第 1 章
序言
Introduction

在一个软件项目中，我们可以做许多各式各样的测试，而且这些测试也是必须的。其中的某些测试需要用户的大量参与；而某些则需要有专门的质量保证小组来进行；或者需要其他的一些昂贵资源。

然而，这些测试并不是我们所要谈论的主题。

我们谈的是单元测试 :它是项目成功、个人成功的一个不可或缺的部分，但对它，人们却又存在各种各样的误解。单元测试其实是相对廉价而简单的技术，但它能让你更高效地写出质量更好的代码。

说到测试，大凡组织和个人都会满怀雄心壮志，但是往往只是在项目快要结束的时候才想起测试。而那时的进度压力一定非常紧迫，所以结果往往只是浅尝辄止或者干脆就不测了。

许多程序员觉得测试只是一件麻烦事：一种自找的烦恼，唯一的“效果”就是让他们没法专注于手上的正经事——cutting code（堆砌代码）。

每个人都同意，是的，该做更多的测试。这种人人同意的事情还多着呢，是的，该多吃蔬菜，该戒烟，该多休息，该多锻炼……这并不意味着我们中的所有人都会这么去做，不是吗？

但是单元测试却远远不仅仅是上面这些——也许你会认为单元测试是花菜那一类的，而我要说它更像一勺美味的调料，它让每份菜肴品尝起来都

更加可口。单元测试的设计目的并不是为了获得一些更好的整体质量。也就是说，它并不是一个针对最终用户、项目经理和开发组长的工具；而是由程序员自己来完成，并且最终受益的也是程序员自己。我们是为了自身的利益去使用单元测试，从而让我们的工作变得更加轻松。

简而言之，在有些时候，使用单元测试本身就能决定你的项目的成败。接下来，让我们来看看下面这个小故事。

1.1 自信地编码

有一次——或许就是上个礼拜二——有两个开发者：Pat 和 Dale。他们面临着相同的最后期限，而这一天也越来越近了。Pat 每天都在着急地编写代码，写完一个类又写一个类，写完一个函数又接着写另一个函数，还经常不得不停下来做一些调整，使得代码能够通过编译。

Pat 一直保持着这种工作方式，直到最后期限的前一天。而这时已经是演示所有代码的时候了。Pat 运行了最上层的程序，但是一点输出也没有，什么都没有。这时只好用调试器来单步跟踪了。“Hmm，决不可能是这样的”，Pat 想，“此时这个变量绝对不是 0 啊”。于是，Pat 只能回过头来看代码，尝试着跟踪一下这个难以琢磨的程序的调用流程。

时间已经越来越晚了，Pat 找到并且纠正了这个 bug；但在这个过程中，Pat 又找到了其他好几个 bug；如此几次过后，bug 还是存在。而程序输出那边，仍然没有结果。这时，Pat 已经筋疲力尽了，完全搞不清楚为什么会这样，认为这种（没有输出的）行为是毫无道理的。

而于此同时，Dale 并没像 Pat 那么快地写代码。Dale 在写一个函数的时候，会附带写一个简短的测试程序来测试这个函数。这里没有什么特殊的地方，只是添加了一个简单的测试，来判断函数的功能是否和程序员期望的一致。显然，考虑如何写，然后把测试写出来，是需要占用一定时间的；但是 Dale 在未对刚写的函数做出确认之前，是不会接着写新代码的。也就是说，只有等到已知函数都得到确认之后，Dale 才会继续编写下一个函数，然后调用前面的函数等等。

在整个过程中，Dale 几乎不使用调试器；而且对 Pat 的模样也有些困惑不解：只见他头埋在两手之间，嘀咕着各种难听的话语，咒骂着计算机，充血的眼球同时盯着好几个调试窗口。

最后期限终于到了，Pat 未能完成任务。而 Dale 的代码被集成到整个系统中，并且能够很好地运行。之后，在 Dale 的模块中，出现了一个小问题；但是 Dale 很快就发现了问题所在，在几分钟之内就把问题解决了。

现在，是该总结一下上面这个小故事的时候了：Dale 和 Pat 的年纪相当，编码能力相当，智力也差不多。唯一的区别就是 Dale 非常相信单元测试；对于每个新写的函数，在其他代码使用这个函数并对它形成依赖之前，都要先做单元测试。

而 Pat 则没有这么做，他总是“知道”代码的行为应该和所期望的完全一样，并且等到所有代码都差不多写完的时候，才想起来运行一下代码。然而到了这个时候，要想定位 bug，或者，甚至是确定哪些代码的行为是正确的，哪些代码的行为是错误的，都为时已晚了。

1.2 什么是单元测试

单元测试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，你可能把一个很大的值放入一个有序 list 中去，然后确认该值出现在 list 的尾部。或者，你可能会从字符串中删除匹配某种模式的字符，然后确认字符串确实不再包含这些字符了。

执行单元测试，是为了证明某段代码的行为确实和开发者所期望的一致。

对于客户或最终使用者而言，这种测试必要吗，它与验收测试有关吗？这个问题仍然很难回答。事实上，我们在此并不关心整个产品的确认、验证和正确性等等；甚至此时，我们都不去关心性能方面的问题。我们所要做的一切就是要证明代码的行为和我们的期望一致。因此，我们所要测试的是规模很小的、非常独立的功能片断。通过对所有单独部分的行为建立起信心，确信它们都和我们的期望一致；然后，我们才能开始组装和测试整个系统。

毕竟，要是我们对手上正在写的代码的行为是否和我们的期望一致都没把握，那么其他形式的测试也都只能是浪费时间而已。在单元测试之后，你还需要其他形式的测试，有可能是更正规的测试，那一切就都要看环境的需要来决定了。总之，做测试如同做善事，总是要从家[1]开始。

1.3 为什么要使用单元测试

单元测试不但会使你的工作完成得更轻松，而且会令你的设计变得更好，甚至大大减少你花在调试上面的时间。

在我们上面的小故事里面，Pat 因为假设底层的代码是正确无误的而卷入麻烦之中，先是在高层代码中使用了底层代码；然后这些高层代码又被更高层的代码所使用，如此往复。在对这些代码的行为没有任何信心的前提下，Pat 等于是在假设上面用竖立卡片堆砌了一间房子——只要将下面卡片轻轻移动，整间房子就会轰然倒塌。

当基本的底层代码不再可靠时，那么必需的改动就无法只局限在底层。虽然你可以修正底层的问题，但是这些对底层代码的修改必然会影响到高层代码，于是高层代码也连带地需要修改；以此递推，就很可能会动到更高层的代码。于是，一个对底层代码的修正，可能会导致对几乎所有代码的一连串改动，从而使修改越来越多，也越来越复杂。于是，整间由卡片堆成的房子就由此倒塌，从而使整个项目也以失败告终。

Pat 总是说："这怎么可能呢？"或者"我实在想不明白为什么会这样"。如果你发现自己有时候也会有这种想法，那么这通常是你对自己的代码还缺乏足够信心的表现——你并不能确认哪些是工作正常的而哪些不是。

为了获得 Dale 所具有的那种对代码的信心，你需要"询问"代码究竟做了什么，并检查所产生的结果是否确实和你所期望的一致。

这个简单的想法描述了单元测试的核心内涵：这个简单有效的技术就是为了令代码变得更加完美。

---

[1] 译注：begins at home，这里的家指的是代码最基本的正确性。

1.4 我需要做什么呢

引入单元测试是很简单的，因为它本身就充满了乐趣。然而在项目交付的时候，我们给客户和最终用户的仍然是产品代码，而不包含单元测试的代码；因此，我们必须对单元测试的目的有充分的认识。首先也是最重要的，使用单元测试是为了使你的工作——以及你队友的工作——完成得更加轻松。

■ 它的行为和我的期望一致吗？

最根本的，你想要回答下面这个问题：“这段代码达到我的目的了吗？”也许就需求而言，代码所做的是错误的事情，但那是另外一个问题了。你要的是代码向你证明它所做的就是**你**所期望的。

■ 它的行为一直和我的期望一致吗？

许多开发者说他们只编写一个测试。也就是让所有代码从头到尾跑一次，只测试代码的一条正确执行路径，只要这样走一遍下来没有问题，测试也就算是完成了。

但是，现实生活当然不会这么事事顺心，事情也不总是那么美好：代码会抛出异常，硬盘会没有剩余空间，网络会掉线，缓冲区会溢出等——而我们写的代码也会出现 bug。这就是软件开发的“工程”部分。就“工程”而言，土木工程师在设计一座桥梁的时候，必须考虑桥梁的负载、强风的影响、地震、洪水等等。电子工程师要考虑频率漂移、电压尖峰、噪音，甚至这些同时出现时所带来的问题。

你不能这样来测试一座桥梁：在风和日丽的某一天，仅让一辆车顺利地开过这座桥。显然，这种测试对于桥梁测试来说是远远不够的。类似地，在测试某段代码的行为是否和你的期望一致时，你需要确认：在任何情况下，这段代码是否都和你的期望一致；譬如在风很大、参数很可疑、硬盘没有剩余空间、网络掉线的时候。

■ 我可以依赖单元测试吗？

不能依赖的代码是没有多大用处的。但更糟糕的是，那些你自认为可以信赖的代码（但是结果证明这些代码是有 bug 的）有时候也会让你花很多时间在跟踪和调试上面。显然，几乎没有项目可以允许你在这上面浪费太多的时间，因此无论如何，你都要避免这种“前进一步，后退两步”的开发方法。也就是说，要让开发过程保持稳定的步伐前进。

没人能够写出完美无缺的代码；但是这并没有关系——只要你知道问题的所在就足够了。许多大型软件项目的失败，诸如只能把坏了的太空船搁浅在遥远的行星，或者在飞行的途中就爆炸了，都能通过认知软件的限制来避免。例如，Arianne 5 号火箭软件重用了来自于之前一个火箭项目的一个程序库，而这个程序库并不能处理新火箭的飞行高度（比原来火箭要高）[2]，从而在起飞 40 秒之后就发生了爆炸，导致 5 亿美元的损失。

显然，我们希望能够依赖于所编写的代码，并且清楚地知道这些代码的功能和约束。

例如，假设你写了一个反转数值序列的方法。在测试的过程中，你也许会传一个空序列给这个程序——但导致了程序崩溃。实际上，程序并没有要求该程序必须能够接收一个空序列，因此你可以只在方法的注释中说明这个约束：如果传递一个空序列给这个方法，那么这个方法将会抛出一个异常。现在你马上就知道了该代码的约束，从而也就不需要用其他很麻烦的方法来解决这个问题（因为在某些地点要解决这个问题并不方便，比如在高空大气层中）。

■ 单元测试说明我的意图了吗？

对于单元测试而言，一个最让人高兴的意外收获就是它能够帮助你充分理解代码的用法。从效果上而言，单元测试就像是能执行的文档，说明了在你用各种条件调用代码时，你所能期望这段代码完成的功能。

---

[2] 致航空迷：数值溢出的原因是由于新轨道增加了火箭的垂直加速度，从而出现了一个更大的水平倾斜。

项目成员能够通过查看单元测试来找到如何使用你所写代码的例子。如果他偶然发现了一个你没有考虑到的测试用例，那么他也可以很快地知道这个事实：你的代码可能并不支持这个用例。

显然，在正确性方面，可执行的文档有它的优势。与普通的文档不同的是，单元测试不会出现与代码不一致的情况（当然，除非你选择不运行这些测试）。

1.5 如何进行单元测试

单元测试本来就是一项简单易学的技术；但是如果能够遵循一些指导性原则(guideline)和基本步骤，那么学习将会变得更加容易和有效。

首先要考虑的是在编写这些测试方法之前，如何测试那些可疑的方法。有了这样一个大概的想法之后，你将可以在编写实现代码的时候，或者之前，编写测试代码本身。

下一步，你需要运行测试本身，或者同时运行系统模块的所有其他测试，甚至运行整个系统的测试，前提是这些测试运行起来相对比较快。在此，我们要确保所有的测试都能够通过，而不只是新写的测试能够通过；这一点是非常重要的。也就是说，在保证不引入直接 bug 的同时，你也要保证不会给其他的测试带来破坏。

在这个测试过程中，我们需要确认每个测试究竟是通过了还是失败了——但这并不意味着你或者其他倒霉的人需要查看每个输出，然后才决定这些代码是正确的，还是错误的。在此，你慢慢地就会养成一个习惯：只要用眼睛瞄一下测试结果，就可以马上知道所有代码是否都是正确的，或者哪些代码是有问题的。关于这个问题，我们将留在讨论如何使用单元测试框架时来具体讨论。

1.6 不写测试的借口

在听过了我们合情合理、热情洋溢的阐述之后，某些开发者会点头并且同意单元测试的必要性，但是也许仍然会告诉我们由于某些原因，不能编写

单元测试。下面就是我们所听到的一些借口，当然其中也包含了我们的辩解。

Joe

什么是间接损害？

间接损害是：在整个系统中，当某一部分加入了新特性，或者修复了一个 bug 之后，给系统的其他（与前面可能是互不相关的）部分引入了一个新的 bug（或者损害）。如果无视这种损害并且继续开发的话，那么将可能带来一个很危险的问题，最后可能会导致整个系统崩溃，并且没人能够修复。

我们有时候把这种损害称为“Whac-a-Mole”效应。在狂欢节中有一种游戏叫做“Whac-a-Mole”，在此游戏中，当小孔中的机械鼠探出头来的时候，参与者的任务就是敲这个机械鼠的头部，然而正在这个时候，机械鼠的头马上又缩了回去，不让你打个正着；而位于另一个孔的另一个机械鼠此时又探出头来。就这样，机械鼠的头以非常快的速度探出又缩回，使你总是打不到他，从而也就得不了分。最后，参与者通常只能无助地乱敲；然而机械鼠的动作却总是出乎你的意料。

实际上，对于代码而言，大范围的间接损害就具有这样类似的效果。

编写单元测试太花时间了

对于开始做单元测试的初学者而言，这是出现次数最多的借口。当然，事实并不是这样的。首先，我们需要关注一下：在编写代码的时候，你在哪些地方花费了更多的时间。

大多数人都把测试看成某种只有在项目结束时才做的工作。但是，如果你到了那个时候才做单元测试的话，那么肯定会花费更多的时间。事实上，要想只在项目快要结束的时候才做单元测试，那简直是扯淡。

至少，我们可以这样来看待单元测试：假设你想用一台割草机来清除两英亩的草地。如果你在草很稀疏的时候就开始行动，那么工作将会很简单。但是如果你等到这块草地长上粗壮大树、缠绕灌木的时候才准备行动，那么

工作将会变得非常困难。

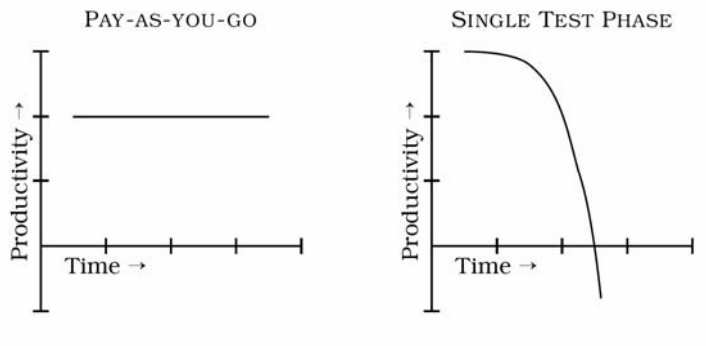

图1.1 “立即测试”和“单一测试阶段”的比较

从长远看来，使用“立即测试模型”的代价比“延后测试模型”的代价要低。在你编写实现代码的时候，同时编写独立的测试代码，在项目最后就可以避免出现做了无用功的问题；代码中的bug也会更少，因为你所依赖的都是已经通过测试的代码。于是，通过在开发过程中多花一点时间在编写单元测试上面，你就可以最小化在项目后期花费大量时间的风险。

从图 1.1 中你可以看到，“立即测试”和“延后测试”之间并没有权衡可言；而是直线效率和指数效率之间的对比，而且对于后者而言，复杂度会不断增加，并且在项目后期，很多工作需要从头再来。所有这些额外的工作都会影响你的工作效率，如图 1.1 所示。

显然，单元测试也并非免费的午餐。在立即测试模型中，单元测试是有开销的（在时间和金钱上面）。但是如果你查看右边曲线的方向，你会发现它花费了更多的开销——效率曲线急剧下降；而且生产率甚至会变成负值；这些生产率损耗可以很容易导致一个项目失败。

因此，如果你仍然认为在编写产品代码的时候，还是没有时间编写测试代码，那么请先考虑下面这些问题：

1. 对于所编写的代码，你在调试上面花了多少时间？

2. 对于以前你自认为正确的代码，而实际上这些代码却存在重大的 bug，你花了多少时间在重新确认这些代码上面？

3. 对于一个别人报告的 bug，你花了多少时间才找出导致这个 bug 的源码位置？

对于那些没有使用单元测试的程序员而言，上面这些问题所耗费的时间的递增速度是很快的，而且随着项目的深入，递增速度会变得更快；而另一方面，适当的单元测试却可以很大程度地减少这些时间，从而能够为你腾出足够的时间来编写所有的单元测试——甚至可能还有剩余的空闲时间。

运行测试的时间太长了

合适的测试是不会让这种情况发生的。实际上，大多数测试的执行都是非常快的，因此你在几秒之内就可以运行成千上万个测试。但是有时某些测试会花费很长的时间，因此我们不能每次都运行这些测试，有时你就要暂时停止运行这些测试。

在这种情况下，需要把这些耗时的测试和其他测试分开。通常可以每天运行这种测试一次，或者几天一次；而对于运行很快的测试，则可以经常运行。

测试代码并不是我的工作

这是一个很有趣的借口。请问，到底什么是你的工作呢？假设你的工作只是为了编写产品代码。如果对于那些没有把握的代码，你就随便地扔给测试组，那么你实际上并没有完成你的工作。实际上，期望别人来清理我们的代码是很不好的做法；甚至在某种情况下，如果别人指出了一大摞有错误的代码，那么也意味着你的职位也就到此为止了。

另一方面，如果测试员或者 QA（Quality Assurance）组发现很难在你的代码中挑出错误，那么你的名声将会一路高升——同时提升的还有你的职位。

我并不清楚代码的行为，所以也就无从测试

如果你实在不清楚代码的行为，那么估计现在并不是编码的时候。或许你应该先建立一个原型，这样才有助于你认清需求。

如果你并不知道代码的行为，那么你又如何知道你编写的代码是正确的呢？

但是这些代码都能够编译通过

OK，还没有人把这个当成一个借口，至少没有大声地说出来。但是估计这很容易会成为一个借口，因为有些人总是认为：一个成功的编译就是成功的标记；通过了编译，就像通过了天堂的大门一样。

但是，编译器的默许只是对代码一种非常粗略的肯定。实际上，任何编译器和解释器都只能验证你的语法是否正确，并不能保证代码的行为也正确。例如，Java 的编译器可以很容易地检测到下面这行代码是错误的：

```
public statuc void main(String args[]) {
```

上面有个明显的打印错误，第 2 个单词应该是 `static`，而不是 `statuc`。这是很容易看出来的。但是现在假设你编写了下面这些代码：

```
public void addit(Object anObject){
  List myList = new List;
  myList.add(anObject);
  myList.add(anObject);
  // more code...
}
```

你是否想加同一个对象到相同 list 两次呢？或许是，或许不是。就这一点来说，编译器并不能告诉你，只有你知道你期望代码完成什么样的功能[3]。

公司请我来是为了写代码，而不是写测试

如果使用同样的逻辑，那么我们可以说：公司付给你薪水，并不是让你整天都在调试代码。显然，公司付给你薪水是为了让你编写产品代码，而单元测试大体上是一个工具，是一个和编辑器、开发环境、编译器等处于同一位置的工具。

[3] 基于你写好的代码产生测试的自动化测试工具也存在同样的问题——它们仅仅能够使用你写出来的东西，而不能知道你的意图是什么。

如果我让测试员或者 QA（Quality Assurance）人员没有工作，那么我会觉得很内疚

你并不需要担心这些。请记住，我们在此只是谈论单元测试，而它只是一种针对源码的、低层次的，为程序员而设计的测试。在整个项目中，还有其他的很多测试需要这些人来完成，如：功能测试、验收测试、性能测试、环境测试、有效性测试、正确性测试、正规分析等等。

我的公司并不会让我在真实系统中运行单元测试

Whoa！我们所讨论的只是针对开发者的单元测试。也就是说，如果你可以在其他的环境下（例如在正式的产品系统中）运行这些测试的话，那么它们就不再是单元测试，而是其他类型的测试了。实际上，你可以在你的本机运行单元测试，使用你自己的数据库，或者使用 mock 对象（见第 6 章）。

如果 QA 部门或者其他测试人员希望在产品或者其他阶段运行这些测试的话，你可以调整一些技术细节，从而使他们也可以运行一些测试；但是同时要让他们清楚，这些调整后的测试已经不再是单元测试了。

1.7 本书概要

第 2 章“你的首个单元测试”将会对单元测试做一个整体的概括；同时我们还要介绍一些编写单元测试的规则，而主要的规则我们将留在第 3 章“使用 JUnit 编写测试”中介绍。接下来，我们将会用几章的篇幅来介绍如何发现哪些部分需要测试，以及如何对他们进行测试。

在第 7 章，我们将会介绍优秀测试所具有的一些品质。在第 8 章，我们会针对你自己的项目，介绍高效使用单元测试的一些前提条件，这一章同时还讨论了如何处理具有遗留代码的项目。第 9 章“设计话题”将会介绍测试如何影响整个项目的设计（从好的方面而言）。

附录包含了很多额外的有用信息：一些常见的与单元测试相关的问题，关于安装 JUnit 的方法，一个样板 JUnit 骨架程序和一些有用的资源，同时包含了参考书目。我们最后还给出了一些总结性的句子，介绍了本书的一些要点和建议。

现在，让我们坐下来，放松一下，开始进入编写更好代码的世界。