单元测试及代码覆盖率提升方案

目录

[1. 概述 2](#_Toc24421991)

[2. 单元测试意义和衡量指标 2](#_Toc24421992)

[2.1 单元测试 2](#_Toc24421993)

[2.2 代码覆盖率 3](#_Toc24421994)

[3. 单元测试改进 4](#_Toc24421995)

[3.1 原本方案不足之处 4](#_Toc24421996)

[3.2 修改后的方案 5](#_Toc24421997)

[需求分析及测试用例 5](#_Toc24421998)

[3.3.1 模块划分 5](#_Toc24421999)

[3.3.2 测试用例设计 6](#_Toc24422000)

[3.3 代码实现 8](#_Toc24422001)

[4. 结果 8](#_Toc24422002)

[4.1 覆盖率测试 8](#_Toc24422003)

[5. 后续 9](#_Toc24422004)

[不足与思考 9](#_Toc24422005)

[参考 9](#_Toc24422006)

# 概述

面试官您好，在这篇文章中，我会先根据自己所了解到的关于测试的内容，基于自己的理解，说明单元测试的意义和重点；随后我会以之前编写的precise-operation项目为例，分析原本设计的不足之处，并尝试使用新的方案进行修改，并将修改后的测试结果附在文章的最后。

# 单元测试意义和衡量指标

## 2.1 单元测试

测试有很多种类型，包括单元测试、集成测试和端到端测试等等。其中单元测试指的是对实现某功能对程序模块进行正确性检验的测试工作。

在我的理解中，单元测试不应关心代码逻辑的实现过程和内部机制，是对模块的具体功能和边界条件进行检测的过程。而且，单元测试应该是开发流程中的一个部分，在设计阶段就应当根据需求进行分析和编写，而非代码实现结束后再补上（TDD）。

在之前我的认知中，测试应该包含两点：一、考虑到所有的逻辑和边界条件；二、测试到所有的业务代码逻辑。在了解了一些测试方法后，我发现前者可以通过对需求进行拆解分析，考虑边界条件和场景，使用诸如等价类划分法、边界值分析法和错误推测法的方法进行设计；后者则更像是一个检测机制，保证测试覆盖了所有代码部分，可以用代码覆盖率来表示。

在对单元测试内容的了解中我也看到了TDD的思想，我对它理解是：第一步，对需求进行分解和梳理，将大的需求分解成小的部分，也就是每一个具体的功能模块；第二步，对模块进行测试用例的设计和编写；最后一步才是每个功能模块的逻辑代码实现。其中第二和第三步是需要相互弥补和共同编写的，但是两者的重点和思考的出发点应该是不同的。

## 2.2 代码覆盖率

代码覆盖率可以分为几个层级：行覆盖率（Line Coverage）、函数覆盖率（Function Coverage）和分支覆盖率（Branch Coverage）。

在我的理解中，代码覆盖率是一个用来判断测试用例和优化代码逻辑的指标。就像测试结果标志着代码的健壮性，代码覆盖率一定程度上可以发现测试用例的不足之处。但是就像前文所说的，单元测试的编写和代码逻辑的编写的关注点是不同的，不应该为了通过测试而改写代码，或者为了提高覆盖率而刻意的添加测试用例。所以我认为代码覆盖率是一种方法，是一个参考的指标，用来增强测试质量，但并不是说覆盖率高的测试就一定更好。

# 单元测试改进

## 3.1 原本方案不足之处

在原本编写单元测试中，我的编写有几个问题：

1. 在主逻辑编写完成后才编写测试用例。虽然在编写测试用例的时候确实是从功能逻辑的角度出发进行设计，但是还是会受到一部分代码逻辑思维的影响；
2. 因为没有采用TDD的思想，因此原本在代码模块的分级和测试分级上有些划分不是很明确的部分；
3. 部分分支逻辑没有覆盖到。我跑了一遍测试的代码覆盖率，结果如下：



图 1 现有单元测试覆盖率



图 2 现有单元测试未覆盖逻辑

可以发现有部份场景在单元测试的时候没有被覆盖到，说明测试用例编写的并不完善。

## 3.2 修改后的方案

根据前文所说的单元测试编写的方案以及之前在项目编写过程中的不足之处，我在重新编写的时候采用了以下的步骤进行测试的设计及需求的实现：

1. 根据TDD的思想，抛开了原本的代码逻辑实现，只从需求角度进行分析和模块划分；
2. 根据等价类划分法，对模块进行测试用例的编写并实现；
3. 重写根据划分的模块编写代码；
4. 根据代码的模块设计划分的和测试结果进行重构，将代码模块重新整合。

## 需求分析及测试用例

题目的需求为“实现精确的四则运算”，因此，需要实现四项独立的运算功能。为简要起见，下文中我只以最复杂的除法操作的测试设计及模块划分的思路为例，加、减与乘的设计思路与之类似，会在代码中实现。

### 3.3.1 模块划分

题目的需求是进行精确的除法计算，而除法计算的不精确主要是由浮点数的不精确造成的。不会造成不精确结果的数字没有必要进行额外的运算，因此，我将精确除法计算分为了四个模块：数据有效性判断模块、数据处理判断模块、数据预处理模块及精确除法模块。

具体的除法计算流程图如下图所示。



图 3除法计算过程流程图

### 3.3.2 测试用例设计

在设计中我准备针对除法编写五个测试用例，分别覆盖前文提到的四个模块及对外提供的整体接口。因为整体的测试用例设计考虑最全，基本覆盖了剩余四个模块的测试用例，所以为简要起见，下文将只说明整体接口部分的用例设计过程，剩余模块的内容在代码中实现。

首先，我采用等价类划分法列出下表：

表格 1 除法测试用例等价划分法分析

|  |  |
| --- | --- |
| **有效等价类** | **无效等价类** |
| 1. 输入两个普通数字（除0,NaN,Inifinity外的数字） 2. 输入两个以上普通数字 3. 输入两个普通数字字符 4. 输入两个以上普通数字字符 5. 输入普通数字与普通数字字符 6. 输入两个以上普通数字与普通数字字符 7. 输入两个非标准格式普通数字字符 8. 输入一个普通数字与一个非标准格式普通数字 9. 输入大小超过安全范围的普通数字 10. 输入精度超过15位的数字 11. 输入两个非普通数字 12. 输入普通数字与非普通数字（在除法中为0, NaN, Infinity） 13. 输入非标准格式普通数字与非普通数字 14. 输入两个以上普通数字与非普通数字组合 15. 输入两个以上非普通数字 | 1. 输入小于两个参数 2. 输入包含非数字字符串 3. 输入包含布尔值 4. 输入包含null 5. 输入包含undefined 6. 输入包含Symbol 7. 输入包含对象 8. 输入包含数组 |

随后，我根据上表，利用边界值分析法和错误推测法，编写了下列测试用例及结果，为简要起见，只列出对精度为21（也就是custom-calculation的默认精度）的预期结果：

表格 2 除法测试用例

|  |  |  |
| --- | --- | --- |
| **对应等价类** | **输入** | **输出** |
| 1 | 1, 2 | '0.5' |
| 1 | 2, -1 | '-2' |
| 1 | 2.24, 100 | '0.0224' |
| 1 | -0.3, 10 | '-0.03' |
| 1 | 0.6, 0.2 | '3' |
| 1 | -0.5, 0.3 | '-1.66666666666666666667' |
| 1 | 1, 0.3 | '3.3333333333333333333' |
| 2 | 2.24, 100, 0.2 | '0.112' |
| 3 | '1', '2' | '0.5' |
| 3 | '-2.24', '100' | '-0.0224' |
| 3 | '0.6', '0.2' | '3' |
| 3 | '224e-2', '10e3' | '0.000224' |
| 3 | '-0.5', '0.3' | '-1.66666666666666666667' |
| 4 | '2.24', '100', '-0.2' | '-0.112' |
| 5 | 1, '-2' | '-0.5' |
| 5 | '0.6', -0.2 | '-3' |
| 5 | -0.5, '0.3' | '-1.66666666666666666667' |
| 6 | '2.24', 100, -0.2 | '-0.112' |
| 7 | '002.24', '01e2' | '0.0224' |
| 7 | '1', '0.00' | 'Inifinity' |
| 8 | '22.4000e-1', 100 | '0.0224' |
| 9 | '27021597764222986.92', '2' | '13510798882111493.46' |
| 10 | '0.1234567891234567891', '-0.2' | '-0.6172839456172839455' |
| 11 | NaN, 0 | 'NaN' |
| 11 | NaN, Infinity | 'NaN' |
| 11 | NaN, NaN | 'NaN' |
| 11 | 0, Infinity | '0' |
| 11 | 0, 0 | 'NaN' |
| 11 | 0, NaN | 'NaN' |
| 11 | Infinity, NaN | 'NaN' |
| 11 | Infinity, Infinity | 'NaN' |
| 11 | Infinity, 0 | 'Infinity' |
| 12 | NaN, 1 | 'NaN' |
| 12 | 1, 0 | 'Infinity' |
| 12 | 1, -0 | '-Infinity' |
| 12 | 0, 1 | '0' |
| 13 | '22.4000e-1', 0 | 'Infinity' |
| 14 | 1, 0, 3 | 'Infinity' |
| 15 | Infinity, 0, Infinity | 'NaN' |
| 16 | 无输入 | Throw error |
| 16 | 1 | Throw error |
| 17 | 1, 'a' | Throw error |
| 18 | 1, true | Throw error |
| 18 | 1, false | Throw error |
| 19 | 1, null | Throw error |
| 20 | 1, undefined | Throw error |
| 21 | 1, Symbol('') | Throw error |
| 22 | 1, {} | Throw error |
| 23 | 1, [] | Throw error |

## 代码实现

逻辑代码与测试用例的实现详见代码；其中改进的地方有：

1. 原本的pre文件夹下的内容重命名划分为三个模块，并根据模块划分的设计，从函数入口处移动至每个策略类的调用处，并将测试内容从pre.test.js移动到对应的每个文件中；
2. 核心逻辑进行了小规模的重构，并根据单元测试的结果进行了部分去重；
3. 添加了精确加、减、乘及除四个模块的测试用例；
4. 根据设计用例的分析及设计，添加了部分原本缺失的用例。

# 结果

## 4.1 覆盖率测试



# 后续

## 5.1 不足与思考

测试用例的设计 冗余重复怎么办？根据代码补充了部分测试用例，怎么保证都覆盖到了？

# 参考

1. <https://martinfowler.com/bliki/TestPyramid.html>
2. <https://www.jianshu.com/p/62f16cd4fef>