深度解读测试驱动开发（TDD）

本文结构：

- 什么是 TDD
- 为什么要 TDD
- 怎么 TDD
- FAQ
- 学习路径
- 延伸阅读

什么是 TDD

TDD 有广义和狭义之分，常说的是狭义的 TDD，也就是 UTDD（Unit Test Driven Development）。广义的 TDD 是 ATDD（Acceptance Test Driven Development），包括 BDD（Behavior Driven Test Development）和 Consumer-Driven Contracts Development 等。

本文所说的 TDD 指狭义上的 TDD，也就是「单元测试驱动开发」。

> TDD 是敏捷开发中的一项核心实践和技术，也是一种设计方法论。TDD的原理是在开发功能代码之前，先编写单元测试用例代码，测试代码确定需要编写什么产品代码。TDD 是 XP（Extreme Programming）的核心实践。它的主要推动者是 Kent Beck。

为什么要 TDD

传统编码方式 VS TDD 编码方式

传统编码方式

- 需求分析，想不清楚细节，管他呢，先开始写
- 发现需求细节不明确，去跟业务人员确认
- 确认好几次终于写完所有逻辑
- 运行起来测试一下，靠，果然不工作，调试
- 调试好久终于工作了
- 转测试，QA 测出 bug，debug， 打补丁

- 终于，代码可以工作了
- 一看代码烂的像坨屎，不敢动，动了还得手工测试，还得让 QA 测试，还得加班…

TDD 编码方式

- 先分解任务，分离关注点（后面有演示）
- 列 Example，用实例化需求，澄清需求细节
- 写测试，只关注需求，程序的输入输出，不关心中间过程
- 写实现，不考虑别的需求，用最简单的方式满足当前这个小需求即可
- 重构，用手法消除代码里的坏味道
- 写完，手动测试一下，基本没什么问题，有问题补个用例，修复
- 转测试，小问题，补用例，修复
- 代码整洁且用例齐全，信心满满地提交

TDD 的好处

降低开发者负担

通过明确的流程，让我们一次只关注一个点，思维负担更小。

保护网

TDD 的好处是覆盖完全的单元测试，对产品代码提供了一个保护网，让我们可以轻松地**迎接需求变化**或**改善代码的设计**。

所以如果你的项目需求稳定，一次性做完，后续没有任何改动的话，能享受到 TDD 的好处就比较少了。

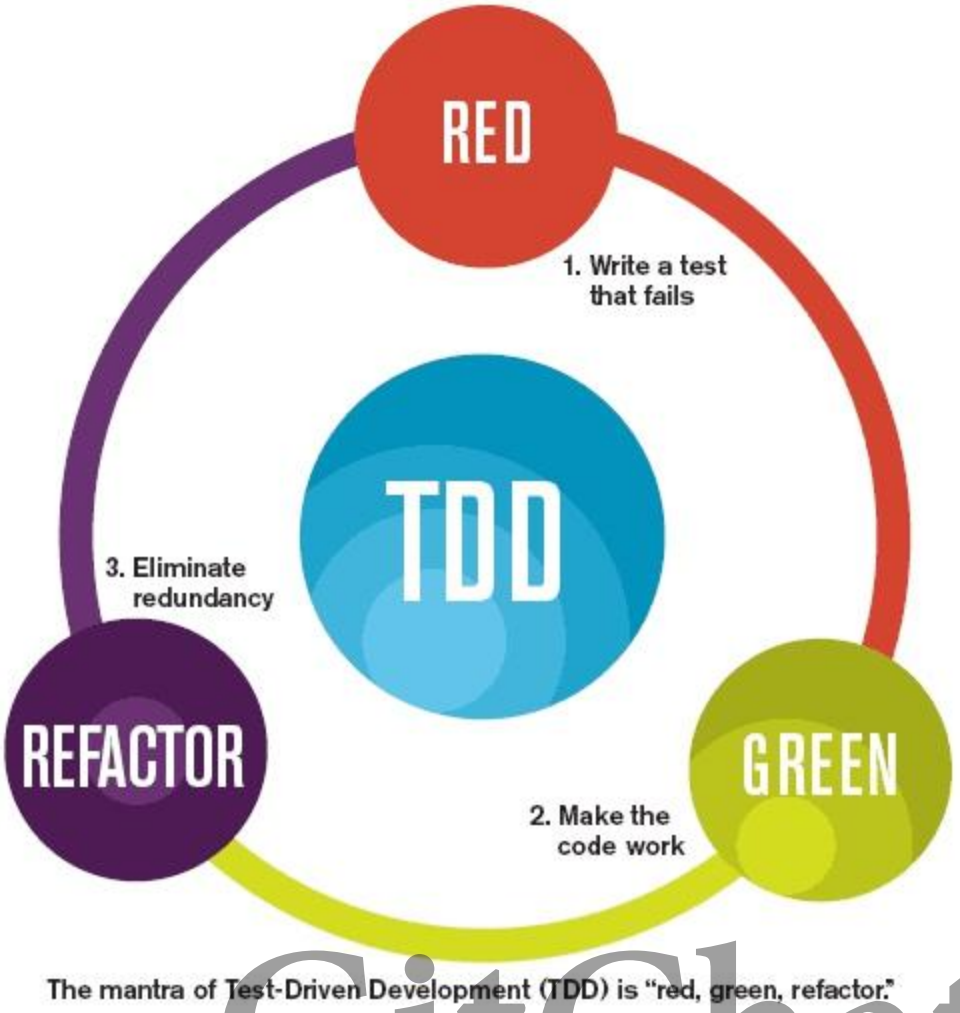
提前澄清需求

先写测试可以帮助我们去思考需求，并提前澄清需求细节，而不是代码写到一半才发现不明确的需求。

快速反馈

有很多人说 TDD 时，我的代码量增加了，所以开发效率降低了。但是，如果没有单元测试，你就要手工测试，你要花很多时间去准备数据，启动应用，跳转界面等，反馈是很慢的。准确说，快速反馈是单元测试的好处。

如何 TDD



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

TDD 的基本流程是：红，绿，重构。更详细的流程是：

- 写一个测试用例
- 运行测试
- 写刚好能让测试通过的实现
- 运行测试
- 识别坏味道，用手法修改代码
- 运行测试

你可能会问，我写一个测试用例，它明显会失败，还要运行一下吗？

是的。你可能以为测试只有成功和失败两种情况，然而，失败有无数多种，运行测试才能保证当前的失败是你期望的失败。

一切都是为了让程序符合预期，这样当出现错误的时候，就能很快定位到错误（它一定是刚刚修改的代码引起的，因为一分钟前代码还是符合我的预期的）。通过这种方式，节省了大量的调试代码的时间。

TDD 的三条规则

1. 除非是为了使一个失败的 unit test 通过，否则不允许编写任何产品代码。
2. 在一个单元测试中，只允许编写刚好能够导致失败的内容（编译错误也算失败）。

3. 只允许编写刚好能够使一个失败的 unit test 通过的产品代码。

如果违反了会怎么样呢？

违反第一条，先编写了产品代码，那这段代码是为了实现什么需求呢？怎么确保它真的实现了呢？
违反第二条，写了多个失败的测试，如果测试长时间不能通过，会增加开发者的压力，另外，测试可能会被重构，这时会增加测试的修改成本。

违反第三条，产品代码实现了超出当前测试的功能，那么这部分代码就没有测试的保护，不知道是否正确，需要手工测试。可能这是不存在的需求，那就凭空增加了代码的复杂性。如果是存在的需求，那后面的测试写出来就会直接通过，破坏了 TDD 的节奏感。

我认为它的本质是：

> 分离关注点，一次只戴一顶帽子。

在我们编程的过程中，有几个关注点：需求，实现，设计。

TDD 给了我们明确的三个步骤，每个步骤关注一个方面。

- 红：写一个失败的测试，它是对一个小需求的描述，只需要关心输入输出，这个时候根本不用关心如何实现。
- 绿：专注在用最快的方式实现当前这个小需求，不用关心其他需求，也不要管代码的质量多么惨不忍睹。
- 重构：既不用思考需求，也没有实现的压力，只需要找出代码中的坏味道，并用一个手法消除它，让代码变成整洁的代码。

注意力控制

人的注意力既可以主动控制，也会被被动吸引。注意力来回切换的话，就会消耗更多精力，思考也会不那么完整。

使用 TDD 开发，我们要主动去控制注意力，写测试的时候，发现一个类没有定义，IDE 提示编译错误，这时候你如果去创建这个类，你的注意力就不在需求上了，已经切换到了实现上，我们应该专注地写完这个测试，思考它是否表达了需求，确定无误后再开始去消除编译错误。

为什么很多人做 TDD 都做不起来？

不会合理拆分任务

TDD 之前要拆分任务，把一个大需求拆成多个小需求。也可以拆出多个函数来。

不会写测试

什么是有效的单元测试，有很多人写测试，连到底在测什么都不清楚，也可能连断言都没有，通过控制台输出，肉眼对比来验证。好的单元测试应该符合几条原则：

- 简单，只测试一个需求
- 符合 Given-When-Then 格式
- 速度快
- 包含断言
- 可以重复执行

不会写刚好的实现

很多人写实现的时候无法专注当前需求，一不小心就把其他需求也实现了，就破坏了节奏感。
实现的时候不会小步快走。

不会重构

不懂什么是 Clean Code，看不出 Smell，没有及时重构，等想要重构时已经难以下手了。不知道用合适的「手法」消除 Smell。

基础设施

对于特定技术栈，没有把单元测试基础设施搭建好，导致写测试时无法专注在测试用例上。

实例

写一个程序来计算一个文本文件 words.txt 中每个单词出现的频率。为了保持简单，假设：

- words.txt 只包含小写字母和空格
- 每个单词只包含小写字母
- 单词之间由一个或多个空格分开

举个例子，假设 words.txt 包含以下内容：

```
the day is sunny the the
the sunny is is
```

你的程序应当输出如下，按频率倒序排序：

```
the 4
is 3
sunny 2
day 1
```

请先不要往下读，思考一下你会怎么做。（思考 3 分钟...）

新手拿到这样的需求呢，就会把所有代码写到一个 main() 方法里，伪代码如下：

```
main() {
    // 读取文件
    ...
    // 分隔单词
    ...
    // 分组
    ...
    // 倒序排序
    ...
    // 拼接字符串
    ...
    // 打印
    ...
}
```

思路很清晰，但往往一口气写完，最后运行起来，输出却不符合预期，然后就开始打断点调试。

这种代码没有任何的封装。这就是为什么很多人一听到说有些公司限制一个方法不超过 10 行，就立马跳出来说，这不可能，10 行能干什么啊，我们的业务逻辑很复杂...这样的代码存在什么样的问题呢？

- 不可测试
- 不可重用
- 难以定位问题

好嘛，那我们来 TDD 嘛，你说读文件，输出控制台的测试代码要怎么写？当然，我们可以通过 Mock 和 Stub 来隔离 IO，但真的有必要吗？

有人问过 Kent Beck 这样一个问题：

> 你真的什么都会测吗？连 getter 和 setter 也会测试吗？

Kent Beck 说：公司请我来是为了实现业务价值，而不是写测试代码。所以我只在没有信心的地方写测试代码。

那对我们这个程序而言，读文件和打印到控制台都是调用系统 API，可以很有信心吧。最没有信心的是中间那写要自己写的业务逻辑。所以我们可以对程序做一些封装，《代码整洁之道》里说，有注释的地方都可以抽取方法，用方法名来代替注释：

```
main() {
    String words = read_file('words.txt')
```

```
        String[] wordArray = split(words)
        Map<String, Integer> frequency = group(wordArray)
        sort(frequency)
        String output = format(frequency)
        print(output)
    }
```

这样是不是就可以单独为 `split` , `group` , `sort` , `format` 这些方法写单元测试了呢？
当然可以， 它们的输入和输出都是很明确的嘛。

等等，你可能会说，不是测试驱动设计吗？你怎么开始做设计了？好问题！

> TDD 要不要做提前设计呢？

Kent Beck 不做提前设计，他会选一个最简单的用例，直接开写，用最简单的代码通过测试。逐渐增加测试，让代码变复杂，用重构来驱动出设计。在这个需求里，最简单的场景是什么呢？**那就是文件内容为空，输出也为空。**

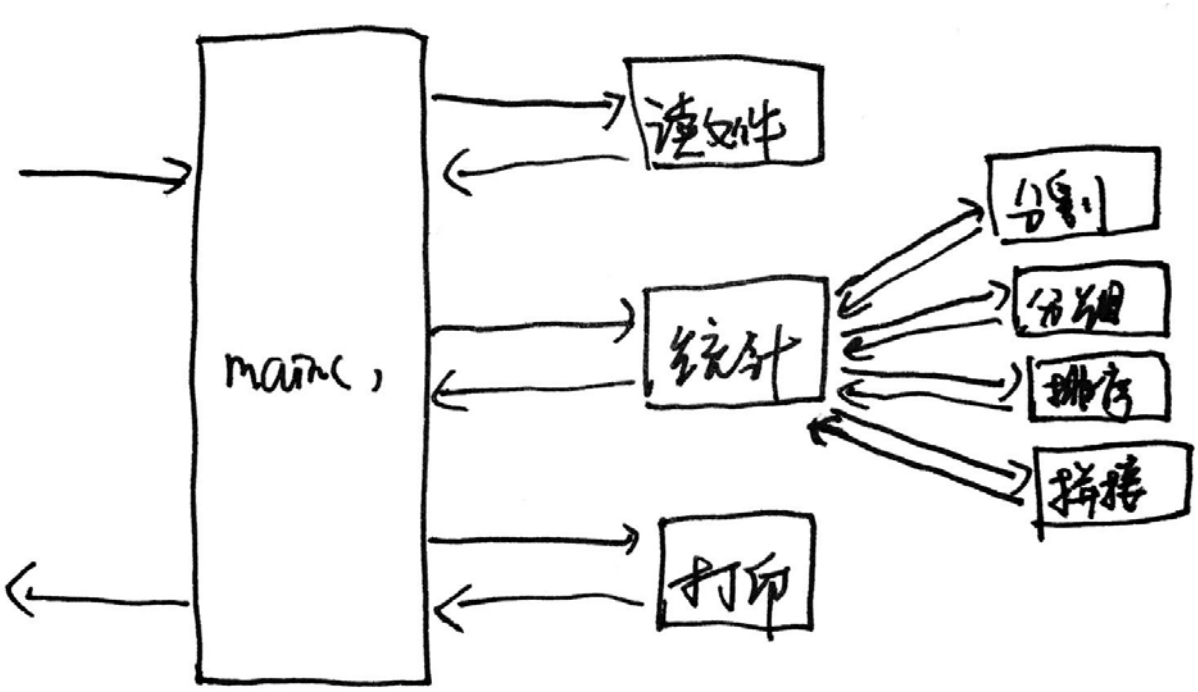
当然，对于复杂问题，可能要一边写一边补充新的用例，但对于这种简单的题目，基本可以提前就想清楚用什么用例驱动去什么产品代码。大概可以想到如下的用例：

- “” => “”
- “he” | “he 1”，一个单词，驱动出格式化字符串的代码
- “he is” | “he 1\r\nis 1”，两个不同单词，驱动出分割单词的代码
- “he he is” | “he 2\r\nis 1”，有相同单词，驱动出分组代码
- “he is is” | “is 2\r\nhe 1”，驱动出分组后的排序代码
- “he is” | “he 1\r\nis 1”，多个空格，完善分割单词的代码

Martin Fowler 的观点是，以前我们写代码要做 Big Front Up Design，在开始写代码前要设计好所有细节。而我们有了重构这个工具后，做设计的压力小了很多，因为有测试代码保护，我们可以随时重构实现了。但这并不代表我们不需要做提前设计了，提前设计可以让我们可以和他人讨论，可以先迭代几次再开始写代码，在纸上迭代总比改代码要快。

我个人比较认同 Martin Fowler 的做法，先在脑子里（当然，我脑子不够用，所以用纸画）做设计，迭代几次之后再开始写，这样，我还是会用最简单的实现通过测试，但重构时就有了方向，效率更高。

回到这个程序，我发现目前的封装不在一个抽象层次上，更理想的设计是：




```
main() {
    String words = read_file('words.txt')
    String output = word_frequency(words)
    print(output)
}

word_frequency(words) {
    String[] wordArray = split(words)
    Map<String, Integer> frequency = group(wordArray)
    sort(frequency)
    return format(frequency)
}
```


这时候，又有两种选择，有人喜欢自顶向下，有人喜欢自底向上，我个人更倾向于前者。

现在开始，只要照着 红-绿-重构 的循环去做就可以。**大部分 TDD 做不好，就是没有前面的任务分解和列 Example 的过程。**想看 TDD 过程的话，可以参考我之前做的一个直播：TDD hangman in Java或者如果需要，我也可以录一个这个题目的视频。

FAQ

为什么一定要先写测试，后补测试行不行？

> 行，但是要写完实现后，马上写测试，用测试来验证实现。如果你先手工测试，把代码都调试好了，再补单元测试，你就会觉得很鸡肋，还增加了工作量。
> 不管测试先行还是后行都可以享受到快速反馈，不过如果测试先行，你就可以享

> 受另一个好处，使用意图驱动编程减少返工。因为你的测试代码就是产品代码的客户端（调用者），你可以在测试代码里写成你理想的样子（方法名，参数，返回值等），再去实现产品代码，比起先写实现后写测试，前者返工更少。

刚写了一个测试，还没写实现。明知道运行测试一定会报错，为什么还要去运行？

> 其实测试的运行结果并非只有通过与不通过两种，因为不通过时有很多种可能。所以在明知道一定失败的情况下去运行测试，目的是看看是不是报了期望的那个错误。

小步快走确实好，但真的需要这么小步吗？

> 步子迈太大，容易扯着蛋。练习的时候需要养成小步的习惯，工作的时候可以自由切换步子的大小。当你自信的时候步子就可以大点，当你不太自信的时候就可以立即切换到小步的模式。如果只会大步，就难以再小步了。

测试代码是否会成为维护的负担？

> 维护时也遵循 TDD 流程，先修改测试代码成需求变更后的样子，让测试失败，再修改产品代码使其通过。
> 这样你就不是在维护测试用例，而是在利用测试用例。

为什么要快速实现？

> 其实是用二分查找法隔离问题，通过 hardcode 实现通过测试后，就基本确定测试是没有问题，这时再去实现产品代码，如果测试不通过，就是产品代码的问题。所以小步快走主要是为了隔离问题，也就是你可以告别 Debug 了。

为什么测试代码要很简单？

> 如果一个测试失败了，修复的时候是改测试代码而不是产品代码，那就是测试代码写的不好。
> 当测试代码足够简单时，如果一个测试失败了，就有足够信心断定一定是产品代码的问题。

什么时候不适合 TDD？

> 如果你是做探索性的技术研究（Spike），不需要长期维护，而且测试基础设施搭建成本很高，那还是手工测试吧。另外还有「可测试性极差的遗留系统」和「使用测试不友好的技术栈」的系统，做 TDD 可能得不偿失。

学习路径

1. 《有效的单元测试》
2. 《代码整洁之道》
3. 《重构》
4. Transformation Priority Premise
5. Test-Driven Development by Example
6. Growing Object-Oriented Software, Guided by Tests

延伸阅读

- 让我们再聊聊 TDD
- TDD 为什么这么难？
- TDD 专题

Chat实录：《李小波：关于TDD的真命题与伪命题》