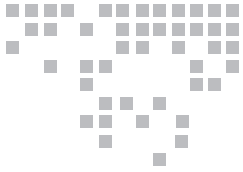

第 2 章 Chapter 2

寻求优秀

本章内容包括：

- 测试怎样才算“优秀”
- 测试相关的行为
- 可靠测试的重要性

我们正在学习优秀的测试。我们想要学习如何识别优秀的测试，书写优秀的测试，改进不那么优秀的测试，这样它们就能成为优秀的测试，或至少接近优秀。问题是，怎么才能算“优秀”？有哪些神奇的要素？以下几个方面要考虑，包括：

- 测试代码的可读性和可维护性
- 代码在项目中及特定源代码中的组织方式
- 测试所检查的内容
- 测试的可靠性及可重复性
- 测试对测试替身的使用

本章将仔细研究这些方面。

上述列表还不够全面。影响测试质量的因素是无穷尽的。同样，一些因素并非在各种情况下都起作用。对一些测试来说，执行速度可能是至关重要的，但对另一些来说，极度专注才是关键。

此外，测试代码的质量取决于观察者的眼睛。如同代码一样，个人偏好关乎“优秀”的定义——我不会忽略偏见的存在。我也不会在本书中假装我能避免自己的偏见和喜好。尽管

我会尽量避免因人而异的问题，但你仍会发现很多章节清晰地凸显了我的个人观点。我觉得没关系。毕竟，我从各位软件牛人那里学到了有关代码，特别是测试代码的内容，形成了基于个人经验的诚恳（和固执己见）看法，这是我能提供的最好的东西。

免责声明之后，我们来讨论一下测试质量的几个方面，看看哪些与我们的兴趣相关。

2.1 可读的代码才是可维护的代码

昨天我从咨询工作现场回到办公室，与同事谈起他近期要参加的 1K 大赛。这种比赛是 demo party 的传统节目——demo party 是一种极客聚会，黑客们会带着计算机、睡袋、能量饮料在巨大的舞台上待上整个周末。

从第一届开始，黑客们就互相较劲，在很多人认为过时的硬件上舞弄着疯狂的技巧来制作 3D 动画。

这种动画的一个典型约束是大小。在我同事要准备的比赛中，其名字 1K 意味着代码编译为二进制之后的大小不能超过 1024 字节。

对，你没听错——1024 字节。为了把有用的程序装入这么小的空间，参赛者需要使用各种奇技淫巧。例如，一个使代码更紧凑的常见手段是让多个变量使用相同的名字——因为这样代码压缩得更好一些。太疯狂了。

生成的代码也同样疯狂。当他们将代码压缩到 1024 字节时，源代码已经面目全非了。你几乎认不出是使用了哪种编程语言！它基本上是一个只写（write-only）代码库——一旦开始压缩，你就无法再改变功能，因为你分辨不出要编辑什么，也不知在哪里编辑和如何编辑。

给你一个鲜活的例子体会一下，这是最近 JS 1K 比赛中实际提交的代码，选用的语言是 JavaScript，而它需要装到 1024 字节内：

```
<script>with(document.body.style){margin="0px";overflow="hidden";}
var w=window.innerWidth;var h=window.innerHeight;var ca=document.
getElementById("c");ca.width=w;ca.height=h;var c=ca.getContext("2d");
m=Math;fs=m.sin;fc=m.cos;fm=m.max;setInterval(d,30);function p(x,y,z){
return{x:x,y:y,z:z};}function s(a,z){r=w/10;R=w/3;b=-20*fc(a*5+t);
return p(w/2+(R*fc(a)+r*fs(z+2*t))/z+fc(a)*b,h/2+(R*fs(a))/z+fs(a)*b);
}function q(a,da,z,dz){var v=[s(a,z),s(a+da,z),s(a+da,z+dz),s(a,z+dz)]
;c.beginPath();c.moveTo(v[0].x,v[0].y);for(i in v)c.lineTo(v[i].x,v[i]
.y);c.fill();}var Z=-0.20;var t=0;function d(){t+=1/30.0;c.fillStyle=
"#000";c.fillRect(0,0,w,h);c.fillStyle="#f00";var n=30;var a=0;var da=
2*Math.PI/n;var dz=0.25;for(var z=Z+8;z>Z;z-=dz){for(var i=0;i<n;i++){
fog=1/(fm((z+0.7)-3,1));if(z<=2){fog=fm(0,z/2*z/2);}var k=(205*(fog*
Math.abs(fs(i/n*2*3.14+t))))>>0;k*=(0.55+0.45*fc((i/n+0.25)*Math.PI*5)
);k=k>>0;c.fillStyle="rgb("+k+","+k+","+k+")";q(a,da,z,dz);if(i%3==0){
c.fillStyle="#000";q(a,da/10,z,dz);}a+=da;}}Z-=0.05;if(Z<=dz)Z+=dz;}
</script>
```

当然，这种情形比一般软件公司中的极端情况还要高几个量级。但我们都在工作中见过让人头大的代码。有时我们称这种代码为**遗留**代码，因为那是从别人那里继承下来并接手维护的——只是它太难维护了，每次试图去理解它都令人头疼。维护这种不可读的代码是一个苦差事，因为我们花了这么大精力去理解我们看到的代码。不仅如此。研究表明，较差的可读性与缺陷密度密切相关。[⊖]

自动化测试是防止缺陷的有效保护。遗憾的是，自动化测试也是代码，其可读性也很容易变差。难以阅读的代码也就难以测试，导致更难为之编写测试。而且，我们编写的测试还远远达不到优秀的地步，因为我们需要围绕拙劣的结构、难懂的 API 调用及非测试友好的结构来组织代码。

我们建立的代码可读性（几乎令人咆哮）对代码可维护性具有可怕的影响。那么测试代码的可读性又如何呢？有多大差别，或者有差别吗？我们看个难读的测试代码的通俗示例，如代码清单 2.1 所示。

代码清单 2.1 并非复杂代码才缺乏可读性

```
@Test
public void flatten() throws Exception {
    Env e = Env.getInstance();
    Structure k = e.newStructure();
    Structure v = e.newStructure();
    //int n = 10;
    int n = 10000;
    for (int i = 0; i < n; ++i) {
        k.append(e.newFixnum(i));
        v.append(e.newFixnum(i));
    }
    Structure t = (Structure) k.zip(e.getCurrentContext(),
            new IObject[] {v}, Block.NULL_BLOCK);
    v = (Structure) t.flatten(e.getCurrentContext());
    assertNotNull(v);
}
```

这个测试在检查什么？你敢说它很容易理解吗？想象自己是团队里的新人——你要花多久才能明白测试的意图？如果该测试突然失败，你要如何调查代码才能搞清状况？根据我对丑陋代码的感觉，我打赌你立即可以从这个烂测试中识别出一些可以改进的地方——可读性是一个常见的改进方面。

⊖ Raymond P.L. Buse，Westley R. Weimer. “Learning a Metric for Code Readability.” IEEE Transactions on Soft-ware Engineering，09 Nov. 2009. IEEE computer Society Digital Library. IEEE Computer Society，http://doi.ieeecomputersociety.org/10.1109/TSE.2009.70

2.2 结构有助于理解事物

我看过无数的代码库，痛并快乐着，天才的美妙步伐并没有在那些源文件中徜徉。某些文件从未跳转到另外的源文件，因为它的全部内容都在一起——所有代码和逻辑，比如，Web 表单的提交全部都放在一个源文件里面。我曾经愚蠢地试图打开一个超大的源文件而导致文本编辑器崩溃。我还见过一个 Web 应用程序报错，是由于 JSP 文件膨胀得太大，导致生成的字节码违反了 Java 类文件的规范。不仅仅说结构是有用的——缺乏结构更是有害的。

对于这些又臭又长的源代码，基本上没人愿意碰它们。即使最简单的概念变化都难以映射到你面前的源代码上。没有结构可以让你的大脑依靠。无法分而治之——你不得不在脑海里处理整件事情，或者准备好用脑袋撞墙。

如图 2.1 所示，你不是想随便要一个结构来帮助理解。你需要这样的结构——用与你的大脑和心智相匹配的方式来分解事物。盲目地将代码外化为单独的源文件、类或方法，在一定时间内能减少代码的数量，从而降低大脑的负担。但是那并不足以隔离和理解我们感兴趣的程序逻辑。于是你需要一个有意义的结构。

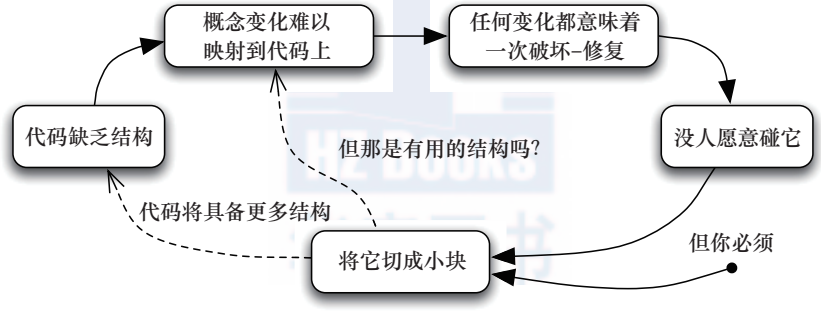

图 2.1 不仅要具备结构——而且要有用的结构

当面对庞然大物，即没完没了的源代码清单时，一个明显的解决方案是将它们切成碎片，将代码块抽取到方法中。可以将一个包含 500 行代码的巨型类分解为 10 个类中的几十个方法，将方法的平均长度降低到 10 行以下。那会向代码中引入更多结构——至少编译器这么认为。这样你也能够在屏幕上看见整个方法，而不用上下滚动。

但是如果分解庞然大物的边界不甚合理——如果它们没能映射到领域和抽象上——我们可能会适得其反，因为现在各个概念之间在物理上可能比之前更加分散，反而增加了你在源文件之间来回切换的时间。很简单。重要的是代码结构是否有助于你快速而可靠地找到高层概念的代码实现所在。

对于这种现象，以测试代码为例是极好的。假设你的应用程序被自动化测试相当好地覆盖

着——但就一个自动化测试。想象一下，这个测试只有一个巨大的测试方法，花了半小时执行应用程序的所有逻辑和行为。假设因为你在程序内部对邮件地址的显示方式做了一点调整，但是却把一些东西搞乱了，因此测试最后失败了，如图 2.2 所示。这是个 bug。接下来会怎样？

图 2.2 缓慢的反馈确实是生产率的杀手

我能想象这要花一段时间才能在测试代码中找到确切的出错位置。测试代码缺乏结构，无助于你理清相互的影响、某个对象是在哪里初始化的、出错时某个变量的值是多少，等等。最终，当你设法找到和修正了错误，你只得再次运行整个测试——整整 30 分钟——来确保你真的修复了问题，并且在这个过程中没有再破坏其他东西。

继续这个思考实验，设想快速地倒退到一个小时前，此时你正要改动另一处。这次你学乖了，你要小心地确保自己理解了当前的实现，保证自己做了正确的改动。那你会怎么做？去阅读代码，特别是精读测试代码，它会具体地告诉你生产代码的预期行为。只是你找不到测试代码的相应部分，因为它缺乏结构。

你需要的是专注的测试，它可读、可达、可理解，这样你才能：

- 找到与手上任务相关的测试类
- 从那些类中识别出合适的测试方法
- 理解测试方法中对象的生命周期

关注测试的结构并确保它有用，你就可以做到这几点。当然，具备有用的结构还不够。

2.3 如果测试了错误的东西就不好了

在阅读和调试代码以找出不良系统行为的原因时，我却最终不止一次地回到了开始的地方。在找 bug 的过程中，尤其容易忽略的一个烦人细节是测试的内容。在挖掘代码时，我要做的头一件事情往往是运行所有测试，让它告诉我哪些正常，哪些不正常。有时我太过相信测试的名称。有时那些测试其实完全是在测试不同的东西。

这与良好的结构有关——如果测试的名字错误地表达了要测试的内容，那就像是跟着错误的路标驾驶。你应该能够信任你的测试。

几年前我为某个产品审计代码，该代码开发了已经超过十年。那是个巨大的代码库，我从结构中可以分辨出某些部分明显比其他部分要新。区分新旧代码的一个线索是自动化测试

的存在。但我很快发现，我无法从测试的名字来分辨出要验证的内容，再仔细看，发现测试根本没有在验证它承诺的内容。它不是 Java 代码，但我把它翻译成了 Java 的例子：

```
public class TestBmap {
    @Test
    public void mask() {
        Bmap bmap = new Bmap();
        bmap.addParameter(IPSEC_CERT_NAME);
        bmap.addParameter(IPSEC_ACTION_START_DAYS, 0);
        bmap.addParameter(IPSEC_ACTION_START_HOURS, 23);
        assertTrue(bmap.validate());
    }
}
```

看到这段代码你立刻能注意到测试的命名不够理想。但再仔细看看，无论“mask”对于“Bmap”意味着什么，测试也仅仅是检查了某些参数是否为有效的组合。如果输入正确的情况下实际行为却仍然有误，那么参数能否通过验证也就变得无关紧要了。

关于测试正确的事物这件事其实有很多话要说，但用正确的方式测试正确的事物也很关键。从可维护性角度尤其重要的是，你的测试应该检查预期行为而非具体实现。下一章会涉及这个话题，现在先按下不表。

2.4 独立的测试易于单独运行

关于测试有很多话要说，哪些该包含，哪些不该包含，哪些该指定，哪些不该指定，如何从可读性角度来组织，等等。对测试外围的考虑有时也起了至关重要的作用。

人类——我们的大脑过于精确——是极其强大的信息处理器。我们几乎可以瞬间评估身体周围的环境并在眨眼间做出反应。我们能在意识到雪球飞过来之前就做出闪避。这些反应根植于我们的 DNA 中。

当我们感知到相似模式时，行为图谱就会指示身体移动。随着时间的推移，这些食谱慢慢变得成熟，我们很快就背负上了一个互联模式和行为的复杂网络。

这种情况也发生在工作中。首次探索别人的代码库时，我们会在 15 分钟内形成对常见惯例、模式、**代码坏味道**和陷阱的清晰图景。这是我们识别相似模式的能力在起作用，并且能够告诉我们可能还会在附近看到哪些其他东西。

代码坏味道是什么？

代码中的坏味道提示我们代码中某些地方可能出问题了。引用 Portland Pattern Repository 的 Wiki，“如果某些东西闻起来发臭了，那绝对需要检查它一下，但是不见得真的需要修复它，或者只能继续忍受。”

例如，当我接触新的代码库时，我注意到的第一件事情就是方法的大小。如果方法过大，我立马明白在那些特定模块、组件或源文件中还有一大堆问题等着我呢。我关注的另一个信号是变量、类和方法名字的描述性如何。

具体说到测试代码，我关注测试的独立水平，尤其是架构边界附近。这样做是因为我在边界上仔细发现了许多代码坏味道，于是我学会了一看到外部依赖时就特别小心，包括：

- 时间
- 随机数
- 并发性
- 基础设施
- 现存数据
- 持久化
- 网络

这些事物的共同之处在于它们往往都很复杂，对于一个项目的测试基础设施（infrastructure）来说，我认为最基本的试金石（litmus test）就是：我能否从版本控制中签出全新的代码，复制到刚刚打开包装的新计算机上，运行一条命令，然后翘起二郎腿，看着整套自动化测试运行并且通过？

隔离和独立很重要，因为没有它们就难以运行和维护测试。开发者为了运行测试而不得不对系统做的每件事都会使事情变得更加繁琐。

无论你是否需要在文件系统中特定位置创建空目录，或确保你具备一个特定版本的MySQL运行在特定端口号上，或添加一条用于测试用户登录的数据库用户记录，或设定一堆环境变量——这些都不是开发者该做的。这些小事增加了工作量并会累积为奇怪的测试失败。[㊀]

例如测试执行时的系统时钟或随机数生成器的下一个值，这些都不在你的控制之中，而这正是此类依赖的特征。作为经验法则，你想要避免由于这种依赖而导致测试古怪地失败。你希望将代码放进一个台钳，通过传入测试替身或者将代码与环境隔离，使其行为符合你的需要，从而控制一切。

在测试类中不要依赖于测试的顺序

一般来说，不让测试互相依赖是指你不该让一个类中的测试依赖另一个类中测试的执行或结果。但这也同样适用于同一个测试类中的依赖。

㊀ 如果你无法避免这些手工配置，那么至少确保开发者只需要做一次。

该错误的典型例子是这样的，当程序员在 @BeforeClass 方法中设置系统的起始状态后，写下了三个连贯的 @Test 方法，每个都在修改系统状态，并相信上一个测试完成了一部分工作。现在，当第一个测试失败时，后面所有测试都会失败，但那还不是最大的问题——至少提示你发现了错误，对吗？

真正的问题是当其中某些测试因为错误的原因而失败。例如，假设测试框架决定以不同的顺序来调用测试方法。虚惊一场。JVM 供应商决定改变反射 API 返回方法的顺序。一场虚惊。测试框架作者决定以字母顺序来运行测试。又是虚惊一场。[⊖]

你不喜欢总是一惊一乍的。当测试要检查的行为正常时，你并不希望你的测试失败。因此，你不该故意让测试执行相互依赖而造成它们很脆弱。

测试意外失败的最不寻常的例子之一，是一个测试作为套件的一部分时可以通过，但单独运行却神秘地失败（反之亦然）。

那些症状散发着测试相互依赖的臭气。它们假设另一个测试在自己之前运行，而且那个测试会将系统置于某个特定状态。当假设不成立时，你就硬着头皮去调试吧。

总而言之，当编写的测试涉及时间、随机数、并发性、基础设施、持久化或网络时，你就应该格外小心。作为经验来说，你应该尽量避免依赖它们，将它们限制到小的隔离单元中，这样你的大部分测试就不会遭受并发症，也不用总是挨个处理它们——只有少数几个地方才用得着操心。

那么在实践中看起来如何呢？你到底该做什么？例如，你可以看看你能否找到一个方式来做下面这些事：

- 用测试替身替换对第三方库的依赖，根据需要将其包装到你自己的适配层中。将各种麻烦封装进适配层以后，你就可以独立地测试其余的程序逻辑。
- 将测试代码与其用到的资源放在一起，或许是在一个包（package）里。
- 让测试代码自己产生所需资源，而不要让它们与源代码分开。
- 令测试自行建立所需的上下文。不要依赖于之前运行的任何测试。
- 对于需要持久化的集成测试，那就使用内存数据库吧，用了干净的数据集，就能极大地简化测试的启动问题。还有，它们通常启动得超级快。
- 将线程代码分为同步和异步两部分，所有程序逻辑都放在一个常规的同步代码单元中，就可以方便地进行测试并且没有并发症，将棘手的并发部分留给一小堆专用测试。

⊖ 如果这听起来有点不可思议，你应该知道 JUnit 并不保证按照特定顺序运行测试方法。事实上，当 Java 7 改变了 Class#getDeclaredMethods() 返回的方法声明顺序，NetBeans 项目中的几个测试就失败了。哦，我猜它们的测试不够独立……

当面对遗留代码时要做到测试隔离是很难的，那些代码在设计时并未考虑可测试性，因此不具备你想要的模块化。但即使这样，仍然值得去打破那些讨厌的依赖从而使你的测试与环境隔离并相互独立。你的测试毕竟得靠得住才行。

2.5 可靠的测试才是可靠的

前一节中我说过，有时候测试的内容与你想象的完全不同。更让人操心的是，有时它们根本什么都没测试。

我的一个同事习惯于称这种测试为**快乐的测试**，指某个测试快乐地执行一段生产代码——或许是全部的执行路径——却没有一句断言。是的，你的测试覆盖率报告看起来很棒，因为测试全面地执行了你写的每句话。问题是这种测试只有在生产代码抛出异常时才会失败。

你无法依靠这种测试来保护自己，对吗？特别是如果程序员惯于将所有测试方法体封装到 try-catch 块中的时候。[⊖]代码清单 2.2 展示了这种坏习惯。

代码清单 2.2 你能指出这个测试的缺陷吗？

```
@Test
public void shouldRefuseNegativeEntries() {
    int total = record.total();
    try {
        record.add(-1);
    } catch (IllegalArgumentException expected) {
        assertEquals(total, record.total());
    }
}
```

某些测试相对来说不太容易失败，代码清单 2.2 是一个典型的极端例子，其中的测试或许永远不会失败（过去也没有过）。如果你仔细观察，你会注意到即使 add（–1）没有如期地抛出异常，测试也不会失败。

几乎不会失败的测试就等于废物。也就是说，间歇性地通过或失败的测试就是在公然地侵害程序员小伙伴们，见图 2.3。

图 2.3 如果测试从不失败或一直失败，那它们就没价值

⊖ 真实的故事。我花了 1 个小时来去掉它们，然后当天剩下的时间里都在修复或删除这些我发掘出的失败测试。

几年前，我为某个项目做咨询，花了大部分时间与客户的技术人员及其他顾问做结对编程。一天早上我和我的搭档接到一个新任务，然后像往常一样先运行一下相关的测试集。我的搭档对代码库相当熟悉，编写了其中大部分代码，也熟悉其中的各种怪异之处。在我们做出任何修改之前，我注意到一些测试在第一次运行时失败了。让我惊讶的是，我的搭档如此来对待失败的测试——他不断地一遍遍重复运行测试，直到四五次以后所有测试至少都通过了一次。我不是 100% 确定，但我不认为所有测试都是在同一次运行中通过的。

我目瞪口呆，我意识到我见到的一堆测试其实全都是不可靠的测试。某些测试会随机地失败，因为被测代码包含了不确定的逻辑，于是测试有 50% 的机会会失败。除了在被测代码中使用了伪随机数生成器之外，这种间歇性行为的另一个常见原因是使用了时间相关的 API。我最喜欢调用 System.currentTimeMillis()，紧随其后的就是在测试异步逻辑时无处不在的 Thread.sleep（1000）。

为了让测试值得依靠，它们就需要可重复。如果运行两遍测试，它就必须给我相同的结果。否则，我就不得不在每次构建之后采取人工干预，因为无法知道 1250/2492 是意味着一切正常，还是说最后一遍时全都挂了。无法知道。

如果你的逻辑包含异步内容或依赖于当前时间，确保将它们隔离在一个接口之后，这样你可以用“测试替身”来替换它们从而使测试可重复——这是测试变得可靠的一个关键要素。

2.6 每个行业都有其工具而测试也不例外

我说的测试替身是什么？如果你的程序员工具箱中没有测试替身，你就错过了测试的许多功能。**测试替身**是程序员熟知的 stub（桩）、fake（伪造对象）、mock（模拟对象）的总称。它们本质上是为了测试目的、用于替换真实协作者的对象，如图 2.4 所示。

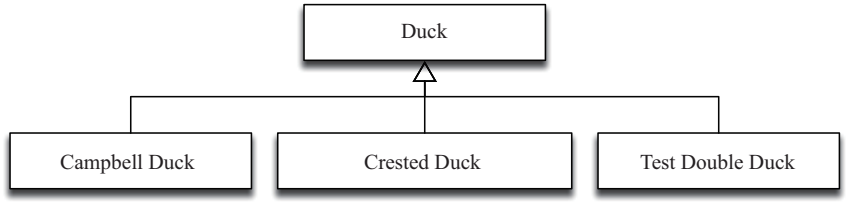

图 2.4 鸭子的测试替身，看起来像鸭子，叫起来几乎也像只鸭子——但当然不是真鸭子

你可以说测试替身是测试感染的程序员的最佳伙伴。因为它们促进了许多改进并为我们提供许多新工具，如：

- 通过简化要执行的代码来加速执行测试

- 模拟难以出现的异常情况
- 观察那些对测试代码不可见的状态和交互

关于测试替身还有很多要说的，下一章将详细讨论这一话题。但测试替身并不是行业中编写自动化测试的仅有工具。

行业中最基本的工具或许就是测试框架了，比如 JUnit。我仍然记得最初如愿以偿地让代码工作起来的时光。每当程序出错卡住了，我就会取消好几条语句用于向控制台输出，然后重启程序，这样我就能通过分析控制台输出来找出失败的位置和原因。

职业生涯最初几个月，我见到商用软件开发者也在用同样的方式工作。与利用 JUnit 之类的工具编写自动化、可重复的测试相比，我相信我不用指出那样做有多浪费和多不专业。

除了合适的测试框架和测试替身之外，在编写自动化测试的前三样工具中还包括另一种——构建工具。无论你的构建过程是怎样的，构建脚本中用到哪种工具或技术，都没理由不将自动化测试集成到构建中。

2.7 小结

本章为优秀测试粗略地定义了几个特征。

我们指出，这些特征都是依赖于上下文的，没有绝对的真理能使得测试变得“优秀”。自动化测试有多优秀取决于它有多符合目标，对此我们识别出一些具有重大影响的普遍问题。

我们首先指出测试的一个主要优点是可读性，因为如果难以阅读和理解，测试就会带来维护问题，其实要解决这个问题也很快——删掉它，因为维护起来成本太高。

接下来我们指出，测试代码的结构有助于使之更好用，允许程序员快速定位到正确的位置，有助于程序员理解发生了什么——与可读性一脉相承。

接下来我们阐明，测试有时候是在测试错误的事物，它将你带入歧途或浑水之中而造成问题，这样反而隐藏了测试的真实逻辑，使测试难以阅读。

关于测试有时候不可靠的话题，我们还为此识别了一些常见原因，以及可重复测试的重要性。

最后，我们认为在行业中编写自动化测试的三个基本工具是——用于编写测试的测试框架、用来运行测试的自动化构建和改善测试及可测试性的测试替身。第三个话题如此重要，以至于我们将在下一章专门讨论如何使用测试替身来编写优秀的测试。