Koala++'s blog

计算广告学 RTB

日志

Google Mock进阶篇 [5] (Google Mock C… Google Mock进阶篇 [7] (Google Mock C…

Google Mock进阶篇 [6] (Google Mock Cookbook译文)

2012-05-02 21:40:34 | 分类： C++ | 标签： google mock 测试 订阅 | 字号 | 举报

我的照片书 | 下载LOFTER

Using Actions

关于我

quweiprotoss

加博友 关注他

如果一个Mock函数的返回类型是引用，你需要用ReturnRef()而不是Return()来返using ::testing::ReturnRef;

```
class MockFoo : public Foo {
 public:
  MOCK_METHOD0(GetBar, Bar&());
};
...

MockFoo foo;
Bar bar;
EXPECT_CALL(foo, GetBar())
  .WillOnce(ReturnRef(bar));
```

Return Live Values from Mock Methods

Return(x)这个动作在*创建时*就会保存一个x的拷贝，在它执行时总是返回相同的值。但有时你可能不想每次返回x的拷贝。

如果Mock函数的返回类型是引用，你可以用ReturnRef(x)来每次返回不同的值。但是Google Mock不允许在Mock函数返回值不是引用的情况下用ReturnRef()返回，这样做的后果通常是提示一个错误，所以，你应该怎么做呢？

你可以尝试ByRef()：

```
using testing::Return;

class MockFoo : public Foo {
public:
MOCK_METHOD0(GetValue, int());
};
...
int x = 0;
MockFoo foo;
EXPECT_CALL(foo, GetValue())
   .WillRepeatedly(Return(ByRef(x)));
x = 42;
EXPECT_EQ(42, foo.GetValue());
```

不幸的是，上面的代码不能正常工作，它会提示以下错误：

```
Value of: foo.GetValue()
  Actual: 0
Expected: 42
```

不能正常工作的原因是在Return(value)这个动作*创建时*将x转换成Mock函数的返回类型，而不是它*执行时*再进行转换(这个特性是为保证当值是代理对象引用一些临时对象时的安全性)。结果是当期望设置时ByRef(x)被转换成一个int值(而不是一个const int&)，且Return(ByRef(x)会返回0。

ReturnPointee(pointer)是用来解决定这个问题的。它在动作执行时返回指针指向的值：

```
...
int x = 0;
MockFoo foo;
EXPECT_CALL(foo, GetValue())
  .WillRepeatedly(ReturnPointee(&x));  // Note the & here.
x = 42;
EXPECT_EQ(42, foo.GetValue());  // This will succeed now.
```

Combining Actions

你想当一个函数被调用时做更多的事吗？这个需求是合理的。DoAll()允许你每次执行一系列动作。只有最后一个动作的返回值会被使用。

```
using ::testing::DoAll;

class MockFoo : public Foo {
 public:
  MOCK_METHOD1(Bar, bool(int n));
};
...

EXPECT_CALL(foo, Bar(_))
  .WillOnce(DoAll(action_1,
                  action_2,
                  ...
                  action_n));
```

有时一个函数的作用不是通过返回值来体现，而是通过副作用。比如，你可以改变一些全局状态或是修改一个输入参数的值。要Mock副作用，通常你可以通过实现::testing::AtionInterface定义你自己的动作。

如果你要做的仅仅是改变一个输入参数，内置的SetArgPointee()动作是很方便的：

```
using ::testing::SetArgPointee;

class MockMutator : public Mutator {
 public:
  MOCK_METHOD2(Mutate, void(bool mutate, int* value));
  ...
};
...

MockMutator mutator;
EXPECT_CALL(mutator, Mutate(true, _))
  .WillOnce(SetArgPointee<1>(5));
```

在这个例子中，当mutator.Mutate()被调用时，我们将赋给由第二个参数指针指向的值为5。

SetArgPointee()将传入的值进行了一次拷贝，所以你不需要保证这个值的生命周期。但这也意味着这个对象必须有一个拷贝构造函数和赋值操作符。

到DoAll()中：

```
using ::testing::_;
using ::testing::Return;
using ::testing::SetArgPointee;

class MockMutator : public Mutator {
public:
...
MOCK_METHOD1(MutateInt, bool(int* value));
};
...

MockMutator mutator;
EXPECT_CALL(mutator, MutateInt(_))
    .WillOnce(DoAll(SetArgPointee<0>(5),
                    Return(true)));
```

如果输出参数是一个数组，用SetArrayArgument<N>[first, last)动作。它将源范围[first, last)中的元素拷贝到一个新的以0开始的新数组中：

```
using ::testing::NotNull;
using ::testing::SetArrayArgument;

class MockArrayMutator : public ArrayMutator {
 public:
  MOCK_METHOD2(Mutate, void(int* values, int num_values));
```

```
};
...

MockArrayMutator mutator;
int values[5] = { 1, 2, 3, 4, 5 };
EXPECT_CALL(mutator, Mutate(NotNull(), 5))
  .WillOnce(SetArrayArgument<0>(values, values + 5));
```

当参数是一个输出迭代器时也是可以工作的：

```
using ::testing::_;
using ::testing::SeArrayArgument;

class MockRolodex : public Rolodex {
public:
MOCK_METHOD1(GetNames, void(std::back_insert_iterator<vector<string> >));
...
};
...

MockRolodex rolodex;
vector<string> names;
names.push_back("George");
names.push_back("John");
names.push_back("Thomas");
EXPECT_CALL(rolodex, GetNames(_))
```

Changing a Mock Object's Behavior Based on the State

如果你期望一个调用改变mock对象的行为，你可以用::testing::InSequence来指定在这个调用之前和之后的对象行为：

```
using ::testing::InSequence;
using ::testing::Return;

...
{
InSequence seq;
EXPECT_CALL(my_mock, IsDirty())
  .WillRepeatedly(Return(true));
EXPECT_CALL(my_mock, Flush());
EXPECT_CALL(my_mock, IsDirty())
  .WillRepeatedly(Return(false));
}
my_mock.FlushIfDirty();
```

这可以让my_mock.IsDirty()在my_mock.Flush()调用之前返回true，而在之后返回false。

如果要改变的对象动作更复杂，你可以保存保存这些效果到一个变量中，并使一个Mock函数从这个变量中得到它的返回值：

```
using ::testing::SaveArg;
using ::testing::Return;

ACTION_P(ReturnPointee, p) { return *p; }
...
int previous_value = 0;
EXPECT_CALL(my_mock, GetPrevValue())
   .WillRepeatedly(ReturnPointee(&previous_value));
EXPECT_CALL(my_mock, UpdateValue(_))
   .WillRepeatedly(SaveArg<0>(&previous_value));
my_mock.DoSomethingToUpdateValue();
```

这样，m_mock.GetPrevValue()总是会返回上一次UpdateValue调用的参数值。

Setting the Default Value for a Return Type

如果一个Mock函数返回类型是一个内置的C++类型或是指针，当它调用时默认会返回0。如果默认值不适合你，你只需要指定一个动作。

有时，你也许想改变默认值，或者你想指定一个Google Mock不知道的类型的默认值。你可以用::testing::DefaultValue类模板：

```
class MockFoo : public Foo {
 public:
  MOCK_METHOD0(CalculateBar, Bar());
};
```

```
Bar default_bar;
// Sets the default return value for type Bar.
DefaultValue<Bar>::Set(default_bar);

MockFoo foo;

// We don't need to specify an action here, as the default
// return value works for us.
EXPECT_CALL(foo, CalculateBar());

foo.CalculateBar();  // This should return default_bar.

// Unsets the default return value.
DefaultValue<Bar>::Clear();
```

请注意改变一个类型的默认值会让你的测试难于理解。我们建议你谨慎地使用这个特性。比如，你最好确保你在使用这个特性代码之前之后要加上Set()和Clear()调用。

Setting the Default Actions for a Mock Method

如果你掌握了如何改变一个类型的默认值。但是也许这对于你也许是不够的：也许你有两个Mock函数，它们有相同的返回类型，并且你想它们有不同的行为。ON_CALL()宏允许你在函数级别自定义你的Mock函数行为：

```
using ::testing::_;
```

```
using ::testing::Gt;
using ::testing::Return;
...
ON_CALL(foo, Sign(_))
    .WillByDefault(Return(-1));
ON_CALL(foo, Sign(0))
    .WillByDefault(Return(0));
ON_CALL(foo, Sign(Gt(0)))
    .WillByDefault(Return(1));

EXPECT_CALL(foo, Sign(_))
    .Times(AnyNumber());

foo.Sign(5);   // This should return 1.
foo.Sign(-9);  // This should return -1.
foo.Sign(0);   // This should return 0.
```

正如你所猜测的，当有多个ON_CALL()语句时，新的语句(即后写的语句)会优先匹配。换言之，**最后一个**匹配参数的Mock函数会被调用。这种匹配顺序允许你开始设置比较宽松的行为，然后再指定这个Mock函数更具体的行为。

Using Functions / Methods / Functors as Actions

如果内置动作不适合你，你可以轻松地用一个已有的函数，方法，仿函数作为一个动作：

```
using ::testing::Invoke;

class MockFoo : public Foo {
 public:
  MOCK_METHOD2(Sum, int(int x, int y));
  MOCK_METHOD1(ComplexJob, bool(int x));
};

int CalculateSum(int x, int y) { return x + y; }

class Helper {
 public:
  bool ComplexJob(int x);
};
...

  MockFoo foo;
  Helper helper;
  EXPECT_CALL(foo, Sum(_, _))
      .WillOnce(Invoke(CalculateSum));
  EXPECT_CALL(foo, ComplexJob(_))
      .WillOnce(Invoke(&helper, &Helper::ComplexJob));

  foo.Sum(5, 6);       // Invokes CalculateSum(5, 6).
  foo.ComplexJob(10);  // Invokes helper.ComplexJob(10);
```

者的参数必须可以隐式转换成Mock函数中相应的参数，前者的返回值可以隐式转换成Mock函数的返回类型。所以，你可以调用一个与Mock函数定义不完全一致的函数，只要这样做是安全的，精彩吧，huh？

Google Mock进阶篇 [5] (Google Mock C… Google Mock进阶篇 [7] (Google Mock C…

评论

登录后你可以发表评论，请先登录。登录>>