

University of South China

**课程作业**

|  |  |
| --- | --- |
| **题 目** | **单元测试概述** |
| **学 院** | **计算机科学与技术学院** |
| **课 程** | **软件测试自动化方法与工具** |
| **学 号** | **201620810172** |
| **学生姓名** | **孙 溢** |

2017年6月5日

目录

# 0 引言

在[计算机编程](https://zh.wikipedia.org/wiki/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BC%96%E7%A8%8B" \o "计算机编程)中，单元测试（英语：Unit Testing）又称为模块测试, 是针对[程序模块](https://zh.wikipedia.org/wiki/%E6%A8%A1%E7%B5%84_(%E7%A8%8B%E5%BC%8F%E8%A8%AD%E8%A8%88)" \o "模块 (程序设计))（[软件设计](https://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E8%AE%BE%E8%AE%A1" \o "软件设计)的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在[过程化编程](https://zh.wikipedia.org/wiki/%E9%81%8E%E7%A8%8B%E5%8C%96%E7%B7%A8%E7%A8%8B" \o "过程化编程)中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

单元测试的目的是检验每个软件单元能否正确地实现其功能，满足其性能和接口要求，并验证程序和详细设计说明的一致性。它将在与程序的其他部分相隔离的情况下进行测试，其目的在于发现侮个程序模块内部在逻辑上和功能上可能存在的错误和缺陷。

通常来说，程序员每修改一次程序就会进行最少一次单元测试，在编写程序的过程中前后很可能要进行多次单元测试，以证实程序达到[软件规格书](https://zh.wikipedia.org/wiki/%E8%A6%8F%E6%A0%BC_(%E6%8A%80%E8%A1%93%E6%A8%99%E6%BA%96)" \o "规格 (技术标准))要求的工作目标，没有[程序错误](https://zh.wikipedia.org/wiki/Bug" \o "Bug)。单元是整个软件的构成基础，因此单元的质量是整个软件质量的基础。

# 1 收益

单元测试的目标是隔离程序部件并证明这些单个部件是正确的。一个单元测试提供了代码片断需要满足的严密的书面规约。因此，单元测试带来了一些益处。 单元测试在[软件开发过程](https://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91%E8%BF%87%E7%A8%8B" \o "软件开发过程)的早期就能发现问题。

## 1.1 适应变更

单元测试允许程序员在未来[重构](https://zh.wikipedia.org/wiki/%E9%87%8D%E6%9E%84" \o "重构)代码，并且确保模块依然工作正确（[复合测试](https://zh.wikipedia.org/w/index.php?title=%E5%A4%8D%E5%90%88%E6%B5%8B%E8%AF%95&action=edit&redlink=1" \o "复合测试（页面不存在）)）。这个过程就是为所有[函数](https://zh.wikipedia.org/wiki/%E5%87%BD%E6%95%B0" \o "函数)和[方法](https://zh.wikipedia.org/wiki/%E6%96%B9%E6%B3%95" \o "方法)编写单元测试，一旦变更导致错误发生，借助于单元测试可以快速定位并修复错误。

可读性强的单元测试可以使程序员方便地检查代码片断是否依然正常工作。良好设计的单元测试案例覆盖程序单元分支和循环条件的所有路径。

在连续的单元测试环境，通过其固有的持续维护工作，单元测试可以延续用于准确反映当任何变更发生时可执行程序和代码的表现。借助于上述开发实践和单元测试的覆盖，可以分分秒秒维持准确性。

## 1.2 简化集成

单元测试消除程序单元的不可靠，采用[自底向上](https://zh.wikipedia.org/w/index.php?title=%E8%87%AA%E5%BA%95%E5%90%91%E4%B8%8A&action=edit&redlink=1" \o "自底向上（页面不存在）)的测试路径。通过先测试程序部件再测试部件组装，使[集成测试](https://zh.wikipedia.org/wiki/%E9%9B%86%E6%88%90%E6%B5%8B%E8%AF%95" \o "集成测试)变得更加简单。

业界对于人工集成测试的必要性存在较大争议。尽管精心设计的单元测试体系看上去实现了集成测试，因为集成测试需要人为评估一些人为因素才能证实的方面，单元测试替代集成测试不可信。一些人认为在足够的自动化测试系统的条件下，人力集成测试组不再是必需的。事实上，真实的需求最终取决于开发产品的特点和使用目标。另外，人工或手动测试很大程度上依赖于组织的可用资源。

## 1.3 文档记录

单元测试提供了系统的一种文档记录。借助于查看单元测试提供的功能和单元测试中如何使用程序单元，开发人员可以直观的理解程序单元的基础API。

单元测试具体表现了程序单元成功的关键特点。这些特点可以指出正确使用和非正确使用程序单元，也能指出需要捕获的程序单元的负面表现（译注：异常和错误）。尽管很多软件开发环境不仅依赖于代码做为产品文档，在单元测试中和单元测试本身确实文档化了程序单元的上述关键特点。

另一方面，传统文档易受程序本身实现的影响，并且时效性难以保证（如设计变更、功能扩展等在不太严格时经常不能保持文档同步更新）。

## 1.4 表达设计

在测试驱动开发的软件实践中，单元测试可以取代正式的设计。每一个单元测试案例均可以视为一项类、方法和待观察行为等设计元素。下面的[Java](https://zh.wikipedia.org/wiki/Java" \o "Java)例可以帮助说明这一点。

这是一个证明一批实现设计元素的测试类。首先，要求有一个名为Adder的接口，和一个不带参数的构造方法名为AdderImpl的实现类。然后，它断言Adder接口包含有一个两个整数参数返回值为整型的add方法。它也通过小范围的值检验说明方法的行为。

public class TestAdder {

public void testSum() {

Adder adder = new AdderImpl();

assert(adder.add(1, 1) == 2);

assert(adder.add(1, 2) == 3);

assert(adder.add(2, 2) == 4);

assert(adder.add(0, 0) == 0);

assert(adder.add(-1, -2) == -3);

assert(adder.add(-1, 1) == 0);

assert(adder.add(1234, 988) == 2222);

}}

这个案例中，单元测试在程序之前写成，用作指明待设计的对象形态和行为的文档，没有任何实现细节，留作程序员练习。以下可能是最简单的工作实践，这个最容易的解决方案可以通过上述测试：

interface Adder {

int add(int a, int b);}class AdderImpl implements Adder {

int add(int a, int b) {

return a + b;

}}

不同于其他基于图的设计方法，用单元测试表达设计有一项显著优点：设计文档（单元测试本身）可以用于验证程序实现匹配设计。UML可能会遇到这样的问题：尽管图上一个类被命名为Customer，但开发人员可以称其为Wibble，而且系统中没有任何地方会显示出这个差异。基于单元测试设计方法，开发人员不遵循设计要求的解决方案永远不会通过测试。

当然，单元测试缺乏图的可读性，但UML图可以在自由工具（通常可从IDE扩展获取）中为大多数现代程序语言生成UML图，很难要求采购昂贵的UML设计套装软件。自由工具，类似于基于xUnit框架的工具，测试结果输出到一些可生成供人工识读的图形化工具系统中去。

# 2 局限

测试不可能发现所有的程序错误，单元测试也不例外。按定义，单元测试只测试程序单元自身的功能。因此，它不能发现集成错误、性能问题、或者其他系统级别的问题。单元测试结合其他软件测试活动更为有效。与其它形式的[软件测试](https://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E6%B5%8B%E8%AF%95" \o "软件测试)类似，单元测试只能表明测到的问题，不能表明不存在未测试到的错误。

软件测试是一个组合问题。例如，每一个布尔型的决断语句需要至少两种测试：一个返回真，一个返回假。因此，针对每行书写的代码，程序员通常需要写3至5行的测试代码。这很明显地很花时间而且对此的投入可能并不值得。也有些问题是根本不能简单地检测出来的——例如具不确定性的或牵扯到多[线程](https://zh.wikipedia.org/wiki/%E7%BA%BF%E7%A8%8B" \o "线程)的问题。此外，替单元测试写的代码可能就像要测试的代码一样有程序错误。[佛瑞德·布鲁克斯](https://zh.wikipedia.org/wiki/%E4%BD%9B%E7%91%9E%E5%BE%B7%C2%B7%E5%B8%83%E9%AD%AF%E5%85%8B%E6%96%AF" \o "佛瑞德·布鲁克斯)在[人月神话](https://zh.wikipedia.org/wiki/%E4%BA%BA%E6%9C%88%E7%A5%9E%E8%A9%B1" \o "人月神话)一书中举例说明：“绝对不要带两个计时器去海边。最好总是带一或三个”。意味着，如果两个[计时器](https://zh.wikipedia.org/w/index.php?title=%E7%B2%BE%E5%AF%86%E8%88%AA%E6%B5%B7%E8%A8%98%E6%99%82%E5%99%A8&action=edit&redlink=1" \o "精密航海记时器（页面不存在）)互相冲突的话，你该怎么知道哪个是对的？为了获得单元测试的好处，在软件开发过程中应形成一套严格纪律意识。仔细保留记录是必要的，不仅仅只保留执行的测试，也包括保留对应的源码和其它软件单元的变更历史。即，使用[版本控制系统](https://zh.wikipedia.org/wiki/%E7%89%88%E6%9C%AC%E6%8E%A7%E5%88%B6%E7%B3%BB%E7%BB%9F" \o "版本控制系统)是必要的。如果后续版本不能通过一个以前测试通过的单元测试，版本控制系统可以提供对应时间段对源代码所做的变更清单。

每天养成查看单元测试案例失败测试并及时确定错误原因的习惯是必要的。如果没有这样的流程，没有在团队工作流程中体现，单元测试系列将走向不同步，造成越来越多的错误和越来越低效的单元测试案例系列。

# 3 测试内容

## 3.1 接口测试

模块接口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，其它测试才有意义。测试接口正确与否应该考虑下列因素：

1. 输入的实际参数与形式参数的个数是否相同；
2. 输入的实际参数与形式参数的属性是否匹配；
3. 输入的实际参数与形式参数的量纲是否一致；
4. 调用其它模块时所给实际参数的个数是否与被调模块的形式参数个数相同；
5. 调用其他模块时所给实际参数的属性是否与被调模块的形式参数属性匹配；
6. 调用其他模块时所给实际参数的量纲是否与被调模块的形式参数量纲一致；
7. 调用预定义函数时所用参数的个数、属性和次序是否正确；
8. 是否存在与当前入口点无关的参数引用；
9. 是否修改了只读型参数；
10. 各模块对全程变量的定义是否一致；
11. 是否把某些约束作为参数传递。

如果模块内包括外部输入输出，还应该考虑下列因素：

1. 文件属性是否正确；
2. OPEN/CLOSE语句是否正确；
3. 格式说明与输入输出语句是否匹配；
4. 缓冲区大小与记录长度是否匹配；
5. 文件使用前是否已经打开；
6. 是否处理了文件尾；
7. 是否处理了输入/输出错误；
8. 输出信息中是否有文字性错误。

## 3.2 局部数据结构测试

测试模块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的完整和正确。局部数据结构往往是错误的根源，应该仔细设计测试用例，力求发现以下几类错误：

a)不合适或不相容的类型说明；

b)变量无初值；

c)变量初始化或缺省值有错误；

d)不正确的变量名（拼写错误或不正确地截断）；

e)出现上溢、下溢和地址异常。

除了局部数据结构外，如果可能，单元测试时还应该查清全局数据(例如FORTRAN的公用区)对模块的影响。

## 3.3 边界条件测试

边界条件测试是单元测试中最后、也是最重要的一项任务。众所周知，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，很有可能发现新的错误。

## 3.4 执行路径测试

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。设计该型测试用例是为了发现因错误的计算、不正确的比较和不适当的控制流所造成的错误，基本路径测试和循环测试是最常用且最有效的模块执行路径测试技术。计算中常见的错误包括：

1. 误解或错用了运算符优先级；
2. 混合类型运算；
3. 变量初始值错误；
4. 精度不够；
5. 表达式符号错误。

比较判断与控制流常常紧密相关，设计测试用例还应注意发现下列错误：

a)不同数据类型的对象之间进行比较；

b)错误地使用逻辑运算符或优先级；

c)因计算机表示的局限性，理论上期望相等而实际上并不相等的两个量相等；

d)比较运算或变量出错；

e)循环终止条件不可能出现；

f)迭代发散时不能退出；

g)错误地修改了循环变量。

## 3.5 出错处理通道测试

一个好的测试用例应能预见各种出错条件，并预设各种出错处理通路，出错处理通路同样需要认真测试，测试应着重检查下列问题：

a)输出的出错信息难以理解；

b)记录的错误与实际遇到的错误不相符；

c)在程序自定义的出错处理段运行之前，系统已介入；

d)异常处理不当；

e)错误陈述中未能提供足够的出错定位信息。

# 4 测试环境

由于被测模块并不是一个独立可运行的程序，因此需要构造该模块的测试环境，要考虑它和外界的联系，用一些辅助模块去模拟与所测模块相联系的其他模块。这些辅助模块分为两种：

①驱动模块:相当于被测模块的主程)乞它接收测试数据，把这些数据传送给所测模块，最后再输出实测结果。

②桩模块:用以代替所测模块调用的子模块，它可以做少量的数据操作，不需要把子模块所有功能都带进来，但不允许千}一么事情也不做。被测模块、与它相关的驱动模块和桩模块共同构成了一个“测试环境”，如图1所示：

图1 单元测试工作环境

测试结果

测试用例

桩模块

桩模块

桩模块

被测模块

驱动模块

# 5 测试方法

在单元测试时，采用的测试技术可分为静态测试和动态测试，而动态测试的方法又可分为白盒测试和黑盒测试。其中以结构化的白盒测试方法为主，辅以黑盒测试。动态测试方法为单元测试的重点与难点。

## 5.1 静态测试

静态测试是单元测试的第一步，要在进行动态测试前先完成该项测试。这一阶段的主要工作是保证代码算法的逻辑正确性、清晰性、规范性、一致性、算法高效性，并尽可能地发现程序中隐含的错误。静态测试通过静态分析和代码审查两种形式进行。

### 5.1.1 静态分析

静态分析一般通过工具来完成对软件进行度量。通过静态分析可以达到以下目的:对软件的质量做出评价:为动态测试建立基本路径和生成环境:发现程序中的多余物，以及程序对推荐的编程标准的违反。例如，通常要对软件单元的控制流、数据流、接口、表达式等内容进行分析。通过静态分析，可使测试人员在很短的时间内就可以对被测软件有一个大概的了解，从而根据测试结果确定哪些部分需要重点分析测试和审查。

### 5.1.2 代码审查

代码审查是在充分理解系统功能需求和设计思想的基础上，利用人工代码走查的手段对软件各模块进行分析，检查被测软件的源代码是否实现了测试需求中规定的内容，检查代码设计的规范性、逻辑表达的正确性、多余物的处理、各模块接口状态的一致性、安全性关键信息等等。例如，对程序格式、入口和出口的连接、程序语言的使用，寄存器的使用等进行的检查。

## 5.2 动态测试

实践经验表明，静态测试能够有效地发现30%-70%的逻辑设计和编码错误，但程序代码中仍存在大量的隐性错误和缺陷无法通过静态测试发现，必须通过动态测试才能捕捉到。动态测试是与静态测试相对而言的，动态测试是将被测软件运行起来，根据需求加入相关测试用例，并取得测试结果，进行分析，检查软件满足需求的情况。

### 5.2.1 动态测试的方法

动态测试的基本方法为黑盒测试和白盒测试。

黑盒测试是把程序看成是一个内部不可见的黑盒，测试者无需顾及程序内部结构、语句、分支等情况，只需根据需求，设计相应的测试用例，根据输出结果判断程序功能以及性能的正确性。

白盒测试是基于结构的测试，是一种对程序内部的逻辑结构和编码结构进行测试的方法。它依赖于对程序细节的严密检验对软件的逻辑路径进行测试。

黑盒测试和白盒测试各有优缺点，构成互补关系，在软件测试中需要把两者结合起来应用，以达到测试覆盖率的要求。

### 5.2.2 动态测试的实现

在单元测试的动态测试过程中，通常要完成如下工作：设计测试用例、测试的执行、测试结果分析。

（1）测试用例设计

对源程序做完静态测试之后，就可以开始进行测试用例设计。利用设计文档，设计可以验证程序功能、找出程序错误的多个测试用例。设计测试用例时采取以下方法：

白盒测试一般采用逻辑覆盖法和基本路径法进行测试。逻辑覆盖是以程序内部的逻辑结构为基础的测试用例设计技术，按照复杂度从小到大可分为：语句覆盖、判定覆盖、条件覆盖、判定一条件覆盖、条件组合覆盖与路径覆盖通常在设计测试用例时根据代码模块的复杂度，选择覆盖方法。基本路径测试法是在程序流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。

使用黑盒测试作为白盒测试的补充。黑盒测试主要采用等价类划分和边界值分析方法。经验表明，用边界值分析方法设计出测试用例发现程序错误能力最强，因为软件经常在边界上失效。用等价类划分方法补充一些测试用例，该方法可以把全部的输入数据划分成若干的等价类，在侮一个等价类中取一个数据来进行测试。这样就能以较少的具有代表性的数据进行测试，而取得较好的测试效果。

（2）测试的执行

测试开始前，需要对将要查看或检测的对象进行设置。测试工具可通过设置数据追踪和检测对象的预期值来完成这项工作。数据追踪即是由测试人员设置要追踪的对象(包括程序或数据内存、寄存器、以及变量等)，在测试结束时，可给出其测试过程中的变化过程包括最大值、最小值、均值以及侮次数据变化的过程)。检测对象即是一次动态测试结束时，测试人员可通过查看某个对象（寄存器、变量或内存单元）的值来判断本次动态测试的程序运行结果是否正确，是否满足预期值的要求。待设置完成后，即可对所有的测试用例执行一次，并对出现错误或异常的测试用例跟踪执行一次，以发现问题根源。

（3）测试结果分析

测试后，测试工具会列出侮次动态测试过程中，被测程序的执行过程(包括分支的走向、语句执行的次数、语句占用的时间等)，以及当前所有CPU寄存器的状态值，检测对象输出值与预期值比较结果，测试人员可以通过这些信息，完全了解程序的执行过程，发现和分析问题。另外濡要分析该模块测试的充分性。通常在单元测试时，对于白盒测试要求语句覆盖和分支覆盖率达到100%，对于黑盒测试要求功能覆盖率达到100%。

# 6 测试工具

# 7 单元测试的实施

从单元测试的效率角度来考虑，由开发人员进行单元测试有其优点。因为从单元测试的过程看，单元测试普遍采用白盒测试的方法，离不开深入被测对象的代码，同时还需要构造驱动模块、桩模块，因此开展单元测试需要有较好的开发知识。从人员的知识结构、对代码的熟悉程度考虑，开发人员具有一定的优势；单元测试由开发人员进行能带来一些特别的收益。当然，根据“程序员应避免检查自己的程序”的测试原则，开发人员进行单元测试一般推荐采用交叉测试的方法。

另一方面，从单元测试效果的角度考虑，必须从组织结构上保证测试组参与单元测试。首先，从目前我国实际现状来看，测试人员质量意识要高于开发人员，测试人员参与单元测试能够提高测试质量；其次，对被测系统越了解，测试才有可能越深入，测试人员参与单元测试，将使得测试人员能够从代码级熟悉被测系统，这对测试组后期集成测试和系统测试活动非常有帮助，会很大的提升集成测试和系统测试质量。

实际上，单元测试由测试部门来做还是由开发部门来做，是一个有争论的话题。一种比较被认同的观点是：在允许条件下，由测试部门和开发部门共同来做，测试部门负责制定规范、培训并检查测试效果，开发部门负责具体的实施，最好是边开发边测试。显然，这种观点主要希望在单元测试的效果与效率上能有较好的综合和平衡。

# 8 应用

## 8.1 极限编程

单元测试是[极限编程](https://zh.wikipedia.org/wiki/%E6%9E%81%E9%99%90%E7%BC%96%E7%A8%8B" \o "极限编程)的基础，依赖于自动化的单元测试框架。自动化的单元测试框架可以来源于第三方，如[xUnit](https://zh.wikipedia.org/w/index.php?title=XUnit&action=edit&redlink=1" \o "XUnit（页面不存在）)，也可以由开发组自己创建。

极限编程创建单元测试用于[测试驱动开发](https://zh.wikipedia.org/wiki/%E6%B5%8B%E8%AF%95%E9%A9%B1%E5%8A%A8%E5%BC%80%E5%8F%91" \o "测试驱动开发)。首先，开发人员编写单元测试用于展示软件需求或者软件缺陷。因为需求尚未实现或者现有代码中存在软件缺陷，这些测试会失败。然后，开发人员遵循测试要求编写最简单的代码去满足它，直到测试得以通过。

系统中大多数代码都经过单元测试，但并非所有代码路径都必需单元测试。极限编程强调“测试所有可能中断”的策略，而传统方法是“测试所有执行路径”。这使得极限编程开发人员比传统开发少写单元测试，但这并不是问题。不争的事实是传统方法很少完全遵循完整地测试所有执行路径的要求。极限编程相互地认识到测试很少能完备（因为完备测试通常需要昂贵的代价和时间消耗，意味着不经济），提供了如何有效地将有限资源集中投入可花费的代价到问题关键的导引。

至关重要的，测试代码应视为第一个项目成品，与实现代码维持同等级别的质量要求，没有重复。开发人员在提交程序单元代码时一并提交单元测试代码到代码库。彻底的极限编程单元测试代码提供上述单元测试的收益，如简化和更可信的程序开发和[重构](https://zh.wikipedia.org/wiki/%E9%87%8D%E6%9E%84" \o "重构)、简化代码集成、精确的文档和模块化的设计。而且，单元测试经常作为[复合测试](https://zh.wikipedia.org/w/index.php?title=%E5%A4%8D%E5%90%88%E6%B5%8B%E8%AF%95&action=edit&redlink=1" \o "复合测试（页面不存在）)的一种形式被运行。

## 8.2 技术

单元测试通常情况下[自动](https://zh.wikipedia.org/w/index.php?title=%E8%87%AA%E5%8A%A8&action=edit&redlink=1" \o "自动（页面不存在）)进行，但也可被手动执行。[IEEE](https://zh.wikipedia.org/wiki/IEEE" \o "IEEE)没有偏爱某一种形式。手动的单元测试可用于step-by-step的教学文档。尽管如此，单元测试的目标是隔离程序单元并验证其正确性。自动执行使目标达成更有效，也可获得本文上述单元测试收益。相反，不细心规划或者精心的单元测试可能被视为包括多个软件组件的集成测试案例，于是将因未完全达到创建单元测试的预定目标，测试可能失去较多收益。

在自动化测试时，为了实现隔离的效果，测试将脱离待测程序单元（或代码主体）本身固有的运行环境之外，即脱离产品环境或其本身被创建和调用的上下文环境，而在测试框架中运行。以隔离方式运行有利于充分显露待测试代码与其它程序单元或者产品数据空间的依赖关系。这些依赖关系在单元测试中可以被消除。

借助于自动化测试框架，开发人员可以抓住关键进行编码并通过测试去验证程序单元的正确性。在测试案例执行期间，框架通过日志记录了所有失败的测试准则。很多测试框架可以自动标记和提交失败的测试案例总结报告。根据失败的程度不同，框架可以中止后续测试。

总体说来，单元测试会激发程序员创造解耦的和内聚的代码体。单元测试实践有利于促进健康的软件开发习惯。设计模式、单元测试和重构经常一起出现在工作中，借助于它们，开发人员可以生产出最为完美的解决方案。

## 8.3 单元测试框架

单元测试框架通常是没有作为编译器包的第三方产品。他们帮助简化单元测试的过程，并且已经为各种编程语言开发。

通常在没有特定框架支持下，通过撰写在测试中的运行单元，并使用[判定](https://zh.wikipedia.org/wiki/%E6%96%B7%E8%A8%80_(%E7%A8%8B%E5%BC%8F)" \o "断言 (程序))、[异常处理](https://zh.wikipedia.org/wiki/%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86" \o "异常处理)、或其他[控制流程](https://zh.wikipedia.org/wiki/%E6%8E%A7%E5%88%B6%E6%B5%81%E7%A8%8B" \o "控制流程)机制来表示失败的用户代码（client code）运行单元测试是可行的。不通过框架的单元测试有用之处在于进行单元测试时会有一个[参进障碍](https://zh.wikipedia.org/w/index.php?title=%E5%8F%83%E9%80%B2%E9%9A%9C%E7%A4%99&action=edit&redlink=1" \o "参进障碍（页面不存在）)（barrier to entry）；进行一点单元测试几乎不比没做好多少，但是一旦使用了框架，加入单元测试相对来说会简单许多。在某些框架中许多先进单元测试特征丢失了或者必须是手工编写的。

## 8.4 语言级单元测试支持

某些编程语言直接支持单元测试。他们的语法允许直接进行单元测试的宣告而不需要导入（不管是第三方的或标准的）。除此之外，单元测试的布尔条件可以用与非单元测试码的布尔表示法相同的语法来表示，例如if和while宣告的用法。

直接支持单元测试的语言包含了：[C#](https://zh.wikipedia.org/wiki/C" \o "C)、[D语言](https://zh.wikipedia.org/wiki/D%E8%AF%AD%E8%A8%80" \o "D语言)。

# 9 总结

软件测试技术的发展日新月异，单元测试是软件测试的基础。通过单元测试为后续软件测试工作提供了重要保障，完整的测试不仅可以对软件质量进行一个正确地评价，而且是提高软件质量和可靠性的重要方法。

单元测试是软件测试的基础。进行单元测试，首先必须明确，要对单元模块哪些方面进行测试；从测试效率或测试效果等综合考虑，采取测试部门和开发部门共同进行单元测试是一种很好的组织方式，当然，这要符合实际情况；掌握好基本的测试方法，引入测试工具作为辅助以求正确而高效地进行单元测试。当然，影响单元测试质量的远不止这些因素，不断地学习和掌握新的测试技术，严格而科学地组织测试流程和配置管理等，才能提高单元测试质量。