测试的道理

在长期的程序语言研究和实际工作中，我摸索出了一些关于测试的道理。然而在我工作过的每一个公司，我发现绝大多数人都不明白这些道理，很多团队集体性的采用错误的做法而不自知。很多人把测试当成一种主义和教条，进行过度的测试，不必要的测试，不可靠的测试，并且把这些错误的做法传授给新手，造成恶性循环。本来目的是提高代码质量，结果不但没能达到目的，反而降低了代码质量，增大了工作量，大幅度延缓工程进度。

我也写测试，但我的测试方式比“测试教条主义者”们的方式聪明很多。在我心目中，代码本身的地位大大的高于测试。我不忽视测试，但我不会本末倒置，过分强调测试，我并不推崇测试驱动开发（TDD）。我知道该测试什么，不该测试什么，什么时候该写测试，什么时候不该写，什么时候应该推迟测试，什么时候完全不需要测试。因为这个原因，再加上高强的编程能力，我多次完成别人认为在短时间不可能完成的任务，并且制造出质量非常高的代码。

测试的道理

现在我就把这些自己领悟到的关于测试的道理总结一下，其中有一些是鲜为人知或者被误解的。

1. 不要以为你处处显示出“重视代码质量”的态度，就能提高代码质量。总有些人，以为自己知道“单元测试”（unit test），“集成测试”（integration test）这样的名词，就很懂编程，就可以教育其他人。可惜，光有态度和口号是不解决问题的，你还必须有实战的技巧，深入的见解和智慧，必须切实地知道应该怎么做。代码的质量不会因为你重视它就得到提升，也不会因为你采取了措施（比如测试，静态分析）就一定会得到改善。你必须知道什么时候该写测试，什么时候不该写测试，需要写测试的时候，要写什么样的测试。其实，提高代码质量唯一可行的手段不是写测试，而是反复的提炼自己的思维，写简单清晰的代码。如果你想真的提高代码质量，我的文章『编程的智慧』是一个不错的出发点。

2. 真正的编程高手不会被测试捆住手脚。是的，你身边那个你认为“不很在乎测试”的家伙，也许是个比你更好的程序员。我喜欢把编程比喻成开赛车，而测试就是放在路边用来防撞的轮胎护栏……

   

   护栏有时候是很有用，可以救命的，然而一个合格的车手，绝对不会一心想着有护栏保护，测试在编程活动中的地位也应该就是这样。优秀的车手会很快看见优雅而简单的路径，恰到好处地掌握速度和时机，直奔终点而去。护栏只是放在最危险的地段，让你出了意外不要死得太惨。护栏并不能让你成为好的车手，不能让你取得冠军。绝大多数时候，你的安全只有靠自己的技术，而不是护栏，你永远有办法可以撞死自己。测试的作用也是一样，即使有了很多的测试，代码的安全仍然只掌握在你的手里。你永远可以制造出新的 bug，而没有测试可以检测到它……

   通常情况下，一个合格的车手是根本碰不到这些护栏的，他们心里想的是更高的目标：快点到达终点。相比之下，一个不合格的车手，他经常撞到赛道外面去，所以在他的心里，护栏有着至高无上的地位，所以他总是跟别人宣扬护栏的重要性。他开车的时候为了防止犯错，要在他经过的路径两边密密麻麻摆上护栏，甚至把护栏摆到赛道中间，以确保自己的转弯幅度正确。他在护栏之间跌跌撞撞，最后只能算是勉强到达终点。鼓吹测试驱动开发的人，就是这种三流车手，这种人写再多的测试也不可能倒腾出可靠的代码来。

3. 在程序和算法定型之前，不要写测试。TDD 的教条者喜欢跟你说，在写程序之前就应该先写测试。为什么写代码之前要写测试呢？这只是一种教条。这些人其实没有用自己的脑子思考过这个问题，而只是人云亦云，觉得这样“很酷”，符合潮流，或者以为这样做了别人就会认为自己是高手。实际上在程序框架完成，算法定型之前，你

都不需要写测试。如果你想知道代码是否正确，用人工方式运行代码，看看结果足以。

如果你发现编程初期需要保证的性质纷繁复杂，如此之多，不写测试你就没信心的话，那你还是想办法先提高下基本的编程技术吧：多做练习，简化代码，让代码更加模块化，看看我的『编程的智慧』或者『SICP』一类的东西。写测试并不能提高你的水平，正好相反，过早的写测试会捆住你的手脚，让你无法自由的修改代码和算法。如果你不能很快的修改代码，不能用直觉感觉到它的变化和结构，而是因为测试而处处卡顿，你的头脑里就不能产生所谓“flow”，就不能写出优雅的代码来，结果到最后你什么也没学会。只有在程序不再需要大幅度的改动之后，才是逐渐加入测试的时候。

4. 不要为了写测试而改变本来清晰的编程方式。很多人为了满足“覆盖”（coverage）的要求，为了可以测试到某些模块，或者为了使用 mock，而把本来简单清晰地代码改成更加复杂而混淆的形式，甚至采用大量 reflection。这样一来其实降低了代码的质量。本来很简单的代码，一眼看去就知道是否正确，可是现在你一眼看过去，到处都是为了方便测试而加进去的各种转接插头，再也无法感觉到代码。这些用来辅助测试的代码，阻碍了你对代码进行直觉思维，而如果你不能把代码的逻辑完全映射在头脑里（进而产生直觉），你是很难写出真正可靠的代码的。

   有些 C# 程序员，为了测试而加入大量的 interface 和 reflection，因为这样可以在测试的时候很方便的把一片代码替换成 mock。结果你就发现这程序里每个类都有一个配套的 interface，还需要写另外一个 mock 类，去实现这个 interface。这样一来，不但代码变得复杂难以理解，而且还损失了 Visual Studio 的协助功能：你不再能按一个键（F12）就直接跳转到方法的定义，而需要先跳到对应的 interface 方法，然后再找到正确的实现。所以你不再能够在代码里面快速的跳转浏览。这种方便性的损失，会大幅度降低头脑产生整体理解的机会。而且为了 mock，每一个构造函数调用都得换成一个含有 reflection 的构造，使得编译器的静态类型检查无法确保类型正确，增加运行时出错的可能性，出错信息还难以理解，得不偿失的后果。

5. 不要测试“实现细节”，因为那等同于把代码写两遍。测试应该只描述程序需要满足的“基本性质”（比如 sqrt(4) 应该等于 2），而不是去描述“实现细节”（比如具体的开平方算法的步骤）。有些人的测试过于详细，甚至把代码的每个实现步骤都兢兢业业的进行测试：第一步必须做A，第二步必须做B，第三步必须做C…… 还有些人喜欢给 UI 写测试，他们的测试里经常这样写：如果你浏览到这个页面，那么你应该在标题栏看见这行字……

   仔细想一下就会发现，这种作法本质上不过是把代码（或者UI）写了两遍而已。本来代码里面明白写着：先做A，再做B，再做C。UI 描述文件里面明白写着：标题栏里面是这些内容。你有什么必要在测试里把它们全都再检查一遍呢？这根本没有增加任何可靠性：你在代码里会犯错，你把同样的逻辑换种形式再写一遍，难道就不会错了吗？

   这就像某些脑子秀逗的人，他出门时总是担心门没锁好，关门之后要推推拉拉好几次，确认门是锁上了的。还没走几步，他仍然在怀疑门没锁好，又走回去推推拉拉好几次，却始终不能放心 :P 这种做法非但不能保证代码的正确，反而给修改代码制造了障碍。理所当然，你把同一段代码写了两遍，每当要修改代码，你就得修改两次！这样的测试就像紧箍咒一样，把代码压得密不透风。每一次修改代码，都会导致很多测试失败，以至于这些测试都不得不重写。本质上就是把代码修改了两遍，只不过更加痛苦一些。

6. 并不是每修复一个 bug 都需要写测试。很多公司都流传一个常见的教条，就是认为每修复一个 bug，都需要为它写测试，用于确保这个 bug 不再发生。甚至有人要求你这样修复一个 bug：先写一个测试，重现这个 bug，然后修复它，确保测试通过。这种思维其实是一种生搬硬套的教条主义，它会严重的减慢工程的进度，而代码的质量却不会得到提高。写测试之前，你应该仔细的思考一个问题：这个 bug 有多大可能会在同一个地方再次发生？很多低级错误一旦被看出来之后，它就不大可能在同一个地方再次出现。在这种情况下，你只需手工验证一下 bug 消失了就可以。

   为不可能再出现的 bug 大费周折，写 reproducer，构造各种数据结构去验证它，保证它下次不会再出现，其实是多此一举。同样的低级错误就算再出现，也很可能不在同一个地方。写测试不但不能保证它不再发生，而且浪费你很多时间。这测试在每次 build 的时候都会消耗时间，每次编译都因为这些测试多花几分钟，累积起来之后，你就发现工程进度明显减慢。只有当发现已有的测试没有抓住程序必须满足的重要性质时，你才应该写新的测试。你不应该是为这个 bug 而写测试，而是为代码的性质而写测试。这个测试的内容不应该只是防止这个 bug 再次发生，而是要确保 bug 所反映出来的，之前缺失的“性质”得到保证。

7. 避免使用 mock，特别是多层的 mock。很多人写测试都喜欢用很多 mock，堆积很多层，以为只有这样才能测试到路径比较深的模块。其实这样不但非常繁琐费事，而且多层的 mock 往往不能产生足够多样化的输入，不能覆盖各种边界情况。如果你发现测试需要进行多层的 mock，那你应该考虑一下，也许你需要的不是 mock，而是改写代码，让它更加模块化。如果你的代码足够模块化，你不应该需要多层的 mock 来测试它。你只需要为每一个模块准备一些输入（包括边界情况），确保它们的输出符合要求。然后你把这些模块像管道一样连接起来，形成一个更大的模块，测试它也符合输入输出要求，以此类推。

8. 不要过分重视“测试自动化”，人工测试也是测试。写测试，这个词往往隐含了“自动运行”的含义，也就是假设了要不经人工操作，完全自动的测试。打一个命令，它过一会就会告诉你哪些地方有问题。然而，人们往往忽略了“人工测试”。他们没有意识到，人工去试验，去观察，也是一种测试。所以你就发现这样的情况，由于自动测试在很多时候非常难以构造（比如，如果你要测试一段复杂的交互式GUI代码的响应），很多人花了很多时间，利用各种测试框架和工具，甚至遥控 WEB 浏览器去做一些自动操作，花太多时间却发现各种不可靠，没法测到很多东西。

其实换一个思路，他们只需要花几分钟的时间，就可以用人工的方式观察到很多深入的问题。过分的重视测试自动化的原因，往往在于一个不切实际的假设，他们假设错误会频繁的再次发生，所以自动化了可以省下人的力气。但是其实，一旦一个 bug 被修好，它反复出现的机会不会很大的。过分的要求测试自动化，不但延缓了工程进度，让程序员恼火，效率低下，而且失去了人工测试的精确性。

9. 避免写太长，太耗时的测试。很多人写测试，叽里呱啦很长一串，到后来再看的时候，他已经不记得自己当时想测什么了。有些人本来用很小的输入就可以测试到需要的性质，他却总喜欢给一个很大的输入，下意识的以为这样更加靠谱，结果这测试每次都会消耗大量的 build 时间，而其实达到的效果跟很小的输入没有任何区别。

10. 一个测试只测试一个方面，避免重复测试。有些人一个测试测很多内容，结果每次那个测试失败，都搞不清楚到底是哪个部件出了问题。有些人为了“放心”，喜欢在多个测试里面“附带”测某些他认为相关的部件，结果每次那个部件出问题，就发现好多个测试失败。如果一个测试只测一个方面，不重复测同一个部件，那么你就可以很快的根据失败的测试，发现出问题的部件和位置。

11. 避免通过比较字符串来进行测试。很多人写测试的时候，喜欢通过打印出一些东西，然后使用字符串比较的方式来决定输出是否符合要求。一个常见的做法是把输出打印成格式化的 JSON，然后对比两个文本。甚至有人 JSON 都不用，直接就比较 printf 输出的结果。这种测试是非常脆弱的。因为字符串输出的格式往往会发生微小的变化，比如有人在里面加了一个空格之类的。把这种字符串作为标准输出，进行字符串比较，很容易因为微小的改动而使大量测试失败，导致很多的测试需要做不必要的修改。正确的做法，应该是进行结构化的比较，如果你要把标准结果存成 JSON，那么你应该先 parse 出 JSON 所表示的对象，然后再进行结构化的对比。PySonar2 的测试就是这样的做法，所以相当的稳定。

12. “测试能帮助后来人”的误区。每当指出测试教条主义的错误，就会有人出来说：“测试不是为了你自己，而是为了你走了以后，以后进来的人不犯错误。” 首先，这种人根本没有看清楚我在说什么，因为我从来没有反对过合理的测试。其次，这种“测试能帮助后来人”，其实是没有经过实践检验，站不住脚的说法。如果你的代码写得很乱，就算你测试再多，后来人也无法理解，反倒被莫名其妙的测试失败给弄得更糊涂，不知道是自己错了还是测试错了。我已经说过了，测试不能完全保证代码不被改错，实际上它们防止代码被改错的作用是非常弱的。无论如何，后来人都必须理解原来的代码的逻辑，知道它在做什么，否则他们不可能做出正确的修改，就算你有再严密的测试也一样。

举一个亲身的例子。我在 Google 做出 PySonar 之后，最后一个测试都没写。第二次我回到 Google，我的上司 Steve Yegge 对我说：“你走了之后，我改了一些你的代码，真是太清晰，太好把握了，修改你的代码是一种快乐！” 这说明什么问题呢？我并不是说你可以不写测试，但这个例子说明，测试对于后来人的作用，并不是你有些人想象的那么大。创造清晰的代码才是解决这个问题的关键。

这种怕人突然走了，代码无法维护的想法，导致了一些人对测试过分的重视，但测试却不能解决这种问题。相反，如果测试太繁琐，做不必要的测试，反而容易让员工不满，容易走人，去加入在这方面更加有见地的公司。有些公司以为有了测试，就可以随便打发人走，这种想法是大错特错的。你需要明白的一个事情是，代码永远是属于写出它的那个人的，就算有测试也一样。如果核心人物真的走了，就算你有再多的测试也没用的，所以解决的方法就是把他们留住！一个有远见的公司总是通过其他的手段解决这个问题，比如优待和尊重员工，创造良好的氛围，使得他们没那么快想走。另外，公司必须注意知识的传承，防止某些代码只有一个人理解。

案例分析

有人会疑问，我凭什么可以给别人讲这些经验，我自己为此有什么成功的案例呢？所以现在来讲讲我做过的几个东西，以及我亲眼目睹的测试教条主义者们的失败案例。

Google

很多人可能听说过我在 Google 做的 PySonar。当时 Google 的队友们战战兢兢，说这么高难复杂的东西要从头做起，几乎是不可能的。特别是某位队友，一开头就吵着要我写测试，一直吵到最后，烦死我了。他们为什么这么担心呢？因为对 Python 做类型推导是非常高难度的代码，需要相当复杂的数据结构和算法，需要精通 Python 的语义实现。

作为一个训练有素的专家，我没有在乎他们的咋呼，没有信他们的教条。我按照自己的方式组织代码，进行精密的思考，设计和推理，最终在三个月之内做出了非常优雅，正确，高性能，而又容易维护的代码。PySonar 到现在仍然是世界上最先进的 Python 类型推导和索引系统，被多家公司采用，用于处理数以百万计的 Python 代码。，

如果我当时按照 Google 队友的要求，采用已有的开源代码，或者过早的写了测试，别说无法在三个月的实习时间之内完成这个东西，就算折腾好几年也没有可能。

Shape Security

这种思维方式最近的成功实例，是给 Shape Security 做的一个先进的 JavaScript 混淆器（obfuscator）和对集群（cluster）管理系统的改进。不要小看了这个 JS 混淆器，它的混淆能力要比 uglify 之类的开源工具强很多，也快很多。它不但包含了 uglify 的变量换名等基本功能，而且含有专门针对人类和编译器的复杂化，使得没人能看出一点线索这个程序到底要干什么，让最先进的 JS 编译器也无法把它简化。

其实这个混淆器也是一种编译器，只不过它把 JavaScript 翻译成不可读的形式。在这个项目中，由于失之毫厘就可以差之千里，我采用了从 Chez Scheme 编译器学过来的，非常严密的测试方法。对每一个编译器的步骤（pass），我都给它设计一些正好可以测到这个步骤的输入代码（比如，具有函数定义的，for循环，try-catch的，等等）。Pass 输出的代码，经过 JavaScript 解释器执行，把结果跟原来程序的执行结果对比。每一个测试程序，经过每一个 pass，输出的中间结果都跟标准结果进行对比，如果错了就表明那个 pass 有问题，出错的小程序会指出大概是哪一个部分出了问题。遵循小巧，不冗余，不重复的原则，我总共只写了40多个非常小的 JavaScript 程序。由于这些测试涵盖了 JavaScript 的所有构造而且几乎不重复，它们能够准确的定位到错误的改动。最后，这个 JS 混淆器能够正确的转换像 AngularJS 那么大的项目，确保语义的正确，让人完全无法读懂，而且能有效地防止被优化器（比如 Closure Compiler）简化掉。

相比之下，过度鼓吹测试和可靠性的人，并没能制造出这么高质量的混淆器。其实在我进入团队之前，里面的两三位高手已经做了一个混淆器，项目延续了好多个月。这片代码一直没能发布给客户用，因为它的换名部件总是会在某些情况下输出错误的代码，修改了好多次仍然会出错。不是100%的正确，这对于程序语言的转换器来说，是不可接受的。换名只是我的混淆器里的一个步骤，它还包含大概十个类似的步骤，可以把代码进行各种转换。

在实现换名器的时候，队友们让我直接拿他们以前写的换名代码过来，把 bug 修好就可以。然而看了代码之后，我发现这代码没法修，因为它采用了错误的思路，缝缝补补也不可能达到100%的正确，而且明显效率低下，所以我决定自己重写一个。由于轻车熟路，我只花了一下午的时间，就完成了一个正确的换名器，它完全符合 JavaScript 的语义，各种奇葩的作用域规则，而且结构非常简单。说白了，这个换名器也是一种解释器。对解释器的深刻理解，让我可以很容易的写出任何语言的换名器。

不幸的是，历史再次重演了 ;) 队友们听说我花一下午重写了一个换名器，非常紧张，咋呼地跟我说：“你知道我们的换名器是花了多少个月的时间做出来的吗？你知道我们写了多少测试来保证它的正确性吗？你现在一下午做出来一个新的，你如何能保证它的正确！” 我不知道他们怎么好意思说出这样的话来，因为事实是，他们花了这么多个月，耗费这么多人力，写了这么多的测试，做出来的换名器却仍然有 bug，没法用。当我把我写的测试和几个大点的 open source 项目（AngularJS, Backbone 等）放进他们的换名器之后，就发现有些地方出问题了，而所有的测试和 open source 项目通过我的换名器，却得到完全正确的代码。另外经过性能测试，我的换名器速度要快四倍的样子。所以就像 Dijkstra 所说：“最优雅的程序往往也是最高效的。”

结束这个项目之后，我换了一个团队（cluster团队），这个团队的人要好很多，低调而且幽默。Shape Security 的产品（Shape Shifter）里面包含一个高可靠（HA）集群管理系统，它可以通过网络，选举 leader，构建一个高容错的并行处理集群。这个集群管理系统一直以来都是公司里很复杂，却是可靠性要求最高的一个部件，一旦出问题就可能有灾难性的后果。确实，它当时可靠性非常高，从来没出过问题。但由于历史原因，它的代码过度复杂而缺乏模块化，以至于很难扩展来应付新的客户需求。我进入这个新团队的任务，就是对它进行大规模的简化，模块化和扩展，让它满足新的需求。

在这个项目中，由于代码的改动幅度很大，在同事和部门领导的理解，信任和支持下，我们决定直接抛弃已有的测试，完全靠严格而及时的 code review，逻辑推理，推敲讨论，手工试验来保证代码的正确。在我修改代码的同时，一位更熟悉已有代码的队友一直通过 git 默默监视着我的每一次改动，根据他自己的经验来判断我的改动是否偏离了原来的语义，及时与我交流和讨论。由于这种灵活而严格的方式，工程不到两个月就完成了。改进后的代码不但更加模块化，更可扩展，适应了新的需求，而且仍然非常可靠。假设部门领导是“测试教条主义者”，不允许抛弃已有的测试，这样的项目是绝对不可能如期完成的。然而在当今世界遇到这样领导的机会，恐怕十个人里面不到一个吧。

Coverity

最后，我举一个由于测试方式不当而非常失败的案例，那就是 Coverity 的 Java 静态分析产品。我承认 Coverity 的 C 和 C++ 分析器也许是非常好的，然而 Java 的分析器，很难说。当我进入 Coverity 的时候，同事们已经忍受了整整一年的管理层的威逼和高压，超时过劳工作，写出了基本的新产品和很多的测试。可是由于技术债太多，再多的测试也没能保证产品的可靠性。

我的任务就是利用我深入的 PL 知识，不停的修补前人留下来的各种蹊跷 bug。有些 bug 需要运行20多分钟之后才出现，一次还看不出是怎么回事，所以修起来非常耗时。有时候我只好趴在电脑前面养神，时不时的睁眼看看结果。Coverity 是如此的在乎测试，他们要求每修复一个 bug 你就必须写出新的测试。测试必须能够如实的重现 bug 的现象，修复之后测试必须能够通过。这看似一个很在乎代码质量的做法，然而它不但没能保证产品的稳定可靠，而且大幅度的减慢了工程进度，并且造成员工的疲惫和不满。

有一次他们分配给我一个 bug：在分析一个中型项目的时候，分析器似乎进入了死循环，好几个小时都不能完成。因为 Coverity 的全局静态分析，其实就是某种图遍历算法。当这个图里面有回路的时候，你就必须小心，如果不问青红皂白就递归进去，就可能进入死循环。避免死循环的办法很简单，你构造一个图节点的集合（Set），然后把它传递到函数里面作为参数。 每当访问一个节点，你先检查这个节点是否已经在这个集合里，如果在你就直接返回，否则你就把这个节点加入到集合里，然后递归处理这个节点的子节点。它的 C++ 代码大概就像这个样子：

```
void traverse(Node node, Set<Node> &visited)
{
  if (visited.contains(node)) {
    return;
  } else {
    visited.add(node);
```

```
    process_node(node, visited);   // 里面会递归调用 traverse
  }
}
```

查看代码之后我发现，代码其实没有进入“死循环”，而是进入了指数复杂度的计算，所以很久都不能完成。这是因为写这函数的人不小心，或者没有理解 C++ 的函数参数缺省是传值（做拷贝）而不是传引用，所以他忘了打那个“&”，所以函数被递归调用的时候不是传递原来的集合，而是做了一个拷贝。每一次递归调用`traverse`，visited 都得到一个新的拷贝，所以返回之后，visited 的值就恢复到之前的状态，就像 node 被自动 remove 了一样。所以这个函数仍然会在某种情况下再次访问这个节点。这样的代码不会进入死循环，然而在某种特殊的图结构下，这会造成指数级的时间复杂度（请想一下这是什么样的一种图）。

本来很明显的一个图论算法问题，加一个“&”就修好了，手工试验也发现问题消失了。然而 Coverity 的测试教条主义者们（包括写出这 bug 的那人自己），吵着闹着，严肃命令我必须写出测试，构造出可以造成这种后果的数据结构，确保这个 bug 不会再重新出现。

为一个我根本不会犯的错误写测试，而且它不可能再次发生，这不是很搞笑吗？就算你写了测试，也不能保证同样的事情不再发生。如果你不小心漏掉“&”，下次同样的问题还会发生，并且发生在另外的地方，而你却没有给那块代码写测试，所以给这个 bug 写测试，并不能防止同样的问题再次发生。这就像一个技术不过关的赛车手，他在别人不大可能撞车的地方撞了车，然后就要求赛场在那个地方装上轮胎护栏。可是下一次，这个车手又会在另一个其他人都不会撞车地方撞车……

稍微有点图论常识，熟悉 C++ 基本概念的人，都不会犯这种错误。防止这种问题，只有靠个人的技术和经验，而不能靠测试。防止它再次发生的最好办法，恐怕是开个会把这个问题讲清楚，让大家理解，下次不要再犯。所以给这个 bug 写测试，完全是多此一举。跟队友们讲解了这个原理，他们听了之后，仿佛什么都没有听到一样，仍然强硬的要求：“可是你还是得写这个测试，因为这是我们的规定！你知道要是出了 bug，送一个销售工程师去客户那里，要花多少钱吗……” 无语了。

Coverity 的 Java 分析，就是经常因为这种测试教条主义，使得项目进展及其痛苦和缓慢，却仍然 bug 百出。Coverity 的其他的问题，还包括我上面指出的，写重复的测试，一个测试测太多东西，使用字符串比较来做测试，等等。你恐怕很难想象，一个制造旨在提高代码质量的产品的公司，自己代码的质量是这样维护的 :P

完

由于绝大多数人对测试的误解如此之深，测试教条主义的流毒如此之广，导致许许多多优秀的程序员沉沦在繁琐的测试驱动开发中，无法舒展自己的长处。为了大家有一个轻松，顺利又可靠的工作环境，我希望大家多多转发这篇文章，改变这个行业的陋习。我希望大家在工程中理性的对待测试，而不是盲目的写测试，只有这样才能更好更快的完成项目。

（由于这篇文章包含了我很多年的经验和深入的见解，希望你觉得有收获的话为此付费。建议价格是5美元，或者30人民币。【付费方式】）