Google C++ Mocking Framework for Dummies

Google Mock 启蒙篇

Version: 0.07

作者： *adrian alexander*

译者： *Koala++ / 屈伟*

What Is Google C++ Mocking Framework

当你写一个原型或是测试的时候，直接去依赖真实的对象通常是不可行的或是不明智的。Mock 对象实现与真实对象有着相同的接口，但你可以去指定 Mock 对象在运行时它做什么（比如，调用哪个函数，以什么顺序，调用多少次，使用什么参数，返回内容是什么，等等）。

注意：*Fake 对象*和 *Mock 对象*两个术语很容易混淆。在测试驱动开发（TDD）语境下，Fake 对象和 Mock 对象是区别很大的两个概念。

- Fakes 是有具体实现的，但通常是一些走了捷径的实现，所以它不能真正的用于发布产品中。比如一个在内存中的文件系统就是一个 Fake 的例子。
- Mocks 是一些有*期望（expections）*预先编程的对象，期望形成了有着期望结果的调用的一个特化。

上面所解释也许太过抽象，但别担心，最重要的就是记住：Mock 可以使你检查它与使用它的代码之间的*交互*。其实，当你开始使用 Mocks 之后，Fakes 和 Mocks 之间的区别会马上清晰起来。

Google C++ Mocking Framework（或简称 **Google Mock**）是一个用于创建 Mock 类使用这些类的库（有时我们也将它称为框架，让它听起来更酷一些）。它和 Java 世界中的 jMock 和 EasyMock 功能相同。

使用 Google Mock 有下面三个基本步骤：

1. 使用简单的宏来描述你想 Mock 的接口，这些宏会自动扩展成你的 Mock 类的实现。
2. 创建一些 Mock 对象，并用一种直观的语法去指定 Mock 对象的期望和行为。
3. 用这些 Mock 对象测试代码。Google Mock 会捕获那些违反期望的冲突。

Why Google Mock?

虽然 Mock 对象可以帮助你在测试中去除不必要的依赖，并使测试更快更可靠，但是在

C++中用 Mocks 却比较难。

- 它需要你自己去实现 Mocks。这项工作通常无聊并且易错。无怪大家都尽可能不去做这种件事。
- 这些自己实现的 Mocks 质量有一点……，嗯，不可预测。你可能见过一些写的很精巧的 Mocks，但你也可能看到一些匆忙间 hack 出来的 Mocks，这些 Mocks 中充斥着各种怪异的限制。
- 你在使用一个 Mock 所获得的经验无法在下一个 Mock 中使用。

相反，Java 和 Python 程序员有一些很好的 Mock 框架，它们可以自动创建 Mocks。结果，在 Java 和 Python 世界 Mocking 是被证明是一种高效的技术，并被广泛地使用。

Google Mock 就是为帮助 C++程序员解决 Mock 问题而生的。它受 jMock 和 EasyMock 的启发，但在设计时思虑了 C++的特性。如果下面的任一问题困扰着你，那么 Google Mock 将成为你的朋友。

- 你被一个优化了一部分的设计困住了，你希望可以做一些原型设计，以免时间不够了。但原型设计在 C++中绝对不可能称之为快。
- 你的测试很慢可能它们依赖很多库或是使用很多资源(比如，数据库)。
- 你的测试很脆弱，因为它所依赖的资源不可靠。
- 你想测试你的代码处理失败的情况，但是很难产生一个失败。
- 你需要保证你的模块与别的模块以正确的方式交互，但很难看到交互的过程，你只能看到最终运行的结果，这无论如何都是尴尬的。
- 你想要“Mock out”你的依赖，但是这些依赖还没有 Mock 实现，坦白地讲，就算是有，你看到这些手写的 Mocks 也会头痛。

我们推荐你在以下面的方式使用 Google Mock:

- 作为一个*设计*工具，因为它可以让你更早更频繁地试验你的接口设计。更多的迭代会产生更好的设计。
- 作为一个*测试*工具，它可以解除外围的依赖，并可以查看你的模板与其它模块的交互。

Get Started

使用 Google Mock 很容易！在你的 C++源文件中，只需要写上#include “gtest/gtest.h”和“gmock/gmock.h”，你就可以开始你的 Goole Mock 之旅了。

A Case for Mock Turtles

让我们看一个例子。假设你在开发一个图形程序，它依赖一个类似 Logo(译注：初一我学的第一门计算机语言，每次我听到它名字都会激动万分，虽然它的命令我几乎忘光了) 的 API 来绘图，你怎么去测试你的程序是正确的呢？嗯，你可以运行它，然后比较你屏幕上的结果和目标屏幕截图，但是必需要承认的是：这种测试很麻烦，并且健壮性不足(如果你升级了你的显卡，这个显卡有更好的抗锯齿能力，那你需要把你用的图形文件都换了)。如果你的测试都是这样的，那你会很痛苦的。幸运的是，你知道依赖注入并且知道该如何去做：不要让你的程序直接去调用绘图 API，而应该将 API 封装成一个接口(Turtle，译注：Logo 语言中的图标像是一个海龟，在 Doc 时代这完全是骗小朋友的，它就是一个没有尾巴的箭头)，并针对接口编程。

```
class Turtle {
  ...
  virtual ~Turtle() {}
  virtual void PenUp() = 0;
  virtual void PenDown() = 0;
  virtual void Forward(int distance) = 0;
  virtual void Turn(int degrees) = 0;
  virtual void GoTo(int x, int y) = 0;
  virtual int GetX() const = 0;
  virtual int GetY() const = 0;
};
```

注意：Turtle 类的析构函数**必须**是虚函数，因为在随后的介绍中要继承这个类。

你可以通过 PenUp()和 PenDown()来控制光标的移动是否会留下痕迹，并用 Forward()，Turn()，和 Goto()来控制它的移动，GetX()和 GetY()会告诉你当前光标的位置。

你的程序通常会使用这个接口的真实实现。但在测试中，你可以使用一个 Mock 实现来代替。这可以让你更早地检查你的程序调用哪些绘图 API，使用什么参数，以什么顺序调用。这样的测试更健壮(这样的测试不会因为你的新显卡在反锯齿性上表现不同而失败)，并且这种代码更容易去理解和维护(测试的目标是用代码表示，而不是用一些二进制图形去表示)，而且会运行的*非常非常*快。

Writing the Mock Class

如果你需要的 Mocks 已经有好心人实现了，那你很走运。但是如果你发现需要自己要去实现 Mock 类，也别紧张，Google Mock 已经将这个任务变成了一个有趣的游戏(嗯，算是吧)。

How to Define It

这里以 Turtle 接口为例子，下面是你需要去做的几个步骤：

1. 继承 Turtle 类得到 MockTurtle 类。
2. 从 Turtle 类中选一个虚函数(也可用模板 Mock 非虚函数，但那涉及的知识就多了一些)，数一下这个函数有几个参数。
3. 在 MockTurtle 的 public:部分，写上 MOCK_METHODn(); (如果你要 Mock 一个 const 函数，就写 MOCK_CONST_METHODn)，其中 n 是函数中的参数个数，如果你真的连数数都能数错，那编译器会坦白地告诉你这个丢脸的事实。
4. 这一步终于是能看到意义的一步了：你把函数名作为宏的第一个参数，然后将函数定义中除函数名以外的部分作为宏的第二个参数。
5. 重复上述步骤，直到你想 Mock 的虚函数都 Mock 了。

在完成上述步骤后，你得到的是类似下面的代码：

```
#include "gmock/gmock.h"  // Brings in Google Mock.
class MockTurtle : public Turtle {
 public:
  ...
  MOCK_METHOD0(PenUp, void());
  MOCK_METHOD0(PenDown, void());
  MOCK_METHOD1(Forward, void(int distance));
  MOCK_METHOD1(Turn, void(int degrees));
  MOCK_METHOD2(GoTo, void(int x, int y));
  MOCK_CONST_METHOD0(GetX, int());
  MOCK_CONST_METHOD0(GetY, int());
};
```

你不需要再在别的地方去定义这些 Mock 函数了，MOCK_METHOD*宏会帮你产生这些函数定义。这很简单！一旦掌握了它的诀窍，你可以产生大量的 Mock 类，可能快到连源代码管理工具都处理不过来。

小提示：如果连定义对你来说工作量都太大，你可以在 scripts/generator 目录下找到一个 gmock_gen.py 工具，这个命令行工具需要安装 Python 2.4。你将 C++文件名和抽象类名作为参数传入这个工具，它会打印 Mock 类的定义给你。但是因为 C++的复杂性，这个脚本还是可能出错，但当它不出错的时候，还是很方便的。更多的细节在用户文档中。

Where to Put It

当你定义一个 Mock 类 ,你需要决定把它的定义放到哪。一些人把它放到一个*_test.cc

文件中。当这个接口(就叫 Foo 吧)是由同一个人或是同一团队维护时，这没什么问题。但如果不是，当 Foo 的维护者修改了它，你的测试就会编译不通过(你总不能指望 Foo 的维护者去修改每个使用 Foo 的测试测试吧)。

所以，经验法则是：如果你需要 Mock Foo 并且它由别人维护时，在 Foo 包中定义 Mock 类(更好的做法是在测试包中定义它，这样可以将测试代码更清晰地独立出来)，把它放到 mock_foo.h 中。那么每个想使用 Mock Foo 类的都可以在他们的测试代码中引用它。如果 Foo 改变了，那么只需要改一份 MockFoo 的代码，并且只有依赖这个变动函数的测试代码需要做相应的修改。

另一种做法是：你可以在 Foo 之上引入一个 FooAdaptor 层，并针对 FooAdaptor 这个新接口编程。因为你对 FooAdaptor 有控制权，你可以很容易地将 Foo 的改变隐藏掉。虽然这意味着在开始有更大的工作量，但认真构造的适配器接口会使你的代码更容易开发，也有更高的可读性，因为你构造的适配器接口 FooAdaptor 会比 Foo 更适合于你的特定领域开发。

Using Mocks in Tests

当你完成 Mock 类的定义之后，使用它是很简单的。典型的流程如下：

1. 引用那些你需要使用的 Google Mock 有关的命名空间(这样你就不用每次都把命名空间加到前面，请牢记，使用命名空间是一个好主意，并且对你的健康大有裨益)。
2. 创建一些 Mock 对象。
3. 对它们指定你的期望(一个函数要被调用多少次？ 用什么参数？ 它返回什么？ 等等)。
4. 用这些 Mocks 来测试一些代码。你可以选择 Google Test Assertions 来检查返回。如果一个 Mock 函数被调用次数多于期望，或是使用了错误的参数，你会马上得到一个错误 提示。
5. 当一个 Mock 对象被析构时，Google Mock 会自动检查在它上面的所有的期望是否都已经满足了。

下面是一个例子：

```
#include "path/to/mock-turtle.h"
#include "gmock/gmock.h"
#include "gtest/gtest.h"
using ::testing::AtLeast;                     // #1

TEST(PainterTest, CanDrawSomething) {
  MockTurtle turtle;                          // #2
```

```
  EXPECT_CALL(turtle, PenDown())              // #3
      .Times(AtLeast(1));

  Painter painter(&turtle);                   // #4

  EXPECT_TRUE(painter.DrawCircle(0, 0, 10));
}                                             // #5

int main(int argc, char** argv) {
  // The following line must be executed to initialize Google Mock
  // (and Google Test) before running the tests.
  ::testing::InitGoogleMock(&argc, argv);
  return RUN_ALL_TESTS();
}
```

正如你所猜测的一样，这个测试是检查 PenDown()是否被调用了至少一次。如果 Painter 对象并没有调用这个函数，你的测试就会失败，提示信息类似如下：

```
path/to/my_test.cc:119: Failure
Actual function call count doesn't match this expectation:
Actually: never called;
Expected: called at least once.
```

技巧 1：如果你从一个 Emacs Buffer 运行这个测试程序，你可以在错误信息的行号上敲 Enter 键，就可以直接跳到期望失败的那一行了。

技巧 2： 如果你的 Mock 对象永不释放，最后的检查是不会发生的。所以当你在堆上分配 Mock 对象时，你用内存泄露工具检查你的测试是一个好主意(译注 :推荐 valgrind)。

重要提示：Google Mock 要求期望在 Mock 函数被调用**之前**就设置好，否则行为将是**未定义**的。特别是你绝不能在 Mock 函数调用中间插入 EXPECT_CALL()。

这意味着 EXPECT_CALL()应该被理解为一个调用**在未来**的期望，而不是已经被调用过函数的期望。为什么 Google Mock 要以这种方式工作呢？嗯……，在前面指定期望可以让 Google Mock 在异常发生时马上可以提示，这时候上下文(栈信息，等等)还是有效的。这样会使调试更容易。

要承认的是，这个测试没有展示出 Google Mock 有什么强大之处。你完全可以不用 Google Mock 来得到相同的效果。但是别急，在下面的展示中，我会让你看到 Google Mock 的强大，它可以让你用 Mock 做**非常多**的事。

Using Google Mock with Any Testing Framework

如果你在用别的测试框架而不是 Google Test(比如，CppUnit 或 CxxUnit)，只需

要把上节中的 main 函数改成下面这样：

```
int main(int argc, char** argv) {
  // The following line causes Google Mock to throw
  // an exception on failure, which will be interpreted
  // by your testing framework as a test failure.
  ::testing::GTEST_FLAG(throw_on_failure) = true;
  ::testing::InitGoogleMock(&argc, argv);
  ... whatever your testing framework requires ...
}
```

这种方法中有一个 catch：它可以让 Google Mock 从 Mock 对象的析构函数中抛出一个异常。但有一些编译器，这会让测试程序崩溃(译注：可以参考 Effect C++第三版的 Item 8)。虽然你仍然可以注意到注意失败了，但这绝不是一个优雅的失败方式。

一个更好的方法是用 Google Test 的 event listener API 来以合理的方式报告一个测试失败给你的测试框架。你需要实现 OnTestPartResult()函数这个事件监听接口，但实现它也很简单。

如果上面的方法对你来说工作量太大，我建议你还是用 Google Test 吧，它与 Google Mock 可以无缝结合。如果你有什么 Google Test 满足不了你需求需求的原因 ,请告诉我们。

Setting Expectations

成功地使用 Mock 对象的关键是在它上面设置合适的期望。如果你设置的期望太过严格，你的测试可能会因为无关的改变而失败。如果你把期望设置的太过松驰，bugs 可能会溜过去。而你需要的是你的测试可以刚好捕获你想要捕获的那一种 bug。Google Mock 提供了一些方法可以让你的测试尺度*刚好*(just right)。

General Syntax

在 Goolge Mock 中，我们用 EXPECT_CALL()宏来设置一个 Mock 函数上的期望。一般语法是：

```
EXPECT_CALL(mock_object, method(matchers))
    .Times(cardinality)
    .WillOnce(action)
    .WillRepeatedly(action);
```

这个宏有两个参数：第一个是 Mock 对象，第二个参数是函数和它的参数。注意两个参数是用逗号(,)分隔的，而不是句号(.)。

这个宏可以跟一些可选*子句*，这些子句可以提供关于期望更多的信息。我们将会在下面

的小节中介绍每个子句有什么意义。

这些语法设计的一个目的是让它们读起来像是英语。比如你可能会直接猜出下面的代码是有什么含义

```
using ::testing::Return;...
EXPECT_CALL(turtle, GetX())
    .Times(5)
    .WillOnce(Return(100))
    .WillOnce(Return(150))
    .WillRepeatedly(Return(200));
```

公布答案，turtle 对象的 GetX()方法会被调用 5 次，它第一次返回 100，第二次返回 150，然后每次返回 200。许多人喜欢称这种语法方式为特定领域语言(Domain-Specific Language (DSL))。

注意：为什么我们要用宏来实现呢？有两个原因：第一，它让期望更容易被认出来(无论是 grep 还是人去阅读)，第二，它允许 Google Mock 可以得到失败期望在源文件的位置，从而使 Debug 更容易。

Matchers: What Arguments Do We Expect?

当一个 Mock 函数需要带参数时，我们必须指定我们期望的参数的是什么；比如：

```
// Expects the turtle to move forward by 100 units.
EXPECT_CALL(turtle, Forward(100));
```

有时你可能不想指定的太精确(还记得前面测试不应太严格吗？指定的太精确会导致测试健壮性不足，并影响测试的本意。所以我们鼓励你只指定那些必须要指定的参数，不要多，也不要少)。如果你只关心 Forward 是否会被调用，而不关心它用什么参数，你可以写_作为参数，它的意义是“任意”参数。

```
using ::testing::_;
...
// Expects the turtle to move forward.
EXPECT_CALL(turtle, Forward(_));
```

_是我们称为 **Matchers** 的一个例子，一个 matcher 是像一个断言，它可测试一个参数是否是我们期望的。你可用在 EXPECT_CALL()中任何写函数参数期望的地方用 matcher。

一个内置的 matchers 可以在 CheatSheet 中找到，比如，下面是 Ge(greater than or equal) matcher 的应用。

```
using ::testing::Ge;...
EXPECT_CALL(turtle, Forward(Ge(100)));
```

这个测试是检查 turtle 是否被告知要至少前进至少 100 个单位。

Cardinalities: How Many Times Will It Be Called?

在 EXPECT_CALL()之后第一个我们可以指定的子句是 Times()。我们称 Times 的参数为 **cardinality**，因为它是指这个函数应该被调用*多少次*。Times 可以让我们指定一个期望多次，而不用去写一次次地写这个期望。更重要的是，cardinality 可以是“模糊”的，就像 matcher 一样。它可以让测试者更准确地表达他测试的目的。

一个有趣的特例是我们指定 Times(0)。你也许已经猜到了，它是指函数在指定参数下不应该被调用，如果这个函数被调用了，Google Mock 会报告一个 Google Test 失败。

我们已经见过 AtLeast(n)这个模糊 cardinalities 的例子了。你可以在 CheatSheet 中找一个内置 cardinalities 列表。

Times()子句可以省略。**如果你省略 Times()，Google Mock 会推断出 cardinality 的值是什么。**这个规则很容易记：

- 如果在 EXPECT_CALL 中**既没有** WillOnce()**也没有** WillRepeatedly()，那推断出的 cardinality 就是 Times(1)。
- 如果有 n 个 WillOnce()，但**没有** WillRepeatedl()，其中 n >= 1，那么 cardinality 就是 Times(n)。
- 如果有 n 个 WillOnce() 和一个 WillRepeatedly() 其中 n >= 0 那么 cardinality 就是 Times(AtLeast(n))。

小测试：如果一个函数期望被调用 2 次，但被调用了 4 次，你认为会发生什么呢？

Actions: What Should It Do?

请记住一个 Mock 对象其实是没有实现的。是我们这些用户去告诉它当一个函数被调用时它应该做什么。这在 Google Mock 中是很简单的。

首先，如果 Mock 函数的返回类型是一个指针或是内置类型，那这个函数是有**默认行为**的(一个 void 函数直接返回，bool 函数返回 false，其它函数返回 0)。如果你不想改变它，那这种行为就会被应用。

其次，如果一个 Mock 函数没有默认行为，或默认行为不适合你，你可以用 WillOnce 来指定每一次的返回值是什么，最后可以选用 WillRepeatedly 来结束。比如：

```
using ::testing::Return;...
EXPECT_CALL(turtle, GetX())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
```

```
    .WillOnce(Return(300));
```

上面的意思是 turtle.GetX()会被调用*恰好* 3 次，并分别返回 100，200，300。

```
using ::testing::Return;...
EXPECT_CALL(turtle, GetY())
    .WillOnce(Return(100))
    .WillOnce(Return(200))
    .WillRepeatedly(Return(300));
```

上面的意思是指 turtle.GetY()将*至少*被调用 2 次，第一次返回 100，第二次返回 200，从第三次以后都返回 300。

当然，你如果你明确写上 Times()，Google Mock 不会去推断 cardinality 了。如果你指定的 cardinality 大于 WillOnce()子句的个数时会发生什么呢？嗯，当 WillOnce()用完了之后，Google Mock 会每次对函数采用*默认*行为。

我们在 WillOnce()里除了写 Return()我们还能做些什么呢？你可以用 ReturnRef(variable)，或是调用一个预先定义好的函数，自己在 Others 中找吧。

重要提示：EXPECT_CALL()只对行为子句求一次值，尽管这个行为可能出现很多次。所以你必须小心这种副作用。下面的代码的结果可能与你想的不太一样。

```
int n = 100;
EXPECT_CALL(turtle, GetX())
    .Times(4)
    .WillRepeatedly(Return(n++));
```

它并不是依次返回 100，101，102...，而是每次都返回 100，因为 n++只会被求一次值。类似的，Return(new Foo)当 EXPECT_CALL()求值时只会创建一个 Foo 对象，所以它会每次都返回相同的指针。如果你希望每次都看到不同的结果，你需要定义一个自定义行为，我们将在 CookBook 中指导你。

现在又是一个小测验的时候了！你认为下面的代码是什么意思？

```
using ::testing::Return;...
EXPECT_CALL(turtle, GetY())
    .Times(4)
    .WillOnce(Return(100));
```

显然，turtle.Get()期望被调用 4 次。但如果你认为它每次都会返回 100，那你就要再考虑一下了！记住，每次调用都会消耗一个 WillOnce()子句，消耗完之后，就会使用默认行为。所以正确的答案是 turtle.GetY()第一次返回 100，以后每次都返回 0，因为 0 是默认行为的返回值。

Using Multiple Expectations

至今为止，我们只展示了如何使用单个期望。但是在现实中，你可能想指定来自不同 Mock 对象的 Mock 函数上的期望。

默认情况下，当一个 Mock 函数被调用时，Google Mock 会通过定义顺序的**逆序**去查找期望，当找到一个与参数匹配的有效的期望时就停下来（你可以把这个它想成是“老的规则覆盖新的规则”）。如果匹配的期望不能再接受更多的调用时，你就会收到一个超出上界的失败，下面是一个例子：

```
using ::testing::_;...
EXPECT_CALL(turtle, Forward(_));  // #1
EXPECT_CALL(turtle, Forward(10))  // #2
       .Times(2);
```

如果 Forward(10)被连续调用 3 次，第 3 次调用它会报出一个错误，因为最后一个匹配期望(#2)已经饱和了。但是如果第 3 次的 Forward(10)替换为 Forward(20)，那它就不会报错，因数现在#1 将会是匹配的期望了。

边注：为什么 Google Mock 会以*逆序*去匹配期望呢？原因是为了可以让用户开始时使用 Mock 对象的默认行为，或是一些比较松驰的匹配条件，然后写一些更明确的期望。所以，如果你在同一个函数上有两个期望，你当然是想先匹配更明确的期望，**然后**再匹配其它的，或是可以说明确的规则会隐藏更宽泛的规则。

Ordered vs Unordered Calls

默认情况下，即使是在前一个期望没有被匹配的情况下，一个期望仍然可以被匹配。换句话说，调用的匹配顺序不会按照期望指定的顺序去匹配。

有时，你可能想让所有的期望调用都以一个严格的顺序来匹配，这在 Google Mock 中是很容易的：

```
using ::testing::InSequence;...
TEST(FooTest, DrawsLineSegment) {
  ...
  {
    InSequence dummy;

    EXPECT_CALL(turtle, PenDown());
    EXPECT_CALL(turtle, Forward(100));
    EXPECT_CALL(turtle, PenUp());
  }
  Foo();
```

```
}
```

创建 InSequence 的一个对象后，在这个对象作用域中的期望都会以*顺序*存放，并要求调用以这个*顺序*匹配。因为我们只是依赖这个对象的构造函数和析构函数来完成任务，所以对象的名字并不重要。

（ 如果你只是关心某些调用的相对顺序，而不是所有调用的顺序？可以指定一个任意的相对顺序吗？答案是...可以！如果你比较心急，你可以在 CookBook 中找到相关的细节。）

All Expectations Are Sticky (Unless Said Otherwise)

现在让我们做一个小测验，看你掌握 Mock 到什么程度了。你如何测试 turtle *恰好*经过原点两次？

当你想出你的解法之后，看一下我们的答案比较一下（ 先自己想，别作弊 ）。

```
using ::testing::_;...
EXPECT_CALL(turtle, GoTo(_, _))  // #1
    .Times(AnyNumber());
EXPECT_CALL(turtle, GoTo(0, 0))  // #2
    .Times(2);
```

假设 turtle.GoTo(0,0)被调用了 3 次。在第 3 次，Google Mock 会找到参数匹配期望#2。因为我们想要的是恰好经过原点两次，所以 Google Mock 会立即报告一个错误。上面的内容其实就是我们在“Using Multiple Expectations”中说过的。

上面的例子说明了 Google Mock 中**默认情况下期望是严格的**，即是指期望在达到它们指定的调用次数上界后仍然是有效的。这是一个很重要的规则，因为它影响着指定的意义，而且这种规则与许多别的 Mock 框架中是**不一样**的（ 我们为什么会设计的不一样？因为我们认为我们的规则会使一般的用例更容易表达和理解 ）。

简单？让我看一下你是不是真懂了：下面的代码是什么意思：

```
using ::testing::Return;
...
for (int i = n; i > 0; i--) {
  EXPECT_CALL(turtle, GetX())
      .WillOnce(Return(10*i));
}
```

如果你认为 turtle.GetX()会被调用 n 次，并依次返回 10，20，30，...，唉，你还是再想想吧！问题是，我们都说过了，期望是严格的。所以第 2 次 turtle.GetX()被调用时，最后一个 EXPECT_CALL()会被匹配，所以马上会引起“超出上界”的错误。上面的代码其实没什么用途。

一个正确表达 turtle.GetX()返回 10，20，30，...，的方法是明确地说明期望*不是*严格的。换句话说，在期望饱和之后就*失效*。

```
using ::testing::Return;
...
for (int i = n; i > 0; i--) {
  EXPECT_CALL(turtle, GetX())
    .WillOnce(Return(10*i))
    .RetiresOnSaturation();
}
```

并且，有一个更好的解决方法，在这个例子中，我们期望调用以特定顺序执行。因为顺序是一个重要的因素，我们应该用 InSequence 明确地表达出顺序：

```
using ::testing::InSequence;
using ::testing::Return;
...
{
  InSequence s;

  for (int i = 1; i <= n; i++) {
    EXPECT_CALL(turtle, GetX())
        .WillOnce(Return(10*i))
        .RetiresOnSaturation();
  }
}
```

顺便说一下，另一个期望可能*不严格*的情况是当它在一个顺序中，当这个期望饱和后，它就自动失效，从而让下一个期望有效。

Uninteresting Calls

一个 Mock 对象可能有很多函数，但并不是所有的函数你都关心。比如，在一些测试中，你可能不关心 GetX()和 GetY()被调用多少次。

在 Google Mock 中，你如果不关心一个函数，很简单，你什么也不写就可以了。如果这个函数的调用发生了，你会看到测试输出一个警告，但它不会是一个失败。

What Now?

恭喜！你已经学习了足够的 Google Mock 的知识了，你可以开始使用它了。现在你也许想加入 googlemock 讨论组，并开始真正地用 Google Mock 开始写一些测试——它是很有意思的，嗨，这可能是会上瘾的，我可是警告过你了喔！

如果你想提高你的Mock等级 ,你可以移步至CookBook。你可以在那学习更多的Google Mock 高级特性——并提高你的幸福指数和测试快乐级别。

Copyright notice