

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

摘要

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。

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)；虽然单元测试不是什么必须的，但也不坏，这牵涉到[项目管理](https://zh.wikipedia.org/wiki/%E5%B0%88%E6%A1%88%E7%AE%A1%E7%90%86" \o "项目管理)的政策决定。

每个理想的[测试案例](https://zh.wikipedia.org/wiki/%E6%B5%8B%E8%AF%95%E6%A1%88%E4%BE%8B" \o "测试案例)独立于其它案例；为测试时隔离模块，经常使用stubs、mock或fake等测试[马甲程序](https://zh.wikipedia.org/w/index.php?title=%E9%A9%AC%E7%94%B2%E7%A8%8B%E5%BA%8F&action=edit&redlink=1" \o "马甲程序（页面不存在）)。单元测试通常由[软件开发人员](https://zh.wikipedia.org/w/index.php?title=%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91%E4%BA%BA%E5%91%98&action=edit&redlink=1" \o "软件开发人员（页面不存在）)编写，用于确保他们所写的代码匹配软件需求和遵循[开发目标](https://zh.wikipedia.org/w/index.php?title=%E5%BC%80%E5%8F%91%E7%9B%AE%E6%A0%87&action=edit&redlink=1" \o "开发目标（页面不存在）)。它的实施方式可以是非常手动的（通过纸笔），或者是做成[构建自动化](https://zh.wikipedia.org/wiki/%E6%A7%8B%E5%BB%BA%E8%87%AA%E5%8B%95%E5%8C%96" \o "构建自动化)的一部分。

目录

[1 收益 1](#_Toc4499)

[1.1 适应变更 1](#_Toc24970)

[1.2 简化集成 1](#_Toc8862)

[1.3 文档记录 1](#_Toc2726)

[1.4 表达设计 2](#_Toc22069)

[2 分离接口和实现 3](#_Toc6209)

[3 局限 3](#_Toc5350)

[4 应用 3](#_Toc970)

[4.1 极限编程 3](#_Toc16659)

[4.2 技术 4](#_Toc518)

[4.3 单元测试框架 5](#_Toc20658)

[4.4 语言层单元测试支持 5](#_Toc31823)

# 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 分离接口和实现

因为很多类会引用其它[类](https://zh.wikipedia.org/wiki/%E7%B1%BB" \o "类)，对这个类的测试经常会要求测试其它的类。一个最普遍的例子是依赖于[数据库](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%BA%93" \o "数据库)的类：为了测试它，测试人员通常编写代码去操作数据库。这是不对的，因为单元测试不应超出待测试的类边界。

作为替代，软件开发人员应创建一个数据库连接的抽象接口，然后实现这个接口的[模拟对象](https://zh.wikipedia.org/wiki/%E6%A8%A1%E6%8B%9F%E5%AF%B9%E8%B1%A1" \o "模拟对象)。通过对代码所需附件的抽象（临时降低了网状的耦合效应），这些独立程序单元较前者更能被完整测试。高质量的代码单元也可提供更好的可维护性。

# 3 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。

# 4 应用

## 4.1 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。

## 4.2 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。

## 4.3 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。

## 4.4 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)