第5章　单元测试

单元测试在工作中到处存在。例如，工厂在组装一台电视机之前，对每个元器件都要进行测试；一辆汽车的零部件有上万个，任何一个零部件存在质量问题，组装起来的汽车就会存在质量问题。这种对每个零件进行的测试，就是单元测试。

经常与单元测试联系起来的另外一些开发活动包括代码走读(Code Walkthrough)、静态分析(Static Analysis)和动态分析(Dynamic Analysis)。静态分析就是对软件的源代码进行研读，查找错误或收集一些度量数据，并不需要对代码进行编译和执行。动态分析，是通过观察软件运行时的动作来提供跟踪、时间分析以及测试覆盖度方面的信息。

5.1　单元测试概述

单元测试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，在一种传统的结构化编程语言中，如C语言中单元指一个函数，在像C++这样的面向对象的语言中，要进行测试的基本单元是类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，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。

5.1.1　单元测试的定义

单元测试(Unit Testing，又称模块测试)是在软件开发过程中进行的最低级别的测试活动，或者说是针对软件设计的最小单位程序模块进行的测试工作，其目的在于发现每个程序模块内部可能存在的差错。

单元测试由程序员自己来完成，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自己期望的一致。

事实上，人们每天都在做单元测试。你写了一个函数，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，当错误/缺陷(Bug)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，为每个函数建立测试函数，很简单的除外。

5.1.2 单元测试的目标

单元测试是在软件测试过程中最低级别的测试活动。保证单元模块被正确地编码是单元测试的主要目标，但还不够，单元测试还要实现以下目标。

（1）单元实现了其特定的功能，如果需要，返回正确的值。

（2）单元的运行能够覆盖预先设定的各种逻辑。

（3）在单元工作过程中，其内部数据能够保持完整性，包括全局变量的处理、内部数据的形式、内容及相互关系等不发生错误。

（4）可以接收正确数据，也能处理非法数据，在数据边界条件上，单元也能够正确工作。

（5）该单元的算法合理，性能良好。

（6）该单元代码经过扫描，没有发现任何安全性问题。

单元测试的活动模型如图 5-1 所示。

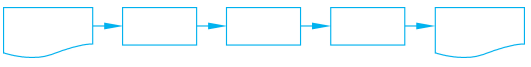

图 5-1 单元测试的活动模型

5.1.3 单元测试的任务

单元测试的主要任务包括逻辑、功能、数据和安全性等方面的测试，具体如下。

1. 检查模块接口是否正确

（1）输入的实际参数与形式参数是否一致。

（2）调用其他模块的实际参数与被调模块的形式参数是否一致。

（3）全程变量的定义在各模块是否一致。

（4）外部输入数据和输出数据。

2. 检查局部数据结构完整性

（1）不适合或不相容的类型说明。

（2）变量无初值。

（3）变量初始化或默认值有错。

（4）不正确的变量名或从来未被使用过。

（5）出现上溢或下溢和地址异常。

3. 检查临界数据处理的正确性

(1) 普通合法数据的处理。
(2) 普通非法数据的处理。
(3) 边界值内合法边界数据的处理。
(4) 边界值外非法边界数据的处理。

4. 检查每一条独立执行路径的测试，保证每条语句被至少执行一次

(1) 运算符优先级。
(2) 混合类型运算。
(3) 精度不够。
(4) 表达式符号。
(5) 循环条件，死循环。

5. 预见、预设的各种出错处理是否正确有效

(1) 输出的出错信息难以理解。
(2) 记录的错误与实际不相符。
(3) 程序定义的出错处理前系统已介入。
(4) 异常处理不当。
(5) 未提供足够的定位出错的信息。

5.2 对单元测试的误解

1. 单元测试浪费了太多的时间

当编码完成，开发人员总是会迫切希望进行软件的集成工作，这样就能够看到实际的系统开始启动工作了。这在外表上看来是一项明显的进步，而像单元测试这样的活动也许会被视为通往这个阶段点的道路上的障碍，推迟了对整个系统进行联调这种真正有意义的工作启动的时间。

在这种开发步骤中，真实意义上的进步被外表上的进步取代了。系统能够正常工作的可能性是很小的，更多的情况是充满了各式各样的 Bug。在实践中，这样一种开发步骤常常会导致这样的结果：软件甚至无法运行。更进一步的结果是大量的时间将被花费在跟踪那些包含在独立单元里的简单的缺陷上面，在个别情况下，这些 Bug 也许是琐碎和微不足道的，但是总的来说，它们会导致在软件集成为一个系统时增加额外的工期，而且当这个系统投入使用时也无法确保它能够可靠运行。

在实际工作中，进行了完整计划的单元测试和编写代码所花费的精力大致上是相同的。一旦完成了这些单元测试工作，很多缺陷将被纠正，在确信手头拥有稳定可靠的部件的情况下，开发人员能够进行更高效的系统集成工作。这才是真实意义上的进步，所以说

完整计划下的单元测试是对时间的更高效的利用，而调试人员的不受控和散漫的工作方式只会花费更多的时间而效果很差。

2. 单元测试仅仅是证明这些代码做了什么

对于那些没有首先为每个单元编写一个详细的规格说明，而直接跳到编码阶段的开发人员而言，当编码完成且面临代码测试任务的时候，就阅读这些代码并求证它实际上做了什么，把测试工作施加在已经写好的代码上。当然，他们无法证明任何事情，所有的这些测试工作能够表明的事情就是编译器工作正常。他们也许能够抓住罕见的编译器缺陷，但是能够做的仅仅是这些。

如果开发人员首先写好一个详细的规格说明，测试能够以规格说明为基础，代码就能够针对它的规格说明，而不是针对自身进行测试。这样的测试仍然能够抓住编译器的缺陷，同时也能找到更多的编码错误，甚至是一些规格说明中的错误。好的规格说明可以使测试的质量更高，所以最后的结论是高质量的测试需要高质量的规格说明。

在实践中会出现这样的情况，一个开发人员要面对测试一个单元时只给出单元的代码而没有规格说明这样吃力不讨好的任务。你怎样做才会有更多的收获，而不仅仅是发现编译器的 Bug？比较有效的方法是倒推出一个概要的规格说明。这个过程的主要输入条件是要阅读那些程序代码和注释，主要针对这个单元及调用它和被它调用的相关代码。画出流程图是非常有帮助的，可以用手工或使用某种工具，可以组织对这个概要规格说明的走读，以确保对这个单元的说明没有基本的错误，有了这种最小程度的代码深层说明，就可以用它来设计单元测试了。

3. 我是个很棒的程序员，我是不是可以不进行单元测试

在每个开发组织中都至少有一个这样的开发人员，他非常擅长编程，他们开发的软件总是在第一时间就可以正常运行，因此不需要进行测试。你是否经常听到这样的借口？

在真实世界里，每个人都会犯错误。即使某个开发人员可以抱着这种态度在很少的一些简单的程序中应付过去。但真正的软件系统是非常复杂的，真正的软件系统不可以寄希望于没有进行广泛的测试和 Bug 修改过程就可以正常工作。

编码不是一个可以一次性通过的过程。在真实世界中，软件产品必须进行维护以对操作需求的改变做出反应，并且要对最初的开发工作遗留下来的 Bug 进行修改。你希望依靠那些原始作者进行修改吗？这些制造出这些未经测试的原始代码的资深专家们还会继续在其他地方制造这样的代码。在开发人员做出修改后进行可重复的单元测试可以避免产生那些令人不快的副作用。

4. 集成测试将会抓住所有的缺陷

在前面的讨论中已经从一个侧面对这个问题进行了部分阐述。这个论点不成立的原因在于规模越大的代码集成意味着复杂性就越高。如果软件的单元没有事先进行测试，开发人员很可能会花费大量的时间仅仅是为了使软件能够运行，而任何实际的测试方案都无法执行。

当软件可以运行了，开发人员又要面对这样的问题：在考虑软件全局复杂性的前提下对每个单元进行全面的测试。这是一件非常困难的事情，甚至在创造一种单元调用的测试条件时，要全面地考虑单元在被调用时的各种入口参数。在软件集成阶段，对单元功能全面测试的复杂程度远远超过独立进行的单元测试过程。

最后的结果是测试将无法达到它所应该有的全面性。一些缺陷将被遗漏，并且很多缺陷将被忽略过去。假设要清洗一台已经完全装配好的食物加工机器，无论你喷了多少水和清洁剂，一些食物的小碎片还是会粘在机器的死角位置，只有任其腐烂并等待以后再想办法。但换个角度想，如果这台机器是拆开的，这些死角也许就不存在或者更容易被接触到了，并且对每一部分都可以毫不费力地进行清洗。

5. 成本效率不高

一个特定的开发组织或软件应用系统的测试水平取决于对那些未发现的 Bug 的潜在后果的重视程度。这种后果的严重程度可以从一个 Bug 引起的小小的不便到发生多次死机的情况。这种后果可能常常会被软件的开发人员所忽视(但是用户可不会这样)，这种情况会长期地损害这些向用户提交带有 Bug 的软件开发组织的信誉，并且会导致对未来的市场产生负面的影响。相反地，一个可靠的软件系统的良好的声誉将有助于一个开发组织获取未来的市场。

很多研究成果表明，无论什么时候，只要修改都要进行完整的回归测试，在生命周期中尽早地对软件产品进行测试将使效率和质量得到最好的保证。Bug 被发现得越晚，修改它所需的费用就越高，因此从经济角度来看，应该尽可能早地查找和修改 Bug。在修改费用变得过高之前，单元测试是一个在早期抓住 Bug 的机会。

相比后阶段的测试，单元测试的创建更简单，维护更容易，并且可以更方便地进行重复。从全程的费用来考虑，相比起那些复杂且旷日持久的集成测试，或是不稳定的软件系统来说，单元测试所需的费用是很低的。

各测试阶段测试所花费时间的示意图，见图 5-2(摘自《实用软件度量》(Capers Jones，McGraw-Hill，1991))，从图中可以看出，单元测试的时间成本效率大约是集成测试的 2 倍、系统测试的 3 倍。

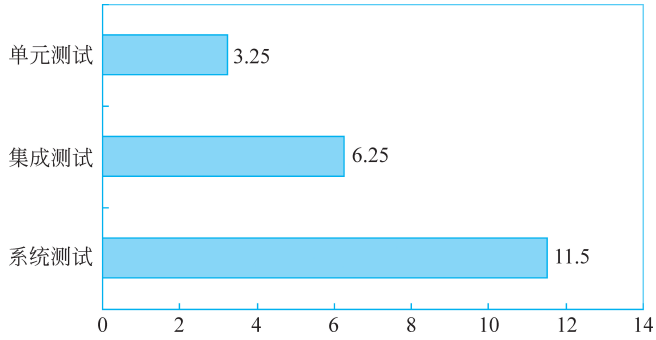

图 5-2　各测试阶段测试所花费时间的示意图

5.3 单元测试的必要性

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给自己的老板。但代码通过编译,只是说明了它的语法正确;却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。好在单元测试会为我们的承诺做担保。编写单元测试就是用来验证这段代码的行为是否与人们期望的一致。有了单元测试,可以自信地交付自己的代码,而没有任何后顾之忧。

1. 单元测试的时间

单元测试越早越好。一般是先编写产品函数的框架,然后编写测试函数,针对产品函数的功能编写测试用例,然后编写产品函数的代码,每写一个功能点都运行测试,随时补充测试用例。所谓先编写产品函数的框架,是指先编写函数空的实现,有返回值的随便返回一个值,编译通过后再编写测试代码,这时,函数名、参数表、返回类型都应该确定下来了,所编写的测试代码以后要修改的可能性比较小。

2. 由谁负责单元测试

单元测试与其他测试不同,单元测试可被视为编码工作的一部分,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。测试部门可以进行一定程度的审核。

3. 测试效果

根据以往的测试经验来看,单元测试的效果是非常明显的。首先,它是测试阶段的基础,做好了单元测试,再做后期的集成测试和系统测试时就很顺利。其次,在单元测试过程中能发现一些很深层次的问题,同时还会发现一些很容易发现而在集成测试和系统测试中却很难发现的问题。再次,单元测试关注的范围也特殊,它不仅是证明这些代码做了什么,最重要的是代码是如何做的,是否做了它该做的事情而没有做不该做的事情。

4. 测试成本

在单元测试时某些问题很容易被发现,如果在后期的测试中发现问题所花的成本将成倍上升。例如,在单元测试时发现1个问题需要1h,则在集成测试时发现该问题需要2h,在系统测试时发现则需要3h,同理,还有定位问题和解决问题的费用也是成倍数上升的,这就是要尽可能早地排除尽可能多的Bug以减少后期成本的因素之一。

5. 产品质量

单元测试的好与坏直接影响到产品的质量,可能就是由于代码中的某一个小错误就

导致了整个产品的质量降低一个指标，或者导致更严重的后果，如果做好了单元测试，这种情况是可以完全避免的。

综上所述，单元测试是构筑产品质量的基石，不要因为节约单元测试的时间不做单元测试或随便做而在后期浪费太多的时间，更不能由于节约那些时间导致开发出来的整个产品失败或重来。单元测试是十分必要的。

6. 单元测试的优点

1）单元测试是一种验证行为

程序中的每一项功能都是通过测试来验证它的正确性，它为以后的开发提供支援。就算是开发后期，也可以轻松地增加功能或更改程序结构，而不用担心这个过程中会破坏重要的东西。而且它为代码的重构提供了保障。这样就可以更自由地对程序进行改进。

2）单元测试是一种设计行为

编写单元测试要从调用者的角度进行观察、思考。特别是先写测试(test-first)，必须把程序设计成易于调用和可测试的，即必须解除软件中的耦合。

3）单元测试是一种编写文档的行为

单元测试是一种无价的文档，它是展示函数或类如何使用的最佳文档。这份文档是可编译、可运行的，并且它保持最新，永远与代码同步。

4）单元测试具有回归性

自动化的单元测试避免了代码出现回归，编写完成之后，可以随时随地快速运行测试。

5.4 单元测试环境和方法

5.4.1 驱动模块和桩模块的定义

由于一个模块并不是一个独立的程序，在考虑测试它时要同时考虑它和外界的联系，因此要用到一些辅助模块，来模拟与所测模块相联系的其他模块。一般把这些辅助模块分为两种。

(1) 驱动模块(driver)：对底层或子层模块进行(单元或集成)测试时所编制的调用被测模块的程序，用以模拟被测模块的上级模块。相当于所测模块的主程序。

(2) 桩模块(stub)：也有人称为存根程序，对顶层或上层模块进行测试时，所编制的替代下层模块的程序，用以模拟被测模块工作过程中所调用的模块。用于代替所测模块调用的子模块。

所测模块和与它相关的驱动模块及桩模块共同构成了一个“测试环境”，如图 5-3 所示。

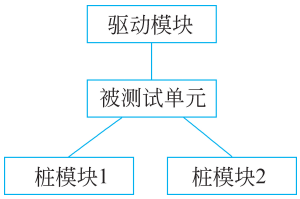

图 5-3 单元测试环境

5.4.2 驱动模块和桩模块的使用条件

1. 驱动模块的使用条件

(1) 必须驱动被测试模块执行。
(2) 必须能够正确接收要传递给被测试模块的各项参数。
(3) 能够对接收到的参数的正确性进行判断。
(4) 能够将接收到的数据传递给被测模块。
(5) 必须接收到被测试模块的执行结果,并对结果的正确性进行判断。
(6) 必须能够将判断结果作为用例执行结果输出测试报告。

2. 桩模块的使用条件

(1) 被测试模块必须调用桩模块。
(2) 必须能够正确接收来自被测试模块传递的各项参数。
(3) 桩模块要能够对接收到的参数的正确性进行判断。
(4) 桩模块对外的接口定义必须符合被测试模块调用的说明。
(5) 桩模块必须向被测试模块返回一个结果。

3. 单元测试的方法

单元测试主要采用白盒测试方法,辅以黑盒测试方法。白盒测试方法应用于代码评审、单元程序检验之中,而黑盒测试方法则应用于模块、组件等大单元的功能测试之中。

静态测试技术:不运行被测试程序,对代码通过检查、阅读进行分析。

三部曲:走查(Walk Through)、审查(Inspection)和评审(Review)。

动态测试需要真正将程序运行起来,需要设计系列的测试用例保证测试的完整性和有效性。动态测试又可以采用白盒测试和黑盒测试。

1) 白盒测试方法

- 语句覆盖;
- 判定覆盖;
- 条件覆盖;
- 判定/条件覆盖;
- 条件组合覆盖;
- 路径覆盖;

⋮

2) 黑盒测试方法

- 等价类划分法;
- 边界值分析法;
- 错误推测法;

- 因果图法；
- 功能图法；

⋮

在单元测试中，白盒及黑盒方法测试用例的使用孰先孰后呢？一般说来，由于黑盒测试是从被测单元外部进行测试，成本较低，可先对被测单元进行黑盒测试，之后再进行白盒测试。

5.5 单元测试策略

1. 自顶向下的单元测试

先对最顶层的基本单元进行测试，然后再对第二层的基本单元进行测试，以此类推，直到测试完所有基本单元。操作步骤如下。

（1）从最顶层开始，把顶层调用的单元作为桩模块。

（2）对第二层测试，使用上面已测试的单元作为驱动模块。

（3）以此类推，直到全部单元测试结束。

自顶向下的单元测试的优点：可以在集成测试之前为系统提供早期的集成途径。

自顶向下的单元测试的缺点：单元测试被桩模块控制，随着单元测试的不断进行，测试过程也会变得越来越复杂，测试难度以及开发和维护的成本都不断增加；要求的低层次的结构覆盖率也难以得到保证；由于需求变更或其他原因而必须更改任何一个单元时，就必须重新测试该单元下层调用的所有单元；低层单元测试依赖顶层测试，无法进行并行测试，使测试进度受到不同程度的影响，延长测试周期。

从上述分析中不难看出，该测试策略的成本要高于孤立的单元测试成本，因此从测试成本方面来考虑，并不是最佳的单元测试策略。

2. 自底向上的单元测试

先对最底层的基本单元进行测试，然后再对上面一层进行测试，以此类推，直到测试完所有单元。操作步骤如下。

（1）对模块调用图上的最底层模块开始测试，模拟调用该模块的模块为驱动模块。

（2）对上一层模块进行单元测试，用已经被测试过的模块作为桩模块。

（3）以此类推，直到全部单元测试结束。

自底向上的单元测试的优点：不需要单独设计桩模块。

自底向上的单元测试的缺点：随着单元测试的不断进行，测试过程会变得越来越复杂，测试周期延长，测试和维护的成本增加；随着各个基本单元的逐步加入，系统会变得异常庞大，因此测试人员不容易控制；越接近顶层的模块的测试其结构覆盖率就越难以保证。另外，顶层测试易受底层模块变更的影响，任何一个模块修改之后，直接或间接调用该模块的所有单元都要重新测试。还有，由于只有在底层单元测试完毕之后才能够进行顶层单元的测试，所以并行性不好。另外，自底向上的单元测试也不能和详细设计、编码

同步进行。

相对其他测试策略而言,该测试策略比较合理,尤其是需要考虑对象或复用时。它属于面向功能的测试,而非面向结构的测试。对那些以高覆盖率为目标或者软件开发时间紧张的软件项目来说,这种测试方法不适用。

3. 孤立单元测试

不考虑每个单元与其他单元之间的关系,为每个单元设计桩模块或驱动模块。每个模块进行独立的单元测试。

操作步骤:无须考虑每个模块与其他模块之间的关系,分别为每个模块单独设计桩模块和驱动模块,逐一完成所有单元模块的测试。

孤立单元测试的优点:该方法简单、容易操作,因此所需测试时间短,能够达到高覆盖率。

孤立单元测试的缺点:不能为集成测试提供早期的集成途径。依赖结构设计信息,需要设计多个桩模块和驱动模块,增加了额外的测试成本。

该方法是比较理想的单元测试方法,如辅助适当的集成测试策略,有利于缩短项目的开发时间。

5.6 单元测试用例设计

从单元测试方法中已经知道,单元测试用例的设计既可以使用白盒测试也可以使用黑盒测试,但以白盒测试为主。

白盒测试进入的前提条件是测试人员已经对被测试对象有了一定的了解,基本上明确了被测试软件的逻辑结构。白盒测试应该达到的目标是:100%的语句覆盖,100%的分支覆盖,并且根据具体软件系统的要求增加其他覆盖测试,如财务软件、银行系统、航空航天系统等。

黑盒测试是要首先了解软件产品具备的功能和性能等需求,再根据需求设计一批测试用例以验证程序内部活动是否符合设计要求的活动。

测试人员在实际工作中设计单元测试用例应该满足以下几点。

(1) 测试程序单元的功能是否实现。

(2) 测试程序单元性能是否满足要求(可选)。

(3) 是否有可选的其他测试特性,如边界、余量、安全性、可靠性、强度测试、人机交互界面测试等。

无论是白盒测试还是黑盒测试,每个测试用例都应该包含以下四个要素。

(1) 被测单元模块初始状态声明,即测试用例的开始状态。

(2) 被测单元的输入,包含由被测单元读入的任何外部数据值。

(3) 该测试用例实际测试的代码,用被测单元的功能和测试用例设计中使用的分析来说明。

(4) 测试用例的期望输出结果。