# 如何写好单元测试

## 代码规范性的审查

1. 代码规范性的审查将帮助于更早地发现缺陷，代码质量的提高，

2. 而且可以帮助我们遵守规则、养成好的习惯，以达到预防缺陷的目的

3. 代码风格和编程规则两者不可缺一，都应列入代码评审的范围

4. 命名规则、缩进于对齐、空行和空格、注释和函数处理等

## 1. 单元测试作用？

**缺点：**增加代码量。测试代码是系统代码的两倍或更多，但是同时节省了调试程序及挑错时间。

**优点：**

1. 提升软件质量、减少bug、提升反馈速度、减少重复工作、提高开发效率。 2. 提升开发者对代码的自信心（无论是自己的代码还是团队其他成员写的代码都有可能因此受益），而且无论是测试即有代码还是为编写新的代码，亦或是为重构老代码做准备，这一点（保证软件质量和提升自信心）都是成立的。

3. 单元测试用例是保证我们在进行重构时，不会影响到代码的外部接口功能，让我们进行的重构是十分安全的。

单元测试和集成测试时IT软件开发过程中两个不同的阶段。 首先，两者的出发点是一致的，都是保证项目的健壮性、可靠性、正确性而言的。

## 单元测试和集成实测区别?

**测试粒度不同：单元测试的颗粒度是在单只程序上,集成测试的颗粒度则在整个系统上.**

**单元测试的目的：**

是对已实现的软件最小单元进行测试，已保证软件系统中各个单元的质量，单元测试强调被测试对象的独立性即隔离性。

**集成测试的目的：**

也叫组装测试或者联合测试。在单元测试的基础上，将所有模块按照设计要求(如根据结构图)组装成为子系统或者系统，进行集成测试。实践表名，一些模块虽然能够单独地工作，但并不能保证连起来也能正常的工作。程序的某些局部反应不出来的问题，在全局上很有可能暴露出来，影响功能的实现

是为了检验各单元协同工作的成果，因此它就是单元测试的直接“上级”了。有时候很容易把集成测试和单元测试混为一谈，这是因为集成测试可以使用和单元测试一样的工具／框架来编写，也就是说集成测试也可以践行 TDD 的实践原则；当然，集成测试也可以用 BDD 向的工具／框架来编写，这个界限是非常模糊的。

****牢牢记住：集成测试和单元测试的区别不在于使用了什么样的工具／框架，也不在于践行了哪种测试驱动方法，而在于它们谁更看重对待测代码的隔离性****

## 3. 先写代码还是先写测试？

TDD是测试驱动开发（Test-Driven Development）的英文简称，是敏捷开发中的一项核心实践和技术，也是一种设计方法论。TDD的原理是在开发功能代码之前，**先编写单元测试用例代码，测试代码确定需要编写什么产品代码**。TDD虽是敏捷方法的核心实践，但不只适用于XP（Extreme Programming），同样可以适用于其他开发方法和过程。

**TDD带来的好处有：**

1、你会更加站在用户的角度去看你将要完成的产品，你要尽可能想到用户所有进行的 操作。而不是从程序员的角度想用户应该会如何去使用我们的产品。

2、测试用例是在对功能进行测试。在写代码之前先写测试用例，可以对我们编写代码 提供指导性的参考。防止我们漏掉一些功能。

3、它使我们对自己代码有了信心，因为我们事先设计好的所有测试用例都Pass了。

4、如果在更改代码后测试用例不能通过，我们可以根据不能通过的测试用例精确的定 位问题，并轻而易举的解决的这个bug。

5、哈！我们的一整套完备的测试用例在这里替我们把关（把的什么关？），我们就可 以十分安全的使用极限编程的另一个最重要的工具——重构。重构改变的是代码的内部 结构，而不会改变外部接口功能。知道在做重构时测试用例是把的什么关了吧！很明显， 测试用例是保证我们在进行重构时，不会影响到代码的外部接口功能。所以我刚刚说， 我们进行的重构是十分安全的。

6、基于第5点，我们找到了重构的信心，必要时候你还可以痛痛快快的并且满怀信心 的对代码做一场大的变革。这样我们的代码变得干净了，扩展性、可以维护性以及易理 解性纷至沓来。

TDD有这么多好处，但它也不是免费的午餐，它需要我们有设计完备的测试用例的能力 （这项能力是长期理论与实践的结合体），否则你将会吃了亏编写了一大堆测试用例， 却没测到点子上。可怕的是，你还对你“测试通过”的糟糕的代码满怀信心。

TDD的主旨是高效，可能这个高效不是非常高的开发速度。

通常的软件开发过程是先写功能，然后再写测试。甚至有时候只进行某些方面的测试， 或者有省事的就略去了某些细小功能的测试，或者是忘了对这些模块的测试。

这些看起来对软件影响不大，但是似乎有点过于自信了。因为我们是人，不是神，所以 难免会出一些这样或那样的错误，而这些小的问题在项目整合起来以后进行排错是非常 令人头疼的。

TDD的思想是以测试推动开发进程。因为我们在软件开发之前，每个程序单元的功能都 已经确定了。程序员在理解完整个程序需求以后，直接进行开发，有可能会因为种种原 因考虑不很周全，似乎功能实现的没有问题了，但是其中却可能隐藏着非常可怕的Bug。 TDD促使开发人员先根据程序单元的功能编写测试代码，就像是先建一个模型，然后向 里面浇注合适功能的代码。最后满足所有的测试验证了，才能正常通过测试，这个程序 单元才算完成。

这样消除了开发人员主观性的对程序单元健壮性的评估，更客观的验证每一个程序单元 的功能实现以及可能出现的Bug。

当然，这些操作都需要有大量的代码支持，所以费事是在所难免的，但是这点“费事” 与健壮性非常强的代码相比，有些人还是偏向于使用TDD。

**首先站在客户方代码的立场，可以获得更好的api。**

**其次可以改善软件质量，支持重构。**

**其三，大幅减少debug时间。**

前期投入大，后期能大幅缩短开销，将问题发现在最源头

提供明确的目标

你很清楚， 一旦结束（测试通过）， 你的工作就完成了（假设你的测试写的很好）. 测试代码会为代码建立一个自然的边界， 使你把重点集中在当前任务上. 一旦测试通过， 就有确切的证据证明你的代码能工作. 相对于人工的测试用户界面或者比较日志文件中的结果， 在一个xUnit框架中运行自动化测试， 速度要快几个数量级. 大多数xUnit测试的运行只需几微秒， 而且大多数采用TDD的人都会一天运行数次测试. 在许多开发小组中， 将代码上传配置库前， 必须成功地通过测试。

## 4. 谁来编写什么样的测试?

单元测试也只能是程序员自己的责任吧？话是没错，但我们现在不是假设理想情况吗？所以理想状况下，即使**单元测试只能由程序员来编写**，**程序员也（最好）不要针对自己实现的代码来编写单元测试**。谁来？其他程序员。

没错，我说的就是**结对编程**。给自己的代码编写单元测试当然不是不可以，但是一个人的思路永远是不够开放的——老祖宗说了：不识庐山真面目，只缘身在此山中嘛。你能保证自己的思路完全正确吗？你能保证自己已经考虑到了所有的边界条件吗？你确信只有这样也是解决问题的最好方式吗？

谁都不能完全做到，即使是结对编程也还是不足以十全十美，但换一个人来为你描述问题往往能收获非常好的效果。我清楚结对编程的代价，但是很多人都不了解结对编程的巨大好处，为同事编写测试（或反过来）只是其中之一，个中妙处你只有经历过才知道，这里就不多谈了。总之如果有什么办法能迅速提高单元测试的质量，我的答案就是结对编程。

## 如何避免无用的测试?

### 5.1 只写必要的测试

**准备编写自己觉得“没谱”的代码：**觉得“没谱”可能存在多种原因，比如说业务逻辑很复杂自己没完全吃透；比如说脑中大概有一个思路，但由于以前没写过所以吃不准是不是能行，也无法确定过程中是否会产生难以预料的变数等等。

### 5.2 只写关键的测试：

有时候必要的测试你写不出来，身边又没有人辅导，那也勉强可以跳过。但是关键性测试不要省。所谓关键性的测试，就是你所写代码里的核心逻辑；再换句话说就是如果一切顺利，它至少能够做到（或者不要去做）的那件事。这就意味着你可能忽略了一些边界条件的处理，而且你也不知道该怎么处理，但是你至少保证了最重要的那条路线是可以走通的。将来重构的时候，这条关键路线就像夜空中的北极星一样能确保你不至于茫然无措。

如果在构造关键性测试用例的时候你发现你很难触碰到那一点（比如说前置条件你不会在测试用例里处理），那么很大的可能是你的这个单元过于复杂了，这是一个极好的立时重构信号。你可以尝试把要触碰的那一点逻辑抽取出来单独测试。这样一来你至少做到了把核心逻辑分离出来，其他的代码就算再糟糕重构起来也会轻松得多。

### 5.3 无用的测试：

**5.3.1 不要去测试语言的核心库和／或标准库函数：**

如果你的代码简单到就调用了一句标准库函数，那还浪费什么时间去编写测试啊，这些代码都是久经考验的。虽然也有语言本身错误的小概率事件发生，但由于标准函数的处理过程你触碰不到（常常深埋于虚拟机中或调用系统底层接口）所以你的测试对你自己的代码丝毫没有帮助。（当然，如果您是专家级程序员则不在此例，说不准您就是等着解决这个问题呢）

**5.3.2不要去测试框架的基础类或工具方法：**

道理和第一条类似，知名的框架都有很完备的自身测试，否则你也不敢用不是？如果你确信是框架自身出了问题，你的测试更应该去应用在框架本身上，说不定你可以做出个补丁为该项目做出贡献。

顺手举个例子：你继承了某框架的 Model 层，然后在里面定义了检查其实例的某一属性是否为空的验证（使用框架自带的验证方法，而不是自己编写的）。这种情况就没有必要测试这个检查是否生效，除非你这个类在初始化的时候返回的是其他类的实例……你项目组里有这么无聊的人吗？

**5.3.3 不要去测试外部依赖的有效性：**

这是初学者常常陷的坑，而且往往把自己折磨到不行。

这里有两个问题：第一，如果你的测试一定需要外部依赖，你首先应该考虑伪造它，而不是在 A 的测试里先检查 B（也就是说，你的测试目标是 A，为了完成这个测试用例，你需要用到 B 并且 B 的某种特性一定要成立，这是先决条件，于是你不得不写一句断言先测试这件事情，然后才能测试真正的目标 A）。如果你能伪造一个 B，叫 B'，那么 B' 不一定非要和 B 完全一样，只要它能表现出来恰好满足本次测试用例的特征就足够了。这样事情就会变得单纯的多。

其次，即使你无法伪造 B（基本上是因为不会），那么你至少应该把对 B 的特性测试转移到 B 自己的单元测试中去。

**5.3.4 最后还有一种测试是“无用”的:**

那就是从来只见它绿（**通过**）没见过它红（**失败**）的测试。你自己都没意识到这种测试可能从头到尾都没有测试任何代码！这也是 TDD 强调先红后绿再重构的原因之一，你至少应该在最开始让测试用例失败一次，否则等测试数量变多以后再去分辨就来不及了。另外重构完了也最好手动破坏一下代码（比如随便往里面打几个无意义的字符）诱使测试报错，以确保测试真的覆盖到了目标代码。

## 测试的代码覆盖率?

我对“无用测试”的态度已经揭示出我对代码覆盖率的态度：无视它。我一直认为代码覆盖率是最形式主义的技术工具，覆盖率再高也不能保证代码本身无懈可击，该出 Bug 的地方 100% 的覆盖率也救不了你。

其实作为一种辅助度量工具，代码覆盖本身并没有什么错，有位仁兄说得好：“在追求精益求精的道路上，我们应该无所不用其极”。错就错在拿代码覆盖率当考核指标，以此来衡量测试人员的工作水平，对此我相当无语，也相当反感。

有识之士一定会说：你也不要以偏概全，路径覆盖所度量出的代码覆盖率还是相当靠谱的嘛。

简单普及下，代码覆盖算法有很多种，大致上对比准确性：路径覆盖 > 条件覆盖 ~= 判定覆盖 > 语句覆盖。而且这只是说条件分支，循环什么的还有别的算法就不多说了。这些算法在覆盖率都达到 100% 的前提下，其“靠谱”程度可能有天壤之别。问题就出在下决策使用代码覆盖率做考核的人往往不明白这种差别，这就给了落地执行的人可趁之机，很容易就演变成了“在追求 100% 代码覆盖率的道路上，我们应该无所不用其极“。若是连落地执行人都不懂，那就更悲剧了，一群人对着水份极大的 100% 乐得嘴都合不拢，想想都难受。

**所以对于代码覆盖率的不当应用，只会让大家越走越偏，浪费时间不说收效还甚微**；反过来恰当的使用代码覆盖率又对团队的要求极高，只有一个人懂行是不够的，因为你没有那么多时间精力去检查结果是不是真的靠谱。如果每一个人都按照靠谱的方式去写代码和测试，不用测试覆盖率也没什么大不了的。因此如果我是初创团队的负责人，我宁可**选择把时间和精力放在测试用例本身上，测试本身靠谱了，测试覆盖率的辅助价值才能靠谱**。

## 7. 测试中的”伪装术”(mock仿件、stubs桩件、)

有时候对被测系统(mock仿件、stubs桩件)进行测试是很困难的，因为它依赖于其他无法在测试环境中使用的对象、组件、或者api。这有可能是因为**这些(对象、组件、api)不可用**，**它们不会返回测试所需要的结果**，**或者执行它们会有不良副作用**。在其他情况下，我们的测试策略要求对被测系统的内部行为有更多控制或更多可见性。

如果在编写测试时无法使用（或选择不使用）实际的依赖组件(DOC)，可以用测试替身来代替。测试替身不需要和真正的依赖组件有完全一样的的行为方式；**他只需要提供和真正的组件同样的 API 即可，这样被测系统就会以为它是真正的组件**！

**===============demo start============**

例如：

1. 我在编写单元测试**Cart**购物车类，依赖**Product**产品类和**User**用户类
2. 依赖**Product**产品类和**User**用户类已经测试过了
3. 或者依赖**Product**产品类和**User**用户类是由其他人写的

问题：

1. Product产品类和User用户类一旦出现情况不至于让你误以为是Cart类的代码出了问题。

2. 不用为了创造很多前置条件，才能做出断言。(如果这样应该放在集成测试)

**===============demo end============**

**什么情况下使用(mock仿件、stubs桩件、)：**

1. **如果外部依赖不存在，则测试肯定无法通过**

2. **如果外部依赖不会返回测试所需的结果，或者执行它会有不良的副作用。**

3. 如果外部依赖变更，则会导致测试失败。严格来说这种后果不是测试的责任**，外部依赖的变更应该保持外部接口不变和返回结果不变，只变更内部的行为。使用伪装术的好处就在于一旦出现这种情况不至于让你误以为是己方的代码出了问题**。

当然你也会想，如果用了伪装对象，那么外部依赖变了己方的测试还浑然不知，这不是很危险吗？有道理，不过单元测试的职责是测试己方代码的正确性，对于外部依赖的模拟不一定非得和模拟对象完全一致，真实的交互应该先由集成测试来捕捉问题，否则很容易迷失在复杂的代码交互之中。

## 编写更好的单元测试要点

单元测试在的工作中使用比较广泛，记住，编写糟糕的测试是在浪费时间，并会在以后造成更大的问题。所以最好把这些准则记在心里。

1. 不应该编写成功通过的单元测试-它们应该被写成不通过的。你可以在几分钟内让任何一组测试通过，但这只是在欺骗你自己。

2. 测试类应该只测试一个功能-你应该用一个功能去测试一个方法。否则，你会违反了单一职责原则。

3. 测试类具备可读性-确保测试类标有注释并且容易理解，就像其他的代码一样。

4. 良好的命名规范-再次测试时应该像其他代码一样-便于人们理解。

5. 把断言从行为中分离出来-你的断言应该用来检验结果，而不是执行逻辑操作的。

6. 使用具体的输入-不要使用任何的自动化测试数据来输入，像date()这些产生的数据会引入差异。

7. 把测试类分类，放在不同的地方-从逻辑的角度看，当没有错误指向特定的问题时这更容易去查找。

8. 好的测试都是一些独立的测试类-你应该让测试类与其他的测试、环境设置等没有任何依赖。这利于创建多个测试点。

9. 不要包含私有的方法-他们都是一些具体的实现，不应该包含在单元测试里。

10. 不要连接数据库或者数据源-这是不靠谱的。因为你不能确保数据服务总是一样的并且能够创建测试点。

11. 一个测试不要超过一个模拟(mock对象)-我们努力去消除错误和不一致性。

12. 单元测试不是集成测试-如果你想测试结果，不要使用单元测试。

13. 测试必须具有确定性-你需要一个确定的预测结果，所以，如果有时候测试通过了，但是不意味着完成测试了。

14. 保持你的测试是幂等的-你应该能够运行你的测试多次而不改变它的输出结果，并且测试也不应该改变任何的数据或者添加任何东西。无论是运行一次还是一百万次，它的效果都应该是一样的。

15. 测试类一次仅测试一个类，测试方法一次仅测试一个方法-组织方法能够在问题出现时检测出来，并帮你确定测试依赖。

16. 在你的测试里使用异常-你在测试里会遇到异常，所以，请不要忽略它，要使用它。

17. 不要使用你自己的测试类去测试第三方库的功能-大多数好的库都应该有它们自己的测试，如果没考虑用mocks去产生一致性的结果的话。

18. 限制规则-当在一些规则下写测试时，记住你的限制和它们（最小和最大）设置成最大的一致性。

19. 测试类不应该需要配置或者自定义安装-你的测试类应该能够给任何人使用并且使它运行。“在我的机器上运行”不应该出现在这。