL	
4		U2 OTHERWISE		FAIL	
Detail Comments:					

思考题

1．试说明 TTCN 一致性测试框架中 PCO 的含义。为什么要保持两个队列？

2．在 TTCN 的通信模型中，接收值的含义是什么？

3．给出与 COMPLEMENT（'00'B,'11'B）匹配的所有值。

4．试指出“ab?z”与“abcz”、“abdz”、“ab_z”、“abz”中哪一个接收值不匹配。

5．试说明扩展的接收语句 RECEIVE_1 MATCH_2 $[\text{QUALIFIER}]_3$ $[\text{ASSIGNMENT\_LIST}]_4$ $[\text{TIMER\_OPERATION}]_5$ 中 MATCH 的含义。

6．如果一个结果变量 R 中记录的初步结果是 fail，而裁决列中返回一个 pass 结果，则 R 中的记录将是什么？

7．已知 Transport Protocol Class 0 的 MSC 图如图 3-10 所示，试给出该图所对应的 TTCN 测试套的 DYNAMIC PART 描述。

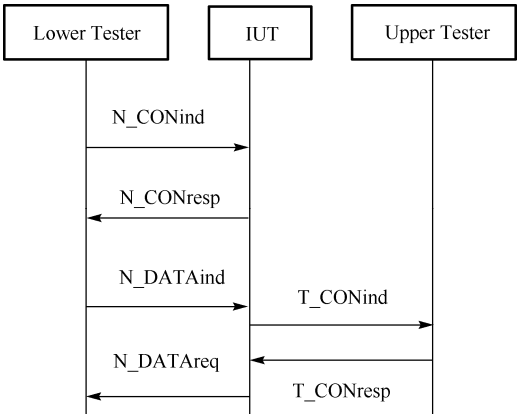


图 3-10 MSC 图

第 4 章　TTCN-3 基本语言元素

本章介绍 TTCN-3 核心语言的基本概念，通过学习可以掌握 TTCN-3 核心语言的语法，为实际从事测试工作奠定理论基础。在全面介绍 TTCN-3 之前，首先介绍 TTCN-3 的基本语言元素，明确各种符号的含义。由于 TTCN-3 核心语言与数表形式语言的基本语言元素有许多相似之处，所以本章重点介绍核心语言的特色，对二者相似的地方不做重点讲解。

4.1　TTCN-3 概述

4.1.1　实例

给出一个完整的 TTCN-3 核心语言的例子，以便读者可以了解 TTCN-3 语言的全貌。

例 1：一个 TTCN-3 测试套实例

```
module Module {
    type record urlType {
        charstring      protocol,
        charstring      host,
        charstring      file
    }
    type set of dinosaurType dinolistType;

    type record dinosaurType {
        charstring      name,
        charstring      len,
        ......
    }
    template urlType urlTemplate:= {
        protocol    := "http://",
        host        := "www.testingtech.de",
        file        := "/TTCN-3_Example/dinolist.xml"
    }
    template dinolistType DinoListTemplate:= {?, ?, BrachiosaurusTemplate, ?, ?, ?, ?};

    template dinosaurType BrachiosaurusTemplate:= {
        name        := "Brachiosaurus",
        len         := ?,
        ......
    }
    type port httpPortType message {
        out urlType;
        in  dinolistType;
```

```
    }
    type component ptcType
{
port httpPortType httpPort;
        timer localTimer:= 3.0;
    }
    type component systemType
{
        port httpPortType  httpPortArray;
    }
    type component mtcType {}
    function ptcBehaviour() runs on ptcType
{
        httpPort.send(urlTemplate);
        localTimer.start;
        alt
        {
             [] httpPort.receive(DinoListTemplate)
             {
                 localTimer.stop;
                 setverdict(pass);
             }
             [] httpPort.receive
             {
                 localTimer.stop;
                 setverdict(fail);
             }
             [] localTimer.timeout
             {
                 setverdict(fail);
             }
         }
     }
    testcase DinoListTest_1() runs on mtcType system systemType
    {
        var ptcType ptcArray;
        ptcArray:= ptcType.create;
        map (ptcArray:httpPort, system:httpPortArray);
        ptcArray.start(ptcBehaviour());
        ptcArray.done;
    }
}
```

在上面的例子中可以看出，TTCN-3 核心语言程序的基本程序设计单元是模块(**Module**)。一个模块中不能包含子模块，但是它可以从其他模块中引入定义。模块可以带有参数列表去提供测试套参数化的一个形式，这与 TTCN-3 数表形式中的 PICS 和 PIXIT 参数化机制相似。

一个模块由一个定义部分和一个控制部分组成。模块的定义部分定义测试成分、通信端口、数据类型、常数、测试数据模板、函数、测试端口上调用的过程信号(**signatures**)、测试用例等。

模块的(控制部分)调用测试用例并控制它们的执行。控制部分也可以声明(局部)变量等，程序语句(如 if-else 和 do-while)可以用于各个测试用例的选择和执行顺序。TTCN-3 不支持全局变量的概念。

与计算机高级程序设计语言类似，TTCN-3 核心语言也有许多预定义的基本数据类型和结构类型，如记录(**records**)、集合(**sets**)、联合(**unions**)、枚举(**enumerated**)类型和数组。引入的 ASN.1 类型和值可以与 TTCN-3 一起使用。

模板是一种特殊的数据结构，它为描述在测试端口上被发送和接收的测试数据提供参数化和匹配机制。在这些通信端口上的操作提供基于消息和基于过程的通信能力，过程调用可以用于非基于消息的测试实现。

测试用例表达动态测试行为，TTCN-3 核心程序语句包括强有力的行为描述机制，如通信和定时器事件的选择性接收。TTCN-3 也支持测试判定赋值和日志机制。

最后，可以给 TTCN-3 语言元素赋予属性，如编码信息和显示属性，同样也可以描述(非标准化的)用户定义的属性。

通常来说，声明的顺序是任意的，它可以用在一个语句和声明块中，如函数体或一个 if-else 语句分支中。但从程序设计角度，所有的声明(如果有的话)应该仅在该块的开始处进行声明。

例 2：//这是一个 TTCN-3 声明

```
    :
    var MyVarType MyVar2:= 3;
        const integer MyConst:= 1;
    if (x > 10)
    {
        var integer MyVar1:= 1;
        :
        MyVar1:= MyVar1 + 10;
        :
    }
:
```

模块定义部分中的定义可以按任何顺序进行，但考虑到可读性因素，应该避免向后(程序没有被说明的语言元素)引用，不过这并不是必需的。例如，调用其他函数和模块的参数化的递归元素就可能导致向后引用。

仅对模块定义中声明允许的向后引用，应该不在模块的控制部分、测试用例的定义部分、函数和选择步中使用向后引用。这就意味着对局部变量、局部定时器和局部常量的向后引用绝不会发生。而这个规则的唯一例外是标签，我们可以在 **goto** 语句中使用用于标签的向后引用来跳转到后面。

TTCN-3 语言元素一览表如表 4-1 所示。

表 4-1　TTCN-3 语言元素一览表

语言元素	相关联的关键字	是否在模块定义中被描述	是否在模块控制中被描述	是否在函数/可选步/测试用例中被描述	是否在测试成分中被描述
TTCN-3 模块定义	**module**				
其他模块的定义引入	**import**	是			
组定义	**group**	是			

续表

语言元素	相关联的关键字	是否在模块定义中被描述	是否在模块控制中被描述	是否在函数/可选步/测试用例中被描述	是否在测试成分中被描述
数据类型定义	**type**	是			
通信端口定义	**port**	是			
测试成分定义	**component**	是			
特征定义	**signature**	是			
外部函数/常量定义	**external**	是			
常量定义	**const**	是	是	是	是
数据/特征模板定义	**template**	是			
函数定义	**function**	是			
可选步定义	**altstep**	是			
测试用例定义	**testcase**	是			
变量声明	**var**		是	是	是
定时器声明	**timer**		是	是	是

4.1.2 范围规则

TTCN-3 核心语言提供以下 6 个基本的范围单位。

(1) 模块定义部分。

(2) 模块的控制部分。

(3) 成分类型。

(4) 函数。

(5) 可选步 (altsteps)。

(6) 测试用例。

(7) 复合语句中的“声明和语句块”。

注意 1：用于组 (**groups**) 的附加范围规则在 4.3 节中给出。

注意 2：用于 **for** 循环中计数的附加范围规则在 4.17 节中给出。

范围由声明 (可选的) 组成。每个单位可以通过使用 TTCN-3 语言语句和操作，用范围单位——模块的控制部分、函数、测试用例、可选步和复合语句中的“声明和语句块”来额外描述行为的某种形式。

模块定义部分全局可见，也就是说它们可以用在模块的任意位置，包括该模块定义的所有函数、测试用例和可选步以及控制部分。从其他模块中引入的标识符对于引入模块来说也是全局可见的。

模块控制部分中的定义具有局部可见性，即只能用在该控制部分中。

通过使用一个 **runs on** 句，定义的元素在测试成分中定义仅可以在调用该子类的成分中可见，例如测试例、可选步等。

函数、测试用例和可选步是独立的范围单位，它们之间没有层次关系，即它们主体开始处做的声明具有局部可见性，且仅可以用在给定的函数、测试用例和可选步中 (例如，在一个测试用例中做的声明对于被这个测试用例调用的函数或被该测试用例使用的可选步来说是不可见的)。

复合语句包括“语句和声明块”，如 **if-else**-、**while**-、**do-while**-或 **alt**-语句。它们可以用在一个模块的控制部分、测试用例、可选步、函数中，或嵌套在其他复合语句中。例如，在一个 **while**-循环中使用 **if-else**-语句。

复合语句和嵌套式复合语句中的“语句和声明块”对于包括给定“语句和声明块”的范围单

位和任意嵌套的“语句和声明块”来说具有层次关系，在一个“语句和声明块”中所做的定义具有局部可见性。

范围单位的层次关系如图 4-1 所示。高层的范围单位声明对其所在层次关系中的同一分支中下面层次的所有单位来说是可见的，而层次关系中低层的范围单位声明对于其上层的那些单位来说是不可见的。

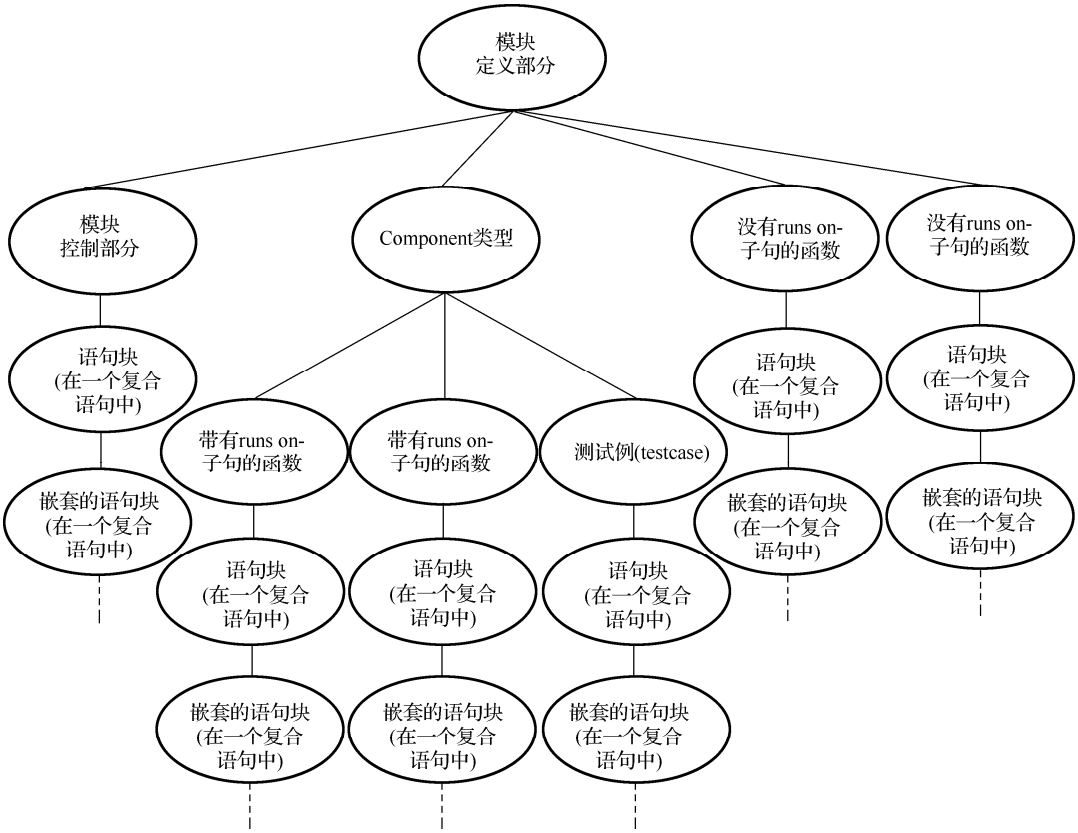


图 4-1　范围单位的层次关系

例 1：

```
module MyModule
{       :
    const   integer MyConst:= 0;
            //对于 MyBehaviourA 和 MyBehaviourB 来说，MyConst 是可见的
        :
    function MyBehaviourA()
    {           :
        const   integer A:= 1;      //常量 A 仅对 MyBehaviourA 是可见的
                :
    }
    function MyBehaviourB()
    {
            :
        const  integer B:= 1;       //常量 B 仅对 MyBehaviourB 是可见的
```

```
        :
        }
    }
```

在一个参数化的语言元素中(如在一个函数调用中)，形参的范围应该应该限定到这些参数出现的定义中以及相同层次关系中的较低的范围层次，这就是说它们遵循正常的范围规则。TTCN-3核心语言要求标识符具有唯一性，即在相同范围层次中的所有标识符互不相同。这就意味着在同一个范围层次的分支中，低层范围中的声明不应该重复使用与高层范围声明中相同的标识符。结构类型字段、枚举值和组的标识符不必全局唯一，然而在枚举值的情况下，标识符应该仅被其他枚举类型中的枚举值重复使用。标识符唯一性规则应该也用于形参标识符。

例 2：

```
module MyModule
{   :
    const integer A:= 1;
    :
    function MyBehaviourA()
    {   :
        const   integer A:= 1;              //不允许
        :
        if(…)
        {   :
            const   boolean A:= true;   //不允许
            :
        }
    }
}
    //下面不在相同范围层次中声明的常量是允许的(假设在模块的头部没有 A 的声明)
function MyBehaviourA()
{
    :
    const   integer A:= 1;
    :
}
function MyBehaviourB()
{
    :
    const   integer A:= 1;
    :
}
```

TTCN-3 标识符对大小写敏感，关键字应该小写。TTCN-3 的关键字既不能作为 TTCN-3 对象的标识符，也不能作为从其他语言的模块中引入对象的标识符。

4.1.3 参数化

TTCN-3 根据以下限制，支持值(**value**)的参数化。

(1)不能参数化的语言元素有 const、var、timer、control、group 和 import。

(2)语言元素模块(**module**)允许静态的值参数化去支持测试套参数。也就是说，在编译时这个参数化既可以是可解析的，也可以是不可解析的，但是它应该在运行开始时被解析(即在运行时是静态的)。这就意味着在运行时，模块的参数值是全局可见的，但是不能改变。

(3)所有用户定义类型定义(包括结构化的类型定义，如 record、set 等)和特殊的配置类型如 address(这个参数化应该在编译时进行解析)。

(4)语言元素 template、signature、testcase、altstep 和 function 支持动态的值参数化(即这个参数化过程应该在开始运行时进行)。

参数化的语言元素一览表如表 4-2 所示。

表 4-2　TTCN 参数化的语言元素一览表

关键字	值参数化	在形参/实参列表中允许出现的值的类型
Module	在运行开始时，静态	所有基本类型、所有用户自定义类型和地址类型(address)的值
type(note)	在编译时，静态	所有基本类型、所有用户自定义类型和地址(address)的值
Template	在运行时，动态	所有基本类型、所有用户自定义类型、地址类型(address)和模板类型(template)的值
Function	在运行时，动态	所有基本类型、所有用户自定义类型、地址(address)、成分(component)端口(port)、默认(default)、模板(template)和定时器(timer)类型的值
Altstep	在运行时，动态	所有基本类型、所有用户自定义类型、地址(address)、成分(component)端口(port)、默认(default)、模板(template)和定时器(timer)类型的值
Testcase	在运行时，动态	所有基本类型、所有用户自定义类型、地址类型(address)和模板类型(template)的值
Signature	在运行时，动态	所有基本类型、所有用户自定义类型、地址类型(address)和成分类型(component)的值
注意 1：record of、set of、enumerated、port、component 和 subtype 类型定义不允许参数化 **注意 2**：不同语言元素中参数化的例子和特殊用法在本文的相关章节中给出		

在参数传递过程中，基本类型、基本串类型、用户定义的结构类型、地址类型和成分类型通过传值来传递所有的实际参数。可以选择使用关键字 **in** 来表示传值。如果使用传参的方法传递以上提到类型的参数，则应该用 **out** 或 **inout**。

定时器和端口类型参数总是通过传参的方法进行参数传递，并通过关键字 **timer** 和 **port** 来标识。可以选择性地使用关键字 **inout** 来表示使用传参的方法进行参数传递。

使用传参的方法来传递参数有以下限制。

(1)只对 **altsteps** 的形参列表进行显式调用，**functions**、**signatures** 和 **testcase** 可以包含传参的参数(pass-by-reference parameters)。

注意：对如何在过程信号中使用传参参数有进一步的限制。

(2)实际参数应仅是变量(例如：不是常量或模板)。

例 1：

```
function MyFunction(inout boolean MyReferenceParameter){ … };
//通过传参来传递 MyReferenceParameter，且可以在该函数中读出和设置该实参
function MyFunction(out Boolean MyReferenceParameter){ … };
//通过传参来传递 MyReferenceParameter，且仅可以在该函数中设置该实参
传值参数可以是变量，也可以是常量、模板等
function MyFunction(in template MyTemplateType MyValueParameter){ … };
//通过传值来传递 MyValueParameter，关键字 in 是可选的
```

在实参列表中出现的语言元素的数目和它们顺序应与其相应的形参列表中的元素数目和出现顺序相同。而且，每个实参的类型应该与相应的形参的类型是兼容的。

例 2：

```
//带有形参列表的一个函数定义
function MyFunction(integer FormalPar1, boolean FormalPar2, bitstring
        FormalPar3) { … }
//带有实参列表的一个函数调用
MyFunction(123, true,'1100'B);
```

如果 TTCN-3 语言元素 **function testcase**、**signature**、**altstep** 或 **external function** 的形参列表是空的，那么在该元素的声明和调用时都应该包含这个空的括号。在所有其他的情况下，这个空的括号是可以省略的。

例 3：

```
//带有空参数列表的函数定义可以写为
function MyFunction(){ … }
//带有空参数列表的记录定义可以写为
type record MyRecord { … }
```

通常，被描述为一个实参的所有参数化实体应该在实参列表中解析它们自己的参数。

例 4：

```
//给定的消息定义
type record MyMessageType
{
    Integer field1,
    Charstring field2,
    boolean field3
}
//一个消息模板可以是
template MyMessageType MyTemplate(integer MyValue):=
{
    field1:=MyValue,
    field2:=pattern"abc*xyz",
    field3:=true
}
//带有一个模板参数的测试用例可以是
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1
{   :
    MyPCO.receive(RxMsg);
}   //当测试用例在控制部分中被调用且该参数化模板用作一个实参时，必须提供该模板的实参
control
{   :
    TC001(MyTemplate(7));
    :
}
```

4.2　数据类型和值

TTCN-3 支持许多预定义的基本类型。这些基本类型包括与程序语言正常关联的基本类型，如整形(**integer**)、布尔类型(**boolean**)和串类型，也包括一些 TTCN-3 特殊的类型，如对象标识类型(**objid**)和判定类型(**verdicttype**)。可以从这些基本类型中构造新的类型，如记录类型(**record**)、集合类型(**set**)和枚举类型(**enumerated**)。

特殊的数据类型——anytype 类型定义为一个模块中所有已知类型的联合(union)。与测试配置相关的特殊类型，如地址类型(**address**)、端口类型(**port**)和成分类型(**component**)可以用来定义测试系统的体系结构。特殊类型 default 类型可以用于默认处理。

TTCN-3 类型一览如表 4-3 所示。

表 4-3　TTCN-3 类型一览

类型分类	关键字	子类型
简单基本类型	**integer**	range, list
	char	range, list
	universal char	range, list
	float	range, list
	boolean	list
	objid	list
	verdicttype	list
基本串类型	**bitstring**	list, length
	hexstring	list, length
	octetstring	list, length
	charstring	range, list, length
	universal charstring	range, list, length
结构类型	**record**	list
	record of	list, length
	set	list
	set of	list, length
	enumerated	list
	union	list
特殊的数据类型	**anytype**	list
特殊的配置类型	**address**	
	port	
	component	
特殊的默认类型	**default**	

4.2.1　基本类型和值

TTCN-3 支持下列基本类型。

(1)**integer**：整型，其值为所有的正、负整数和零。整型值应该用一个或多个数字表示，且除 0 值外其第一位不应该是 0，而 0 应该由一位数字表示(即单个 0)。

(2)**char**：字符型，其值为与 ISO/IEC 646 [5]的 8.2 节中描述的国际参考版本 International Reference Version，IRV)相符的 ISO/IEC 646 [5] 版本中的字符。

注意 1：ISO/IEC 646 [5]中的 IRV 版本与 ITU-T Recommendation T.50（见参考书目）中描述的国际参考字母（International Reference Alphabet，以前是 International Alphabet No.5 IA5）的 IRV 版本等价。字符类型的值可以用双引号（"）括起来给出，或者使用预定义的带有编码参数的转换函数计算获得。相关的操作符号相等（==）和不相等（!=）可以用来比较 **char** 类型的值。

（3）**universal char**：通用字符类型，其值为来自 ISO/IEC 10646 [6]的单个字符。**universal char** 类型的值可以用双引号（"）括起来给出，或者使用预定义的带有编码参数的转换函数计算获得或由一个四元组（quadruple）给出。 这个四元组仅能表示一个单个字符，并且它表示一个字符是根据 ISO/IEC 10646 [6]使用该字符的组（**group**）、容器（**plane**）、行（**row**）和单元格（**cell**）的十进制数来表示，由关键字 **char** 来引导，带有一对括号，并用逗号分隔（例如，char（0, 0, 1, 113）表示匈牙利字符“ű”）。

注意 2：控制字符仅可以使用四元组的格式表示。默认情况下，universal char 应该与 ISO/IEC 10646 [6]的 14.2 节中描述的 UCS-4 编码表示格式相一致。这个默认的编码可以通过使用定义的编码属性来对其进行重写（**override**）。

注意 3：UCS-4 是一种编码格式，它使用一个固定的、32 位长的字段来表示任意 UCS 字符。相关的操作符号相等（==）和不相等（!=）可以用来比较 universal char 类型的值。

（4）**float**：浮点类型，描述浮点数的一个类型。浮点数表示为：<尾数> × <基数><指数>。其中，<尾数>是一个正或负整数， <基数>是一个正整数（多数情况为 2、10 或 16），<指数>为一个正或负整数。浮点数的表示限定为以值 10 为基数，浮点值可以使用下列任一方式表示：在一个数字序列中使用小数点的正常表示，如 1.23（表示 123*10-2），2.783（即 2783 × 10-3），或 -123.456789（表示-123456789 × 10-6）；或者使用 E 来分开两个数字来表示，前一个数字描述尾数，第二个数字描述指数，例如 12.3E4（表示 12.3 × 104），-12.3E-4（表示-12.3 × 10-4）。

（5）**boolean**：布尔类型，该类型有两个不同值。布尔类型的值应使用 true 和 false 来表示。

（6）**objid**：对象标识类型，其值为与 ITU-T Recommendation X.660 的所有对象标识符的集合一致，标识符中的连字符被替换为下划线。

例 1：

```
{itu_t(0) identified_organization(4) etsi(0)}
or alternatively {itu_t identified_organization etsi}
or alternatively { 0 4 0}
```

（7）**verdicttype**：判定类型，该类型有五个不同的值。Verdicttype 类型的值使用 pass、fail、inconc、none 和 error 表示。

除上述基本类型外，TTCN-3 还支持下列基本串类型。

注意 4：TTCN-3 中的通用术语“串”或“串类型”指的是比特串（**bitstring**）、十六进制串（**hexstring**）、八位组串（**octetstring**）、字符串（**charstring**）和通用字符串（**universal charstring**）。

（8）**bitstring**：比特串类型，其值为 0 位、1 位或多位的 0、1 序列。bitstring 类型值应该用任意数目的比特数 0、1 来表示（可能为 0 位），以字符“'”开始，字符“'B”结束。

例 1：'01101'B。

（9）**hexstring**：十六进制串类型，其值为 0 位、1 位或多位十六进制数的有序序列，每个十六进制数与一个有序的四比特序列相符。hexstring 类型值应该用任意数目的十六进制数来表示（可能为 0 位）： 0 1 2 3 4 5 6 7 8 9 A B C D E F。以字符“'”开始，后接字符“'H”；使用十六进制表示每个十六进制数用于表示半个八位组的值。

例 2：'AB01D'H。

10）**octetstring**：八位组串类型，0 个或正偶数个十六进制数的有序序列（每对数字有一个有序的八比特序列与之对应）。octetstring 类型值用任意数字但必须用偶数数目的十六进制数来表示（可能为 0 位）：0 1 2 3 4 5 6 7 8 9 A B C D E F。以字符“'”开始，后接字符“'O”；使用十六进制表示，每个十六进制数字用于表示半个八位组的值。

例 3：'FF96'O。

11）**charstring**：字符串类型，其值为 0 个、1 个或多个与 ISO/IEC 646[5]的 8.2 节中描述的国际参考版本 International Reference Version，IRV）相符的 ISO/IEC 646 [5]版本的字符（characters of the version of ISO/IEC 646 [5]）（见注意 1）。由关键字 universal 引导的字符串类型表示类型值为来自 ISO/IEC 10646 [6]的 0 个、1 个或多个字符的类型。Charstring 类型值应该由来自相关字符集合的任意数目个字符表示，并使用双引号（"）把它括起来。在串中需要包含字符双引号（"）的情况下，在同一行中使用一对双引号来表示该双引号字符，且其间没有空格字符。

例 4：""abcd""表示文字串"abcd"。

（12）**universal charstring**：类型值也可以用来自相关字符集合的任意数目个字符表示，并使用双引号（"）把它括起来或使用一个四元组（**quadruple**）。这个四元组仅能表示一个单个字符，且它使用 ISO/IEC 10646 [6]中该字符的组（**group**）、容器（**plane**）、行（**row**）和单元格（**cell**）的十进制数来表示这个字符，由关键字 char 来引导，带有一对括号，并用逗号分隔（例如，char（0, 0, 1, 113）表示匈牙利字符“ű”）。依据第一种方法（使用一对双引号）时，在串中需要包含字符双引号（"）的情况下，在同一行中使用一对双引号来表示该双引号字符，且其间没有空格字符。在使用连接操作符（**concatenation operator**）的一个串值的表示法中，可以混合地使用两种方法。

例 2：

赋值："the Braille character" & **char**（0, 0, 40, 48）& "looks like this"表示文字串 the Braille character looks like this。

注意 5：仅可以使用四元组来表示控制字符。默认地，universal charstring 应该与 ISO/IEC 10646 [6]14.2 节中描述的 UCS-4 编码表示格式一致。

注意 6：UCS-4 是一种编码格式，它使用一个固定的、32 位长的字段来表示任意 UCS 字符。可以使用定义的编码属性（见 28.2.1 节）来重写这个默认编码。

TTCN-3 中存取单个字符可使用一个类似数组的文法来访问一个串类型中的各元素，串中仅一个元素可以被存取。表 4-4 指出了不同串类型元素的长度单位。索引应该以 0 值开始。

表 4-4　长度描述字段中使用的长度单位

类型	长度单位
bitstring	比特
hexstring	十六进制数
octetstring	八位组
character strings	字符

例 3：

```
//给定
MyBitString:= '11110111'B;
//然后做
MyBitString[4]:= '1'B;
//以比特串表示的结果为 '11111111'B
```

4.2.2　基本类型的子类型

用户定义类型用关键字 type 表示，可以根据表 4-3 中的简单基本类型定义用户数据类型，也可以在简单串类型上创建子基础类型（如列表（**lists**）、范围（**ranges**）和长度（**length**）限制）。

例 1：

```
        type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
        type float pi (3.1415926);
        type universal char SpecialLetter (char(0, 0, 1, 111), char(0, 0, 1, 112),
char(0, 0, 1, 113));
```

TTCN-3 允许对类型 **integer**、**char**、**universal char** 和 **float** 类型(或这些类型的派生类型)值域的描述。这个值域定义的子类型限定了该子类型的值可为值域中的值，且包括该值域的上下界。

例 2：

```
        type    integer     MyIntegerRange (0 .. 255);
        type    char        MyCharRange ("a" .. "z");
        type    float       piRange (3.14 .. 3142E-3);
```

char 类型的值域描述也可以用在一个 **charstring** 子类型定义中，而 **universal char** 类型的值域描述可以用在 **universal charstring** 子类型定义中。在这样的情况下，值域为串中每个单独字符限定了允许的取值范围。

例 3：

```
        type charstring MyCharString ("a" .. "z");
        //定义了一个任意长度的串类型，且串中的每个字符都在所描述的值域之内
        type universal charstring MyUCharString1 ("a" .. "z");
        //定义了一个任意长度的串类型，且串中的每个字符都在使用双引号所描述的值域之内
        type universal charstring MyUCharString2 (char(0, 0, 1, 111) ..char(0, 0, 1, 113));
        //定义了一个任意长度的串类型，且串中的每个字符都在使用四元组所描述的值域之内
```

特别是，TTCN-3 还提供了无限值域。

为了描述一个无限的整型或浮点型的值域，可以使用关键字 **infinity** 来代替一个值来表示没有上下边界。

例 4：

```
        type integer MyIntegerRange (-infinity .. -1); //所有负整数
```

注意：无限的“值”是依赖于实现的，这个特性的使用可能会导致可移植性问题。

另外，对于类型 **integer**、**char**、**universal char** 和 **float**(或这些类型的派生类型)的值来说，也可以混合使用列表和值域。

例 5：

```
        type integer    MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);
        type char       MyCharRange ("a", "b", "c", "0" .. "9");
```

在 **charstring** 和 **universal charstring** 子类型定义中，不在相同的子类型定义中混合使用列表和值域。

关于串的长度，TTCN-3 也有相关限定。

TTCN-3 允许在串类型上对长度限定进行描述。根据使用长度边界的串类型的不同，该长度边界具有不同的复杂度。在所有的情况下，这些边界都应该为非负整型值(或是派生的整型值)。

例 6：

```
        type bitstring MyByte length(8);                    //精确的长度值 8
```

```
type bitstring MyByte length(8 .. 8);            //精确的长度值 8
type bitstring MyNibbleToByte length(4 .. 8);    //最小长度为 4，对大长度为 8
```

关键字 **infinity** 用于上界的时候表示长度没有上限。上限应该大于等于下限。

除了基本类型外，TTCN-3 里也定义了结构化类型，，如记录类型(**record**)、record of 类型、集合类型(**set**)、set of 类型、枚举类型(**enumerated**)和联合类型(**union**)。关键字 type 也可用来对结构化的类型进行描述。

先来看看结构化类型是如何定义和赋值的。可以使用一个明确的赋值表示或一个简写的值列表给出这些类型的值。

例 7：

```
const MyRecordType MyRecordValue:=  //赋值表示
{
    field1:= '11001'B,
    field2:= true,
    field3:= "A string"
}
//或者
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}
 //值列表表示
```

当使用赋值表示方法描述部分值的时候(即仅设置一个结构变量字段子集的值)，只有被赋值的字段才必须被描述。在结构中使用值列表表示时，应该使用一个值来描述所有的字段，用符号“-”或关键字 **omit** 表示不使用的字段。

例 8：

```
var MyRecordType MyVariable:=       //赋值
{
    field1:= '11001'B,
    field3:= "A string"
}
//或
var MyRecordType MyVariable:= {'11001'B, -, "A string"}  //值列表表示
```

在同一(紧接着的)上下文中，不允许混合使用这两种值表示方法。

例 9：

```
//这是不允许的
const MyRecordType MyRecordValue:= {MyIntegerValue, field2:= true, "A string"}
```

无论是在赋值表示方法还是在值列表表示方法中，都应该对可选字段使用明确的值 **omit** 来省略相关字段。省略一个字段会引起相关字段值变成未定义字段，而不管该字段以前具有什么样的值。对强制字段(**mandatory fields**)不应该使用关键字 **omit**。

下面将分别对记录类型、集合类型、枚举类型和联合类型加以介绍。

4.2.3　记录类型

TTCN-3 支持有序的结构化类型，即记录类型(**record**)。一个 **record** 类型元素可以是基本类型或用户定义数据类型(如其他记录、集合或数组类型)的任一种，一个 **record** 值应该与该 **record**

字段的类型兼容。对于 **record**，其元素标识符是该 **record** 的本地标识符，且在该 **record** 中是唯一的(但不必全局唯一)。**record** 类型的常量应该既不直接也不间接包含变量或模块参数作为字段值。

```
type    record MyRecordType
{
    integer field1,
    MyOtherRecord Typefield2 optional,
    charstring  field3
}
type    record MyOtherRecordType
{
    bitstring   field1,
    boolean field2
}
```

可以定义记录没有字段(即作为一个空记录)。

例 1：

```
type record  MyEmptyRecord {}
```

把一个记录值赋值给一个单个的元素基(element basis)。

例 2：

```
var integer MyIntegerValue:= 1;
const MyOtherRecordType MyOtherRecordValue:=
{
    field1:= '11001'B,
    field2:= true
}
var MyRecordType MyRecordValue:=
{
    field1:= MyIntegerValue,
    field2:= MyOtherRecordValue,
    field3:= "A string"
}
```

或使用一个值列表。

例 3：

```
MyRecordValue:= {MyIntegerValue, {'11001'B,true}, "A string"};
```

应该使用省略符号省略可选字段。

例 4：

```
MyRecordValue:= {MyIntegerValue,omit , "A string"};
//注意这与下面的写法不同
//MyRecordValue:= {MyIntegerValue, -, "A string"}
//后面的写法意味着 field2 的值不变
```

在引用一个 record 类型的字段时，应使用点号来对 record 类型进行引用。格式如下：类型或值标识符.元素标识符。其中，类型或值标识符用来解析一个结构化类型或变量的名字，元素标识符用来解析结构化类型中一个字段的名字。

例 5：

```
MyVar1:= MyRecord1.myElement1;
//如果一个 record 类型嵌套在另外一个类型中，那么对它的应用可以看起来像下面的格式
MyVar2:= MyRecord1.myElement1.myElement2;
```

record 类型还有可选元素，应使用关键字 optional 来描述一个 record 类型的可选元素。

例 6：

```
type    record  MyMessageType
{
    FieldType1  field1,
    FieldType2  field2   optional,
    :
    FieldTypeN field_N
}
```

4.2.4　集合类型

TTCN-3 还支持无序的结构化类型，即集合类型(**set**)。**set** 类型和值与 **record** 类型很相似，只是 **set** 类型字段的顺序是没有意义的。

例 1：

```
type    set MySetType
{
    integer field1,
    charstring field2
}
```

Set 类型字段标识符对于 **set** 类型来说是本地的，且在该 **set** 类型中应该是唯一的(但不必是全局唯一的)。

set 类型值不应使用值列表表示方法。

应使用点号对 **set** 类型元素进行引用。

例 2：

```
MyVar3:= MySet1.myElement1;
//如果一个 set 类型嵌套在另一个类型中，那么该引用可以看起来像下面的格式
MyVar4:= MyRecord1.myElement1.myElement2;
//注意，带有被引用标识符 myElement2 的 set 类型嵌套在一个 record 类型中
```

同样，应使用关键字 **optional** 来描述 **set** 类型中的可选元素。

另外，TTCN-3 还支持对所有元素为同一类型的 **record** 和 **set** 类型的描述，并使用关键字 **of** 来表示。这些记录和集合没有元素标识符，可以认为它们分别与有序和无序数组相似。

使用关键字 length 来限定 record of 和 set of 类型的长度。

例 3：

```
type record length(10) of integer MyRecordOfType;
                          //是一个记录，正好有 10 个整数
type record length(0..10) of integer MyRecordOfType;
```

```
                                    //是一个记录，最多有 10 个整数
type record length(10..infinity) of integer MyRecordOfType;
                                    //最少 10 个整数的记录类型
type set of boolean MySetOfType; //布尔值的一个无限集合
type record length(0..10) of charstring StringArray length(12);
//一个记录类型，最多有 10 个串，每个串正好 12 个字符
```

record of 和 **set of** 类型的值表示应该是一个值列表表示法或是一个对各元素进行索引的表示方法。

当使用值列表表示法时，列表中的第一个值被赋值给第一个元素，第二个值被赋值给第二个元素，依此类推。赋空值是被允许的(例如，两个逗号紧挨着或之间仅有一个空格)，应该在该列表中明确地跳过或省略赋值中要省去的元素。

索引值表示法既可以用在赋值符号的左边，也可以用在赋值符号的右边。第一个元素的索引值应为 0，且该索引值不应超出子类型设定的长度限制。如果在赋值符号右边索引表示的元素值没有被定义，那么将会导致语义或运行错误。如果在赋值符号左边的一个索引操作符引用了一个不存在的元素，那么赋值符号右边的值将被赋给该元素，并同时创建所有比实际的索引值小的带索引值元素，但并不对这些元素赋值，使这些元素的值为未定义。仅在中间状态(**Transient State**)允许未定义元素(而值仍是不可见的)。发送带有未定义元素的一个 **record of** 类型值将会导致一个动态测试用例错误。

例 4：

```
//给出
type record of integer MyRecordOf;
var integer MyVar;
var MyRecordOf MyRecordVar:= { 0, 1, 2, 3 };
MyVar:= MyRecordVar[0]; //record of 类型值中的第一个元素赋值给了 MyVar
//也允许索引值在赋值符号的左边：
MyRecordVar[1]:= MyVar; //MyVar 被赋给了第二个元素
//以及下列两个赋值形式
MyRecordVar:= { 0, 1, -, 2, omit };
MyRecordVar[6]:= 6;
//将导致{ 0, 1, <unchanged>, 2, <undefined>, <undefined>, 6 };
//注意，如果第三个元素以前没有被赋值，那么它仍是未定义的
//而且，第六个元素(索引值为 5)在这次赋值之前没有被赋值过
```

注意：这就使得在一个 for 循环中一个元素接一个元素地拷贝 record of 类型值成为可能。例如，下面的函数翻转了一个 record of 类型值的元素：

```
function reverse(in MyRecord src) return MyRecord{
var MyRecord dest;
var integer I;
for(I:= 0; I <sizeof(src); I:= I + 1) {
    dest[sizeof(src) - 1 - I]:= src[I];
    }
return dest;
}
```

嵌套的 record of 和 set of 类型会导致一个类似多维数组的一个数据结构(见 4.2.5 节)。

例 5：

```
//给出
type record of integer MyBasicRecordOfType;
type record of MyBasicRecordOfType MyRecordOfType;
//那么变量 myRecordOfArray 将具有与一个二维数组相似的属性：
var MyRecordOfType myRecordOfArray;
//且对一个特定元素的引用可以看起来如下面的格式
//(第三个 MyBasicRecordOfType 构造的第二个元素的值)
myRecordOfArray [2][1]:= 1;
```

4.2.5　枚举类型

TTCN-3 支持枚举类型(**enumerated**)。枚举类型用于对只采用值的不同命名集的类型进行建模，每个枚举应该有一个标识符。对枚举类型的操作应该仅使用这些标识符，且仅限于赋值、等价和排序操作符。枚举标识符应该对该枚举类型来说是唯一的(但不必是全局唯一的)，因而也就仅在给定类型中的上下文中是可见的。枚举标识符仅会在其他结构化类型定义中重复使用，且不会在相同范围层次关系分支中同层或低层中被作为局部或全局可见性的标识符。

例 1：

```
type enumerated MyFirstEnumType {
    Monday, Tuesday, Wednesday, Thursday, Friday
};
type integer monday;
    //因为类型的名字具有局部或全局可见性，所以这个定义是非法的
    type enumerated MySecondEnumType {
    Saturday, Sunday, Monday
};
//因为该定义在另一个枚举类型中重复使用了枚举标识符 Monday，所以它是合法的
type record MyRecordType {};
integer Monday
//因为该定义在另一个结构化类型中使用枚举标识符 Monday 作为该结构化类型一个字段的标识
//符，所以它是合法的
type record MyNewRecordType {
    MyFirstEnumType firstField,
    Integer secondField
};
var MyNewRecordType newRecordValue:= { Monday, 0 }
//通过 MyNewRecordType 的 firstField 元素隐式地引用 MyFirstEnumType
const integer Monday:= 7
//因为该定义对相同范围单位中的不同 TTCN-3 对象重复使用枚举标识符 firstField，
//所以它是不合法的。
```

每个枚举可以选择性地在其枚举名字后面的括号中定义一个分配给它的整型值，在一个 **enumerate** 类型内部，每个元素分配的整型数应该是不同的。对于没有分配整型值的枚举，系统按文本顺序陆续关联一个整型数，从左边以 0 为开始，步长为 1，同时跳过任一手工分配值占用的数字，这些值仅被系统用于允许关系操作符的使用。

注意1：整型值也可以被系统用来编码/解码枚举值，不过这超出了本文讨论范围（除了 TTCN-3 允许的把编码属性关联到 TTCN-3 条目中去的情况）。

对一个 **enumerated** 类型的任意实例化或值引用来说，应隐式地或显式地引用给定类型。

注意 2：如果枚举类型是一个用户定义的结构化类型的一个元素，那么在值赋值、实例化等过程中，通过该元素隐式地引用此枚举类型（即通过该元素的标识符或在一个值列表表示法中该值的位置）。

例 2：

```
//MyFirstEnumType 和 MySecondEnumType 的有效实例化可以是
    var MyFirstEnumType Today:= Tuesday;
    var MySecondEnumType Tomorrow:= Monday;
//但是，因为两个枚举类型是不兼容的，所以下面的语句是非法的
    Today:= Tomorrow
```

4.2.6 联合类型

TTCN-3 支持联合类型（**union**）类型，**union** 类型是字段的汇集，这些字段每个都由一个标识符来标识，**union** 类型在对采用有限个已知类型之一的一个结构化类型建模时很有用。例：

```
type union MyUnionType
{
    Integer    number,
    charstring  string
};
//MyUnionType 的一个有效实例化可以是
var MyUnionType age, oneYearOlder;
var integer ageInMonths;
age.number:= 34; //使用引用字段的值表示。注意这种表示法使得给出的字段就是所选字段
oneYearOlder:= {number:= age.number+1};
ageInMonths:= age.number * 12;
```

用于设置值的值列表表示法不应该用于 **union** 类型的值。

同样，应使用点号来引用 **union** 类型。

例：

```
MyVar5:= MyUnion1.myChoice1;
//如果一个 union 类型嵌套在另外一个类型中，那么对它的引用可以看起来像如下格式
MyVar6:= MyRecord1.myElement1.myChoice2;
//注意，被引用的带有标识符为 myChoice2 的字段的 union 类型嵌套在一个 record 类型中
```

特别要说明的是，union 类型不允许使用可选字段（Optional fields），这就意味着关键字 Optional 不会与 union 类型一起被使用。

4.3 任意类型

特殊类型 anytype 类型被定义为一个 TTCN-3 模块中所有已知类型（**known types**）的集合的简写。

anytype 类型的字段名应由相应的类型名字唯一标识。

例：

```
//类型的一个有效用法可以是
var anytype MyVarOne, MyVarTwo;
var integer MyVarThree;
MyVarOne.integer:= 34;
MyVarTwo:= {integer:= MyVarOne + 1};
MyVarThree:= MyVarOne * 12;
```

对一个模块来说，**anytype** 是在本地定义的，(与其他预定义类型一样)不能直接由另外一个模块所引入。然而，**anytype** 类型的一个用户定义类型是可以被另一个模块引入的，这就使得该模块的所有类型都被引入。

注意：anytype 类型的用户定义类型“包含”声明该用户定义类型的模块引入的所有类型，引入这样一个用户定义类型到一个模块中去可能会引起副作用，因此对这种情况应谨慎处理。

4.4　数组

与许多编程语言相同，在 TTCN-3 中不认为数组是类型，但是可以在变量声明中对它们加以描述。数组可以声明为单维或多维的。

例 1：

```
var integer     MyArray1[3];    //例示一个带有 3 个元素的整型数组，其下标为 0～2
var integer     MyArray2[2][3];
//例示了一个具有 2×3 个元素的二维整型数组，其标号为(0,0)～(1,2)
```

数组的维数应该用一个结果为整型值的常数表达式来描述，也可以使用范围来描述，在后一种情况下，该范围上下界的值定义了该数组元素标号的上下界值。

例 2：

```
var integer MyArray3[1 .. 5];   //例示一个有 5 个元素的整型数组，其元素标号为 1～5
MyArray3[1]:= 10;               //最小标号 lowest index
MyArray3[5]:= 50;               //最高标号 highest index
var integer MyArray4[1 .. 5][2 .. 3 ];
//例示了一个有 5×2 个元素的二维整型数组，标号为(1,2)～(5,3)
```

数组元素的值应该与相应的变量声明兼容，这些值可以使用值列表表示法对它们分别进行赋值，或者使用索引表示法或多次使用值列表表示法进行赋值。当使用值列表表示法时，该列表的第一个值被赋给了数组的第一个元素(该元素的索引值为 0)，第二个值被赋给了第二个元素，以此类推。不予考虑赋值的元素应该在列表中显式地跳过或省略。赋值给多维数组时，要赋值的每一维都应解析到花括号中的一个值集合中。

例 3：

```
MyArray1[0]:= 10;
MyArray1[1]:= 20;
MyArray1[3]:= 30;
//或使用一个值列表
MyArray1:= {10, 20, -, 30};
MyArray4:= {{1, 2, 3, 4, 5}, {11, 12, 13, 14, 15}}
```

注意：使用多维数据结构的一个替换方式就是使用 record、record of、set 或 set of 类型。

例 4：

```
//给出
type record MyRecordType
{
    integer field1,
    MyOtherStruct field2,
    charstring field3
}
//MyRecordType 类型的一个数组可以是
var MyRecordType myRecordArray[10];
//对一个特定元素的引用可以看起来像如下格式
myRecordArray[1].field1:= 1;
```

4.5 递归类型

可应用的 TTCN-3 类型定义可以是递归的。然而，用户应该确保所有的类型递归是可解析的，且不会发生无限递归。

4.6 类型的兼容

通常，在赋值、实例化和比较时，TTCN-3 要求值的类型兼容。

为了本节论述方便，把值“b”称为要赋予、作为参数传递等情况的实际值，类型“B”称为值“b”的类型，类型“A”称为要获得的实际值“b”的值类型定义。

对于非结构化变量、常量、模板等，如果类型“B”解析为与类型“A”相同的源类型且不违反对类型“A”的子类型定义(例如范围、长度限制)，则值“b”与类型 A 是兼容的。

例：

```
//给出
Type integer MyInteger(1 .. 10);
:
Var integer x;
Var MyInteger y;
//那么
y:= 5;  //是一个有效的赋值
x:= y;  //是一个有效的赋值，因为 y 与 x 有相同的源类型，且不违反 x 的子类型定义
x:= 20; //是一个有效赋值
y:= x;  //是一个无效赋值，因为 x 的值超出了 MyInteger 的值域
x:= 5;  //是一个有效赋值
y:= x;  //是一个有效赋值，因为 x 的值在 MyInteger 的值域
```

4.6.1 记录类型兼容性

在结构化类型的情况下(enumerated 类型除外)，如果类型“B”的有效值与类型“A”兼容，类型“B”的一个值“b”与类型“A”兼容，则在这种情况下允许赋值、实例化和比较。

4.6.2　枚举类型兼容性

枚举类型与其他基本类型或结构化类型从不兼容(也就是说,对于枚举类型要求强类型机制)。

对于 record 类型，如果在定义的文本顺序上字段的数目、类型和可选性与值结构是相同的，且值“b”的每个已有字段与类型“A”的相应字段的类型兼容，那么这些有效值结构是相互兼容的。值“b”的每个字段的值被赋值给类型“A”的相应字段。

例 1：

```
//给出
type record AType {
    integer   (0..10) a optional,
    integer   (0..10) b optional,
    boolean       c
}
type record BType {
    integer        a   optional,
    integer   (0..10) b   optional,
    boolean      c
}
type record CType {     //带有不同字段名的类型
    integer  d   optional,
    integer  e   optional,
    boolean  f
}
type record DType {     //带有可选字段 c 的类型
    integer       a    optional,
    integer       boptional,
    boolean      c optional
}
type record EType {     //带有一个额外字段 d 的类型
    integer       a   optional,
    integer       b   optional,
    boolean      c,
    integer       d   optional,
}
var AType MyVarA:= { -, 1, true};
var BType MyVarB:= { omit, 2, true};
var CType MyVarC:= { 3, omit, true};
var DType MyVarD:= { 4, 4, true};
var EType MyVarE:= { 5, 5, true, omit};
//Then
MyVarA:= MyVarB;
//是一个有效的赋值，MyVarA 的值是( a:= <undefined>, b:= 2, c:= true)
MyVarC:= MyVarB;
//是一个有效赋值，MyVarC 的值是( d:= <undefined>, e:= 2, f:= true)
MyVarA:= MyVarD;        //因为字段的可选性不匹配，所以不是一个有效的赋值
MyVarA:= MyVarE;        //因为字段数目不匹配，所以不是一个有效赋值
```

```
MyVarC:= { d:= 20 };    //MyVarC 的实际值是{ d:=20, e:=2,f:= true }
MyVarA:= MyVarC
//不是一个有效的赋值，因为 MyVarC 的字段'd'违反 AType 类型字段'a'的子类型定义
```

对于 **record of** 类型和数组类型，如果它们的成分类型是兼容的，且类型“B”的值“b”不违反 **record of** 类型的长度限定的子类型或类型“A”数组的维数，则其有效的值结构是可兼容的。值“b”元素的值应该顺序赋值给类型“A”的实例，其中包括未定义的元素。

record of 类型和单维数组类型的有效值结构与 **record** 类型的兼容，且 **record of** 类型“B”的值“b”的元素数目或数组“B”的维数与 **record** 类型“A”的元素数目精确相同，那么 **record of** 类型和单维数组类型与 **record** 类型是兼容的。在确定兼容性的时候，**record** 类型字段的可选性并不重要，也就是说，它不影响字段的计数(这就意味着可选字段在计数的时候总是被包括在内)。使用 **record of** 类型元素值或数组对一个 **record** 类型实例的赋值应该按照相应 **record** 类型定义的文本顺序进行，未定义元素也包括在内。如果一个带有未定义值的元素被赋给该 **record** 类型的一个可选字段，将会引起对该可选元素的省略。而把一个带有未定义值的元素赋值给一个 **record** 类型的必要元素将导致错误。

注意：如果 **record of** 类型没有长度限制或已有的长度限制超出了相比较的 **record** 类型元素数目，且任意该 **record of** 类型定义的元素的索引小于或等于 **record** 类型元素数目的最小值，那么总是要履行兼容性要求。

如果一个 **record** 类型没有违反 **record of** 类型的长度限制或数组的维数大于等于该 **record** 类型元素数目，那么这个 **record** 类型值总是可以被赋值给该 **record of** 类型实例或单维数组。应该以未定义值给 **record** 类型值中遗漏的可选元素赋值。

例 2：

```
//给出
type record HType {
    integer a,
    integer b   optional,
    integer c
}
type record of integer IType
var  HType MyVarH:= { 1, omit, 2};
var  IType MyVarI;
var integer MyArrayVar[2];
//那么
MyArrayVar:= MyVarH;
//是一个有效的赋值，因为类型 MyArrayVar 和 HType 是兼容的
MyVarI:= MyVarH;
//是一个有效的赋值，因为类型兼容，且没有违反子类型定义
MyVarI:= { 3, 4 };
MyVarH:= MyVarI;
//是一个无效赋值，因为 Htype 的必要字段"c"没有接到赋值
```

仅 **set** 类型与其他 **set** 类型和 **set of** 类型兼容，**set** 类型和 **set of** 类型使用与 **record** 和 **record of** 类型相同的兼容性规则。

注意：这就暗示了，尽管不知道元素的发送和接收顺序，但在为 **set** 类型决定类型兼容性的时候，类型定义中字段的文本顺序是决定性的因素。

例 3：

```
//给出
type set FType {
    integer a   optional,
    integer b   optional,
    boolean c
}
type set GType {
    integer   d   optional,
    integer   e   optional,
    boolean   f
}
var FType MyVarF:= { a:=1, c:=true };
var GType MyVarG:= { f:=true, d:=7};
//那么
MyVarF:= MyVarG;   //是一个有效的赋值，因为类型 Ftype 和 Gtype 是兼容的
MyVarF:= MyVarA;   //是一个无效赋值，因为 MyVarA 是一个 record 类型
```

4.6.3　子结构化的兼容性

为结构化类型兼容性在本节中定义的规则对于这些类型的子类型来说也是有效的，即子结构的等价。

例：

```
//如果考虑上面的声明，那么
MyVarJ.H:= MyVarH;
//是一个有效的赋值，因为 Jtype 的字段 H 的类型与 Htype 是兼容的
MyVarI:= MyVarJ.H;
//是一个有效的赋值，因为 Itype 和 Jtype 的字段 H 的类型是兼容的
```

4.6.4　成分类型的类型兼容性

如果成分类型“B”的定义包含成分类型“A”的定义，则“B”的一个成分引用“b”与成分类型“A”兼容。这就意味着“B”包含至少与“A”相同的端口、变量和定时器实例以及常量声明，对于端口、变量和定时器实例来说，它们的类型和标识符都应该是相同的。如果成分类型“B”的一个成分引用“b”与成分类型“A”兼容，则该成分引用“b”可以被赋值给一个类型为成分类型“A”的变量“a”。

4.6.5　通信操作的类型兼容性

通信操作 **send**、**receive**、**trigger**、**call**、**getcall**、**reply**、**getreply** 和 **raise** 是较弱的类型兼容性规则的例外，它们要求强类型机制(**Strong typing**)。用于这些操作的值的类型或直接作为参数的模板必须也明确地定义在相关联的端口类型定义中。在 **receive** 或 **trigger** 操作中，也使用强类型机制来来存储接收到的值、地址或成分引用。

4.6.6　类型变换

在需要把一个类型的值变换为另一个非相同源类型派生的类型的值时，将使用预定义转换函数或用户定义的函数。

例：

```
//使用预定义函数 int2hex 把一个整型值转换为一个十六进制值
MyHstring:= int2hex(123, 4);
```

4.7 模块(Modules)

模块是 TTCN-3 的基本构造块。例如，一个模块可以定义一个完整的可执行测试套或仅仅是一个库，一个模块由定义部分(可选的)和一个模块控制部分(可选的)组成。

注意：术语“测试套”与一个包含测试用例和一个控制部分的完整 TTCN-3 模块是同义的。

4.7.1 模块命名

模块名具有 TTCN-3 标识符的格式，后接一个可选的对象标识符。

注意 1：模块标识符是模块非正式的文本名。

注意 2：模块名可以仅在对象标识符部分不同，不过在这种情况下，因为标识符的前缀不能解析这种名称冲突，所以在引入的时候应谨慎以避免名称冲突。

4.7.2 模块参数

模块(**module**)参数列表定义了由测试环境在运行时提供的一个值的集合，在测试执行时，将应按照常量来对待这些值。通过一对紧接在关键字 **modulepar** 之后的花括号中列出模块参数的标识符和类型来声明模块参数，模块应该仅在模块的定义部分中声明，允许模块参数声明的多次出现，但是每个参数只能声明一次(即不允许模块参数的重复定义)。

例 1：

```
module MyModulewithParameters
{
    modulepar {integer TS_Par0, TS_Par1; boolean TS_Par2 };
    :
    template MyType Mytemplate
    {
        Field TS_Par3
    };
    modulepar {hexstring TS_Par3 };
}
```

注意：这提供与 TTCN-2 中给测试套提供 PICS 和 PIXIT 值的测试套参数相似的功能性。

描述模块参数的默认值(**default values**)是允许的，这应该在模块参数列表中使用赋值来完成。一个默认值可以只是一个常量值(**literal value**)，且仅在该参数声明的地方赋值。如果测试系统对给定的参数不提供一个实际的运行值，则应该在测试执行期间使用默认值，否则由测试系统提供该实际值。

例 2：

```
module MyModuleDefaultParameter
{
    modulepar {integer TS_Par0:= 0, TS_Par1; boolean TS_Par2:= True};
    :
}
```

4.7.3　模块定义

模块定义部分描述该模块的顶层(**top-level**)定义，可以从其他模块引入标识符。可以在 TTCN-3 模块中定义的那些语言元素列在表 4-1 中，模块定义可以被其他模块引入。

例 1：

```
module MyModule
{
    //这个模块仅包含定义
    :
    const integer MyConstant:= 1;
    type record MyMessageType { … }
    :
    function TestStep(){ … }
    :
}
```

动态语言元素(如 **var** 或 **timer**)的声明应该仅在控制部分、测试用例、函数、可选步或成分类型中进行。

注意：TTCN-3 不支持在模块定义部分进行变量声明，这就意味着在 TTCN-3 中不能定义全局变量。但是，在一个测试成分中定义的变量可以被在该成分上运行的所有的测试用例、函数等所使用，且在控制部分中定义的变量有能力使它们的值独立于测试用例执行。

在模块定义部分中，定义可以被汇集在所谓的组(**Groups**)中。可以在允许单个声明的任何地方描述一个声明组。组可以被嵌套，即组可能包含其他组，这就允许测试套描述符在其他的概念中构建测试数据或描述测试行为的函数的类集。

如果需要的话，使用分组来提高可读性和为测试套增加逻辑结构。除了在组标识符的上下文中和通过使用关联的 **with** 语句给出的属性的情况之外，组和嵌套的组没有范围，这就意味着：

(1)整个模块的组标识符不必唯一，但是在相同层次上所有组标识符应该是唯一的，且层次中低层的子组不应该与高层的组具有相同的组名，如果必要的话，应使用点号来唯一标识组层次中的子组，例如对一个特定子组的引入。

(2)属性的重写规则(Overriding rules)。

例 2：

```
//一个定义集
group MyGroup {
    const   integer MyConst:= 1;
    :
    type record MyMessageType { … };
    group MyGroup1 {    //带有定义的子组
    type    record AnotherMessageType { … };
    const   boolean MyBoolean:=  false
    }
}
//  一个可选步组
    group MyStepLibrary {
    group MyGroup1 {    //具有与上面带有定义的子组相同名字的子组
```

```
        altstep MyStep11() { … }
        altstep MyStep12() { … }
        :
        altstep MyStep1n() { … }
    }
    group MyGroup2 {
        altstep MyStep21() { … }
        altstep MyStep22() { … }
        :
        altstep MyStep2n() { … }
    }
}
//在 S ibrary 中引入 MyGroup1 的一个引入语句
import from MyModule() {
        group MyStepLibrary.MyGroup1
}
```

4.7.4 模块控制

模块控制部分可以包含局部定义，描述实际测试用例的执行顺序(可能是重复的)，应该在模块的定义部分定义测试用例，在控制部分调用该测试用例。

例：

```
module MyTestSuite
{   //这个模块包含定义…
    :
    const integer MyConstant:= 1;
    type record MyMessageType { … }
    template MyMessageType MyMessage:= { … }
    :
    function MyFunction1() { … }
    function MyFunction2() { … }
    :
    testcase MyTestcase1()  runs on MyMTCType { … }
    testcase MyTestcase2()  runs on MyMTCType { … }
    :
    //…和控制部分，因此它是可执行的
    control
    {
        var boolean MyVariable;     //局部控制变量
        :
        execute MyTestCase1();      //测试用例的顺序执行
        execute MyTestCase2();
        :
    }
}
```

4.7.5　从模块导入

可以使用 import 语句重复使用不同模块中说明的定义。TTCN-3 没有明确的输出构架(export construct)，因此，默认模块定义部分中的所有模块定义都可以被引入。可以在模块定义部分的任何地方使用 import 语句，但是不应在控制部分使用该语句。

如果在 **import** 语句中对象标识符作为模块名(被引入该定义的模块的名字)被提供，那么应使用这个对象标识符去标识该正确的模块。

从一个模块中引入的所有定义应该仅在一个 **import** 语句中被引用。

如果一个引入的定义具有(使用 **with** 语句定义的)属性，那么这些属性也应该被引入。

注意：如果模块具有全局属性，那么这些属性与不带有这些属性的定义相关联。

例 1：

```
module MyModuleA
{   //这个模块包含定义和引入的定义
    :
    const   integer MyConstant:= 1;
    import from  MyModuleB all;  //引入定义的范围对 MyModuleA 来说是全局的
    import from   MyModuleC {
        type    MyType1y, MyType2;
        template    all
    }
    type record MyMessageType { … }
    :
    function MyBehaviourC()
    {
        const   integer MyConstant:= 2;
        //这里不能使用引入操作
    }
        :
    control
    {//这里不能使用引入操作
    :
    }
}
```

TTCN-3 支持对下列定义的引入：模块参数、用户定义类型、信号、常量、外部常量、数据模板、信号模板、函数、外部函数、可选步和测试用例。每个定义有一个名字(**name**，定义该元素的标识符，如一个函数名)、一个细节描述(**specification**，如一个类型描述或一个函数的特性)，在函数、可选步和测试用例的情况下，还包括一个相关的行为描述(**behaviour description**)。

例 2：

	Name	Specification	Behaviour description
Function	MyFunction	(**inout** MyType1 MyPar) **return** MyType2 **runs on** MyCompType	{const MyType3 MyConst:= …; : //更多的行为 }

	Specificaion	Name	Description

Type	record	MyRecordType	{
			field1 MyType4,
			field2 integer
			}

	Specificaion	Name	Description
Type	MyType5	MyTemplete	{ field1:= 1, field2:= MyConst, //MyConst 是一个模块常量 field3:= ModulePar //ModulePar 是一个模块参数 }

因为在相应的函数、可选步或测试用例被引入的时候，我们认为行为描述的内部对引入者来说是不可见的，所以行为描述对引入机制没有影响，因此也不在后续的说明中考虑它们。可引入定义的说明部分包含本地定义(**local definitions**，如结构化类型定义的字段名或枚举类型值)和引用的定义(**referenced definitions**，对类型定义、模板、常量或模块参数的引用)，对于上面的例子，可引入定义的说明部分如表 4-5 所示。

表 4-5 可引入定义的说明部分

	名字	本地定义	引用的定义
function	**MyFunction**	**MyPar**	**MyType1, MyType2, MyCompType**
type	**MyRecordType**	**field1, field2**	**MyType4, integer**
template	**MyTemplate**		**MyType5, field1, field2, field3, MyConst, ModulePar**

注意 1：本地定义一栏指的是新定义的标识符，它的值在定义中被赋值(如在模板定义中)，但是它们对引入机制并不重要。

注意 2：引用定义栏是那些来自于其他引用中的定义。例如，模板 MyTemplate 被引用的字段 field1、field2 和 field3 是 MyType5 的字段名，通过 MyType5 来引用它们。而 MyType5 来自于其他定义。

可引入定义的本地定义和引用定义如表 4-6 所示。

表 4-6 可引入定义的本地定义和引用定义

可引入的定义 Importable Definition	可能的本地定义 Possible Local Definitions	可能的引用定义 Possible Referenced Definitions
模块参数	参数名	模块参数类型
用户定义类型(对于所有的)		参数类型
枚举类型	具体值	
结构化类型	字段名	字段类型
端口类型		消息类型，特性
成分类型	常量名、变量名、定时器名和端口名	常量类型, 变量类型, 端口类型
特征	参数名	
常量		
外部常量		常量类型
数据模板	参数名	模板类型, 参数类型, 常量, 模块参数, 函数
特性模板		特性定义, 常量, 模块参数, 函数
函数	参数名	参数类型, 返回类型, 成分类型(**runs on**-子句)
外部函数	参数名	参数类型, 返回类型
可选步	参数名	参数类型, 成分类型(**runs on**-子句)
测试用例	参数名	参数类型, 成分类型(**runs on**-和 **system**-子句)

4.7.6　引入规则

TTCN-3 引入机制区分引入定义中引用定义的标识符和引用定义使用的必要信息，对于引用定义的使用，不要求引用定义的标识符，因此不是自动引入。

使用引入操作时，应该应用如下规则。

(1)只有模块最上层的定义可以被引入。出现在较低层次范围的定义(如定义在一个函数中的局部常量)不应被引入。

(2)只允许从源模块(即在 import 语句中引用的标识符的实际定义所在的模块)的直接引入。

(3)定义的名字和所有局部定义一起被引入。

注意 1：一个本地定义，如一个用户定义记录类型的一个字段名，仅在定义它的上下文中有意义，例如一个记录类型的字段名只能用来访问该记录类型的字段，而不能出了这个范围。

(4)引用定义中的所有信息一起被引入。

注意 2：import 语句是传递的，例如，如果一个模块 A 从使用模块 C 中定义的一个类型引用的模块 B 中引入一个定义，则该类型使用所需的相应信息自动被引入到模块 A 中。

(5)默认地，不自动引入引用定义的标识符。如果希望隐式引入该引用定义的标识符，应使用 recursive 命令。

注意 3：如果在使用默认引入机制时(如变量实例化)，希望在引入模块中使用被引用的定义，应该从它的源模块显式地引入。

(6)引入一个函数、可选步或测试用例时，相应的行为说明和行为说明内使用的所有定义对于引入模块来说仍是不可见的。

例 1：

```
module ModuleONE {
    modulepar {
        integer ModPar1, ModPar2:= 7
    }
    type    record RecordType_T1 {
        integer Field1_T1,
        boolean Field2_T1
    }
    type    record RecordType_T2 {
        MyRecordType_T1 Field1_T2,          //RecordType_T1 的使用
        MyRecordType_T1 Field2_T2,
        integer Field3_T2
    }
    const   integer MyConst:= 13;
    template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= {
        //参数化的模板
        Field1_T2:= TempPar_T2,             //对模板参数的引用
        Field2_T2:= {MyConst, true},        //对模块常量的引用
        Field3_T2:= ModPar1                 //对模块参数的引用
    }
} //结束模块 ModuleONE

module ModuleTWO {
```

```
    import from ModuleONE {
        template Template_T2
    }
    //在 ModuleTWO 中只有名字 Template_T2 和 TempPar_T2 是可见的
    //注意，标识符 TempPar_T2 只能被用在 Template_T2 的上下文中，如提供一个实参值时
    //为引用定义 RecordType_T2、RecordType_T1、Field1_T2、Field2_T2、Field3_T3、
    //MyConst 和 ModPar1
    //引入 Template_T2 使用所需的所有信息(如用于类型检查目的信息)，但是它们的标识
    //符在 ModuleTWO 中是可见的
    //这意味着，不可能在不显式引入 ModuleTWO 中类型 RecordType_T1 或 RecordType_T2 的
    //情况下，使用这些类型的常量 MyConst 或声明这些类型的变量
    import from ModuleONE {
        modulepar ModPar2
    }
    //从 ModuleONE 引入的 ModuleONE 的模块参数 ModPar2 可以作为一个整型的常量
} //结束模块 ModuleTWO

module ModuleTHREE {
    import from ModuleONE all;       //从 ModuleONE 引入所有定义
    type port MyPortType {
        inout RecordType_T2
    }
    type component MyCompType {
        var integer MyComponentVar:= ModPar2; //对 ModuleONE 的一个模块参数的引用
    port MyPortType MyPort
    }
    function MyFunction () return integer {
        return MyConst//返回 ModuleONE 中定义的一个模块常量
    }
    testcase MyTestCase (out RecordType_T2 MyPar) runs on MyCompType {
        var integer MyTCVar:= ModPar2; //对 ModuleONE 的一个模块参数的引用
        MyPort.send(Template_T2);
        //发送 ModuleONE 中定义的一个模板
        MyPort.receive(RecordType_T2: ?) ->value MyPar;
        //接收到的值赋给输入/出参数 MyPar
    } //结束测试用例 MyTestCase
} //end ModuleTHREE

module ModuleFOUR {
    import from ModuleTHREE {
        testcase MyTestCase
    }
    //只有名字 MyTestCase 和 MyPar 在 ModuleFOUR 中是可见的和可用的
    //通过 ModuleTHREE 从 ModuleONE 引入 RecordType_T2 的类型信息
    //从 ModuleTHREE 引入 MyCompType 的类型信息
    //MyTestCase 行为部分使用的所有定义对于 ModuleFOUR 使用者来说仍是隐藏的
} //结束 ModuleFOUR
```

TTCN-3 默认的引入机制引入引用定义而不需它们的标识符，这就意味着在引入模块中不能使用引用定义，例如声明一个变量或在一个端口上发送。尽管这个默认引入机制避免引入模块名字空间的混乱，但是在一些情况下，和标识符一起引入所有引用定义还是被期望的。在 TTCN-3 中，关键字 **recursive** 提供这个特性。

递归的引入定义与所有引用定义一起被引入，即所有引用定义的标识符在引入模块中是可见的和可用的。

注意 1：递归的引入语句在源模块中是传递的，例如，一个模块 A 从使用类型 T 的模块 B(类型 T 也是定义在模块 B 中)中递归地引入一个定义，那么类型 T 自动地被引入到模块 A 中。

注意 2：递归的引入语句在模块的边界间是不传递的，这就是说，如果模块 A 从使用类型 T(类型 T 在模块 C 中定义)的模块 B 中递归地引入一个定义，那么类型 T 不能自动地被引入到模块 A 中去，类型 T 必须被显式地从模块 C 中引入，即从它的源模块中引入。

例 2：

```
module ModuleFIVE {
    import from ModuleONE  recursive {
        template Template_T2
    }
    //Template_T2 的递归引入将从 ModuleONE 引入 RecordType_T2、RecordType_T1、
    //MyConst 和 ModPar1 的定义
    //由于类型 RecordType_T2 和 RecordType_T1 的引入，这些类型的字段名 Field1_T1、
    //Field2_T1、Field1_T2、Field2_T2 和 Field3_T3 在 ModuleFIVE 中将变为可见的
} //结束模块 ModuleFIVE

module ModuleSIX {
    import from ModuleTHREE recursive {
        testcase MyTestCase
    }
    //如果该模块不包括从 ModuleONE 中递归引入 RecordType_T2 的一个引入语句将导致一个错误
    //从 ModuleTHREE 对 MyTestCase 的递归引入需要从 RecordType_T2 和 MyCompType
    //的源模块中对它们进行递归引入
    //RecordType_T2 的源模块是模块 ModuleONE
    //尽管 MyCompType 的源模块是模块 ModuleTHREE，但是对它的递归引入仍将导致一个错误，
    //因为这个定义需要 ModuleONE 中的定义
} //结束 ModuleSIX

module ModuleSEVEN {
    import from ModuleONE recursive {
    modulepar ModPar2;
    type RecordType_T2
    }
    //从 ModuleONE 引入 ModPar2、RecordType_T2 和 RecordType_T1(RecordType_T1
    //被 RecordType_T2 使用)
    //通过递归引入，RecordType_T2 和 RecordType_T1 的字段名 Field1_T1、Field2_T1、
    //Field1_T2、Field2_T2、Field3_T3 将变为可见的
    import from ModuleTHREE recursive {
    testcase MyTestCase
```

```
        }
        //从 ModuleTHREE 引入 MyTestCase、MyCompType(被 MyTestCase 使用)和 MyPortType
        //(被 MyCompType 使用)
        //通过对 MyTestCase 和 MyComptype 的递归引入,标识符 MyPar(在 MyTestCase 中定义)、
        //MyComponentVar 和 MyPort(都在 MyCompType 中定义)变为可见的
        //MyTestCase 递归引入所需的来自 ModuleOne 的定义通过前面的 import 语句被递归引入

    } //结束 ModuleSEVEN

    module ModuleEIGHT {
        import from ModuleONE {
            modulepar ModPar2;
            type RecordType_T2
        }
        import from ModuleTHREE recursive {
            testcase MyTestCase
        }
        //将导致一个错误,因为为了对 MyTestCase 的递归引入,也需要从 ModuleONE 完全引入类型
        //RecordType_T1,或者,换句话说,需要递归地引入 RecordType_T2
    } //结束 ModuleEIGHT
```

可以引入单个定义。

例 3:

```
    import from MyModuleA {
        type MyType1                                //从 MyModuleA 引入一个类型定义
        }
        import from MyModuleB {
            type MyType2, Mytype3, MyType4;         //引入三个类型
            template    MyTemplate1;                //引入一个模板
            const   MyConst1, MyConst2              //引入两个常量
    }
```

可以使用关键字 **all** 来引入一个模块定义部分的所有定义。如果使用关键字 **all** 来引入一个模块的所有定义,那么应该没有其他的引入格式用于相同的 **import** 语句。

例 4:

```
    import from MyModule all;
```

如果有些声明不希望被引入,那么它们的种类和标识符应该列在关键字 **except** 后一对花括号中的例外列表中。

例 5:

```
    Import from MyModule all except {
        type MyType3, MyType5
        //从 import 语句中除去类型声明 MyType3 和 MyType5
        //但是引入 MyModule 的所有其他声明
    }
```

也允许在例外列表中使用关键字 **all**,这就排除了来自引入语句的相同种类的所有声明。

例 6：

```
import from MyModule all except {
    type    MyType3, MyType5; //从 import 语句中排除了两个类型
    template all    //从 import 语句中排除了 MyModule 中声明的所有模板
}
```

可以引入定义组(group)。

例 7：

```
import from MyModule {
    group MyGroup
}
```

引入一个组的效果与一个列出这个组所有可引入定义(包括子组 sub-groups)的 **import** 语句相同。

TTCN-3 的组只是用于结构化的目的，而不是范围单元，因此不允许直接引入子组(即定义在另一个组中的组)，即不通过子组所嵌套的组就直接引入子组。如果应引入的子组名与相同模块中另一个子组名相同，应使用点号来唯一标识要被引入的子组。

如果不愿引入一个组的一些定义，应在关键字 **except** 后面花括号中的例外列表中列出这些定义的种类和标识符。

例 8：

```
import from MyModule
{
    group MyGroup except  {
    type MyType3, MyType5
    //从 import 语句中除去类型定义 MyType3 和 MyType5，但引入 MyGroup 的所有其他定义
    }
}
```

允许在例外列表中使用关键字 **all**，这从 import 语句中排除了相同的所有定义。

例 9：

```
import from MyModule {
    group   MyGroup except  {
    type    MyType3, MyType5;   //从 import 语句中排除了两个类型
    }
}
template all //从 import 语句中排除了 MyGroup 中定义的所有模板
```

关键字 **all** 可以用来引入一个模块中相同种类的所有定义。

例 10：

```
import from MyModule {
    type all;
    template all
}
```

如果希望从一个给定的 import 语句中排除一个种类的一些声明，那么这些声明使用的标识符应列在关键字 **except** 之后。

例 11：

```
import from MyModule {
    type all except MyType3, MyType5;   //引入除MyType3和MyType5外的所有类型
    template all                        //引入 Mymodule 中定义的所有模板
}
```

所有的 TTCN-3 模块应该有它们自己的名字空间，在各自的名字空间里所有的定义应该被唯一标识。由于引入可能引起名字冲突(例如来自不同模块的引入、组引入或递归定义引入)，应该使用被引入定义的模块的标识符作为(引起名字冲突的)引入定义的前缀来解决名字冲突，前缀和标识符使用点号(.)隔开。

在没有多义性的情况下，使用引入定义时，前缀不必(但可以)出现。当定义在定义它的模块中被引用时，该模块(当前模块)的标识符也可作为该定义标识符的前缀。

例 12：

```
module MyModuleA {
    :
    type bitstring MyTypeA;
    import from SomeModuleC {
    type  MyTypeA,      //MyTypeA 是字符串类型
        MyTypeB         //MyTypeB 是字符串类型
    }
    :
    control {
        :
        var SomeModuleC.MyTypeA MyVar1:= "Test String"; //必须使用前缀
        var MyTypeA MyVar2:= '10110011'B;      //这是最初的 MyTypeA
                :
        var MyTypeB MyVar3:= "Test String";    //不必使用前缀 …
        var SomeModuleC.MyTypeB MyVar3:= "Test String";
        //… 但如果希望，它可以用前缀
        :
    }
}
```

注意：即便在不同模块中带有相同名字的定义的实际定义是相同的，也总是假设在不同模块中带有相同名字的定义是不同的。例如，引入一个已经在本地定义了的类型，甚至使用了相同的名字，会导致在该模块中两个不同的有效类型。

在单个定义、定义组、相同种类定义等上使用 **import** 可能导致对相同定义的多次引用，这样的情况应该通过系统去解决，且定义应该仅被引入一次。

注意：解决这样的多义性的机制(如重写(**overwriting**)和给用户发送警告)不在本文研究范围之内，应由 TTCN-3 工具提供。

所有 **import** 语句和引入语句中的定义应该按照它们出现的顺序一个接一个地分别加以考虑。有必要指出的是，**except** 语句通常不从被引入语句中排除所列出的定义。相同种类定义的所有引入定义语句可以看成与单个定义标识符的等价列表的简写表示，**except** 语句仅从这个单个的列表中排除定义。

例 13：

```
import from MyModule {
    type all except MyType3;    //引入 MyModule 中除 MyType3 外的所有类型
    type  MyType3               //明确引入 MyType3
}
```

在从其他来源而非 TTCN-3 模块中引入定义的情况下，应使用语言说明来指示被引入定义来源的语言(如模块(**module**)、包(**package**)、库(**library**)或者甚至是文件(**file**)，可能带有版本号)，由关键字 **language** 和一个随后的表示语言的文本声明组成。当从一个与引入模块相同版本的 TTCN-3 模块中引入时，该语言说明的使用是可选的。

例 14：

```
import from MyASN1Module language "ASN.1:1997"
{
    type MyASN1Type
}
```

注意：设计引入机制来允许重复使用来自其他 TTCN-3 或 ASN.1 模块的 TTCN-3 和 ASN.1 定义。引入使用其他语言(如 SDL 包)书写的说明中的定义的规则可以遵循 TTCN-3 规则，或可能是必须分别定义的规则。对非 TTCN-3 和 ASN.1 语言的引入规则不包括在本书之中。

4.8　运算符

TTCN-3 支持许多可用于 TTCN-3 表达式的术语中的预定义操作符，它们分为以下七类。

(1)算术运算符(arithmetic operators)。

(2)串运算符(string operators)。

(3)关系运算符(relational operators)。

(4)逻辑运算符(logical operators)。

(5)位运算符(bitwise operators)。

(6)移位运算符(shift operators)。

(7)循环移位符(rotate operators)。

TTCN-3 运算符列表如表 4-7 所示。

表 4-7　TTCN-3 运算符列表

类别(Category)	运算符(Operator)	符号或关键字(Symbol or Keyword)
算术运算符(Arithmetic operators)	Addition	+
	subtraction	-
	multiplication	*
	division	/
	modulo	mod
	remainder	rem
串运算符(String operators)	concatenation	&

续表

类别(Category)	运算符(Operator)	符号或关键字(Symbol or Keyword)	
关系运算符(Relational operators)		equal	==
		less than	<
		greater than	>
		not equal	!=
		greater than or equal	>=
		less than or equal	<=
逻辑运算符(Logical operators)		logical not	not
		logical and	and
		logical or	or
		logical xor	xor
位运算符(Bitwise operators)		Bitwise not	not4b
		Bitwise and	and4b
		Bitwise or	or4b
		Bitwise xor	xor4b
移位运算符(Shift operators)		shift left	<<
		shift right	>>
循环移位运算符(Rotate operators)		rotate left	<@
		rotate right	@>

表 4-8 表示了这些运算符的优先级别，表中每一行中所列的运算符都具有相同的优先级别。如果一个表达式中出现多个具有相同优先级的运算符，则按从左至右的顺序评定优先级别。括号可以用于把操作数集合在一起，在这种情况下，括号具有最高优先级。

表 4-8 运算符优先级

优先级	操作符类型	操作符
最高		**(…)**
	一元，二元	**+, - *, /, mod, rem**
	二元	**+, -, &**
	一元	**not4b**
	二元	**and4b**
	二元	**xor4b**
	二元	**or4b**
	二元	**<<, >>, <@, @>**
	二元	**<, >, < =, >=**
	二元	**==, !=**
	二元	**not**
	二元	**and**
	一元	**xor**
	二元	**or**
最低	二元	

4.8.1 算术运算符

算术运算符指加(**addition**)、减(**subtraction**)、乘(**multiplication**)、除(**division**)、取模(**modulo**)和取余(**remainder**)操作。除了取模操作(**mod**)外，这些操作符的操作数应该为整型(**integer**，包

括整型的派生类型)或浮点型(**float**，包括浮点型的派生类型)；取模操作只能使用整型(**integer**，包括整型的派生类型)。

对整型操作数进行算术运算，结果仍为整型。对浮点型操作数进行算术运算，结果为浮点型。

在加或减法操作采用一元操作符时，操作上述操作数的规则也同样适用。对正数进行减法的结果可以是负数；反之对负数进行减法，结果可以是正数。

执行除法(/)操作的两个结果如下。

(1) 当被除数和除数均为整数时，结果为整数(也就是说，分数部分被丢弃)。

(2) 当被除数和除数均为浮点数时，结果为浮点数(也就是说，不会丢弃分数部分)。

对整型操作数进行取余(**rem**)和取模(**mod**)操作，结果为整型，x **rem** y 和 x **mod** y 都计算 x 被 y 除所得的余数。因此，y 仅被定义为非零的操作数。对于正数 x 和 y，x **rem** y 和 x **mod** y 操作的结果一样，但是对于负数 x 和 y，结果则不一样。

mod 和 **rem** 操作的形式化定义如下：

x **rem** y　=　x-y*（x/y）

x **mod** y　=　x **rem** |y|　　如果 x >= 0

　　=　0　　如果 x < 0 且 x **rem** |y| = 0

　　=　|y| + x **rem** |y|　　如果 x < 0 且 x **rem** |y| < 0

表 4-9 举例说明了 **mod** 和 **rem** 操作的区别。

表 4-9　mod 和 rem 操作的结果

X	-3	-2	-1	0	1	2	3
x mod 3	0	1	2	0	1	2	0
x rem 3	0	-2	-1	0	1	2	0

4.8.2　串运算符

预定义的串运算符完成可兼容串类型的连接，该操作是从左至右的简单连接，没有暗含算术加法的形式，结果类型为操作数的原始类型。

例：

'1111'B　**&**　'0000' B　**&**　'1111'B　　结果为　　'111100001111'B

4.8.3　关系运算符

预定义的关系运算符表示的关系有等于(**equality**，==)、小于(**less than**，<)、大于(**greater than**，>)、不等于(**non-equality to**，!=)、大于等于(**greater than or equal to**，>=)和小于等于(**less than or equalto**，<=)。除了枚举类型(**enumerated**)这个例外，相等和不相等的操作数可以是任意的但兼容的类型。而其他关系运算操作的操作数类型只能是整型(**integer**，包括整型的派生类型)、浮点型(**float**，包括浮点型的派生类型)或一些枚举类型(**enumerated**)的实例。该运算的结果为布尔类型。

如果两个字符串(**charstring**)或通用字符串(**universal charstring**)的长度以及所有对应位置的字符都相等，则这两个串相等。**bitstring**、**hexstring** 或 **octetstring** 类型的值使用相同的相等规则，但有例外——所有位置上应该相等的部分是相应的是比特串(**bits**)、十六进制串(**hexadecimal digits**)或十六进制位组串(**pairs of hexadecimal digits**)。

两个记录类型(**record**)、集合类型(**set**)、**record of** 类型或 **set of** 类型值相等，当且仅当它们

的有效值结构兼容且所有相应域的值相等。Record 类型值可以与 **record of** 类型值比较，而 **set** 值可以与 **set of** 值比较，在这些情况下的比较规则与比较两个 **record** 类型值或 **set** 类型值的规则一样。

注意：“所有字段”意味着在一个 **record** 类型没有表示实际值的可选字段将被认为是未定义值。这个字段在和另一个 **record** 类型比较时，仅和遗漏的可选字段等价(同样被认为是一个未定义字段值)；当它与 **record of** 类型的一个值比较时，仅与一个未定义值的元素相等。这个规则同样适用于两个 **set** 类型值或一个 **set** 类型值和一个 **set of** 类型值做比较时。

两个 **union** 类型值相等，当且仅当在这两个 **union** 类型的值中，被选中的字段的类型兼容且它们的实际值相等。

例：

```
//给定
type set SetA {
    integer a1  optional,
    integer     a2  optional,
    integer     a3  optional
};
type set SetB {
    integer     b1  optional,
    integer     b2  optional,
    integer     b3  optional
};
type set SetC {
    integer     c1  optional,
    integer     c2  optional,
};
type set of integer     SetOf;
type union UniD {
    integer     d1,
    integer     d2,
};
type union UniE {
    integer     e1,
    integer     e2,
};
type union UniF {
    integer     f1,
    integer     f2,
    boolean f3,
};
//以及
const SetA  conSetA1    :=  { a1    :=   0,  a3 :=  2 };
//注意字段值的顺序无关紧要
const SetB  conSetB1    := { b1:= 0, b3:= 2 };
const SetB  conSetB2    := { b2:= 0, b3:= 2 };
const SetC  conSetC1    := { c1:= 0, c2:=2 };
const SetOf     conSetOf1       := { 0, omit, 2 };
const SetOf     conSetOf2   := { 0, 2 };
```

```
const UniD  conUniD1        := { d1:= 0 };
const UniE  conUniE1        := { e1:= 0 };
const UniE  conUniE2        := { e2:= 0 };
const UniF  conUniF1        := { f1:= 0 };

//那么
conSetA1 == conSetB1;
//返回 true
A1 == conSetB2;
//返回 false，因为 a1 和 a2 的值与它们在 conSetB2 中的对应成分的值不相等
//(对应元素没有被省略)
conSetA1 == conSetC1;
//返回 false，因为 SetA 和 SetC 中的有效的值结构是不兼容的
conSetA1 == conSetOf1;
//返回 true
conSetA1 == conSetOf2;
//返回 false，作为与 a2 对应成分的值是 2，但是与 a3 对应的成分却没有定义
conSetC1 == conSetOf2;
 //返回 true
conUniD1 == conUniE1;
//返回 true
conUniD1 == conUniE2;
//返回 false，作为被选中字段的 e2 与 UniD1 中的 d1 字段并不是对应字段
conUniD1 == conUniF1;
//返回 false, UniD1 和 UniF 的有效的值结构是不兼容的
```

4.8.4　逻辑运算符

预定义的布尔操作符执行否定(**negation**)、逻辑与(**logical** and)、逻辑或(**logical or**)和逻辑异或(**logical xor**)操作，它们的操作数应该是布尔类型(**boolean**)，逻辑运算的结果也是布尔类型。

逻辑否定(**not**)是一个一元运算符，如果其操作数的值是 **false**，则返回 **true**；反之，如果操作数的值为 **true**，则返回 **false**。

如果逻辑与(**and**)运算的两个操作数的值均为 **true**，则返回 **true**；否则返回 **false**。

如果逻辑或(**or**)运算的两个操作数中至少有一个操作数的值为 **true**，则返回 **true**；否则返回 **false**。

如果逻辑异或(**xor**)运算的两个操作数中有且只有一个操作数的值为 **true**，则返回 **true**；如果其两个操作数的值同为 **false** 或同为 **true**，则返回 **false**。

4.8.5　位运算符

预定义的位运算符执行按位取反(bitwise not)、按位与(bitwise and)、按位或(bitwise or)或按位异或(bitwise xor)操作，这些运算符分别为 not4b、and4b、or4b 和 xor4b。

注意：读成“按位取反”、“按位与”等。

位运算符的操作数的类型应该是 **bitstring**、**hexstring**、**octetstring**。**and4b**、**or4b** 和 **xor4b** 的三个操作数的类型应该是可兼容的，位运算的结果类型应该是该运算操作数的原始类型。

位运算 **not4b** 是一个一元运算符，它对其操作数的每一位进行取反操作，将操作数中为 1 的位变为 0，为 0 的位变为 1，即

```
not4b '1'B 得到 '0'B
not4b '0'B 得到 '1'B
```

例 1：

```
not4b '1010'B 得到 '0101'B
not4b '1A5'H 得到 'E5A'H
not4b '01A5'O 得到 'FE5A'O
```

位运算符 **and4b** 接受两个等长的操作数，对于两个操作数中相应位的值均为 1 时，结果中相应位的值也为 1，否则为 0，即

```
'1'B and4b '1'B 得到 '1'B
'1'B and4b '0'B 得到 '0'B
'0'B and4b '1'B 得到 '0'B
'0'B and4b '0'B 得到 '0'B
```

例 2：

```
'1001'B and4b '0101'B 得到 '0001'B
'B'H and4b '5'H 得到 '1'H
'FB'O and4b '15'O 得到 '11'O
```

位运算符 **or4b** 接受两个等长的操作数，对于两个操作数中相应位的值均为 0 时，结果中相应位的值也为 0，否则为 1，即

```
'1'B or4b '1'B 得到 '1'B
'1'B or4b '0'B 得到 '1'B
'0'B or4b '1'B 得到 '1'B
'0'B or4b '0'B 得到 '0'B
```

例 3：

```
'1001'B or4b '0101'B    得到 '1101'B
'9' H    or4b '5'H  得到 'D'H
'A9 'O   or4b 'F5'O 得到 'FD'O
```

位运算符 xor4b 接受两个等长的操作数，对于两个操作数中相应位的值均为 0 或均为 1 时，结果中相应位的值也为 0，否则为 1，即

```
'1'B xor4b '1'B 得到 '0'B
'0'B xor4b '0'B 得到 '0'B
'0'B xor4b '1'B 得到 '1'B
'1'B xor4b '0'B 得到 '1'B
```

例 4：

```
'1001'B xor4b '0101'B 得到 '1100'B
'9'H    xor4b '5'H 得到 'C'H
'39'O   xor4b '15'O 得到 '2C'O
```

4.8.6 移位运算符

预定义的移位运算符执行左移（**shift left**，<<）和右移（**shift right**，>>）操作。此类运算符左侧

的操作数类型应该是 **bitstring**、**hexstring** 或 **octetstring**，右侧的操作数类型应该是整型 **integer**，而运算结果的类型则应该与其左侧操作数的类型相同。

移位运算根据其左侧的操作数类型的不同而不同，如果其左侧操作数的类型是

(1) **bitstring**，那么应用的移位单位是 1bit。

(2) **hexstring**，那么应用的移位单位是 1 个十六进制位。

(3) **octetstring**，那么应用的移位单位是 1 个八位组 bit。

左移(<<)操作符接受两个操作数，它将左侧操作数左移右侧的操作数所描述的位数，移出的过多的位(**bits**、**hexadecimal digits** 或 **octets**)被丢弃。左侧操作数的每个移动单位在向左移的过程中，低位(右边的位)补 0(根据左侧操作数的类型决定补'0'B、'0'H 或是'00'O)。

注意：如果对左侧的操作数执行左移操作时，发生了一个与系统相关的溢出，则应该指定一个错误判定。

例 1：

```
'111001'B << 2  得到 '100100'B
'12345'H << 2   得到 '34500'H
'1122334455'O << (1+1)  得到 '3344550000'O
```

右移(>>)操作符接受两个操作数，它将左侧操作数右移右侧的操作数所描述的位数，移出的过多的位(**bits**、**hexadecimal digits** 或 **octets**)被丢弃。左侧操作数的每个移动单位在向右移的过程中，高位(左边的位)补 0(根据左侧操作数的类型决定补'0'B、'0'H 或是'00'O)。

例 2：

```
'111001'B >> 2  得到 '001110'B
'12345 'H >> 2  得到 '00123'H
'1122334455'O >> (1+1)  得到'0000112233'O
```

4.8.7　循环移位运算符

预定义的循环移位运算符执行循环左移(**rotate left**，<@)和循环右移(**rotate right**，@>)操作。此类运算符左侧的操作数类型应该是 **bitstring**、**hexstring**、**octetstring**、**charstring** 或 **universal charstring**，右侧的操作数类型应该是整型 **integer**，而运算结果的类型则应该与其左侧操作数的类型相同。

循环移位运算根据其左侧的操作数类型的不同而不同，如果其左侧操作数的类型是

(1) **bitstring**，那么应用的循环移位单位是 1bit。

(2) **hexstring**，那么应用的循环移位单位是 1 个十六进制位。

(3) **octetstring**，那么应用的循环移位单位是 1 个八位组 bit。

(4) **charstring** 或 **universal charstring**，那么那么应用的循环移位单位是 1 个字符。

循环左移(<@)操作符接受两个操作数，它将左侧操作数循环左移右侧的操作数所描述的位数，移出的过多的位(**bits**、**hexadecimal digits**、**octets** 或 **characters**)被从左侧操作数的右边重新插入到该操作数中去。

例 1：

```
'101001'B <@ 2  得到     '100110'B
'12345'H <@ 2   得到     '34512'H
'1122334455'O <@ (1+2)  得到     '4455112233'O
"abcdefg" <@ 3  得到     "defgabc"
```

循环右移(@>)操作符接受两个操作数，它将左侧操作数循环右移右侧的操作数所描述的位数，移出的过多的位(**bits**、**hexadecimal digits**、**octets** 或 **characters**)被从左侧操作数的左边重新插入到该操作数中去。

例 2：

```
'100001'B @> 2  得到     '011000'B
'12345'H @> 2   得到     '45123'H
'1122334455'O @> (1+2)  得到     '3344551122'O
"abcdefg" @> 3  得到     "efgabcd"
```

思考题

1．试说明 TTCN 中有哪几种基本类型。

2．在 TTCN 的模块中能否定义变量？为什么？

3．TTCN 的运算符有哪些？它们的优先级是怎样的？

第5章 类 型 声 明

本章介绍 TTCN-3 核心语言元素的各种声明，其中包括常量声明、变量声明、定时器声明、消息声明、过程特性声明、模板声明。通过学习这些声明的语法，能够知道进行常量、变量等声明时需要注意的各种情况，为后续利用 TTCN-3 进行测试打下基础。

5.1 常量声明

可以在模块的定义、成分类型定义、模块控制部分、测试用例、函数和可选步(altsteps)中声明和使用常量。常量由关键字 const 来标识，应该在常量声明的位置为其赋值。

例 1：

```
const   integer MyConst1:= 1;
const   boolean MyConst2:=true, MyConst3:=false;
```

可以在模块内部给常量赋值，也可以在模块外部进行，后一种情况用关键字 external 来标识外部常量声明。

例 2：

```
external const integer MyExternalConst; //外部常量声明
```

一个外部常量可以具有模块中已知类型之外的任意类型，即模块中定义的源类型(root type)或用户定义类型以及从其他模块中引入的类型之外的其他类型。从该类型到一个外部常量的外部表示的映射超出了本文讨论范围，如何给一个模块传入外部常量值的机制也超出了本文的讨论范围。

5.2 变量声明

变量用关键字 var 表示，可以在模块的控制部分、测试用例、函数或可选步中声明、使用变量。此外，还可以在成分类型定义中声明变量，并在测试用例、可选步和使用这些给定成分类型的函数中使用它们。但是，不能在模块的定义部分声明或使用变量，也就是说 TTCN-3 不支持全局变量的概念。变量声明时可以给它赋一个的初始值，这个赋值是可选的。

例：

```
var integer MyVar1:=1;
var boolean MyVar2:=true, MyVar3:=false;
```

运行时使用未初始化的变量会导致测试用例错误。

5.3 定时器声明

可以在模块的控制部分、测试用例、函数或可选步中声明、使用定时器。此外，还可以在成分类型定义中声明定时器，并在测试用例、可选步(和使用这些给定成分类型的函数中使用它们。

声明一个定时器时，可以选择给它赋一个默认持续时间值，如果没有再指定其他值，该定时器将使用该默认值开始计时。定时器的值是一个非负浮点数(即大于等于 0.0)，以秒为单位。

例 1：

```
timer MyTimer1:= 5E-3; //声明定时器 MyTimer1，默认值为 5ms
timer MyTimer2;//声明定时器 MyTimer2，但没有赋默认值，将在运行时给该定时器赋值
```

除了声明单个的定时器外，还可以声明定时器数组。定时器数组中每个元素的持续时间值由一个值数组来赋值，值数组中的第一个元素赋值给定时器数组的第一个元素，以此类推。如果对定时器默认的持续时间赋值时希望跳过定时器数组的一些元素，则必须明确地表明使用的符号("-")。

注意：这意味着定时器数组元素个数与值数组个数必须保持完全一致。

例 2：

```
timer t_Mytimer1[5]:= { 1.0, 2.0, 3.0, 4.0, 5.0 }
                  //定时器数组的所有元素都有一个默认持续时间值
timer t_Mytimer2[5]:= { 1.0, 2.0, 3.0, 4.0, - }
                  //定时器数组的最后一个元素 t_Mytimer2[4]没有被赋值
```

定时器只能在函数和可选步中作为参数传递，传入函数和可选步的定时器在这些函数和可选步的行为定义中是已知的。

作为参数传递的定时器的使用与其他定时器一样，即它不必被声明。一个已经启动了的定时器也可以传递给函数和可选步，这时，该定时器仍继续运行，也就是说它不会在暗中被中断。因此，被传入定时器的函数和可选步能够在其内部处理可能发生的超时事件。

例 3：

```
//在形参列表中带有一个定时器声明的函数
function MyBehaviour (timer MyTimer)
{   :
    MyTimer.start;
    :
}

//在形参列表中带有一个定时器声明的函数
function MyBehaviour (timer MyTimer)
{   :
    MyTimer.start;
    :
}
```

5.4 消息声明

TTCN-3 的一个关键元素就是在测试配置中定义的通信端口上收发复杂消息的能力。这些消息可以是与 SUT 明确相关的消息，也可以是与特定测试配置相关的内部协调或控制消息。

注意：在 TTCN-2 中，这些消息是抽象服务元语 ASP(Abstract Service Primitives)、协议数据单元 PDUs(Protocol Data Units)和协调消息(co-ordination messages)。而在 TTCN-3 核心语言中，消息没有任何语法和语义的区别，从这个意义上讲是通用的。

5.5　过程特征声明

在基于过程的通信中，过程特征(Procedure Signatures，简称特征)是必需的。基于过程的通信可以用于测试系统(Test System)内部的通信，即测试成分之间或测试系统与 SUT 之间的通信。在后面的章节中，一个过程可以在被测系统 SUT 中被调用(invoke，即测试系统执行这个调用)或在测试系统中被调用(invoke，即被测系统 SUT 执行这个调用)。对于所有使用到的过程(即用于测试成分之间通信的过程、被测系统 SUT 调用的过程和测试系统调用的过程)，都要在 TTCN-3 的模块中定义完整的过程特征特性(Procedure Signature)。

5.5.1　阻塞的和非阻塞的通信中的过程特征

TTCN-3 支持基于过程的阻塞的(Blocking)和非阻塞的(Non-Blocking)的通信。对于非阻塞的(NonBlocking)通信中的特性(Signature)使用关键字 noblock，只有 in 参数，且没有返回值，但可能出现例外(exceptions)。没有 noblock 关键字的特性默认为阻塞的基于过程的通信。

例：

```
signature MyRemoteProcOne (); //MyRemoteProcOne 将用于阻塞的基于过程的通信
//它既没有参数也没有返回值
signature MyRemoteProcTwo() noblock;
//MyRemoteProcTwo 将用于非阻塞的基于过程的通信
//它既没有参数也没有返回值
```

5.5.2　过程信号的参数

过程特征定义中可以带有参数。在一个 signature 定义中，参数列表可以包括参数标识符、参数类型和它们的方向，即输入(in)、输出(out)或输入输出(inout)。方向输入/出(inout)和输出(out)表示这些参数用于远程过程(Remote Procedure)重新获得信息。值得注意的是，参数的方向指的是从被调用方(Called Party)来看的，而不是调用方(Calling Party)。

例：

```
signature MyRemoteProcTh ree (in integer Par1, out float Par2, inout integer Par3);
//MyRemoteProcThree 将用于阻塞的基于过程的通信
//这个过程有三个参数：Par1 是整型输入参数
//Par2 是一个浮点型输出参数，Par3 是一个整型的输入/出参数
```

5.5.3　远程过程的返回值

一个远程过程可以在其终止后返回一个值返回值的类型要通过相应的 signature 定义中的 return 子句来指明。

例：

```
signature MyRemoteProcFour (in integer Par1) return integer;
//MyRemoteProcFour 将用于阻塞的基于过程的通信，它有一个整型输入参数 Par 1，在过程
//终止后返回一个整型值
```

5.5.4 例外描述

在 TTCN-3 中，用一种特定类型的值来表示可能在远程过程中出现的例外(Exceptions)。因此，可以使用模板机制和匹配机制来描述和检查远程过程的返回值。

注意：由于从被测系统 SUT 产生或发送到被测系统 SUT 的例外向相应的 TTCN-3 类型或 SUT 表示方法的转换属于工具和系统特性(system specific)类，因此它超出了本文的讨论范围。

例外是在 signature 定义中以例外列表的形式被定义的。这个列表定义了与可能出现的例外相关的所有可能的各种类型(在例外的含义中，例外通常只是通过它们自身类型的特定值来区分是否是例外)。

例：

```
signature MyRemoteProcFive (inout flo at Par1) return integer
exception (ExceptionType1, ExceptionType2);
//MyRemoteProcFive 将用于阻塞的基于过程的通信。它可以通过输入输出参数 Par1 返回一
//个浮点值和一个整型值，或是产生 ExceptionType1 类型或 ExceptionType2 类型的例外
signature MyRemoteProcSix (in integer Par1) noblock
exception (integer, float);
//MyRemoteProcSix 将用于非阻塞的基于过程的通信。在过程非成功终止的情况下，
//MyRemoteProcSix 可能会产生整型或浮点型例外
```

5.6 模板声明

模板(Templates)用于传送一个特定值的集合或是测试接收的值的集合是否与模板说明匹配。

模板具有下列特性。

(1)模板提供了一种组织和重复使用测试数据的方法，其中包括继承的简单形式。

(2)模板能够被参数化。

(3)模板允许匹配机制。

(4)模板既能用于基于消息的通信，也能用于基于过程的通信。

可以在一个模板的值集合中说明范围(ranges)和匹配属性，然后在基于消息的通信或基于过程的通信中中使用。模板可以用来说明任意 TTCN-3 类型或过程特征。基于类型的模板(Type-based Templates)用于基于消息的通信，而过程模板(Signature Templates)用于基于过程的通信。

一个模板声明必须详细说明一个基本值的集合，或是一个与在相应的类型或过程特征(Signature)中定义的每一个字段(field)相匹配的符号的集合，即必须完整地描述它。一个模板修改声明(Modified Template Declaration)仅说明原模板中要改变的字段即可，也就是说，它是一个对模板部分说明。在用于消息的模板说明中不能使用“未被使用符号”(“-”)，但“未被使用符号”(“-”)可以在过程特征模板的不相关参数和所有模板修改声明中用于指明特定字段或元素保持不变。

5.6.1 消息模板声明

可以用模板来说明带有实际值的消息实例，一个模板可以被作为创建一个发送消息或匹配一个接收消息的指令集合。

可以定义除特定配置和默认类型(port, component, address 和 default 类型)外的任意 TTCN-3 类型说明模板。

例 1：

```
//用于接收操作时，该模板将匹配任意整型值
template integer Mytemplate:= ?;
//该模板仅匹配整型值 1、2、3
template integer Mytemplate:= (1, 2, 3);
```

在一个发送操作中使用的模板定义一个完整的字段值集合，其中包含在测试端口上传输的消息。用于发送消息的模板在执行发送操作(send)时会被完全定义，也就是说，所有字段都会解析到实际值，而且在模板的字段中不会直接或间接地使用匹配机制。

注意：对于用于发送消息的模板来讲，省略一个可选择字段会被看成一个值符号(value notation)，而不是一个匹配机制。

例 2：

```
//给定的消息的定义
type record MyMessageType
{
    integer field1 optional,
    chars tring fie ld2,
    boolean field3
}
//一个消息模板可以是
template MyMessageType MyTemplate:={
    field1:= omit,
    field2:= "My string",
    field3:= true
}
//一个相应的发送操作可以是
MyPCO.send(MyTemplate);
```

用于接收操作(receive)的模板定义了一个与接收消息相匹配的数据模板，用于接收的模板可以使用匹配机制，且接收值与该模板不绑定。

例 3：

```
//给出的消息的定义
type record MyMessageType
{
    integer field1 optional,
    charstring field2,
    boolean field3
}
//一个消息模板可以是
template MyMessageType MyTemplate:=
{
    field1:= ?,
    field2:= pattern "abc*xyz",
    field3:= true
}
//一个相应的接受操作可以是
MyPCO.receive(MyTemplate);
```

5.6.2 过程信号模板声明

可以使用模板说明带有实际值的过程参数列表的实例，可以通过引用相关的过程特征定义来为任意过程定义模板。

例 1：

```
//一个远程过程的过程特征定义
signature RemoteProc(in integer Par1, out integer Par2, inout integer Par3)
                        return integer;
//与定义的过程特征相关的模板例子
template RemoteProc Template1:=
{
    Par1:= 1,
    Par2:= 2,
    Par3:= 3
}
template RemoteProc Template2:=
{
    Par1:= 1,
    Par2:= ?,
    Par3:= 3
}
template RemoteProc Template3:=
{
    Par1:= 1,
    Par2:= ?,
    Par3:= ?
}
```

用于 call 或 reply 操作的模板为所有的输入(in)参数和输入/出(inout)参数定义了一个完整的字段值集合。在 call 操作中，模板中的所有的 in 和 inout 参数将解析到实际的值，在这些字段中不会直接或间接地使用匹配机制。任意 out 参数的模板说明都被简单地忽略掉了，因此允许为这些字段说明匹配机制或省略匹配机制。

例 2：

```
//在 14.2 介绍中给定的例子
//有效的调用，因为所有的 in 和 inout 参数都具有一个确切的值
MyPCO.call(RemoteProc:Template1);
//有效的调用，因为所有的 in 和 inout 参数都具有一个确切的值
MyPCO.call(RemoteProc:Template2);
//无效的调用，因为 inout 参数 Par3 具有的是匹配属性，而不是一个确切的值
My PCO.call(RemoteProc:Template3);
//模板从不返回值，对于 Par2 和 Par3 的情况，必须使用在调用语句的末尾处的赋值子句
//重新获得通过调用操作返回的值
```

用于 getcall 操作的模板定义了一个与接收参数字段相匹配的数据模板。匹配机制可能用于任意被 getcall 操作使用的模板，但是不会发生接收值与模板的绑定。在匹配过程中，任意 out 参数都将被忽略。

例 3：

```
//有效的 getcall，如果 Par1 == 1 且 Par3 == 3，则匹配
MyPCO.getcall(RemoteProc:Template1);
//有效的 getcall，如果 Par1==1 且 Par3==3，则匹配
MyPCO.getcall(RemoteProc:Template2);
//有效的 getcall，当 Par1==1 且 Par3 为任意整型值时都将匹配
MyPCO.getcall(RemoteProc:Template3);
```

5.6.3　模板匹配机制

一般来讲，匹配机制用来替换单个模板字段的值或是整个模板的内容，其中一些匹配机制可以联合使用。

匹配机制和通配符也可以嵌入在接收事件(如 receive, getcall, getreply 和 catch 操作)语句行中使用，它们可以出现在明确的模板字段值中。

例 1：

```
MyPCO.receive(charstring:"abcxyz");
MyPCO.receive (integer:complement(1, 2, 3));
```

当模板字段值明确地标识了类型时，类型标识符可以省略。

例 2：

```
MyPCO.receive('AAAA'O);
```

注意：下列类型可以被省略：integer, float, boolean, objid, bitstring, hexstring, octetstring。

然而，嵌入式模板(in-line template)的类型应该在接收该模板所通过的端口的列表中。在列出的类型和模板字段值出现分歧的情况下(例如由子类型造成的)，类型名应该包含在接收语句中。

匹配机制分为以下四类。

(1)特定的字段值。

- 求特定值的表达式。
- omit：字段值将被省略。

(2)可以用来代替字段值的特殊符号。

- (…)：值列表。
- complement (…)：值列表的补集(Complement)。
- ?：表示任意值的通配符。
- *：表示任意值或空的通配符(例如一个省略值)。
- (下限 to 上限)：上下限范围内的整型值域，包括上下限。

(3)可以作为内部值(Inside Value)的特殊符号。

- ?：表示串、数组、record of 或 set of 中任意单个元素的通配符。
- *：表示串、数组、record of 或 set of 中任意数目的连续元素或根本为空(例如一个省略的元素)。

(4)描述字段值属性的特殊符号。

- length：用于串类型的串长度限制以及 record of, set of 和数组的元素数目的限制。
- ifpresent：用于匹配没有省略的可选择字段值。

TTCN-3 匹配机制如表 5-1 所示。

表 5-1 TTCN-3 匹配机制

Used with values of	值		代替值							内部值		属性	
	特定值	省略值	补集	值列表	任意值(?)	任意值或空(*)	值域	超集	子集	任意元素(?)	任意元素或空(*)	长度限制	ifpresent
boolean	是	是	是	是	是	是							是
Integer	是	是	是	是	是	是	是						是
Char	是	是	是	是	是	是	是						是
universal char	是	是	是	是	是	是	是						是
Float	是	是	是	是	是	是	是						是
bitstring	是	是	是	是	是	是				是	是	是	是
octetstring	是	是	是	是	是	是				是	是	是	是
hexstring	是	是	是	是	是	是				是	是	是	是
character strings	是	是	是	是	是	是	是			是	是	是	是
Record	是	是	是	是	是	是							是
Record of	是	是	是	是	是	是				是	是	是	是
Array	是	是	是	是	是	是				是	是	是	是
Set	是	是	是	是	是	是							是
set of	是	是	是	是	是	是		是	是	是	是	是	是
enumerated	是	是	是	是	是	是							是
Union	是	是	是	是	是	是							是

5.6.4 模板参数化

用于发送和接收操作的模板都可以被参数化。一个模板的实参可以包括字段值和模板、函数和特定的匹配机制。

例 1：

```
//模板
template MyMessageType  MyTemplate (integer MyFormalParam) :=
{
    field1  :=  MyFormalParam,
    field2  :=  pattern "abc*xyz",
    field3  :=  true
}
//可以以如下方式被使用
Pco1.send(MyTemplate(123));
```

为了使模板或匹配符号可以作为参数传送，需要在类型字段前加外部关键字 template。这样做使得一个参数类型为模板类型，并有效地扩展相关类型所容许的参数，使其包括正常值集和适当的匹配属性集合。模板参数字段不能通过传参来调用。

例 2：

```
//模板
template MyMessageType  MyTemplate (template integer MyFormalParam)    :=
{
    field1  :=  MyFormalParam,
```

```
    field2  :=  pattern "abc*xyz",
    field3  :=  true
}
//可以以如下方式使用
pco1.receive ( MyTemplate (?) );
//或
pco1.receive ( MyTemplate (omit) );
```

5.6.5　作为参数传递模板

只有函数(**function**)、测试用例(**testcase**)、可选步(**altstep**)和模板(**template**)定义中可以带有作为形参的模板。

例：

```
function MyBehaviour ( template MyMsgType MyFormalParameter ) runs on MyComponentType
{
    :
    pco1.receive(MyFormalParameter);
    :
}
```

5.6.6　修改模板

通常，一个模板描述了一个基本的或默认的值的集合，或定义在适当的定义中用于每个匹配符的值集。在仅有少量变化而需要去说明一个新的模板的情况下，我们可以说明一个修改模板(Modified Template)。一个修改模板说明了对原始模板的特定字段的直接或间接的更改。

用关键字 modifies 来表示修改模板所源于的父模板。这个父模板可以是一个原始模板，也可以是一个修改模板。

更改以链接的方式发生，最终要追溯到原始模板。如果一个模板字段和该字段相应的值或匹配符号在修改模板中没有被说明，那么将使用父模板中该字段值或匹配符号。当被修改的字段嵌套在一个结构类型的模板字段中时，除了这个被明确说明的字段外，父模板中相应的结构类型字段中的其他字段都保持不变。

无论是直接还是间接方式，修改模板都不能自己调用自己，也就是说不允许递归调用。

例 1：

```
//给定
template MyRecordType MyTemplate1:=
{
    field1  :=  123,
    field2  :=  "A string",
    field3  :=  true
}
//再定义
template MyRecordType MyTemplate2 modifies MyTemplate1:=
{
    field1  :=  omit,       //field1 是可选项,但在 MyTemplate1 中则要定义清楚
    field2  :=  "A modified string"     //field3 没有变化
}
```

```
//也可以写为
template MyRecordType MyTemplate2:=
{
    field1  :=  omit,
    field2  :=  "A modified string",
    field3  :=  true
}
```

如果一个基础模板有一个形参列表，则下列规则将应用于所有该基础模板所派生出的修改模板(可经过一个或多个修改步骤产生)。

(1)派生模板(Derived Template)不能省略从基础模板到实际的修改模板的任何一个修改步骤中定义的参数。

(2)如果需要，派生模板可以拥有额外的(附加的)参数。

(3)对每一个修改模板，形参列表都要跟在模板名称之后。

(4)在修改模板中，包含有参数化模板的基础模板的字段不能被修改或明确地被表示为省略字段。

例 2：

```
//给定
template MyRecordType MyTemplate1(integer Mypar):=
{
    field1  :=  MyPar,
    field2  :=  "A string",
    field3  :=  true
}
//则可以有如下改变
template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1:=
{
    //在 Template1 中，field1 是参数化模板，因而在 Temp late2 中，field1 仍是
    field2  :=  "A modified string",
}
```

与明确地命名修改模板一样，在 TTCN-3 中还允许定义嵌入式修改模板。

例 3：

```
//给定
template MyMessageType Setup:=
{
    field1  :=  75,
    field2  :=  "abc",
    field3  :=  true
}
//可以定义 Setup 的一个嵌入式修改模板
pco1.send ( modifies Setup  :=  {field1 :=  76} );
```

5.6.7 改变模板字段

在通信操作(如 **send**, **receive**, **call**, **getcall** 等)中，只允许通过参数化或嵌入式派生模板来改变模板字段。在该模板随后的相关通信事件中，这些改变对模板字段值将不会造成影响。

在通信事件中，不使用带点的表示符号 MyTemplateId.FieldId 设置或重新得到模板的值，而使用“->”符号。

5.6.8　匹配操作

匹配(**Match**)操作允许比较模板值与变量值，该操作返回一个布尔值。如果模板值与变量值类型不兼容，则该操作返回假(**false**)。如果类型值项匹配，则该操作的返回值指出变量值与给定模板值是否一致。

例：

```
template integer LessThan10    :=   ( -infinity  ..  9 );
testcase TC001()
runs on MyMTCType
{
    var integer     RxValue;
    :
    PCO1.receive(integer:?) -> value RxValue;
    if( match( RxValue, LessThan10)) { … }
    //如果 Rxvalue 的实际值小于 10，则返回真(true)，否则返回(false)
    :
}
```

5.6.9　操作的值

valueof 操作允许把模板中指定的值赋值给变量的字段，但该变量和模板字段的类型应该是兼容的，模板的每个字段可以解析为一个单个的值。

例：

```
type record ExampleType
{
    integer         field1,
    boolean         field2
}
template ExampleType SetupTemplate:=
{
    field 1     :=  1,
    field2      :=  true
}
:
var ExampleType     RxValue:= valueof( SetupTemplate);
```

思考题

1．定义一个 5.0s 的定时器，定时器的名称为 Timer。

2．模板参数化与变量赋值有什么区别？

3．给出使用远程过程返回值的具体步骤。

第 6 章　语句、函数、可选步与通信

本章介绍 TTCN-3 核心语言中的语句和函数，其中包括基本程序语句、行为语句、函数和可选步。

6.1　程序语句和操作

TTCN-3 模块的基本程序元素是基本的程序语句(如表达式、赋值、循环构造(**Loop Constructs**)等)、行为语句(如顺序行为、选择对象行为、交叉、默认等)和操作(如 **send**、**receive**、**create** 等)。

语句既可以是单一语句(不包括其他程序语句)，也可以是复合语句(可以包含其他语句或语句和声明块)。

语句将按照它们出现的先后顺序执行，也就是说，按如图 6-1 所示顺序执行。

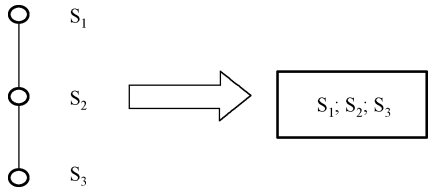


图 6-1　顺序执行图示

在语句序列中，用分隔符“;”来分隔各个单独的语句。例：

```
MyPort.send(Mymessage);
MyTimer.start;
log("Done!");
```

对一个语句和声明空块的说明，即{}，可以出现在复合语句中。例如，语句中的一个分支意味着采取空动作，即什么都不做。

TTCN-3 语句和操作纵览如表 6-1 所示。

表 6-1　TTCN-3 语句和操作纵览

语句(Statement)	相关的关键字或符号(Associated keyword or symbol)	能用在模块控制中(Can be used in module control)	能用在函数、测试用例和可选步中(Can be used in functions, test cases and altsteps)
基本程序语句(Basic program statements)			
表达式(Expressions)	(…)	是	是
赋值(Assignments)	:=	是	是
日志(Logging)	log	是	是
标记和跳转(Label and Goto)	label / goto	是	是
条件(If -else)	if (…) {…} else {…}	是	是
For 循环(For loop)	for (…) {…}	是	是
While 循环(While loop)	while (…) {…}	是	是
Do while 循环(Do while loop)	do {…} while (…)	是	是
停止执行(Stop execution)	stop	是	是
行为程序语句(Behavioural program statements)			
选择对象行为(Alternative behaviour)	alt {…}	是	是
重新确定选择对象行为(Re-evaluation of alternative behaviour)	repeat	是	是
交叉行为(Interleaved behaviour)	interleave {…}	是	是
返回控制(Returning control)	return		是

续表

语句（Statement）	相关的关键字或符号（Associated keyword or symbol）	能用在模块控制中（Can be used in module control）	能用在函数、测试用例和可选步中（Can be used in functions, test cases and altsteps）
用于默认处理的语句（Statements for default hanling）			
激活一个默认（Activate a default）	activate	是	是
停用一个默认（Deactivate a default）	deactivate	是	是
配置操作（Configuration operations）			
创建并行测试成分（Create parallel test component）	create		是
连接两个成分	connect		是
断开两个成分	disconnect		是
映射端口到测试接口	map		是
从测试系统接口取消端口的映射	unmap		是
获得 MTC 地址	mtc		是
获得测试系统接口地址	system		是
获得自身地址	self		是
开始测试成分的执行	start		是
停止测试成分的执行（Stop execution of test component）	stop		是
检查一个 PTC 的终止	running		是
等待一个 PTC 的终止	done		是
通信操作（Communication operations）			
发送消息	send		是
调用过程调用	call		是
从远程实体回答过程调用	reply		是
提出例外（对一个已被接收的调用）[Raise exception (to an accepted call)]	raise		是
接收消息	receive		是
触发消息（Trigger on message）	trigger		是
从远程实体接收过程调用	getcall		是
处理来自以前调用的响应	getreply		是
抓住例外（从被调用实体）[Catch exception (from called entity)]	catch		是
检查（当前）消息/接收到的调用[Check (current) message/call received]	check		是
清除端口（Clear port）	clear		是
Clear and give access to port	start		是
Stop access (receiving & sending) at port	stop		是
定时器操作（Timer operations）			
启动定时器	start	是	是
停止定时器	stop	是	是
读取经过的时间（Read elapsed time）	read	是	是
检查定时器是否运行（Check if timer running）	running	是	是
超时事件（Timeout event）	timeout	是	是
判定操作（Verdict operations）			
置本地判定（Set local verdict）	setverdict		是
获得本地判定（Get local verdict）	getverdict		是
外部事件（External actions）			
外部模拟一个（SUT）活动[Stimulate an (SUT) action externally]	action	是	是
测试用例执行（Execution of test cases）			
执行测试用例	execute	是	是
注意 1：仅能用于控制定时器操作。 注意 2：仅能用于模块控制中的函数和可选步。			

6.2 基本的程序语句

基本的程序语句是指表达式、操作、循环构造(**loop constructs**)等。所有基本程序语句都可以用在模块的控制部分和 **TTCN-3** 的函数、可选步和测试用例中。

TTCN-3 基本的程序语句纵览如表 6-2 所示。

表 6-2 TTCN-3 基本的程序语句纵览

基本程序语句	
语句	相关的关键字或符号
Expressions	(…)
Assignments	:=
Logging	log
Label and Goto	label / goto
If-else	if (…) { … } else { … }
For loop	for (…) { … }
While loop	while (…) { … }
Do while loop	do { … } while (…)
Stop execution	stop

6.2.1 表达式

TTCN-3 中的表达式是由其他(简单)表达式构建的，可以只使用返回值的函数。一个表达式的结果应该是一个特定类型值，且操作符应该和操作数类型兼容。

例 1：

```
( x + y - increment( z ) ) * 3;
```

一个布尔表达式应该仅包含布尔值和/或布尔操作符以及/或关系运算符，并计算一个布尔值，即 **true** 或 **false**。

例 2：

```
( ( A and B) or (not C) or (j<10) );
```

6.2.2 赋值

值可以赋给变量，用符号“:=”表示。在执行赋值的过程中，“:=”的右边用于计算出左边相同类型的元素值。赋值的结果就是把一个变量和一个表达式的值绑定，这个表达式不应该包含没有绑定的变量。所有赋值的发生顺序按照它们的出现顺序进行，也就是说，按从左至右处理。

例：

```
MyVariable := ( x + y - increment( z ) ) * 3;
```

6.2.3 日志语句

日志语句(**log**)提供了向有关测试控制设备或使用该语句的测试成分写字符串的方法。

例：

```
log("Line 248 in PTC_A");
//把串"Line 248 in PTC_A"写到测试系统的一些日志设备中
```

注意：定义复杂的日志和可能依赖工具的回溯能力，这超出了本文的范围。

6.2.4　标签语句

标签语句(**label**)允许在测试用例、函数、可选步和模块控制部分中对标签进行说明，可以根据定义的语法规则，像其他 **TTCN-3** 行为的程序语句一样自由地使用标签语句。标签语句可以在一个 TTCN-3 语句前或后使用，但是不能作为一个选择对象的第一个语句，或是一个 **alt** 语句、**interleave** 语句或可选步中顶层的选择对象。用在关键字之后的标签在同一个测试用例、函数、可选步或控制部分中的所有标签中应该是唯一的。

例：

```
label       MyLabel;
//定义标签 MyLabel
//在下面的 TTCN-3 代码段中定义标签 L1、L2 和 L3
:
label       L1;
//标签 L1 的定义
alt{
    [] PCO1.receive(MySig1)
    {
        label   L2;                  //标签 L2 的定义
        PCO1.send(MySig2);
        PCO1.receive(MySig3)
    }
    [] PCO2.receive(MySig4)
    {
        PCO2.send(MySig5);
        PCO2. send(MySig6);
        label L3;                    //标签 L3 的定义
        PCO2.receive(MySig7);
    }
    goto    L1;                      //跳到标签 L1
}
:
```

6.2.5　Goto 语句

goto 语句可以用在函数、测试用例、可选步和 **TTCN-3** 模块的控制部分中，它执行一个跳转，跳转到一个标签(**label**)。

goto 语句提供自由跳转(即在一个语句序列里向前和向后)、跳出一个单个的复合语句(如一个 **while** 循环)和跳过嵌套的复合语句数层的可能性。然而，**goto** 语句的使用将受到下面规则的限制。

(1)不允许使用 **goto** 语句跳出或跳入函数、测试用例、可选步和 **TTCN-3** 模块的控制部分。

(2)不允许使用 **goto** 语句跳入在一个复合语句(即 **alt** 语句、**while** 循环、**for** 循环、**if-else** 语句、**do-while** 循环和 **interleave** 语句)中定义的语句序列。

(3)不允许在一个 **interleave** 语句中使用 **goto** 语句。

例：

```
//下面的 TTCN-3 代码段包括
:
```

```
label   L1;                                  //… 标签 L1 的定义
MyVar   :=  2 * MyVar;
if (MyVar < 2000) { goto L1; }               //… 跳回到 L1
MyVar2  :=  Myfunction(MyVar);
if (MyVar2 > MyVar){ goto L2; }              //… 向前跳到 L2
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;                                    //… 标签 L2 的定义
PCO2.send(integer: 21);
alt {
    [] PCO1.receive { }
    [] PCO2.receive(integer: 67) {
        label       L3;                      //… 标签 L3 的定义
        PCO2.send(MyVar);
        alt {
            [] PCO1.receive { }
            [] PCO2.receive(integer: 90) {
                PCO2.send(integer: 33);
                PCO2.receive(integer: 13);
                goto    L4;                  //… 向前跳出两层嵌套 alt 语句
            }
            [] PCO2.receive(MyError) {
                goto    L3;                  //… 跳回到当前的 alt 语句
            }
            [] any port.receive {
                goto    L2;                  //… 向后跳出两层嵌套 alt 语句
            }
        }
    }
    [] any port.receive {
        goto    L2;                          //… 一个大的回跳，跳出 alt 语句
    }
}
label       L4;
:
```

6.2.6 If-else 语句

if-else 语句，也是通常所说的条件语句，用于表示布尔表达式导致的控制流中的分支，其用法如下：

if (表达式 1)

　　语句块 1

else

　　语句块 2

语句块 x 处引用一个语句块。

例：

```
    if (date == "1.1.2000") return { fail };
```

```
if (MyVar < 10) {
    MyVar :=    MyVar * 10;
    log ("MyVar < 10");
}
else{
    MyVar :=    MyVar/5;
}
```

一个更复杂的方案可以是：

```
if (expression1)
    statementblock1
else if (expression2)
    statementblock2
:
else if (expressionn)
    statementblockn
else
    statementblockn+1
```

在这种情况下，可读性主要依赖于格式，但格式不会具有语法或语义的意义。

6.2.7　for 语句

for 语句定义了计数循环，增加、减少下标变量的值，或操纵下标变量的值使其在执行一定次数的循环后到达终止条件。

for 语句包含两个赋值和一个布尔表达式。第一个赋值给初始化循环下标(**index**，或计数器)变量，布尔表达式终止这个循环，而第二个赋值用于控制该下标变量。

例 1：

```
for (j:=1; j<=10; j:= j+1) { … }
```

循环的终止条件应该由布尔表达式表示，在每次新的循环反复的开始时检查该布尔表达式。如果表达式值为 **true**，则继续执行 **for** 循环语句后面的语句。

可以在 **for** 语句使用下标变量前声明该 **for** 循环语句的下标变量，也可以在 **for** 语句头部中声明并初始化该下标变量。如果该下标变量是在 **for** 语句头部中声明并初始化的，那么它的作用范围仅限于该循环体内，即它仅在该循环体内是可见的。

例 2：

```
var integer     j;                                  //整型变量 j 的声明
for (j:=1;   j<=10; j:= j+1) { … }                  //作为循环的下标变量的 j 的用法
for (var float   i:=1.0;     i<7.9; i:= i*1.35) { … }
                  //在循环的头部声明和初始化下标变量 i，变量 i 仅在循环体内是可见的
```

6.2.8　While 语句

只要满足循环条件，就一直执行 **while** 循环。在每次新的循环反复的开始时检查循环条件。如果不满足循环条件了，那么退出该循环，继续执行紧跟在 **while** 循环后的语句。

例：

```
while (j<10)    { … }
```

6.2.9 do-while 语句

do-while 循环与 **while** 循环类似，只是 **do-while** 循环在每次循环反复结束时检查循环条件，这就意味着如果使用 **do-while** 循环，在第一次对循环条件求值之前，循环体的行为至少被执行一次。

例：

```
do { … } while (j<10);
```

6.2.10 停止执行语句

根据使用 **stop** 语句的环境不同，**stop** 语句以不同的方式终止执行操作。如果在一个模块的控制部分或模块控制部分使用的函数中使用 **stop** 语句，则它终止该测试执行。如果在一个测试成分上执行的测试用例、可选步或在函数中使用 **stop** 语句，则它终止相关的测试成分。

例：

```
module MyModule {
    :                   //模块定义
    Testcase MyTestCase()  runs on   MyMTCType   system  MySystemType
    {
        :
        stop            //停止一个测试成分
    }
    control {
        :               //测试执行
        stop            //停止测试活动
    }                   //结束控制
}                       //结束模块
```

注意：停止一个测试成分的 stop 语句的语义与停止成分操作 **self.stop** 的语义相同。

6.3 行为的程序语句

行为的程序语句可以用在测试用例、函数和可选步中，除了下列情况：

(1) 仅用在函数中的 **return** 语句中。

(2) 也可用在模块控制中的 **alt** 语句、**interleave** 语句和 **repeat** 语句中。

行为的程序语句明确地描述了通过通信端口上的测试成分的动态行为，可以作为一个选择对象集或它们的组合来顺序地表达测试行为。一个交叉操作符允许对交叉序列或选择对象的说明。

TTCN-3 行为的程序语句纵览如表 6-3 所示。

表 6-3 TTCN-3 行为的程序语句纵览

行为的程序语句	
语句	相关的关键字或符号
选择性行为	alt { … }
alt 语句的重新求值	repeat
交叉行为	interleave { … }
返回控制	return

6.3.1 选择性行为

行为的一个更为复杂的形式是，语句序列被表示成形成执行路径树的可能的选择对象的集合，如图 6-2 所示。

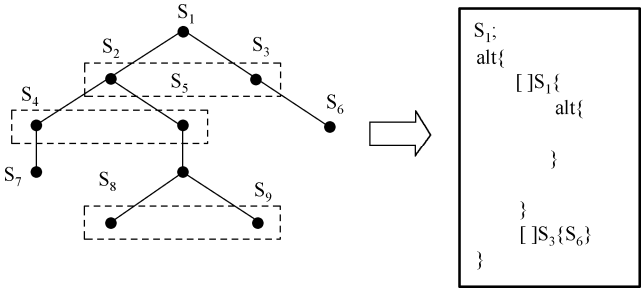


图 6-2　选择性行为的例示

alt 语句表示由通信的接收和处理和/或定时器事件以及/或并行测试成分终止引起的测试行为的分支，也就是说，**alt** 语句与 **TTCN-3** 的 **receive**、**trigger**、**getcall**、**getreply**、**catch**、**check**、**timeout** 和 **done** 操作有关。**alt** 语句表示将与一个特定快照相匹配的可能的事件集合。

注意：alt 语句与 **TTCN-2** 中缩进排列的选择对象相似，但有明显的区别，例如:

(1)不能在通过使用布尔表达式来检查端口队列之后使一个选择对象无效。

(2)在 **alt** 语句中，函数不能作为一个选择对象被调用，除非在一个 **else** 防卫表达式(**else guard**，即[**else**])是选择对象中最后的那个选择对象的情况下。

例 1：

```
//嵌套的选择对象语句的使用
:
alt {
    [] L1.receive(DL_REL_CO:*) {
        setverdict(pass);
        TAC.stop;
        TNOAC.start;
        alt {
            [] L1.receive(DL_EST_IN) {
                TNOAC.stop;
                setverdict(pass);
            }
            [] TNOAC.timeout {
                L1.send(DEL_EST_RQ:*);
                TAC.start;
                alt {
                    [] L1.receive(DL_EST_CO:*) {
                        TAC.stop;
                        setverdict(pass)
                    }
                    [] TAC.timeout {
                        setverdict(inconc);
                    }
                    [] L1.receive {
                        setverdict(inconc)
                    }
                }
```

```
                }
                [] L1.receive {
                    setverdict(inconc)
                }
            }
        }
        [] TAC.timeout {
            setverdict(inconc)
        }
        [] L1.receive {
            setverdict(inconc)
        }
    }
    :
```

进入一个 **alt** 语句时，照一张快照。快照被认为是一个测试成分的部分状态。这个测试成分包括计算防卫选择对象分支的布尔条件的所有必需信息，所有有关的被终止的测试成分，所有相关的超时事件和相关的输入端口队列的队头消息、调用、应答以及例外。认为在 **alt** 语句中至少一个选择对象中被引用的任意测试成分、定时器和端口，或者在 **alt** 语句中作为一个选择对象被调用的测试步或作为默认被激活的测试步的顶层选择对象是相关的。

注意 1：TTCN-3 语义假设照快照是瞬时的，即没有延迟。在实际的实现中，照快照可能占用一些时间，可能出现竞态条件(**race conditions**)。这些竞态条件的处理超出了本标准的范围。

alt 语句中的选择对象分支和被调用可选步中的顶层选择对象以及被激活为默认的可选步的最上层选择对象是按照它们的出现顺序被处理的。如果激活了数个默认，则激活顺序决定了确定默认中的最上层选择对象的值的顺序。

各选择对象分支是可以被一个布尔表达式或一个 **else** 分支防卫的分支，即选择对象分支用[**else**]开始在到达 **else** 分支时，总是选中和执行它们。

可被一个布尔表达式防卫的分支或激活一个可选步(可选步分支，**altstep-branch**)，或以 **done** 操作(完成分支，**done-branch**)、**timeout** 操作(超时分支，**timeout-branch**)或接收操作(接收分支，**receiving-branch**)开始，即 **receive**、**trigger**、**getcall**、**getrepy** 或 **catch**。布尔防卫表达式的取值基于快照。如果没有定义布尔防卫或是其值为真，则认为通过了布尔防卫。按照以下的方式执行和处理分支。

(1)如果通过了布尔防卫，则选中一个可选步分支(**altstep-branch**)。一个可选步分支的选择引起对被引用的可选步的调用，即调用该可选步且在这个可选步中继续对快照求值。

(2)如果通过了布尔防卫，且如果指定的测试成分在快照的被停止成分的列表中，则选中一个完成分支(**done-branch**)。该选择引起 **done** 操作后的语句块的执行，而对 **done** 操作本身没有更进一步的影响。

(3)如果通过了布尔防卫，且如果指定的超时事件在快照的超时事件列表中，则选中一个超时分支(**timeout-branch**)。该选择引起指定的超时(**timeout**)操作的执行，也就是说从超时队列(**timeout-list**)中移去该超时事件，并执行 **timeout** 操作后的语句块。

(4)如果通过了布尔防卫，且如果快照中的一个消息(**messages**)、调用(**calls**)、应答(**replies**)或例外(**exceptions**)与接收操作的匹配标准相匹配，则选中一个接收分支(**receiving-branch**)。该选择导致执行接收操作，即从端口队列中移去相匹配的消息、调用、应答或例外，或许是把接收

到的消息赋值给一个变量，并执行接收操作后的语句块。在 trigger 操作的情况下，如果通过了布尔防卫但不符合匹配标准，也从端口队列中移去队头消息，在这种情况下不执行语句块中给定的选择对象。

注意 2：TTCN-3 语义描述了对作为测试成分的一系列不可以分割活动的快照的求值。该语义并不假设对快照求值没有延时。在对快照求值过程中，测试成分可以停止，定时器可以超时，新消息、调用、应答或例外可以进入该测试成分的端口队列。然而，这些事件并不改变实际的快照，因此快照的求值不考虑这些事件。

如果在 **alt** 语句中、被调用可选步的顶层选择对象或被激活的默认中没有选择分支被激活和执行，则应该再次执行 **alt** 语句，也就是说，照一个新的快照，并以新快照重复对选择分支的求值。这个重复求值过程应该继续直到选中和执行一个选择分支，或是该测试用例被另一个测试成分或测试系统停止。

如果测试成分被完全阻塞，测试用例应该停止并指示一个动态错误。这就意味着没有选择对象被选中，没有相关的测试成分、定时器在执行，所有的相关端口包含至少一条消息、调用、应答或例外。

注意 3：照完整快照和为所有选择对象求值的反复过程对于描述 **alt** 语句的语义只是一个概念上的方法。实现这个语义的具体算法超出了本标准的范围。

如果必要的话，可以利用置于选择对象方括号“[ ]”之间的一个布尔表达式去启动/禁止（**enable**/**disable**）一个选择对象。

对防卫一个选择对象的布尔表达式的求值可能有副作用。为了避免导致实际快照和成分状态之间不一致的这种副作用，应使用于初始化可选步内部定义的限制相同的限制。

方括号的开和关“[”“]”应在每个选择对象的开始处出现，即便它们之间为空。这不仅对可读性有帮助，而且对从语法上区分一个选择对象和另一个选择对象也是必要的。

例 2：

```
//带有布尔表达式的选择对象的使用(或防卫)
:
alt {
    [x>1] L2.receive{                    //布尔防卫/表达式
        setverdict (pass);
    }
    [x<=1] L2.receive {                  //布尔防卫/表达式
        setverdict(inconc);
    }
}
:
```

可以通过在选择对象开始处开和关括号之间包含关键字 **else**，定义 **alt** 语句的最后一个分支为一个 **else** 分支。这个 **else** 分支不应包含布尔表达式防卫的分支中允许的任何动作，即可选步调用、完成、超时或接收操作。如果这个 **else** 分支前（文本上顺序）没有处理过其他的选择对象，则总是执行这个 **else** 分支的语句块。

例 3：

```
//带有布尔表达式的选择对象的使用(或防卫)
:
```

```
alt {
    [x>1] L2.receive {
        setverdict(pass);
    }
    [x<=1] L2.receive {
        setverdict(inconc);
    }

    [else] {                    //else 分支
        MyErrorHandling();
        setverdict(fail);
        stop;
    }
}
:
```

应该注意，默认机制总是在所有选择对象的结束时才被调用。如果定义了一个 **else** 分支，将步调用默认机制，也就是说决不进入活动的默认部分。

注意 4：可以在可选步内使用 **else**。

注意 5：允许使用一个 **repeat** 语句作为 **else** 分支的最后一个语句。

可以使用一个 **repeat** 语句来描述对 **alt** 语句的重新求值。

例 4：

```
alt {
    [] PCO3.receive {
        count   :=  count + 1;
        repeat                  //repeat 的用法
    }
    [] T1.timeout { }
    [] any port.receive {
        setverdict(fail);
        stop;
    }
}
```

TTCN-3 允许在 **alt** 语句中作为选择对象调用可选步。

例 5：

```
:
alt {
    [] PCO3.receive { }
    [] AnotherAltStep();            //对作为 alt 语句的选择对象的可选步
                                    //AnotherAltStep 的显式调用
    [] MyTimer.timeout { }

}
:
```

6.3.2　repeat 语句

repeat 语句引起对 **alt** 语句的重新求值(**Re-evaluation**)，即照一个新的快照，并根据 **alt** 语句的选择对象的说明顺序来对这些选择对象求值。repeat 语句应仅作为 **alt** 语句中选择对象的最后一个语句，或者是可选步定义中顶层的选择对象的最后一个语句。

如果一个 **repeat** 语句作为 **alt** 语句中选择对象的最后一个语句，则该 **repeat** 语句引起一个新的快照和对该 **alt** 语句的重新求值。

例 1：

```
//alt 语句中 repeat 的用法
alt {
    [] PCO3.receive {
        count   :=   count + 1;
        repeat                          //repeat 的用法
    }
    [] T1.timeout { }
    [] any port.receive {
        setverdict (fail);
        stop;
    }
}
```

如果把 **repeat** 语句作为可选步定义中最上层的选择对象的最后一个语句，则该 **repeat** 语句引起一个新快照和对调用可选步的 alt 语句的重新求值。可以通过默认机制隐式调用该可选步，也可以在 **alt** 语句中显式调用该可选步。

例 2：

```
//可选步中 repeat 的用法
altstep     AnotherAltStep()    runs on     MyComponentType {
    [] PCO1.receive{
        setverdict(inconc);
        repeat                          //repeat 的用法
    }
    [] PCO2.receive {}
}
```

6.3.3　交叉的行为

interleave 语句允许描述 **done**、**timeout**、**receive**、**trigger**、**getcall**、**catch** 和 **check** 语句的交叉发生(**interleaved occurrence**)和处理。

包括通信操作的控制转移语句(**for**、**while**、**do-while**、**goto**、**activate**、**deactivate**、**stop**、**repeat**、**return**)作为选择对象的可选步的直接调用和用户定义函数的(直接和间接)调用，不应用在 **interleave** 语句中。此外，不允许用布尔表达式去防卫 **interleave** 语句的分支(即“[ ]”应该总是为空)，也不允许在交叉的行为中指定 **else** 分支。

交叉的行为总可以被一个等价的嵌套选择对象集合代替。

interleave 语句求值的规则如下。

(1)不论何时执行一个接收语句,直到到达下一个接收语句或交叉语句结束时才执行后面的非接收语句。

注意:接收语句是可以在选择对象集合中出现的 TTCN-3 语句,即 receive、check、trigger、getcall、getreply、catch、done 和 timeout。非接收语句指所有可用于 interleave 语句内的非控制转移语句。

(2)求值后继续照下一个快照。

例:

```
//下面的 TTCN-3 代码段
:
interleave {
    [] PCO1.receive(MySig1)
    {
        PCO1.send(MySig2);
        PCO1.receive(MySig3);
    }
    [] PCO2.receive(MySig4)
    {
        PCO2.send(MySig5);
        PCO2.send(MySig6);
        PCO2.receive(MySig7);
    }
}
:
//可以解释为下面代码段的简写
:
alt {
    [] PCO1.receive(MySig1)
    {
        PCO1.send(MySig2);
        alt {
            [] PCO1.receive(MySig3)
            {
                PCO2.receive(MySig4);
                PCO2.send(MySig5);
                PCO2.send(MySig6);
                PCO2.receive(MySig7)
            }
            [] PCO2 .receive(MySig4)
            {
                PCO2.send(MySig5);
                PCO2.send(MySig6);
                alt {
                    [] PCO1.receive(MySig3)
                    {
                        PCO2.receive(MySig7);
```

```
                        }
                        [] PCO2.receive(MySig7)
                        {
                            PCO1.receive(MySig3);
                        }
                    }
                }
            }
            [] PCO2.receive(MySig4)
            {
                PCO2.send(MySig5);
                PCO2.send(MySig6);
                alt {
                    [] PCO1.receive(MySig1)
                    {
                        PCO1.send(MySig2);
                        alt{
                            []PCO1.receive(MySig3)
                            {
                                PCO2.receive(MySig 7) ;
                            }
                            []PCO2.receive(MySig7)
                            {
                                PCO1.receive(MySig3);
                            }
                        }
                    }
                    [] PCO2.receive(MySig7)
                    {
                        PCO1.receive(MySig1);
                        PCO1.send(MySig2);
                        PCO1.receive(MySig3);
                    }
                }
            }
        }
        :
```

6.3.4　返回语句

return 语句终止函数的执行，并把控制返回到函数的调用点。**return** 语句可以选择是否与一个返回值关联。**return** 语句仅用在函数中。

例：

```
    function    MyFunction()    return  boolean {
        :
        if (date == "1.1.2000") {
```

```
            return false;              //在 1.1.2000 执行结束,并返回一个布尔值 false
        }
        :
        return true;                   //返回 true
    }

    function    MyBehaviour()    return verdicttype {
        :
        if (MyFunction()) {
            setverdict(pass);          //在 if 语句中 MyFunction 的使用
        }
        else {
            setverdict(inconc);
        }
        :
        return getverdict;             //判定的显式返回
    }
```

6.4 函数和可选步

在 **TTCN**-3 中，函数(**Functions**)和可选步(**Altsteps**)用于表示和构造测试行为、定义一个模块中的默认行为和组织计算。下面的章节对此做了详细的描述。

6.4.1 函数

在 **TTCN-3** 的一个模块里用函数来表达测试行为，组织测试执行或是计算，如计算一个单个值，对一个变量集合进行初始化，或者检查条件。函数可以返回一个值，用后面跟了类型标识符的关键字 **return** 来表示函数要返回一个值。当关键字 **return** 用在函数体中且返回类型定义在函数头中时，关键字 **return** 后要接一个值来表示该返回值，即其后接一个常数、变量引用或一个表示返回值的表达式。返回值的类型和返回类型应该是兼容的。函数体中的返回语句终止函数执行，并在函数调用处返回返回值。

例 1：

```
//不带参数的函数 MyFunction 的定义
function MyFunction() return integer
{
    return  7;           //函数终止时返回整型值 7
}
```

函数可以被定义在一个模块中，或被声明为在外部被定义的(external)。对于一个外部函数，在 TTCN-3 的模块中仅仅必须提供函数的接口。外部函数的实现在本文讨论范围之外。外部函数不允许包含端口操作。

```
external function MyFunction4() return integer;//返回一个整型值的不带参数的外部函数
external function InitTestDevices(); //一个在 TTCN-3 模块外仅有一个作用的外部函数
```

注意 1：**TTCN-3** 的函数代替 **TTCN-2** 中的测试步和测试套程序的定义，而外部函数代替

TTCN-2 中的测试套操作。非形式化的函数(**Informal functions**)可以被声明为带有说明性注释的外部函数，或通过使用带有注释的空的形式化的函数(**Formal functions**)声明为外部函数。

在一个模块中，可以使用程序语句和操作定义函数的行为。如果一个函数使用了一个成分类型定义中声明的变量、常量、定时器或端口，则应该在该函数头中使用关键字 **runs on** 来引用这个成分类型定义。这个规则的一个例外是该成分的作用范围内的信息是否被作为参数传入。

例 2：

```
function    MyFunction3()   runs on     MyPTCType {
                            //MyFunction3 不返回一个值，但是却使用端口操作
    var integer         MyVar := 5;
    PCO1.send(MyVar);       //通过引用一个成分类型来发送，
                            //且因此要求一个 runs on 子句来解析端口标识符
}
```

不带有 **runs on** 子句的函数，从不会调用一个函数或可选步，或者激活一个作为默认的带有局部 **runs on** 子句的可选步。

被测试成分操作 start 启动的函数总是有一个 **runs on** 子句，并且被认为是在该成分中被调用时启动，也就是说，它不具有局部意义。不过，在不带有 **runs on** 子句的函数中调用这个测试成分操作 start 是允许的。

注意 2：关于 runs on 子句的这个限制仅与函数和可选步有关，而与测试用例无关。

在 **TTCN-3** 模块控制部分中使用的函数不会有 **runs on** 子句，不过，它们被允许执行测试用例。

函数可以被参数化。

例 3：

```
function MyFunction2( inout integer MyPar1) {   //MyFunction2 不返回值
    MyPar1 := 10 * MyPar1;                      //但是改变作为形参传入的 MyPar1 的值
}
```

可以通过引用函数名并提供实参列表来调用函数。不返回值的函数应该直接调用，有返回值的函数既可直接调用，也可以在表达式中调用。

例 4：

```
MyVar   :=  MyFunction4();  //MyFunction4 返回的值被赋给 MyVar，
                            //返回值的类型与 MyVar 的类型必须相同
MyFunction2(MyVar2);        //MyFunction2 不返回值，带有实参 MyVar2 被调用，
                            //可以通过传参来传入 MyVar2
MyVar3  :=  MyFunction6(4)  +  MyFunction7(MyVar3);//在表达式中使用的函数
```

对于使用测试成分操作 start 来绑定到测试成分的函数。

TTCN-3 包含了许多使用前不需声明的预定义(内置，**built-in**)函数。

调用一个预定义函数时：

(1)实参的数目应该和形参的数目相同。

(2)每个实参应该确定其相应的形参类型元素的值。

(3)所有出现在实参列表中的变量都应该被绑定。

TTCN-3 预定义函数列表如表 6-4 所示。

表 6-4 TTCN-3 预定义函数列表

类别	函数	关键字
转换函数(Conversion functions)	转换 integer 值为 char 值	int2char
	转换 integer 值为 universal char 值	int2unichar
	转换 integer 值为 bitstring 值	int2bit
	转换 integer 值为 hexstring 值	int2hex
	转换 integer 值为 octetstring 值	int2oct
	转换 integer 值为 charstring 值	int2str
	转换 integer 值为 float 值	int2float
	转换 float 值为 integer 值	float2int
	转换 char 值为 integer 值	char2int
	转换 universal char 值为 integer 值	unichar2int
	转换 bitstring 值为 integer 值	bit2int
	转换 bitstring 值为 hexstring 值	bit2hex
	转换 bitstring 值为 octetstring 值	bit2oct
	转换 bitstring 值为 charstring 值	bit2str
	转换 hexstring 值为 integer 值	hex2int
	转换 hexstring 值为 bitstring 值	hex2bit
	转换 hexstring 值为 octetstring 值	hex2oct
	转换 hexstring 值为 charstring 值	hex2str
	转换 octetstring 值为 integer 值	oct2int
	转换 octetstring 值为 bitstring 值	oct2bit
	转换 octetstring 值为 hexstring 值	oct2hex
	转换 octetstring 值为 charstring 值	oct2str
	转换 charstring 值为 integer 值	str2int
	转换 charstring 值为 octetstring 值	str2oct
长度/大小函数(Length/size functions)	返回任意串类型的长度	lengthof
	返回一个 record, record of, template, set, set of or array 类型的元素个数	sizeof
出现(Presence/choice functions)	确定一个 record, record of, template, set or set of 类型是否可选	ispresent
	确定在一个 union 类型中做了哪些选择	ischosen
串函数(String functions)	返回输入串中与指定的模式描述匹配的部分	regexp
	返回输入串的指定部分	substr
其他函数	产生一个随机浮点数	rnd

6.4.2 可选步

TTCN-3 使用可选步(**Altsteps**)来描述默认行为或构造一个 **alt** 语句的选择对象(**Alternatives**)。可选步是与函数相似的范围单位。可选步主体定义了一个局部定义的可选集合和选择对象的集合，所谓的顶层选择对象(**top alternatives**)构成了可选步的主体。顶层选择对象使用与 alt 语句的选择对象的语法规则相同的语法规则。

使用程序语句和操作可以定义可选步的行为。如果一个可选步包含端口操作或使用成分变量、常数或定时器，则应该在该可选步头部使用关键字 **runs on** 来引用相关的成分类型。这个规则的一个例外是该可选步中使用的所有端口、变量、常量和定时器是否作为参数传入的情况。

例 1：

```
//给定
type component MyComponentType {
    var integer         MyIntVar    :=  0;
```

```
        timer       MyTimer;
        port    MyPortTypeOne   PCO1, PCO2;
        port    MyPortTypeTwo   PCO3;
    }
    //使用 PCO1、PCO2、MyIntVar 和 MyTimer of MyComponentType 的可选步的定义

    altstep     AltSet_A(in integer MyPar1)  runs on  MyComponentType {
        [] PCO1.receive(MyTemplate(MyPar1, MyIntVar) ) {
            setverdict(inconc);
        }
        [] PCO2.receive {
            repeat
        }
        [] MyTimer.timeout {
            setverdict(fail);
            stop
        }
    }
```

可选步可以调用函数和可选步，或作为默认来被激活。一个不带有 **runs on** 子句的可选步从不会调用带有 **runs on** 子句的函数、可选步或作为默认来激活可选步。

可选步可以被参数化。被作为默认来激活的一个可选步应该仅带有值参数，也就是说输入(**in**)参数。在 **alt** 语句中仅作为一个选择对象或在 **TTCN-3** 行为描述中仅作为独立(**stand-alone**)语句被调用的可选步可以有输入(**in**)、输出(**out**)和输入/出(**in**/**out**)参数。

可选步可以定义常量、变量和定时器的局部定义。应该在选择对象前定义该局部定义。

例 2：

```
    altstep  AnotherAltStep(in integer  MyPar1) runs on  MyComponentType {
        var integer         MyLocalVar  :=  MyFunction();       //局部变量
        const float         MyFloat         :=  3.41;           //局部常量
        [] PCO1.receive(MyTemplate(MyPar1, MyLocalVar) {
            setverdict(inconc);
        }
        [] PCO2.receive {
            repeat
        }
    }
```

通过调用值返回函数来进行局部定义的初始化可能会有副作用。为避免成分实际的快照(**Snapshot**)和其状态之间不一致的副作用，在一个局部定义的初始化过程中，不应调用下列操作：

(1) **done** 操作。

(2) 所有的端口操作，即 **start**(**port**)、**stop**(**port**)、**clear**、**send**、**receive**、**trigger**、**call**、**getcall**、**reply**、**getreply**、**raise**、**catch**、**check**。

注意 1：**done**、**start** (**port**)、**stop** (**port**)、**clear**、**receive**、**trigger**、**getcall**、**getreply**、**catch** 和 **check** 操作的执行可能导致与实际快照的冲突。它们的执行可能会从端口队列中删除信息，限制对端口队列的访问和/或在实际快照确定值的过程中导致新的快照。

注意 2：应该避免为了实现可读性而使用 **send**、**call**、**reply** 和 **raise** 操作，也就是说所有通信都应该明确，而不是作为通信中的副作用。

(3)定时器操作 **start(timer)**、**timeout** 和 **stop(timer)**。

注意 3：允许使用读定时器(**readtimer**)和定时器运行(**running**)操作。

注意 4：这些用于可选步中局部定义初始化的限制，与避免 **alt** 语句中或可选步中用来选择和取消选择的布尔表达式的副作用的限制相同。

可选步的调用总是与 alt 语句相关。该调用可以通过默认机制隐式地完成，或者通过在 **alt** 语句中的一个直接调用显式地完成。可选步的调用不导致新的快照，通过使用调用该可选步的 **alt** 语句的实际快照，来完对成可选步的顶层选择对象求值。

注意 5：如果在一个选定的顶层选择对象中，指定并进入了一个新 alt 语句，则会采纳可选步中的一个新快照。

对于借助于默认机制的可选步的一个隐式调用，在到达调用位置之前，必须通过 **activate** 语句将该可选步激活为一个默认。

例 3：

```
:
var default MyDefVarTwo  :=  activate(MySecondAltStep()); //一个可选步激活为默认
:
```

在 alt 语句中的可选步的显式调用看起来像作为选择对象的函数调用。

例 4：

```
:
alt{
    [] PCO3.receive {
        …
    }
    [] AnotherAltStep(); //作为一个 alt 语句的选择对象显式调用可选步
    [] MyTimer.timeout {}
}
```

在 **alt** 语句中显式调用可选步时，下一个要检查的选择对象是该可选步中的第一个选择对象。可以用与带有例外(进入该可选步时没有照新快照)的 **alt** 语句的选择对象一样的方式来检查和执行该可选步选择对象。可选步的非成功终止(即所有该可选步的顶层选择对象都被检查了，但没有发现匹配分支)引起对下一个选择对象求值，或调用默认机制(如果显式调用是该可选步的最后的选择对象)。一个成功终止会引起测试成分的终止，即可选步以一个 **stop** 语句结束；或引起该 alt 语句的一个新快照和重新求值，即带有可选步以 **repeat** 语句结束，或是该 **alt** 语句的立即继续，即该测试步的被选定的顶层的选择对象不以显式的 **repeat** 语句结束。

在 **TTCN-3** 行为描述中，可选步可以作为独立语句被调用。在这种情况下，对可选步的调用可以解释为用于仅带有一个描述显式调用该可选步的选择对象的 **alt** 语句的简写(**shorthand**)。

例 5：

```
//语句
AnotherAltStep();              //假设 AnotherAltStep 是一个正确定义的 altstep
//是下面的 alt 语句一个简写:
alt {
    [] AnotherAltStep();
}
```

6.4.3　用于不同成分类型的函数和可选步

如果类型“A”和“B”兼容的话，可以在成分类型“A”的一个实例上启动在自己的 **runs on** 子句中引用了成分类型“B”的函数或可选步。

6.5　默认处理

TTCN-3 允许激活一个可选步为默认(**Default**)。对每个测试成分，以列表的形式存储默认(即激活的可选步)，并根据它们的激活顺序进行列表。TTCN-3 使用 **activate** 和 deactivate 操作对默认列表进行操作。**activate** 语句添加一个新的默认到默认列表的末端，而 **deactivate** 语句从默认列表中移去一个默认。默认列表的一个默认可以通过作为相应 **activate** 操作结果产生的默认引用来识别。

用于默认处理的 TTCN-3 语句一览表如表 6-5 所示。

表 6-5　用于默认处理的 TTCN-3 语句一览表

用于默认处理的语句	
语句	相关的关键字或符号
激活一个默认	**activate**
停用(**Deactivate**)一个默认	**deactivate**

6.5.1　默认机制

如果由于实际的快照而导致没有指明的选择对象能被执行，则在各 **alt** 语句结束处唤醒默认机制。被唤醒的默认机制调用默认类表中第一个可选步，并等待它的终止结果，可以是成功终止，也可以是非成功终止。非成功终止意味着该可选步中没有定义默认行为的顶层选择对象可以被选中，而成功终止意味着选中并执行了一个顶层的选择对象。

在非成功终止的情况下，默认机制调用默认列表中的下一个默认。如果默认列表中的最后一个默认都非成功终止了，那么默认机制将返回到 **alt** 语句中它的调用点。也就是说，返回到 **alt** 语句的结尾，并指示一个非成功的默认执行。当默认列表为空时，也将指示一个非成功的默认执行。

如果测试成分被阻塞，那么一个非成功默认执行可以导致一个新的快照，或一个动态错误。

在成功终止的情况下，默认情况下可以利用一个 **stop** 语句停止测试成分，或在调用默认机制的 **alt** 语句之后立即继续测试成分的主控制流，或给测试成分照一个新的快照并对 **alt** 语句重新求值。后者用 **repeat** 语句指示它。如果选中的默认的顶层选择对象不以 **repeat** 语句结束，那么在 **alt** 语句之后立即继续该测试成分的控制流。

注意：TTCN-3 不限制默认机制的实现。默认机制可以在每个 **alt** 语句的结尾处以被隐式调用的进程的形式实现，也可以以只对默认处理负责的单独线程的形式实现，唯一的要求是根据其激活顺序调用默认机制。

6.5.2　缺省引用

缺省引用(**Default references**)是激活默认对象的唯一的一种引用，这样的唯一的缺省引用是在一个可选步被激活为一个默认对象时产生的。也就是说，一个缺省引用是一个 **activate** 操作的结果。

缺省引用具有特殊的预定义类型 **default**。**default** 类型变量可以用于处理激活测试成分中的默认对象。对于未定义的缺省引用来说，特定的空值(**null**)是有效的，例如用于处理缺省引用的变量初始化。

在 **deactivate** 操作中，使用缺省引用来标识要被去激活的默认对象。

应该测试系统外部解析 **default** 类型的实际数据表示。这允许对抽象测试用例的说明独立于任意实际 TTCN3 运行环境之外。换句话说，**TTCN-3** 不限制带有有关默认处理和识别机制的测试系统的实现。

例：

```
//用于缺省的变量的声明，并初始化该变量为空
var default     MyDefaultVar := null;
:
//为保存一个激活的默认 MyDefaultVar 的用法
MyDefaultVar := activate(MyDefAltStep()); //MyDefAltStep 被激活为一个默认对象
:
//为去激活默认 MyDefAltStep 的 MyDefaultVar 的用法
deactivate(MyDefaultVar);
:
```

6.5.3 激活操作

activate 操作用于把可选步激活为默认对象。**activate** 操作将把被引用的可选步添加到默认列表中，并返回一个缺省引用。缺省引用是用于默认的一个唯一标识符，可以用在 **deactivate** 操作中去激活默认对象。

activate 操作的影响局限于调用它的测试成分本地，这就意味着一个测试成分不能激活另外一个测试成分中的默认对象。

例 1：

```
:
//用于默认处理的变量的声明
var default     MyDefaultVar := null;
:
//一个默认引用变量的声明和作为默认的一个可选步的激活
var default     MyDefVarTwo := activate(MySecondAltStep());
:
//默认的一个可选步的 MyAltStep 的激活
MyDefaultVar := activate(MyAltStep());  //MyAltStep 被激活为一个默认对象
:
```

应该在相应的 **activate** 语句中提供要被激活为默认对象可选步的实际参数，即在默认被激活时，实际参数与该默认对象绑定(不是在它被默认机制调用时)。

例 2：

```
:
var default     MyDefaultVar := null;
:
MyDefaultVar := activate(MyAltStep2(5, MyVar);
//MyAltStep2 被激活为默认对象，带有实际参数 5 和 MyVar 的值
//使用默认机制的一个 MyAltStep2 调用之前，MyVar 的变化将不改变该调用的实际参数
:
```

6.5.4 去激活操作

去激活(**deactivate**)操作用来去激活默认对象，即以前被激活的可选步。**deactivate** 操作将从默认列表中移去被引用的默认对象。

deactivate 操作的影响局限于调用它的测试成分，这就意味着一个测试成分不能去激活另外一个测试成分中的默认对象。

不带参数的 **deactivate** 操作去激活一个测试成分的所有默认对象。

调用一个带有特定值 **null** 的 **deactivate** 操作没有任何作用。调用一个带有未定义的缺省引用的 **deactivate** 操作，如已经被去激活的一个缺省的旧引用或没有被初始化的缺省引用变量，将导致一个运行错误。

例：

```
:
var default     MyDefaultVar    :=  null;
var default     MyDefVarTwo     :=  activate(MySecondAltStep());
var default     MyDefVarThree   :=  activate(MyThirdAltStep());
:
MyDefaultVar    :=  activate(MyAltStep());
:
deactivate(MyDefaultVar);       //去激活 MyAltStep:
deactivate; //去激活所有其他默认对象，即这个情况中的 MySecondAltStep 和 MyThirdAltStep
```

6.6　通信操作

TTCN-3 支持基于消息的(message-based)和基于过程的(procedure-based)通信。而且，TTCN-3 允许检查输入队列的队头元素和利用控制操作去控制对端口的访问。

TTCN-3 通信操作一览表如表 6-6 所示。

表 6-6　TTCN-3 通信操作一览表

通信操作			
操作	关键字	可用于基于消息的端口	可用于基于过程的端口
基于消息的通信(**Message-based communication**)			
发送消息	**send**	是	
接收消息	**receive**	是	
消息触发(Trigger on message)	**trigger**	是	
基于过程的通信(**Procedure-based communication**)			
调用过程调用	**call**		是
接收来自远程实体过程调用	**getcall**		是
回答来自远程实体过程调用	**reply**		是
(对一个已接收的调用)提出例外	**raise**		是
处理来自以前的调用的响应	**getreply**		是
捕获例外(从被调用实体)	**catch**		是
检查输入端口队列顶端元素(**Examine top element of incoming port queues**)			
检查接收到的消息/调用/例外/应答 msg/call/exception/reply received	**check**	是	是
控制操作(**Controlling operations**)			
清除端口(Clear port)	**clear**	是	是
清理并访问端口(Clear and give access toport)	**start**	是	是
停止对端口的访问(接收和发送)	**stop**	是	是

6.6.1　通信操作的通用格式

send 和 **call** 这样的操作用于测试成分之间以及被测系统 SUT 和测试成分之间交换信息。为了解释这些操作的通用格式，可以把它们分成以下两组。

(1)一个测试成分发送消息(**send** 操作)、调用一个过程(**call** 操作)、回答一个已接收的调用(**reply** 操作)或提出一个例外(**raise** 操作)。这些动作合起来称为发送操作(**sending operations**)。

(2)一个测试成分消息(**receive**操作)、等待一条消息(**trigger**操作)、接收一个过程调用(**getcall**操作)、收到一个以前被调用过程的应答(**getreply** 操作)或捕获一个例外(**catch** 操作)。这些动作合起来称为接收操作(**receiving operations**)。

其中发送操作的通用格式是由一个发送部分组成,在一个阻塞的基于过程的调用操作情况下,由一个响应和例外处理部分组成。

发送部分:

(1)指定要发生指定操作的端口。

(2)定义要传输的信息的值。

(3)给定一个在一对多连接的情况下唯一标识通信伙伴(**communication partner**)的地址表达式(可选的)。

端口名、操作名和值应出现在所有的发送操作中。通信伙伴的标识符(用关键字 to 表示)是可选的,且仅在需要显式地标识接收实体的一对多的情况下才需要被指定。

发送操作的通用格式(一)如表 6-7 所示。

表 6-7 发送操作的通用格式(一)

发送部分			(可选的)响应和例外处理部分
端口和操作	值部分	(可选的)地址表达	
MyP1.send	(MyVariable + YourVariable – 2)	to MyPartner;	

仅在基于过程的通信的情况下,才需要响应和例外处理。调用操作(**call**)的响应和例外处理是可选的,它们只有在如下情况下才是必需的:被调用过程返回一个值;或被调用过程有输出(**out**)或输入/出(**inout**)参数值,且在调用成分中需要这些参数的值;或被调用过程可能引起需要调用成分处理的例外。

调用操作的响应和例外处理部分使用 **getreply** 和 **catch** 操作提供必需的功能性(**functionality**)。

发送操作的通用格式(二)如表 6-8 所示。

表 6-8 发送操作的通用格式(二)

发送部分(Send part)			(可选的)响应和例外处理部分
端口和操作	值部分	(可选的)地址表达	

MyP1.call	(MyProc:{MyVar1})		{
			[] MyP1.getreply(MyProc:{MyVar2}) {}
			[] MyP1.catch(MyProc, ExceptionOne) {}
			}

接收操作的通用格式由一个接收部分和一个(可选)赋值部分组成。

接收部分:

(1)指定操作发生的端口。

(2)定义一个匹配部分,指定将匹配该语句的可接收的输入。

(3)给定一个唯一标识通信伙伴的(可选)地址表达式(在一对多连接的情况下)。

端口名、操作名和值应出现在所有的接收操作中。通信伙伴的标识符(用关键字 **from** 表示)是可选的,且仅在需要显式标识接收实体的一对多的情况下才需要被指定。

一个接收操作的(可选的)赋值部分是可选的。对于基于消息的端口来说,当需要它去存储接收到的消息时,赋值部分是必需的。在基于过程的端口的情况下,它用于存储一个已接收的调用的输入(**in**)和输入/出(**inout**)参数或存储例外。对于赋值部分,需要强类型机制,例如用于存储消息的变量类型应该与输入消息的类型相同。

此外，赋值部分也可以用于把一个消息、例外、应答(**reply**)或调用(**call**)的发送方地址赋给一个变量。这对一对多连接是很有用的，例如，可以接收来自不同成分的相同的消息或调用，但是这个消息、例外、应答(**reply**)或调用(**call**)必须发送回原始的发送成分。

接收操作的通用格式(一)如表 6-9 所示。

表 6-9　接收操作的通用格式(一)

接收部分				(可选)赋值部分		
端口和操作	匹配部分	(可选)地址表达式		(可选)值赋值	(可选)参数值赋值	(可选)发送者赋值
MyP1.getreply	(AProc:{?} value 5)		->		param (V1)	sender APeer

接收操作的通用格式(二)如表 6-10 所示。

表 6-10　接收操作的通用格式(二)

接收部分				(可选)赋值部分		
端口和操作	匹配部分	(可选)地址表达式		(可选)值赋值	(可选)参数值赋值	(可选)发送者赋值
MyP2.receive	(MyTemplate(5,7))	from APeer	->	value MyVar		

6.6.2　基于消息的通信

基于消息的通信是基于异步消息交换的通信。它在 **send** 操作上是非阻塞的，如图 6-3 所示，发送方在 **send** 操作之后立即继续它的处理过程。接收方在 **receive** 操作上被阻塞，直到它去处理接收消息。

除了 **receive** 操作外，TTCN-3 还提供了一个触发(**trigger**)操作，来根据一定的匹配标准过滤来自给定输入端口上接收信息流的消息。从该端口移去队列头部不满足匹配标准的消息，而不采取进一步的行动。

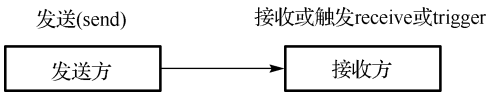


图 6-3　异步发送和接收的图示

图 6-3 中的发送(**send**)操作用于把一个值放到一个输出端口队列，我们可以通过引用一个模板、变量或一个常数来指定这个值，或可用一个表达式(当然是能给出明确值的)内嵌在一个语句行内来定义这个值。如果被发送的值的类型模糊，那么应在嵌入地定义值时使用可选的类型字段。

send 操作应该只能用在基于消息的(或混合型的)端口，且要发送的值的类型应该在该端口类型定义的输出类型列表中。

例 1：

```
MyPort.send(MyTemplate(5,MyVar));    //通过 MyPort 发送带有实参 5 和 MyVar 的模板
MyPort.send(5);                      //发送整型值 5
```

在一对多连接的情况下，应该唯一地指定通信伙伴，用关键字 to 表示。

例 2：

```
MyPort.send(charstring:"My string") to MyPartner;
//给存储在变量 MyPartner 中的成分引用发送串"My string"
MyPCO.send(MyVariable + YourVariable - 2) to MyPartner;
//给 MyPartner 发送一个算术表达式的结果
```

接收(**receive**)操作用于从一个输入消息端口队列中接收一个值。我们可以通过引用一个模板、变量或一个常数来指定这个值，或可用一个表达式(当然是能给出明确值的)内嵌在一个语句行内

来定义这个值。在内嵌地定义值的时候，应使用可选的类型字段来避免任何要接收值的类型的模糊。**receive** 操作应该只能用在基于消息的(或混合型的)端口，且要接收值的类型应该被包含在该端口类型定义的输入类型列表中。

当且仅当输入端口队列的队头消息满足所有 **Receive** 操作相关的匹配机制时，**Receive** 操作从相关的输入端口队列中移去该消息。不会发生把输入的值绑定到表达式或模板术语上。

如果匹配不成功，输入端口队列中的队头消息就不能被移去，也就是说，如果 **receive** 操作是 **alt** 语句中的一个选择对象且它不成功的话，测试用例的执行从 **alt** 语句中的下一个选择对象开始继续。

匹配标准与要接收消息的类型和值有关。要接收消息的类型和值可以来自于一个模板或一个表达式的结果值(当然是能给出明确值的)，应使用 **receive** 操作匹配标准中的可选类型字段来避免要接收值的类型的模糊。

注意：为了防止解码器重接收到的值，不是用编码属性描述来编码的消息中产生的一个抽象值，所以编码属性也以隐式的方式进行匹配。

在一对多连接的情况下，可以限制 **receive** 操作到一个特定的通信伙伴，使用关键字 **from** 来表示这种限制。

例 3：

```
MyPort.receive(MyTemplate(5, MyVar));    //在 MyPort 上匹配一个满足由模板
                                          //MyTemplate 定义的条件的一条消息
MyPort.receive(A<B);                      //匹配一个依赖于 A<B 结果的布尔值
MyPort.receive(integer:MyVar);            //在MyPort 上匹配一个整型值和MyVar 的值
MyPort.receive(MyVar);                    //前一个例子的一个替换
MyPort.receive(charstring:"Hello")from MyPeer;//匹配来自MyPeer 的字符串"Hello"
```

如果匹配成功，那么从端口队列中移去的值可以存储在一个变量中，且可以重新获得发送消息的成分的地址并存储到一个变量中，用符号'->'和关键字 **value** 来表示。

也可能重新获得和存储消息发送者的成分引用或地址，用关键字 **sender** 表示。

例 4：

```
MyPort.receive(MyType:?) -> value MyVar;    //把接收到的消息的值赋给 MyVar
MyPort.receive(A<B) -> sender MyPeer;       //把发送者的地址赋给 MyPeer
MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
//把接收到的消息的值赋给 MyVar，并把发送者的地址赋给 MyPeer
```

receive 操作接收任意消息，即对于不带有用于要接收消息的类型和值匹配标准的参数列表的 **receive** 操作，如果满足其他所有匹配标准，则移去输入端口队列的队头消息(如果有的话)。

注意：这与 TTCN-2 中的 OTHERWISE 语句等价。

通过 ReceiveAnyMessage 接收到的消息不会赋值给一个变量。

例 5：

```
MyPort.receive;
//从 MyPort 移去队列头部值 MyPort
receive from MyPeer;
//如果发送者是 MyPeer 的话，从 MyPort 移去队列的队头值
MyPort.receive -> sender MySenderVar;
//从 MyPort 移去队列的队头值，并把发送方地址赋给 MySenderVar
receive 操作使用关键字 any 在任意端口上接收消息
```

例 6：**any port**.**receive**(MyMessage);

触发(**trigger**)操作从相关的输入端口队列中移去头部消息。如果头部消息满足匹配标准的话，**trigger** 操作与 **receive** 操作的一样。如果头部消息不满足匹配标准的话，**trigger** 操作会从输入端口队列中移去队头消息，但没有进一步的动作。trigger 操作应该只能用在基于消息的(或混合型的)端口，且要接收的值的类型应该被包含在该端口类型定义的输入类型列表中。

在一个行为描述中，能够把 **trigger** 操作作为一个独立的语句。在后面的这种情况下，**trigger** 操作可以被看成只有一个选择对象的 **alt** 语句的简写。也就是说，它有阻塞语义，并因此提供等待下一个消息匹配指定的模板或那个队列上的值的能力。

例 7：

```
MyPort.trigger(MyType:?);
//在端口 MyPort 上，对观察到的第一个带有一个任意值的 MyType 类型消息的接收将触发该操作
```

trigger 操作需要端口名、用于类型和值的匹配标准、一个可选的 **from** 限制(即通信伙伴选择)以及一个可选的赋值(匹配消息并将发送方测试成分赋给变量)。

例 8：

```
MyPort.trigger(MyType:?) from MyPartner;
//在端口 MyPort 上，在接收来自 MyPartner 的第一个消息时触发
MyPort.trigger(MyType:?) from MyPartner -> value MyRecMessage;
//这个例子与上一个例子几乎是相同的。另外，触发的消息(即满足所有匹配标准的消息)存储在变
  量 MyRecMessage 中
MyPort.trigger(MyType:?) -> sender MyPartner;
//这个例子与第一个例子几乎是相同的。另外，发送方测试成分的引用将被重新获得，并存储在变量
  MyPartner 中
 MyPort.trigger(integer:?) -> value MyVar sender MyPartner;
//在接收一个后来存储在变量 MyVar 中的任意整型值时触发。发送方成分的引用将被存储在变量
  MyPartner 中
```

不带参数(**argument**)列表的 trigger 操作会在任意消息的接收上触发。因此，它的含义与接收任意消息的含义相同。被 TriggerOnAnyMessag 接收到的消息不会赋值给一个变量。

例 9：

```
MyPort.trigger;
MyPort.trigger from MyPartner;
MyPort.trigger -> sender MySenderVar;
```

trigger 操作在任意端口的消息上触发使用关键字 any。

例 10：any port.trigger;

6.6.3　基于过程的通信

基于过程的通信是调用远程实体的过程。TTCN-3 支持阻塞的(**blocking**)和非阻塞的(**non-blocking**)两种基于过程的通信。基于阻塞的过程通信是在调用方和被调用方停止系统的执行，即阻塞，直到过程调用结束。基于非阻塞的过程通信是仅被调用方被阻塞。

阻塞的基于过程的通信方案如图 6-4 所示。调用方(CALLER)通过 **call** 操作调用被调用方(CALLEE)中的一个远程过程，被调用方通过 **getcall** 操作接收这个调用，并通过使用一个 **reply** 操作来回答调用方或提出(**raise** 操作)一个例外来响应调用。调用方通过使用 **getreply** 操作或 **catch** 操作来处理应答或例外。在图 6-4 中，用虚线表示调用方和被调用方的阻塞。

非阻塞的基于过程的通信方案如图 6-5 所示。调用方通过 **call** 操作调用被调用方中的一个远程过程，并继续它自身的执行，即并不等待一个应答或例外。被调用方通过 **getcall** 操作接收这个调用，并执行被请求的过程。如果这个执行不成功，则被调用方可以产生一个例外去通知调用方。调用方可以通过使用 **alt** 语句中的 **catch** 操作来处理例外(**exception**)。在图 6-5 中，用虚线表示被调用方的阻塞(直到调用处理结束并可能提出一个例外)。

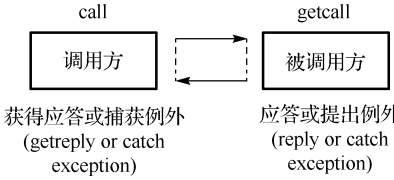


图 6-4 阻塞的基于过程的通信方案

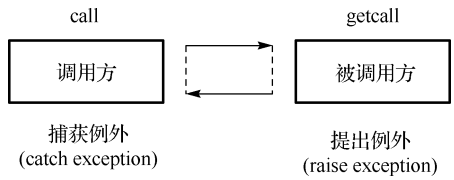


图 6-5 非阻塞的基于过程的通信方案

基于过程的通信主要有 6 种操作：调用(**call**)操作、**getcall** 操作、应答操作、获得应答操作、**Raise** 操作和捕获操作。

其中，调用(**call**)操作用于指定一个测试成分调用被测系统 SUT 或另一个测试成分中的过程。**call** 操作应该仅用在基于过程(或混合型)的端口上。调用操作发生的端口的类型定义应该在它的输出(**out**)或输入/出(**inout**)列表中包括过程名，也就是说必须允许在这个端口上调用这个过程。

在 **call** 操作的发送部分中传输的信息是一个可以在特征模板格式中定义的一个特征，也可以在语句行内嵌入式地定义一个特征。这个特征的所有输入(**in**)和输入/出(**inout**)应有一个特定值，也就是说不允许使用匹配机制，如 AnyValue。

不使用 **call** 操作的特征参数为输出(**out**)或输入/出(**inout**)参数重新获得变量名。过程返回值和输出(**out**)或输入/出(**inout**)参数对变量的实际赋值，应该在 **call** 操作的响应和例外处理部分中通过 **getreply** 和 **catch** 操作显式地进行。这允许 **call** 操作中特征模板的使用(与模板的使用方式相同)可以用于多种类型。

例 1：

```
//给定
signature MyProc (out integer MyPar1, inout boolean MyPar2);
:
//MyProc 的一个调用
MyPort.call(MyProc:{ -, MyVar2}) {
//用于 MyProc 的调用的嵌入在语句行内的特征模板
[] MyPort.getreply(MyProc:{?, ?}) { }
}

//以及 MyProc 的另一个调用
MyPort.call(MyProcTemplate) {
//为 MyProc 的调用使用特征模板
[] MyPort.getreply(MyProc:{?, ?}) { }
}
```

在一对多连接的情况下，应该唯一地指定通信伙伴，用关键字 to 表示。

例 2：

```
MyPort.call(MyProcTemplate) to MyPeer {
//在 MyPeer 调用 MyProc
[] MyPort.getreply(MyProc:{?, ?}) { }
}
```

在非阻塞的基于过程的通信的情况下，或者如果没有使用 **nowait** 选项的话，则通过使用作为 **alt** 语句的选择对象的 **getreply** 操作和 **catch** 操作来完成对 **call** 操作的响应和例外的处理。

在阻塞的基于过程的通信的情况下，在 **call** 操作的响应和例外处理部分中，通过使用 **getreply** 操作和 **catch** 操作来完成对 **call** 操作的响应和例外的处理。

一个 **call** 操作的响应和例外处理部分看起来与一个 **alt** 语句体相似，它定义了一个选择对象集合来描述对该调用的可能的响应和例外。对选择对象的选择仅仅基于用于被调用过程的 **getreply** 和 **catch** 操作，这意味着不允许使用 **else** 语句分支和调用 **alt** 可选步。

如果必要的话，可以通过置于选择对象的方括号“[ ]”之间的一个布尔表达式来激活/去激活这个选择对象。

一个 **call** 操作的响应和例外处理部分的执行就像一个没有任何活动的默认的 **alt** 语句。这就意味着一个相关的快照包含这个(可选的)布尔防卫定值所需的所有信息，可以包括被调用过程所运行之端口的顶端元素(如果有的话)，也可以包括一个由监视这个调用的(可选的)定时器产生的超时例外。

在响应和例外处理部分中，对防卫选择对象的布尔表达式的求值可能有副作用。为了避免意外的副作用，将使用与 **alt** 语句中一样的布尔防卫规则。

例 3：

```
//给定
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
exception (ExceptionTypeOne, ExceptionTypeTwo);
:
//MyProc3 的调用
MyPort.call(MyProc3:{ -, true }) to MyPartner {
    [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param (MyPar1Var,MyPar2Var)
    { }
    [] MyPort.catch(MyProc3, MyExceptionOne) {
    setverdict(fail);
    stop;
    }
    [] MyPort.catch(MyProc3, ExceptionTypeTwo : ?) {
    setverdict(inconc);
    }
    [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) { }
}
```

call 操作可以选择是否包含一个超时，这个超时作为一个明确的浮点类型值或常数来定义，并在启动 **call** 操作后定义这个时间的长短。超时例外应该由测试系统产生。如果在 **call** 操作中没有出现超时值部分，那么将不会产生超时例外。

例 4：

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
      [] MyPort.getreply(MyProc:{?, ?}) { }
      [] MyPort.catch(timeout) {
      //20ms 后超时例外
      setverdict(fail);
      stop;
```

```
        }
    }
```

在 **call** 操作中使用关键字 **nowait** 来代替超时例外值，允许不用等待响应、被调用过程提出(**raised**)的例外或超时例外而继续调用一个过程。

例 5：

```
MyPort.call(MyProc:{5, MyVar}, nowait);
//调用方测试成分不等待 MyProc 的终止而继续自身的执行
```

在使用了关键字 **nowait** 的地方，必须通过随后的 **alt** 语句中的一个 **getreply** 或 **catch** 操作来从端口队列中移去被调用过程的一个可能的响应或例外。

一个阻塞的过程可以没有返回值，没有输出和输入/出参数，也可以不提出例外。用于这样过程实例的调用操作应该有一个响应和例外处理部分以统一的方式去处理这个阻塞。

例 6：

```
//给定 signature MyBlockingProc (in integer MyPar1, in boolean MyPar2);
:
//MyBlockingProc 的一个调用
MyPort.call(MyBlockingProc:{ 7, false }) {
[] MyPort.getreply( MyBlockingProc:{ -, - } ) { }
}
```

非阻塞的过程没有输出和输入/出参数，没有返回值，且通过关键字 **noblock** 在相关的特征定义中指示非阻塞特性。对于一个非阻塞类的过程的 **call** 操作应该没有响应和例外处理部分，不提出超时例外，也不使用关键字 **noblock**。非阻塞类的过程可能提出的例外必须使用随后的 **alt** 语句中的 **catch** 操作来把它们从端口队列中移去。

getcall 操作用来说明一个测试成分接收来自被测系统或另一个测试成分的调用。getcall 操作应该仅用在基于过程的(或混合型的)端口上，且应该在该端口类型定义的允许引入过程列表中包含要被接收的过程调用的特征。

当且仅当满足 **getcall** 操作相关的匹配标准时，**getcall** 操作会从输入端口队列中移去队列头部的那个调用。这些匹配标准与要被处理的调用特征和通信伙伴有关，用于特征的匹配标准可以在语句行内嵌入式地说明或从一个特征模板派生。

在一对多连接的情况下，**getcall** 操作可能会限制到一个特定的通信伙伴，这个限制用关键字 **from** 表示。

例 7：

```
MyPort.getcall(MyProc(5, MyVar));
//在 MyPort 接收 MyProc 的一个调用
MyPort.getcall(MyProc:{5, MyVar}) from MyPeer;
//在 MyPort 接收一个来自 MyPeer 的 MyProc 的调用
```

getcall 操作的特征参数不应用来为输入(**in**)和输入/出(**inout**)参数传入变量名。输入(**in**)和输入/出(**inout**)参数值到变量的赋值应该在 **getcall** 操作的赋值部分中进行。这允许与模板使用方式相同的方式的 getcall 操作中特征模板的使用方式可以为各类型所使用。

getcall 操作的赋值部分(可选的)包含输入(**in**)和输入/出(**inout**)参数值到变量的赋值和调用成分地址的取回。关键字 **param** 用于取回调用的参数值。

当必须取回发送方地址(例如，在一对多配置中，用来寻找对调用伙伴(calling party)的应答 reply 或例外的地址)时，使用关键字 sender。

例 8：

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
//MyProc 的输入(in)或输入/出(inout)参数值赋给 MyPar1Var 和 MyPar2Var
MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
//在 MyPort 接收 MyProc 的一个带有输入(in)或输入/出(inout)参数 5 和 MyVar 的调用
//取回调用伙伴(calling party)的地址并存储在 MySenderVar 中
//下面 getcall 的例子说明了使用匹配属性和省略对测试说明不重要的可选部分的可能性
MyPort.getcall(MyProc:{5, MyVar}) -> param(MyVar1, MyVar2) sender MySenderVar;
MyPort.getcall(MyProc:{5, ?}) -> param(MyVar1, MyVar2);
MyPort.getcall(MyProc:{?, MyVar}) -> param( - , MyVar2);
//第一个输入/出(inout)参数不重要或不被用到
//下面的例子将解释赋值输入(in)或输入/出(inout)参数到变量中去的可能性。假设下面的特
//征用于被调用过程
signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);
MyPort.getcall(MyProc2:{?, ?, 3, - , ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
//把参数 A、B 和 E 赋值给变量 MyVarA、MyVarB 和 MyVarE。不需考虑输出(out)参数 D
MyPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B,
MyVarE:= E);
//用于输入(in)或输入/出(inout)参数赋值到变量的替代表示
//注意，赋值列表中的名字指用在 MyProc2 特征中的名字
MyPort.getcall(MyProc2:{1, 2, 3, -, *}) -> param (MyVarE:= E);
//在后面(further)的测试用例执行中，只有输入/出(inout)参数值需要被用到
```

不带用于特征匹配标准的参数列表的 getcall 操作，如果满足其他所有匹配标准，将移去输入端口队列头部的调用(如果有的话)。通过 AcceptAnyCall 接收的调用的参数不会赋值给变量。

例 9：

```
MyPort.getcall;                    //MyPort 移去队列头部的调用
MyPort.getcall from MyPartner;  //从端口 MyPort 移去一个来自 MyPartner 的调用
MyPort.getcall -> sender MySenderVar; //从MyPort 移去一个调用并取回调用实体的地址
```

用关键字 any 表示在任意端口上的 getcall 操作。

例 10：

```
any port.getcall(MyProc)
```

应答(**reply**)操作用于根据过程特征回答一个以前接收的调用。**reply** 操作应该仅用在基于过程的(或混合型的)端口上，且端口的类型定义应该包括 **reply** 操作所属的过程名。

注意：不能总是静态地检查一个已接收的调用和一个应答操作之间的关系，对于测试来说，允许指定一个 reply 操作但不带有相关联的 getcall 操作。

reply 操作的值部分由一个带有相关实参列表的特征引用和(可选的)返回值组成，这个特征可以定义在特征模板格式中，或在语句行内嵌入式地定义它。特征的所有输入(**in**)和输入/出(**inout**)参数应该有特定的值，也就是说不允许使用像 **AnyValue** 这样的匹配机制。

在一对多连接的情况下，应该显式地指定通信伙伴，且该通信伙伴应该是唯一的，用关键字 to 表示。

如果要返回一个值给调用方，要用关键字 value 来显式说明。

例 11：

```
MyPort.reply(MyProc2:{ - ,5});          //回答接到的 MyProc2 的一个调用
MyPort.reply(MyProc2:{ - ,5}) to MyPeer;//回答接到的来自 MyPeer 的 MyProc2 的一
                                          个调用
MyPort.reply(MyProc3:{5,MyVar} value 20);//回答接到的 MyProc2 的一个调用
```

获得应答(**getreply**)操作用于处理来自先前被调用过程的应答，仅能用在基于过程(或混合型的)端口上。

当且仅当满足相关的 **getreply** 操作的匹配标准时，**getreply** 操作会移去输入端口队列队头的应答。这些匹配标准与被处理过程特征和通信伙伴有关，用于特征的匹配标准可以在语句行内嵌入式地说明，或从一个特征模板上派生。

可以用关键字 **value** 说明对一个返回值的匹配。

在一个一对多连接的情况下，可能限制一个 **getreply** 操作到一个特定的通信伙伴，用关键字 **from** 表示这个限制。

例 12：

```
MyPort.getreply(MyProc:{5, ?} value 20);
//接收一个 MyProc 的应答，该应答带有两个输出(out)或输入/出(inout)参数和一个返回值 20
MyPort.getreply(MyProc2:{ - , 5}) from MyPeer;
//接收来自 MyPeer 的 MyProc 的一个应答
```

getreply 操作的特征参数不应用来为输出(**out**)和输入/出(**inout**)参数传入变量名。输出(**out**)和输入/出(**inout**)参数值到变量的赋值应该在 **getreply** 操作的赋值部分中进行。这允许与模板使用方式相同的方式的 **getreply** 操作中特征模板的使用方式可以为各类型所使用。

getreply 操作的赋值部分(可选的)包含输出(**out**)和输入/出(**inout**)参数值到变量的赋值和应答发送方地址的获得。关键字 param 用于取回应答的参数值。当需要取回发送方地址时，使用关键字 **sender**。

例 13：

```
MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param(MyPar1,MyPar2);
//返回值赋给变量 MyRetValue，两个输出(out)或输入/出(inout)参数赋值给变量 MyPar1 和 MyPar2
MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2)
sender MySender;
//在后面的测试执行中，不考虑第一个参数值。取回发送方成分的地址并存储在变量 MySender 中
//下面的例子描述了把输出(out)或输入/出(inout)参数赋值给变量的可能性。为已被调用的过
  程假设如下特征
signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);
MyPort.getreply(ATemplate) -> param( - , - , - , MyVarOut1, MyVarInout1);
MyPort.getreply(ATemplate) -> param(MyVarOut1:=D, MyVarOut2:=E);
MyPort.getreply(MyProc2:{ - , - , - , 3, ?}) -> param(MyVarInout1:=E);
```

而获得任意应答，即如果满足所有其他匹配标准，那么不带有用于特征匹配标准的 **getreply** 操作将会从输入端口队列的队头移去一个应答(如果有的话)。通过 **GetAnyReply** 接收到的参数或返回值不会分配给变量。

例 14：

```
MyPort.getreply;                          //移去 MyPort 队头的应答
MyPort.getreply from MyPeer;              //移去 MyPort 队头来自 MyPeer 的应答
MyPort.getreply -> sender MySenderVar;//移去 MyPort 队头的应答，取回发送方实体的地址
```

另外在任意端口上获得一个回答使用关键字 **any**。

例 15：

```
any port.getreply(Myproc)
```

raise 操作用于提出一个例外，仅能在基于过程的(或混合型的)端口上提出例外。例外是对导致例外事件的已接收的过程调用的一个反应。应在被调用过程特征中说明例外的类型。端口的类型定义应该在其已接收过程调用的列表中包含例外所属的过程名。

注意：不可能总是静态地检查一个已接收的调用和 raise 操作的关系，对于测试来说，允许说明一个 raise 操作与一个 getcall 操作无关。raise 操作的值部分由跟着例外值的特征引用组成。

例外被作为类型描述，因此例外值可能来自一个模板，或是一个表达式的结果(当然表达式可以是一个确定的值)。在必须避免发送值类型的含糊的情况下，对 raise 操作将使用值定义中的可选类型字段。

在一对多连接的情况下，应该唯一指定通信伙伴，用关键字 to 表示。

例 16：

```
MyPort.raise(MySignature, MyVariable + YourVariable - 2);
//为 MyPort 提出带有算术表达式结果值的一个例外
MyPort.raise(MyProc, integer:5));
//为 MyProc 提出带有整型值 5 的一个例外
MyPort.raise(MySignature, "My string") to MyPartner;
//为 MyPort 提出带有值"My string"的一个例外，并把它发送给 MyPeer
```

捕获(**catch**)操作用于捕获一个例外，这个例外由对等实体提出，作为对一个过程调用的响应。**Catch** 操作仅能被用在基于过程的(或混合型的)端口上，应该在提出例外的过程特征中说明被捕获例外的类型。例外被说明成类型，因此可以像消息一样对待它，例如，可以用模板来区分相同例外类型的不同值。

当且仅当相关输入端口队列头部的例外满足所有 **catch** 操作相关的匹配标准时，**catch** 操作将移去该队列头部的这个例外。不会发生输入值到表达式术语(**terms**)或模板例外的绑定。

在一对多连接的情况下，**catch** 操作可能被限制到一个特定的通信伙伴，用关键字 **from** 表示这个限制。

例 17：

```
MyPort.catch(MyProc, integer: MyVar);
//在端口捕获 MyProc 提出的一个值为 MyVar 的整型例外
MyPort.catch(MyProc, MyVar);                    //上个例子的一个替换
MyPort.catch(MyProc, A<B);                      //捕获一个布尔型例外
MyPort.catch(MyProc, MyType:{5, MyVar});        //一个例外值的嵌入式模板
MyPort.catch(MyProc, charstring:"Hello")from MyPeer;//从 MyPeer 捕获"Hello"例外
```

catch 操作的赋值部分(可选的)包含例外值的赋值和调用成分地址的取回。用关键字 **value** 取回一个例外值，当需要取回发送方地址的时候，使用关键字 **sender**。

例 18：

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
//捕获来自 MyPartner 的一个例外，并把它的值赋给 MyVar
MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
//捕获来自 MyPartner 的一个例外，把它的值赋给 MyVar，并取回发送方地址
```

catch 操作可以是 **call** 操作的响应和例外处理中的一部分，或用于决定 **alt** 语句的一个选择对象。如果 **catch** 操作用在 **call** 操作的接收部分，关于端口名的信息和用来指示提出例外的过程的特征引用是多余的，因为这些信息跟在 **call** 操作后面。然而，由于可读性原因(如在复杂的 **call** 语句的情况下)，应该重复这些信息。

catch 操作捕获一个特别的超时例外(**timeout**)。对于被调用过程不在预定的时间内既不响应也不提出例外的情况，超时例外是一个紧急出口。

例 19：

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
    [] MyPort.getreply(MyProc:{?, ?}) { }
    [] MyPort.catch(timeout) {
    //20ms 之后，超时例外
    setverdict(fail);
    stop;
    }
}
```

捕获超时(**timeout**)例外应该限制在一个调用的例外处理部分。对于处理超时(**timeout**)例外的 catch 操作来说，不允许更多的匹配标准(包括一个 **from** 部分)和赋值部分。

没有参数列表的 **catch** 允许捕获任意有效的例外。最通常的情况是不使用关键字 **from** 和一个赋值部分，这个语句也捕获 timeout 例外。

例 20：

```
MyPort.catch;
MyPort.catch from MyPartner;
MyPort.catch -> sender MySenderVar;
```

catch 操作使用关键字 **any** 在任意端口上捕获一个例外。

例 21：

```
any port.catch;
```

6.6.4 检查操作

check 操作是一个普通的操作，它允许读取基于消息的和基于过程的输入端口队列的队头元素，但不从该队列移去队头元素。**check** 操作必须在基于消息的端口上处理某个类型值，在基于过程的端口上区别要接收的调用、要捕获的例外和来自先前调用的应答。

check 操作使用接收操作(**receive**、**getcall**、**getreply** 和 **catch**)和它们的匹配和赋值部分一起去定义必须检查的条件，并在必要时提取它的值。

要检查的是输入端口队列的队头元素(不可能检查队列的内部)。如果队列是空的，则 check 操作失败。

如果队列不空，复制队列的队头元素，并在这个复制的队头元素上执行 **check** 操作中指定的接收操作。如果接收操作失败，则 **check** 操作失败，即不满足匹配标准。在这种情况下，丢弃队头元素并按正常方式继续测试执行，为 **check** 操作的下一条语句或选择对象定值。如果接收操作成功，则 **check** 操作成功。

按照错误的方式使用 **check** 操作，如在一个基于消息的端口检查例外，会导致一个测试用例错误。

注意：在大多数情况下，可以静态地检查 **check** 操作的正确用法，即在编译前检查。

例 1：

```
MyPort1.check(receive(5));        //检查一个值为 5 的整型消息
MyPort2.check(getcall(MyProc:{5, MyVar}) from MyPartner);
//检查端口 MyPort2 上来自 MyPartner 的对 MyProc 的一个调用
MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
//在端口 MyPort 上检查返回值为 20 且两个输出(out)或输入/出(inout)参数的值为 5 和 MyVar
  的来自过程 MyProc 的一个应答
MyPort2.check(catch(MyProc, MyTemplate(5, MyVar)));
MyPort2.check(getreply(MyProc1:{?, MyVar} value *) -> value MyReturnValue
param(MyPar1));
MyPort.check(getcall(MyProc:{5,MyVar}) fromMyPartner->param(MyPar1Var,
MyPar2Var));
MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
```

没有参数列表的 check 操作允许检查在输入端口队列中是否有元素等待处理。CheckAny 操作允许通过使用一个 **from** 子句区别不同的发送方(在一对多连接的情况下)，以及通过使用带有一个 **sender** 子句的简写的赋值部分来重新获得发送方。

例 2：

```
MyPort.check;
MyPort.check(from MyPartner);
MyPort.check(-> sender MySenderVar);
```

使用关键字 **any** 在任意端口上进行检查。

例：

```
any port.check;
```

6.6.5　控制通信端口

TTCN-3 中用于控制基于消息的、基于过程的和混合型的端口的控制如下。

- **clear**：移去一个输入端口队列的内容。
- **start**：启动对一个端口的监听和对端口的访问。
- **stop**：停止对端口的监听和取消在该端口上对发送操作的允许。

clear 操作移去指定端口的输入(incoming)队列的内容。如果输入队列已经是空的，那么该操作什么也不做。

例 1：

```
MyPort.clear;        //清除端口 MyPort
```

如果一个端口定义为允许接收操作，如 **receive**、**getcall** 等，Start 命令启动对通过该端口上的通信流的监听。如果定义该端口允许发送操作，那么 **send**、**call**、**raise** 等操作可以在这个端口上执行。

例 2：

```
MyPort. start;       //启动 MyPort
```

在默认情况下，当一个成分在被创建时，其所有端口都被隐式地启动了。启动端口操作将通过移去在输入队列中等待的所有消息来重新启动没有被停止的端口。

如果定义一个端口允许接收操作，如 **receive** 和 **getcall**，**stop** 操作导致停止对指定端口的监听。如果定义端口允许发送操作，那么停止端口不允许执行 **send**、**call**、**raise** 等操作。

例 3：

```
MyPort. stop;              //停止 MyPort
```

注意：停止端口上的监听意味着在 stop 操作之前定义的所有接收操作应该在端口的工作被挂起之前就全部执行了。

例 4：

```
MyPort. receive (MyTemplate1) -> RecPDU;
//解码接收到的值，并和 MyTemplate1 匹配，将匹配值存入变量 RecPDU
MyPort. stop;
//没有定义在 stop 操作后的接收操作被执行(除非通过后面的 start 操作重新启动该端口
MyPort. receive (MyTemplate2);
//这个操作没有被执行
```

6.6.6 any 和 all 与端口一起使用

如表 6-11 所示，关键字 **any** 和 **all** 可以和配置和通信操作一起使用。

表 6-11 与端口一起的 Any 和 All

操作	被允许		例子
	any	all	
receive, trigger, getcall, getreply, catch, check	是		any port.receive
connect / map			
start, stop, clear		是	all port.start

6.7 定时器操作

TTCN-3 支持许多定时器操作，这些操作可以用在测试用例、函数、可选步和模板控制中。假设每个 TTCN-3 中声明定时器的范围单元维护它自己的运行的定时器列表(running-timers list)和超时列表(timeout-list)，即所有运行的定时器的列表和所有超时的定时器列表。超时列表是测试用例执行时照的快照的一部分，如果在定时器范围单元中一个定时器被启动、停止、超时或执行 timeout 操作，那么更新超时列表。

注意 1：运行的定时器列表和超时列表只是一个概念上的(conceptual)列表，并不限制定时器的实现。也可能使用其他的数据结构，如集合(set)，此时对超时事件的访问不受超时事件发生的顺序限制。

注意 2：对每个测试成分，假设在相应的成分类型定义中声明了处理定时器启动/停止和定时器超时事件的一个特殊的运行的定时器列表和超时列表。

当一个定时器到期时(从概念上讲的选择对象事件集合的快照处理之前)，超时事件被放在声明该定时器的范围单元的超时列表中。定时器立即变为不活动。在任意时间，对于任意特定定时器，仅有一个登录条目会出现在声明定时器的范围单元的超时列表中。

在测试成分被显式或隐式停止时，所有运行中的定时器被自动取消。

TTCN-3 定时器操作一览表如表 6-12 所示。

表 6-12 TTCN-3 定时器操作一览表

定时器操作	
语　　句	相关的关键字或符号
启动定时器	Start
停止定时器	Stop
读取定时器经过的时间	Read
检查定时器是否运行	Running
超时时间	Timeout

6.7.1　启动定时器操作

启动定时器操作(**start**)用来指出一个定时器的开始运行，定时器的值应该是非负浮点数(即大于等于 0.0)。当一个定时器被启动时，它的名字就被添加到运行的定时器列表中(对给定的范围单元)。

例 1：

```
MyTimer1.start;
//启动 MyTimer1，带有一个默认的持续时间
MyTimer2.start(20E-3);
//启动 MyTimer2，持续时间为 20ms 可以在一个循环中启动定时器数组的元素
```

例 2：

```
timer t_Mytimer [5];
var float v timerValues [5];
for (var integer i := 0; i==4; i:=1)
{
    v timerValues [i] := 1.0
}
for (var integer i := 0; i==4; i:=1)
{
    t_Mytimer [i].start ( v_timerValues [i])
}
```

如果没有给定默认的持续时间或想要重写定时器声明中描述的默认值，应该使用可选的定时器值参数。当重写定时器持续时间时，新的值仅用于当前这个定时器实例，以后这个定时器的任意 **start** 操作，如果没有指定持续时间的话，将使用默认持续时间。

启动一个定时器值为 0.0 的定时器意味着这个定时器立刻就超时了。启动一个带有负数定时器值的定时器，如定时器的值是一个表达式的结果，或者不带有指定的定时器值，将会导致一个运行错误。

定时器时钟运行从浮点值零(0.0)到持续时间参数说明的最大值。

可以对一个正在运行的定时器应用 **start** 操作，在这种情况下，定时器被停止并重新启动。在超时列表中用于这个定时器的任意条目都将被从这个超时列表中移去。

6.7.2　停止定时器操作

停止(**stop**)操作用来停止一个正在运行的定时器，并从正在运行的定时器列表中把它移去。一个被停止的定时器变成不活动的，它的经过时间被设为浮点值零(0.0)。

尽管没有任何影响，但停止一个不活动的定时器是一个有效的操作。超时列表中用于这个定时器任意条目将被从该超时列表中移去。

关键字 **all** 可以用来停止调用 **stop** 操作的范围单元中所有可见的定时器。

例：

```
MyTimer1.stop;              //停止 MyTimer1
all timer. stop;            //停止所有正在运行的定时器
```

6.7.3　读定时器操作

读操作(**read**)用来取回指定的定时器启动后经过的时间，并把这个时间存储到指定的变量中去，这个变量的类型应该是浮点型(**float**)的。

例：

```
var float Myvar;
MyVar := MyTimer1. read;    //把 MyTimer1 启动后经过的时间赋给 MyVar
```

对一个不活动的定时器应用 **read** 操作，即不在正在运行的定时器列表中的定时器，将返回值零。

6.7.4 运行定时器操作

运行定时器(**running**)操作用来检查一个定时器是否在给定范围单元的正在运行的定时器列表中(即该定时器已经被启动，且没有超时或被停止)。如果该定时器在列表中，返回真值(**true**)，否则返回假值(**false**)。

例：

```
if (MyTimer1. running){ … }
```

6.7.5 超时操作

超时操作(**timeout**)允许检查一个测试成分的范围单元内或调用超时操作的模块控制中的一个定时器或所有定时器的时间，是否在所规定的时间范围内。

处理 **timeout** 操作时，如果指定定时器的名字，则根据 TTCN-3 的范围规则搜索这个成分或模块控制的超时列表。如果有与该定时器名匹配的超时事件，从超时列表中移去该事件，**timeout** 操作成功。**timeout** 操作不应用在一个布尔表达式中，但它能用来决定 **alt** 语句中的一个选择对象，或作为一个行为描述中的独立的语句。在后面的这种情况下，可以把 **timeout** 操作认为只有一个选择对象的 **alt** 语句的简写，也就是说，它具有阻塞语义，并因此提供被等待定时器超时的能力。

注意：TTCN-3 的 timeout 操作与 TTCN-2 的 TIMEOUT 操作有相同的语义。

例 1：

```
MyTimer1.timeout;              //检查先前启动的定时器 MyTimer1 的超时
```

如果超时列表为空的话，使用关键字 **any** 的 **timeout** 操作成功(不是明确指定的定时器)。

例 2：

```
any timer. timeout;            //检查任意以前启动的定时器的超时
```

6.7.6 与定时器一起使用的 any 和 all 的总结

如表 6-13 所示，关键字 **any** 和 **all** 可以和定时器操作一起使用。

表 6-13 带有 Any 和 All 的定时器操作

操作	被允许		例子
	any	all	
start			
stop		是	all timer.stop
read			
running	是		if (any timer.running) {…}
timeout	是		any timer.timeout

思考题

1．TTCN 中可以采用哪几种通信操作？分别描述这几种操作的一般步骤。

2．定时器有哪几种操作？遇到超时定时器做出的响应是什么？

3．TTCN 中基本的程序语句有哪些？

第 7 章　TTCN-3 核心语言程序设计

本章介绍 TTCN-3 核心语言的基本概念、语法结构和测试系统结构。通过学习可以掌握 TTCN-3 核心语言测试系统的设计与实现方法，能够设计小型协议软件、应用软件、嵌入式软件的测试系统，为实际从事测试工作奠定理论基础。在全面介绍 TTCN-3 之前，首先介绍 TTCN-3 的基本语言元素，明确各种符号的含义。由于 TTCN-3 核心语言与数表形式语言的基本语言元素有许多相似之处，所以本章重点介绍核心语言的特色，对二者相似的内容不做重点讲解。

7.1　测试配置

TTCN-3 允许对并发测试配置(或简称配置)的(动态)描述，一个配置由一个带有良好定义的通信端口的互连测试成分集合和定义该测试系统边界的明确的测试系统接口组成。

在每个配置中应该有一个(且仅有一个)主测试成分(MTC)，非主测试成分被称为并行测试成分(PTCs)。MTC 应该在每个测试用例执行开始时由系统创建，而测试用例主体中定义的行为应该在该成分上执行。在一个测试用例执行期间，通过显式使用 **create** 操作来动态创建其他成分。

应在 MTC 终止时结束测试用例执行，平等对待所有其他的 PTCs，即在它们之间没有明确的层次关系，一个 PTC 既不能终止其他 PTCs，也不能终止 MTC。当 MTC 终止时，测试系统必须停止所有测试用例结束时未被终止的 PTCs。

测试成分之间以及测试成分和测试系统接口之间的通信由通信端口实现。

用关键字 **component** 和 **port** 指示的测试成分类型和端口类型应该在模块定义部分中定义，成分的实际配置以及它们之间的连接通过在其测试用例行为中执行 **create** 和 **connect** 操作来实现，利用 **map** 操作将成分端口连接到测试系统接口的端口上。

一个典型的 TTCN-3 测试配置的概念化视图如图 7-1 所示。

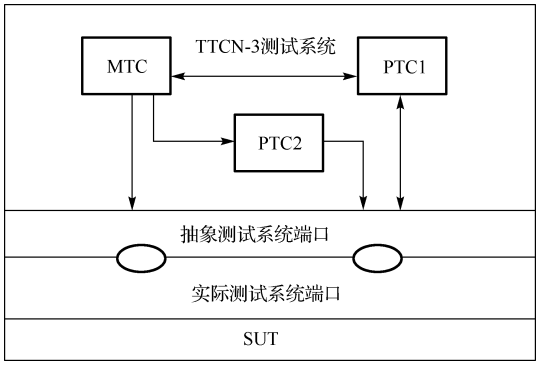


图 7-1　一个典型的 TTCN-3 测试配置的概念化视图

7.1.1　端口通信模型

测试成分通过它们的端口连接。也就是说，测试成分之间以及一个成分和测试系统接口之间的连接是面向端口的。每个端口被建模为一个无限的先进先出(FIFO)队列用于存储进来的消息，或者通过过程调用拥有该端口的成分直接处理。

注意：原则上，**TTCN-3** 端口是无限的，然而在实际的测试系统中，它们可能会溢出，这时应该以一个测试用例错误来对待它。

TTCN-3 通信端口模型如图 7-2 所示。

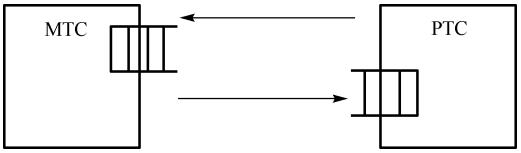


图 7-2　TTCN-3 通信端口模型

7.1.2　连接上的限制

TTCN-3 连接是端口到端口(port-to-port)和端口到测试系统接口(port-to-test system interface)的连接。对于一个成分可以维护的连接数目没有限制，也允许一对多的连接。允许的连接如图 7-3 所示。

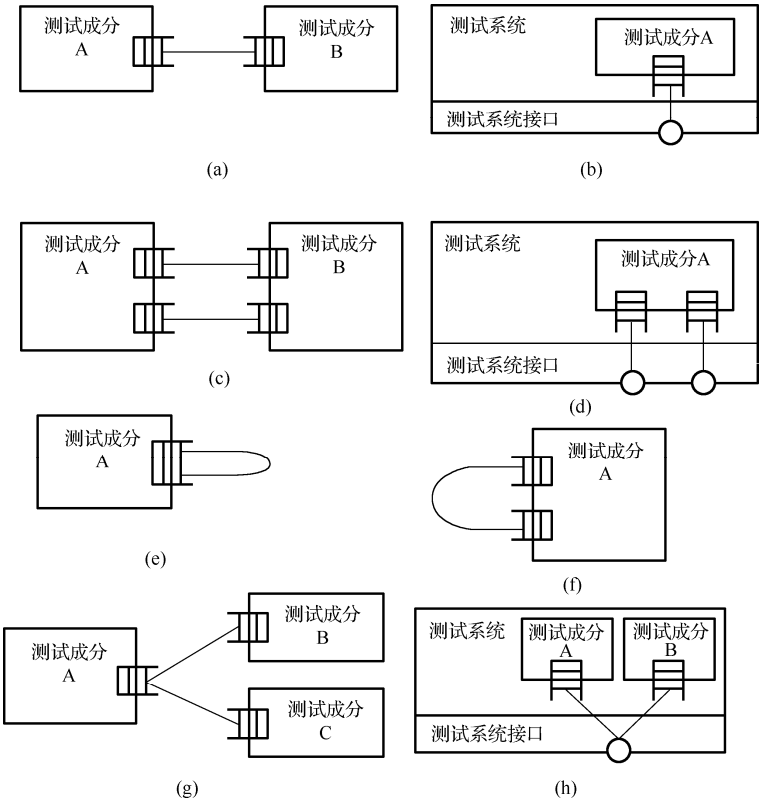


图 7-3　允许的连接

TTCN-3 不允许以下连接。

(1)一个测试成分 A 的一个端口不应该与该测试成分上的两个或两个以上的端口相连，见图 7-4(a)和图 7-4(e)。

(2)一个测试成分 A 的一个端口不应该与测试成分 B 上的两个或两个以上的端口相连，见图 7-4(c)。

(3)一个测试成分 A 的一个端口与测试系统接口可能仅有一个一对一的连接，这就意味着图 7-4(b)和图 7-4(d)所示的连接是不允许的。

(4)不允许测试系统接口内部的连接，见图 7-4(f)。

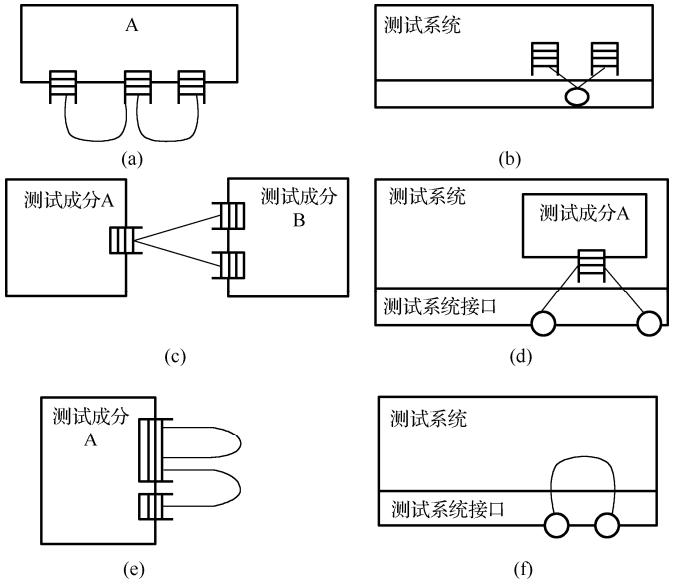


图 7-4　不允许的连接

因为 TTCN-3 允许动态配置和动态地址，所以不可能在编译时总是检查连接上的限制，从而应该在运行时进行检查，且在失败时引起一个测试用例错误。

7.1.3　抽象测试系统接口

TTCN-3 的对象被认为是被测实现(Implementation Under Test，IUT)。IUT 可以为测试提供直接接口，或者称为被测系统(System Under Test，SUT)时，它可以是该系统的一部分。在本文中，术语 SUT 是一个通用的概念，意味着 SUT 或 IUT。

在一个实际的测试环境中，测试用例需要与 SUT 进行通信。然而，对实际物理连接的说明超出了 TTCN-3 的范围，作为替代，一个良好定义的(但是是抽象的)测试系统接口应该与每个测试用例相关联。一个测试系统接口定义与一个成分定义相同，即它是一个所有可能通信端口的集合，通过这些端口测试用例被连接到 SUT。

测试系统接口静态地定义测试运行时与 SUT 连接的端口数目和类型，但测试系统接口和 TTCN-3 测试成分之间的连接实际上是动态的，可以在一个测试运行时使用 **map** 和 **unmap** 操作对其进行修改。

7.1.4　定义通信端口类型

端口便于测试成分之间以及测试成分与测试系统接口之间的通信。

TTCN-3 支持基于消息和基于过程的两种端口。每个端口应该定义为基于消息的或基于过程的(或者二者混合型)，通过关键字 **message** 来标识基于消息的端口，而关键字 **procedure** 则是用来标识基于过程的端口的。

端口是有方向的，它的方向通过关键字 **in**(输入方向)、**out**(输出方向)和 **inout**(输入/输出方向)来标识。每个端口类型定义应该由一个或多个列表来说明通信方向。

例 1：

```
//基于消息的端口，其上允许接收 MsgType1 和 MsgType2 类型，发送 MsgType3 类型，并可发送
  和接收任意整型值
```

```
type port MyMessagePortType  message
{
    in      MsgType1, MsgType2;
    out     MsgType3;
    inout   integer;
}
//基于过程的端口，它允许过程 Proc1、Proc2 和 Proc3 的远程调用
//注意 Proc1、Proc2 和 Proc3 定义为过程特征(signatures)
type port MyProcedurePortType procedure
{
    out Proc1, Proc2, Proc3
}
```

注意：消息通常用模板(templet)和表达式的实际值来表示，因此，当定义基于消息的端口时，消息就只是一个简单的类型名称列表。

使用关键字 **all** 来说明在该模块中，允许所有的用户定义类型、嵌入类型和简单类型数据在该端口上传递。

例 2：

```
//基于消息的端口，它允许该端口上任意方向上传输所有构造类型和用户自定义类型的值
type  port MyAllMesssagesPortType message
{
    inout all
}
```

把一个端口定义为基于消息的通信和基于过程的通信混合型的端口也是可能的，在这种情况下使用关键字 mixed 表示。这就意味着混合型端口的列表也将是混合的，它包括了过程信号和类型。此时，关键字 all 表示该模块中定义的所有类型和过程信号，在定义中也不用分别说明。

```
//混合型端口，定义了相同名字的一个基于消息的和一个基于过程的端口
//out 和 inout 列表也是混合型的：MsgType1、MsgType2、MsgType3 和 integer 涉及混合型
  端口中基于消息的部分
//Proc1、Proc2、Proc3、Proc4 和 Proc5 涉及端口中基于过程的部分
type  port MyMixedPortType  mixed
{
    in      MsgType1, MsgType2, Proc1, Proc2;
    out     MsgType3, Proc3, Proc4;
    inout   integer, Proc5;
}
//混合型端口，模块中定义的所有类型和所有过程特征都可以用在这个端口上与 SUT 或其他测试成分
  进行通信
type port MyAllMixedPortType mixed
{
    inout  all
}
```

TTCN-3 中的混合型端口实际是两类端口定义的缩写表示，也就是说，一个基于消息的端口和一个基于过程的端口具有相同名字。在运行时，可以通过通信操作来区分两类端口。

如果调用一个混合性端口的标识符，用于控制端口的操作，即 **start**、**stop** 和 **clear** 应该在两个队列上(按照任意的顺序)都执行操作。

7.1.5　定义通信类型

成分类型(component)定义了与一个成分相关联的端口。这些定义应该在模块的定义部分中进行定义，一个成分中的端口名字对该成分来说是本地的，也就是说另外的成分可以有相同名字的端口。但是，相同成分的所有端口名字应该是唯一的。一个成分的定义本身并不意味着该成分在这些端口上有任何连接。

注意：*在这个方面，TTCN-3 与 TTCN-2 有区别。测试配置是静态的，而对测试成分、PCOs 和 ASPs 的声明则隐含了在初始化测试用例执行时它们的自动连接。*

典型的测试成分如图 7-5 所示。

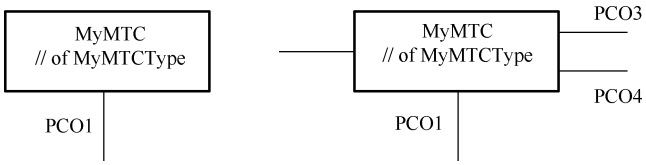


图 7-5　典型的测试成分

例 1：

```
type  component  MyMTCType
{
    port  MyMessagePortType  PCO1
}
type component  MyPTCType
{
    port  MyMessagePortType  PCO1, PCO4;
    port  MyProcedurePortType  PCO2;
    port  MyAllMesssagesPortType  PCO3
}
```

可以声明一个特殊成分的本地常量、变量和定时器。

例 2：

```
type component MyMTCType
{
    var integer MyLocalInteger;
    timer MyLocalTimer;
    port MyMessagePortType PCO1
}
```

这些声明对于该成分上运行的所有函数和可选步都是可见的，这应该使用关键字 runs on 来显式说明。

成分变量和定时器与该成分实例相关联，并遵循 4.1.3 节定义的范围规则。一个成分的每个新实例也将有它自己的变量和定时器集合，就像成分定义中所描述的那样(如果说明了，则还包括任意初始化的值)。

注意：*当成分被作为测试系统接口时，该成分不能使用成分中声明的任一常量、变量和定时器。*

在成分类型定义中可以定义端口数组。

例 3：

```
type component My3pcoCompType
{
    port MyMessageInterfaceType  PCO[3]
    //定义一个成分类型，它带有的数组由三个端口组成
}
```

7.1.6 SUT 内部的编址实体

一个 SUT 可以由数个必须分别编址的实体组成。地址数据类型是用来与端口操作结合起来给 SUT 实体编址的一种数据类型。当地址数据类型与 to、from 和 sender 一起使用的时候，它只能用在映射到测试系统接口的端口的接收和发送操作中。address 的实际数据表示由该测试套中一个明确的类型定义来解析，或者由测试系统来进行外部解析(也就是说，address 类型被作为 TTCN-3 说明中的一种开放式的类型)。这就允许独立于 SUT 任意特定的实际地址机制描述抽象测试用例。

如果地址类型定义在一个模块内部，那么应该仅在该 TTCN-3 模块内部生成明确的 SUT 地址。如果该类型不是定义在一个该模块的内部，那么明确的 SUT 地址应该仅可以在消息中作为字段参数被传入或接收，或者作为远程过程调用的参数被传入或接收。

此外，可以使用特定类型 null 来指示一个未定义的地址，例如用于地址类型变量的初始化中。

例：

```
//把整型与开放式类型——地址类型相关联
type integer address;
:
//用 null 初始化新的地址类型变量
var address MySUTentity :=null;
:
//接收一个地址值，并把它赋值给变量 MySUTentity
PCO.receive(address:*) ->value MySUTentity;
:
//在发送模板 MyResult 的操作中接收地址的用法
PCO.send(MyResult) to MySUTentity;
:
//在接收一个确认模板操作中，接收地址的用法
PCO.receive(MyConfirmation) from MySUTentity;
```

7.1.7 成分引用

成分引用是对在一个测试用例执行时被创建的测试成分的唯一引用。这个唯一引用在一个成分被创建时由测试系统产生，也就是说，一个成分引用是 create 操作的结果。另外，还有三个特殊的成分引用：预定义操作 system(返回标识测试系统接口的端口的成分引用)、mtc(返回 MTC 的成分引用)和 self(返回调用 self 的成分引用)。

在配置 connect、map 和 start 操作中，使用成分引用建立测试配置，而出于编址的目的，应在连接到测试成分端口(而非测试系统接口)的通信操作的 from、to 和 sender 部分中使用成分引用。

此外，特殊类型 null 可用来指示一个未定义的成分引用，例如用于处理成分引用的变量的初始化。

成分引用的实际数据表示方法由测试系统来进行外部解析，这也就允许独立于任意实际 TTCN-3 运行环境描述抽象测试用例，换句话说，就是 TTCN-3 并不对有关测试成分处理和标识的一个测试系统实现加以限制。

注意：一个成分引用包括成分类型信息。举个例子，这就意味着用于处理成分引用的变量在其声明中必须使用相符的成分类型名。

例：

```
//一个成分类型定义
type component MyCompType {
    port PortTypeOne PCO1;
    port PortTypeTwo PCO2
}
//为处理 MyCompType 类型成分的引用声明两个变量，并创建这个类型的一个成分
var MyCompType MyCompInst := MyCompType.create;
//在配置操作中成分引用的用法
//总是引用上面已经创建的成分
connect(self:MyPCO1,MyCompInst:PCO1);
map(MyCompInst:PCO2,system:ExtPCO1);
MyCompInst.start(MyBehavior(self)); //self 被作为参数传递给 Behavior
//在 from-和 to-子句中的成分引用的用法
MyPCO1.receive from MyCompInst;
:
MyPCO2.receive(integer:?) ->sender MyCompInst;
:
MyPCO1.receive(MyTemplate)from MyCompInst;
:
MPCO2.send(integer:5) to MyCompInst;
//下面的例子解释了端口 PCO1 上一对多连接的情况,此时可以从不同类型 CompType1、CompType2
  和 CompType3
//的多个成分类型接收 M1 的值，且重新获得发送者。在这种情况下，可以使用如下方案:
:
var M1 MyMessage, MyResult;
var MyCompType1 MyInst1 := null;
var MyCompType2 MyInst2 :=null;
var MyCompType3 MyInst3 := null;
:
alt {
    [] PCO1.receive(M1:?)from MyInst1 ->value MyMessage sender MyInst1 {}
    [] PCO1.receive(M1:?)from MyInst2 ->value MyMessage sender MyInst2 {}
    []PCO1.receive(M1:?)from MyInst3 ->value MyMessage sender MyInst3 {}
}
:
MyResult := MyMessageHandling(MyMessage);  //一些结果从函数中获得
:
if (MyInst1 !=null) {PCO1.send(MyResult)    to MyInst1};
if (MyInst2 !=null) {PCO1.send(MyResult)    to MyInst2};
if (MyInst3 !=null) {PCO1.send(MyResult)    to MyInst3};
:
```

7.1.8 定义测试系统接口

从概念上讲，因为成分类型定义和测试系统定义具有相同的形式(都是定义可能的连接点的端口集合)，因此成分类型定义被用来定义测试系统接口。

```
type component MyISDNTestSystemInterface
{
    port    MyBchannelInterfaceType
    port    MyCchannelInterfaceType
    port    MyDchannelInterfaceType
}
```

通常，定义测试系统接口的一个成分类型引用应该与使用多个测试成分的每个测试用例都相关联。当测试用例启动执行时，测试系统接口的端口应该与 MTC 一起由该系统自动初始化，即从模块的控制部分调用该测试用例。

返回测试系统接口的成分引用的操作是 system，这个操作被用于对测试系统端口进行编址。

例：

```
map(MyMTCComponent:Port2,system:PCO1);
```

在测试用例执行时初始化的唯一测试成分是在 MTC 的情况下，测试系统接口不需要与测试用例相关联。在这种情况下，与 MTC 相关联的成分类型定义隐式地定义了相应的测试系统接口。

7.2 配置操作

配置操作用于建立和控制测试成分。这些操作应仅用于 TTCN-3 测试用例、函数和可选步(也就是说，不能用于模块控制部分)。

TTCN-3 配置操作一览表如表 7-1 所示。

表 7-1 TTCN-3 配置操作一览表

配置操作	
语句	操作名
创建并行测试成分	create
连接一个测试成分到另一个	connect
断开两个测试成分的连接	disconnect
映射成分端口到测试接口端口	Map
去除端口到测试系统接口的映射	unmap
获得 MTC 地址	Mtc
获得测试系统接口地址	system
获得自身地址	Self
启动测试成分的执行	Start
停止测试成分的执行	Stop
检查一个 PTC 的终止	running
等待一个 PTC 的终止	Done

7.2.1 创建操作

MTC 是测试用例启动时自动创建的唯一一个测试成分，所有其他的测试成分(并行测试成分，**PTCs**)都应该在测试执行期间用 **create** 操作显式地创建。一个测试成分被创建时，带有它所有的端口集合，且这些端口的输入队列为空。而且，如果定义一个端口类型为 **in** 或 **inout**，该端口将处于监听状态(**listening state**)，准备接收连接上的流量。

当显式或隐式创建测试成分时，所有成分变量和定时器重新设置它们的初值(如果有的话)，且重新指派所有成分常量的值。

因为每个测试用例终止时隐式地销毁所有的测试成分，在调用每个测试用例时，应该完全地创建它所需要的测试成分和连接的配置。

```
//这个例子声明了一个地址类型变量
//用来存储一个类型为 MyComponentType 的新创建的测试成分的引用
//它是 create 操作的结果
:
var MyComponenttype MyNewComponent;
:
MyNewComponent :=MyComponentType.create;
:
```

create 操作会返回一个新创建实例的唯一的成分引用(**component reference**)，典型地，该成分的唯一引用将保存在一个变量中，且可用于连接实例和通信目的(如发送和接收)。

可以在行为定义的任何地方创建测试成分，提供有关动态配置全部的灵活特性(即任意测试成分可以创建任意其他 **PTC**)。测试成分引用的可见性(**visibility**)应遵循与变量相同的范围规则，为了可以在测试成分的创建范围之外引用它们，测试成分引用应作为一条消息中的一个参数或字段来传递。

7.2.2　连接和映射操作

一个测试成分的端口可以连接到另一个测试成分或测试系统接口的端口。在两个测试成分连接的情况下，应使用连接(**connect**)操作。当连接一个测试成分到一个测试系统接口时，应使用映射(**map**)操作。**connect** 操作直接把一个端口连接到另一个端口，即一个端口的输出口连到另一个端口的输入口，而输入口则连到另一个端口的输出口。另一方面，**map** 操作可以完全看成定义如何引用通信流的名字翻译操作。

连接和映射操作的图示如图 7-6 所示。

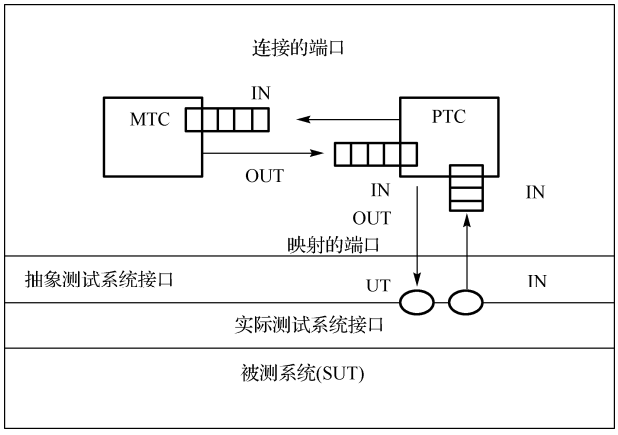


图 7-6　连接和映射操作的图示

伴随着 **connect** 操作和 **map** 操作，通过对被连接的测试成分的成分引用和连接到的端口名字来标识被连接的端口。

mtc 操作标识主测试成分 **MTC**，**system** 操作标识测试系统接口，且这两个操作都可用来标识和连接端口。

connect 和 **map** 操作都可以被除模块控制部分之外的任意行为定义调用。然而，在它们被调用之前，要被连接的测试成分应该已经创建了，且它们的成分引用和相关的端口名字都应该是已知的。

connect 和 **map** 操作都允许把一个端口连接到一个以上的其他端口，但不允许连接到一个映射端口或映射到一个已连接的端口。

例：

```
//假设在相应的端口类型和成分类型定义中正确定义和声明了端口 Port1、Port2、Port3 和 PCO1
:
var MyComponentType MyNewPTC;
MyNewPTC := MyComponentType.create;
:
:
connect(MyNewPTC:Port1, mtc:Port3);
map(MyNewPTC:Port2, system:PCO1);
:
:

//在这个例子中，创建了类型 MyComponentType 的一个新的成分，并把对它的引用存储在变量
  MyNewPTC 中。然后在 connect 操作中，这个新成分的 Port1 端口与 MTC 的 Port3 端口连接。
  最后，利用 map 操作，将新成分的 Port2 端口连接到测试系统接口的 PCO1 端口
```

对于 **connect** 和 **map** 操作，只允许一致性的连接。

具有如下假设。

(1) 要连接的端口为 PORT1 和 PORT2。

(2) T1 输入方向的消息和过程；inlist-PORT1 定义端口 PORT1。

(3) outlist-PORT1 定义端口 PORT1 输出方向的消息和过程。

(4) inlist-PORT2 定义端口 PORT2 输入方向的消息和过程。

(5) outlist-PORT2 定义端口 PORT2 输出方向的消息和过程。

connect 操作被允许，当且仅当：

outlist-PORT1⊆inlist-PORT2 且 outlist-PORT2⊆inlist-PORT1。

map 操作(假设 PORT2 是测试系统接口端口)被允许，当且仅当：

outlist-PORT1⊆outlist-PORT2 且 inlist-PORT2⊆inlist-PORT1。

在所有其他的情况下，都不允许 **connect** 和 **map** 操作。因为 TTCN-3 允许动态配置和动态地址，所以在编译时，不是所有的一致性检查都可以静态地执行。所有不能在编译时执行的一致性检查，应该在运行时执行，且发生错误时将导致一个测试用例错误。

7.2.3 断开连接和取消映射操作

断开连接(**disconnect**)和取消映射(**unmap**)操作是 **connect** 和 **map** 操作的反向操作。它们执行操作来断开(以前已连接)测试成分端口的连接和取消(以前映射的)测试成分端口和测试系统接口中端口之间的映射。

如果有关的成分引用和有关的端口名是已知的，那么任意成分都可以调用 **disconnect** 和 **unmap** 操作。如果要被移去的连接和映射已经被事先创建了，那么 **disconnect** 和 **unmap** 操作只有一个作用。

例：

```
:
:
connect(MyNe wComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
:
:
disconnect(MyNewComponent:Port1, mtc:Port3);          //断开以前建立的连接
unmap(MyNewComponent:Port2, system:PCO1);             //取消以前做的映射
```

7.2.4　MTC、System 和 Self 操作

成分引用有两个操作——**mtc** 和 **system**，它们分别返回对主测试成分和测试系统接口的引用。此外，操作可以用于返回调用它的测试成分引用。

例：

```
var MyComponentType MyAddress;
MyAddress   := self;    //存储当前的成分引用
```

在成分引用上仅允许赋值和等值操作。

7.2.5　启动测试成分操作

一旦创建了一个 **PTC**，且把连接行为和这个 **PTC** 绑定，就必须启动这个 **PTC** 的执行，这是通过 **start** 操作来完成的(创建 **PTC** 并不启动这个测试成分行为的执行)。区别 **create** 和 **start** 操作的原因是，允许在实际运行测试成分之前完成连接操作。

Start 操作会把必需的行为绑定到测试成分，通过对已定义函数的引用来定义这个行为。

例：

```
//假设在相应的端口类型和成分类型定义中正确定义和声明了端口 Port1、Port2、Port3 和 PCO1
:
var MyComponentType MyNewPTC;
:
MyNewPTC := MyComponentType.create;                   //创建一个新测试成分
:
connect(MyNewPTC:Port1 , mtc:Port3);                  //新测试成分和
map(MyNewPTCt:Port2, system:PCO1);                    //它的环境的连接
:
:
MyNewPTC.start(MyPTCBehaviour());                     //启动新测试成分
:
```

在一个启动测试成分(**start**)操作中，调用的函数使用如下限制。

(1) 如果这个函数有参数，那么这些参数只能是输入(**in**)参数，即带有值的参数。

(2) 这个函数应该有一个 **runson** 子句定义来引用与新创建成分相同的成分类型。

(3) 不能将端口和定时器传入该函数。

注意：当创建测试成分时，由于输入(in)和输入/出(inout)端口启动监听，这是测试成分同时启动测试执行，因此在这些端口的输入队列中可能有消息等待被处理。

7.2.6 停止测试成分操作

通过使用停止测试成分语句(**stop**)，一个测试成分可以停止它自己的执行或是另一个测试成分的执行。如果一个测试成分不停止自身，而是被测试系统中另一个测试成分停止，那么使用这个被停止的成分引用来标识它。一个成分可以通过使用一个简单的 **stop** 执行语句来停止自身，也可以通过在 stop 操作中选择自身地址来停止自身，如通过使用 **self** 操作。停止测试成分操作(**stop**)没有变量(**arguments**)。

例 1：

注意 1：stop 操作可以用于 MTC 和 PTC(s)，而 create、start、running 和 done 操作只能用于 PTC(s)。

```
var MyComponentType MyComp := MyComponentType.create;//创建一个新的测试成分
MyComp.start(CompBehaviour());  //启动这个新的测试成分
:
if (date == "1.1.2003"){
    MyComp.stop                 //在 1.1.2003 时停止这个新的测试成分
}
:
if ( cond1 ) {
    :                           //一些行为
    self.stop                   //通过使用 self 操作，测试成分停止自身
}
else {
    stop                        //通过使用一个简单的 stop 操作，测试成分停止自身
}
:
```

如果被停止的测试成分是 **MTC**，那么所有仍然运行的剩余的 **PTCs** 也将被停止，且测试用例终止。

注意 2：一个 **PTC** 可以通过停止 **MTC** 来停止测试用例的执行。

当一个测试成分终止时，无论是显式地使用 **stop** 操作或到达最初启动该测试成分的函数中的 **return** 语句时，还是隐式地到达该测试成分行为定义的结尾，都应该释放所有资源。存储在一个已停止的成分引用中的任一变量将成为未定义的。

注意 3：未定义意味着值不能为任意计算流程所使用，也就是说，它与什么都无关，也不认为是空(**null**)。

关键字 **all** 仅能被 **MTC** 用来停止所有正在运行的 **PTCs**。在这种情况下，**MTC** 不会被停止，在 **stop** 语句之后，它继续它自身的执行。

例 2：

```
:
all component.stop        //MTC 停止测试用例所有的 PTCs，但不停止它自身的执行
:
```

注意 4：停止 **PTCs** 的具体机制在本文的讨论范围之外。

7.2.7 运行操作

运行操作(**running**)允许在一个测试成分上执行行为来确定在一个不同的测试成分上运行的行为是否已经完成，**running** 操作只能用于 **PTCs**。**running** 操作被认为是一个布尔表达式，因此

它返回一个布尔值来指出特定的测试成分(或所有的测试成分)是否已经终止。与 **done** 操作相比，**running** 操作可以自由地用在布尔表达式中。

当关键字 **all** 与 **running** 操作一起使用时，如果所有已启动且没有被另外一个测试成分显式停止的 **PTCs** 都在执行它的行为，则返回真(**true**)，否则返回假(**false**)。

当关键字 **any** 与 **running** 操作一起使用时，如果至少有一个 PTC 在执行它的行为，则返回真(**true**)，否则返回假(**false**)。

例：

```
if (PTC1.running)                          //在一个 if 语句中 running 的用法
{
    //Do something!
}
while (all component.running != true) { //在一个循环条件中 urunning 的用法
    MySpecialFunction()
}
```

7.2.8　完成操作

完成操作(**done**)允许在一个测试成分上执行行为去确定在一个不同的测试成分上运行的行为是否已经完成。**done** 操作只能用于 **PTCs**。

done 操作的使用方式应该与接收操作或超时操作(**timeout**)的一样，这就意味着它不会用在布尔表达式中，但它能用来决定 **alt** 语句中的一个选择对象或作为一个行为描述中的独立语句。在后面这种情况下，一个 **done** 操作可以被看成一个只有一个选择对象的 **alt** 语句的简写，也就是说，它具有阻塞语义，因而提供了被动等待测试成分终止的能力。

当关键字 **all** 与 **done** 操作一起使用时，如果没有 **PTCs** 在执行它的行为，则返回真(**true**)；如果没有 PTC 被创建或启动，它也返回真(**true**)；否则返回假(**false**)。

当关键字 **any** 与 **running** 操作一起使用时，如果至少有一个已启动但没有显式地被另一个测试成分被停止的 **PTC** 完成了其行为的执行，则返回真(**true**)，否则返回假(**false**)。

注意：TTCN-3 的 done 操作与 TTCN-2 的 DONE 操作有相同的语义。

例：

```
//选择对象中 done 操作的使用
:
alt {
    [] MyPTC. done {
        setverdict(pass)
    }
    [] any port.receive {
        goto alt
    }
}
:
//下面的 done 操作作为独立语句(stand-alone statement):
:
all component.done;
:
//有如下含义:
```

```
:
alt {
    [] all component.done {}
}
:
//因此，阻塞执行，直到所有并行测试成分都已经终止
```

7.2.9 使用成分数组

create、**connect**、**start** 和 **stop** 操作不能直接在成分数组上工作，作为代替，应作为参数来提供数组的一个特定元素。成分通过一个成分引用数组实现数组的作用，并把有关数组元素赋值给 **create** 操作的结果。

例：

```
//这个例子表示了如何通过使用循环、在一个成分引用数组中存储被创建的成分引用
//为成分创建(creating)、连接(connecting)和运行(running)数组的结果建模
 Testcase  MyTestCase () runs on MyMtcType  system  MyTestSystemInterface
{
    :
    var integer  i;
    var MyPTCType1  MyPtcType[11];
    :
    for (i:= 0; i<=10; i:=i+1)
    {
        MyPtc [i]    :=     MyPtcType1.create;
        connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCoordination);
        MyPtc[i].start(MyPtcBehaviour());
    }
    :
}
```

7.2.10 带有成分的 any 和 all 的使用总结

关键字 **any** 和 **all** 可以和表 7-2 给出的配置操作一起使用。

表 7-2 带有成分的 Any 和 All

操作	允许的		例子
	any	all	
create			
start			
running	是，但仅来自于 MTC	是，但仅来自于 MTC	any omponent.running all component.running
done	是，但仅来自于 MTC	是，但仅来自于 MTC	any component.done all component.done
stop		是，但仅来自于 MTC	all component.stop

思考题

1．TTCN 中怎样进行端口通信配置？

2．TTCN 中怎样定义测试系统接口？

3．TTCN 中的测试成分起什么作用？如何启动和停止测试成分？

第 8 章　测试描述与测试控制

本章介绍 TTCN-3 中的测试描述与测试控制相关的内容，其中测试描述主要包括显示属性、值的编码、扩展属性、属性的范围、属性的重写规则和改变引入语言元素的属性。

8.1　描述属性

可以使用 **with** 语句来关联属性和 TTCN-3 的语言要素。**with** 语句的参数（即实际属性）被简单地定义为一个自由文本串。

有以下四种属性。

（1）显示（**display**）：允许与特定的表示格式有关的显示属性的说明。

（2）编码（**encode**）：允许引用特定的编码规则。

（3）变量（**variant**）：允许引用特定的编码变量。

（4）扩展（**extension**）：允许用户定义的属性说明。

8.1.1　显示属性

所有的 TTCN-3 语言要素都可以有用来说明如何显示特别的语言要素的显示（**display**）属性，如在一个表格格式中显示特定的语言要素。

可以在 ES 201 873-2 [1]中找到用于表格（一致性）表示格式的与显示属性有关的专用属性串。

可以在 TR 101 873-3 [2]中找到用于图形表示格式的与显示属性有关的专用属性串。

其他的显示（**display**）属性可以由用户定义。

注意：因为用户定义的属性是非标准化的，所以不同工具对这些属性的解释可能不同，甚至并不支持用户定义的属性。

8.1.2　值的编码

编码规则定义一个特别的值、模板等如何被编码，如何在一个通信端口上被传输以及如何解码接收到的信号。TTCN-3 没有默认的编码机制，这就意味着以 TTCN-3 的一些外部规则来定义编码规则或编码指令。

在 TTCN-3 中，可以使用编码（**encode**）属性和变量（**variant**）属性来说明通用的或特别的编码规则。

编码（**encode**）属性允许用于 TTCN-3 定义中某些引用的编码规则或编码指令的关联。定义实际编码规则的属性串（如 **prose**, **functions** 等）在本文讨论范围之外。如果没有引用特定的规则，那么编码应该是各实现自身的事。

在大多数情况下，用层次的方法使用编码属性。顶层是完整的模块，下面一层是一个组，最下层是一个单个的类型或定义。

（1）模块（**module**）：编码适用于模块中定义的所有类型，包括 TTCN-3 类型（嵌入类型）。

（2）组（**group**）：编码适用于一组用户定义的类型定义。

（3）类型或定义（**type or definition**）：编码适用于一个单个的用户定义的类型或定义。

(4)字段(**field**)：编码适用于记录类型(**record**)、集合类型(**set**)或模板(**template**)中的一个字段。

例 1：

```
module  MyTTCNmodule
{
:
    import from MySecondModule {
    type MyRecord
}
with
{
    encode "MyRule 1"
}                                  //将根据 MyRule 1 编码 MyRecord 的实例
:
Type  charstring MyType;           //通常根据全局规则编码
:
group MyRecords
{:
type record MyPDU1
{
    Integer field1,                //将根据 Rule 3 编码 field1
    Boolean field2,                //将根据 Rule 3 编码 field2
    Mytype field3                  //将根据 Rule 3 编码 field3
}
with {encode (field1, field2) "Rule 3" }
  :
}
with {encode "Rule 2" }
}
with {
encode "Global encoding rule"
 }
```

使用变量(**variant**)属性描述当前说明是编码方案的一个改进，而不是它的替代。

例 2：

```
Module  MyTTCNmodule1
{    :
    type    charstring MyType;  //通常根据全局规则编码
    :
group MyRecords
{   :
    type record MyPDU1
{
    Nteger field1,  //将根据 Rule 2 编码 field1，使用编码变量"length form 3"
    Mytype field3   //将根据 Rule 2 编码 field3，使用任意可能的长度编码格式
}
with
```

```
    {
        variant (field1) "length form 3"
     }
    :
    }
    with {encode "Rule 2" }
    }
    with
    {
        encode "Global encoding rule"
    }
```

下列串是为简单的基本类型预定义的(标准化的)变量(**variant**)属性。

(1) 当用于整型数和枚举类型时，“8 比特”("8 bit")和“无符号 8 比特”("unsigned 8 bit")意味着整型值或与枚举类型关联的整型数在系统中按 8-比特(单字节)表示来处理。

(2) 当用于整型数和枚举类型时，“16 比特”("16bit")和“无符号 16 比特”("unsigned 16bit")意味着整型值或与枚举类型关联的整型数在系统中按 16-比特(双字节)表示来处理。

(3) 当用于整型数和枚举类型时，“32 比特”("32bit")和“无符号 32 比特”("unsigned 32bit")意味着整型值或与枚举类型有关的整型数在系统中按 32-比特(四字节)表示来处理。

(4) 当用于整型数和枚举类型时，“64 比特”("64bit")和“无符号 64 比特”("unsigned 64bit")意味着整型值或与枚举类型有关的整型数在系统中按 64-比特(八字节)表示来处理。

(5) 当"IEEE754 float"、"IEEE754 double"、"IEEE754 extended float"和"IEEE754 extended double" 用于浮点类型时，意味着根据 IEEE 754 标准来编码和解码值。

下列串为字符型(**char**)、通用字符型(**universal char**)、字符串型(**charstring**)和通用字符串型(**universal charstring**)(标准化)预定义的变量(**variant**)属性。

(1) 当用于通用字符(**char**)或通用字符串(**universal char**)类型时，"UTF-8"意味着都应该根据 ISO/IEC 10646 [6]中附录 R 定义的 UCS 转换格式(UTF-8)分别编码和解码值的每一个字符。

(2) 当用于通用字符(**char**)或通用字符串(**universal char**)类型时，" UCS-2"意味着都应该根据 UCS-2 编码表示形式(见 ISO/IEC 10646 [6]的 14.1 节)分别编码和解码值的每一个字符。

(3) 当用于通用字符(**char**)或通用字符串(**universal char**)类型时，"UTF-16"意味着都应该根据 ISO/IEC 10646 [6]中附录 Q 定义的 UCS 转换格式(UTF-16)分别编码和解码值的每一个字符。

(4) 当用于字符(**char**)、通用字符(**universal char**)、字符串(**charstring**)和通用字符串(**universal charstring**)类型时，"8 bit "意味着都应该根据 ISO/IEC 8859 (一个 8-比特编码)中描述的编码表示分别编码和解码值的每一个字符。

下列串是为结构化类型(标准化)预定义的变量(**variant**)属性：

当用于一个记录类型时，"IDL:fixed FORMAL/01-12-01 v.2.6"意味着把值作为一个 IDL 定点十进制值。

这些变量属性可以被用于与更多的高层描述的编码属性的结合。例如，在一个自身具有全局编码属性"BER:1997"的模块中，描述的带有"UTF-8"变量属性的一个通用字符串(**universal charstring**)，将导致串中值的每个字符先根据 UTF-8 规则被编码，然后根据更全局的 BER 规则编码这个 UTF-8 值。

如果想要描述无效的编码规则，那么可以按照与引用有效编码规则相同的方式，在该模块外部的引用源中描述这些无效的编码规则。

8.1.3 扩展属性

所有 TTCN-3 语言要素都可以带有用户描述扩展(**extension**)属性。

注意：因为用户定义的属性没有被标准化，因此不同厂商提供的工具对这些属性的解释可能不同或甚至不支持这些属性。

8.1.4 属性的范围

with 语句可以把属性和一个单个的语言要素关联在一起，也可以通过诸如列表属性语句中结构类型字段的之类的方法把属性和多个的语言要素关联在一起，而这个属性语句与一个单个类型定义关联或通过一个 **with** 语句与环境范围单元或语言要素组(**group**)关联。

例：

```
//将把 MyPDU1 作为 PDU 显示
type  record  MyPDU1 { … }
with {display "PDU"}
//将把 MyPDU1 作为带有特定的应用扩展属性 MyRule 的 PDU 显示
type  record  MyPDU2 { … }
with
{
    display "PDU";
    extension "MyRule"
}
//下列组定义 …
group MyPDUs {
    type record MyPDU3 { … }
    type record MyPDU4 { … }
}
with
{display "PDU"}
//将把组 MyPDUs 的所有类型作为 PDU 显示与下面的定义相同
group MyPDUs {
    type record MyPDU3 { … }    with {  display "PDU"}
    type record MyPDU4 { … }    with {  display "PDU"}
}
```

8.1.5 属性的重写规则

一个较低范围单元中的属性定义会重写(**overwriting**)较高范围的通用属性定义。

例 1：

```
type  record  MyRecordA
{
    :
}   with {encode "RuleA"}
//在下面，MyRecordA 根据规则 RuleA 编码，而不是 RuleB
type record MyRecordB
{
```

```
    :
    field MyRecordA
}
with {encode "RuleB"}
```

放在另一个 **with** 语句的范围内的 with 语句将重写最外面的那个 **with** 语句，这同样适用于带有组的 **with** 语句使用的情况。当重写方案用在引用与单个定义相结合时要小心。通用的规则指的是应根据属性出现的顺序指定和重写这些属性。

```
//with 语句重写方案使用的例子
group MyPDUs
{
    type record MyPDU1 { … }
    type record MyPDU2 { … }
group MySpecial PDUs
{
    type record MyPDU3 { … }
    type record MyPDU4 { … }
}
with{
extension "MySpecialRule"
}   //MyPDU3 和 MyPDU4 将具有特定的应用扩展属性//MySpecialRule
}
with
{
display  "PDU";//组 MPDUs 的所有类型都将作为 PDU 被显示，且(如果没有被重写)具有扩
                展属性 MyRule
extension "MyRule";
}
//与下面相同…
group MyPDUs
{
    Type  record  MyPDU1 { … }  with {display "PDU"; extension "MyRule" }
    Type  record  MyPDU2 { … }  with {display "PDU"; extension "MyRule" }
group MySpecial  PDUs
{
    type  record  MyPDU3 { … } with {display "PDU"; extension "MySpecialRule" }
    type  record  MyPDU4 { … }with {display "PDU"; extension "MySpecialRule" }
}
}
```

通过使用 override 指令，可在较高范围中重写较低范围的属性定义。

例 2：

```
Type  record  MyRecordA
{
    :
}   with    {encode "RuleA" }
//在下面，根据 RuleB 编码 MyRecordA
```

```
type record MyRecordB
{
    :
    fieldA  MyRecordA
}   with    {encode override "RuleB" }
```

override 指令迫使所有较低范围内的类型具有指定的属性。

8.1.6 改变引入语言元素的属性

通常，一个语言元素与它的属性一起被引入。在一些情况下，当引入语言元素时，这些属性可能不得不改变，例如，可以在一个模块中把一个类型显示为 ASP，而在引入它的另一个模块中应该显示为 PDU。对于这样的情况，允许在 **import** 语句中改变属性。

例：

```
import from MyModule {
    type MyType
}
with { display  "ASP" }          //MyType 将被显示为 ASP
mport from MyModule
{
    group MyGroup
}
with {
    display "PDU";               //通过默认，所有类型都将被显示为 PDU
    extension "MyRule"
}
```

8.2 测试用例

测试用例(**Test cases**)是函数的一个特殊种类。在一个模块控制部分，使用 **execute** 语句来启动测试用例，一个测试用例的执行结果总是一个 **verdicttype** 类型值。每个测试用例应该包含一个且仅一个 **MTC**，并在该测试用例定义的头部引用 **MTC** 类型。定义在测试用例主体中的行为是 **MTC** 的行为。

调用一个测试用例时创建 **MTC**，并实例化 **MTC** 端口和测试系统接口，测试用例定义中指定的行为在 MTC 上开始。所有这些行为将会被隐式地执行，也就是说，不带有明确的 **create** 和 **start** 操作。

在测试用例头部，为允许这些隐式操作提供信息有以下两部分内容。

(1)接口部分(必需的)：用关键字 **runs on** 表示，关键字 **runs on** 为 **MTC** 引用必要的成分类型，并使相关联的端口名在 **MTC** 行为中是可见的。

(2)测试系统部分(可选的)：用关键字 **system** 表示，关键字 **system** 引用为测试系统接口定义所需端口的成分类型。在测试执行期间，仅当 **MTC** 被实例化了，测试系统部分才应该被省略。在这种情况下，**MTC** 隐式地定义了测试系统接口端口。

例：

```
testcase    MyTestCaseOne()
runs on  MyMtcType1                  //定义 MTC 类型
system  MyTestSystemType             //使 TSI 的端口名对于 MTC 可见
```

```
{
    :
    //当测试用例被调用时，这里定义的行为在 MTC 上执行
}
//或者，一个仅 MTC 被实例化的测试用例
testcase    MyTestCaseTwo() runs on     MyMtcType2
{
    :
    //当测试用例被调用时，这里定义的行为在 MTC 上执行
}
```

8.3　测试判定操作

判定操作允许使用 **getverdict** 和 **setverdict** 操作来设置和取回判定。这些操作应该仅用于测试用例、可选步和函数中。TTCN-3 测试判定操作一览表如表 8-1 所示。

表 8-1　TTCN-3 测试判定操作一览表

测试判定操作	
语句	相关的关键字或符号
设置本地判定	**setverdict**
获得本地判定	**getverdict**

活动的配置的每个测试成分应该维护自己的本地判定。这个本地判定是测试成分实例化时为每个测试成分创建的对象，它用于在每个测试成分中(即在 MTC 和每个 PTC 中)追踪单个的判定。

注意：与 TTCN-2 不一样，TTCN-3 不能把最终的判定分配给一个测试成分，因此分配一个判定从不中断行为执行所在的测试成分的执行。如果必要的话，应使用语句来显式完成。

8.3.1　测试用例判定

另外，在每个测试成分(即在 MTC 和每个 PTC 中)终止执行时，有一个全局判定被更新，这个全局判定对于 **getverdict** 和 **setverdict** 操作来说是不可访问的。当测试用例终止执行时，这个测试用例返回判定值。如果这个返回的判定没有显式地保存在控制部分(如赋值给一个变量)，那么就它就被丢掉了。

注意：TTCN-3 不描述执行本地判定和测试用例判定的实际机制。这些机制是依赖于实现的。

判定之间关系的图示如图 8-1 所示。

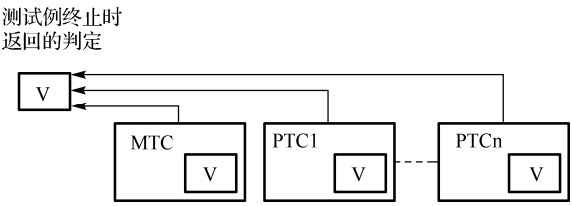


图 8-1　判定之间关系的图示

8.3.2　判定值和重写规则

判定有五个不同的值：通过(**pass**)、失败(**fail**)、不确定的(**inconc**)、空(**none**)和错误(**error**)，即判定类型 **verdicttype** 的不同值。

注意：inconc 意味着一个不确定的判定。

使用 setverdict 操作应仅带有值 **pass**、**fail**、**inconc** 和 **none**。

例 1：

```
setverdict (pass);
setverdict (inconc);
```

可以通过使用 **getverdict** 操作来取回本地判定的值。

例 2：

```
MyResult := getverdict; //MyResult 是一个判定类型 verdicttype 的变量
```

当一个测试成分被实例化时，它的本地判定对象被创建且设置为空值(**none**)。

当改变本地判定值时(即使用 **setverdict** 操作)，这个改变的结果将遵循表 8-2 中所列的重写规则(**overwriting rules**)。测试用例判定在一个测试成分终止时隐式地更新，这个隐式操作的结果将遵循表 8-2 中所列的重写规则。

表 8-2 判定的重写规则

判定的当前值	新判定的指派值			
	pass	inconc	fail	none
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail

例 3：

```
:
setverdict (pass);  //本地判定设为 pass
setverdict (fail);  //直到执行到这行，将导致本地判定值被改写为 fail
:                   //当 PTC 终止测试用例时，设判定为 fail
```

错误判定(**error**)是特别的，因为它是由测试系统设置的、用来指出发生了一个测试用例错误(即运行时间错误)。它不能由 **setverdict** 操作来设置，也不会通过 **getverdict** 操作来返回。没有其他判定值可以重写一个错误(**error**)判定，这就意味着一个错误判定(**error**)只能是一个执行(**execute**)测试用例操作的结果。

8.4 外部动作

在一些测试情况下，有些到 SUT 的电子接口可能被忽略，或从推理的角度讲是未知的(如管理接口)，但是刺激 SUT 完成特定行为时可能是必要的(如发送一条消息到测试系统)。而且测试执行人员也可能要求特定的动作(如改变环境条件，如温度、供电电压等)。

要求的动作可以定义为一个串。

例 1：

```
action("Send MyTemplate on lower PCO"); //SUT 动作的非形式化描述或作为描述 SUT
                                        所发消息结构的模板的一个引用
```

例 2：

```
action(MyTemplate);                     //这等价于 TTCN-2 的隐式发送语句
```

在这两种情况下，仅有被要求的动作自身的非形式化的描述，而没有 SUT 所做的用于触发这个动作的描述。

可以在测试用例、函数、可选步和模板控制中描述 SUT 动作。

8.5　模块控制部分

测试用例定义在模块定义部分，而模块控制部分管理它们的执行。如果要在行为定义中使用定义在一个模块控制部分的所有的变量、定时器等(如果有的话)，应该通过参数化来将它们传入测试用例。也就是说，TTCN-3 不支持任意种类的全程变量和定时器。

在每个测试用例启动时，测试配置将被重新设置。这就意味着当测试用例停止时，毁掉以前的测试用例中由 **create**、**connect** 等操作管理的所有成分和端口(因此对新测试用例来说是不“可见的”，**'visible'**)。

8.5.1　测试用例的执行

使用执行(**execute**)语句调用一个测试用例。作为测试用例的执行结果，返回一个测试判定(**none**、**pass**、**inconclusive**、**fail** 或 **error**)，且可以把这个判定赋值给一个变量来完成进一步的处理。

可选择：**execute** 语句允许通过定时器持续时间来监管测试用例。

例：

```
execute(MyTestCase1());        //执行 MyTestCase1,不存储返回的测试判定和时间监管
MyVerdict := execute(MyTestCase2());//执行MyTestCase2，把结果判定存储在变量MyVerdict 中
MyVerdict := execute(MyTestCase3(),5E-3); //执行 MyTestCase3，把结果判定存储在
                                                变量 MyVerdict 中
```

如果测试用例在 5ms 内没有终止，MyVerdict 将得到值'error'。

8.5.2　测试用例的终止(Termination of test cases)

测试用例和 MTC 一起终止。在 MTC(显式或隐式地)终止时，所有运行的并行测试成分(PTC)都将被测试系统终止。

注意 1：停止所有 PTC 的具体机制是一个特定的工具，所以在本文讨论范围之外。

根据规则，基于不同测试成分的本地判定来计算测试用例最终的判定。当测试成分自己终止时或被自身、另一个测试成分或测试系统停止时，该测试成分的实际本地判定变成它的最终的本地判定。

注意 2：为了避免由于被延时的 PTC 停止所引起的计算测试判定的紊乱情况，MTC 应该确保所有的 PTC 在停止自己以前已经停止了(使用 done 语句)。

8.5.3　测试用例的控制执行

例 1：

```
module MyTestSuite () {
:
control {
 :
//做 10 次这个测试
```

```
count:=0;
while (count < 10)
{
    execute (MySimpleTestCase1());
    count := count+1;
}
```

如果没有使用编程语句，那么默认为按照它们在模块控制部分中出现的先后顺序执行测试用例。

注意：这并不排除某些工具可能希望重写这个默认顺序而允许用户或工具去选择一个不同的执行顺序。测试用例返回一个 verdicttype 类型的值，因此，根据测试用例的结果控制执行顺序是可能的。

例 2：

```
if (execute (MySimpleTestCase()) == pass) { execute (MyGoOnTestCase) }
else {execute (MyErrorRecoveryTestCase) };
```

8.5.4 测试用例选择

可以使用布尔表达式选择和选择要执行的测试用例。当然，这包括返回布尔类型(boolean)值的函数的使用。

注意：这与 TTCN-2 中所谓的测试选择表达式是等价的。

例 1：

```
module MyTestSuite () {
    :
    control {
    :
    if (MySelectionExpression1()) {
    execute(MySimpleTestCase1());
    execute(MySimpleTestCase2());
    execute(MySimpleTestCase3());
}
if  (MySelectionExpression2()) {
    execute(MySimpleTestCase4());
    execute(MySimpleTestCase5());
    execute(MySimpleTestCase6());
}
 :
}
}
```

另外一种作为一个组来执行测试用例的方法是把这些测试用例集中在一个函数中，并从该模块的控制部分执行这个函数。

例 2：

```
function MyTestCaseGroup1()
{   execute(MySimpleTestCase1());
    execute(MySimpleTestCase2());
    execute(MySimpleTestCase3());
```

```
}
function MyTestCaseGroup2()
{   execute(MySimpleTestCase4());
    execute(MySimpleTestCase5());
    execute(MySimpleTestCase6());
}
:
control
{    if (MySelectionExpression1()) { MyTestCaseGroup1(); }
    if (MySelectionExpression1()) { MyTestCaseGroup2(); }
:
}
```

8.5.5　控制部分中定时器的使用

可以用定时器来监管测试用例的执行，可以在 **execute** 语句中使用一个显式的超时来完成这种监管。如果测试用例没有在定时器的持续时间内结束，那么测试用例执行的结果应该是一个错误判定，同时测试系统终止该测试用例。用于测试用例监管的这个定时器是一个系统定时器，且不必声明或启动它。

例 1：

```
MyReturnVal := execute (MyTestCase(), 7E-3);
//如果 7ms 内 MyTestCase 没有完成执行的话，返回的判定将会是错误(error)的，也可以显
  式地使用定时器操作来控制测试用例的执行
```

例 2：

```
//使用运行的定时器操作的例子
while (T1.running   or x<10)          //T1 是一个以前启动了的定时器
{   execute(MyTestCase());
    x := x+1;
}                                     //启动和超时操作使用的例子
timer   T1 := 1;
:
execute(MyTestCase1());
T1.start;
T1.timeout;                           //在执行下一个测试用例前暂停
execute(MyTestCase2());
```

思考题

1．TTCN 中怎样进行测试描述？请举例说明。

2．TTCN 中测试判定有几种？分别代表什么含义？

3．TTCN 中的模块控制的作用是什么？请举例说明。

第 9 章　系统测试及测试工具

本章介绍系统测试的基本概念、方法和工具。通过本章的学习，可以进一步了解系统测试的基本概念，以及针对具体测试需求选择相应的测试工具。

9.1　性能测试

本节首先讨论性能测试的基本概念，然后介绍常用的性能测试方法，最后通过一个案例介绍性能测试的应用。

9.1.1　性能测试的基本概念

一般来讲，性能是一种表明软件系统或构件对于及时性要求的符合程度的指标；性能是软件产品的一种特性，可以用时间来度量。性能的及时性通常用系统对请求做出响应所需要的时间来衡量。在国际标准化组织的定义中，响应时间被定义为：对计算机系统的查询或请求结束到一个响应开始所使用的时间。对某个系统或应用的用户来讲，响应时间就是用户必须等待服务所花的时间量。响应时间越短，用户就越满意，反之用户就越不满意。可以想象，当响应时间越来越长的时候，用户的耐心也会达到一定的限度，从而致使客户流失，最终导致公司效益下降。

性能测试主要检验软件是否达到需求规格说明书中规定的各类性能指标，并满足一些性能相关的约束和限制条件。性能测试的目的就是通过测试，确认软件是否满足产品的性能需求，同时发现系统中存在的性能瓶颈，并对系统进行优化优化。性能测试包括以下三个方面。

(1) 评估系统的能力。测试中得到的负荷和响应时间等数据可以被用于验证所计划的模型的能力，并帮助做出决策。

(2) 识别系统中的弱点。受控的负荷可以被增加到一个极端的水平并突破它，从而修复系统的瓶颈或薄弱的地方。

(3) 系统调优。重复运行测试，验证调整系统的活动得到了预期的结果，从而改进性能，检测软件中的问题。

9.1.2　性能测试方法

有多种可选择的性能测试方法，如基准测试、性能下降曲线分析法等。本节主要介绍基准法。关于性能测试的基准大体有以下几方面。

(1) 响应时间。这里的响应时间定义为从应用系统发出请求开始，到客户端接收到最后一个字节数据为止所消耗的时间。对用户来讲，响应时间的长短并没有绝对的区别。例如一个税务报账系统，用户每月使用一次该系统，每次进行数据录入等操作需要 2h 以上的时间，当用户选择提交后，即使系统在 20min 后才给出处理成功的消息，用户仍然不会认为系统的响应时间不能接受。因为相对于一个月才进行一次的操作来说，20min 是一个可以接受的等待时间。所以在进行性能测试的时候，合理的响应时间取决于实际的用户需求，而不能根据测试人员自己的设想来决定。

(2) 并发用户数。并发用户数一般是指在同一时间段内访问系统的用户数量。在实际的性能测试中，经常接触到的与并发用户数相关的概念还包括“系统用户数”和“同时在线用户人数”。例

如，某大学的网站有邮件服务及学生信息、选课系统等业务，基于这些业务的用户共有 10000 人，则这 10000 个用户就称为系统用户数。假设整个网站在最高峰时有 6000 人同时在线，则这 6000 人可以称为同时在线用户人数，还可以作为系统的最大并发用户数。

(3)吞吐量。吞吐量指单位时间内系统处理的客户请求数量。一般用请求数/秒或处理页面数/秒来衡量。吞吐量指标可以直接体现软件系统的性能。

(4)性能计数器。性能计数器是描述服务器或操作系统性能的一些数据指标。计数器在性能测试中发挥着监控和分析关键作用，尤其是在分析系统的可扩展性、进行性能瓶颈定位时，对计数器的取值的分析比较关键。比如 Windows 任务管理器就是一个性能计数器(如图 9-1 所示)，它提供了测试机 CPU 的使用率以及内存的使用率等信息。

图 9-1　Windows 任务管理器

性能测试基准法就是要根据上述基准，分别设计系统测试用例。通过测试试图回答系统响应时间、并发用户数、业务吞吐量等性能参数，为用下面的性能模型评价提供数据。

9.1.3　性能测试执行

可以将性能测试的执行分为以下三个阶段。

(1)计划阶段。该阶段主要完成下面各项工作。

- 定义目标并设置期望值。
- 收集系统和测试要求。
- 定义工作负载。
- 选择要收集的性能度量值。
- 标出要运行的测试并决定什么时候运行它们。
- 决定工具选项和生成负载。
- 编写测试计划，设计用户场景并创建测试脚本。

(2)测试阶段。该阶段主要完成以下工作。

- 做准备工作(如建立测试服务器或布置其他设备)。

- 运行测试。
- 收集数据。

(3)分析阶段。该阶段的主要工作如下。

- 分析结果。
- 改变系统以优化性能。
- 设计新的测试。

9.1.4 性能测试案例分析

下面介绍一个数据库应用系统性能测试的具体应用。现在的软件开发已从以前的单层结构进入了三层架构甚至多层架构的设计，数据库系统在整个系统中占的比重也越来越大。随着数据库开发在软件开发中的比重逐步提高，随之而来的问题也更加突出，因此对数据库的测试必须提升到一个新的高度上。

在数据库开发的过程中，为了测试应用程序对数据库的访问，应当在数据库中生成测试数据。因为当数据库中只有少量的数据时，系统可能不会出现问题，但是当数据库真正投入到使用中并产生大量数据时，问题就出现了。这往往是程序的编写问题，因此及早通过在数据库中生成大量数据来帮助开发人员尽快完善这部分性能是很必要的。然而，长期以来这些工作是靠手工来完成的，需要耗费大量的人力、物力。目前，有许多用于功能测试的自动化测试工具可供用户使用以节省测试时间、提高测试效率。本节结合现在比较流行的 JMeter 这一开源的自动化测试工具介绍数据库系统的性能测试。

(1)系统介绍：被测系统是一个分布式数据库系统 Testbase。该数据库采用 Oracle 数据库，Testbase 包括三张表，这里仅取其中一张名为 City 的表来说明测试过程。表的创建语句如下：

```
create table City (Country varchar(20) not null,
                   Name varchar(20) not null,
                   Des varchar(20) not null)
```

(2)测试目的：测试 Testbase 数据库的查询性能。

(3)测试工具的选择：JMeter。

(4)测试步骤如下。

安装必要的 JDBC 驱动。另外，要测试数据库的查询性能，必须用大量的数据来填充表，10 万～30 万条即可，可以自己编写程序来填充，也可以用专门的数据生成工具 DataFactory，这里不再赘述。

首先准备好 JMeter 测试工具(JMeter 是开源工具，可以在其官方网站：http://jakarta.apache.org/jmeter/index.html下载源代码和查看相应文档)，然后调用程序根目录下 bin 文件夹里的 jmeter.bat 批处理文件来启动 JMeter 的图形界面。

建立测试计划，测试计划描述了执行测试过程中 JMeter 的执行过程和步骤，一个完整的测试计划包括一个或多个线程组(Thread Groups)、逻辑控制(Logic Controller)、实例产生控制器(Sample Generating Controllers)、侦听器(Listener)、定时器(Timer)、比较(Assertions)、配置元素(Config Elements)。打开 JMeter 时，它已经建立一个默认的测试计划，一个 JMeter 应用的实例只能建立或者打开一个测试计划。

下一步需要建立线程组来模拟请求者数量，详细步骤如下。

(1)选中可视化界面中左边的测试计划(Test Plan)节点，单击右键，选择添加线程组(Add Thread Group)，界面右边将会出现其设置信息框。

(2)线程组里有以下三个和负载信息相关的属性参数。

- 线程数(Number of Threads)：设置发送请求的用户数目。

- Ramp-Up Period：每个请求发生的总时间间隔，单位是秒(s)。比如请求数目是 20，而这个参数是 100，那么每个请求之间的间隔就是 100/20，也就是 5s。
- 循环次数(Loop Count)：请求发生的重复次数，如果选择后面的“永远(Forever)”，那么请求将一直继续，如果不选择“永远”，而在输入框中输入数字，那么请求将重复指定的次数，如果输入 0，那么请求将执行一次。设置相应的参数，如图 9-2 所示。

图 9-2　创建 JMeter 线程组

下一步配置与数据库的连接，这里以 JDBC 请求的方式和 Oracle 数据库通信，在 DTest(线程组)处单击右键选择“添加—Sample—JDBC 请求”，在打开的页面里可以设置 SQL 语句的类型和具体的语句，此处设置为查询语句，如图 9-3 所示。

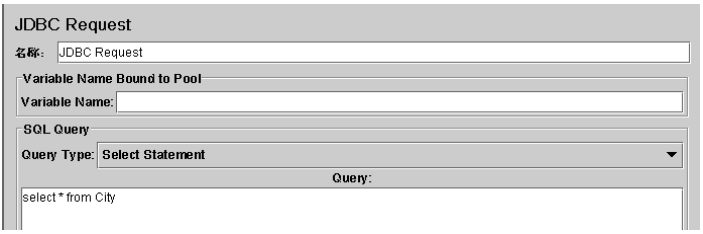


图 9-3　添加 JDBC 请求

然后配置数据库的连接信息，选择“添加配置文件——JDBC 连接配置”，此处可以设置一些必要的信息与数据库通信，如 DataBase URL、JDBC Driver Class，以及用户名和口令、连接池等信息。设置好后如图 9-4 所示。

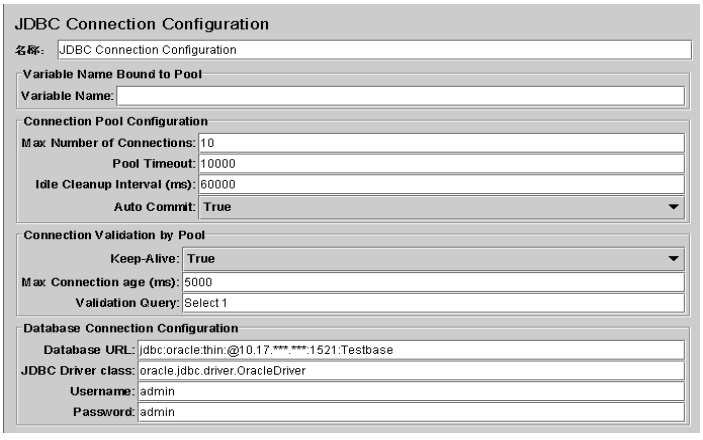


图 9-4　配置数据库连接

最后设置结果的查看方式，用图形方式来查看结果，选择“添加—监听器—图形结果”，结果如图 9-5 所示。

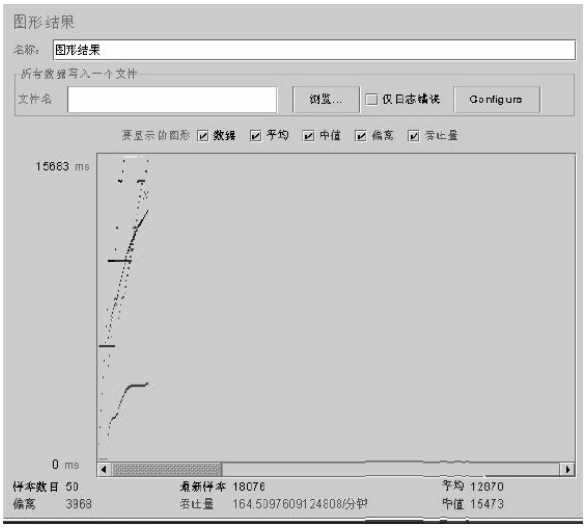


图 9-5 测试结果

在结果显示域，我们可以根据曲线或点的颜色的不同来分析系统的吞吐量等性能信息。这时可以增加用户负载，来查看不同负载时执行相同的查询对平均时间的影响。如果要分析查询语句的执行情况等信息还需要添加断言和断言结果，具体方法为“添加—断言—响应断言”和“添加—监听器—断言结果”。这样就可以看到断言的结果屏幕，并可以在该屏幕中指定一个数据文件，把断言数据写入数据文件。当运行测试对象的时候，应该看到用否定或肯定断言填充的断言结果屏幕，是肯定还是否定则取决于是否满足断言中的条件。成功的断言格式为：select * from City；如果断言结果为否定则显示：select * from City Test failed, text expected to contain/8/；如果结果没有响应则显示：select * from City Response was null。

本节用 JMeter 分析了 Oracle 数据库的查询性能，这仅仅是 JMeter 功能中很简单的部分，可以建立更加复杂的测试计划来应对不同的测试。JMeter 以图形和活动曲线的形式，提供关于系统性能的可视反馈。

9.2 压力测试(负载测试、并发测试)

本节介绍压力测试的相关概念和测试方法。

9.2.1 压力测试的基本概念

压力测试(Stress Testing)是指模拟巨大的工作负荷，以查看系统在峰值使用情况下是否可以正常运行。压力测试是通过逐步增加系统负载测试系统性能的变化，并最终确定在什么负载条件下系统性能处于失效状态，以此来获得系统性能提供的最大服务级别的测试。

压力测试方法具有如下特点。

- 压力测试是检查系统处于压力情况下的能力表现。顾名思义，压力测试就是通过不断增加系统压力，来检测系统在不同压力情况下所能够到达的工作能力和水平。比如，通过增加并发用户的数量，检测系统的服务能力和水平；通过增加文件记录数来检测数据处理的能力和水平等。

- 压力测试一般通过模拟方法进行。压力测试是一种极端情况下的测试，所以为了捕获极端状态下的系统表现往往采用模拟方法进行。通常，在系统对内存和 CPU 利用率上进行模拟，以获得测量结果。如将压力的基准设定为：内存使用率达到 75%以上、CPU 使用率达到 75%以上，并在此观测系统响应时间、系统有无错误产生。除了对内存和 CPU 的使用率进行设定外，数据库的连接数量、数据库服务器的 CPU 利用率等也都可以作为压力测试的依据。
- 压力测试一般用于测试系统的稳定性。如果一个系统能够在压力环境下稳定运行一段时间，那么该系统在普遍的运行环境下就应该可以达到令人满意的稳定程度。在压力测试中，通常会考察系统在压力下是否会出现错误等方面的问题。

压力测试与性能测试的联系与区别：压力测试是用来保证产品发布后系统能够满足用户需求，关注的重点是系统整体；性能测试可以发生在各个测试阶段，即使是在单元层，对一个单独模块的性能也可以进行评估。压力测试是通过确定一个系统的瓶颈，来获得系统能提供的最大服务级别的测试。性能测试是检测系统在一定负荷下的表现，是正常能力的表现；而压力测试是极端情况下的系统能力的表现。例如对一个网站进行测试，模拟 10～50 个用户同时在线并观测系统表现，就是在进行常规性能测试；当用户增加到系统出项瓶颈时，如达到 1000 乃至上万个用户时，就变成了压力测试。

压力测试和负载测试(Load Test)：负载测试是通过逐步增加系统工作量，测试系统能力的变化，并最终确定在满足功能指标的情况下，系统所能承受的最大工作量的测试。压力测试实质上就是一种特定类型的负载测试。

压力测试和并发性测试：并发性测试是一种测试手段，在压力测试中可以利用并发测试来进行压力测试。

9.2.2　压力测试方法

压力测试应该尽可能逼真地模拟系统环境。对于实时系统，测试者应该以正常和超常的速度输入要处理的事务，从而进行压力测试。批处理的压力测试可以利用大批量的批事务进行，被测事务中应该包括错误条件。压力测试中使用的事务可以通过如下三种途径获得。

(1) 测试数据生成器。

(2) 由测试小组创建的测试事务。

(3) 原来在系统环境中处理过的事务。

压力测试中应该模拟真实的运行环境。测试者应该使用标准文档，输入事务的人员或者系统使用人员应该和系统产品化之后的参与人员一样。实时系统应该测试其扩展的时间段，批处理系统应该使用多于一个事务的批量进行测试。

下面以对 Web 服务进行压力测试为例。设计压力测试时，要让它们以某种特定的方式运行代码。其目的是观测被测 Web 服务能否完成其基本功能，并且在压力的情况下仍能正常运行。有效的压力测试将可以采用以下测试手段。

(1) 重复(Repetition)测试：重复测试就是一遍又一遍地执行某个操作或功能，比如重复调用一个 Web 服务。压力测试的一项任务就是确定在极端情况下一个操作能否正常执行，并且能否持续不断地在每次执行时都正常。这对于推断一个产品是否适用于某种生产情况至关重要，客户通常会重复使用产品。重复测试往往与其他测试手段一并使用。

(2) 并发(Concurrency)测试：并发是同时执行多个操作的行为，即在同一时间执行多个测试线程。例如，在同一个服务器上同时调用许多 Web 服务。并发测试原则上不一定适用于所有产品(比如无状态服务)，但多数软件都具有某个并发行为或多线程行为元素，这一点只能通过执行多个代码测试用例才能得到测试结果。

(3)量级(Magnitude)增加：压力测试可以重复执行一个操作，但是操作自身也要尽量给产品增加负担。例如一个 Web 服务允许客户机输入一条消息，测试人员可以通过模拟输入超长消息来使操作进行高强度的使用，即增加这个操作的量级。这个量级的确定总是与应用系统有关，可以通过查找产品的可配置参数来确定量级，例如数据量的大小、延迟时间的长度、输入速度以及输入的变化等。

(4)随机变化：该手段是指对上述测试手段进行随机组合，以便获得最佳的测试效果。例如，使用重复时，在重新启动或重新连接服务之前，可以改变重复操作间的时间间隔、重复的次数，也可以改变被重复的 Web 服务的顺序；使用并发时，可以改变一起执行的 Web 服务、同一时间运行的 Web 服务数目，也可以改变关于是运行许多不同的服务还是运行许多同样的实例的决定。量级测试时,每次重复测试都可以更改应用程序中出现的变量(例如发送各种大小的消息或数字输入值)。如果测试完全随机的话，因为很难一致地重现压力下的错误，所以一些系统使用基于一个固定随机种子的随机变化。这样，使用同一个种子，重现错误的机会就会更大。

9.2.3 压力测试执行

可以设计压力测试用例来测试应用系统的整体或部分能力。压力测试用例选取可以从以下几个方面考虑。

- 输入待处理事务来检查是否有足够的磁盘空间。
- 创造极端的网络负载。
- 制造系统溢出条件。

当应用系统所能正常处理的工作量并不确定时需要使用压力测试。压力测试意图通过对系统施加超负载事务量来达到破坏系统的目的。压力测试和在线应用程序非常类似，因为很难利用其他测试技术来模拟高容量的事务。压力测试的弱点在于准备测试的时间与在测试的实际执行过程中所消耗的资源数量都非常庞大。这些消耗需要与那些尚未识别的容量相关的风险进行权衡，通常在应用程序投入使用之前，这种衡量是无法进行的。

9.3 容量测试

本节介绍容量测试及其方法。

9.3.1 容量测试基本概念

所谓的容量测试(Volume Testing)，是指采用特定的手段测试系统能够承载处理任务的极限值所从事的测试工作。这里的特定手段是指测试人员根据实际运行中可能出现极限，制造相对应的任务组合，来激发系统出现极限的情况。

容量测试的目的是使系统承受超额的数据容量来发现它能否正确处理，通过测试预先分析出反映软件系统应用特征的某项指标的极限值(如最大并发用户数、数据库记录数等)，确定系统在其极限值状态下是否还能保持主要功能正常运行。容量测试还将确定测试对象在给定时间内能够持续处理的最大负载或工作量。

对软件容量的测试，能让软件开发商或用户了解该软件系统的承载能力或提供服务的能力，如电子商务网站所能承受的、同时进行交易或结算的在线用户数。知道了系统的实际容量，如果不能满足设计要求，就应该寻求新的技术解决方案，以提高系统的容量。有了对软件负载的准确预测，不仅能对软件系统在实际使用中的性能状况充满信心，同时也可以帮助用户经济地规划应用系统，优化系统的部署。

与容量测试十分相近的概念是压力测试。二者都是检测系统在特定情况下，能够承担的极限值。然而，两者的侧重点有所不同，压力测试主要是使系统承受速度方面的超额负载，例如一个短时间之内的吞吐量。容量测试关注的是数据方面的承受能力，并且它的目的是显示系统可以处理的数据容量。容量测试往往应用于数据库方面的测试，数据库容量测试使测试对象处理大量的数据，以确定是否达到了将使软件发生故障的极限。容量测试还将确定测试对象在给定时间内能够持续处理的最大负载或工作量。例如，如果测试对象正在为生成一份报表而处理一组数据库记录，那么容量测试就会使用一个大型的测试数据库，检验该软件是否正常运行并生成了正确的报表。做这种测试通常通过书写存储过程向数据库某个表中插入一定数量的记录，计算相关页面的调用时间。比如在电子商务系统中，通过 insert customer 向 user 表中插入“10000”数据，看其是否可以正常显示顾客信息列表页面，如果要求达到最多可以处理 100000 个客户，但是顾客信息列表页面不能在规定的时间内显示出来，就需要调整程序中的 SQL 查询语句；如果在规定的时间内显示出来，可以将用户数分别提高到 20000、50000、100000 进行测试。

更确切地说，压力测试可以看成容量测试、性能测试和可靠性测试的一种手段，不是直接的测试目标。压力测试的重点在于发现功能性测试所不易发现的系统方面的缺陷，而容量测试和性能测试是系统测试的主要目标内容，也就是确定软件产品或系统的非功能性方面的质量特征，包括具体的特征值。容量测试和性能测试更着力于提供性能与容量方面的数据，为软件系统部署、维护、质量改进服务，并可以帮助市场定位、销售人员对客户的解释、广告宣传等。

压力测试、容量测试和性能测试的测试方法相通，在实际测试工作中，往往结合起来进行以提高测试效率。一般会设置专门的性能测试实验室完成这些工作，即使用虚拟的手段模拟实际操作，所需要的客户端有时还很大，所以性能测试实验室的投资较大。对于许多中小型软件公司，可以委托第三方完成性能测试，可以大幅降低成本。

9.3.2　容量测试方法

进行容量测试的首要任务就是确定被测系统数据量的极限，即容量极限。这些数据可以是数据库所能容纳的最大值，可以是一次处理所能允许的最大数据量，等等。系统出现问题，通常是发生在极限数据量产生或临界产生的情况下，这时容易造成磁盘数据的丢失、缓冲区溢出等一些问题。为了更清楚地说明如何确定容量的极限值，参看图 9-6。

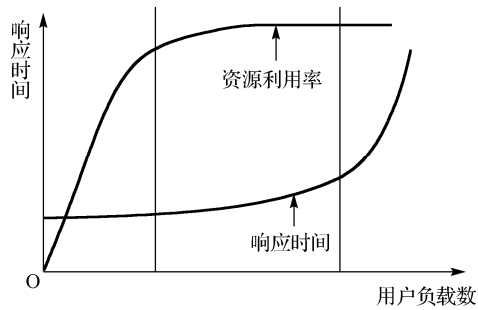


图 9-6　资源利用率、响应时间、用户负载数的关系

图 9-6 中反映了资源利用率、响应时间与用户负载数之间的关系。可以看到，用户负载数增加，响应时间也缓慢地增加，而资源利用率几乎是线性增长。这是因为当应用做更多的工作时，它需要更多的资源。一旦资源利用率接近百分之百时，就会出现一个有趣的现象，就是响应以指数曲线方式下降，这点在容量评估中被称为饱和点。饱和点是指所有性能指标都不满足，随后应用发生恐慌的时间点。执行容量评估的目标是保证用户知道这一点在哪里，并且永远不要出现这种情况。在这种负载发生前，管理者应优化系统或者增加适当额外的硬件。

为了确定容量极限，可以进行一些组合条件下的测试，如核实测试对象在以下高容量条件下能否正常运行：

- 链接或模拟了最大(实际或实际允许)数量的客户机。

- 所有客户机在长时间内执行相同的、可能性能不稳定的重要业务功能。
- 已达到最大的数据库大小(实际的或按比例缩放的)，而一起同时执行多个查询或报表事务。

当然需要注意，不能简单地说在某一标准配置服务器上运行某软件的容量是多少，选用不同的加载策略可以反映不同状况下的容量。举个简单的例子，网上聊天室软件的容量是多少？在一个聊天室内有 1000 个用户，和 100 个聊天室每个聊天室内有 10 个用户，同样都是 1000 个用户，在性能表现上可能会出现很大的不同，在服务器端的数据输出量、传输量更是截然不同。在更复杂的系统内，就需要分别为多种情况提供相应的容量数据作为参考。

9.3.3 容量测试执行

开始进行容量测试的第一步也和其他测试工作一样，通常是获取测试需求。系统测试需求确定测试的内容，即测试的具体对象。测试需求主要来源于各种需求配置项，它可能是一个需求规格说明书，或是由场景、用例模型、补充规约等组成的一个集合。其中，容量测试需求来自于测试对象的指定用户数和业务量。容量需求通常出现在需求规格说明书中的基本性能指标、极限数据量要求和测试环境部分。

容量测试常用的用例设计方法有规范导出法、边界值分析、错误猜测法。进行容量测试一般可以通过以下几个步骤来完成。

- 分析系统的外部数据源，并进行分类。
- 对每类数据源分析可能的容量限制，对于记录类型数据需要分析记录长度限制，记录中每个域长度限制和记录数量限制。
- 对每个类型数据源，构造大容量数据对系统进行测试。
- 分析测试结果，并与期望值比较，确定目前系统的容量瓶颈。
- 对系统进行优化并重复以上四步，直到系统达到期望的容量处理能力。

常见的容量测试例子包括：

- 处理数据敏感操作时进行的相关数据比较。
- 使用编译器编译一个极其庞大的源程序。
- 使用一个链接编辑器编辑一个包含成千上万模块的程序。
- 一个电路模拟器模拟包含成千上万块的电路。
- 一个操作系统的任务队列被充满。
- 一个测试形式的系统被灌输了大量文档格式。
- 互联网中庞大的 E-mail 信息和文件信息。

9.3.4 一个容量测试案例分析

容量测试用于研究程序加载非常大量的数据时、处理很少量或很大量数据任务时的运行情况，这一测试主要关注一次处理合理需求的大量数据，而且在一段较长时间内高频率地重复任务。对于像银行终端监控系统这样的产品来讲，容量测试是至关重要的。在下面的内容中将选取一个银行系统完成容量测试的案例进行简单的分析。

首先根据某银行终端监控系统的需求说明，做出如下分析：

- 服务器支持挂接 100 台业务前置机。
- 每台前置机支持挂接 200 台字符终端。
- 字符终端有两种登录前置机的模式，即终端服务器模式和 Telnet 模式。
- 不同的用户操作仅反映为请求数据量的不同。

- 不同的配置包括不同的系统版本(如 SCO、SOLARIS 等)、不同的 shell(shell、cshell、kshell)。

对应上面五条容量需求分析，分别制定如下策略：

对于需求 1、2，挂接 100 台业务前置机，200 台字符终端的容量环境不可能真实地构造，所以这里采取虚拟用户数量的方式，多台业务前置机采用在一台前置机上绑定多个 IP 地址的方式实现，同时启动多个前置程序。对于需求 3 可以给出两种字符终端登录前置机的模式。对于需求 4，不同数据量可以执行不同的 shell 脚本来实现。实际上可以执行相同的脚本，而循环输出不同字节数的文本文件内容。最后，对于不同的系统版本，则只能逐一测试，因为谁也代替不了谁。当然，以用户实际使用的环境为重点。

测试工作离不开测试用例的设计。不完全、不彻底是软件测试的致命缺陷，任何程序只能进行少量而有限的测试。测试用例在此情况下产生，同时它也是软件测试系统化、工程化的产物。当明确了测试需求和策略后，设计用例只是一件顺水推舟的事。从测试需求可以提取出许多的测试点，而测试用例则是测试点的组合。怎样组合呢？可以参考这样一个原则：一个测试用例是为验证某一个具体的需求，在一个测试场景下，进行的若干必要操作的最小集合。也就是说，只要明确地定义目的、场景、操作，就形成了用例的基本轮廓。再加上不同类型测试必需的测试要素，就构成了完整的测试用例。对于容量测试来说，测试要素无外乎容量值、一定容量下正常工作的标准等。表 9-1 给出一个容量测试用例模板的例子。

表 9-1　容量测试用例模板

<table>
<tr><td colspan="2" rowspan="4">系统测试用例</td><td>用例标识</td><td></td></tr>
<tr><td>用例类型</td><td>容量测试</td></tr>
<tr><td>编写人</td><td></td></tr>
<tr><td>编写日期</td><td></td></tr>
<tr><td colspan="2">测试目的</td><td colspan="2"></td></tr>
<tr><td colspan="2">需求可追踪性</td><td colspan="2"></td></tr>
<tr><td colspan="2">测试约束</td><td colspan="2"></td></tr>
<tr><td colspan="2">测试环境</td><td colspan="2"></td></tr>
<tr><td colspan="2">测试工具</td><td colspan="2"></td></tr>
<tr><td colspan="2">初始化</td><td colspan="2"></td></tr>
<tr><td>N.</td><td>操作步骤及输入</td><td colspan="2">预期结果及通过准则</td></tr>
<tr><td>1.</td><td></td><td colspan="2"></td></tr>
<tr><td>2.</td><td></td><td colspan="2"></td></tr>
<tr><td colspan="2">测试结果问题报告标识码</td><td colspan="2"></td></tr>
<tr><td colspan="4">审核：</td></tr>
<tr><td colspan="4">附注：</td></tr>
</table>

作为前面例子的延续，下面简单说明各栏目的填法。

(1)测试目的：需要验证的测试需求，如“在 XXX 的容量条件下，前置程序是否能正常工作”。

(2)需求可追踪性：对应测试需求的标识号。

(3)测试约束条件：本次测试需遵循的制约条件，如终端以终端服务器模式登录到主机。

(4)测试环境：前置程序的版本等。

(5)测试工具：来源于测试策略，如前面提到的终端服务器模拟程序。

(6)初始化：在测试前需做的准备工作。

(7)操作步骤及输入：如终端登录，不同的数据量操作等。

(8)预期结果及通过准则：一定容量下正常工作的标准，如正常录像、正常压缩传送、资源占用率等。

对于不同的容量条件，因为测试场景不一样，建议编写不同的用例。

执行出了测试策略，也完成了测试用例的设计，接下来的事就是真正地去操作，即测试执行。容量测试执行的具体步骤与其他类型测试没有太多区别，大致分为以下七步。

(1)按用例中测试环境的描述建立测试系统。

(2)准备测试过程，合理的组织用例的测试流程。

(3)根据用例中“初始化”内容运行初始化过程。

(4)执行测试，从终止的测试恢复。

(5)验证预期结果，对应测试用例中描述的测试目的。

(6)调查突发结果，即对异常现象进行研究，适当地进行一些回归测试。

(7)记录问题报告。

以上便是这次容量测试的全过程。注意，在这个过程中所产生的各种信息要妥善管理，因为容量测试的可重复性是很高的。

9.4 健壮性测试

本节讨论健壮性测试及其方法。

9.4.1 健壮性测试基本概念

健壮性测试(Robustness Testing)主要用于测试系统抵御错误的能力。这里的错误通常指的是由于设计缺陷而带来的系统错误。测试的重点为当出现故障时，能否自动恢复或忽略故障继续运行。

对于一般的软件企业来讲，由于受到开发成本、时间和人员等条件的约束，经常把软件测试的关注点放在功能正确性上面，往往分配少量的资源用于确定系统在异常处理方面，从而忽略系统健壮性。这个矛盾随着软件应用的日益普遍而异常突出，所以一个好的软件系统必须经过健壮性测试之后才能最终交付给用户。

健壮性有两层含义：一是高可靠性，二是从错误中恢复的能力。前者体现了软件系统的质量；后者体现了软件系统的适应性。二者也给测试工作提出了不同的测试要求，前者需要根据符合规格说明的数据选择测试用例，用于检测在正常情况下系统输出的正确性；后者需要在异常数据中选择测试用例，检测非正常情况下的系统行为。

9.4.2 健壮性测试方法

健壮性测试可以根据以下方面评价系统的健壮性。

- 通过：系统调用运行输入的参数产生预期的正常结果。
- 灾难性失效：这是系统健壮性测试中最严重的失效，这种失效只有通过系统重新引导才能得到解决。
- 重启失效：一个系统函数的调用没有返回，使得调用它的程序挂起或停止。
- 夭折失效：程序执行时由于异常输入，系统发出错误提示使程序中止。
- 沉寂失效：异常输入时，系统应当发出错误提示，但是测试结果却没有发生异常。
- 干扰失效：指系统异常时返回了错误的提示，但是该错误提示不是期望中的错误。

自动化实现上述测试内容需要把握以下原则。

- 可移植性：健壮性测试基准程序用于比较不同系统的健壮性，因此移植性是测试基准程序的基本要求。

- 覆盖率：理想的基准程序能够覆盖所有的系统模块，然而这种开销是巨大的。因此，一般选取使用频度最高的模块进行测试。
- 可扩展性：可扩展性体现在当需要扩展测试集时能够前后一致。这种可扩展性不仅指为已有模块增加测试集，还包括为新增加的模块增加测试集。
- 测试结果的记录：健壮性测试的目的是找出系统的不健壮性因素，因此应详细地记录测试结果。

健壮性是指在异常情况下，软件能够正常运行的能力。这一能力的评价在实际中难以量化，但可以通过对异常情况下，软件不能正常运行的情况进行评价。所以对于健壮性，可以采用输入错误、操作错误和环境错误的数量来衡量。因此，在设计健壮性测试时可以从以下几个方面考虑。

- 基于错误的策略。确认所有可能的错误源，为每一类错误开发错误注入技术。
- 基于覆盖率的策略。接口覆盖的数量、故障位置覆盖的数量、例外覆盖的数量。
- 基于失效的策略。用例设计故障是否被处理了，例外是否被处理了，一个组件中的失效是否影响另一个组件。

在进行健壮性测试时，常用的用例设计方法主要有三种：故障插入测试、变体测试和错误猜测法。健壮性测试方法通常需要构造一些不合理的输入来引诱软件出错，如输入错误的数据类型、输入定义域之外的数值等。

9.4.3　一个健壮性测试案例分析

为了更清楚地让读者了解什么是健壮性测试，在这里举个简单的例子。假如定义了两个变量 x_1 和 x_2，两个变量都有自己的取值范围，则写成如下这种形式：$a<x_1<b$；$c<x_2<d$；用坐标的形式表示，如图 9-7 所示。

在图 9-7 中，灰色区域外的 4 个点就是健壮性测试要重点考虑的情况。一般情况下，边界值分析的大部分讨论都直接适用于健壮性测试。健壮性测试最需要关注的部分不是输入，而是预期的输出。比如，当物理量超过其最大值时，会出现什么情况。如果是飞机机翼的迎角，则飞机可能失速，健壮性测试的主要价值是观察例外处理情况。

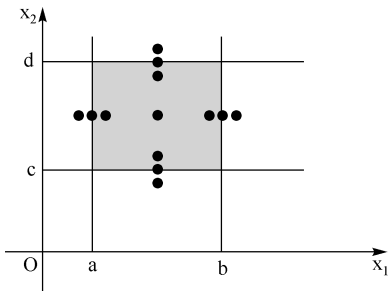


图 9-7　两变量函数的健壮性测试用例

9.5　安全性测试

本节讨论安全测试及其方法。

9.5.1　安全性测试基本概念

安全性测试是检查系统对非法侵入的防范能力，其目的是为了发现软件系统中是否存在安全漏洞。软件安全性是指在非正常条件下不发生安全事故的能力。

在安全测试过程中，测试者扮演着一个试图攻击系统的角色。测试者可以尝试通过外部的手段来获取系统的密码，可以使用能够瓦解任何防护的客户软件来攻击系统；可以把系统“制服”，使别人无法访问；可以有目的地引发系统错误，期望在系统恢复过程中侵入系统；可以通过浏览非保密的数据，从中找到进入系统的钥匙等。只要有足够的时间和资源，好的安全测试就一定能够侵入一个系统。

系统安全设计的准则是，使非法侵入的代价超过被保护的信息的价值，从而令非法侵入者无利可图。一般来讲，如果黑客为非法入侵花费的代价(考虑时间、费用、危险等因素)高于得到的好处，那么这样的系统可以认为是安全的系统。

安全性一般分为两个层次，即应用程序级的安全性和系统级别的安全性。它们的关系如下：

- 应用程序级别的安全性包括对数据或业务功能的访问；而系统级别的安全性包括对系统的登录或远程访问。
- 应用程序级别的安全性可确保在预期的安全性情况下，操作者只能访问特定的功能或用例，或者只能访问有限的数据。例如，某财务系统可能会允许所有人输入数据，创建新账户，但只有管理员才能删除这些数据或账户。
- 系统级别的安全性对确保只有具备系统访问权限的用户才能访问应用程序，而且只能通过相应的入口来访问。

9.5.2 安全性测试方法

首先讨论安全性测试方法。

1. 安全测试方法

(1)功能验证。功能验证是采用软件测试当中的黑盒测试方法，对涉及安全的软件功能，如用户管理模块、权限管理模块、加密系统、认证系统等进行测试，主要是验证上述功能是否有效。一些功能性的安全性问题包括：

- 控制特性是否工作正确？
- 无效的或者不可能的参数是否被检测并且适当处理？
- 无效的或者超出范围的指令是否被检测并且适当处理？
- 错误和文件访问是否被适当记录？
- 是否有变更安全性表格的过程？
- 系统配置数据能否正确保存，系统故障时能否恢复？
- 系统配置数据能否导出，在其他机器上进行备份？
- 系统配置数据能否导入，导入后能否正常使用？
- 系统配置数据保存时是否加密？
- 没有口令能否登录到系统中？
- 有效的口令是否被接受，无效的口令是否被拒绝？
- 系统对多次无效口令是否有适当的反应？
- 系统初始的权限功能是否正确？
- 各级用户权限划分是否合理？
- 用户的生命期是否有限制？
- 低级别的用户能否操作高级别用户命令？
- 高级别的用户能否操作低级别用户命令？
- 用户是否会自动超时推出，超时的时间是否设置合理，用户数据是否会丢失？
- 登录用户修改其他用户的参数是否会立即生效？
- 系统在最大用户数量时是否操作正常？
- 对于远端操作是否有安全方面的特性？
- 防火墙能否被激活和取消激活？

- 防火墙功能激活后是否会引起其他问题？

(2)漏洞扫描。安全漏洞扫描通常都借助于特定的漏洞扫描器完成。漏洞扫描器是一种能自动检测远程或本地主机安全性弱点的程序，通过使用漏洞扫描器，系统管理员能够发现所维护信息系统存在的安全漏洞，从而在信息系统网络安全防护过程中做到有的放矢，及时修补漏洞。

按常规标准，可以将漏洞扫描器分为两种类型：主机漏洞扫描器(Host Scanner)和网络漏洞扫描器(Network Scanner)。主机漏洞扫描器是指在系统本地运行检测系统漏洞的程序，如著名的 COPS、Tripewire、Tiger 等自由软件。网络漏洞扫描器是指基于网络远程检测目标网络和主机系统漏洞的程序，如 Satan、ISS Internet Scanner 等。

安全漏洞扫描可以用于日常安全防护，同时可以作为对软件产品或信息系统进行测试的手段，可以在安全漏洞造成严重危害前发现漏洞并加以防范。

(3)模拟攻击试验。对于安全测试来说，模拟攻击试验是一组特殊的黑盒测试案例，通常以模拟攻击来验证软件或信息系统的安全防护能力，下面列举在数据处理与数据通信环境中特别关心的几种攻击方法。

- 冒充：就是一个实体假装成另外一个不同的实体。冒充常与某些主动攻击形式一起使用，特别是消息的重演与篡改。例如，截获鉴别序列，并在一个有效的鉴别序列使用过一次后再次使用。特权很少的实体为了得到额外的特权，可能使用冒充成为具有这些特权的实体，举例如下。
  - ✧ 口令猜测：一旦旦黑客识别了一台主机，而且发现了基于 NetBIOS、Telnet 或 NFS 服务的可利用的用户账号，并成功地猜测出了口令，就能对机器进行控制。
  - ✧ 缓冲区溢出；在服务程序中，如果程序员使用类似于 strcpy()、strcat()等字符串函数，由于这些函数不进行有效位检查的函数，所以可能导致恶意用户编写一小段程序来进一步打开缺口，将该代码放在缓冲区有效载荷末尾。这样，当发生缓冲区溢出时，返回指针指向恶意代码，执行恶意指令，就可以得到系统的控制权。
- 重演：当一个消息或部分消息为了产生非授权效果而被重复时，就出现了重演。例如，一个含有鉴别信息的有效消息可能被另一个实体所重演，目的是鉴别它自己(把它当成其他实体)。
- 消息篡改：数据所传送的内容被改变而未被发觉，并导致非授权后果，有以下两种形式。
  - ✧ DNS 高速缓存污染：由于 DNS 服务器与其他名称服务器交换信息时并不进行身份验证，这就使黑客可以加入不正确的信息，并把用户引向黑客的主机。
  - ✧ 伪造电子邮件：由于 SMTP 并不对邮件发送附件的身份进行鉴定，因此黑客可以对内部客户伪造电子邮件，声称是来自某个客户认识并相信的人，并附上可安装的特洛伊木马程序或者一个指向恶意网站的链接。
- 服务拒绝：当一个实体不能执行它的正常功能，或它的动作妨碍了别的实体执行它们的正常功能的时候，便发生服务拒绝。这种攻击可能是一般性的，比如一个实体抑制所有的消息；也可能是有具体目标的，例如一个实体抑制所有流向某一特定目的端的消息，如安全审计服务。这种攻击可以是对通信业务流的抑制，或产生额外的通信业务流，也可能制造出试图破坏网络操作的消息。特别是如果网络具有中继实体，这些中继实体根据从别的中继实体那里接收到的状态报告，来做出路由选择的决定。拒绝服务攻击种类很多，举例如下。
  - ✧ 死亡之 Ping(Ping of Death)：由于在早期阶段，路由器对包的最大尺寸都有限制，许多操作系统对 TCP/IP 栈的实现在 ICMP 包上都规定为 64KB，并且在读取包的标题头之后，要根据该标题头里包含的信息来为有效载荷生成缓冲区。当产生畸形的、声称

自己的尺寸超过 ICMP 上限，也就是加载尺寸超过 64K 上限的包时，就会出现内存分配错误，导致 TCP/IP 堆栈崩溃，致使接收方宕机。

- ✧ 泪滴(Teardrop)：泪滴攻击利用那些在 TCP/IP 堆栈实现中信任 IP 碎片中的包的标题头所包含的信息来实现自己的攻击。IP 分段含有指示该分段所包含的是原包的哪一段的信息，某些 TCP/IP(包括 Service Pack 4 以前的 NT)在收到含有重叠偏移的伪造分段时将崩溃。
- ✧ UDP 洪水(UDP Flood)：利用简单的 TCP/IP 服务进行各种各样的假冒攻击，如 Chargen 和 Echo 来传送毫无用处的数据以占满带宽。通过伪造与某一主机的 Chargen 服务之间的一次的 UDP 连接，回复地址指向开着 Echo 服务的一台主机，这样就生成在两台主机之间的足够多的无用数据流，如果数据流足够多，就会导致带宽的服务攻击。
- ✧ SYN 洪水(SYN Flood)：一些 TCP/IP 栈的实现，只能等待从有限数量的计算机发来的 ACK 消息，因为它们只有有限的内存缓冲区用于创建连接，如果这一缓冲区充满了虚假连接的初始信息，该服务器就会对接下来的连接请求停止响应，直到缓冲区里的连接企图超时为止。在一些创建连接不受限制的实现中，SYN 洪水也具有类似的影响。
- ✧ Land 攻击：在 Land 攻击中，一个特别打造的 SYN 包的原地址和目标地址都被设置成某一个服务器地址，这将导致接收服务器向它自己的地址发送 SYN-ACK 消息，结果这个地址又发回 ACK 消息并创建一个空连接，每一个这样的连接都将保留，直到超时。各种系统对 Land 攻击的反应不同——许多 UNIX 实现将崩溃；NT 变得极其缓慢(大约持续 5min)。
- ✧ Smurf 攻击：一个简单的 Smurf 攻击，通过使用将回复地址设置成受害网络的广播地址的 ICMP 应答请求(ping)数据包来淹没受害主机的方式进行，最终导致该网络的所有主机都对此 ICMP 应答请求做出答复，导致网络阻塞，比死亡之 Ping 洪水的流量高出一或两个数量级。更加复杂的 Smurf 将源地址改为第三方的受害者，最终导致第三方雪崩。
- ✧ Fraggle 攻击：Fraggle 攻击对 Smurf 攻击做了简单的修改，使用的是 UDP 应答消息，而非 ICMP。
- ✧ 电子邮件炸弹：电子邮件炸弹是最古老的匿名攻击之一，通过设置一台机器，不断大量地向同一地址发送电子邮件，攻击者能够耗尽接收者网络的带宽。
- ✧ 畸形消息攻击：各类操作系统上的许多服务都存在此类问题，由于这些服务在处理信息之前没有进行适当正确的错误校验，在收到畸形的信息时可能会崩溃。

- 内部攻击：当系统的合法用户以非故意或非授权方式进行动作时就成为内部攻击。多数已知的计算机犯罪都和使系统安全遭受损害的内部攻击有密切的关系。能用来防止内部攻击的保护方法包括：对所有管理数据流进行加密；利用包括使用强口令在内的多级控制机制和集中管理机制来加强系统的控制能力：为分布在不同场所的业务部门划分 VLAN，将数据流隔离在特定部门；利用防火墙为进出网络的用户提供认证功能，提供访问控制保护；使用安全日志记录网络管理数据流等。
- 外部攻击：外部攻击可以使用的办法有搭线(主动的与被动的)、截取辐射、冒充为系统的授权用户，冒充为系统的组成部分、为鉴别或访问控制机制设置旁路等。
- 陷阱门：当系统的实体受到改变，致使一个攻击者能对命令或对预定的事件或事件序列产生非授权的影响时，其结果就称为陷阱门。例如，口令的有效性可能被修改，使其除了正常效力之外，也使攻击者的口令生效。
- 特洛伊木马：对系统而言的特洛伊木马，是指它不但具有自己的授权功能，而且还有非授

权功能。一个向非授权信道复制消息的中继就是一个特洛伊木马。典型的特洛伊木马有 NetBus、BackOrifice 和 BO2k 等。

(4) 侦听技术。侦听技术实际上是在数据通信或数据交互中，对数据进行截取分析的过程。目前最为流行的是网络数据包的捕获技术，通常称为 Capture，黑客可以利用该项技术实现数据的盗用，而测试人员同样可利用该项技术实现安全测试。该项技术主要用于对网络加密的验证。

2．确定安全性标准

(1) 安全目标。

- 预防：对有可能被攻击的部分采取必要的保护措施，如密码验证等。
- 跟踪审计：从数据库系统本身、主体和客体三个方面来设置审计选项，审计对数据库对象的访问以及与安全相关的事件。数据库审计员可以分析审计信息、跟踪审计事件、追查责任，并将对系统效率的影响减至最小。
- 监控：能够对针对软件或数据库的实时操作进行监控，并对越权行为或危险行为发出警报信息。
- 保密性和机密性：可防止非授权用户的侵入和机密信息的泄露。
- 多级安全性：指多级安全关系数据库在单一数据库系统中存储和管理不同敏感性的数据，同时通过自主访问控制和强制访问控制机制保持数据的安全性。
- 匿名性：防止匿名登录。
- 数据的完整性：可防止数据在异常情况下保证完整性。

(2) 安全的原则。

- 加固最脆弱的连接：进行风险分析并提交报告，加固其薄弱环节。
- 实行深度防护：利用分散的防护策略来管理风险。
- 失败安全：在系统运行失败时有相应的措施保障软件安全。
- 最小优先权：原则是对于一个操作，只赋予所必需的最小的访问权限，而且只分配所必需的最少时间。
- 分割：将系统尽可能分割成小单元，隔离那些有安全特权的代码，将对系统可能的损害减到最小。
- 简单化：软件设计和实现要尽可能直接，在满足安全需求的前提下构筑尽量简单的系统，关键的安全操作都部署在系统不多的关键点(choke points)上。
- 保密性：避免滥用用户的保密信息。

(3) 缓冲区溢出。防止内部缓冲区溢出的实现、防止输入溢出的实现、防止堆和堆栈溢出的实现。

(4) 密码学的应用。

- 使用密码学的目标：机密性、完整性、可鉴别性、抗抵赖性。
- 密码算法(对称和非对称)：考虑算法的基本功能、强度、弱点及密钥长度的影响。
- 密钥管理的功能：生成、分发、校验、撤销、破坏、存储、恢复、生存期和完整性。
- 密码术编程：加密、散列运算、公钥密码加密、多线程、加密 cookie 私钥算法、公钥算法及 PKI、一次性密码、分组密码等。

(5) 信任管理和输入的有效性。信任的可传递、防止恶意访问、安全调用程序、网页安全、客户端安全、格式串攻击。

(6) 口令认证。口令的存储、添加用户、口令认证、选择口令、数据库安全性、访问控制(使用视图)、保护域、抵抗统计攻击。

(7)客户端安全性。版权保护机制(许可证文件、对不可信客户的身份认证)、防篡改技术(反调试程序、检查和、对滥用的响应)、代码迷惑技术、程序加密技术。

(8)安全控制/构架。过程隔离、权利分离、可审计性、数据隐藏、安全内核。

3．安全性评价模型

本节主要介绍两种软件安全性评价模型：贝叶斯模型和 3M 评价模型。

(1)贝叶斯模型。设 D 为软件的输入空间，其运行剖面为{〈p_i，i〉，i∈D}，$\sum p_i=1$，测试剖面为{〈d_i，i〉，i∈D}，$\sum d_i=1$。其中 $d_i=z_i \cdot p_i$，z_i 为根据失效危害严重度将测试输入出现概率放大或缩小的调整因子。对于划分为相同失效危害严重度的输入取相同的调整因子，也即将 D 划分成{CⅠ，CⅡ，CⅢ，CⅣ}，对于每一输入子空间 C_j，j∈{Ⅰ，Ⅱ，Ⅲ，Ⅳ}，取同一调整因子，令其分别为 ZⅠ、ZⅡ、Z Ⅲ和 ZⅣ。又设总测试次数为 n，其中 C_j 进行了 n_j 次测试，且有 x_j 次失效，θ_j 为第 j 级错误的失效率，其运行剖面下的验前分布是参数为 a_j 和 b_j 的 Beta 分布。

根据贝叶斯推断，θ_j 的验后分布为：

$$\frac{a_j + x_j}{nZ_j + a_j + b_j}$$

(2)3M 评价模型。在 3M 评价法中，用软件中残留致险缺陷数的估计作为安全性评价的指标，它所依据的信息包括：被评估软件所控制对象的功能复杂性对安全性的影响 C，用复杂度因子 A_C 来反映这一方面的信息；该安全苛求系统所拥有的技术支持 S，包括宿主系统的质量(硬件和系统软件可靠性、容错结构)、软件开发工具对安全性的影响等，这方面的信息用技术支持因子 A_S 来衡量；软件开发队伍的技术素质和水平对安全性的影响 L，用开发水平因子 A_L 来衡量。因此一个软件产品在系统测试之前的内在缺陷数可估计为 $f(A_C, A_S, A_L)$，它是 A_C 的单调递增函数，A_S，A_L 的单调递减函数。不失一般性，可将测试前致险缺陷估计值用 $D_\lambda = K\frac{A_C^\alpha}{A_S^\beta \times A_L^\gamma}$ 来表达，其中 α、β、γ 为正实数，分别描述了各因子对残留致险缺陷的影响规律，K 也是正实数，为描述的各因子对残留缺陷数的比例值。对于一特定系统，α、β、γ、K 为常数，其值可通过数据拟合取得，如果 α=l，则表明 D_P 和 A_C 成正比，如果 0<α<l，则 A_C^α 是一个凸函数，如果 α>l，则 A_C^α 是一个凹函数，A_S^β 和 A_L^γ 随 β、γ 的变化情况同样如此。

4．安全性测试执行

安全测试步骤如下：

- 危险和威胁分析。执行系统和它的实用环境的风险和威胁分析。
- 以一种它们可以和系统的安全性动作相比较的方式来定义安全性需求和划分优先级。基于威胁分析，为系统定义安全需求，最关键的安全性需求应该得到最大程度的关注。注意，系统最弱的链接也是重要的，安全性需求的定义是一个反复的过程。
- 模拟安全行为。基于划分的安全需求的优先次序，识别形成系统安全动作的功能和它们依赖的优先顺序。
- 执行安全性测试。实用合适的证据收集和测试工具。
- 估计基于证据的安全活动的可能性和影响。合计出一个准确的结果及系统是否满足安全性需求。

执行风险分析和构建安全需求是安全测试活动的一个整体的部分。没有合适的需求，将很难安排测试计划和达到有意义的结果。

9.5.3　一个安全性测试案例分析

以 IPv6 防火墙安全性测试进行案例分析。

(1) 风险和威胁分析。防火墙代表了最重要的网络安全机制，通常在本地网络和因特网(Internet)之间扮演着网络交通过滤器的角。

因为有很多的软件防火墙(有免费软件和商业软件)，可以为用户提供能容易定义过滤规则的友好的图形界面，大多数防火墙已经为频繁使用的应用程序(Web 浏览器，E-mail 客户端等)预先定义了过滤规则集，但是用户可以更改存在的规则，并根据他们的需要添加新的规则。由于 IPv4 和 IPv6 通信的规则必须单独定义，所以欲在 IPv6 中使用的防火墙必须建立支持 IPv6 协议。IPv6 协议引入一个新的包头形式(和 IPv4 头不同)，必须被 IPv6 防火墙适当地验证和处理。其他和 IPv6 有关的协议，像 ICMPv6 协议，也必须被 IPv6 防火墙适当地支持。IPv4 和 IPv6 协议在包过滤可能性上有一些不同，为了配置 IPv6 防火墙有一种工具称为“ip6tables”，在现在的 Linux 分区中都有，而且它和“iptables”工具很相似。在 MS Windows 平台上有一个 Windows 防火墙支持 IPv6 协议。

(2) 测试环境。本案例中以实验为目的搭建了一个小的 IPv6 网络，该网络由三台计算机、两台 PC(Intel 赛扬和 Intel 奔 4 的 CPU)和一台笔记本(Intel 赛扬 CPU)组成。所有的计算机都被配置成有 MS Windows XP 和 Mandrake Linux 10 的双操作系统。此外，在这个实验的网络上，所有的计算机都被配置了双堆栈的设备支持 IPv4 和 IPv6。一个本地 IPv6 网络链接到 CAR6Net 网络。

所有在 IPv6 网络的安全性方面的测试都在实验网络上执行。在这个测试环境下，在两种系统上做了防火墙的不同的测试，还做了不同类型的侦查攻击，分析了一些可能成功的检测。所有的安全测试都是在两种系统上进行的。

(3) IPv6 防火墙测试。为了设置防火墙过滤规则命令行应用，使用了 Netshell(在 Windows XP 平台)和 ip6tables(Linux 平台)。在 Windows XP 和 Linux 上使用同样的防火墙是为了更好地比较。为了测试目的(扫描安全攻击)使用了 Nmap 应用程序。Nmap 应用程序的官方的版本支持 IPv6 协议，但是有一个基于老的官方版本可以更好地支持 IPv6 协议，在这里选择了这个合适的版本。这个 Nmap 版本支持更多的扫描技术，比如 TCP 连接扫描、SYN 扫描、ACK 扫描、FIN 扫描、Xmas 树扫描和 UDP 扫描。TCP 连接扫描技术是 Nmap 应用程序的默认扫描类型，它意味着试图在目标主机的不同端口上建立一个 TCP 连接。如果目标端口在监听，连接将会被建立，否则这个端口将是不可到达的。这种类型的扫描使用一个 Connect 系统调用(Web 浏览器和其他的激活网络是为了建立连接使用相同的高级系统调用)。由于它不使用原始笔迹网络包，故这个扫描方法可以被任何人使用。通过这种扫描方法建立到监听目标端口的完全的 TCP 连接，然后不发送数据就关闭。正是因为这样，TCP 连接扫描方法很容易被目标主机的 IDE 系统检测到。SYN 扫描技术经常被作为部分开放扫描涉及，因为它并不建立一个完整的 TCP 连接。通过 SYN 扫描技术，一个 SYN 包被发送到目标主机，和建立一个完全的 TCP 连接过程相似。一个 SYN/ACK 响应显示目标主机正在监听，RST 响应显示一个没有在监听的端口。这种扫描方法是秘密相关的，因为它从不建立一个完全的 TCP 连接，也就是说它能够很容易地避免在目标网络中的准许进入过滤检查和很难被 IDS 系统检测到。ACK 扫描方法发送一个 ACK 包(仅仅包含 SACK 标记设置的探测包)随机地查看到指定的端口的确认/顺序号。未被过滤得端口将发送一个 RST 响应给 ACK 探测包。Xmas 树扫描方法在探测包中设置标记 FIN、URG 和 PUSH。一个关闭的端口应该用 RST 包答复，而一个开着的端口忽略这个探测包。FIN 扫描技术除了在探测包中只设置 FIN 标记外，和 Xmas 树扫描方法完全一样。通过 UDP 扫描方法 0 字节的 UDP 包(仅仅 UDP 报头)被发送到目标端口。在那种

情况下，几个接收到的 ICMP 端口不可到达的消息表示一个关闭的端口。所有这些描述的扫描方法在用于实验的 IPv6 网络上都执行了。

在 Linux 平台上，任何扫描方法都不能通过防火墙，不能在目标主机上发现该端口设置。

在 Linux 平台上的扫描结果如图 9-8 所示。

```
TCP connect scan:
Base-2,OSb nmap -6 -sT -p0 2001:b68:8001:: 4
Starting nmap V.2.S4BCTA36(www.insecure.org/nmap/)
ALL 1Ss scanned ports on 2001:b68:s001::4(2001::b6s:s001::4)are:filtered
Namp run completed --1 IP address(1 host up) scanned in 1725 seconds
TCP SYN scan:
Bash-2.05b nmap -6 -sS -p0 2001:b68:8001:: 4
Starting nmap V.2.54BLTA36(www.insecure.org/nmap/)
ALL 1SSs scanned ports on 2001:b68:s001:: 4(2001::b6s:s001::4)are:filtered
Namp run completed --1 IP address(1 host up) scanned in 168s seconds
ACK scan:
Bash-2.05b nmap -6 -sA -p0 2001:b68:8001:: 4
Starting nmap V.2.54BTA36(www.insecure.org/nmap/)
ALL 1Ss8 scanned ports on 2001:b68:8001:: 4(2001:b68:8001::4)are:filtered
Namp run completed --1 IP address(1 host up) scanned in 1687 seconds
FTN scan:
bash-2.05# nmap -6 -sF -p0 2001:b68:8001:: 4
Starting nmap V.2.54BETA36 (www.insecure.org/nmap/)
ALL 1SS8 scanned ports on 2001:b68:8001:: 4(2001:: b68:8001:: 4)are: filtered
Namp run completed --1 IP address (1 host up) scanned in 187S seconds
Xmas Tree scans:
bash-2.05# nmap -6 -sx -p0 2001:b68:8001:: 4
Starting nmap V.2.54BETA36 (www.insecure.org/nmap/)
ALL 1SS8 scanned ports on 2001:b68:8001:: 4(2001:: b68:8001::4)are:filtered
Namp run completed --1 IP address (1 host up) scanned in 187S seconds
UDP scan:
bash-2.05# nmap -6 -sv -p0 2001:b68:8001::4
Starting nmap V.2.54BETA36(www.insecure.org/nmap/)
ALL 1460 scanned ports on 2001:b68:8001::4(2001::b68:8001::4)are:filtered
Namp run completed --1 IP address(1 host up) scanned in 17S8 seconds
```

图 9-8　在 Linux 平台上的扫描结果

在 Windows XP 平台上一些扫描技术(TCP 连接和 SYN 扫描)成功地通过防火墙发现了目标主机的端口设置。

在 MS Windows 平台上的扫描结果如图 9-9 所示。

```
bash-2.05b # nmap -6 -ST -p0 2001:b68:8001::3

Starting nmap V.2.54BETA36(www.insecure.org/nmap/)

Found route through interface:eth0
Interesting ports on 2001;b68;8001;;3(2001:b68:8001::3)
(The 1554 ports scanned but not shown below are in state: filtered)
Port      State      Service
22/tcp    closed      ssh
23/tcp    closed      telnet
25/tcp    closed      smtp
80/tcp    closed      http
```

图 9-9　在 MS Windows 平台上的扫描结果

因此，目前 Linux 防火墙比 Windows 防火墙提供了更高的安全级别。对 IPv6 网络防火墙的测试表明，Linux 防火墙比 Windows 防火墙更安全，IPv4 网络的情况与 IPv6 相似，也是 Linux 防

火墙比 Windows 防火墙提供了更高的安全级别。因此，在一个要求更高安全级别的网络中建议使用 Linux 防火墙。

9.6　可靠性测试

本节讨论可靠性测试概念及其方法。

9.6.1　可靠性测试基本概念

软件可靠性是指在规定时间周期内，在一定条件下，程序执行所要求的功能的能力。软件可靠性测试包括软件可靠性增长测试和软件可靠性验证测试。软件可靠性增长测试是为了满足用户对软件的可靠性要求，通过对软件进行测试，发现并纠正软件中的缺陷，提高软件可靠性水平的一种软件测试方法。这种测试是为了满足软件的可靠性指标，不断发现程序中的错误而执行程序的过程，是一种软件测试—修改—测试的动态方法，是一个闭环过程。软件可靠性验证测试是为了验证在给定的统计置信度下，软件当时的可靠性水平是否满足用户要求而进行的测试，即用户在接收软件时，确定它是否满足软件规格说明书中规定的可靠性指标。

软件可靠性测试不同于硬件可靠性测试。它强调按实际使用的概率分布随机选择输入，并强调测试需求的覆盖度。软件可靠性测试也不同于一般的软件功能测试，它更强调与典型使用环境输入统计特性的一致，要求准确记录软件运行时间，输入覆盖也要大于普通软件功能测试的要求，测试输入应对各种使用功能考虑重要输入变量值、相关输入变量的可能组合以及不合法输入域的覆盖，对使用环境的覆盖比一般的软件测试要求更高。

9.6.2　可靠性测试方法

在软件测试过程中，根据测试对象不同，选用的测试方法也不同。结合软件测试方法，可以采取静态黑盒测试和动态黑盒测试法。所谓的静态黑盒，是通过阅读软件的需求说明书及各阶段的文档验证可靠性设计的正确性；而动态黑盒测试，是通过执行系统各项功能来检验软件可靠性。

通常，软件可靠性有自己的行业标准。常见的标准有：

- ANSI/AIAA R-013。
- MIL-STD-882C。
- RTCA DO-178B。
- IEEE-Std-730。
- MIL-HDBK 338,Electronic Reliability Design Handbook。
- MIL-HDBK-217,Reliability Prediction of Electronic Equipment。
- MIL-HDBK-781 “Reliability Test Methods, Plans and Environments…”。
- IEEE-Guide-982.1 and .2。
- NHB 1700.1Preliminary Hazards Analysis。
- NASA STD 8719.13A Software Safety。
- NASA GB 1740.13 Guidebook for Safety Critical Software。

9.6.3　可靠性评价模型

软件可靠性模型用于在开发早期预计软件的可靠性，它主要关注于缺陷的去除情况。软件可靠性模型包括 3 个部分：假设、可靠性因子以及把可靠性和这些因子相关联的数学函数。通过软

件可靠性模型，可以在项目的测试阶段评价开发的状态，监视软件的操作性能，控制设计的变更和新特性的引入。

下面介绍一些比较流行的可靠性模型，主要有 Jelinski-Moranda 的故障分离模型(Deeutrophication Model)、Goel-Okumoto 的 NHPP 模型(Non-Homogeneous Poisson Process Model)、增强的 NHPP 模型(the Enhanced NHPP Model)以及 Littlewood-Verrall 的贝叶斯判定模型(Bayesian Model)。对于相同的数据，不同的模型可以得到不同的结果，有些结果可能大相径庭，这往往是因为不同的模型所基于的假设条件的不同造成的。

一些常见的数学术语如表 9-2 所示。

表 9-2 一些常见的数学术语

术语	解释
M(t)	到时间 t 发现的缺陷总数
μ(t)	软件可靠性增长模型(SRGM：Software ReliabilityReliability Growth Model)的平均值函数，它表示该模型估计的到时间 t 期望的缺陷的数量。因此有 $\mu(t_0 = E[M(t)])$
λ(t)	缺陷强度，由平均值函数派生出来的。因此有 $\lambda(t) = \mu'(t)$
$Z(\Delta t/t_{i-1})$	软件的机会概率，它表示第 i 个错误在给定的 $t_{i-1}+\Delta t$ 时间内出现的可能密度，其中第 i–1 个错误出现在 t_{i-1}
z(t)	每个故障的机会概率，它表示了一个还没有被激活的故障在其被激活时会立刻引起一个失效的概率。这个术语在很多模型中经常被假设为一个常量(φ)
N	软件在提交测试之前出现的最初缺陷数量

1．故障分离模型

Jelinski-Moranda 的故障分离模型(J-M 模型)是最早开发出来的可靠性模型之一。这个模型假设：

(1)在代码提交测试之前，代码中的原始故障是个固定值 N，但不知 N 是多少。

(2)失效之间没有关系，并且失效之间的时间是独立的且呈指数分布的随机变量。

(3)失效产生后，故障的去除是及时的且不会引入任何新的故障到被测系统中。

(4)每个故障的机会概率 z(t)在时间上是不变的一个常量(φ)。此外，每个故障在引起失效方面是等效的。

这个假设导致软件在第 i–1 个缺陷被去除之后的机会概率是和软件中遗留缺陷数量成比例关系。因此可以得到如下公式：

$$Z(\Delta t / t_{i-1}) = \varphi(N - M(t_{i-1}))$$

这个模型的平均值函数和故障强度函数就变成：

$$\mu(t) = N(1 - e^{-\varphi t})$$

$$\lambda(t) = N\varphi e^{-\varphi t} = \varphi(N - \mu(t))$$

从这个模型中得到的软件可靠性可以表示成如下公式：

$$R(t_i) = e^{-\varphi(N-(i-1)t_i)}$$

这个模型需要两个故障之间的时间值。

2．NHPP 模型

Goel-Okumoto 的 NHPP 模型(G-O 模型)与 J-M 模型的假设略有不同。两个假设之间的最大不同是在时间 t 观察到的期望的故障数量遵循柏松(Poisson)分布，有一个带边界且非递减的均值函数 l(t)。在无限时间内观察到的故障的一个期望值是一个有限的值 N。

这个模型还做了下面的假设：

(1) 出现在(t，t+Δt)的软件失效数量和期望的未检测到的失效数量 N-l(t)成正比。

(2) 在内部失效间隔(0，t_1)(t_1,t_2)……(t_{n-1},t_n)内检测到的失效数量之间没有联系。

(3) 每个故障的机会概率 Z(t)在时间上是不变的一个常量(φ)。

(4) 当故障检测到时，缺陷的排除是及时且完好的(不会引入新错误)。

假设导致了下面的均值函数用于表示在时间 t 被观察到的期望的失效数：

$$\mu(t) = N(1 - e^{-\varphi t})$$

$$\lambda(t) = N\varphi e^{-\varphi t} = \varphi(N - \mu(t))$$

该模型的可靠性可以使用与 J-M 模型一样的公式，因此可以用上面两个公式替换 J-M 模型中相应的公式。

3．增强的 NHPP 模型

增强的 NHPP 模型是对于有限失效 NHPP 模型的一个统一框架，即其他有边界均值函数的 NHPP 是增强 NHPP(ENHPP)模型的一个特例。该模型在它的分析公式中明确地包含了随时间变化的测试覆盖率和不完整的故障检测。

这个模型中的测试覆盖率被定义成对一个测试敏感的潜在故障场所占总潜在故障场所数的比例。潜在故障场所指的是“程序结构或功能元素实体，这个实体的易感染性被认为对建立软件产品运算一致性很关键”。

这个模型做出下列假设：

(1) 故障被一致地分布在所有潜在故障场所中。

(2) 当一个故障场所在时间 t 变得敏感时，其故障被检测的可能性是 $c_d(t) = K$（一个常量），表示故障检测覆盖率。

(3) 故障的排除是及时且完好的(不会引入新错误)。

这个模型的均值函数是：$\mu(t) = c(t)N$

其中 c(t) 是一个随时间变化的测试覆盖率函数，N 是在全覆盖率中期望被暴露的故障数量。这个模型的失效强度公式如下：

$$z(t) = c'(t)(1 - c(t))^{-1}$$

$$\lambda(t) = z(t)(N - \mu(t))$$

其中，z(t) 是一个随时间变化的每个故障的机会概率。这个模型允许有缺陷的覆盖率想法被包含到可靠性估计当中。不同的覆盖率函数分布导致不同的 NHPP 模型，即 G-O 模型、Yamada S-shaped 模型等。这个模型获得的可靠性可以表示成如下公式：

$$R(t/s) = e^{-NK(c(s+t)-c(s))}$$

其中，s 是最后一个失效的时间，t 是上一个失效被检测到的时间。这个模型的主要优点是，可以作为 NHPP 模型的一个统一的框架。此外，每个故障机会概率的依赖性唯一依赖于随时间变化的测试覆盖率，而忽略其他影响因子。

4．贝叶斯判定模型

贝叶斯软件可靠性增长模型认为，可靠性在已被检测的故障数量和无故障操作的上下文之内增长。此外，在没有失效数据情况下，贝叶斯模型认为模型参数有一个优先的分布，它反映了对基于历史的未知数据的判断。

Littlewood-Verrall 模型是贝叶斯判定模型的一个例子，它假定失效之间的时间是独立指数随机变量，拥有一个参数 ε_i, i=1,2,…,n，该参数本身还包含参数 ψ(i)和 α，分别表示程序员质量和任务难度。该模型有一个优先的伽马分布。从该模型观察到的失效强度公式如下[使用线性形式替换了 ψ(i)]：

$$\lambda(t) = (\alpha - 1)(N^2 + 2B\varphi(\alpha - 1))^{-1/2}$$

其中，B 表示故障缩减因子，类似于 Musa 的基本执行时间模型。这个模型需要在时效产生之间进行调整，以便从先前分布中获得较后的分布。

9.6.4 可靠性测试执行

软件可靠性测试的一般过程如图 9.10 所示。主要活动包括构造运行剖面、生成测试用例、准备测试环境、测试运行、收集数据、可靠性数据分析和失效纠正。

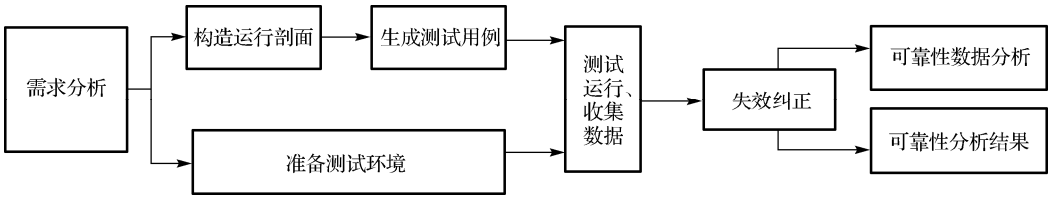


图 9-10 软件可靠性测试的一般过程

(1)构造运行剖面：软件的运行剖面是指对系统使用条件的定义，即系统的输入值用其按时间的分布或按它们在可能输入范围内的出现概率的分布来定义。粗略地说，运行剖面是用来描述软件的实际使用情况的。运行剖面能否代表并刻画软件的实际使用，取决于可靠性工程人员对软件的系统模式、功能、任务需求及相应的输入机理的分析，取决于他们对用户使用这些系统模式、功能、任务的概率的了解。运行剖面构造的质量将对测试、分析的结果是否可信产生最直接的影响。

(2)生成测试用例：软件可靠性测试采用的是按照运行剖面对软件进行可靠性测试的方法。因此，可靠性测试所用的测试用例是根据运行剖面随机选取得到的。

(3)准备测试环境：为了得到尽可能真实的可靠性测试结果，可靠性测试应尽量在真实的环境下进行。但是在许多情况下，在真实的环境下进行软件的可靠性测试很不实际，因此需要开发软件可靠性仿真测试环境。比如，对于多数嵌入式软件，由于与之交联的环境的开发常常与软件的开发是同步甚至是滞后的，因此无法及时进行软件可靠性测试；在有些系统中，由于交联的环境非常昂贵而无法用于需要进行大量运行的可靠性测试。

(4)测试运行：即在真实的测试环境中或可靠性仿真测试环境中，用按照运行剖面生成的测试用例对软件进行测试。

(5)收集数据：收集的数据包括软件的输入数据、输出结果，以便进行失效分析和进行回归测试；软件运行时间数据，可以是 CPU 执行时间、日历时间、时钟时间等；可靠性失效数据包括每次失效发生的时间或一段时间内发生的失效数，失效数据可以通过实时分析得到，也可以通过事后分析得到。数据收集的质量对于最终的可靠性分析结果有着很大的影响，应尽可能采用自动化手段进行数据的收集，以提高效率、准确性和完整性。

(6)可靠性数据分析：主要包括失效分析和可靠性分析。失效分析是根据运行结果判断软件是否失效，以及失效的后果、原因等；而可靠性分析主要是指根据失效数据，估计软件的可靠性水平，预计可能达到的水平，评价产品是否已经达到要求的可靠性水平，为管理决策提供依据。

(7)失效纠正：如果软件的运行结果与需求不一致，则称软件发生失效。通过失效分析，找到并纠正引起失效的程序中的缺陷，从而实现软件可靠性的增长。

9.6.5　一个可靠性测试案例分析

某故障诊断专家系统软件可靠性测试："××飞机起落架故障诊断专家系统"软件是一个使用专家系统方法对××飞机起落架进行故障诊断的软件。该软件是用 VC++开发的，源代码近 1 万条。在该软件调试后期对其进行了软件可靠性测试，主要工作包括运行剖面的构造及测试用例的生成、测试运行及数据收集、可靠性数据分析。

(1)运行剖面的构造及测试用例的生成。根据被测软件功能的说明，结合软件的有关文档，以及对相关概率的估计，可构造软件的系统模式剖面、功能剖面和运行剖面。

运行剖面的构造是一个自顶向下的层次结构。通过不断细化被测软件的输入空间，即从划分系统模式剖面到功能剖面，直到各功能输入变量的取值区间在概率空间的划分，形成最终的运行剖面。这里，一个运行的规定如下：运行是由完成某一功能的一系列输入变量的某一取值区间的有序组合。

各运行在概率空间的划分构成运行剖面。测试用例是根据运行剖面生成的，完成对某一功能的测试，按顺序输入到被测软件的一系列输入变量值的有序组合。

由于运行剖面描述了完成某一功能输入变量的取值区间，通过两次随机抽样，可以得到一个测试用例。第一次抽样选择运行，第二次抽样在每一个输入变量取值区间内随机抽取输入变量的具体取值。将其按照测试过程中的输入顺序组合起来形成测试用例。一个测试用例的具体形式如下：

测试用例序号	输入变量名称	输入变量的具体取值
1	var_1	$nenu_{1.1}$
…	…	…

(2)测试运行及数据收集。按照上述方法生成了 400 个测试用例。在一台配置为 Pentium 586 at 133MHz、内存为 16MB、操作系统为 Windows 95 中文版环境的计算机上，通过手动方式将测试用例输入到被测软件，利用一个为配合这种软件可靠性测试方法而开发的数据辅助收集软件，采集测试运行的时间与失效信息，包括测试用例序号、测试日期、测试开始时刻、测试结束时刻/失效发生时刻、测试运行时间、累计运行时间、失效现象等。通过测试，共记录了 60 次失效，收集到的失效数据如表 9-3 所示，数据从左至右，从上至下，为每次失效发生的累计运行时间(执行时间，单位：s)。

表 9-3　某软件可靠性测试的失效数据

故障时刻	故障时刻	故障时刻	故障时刻	故障时刻
897.8	977.4	1419.9	1539.0	1615.8
1711.1	1923.3	1997.0	2070.3	2708.2
3067.9	3460.5	4078.2	4471.8	5695.8
5807.8	6120.8	6444.8	6704.1	6879.7
7002.5	8688.7	8947.7	9134.9	9687.3
10229.2	10954.5	11084.0	11387.5	12089.7
12329.4	12951.9	13084.6	13608.8	13707.2
13975.7	14068.3	14451.1	14552.2	14634.0
14800.3	15398.8	15455.5	15694.3	15804.5
15897.5	15971.8	16090.6	16618.0	17820.9
18622.4	18731.6	18969.8	20359.5	20510.6
21116.0	21248.0	21320.6	21655.8	21783.7

(3)可靠性数据分析。如表 9-3 所示的失效数据是一组完全失效数据，首先用自行开发的软件可靠性分析工具(SRAT)对数据进行了趋势分析。分析表明，软件的可靠性呈现稳定趋势，即软件具有不变的失效率，失效时间服从指数分布。因此，可计算失效率和平均失效间隔时间分别为：

$$\lambda=\text{总失效数}/\text{总运行时间}=60/21783.7=0.002\ 75(\text{失效数/s})$$

$$MTBF=1/\lambda=363s$$

事实上，在测试中，每次失效发生后，并没有对软件进行失效纠正，因此失效率应该是不变的，数据分析结果也验证了这一点。分析结果表明，该软件的可靠性尚需进一步提高。事实上，在上述失效数据中，许多失效是由相同的缺陷造成的。如果对相同的失效只考虑首次发生的失效，即首次发现就加以纠正的话，软件的可靠性将得到很大的提高。需要强调的是，该分析结果是在给定的运行剖面下、在给定的运行环境下进行测试得到的分析结果。不同的运行剖面，不同的运行环境(如不同的机器速度)会得到不同的可靠性估计。另外，所收集的失效时间数据的类型也会影响数据分析的结果。

9.7 恢复性测试与备份测试

恢复性测试主要检查系统的容错能力，即当系统出错时，能否在指定时间间隔内修正错误并重新启动系统。恢复性测试首先要采用各种办法强迫系统失败，然后验证系统能否尽快恢复。对于自动恢复，需验证重新初始化、检查点、数据恢复和重新启动等机制的正确性；对于人工干预的恢复系统，还需估测平均修复时间，确定其是否在可接受的范围内。

备份测试是恢复性测试的一个补充，也是恢复性测试的一个部分。备份测试的目的是，验证系统在软件或者硬件失败时备份数据的能力。

在设计恢复性测试用例时，需要考虑下列关键问题。

(1)测试是否存在潜在的灾难，以及它们可能造成的损失？消防训练式的布置灾难场景是一种有效的方法。

(2)保护和恢复工作是否为灾难做了足够的准备？评审人员应该评审测试工作及测试步骤，以便检查对灾难的准备情况。评审人员包括主要事件专家和系统用户。

(3)当真正需要时，恢复过程是否能够正常工作？模拟的灾难需要和实际的系统一起被创建以验证恢复过程。用户、供应商应当共同完成测试工作。

备份测试需要从以下几个角度来进行设计：

- 备份文件，并且比较备份文件与最初的文件的区别。
- 存储文件和数据。
- 完善系统备份工作的步骤。
- 检查点数据备份。
- 备份引起系统性能衰减程度。
- 手工备份的有效性。
- 系统备份“触发器”的检测。
- 备份期间的安全性。
- 备份过程日志。

9.8　兼容性测试

兼容性测试是指检查软件之间能否正确地交互和共享信息。软件兼容性测试需要解决以下问题：

- 软件设计需求与运行平台的兼容性。如果被测软件本身是一个支持平台，那么还要设计一个在该平台上运行的应用程序。
- 软件的行业标准或规范，以及如何达到这些标准和规范的条件。
- 被测软件与其他平台、其他软件交互或共享的信息。

上述问题是兼容性测试用例设计的依据，在具体应用时还应该考虑被测软件的具体情况。例如，Microsoft Windows 认证软件要求如下。

- 支持三键以上的鼠标。
- 支持在 C:和 D:以外的磁盘上安装。
- 支持超过 DOS 8.3 格式文件名长度的文件名。
- 不能读、写，或者以其他形式使用旧系统中的 win.ini、system.ini、autoexec.bat 和 config.sys 文件。

例：浏览器测试。浏览器是 Web 客户端最核心的构件，来自不同厂商的浏览器对 Java、JavaScript、ActiveX、plug-ins 或不同的 HTML 规格有不同的支持。例如，ActiveX 是 Microsoft 为 Internet Explorer 而设计的产品，JavaScript 是 Sun 为 Netscape 设计的 Java 产品。浏览器的测试就是要测试 Netscape 对 ActiveX，以及 Explorer 对 JavaScript 兼容性的测试。通常可以用一个兼容性测试矩阵(如表 9-4 所示)，即被测软件与被测环境(平台)的二维方阵，来设计测试用例。

表 9-4　兼容性测试矩阵

	Explorer 浏览器	Netscape 浏览器
ActiveX	√	√
JavaScript	√	√

9.9　安装性测试

软件运行的第一件事就是安装(嵌入式软件除外)，所以安装性测试是软件测试首先需要解决的问题。安装性测试看上去简单，但实际上并非如此。安装性测试不仅要考虑在不同的操作系统上运行，而且还要考虑与现有软件系统的配合使用问题。因此，安装性测试应考虑多个方面的内容，可以从以下几个方面考虑：

- 应参照安装手册中的步骤进行安装，主要考虑到安装过程中所有的默认选项和典型选项的验证。安装前，应先备份测试机的注册表。
- 安装有自动安装和手工配置之分，应测试不同的安装组合的正确性，最终使所有组合均能安装成功。
- 对安装过程中异常配置或状态情况(继电等)要进行测试。
- 检查安装后能否产生正确或多余的目录结构和文件，以及文件属性是否正确。
- 安装性测试应该在所有的运行环境上进行验证，如操作系统、数据库、硬件环境、网络环境等。
- 至少要在一台笔记本上进行安装性测试，台式机和笔记本硬件的差别会造成其安装时出现问题。
- 安装后应执行卸载操作，检测系统能否正确完成任务。

- 检测安装该程序是否对其他的应用程序造成影响。
- 如有 Web 服务，应检测会不会引起多个 Web 服务的冲突。

9.10 可用性测试

以下从基本概念和测试方法两个方面介绍可用性测试。

9.10.1 可用性测试的概念

可用性测试(Usability Testing)是对于用户友好性的测试，是指在设计过程中被用来改善易用性的一系列方法。测试人员为用户提供一系列操作场景和任务让他们去完成，这些场景和任务与产品或服务密切相关，通过观察来发现完成过程中出现了什么问题、用户喜欢或不喜欢哪些功能和操作方式，原因是什么，针对问题所在提出改进的建议。

可用性区别于实用性。可用性是指产品在特定使用环境下为特定用户用于特定用途时所具有的有效性、效率和用户主观满意度。有效性是指用户完成特定任务时所具有的正确和完整程度；效率是指用户完成任务的正确完整程度与所用资源(如时间)之间的比率；满意度是指用户在使用产品过程中具有的主观满意和接受程度。可用性体现的是用户在使用过程中所实际感受到的产品质量，即使用质量；而实用性体现的是产品功能，即产品本身所具有的功能模块。与实用性相比，可用性重视了人的因素，重视了产品是被最终用户使用的。

可用性测试的价值在于能够及早发现产品或服务中将会出现的存在于用户使用过程中的问题，从而在产品开发或正式投产之前给出改进建议，以较小的投入帮助开发人员全面改善产品，节约开发成本。典型的可用性测试会包含以下维度：

- 任务操作的成功率。
- 任务操作效率。
- 任务操作前的用户期待。
- 任务操作后的用户评价。
- 用户满意度。
- 各任务出错率。
- 二次操作成功率。
- 二次识别率。

可用性测试的文档主要包括：

- 日程安排文档。
- 用户背景资料文档。
- 用户协议。
- 测试脚本。
- 测试前问卷。
- 测试后问卷。
- 任务卡片。
- 测试过程检查文档。
- 过程记录文档。
- 测试报告。
- 影音资料。

9.10.2　可用性测试方法

(1)一对一用户测试：一个可用性测试部分包括测试人员(主持人/助理)和一个目标用户，这个目标用户会在测试人员的陪同下完成一系列的典型任务。在征得参与者的同意后，测试过程将被摄像，测试人员将持续观察、了解用户的操作过程、思维过程以及相关各项指标(包括用户出错次数、完成任务的时间等)，记录用户遇到的可用性问题并分析。

(2)启发式评估：邀请 5～8 名用户作为评估人员来评价产品使用中的人机交互状况，发现问题，并根据可用性设计原则提出改进方案。

启发式评估法是一种用来发现用户界面设计中的可用性问题从而使这些问题作为再设计过程中的一部分被重视的内容的可用性检查方法。启发式评估法旨在利用已确立的可用性原则来解释每个发现的可用性问题，所以要根据由已经被违背的、好的交互系统需具备的原则所规定的设计准则来制定一个修正的设计方案是相当容易的。据研究发现，一般情况下，5 个评估人员能够发现 75%的可用性问题，从可用性问题产出的市场价值与评估费用的比率来看，是较为理想的数字。

(3)焦点小组：是在可用性工程中使用比较多的一种方法，通常用于产品功能的界定、工作流程的模拟、用户需求的发现、用户界面的结构设计和交互设计、产品的原型的接受度测试、用户模型的建立等。焦点小组是依据群体动力学原理、由 6～12 个参试人组成的富有创造力的小群体，在一名专业主持人的引导下对某一主题或观念进行深入讨论，主持人要在不限制用户自由发表观点和评论的前提下，保持谈论的内容不偏离主题。同时，主持人还要让每个参试人都积极地参与讨论，避免部分用户主导讨论，部分消极用户较少地参与讨论。焦点小组实施之前，通常需要列出一张清单，包括要讨论的问题及各类数据收集目标。小组借由参与者之间的互动来激发想法和思考，从而使讨论更加深入、完整。

典型的可用性测试通常需要 2 周的前期沟通和准备、1 周的测试、2 周的提交分析报告时间。可根据测试的内容及项目规模做具体调整。

一些测试人员应当关注的可用性问题包括：

- 过分复杂的功能或者指令。
- 困难的安装过程。
- 错误信息过于简单，例如“系统错误”。
- 语法难于理解和使用。
- 非标准的 GUI 接口。
- 用户被迫记住太多的信息。
- 难以登录。
- 帮助文本上下文不敏感或者不够详细。
- 和其他系统之间的连接太弱。
- 默认不够清晰。
- 接口太简单或者太复杂。
- 语法、格式和定义不一致。
- 输入不够全面。

9.11　配置性测试

本节介绍配置性测试的基本概念和基本方法。

9.11.1 配置性测试的概念

配置性测试(Configuration Testing)验证系统在不同的系统配置下能否正确工作，这些配置包括软件、硬件、网络等。如果开始准备进行软件的配置测试，就要考虑哪些配置与程序的关系最密切。通常认为的理想状况是所有生产厂家都严格遵照一套标准来设计硬件，那么所有使用这些硬件的软件就可以正常运行了。但是，在实际应用中，标准并没有被严格遵守，一般都是由各个组织或公司自行定义规范。因此，就有必要进行配置性测试，配置性测试的目的就是促进被测软件在尽可能多的硬件平台上运行。配置性测试有时经常会与兼容性测试或安装性测试一起进行。

9.11.2 配置性测试方法

在进行配置性测试之前，有两项准备工作需要提前完成。第一，分离配置缺陷。配置缺陷不是普通的缺陷，可用一个简单有效的办法来进行判断，即在另外一台有完全不同配置的计算机上一步步地执行导致问题的相同操作，如果缺陷没有产生，就极有可能是特定的配置问题，在独特的硬件配置下才会暴露出来。第二，计算配置测试工作量。假设有一款新的 3D 游戏，其画面丰富，具有多种音效，允许多个用户联机对战，还可以打印游戏细节以便进行策划。那么此时，至少需要考虑对各种图形卡、声卡、网卡和打印机进行配置性测试。如果决定进行完整、全面的配置性测试，检查所有可能的制造者和组合情况，就会导致巨大的工作量。有效的解决办法是划分等价类，需要找出一个方法把巨大无比的配置可能性减少到尽可能控制的范围。由于没有完备的测试，因此存在一定的风险，但这正是软件测试的特点。配置性测试用例选择应该从以下几个方面考虑：

- 确定所需的硬件类型。联机注册：在选择用哪些硬件来测试时容易忽略的一个特性例子是联机注册。如果软件有联机注册功能，就需要把调制解调器和网络通信考虑在配置性测试中。
- 确定有哪些厂商的硬件、型号和驱动程序可用。确定要测试的设备驱动程序，一般选择操作系统附带的驱动程序，硬件附带的驱动程序或者硬件或操作系统公司网站上提供的最新的驱动程序。
- 确定可能的硬件特性、模式和选项。
- 将确定后的硬件配置缩减到可控制的范围。假设存在成千上万种配置，此时就需要把它们缩减到可以接受的范围内，即测试的范围。一种方法是把所有配置信息放在电子表格中，列出生产厂商、型号、驱动程序版本和可选项。软件测试员和开发小组可以审查这张表，确定要测试哪些配置。注意：用于把众多配置等价划分为较小范围的决定过程最终取决于软件测试员和开发小组。这里没有一个定式，每一个软件工程都不相同，都有不同的选择标准。
- 明确与硬件配置有关的软件唯一特性。不应该也没有必要在每一种配置中完全测试软件。只需测试那些与硬件交互时互不相同的特性即可。选择唯一特性进行尝试并非那么容易，首先应该进行黑盒测试，通过查看产品找出明显的特性，然后与小组成员交流，了解其内部的白盒情况，最后就会发现这些特性与配置有一些紧密的联系。
- 设计在每一种配置中执行的测试用例。
- 在每种配置中执行测试。执行测试用例，仔细记录并向开发小组报告结果，必要时还要向硬件生产厂商报告。明确配置问题的准确原因通常很困难，并且非常耗时，软件测试员需要和程序员紧密合作。如果软件缺陷是硬件的原因，就利用生产厂商的网站向其报告问题。
- 反复测试直到小组对结果满意为止。

配置测试一般不会贯穿整个项目期间。最初，我们可能会尝试一些配置，接着整个测试通过，

然后在越来越小的范围内确认缺陷的修复，最后达到没有未解决的缺陷或缺陷限于不常见或不可能的配置上。

9.12　文档性测试

本节介绍文档性测试的基本概念和方法。

9.12.1　文档性测试的概念

软件产品由可运行的程序、数据和文档组成。文档是软件的一个重要组成部分。在软件的整个生命周期中，会产生许多文档，在各个阶段中以文档作为前阶段工作成果的总结和后阶段工作的依据。软件文档的分类如表 9-5 所示。

表 9-5　软件文档的分类

用户文档	开发文档	管理文档
用户手册、操作手册、维护修改建议	软件需求说明书、数据库设计说明书、概要设计说明书、详细设计说明书、可行性研究报告	项目开发计划、测试计划、测试报告、开发进度月报、开发总结报告

文档性测试(Documentation Testing)主要针对系统提交给用户的文档进行验证，目标是验证软件文档是否正确记录系统的开发全过程的技术细节。通过文档性测试可以改进系统的可用性、可靠性、可维护性和安装性。下面按照表 9-5 列举的文档类型，分别讨论文档性测试的内容。

(1)用户文档测试内容。在测试用户文档时，测试人员假定自己是用户，按照文档中的说明进行操作。在进行文档性测试的时候，可以考虑以下几个方面：

- 把用户文档作为测试用例选择依据。
- 正确地按照文档所描述的方法使用系统。
- 测试每个提示和建议，检查每条陈述。
- 查找容易误导用户的内容。
- 把缺陷并入缺陷跟踪库。
- 测试每个在线帮助的超链接。
- 测试每条语句。
- 表现得像一个技术编辑而不是一个被动的评审者。
- 首先对整个文档进行一般的评审，然后进行一个详细的评审。
- 检查所有的错误信息。
- 测试文档中提供的每个样例。
- 保证所有索引的入口有文档文本。
- 保证文档覆盖所有关键用户功能。
- 保证阅读类型不是太技术化。
- 寻找相对较弱的区域，这些区域需要更多的解释。

(2)开发文档测试内容。

- 系统定义的目标是否与用户的要求一致。
- 系统需求分析阶段提供的文档资料是否齐全。
- 文档中的所有描述是否完整、清晰，准确地反映用户要求。
- 与所有其他系统成分的重要接口是否都已经描述。
- 被开发项目的数据流与数据结构是否足够、确定。

- 所有图表是否清楚，在不补充说明时能否理解。
- 主要功能是否已包括在规定的软件范围之内，是否都已充分说明。
- 软件的行为和它必须处理的信息、必须完成的功能是否一致。
- 设计的约束条件或限制条件是否符合实际。
- 是否考虑了开发的技术风险。
- 是否考虑过软件需求的其他方案。
- 是否考虑过将来可能会提出的软件需求。
- 是否详细制定了检验标准，它们能否对系统定义是否成功进行确认。
- 是否有遗漏、重复或不一致的地方。
- 用户是否审查了初步的用户手册或原型。
- 项目开发计划中的估算是否受到了影响。
- 接口(即分析软件各部分之间的联系，确认软件的内部接口与外部接口是否已经明确定义。模块是否满足高内聚低耦合的要求。模块作用范围是否在其控制范围之内)。
- 风险(即确认该软件设计在现有的技术条件下和预算范围内能否按时实现)。
- 实用性(即确认该软件设计对于需求的解决方案是否实用)。
- 技术清晰度(即确认该软件设计是否以一种易于翻译成代码的形式表达)。
- 可维护性(从软件维护的角度出发，确认该软件设计是否考虑了方便未来的维护)。
- 质量(即确认该软件设计是否表现出良好的质量特征)。
- 各种选择方案(是否考虑过其他方案，比较各种选择方案的标准是什么)。
- 限制(评估对该软件的限制是否实现，是否与需求一致)。
- 其他具体问题(对于文档、可测试性、设计过程等进行评估)。

9.12.2 文档性测试方法

非代码的文档性测试主要检查文档的正确性、完备性和可理解性。正确性是指不要把软件的功能和操作写错，也不允许文档内容前后矛盾。完备性是指文档必须完整，不许漏掉关键内容。文档中很多内容对开发者可能是显然的，但对用户而言不见得都是显然的。文档能否让大众用户看得懂，能否理解术语？缩写能否被用户理解？内容和主题是否一致？很多程序员能编写出好程序，却写不出清晰的文档。与文档作者密切合作，对文档仔细阅读，跟随每个步骤，检查每个图形，尝试每个示例是进行文档测试的基本方法。

行之有效的用户文档测试方法可以分两大类：一是走查，只通过阅读文档，不必执行程序就可完成测试，如文档走查、边界值检查、标识符检查、标题及标题编号检查、引用测试、可用性测试；二是验证，对比文档和程序执行结果，用于测试操作步骤、示例和屏幕截图，如操作流程检查、链接测试、界面截图测试。表 9-6 列出用户文档测试技术与 Bug 类型的关系。

表 9-6 用户文档测试技术与 Bug 类型的关系

	语言类错误		版面类错误	逻辑类错误	一致性错误	联机文档功能错误
文档走查	√		√	√	√	√
数据校对				√	√	
操作流程检查				√		√
引用测试					√	
链接测试						√
可用性测试				√		
界面截图测试					√	

(1) 文档走查。熟悉软件特性的人，只通过阅读文档，来检查文档的质量。走查最有效的工具是检查单。检查单的设计有两条原则：一是横向分块，将文档分为若干部分，划分的基本单位是文档的章节；二是纵向分类，将同一类错误，设计在一个检查单中，只检查规定的检查项。

(2) 数据校对。只需检查文档中数据所在部分，而不必检查全部文档。检查的数据主要有边界值、软件版本、硬件配置、参数默认值等。

① 边界值校对：通过查阅设计文档，检查用户文档中的边界值，例如所需内存最小值、数据表示范围等。如果设计文档中没有给出明确值，需要测试人员测试这些值。

② 软件版本校对：检查操作系统、数据库管理系统、中间件、软件补丁等，保证说明的准确和完整。校对的标准首先是需求文档，其次是软件规格或设计文档。

③ 硬件配置校对：检查软件运行所需要的硬件环境中 CPU、内存、I/O 设备、网络设备，以及专用设备等的名称和型号，保证硬件配置的正确和完整。校对的标准首先是硬件需求配置文档。

(3) 操作流程检查。程序的操作流程主要有：安装/卸载操作过程、参数配置操作过程、功能操作和向导功能。对这些操作流程的检查如同程序的测试，需要运行程序，检查的方法是对比文档是否符合程序的执行流程，检查文档的描述是否准确和易于理解。

操作流程检查与程序测试相似，但是测试人员不需要编写测试用例，文档的输入/输出就是测试输入/输出，如果程序执行的结果与文档不一致，则需要进一步确认是文档的错误还是程序的错误。

(4) 引用测试。文档之间的相互引用，如术语、图、表和示例等，是 Bug 的多发处。加之对于文档中究竟有多少处引用，事先并不清楚。因此，测试起来比较困难。引用是单向指针，适用追踪法，即从文档开始处，逐项检查引用的正确性。

(5) 链接测试。与引用测试类似，但是链接测试是专用于测试电子文档中的超级链接。当超级链接关系复杂时，这项测试也较复杂，需要借助于有向图，否则可能迷失在链接中。测试方法是，为每个链接在有向图中画一条有向边，直到所有的链接都反映到有向图中，如果有失败的链接或不正确的链接，就找到了 Bug。然后，还要分析有向图，每个节点的出度 dego=1，而入度 d≥1，不能有孤立节点。

(6) 可用性测试。本项测试只针对文档的可用性，不涉及整个软件的可用性，软件可用性测试是更复杂的问题。这项测试又分为两种策略：一是由软件专家进行测试，要求测试者是软件专家，对被测试软件的功能非常熟悉，掌握相应领域知识，专家依靠他们的经验和知识完成测试；二是用户测试，选择一些对软件不熟悉但具有操作软件必需领域知识的人员来承担测试工作，他们以用户加初学者的身份测试文档的可用性。

(7) 界面截图测试。界面截图测试需要分为两种情况进行分别测试：一是走查，检查文档的图片的大小、编号、色彩和文档中的位置，以及引用的界面是否正确、合理和有代表性；二是执行程序，对比文档的界面截图与程序是否一致，保证界面截图的连续性，例如，标题、菜单、列表内容、用户名、系统响应等是否与实际程序一致。

9.13　GUI 测试

GUI (Graphics User Interface) 是一种常见的软件系统界面设计模式，通过 GUI，用户可以方便地使用软件产品的功能，所以 GUI 测试在整个软件产品测试中占有非常重要的地位。

9.13.1 GUI 测试的概念及方法

GUI 测试是功能测试的一种表现形式，这种测试不但要考虑 GUI 本身的测试，也要考虑 GUI 所表现的系统功能的测试。一般来说，当一个软件产品完成 GUI 设计后，就确定了它的外观架构和 GUI 元素，这时 GUI 本身的测试工作就可以进行；而 GUI 对应的功能完成之后，才进入功能测试阶段。有时，可以把上述两个阶段加以合并，待功能代码完成后一并进行。GUI 测试可以采用手工测试方法和自动化测试方法完成。

(1)GUI 的手工测试。手工测试方法是按照软件产品的文档说明书设计测试用例，依靠人工单击鼠标的方式输入测试数据，然后把实际运行结果与预期的结果相比较后，得出测试结论。但是，当今的软件产品的功能越来越复杂，越来越完善，一般一套软件包括丰富的用户界面，每个界面里又有相当数量的对象元素，所以 GUI 测试完全依靠手工测试方法是难以达到测试目标的。

(2)GUI 的自动化测试。GUI 的自动化测试方法包括两个方面：一是选择一个能够完全满足测试自动化需要的测试工具；二是使用编程语言，如 Java、C++等编写自动化测试脚本。但是，任何一种工具都不能够完全支持众多不同应用的测试，所以常用的做法是使用一种主要的自动化测试工具，并且使用编程语言编写自动化测试脚本以弥补测试工具的不足。自动化测试的引入大大提高了测试的效率和准确性，而且专业测试人员设计的脚本可以在软件生命周期的各个阶段重复使用。

GUI 的自动化测试可以分为三类。

(1)记录回放。这种方法不需要太多的计划、编程和调试。优点是简单方便。缺点是稳定性差、兼容性差，所以脚本运行寿命很短。同时，由于缺少结果的验证部分，所以很难找出 Bug。

(2)测试用例自动化。这种方法是指将需要反复测试或在多种配置下重复测试的用例自动化。基本实现过程如下：

- 制订测试计划。
- 设计测试用例。
- 针对每一个测试用例评估自动化的可行性和效益。
- 对测试用例做详细步骤分解。
- 编写公用资源库(日志记录、异常处理等)。
- 编写自动化程序。
- 调试。
- 运行。

这类自动化测试最为灵活，能够发现较多的 Bug，并且可以较好地与测试计划相协调。当前，大中型软件企业主要使用这种类型的自动化测试。

(3)自动测试。这种方法是指自动生成测试用例并自动运行。它的最大优点在于可以重复使用。另外，它通常能发现手工测试极难发现的错误。而且一旦实现了这种自动化，其维护费用将大大低于前两类测试。不过，这类自动化测试的初期投入成本非常高，而且它的测试效果受其智能化程度的制约也非常大。这类测试的基本实现过程如下：

- 购买或开发基本测试自动化框架。
- 编写必要的接口及其他公用资源。
- 建立行为模型。
- 设立测试目标参数。
- 自动生成测试计划和测试用例。
- 筛选并执行测试用例。

GUI 开发环境提供了可复用的构件，使得开发工作更加省时、更精确；但从另一个角度看，GUI 的复杂性也增加了设计和执行测试用例的难度，GUI 对软件测试提出了新的挑战。由于 GUI 软件的普遍性，也产生了一些 GUI 测试用例的选择规范。

(1)窗口相关标准。

- 窗口是否基于相关的输入和菜单命令打开。
- 窗口能否改变大小、移动和滚动。
- 窗口中的数据内容能否用鼠标、功能键、方向键和键盘访问。
- 当被覆盖并重新调用后，窗口能否正确地显示。
- 需要时能否使用所有窗口相关的功能。
- 所有窗口相关的功能能否操作。
- 是否有相关的下拉式菜单、工具条、滚动条、对话框、按钮、图标和其他控制可被窗口使用，并适当地显示。
- 显示多个窗口时，窗口的名称是否被适当地显示。
- 活动窗口是否被适当地加亮。
- 如果使用多任务，是否所有的窗口被实时更新。
- 多次或不正确单击是否会导致无法预料的情况。
- 窗口的声音和颜色提示与窗口的操作顺序是否符合要求。
- 窗口是否正确地被关闭。

(2)下拉式菜单和鼠标。

- 菜单项是否显示在合适的语境中。
- 应用程序的菜单项是否显示系统相关的特性(如时钟显示)。
- 下拉式操作是否运行正确。
- 菜单、调色板和工具条是否运行正确。
- 是否适当地列出了所有的菜单功能和下拉式子功能。
- 能否可以通过鼠标访问所有的菜单功能。
- 文本字体、大小和格式是否正确。
- 能否用其他的文本命令激活每个菜单功能。
- 菜单功能是否根据当前的窗口操作加亮或变灰。
- 菜单功能是否正确执行。
- 菜单项是否重复。
- 菜单功能的名字是否具有自解释性。
- 菜单项是否有帮助。
- 在整个交互式语境中，是否可以识别鼠标操作。
- 如果要求多次单击鼠标，能否在语境中正确识别。
- 光标、处理指示器和识别指针是否根据操作做适当的改变。

(3)数据项。

- 字母数字数据项能否正确回显，并输入到系统中。
- 图形模式的数据项(如滚动条)是否正常工作。
- 能否识别非法数据。
- 数据输入消息是否可理解。

9.13.2 一个 GUI 测试案例分析

重复菜单是 GUI 软件开发过程中经常出现的问题之一。在软件开发过程中，由于一些功能相似或一种功能在多处使用，就可能会出现重复菜单。图 9-11 给出一个重复菜单的例子。

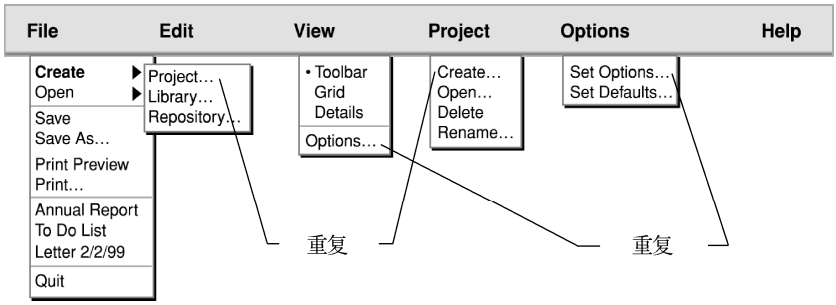


图 9-11 重复的菜单项

其中，File 菜单中出现 Project 命令，而 Project 菜单中也出现 Create 命令；再如，View 菜单中出现 Options 命令，而主菜单中也包括一个 Options 菜单。

重复菜单的出现有时并不是坏事，但一定要明确功能重复后的执行结果的一致性问题。所以，对重复菜单还是要进行检测，以便核实功能的一致性问题。一种检测重复菜单的方法是对每一个重复菜单建立数据字典，并逐一检验其一致性。

9.14 验收测试

验收测试是为了向用户表明系统能够按照用户需求正常运行。经过集成测试后，已经把所有的模块组装成一个完整的软件系统，接口错误也已被基本排除，接着就应当进一步验证功能与性能方面的有效性，这就是验收测试的任务。

9.14.1 验收测试内容与策略

验收测试是部署软件之前的最后一项测试。验收测试的目的是确保软件准备就绪，并且可以让最终用户使用。验收测试的常用策略有三种：正式验收测试、非正式验收测试和 Beta 测试。策略的选择通常建立在合同需求、公司标准以及应用领域的基础上。

(1) 正式验收测试。正式验收测试是一项管理严格的过程，它通常是系统测试的延续。计划和设计这些测试的周密和详细程度不低于系统测试。选择的测试用例应当是系统测试中所执行测试用例的子集。在很多项目中，正式验收测试是通过自动化测试工具执行的。

(2) 非正式验收测试。非正式测试过程的限制不像正式验收测试中那样严格，是仅对需要重点解决的功能和业务进行的测试，测试内容由各测试人员决定。多数情况下，非正式验收测试是由内部测试人员组织执行的测试。

(3) Beta 测试。是开发人员模拟用户进行的测试，可以分别用于上面两种测试工作中。在 Beta 测试中，采用的数据、方法和测试环境完全由各测试人员决定，并决定要研究的功能、特性和任务。测试人员负责确定自己对于系统当前状态的接受标准。

9.14.2　验收测试方法

验收测试用例的设计可以从以下几个方面考虑：

(1) 验收测试的目的是验证软件功能的正确性和软件需求的一致性，所以验收测试的测试用例应该由用户确认验证的标准。

(2) 验收测试用例所覆盖的范围应该只是软件功能的子集，而不是软件的所有功能。在软件开发 V 模型中，验收测试与需求分析阶段是相对应的。因此，验收测试用例与软件需求规格说明书之间具有可追溯性。一个软件产品可能在多个项目中使用，可能具有复杂多样的功能，验收测试不可能也没有必要把研发阶段所有的测试用例都重新执行一遍。

(3) 验收测试用例应当是粗粒度的、结构简单的、条理清晰的测试，而不应当过多地描述软件内部实现的细节。验收测试预期结果的描述，要从用户可以直观感知的方面体现，而不是针对内部数据结构的展示。因此，需要用黑盒测试的方法，尽量屏蔽软件的内部结构。

(4) 验收测试用例的组织应当面向客户，从客户使用和业务场景的角度出发，而不是从开发者实现的角度出发。使用客户习惯的业务语言来描述业务逻辑，根据业务场景来组织测试用例和流程，适当迎合客户的思维方式和使用习惯，便于客户的理解和认同。

(5) 设计验收测试用例应当充分把握客户的关注点。在保证系统完整性的基础上，把客户关心的主要功能点和性能点作为测试的重点，其他的功能点可以忽略，避免画蛇添足。

(6) 验收测试用例可以适当地展示软件的某些独有特性，引导和激发客户的兴趣，达到超出客户预期效果的目的。适当展示软件在某些方面的独特功能，能够为软件增色，特别是在针对招标入围、设备选型、系统演示等目的的测试活动中，可以弥补软件在其他方面的不足，赢得加分的效果。

9.15　回归测试

在软件生命周期中的任何一个阶段，只要软件发生了改变，就可能给该软件带来新的问题。软件的改变可能是源于发现了错误并做了修改，也可能是因为在集成或维护阶段加入了新的模块。为了验证软件修改后的正确性就需要进行回归测试。

9.15.1　回归测试的概念

回归测试是在软件发生变动时保证原有功能正常运作的一种测试策略和方法。回归测试不需要进行全面的测试，而是根据修改的情况进行有选择性的测试。这里所说的保证软件原有功能正常运作，或称为软件修改的正确性，可以从两方面来理解：

- 所做的修改达到了预期的目的，例如缺陷得到了修改，新增加的功能得到了实现。
- 软件的修改没有引入新的缺陷，没有影响原有的功能实现。

当发现软件中有错误时，如果跟踪与管理系统不完善，就可能会遗漏对这些错误的修改；而开发者对错误理解的不透彻，也可能导致所做的修改只修正了错误的外在表现，而没有修复错误本身，从而造成修改失败；修改还有可能产生副作用，从而导致软件未被修改的部分产生新的问题，使本来工作正常的功能产生错误。同样，在有新代码加入软件的时候，除了新加入的代码中有可能会含有错误外，新代码还有可能对原有的代码带来影响。因此，每当软件发生变化时，必须重新测试现有的功能，以便确定修改是否达到了预期的目的，检查修改是否损害了原有的正常功能。同时，还需要补充新的测试用例来测试新的或被修改了的功能。

9.15.2 回归测试方法

回归测试作为软件生命周期的一个组成部分，在整个软件测试过程中占有很大的比重，软件开发的各个阶段都会进行多次回归测试。在渐进和快速迭代的开发过程中，新版本的连续发布使得回归测试进行得更加频繁，而在极限编程方法中，更是要求每天都进行若干次回归测试。因此，通过选择正确的回归测试策略来改进回归测试的有效性是非常有意义的。

对于一个软件开发项目来说，项目的测试组在实施测试的过程中，会将所开发的测试用例保存到测试用例库中，并对其进行维护和管理。当得到一个软件的基线版本时，用于基线版本测试的所有测试用例就形成了基线测试用例库。在需要进行回归测试的时候，就可以根据所选择的回归测试策略，从基线测试用例库中提取合适的测试用例组成回归测试包，通过运行回归测试包来实现回归测试。保存在基线测试用例库中的测试用例可能是自动测试脚本，也有可能是测试用例的手工实现过程。

为了在给定的预算和进度下，高效地进行回归测试，需要对测试用例库进行维护，并且依据一定的策略选择相应的回归测试包。

(1)测试用例库的维护。为了最大限度地满足客户的需要和适应应用的要求，软件在其生命周期中会频繁地被修改和不断推出新的版本。修改后的或者新版本的软件会添加一些新的功能或者在软件功能上产生某些变化。随着软件的改变，软件的功能和应用接口以及软件的实现发生了演变，测试用例库中的一些测试用例可能会失去针对性和有效性，而另一些测试用例可能会变得过时，还有一些测试用例将完全不能运行。为了保证测试用例库中测试用例的有效性，必须对测试用例库进行维护。同时，被修改的或新增添的软件功能仅仅依靠重新运行以前的测试用例并不足以揭示其中的问题，有必要追加新的测试用例来测试这些新的功能或特性。因此，测试用例库的维护工作还应包括开发新的测试用例，这些新的测试用例用来测试软件的新特征或者覆盖现有测试用例无法覆盖的软件功能或特性。

(2)回归测试包的选择。在软件生命周期中，即使一个得到良好维护的测试用例库也可能会变得相当庞大，这使得每次回归测试都重新运行完整的测试包变得不切实际。一个完整的回归测试包括每个基线测试用例，时间和成本的约束可能阻碍运行这样的测试，有时测试组不得不选择一个缩减的回归测试包来完成回归测试。

回归测试的价值在于它是一个能够检测到回归错误的受控实验。当测试组选择缩减的回归测试时，有可能删除了将揭示回归错误的测试用例，消除了发现回归错误的机会。然而，如果采用了代码相依性分析等安全的缩减技术，就可以决定哪些测试用例可以被删除而不会让回归测试的意图遭到破坏。

(3)回归测试的基本过程。有了测试用例库的维护方法和回归测试包的选择策略，回归测试可遵循下述基本过程进行：

① 识别出软件中被修改的部分。

② 从原基线测试用例库 T 中排除所有不再适用的测试用例，确定那些对新的软件版本依然有效的测试用例，其结果是建立一个新的基线测试用例库 T0。

③ 依据一定的策略从 T0 中选择测试用例测试被修改的软件。

④ 生成新的测试用例集 T1，用于测试 T0 无法充分测试的软件部分。

⑤ 用 T1 执行修改后的软件。

第②和第③步测试验证修改是否破坏了现有的功能，第④和第⑤步测试验证修改后的功能。

9.16　测试工具及其应用

测试工具一般分为白盒测试工具、黑盒测试工具、性能测试工具；另外还有测试管理(测试流程管理、缺陷跟踪管理、测试用例管理)工具。

9.16.1　测试种类

在进行软件测试的过程中，要考虑到多种测试类型，比如黑盒测试、白盒测试、性能测试等。常用的测试类型如表 9-7 所示。

表 9-7　常用的测试类型

名称	说明
黑盒测试	基于软件需求，而不是基于软件内部设计和程序实现的测试方式
白盒测试	基于软件内部设计和程序实现的测试方式
单元测试	主要测试软件模块的源代码。一般由开发人员而非独立测试人员来执行，因为测试者需要懂得该单元的设计与程序实现，测试者可能需要编写额外的测试驱动程序
集成测试	将一些“构件”集成一起时，测试它们能否正常运行。构件在这里可以是程序模块、客户机－服务器程序等
功能测试	测试软件的功能是否符合功能性需求，通常采用黑盒测试方式。一般由独立测试人员执行
系统测试	测试软件系统是否符合所有需求，包括功能性需求与非功能性需求。一般由独立测试人员执行，通常采用黑盒测试方式
回归测试	指错误被修正后或软件功能、环境发生变化后进行的重新测试。回归测试的困难在于不好确定哪些内容应当被重新测试
验收测试	由客户或最终用户执行，测试软件系统是否符合需求规格说明书
负载测试	测试软件系统的最大负载，超出此负载软件可能会失常
压力测试	概念上与负载测试相似，叫法不同
性能测试	测试软件在各种状况下的性能，如在正常或最大负载下的状况
易用性测试	测试软件是否易用，主观性比较强。一般要根据很多用户的测试反馈信息，才能评价易用性
安装	测试软件在“全部、部分、升级”等状况下的安装
恢复测试	测试该系统从故障中恢复过来的能力
安全性测试	测试该系统防止非法侵入的能力
兼容性测试	测试该系统与其他软件硬件兼容的能力
比较测试	通过与同类产品比较，考察该系统的优点、缺点
Alpha 测试	一种先期的用户测试，此时系统刚刚开发完成
Beta 测试	一种后期的用户测试，此时系统已经通过内部测试，大部分错误已经改正，即将正式发行

1．Parasoft C++ Test

C++ Test 是 Parasoft 公司出品的一个针对 C/C++源代码进行自动化单元测试的工具。它可以对源代码进行三种测试：白盒测试、黑盒测试以及回归测试。

(1)白盒测试。C++ Test 对 C/C++源代码进行分析，针对所有的类的成员函数(包括公共的、保护的以及私有类型的)进行测试。测试的方法是，判断当输入一个非法的参数时有关函数能否正确处理。在此状态下，软件针对指定的文件、类或者函数自动生成测试用例。

(2)黑盒测试。不对源代码进行分析，并且只针对类的公共接口函数进行测试。在此状态下，软件不自动生成测试用例，而是直接运行在“测试用例编辑器”中已有的测试用例(手工添加的)。

(3)回归测试。回归测试是在修改了源代码后，用原有的测试用例进行重新测试的过程。

C++ Test 的使用比较简单，即可以针对一个 VC 工程进行全面的测试，也可以一次只对一个 C/C++源文件进行测试。当项目比较大时，最好不要直接对一个工程进行自动测试，而应按文件一个一个地测试，否则可能会导致程序死掉。由于其采用 Java 技术开发，所以在使用时最好使用运算速度较快的机器。

2．LoadRunner

Mercury LoadRunner 是一种预测系统行为和性能的负载测试工具。通过模拟上千万个用户实施并发负载及实时性能监测的方式来确认和查找问题，LoadRunner 能够对整个企业架构进行测试。通过使用 LoadRunner，企业能最大限度地缩短测试时间，优化性能和加速应用系统的发布周期。

LoadRunner 的测试原理很简单，用多线程或多进程的方式向服务器端发送大量的数据包，同时接收服务器的返回结果。下面通过一个实例介绍如何使用 LoadRunner。

被测系统：Tomcat 中自带的猜数游戏。

测试目的：测试该应用中的 Jsp 提交表单的性能。

测试步骤：

(1)录制脚本。启动 LoadRunner 后选择 Visual User Generator，在协议选择框中选择 Web(HTTP/ HTML)协议，进入主界面。在工具条上选择 Start Record，弹出启动 Start Recording 对话框。在 URL 输入框中输入上述要测试的第一个页面的 URL。同时，取消选择 Record the application startup，以便手工控制录制开始的时间，跳过刚开始的输入页面。单击 OK 按钮，这时 LoadRunner 会启动浏览器，并指向第一个输入页面，同时在浏览器窗口上方将出现一个“Recording Suspended…”的工具条窗口。等待输入页面显示完全以后，单击工具条窗口中的 Record 按钮，进入录制状态，从现在开始，在打开的浏览器上的所有操作将被录制成测试的脚本。执行预定的表单提交动作，等结果页面显示完整以后，单击工具条上的黑色方框按钮，停止录制，回到 Visual User Generator 的主窗口，脚本录制成功，然后保存脚本。

(2)生成测试场景。选择 Tools 菜单中的 Create Controller Scenario 选项，弹出 Create Scenario 对话框，如需更改测试结果文件生成的路径，可在此更改，其余保持默认值不变，单击 OK 按钮。这时，将启动 LoadRunner 的另一个工具 Controller，这是执行压力测试的环境。首先进入的是 Design 界面，在这里可以调整运行场景的各种参数。如果只是做强度测试，唯一需要调整就是并发用户数，如图 9-12 所示。

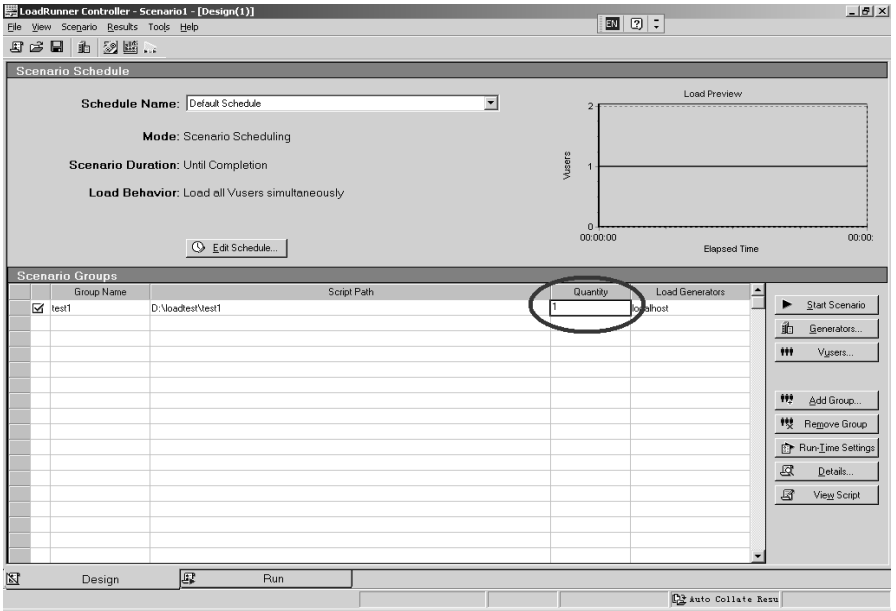


图 9-12　测试场景的生成

这时，运行场景设置完毕，切换到 Run 界面，如图 9-13 所示。

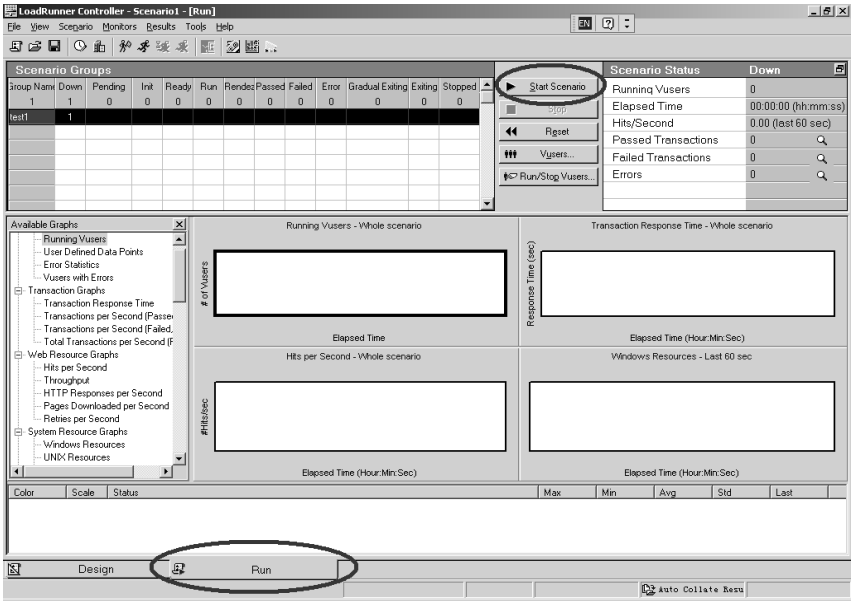


图 9-13　运行场景

单击 Start Scenario 按钮，开始执行测试场景。在执行过程中，左上方的运行状态表格会实时显示当前执行中的虚拟用户的情况，等到所有虚拟用户都执行完毕以后，左下方的四个曲线窗口和底部的数据窗口会显示出测试结果。

(3) 查看测试结果。在上述结果界面上，有四个曲线窗口，主要的结果信息集中反映在上面两个界面上，单击各个窗口，可以看到底部的数据窗口会显示响应数据。左上角的曲线代表随时间变化的虚拟用户数，响应的数据是各个虚拟用户的执行情况，如图 9-14 所示。

Color	Scale	Status	Max	Min	Avg	Std	Last
	1	Error	0.000	0.000	0.000	0.000	0.000
	1	Finished	10.000	0.000	0.833	2.764	10.000
	1	Ready	0.000	0.000	0.000	0.000	0.000
	1	Running	0.000	0.000	0.000	0.000	0.000

图 9-14　测试结果 1

从第二行可以看到，总共有 10 个虚拟用户，都成功执行了预定操作，而采用默认执行方式，意味着所有并发用户一起同步运行，没有分组和时间的先后关系，所以其他数据没有意义，可以不看。右上方的曲线代表响应时间，响应的数据如图 9-15 所示。

Color	Scale	Transaction	Max	Min	Avg	Std	Last
	1	vuser_init_Transaction	0.005	0.002	0.003	0.000	0.003
	1	vuser_end_Transaction	0.000	0.000	0.000	0.000	0.000
	1	Action_Transaction	0.326	0.210	0.272	0.039	0.272

图 9-15　测试结果 2

由于所录制的脚本只有一个动作，而且没有前导和后续动作，所以只需要看“Action_Transaction”一行数据即可。从数据中可以看到，这个表单提交动作在当前压力测试场景下，最长的执行时间是 0.326s，最短的是 0.210s，平均是 0.272s，标准差是 0.039，最后一次响应时间是 0.272s。

LoadRunner 执行时随着虚拟用户数的增加，耗用的系统资源也会增加。一般来说，在 512MB 的机器上可以模拟 500 个并发用户，所以应根据运行 LoadRunner 的机器的性能决定最大的并发用户数。一般的应用系统在 100 个并发用户的情况下就已经是满负载了。

LoadRunner 还有很多图表和数据分析方法，Controller 主界面左下方的树状列表列出了所有可用的数据查看方式。LoadRunner 还有一个专门的数据分析工具，可以根据统计学的原理做进一步的分析。

3．TTworkbench

TTworkbench 是德国 Testing Tech 公司的一款测试平台软件。TestingTech 公司除推出了 TTCN-3 开发和执行工具产品系列 TTworkbench 外，还提供针对 VoIP 进行测试的 TTsuite-VoIP 解决方案产品套件、IPv6 测试、WiMAX 测试等解决方案。结合 TTCN-3 测试语言使用这一平台来进行协议一致性测试，相当方便。

TTworkbench 是使用 TTCN-3 核心语言(TTCN-3 Core Language)来开发测试方案的专业软件。TTworkbench 是一个全整合环境，涵盖测试方案的规格撰写、编译、管理、执行与分析。它包含以下两个组件。

- TTCN-3 Core Language Editor（CL-Editor 组件）：以用户友善之文字为基础的方式，来撰写测试规格定义。
- TTCN-3 Compiler（TTthree 组件）：将以 TTCN-3 语言撰写的模块，编译成可执行的测试套件。

其优点可以归结如下。

- 完整支持 ETSI 的 TTCN-3 标准，包含动态设定、信息导向(message-based)通信、程序导向(procedure-based)通信、模块化和测试控制功能。
- 搭配 Java 可供跨平台使用。
- 通过 TTCN-3 Runtime Interface（TRI），能有弹性地导入要测试的设备。
- 通过 TTCN-3 Control Interface（TCI），能轻易地整合其他的编码解码器。

9.16.2 QACenter

QACenter 是软件黑盒测试工具，Compuware 的 QACenter 家族集成了一些强大的自动工具，这些工具符合大型机应用的测试要求，使开发组获得一致而可靠的应用性能。

QACenter 可以帮助所有的测试人员创建一个快速、可重用的测试过程。这些测试工具自动协助管理测试过程，快速分析和调试程序，包括针对回归、强度、单元、并发、集成、移植、容量和负载建立测试用例，自动执行测试和产生文档结果。QACenter 主要包括以下几个模块。

- QARun：应用的功能测试工具。在 QACenter 测试产品套件中，QARun 组件主要用于客户/服务器应用中客户端的功能测试。在功能测试中主要包括对应用的 GUI(图形用户界面)的测试、回归测试及客户端事务逻辑的测试。由于不断变化的需求将导致应用不同版本的产生，每一个版本都需要对它进行测试，因为每一个被调整的内容往往容易隐含错误，所以回归测试是测试中最重要的阶段，而回归测试通过手工方式是很难达到的，工具在这方面可以大大地提高测试的效率，使测试更具完整性。QARun 组件的测试实现方式是通过鼠标移动、键盘操作被测应用，从而得到相应的测试脚本，对该脚本可以进行编辑和调试。
- QALoad：强负载下应用的性能测试工具。QALoad 是企业范围的负载测试工具，该工具

支持的范围广，测试的内容多，可以帮助软件测试人员、开发人员和系统管理人员对于分布式的应用执行有效的负载测试。负载测试能够模拟大批量用户的活动，从而发现大量用户负载下对 C/S 系统的影响。QALoad 的适用范围很广，可以支持 DB2、DCOM、ODBC、Oracle、NETLoad、Corba、QARun、SAP、SQLServer、Sybase、Telnet、TUXEDO、UNIFACE、WinSock、WWW 等。

- QADirector：测试的组织设计和创建以及管理工具。
- TrackRecord：集成的缺陷跟踪管理工具。
- EcoTools：高层次的性能监测工具。EcoTools 是 EcoSystem 组件产品的基础，可用来应对可用性中计划、管理、监控和报告中的挑战。EcoTools 支持一些主流成型的应用，如 SAP、PeopleSoft、Baan、Oracle、UNIFACE 和 LotusNotes，以及定制的应用。
- TestBytes：是一个用于自动生成测试数据的强大易用的工具，通过简单的单击式操作，就可以确定需要生成的数据类型(包括特殊字符的定制)，并通过与数据库的连接来自动生成数百万行正确的测试数据，可以极大地提高数据库开发人员、QA 测试人员、数据仓库开发人员、应用开发人员的工作效率。
- EcoScope：应用性能优化工具。EcoScope 是一套定位于应用(即服务提供者本身)及其所依赖的所有网络计算资源的解决方案。EcoScope 可以提供应用视图，并标出应用是如何与基础架构相关联的。这种视图是其他网络管理工具所不能提供的。EcoScope 能解决在大型企业复杂环境下分析与测量应用性能的难题。通过提供应用的性能级别及其支撑架构的信息，EcoScope 能帮助 IT 部门就如何提高应用性能提出多方面的决策方案。
- DataFactory：DataFactory 是一种快速的、易于产生测试数据的带有直觉用户接口的工具，它能建模复杂数据关系。DataFactory 是一种强大的数据产生器，它允许开发人员和测试人员可以很容易地产生百万行有意义的正确测试数据库，DataFactory 首先读取一个数据库方案，用户随后单击鼠标产生一个数据库。

DataFactory 支持 DB2、Oracle 等任何与 ODBC 兼容的数据库，支持灵活多样的数据导入和导出操作，并能维持引用关系的完整性和对多种外键的支持。

诸如以上介绍的测试工具还有很多，如数据库测试工具 TestBytes，回归测试工具Rational TeamTest、WinRunner(MI 公司)等。以上只选了几个常用的或前面章节没有涉及的工具来简单介绍其功能和应用，针对不同的测试类型，有不同的测试方法。

思考题

1．什么是压力测试？在实际设计中，压力测试的侧重点是什么？
2．压力测试与其他功能测试相比较有哪些区别？
3．什么是容量测试？
4．容量测试与压力测试的区别有哪些？
5．容量测试分为哪几个步骤？
6．什么是健壮性测试？
7．健壮性测试的执行可以从哪几方面来考虑？
8．结合平时自己开发的小程序，设计一个健壮性测试用例。
9．判断正误：安全性测试最终证明应用程序是安全的。
10．如何测试缓冲区溢出？

11．Windows 画图程序帮助索引包含 200 多个条目(从 airbrush tool 到 zooming in or out.)，是否要测试每一个条目看能否到达正确的帮助主题？假如有 10000 个索引条目呢？

12．简要说明安装测试要考虑的内容。

13．判断正误：测试错误提示信息属于文档测试范围。

14．对一个 GUI 测试案例进行分析。

15．举例说明验收测试的流程。

16．验收测试需要提交哪些文档？

17．回归测试都有哪些方法？

18．请用自己的语言描述：什么是配置性测试。

19．配置性测试分为哪几个步骤？

20．简述软件质量可靠性评估的常用模型。

第 10 章　基于 TTCN-3 的软件测试案例

本章介绍用 TTCN-3 完成的软件测试案例。其中，前面两个案例采用 TTCN-3 树表形式设计测试系统，在 Telelaugic Tau 平台上执行；后面五个案例采用 TTCN-3 核心语言完成，在 TTtree 上运行。

10.1　TTCN-3 在 IPv6 一致性测试中的应用

IPv6 是 IP 的一种新版本，在 Internet 通信协议 TCP/IP 中，对应着 OSI 模型第 3 层(网络层)的传输协议。IP 已经用了 20 多年，随着网络规模的急速发展，出现了许多问题，如规模问题、安全问题等。20 多年前开发的 IP 协议已经难以胜任工作。作为下一代 Internet 基础的 IPv6 经过几年的开发，终于开始由测试阶段向实用阶段过渡，而且已经在国外高速网上试运行。它同目前广泛使用的 IPv4 相比，主要有以下优点。

- 扩展了地址容量。IP 定址空间饱和，IPv6 使得 IP 地址长度从 32 位增到 128 位，增加了大量可以编址的网络节点。在现行的网络中，随着主机数目的增加，决定数据传输路由的路由表在不断加大。路由器的负担已经成为问题。虽然由于使用了 CIDR(无类区域间路由化)技术使路由数目减少，缓和了这一问题。但随着路由数目的增加，这一问题又再次突显出来。IPv6 的 128 位的地址及层次结构的地址分配方案无疑是目前可以看到的最佳解决方案。
- IPv6 对数据包头做了简化，以减少处理器开销并节省网络带宽。
- 改进了对扩展特性和选项的支持。改变了报头选项的编码方式，以允许更有效的转发，减少对选项长度的限制，获得在未来引入新选项的更大的灵活性。
- 对通信服务质量的支持。IPv6 的数据包具有标记以说明它从属的信息流类型，以提供相应的通信服务质量，如实时性视讯、语讯服务。
- 身份认证和安全加密功能。IPv6 提供了对数据确认和完整性的支持，并可通过对数据加密来提高可靠性。
- IPv6 把自动将 IP 地址分配给用户的功能作为标准功能。只要机器一连接上网络便可自动设定地址。它有两个优点：一是最终用户用不着花费精力进行地址设定，二是可以大大减轻网络管理者的负担。IPv6 有全状态自动设定 DHCPv6 和无状态自动设定两种自动设定功能。

由于 IPv6 克服了 IPv4 的不足，必将很快走向实际应用，支持 IPv6 的产品正大量出现。对应同一标准存在许多不同的实现版本。为了保证各种实现版本与 IPv6 协议标准一致及相互之间能够安全、可靠地相互通信，协议的一致性测试是最重要、最基本的手段。

下面根据 IPv6 的特点，参照 TTCN 的语义、语法，提出 IPv6 的形式化测试集描述，并提出一种基于解释的 IPv6 的测试执行策略，以此为指导设计开发 IPv6 测试器将实现对 IPv6 的一致性测试。

10.1.1　IPv6 测试集合的形式化描述

一般的通信协议，特别是数据通信协议的标准，都是文本形式的，其相应的测试集也是自然语言描述，这种自然语言描述的测试集合的最大缺点是二义性和难以用机器处理。形式化的工具则可以避免这两个问题。标准的抽象测试集应使用一种严格定义的、独立于任何实现的形式化描

述方法来描述。目前，已经被标准化的形式化描述语言有 SDL、Estelle 和 LOTOS 等。国际标准化组织(ISO)推荐了一种专门描述测试集的半形式化的描述语言——树表描述语言 TTCN。TTCN 具有两种格式：便于人理解的 GR 格式和便于机器存储和处理的 MP 格式。MP 格式是需要机器识别的格式，因此它必须有形式化的语法定义。ISO9646-3 中给出了 700 多条 BNF(Backus-Naur-Form)语法定义，完整地定义了 TTCN.MP 的语法。用 TTCN 描述的标准的抽象测试集由四部分组成。

IPv6 协议是一个无连接的协议，故只有一个状态(Idle)。两次连续的发送和接收并不会影响彼此的状态。这对协议测试集的设计是一种大大的简化，无须考虑状态的变迁，从而降低了测试集的复杂度。但是，由于难于用状态机模型进行描述，也就无法用基于状态机模型的自动生成技术产生测试集。另外，ICMPv6(Internet 控制信息协议)是 IPv6 不可分割的一部分，必须被所有 IPv6 实现完全支持。ICMPv6 综合了以前在不同协议之间划分的功能，例如 ICMP、IGMP(Internet 组成员协议)和 ARP(地址解析协议)，它既可以发送信息消息，如组成员资格查询、组成成员资格报告、邻居请求、邻居宣告等，又可以发送错误消息来报告信息包处理过程中出现的错误。ICMPv6 消息在 IPv6 信息包中传输，为 IPv6 的测试提供了很多方便。

根据上述特点，设计 IPv6 的测试集合。

为了符合人的思维习惯，得到比较好的测试覆盖，在对 IPv6 协议测试集的设计过程中，采用了基于目的的测试集合设计方法，即根据协议标准，提供测试目的(测试目的的集合要覆盖协议标准的全部行为)，按照测试目的设计测试集。

用此方法抽象化的 IPv6 基本描述(IPv6 Base Specification)协议的测试集共有 6 个测试组，每个测试组又分为几个测试例，每个测试例有若干个测试步。

IPv6 的测试集合基本结构如下。

(1)测试集概述部分：包括测试集合的名称、被测协议标准、PICS、PIXIT、测试方法、测试例库索引、测试步库索引、测试缺省行为库索引、可被重复应用的测试集对象输出列表。

(2)说明部分：包括测试集所有对象的名称、类型、取值范围，如数据结构、定时器、测控点、测试例变量、抽象服务原语、协议数据单元等。

(3)约束部分：定义抽象。

10.1.2 测试方法

协议测试使用测试观测点 PCO(Point of Control and Observation)对被测实现的层间服务原语和协议数据单元进行控制和观察。根据 PCO 的不同，在 ISO9646 中已经被标准化的测试方法有本地测试法和外部测试法。本地测试法适合于在产品内部测试。外部测试法适合于远程的第三方测试，又可分为分布式、协调式和远程式测试。对诸如网关或路由器等中继系统的测试，由于其非开放性，获取其上下接口非常困难，ISO9646 规定的测试法有两种：穿越法和回绕法。

对 IPv6 基本描述协议的测试可以将主机和路由器统一为节点，即通用测试集把主机或路由器当成 IPv6 网络中的一个节点来进行测试，只在测试例的设计上加以区分，测试结构统一采用远程测试法。

IPv6 协议的测试实现过程简述如下。

- 测试集合的设计：由测试集设计人员在对协议文本描述充分理解的基础上，严格按照 TTCN 的标准，利用 IPv6 测试系统的测试集编辑器，设计出适合于人理解的图表格式的测试集 IPv6TC.GR。
- 测试集翻译器：将图表格式的 IPv6 测试集 GR 翻译成机器处理的测试集合 IPv6TC.MP。该翻译器仅仅能翻译与 IPv6 基本描述协议相关的语义语法格式，所以它不是一个通用的

TT CN 翻译器。但作为 TTCN 翻译器的一个子模块，可以在后续的工作中追加、完善。

- 测试例选择：选择器根据某一个 IPv6 实现的 PICS（协议实现一致性声明）和 PIXIT（与测试相关的协议实现附加信息），对测试集进行选择，产生出对该实现进行测试所需要的测试子集 ETS（可执行测试集）。
- 测试执行：用 ETS（IPv6TC.MP 的子集）对被测实现 IUT 进行激发/响应测试。采用了基于解释的测试执行策略，即测试执行器直接调用 ETS 中的测试例进行解释执行，而不需要对整个测试集进行编译预处理，使得用户可以对测试过程进行动态的观察和控制。对于每一个测试例，测试执行器顺序地读入测试步，编码并执行该测试步，激发被测实现 IUT 做出响应，在后续的测试步中，测试执行器接收被测实现响应的 PDU，解码并与测试例中预期的 PDU 进行比较，给出测试结果。
- 测试结果分析和评价：对测试执行产生的测试记录文件进行分析，生成测试判断和协议一致性测试报告。

10.1.3　IPv6 测试集中的一个测试例

在对 IPv6 基本描述协议进行的测试中，共抽化出 6 个测试组，每个测试组又分为几个测试例。下面选取其中的一个测试例加以说明（见表 10-1～表 10-5）。

表 10-1　测试例的动态行为描述

Test Case Dynamic Behavior					
Test Case Name: Unrecognized_next_header_Spec. IPv6					
Reference:		IPv6_Spec_4（RFC2460-4）			
		IcmpV6_3.4（RFC2463-3.4）			
Purpose:		Verify that a node discards a packet with an unknown next header and transmits			
		An ICMPv6 Parameter Problem message to the source of the packet			
Default:		IPv6_no_next_header			
Comment:		Null			
Nr	Label	Behavior Description	Constraints Ref	Verdict	Comment
		+Router_Adver			Preamble
		+Echo_Request			Preamble
		TR! IPv6_2			Unrecognized_next_header
		START TM1	Unreco_next_hdr		
		TR? IPv6_114		pass	Parameter_Problem
		TR?otherwise	Para_Problem	fail	
		?Timeout TM1		fail	

表 10-2　测试例前序动态行为 Router_Adver 的描述

Test Case Dynamic Behavior					
Test Case Name: Echo_Request					
Nr	Label	Behavior Description	Constraints Ref	Verdict	Comment
		TR! IPv6_101	ICMPv6_Msg_01		Preamble
		START TM1			Preamble
		TR? IPv6_102	ICMPv6_Msg-02	pass	Unrecognized_next_header
		TR?otherwise		fail	
		?Timeout TM1		fail	Parameter_Problem

表 10-3 测试例的动态行为描述

Test Case Dynamic Behavior					
Test Case Name: Router_Adver					
Nr	Label	Behavior Description	Constraints Ref	Verdict	Comment
		TR!Ipv6_105 START TM1 TR? IPv6_106 TR?otherwise ?Timeout TM1	 Unreco_next_hdr Para_Problem	 pass fail fail	
Comment1: Router Advertisement cause the NUT to add TR to its Default Router List, and cause the NUT to auto-configure its global address from the link's prefix, and cause the NUT to compute Reachable Time, and cause the NUT to send a Neighbor Solicitation message to TR for MAC address					

表 10-4 测试例协议数据单元 Unreco_next_hdr 的限定

PDU Constraint Declaration		
Constraint Name: Unreco_next_hdr PDU Type Ipv6_02 Derivation Path: Comments Unrecognized header		
Field Name	Field Name	Comment
Ver	'6'H	Version
Traf	'0H	Traffic class
Flow	'0H	Flow label
PayL	'0H	Payload Length
NextH	'138'	Next Header Unrecognized
Hoplim	'255'	Hop Limit
SA	TR_Address	Source Address
DA	NUT_Address	Destination Address

表 10-5 测试例协议数据单元 Para_Problem 的限定

PDU Constraint Declaration		
Constraint Name: Para_Problem PDU Type Ipv6_114 Derivation Path: Comments: Parameter Problem Message		
Field Name	Field Name	Comment
Ver	'6'H	Version
Traf	'0H	Traffic class
Flow	'0H	Flow label
PayL	?	Payload Length
NextH	'58'	Next Header Unrecognized
Hoplim	'255'	Hop Limit
SA	NUT_Address	Source Address
DA	TR_Address	Destination Address
Type	'4'	Parameter Problem
Code	'1'	Unrecognized next header
Checksum		Chechsum
Pointer	ox40	Offset of the Next Head Field

测试器先执行两个前序(Preamble)测试行为：Router_Adver 和 Echo_Request(表 10-2、表 10-3 给出其动态行为描述)。TR(Test router or test node)发送数据报 IPv6_2(其格式及参数值在表的 Unreco_next_hdr 中限定)给 NUT(Node Under Test)：启动时间计数器，如收到 IPV6_114 则表明 NUT 做出了正确的响应，判断为不能识别的下一报头，测试例通过。如果收到其他包或时钟超时，则判断为测试失败。

以上根据 IPv6 的特点，按照 ISO9646-3 定义的标准，设计出符合 TTCN 语义、语法结构的 IPv6 的形式化测试集，并参照 ISO9646 标准，以此为指导设计开发的 Ipv6 测试器将用于对 IPv6 的一致性测试，并为实现通用的协议一致性测试系统奠定基础。

10.2　基于 HTTP 协议应用系统的测试

本节介绍基于 HTTP 协议应用系统的网络软件一致性测试。首先简单介绍 HTTP 协议过程，然后建立 HTTP 的形式化模型，最后介绍 TTCN-3 测试套。

10.2.1　HTTP 协议

HTTP 协议是一种应用层协议，HTTP 是 HyperText Transfer Protocol(超文本传输协议)的英文缩写。HTTP 可以通过传输层的 TCP 协议在客户端和服务器之间传输数据。HTTP 协议主要用于 Web 浏览器和 Web 服务器之间的数据交换。我们在使用 IE 或 Firefox 浏览网页或下载 Web 资源时，在地址栏中输入 http://host:port /path，开头的 4 个字母 http 就相当于通知浏览器使用 HTTP 协议来和 host 所确定的服务器进行通信。

无论你是从事网络程序开发或 Web 开发，还是网站的维护人员，都必须对 HTTP 协议有一个比较深入的了解。因此，HTTP 协议不仅是 Internet 上应用最为广泛的协议，也是应用协议家族中比较简单的一种入门级协议；而且所有的 Web 服务器无一例外地都支持 HTTP 协议。这也充分地说明，对于那些开发网络程序，尤其是开发各种类型的 Web 服务器的开发人员，透彻地掌握 HTTP 协议将对你所开发的基于 HTTP 协议的系统产生直接的影响。本节的目的并不是讲解 HTTP 协议，只讨论了 HTTP 协议的主要部分。

HTTP 协议采用了请求/响应的工作方式。基于 HTTP1.0 协议的客户端在每次向服务器发出请求后，服务器就会向客户端返回响应消息(包括请求是否正确，以及所请求的数据)，在确认客户端已经收到响应消息后，服务端就会关闭网络连接(其实是关闭 TCP 连接)。在这个数据传输过程中，并不保存任何历史信息和状态信息。因此，HTTP 协议也被认为是无状态的协议，图 10-1 描绘了 HTTP1.0 协议的通信过程。

在 HTTP1.0 协议中，当 Web 浏览器发出请求时，就意味着一个请求/响应会话已经开始。在请求、响应结束后，服务器就会立刻关闭这个连接。这种会话方式虽然简便，但它会带来另外一个问题。如果客户端浏览器访问的某个 HTML 或其他类型的 Web 页中包含其他的 Web 资源，如 JavaScript 文件、图像文件、CSS 文件等；当浏览器每遇到这样一个 Web 资源时，就会建立一个 HTTP 会话。如果这样的资源很多，就会加重服务器的负担，同时也会影响客户端浏览器加载 HTML 等 Web 资源的效率。

在对上述的缺陷进行改进和完善后，HTTP1.1 协议进入了我们的视线。 HTTP1.1 和 HTTP1.0 相比较而言，最大的区别就是增加了持久连接支持。当客户端使用 HTTP1.1 协议连接到服务器后，服务器就将关闭客户端连接的主动权交还给客户端；也就是说，在客户端向服务器发送一个请求并接收以一个响应后，只要不调用 Socket 类的 close 方法关闭网络连接，就可以继续向服务器发

送 HTTP 请求。当 HTML 中含有其他的 Web 资源时，浏览器就可以使用同一个网络连接下载这些资源，这样就可以大大减轻服务器的压力。图 10-2 演示了这一过程。

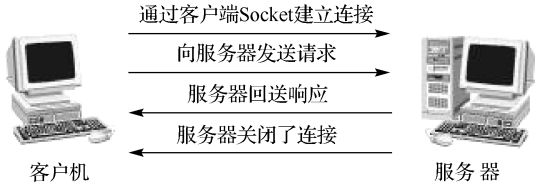


图 10-1　HTTP1.0 协议的通信过程

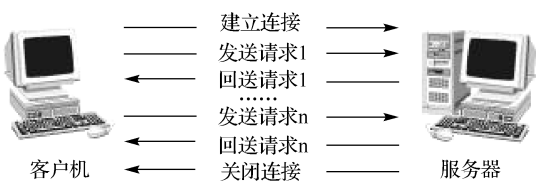


图 10-2　HTTP1.1 协议的通信过程

HTTP1.1 除了支持持久连接外，还将 HTTP1.0 的请求方法从原来的 3 个(GET、POST 和 HEAD)扩展到了 8 个(OPTIONS、GET、HEAD、POST、PUT、DELETE、TRACE 和 CONNECT)。而且还增加了很多请求和响应字段，如上述的持久连接的字段 Connection。这个字段有两个值：Close 和 Keep-Alive。如果使用 Connection:Close，则关闭 HTTP1.1 的持久连接的功能。要打开 HTTP1.1 的持久连接的功能，必须使用 Connection:Keep-Alive，或者不加 Connection 字段(因为 HTTP1.1 在默认情况下就是持久连接的)。除此之外，还提供了身份认证、状态管理和缓存(Cache)等相关的请求头和响应头。

10.2.2　HTTP 协议软件一致性测试

1．被测系统简介

根据 RFC2616 规范，HTTP 协议消息的发送方法有多种。我们选取两个测试例来对其进行一致性测试。其一为发送方法使用“GET”方法的网站服务器：www.163.com；其二为使用“POST”方法的网站服务器：www.ip138.com。针对这两个不同发送方法的服务器对 HTTP 协议实现的正确性进行验证。

2．抽象测试套的设计

测试套的设计完全符合 RFC2616 规范的规定。

(1)首先根据 RFC2616 定义和 TTCN-3 的命名规范，定义其数据类型。

请求格式定义如下：

```
type record Request {
    RequestLine requestLine,                  //定义请求行
    RequestHeader requestHeader optional,     //请求消息头部域
    GeneralHeader generalHeader optional,     //通用头部域
    EntityHeader entityHeader optional,       //实体头部域
    MessageBody messageBody optional          //消息体
}
```

以上定义中的 optional 代表可选部分，用户可以根据自己的需要来选择；RequestLine、RequestHeader 等均为用户自定义类型，根据请求行的规范 RFC2616 对其格式进行定义：

```
type record RequestLine {
    Method method,                            //请求方法
    URL requestUri,                           //Request-URI
    Version version                           //协议版本号
}
```

同样，请求行里各个子项也均为用户自定义类型，根据规范 RFC2616 的定义来进一步定义 Method、URL、Version 的数据类型。针对 HTTP 协议请求的多种方法，我们定义其为枚举类型(enumerated)，在 URL 里定义协议名、主机名、端口名以及路径名；最后定义协议版本号：

```
type enumerated Method {
    OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT
}
type record URL {
    charstring protocol,
    charstring hostName,
    integer portNumber optional,
    charstring path optional
}
type charstring Version;
```

与请求(request)相同，定义返回消息(response)的方法，根据规范 RFC2616 的规范定义 TTCN-3 数据类型：

```
type record Response {
    StatusLine statusLine,
    ResponseHeader responseHeader optional,
    GeneralHeader generalHeader optional,
    EntityHeader entityHeader optional,
    EntityBody entityBody optional
}
```

以状态行为例来说明自定义类型的定义过程：

```
type record StatusLine {
    Version version,//与以上定义相同
    StatusCode statusCode,
    ReasonPhrase reasonPhrase
}
type integer StatusCode;
type charstring ReasonPhrase;
```

(2)根据类型定义设计相应的模板。

在模板里主要根据类型定义和用户对测试例的设计来填写测试数据，以此作为发送消息和预期接收消息的内容。

```
template GET_Request GET_Request_1 := {  //GET 方法
        requestLine := {   //请求行的请求数据
            method := GET,
            requestUri := {
                protocol := "http://",
                hostName := "www.ncut.edu.cn",
                portNumber := 80,
                path :="/index.html"
            },
            version := "HTTP/1.1"
```

```
            }
    }
    template POST_Request POST_Request_1 := { //POST 方法
            requestLine := {
                method := POST,
                requestUri := {
                    protocol := "http://",
                    hostName := "www.ip138.com",
                    portNumber :=80,
                    path := "/ips8.asp"
                },
    version := "HTTP/1.1"
            }
    }
```

3．编码解码器的设计

HTTP 协议一致性测试中数据格式的转换如表 10-6 所示。

表 10-6　HTTP 协议一致性测试中数据格式的转换

TTCN-3 中的模板数据	网络上传输的数据类型

template GET_RequestGET_Request_1 := {
requestLine := {
method := GET,
requestUri := {
protocol := "http://",
hostName := "www.ncut.edu.cn",
portNumber := 80,
path :="/index.html"
},
version := "HTTP/1.1"
	method↓protocal hostName protNumber path↓version

下面给出 encode()方法的伪代码表示：

```
Value 类型的对象转换为 RecordValue 类型的对象；
通过 RecordValue 对象得到各域名并存放在字符串数组中；
第一域值赋给 EnumeratedValue 对象；
通过 EnumeratedValue 对象得到第一域值字符串并添加到 StringBuffer 对象中；
添加换行符到 StringBuffer 对象中；
第二域值赋给 RecordofValue 对象；
For(int i=0；i<RecordofValue 对象的域长；i++)
    各域值赋给 CharstringValue 对象；
添加换行符；
通过 CharstringValue 对象得到各域值字符串并添加到 StringBuffer 对象中；
尾域值赋给 CharstringValue 对象；
通过 CharstringValue 对象得到第二域值字符串并添加到 StringBuffer 对象中；
通过 StringBuffer 对象得到字节流并创建 TriMessage 实例；
返回 TriMessage 实例；
同时解码过程为编码过程的逆过程，下面同样给出伪代码表示：
得到待解码的字符串；
If(type.getTypeClass()等于 TciTypeClass.RECORD)
```

```
        创建 RecordValue 新实例 A;
        通过 RecordValue 对象得到各域名并存放在字符串数组中;
        通过 RecordValue 对象得到头域并创建 RecordValue 新实例 B;
        通过 A 对象得到各域名并存放在字符串数组 S 中;
        设置 CharstringValue 对象的字符串;
        将 CharstringValue 对象添加到 B 对象的域中;
        设置 IntegerValue 对象的字符串;
        将 IntegerValue 对象添加到 B 对象的域中;
        设置 CharstringValue 对象的字符串;
        将 CharstringValue 对象添加到 B 对象的域中;
        设置 RecordValue 对象 B 的尾域;
        返回 RecordValue 对象 A;
    Else
        记录消息日志;
    返回空;
```

4．被测系统适配器的设计

triSend()的实现方法如下。

创建连接后判断不同的方法来发送消息，以下为伪代码表示：

```
    得到发送消息的字节流;
    将字节流转换为请求字符串;
    截取消息发送方法 method 字符串;
    URI 及 HTTP 协议版本 version;
    建立 URL 连接;
    if(连接方法 method 为 POST)
    设置请求方法为 POST 方法;
    截取请求内容字符串;
    得到输出流;
    请求内容写入输出流;
    else if(连接方法 method 为 GET)
            设置请求方法为 GET 方法;
    创建接收线程接收从被测系统返回的消息:
        创建接收线程;
        得到输入流;
        通过输入流读取响应字符串;
        通过响应字符串创建 TriMessage 对象;
        将 TriMessage 对象传送给 TTCN-3 执行环境;
        关闭输入流;
        断开连接;
        开启接收线程;
        关闭输出流;
    返回 TriStatus 实例;
```

5．测试执行

(1)运行测试套生成向导(如图 10-3 所示)。

填写测试所需的测试用例消息。

Container：抽象测试套存放位置选择。

File name：抽象测试套名(用户自定义)。

Method：消息发送方法(GET/POST)。

Protocol：协议类型。

HostName：请求地址信息。

Port：协议端口号。

Version：协议版本号。

State Code：返回状态码。

Reason Phrase：返回消息。

(2)将生成的抽象测试套编译生成可执行测试套。

(3)加载编码解码器和被测系统适配器的 jar 包。

(4)运行测试。

(5)查看测试结果(如图 10-4 所示)。

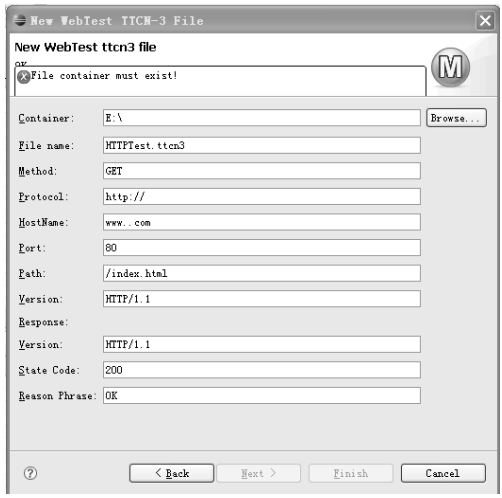


图 10-3 抽象测试套生成向导

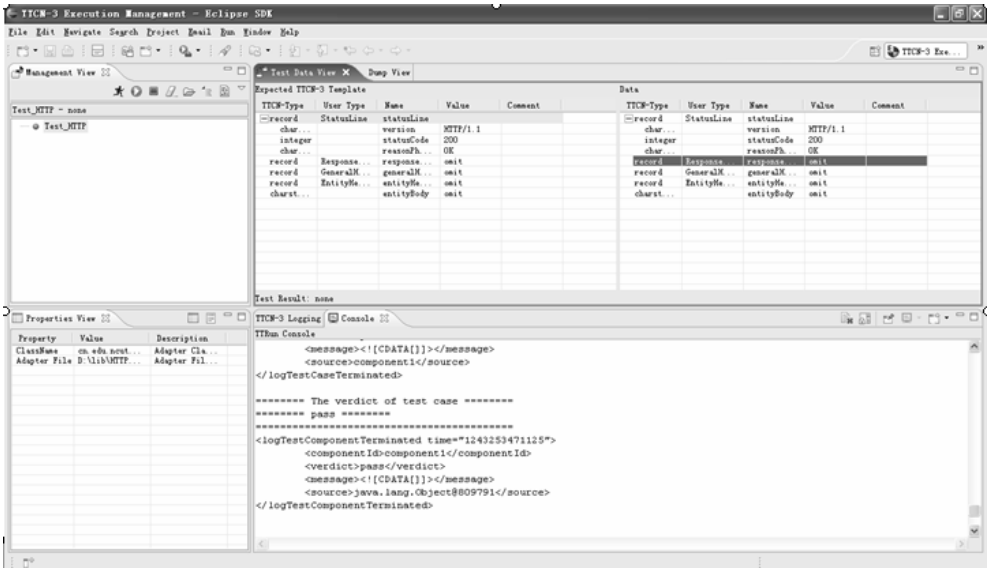


图 10-4 测试结果

10.3 天气预报服务的功能测试

1．测试目的

测试天气预报服务的功能，验证基于 TTCN-3 的 Web Service 功能测试框架的正确性。

2．被测系统简介

该 Web Service 服务来自 http://www.webservicex.net 上的天气预报服务(如图 10-5、图 10-6 所示)。该服务向所有用户提供其服务的 URL，用户通过该 URL 访问服务，可查询全球各国部分重要城市的天气。同时，网站提供了该服务的 WSDL 文件。

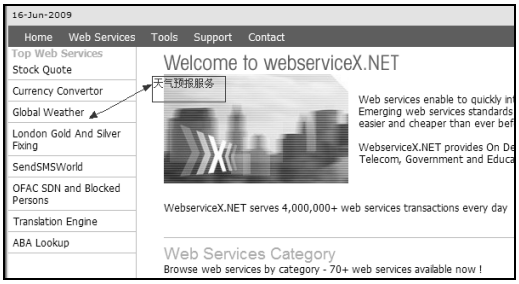


图 10-5　天气预报服务

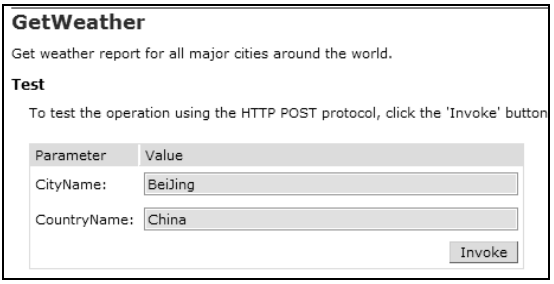


图 10-6　选择 GetWeather 后的界面

3．抽象测试套的设计

根据其 WSDL 文件，得知其消息的请求格式，用户向服务器端发送服务名、城市名和国家名，服务器得到消息后，返回给用户相应的返回消息，主要包括请求状态、气温、风力、时间等消息。如果查询失败，则返回“Data Not Found”消息。具体过程如图 10-7 所示。

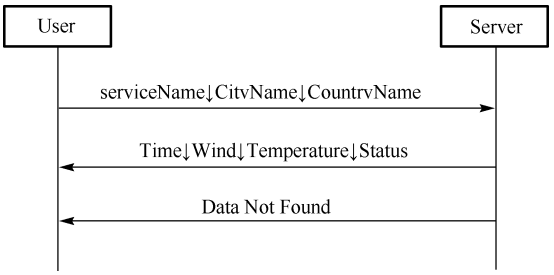


图 10-7　消息请求交互过程

根据图 10-7 的消息格式来设计 TTCN-3 抽象测试套的格式。

首先，定义其消息请求格式：

```
type record wsRequestType {
    charstring serviceName,            //服务名
    charstring cityName,               //城市名
    charstring countryName             //国家名
}
```

根据数据类型的定义来设计服务请求和接收数据的模板：

```
template wsRequestType Request_001 := {
        serviceName := "GetWeather",
        cityName := "Beijing",
        countryName := "China"
        }
template charstring Response_001 := "Success"; //判断返回的状态是否为成功
```

接着定义端口和测试成分的设置，最后将测试例的执行序列显示如下：

```
testcase TC_001() runs on mtcType system systemType {
           var ptcType ptc;
           ptc := ptcType.create;
           map(ptc:wsPort, system:wsPort);
```

```
                wsPort.send(Request_001);
                localTimer.start;
                alt {
                    [] wsPort.receive(Response_001) {
                           localTimer.stop;
                           setverdict(pass);
                     }
                    [] wsPort.receive {
                           localTimer.stop;
                           setverdict(fail);
                     }
                    [] localTimer.timeout {
                           setverdict(fail);
                     }
                }
            }
```

4．编码解码器的设计

编码解码器的设计与上节介绍的思想相同。

5．被测试系统适配器的设计

triSend 创建了一个 SOAP 消息，并向服务发送请求，然后，监听返回的 SOAP 消息并解析。triSend 利用 WSDLToJava 工具生成的 SOAP 客户端存根类，创建存根对象并调用 TranslateService 提供的 Translate 方法，得到返回的译文。在下面的代码中，stub.translate()方法是调用 SOAP 客户端存根类里的方法。

```
TranslateServiceSoapStub stub=New TranslateServiceSoapStub(new URL
("http://www.webservicex.net/TranslateService.asmx "),new Service());
String s = stub.translate(Language.fromString(para), text);
```

6．测试执行

加载编译生成的可执行测试套(.mlf 文件)，同时加载编码解码器与适配器的 JAR 包，执行测试，查看日志。

10.4 魔兽游戏的测试

1．魔兽游戏简介

魔兽游戏中类的定义和设计非常重要，合理的设计才会令程序更容易编写，才能实现程序的设计。在魔兽网络游戏中将游戏分为客户端和服务器，两者都可以作为独立的程序或包运行。

客户端定义了 10 个主要类，关于窗口的有 4 个类，分别是登录类(Logon)、游戏大厅类(ChatHall)、启动游戏类(StartFrame)和主游戏类(War)，这些类在游戏过程中按顺序启动。关于游戏过程和动画实现的有 6 个类，分别是超角色类(SuperSprite)以及继承此类的兽人类(Orc)和暗夜类(Ne)、超武器类(SuperItem)以及继承此类的兽人武器类(Orc_Item)和暗夜武器类(Ne_Item)。这 6 个类的对象都是由主游戏类实例化的。下面分别详细介绍客户端类的定义以及实现方法。

1) 登录类 Logon

图 10-8 是一个无标题栏无菜单的窗口，完全是模仿多数游戏的启动画面，等待游戏载入内存中后，此窗口消失。此窗口是在等待连接服务器，如果成功则启动游戏大厅窗口，否则弹出“是否重新连接服务器”对话框。

Java 要实现此类窗口可以通过继承 Window 类，此类不同于 Frame 类的就是不具有标题栏、菜单和边框等。为了使本窗口提示连接服务器正在进行，还用线程实现了动画效果。

2) 游戏大厅类 ChatHall

在图 10-9 中，看到游戏大厅的感觉与 QQ 游戏大厅一样。在这里，玩家可以选择喜欢的桌子坐下开始游戏，也可以和其他玩家一起聊天。由此也可以看出游戏大厅类内需要两线程：绘图线程和聊天线程。

图 10-8　登录窗口

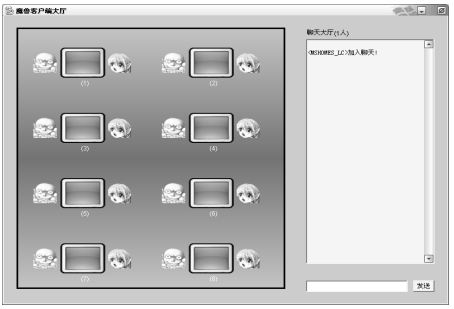


图 10-9　游戏大厅窗口

绘图线程负责在窗口左边的画布里绘制桌子与头像，在画布中添加鼠标监听事件，实时判断鼠标位置，当鼠标进入人物头像时，头像周围会显示边框，单击后开始游戏。8 张桌子分别定义为 8 个 Desk 类数组，然后在不同位置分别绘制 8 个桌子，同时在桌子上显示玩家的主机名。

Desk 类内的主要方法有：getBorder 取出头像的边框范围，以便在移动鼠标时判断鼠标是否移动到边框内；paintBorder 绘制人物边框，在鼠标移动进入头像时调用；paintName 绘制玩家名字；paintDesk 绘制桌面图片；构造函数 public Desk(int index_i, int index_j)用来区别数组在画布的第几行第几列，分别在间隔的位置绘制。

聊天线程 ClientThread 继承 Thread 类，负责发送玩家的聊天内容和坐上桌子请求，以及接收服务器广播的聊天内容和其他消息。ClientThread 的 Socket 占用客户端的 6000 端口与服务器通信。

```
sPort = 6000;
sName = InetAddress.getLocalHost().getHostAddress();
cSocket = new Socket(sName, sPort);
write = new PrintStream(cSocket.getOutputStream());
read = new BufferedReader(new InputStreamReader(cSocket.getInputStream()));
```

ClientThread 的 run 函数不断监听是否有服务器发出消息，一旦收到消息就根据消息的不同执行不同的操作。

```
public void run() {
    String m;
    while (true) {
      try {
        m = read.readLine();
```

```
//      接收服务器当前有多少人在线
        if (m.equals("&hallcount&")) {
          chatHall.chatCout.setText("聊天大厅(" + read.readLine() + "人)");
        }
//      接收谁在某桌子上游戏，选择某角色
        else if (m.equals("&onDesk&over")) {
         ……
         }
         else if (m != null) {
          chatHall.message.append("\n" + m);
        }
         ……
```

在鼠标单击事件中有如下代码：

```
new StartFrame("ORC",i); //启动选兽人的界面
clientThread.write.println("&onDesk& ");
clientThread.write.println(i + "\nORC");
```

通知服务器要坐到位置上并且是坐到 i 号桌子的兽人位置。这样服务器收到消息就可以为此玩家匹配对应的玩家了。

图 10-10 游戏启动窗口

3) *启动游戏类* StartFrame

如图 10-10 所示，在单击座位进入此窗口后，刚开始显示的只有玩家选择的人物头像而非显示全部头像，只有当另一玩家也进入和自己同桌的游戏时才显示对方，“开始”按钮也才会有效。在两玩家都单击“开始”按钮后，游戏才能继续，而其中某个玩家同意并不能令游戏开始。

此类比较简单，只需绘制图片并产生命令线程(使用 6001 端口通信)即可，但如何判断两个玩家都已经单击过开始？这一判断可以交给服务器处理，也可以在客户端判断；在服务器上处理必然会加大服务器的负载，而且需要设立单独的变量，当判断所有在线玩家时必然使程序复杂化，所以将此处理客户端比较合理。方法如下：

```
//鼠标单击事件
  public void mouseClicked(MouseEvent e) {
     actionThread.write.println("&Start&");
  }
// ActionThread 监听消息的 run 方法
public void run() {
    while (true) {
      try {
        String action = read.readLine();
        if (action.equals("&StartFrame&over"))
          startFrame.startframe = true;
        else if (action.equals("&Start&over")) {
          if(!startFrame.mouseEnable){
```

```
                write.println("&HaveStart&");
                war = new War(startFrame.selectID, this);
              }
            }
            else if (action.equals("&HaveStart&over")){
              war = new War(startFrame.selectID, this);
            }
        …
```

此方法类似于 TCP 的三次握手，玩家 1 单击“开始”按钮后向玩家 2 发送“开始 1”消息，只有当玩家 2 收到此消息且单击“开始”按钮后才向玩家 1 发送“开始 2”消息，同时启动游戏，而玩家 1 收到“开始 2”后即可启动游戏(此时玩家 1 必定已经单击过“开始”按钮，否则就不会收到此消息)，这样便实现了只有两个玩家都单击过“开始”按钮后才会启动游戏。

4) 主游戏类 War

如图 10-11 所示，主游戏类的通信线程并没有重新生成，而是直接从启动游戏类中得到 ActionThread 类来发送和接收消息的。主游戏类窗口出现时先显示的是关于如何游戏和版权的说明，单击“Let’s Play”后出现此画面开始游戏。

主游戏类的主要作用是控制整个游戏的流程、玩家同步显示、算法设计及动画的绘制等，而其中最主要的是动画的绘制。

魔兽网络游戏中服务器的架设与游戏的动画制作同等重要，服务器的任务是与客户端联机、接收客户端消息、处理消息并发送消息给客户端。从客户端收到的消息有两类：聊天消息和命令消息。所以要求服务器占用 6000 和 6001 端口对客户端的请求进行监听。服务器启动界面如图 10-12 所示，左边显示关于聊天请求的内容，右边显示关于命令操作的内容。

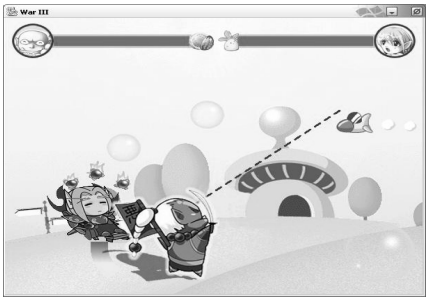


图 10-11　主游戏类窗口

图 10-12　魔兽服务器启动界面

魔兽服务器设计为服务器主程序(WarServer)、服务器线程(ServerThread)、客户端线程(SClientThread)、广播线程(BroadCast)、清除线程(CleanDeadConnect)和游戏命令线程(GameThread)，其中游戏命令线程又包括命令接收线程(GameClientThread)和命令发送线程(GameSendThread)。

魔兽聊天服务的端口是 6000，当窗口启动并单击“开始服务”按钮后，服务器便等待客户端的联机，服务器每接收到一个成功的客户端联机后便由服务器线程产生了一个 Socket 对象，每一个 Socket 对象对应一个客户端线程(SClientThread)，保持与客户端的联机操作，接收客户端的消

息。当 SClientThread 对象建立成功后插入到客户端队列类中，等待广播线程对队列中客户端进行消息广播。

```
public void run() {
    while (true) {
      try {
        Socket cSocket = sSocket.accept(); //建立与客户端联机
        SClientThread cThread = new SClientThread(cSocket, this);
        cThread.start();
        synchronized (Clients) {
          Clients.addElement(cThread);
        }
      }
      catch (IOException ex) {
      }
    }
  }
```

当 SClientThread 的对象接收到消息后将消息内容存放到消息队列类中，等待广播线程取出消息。

```
public void run() {
    while (true) {
      try {
        String Message = read.readLine();
        synchronized (sThread.messages) {
          if (Message != null) {
              if (Message.equals("bye")) {
              ……
           }
      }
      catch (IOException E) {
        break;
      }
```

广播线程(BroadCast)每隔 1s 的时间查询消息队列中是否有消息。如果有消息则从队列中取出该消息，将该消息的内容发送给客户端队列中的所有客户，然后等待 1s 继续执行以上操作，便实现了消息的广播。

```
public void run() {
    while (true) {
      try {
        Thread.sleep(1000);
      }
      catch (InterruptedException E) {}
      synchronized (sThread.messages) {
        if (sThread.messages.isEmpty())
          continue;
        m = (String) sThread.messages.firstElement();
        sThread.messages.removeElement(m);
```

```
        }
        synchronized (sThread.Clients) {
          for (int i = 0; i < sThread.Clients.size(); i++) {
            c = (SClientThread) sThread.Clients.elementAt(i);
            c.write.println(m);
          }
```

清除线程(CleanDeadConnect)每隔 10s 查询客户端队列中的客户端联机是否正常。若不正常，则从队列中清除该客户端线程。

```
    public void run() {
      while (true) {
        try {
          Thread.sleep(10000);
        }
        catch (InterruptedException E) {}
        synchronized (sThread.Clients) {
          for (int i = 0; i < sThread.Clients.size(); i++) {
            temp = (SClientThread) sThread.Clients.elementAt(i);
            if (!temp.isAlive()) {
              SClientThread.ConnectNumber--;
              …
              sThread.warServer.serverConnect.append(m);
              synchronized (sThread.messages) {
                sThread.messages.addElement(m);
              }
              sThread.Clients.removeElementAt(i);
            }
```

游戏命令线程(GameThread)与服务器线程(ServerThread)功能类似，没有以上的三个线程，取而代之的是接收线程和发送线程。因为游戏消息的发送不同于消息的广播，它只允许发送给指定的玩家，而不是所有的玩家。所以设立了和桌子数目一样多的数组来管理。并且定义了继承 Vector 的新队列类 ConnectVector，增加了更多队列的操作功能：直接更新指定位置元素 updateElementAt(Object obj, int index)和得到当前同桌玩家的线程 getNearElemAt(Object obj)。

```
    public synchronized void updateElementAt(Object obj, int index) {
        removeElementAt(index);
        add(index, obj);
    }
    public synchronized Object getNearElemAt(Object obj) {
        int index;
        index=this.indexOf(obj);
        if (index % 2==0)
          return elementAt(index + 1);
        else
          return elementAt(index - 1);
      }
```

游戏命令线程的接收消息和发送消息代码如下：

```
//接收消息
  public void run() {
    while (true) {
      try {
        String action = read.readLine();
        synchronized (Action) {
          if (action != null) {
            Action.addElement(action);
          }
        }
      }
      catch (IOException ex) {
        break;
      }
     …
//发送消息
  public void run() {
    while (true) {
      try {
        Thread.sleep(40);
      }
      catch (InterruptedException ex) {
      }
      neargClientThread=(GameClientThread)gClientThread.gameThread.Clients.
                        getNearElemAt(gClientThread);
      synchronized (gClientThread.Action) {
        if (gClientThread.Action.isEmpty()||neargClientThread==null)
          continue;
        action = (String) gClientThread.Action.firstElement();
        gClientThread.Action.removeElement(action);
        if(action.equals("&StartFrame&"))
          neargClientThread.write.println("&StartFrame&over");……
```

与服务器相比，单机部分的网络功能实现要相对简单些，只需在触发事件中发送，在线程中接收。单机网络功能的实现是建立一个含有服务器地址和端口号的 Socket 对象，如果服务器地址和端口发生变化，则客户端也要更改。为了在任何地址的主机上都能测试，将客户端 IP 设置为本机地址。

```
public ClientThread(ChatHall chatHall) throws UnknownHostException,
     IOException {
   this.chatHall = chatHall;
   sPort = 6000;
   sName = InetAddress.getLocalHost().getHostAddress();
   cSocket = new Socket(sName, sPort);
   write = new PrintStream(cSocket.getOutputStream());
   read = new BufferedReader(
      new InputStreamReader(cSocket.getInputStream()));
}
```

客户端的游戏线程建立同样使用此方法，只需把端口变为 6001 即可。在消息发送中，游戏大厅的触发事件有单击桌子的座位和点击发送聊天内容，而游戏中的发送消息事件有选择角色、拖动鼠标、抛出武器和击中对手等事件。这些事件的发送方法都很类似，只是标识不同代表触发的事件不同。

```
public void mouseClicked(MouseEvent e) {
    …
    clientThread.write.println("&onDesk&Dongqiang");
    clientThread.write.println(i + "\nORC");
    }
    …
```

客户端消息的接收与服务器类似，都是在不停查询指定端口是否有数据。

```
public void run() {
   String m;
   while (true) {
     try {
       m = read.readLine();
//接收服务器当前有多少人在线
       if (m.equals("&hallcount&")) {
         chatHall.chatCout.setText("聊天大厅(" + read.readLine() + "人)");
       }
```

2．抽象测试套设计

本节主要介绍抽象测试套的设计，这里只介绍登录过程的抽象测试套设计，其他测试套设计与此相同。

(1)首先根据游戏交互的数据格式设计其数据类型并进行定义：

```
type record of charstring payload;//自定义类型 payload,为 record of 类型的
                                  //值，其中数据均为 charstring 类型
type record request{              //定义 record 数据类型，名字为 request
    charstring clientnum,         //charstring 类型的客户端号声明
    charstring portnum,           //charstring 类型的端口号声明
    payload content               //payload 类型的 content 声明
};
type record response{ payload content};
```

(2)根据自定义的数据类型，定义测试数据模板用来描述测试输入和测试期望的结果：

```
template response response_001:= {
    content := {"conDeskcover","4","ORC","SUYULAN"}
}
template request Request_chose_001_1:= {
    clientnum := "client1",
    portnum := "6001",
    content := {"ORC","4","&StartFrame&"}
}
template request Request_chose_001_2:= {
    clientnum := "client1",
```

```
    portnum := "6000",
    content := {"&onDesk&Dongqiang","4","ORC"}
}
```

(3) 定义测试配置用来描述通信端口类型和通信组件类型：

```
type port PortType message {
    out    request;
    in     response;
}
type component SystemType {
    port PortType systemPort1;
    port PortType systemPort2;
}
type component  mtcType{
    port PortType mtcPort1;
    port PortType mtcPort2;
    timer localTimer := 60.0;
}
```

(4) 依据客户端与服务器端的交互方式，创建测试用例：

```
testcase Wartest_LoginN1()  runs on mtcType  system SystemType {
    map(mtc:mtcPort1, system:systemPort1);
    map(mtc:mtcPort2, system:systemPort2);
    mtcPort1.send(Request_chose_001_1);
    mtcPort2.send(Request_chose_001_2);
    localTimer.start;
    alt {
        []mtcPort2.receive(response_001){
            localTimer.stop;
            setverdict(pass);
        }
        []localTimer.timeout{
            setverdict(fail);
        }
    }
}
```

(5) 在控制部分描述测试用例的执行：

```
control {
    execute(Wartest_LoginN1());
}
```

3．编码解码器设计

编码解码器的编写需要根据抽象测试套的数据格式进行定义，魔兽网络游戏测试套数据格式设计，request 数据类型是发送的请求数据类型，在测试套中作为 send() 方法的参数，response 数据类型是从服务器接收的响应数据类型，在测试套中作为 receive() 方法的参数类型。

```
type record of charstring payload;
type record request{
        charstring clientnum,
        charstring portnum,
        payload content
    };
type record response{ payload content};
```

编码解码器需要实现 TciCDProvided 接口中定义的 encode()方法和 decode()方法，该接口定义在 TTCN-3 控制接口规范中，TTman 开源包(TTman 实现了 TTCN-3 的基础运行时环境)中提供了这两个方法的基本实现。encode()方法主要实现将 TTCN-3 数据类型转换为字符流，decode()方法是将接收到的消息转换成 TTCN-3 测试系统可以接收的类型。

编码方法 encode()过程描述：首先，从抽象测试套中获取 send()方法中的参数，读取其数据类型，本例中为 request 数据类型，根据参数数据类型获取类型定义，本例中为 record 数据类型，record 数据类型为 TTCN-3 数据类型。将其编码为 RecordValue 类型并获取其子类各域名放入字符数组中；其子域分别为两个 charstring 类型和一个用户定义类型 payload 类型，继续追溯 payload 为 record of 类型，每个数据又为 charstring 类型。对找到的 TTCN-3 基本类型分别进行编码，并找到 record 类型的子域依次编码。编码过程的伪代码如图 10-13 所示。

```
Value 类型的对象转换为 RecordValue 类型的对象；
通过 RecordValue 对象得到各域名并存放在字符串数组中；
第一域值赋给 CharstringValue 对象；
通过 CharstringValue 对象得到第一域值字符串并添加到 StringBuffer 对象中；
第二域值赋给 CharstringValue 对象；
通过 CharstringValue 对象得到第一域值字符串并添加到 StringBuffer 对象中；
第三域值赋给 RecordofValue 对象；
For(int i=0；i<RecordofValue 对象的域长；i++)
各域值赋给 CharstringValue 对象；
通过 CharstringValue 对象得到各域值字符串并添加到 StringBuffer 对象中；
通过 StringBuffer 对象得到字节流并创建 TriMessage 实例；
返回 TriMessage 实例；
```

图 10-13　encode()方法实现伪代码

解码方法 decode()过程可看成编码过程的逆过程。首先读取测试套文件，在测试例描述部分获取 receive()方法中的参数，读取其数据类型，并保存需要接收到的数据。通过保存的数据查找数据模板和数据模板的用户自定义类型，最终进行转换操作，所做操作与编码相似。

4．适配器设计

被测系统适配器负责测试控制执行，负责和测试系统的通信等功能。适配器必须同时了解测试系统和被测系统才能完成工作。在 TTman 开源包中提供了一个基类——TestAdapter 类，该类是一个线程类，实现了 TriCommunicationSA、TriPlatformPA 和 TciEncoding 接口。在本案例适配器的实现中需要实现这三个接口，主要有以下几个方法需要实现。

(1) getCodec()方法：负责返回一个在发送和接收消息时使用的编码解码器对象。

(2) triExecuteTestcase()方法：负责测试用例的执行。在测试用例执行前由 TTCN-3 运行环境调用该方法。返回结果为该操作的状态。成功返回 TRI_OK，失败返回 TRI_Error。

(3) triSend()方法：负责发送测试数据到网络游戏服务器端。网络游戏软件的服务器端通过 TSI(Test System Interface)端口接收测试数据。该方法是适配器的主要实现方法。

(4) triEnqueueMsg()方法：负责从网络游戏软件的服务器端接收响应数据。接收到的响应数据通过 TSl 端口传送给 TTCN-3 运行环境。

(5) triMap()方法和 triUnmap()方法：当有多个测试组件映射到一个 TSl 端口时，为接收到的测试数据指定正确的目的测试组件。

魔兽网络游戏客户端和服务器端通过 socket 编程实现通信。一个客户端通过两个端口和服务器端通信，分别是 6000 端口和 6001 端口。6000 端口负责游戏大厅的消息交互，6001 端口负责游戏消息交互。如果进行游戏，最少需要启动两个客户端。也就是说在适配器中需要判别客户端和端口号，以便进行对应。在通过端口 6000 和 6001 进行通信时，又分别需要两个 socket，所以在适配器实现中设置了四个 socke 变量、两个端口变量，对应每一个 socket 都有一个 PrintStream 和 BufferedReader，用来发送消息和存放接收到的消息。

下面主要介绍 triSend()方法的实现。

在 triSend()方法中，需要根据判断客户端号和端口号判断是哪个客户端发送的消息类型。还需要在 triSend()方法中实现 receiveThread 线程来接收服务器段返回的消息，并将其传送给可执行测试套。在 receiveThread 线程的实现中，同样需要根据端口号和客户端号判断通过哪个 socket 接收消息。triSend()方法处理逻辑如下所示：

```
public TriStatus triSend(final TriComponentId componentId, final
TriPortId siPortId,TriAddress address,TriMessage sendMessage) {
获取测试成分和端口号
判断 tsiPortId 是哪个系统端口，并将 tsiPortId 赋值给该系统端口
try {
    byte[] mesg = sendMessage.getEncodedMessage();
    String message = new String(mesg);
    StringTokenizer st = new StringTokenizer(message, "\n");
    获取客户端和端口号，分别保存在 clientnum 和 portnum 中
    String sName = InetAddress.getLocalHost().getHostAddress();
    如果是客户端 1 发送的消息 {
    如果是 6000 端口 {
        port = 6000;//记录端口号
        如果客户端 1 对应 6000 端口的 socket 为空 {
            建立对应 socket 连接
        }
    }
    else {
    如果是 6001 端口{
        port = 6001;
          如果客户端 1 对应 6001 端口的 socket 为空  {
            建立对应 socket 连接
        }
    }
    else {
        如果既不是 6000 端口也不是 6001 端口，则错误!
        }
    }
}
    对客户端 2 做与客户端 1 同样的处理
```

```
        String messg = message.substring(cl + pl + 2, message.length() - 1);
        StringBuffer sb = new StringBuffer();
        如果是客户端 1 发送的消息 {
            component3=componentId; //保存测试成分
            如果是从 6000 端口发送的消息 {
                for (int i = 0; i < messg.length(); i++) {
                    sb.append(messg.charAt(i));
                }
                write11.println(sb); //向服务器发送消息
        }
        else {
            如果是从 6001 端口发送的消息 {
                for (int i = 0; i < messg.length(); i++) {
                    sb.append(messg.charAt(i));
                }
                write12.println(sb);
            }
        }
    }
```

对从客户端 2 发送消息做同样的处理，发送消息部分即已经完成。

以下是接收消息的线程实现：

```
    Thread  receiverThread = new Thread(){
    public void run() {
    try {
        如果客户端 1 对应 6000 端口的 socket 为空{
            不做任何操作
        }
        else {
            如果是通过客户端 1 的 6000 端口发送的消息{
                则服务器返回的消息加入到客户端 1 对应的 6000 端口的接收消息队列中
            }
        }
    如果客户端 1 对应 6001 端口的 socket 为空{
        不做任何操作
    }
    else {
        如果是通过客户端 1 的 6001 端口发送的消息 {
            如果客户端 2 的 6001 对应的 socket 为空{
                则服务器返回的消息加入到客户端 1 对应的 6001 端口的接收消息队列中
            }
        else {
            则服务器返回的消息加入到客户端 2 对应的 6001 端口的接收消息队列中
        }
      }
    }
        对客户端 2 对应的 socket 做同样的处理，完成接收消息队列的存放
      } catch (Exception ex) {
```

```
        处理异常
        }
    }
    };
        启动接收线程
    }捕获并处理异常
        return new TriStatusImpl();
    }
```

5．测试执行和结果

在测试平台中打开 TTCN-3 核心语言测试套 Login.ttcn3，单击工具栏中的“编译”按钮，把该测试套编译成可执行测试套 Login.mlf，并自动保存在抽象测试套所在文件夹中。打开 TTCN-3 执行管理界面，加载 Login.mlf 文件及对应的适配器，启动魔兽游戏服务器，即可执行测试例，进行测试。

单击执行测试例后，游戏服务器界面如图 10-14 所示。

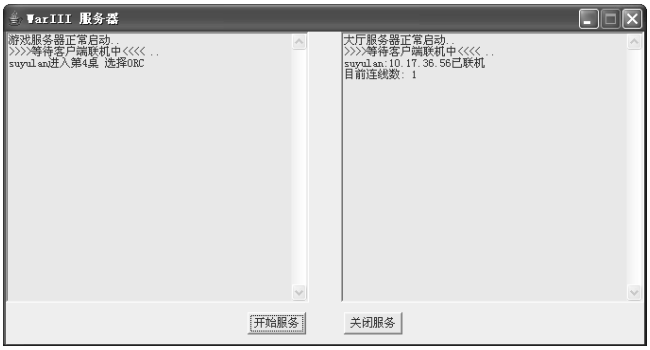


图 10-14　魔兽游戏服务器界面

测试平台中的测试结果如图 10-15 所示：在属性视图中显示判断为 pass，在数据视图中显示期望数据和实际数据相同。

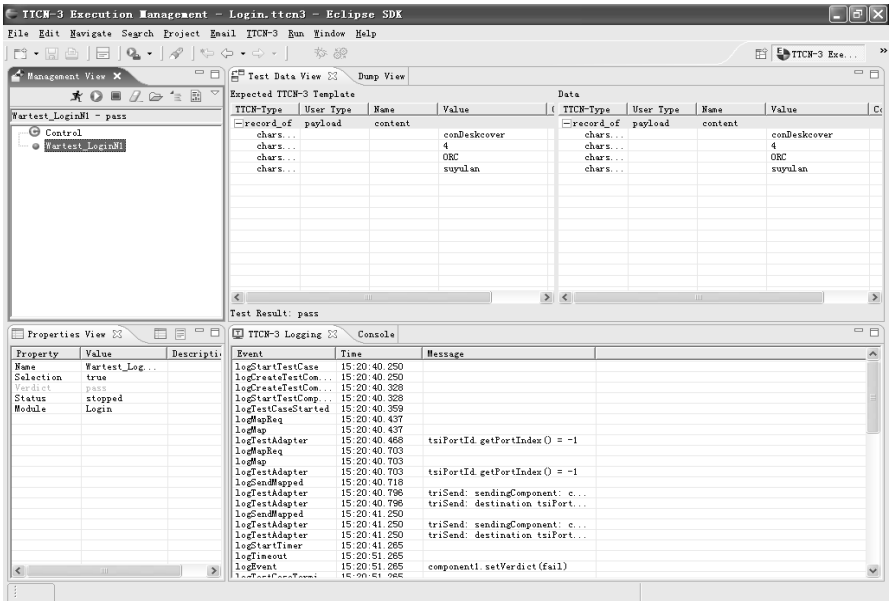


图 10-15　魔兽游戏登录过程测试结果

10.5　水果机游戏测试

1．水果机游戏简介

FruitMachine 手机网络游戏是由 Nokia 开发的开源网络游戏。FruitMachine 手机网络游戏使用 C/S 结构开发，客户端通过 JZME 实现，可运行在所有支持 MDP1.0 的手机上；服务器端采用 Javaservlet 技术实现。客端提供了一个用户接口用于玩家通过手机登录服务器并运行游戏。服务器端运行在支持 HTTP servlet 容器的 Tomcat 上，它负责维护玩家的账户数据库，并且根据客户端发送的请求返回游戏的运行结果。由于客户端是通过基于 HTTP 协议的移动应用协议与服务器端进行交互的，因此出于安全方面的考虑，FruitMachine 手机网络游戏的游戏逻辑设计在服务器端实现，并且对用户保持透明。

FruitMachine 手机网络游戏的设计实现是由多个游戏逻辑模块所组成的。下面将说明如何针对游戏中“买币”逻辑模块进行测试。“买币”逻辑模块的消息交互过程如图 10-16(a)所示。客户端把玩家所买钱币总数发送给服务器端。服务器端处理该请求进行购买交易，并相应的返回购买成功或购买错误的响应信息。

水果机游戏客户端从登录连接网络到完成买币的过程如图 10-16(b)所示。

(a)　　(b)

图 10-16　游戏过程

2．抽象测试套的设计

抽象测试套的设计和魔兽游戏抽象测试套的设计方法和步骤相同。这里介绍买币功能测试套的设计。

(1)首先根据游戏交互的数据格式设计其数据类型并进行定义：

```
type record of charstring payload; //自定义类型 payload,为 record of 类型的
                                   //值，其中数据均为 charstring 类型
type record request {//定义 record 数据类型，名字为 request,发送请求的数据格式
    charstring url,          //charstring 类型的 url
    charstring command,      //charstring 类型的 command
    payload content          //payload 类型的 content
}
```

```
type record response {      //接收请求的数据格式
    charstring command,
    payload content
}
```

(2)根据自定义的数据类型，定义测试数据模板用来描述测试输入和测试期望的结果：

```
template request LoginRequest_001 := {
    url := "http://localhost:8080/FruitMachineServlet/game",
    command := "login",
    content := {"ssc", "111"}
}
template response LoginResponse_001 := {
    command := "login-OK",
    content := {"50"}
}
template request BuyRequest_001 := {
    url := "http://localhost:8080/FruitMachineServlet/game",
    command := "buy",
    content := {"10"}
}
template response BuyResponse_001 := {
    command := "buy-OK",
    content := {"60"}
}
template request LogoutRequest_001 := {
    url := "http://localhost:8080/FruitMachineServlet/game",
    command := "logout",
    content := {""}
}
template response LogoutResponse_001 := {
    command := "logout-OK",
    content := ?
}
```

(3)定义测试配置用来描述通信端口类型和通信组件类型：

```
type port httpPortType message {
    out request;
    in response;
}
type component systemType {
    port httpPortType systemPort;
}
type component mtcType {
    port httpPortType mtcPort;
    timer localTimer := 10.0;
}
```

(4)依据客户端与服务器端的交互方式，创建测试用例：

```
testcase TC_Login_001() runs on mtcType system systemType {
    map(mtc:mtcPort, system:systemPort);
```

```
        mtcPort.send(LoginRequest_001);
        localTimer.start;
        alt {
            [] mtcPort.receive(LoginResponse_001) {
                localTimer.stop;
                setverdict(pass);
            }
            [] mtcPort.receive {
                localTimer.stop;
                setverdict(fail);
            }
            [] localTimer.timeout {
                setverdict(fail);
            }
        }
        all component.done;
    }
    testcase TC_Buy_001() runs on mtcType system systemType {
        map(mtc:mtcPort, system:systemPort);
        mtcPort.send(BuyRequest_001);
        localTimer.start;
        alt {
            []mtcPort.receive(BuyResponse_001) {
                localTimer.stop;
                setverdict(pass);
            }
            []mtcPort.receive {
                localTimer.stop;
                setverdict(fail);
            }
            []localTimer.timeout {
                setverdict(fail);
            }
        }
        all component.done;
    }
    testcase TC_Logout_001() runs on mtcType system systemType {
    map(mtc:mtcPort, system:systemPort);
    mtcPort.send(LogoutRequest_001);
    localTimer.start;
    alt {
        []mtcPort.receive(LogoutResponse_001) {
            localTimer.stop;
            setverdict(pass);
        }
        []mtcPort.receive {
            localTimer.stop;
```

```
            setverdict(fail);
        }
        []localTimer.timeout {
            setverdict(fail);
        }
    }
    all component.done;
}
```

(5)在控制部分描述测试用例的执行：

```
control {
    execute(TC_Login_001());
    execute(TC_Buy_001());
    execute(TC_Logout_001());
}
```

3．编码解码器和被测系统适配器设计

水果机游戏编码解码器和被测系统适配器的设计和魔兽网络游戏编码解码器与适配器设计方法相同，不同之处在于需要根据水果机游戏的交互数据类型和数据发送与接收方法进行设计其编解码过程与数据发送方式。

4．测试执行和结果

测试执行过程和魔兽游戏的测试执行过程相同，加载可执行测试套和适配器。

登录测试例的执行结果如图 10-17 所示。

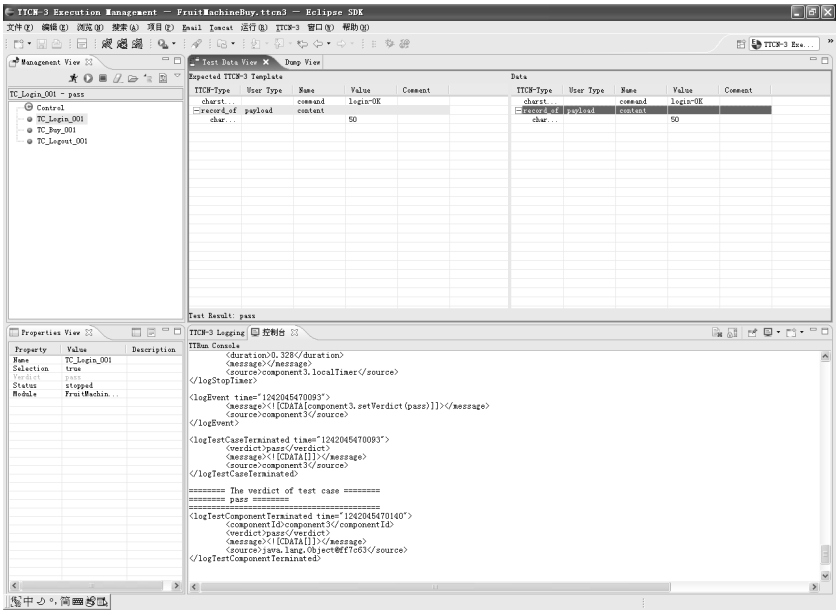


图 10-17　登录测试例的执行结果

买币测试例的执行结果如图 10-18 所示。

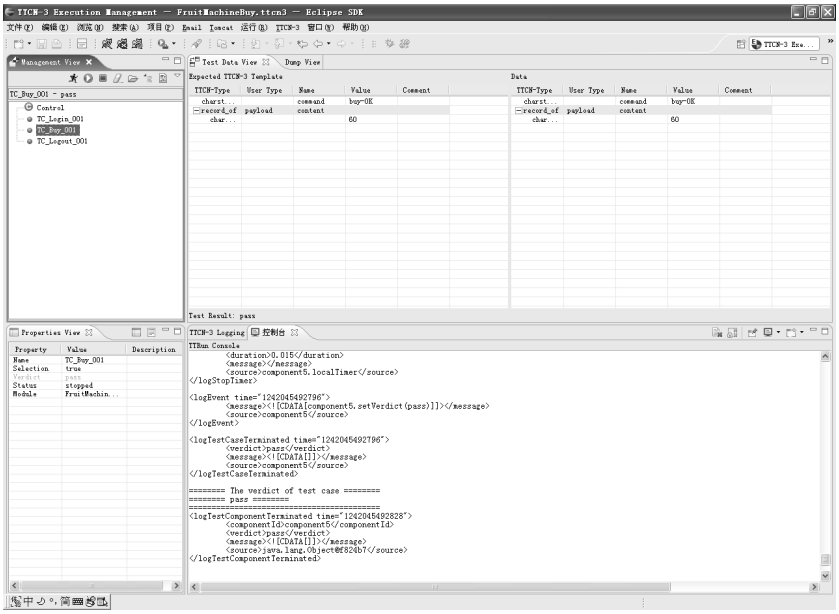


图 10-18　买币测试例的执行结果

退出登录测试例的执行结果如图 10-19 所示。

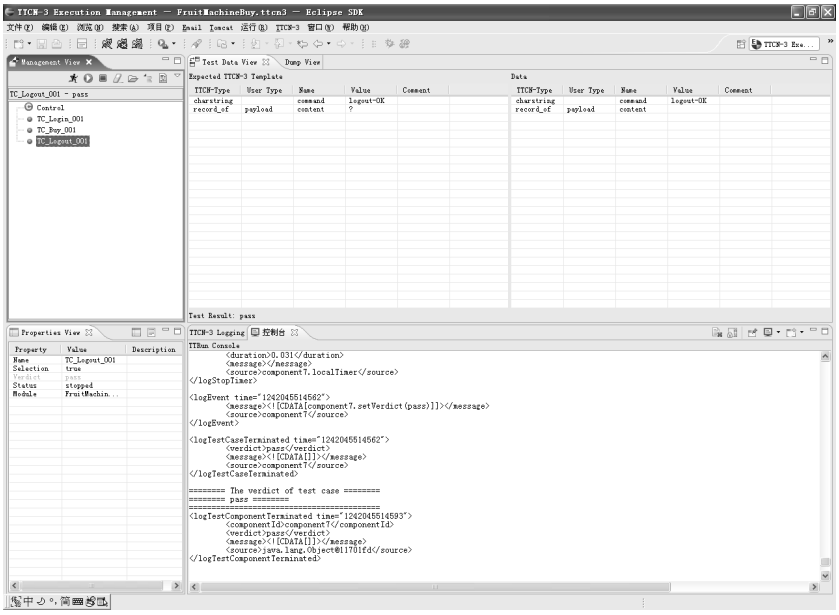


图 10-19　退出登录测试例的执行结果

10.6　即时通信软件测试案例分析

1．被测系统简介

该被测系统是一个基于 SIP 协议开发的即时通信软件，采用了 JAIN SIP 技术。JAIN SIP 是用 Java 语言实现的 SIP 类库，由于 JAIN SIP API 是对 SIP 标准的完整定义，所以任何基于 SIP 的应

用都可将 JAIN SIP API 作为 Java 标准界面，用到任何 SIP 应用实例中。JAIN SIP 技术在应用方面也十分广泛，可以实现应用服务器、SIP 电话、网关及网关控制器、SIP 服务器、基于 SIP 的业务、SIP 计费解决方案、开发工具包、SIP 测试工具、SIP 用户代理以及 SIP 网络管理。

本文中的这个即时通信软件实现了用户与用户间的相互通信，如图 10-20 所示。

“From”文本框中显示的是呼叫方地址，“To”文本框中显示的是被呼叫方地址。“Send Message”文本框中显示的是要发送的信息。当呼叫方将信息全部写好后，单击“send”按钮发送消息。如果发送成功，在“Received Messages”文本框中将显示“Sent”已发送信息。被呼叫方接收到消息后，将在“Received Messages”文本框中显示呼叫一方的地址及消息信息，如图 10-21 所示。

图 10-20　通信软件界面

图 10-21　消息接收

2．抽象测试套设计

抽象测试套是测试执行的关键依据，而软件的功能实现及协议要求又是开发测试套的依据。本文例子中的实例可以实现用户定位并与所定位的用户通信，针对该功能，我们设计抽象测试套。

首先，定义消息用到的数据类型。在 SIP 协议中，消息分为两类——请求消息和响应消息。请求消息包括三个部分：请求行、消息包头、消息体部分。在请求行中定义了方法类别、请求地址以及 SIP 协议的版本号。本实例中用到的方法为 MESSAGE 方法，地址为 10.17.36.2：5062，SIP 版本号为 2.0。在消息包头部分，SIP 协议中共定义了 43 种头域类型，分别是：Accept，AcceptEncoding，AcceptLanguage，AlertInfo，Allow，AuthenticationInfo，CallId，CallInfo，Contact，ContentDisposition，ContentEncoding，ContentLanguage，ContentLength，ContentType，CSeq，Date，ErrorInfo，Expires，From，InReplyTo，MaxForwards，MimeVersion，MinExpires，Organization，Priority，ProxyAuthenticate，ProxyAuthorization，ProxyRequire，RecordRoute，ReplyTo，Require，RetryAfter，Route，Server，Subject，Supported，Timestamp，To，Unsupported，UserAgent，Via，Warning，WwwAuthenticate。在消息体中定义了有关音频和视频传输的内容，由于本实例中不涉及此项内容，所以该项中的值为空。

根据以上说明，请求消息的抽象数据类型如下：

```
type record request {
        charstring from_username,
        charstring to_username,
        charstring from_ip,
        charstring to_ip,
        charstring from_port,
        charstring to_port,
```

```
        charstring msg,
        charstring method,
        charstring accept,
        charstring accept_Encoding,
        charstring accept_Language,
        charstring alert_Info,
        charstring allow,
        charstring authentication_Info,
        charstring authorization,
        charstring call_ID,
        charstring call_Info,
        charstring contact,
        charstring content_Disposition,
        charstring content_Encoding,
        charstring content_Language,
        charstring content_Length,
        charstring content_Type,
        charstring cseq,
        charstring date,
        charstring error_Info,
        charstring expires,
        charstring fromHeader,
        charstring in Reply To,
        charstring max Forwards,
        charstring min Expires,
        charstring mime Version,
        charstring organization,
        charstring priority,
        charstring proxy Authenticate,
        charstring proxy_Authorization,
        charstring proxy_Require,
        charstring record_Route,
        charstring reply_To,
        charstring require,
        charstring retry_After,
        charstring route,
        charstring server,
        charstring subject,
        charstring supported,
        charstring timestamp,
        charstring toHeader,
        charstring unsupported,
        charstring user_Agent,
        charstring via,
        charstring warning,
        charstring www_Authenticate
    }
```

接下来介绍响应消息，SIP 协议的响应消息包括了协议版本号、响应码及响应说明。以此实例为例，协议版本号依然为 2.0，如果两个用户间可以正常接收到消息，则响应码为 200，响应说明为 OK。

根据以上说明，响应消息的抽象数据类型如下：

```
type record response {
        charstring ResponseMessage
}
```

当抽象数据类型的结构定义好后，就要开始根据自定义的数据类型，定义测试数据模板用来描述测试输入和测试期望的结果。根据上文中的说明，请求消息的数据模板如下：

```
template request Request_001 := {
        from_username := "aaa",
        to_username := "bbb",
        from_ip := "10.17.36.200",
        to_ip := "10.17.36.200",
        from_port := "5061",
        to_port := "5062",
        msg := "hello",
        method := "MESSAGE",//
        accept := "NULL",
        accept_Encoding := "NULL",
        accept_Language := "NULL",
        alert_Info := "NULL",
        allow := "NULL",
        authentication_Info := "NULL",
        authorization := "NULL",
        call_ID := "NULL",
        call_Info := "NULL",
        contact := "NULL",
        content_Disposition := "NULL",
        content_Encoding := "NULL",
        content_Language := "NULL",
        content_Length := "NULL",
        content_Type := "NULL",
        cseq := "NULL",
        date := "NULL",
        error_Info := "NULL",
        expires := "NULL",
        fromHeader := "NULL",
        in_Reply_To := "NULL",
        max_Forwards := "NULL",
        min_Expires := "NULL",
        mime_Version := "NULL",
        organization := "NULL",
        priority := "NULL",
        proxy_Authenticate := "NULL",
        proxy_Authorization := "NULL",
        proxy_Require := "NULL",
        record_Route := "NULL",
        reply_To := "NULL",
```

```
        require := "NULL",
        retry_After := "NULL",
        route := "NULL",
        server := "NULL",
        subject := "NULL",
        supported := "NULL",
        timestamp := "NULL",
        toHeader := "NULL",
        unsupported := "NULL",
        user_Agent := "NULL",
        via := "NULL",
        warning := "NULL",
        www_Authenticate := "NULL"
}
```

响应消息的数据模板如下：

```
template response Response_001 := {
        ResponseMessage := "200"
}
```

定义测试配置用来描述通信端口类型和通信组件类型：

```
type port SIPPortType message {          //基于消息的通信端口定义
        in Request;                      //发送请求
        out Response;                    //接收回应
}
type component systemType {              //定义系统通信组件
        port SIPPortType systemPort;     //系统通信组件包含的端口
}
type component mtcType {                 //定义主测试通信组件
        port SIPPortType mtcPort;        //主测试组件包含的端口
        timer localTimer := 5.0;         //定时器
}
```

依据客户端与服务器端的交互方式，编写测试例及测试验证过程：

```
testcase TC_001() runs on mtcType system systemType {
        map(mtc:mtcPort, system:systemPort);    //端口映射
        mtcPort.send(Request_001);              //发送请求消息模板
        localTimer.start;                       //开启定时器
        alt {
            [] mtcPort.receive(Response_001) {  //接收响应消息模板
                localTimer.stop;                //停止定时器
                setverdict(pass);               //验证通过
            }
            [] mtcPort.receive {                //接收其他响应数据
                localTimer.stop;                //停止定时器
                setverdict(fail);               //验证失败
            }
```

```
                [] localTimer.timeout {                    //定时器到时
                    setverdict(fail);                      //验证失败
                }
            }
            all component.done;                            //关闭所有的通信组件
        }
```

最后，在控制部分描述测试用例的执行：

```
    control {
            execute(TC_ 001());                            //执行测试用例
    }
```

3．编码解码器设计

具体实现上，编码解码器需要实现 TTCN-3 控制接口规范定义的 TciCDProvided 接口的 encode()方法和 decode()方法，它们分别负责发送数据的编码和接收数据的解码。TTman 开源包中提供了一个 AbstractBaseCodec 抽象基类，它提供了 encode()方法和 decode()方法的基本实现。开发者可以扩展或重写不同的实现方法以满足其需求。编码操作的程序流程图如图 10-22 所示。

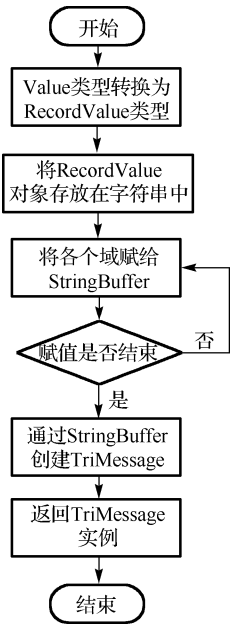


图 10-22　编码操作的程序流程图

编码操作需实现 encode()方法的具体算法描述如下：

将 Value 类型的对象转换为 RecordValue 类型的对象；
通过 RecordValue 对象得到各域名并存放在字符串数组中；
For(int i=0；i<RecordofValue 对象的域长；i++)
各域值赋给 CharstringValue 对象；
通过 CharstringValue 对象得到各域值字符串并添加到 StringBuffer 对象中；
通过 StringBuffer 对象得到字节流并创建 TriMessage 实例；
返回 TriMessage 实例；

解码操作的程序流程图如图 10-23 所示。

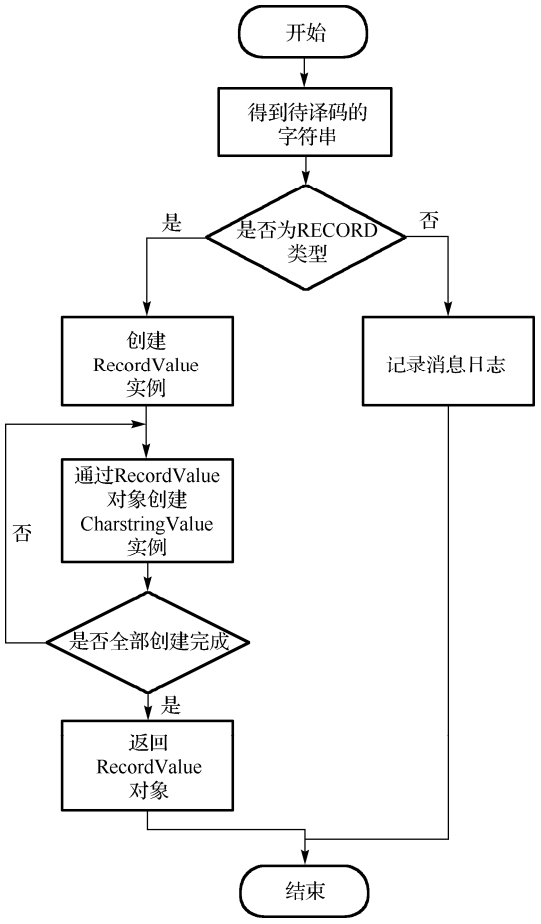


图 10-23　解码操作的程序流程图

解码操作需实现 decode()方法的具体算法描述如图 10-24 所示。

```
得到待解码的字符串；
If(type.getTypeClass()等于 TciTypeClass.RECORD)
创建 RecordValue 新实例；
通过 RecordValue 对象得到各域名并存放在字符串数组中；
通过 RecordValue 对象得到头域并创建 CharstringValue 新实例；
设置 CharstringValue 对象的字符串；
设置 RecordValue 对象的头域；
通过 RecordValue 对象得到尾域并创建 RecordofValue 新实例；
For(int i=0；i<RecordofValue 对象的域长；i++)
通过 RecordofValue 对象得到域元素类型并创建 CharstringValue 新实例；
设置 CharstringValue 对象的字符串；
将 CharstringValue 对象添加到 RecordofValue 对象的域中；
设置 RecordValue 对象的尾域；
返回 RecordValue 对象；
Else
记录消息日志；
返回空；
```

图 10-24　decode()方法的具体算法

4．系统适配器设计

一般来说，被测系统适配器需要实现 TTCN-3 运行时接口规范定义的 TriCommunicationSA 接口和 TriPlatformPA 接口的方法，然而通过分析 SIP 应用软件客户端与服务器端的交互方式，在 SIP 应用软件测试中，需要实现的 TTCN-3 运行时接口操作的最小集合包括以下五个函数方法。

- triExecuteTestCase()负责执行测试用例。该函数在测试用例执行前由 TTCN-3 运行环境调用。
- triSend()负责发送测试数据到 SIP 应用软件的服务器端。SIP 应用软件的服务器端通过 TSI(Test System Interface)端口接收测试数据。
- triEnqueueMsg()负责从 SIP 应用软件的服务器端接收响应数据。接收到的响应数据通过 TSI 端口传送给 TTCN-3 运行环境。
- triMap()和 triUnmap()负责当有多个测试组件映像到一个 TSI 端口时，为接收到的测试数据指定正确的目的测试组件。

适配器中负责消息通信的 triSend()方法的程序设计流程图如图 10-25 所示。

适配器中负责消息通信的 triSend()方法的具体实现的算法描述如图 10-26 所示。

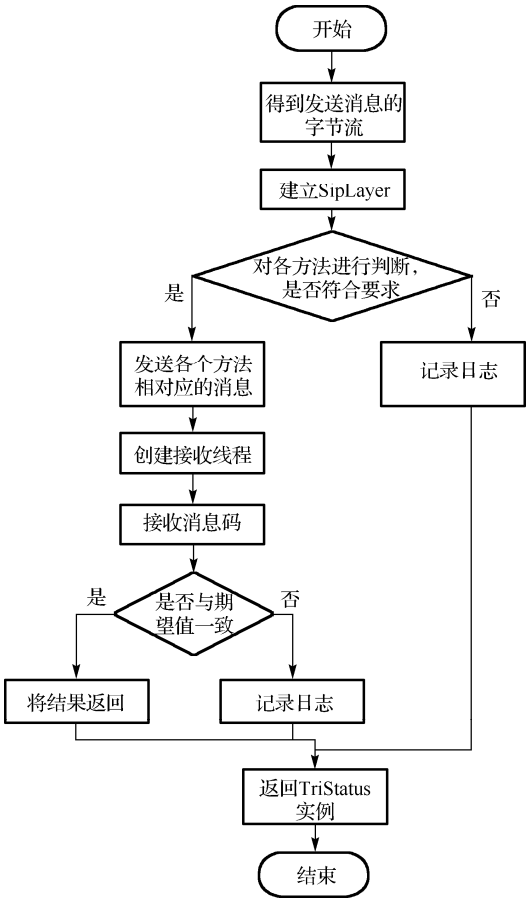


图 10-25 triSend()方法的程序设计流程图

```
得到发送消息的字节流；
将字节流转换为请求字符串；
将请求字符串各域值分别存入相对应的字符串中；
建立 SipLayer；
判断所要用到的 SIP 方法；
If(方法 = INVITE)
    发送 INVITE 方法相对应的 SIP 消息；
If(方法 = ACK)
    发送 ACK 方法相对应的 SIP 消息；
If(方法 = OPTIONS)
    发送 OPTIONS 方法相对应的 SIP 消息；
If(方法 = BYE)
    发送 BYE 方法相对应的 SIP 消息；
If(方法 = CANCEL)
    发送 CANCEL 方法相对应的 SIP 消息；
If(方法 = REGISTER)
    发送 REGISTER 方法相对应的 SIP 消息；
If(方法 = MESSAGE)
    发送 MESSAGE 方法相对应的 SIP 消息；
创建接收线程；
记录收到的消息状态码；
将收到的消息状态码与期望的消息状态码相比较；
If(收到的消息状态码 = 期望的消息状态码)
    得到输入流；
    通过输入流读取响应字符串；
    通过响应字符串创建 TriMessage 对象；
    将 TriMessage 对象传送给 TTCN-3 执行环境；
    关闭输入流；
    断开连接；
Else
    记录消息日志；
开启接收线程；
关闭输出流；
返回 TriStatus 实例；
```

图 10-26 triSend()方法的具体实现的算法描述

5．测试执行及结果

在测试中，首先载入生成好的可执行测试套，然后加载系统适配器，最后单击“运行”按钮。此时，系统中会显示执行过程及执行结果，如图 10-27 所示。

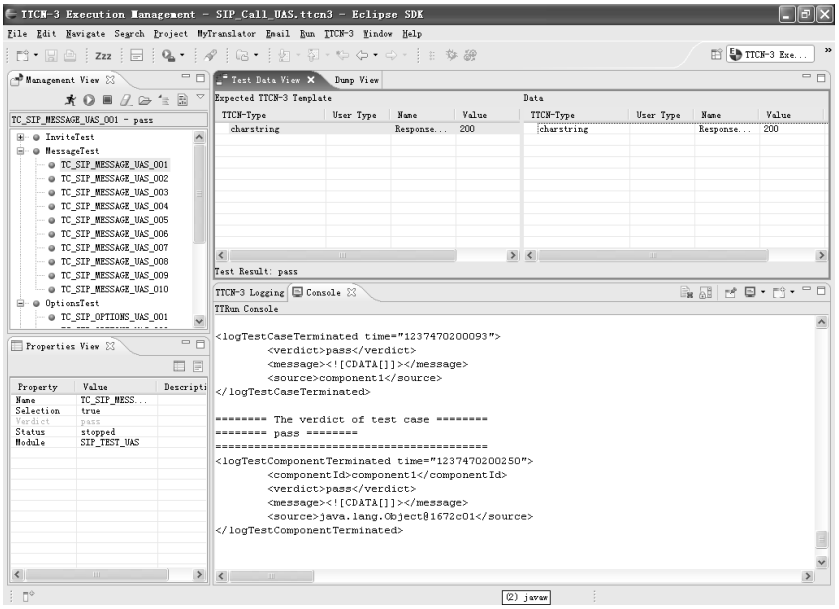


图 10-27　执行结果

我们期望收到 200 OK 的响应码，实际收到的是 200 OK 响应码，因此给出 pass 的判决。

10.7　QQ 是否在线测试

www.webxml.com.cn 网站发布了检测 QQ 是否在线的服务，该服务提供 qqCheckOnline 操作，通过输入字符串类型的 QQ 号码，返回检测结果。Y 表示在线，N 表示离线。本实验采用契约优先方法开发检测 QQ 是否在线的仿真 Web 服务，步骤如下：

⑴ 使用 Apache Axis2 提供的 wsdl2java.bat 工具，创建服务的框架。

⑵ 在 QqOnlineWebServiceSkeleton.java 中，添加简单的业务逻辑实现代码，直接返回字符串“Y”。

由于该服务是基于 SOAP 协议的，所以本节先对 SOAP 协议进行介绍，然后才介绍具体的测试流程。

1．SOAP 协议

在基于 SOAP 协议的 webservice 中，Web 服务的使用者和 Web 服务提供者需要通过 SOAP 提供的消息机制来进行交互。SOAP 采用 XML 技术定义消息处理框架，该框架是一种可以通过多种底层协议进行交换的 XML 消息结构，而且是可扩展的。SOAP 消息具有三个主要特征：可扩展性、底层传输协议无关性和编程模型独立性。

SOAP 的结构如下：

```
<?xml version="1.0" encoding="utf-8"?>
    <env:Envelope xmlns:env=http://schemas.xmlsoap.org/soap/envelope/
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
< env:Header>
</ env:Header>
< env:Body>
```

```
        <qqCheckOnline xmlns="http://WebXml.com.cn/">
            <qqCode>123456789</qqCode>
        </qqCheckOnline>
    </ env:Body>
    </ env:Envelope>
```

其中，Envelope 元素是 SOAP 消息的根元素，该元素表明 XML 文档是 SOAP 消息。应用程序解读到这一部分时，就可以确定<env:Envelope></ env:Envelope>之间的是 SOAP 消息，并可通过命名空间确定 SOAP 版本。http://schemas.xmlsoap.org/soap /envelope/是 SOAP1.1 版本的命名空间，http://www.w3.org/2003/05/soap-envelope 是 SOAP1.2的命名空间。

在 SOAP 消息中，Heade 元素是可选的，且 SOAP 协议没有限定 Header 中必须存放的内容，来自任何除 SOAP 命名空间外的命名空间的任意数量的元素都可以放入 Head 中。Header 元素的这个特性正是 SOAP 协议可扩展性的表现。

SOAP 消息的实际数据存储在 Body 元素中，具体内容根据不同应用会有不同。在本例中，qqCheckOnline 请求被发送到接收方，请求的内容是 qqCode 的值 123456789。

SOAP 消息的传输不依赖于特定的底层传输协议。如图 10-28 所示，SOAP 发送者先后使用了 HTTP 协议、TCP 协议、MSMQ 协议和 SMTP 协议传输 SOAP 消息，到达 SOAP 消息接收者。

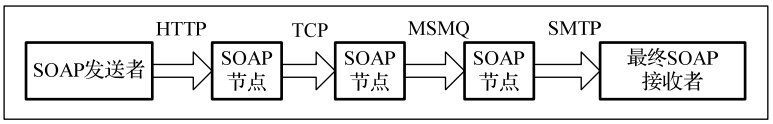


图 10-28　SOAP 消息的传递

2．抽象测试套

通过分析 SOAP 消息的结构，可知 SOAP 消息中必要的信息包括操作名、参数、命名空间。另外，在发送 SOAP 消息时需要知道目标地址。根据 SOAP 消息发送机制和 WSDL 规范，对 Web 服务测试中的数据类型进行如下定义：

```
type record paramNameList       //参数列表
{
        …
}
type record wsRequestType       //发送数据类型
{
        charstring operationName,
        charstring location,
        charstring namespace,
        paramNameList paramList
}
type record wsResponseType      //接收数据类型
{
        …
}
```

参数列表类型 paramNameList 是 record 的自定义类型，存放 SOAP 请求操作中携带的参数信息。

发送数据类型 wsRequestType 是 record 的自定义类型，存放发送到 Web 服务端所需的信息，包括操作名称、目标地址、命名空间和参数列表。

接收数据类型 wsResponseType 是 record 的自定义类型，存放从 Web 服务器返回的处理结果。

完整的抽象测试套编码见附录A。

3．编解码器

Web Service 的编解码器的作用是，将抽象测试套中的数据和 SOAP 消息进行相互转换。编解码器继承 com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec 类，实现 org.etsi.ttcn.tci.TciCDProvided 接口的 encode 方法和 decode 方法。encode 方法负责编码，decode 方法负责解码。

1) 编码部分

编码是将 wsResponseType 类型的 TTCN-3 数据转换为 SOAP 消息。编码过程主要分为两步：第一步是将 template 中的数据取出，存储在数据结构中；第二步是将数据结构中取出，通过 buildSOAP 函数，填充成 SOAP 消息。

创建 ParamList 类，用于存储参数列表信息(paramNameList 和 paramValueList)；创建 SoapBinding 类，用于存储构件 SOAP 消息所需的信息(operationName，location，namespace 和 paramList)。

wsResponseType 中包含四种数据类型，分别是 operationName、location、namespace 和 paramList。operationName、location 和 namespace 都是 charstring 类型数据，TTRun 平台提供对 TTCN-3 基本数据类型的编解码，这里不再赘述。paramList 是 record 类型，针对 record 类型进行编码的伪代码如下：

```
创建两个 ArrayList 类型变量 paramNameList 和 paramValueList
创建 ParamList 对象 paramList;
for(int i=0;i<paramList.length;i++){
    if(paramlList[i] is charstringType)
    {
        paramNameList.add(paramList[i]);
        paramValueList.add(paramList[i].getValue());
    }
    if(paramlList[i] is intType)
    {
        paramNameList.add(paramList[i]);
        paramValueList.add(paramList[i].getValue());
    }
    if(paramlList[i] is floatType)
    {
        paramNameList.add(paramList[i]);
        paramValueList.add(paramList[i].getValue());
    }
}
```

将参数名和参数值分别存入参数列表对象中。

将参数列表对象存入 SoapBinding 对象中。

将 template 中的数据取出，存入 SoapBinding 对象后，就可以进行 SOAP 消息的填充了。填

充 SOAP 消息时，使用了 JAXM 的 API 函数。JAXM(Java API for XML Messaging)是 Java 语言编写的 API 函数，支持以同步和异步的方式进行 SOAP 消息传递。

JAXM 的 API 包括两个主要的包：

(1)javax.xml.messaging，提供发送和接收 SOAP 消息的接口和类。

(2)javax.xml.soap，提供对 SOAP 消息的封装和解析的类。

从 SOAP 消息的根节点开始，逐层创建 SOAP 消息，过程如下：

(1)通过 SOAPConnectionFactory 接口创建 SOAPMessage。

(2)创建 SOAPPart。

(3)创建 SOAPEnvelope。

(4)向 SOAPEnvelope 中填充 Head 和 Body 部分。

2)解码部分

解码是将 SOAP 消息转换为 wsResponseType 类型的 TTCN-3 数据。

从适配器中接收到 Web Service 服务器响应的 SOAP 消息后，TTCN-3 测试系统将 SOAP 消息发送给解码器进行解码，解码的过程主要分为两步，首先从 SOAP 消息中获取数据，然后将这些数据填充为 wsResponseType 类型的模板，传送给 TTCN-3 测试平台。

handleSoapResponse 函数负责解析 SOAP 消息中的内容，SOAPMessage 类型中保存的是 SOAP 消息的全部内容，按照 XML 中的节点层次，逐层解析出 SOAPPart、SOAPEnvelope、SOAPBody、SOAPElement，最后通过遍历 SOAPElement，获取全部的 SOAP 消息中封装的服务器返回信息。

伪代码如下所示：

```
获取 SOAPMessage 的 Envelope 部分 responseEnvelope;
获取 Envelope 的 Body 部分 responseBody;
获取 Body 部分的响应函数元素节点 responseElement
获取 Body 部分的响应函数返回值元素节点 resultElement
获取 resultElement 节点中所有的返回值 dataElements
if(dataElements 有子元素个数大于 1)
{
    While(dataElements.hasNest)
    {
        将 dataElement 中的数据加入到 ArrayList 中;
    }
}
else
{
    dataElements 中的数据加入到 ArrayList 中;
}
```

获取 SOAP 消息中的数据信息之后，就可以将这些数据填充成 wsResponseType 类型的 template。关键代码如下：

```
创建 wsResponseType 类型变量 allFields;
for(int i=0;i<allFields.length;i++)
{
    获取 allFields 中第 i 个元素的类型名 typeName
    if(typeName 等于 charstring)
```

```
        {
            将 ArrayList 中第 i 个元素的值创建 charstring 类型对象;
            将该对象赋值给 allFields 的第 i 个元素
        }
    if(typeName 等于 int)
        {
            将 ArrayList 中第 i 个元素的值创建 int 类型对象;
            将该对象赋值给 allFields 的第 i 个元素
        }
        if(typeName 等于 float)
        {
            将 ArrayList 中第 i 个元素的值创建 float 类型对象;
            将该对象赋值给 allFields 的第 i 个元素
        }
    }
```

4．适配器

Web 服务的适配器模拟 Web 服务调用的过程，在 TTCN-3 测试平台和被测系统之间起到连接作用。适配器调用编解码器获取发送数据，并发送到 Web Service 的服务器端，然后从 Web 服务端接收数据，调用解码器，将解码后的数据传递给 TTCN-3 测试平台。

TRI 标准中对适配器的接口进行了详细的说明，主要需要实现 getCodec、triMap、triSend、enqueueMsg 方法。

getCodec 方法：通过抽象测试套中指明的编解码器名称，创建编解码器，作为返回值返回。

triMap 方法：成分端口与系统端口的映射。

triSend 方法：发送数据到被测系统。

enqueueMsg 方法：将从被测系统接收到的返回值加入到 TTCN-3 测试平台的接收消息队列中。

具体发送 SOAP 消息时，使用的是 HTTP 协议。创建并向目标地址发送 HTTP 协议请求，获取 Web 服务的请求结果 SOAP 消息。sendSOAP 函数提供发送 SOAP 消息的功能。发送 SOAP 消息的伪代码如下所示：

```
创建 URL 对象;
创建 HTTP 连接对象，设置 HTTP 连接的头元素;
用 HTTP 对象发送 SOAP 消息;
用 HTTP 对象接收 SOAP 消息，存入 BufferedReader 对象中;
while(BufferReader 的下一行不为空)
{
    取出该行的文本，append 到 StringBuffer 中;
}
断开 HTTP 连接
```

将 StringBuffer 类型转换为 ByteArray 类型，加入到 TTCN-3 测试系统的接收队列中。

5．测试结果

对某个 QQ 号进行测试，得到如图 10-29 所示的结果。

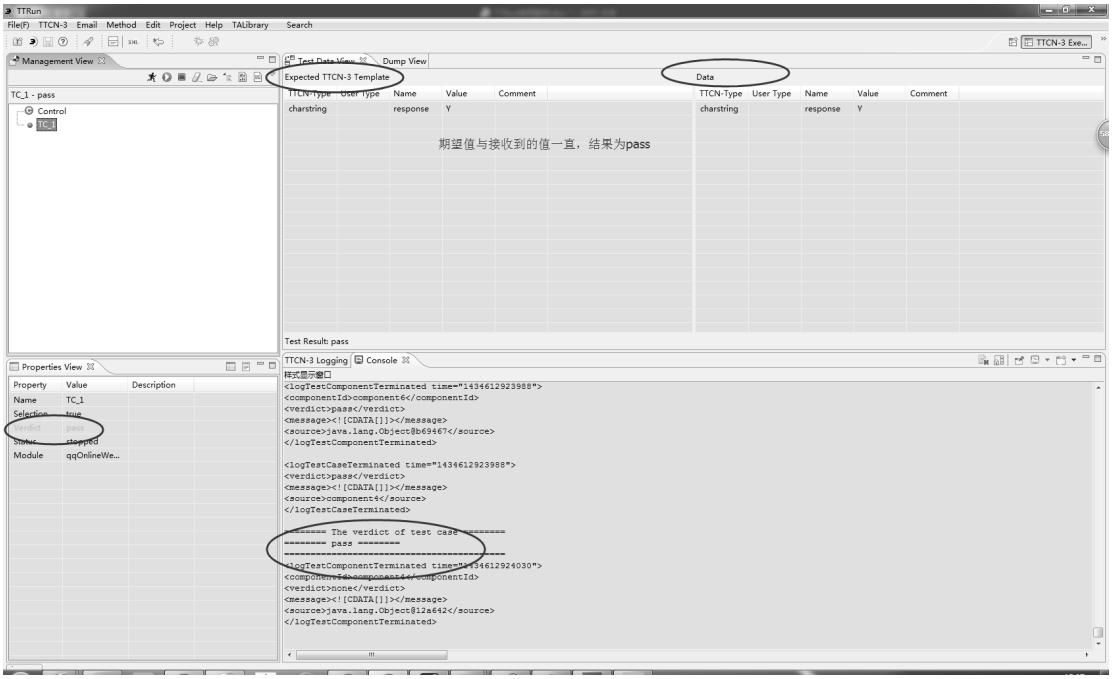


图 10-29　测试结果

10.8　Web 应用测试

1．被测系统介绍

实验室以前做过适配器库管理系统，在本案例中对该系统的登录和添加用户功能进行测试。下面对适配器库管理系统的被测部分进行介绍。系统管理员登录系统后可以进行添加用户等系列操作。下面对管理员登录和添加用户功能部分进行说明。

登录页面如图 10-30 所示。

图 10-30　登录页面

登录成功后进入登录成功页面，可以进行添加用户操作，如图 10-31 所示。

添加用户界面如图 10-32 所示。

添加成功后进入成功添加用户页面，如图 10-33 所示。

图 10-31　登录成功页面

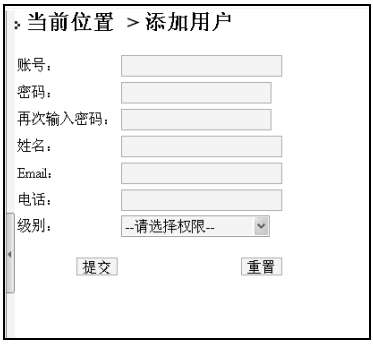


图 10-32　添加用户页面

图 10-33　成功添加用户页面

2．抽象测试套

根据被测系统中登录页面和添加用户页面所提供的信息，在抽象测试套中定义的测试数据类型如下：

```
type record login_request{//登录操作所发送的数据类型
        charstring method,
        charstring url,
        charstring username,
        charstring password
    }
type record addUser_request{//添加用户操作所发送的数据类型

        charstring method,
        charstring url,
        charstring username,
        charstring password,
        charstring confpassword,
        charstring realname,
        charstring email,
        charstring phone,
```

```
        charstring privi
    }
```

因为此案例的接收数据类型为基本数据类型——charstring 类型，所以没有进行单独定义。详细的 TTCN-3 代码见附录B。

3．编解码器

编解码器中具体继承的父类与实现的接口与上一案例中介绍的相同。本例中编解码过程简介如下：

(1)编码部分。

编码过程：

① 获取记录类型的测试数据。

② 获取记录类型数据中的每一个域值。

③ 将获得的每个域值组合成 Java 数据类型的数据。

④ 把组合后的数据转换成字符流。

(2)解码部分。

解码过程：

① 判断被测系统返回的数据类型是否为测试套中预期的接收类型。

② 如果是预期的数据类型，则从字符流中取出将其转换为 TTCN-3 能识别的数据类型，返回测试成功。

③ 如果不是，则做出相应处理，返回测试失败。

4．适配器

适配器的结构及其所包含的基本方法与上一案例相同，具体的方法及功能介绍参照上一案例。本案例中重点说明适配器的 TriSend 方法的具体设计。

```
得到发送消息的字节流；
将字节流转换为请求字符串；
截取 URL 字符串；
建立 URL 连接；
If(会话 ID 不为空)
设置 Cookie 属性为会话 ID；
截取请求方法字符串；
设置请求方法；
截取请求内容字符串；
得到输出流；
请求内容写入输出流；
创建接收线程；
If(HTTP 响应状态码不等于 200)
        记录消息日志；
Else
        得到输入流；
        If(会话 ID 为空)
                设置会话 ID 为 HTTP 头域的 Set-Cookie 属性值；
        通过输入流读取响应字符串；
        通过响应字符串创建 TriMessage 对象；
```

```
                将 TriMessage 对象传送给 TTCN-3 执行环境；
                关闭输入流；
                断开连接；
        开启接收线程；
        关闭输出流；
        返回 TriStatus 实例；
```

5．测试结果

先执行 TC_1 进行登录，然后再执行 TC_2 进行添加用户操作，成功测试结果如图 10-34 所示。

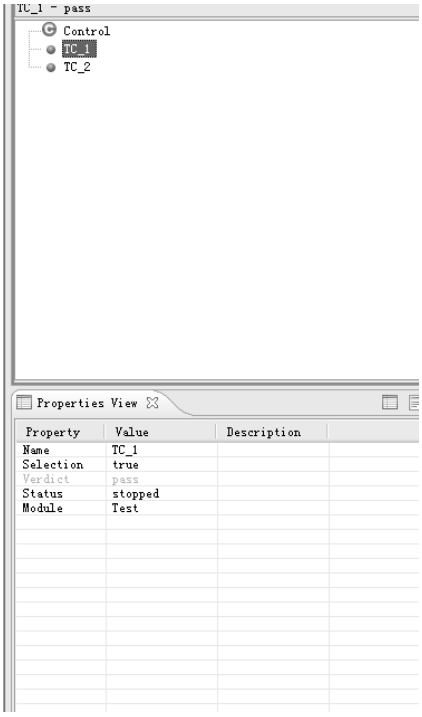


图 10-34　成功测试结果

如果测试时没有执行 TC_1 就先执行了 TC_2 即没有登录就去添加用户，那么测试就不能通过，测试结果为 error，如图 10-35 所示。

Property	Value	Description
Name	TC_2	
Selection	true	
Verdict	error	
Status	stopped	
Module	Test	

图 10-35　错误测试结果

思考题

用 TTCN 对一个 Web 应用进行测试。

附录 A　QQ 在线测试抽象测试套编码

```
 qqOnlineWebService 的抽象测试套
module qqOnlineWebService
{
    type record paramNameList
    {
        charstring qqCode
    }

    type record wsRequestType
    {
        charstring operationName,
        charstring location,
        charstring namespace,
        paramNameList paramList
    }

    type record wsResponseType
    {
        charstring response
    }

    template paramNameList paramNames :=
    {
        qqCode := "109"
    }

    template wsRequestType Request_1 :=
    {
        operationName := "qqCheckOnline",
        paramList := paramNames,
        location := "http://webservice.webxml.com.cn/WebServices/
                    qqOnlineWebService.asmx",
        namespace := "http://WebXml.com.cn/"
    }

    template wsResponseType Response_1 := {
        response := "Y"
    }

    type port wsPortType message
    {
        out wsRequestType;
```

```
            in charstring;
        }

        type component ptcType
        {
            port wsPortType ptcPort;
            timer localTimer := 10.0;
        }

        type component sysType
        {
            port wsPortType sysPort;
        }

        type component mtcType { }

        function ptcBehavior(in wsRequestType wsRequest,wsResponseType wsResponse)
                            runs on ptcType
        {
            ptcPort.send(wsRequest);
            localTimer.start;
            alt
             {
                []ptcPort.receive(wsResponse)
                {
                    localTimer.stop;
                    setverdict(pass);
                }
                []ptcPort.receive
                {
                    localTimer.stop;
                    setverdict(fail);
                }
                []localTimer.timeout
                {
                    setverdict(fail);
                }
            }
        }

        testcase TC_1() runs on mtcType system sysType
        {
            var ptcType ptc;
            ptc := ptcType.create;
            map(ptc:ptcPort,system:sysPort);
            ptc.start(ptcBehavior(Request_1,Response_1));
            ptc.done;
        }

        control
        {
            execute (TC_1());
        }
    }
```

附录 B　Web 应用测试详细的 TTCN-3 代码

```
module login_addUser {

    type record login_request{
        charstring method,
        charstring url,
        charstring username,
        charstring password
    }
    type record addUser_request{

        charstring method,
        charstring url,
        charstring username,
        charstring password,
        charstring confpassword,
        charstring realname,
        charstring email,
        charstring phone,
        charstring privi
    }

    template login_request login_request_1 := {
        method := "POST",
        url := "http://localhost:8080/testadapterlib/cn.edu.ncut.login.
               LoginServlet.action",
        username := "1",
        password := "1"
    }

    template addUser_request addUser_request_1 := {

        method := "POST",
        url := "http://localhost:8080/testadapterlib/cn.edu.ncut.user.
                doadduser.action",
        username := "zhangsan",
        password := "123",
        confpassword := "123",
        realname :="zhangsan",
        email :="zhangsan",
        phone :="123456",
        privi :="administrator"
```

```
}

type port loginPort message
{
    out login_request;
    in charstring

}

type port addUserPort message
{
    out addUser_request;
    in charstring
}

type component logincom
{
    port loginPort ptc_login
}
type component addUsercom
{
    port addUserPort ptc_addUser
}

type component sysType
{
    port loginPort sysLoginPort;
    port addUserPort sysAddUserPort
}

type component mtcType { }



function loginBehavior(in template login_request request,in charstring
                       response)runs on logincom
{
    ptc_login.send(request);
    timer localTimer := 5.0;
    localTimer.start;
    alt
     {
        []ptc_login.receive(response)
        {
            localTimer.stop;
            setverdict(pass);
        }
```

```
        []ptc_login.receive
        {
            localTimer.stop;
            setverdict(fail);
        }
        []localTimer.timeout
        {
            setverdict(fail);
        }
    }
}

function addUserBehavior(in template addUser_request request,in
                         charstring response)runs on addUsercom
{
    ptc_addUser.send(request);
    timer localTimer := 5.0;
    localTimer.start;
    alt
     {
        []ptc_addUser.receive(response)
        {
            localTimer.stop;
            setverdict(pass);
        }
        []ptc_addUser.receive
        {
            localTimer.stop;
            setverdict(fail);
        }
        []localTimer.timeout
        {
            setverdict(fail);
        }
    }
}

testcase TC_1() runs on mtcType system sysType
{
    var logincom ptc;
    ptc := logincom.create;
    map(ptc:ptc_login,system:sysLoginPort);
    ptc.start(loginBehavior(login_request_1,"Login success."));
    ptc.done;
    unmap(ptc:ptc_login,system:sysLoginPort);
}
```

```
    testcase TC_2() runs on mtcType system sysType
    {
        var addUsercom ptc;
        ptc := addUsercom.create;
        map(ptc:ptc_addUser,system:sysAddUserPort);
        ptc.start(addUserBehavior(addUser_request_1,"Add user success."));
        ptc.done;
        unmap(ptc:ptc_addUser,system:sysAddUserPort);
    }

    control
    {
        execute (TC_1());
        execute (TC_2());
    }


}
```