

Googletest

Google C++ Testing Framework

Google C++ 测试框架基于著名的 xUnit 架构，帮助你更好地编写 C++ 测试。无论是在 Linux, Windows, 还是 Mac 环境下工作，只要你编写 C++ 代码，Google 测试框架都可以帮上忙。

本文主要介绍 google test 在 Linux 下的应用，结合例子尝试对其进行深入解析。本文主要借鉴《[玩转 Google 开源 C++ 单元测试框架 Google Test 系列](#)》。

1. 初识 gtest

1.1 下载

官网：<http://code.google.com/p/googletest/>。目前最新为 1.6.0 版。

1.2 编译

将 zip 包解压后进入该文件夹主目录，按如下方法编译：（若更新后此方法不再适用，则参见 zip 包中的 README 说明）

```
g++ -I./include -I./ -c ./src/gtest-all.cc
```

```
ar -rv libgtest.a gtest-all.o
```

成功后将产生 libgtest.a 这个库函数，是将来需要用到的。

1.3 简易 Demo

在该主目录新建 mytest.cpp，编译及运行结果如下。

```
g++ -I./include mytest.cpp libgtest.a -lpthread -o mytest
```

```
#include "gtest/gtest.h"

int Foo(int a) {
    return a*2;
}

TEST(FooTest, NAME) {
    EXPECT_EQ(2, Foo(1));
}

int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

```
hz@ubt:~/gtest$ ./mytest
[*****] Running 1 test from 1 test case.
[*****] Global test environment set-up.
[*****] 1 test from FooTest
[ RUN    ] FooTest.NAME
[ OK     ] FooTest.NAME (0 ms)
[*****] 1 test from FooTest (10 ms total)
[*****] Global test environment tear-down
[*****] 1 test from 1 test case ran. (12 ms total)
[ PASSED ] 1 test.
```

2. 玩转 gtest

2.1 断言

gtest 中，断言的宏可以理解为分为两类，一类是 ASSERT 系列，一类是 EXPECT 系列。一个直观的解释就是：

- (1) ASSERT_* 系列的断言，当检查点失败时，退出当前函数（注意：并非退出当前案例）。
- (2) EXPECT_* 系列的断言，当检查点失败时，继续往下执行。

2.1.1 布尔值检查

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_TRUE(<i>condition</i>)	EXPECT_TRUE(<i>condition</i>)	<i>condition</i> is true
ASSERT_FALSE(<i>condition</i>)	EXPECT_FALSE(<i>condition</i>)	<i>condition</i> is false

2.1.2 数值型数据检查

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_EQ(<i>expected</i> , <i>actual</i>);	EXPECT_EQ(<i>expected</i> , <i>actual</i>);	<i>expected</i> == <i>actual</i>
ASSERT_NE(<i>val1</i> , <i>val2</i>);	EXPECT_NE(<i>val1</i> , <i>val2</i>);	<i>val1</i> != <i>val2</i>
ASSERT_LT(<i>val1</i> , <i>val2</i>);	EXPECT_LT(<i>val1</i> , <i>val2</i>);	<i>val1</i> < <i>val2</i>
ASSERT_LE(<i>val1</i> , <i>val2</i>);	EXPECT_LE(<i>val1</i> , <i>val2</i>);	<i>val1</i> <= <i>val2</i>
ASSERT_GT(<i>val1</i> , <i>val2</i>);	EXPECT_GT(<i>val1</i> , <i>val2</i>);	<i>val1</i> > <i>val2</i>
ASSERT_GE(<i>val1</i> , <i>val2</i>);	EXPECT_GE(<i>val1</i> , <i>val2</i>);	<i>val1</i> >= <i>val2</i>

2.1.3 字符串检查

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_STREQ(<i>expected_str</i> , <i>actual_str</i>);	EXPECT_STREQ(<i>expected_str</i> , <i>actual_str</i>);	the two C strings have the same content
ASSERT_STRNE(<i>str1</i> , <i>str2</i>);	EXPECT_STRNE(<i>str1</i> , <i>str2</i>);	the two C strings have different content
ASSERT_STRCASEEQ(<i>expected_str</i> , <i>actual_str</i>);	EXPECT_STRCASEEQ(<i>expected_str</i> , <i>actual_str</i>);	the two C strings have the same content, ignoring case. (即不区分大小写)
ASSERT_STRCASENE(<i>str1</i> , <i>str2</i>);	EXPECT_STRCASENE(<i>str1</i> , <i>str2</i>);	the two C strings have different content, ignoring case.

*STREQ*和*STRNE*同时支持 `char*`和 `wchar_t*`类型的，*STRCASEEQ*和*STRCASENE*却只接收 `char*`，估计是不常用吧。

2.1.4 显示返回成功或失败

直接返回成功：SUCCEED();

返回失败：

Fatal assertion	Nonfatal assertion
FAIL();	ADD_FAILURE();

2.1.5 异常检查

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_THROW(statement, exception_type);</code>	<code>EXPECT_THROW(statement, exception_type);</code>	<i>statement</i> throws an exception of the given type
<code>ASSERT_ANY_THROW(statement);</code>	<code>EXPECT_ANY_THROW(statement);</code>	<i>statement</i> throws an exception of any type
<code>ASSERT_NO_THROW(statement);</code>	<code>EXPECT_NO_THROW(statement);</code>	<i>statement</i> doesn't throw any exception

例如：

```
int Foo(int a, int b){
    if (a == 0 || b == 0){
        throw "don't do that";
    }
    int c = a % b;
    if (c == 0)
        return b;
    return Foo(b, c);
}

TEST(FooTest, HandleZeroInput)
{
    EXPECT_ANY_THROW(Foo(10, 0));
    EXPECT_THROW(Foo(0, 5), const char*); //注意，在 windows 下写为 char* 不会报错，但是在 Linux 下会。
}
```

2.1.6 Predicate Assertions

在使用 `EXPECT_TRUE` 或 `ASSERT_TRUE` 时，有时希望能够输出更加详细的信息，比如检查一个函数的返回值 `TRUE` 还是 `FALSE` 时，希望能够输出传入的参数是什么，以便失败后好跟踪。因此提供了如下的断言：

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_PRED1(pred1, val1);</code>	<code>EXPECT_PRED1(pred1, val1);</code>	<i>pred1(val1)</i> returns true
<code>ASSERT_PRED2(pred2, val1, val2);</code>	<code>EXPECT_PRED2(pred2, val1, val2);</code>	<i>pred2(val1, val2)</i> returns true
...

Google 人说了，他们只提供 ≤ 5 个参数的，如果需要测试更多的参数，直接告诉他们。下面看看这个东西怎么用。

```
bool MutuallyPrime(int m, int n)
{
```

```

        return Foo(m, n) > 1;
    }

TEST(PredicateAssertionTest, Demo)
{
    int m = 5, n = 6;

    EXPECT_PRED2(MutuallyPrime, m, n);
}

```

当失败时，返回错误信息：

error: MutuallyPrime(m, n) evaluates to false, where

m evaluates to 5

n evaluates to 6

如果对这样的输出不满意的话，还可以自定义输出格式，通过如下：

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_PRED_FORMAT1(<i>pred_for</i> <i>mat1, val1</i>);`	EXPECT_PRED_FORMAT1(<i>pred_for</i> <i>mat1, val1</i>);	<i>pred_format1(val1</i> <i>)</i> is successful
ASSERT_PRED_FORMAT2(<i>pred_for</i> <i>mat2, val1, val2</i>);	EXPECT_PRED_FORMAT2(<i>pred_for</i> <i>mat2, val1, val2</i>);	<i>pred_format2(val1</i> <i>, val2)</i> is successful
...	...	

用法示例：

```

testing::AssertionResult AssertFoo(const char* m_expr, const char* n_expr, const char* k_exp
r, int m, int n, int k) {
    if (Foo(m, n) == k)
        return testing::AssertionSuccess();

    testing::Message msg;
    msg << m_expr << " 和 " << n_expr << " 的最大公约数应该是：" << Foo(m, n) << " 而不是：" << k
_expr;

    return testing::AssertionFailure(msg);
}

TEST(AssertFooTest, HandleFail)
{
    EXPECT_PRED_FORMAT3(AssertFoo, 3, 6, 2);
}

```

失败时，输出信息：

error: 3 和 6 的最大公约数应该是：3 而不是：2

是不是更温馨呢，呵呵。

2.1.7 浮点型检查

Fatal assertion	Nonfatal assertion	Verifies
-----------------	--------------------	----------

<code>ASSERT_FLOAT_EQ(<i>expected</i>, <i>actual</i>) ;</code>	<code>EXPECT_FLOAT_EQ(<i>expected</i>, <i>actual</i>) ;</code>	the two float values are almost equal
<code>ASSERT_DOUBLE_EQ(<i>expected</i>, <i>actual</i>) ;</code>	<code>EXPECT_DOUBLE_EQ(<i>expected</i>, <i>actual</i>) ;</code>	the two double values are almost equal

对相近的两个数比较：

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_NEAR(<i>val1</i>, <i>val2</i>, <i>abs_error</i>) ;</code>	<code>EXPECT_NEAR(<i>val1</i>, <i>val2</i>, <i>abs_error</i>) ;</code>	the difference between <i>val1</i> and <i>val2</i> doesn't exceed the given absolute error

同时，还可以使用：

```
EXPECT_PRED_FORMAT2(testing::FloatLE, val1, val2);
EXPECT_PRED_FORMAT2(testing::DoubleLE, val1, val2);
```

2.1.8 Windows HRESULT assertions

Fatal assertion	Nonfatal assertion
<code>ASSERT_HRESULT_SUCCEEDED(<i>expression</i>) ;</code>	<code>EXPECT_HRESULT_SUCCEEDED(<i>expression</i>) ;</code>
<code>ASSERT_HRESULT_FAILED(<i>expression</i>) ;</code>	<code>EXPECT_HRESULT_FAILED(<i>expression</i>) ;</code>

例如：

```
CComPtr shell;
ASSERT_HRESULT_SUCCEEDED(shell.CoCreateInstance(L"Shell.Application"));
CComVariant empty;
ASSERT_HRESULT_SUCCEEDED(shell->ShellExecute(CComBSTR(url), empty, empty, empty, empty));
```

2.1.9 类型检查

类型检查失败时，直接导致代码编不过，难得用处就在这？看下面的例子：

```
template <typename T> class FooType {
public:
    void Bar() { testing::StaticAssertTypeEq<int, T>(); }
};

TEST(TypeAssertionTest, Demo)
{
    FooType<bool> fooType;
    fooType.Bar();
}
```

2.2 事件机制

2.2.1 前言

gtest 提供了多种事件机制,非常方便我们在案例之前或之后做一些操作。总结一下 gtest 的事件一共有 3 种:

1. 全局的, **所有**案例执行前后。
2. TestSuite 级别的,在**某一批**案例中第一个案例前,最后一个案例执行后。
3. TestCase 级别的, **每个** TestCase 前后。

2.2.2 全局事件

要实现全局事件,必须写一个类,继承 `testing::Environment` 类,实现里面的 `SetUp` 和 `TearDown` 方法。

1. `SetUp()`方法在所有案例执行前执行
2. `TearDown()`方法在所有案例执行后执行

```
class FooEnvironment : public testing::Environment
{
public:
    virtual void SetUp()
    {
        std::cout << "Foo FooEnvironment SetUP" << std::endl;
    }
    virtual void TearDown()
    {
        std::cout << "Foo FooEnvironment TearDown" << std::endl;
    }
};
```

当然,这样还不够,我们还需要告诉 gtest 添加这个全局事件,我们需要在 `main` 函数中通过 `testing::AddGlobalTestEnvironment` 方法将事件挂进来,也就是说,我们可以写很多个这样的类,然后将他们的事件都挂上去。

```
int _tmain(int argc, _TCHAR* argv[])
{
    testing::AddGlobalTestEnvironment(new FooEnvironment);
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

2.2.3 TestSuite 事件

我们需要写一个类,继承 `testing::Test`,然后实现两个静态方法

1. `SetUpTestCase()` 方法在第一个 TestCase 之前执行
2. `TearDownTestCase()` 方法在最后一个 TestCase 之后执行

```
class FooTest : public testing::Test {
protected:
```

```

static void SetUpTestCase() {
    shared_resource_ = new ...;
}

static void TearDownTestCase() {
    delete shared_resource_;
    shared_resource_ = NULL;
}

// Some expensive resource shared by all tests.
static T* shared_resource_;
};

```

在编写测试案例时，我们需要使用 TEST_F 这个宏，第一个参数必须是我们上面类的名字，代表一个 TestSuite。

```

TEST_F(FooTest, Test1)
{
    // you can refer to shared_resource here ...
}

TEST_F(FooTest, Test2)
{
    // you can refer to shared_resource here ...
}

```

2.2.4 TestCase 事件

TestCase 事件是挂在每个案例执行前后的，实现方式和上面的几乎一样，不过需要实现的是 SetUp 方法和 TearDown 方法：

1. SetUp()方法在每个 TestCase 之前执行
2. TearDown()方法在每个 TestCase 之后执行

```

class FooCalcTest:public testing::Test
{
protected:
    virtual void SetUp()
    {
        m_foo.Init();
    }

    virtual void TearDown()
    {
        m_foo.Finalize();
    }

    FooCalc m_foo;
};

TEST_F(FooCalcTest, HandleNoneZeroInput)
{

```

```

    EXPECT_EQ(4, m_foo.Calc(12, 16));
}

TEST_F(FooCalcTest, HandleNoneZeroInput_Error)
{
    EXPECT_EQ(5, m_foo.Calc(12, 16));
}

```

2.2.5 总结

gtest 提供的这三种事件机制还是非常的简单和灵活的。同时，通过继承 Test 类，使用 TEST_F 宏，我们可以在案例之间共享一些通用方法，共享资源。使得我们的案例更加的简洁，清晰。

2.3 参数化

2.3.1 前言

在设计测试案例时，经常需要考虑给被测函数传入不同的值的情况。我们之前的做法通常是写一个通用方法，然后编写在测试案例调用它。即使使用了通用方法，这样的工作也是有很多重复性的，程序员都懒，都希望能够少写代码，多复用代码。Google 的程序员也一样，他们考虑到了这个问题，并且提供了一个灵活的参数化测试的方案。

2.3.2 旧的方案

为了对比，我还是把旧的方案提一下。假如我要编写判断结果为 True 的测试案例，我需要传入一系列数值让函数 IsPrime 去判断是否为 True（当然，即使传入再多值也无法确保函数正确，呵呵），因此我需要这样编写如下的测试案例：

```

TEST(IsPrimeTest, HandleTrueReturn)
{
    EXPECT_TRUE(IsPrime(3));
    EXPECT_TRUE(IsPrime(5));
    EXPECT_TRUE(IsPrime(11));
    EXPECT_TRUE(IsPrime(23));
    EXPECT_TRUE(IsPrime(17));
}

```

我们注意到，在这个测试案例中，我至少复制粘贴了 4 次，假如参数有 50 个，100 个，怎么办？同时，上面的写法产生的是 1 个测试案例，里面有 5 个检查点，假如我要把 5 个检查变成 5 个单独的案例，将会更加累人。

接下来，来看看 gtest 是如何为我们解决这些问题的。

2.3.3 使用参数化后的方案

(1) 告诉 gtest 你的参数类型是什么

你必须添加一个类，继承 `testing::TestWithParam<T>`，其中 `T` 就是你需要参数化的参数类型，比如上面的例子，我需要参数化一个 `int` 型的参数

```
class IsPrimeParamTest : public::testing::TestWithParam<int>
{
...
};
```

(2) 告诉 gtest 你拿到参数的值后，具体做些什么样的测试

这里，我们要使用一个新的宏（嗯，挺兴奋的）：`TEST_P`，关于这个“P”的含义，Google 给出的答案非常幽默，就是说你可以理解为“parameterized”或者“pattern”。我更倾向于“parameterized”的解释，呵呵。在 `TEST_P` 宏里，使用 `GetParam()` 获取当前的参数的具体值。

```
TEST_P(IsPrimeParamTest, HandleTrueReturn)
{
    int n = GetParam();
    EXPECT_TRUE(IsPrime(n));
}
```

嗯，非常的简洁！

(3) 告诉 gtest 你想要测试的参数范围是什么

使用 `INSTANTIATE_TEST_CASE_P` 这宏来告诉 gtest 你要测试的参数范围：

```
INSTANTIATE_TEST_CASE_P(TrueReturn, IsPrimeParamTest, testing::Values
(3, 5, 11, 23, 17));
```

第一个参数是测试案例的前缀，可以任意取。

第二个参数是测试案例的名称，需要和之前定义的参数化的类的名称相同，如：`IsPrimeParamTest`

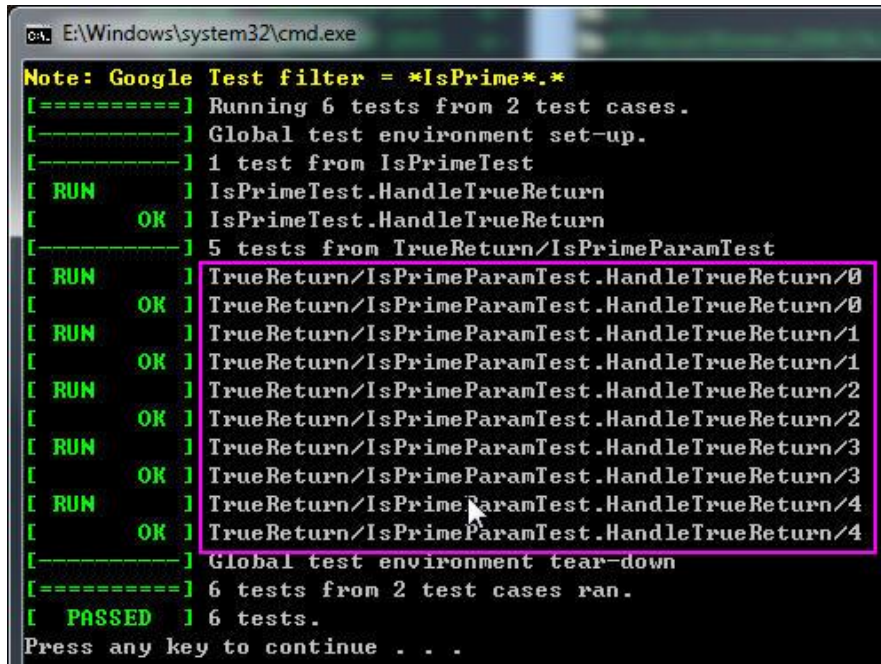
第三个参数是可以理解为参数生成器，上面的例子使用 `test::Values` 表示使用括号内的参数。Google 提供了一系列的参数生成的函数：

<code>Range(begin, end[, step])</code>	范围在 <code>begin~end</code> 之间，步长为 <code>step</code> ，不包括 <code>end</code>
<code>Values(v1, v2, ..., vN)</code>	<code>v1,v2</code> 到 <code>vN</code> 的值
<code>ValuesIn(container) and ValuesIn(begin, end)</code>	从一个 C 类型的数组或是 STL 容器，或是迭代器中取值
<code>Bool()</code>	取 <code>false</code> 和 <code>true</code> 两个值
<code>Combine(g1, g2, ..., gN)</code>	<p>这个比较强悍，它将 <code>g1,g2,...gN</code> 进行排列组合，<code>g1,g2,...gN</code> 本身是一个参数生成器，每次分别从 <code>g1,g2,...gN</code> 中各取出一个值，组合成一个元组(Tuple)作为一个参数。</p> <p>说明：这个功能只在提供了 <code><tr1/tuple></code> 头的系统中有效。<code>gtest</code> 会自动去判断是否支持 <code>tr/tuple</code>，如果你的系统确实支持，而 <code>gtest</code> 判断错误的话，你可以重新</p>

定义宏 GTEST_HAS_TR1_TUPLE=1。

2.3.4 参数化后的测试案例名

因为使用了参数化的方式执行案例，我非常想知道运行案例时，每个案例名称是如何命名的。我执行了上面的代码，输出如下：



```

E:\Windows\system32\cmd.exe

Note: Google Test filter = *IsPrime*.
[=====] Running 6 tests from 2 test cases.
[-----] Global test environment set-up.
[-----] 1 test from IsPrimeTest
[ RUN     ] IsPrimeTest.HandleTrueReturn
[ OK      ] IsPrimeTest.HandleTrueReturn
[-----] 5 tests from TrueReturn/IsPrimeParamTest
[ RUN     ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/0
[ OK      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/0
[ RUN     ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/1
[ OK      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/1
[ RUN     ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/2
[ OK      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/2
[ RUN     ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/3
[ OK      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/3
[ RUN     ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/4
[ OK      ] TrueReturn/IsPrimeParamTest.HandleTrueReturn/4
[-----] Global test environment tear-down
[=====] 6 tests from 2 test cases ran.
[ PASSED ] 6 tests.
Press any key to continue . . .
```

从上面的框框中的案例名称大概能够看出案例的命名规则，对于需要了解每个案例的名称的我来说，这非常重要。命名规则大概为：

prefix/test_case_name.test.name/index

2.3.5 类型参数化

gtest 还提供了应付各种不同类型的数据时的方案，以及参数化类型的方案。我个人感觉这个方案有些复杂。首先要了解一下类型化测试，就用 gtest 里的例子了。

首先定义一个模版类，继承 testing::Test：

```
template <typename T>
class FooTest : public testing::Test {
public:
    ...

    typedef std::list<T> List;

    static T shared_;

    T value_;
};
```

接着我们定义需要测试到的具体数据类型，比如下面定义了需要测试 char,int 和 unsigned int：

```
typedef testing::Types<char, int, unsigned int> MyTypes;

TYPED_TEST_CASE(FooTest, MyTypes);
```

又是一个新的宏，来完成我们的测试案例，在声明模版的数据类型时，使用 `TypeParam`

```
TYPED_TEST(FooTest, DoesBlah) {

    // Inside a test, refer to the special name TypeParam to get the type
    // parameter. Since we are inside a derived class template, C++ requires
    // us to visit the members of FooTest via 'this'.

    TypeParam n = this->value_;

    // To visit static members of the fixture, add the 'TestFixture::'
    // prefix.

    n += TestFixture::shared_;

    // To refer to typedefs in the fixture, add the 'typename TestFixture::'
    // prefix. The 'typename' is required to satisfy the compiler.

    typename TestFixture::List values;

    values.push_back(n);

    ...

}
```

上面的例子看上去也像是类型的参数化，但是还不够灵活，因为需要事先知道类型的列表。gtest 还提供一种更加灵活的类型参数化的方式，允许你在完成测试的逻辑代码之后再考虑需要参数化的类型列表，并且还可以重复的使用这个类型列表。下面也是官方的例子：

```
template <typename T>

class FooTest : public testing::Test {

    ...

};
```

```
TYPED_TEST_CASE_P(FooTest);
```

接着又是一个新的宏 `TYPED_TEST_P` 来完成我们的测试案例：

```
TYPED_TEST_P(FooTest, DoesBlah) {

    // Inside a test, refer to TypeParam to get the type parameter.

    TypeParam n = 0;

    ...

}

TYPED_TEST_P(FooTest, HasPropertyA) { ... }
```

接着，我们需要我们上面的案例，使用 `REGISTER_TYPED_TEST_CASE_P` 宏，第一个参数是 testcase 的名称，后面的参数是 test 的名称

```
REGISTER_TYPED_TEST_CASE_P(FooTest, DoesBlah, HasPropertyA);
```

接着指定需要的类型列表：

```
typedef testing::Types<char, int, unsigned int> MyTypes;  
INSTANTIATE_TYPED_TEST_CASE_P(My, FooTest, MyTypes);
```

这种方案相比之前的方案提供更加好的灵活度，当然，框架越灵活，复杂度也会随之增加。

2.3.6 总结

gtest 为我们提供的参数化测试的功能给我们的测试带来了极大的方便，使得我们可以写更少更优美的代码，完成多种参数类型的测试案例。

2.4 死亡测试

2.4.1 前言

“死亡测试”名字比较恐怖，这里的“死亡”指的是程序的崩溃。通常在测试过程中，我们需要考虑各种各样的输入，有的输入可能直接导致程序崩溃，这时我们就需要检查程序是否按照预期的方式挂掉，这也就是所谓的“死亡测试”。gtest 的死亡测试能做到在一个安全的环境下执行崩溃的测试案例，同时又对崩溃结果进行验证。

2.4.2 使用的宏

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_DEATH(statement, regex`);</code>	<code>EXPECT_DEATH(statement, regex`);</code>	<i>statement</i> crashes with the given error
<code>ASSERT_EXIT(statement, predicate, regex`);</code>	<code>EXPECT_EXIT(statement, predicate, regex`);</code>	<i>statement</i> exits with the given error and its exit code matches <i>predicate</i>

由于有些异常只在 Debug 下抛出，因此还提供了*_DEBUG_DEATH，用来处理 Debug 和 Release 下的不同。

2.4.3 *_DEATH(statement, regex`)

1. statement 是被测试的代码语句
2. regex 是一个正则表达式，用来匹配异常时在 stderr 中输出的内容

如下面的例子：

```
void Foo()  
{  
    int *pInt = 0;  
    *pInt = 42;  
}  
  
TEST(FooDeathTest, Demo)  
{
```

```
EXPECT_DEATH(Foo(), "");  
}
```

重要：编写死亡测试案例时，TEST 的第一个参数，即 testcase_name，请使用 DeathTest 后缀。原因是 gtest 会优先运行死亡测试案例，应该是为线程安全考虑。

2.4.4 *_EXIT(statement, predicate, regex)

1. statement 是被测试的代码语句
2. predicate 在这里必须是一个委托，接收 int 型参数，并返回 bool。只有当返回值为 true 时，死亡测试案例才算通过。gtest 提供了一些常用的 predicate：

```
testing::ExitedWithCode(exit_code)
```

如果程序正常退出并且退出码与 exit_code 相同则返回 true

```
testing::KilledBySignal(signal_number) // Windows 下不支持
```

如果程序被 signal_number 信号 kill 的话就返回 true

3. regex 是一个正则表达式，用来匹配异常时在 stderr 中输出的内容

这里，要说明的是，*_DEATH 其实是对 *_EXIT 进行的一次包装，*_DEATH 的 predicate 判断进程是否以非 0 退出码退出或被一个信号杀死。

例子：

```
TEST(ExitDeathTest, Demo)  
{  
    EXPECT_EXIT(_exit(1), testing::ExitedWithCode(1), "");  
}
```

2.4.5 *_DEBUG_DEATH

先来看定义：

```
#ifdef NDEBUG  
  
#define EXPECT_DEBUG_DEATH(statement, regex) \  
    do { statement; } while (false)  
#define ASSERT_DEBUG_DEATH(statement, regex) \  
    do { statement; } while (false)  
#else  
#define EXPECT_DEBUG_DEATH(statement, regex) \  
    EXPECT_DEATH(statement, regex)  
#define ASSERT_DEBUG_DEATH(statement, regex) \  
    ASSERT_DEATH(statement, regex)  
  
#endif // NDEBUG for EXPECT_DEBUG_DEATH
```

可以看到，在 Debug 版和 Release 版本下，*_DEBUG_DEATH 的定义不一样。因为很多异常只会在 Debug 版本下抛出，而在 Release 版本下不会抛出，所以针对 Debug 和 Release 分别做了不同的处理。看 gtest 里自带的例子就明白了：

```
int DieInDebugElse12(int* sideeffect) {  
    if (sideeffect) *sideeffect = 12;  
}
```

```

#ifndef NDEBUG
    GTEST_LOG_(FATAL, "debug death inside DieInDebugElse12()");
#endif // NDEBUG
    return 12;
}

TEST(TestCase, TestDieOr12WorksInDgbAndOpt)
{
    int sideeffect = 0;

    // Only asserts in dbg.
    EXPECT_DEBUG_DEATH(DieInDebugElse12(&sideeffect), "death");

    #ifdef NDEBUG
        // opt-mode has sideeffect visible.
        EXPECT_EQ(12, sideeffect);
    #else
        // dbg-mode no visible sideeffect.
        EXPECT_EQ(0, sideeffect);
    #endif
}

```

2.4.6 关于正则表达式

在 POSIX 系统 (Linux, Cygwin, 和 Mac) 中, gtest 的死亡测试中使用的是 POSIX 风格的正则表达式, 想了解 POSIX 风格表达式可参考:

- 1). [POSIX extended regular expression](#)
- 2). [Wikipedia entry](#).

在 Windows 系统中, gtest 的死亡测试中使用的是 gtest 自己实现的简单的正则表达式语法。相比 POSIX 风格, gtest 的简单正则表达式少了很多内容, 比如 ("x|y"), ("(xy)"), ("[xy]") 和 ("x{5, 7}") 都不支持。

下面是简单正则表达式支持的一些内容:

	matches any literal character c
\\d	matches any decimal digit
\\D	matches any character that's not a decimal digit
\\f	matches \\f
\\n	matches \\n
\\r	matches \\r
\\s	matches any ASCII whitespace, including \\n
\\S	matches any character that's not a whitespace
\\t	matches \\t

<code>\\v</code>	matches <code>\v</code>
<code>\\w</code>	matches any letter, <code>_</code> , or decimal digit
<code>\\W</code>	matches any character that <code>\\w</code> doesn't match
<code>\\c</code>	matches any literal character <code>c</code> , which must be a punctuation
<code>.</code>	matches any single character except <code>\n</code>
<code>A?</code>	matches 0 or 1 occurrences of <code>A</code>
<code>A*</code>	matches 0 or many occurrences of <code>A</code>
<code>A+</code>	matches 1 or many occurrences of <code>A</code>
<code>^</code>	matches the beginning of a string (not that of each line)
<code>\$</code>	matches the end of a string (not that of each line)
<code>xy</code>	matches <code>x</code> followed by <code>y</code>

gtest 定义两个宏，用来表示当前系统支持哪套正则表达式风格：

- 1). POSIX 风格：`GTEST_USES_POSIX_RE = 1`
- 2). Simple 风格：`GTEST_USES_SIMPLE_RE=1`

2.4.7 死亡测试运行方式

1. fast 方式（默认的方式）

```
testing::FLAGS_gtest_death_test_style = "fast";
```

2. threadsafe 方式

```
testing::FLAGS_gtest_death_test_style = "threadsafe";
```

你可以在 `main()` 里为所有的死亡测试设置测试形式，也可以为某次测试单独设置。Google Test 会在每次测试之前保存这个标记并在测试完成后恢复，所以你不需要去管这部分工作。如：

```
TEST(MyDeathTest, TestOne) {
    testing::FLAGS_gtest_death_test_style = "threadsafe";
    // This test is run in the "threadsafe" style:
    ASSERT_DEATH(ThisShouldDie(), "");
}

TEST(MyDeathTest, TestTwo) {
    // This test is run in the "fast" style:
    ASSERT_DEATH(ThisShouldDie(), "");
}

int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);
    testing::FLAGS_gtest_death_test_style = "fast";
    return RUN_ALL_TESTS();
}
```

2.4.8 注意事项

1. 不要在死亡测试里释放内存。
2. 在父进程里再次释放内存。
3. 不要在程序中使用内存堆检查。

2.4.9 总结

关于死亡测试，gtest 官方的文档已经很详细了，同时在源码中也有大量的示例。如想了解更多的请参考官方的文档，或是直接看 gtest 源码。

简单来说，通过 `*_DEATH(statement, regex)` 和 `*_EXIT(statement, predicate, regex)`，我们可以非常方便的编写导致崩溃的测试案例，并且在不影响其他案例执行的情况下，对崩溃案例的结果进行检查。

2.5 运行参数

2.5.1 前言

使用 gtest 编写的测试案例通常本身就是一个可执行文件，因此运行起来非常方便。同时，gtest 也为我们提供了一系列的运行参数（环境变量、命令行参数或代码里指定），使得我们可以对案例的执行进行一些有效的控制。

2.5.2 基本介绍

前面提到，对于运行参数，gtest 提供了三种设置的途径：

1. 系统环境变量
2. 命令行参数
3. 代码中指定 FLAG

因为提供了三种途径，就会有优先级的问题，有一个原则是，最后设置的那个会生效。不过总结一下，通常情况下，比较理想的优先级为：

命令行参数 > 代码中指定 FLAG > 系统环境变量

为什么我们编写的测试案例能够处理这些命令行参数呢？是因为我们在 main 函数中，将命令行参数交给了 gtest，由 gtest 来搞定命令行参数的问题。

```
int _tmain(int argc, _TCHAR* argv[])
{
    testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}
```

这样，我们就拥有了接收和响应 gtest 命令行参数的能力。如果需要在代码中指定 FLAG，可以使用 `testing::GTEST_FLAG` 这个宏来设置。比如相对于命令行参数 `--gtest_output`，可以使用 `testing::GTEST_FLAG(output) = "xml:"` 来设置。注意到了，不需要加 `--gtest` 前缀了。同时，推荐将这句放置 `InitGoogleTest` 之前，这样就可以使得对于同样的参数，命令行参数优先级高于代码中指定。

```
int _tmain(int argc, _TCHAR* argv[])
{
    testing::GTEST_FLAG(output) = "xml:";
    testing::InitGoogleTest(&argc, argv);
}
```



```
        return RUN_ALL_TESTS();
    }
}
```

最后再来说下第一种设置方式-系统环境变量。如果需要 gtest 的设置系统环境变量，必须注意的是：

1. 系统环境变量全大写，比如对于--gtest_output，响应的系统环境变量为：GTEST_OUTPUT
2. 有一个命令行参数例外，那就是--gtest_list_tests，它是不接受系统环境变量的。（只是用来罗列测试案例名称）

2.5.3 参数列表

了解了上面的内容，我这里就直接将所有命令行参数总结和罗列一下。如果想要获得详细的命令行说明，直接运行你的案例，输入命令行参数：/? 或 --help 或 -help

1. 测试案例集合

命令行参数	说明
--gtest_list_tests	使用这个参数时，将不会执行里面的测试案例，而是输出一个案例的列表。
--gtest_filter	<p>对执行的测试案例进行过滤，支持通配符</p> <ul style="list-style-type: none">? 单个字符*- 排除，如，-a 表示除了 a:取或，如，a:b 表示 a 或 b <p>比如下面的例子：</p> <p>./foo_test 没有指定过滤条件，运行所有案例</p> <p>./foo_test --gtest_filter=* 使用通配符*，表示运行所有案例</p> <p>./foo_test --gtest_filter=FooTest.* 运行所有“测试案例名称(testcase_name)”为 FooTest 的案例</p> <p>./foo_test --gtest_filter=*Null*:~Constructor* 运行所有“测试案例名称(testcase_name)”或“测试名称(test_name)”包含 Null 或 Constructor 的案例。</p> <p>./foo_test --gtest_filter=~DeathTest.* 运行所有非死亡测试案例。</p> <p>./foo_test --gtest_filter=FooTest.*~FooTest.Bar 运行所有“测试案例名称(testcase_name)”为 FooTest 的案例,但是除了 FooTest.Bar 这个案例</p>
--gtest_also_run_disabled_tests	<p>执行案例时，同时也执行被置为无效的测试案例。关于设置测试案例无效的方法为：</p> <p>在测试案例名称或测试名称中添加 DISABLED 前缀，比如：</p> <pre>// Tests that Foo does Abc. TEST(FooTest, DISABLED_DoesAbc) { ... }</pre> <pre>class DISABLED_BarTest : public testing::Test { ... };</pre>

	<pre>// Tests that Bar does Xyz. TEST_F(DISABLED_BarTest, DoesXyz) { ... }</pre> 复制代码
<code>--gtest_repeat=[COUNT]</code>	设置案例重复运行次数，非常棒的功能！比如： <code>--gtest_repeat=1000</code> 重复执行 1000 次，即使中途出现错误。 <code>--gtest_repeat=-1</code> 无限次数执行。。。。 <code>--gtest_repeat=1000 --gtest_break_on_failure</code> 重复执行 1000 次，并且在第一个错误发生时立即停止。这个功能对调试非常有用。 <code>--gtest_repeat=1000 --gtest_filter=FooBar</code> 重复执行 1000 次测试案例名称为 FooBar 的案例。

2. 测试案例输出

命令行参数	说明
<code>--gtest_color=(yes no auto)</code>	输出命令行时是否使用一些五颜六色的颜色。默认是 <code>auto</code> 。
<code>--gtest_print_time</code>	输出命令行时是否打印每个测试案例的执行时间。默认是不打印的。
<code>--gtest_output=xml[:DIRECTOR_PATH\]:FILE_PATH]</code>	将测试结果输出到一个 xml 中。 1. <code>--gtest_output=xml:</code> 不指定输出路径时，默认为案例当前路径。 2. <code>--gtest_output=xml:d:\</code> 指定输出到某个目录 3. <code>--gtest_output=xml:d:\foo.xml</code> 指定输出到 d:\foo.xml 如果不是指定了特定的文件路径，gtest 每次输出的报告不会覆盖，而会以数字后缀的方式创建。xml 的输出内容后面介绍吧。

3. 对案例的异常处理

命令行参数	说明
<code>--gtest_break_on_failure</code>	调试模式下，当案例失败时停止，方便调试
<code>--gtest_throw_on_failure</code>	当案例失败时以 C++异常的方式抛出
<code>--gtest_catch_exceptions</code>	是否捕捉异常。gtest 默认是不捕捉异常的，因此假如你的测试案例抛了一个友好，同时也阻碍了测试案例的运行。如果想不弹这个框，可以通过设置这个参数为一个非零的数。 注意：这个参数只在 Windows 下有效。

2.5.4 XML 报告输出格式

```
<?xml version="1.0" encoding="UTF-8"?>

<testsuites tests="3" failures="1" errors="0" time="35" name="AllTests">

  <testsuite name="MathTest" tests="2" failures="1" errors="0" time="15">
```

```

<testcase name="Addition" status="run" time="7" classname="">
  <failure message="Value of: add(1, 1) Actual: 3 Expected: 2" type=""/>
  <failure message="Value of: add(1, -1) Actual: 1 Expected: 0" type=""/>
</testcase>

<testcase name="Subtraction" status="run" time="5" classname="">
</testcase>
</testsuite>

<testsuite name="LogicTest" tests="1" failures="0" errors="0" time="5">
  <testcase name="NonContradiction" status="run" time="5" classname="">
  </testcase>
</testsuite>
</testsuites>

```

从报告里可以看出,我们之前在 TEST 等宏中定义的测试案例名称(testcase_name)在 xml 测试报告中其实是一个 testsuite name,而宏中的测试名称(test_name)在 xml 测试报告中是一个 testcase name,概念上似乎有点混淆,就看你怎么看吧。

当检查点通过时,不会输出任何检查点的信息。当检查点失败时,会有详细的失败信息输出来 failure 节点。在我使用过程中发现一个问题,当我同时设置了--gtest_filter 参数时,输出的 xml 报告中还是会包含所有测试案例的信息,只不过那些不被执行的测试案例的 status 值为“notrun”。而我之前认为的输出的 xml 报告应该只包含我需要运行的测试案例的信息。不知是否可提供一个只输出需要执行的测试案例的 xml 报告。因为当我需要在 1000 个案例中执行其中 1 个案例时,在报告中很难找到我运行的那个案例,虽然可以查找,但还是很麻烦。

2.5.5 总结

本篇主要介绍了 gtest 案例执行时提供的一些参数的使用方法,这些参数都非常有用。在实际编写 gtest 测试案例时肯定会需要用到。至少我现在比较常用的就是:

1. --gtest_filter
2. --gtest_output=xml[:DIRECTORY_PATH\];:FILE_PATH]
3. --gtest_catch_exceptions

最后再总结一下我使用过程中遇到的几个问题:

1. 同时使用--gtest_filter 和--gtest_output=xml:时,在 xml 测试报告中能否只包含过滤后的测试案例的信息。
2. 有时,我在代码中设置 testing::GTEST_FLAG(catch_exceptions) = 1 和我在命令行中使用--gtest_catch_exceptions 结果稍有不同,在代码中设置 FLAG 方式有时候捕捉不了某些异常,但是通过命令行参数的方式一般都不会有问题。这是我曾经遇到过的一个问题,最后我的处理办法是既在代码中设置 FLAG,又在命令行参数中传入--gtest_catch_exceptions。不知道是 gtest 在 catch_exceptions 方面不够稳定,还是我自己测试案例的问题。

3. 深入解析 gtest

3.1 前言

“深入解析”对我来说的确有些难度，所以我尽量将我学习到和观察到的 gtest 内部实现介绍给大家。本文算是抛砖引玉吧，只能是对 gtest 的整体结构的一些介绍，想要了解更多细节最好的办法还是看 gtest 源码，如果你看过 gtest 源码，你会发现里面的注释非常的详细！好了，下面就开始了解 gtest 吧。

3.2 从 TEST 宏开始

前面的文章已经介绍过 TEST 宏的用法了，通过 TEST 宏，我们可以非常简单的、方便的编写测试案例，比如：

```
TEST(FooTest, Demo)
{
    EXPECT_EQ(1, 1);
}
```

我们先不去看 TEST 宏的定义，而是先使用 /P 参数将 TEST 展开。如果使用的是 Visual Studio 的话：

1. 选中需要展开的代码文件，右键 - 属性 - C/C++ - Preprocessor
2. Generate Preprocessed File 设置 Without Line Numbers (/EP /P) 或 With Line Numbers (/P)
3. 关闭属性对话框，右键选中需要展开的文件，右键菜单中点击：Compile

编译过后，会在源代码目录生成一个后缀为.i 的文件，比如我对上面的代码进行展开，展开后的内容为：

```
class FooTest_Demo_Test : public ::testing::Test
{
public:
    FooTest_Demo_Test() {}
private:
    virtual void TestBody();
    static ::testing::TestInfo* const test_info_;
    FooTest_Demo_Test(const FooTest_Demo_Test &);
    void operator=(const FooTest_Demo_Test &);
};

::testing::TestInfo* const FooTest_Demo_Test
::test_info_ =
    ::testing::internal::MakeAndRegisterTestInfo(
        "FooTest", "Demo", "", "",
        (::testing::internal::GetTypeId()),
        ::testing::Test::SetUpTestCase,
        ::testing::Test::TearDownTestCase,
        new ::testing::internal::TestFactoryImpl< FooTest_Demo_Test>);

void FooTest_Demo_Test::TestBody()
{
    switch (0)
    case 0:
```

```

        if (const ::testing::AssertionResult
            gtest_ar =
                (::testing::internal::EqHelper<(sizeof(::testing::internal::IsNullLiteralHelper(1)) == 1)>::Compare("1", "1", 1, 1)))
        {
            ;
        }
        else
        {
            ::testing::internal::AssertHelper(
                ::testing::TPRT_NONFATAL_FAILURE,
                ".\\gtest_demo.cpp",
                9,
                gtest_ar.failure_message()
            ) = ::testing::Message();
        }
    }
}

```

展开后，我们观察到：

1. TEST 宏展开后，是一个继承自 testing::Test 的类。
2. 我们在 TEST 宏里面写的测试代码，其实是被放到了类的 TestBody 方法中。
3. 通过静态变量 test_info_，调用 MakeAndRegisterTestInfo 对测试案例进行注册。

如下图：



上面关键的方法就是 MakeAndRegisterTestInfo 了，我们跳到 MakeAndRegisterTestInfo 函数中：

```

// 创建一个 TestInfo 对象并注册到 Google Test;
// 返回创建的 TestInfo 对象
//
// 参数:
//
// test_case_name:      测试案例的名称
// name:                测试的名称
// test_case_comment:   测试案例的注释信息
// comment:             测试的注释信息
// fixture_class_id:    test fixture 类的 ID

```

```

//  set_up_tc:                事件函数 SetUpTestCases 的函数地址
//  tear_down_tc:            事件函数 TearDownTestCases 的函数地址
//  factory:                  工厂对象，用于创建测试对象 (Test)

TestInfo* MakeAndRegisterTestInfo(
    const char* test_case_name, const char* name,
    const char* test_case_comment, const char* comment,
    TypeId fixture_class_id,
    SetUpTestCaseFunc set_up_tc,
    TearDownTestCaseFunc tear_down_tc,
    TestFactoryBase* factory) {
    TestInfo* const test_info =
        new TestInfo(test_case_name, name, test_case_comment, comment,
                     fixture_class_id, factory);
    GetUnitTestImpl()->AddTestInfo(set_up_tc, tear_down_tc, test_info);
    return test_info;
}

```

我们看到，上面创建了一个 TestInfo 对象，然后通过 AddTestInfo 注册了这个对象。TestInfo 对象到底是一个什么样的东西呢？

TestInfo 对象主要用于包含如下信息：

1. 测试案例名称 (testcase name)
2. 测试名称 (test name)
3. 该案例是否需要执行
4. 执行案例时，用于创建 Test 对象的函数指针
5. 测试结果

我们还看到，TestInfo 的构造函数中，非常重要的一个参数就是工厂对象，它主要负责在运行测试案例时创建出 Test 对象。我们看到我们上面的例子的 factory 为：

```
new ::testing::internal::TestFactoryImpl< FooTest_Demo_Test>
```

我们明白了，Test 对象原来就是 TEST 宏展开后的那个类的对象(FooTest_Demo_Test)，再看看 TestFactoryImpl 的实现：

```

template <class TestClass>
class TestFactoryImpl : public TestFactoryBase {
public:
    virtual Test* CreateTest() { return new TestClass; }
};

```

这个对象工厂够简单吧，嗯，Simple is better。当我们需要创建一个测试对象(Test)时，调用 factory 的 CreateTest()方法就可以了。

创建了 TestInfo 对象后，再通过下面的方法对 TestInfo 对象进行注册：

```
GetUnitTestImpl()->AddTestInfo(set_up_tc, tear_down_tc, test_info);
```

GetUnitTestImpl()是获取 UnitTestImpl 对象：

```
inline UnitTestImpl* GetUnitTestImpl() {
```

```

        return UnitTest::GetInstance()->impl();
    }

```

其中 UnitTest 是一个单件(Singleton)，整个进程空间只有一个实例，通过 UnitTest::GetInstance()获取单件的实例。上面的代码看到，UnitTestImpl 对象是最终是从 UnitTest 对象中获取的。那么 UnitTestImpl 到底是一个什么样的东西呢？可以这样理解：

UnitTestImpl 是一个在 UnitTest 内部使用的，为执行单元测试案例而提供了一系列实现的那么一个类。
(自己归纳的，可能不准确)

我们上面的 AddTestInfo 就是其中的一个实现，负责注册 TestInfo 实例：

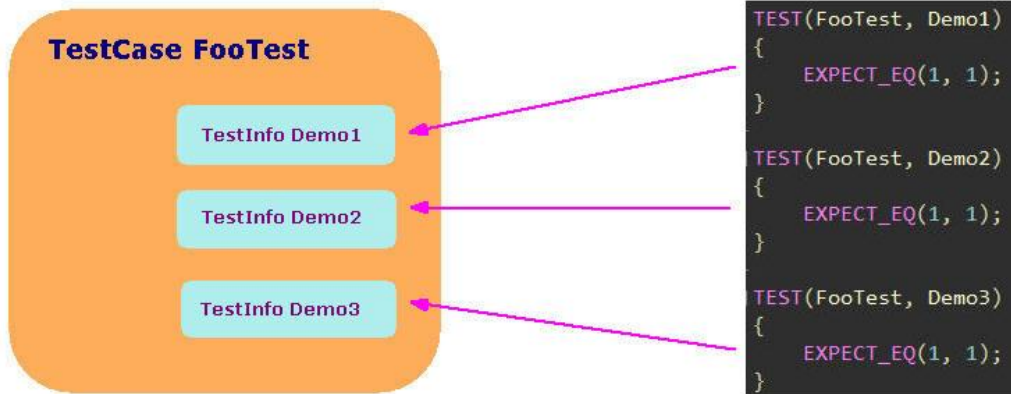
```

// 添加 TestInfo 对象到整个单元测试中
//
// 参数:
//
// set_up_tc:      事件函数 SetUpTestCases 的函数地址
// tear_down_tc:  事件函数 TearDownTestCases 的函数地址
// test_info:      TestInfo 对象
void AddTestInfo(Test::SetUpTestCaseFunc set_up_tc,
                 Test::TearDownTestCaseFunc tear_down_tc,
                 TestInfo * test_info) {
// 处理死亡测试的代码，先不关注它
if (original_working_dir_.IsEmpty()) {
    original_working_dir_.Set(FilePath::GetCurrentDir());
    if (original_working_dir_.IsEmpty()) {
        printf("%s\n", "Failed to get the current working directory.");
        abort();
    }
}
}
// 获取或创建了一个 TestCase 对象，并将 testinfo 添加到 TestCase 对象中。
GetTestCase(test_info->test_case_name(),
            test_info->test_case_comment(),
            set_up_tc,
            tear_down_tc)->AddTestInfo(test_info);
}

```

我们看到，TestCase 对象出来了，并通过 AddTestInfo 添加了一个 TestInfo 对象。这时，似乎豁然开朗了：

1. TEST 宏中的两个参数 第一个参数 testcase_name 就是 TestCase 对象的名称 第二个参数 test_name 就是 Test 对象的名称。而 TestInfo 包含了一个测试案例的一系列信息。
2. 一个 TestCase 对象对应一个或多个 TestInfo 对象。



我们来看看 TestCase 的创建过程(UnitTestImpl::GetTestCase)：

```
// 查找并返回一个指定名称的 TestCase 对象。如果对象不存在，则创建一个并返回
//
// 参数：
//
// test_case_name: 测试案例名称
// set_up_tc:      事件函数 SetUpTestCases 的函数地址
// tear_down_tc:   事件函数 TearDownTestCases 的函数地址
TestCase* UnitTestImpl::GetTestCase(const char* test_case_name,
                                    const char* comment,
                                    Test::SetUpTestCaseFunc set_up_tc,
                                    Test::TearDownTestCaseFunc tear_down_tc) {
    // 从 test_cases 里查找指定名称的 TestCase
    internal::ListNode<TestCase*>* node = test_cases_.FindIf(
        TestCaseNameIs(test_case_name));

    if (node == NULL) {
        // 没找到，我们来创建一个
        TestCase* const test_case =
            new TestCase(test_case_name, comment, set_up_tc, tear_down_tc);

        // 判断是否为死亡测试案例
        if (internal::UnitOptions::MatchesFilter(String(test_case_name),
                                                    kDeathTestCaseFilter)) {
            // 是的话，将该案例插入到最后一个死亡测试案例后
            node = test_cases_.InsertAfter(last_death_test_case_, test_case);
            last_death_test_case_ = node;
        } else {
            // 否则，添加到 test_cases 最后。
            test_cases_.PushBack(test_case);
            node = test_cases_.Last();
        }
    }
}
```



```

    }

    // 返回 TestCase 对象
    return node->element();
}

```

3.3 回过头看看 TEST 宏的定义

```

#define TEST(test_case_name, test_name)\
    GTEST_TEST_(test_case_name, test_name, \
        ::testing::Test, ::testing::internal::GetTestTypeId())

```

[复制代码](#)

同时也看看 TEST_F 宏

```

#define TEST_F(test_fixture, test_name)\
    GTEST_TEST_(test_fixture, test_name, test_fixture, \
        ::testing::internal::GetTypeId<test_fixture>())

```

都是使用了 GTEST_TEST_宏，在看看这个宏如何定义的：

```

#define GTEST_TEST_(test_case_name, test_name, parent_class, parent_id)\
class GTEST_TEST_CLASS_NAME_(test_case_name, test_name) : public parent_class {\
public:\
    GTEST_TEST_CLASS_NAME_(test_case_name, test_name)() {} \
private:\
    virtual void TestBody(); \
    static ::testing::TestInfo* const test_info_; \
    GTEST_DISALLOW_COPY_AND_ASSIGN_(\
        GTEST_TEST_CLASS_NAME_(test_case_name, test_name)); \
}; \
\
::testing::TestInfo* const GTEST_TEST_CLASS_NAME_(test_case_name, test_name)\
    ::test_info_ = \
    ::testing::internal::MakeAndRegisterTestInfo(\
        #test_case_name, #test_name, "", "", \
        (parent_id), \
        parent_class::SetUpTestCase, \
        parent_class::TearDownTestCase, \
        new ::testing::internal::TestFactoryImpl<\
            GTEST_TEST_CLASS_NAME_(test_case_name, test_name)>()) \
void GTEST_TEST_CLASS_NAME_(test_case_name, test_name)::TestBody()

```

不需要多解释了，和我们上面展开看到的差不多，不过这里比较明确的看到了，我们在 TEST 宏里写的就是 TestBody 里的东西。这里再补充说明一下里面的 GTEST_DISALLOW_COPY_AND_ASSIGN_宏，我们上面的例子看出，这个宏展开后：

```

FooTest_Demo_Test(const FooTest_Demo_Test &);

```

```
void operator=(const FooTest_Demo_Test &);
```

正如这个宏的名字一样，它是用于防止对对象进行拷贝和赋值操作的。

3.4 再来了解 RUN_ALL_TESTS 宏

我们的测试案例的运行就是通过这个宏发起的。RUN_ALL_TEST 的定义非常简单：

```
#define RUN_ALL_TESTS() \
    (::testing::UnitTest::GetInstance()->Run())
```

我们又看到了熟悉的::testing::UnitTest::GetInstance()，看来案例的执行时从 UnitTest 的 Run 方法开始的，我提取了一些 Run 中的关键代码，如下：

```
int UnitTest::Run() {
    __try {
        return impl_->RunAllTests();
    } __except(internal::UnitTestOptions::GTestShouldProcessSEH(
        GetExceptionCode())) {
        printf("Exception thrown with code 0x%x.\nFAIL\n", GetExceptionCode());
        fflush(stdout);
        return 1;
    }
    return impl_->RunAllTests();
}
```

我们又看到了熟悉的 impl (UnitTestImpl)，具体案例该怎么执行，还是得靠 UnitTestImpl。

```
int UnitTestImpl::RunAllTests() {

    // ...

    printer->OnUnitTestStart(parent_);

    // 计时
    const TimeInMillis start = GetTimeInMillis();

    printer->OnGlobalSetUpStart(parent_);
    // 执行全局的 SetUp 事件
    environments_.ForEach(SetupEnvironment);
    printer->OnGlobalSetUpEnd(parent_);

    // 全局的 SetUp 事件执行成功的话
    if (!Test::HasFatalFailure()) {
        // 执行每个测试案例
        test_cases_.ForEach(TestCase::RunTestCase);
    }

    // 执行全局的 TearDown 事件
    printer->OnGlobalTearDownStart(parent_);
    environments_in_reverse_order_.ForEach(TearDownEnvironment);
```

```

printer->OnGlobalTearDownEnd(parent_);

elapsed_time_ = GetTimeInMillis() - start;

// 执行完成
printer->OnUnitTestEnd(parent_);

// Gets the result and clears it.
if (!Passed()) {
    failed = true;
}

ClearResult();

// 返回测试结果
return failed ? 1 : 0;
}

```

上面,我们很开心的看到了我们前面讲到的[全局事件](#)的调用。`environments_`是一个 `Environment` 的链表结构 (`List`), 它的内容是我们在 `main` 中通过 :

```
testing::AddGlobalTestEnvironment(new FooEnvironment);
```

添加进去的。`test_cases_`我们之前也了解过了,是一个 `TestCase` 的链表结构 (`List`)。gtest 实现了一个链表,并且提供了一个 `Foreach` 方法,迭代调用某个函数,并将里面的元素作为函数的参数:

```

template <typename F> // F is the type of the function/functor
void ForEach(F functor) const {
    for ( const ListNode<E> * node = Head();
          node != NULL;
          node = node->next() ) {
        functor(node->element());
    }
}

```

因此,我们关注一下:`environments_.ForEach(SetupEnvironment)`,其实是迭代调用了 `SetupEnvironment` 函数:

```
static void SetupEnvironment(Environment* env) { env->Setup(); }
```

最终调用了我们定义的 `Setup()`函数。

再看看 `test_cases_.ForEach(TestCase::RunTestCase)`的 `TestCase::RunTestCase` 实现:

```
static void RunTestCase(TestCase * test_case) { test_case->Run(); }
```

再看 `TestCase` 的 `Run` 实现:

```

void TestCase::Run() {
    if (!should_run_) return;

    internal::UnitTestImpl* const impl = internal::GetUnitTestImpl();
    impl->set_current_test_case(this);
}

```

```

UnitTestEventListenerInterface * const result_printer =
impl->result_printer();

result_printer->OnTestCaseStart(this);
impl->os_stack_trace_getter()->UponLeavingGTest();
// 哈! SetUpTestCases 事件在这里调用

set_up_tc_();

const internal::TimeInMillis start = internal::GetTimeInMillis();
// 嗯, 前面分析的一个 TestCase 对应多个 TestInfo, 因此, 在这里迭代对 TestInfo 调用 RunTest 方法

test_info_list_->ForEach(internal::TestInfoImpl::RunTest);
elapsed_time_ = internal::GetTimeInMillis() - start;

impl->os_stack_trace_getter()->UponLeavingGTest();
// TearDownTestCases 事件在这里调用

tear_down_tc_();

result_printer->OnTestCaseEnd(this);

impl->set_current_test_case(NULL);
}

```

第二种事件机制又浮出我们眼前, 非常兴奋。可以看出, `SetUpTestCases` 和 `TearDownTestCases` 是在一个 `TestCase` 之前和之后调用的。接着看 `test_info_list_->ForEach(internal::TestInfoImpl::RunTest)` :

```

static void RunTest(TestInfo * test_info) {
    test_info->impl()->Run();
}

```

哦? `TestInfo` 也有一个 `impl`? 看来我们之前漏掉了点东西, 和 `UnitTest` 很类似, `TestInfo` 内部也有一个主管各种实现的类, 那就是 `TestInfoImpl`, 它在 `TestInfo` 的构造函数中创建了出来 (还记得前面讲的 `TestInfo` 的创建过程吗?) :

```

TestInfo::TestInfo(const char* test_case_name,
                  const char* name,
                  const char* test_case_comment,
                  const char* comment,
                  internal::TypeId fixture_class_id,
                  internal::TestFactoryBase* factory) {
    impl_ = new internal::TestInfoImpl(this, test_case_name, name,
                                       test_case_comment, comment,
                                       fixture_class_id, factory);
}

```

因此, 案例的执行还得看 `TestInfoImpl` 的 `Run()` 方法, 同样, 我简化一下, 只列出关键部分的代码:

```

void TestInfoImpl::Run() {
    // ...

    UnitTestEventListenerInterface* const result_printer =
        impl->result_printer();

    result_printer->OnTestStart(parent_);
}

```

```

// 开始计时
const TimeInMillis start = GetTimeInMillis();

Test* test = NULL;

__try {
    // 我们的对象工厂，使用 CreateTest() 生成 Test 对象
    test = factory_->CreateTest();
} __except(internal::UnitTestOptions::GTestShouldProcessSEH(
    GetExceptionCode())) {
    AddExceptionThrownFailure(GetExceptionCode(),
                              "the test fixture's constructor");

    return;
}

// 如果 Test 对象创建成功
if (!Test::HasFatalFailure()) {
    // 调用 Test 对象的 Run() 方法，执行测试案例
    test->Run();
}

// 执行完毕，删除 Test 对象
impl->os_stack_trace_getter()->UponLeavingGTest();
delete test;
test = NULL;

// 停止计时
result_.set_elapsed_time(GetTimeInMillis() - start);
result_printer->OnTestEnd(parent_);
}

```

上面看到了我们前面讲到的对象工厂 fatory，通过 fatory 的 CreateTest()方法，创建 Test 对象，然后执行案例又是通过 Test 对象的 Run()方法：

```

void Test::Run() {
    if (!HasSameFixtureClass()) return;

    internal::UnitTestImpl* const impl = internal::GetUnitTestImpl();
    impl->os_stack_trace_getter()->UponLeavingGTest();

    __try {
        // Yeah! 每个案例的 SetUp 事件在这里调用
        SetUp();
    } __except(internal::UnitTestOptions::GTestShouldProcessSEH(
        GetExceptionCode())) {
        AddExceptionThrownFailure(GetExceptionCode(), "SetUp()");
    }
}

```

```

// We will run the test only if SetUp() had no fatal failure.
if (!HasFatalFailure()) {
    impl->os_stack_trace_getter()->UponLeavingGTest();

    __try {
        // 哈哈!! 千辛万苦,我们定义在 TEST 宏里的东西终于被调用了!

        TestBody();
    } __except (internal::UnitTestOptions::GTestShouldProcessSEH(
        GetExceptionCode())) {
        AddExceptionThrownFailure(GetExceptionCode(), "the test body");
    }
}

impl->os_stack_trace_getter()->UponLeavingGTest();

__try {
    // 每个案例的 TearDown 事件在这里调用

    TearDown();
} __except (internal::UnitTestOptions::GTestShouldProcessSEH(
    GetExceptionCode())) {
    AddExceptionThrownFailure(GetExceptionCode(), "TearDown()");
}
}

```

上面的代码里非常极其以及特别的兴奋的看到了执行测试案例的前后事件,测试案例执行 TestBody()的代码。仿佛整个 gtest 的流程在眼前一目了然了。

3.5 总结

本文通过分析 TEST 宏和 RUN_ALL_TEST 宏,了解到了整个 gtest 运作过程,可以说整个过程简洁而优美。之前读《代码之美》,感触颇深,现在读过 gtest 代码,再次让我感触深刻。记得很早前,我对设计的理解是“功能越强大越好,设计越复杂越好,那样才显得牛”,渐渐得,我才发现,简单才是最好。我曾总结过自己写代码的设计原则:功能明确,设计简单。了解了 gtest 代码后,猛然发现 gtest 不就是这样吗,同时 gtest 也给了我很多惊喜,因此,我对 gtest 的评价是:功能强大,设计简单,使用方便。

总结一下 gtest 里的几个关键的对象:

1. UnitTest 单例,总管整个测试,包括测试环境信息,当前执行状态等等。
2. UnitTestImpl UnitTest 内部具体功能的实现者。
3. Test 我们自己编写的,或通过 TEST,TEST_F 等宏展开后的 Test 对象,管理着测试案例的前后事件,具体的执行代码 TestBody。
4. TestCase 测试案例对象,管理着基于 TestCase 的前后事件,管理内部多个 TestInfo。
5. TestInfo 管理着测试案例的基本信息,包括 Test 对象的创建方法。
6. TestInfoImpl TestInfo 内部具体功能的实现者。

本文还有很多 gtest 的细节没有分析到,比如运行参数,死亡测试,跨平台处理,断言的宏等等,希望读者自己把源码下载下来慢慢研究。如本文有错误之处,也请大家指出,谢谢!

4. 打造自己的单元测试框架

4.1 前言

上一篇我们分析了 gtest 的一些内部实现，总的来说整体的流程并不复杂。本篇我们就尝试编写一个精简版本的 C++ 单元测试框架：nancystest，通过编写这个简单的测试框架，将有助于我们理解 gtest。

4.2 整体设计

使用最精简的设计，我们就用两个类，够简单吧：

1. TestCase 类

包含单个测试案例的信息。

2. UnitTest 类

负责所有测试案例的执行，管理。

4.3 TestCase 类

TestCase 类包含一个测试案例的基本信息，包括：测试案例名称，测试案例执行结果，同时还提供了测试案例执行的方法。我们编写的测试案例都继承自 TestCase 类。

```
class TestCase
{
public:
    TestCase(const char* case_name) : testcase_name(case_name){}

    // 执行测试案例的方法
    virtual void Run() = 0;

    int nTestResult; // 测试案例的执行结果
    const char* testcase_name; // 测试案例名称
};
```

4.4 UnitTest 类

我们的 UnitTest 类和 gtest 的一样，是一个单件。我们的 UnitTest 类的逻辑非常简单：

1. 整个进程空间保存一个 UnitTest 的单例。
2. 通过 RegisterTestCase() 将测试案例添加到测试案例集合 testcases_ 中。
3. 执行测试案例时，调用 UnitTest::Run()，遍历测试案例集合 testcases_，调用案例的 Run() 方法

```
class UnitTest
{
public:
    // 获取单例
    static UnitTest* GetInstance();

    // 注册测试案例
    TestCase* RegisterTestCase(TestCase* testcase);
```

```

// 执行单元测试

int Run();

TestCase* CurrentTestCase; // 记录当前执行的测试案例

int nTestResult; // 总的执行结果

int nPassed; // 通过案例数

int nFailed; // 失败案例数

protected:
    std::vector<TestCase*> testcases_; // 案例集合
};

```

下面是 UnitTest 类的实现：

```

UnitTest* UnitTest::GetInstance()
{
    static UnitTest instance;
    return &instance;
}

TestCase* UnitTest::RegisterTestCase(TestCase* testcase)
{
    testcases_.push_back(testcase);
    return testcase;
}

int UnitTest::Run()
{
    nTestResult = 1;
    for (std::vector<TestCase*>::iterator it = testcases_.begin();
        it != testcases_.end(); ++it)
    {
        TestCase* testcase = *it;
        CurrentTestCase = testcase;

        std::cout << green << "======" << std::endl;
        std::cout << green << "Run TestCase:" << testcase->testcase_name << std::endl;
        testcase->Run();
        std::cout << green << "End TestCase:" << testcase->testcase_name << std::endl;
        if (testcase->nTestResult)
        {
            nPassed++;
        }
        else
        {
            nFailed++;
        }
    }
    nTestResult = 0;
}

```



```

    }

}

std::cout << green << "======" << std::endl;
std::cout << green << "Total TestCase : " << nPassed + nFailed << std::endl;
std::cout << green << "Passed : " << nPassed << std::endl;
std::cout << red << "Failed : " << nFailed << std::endl;

return nTestResult;
}

```

4.5 NTEST 宏

接下来定一个宏 NTEST，方便我们写我们的测试案例的类。

```

#define TESTCASE_NAME(testcase_name) \
    testcase_name##_TEST

#define NANCY_TEST_(testcase_name) \
class TESTCASE_NAME(testcase_name) : public TestCase \
{ \
public: \
    TESTCASE_NAME(testcase_name)(const char* case_name) : TestCase(case_name){}; \
    virtual void Run(); \
private: \
    static TestCase* const testcase_; \
}; \
\
TestCase* const TESTCASE_NAME(testcase_name) \
::testcase_ = UnitTest::GetInstance()->RegisterTestCase( \
    new TESTCASE_NAME(testcase_name)(#testcase_name)); \
void TESTCASE_NAME(testcase_name)::Run()

#define NTEST(testcase_name) \
    NANCY_TEST_(testcase_name)

```

4.6 RUN_ALL_TEST 宏

然后是执行所有测试案例的一个宏：

```

#define RUN_ALL_TESTS() \
    UnitTest::GetInstance()->Run();

```

4.7 断言的宏 EXPECT_EQ

这里，我只写一个简单的 EXPECT_EQ：

```

#define EXPECT_EQ(m, n) \
    if (m != n) \
    { \
        UnitTest::GetInstance()->CurrentTestCase->nTestResult = 0; \
    }

```

```
std::cout << red << "Failed" << std::endl; \

std::cout << red << "Expect:" << m << std::endl; \

std::cout << red << "Actual:" << n << std::endl; \

}
```

4.8 案例 Demo

够简单吧，再来看看案例怎么写：

```
#include "nancytest.h"

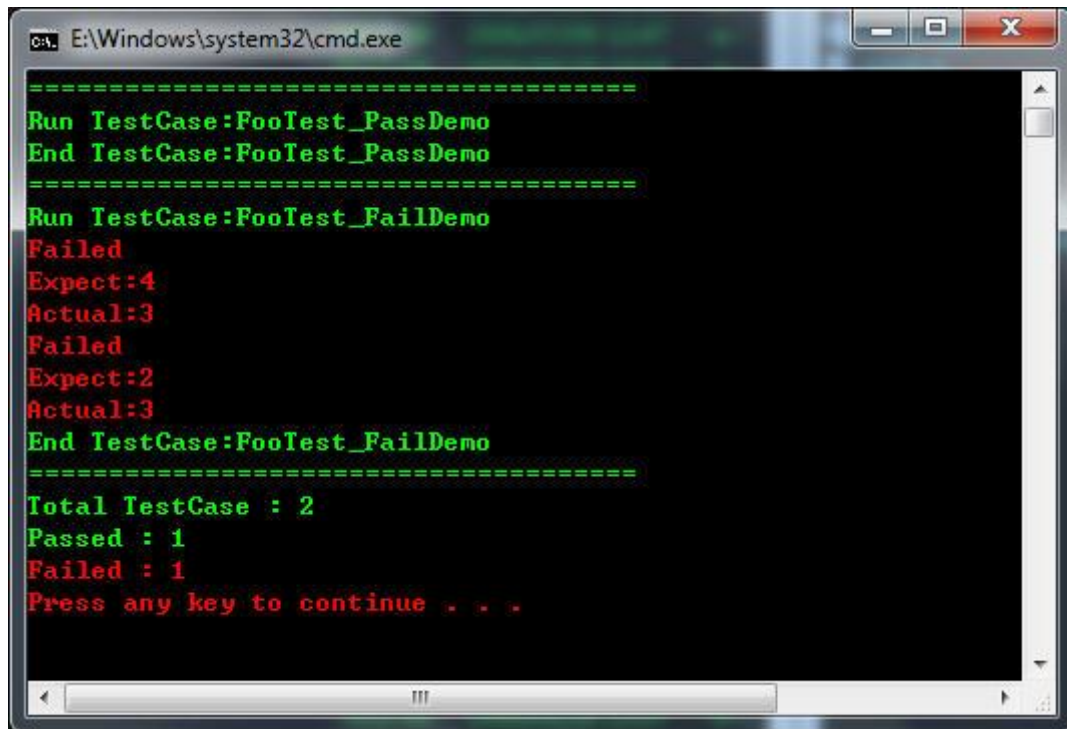
int Foo(int a, int b)
{
    return a + b;
}

NTEST(FooTest_PassDemo)
{
    EXPECT_EQ(3, Foo(1, 2));
    EXPECT_EQ(2, Foo(1, 1));
}

NTEST(FooTest_FailDemo)
{
    EXPECT_EQ(4, Foo(1, 2));
    EXPECT_EQ(2, Foo(1, 2));
}

int _tmain(int argc, _TCHAR* argv[])
{
    return RUN_ALL_TESTS();
}
```

整个一山寨版 gtest，呵。执行一下，看看结果怎么样：



```
=====  
Run TestCase:FooTest_PassDemo  
End TestCase:FooTest_PassDemo  
=====  
Run TestCase:FooTest_FailDemo  
Failed  
Expect:4  
Actual:3  
Failed  
Expect:2  
Actual:3  
End TestCase:FooTest_FailDemo  
=====  
Total TestCase : 2  
Passed : 1  
Failed : 1  
Press any key to continue . . .
```

4.9 总结

本章介绍性的文字比较少，主要是我们在上一篇深入解析 gtest 时已经将整个流程弄清楚了，而现在编写的 nancytest 又是其非常的精简版本，所有直接看代码就可以完全理解。希望通过这个 Demo，能够让大家对 gtest 有更加直观的了解。回到开篇时所说的，我们没有必要每个人都造一个轮子，因为 gtest 已经非常出色的为我们做好了这一切。如果我们每个人都写一个自己的框架的话，一方面我们要付出大量的维护成本，一方面，这个框架也许只能对你有用，无法让大家从中受益。

gtest 正是这么一个优秀 C++ 单元测试框架，它完全开源，允许我们一起为其贡献力量，并能让更多人从中受益。如果你在使用 gtest 过程中发现 gtest 不能满足你的需求时（或发现 BUG），gtest 的开发人员非常急切的想知道他们哪来没做好，或者是 gtest 其实有这个功能，但是很多用户都不知道。所以你可以直接联系 gtest 的开发人员，或者你直接在这里回帖，我会将您的意见转告给 gtest 的主要开发人员。