28-理论二：为了保证重构不出错，有哪些非常能落地的技术手段？

上一节课中，我们对“为什么要重构、到底重构什么、什么时候重构、该如何重构”，做了概括性介绍，强调了重构的重要性，希望你建立持续重构意识，将重构作为开发的一部分来执行。

据我了解，很多程序员对重构这种做法还是非常认同的，面对项目中的烂代码，也想重构一下，但又担心重构之后出问题，出力不讨好。确实，如果你要重构的代码是别的同事开发的，你不是特别熟悉，在没有任何保障的情况下，重构引入bug的风险还是很大的。

那如何保证重构不出错呢？你需要熟练掌握各种设计原则、思想、模式，还需要对所重构的业务和代码有足够的了解。除了这些个人能力因素之外，最可落地执行、最有效的保证重构不出错的手段应该就是**单元测试**（Unit Testing）了。当重构完成之后，如果新的代码仍然能通过单元测试，那就说明代码原有逻辑的正确性未被破坏，原有的外部可见行为未变，符合上一节课中我们对重构的定义。

那今天我们就来学习一下单元测试。今天的内容主要包含这样几个内容：

- 什么是单元测试？
- 为什么要写单元测试？
- 如何编写单元测试？
- 如何在团队中推行单元测试？

话不多说，让我们现在就开始今天的学习吧！

什么是单元测试？

单元测试由研发工程师自己来编写，用来测试自己写的代码的正确性。我们常常将它跟集成测试放到一块来对比。单元测试相对于集成测试（Integration Testing）来说，测试的粒度更小一些。集成测试的测试对象是整个系统或者某个功能模块，比如测试用户注册、登录功能是否正常，是一种端到端（end to end）的测试。而单元测试的测试对象是类或者函数，用来测试一个类和函数是否都按照预期的逻辑执行。这是代码层级的测试。

这么说比较理论，我举个例子来解释一下。

```
public class Text {
  private String content;

  public Text(String content) {
    this.content = content;
  }

  /**
   * 将字符串转化成数字，忽略字符串中的首尾空格；
   * 如果字符串中包含除首尾空格之外的非数字字符，则返回null。
   */
  public Integer toNumber() {
    if (content == null || content.isEmpty()) {
      return null;
    }
    //...省略代码实现...
    return null;
```

```
  }
}
```

如果我们要测试Text类中的toNumber()函数的正确性，应该如何编写单元测试呢？

实际上，写单元测试本身不需要什么高深技术。它更多的是考验程序员思维的缜密程度，看能否设计出覆盖各种正常及异常情况的测试用例，来保证代码在任何预期或非预期的情况下都能正确运行。

为了保证测试的全面性，针对toNumber()函数，我们需要设计下面这样几个测试用例。

- 如果字符串只包含数字：“123”，toNumber()函数输出对应的整数：123。
- 如果字符串是空或者null，toNumber()函数返回：null。
- 如果字符串包含首尾空格：“ 123”，“123 ”，“ 123 ”，toNumber()返回对应的整数：123。
- 如果字符串包含多个首尾空格：“ 123 ”，toNumber()返回对应的整数：123；
- 如果字符串包含非数字字符：“123a4”，“123 4”，toNumber()返回null；

当我们设计好测试用例之后，剩下的就是将其翻译成代码了。翻译成代码的过程非常简单，我把代码贴在下面了，你可以参考一下（注意，我们这里没有使用任何测试框架）。

```
public class Assert {
  public static void assertEquals(Integer expectedValue, Integer actualValue) {
    if (actualValue != expectedValue) {
      String message = String.format(
              "Test failed, expected: %d, actual: %d.", expectedValue, actualValue);
      System.out.println(message);
    } else {
      System.out.println("Test succeeded.");
    }
  }

  public static boolean assertNull(Integer actualValue) {
    boolean isNull = actualValue == null;
    if (isNull) {
      System.out.println("Test succeeded.");
    } else {
      System.out.println("Test failed, the value is not null:" + actualValue);
    }
    return isNull;
  }
}

public class TestCaseRunner {
  public static void main(String[] args) {
    System.out.println("Run testToNumber()");
    new TextTest().testToNumber();

    System.out.println("Run testToNumber_nullorEmpty()");
    new TextTest().testToNumber_nullorEmpty();

    System.out.println("Run testToNumber_containsLeadingAndTrailingSpaces()");
    new TextTest().testToNumber_containsLeadingAndTrailingSpaces();
```

```
    System.out.println("Run testToNumber_containsMultiLeadingAndTrailingSpaces()");
    new TextTest().testToNumber_containsMultiLeadingAndTrailingSpaces();

    System.out.println("Run testToNumber_containsInvalidCharaters()");
    new TextTest().testToNumber_containsInvalidCharaters();
  }
}

public class TextTest {
  public void testToNumber() {
    Text text = new Text("123");
    Assert.assertEquals(123, text.toNumber());
  }

  public void testToNumber_nullorEmpty() {
    Text text1 = new Text("");
    Assert.assertNull(text1.toNumber());

    Text text2 = new Text("");
    Assert.assertNull(text2.toNumber());
  }

  public void testToNumber_containsLeadingAndTrailingSpaces() {
    Text text1 = new Text(" 123");
    Assert.assertEquals(123, text1.toNumber());

    Text text2 = new Text("123 ");
    Assert.assertEquals(123, text2.toNumber());

    Text text3 = new Text(" 123 ");
    Assert.assertEquals(123, text3.toNumber());
  }

  public void testToNumber_containsMultiLeadingAndTrailingSpaces() {
    Text text1 = new Text("  123");
    Assert.assertEquals(123, text1.toNumber());

    Text text2 = new Text("123  ");
    Assert.assertEquals(123, text2.toNumber());

    Text text3 = new Text("  123  ");
    Assert.assertEquals(123, text3.toNumber());
  }

  public void testToNumber_containsInvalidCharaters() {
    Text text1 = new Text("123a4");
    Assert.assertNull(text1.toNumber());

    Text text2 = new Text("123 4");
    Assert.assertNull(text2.toNumber());
  }
}
```

为什么要写单元测试?

单元测试除了能有效地为重构保驾护航之外，也是保证代码质量最有效的两个手段之一（另一个是Code Review）。我在Google工作的时候，写了大量的单元测试代码，结合我的这些开发经验，我总结了以下几点单元测试的好处。尽管有些听起来有点“务虚”，但如果你认真写过一些单元测试的话，应该会很有共鸣。

1.单元测试能有效地帮你发现代码中的bug

能否写出bug free的代码，是判断工程师编码能力的重要标准之一，也是很多大厂面试考察的重点，特别是像FLAG这样的外企。即便像我这样代码写了十几年，逻辑还算缜密、清晰的人，通过单元测试也常常会发现代码中的很多考虑不全面的地方。

在离开Google之后，尽管我就职的很多公司，其开发模式都是“快、糙、猛”，对单元测试根本没有要求，但我还是坚持为自己提交的每一份代码，都编写完善的单元测试。得益于此，我写的代码几乎是bug free的。这也节省了我很多fix低级bug的时间，能够有时间去做其他更有意义的事情，我也因此在工作上赢得了很多人的认可。可以这么说，坚持写单元测试是保证我的代码质量的一个“杀手锏”，也是帮助我拉开与其他人差距的一个“小秘密”。

2.写单元测试能帮你发现代码设计上的问题

前面我们提到，代码的可测试性是评判代码质量的一个重要标准。对于一段代码，如果很难为其编写单元测试，或者单元测试写起来很吃力，需要依靠单元测试框架里很高级的特性才能完成，那往往就意味着代码设计得不够合理，比如，没有使用依赖注入、大量使用静态函数、全局变量、代码高度耦合等。

3.单元测试是对集成测试的有力补充

程序运行的bug往往出现在一些边界条件、异常情况下，比如，除数未判空、网络超时。而大部分异常情况都比较难在测试环境中模拟。而单元测试可以利用下一节课中讲到的mock的方式，控制mock的对象返回我们需要模拟的异常，来测试代码在这些异常情况的表现。

除此之外，对于一些复杂系统来说，集成测试也无法覆盖得很全面。复杂系统往往有很多模块。每个模块都有各种输入、输出、异常情况，组合起来，整个系统就有无数测试场景需要模拟，无数的测试用例需要设计，再强大的测试团队也无法穷举完备。

尽管单元测试无法完全替代集成测试，但如果我们能保证每个类、每个函数都能按照我们的预期来执行，底层bug少了，那组装起来的整个系统，出问题的概率也就相应减少了。

4.写单元测试的过程本身就是代码重构的过程

上一节课中，我们提到，要把持续重构作为开发的一部分来执行，那写单元测试实际上就是落地执行持续重构的一个有效途径。设计和实现代码的时候，我们很难把所有的问题都想清楚。而编写单元测试就相当于对代码的一次自我Code Review，在这个过程中，我们可以发现一些设计上的问题（比如代码设计的不可测试）以及代码编写方面的问题（比如一些边界条件处理不当）等，然后针对性的进行重构。

5.阅读单元测试能帮助你快速熟悉代码

阅读代码最有效的手段，就是先了解它的业务背景和设计思路，然后再去看代码，这样代码读起来就会轻松很多。但据我了解，程序员都不怎么喜欢写文档和注释，而大部分程序员写的代码又很难做到“不言自明”。在没有文档和注释的情况下，单元测试就起了替代性作用。单元测试用例实际上就是用户用例，反映了代码的功能和如何使用。借助单元测试，我们不需要深入的阅读代码，便能知道代码实现了什么功能，有哪些特殊情况需要考虑，有哪些边界条件需要处理。

6.单元测试是TDD可落地执行的改进方案

测试驱动开发（Test-Driven Development，简称TDD）是一个经常被提及但很少被执行的开发模式。它的

核心指导思想就是测试用例先于代码编写。不过，要让程序员能彻底地接受和习惯这种开发模式还是挺难的，毕竟很多程序员连单元测试都懒得写，更何况在编写代码之前先写好测试用例了。

我个人觉得，单元测试正好是对TDD的一种改进方案，先写代码，紧接着写单元测试，最后根据单元测试反馈出来问题，再回过头去重构代码。这个开发流程更加容易被接受，更加容易落地执行，而且又兼顾了TDD的优点。

如何编写单元测试？

前面在讲什么是单元测试的时候，我们举了一个给toNumber()函数写单元测试的例子。根据那个例子，我们可以总结得出，写单元测试就是针对代码设计覆盖各种输入、异常、边界条件的测试用例，并将这些测试用例翻译成代码的过程。

在把测试用例翻译成代码的时候，我们可以利用单元测试框架，来简化测试代码的编写。比如，Java中比较出名的单元测试框架有Junit、TestNG、Spring Test等。这些框架提供了通用的执行流程（比如执行测试用例的TestCaseRunner）和工具类库（比如各种Assert判断函数）等。借助它们，我们在编写测试代码的时候，只需要关注测试用例本身的编写即可。

针对toNumber()函数的测试用例，我们利用Junit单元测试框架重新实现一下，具体代码如下所示。你可以拿它跟之前没有利用测试框架的实现方式对比一下，看是否简化了很多呢？

```
import org.junit.Assert;
import org.junit.Test;

public class TextTest {
  @Test
  public void testToNumber() {
    Text text = new Text("123");
    Assert.assertEquals(new Integer(123), text.toNumber());
  }

  @Test
  public void testToNumber_nullorEmpty() {
    Text text1 = new Text("");
    Assert.assertNull(text1.toNumber());

    Text text2 = new Text("");
    Assert.assertNull(text2.toNumber());
  }

  @Test
  public void testToNumber_containsLeadingAndTrailingSpaces() {
    Text text1 = new Text(" 123");
    Assert.assertEquals(new Integer(123), text1.toNumber());

    Text text2 = new Text("123 ");
    Assert.assertEquals(new Integer(123), text2.toNumber());

    Text text3 = new Text(" 123 ");
    Assert.assertEquals(new Integer(123), text3.toNumber());
  }

  @Test
  public void testToNumber_containsMultiLeadingAndTrailingSpaces() {
    Text text1 = new Text("  123");
```

```
    Assert.assertEquals(new Integer(123), text1.toNumber());

    Text text2 = new Text("123  ");
    Assert.assertEquals(new Integer(123), text2.toNumber());

    Text text3 = new Text("  123  ");
    Assert.assertEquals(new Integer(123), text3.toNumber());
  }

  @Test
  public void testToNumber_containsInvalidCharaters() {
    Text text1 = new Text("123a4");
    Assert.assertNull(text1.toNumber());

    Text text2 = new Text("123 4");
    Assert.assertNull(text2.toNumber());
  }
}
```

对于如何使用这些单元测试框架，大部分框架都给出了非常详细的官方文档，你可以自行查阅。这些东西理解和掌握起来没有太大难度，所以这不是专栏要讲解的重点。关于如何编写单元测试，我更希望传达给你一些我的经验总结。具体包括以下几点。

1.写单元测试真的是件很耗时的事情吗？

尽管单元测试的代码量可能是被测代码本身的1～2倍，写的过程很繁琐，但并不是很耗时。毕竟我们不需要考虑太多代码设计上的问题，测试代码实现起来也比较简单。不同测试用例之间的代码差别可能并不是很大，简单copy-paste改改就行。

2.对单元测试的代码质量有什么要求吗？

单元测试毕竟不会在产线上运行，而且每个类的测试代码也比较独立，基本不互相依赖。所以，相对于被测代码，我们对单元测试代码的质量可以放低一些要求。命名稍微有些不规范，代码稍微有些重复，也都是没有问题的。

3.单元测试只要覆盖率高就够了吗？

单元测试覆盖率是比较容易量化的指标，常常作为单元测试写得好坏的评判标准。有很多现成的工具专门用来做覆盖率统计，比如，JaCoCo、Cobertura、Emma、Clover。覆盖率的计算方式有很多种，比较简单的是语句覆盖，稍微高级点的有：条件覆盖、判定覆盖、路径覆盖。

不管覆盖率的计算方式如何高级，将覆盖率作为衡量单元测试质量的唯一标准是不合理的。实际上，更重要的是要看测试用例是否覆盖了所有可能的情况，特别是一些corner case。我来举个简单的例子解释一下。

像下面这段代码，我们只需要一个测试用例就可以做到100%覆盖率，比如cal(10.0, 2.0)，但并不代表测试足够全面了，我们还需要考虑，当除数等于 0 的情况下，代码执行是否符合预期。

```
public double cal(double a, double b) {
  if (b != 0) {
    return a / b;
  }
```

```
}
```

实际上，过度关注单元测试的覆盖率会导致开发人员为了提高覆盖率，写很多没有必要的测试代码，比如get、set方法非常简单，没有必要测试。从过往的经验上来讲，一个项目的单元测试覆盖率在60～70%即可上线。如果项目对代码质量要求比较高，可以适当提高单元测试覆盖率的要求。

4.写单元测试需要了解代码的实现逻辑吗？

单元测试不要依赖被测试函数的具体实现逻辑，它只关心被测函数实现了什么功能。我们切不可为了追求覆盖率，逐行阅读代码，然后针对实现逻辑编写单元测试。否则，一旦对代码进行重构，在代码的外部行为不变的情况下，对代码的实现逻辑进行了修改，那原本的单元测试都会运行失败，也就起不到为重构保驾护航的作用了，也违背了我们写单元测试的初衷。

5.如何选择单元测试框架？

写单元测试本身不需要太复杂的技术，大部分单元测试框架都能满足。在公司内部，起码团队内部需要统一单元测试框架。如果自己写的代码用已经选定的单元测试框架无法测试，那多半是代码写得不够好，代码的可测试性不够好。这个时候，我们要重构自己的代码，让其更容易测试，而不是去找另一个更加高级的单元测试框架。

单元测试为何难落地执行？

虽然很多书籍中都会讲到，单元测试是保证重构不出错的有效手段；也有非常多人已经认识到单元测试的重要性。但是有多少项目有完善的、高质量的单元测试呢？据我了解，真的非常非常少，包括BAT这样级别公司的项目。如果不相信的话，你可以去看一下国内很多大厂开源的项目，有很多项目完全没有单元测试，还有很多项目的单元测试写得非常不完备，仅仅测试了逻辑是否运行正确而已。所以，100%落实执行单元测试是件“知易行难”的事。

写单元测试确实是一件考验耐心的活儿。一般情况下，单元测试的代码量要大于被测试代码量，甚至是要多出好几倍。很多人往往会觉得写单元测试比较繁琐，并且没有太多挑战，而不愿意去做。有很多团队和项目在刚开始推行单元测试的时候，还比较认真，执行得比较好。但当开发任务紧了之后，就开始放低对单元测试的要求，一旦出现破窗效应，慢慢的，大家就都不写了，这种情况很常见。

还有一种情况就是，由于历史遗留问题，原来的代码都没有写单元测试，代码已经堆砌了十几万行了，不可能再一个一个去补单元测试。这种情况下，我们首先要保证新写的代码都要有单元测试，其次，每次在改动到某个类时，如果没有单元测试就顺便补上，不过这要求工程师们有足够强的主人翁意识（ownership），毕竟光靠leader督促，很多事情是很难执行到位的。

除此之外，还有人觉得，有了测试团队，写单元测试就是浪费时间，没有必要。程序员这一行业本该是智力密集型的，但现在很多公司把它搞成劳动密集型的，包括一些大厂，在开发过程中，既没有单元测试，也没有Code Review流程。即便有，做的也是差强人意。写好代码直接提交，然后丢给黑盒测试狠命去测，测出问题就反馈给开发团队再修改，测不出的问题就留在线上出了问题再修复。

在这样的开发模式下，团队往往觉得没有必要写单元测试，但如果我们把单元测试写好、做好Code Review，重视起代码质量，其实可以很大程度上减少黑盒测试的投入。我在Google的时候，很多项目几乎没有测试团队参与，代码的正确性完全靠开发团队来保障，线上bug反倒非常少。

以上是我对单元测试的认知和实践心得。现在互联网信息如此的公开透明，网上有很多文章可以参考，对于程序员这个具有很强学习能力的群体来说，学会如何写单元测试并不是一件难事，难的是能够真正感受到它的作用，并且打心底认可、能100%落地执行。这也是我今天的课程特别想传达给你的一点。

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要掌握的重点内容。

1.什么是单元测试?

单元测试是代码层面的测试，由研发自己来编写，用于测试“自己”编写的代码的逻辑的正确性。单元测试顾名思义是测试一个“单元”，有别于集成测试，这个“单元”一般是类或函数，而不是模块或者系统。

2.为什么要写单元测试?

写单元测试的过程本身就是代码Code Review和重构的过程，能有效地发现代码中的bug和代码设计上的问题。除此之外，单元测试还是对集成测试的有力补充，还能帮助我们快速熟悉代码，是TDD可落地执行的改进方案。

3.如何编写单元测试?

写单元测试就是针对代码设计各种测试用例，以覆盖各种输入、异常、边界情况，并将其翻译成代码。我们可以利用一些测试框架来简化单元测试的编写。除此之外，对于单元测试，我们需要建立以下正确的认知：

- 编写单元测试尽管繁琐，但并不是太耗时；
- 我们可以稍微放低对单元测试代码质量的要求；
- 覆盖率作为衡量单元测试质量的唯一标准是不合理的；
- 单元测试不要依赖被测代码的具体实现逻辑；
- 单元测试框架无法测试，多半是因为代码的可测试性不好。

4.单元测试为何难落地执行?

一方面，写单元测试本身比较繁琐，技术挑战不大，很多程序员不愿意去写；另一方面，国内研发比较偏向“快、糙、猛”，容易因为开发进度紧，导致单元测试的执行虎头蛇尾。最后，关键问题还是团队没有建立对单元测试正确的认识，觉得可有可无，单靠督促很难执行得很好。

课堂讨论

今天的课堂讨论有以下两个：

1. 你参与的项目有没有写单元测试？单元测试是否足够完备？贯彻执行写单元测试的过程中，遇到过哪些问题？又是如何解决的？
2. 在面试中，我经常会让候选人写完代码之后，列举几个测试用例，以此来考察候选人考虑问题是否全面，特别是针对一些边界条件的处理。所以，今天的另一个课堂讨论话题就是：写一个二分查找的变体算法，查找递增数组中第一个大于等于某个给定值的元素，并且为你的代码设计完备的单元测试用例。

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 流年 2020-01-06 02:57:41

  半年前，因为团队项目太多太乱已经很难维护和协作开发(10人的开发团队，每人负责一些项目，水平参差不齐，各自独立开发)，作为团队中的资深者，我被leader要求开发一套通用的底层框架。

  为保证代码质量，刚开始时对自己要求严格，每个方法都必须要有多种case的单元测试；然后发现有时候写出来的单元测试代码比被测试的方法的代码量多很多，在一定程度上影响了开发速度。另外leader还经常安排我去修复一些仍在艰难运行的旧系统的故障(大多是累积下来的技术债)，导致框架开发进展一再拖延。同时团队其他人很少有写单元测试代码，测试工作完全依赖测试人员完成，对自己也就逐渐放松了要求，单元测试不再追求完备，只在核心的方法中加入常规的实现逻辑测试，其他代码写完多看两遍确认无bug就提交。

  争哥的这节课程我完全理解，单元测试的重要性毋庸置疑，可是在实际开发过程中完全落实存在一定的困难，遇到这种问题我还真没啥解决的办法除了让自己拼命的加班，真的太难了。。。 [17赞]

- 桂城老托尼 2020-01-06 12:32:16

  感谢争哥分享，单元测试很重要，除此之外，集成用例和回归用例库同样重要，以及上线后的ab比对切流，这些在大厂其实都是落地了的常规武器。这里争哥没有提到。
  大厂之外，能落地的除了单测，还有简单的ab框架，集成平台自动化程度，否则一次重构下来非常耗费精力，而且还是冒着风险。
  另外，单测代码本身的质量也要有要求，tl要求单测代码和生产代码一样要遵守规范(视各厂情况定吧)。
  所以每次迭代开发测试时间比是1比2差不多了。 哭晕 [2赞]

- 李小四 2020-01-06 11:06:07

  设计模式_28
  1. 有过一次失败的单元测试经验：好不容易申请到了2周的预研时间，我开开心心地研究怎么把JUnit引入项目，刚开始了两天，新的开发任务打断了我的计划，然后就再也没有继续了。。。

  2.
  代码：

```
/**
* 查找递增数组中第一个大于等于某个给定值的元素
* @return -1: 未找到
*/
public int findFirstEqualOrLargerIndex(int[] array, int num) {
if (array == null || array.length == 0) return -1;

int start = 0;
int end = array.length - 1;

while (start != end) {
int middle = start + (end - start) / 2;

if (array[middle] >= num) {
if (start == middle) return middle;
else
if (array[middle - 1] < num) return middle;
```

```
else end = middle -1;

} else {
start = middle + 1;
}
}
//start == end
if (array[start] >= num) {
return start;
} else {
return -1;
}
}
```

测试用例:

```
findFirstEqualOrLargerIndex(null, 1)
findFirstEqualOrLargerIndex(new int [0], 1)
findFirstEqualOrLargerIndex(new int [] {0}, 1)
findFirstEqualOrLargerIndex(new int [] {1}, 1)
findFirstEqualOrLargerIndex(new int [] {0, 0}, 1)
findFirstEqualOrLargerIndex(new int [] {0, 1}, 1)
findFirstEqualOrLargerIndex(new int [] {1, 1}, 1)
findFirstEqualOrLargerIndex(new int [] {0, 1, 2}, 1)
findFirstEqualOrLargerIndex(new int [] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, 1)
findFirstEqualOrLargerIndex(new int [] {0, 1, 1, 1, 1, 1, 6, 7, 8, 9}, 1)
findFirstEqualOrLargerIndex(new int [] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, 10)
```

我估计应该有漏洞，请老师和同学们指正~ [2赞]

- 辣么大 2020-01-06 08:50:59
  关于问题2，尝试写了一下单元测试：
  https://github.com/gdhucoder/Algorithms4/tree/master/designpattern/u28 [2赞]

- leon 2020-01-06 01:22:20
  坐标阿里，周边同事几乎没有写单元测试的习惯，就算写也只是为了测试当前版本的功能，而不是为了以后迭代和重构。
  很可悲 [2赞]

- 逍遥思 2020-01-06 15:12:05
  独立开发者，项目代码量 10W 行以内，在可以预见的未来不会超过 20W 行
  以前试过 git 各种最佳实践，最后发现一个分支基本就够用了
  所以还是忍不住想问问老师，如果项目真没那么大，是否需要单元测试？ [1赞]

- 黄林晴 2020-01-06 08:48:52
  打卡✔
  我觉得写单元测试的难点是覆盖测试用例
  我们出的bug 不都是没考虑特殊情况吗，如果单元测试可以想到全部的测试用例，代码有bug 的可能性应该不大 [1赞]

- 再见孙悟空 2020-01-06 08:30:32

确实单元测试只在一开始才写过，后来发现拖慢了开发的进度，就渐渐放弃了，现在我们的开发团队里几乎没什么单元测试，除非一些涉及到优惠券，订单奖励计算等和钱挂钩的业务，我觉得一方面是因为消耗时间，另一方面业务代码没写好，很多时候很不利于进行单测，要造各种数据。我觉得单测最好是在开发一个方法或者函数之后就进行，且要在测试介入之前，否则测试介入以后再补充单元测试，有可能会改动到已写好的业务，那么就又需要回归测试一遍，对开发，测试都是很消耗的。 [1赞]

- 峰 2020-01-06 08:20:24
  我感觉我写单测最大的问题在于很难把代码写成那种细粒度可测的模样，而不是要去写。 [1赞]

- William 2020-01-07 09:11:10
  单元测试很重要，之前写某项目的时间足够多，写过场景覆盖率很广的测试案例结果就是0bug上线.
  最近的项目由于开发时间紧，现在有bug了...

- 許敲敲 2020-01-07 08:15:21
  现在就在写单元测试，感觉这个需要对原程序很懂才行。还有单元测试虽然粒度小，不过有的依赖很多的话，需要各种Mock，感觉也不简单啊…… 不过香是香

- 张迪 2020-01-07 00:38:24
  单元测试是测试应用层还是领域层?

- 张迪 2020-01-07 00:30:17
  写单元测试就是不知道如何命名单元测试的方法名，有时候这个方法都不知道如何描述好，

- Frank 2020-01-06 23:46:11
  以前在开发中，没有写单元测试的意识。开发完功能后，直接去测试一个完整的流程。即前端发请求，服务端处理，看数据库数据。如果功能正确就过。这是从一个功能宏观去考虑测试。而单元测试是更细粒度的测试，它在保证各个“单元”都测试通过的情况下整个功能模块就测试通过了。这样的方式对于我们自己来说对代码可控粒度更细。更能比较清楚的理解某个“单元”在整个功能模块调用链路上的位置，承担什么职责，以及有什么行为。而不是一开始就站在模块宏观角度来思考。通过一个个单元测试的编写，将整个功能模块串联起来，最终达到整个功能模块的全局认知。 这也体现了任务分解的思想。通过单元测试，可以从另外一方面实现对已编写的代码的CodeReview，重新梳理流程。也为以后有重构需求打下基础。
  目前参与的项目中有单元测试，但是不够完备。可能由于某些原因（开发人员意识问题，团队对单元测试的执行落地程度不够等）。在写单元测试的过程中，遇到单元测试依赖数据库查询问题，因为存在多套环境，如开发环境，仿真环境，线上环境。对于依赖数据查询的单元测试，只能自己造假数据来解决。不知道还有什么好的解决办法。

- DullBird 2020-01-06 23:18:05
  刚开始接触单元测试的时候。会编写完整的单元测试，但是当时接触的是CURD的接口，比如根据条件批量查询符合条件的员工的一个service接口，然后部分数据通过缓存，部分数据通过db组合在一起，比如调用cacheManager + UserMapper,测试这个接口就需要mock cacheManager和UserMapper,导致代码测试起来比较麻烦，大量时间花在编写mock对象，但是其实和这个接口对外的功能又没关系，是内部的实现逻辑有关系。这一点比较疑惑，觉得这样测试已经违背了理解代码内部逻辑的原则，但是不构造这些异常，这个接口又没什么好测试的，对于代码的可测试性的概念，如何提升可测试性，还是模糊的。
  后面虽然没有坚持写完整的单元测试了，但是程序的正确性流程。还是会编写单元测试走走一遍。

- 平风造雨 2020-01-06 23:09:15
  https://github.com/zhangw/misc.java.jimohou.me/blob/master/src/test/java/test/geekbang/design/pattern/beauty/artical28/HomeWorkTest.java

- 刘大明 2020-01-06 22:20:23
  说起来真的是难受。整个项目中就我一个人写单元测试。每次做的功能都有单元测试覆盖。而且项目中junit包都是我导入的。更加奇葩的是我的功能单元测试领导还不让我提交。说是你的测试代码为什么要提交，我瞬间无语了。

- 七楼 2020-01-06 19:10:31
  单元测试 让我的逻辑更缜密了 的确有好处 而且bug也少

- Jeff.Smile 2020-01-06 19:00:28
  程序员这一行业本该是智力密集型的，但现在很多公司把它搞成劳动密集型的。
  ————————————————————
  你这句话道出了现实！哈哈😓

- Arthur.Li 2020-01-06 18:59:52
  项目越来越大，和复杂化，每次新增功能或者改造测试都很麻烦，还容易测试不到。就是单元测试不够，今年目标是把核心功能加上单元测试，估计能节省大量的测试时间。
  单元测试能检验代码好坏，是不是高耦合，确实，如果不重构，单元测试都没法写了。