单元测试及代码覆盖率提升方案

目录

1. 概述

面试官您好，在这篇文章中，我会先根据这周对测试方法的学习和基于其的个人理解，说明单元测试的意义和重点；随后我会以之前写的 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。

2. 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。

单元测试不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）。

一个完善的测试应该包含两点：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。在查阅了一些资料并学习了一些测试方法后，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；相对来说，后者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，我对它的理解是：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类别：行覆盖率（Line Coverage）、函数覆盖率（Function Coverage）和分支覆盖率（Branch Coverage）。

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。像测试结果标志着代码的健壮性一样，代码覆盖率一定程度上可以发现测试用例的不足。但是，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，或是为了提高覆盖率而刻意的添加测试用例。所以，代码覆盖率是一种测试的方法，也是一个参考的指标，可以用来增强测试的质量，但并不是说覆盖率高的测试就一定更好。

3. 单元测试改进

3.1 原本方案不足之处

在总局我原本单元测试的编写过程中，我发现存在几个问题：

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：

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	81.93	87.56	67.21	81.72	
src	89.66	83.08	85.71	92.86	
config.js	100	100	100	100	
error.js	33.33	0	33.33	33.33	16,23
index.js	100	100	100	100	
utils.js	82.5	80.85	92.86	90.32	67,69,70
src/calc	75.94	84.38	46.88	75.13	
custom-calculation.js	98.16	92.05	100	98.63	208,209
index.js	20	100	20	20	11,15,19,23
parse-to-int-calculation.js	0	0	0	0	... 03,105,106,108
precision-calculation.js	0	100	0	0	11,16,20,24,28
src/pre	100	100	100	100	
pre-calc.js	100	100	100	100	
pre-check.js	100	100	100	100	
pre-process.js	100	100	100	100	

图表 1 现有单元测试覆盖率

可以看到，代码覆盖率已经达到 90%以上了（有部分方法是另两个 calculation 方法特定使用的，因此不算在内）。

```
if (flag) {
  e++;
  count.unshift(1);
}
```

图表 2 现有单元测试未覆盖逻辑

但是经过分析可以看到，仍有部份场景在单元测试的时候没有被覆盖到，说明用例的编写的并不完善。

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，并且确定业务流程及所有可达边界；
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. 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。

这种方式不会针对覆盖率的数据指标进行刻意的设计，但是通过这样的流程实现后，代码覆盖率会自然的提高。

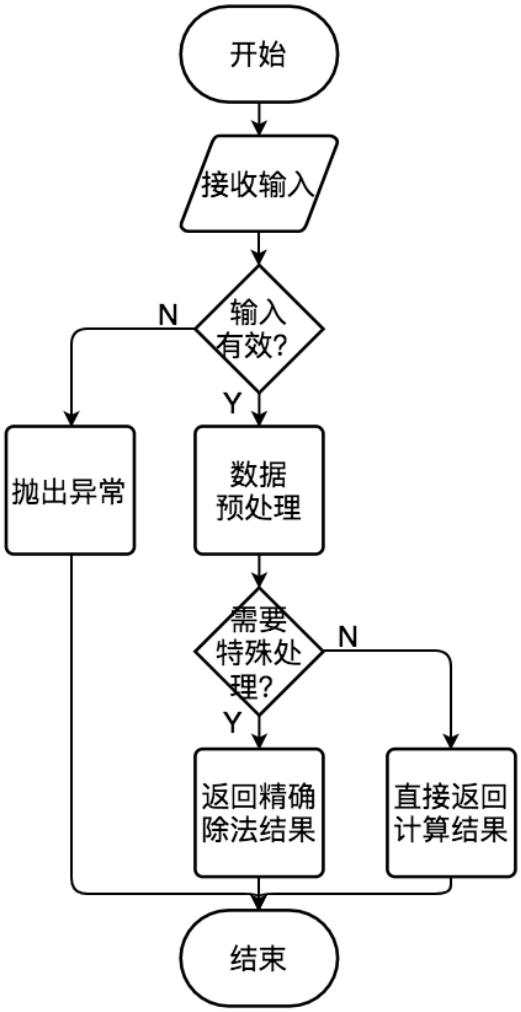
3.3 需求分析及测试用例

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，下文中我会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五个模块：***数据有效性判断模块***、***数据预处理模块***、***数据特殊处理判断模块***、***数据特殊处理模块***及***精确除法模块***。

具体的除法计算流程图如下图所示。



图表 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六个单元测试，分别覆盖前文提到的五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的方法尝试覆盖所有可能的计算场景。需要说明的是，在设计时我对除数与被除数分别进行等价类划分，然后对每一个项两两正交，最后合并冗余部分。等价类划分如下：

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	16. 输入小于两个参数 17. 输入包含非数字字符串 18. 输入包含布尔值 19. 输入包含 null 20. 输入包含 undefined 21. 输入包含 Symbol 22. 输入包含对象 23. 输入包含数组

随后，我根据上表，利用边界值分析法，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（即 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'	'-0.61728394561728394 55'
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

此外，测试用例会根据覆盖率结果进行修改。例如，图表 2 中的代码片段是精确加法中用于判断进位的分支逻辑，但在最早的测试用例编写过程中没有考虑到，因此出现了代码逻辑没有覆盖到的地方。在添加了相应的测试用例后，这条逻辑也得到了验证。

3.4 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四个模块，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，除 pre-process 和 pre-check 公用外，其余 test 文件都移动到了对应的模块测试文件夹下；
2. 策略模式的入口单独抽离为一层，供单元测试调用；并添加了精确加、减、乘及除四个模块的测试用例；
3. 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；
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。

4. 结果

4.1 覆盖率测试

由于我在项目中用了策略模式，所以单一一次的单元测试肯定没法覆盖全部的代码逻辑。因此，我对每种 calculation 策略进行了单独的单元测试和统计，结果如下。

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	82.45	87.56	70.15	82.2	
src	88.73	84.62	85.71	91.38	
config.js	100	100	100	100	
error.js	33.33	0	33.33	33.33	16,23
index.js	100	100	100	100	
utils.js	85	82.98	92.86	90.32	67,69,70
src/calc	78.77	85.42	46.88	78.24	
calculation.js	100	100	20	100	
custom-calculation.js	99.39	93.18	100	100	44,67,137
index.js	100	100	100	100	
parse-to-int-calculation.js	0	0	0	0	... 03,105,106,108
precision-calculation.js	0	100	0	0	11,16,20,24,28
src/pre	100	100	100	100	
pre-calc.js	100	100	100	100	
pre-check.js	100	100	100	100	
pre-process.js	100	100	100	100	
pre-valid.js	100	100	100	100	

图表 4 custom-calculation 测试代码覆盖率

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	42.72	50.78	64.18	43.28	
src	94.37	92.31	95.24	98.28	
config.js	100	100	100	100	
error.js	66.67	100	66.67	66.67	16
index.js	100	100	100	100	
utils.js	92.5	89.36	100	100	67,79,83,96,145
src/calc	20.28	6.25	28.13	21.76	
calculation.js	100	100	20	100	
custom-calculation.js	0	0	0	0	... 37,338,339,342
index.js	100	100	100	100	
parse-to-int-calculation.js	97.44	75	100	100	16,17
precision-calculation.js	0	100	0	0	11,16,20,24,28
src/pre	100	100	100	100	
pre-calc.js	100	100	100	100	
pre-check.js	100	100	100	100	
pre-process.js	100	100	100	100	
pre-valid.js	100	100	100	100	

图表 5 parse-to-int-calculation 测试代码覆盖率

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	30.46	45.08	56.72	29.85	
src	88.73	84.62	85.71	91.38	
config.js	100	100	100	100	
error.js	33.33	0	33.33	33.33	16,23
index.js	100	100	100	100	
utils.js	85	82.98	92.86	90.32	67,69,70
src/calc	4.72	0	18.75	5.18	
calculation.js	100	100	20	100	
custom-calculation.js	0	0	0	0	... 37,338,339,342
index.js	100	100	100	100	
parse-to-int-calculation.js	0	0	0	0	... 03,105,106,108
precision-calculation.js	100	100	100	100	
src/pre	100	100	100	100	
pre-calc.js	100	100	100	100	
pre-check.js	100	100	100	100	
pre-process.js	100	100	100	100	
pre-valid.js	100	100	100	100	

图表 6 precise-calculation 测试代码覆盖率

三者交叉后，基本上测试代码的行覆盖率达到了 100%，相比之前的代码覆盖率提高了 5%。

在复盘中，我发现核心逻辑中还有一句逻辑没有被覆盖到，代码如下。

```
 */
function minusArray(arr1, arr2) {
  I if (compareArrNum(arr1, arr2) < 0) return minusArray(arr2, arr1);
```

图表 7 custom-calculation 中测试未覆盖到的 line

这个函数的作用是计算 arr1 与 arr2 两个数组计算手工减法差值的绝对值。在代码中对这句话的调用之前已经判断过使得 arr1 永远会大于 arr2，因此这句话确实永远不会被执行到，但不会影响最终的效果或产生 bug。

如果需要覆盖这句话，需要对*精确除法模块*的内部进行进一步的分解和重构——也就是将模块拆成更小的部分，对这个函数进行单独的单元测试。在项目中，这个模块对外作为黑盒接口提供，其内部的代码逻辑的测试优先级没有那么高，因此，鉴于时间原因，我准备将这块的重构工作和单元测试的编写放到后续去做。

5. 后续

5.1 总结

单元测试的意义在于保证代码质量。对单元测试和 TDD 来说，测试的编写更像是一个预防的过程 :在真实开发之前预先通过对需求的分析拆解和测试用例的编写，减少后续可能引入的缺陷。因此，在设计时保证测试能覆盖更多的逻辑、拥有更高的覆盖率是十分重要的事情。

为了达到这个目的，我采取了两步操作：第一步是分析需求，画出流程图，将需求拆解成对应的具体流程，以确定业务可以到达的所有边界，保证对业务流程的覆盖，这部分写在 3.3.1 模块划分 中；第二步是针对业务流程中的每一个最小测试模块，通过各种方式确定其输入和输出的状态机模型，也就是测试用例，从而保证这个模块的逻辑正确，如 3.3.2 测试用例设计 所示。

测试的代码覆盖率是测试用例对业务代码的覆盖率的统计信息，从另一个层级标志着测试用例覆盖的全面性。在实际开发过程中，可以根据代码覆盖率的结果反向的查测用例有没有遗漏的部分，并完成测试用例的补充。同时，通过分析代码覆盖率中没有被覆盖到的内容，也可以发现冗余和设计不合理的代码逻辑，进而优化。

5.2 后续思考

在回答这个问题，动手实践的过程中，我有几个感觉还有提升空间，自己也还在思考和比较好奇的点：

1. 面试官留的第一个问题是如何保证单元测试尽可能的覆盖了所有的业务逻辑，

我的想法是分析需求的流程，确定其可以达到的所有边界，从而实现对业务的最大程度的覆盖。但是，与测试用例编写阶段有专门的方法、代码覆盖率有专门的指标相比，这种方式没有特定的方法论。所以，如果有一个业务更为复杂的项目，有一个统一的方法论，来保证能覆盖它所有的业务逻辑。

2. 在测试用例的设计阶段，我采用了几种方法：比如在等价类划分的过程中，我分别先对除数和被除数进行了划分随后再进行组合。这种方式造成了大量的冗余，虽然后续通过整合去掉了其中重复的部分，但是在编写的时候会发现还是存在可以去掉的情况，而且这种方式在设计分析上的代价非常高。在真实项目中，应该将有效的测试资源用先用于核心的内容的测试上，但是如何界定内容的核心程度，又如何在优先测试核心内容的基础上保证测试的全面性。这也是我想探讨和讨论的一个点。

参考

1. https://martinfowler.com/bliki/TestPyramid.html
2. https://juejin.im/post/5c3e73876fb9a049d37f5db1
3.