

以下文章原版位置：<http://www.cnblogs.com/coderzh/archive/2009/03/31/1426758.html>

## 玩转 Google 开源 C++ 单元测试框架 Google Test 系列(gtest)之一 - 初识 gtest

### 一、前言

本篇将介绍一些 gtest 的基本使用，包括下载，安装，编译，建立我们第一个测试 Demo 工程，以及编写一个最简单的测试案例。

### 二、下载

如果不记得网址，直接在 google 里搜 gtest，第一个就是。目前 gtest 的最新版本为 1.3.0，从下列地址可以下载到该最新版本：

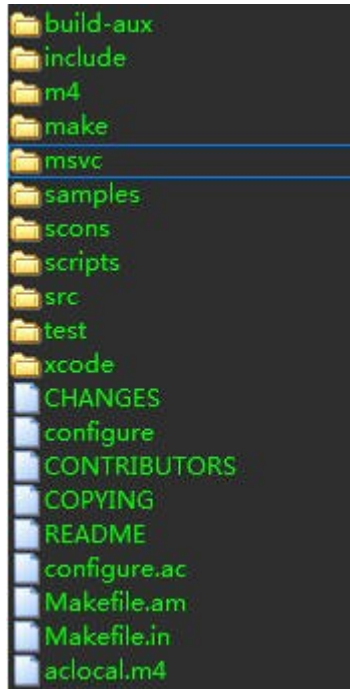
<http://googletest.googlecode.com/files/gtest-1.3.0.zip>

<http://googletest.googlecode.com/files/gtest-1.3.0.tar.gz>

<http://googletest.googlecode.com/files/gtest-1.3.0.tar.bz2>

### 三、编译

下载解压后，里面有个 msvc 目录：



使用 VS 的同学可以直接打开 msvc 里面的工程文件，如果你在使用的是 VS2005 或是 VS2008，打开后会提示你升级，升完级后，我们直接编译里面的“gtest”工程，可以直接编过的。

这里要提醒一下的是，如果你升级为 VS2008 的工程，那么你的测试 Demo 最好也是 VS2008 工程，不然你会发现很郁闷，你的 Demo 怎么也编不过，我也曾折腾了好久，当时我升级为了 VS2008 工程，结

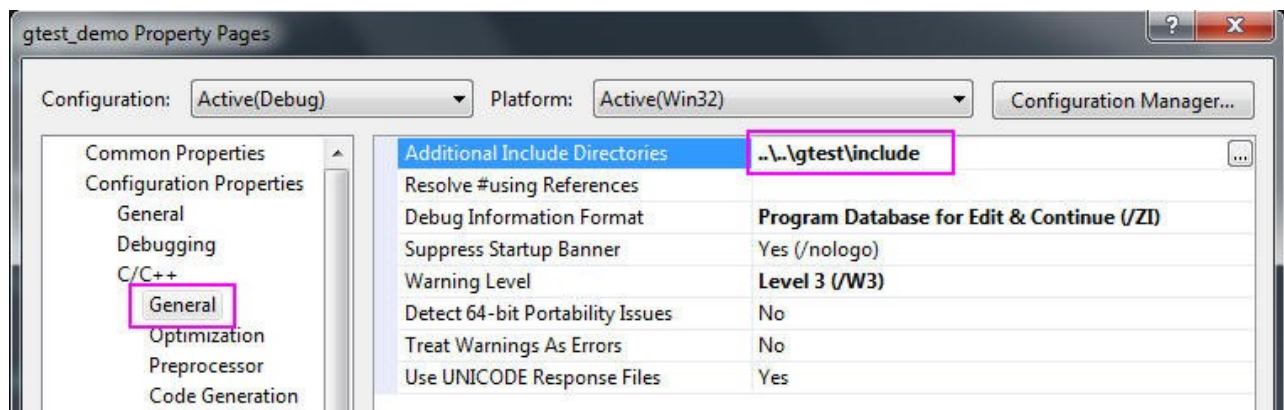
果我使用 VS2005 工程建 Demo，死活编不过。(这里有人误解了，并不是说只能在 VS2008 中编译，在 VS2005 中同样可以。如果要编译 VS2005 版本，最好保证 gtest 和你的测试工程都使用 VS2005 工程。)

编译之后，在 msvc 里面的 Debug 或是 Release 目录里看到编译出来的 gtestd.lib 或是 gtest.lib 文件。

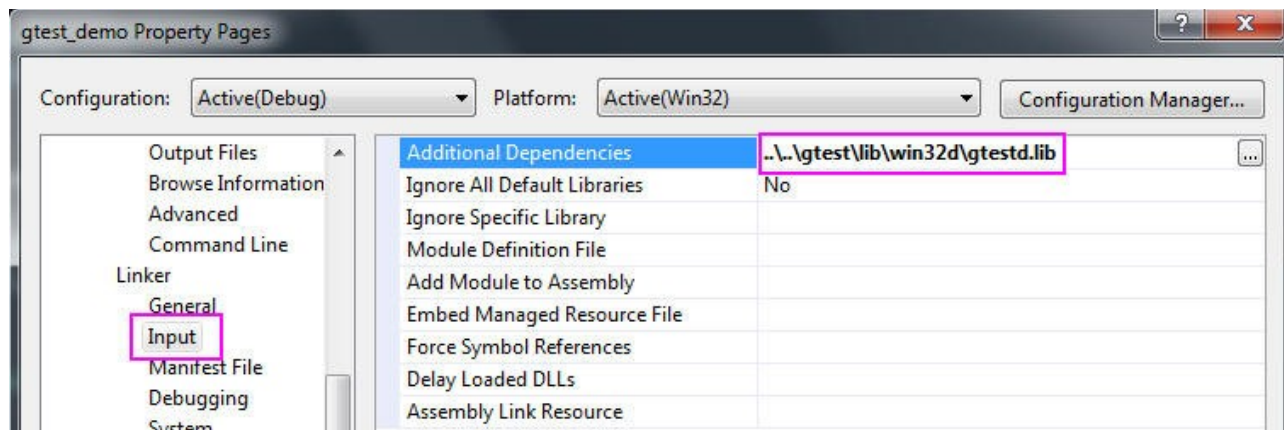
## 四、第一个 Demo

下面我们开始建立我们的第一个 Demo 了，假如之前使用的 VS2008 编译的 gtest，那么，我们在 VS2008 中，新建一个 Win32 Console Application。接着就是设置工程属性，总结如下：

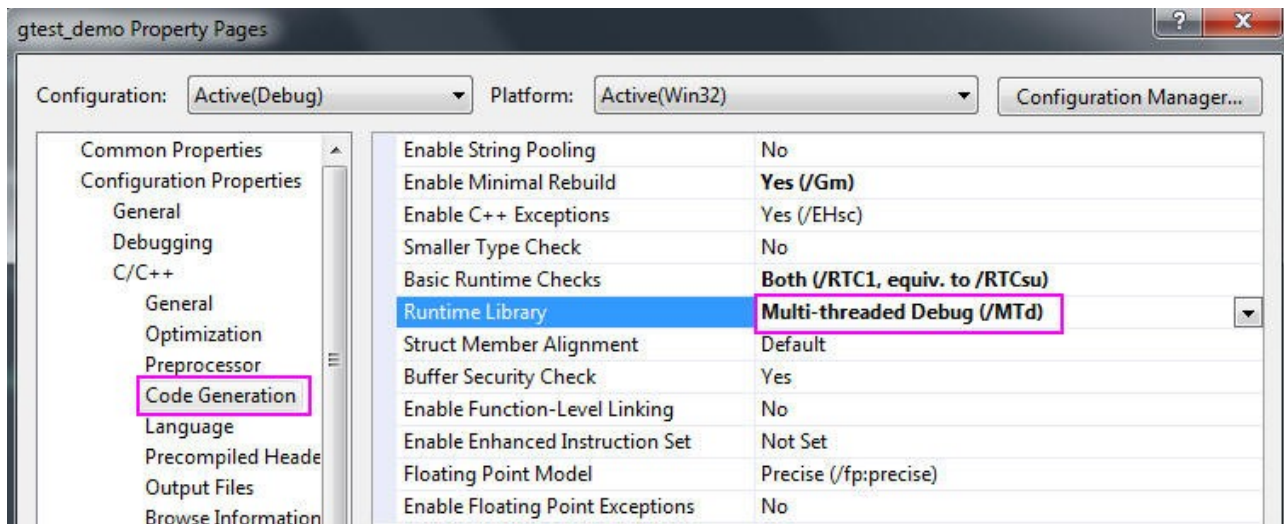
### 1. 设置 gtest 头文件路径



### 2. 设置 gtest.lib 路径



### 3. Runtime Library 设置



如果是 Release 版本，Runtime Library 设为 /MT。当然，其实你也可以选择动态链接 (/MD)，前提是你之前编译的 gtest 也使用了同样是 /MD 选项。

工程设置好了后，我们来编写一个最简单测试案例试试，我们先来写一个被测试函数：

```
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);
}
```

没错，上面的函数是用来求最大公约数的。下面我们就来编写一个简单的测试案例。

```
#include <gtest/gtest.h>
```

```
TEST(FooTest, HandleNoneZeroInput)
{
    EXPECT_EQ(2, Foo(4, 10));
    EXPECT_EQ(6, Foo(30, 18));
}
```

上面可以看到，编写一个测试案例是多么的简单。我们使用了 TEST 这个宏，它有两个参数，官方的对这两个参数的解释为：[TestCaseName, TestName]，而我对这两个参数的定义是：[TestSuiteName, TestCaseName]，在下一篇我们再来看为什么这样定义。

对检查点的检查，我们上面使用到了 EXPECT\_EQ 这个宏，这个宏用来比较两个数字是否相等。Google 还包装了一系列 EXPECT\_\* 和 ASSERT\_\* 的宏，而 EXPECT 系列和 ASSERT 系列的区别是：

1. EXPECT\_\* 失败时，案例继续往下执行。
2. ASSERT\_\* 失败时，直接在当前函数中返回，当前函数中 ASSERT\_\* 后面的语句将不会执行。

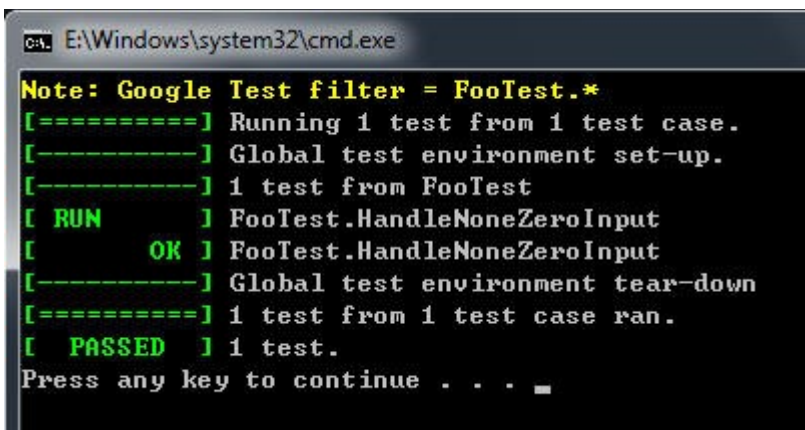
在下一篇，我们再来具体讨论这些断言宏。为了让我们的案例运行起来，我们还需要在 main 函数中添加如下代码：

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

“testing::InitGoogleTest(&argc, argv);”：gtest 的测试案例允许接收一系列的命令行参数，因此，我们将命令行参数传递给 gtest，进行一些初始化操作。gtest 的命令行参数非常丰富，在后面我们也会详细了解到。

“RUN\_ALL\_TESTS()”：运行所有测试案例

OK，一切就绪了，我们直接运行案例试试（一片绿色，非常爽）：



## 五、总结

本篇内容确实是非常的初级，目的是让从来没有接触过 gtest 的同学了解 gtest 最基本的使用。gtest 还有很多更高级的使用方法，我们将会在后面讨论。总结本篇的内容的话：

1. 使用 VS 编译 gtest.lib 文件
2. 设置测试工程的属性（头文件，lib 文件，/MT 参数（和编译 gtest 时使用一样的参数就行了））

3. 使用 TEST 宏开始一个测试案例，使用 EXPECT\_\*,ASSER\_\*系列设置检查点。

4. 在 Main 函数中初始化环境，再使用 RUN\_ALL\_TEST()宏运行测试案例。

优点：

1. 我们的测试案例本身就是一个 exe 工程，编译之后可以直接运行，非常的方便。

2. 编写测试案例变的非常简单（使用一些简单的宏如 TEST），让我们将更多精力花在案例的设计和编写上。

3. 提供了强大丰富的断言的宏，用于对各种不同检查点的检查。

4. 提高了丰富的命令行参数对案例运行进行一系列的设置。

# 玩转 Google 开源 C++ 单元测试框架 Google Test 系列(gtest)之二 - 断言

## 一、前言

这篇文章主要总结 gtest 中的所有断言相关的宏。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。

## 二、示例

```
// int 型比较，预期值：3，实际值：Add(1, 2)
EXPECT_EQ(3, Add(1, 2))
```

假如你的 Add(1, 2) 结果为 4 的话，会在结果中输出：

```
g:\myproject\c++\gtestdemo\gtestdemo\gtestdemo.cpp(16): error: Value of: Add(1, 2)
  Actual: 4
Expected:3
```

如果是将结果输出到 xml 里的话，将输出：

```
<testcase name="Demo" status="run" time="0" classname="AddTest">
  <failure message="Value of: Add(1, 2)  Actual: 4 Expected: 3" type=""><![CDATA[g:\myproject\c++\gtestdemo\gtestdemo\gtestdemo.cpp:16
Value of: Add(1, 2)
  Actual: 4
Expected: 3]]></failure>
</testcase>
```

如果你对自动输出的出错信息不满意的话，你还可以通过操作符<<将一些自定义的信息输出，通常，这对于调试或是对一些检查点的补充说明来说，非常有用！

下面举个例子：

如果不使用<<操作符自定义输出的话：

```
for (int i = 0; i < x.size(); ++i)
{
    EXPECT_EQ(x[i], y[i]);
}
```

看到的结果将是这样的，你根本不知道出错时 i 等于几：

```
g:\myproject\c++\gtestdemo\gtestdemo\gtestdemo.cpp(25): error: Value of: y[i]
  Actual: 4
```

Expected: x[i]

Which is: 3

如果使用<<操作符将一些重要信息输出的话：

```
for (int i = 0; i < x.size(); ++i)
{
    EXPECT_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;
}
```

从输出结果中就可以定位到在 i = 2 时出现了错误。这样的输出结果看起来更加有用，容易理解：

g:\myproject\c++\gtestdemo\gtestdemo\gtestdemo.cpp(25): error: Value of: y[i]

Actual: 4

Expected: x[i]

Which is: 3

Vectors x and y differ at index 2

### 三、布尔值检查

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

### 四、数值型数据检查

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

### 五、字符串检查

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_STREQ(<i>expected_str</i>, <i>actual_str</i>);</code>	<code>EXPECT_STREQ(<i>expected_str</i>, <i>actual_str</i>);</code>	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
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。下面是几个例子：

```
TEST(StringCmpTest, Demo)
{
    char* pszCoderZh = "CoderZh";
    wchar_t* wszCoderZh = L"CoderZh";
    std::string strCoderZh = "CoderZh";
    std::wstring wstrCoderZh = L"CoderZh";

    EXPECT_STREQ("CoderZh", pszCoderZh);
    EXPECT_STREQ(L"CoderZh", wszCoderZh);

    EXPECT_STRNE("CnBlogs", pszCoderZh);
    EXPECT_STRNE(L"CnBlogs", wszCoderZh);

    EXPECT_STRCASEEQ("coderzh", pszCoderZh);
    //EXPECT_STRCASEEQ(L"coderzh", wszCoderZh); 不支持

    EXPECT_STREQ("CoderZh", strCoderZh.c_str());
    EXPECT_STREQ(L"CoderZh", wstrCoderZh.c_str());
}
```

## 六、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( )；

```
TEST(ExplicitTest, Demo)
{
    ADD_FAILURE() << "Sorry"; // None Fatal Asserton, 继续往下执行。
```



```
//FAIL(); // Fatal Assertion, 不往下执行该案例。
```

```
SUCCEED();  
}
```

## 七、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), char*);  
}
```

## 八、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,</code>	<code>EXPECT_PRED2(pred2, val1,</code>	<i>pred2(val1, val2)</i> returns

<code>val2);</code>	<code>val2);</code>	<code>true</code>
<code>...</code>	<code>...</code>	<code>...</code>

Google 人说了，他们只提供 $\leq 5$ 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
<code>ASSERT_PRED_FORMAT1(pred_format1, val1);</code>	<code>EXPECT_PRED_FORMAT1(pred_format1, val1);</code>	<code>pred_format1(val1)</code> is successful
<code>ASSERT_PRED_FORMAT2(pred_format2, val1, val2);</code>	<code>EXPECT_PRED_FORMAT2(pred_format2, val1, val2);</code>	<code>pred_format2(val1, val2)</code> is successful
<code>...</code>	<code>...</code>	

用法示例：

```
testing::AssertionResult AssertFoo(const char* m_expr, const char* n_expr, const char* k_expr, 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。

九、浮点型检查

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

对相近的两个数比较：

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_NEAR( <i>val1</i> , <i>val2</i> , <i>abs_error</i> );	EXPECT_NEAR( <i>val1</i> , <i>val2</i> , <i>abs_error</i> );	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);
```

十、Windows HRESULT assertions

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

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));
```

## 十一、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();  
}
```

## 十二、总结

本篇将常用的断言都介绍了一遍，内容比较多，有些还是很有用的。要真的到写案例的时候，也行只是一两种是最常用的，现在时知道有这么多种选择，以后才方便查询。

# 玩转 Google 开源 C++ 单元测试框架 Google Test 系列(gtest)之三 - 事件机制

## 一、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。

## 二、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();  
}
```

### 三、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 ...  
}
```

### 四、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));
}
```

## 五、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。

# 玩转 Google 开源 C++ 单元测试框架 Google Test 系列(gtest)之四 - 参数化

## 一、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。

## 二、旧的方案

为了对比，我还是把旧的方案提一下。首先我先把被测函数 IsPrime 帖过来(在 gtest 的 example1.cc 中)，这个函数是用来判断传入的数值是否为质数的。

```
// Returns true iff n is a prime number.
bool IsPrime(int n)
{
    // Trivial case 1: small numbers
    if (n <= 1) return false;

    // Trivial case 2: even numbers
    if (n % 2 == 0) return n == 2;

    // Now, we have that n is odd and n >= 3.

    // Try to divide n by every odd number i, starting from 3
    for (int i = 3; ; i += 2) {
        // We only have to try i up to the square root of n
        if (i > n/i) break;

        // Now, we have i <= n/i < n.
        // If n is divisible by i, n is not prime.
        if (n % i == 0) return false;
    }
    // n has no integer factor in the range (1, n), and thus is prime.
    return true;
}
```

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，就来看看 gtest 是如何为我们解决这些问题的。

### 三、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 `INSTITUTE_TEST_CASE_P` 这宏来告诉 gtest 你要测试的参数范围：

```
INSTITUTE_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)</code> and <code>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>&lt;tr1/tuple&gt;</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，你可以重新定义宏 <code>GTEST_HAS_TR1_TUPLE=1</code>。</p>

## 四、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

## 五、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去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。

## 六、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。

# 玩转 Google 开源 C++ 单元测试框架 Google Test 系列(gtest)之五 - 死亡测试

## 一、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。

## 二、使用的宏

Fatal assertion	Nonfatal assertion	Verifies
ASSERT_DEATH( <i>statement</i> , <i>regex`</i> );	EXPECT_DEATH( <i>statement</i> , <i>regex`</i> );	<i>statement</i> crashes with the given error
ASSERT_EXIT( <i>statement</i> , <i>predicate</i> , <i>regex`</i> );	EXPECT_EXIT( <i>statement</i> , <i>predicate</i> , <i>regex`</i> );	<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。

## 三、\*\_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。

#### 四、\*\_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), "");
}
```

#### 五、\*\_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;
#ifdef 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
}
```

## 六、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
\\v	matches \v
\\w	matches any letter, _, or decimal digit
\\W	matches any character that \\w doesn't match
\\.c	matches any literal character c, which must be a punctuation
.	matches any single character except \n
A?	matches 0 or 1 occurrences of A
A*	matches 0 or many occurrences of A
A+	matches 1 or many occurrences of A
^	matches the beginning of a string (not that of each line)
\$	matches the end of a string (not that of each line)
xy	matches x followed by y

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

## 七、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();  
}
```

## 八、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。

## 九、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。

# 玩转 Google 开源 C++ 单元测试框架 Google Test 系列(gtest)之六 - 运行参数

## 一、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。

## 二、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）

### 三、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> <p>? 单个字符</p> <p>* 任意字符</p> <p>- 排除，如，-a 表示除了 a</p> <p>: 取或，如，a:b 表示 a 或 b</p> <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</p>

	这个案例
-- 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。</p> <p>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) { ... }  class DISABLED_BarTest : public testing::Test { ... };  // Tests that Bar does Xyz. TEST_F(DISABLED_BarTest, DoesXyz) { ... }</pre> 
--gtest_repeat=[COUNT]	<p>设置案例重复运行次数，非常棒的功能！比如：</p> <pre>--gtest_repeat=1000    重复执行 1000 次，即使中途出现错误。 --gtest_repeat=-1      无限次数执行。。。. --gtest_repeat=1000 --gtest_break_on_failure  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。</pre> <pre>--gtest_repeat=1000 --gtest_filter=FooBar</pre> <p>重复执行 1000 次测试案例名称为 FooBar 的案例。</p>

2. 测试案例输出

命令行参数	说明
--gtest_color=(yes no auto)	输出命令行时是否使用一些五颜六色的颜色。默认是 auto。
--gtest_print_time	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
-- gtest_output=xml[:DIRECTORY_PATH\ :FILE_PATH]	<p>将测试结果输出到一个 xml 中。</p> <p>1.--gtest_output=xml: 不指定输出路径时，默认为案例当前路径。</p>

	<p>2.--gtest_output=xml:d:\ 指定输出到某个目录</p> <p>3.--gtest_output=xml:d:\foo.xml 指定输出到 d:\foo.xml</p> <p>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。</p> <p>xml 的输出内容后面介绍吧。</p>
--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 3. 对案例的异常处理

命令行参数	说明
--gtest_break_on_failure	调试模式下，当案例失败时停止，方便调试
--gtest_throw_on_failure	当案例失败时以 C++异常的方式抛出
--gtest_catch_exceptions	<p>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异常，很可能会弹出一个对话框，这非常的不友好，同时也阻碍了测试案例的运行。</p> <p>如果想不弹这个框，可以通过设置这个参数来实现。如将 --gtest_catch_exceptions 设置为一个非零的数。</p> <p>注意：这个参数只在 Windows 下有效。</p>

## 四、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。

## 五、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。