目录

1 单元测试简介

单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证，基本属于白盒测试范畴。要根据实际情况去判定其具体含义，如 C 语言中单元指一个函数，Java 里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。

单元测试过程分 5 个步骤，说明如下：

(1) 计划单元测试—确定测试需求，制定测试策略，确定测试所需要的资源，创建测试任务的时间表等。

(2) 设计单元测试—设计单元测试模型，制定测试方案，确认并结构化测试过程。

(3) 实现单元测试—参考测试模型和测试方案，制定具体的测试用例，创建可重用的测试脚本。

(4) 执行单元测试—根据单元测试方案、用例对单元进行测试，验证测试的结果并记录测试过程中出现的缺陷。

(5) 评估单元测试—对单元测试的结果进行评估，主要从需求覆盖和代码覆盖角度进行测试完备性的评估。

单元测试是用来帮助代码重构的一种回归测试手段而不是一种需求管理手段，应该在只在需要重构的地方添加单元测试。

1.1 单元测试模型构造

1.1.1 基本概念

由于单元测试针对程序单元，而程序单元并不是一个独立可运行的程序，因此，在考虑测试模块时，同时要考虑到它和外界其他模块的联系，用一些辅助模块去模拟与被测模块关联的其他模块。这些模块分为两种：

- 驱动模块：相当于所测模块的主程序。它接收测试数据，把这些测试数据传送给被测模块，最后再输出实测结果。
- 桩模块：由被测模块调用，用以代替由被测单元所调用的模块的功能，返回适当的数据或进行适当的操作使被测单元能继续运行下去，同时还要进行一定的数据处理，如打印入口和返回等，以便检验被测模块与其下级模块的接口。

1.1.2 主要工作

驱动模块和桩模块都是额外的开销，两种都属于必须开发，但又不能和最终软件一起提交的软件。驱动模块和桩模块为程序单元的执行构成了一个完整的环境，如图下所示。

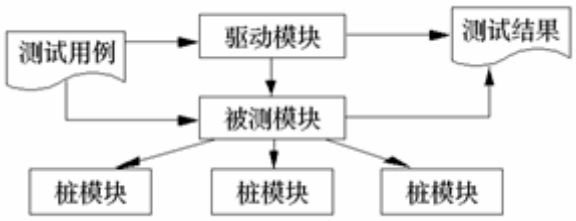

图 1.1 单元测试的测试环境

构造单元测试模型的主要工作有：

(1) 构造最小运行调度系统，即设计驱动模块。

(2) 模拟实现单元接口，即设计桩模块。

(3) 模拟生成测试数据或状态，为单元运行准备动态环境，即设计测试用例、测试结果。

测试模型除了能使被测的对象运转起来之外，还应考虑对测试过程的支持，例如对测试结果的保留，对测试覆盖率的记录等。

1.2 单元测试用例设计

1. 首先要了解单元测试内容，测试工作主要在 5 个方面对被测模块进行检查：
   (1) 模块接口测试：应该对通过所有被测模块的数据流进行测试。如果数据不能正常地输入及输出，那么其他的全部测试都说明不了问题。
   (2) 局部数据结构测试：模块的局部数据结构是最常见的错误来源。
   (3) 路径测试：检查由于计算错误、判定错误、控制流错误导致的程序错误。
   (4) 错误处理测试：是可能引发错误处理的路径及进行错误处理的路径，错误出现时错误处理程序重新安排执行路线，或通知用户处理，或干脆停止执行使程序进入一种安全等待状态。
   (5) 边界测试：单元测试中最后的任务。软件常常在边界上出错。

2. 其次要了解用例设计的方法，如下列举了设计用例的常用方法：
   (1) 规范导出法：是根据相关的规范描述来设计测试用例的。每一个测试用例用来测试一个或多个规范陈述语句。一个比较实际的方法，是根据陈述规范所用语句的顺序来相应的为被测单元设计测试用例。
   例如，考虑一个计算平方根的规范：
   - 输入：实数
   - 输出：实数
   - 规范：当输入一个 0 或比 0 大的数，返回其正的平方根；当输入一个小于 0 的数时，显示错误信息“平方根非法”，返回 0；库函数 Print_Line 可以输出错误信息。

   在这个规范中有 3 个陈述，可以用两个测试用例来对应：
   - 测试用例 1：输入 4，输出 2。对应规范中的第一句陈述（“当输入一个 0 或比 0 大的数，返回其正的平方根”）。
   - 测试用例 2：输入-1，输出 0。对应规范中的第二、三句陈述（“当输入一个小于 0 的数时，显示错误信息“平方根非法”，返回 0；库函数 Print_Line 可以输出错误信息”）。

   (2) 等价类划分：是一种正式的测试用例设计方法，它基于对被测试单元的输入、输出所做的划分，对每一个划分中的所有输入、被测单元有等价的行为。划分也可以根据软件所能存取的数据确定，包括时间、输入输出顺序、状态。
   (3) 边界值分析法、状态转移测试法、分支测试法、条件测试法、数据定义（数据流测试法）、内部边界值测试法、错误猜测法
3. 最后根据项目的实际需求、功能函数特征等，以及编写原则输出测试用例集。

1.3 执行单元测试

采用自动测试能大大提高工作效率，便于回归测试。一般来说，自动测试通过自定义的脚本文件将测试用例逐条放入，通过驱动模块读取脚本文件，驱动被测单元执行每条用例，将相应的结果返回驱动模块，驱动模块将结果保存在一个文本文件中，通过与预期的结果比较（也是一个文本文件），可以判断出是否所有的测试用例都通过测试。

2 Cmockery 使用介绍

本节是根据 cmockery 提供的 index.html 翻译而来的。而且如下的操作本人都在 VS2008 上实验过。

2.1 测试环境搭建

源码下载地址：http://code.google.com/p/cmockery/downloads/list

Window 下编译方法：

打开使用 VS2003/2005/2008 提供的命令提示窗口；

cd 到 CMockery 的目录的 window 目录

运行 nmake 命令

cmockery.lib 文件以及一些测试代码都在 Windows 目录下

示例如下：

E:

cd E:/OpenSource/c/cMockery

cd windows

nmake

2.2 使用说明

Cmockery 单元测试用例就是函数，命名为 **void function(void **state)** .

Cmockery 测试程序将（多个）测试用例的函数指针使用 **unit_test*()** 宏初始化到一个表中，这个表会传给 **run_tests()**从而执行测试用例。 **run_tests()**将适当的异常/信号句柄，以及其他数据结构的指针装入到测试函数。当单元测试结束时，**run_tests()** 会显示出各种定义的测试是否成功。

使用示例：

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>

//一个测试用例，空实现
void null_test_success(void **state)
{
}

int main(int argc, char* argv[])
{
    const UnitTest tests[] =
    {
        unit_test(null_test_success),
    };

    return run_tests(tests);
}
```

注：http://see-see.appspot.com/?p=12001，对该测试框架内容进行了翻译。

2.3 使用例子

1. 按 2.1 小节编译产生 cmockery.lib。
2. 新建一个项目：math_test，此项目中有 2 个文件：
   - math.c —待测代码模块，只有功能加法和减法;
   - math_test.c —包含测试用例和 main 函数。
   - 源码参见附录 1
3. 加入 cmockery.h 以及 cmockery.lib 文件，编译运行后输出：

```
test_add: Starting test
test_add: Test completed successfully.
test_sub: Starting test
test_sub: Test completed successfully.
All 2 tests passed
```

注：以下是框架提供的测试 demo，移植到 VS2008 里运行，可测试。

3 Cmockery 测试框架评估

3.1 评估标准

虽然该框架是轻量级的，但是不管是 google 自己还是百度的结果，对该框架的使用是比较冷门的。如果以前没有过单元测试的经验，想要在没有详细技术支持的测试框架上搭建自动化测试是很困难的。

从 cmockery 提供的测试方法看，将所需测试的模块通过打桩的方法实施单元测试，维护重复桩代码很难让团队做到真心拥抱单元测试。因为单元测试代码也是代码，也需要维护，而代码维护的代价比文档大得多。

3.2 评估结论

如下引用一些专家对使用免费 C/C++单元测试框架做单元测试的评价：其结果是费钱、费力、误事。

费钱：工具可以免费，人才不能免费，相反，人才极昂贵，而且越来越最昂贵。

费力：程序员为什么不愿“写”单元测试？请注意，是“写”单元测试，不是“做”单元测试。程序员不是不知道单元测试的好，而是不愿意写测试代码，因为太烦了。程序员不怕难，最怕烦。创造往往很难，但却是程序员的最爱，也是程序员的价值所在。测试代码既然用工具都可以生成，自然没有创造性，程序员怎么可能喜欢写呢？让程序员做自己很不喜欢做的事，这是很累人的，特别费力。

误事：误事表现在三方面：

一、花很多时间来写测试代码，延长了项目的周期，严重时可能失去市场机会。

二、太费力，程序员可能消极应对，最终做不下去，折腾很长时间，又回到原点。

三、测试不充分，很多测试点，是手工编写代码难以达到的，举个简单的例子，要让 malloc()在第一个用例返回 NULL，其他用例正常申请内存，手工编写代码就很难做到。用开源框架，完成代码覆盖都几乎不可能，而使用合适的商业工具，完成 MCDC 覆盖一点也不困难。

附录 1 例子源码

math.c:

```c
int add(int a, int b)
{
    return a + b;
}

int sub(int a, int b)
{
    return a - b;
}
```

math_test.c:

```c
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>

extern int add(int a, int b);
extern int sub(int a, int b);

/* Ensure add() adds two integers correctly. */
void test_add(void **state)
{
    assert_int_equal(add(3, 3), 6);
    assert_int_equal(add(3, -3), 0);
}

/* Ensure sub() subtracts two integers correctly.*/
void test_sub(void **state)
{
    assert_int_equal(sub(3, 3), 0);
    assert_int_equal(sub(3, -3), 6);
}

int main(int argc, char *argv[])
{
    const UnitTest tests[] =
    {
        unit_test(test_add),
        unit_test(test_sub),
    };

    return run_tests(tests);
}
```