您可以在这里找到使用谷歌Mock的方法。如果您还没有，请先阅读ForDummies文档，以确保您理解了基础知识。

**注意:谷歌模拟存在于测试名称空间中。**为了可读性，建议使用::testing::Foo;在使用由谷歌Mock定义的名称Foo之前，在您的文件中使用一次。为了简单起见，我们在本页中省略了using语句，但是您应该在自己的代码中这样做。

# 创建模拟类

## 模拟私有或受保护的方法

您必须始终将模拟方法定义(MOCK\_METHOD\*)放在模拟类的public:部分中，而不管模拟的方法在基类中是公共的、受保护的还是私有的。这允许ON\_CALL和EXPECT\_CALL从模拟类外部引用模拟函数。(是的，c++允许子类改变基类中虚函数的访问级别。)例子:

|  |
| --- |
| class Foo {  public:  ...  virtual bool Transform(Gadget\* g) = 0;  protected:  virtual void Resume();  private:  virtual int GetTimeOut();  };  class MockFoo : public Foo {  public:  ...  MOCK\_METHOD1(Transform, bool(Gadget\* g));  // The following must be in the public section, even though the  // methods are protected or private in the base class.  MOCK\_METHOD0(Resume, void());  MOCK\_METHOD0(GetTimeOut, int());  }; |

## Mocking重载方法

您可以像往常一样模拟重载函数。不需要特别注意:

|  |
| --- |
| class Foo {  ...  // Must be virtual as we'll inherit from Foo.  virtual ~Foo();  // Overloaded on the types and/or numbers of arguments.  virtual int Add(Element x);  virtual int Add(int times, Element x);  // Overloaded on the const-ness of this object.  virtual Bar& GetBar();  virtual const Bar& GetBar() const;  };  class MockFoo : public Foo {  ...  MOCK\_METHOD1(Add, int(Element x));  MOCK\_METHOD2(Add, int(int times, Element x);  MOCK\_METHOD0(GetBar, Bar&());  MOCK\_CONST\_METHOD0(GetBar, const Bar&());  }; |

**注意:如果你不模仿所有版本的重载方法，编译器会给你一个关于基类中某些方法被隐藏的警告。**要解决这个问题，可以使用using将它们纳入范围:

|  |
| --- |
| class MockFoo : public Foo {  ...  using Foo::Add;  MOCK\_METHOD1(Add, int(Element x));  // We don't want to mock int Add(int times, Element x);  ...  }; |

## 模拟类模板

要模拟类模板，可以将\_T附加到MOCK\_\*宏:

|  |
| --- |
| template <typename Elem>  class StackInterface {  ...  // Must be virtual as we'll inherit from StackInterface.  virtual ~StackInterface();  virtual int GetSize() const = 0;  virtual void Push(const Elem& x) = 0;  };  template <typename Elem>  class MockStack : public StackInterface<Elem> {  ...  MOCK\_CONST\_METHOD0\_T(GetSize, int());  MOCK\_METHOD1\_T(Push, void(const Elem& x));  }; |

## Mocking Nonvirtual方法

Google Mock可以模拟非虚函数，以便在所谓的hi-perf依赖项注入中使用。

在这种情况下，与实际类共享一个公共基类不同，模拟类与实际类无关，但是包含具有相同签名的方法。模拟非虚拟方法的语法与模拟虚拟方法相同:

|  |
| --- |
| // A simple packet stream class. None of its members is virtual.  class ConcretePacketStream {  public:  void AppendPacket(Packet\* new\_packet);  const Packet\* GetPacket(size\_t packet\_number) const;  size\_t NumberOfPackets() const;  ...  };  // A mock packet stream class. It inherits from no other, but defines  // GetPacket() and NumberOfPackets().  class MockPacketStream {  public:  MOCK\_CONST\_METHOD1(GetPacket, const Packet\*(size\_t packet\_number));  MOCK\_CONST\_METHOD0(NumberOfPackets, size\_t());  ...  }; |

注意，与实际类不同，模拟类不定义AppendPacket()。只要测试不需要调用它，就没有问题。

接下来，您需要一种方式来表示希望在生产代码中使用ConcretePacketStream，并在测试中使用MockPacketStream。由于函数不是虚拟的，而且这两个类是不相关的，所以必须在编译时(而不是运行时)指定您的选择。

一种方法是对需要使用包流的代码进行模板化。更具体地说，您将为您的代码提供一个包流类型的模板类型参数。在生产环境中，您将使用ConcretePacketStream作为类型参数实例化您的模板。在测试中，您将使用MockPacketStream实例化相同的模板。例如，你可以这样写:

|  |
| --- |
| template <class PacketStream>  void CreateConnection(PacketStream\* stream) { ... }  template <class PacketStream>  class PacketReader {  public:  void ReadPackets(PacketStream\* stream, size\_t packet\_num);  }; |

然后可以在生产代码中使用CreateConnection<ConcretePacketStream>()和PacketReader<ConcretePacketStream>，在测试中使用CreateConnection<MockPacketStream>()和PacketReader<MockPacketStream>。

|  |
| --- |
| MockPacketStream mock\_stream;  EXPECT\_CALL(mock\_stream, ...)...;  .. set more expectations on mock\_stream ...  PacketReader<MockPacketStream> reader(&mock\_stream);  ... exercise reader ... |

## Mocking free function

可以使用google Mock来模拟自由函数(即c风格的函数或静态方法)。您只需重写代码以使用接口(抽象类)。

不是直接调用一个自由函数(比如OpenFile)，而是为它引入一个接口，并有一个具体的子类来调用这个自由函数:

|  |
| --- |
| class FileInterface {  public:  ...  virtual bool Open(const char\* path, const char\* mode) = 0;  };  class File : public FileInterface {  public:  ...  virtual bool Open(const char\* path, const char\* mode) {  return OpenFile(path, mode);  }  }; |

您的代码应该与FileInterface对话以打开文件。现在很容易模拟出这个函数。

这可能看起来很麻烦，但是在实践中，您经常有多个相关的函数可以放在同一个接口中，因此每个函数的语法开销要低得多。

如果您担心虚拟函数带来的性能开销，而分析证实了您的担心，那么您可以将此与模拟非虚拟方法相结合。

## **The Nice, the Strict, and the Naggy**

如果模拟方法没有EXPECT\_CALL规范，但是被调用了，那么谷歌模拟将打印关于“无兴趣的调用”的警告。基本原理是:

* 在编写测试之后，可以向接口添加新方法。我们不应该因为调用了一个它不知道的方法而使测试失败。
* 然而，这也可能意味着在测试中有一个bug，所以谷歌Mock也不应该是静默的。如果用户认为这些调用是无害的，那么他可以添加一个EXPECT\_CALL()来抑制警告。

然而，有时您可能想要隐藏所有“无趣的调用”警告，而有时您可能想要相反的结果，即将所有警告都视为错误。谷歌Mock允许您在每个模拟对象的基础上做出决策。

假设您的测试使用一个模拟类MockFoo:

|  |
| --- |
| TEST(...) {  MockFoo mock\_foo;  EXPECT\_CALL(mock\_foo, DoThis());  ... code that uses mock\_foo ...  } |

如果调用了DoThis()之外的mock\_foo方法，则谷歌Mock将报告该方法作为警告。然而，如果你重写你的测试使用NiceMock<MockFoo>代替，警告将消失，导致一个干净的测试输出:

|  |
| --- |
| using ::testing::NiceMock;  TEST(...) {  NiceMock<MockFoo> mock\_foo;  EXPECT\_CALL(mock\_foo, DoThis());  ... code that uses mock\_foo ...  } |

NiceMock<MockFoo>是MockFoo的子类，所以它可以在接受MockFoo的任何地方使用。

如果MockFoo的构造函数接受了一些参数，它也可以工作，因为NiceMock<MockFoo>“继承”了MockFoo的构造函数:

|  |
| --- |
| using ::testing::NiceMock;  TEST(...) {  NiceMock<MockFoo> mock\_foo(5, "hi"); // Calls MockFoo(5, "hi").  EXPECT\_CALL(mock\_foo, DoThis());  ... code that uses mock\_foo ...  } |

StrictMock的用法与此类似，只是它会导致所有无趣的调用失败:

|  |
| --- |
| using ::testing::StrictMock;  TEST(...) {  StrictMock<MockFoo> mock\_foo;  EXPECT\_CALL(mock\_foo, DoThis());  ... code that uses mock\_foo ...  // The test will fail if a method of mock\_foo other than DoThis()  // is called.  } |

尽管有一些警告(我不像下一个家伙那样喜欢它们，但遗憾的是它们是c++局限性的副作用):

1. NiceMock<MockFoo>和StrictMock<MockFoo>只适用于直接在MockFoo类中使用MOCK\_METHOD\*宏家族定义的模拟方法。如果mock方法是在MockFoo的基类中定义的，那么“nice”或“strict”修饰符可能不会影响它，具体取决于编译器。特别是，不支持嵌套NiceMock和StrictMock(例如NiceMock<StrictMock<MockFoo> >)。
2. 基本mock (MockFoo)的构造函数不能通过非const引用传递参数，而这恰好是谷歌c++风格指南所禁止的。
3. 在MockFoo的构造函数或析构函数期间，模拟对象既不友好也不严格。如果构造函数或析构函数调用该对象上的模拟方法，这可能会导致意外。(然而，这种行为与c++的一般规则是一致的:如果构造函数或析构函数调用此对象的虚方法，则该方法将被视为非虚方法。换句话说，对于基类的构造函数或析构函数，此对象的行为类似于基类的实例，而不是派生类。为了安全起见，这条规则是必要的。否则，基构造函数可能在初始化派生类成员之前使用它们，或者基析构函数可能在销毁派生类成员之后使用它们。

最后，对于何时使用naggy或strict模拟，您应该非常谨慎，因为它们会使测试更脆弱，更难于维护。在重构代码而不改变其外部可见行为时，理想情况下不需要更新任何测试。但是，如果您的代码与naggy模拟进行交互，您可能会因为更改而收到大量警告。更糟糕的是，如果您的代码与一个严格的mock交互，那么您的测试可能会开始失败，您将不得不修复它们。我们的一般建议是在大多数情况下使用漂亮的模拟(还不是默认的)，在开发或调试测试时使用naggy模拟(当前的默认)，最后使用严格的模拟。

## 在不破坏现有代码的情况下简化接口

有时一个方法有一长串的参数，这些参数大多是无趣的。例如,

|  |
| --- |
| class LogSink {  public:  ...  virtual void send(LogSeverity severity, const char\* full\_filename,  const char\* base\_filename, int line,  const struct tm\* tm\_time,  const char\* message, size\_t message\_len) = 0;  }; |

这个方法的参数列表很长，很难处理(假设消息参数甚至没有以0结尾)。如果我们按原样模仿它，那么使用模仿将会很尴尬。然而，如果我们试图简化这个接口，我们将需要修复所有依赖于它的客户机，这通常是不可行的。

技巧是在模拟类中重新调度方法:

|  |
| --- |
| class ScopedMockLog : public LogSink {  public:  ...  virtual void send(LogSeverity severity, const char\* full\_filename,  const char\* base\_filename, int line, const tm\* tm\_time,  const char\* message, size\_t message\_len) {  // We are only interested in the log severity, full file name, and  // log message.  Log(severity, full\_filename, std::string(message, message\_len));  }  // Implements the mock method:  //  // void Log(LogSeverity severity,  // const string& file\_path,  // const string& message);  MOCK\_METHOD3(Log, void(LogSeverity severity, const string& file\_path,  const string& message));  }; |

通过定义一个带有修剪后的参数列表的新mock方法，我们使mock类更加友好。

## 替代模拟具体类

您经常会发现自己使用的类没有实现接口。为了测试使用这样一个类(让我们称它为Concrete)的代码，您可能会试图将Concrete方法变为虚方法，然后对其进行模拟。

尽量不要那么做。

使一个非虚拟函数虚拟是一个重大的决定。它创建了一个扩展点，子类可以在这里调整类的行为。这削弱了对类的控制，因为现在很难维护类的不变量。只有在子类有正当理由覆盖某个函数时，才应该使该函数为虚函数。

直接模拟具体类是有问题的，因为它在类和测试之间创建了一个紧密的耦合——类中的任何小的改变都可能使您的测试无效，并使测试维护成为一件痛苦的事情。

为了避免这些问题，许多程序员一直在实践“对接口编码”:您的代码将定义接口并与之通信，而不是与具体类通信。然后将该接口作为适配器在具体的基础上实现。在测试中，您可以轻松地模拟该接口，以观察代码的运行情况。

这种技术需要一些开销:

* 您需要支付虚拟函数调用的成本(通常不成问题)。
* 程序员需要学习更多的抽象。

然而，除了更好的可测试性之外，它还可以带来显著的好处:

* Concrete的API可能不太适合您的问题域，因为您可能不是它试图服务的唯一客户端。通过设计你自己的接口，你有机会根据你的需要来调整它——你可以添加更高级的功能，重命名一些东西，等等，而不仅仅是修饰类。这允许您以一种更自然的方式编写代码(接口的用户)，这意味着它将更具可读性、更易于维护，并且您将更有生产力。
* 如果混凝土的实现必须更改，您不必在使用它的任何地方重写。相反，您可以吸收接口实现中的更改，其他代码和测试将与此更改隔离。

有些人担心，如果每个人都在实践这种技术，他们最终会编写大量冗余代码。这种担忧是完全可以理解的。然而，原因有二:

* 不同的项目可能需要以不同的方式使用混凝土，因此它们的最佳接口将是不同的。因此，它们中的每一个都有自己特定于领域的接口，并且它们不是相同的代码。
* 如果有足够多的项目想要使用相同的接口，他们总是可以共享它，就像他们一直在共享具体的东西一样。您可以将接口和适配器签入到Concrete附近(可能在contrib子目录中)，让许多项目使用它。

您需要为您的特定问题仔细权衡利弊，但是我想向您保证，Java社区已经实践了很长时间，并且它是一种经过验证的有效技术，可以应用于各种各样的情况。:-)

## **Delegating Calls to a Fake**

有时，您有一个非常重要的接口伪实现。例如:

|  |
| --- |
| class Foo {  public:  virtual ~Foo() {}  virtual char DoThis(int n) = 0;  virtual void DoThat(const char\* s, int\* p) = 0;  };  class FakeFoo : public Foo {  public:  virtual char DoThis(int n) {  return (n > 0) ? '+' :  (n < 0) ? '-' : '0';  }  virtual void DoThat(const char\* s, int\* p) {  \*p = strlen(s);  }  }; |

现在您想要模拟这个接口，这样就可以对它设置期望。但是，您还希望使用FakeFoo作为默认行为，因为在模拟对象中复制它需要大量工作。

当您使用谷歌mock定义模拟类时，您可以让它将其默认操作委托给您已经拥有的一个伪类，使用以下模式:

|  |
| --- |
| using ::testing::\_;  using ::testing::Invoke;  class MockFoo : public Foo {  public:  // Normal mock method definitions using Google Mock.  MOCK\_METHOD1(DoThis, char(int n));  MOCK\_METHOD2(DoThat, void(const char\* s, int\* p));  // Delegates the default actions of the methods to a FakeFoo object.  // This must be called \*before\* the custom ON\_CALL() statements.  void DelegateToFake() {  ON\_CALL(\*this, DoThis(\_))  .WillByDefault(Invoke(&fake\_, &FakeFoo::DoThis));  ON\_CALL(\*this, DoThat(\_, \_))  .WillByDefault(Invoke(&fake\_, &FakeFoo::DoThat));  }  private:  FakeFoo fake\_; // Keeps an instance of the fake in the mock.  }; |

有了它，您可以像往常一样在测试中使用MockFoo。只需记住，如果您没有显式地在ON\_CALL()或EXPECT\_CALL()中设置一个动作，就会调用fake来执行此操作:

|  |
| --- |
| using ::testing::\_;  TEST(AbcTest, Xyz) {  MockFoo foo;  foo.DelegateToFake(); // Enables the fake for delegation.  // Put your ON\_CALL(foo, ...)s here, if any.  // No action specified, meaning to use the default action.  EXPECT\_CALL(foo, DoThis(5));  EXPECT\_CALL(foo, DoThat(\_, \_));  int n = 0;  EXPECT\_EQ('+', foo.DoThis(5)); // FakeFoo::DoThis() is invoked.  foo.DoThat("Hi", &n); // FakeFoo::DoThat() is invoked.  EXPECT\_EQ(2, n);  } |

**一些建议:**

* 如果需要，您仍然可以通过提供自己的ON\_CALL()或在EXPECT\_CALL()中使用. willonce () / . will()来覆盖默认操作。
* 在DelegateToFake()中，您只需要委托您打算使用其伪实现的方法。
* 这里讨论的通用技术适用于重载方法，但是您需要告诉编译器您指的是哪个版本。要消除对模拟函数(在ON\_CALL()的括号内指定的函数)的歧义，请参阅本页的“在重载函数之间进行选择”部分;要消除伪函数(放置在Invoke()中的函数)的歧义，可以使用static\_cast来指定函数的类型。例如,如果字符类Foo有方法做(int n)和bool这么做(双x)常量,和您想调用后者,您需要编写调用(&fake\_, static\_cast < bool (FakeFoo:: \*)(双)const > (&FakeFoo::这么做))而不是调用(&fake\_, &FakeFoo::这么做)(static\_cast尖括号内的奇怪的事情是一个函数指针的类型,第二个就是()方法)。
* 不得不把假的和假的混在一起通常是出了问题的迹象。也许您还没有习惯基于交互的测试方法。或者您的接口承担了太多的角色，应该进行拆分。因此，不要滥用它。我们只建议在重构代码时将其作为中间步骤。

关于混合mock和fake的技巧，这里有一个示例，说明为什么它可能是不好的信号:假设您有一个用于低级系统操作的类系统。特别是，它执行文件和I/O操作。假设您想测试代码如何使用System执行I/O，并且只想让文件操作正常工作。如果模拟整个系统类，就必须为文件操作部分提供一个假实现，这表明系统承担了太多的角色。

相反，您可以定义一个FileOps接口和一个IOOps接口，并将系统的功能分为这两个部分。然后您可以模拟IOOps而不需要模拟FileOps。

## **Delegating Calls to a Real Object**

当使用测试双精度对象(模拟、假对象、存根等)时，有时它们的行为与真实对象不同。这种差异可能是有意的(比如在模拟错误以便测试错误处理代码时)，也可能是无意的。如果您的模拟错误地具有与实际对象不同的行为，那么您最终得到的代码可能通过了测试，但在生产中失败了。

您可以使用委托到实际的技术来确保模拟具有与实际对象相同的行为，同时保留验证调用的能力。这一技巧与“委托伪造”技巧非常相似，不同之处在于我们使用的是真实物体而不是赝品。这里有一个例子:

|  |
| --- |
| using ::testing::\_;  using ::testing::AtLeast;  using ::testing::Invoke;  class MockFoo : public Foo {  public:  MockFoo() {  // By default, all calls are delegated to the real object.  ON\_CALL(\*this, DoThis())  .WillByDefault(Invoke(&real\_, &Foo::DoThis));  ON\_CALL(\*this, DoThat(\_))  .WillByDefault(Invoke(&real\_, &Foo::DoThat));  ...  }  MOCK\_METHOD0(DoThis, ...);  MOCK\_METHOD1(DoThat, ...);  ...  private:  Foo real\_;  };  ...  MockFoo mock;  EXPECT\_CALL(mock, DoThis())  .Times(3);  EXPECT\_CALL(mock, DoThat("Hi"))  .Times(AtLeast(1));  ... use mock in test ... |

这样，谷歌Mock将验证您的代码进行了正确的调用(使用正确的参数，以正确的顺序，调用正确的次数，等等)，一个实际的对象将响应这些调用(因此行为将与生产中一样)。这是两全其美的。

## **Delegating Calls to a Parent Class**

理想情况下，您应该编写接口的代码，其方法都是纯虚拟的。在现实中，有时您确实需要Mocking一个不纯的虚拟方法(i。e，它已经有一个实现)。例如:

|  |
| --- |
| class Foo {  public:  virtual ~Foo();  virtual void Pure(int n) = 0;  virtual int Concrete(const char\* str) { ... }  };  class MockFoo : public Foo {  public:  // Mocking a pure method.  MOCK\_METHOD1(Pure, void(int n));  // Mocking a concrete method. Foo::Concrete() is shadowed.  MOCK\_METHOD1(Concrete, int(const char\* str));  }; |

有时您可能想要调用Foo::Concrete()而不是MockFoo::Concrete()。也许您想要将它作为存根操作的一部分，或者您的测试根本不需要模拟Concrete()(但是，当您不需要模拟它的一个方法时，就必须定义一个新的模拟类，这太痛苦了)。

诀窍是在模拟类中为访问基类中的实际方法留一个后门:

|  |
| --- |
| class MockFoo : public Foo {  public:  // Mocking a pure method.  MOCK\_METHOD1(Pure, void(int n));  // Mocking a concrete method. Foo::Concrete() is shadowed.  MOCK\_METHOD1(Concrete, int(const char\* str));  // Use this to call Concrete() defined in Foo.  int FooConcrete(const char\* str) { return Foo::Concrete(str); }  }; |

现在，你可以调用Foo::Concrete()内的一个动作:

|  |
| --- |
| using ::testing::\_;  using ::testing::Invoke;  ...  EXPECT\_CALL(foo, Concrete(\_))  .WillOnce(Invoke(&foo, &MockFoo::FooConcrete)); |

或者告诉模拟对象你不想模拟Concrete():

|  |
| --- |
| using ::testing::Invoke;  ...  ON\_CALL(foo, Concrete(\_))  .WillByDefault(Invoke(&foo, &MockFoo::FooConcrete)); |

(为什么我们不直接写Invoke(&foo， &foo::Concrete)?如果这样做，MockFoo::Concrete()将被调用(并导致无限递归)，因为Foo::Concrete()是虚的。这就是c++的工作原理。)

# 使用匹配器

## 精确匹配实参值

你可以指定一个mock方法需要哪些参数:

测试使用::::返回;…  
  
EXPECT\_CALL (foo,这么做(5).WillOnce(返回(' a '));  
  
EXPECT\_CALL (foo, DoThat (“Hello”、酒吧));

## 使用简单的匹配器

你可以使用匹配器来匹配具有特定属性的参数:

使用::testing::Ge;使用::testing::NotNull;使用::testing::Return;…  
  
  
  
//参数必须是>= 5. willonce (Return('a');  
  
EXPECT\_CALL (foo, DoThat (“Hello”, NotNull ()));  
//第二个参数不能为空。

一个常用的匹配器是\_，它可以匹配任何东西:

测试使用::::\_;使用::测试::NotNull;…  
  
  
EXPECT\_CALL (foo, DoThat (\_ NotNull ()));

## 结合匹配器

您可以使用AllOf()、AnyOf()和Not()从现有的匹配器构建复杂的匹配器:

使用::testing::AllOf;使用::testing::Gt;使用::testing::HasSubstr;使用::testing::Ne;使用::testing::Not;…  
  
  
  
  
  
//参数必须是> 5和!= 10。  
EXPECT\_CALL (foo,这么做(所有(Gt (5)、Ne (10))));  
  
  
//第一个参数不能包含子字符串“blah”。  
EXPECT\_CALL (foo, DoThat(不是(HasSubstr(“废话”)),NULL));

## 铸造匹配器

谷歌模拟匹配器是静态类型的，这意味着如果您使用了错误类型的匹配器(例如，如果您使用Eq(5)来匹配字符串参数)，那么编译器可以捕捉到您的错误。对你有好处!

然而，有时您知道自己在做什么，希望编译器给您一些宽松。长时间的一个例子是,你有一个匹配器和想匹配的参数是int。尽管这两种类型并不是完全一样的,没有什么真的错了使用一个匹配器<时间>匹配int——毕竟,我们可以先把int参数之前给匹配器。

为了支持这种需求，谷歌Mock提供了SafeMatcherCast<T>(m)函数。它将一个匹配器m强制转换为类型匹配器<T>。为了确保安全，谷歌模拟检查(让U是m接受的类型):

1. 类型T可以隐式转换为类型U;
2. 当T和U都是内置的算术类型(bool、integers和浮点数)时，从T到U的转换是无损的(换句话说，任何可以由T表示的值也可以由U表示);和
3. 当U是引用时，T也必须是引用(因为底层匹配器可能对U值的地址感兴趣)。

如果不满足这些条件，代码将无法编译。

这里有一个例子:

使用::testing::SafeMatcherCast;//一个基类和一个子类。  
  
  
类基{…};class Derived: public Base{…  
{public: MOCK\_METHOD1(DoThis, void(Derived\* Derived));};  
  
  
  
  
  
  
  
MockFoo foo;  
// m是我们从某个地方获得的一个匹配器<Base\*>。  
EXPECT\_CALL (foo,这么做(SafeMatcherCast <派生\* > (m)));

如果您认为SafeMatcherCast<T>(m)太有限，您可以使用类似的函数MatcherCast<T>(m)。不同之处在于MatcherCast只要能够将类型T静态转换为类型U就可以工作。

MatcherCast实际上允许您绕过c++的类型系统(例如，static\_cast并不总是安全的，因为它可能会丢弃信息)，所以要小心不要误用/滥用它。

## 选择重载函数

如果您希望调用一个重载函数，编译器可能需要一些帮助来确定它是哪个重载版本。

若要消除在此对象的一致性上重载的函数的歧义，请使用Const()参数包装器。

类MockFoo: public Foo{…  
  
  
  
MOCK\_METHOD0 (GetBar酒吧());  
MOCK\_CONST\_METHOD0 (GetBar const酒吧());},…  
  
  
  
MockFoo foo;  
酒吧bar1 bar2;  
//非const GetBar(). willonce (ReturnRef(bar1));  
  
GetBar(). willonce (ReturnRef(bar2));

(Const()是由谷歌Mock定义的，它的参数返回一个Const引用。)

为了消除具有相同数量的参数但不同参数类型的重载函数的歧义，您可能需要指定matcher的确切类型，可以在matcher <type>()中包装您的matcher，或者使用类型固定的matcher (TypedEq<type>， An<type>()，等等):

类MockPrinter: public Printer {public: MOCK\_METHOD1(Print, void(int n));  
  
  
  
  
  
  
  
{MOCK\_METHOD1(Print, void(char c));};  
  
  
  
  
  
EXPECT\_CALL(打印机,打印(一个< int > ()));/ /空白打印(int);  
EXPECT\_CALL(打印机,打印(匹配器< int > (Lt (5))));/ /空白打印(int);  
EXPECT\_CALL(打印机,打印(TypedEq < char > (' a ')));/ /空白打印(char);  
  
printer.Print (3);  
printer.Print (6);  
printer.Print (' a ');}

## 根据参数执行不同的操作

当调用模拟方法时，将选择最后一个仍处于活动状态的匹配期望(请考虑“更新的覆盖旧的”)。你可以让一个方法根据它的参数值做不同的事情，就像这样:

使用::testing::\_;使用::testing::Lt;使用::testing::Return;…  
  
  
  
//默认情况。  
EXPECT\_CALL (foo,这么做(\_)).WillRepeatedly(返回(b));  
  
  
//更具体的情况。  
EXPECT\_CALL (foo,这么做(Lt (5))) .WillRepeatedly(返回(' a '));

现在，如果以小于5的值调用foo.DoThis()，将返回'a';否则将返回“b”。

## 将多个参数作为一个整体进行匹配

有时单独匹配这些参数是不够的。例如，我们可能想说第一个参数必须小于第二个参数。With()子句允许我们将模拟函数的所有参数作为一个整体进行匹配。例如,

测试使用::::\_;使用::测试::Lt;使用::测试::不,…  
  
  
  
EXPECT\_CALL (foo, InRange (Ne (0) \_)); (Lt ());

表示InRange()的第一个参数必须不为0，并且必须小于第二个参数。

With()中的表达式必须是matcher <::testing::tuple<A1，…， An> >，其中A1，…， An是函数参数的类型。

你也可以在。with()里面写AllArgs(m)而不是m。这两种形式是等价的，但是. with (AllArgs(Lt()))比. with (Lt())可读性更好。

你可以使用Args<k1，…， kn>(m)来匹配m的n个选定参数(作为一个元组)。

使用::测试::\_;使用::测试::AllOf;使用::测试::Args;使用::测试::Lt;  
  
  
  
  
EXPECT\_CALL (foo,等等(\_、\_、\_));(所有(Args < 0,1 > (Lt ()), arg游戏< 1,2 > (Lt ())));

在x < y < z的地方，通过参数x, y, z调用Blah()。

作为一个方便的示例，谷歌Mock为二元组提供了一些匹配器，包括上面的Lt()匹配器。完整的列表请参阅备忘单。

注意，如果您希望将参数传递给自己的谓词(例如，. with (Args<0, 1>(Truly(&MyPredicate)))，则必须将该谓词编写为以::testing::tuple作为参数;谷歌Mock将把n个选定的参数作为一个元组传递给谓词。

## 使用匹配器作为谓词

您是否注意到，匹配器只是一个奇特的谓词，它也知道如何描述自己?许多现有的算法都采用谓词作为参数(例如STL的<algorithm>标头中定义的那些)，如果不允许谷歌模拟匹配器参与，那就太可惜了。

幸运的是，您可以使用matcher，将一元谓词函子包装在Matches()函数中，从而期望使用一元谓词函子。例如,

#include <vector>std::vector<int> v;…  
  
  
  
  
//在v中有多少元素>= 10?  
const int count = count\_if(v.begin()， v.end()， Matches(Ge(10));

由于您可以使用谷歌Mock轻松地从简单的匹配器构建复杂的匹配器，这为您提供了一种方便地构建复合谓词的方法(使用STL的<functional>标头进行相同的操作是非常痛苦的)。例如，这里有一个谓词，它可以由任何数字>= 0、<= 100和!= 50来满足:

匹配(所有(Ge (0) (100), Ne (50)))

## 在谷歌测试断言中使用匹配器

因为匹配器基本上是也知道如何描述自己的谓词，所以有一种方法可以在谷歌测试断言中利用它们。它被称为ASSERT\_THAT和EXPECT\_THAT:

ASSERT\_THAT(价值,匹配器);//断言该值与matcher匹配。  
EXPECT\_THAT(价值,匹配器);//非致命版本。

例如，在谷歌测试中，你可以这样写:

# include“gmock / gmock。使用::test::AllOf;使用::test::Ge;使用::test::Le;使用::test::MatchesRegex;使用::testing::StartsWith;…  
  
  
  
  
  
  
  
  
EXPECT\_THAT (Foo (), StartsWith("你好"));  
EXPECT\_THAT (Bar (), MatchesRegex(线\ \ d +));  
ASSERT\_THAT(巴兹(),所有(通用电气(5),(10)));

执行Foo()、Bar()和Baz()，并验证它:

* Foo()返回一个以“Hello”开头的字符串。
* Bar()返回一个与正则表达式“Line \\d+”匹配的字符串。
* Baz()返回一个范围为[5,10]的数字。

这些宏的好处是它们读起来像英语。它们也产生信息信息。例如，如果上面的第一个EXPECT\_THAT()失败，消息将类似于:

实际:“Hi, world!”  
  
预期:以“Hello”开头

**Credit: (ASSERT|EXPECT)\_THAT是从Hamcrest项目窃取的，它将assertThat()添加到JUnit。**

## 使用谓词作为匹配器

谷歌Mock提供了一组内置的匹配器。如果您发现它们缺少，您可以使用任意的一元谓词函数或functor作为匹配器—只要谓词接受您想要的类型的值。你可以通过在Truly()函数中封装谓词来做到这一点，例如:

使用::testing::Truly;int IsEven(int n) {return (n % 2) == 0 ?  
  
1:0;}……  
  
  
必须使用偶数调用// Bar()。  
EXPECT\_CALL (foo, Bar(真正(IsEven)));

注意，谓词函数/ functor不必返回bool。只要返回值可以用作语句中的条件(条件)…

## 匹配不可复制的参数

当您执行EXPECT\_CALL(mock\_obj, Foo(bar))时，谷歌Mock将保存bar的一个副本。当以后调用Foo()时，谷歌Mock会将与Foo()的参数与bar的保存副本进行比较。这样，您就不必担心在执行完EXPECT\_CALL()之后修改或销毁bar。当您使用Eq(bar)、Le(bar)等匹配器时也是如此。

但是如果bar不能被复制(即没有复制构造函数)呢?您可以定义自己的matcher函数，并将其与Truly()一起使用，如前面两个菜谱所示。或者，如果您可以保证在执行EXPECT\_CALL()之后bar不会被更改，那么您就可以避开它。只需告诉谷歌Mock它应该保存对bar的引用，而不是它的副本。方法如下:

使用::test::Eq;使用::testing::ByRef;使用::testing::Lt;  
  
  
  
//期望Foo()的参数== bar。  
时EXPECT\_CALL (mock\_obj Foo (Eq (ByRef (bar))));  
  
//期望Foo()的参数< bar。  
时EXPECT\_CALL (mock\_obj Foo (Lt (ByRef (bar))));

请记住:如果这样做，不要在EXPECT\_CALL()之后更改bar，否则结果是未定义的。

## 验证对象的成员

模拟函数通常以对象的引用作为参数。在匹配参数时，您可能不希望将整个对象与固定对象进行比较，因为这可能过于详细。相反，您可能需要验证某个成员变量或对象的某个getter方法的结果。您可以使用Field()和Property()实现这一点。更具体地说,

字段(foo:酒吧,m)

是一个匹配一个Foo对象的匹配器，该对象的bar成员变量满足匹配器m。

财产(foo::巴兹,m)

是一个匹配一个Foo对象的matcher，该对象的baz()方法返回一个满足matcher m的值。

例如:

|  |  |
| --- | --- |
| 字段(foo:数量,通用电气(3)) | 匹配x。数量> = 3。 |
| 属性(foo:名字,StartsWith(“约翰”)) | 匹配x，其中x.name()以“John”开头。 |

注意，在Property(&Foo::baz，…)中，baz()方法必须不带参数并声明为const。

顺便说一句，Field()和Property()也可以匹配对象的普通指针。例如,

字段(foo:数量,通用电气(3))

匹配一个普通指针p，其中p->编号>= 3。如果p为空，则无论内部匹配器如何，匹配都将失败。

如果您希望同时验证多个成员，该怎么办?记住，有AllOf()。

## 验证指针参数指向的值

c++函数通常采用指针作为参数。可以使用IsNull()、NotNull()等匹配器和其他比较匹配器来匹配指针，但是如果希望确保指针指向的值(而不是指针本身)具有某个属性，该怎么办呢?你可以使用Pointee(m)匹配器。

Pointee(m)匹配一个指针，如果m匹配指针指向的值。例如:

测试使用::::通用;使用::测试::指针数据;…  
  
  
EXPECT\_CALL (foo, Bar(指针数据(通用电气(3))));

期望使用指向大于或等于3的值的指针调用foo.Bar()。

Pointee()的一个好处是，它将NULL指针视为匹配失败，因此可以编写Pointee(m)而不是

所有(NotNull(),指针数据(m))

不用担心空指针会使您的测试崩溃。

另外，我们是否告诉过您Pointee()既能处理原始指针，也能处理智能指针(linked\_ptr、shared\_ptr、scoped\_ptr等)?

如果你有一个指向指针的指针呢?您猜对了——您可以使用嵌套的Pointee()深入探测值的内部。例如，Pointee(Pointee(Lt(3))匹配一个指针，该指针指向一个指向小于3的数字的指针(多么拗口……)。

## 测试对象的某个属性

有时，您希望指定一个对象参数具有某个属性，但是现有的matcher没有这样做。如果希望获得良好的错误消息，应该定义一个匹配器。如果你想又快又脏，你可以写一个普通的函数。

假设您有一个模拟函数，它接受一个Foo类型的对象，该对象有一个int bar()方法和一个int baz()方法，您希望约束参数的bar()值加上它的baz()值是一个给定的数字。你可以这样定义一个匹配器:

测试使用::::MatcherInterface;使用::测试::MatchResultListener;类BarPlusBazEqMatcher:公共MatcherInterface < const foo >{公众:明确BarPlusBazEqMatcher (int expected\_sum): expected\_sum\_ (expected\_sum){}虚拟bool MatchAndExplain (const foo foo, MatchResultListener \*侦听器)const{返回(foo.bar foo.baz () + () = = expected\_sum\_;  
  
  
  
  
  
  
  
  
  
  
{\*os < "bar() + baz() = " << expected\_sum\_;  
  
  
  
{\*os < "bar() + baz() is not equal " << expected\_sum\_;  
  
  
  
{return MakeMatcher(new BarPlusBazEqMatcher(expected\_sum));}…  
  
  
  
  
  
  
  
  
  
  
EXPECT\_CALL (…,这么做(BarPlusBazEq (5)))……;

## 匹配的容器

有时STL容器(例如list、vector、map、…)被传递给模拟函数，您可能需要验证它。由于大多数STL容器都支持==操作符，所以您可以编写Eq(expected\_container)或直接编写expected\_container来精确匹配容器。

不过，有时您可能希望更灵活一些(例如，第一个元素必须是精确匹配的，但是第二个元素可以是任何正数，等等)。此外，测试中使用的容器通常只有少量的元素，而且必须在行外定义预期的容器有点麻烦。

在这种情况下，你可以使用ElementsAre()或者UnorderedElementsAre() matcher:

使用::test::\_;使用::testing:: elementare;使用::testing::Gt;…  
  
  
  
  
MOCK\_METHOD1(Foo, void(const vector<int>& numbers));…  
  
  
EXPECT\_CALL(mock, Foo(ElementsAre(1, Gt(0)， \_， 5));

上面的matcher说容器必须有4个元素，分别是1、大于0、任意数和5。

如果你写:

使用::testing::\_;使用::testing::Gt;使用::testing:: unorderedelementare;…  
  
  
  
  
MOCK\_METHOD1(Foo, void(const vector<int>& numbers));…  
  
  
(unorderedelementare (1, Gt(0)， \_， 5));

这意味着容器必须有4个元素，它们在某些排列下必须分别是1、大于0、任何数和5。

ElementsAre()和UnorderedElementsAre()被重载以接受0到10个参数。如果需要更多的元素，你可以将它们放在一个c风格的数组中，并使用ElementsAreArray()或UnorderedElementsAreArray()来代替:

测试使用::::ElementsAreArray;…  
  
  
// ElementsAreArray接受一个元素值数组。  
const int expected\_vector1[] ={1, 5, 2, 4，…};  
EXPECT\_CALL(模拟Foo (ElementsAreArray (expected\_vector1)));  
  
//或者，一个元素匹配器数组。  
Matcher<int> expected\_vector2 = {1, Gt(2)， \_， 3，…};  
EXPECT\_CALL(模拟Foo (ElementsAreArray (expected\_vector2)));

如果需要动态创建数组(因此编译器无法推断数组的大小)，可以给ElementsAreArray()一个额外的参数来指定数组的大小:

测试使用::::ElementsAreArray;…  
  
int\* const expected\_vector3 =新int[count];  
…用值填充expected\_vector3…  
EXPECT\_CALL(模拟Foo (ElementsAreArray (expected\_vector3计数)));

**小贴士:**

* ElementsAre\*()可以用来匹配任何实现STL迭代器模式的容器(例如，它具有一个const\_iterator类型并支持begin()/end()，而不仅仅是STL中定义的那些。它甚至可以用于尚未编写的容器类型——只要它们遵循上述模式即可。
* 您可以使用嵌套的ElementsAre\*()来匹配嵌套的(多维的)容器。
* 如果容器是通过指针而不是引用来传递的，那么只需编写Pointee(ElementsAre\*(…))。
* 元素的顺序关系到元素是\*()。因此，不要将其用于元素顺序未定义的容器(例如hash\_map)。

## 分享匹配器

在底层，谷歌模拟matcher对象由一个指向ref计数的实现对象的指针组成。允许复制匹配器，而且非常有效，因为只复制指针。当引用实现对象的最后一个匹配器死亡时，将删除实现对象。

因此，如果您希望反复使用一些复杂的匹配器，则不需要每次都构建它。只需将其分配给matcher变量并重复使用该变量!例如,

Matcher<int> in\_range = AllOf(Gt(5)， Le(10));  
…在多个EXPECT\_CALLs中使用in\_range作为匹配器…

# 设定的期望

## 知道何时期待

ON\_CALL可能是谷歌Mock中利用率最低的一个构造。

基本上有两个构造用于定义模拟对象的行为:ON\_CALL和EXPECT\_CALL。区别呢?ON\_CALL定义了在调用模拟方法时发生的事情，但并不意味着对调用的方法有任何期望。EXPECT\_CALL不仅定义了行为，而且还设置了一个期望，即在给定次数(以及指定顺序时的给定顺序)的情况下，使用给定的参数调用方法。

因为EXPECT\_CALL做的更多，它不是比ON\_CALL更好吗?不是真的。每个EXPECT\_CALL都在被测试代码的行为上添加一个约束。拥有比需要更多的约束是baaad—甚至比没有足够的约束更糟糕。

这可能有违直觉。验证次数多的测试怎么会比验证次数少的测试更糟糕呢?验证不是测试的全部吗?

答案在于测试应该验证什么。**好的测试验证代码的契约。**如果测试过度指定，它就没有为实现留下足够的自由。因此，在不破坏契约的情况下更改实现(例如重构和优化)(这样做应该是完全可以的)会破坏这样的测试。然后，您必须花时间来修复它们，但在下一次更改实现时又会再次看到它们被破坏。

请记住，在一个测试中不必验证多个属性。实际上，在一个测试中只验证一件事是一种很好的方式。如果这样做，一个bug可能只会破坏一个或两个测试，而不是几十个(您更愿意调试哪种情况?)如果您还习惯于给测试提供描述性的名称来说明它们所验证的内容，那么您常常可以很容易地从测试日志本身猜出问题所在。

因此，在缺省情况下使用ON\_CALL，并且仅在您实际打算验证调用是否已发出时使用EXPECT\_CALL。例如，您可能在测试装置中有一组ON\_CALLs来设置同一组中所有测试共享的公共模拟行为，并在不同的TEST\_Fs中编写(几乎不同的)EXPECT\_CALLs来验证代码行为的不同方面。相比之下,每个测试有许多EXPECT\_CALLs的风格,这将导致测试更有弹性implementational变化(因此不太可能需要维护),使测试的意图更明显(所以他们更容易保持当你需要维护)。

如果你是烦恼的“无趣的模拟函数调用”消息打印当没有EXPECT\_CALL模拟方法被调用时,你可能使用一个NiceMock而不是压制所有此类信息为模拟对象,或为特定的方法抑制消息通过添加EXPECT\_CALL(…)同学(AnyNumber ())。不要盲目地添加EXPECT\_CALL(…)来抑制它，否则维护起来会很麻烦。

## 忽略无趣的电话

如果您对模拟方法的调用方式不感兴趣，那么就不要谈论它。在这种情况下，如果方法被调用，谷歌Mock将执行其默认操作，以允许测试程序继续运行。如果您对谷歌Mock所采取的默认操作不满意，可以使用DefaultValue<T>::Set()(稍后将在本文档中描述)或ON\_CALL()覆盖它。

请注意，一旦您表达了对特定模拟方法的兴趣(通过EXPECT\_CALL())，对它的所有调用必须匹配一些期望。如果这个函数被调用，但是参数不匹配任何EXPECT\_CALL()语句，那么它将是一个错误。

## 禁止意料之外的电话

如果根本不应该调用mock方法，那么明确地说:

测试使用::::\_;…  
  
EXPECT\_CALL (foo, Bar(\_))同学(0);

如果允许对该方法进行一些调用，但不允许对其他方法进行调用，那么只需列出所有预期的调用:

测试使用::::AnyNumber;使用::测试::Gt;…  
  
  
EXPECT\_CALL (foo, Bar (5));  
EXPECT\_CALL (foo, Bar (Gt(10)))同学(AnyNumber ());

对foo.Bar()的调用不匹配任何EXPECT\_CALL()语句都将是一个错误。

## 理解无趣和意外的调用

*在谷歌模拟中，无趣调用和意外调用是不同的概念。非常不同的。*

如果甚至没有一个EXPECT\_CALL(x, Y(…))集，那么调用x.Y(…)就没有意义了。

如果设置了一些EXPECT\_CALL(x, Y(…))，但是没有一个与调用匹配，则调用x.Y(…)是不可预料的。换句话说，测试对x.Y()方法感兴趣(因此它显式地设置了一些EXPECT\_CALL来验证它是如何被调用的);但是，验证失败了，因为测试并不期望这个特定的调用发生。

**意外调用总是一个错误，因为被测试的代码没有按照测试期望的方式运行。**

**默认情况下，无兴趣的调用不是错误，因为它没有违反测试指定的约束。**(谷歌Mock的哲学是，什么都不说意味着没有约束。)但是，它会导致一个警告，因为它可能指出一个问题(例如，测试作者可能忘记指定一个约束)。

在谷歌Mock中，可以使用NiceMock和StrictMock使模拟类“nice”或“strict”。这如何影响无趣调用和意外调用?

一个漂亮的mock会抑制无趣的调用警告。它没有默认mock那么多废话，但其他方面是一样的。如果一个测试使用默认mock失败，那么它也将使用一个漂亮的mock失败。反之亦然。不要期望通过模拟来更改测试结果。

严格的mock会将无趣的调用警告变成错误。因此，严格模拟可能会改变测试结果。

让我们来看一个例子:

{NiceMock<MockDomainRegistry> mock\_registry;  
  
GetDomainOwner("google.com")) . will(Return("Larry Page");  
  
  
//在测试代码中使用mock\_registry。  
&mock\_registry…}

这里唯一的EXPECT\_CALL表示对GetDomainOwner()的所有调用必须以“google.com”作为参数。如果GetDomainOwner("yahoo.com")被调用，它将是一个意外调用，因此会出现一个错误。一个好的mock不会改变意外调用的严重性。

那么，我们如何告诉谷歌Mock GetDomainOwner()也可以与其他一些参数一起调用呢?标准的技术是添加一个“catch all”EXPECT\_CALL:

EXPECT\_CALL (mock\_registry GetDomainOwner(\_))同学(AnyNumber ());  
//捕获对这个方法的所有其他调用。  
GetDomainOwner("google.com")) . will(Return("Larry Page");

记住\_是匹配任何内容的通配符。这样，如果GetDomainOwner("google.com")被调用，它将做第二个EXPECT\_CALL说;如果使用不同的参数调用它，它将执行第一个EXPECT\_CALL所表示的操作。

注意，两个EXPECT\_CALL的顺序很重要，因为新的EXPECT\_CALL优先于旧的EXPECT\_CALL。

要了解更多关于无趣调用、nice mock和strict mock的信息，请阅读“The nice, The strict, and The Naggy”。

## 期待命令调用

虽然前面定义的EXPECT\_CALL()语句在谷歌Mock尝试将函数调用与期望匹配时优先，但是在缺省情况下，调用不必按照编写EXPECT\_CALL()语句的顺序进行。例如，如果参数匹配第三个EXPECT\_CALL()中的匹配器，但是不匹配前两个中的匹配器，那么将使用第三个期望。

如果您希望所有调用都按照期望的顺序发生，那么可以将EXPECT\_CALL()语句放在一个块中，在这个块中定义一个InSequence类型的变量:

测试使用::::\_;  
测试使用::::InSequence;  
  
{InSequence年代;  
  
  
EXPECT\_CALL (foo,就是(5));  
EXPECT\_CALL(酒吧,DoThat(\_))同学(2);  
  
EXPECT\_CALL (foo,这么做(6));  
}

在本例中，我们希望调用foo.DoThis(5)，然后调用bar.DoThat()，其中参数可以是任何内容，然后调用foo.DoThis(6)。如果调用无序发生，谷歌Mock将报告错误。

## 期待部分有序呼叫

有时要求所有的事情都按照预定的顺序进行会导致脆性测试。例如，我们可能关心A在B和C之前发生，但是对B和C的相对顺序不感兴趣。在这种情况下，测试应该反映我们的真实意图，而不是过度的约束。

谷歌Mock允许对调用强制使用任意的DAG(有向无环图)。表达DAG的一种方法是使用EXPECT\_CALL的After子句。

另一种方法是通过InSequence()子句(与InSequence类不同)，它是从jMock 2中借来的。它不如After()灵活，但是当您有很长的顺序调用链时更方便，因为它不需要您为链中的期望提供不同的名称。它是这样工作的:

如果我们将EXPECT\_CALL()语句视为图中的节点，并将节点a到节点B之间的边添加到a必须出现在B之前的任何地方，我们就可以得到DAG。我们用“序列”这个术语来表示DAG中的有向路径。现在，如果我们将DAG分解为序列，我们只需要知道每个EXPECT\_CALL()属于哪个序列，以便能够重构原始DAG。

因此，要指定期望的部分顺序，我们需要做两件事:首先定义一些序列对象，然后为每个EXPECT\_CALL()指定它属于哪个序列对象。相同顺序的期望必须按照它们被写入的顺序出现。例如,

使用::测试::序列;  
  
s1, s2序列;  
  
.InSequence(s1, s2);  
  
EXPECT\_CALL(酒吧,B ()) .InSequence (s1);  
  
EXPECT\_CALL(酒吧,C ()) .InSequence (s2);  
  
EXPECT\_CALL (foo, D ()) .InSequence (s2);

指定以下DAG(其中s1为A -> B, s2为A ->C -> D):

+ - - - - - - > B | - - - | | + C - - - - - - - - - > > D

这意味着A必须出现在B和C之前，C必须出现在d之前。除了这些顺序没有限制。

## 控制期望何时消退

当调用模拟方法时，谷歌模拟只考虑仍然处于活动状态的期望。一个期望在创建时是活动的，当必须稍后发生的调用发生时，它就变成不活动的(也就是退休)。例如,在

测试使用::::\_;  
使用::测试::序列;  
  
s1, s2序列;  
  
// 1 . times (AnyNumber() . insequence (s1, s2);  
  
  
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =  
  
(日志，日志(警告，\_，“用户未找到”))// #3 . insequence (s2);

一旦#2或#3匹配上，#1就会退役。如果在此之后记录了一个警告“文件太大”，那么这将是一个错误。

请注意，当期望达到饱和时，它不会自动退役。例如,

测试使用::::\_;…  
  
EXPECT\_CALL(log, log(警告，\_，\_));// #1 EXPECT\_CALL(log, log(警告，\_，“文件太大”);  
/ / # 2

说将会有一个警告信息“文件太大”。  
如果第二个警告也包含此消息，#2将再次匹配，并导致违反上限的错误。

如果这不是你想要的，你可以要求你的期望在它饱和的时候就退休:

测试使用::::\_;…  
  
EXPECT\_CALL(log, log(警告，\_，\_));// #1 . EXPECT\_CALL(log, log(警告，\_，“文件太大”))// #2 . reti谐振饱和();

这里#2只能使用一次，所以如果您有两个警告消息“文件太大”。，第一个将匹配#2，第二个将匹配#1 -将没有错误。

# 使用行为

## 从模拟方法返回引用

如果模拟函数的返回类型是引用，则需要使用ReturnRef()而不是return()来返回结果:

{public: MOCK\_METHOD0(GetBar, Bar&());};  
  
  
  
  
  
  
  
MockFoo foo;  
酒吧酒吧;  
EXPECT\_CALL (foo, GetBar ()) .WillOnce (ReturnRef (bar));

## 从模拟方法返回活动值

Return(x)操作在创建操作时保存x的一个副本，并且在执行操作时总是返回相同的值。有时，您可能希望返回x的活动值(即在执行操作时它的值)。

如果模拟函数的返回类型是引用，可以使用ReturnRef(x)来实现，如前面的菜谱所示(“从模拟方法返回引用”)。但是，谷歌Mock不允许在返回类型不是引用的Mock函数中使用ReturnRef()，因为这样做通常会指出用户错误。那么，你该怎么办呢?

你可能想尝试ByRef():

{public: MOCK\_METHOD0(GetValue, int());};  
  
  
  
  
  
  
  
int x = 0;  
MockFoo foo;  
EXPECT\_CALL (foo, GetValue ()) .WillRepeatedly(返回时(ByRef (x)));  
  
x = 42;  
EXPECT\_EQ(42岁,foo.GetValue ());

不幸的是，这在这里行不通。以上代码将失败与错误:

getvalue()实际的:0Expected: 42

原因是Return(value)在创建操作时将值转换为模拟函数的实际返回类型，而不是在执行操作时。(当值是引用某些临时对象的代理对象时，为使操作安全，选择此行为。)因此，在设置期望时，ByRef(x)被转换为int值(而不是const int&)，而Return(ByRef(x))总是返回0。

ReturnPointee(pointer)是专门用来解决这个问题的。它返回执行动作时指针指向的值:

使用测试::ReturnPointee;…  
  
int x = 0;  
MockFoo foo;  
EXPECT\_CALL (foo, GetValue ()) .WillRepeatedly (ReturnPointee (x));  
//注意这里的&号。  
x = 42;  
EXPECT\_EQ(42岁,foo.GetValue ());//现在成功了。

## 结合行为

当一个函数被调用时，想要做不止一件事?这很好。DoAll()允许您每次都执行一系列操作。只使用序列中最后一个操作的返回值。

{public: MOCK\_METHOD1(Bar, bool(int n));};  
  
  
  
  
  
  
  
.WillOnce(DoAll(action\_1, action\_2，…  
  
  
  
action\_n));

## Mocking的副作用

有时，一个方法不是通过返回一个值来显示其效果，而是通过副作用来显示其效果。例如，它可能更改某个全局状态或修改输出参数。为了模拟副作用，通常你可以通过实现::testing::ActionInterface来定义你自己的动作。

如果您需要做的只是更改输出参数，那么内置的SetArgPointee()操作非常方便:

{public: MOCK\_METHOD2(Mutate, void(bool Mutate, int\* value));  
  
  
  
  
},…  
  
  
  
MockMutator突变;  
.WillOnce(SetArgPointee<1>(5));

在本例中，当调用mutator.Mutate()时，我们将把5赋给参数#1(基于0)所指向的int变量。

SetArgPointee()可以方便地创建传递给它的值的内部副本，这样就不需要将该值保存在作用域内。但是，这意味着该值必须有一个复制构造函数和赋值操作符。

如果mock方法也需要返回一个值，您可以使用DoAll()将SetArgPointee()与return()链接起来:

使用::testing:::;使用::testing::Return;使用::testing::SetArgPointee;类MockMutator: public Mutator {public:…  
  
  
  
  
  
  
MOCK\_METHOD1 (MutateInt bool (int \*值));},…  
  
  
  
MockMutator突变;  
.WillOnce(DoAll(SetArgPointee<0>(5)， Return(true));

如果输出参数是一个数组，则使用SetArrayArgument<N>(first, last)操作。它将源范围内的元素[first, last]复制到第n个(基于0的)参数指向的数组中:

{public: MOCK\_METHOD2(Mutate, void(int\* values, int num\_values));  
  
  
  
  
  
},…  
  
  
  
MockArrayMutator突变;  
int值[5]= {1,2,3,4,5};  
.WillOnce(SetArrayArgument<0>(values, values + 5));

这也适用于参数是输出迭代器:

{public: MOCK\_METHOD1(GetNames, void(std::back\_insert\_iterator<vector<string> >));  
  
  
  
  
  
},…  
  
  
  
MockRolodex名片盒;  
向量<字符串>名称;  
names.push\_back(“乔治”);  
names.push\_back(“约翰”);  
names.push\_back(“托马斯”);  
(1) .WillOnce(SetArrayArgument<0>(names.begin()， names.end());

## 根据状态更改模拟对象的行为

如果您希望调用更改模拟对象的行为，可以使用::testing::InSequence来指定调用之前和之后的不同行为:

测试使用::::InSequence;使用::测试::返回;…  
  
  
  
{InSequence seq;  
  
EXPECT\_CALL (my\_mock IsDirty ()) .WillRepeatedly(返回(真正的));  
  
EXPECT\_CALL (my\_mock冲洗());  
EXPECT\_CALL (my\_mock IsDirty ()) .WillRepeatedly(返回(false));  
  
}my\_mock.FlushIfDirty ();

这使得my\_mock.IsDirty()在调用my\_mock.Flush()之前返回true，然后返回false。

如果行为变化更复杂，你可以把效果存储在一个变量中，让一个模拟方法从这个变量中获得它的返回值:

使用::testing:::\_;使用::testing::SaveArg;使用::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 ();

在这里，my\_mock.GetPrevValue()将始终返回最后一次UpdateValue()调用的参数。

## 设置返回类型的默认值

如果模拟方法的返回类型是内置的c++类型或指针，那么在缺省情况下，它在调用时将返回0。同样，在c++ 11及以上版本中，如果一个mock方法的返回类型有一个默认构造函数，那么它将默认返回一个默认构造的值。如果这个默认值不适合您，您只需要指定一个操作。

有时，您可能希望更改这个默认值，或者您可能希望为谷歌Mock不知道的类型指定一个默认值。可以使用::testing::DefaultValue类模板:

{public: MOCK\_METHOD0(CalculateBar, Bar());};…  
  
  
  
  
  
酒吧default\_bar;  
设置类型栏的默认返回值。  
DefaultValue <栏>::设置(default\_bar);  
  
MockFoo foo;  
  
//我们不需要在这里指定操作，因为默认的//返回值对我们有用。  
  
EXPECT\_CALL (foo, CalculateBar ());  
  
foo.CalculateBar ();//这个应该返回default\_bar。  
  
//取消默认返回值。  
DefaultValue <栏>::Clear ();

请注意，更改类型的默认值会使您的测试难以理解。我们建议您明智地使用这个特性。例如，您可能希望确保Set()和Clear()调用紧挨着使用mock的代码。

## 为模拟方法设置默认操作

您已经了解了如何更改给定类型的默认值。但是，对于您的目的来说，这可能太粗糙了:可能您有两个具有相同返回类型的模拟方法，您希望它们具有不同的行为。ON\_CALL()宏允许您在方法级别自定义模拟的行为:

使用::testing:::\_;使用::testing::AnyNumber;使用::testing::Gt;使用::testing::Return;…  
  
  
  
  
ON\_CALL (foo,符号(\_)).WillByDefault(返回(1));  
  
ON\_CALL (foo,符号(0)).WillByDefault(返回(0));  
  
ON\_CALL (foo,标志(Gt (0))) .WillByDefault(返回(1));  
  
  
EXPECT\_CALL (foo,符号(\_))同学(AnyNumber ());  
  
  
foo.Sign (5);//这个应该返回1。  
foo.Sign (9);//这个应该返回-1。  
foo.Sign (0);//这个应该返回0。

您可能已经猜到，当有多个ON\_CALL()语句时，新闻顺序优先于旧的顺序。换句话说，将使用与函数参数匹配的最后一个参数。这种匹配顺序允许您在模拟对象的构造函数或测试装置的设置阶段中设置公共行为，并在稍后专门化模拟的行为。

## 使用函数/方法/函数作为操作

如果内置的动作不适合你，你可以很容易地使用一个现有的功能，方法，或函子作为一个动作:

{public: MOCK\_METHOD2(Sum, int(int x, int y));  
  
  
  
  
  
{1}; {2}; {3};  
  
  
{public: bool ComplexJob(int x);};…  
  
  
  
  
  
  
  
MockFoo foo;  
助手辅助;  
.WillOnce(Invoke(CalculateSum));  
  
.WillOnce(Invoke(&helper， &helper::ComplexJob));  
  
  
foo。和(5、6);//调用CalculateSum(5,6). foo.ComplexJob(10);  
/ /调用helper.ComplexJob (10);

唯一的要求就是,函数的类型,必须兼容模拟的签名等功能,这意味着后者的参数可以隐式转换为相应的参数前,和前的返回类型可以隐式转换为后者。所以，您可以调用与mock函数类型不完全相同的函数，只要这样做是安全的—很好，不是吗?

## 调用没有参数的函数/方法/函数

Invoke()对于执行更复杂的操作非常有用。它将模拟函数的参数传递给被调用的函数或functor，以便被调用方拥有要处理的调用的完整上下文。如果被调用的函数对某些或所有参数不感兴趣，它可以简单地忽略它们。

然而，一种常见的模式是，测试作者希望调用没有模拟函数参数的函数。Invoke()允许她使用一个包装器函数来实现这一点，该函数在调用下划线的空函数之前丢弃参数。不用说，这可能是乏味的，并且模糊了测试的意图。

InvokeWithoutArgs()解决了这个问题。它与Invoke()类似，只是没有将模拟函数的参数传递给被调用方。这里有一个例子:

{public: MOCK\_METHOD1(ComplexJob, bool(int n));};bool Job1(){…  
  
  
  
  
  
  
  
}……  
  
  
MockFoo foo;  
EXPECT\_CALL (foo, ComplexJob (\_)) .WillOnce (InvokeWithoutArgs (Job1));  
  
  
foo.ComplexJob (10);/ /调用Job1 ()。

## 调用模拟函数的参数

有时，模拟函数将接收一个函数指针或一个函子(换句话说，一个“可调用的”)作为参数，例如。

{public: MOCK\_METHOD2(DoThis, bool(int n, bool(\*fp)(int)));};

你可能想要调用这个可调用的参数:

测试使用::::\_;…  
  
MockFoo foo;  
EXPECT\_CALL (foo,这么做(\_,\_)).WillOnce (…);  
  
//将执行(\*fp)(5)，其中fp是DoThis()接收到的//第二个参数。

你需要引用一个模拟函数的参数，但是c++还没有lambda，所以你必须定义你自己的动作。-真的吗?

谷歌Mock有一个动作可以解决这个问题:

InvokeArgument < N > (arg\_1 arg\_2,……arg\_m)

将使用arg\_1、arg\_2、…,arg\_m。无论参数是函数指针还是仿函数，谷歌Mock都会处理它们。

你可以这样写:

测试使用::::\_;使用::测试::InvokeArgument;…  
  
  
.WillOnce(InvokeArgument<1>(5));  
  
//将执行(\*fp)(5)，其中fp是DoThis()接收到的//第二个参数。

如果callable引用了一个参数怎么办?没有问题-只要把它包装在ByRef():

…  
MOCK\_METHOD1(Bar, bool(bool (\*fp)(int, const helper&)));…  
  
使用::testing:::\_;使用::testing::ByRef;使用::testing::InvokeArgument;…  
  
  
  
  
MockFoo foo;  
助手辅助;  
…  
.WillOnce(InvokeArgument<0>(5, ByRef(helper)));  
  
// ByRef(helper)保证将一个对helper的引用(而不是它的副本)传递给可调用对象。

如果callable通过引用获取一个参数，而我们没有在ByRef()中封装该参数，该怎么办?然后InvokeArgument()将创建该参数的一个副本，并将一个引用传递给该副本，而不是将一个对原始值的引用传递给callable。这在参数是临时值时特别方便:

…  
MOCK\_METHOD1(DoThat, bool(bool (\*f)(const double&x, const string&s));  
  
测试使用::::\_;使用::测试::InvokeArgument;…  
  
  
  
MockFoo foo;  
…  
.WillOnce(InvokeArgument<0>(5.0, string("Hi"));  
  
//将执行(\*f)(5.0, string("Hi"))，其中f是函数指针// DoThat()接收到的。  
注意，当EXPECT\_CALL()语句结束时，值5.0和字符串(“Hi”)是//临时的和死的。  
但是//可以稍后执行这个操作，因为值//的副本保存在InvokeArgument操作中。

## 忽略操作的结果

有时您有一个返回某些内容的操作，但是您需要一个返回void的操作(可能您想在一个返回void的模拟函数中使用它，或者可能需要在DoAll()中使用它，而它不是列表中的最后一个)。IgnoreResult()允许您这样做。例如:

使用::testing::\_;使用::testing::Invoke;使用::testing::Return;  
  
  
  
  
  
  
  
  
  
MOCK\_METHOD0 (Xyz, bool ());},…  
  
  
  
MockFoo foo;  
// .WillOnce(Invoke(Process));  
  
//上面的代码不会编译为Process()返回int，但是Abc()需要//返回void. . willonce (IgnoreResult(Invoke(Process));  
  
  
  
. willonce (DoAll(IgnoreResult(Invoke(DoSomething))， //忽略字符串DoSomething()返回。  
  
  
返回(真正的)));

注意，您不能对已经返回void的操作使用IgnoreResult()。这样做将导致严重的编译错误。

## 选择一个动作的参数

假设您有一个模拟函数Foo()，该函数有7个参数，并且您有一个自定义操作，希望在调用Foo()时调用它。问题是，自定义动作只需要三个参数:

测试使用::::\_;使用::测试::调用;…  
  
  
MOCK\_METHOD7(Foo, bool(bool visible, const string& name, int x, int y, const map<pair<int, int>， double>& weight, double min\_weight, double max\_wight));  
  
  
  
  
bool IsVisibleInQuadrant1(bool visible, int x, int y) {return visible && x >= 0 & y >= 0;}…  
  
  
  
  
方法如下:= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =  
//呃，无法编译。:- (

为了取悦编译器之神，你可以定义一个“适配器”，它与Foo()具有相同的签名，并使用正确的参数调用自定义动作:

bool MyIsVisibleInQuadrant1(bool visible, const string& name, int x, int y, const map<pair<int, int>， double>& weight, double min\_weight, double max\_wight) {return IsVisibleInQuadrant1(visible, x, y);}…  
  
  
  
  
  
  
  
  
  
(= =) .WillOnce(Invoke(MyIsVisibleInQuadrant1));  
//现在成功了。

但这不是很尴尬吗?

谷歌Mock提供了一个通用的动作适配器，因此您可以将时间花在更重要的业务上，而不是编写自己的适配器。这里的语法:

WithArgs < N1、N2…Nk >(行动)

创建一个操作，该操作在给定索引(基于0)处将模拟函数的参数传递给内部操作并执行它。使用WithArgs，我们的原始示例可以写成:

使用::testing:::\_;使用::testing::Invoke;使用::testing::WithArgs;…  
  
  
  
.WillOnce(WithArgs<0, 2, 3>(Invoke(IsVisibleInQuadrant1));  
  
//不需要定义自己的适配器。

为了更好的可读性，谷歌Mock还提供:

* 当内部动作没有参数时，则为
* WithArg<N>(action)(在Arg之后没有s)当内部动作需要一个参数时。

您可能已经意识到，InvokeWithoutArgs(…)只是WithoutArgs(Invoke(…))的语法糖。

这里有更多的提示:

* WithArgs和friends中使用的内部操作不一定是Invoke()——它可以是任何东西。
* 如果需要，你可以在参数列表中重复一个参数，例如WithArgs< 2,3,3,5 >(…)。
* 你可以改变参数的顺序，例如WithArgs< 3,2,1 >(…)。
* 所选参数的类型不必与内部操作的签名完全匹配。只要它们可以隐式地转换为内部操作的相应参数，就可以工作。例如，如果mock函数的第4个参数是int，并且my\_action使用双精度，那么WithArg<4>(my\_action)将起作用。

## 忽略动作函数中的参数

select -an-action -arguments方法向我们展示了一种方法，可以使模拟函数和具有不兼容参数列表的操作结合在一起。缺点是，用WithArgs<…

如果您正在定义一个函数、方法或functor来与Invoke\*()一起使用，并且您对它的一些参数不感兴趣，那么可以使用WithArgs的另一种方法是将不感兴趣的参数声明为未使用。这使得定义不那么混乱和脆弱，以防不感兴趣的参数类型发生变化。它还可以增加动作函数被重用的机会。例如,给定

MOCK\_METHOD3(Foo, double(const string&label, double x, double y));  
MOCK\_METHOD3(Bar, double(int index, double x, double y));

而不是

使用::testing::\_;使用::testing::Invoke;double DistanceToOriginWithLabel(const string&label, double x, double y) {return sqrt(x\*x + y\*y);}  
  
  
  
  
  
  
  
  
  
  
  
EXEPCT\_CALL(mock, Foo("abc"， \_， \_)) .WillOnce(Invoke(DistanceToOriginWithLabel));  
  
EXEPCT\_CALL(mock, Bar(5， \_， \_)) .WillOnce(Invoke(DistanceToOriginWithIndex));

您可以编写

使用::testing::\_;使用::testing::Invoke;使用::testing::  
  
  
  
  
  
  
  
  
EXEPCT\_CALL(mock, Foo("abc"， \_， \_)) .WillOnce(Invoke(DistanceToOrigin));  
  
EXEPCT\_CALL(mock, Bar(5， \_， \_)) .WillOnce(Invoke(DistanceToOrigin));

## 分享行为

与匹配器一样，谷歌模拟操作对象由一个指针组成，该指针指向一个ref计数的实现对象。因此，复制操作也是允许的，而且非常有效。当引用实现对象的最后一个操作终止时，将删除实现对象。

如果您有一些需要反复使用的复杂操作，则不必每次都从头构建。如果动作没有内部状态(例如，不管它被调用多少次，它总是做同样的事情)，您可以将它分配给一个动作变量，并重复使用该变量。例如:

动作<bool(int\*)> set\_flag = DoAll(SetArgPointee<0>(5)， Return(true));  
  
…在.WillOnce()和. will()中使用set\_flag…

但是，如果操作有自己的状态，那么如果共享操作对象，您可能会感到惊讶。假设您有一个action factory IncrementCounter(init)，它创建一个递增并返回初始值为init的计数器的操作，使用从同一个表达式创建的两个操作和使用共享操作将执行不同的行为。例子:

EXPECT\_CALL (foo,这么做()).WillRepeatedly (IncrementCounter (0));  
  
EXPECT\_CALL (foo, DoThat ()) .WillRepeatedly (IncrementCounter (0));  
  
foo.DoThis ();/ /返回1。  
foo.DoThis ();/ /返回2。  
foo.DoThat ();//返回1 - Blah()使用与Bar()不同的//计数器。

与

动作<int()> increment = IncrementCounter(0);  
  
EXPECT\_CALL (foo,这么做()).WillRepeatedly(增量);  
  
EXPECT\_CALL (foo, DoThat ()) .WillRepeatedly(增量);  
  
foo.DoThis ();/ /返回1。  
foo.DoThis ();/ /返回2。  
foo.DoThat ();//返回3 -共享计数器。

# 关于使用谷歌模拟的杂项菜谱

## 使用纯移动类型的模拟方法

c++ 11引入了纯移动类型。只能移动类型的值可以从一个对象移动到另一个对象，但不能复制。unique\_ptr<T>可能是最常用的纯移动类型。

模拟一个只接受和/或返回移动类型的方法会带来一些挑战，但是没有什么是不可克服的。这个食谱告诉你怎么做。

假设我们正在进行一个虚构的项目，该项目允许发布和共享名为“buzzes”的代码片段。你的代码使用这些类型:

类Buzzer {public: virtual ~Buzzer(){}虚拟std::unique\_ptr<Buzz> MakeBuzz(const std:: string&text) = 0;  
  
  
  
  
  
  
  
  
  
  
  
虚拟bool ShareBuzz(std::unique\_ptr<Buzz> Buzz, Time timestamp) = 0;  
…};

Buzz对象表示要发布的代码段。实现Buzzer接口的类能够创建和共享Buzz。Buzzer中的方法可以返回一个unique\_ptr<Buzz>或取一个unique\_ptr<Buzz>。现在我们需要在测试中模拟Buzzer。

要模拟一个只返回移动类型的方法，只需像往常一样使用熟悉的MOCK\_METHOD语法:

类MockBuzzer: public Buzzer {public: MOCK\_METHOD1(MakeBuzz, std::unique\_ptr<Buzz>(const std:: string&text));  
  
  
…};

然而，如果你尝试使用相同的MOCK\_METHOD模式来模拟一个只接受移动参数的方法，你会得到一个编译器错误:

//不编译!  
MOCK\_METHOD2(ShareBuzz, bool(std::unique\_ptr<Buzz> Buzz, Time timestamp));

虽然让这种语法能够正常工作是非常可取的，但它并不是微不足道的，而且工作还没有完成。幸运的是，现在有一个小技巧可以让你得到几乎和这个一样的效果。

诀窍在于，将ShareBuzz()方法委托给一个模拟方法(让我们称它为DoShareBuzz())，它不接受只移动的参数:

类MockBuzzer: public Buzzer {public: MOCK\_METHOD1(MakeBuzz, std::unique\_ptr<Buzz>(const std:: string&text));  
  
  
MOCK\_METHOD2(DoShareBuzz, bool(Buzz\* Buzz, Time timestamp));  
bool ShareBuzz(std::unique\_ptr<Buzz> Buzz, Time timestamp){返回DoShareBuzz(Buzz .get()， timestamp);  
  
}};

注意，不需要在基类中定义或声明DoShareBuzz()。您只需要将它定义为mock类中的MOCK\_METHOD。

既然我们已经定义了模拟类，我们可以在测试中使用它。在下面的代码示例中，我们假设已经定义了一个名为mock\_buzzer\_的MockBuzzer对象:

MockBuzzer mock\_buzzer\_;

首先，让我们看看如何设置MakeBuzz()方法的期望，它返回一个unique\_ptr<Buzz>。

通常，如果您设置了一个没有操作的期望(即. willonce()或. willrepeat()子句)，当该期望触发时，将采取该方法的默认操作。因为unique\_ptr<>有一个默认的构造函数，返回一个空的unique\_ptr，如果你不指定一个动作，你会得到:

//使用默认操作。  
EXPECT\_CALL (mock\_buzzer\_ MakeBuzz("你好"));  
  
触发前面的EXPECT\_CALL。  
EXPECT\_EQ (nullptr mock\_buzzer\_.MakeBuzz("你好"));

如果你对默认的操作不满意，你可以调整它。根据需要，可以使用ON\_CALL()调整特定(模拟对象、模拟方法)组合的默认操作，也可以调整返回特定类型的所有模拟方法的默认操作。ON\_CALL()的用法类似于EXPECT\_CALL()，因此我们将跳过它，只解释如何实现后者(为特定的返回类型调整默认操作)。通过DefaultValue<>::SetFactory()和DefaultValue<>::Clear() API:

//将返回类型std::unique\_ptr<Buzz>的默认操作设置为//每次创建一个新的Buzz。  
  
DefaultValue<std::unique\_ptr<Buzz>>::SetFactory([] {return MakeUnique<Buzz>(AccessLevel::kInternal);  
});  
  
//当这个触发时，MakeBuzz()的默认操作将运行，该操作将返回一个新的Buzz对象。  
  
EXPECT\_CALL (mock\_buzzer\_ MakeBuzz (“hello”))同学(AnyNumber ());  
  
autobuzz1 = mock\_buzzer\_.MakeBuzz("hello");  
autobuzz2 = mock\_buzzer\_.MakeBuzz("hello");  
EXPECT\_NE (nullptr buzz1);  
EXPECT\_NE (nullptr buzz2);  
EXPECT\_NE (buzz1 buzz2);  
  
//重置返回类型std::unique\_ptr<Buzz>的默认操作，//以避免干扰其他测试。  
  
DefaultValue < std:: unique\_ptr < Buzz > >::明确();

如果希望该方法执行其他操作而不是默认操作，该怎么办?如果你只需要返回一个预定义的纯移动值，你可以使用return (ByMove(…))操作:

//当这个触发时，将返回由ByMove(…)指定的unique\_ptr<>。  
  
EXPECT\_CALL (mock\_buzzer\_ MakeBuzz(“世界”)).WillOnce(返回(ByMove (MakeUnique < Buzz > (AccessLevel:: kInternal))));  
  
  
EXPECT\_NE (nullptr mock\_buzzer\_.MakeBuzz("世界"));

注意，ByMove()在这里很重要——如果删除它，代码将无法编译。

测试时间!你认为如果一个返回(ByMove(…))操作被执行不止一次(例如，你写…will repeat (Return(ByMove(…));)会发生什么?请考虑一下，在第一次运行操作之后，源值将被消耗(因为它是一个只移动的值)，所以下一次将没有可以移动的值——您将得到一个运行时错误，Return(ByMove(…))只能运行一次。

如果你需要你的mock方法做的不仅仅是移动一个预定义的值，记住你总是可以使用Invoke()来调用一个lambda或一个可调用的对象，它可以做几乎任何你想做的事情:

. willrepeat (Invoke([](const std:: string&text) {return std::make\_unique<Buzz>(AccessLevel::kInternal);  
  
  
}));  
  
EXPECT\_NE (nullptr mock\_buzzer\_.MakeBuzz (" x "));  
EXPECT\_NE (nullptr mock\_buzzer\_.MakeBuzz (" x "));

每次这个EXPECT\_CALL触发时，都会创建并返回一个新的unique\_ptr<Buzz>。对于Return(ByMove(…))，您不能这样做。

现在有一个主题我们还没有涉及:如何设置ShareBuzz()的期望，它接受一个移动唯一类型的参数?答案是你不知道。相反，您可以对DoShareBuzz()模拟方法设置期望(记住，我们为DoShareBuzz()定义了MOCK\_METHOD，而不是ShareBuzz()):

EXPECT\_CALL (mock\_buzzer\_ DoShareBuzz (NotNull () \_));  
  
//当你像这样在MockBuzzer上调用ShareBuzz()时，调用会被//转发到DoShareBuzz()，它会被模拟。  
因此，这个语句//将触发上面的EXPECT\_CALL。  
  
mock\_buzzer\_.ShareBuzz (MakeUnique&lt; Buzz&gt; (AccessLevel:: kInternal)::基础::现在());

有些人可能已经发现了这种方法的一个问题:DoShareBuzz()模拟方法与真正的ShareBuzz()方法的区别在于它不能获得buzz参数的所有权——ShareBuzz()总是在DoShareBuzz()返回后删除buzz。如果您需要将buzz对象保存到某个地方，以便在调用ShareBuzz()时使用，该怎么办?事实上，你会被困住。

我们遇到的DoShareBuzz()的另一个问题是，它会让阅读或维护测试的人感到惊讶，因为人们会认为DoShareBuzz()(逻辑上)与ShareBuzz()有相同的契约。

幸运的是，这些问题可以通过更多的代码得到解决。让我们试着做对这一次:

{public: MockBuzzer(){//既然DoShareBuzz(buzz, time)应该获得// buzz的所有权，那么为DoShareBuzz(buzz, time)定义一个默认的行为来删除buzz。  
  
  
  
  
  
ON\_CALL(\*this, DoShareBuzz(\_， \_)) .WillByDefault(Invoke([](Buzz\* Buzz, Time timestamp))删除Buzz;  
  
  
返回true;  
}));  
} MOCK\_METHOD1(MakeBuzz, std::unique\_ptr<Buzz>(const std:: string&text));  
  
  
  
//拥有buzz。  
MOCK\_METHOD2(DoShareBuzz, bool(Buzz\* Buzz, Time timestamp));  
bool ShareBuzz(std::unique\_ptr<Buzz> Buzz, Time timestamp){返回DoShareBuzz(Buzz .release()， timestamp);  
  
}};

现在，mock DoShareBuzz()方法可以免费保存buzz参数供以后使用，如果您希望这样:

std:: unique\_ptr < Buzz > intercepted\_buzz;  
. willonce (Invoke([&intercepted\_buzz](Buzz\* Buzz, Time timestamp)){//将Buzz保存在intercepted\_buzz中以供以后分析。  
  
  
intercepted\_buzz.reset (buzz);  
返回错误;  
}));  
  
mock\_buzzer\_.ShareBuzz (std:: make\_unique < Buzz > (AccessLevel:: kInternal)、现在());  
  
EXPECT\_NE (nullptr intercepted\_buzz);

使用本配方中介绍的技巧，您现在可以模拟那些只接受和/或返回移动类型的方法。充分利用新获得的能力——在设计新API时，现在可以适当地使用unique\_ptrs，而不必担心这样做会影响测试。

## 加快编译速度

信不信由你，编译一个模拟类的绝大多数时间都花在生成它的构造函数和析构函数上，因为它们执行的是一些重要的任务(例如验证期望)。而且，具有不同签名的模拟方法具有不同的类型，因此编译器需要分别生成它们的构造函数/析构函数。因此，如果您模拟许多不同类型的方法，那么编译模拟类会变得非常慢。

如果编译缓慢，可以将模拟类的构造函数和析构函数的定义移出类体，并移到.cpp文件中。这样，即使您将模拟类包含在N个文件中，编译器也只需要生成一次构造函数和析构函数，从而大大加快了编译速度。

让我们用一个例子来说明这个想法。在应用这个配方之前，下面是模拟类的定义:

/ /文件mock\_foo.h ....  
  
{public: //由于我们没有声明构造函数或析构函数，编译器将在使用这个模拟类的每个翻译单元中生成它们。  
  
  
  
  
  
MOCK\_METHOD0(这么做,int ());  
MOCK\_METHOD1(DoThat, bool(const char\* str));  
…更多的模拟方法…};

更改之后，它看起来像:

/ /文件mock\_foo.h ....  
  
这里声明了构造函数和析构函数，但没有定义它们。  
  
  
MockFoo ();  
虚拟~ MockFoo ();  
  
MOCK\_METHOD0(这么做,int ());  
MOCK\_METHOD1(DoThat, bool(const char\* str));  
…更多的模拟方法…};

和

/ /文件mock\_foo.cpp。  
# include“路径/ / mock\_foo。这些定义可能看起来很简单，但是这些函数实际上是通过成员//变量的构造函数/析构函数来实现模拟方法的。

## 迫使一个验证

当它被删除时，友好的模拟对象将自动验证对它的所有期望都已满足，如果没有满足，将生成谷歌测试失败。这很方便，因为它让你少了一件事要担心。也就是说，除非您不确定是否要删除模拟对象。

为什么模拟对象最终不会被销毁?它可能是在堆上创建的，并由您正在测试的代码拥有。假设代码中有一个bug，并且它没有正确地删除模拟对象—当实际上有一个bug时，您可以通过测试。

使用堆检查器是一个好主意，可以缓解这个问题，但是它的实现可能不是100%可靠的。因此，有时您确实希望在销毁模拟对象之前强制谷歌Mock验证它。你可以用Mock:: verifyandclearexpecations (&mock\_object):

TEST(MyServerTest, ProcessesRequest){使用::testing::Mock;  
  
  
\* const foo = new MockFoo;  
EXPECT\_CALL (\* foo,……;  
/ /……其他期望……  
  
// server现在拥有foo。  
MyServer服务器(foo);  
server.ProcessRequest (…);  
  
//如果服务器的析构函数忘记删除foo， //这将验证期望。  
  
Mock:: verifyandclearexpecations (foo);} //服务器在超出这里的作用域时被销毁。

**提示:Mock:: verifyandclearexpections()函数返回一个bool来指示验证是否成功(yes为真)，因此，如果在验证失败时没有进一步的理由，您可以将该函数调用封装在ASSERT\_TRUE()中。**

## 使用止点

有时，您可能希望在测试中的各个检查点“重置”模拟对象:在每个检查点，您验证模拟对象上的所有现有期望都已得到满足，然后对其设置一些新的期望，就好像它是新创建的一样。这允许您在“阶段”中使用大小都是可管理的模拟对象。

其中一种情况是，在测试的SetUp()函数中，您可能希望在模拟对象的帮助下将测试的对象置于某种状态。一旦进入了期望的状态，就需要清除模拟上的所有期望，这样在TEST\_F主体中就可以对它设置新的期望。

您可能已经知道，我们在前面的配方中看到的Mock:: verifyandclearexpecations()函数可以帮助您解决这个问题。或者，如果您使用ON\_CALL()来设置模拟对象上的默认操作，并且也想清除默认操作，那么可以使用mock::VerifyAndClear(&mock\_object)。这个函数执行Mock:: verifyandclearexpections (&mock\_object)执行的操作，并返回相同的bool，另外它还清除mock\_object上的ON\_CALL()语句。

为了达到同样的效果，您可以使用的另一个技巧是将期望放入序列中，并在特定位置插入对虚拟“检查点”函数的调用。然后可以验证模拟函数调用确实在正确的时间发生。例如，如果你正在练习代码:

Foo (1)、Foo (2) Foo (3);

验证Foo(1)和Foo(3)都调用了mock.Bar("a")，但Foo(2)没有调用任何东西。你可以写:

使用::testing::MockFunction;  
  
  
  
类MockFunction<F>只有一个模拟方法。它名为// Call()，类型为F. MockFunction<void(string check\_point\_name)> check;  
  
  
{InSequence年代;  
  
  
EXPECT\_CALL(模拟、酒吧(a));  
EXPECT\_CALL(检查,调用(" 1 "));  
EXPECT\_CALL(检查,调用(“2”));  
EXPECT\_CALL(模拟、酒吧(a));  
}Foo (1);  
  
check.Call (" 1 ");  
Foo (2);  
check.Call (“2”);  
Foo (3);}

期望规范说第一个条(“a”)必须在检查点“1”之前发生，第二个条(“a”)必须在检查点“2”之后发生，两个检查点之间不应该发生任何事情。显式的检查点使得判断哪个调用Foo()调用了哪个Bar(“a”)变得很容易。

## Mocking析构函数

有时您希望确保在正确的时间析构模拟对象，例如在调用bar-> a()之后，而在调用bar->B()之前。我们已经知道可以指定模拟函数调用顺序的约束，所以我们需要做的就是模拟模拟函数的析构函数。

这听起来很简单，除了一个问题:析构函数是一个特殊的函数，它有特殊的语法和特殊的语义，MOCK\_METHOD0宏对它不起作用:

MOCK\_METHOD0 (~ MockFoo,空白());/ /不会编译!

好消息是，您可以使用一个简单的模式来实现相同的效果。首先，将模拟函数Die()添加到模拟类中，并在析构函数中调用它，如下所示:

类MockFoo: public Foo{…  
  
//在模拟类中添加以下两行。  
MOCK\_METHOD0(死,无效());  
虚拟~MockFoo() {Die();}};

(如果名称Die()与现有符号冲突，则选择另一个名称。)现在，我们已经将MockFoo对象死亡时的测试问题转化为调用其Die()方法时的测试:

MockFoo\* foo = new MockFoo;  
MockBar\* bar =新建MockBar;  
…  
{InSequence年代;  
  
  
//期望\*foo在bar->A()之后，bar->B()之前死亡。  
EXPECT\_CALL(\*酒吧,());  
EXPECT\_CALL (\* foo, Die ());  
EXPECT\_CALL(\*酒吧,B ());  
}

那就是了。

## 使用谷歌模拟和线程

**重要提示:我们在此配方中描述的仅适用于谷歌Mock是线程安全的平台。**目前，这些只是支持pthreads库的平台(包括Linux和Mac)。为了使它在其他平台上是线程安全的，我们只需要在“gtest/internal/gtest-port.h”中实现一些同步操作。

在单元测试中，最好能够在单线程上下文中隔离和测试一段代码。这避免了竞争条件和死锁，并使调试测试更加容易。

然而，许多程序是多线程的，有时为了测试某些东西，我们需要从多个线程对其进行反复测试。谷歌Mock也适用于此目的。

记住使用mock的步骤:

1. 创建一个模拟对象foo。
2. 使用ON\_CALL()和EXPECT\_CALL()设置它的默认操作和期望。
3. 被测试的代码调用foo的方法。
4. 可以选择验证和重置模拟。
5. 自己销毁mock，或者让测试中的代码销毁它。析构函数会自动验证它。

如果遵循以下简单的规则，那么模拟和线程就可以和谐地共存:

* 在一个线程中执行测试代码(与被测试的代码相反)。这使您的测试易于遵循。
* 显然，您可以在不锁定的情况下执行步骤1。
* 在执行第2步和第5步时，确保没有其他线程正在访问foo。明显的,是吧?

# 3和#4可以在一个线程中完成，也可以在多个线程中完成——随你喜欢。谷歌Mock负责锁定，所以您不必执行任何操作—除非您的测试逻辑需要这样做。

如果您违反了这些规则(例如，如果您在另一个线程调用其方法时在mock上设置期望)，您将得到未定义的行为。那不好玩，所以不要做。

谷歌Mock保证模拟函数的操作在调用模拟函数的同一个线程中执行。例如,在

EXPECT\_CALL(模拟Foo (1)) .WillOnce (action1);  
  
EXPECT\_CALL(模拟Foo (2)) .WillOnce (action2);

如果在线程1中调用Foo(1)，在线程2中调用Foo(2)，那么谷歌Mock将在线程1中执行action1，在线程2中执行action2。

谷歌Mock不会对在不同线程中执行的操作施加顺序(这样做可能会产生死锁，因为这些操作可能需要协作)。这意味着上面例子中action1和action2的执行可能会交错。如果这是一个问题，您应该向action1和action2添加适当的同步逻辑，以使测试是线程安全的。

另外，请记住DefaultValue<T>是一个全局资源，它可能会影响程序中所有活动的模拟对象。当然，您不会希望在多个线程中或者在仍然有模拟在运行时对其进行干扰。

## 控制多少信息谷歌模拟打印

当谷歌模仿看到的东西有可能是一个错误(例如,没有预期的模拟功能,即一个无趣的电话,是允许的,但也许你忘了明确禁止调用),它打印一些警告信息,包括函数的参数和返回值。希望这能提醒你看看是否真的有问题。

有时，您确信您的测试是正确的，可能不喜欢这样友好的信息。其他时候，您正在调试测试或了解正在测试的代码的行为，并希望能够观察到发生的每个模拟调用(包括参数值和返回值)。显然，一种模式并不适合所有人。

您可以使用——gmock\_verbose=LEVEL命令行标志来控制谷歌Mock告诉您的内容，其中LEVEL是一个包含三个可能值的字符串:

* info:谷歌Mock将打印所有信息消息、警告和错误(最详细)。在此设置下，谷歌Mock还将记录对ON\_CALL/EXPECT\_CALL宏的任何调用。
* 警告:谷歌Mock将同时打印警告和错误(更简洁)。这是默认设置。
* 错误:谷歌Mock将只打印错误(最少的冗余)。

或者，你可以调整该标志的值从您的测试像这样:

::测试::FLAGS\_gmock\_verbose =“错误”;

现在，明智地使用正确的标志，使谷歌Mock更好地为您服务!

## 在模拟呼叫中获得超视觉

您有一个使用谷歌Mock的测试。它失败了:谷歌Mock告诉您有些期望没有得到满足。然而，你不确定原因:是否有一个错字在matchers?你打乱了EXPECT\_CALLs的顺序了吗?或者被测试的代码做错了什么?你怎样才能找出原因呢?

如果您有x射线视野，并且能够看到所有expect\_call和模拟方法调用的执行轨迹，这不是很好吗?对于每个调用，您是否希望看到它的实际参数值，以及哪个EXPECT\_CALL谷歌Mock认为它匹配?

您可以通过使用—gmock\_verbose=info标志运行测试来解锁此功能。例如，给定测试程序:

类MockFoo {public: MOCK\_METHOD2(F, void(const string&x, const string&y));};TEST(Foo, Bar) {MockFoo mock;  
  
  
  
  
  
  
  
  
  
  
EXPECT\_CALL(模拟,F (\_, \_)) .WillRepeatedly(返回());  
EXPECT\_CALL(模拟,F (“a”、“b”));  
EXPECT\_CALL(模拟、F (“c”HasSubstr (d)));  
  
模拟。F (“a”,“好”);  
模拟。F (“a”、“b”);}

如果用——gmock\_verbose=info运行它，您将看到以下输出:

[运行]Foo.Barfoo\_test。  
  
cc:14: EXPECT\_CALL(mock, F(\_， \_)) invokedfoo\_test。  
cc:15: EXPECT\_CALL(mock, F(“a”，“b”))invokedfoo\_test。  
cc:16: EXPECT\_CALL(mock, F("c"， HasSubstr("d"))) invokedfoo\_test。  
cc:14: Mock函数调用匹配EXPECT\_CALL(Mock, F(\_， \_))…  
函数调用:F(@0x7fff7c8dad40"a"， @0x7fff7c8dad10"good")foo\_test。  
cc:15:模拟函数调用匹配EXPECT\_CALL(模拟，F(“a”，“b”))…  
函数调用:F(@0x7fff7c8dada0"a"， @0x7fff7c8dad70"b")foo\_test。  
cc:16: FailureActual function call count不匹配EXPECT\_CALL(mock, F("c"， HasSubstr("d"))…  
  
期望:被调用一次实际:从未调用-不满意和活动[失败]页。栏

假设错误是第三个EXPECT\_CALL中的“c”是一个输入错误，实际上应该是“a”。使用上面的消息，您应该看到实际的F(“a”、“good”)调用与第一个EXPECT\_CALL匹配，而不是您所想的第三个。从这一点可以明显看出，第三个EXPECT\_CALL写错了。情况下解决。

## 在Emacs中运行测试

如果您在Emacs中构建并运行测试，则会突出显示谷歌Mock和谷歌测试错误的源文件位置。只要在其中一个上按下<Enter>，就会转到出错的行。或者，您可以输入“C-x”来跳转到下一个错误。

为了使它更简单，您可以将以下行添加到~/中。emacs文件:

(全局设置键“\M-m”'编译);m表示make(全局设置键[m -down] 'next-error])(全局设置键[m -up] '(lambda()(交互)(next-error -1))

然后您可以输入M-m来开始构建，或者输入M-up/M-down来在错误之间来回移动。

## 融合谷歌模拟源文件

谷歌Mock的实现由几十个文件组成(不包括它自己的测试)。有时，您可能希望将它们打包到更少的文件中，这样您就可以轻松地将它们复制到新机器中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\_gmock\_files。脚本/目录中的py(从版本1.2.0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\_gmock\_files。py OUTPUT\_DIR

您应该看到使用文件gtest/gtest创建的OUTPUT\_DIR目录。h, gmock / gmock。h, gmock-gtest-all。cc。这三个文件包含使用谷歌Mock(和谷歌测试)所需的所有内容。只需将它们复制到任何您想要的地方，就可以编写测试和使用模拟了。您可以使用scrpts/test/Makefile文件作为例子，说明如何根据它们编译测试。

# 延长谷歌模拟

## 快速编写新的匹配器

MATCHER\*宏家族可用于方便地定义自定义匹配器。语法:

MATCHER(name, description\_string\_expression){语句;}

将定义一个使用给定名称执行语句的匹配器，该匹配器必须返回一个bool来指示匹配是否成功。在语句中，您可以引用由arg匹配的值，并通过arg\_type引用它的类型。

description字符串是一个字符串类型的表达式，它记录了matcher的工作，并在匹配失败时用于生成失败消息。它可以(也应该)引用特殊的bool变量求反，当求反为假时求匹配器的描述，当求反为真时求匹配器的描述。

为了方便起见，我们允许描述字符串为空("")，在这种情况下，谷歌Mock将使用matcher名称中的单词序列作为描述。

例如:

{return (arg % 7) == 0;}

允许你写

//期望调用mock\_foo.Bar(n)，其中n可以被7整除。  
EXPECT\_CALL (mock\_foo栏(IsDivisibleBy7 ()));

或者,

测试使用::::,…  
  
EXPECT\_THAT (some\_expression IsDivisibleBy7 ());  
EXPECT\_THAT (some\_other\_expression,而不是(IsDivisibleBy7 ()));

如果上面的断言失败，它们将输出以下内容:

的值:some\_expression期望:可被7实际:27…  
  
  
  
期望的some\_other\_expression值:not(可被7整除)实际值:21

当描述“能被7整除”和“不能(被7整除)”时，将自动从matcher名称IsDivisibleBy7计算出来。

您可能已经注意到，自动生成的描述(特别是那些用于否定的描述)可能不是很好。你可以用你自己的字符串表达式覆盖它们:

匹配器(IsDivisibleBy7, std:: string(否定吗?"isn't": "is") + "可被7整除"){return (arg % 7) == 0;}

您还可以选择将附加信息流到名为result\_listener的隐藏参数，以解释匹配结果。例如，IsDivisibleBy7更好的定义是:

如果(((arg % 7) == 0)返回true;  
  
  
  
\*result\_listener <<“余数是”<< (arg % 7);  
返回false;}

有了这个定义，上面的断言将给出更好的信息:

值:some\_expression Expected:可被7实际:27整除(余数为6)

您应该让MatchAndExplain()打印任何能够帮助用户理解匹配结果的附加信息。注意，它应该解释匹配成功的原因(除非很明显)——这在Not()内部使用matcher时很有用。不需要打印参数值本身，因为谷歌Mock已经为您打印了。

**注:**

1. 正在匹配的值的类型(arg\_type)由使用matcher的上下文决定，并由编译器提供给您，因此您不必担心声明它(也不必担心)。这允许匹配器是多态的。例如，IsDivisibleBy7()可以用于匹配任何类型，其中(arg % 7) == 0的值可以隐式转换为bool。在上面的Bar(IsDivisibleBy7())例子中，如果Bar()方法采用int类型，arg\_type将是int类型;如果它采用无符号长，arg\_type将是无符号长;等等。
2. 谷歌Mock不保证何时或多少次调用一个匹配器。因此，匹配器逻辑必须是纯功能性的(即，它不能有任何副作用，并且结果必须只依赖于被匹配的值和匹配器参数)。无论您如何定义匹配器(例如，使用以下配方中描述的方法之一)，都必须满足这个需求。特别是，matcher永远不能调用模拟函数，因为这会影响模拟对象和谷歌模拟的状态。

## 快速编写新的参数化匹配器

有时需要定义具有参数的匹配器。你可以使用宏:

MATCHER\_P(name, param\_name, description\_string){语句;}

其中描述字符串可以是“”，也可以是引用否定和param\_name的字符串表达式。

例如:

返回abs(arg) == value;}

将允许你写:

EXPECT\_THAT(废话(“a”), HasAbsoluteValue (n));

这可能导致这样的消息(假设n是10):

值:Blah(“a”)Expected:有绝对值10 Actual: -9

注意，matcher描述和它的参数都被打印出来，使得消息对人类友好。

在matcher定义主体中，可以编写foo\_type来引用名为foo的参数的类型。例如，在MATCHER\_P(HasAbsoluteValue, value)的主体中，可以编写value\_type来引用值的类型。

谷歌Mock还提供MATCHER\_P2、MATCHER\_P3……，直至MATCHER\_P10，以支持多参数匹配器:

MATCHER\_Pk(名称、param\_1…， param\_k, description\_string){语句;}

请注意，自定义描述字符串是针对matcher的特定实例的，其中的参数已绑定到实际值。因此，通常希望参数值是描述的一部分。谷歌Mock允许您通过在描述字符串表达式中引用matcher参数来实现这一点。

例如,

测试使用::::PrintToString;  
MATCHER\_P2(InClosedRange, low, hi, std::string(?  
"isn't": "is") + " in range [" + PrintToString(low) + "， " + PrintToString(hi) + "] {return low <= arg && arg <= hi;  
  
  
}……  
  
EXPECT\_THAT (InClosedRange (4、6));

将生成一个包含消息的失败:

期望:在[4,6]范围内

如果指定“”作为描述，则失败消息将包含matcher名称中的单词序列，后面是作为元组打印的参数值。例如,

MATCHER\_P2(InClosedRange, low, hi， ""){…}……  
  
EXPECT\_THAT (InClosedRange (4、6));

将生成一个包含文本的失败:

期望:在闭合范围内(4,6)

对于输入的目的，您可以查看

MATCHER\_Pk (Foo, p1,…， pk, description\_string){…}

作为速记

模板<typename p1\_type，…， typename pk\_type>FooMatcherPk<p1\_type，…  
， pk\_type>Foo(p1\_type p1，…  
， pk\_type pk){…}

当你写Foo(v1，…，编译器推断参数v1，…， vk是给你的。如果对类型推断的结果不满意，可以通过显式实例化模板来指定类型，如Foo<long, bool>(5, false)。如前所述，您不必(或不必)指定arg\_type，因为它是由使用matcher的上下文决定的。

可以将表达式Foo(p1，…类型FooMatcherPk<p1\_type，…pk\_type >。这在组合匹配器时非常有用。没有参数或只有一个参数的匹配器具有特殊类型:可以将Foo()分配给foomatcher类型的变量，并将Foo(p)分配给FooMatcherP<p\_type>类型的变量。

虽然可以使用引用类型实例化matcher模板，但通过指针传递参数通常会使代码更具可读性。但是，如果仍然希望通过引用传递参数，请注意，在matcher生成的失败消息中，您将看到引用对象的值，而不是它的地址。

您可以使用不同数量的参数来重载匹配器:

MATCHER\_P(Blah, a, description\_string\_1){…}MATCHER\_P2(Blah, a, b, description\_string\_2){…  
}

虽然很容易时总是使用匹配器\*宏定义一个新的匹配器,您还应该考虑实现MatcherInterface或使用MakePolymorphicMatcher(),而不是(见下面的食谱),特别是如果你需要使用匹配器很多。而这些方法需要更多的工作,他们会给你更多的控制类型的值匹配,匹配器参数,这通常会导致更好的编译器错误消息,偿还从长远来看。它们还允许基于参数类型(而不是仅仅基于参数数量)来重载匹配器。

## 编写新的单形匹配器

参数类型T的matcher实现::testing::MatcherInterface<T>，做两件事:它测试类型T的值是否与matcher匹配，并且可以描述匹配的值的类型。后一种功能用于在违背预期时生成可读的错误消息。

界面是这样的:

类MatchResultListener {public:…  
  
  
//将x流到底层的ostream;如果ostream //为空，则不执行任何操作。  
  
模板<typename T> MatchResultListener& operator<<(const t&x);  
  
  
返回底层的ostream。  
模板<typename T>class MatcherInterface {public: virtual ~MatcherInterface();  
  
  
  
  
  
  
  
//如果匹配器匹配x，则返回true;也解释匹配//结果到“listener”。  
  
bool MatchAndExplain(T x, MatchResultListener\* listener) const = 0;  
  
//将此匹配器描述为ostream。  
(::std::ostream\* os) const = 0;  
  
//描述此匹配器对ostream的否定。  
(::std::ostream\* os);};

如果你需要一个定制的匹配器但真正()并不是一个很好的选择(例如,您可能不满意的方式真正(谓词)描述本身,或者你可能想要你的匹配器Eq(值)多态),您可以定义一个匹配器,做任何你想做的两个步骤:首先实现匹配器接口,然后定义一个工厂函数来创建一个匹配器实例。第二步并不是严格需要的，但是它使使用matcher的语法更好。

例如，你可以定义一个匹配器来测试一个整数是否能被7整除，然后像这样使用它:

测试使用::::MakeMatcher;使用::测试::匹配器;使用::测试::MatcherInterface;使用::测试::MatchResultListener;类DivisibleBy7Matcher:公共MatcherInterface < int >{公众:虚拟bool MatchAndExplain (int n, MatchResultListener \*侦听器)const{返回(n % 7) = = 0;  
  
  
  
  
  
  
  
  
{\*os << "is divisible by 7";  
  
  
  
{\*os << "不能被7整除";  
  
  
  
}};内联Matcher<int> DivisibleBy7() {}  
  
  
  
  
  
  
  
EXPECT\_CALL (foo, Bar (DivisibleBy7 ()));

您可以通过向MatchAndExplain()中的listener参数流附加信息来改进matcher消息:

{public: virtual bool MatchAndExplain(int n, MatchResultListener\* listener) {const int余数= n % 7;  
  
  
  
  
如果(余数!= 0){\*监听器<<“余数是”<<余数;  
  
}返回余数== 0;  
  
}……};

然后,EXPECT\_THAT (x, DivisibleBy7 ());可以这样概括一条信息:

xExpected:的值可被7个实际的23整除(余数为2)

## 编写新的多态匹配器

在前面的菜谱中，您已经学习了如何编写自己的匹配器。只有一个问题:使用MakeMatcher()创建的匹配器只适用于一种特定类型的参数。如果你想要一个与参数的多态匹配器的几种类型(例如,Eq (x)可以用来匹配一个价值只要价值= = x编译——价值和不需要共享同一类型),你可以从“gmock / gmock-matchers学到的技巧。h“但是有点复杂。

幸运的是，在大多数情况下，您可以在make多态性匹配器()的帮助下轻松定义多态匹配器。下面是如何定义NotNull()作为一个例子:

测试使用::::MakePolymorphicMatcher;使用::测试::MatchResultListener;使用::测试::NotNull;使用::测试::PolymorphicMatcher;类NotNullMatcher{公众:/ /实现多态的匹配器,首先定义一个复制类/ /有三个成员MatchAndExplain (), DescribeTo(),和/ / DescribeNegationTo(),如下。  
  
  
  
  
  
  
  
  
  
  
//在本例中，我们希望对任何指针使用NotNull()，因此// MatchAndExplain()接受任何类型的指针作为它的第一个参数。  
  
//通常，可以将MatchAndExplain()定义为普通方法或//方法模板，甚至重载它。  
  
模板<typename T> bool MatchAndExplain(T\* p, MatchResultListener\* /\* listener \*/) const {return p != NULL;  
  
  
  
} //描述与此匹配器匹配的值的属性。  
  
  
void characterbeto (::std::ostream\* os) const {\*os << "is not NULL";} //描述不匹配此匹配器的值的属性。  
  
  
void description to (::std::ostream\* os) const {\*os << "is NULL";//要构造一个多态匹配器，传递一个类的实例给make多态性匹配器()。  
  
  
  
注意返回类型。  
{return make多态性matcher (NotNullMatcher());}…  
  
  
  
  
EXPECT\_CALL (foo, Bar (NotNull ()));参数必须是非空指针。

**注意:您的多态matcher类不需要从MatcherInterface或任何其他类继承，它的方法也不需要是虚的。**

与单态匹配器类似，您可以通过向MatchAndExplain()中的侦听器参数传输附加信息来解释匹配结果。

## 编写新基数

在Times()中使用基数来告诉谷歌Mock您希望调用发生多少次。不一定要精确。例如，你可以说至少(5)或介于(2,4)之间。

如果内置的基数集不适合你，你可以自由定义自己的实现以下接口(在名称空间测试):

类CardinalityInterface {public: virtual ~CardinalityInterface();  
  
  
  
//返回true iff call\_count调用将满足这个基数。  
IsSatisfiedByCallCount(int call\_count) const = 0;  
  
//返回true iff call\_count调用将会饱和这个基数。  
IsSaturatedByCallCount(int call\_count) const = 0;  
  
//向别人描述自己。  
(::std::ostream\* os) const = 0;};

例如，要指定调用必须出现偶数次，可以编写

使用::testing::CardinalityInterface;使用::testing::CardinalityInterface;使用::testing::MakeCardinality;类偶数基数:public CardinalityInterface {public: virtual bool IsSatisfiedByCallCount(int call\_count) const {return (call\_count % 2) == 0;  
  
  
  
  
  
  
  
{}虚拟bool IsSaturatedByCallCount(int call\_count) const {return false;  
  
  
  
}虚虚描述(::std::ostream\* os) const {\*os <<“调用偶数次”;  
  
  
  
}};Cardinality EvenNumber(){返回MakeCardinality(新的偶数基数);}…  
  
  
  
  
  
  
  
EXPECT\_CALL (foo, Bar(3)同学(EvenNumber ());

## 快速编写新动作

如果内置操作对您不起作用，并且您发现使用Invoke()很不方便，那么您可以使用来自ACTION\*系列的宏来快速定义一个新操作，该操作可以像内置操作一样在您的代码中使用。

通过编写

行动(名字){语句;}

在名称空间范围内(例如，不在类或函数中)，您将使用给定的名称定义一个执行语句的操作。语句返回的值将用作操作的返回值。在这些语句中，可以将模拟函数的第k(基于0)参数引用为argK。例如:

ACTION(IncrementArg1) {return ++(\*arg1);}

允许你写

…WillOnce (IncrementArg1 ());

注意，您不需要指定模拟函数参数的类型。请放心，您的代码是类型安全的:如果\*arg1不支持++操作符，或者如果++(\*arg1)的类型与模拟函数的返回类型不兼容，您将得到一个编译器错误。

另一个例子:

操作(Foo){(\*最长)(5);  
  
等等();  
\* \_\_arg1 = 0;  
返回arg0;}

定义一个Foo()动作，用5调用参数#2(一个函数指针)，调用函数Blah()，设置参数指向的值

# 从1到0，返回参数#0。

为了更加方便和灵活，你还可以在动作主体中使用以下预定义的符号:

|  |  |
| --- | --- |
| argK\_type | mock函数的第k(基于0)参数的类型 |
| arg游戏 | 模拟函数的所有参数都是一个元组 |
| args\_type | 模拟函数的所有参数的类型为一个元组 |
| return\_type | 模拟函数的返回类型 |
| function\_type | 模拟函数的类型 |

例如，当使用一个动作作为模拟函数的存根动作时:

int DoSomething(bool flag, int\* ptr);

我们有:

|  |  |
| --- | --- |
| **预定义的符号** | **一定会** |
| arg0 | 标志的值 |
| arg0\_type | bool类型 |
| \_\_arg1 | ptr的值 |
| arg1\_type | int类型\* |
| arg游戏 | tuple(标记，ptr) |
| args\_type | 类型::testing::tuple<bool, int\*> |
| return\_type | int类型 |
| function\_type | 类型int(bool, int\*) |

## 快速编写新的参数化操作

有时需要对定义的操作进行参数化。这是另一个宏

ACTION\_P(name, param){语句;}

例如,

ACTION\_P(Add, n){返回arg0 + n;}

会允许你写信吗

//返回参数#0 + 5…  
WillOnce(添加(5));

为了方便起见，我们使用术语参数表示用于调用模拟函数的值，使用术语参数表示用于实例化操作的值。

注意，您也不需要提供参数的类型。假设参数名为param，您还可以使用google - mock定义的符号param\_type来引用编译器推断的参数类型。例如，在上面ACTION\_P(Add, n)的主体中，您可以为n的类型编写n\_type。

谷歌Mock还提供了ACTION\_P2、ACTION\_P3等来支持多参数操作。例如,

ACTION\_P2(ReturnDistanceTo, x, y) {double dx = arg0 - x;  
  
双dy = arg1 - y;  
返回sqrt(dx\*dx + dy\*dy);}

让你写

…WillOnce (ReturnDistanceTo (5.0, 26.5));

您可以将操作视为参数数量为0的退化参数化操作。

你也可以很容易地定义动作重载的数量的参数:

ACTION\_P(Plus, a){…}ACTION\_P2(Plus, a, b){…  
}

## 限制操作中参数或参数的类型

为了获得最大的简便性和可重用性，ACTION\*宏不要求您提供模拟函数参数和操作参数的类型。相反，我们让编译器为我们推断类型。

然而，有时我们可能希望更明确地说明类型。有几个技巧可以做到这一点。例如:

ACTION(Foo){//确保arg0可以被转换成int. int n = arg0;  
  
  
…这里用n代替arg0…  
}ACTION\_P(Bar, param){//确保arg1的类型是const char\*。  
  
  
  
::测试::StaticAssertTypeEq < const char \*, arg1\_type > ();  
  
//确保参数可以转换为bool。  
bool flag = param;}

其中StaticAssertTypeEq是谷歌测试中的编译时断言，用于验证两种类型是否相同。

## 快速编写新的动作模板

有时，您希望为操作提供显式模板参数，而这些参数不能从其值参数中推断出来。ACTION\_TEMPLATE()支持该方法，可以将其视为ACTION()和ACTION\_P\*()的扩展。

语法:

ACTION\_TEMPLATE(ActionName, HAS\_m\_TEMPLATE\_PARAMS) (kind1, name1，…  
， kind\_m, name\_m)，和\_n\_value\_params (p1，…  
， p\_n)){语句;}

定义一个动作模板，它接受m个显式模板参数和n个值参数，其中m在1到10之间，n在0到10之间。name\_i是第i个模板参数的名称，kind\_i指定它是typename、整数常量还是模板。p\_i是第i个值参数的名称。

例子:

// DuplicateArg<k, T>(output)将mock//函数的第k个参数转换为类型T，并将其复制到\*output。  
  
注意int和k之间的逗号:HAS\_2\_TEMPLATE\_PARAMS(int, k, typename, T)， and \_1\_value\_params (output) {\*output = T(::testing::get<k>(args));}

创建一个动作模板的实例，写:

ActionName < t1,…,t\_m > (v1,…v\_n)

其中ts是模板参数，vs是值参数。值参数类型由编译器推断。例如:

测试使用::::\_;…  
  
int n;  
.WillOnce(DuplicateArg<1, unsigned char>(&n));

如果你想显式指定值参数类型，你可以提供额外的模板参数:

ActionName < t1,…， t\_m, u1，…,u\_k > (v1,…v\_n)

其中u\_i是所需的v\_i类型。

ACTION\_TEMPLATE和ACTION/ACTION\_P\*可以在值参数的数量上重载，但是不能在模板参数的数量上重载。没有这个限制，下面的意思就不清楚了:

OverloadedAction < int, bool > (x);

我们是使用单个模板-参数操作(bool指的是x的类型)，还是使用两个模板-参数操作(要求编译器推断x的类型)?

## 使用ACTION对象的类型

如果您正在编写一个返回ACTION对象的函数，那么您需要知道它的类型。类型取决于用于定义操作和参数类型的宏。规则相对简单:

|  |  |  |
| --- | --- | --- |
| **给出定义** | **表达式** | **有类型** |
| 操作(Foo) | Foo () | FooAction |
| ACTION\_TEMPLATE (Foo, HAS\_m\_TEMPLATE\_PARAMS (…), AND\_0\_VALUE\_PARAMS ()) | Foo < t1,…,t\_m > () | FooAction < t1,…,t\_m > |
| ACTION\_P(酒吧、参数) | 栏(int\_value) | BarActionP < int > |
| ACTION\_TEMPLATE(酒吧,HAS\_m\_TEMPLATE\_PARAMS (…), AND\_1\_VALUE\_PARAMS (p1)) | 酒吧< t1,…t\_m > (int\_value) | FooActionP < t1,…t\_m, int > |
| ACTION\_P2 (Baz, p1, p2) | 巴兹(bool\_value int\_value) | BazActionP2 < bool, int > |
| ACTION\_TEMPLATE(Baz, HAS\_m\_TEMPLATE\_PARAMS(…)和\_2\_value\_params (p1, p2)) | 巴兹< t1,…t\_m > (bool\_value int\_value) | FooActionP2 < t1,…， t\_m, bool, int> |
| … | … | … |

请注意，我们必须为具有不同数量的值参数的操作选择不同的后缀(Action、ActionP、ActionP2等)，否则操作定义不能因它们的数量而重载。

## 编写新的单态动作

虽然ACTION\*宏非常方便，但有时并不合适。例如，尽管前面的菜谱中展示了一些技巧，但是它们不允许您直接指定模拟函数参数和操作参数的类型，这通常会导致编译器错误消息无法处理，这可能会让不熟悉的用户感到困惑。它们也不允许基于参数类型的重载操作，除非跳过一些限制。

ACTION\*宏的另一种替代方法是实现::testing::ActionInterface<F>，其中F是将要使用ACTION的模拟函数的类型。例如:

模板<typename F>类ActionInterface {public: virtual ~ActionInterface();  
  
  
  
//执行动作。Result是函数类型// F的返回类型，ArgumentTuple是F. // //的参数的元组，例如，如果F是int(bool, const string&)，那么Result就是int, ArgumentTuple是::testing::tuple<bool, const string&>。  
  
  
  
  
虚拟执行结果(const ArgumentTuple& args) = 0;};使用::测试::\_;使用::测试::行动;使用::测试::ActionInterface;使用::测试::MakeAction; typedef int IncrementMethod (int \*);类IncrementArgumentAction:公共ActionInterface < IncrementMethod >{公众:虚拟int执行(const::测试::元组< int \* > & args) {int \* p =::测试::< 0 > (args);  
  
  
  
  
  
  
  
  
  
  
  
  
//获取第一个参数。  
返回\* p + +;  
}};Action<IncrementMethod> IncrementArgument() {return MakeAction(new IncrementArgumentAction);}…  
  
  
  
  
  
  
  
EXPECT\_CALL (foo,巴兹(\_)).WillOnce (IncrementArgument ());  
  
  
int n = 5;  
foo.Baz (n);//应该返回5并将n改为6。

## 编写新的多态动作

前面的菜谱向您展示了如何定义自己的操作。这些都很好，除了您需要知道将在其中使用操作的函数的类型之外。有时这可能是个问题。例如，如果希望在具有不同类型的函数中使用操作(例如Return()和SetArgPointee())。

如果一个动作可以用在几种类型的模拟函数中，我们就说它是多态的。函数模板使得定义这样的动作变得很容易:

名称空间测试{模板<typename Impl>多态性动作<Impl> make多态性动作(const impl&impl);} //名称空间测试

作为一个示例，让我们定义一个操作，该操作返回模拟函数的参数列表中的第二个参数。第一步是定义一个实现类:

类ReturnSecondArgumentAction {public: template <typename Result, typename ArgumentTuple> Result Perform(const argumenttuple&args) const{//要获得第i个(基于0的)参数，使用::testing::get<i>(args)。  
  
  
  
  
返回:测试::< 1 > (args);  
}};

这个实现类不需要继承任何特定的类。重要的是它必须有一个执行()方法模板。此方法模板将模拟函数的参数作为单个参数中的元组，并返回操作的结果。它可以是const，也可以不是，但是必须能够调用一个模板参数，即结果类型。换句话说，您必须能够调用执行<R>(args)，其中R是模拟函数的返回类型，args是元组中的参数。

接下来，我们使用makemorphicaction()将实现类的一个实例转换成我们需要的多态操作。这将是方便的包装:

使用::testing:: make多态性作用;使用::testing::多态性作用;多态性作用<ReturnSecondArgumentAction> ReturnSecondArgument() {return make多态性作用(ReturnSecondArgumentAction());}

现在，你可以使用这个多态动作，就像你使用内置的:

{public: MOCK\_METHOD2(DoThis, int(bool flag, int n));  
  
  
  
  
MOCK\_METHOD3(DoThat, string(int x, const char\* str1, const char\* str2));};  
  
  
  
MockFoo foo;  
.WillOnce(ReturnSecondArgument());  
  
方法如下:= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =  
  
…  
foo。做的(真的,5);//将返回5。  
foo。DoThat(1,“你好”,“再见”);//将返回“Hi”。

## 教谷歌Mock如何打印您的值

当发生无趣或意外调用时，谷歌Mock将打印参数值和堆栈跟踪，以帮助您调试。当断言失败时，像EXPECT\_THAT和EXPECT\_EQ这样的断言宏也会打印有问题的值。谷歌Mock和谷歌测试使用谷歌测试的用户可扩展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。[谷歌测试的高级指南解释了如何扩展打印机来更好地打印您的特定类型，而不是转储字节。](../../googletest/docs/AdvancedGuide.md#teaching-google-test-how-to-print-your-values)