

转

深入理解gtest : C/C++单元测试经验谈

2013年08月05日 20:41:03

dctfjy

阅读数 : 918

标签 :

单元测试

更多

深入理解gtest : C/C++单元测试经验谈

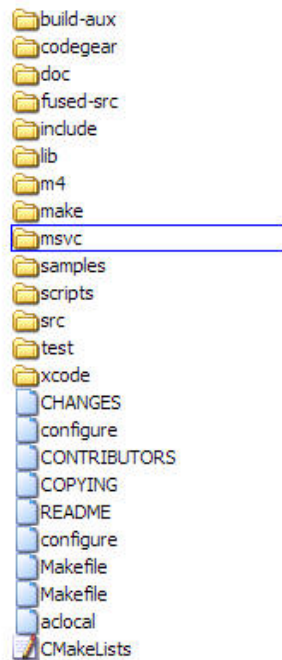
发布于2012-2-13

GoogleC++TestingFramework ( 简称gtest , [http : //code. google. com/p/googletest/](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ChromeWeb浏览器、LLVM编译器架构、ProtocolBuffers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\\_ uni t\\_ testing\\_ frameworks](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VisualC++编译 , 要求编译器版本不低于7. 1 ( VisualC++2003 ) 。如下图所示 , 它的msvc文件夹包含VisualC++工程和项目文件 , samples文件夹包含10个使用范例。



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（TestCase）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：

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

对应的单元测试程序可以这样写：

```

// add_unittest.cpp
#include "add.h"
#include <gtest/gtest.h>

TEST(Add, 负数) {
    EXPECT_EQ(Add(-1,-2), -3);
    EXPECT_GT(Add(-4,-5), -6); // 故意的
}

TEST(Add, 正数) {
    EXPECT_EQ(Add(1,2), 3);
    EXPECT_GT(Add(4,5), 6);
}

```

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

```

C:\WINDOWS\system32\cmd.exe
D:\Test>cl /nologo /EHsc /MT /O2 add_unittest.cpp /link
gtest_mt.lib gtest_main_mt.lib
add_unittest.cpp

D:\Test>add_unittest.exe
Running main() from gtest_main.cc
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from Add
[ RUN      ] Add.负数
add_unittest.cpp(7): error: Expected: <Add(-4,-5)> > <
-6>, actual: -9 vs -6
[ FAILED   ] Add.负数 <0 ms>
[ RUN      ] Add.正数
[ OK       ] Add.正数 <0 ms>
[-----] 2 tests from Add <0 ms total>

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. <0 ms total>
>
[ PASSED   ] 1 test.
[ FAILED   ] 1 test, listed below:
[ FAILED   ] Add.负数

1 FAILED TEST

D:\Test>

```

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：

```
| ASSERT_EQ(M[i], N[j]) << "i = " << i << ", j = " << j;
```

编译命令中, 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:

```
/* gtest-main.cc  
int main(int argc, char **argv) {  
    std::cout << "Running main() from gtest_main.cc\n";  
    testing::InitGoogleTest(&argc, argv);  
    return RUN_ALL_TESTS();  
}
```

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 (TestFixture) 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virtualvoidSetUp()。
- 3、(可选) 销毁环境: 使用析构函数, 或定义一个虚成员函数virtualvoidTearDown()。
- 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。

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:

```
// add_unittest2.cpp
#include "add.h"
#include <stdio.h>
#include <gtest/gtest.h>

class AddTest: public testing::Test
{
public:
    virtual void SetUp() { puts("SetUp()"); }
    virtual void TearDown() { puts("TearDown()"); }
};

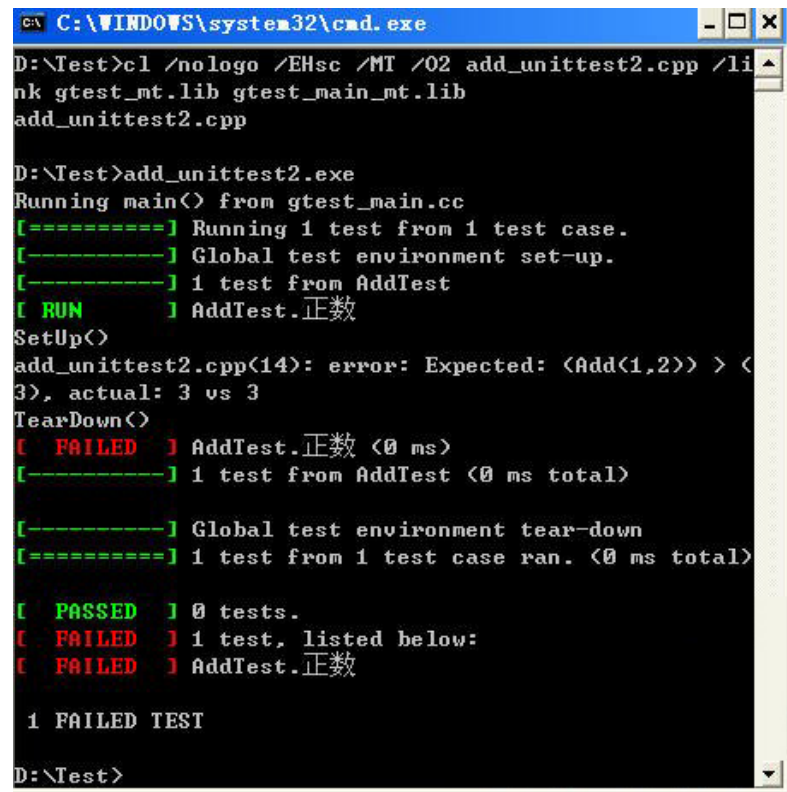
TEST_F(AddTest, 正数) {
    ASSERT_GT(Add(1, 2), 3); // 故意的
    ASSERT_EQ(Add(4, 5), 6); // 也是故意的
}
```

#### 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，又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”（TestSuite），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staticvoidSetUpTestCase（）。
- 3、（可选）销毁共享环境：定义一个静态成员函数staticvoidTearDownCase（）。





```

C:\WINDOWS\system32\cmd.exe
D:\Test>cl /nologo /EHsc /MT /O2 add_unittest2.cpp /link gtest_mt.lib gtest_main_mt.lib add_unittest2.cpp

D:\Test>add_unittest2.exe
Running main() from gtest_main.cc
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from AddTest
[ RUN      ] AddTest.正数
add_unittest2.cpp(14): error: Expected: <Add(1,2)> > <3>, actual: 3 vs 3
TearDown()
[ FAILED   ] AddTest.正数 <0 ms>
[-----] 1 test from AddTest <0 ms total>

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. <0 ms total>

[ PASSED   ] 0 tests.
[ FAILED   ] 1 test, listed below:
[ FAILED   ] AddTest.正数

1 FAILED TEST

D:\Test>

```

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“Globaltestenvironmentset-up”和“Globaltestenvironmenttear-down”）：

```

class Environment {
public:
    virtual ~Environment() {}
    virtual void SetUp() {}
    virtual void TearDown() {}
};

```

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：

```

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
<code>ASSERT_NO_THROW</code>	<code>EXPECT_NO_THROW</code>	不抛出异常, 参数为 <code>(statement)</code>
<code>ASSERT_ANY_THROW</code>	<code>EXPECT_ANY_THROW</code>	抛出异常, 参数为 <code>(statement)</code>
<code>ASSERT_THROW</code>	<code>EXPECT_THROW</code>	抛出特定类型的异常, 参数为 <code>(statement, type)</code>

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:

```
// divide.h
#pragma once
#include <stdexcept>

int divide(int dividend, int divisor) {
    if(!divisor) {
        throw std::length_error("can't be divided by 0"); // 故意的
    }
    return dividend / divisor;
}
```

测试程序如下:

```
// divide-unittest.cpp
#include <gtest/gtest.h>
#include "divide.h"

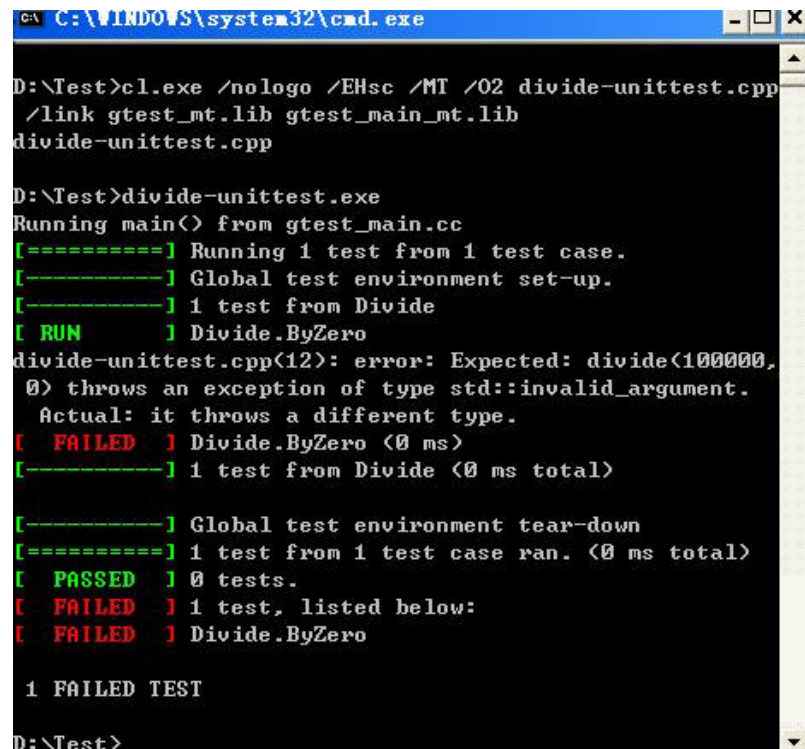
TEST(Divide, ByZero) {
    EXPECT_NO_THROW(divide(-1, 2));

    EXPECT_ANY_THROW({
        int k = 0;
        divide(k, k);
    });

    EXPECT_THROW(divide(100000, 0), std::invalid_argument);
}
```

编译运行效果如下:





```

C:\WINDOWS\system32\cmd.exe

D:\Test>cl.exe /nologo /EHsc /MT /O2 divide-unittest.cpp
/link gtest_mt.lib gtest_main_mt.lib
divide-unittest.cpp

D:\Test>divide-unittest.exe
Running main() from gtest_main.cc
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from Divide
[ RUN      ] Divide.ByZero
divide-unittest.cpp(12): error: Expected: divide(100000,
0) throws an exception of type std::invalid_argument.
Actual: it throws a different type.
[ FAILED   ] Divide.ByZero <0 ms>
[-----] 1 test from Divide <0 ms total>

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. <0 ms total>
[ PASSED   ] 0 tests.
[ FAILED   ] 1 test, listed below:
[ FAILED   ] Divide.ByZero

1 FAILED TEST

D:\Test>

```

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：

```

try {
    statement;
}
catch(type const&) {
    // throw
}
catch(...) {
    // any throw
}
// 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INSTITUTE\_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
<code>Bool()</code>	生成序列{false, true}
<code>Range(begin, end[, step])</code>	生成序列{begin, begin+step, begin+2*step, ...} (不含 end), step 默认为 1
<code>Values(v1, v2, ..., vN)</code>	生成序列{v1, v2, ..., vN}
<code>ValuesIn(container), ValuesIn(iterator1, iterator2)</code>	枚举 STL container, 或枚举迭代器范围[iterator1, iterator2)
<code>Combine(g1, g2, ..., gN)</code>	生成 g1, g2, ..., gN 的笛卡儿积, 其中 g1, g2, ..., gN 成为参数值序列生成函数 (要求 C++11 编译器支持)

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：

```
// addupto.h

#pragma once

inline unsigned NaiveAddUpTo(unsigned n) {
    unsigned sum = 0;
    for(unsigned i = 1; i <= n; ++i) sum += i;
    return sum;
}

inline unsigned FastAddUpTo(unsigned n) {
    return n*(n+1)/2;
}
```

测试程序如下：

```
// addupto_test.cpp

#include <gtest/gtest.h>
#include "addupto.h"

class AddUpToTest : public testing::TestWithParam<unsigned>
{
public:
    AddUpToTest() { n_ = GetParam(); }
protected:
    unsigned n_;
};

TEST_P(AddUpToTest, Calibration) {
    EXPECT_EQ(NaiveAddUpTo(n_), FastAddUpTo(n_));
}

INSTANTIATE_TEST_CASE_P(
    NaiveAndFast, // prefix
    AddUpToTest, // test case name
    testing::Range(1u, 1000u) // parameters
);
```

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(1, 1000)。



## 常用数据采集卡

数据采集卡主要类型及技术参数

想对作者说点什么？

我来说两句