浅谈测试驱动开发（TDD）

测试驱动开发(TDD)是极限编程的重要特点，它以不断的测试推动代码的开发，既简化了代码，又保证了软件质量。本文从开发人员使用的角度，介绍了 TDD 优势、原理、过程、原则、测试技术、Tips 等方面。

李群当前关注于网络安全产品的开发、研究；软件开发过程等方面。您可以通过 liqun@nsfocus.com和他联系。

2004 年 11 月 19 日

背景

一个高效的软件开发过程对软件开发人员来说是至关重要的，决定着开发是痛苦的挣扎，还是不断进步的喜悦。国人对软件蓝领的不屑，对繁琐冗长的传统开发过程的不耐，使大多数开发人员无所适从。最近兴起的一些软件开发过程相关的技术，提供一些比较高效、实用的软件过程开发方法。其中比较基础、关键的一个技术就是测试驱动开发（Test-Driven Development）。虽然TDD光大于极限编程，但测试驱动开发完全可以单独应用。下面就从开发人员使用的角度进行介绍，使开发人员用最少的代价尽快理解、掌握、应用这种技术。下面分优势，原理，过程，原则，测试技术，Tips等方面进行讨论。

1. 优势

TDD的基本思路就是通过测试来推动整个开发的进行。而测试驱动开发技术并不只是单纯的测试工作。

需求向来就是软件开发过程中感觉最不好明确描述、易变的东西。这里说的需求不只是指用户的需求，还包括对代码的使用需求。很多开发人员最害怕的就是后期还要修改某个类或者函数的接口进行修改或者扩展，为什么会发生这样的事情就是因为这部分代码的使用需求没有很好的描述。测试驱动开发就是通过编写测试用例，先考虑代码的使用需求（包括功能、过程、接口等），而且这个描述是无二义的，可执行验证的。

通过编写这部分代码的测试用例，对其功能的分解、使用过程、接口都进行了设计。而且这种从使用角度对代码的设计通常更符合后期开发的需求。可测试的要求，对代码的内聚性的提高和复用都非常有益。因此测试驱动开发也是一种代码设计的过程。

开发人员通常对编写文档非常厌烦，但要使用、理解别人的代码时通常又希望能有文档进行指导。而测试驱动开发过程中产生的测试用例代码就是对代码的最好的解释。

快乐工作的基础就是对自己有信心，对自己的工作成果有信心。当前很多开发人员却经常在担心：“代码是否正确？”“辛苦编写的代码还有没有严重bug？”“修改的新代码对其他部分有没有影响？”。这种担心甚至导致某些代码应该修改却不敢修改的地步。测试驱动开发提供的测试集就可以作为你信心的来源。

当然测试驱动开发最重要的功能还在于保障代码的正确性，能够迅速发现、定位bug。而迅速发现、定位

bug是很多开发人员的梦想。针对关键代码的测试集，以及不断完善的测试用例，为迅速发现、定位bug提供了条件。

我的一段功能非常复杂的代码使用TDD开发完成，真实环境应用中只发现几个bug，而且很快被定位解决。您在应用后，也一定会为那种自信的开发过程，功能不断增加、完善的感觉，迅速发现、定位bug的能力所感染，喜欢这个技术的。

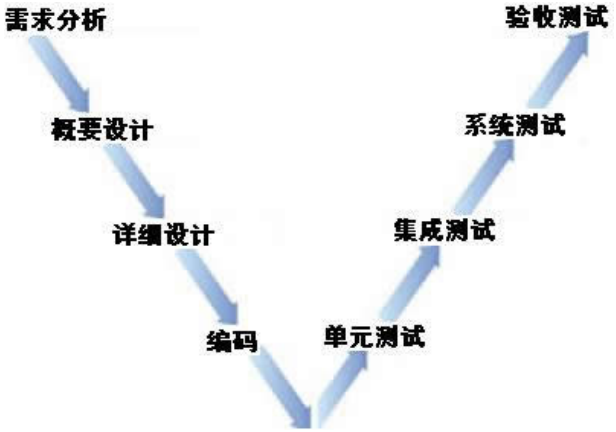
那么是什么样的原理、方法提供上面说的这些好处哪？下面我们就看看TDD的原理。

2. 原理

测试驱动开发的基本思想就是在开发功能代码之前，先编写测试代码。也就是说在明确要开发某个功能后，首先思考如何对这个功能进行测试，并完成测试代码的编写，然后编写相关的代码满足这些测试用例。然后循环进行添加其他功能，直到完全部功能的开发。

我们这里把这个技术的应用领域从代码编写扩展到整个开发过程。应该对整个开发过程的各个阶段进行测试驱动，首先思考如何对这个阶段进行测试、验证、考核，并编写相关的测试文档，然后开始下一步工作，最后再验证相关的工作。下图是一个比较流行的测试模型：V测试模型。

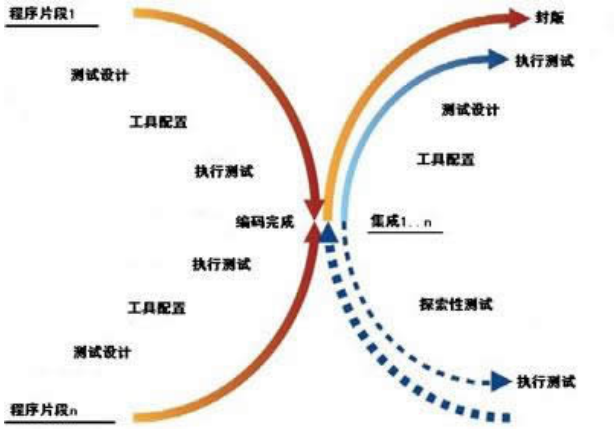
【图 V测试模型】



在开发的各个阶段，包括需求分析、概要设计、详细设计、编码过程中都应该考虑相对应的测试工作，完成相关的测试用例的设计、测试方案、测试计划的编写。这里提到的开发阶段只是举例，根据实际的开发活动进行调整。相关的测试文档也不一定是非常详细复杂的文档，或者什么形式，但应该养成测试驱动的习惯。

关于测试模型，还有X测试模型。这个测试模型，我认为，是对详细阶段和编码阶段进行建模，应该说更详细的描述了详细设计和编码阶段的开发行为。及针对某个功能进行对应的测试驱动开发。

【图 X测试模型】



基本原理应该说非常简单，那么如何进行实际操作哪，下面对开发过程进行详细的介绍。

3. 过程

软件开发其他阶段的测试驱动开发，根据测试驱动开发的思想完成对应的测试文档即可。下面针对详细设计和编码阶段进行介绍。

测试驱动开发的基本过程如下：

1） 明确当前要完成的功能。可以记录成一个 TODO 列表。

2） 快速完成针对此功能的测试用例编写。

3） 测试代码编译不通过。

4） 编写对应的功能代码。

5） 测试通过。

6） 对代码进行重构，并保证测试通过。

7） 循环完成所有功能的开发。

为了保证整个测试过程比较快捷、方便，通常可以使用测试框架组织所有的测试用例。一个免费的、优秀的测试框架是 Xunit 系列，几乎所有的语言都有对应的测试框架。我曾经写过一篇文章介绍CppUnit的文章（http://www.ibm.com/developerworks/cn/linux/l-cppunit/index.html）。

开发过程中，通常把测试代码和功能代码分开存放，这里提供一个简单的测试框架使用例子，您可以通过它了解测试框架的使用。下面是文件列表。

```
project/                        项目主目录
project/test                    测试项目主目录
project/test/testSeq.cpp        测试seq_t 的测试文件，对其他功能文件的测试文件复制后修改即可
project/test/testSeq.h
project/test/Makefile           测试项目的 Makefile
project/test/main.cpp           测试项目的主文件，不需要修改
project/main.cpp                项目的主文件
project/seq_t.h                 功能代码，被测试文件
project/Makefile                项目的 Makefile
```

主要流程基本如此，但要让你的代码很容易的进行测试，全面又不繁琐的进行测试，还是有很多测试原则和技术需要考虑。

4. 原则

测试隔离。不同代码的测试应该相互隔离。对一块代码的测试只考虑此代码的测试，不要考虑其实现细节（比如它使用了其他类的边界条件）。

一顶帽子。开发人员开发过程中要做不同的工作，比如：编写测试代码、开发功能代码、对代码重构等。做不同的事，承担不同的角色。开发人员完成对应的工作时应该保持注意力集中在当前工作上，而不要过多的考虑其他方面的细节，保证头上只有一顶帽子。避免考虑无关细节过多，无谓地增加复杂度。

测试列表。需要测试的功能点很多。应该在任何阶段想添加功能需求问题时，把相关功能点加到测试列表中，然后继续手头工作。然后不断的完成对应的测试用例、功能代码、重构。一是避免疏漏，也避免干扰当前进行的工作。

测试驱动。这个比较核心。完成某个功能，某个类，首先编写测试代码，考虑其如何使用、如何测试。然后在对其进行设计、编码。

先写断言。测试代码编写时，应该首先编写对功能代码的判断用的断言语句，然后编写相应的辅助语句。

可测试性。功能代码设计、开发时应该具有较强的可测试性。其实遵循比较好的设计原则的代码都具备较好的测试性。比如比较高的内聚性，尽量依赖于接口等。

及时重构。无论是功能代码还是测试代码，对结构不合理，重复的代码等情况，在测试通过后，及时进行重构。关于重构，我会另撰文详细分析。

小步前进。软件开发是个复杂性非常高的工作，开发过程中要考虑很多东西，包括代码的正确性、可扩展性、性能等等，很多问题都是因为复杂性太大导致的。极限编程提出了一个非常好的思路就是小步前进。把所有的规模大、复杂性高的工作，分解成小的任务来完成。对于一个类来说，一个功能一个功能的完成，如果太困难就再分解。每个功能的完成就走测试代码－功能代码－测试－重构的循环。通过分解降低整个系统开发的复杂性。这样的效果非常明显。几个小的功能代码完成后，大的功能代码几乎是不用调试就可以通过。一个个类方法的实现，很快就看到整个类很快就完成啦。本来感觉很多特性需要增加，很快就会看到没有几个啦。你甚至会为这个速度感到震惊。（我理解，是大幅度减少调试、出错的时间产生的这种速度感）

5. 测试技术

5.1. 测试范围、粒度

对哪些功能进行测试？会不会太繁琐？什么时候可以停止测试？这些问题比较常见。按大师 Kent Benk 的话，对那些你认为应该测试的代码进行测试。就是说，要相信自己的感觉，自己的经验。那些重要的功能、核心的代码就应该重点测试。感到疲劳就应该停下来休息一下。感觉没有必要更详细的测试，就停止本轮测试。

测试驱动开发强调测试并不应该是负担，而应该是帮助我们减轻工作量的方法。而对于何时停止编写测试用例，也是应该根据你的经验，功能复杂、核心功能的代码就应该编写更全面、细致的测试用例，否则测试流程即可。

测试范围没有静态的标准，同时也应该可以随着时间改变。对于开始没有编写足够的测试的功能代码，随着bug的出现，根据bug补齐相关的测试用例即可。

小步前进的原则，要求我们对大的功能块测试时，应该先分拆成更小的功能块进行测试，比如一个类A使用了类B、C，就应该编写到A使用B、C功能的测试代码前，完成对B、C的测试和开发。那么是不是每个小类或者小函数都应该测试哪？我认为没有必要。你应该运用你的经验，对那些可能出问题的地方重点测试，感觉不可能出问题的地方就等它真正出问题的时候再补测试吧。

5.2. 怎么编写测试用例

测试用例的编写就用上了传统的测试技术。

操作过程尽量模拟正常使用的过程。

全面的测试用例应该尽量做到分支覆盖，核心代码尽量做到路径覆盖。

测试数据尽量包括：真实数据、边界数据。

测试语句和测试数据应该尽量简单，容易理解。

为了避免对其他代码过多的依赖，可以实现简单的桩函数或桩类（Mock Object）。

如果内部状态非常复杂或者应该判断流程而不是状态，可以通过记录日志字符串的方式进行验证。

6. Tips

很多朋友有疑问，“测试代码的正确性如何保障？是写测试代码还是写测试文档？”这样是不是会陷入“鸡生蛋，蛋生鸡”的循环。其实是不会的。通常测试代码通常是非常简单的，通常围绕着某个情况的正确性判断的几个语句，如果太复杂，就应该继续分解啦。而传统的开发过程通常强调测试文档。但随着开发节奏的加快，用户需求的不断变化，维护高层（需求、概要设计）的测试文档可以，更低层的测试文档的成本的确太大了。而且可实时验证功能正确性的测试代码就是对代码最好的文档。

软件开发过程中，除了遵守上面提到的测试驱动开发的几个原则外，一个需要注意的问题就是，谨防过度设计。编写功能代码时应该关注于完成当前功能点，通过测试，使用最简单、直接的方式来编码。过多的考虑后期的扩展，其他功能的添加，无疑增加了过多的复杂性，容易产生问题。应该等到要添加这些特性时在进行详细的测试驱动开发。到时候，有整套测试用例做基础，通过不断重构很容易添加相关特性。

参考资料

推荐 Kent Beck 的书 Test-Driven Development: By Example ， 中文版是《测试驱动开发》 http://www.china-pub.com/computers/common/info.asp?id=14701

免费、优秀的测试框架 *unit 系列，包括：
Junit http://www.junit.org
CppUnit http://cppunit.sourceforge.net/

我曾经写的一篇介绍 CppUnit 使用的文章 《便利的开发工具CppUnit 快速使用指南》 http://www.ibm.com/developerworks/cn/linux/l-cppunit/index.shtml

介绍测试模型的文章 《This or That, V or X? 》
http://www.sdbestpractices.com/documents/s=8815/sdm0208e/
中文翻译见 《软件测试：不可忽略的阶段》
http://developer.ccidnet.com/pub/disp/Article?columnID=291&articleID=37924&pageNO=1 《软件测试：V模型，还是X模型？》
http://developer.ccidnet.com/pub/disp/Article?columnID=291&articleID=37975&pageNO=1

《Unit Test 研究报告》 http://blog.aspcool.com/tim/posts/349.aspx

本文相关的 代码。