如果你在这里找不到问题的答案，并且你已经阅读了Primer和AdvancedGuide，那么把它发送到googletestframework@googlegroups.com。

## 为什么我应该使用谷歌测试而不是我最喜欢的c++测试框架?

首先，让我们明确表示，我们不想卷入哪一个c++测试框架是最好的争论。现在有很多优秀的框架可以用来编写c++测试，我们非常尊重这些框架的开发者和用户。我们不认为存在(或将会存在)单一的最佳框架——您必须为您正在处理的特定任务选择合适的工具。

我们创建谷歌测试是因为我们无法在现有框架中找到合适的特性和便利来满足我们的需求。下面是我们喜欢谷歌测试的原因。我们并不要求它们是谷歌测试所特有的—相反，它们的组合使谷歌测试成为我们的选择。我们希望这份清单能帮助你决定是否也适合你。

* 谷歌测试被设计成可移植的:它不需要异常或RTTI;它可以解决各种编译器和环境中的各种bug;等。因此，它适用于Linux、Mac OS X、Windows和一些嵌入式操作系统。
* 非致命断言(EXPECT\_\*)被证明是节省时间的好方法，因为它们允许测试在一个编辑-编译-测试周期中报告多个故障。
* 编写生成信息消息的断言很容易:只需使用流语法附加任何附加信息，例如ASSERT\_EQ(5, Foo(i)) << " where i = " << i;它不需要一组新的宏或特殊函数。
* 谷歌测试自动检测您的测试，不需要您枚举它们来运行它们。
* 死亡测试对于确保生产代码中的断言由适当的条件触发非常方便。
* SCOPED\_TRACE可以帮助您理解来自子例程或循环内部的断言失败的上下文。
* 您可以使用名称模式来决定运行哪些测试。当您想快速重现一个测试失败时，这节省了时间。
* 谷歌测试可以生成XML测试结果报告，这些报告可以由流行的连续构建系统(如Hudson)解析。
* 在谷歌测试简单的事情很容易,而艰难的事情是有可能的:除了先进的功能,如全球测试环境和测试参数化的值或类型,谷歌测试支持多种方式为用户扩展框架,如果谷歌测试不做的,很可能用户可以使用谷歌测试的公共API实现的特性,在不改变谷歌测试本身。特别是，你可以:
  + 通过定义自定义谓词来扩展您的测试词汇表，
  + 教谷歌测试如何打印你的类型，
  + 定义您自己的测试宏或实用程序，并使用谷歌测试的服务提供者接口验证它们
  + 通过拦截测试事件来反映测试用例或更改测试输出格式。

## 我在编译谷歌测试时得到警告。你能修好它们吗?

我们尽量减少编译器警告谷歌测试生成。在发布新版本之前，我们进行了测试，以确保在Windows、Linux和Mac OS上使用其CMake脚本编译时不会生成警告。

不幸的是，这并不意味着你可以保证在你的环境中编译谷歌测试时不会看到任何警告:

* 您可能使用与我们相同的另一个编译器，或者同一编译器的不同版本。我们不可能对所有编译器进行测试。
* 您可能与我们一样在不同的平台上进行编译。
* 您的项目可能会像我们一样使用不同的编译器标志。

不可能总是使谷歌测试警告对每个人都是免费的。或者，如果很少启用警告，并且修复违规会使代码更加复杂，则可能不需要这样做。

如果您在编译谷歌测试时看到警告，我们建议您使用-isystem标志(假设您正在使用GCC)来将谷歌测试标头标记为系统标头。这将抑制来自谷歌测试头的警告。

## 为什么测试用例名称和测试名称不应该包含下划线?

下划线(\_)是特殊的，因为c++保留了以下内容供编译器和标准库使用:

1. 以\_开头，后跟大写字母和的任何标识符
2. 在其名称中包含两个连续下划线(即\_\_)的任何标识符。

禁止用户代码使用此类标识符。

现在让我们看看这对TEST和TEST\_F意味着什么。

当前TEST(TestCaseName, TestName)生成一个名为TestCaseName\_TestName\_Test的类。如果TestCaseName或TestName包含\_会发生什么?

1. 如果TestCaseName以\_开头，然后是一个大写字母(比如\_Foo)，那么我们将得到\_Foo\_TestName\_Test，它是被保留的，因此是无效的。
2. 如果TestCaseName以\_(例如，Foo\_)结尾，那么我们就得到了Foo\_ testname\_test，这是无效的。
3. 如果TestName以\_(比如\_Bar)开头，那么我们将得到testcasename\_ bar\_test，这是无效的。
4. 如果TestName以\_(比如，Bar\_)结尾，那么我们将得到testcasename\_bar\_ test，它是无效的。

因此，TestCaseName和TestName显然不能以\_开头或结尾(实际上，TestCaseName可以以\_开头——只要\_后面没有大写字母。但这越来越复杂了。所以为了简单起见，我们只说它不能以\_.)开头。

TestCaseName和TestName中间包含\_似乎没有问题。然而,考虑一下:

TEST(Time, Flies\_Like\_An\_Arrow){…}TEST(Time\_Flies, Like\_An\_Arrow){…  
}

现在，这两个测试都将生成相同的类(Time\_Files\_Like\_An\_Arrow\_Test)。那不是很好。

简单起见，我们只要求用户在TestCaseName和TestName中避免\_。这条规则的约束比必要的多，但它很简单，也很容易记住。它还为谷歌测试提供了一些灵活性，以备将来需要更改其实现。

如果您违反了规则，可能不会有立即的结果，但是您的测试可能(只是可能)与新编译器(或您正在使用的编译器的新版本)或谷歌测试的新版本发生冲突。因此，最好遵守这个规则。

## 为什么不建议安装预编译的谷歌测试副本(例如，安装到/usr/local)?

在早期，我们说过可以使用make install在\*nix系统上安装已编译的谷歌测试库。这样，您机器的每个用户都可以编写测试，而无需重新编译谷歌测试。

这似乎是一个好主意，但它有一个got-cha:每个用户都需要使用编译安装的谷歌测试库所用的编译器标志来编译他的测试;否则，他可能会遇到未定义的行为(例如，测试可能表现奇怪，甚至可能在没有明显原因的情况下崩溃)。

为什么?因为c++有一个所谓的单一定义规则:如果两个c++源文件包含同一个类/函数/变量的不同定义，而您将它们链接在一起，就违反了这个规则。链接器可能捕获错误，也可能不捕获错误(在许多情况下，c++标准并不要求捕获错误)。如果没有，就会得到奇怪的运行时行为，这些行为是无法预料的，而且很难调试。

如果您使用不同的编译器标志编译谷歌测试和您的测试代码，他们可能会看到相同的类/函数/变量的不同定义(例如，由于在谷歌测试中使用了# If)。因此，为了您的理智，我们建议避免安装预编译的谷歌测试库。相反，每个项目应该编译谷歌测试本身，以确保谷歌测试和测试使用相同的标志。

## 如何在Windows上生成64位二进制文件(使用Visual Studio 2008)?

(特雷弗·罗宾逊回答)

加载提供的Visual Studio解决方案文件，可以是msvc\gtest-md。sln或msvc \ gtest.sln。通过迁移向导将解决方案和项目文件迁移到Visual Studio 2008。选择配置管理器…从“生成”菜单中。选择<新…活动解决方案平台下拉菜单中的>。从“新平台”下拉菜单中选择“x64”，将“设置”中的“复制设置”保留为“Win32”，选中“创建新项目平台”，然后单击“确定”。您现在有了Win32和x64平台配置，可以从标准工具栏中选择，这允许您在构建32位或64位二进制文件之间切换(或者同时使用批处理构建)。

为了防止生成的输出文件相互覆盖，您需要跨所有项目更改新创建的平台配置的中间目录设置。为此，在解决方案资源管理器中多选(例如，使用shift-click)所有项目(但不包括解决方案)。右键单击其中一个并选择Properties。在左侧窗格中，选择Configuration Properties，并从Configuration下拉菜单中选择All Configurations。确保选择的平台是x64。对于中间目录设置，将值从$(PlatformName)\$(ConfigurationName)更改为$(OutDir)\$(ProjectName)。单击OK，然后构建解决方案。构建完成后，64位二进制文件将位于msvc\x64\Debug目录中。

## 我可以在MinGW上使用谷歌测试吗?

我们自己还没有测试过，但是Per Abrahamsen报告说，当使用来自Cygwin的MinGW时，他能够成功地编译和安装谷歌测试。你需要配置它与:

CXX="g++ -mno-cygwin"

您应该能够将-mno-cygwin选项替换为直接链接到真正的MinGW二进制文件，但是我们还没有尝试这样做。

警告:

* 编译时有很多警告。
* make check会产生一些错误，因为并不是所有针对谷歌测试本身的测试都与MinGW兼容。

我们还报告了使用WxWidgets站点上的这些说明在Linux上成功地交叉编译谷歌测试MinGW二进制文件。

如果您对改进MinGW的支持感兴趣，请联系googletestframework@googlegroups.com。

## 为什么谷歌测试支持EXPECT\_EQ(NULL, ptr)和ASSERT\_EQ(NULL, ptr)，而不支持EXPECT\_NE(NULL, ptr)和ASSERT\_NE(NULL, ptr)?

由于c++的一些特性，它需要一些非平凡的模板元编程技巧来支持使用NULL作为EXPECT\_XX()和ASSERT\_XX()宏的参数。因此，我们只在最需要的地方进行测试(否则，我们将使谷歌测试的实现难于维护，并且比必要时更容易出错)。

EXPECT\_EQ()宏的第一个参数是期望值，第二个参数是实际值。有人希望编写EXPECT\_EQ(NULL, some\_expression)是合理的，这确实被请求了几次。因此我们实现了它。

对EXPECT\_NE(NULL, ptr)的需求没有那么强烈。当断言失败时，您已经知道ptr必须为NULL，因此在本例中，它不会为打印ptr添加任何信息。这意味着EXPECT\_TRUE(ptr != NULL)也可以工作。

如果我们要支持EXPECT\_NE(NULL, ptr)，为了一致性，我们将不得不支持EXPECT\_NE(ptr, NULL)以及，与EXPECT\_EQ不同，我们没有一个约定的顺序的两个参数的EXPECT\_NE。这意味着在实现中两次使用模板元编程技巧，使其更难理解和维护。我们认为利益与代价不相称。

最后，随着谷歌Mock的matcher库的增长，我们鼓励人们在测试中更多地使用统一的EXPECT\_THAT(值，matcher)语法。matcher方法的一个重要优点是可以很容易地组合matcher来形成新的matcher，而EXPECT\_NE等宏不能很容易地组合。因此，我们希望在匹配器上的投资比在EXPECT\_XX()宏上的投资更多。

## 谷歌测试是否支持并行运行测试?

测试运行程序往往与构建/测试环境紧密耦合，而谷歌测试并不试图解决并行运行测试的问题。相反，我们尝试让谷歌测试能够很好地与测试运行程序一起工作。例如，谷歌测试的XML报告包含在每个测试上花费的时间，它的gtest\_list\_tests和gtest\_filter标记可用于将测试方法的执行分割为多个进程。这些功能可以帮助测试运行程序并行地运行测试。

## 为什么不在不同的线程中运行谷歌测试来加快速度呢?

很难编写线程安全的代码。大多数测试都没有考虑到线程安全，因此在多线程设置中可能无法正常工作。

仔细想想，当您知道其他线程在做什么时，让您的代码工作已经很困难了。当您不知道其他线程在做什么时，让您的代码正常工作要困难得多，有时甚至是不可能的(请记住，可以在编写测试之后添加、删除或修改测试方法)。如果您想并行运行测试，最好在不同的进程中运行它们。

## 为什么不使用异常实现谷歌测试断言?

我们最初的动机是能够在禁用异常的项目中使用谷歌测试。后来我们意识到这种方法的一些额外好处:

1. 在c++中，抛出析构函数是未定义的行为。不使用异常意味着在析构函数中使用谷歌测试的断言是安全的。
2. 即使在失败之后，EXPECT\_\*宏家族也将继续，这允许在一次运行中报告测试中的多个失败。这是一个很受欢迎的特性，因为在c++中，编辑-编译-测试周期通常很长，能够同时修复多个对象是件好事。
3. 如果断言是使用异常实现的，测试可能会错误地忽略一个失败，如果它被用户代码捕获:

尝试{…ASSERT\_TRUE (…)…}catch(…){…  
}

即使ASSERT\_TRUE抛出，上面的代码也会传递。虽然不太可能有人在测试中编写这个模式，但是当您在被测试的代码调用的回调中编写断言时，可能会遇到这种模式。

不使用异常的缺点是ASSERT\_\*(使用return实现)将只中止当前函数，而不是当前测试。

## 为什么我们使用两个不同的宏来进行带有和不带有fixture的测试?

不幸的是，c++的宏系统不允许我们在两种情况下使用相同的宏。一种可能是只提供一个带有fixture的宏用于测试，有时需要用户定义一个空fixture:

TEST\_F(FooTest, DoesThis){…  
  
}

或

TEST\_F(FooTest, DoesThat){…  
  
}

然而，许多人认为这一行太多了。:-)我们的目标是让编写测试变得非常容易，所以我们试图让简单的测试变得不重要。这意味着对这样的测试使用单独的宏。

我们认为这两种方法都不理想，但都是合理的。最后，这两种情况都可能不太重要。

## 为什么我们不使用结构体作为测试装置?

我们喜欢仅在表示被动数据时使用结构。结构和类之间的这种区别有助于记录代码作者的意图。由于测试装置具有诸如SetUp()和TearDown()之类的逻辑，所以最好将它们定义为类。

## 为什么将死亡测试实现为断言，而不是使用测试运行器?

我们的目标是让死亡测试尽可能方便用户。特别是:

* 运行器样式需要将信息分成两部分:死亡测试本身的定义，以及运行器关于如何运行死亡测试和预期结果的规范。死亡测试将用c++编写，而运行器规范可能是也可能不是。用户需要小心地保持两者同步。ASSERT\_DEATH(statement, expected\_message)用一种语言在一个地方指定所有必要的信息，没有样板代码。它是非常声明性的。
* ASSERT\_DEATH具有与其他谷歌测试断言类似的语法和错误报告语义，因此很容易学习。
* ASSERT\_DEATH可以随意与其他断言和其他逻辑混合使用。每个测试方法不限于一个死亡测试。例如，你可以这样写:

if (FooCondition()) {ASSERT\_DEATH(Bar()， "blah");  
  
} else {ASSERT\_EQ(5, Bar());  
  
}

如果您希望每个测试方法都有一个死亡测试，那么您也可以使用这种风格来编写测试，但是我们不想把这种风格强加给用户。人为的限制越少越好。

* ASSERT\_DEATH可以引用当前函数中的局部变量，并且可以根据运行时信息决定需要多少个终止测试。例如,

const int count = GetCount();//只有在运行时才知道。  
for (int i = 1;我< =计数;i++) {ASSERT\_DEATH({double\* buffer = new double[i];  
  
  
…初始化缓冲……  
Foo(buffer, i)}，“blah blah”);  
  
}

基于runner的方法往往更静态，灵活性更差，或者需要更多用户努力来获得这种灵活性。

关于ASSERT\_DEATH的另一个有趣的事情是，它调用fork()来创建一个子进程来运行死亡测试。这非常快，因为fork()使用的是即写即拷的页面，开销几乎为零，而且子进程直接从用户提供的语句开始，跳过了所有全局和本地初始化，以及指向给定语句的任何代码。如果您从头开始启动子进程，那么如果测试动态地链接到许多库，那么加载所有内容并开始运行可能需要几秒钟的时间。

## 我的死亡测试修改了一些状态，但是更改似乎在死亡测试结束后丢失了。为什么?

死亡测试(EXPECT\_DEATH，等等)在子进程s.t中执行。预期的崩溃不会杀死测试程序(即父进程)。因此，它们在内存中产生的任何副作用都可以在它们各自的子进程中观察到，但在父进程中却看不到。你可以把它们想象成在平行宇宙中运行，或多或少。

## 编译器抱怨对一些静态const成员变量的“未定义引用”，但我确实在类体中定义了它们。怎么了?

如果你的类有一个静态数据成员:

/ / foo。类Foo{…  
  
  
kBar = 100;};

你还需要在foo.cc的类体外定义它:

const int Foo:千巴;这里没有初始化器。

否则，你的代码是无效的c++，并可能以意想不到的方式中断。特别是，在谷歌测试比较断言(EXPECT\_EQ等)中使用它将生成“未定义引用”链接器错误。

## 我有一个接口，它有几个实现。我可以一次编写一组测试并在所有实现中重复它们吗?

谷歌测试还没有对这类测试或一般的数据驱动测试提供良好的支持。我们希望能尽快在这方面取得进展。

## 我能从另一个派生出一个测试夹具吗?

是的。

每个测试装置都有一个相应的、命名相同的测试用例。这意味着只有一个测试用例可以使用特定的fixture。然而，有时多个测试用例可能希望使用相同的或稍微不同的fixture。例如，您可能希望确保GUI库的所有测试用例不会泄漏重要的系统资源，比如字体和画笔。

在谷歌测试中，您可以通过将共享的逻辑放在一个基本测试装置中来在测试用例之间共享一个装置，然后从这个基本测试装置中为每个想要使用这个公共逻辑的测试用例派生出一个单独的装置。然后使用TEST\_F()来使用每个派生的fixture编写测试。

通常，你的代码是这样的:

//定义一个基本的测试夹具。  
类BaseTest: public::testing::Test {protected:…};//从BaseTest派生一个fixture FooTest。  
  
  
  
  
  
类FooTest: public BaseTest {protected: virtual void SetUp() {BaseTest::SetUp();  
  
  
//先安装基础夹具。  
额外的设置工作…  
虚拟空间TearDown(){…  
  
FooTest的清洁工作…  
baset: TearDown ();//清理完FooTest后，记得把基础夹具拆下来!  
  
}……  
//使用fixture FooTest的测试。  
  
  
  
TEST\_F(FooTest, Bar){…TEST\_F(FooTest, Baz){…  
}……  
  
来自BaseTest的附加装置…

如果需要，您可以继续从派生的fixture派生测试fixture。谷歌测试对层次结构的深度没有限制。

有关使用派生测试装置的完整示例，请参见sample5。

## 我的编译器抱怨“void值没有被忽略，因为它应该是。”这是什么意思?

您可能在一个不返回void的函数中使用ASSERT\_\*()。ASSERT\_\*()只能在void函数中使用。

## 我的死亡测试挂起(或seg-fault)。我怎么修理它?

在谷歌测试中，死亡测试在子进程中运行，它们的工作方式非常微妙。要编写死亡测试，您确实需要了解它们是如何工作的。请确认你已经读过了。

特别是，死亡测试不喜欢在父进程中有多个线程。因此，您可以尝试的第一件事是消除在EXPECT\_DEATH()之外创建线程。

有时这是不可能的，因为一些必须使用的库可能在到达main()之前就已经在创建线程了。在这种情况下，您可以尝试通过在EXPECT\_DEATH()中移动尽可能多的活动(在极端情况下，您希望将所有内容都移动到其中)，或者在其中尽可能少地保留内容，从而最小化冲突的机会。另外，您可以尝试将死亡测试样式设置为“threadsafe”，这更安全，但速度更慢，然后看看是否有帮助。

如果您使用线程安全的终止测试，请记住它们在子进程中从头开始重新运行测试程序。因此，请确保您的程序可以与自身并行运行，并且是确定的。

最后，这归结为良好的并发编程。您必须确保程序中没有竞争条件或死锁。没有银弹——对不起!

## 我应该使用测试夹具的构造函数/析构函数还是设置/拆卸函数?

首先要记住的是谷歌测试不会跨多个测试重用相同的测试装置对象。对于每个TEST\_F，谷歌测试将创建一个新的测试装置对象，立即调用SetUp()，运行测试体，调用TearDown()，然后立即删除测试装置对象。

当您需要编写每个测试设置和拆卸逻辑时，您可以选择使用测试装置构造函数/析构函数或SetUp()/TearDown()。前者通常是首选，因为它有以下好处:

* 通过在构造函数中初始化成员变量，我们可以选择将其设置为const，这有助于防止意外更改其值，并使测试更加明显地正确。
* 如果我们需要子类化测试fixture类，子类的构造函数将首先调用基类的构造函数，子类的析构函数将在之后调用基类的析构函数。使用SetUp()/TearDown()，子类可能会忘记调用基类的SetUp()/TearDown()或者在错误的时间调用它们。

在以下罕见的情况下，您可能仍然需要使用SetUp()/TearDown():

* 如果拆析操作可以抛出异常，则必须使用TearDown()，而不是析析函数，因为抛出析析函数会导致未定义的行为，通常会立即终止程序。注意，当编译器中启用异常时，许多标准库(如STL)可能会抛出异常。因此，如果您想编写可移植的测试，不管有没有异常，您都应该选择TearDown()。
* 当指定gtest\_throw\_on\_failure标志时，断言宏抛出异常。因此，如果您计划使用此标志运行测试，则不应该在析构函数中使用谷歌测试断言。
* 在构造函数或析构函数中，不能对该对象进行虚函数调用。(您可以调用一个声明为virtual的方法，但它将被静态绑定。)因此，如果需要调用将在派生类中覆盖的方法，则必须使用SetUp()/TearDown()。

## 当我使用ASSERT\_PREDn时，编译器会提示“没有要调用的匹配函数”。我怎么修理它?

如果在ASSERT\_PRED\*或EXPECT\_PRED\*中使用的谓词函数是重载的或模板，编译器将很难确定应该使用哪个重载版本。ASSERT\_PRED\_FORMAT\*和EXPECT\_PRED\_FORMAT\*没有这个问题。

如果您看到这个错误，您可能希望切换到(ASSERT|EXPECT)\_PRED\_FORMAT\*，这也将为您提供更好的失败消息。但是，如果这不是一个选项，您可以通过显式地告诉编译器选择哪个版本来解决这个问题。

例如，假设你有

bool IsPositive(int n) {return n > 0;}bool IsPositive(double x) {return x > 0;}

如果你写，你会得到一个编译错误

EXPECT\_PRED1 (IsPositive 5);

然而，这将工作:

EXPECT\_PRED1 (\* static\_cast < bool (\*) (int) > \* (IsPositive), 5);

(static\_cast操作符的尖括号内的内容是IsPositive()的int版本的函数指针的类型。)

另一个例子，当你有一个模板函数

模板<typename T>bool IsNegative(T x) {return x < 0;}

你可以像这样在断言中使用它:

ASSERT\_PRED1 (IsNegative \* < int > \* 5);

如果模板有多个参数，事情就更有趣了。以下代码无法编译:

ASSERT\_PRED2(\*大于<int, int>\*， 5,0);

由于c++预处理器认为您正在给出ASSERT\_PRED2 4个参数，这比预期多了一个。解决方法是将谓词函数括在括号中:

ASSERT\_PRED2(\*(大于<int, int>)\*， 5, 0);

## 当我调用RUN\_ALL\_TESTS()时，我的编译器会抱怨“忽略返回值”。为什么?

有些人忽略了RUN\_ALL\_TESTS()的返回值。也就是说，代替

返回RUN\_ALL\_TESTS ();

他们写

RUN\_ALL\_TESTS ();

这是错误和危险的。测试运行程序需要查看RUN\_ALL\_TESTS()的返回值，以确定测试是否通过。如果main()函数忽略它，即使它有一个谷歌测试断言失败，您的测试也会被认为是成功的。非常糟糕。

为了帮助用户避免这个危险的bug, RUN\_ALL\_TESTS()的实现导致gcc在忽略返回值时发出这个警告。如果您看到这个警告，解决方法很简单:只要确保它的值用作main()的返回值即可。

## 我的编译器抱怨构造函数(或析构函数)不能返回值。这是怎么呢

由于c++的特性，为了支持将消息流到ASSERT\_\*的语法，例如。

ASSERT\_EQ(1, Foo()) << "blah blah" << Foo;

我们不得不放弃在构造函数和析构函数中使用ASSERT\*和FAIL\*(但不要期望\*和ADD\_FAILURE\*)。解决方法是将构造函数/析构函数的内容移动到一个私有的void成员函数中，或者切换到EXPECT\_\*()(如果可以的话)。用户指南的这一部分对此进行了解释。

## 我的设置函数没有被调用。为什么?

c++是区分大小写的。它应该拼写为SetUp()。您将它拼写为Setup()吗?

类似地，有时人们将SetUpTestCase()拼写为SetUpTestCase()，并想知道为什么它从来没有被调用。

## 如何直接跳转到Emacs中的故障行?

Emacs和许多其他ide(如acme和XCode)可以理解谷歌测试的失败消息格式。如果谷歌测试消息在Emacs的编译缓冲区中，那么它是可单击的。现在可以在消息上按回车键跳转到相应的源代码，或者使用“C-x”跳转到下一个失败。

## 我有几个测试用例，它们共享相同的测试装置逻辑，我必须为它们每个定义一个新的测试装置类吗?这看起来很乏味。

你不必这么做。而不是

TEST\_F(FooTest, Abc){…  
  
TEST\_F(FooTest, Def){…  
TEST\_F(BarTest, Abc){…  
  
  
  
}TEST\_F(BarTest, Def){…  
}

你可以简单地定义测试装置:

TEST\_F(FooTest, Abc){…  
  
TEST\_F(FooTest, Def){…  
TEST\_F(BarTest, Abc){…  
  
  
  
}TEST\_F(BarTest, Def){…  
}

## 谷歌测试输出隐藏在一大堆日志消息中。我该怎么办?

谷歌测试输出应该是一份简洁且对人类友好的报告。如果您的测试本身生成文本输出，那么它将与谷歌测试输出混合在一起，使其难以阅读。然而，有一个简单的方法可以解决这个问题。

由于大多数日志消息都被发送到stderr，所以我们决定将谷歌测试输出发送到stdout。通过这种方式，您可以使用重定向轻松地将两者分开。例如:

/ my\_test > googletest\_output.txt

## 为什么我应该选择测试装置而不是全局变量?

有几个很好的理由:

1. 您的测试可能需要更改其全局变量的状态。这使得避免一个测试的副作用和污染其他测试变得困难，使得调试变得困难。通过使用fixture，每个测试都有一组不同的变量(但是名称相同)。因此，测试是相互独立的。
2. 全局变量影响全局名称空间。
3. 测试fixture可以通过子类化来重用，而使用全局变量很难做到这一点。如果许多测试用例有一些共同之处，这是很有用的。

## 如何在不编写FRIEND\_TEST()s的情况下测试私有类成员?

您应该尝试编写可测试的代码，这意味着类应该很容易从它们的公共接口进行测试。实现这一点的一种方法是Pimpl习语:将类的所有私有成员移动到helper类中，并将helper类的所有成员公开。

你有几个不需要使用FRIEND\_TEST的选项:

* 将测试作为fixture类的成员来编写:

类Foo{朋友类FooTest;  
  
类FooTest: public::testing::Test {protected:…  
  
  
  
  
  
空白Test1 () {…} //访问Foo类的私有成员。  
空白Test2 () {…TEST\_F(FooTest, Test1) {Test1();}TEST\_F(FooTest, Test2) {Test2();}

* 在fixture类中，为被测试类的私有成员编写访问器，然后在测试中使用访问器:

类Foo{朋友类FooTest;  
  
类FooTest: public::testing::Test {protected:…  
  
  
  
  
  
get\_private\_member1(Foo\* obj){返回obj->private\_member1\_;  
  
TEST\_F(FooTest, Test1){…  
  
  
  
  
get\_private\_member1 (x)…}

* 如果这些方法被声明为受保护的，你可以在一个只测试的子类中改变它们的访问级别:

类你的类{…  
  
受保护://受保护的可测试性访问。  
int DoSomethingReturningInt ();  
};//在your\_class\_test中。  
  
  
类TestableYourClass: public YourClass{…  
  
  
使用YourClass公众:::DoSomethingReturningInt;TEST\_F(YourClassTest, DoSomethingTest) {TestableYourClass obj;  
  
  
  
  
  
assertequal (expected\_value obj.DoSomethingReturningInt ());}

## 如何在不编写FRIEND\_TEST()s的情况下测试私有类静态成员?

我们发现私有静态方法使头文件杂乱。它们是实现细节，理想情况下应该远离.h。所以我经常把它们变成自由函数。

而不是:

/ / foo。类Foo{…  
  
  
私有:static bool Func(int n);};// foo。  
  
  
  
Func(int n){…  
}/ / foo\_test.ccEXPECT\_TRUE (Foo:: Func (12345);

你最好这样写:

/ / foo。类Foo{…};// Foo。  
  
  
  
  
bool Func(int n){…  
  
}}/ / foo\_test。  
  
  
{bool Func(int n);}EXPECT\_TRUE(internal::Func(12345));

## 我希望使用不同的参数多次运行测试。我需要写几份类似的吗?

不。您可以使用一个称为值参数化测试的特性，该特性允许您使用不同的参数重复您的测试，而不需要多次定义它。

## 如何测试定义main()的文件?

测试一个foo。cc文件，您需要编译并将其链接到您的单元测试程序中。但是，当文件包含main()函数的定义时，它将与单元测试的main()冲突，并将导致构建错误。

正确的解决办法是把它分成三个文件:

1. foo。h包含声明，
2. foo。cc包含除main()之外的所有定义
3. foo\_main。它只包含main()的定义。

然后foo。cc可以很容易地测试。

如果您正在将测试添加到现有文件中，并且不希望出现这样的干扰性更改，那么有一个技巧:只包含整个foo。单元测试中的cc文件。例如:

/ /文件foo\_unittest。抄送//页眉部分…  
  
  
  
  
//在foo中重命名main()。为单元测试腾出空间  
  
  
"//测试从这里开始…

然而，请记住这是一个黑客，应该只作为最后的手段。

## ASSERT\_DEATH()中的语句参数是什么?

ASSERT\_DEATH(\_statement\_， \_regex\_)(或任何死亡断言宏)可以在\_statement\_有效的地方使用。所以基本上\_statement\_可以是任何在当前上下文中有意义的c++语句。特别是，它可以引用全局和/或局部变量，可以是:

* 一个简单的函数调用(通常是这样)，
* 一个复杂的表达式，或者
* 一个复合语句。

以下是一些例子:

//死亡测试可以是一个简单的函数调用。  
{ASSERT\_DEATH(Xyz(5)， "Xyz failed");}//或引用变量和函数的复杂表达式。  
  
  
  
  
{const bool c = Condition();  
  
ASSERT\_DEATH (c ?方法("test")， "(Func1|方法)失败";}//死亡断言可以在函数的任何地方使用。  
  
  
  
特别是，它们可以在一个循环中。  
  
{//验证Foo(0)， Foo(1)，…  
，则Foo(4)全部死亡。  
for (int i = 0;我< 5;其中，"Foo有\\d+错误"，::testing::Message() << "where i is " << i);  
  
  
一个死亡断言可以包含一个复合语句。  
  
  
  
TEST(MyDeathTest, CompoundStatement){//验证Bar(0)、Bar(1)、…  
，和// Bar(4)死亡。  
  
ASSERT\_DEATH({for (int i = 0;  
我< 5;我+ +){酒吧(我);  
  
}}，“Bar有\\d+错误”);

googletest\_unittest。如果您感兴趣，cc包含更多的示例。

## ASSERT\_DEATH中的正则表达式使用什么语法?

在POSIX系统上，谷歌测试使用POSIX扩展正则表达式语法(http://en.wikipedia.org/wiki/Regularexpression#POSIXExtendedRegularExpressions)。在Windows上，它使用正则表达式语法的有限变体。有关详细信息，请参见正则表达式语法。

## 我有一个装置类Foo，但TEST\_F(Foo, Bar)给我错误"没有匹配的函数调用Foo::Foo()"。为什么?

谷歌测试需要能够创建测试装置类的对象，所以它必须有一个默认构造函数。通常编译器会为你定义一个。然而，在有些情况下，你必须定义自己的:

* 如果您显式地声明类Foo的非默认构造函数，那么您需要定义一个默认构造函数，即使它是空的。
* 如果Foo有一个const非静态数据成员，那么必须定义默认构造函数，并在构造函数的初始化器列表中初始化const成员。(gcc的早期版本并不强制您初始化const成员。这是一个在gcc 4中已经修复的bug。)

## 为什么ASSERT\_DEATH会抱怨以前已经连接的线程?

在Linux pthread库中，一旦从单个线程跨越到多个线程，就没有回头路可走。第一次创建线程时，还会创建一个管理器线程，所以会得到3个线程，而不是2个。稍后，当您创建的线程加入主线程时，线程数减少1，但是管理器线程永远不会被杀死，所以您仍然有2个线程，这意味着您不能安全地运行死亡测试。

新的NPTL线程库不会遇到这个问题，因为它不创建管理器线程。但是，如果您不能控制测试运行在哪台机器上，就不应该依赖于它。

## 为什么谷歌测试在使用ASSERT\_DEATH时需要将整个测试用例而不是单个测试命名为FOODeathTest ?

谷歌测试不交错来自不同测试用例的测试。也就是说，它首先在一个测试用例中运行所有测试，然后在下一个测试用例中运行所有测试，以此类推。谷歌测试之所以这样做，是因为它需要在运行第一个测试之前建立一个测试用例，并在事后删除它。分割测试用例将需要多个设置和拆卸过程，这是低效的，并且使语义不清晰。

如果我们要根据测试名而不是测试用例名来决定测试的顺序，那么我们将会遇到以下情况:

TEST\_F(FooTest, AbcDeathTest){…TEST\_F(FooTest, Uvw){…  
}TEST\_F(BarTest, DefDeathTest){…  
  
}TEST\_F(BarTest, Xyz){…  
}

因为脚。死亡测试需要在测试前进行。Xyz，我们不交错来自不同测试用例的测试，我们需要在FooTest用例中运行所有测试，然后在BarTest用例中运行任何测试。这与运行BarTest的要求相矛盾。DefDeathTest FooTest.Uvw之前。

## 但是当我的整个测试用例包含死亡测试和非死亡测试时，我不喜欢调用FOODeathTest。我该怎么办?

您不必这样做，但是如果您愿意，您可以将测试用例划分为FooTest和FooDeathTest，其中的名称清楚地表明它们是相关的:

类FooTest: public::testing::Test{…TEST\_F(FooTest, Abc){…  
  
TEST\_F(FooTest, Def){…  
TEST\_F(FooDeathTest, Uvw){…  
  
  
  
EXPECT\_DEATH (…)…TEST\_F(FooDeathTest, Xyz){…  
ASSERT\_DEATH (…)…}

## 当我使用断言时，编译器会抱怨“没有匹配‘运算符<<’”。到底发生了什么事?

如果在断言中使用用户定义的类型FooType，则必须确保定义了std:: ostream&operator <<(std::ostream&， const footype&)函数，以便可以打印FooType的值。

此外，如果在名称空间中声明了FooType，那么<<操作符也需要在相同的名称空间中定义。

## 如何抑制Windows上的内存泄漏消息?

由于静态初始化的谷歌测试单例需要在堆上分配内存，Visual c++内存泄漏检测器将在程序运行结束时报告内存泄漏。避免这种情况的最简单方法是使用\_CrtMemCheckpoint和\_CrtMemDumpAllObjectsSince调用来不报告任何静态初始化的堆对象。有关更多细节和附加的堆检查/调试例程，请参见MSDN。

## 我在Visual Studio中使用谷歌测试构建我的项目，我得到的只是一堆链接器错误(或警告)。的帮助!

如果您尝试将您的测试项目与谷歌测试库链接，而您的项目和测试库不是使用相同的编译器设置构建的，那么您可能会得到以下链接器错误或警告。

* 在对象中已经定义的符号
* LNK4217:本地定义的符号'symbol'导入函数'function'
* LNK4049:本地定义的符号“符号”导入

谷歌测试项目(gtest.vcproj)将运行时库选项设置为/MT(使用多线程静态库，/MTd用于调试)。如果您的项目使用其他东西，例如/MD(使用多线程dll， /MDd用于调试)，您需要更改谷歌测试项目中的设置以匹配您的项目。

要更新此设置，请在Visual Studio IDE中打开项目属性，然后选择分支配置属性| C/ c++ |代码生成，并更改“运行时库”选项。您也可以尝试使用gtest-md。用vcproj代替gtest。

## 我把测试放在一个库中，谷歌测试不运行它们。发生什么事情了?

你读过谷歌测试入门页面上的警告吗?

## 我想用谷歌测试与Visual Studio，但不知道从哪里开始。

许多人都在你的位置，其中一个张贴了他的解决方案到我们的邮件列表。

## 当我试图在Solaris上使用谷歌测试时，我看到了提到std::type\_traits的编译错误。

谷歌测试使用了SunStudio不支持的部分标准c++库。我们的用户报告了使用替代实现的成功。尝试运行后，运行这个commad:

导出CC= CC CXX=CC CXXFLAGS='-library=stlport4'

## 我的代码如何检测它是否在测试中运行?

如果您编写的代码嗅探出它是否在测试中运行，并相应地执行不同的操作，那么您就将只测试的逻辑泄漏到了生产代码中，并且没有简单的方法可以确保只测试的代码路径不会在生产中错误地运行。这样的聪明也导致了海森的错误。因此，我们强烈建议不要这样做，并且谷歌测试并没有提供这样做的方法。

通常，使代码在测试中表现不同的推荐方法是依赖项注入。您可以从测试和生产代码中注入不同的功能。因为您的产品代码根本没有链接到for-test逻辑中，所以意外地运行它是没有危险的。

但是，如果您真的、真的、真的别无选择，并且如果您遵循以\_test结束测试程序名称的规则，那么您可以使用可怕的方法来嗅探您的可执行程序名称(main()中的argv[0])，以了解代码是否处于测试状态。

## 谷歌测试定义了一个与另一个库定义的宏冲突的宏。我该怎么处理呢?

在c++中，宏不服从名称空间。因此，如果您#include了两个定义相同名称的宏，则两个定义相同名称的库将发生冲突。如果谷歌测试宏与另一个库发生冲突，您可以强制谷歌测试重命名其宏以避免冲突。

具体来说，如果谷歌测试和其他一些代码都定义了宏FOO，您可以添加

-DGTEST\_DONT\_DEFINE\_FOO = 1

到编译器标志，告诉谷歌测试更改宏的名称从FOO到GTEST\_FOO。例如，使用-DGTEST\_DONT\_DEFINE\_TEST=1，您需要编写

GTEST\_TEST(SomeTest, DoesThis){…}

而不是

这是什么意思?}

为了定义一个测试。

目前，下面的测试、失败、成功和基本比较断言宏可以有替代名称。你可以在这里看到完整的宏列表。更多信息可以在README文件的“避免宏名称冲突”部分找到。

## 如果我在不同的名称空间中定义了两个独立的测试方法(Foo, Bar)，可以吗?

是的。

规则是相同测试用例中的所有测试方法必须使用相同的fixture类。这意味着以下内容是允许的，因为两个测试使用相同的fixture类(::testing::Test)。

{TEST(CoolTest, DoSomething) {SUCCEED();}} //命名空间foonamespace bar {TEST(CoolTest, DoSomething) {SUCCEED();}} //命名空间foo

但是，下面的代码是不允许的，并且会从谷歌测试中产生一个运行时错误，因为测试方法使用了具有相同测试用例名称的不同测试装置类。

名称空间foo {class CoolTest: public::testing::Test {};  
// Fixture foo::CoolTestTEST\_F(CoolTest, DoSomething) {SUCCEED();} //命名空间foonamespace bar {class CoolTest: public::testing::Test {};  
  
  
  
  
  
  
// Fixture: bar::CoolTestTEST\_F(CoolTest, DoSomething) {SUCCEED();}} //命名空间foo

## 如何使用Xcode 4构建谷歌测试框架?

如果您试图使用Xcode 4.0或更高版本构建谷歌测试的Xcode项目，您可能会遇到类似于“目标gtest\_framework: /Developer/SDKs/ macosx10.4 . SDK中缺少SDK”的错误消息。这意味着Xcode不支持项目所针对的SDK。有关如何解决此问题，请参阅README文件中的Xcode部分。

## 我的问题不在你的FAQ里!

如果你不能在这个常见问题解答中找到你的问题的答案，你可以使用一些其他的资源:

1. 阅读其他wiki页面，
2. 搜索邮件列表存档，
3. 在googletestframework@googlegroups.com上提问，就会有人回答(为了防止垃圾邮件，我们要求你在发帖前加入讨论组)。

请注意，在问题跟踪器中创建一个问题并不是获得答案的好方法，因为它很少被少数人监控。

在提问时，提供尽可能多的信息是有帮助的(如果你的问题中没有足够的信息，人们无法帮助你):

* 您使用的谷歌测试的版本(如果直接从Git签出，则提交哈希)(谷歌测试正在积极开发中，因此您的问题可能在后续版本中得到了解决)，
* 你的操作系统,
* 编译器的名称和版本，
* 你给编译器的完整命令行标志，
* 完整的编译器错误消息(如果问题是关于编译的)，
* 遇到问题的实际代码(理想情况下是最小但完整的程序)。