**深入理解gtest C/C++单元测试经验谈**

2011-08-22 13:57 杨玚 51CTO 字号：[**T**](javascript:setfont(12);) | [**T**](javascript:setfont(16);)

[一键收藏，随时查看，分享好友！](javascript:favorBox('open');)

本文基于笔者的实际开发经验，言简意赅地讲解了C/C++单元测试框架gtest的主要使用方法和注意事项，并设计了若干可编译的精简示例，给出了运行效果图。既可以用作gtest的入门教程，也适合作为工作中的快速参考。

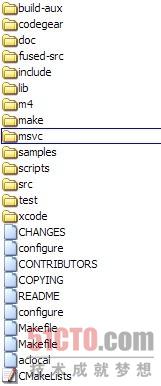
AD：[51CTO 网+ 第十二期沙龙：大话数据之美\_如何用数据驱动用户体验](http://mobile.51cto.com/mobile/mdsa12/)

Google C++ Testing Framework（简称gtest，http://code.google.com/p/googletest/）是Google公司发布的一个开源C/C++单元测试框架，已被应用于多个开源项目及Google内部项目中，知名的例子包括Chrome Web浏览器、LLVM编译器架构、Protocol Buffers数据交换格式及工具等。

优秀的C/C++单元测试框架并不算少，相比之下gtest仍具有明显优势。与CppUnit比，gtest需要使用的头文件和函数宏更集中，并支持测试用例的自动注册。与CxxUnit比，gtest不要求Python等外部工具的存在。与Boost.Test比，gtest更简洁容易上手，实用性也并不逊色。Wikipedia给出了各种编程语言的单元测试框架列表（http://en.wikipedia.org/wiki/List\_of\_unit\_testing\_frameworks）。

**一、基本用法**

gtest当前的版本是1.5.0，如果使用Visual C++编译，要求编译器版本不低于7.1（Visual C++ 2003）。如下图所示，它的msvc文件夹包含Visual C++工程和项目文件，samples文件夹包含10个使用范例。

[](http://images.51cto.com/files/uploadimg/20110822/140242979.jpg)

一般情况下，我们的单元测试代码只需要包含头文件*gtest.h*。gtest中常用的所有结构体、类、函数、常量等，都通过命名空间testing访问，不过gtest已经把最简单常用的单元测试功能包装成了一些带参数宏，因此在简单的测试中常常可以忽略命名空间的存在。

按照gtest的叫法，宏TEST为特定的测试用例（Test Case）定义了一个可执行的测试（Test）。它接受用户指定的测试用例名（一般取被测对象名）和测试名作为参数，并划出了一个作用域供填充测试宏语句和普通的C++代码。一系列TEST的集合就构成一个简单的测试程序。

常用的测试宏如下表所示。以ASSERT\_开头和以EXPECT\_开头的宏的区别是，前者在测试失败时会给出报告并立即终止测试程序，后者在报告后继续执行测试程序。

| **ASSERT宏** | **EXPECT宏** | **功能** |
| --- | --- | --- |
| ASSERT\_TRUE | EXPECT\_TRUE | 判真 |
| ASSERT\_FALSE | EXPECT\_FALSE | 判假 |
| ASSERT\_EQ | EXPECT\_EQ | 相等 |
| ASSERT\_NE | EXPECT\_NE | 不等 |
| ASSERT\_GT | EXPECT\_GT | 大于 |
| ASSERT\_LT | EXPECT\_LT | 小于 |
| ASSERT\_GE | EXPECT\_GE | 大于或等于 |
| ASSERT\_LE | EXPECT\_LE | 小于或等于 |
| ASSERT\_FLOAT\_EQ | EXPECT\_FLOAT\_EQ | 单精度浮点值相等 |
| ASSERT\_DOUBLE\_EQ | EXPECT\_DOUBLE\_EQ | 双精度浮点值相等 |
| ASSERT\_NEAR | EXPECT\_NEAR | 浮点值接近（第3个参数为误差阈值） |
| ASSERT\_STREQ | EXPECT\_STREQ | C字符串相等 |
| ASSERT\_STRNE | EXPECT\_STRNE | C字符串不等 |
| ASSERT\_STRCASEEQ | EXPECT\_STRCASEEQ | C字符串相等（忽略大小写） |
| ASSERT\_STRCASENE | EXPECT\_STRCASENE | C字符串不等（忽略大小写） |
| ASSERT\_PRED1 | EXPECT\_PRED1 | 自定义谓词函数，(pred, arg1)（还有\_PRED2, ..., \_PRED5） |

写个简单的测试试一下。假设我们实现了一个加法函数：

1. // add.h
2. #pragma once
3. **inline** **int** Add(**int** i, **int** j) { **return** i+j; }

对应的单元测试程序可以这样写：

1. // add\_unittest.cpp
2. #include "add.h"
3. #include <gtest/gtest.h>
5. TEST(Add, 负数) {
6. EXPECT\_EQ(Add(-1,-2), -3);
7. EXPECT\_GT(Add(-4,-5), -6); // 故意的
8. }
10. TEST(Add, 正数) {
11. EXPECT\_EQ(Add(1,2), 3);
12. EXPECT\_GT(Add(4,5), 6);
13. }

代码中，测试用例Add包含两个测试，正数和负数（这里利用了Visual C++ 2005以上允许标识符包含Unicode字符的特性）。编译运行效果如下：

在控制台界面中，通过的测试用绿色表示，失败的测试用红色表示。双横线分隔了不同的测试用例，其中包含的每个测试的启动与结果用单横线和RUN ... OK或RUN ... FAILED标出。失败的测试会打印出代码行和原因，测试程序最后为所有用例和测试显示统计结果。建议读者试一下换成ASSERT\_宏的不同之处。

每个测试宏还可以使用<<运算符在测试失败时输出自定义信息，如：

1. ASSERT\_EQ(M[i], N[j]) << "i = " << i << ", j = " << j;

编译命令行中，gtest\_mt.lib和gtest\_main\_mt.lib就是前面使用VC项目文件生成的静态库。有意思的是，测试代码不需要注册测试用例，也不需要定义main函数，这是gtest通过后一个静态库自动完成的，它的实现代码如下：

1. // gtest-main.cc
2. **int** main(**int** argc, **char** \*\*argv) {
3. std::cout << "Running main() from gtest\_main.cc\n";
4. testing::InitGoogleTest(&argc, argv);
5. **return** RUN\_ALL\_TESTS();
6. }

其中，函数InitGoogleTest负责注册需要运行的所有测试用例，宏RUN\_ALL\_TEST负责执行所有测试，如果全部成功则返回0，否则返回1。当然，我们也可以仅链接gtest\_mt.lib，自己提供main函数。

**二、测试固件**

很多时候，我们想在不同的测试执行前创建相同的配置环境，在测试执行结束后执行相应的清理工作，测试固件（Test Fixture）为这种需求提供了方便。“Fixture”是一个汉语中不易直接对应的词，《美国传统词典》对它的解释是“（作为附属物的）固定装置；被固定的状态”。在单元测试中，Fixture的作用是为测试创建辅助性的上下文环境，实现测试的初始化和终结与测试过程本身的分离，便于不同测试使用相同代码来搭建固定的配置环境。用体操比赛的说法，测试过程体现了特定测试的自选动作，测试固件则体现了对一系列测试（在开始和结束时）的规定动作。有些讲单元测试的书籍直接把测试固件称为Scaffolding（脚手架）。

使用测试固件比单纯调用TEST宏稍微麻烦一些：

1.         从gtest的testing::Test类派生一个类，用public或protected定义以下所有成员。

2.         （可选）建立环境：使用默认构造函数，或定义一个虚成员函数virtual void SetUp()。

3.         （可选）销毁环境：使用析构函数，或定义一个虚成员函数virtual void TearDown()。

4.         用TEST\_F定义测试，写法与TEST相同，但测试用例名必须为上面定义的类名。

每个带固件的测试的执行顺序是：

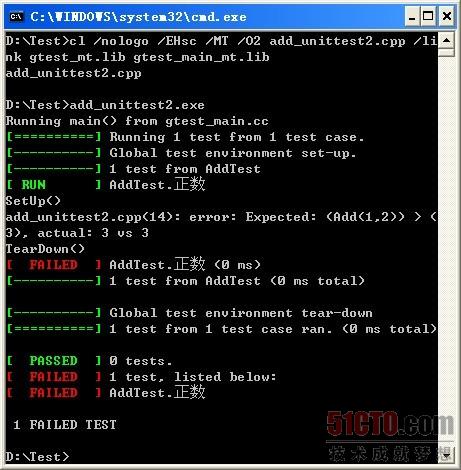
1.         调用默认构造函数创建一个新的带固件对象。

2.         立即调用SetUp函数。

3.         运行TEST\_F体。

4.         立即调用TearDown函数。

5.         调用析构函数销毁类对象。

[](http://images.51cto.com/files/uploadimg/20110822/140906927.jpg)

从gtest的实现代码可以看到，TEST\_F又从用户定义的类自动派生了一个类，因此要求public或protected的访问权限；大括号里的内容被扩展成一个名为TestBody的虚成员函数的函数体，因此可以在其中直接访问成员变量和成员函数。其实TEST也采用了相同的实现机制，只是它直接从gtest的testing::Test自动派生类，所以可以指定任意用例名。testing::Test类的SetUp和TearDown都是空函数，所以它只执行测试步骤，没有环境的创建和销毁。

借用上面Add函数写个固件测试的例子：

1. // add\_unittest2.cpp
2. #include "add.h"
3. #include <stdio.h>
4. #include <gtest/gtest.h>
6. **class** AddTest: **public** testing::Test
7. {
8. **public**:
9. **virtual** **void** SetUp()    { puts("SetUp()"); }
10. **virtual** **void** TearDown() { puts("TearDown()"); }
11. };
13. TEST\_F(AddTest, 正数) {
14. ASSERT\_GT(Add(1,2), 3); // 故意的
15. ASSERT\_EQ(Add(4,5), 6); // 也是故意的
16. }

编译运行效果如下：

必须强调，每个TEST\_F开始都创建了一个新的带固件对象，因此每个测试都使用独立的完全相同的初始环境，各测试可以按任意顺序执行（参见--gtest\_shuffle命令行选项）。但在某些情况下，我们可能需要在各个测试间共享一个相同的环境来保存和传递状态，或者环境的状态是只读的，可以只初始化一次，再或者创建环境的过程开销很高，要求只初始化一次。共享某个固件环境的所有测试合称为一个“测试套件”（Test Suite），gtest中利用静态成员变量和静态成员函数实现这个概念：

1.         （可选）在testing::Test的派生类中，定义若干静态成员变量来维护套件的状态。

2.         （可选）建立共享环境：定义一个静态成员函数static void SetUpTestCase()。

3.         （可选）销毁共享环境：定义一个静态成员函数static void TearDownCase()。

另外，还可以使用gtest的Environment类来建立和销毁所有测试共用的全局环境（对应于上图显示的“Global test environment set-up”和“Global test environment tear-down”）：

1. **class** Environment {
2. **public**:
3. **virtual** ~Environment() {}
4. **virtual** **void** SetUp() {}
5. **virtual** **void** TearDown() {}
6. };

gtest文档建议测试程序自己定义main函数并在其中创建和注册全局环境对象：

1. Environment\* AddGlobalTestEnvironment(Environment\* env);

**三、异常测试**

C程序中要返回出错信息，可以利用特定的函数返回值、函数的输出（outbound）参数、或者设置全局变量（如C标准库定义的errno，Windows API中的“上次错误”（last error）代码，Winsock中与每个socket相关联的错误代码）。C++程序常用异常（exception）来返回出错信息，gtest为异常测试提供了专用的测试宏：

| **ASSERT宏** | **EXPECT宏** | **功能** |
| --- | --- | --- |
| ASSERT\_NO\_THROW | EXPECT\_NO\_THROW | 不抛出异常，参数为(statement) |
| ASSERT\_ANY\_THROW | EXPECT\_ANY\_THROW | 抛出异常，参数为(statement) |
| ASSERT\_THROW | EXPECT\_THROW | 抛出特定类型的异常，参数为(statement, type) |

需要注意，这些测试宏都接受C/C++语句作为参数，所以既可以像前面那样传递表达式，也可以传递用大括号包起来的代码块。

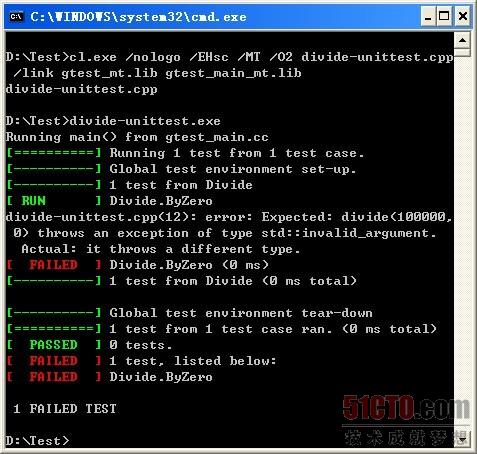
借助下面的被测函数：

1. // divide.h
2. #pragma once
3. #include <stdexcept>
5. **int** divide(**int** dividend, **int** divisor) {
6. **if**(!divisor) {
7. **throw** std::length\_error("can't be divided by 0"); // 故意的
8. }
9. **return** dividend / divisor;
10. }

测试程序如下：

1. // divide-unittest.cpp
2. #include <gtest/gtest.h>
3. #include "./divide.h"
5. TEST(Divide, ByZero) {
6. EXPECT\_NO\_THROW(divide(-1, 2));
8. EXPECT\_ANY\_THROW({
9. **int** k = 0;
10. divide(k, k);
11. });
13. EXPECT\_THROW(divide(100000, 0), std::invalid\_argument);
14. }

编译运行效果如下

[](http://images.51cto.com/files/uploadimg/20110822/141053243.jpg)

容易想到，gtest的这些异常测试宏是用C++的try ... catch语句来实现的：

1. **try** {
2. statement;
3. }
4. **catch**(type **const**&) {
5. // throw
6. }
7. **catch**(...) {
8. // any throw
9. }
10. // no throw

如果把上图中Visual C++的编译选项/EHsc换成/EHa，try ... catch就可以同时支持C++风格的异常和Windows系统的结构化异常（SEH）。这样，即使删掉divide函数里的if判断，测试代码的EXPECT\_ANY\_THROW宏也会成功捕获异常。

遗憾的是，目前仅使用这些测试宏无法得到获得被抛出异常的详细信息（如divide函数中的报错文本），这和gtest自身不愿意使用C++异常有关。

**四、值参数化测试**

有些时候，我们需要对代码实现的功能使用不同的参数进行测试，比如使用大量随机值来检验算法实现的正确性，或者比较同一个接口的不同实现之间的差别。gtest把“集中输入测试参数”的需求抽象出来提供支持，称为值参数化测试（Value Parameterized Test）。

值参数化测试包括4个步骤：

1.         从gtest的TestWithParam模板类派生一个类（记为C），模板参数为需要输入的测试参数的类型。由于TestWithParam本身是从Test派生的，所以C就成了一个测试固件类。

2.         在C中，可以实现诸如SetUp、TearDown等方法。特别地，测试参数由TestWithParam实现的GetParam()方法依次返回。

3.         使用TEST\_P（而不是TEST\_F）定义测试。

4.         使用INSTANTIATE\_TEST\_CASE\_P宏集中输入测试参数，它接受3个参数：任意的文本前缀，测试类名（这里即为C），以及测试参数值序列。gtest框架依次使用这些参数值生成测试固件类实例，并执行用户定义的测试。

gtest提供了专门的模板函数来生成参数值序列，如下表所示：

| **参数值序列生成函数** | **含义** |
| --- | --- |
| Bool() | 生成序列{false, true} |
| Range(begin, end[, step]) | 生成序列{begin, begin+step, begin+2\*step, ...} (不含end)，step默认为1 |
| Values(v1, v2, ..., vN) | 生成序列{v1, v2, ..., vN} |
| ValuesIn(container), ValuesIn(iter1, iter2) | 枚举STL container，或枚举迭代器范围[iter1, iter2) |
| Combine(g1, g2, ..., gN) | 生成g1, g2, ..., gN的笛卡尔积，其中g1, g2, ..., gN均为参数值序列生成函数（要求C++0x的*<tr1/tuple>*） |

写个小程序试一下。假设我们实现了一种快速累加算法，希望使用另一种直观算法进行正确性校验。算法实现和测试代码如下

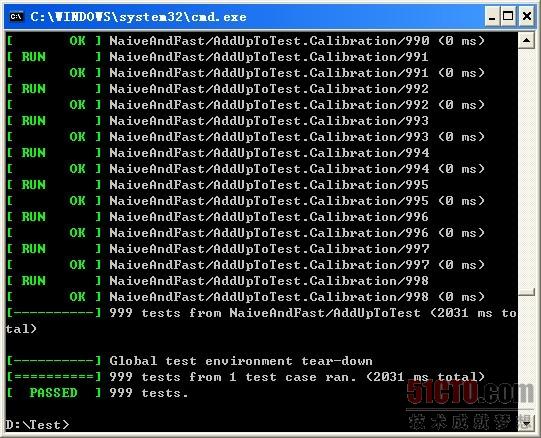
1. // addupto.h
3. #pragma once
5. **inline** unsigned NaiveAddUpTo(unsigned n) {
6. unsigned sum = 0;
7. **for**(unsigned i = 1; i <= n; ++i) sum += i;
8. **return** sum;
9. }
11. **inline** unsigned FastAddUpTo(unsigned n) {
12. **return** n\*(n+1)/2;
13. }

测试程序如下：

1. // addupto\_test.cpp
3. #include <gtest/gtest.h>
4. #include "addupto.h"
6. **class** AddUpToTest : **public** testing::TestWithParam<unsigned>
7. {
8. **public**:
9. AddUpToTest() { n\_ = GetParam(); }
10. **protected**:
11. unsigned n\_;
12. };
14. TEST\_P(AddUpToTest, Calibration) {
15. EXPECT\_EQ(NaiveAddUpTo(n\_), FastAddUpTo(n\_));
16. }
18. INSTANTIATE\_TEST\_CASE\_P(
19. NaiveAndFast, // prefix
20. AddUpToTest,   // test case name
21. testing::Range(1u, 1000u) // parameters
22. );

注意TestWithParam的模板参数设置为unsigned类型，而在代码倒数第2行，两个常量值都加了u后缀来指定为unsigned类型。熟悉C++的读者应该知道，模板函数在进行类型推断（deduction）时匹配相当严格，不像普通函数那样允许类型提升（promotion）。如果上面省略u后缀，就会造成编译错误。当然还可以显式指定模板参数：testing::Range<unsigned>(1, 1000)。

运行效果如下，这里省略了开头的大部分输出（命令行窗口设置的缓冲区高度为3000行）。

[](http://images.51cto.com/files/uploadimg/20110822/141233681.jpg)

作者简介

杨玚，1980年生，2009年毕业于中国科学技术大学，获博士学位。2009年8月加入中国软件评测中心重大专项测试部，任开发测试工程师，负责“软件测试能力优化升级” 项目工具研发。关注领域为网络信息安全。