# 简介:为什么是谷歌c++测试框架?

*谷歌c++测试框架帮助您编写更好的c++测试。*

无论您使用的是Linux、Windows还是Mac，如果您编写c++代码，谷歌测试都可以帮助您。

那么，什么构成了一个好的测试，谷歌c++测试框架是如何适应的呢?我们相信:

1. 测试应该是独立的和可重复的。调试一个由于其他测试而成功或失败的测试是一件痛苦的事情。谷歌c++测试框架通过在不同的对象上运行它们来隔离测试。当测试失败时，谷歌c++测试框架允许您单独运行它，以便快速调试。
2. 测试应该组织良好，并反映测试代码的结构。谷歌c++测试框架将相关测试分组到可以共享数据和子例程的测试用例中。这种常见的模式很容易识别，并且使测试很容易维护。当人们转换项目并开始使用新的代码库时，这种一致性尤其有用。
3. 测试应该是可移植和可重用的。开源社区有很多代码是平台无关的，它的测试也应该是平台无关的。谷歌c++测试框架可以在不同的操作系统上工作，使用不同的编译器(gcc、MSVC等)，可以有例外，也可以没有例外，因此谷歌c++测试框架测试可以轻松地使用各种配置。(注意，当前版本只包含用于Linux的构建脚本—我们正在积极地处理用于其他平台的脚本。)
4. 当测试失败时，他们应该提供尽可能多的关于问题的信息。谷歌c++测试框架不会在第一次测试失败时就停止。相反，它只停止当前的测试并继续下一个测试。您还可以设置报告非致命故障的测试，然后继续当前测试。因此，您可以在一个运行-编辑-编译周期中检测并修复多个bug。
5. 测试框架应该把测试编写者从家务杂务中解放出来，让他们专注于测试内容。谷歌c++测试框架自动跟踪所有定义的测试，并且不需要用户为了运行它们而枚举它们。
6. 测试应该是快速的。使用谷歌c++测试框架，您可以跨测试重用共享资源，并且只需为设置/拆卸支付一次费用，而无需使测试彼此依赖。

由于谷歌c++测试框架是基于流行的xUnit架构的，所以如果您以前使用过JUnit或PyUnit，您就会觉得很熟悉。如果没有，你需要花大约10分钟来学习基础知识并开始。让我们走吧!

*注意:我们有时将谷歌c++测试框架非正式地称为谷歌测试。*

# 建立一个新的测试项目

要使用谷歌测试编写测试程序，您需要将谷歌测试编译到一个库中，并将您的测试与其链接。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，以及autotools脚本(不推荐)和cmaknames。在谷歌测试根目录中为CMake(推荐)添加txt。如果您的构建系统不在此列表中，您可以查看make/Makefile以了解如何编译谷歌测试(基本上您希望编译src/gtest-all。在头搜索路径中包含GTEST\_ROOT和GTEST\_ROOT/include的cc，其中GTEST\_ROOT是谷歌测试根目录)。

一旦您能够编译谷歌测试库，您应该为您的测试程序创建一个项目或构建一个目标。确保在头搜索路径中包含GTEST\_ROOT/include，以便编译器能够找到“gtest/gtest”。编译测试的时候。设置您的测试项目以链接到谷歌测试库(例如，在Visual Studio中，这是通过添加对gtest.vcproj的依赖来完成的)。

如果您仍然有问题，可以看看谷歌测试如何构建自己的测试，并将它们用作示例。

# 基本概念

在使用谷歌测试时，首先编写断言，断言是检查条件是否为真的语句。断言的结果可以是成功、非致命失败或致命失败。如果发生致命故障，它将中止当前函数;否则程序将正常运行。

*测试使用断言来验证被测试代码的行为。*如果一个测试崩溃或者有一个失败的断言，那么它就失败了;否则它成功。

一个测试用例包含一个或多个测试。您应该将测试分组到反映测试代码结构的测试用例中。当一个测试用例中的多个测试需要共享公共对象和子例程时，您可以将它们放到一个测试装置类中。

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一个测试程序，从单个断言级别开始，逐步构建到测试和测试用例。

# 断言

谷歌测试断言是类似于函数调用的宏。通过对类或函数的行为进行断言来测试它。当断言失败时，谷歌测试打印断言的源文件和行号位置，以及一条失败消息。您还可以提供一个自定义失败消息，它将被附加到谷歌测试的消息中。

断言成对出现，它们测试相同的东西，但对当前函数有不同的影响。ASSERT\_\*版本在失败时会产生致命的错误，并中止当前函数。EXPECT\_\*版本生成非致命故障，不会中止当前函数。通常，EXPECT\_\*是首选的，因为它们允许在测试中报告多个失败。但是，如果在问题中的断言失败时继续没有意义，则应该使用ASSERT\_\*。

因为一个失败的ASSERT\_\*会立即从当前函数返回，可能会跳过它之后的清理代码，这可能会导致空间泄漏。根据泄漏的性质，它可能值得修复，也可能不值得修复—因此，如果除了断言错误外还出现堆检查器错误，请记住这一点。

要提供自定义失败消息，只需使用<<操作符或此类操作符的序列将其流到宏中。一个例子:

ASSERT\_EQ(x.size()， y.size()) <<“向量x和y的长度不相等”;  
  
我< x.size ();++i) {EXPECT\_EQ(x[i]， y[i]) <<“向量x和y下标不同”<< i;}

任何可以流到ostream的东西都可以流到断言宏——特别是C字符串和字符串对象。如果一个宽字符串(wchar\_t\*， TCHAR\* in UNICODE mode on Windows，或std::wstring)被流到一个断言，那么在打印时它将被转换为UTF-8。

## 基本的断言

这些断言执行基本的真/假条件测试。

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_TRUE(条件); | EXPECT\_TRUE(条件); | *条件为真* |
| ASSERT\_FALSE(条件); | EXPECT\_FALSE(条件); | *条件是假的* |

请记住，当它们失败时，ASSERT\_\*将产生一个致命的失败并从当前函数返回，而EXPECT\_\*将产生一个非致命的失败，从而允许函数继续运行。在这两种情况下，断言失败意味着其包含的测试失败。

*可用性:Linux、Windows、Mac。*

## 二进制比较

本节描述比较两个值的断言。

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| 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，谷歌测试同时打印val1和val2。

值参数必须通过断言的比较操作符进行比较，否则将得到编译器错误。我们曾经需要参数来支持<<操作符来流到一个ostream中，但是从v1.6.0开始就不再需要了(如果<<被支持，当断言失败时将调用它来打印参数;否则谷歌测试将尝试以最佳方式打印它们。有关更多细节以及如何自定义参数的打印，请参阅此谷歌模拟配方。

这些断言可以使用用户定义的类型，但必须定义相应的比较操作符(例如==、<等)。如果定义了相应的操作符，最好使用ASSERT\_\*()宏，因为它们不仅会打印比较的结果，还会打印两个操作数。

参数总是精确计算一次。因此，争论产生副作用是可以的。但是，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)，代码不应该依赖于任何特定的参数求值顺序。

ASSERT\_EQ()在指针上执行指针相等。如果在两个C字符串上使用，它将测试它们是否位于相同的内存位置，而不是它们是否具有相同的值。因此，如果您想按值比较C字符串(例如const char\*)，请使用ASSERT\_STREQ()，稍后将对此进行描述。特别是，要断言C字符串为空，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。

本节中的宏既可以处理窄字符串对象，也可以处理宽字符串对象(string和wstring)。

*可用性:Linux、Windows、Mac。*

*历史记录:在2016年2月之前\*\_EQ有一个约定，把它叫做ASSERT\_EQ(期望的，实际的)，所以很多现有的代码都使用这个顺序。*现在\*\_EQ以相同的方式处理两个参数。

## 字符串比较

该组中的断言比较两个C字符串。如果您想比较两个字符串对象，可以使用EXPECT\_EQ、EXPECT\_NE等。

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_STREQ (str1 str2); | EXPECT\_STREQ (str1 str2); | 两个C字符串具有相同的内容 |
| ASSERT\_STRNE (str1 str2); | EXPECT\_STRNE (str1 str2); | 两个C字符串有不同的内容 |
| ASSERT\_STRCASEEQ (str1 str2); | EXPECT\_STRCASEEQ (str1 str2); | 两个C字符串有相同的内容，忽略大小写 |
| ASSERT\_STRCASENE (str1 str2); | EXPECT\_STRCASENE (str1 str2); | 两个C字符串有不同的内容，忽略大小写 |

请注意，断言名称中的“CASE”表示忽略该CASE。

\*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。

空指针和空字符串被认为是不同的。

*可用性:Linux、Windows、Mac。*

请参阅:有关更多字符串比较技巧(例如，子字符串、前缀、后缀和正则表达式匹配)，请参阅高级谷歌测试指南。

# 简单的测试

创建一个测试:

1. 使用TEST()宏来定义和命名一个测试函数，这些是不返回值的普通c++函数。
2. 在这个函数中，以及您希望包含的任何有效的c++语句中，使用各种谷歌测试断言来检查值。
3. 测试结果由断言决定;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,它成功了。

TEST(test\_case\_name, test\_name){…  
测试身体…}

TEST()参数从一般到特殊。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，并且它们不应该包含下划线(\_)。一个测试的全名由它所包含的测试用例和它的独立名称组成。来自不同测试用例的测试可以具有相同的单独名称。

例如，让我们以一个简单的整数函数为例:

整数的阶乘(int n);//返回n的阶乘

该函数的测试用例可能如下所示:

// test ! (0)  
TEST(FactorialTest, HandlesZeroInput) {EXPECT\_EQ(1, Factorial(0));}//测试正数的阶乘。  
  
  
  
  
TEST(FactorialTest, HandlesPositiveInput) {EXPECT\_EQ(1, Factorial(1));  
  
EXPECT\_EQ (2) (2));  
EXPECT\_EQ (6) (3));  
EXPECT\_EQ(40320的阶乘(8));}

谷歌测试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相同的测试用例中;换句话说，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，它们属于同一个测试用例FactorialTest。

*可用性:Linux、Windows、Mac。*

# 测试装置:对多个测试使用相同的数据配置

如果您发现自己正在编写两个或多个对类似数据进行操作的测试，那么您可以使用一个测试装置。它允许您为几个不同的测试重用相同的对象配置。

要创建一个fixture，只需:

1. 从::testing::Test派生一个类。用protected:或public:开始它的主体，因为我们想从子类访问fixture成员。
2. 在类内部，声明计划使用的任何对象。
3. 如果需要，可以编写一个默认构造函数或SetUp()函数来为每个测试准备对象。一个常见的错误是将SetUp()拼写为SetUp()，带有一个小u -不要让这种情况发生在你身上。
4. 如果需要，可以编写析构函数或TearDown()函数来释放在SetUp()中分配的资源。要了解什么时候应该使用构造函数/析构函数，什么时候应该使用SetUp()/TearDown()，请阅读这个FAQ条目。
5. 如果需要，定义要共享的测试的子例程。

使用fixture时，使用TEST\_F()而不是TEST()，因为它允许您访问测试fixture中的对象和子例程:

TEST\_F(test\_case\_name, test\_name){…  
测试身体…}

与TEST()类似，第一个参数是测试用例名称，但是对于TEST\_F()来说，它必须是测试装置类的名称。您可能已经猜到:\_F表示fixture。

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测试的宏。使用错误的宏会导致编译错误。

另外，在TEST\_F()中使用测试装置类之前，必须先定义它，否则会得到编译器错误“类声明之外的虚拟”。

对于使用TEST\_F()定义的每个测试，谷歌测试将:

1. 在运行时创建一个新的测试装置
2. 通过SetUp()立即初始化它，
3. 运行测试
4. 通过调用TearDown()进行清理
5. 删除测试夹具。注意，同一个测试用例中的不同测试有不同的测试装置对象，并且谷歌测试总是在创建下一个测试装置之前删除它。谷歌测试不为多个测试重用相同的测试装置。一个测试对夹具所做的任何更改都不会影响其他测试。

例如，我们写一个FIFO队列类queue的测试，它有如下接口:

模板<typename E> // E是元素类型。  
class Queue {public: Queue();  
  
  
void Enqueue(const e&element);  
E \*出列();如果队列为空，则返回NULL。  
size\_t大小()常量;  
…};

首先，定义一个fixture类。按照惯例，应该给它起一个FooTest的名字，其中Foo是被测试的类。

类QueueTest: public::testing::Test {protected: virtual void SetUp() {q1\_.Enqueue(1);  
  
  
  
q2\_.Enqueue (2);  
q2\_.Enqueue (3);  
{}队列<int> q0\_;  
  
  
  
  
队列< int > q1\_;  
队列< int > q2\_;};

在这种情况下，TearDown()是不需要的，因为我们不需要在每次测试之后进行清理，除了析构函数已经完成的工作。

现在，我们将使用TEST\_F()和这个fixture编写测试。

TEST\_F(QueueTest, isemptyfirst) {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 ());  
删除n;  
  
n = q2\_.Dequeue ();  
ASSERT\_TRUE (n ! = NULL);  
EXPECT\_EQ (2, \* n);  
EXPECT\_EQ (q2\_.size ());  
删除n;}

上面使用ASSERT\_\*和EXPECT\_\*断言。经验法则是，当您希望测试在断言失败后继续显示更多错误时使用EXPECT\_\*，而在断言失败后继续使用ASSERT\_\*没有意义。例如，Dequeue测试中的第二个断言是ASSERT\_TRUE(n != NULL)，因为我们稍后需要取消对指针n的引用，这将在n为NULL时导致segfault。

当这些测试运行时，会发生以下情况:

1. 谷歌测试构造了一个QueueTest对象(我们称之为t1)。
2. setup()初始化t1。
3. 第一个测试(isemptyinitial)在t1上运行。
4. t1.TearDown()在测试结束后进行清理。
5. t1已经被破坏。
6. 在另一个QueueTest对象上重复上述步骤，这次运行DequeueWorks测试。

*可用性:Linux、Windows、Mac。*

*注意:当一个测试对象被构造时，谷歌测试自动保存所有的谷歌测试标志，当它被析构时，恢复它们。*

# 调用测试

TEST()和TEST\_F()隐式地将它们的测试注册到谷歌测试中。因此，与许多其他c++测试框架不同，您不必为了运行它们而重新列出所有已定义的测试。

定义测试之后，可以使用RUN\_ALL\_TESTS()运行它们，如果所有测试都成功，则返回0，否则返回1。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。

调用RUN\_ALL\_TESTS()宏时:

1. 保存所有谷歌测试标志的状态。
2. 为第一个测试创建一个测试装置对象。
3. 通过SetUp()初始化它。
4. 在fixture对象上运行测试。
5. 通过TearDown()清理夹具。
6. 删除设备。
7. 恢复所有谷歌测试标志的状态。
8. 为下一个测试重复上述步骤，直到所有测试都运行完毕。

此外，如果文本fixture的构造函数在步骤2中产生了致命的错误，那么步骤3 - 5就没有意义了，因此会跳过它们。类似地，如果步骤3生成了一个致命的失败，那么步骤4将被跳过。

*重要提示:不能忽略RUN\_ALL\_TESTS()的返回值，否则gcc会给你一个编译错误。*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通过;因此main()函数必须返回RUN\_ALL\_TESTS()的值。

另外，您应该只调用RUN\_ALL\_TESTS()一次。多次调用它与某些高级谷歌测试特性(例如，线程安全死亡测试)冲突，因此不受支持。

*可用性:Linux、Windows、Mac。*

# 编写main()函数

你可以从这个样板文件开始:

# include“这/包/ foo。h”#包括“gt / gt。  
h"命名空间{//测试类Foo的装置。  
  
  
  
  
class FooTest: public::testing::Test {protected: //如果它的主体//是空的，你可以删除下面的任何或所有函数。  
  
  
  
  
FooTest(){//你可以在这里为每个测试做设置工作。  
  
} virtual ~FooTest(){//你可以做一些清理工作，不会抛出异常。  
  
  
  
} //如果构造函数和析构函数不足以设置//和清除每个测试，那么可以定义以下方法:  
  
  
  
  
  
  
  
这里的代码将在每次测试之后(析构函数之前)立即被调用。  
  
  
  
  
这里声明的对象可以被Foo.}测试用例中的所有测试使用;//测试Foo::Bar()方法是否执行Abc。  
  
  
  
  
  
TEST\_F(FooTest, MethodBarDoesAbc) {const string input\_filepath = "this/package/testdata/myinputfile.dat";  
  
const字符串output\_filepath = "this/package/testdata/myoutputfile.dat";  
Foo f;  
EXPECT\_EQ (0 f。Bar(input\_filepath, output\_filepath);}//测试Foo执行的Xyz操作。  
  
  
  
TEST\_F(FooTest, DoesXyz){//练习Foo的Xyz特性。  
  
{::testing::InitGoogleTest(&argc, argv);  
  
  
  
  
  
返回RUN\_ALL\_TESTS ();}

函数的作用是::testing::InitGoogleTest()解析谷歌测试标志的命令行，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中介绍这些标志。您必须在调用RUN\_ALL\_TESTS()之前调用这个函数，否则标志将无法正确初始化。

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

但是，也许您认为编写所有这些main()函数工作量太大了?我们完全同意您的观点，这就是为什么谷歌测试提供了main()的基本实现。如果它符合您的需要，那么只需将您的测试与gtest\_main库链接起来就可以了。

## Visual c++用户的重要注意事项

如果您将测试放到一个库中，而main()函数位于另一个库中或.exe文件中，那么这些测试将不会运行。原因是Visual c++中的一个bug。在定义测试时，谷歌测试将创建某些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++ linker看到库中的任何内容都没有从其他地方引用时，它就会抛出库。您必须使用主程序中的测试来引用库，以防止链接器丢弃它。以下是如何做到这一点。在你的库代码某处声明一个函数:

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*}

如果你把你的测试放在一个静态库中(不是DLL)，那么不需要使用\* \* \* \* (dllexport)。现在，在你的主程序中，编写一个代码调用该功能:

静态int dummy = PullInMyLibrary();

这将保持您的测试被引用，并使它们在启动时自动注册。

另外，如果您在静态库中定义了您的测试，那么将/OPT:NOREF添加到您的主程序链接器选项中。如果您使用msvc++ IDE，请转到您的.exe项目属性/配置属性/链接器/优化，并设置引用设置以保存未引用的数据(/OPT:NOREF)。这将使Visual c++ linker不会丢弃您的测试从最终可执行文件生成的单个符号。

不过，还有一个陷阱。如果您使用谷歌测试作为静态库(在gtest.vcproj中是这样定义的)，那么您的测试也必须驻留在静态库中。如果必须将它们放在DLL中，则必须更改谷歌测试以构建到DLL中。否则，您的测试将无法正确运行或根本无法运行。总的结论是:让您的生活更轻松——不要在库中编写测试!

# 接下来该怎么办

恭喜你!您已经学习了谷歌测试基础知识。您可以开始编写和运行谷歌测试测试，阅读一些示例，或者继续使用AdvancedGuide，它描述了许多更有用的谷歌测试特性。

# 已知的限制

谷歌测试被设计为线程安全的。在pthreads库可用的系统上，实现是线程安全的。当前，在其他系统(例如Windows)上同时使用来自两个线程的谷歌测试断言是不安全的。在大多数测试中，这不是一个问题，因为断言通常是在主线程中完成的。如果需要帮助，您可以自愿在gtest-port中实现必要的同步原语。h代表你的平台。