第3章 面向传统开发过程的软件测试

本章学习重点

- 熟悉常见的软件测试模型以及它们之间的区别。
- 掌握软件生命周期的几个阶段。
- 了解常见的软件开发模型。
- 了解什么是单元测试、集成测试和系统测试。
- 熟悉单元测试、集成测试和系统测试的主要工作内容。
- 掌握编写测试用例的方法。

本章学习难点

熟悉常见软件测试模型之间的区别。

3.1 软件测试模型

目前常见的主流软件生命周期模型或软件开发过程模型有瀑布模型、原型模型、螺旋模型、增量模型、渐进模型、快速软件开发(RAD)以及 Rational 统一过程(RUP)等,这些模型对于软件开发过程具有很好的指导作用。但在这些过程方法中,软件测试的地位和价值并没有体现出来,也没有给软件测试以足够的重视,利用这些模型显然无法更好地指导测试实践。软件测试是与软件开发紧密相关的一系列有计划、系统性的活动,软件测试也需要测试模型去指导实践。下面先对主要的软件测试模型做一些简单的介绍,再补充软件生命周期做介绍。

1. V 模型

V 模型是最具有代表性的测试模型。V 模型最早是由 Paul Rook 在 20 世纪 80 年代后期提出的,V 模型在英国国家计算中心文献中发布,旨在改进软件开发的效率和效果。

在传统的开发模型中,例如瀑布模型,通常把测试过程作为在需求分析、概要设计、详细设计和编码全部完成之后的一个阶段,尽管有时测试工作会占用整个项目周期一半的时间,但是有人仍认为测试只是一个收尾工作,而不是主要的工程。V 模型的推出就是对此种认

识的改进。V 模型是软件开发瀑布模型的变种，它反映了测试活动与分析和设计的关系，从左到右，描述了基本的开发过程和测试行为，明确地标明了测试工程中存在的不同级别，清楚地描述了这些测试阶段和开发过程期间各阶段的对应关系，如图 3-1 所示。

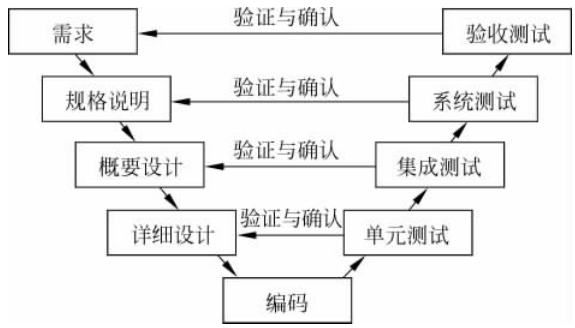


图 3-1　V 模型

图 3-1 中，V 模型图中箭头代表了时间方向，左边下降的是开发过程各阶段，与此相对应的是右边上升的部分，即测试过程的各个阶段。

V 模型的软件测试策略既包括底层测试又包括上层测试，底层测试是为了确保源代码的正确性，上层测试是为了使整个系统满足用户的需求。

V 模型指出，单元和集成测试是验证程序设计，开发人员和测试组应检测程序的执行是否满足软件设计的要求；系统测试应当验证系统设计，检测系统功能、性能的质量特性是否达到系统设计的指标；由测试人员和用户进行软件的确认测试和验收测试，追溯软件需求说明书进行测试，以确定软件的实现是否满足用户需求或合同的要求。

V 模型存在一定的局限性，它仅把测试过程作为在需求分析、概要设计、详细设计及编码之后的一个阶段。容易使人理解为测试是软件开发的最后一个阶段，主要是针对程序进行测试寻找错误，而需求分析阶段隐藏的问题一直到后期的验收测试才被发现。

类比记忆：此模型与软件开发模式中的线性模型（典型的瀑布模型）有相似的不足，在瀑布模型中，测试阶段处于软件实现后，这意味着必须在代码完成后有足够的时间预留给测试活动，否则将导致测试不充分，开发前期未发现的错误会传递并扩散到后面的阶段，而在后面发现这些错误时，可能已经很难回头再修正，从而导致项目的失败。

2. W 模型

V 模型的局限性在于没有明确地说明早期的测试，不能体现“尽早地和不断地进行软件测试”的原则。在 V 模型中增加软件各开发阶段应同步进行的测试，被演化为一种 W 模型，因为实际上开发是“V”，测试也是与此相并行的“V”。只不过把这二者结合起来进行“并行工程”而已，基于“尽早地和不断地进行软件测试”的原则，在软件的需求和设计阶段的测试活动应遵循 IEEE STD 1012—1998《软件验证和确认（V&V）》的原则。

一个基于 V&V 原理的 W 模型示意图如图 3-2 所示。

相对于 V 模型，W 模型更科学。W 模型可以说是 V 模型自然而然的发展。W 模型强调测试伴随着整个软件开发周期，而且测试的对象不仅是程序，需求、功能和设计同样要测试。这样，只要相应地开发活动完成，就可以开始执行测试，可以说，测试与开发是同步进行

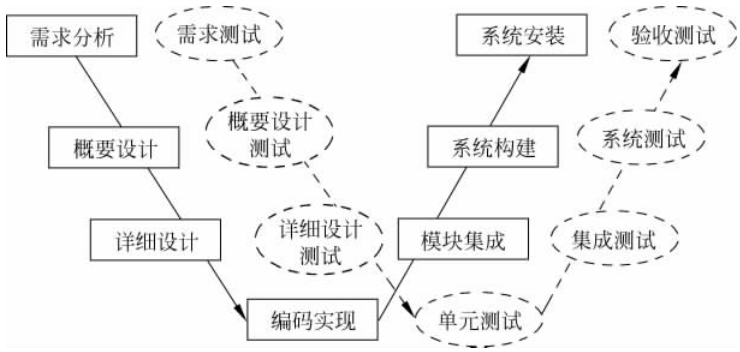


图 3-2　W 模型

的，从而有利于尽早地发现问题。以需求为例，需求分析一完成，就可以对需求进行测试，而不是等到最后才进行针对需求的验收测试。

如果测试文档能尽早提交，那么就有了更多的检查和检阅的时间，这些文档还可用于评估开发文档。另外还有一个很大的益处是，测试者可以在项目中尽可能早地面对规格说明书中的挑战。这意味着测试不仅是评定软件的质量，还可以尽可能早地找出缺陷所在，从而帮助改进项目内部的质量。参与前期工作的测试者可以预先估计问题和难度，这将可以显著地减少总体测试时间，加快项目进度。

根据 W 模型的要求，一旦有文档提供，就要及时确定测试条件，以及编写测试用例，这些工作对测试的各级别都有意义。当需求被提交后，就需要确定高级别的测试用例来测试这些需求。当概要设计编写完成后，就需要确定测试条件来查找该阶段的设计缺陷。

W 模型也是有局限性的。这一点实际上源于 V 模型的缺陷，W 模型和 V 模型都把软件的开发视为需求、设计、编码等一系列串行的活动。同样，软件开发和测试保持一种线性的前后关系，需要有严格的指令表示上一阶段完全结束，才可以正式开始下一个阶段。这样就无法支持迭代、自发性以及变更调整。对于当前很多文档需要事后补充，或者根本没有文档的做法（这已成为一种开发的文化），开发人员和测试人员都面临同样的困惑。

类比记忆：W 模型相当于两个 V 模型的叠加，一个是开发的 V，一个是测试的 V，由于在项目中开发和测试是同步进行的，相当于两个 V 是并列同步进行的，测试在一定程度是随着开发的进展而不断向前进行的。

3. H 模型

V 模型和 W 模型均存在一些不妥之处。首先，如前所述，它们都把软件的开发视为需求、设计、编码等一系列串行的活动，而事实上，虽然这些活动之间存在相互牵制的关系，但在大部分时间内，它们是可以交叉进行的。虽然软件开发期望有清晰的需求、设计和编码阶段，但实践告诉我们，严格的阶段划分只是一种理想状况。试问，有几个软件项目是在有了明确的需求之后才开始设计的呢？所以，相应的测试之间也不存在严格的次序关系。同时，各层次之间的测试也存在反复触发、迭代和增量关系。其次，V 模型和 W 模型都没有很好地体现测试流程的完整性。

为了解决以上问题，提出了 H 模型。它将测试活动完全独立出来，形成一个完全独立的流程，将测试准备活动和测试执行活动清晰地体现出来。H 模型的简单示意图如图 3-3 所示。

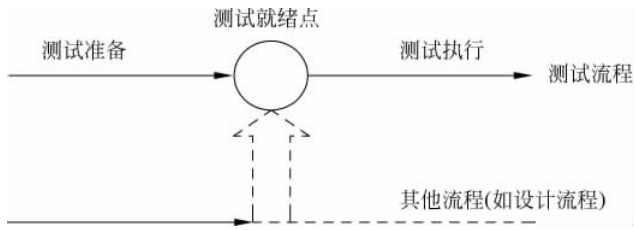


图 3-3 H 模型

H 模型图仅演示了在整个生存周期中某个层次上的一次测试“微循环”。图中的其他流程可以是任意开发流程。例如,设计流程和编码流程。也可以是其他非开发性流程,例如 SQA 流程,甚至是测试流程自身。也就是说,只要测试条件成熟,测试准备活动完成了,测试执行活动就可以(或者说需要)进行了。

概括地说,H 模型揭示了:

(1) 软件测试不仅指测试的执行,还包括很多其他的活动。

(2) 软件测试是一个独立的流程,贯穿产品整个生命周期,与其他流程并发地进行。

(3) 软件测试要尽早准备,尽早执行。

(4) 软件测试是根据被测物的不同而分层次进行的。不同层次的测试活动可以是按照某个次序先后进行的,但也可能是反复的。

在 H 模型中,软件测试模型是一个独立的流程,贯穿于整个产品周期,与其他流程并发地进行。当某个测试时间点就绪时,软件测试即从测试准备阶段进入测试执行阶段。

4. X 模型

下面介绍另外一种测试模型,即 X 模型,其目标是弥补 V 模型的一些缺陷。该模型也是对 V 模型的改进,X 模型提出针对单独的程序片段进行相互分离的编码和测试,此后通过频繁的交接,通过集成最终合成为可执行的程序。软件测试 X 模型如图 3-4 所示。

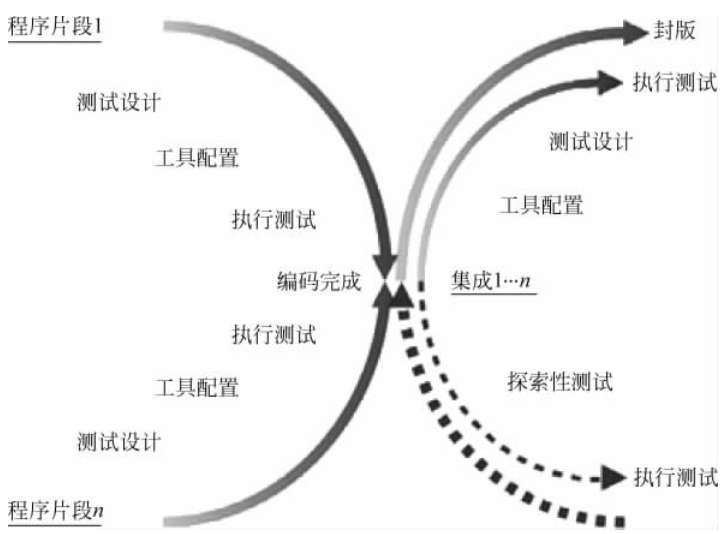


图 3-4 X 模型

X 模型的基本思想是由 Marick 提出的，Marick 对 V 模型最主要的批评是 V 模型无法引导项目的全部过程。他认为一个模型必须能处理开发的所有方面，包括交接、频繁重复的集成以及需求文档的缺乏等。Marick 认为一个模型不应该规定那些和当前所公认的实践不一致的行为。

X 模型的左边描述的是针对单独程序片段所进行的相互分离的编码和测试，此后将进行频繁的交接，通过集成最终成为可执行的程序，然后再对这些可执行程序进行测试。已通过集成测试的成品可以进行封装并提交给用户，也可以作为更大规模和范围内集成的一部分。多根并行的曲线表示变更可以在各个部分发生。从图 3-4 中可见，X 模型还定位了探索性测试，这是不进行事先计划的特殊类型的测试，这一方式往往能帮助有经验的测试人员在测试计划之外发现更多的软件错误。但这样可能对测试造成人力、物力和财力的浪费，对测试员的熟练程度要求比较高。

Marick 对 V 模型提出质疑，也是因为 V 模型是基于一套必须按照一定顺序严格排列的开发步骤，而这很可能并没有反映实际的实践过程。因为在实践过程中，很多项目是缺乏足够的需求的，而 V 模型还是从需求处理开始。

Marick 也质疑了单元测试和集成测试的区别，因为在某些场合人们可能会跳过单元测试而热衷于直接进行集成测试。Marick 担心人们盲目地跟随“学院派的 V 模型”，按照模型所指导的步骤进行工作，而实际上某些做法并不切合实际。

5. 前置测试模型

前置测试模型是将测试和开发紧密结合的模型，该模型提供了轻松的方式，可以使项目加快速度。

前置测试模型如图 3-5 所示。

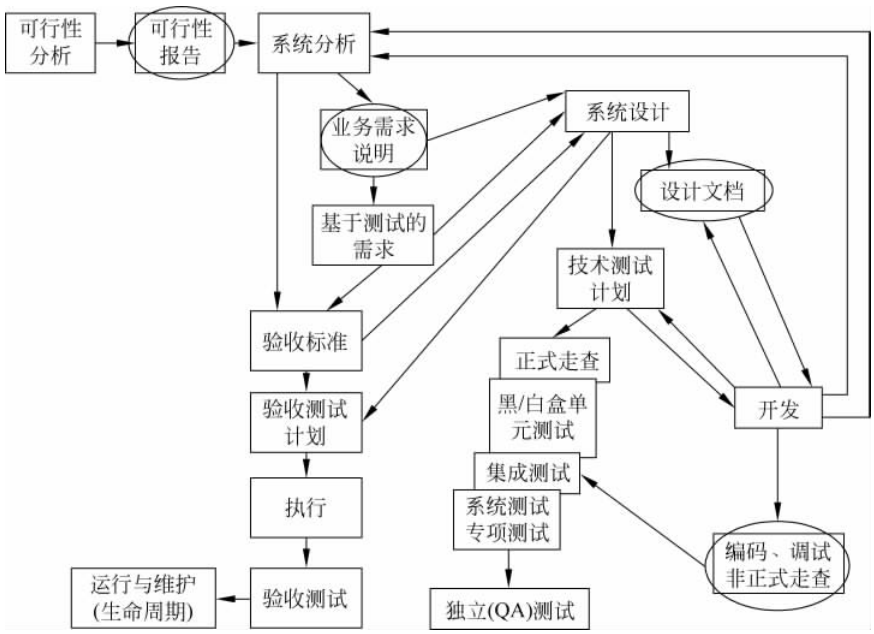


图 3-5　前置测试模型

前置测试模型体现了以下要点。

(1) 开发和测试相结合。前置测试模型将开发和测试生命周期整合在一起,标识了项目生命周期从开始到结束之间的关键行为,并且标识了这些行为在项目周期中的价值所在。如果其中有些行为没有得到很好的执行,那么项目成功的可能性就会因此而有所降低。如果有业务需求,则系统开发过程将更有效率。我们认为在没有业务需求的情况下进行开发和测试是不可能的。而且,业务需求最好在设计和开发之前就被正确定义。

(2) 对每一个交付内容进行测试。每一个交付的开发结果都必须通过一定的方式进行测试。源程序代码并不是唯一需要测试的内容。图中的椭圆框表示了其他一些要测试的对象,包括可行性报告、业务需求说明,以及系统设计文档等。这同 V 模型中开发和测试的对应关系是一致的,并且在其基础上有所扩展,变得更为明确。

(3) 在设计阶段进行测试计划和测试设计。设计阶段是做测试计划和测试设计的最好时机。很多组织要么根本不做测试计划和测试设计,要么在即将开始执行测试之前才飞快地完成测试计划和测试设计。在这种情况下,测试只是验证了程序的正确性,而不是验证整个系统本该实现的东西。

(4) 测试和开发结合在一起。前置测试将测试执行和开发结合在一起,并在开发阶段以编码—测试—编码—测试的方式来体现。也就是说,程序片段一旦编写完成,就会立即进行测试。一般情况下,先进性的测试是单元测试,因为开发人员认为通过测试来发现错误是最经济的方式。但也可参考 X 模型,即一个程序片段也需要相关的集成测试,甚至有时还需要一些特殊测试。对于一个特定的程序片段,其测试的顺序可以按照 V 模型的规定,但其中还会交织一些程序片段的开发,而不是按阶段完全地隔离。

(5) 让验收测试和技术测试保持相对独立。验收测试应该独立于技术测试,这样可以提供双重的保险,以保证设计及程序编码能够符合最终用户的要求。验收测试既可以在实施的第一步来执行,也可以在开发阶段的最后一步执行。前置测试模型提倡验收测试和技术测试沿循两条不同的路线来进行,每条路线分别验证系统是否能够如预期设想的那样进行正常工作。这样,当单独设计好的验收测试完成了系统的验证时,即可确信这是一个正确的系统。

前置测试模型包括两项测试计划技术,这也是需求测试技术中的一部分。其中的第一项技术是开发基于需求的测试用例。这并不仅仅是为以后提交上来的程序的测试做好初始化准备,也是为了验证需求是否是可测试的。这些测试可以交由用户来进行验收测试,或者由开发部门做某些技术测试。很多测试团体都认为,需求的可测试性即使不是需求首要的属性,也应是其最基本的属性之一。因此,在必要的时候可以为每一个需求编写测试用例。不过,基于需求的测试最多也只是和需求本身一样重要。一项需求可能本身是错误的,但它仍是可测试的。而且,无法为一些被忽略的需求来编写测试用例。

第二项技术是定义验收标准。在接受交付的系统之前,用户需要用验收标准来进行验证。验收标准并不仅仅是定义需求,还应在前置测试之前进行定义,这将帮助揭示某些需求是否正确,以及某些需求是否被忽略了。

在 V 模型中,验收测试最早被定义好,并在最后执行,以验证所交付的系统是否真正符合用户业务的需求。

与 V 模型不同的是,前置测试模型认识到验收测试中所包含的三种成分,其中的两种

都与业务需求定义相联系：即定义基于需求的测试，以及定义验收标准。但是，第三种则需要等到系统设计完成，因为验收测试计划是由针对按设计实现的系统来进行的一些明确操作定义所组成，这些定义包括：如何判断验收标准已经达到，以及基于需求的测试已算成功完成。

技术测试主要是针对开发代码的测试，例如，V 模型中所定义的动态的单元测试，集成测试和系统测试。另外，前置测试还提示我们应增加静态审查，以及独立的 QA 测试。

QA 测试通常跟随在系统测试之后，从技术部门的意见和用户的预期方面出发，进行最后的检查。同样的还有特别测试。将其取名为特别测试，并把该名称作为很多测试的一个统称，这些测试包括负载测试、安全性测试、可用性测试等，它们不是由业务逻辑和应用来驱动的。

对技术测试最基本的要求是验证代码的编写和设计的要求是否相一致。一致的意思是系统确实提供了要求提供的，并且系统并没有提供不要求提供的。技术测试在设计阶段进行计划和设计，并在开发阶段由技术部门来执行。

6. 测试模型的使用

前面介绍了几种典型的测试模型，应该说这些模型对指导测试工作的进行具有重要的意义，但任何模型都不是完美的。应该尽可能地去应用模型中对项目有实用价值的方面，但不强行地为使用模型而使用模型，否则也没有实际意义。

在这些模型中，V 模型强调了在整个软件项目开发中需要经历的若干个测试级别，而且每一个级别都与一个开发级别相对应，但它忽略了测试的对象不应该仅包括程序，或者说它没有明确地指出应该对软件的需求、设计进行测试，而这一点在 W 模型中得到了补充。W 模型强调了测试计划等工作的先行和对系统需求和系统设计的测试，但 W 模型和 V 模型一样也没有专门对软件测试流程予以说明，因为事实上，随着软件质量要求越来越为大家所重视，软件测试也逐步发展成为一个独立于软件开发的一系列活动，就每一个软件测试的细节而言，它都有一个独立的操作流程。例如，现在的第三方测试，就包含从测试计划和测试用例编写，到测试实施以及测试报告编写的全过程，这个过程在 H 模型中得到了相应的体现，表现为测试是独立的。也就是说，只要测试前提具备了，就可以开始进行测试。当然，X 模型和前置测试模型又在此基础上增加了许多不确定因素的处理情况，因为在真实项目中，经常会有变更的发生，例如，需要重新访问前一阶段的内容，或者跟踪并纠正以前提交的内容，修复错误，排除多余的成分，以及增加新发现的功能等。

因此，在实际的工作中，要灵活地运用各种模型的优点，在 W 模型的框架下，运用 H 模型的思想进行独立的测试，并同时将测试与开发紧密结合，寻找恰当的就绪点开始测试并反复迭代测试，最终保证按期完成预定目标。

3.2　软件生命周期

和任何事物一样，软件也有其孕育、诞生、成长、成熟和衰亡的生存过程，一般称其为“软件生命周期”。软件生命周期一般分为 6 个阶段，即制定计划、需求分析、设计、编码、测试、运行和维护。现实的软件开发的各个阶段之间的关系不可能是顺序且线性的，而应该是带有反馈的迭代过程。在软件工程中，这个复杂的过程用软件开发模型来描述和表示。

软件开发模型是跨越整个软件生存周期的系统开发、运行和维护所实施的全部工作和任务的结构框架，它给出了软件开发活动各阶段之间的关系。目前，常见的软件开发模型大致可分为如下三种类型。

(1) 以软件需求完全确定为前提的瀑布模型(Waterfall Model)。

(2) 在软件开发初始阶段只能提供基本需求时采用的渐进式开发模型，如螺旋模型(Spiral Model)。

(3) 以形式化开发方法为基础的变换模型(Transformational Model)。

1. 瀑布模型

瀑布模型是最早出现的软件开发模型，在软件工程中占有重要的地位，它提供了软件开发的基本框架。其过程是从上一项活动接收该项活动的工作对象作为输入，利用这一输入实施该项活动应完成的内容给出该项活动的工作成果，并作为输出传给下一项活动，如图 3-6 所示。

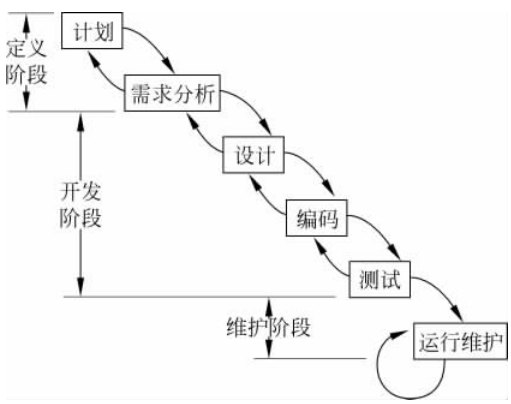


图 3-6 瀑布模型

瀑布模型核心思想是按工序将问题化简，将功能的实现与设计分开，便于分工协作，即采用结构化的分析与设计方法将逻辑实现与物理实现分开。将软件生命周期划分为制定计划、需求分析、软件设计、程序编写、软件测试和运行维护 6 个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。

1) 瀑布模型的优点

(1) 为项目提供了按阶段划分的检查点。

(2) 当前一阶段完成后，只需要去关注后续阶段。

(3) 可在迭代模型中应用瀑布模型。

增量迭代应用于瀑布模型。迭代 1 解决最大的问题。每次迭代产生一个可运行的版本，同时增加更多的功能。每次迭代必须经过质量和集成测试。

2) 瀑布模型的缺点

(1) 在项目各个阶段之间极少有反馈。

(2) 只有在项目生命周期的后期才能看到结果。

(3) 通过过多的强制完成日期和里程碑来跟踪各个项目阶段。

2. 原型模型

原型模型如图 3-7 所示。

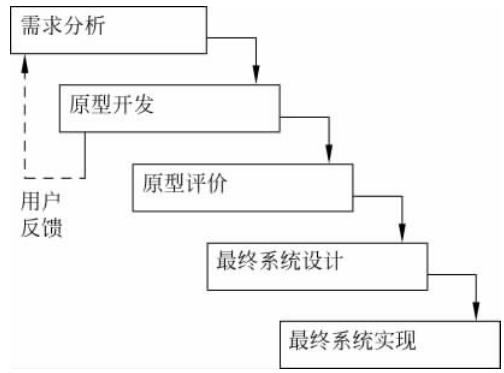


图 3-7　原型模型

原型模型的主要思想：先借用已有系统作为原型模型，通过“样品”不断改进，使得最后的产品就是用户所需要的。

原型模型通过向用户提供原型获取用户的反馈，使开发出的软件能够真正反映用户的需求。同时，原型模型采用逐步求精的方法完善原型，使得原型能够“快速”开发，避免了像瀑布模型一样在冗长的开发过程中难以对用户的反馈做出快速的响应。相对瀑布模型而言，原型模型更符合人们开发软件的习惯，是目前较流行的一种实用软件生存期模型。

1）原型模型的优点

（1）开发人员和用户在“原型”上达成一致。这样一来，可以减少设计中的错误和开发中的风险，也减少了对用户培训的时间，从而提高了系统的实用、正确性以及用户的满意程度。

（2）缩短了开发周期，加快了工程进度。

（3）降低成本。

2）原型模型的缺点

当告诉用户还必须重新生产该产品时，用户是很难接受的。这往往给工程继续开展带来不利因素。

不宜利用原型系统作为最终产品。采用原型模型开发系统，用户和开发者必须达成一致：原型被建造仅仅是用户用来定义需求，之后便部分或全部抛开，最终的软件是要充分考虑了质量和可维护性等方面之后才被开发出来。

3. 螺旋模型

螺旋模型采用一种周期性的方法来进行系统开发，这会导致开发出众多的中间版本。使用它，项目经理在早期就能够为客户实证某些概念。该模型是快速原型法，以进化的开发方式为中心，在每个项目阶段使用瀑布模型法。这种模型的每一个周期都包括需求定义、风险分析、工程实现和评审 4 个阶段，由这 4 个阶段进行迭代。软件开发过程每迭代一次，软件开发又前进一个层次。采用螺旋模型的软件过程如图 3-8 所示。

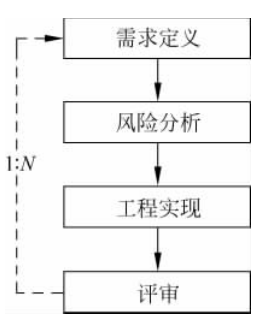


图 3-8　螺旋模型的软件过程

螺旋模型基本做法是在“瀑布模型”的每一个开发阶段前引入一个非常严格的风险识别、风险分析和风险控制，它把软件项目分解成一个个小项目。每个小项目都标识一个或多个主要风险，直到所有的主要风险因素都被确定。

螺旋模型强调风险分析，使得开发人员和用户对每个演化层出现的风险有所了解，继而做出应有的反应，因此特别适用于庞大、复杂并具有高风险的系统。对于这些系统，风险是软件开发不可忽视且潜在的不利因素，它可能在不同程度上损害软件开发过程，影响软件产品的质量。减小软件风险的目标是在造成危害之

前，及时对风险进行识别及分析，决定采取何种对策，进而消除或减少风险的损害。

螺旋模型沿着螺线进行若干次迭代，图 3-9 中的 4 个象限代表了以下活动。

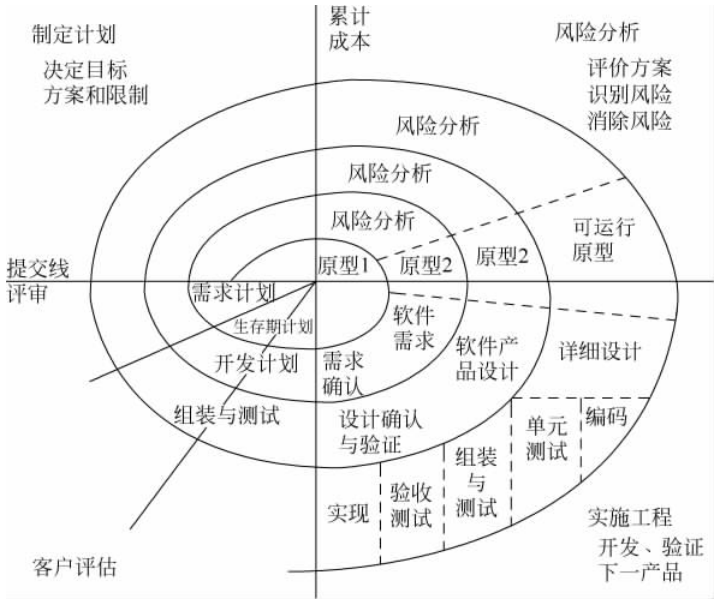


图 3-9　螺旋模型

(1) 制定计划：确定软件目标，选定实施方案，弄清项目开发的限制条件。

(2) 风险分析：分析评估所选方案，考虑如何识别和消除风险。

(3) 实施工程：实施软件开发和验证。

(4) 客户评估：评价开发工作，提出修正建议，制定下一步计划。

螺旋模型由风险驱动，强调可选方案和约束条件从而支持软件的重用，有助于将软件质量作为特殊目标融入产品开发之中。

螺旋模型很大程度上是一种风险驱动的方法体系，因为在每个阶段之前及经常发生的循环之前，都必须首先进行风险评估。在实践中，螺旋法技术和流程变得更为简单。迭代方法体系更倾向于按照开发/设计人员的方式工作，而不是项目经理的方式。螺旋模型中存在众多变量，并且在将来会有更大幅度的增长，该方法体系正良好运作着。表 3-1 是螺旋法能够解决的各种问题。

表 3-1　螺旋模型解决方案

经常遇到的问题	螺旋模型的解决方案
用户需求不够充分	允许并鼓励用户反馈信息
沟通不明	在项目早期就消除严重的曲解
刚性的体系	开发首先关注重要的业务和问题
主观臆断	通过测试和质量保证，做出客观的评估
潜在的不一致	在项目早期就发现不一致问题
糟糕的测试和质量保证	从第一次迭代就开始测试
采用瀑布法开发	在早期就找出并关注风险

(1) 螺旋模型强调风险分析,但要求许多客户接受和相信这种分析,并做出相关反应是不容易的,因此,这种模型往往适应于内部的大规模软件开发。

(2) 如果执行风险分析将大大影响项目的利润,那么进行风险分析毫无意义,因此,螺旋模型只适合大规模软件项目。

(3) 软件开发人员应该擅长寻找可能的风险,准确地分析风险,否则将会带来更大的风险。

一个阶段首先是确定该阶段的目标,完成这些目标的选择方案及其约束条件,然后从风险角度分析方案的开发策略,努力排除各种潜在的风险,有时需要通过建造原型来完成。如果某些风险不能排除,该方案立即终止,否则启动下一个开发步骤。最后,评价该阶段的结果,并设计下一个阶段。

1) 螺旋模型的优点

(1) 设计上的灵活性,可以在项目的各个阶段进行变更。

(2) 以小的分段来构建大型系统,使成本计算变得简单容易。

(3) 客户始终参与每个阶段的开发,保证了项目不偏离正确方向以及项目的可控性。

(4) 随着项目推进,客户始终掌握项目的最新信息,从而能够和管理层有效地交互。

(5) 客户认可这种公司内部的开发方式带来的良好的沟通和高质量的产品。

2) 螺旋模型的缺点

很难让用户确信这种演化方法的结果是可以控制的。建设周期长,而软件技术发展比较快,所以经常出现软件开发完毕后,和当前的技术水平有了较大的差距,无法满足当前用户需求。

对于新近开发,需求不明确的情况下,适合用螺旋模型进行开发,便于风险控制和需求变更。

3.3　单元测试

1. 单元测试概述

单元测试(Unit Testing)是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义,一般来说,要根据实际情况去判定其具体含义,如 C 语言中单元指一个函数,Java 里单元指一个类,图形化的软件中可以指一个窗口或一个菜单等。总地来说,单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动,软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

单元测试(模块测试)一般是开发者编写的一小段代码,用于检验被测代码的一个很小的、很明确的功能是否正确。通常而言,一个单元测试是用于判断某个特定条件(或者场景)下某个特定函数的行为。例如,可能把一个很大的值放入一个有序 list 中,然后确认该值出现在 list 的尾部。或者,可能会从字符串中删除匹配某种模式的字符,然后确认字符串中不再包含这些字符。

单元测试是由程序员自己来完成的,最终受益的也是程序员自己。可以这么说,程序员有责任编写功能代码,同时也就有责任为自己的代码编写单元测试。执行单元测试,就是为了证明这段代码的行为和期望功能一致。

工厂在组装一台电视机之前，会对每个元件都进行测试，这就是单元测试。其实我们每天都在做单元测试。编写一个函数，除了极简单的外，总是要执行一下，看看功能是否正常，有时还要想办法输出些数据，如弹出信息窗口，这也是单元测试。这种单元测试称为临时单元测试。只进行了临时单元测试的软件，针对代码的测试很不完整，代码覆盖率要超过70%都很困难，未覆盖的代码可能遗留大量的细小的错误，这些错误还会互相影响，当软件缺陷暴露出来的时候难于调试，大幅度提高后期测试和维护成本，也降低了开发商的竞争力。可以说，进行充分的单元测试，是提高软件质量，降低开发成本的必由之路。

对于程序员来说，如果养成了对自己写的代码进行单元测试的习惯，不但可以写出高质量的代码，而且还能提高编程水平。

要进行充分的单元测试，应专门编写测试代码，并与产品代码隔离。比较简单的办法是为产品工程建立对应的测试工程，为每个类建立对应的测试类，为每个函数（很简单的除外）建立测试函数。

一般认为，在结构化程序时代，单元测试所说的单元是指函数，在当今的面向对象时代，单元测试所说的单元是指类。以实践来看，以类作为测试单位，复杂度高，可操作性较差，因此仍然主张以函数作为单元测试的测试单位，但可以用一个测试类来组织某个类的所有测试函数。单元测试不应过分强调面向对象，因为局部代码依然是结构化的。单元测试的工作量较大，简单、实用、高效才是进行单元测试的硬道理。

有一种看法是，只测试类的接口（公有函数），不测试其他函数，从面向对象角度来看，确实有其道理，但是，测试的目的是找错并最终排错，因此，只要是包含错误的可能性较大的函数都要测试，跟函数是否私有没有关系。对于C++语言来说，可以用一种简单的方法区隔需测试的函数：简单的函数如数据读写函数的实现在头文件中编写（inline 函数），所有在源文件中编写实现的函数都要进行测试（构造函数和析构函数除外）。

程序员编写代码时，一定会反复调试保证它能够编译通过。如果是编译没有通过的代码，没有任何人会愿意交付。但代码通过编译，只是说明了它的语法正确，无法保证它的语义也一定正确，没有任何人可以轻易承诺这段代码的行为一定是正确的。

幸运的是，单元测试会为单位功能的正确性承诺做保证。编写单元测试就是用来验证这段代码的行为是否与期望的一致。有了单元测试，程序员可以自信地交付自己的代码，而没有任何的后顾之忧。

单元测试与其他测试不同，单元测试可看作是编码工作的一部分，应该由程序员完成，也就是说，经过了单元测试的代码才是已完成的代码，提交产品代码时也要同时提交测试代码，测试部门可以做一定程度的审核。

2. 单元测试的任务

软件测试是为了发现错误而执行程序的过程。它是根据程序开发阶段的规格说明及程序内部结构而精心设计的一批测试用例（输入数据及其预期结果的集合），并利用该测试用例去运行程序，以发现错误的过程。软件测试的目的是为了尽早尽可能多地发现软件中的缺陷，提高软件产品的质量。

软件的单元测试是测试过程中最基本的测试，单元是软件的构成基础，因此单元的质量是整个软件质量的基础。单元测试是软件测试过程中直接与代码相关的测试，它对其他测

试步骤的进行有重要影响。单元测试在软件的编码阶段进行，主要完成的工作是根据详细设计说明书编写程序源代码，包括必要的数据文件，并进行单元测试，及时更正测试问题。

单元测试包括静态的代码审查和动态测试两个阶段。代码审查阶段对程序进行静态分析，包括编程风格检查、模块接口检查、程序语言检查、内存检查、比较和转移检查、性能检查、可维护性检查、逻辑检查、软件多余物检查等。动态测试阶段首先编写驱动模块和桩模块，在驱动模块和桩模块中设计相应的测试用例，测试用例应该覆盖单元模块的所有功能项，如果单元模块有性能等其他测试特性要求，则必须设计相应的测试用例测试这些特性，然后运行测试，比较测试结果。

单元测试的任务包括模块接口测试、模块局部数据结构测试、模块边界条件测试、模块中所有独立执行通路测试、模块的各条错误处理通路测试。模块测试是单元测试的基础。只有在数据能正确流入、流出模块的前提下，其他测试才有意义。检查局部数据结构是为了保证临时存储在模块内的数据在程序执行过程中完整、正确。除局部数据外，如果可能，单元测试时还应该查清全局数据对模块的影响。在模块中应对每一条独立执行路径进行测试，单元测试的基本任务是保证模块中每条语句至少执行一次。一个好的设计应能预见各种出错条件，并预设各种出错处理通路。边界条件测试是单元测试中最后也是最重要的一项任务，软件经常在边界上失效，采用边界值分析技术，针对边界值及其左右设计测试用例。

3. 单元测试工作内容及其流程

单元测试工作内容及其流程如表 3-2 和图 3-10 所示。

表 3-2　单元测试工作内容及其流程

活　动	输　入	输　出	参与角色和职责
制定单元测试计划	1. 详细设计 2. 实现代码(可选)	单元测试计划(该计划可以不是一个独立的计划，可包含在实施计划中)	详细设计人员负责制定单元测试计划
设计单元测试	1. 单元测试计划 2. 详细设计 3. 实现代码(可选)	1. 单元测试用例 2. 已设计好的单元测试驱动块 3. 已设计好的单元测试桩模块	详细设计人员负责设计单元测试用例、设计驱动程序桩
实施单元测试	单元测试用例	1. 单元测试驱动模块 2. 单元测试桩模块	程序员负责编写测试驱动程序和稳定桩
执行单元测试	1. 实现代码 2. 单元测试计划 3. 单元测试用例 4. 被测试单元 5. 单元测试驱动模块和桩模块	1. 测试结果 2. 测试问题报告	程序员执行测试并记录测试结果
评估单元测试	1. 单元测试计划 2. 测试结果	测试评估报告	详细设计人员负责评估此次测试并生成测试评估报告

4. 单元测试用例设计方法

现在的软件几乎都是用事件触发来控制流程的，事件触发时的情景便形成了场景，而同一事件不同的触发顺序和处理结果就形成事件流。这种在软件设计方面的思想也可以引入

到软件测试中，可以比较生动地描绘出事件触发时的情景，有利于测试设计者设计测试用例，同时使测试用例更容易理解和执行。

基本流和备选流：如图 3-11 所示，图中经过用例的每条路径都用基本流和备选流来表示，直黑线表示基本流，是经过用例的最简单的路径。备选流可以用不同的色彩和样式表示，一个备选流可能从基本流开始，在某个特定条件下执行，然后重新加入基本流中（如备选流 1 和 3）；也可能起源于另一个备选流（如备选流 2），或者终止用例而不再重新加入到某个流（如备选流 2 和 4）。

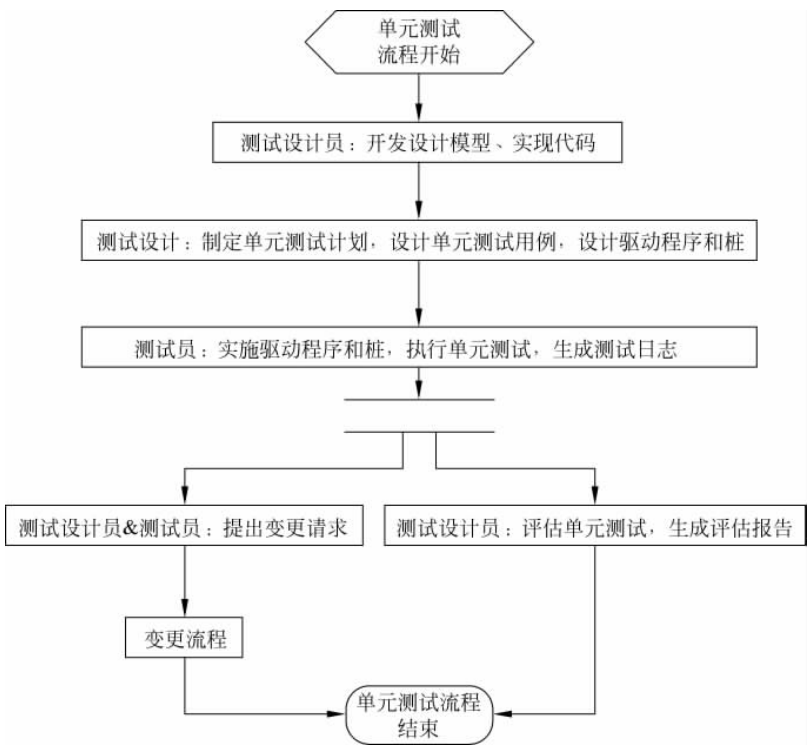


图 3-10　单元测试工序

案例：ATM 的流程示意图如图 3-12 所示。

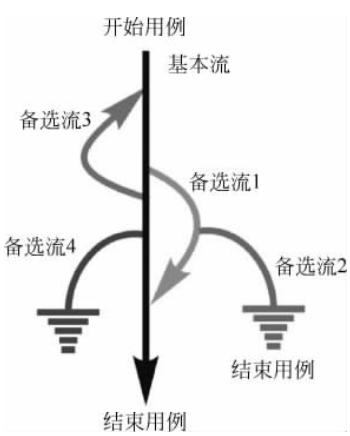


图 3-11　用例设计流程

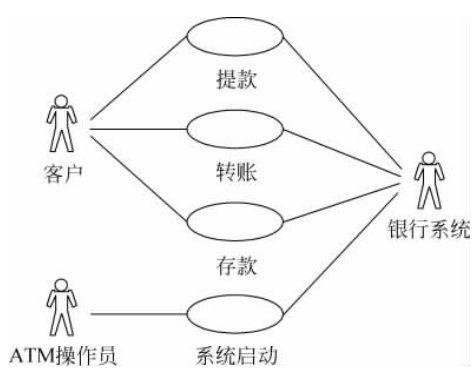


图 3-12　ATM 场景图

场景设计：如表 3-3 所示是生成的 ATM 场景。

表 3-3　场景设计

场景 1——成功提款	基本流	
场景 2——ATM 内没有现金	基本流	备选流 2
场景 3——ATM 内现金不足	基本流	备选流 3
场景 4——PIN 有误(还有输入机会)	基本流	备选流 4
场景 5——PIN 有误(不再有输入机会)	基本流	备选流 4
场景 6——账户不存在/账户类型有误	基本流	备选流 5
场景 7——账户余额不足	基本流	备选流 6

注：为方便起见，备选流 3 和 6(场景 3 和 7)内的循环以及循环组合未纳入表 3-3 中。

用例设计：对于这 7 个场景中的每一个场景都需要确定测试用例。可以采用矩阵或决策表来确定和管理测试用例。下面显示了一种通用格式，其中各行代表各个测试用例，而各列则代表测试用例的信息。本示例中，对于每个测试用例，存在一个测试用例 ID、条件(或说明)、测试用例中涉及的所有数据元素(作为输入或已经存在于数据库中)以及预期结果。

表 3-4　测试用例

TC(测试用例)ID 号	场景/条件	PIN	账号	输入(或选择)的金额	账面金额	ATM 内的金额	预期结果
CW1	场景 1：成功提款	V	V	V	V	V	成功提款
CW2	场景 2：ATM 内没有现金	V	V	V	V	I	提款选项不可用，用例结束
CW3	场景 3：ATM 内现金不足	V	V	V	V	I	警告消息，返回基本流步骤 6，输入金额
CW4	场景 4：PIN 有误(还有不止一次输入机会)	I	V	n/a	V	V	警告消息，返回基本流步骤 4，输入 PIN
CW5	场景 4：PIN 有误(还有一次输入机会)	I	V	n/a	V	V	警告消息，返回基本流步骤 4，输入 PIN
CW6	场景 4：PIN 有误(不再有输入机会)	I	V	n/a	V	V	警告消息，卡予以保留，用例结束

数据设计：一旦确定了所有的测试用例，则应对这些用例进行复审和验证以确保其准确且适度，并取消多余或等效的测试用例。

测试用例一经认可，就可以确定实际数据值(在测试用例实施矩阵中)并且设定测试数据，如表 3-5 所示。

表 3-5　测试用例表

TC(测试用例)ID 号	场景/条件	PIN	账号	输入(或选择)的金额/元	账面金额/元	ATM 内的金额/元	预期结果
CW1	场景 1：成功提款	4987	809-498	50.00	500.0	2000	成功提款。账户余额被更新为 450.00
CW2	场景 2：ATM 内没有现金	4987	809-498	100.00	500.0	0.00	提款选项不可用，用例结束

续表

TC(测试用例)ID号	场景/条件	PIN	账号	输入(或选择)的金额/元	账面金额/元	ATM内的金额/元	预期结果
CW3	场景3：ATM内现金不足	4987	809-498	100.00	500.0	70.00	警告消息，返回基本流步骤6，输入金额
CW4	场景4：PIN有误(还有不止一次输入机会)	4978	809-498	n/a	500.00	2000	警告消息，返回基本流步骤4，输入PIN
CW5	场景4：PIN有误(还有一次输入机会)	4978	809-498	n/a	500.00	2000	警告消息，返回基本流步骤4，输入PIN
CW6	场景4：PIN有误(不再有输入机会)	4978	809-498	n/a	500.00	2000	警告消息，卡予以保留，用例结束

1）等价类划分法

等价类是指某个输入域的子集合。在该子集合中，各个输入数据对于揭露程序中的错误都是等效的，并合理地假定：测试某等价类的代表值就等于对这一类其他值的测试，因此，可以把全部输入数据合理划分为若干等价类，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用少量代表性的测试数据取得较好的测试结果。等价类划分可有两种不同的情况：有效等价类和无效等价类。

(1) 有效等价类

它是指对于程序的规格说明来说是合理的、有意义的输入数据构成的集合。利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。

(2) 无效等价类

它与有效等价类的定义恰巧相反。无效等价类指对程序的规格说明是不合理的或无意义的输入数据所构成的集合。对于具体的问题，无效等价类至少应有一个，也可能有多个。

设计测试用例时，要同时考虑这两种等价类。因为软件不仅要能接收合理的数据，也要能经受意外的考验，这样的测试才能确保软件具有更高的可靠性。

(3) 划分等价类的标准

① 完备测试、避免冗余。

② 划分等价类重要的是：集合的划分，划分为互不相交的一组子集，而子集的并是整个集合。

③ 并是整个集合：完备性。

④ 子集互不相交：保证一种形式的无冗余性。

⑤ 同一类中标识(选择)一个测试用例，同一等价类中，往往处理相同，相同处理映射到“相同的执行路径”。

(4) 划分等价类的方法

① 在输入条件规定了取值范围或值的个数的情况下，则可以确立一个有效等价类和两个无效等价类。例如，输入值是学生成绩，范围是0～100，如图3-13所示。

② 在输入条件规定了输入值的集合或者规定了“必须如何”的条件的情况下，可确立一个有效等价类和一个无效等价类。

③ 在输入条件是一个布尔量的情况下，可确定一个有效等价类和一个无效等价类。

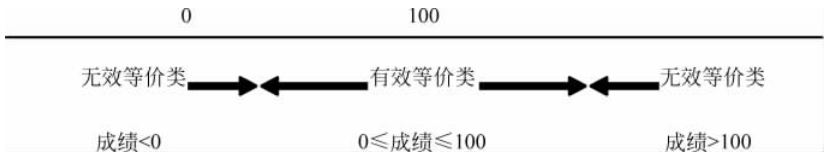


图 3-13　学生成绩的等价类划分

④ 在规定了输入数据的一组值(假定 n 个),并且程序要对每一个输入值分别处理的情况下,可确立 n 个有效等价类和一个无效等价类。

例如,输入条件说明学历可为专科、本科、硕士、博士四种之一,则分别取 4 个值作为 4 个有效等价类,另外把 4 种学历之外的任何学历作为无效等价类。

⑤ 在规定输入数据必须遵守的规则的情况下,可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)。

⑥ 在确知已划分的等价类中各元素在程序处理中的方式不同的情况下,则应再将该等价类进一步地划分为更小的等价类。

(5) 设计测试用例

在确立等价类后,可建立等价类表,列出所有划分出的等价类输入条件。然后从划分出的等价类中按以下三个原则设计测试用例。

① 为每一个等价类规定一个唯一的编号;

② 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;

③ 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。

2) 边界值分析方法

边界值分析法就是对输入或输出的边界值进行测试的一种黑盒测试方法。通常边界值分析法是作为对等价类划分法的补充,在这种情况下,其测试用例来自等价类的边界。

长期的测试工作经验告诉我们,大量的错误是发生在输入或输出范围的边界上,而不是发生在输入输出范围的内部。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。

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

常见的边界值有以下几种。

① 对 16b 的整数而言,32 767 和－32 768 是边界。

② 屏幕上光标在最左上、最右下位置。

③ 报表的第一行和最后一行。

④ 数组元素的第一个和最后一个。

⑤ 循环的第 0 次、第一次和倒数第二次、最后一次。

边界值分析使用与等价类划分法相同的划分,只是边界值分析假定错误更多地存在于

划分的边界上，因此在等价类的边界上以及两侧的情况设计测试用例。

例如，测试计算平方根的函数。

输入：实数

输出：实数

规格说明：当输入一个 0 或比 0 大的数的时候，返回其正平方根；当输入一个小于 0 的数时，显示错误信息“平方根非法-输入值小于 0”并返回 0；库函数 Print-Line 可以用来输出错误信息。

(1) 可以考虑做出如下划分。

输入：①＜0 和②≥0。

输出：①≥0 和②Error。

(2) 测试用例有两个：

① 输入 4，输出 2。

② 输入－10，输出 0 和错误提示。

第一组划分的边界为 0 和最大正实数；第二组划分的边界为最小负实数和 0。由此得到以下测试用例。

(1) 输入{最小负实数}

(2) 输入{绝对值很小的负数}

(3) 输入 0

(4) 输入{绝对值很小的正数}

(5) 输入{最大正实数}

常见等价类划分方法如表 3-6 所示。

表 3-6 常见等价类划分方法

项	边 界 值	测试用例的设计思路
字符	起始－1 个字符/结束＋1 个字符	假设一个文本输入区域允许输入 1～255 个字符，输入 1 个和 255 个字符作为有效等价类；输入 0 个和 256 个字符作为无效等价类，这几个数值都属于边界条件值
数值	最小值－1/最大值＋1	假设某软件的数据输入域要求输入 5 位的数据值，可以使用 10 000 作为最小值、99 999 作为最大值；然后使用刚好小于 5 位和大于 5 位的数值来作为边界条件
空间	小于空余空间一点儿/大于满空间一点儿	例如，在用 U 盘存储数据时，使用比剩余磁盘空间大一点儿(几 KB)的文件作为边界条件

在多数情况下，边界值条件是基于应用程序的功能设计而需要考虑的因素，可以从软件的规格说明或常识中得到，也是最终用户可以很容易发现问题的。然而，在测试用例设计过程中，某些边界值条件是不需要呈现给用户的，或者说用户是很难注意到的，但同时确实属于检验范畴内的边界条件，称为内部边界值条件或子边界值条件。

内部边界值条件主要有下面几种。

(1) 数值的边界值检验：计算机是基于二进制进行工作的，因此，软件的任何数值运算都有一定的范围限制，如表 3-7 所示。

表 3-7　数值的边界值

项	范围或值
位(bit)	0 或 1
字节(byte)	0～255
字(word)	0～65 535(单字)或 0～4 294 967 295(双字)
千(K)	1024
兆(M)	1 048 576
吉(G)	1 073 741 824

(2) 字符的边界值检验：在计算机软件中，字符也是很重要的表示元素，其中 ASCII 和 Unicode 是常见的编码方式。表 3-8 中列出了一些常用字符对应的 ASCII 码值。

表 3-8　字符的边界值

字符	ASCII 码值	字符	ASCII 码值
空(null)	0	A	65
空格(space)	32	a	97
斜杠(/)	47	Z	90
0	48	z	122
冒号(:)	58	单引号(‘)	96
@	64		

基于边界值分析方法选择测试用例的原则如下。

(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，以及刚刚超越这个范围边界的值作为测试输入数据。

例如，如果程序的规格说明中规定：“重量在 10～50kg 范围内的邮件，其邮费计算公式为……”作为测试用例，应取 10 及 50，还应取 10.01，49.99，9.99 及 50.01 等。

(2) 如果输入条件规定了值的个数，则用最大个数，最小个数，比最小个数少 1，比最大个数多 1 的数作为测试数据。

例如，一个输入文件应包括 1～255 个记录，则测试用例可取 1 和 255，还应取 0 及 256 等。

(3) 将规则(1)和(2)应用于输出条件，即设计测试用例使输出值达到边界值及其左右的值。

例如，某程序的规格说明要求计算出“每月保险金扣除额为 0～1165.25 元”，其测试用例可取 0.00 及 1165.24、还可取 0.01 及 1165.26 等。

再如一程序属于情报检索系统，要求每次“最少显示 1 条、最多显示 4 条情报摘要”，这时应考虑的测试用例包括 1 和 4，还应包括 0 和 5 等。

(4) 如果程序的规格说明给出的输入域或输出域是有序集合，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。

(5) 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。

(6) 分析规格说明，找出其他可能的边界条件。

案例：现有一个学生标准化考试批阅试卷，产生成绩报告的程序。其规格说明如下：

程序的输入文件由一些有 80 个字符的记录组成，如图 3-14 所示，所有记录分为三组。

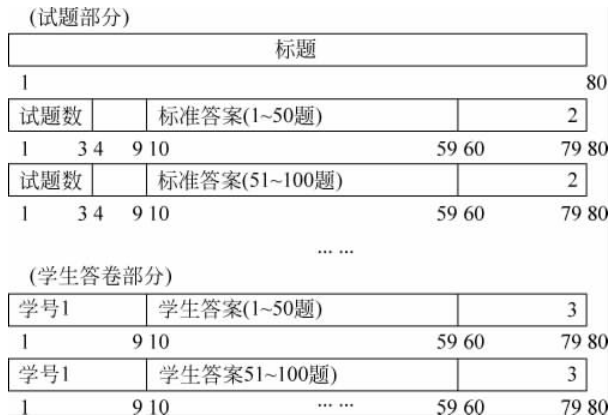


图 3-14 试卷批阅输入文件记录

(1) 标题：这一组只有一个记录，其内容为输出成绩报告的名字。

(2) 试卷各题标准答案记录：每个记录均在第 80 个字符处标以数字“2”。该组的第一个记录的第 1～3 个字符为题目编号(取值为 1～999)。第 10～59 个字符给出第 1～50 题的答案(每个合法字符表示一个答案)。该组的第 2，第 3…个记录相应为第 51～100，第 101～150，…题的答案。

输入条件及相应测试用例如表 3-9 所示。

表 3-9 输入条件及相应测试用例

输入条件	测 试 用 例
输入文件	空输入文件
标题	没有标题
	标题只有一个字符
	标题有 80 个字符
试题数	试题数为 1
	试题数为 50
	试题数为 51
	试题数为 100
	试题数为 0
	试题数含有非数字字符
标准答案记录	没有标准答案记录，有标题
	标准答案记录多于一个
	标准答案记录少一个
学生人数	0 个学生
	1 个学生
	200 个学生
	201 个学生
学生答题	某学生只有一个回答记录，但有两个标准答案记录
	该学生是文件中的第一个学生
	该学生是文件中的最后一个学生(记录数出错的学生)

续表

输入条件	测 试 用 例
学生答题	某学生有两个回答记录,但只有一个标准答案记录
	该学生是文件中的第一个学生(记录数出错的学生)
	该学生是文件中的最后一个学生
学生成绩	所有学生的成绩都相等
	每个学生的成绩都不相等
	部分学生的成绩相同
	(检查是否能按成绩正确排名次)
	有个学生 0 分
	有个学生 100 分

表 3-10 是输出条件及相应的测试用例表。

表 3-10　输出条件及相应的测试用例表

输出条件	测 试 用 例
输出报告 a、b	有个学生的学号最小(检查按序号排序是否正确)
	有个学生的号最大(检查按序号排序是否正确)
	适当的学生人数,使产生的报告刚好满一页(检查打印页数)
	学生人数比刚才多出 1 人(检查打印换页)
输出报告 c	平均成绩 100
	平均成绩 0
	标准偏差为最大值(有一半的 0 分,其他 100 分)
	标准偏差为 0(所有成绩相等)
输出报告 d	所有学生都答对了第一题
	所有学生都答错了第一题
	所有学生都答对了最后一题
	所有学生都答错了最后一题
	选择适当的试题数,是第 4 个报告刚好打满一页
	试题数比刚才多 1,使报告打满一页后,刚好剩下一题未打

(3) 每个学生的答卷描述：该组中每个记录的第 80 个字符均为数字“3”。每个学生的答卷在若干个记录中给出。如甲的首记录第 1～9 字符给出学生姓名及学号,第 10～59 字符列出的是甲所做的第 1～50 题的答案。若试题数超过 50,则第 2,第 3…记录分别给出他的第 51～100,第 101～150…题的解答。然后是学生乙的答卷记录。

(4) 学生人数不超过 200,试题数不超过 999。

(5) 程序的输出有 4 个报告：

① 按学号排列的成绩单,列出每个学生的成绩、名次。

② 按学生成绩排序的成绩单。

③ 平均分数及标准偏差的报告。

④ 试题分析报告。按试题号排序,列出各题学生答对的百分比。

解答：分别考虑输入条件和输出条件,以及边界条件。给出如表 3-9 所示的输入条件及相应的测试用例。

3）因果图法

因果图法是一种利用图解法分析输入的各种组合情况，从而设计测试用例的方法，它适合检查程序输入条件的各种组合情况。

等价类划分法和边界值分析方法都是着重考虑输入条件，但没有考虑输入条件的各种组合、输入条件之间的相互制约关系。这样虽然各种输入条件可能出错的情况已经测试到，但多个输入条件组合起来可能出错的情况却被忽视了。

如果在测试时必须考虑输入条件的各种组合，则可能的组合数目将是天文数字，因此必须考虑采用一种适合描述多种条件的组合、相应产生多个动作的形式来进行测试用例的设计，这就需要利用因果图（逻辑模型）。

图 3-15 中 4 种符号分别表示了规格说明中的 4 种因果关系。

因果图中使用了简单的逻辑符号，以直线连接左右节点。左节点表示输入状态（或称原因），右节点表示输出状态（或称结果）。C_i 表示原因，通常置于图的左部；e_i 表示结果，通常在图的右部。C_i 和 e_i 均可取值 0 或 1，0 表示某状态不出现，1 表示某状态出现。

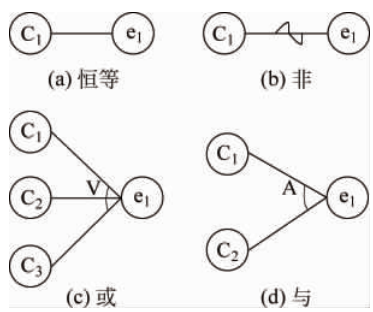


图 3-15　4 种因果关系

（1）关系

① 恒等：若 C_i 是 1，则 e_i 也是 1；否则 e_i 为 0。

② 非：若 C_i 是 1，则 e_i 是 0；否则 e_i 是 1。

③ 或：若 C_1 或 C_2 或 C_3 是 1，则 e_i 是 1；否则 e_i 为 0。“或”可有任意个输入。

④ 与：若 C_1 和 C_2 都是 1，则 e_i 为 1；否则 e_i 为 0。“与”也可有任意个输入。

（2）约束

输入状态相互之间还可能存在某些依赖关系，称为约束。例如，某些输入条件本身不可能同时出现。输出状态之间也往往存在约束。在因果图中，用特定的符号标明这些约束。

① 输入条件约束类型

- E 约束（异）：a 和 b 中至多有一个可能为 1，即 a 和 b 不能同时为 1。
- I 约束（或）：a、b 和 c 中至少有一个必须是 1，即 a、b 和 c 不能同时为 0。
- O 约束（唯一）：a 和 b 必须有一个，且仅有一个为 1。
- R 约束（要求）：a 是 1 时，b 必须是 1，即不可能 a 是 1 时 b 是 0。

② 输出条件约束类型

输出条件的约束只有 M 约束（强制）：若结果 a 是 1，则结果 b 强制为 0。

因果图输入、输出条件约束如图 3-16 所示。采用因果图法设计测试用例的步骤如下。

（1）分析软件规格说明描述中，哪些是原因（即输入条件或输入条件的等价类），哪些是结果（即输出条件），并给每个原因和结果赋予一个标识符。

（2）分析软件规格说明描述中的语义，找出原因与结果之间、原因与原因之间对应的关系，根据这些关系画出因果图。

（3）由于语法或环境限制，有些原因与原因之间，原因与结果之间的组合情况不可能出

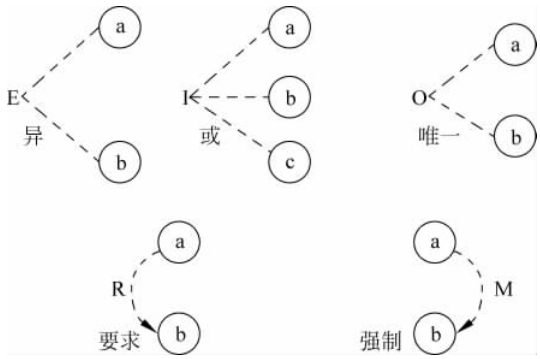


图 3-16　因果图约束

现，为表明这些特殊情况，在因果图上用一些记号表明约束或限制条件。

（4）把因果图转换为判定表。

（5）把判定表的每一列拿出来作为依据，设计测试用例。

案例：某软件规格说明书包含这样的要求，第一列字符必须是 A 或 B，第二列字符必须是一个数字，在此情况下进行文件的修改，但如果第一列字符不正确，则给出信息 L；如果第二列字符不是数字，则给出信息 M。

解答：

（1）根据题意，原因和结果如下：

原因：

1——第一列字符是 A；

2——第一列字符是 B；

3——第二列字符是一数字。

结果：

21——修改文件；

22 ——给出信息 L；

23——给出信息 M。

（2）其对应的因果图如图 3-17 所示。

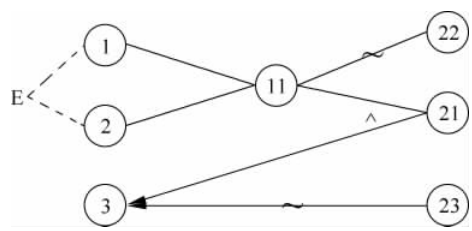


图 3-17　因果图

11 为中间节点；考虑到原因 1 和原因 2 不可能同时为 1，故在因果图上施加 E 约束。

（3）根据因果图建立判定表，如表 3-11 所示。

表 3-11　判定表

		1	2	3	4	5	6	7	8
条件(原因)	1	1	1	1	1	0	0	0	0
	2	1	1	0	0	1	1	0	0
	3	1	0	1	0	1	0	1	0
	11			1	1	1	1	0	0

续表

		1	2	3	4	5	6	7	8
动作(结果)	22			0	0	0	0	1	1
	21			1	0	1	0	0	0
	23			0	1	0	1	0	1
测试用例				A3	AM	B5	BN	C2	DY
				A8	A?	B4	B!	X6	P;

表 3-11 中 8 种情况的左面两列情况中,原因①和原因②同时为 1,这是不可能出现的,故应排除这两种情况。表的最下一栏给出了 6 种情况的测试用例,这是我们所需要的数据。

4) 错误推测方法

错误推测方法是基于经验和直觉推测程序中所有可能存在的各种错误,从而有针对性地设计测试用例的方法。

列举出程序中所有可能有的错误和容易发生错误的特殊情况,根据它们选择测试用例。

例如,输入数据和输出数据为 0 的情况;输入表格为空格或输入表格只有一行。这些都是容易发生错误的情况。可选择这些情况下的例子作为测试用例。

再如,前面例子中成绩报告的程序,采用错误推测法还可补充设计以下一些测试用例。

(1) 程序是否把空格作为回答。

(2) 在回答记录中混有标准答案记录。

(3) 除了标题记录外,还有一些记录最后一个字符既不是 2 也不是 3。

(4) 有两个学生的学号相同。

(5) 试题数是负数。

再如,测试一个对线性表(例如数组)进行排序的程序,可推测列出以下几项需要特别测试的情况。

(1) 输入的线性表为空表;

(2) 表中只含有一个元素;

(3) 输入表中所有元素已排好序;

(4) 输入表已按逆序排好;

(5) 输入表中部分或全部元素相同。

3.4 集成测试

集成测试(也叫组装测试,联合测试)是单元测试的逻辑扩展。它的最简单的形式是:两个已经测试过的单元组合成一个组件,并且测试它们之间的接口。从这一层意义上讲,组件是指多个单元的集成聚合。在现实方案中,许多单元组合成组件,而这些组件又聚合成程序的更大部分。方法是测试片段的组合,并最终扩展进程,将用户的模块与其他组的模块一起测试。最后,将构成进程的所有模块一起测试。此外,如果程序由多个进程组成,应该成对测试它们,而不是同时测试所有进程。

集成测试识别组合单元时出现的问题。通过使用要求在组合单元前测试每个单元并确保每个单元的生存能力的测试计划，可以知道在组合单元时所发现的任何错误很可能与单元之间的接口有关。这种方法将可能发生的情况数量减少到更简单的分析级别。

集成测试是在单元测试的基础上，测试在将所有的软件单元按照概要设计规格说明的要求组装成模块、子系统或系统的过程中各部分工作是否达到或实现相应技术指标及要求的活动。也就是说，在集成测试之前，单元测试应该已经完成，集成测试中所使用的对象应该是已经经过单元测试的软件单元。这一点很重要，因为如果不经过单元测试，那么集成测试的效果将会受到很大影响，并且会大幅增加软件单元代码纠错的代价。

集成测试是单元测试的逻辑扩展。在现实方案中，集成是指多个单元的聚合，许多单元组合成模块，而这些模块又聚合成程序的更大部分，如分系统或系统。集成测试采用的方法是测试软件单元的组合能否正常工作，以及与其他组的模块能否集成起来工作。最后，还要测试构成系统的所有模块组合能否正常工作。

所有的软件项目都不能摆脱系统集成这个阶段。不管采用什么开发模式，具体的开发工作总要从一个一个的软件单元做起，软件单元只有经过集成才能形成一个有机的整体。具体的集成过程可能是显性的也可能是隐性的。只要有集成，总是会出现一些常见问题，工程实践中，几乎不存在软件单元组装过程中不出任何问题的情况。

集成测试的必要性还在于一些模块虽然能够单独地工作，但并不能保证连接起来也能正常工作。程序在某些局部反映不出来的问题，有可能在全局上会暴露出来，影响功能的实现。此外，在某些开发模式中，如迭代式开发，设计和实现是迭代进行的。在这种情况下，集成测试的意义还在于它能间接地验证概要设计是否具有可行性。

集成测试的目的是确保各单元组合在一起后能够按既定意图协作运行，并确保增量的行为正确。它所测试的内容包括单元间的接口以及集成后的功能。使用黑盒测试方法测试集成的功能，并且对以前的集成进行回归测试。

1. 集成测试过程

1）软件测试过程图解

软件测试过程如图 3-18 所示。

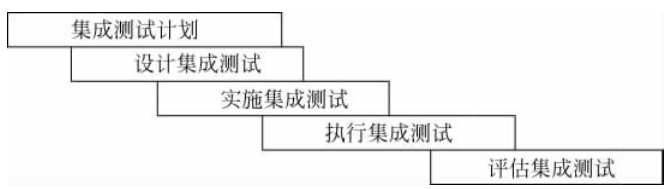


图 3-18　软件测试过程

2）集成测试需求获取

集成测试需求所确定的是对某一集成工作版本的测试的内容，即测试的具体对象。集成测试需求主要来源于设计模型(Design Model)和集成构件计划(Integration Build Plan)。集成测试着重于集成版本的外部接口的行为。因此，测试需求需具有可观测、可测评性。

(1) 集成工作版本应分析其类协作与消息队列，从而找出该工作版本的外部接口。

(2) 由集成工作版本的外部接口确定集成测试用例。

(3) 测试用例应覆盖工作版本每一外部接口的所有消息流序列。

注意：一个外部接口和测试用例的关系是多对多，部分集成工作版本的测试需求可映射到系统测试需求，因此对这些集成测试用例可采用重用系统测试用例技术。

3) 集成测试工作机制

软件集成测试工作需要产品评测部担任，需要项目组相关角色配合完成。具体角色和职责如表 3-12 和表 3-13 所示。

表 3-12 评测部的角色和职责

角色	职 责
测试设计员	负责制定集成测试计划、设计集成测试、实施集成测试、评估集成测试
测试员	执行集成测试，记录测试结果

表 3-13 软件项目组的角色和职责

角色	职 责
实施员	负责实施类(包括驱动程序和桩)，并对其进行单元测试。根据集成测试发现的缺陷提出变更申请
配置管理员	负责对测试工件进行配置管理
集成员	负责制定集成构建计划，按照集成计划将通过了单元测试的类集成
设计员	负责设计测试驱动程序和桩。根据集成测试发现的缺陷提出变更申请

集成测试工作内容及其工作流程如图 3-19 所示。

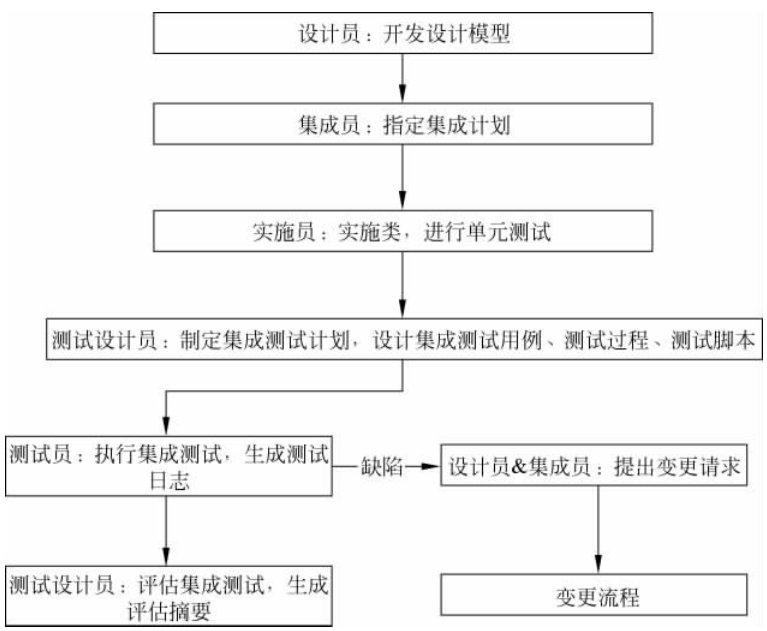


图 3-19 集成测试工作流程

4）集成测试产生的工作清单

（1）软件集成测试计划。

（2）集成测试用例。

（3）测试过程。

（4）测试脚本。

（5）测试日志。

（6）测试评估摘要。

2. 集成测试常用方案选型

集成测试的实施方案有很多种，如自底向上集成测试、自顶向下集成测试、Big-Bang 集成测试、三明治集成测试、核心集成测试、分层集成测试、基于使用的集成测试等。下面是实践中证实有效的集成测试方案。

1）自底向上集成测试

自底向上的集成(Bottom-Up Integration)方式是最常使用的方法。其他集成方法都或多或少地继承、吸收了这种集成方式的思想。自底向上集成方式从程序模块结构中最底层的模块开始组装和测试。因为模块是自底向上进行组装的，对于一个给定层次的模块，它的子模块(包括子模块的所有下属模块)事先已经完成组装并经过测试，所以不再需要编制桩模块(一种能模拟真实模块，给待测模块提供调用接口或数据的测试用软件模块)。自底向上集成测试的步骤大致如下。

步骤一：按照概要设计规格说明，明确有哪些被测模块。在熟悉被测模块性质的基础上对被测模块进行分层，在同一层次上的测试可以并行进行，然后排出测试活动的先后关系，制定测试进度计划。图 3-20 给出了自底向上的集成测试过程中各测试活动的拓扑关系。利用图论的相关知识，可以排出各活动之间的时间序列关系，处于同一层次的测试活动可以同时进行，而不会相互影响。

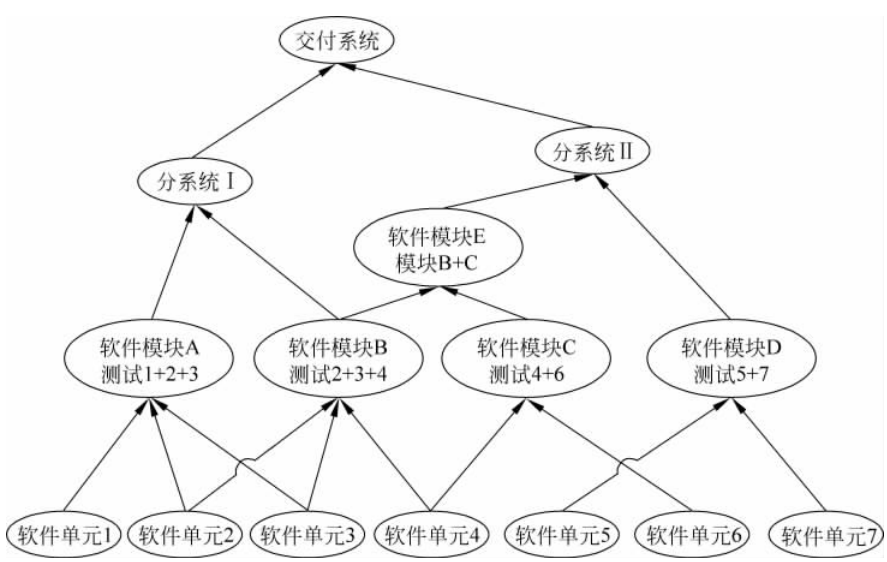


图 3-20　测试方案选型

步骤二：在步骤一的基础上，按时间线序关系，将软件单元集成为模块，并测试在集成过程中出现的问题。这里，可能需要测试人员开发一些驱动模块来驱动集成活动中形成的被测模块。对于比较大的模块，可以先将其中的某几个软件单元集成为子模块，然后再集成为一个较大的模块。

步骤三：将各软件模块集成为子系统(或分系统)。检测各自子系统是否能正常工作。同样，可能需要测试人员开发少量的驱动模块来驱动被测子系统。

步骤四：将各子系统集成为最终用户系统，测试是否存在各分系统能否在最终用户系统中正常工作。

方案点评：自底向上的集成测试方案是工程实践中最常用的测试方法。相关技术也较为成熟。它的优点很明显：管理方便，测试人员能较好地锁定软件故障所在位置。但它对于某些开发模式不适用，如使用 XP 开发方法，它会要求测试人员在全部软件单元实现之前完成核心软件部件的集成测试。尽管如此，自底向上的集成测试方法仍不失为一个可供参考的集成测试方案。

2）核心系统先行集成测试

核心系统先行集成测试法的思想是先对核心软件部件进行集成测试，在测试通过的基础上再按各外围软件部件的重要程度逐个集成到核心系统中。每次加入一个外围软件部件都产生一个产品基线，直至最后形成稳定的软件产品。核心系统先行集成测试法对应的集成过程是一个逐渐趋于闭合的螺旋形曲线，代表产品逐步定型的过程。其步骤如下。

步骤一：对核心系统中的每个模块进行单独的、充分的测试，必要时使用驱动模块和桩模块。

步骤二：对于核心系统中的所有模块一次性集合到被测系统中，解决集成中出现的各类问题。在核心系统规模相对较大的情况下，也可以按照自底向上的步骤，集成核心系统的各组成模块。

步骤三：按照各外围软件部件的重要程度以及模块间的相互制约关系，拟定外围软件部件集成到核心系统中的顺序方案。方案经评审以后，即可进行外围软件部件的集成。

步骤四：在外围软件部件添加到核心系统以前，外围软件部件应先完成内部的模块级集成测试。

步骤五：按顺序不断加入外围软件部件，排除外围软件部件集成中出现的问题，形成最终的用户系统。

方案点评：该集成测试方法对于快速软件开发很有效果，适合较复杂系统的集成测试，能保证一些重要的功能和服务的实现。缺点是采用此法的系统一般应能明确区分核心软件部件和外围软件部件，核心软件部件应具有较高的耦合度，外围软件部件内部也应具有较高的耦合度，但各外围软件部件之间应具有较低的耦合度。

3）高频集成测试

高频集成测试是指同步于软件开发过程，每隔一段时间对开发团队的现有代码进行一次集成测试。如某些自动化集成测试工具能实现每日深夜对开发团队的现有代码进行一次集成测试，然后将测试结果发到各开发人员的电子邮箱中。该集成测试方法频繁地将新代码加入到一个已经稳定的基线中，以免集成故障难以发现，同时控制可能出现的基线偏差。使用高频集成测试需要具备一定的条件：可以持续获得一个稳定的增量，并且该增量内部

已被验证没有问题；大部分有意义的功能增加可以在一个相对稳定的时间间隔（如每个工作日）内获得；测试包和代码的开发工作必须是并行进行的，并且需要版本控制工具来保证始终维护的是测试脚本和代码的最新版本；必须借助于使用自动化工具来完成。高频集成一个显著的特点就是集成次数频繁，显然，人工的方法是不胜任的。

高频集成测试一般采用如下步骤来完成。

步骤一：选择集成测试自动化工具。如很多 Java 项目采用 JUnit＋Ant 方案来实现集成测试的自动化，也有一些商业集成测试工具可供选择。

步骤二：设置版本控制工具，以确保集成测试自动化工具所获得的版本是最新版本。如使用 CVS 进行版本控制。

步骤三：测试人员和开发人员负责编写对应程序代码的测试脚本。

步骤四：设置自动化集成测试工具，每隔一段时间对配置管理库的新添加的代码进行自动化的集成测试，并将测试报告汇报给开发人员和测试人员。

步骤五：测试人员监督代码开发人员及时关闭不合格项。

按照步骤三至步骤五不断循环，直至形成最终软件产品。

方案点评：该测试方案能在开发过程中及时发现代码错误，能直观地看到开发团队的有效工程进度。在此方案中，开发维护源代码与开发维护软件测试包被赋予了同等的重要性，这对有效防止错误、及时纠正错误都很有帮助。该方案的缺点在于测试包有时候可能不能暴露深层次的编码错误和图形界面错误。

以上介绍了几种常见的集成测试方案，一般来讲，在现代复杂软件项目集成测试过程中，通常采用核心系统先行集成测试和高频集成测试相结合的方式进行，自底向上的集成测试方案在采用传统瀑布式开发模式的软件项目集成过程中较为常见。实践中应该结合项目的实际工程环境及各测试方案适用的范围进行合理的选型。

3.5　系统测试

系统测试是将已经确认的软件、计算机硬件、外设、网络等其他元素结合在一起，进行信息系统的各种组装测试和确认测试，系统测试是针对整个产品系统进行的测试，目的是验证系统是否满足了需求规格的定义，找出与需求规格不符或与之矛盾的地方，从而提出更加完善的方案。系统测试发现问题之后要经过调试找出错误原因和位置，然后进行改正。基于系统整体需求说明书的黑盒类测试，应覆盖系统所有联合的部件。对象不仅包括需测试的软件，还要包含软件所依赖的硬件、外设甚至包括某些数据、某些支持软件及其接口等。

1. 非功能性测试

1）安全性测试

软件安全性轻则造成操作的不方便，重则造成数据的破坏或丢失甚至系统的崩溃和人身的安全，因此，软件安全性是一个不容忽视的重要问题。我们可以简单地把软件的安全性作为一个或多个特定的功能来考虑，从而在软件生命周期的早期就加以考虑。

为了帮助设计一个安全的信息系统，在产品设计的最开始就必须注意安全的问题，例如需求中应有安全性的相关项目、设计和代码评审应有专门针对安全性的内容等，然后才是测试。

测试员仅能测试验证软件的安全性。当然,对于没有在软件需求书上标明的可能影响系统运行安全的隐性需求测试人员也要努力地发现,这也是一个有经验的安全性测试人员的可贵之处。

当然,理论上没有任何一个信息系统是安全的,因为只要进行攻击,任何系统都能被攻破,只不过付出的代价的大小不同。而一般说某个信息系统是安全的就是基于如果要攻破该系统所必须付出的代价要高于或远远高于攻破系统后获得的利益。

软件安全性测试策略如表 3-14 所示。

表 3-14 软件安全性测试策略

测试目标	测试应用或系统的安全机制,保证系统运行和使用的安全性
测试策略	采取静态分析技术和功能测试两种方式拦截系统开发时存在的漏洞 软件生命周期早期的代码、设计评审(由开发人员完成)对软件安全性的提高非常有效和重要,可以人工评审和自动化工具结合的方法进行
测试重点考虑	• 相关信息安全法律法规的要求 • 测试环境 • 网络安全 • 日志评审 • 文件完整性检查 • 系统软件安全 • 客户端应用安全 • 客户端到服务端应用通信安全 • 服务端应用安全
测试通过准则	应用满足和符合采用的安全机制,在应用规定的程度上能保证系统正常运行和安全使用

软件安全性测试包括应用软件、网络系统、数据库和系统软件安全性测试。根据系统安全指标不同测试策略也不同。

(1) 用户认证安全的测试要考虑问题:

① 系统中是否有不同的用户使用权限;

② 系统会不会因用户的权限的改变造成混乱;

③ 系统中会不会出现用户冲突;

④ 用户登录密码是否可见、可复制;

⑤ 是否可以通过绝对路径登录系统(复制用户登录后的链接直接进入系统);

⑥ 用户退出系统后是否删除了所有鉴权标记,是否可以使用后退键而不通过输入口令进入系统。

(2) 应用方面安全的测试要考虑问题:

① 缓冲区溢出;

② 无效的数据类型;

③ 关键信息是否采用加密技术;

④ 是否可以通过绝对路径进入系统;

⑤ 使用的端口号;

⑥ 远程进入服务(如有);

⑦ 文件完整性检查;

⑧ 日志评审；

⑨ 模拟黑客攻击。

(3) 系统网络安全的测试要考虑问题：

① 物理连接上的安全，包括无线部分(如果有)；

② 防火墙、防病毒软件的安全；

③ 测试采取的防护措施是否正确装配好，有关系统的补丁是否打上；

④ 模拟非授权攻击，看防护系统是否坚固；

⑤ 采用成熟的网络漏洞检查工具检查系统相关漏洞；

⑥ 入侵网络监察系统和防护系统；

⑦ 采用各种木马检查工具检查系统木马情况；

⑧ 采用各种防外挂工具检查系统各组程序的外挂漏洞。

(4) 数据库安全考虑问题：

① 系统数据是否机密；

② 系统数据的完整性；

③ 系统数据可管理性；

④ 系统数据的独立性；

⑤ 系统数据可备份和恢复能力(数据备份是否完整，可否恢复，恢复是否完整)。

(5) 系统级别软件安全考虑问题：

① 系统级别软件(如操作系统、数据库系统和中间件系统等)是否是最新的，尤其如果使用开源软件或免费软件；

② 系统软件是否有相应的补丁，尤其是安全性方面的补丁包；

③ 从安全性考虑，系统级别软件是否是最可靠的(如使用 Tomcat 还是 WebLogic)；

④ 从安全性考虑，系统级别软件是否匹配应用设计技术。

2) 安装测试

安装测试有两个目的。第一个目的是确保软件能够在不同的条件下进行安装，例如，首次安装、升级安装、完整或自定义安装；在正常和异常条件下的安装。异常条件例如磁盘空间不足、安装用户缺少目录创建权限等。第二个目的是验证软件在安装后能正确操作。这通常意味着要运行大量为功能测试开发的测试用例。

建议对于安装测试，其重点应该放在第一个目的上，对于第二个目的可以根据需要和实际情况穿插在后续的测试，如冒烟测试、功能测试中体现。

软件安装测试策略如表 3-15 所示。

表 3-15　软件安装测试策略

测试目标	验证被测软件在各种需要的软硬件配置下正确安装，包括下列情况： • 新安装。新的机器，之前从未安装过该软件 • 更新。该机器之前安装过该软件同样的版本 • 更新。该机器之前安装过该软件的老版本
测试策略	手工或自动化 不管是手工安装还是自动化脚本安装，都要严格按照用户安装手册规定的操作步骤进行

续表

测试重点考虑	有没有安装手册 对于第二个目的的安装测试，怎样选取预先规定的功能测试的子集
测试通过准则	对于第一个目的：确保软件能够在不同的条件下进行安装。成功的安装应该是在整个安装过程中和安装完成后没有任何异常错误出现，能成功登录该应用 对于第二个目的：验证软件在安装后能正确操作。这个需要运行预先规定的一功能测试的子集

3）配置和兼容性测试

配置和兼容性测试验证被测软件在不同的软硬件配置下的操作和表现。

在大多数生产环境中，特定的硬件规格，如客户端工作站、网络情况、服务器配置甚至型号都会不同。另外，不同客户端工作站上也许有不同的软件负载，如不同的应用软件、不同的浏览器、浏览器的不同版本的插件、驱动程序等；甚至不同的操作系统，如 Windows XP，Windows Vista 等；服务器软件也会不一致，如数据库软件，如 Oracle，Microsoft SQL Server、中间件软件等。

该类测试对产品显得尤其重要，对于项目来说也有巨大的指导作用，如项目经理为了最优化配置（从成本和实用的角度等），就需要进行不同软硬件情况下的配置和兼容性测试。该类测试也往往和性能测试相结合以决定在不同软硬件情况下的最优化配置（如从成本和实用的角度等）。

软件配置和兼容性测试策略如表 3-16 所示。

表 3-16　软件配置和兼容性测试策略

测试目标	验证被测软件在各种需要的软硬件配置下能正确运行
测试策略	使用功能测试脚本。首先，针对特定硬件配置下： • 打开和关闭许多和被测软件无关的、但和被测软件共同在同一机器上的其他许多软件，如同机上的 Microsoft 应用软件，Excel 或 Word 等（如果有），可以在测试过程中和测试前就打开 • 在此基础上执行相应的功能测试
测试重点考虑	硬件配置情况复杂多样，使用哪些硬件配置呢？建议最好做些调研，如考虑目标用户或最终用户的使用习惯和经济承受能力 打开哪些和被测软件无关的、但和被测软件共同在同一机器上的其他许多软件？建议最好做些调研，如考虑目标用户或最终用户会使用哪些其他的常用软件？尤其在使用目标软件时往往同时会打开或使用的软件
测试通过准则	在某一特定的硬件配置下，在许多和被测软件无关的、但和被测软件共同在同一机器上的其他许多软件运行或使用情况下，相应的功能测试没有失败

4）易用性测试

易用性是人类工程学的目标。软件的易用性是一个比较有特点的问题，会随着具体产品或项目的特征和要求而有巨大差异，例如，手机软件和一般 Windows 平台下的软件易用性相差很大，又如一核反应堆的关闭序列和一语音信箱的菜单系统的易用性有着天壤之别。即使对于同一个软件，不同的用户也会有不同的感受。当然，也不是说对软件的易用性就毫无测试办法和标准，对于同一类软件还是有它的通用性可言的。因此下面尝试从一些比较通用的方面讨论它。

易用性主要考虑软件使用时人的因素和感受，因此先介绍两个概念。用户与程序相互交互的介质称为用户接口（User Interface，UI），用户接口对不同软件种类也会有巨大的差异，对于 PC 来说，现在有了精致的完善的图形用户接口（Graphical User Interface，GUI）。很快人们就可以对着 PC 听和说，那时又会有新的 UI。

另外，Accessibility Testing 也属于易用性测试范畴。这种测试一般针对对软件有特殊要求的群体，例如部分残疾人。

软件易用性测试策略如表 3-17 所示。

表 3-17　软件易用性测试策略

测试目标	考虑软件使用时人的因素和感受
测试策略	使用功能测试脚本 对于一些比较通用的规则可以做成检查表的形式
测试重点考虑	• 遵守通用的标准和指导方针。如 Windows 平台下的软件应和 Windows 平台的风格一致，而 Mac 平台下的软件显然应该和 Mac 平台的风格一致 • 直觉的。如用户界面是干净的、不多余的，用户界面是经过良好组织和布局的 • 一致性。如快捷键和菜单选项间；术语和命名；“确认”和“取消”按钮的位置布局等 • 灵活性。如用户希望能手工输入、粘贴、插入文件内容等方式作为输入方法 • 舒服的。如较长时间的等待应有进度条提示，一般情况下程序应有较快的反应。但也不是绝对的，举个例子，高性能就一定好吗？未必如此，例如当快速到连提示信息或出错信息都不能看清时用户反而会觉得很不舒服 • 正确的。例如拼写；图标是否有同样的大小和背景；还有是否所见即所得（What You See Is What You Get，WYSIWYG） • 有用的。没有大量的过量功能，这个特征尤其在产品上表现突出，用户往往要面对大量的无用的或者不需要的功能
测试通过准则	该类测试对于不同产品和项目有较大的差异性，因此尤其要加强对该类测试用例的评审，在业务人员、设计人员等一致同意的基础上进行测试，测试人员也要关注软件需求中应有该方面的相关描述

5）数据和数据库完整性测试

在整个系统测试过程中，数据库和数据库过程应作为系统的子系统进行测试。这些测试不使用用户界面作为数据进入界面进行测试。对不同的数据库管理系统，可能存在不同的工具和技术，如 Oracle 和 Microsoft SQL Server。

软件数据和数据库完整性测试策略如表 3-18 所示。

表 3-18　软件数据和数据库完整性测试策略

测试目标	保证数据存取方法和过程正确运行，在整个数据操作过程中和结束后没有数据破坏
测试策略	使用有效的和无效的数据或数据的请求调用各个数据库存取方法和过程 检查数据库保证数据正如预期地进入，所有的数据库事件正确地发生，或者检查数据保证对于正确的请求取得正确的数据
测试重点考虑	测试也许需要一个 DBMS 开发环境或驱动程序以便在数据库中直接进入或更改数据 确定哪些数据存取方法和数据过程重点测试 测试的先后顺序
测试通过准则	所有的数据库存取方法和过程正如设计要求一样运行，没有任何的数据破坏

6）接口测试

在集成测试过程中，接口测试显得特别重要，首先要确认被测软件是否集成了规定的单元或子系统、外部系统（如果有）；其次，对相应的接口要进行详细的测试；最后也应该对集成后的所有功能进行相应的测试以确保集成后整个系统功能的正确性。

软件接口测试策略如表 3-19 所示。

表 3-19　软件接口测试策略

测试目标	确保“测试需求”中对应的所有工作版本的内部单元组合到一起后能够按照设计的意图协作运行，接口的调用正确
测试策略	使用功能测试脚本 按照设计规范，对所有接口设计相应的测试用例进行手工或自动化测试，包括系统内部接口和外部接口，尤其注意和第三方的软硬件接口（如果有）
测试重点考虑	测试的顺序性 数据是否能正确传递 接口之间的调用 状态的变化是否正确（如果有） 接口之间的异常处理功能
测试通过准则	所有与接口有关的测试用例功能没有失败

7）文档测试

文档也是发布的软件产品的重要组成部分，因此也应该对文档进行相应的测试，尤其对安装手册（前面已有所述）、用户使用手册和在线帮助手册要进行重点测试。正确的文档能减少维护费用和提高软件的可维护性、改善软件易用性、减少责任等。

软件文档测试策略如表 3-20 所示。

表 3-20　软件文档测试策略

测试目标	保证发布的软件产品中的文档的正确性
测试策略	按照手册描述，逐章节阅读、检查和验证相应的内容，例如，严格地按照用户使用手册上的描述去操作和使用程序
测试重点考虑	与用户相关的文档的测试（如安装手册、用户使用手册和在线帮助手册） 术语的正确性和一致性 是否覆盖了产品的所有方面 功能描述的准确性 图形和屏幕截屏是否正确 举例是否正确 拼写和语法 超文本链接
测试通过准则	文档所有内容描述准确和正确

8）失效恢复测试

失效恢复测试保证被测软件能成功地从硬件、软件或网络故障中失效恢复而不会有数据或事务的丢失。

对于必须时刻保持运行的系统，失效备援测试保证当一个失效备援条件发生时，替代或

备份系统正确地接管失败的系统而不会有数据或事务的丢失。

恢复测试是一种对立的测试过程，让应用或系统遭受极端或仿真条件来引起一个故障，例如设备的输入、输出错误或无效的数据库指针和键。调用恢复过程、监控和检查应用或系统来验证已完成正确的应用、系统和数据恢复。

软件失效恢复测试策略如表 3-21 所示。

表 3-21　软件失效恢复测试策略

测试目标	验证恢复过程(手工或自动化)正确恢复数据库、应用和系统到期望的、已知的状态
测试策略	手工或自动化测试 如电源中断、通信中断、网络中断、操作中断等可以手工直接执行；操作中断，尤其是有关数据库的操作、网络操作等也可以使用一些自动化工具执行
测试重点考虑	测试设计或执行中应考虑： • 客户端电源中断 • 服务端电源中断 • 客户端和服务器之间通信中断 • 网络异常问题，如瞬间的断开 • 关键功能执行过程中操作异常中断 • 备份系统有效性(如果有) • 数据过滤过程中断、数据同步过程中断等 • 无效的数据库指针或键 • 数据库中无效的或被破坏的数据元素
测试通过准则	在上面所有的测试中，当恢复程序完成时，应用、数据库和系统应该成功返回到期望的、已知的状态

2. 性能测试

提到软件性能测试的时候，有一点是很明确的：测试关注的重点是“性能”。那么，本书要解决的第一个问题就是：究竟什么是“软件性能”？

一般来说，性能是一种指标，表明软件系统或构件对于其及时性要求的符合程度；其次，性能是软件产品的一种特性，可以用时间来进行度量。

性能的及时性用响应时间或者吞吐量来衡量。响应时间是对请求做出响应所需要的时间。

对于单个事务，响应时间就是完成事务所需的时间；对于用户任务，响应时间体现为端到端的时间。例如，“用户单击 OK 按钮后两秒内收到结果”就是一个对用户任务响应时间的描述，具体到这个用户任务中，可能有多个具体的事务需要完成，每个事务都有其单独的响应时间。

对交互式的应用(例如典型的 Web 应用)来说，一般以用户感受到的响应时间来描述系统的性能，而对非交互式应用(嵌入式系统或是银行等的业务处理系统)而言，响应时间是指系统对事件产生响应所需要的时间。

通常，对软件性能的关注是多个层面的：用户关注软件性能，管理员关注软件性能，产品的开发人员也关注软件性能，那么这些不同的关注者所关注的“性能”的具体内容是不是都完全相同呢？如果不同，这些不同又在哪里？最后，作为软件性能测试工程师，不同层面

的软件性能都需要关注，在关注全部这些层面的性能体现的时候，又应该注意哪些内容呢？下面从三个不同层面来对软件性能进行阐述。

1）用户视角的软件性能

从用户的角度来说，软件性能就是软件对用户操作的响应时间。说得更明确一点，对用户来说，当用户单击一个按钮、发出一条指令或是在 Web 页面上单击一个链接，从用户单击开始到应用系统把本次操作的结果以用户能察觉的方式展示出来，这个过程所消耗的时间就是用户对软件性能的直观印象。图 3-21 以一个 Web 系统为例，说明了用户的这种印象。

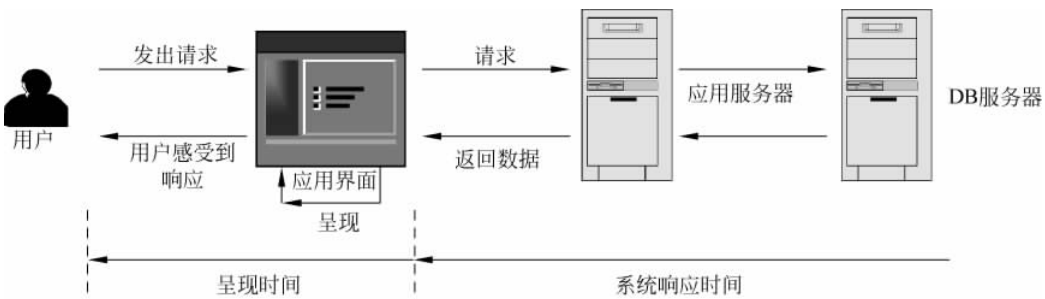


图 3-21　Web 系统的响应

必须要说明的是，用户所体会到的“响应时间”既有客观的成分，也有主观的成分。例如，用户执行了某个操作，该操作返回大量数据，从客观的角度来说，事务的结束应该是系统返回所有的数据，响应时间应该是从用户操作开始到所有数据返回完成的整个耗时；但从用户的主观感知来说，如果采用一种优化的数据呈现策略，当少部分数据返回之后就立刻将数据呈现在用户面前，则用户感受到的响应时间就会远远小于实际的事务响应时间（顺便说一下，这种技巧是在 C/S 结构的管理系统中开发人员常用的一种技巧）。

2）管理员视角的软件性能

从管理员的角度来看，软件系统的性能首先表现在系统的响应时间上，这一点和用户视角是一样的。但管理员是一种特殊的用户，和一般用户相比，除了会关注一般用户的体验之外，他还会关心和系统状态相关的信息。例如，管理员已经知道，在并发用户数为 100 时，A 业务的响应时间为 8s，那么此时的系统状态如何呢？服务器的 CPU 使用是不是已经达到了最大值？是否还有可用的内存？应用服务器的状态如何？设置的 JVM 可用内存是否足够？数据库的状况如何？是否还需要进行一些调整？这些问题普通的用户并不关心，因为这不在他们的体验范围之内；但对管理员来说，要保证系统的稳定运行和持续的良好性能，就必须关心这些问题。

另一方面，管理员还会想要知道系统具有多大的可扩展性，处理并发的能力如何；而且，管理员还会希望知道系统可能的最大容量是什么，系统可能的性能瓶颈在哪里，通过更换哪些设备或是进行哪些扩展能够提高系统性能，了解这些情况，管理员才能根据系统的用户状况制定管理措施，在系统出现计划之外的用户增长等紧急情况的时候能够立即制定相应措施，进行迅速的处理；此外，管理员可能还会关心系统在长时间的运行中是否足够稳定，是否能够不间断地提供业务服务等。

因此，从管理员的视角来看，软件性能绝对不仅仅是应用的响应时间这么一个简单的

问题。

表 3-22 给出了管理员关注的部分性能相关问题的列表。

表 3-22　管理员关注的部分性能相关问题

管理员关心的问题	软件性能描述
服务器的资源使用状况合理吗	资源利用率
应用服务器和数据库的资源使用状况合理吗	资源利用率
系统是否能够实现扩展	系统可扩展性
系统最多能支持多少用户的访问？系统最大的业务处理量是多少	系统容量
系统性能可能的瓶颈在哪里	系统可扩展性
更换哪些设备能够提高系统性能	系统可扩展性
系统能否支持 7×24h 的业务访问	系统稳定性

3）开发视角的软件性能

从开发人员的角度来说，对软件性能的关注就更加深入了。开发人员会关心主要的用户感受——响应时间，因为这毕竟是用户的直接体验；另外，开发人员也会关心系统的扩展性等管理员关心的内容，因为这些也是产品需要面向的用户（特殊的用户）。但对开发人员来说，其最想知道的是“如何通过调整设计和代码实现，或是如何通过调整系统设置等方法提高软件的性能表现”，以及“如何发现并解决软件设计和开发过程中产生的由于多用户访问引起的缺陷”，因此，其最关注的是使性能表现不佳的因素和由于大量用户访问引发的软件故障，也就是人们通常所说的“性能瓶颈”和系统中存在的在大量用户访问时表现出来的缺陷。

举例来说，对于一个没有达到预期性能规划的应用，开发人员最想知道的是，这个糟糕的性能表现究竟是由于系统架构选择的不合理还是由于代码实现的问题引起？由于数据库设计的问题引起？抑或是由于系统的运行环境引发？

或者，对于一个即将发布到现场给用户使用的应用，开发人员可能会想要知道当大量用户访问这个系统时，系统会不会出现某些故障，例如，是否存在由于资源竞争引起的挂起？是否存在由于内存处理等问题引起的系统故障？

因此，对开发人员来说，单纯获知系统性能“好”或者“不好”的评价并没有太大的意义，他们更想知道的是“哪些地方是引起不好的性能表现的根源”或是“哪里可能存在故障发生的可能”。

表 3-23 给出了开发视角的软件性能关注内容。

表 3-23　开发人员关注的性能问题

开发人员关心的问题	问题所属层次
架构设计是否合理	系统架构
数据库设计是否存在问题	数据库设计
代码是否存在性能方面的问题	代码
系统中是否有不合理的内存使用方式	代码
系统中是否存在不合理的线程同步方式	设计与代码
系统中是否存在不合理的资源竞争	设计与代码

4）SEI 负载测试计划过程

SEI 负载测试计划过程(SEI Load Testing Planning Process)是一个关注于负载测试计划的方法，其目标是产生“清晰、易理解、可验证的负载测试计划”。SEI 负载测试计划过程包括 6 个关注的区域(Area)：目标、用户、用例、生产环境、测试环境和测试场景。

SEI 负载测试计划过程将以上述 6 个区域作为负载测试计划需要重点关注和考虑的内容，其重点关注以下几个方面的内容。

(1) 生产环境与测试环境的不同。由于负载测试环境与实际的生产环境存在一定的差异，因此，在测试环境上对应用系统进行的负载测试结果很可能不能准确反映该应用系统在生产环境上的实际性能表现，为了规避这个风险，必须仔细设计测试环境。

(2) 用户分析。用户是对被测应用系统性能表现最关注和受影响最大的对象，因此，必须通过对用户行为进行分析，依据用户行为模型建立用例和场景。

(3) 用例。用例是用户使用某种顺序和操作方式对业务过程进行实现的过程，对负载测试来说，用例的作用主要在于分析和分解出关键的业务，判断每个业务发生的频度、业务出现性能问题的风险等。

从 SEI 负载测试计划过程的描述中可以看到，SEI 负载测试计划过程给出了负载测试需要关注的重点区域，但严格来说，其并不能被称为具体的方法论，因为其仅给出了对测试计划过程的一些关注内容，而没有能够形成实际的可操作的过程。

同功能测试一样，性能测试也必须经历测试需求、测试设计、测试执行、测试分析等阶段，但由于性能测试自身的特殊性(例如，需要引入工具，分析阶段相对重要)，性能测试过程又不能完全套用功能测试过程。

SEI 负载测试计划过程在负载测试需要关注的具体内容上提供了参考，但其并不是一个完整的测试过程。

5）RBI 方法

RBI(Rapid Bottleneck Identify)方法是 Empirix 公司提出的一种用于快速识别系统性能瓶颈的方法。该方法基于以下一些事实。

(1) 发现的 80％系统的性能瓶颈都由吞吐量制约；

(2) 并发用户数和吞吐量瓶颈之间存在一定的关联；

(3) 采用吞吐量测试可以更快速定位问题。

RBI 方法首先访问服务器上的“小页面”和“简单应用”，从应用服务器、网络等基础的层次上了解系统吞吐量表现；其次选择不同的场景，设定不同的并发用户数，使其吞吐量保持基本一致的增长趋势，通过不断增加并发用户数和吞吐量，观察系统的性能表现。

在确定具体的性能瓶颈时，RBI 将性能瓶颈的定位按照一种“自上而下”的分析方式进行分析，首先确定是由并发还是由吞吐量引发的性能表现限制，然后从网络、数据库、应用服务器和代码本身 4 个环节确定系统性能具体的瓶颈。

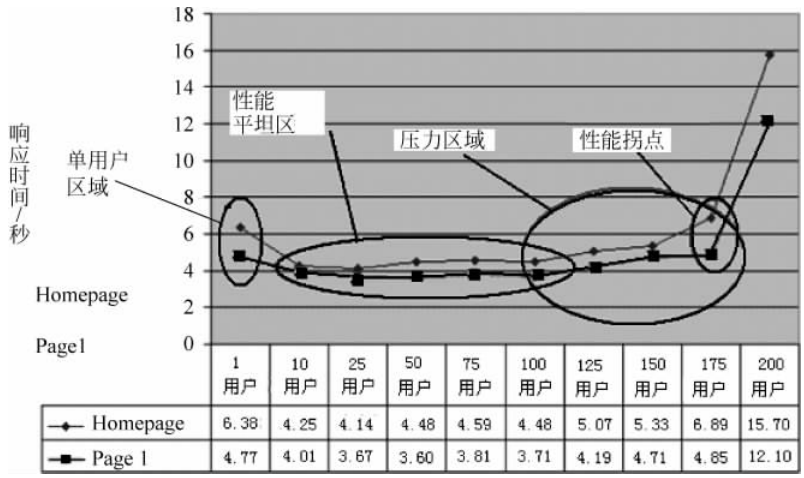
RBI 方法在性能瓶颈的定位过程中能发挥良好的作用，其对性能分析和瓶颈定位的方法值得借鉴，但其也不是完整的性能测试过程。

6）性能下降曲线分析法

性能下降曲线实际上描述的是性能随用户数增长而出现下降趋势的曲线。而这里所说的“性能”可以是响应时间，也可以是吞吐量或是单击数/秒的数据。当然，一般来说，“性能”

主要是指响应时间。

图 3-22 给出了一个“响应时间下降曲线”的示例。



	1用户	10用户	25用户	50用户	75用户	100用户	125用户	150用户	175用户	200用户
Homepage	6.38	4.25	4.14	4.48	4.59	4.48	5.07	5.33	6.89	15.70
Page 1	4.77	4.01	3.67	3.60	3.81	3.71	4.19	4.71	4.85	12.10

图 3-22　一条典型的响应时间性能下降曲线示例

从图 3-22 可以看到，一条曲线可以分为以下几个部分。

(1) 单用户区域——对系统的一个单用户的响应时间。这对建立性能的参考值很有作用。

(2) 性能平坦区——在不进行更多性能调优情况下所能期望达到的最佳性能。这个区域可被用作基线或是 benchmark。

(3) 压力区域——应用“轻微下降”的地方。典型的、最大的建议用户负载是压力区域的开始。

(4) 性能拐点——性能开始“急剧下降”的点。

这几个区域实际上明确标识了系统性能最优秀的区间，系统性能开始变坏的区间，以及系统性能出现急剧下降的点。对性能测试来说，找到这些区间和拐点，也就可以找到性能瓶颈产生的地方。

因此，对性能下降曲线分析法来说，主要关注的是性能下降曲线上的各个区间和相应的拐点，通过识别不同的区间和拐点，从而为性能瓶颈识别和性能调优提供依据。

7) Segue 提供的性能测试过程

图 3-23 给出了 Segue 公司 Silk Performer 提供的性能测试过程。该性能测试过程是一个不断 try-check 的过程。

Silk Performer 提供的性能测试过程从确定性能基线开始，通过单用户对应用的访问获取性能取值的基线，然后设定可接受的性能目标(响应时间)，用不同的并发用户数等重复进行测试。

Segue 提供的这种性能测试方法非常适合性能调优和性能优化，通过不断重复的 try-check 过程，可以逐一找到可能导致性能瓶颈的地方并对其进行优化。

但 Segue 提供的这个性能测试过程模型存在与 LoadRunner 的性能测试过程同样的问题，就是过于依赖工具自身，另外，该过程模型缺乏对计划、设计的阶段的明确划分，也没有给出具体的活动和目标。

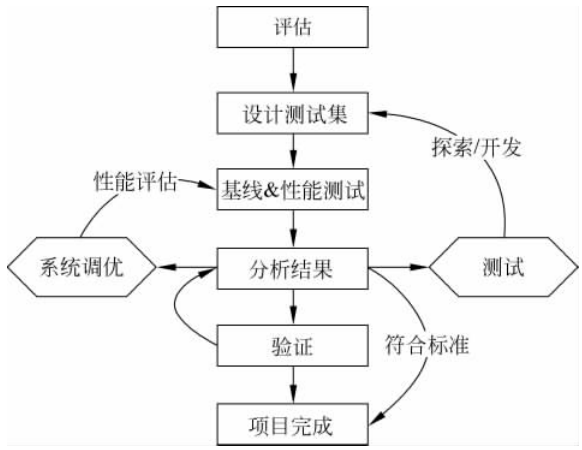


图 3-23 Segue Silk Performer 提供的性能测试过程

思考题

1. 软件测试过程模型有哪几种？它们之间有什么区别？
2. 软件生命周期一般分为哪几个阶段？
3. 常见的软件开发模型有哪几种？各有什么特点？
4. 单元测试、集成测试和系统测试的主要工作内容是什么？
5. 请设计一个单元测试用例。
6. 进行软件安全性测试的策略是什么？