Google C++ 单元测试框架——Gtest框架简介

# 一、设置一个新的测试项目

在用google test写测试项目之前，需要先编译gtest到library库并将测试与其链接。我们为一些流行的构建系统提供了构建文件: msvc/ for Visual Studio, xcode/ for Mac Xcode, make/ for GNU make, codegear/ for Borland C++ Builder.

如果你的构建系统不在这个名单上，在googletest根目录有autotools的脚本（不推荐使用）和CMakeLists.txt CMake（推荐）。你可以看看make / Makefile来了解如何编译Google Test（基本上你想在头文件中使用GTEST\_ROOT和GTEST\_ROOT/include来编译src/gtest-all.cc路径，其中GTEST\_ROOT是Google测试根目录）。

一旦你能够编译google test库，您应该为您的测试程序创建一个项目或构建目标。Make sure you have GTEST\_ROOT/include in the header search path so that the compiler can find "gtest/gtest.h" when compiling your test.把google test库加到你的测试项目中（比如：在VS中在gtest.vcproj上添加依赖）。

# 二、基本概念

当时用googletest时，首先要写断言，断言是检查条件是否为真的语句。一个断言的结果可以是成功，非致命性失败，或致命的失败。如果一个致命失败出现，它会终止当前的函数；否则程序继续正常运行。

测试使用断言验证代码的行为。如果一个测试崩溃或者有一个失败的断言，那么失败；否则成功。

一个测试用例包含一个或多个测试。 你应该将测试分组为反映测试代码结构的测试用例。当测试用例中的多个测试需要共享公共的对象和子程序时，你可以把它们放进一个test fixture class（测试夹具类）。

一个测试程序可以包含多个测试用例。

现在我们将解释如何编写测试程序，从单个断言级别开始，并构建测试和测试用例。

# 三、断言

Google Test断言是类似于函数调用的宏。你可以通过对其行为进行断言来测试类或函数。当断言失败时，Google Test会打印断言的源文件和行号位置以及失败消息。您还可以提供自定义失败消息，该消息将附加到Google测试的信息中。

断言是成对的，测试同一件事，但对当前函数有不同的影响。ASSERT\_ \*版本在失败时会生成致命错误，并中止当前函数。EXPECT\_ \*版本生成非致命性故障，不会中止当前函数。通常优先使用EXPECT\_ \*，因为它们允许在测试中报告多个故障。但是，如果失败时函数继续运行没有意义，则应使用ASSERT\_ \*。

因为失败的ASSERT\_ \*立即从当前函数返回，可能跳过其后的清理代码，它可能导致资源泄漏。根据泄漏的性质，它可能值得修复也可能不值得修复--所以把这个记在心里，如果你有一个堆检测错误需要注意是什么导致的。

要提供自定义失败消息，只需使用<<运算符或一系列此类运算符将其流式传输到宏中即可。一个例子：

ASSERT\_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i) {

  EXPECT\_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;

}

任何可以流式传输到ostream的东西都可以流式传输到断言宏，特别是C字符串和字符串对象。 如果一个宽字符串（Windows上的wchar\_t \*，TCHAR \*在UNICODE模式下，或者std :: wstring）被流化到一个断言，当打印时它将被转换为UTF-8。

# 四、基本断言

这些断言做基本的真/假条件测试。

| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| --- | --- | --- |
| ASSERT\_TRUE(*condition*); | EXPECT\_TRUE(*condition*); | *condition* is true |
| ASSERT\_FALSE(*condition*); | EXPECT\_FALSE(*condition*); | *condition* is false |

当它们失败时，ASSERT\_ \*产生致命失败并从当前函数返回，而EXPECT\_ \*产生非致命失败，允许函数继续运行。 在任一情况下，断言失败意味着其包含的测试失败。

# 五、二进制比较

本节介绍比较两个值的断言。

| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| --- | --- | --- |
| ASSERT\_EQ(*val1*,*val2*); | EXPECT\_EQ(*val1*,*val2*); | *val1* == *val2* |
| ASSERT\_NE(*val1*,*val2*); | EXPECT\_NE(*val1*,*val2*); | *val1* != *val2* |
| ASSERT\_LT(*val1*,*val2*); | EXPECT\_LT(*val1*,*val2*); | *val1* < *val2* |
| ASSERT\_LE(*val1*,*val2*); | EXPECT\_LE(*val1*,*val2*); | *val1* <= *val2* |
| ASSERT\_GT(*val1*,*val2*); | EXPECT\_GT(*val1*,*val2*); | *val1* > *val2* |
| ASSERT\_GE(*val1*,*val2*); | EXPECT\_GE(*val1*,*val2*); | *val1* >= *val2* |

在发生故障时，Google测试同时打印val1和val2。

值参数通过断言的比较运算符必须可以比较，否则会出现编译错误。我们曾经要求参数支持<<运算符，用于流传输到ostream，但从v1.6.0它不再需要（如果支持<<，则会在断言失败时调用它来打印参数;否则Google Test将尝试以最佳方式打印它们。有关更多详细信息和如何自定义参数的打印，请参阅此Google Mock [recipe](https://github.com/google/googletest/blob/master/googlemock/docs/CookBook.md#teaching-google-mock-how-to-print-your-values).。

这些断言可以使用用户定义的类型，但是只有当你定义了相应的比较运算符（例如==，<，etc）。如果定义了相应的操作符，则更喜欢使用ASSERT\_ \*()宏，因为它们不仅会打印比较结果，而且还会打印出两个操作数。

参数总是只计算一次。因此，参数有副作用没关系。然而，与任何普通的C /C++函数一样，参数的求值顺序是未定义的（**即编译器可以自由选择任何顺序**），你的代码不应该依赖于任何特定的参数求值顺序。

ASSERT\_EQ()指针的指针相等。如果在两个C字符串上使用，它会测试它们是否在同一个内存位置，而不是它们具有相同的值。因此，如果你想比较C字符串（例如const char \*）的值，使用ASSERT\_STREQ（），稍后将会描述。特别地，要断言C字符串为NULL，请使用ASSERT\_STREQ（NULL，c\_string）。但是，要比较两个字符串对象，应该使用ASSERT\_EQ。

本节中的宏适用于窄和宽字符串对象（string和wstring）。

历史记录：2016年2月之前\* \_EQ有一个约定，称为ASSERT\_EQ（expected,actual），所以很多现有的代码使用这个顺序。 现在\* \_EQ以相同的方式处理这两个参数。

# 六、字符串比较

该组中的断言比较两个C字符串的值。 如果要比较两个字符串对象，请改用EXPECT\_EQ，EXPECT\_NE和etc。

| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| --- | --- | --- |
| ASSERT\_STREQ(*str1*,*str2*); | EXPECT\_STREQ(*str1*,\_str\_2); | the two C strings have the same content |
| ASSERT\_STRNE(*str1*,*str2*); | EXPECT\_STRNE(*str1*,*str2*); | the two C strings have different content |
| ASSERT\_STRCASEEQ(*str1*,*str2*); | EXPECT\_STRCASEEQ(*str1*,*str2*); | the two C strings have the same content, ignoring case(忽略大小写) |
| ASSERT\_STRCASENE(*str1*,*str2*); | EXPECT\_STRCASENE(*str1*,*str2*); | the two C strings have different content, ignoring case |

注意，断言名中的“CASE”表示忽略大小写。

\* STREQ \*和\* STRNE \*也接受宽C字符串（wchar\_t \*）。 如果两个宽字符串的比较失败，它们的值将打印为UTF-8窄字符串。

NULL指针和空字符串被认为是不同的。

可用性：Linux，Windows，Mac。

# 七、简单测试

创建测试：

1.使用TEST（）宏来定义和命名测试函数，这些是不返回值的普通C++函数。  
  2.在此函数中，连同要包括的任何有效的C++语句，使用各种Google Test断言来检查值。  
  3.测试的结果由断言确定; 如果测试中的任何断言失败（致命或非致命），或者如果测试崩溃，则整个测试失败。 否则，它成功。

|  |  |
| --- | --- |
| 1  2  3 | TEST(test\_case\_name,test\_name) {   ... test body ...  } |

TEST（）参数从一般到特定。 第一个参数是测试用例的名称，第二个参数是测试用例中的测试名称。 这两个名称必须是有效的C ++标识符，并且它们不应包含下划线（\_）。 测试的全名由其包含的测试用例及其个人名称组成。来自不同测试用例的测试可以具有相同的个人名称。

例如，让我们使用一个简单的整数函数：

|  |  |
| --- | --- |
| 1 | int Factorial(int n); // Returns the factorial of n；n！ |

此函数的测试用例可能如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // Tests factorial of 0.  TEST(FactorialTest, HandlesZeroInput) {    EXPECT\_EQ(1, Factorial(0));  }    // Tests factorial of positive numbers.  TEST(FactorialTest, HandlesPositiveInput) {    EXPECT\_EQ(1, Factorial(1));    EXPECT\_EQ(2, Factorial(2));    EXPECT\_EQ(6, Factorial(3));    EXPECT\_EQ(40320, Factorial(8));  } |

Google Test通过测试用例对测试结果进行分组，因此逻辑相关的测试应该在同一测试用例中; 换句话说，它们的TEST（）的第一个参数应该是相同的。 在上面的例子中，我们有两个测试，HandlesZeroInput和HandlesPositiveInput，属于同一个测试用例FactorialTest。

# 八、测试夹具

如果你发现自己写了两个或更多的测试来操作类似的数据，你可以使用测试夹具。它允许您为几个不同的测试重复使用相同的对象配置。

要创建夹具，只需：

  1.从:: testing :: Test派生一个类。 使用protected：或public：开始它的主体，因为我们想从子类访问fixture成员。

  2.在类中，声明你打算使用的任何对象。

3.如果需要，可以编写默认构造函数或SetUp（）函数来为每个测试准备对象。

4.如果需要，写一个析构函数或TearDown（）函数来释放你在SetUp（）中分配的任何资源。 要学习什么时候应该使用构造函数/析构函数，当你应该使用SetUp()/TearDown()时。  
  5.如果需要，定义要分享的测试的子程序。

当使用夹具时，使用TEST\_F（）而不是TEST（），因为它允许您访问测试夹具中的对象和子程序：

|  |  |
| --- | --- |
| 1  2  3 | TEST\_F(test\_case\_name, test\_name) {   ... test body ...  } |

和TEST（）一样，第一个参数是测试用例名，但是对于TEST\_F（）第一个参数必须是测试夹具类的名称。你可能猜到了：\_F是夹具。

不幸的是，C ++宏系统不允许我们创建一个可以处理两种类型的测试的宏。 使用错误的宏会导致编译器错误。

另外，在TEST\_F（）中使用它之前，你必须首先定义一个测试夹具类，否则将得到编译器错误“virtual outside class declaration”。

对于使用TEST\_F（）定义的每个测试，Google Test将：

1.在运行时创建一个新的测试夹具

2.立即通过SetUp（）初始化，

3.运行测试

4.通过调用TearDown（）清除

5.删除测试夹具。 请注意，同一测试用例中的不同测试具有不同的测试夹具对象，Google测试始终会删除测试夹具，然后再创建下一个测试夹具。 Google测试不会为多个测试重复使用相同的测试夹具。一个测试对夹具的任何更改不会影响其他测试。

例如，让我们为名为Queue的FIFO队列类编写测试，它有以下接口：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | template <typename E> // E is the element type.  class Queue {   public:    Queue();    void Enqueue(const E& element);    E\* Dequeue(); // Returns NULL if the queue is empty.    size\_t size() const;    ...  }; |

首先定义一个夹具类。按照惯例，你应该给它名称FooTest，其中Foo是被测试的类。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class QueueTest:public ::testing::Test {   protected:    virtual void SetUp() {      q1\_.Enqueue(1);      q2\_.Enqueue(2);      q2\_.Enqueue(3);    }      // virtual void TearDown() {}      Queue<int> q0\_;    Queue<int> q1\_;    Queue<int> q2\_;  }; |

在这种情况下，不需要TearDown（），因为我们不必在每次测试后清理，除了析构函数已经做了什么。

现在我们将使用TEST\_F（）和这个夹具编写测试。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | TEST\_F(QueueTest, IsEmptyInitially) {    EXPECT\_EQ(0, q0\_.size());  }    TEST\_F(QueueTest, DequeueWorks) {    int\* n = q0\_.Dequeue();    EXPECT\_EQ(NULL, n);      n = q1\_.Dequeue();    ASSERT\_TRUE(n != NULL);    EXPECT\_EQ(1, \*n);    EXPECT\_EQ(0, q1\_.size());    delete n;      n = q2\_.Dequeue();    ASSERT\_TRUE(n != NULL);    EXPECT\_EQ(2, \*n);    EXPECT\_EQ(1, q2\_.size());    delete n;  } |

上面使用ASSERT\_ \*和EXPECT\_ \*断言。 经验法则( The rule of thumb )是当你希望测试在断言失败后继续显示更多错误时使用EXPECT\_ \*，或是在失败后继续使用ASSERT\_ \*没有意义。 例如，Dequeue测试中的第二个断言是ASSERT\_TRUE（n！= NULL），因为我们需要稍后解引用指针n，这将导致n为NULL时的segfault。

当这些测试运行时，会发生以下情况：

1.Google Test构造了一个QueueTest对象（我们称之为t1）。

2.t1.SetUp（）初始化t1。

3.第一个测试（IsEmptyInitially）在t1上运行。

4.t1.TearDown（）在测试完成后清理。

5.t1被析构。

6.以上步骤在另一个QueueTest对象上重复，这次运行DequeueWorks测试。

# 九、调用测试

TEST()和TEST\_F()用Google Test隐式注册他们的测试。 因此，与许多其他C ++测试框架不同，您不必重新列出所有定义的测试以运行它们。

定义测试后，可以使用RUN\_ALL\_TESTS()运行它们，如果所有测试成功则返回0，否则返回1。请注意，**RUN\_ALL\_TESTS()运行链接单元中的所有测试**-它们可以来自不同的测试用例，甚至是不同的源文件。

调用时，RUN\_ALL\_TESTS()宏：

1. 保存所有Google测试标记的状态。

2. 为第一个测试创建测试夹具对象。

3. 通过SetUp（）初始化它。

4. 在fixture对象上运行测试。

5. 通过TearDown（）清除夹具

6. 删除夹具。

7. 恢复所有Google测试标志的状态。

8. 重复上述步骤进行下一个测试，直到所有测试运行结束。

此外，如果测试夹具的构造函数在步骤2中产生致命故障，则步骤3-5没有意义，因此它们被跳过。类似地，如果步骤3产生致命故障，则将跳过步骤4。

**重要：**您不能忽略RUN\_ALL\_TESTS()的返回值，否则gcc将给您一个编译器错误。此设计的基本原理是自动测试服务基于其退出代码而不是其stdout/ stderr输出来确定测试是否已通过；因此您的**main()函数必须返回RUN\_ALL\_TESTS()的值**。

此外，您应该只调用一次RUN\_ALL\_TESTS()。多次调用它会与一些高级Google测试功能（例如线程安全死亡测试）冲突，因此不受支持。

# 十、写Main()函数

你可以从这个样板开始：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | #include "this/package/foo.h"  #include "gtest/gtest.h"    namespace {  // The fixture for testing class Foo.  class FooTest : public ::testing::Test {   protected:    // You can remove any or all of the following functions if its body    // is empty.      FooTest() {      // You can do set-up work for each test here.    }      virtual ~FooTest() {      // You can do clean-up work that doesn't throw exceptions here.    }      // If the constructor and destructor are not enough for setting up    // and cleaning up each test, you can define the following methods:      virtual void SetUp() {      // Code here will be called immediately after the constructor (right      // before each test).    }      virtual void TearDown() {      // Code here will be called immediately after each test (right      // before the destructor).    }      // Objects declared here can be used by all tests in the test case for Foo.  };    // Tests that the Foo::Bar() method does Abc.  TEST\_F(FooTest, MethodBarDoesAbc) {    const string input\_filepath = "this/package/testdata/myinputfile.dat";    const string output\_filepath = "this/package/testdata/myoutputfile.dat";    Foo f;    EXPECT\_EQ(0, f.Bar(input\_filepath, output\_filepath));  }    // Tests that Foo does Xyz.  TEST\_F(FooTest, DoesXyz) {    // Exercises the Xyz feature of Foo.  }    }  // namespace    int main(int argc, char \*\*argv) {    ::testing::InitGoogleTest(&argc, argv);    return RUN\_ALL\_TESTS();  } |

:: testing :: InitGoogleTest（）函数解析Google测试标志的命令行，并删除所有已识别的标志。这允许用户通过各种标志控制测试程序的行为，我们将在[AdvancedGuide](https://github.com/google/googletest/blob/master/googletest/docs/AdvancedGuide.md)中介绍。在调用RUN\_ALL\_TESTS()之前必须调用此函数，否则标志将无法正确初始化。

在Windows上，InitGoogleTest()也适用于宽字符串，因此它也可以在以UNICODE模式编译的程序中使用。

但也许你认为编写所有这些main()函数是太多的工作？ 我们完全同意你的看法，这就是为什么Google Test提供了main（）的基本实现。 如果它适合你的需要，然后只是链接你的测试与gtest\_main库。