GoogleTest使用笔记

# 准备

GoogleTest的Github地址：

<https://github.com/google/googletest.git>

CMake的下载地址：<https://cmake.org/>

${GTEST\_DIR}：Google Test目录

# Google C++ 测试框架

1. 测试应该是独立的和可重复的。通过其他测试调试成功或失败的测试是一种痛苦。Google C ++测试框架通过在不同的对象上运行每个测试来隔离测试。当测试失败时，Google C ++测试框架允许您将其单独运行以进行快速调试。
2. 测试应该组织良好，并反映测试代码的结构。 Google C ++测试框架将相关测试分组到可以共享数据和子程序的测试用例中。 这种常见的模式很容易识别，并使测试易于维护。 当人们切换项目并开始在新的代码库上工作时，这种一致性特别有用。
3. 测试应该是便携式和可重复使用的。 开源社区有很多平台无关的代码，其测试也应该是平台中立的。 Google C ++测试框架适用于不同的操作系统，不同的编译器（gcc，MSVC等），有或没有异常，所以Google C ++ Testing Framework测试可以轻松地处理各种配置。
4. 当测试失败时，应尽可能多地提供有关问题的信息。 Google C ++测试框架不会在第一次测试失败时停止。 相反，它只会停止当前的测试，并继续下一个测试。 您还可以设置测试报告非致命故障(non-fatal)，之后继续进行当前测试。 因此，您可以在单个运行-编辑-编译周期中检测并修复多个错误。
5. 测试框架应该把测试作者从家务中解放出来，让他们专注于测试内容。 Google C ++测试框架自动跟踪定义的所有测试，并且不需要用户枚举它们就可以运行它们。
6. 测试应该很快。 借助Google C ++测试框架，您可以在测试之间重复使用共享资源，并且仅支付一次设置/拆除费用，而不会使测试互相依赖。

## 谨防命名

由于“Test”，“Test Case”和“Test Suite”这几个术语的定义不同，可能会有一些混淆的想法，所以要小心这些误解。

历史上，Google C++测试框架开始使用术语“Test Case”来分组相关测试，而包括国际软件测试认证委员会（ISTQB）和各种软件质量教科书在内的当前出版物使用术语“Test Suite”。

在Google C++测试框架中使用的相关术语“Test”对应于ISTQB和其他资料的术语“Test Case”。

不幸的是，在整个Google C ++测试框架中用“Test Suite”替代“Test Case”这个术语并不容易，因为TestCase是各个地方的公共API的一部分。

|  |  |  |
| --- | --- | --- |
| 意义 | Google Test术语 | ISTQB术语 |
| 用特定的输入值来执行特定的程序路径，并验证结果 | Test | Test Case |
| 与一个组件相关的一组测试 | Test Case | Test Suite |

## 建立一个新的测试项目

Google Test目录中提供了常用平台的编译文件。使用时需要包含Google Test的头文件，并连接到库文件。

## 基本概念

通过编写断言来使用Google Test。断言是检查条件是否成立的语句。 断言的结果可能是成功(success)，非致命的失败(nonfatal failure)或致命的失败(fatal failure)。 如果发生致命失败，则中止当前功能; 否则程序正常继续。

Tests使用断言来验证测试代码的行为。 如果一个测试崩溃或断言失败，则失败; 否则成功。

Test Case包含一个或多个Test。 应该将test分组到Test Case中，以反映测试代码的结构。 当测试用例中的多个测试需要共享通用对象和子程序时，可以将它们放入测试夹具类(text fixture class)中。

一个测试程序可以包含多个测试用例。

## 断言(Assertions)

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。 也可提供自定义失败消息，并将其附加到Google测试消息中。

断言成对地测试相同的事物，但对当前函数有不同的影响。 ASSERT\_ \*版本失败时会生成致命故障，并中止当前的功能。 EXPECT\_ \*版本会产生非致命的故障，不会中止当前的功能。 通常，EXPECT\_ \*是首选，因为它们允许在测试中报告多个故障。 但是，如果在有问题的断言失败时无法继续，则应该使用ASSERT\_ \*。

由于失败的ASSERT\_ \*立即从当前函数返回，可能会跳过清理后的代码，这可能会导致空间泄漏。根据泄漏的性质，它可能会也可能不值得修复 - 所以如果除了断言错误之外还有堆检查器错误，请记住这一点。

要提供自定义失败消息，只需使用<<运算符或这些运算符的序列将其流式传输到宏。 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一个断言宏 - 特别是C字符串和string对象。 如果将一个宽字符串（wchar\_t \*，Windows上的UNICODE模式的TCHAR \*或std :: wstring）流式传输到一个断言，打印时将转换为UTF-8。

### 基本断言

这些断言做基本的真/假状态测试。

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

### 二进制比较

本节介绍了比较两个值的断言。

|  |  |  |
| --- | --- | --- |
| **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 |

如果发生错误，Google测试会同时打印val1和val2。

参数值必须可以通过断言的比较运算符进行比较，否则会出现编译器错误。我们以前要求参数支持到ostream的<<流操作符，但是从v1.6.0开始不再需要（如果<<被支持，当断言失败的时候会被调用来打印参数;否则Google Test将尝试以最好的方式打印它们）。

这些断言可以使用用户定义的类型，但只有在定义相应的比较运算符（例如==，<等）时才可以。如果定义了相应的运算符，则首选使用ASSERT \_ \*（）宏，因为它们不仅会打印出比较结果，还会打印两个操作数。

参数总是被计算一次。因此，参数有副作用是可以的。但是，与任何普通的C / C ++函数一样，参数的计算顺序是未定义的（即编译器可以自由选择任何顺序），而且代码不应该依赖于任何特定的参数计算顺序。

ASSERT\_EQ（）在指针上做指针相等比较。如果在两个C字符串上使用，它将测试它们是否在相同的内存位置，而不是如果它们具有相同的值。因此，如果要按值比较C字符串（例如，const char \*），请使用ASSERT\_STREQ（），稍后将对其进行介绍。特别是，要断言C字符串是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。

本节中的宏使用窄字符串对象（string和wstring）。

### 字符串比较

这个组中的断言比较两个C字符串。 如果要比较两个字符串对象，请使用EXPECT\_EQ，EXPECT\_NE等。

|  |  |  |
| --- | --- | --- |
| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| ASSERT\_STREQ(str1,str2); | EXPECT\_STREQ(str1,str2); | 两个字符串有相同的内容 |
| ASSERT\_STRNE(str1,str2); | EXPECT\_STRNE(str1,str2); | 两个字符串有不同的内容 |
| ASSERT\_STRCASEEQ(str1,str2); | EXPECT\_STRCASEEQ(str1,str2); | 两个字符串有相同的内容，忽略大小写 |
| ASSERT\_STRCASENE(str1,str2); | EXPECT\_STRCASENE(str1,str2); | 两个字符串有不同的内容，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，则它们的值将被打印为UTF-8窄字符串。

空指针和空字符串被认为是不同的。

## 简单的测试

创建一个测试的步骤：

1. 通过TEST()宏定义和命名一个测试函数。它们是没有返回值的C++函数；
2. 在这个函数中，除了要包含的任何有效的C ++语句之外，还可以使用各种Google Test断言来检查值。
3. 测试的结果由断言决定; 如果测试中的任何断言失败（无论是致命的还是非致命的），或者测试崩溃，则整个测试失败。 否则，它会成功。

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。 这两个名称都必须是有效的C ++标识符，并且不能包含下划线（\_）。 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。

一个简单的示例：

int Factorial(int n); // Returns the factorial of n

// 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测试会将测试结果按测试用例进行分组，因此逻辑相关的测试应该在相同的测试用例中; 换句话说，他们的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。

## 测试套件：使用相同的数据配置进行多个测试

如果发现自己正在编写两个或更多以类似数据进行操作的测试，则可以使用测试夹具(test fixture)。 它允许你重复使用相同配置的对象进行多个不同的测试。

要创建一个夹具(test fixture)，只需：

1. 从:: testing :: Test派生一个类。 并将权限设置为protected或public，因为我们要从子类访问fixture成员。
2. 在类中，声明你打算使用的任何对象。
3. 如有必要，编写一个默认的构造函数或SetUp（）函数为每个测试准备对象。
4. 如有必要，编写一个析构函数或TearDown（）函数来释放在SetUp（）中分配的所有资源。
5. 如果需要，为测试定义共享的子程序。

使用夹具时，请使用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：

TEST\_F(test\_case\_name, test\_name) {

... test body ...

}

像TEST（）一样，第一个参数是测试用例名称，但是对于TEST\_F（），它必须是测试夹具类的名称。

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，否则你会得到编译器错误“virtual outside class declaration”。

对于使用TEST\_F（）定义的每个测试，Google Test将会：

1. 在运行时创建一个新的测试夹具。
2. 立即通过SetUp（）初始化它。
3. 运行测试。
4. 通过调用TearDown（）来清理。
5. 删除测试夹具。 请注意，同一个测试用例中的不同测试具有不同的测试夹具对象，Google Test在创建下一个测试夹具之前总是删除测试夹具。 Google测试不会重复使用相同的测试夹具进行多个测试。 一个测试对夹具进行的任何更改都不会影响其他测试。

一个简单的示例：

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;

...

};

定义一个fixture类：

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，TearDown（）是不需要的，因为除了析构函数已经做了什么以外，我们不必在每次测试之后清理。

现在我们将使用TEST\_F（）和这个fixture来编写测试。

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;

}

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（）在测试结束后清理。
5. t1被破坏。
6. 上述步骤在另一个QueueTest对象上重复，这次运行DequeueWorks测试。

## 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。

在定义你的测试之后，你可以用RUN\_ALL\_TESTS（）来运行它们，如果所有的测试都成功的话返回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，甚至不同的源文件。

当被调用时，RUN\_ALL\_TESTS（）宏：

1. 保存所有Google测试标志的状态。
2. 为第一个测试创建一个测试夹具对象。
3. 通过SetUp（）初始化它。
4. 在夹具对象上运行测试。
5. 通过TearDown（）清理夹具。
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，直到所有测试都已经运行。

另外，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。同样，如果第3步生成致命故障，第4步将被跳过。

重要：你不能忽略RUN\_ALL\_TESTS（）的返回值，否则gcc会给你一个编译器错误。这种设计的基本原理是，自动化测试服务根据退出代码而不是stdout / stderr输出来确定测试是否已经通过;因此你的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。

## 写main()函数

#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，并删除所有可识别的标志。 这允许用户通过各种标志来控制测试程序的行为，我们将在AdvancedGuide中介绍。 您必须在调用RUN\_ALL\_TESTS（）之前调用此函数，否则标志将无法正确初始化。

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

但是也许你认为编写所有main（）函数的工作太多了？ 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，那么只要把你的测试与gtest\_main库联系起来，你就可以走了。

### 对Visual C++用户的忠告

如果将测试放到一个库中，并且main（）函数位于不同的库或.exe文件中，那么这些测试将无法运行。 原因是Visual C ++中的一个错误。 当您定义测试时，Google Test会创建特定的静态对象来注册它们。 这些对象没有从其他地方引用，但是它们的构造函数仍然应该运行。 当Visual C ++链接程序发现库中没有任何内容从其他位置引用时，它将引发该库。 您必须从您的主程序中引用您的库，以防止链接器丢弃它。 这是如何做到这一点。 库代码中的某处声明了一个函数：

\_\_declspec(dllexport) int PullInMyLibrary() { return 0; }

如果你把你的测试放在一个静态库（不是DLL）中，那么不需要\_\_declspec（dllexport）。 现在，在你的主程序中，编写一个调用该函数的代码：

int PullInMyLibrary();

static int dummy = PullInMyLibrary();