**Moq**

Mock进行测试的基本过程：

创建Mock Object  =》赋值（Expect）=》重播（Replay）=》诊断（Assert&Verify）

**Moq中几个重要的类:**

Mock<T>：通过这个类我们能够得到一个Mock<T>对象，T可以是接口和类。它有一个公开的Object属性，这个就是我们Moq为我们模拟出的对象。

It：这是一个静态类，用于过滤参数。

MockBehavior：用于配置MockObject的行为，比如是否自动mock。

MockFactory：Mock对象工厂，能够批量生产统一自定义配置的Mock对象，也能批量的进行Mock对象测试。

Match<T>：如果你先嫌It不够用就用Match<T>，通过它能够完全自定义规则。

**初识Moq**：

新建一个测试，我们用三行代码演示一个Moq的使用。

|  |
| --- |
| [TestMethod()] [Owner(wJiang)] public void MoqTest0()         {             //make a Mock Object by Moq             var mo = new Mock<TargetInterfaceOne>();              //Setup our Mock Object             mo.Setup(p => p.MethodWithParamAndResult("abc")).Returns("123");              //Assert it!             Assert.AreEqual("123", mo.Object.MethodWithParamAndResult("abc"));         } |

说明：

new Mock<T>返回一个Mock对象，我们可以用var接收，这样写起来更方便些，Mock<T>有一个Object属性，存储的就是我们的模拟对象实例。

Setup的参数是一个Lambda Expression，我们可以理解为：“当被mock的对象p调用MethodWithParamAndResult方法 并且参数为”abc”的时候”。后面再加一个Return(“123”)我们可以理解为：（在之前Setup的情况下）返回的值为”123”。这样，我们就填充好了一个“伪对象”的行为，我们只让它做一件实事儿：当我们调用mo.Object.MethodWithParaAndResult方法并且参数为”abc”时会返回”123”。

实际上我们不仅能够在Setup后面接Returns方法还能接诸如Throws、Verify之类的方法，这是为什么呢？Setup方法会返回一个ISetup对象，看看ISetup的定义：

public interface ISetup<TMock, TResult> : ICallback<TMock, TResult>, IReturnsThrows<TMock, TResult>, IReturns<TMock, TResult>, IThrows, INever, IVerifies, IHideObjectMembers where TMock : class

恩，是链式编程，ISetup接口继承了很多接口，这里我们注意到IReturns<TMock，TResult>，看看IReturns<TMock, TResult>定义：

public interface IReturns<TMock, TResult> : IHideObjectMembers where TMock : class。

里面有一个方法:IReturnsResult<TMock> Returns<T>(Func<T, TResult> valueFunction);

所以我们还能写出这样的代码：

mo.Setup(p => p.MethodWithParamAndResult("abc"))

.Returns("123")

.Callback(……)

.Throws(……)

.Verifiable(……);

呵呵，这种代码理解起来是很自然的。Moq设计的是不是很人性化呢。

 下面说说Moq中的参数匹配。先看Mock<T>的一个方法。

public ISetup<T> Setup(Expression<Action<T>> expression);

熟悉.NET框架尤其是开发过基于MVVM的WPF应用程序的朋友对Action<T>和Prediect<T>这两个泛型委托应该不陌生，这两个委托的含义很简明，前者表示给定一个参数然后施展一个行为，后者表示施行行为的前提。

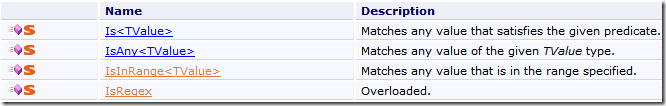
假如我们有一个接口IA，IA有一个方法签名string MethodA1(stringidentity)。那么我们怎么模拟一个实现IA接口的对象MethodA1呢？

|  |
| --- |
| var mo = new Mock<IA>(); mo.Setup( p => p.MethodA1(“50”)).Return(“Hello, mocker”); |

这个例子和上一篇是一样的，这里在额外说明下:Return的参数类型取决于方法的返回类型,如果我们把MethodA1返回void,那么就没法Return方法了,这里由于返回string类型，所以能Return。

当然，方法参数可可选值很多，"51"，”52”，”53”，”54”，”55”，”56”……如果我们一个一个的Setup怎么能行？我向大家隆重推出两个类：It,Match<T>。

先说It，It很适合用来匹配数字，字符串参数，它提供了如下几个静态方法(取自Moq的官方API文档):

[](http://images.cnblogs.com/cnblogs_com/wJiang/WindowsLiveWriter/WhatisMockYouMoq_BFCA/image_2.png)

Is<TValue> 第一个方法的参数Expression<Predict<TValue>>类型，当你需要某种类型并且这种类型要通过代码来判断的话可以使用它。

IsAny<TValue> 第二个方法没有参数，只要是TValue类型的就能匹配成功。

IsInRange<TValue> 第三个方法用来匹配两个的TValue类型值之间的参数。（Range参数可以设定开闭区间）

IsRegex 第四个是用正则表达式匹配。（仅限于字符串类型参数）

举个例子：

|  |
| --- |
| [TestMethod] [Owner(wJiang)] public void MoqTest1() {             var mo = new Mock<TargetInterfaceOne>();             mo.Setup(p => p.MethodWithParamAndResult(It.IsRegex("^God.\*$"))).Returns("bless me");             mo.Setup(p => p.MethodWithParamAndResult(It.Is<string>((param => param.IndexOf("Evil") >= 0)))).Returns("away from me");             //mo.Setup(p => p.MethodWithParamAndResult(It.):             Assert.AreEqual("bless me", mo.Object.MethodWithParamAndResult("God comes"));             Assert.AreEqual("away from me", mo.Object.MethodWithParamAndResult("Evil is here"));         } |
| 这样参数就能自动匹配了，如果参数为”God comes”，MockObject会返回”bless me”结果，如果参数中含有Evil则返回”away from me”结果。 |

但是It提供的功能还是显得有些弱，这时候我们可以自定义匹配验证规则。这就用到了Match<T>。

Match<T>是个静态类，它值公开了一个静态方法（重载了两个版本）：public static T Create(Predict<T> condition, ……)。

先看下下面的代码便于讲解，我们写个用于参数匹配的静态帮助类。

|  |
| --- |
| [TestMethod()]         public void MoqTestA()         {             var mo = new Mock<TargetInterfaceOne>();             mo.Setup(p => p.MethodWithParamAndResult(MatchHelper.CustomMatcher("abc"))).Returns("123");              Assert.AreEqual(mo.Object.MethodWithParamAndResult("abc"), “123);              Assert.IsNull(mo.Object.MethodWithParamAndResult(“newyorktimesbyflex”));         }  public static class MatchHelper         {             public static string CustomMatcher(string arg)             {                 return Match<string>.Create(                      p => p.Equals(arg), ()=>null);             }              public static IEnumerable<string> Contains(string key)             {                 return Match<IEnumerable<string>>.Create(p=>p.Contains(key));             }         } |

对我们而言CustomMatcher(string arg)和Contains(string key)就是验证方法。和上个例子比较，就是将p => p.MethodWithParamAndResult(……)里面的东西换成了我们自己的方法。另外一个Contains方法是可用来满足这样一个需求：给定的参数为可迭代类型，只有包含特定的元素时才能匹配

**Raise**

如果你说会用Setup，那么Raise就更简单了。这里注意下它是无返回值类型。

mockView.Raise(v => v.SelectionChanged += null, new OrderEventArgs { Order = new Order("moq", 500) });

**Callback**

Callback嘛，顾名思义就是回调。使用Callback可以使我们在某个使用特定参数匹配的方法在被调用时得到通知。比如我们要得知在一次测试中某个方法被调用了几次，可以这么做：

|  |
| --- |
| [TestMethod]         public void MoqTest2()         {             var mo = new Mock<TargetInterfaceOne>();             int counter = 0;             mo.Setup(p => p.MethodPure()).Callback( () => counter++ );              mo.Object.MethodPure();             mo.Object.MethodPure();              Assert.AreEqual(2, counter);         } |

在这段代码中我们在Setup方法后接了个Callback方法（或者说是调用了ISetup的Callabck方法实现）。这段代码的意思就是在调用MethodPure方法时会执行Callback中的Action委托。

调用两次MethodPure()，测试结果证明确实累加了两次counter。

**Verify**

有些时候我们并不关注方法的返回结果，而是关注某个方法是不是在内部被调用。

这时我们就用到了Verify/VerifyAll。同时有个有用的类型Times，规定应该调用多少次。如果验证失败则抛出异常。

|  |
| --- |
| [TestMethod()]  public void MoqTest3() {     var mo = new Mock<TargetInterfaceOne>();     mo.Setup( p => p.MethodPure() );     mo.Setup( p => p.MethodWithParam("123")).Verifiable("it should be invoked");     //mo.Object.MethodPure();     mo.Object.MethodWithParam("123");     mo.Verify( p => p.MethodPure(), Times.AtLeastOnce() );     mo.Verify(p => p.MethodWithParam("thto"), Times.AtLeastOnce(),          "this method  invoking of MethodWithParam() with the parameter: \"thto\" is not happened");      mo.Object.MethodPure(); } |

如果在MethodPure前调用mo.Verify(p => p.MethodPure())则会抛出异常，因为不符合条件：在执行verify前至少调用一次。

**关于Verify和VerifyAll**

这两个方法会对Mock对象的所有Setup过的方法进行验证，那么有什么不同呢？注意到上面代码中绿色字体部分，有一个Verifiable方法，可以理解为为这个Setup的东西加了个验证标记。而Setup(p=>p.MethodPure())时就没有些，那么我们在使用调用Verify()时只会对MethodWithParam(“123”)进行验证而不会对MethodPure()是否被调用过进行验证。

何谓Mock对象行为？

由于模拟出的对象终究是用来“糊弄人”的。我们在UnitTest中不一定会将一个对象的所有方法都Mock掉。而且如果一个Mock对象中有还有用接口/抽象类表示的对象，那么我们不一定就要将它们一起Mock掉。Moq为我们提供了自定义配置这些细节规则的办法。

**MockBehavior**

Moq有个枚举类型MockBehavior，有三个值Strict，Loose，Default。

Strict表示Mock对象在调用一个方法前这个方法必须被Mock掉，否则就会引发MockException。

而Loose与之相反，如果调用没有Mock的方法也不会出错。Default默认为Loose。

具体的设置方法是在new一个Mock<T>的时候。要注意Mock<T>中的Behavior属性是只读的。

|  |
| --- |
| [TestMethod]         [ExpectedException(typeof(MockException))]         public void MoqTest4()         {             var mo = new Mock<TargetInterfaceOne>(                 MockBehavior.Strict/\*如果设置为Loose则不会引发异常，当前默认为Loose\*/                 );//还有要注意的是Mock.Behavior是只读属性，所以只能在构造方法中设置             mo.Object.MethodPure();//在MockBehavior.Strict设置下，一切调用未填充的方法/属性/事件时会抛出异常         } |

**DefaultValue**

我们再添加一个接口TargetInterfaceTwo用来演示DefaultValue在不同设定下Mock对象的不同行为。代码如下：

|  |
| --- |
| [TestMethod]         [ExpectedException(typeof(NullReferenceException))]         public void MoqTest5()         {             var mo = new Mock<TargetInterfaceTwo>             {                 DefaultValue = DefaultValue.Mock/\*如果设置为DefaultValue.Mock就不会引发异常，没有引用的成员会被自动Mock\*/             };              mo.Object.one.MethodPure();             /\*var inner\_mo = Mock.Get(mo.Object.one);Mock.Get可以用于获得其中自动Mock的对象实例，然后再对其进一步操作\*/         }  public interface TargetInterfaceTwo         {             TargetInterfaceOne one { get; set; }             void Two();         } |

这里注意Mock类（一个抽象类）有一个静态方法Get<T>(T mock)，如果Mock对象是被自动创建的，我们可以用它来获得这个Mock对象。

**MockFactory**

通过MockFactory我们可以批量生产我们自定义配置的Mock对象，并通过MockFactory.Verify/VerifyAlll来统一验证。示例如下。

|  |
| --- |
| [TestMethod]         public void MoqTest6()         {             MockFactory mf = new MockFactory(MockBehavior.Loose) { DefaultValue = DefaultValue.Mock };             var mo = mf.Create<TargetInterfaceOne>();             var mo2 = mf.Create<TargetInterfaceOne>();             mo.Setup(p => p.MethodPure()).Verifiable("must be invoked");             mo2.Setup(p => p.MethodPure());             mf.Verify();         } |

[**Moq & RhinoMocks**](http://www.cnblogs.com/huyh/archive/2010/06/14/1758143.html)

使用Mock对象进行测试一般都会有以下三个关键步骤:

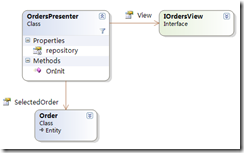
1. 使用接口来描述需要测试的对象
2. 为实际的产品代码实现这个接口
3. 以测试为目的,在Mock对象中实现这个接口

在使用Mock对象的过程中，充分体现出了“面向接口编程”的设计原则，同时也促成类的良好设计。

自行实现Mock对象是相当繁琐的工作，让人幸运的是，在.NET世界中有多个优秀的Mock框架可以供大家选择，目前最常使用的无非[Moq](http://code.google.com/p/moq/" \t "_blank)与[Rhino Mocks](http://ayende.com/Wiki/Rhino+Mocks+3.5.ashx" \t "_blank)这两个框架。两者的最新版本在Mocking API方面的用法已日趋一致，都依托Lambda表达式、泛型和扩展方法做了很大改进，目标都是让Mock对象以一种更自然的方式与多个单元测试框架进行集成，以一种清晰的语法来描述期望值、参数约束、返回值等，极大的方便开发者的使用。

由于Moq和Rhino Mocks都使用了Castle DynamicProxy这个类库动态生成代理类，因此对需要Mock的对象有一定的限制：所测试的方法必须是virtual类型。

下面就用一个例子来看看两者的不同实现(这个例子摘自Moq源代码包中的Samples，只是略做了些修改以便于展现两者的特点)：

[](http://images.cnblogs.com/cnblogs_com/huyh/WindowsLiveWriter/MockFramework_1412C/mocks_2.png)

需要进行测试的对象如下示之：

|  |
| --- |
| public class OrdersPresenter  {  private IRepository<Order> repository { get; set; }  public IOrdersView View { get; set; }  public OrdersPresenter(IOrdersView view, IRepository<Order> repository) {  this.View = view;  this.repository = repository;  this.View.OrderSelected += (sender, args) => DoOrderSelection(args.Order);  }  public Order SelectedOrder { get; private set; }  private void DoOrderSelection(Order selectedOrder) {  // Do something when the view selects an order.  SelectedOrder = selectedOrder;  }  public virtual void OnInit() {  View.Orders = repository.FindAll();  } |

[折叠](javascript:void(0))

public class OrdersPresenter

{

private IRepository<Order> repository { get; set; }

public IOrdersView View { get; set; }

public OrdersPresenter(IOrdersView view, IRepository<Order> repository) {

this.View = view;

this.repository = repository;

this.View.OrderSelected += (sender, args) => DoOrderSelection(args.Order);

}

public Order SelectedOrder { get; private set; }

private void DoOrderSelection(Order selectedOrder) {

// Do something when the view selects an order.

SelectedOrder = selectedOrder;

}

public virtual void OnInit() {

View.Orders = repository.FindAll();

}

}

下面是用这两个Mock框架分别实现的单元测试代码：

|  |  |
| --- | --- |
| Moq 4.0 | Rhino Mocks 3.6 |
| [Test]  public void TestPresenterSelection() {  // arrange  var mView = new Mock<IOrdersView>();  var mRepository = new Mock<IRepository<Order>>();  var presenter = new OrdersPresenter(mView.Object, mRepository.Object);  // check that the presenter has no selection by default  Assert.Null(presenter.SelectedOrder);  // raise event  mView.Raise(io => io.OrderSelected += null,  new OrderEventArgs { Order = new Order("moq", 50) });  // assert  Assert.NotNull(presenter.SelectedOrder);  Assert.AreEqual("moq", presenter.SelectedOrder.ProductName);  } | [Test]  public void TestPresenterSelection() {  // arrange  var mView = MockRepository.GenerateMock<IOrdersView>();  var mRepository = MockRepository.GenerateMock<IRepository<Order>>();  var presenter = new OrdersPresenter(mView, mRepository);  // check that the presenter has no selection by default  Assert.Null(presenter.SelectedOrder);  // raise event  mView.Raise(io => io.OrderSelected += null, null,  new OrderEventArgs { Order = new Order("moq", 50) });  // assert  Assert.NotNull(presenter.SelectedOrder);  Assert.AreEqual("moq", presenter.SelectedOrder.ProductName);  } |
| [Test]  public void TestRetrieveOrders() {  // arrange  var mView = new Mock<IOrdersView>();  var mRepository = new Mock<IRepository<Order>>();  var presenter = new OrdersPresenter(mView.Object, mRepository.Object);  List<Order> defaultOrders = new List<Order>  { new Order("moq"), new Order("RhinoMock") };  mRepository.Setup(r => r.FindAll()).Returns(defaultOrders);  // exercise mocks  presenter.OnInit();  // assert  mView.VerifySet(v => v.Orders = defaultOrders);  } | [Test]  public void TestRetrieveOrders() {  // arrange  var mView = MockRepository.GenerateMock<IOrdersView>();  var mRepository = MockRepository.GenerateStub<IRepository<Order>>();  var presenter = new OrdersPresenter(mView, mRepository);  List<Order> defaultOrders = new List<Order>  { new Order("moq"), new Order("RhinoMock") };  mRepository.Stub(ir => ir.FindAll()).Return(defaultOrders);  // exercise mocks  presenter.OnInit();  // assert  mView.AssertWasCalled(v => v.Orders = defaultOrders);  } |

**Conclusion**

通过上面的实例我们可以很容易看出两者的Syntax与API都非常接近，使用两者任何一个都能方便实现你的测试目的。