Unit Test – moq

<https://docs.microsoft.com/en-us/ef/ef6/fundamentals/testing/mocking>

<https://docs.microsoft.com/en-us/ef/ef6/fundamentals/testing/mocking>

<https://www.cnblogs.com/haogj/archive/2011/07/22/2113496.html>

<https://www.cnblogs.com/huanxiaolu/p/6892549.html>

<https://www.cnblogs.com/zjoch/p/6565956.html>

public classMock**<**T**> :** Mock**,** IMock**<**T**>** whereT **:** class

**{**

**public** Mock**();**

**public** Mock**(params object[] args);**

**public** Mock**(MockBehavior behavior);**

**public** Mock**(MockBehavior behavior, params object[] args);**

**public virtual T** Object **{ get; }**

**public override bool** CallBase **{ get; set; }**

**public override MockBehavior** Behavior **{ get; }**

**public string** Name **{ get; set; }**

**public override** Switches **{ get; set; }**

**public void** Raise**(Action<T> eventExpression, EventArgs args);**

**public void** Raise**(Action<T> eventExpression, params object[] args);**

**public ISetup<T, TResult>** Setup**<TResult>(Expression<Func<T, TResult>> expression);**

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

**public Mock<T>** SetupAllProperties**();**

**public ISetupGetter<T, TProperty>** SetupGet**<TProperty>(Expression<Func<T, TProperty>> expression);**

**public Mock<T>** SetupProperty**<TProperty>(Expression<Func<T, TProperty>> property);**

**public Mock<T>** SetupProperty**<TProperty>(Expression<Func<T, TProperty>> property, TProperty initialValue);**

**public ISetupSequentialResult<TResult>** SetupSequence**<TResult>(Expression<Func<T, TResult>> expression);**

**public ISetupSequentialAction** SetupSequence**(Expression<Action<T>> expression);**

**public ISetup<T>** SetupSet**(Action<T> setterExpression);**

**public ISetupSetter<T, TProperty>** SetupSet**<TProperty>(Action<T> setterExpression);**

**public override string** ToString**();**

**public void** Verify**<TResult>(Expression<Func<T, TResult>> expression, string failMessage);**

**public void** Verify**<TResult>(Expression<Func<T, TResult>> expression, Func<Times> times);**

**public void** Verify**(Expression<Action<T>> expression);**

**public void** Verify**<TResult>(Expression<Func<T, TResult>> expression, Times, string failMessage);**

**public void** Verify**(Expression<Action<T>> expression, Times times);**

**public void** Verify**(Expression<Action<T>> expression, Func<Times> times);**

**public void** Verify**(Expression<Action<T>> expression, string failMessage);**

**public void** Verify**(Expression<Action<T>> expression, Times, string failMessage);**

**public void** Verify**(Expression<Action<T>> expression, Func<Times> times, string failMessage);**

**public void** Verify**<TResult>(Expression<Func<T, TResult>> expression);**

**public void** Verify**<TResult>(Expression<Func<T, TResult>> expression, Times times);**

**public void** VerifyGet**<TProperty>(Expression<Func<T, TProperty>> expression);**

**public void** VerifyGet**<TProperty>(Expression<Func<T, TProperty>> expression, Times times);**

**public void** VerifyGet**<TProperty>(Expression<Func<T, TProperty>> expression, Func<Times> times, string failMessage);**

**public void** VerifyGet**<TProperty>(Expression<Func<T, TProperty>> expression, Times, string failMessage);**

**public void** VerifyGet**<TProperty>(Expression<Func<T, TProperty>> expression, Func<Times> times);**

**public void** VerifyGet**<TProperty>(Expression<Func<T, TProperty>> expression, string failMessage);**

**public void** VerifyNoOtherCalls**();**

**public void** VerifySet**(Action<T> setterExpression, Func<Times> times, string failMessage);**

**public void** VerifySet**(Action<T> setterExpression, Times, string failMessage);**

**public void** VerifySet**(Action<T> setterExpression, string failMessage);**

**public void** VerifySet**(Action<T> setterExpression, Func<Times> times);**

**public void** VerifySet**(Action<T> setterExpression, Times times);**

**public void** VerifySet**(Action<T> setterExpression);**

**public ISetupConditionResult<T>** When**(Func<bool> condition);**

**protected override object** OnGetObject**();**

**}**

**我们先创建一个案例使用的**DbContext

[Table("Teachers")]

public class Teacher

{

public Teacher()

{

this.Students = new Collection<Student>();

}

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string TeacherName { get; set; }

public int Years { get; set; }

public ICollection<Student> Students { get; set; }

}

[Table("Students")]

public class Student

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTime BirthDate { get; set; }

[NotMapped]

public string FullName {

get

{

return $"{FirstName} {LastName}";

}

}

}

public class TeacherDB : DbContext

{

public TeacherDB() { }

public TeacherDB(DbContextOptions<TeacherDB> options) : base(options) {}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) { }

protected override void OnModelCreating(ModelBuilder modelBuilder) { }

public DbSet<Teacher> Teachers { get; set; }

public DbSet<Student> Students { get; set; }

}

**我们要测试一个方法：**

public static class TOperation

{

public static int getCount(TeacherDB db)

{

return db.Teachers.Count();

}

}

[Fact]

public void Test1()

{

var dbMock = new Mock<TeacherDB>();

var tMock = dbMock.Object.Set<Teacher>();

dbMock.Setup(x => x.Set<Teacher>()).Returns(tMock);

int count = TOperation.getCount(dbMock.Object);

Assert.Equal(count, 2);

}

注：

1. new Mock<T>() - 这个是 Mock 对象
2. Mock.Object - 返回 T 才是我们需要的模拟对象
3. dbMock.Object.Set<Teacher>() – 错误的方法来 生成 DbSet<Teacher>.

var tMock = dbMock.Object.Set<Teacher>();

tMock.Add(new Teacher { Id = 20, TeacherName = "T222", Years = 22 });

1. Mock.Object – 是我们的模拟对象，它拥有真实对象所定义的各种属性与方法。

dbMock.Object.Set<Teacher>(); 实际上对应的是 DbContext.Set<T> - 返回的是 DbSet<T>

我们要搞清楚 DbContext.Set<Teacher> 和 public DbSet<Teacher> Teachers { get; set; } 的区别：

从对象的角度来说， 他们并不是同一个对象。通过以下案例可以看到：

public void Test1()

{

var dbMock = new Mock<TeacherDB>();

var tMock = dbMock.Object.Set<Teacher>();

tMock.Add(new Teacher { Id = 10, TeacherName = "T111", Years = 11 });

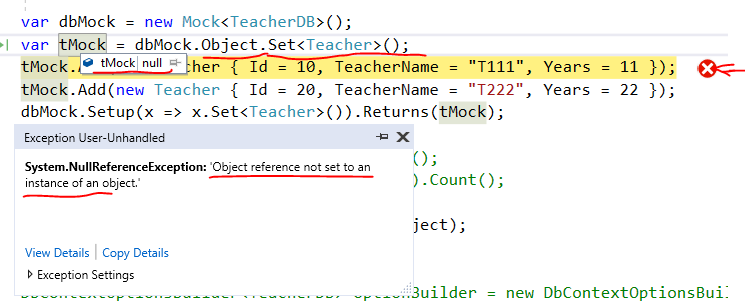
tMock.Add(new Teacher { Id = 20, TeacherName = "T222", Years = 22 });

dbMock.Setup(x => x.Set<Teacher>()).Returns(tMock);

int count = TOperation.getCount(dbMock.Object);

Assert.Equal(count, 2);

}



理解错误一： 在于对Mock 对象的理解错误：

var tMock = dbMock.Object.Set<Teacher>();

tMock.Add(new Teacher { Id = 10, TeacherName = "T111", Years = 11 }); - 空值是不会有.Add() 方法的

- dbMock.Object 本来是模拟对象，.Set<Teacher> 并不会返回 DbSet<Teacher> 对象.

- 所以不能依赖于 dbMock.Object 里面的方法来生成其他依赖对象。

理解错误二： 在于对Mock 对象的理解错误：

dbMock.Setup(x => x.Set<Teacher>()).Returns(tMock); - DbContext.Set<T>() 来构造 db.Teachers

这是张冠李戴，不是同一个对象.

public void Test1()

{

DbContextOptionsBuilder<TeacherDB> optionBuilder = new DbContextOptionsBuilder<TeacherDB>();

//optionBuilder.UseInMemoryDatabase("MyTestDB");

optionBuilder.UseSqlServer("Data Source=(localdb)\\lwh;Initial Catalog=ClassRoom;Integrated Security=true;User=WILLIAM\\Administrator;password=Liu011225;Connect Timeout=30;");

TeacherDB tdb = new TeacherDB(optionBuilder.Options);

int tcnt = tdb.Teachers.ToList().Count();

var dbMock = new Mock<TeacherDB>();

dbMock.Setup(x => x.Set<Teacher>()).Returns(tdb.Teachers);-这里使用了真实的DbSet<Teacher>对象来模拟

int count = TOperation.getCount(dbMock.Object);

Assert.Equal(count, 2); - 数据库里有两个记录， 刚好相等，测试成功

}

public static class TOperation

{

public static int getCount(TeacherDB db)

{

return db.Set<Teacher>().ToList().Count(); - 这里DbSet<Teacher>()不是空值，返回是2个数据

}

}

可以成功测试!

-----------------------------------------------------------------------------------------------------

如果换成 db.Teachers.ToList()， 则测试代码是出错的。

public static class TOperation

{

public static int getCount(TeacherDB db)

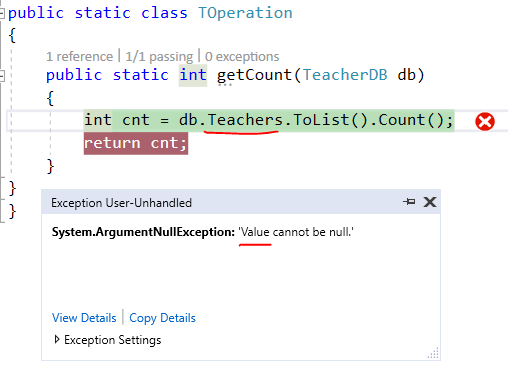
{

int cnt = db.Teachers.ToList().Count();

return cnt;

}

}



--------------------------------------------------------

如果都是 db.Teachers 则测试成功

dbMock.Setup(x => x.Teachers).Returns(tdb.Teachers);

int cnt = db.Teachers.ToList().Count();

-------------------------------------------------------------------------------------------------------------

如何Mock 可以查询的数据： 主要是 IQueryable<T> 才支持数据库查询

var tMock = new Mock<DbSet<Teacher>>();

var teachers = new List<Teacher>

{

new Teacher{ Id = 10, TeacherName="T1", Years=11 },

new Teacher{Id =20, TeacherName="T2", Years=22}

}.AsQueryable();

tMock.As<IQueryable<Teacher>>().Setup(x => x.Provider).Returns(teachers.Provider);

tMock.As<IQueryable<Teacher>>().Setup(x => x.Expression).Returns(teachers.Expression);

tMock.As<IQueryable<Teacher>>().Setup(x => x.ElementType).Returns(teachers.ElementType);

tMock.As<IQueryable<Teacher>>().Setup(x => x.GetEnumerator()).Returns(teachers.GetEnumerator());

dbMock.Setup(x => x.Teachers).Returns(tMock.Object);

int count = TOperation.getCount(dbMock.Object);

Assert.Equal(count, 2);

public static class TOperation

{

public static int getCount(TeacherDB db)

{

int cnt = db.Teachers.ToList().Count();

return cnt;

}

}

如果这个方法设置起来太复杂, 我们应该使用原始的 DbContext 对象，只是不直接连接数据库，使用内存数据库

[Fact]

public void Test1()

{

DbContextOptionsBuilder<TeacherDB> optionBuilder = new DbContextOptionsBuilder<TeacherDB>();

optionBuilder.UseInMemoryDatabase("MockDB");

TeacherDB mdb = new TeacherDB(optionBuilder.Options);

mdb.Teachers.AddRange(

new Teacher { Id = 10, TeacherName="T1", Years = 11},

new Teacher { Id = 20, TeacherName="T2", Years = 20},

new Teacher { Id= 30, TeacherName="T3", Years = 33 }

);

mdb.SaveChanges(); - 这个很重要， 一定要 Save

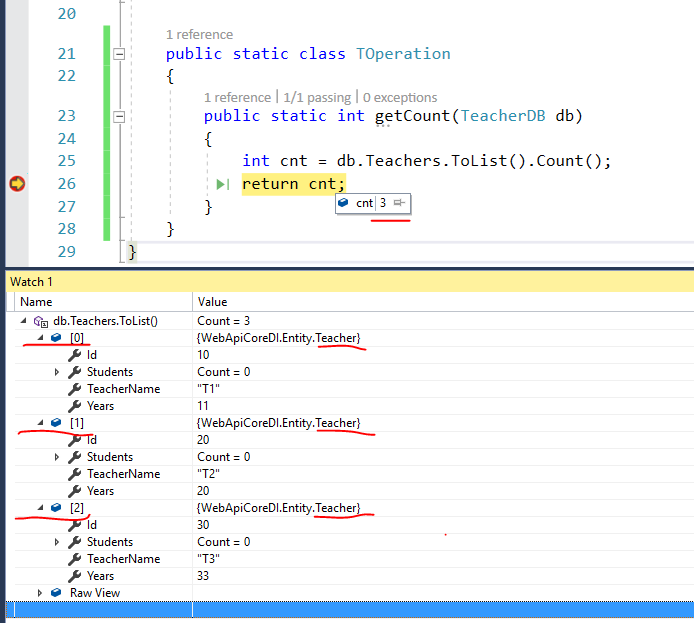
var dbMock = new Mock<TeacherDB>();

dbMock.Setup(x => x.Teachers).Returns(mdb.Teachers); - 使用内存模拟的对象

int count = TOperation.getCount(dbMock.Object); - 返回 3 个数据

Assert.Equal(count, 2);

}



--------------------------------------------------------------------------------------------------------------------------

测试方法： 代码应该安全可靠，不抛出系统错误

TeacherDB mdb = new TeacherDB(optionBuilder.Options);

/\*

mdb.Teachers.AddRange(

new Teacher { Id = 10, TeacherName="T1", Years = 11},

new Teacher { Id = 20, TeacherName="T2", Years = 20},

new Teacher { Id= 30, TeacherName="T3", Years = 33 }

);

mdb.SaveChanges();

\*/

var dbMock = new Mock<TeacherDB>();

dbMock.Setup(x => x.Teachers).Returns(mdb.Teachers);

int count = TOperation.getCount(dbMock.Object); - 返回 0, 而不会因为对象为空，抛出错误

Assert.Equal(count, 2);

即使不添加数据，测试也不会因为对象不存在而抛出系统错误，测试仍然可以继续进行。

public static class TOperation

{

public static int getCount(TeacherDB db)

{

int cnt = db.Teachers.ToList().Count();

return cnt;

}

}

-------------------------------------------------------------------------------------------------------------

如何Mock 可以异步查询的数据: 模拟异步的数据比较复杂，还是使用最近简单的内存数据库

public static async Task<IList<Teacher>> getTeachers(TeacherDB db)

{

IList<Teacher> teachers = new List<Teacher>();

try

{

teachers = await db.Teachers.ToListAsync();

}

catch (Exception err)

{

var err1 = err;

}

return teachers;

}

public async void Test1()

{

var dbMock = new Mock<TeacherDB>();

var tMock = new Mock<DbSet<Teacher>>();

var teachers = new List<Teacher>

{

new Teacher{ Id = 10, TeacherName="T1", Years=11 },

new Teacher{Id =20, TeacherName="T2", Years=22}

}.AsQueryable();

tMock.As<IQueryable<Teacher>>().Setup(x => x.Provider).Returns(teachers.Provider);

tMock.As<IQueryable<Teacher>>().Setup(x => x.Expression).Returns(teachers.Expression);

tMock.As<IQueryable<Teacher>>().Setup(x => x.ElementType).Returns(teachers.ElementType);

tMock.As<IQueryable<Teacher>>().Setup(x => x.GetEnumerator()).Returns(teachers.GetEnumerator());

dbMock.Setup(x => x.Teachers).Returns(tMock.Object);

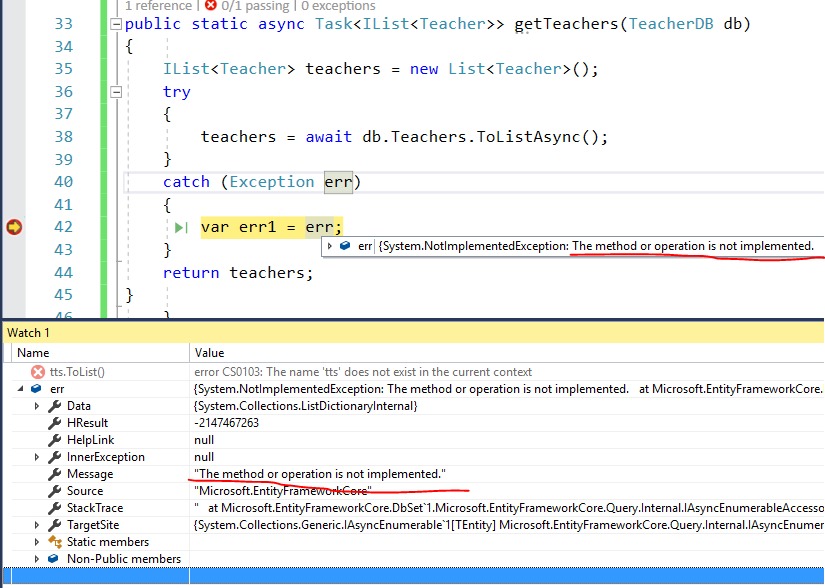
var tts = await TOperation.getTeachers(dbMock.Object);

int count = tts.Count();

Assert.Equal(count, 2);

}

执行异步方法：



-----------------------------------------------------------------------------------------------------------------------

使用最简单的方法： 使用内存数据库来生成模拟数据， 并且模拟数据类型完全一致。

[Fact]

public async void Test1()

{

DbContextOptionsBuilder<TeacherDB> optionBuilder = new DbContextOptionsBuilder<TeacherDB>();

optionBuilder.UseInMemoryDatabase("MockDB");

TeacherDB mdb = new TeacherDB(optionBuilder.Options);

mdb.Teachers.AddRange(

new Teacher { Id = 10, TeacherName="T1", Years = 11},

new Teacher { Id = 20, TeacherName="T2", Years = 20},

new Teacher { Id= 30, TeacherName="T3", Years = 33 }

);

mdb.SaveChanges();

var dbMock = new Mock<TeacherDB>();

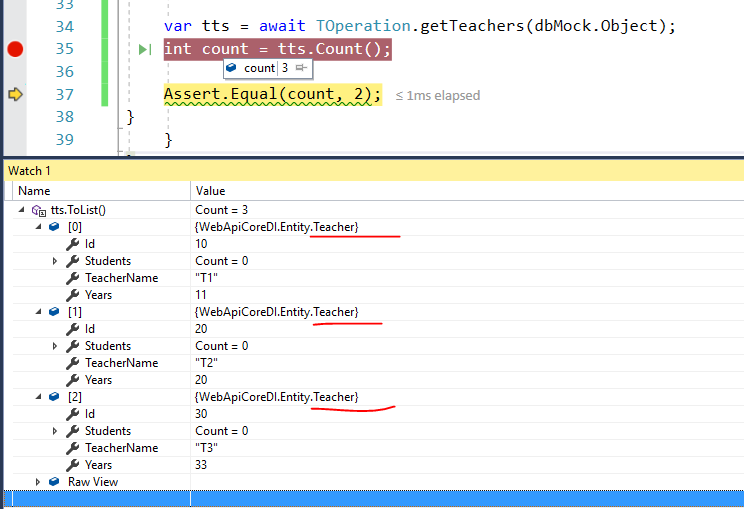
dbMock.Setup(x => x.Teachers).Returns(mdb.Teachers);

var tts = await TOperation.getTeachers(dbMock.Object);

int count = tts.Count();

Assert.Equal(2, count);

}



-----------------------------------------------------------------------------------------------------------------------

### Virtual DbSet properties with EF Designer

### Note that the DbSet properties on the context are marked as virtual. This will allow the mocking framework to derive from our context and overriding these properties with a mocked implementation.

public class TeacherDB : DbContext

{

public TeacherDB() {

}

public TeacherDB(DbContextOptions<TeacherDB> options) : base(options) {}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) { }

protected override void OnModelCreating(ModelBuilder modelBuilder) { }

public virtual DbSet<Teacher> Teachers { get; set; }

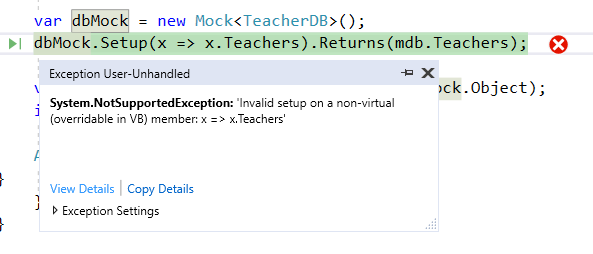
public virtual DbSet<Student> Students { get; set; }

}

virtual DbSet<T> - 必须是virtual, 否则不能被模拟数据 override

下面的错误就是没有定义virtual

public DbSet<Teacher> Teachers { get; set; }



对方法的模拟：

public class TeacherDB : DbContext

{

public TeacherDB() {

}

public TeacherDB(DbContextOptions<TeacherDB> options) : base(options) {}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) { }

protected override void OnModelCreating(ModelBuilder modelBuilder) { }

public virtual DbSet<Teacher> Teachers { get; set; }

public virtual DbSet<Student> Students { get; set; }

public virtual bool getFlag() - 要想方法，可以被模拟的方法所覆盖， 必须是virtual，否则出错

{

return Teachers.Count() > 0 ? true : false;

}

}

public async void Test1()

{

DbContextOptionsBuilder<TeacherDB> optionBuilder = new DbContextOptionsBuilder<TeacherDB>();

optionBuilder.UseInMemoryDatabase("MockDB");

TeacherDB mdb = new TeacherDB(optionBuilder.Options);

/\*

mdb.Teachers.AddRange(

new Teacher { Id = 10, TeacherName="T1", Years = 11},

new Teacher { Id = 20, TeacherName="T2", Years = 20},

new Teacher { Id= 30, TeacherName="T3", Years = 33 }

);

mdb.SaveChanges();

\*/

var dbMock = new Mock<TeacherDB>();

dbMock.Setup(x => x.Teachers).Returns(mdb.Teachers);

dbMock.Setup(x => x.getFlag()).Returns(true);

bool ff = TOperation.getFlag(dbMock.Object);

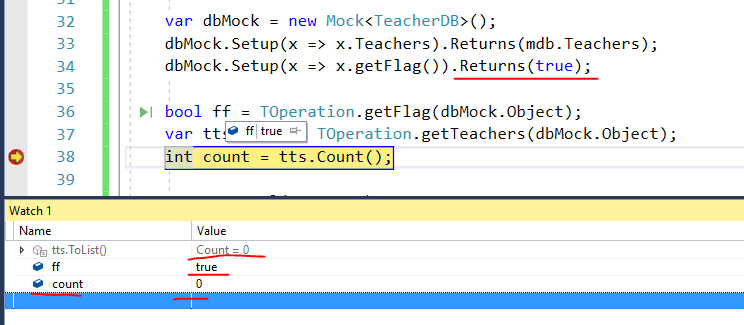
var tts = await TOperation.getTeachers(dbMock.Object);

int count = tts.Count();

Assert.Equal(2, count);

}

虽然没有数据， 但是放回的值仍然是 true



绕了一大圈如果是要模拟DbContext ,为何还要使用Mock , 直接使用内存数据库来得更直接

public async void Test1()

{

DbContextOptionsBuilder<TeacherDB> optionBuilder = new DbContextOptionsBuilder<TeacherDB>();

optionBuilder.UseInMemoryDatabase("MockDB");

TeacherDB mdb = new TeacherDB(optionBuilder.Options);

mdb.Teachers.AddRange(

new Teacher { Id = 10, TeacherName="T1", Years = 11},

new Teacher { Id = 20, TeacherName="T2", Years = 20},

new Teacher { Id= 30, TeacherName="T3", Years = 33 }

);

mdb.SaveChanges();

bool ff = TOperation.getFlag(mdb);

var tts = await TOperation.getTeachers(mdb);

int count = tts.Count();

Assert.Equal(2, count);

}

根据自己的需要，直接添加模拟数据即可。

**Mock 的各种属性**

对 mock 的认识： 当我们mock 一个类型，

var dbMock = new Mock<TeacherDB>(); - 相当于 dbMock.Object = new TeacherDB();

**Setup()** – 测试方法

* 可以用来赋值：

dbMock.Setup(x => x.Teachers).Returns(mdb.Teachers);

每当我们 set/get x.Teachers, 则返回 mdb.Teachers

* Dsfads

**SetupGet()** – Setup() – 去获取Get 数据时，它就会调用 SetupGet() 方法

* dbMock.SetupGet(x => x.Teachers).Returns(mdb.Teachers);

dbMock.SetupGet(x => x.tableName).Returns("GoodName");

bool ff = TOperation.getFlag(dbMock.Object, 2);

public static bool getFlag(TeacherDB db, int a)

{

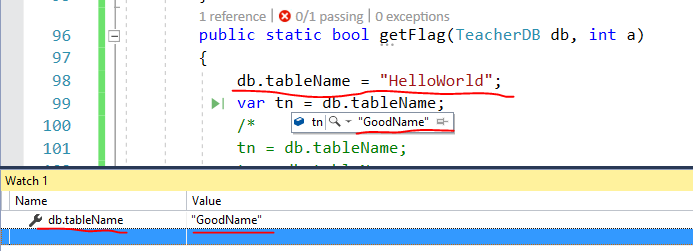
db.tableName = "HelloWorld"; - 虽然我们重新赋值

var tn = db.tableName; - 但是当我们取值时，还是强制使用我们预设的值。有点只读属性的概念

bool f = db.getFlag(a);

return f;

}



string nn = "";

dbMock.SetupGet(x => x.tableName).Returns("GoodName").Callback(() => nn="No Arguement");

var nn1 = nn; -- 此时并没有触发callback, 所以 nn1=""

bool ff = TOperation.getFlag(dbMock.Object, 2);

var nn2 = nn; -- 此时并触发callback, 所用 nn2="No Arguement"

public static bool getFlag(TeacherDB db, int a)

{

db.tableName = "HelloWorld"; - 虽然此处把值更改成 "HelloWorld"

var tn = db.tableName; - 由于属性是只读的，所以获取值仍然是"GoodName"

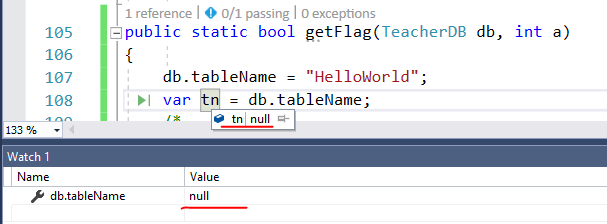
bool f = db.getFlag(a);

return f;

}

---------------------------------------------------------------------------

dbMock.SetupGet(x => x.tableName); - 此处没有Returns, 那么只读属性的值是 null



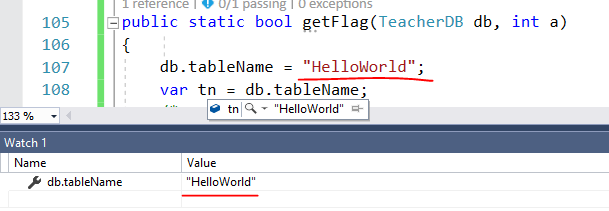
-------------------------------------------------------------------------------

只读属性也可以被取消，变回可读写属性：

dbMock.SetupGet(x => x.tableName).Returns("GoodName"); - 此处设置只读属性

dbMock.SetupProperty(x => x.tableName); - 此处取消只读属性

bool ff = TOperation.getFlag(dbMock.Object, 2);



**SetupSet()** – Setup() – 去设置Set 数据时，它就会调用 SetupSet() 方法, 这是用来设置期望值的

记住：这并不是给属性设置值用的，而是设置完值以后，用来判断是否与期望值相等，相等则满足条件

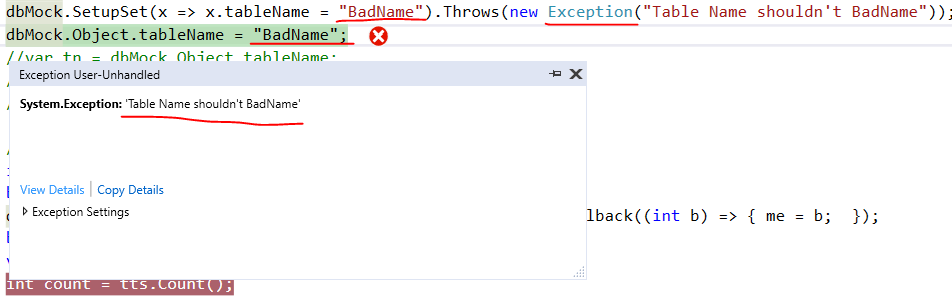
 // 期望设置属性的值为 "foo"   
 mock.SetupSet(foo => foo.Name = "foo");  - 这里相当于 if (foo.Name == “foo”) then execute chained method.

看看案例： 就能正确理解此方法：

dbMock.SetupProperty(x => x.tableName, "BadName"); - 使用 SetupProperty() 并不会触发SetupSet()

dbMock.SetupSet(x => x.tableName = "BadName").Throws(new Exception("Table Name shouldn't BadName"));

dbMock.Object.tableName = "BadName"; - 如果此处赋值触发SetupSet()方法，刚好满足条件

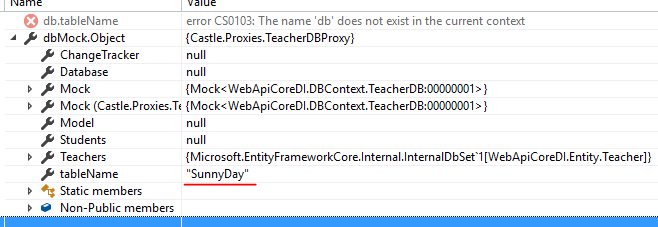


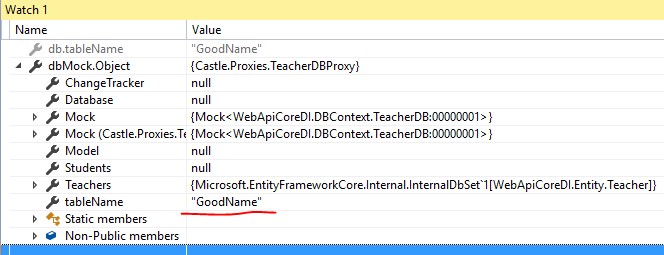
dbMock.Object.tableName = "YesName"; - 只要不等于"BadName" 则顺利通过，不会抛出异常

**SetupProperty()** – 设置值，并不会触发SetupSet() 方法，直接修改属性的值：

dbMock.SetupProperty(x => x.tableName, "SunnyDay"); - 设置属性的值。

dbMock.SetupGet(x => x.tableName).Returns("GoodName"); - 但是遇到它，还是被强制改为只读属性值"GoodName");





dbMock.SetupProperty(); 只要执行这句，如果是只读属性，开始 "tracking" 属性的 sets/gets，

也就是说只读属性将被取消，恢复为可读写属性

dbMock.SetupGet(x => x.tableName).Returns("GoodName");

dbMock.Object.tableName = "BadName";

var tn1 = dbMock.Object.tableName; - 此时是只读属性： 所以 tn1="GoodName", 不是"BadName"

dbMock.SetupProperty(x => x.tableName, "YeildTable");

var tn2 = dbMock.Object.tableName; - 此时恢复读写，所以 tn2="YeildTable"

bool ff = TOperation.getFlag(dbMock.Object, 2);

var tn3 = dbMock.Object.tableName; - tn3="HelloWorld"

public static bool getFlag(TeacherDB db, int a)

{

db.tableName = "HelloWorld";

var tn = db.tableName; - 获取值是"HelloWorld"

bool f = db.getFlag(a);

return f;

}

**Callback()** - Action<T,T, ..> 取决于有多少个参数

dbMock.SetupGet(x => x.tableName).Returns("GoodName").Callback(() => nn="No Arguement");

tableName – 是属性，相当于没有参数的方法， 所以callback是 Action = ()=> {}

dbMock.Setup(x => x.getFlag(2)).Returns( (int a)=> a<=3 ).Callback((int b) => { me = b; });

getFlag(2) – 有一个整型参数，所有 Action<int> = (int a) => { var b=a; }

**Returns()** – Func<T,T,.., result\_Type>

dbMock.Setup(x => x.getFlag(2)).Returns( (int a)=> a<=3 )

Func<int, bool>

**Throws<TException>() –** 满足条件时，抛出异常

**Throws() –** 满足条件时，抛出异常

dbMock.SetupSet(x => x.tableName = "BadName").Throws(new Exception("Table Name shouldn't BadName"));

soMock.Setup(x => x.sum(2, 2)).Throws<DivideByZeroException>();

**.Raises(event, eventArgs) –** 抛出一个事件

public class TeacherDB : DbContext {

public virtual event EventHandler<MyArgs> MyEvent;

}

private void eventHandler(Object sender, MyArgs e)

{

var meobj = sender;

var eo = e;

var i = 100;

}

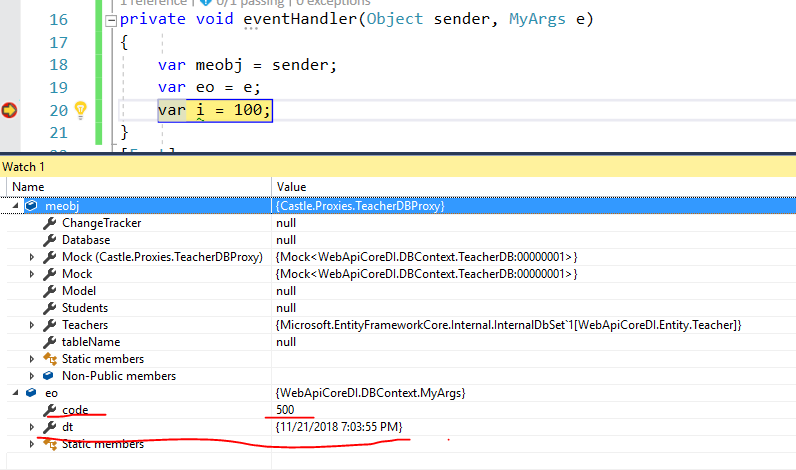
dbMock.Object.MyEvent += eventHandler; - 模拟的Event handler

dbMock.Setup(x => x.getFlag(2)).Returns( (int a)=> a<=3 )

.Raises(m => m.MyEvent+=null, new MyArgs { code = 500, dt = DateTime.Now });

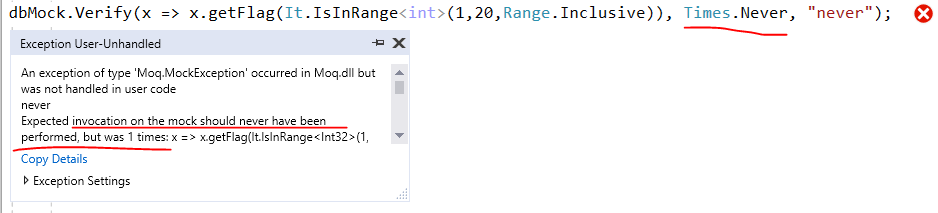
因为 Event 已经有了，只要 +=null 即可

bool ff = TOperation.getFlag(dbMock.Object, 2);

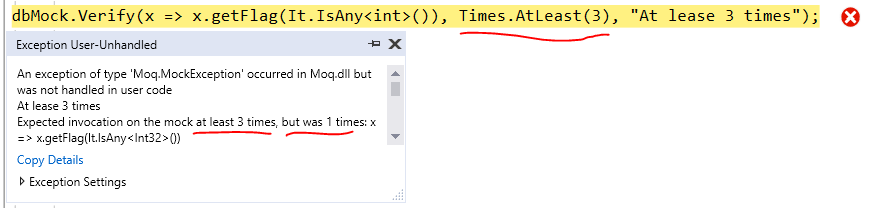


**.Verify() –**  主要是对满足条件的method 的执行次数进行验证,所有Verify一定是要在执行相关操作结束以后来对结果验证

dbMock.Verify(x => x.getFlag(It.IsInRange<int>(1,20,Range.Inclusive)), Times.Never, "never");

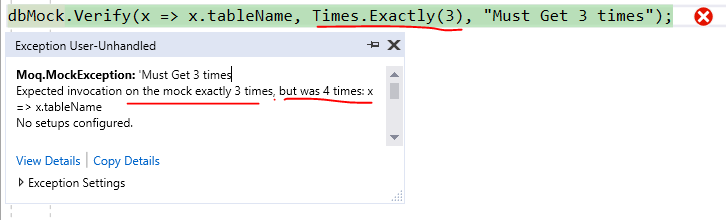


dbMock.Verify(x => x.getFlag(It.IsAny<int>()), Times.AtLeast(3), "At lease 3 times");



也可以对属性进行验证，对属性的验证其实是对get的次数, 而不会对set的次数

dbMock.Verify(x => x.tableName, Times.Exactly(3), "Must Get 3 times");



bool ff = TOperation.getFlag(dbMock.Object, 2);

dbMock.Verify(x => x.tableName, Times.Exactly(3), "Must Get 3 times");

public static bool getFlag(TeacherDB db, int a)

{

db.tableName = "HelloWorld";

db.tableName = "BlackFriday";

var tn = db.tableName; - 1

tn = db.tableName; - 2

tn = db.tableName; - 3

bool f = db.getFlag(a);

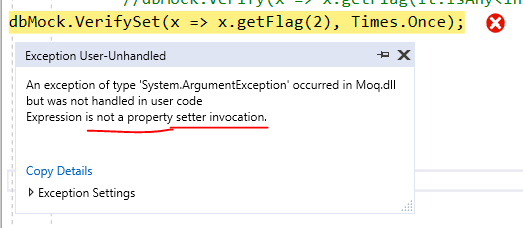
return f;

}

测试通过， get 3次， 而set 2次不会被计算在内

**.VerifySet() -** 主要是针对属性的Set 进行验证，由于方法没有Setter,所以不能使用方法。

dbMock.VerifySet(x => x.getFlag(2), Times.Once);



bool ff = TOperation.getFlag(dbMock.Object, 2);

dbMock.VerifySet(x => x.tableName=It.IsAny<string>(), Times.Exactly(5), "At lease set 5 times");

public static bool getFlag(TeacherDB db, int a)

{

db.tableName = "HelloWorld";

db.tableName = "BlackFriday99";

db.tableName = "Black";

db.tableName = "BlackCoom";

var tn = db.tableName;

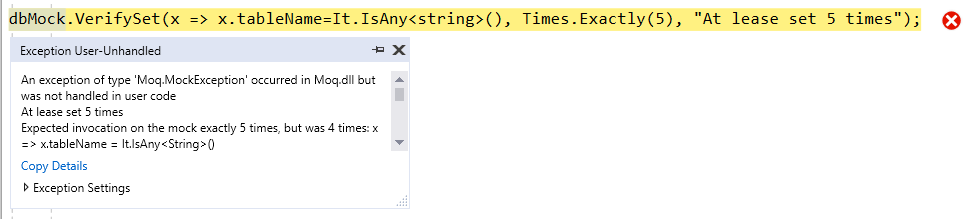
tn = db.tableName;

tn = db.tableName;

bool f = db.getFlag(a);

return f;

}



dbMock.VerifySet(x => x.tableName=

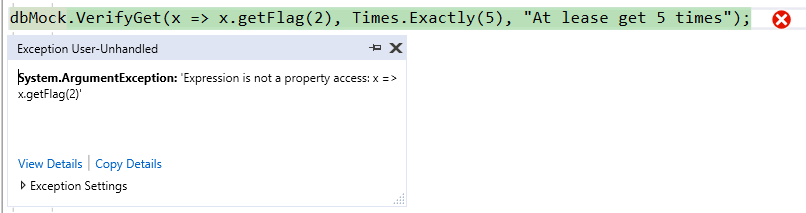
It.IsRegex(@"^(Black|Hello).\*$",System.Text.RegularExpressions.RegexOptions.IgnoreCase), Times.AtLeast(5),

"At lease set 5 times"

);

**.VerifyGet() -** 主要是针对属性的Get 进行验证，由于方法没有Getter,所以不能使用方法。

dbMock.VerifyGet(x => x.getFlag(2), Times.Exactly(5), "At lease get 5 times"); - 不能Get方法



VerifyGet() – 简单，只能对属性访问的次数进行验证，没有其他的功能

dbMock.VerifyGet(x => x.tableName, Times.Exactly(5), "At lease get 5 times");

**It.xxx () –** 主要用来模拟或者验证参数值或者返回值

It.Is<int>(a => a > 100);

It.IsAny<string>();

It.IsAny<DateTime>();

It.IsIn<string>("a", "b", "c");

It.IsIn<int>(1, 2, 3);

It.IsInRange<int>(10, 20, Range.Inclusive);

It.IsNotNull<Teacher>();

It.IsRegex(@"^.\*$", RegexOptions.IgnoreCase);

**FluentAssertions: Nuget Package**

**有比较好用的断言： Should().Be()**

dbMock.Object.tableName.Should().Be("HelloWorld1");

**Mock<T>() - Mock.Object 的理解：**

我们来看看一个测试案例:

public interface iShow

{

int sum(int a, int b);

}

public class : iShow

{

public TShow()

{

this.total = 10000;

}

public int total { get; set; }

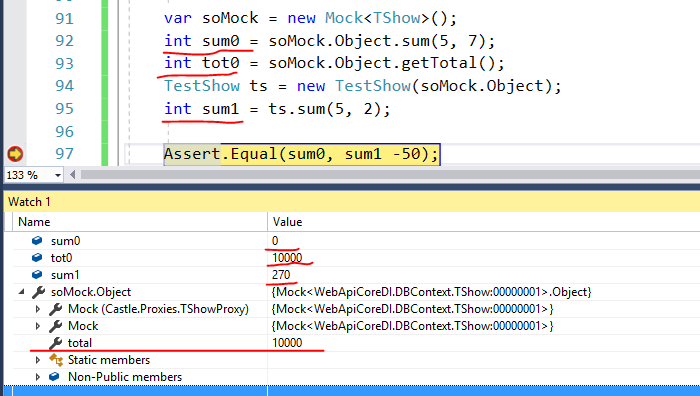
public virtual int sum(int a, int b)

{

int t = this.total + a + b;

return t;

}

 public int getTotal()

{

return this.total;

}

}

public class TestShow

{

public iShow ishow { get; set; }

public TestShow(iShow sh)

{

this.ishow = sh;

}

public int sum(int a, int b)

{

int t = 200 + a \* 10 + b \* 10;

return t + ishow.sum(a, b);

}

}

public async void Test1()

{

var soMock = new Mock<TShow>(); - 模拟出来的对象，实际是构造一个 new TShow()对象，

int sum0 = soMock.Object.sum(5, 7); - 模拟对象sum()方法，按常理应该返回 100+5+7=112，实际返回 0

int tot0 = soMock.Object.getTotal(); - 模拟对象getTotal() 返回 10000

TestShow ts = new TestShow(soMock.Object);

int sum1 = ts.sum(5, 2); - 实际要测试对象的方法sum(), 按常理 200+50+20+(10000+5+2)，实际返回 270

Assert.Equal(sum0, sum1 -50);

}

上面的测试结果，问题出在哪里？

Mock<T>() 对象与 方法和属性是否是 Virtual 有极大的关系

1. 如果Mock对象的方法是Virtual方法，则方法里的代码逻辑被 Mock.Setup() 覆盖掉，如果没有Setup,则是空函数。返回-返回类型的默认值。

var soMock = new Mock<TShow>();

soMock.Setup(x => x.sum(It.IsAny<int>(), It.IsAny<int>())).Returns((int a, int b) => a + b);

soMock.Setup(x => x.sum(5, 7)).Returns(**550**);

int sum0 = soMock.Object.sum(5, 7); - 返回 550; .Setup(x => x.sum(5, 7)).Returns(**550**);

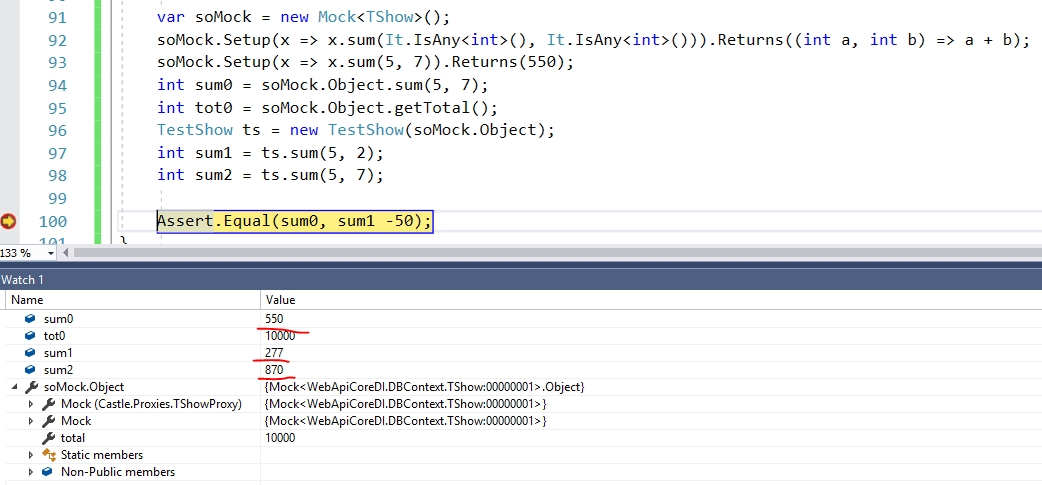
int tot0 = soMock.Object.getTotal();

TestShow ts = new TestShow(soMock.Object);

int sum1 = ts.sum(5, 2); - 返回： 200+50+20+(5+2); (5+2) = Returns((int a, int b) => a + b);

int sum2 = ts.sum(5, 7); - 返回： 200+50+20+550； 550=Setup(x => x.sum(5, 7)).Returns(**550**);

Assert.Equal(sum0, sum1 -50);



1. 如果Mock对象的方法是不带Virtual的方法

public interface iShow

{

int sum(int a, int b);

}

public class TShow : iShow

{

public TShow()

{

this.total = 10000;

}

public int total { get; set; }

public int sum(int a, int b)

{

int t = this.total + a + b;

return t;

}

public int getTotal()

{

return this.total;

}

}

public class TestShow

{

public iShow ishow { get; set; }

public TestShow(iShow sh)

{

this.ishow = sh;

}

public int sum(int a, int b)

{

int t = 200 + a \* 10 + b \* 10;

return t + ishow.sum(a, b);

}

}

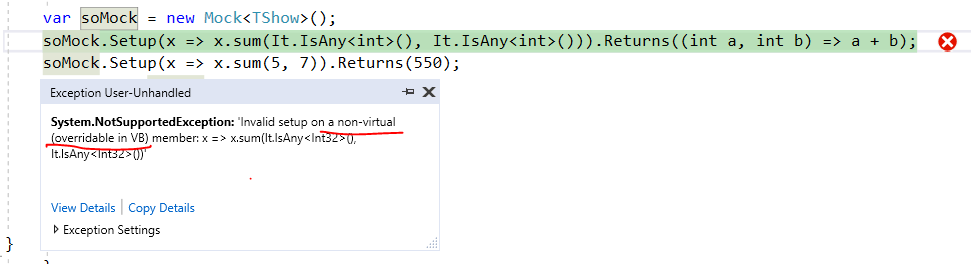
var soMock = new Mock<TShow>();

soMock.Setup(x => x.sum(It.IsAny<int>(), It.IsAny<int>())).Returns((int a, int b) => a + b);

soMock.Setup(x => x.sum(5, 7)).Returns(550);

如果不是Virtual方法或者属性，是不允许Mock重新覆盖定义的。如果是Virtual方法或者属性,则可以，而且不定义的话实

际上已经被Mock使用默认值替换掉了



1. 如果Mock对象的方法是不带Virtual的方法

var soMock = new Mock<TShow>();

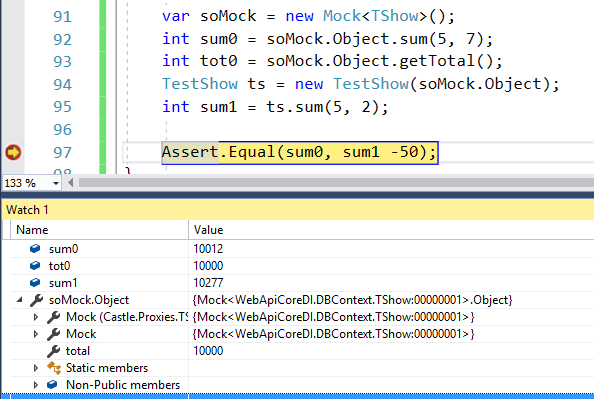
int sum0 = soMock.Object.sum(5, 7);

int tot0 = soMock.Object.getTotal();

TestShow ts = new TestShow(soMock.Object);

int sum1 = ts.sum(5, 2);

Assert.Equal(sum0, sum1 -50);



1. 测试对象的方法，带不带Virtual 跟Mock对象的方法带不带Virtual，毫无关系

public interface iShow

{

int sum(int a, int b);

}

public class TShow : iShow

{

public TShow()

{

this.total = 10000;

}

public int total { get; set; }

public int sum(int a, int b)

{

int t = this.total + a + b;

return t;

}

public int getTotal()

{

return this.total;

}

}

public class TestShow

{

public iShow ishow { get; set; }

public TestShow(iShow sh)

{

this.ishow = sh;

}

public **virtual** int sum(int a, int b)

{

int t = 200 + a \* 10 + b \* 10;

return t + ishow.sum(a, b);

}

}

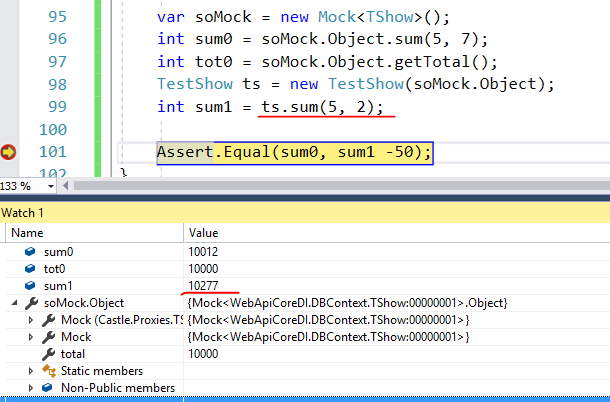
var soMock = new Mock<TShow>();

int sum0 = soMock.Object.sum(5, 7);

int tot0 = soMock.Object.getTotal();

TestShow ts = new TestShow(soMock.Object); - 这个是测试对象，而不是Mock对象。

int sum1 = ts.sum(5, 2); -这个是测试对象的方法,并不因为是virtual而返回0, 而是返回 200+50+20+(1000+5+2)



-----------------------------------------

关键是Mock对象的方法是否带有virtual

var soMock = new Mock<TShow>();

int sum0 = soMock.Object.sum(5, 7);

int tot0 = soMock.Object.getTotal();

TestShow ts = new TestShow(soMock.Object);

int sum1 = ts.sum(5, 2);

Assert.Equal(sum0, sum1 -50);

public class TShow : iShow

{

public TShow()

{

this.total = 10000;

}

public int total { get; set; }

public **virtual** int sum(int a, int b) - 如果带有Virtual, Mock又没有设置Setup, 则返回 default(int) = 0

{

int t = this.total + a + b;

return t;

}

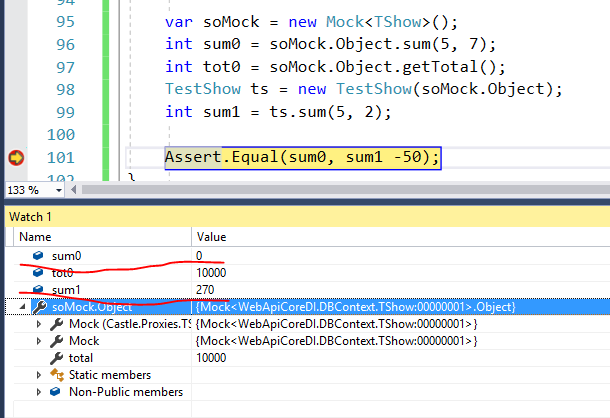
public int getTotal()

{

return this.total;

}

}



1. 测试对象的方法，返回的是对象，对象的默认值是 null

public interface iShow

{

Result sum(int a, int b);

}

public class TShow : iShow

{

public TShow()

{

this.total = 10000;

}

public int total { get; set; }

public virtual Result sum(int a, int b) – Mock 对象的 Virtual 方法，如果没有Setup 则返回默认 null

{

Result t = new Result

{

width = this.total + a,

height = this.total + b

};

return t;

}

public int getTotal()

{

return this.total;

}

}

public class TestShow

{

public iShow ishow { get; set; }

public TestShow(iShow sh)

{

this.ishow = sh;

}

public Result sum(int a, int b)

{

Result t = new Result

{

width = a \* 10,

height = b \* 10

};

t.Add(this.ishow.sum(a, b)); - t.Add(null) 抛出错误

return t;

}

}

public class Result

{

public int width { get; set; }

public int height { get; set; }

public void Add(Result t)

{

this.width += t.width;

this.height += t.height;

}

}

var soMock = new Mock<TShow>();

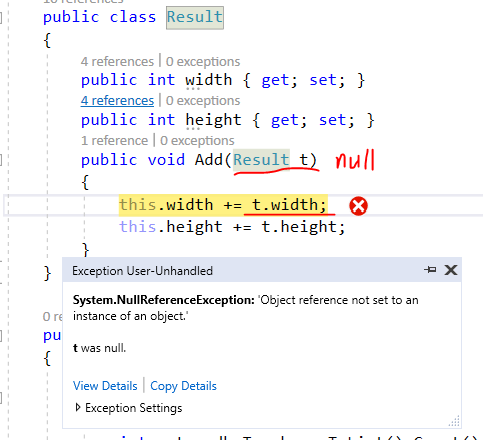
Result sum0 = soMock.Object.sum(5, 7);

int tot0 = soMock.Object.getTotal();

TestShow ts = new TestShow(soMock.Object);

Result sum1 = ts.sum(5, 2);

Assert.Equal(sum0, sum1);



1. Mock<T>(params[] object args) – Mock对象可以是带参数的构造函数所构造

public class TShow : iShow

{

public Result;

public int total;

public TShow()

{

this.total = 10000;

}

public TShow(int a)

{

this.total = a;

}

public TShow(int a, Result t) { - 带参数的构造函数

this.result = t;

this.total = a;

}

public Result sum(int a, int b)

{

Result t = new Result

{

width = this.total + a,

height = this.total + b

};

return t;

}

public int getTotal()

{

return this.total;

}

}

Result p1 = new Result { width = 2010, height = 4080 };

var soMock = new Mock<TShow>(6000, p1); - 可以通过带参数的构造函数来构造Mock对象

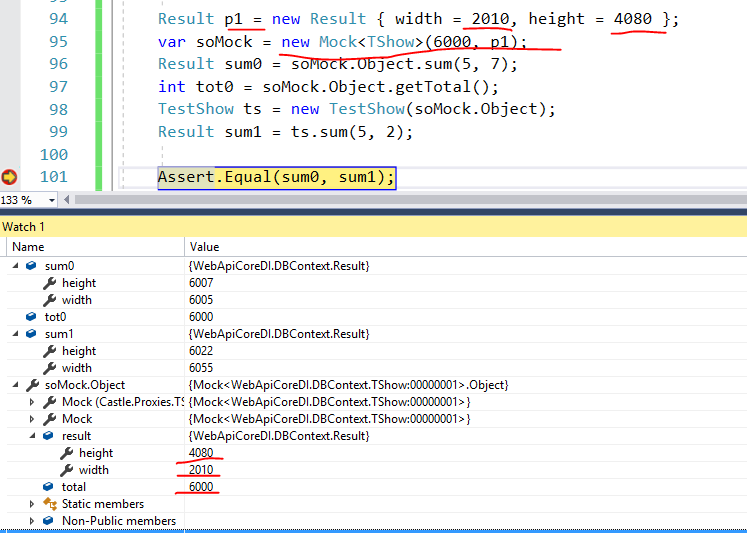
Result sum0 = soMock.Object.sum(5, 7);

int tot0 = soMock.Object.getTotal();

TestShow ts = new TestShow(soMock.Object);

Result sum1 = ts.sum(5, 2);

Assert.Equal(sum0, sum1);



Public virtual Result **sum**(int a, int b)

{

Result t = new Result

{

width = this.total + a,

height = this.total + b

};

return t;

}

Result p1 = new Result { width = 2010, height = 4080 };

var soMock = new Mock<TShow>(6000, p1);

soMock.Setup(x => x.**sum**(It.IsAny<int>(), It.IsAny<int>()))

.Returns((int a, int b) => new Result { width=2222, height=3333});

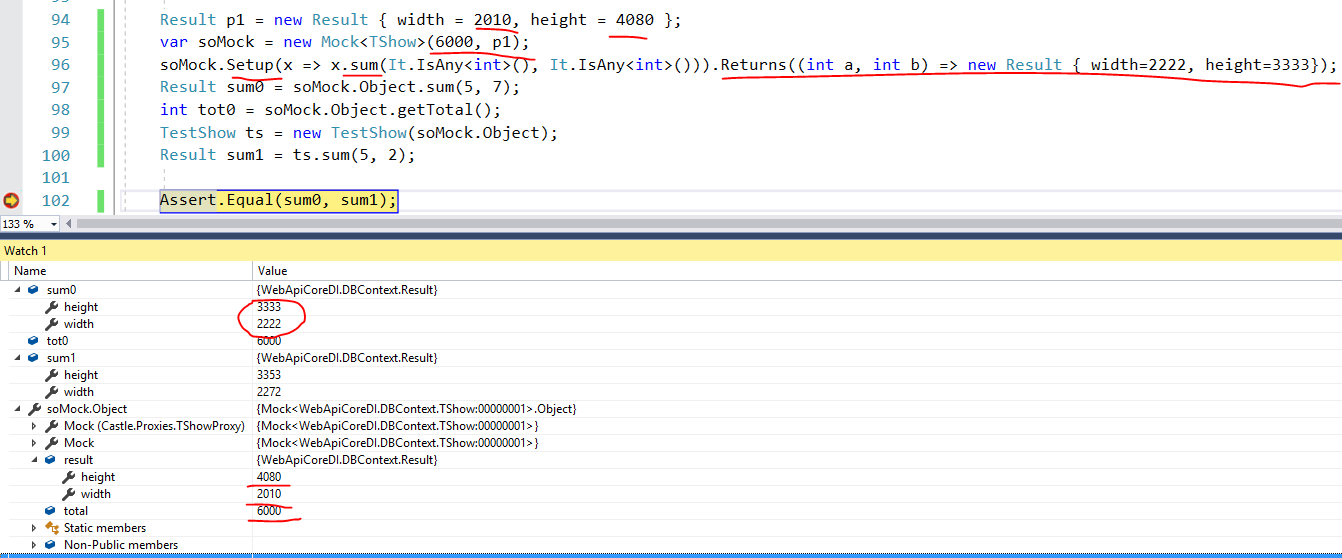
Result sum0 = soMock.Object.sum(5, 7);

int tot0 = soMock.Object.getTotal();

TestShow ts = new TestShow(soMock.Object);

Result sum1 = ts.sum(5, 2);

Assert.Equal(sum0, sum1);



如果Mock ASP.Net Core Request(HttpRequest) Headers

**如果在我们的WebAPI 的 Controller 里通常需要使用 Request 对象**

HttpRequest 是抽象Class

[Fact]

public void Get\_Return\_OK()

{

var userTypeMock = new Mock<DbSet<UserType>>();

userTypeMock.Object.AddRange(

new UserType { Id = 1, Title = "Student" },

new UserType { Id = 2, Title = "Parent" },

new UserType { Id = 3, Title = "Teacher" },

new UserType { Id = 4, Title = "TeacherAssistant" }

);

var userMock = new Mock<DbSet<User>>();

userMock.Object.AddRange(

new User { Id = 1, FirstName = "Student"},

new User { Id = 2, FirstName = "Parent" },

new User { Id = 3, FirstName = "Teacher"},

new User { Id = 4, FirstName = "TA"}

);

var courseMock = new Mock<DbSet<Course>>();

courseMock.Object.AddRange(

new Course { Id = 1, Title = "Test Course", TaId = 4, TeacherId = 3 }

);

var classMock = new Mock<DbSet<Classes>>();

classMock.Object.AddRange(

new Classes { Id = 11, Title = "Test Class", Status = 1, Deleted = 0 }

);

var classStudentMock = new Mock<DbSet<ClassStudent>>();

classStudentMock.Object.AddRange(

new ClassStudent { Id = 111, ClassId = 11, StudentId = 1 }

);

var dbMock = new Mock<MyDBContext>();

dbMock.Setup(x => x.UserTypes).Returns(userTypeMock.Object);

dbMock.Setup(x => x.Users).Returns(userMock.Object);

dbMock.Setup(x => x.Courses).Returns(courseMock.Object);

dbMock.Setup(x => x.Classes).Returns(classMock.Object);

dbMock.Setup(x => x.ClassStudents).Returns(classStudentMock.Object);

ClassesController classController = new ClassesController(dbMock.Object);

**classController.ControllerContext = new ControllerContext();**

**classController.ControllerContext.HttpContext = new ControllerContext ();**

**classController.ControllerContext.HttpContext.Request.Headers.Add("userid", "1");**

IActionResult result = classController.Get();

Assert.IsType<OkObjectResult>(result);

}

如果Mock Entities 有使用ToList(),Where()方法，则必须要将Entities设置成AsQueryable

public static class MockExtensions

{

public static void AddList<T>(this Mock mock, params T[] entities)

{

var list = entities.ToList<T>().AsQueryable();

mock.As<IQueryable<T>>().Setup(x => x.Provider).Returns(list.Provider);

mock.As<IQueryable<T>>().Setup(x => x.Expression).Returns(list.Expression);

mock.As<IQueryable<T>>().Setup(x => x.ElementType).Returns(list.ElementType);

mock.As<IQueryable<T>>().Setup(x => x.GetEnumerator()).Returns(list.GetEnumerator());

}

public static void AsQuerable<T>(this Mock mock)

{

var list = new List<T>().AsQueryable();

mock.As<IQueryable<T>>().Setup(x => x.Provider).Returns(list.Provider);

mock.As<IQueryable<T>>().Setup(x => x.Expression).Returns(list.Expression);

mock.As<IQueryable<T>>().Setup(x => x.ElementType).Returns(list.ElementType);

mock.As<IQueryable<T>>().Setup(x => x.GetEnumerator()).Returns(list.GetEnumerator());

}

}

// mock entities

var userTypeMock = new Mock<DbSet<UserType>>();

UserType usertype1 = new UserType { Id = 1, Title = "Student" };

UserType usertype2 = new UserType { Id = 2, Title = "Parent" };

UserType usertype3 = new UserType { Id = 3, Title = "Teacher" };

UserType usertype4 = new UserType { Id = 4, Title = "TeacherAssistant" };

userTypeMock.AddList(usertype1, usertype2, usertype3, usertype4);

[HttpGet("collection")]

public IActionResult GetUsers()

{

try

{

var studentList = this.DB.Users.**Where**(p => p.Status == 1).

Select(p => new UserDisplay {id = p.Id, email = p.Email, phone = p.Phone});

var teacherList = this.DB.Users.**Where**(p => p.Status == 1).

Select(p => new UserDisplay { id = p.Id, email = p.Email, phone = p.Phone});

var taList = this.DB.Users.**Where**(p => p.Status == 1).

Select(p => new UserDisplay { id = p.Id, email = p.Email, phone = p.Phone});

var courseList = this.DB.Courses.**Where**(p => p.Status == 1).

Select(p => new CourseDisplay { id = p.Id, title = p.Title });

var collection = new

{

students = studentList,

teachers = teacherList,

tas = taList,

courses = courseList

};

return Ok(collection);

}

catch (Exception err)

{

return BadRequest(err.ToString());

}

}

如果Mock Interface 与 Class 的区别：

Mock Interface

public interface ITwilioHandler

{

bool SendMessage(string phone, string body); //注意并没有指定 virtual, 而且接口也不允许指定

}

var mockHandler = new Mock<ITwilioHandler>();

mockHandler.Setup(x => x.SendMessage(It.IsAny<string>(), It.IsAny<string>())).Returns(true);

注意并没有指定 virtual，而 Mock 是允许并不会抛出错误。

Mock Class

public class MyTwilioHandler : ITwilioHandler

{

public MyTwilioHandler()

{

TwilioClient.Init("ACe8ae0e37af2f2cc9a017321e4af388b8", "9891a974d431f249340b52a38625cf24");

}

public bool SendMessage(string phone, string body)

{

var message = MessageResource.Create(

new PhoneNumber(phone),

from: new PhoneNumber("+16042628875"),

body: body

);

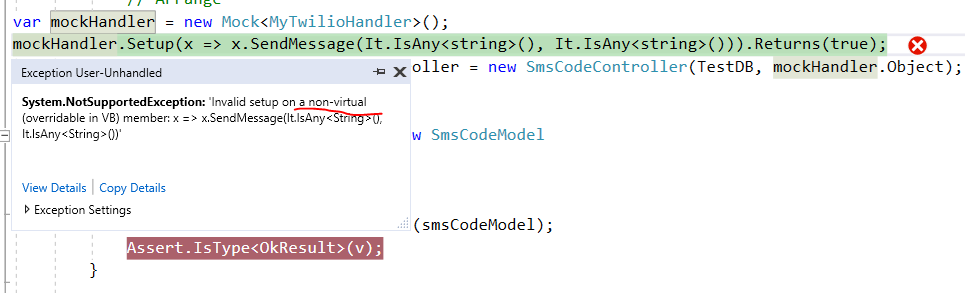
return true;

}

}

var mockHandler = new Mock<MyTwilioHandler>();

mockHandler.Setup(x => x.SendMessage(It.IsAny<string>(), It.IsAny<string>())).Returns(true);



var mockHandler = new Mock<MyTwilioHandler>();

//mockHandler.Setup(x => x.SendMessage(It.IsAny<string>(), It.IsAny<string>())).Returns(true);

如果不 Mock - SendMessage 方法，则会实实在在的执行这个方法里面的代码逻辑，对于单元测试来说，需要评估是否需要执行里面的代码逻辑。

public class MyTwilioHandler : ITwilioHandler

{

public MyTwilioHandler()

{

TwilioClient.Init("ACe8ae0e37af2f2cc9a017321e4af388b8", "9891a974d431f249340b52a38625cf24");

}

public virtual bool SendMessage(string phone, string body) – 如果指定 virtual

{

var message = MessageResource.Create(

new PhoneNumber(phone),

from: new PhoneNumber("+16042628875"),

body: body

);

return true;

}

}

如果对指定为 virtual bool SendMessage， 则不管Mock.Setup与否，都不会执行函数的代码逻辑。

没有 Mock.Setup， 则永远返回bool默认值false

//mockHandler.Setup(x => x.SendMessage(It.IsAny<string>(), It.IsAny<string>())).Returns(true);