# 定义一个模拟类

## 模拟一个普通类

鉴于

类Foo{…  
  
虚拟~ Foo ();  
虚拟int GetSize() const = 0;  
虚拟字符串Describe(const char\* name) = 0;  
虚拟字符串description (int类型)= 0;  
虚拟bool进程(Bar elem, int count) = 0;

(注意~Foo()必须是虚的)我们可以将它的mock定义为

# include“gmock / gmock。{MOCK\_CONST\_METHOD0(GetSize, int());  
  
  
  
MOCK\_METHOD1(Describe, string(const char\* name));  
MOCK\_METHOD1(描述字符串(int类型));  
MOCK\_METHOD2(进程，bool(Bar elem, int count));};

创建一个忽略所有无趣调用的“漂亮”模拟对象，或者创建一个将调用视为失败的“严格”模拟对象:

NiceMock < MockFoo > nice\_foo;类型是MockFoo的子类。  
StrictMock < MockFoo > strict\_foo;类型是MockFoo的子类。

## 模拟类模板

嘲笑

模板<typename Elem>类StackInterface {public:…  
  
  
  
虚拟~ StackInterface ();  
虚拟int GetSize() const = 0;  
虚虚推(常量元素& x) = 0;};

(注意~StackInterface()必须是虚拟的)只是附加\_T到MOCK\_\*宏:

模板<typename Elem>类MockStack: public StackInterface<Elem> {public:…  
  
  
  
MOCK\_CONST\_METHOD0\_T (GetSize int ());  
MOCK\_METHOD1\_T(Push, void(const Elem& x));};

## 指定模拟函数的调用约定

如果模拟函数不使用默认的调用约定，可以通过将\_WITH\_CALLTYPE附加到前面两节中描述的任何宏中来指定它，并将调用约定作为宏的第一个参数提供。例如,

MOCK\_METHOD\_1\_WITH\_CALLTYPE(STDMETHODCALLTYPE, Foo, bool(int n));  
MOCK\_CONST\_METHOD2\_WITH\_CALLTYPE(STDMETHODCALLTYPE, Bar, int(double x, double y));

其中STDMETHODCALLTYPE由<objbase定义。在Windows上h >。

# 在测试中使用模拟

典型流程为:

1. 导入需要使用的谷歌模拟名称。所有谷歌模拟名都在测试名称空间中，除非它们是宏或以其他方式标记。
2. 创建模拟对象。
3. 可以选择设置模拟对象的默认操作。
4. 将您的期望设置在模拟对象上(如何调用它们?他们会做什么?
5. 使用模拟对象的练习代码;如果需要，使用谷歌测试断言检查结果。
6. 当一个模拟对象被析构时，谷歌模拟会自动验证它满足了所有的期望。

下面是一个例子:

测试使用::::返回;// #1TEST(BarTest, DoesThis) {MockFoo foo;  
  
  
// #2 ON\_CALL(foo, GetSize()) // #3 .WillByDefault(Return(1));  
  
  
  
/ /……其他默认操作…  
  
// 4 .Times(3) . will(Return("Category 5");  
  
  
/ /……其他期望……  
  
EXPECT\_EQ(“好”,MyProductionFunction (foo));// 5} // 6

# 设置默认的行为

对于任何返回void、bool、数值或指针的函数，谷歌Mock都有一个内置的默认操作。

要自定义返回类型为T的函数的默认操作:

使用::testing::DefaultValue;//设置要返回的默认值。  
  
defaultvalue <T>::Set(value);//设置一个工厂。  
  
将按需调用。T必须是MoveConstructible。  
/ / T市场();DefaultValue < T >:: SetFactory(市场);/ /……  
  
使用模拟…  
//重置默认值。defaultvalue <T>::Clear();

要定制特定方法的默认操作，请使用ON\_CALL():

ON\_CALL(mock\_object, method(matchers)) .With(multi\_argument\_matcher) ?  
  
.WillByDefault(行动);

# 设定的期望

EXPECT\_CALL()设置对模拟方法的期望(如何调用它?它会做什么?):

使用(multi\_argument\_matcher) ?  
  
同学们(基数)?  
.InSequence(sequences) \* .After(expectations) \* .WillOnce(action) \* . will(action) ?  
  
  
  
.RetiresOnSaturation ();吗?

如果忽略Times()，则假设基数为:

* 乘以(1)既没有WillOnce()也没有will();
* \* (n)当有n个WillOnce()s但没有will()时，其中n个>= 1;或
* 乘以(至少(n))，其中n个WillOnce()s和一个will()，其中n个>= 0。

没有EXPECT\_CALL()的方法可以自由调用任意次数，并且每次都将执行默认操作。

# 匹配器

匹配器匹配单个参数。您可以在ON\_CALL()或EXPECT\_CALL()中使用它，或者使用它直接验证一个值:

|  |  |
| --- | --- |
| EXPECT\_THAT(价值,匹配器) | 断言该值与matcher匹配。 |
| ASSERT\_THAT(价值,匹配器) | 与EXPECT\_THAT(值，matcher)相同，只是它会生成一个致命的错误。 |

内置的匹配器(参数是函数参数)分为几个类别:

## 通配符

|  |  |
| --- | --- |
| \_ | 参数可以是正确类型的任何值。 |
| 一种< >()或一种< > () | 参数可以是type类型的任何值。 |

## 通用的比较

|  |  |
| --- | --- |
| Eq(值)或价值 | 参数= =值 |
| 通用电气(值) | 参数> =值 |
| Gt(值) | 参数>值 |
| 勒(值) | 参数< =值 |
| Lt(值) | 参数<值 |
| Ne(值) | 论点! =值 |
| IsNull () | 参数是一个空指针(原始或智能)。 |
| NotNull () | 参数是非空指针(原始或智能)。 |
| 裁判(变量) | 参数是对变量的引用。 |
| TypedEq <类型>(值) | 参数具有类型类型且等于值。当模拟函数重载时，您可能需要使用这个函数而不是Eq(value)。 |

除了Ref()之外，这些匹配器会复制值，以防以后修改或销毁它。如果编译器抱怨该值没有公共复制构造函数，尝试将其封装在ByRef()中，例如Eq(ByRef(non\_copyable\_value))。如果这样做，请确保之后不会更改non\_copyable\_value，否则您的匹配器的含义将被更改。

## 浮点匹配器

|  |  |
| --- | --- |
| DoubleEq (a\_double) | 参数是一个近似等于a\_double的双值，将两个nan视为不相等。 |
| FloatEq (a\_float) | 参数是一个近似等于a\_float的浮点值，将两个NaNs视为不相等。 |
| NanSensitiveDoubleEq (a\_double) | 参数是一个近似等于a\_double的双精度值，它将两个nan视为相等。 |
| NanSensitiveFloatEq (a\_float) | 参数是一个近似等于a\_float的浮点值，它将两个NaNs视为相等。 |

上述匹配器使用基于ulp的比较(与谷歌测试中使用的相同)。它们会根据期望值的绝对值自动选择一个合理的误差范围。DoubleEq()和FloatEq()符合IEEE标准，该标准要求比较两个NaNs是否相等才能返回false。相反，NanSensitive\*版本将两个NaNs视为平等的，这通常是用户想要的。

|  |  |
| --- | --- |
| DoubleNear (a\_double max\_abs\_error) | 参数是一个接近a\_double的双值(绝对错误<= max\_abs\_error)，将两个NaNs视为不相等。 |
| FloatNear (a\_float max\_abs\_error) | 参数是一个接近a\_float的浮点值(绝对错误<= max\_abs\_error)，将两个NaNs视为不相等。 |
| NanSensitiveDoubleNear (a\_double max\_abs\_error) | 参数是一个接近于a\_double的双值(绝对错误<= max\_abs\_error)，将两个nan视为相等。 |
| NanSensitiveFloatNear (a\_float max\_abs\_error) | 参数是一个接近a\_float的浮点值(绝对错误<= max\_abs\_error)，将两个NaNs视为相等。 |

## 字符串匹配器

参数可以是C字符串，也可以是c++字符串对象:

|  |  |
| --- | --- |
| ContainsRegex(字符串) | 参数匹配给定的正则表达式。 |
| EndsWith(后缀) | 参数以字符串后缀结束。 |
| HasSubstr(字符串) | 参数将字符串作为子字符串包含。 |
| MatchesRegex(字符串) | 参数匹配给定的正则表达式，匹配从第一个字符开始，以最后一个字符结束。 |
| StartsWith(前缀) | 参数以字符串前缀开始。 |
| StrCaseEq(字符串) | 参数等于字符串，忽略大小写。 |
| StrCaseNe(字符串) | 参数不等于字符串，忽略大小写。 |
| StrEq(字符串) | 参数等于字符串。 |
| StrNe(字符串) | 参数不等于字符串。 |

ContainsRegex()和MatchesRegex()使用这里定义的正则表达式语法。StrCaseEq()、StrCaseNe()、StrEq()和StrNe()也适用于宽字符串。

## 容器匹配器

大多数stl风格的容器都支持==，所以您可以使用Eq(expected\_container)或直接使用expected\_container来精确匹配容器。如果你想把这些元素写在一行里，更灵活地匹配它们，或者得到更有用的信息，你可以使用:

|  |  |
| --- | --- |
| ContainerEq(容器) | 与Eq(container)相同，不同的是失败消息还包括哪些元素在一个容器中，但不包括另一个容器。 |
| 包含(e) | 参数包含匹配e的元素，e可以是值，也可以是匹配器。 |
| 每个(e) | 参数是每个元素都匹配e的容器，e可以是值，也可以是匹配器。 |
| 元素(e0、e1,……,) | 参数有n + 1个元素，其中第i个元素与ei匹配，ei可以是一个值，也可以是一个匹配器。允许0到10个参数。 |
| ElementsAreArray({e0, e1，…、ElementsAreArray(array)或ElementsAreArray(array, count) | 与ElementsAre()相同，只是预期的元素值/匹配程序来自初始化器列表、stl风格的容器或c风格的数组。 |
| IsEmpty () | 参数是一个空容器(container.empty())。 |
| 点态(m,容器) | 参数包含与容器中相同数量的元素，对于所有i(参数中的第i个元素，容器中的第i个元素)匹配m, m是二元组上的一个匹配器。例如，Pointwise(Le()， upper\_bounds)验证参数中的每个元素都不超过upper\_bounds中相应的元素。详见下文。 |
| SizeIs (m) | 参数是一个容器，它的大小与m匹配。例如，SizeIs(2)或SizeIs(Lt(2))。 |
| UnorderedElementsAre (e0、e1,……,) | 参数有n + 1个元素，在某种排列下，每个元素匹配一个ei(表示不同的i)，它可以是一个值或一个匹配器。允许0到10个参数。 |
| UnorderedElementsAreArray({e0, e1，…、UnorderedElementsAreArray(array)或UnorderedElementsAreArray(array, count) | 与UnorderedElementsAre()相同，只是预期的元素值/匹配程序来自初始化器列表、stl风格的容器或c风格的数组。 |
| WhenSorted (m) | 当使用<操作符对参数排序时，它匹配容器匹配器m。例如，当排序(UnorderedElementsAre(1,2,3))验证参数包含元素1,2,3时，忽略顺序。 |
| WhenSortedBy(比较器,m) | 与when ordered (m)相同，不同之处是使用给定的比较器而不是<来对参数排序。例如，当sortedby (std::greater<int>()， ElementsAre(3,2,1))。 |

注:

* 这些匹配器也可以匹配:
  1. 一个通过引用传递的原生数组(例如在Foo(const int (&a)[5])中)，和
  2. 以指针和计数形式传递的数组(例如在Bar中(const T\* buffer, int len)——参见多参数匹配器)。
* 被匹配的数组可以是多维的(即它的元素可以是数组)。
* 点态中的m (m，…)应该是::testing::tuple<T, U>的匹配器，其中T和U分别是实际容器和预期容器的元素类型。例如，要比较两个Foo容器，其中Foo不支持操作符==但有一个Equals()方法，可以这样写:

测试使用::::,匹配器(FooEq, " "){回归得到< 0 > (arg) .Equals (< 1 > (arg));}…  
  
  
  
  
EXPECT\_THAT (actual\_foos点态(FooEq (), expected\_foos));

## 成员匹配器

|  |  |
| --- | --- |
| 字段(类::字段,m) | 论点。字段(当实参是纯指针时为>字段)匹配匹配器m，其中实参是类类型的对象。 |
| 键(e) | 论点。首先匹配e，它可以是一个值，也可以是一个匹配器。例如Contains(Key(Le(5)))可以验证一个map包含一个Key <= 5。 |
| 对(m1, m2) | 参数是一个std::对，它的第一个字段匹配m1，第二个字段匹配m2。 |
| 属性(类::财产,m) | property()(或argument->property()当实参是一个普通指针时)匹配matcher m，其中实参是class类型的对象。 |

## 匹配函数或函子的结果

|ResultOf(f, m)|f(实参)匹配matcher m，其中f是一个函数或函子。| |:- - - - - - - - - - - - - - - - - - |:- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - |

## 指针匹配器

|  |  |
| --- | --- |
| 指针数据(m) | 参数(智能指针或原始指针)指向与匹配器m匹配的值。 |
| WhenDynamicCastTo < T > (m) | 当参数通过dynamic\_cast<T>()传递时，它匹配匹配器m。 |

## Multiargument匹配器

从技术上讲，所有的匹配器匹配一个值。“多参数”匹配器只是一个匹配元组的匹配器。下列匹配器可用于匹配元组(x, y):

|  |  |
| --- | --- |
| Eq () | x = = y |
| 通用电气() | x > = y |
| Gt () | x > y |
| 勒() | x < = y |
| Lt () | x < y |
| 不() | x ! = y |

您可以使用以下选择器来选择参数的子集(或重新排序)来参与匹配:

|  |  |
| --- | --- |
| AllArgs (m) | 相当于在. with (AllArgs(m))中用作语法糖。 |
| 参数< N1、N2…,Nk > (m) | k选择的元组(使用基于0的索引)参数匹配m，例如Args< 1,2 >(Eq())。 |

## 复合匹配器

你可以从一个或多个其他匹配器:

|  |  |
| --- | --- |
| 所有(m1, m2,…、锰) | 参数匹配所有的matchers m1到mn。 |
| 任何(m1, m2,…、锰) | 参数至少匹配一个匹配器m1到mn。 |
| (米) | 参数不匹配匹配器m。 |

## 适配器的匹配器

|  |  |
| --- | --- |
| MatcherCast < T > (m) | 将matcher m类型转换为matcher <T>。 |
| SafeMatcherCast < T > (m) | [将匹配器m安全地强制转换为类型匹配器<T>。](CookBook.md#casting-matchers) |
| 真正的(谓词) | 谓词(参数)返回c++认为为真的东西，其中谓词是一个函数或函子。 |

## 匹配器的谓词

|  |  |
| --- | --- |
| 匹配(m)(值) | 如果值与m匹配，则计算为true。您可以单独使用matches (m)作为一元仿函子。 |
| ExplainMatchResult (m值,result\_listener) | 如果值匹配m，则计算为true，并向result\_listener解释结果。 |
| 价值(价值,米) | 如果值与m匹配，则计算为true。 |

## 定义匹配器

|  |  |
| --- | --- |
| MATCHER(IsEven， "") {return (arg % 2) == 0;} | 定义matcher IsEven()来匹配偶数。 |
| \*result\_listener << "其中余数为" << (arg % n);return (arg % n) == 0;} | 定义一个macher IsDivisibleBy(n)来匹配一个能被n整除的数。 |
| MATCHER\_P2(IsBetween, a, b, std::string(否?"isn't": "is") + " between " + PrintToString(a) + " and " + PrintToString(b)) {return a <= arg && arg <= b;} | 定义一个matcher IsBetween(a, b)来匹配[a, b]范围内的值。 |

**注:**

1. 不能在函数或类中使用MATCHER\*宏。
2. matcher主体必须是纯功能性的(即它不能有任何副作用，并且结果只能依赖于被匹配的值和matcher参数)。
3. 可以使用PrintToString(x)将任何类型的值x转换为字符串。

## 匹配器作为测试断言

|  |  |
| --- | --- |
| ASSERT\_THAT(表情,米) | 如果表达式的值与匹配程序m不匹配，则生成致命错误。 |
| EXPECT\_THAT(表情,米) | 如果表达式的值与匹配程序m不匹配，则生成非致命错误。 |

# 行动

**操作指定调用模拟函数时应该执行的操作。**

## 返回一个值

|  |  |
| --- | --- |
| 返回() | 从void模拟函数返回。 |
| 返回(值) | 返回值。如果值的类型与模拟函数的返回类型不同，则在设置期望时将值转换为后一种类型，而不是在执行操作时。 |
| ReturnArg < N > () | 返回第n个(基于0的)参数。 |
| ReturnNew < T > (a1,…正义与发展党) | 返回新T(a1，…ak);每次都创建一个不同的对象。 |
| ReturnNull () | 返回一个空指针。 |
| ReturnPointee (ptr) | 返回ptr指向的值。 |
| ReturnRef(变量) | 返回对变量的引用。 |
| ReturnRefOfCopy(值) | 返回对值的副本的引用;只要行动，副本就会一直存在。 |

## 副作用

|  |  |
| --- | --- |
| 分配(变量值) | 给变量赋值。 |
| DeleteArg < N > () | 删除第n个(基于0的)参数，它必须是一个指针。 |
| SaveArg < N >(指针) | 将第n个(基于0的)参数保存为\*指针。 |
| SaveArgPointee < N >(指针) | 将第n个(基于0的)参数指向的值保存为\*pointer。 |
| SetArgReferee < N >(值) | 为第n个(基于0的)参数引用的变量赋值。 |
| SetArgPointee < N >(值) | 为第n个(基于0的)参数所指向的变量赋值。 |
| SetArgumentPointee < N >(值) | 一样SetArgPointee < N >(值)。弃用。将在v1.7.0中删除。 |
| SetArrayArgument < N >(第一,最后) | 将源范围[first, last]中的元素复制到第n个(基于0的)参数所指向的数组中，该参数可以是指针，也可以是迭代器。该操作不获取源范围内元素的所有权。 |
| SetErrnoAndReturn(错误,价值) | 将errno设置为error并返回值。 |
| 把(异常) | 抛出给定的异常，该异常可以是任何可复制的值。自v1.1.0可用。 |

## 将函数或函子用作操作

|  |  |
| --- | --- |
| 调用(f) | 使用传递给模拟函数的参数调用f，其中f可以是全局/静态函数或仿函数。 |
| 调用(object\_pointer,类::方法) | 使用传递给模拟函数的参数调用对象上的{方法。 |
| InvokeWithoutArgs (f) | 调用f，它可以是一个全局/静态函数，也可以是一个仿函数。f不接受任何参数。 |
| InvokeWithoutArgs (object\_pointer,类::方法) | 调用对象上的方法，该方法不接受参数。 |
| InvokeArgument < N > (\_\_arg1、最长,……argk) | 使用k个参数调用模拟函数的第n个(基于0)参数，该参数必须是函数或函数。 |

被调用函数的返回值用作操作的返回值。

在定义要与Invoke\*()一起使用的函数或函子时，可以将任何未使用的参数声明为未使用:

返回sqrt(x\*x + y\*y);}……  
  
().WillOnce(Invoke(Distance));

在InvokeArgument<N>(…)中，如果一个参数需要通过引用传递，那么将它包装在ByRef()中。例如,

InvokeArgument < 2 >(时5,字符串(“嗨”),ByRef (foo))

调用模拟函数的#2参数，通过值传递给它5和字符串(“Hi”)，通过引用传递给它foo。

## 默认的动作

|DoDefault()|执行默认操作(由ON\_CALL()或内置操作指定)。| |:- - - - - - - - - - - - |:- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - |

**注意:由于技术原因，不能在复合操作中使用DoDefault()—尝试这样做将导致运行时错误。**

## 复合动作

|  |  |
| --- | --- |
| DoAll (a1, a2,…一个) | 对an执行所有操作a1，并在每次调用中返回an的结果。前n - 1个子动作必须返回void。 |
| IgnoreResult (a) | 执行动作a并忽略其结果。一个不能返回空。 |
| WithArg < N > (a) | 将模拟函数的第n个(基于0的)参数传递给动作a并执行它。 |
| WithArgs < N1、N2…Nk >(一个) | 将模拟函数的选定(基于0)参数传递给动作a并执行它。 |
| WithoutArgs (a) | 执行动作a，不带任何参数。 |

## 定义的行为

|  |  |
| --- | --- |
| 操作(Sum){返回arg0 + arg1;} | 定义一个action Sum()来返回模拟函数的参数#0和#1的和。 |
| ACTION\_P(+， n){返回arg0 + n;} | 定义一个action Plus(n)来返回模拟函数的参数#0和n的和。 |
| ACTION\_Pk (Foo, p1,…， pk){语句;} | 定义参数化操作Foo(p1，…， pk)来执行给定的语句。 |

ACTION\*宏不能在函数或类中使用。

# 基数

这些在Times()中用于指定调用模拟函数的次数:

|  |  |
| --- | --- |
| AnyNumber () | 这个函数可以被调用任意次数。 |
| 至少(n) | 这个调用被期望至少n次。 |
| AtMost (n) | 这个调用最多被期望n次。 |
| (m, n)之间 | 该调用预期在m和n(包括)次之间。 |
| (n)或n | 这个调用被期望n次。特别是，当n = 0时，调用不会发生。 |

# 期待订单

默认情况下，期望可以以任何顺序匹配。如果必须按照给定的顺序匹配部分或所有期望，则有两种方法指定它。它们可以单独使用，也可以一起使用。

## 在条款

测试使用::::期望;…  
  
(foo, Bar()) .After(init\_x, init\_y);

说Bar()只能在调用InitX()和InitY()之后才能调用。

如果你不知道一个期望在你写它的时候有多少先决条件，你可以使用一个ExpectationSet来收集它们:

测试使用::::ExpectationSet;…  
  
for (int i = 0;  
我< element\_count;{all\_inits += EXPECT\_CALL(foo, InitElement(i));}

说Bar()只能在所有的元素被初始化之后才能被调用(但是我们并不关心哪些元素在其他元素之前被初始化)。

在. after()中使用后修改ExpectationSet不会影响. after()的含义。

## 序列

当您有一个很长的序列期望链时，使用序列来指定顺序会更容易一些，这样就不需要为链中的每个期望指定不同的名称。相同序列中的所有预期<br>调用必须按照指定的顺序发生。

使用::test::Sequence;Sequence s1, s2;…  
  
  
(1) . EXPECT\_CALL(foo, Reset()) .InSequence(s1, s2) .WillOnce(Return(true));

说明必须在GetSize()和Describe()之前调用Reset()，后两者可以以任何顺序出现。

方便地把许多期望按顺序排列:

使用::testing::InSequence;  
  
  
  
EXPECT\_CALL (…)…;  
EXPECT\_CALL (…)…;  
…  
EXPECT\_CALL (…)…;}

表示虚拟范围内所有预期的调用必须严格按顺序发生。假名字是不相关的。)

# 验证和重置模拟

当模拟对象被销毁时，谷歌Mock将验证它的期望，或者您可以更早地这样做:

测试使用::::模拟;…  
  
//验证并移除mock\_obj上的期望;//返回true iff success . mock:: verifyandclearexpecations (&mock\_obj);…  
  
  
  
//也删除ON\_CALL()设置的默认操作;//返回true iff success . mock::VerifyAndClear(&mock\_obj);

你也可以告诉谷歌Mock一个模拟对象可能会被泄露，不需要验证:

模拟:AllowLeak (&mock\_obj);

# 模拟类

谷歌Mock定义了一个方便的模拟类模板

类MockFunction < R (A1,…MOCK\_METHODn(调用，R(A1，…  
  
一个);};

请参阅此配方中的一个应用程序。

# 旗帜

|  |  |
| --- | --- |
| ——gmock\_catch\_leaked\_mocks = 0 | 不要将泄漏的模拟对象报告为失败。 |
| ——gmock\_verbose =水平 | 设置谷歌模拟消息的默认详细级别(信息、警告或错误)。 |