漂亮的测试

——《代码之美》第七章

——By Alberto Savoia

——史际帆

作者简介：Alberto Savoia

Alberto Savoia is co-founder and CTO of Agitar Software. Before Agitar, he was Senior Director of Engineering at Google; prior to that he was the Director of Software Research at Sun Microsystems Laboratories.

Alberto's passion and main body of work and research is in the area of software development technology—in particular, tools and technology to help programmers test and verify their own code during the design and development phase.

Alberto Savoia 是Agitar软件公司创始人之一和CEO。在创建Agitar之前，它是Google的高级工程主管；在这之前，他还是Sun Microsystems实验室软件研究中心的主管。其主要研究领域是软件开发技术——尤其是那些帮助程序员在设计和开发阶段尽心测试和代码验证的工具和技术。

- 何以谓“漂亮的代码”？

  - ✓ 实现所要求的功能
  - ✓ 优雅、简洁

- “漂亮的测试”呢？

测试是对于代码正确与否的检验，当然也应该漂亮。而测试天生带有组合性与试探性，单一测试的漂亮并不一定组合起来还漂亮。

例如：代码中的每一条if语句至少需要两个测试：一个测试条件表达式为真，另一个测试其为假的情况。

而 if (a || b || c) 则理论上需要八个测试。再考虑到循环中的异常，多个输入参数，对外部代码的依赖，不同的软硬件平台等，所需测试的数量和类型将大大增加。

- 所以除最简单外，任何代码都需要一组测试而不是一个。其中每一个测试专注于检查代码的一个特定方面。就像球队中不同球员有不同职责，负责不同区域一样。

测试一般是逐步建立的，并逐步来验证所测试代码的正确与高效性。

- 测试的漂亮：

➢简单、迅速

➢揭示出代码的本质

找出逻辑错误，发现结构和设计上的问题，使代码更健壮，更有可读性。

➢测试具有深度与广度

深入彻底、覆盖无遗，增强开发者信心。

■讨论漂亮的测试，必先寻找足够生动有趣，常出bug而本身简单漂亮的代码。

■作者找到了Jon Bentley 《Programming Pearls》书中的二分法查找算法。

二分查找

Here is a Java implementation with the infamous bug:

```
public static int buggyBinarySearch(int[] a, int target) {
        int low = 0;
        int high = a.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int midVal = a[mid];
            if (midVal < target)
                low = mid + 1;
            else if (midVal > target)
                high = mid - 1;
            else
                return mid;
        return -1;
    }
```

Bug: 如果low和high的和大于Integer.MAX_VALUE(Java中是2^{31}-1)，计算将会溢出使之成为一个负数。

包含target则返回其位置（可以提到外面）

不含target则返回-1

Jon Bentley 向学生们讲述此算法：
在一个包含t的数组中，通过将范围中间的元素与t进行比较并丢弃一半的范围，一直持续直到t被发现或范围为空。

但他又记述了怎样在多年时间中让上百个专业程序员来实现，每次给他们2小时并随意选择高级语言，但令人惊讶的是大约只有10%的专业程序员正确实现了二分查找。

Donald Knuth 在《Sorting and Searching》中指出： even though the first binary search was published in 1946, it took 12 more years for the first binary search *without bugs to be published.*

修改方法：

1.使用减法

```
int mid = low + ((high - low) / 2);
```

2.无符号位移运算

```
int mid = (low + high) >>> 1;
```

二分查找

- 那是一个在好几十年时间里，连一些最好的程序员都抓不到的Bug，它教导我们：哪怕是最简单的一段代码，要写正确也并非易事，更别提我们现实世界中的系统——他们跑在大段大段的复杂代码之上。

- 但是这段代码在修改之后还是有问题，或许不是Bug，但还有可以而且应该被修改的地方。这些修改可以使代码不仅更加健壮，而且可读性，可维护性，可测试性都会比原来好。

Java 程序测试的框架：JUnit

◆**Kent Beck** 和 **Erich Gamma** 设计
◆简单而宏伟的目标：使程序开发者更容易去做他们本来就应该做的事情——测试自己的代码。

- Never in the field of software development was so much owed by so many to so few lines of code.

软件开发领域中从此之前从未有过这样的事情：
很少的几行代码对大量代码起了重要的作用。

——Martin Fowler

Java 程序测试的框架：JUnit

- 框架：框架是一个应用程序的半成品。框架提供了可以在应用程序之外共享可复用的公共结构。开发者把框架融入他们自己的应用程序，并加以扩展，以满足他们特定的需要。框架和工具包的不同之处在于，框架提供了一致的结构，而不仅仅是一组工具类。

Java 程序测试的框架：JUnit

- 单元测试：单元测试测的是独立的一个工作单元。在Java应用程序中，“独立的一个工作单元”常常指的是一个方法（但不总是如此）。作为对比，集成测试和接收测试则检查多个组件如何交互。一个工作单元是一项任务，它不依赖于其他任何任务的完成。

① 单元测试可以降低不确定性从而降低风险
② 单元测试可以帮助优化设计
③ 单元测试用例可以当文档使用

Java 程序测试的框架：JUnit

- 所有的代码都要通过测试
- 调试代码可能比写代码用的时间还长

- 手工测试：增加记录、查看记录、编辑记录、删除记录。一遍一遍的做测试直到通过，还有可能出错。
- 如果不喜欢重复工作，需要自动测试。

Java 程序测试的框架：JUnit

- 本章中使用的方法：

1. 当你要测试一样东西时，为一个方法加上@org.junit.Test；
2. 当你要检查一个值时，输入org.junit.Assert.*，然后调用assertTrue( )，并传递一个Boolean 对象，当测试成功时它为true。

例如：测试两个Money对象相加

```
@Test
public void simpleAdd( ) {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

JUnit的核心

在《JUnit in action》中使用的是JUnit3.8.1，它的写法：

```
import junit.frame.TestCase;

public class TestCalculator extends TestCase
{
   public void testAdd()
   {
         Calculator calculator = new Calculator();
         double result = calculator.add(10, 50);
         assertEquals(60, result, 0);
   }
}
```

其中assertEquals为：

```
static public void assertEquals(double expected, double actual, double delta)
```

JUnit的核心

- 七个核心类及接口：

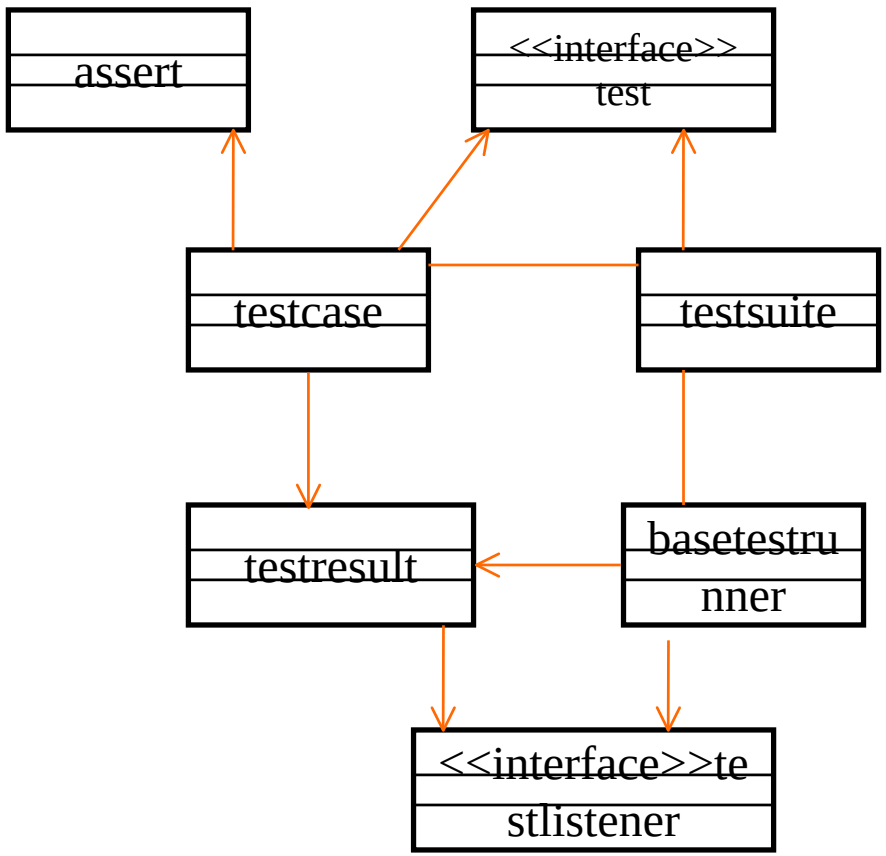
类/接口	作用
Assert	但条件成立时assert方法保持沉默，但若条件不成立就抛出异常
TestResult	TestResult包含了测试中发生的所有错误或者失败
Test	可以运行Test并把结果传递给TestResult
TestListener	测试中若产生事件（开始、结束、错误、失败）会通知TestListener
TestCase	TestCase定义了可以运行与多项测试的环境（或者说是固定设备）
TestSuite	TestSuite运行一组TestCase（他们可能包含其他TestSuite），它是Test的组合
BaseTestRunner	TestRunner是用来启动测试的用户界面，BaseTestRunner是所有TestRunner的超类

JUnit的核心： Assert超类型

方法	描述
AssertTrue	断言条件为真，否则抛出AssertionFailedError
AssertFalse	断言条件为假，否则抛出AssertionFailedError
AssertEquals	断言两对象相等，否则抛出AssertionFailedError
AssertNotNull	断言对象不为null，否则抛出AssertionFailedError
AssertNull	断言对象为null，否则抛出AssertionFailedError
AssertSame	断言两引用指向同一对象，否则抛出AssertionFailedError
AssertNotSame	断言断言两引用指向不同对象，否则抛出AssertionFailedError
fail	让测试失败并给出信息

JUnit的核心

- 而TestCase 有10个基本方法……
- 整体较复杂



测试二分查找算法的步骤：

- 冒烟测试；
- 边界测试；
- 彻底、全覆盖类型的随机测试
- 性能测试

➢关于**冒烟测试**，应该是微软首先提出来的一个概念

➢**冒烟测试**就是在每日build建立后，对系统的基本功能进行简单的测试。这种测试强调功能的覆盖率，而不对功能的正确性进行验证。

➢至于**冒烟测试**这个名称的来历，大概是从电路板测试得来的。因为当电路板做好以后，首先会加电测试，如果板子没有冒烟再进行其它测试，否则就必须重新来过。

➢ **冒烟测试**的说法据说是：象生产汽车一样，汽车生产出来以后，首先发动汽车，看汽车能否冒烟，如果能，证明汽车最起码可以开动了。说明完成了最基本的功能。

第一道防线：冒烟测试

基本、给人很大信心

```
import static org.junit.Assert.*;
import org.junit.Test;
public class BinarySearchSmokeTest {

@Test
public void smokeTestsForBinarySearch( ) {
        int[] arrayWith42 = new int[] { 1, 4, 42, 55, 67, 87, 100, 245 };
        assertEquals(2, Util.binarySearch(arrayWith42, 42));
        assertEquals(-1, Util.binarySearch(arrayWith42, 43));
        }
}
```

一般将许多（几千个）冒烟测试组成一个测试组，每一次构建时运行。将每一个测试足够小，来保持其速度快。

而在编写代码之前就编写冒烟测试，就叫做“测试驱动开发（TDD）”

第二道防线：边界测试

探测和验证代码在处理极端或偏门情况时会发生什么。
首先测试数组长度边界（空数组、长度为1的数组、非常大的数组）

```
int[] testArray;

@Test
public void searchEmptyArray( ) {
        testArray = new int[] {};
        assertEquals(-1, Util.binarySearch(testArray, 42));
}

@Test
public void searchArrayOfSizeOne( ) {
        testArray = new int[] { 42 };
        assertEquals(0, Util.binarySearch(testArray, 42));
        assertEquals(-1, Util.binarySearch(testArray, 43));
}
```

足够大的数组如何测试？ 创建足够大的数组？ 只需测试中间值的计算——不是溢出一个负数即可。具体思路：

1.无法使用足够大数组直接测试；

2.可以编写足够测试，确信BinarySearch在小数组上执行正确；

3.当用到相当大的数值时，可单独测试计算中间值的方法，这与数组无关；

4.于是只需保证：计算中间值没问题时，BinarySearch刻正确实现；并且计算中间值过程没有问题。

将易于溢出的计算隔离出来单独测试，编写函数：

```
static int calculateMidpoint(int low, int high) {
        return (low + high) >>> 1;
}
```

然后将代码中的

```
int mid = (low + high) >>> 1;
```

变成

```
int mid = calculateMidpoint(low, high);
```

然后不断测试calculateMidpoint来保证其正确进行。

如此改动：
1.运用了内联的方法，没有损失性能；
2.提高了代码的可读性；
3.提高了代码的可测试性。

这段代码通过了测试，
则可以确定“位运算符”
可以正确实现其功能！

这样构造出了新的测试代码：

```
@Test
public void calculateMidpointWithBoundaryValues( )
    assertEquals(0, calculateMidpoint (0, 1));
    assertEquals(1, calculateMidpoint (0, 2));
    assertEquals(1200000000, calculateMidpoint (1100000000, 1300000000));
    assertEquals(Integer.MAX_VALUE - 2,
            calculateMidpoint (Integer.MAX_VALUE-2, Integer.MAX_VALUE-1));
    assertEquals(Integer.MAX_VALUE - 1,
            calculateMidpoint (Integer.MAX_VALUE-1, Integer.MAX_VALUE));
}
```

其次，测试关于目标元素位置的边值：数组中的第一项、最后一项和正当中一项。

```
@Test
public void testBoundaryCasesForItemLocation( ) {
        testArray = new int[] { -324, -3, -1, 0, 42, 99, 101 };
        assertEquals(0, Util.binarySearch(testArray, -324)); // first position
        assertEquals(3, Util.binarySearch(testArray, 0)); // middle position
        assertEquals(6, Util.binarySearch(testArray, 101)); // last position
}
```

运用一些0与负数，最大和最小数值来测试：

```
public void testForMinAndMaxInteger( ) {
        testArray = new int[] {
        Integer.MIN_VALUE, -324, -3, -1, 0, 42, 99, 101,
Integer.MAX_VALUE
        };
        assertEquals(0, Util.binarySearch(testArray, Integer.MIN_VALUE));
        assertEquals(8, Util.binarySearch(testArray, Integer.MAX_VALUE));
}
```

上述所有的测试都用过了，那么代码就一定能在任何场景下都正常工作吗？

The words of Joshua Bloch echo in my mind: "*…It is hard to write even the smallest piece of code correctly.*"

编写代码时，程序员的测试总是忽略一些应用场景，所以要创新必须跳出旧的思维模式。如此，我们需要覆盖更广的测试，而不仅是一些具体例子。

第三道防线：随机测试

上述具体测试的例子仅仅为所有可能输入的一个很小的子集，为达到全面测试，我们需要：

1.一种能产生大量具有不同特征的数据量的方法；

2.一组能针对各种输入集合的断言。

运用java.util 包中随机数产生器和Arrays类中的方法，解决第一个需要

```
public int[] generateRandomSortedArray(int maxArraySize, int maxValue) {
    int arraySize = 1 + rand.nextInt(maxArraySize);
    int[] randomArray = new int[arraySize];
    for (int i = 0; i < arraySize; i++) {
        randomArray[i] = rand.nextInt(maxValue);
    }
    Arrays.sort(randomArray);//保证数组有序
    return randomArray;
}
```

第二个需要，指的是对于任何输入数组和查找目标值，assert……必须为真，作者称之为“推理”。

推理测试越充分，即其正确的时候越多，我们对代码的正确性越有信心。

推理1：如果BinarySearch(testArray , int target)返回-1，则testArray不含target；
推理2：如果 BinarySearch(testArray , int target)返回n（>=0），则testArray的第n个位置是target。

测试这两个推理：

```
public class BinarySearchTestTheories {
Random rand;

@Before
public void initialize( ) {
rand = new Random( );
}

@Test
public void testTheories( ) {
        int maxArraySize = 1000;
        int maxValue = 1000;
        int experiments = 1000;
        int[] testArray;
        int target;
        int returnValue;
        while (experiments-- > 0) {
                testArray = generateRandomSortedArray(maxArraySize,
maxValue);
```

```
                    if (rand.nextBoolean( )) {
                        target = testArray[rand.nextInt(testArray.length)];
                    } else {
                        target = rand.nextInt( );
                    }
        returnValue = Util.binarySearch(testArray, target);
        assertTheory1(testArray, target, returnValue);
        assertTheory2(testArray, target, returnValue);
        }
}
public void assertTheory1(int[] testArray, int target, int returnValue) {
        if (returnValue == -1)
            assertFalse(arrayContainsTarget(testArray, target));
}
public void assertTheory2(int[] testArray, int target, int returnValue) {
        if (returnValue >= 0)
            assertEquals(target, testArray[returnValue]);
}
public boolean arrayContainsTarget(int[] testArray, int target) {
        for (int i = 0; i < testArray.length; i++)
            if (testArray[i] == target)
                return true;
        return false;
}
```

先1000组测试，再将maxArraySize设成10000，都通过了测试。可是这就完了吗？

◆这只测试了一半——“如果binarysearch的返回值为有关某事为真，那么testarray和目标有关的这件事为真”，可是“如果testarray和目标有关的这件事为真，那么binarysearch的返回值为有关某事为真”对不对呢？

◆例如： binarysearch的返回值为-1时，target不在数组中，而我们不能肯定当target不再数组中时，binarysearch一定返回-1。同理，当：binarysearch的返回值为n时， target在数组中第n个位置，而我们不能肯定当， target在数组中第n个位置时，binaryseach一定返回n。

◆这样，我们需要Jeff Offut 发明的“变异测试”（mutation text）。其基本思想是修改被测代码使之带有一些bug，如果测试依然正确通过，则说明这些测试或源代码无法满足需要。
下面我们将binarysearch将一定修改：如果target大于424242而且不在数组中，返回-42而不是-1。

```
public static int binarySearch(int[] a, int target) {
        int low = 0;
        int high = a.length - 1;
        while (low <= high) {
                int mid = (low + high) / 2;
                int midVal = a[mid];
                if (midVal < target)
                        low = mid + 1;
                else if (midVal > target)
                        high = mid - 1;
                else
                        return mid;
        }
        if (target <= 424242)
                return -1;
        else
                return -42;
}
```

测试通过了，说明了原来的测试并不全面！

增加推理3与推理4

推理3：如果testArray不包含target，它就必须返回-1。

推理4：如果textArray 在位置n上包含target，那么binarySearch必须返回n。

于是增加测试：

```
public void assertTheory3(int[] testArray, int target, int returnValue) {
    if (!arrayContainsTarget(testArray, target))
            assertEquals(-1, returnValue);
}
public void assertTheory4(int[] testArray, int target, int returnValue)
    assertEquals(getTargetPosition(testArray, target), returnValue);
}
public int getTargetPosition(int[] testArray, int target) {
    for (int i = 0; i < testArray.length; i++)
            if (testArray[i] == target)
                    return i;
    return -1;
}
```

多余的，推理4中已包含推理3

为防止代码重复：

```
arrayContainsTarget(int[ ] testArray, int target) {
        return getTargetPosition(testArray, target) >= 0;
}
```

这回测试中出现了错误！

用以下修改的程序来使JUint在测试失败时给出更多的文本信息：

```
public void assertTheory4(int[] testArray, int target, int returnValue) {
    String testDataInfo = "Theory 4 - Array=" +
            printArray(testArray)
            + " target="
            + target;
    assertEquals(testDataInfo, getTargetPosition(testArray, target),
    returnValue);
}
```

再次运行测试程序，出现以下结果：

java.lang.AssertionError: Theory 4 - Array=[2, 11, 36, 66, 104, 108, 108, 108, 122,155, 159, 161, 191] target=108 expected:<5> but was:<6>

我们马上知道错误出在了那里：我们没有把重复值考虑在内！这就是前边所说的可以继续修改binarySearch的地方。

Mistakes are the portals of discovery.

错误是发现之门

——James Joyce

想法：

1.在binarySearch搜到target时，记录下位置，再接着在左边继续二分查找，最后返回最小的target位置。

2.修改getTargetPosition在参数中用两个int*记录target出现最早和最晚的位置，只要binarySearch的返会值在此范围内即可。

性能测试

- 经过我们的检验，此程序很难再有任何bug了，但是，对于一个漂亮的程序，不检验其速度是不能展现其漂亮特性的。

- 所以，我们增加性能测试，来测试二分查找的速度。

- 首先想到的是用系统时钟，可是二分查找的速度实在太快了，系统时钟没有那么精确的时钟。

- 在其中插入一个计数器是个不错的选择。

```
public static int binarySearchComparisonCount(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;
    int comparisonCount = 0;
    while (low <= high) {
            comparisonCount++;
            int mid = (low + high) >>> 1;
            int midVal = a[mid];
            if (midVal < target)
                    low = mid + 1;
            else if (midVal > target)
                    high = mid - 1;
            else
                    return comparisonCount;
    }
    return comparisonCount;
}
```

基于这个代码，我们给出**推理5**：

如果testArray的长度为n，那么binarySearchComparisonCount必须返回一个小于或等于$1+\log_2 n$的数。

最后将这些推理添加到推理列表中：

```
...
while (experiments-- > 0) {
    testArray = generateRandomSortedArray( );
    if (rand.nextInt( ) % 2 == 0) {
            target = testArray[rand.nextInt(testArray.length)];
    } else {
            target = rand.nextInt( );
    }
    returnValue = Util.binarySearch(testArray, target);
    assertTheory1(testArray, target, returnValue);
    assertTheory2(testArray, target, returnValue);
    assertTheory3(testArray, target, returnValue);
    assertTheory4(testArray, target, returnValue);
    assertTheory5(testArray, target);
}
...
```

设定不同的maxArraySize多跑几次，作者设成1,000,000去吃午饭！

推理5“好像”不成立？？？

结论

- 冒烟测试；
- 边界测试；
- 彻底、全覆盖类型的随机测试
- 性能测试

The only time you don’t fail is the last time you try anything ——and it works.

——William Strong

这些测试完成后，我们对这段二分查找代码有了很大的信心！！！几乎可以断言代码的正确性。

结论

- 这一章，展示了测试的代码也可以相当漂亮，测试代码的编写可以像原代码本身一样需要创造力。
- Some tests are beautiful for their simplicity and efficiency.
- Other tests are beautiful because, in the process of writing them, they help you improve the code they are meant to test in subtle but important ways.
- Finally, some tests are beautiful for their breadth and thoroughness.

向艺术家们学习：
画家经常放下手中的画笔，远离画布一段距离，围它转一转，翘着脑袋左右看看，再在不同光线不同角度下看看。
再寻求美的过程中，编写代码也需要不时用挑剔的眼光从不同的角度审视代码，这将使我们成为更优秀的程序员，并写出更漂亮的代码！

谢谢！