Google C++ Mocking Cookbook

Version: 0.32

作者：*Adrian Alexander*

译者：*Koala++ / 屈伟*

你来对地方了，这里你可以找到 Google Mock 的使用方法，但如果你还没有读过启蒙篇，建议你还是先去读一下吧，了解些基本知识。

注意，Google Mock 定义在 testing 命名空间中。你可以用 using ::testing::Foo 来让代码有更好的可读性。在本文中为了简洁起见，并不采用这种写法，但是在你自己的代码中应该用 using。

Create Mock Classes

Mocking Private or Protected Methods

你必须将 Mock 函数定义(MOCK_METHOD*)放到 Mock 类的 public:部分中，无论被 Mock 的函数在基类中是 public，protected，还是 private。这样做是为了让 ON_CALL 和 EXPECT_CALL 可以从 Mock 类外引用 Mock 函数。(是的，C++允许子类改变一个基类虚函数的访问权限)。比如：

```
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 Overloaded Methods

Mock 重载函数的方法也是一样的，不需要使用别的方式：

```
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 如果你并不 Mock 所有的重载函数 编译器会警告你基类中的一些函数被隐藏了。修正的方法是用 using 将它们引入域中：

```
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);
  ...
};
```

Mocking Class Templates

Mock 一个模板类，需要在 MOCK_*宏后加上_T：

```
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 Non-virtual Methods

Google Mock 可以 Mock 非虚函数用在 *hi-perf dependency injection* 中。

Mock 非虚函数时并不与真实的类共享一个公共的基类，你的 Mock 类与真实类将*毫无关系*，但两者所定义的函数却是一致的。Mock 非虚函数与 Mock 虚函数的语法是*一致*的：

```
// 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());
  ...
```

```
};
```

注意与真实类不同的是 Mock 类没有定义 AppenPacket()，但只要测试中没有调用到这个函数，这种写法是没有问题的。

接下来，你需要想出一种在正式代码中使用 ConcretePacketStream，在测试代码中使用 MockPacketStream 的方法。因为函数是非虚的，而两个类也是毫无关系的，所以你必须在编译时(而不是运行时)决定你使用的类。

其中一种方法是模板化需要用 Packet Stream 的代码。具体一点，你在代码中使用一个针对 packet stream 模板参数。在正式代码中，你可以用 ConcretePacketStream 来实例化，在测试中你用 MockPacketStream 来实例化。下面是一个例子：

```
template <class PacketStream>
void CreateConnection(PacketStream* stream) { ... }

template <class PacketStream>
class PacketReader {
 public:
  void ReadPackets(PacketStream* stream, size_t packet_num);
};
```

然后你可以在正式代码中使用 CreateConnection<ConcretePacketStream>()和 PacketReader<ConcretePacketStream> ，在测试代码中使用 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 Functions

可以使用Google Mock来Mock一个自由函数(比如 ,普通C风格函数或是静态函数)。但你需要一个接口(抽象类)重写你的代码。

Mock 并不直接调用自由函数(暂且称之为 OpenFile)，而是为它引入一个接口，并需要针对这个接口实现对函数对自由函数的调用：

```
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);
  }
};
```

你的代码可以通过 FileInterface 打开一个文件，现在函数更容易被 Mock。

这看起来太麻烦了，但在现实中你通常可以将多个相关的函数放到一个接口中，所以为每个函数定义一个接口这种额外工作会少很多。

Nice Mocks and Strict Mocks

如果一个没有指定 EXPECT_CALL 的 Mock 函数被调用了，Google Mock 会打印一个”uninteresting call”警告。这样做的合理性如下：

- 当测试写完之后，可能有新的函数加入到接口中。而我们不能仅因为一个测试它不知道某个函数要被调用就失败。
- 但是，这种情况也可能意味着测试中有 bug，所以 Google Mock 也不能什么都不提示。如果用户认为这些调用是无关的，它可以加入一个 EXPECT_CALL 来消除警告。

但是，有时你可能想消除所有的”uninteresting call”警告，但有时你可能想做刚好相反的事，即认为所有的”uninteresting call”都是错误。Google Mock 能让你在 Mock 对象这个级别上选择你的决定。

```
TEST(...) {
  MockFoo mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

如果 mock_foo 中一个不是 DoThis 的函数被调用了，Google Mock 会给出一个警告，但是你用 NiceMock<MockFoo>重写你的测试，警告会消失，你会得到一个更清爽的输出：

```
using ::testing::NiceMock;

TEST(...) {
  NiceMock<MockFoo> mock_foo;
  EXPECT_CALL(mock_foo, DoThis());
  ... code that uses mock_foo ...
}
```

NiceMock 是 MockFoo 的一个子类，所以它在任何接受 MockFoo 类型的地方使用。

在 MockFoo 的构造函数是有参数的时候也是可以用的，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 ...
}
```

StickMock 的用法也是相似的，只是它的目的是让所有“uninteresting call”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.
}
```

下面还有一些说明（我不太喜欢这些说明，但遗憾的是它们是C++限制的副作用）：

1. NiceMock<MockFoo>和StickMock<MockFoo>仅在**直接**在MockFoo类中使用MOCK_METHOD*定义的Mock函数。如果一个Mock函数在MockFoo的**基类**中定义，那么“nice”或是“strict”是否会影响它则取决于编译器。特别要指出的是，嵌套的NiceMock和StrickMock是不支持的（比如，NiceMock<StrictMock<MockFoo>>）。
2. Mock基类（MockFoo）**不能**传递非常量引用给构造函数，因为这种做法被Google C++编码规范禁止了。
3. 在构造函数和析构函数运行中，Mock对象不是nice也不是strict。如果在这个对象的构造函数或是析构函数中调用一个Mock函数，可能会因为这个原因造成意外。（译注：Effective C++ Item 8）。

最后，你必须在使用这个特性时**特别小心**，因为你所做的这个决定会应用到Mock类未来**所有**的改动上。如果你所Mock的接口做了一个重要的改变，它会让你的测试（如果你用StrictMock）失败或是在没有警告提示的情况下让bug溜过（如果你使用NiceMock）。所以，应该显式地调用EXPECT_CALL来指定mock的行为，仅在最后将Mock对象换为NiceMock或是StrictMock的。

Simplifying the Interface without Breaking Existing Code

有时候一个函数有相当长的参数列表，那Mock的时候是相当无趣的，比如：

```
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;
};
```

这个函数的参数列表很长且难用(这么说吧，message 参数甚至都不是以'\0'结尾的)。如果我们执意要 Mock 它，那结果必是不雅的。然而如果我们试着简化这个接口，又需要将所有使用这个接口的代码全部改了，这通常是不可行的。

技巧就是在 Mock 类中修改这个函数：

```
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));
};
```

通过定义一个新有较少参数的 Mock 函数，我们让 Mock 类更易用。

Alternative to Mocking Concrete Classes

你经常会发现你正在用一些没有针对接口实现的类。你为了可以用这种类(且称为 Concrete 类)来测试自己的代码，你可能会试着将 Concrete 的函数变为虚函数，然后再去 Mock 它。

请不要这样做。

将非虚函数改为虚函数是一个重大决定。这样做之后，子类会改变父类的行为。这样就会更难保持类的不变性，而从降低了你对类的控制力。你只应在一个合理的理由下将非虚函数变为虚函数。

直接 Mock 具体的类会产生类和测试的高度耦合，任何对类的小的改动都会让你测试失效，这会让你陷入维护测试的痛苦中。

为了避免这种痛苦，许多程序员开始了“针对接口”的实践：并不直接调用 Concrete 类，而是定义一个接口去调用 Concrete 类。然后你在 Concrete 类之上实现这个接口，即配接器。

这种技术可能会带来一些负担：

- 你要为虚函数的调用买单（ 通常不是问题 ）
- 程序员需要掌握更多的抽象

但是，它同时也能巨大的好处，当然也有更好的可测性：

- Concrete 的 API 也许并不是很适合你的问题领域，因为你可能不是这个 API 唯一的调用方。通过设计你自己的接口，你有一个将这个类修改成自己所需的类的机会，你可加入一些特定功能，重命名接口函数，等等，你可以做的不是只是减少几个自己不使用的 API。这可以让你自己以更自然的方式实现你的代码，因为它有更好的可读性，更好的可维护性，你也会有更高的编程效率。
- 如果 Concrete 的实现改变了，你不需要重写与改动相关的所有测试。相反你可以将改动在你自己的接口中隐藏，使你的调用代码和测试与 Concrete 改动绝缘。

有些人会担心如果每个人都在实践这个技术，将会产生大量的重复代码。这个担心是可以理解的。但是，有两个理由可以证明这种情况可能不会发生。

- 不同的工程可能会以不同的方式使用 Concrete，所以最适合每个工程的接口是不同的。所以每个工程都有在 Concrete 之上的自己的领域相关的接口，这些接口是各不相同的。
- 如果有很多的工程用相同的接口，它们可以共用一个接口，就像它们共用 Concrete 一样。你可以在 Concrete 类的旁边提交接口和配接器的代码（ 也许是在一个 contrib 子目录中 ）并让许多工程使用它。

你需要仔细衡量针对你特定问题这种做法的优缺点，但我可以向你保证的是：Java 世界的人已经实践这种方法很久了，并且它已经被证明在很广泛的领域中是一种有效的技术。

Delegating Calls to a Fake

有时你已经有一个对某一接口的 Fake 实现了。比如：

```
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);
  }
};
```

现在你想要 Mock 这个接口，比如你想在它上面设置期望。但是你还想用 FakeFoo 作为 Mock 类函数的默认行为，当然你可以选择将代码复制到 Mock 对象里，但是这会有很大的工作量。

当你用 Google Mock 来定义 Mock 类，你可以代理对象的默认行为给你已经有的 Fake 类，用下面的方法：

```
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.
};
```

你现在可以像以前一样在你的测试中使用 MockFoo。只是你要记得如果你没有明确地设置 ON_CALL 或是 EXPECT_CALL()的行为，那 Fake 函数就会被调用：

```
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);
}
```

一些技巧：

- 如果你不想用 FakeFoo 中的函数，你仍然可以通过在 ON_CALL 或是在 EXPECT_CALL 中用.WillOnce() / .WillRepeated()覆盖默认行为。
- 在 DelegateToFake()中，你只需要代理那些你要用的函数的 Fake 实现。
- 这里所讲的技术对重载函数也是适用的，但你需要告诉编译器你是指重载函数中的哪一个。消除一个 Mock 函数的歧义(即你在 ON_CALL 中指定的)，参见“Selecting Between Overloaded Functions”一节，消除一个 Fake 函数的歧义(即在 Invoke 中的)，使用 static_cast 来指定函数的类型。
- 将一个 Mock 和一个 Fake 搅在一起通常是某种错误的信号。也许你还没有习惯基于交互方式的测试。或是你的接口融合了过多的角色，应该把这个接口分开。所以**别滥用这个技术**。我们建议这仅应该用做你重构你代码时的中间步骤。

再思考一个 Mock 和一个 Fake 混在一起的问题，这里有一个例子来说明为什么这是一个错误信号：假设你有一个 System 类，它实现了一些低层的系统操作。具体一些，它处理文件操作和 I/O 操作。假设你想测试你的代码是如何使用 System 来进行 I/O 操作，你只是想让文件操作工作正常就可以了。如果你想要 Mock 整个 System 类，你就必须提供一个关于文件操作的 Fake 实现，这表明 System 拥有了太多的角色。

相反，你可以定义一个 FileOps 和一个 IOOps 接口来拆分 System 的功能。然后你可以 Mock IOOps 而不用 Mock FileOps。

Delegating Calls to a Real Object

当使用测试 doubles(替身的意思 mocks, fakes, stubs 等等)时，有时它们的行为与真实对象的行为不一样。这种差别可能是有意为之(比如模拟一个错误，假设你的代码

中有错误处理逻辑)或是无意的。如果你的 Mock 与真实对象的差别是错误造成的，你可能会得到能通过测试，却在正式代码中失败的代码。

你可以使用 delegating-to-real 技术来保证你的 Mock 与真实对象有着相同的行为，并且拥有验证调用的能力。这个技术与 delegating-to-fake 技术很相似，区别在于我们使用真实对象而不是一个 Fake。下面是一个例子：

```
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 ...
```

用上面的代码 ,Google Mock 会验证你的代码是否做了正确的调用(有着正确的参数，以正确的顺序，有着正确的调用次数)，并且真实的对象会处理这些调用(所以行为将会和正式代码中表现一致)。这会让你在两个世界都表现出色。

Delegating Calls to a Parent Class

理想中，你应该针接口编程，并接口的函数都是虚函数。现实中，有时候你需要 Mock

一个非纯虚函数(比如，它已经有了实现)。比如：

```
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));
  };
```

有时你想调用 Foo::Concrete()而不是 MockFoo::Concrete()。也许你想将它做为 Stub 行为的一部分，或是也许你的测试根本不需要 Mock Concrete() （当你不需要 Mock 一个新的 Mock 类的任何一个函数时候，而定义一个新的 Mock 类时，将会是出奇的痛苦)。

解决这个问题的技巧就是在你的 Mock 类中留下一个后门，可以通过它去访问基类中的真实函数：

```
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); }
  };
```

现在你可以在一个动作中调用 Foo::Concrete()：

```
  using ::testing::_;
  using ::testing::Invoke;
  ...
    EXPECT_CALL(foo, Concrete(_))
        .WillOnce(Invoke(&foo, &MockFoo::FooConcrete));
```

或是告诉 Mock 对象你不想 MockConcrete()：

```
using ::testing::Invoke;
  ...
```

```
    ON_CALL(foo, Concrete(_))
        .WillByDefault(Invoke(&foo, &MockFoo::FooConcrete));
```

（为什么我们不只写 Invoke(&foo, &Foo::Concrete)？如果你这样做，MockFoo::Concrete 会被调用（从而导致无穷递归），因为 Foo::Concrete()是虚函数。这就是 C++的工作方式）。

Using Matchers

Matching Argument Values Exactly

你可以精确指定一个 Mock 函数期望的参数是什么：

```
using ::testing::Return;
...
  EXPECT_CALL(foo, DoThis(5))
      .WillOnce(Return('a'));
  EXPECT_CALL(foo, DoThat("Hello", bar));
```

Using Simple Matchers

你可以用 Matchers 去匹配有一定特征的参数：

```
using ::testing::Ge;
using ::testing::NotNull;
using ::testing::Return;
...
EXPECT_CALL(foo, DoThis(Ge(5)))  // The argument must be >= 5.
    .WillOnce(Return('a'));
EXPECT_CALL(foo, DoThat("Hello", NotNull()));
// The second argument must not be NULL.
```

一个常用的 Matcher 是_，它表示匹配任何参数：

```
using ::testing::_;
using ::testing::NotNull;
...
  EXPECT_CALL(foo, DoThat(_, NotNull()));
```

Combining Matchers

你可以使用已有的 AllOf()，AnyOf()，Not()，组合产生一些复杂的 Matchers：

```
using ::testing::AllOf;
using ::testing::Gt;
```

```
using ::testing::HasSubstr;
using ::testing::Ne;
using ::testing::Not;
...
  // The argument must be > 5 and != 10.
  EXPECT_CALL(foo, DoThis(AllOf(Gt(5),
                                Ne(10))));

  // The first argument must not contain sub-string "blah".
  EXPECT_CALL(foo, DoThat(Not(HasSubstr("blah")),
                          NULL));
```

Casting Matchers

Google Matchers 都是静态类型的，即如果你错误地使用 Matcher 的类型(比如，如果你使用 Eq(5)去匹配一个 string 参数)，编译器会报错。这也是为你好。

但有时，你知道你自己在做什么，并希望编译器放你一马。举例来说：如果你有一个针对 long 类型的 Matcher，但你想匹配的是 int。虽然这两个类型不同，但实际上用 Matcher<long>去匹配 int 并没有错，毕竟，我们可以先将 int 参数转换成 long，再传给 Matcher。

为了支持这种需求，Google Mock 提供了 SafeMatcherCast<T>(m)函数。它将一个 Matcher m 转换成 Matcher<T>。为了保证它是安全的转换，Google Mock 如检查(令 U 为 m 接受的参数)：

1. T 类型可以隐式地转换为 U 类型。
2. 当 T 和 U 都是内置数值类型时(bool，integers，float)，从 T 到 U 的转换是无损的(换句话说，用 T 类型的任何值都可以用 U 类型表示)。
3. 当 U 是一个引用，T 必须也是一个引用(因为底层的 Matcher 也许会对 U 类型参数的地址感兴趣)。

如果上述条件中任何一个没满足，是不会通过编译的。

下面是一个例子：

```
using ::testing::SafeMatcherCast;

// A base class and a child class.
class Base { ... };
class Derived : public Base { ... };

class MockFoo : public Foo {
```

```
 public:
  MOCK_METHOD1(DoThis, void(Derived* derived));
};
...

  MockFoo foo;
  // m is a Matcher<Base*> we got from somewhere.
  EXPECT_CALL(foo, DoThis(SafeMatcherCast<Derived*>(m)));
```

如果你发现 SafeMatcherCast<T>(m) 太严格，你可以用一个类似的函数 MatcherCast<T>(m)。两个函数的区别是如果 static_cast 可以将 T 类型转换成类型 U，那么 MatcherCast 就可以转换。

MatcherCast 对你凌驾于 C++类型系统之上很重要的(static_cast 不总是安全的，比如它可以丢弃一部分信息)，所以要小心不要误用/滥用它。

Selecting Between Overloaded Functions

如果你期望一个重载函数被调用，编译器需要你来指明你指的是重载函数中的哪一个。

消除一个对象上关于常量的重载的歧义，使用 Const()来指明。

```
using ::testing::ReturnRef;

class MockFoo : public Foo {
...
MOCK_METHOD0(GetBar, Bar&());
MOCK_CONST_METHOD0(GetBar, const Bar&());
};
...

MockFoo foo;
Bar bar1, bar2;
EXPECT_CALL(foo, GetBar())         // The non-const GetBar().
    .WillOnce(ReturnRef(bar1));
EXPECT_CALL(Const(foo), GetBar())  // The const GetBar().
    .WillOnce(ReturnRef(bar2));
```

(Const()由 Google Mock 定义，并返回它的参数的 const 引用。)

消除函数个数相同，但参数类型不同重载函数的歧义，你也许需要精确指定一个 Matcher 的匹配类型，在 Matcher<type>中修饰你的 Matcher，或是使用一个类型是确定的 Matcher(TypedEq<type>，An<type>()等等):

```
using ::testing::An;
```

```
using ::testing::Lt;
using ::testing::Matcher;
using ::testing::TypedEq;

class MockPrinter : public Printer {
 public:
  MOCK_METHOD1(Print, void(int n));
  MOCK_METHOD1(Print, void(char c));
};

TEST(PrinterTest, Print) {
  MockPrinter printer;

  EXPECT_CALL(printer, Print(An<int>()));            // void
Print(int);
  EXPECT_CALL(printer, Print(Matcher<int>(Lt(5))));  // void
Print(int);
  EXPECT_CALL(printer, Print(TypedEq<char>('a')));   // void
Print(char);

  printer.Print(3);
  printer.Print(6);
  printer.Print('a');
}
```

Performing Different Actions Based on the Arguments

当一个 Mock 函数被调用时，最后一个有效的期望会被匹配（“新规则覆盖老规则”）。所以你可以让一个函数根据它的参数值去做不同的事，如下：

```
using ::testing::_;
using ::testing::Lt;
using ::testing::Return;
...
  // The default case.
  EXPECT_CALL(foo, DoThis(_))
      .WillRepeatedly(Return('b'));

  // The more specific case.
  EXPECT_CALL(foo, DoThis(Lt(5)))
      .WillRepeatedly(Return('a'));
```

现在，如果 foo.DoThis()被调用时参数值小于 5，就会返回’a’，否则返回’b’。

Matching Multiple Arguments as a Whole

有时候只能单独去匹配参数是不够的。比如，我们想设置第一个参数的值必须小于第二个参数的值。With 子句可以让我们将 Mock 函数的参数做为一个整体去匹配。比如：

```
using ::testing::_;
using ::testing::Lt;
using ::testing::Ne;
...
EXPECT_CALL(foo, InRange(Ne(0), _))
    .With(Lt());
```

上面代码意为 InRange 第一个参数必须非 0，并且必须小于第二个参数。

在 With 内的语句必须是一个 Match<tr1::tuple<A1, ..., An> >类型的 Matcher，其中 A1,..., An 是函数参数的类型。

你还可以用 AllArgs(m)来代替将 m 写在.With()里的写法。两种形式意义相同，但是.With(AllArgs(Lt()))比.With(Lt())更具有可读性。

你可以用 Args<k1, ..., kn>(m) 根据 m 规则来匹配 n 个选择的参数。比如：

```
using ::testing::_;
using ::testing::AllOf;
using ::testing::Args;
using ::testing::Lt;
...
EXPECT_CALL(foo, Blah(_, _, _))
    .With(AllOf(Args<0, 1>(Lt()), Args<1, 2>(Lt())));
```

为了方便和举例起见，Google Mock 提供了关于 2-tuples 的 Matchers，包括上面的 Lt() Matcher。可以到 CheatSheet 中找到完整的列表。

注意如果你想将这些参数传递给你自己的 Predicate(比如 .With(0, 1)(Truly(&MyPredicate)))，你的必须以 tr1::tuple 做为它的参数。Google Mock 会将 n 个选中的参数作为单个 tuple 传递给 Predicate。

Using Matchers as Predicates

你是否注意到 Matcher 只是一个好看一些的 Pridicate，许多已有的算法可将 Predicates 作为参数(比如，那些在 STL 的<algorithm>中定义的算法)，如果 Google Mock Matchers 不能参与到其中，那将是一个遗憾。

幸运的地，你可以将一元 Predicate 仿函数放到 Matches()函数中来使用 Matcher，比如：

```
#include <algorithm>
#include <vector>

std::vector<int> v;
...
// How many elements in v are >= 10?
const int count = count_if(v.begin(), v.end(), Matches(Ge(10)));
```

因为你可通过将简单的 matchers 组合来产生复杂的 matchers，那么这就给你了一种构造组合 Predicates 的方便方法(与在 STL 中使用<functional>中的函数一样)。比如，下面是一个任何满足 >=0，<=100，!=50 的 Predicate。

```
Matches(AllOf(Ge(0), Le(100), Ne(50)))
```

Using Matchers in Google Test Assertions

因为 Matchers 基本上就是 Predicates，所以这就提供了一种在 Google Test 中使用它们的好方法。它叫 ASSERT_THAT 和 EXPECT_THAT：

```
  ASSERT_THAT(value, matcher);  // Asserts that value matches matcher.
  EXPECT_THAT(value, matcher);  // The non-fatal version.
```

例如，在 Google Test 中你可以写：

```
#include "gmock/gmock.h"

using ::testing::AllOf;
using ::testing::Ge;
using ::testing::Le;
using ::testing::MatchesRegex;
using ::testing::StartsWith;
...

  EXPECT_THAT(Foo(), StartsWith("Hello"));
  EXPECT_THAT(Bar(), MatchesRegex("Line \\d+"));
  ASSERT_THAT(Baz(), AllOf(Ge(5), Le(10)));
```

上面的代码(正如你所猜测的)执行 Foo()，Bar()，和 Baz()，并验证：

- Foo()返回一个以”Hello”开头的字符串。
- Bar()返回一个匹配”Line \\d+”的正则表达式。
- Baz()返回一个在[5, 10]区间内的数字。

这些宏带来的好处是它们*读起来像是*英语。它们也会产生提示消息。比如，如果第一个

EXPECT_THAT 失败，消息会类似下面的：

```
Value of: Foo()
  Actual: "Hi, world!"
Expected: starts with "Hello"
```

荣誉：(ASSERT|EXPECT)_THAT 的想来是从 Hamcrest 中获取的，它以 assertThat 加入 JUnit 中。

Using Predicates as Matchers

Google Mock 提供了一系列的内置 Matchers。如果发现它们还是不够，你可以用一个任意的一元 Predicate 函数或是仿函数作为一个 Matcher，只要它能接受你想用的类型。你就可以将这个 Predicate 入到 Truly()函数中，比如：

```
using ::testing::Truly;

int IsEven(int n) { return (n % 2) == 0 ? 1 : 0; }
...

// Bar() must be called with an even number.
EXPECT_CALL(foo, Bar(Truly(IsEven)));
```

注意 Predicate 函数/仿函数不需要一定返回 bool 类型。它只要求返回值可以用于 if (condition)语句中的 condition。

Matching Arguments that Are Not Copyable

当你设置一个 EXPECT_CALL(mock_obj, Foo(bar))时，Google Mock 会保存 bar 的一个拷贝。当 Foo()被调用时之后时，Google Mock 会比较传递给 Foo 的参数和所保存的 bar 的拷贝。通过这种方式，你不需要担心在 EXPECT_CALL()执行之后 bar 被修改了或是被销毁了。当你使用如 Eq(bar)，Le(bar)等等 Matcher 时也是这样。

但如果 bar 对象不能拷贝(比如，没有拷贝构造函数)？你可以定义自己的 Matcher 并将它放到 Truly()中，前几小节已经介绍过如何去做了。或是如果你自己可以保证 bar 不会在调用 EXPECT_CALL 之后改变，这样你可以轻松点。只需要告诉 Google Mock 它应该保存 bar 的引用 ，而不是去拷贝它。下面是一个例子：

```
using ::testing::Eq;
using ::testing::ByRef;
using ::testing::Lt;
...
// Expects that Foo()'s argument == bar.
```

```
EXPECT_CALL(mock_obj, Foo(Eq(ByRef(bar))));

// Expects that Foo()'s argument < bar.
EXPECT_CALL(mock_obj, Foo(Lt(ByRef(bar))));
```

切记 :如果你这样做 ,不要在调用 EXPECT_CALL 之后改变 bar 对象 ,否则结果是未定义的。

Validating a Member of an Object

通常 Mock 函数将一个对象的引用作为参数。当匹配这个参数时，你可能不想将整个对象与一个固定的对象比较，因为这样过于精确了。相反，你可能想验证几个特定的对象成员或是几个特定的 getter 函数的结果。你可以用 Field()和 Property 来实现这个功能。具体地讲：

```
Field(&Foo::bar, m)
```

这是一个匹配 Foo 对象的 bar 成员满足 Machter m 的一个 Matcher。

```
Property(&Foo::baz, m)
```

这是一个匹配 Foo 对象的 baz()函数返回的值满足 Matcher m 的一个 Matcher。

例如：

```
Field(&Foo::number, Ge(3))
Property(&Foo::name, StartsWith("John "))
```

分别表示：

匹配 x.number >=3 的 x 对象。

匹配 x.name()以”John ”开头的 x 对象。

注意，在 Property(&Foo::baz, ...)中，函数 baz()必须是无参的，而且需要声明为 const。

随便提一下，Field()和 Property()同样可以匹配指向对象的普通指针，比如：

```
Field(&Foo::number, Ge(3))
```

它的意思是匹配一个 p->number>=3 的普通指针 p ,如果 p 是 NULL ,匹配总是会失败。

如果你想一次验证多个成员呢？切记你可以使用 AllOf()。

Validating the Value of Pointed to by a Pointer Argument

C++函数经常使用指针型参数。你可以用如 NULL ， NotNULL()以及其它的一些比较

Matcher去匹配一个指针，但如果你想通过指针所指向的值去匹配呢？你可以用Pointee(m) Matcher。

Pointee(m)当且仅当指针指向的值匹配 m 时才匹配一个指针。比如：

```
using ::testing::Ge;
using ::testing::Pointee;
...
EXPECT_CALL(foo, Bar(Pointee(Ge(3))));
```

上面的代码会在传入 Bar 函数的指针参数所指向的值大于 3 才匹配。

Pointee()的一个亮点是它将 NULL 指针视为匹配失败，所以你可以写 Pointee(m)而不用写：

```
  AllOf(NotNull(), Pointee(m))
```

以这种方式来避免 NULL 使你的测试崩溃。

是否我已经告诉你 Pointee()还可以使用智能指针(linked_ptr，shared_ptr，scoped_ptr，等等)？

如果一个指向指针的指针呢？你可以嵌套使用 Pointee 来匹配值。比如，Pointee(Pointee(Lt(3)))匹配一个指向一个指向一个大于 3 的指针的指针(好绕呀)。

Testing a Certain Property of an Object

有时你想指定一个对象参数有某种属性，但又没有现有的 Matcher 来指定。如果你想要一个好的错误信息，你需要定义一个 Matcher。如果你想简单粗暴地解决，你可以用写一个普通函数来解决这个问题。

例如你有一个接受 Foo 类型对象的 Mock 函数，Foo 有一个 int bar()函数一个 int baz()函数，并且你想限定参数对象的 bar()的值加上 baz()的值等于某个值。你可以像下面这样做：

```
using ::testing::MatcherInterface;
using ::testing::MatchResultListener;

class BarPlusBazEqMatcher : public MatcherInterface<const Foo&> {
public:
explicit BarPlusBazEqMatcher(int expected_sum)
  : expected_sum_(expected_sum) {}

virtual bool MatchAndExplain(const Foo& foo,
                         MatchResultListener* listener) const {
return (foo.bar() + foo.baz()) == expected_sum_;
```

```
}

virtual void DescribeTo(::std::ostream* os) const {
*os << "bar() + baz() equals " << expected_sum_;
}

virtual void DescribeNegationTo(::std::ostream* os) const {
*os << "bar() + baz() does not equal " << expected_sum_;
}
private:
const int expected_sum_;
};

inline Matcher<const Foo&> BarPlusBazEq(int expected_sum) {
return MakeMatcher(new BarPlusBazEqMatcher(expected_sum));
}

...

EXPECT_CALL(..., DoThis(BarPlusBazEq(5)))...;
```

Matching Containers

有时候 Mock 函数中的参数是 STL 容器(比如：list，vector，map，...)，你可能想去匹配参数 ,因为大多 STL 容器支持==操作符 ,所以你可以写 Eq(expected_container)，或是直接就写 expected_container 去匹配一个容器。

可能有时你想更灵活一些(比如，你想第一个元素精确匹配，第二个元素是一个正数，等等)。同时，用于测试的容器通常都只有很少一些元素，再说定义一个期望的容器也有些麻烦。

你可以在下面的情况中用 ElementsAre() Matcher：

```
using ::testing::_;
using ::testing::ElementsAre;
using ::testing::Gt;
...

MOCK_METHOD1(Foo, void(const vector<int>& numbers));
...

EXPECT_CALL(mock, Foo(ElementsAre(1, Gt(0), _, 5)));
```

上面 Matcher 是指 container 必须有 4 个元素，分别是 1，大于 0，任意值，和 5。

重载的 ElementsAre()可以取 0 到 10 个参数。如果你需要指定更多参数，你可以把它们放到 C 风格的数组中并且 ElementsAreArray()：

```
using ::testing::ElementsAreArray;
...

// ElementsAreArray accepts an array of element values.
const int expected_vector1[] = { 1, 5, 2, 4, ... };
EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector1)));

// Or, an array of element matchers.
Matcher<int> expected_vector2 = { 1, Gt(2), _, 3, ... };
EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector2)));
```

如果是数组需要动态创建的情况(所以数组的大小不可能在编译时知道)，你可以给 ElementsAreArray()一个附加的参数指定数组的大小：

```
using ::testing::ElementsAreArray;
...
int* const expected_vector3 = new int[count];
... fill expected_vector3 with values ...
EXPECT_CALL(mock, Foo(ElementsAreArray(expected_vector3, count)));
```

技巧：

ElementsAre*()可以用于任意实现了 STL iterator 概念的容器(比如它有一个 const_iterator 并支持 begin()和 end())并且支持 size()，它不仅支持 STL 中的容器，也支持任何满意上述两个条件的任何容器。

你可以用嵌套的 ElementAre*()去匹配嵌套的(多维)容器。

如果容器是通过指针而不是引用传递的，你只需要写 Pointee(ElementAre*(...))。

顺序对于 ElementsAre*()是有影响的。所以不要将它用于顺序是不确定的容器(比如，hash_map)。

Sharing Matchers

本质上，一个 Google Mock Matcher 对象包含一个指向引用计数的对象。拷贝 Matchers 是允许的并且很高效，因为只是指针被拷贝了。当最后一个引用实现对象的 Matcher 生命结束时，实现对象也会被释放。

所以，如果你有一些复杂的 Matcher，你想重复使用，是不需要每次都创建一个的。只需要将它赋值给另一个 Matcher 变量，并使用那个变量！比如：

```
  Matcher<int> in_range = AllOf(Gt(5), Le(10));
```

```
... use in_range as a matcher in multiple EXPECT_CALLs ...
```

Setting Expectations

Ignoring Uninteresting Calls

如果你对一个 Mock 函数如何被调用不感兴趣，你可以不对它指定任何事。如果你这样做了，当这个函数被调用时，Google Mock 会使用它的默认行为使测试可以得以进行下去。如果你不太喜欢这个默认的行为，你可以用 DefaultValue<T>::Set()(在这个文档的后面会讲到)或是 ON_CALL 去覆盖默认行为。

请注意一旦你对某个 Mock 函数表现出了兴趣(通过 EXPECT_CALL())，所在对它的调用都需要匹配某个期望。如果这个函数被调用了，但参数没有匹配任何一个 EXPECT_CALL 语句，它将会产生一个错误。

Disallowing Unexpected Calls

如果一个 Mock 函数根本不应该被调用，可以明确地指出：

```
using ::testing::_;
...
EXPECT_CALL(foo, Bar(_))
  .Times(0);
```

如果对一个函数的某些调用是允许的，其它的调用则不行，则可以列出所有期望的调用：

```
using ::testing::AnyNumber;
using ::testing::Gt;
...
EXPECT_CALL(foo, Bar(5));
EXPECT_CALL(foo, Bar(Gt(10)))
  .Times(AnyNumber());
```

如果一个调用不匹配任一 EXPECT_CALL()，刚它会产生一个错误。

Expecting Ordered Calls

尽管在 Google Mock 尝试匹配一个设置了期望的函数时，会优先匹配先定义的 EXPECT_CALL 语句，但默认的调用并不是必须以 EXPECT_CALL()所写的顺序进行匹配。比如，如果参数匹配了第三个 EXPECT_CALL，但没有匹配前两个 EXPECT_CALL，那么就会使用第三个期望。

如果你更好所有调用都以期望的顺序进行，将 EXPECT_CALL()语句放到一个 InSequence 对象的生命周期中：

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

{
   InSequence s;

   EXPECT_CALL(foo, DoThis(5));
   EXPECT_CALL(bar, DoThat(_))
      .Times(2);
   EXPECT_CALL(foo, DoThis(6));
}
```

在这个例子中，我们期望调用以顺序如下：先调用 foo.DoThis(5)，然后两次参数为任意的 bar.DoThat()调用，最后调用一次 foo.DoThis()。如果调用的顺序与上面不符，则 Google Mock 会报告一个错误。

Expecting Partially Ordered Calls

有时要求所有的调用都以一个预定义的顺序会导致测试脆弱。比如，也许我们会关心 A 在 B 和 C 之前调用，但不关心 B 和 C 的调用顺序先后。在这种情况下，测试应当反应我们真正的意图，而不是写一个约束过强的语句。

Google Mock 在调用上设置一个顺序的任意 DAG(directed acyclic graph 有向无环图)。表达有 DAG 的一种方式是用 EXPECT_CALL 的 After 子句。

另一种方法是通过 InSequence()子句(不是 InSequence 类)，这是我们从 jMock 2 中借鉴而来的。它比之 After()稍失灵活，但当你有一长串顺序调用之时会更方便，因为它不要求你对长串中的期望都起一个不同的名字，下面是它如何工作的：

如果我们视 EXPECT_CALL()语句为图中的结点，添加一条从结点 A 到结点 B 的边，意思是 A 必须先于 B 出现，我们可以得到一个 DAG。如果 DAG 被分解成了单独的顺序，我们只需要知道每个 EXPECT_CALL 在顺序中的位置，我们就可以重构出原来的 DAG。

因此要为指定在期望之上的部分有序我们需要做两件事 :第一定义一些 Sequence 对象，然后指明 Sequence 在 DAG 中的部分。在同一顺序(sequence)中的期望必须以定义的先后(order)出现。比如：

```
using ::testing::Sequence;

Sequence s1, s2;

EXPECT_CALL(foo, A())
```

```
  .InSequence(s1, s2);
EXPECT_CALL(bar, B())
  .InSequence(s1);
EXPECT_CALL(bar, C())
  .InSequence(s2);
EXPECT_CALL(foo, D())
  .InSequence(s2);
```

以上代码所指定的顺序为(其中 s1 是 A->B，s2 是 A->C->D)：

```
      +---> B
      |
  A ---|
      |
      +---> C ---> D
```

这意味着 A 必须先于 B 和 C 出现，并且 C 必须在 D 之前出现，并除此之外没有其它的顺序要求。

Controlling When an Expectation Retires

当一个 Mock 函数被调用时，Google Mock 只考虑那些仍然有效的期望。一个期望在创建之时是有效的，当在它之上发生一次调用后，它就变为失效的了。比如，在：

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

Sequence s1, s2;

EXPECT_CALL(log, Log(WARNING, _, "File too large."))     // #1
  .Times(AnyNumber())
  .InSequence(s1, s2);
EXPECT_CALL(log, Log(WARNING, _, "Data set is empty."))  // #2
  .InSequence(s1);
EXPECT_CALL(log, Log(WARNING, _, "User not found."))     // #3
  .InSequence(s2);
```

只要#2 或#3 任一匹配，#1 就会失效。如果一个警告"File too large"在此之后调用，它将会产生一个错误。

注意一个期望在它饱和后不会自动失效。例如：

```
using ::testing::_;
...
EXPECT_CALL(log, Log(WARNING, _, _));                  // #1
EXPECT_CALL(log, Log(WARNING, _, "File too large."));  // #2
```

上面的代码意思是有仅只能有一个”File too large”的警告。如果第二个警告仍然是”File too large”，#2 仍然会匹配并且产生一个超出上界的错误。

如果这不是你想要的，你可以让一个期望在饱和之后就失效：

```
using ::testing::_;
...
EXPECT_CALL(log, Log(WARNING, _, _));                // #1
EXPECT_CALL(log, Log(WARNING, _, "File too large."))  // #2
  .RetiresOnSaturation();
```

Using Actions

Return References from Mock Methods

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::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：

```
using testing::ReturnPointee;
...
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));
```

Mock Side Effects

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。

如果 Mock 函数还需要返回一个值，你可以将 SetArgPointee()和 Return()放到 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(_))
```

```
    .WillOnce(SetArrayArgument<0>(names.begin(), names.end()));
```

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::_;
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::AnyNumber;
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::_;
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);
```

唯一的要求是这些函数，方法，仿函数的类型必须与 Mock 函数兼容，即后者的参数必

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？

Invoking a Function / Method / Functor Without Arguments

Invoke()在做一些比较复杂的动作时非常有用。它将 Mock 函数的参数传递给被调用的函数或是仿函数，即被调函数有完整的上下文。如果被调函数或仿函数对其中一些或全部参数不感兴趣，它可以简单地忽略它们。

但一个单元测试者通常想调用一个不带任何一个 Mock 函数参数的函数。Invoke 允许你使用一个包装函数丢弃所有的参数。不消说，这种工作是无趣和并将测试意图晦涩化。

InvokeWithoutArgs()是用来解决这个问题的。它类似 Invoke()，只是它不需要将递 Mock 函数的参数给被调者。下面是一个例子：

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

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

bool Job1() { ... }
...

MockFoo foo;
EXPECT_CALL(foo, ComplexJob(_))
  .WillOnce(InvokeWithoutArgs(Job1));

foo.ComplexJob(10);  // Invokes Job1().
```

Invoking an Argument of the Mock Function

有时一个 Mock 函数会接收一个函数指针或是一个仿函数(换言之，一个”callable”）参数，比如：

```
class MockFoo : public Foo {
 public:
  MOCK_METHOD2(DoThis, bool(int n, bool (*fp)(int)));
```

```
};
```

你也许想调用这个函数指针参数：

```
using ::testing::_;
...
MockFoo foo;
EXPECT_CALL(foo, DoThis(_, _))
  .WillOnce(...);
// Will execute (*fp)(5), where fp is the
// second argument DoThis() receives.
```

啊，你需要引用一个 Mock 函数的参数，但 C++还没有 lambda 表示式，所以你需要定义你自己的动作。:-(或是你真需要这么做吗？

嗯，Google Mock 有一个动作*特地*来解决这个问题：

```
InvokeArgument<N>(arg_1, arg_2, ..., arg_m)
```

它会调用 Mock 函数接收到的第 N 个参数，并将 arg_1，arg2，...，arg_m 作为参数。无论参数是一个函数指针或是一个仿函数，Google Mock 都可以处理。

使用它，你可以写：

```
using ::testing::_;
using ::testing::InvokeArgument;
...
  EXPECT_CALL(foo, DoThis(_, _))
      .WillOnce(InvokeArgument<1>(5));
  // Will execute (*fp)(5), where fp is the
  // second argument DoThis() receives.
```

如果一个函数是有一个参数是引用呢？没问题，把它放到 ByRef()中：

```
...
  MOCK_METHOD1(Bar, bool(bool (*fp)(int, const Helper&)));
...
using ::testing::_;
using ::testing::ByRef;
using ::testing::InvokeArgument;
...

MockFoo foo;
Helper helper;
...
EXPECT_CALL(foo, Bar(_))
  .WillOnce(InvokeArgument<0>(5, ByRef(helper)));
// ByRef(helper) guarantees that a reference to helper, not a copy
```

```
of it,
// will be passed to the callable.
```

如果函数指针接收的是引用参数，但我们没有将参数放到 ByRef()中呢？那么 InvokeArgument()会拷贝这个参数，将传递拷贝后的值的引用给函数指针，而不是原来值的引用。这在参数是一个临时变量时特别方便：

```
...
  MOCK_METHOD1(DoThat, bool(bool (*f)(const double& x, const string& s)));
...
MOCK_METHOD1(DoThat, bool(bool (*f)(const double& x, const string&
s)));
...
using ::testing::_;
using ::testing::InvokeArgument;
...

MockFoo foo;
...
EXPECT_CALL(foo, DoThat(_))
    .WillOnce(InvokeArgument<0>(5.0, string("Hi")));
// Will execute (*f)(5.0, string("Hi")), where f is the function
pointer
// DoThat() receives.  Note that the values 5.0 and string("Hi")
are
// temporary and dead once the EXPECT_CALL() statement finishes.
Yet
// it's fine to perform this action later, since a copy of the values
// are kept inside the InvokeArgument action.
```

Ignoring an Action's Result

有时你有一个返回值的动作，但你需要一个返回 void 的动作(也许你想在一个返回 void 的 Mock 函数中用它，或是它在 DoAll()中要用它，但它不是 DoAll()中最后一个)。IgnoreResult()允许你实现这个功能。比如：

```
using ::testing::_;
using ::testing::Invoke;
using ::testing::Return;

int Process(const MyData& data);
string DoSomething();

class MockFoo : public Foo {
```

```
 public:
  MOCK_METHOD1(Abc, void(const MyData& data));
  MOCK_METHOD0(Xyz, bool());
};
...

MockFoo foo;
EXPECT_CALL(foo, Abc(_))
// .WillOnce(Invoke(Process));
// The above line won't compile as Process() returns int but Abc()
needs
// to return void.
  .WillOnce(IgnoreResult(Invoke(Process)));

EXPECT_CALL(foo, Xyz())
  .WillOnce(DoAll(IgnoreResult(Invoke(DoSomething)),
  // Ignores the string DoSomething() returns.
                Return(true)));
```

注意你**不能**将 IgnoreResult()用在一个已经是返回 void 的动作上。如果你这样做，你会得到一个丑陋的编译错误。

Selecting an Action's Arguments

假使你有一个 Mock 函数 Foo()，它接受七个参数，并且你想在 Foo 调用时使用一个自定义的动作。但问题是，这个自定义的动作只有三个参数：

```
using ::testing::_;
using ::testing::Invoke;
...
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;
}
...

EXPECT_CALL(mock, Foo(_, _, _, _, _, _, _))
    .WillOnce(Invoke(IsVisibleInQuadrant1));     //   Uh,   won't
compile. :-(
```

为了取悦编译器，你可以定义一个配接器，它有着与 Foo()相同的定义，然后用它调用自定义动作：

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

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);
}
...

EXPECT_CALL(mock, Foo(_, _, _, _, _, _, _))
    .WillOnce(Invoke(MyIsVisibleInQuadrant1));  // Now it works.
```

但这要写不笨拙吗？

Google Mock 提供了一个通用的动作配接器，所以你可以把时间用到更重要的事情上去，而不是写你自己的配接器。下面是它的语法：

```
WithArgs<N1, N2, ..., Nk>(action)
```

它创建一个动作，将 Mock 函数的参数传给内部的动作，用 WithArgs，我们前面的例子可以写为：

```
using ::testing::_;
using ::testing::Invoke;
using ::testing::WithArgs;
...
EXPECT_CALL(mock, Foo(_, _, _, _, _, _, _))
  .WillOnce(WithArgs<0, 2, 3>(Invoke(IsVisibleInQuadrant1)));
  // No need to define your own adaptor.
```

为了更好的可读性，Google Mock 提供给你了：

- WithoutArgs(action)当内部动作不接受参数
- WithArg<N>(action)(在 Arg 后没有 s)当内部动作接受一个参数。

正如你所认识到的，InvokeWithoutArgs(...)只是 WithoutArgs(Invoke(...))的语法糖。

这里有几个小提示：

- 在 WithArgs 内部的动作并不一定要是 Invoke()，它可以是任意的。

- 在参数列表中的参数可以重复的，比如 WithArgs<2,3,3,5>(...)。
- 你可以改变参数的顺序，比如 WithArgs<3, 2, 1>(...)。
- 所选的参数类型并不一定要完全匹配内部动作的定义。只要它们可以隐式地被转换成内部动作的相应参数就可以了。例如 ,如果 Mock 函数的第 4 个参数是 int ,而 my_action 接受一个 double 参数，WithArg<4>(my_action)可以工作。

Ignoring Arguments in Action Functions

Selecting-an-action's-arguments 中介绍了一种使参数不匹配的动作和 Mock 函数结合使用的方法。但这种方法的缺点是要将动作封装到 WithArgs<...>()中，这会使测试者感到麻烦。

如果你定义要用于 Invoke*的一个函数，方法，或是仿函数，并且你对它的一些函数不感兴趣，另一种做法是声明你不感兴趣的参数为 Unused。这会吏定义更清爽，并在不感兴趣的参数发生变化时更健壮。而且它可以增加一个动作函数被重用的可能性。比如，有：

```
MOCK_METHOD3(Foo, double(const string& label, double x, double
y));
MOCK_METHOD3(Bar, double(int index, double x, double y));
```

你除了可以像下面一样写：

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

double DistanceToOriginWithLabel(const string& label, double x,
double y) {
  return sqrt(x*x + y*y);
}

double DistanceToOriginWithIndex(int index, 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));
```

你还可以写：

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

```
using ::testing::Invoke;
using ::testing::Unused;

double DistanceToOrigin(Unused, double x, double y) {
  return sqrt(x*x + y*y);
}
...

EXEPCT_CALL(mock, Foo("abc", _, _))
  .WillOnce(Invoke(DistanceToOrigin));
EXEPCT_CALL(mock, Bar(5, _, _))
  .WillOnce(Invoke(DistanceToOrigin));
```

Sharing Actions

如匹配器一样，Google Mock 动作对象中也有一个指针指向引用计数的实现对象。所以拷贝动作是允许的并且也是高效的。当最后一个引用实现对象的动作死亡后，现实对象会被 delete。

如果你有一些想重复使用的复杂动作。你也许不想每次都重新产生一次。如果这个动作没有一个内部状态(比如 :它在每次调用都做相同的事)，你可以将它赋值给一个动作变量，以后就可以重复使用这个变量了，比如：

```
Action<bool(int*)> set_flag = DoAll(SetArgPointee<0>(5),
                                    Return(true));
... use set_flag in .WillOnce() and .WillRepeatedly() ...
```

但是，如果一个动作有自己的状态，那你共享这个动作对象时，你也许会得到一些意外的结果。假设你有一个动作工厂 IncrementCounter(init)，它创建一个动作，这个动作中的计数器初始值是 init，每次调用增加计数器的值并返回计数器值，使用从相同的语句中产生的两个动作和使用一个共享动作会产生不同的行为。比如：

```
EXPECT_CALL(foo, DoThis())
  .WillRepeatedly(IncrementCounter(0));
EXPECT_CALL(foo, DoThat())
  .WillRepeatedly(IncrementCounter(0));
foo.DoThis();  // Returns 1.
foo.DoThis();  // Returns 2.
foo.DoThat();  // Returns 1 - Blah() uses a different
           // counter than Bar()'s.
```

相较：

```
Action<int()> increment = IncrementCounter(0);
```

```
EXPECT_CALL(foo, DoThis())
  .WillRepeatedly(increment);
EXPECT_CALL(foo, DoThat())
  .WillRepeatedly(increment);
foo.DoThis();  // Returns 1.
foo.DoThis();  // Returns 2.
foo.DoThat();  // Returns 3 - the counter is shared.
```

Misc Recipes on Using Google Mock

Making the Compilation Faster

无论你相信与否，编译一个 Mock 类的*大部分时间*都花费在产生它的构造函数和析构函数上了，因为它们要做很多的任务(比如，对期望的验证)。更严重的是，有不同函数声明的 Mock 函数，它们的构造函数/析构函数需要由编译器分别产生，所以，如果你 Mock 许多不同类型的函数，编译你的 Mock 类会非常慢。

如果你现在发现编译很慢，你可以将 Mock 类的构造函数/析构函数移出 Mock 类，将它们放到.cpp 文件中。这样做后，即使你在多个文件中#include 你的 Mock 文件，编译器只用产生一次 constructor 和 destructor，这样做编译的更快。

让我们以一个例子说明一下，下面是一个原有的 Mock 类：

```
// File mock_foo.h.
...
class MockFoo : public Foo {
 public:
  // Since we don't declare the constructor or the destructor,
  // the compiler will generate them in every translation unit
  // where this mock class is used.

  MOCK_METHOD0(DoThis, int());
  MOCK_METHOD1(DoThat, bool(const char* str));
  ... more mock methods ...
};
```

修改后，变为：

```
// File mock_foo.h.
...
class MockFoo : public Foo {
 public:
  // The constructor and destructor are declared, but not defined, here.
  MockFoo();
```

```
  virtual ~MockFoo();

  MOCK_METHOD0(DoThis, int());
  MOCK_METHOD1(DoThat, bool(const char* str));
  ... more mock methods ...
};
```

和：

```
// File mock_foo.cpp.
#include "path/to/mock_foo.h"

// The definitions may appear trivial, but the functions actually do a
// lot of things through the constructors/destructors of the member
// variables used to implement the mock methods.
MockFoo::MockFoo() {}
MockFoo::~MockFoo() {}
```

Forcing a Verification

当你的 Mock 对象销毁的时候，它会自动检查所有的期望是否满足，如果没满足，会产生一个 Google Test 失败。这种方式让你可以少去操心一件事。但是如果你不确定你的 Mock 对象是否会被销毁时，你还是要操心了。

一个 Mock 对象怎么会最终没有被销毁呢？嗯，它可以是在由被测试的代码在堆上分配的。假设代码中有一个 bug，它没能正常地 delete Mock 对象，你最终可能会在测试中有 bug 时，让测试通过。

使用一个堆检查工具是一个好主意，可以减少了一些担心，但它的实现不是 100%可靠的。所以有时你想在一个 Mock 对象(希望如些) 销毁前，强制 Google Mock 去检查它。你可以写：

```
TEST(MyServerTest, ProcessesRequest) {
  using ::testing::Mock;

  MockFoo* const foo = new MockFoo;
  EXPECT_CALL(*foo, ...)...;
  // ... other expectations ...

  // server now owns foo.
  MyServer server(foo);
  server.ProcessRequest(...);

  // In case that server's destructor will forget to delete foo,
  // this will verify the expectations anyway.
```

```
  Mock::VerifyAndClearExpectations(foo);
}  // server is destroyed when it goes out of scope here.
```

提示：Mock::VerifyAndClearExpectations()函数返回一个 bool 值来标明检查是否成功(成功为 true)，所以你可以将函数放到 ASSERT_TRUE()中，如果这个断言失败，就没有必要再继续了。

Using Check Points

有时你也许也在多个检查点”重置”一个 Mock 对象：在每个检查点，你可以检查在这个 Mock 对象上的所有设置的期望是否满足，并且你可以设置在它上面设置一些新的期望，就如同这个 Mock 对象是新创建的一样。这样做可以让你让你的测试”分段”地使用 Mock 对象。

其中一个使用场景是在你的测试的 SetUp()函数中，你也许想借助 Mock 对象，将你测试的对象放到一个特定的状态。如果在一个合适的状态后，你清除所在 Mock 对象上的所有期望，这样你可以在 TEST_F 中设置新的期望。

正如你也许会发现的一样，Mock::VerifyAndClearExpectations()函数会在这帮助你。或是如果你正用 ON_CALL()设置在这个 Mock 对象上的默认动作，并且想清除这个 Mock 对象上的默认动作，就用 Mock::VerifyAndClear(&mock_object)。这个函数会做如 Mock::VerifyAndClearExpectations(&mock_object)相同的工作，并返回相同的 bool 值，但它还会清除设置在 mock_object 上设置的 ON_CALL()语句。

另一个可以达到相同效果的技巧是将期望放到序列(sequence)中，在指定的位置将调用放到无效果的(dummy)检查点函数中。然后你可以检查 Mock 函数在指定时间的行为了。比如，你有下面的代码：

```
Foo(1);
Foo(2);
Foo(3);
```

你想验证 Foo(1)和 Foo(3)都调用了 mock.Bar(“a”)，但 Foo(2)没有调用任何函数，你可以写：

```
using ::testing::MockFunction;

TEST(FooTest, InvokesBarCorrectly) {
  MyMock mock;
  // Class MockFunction<F> has exactly one mock method.  It is named
  // Call() and has type F.
  MockFunction<void(string check_point_name)> check;
  {
    InSequence s;
```

```
    EXPECT_CALL(mock, Bar("a"));
    EXPECT_CALL(check, Call("1"));
    EXPECT_CALL(check, Call("2"));
    EXPECT_CALL(mock, Bar("a"));
  }
  Foo(1);
  check.Call("1");
  Foo(2);
  check.Call("2");
  Foo(3);
}
```

期望指明了第一个 Bar(“a”)必须在检查点”1”之前发生 ,第二个 Bar(“a”)必须在检查点”2”之后发生，并且两个检查点之间不应该发生任何事。这种使用检查点的明确写法很容易指明哪个 Foo 函数调用的 Bar(“a”)。

Mocking Destructors

有时你想明确一个 Mock 对象在指定时间销毁，比如，在 bar->A()调用之后，但在 bar->()调用之前。我们已经知道可以指定 Mock 函数调用的顺序，所以我们需要做的是 Mock 所 Mock 函数的析构函数。

这听起来简单，但一个问题：析构函数是一个特殊的函数，它有着特殊的语法和语义，MOCK_METHOD0 对它是无效的：

```
  MOCK_METHOD0(~MockFoo, void());  // Won't compile!
```

好消息是你可以用一个简单的模式来达到相同的效果。首先，在你的 Mock 类中添加一个 Mock 函数 Die()，并在析构函数中调用它，如下：

```
class MockFoo : public Foo {
  ...
  // Add the following two lines to the mock class.
  MOCK_METHOD0(Die, void());
  virtual ~MockFoo() { Die(); }
};
```

(如果 Die()与已有符号冲突，选另一个名字)，现在，我们将测试一个 MockFoo 对象析构时的问题，转化为测试当 Die 函数被调用时的问题了：

```
MockFoo* foo = new MockFoo;
MockBar* bar = new MockBar;
...
{
    InSequence s;
```

```
    // Expects *foo to die after bar->A() and before bar->B().
    EXPECT_CALL(*bar, A());
    EXPECT_CALL(*foo, Die());
    EXPECT_CALL(*bar, B());
}
```

Using Google Mock and Threads

重要提示：我们这节所描述的**只有**在 Google Mock 是线程安全的平台上才是成立的。现在只有支持 pthreads 库的平台(这包括 Linux 和 Mac)才可以。要让它在其它平台上线程安全，我们只需要在”gtest/internal/gtest-port.h”中实现一些同步操作。

在一个单元测试中，如果你可以将一块代码独立出来在单线程环境中测试是最好的。这可以防止竞争和死锁，并使你 debug 你的测试容易的多。

但许多程序是多线程的，并有时我们需要在多线程环境中测试它，Google Mock 也提供了这个功能。

回忆使用一个 Mock 的步骤：

1. 创建一个 Mock 对象 foo。

2. 用 ON_CALL() 和 EXPECT_CALL()设置它的默认行为和期望。

3. 测试调用 foo 的函数的代码。

4. 可选的，检查和重置 mock。

5. 你自己销毁 mock，或是让测试代码销毁 mock，析构函数会自动检查是否满足。

如果你能遵循下面的简单规则，你的 Mock 和线程可以幸福地生活在一起：

- 在单线程中执行你的测试代码(相对于被测试代码)。这可以保证你的测试容易被跟踪。
- 显然，你可以在第一步中不用锁。
- 当你做第 2 步和第 5 步，要保证没有其它线程访问 foo。很显然，不是吗？
- 第 3 步和第 4 步可以在单线程或是多线程中进行–随便你。Google Mock 会去处理锁，所以你不需要再做什么事，除非是你测试需要。

如果你违反了这些规则(比如 如果你在其它线程正在访问的一个 Mock 上测试了期望)，你会得到一个不确定的行为。这不会是什么令你开心的事，所以别去尝试。

Google Mock 保证一个 Mock 函数的动作会在调用这个 Mock 的线程中进行 。比如：

```
EXPECT_CALL(mock, Foo(1))
```

```
  .WillOnce(action1);
EXPECT_CALL(mock, Foo(2))
  .WillOnce(action2);
```

如果在线程 1 中被调用 Foo(1)，且在线程 2 中调用 Foo(2)，Google Mock 会在线程 1 中执行 action1，在线程 2 中执行 action2。

Google Mock *不会*对多个线程设置动作的顺序(这样做可能会产生死锁，因为动作可能需要配合)。这意味着在上面的例子中执行 action1 和 action2 可能会交错。如果这样不符合你的意思，你应该对 action1 和 action2 加入同步逻辑，来保证测试线程安全。

同样，要记得 DefaultValue<T>是一个全局资源，所以它会影响你程序中的所有活动的 Mock 对象。很自然的，你不会想在多线程环境中与它纠缠不清，或是你在 Mock 对象的动作正在进行时，你去使用 DefaultValue。

Controlling How Much Information Google Mock Prints

当 Google Mock 看到有潜在的错误 时(比如，一个没有设置期望的 Mock 函数被调用了)，它会打印一些警告信息，包括这个函数的参数和返回值，并希望你可以提醒你关注一下，看这里是不是的确是一个错误。

有时你对你的测试正确性比较自信，也许就不喜欢这些友好的提示。有的时候，你在 debug 你的测试，并在推敲你所测试的代码的行为，那你也许会希望看到每个 Mock 函数被调用的信息(包括参数值和返回值)。很明显，一种打印级别是不是满足所有需求的。

你可以通过–gmock_verbose=LEVEL 命令参数来设置打印级别，其中 LEVEL 是一个有三种取值的字符串。

- info：Google Mock 会打印所有的信息，包括正常的消息，警告，错误。在这种级别设置上，Google Mock 会记录所有对于 ON_CALL/EXPECT_CALL 的调用。
- warning：Google Mock 会打印警告和错误信息，这是默认的。
- error：Google 仅会打印错误。

另外，你可以在你的测试中设置打印级别，如：

```
::testing::FLAGS_gmock_verbose = "error";
```

现在，请明智地选择打印级别，让 Google Mock 更好地为你服务。

Running Tests in Emacs

如果你在 Emacs 中运行你的测试，错误相关的 Google Mock 和 Google Test 源文件位置会被高亮标记。只用其中一行上敲回车就会跳到相应的行。或是你可以敲 C-x ` 跳到下

一个错误。

为了操作更简单，你可以将加下面几行加入~/.emacs 文件中：

```
(global-set-key "\M-m"   'compile)  ; m is for make

(global-set-key [M-down] 'next-error)

(global-set-key [M-up]   '(lambda () (interactive) (next-error -1)))
```

然后你可以敲 M-m 编译，或是用 M-up/M-down 在错误提示中上下移动。

Fusing Google Mock Source Files

Google Mock 的实现包括几十个文件(包括它自己的测试)。有时你也许将它们放到几个文件中，这样你就可以更容易地把它们拷贝到一个新机器上。对于这个需求，我们提供了 Python 脚本 fuse_gmock_files.py ,它在 srcipts/目录下。假设你已经安装了 Python 2.4 或更高的版本，你进入目录后运行：

```
Python fuse_gmock_files.py OUTPUT_DIR
```

你应该会看到 OUTPUT_DIR 被创建了，并且里面有 gtest/gtest.h，gmock/gmock.h 和 gmock-gtest-all.cc。这三个文件包含你需要使用 Google Mock(和 Google Test) 的所有东西。只需要把它们拷贝到任何地方，然后你就可以开始用 Mock 写测试了。你可以用 sripts/test/Makefile 文件作为一个编译你的测试的例子。

Extending Google Mock

Writing New Matchers Quickly

MATCHER*宏系列可以很容易地用来定义自己的匹配器。语法是：

```
MATCHER(name, description_string_expression) { statements; }
```

这个宏会定义一个名为 name 的匹配器，这个匹配器执行 statements 语句，statetements 必须返回一个 bool 值 ,以来表示这次匹配是否成功。在 statements 内部，你可以用 arg 来表示被匹配的值，这个值的类型用 arg_type 表示。

Description_string 是一个字符串，用来描述这个匹配器的行为，并且在匹配失败的时候产生失败信息。它能(并且应该)对非逻辑进行判断(::test::Not)，产生对应的错误描述。

为了方便起见，我们允许描述字符串为空，在这种情况下 Google Mock 会用匹配器的名字中的单词作为描述。

比如：

```
MATCHER(IsDivisibleBy7, "") { return (arg % 7) == 0; }
```

允许你写：

```
// Expects mock_foo.Bar(n) to be called where n is divisible by
7.
EXPECT_CALL(mock_foo, Bar(IsDivisibleBy7()));
```

或是：

```
using ::testing::Not;
...
EXPECT_THAT(some_expression, IsDivisibleBy7());
EXPECT_THAT(some_other_expression, Not(IsDivisibleBy7()));
```

当上面的断言失败时，它们会打印下面的信息：

```
Value of: some_expression
Expected: is divisible by 7
Actual: 27
...
Value of: some_other_expression
Expected: not (is divisible by 7)
Actual: 21
```

其中描述”is divisible by 7”和”not (is divisiable by 7)”是通过 IsDivisibleBy7 这个匹配器名字自动产生的。

正如你所注意到的，自动产生的描述(特别是由非逻辑产生的)并不是那么好。你可以自定义描述。

```
MATCHER(IsDivisibleBy7, std::string(negation ? "isn't" : "is") +
                        " divisible by 7") {
  return (arg % 7) == 0;
}
```

或者，你可以将更多的信息用一个隐藏变量 result_listener 输出，来解释匹配结果。比如，一个更好的 IsDivisibleBy7 的更好定义是：

```
MATCHER(IsDivisibleBy7, "") {
   if ((arg % 7) == 0)
      return true;

   *result_listener << "the remainder is " << (arg % 7);
   return false;
}
```

有了这个定义，上面断言会给出一个更好的提示：

```
Value of: some_expression
Expected: is divisible by 7
Actual: 27 (the remainder is 6)
```

你应该让 MatchAndExplain()打印其它附加信息，这些信息可以帮助一个用户理解匹配结果。注意它可以在成功的情况下解释为什么匹配成功(除非它是显然的)-这在 Not 内部的匹配器中是很有效的。没有必要打印参数本身，因为 Google Mock 已经为你打印了。

注意：

1. 所匹配的值的类型(arg_type)是由你使用匹配器的上下文决定的，它是由编译器提供给你的，所以你不用操心如何去定义它(你也没法定义)。这样允许匹配器是多形的(即支持多种类型的)。比如，IsDivisibleBy7()可以用于匹配任何支持 arg % 7 == 0 转换为 bool 的类型。在上面的 Bar(IsDivisibleBy7())例子中，如果 Bar()接受 int 参数，arg_type 就是 int，如果它接受 unsigned long 整形，arg_type 就是 unsigned long，等等。
2. Google Mock 不保证匹配器何时和被调用多少次。所以匹配器的逻辑必须是纯功能性的(比如，它不能有任何副作用，并且结果也不能依赖于匹配的值和匹配器参数之外的东西)。无论你如何定义，这个条件是你在定义一个匹配器时必须要满足的 (比如，用下面章节介绍的方法)。特别是，一个匹配器决不能调用一个 Mock 函数，因为这会改变 Mock 对象和 Google Mock 的状态

Writing New Parameterized Matchers Quickly

有时你想定义有一个有参数的匹配器。对于这个要求你可以使用宏：

```
MATCHER_P(name, param_name, description_string) { statements; }
```

其中 description_string 可以是""或是引用了 param_name 的描述。

比如：

```
MATCHER_P(HasAbsoluteValue, value, "") { return abs(arg) == value; }
```

你可以写：

```
 EXPECT_THAT(Blah("a"), HasAbsoluteValue(n));
```

这会得到下面的信息(假设 n 是 10)：

```
Value of: Blah("a")
Expected: has absolute value 10
```

```
Actual: -9
```

注意匹配器的描述和它的参数都被打印了，使信息更友好。

在匹配器定义内，你可以写 foo_type 来引用一个名为 foo 的参数类型。比如在上例 MATCHER_P(HasAbsoluteValue, Value)中，你可以用 value_type 来引用 value 的类型。

Google Mock 还提供 MATCHER_P2，MATCHER_P3，...，MATCHER_P10 以来支持多参数的匹配器：

```
MATCHER_Pk(name, param_1, ..., param_k, description_string)
{ statements; }
```

请注意匹配器的提示信息是针对匹配器一个特定的实例的，即参数是与真实值相关的。所以通常你会想要参数值成为描述的一部分。Google Mock 可以让你通过在描述字符串中引用匹配器参数来达到这个目的。

比如：

```
using ::testing::PrintToString;
MATCHER_P2(InClosedRange, low, hi,
       std::string(negation ? "isn't" : "is") + " in range [" +
       PrintToString(low) + ", " + PrintToString(hi) + "]") {
   return low <= arg && arg <= hi;
}
...
EXPECT_THAT(3, InClosedRange(4, 6));
```

会产生如下的失败信息：

```
Expected: is in range [4, 6]
```

如果你用""作为描述信息，失败信息中会包含匹配器的名字，后面跟着以元组形式打印的参数值。比如：

```
MATCHER_P2(InClosedRange, low, hi, "") { ... }
...
EXPECT_THAT(3, InClosedRange(4, 6));
```

会产生一个如下的失败信息：

```
Expected: in closed range (4, 6)
```

出于输入的方便，你可以视

```
MATCHER_Pk(Foo, p1, ..., pk, description_string) { ... }
```

为下面的简写：

```
template <typename p1_type, ..., typename pk_type>
FooMatcherPk<p1_type, ..., pk_type>
Foo(p1_type p1, ..., pk_type pk) { ... }
```

当你写 Foo(v1, ..., vk)，编译器会自己推出 v1, ..., vk 参数的类型。如果你推出的类型结果不满意，你可以指定模板参数类型，比如 Foo<long, bool>(5, false)。如前面所提到的，你不需要去指定 arg_type，因为它是由所用匹配器的上下文所决定的。

你可以将 Foo(p1, ..., pk)的结果赋给 FooMatcherPk<p1_typem, ..., pk_type>类型的变量。这在组合匹配器时会比较有用。无参的匹配器或只有一个参数的匹配器有特殊的类型：你可以将 Foo()赋值给一个 FooMatcher 类型变量，将 Foo(p)赋值给一个 FooMatcher<p_type>类型变量。

尽管你可以用引用类型来实例化匹配器模板，然而用指针传递参数通常会使你的代码更可读。但是如果你还是想通过引用传递参数，注意由匹配器产生的失败信息中的值是对象引用的值，而不是它的地址。

你可以重载不同参数个数的匹配器。

```
MATCHER_P(Blah, a, description_string_1) { ... }
MATCHER_P2(Blah, a, b, description_string_2) { ... }
```

虽然总是用 MATCHER*来定义一个新的匹配器是很有吸引力的，但你也应该考虑用 MatcherInterface 或是用 MakePolymorphicMatcher()来定义(下面会介绍)，特别是你会经常用这个匹配器的时候。尽管这些方法会花费更多的力气，但它们会给你更多的控制能力：可以控制匹配值的类型，匹配器参数，这样一般也会产生更好的编译提示信息，用这些方法以长远的目光来看是更好的选择。它们还允许重载不同参数类型(而不是仅能通过不同参数个数重载)。

Wrting New Monomorphic Matchers

一个实现了::testing::MatcherInterface<T>的 T 参数类型的匹配器可以做两种事：它判断参数 T 是否匹配匹配器，并可以描述它所匹配的类型。后一种能力可以在期望失败时给出可读的错误信息。

```
class MatchResultListener {
 public:
  ...
  // Streams x to the underlying ostream; does nothing if the ostream
  // is NULL.
  template <typename T>
  MatchResultListener& operator<<(const T& x);

  // Returns the underlying ostream.
```

```
  ::std::ostream* stream();
};

template <typename T>
class MatcherInterface {
 public:
  virtual ~MatcherInterface();

  // Returns true iff the matcher matches x; also explains the match
  // result to 'listener'.
  virtual bool MatchAndExplain(T x, MatchResultListener* listener)
const = 0;

  // Describes this matcher to an ostream.
  virtual void DescribeTo(::std::ostream* os) const = 0;

  // Describes the negation of this matcher to an ostream.
  virtual void DescribeNegationTo(::std::ostream* os) const;
};
```

如果你需要一个自定的 Matcher，但 Truly 不是一个好选择(比如，你也不会对 Truly(predicate)的提示信息不满意，或是你想让你的匹配器是多形的(接受多种类型)，如 Eq(value)一样)，你可以定义通过两步来定义你想的任何匹配器：第一步定义匹配器接口，第二步定义创建一个匹配器实例的工厂函数。第二步不是必须的，但它会使使用匹配器的语法更优雅。

比如，你可以定义一个匹配器来判断一个 int 值是否可以被 7 整除，然后使用它：

```
using ::testing::MakeMatcher;
using ::testing::Matcher;
using ::testing::MatcherInterface;
using ::testing::MatchResultListener;

class DivisibleBy7Matcher : public MatcherInterface<int> {
 public:
  virtual bool MatchAndExplain(int n, MatchResultListener*
listener) const {
    return (n % 7) == 0;
  }

  virtual void DescribeTo(::std::ostream* os) const {
    *os << "is divisible by 7";
  }

  virtual void DescribeNegationTo(::std::ostream* os) const {
```

```
    *os << "is not divisible by 7";
  }
};

inline Matcher<int> DivisibleBy7() {
  return MakeMatcher(new DivisibleBy7Matcher);
}
...

EXPECT_CALL(foo, Bar(DivisibleBy7()));
```

你可以通过输出更多的信息到 MatchAndExplan()函数中的 listener 变量来改进提示信息：

```
class DivisibleBy7Matcher : public MatcherInterface<int> {
 public:
  virtual bool MatchAndExplain(int n,
                               MatchResultListener* listener)
const {
    const int remainder = n % 7;
    if (remainder != 0) {
      *listener << "the remainder is " << remainder;
    }
    return remainder == 0;
  }
  ...
};
```

然后，EXPECT_THAT(x, DivisiableBy7())可能会产生如下的信息：

```
Value of: x
Expected: is divisible by 7
Actual: 23 (the remainder is 2)
```

Writing New Polymorphic Matchers

在前一节中的你了解了如何去写自己的匹配器。只有还有一个问题：一个用 MakeMatcher()创建的匹配器只能在参数类型确定的情况下用。如果你想要一个多形的(接受多种类型)匹配器(比如,Eq(x)可以匹配任何 value==x 的 value，value 和 x 不一定是同一类型)，你可以通过”gmock/gmock-matchers.h”学习这个技巧，但这又要了解太多。

幸运的是，大多数时间你在 MakePolymorphicMatcher()的帮助下， 很容易定义一个多形匹配器。下面是以定义 NotNull()为例：

```
using ::testing::MakePolymorphicMatcher;
```

```
using ::testing::MatchResultListener;
using ::testing::NotNull;
using ::testing::PolymorphicMatcher;

class NotNullMatcher {
 public:
  // To implement a polymorphic matcher, first define a COPYABLE
class
  // that has three members MatchAndExplain(), DescribeTo(), and
  // DescribeNegationTo(), like the following.

  // In this example, we want to use NotNull() with any pointer,
so
  // MatchAndExplain() accepts a pointer of any type as its first
argument.
  // In general, you can define MatchAndExplain() as an ordinary
method or
  // a method template, or even overload it.
  template <typename T>
  bool MatchAndExplain(T* p,
                       MatchResultListener* /* listener */) const {
    return p != NULL;
  }

  // Describes the property of a value matching this matcher.
  void DescribeTo(::std::ostream* os) const { *os << "is not
NULL"; }

  // Describes the property of a value NOT matching this matcher.
  void DescribeNegationTo(::std::ostream* os) const { *os << "is
NULL"; }
};

// To construct a polymorphic matcher, pass an instance of the class
// to MakePolymorphicMatcher().  Note the return type.
inline PolymorphicMatcher<NotNullMatcher> NotNull() {
  return MakePolymorphicMatcher(NotNullMatcher());
}
...

EXPECT_CALL(foo, Bar(NotNull()));   // The argument must be a
non-NULL pointer
```

注意：你的多形匹配器类**不需要**继承任何 `MatcherInterface` 或是其它类，它的函数

也**不需要**是虚函数。

你可以像在单形匹配器中的一样，将更多的信息输出到 MatchAndExplain()中的 listener 参数中。

Writing New Cardinalities

在 Times()中的 Cardinality 用于告诉 Google Mock 调用发生多少次。它不必是准确的。比如，你可以说 AtLeast(5)或 Between(2,4)。

如果内置的 cardinality 集合不适合你，你可以通过下面的接口定义你自己的 cardinality。

```
class CardinalityInterface {
 public:
  virtual ~CardinalityInterface();

  // Returns true iff call_count calls will satisfy this
cardinality.
  virtual bool IsSatisfiedByCallCount(int call_count) const = 0;

  // Returns true iff call_count calls will saturate this
cardinality.
  virtual bool IsSaturatedByCallCount(int call_count) const = 0;

  // Describes self to an ostream.
  virtual void DescribeTo(::std::ostream* os) const = 0;
};
```

比如，指定一个调用必须发生偶数次，你可以写：

```
using ::testing::Cardinality;
using ::testing::CardinalityInterface;
using ::testing::MakeCardinality;

class EvenNumberCardinality : public CardinalityInterface {
 public:
  virtual bool IsSatisfiedByCallCount(int call_count) const {
    return (call_count % 2) == 0;
  }

  virtual bool IsSaturatedByCallCount(int call_count) const {
    return false;
  }
```

```
  virtual void DescribeTo(::std::ostream* os) const {
    *os << "called even number of times";
  }
};

Cardinality EvenNumber() {
  return MakeCardinality(new EvenNumberCardinality);
}
...

EXPECT_CALL(foo, Bar(3));
```

Writing New Actions Quickly

如果内置的动作不适合你，并且发现用 Invoke()很不方便，你可以用 ACTION*宏系列中的宏来快速定义一个新动作，它可以像内置的动作一样用于你的代码中。

通过在命名空间中写：

```
ACTION(name) { statements; }
```

你可以定义一个执行 statements 名为 name 的动作。Statements 返回的值会作为 action 返回的值。在 statements 中，你可以通过 argK 来引用 Mock 函数的第 K 个参数(从 0 开始)。比如：

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

允许你写：

```
... WillOnce(IncrementArg1());
```

注意，你不需要指定 Mock 函数参数的类型，另外你要保证你的代码是类型安全的：如果*arg1 不支持++运算符，或是如果++(*arg1)与 Mock 函数的返回值类型不兼容，你会得到一个编译错误。

另一个例子：

```
ACTION(Foo) {
  (*arg2)(5);
  Blah();
  *arg1 = 0;
  return arg0;
}
```

定义一个动作 Foo()，它会传入 5 调用第二个参数(一个函数指针)，调用函数 Blah()，设置第一个参数指针向指向的值为 0，返回第零个参数。

以了更方便更灵活，你可以在 ACTION 中用下面预定义的符号：

argK_type	Mock 函数第 K 个(从 0 开始 0)参数的类型
args	Mock 函数所有参数组成的一个元组
args_type	Mock 函数所有参数类型组成的一个元组
return_type	Mock 函数的返回类型
function_type	Mock 函数的类型

比如，用 ACTION 为 MOCK 函数作为一个 stub 动作：

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

我们有：

Pre-defined Symbol	Is Bound To
arg0	flag 的值
arg0_type	flag 的类型 bool
arg1	ptr 的值
arg1_type	ptr 的类型 int*
args	参数元组(flag ptr)
args_type	参数元组 std::tr1::tuple<bool, int*>
return_type	返回类型 int
function_type	函数类型 int(bool, int*)

Writing New Parameterized Actions Quickly

有时你想参数化你定义的一个动作。对此我们有另一个宏：

```
ACTION_P(name, param) { statements; }
```

比如：

```
ACTION_P(Add, n) { return arg0 + n; }
```

允许你写：

```
// Returns argument #0 + 5.
... WillOnce(Add(5));
```

了为方便起来，我们用术语 arguments 表示 用于调用 Mock 函数的值，用术语 parameters 表示实例化动作的值。

现在你同样不需要提供参数(parameter)的类型。假设参数的名称为 param ,你可以用 Google Mock 定义的符号 param_type 来表示 parameter 的类型，它是由编译器推出的。比如在上面的 ACTION_P(Add, n)，你可以用 n_type 来表示 n 的类型。

Google Mock 要同样提供 ACTION_P2 ，ACTION_P3 ，等等来支持多参数(multi-parameter)动作，比如：

```
ACTION_P2(ReturnDistanceTo, x, y) {
  double dx = arg0 - x;
  double dy = arg1 - y;
  return sqrt(dx*dx + dy*dy);
}
```

可以让你写：

```
... WillOnce(ReturnDistanceTo(5.0, 26.5));
```

你可以视 ACTION 为一个退化的参数化动作，它的参数个数为 0.

你同样可以很容易的定义重载不同参数个数的动作。

```
ACTION_P(Plus, a) { ... }
ACTION_P2(Plus, a, b) { ... }
```

Restricting the Type of an Argument or Parameter in an ACTION

为了最大化简洁性和可重用性，ACTION*宏不用你提供 MOCK 函数 arguments 和动作 parameters。相反，我们让编译器帮我们推导出类型。

但有时，我们想让类型更准确一些，有几个技巧可以做到这点。比如：

```
ACTION(Foo) {
   // Makes sure arg0 can be converted to int.
   int n = arg0;
   ... use n instead of arg0 here ...
}

ACTION_P(Bar, param) {
   // Makes sure the type of arg1 is const char*.
   ::testing::StaticAssertTypeEq<const char*, arg1_type>();
```

```
    // Makes sure param can be converted to bool.
    bool flag = param;
}
```

其中 StaticAssertTypeEq 在 Google Test 中一个编译期判断两个类型是否匹配的断言。

Writing New Action Templates Quickly

有时你想给一个动作明确的模板参数，而不是由编译器推导出参数类型。ACTION_TEMPLATE()支持这个功能,它可以视为是ACTION()和ACTION_P*()的一个扩展。

语法：

```
ACTION_TEMPLATE(ActionName,
    HAS_m_TEMPLATE_PARAMS(kind1, name1, ..., kind_m, name_m),
    AND_n_VALUE_PARAMS(p1, ..., p_n)) { statements; }
```

上面定义了一个动作模板，它接受 m 个明确的模板参数和 n 个参数，其中 m 取值在 1 到 10，n 取值在 0 到 10。name_i 是第 i 个模板参数的名字，kind_i 表明它是否是一个 typename，或是一个整形常量，或是一个模板。P_i 是第 i 个参数值的名字。

比如：

```
// DuplicateArg<k, T>(output) converts the k-th argument of the
mock
// function to type T and copies it to *output.
ACTION_TEMPLATE(DuplicateArg,
                // Note the comma between int and k:
                HAS_2_TEMPLATE_PARAMS(int, k, typename, T),
                AND_1_VALUE_PARAMS(output)) {
  *output = T(std::tr1::get<k>(args));
}
```

创建一个动作模板的实现，可以写：

```
ActionName<t1, ..., t_m>(v1, ..., v_n)
```

其中 ts 是模板参数，vs 是参数值。参数类型由编译器推出。比如：

```
using ::testing::_;
...
int n;
EXPECT_CALL(mock, Foo(_, _))
  .WillOnce(DuplicateArg<1, unsigned char>(&n));
```

如果你想明确地指明参数类型，你可以提供更多的模板参数：

```
ActionName<t1, ..., t_m, u1, ..., u_k>(v1, ..., v_n)
```

其中 u_i 是 v_i 参数所期望的类型。

ACTION_TEMPLATE 和 ACTION/ACTION_P*可以通过参数个数重载，但不能通过模板参数个数重载。如果没有这个限制，下面的代码含义就不明确了：

```
OverloadedAction<int, bool>(x);
```

我们是在用一个模板参数的功能，其中 bool 是指 x 的类型呢？或是两个板板参数，其中编译器需要推出 x 的类型。

Using the ACTION Object's Type

如果你写一个返回一个 ACTION 对象的函数，你将需要知道它的类型，类型依赖于用于定义动作和参数类型的达能。规则是很简单的：

Given Definition	Expression	Has Type
ACTION(Foo)	Foo()	FooAction
ACTION_TEMPLATE(Foo, HAS_m_TEMPLATE_PARAMS(...), AND_0_VALUE_PARAMS())	Foo<t1, ..., t_m>()	FooAction<t1, ..., t_m>
ACTION_P(Bar, param)	Bar(int_value)	BarActionP<int>
ACTION_TEMPLATE(Bar, HAS_m_TEMPLATE_PARAMS(...), AND_1_VALUE_PARAMS(p1))	Bar<t1, ..., t_m>(int_value)	FooActionP<t1, ..., t_m, int>
ACTION_P2(Baz, p1, p2)	Baz(bool_value, int_value)	BazActionP2<bool, int>
ACTION_TEMPLATE(Baz, HAS_m_TEMPLATE_PARAMS(...), AND_2_VALUE_PARAMS(p1, p2))	Baz<t1, ..., t_m>(bool_value, int_value)	FooActionP2<t1, ..., t_m, bool, int>
...	...	...

注意，我们要选择不同的前缀(Action，ActionP，ActionP2，等等)用于区别有不同参数个数的动作，否则不能通过参数个数重载的动作。

Writing New Monomorphic Actions

虽然 ACTION*宏很方便，但有时它们是不合适的。比如，在前一节中介绍的技巧，不能让你直接指定 Mock 函数参数和动作参数，这通常会引发一些没有优化的错误信息，这又会困扰一些不熟悉此的用户。实现根据参数类型重载动作，需要越过重重障碍。

另一个方法是实现::testing::ActionInterface<F>，其中 F 是用于动作的 Mock 函数的类型。比如：

```
template <typename F>class ActionInterface {
public:
   virtual ~ActionInterface();

   // Performs the action.  Result is the return type of function
type
   // F, and ArgumentTuple is the tuple of arguments of F.
   //
   // For example, if F is int(bool, const string&), then Result
would
   // be int, and ArgumentTuple would be tr1::tuple<bool, const
string&>.
   virtual Result Perform(const ArgumentTuple& args) = 0;
};

using ::testing::_;
using ::testing::Action;
using ::testing::ActionInterface;
using ::testing::MakeAction;

typedef int IncrementMethod(int*);

class IncrementArgumentAction : public
ActionInterface<IncrementMethod> {
public:
   virtual int Perform(const tr1::tuple<int*>& args) {
   int* p = tr1::get<0>(args);  // Grabs the first argument.
   return *p++;
   }
};

Action<IncrementMethod> IncrementArgument() {
   return MakeAction(new IncrementArgumentAction);
}
...
```

```
EXPECT_CALL(foo, Baz(_))
  .WillOnce(IncrementArgument());

int n = 5;
foo.Baz(&n);  // Should return 5 and change n to 6.
```

Writing New Polymorphic Actions

上一节中介绍了如何定义自己的动作。这是不错，除了你需要知道动作中要用的函数类型这一点。有时这会是一个问题。比如，如果你想用在动作中用不同类型的函数(比如Return()和SetArgPointee())。

如果一个动作可以用在不同类型的 Mock 函数中，我们就称它是多形的。MakePolymorphicActions()函数模板让这种定义很容易。

```
namespace testing {

template <typename Impl>
   PolymorphicAction<Impl> MakePolymorphicAction(const Impl&
impl);

}  // namespace testing
```

举一例子，我们定义一个返回 Mock 函数参数列表中第二个参数的动作。第一步是定义一个实现类：

```
class ReturnSecondArgumentAction {
public:
   template <typename Result, typename ArgumentTuple>
   Result Perform(const ArgumentTuple& args) const {
     // To get the i-th (0-based) argument, use tr1::get<i>(args).
     return tr1::get<1>(args);
   }
};
```

实现类不需要继承任何特殊的类。重要的是它必须有一个 Perform()模板函数。这个函数模板将 Mock 函数的参数视为一个元组参数，并返回动作的结果。它可以是 const 的，也可以不是，但它必须有且仅有一个模板参数，它是结果类型。另句话说，你必须可以调用 Perform<R>(args) ,其中 R 是 Mock 函数的返回类型 ,args 是它的参数以元组形式的表示。

接下来 ,我们用 MakePolymorphicAction()将这个实现类对象变为我们所需的多形动作。它封装成下面这种形式会很方便：

```
using ::testing::MakePolymorphicAction;
```

```
using ::testing::PolymorphicAction;

PolymorphicAction<ReturnSecondArgumentAction>
ReturnSecondArgument() {
   return MakePolymorphicAction(ReturnSecondArgumentAction());
}
```

现在，你可以像用内置动作一样的方式来用这个多形的动作：

```
using ::testing::_;

class MockFoo : public Foo {
public:
   MOCK_METHOD2(DoThis, int(bool flag, int n));
   MOCK_METHOD3(DoThat, string(int x, const char* str1, const
char* str2));
};
...

MockFoo foo;
EXPECT_CALL(foo, DoThis(_, _))
   .WillOnce(ReturnSecondArgument());
EXPECT_CALL(foo, DoThat(_, _, _))
   .WillOnce(ReturnSecondArgument());
...
foo.DoThis(true, 5);         // Will return 5.
foo.DoThat(1, "Hi", "Bye");  // Will return "Hi".
```

Teaching Google Mock How to Print Your Values

当一个未设置或是未期望的调用发生时，Google Mock 会打印参数值和栈 trace 帮你 debug。像 EXPECT_THAT 和 EXPECT_EQ 这些断言宏会在断言失败时打印这些值。Google Mock 和 Google Test 会用 Google Test 的用户可扩展值打印器。

这个打印器知道如何打印 C++内置类型，普通数据，STL 容器，和支持<<操作符的任何类型。其它类型，它会以值的原始字符的形式打印，希望可以帮助到你。Google Test 的高级指南中解释了如何扩展打印器来打印你自己的类型。