既然您已经阅读了初级阅读并学习了如何使用谷歌测试编写测试，那么现在就该学习一些新技巧了。本文档将向您展示更多的断言，以及如何构造复杂的失败消息、传播致命的失败、重用和加速测试装置，以及如何在测试中使用各种标志。

# 更多的断言

本节将介绍一些不太常用但仍然很重要的断言。

## 明确的成功和失败

这三个断言实际上并不测试值或表达式。相反，它们直接产生成功或失败。与实际执行测试的宏一样，您可以将自定义失败消息传入宏。

|成功();| |:- - - - - - - - - - - - - |

生成一个成功。这并不能使整个测试成功。只有在执行过程中没有断言失败时，测试才被认为是成功的。

注意:SUCCEED()纯粹是纪录片，目前不生成任何用户可见的输出。但是，将来我们可能会将SUCCEED()消息添加到谷歌测试的输出中。

|失败();| ADD\_FAILURE ();| ADD\_FAILURE\_AT(“文件\路径”,line\_number\_);| |:- - - - - - - - - - - - |:- - - - - - - - - - - - - - - - - - |:- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - |

FAIL()生成一个致命错误，而ADD\_FAILURE()和ADD\_FAILURE\_AT()生成一个非致命错误。当控制流而不是布尔表达式决定测试的成功或失败时，这些是有用的。例如，你可以这样写:

switch(表达式){case 1:…  
一些检查…  
案例2:……其他一些支票…  
  
默认:FAIL() << "We shouldn't get here.";}

注意:只能在返回void的函数中使用FAIL()。有关更多信息，请参见断言放置部分。

*可用性:Linux、Windows、Mac。*

## 异常的断言

这些用于验证一段代码抛出(或不抛出)给定类型的异常:

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_THROW(声明、异常\ type\_); | EXPECT\_THROW(声明、异常\ type\_); | *语句抛出给定类型的异常* |
| ASSERT\_ANY\_THROW(声明); | EXPECT\_ANY\_THROW(声明); | *语句抛出任何类型的异常* |
| ASSERT\_NO\_THROW(声明); | EXPECT\_NO\_THROW(声明); | *语句不会抛出任何异常* |

例子:

ASSERT\_THROW(Foo(5)， bar\_exception);  
  
  
  
栏(n);});

*可用性:Linux、Windows、Mac;*因为版本1.1.0。

## 更好的错误消息的断言

即使谷歌测试有一组丰富的断言，它们也不可能是完整的，因为预测用户可能遇到的所有场景是不可能的(也不是一个好主意)。因此，有时用户不得不使用EXPECT\_TRUE()来检查复杂的表达式，因为没有更好的宏。它的问题是没有显示表达式各部分的值，因此很难理解哪里出错了。作为一种解决方案，一些用户选择自己构造失败消息，将其流化为EXPECT\_TRUE()。然而，这是很尴尬的，尤其是当表达式有副作用或者计算代价很高的时候。

谷歌测试给你三个不同的选择来解决这个问题:

### 使用现有的布尔函数

如果你已经有一个函数或者一个返回bool(或者一个可以隐式转换为bool的类型)的函式，你可以在断言中使用它，让函数参数免费打印:

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_PRED1 (pred1 val1); | EXPECT\_PRED1 (pred1 val1); | *pred1 (val1)返回true* |
| ASSERT\_PRED2 pred2 (val1, val2); | EXPECT\_PRED2 pred2 (val1, val2); | *pred2(val1, val2)返回true* |
| … | … | … |

在上面，predn是一个n元谓词函数或函子，其中val1, val2，…， valn是它的参数。如果将谓词应用于给定的参数时返回true，则断言成功，否则将失败。当断言失败时，它打印每个参数的值。在这两种情况下，参数都只计算一次。

这是一个例子。鉴于

如果m和n除1外没有公约数，则返回true。  
bool MutuallyPrime(int m, int n){…}const int a = 3;const int b = 4;const int c = 10;

断言EXPECT\_PRED2(互质数，a, b);将成功，而断言EXPECT\_PRED2(MutuallyPrime, b, c);消息会失败吗

**注:**

1. 如果您在使用ASSERT\_PRED\*或EXPECT\_PRED\*时看到编译器错误“没有要调用的匹配函数”，请参阅此FAQ了解如何解决此问题。
2. 目前我们只提供了圆度<= 5的断言。如果您需要高特性断言，请告诉我们。

*可用性:Linux、Windows、Mac*

### 使用返回AssertionResult的函数

虽然EXPECT\_PRED\*()和friends对于快速工作非常方便，但是其语法并不令人满意:必须为不同的宏使用不同的宏，而且它感觉更像是Lisp而不是c++。AssertionResult类解决了这个问题。

AssertionResult对象表示断言的结果(无论是成功还是失败，以及相关的消息)。你可以创建一个AssertionResult使用这些工厂功能之一:

名称空间测试{//返回AssertionResult对象，以指示断言已成功。  
  
  
  
AssertionResult AssertionSuccess();//返回一个AssertionResult对象，指示断言已经//失败。  
  
  
  
AssertionResult AssertionFailure ();}

然后可以使用<<操作符将消息流传递到AssertionResult对象。

要在布尔断言(如EXPECT\_TRUE())中提供更具可读性的消息，请编写一个返回AssertionResult而不是bool的谓词函数。例如，如果您将IsEven()定义为:

::testing::AssertionResult IsEven(int n) {if ((n % 2) == 0) return::testing::AssertionSuccess();  
  
  
else return::testing::AssertionFailure() << n << " is odd";}

而不是:

bool IsEven(int n) {return (n % 2) == 0;}

失败的断言EXPECT\_TRUE(IsEven(Fib(4))将打印:

而不是更不透明

如果你想在EXPECT\_FALSE和ASSERT\_FALSE的信息，并在成功的情况下使谓词变慢，你可以提供一个成功消息:

::AssertionResult IsEven(int n) {if ((n % 2) == 0) return::testing::AssertionSuccess() << n << " is偶";  
  
  
else return::testing::AssertionFailure() << n << " is odd";}

然后语句EXPECT\_FALSE(IsEven(Fib(6))将被打印出来

*可用性:Linux、Windows、Mac;*1.4.1自版本。

### 使用Predicate-Formatter

如果你发现生成的默认消息(断言|预计)\_PRED \*和(断言|预计)\_(真|假)不满意,或一些参数上ostream谓词不支持流媒体,您可以使用以下predicate-formatter断言完全自定义消息格式:

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_PRED\_FORMAT1 (pred \ format1 val1\_); | EXPECT\_PRED\_FORMAT1 (pred \ format1 val1\_); | *pred \ format1 (val1) \_是成功的* |
| ASSERT\_PRED\_FORMAT2 (val1, pred \ format2 val2\_); | EXPECT\_PRED\_FORMAT2 (val1, pred \ format2 val2\_); | *pred\format2(val1, val2)\_成功* |
| … | … | … |

这组宏与前两组宏的区别在于，\_PRED\_FORMAT\*不是一个谓词(ASSERT|EXPECT)，而是一个谓词-formatter (pred\formatn\_)，它是一个带有签名的函数或函子:

::testing::AssertionResult PredicateFormattern(const char\*expr1, const char\*expr2，…T1val1, T2val2，…Tnvaln);

其中val1, val2，…， valn是谓词参数的值，expr1, expr2，…， exprn是源代码中出现的对应表达式。类型T1, T2，…， Tn可以是值类型，也可以是引用类型。例如，如果一个参数的类型是Foo，那么您可以将它声明为Foo或const Foo&，任何一个都可以。

谓词-格式程序返回一个::testing::AssertionResult对象，以指示断言是否成功。创建这样一个对象的唯一方法是调用这些工厂函数之一:

作为一个例子，让我们改进前一个例子中的失败消息，它使用了EXPECT\_PRED2():

//返回m和n的最小素数公约数，//或当m和n互为素数时为1。  
  
}//断言两个整数互为质数的谓词-格式化程序。  
  
  
::testing::AssertionResult AssertMutuallyPrime(const char\* m\_expr, const char\* n\_expr, int m, int n) {if (MutuallyPrime(m, n))返回::testing::AssertionSuccess();

使用这个谓词-formatter，我们可以使用

EXPECT\_PRED\_FORMAT2 (AssertMutuallyPrime, b, c);

生成消息

您可能已经意识到，我们前面介绍的许多断言都是\_PRED\_FORMAT\*的特殊情况(除了|断言)。事实上，它们中的大多数确实是使用\_PRED\_FORMAT\*定义的(除了|断言之外)。

*可用性:Linux、Windows、Mac。*

## 浮点数比较

比较浮点数比较复杂。由于舍入错误，两个浮点数不太可能完全匹配。因此，ASSERT\_EQ的天真的比较通常不工作。由于浮点数可以有很宽的取值范围，所以没有固定的误差范围。比较时最好使用一个固定的相对误差边界，除了由于精度损失而接近于0的值之外。

通常，为了使浮点比较有意义，用户需要仔细选择错误范围。如果他们不想或者不关心，比较最后一个位置的单位(ULPs)是一种很好的默认方法，谷歌测试提供了断言来实现这一点。关于ulp的详细信息很长;如果您想了解更多，请参阅这篇有关浮动比较的文章。

### 浮点宏

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_FLOAT\_EQ (val1, val2); | EXPECT\_FLOAT\_EQ (val1, val2); | 这两个浮点值几乎相等 |
| ASSERT\_DOUBLE\_EQ (val1, val2); | EXPECT\_DOUBLE\_EQ (val1, val2); | 两个双精度值几乎相等 |

我们所说的“几乎相等”是指两个值之间的距离在4个ULP之内。

以下断言允许您选择可接受的错误范围:

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| abs \ error\_ ASSERT\_NEAR (val1, val2); | abs \ error\_ EXPECT\_NEAR (val1, val2); | val1和val2之间的差值不超过给定的绝对误差 |

*可用性:Linux、Windows、Mac。*

### 浮点Predicate-Format功能

有些浮点运算是有用的，但不常用。为了避免出现大量新的宏，我们将它们作为谓词断言宏(如EXPECT\_PRED\_FORMAT2等)中使用的谓词格式函数提供。

EXPECT\_PRED\_FORMAT2(::testing::FloatLE, val1, val2);

验证val1小于或几乎等于val2。可以用ASSERT\_PRED\_FORMAT2替换上面表中的EXPECT\_PRED\_FORMAT2。

*可用性:Linux、Windows、Mac。*

## Windows HRESULT断言

这些断言测试HRESULT是否成功。

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_HRESULT\_SUCCEEDED(表达式); | EXPECT\_HRESULT\_SUCCEEDED(表达式); | *表达是一个成功的结果* |
| ASSERT\_HRESULT\_FAILED(表达式); | EXPECT\_HRESULT\_FAILED(表达式); | *表达式是一个失败的结果* |

生成的输出包含与表达式返回的HRESULT代码相关联的人类可读的错误消息。

你可以这样使用它们:

CComVariant empty; assert\_hresult\_succeed (shell->ShellExecute(CComBSTR(url)， empty, empty, empty));

*可用性:窗户。*

## 类型的断言

你可以调用这个函数

::测试::StaticAssertTypeEq < T1, T2 > ();

以断言类型T1和T2是相同的。如果满足断言，函数将不执行任何操作。如果类型不同，函数调用将无法编译，编译器错误消息将可能(取决于编译器)显示T1和T2的实际值。这在模板代码中很有用。

*注意:当在类模板或函数模板的成员函数中使用时，StaticAssertTypeEq<T1, T2>()仅在函数实例化时有效。*例如,给定:

模板<typename T>类Foo {public: void Bar() {::testing::StaticAssertTypeEq<int, T>();  
  
}};

代码:

void Test1() {Foo<bool> Foo;}

不会生成编译器错误，因为Foo<bool>::Bar()从未实际实例化。相反,您需要:

void Test2() {Foo<bool> Foo;foo.Bar ();}

导致编译器错误。

*可用性:Linux、Windows、Mac;*因为1.3.0版本版本。

## 断言位置

您可以在任何c++函数中使用断言。特别是，它不必是测试装置类的方法。一个约束条件是，生成致命错误(FAIL\*和ASSERT\_\*)的断言只能用于空洞返回函数。这是谷歌测试没有使用异常的结果。通过将它放在一个非void函数中，您将得到一个令人困惑的编译错误，比如“error: void value not ignore as it ought to be”。

如果需要在返回非void的函数中使用断言，一种方法是让函数返回out参数中的值。例如，你可以重写T2 Foo(T1 x)来取消Foo(T1 x, T2\* result)。您需要确保\*result包含一些合理的值，即使函数提前返回也是如此。由于该函数现在返回void，您可以在其中使用任何断言。

如果不能更改函数的类型，则应该使用生成非致命错误的断言，如ADD\_FAILURE\*和EXPECT\_\*。

*注意:根据c++语言规范，构造函数和析构函数不被认为是无效返回函数，因此您不能在它们中使用致命的断言。*如果尝试，将得到编译错误。一个简单的解决方法是将构造函数或析构函数的整个主体转移到一个私有的空洞返回方法。但是，您应该意识到构造函数中的致命断言失败并不会终止当前测试，这与您的直觉是一致的;它只是提前从构造函数返回，可能使对象处于部分构造的状态。同样，析构函数中的致命断言失败可能会使对象处于部分析构状态。在这些情况下小心使用断言!

# 教谷歌测试如何打印您的值

当像EXPECT\_EQ这样的测试断言失败时，谷歌测试将打印参数值来帮助您调试。它使用一个用户可扩展的值打印机来实现这一点。

该打印机知道如何打印内置的c++类型、本机数组、STL容器和任何支持<<操作符的类型。对于其他类型，它将在值中打印原始字节，并希望您(用户)能够找到它。

如前所述，打印机是可扩展的。这意味着您可以教它更好地打印您的特定类型，而不是转储字节。为此，为您的类型定义<<:

#include <iostream>命名空间foo {class Bar{…  
  
  
  
};//我们希望谷歌测试能够打印这个实例。  
  
//在定义Bar的//名称空间中定义<<操作符是很重要的。  
c++的查找规则依赖于此。  
  
//将bar打印到os}} // namespace foo所需要的任何东西

有时，这可能不是一个选项:您的团队可能认为Bar有一个<<操作符是一种糟糕的风格，或者Bar可能已经有一个<<操作符，它不能做您想做的事情(并且您不能更改它)。如果是这样，您可以定义一个PrintTo()函数，如下所示:

#include <iostream>命名空间foo {class Bar{…  
  
  
  
在定义Bar的//名称空间中定义PrintTo()是很重要的。  
  
  
c++的查找规则依赖于此。  
void PrintTo(const bar & bar，::std::ostream\* os) {\*os << bar. debugstring ();  
//将bar打印到os}} // namespace foo所需要的任何东西

如果您同时定义了<<和PrintTo()，则在涉及到谷歌测试时将使用后者。这允许您在不影响依赖于其<<操作符行为的代码的情况下，自定义值如何出现在谷歌测试的输出中。

如果您希望自己使用谷歌测试的值打印机打印一个值x，只需调用::testing::PrintToString(x)，它将返回一个std::string:

第三个方法是:使用bintvector (bar\_ints)来创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数，它的作用是:创建一个新的函数。

# 死亡测试

在许多应用程序中，如果条件不满足，断言可能导致应用程序失败。这些完整性检查(确保程序处于已知的良好状态)是为了在某些程序状态被破坏后尽早失败。如果断言检查了错误的条件，则程序可能处于错误状态，这可能导致内存损坏、安全漏洞或更糟的情况。因此，测试这样的断言语句是否按预期工作非常重要。

因为这些前置条件检查会导致进程死亡，所以我们称这样的测试为死亡测试。更一般地说，任何检查程序是否按预期方式终止(抛出异常除外)的测试也是死亡测试。

请注意，如果一段代码抛出一个异常，我们不认为它是“死亡”的死亡测试，因为代码调用者可以捕获异常并避免崩溃。如果希望验证代码抛出的异常，请参阅异常断言。

如果您想测试测试代码中的EXPECT\_\*()/ASSERT\_\*()故障，请参阅捕获故障。

## 如何编写死亡测试

谷歌测试有以下宏来支持死亡测试:

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_DEATH(声明中,正则表达式); | EXPECT\_DEATH(声明中,正则表达式); | *语句会因为给定的错误而崩溃* |
| ASSERT\_DEATH\_IF\_SUPPORTED(声明中,正则表达式); | EXPECT\_DEATH\_IF\_SUPPORTED(声明中,正则表达式); | 如果支持死亡测试，则用给定的错误验证语句崩溃;否则验证什么 |
| ASSERT\_EXIT(声明、谓词regex); | EXPECT\_EXIT(声明、谓词regex); | *语句使用给定的错误退出，其退出代码匹配谓词* |

其中，语句是预期导致进程死亡的语句，谓词是计算整数退出状态的函数或函数对象，regex是预期语句的stderr输出将匹配的正则表达式。注意，语句可以是任何有效的语句(包括复合语句)，并且不必是表达式。

与往常一样，ASSERT变体终止当前测试函数，而EXPECT变体不终止。

**注意:我们在这里使用“崩溃”一词表示进程以非零退出状态码结束。**有两种可能性:进程使用非零值调用exit()或\_exit()，或者可能被信号终止。

这意味着，如果语句使用0退出码终止进程，它不会被EXPECT\_DEATH认为是崩溃。如果是这种情况，或者希望更精确地限制退出代码，则使用EXPECT\_EXIT。

这里的谓词必须接受int并返回bool。只有谓词返回true时，死亡测试才会成功。谷歌测试定义了一些处理最常见情况的谓词:

::测试::ExitedWithCode (exit\_code)

如果程序使用给定的退出码正常退出，则该表达式为真。

:testing::KilledBySignal(signal\_number) //在Windows上不可用。

如果程序被给定的信号终止，则该表达式为真。

\*\_DEATH宏是\*\_EXIT的方便包装器，它使用一个谓词来验证进程的退出码是否为非零。

请注意，死亡测试只关心三件事:

1. 语句是否中止或退出进程?
2. (在ASSERT\_EXIT和EXPECT\_EXIT的情况下)退出状态是否满足谓词?或者(在ASSERT\_DEATH和EXPECT\_DEATH的情况下)退出状态是否为非零?和
3. stderr输出是否与regex匹配?

特别是，如果语句生成ASSERT\_\*或EXPECT\_\*失败，它不会导致死亡测试失败，因为谷歌测试断言不会中止进程。

要编写死亡测试，只需在测试函数中使用上述宏之一即可。例如,

{//这个死亡测试使用一个复合语句。  
  
ASSERT\_DEATH({int n = 5;Foo (n);}TEST(MyDeathTest, NormalExit) {EXPECT\_EXIT(NormalExit()，::testing::ExitedWithCode(0)， "Success";}TEST(MyDeathTest, KillMyself) {EXPECT\_EXIT(KillMyself()，::testing::KilledBySignal(SIGKILL)， "发送自己的不可阻塞的信号");

验证:

* 调用Foo(5)会导致进程死于给定的错误消息，
* 调用NormalExit()会导致进程将“Success”打印为stderr，退出时退出码为0和
* 调用KillMyself()使用SIGKILL信号杀死进程。

如果需要，测试函数体也可以包含其他断言和语句。

*重要提示:我们强烈建议您遵循命名您的测试用例(不是测试)\*死亡测试的约定，当它包含一个死亡测试时，如上例所示。*下面的死测试和线程一节解释了原因。

如果一个测试装置类被普通测试和死测试共享，你可以使用typedef来为这个装置类引入一个别名，并避免重复它的代码:

类FooTest: public::testing::Test{…TEST\_F(FooTest, DoesThis) {// normal test}TEST\_F(FooDeathTest, DoesThat) {// death test}

*可用性:Linux、Windows(要求MSVC 8.0或以上)、Cygwin和Mac(从v1.3.0开始支持后三种)。*\_DEATH\_IF\_SUPPORTED是v1.4.0中新增的。

## 正则表达式语法

在POSIX系统(例如Linux、Cygwin和Mac)上，谷歌测试在死测试中使用POSIX扩展正则表达式语法。要了解这种语法，您可能需要阅读这篇Wikipedia条目。

在Windows上，谷歌测试使用它自己的简单正则表达式实现。它缺乏POSIX扩展正则表达式中的许多特性。例如，我们不支持union(“x|y”)、分组(“(xy)”)、方括号(“[xy]”)和重复计数(“x{5,7}”)等。下面是我们所支持的(字母A表示文字字符、句点(.)或单个\\ escape序列;x和y表示正则表达式。

|  |  |
| --- | --- |
| c | 匹配任何文字字符c |
| \ \ d | 匹配任何十进制数字 |
| \ \ D | 匹配任何不是十进制数字的字符 |
| \ \ f | 匹配\ f |
| \ \ n | 匹配\ n |
| \ \ r | 匹配\ r |
| \ \年代 | 匹配任何ASCII空白，包括\n |
| \ \年代 | 匹配任何不是空白的字符 |
| \ \ t | 匹配\ t |
| \ \ v | 匹配\ v |
| \ \ w | 匹配任何字母、\_或十进制数字 |
| \ \ W | 匹配任何不匹配的字符 |
| \ \ c | 匹配任何文字字符c，该字符必须是标点符号 |
| \ \。 | 匹配。字符 |
| 。 | 匹配除\n以外的任何单个字符 |
| 一个吗? | 匹配0或1个出现的A |
| 一个\* | 匹配0或多次出现的A |
| A+ | 匹配一个或多个A |
| ^ | 匹配字符串的开头(而不是每一行的开头) |
| 美元 | 匹配字符串的结尾(不是每行的结尾) |
| xy | 匹配x和y |

为了帮助您确定系统上可用的功能，谷歌测试在使用POSIX扩展正则表达式时定义了宏GTEST\_USES\_POSIX\_RE=1，在使用简单版本时定义了GTEST\_USES\_SIMPLE\_RE=1。如果您希望死亡测试在这两种情况下都能工作，那么您可以对这些宏使用# If，或者只使用更有限的语法。

## 它是如何工作的

在底层，ASSERT\_EXIT()生成一个新进程，并在该进程中执行死亡测试语句。具体情况取决于平台和变量::testing::GTEST\_FLAG(death\_test\_style)(从命令行标志gtest\_death\_test\_style初始化)。

* 在POSIX系统上，fork()(或Linux上的clone())用于衍生子进程，然后:
  + 如果变量的值是“fast”，则立即执行死亡测试语句。
  + 如果变量的值是“threadsafe”，那么子进程将重新执行单元测试二进制文件，就像它最初被调用时一样，但是要使用一些额外的标记，以只运行考虑中的单个终止测试。
* 在Windows上，使用CreateProcess() API生成子进程，然后重新执行二进制代码，只运行考虑中的单一死亡测试——这与POSIX上的线程安全模式非常相似。

该变量的其他值是非法的，将导致死亡测试失败。目前，该标志的默认值是“fast”。但是，我们保留将来更改的权利。因此，您的测试不应该依赖于此。

无论哪种情况，父进程都会等待子进程完成并进行检查

1. 子元素的退出状态满足谓词，且
2. 子表达式的stderr与正则表达式匹配。

如果死亡测试语句运行到完成而没有死亡，子进程仍然会终止，并且断言失败。

## 死亡测试和线程

这两种死亡测试样式的原因与线程安全性有关。由于众所周知的存在线程的分叉问题，死测试应该在单线程上下文中运行。然而，有时安排那种环境是不可行的。例如，静态初始化的模块可能会在main到达之前启动线程。一旦创建了线程，就很难或不可能清理它们。

谷歌测试有三个旨在提高线程问题意识的特性。

1. 如果遇到死亡测试时多个线程正在运行，则会发出警告。
2. 以“死亡测试”结尾的测试用例在所有其他测试之前运行。
3. 它使用clone()而不是fork()在Linux上衍生子进程(clone()在Cygwin和Mac上不可用)，因为当父进程有多个线程时，fork()更可能导致子进程挂起。

在死亡测试语句中创建线程是完全可以的;它们在单独的进程中执行，不会影响父进程。

## 死亡测试风格

引入“线程安全”死亡测试风格是为了帮助降低在可能的多线程环境中进行测试的风险。它用增加的测试执行时间(可能会显著增加)来换取线程安全性的提高。我们建议使用更快的默认“快速”样式，除非您的测试有特定的问题。

你可以选择一个特定的风格的死亡测试设置标志编程:

::测试::FLAGS\_gtest\_death\_test\_style =“线程安全”;

您可以在main()中这样做，为二进制或个别测试中的所有死亡测试设置样式。回想一下，在运行每个测试之前会保存标志，在测试之后会恢复标志，所以您不需要自己做这件事。例如:

TEST(MyDeathTest, TestOne) {::testing::FLAGS\_gtest\_death\_test\_style = "threadsafe";  
  
//这个测试运行在"线程安全"的风格:ASSERT\_DEATH(ThisShouldDie()， "");}测试(MyDeathTest, TestTwo){//这个测试运行在"快速"的风格:ASSERT\_DEATH(ThisShouldDie()， "");}int main(int argc, char\*\* argv) {::testing::InitGoogleTest(&argc, argv);  
  
  
  
  
  
  
  
  
  
  
::测试::FLAGS\_gtest\_death\_test\_style =“快速”;  
返回RUN\_ALL\_TESTS ();}

## 警告

ASSERT\_EXIT()的语句参数可以是任何有效的c++语句。如果通过返回语句或抛出异常离开当前函数，则认为死亡测试失败。一些谷歌测试宏可能会从当前函数返回(例如ASSERT\_TRUE())，所以一定要在语句中避免它们。

由于语句在子进程中运行，因此它在内存中的任何副作用(例如修改变量、释放内存等)都不会在父进程中被观察到。特别是，如果在死亡测试中释放内存，您的程序将无法通过堆检查，因为父进程永远不会看到内存回收。要解决这个问题，你可以

1. 在死亡测试中尽量不要释放记忆;
2. 在父进程中再次释放内存;或
3. 不要在程序中使用堆检查器。

由于实现细节，您不能将多个死亡测试断言放在同一行;否则，编译将失败，错误消息不明显。

尽管“threadsafe”风格的死亡测试提高了线程安全性，但在使用pthread\_atfork注册的处理程序(3)时，仍然可能出现死锁等线程问题。

# 在子例程中使用断言

## 向断言添加跟踪

如果从多个地方调用测试子例程，当其中的断言失败时，很难判断失败来自子例程的哪个调用。您可以使用额外的日志记录或自定义失败消息来缓解这个问题，但这通常会使您的测试变得混乱。更好的解决方案是使用SCOPED\_TRACE宏:

| SCOPED\_TRACE(消息);| |:- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - |

消息可以是任何流的std::ostream。此宏将导致在每个失败消息中添加当前文件名、行号和给定消息。当控件离开当前词法作用域时，该效果将被撤消。

例如,

10: void Sub1(int n) {11: EXPECT\_EQ(1, Bar(n));12: EXPECT\_EQ(2, Bar(n + 1));13:}14:15: TEST(FooTest, Bar) {16: {17: SCOPED\_TRACE("A");  
  
  
  
  
  
  
//这个跟踪点将包含在18://这个范围内的每个故障中。  
  
21: //现在不会了。  
  
  
22:Sub1 (9); 23:}

可能会导致这样的信息:

路径/ / foo\_test。cc:11: FailureValue of: Bar(n)  
  
  
  
  
答:17:Apath / / foo\_test。  
  
cc:12:故障值:Bar(n + 1)期望:2实际:3

如果没有跟踪，就很难知道这两个失败分别来自于Sub1()的哪个调用。(您可以向Sub1()中的每个断言添加一条额外的消息，以指示n的值，但这很繁琐。)

使用SCOPED\_TRACE的一些技巧:

1. 对于合适的消息，通常在子例程的开头使用SCOPED\_TRACE就足够了，而不是在每个调用站点。
2. 当在循环中调用子例程时，使SCOPED\_TRACE中的循环迭代器成为消息的一部分，这样您就可以知道失败来自哪个迭代。
3. 有时，跟踪点的行号对于标识子例程的特定调用就足够了。在这种情况下，不必为SCOPED\_TRACE选择唯一的消息。你可以简单地使用“”。
4. 当外部范围中有SCOPED\_TRACE时，可以在内部范围中使用它。在这种情况下，所有活动跟踪点都将包含在失败消息中，它们的出现顺序是相反的。
5. 跟踪转储在Emacs的编译缓冲区中是可单击的——在行号上单击return，您将被带到源文件中的该行!

*可用性:Linux、Windows、Mac。*

## 传播致命的失败

使用ASSERT\_\*和FAIL\*时的一个常见陷阱是不理解当它们失败时，它们只中止当前函数，而不是整个测试。例如，下面的测试将会出现segfault:

void子例程(){//生成一个致命错误并中止当前函数。  
  
ASSERT\_EQ (1、2);  
//以下不会执行…  
  
}TEST(FooTest, Bar){子例程();  
  
  
  
//预期的行为是子例程()中的致命错误//终止整个测试。  
  
//实际行为:函数在子例程()返回后继续执行。  
int\* p = NULL;  
\* p = 3;/ /段错误!}

由于我们不使用异常，因此在这里实现预期的行为在技术上是不可能的。为了缓解这个问题，谷歌测试提供了两个解决方案。您可以使用(ASSERT|EXPECT)\_NO\_FATAL\_FAILURE断言或HasFatalFailure()函数。它们将在下面的两个小节中进行描述。

### 主张在子例程

如上所示，如果您的测试调用了一个子例程，其中包含ASSERT\_\* failure，那么在该子例程返回之后，测试将继续进行。这可能不是你想要的。

人们通常希望致命的失败像异常一样传播。对于谷歌测试提供以下宏:

|  |  |  |
| --- | --- | --- |
| **致命的断言** | **非致命的断言** | **验证** |
| ASSERT\_NO\_FATAL\_FAILURE(声明); | EXPECT\_NO\_FATAL\_FAILURE(声明); | *语句不会在当前线程中生成任何新的致命错误。* |

只检查执行断言的线程中的故障，以确定这种类型断言的结果。如果语句创建了新线程，则忽略这些线程中的故障。

例子:

ASSERT\_NO\_FATAL\_FAILURE({i = Bar();});

*可用性:Linux、Windows、Mac。目前不支持来自多个线程的断言。*

### 检查当前测试中的故障

Test类中的HasFatalFailure()如果当前测试中的断言发生了致命的错误，则返回true。这允许函数捕获子例程中的致命错误并提前返回。

类测试{public:…  
  
  
静态bool HasFatalFailure ();};

典型的用法是:

测试(FooTest, Bar){子例程();  
  
如果子例程()发生致命错误，则中止。  
如果(HasFatalFailure())返回;  
  
//以下不会执行…

如果HasFatalFailure()在TEST()、TEST\_F()或测试装置之外使用，则必须添加::testing:: TEST::前缀，如:

如果(::测试:测试:HasFatalFailure())返回;

类似地，如果当前测试至少有一个非致命错误，则HasNonfatalFailure()返回true;如果当前测试至少有一个非致命错误，则HasFailure()返回true。

*可用性:Linux、Windows、Mac. HasNonfatalFailure()和HasFailure()自1.4.0版本起就可用。*

# 日志记录附加信息

在您的测试代码中，您可以调用RecordProperty(“key”，value)来记录额外的信息，其中的值可以是字符串，也可以是整数。例如，测试

TEST\_F(WidgetUsageTest, MinAndMaxWidgets) {RecordProperty("MaximumWidgets"， ComputeMaxUsage());  
  
RecordProperty (“MinimumWidgets ComputeMinUsage ());}

将输出这样的XML:

…  
<testcase name="MinAndMaxWidgets" status="run" time="6" classname="WidgetUsageTest" MaximumWidgets="12" MinimumWidgets="9" />…

*注意:*

* RecordProperty()是测试类的静态成员。因此，如果在测试体和测试夹具类之外使用，则需要在前面加上::testing::Test:: Test::。
* 键必须是一个有效的XML属性名，并且不能与谷歌测试已经使用的那些属性名(名称、状态、时间、类名、type\_param和value\_param)冲突。
* 允许在测试的生命周期之外调用RecordProperty()。如果它是在测试之外调用的，但是是在测试用例的SetUpTestCase()和TearDownTestCase()方法之间调用的，那么它将被归于测试用例的XML元素。如果在所有测试用例(例如，在测试环境中)之外调用它，它将归属于顶级XML元素。

*可用性:Linux、Windows、Mac。*

# 在相同测试用例中的测试之间共享资源

谷歌测试为每个测试创建一个新的测试装置对象，以使测试独立且更容易调试。但是，有时测试使用的资源设置起来很昂贵，使得每个测试只复制一个模型的代价非常昂贵。

如果测试不更改资源，那么它们共享一个资源副本也没有害处。因此，除了每个测试设置/拆解之外，谷歌测试还支持每个测试用例设置/拆解。使用它:

1. 在您的测试装置类(如FooTest)中，将一些成员变量定义为静态变量来保存共享资源。
2. 在同一个测试装置类中，定义一个静态void SetUpTestCase()函数(记住不要将它拼写为带有一个小u的SetUpTestCase !)来设置共享资源，并定义一个静态void TearDownTestCase()函数来销毁它们。

就是这样!谷歌测试在运行FooTest测试用例中的第一个测试之前(即创建第一个FooTest对象之前)自动调用SetUpTestCase()，在运行最后一个测试之后(即删除最后一个FooTest对象之后)自动调用TearDownTestCase()。在这之间，测试可以使用共享的资源。

请记住，测试顺序是未定义的，因此您的代码不能依赖于前面或后面的测试。此外，测试必须要么不修改任何共享资源的状态，要么，如果它们确实修改了状态，它们必须在将控制权传递给下一个测试之前将状态恢复到其原始值。

下面是每个测试用例设置和拆卸的例子:

类FooTest: public::testing::Test {protected: //每个测试用例设置。  
  
  
在这个测试用例的第一个测试之前调用。  
如果不需要可以省略。  
静态void SetUpTestCase() {shared\_resource\_ = new…  
  
} //每个测试用例的拆解。  
  
  
在这个测试用例的最后一个测试之后调用。  
如果不需要可以省略。  
静态void TearDownTestCase(){删除shared\_resource\_;  
  
shared\_resource\_ =零;  
} //你可以像往常一样定义每个测试的设置和拆卸逻辑。  
  
  
虚拟空间设置(){…虚拟空间TearDown(){…  
} //一些由所有测试共享的昂贵资源。  
  
  
\* FooTest::shared\_resource\_ = NULL;TEST\_F(FooTest, Test1){…  
  
  
  
  
  
你可以在这里引用shared\_resource…  
TEST\_F(FooTest, Test2){…  
  
你可以在这里引用shared\_resource…}

*可用性:Linux、Windows、Mac。*

# 全球设置和拆卸

正如您可以在测试级别和测试用例级别进行设置和拆卸一样，您也可以在测试程序级别进行设置和拆卸。这是如何。

首先，子类化::testing::Environment类来定义一个测试环境，它知道如何设置和拆卸:

类Environment {public: virtual ~Environment(){} //覆盖它来定义如何设置环境。  
  
  
  
virtual void SetUp(){} //覆盖它来定义如何破坏环境。  
  
虚拟void TearDown() {}};

然后，通过调用::testing::AddGlobalTestEnvironment()函数，用谷歌测试注册环境类的一个实例:

\* env环境\* AddGlobalTestEnvironment(环境);

现在，当调用RUN\_ALL\_TESTS()时，它首先调用环境对象的SetUp()方法，然后在没有致命错误的情况下运行测试，最后调用环境对象的TearDown()。

可以注册多个环境对象。在这种情况下，将按注册的顺序调用它们的SetUp()，而按相反的顺序调用它们的TearDown()。

注意，谷歌测试获取注册环境对象的所有权。因此，不要自己删除它们。

您应该在调用RUN\_ALL\_TESTS()之前调用AddGlobalTestEnvironment()，可能是在main()中。如果使用gtest\_main，则需要在main()启动之前调用它，以使其生效。一种方法是这样定义一个全局变量:

::test::Environment\* const foo\_env =::testing::AddGlobalTestEnvironment(新FooEnvironment);

然而,我们强烈建议你写自己的主要()和调用AddGlobalTestEnvironment(),依赖于初始化的全局变量使代码难以阅读和可能引起的问题当你注册多个环境中从不同的翻译单元和环境之间的依赖关系(记住,编译器并不能保证全局变量的顺序从不同的翻译单元初始化)。

*可用性:Linux、Windows、Mac。*

# 值参数化测试

*值参数化测试允许您使用不同的参数测试您的代码，而不需要编写相同测试的多个副本。*

假设您为代码编写了一个测试，然后发现代码受到布尔命令行标志的影响。

TEST(MyCodeTest, TestFoo){//一个测试foo()的代码。}

在这种情况下，人们通常会将测试代码分解成带有布尔参数的函数。该函数设置标记，然后执行测试代码。

void TestFooHelper(bool flag\_value) {flag = flag\_value;  
  
//一个测试foo()的代码。  
}TEST(MyCodeTest, TestFoo) {TestFooHelper(false);  
  
  
  
TestFooHelper(真正);}

但是这种设置有严重的缺陷。首先，当测试断言在您的测试中失败时，就不清楚是参数的什么值导致了它的失败。您可以将澄清消息流到EXPECT/ASSERT语句中，但是必须对所有这些语句进行处理。其次，您必须为每个测试添加一个这样的帮助函数。如果你有10个测试呢?二十个?一百年?

值参数化测试将允许您只编写一次测试，然后使用任意数量的参数值轻松实例化和运行测试。

这里有一些其他的情况，当值参数化测试变得方便:

* 您需要测试OO接口的不同实现。
* 您希望在各种输入上测试代码(也称为数据驱动测试)。这个功能很容易被滥用，所以请在使用它的时候锻炼你的判断力!

## 如何编写值参数化测试

要编写值参数化的测试，首先应该定义一个fixture类。它必须派生自::testing::Test和::testing::WithParamInterface<T>(后者是一个纯接口)，其中T是参数值的类型。为了方便起见，可以直接从::testing::TestWithParam<T>派生出fixture类，它本身派生自::testing::Test和::testing::WithParamInterface<T>。T可以是任何可复制的类型。如果是原始指针，则负责管理所指向值的生命周期。

类FooTest: public::testing::TestWithParam<const char\*>{//你可以在这里实现所有常用的fixture类成员。  
  
//要访问测试参数，可以从class // TestWithParam<T>.}中调用GetParam();

然后，使用TEST\_P宏来定义尽可能多的使用这个fixture的测试模式。\_P后缀表示“参数化”或“模式”，随您喜欢。

TEST\_P(FooTest, DoesBlah){//在测试内部，使用TestWithParam<T>类的GetParam()方法//访问测试参数:EXPECT\_TRUE(foo.Blah(GetParam());  
  
  
  
…  
TEST\_P(FooTest, HasBlahBlah){…}

最后，您可以使用INSTANTIATE\_TEST\_CASE\_P用您想要的任何一组参数来实例化测试用例。谷歌测试定义了许多用于生成测试参数的函数。它们返回我们所称的参数生成器。下面是它们的摘要，它们都在测试名称空间中:

|  |  |
| --- | --- |
| 范围(开始、结束(步骤) | 生成值{begin, begin+step, begin+step+step，…}。这些值不包括end。step默认为1。 |
| 值(v1、v2,…vN) | 生成值{v1, v2，…vN}。 |
| ValuesIn(容器)和ValuesIn(开始，结束) | 从c样式数组、stl样式容器或迭代器范围[begin, end]中生成值。容器、开始和结束可以是其值在运行时确定的表达式。 |
| 布尔值() | 产生序列{false, true}。 |
| 结合(g1, g2,…gN) | 生成所有由N个生成器生成的值的组合(用于数学计算的笛卡尔积)。只有在您的系统提供<tr1/tuple>标头时才可以使用。如果您确信您的系统已经这样做了，而谷歌测试不同意，那么您可以通过定义GTEST\_HAS\_TR1\_TUPLE=1来覆盖它。参见include/gtest/internal/gtest-port中的注释。[h获取更多信息。](../include/gtest/internal/gtest-port.h) |

有关更多细节，请参见源代码中这些函数定义的注释。

下面的语句将用参数值“meeny”、“miny”和“moe”实例化来自FooTest测试用例的测试。

test\_test\_case\_p (InstantiationName, FooTest，::testing::Values("meeny"， "miny"， "moe"));

为了区分模式的不同实例(是的，您可以不止一次地实例化它)，INSTANTIATE\_TEST\_CASE\_P的第一个参数是一个将添加到实际测试用例名称中的前缀。记住，要为不同的实例化选择唯一的前缀。上面实例化的测试将有这些名称:

* InstantiationName /脚。“meeny”DoesBlah / 0
* InstantiationName /脚。“如矿坑的”DoesBlah / 1
* InstantiationName /脚。“萌”DoesBlah / 2
* InstantiationName /脚。“meeny”HasBlahBlah / 0
* InstantiationName /脚。“如矿坑的”HasBlahBlah / 1
* InstantiationName /脚。“萌”HasBlahBlah / 2

您可以在——gtest\_filter中使用这些名称。

这个语句将再次实例化来自FooTest的所有测试，每个测试都有参数值“cat”和“dog”:

实例化\_test\_case\_p(另一个实例化名，FooTest，::testing::ValuesIn(pets));

上面实例化的测试将有这些名称:

* AnotherInstantiationName /脚。“猫”DoesBlah / 0
* AnotherInstantiationName /脚。“狗”DoesBlah / 1
* AnotherInstantiationName /脚。“猫”HasBlahBlah / 0
* AnotherInstantiationName /脚。“狗”HasBlahBlah / 1

请注意，INSTANTIATE\_TEST\_CASE\_P将实例化给定测试用例中的所有测试，无论它们的定义是在INSTANTIATE\_TEST\_CASE\_P语句之前还是之后。

有关更多示例，可以查看这些文件。

*可用性:Linux, Windows(要求MSVC 8.0或以上)，Mac;*自从1.2.0版本。

## 创建值参数化的抽象测试

在上面的例子中，我们在同一个源文件中定义和实例化了FooTest。有时，您可能希望在库中定义值参数化的测试，并让其他人稍后实例化它们。这种模式称为抽象测试。作为它的应用程序的一个例子，当您设计一个接口时，您可以编写一个标准的抽象测试套件(可能使用一个工厂函数作为测试参数)，所有接口的实现都应该通过该套件。当某人实现接口时，他可以实例化您的套件来免费获得所有接口一致性测试。

要定义抽象测试，你应该这样组织你的代码:

1. 将参数化测试装置类(例如FooTest)的定义放在头文件中，比如foo\_param\_test.h。可以把这看作是对抽象测试的声明。
2. 将TEST\_P定义放在foo\_param\_test中。cc，其中包括foo\_param\_test.h。可以将其视为实现抽象测试。

一旦定义了它们，就可以通过包含foo\_param\_test来实例化它们。调用INSTANTIATE\_TEST\_CASE\_P()，并链接到foo\_param\_test.cc。您可以多次实例化同一个抽象测试用例，可能是在不同的源文件中。

# 输入测试

假设您有相同接口的多个实现，并且希望确保所有实现都满足某些公共需求。或者，您可能已经定义了几个类型，这些类型应该符合相同的“概念”，您希望验证它。在这两种情况下，您都希望对不同类型重复相同的测试逻辑。

虽然您可以编写一个测试或TEST\_F每种类型您想要测试(你甚至可能因素测试逻辑放在一个函数模板,您调用的测试),它是乏味的和不规模:如果你想要/ n m测试类型,您将编写m \* n测试。

*类型化测试允许您在类型列表上重复相同的测试逻辑。*您只需要编写一次测试逻辑，尽管在编写类型化测试时必须知道类型列表。你可以这样做:

首先，定义一个fixture类模板。它应该由类型参数化。记住要从::testing::Test:

模板<typename T>类FooTest: public::testing::Test {public:…  
  
  
  
typedef std:: < T >列表;  
静态T shared\_;  
T value\_;};

接下来，将一个类型列表与测试用例相关联，测试用例将对列表中的每个类型重复:

typedef::testing::Types<char, int, unsigned int> MyTypes;TYPED\_TEST\_CASE(FooTest, MyTypes);

typedef是TYPED\_TEST\_CASE宏正确解析所必需的。否则，编译器会认为类型列表中的每个逗号都引入了一个新的宏参数。

然后，使用TYPED\_TEST()而不是TEST\_F()来为这个测试用例定义一个类型化测试。你可以想重复多少次就重复多少次:

TYPED\_TEST(FooTest, DoesBlah){//在一个测试中，引用特殊的名称TypeParam来获取类型//参数。  
  
因为我们在一个派生类模板中，c++要求//我们通过“this”访问FooTest的成员。  
  
TypeParam n = this->value\_;  
  
//若要访问fixture的静态成员，请添加'TestFixture::' //前缀。  
  
n + =有关::shared\_;  
  
要引用fixture中的typedefs，请添加“typename TestFixture::”//前缀。  
需要'typename'来满足编译器。  
typename有关:列表值;  
values.push\_back (n);  
…  
}TYPED\_TEST(FooTest, HasPropertyA){…  
  
}

您可以看到samples/sample6\_unittest。一个完整的例子。

*可用性:Linux, Windows(要求MSVC 8.0或以上)，Mac;*因为版本1.1.0。

# 参数化测试

*类型参数化测试类似于类型化测试，只是它们不需要您提前知道类型列表。*相反，您可以先定义测试逻辑，然后用不同的类型列表实例化它。您甚至可以在同一个程序中多次实例化它。

如果正在设计接口或概念，则可以定义一组类型参数化测试，以验证接口/概念的任何有效实现都应该具有的属性。然后，每个实现的作者可以用他的类型实例化测试套件来验证它是否符合需求，而不必重复编写类似的测试。这里有一个例子:

首先，定义一个fixture类模板，就像我们在类型化测试中做的那样:

模板<typename T>类FooTest: public::testing::Test{…};

接下来，声明您将定义一个类型参数化的测试用例:

TYPED\_TEST\_CASE\_P(脚);

\_P后缀表示“参数化”或“模式”，随您喜欢。

然后，使用TYPED\_TEST\_P()来定义一个类型参数化的测试。你可以想重复多少次就重复多少次:

TYPED\_TEST\_P(FooTest, DoesBlah){//在测试中，引用TypeParam获取类型参数。  
  
类型参数n = 0;  
…  
}TYPED\_TEST\_P(FooTest, HasPropertyA){…  
  
}

现在是棘手的部分:您需要使用REGISTER\_TYPED\_TEST\_CASE\_P宏注册所有测试模式，然后才能实例化它们。宏的第一个参数是测试用例名称;其余是测试用例中的测试名称:

REGISTER\_TYPED\_TEST\_CASE\_P(脚、DoesBlah HasPropertyA);

最后，您可以自由地用需要的类型实例化模式。如果您将上面的代码放在头文件中，您可以将它包含在多个c++源文件中，并实例化它多次。

类型<char, int, unsigned int> MyTypes;实例化\_typed\_test\_case\_p (My, FooTest, MyTypes);

为了区分模式的不同实例，INSTANTIATE\_TYPED\_TEST\_CASE\_P宏的第一个参数是一个将添加到实际测试用例名称中的前缀。记住要为不同的实例选择唯一的前缀。

在类型列表只包含一种类型的特殊情况下，您可以直接编写该类型，而不需要::testing::Types<…>,是这样的:

INSTANTIATE\_TYPED\_TEST\_CASE\_P(我的脚,int);

您可以看到samples/sample6\_unittest。一个完整的例子。

*可用性:Linux, Windows(要求MSVC 8.0或以上)，Mac;*因为版本1.1.0。

# 测试私人代码

如果您更改了软件的内部实现，只要用户看不到更改，测试就不应该中断。因此，根据黑盒测试原则，大多数时候应该通过代码的公共接口来测试代码。

如果您仍然需要测试内部实现代码，请考虑是否有更好的设计不需要您这样做。如果您必须测试非公共接口代码，您可以这样做。有两种情况需要考虑:

* 静态函数(与静态成员函数不同!)或未命名的名称空间，以及
* 私有或受保护的类成员

## 静态函数

静态函数和未命名名称空间中的定义/声明只在同一个翻译单元中可见。要测试它们，可以在\*\_test中测试整个.cc文件。cc文件。(#包含。cc文件不是重用代码的好方法——您不应该在生产代码中这样做!)

但是，更好的方法是将私有代码移动到foo::internal名称空间中，其中foo是您的项目通常使用的名称空间，并将私有声明放在\*-internal中。h文件。您的产品。cc文件和测试可以包含这个内部标头，但是您的客户端不可以。通过这种方式，您可以完全测试您的内部实现，而不会将其泄露给您的客户端。

## 私人类成员

私有类成员只能从类内部或由朋友访问。要访问类的私有成员，可以将测试装置声明为类的朋友，并在装置中定义访问器。然后，使用fixture的测试可以通过fixture中的访问器访问生产类的私有成员。请注意，即使您的fixture与您的产品类是朋友关系，您的测试也不会自动成为它的朋友，因为它们在技术上是在fixture的子类中定义的。

测试私有成员的另一种方法是将它们重构到一个实现类中，然后在\*-internal中声明该类。h文件。您的客户端不允许包含这个头，但是您的测试可以。这就是所谓的Pimpl(私有实现)风格。

或者，您可以通过在类体中添加这一行来声明一个单独的测试:

FRIEND\_TEST (TestCaseName TestName);

例如,

/ / foo。h # include“gt / gtest\_prod。  
h”/ /定义FRIEND\_TEST。  
  
  
类Foo{…  
  
私人:FRIEND\_TEST(脚、BarReturnsZeroOnNull);  
  
// foo\_test.cc…  
  
  
  
  
TEST(FooTest, BarReturnsZeroOnNull) {Foo Foo;  
  
EXPECT\_EQ (0, foo.Bar (NULL));  
//使用Foo的private member Bar().}

在名称空间中定义类时要特别注意，因为如果希望测试装置和测试成为类的朋友，就应该在相同的名称空间中定义它们。例如，如果要测试的代码如下:

朋友类FooTest;  
  
  
  
FRIEND\_TEST(脚、酒吧);  
FRIEND\_TEST(脚,Baz);  
…  
类Foo…};}// namespace my\_namespace的定义

你的测试代码应该是这样的:

TEST\_F(FooTest, Bar){…  
  
  
  
  
  
TEST\_F(FooTest, Baz){…  
//命名空间my\_namespace

# 捕捉失败

如果您正在谷歌测试的基础上构建一个测试实用程序，那么您需要测试您的实用程序。您将使用什么框架来测试它?当然是谷歌测试。

挑战在于验证您的测试工具是否正确地报告了故障。在通过抛出异常报告失败的框架中，您可以捕获异常并对其进行断言。但是谷歌测试不使用异常，所以我们如何测试一段代码生成预期的失败?

“gt / gtest-spi。h"包含一些构造来完成这个。包含了这个标题之后，就可以使用了

| EXPECT\_FATAL\_FAILURE(声明,substring);| |:- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - |

若要断言该语句，则会生成一个致命的失败(例如ASSERT\_\*)，其消息包含给定的子字符串或use

| EXPECT\_NONFATAL\_FAILURE(声明,substring);| |:- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - |

如果您正在期待一个非致命的失败(例如EXPECT\_\*)。

出于技术原因，这里有一些警告:

1. 您不能将失败消息流到任何一个宏。
2. *EXPECT\_FATAL\_FAILURE()中的语句不能引用该对象的本地非静态变量或非静态成员。*
3. *语句EXPECT\_FATAL\_FAILURE()不能返回值。*

*注意:谷歌测试在设计时考虑了线程。*一旦同步原语在“gtest/internal/gtest-port”。实现后，谷歌测试将成为线程安全的，这意味着您可以在多个线程中并发地使用断言。然而，在此之前，谷歌测试只支持单线程使用。一旦线程安全，EXPECT\_FATAL\_FAILURE()和EXPECT\_NONFATAL\_FAILURE()将仅捕获当前线程中的故障。如果语句创建了新线程，这些线程中的失败将被忽略。如果你想从所有线程捕获失败，你应该使用以下宏:

|  |
| --- |
| EXPECT\_FATAL\_FAILURE\_ON\_ALL\_THREADS(声明,substring); |
| EXPECT\_NONFATAL\_FAILURE\_ON\_ALL\_THREADS(声明,substring); |

# 获取当前测试的名称

有时函数可能需要知道当前运行的测试的名称。例如，您可能正在使用您的测试装置的SetUp()方法来根据正在运行的测试设置黄金文件名。TestInfo类有以下信息:

名称空间测试{类TestInfo {public: //分别返回测试用例名称和测试名称。  
  
  
  
  
// //不要删除或释放返回值——它是由// TestInfo类管理的。  
  
  
\* test\_case\_name();  
//名称空间测试

要为当前运行的测试获取一个TestInfo对象，在UnitTest单例对象上调用current\_test\_info():

获取当前正在运行的测试的相关信息。  
//不要删除返回的对象——它是由UnitTest类管理的。  
const::testing::TestInfo\* const test\_info =::testing::UnitTest::GetInstance()->current\_test\_info();  
  
test\_info \ n”——>名称(),test\_info - > test\_case\_name ());

如果没有运行测试，则current\_test\_info()返回一个空指针。特别是，您不能在TestCaseSetUp()、TestCaseTearDown()(其中您隐式地知道测试用例的名称)或从它们调用的函数中找到测试用例的名称。

*可用性:Linux、Windows、Mac。*

# 通过处理测试事件扩展谷歌测试

谷歌测试提供了一个事件监听器API，可以让您接收有关测试程序进度和测试失败的通知。可以侦听的事件包括测试程序的开始和结束、测试用例或测试方法等。您可以使用这个API来扩展或替换标准控制台输出、替换XML输出，或者提供完全不同的输出形式，例如GUI或数据库。例如，您还可以使用测试事件作为检查点来实现资源泄漏检查器。

*可用性:Linux、Windows、Mac;*因为v1.4.0。

## 定义事件监听器

要定义一个事件监听器，可以子类化test::TestEventListener或test::EmptyTestEventListener。前者是一个(抽象)接口，其中可以覆盖每个<br>的纯虚方法来处理一个测试事件(例如，当一个测试开始时，将调用OnTestStart()方法)。后者提供了接口中所有方法的空实现，这样子类只需覆盖它关心的方法。

当触发事件时，它的上下文作为参数传递给处理程序函数。使用以下参数类型:

* [UnitTest反映了整个测试程序的状态，](../include/gtest/gtest.h#L1151)
* [TestCase有关于一个测试用例的信息，它可以包含一个或多个测试，](../include/gtest/gtest.h#L778)
* [TestInfo包含测试的状态，以及](../include/gtest/gtest.h#L644)
* [TestPartResult表示测试断言的结果。](../include/gtest/gtest-test-part.h#L47)

事件处理程序函数可以检查它接收到的参数，以发现关于事件和测试程序状态的有趣信息。这里有一个例子:

类MinimalistPrinter: public::testing::EmptyTestEventListener{//在测试开始前调用。  
  
\*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*  
% s开始。\ n”, test\_info.test\_case\_name (), test\_info.name ());  
  
} //在断言失败或成功()调用后调用。  
  
  
printf("%s in %s:%d\n%s\n"， test\_part\_result.failed() ?  
  
  
"\*\*\*失败":"成功"，test\_part\_result.file\_name()， test\_part\_result.line\_number()， test\_part\_result.summary();  
  
  
  
} //测试结束后调用。  
  
  
{printf("\*\*\* \* Test %s。  
% s的结局。\ n”, test\_info.test\_case\_name (), test\_info.name ());  
  
}};

## 使用事件监听器

要使用您已定义的事件监听器，请将其实例添加到谷歌测试事件监听器列表中(由类testeventlistener表示)

* 在调用RUN\_ALL\_TESTS()之前，请注意main()函数中名称末尾的“s”:

int main(int argc, char\*\* argv) {::testing::InitGoogleTest(&argc, argv);  
  
获取事件监听器列表。  
::test:: testeventlisteners&listener =::testing::UnitTest::GetInstance()->listener ();  
  
//在结尾添加了一个监听器。谷歌测试取得所有权。  
听众。追加(新MinimalistPrinter);  
返回RUN\_ALL\_TESTS ();}

只有一个问题:默认的测试结果打印机仍然有效，所以它的输出将与极简打印机的输出混合在一起。要禁用默认打印机，只需从事件监听器列表中释放它并删除它。你可以通过添加一行:

…  
删除listeners.Release (listeners.default\_result\_printer ());  
听众。追加(新MinimalistPrinter);  
返回RUN\_ALL\_TESTS ();

现在，坐下来享受测试的完全不同的输出。要了解更多细节，您可以阅读这个示例。

您可以向列表追加多个侦听器。当触发On\*Start()或OnTestPartResult()事件时，侦听器将按照它们在列表中出现的顺序接收该事件(因为新侦听器被添加到列表的末尾，所以默认文本打印机和默认XML生成器将首先接收该事件)。侦听器将以相反的顺序接收一个On\*End()事件。这允许以后添加的侦听器的输出被以前添加的侦听器的输出框起来。

## 监听器产生故障

在处理事件时，可以使用失败引发宏(EXPECT\_\*()、ASSERT\_\*()、FAIL()等。有一些限制:

1. 您不能在OnTestPartResult()中生成任何错误(否则将导致递归调用OnTestPartResult())。
2. 处理OnTestPartResult()的侦听器不允许生成任何失败。

当您将侦听器添加到侦听器列表时，您应该将处理OnTestPartResult()的侦听器置于可能生成故障的侦听器之前。这确保了由后者产生的失败是由于前者的正确测试。

我们这里有一个失败引发侦听器的示例。

# 运行测试程序:高级选项

测试程序是普通的可执行程序。一旦构建完成，您就可以直接运行它们，并通过以下环境变量和/或命令行标志来影响它们的行为。要使这些标志起作用，您的程序必须在调用RUN\_ALL\_TESTS()之前调用::testing::InitGoogleTest()。

要查看受支持标志的列表及其用法，请使用—help标志运行您的测试程序。你也可以用-h， -?还是/ ?为短。该特性是在1.3.0版本中添加的。

如果一个选项由环境变量和标志指定，则后者优先。大多数选项也可以在代码中设置/读取:为了访问命令行标志——gtest\_foo的值，写::testing::GTEST\_FLAG(foo)。一个常见的模式是在调用::testing::InitGoogleTest()之前设置标志的值，以更改标志的默认值:

int main(int argc, char\*\* argv){//默认禁用运行时间。  
  
::测试::GTEST\_FLAG (print\_time) = false;  
  
这允许用户覆盖命令行上的标志。  
::测试::InitGoogleTest(命令行参数个数,argv);  
  
返回RUN\_ALL\_TESTS ();}

## 选择测试

本节展示了选择运行哪些测试的各种选项。

### 清单测试名称

有时有必要在运行程序之前列出可用的测试，以便在需要时应用筛选器。包含标志——gtest\_list\_tests覆盖所有其他标志，并以以下格式列出测试:

TestCase1。  
TestName1 TestName2TestCase2。  
  
  
TestName

如果提供了该标志，那么所列出的测试实际上都不会运行。此标志没有对应的环境变量。

*可用性:Linux、Windows、Mac。*

### 运行测试的子集

默认情况下，谷歌测试程序运行用户定义的所有测试。有时，您只想运行测试的一个子集(例如用于调试或快速验证更改)。如果将GTEST\_FILTER环境变量或——GTEST\_FILTER标志设置为筛选器字符串，则谷歌测试将只运行其全名(以TestCaseName.TestName的形式)与筛选器匹配的测试。

过滤器的格式是一个':'-分隔的通配符模式列表(称为正模式)，后面可选跟随一个'-'和另一个':'-分隔的模式列表(称为负模式)。当且仅当测试匹配任何积极模式但不匹配任何消极模式时，测试才匹配筛选器。

模式可以包含'\*'(匹配任何字符串)或'?(匹配任何单个字符)。为了方便起见，过滤器“\*-NegativePatterns”也可以写成“-NegativePatterns”。

例如:

* ./foo\_test没有标记，因此运行它的所有测试。
* ./foo\_test——gtest\_filter=\*也会运行所有的东西，因为只有一个match-everything \*值。
* / foo\_test——gtest\_filter =脚。\*在测试用例FooTest中运行所有东西。
* ./foo\_test——gtest\_filter=\*Null\*:\*Constructor\*运行任何全名包含“Null”或“Constructor”的测试。
* / foo\_test——gtest\_filter = - \* DeathTest。\*运行所有非死亡测试。
* / foo\_test——gtest\_filter =脚。\*英尺。Bar在测试用例FooTest中运行除FooTest.Bar之外的所有内容。

*可用性:Linux、Windows、Mac。*

### 暂时禁用测试

如果您有一个不能立即修复的坏掉的测试，您可以将DISABLED\_前缀添加到它的名称中。这将把它排除在执行之外。这比注释代码或使用#if 0要好，因为禁用的测试仍然会被编译(因此不会腐烂)。

如果您需要在一个测试用例中禁用所有测试，您可以将DISABLED\_添加到每个测试名称的前面，或者将它添加到测试用例名称的前面。

例如，下面的测试不会被谷歌测试运行，即使它们仍然会被编译:

//测试Foo执行Abc。  
TEST(FooTest, DISABLED\_DoesAbc){…类DISABLED\_BarTest: public::testing::Test{…  
  
};//测试Bar是否执行Xyz操作。  
  
  
TEST\_F(DISABLED\_BarTest, DoesXyz){…}

*注意:此功能仅用于暂时缓解疼痛。*您仍然需要在以后的某个日期修复禁用的测试。作为提醒，如果一个测试程序包含任何被禁用的测试，谷歌测试将打印一条横幅警告您。

*提示:使用grep可以轻松地计算禁用测试的数量。*这个数字可以作为改进测试质量的度量。

*可用性:Linux、Windows、Mac。*

### 临时启用禁用测试

要在测试执行中包含被禁用的测试，只需使用——gtest\_examda\_run\_disabled\_tests标志调用测试程序，或者将gtest\_examda\_run\_disabled\_tests环境变量设置为0以外的值。您可以将其与——gtest\_filter标志结合使用，以进一步选择要运行哪些禁用的测试。

*可用性:Linux、Windows、Mac;*因为1.3.0版本版本。

## 重复的测试

偶尔你会遇到一个结果是“打或不打”的测试。也许它失败的几率只有1%，这使得在调试器下重现错误变得相当困难。这可能是沮丧的主要来源。

—gtest\_repeat标志允许您在程序中多次重复所有(或选择的)测试方法。希望一个不可靠的测试最终会失败，并给您一个调试的机会。以下是使用方法:

|  |  |
| --- | --- |
| foo\_test——gtest\_repeat = 1000美元 | 重复foo\_test 1000次，失败时不要停止。 |
| foo\_test——gtest\_repeat = 1美元 | 负数意味着无限循环。 |
| $ foo\_test—gtest\_repeat=1000—gtest\_break\_on\_failure | 重复foo\_test 1000次，在第一次失败时停止。这在调试器下运行时特别有用:当测试失败时，它将进入调试器，然后您可以检查变量和堆栈。 |
| $ foo\_test—gtest\_repeat=1000—gtest\_filter=FooBar | 重复名称与筛选器匹配的测试1000次。 |

如果您的测试程序包含使用AddGlobalTestEnvironment()注册的全局设置/拆卸代码，那么它也将在每个迭代中重复，因为其中可能存在问题。您还可以通过设置GTEST\_REPEAT环境变量来指定重复计数。

*可用性:Linux、Windows、Mac。*

## 洗牌的测试

您可以指定——gtest\_shuffle标志(或者将gtest\_shuffle环境变量设置为1)来以随机顺序运行程序中的测试。这有助于揭示测试之间的不良依赖关系。

默认情况下，谷歌测试使用从当前时间计算的随机种子。因此每次你都会得到不同的顺序。控制台输出包含随机种子值，这样您可以在稍后重现与订单相关的测试失败。要显式地指定随机种子，使用——gtest\_random\_seed= seed标志(或设置gtest\_random\_seed环境变量)，其中seed是0到99999之间的整数。种子值0很特殊:它告诉谷歌测试执行从当前时间计算种子的默认行为。

如果您将它与——gtest\_repeat=N相结合，则谷歌测试将选择一个不同的随机种子，并在每个迭代中重新洗牌测试。

*可用性:Linux、Windows、Mac;*因为v1.4.0。

## 控制测试输出

本节教如何调整报告测试结果的方式。

### 彩色的终端输出

谷歌测试可以在其终端输出中使用颜色，以便更容易地发现测试之间的分离，以及测试是否通过。

您可以设置GTEST\_COLOR环境变量或将——GTEST\_COLOR命令行标志设置为yes、no或auto(默认值)来启用颜色、禁用颜色，或者让谷歌测试来决定。当值为auto时，当且仅当输出到终端且(在非windows平台上)术语环境变量设置为xterm或xterm-color时，谷歌测试才会使用颜色。

*可用性:Linux、Windows、Mac。*

### 抑制运行时间

默认情况下，谷歌测试打印运行每个测试所需的时间。为了避免这种情况，使用——gtest\_print\_time=0命令行标志运行测试程序。将GTEST\_PRINT\_TIME环境变量设置为0具有相同的效果。

*可用性:Linux、Windows、Mac(在谷歌Test 1.3.0及以下版本中，默认行为是不打印运行时间)。*

### 生成XML报告

除了正常的文本输出之外，谷歌测试还可以向文件发出详细的XML报告。报告包含每个测试的持续时间，因此可以帮助您识别慢速测试。

要生成XML报告，将GTEST\_OUTPUT环境变量或——GTEST\_OUTPUT标志设置为字符串“XML:\_path\_to\_output\_file\_”，它将在给定位置创建文件。您还可以使用字符串“xml”，在这种情况下，可以在test\_detail中找到输出。当前目录中的xml文件。

如果您指定了一个目录(例如，Linux上的“xml:output/directory/”或Windows上的“xml:output\directory\”)，谷歌测试将在该目录中创建xml文件，以测试可执行文件(例如，foo\_test)命名。用于测试程序foo\_test或foo\_test.exe的xml)。如果文件已经存在(可能是上次运行遗留下来的)，谷歌测试将选择一个不同的名称(例如foo\_test\_1.xml)以避免覆盖它。

报告使用这里描述的格式。它基于junitreport Ant任务，可以由流行的连续构建系统(如Hudson)解析。由于该格式最初是为Java设计的，因此需要对其进行一些解释，以使其适用于谷歌测试，如下所示:

< testsuite name = " AllTests "……> <testsuite name="test\_case\_name"…  
> <testcase name="test\_name"…  
> <失败消息= "…  
" / > <失败消息= "…"  
/ > <失败消息= "……  
" / > < / testcase > < / testsuite > < / testsuite文件>

* 根<testsuites>元素对应于整个测试程序。
* <testsuite>元素对应于谷歌测试用例。
* <testcase>元素对应于谷歌测试测试函数。

例如，下面的程序

TEST(MathTest，加法){…}TEST(MathTest, minus){…  
}TEST(逻辑测试，不矛盾){…  
}

可以生成以下报告:

< ?xml version = " 1.0 " encoding = " utf - 8 " ?><testsuites test ="3" failure ="1" error ="0" time="35" error ="AllTests"  
  
  
  
实际:3 & # x0A;预期:2 " type = " " / > <失败消息= "的价值:添加(1,1)& # x0A;

注意事项:

* 一个<testsuites>或<testsuite>元素的tests属性告诉我们谷歌测试程序或测试用例包含了多少测试函数，而failure属性告诉我们失败了多少。
* time属性以毫秒为单位表示测试、测试用例或整个测试程序的持续时间。
* 每个<failure>元素对应一个失败的谷歌测试断言。
* 一些JUnit概念并不适用于谷歌测试，但是我们必须遵守DTD。因此，您将在报告中看到一些虚构的元素和属性。你完全可以忽略这些部分。

*可用性:Linux、Windows、Mac。*

## 控制如何报告故障

### 将断言失败转换为断点

在调试器下运行测试程序时，如果调试器能够捕获断言失败并自动切换到交互模式，这将非常方便。谷歌测试的break-on-failure模式支持这种行为。

要启用它，请将GTEST\_BREAK\_ON\_FAILURE环境变量设置为0以外的值。或者，您可以使用——gtest\_break\_on\_failure命令行标志。

*可用性:Linux、Windows、Mac。*

### 禁用捕获抛出的测试异常

谷歌测试可以在启用或不启用异常的情况下使用。如果一个测试抛出一个c++异常或(在Windows上)一个结构化异常(SEH)，默认情况下谷歌测试捕获它，报告它为一个测试失败，然后继续下一个测试方法。这最大化了测试运行的覆盖率。此外，在Windows上，未捕获的异常将导致弹出窗口，因此捕获异常允许您自动运行测试。

但是，在调试测试失败时，您可能希望由调试器处理异常，以便在抛出异常时检查调用堆栈。为此，将GTEST\_CATCH\_EXCEPTIONS环境变量设置为0，或者在运行测试时使用——GTEST\_CATCH\_EXCEPTIONS =0标志。

**可用性:Linux、Windows、Mac。**

### 让另一个测试框架驱动

如果您的项目已经使用了另一个测试框架，并且还没有准备好完全切换到谷歌测试，那么您可以通过在现有的测试中使用它的断言来获得谷歌测试的很多好处。只需将main()函数更改为:

# include“gt / gt。测试::GTEST\_FLAG(throw\_on\_failure) = true;  
  
  
  
重要提示:必须初始化谷歌测试。  
::测试::InitGoogleTest(命令行参数个数,argv);  
  
…无论您现有的测试框架需要什么…

这样，除了您的测试框架提供的本机断言外，您还可以使用谷歌测试断言，例如:

void TestFooDoesBar() {Foo Foo;  
  
EXPECT\_LE (foo.Bar (1), 100);//一个谷歌测试断言。  
CPPUNIT\_ASSERT (foo.IsEmpty ());//原生断言。

如果谷歌测试断言失败，它将打印错误消息并抛出异常，主机测试框架将把这视为失败。如果在禁用异常的情况下编译代码，失败的谷歌测试断言将以非零代码退出程序，这也将向测试运行器发出测试失败的信号。

如果不写::testing::GTEST\_FLAG(throw\_on\_failure) = true;在main()中，也可以通过在命令行上指定——gtest\_throw\_on\_failure标志或将gtest\_throw\_on\_failure环境变量设置为非零值来启用此特性。

当使用其他测试框架组织测试时，不支持死测试。

*可用性:Linux、Windows、Mac;*因为v1.3.0。

## 将测试功能分配给多台机器

如果您可以使用不止一台机器来运行测试程序，那么您可能希望并行地运行测试函数，从而更快地获得结果。我们将这种技术称为切分，其中每个机器都称为切分。

谷歌测试与测试分片兼容。为了利用这个特性，您的测试运行程序(不是谷歌测试的一部分)需要做以下工作:

1. 分配一些机器(碎片)来运行测试。
2. 在每个碎片上，将GTEST\_TOTAL\_SHARDS环境变量设置为碎片的总数。对于所有碎片必须是相同的。
3. 在每个切分上，将GTEST\_SHARD\_INDEX环境变量设置为切分的索引。必须为不同的碎片分配不同的索引，这些索引必须在[0,GTEST\_TOTAL\_SHARDS - 1]范围内。
4. 在所有碎片上运行相同的测试程序。当谷歌测试看到上述两个环境变量时，它将选择要运行的测试函数的一个子集。在所有shard中，程序中的每个测试函数都只运行一次。
5. 等待所有碎片完成，然后收集和报告结果。

您的项目可能有一些没有经过谷歌测试的测试，因此不理解这个协议。为了让您的测试运行程序找出哪些测试支持分片，它可以将环境变量GTEST\_SHARD\_STATUS\_FILE设置为不存在的文件路径。如果测试程序支持分片，它将创建这个文件来确认这个事实(此时文件的实际内容并不重要;尽管将来我们可能会在里面贴一些有用的信息。否则它不会创建它。

这里有一个例子可以说明这一点。假设您有一个包含以下5个测试函数的测试程序foo\_test:

测试(V)测试(W),测试(B, X)测试(B, Y)测试(B, Z)

你有3台机器供你使用。要并行运行测试函数，需要在所有机器上将GTEST\_TOTAL\_SHARDS设置为3，并分别在机器上将GTEST\_SHARD\_INDEX设置为0、1和2。然后在每台机器上运行相同的foo\_test。

谷歌测试保留更改工作如何跨碎片分布的权利，但这里有一个可能的场景:

* 机器#0运行A。V和B.X.
* 机器1运行A。W和B.Y.
* 机器2运行b.z。

*可用性:Linux、Windows、Mac;*因为1.3.0版本版本。

# 融合谷歌测试源文件

谷歌测试的实现由~30个文件组成(不包括它自己的测试)。有时您可能希望将它们打包成两个文件(.h和.cc)，以便您可以轻松地将它们复制到新机器上并开始在那里进行攻击。为此，我们提供了一个实验性的Python脚本fuse\_gtest\_files。脚本/目录中的py(从1.3.0版本开始)。假设您的计算机上安装了Python 2.4或更高版本，那么只需转到该目录并运行即可

python fuse\_gtest\_files。py OUTPUT\_DIR

您应该看到使用文件gtest/gtest创建的OUTPUT\_DIR目录。h和gt / gtest-all。cc。这些文件包含使用谷歌测试所需的所有内容。只要将它们复制到您想要的任何地方，就可以编写测试了。您可以使用脚本/test/Makefile文件作为例子，说明如何根据它们编译测试。

# 接下来该怎么办

恭喜你!您现在已经学习了更高级的谷歌测试工具，并准备好处理更复杂的测试任务。如果你想更深入地研究，你可以阅读那些经常被问到的问题。