

从零开始实现DDD应用

第一章 Entity

作者：贺传珺

项目概览

- 项目名称：润建房屋租赁系统
- 项目需求：员工和公司能在平台上发布空置的房产进行出租，管理员审核之后即可发布。



项目环境要求

- Visual Studio 2019, 最好是2022
- PostgreSQL (其他关系型数据库也行, 代码上会有细微区别)
- C#9+ (Record类型)
- .Net 6 (最新的7也行)

新建项目

- 新建一个ASP.NET Core Web API项目， 名称为RJRentalOfHousing

配置新项目

ASP.NET Core Web API C# Linux macOS Windows 云 服务 Web WebAPI

项目名称(J)
RJ-Rental-Of-Housing

位置(L)
...

解决方案名称(M) ⓘ
RJ-Rental-Of-Housing

☐ 将解决方案和项目放在同一目录中(D)

新建项目

- 在解决方案中新增一个类库项目，命名为RJRentalOfHousing.Domain（截图略）
- 在解决方案中新增一个xUnit测试项目，命名为RJRentalOfHousing.Tests

配置新项目

xUnit 测试项目

C#

Linux

macOS

Windows

测试

项目名称(J)

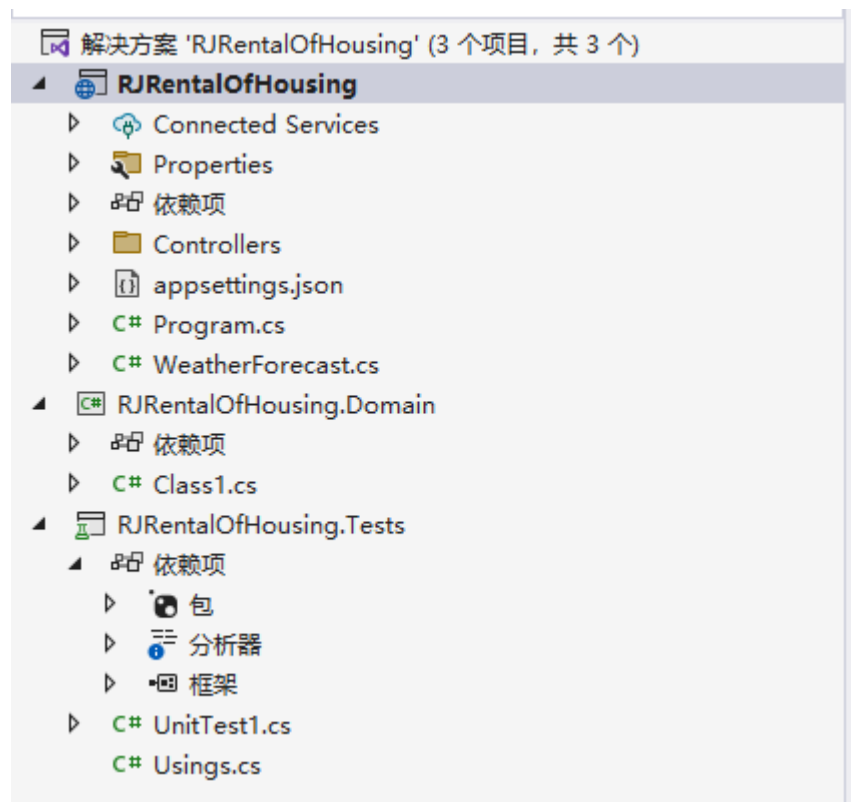
RJRentalOfHousing.Tests

位置(L)

...

新建项目

- 新建项目完成之后，项目结构是这样的：



项目引用关系

- Web项目引用Domain
- Tests项目引用Domain
- Domain项目不要有任何项目引用，保持领域层足够干净

清理项目

- 把Domain和Tests中的Class1.cs和UnitTest1.cs文件删除

新建Entity（实体）

- 在Domain中新建一个类，命名为Apartment

```
using Domain
namespace RJRentalOfHousing.Domain
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class Apartment
    {
    }
}
```

添加字段

让我们想象一下，一套要出租的房子有什么属性？

- 面积
- 地点
- 租金
- 押金
- 发布者
- 备注
- 是否有家具
- 是否有家电
- 审核人
- 图片
-

添加字段

```
1 namespace RJRentalOfHousing.Domain
2 {
3     0 个引用 | 0 项更改 | 0 名作者, 0 项更改
4     public class Apartment
5     {
6         0 个引用 | 0 项更改 | 0 名作者, 0 项更改
7         public Guid Id { get; internal set; }
8
9         0 个引用 | 0 项更改 | 0 名作者, 0 项更改
10        public decimal Areas { get; internal set; }
11
12        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
13        public string Address { get; internal set; }
14
15        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
16        public decimal Rent { get; internal set; }
17
18        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
19        public decimal Deposit { get; internal set; }
20
21        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
22        public Guid Owner { get; internal set; }
23
24        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
25        public string Remark { get; internal set; }
26
27        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
28        public Guid ApprovedBy { get; internal set; }
29    }
30 }
```

模型里的Set方法都是Internal的，表示只能在程序集内部调用Domain的Set方法。

平时我们开发的时候习惯于把字段的Set方法也设置为Public，在模型外部或者程序集外部去改变字段的值。其实这对我们实现业务逻辑没有什么好处。

举个例子，把我们每个人看做一个类的实例，我们都有自己的身高体重，有呼吸心跳，有血氧有血糖有肌酐有转氨酶等。这些都是我们实例中的属性或者字段。这些属性或者字段只能通过我们自身去调节或改变，你能想象如果哪一天有什么东西能直接改变你的这些属性会有什么效果吗？当然，如果能把我的身高变成1米8我就没意见（手动狗头）。

附：C#中访问修饰符的文档：[访问修饰符 - C# 编程指南 | Microsoft Learn](#)

添加业务

我们现在有一个最基础的业务信息：Id不能为空，在代码里我们要做如下限制：

1 个引用 10 项更改 10 名作者，0 项更改

```
public class Apartment
```

```
{
```

0 个引用 10 项更改 10 名作者，0 项更改

```
public Apartment(Guid id)
```

```
{
```

```
    if (id == default)
```

```
        throw new ArgumentNullException(nameof(id), "Id不能为空");
```

```
    Id = id;
```

```
}
```

现在我们在代码里有了一个最基本的业务检查，让这个实体不会处于无效的状态，接下来让我们看看如何在实体中添加更多限制。

添加业务

现在要求房屋拥有者 (Owner) 也不能为空，于是我们要在构造函数中再加一个限制：

```
1 | 1引用10 项更改10 名作者，0 项更改
public class Apartment
{
    0 个引用 | 0 项更改 | 0 名作者，0 项更改
    public Apartment(Guid id, Guid ownerId)
    {
        if (id == default)
            throw new ArgumentNullException(nameof(id), "Id不能为空");
        if (ownerId == default)
            throw new ArgumentNullException(nameof(ownerId), "所有者不能为空");
        Id = id;
        Owner = ownerId;
    }
}
```

现在我们的实体在创建的时候就能保证它的有效性，但是这也带来新的问题，构造函数的两个参数都是Guid类型，我们调用的时候可能会这样写：Apartment A=new Apartment(Guid,Guid); 除非你翻看定义，要不一眼是看不出哪个是Id，哪个是OwnerId，非常容易搞混。并且后续肯定会有更多的业务限制加入，构造函数会臃肿不堪。如何解决呢？这时候值对象 (Value Object) 闪亮登场。

值对象

什么是值对象（Value Object）？它和实体有什么区别？

值对象就是一组值，没有它自己的ID也没有状态，只有值。那和实体有什么区别呢？

还是举个例子来说明：我们用纸币付款的时候，纸币就是值对象，因为在这个情景中，两张面值相等的纸币就是相同的。

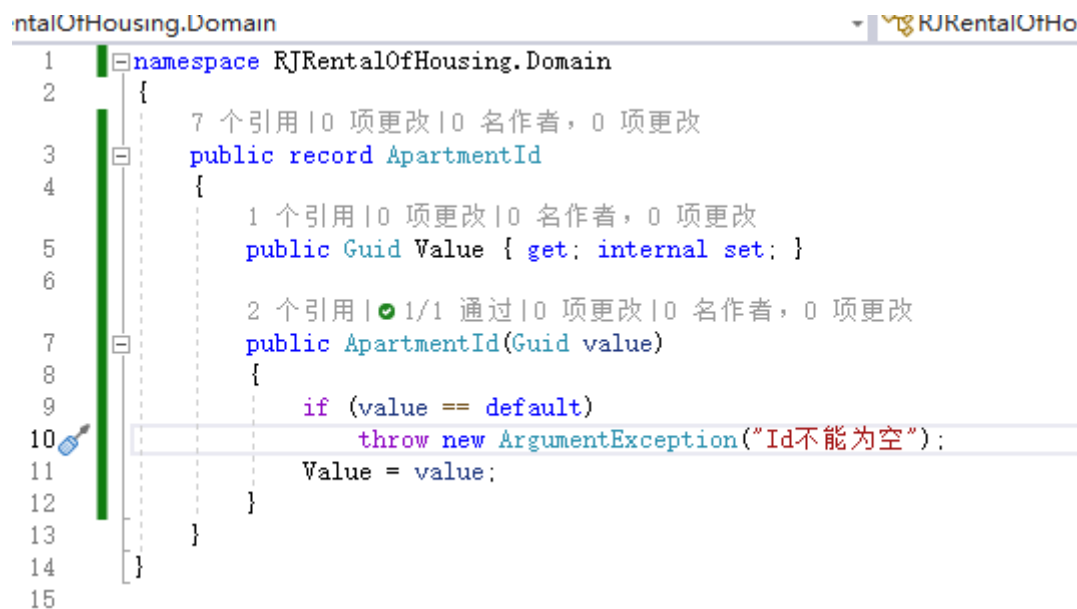
什么时候纸币是实体呢？应该是作为收藏品的时候，因为纸币的面值相等，但是编号不同，所以即使面值相等不同编号的两张纸币也不相同。

思考题：什么场景中人是值对象？那什么场景中是实体？

值对象

如何在代码中实现值对象？

首先，我们在Domain中新建一个类，命名为ApartmentId，代码如下：



```
1 namespace RJRentalOfHousing.Domain
2 {
3     public record ApartmentId
4     {
5         public Guid Value { get; internal set; }
6
7         public ApartmentId(Guid value)
8         {
9             if (value == default)
10                 throw new ArgumentException("Id不能为空");
11             Value = value;
12         }
13     }
14 }
15
```

值对象

刚才我们说到，两个值对象中的值相等，这两个对象就视为同一个对象。现在我们新建一个单元测试，测试一下刚才新建的值对象能否符合要求

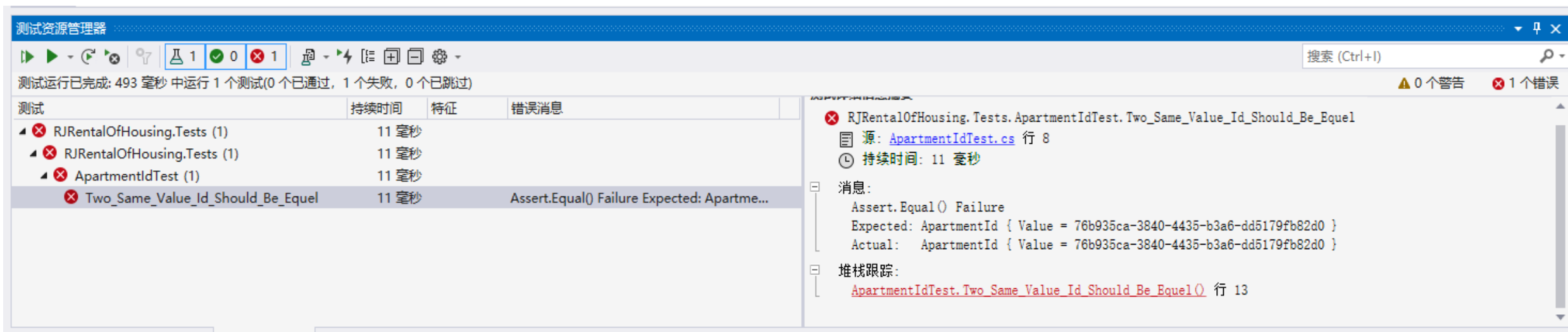
- 在Tests项目中，新建一个单元测试命名为ApartmentIdTest，代码如下

```
using RJRentalOfHousing.Domain;

namespace RJRentalOfHousing.Tests
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class ApartmentIdTest
    {
        [Fact]
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public void Two_Same_Value_Id_Should_Be_Equal()
        {
            Guid id = Guid.NewGuid();
            ApartmentId apartmentId1 = new ApartmentId(id);
            ApartmentId apartmentId2 = new ApartmentId(id);
            Assert.Equal(apartmentId1, apartmentId2);
        }
    }
}
```


值对象

运行单元测试，发现居然测试失败



测试资源管理器

测试运行已完成: 493 毫秒 中运行 1 个测试(0 个已通过, 1 个失败, 0 个已跳过)

测试	持续时间	特征	错误消息
✖ RJRentalOfHousing.Tests (1)	11 毫秒		
✖ RJRentalOfHousing.Tests (1)	11 毫秒		
✖ ApartmentIdTest (1)	11 毫秒		
✖ Two_Same_Value_Id_Should_Be_Equal	11 毫秒		Assert.Equal() Failure Expected: Apartme...

消息:

```
Assert.Equal() Failure
Expected: ApartmentId { Value = 76b935ca-3840-4435-b3a6-dd5179fb82d0 }
Actual:   ApartmentId { Value = 76b935ca-3840-4435-b3a6-dd5179fb82d0 }
```

堆栈跟踪:

```
ApartmentIdTest.Two_Same_Value_Id_Should_Be_Equal() 行 13
```

这就违背了上面说的，两个值对象的值相等，就视为相同的对象。原因在于Class实例化两次，即使是值相等，两个实例也不相等。接下来看如何解决这个小麻烦。

值对象

上面说到一个类的两个实例即使是值相等也不相等，这样会导致测试失败，如何解决这个问题？方法有两种：

- 运算符重载
- 使用Record类型

运算符重载的方法较为复杂，这里不多做介绍，有兴趣自己去百度。建议使用Record类型来解决这个问题。现在我们来修改我们的值对象，把class改为record。然后再次运行单元测试。

5 个引用 | 0 项更改 | 0 名作者，0 项更改

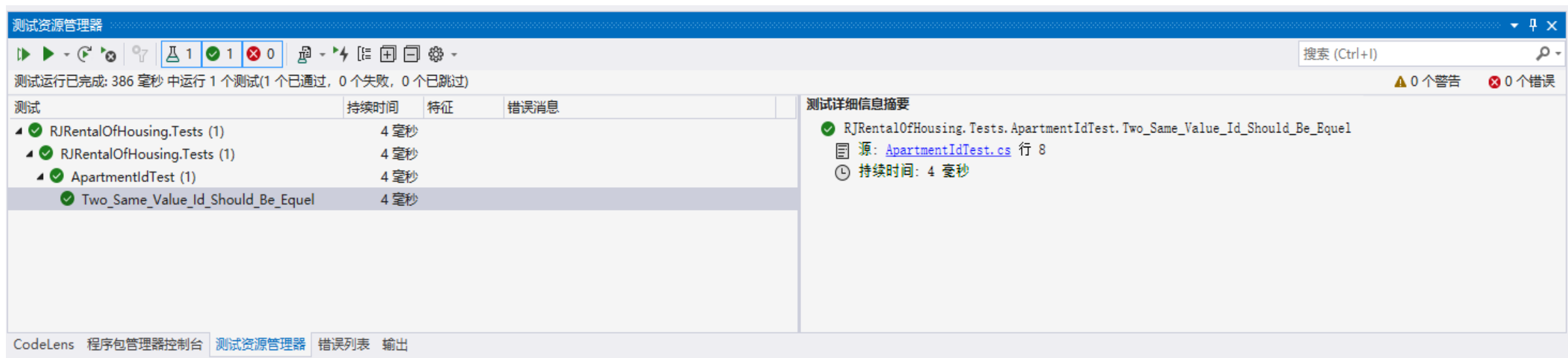
```
public record ApartmentId
```

```
{
```

```
    // ...
```

值对象

再次运行单元测试，测试通过



The screenshot shows the Visual Studio Test Explorer window. The top bar indicates the test run is complete: "测试运行已完成: 386 毫秒 中运行 1 个测试(1 个已通过, 0 个失败, 0 个已跳过)". The main pane displays a list of tests, with the selected test "Two_Same_Value_Id_Should_Be_Equal" highlighted. The right pane shows the test details, including the source file "ApartmentIdTest.cs" and the duration "4 毫秒".

测试	持续时间	特征	错误消息
✓ RJRentalOfHousing.Tests (1)	4 毫秒		
✓ RJRentalOfHousing.Tests (1)	4 毫秒		
✓ ApartmentIdTest (1)	4 毫秒		
✓ Two_Same_Value_Id_Should_Be_Equal	4 毫秒		

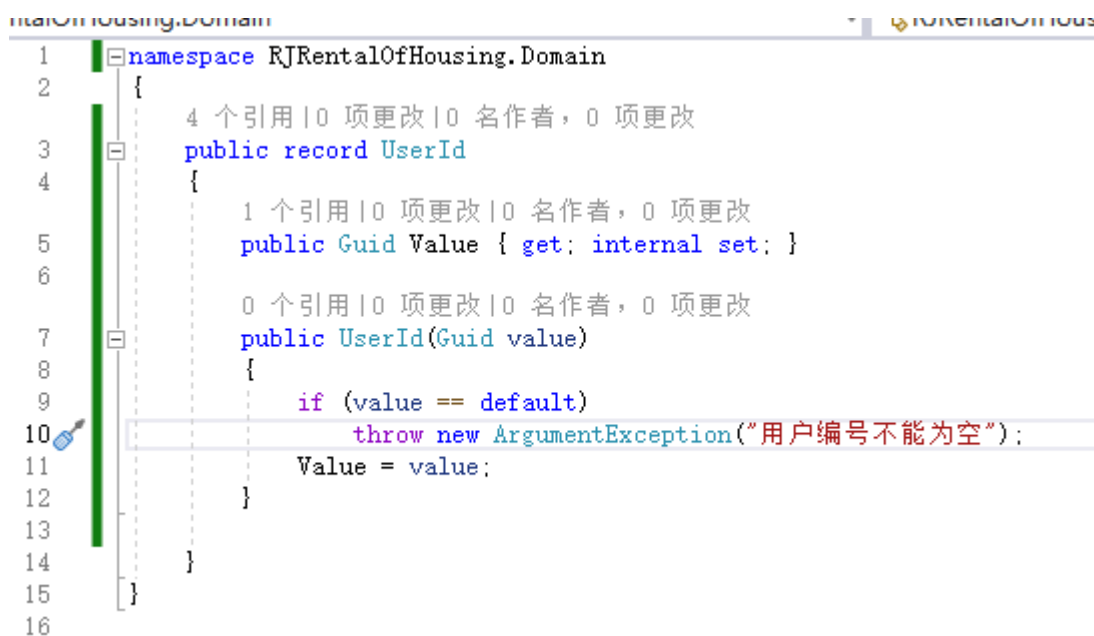
测试详细信息摘要

- ✓ RJRentalOfHousing.Tests.ApartmentIdTest.Two_Same_Value_Id_Should_Be_Equal
 - 源: [ApartmentIdTest.cs](#) 行 8
 - 持续时间: 4 毫秒

是不是很神奇。关于更多Record类型的用法，请查看：[记录 - C# 参考 | Microsoft Learn](#)

值对象

刚才我们已经新建了一个Id的值对象，现在参考刚才新建的值对象，再在Domain中新建一个UserId的值对象。



```
1 namespace RJRentalOfHousing.Domain
2 {
3     4 个引用 | 0 项更改 | 0 名作者, 0 项更改
4     public record UserId
5     {
6         1 个引用 | 0 项更改 | 0 名作者, 0 项更改
7         public Guid Value { get; internal set; }
8
9         0 个引用 | 0 项更改 | 0 名作者, 0 项更改
10        public UserId(Guid value)
11        {
12            if (value == default)
13                throw new ArgumentException("用户编号不能为空");
14            Value = value;
15        }
16    }
```

值对象

我们刚才新建的两个值对象已经包含了业务所需要的限制，现在改造一下我们的实体：

```
Housing.Domain
RJRentalOfHousing.Domain
namespace RJRentalOfHousing.Domain
{
    1 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class Apartment
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Apartment(ApartmentId id, UserId ownerId)
        {
            Id = id;
            Owner = ownerId;
        }

        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public ApartmentId Id { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public decimal Areas { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public string Address { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public decimal Rent { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public decimal Deposit { get; internal set; }

        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public UserId Owner { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public string Remark { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public UserId ApprovedBy { get; internal set; }
    }
}
```

现在构造函数已经由GUID类型改为值对象类型，这样在创建的时候就很难搞混参数。并且每个值对象的业务限制都在它的定义里面，不会外包到其他地方，方便日后查看和维护。

添加业务

接下来让我们回到业务上，我们来分析一下还有那些业务规则，比如，面积（Areas）必须大于零。按照以前的做法，这个限制会在服务层实现，加一个判断语句，如果值大于0才能对字段进行赋值。虽然这是比较容易实现，但是这会使得这段代码失去领域含义，而且会将领域知识泄露到Domain层以外的地方。

在这里我们选择用值对象的方式来实现这个需求，在Domain新建一个类，命名为Area

添加业务

```
1 namespace RJRentalOfHousing.Domain
2 {
3     2 个引用 | 0 项更改 | 0 名作者, 0 项更改
4     public record Area
5     {
6         1 个引用 | 0 项更改 | 0 名作者, 0 项更改
7         public decimal Value { get; internal set; }
8
9         0 个引用 | 0 项更改 | 0 名作者, 0 项更改
10        public Area(decimal value)
11        {
12            if (value <= 0)
13                throw new ArgumentOutOfRangeException("面积必须大于0");
14            Value = value;
15        }
16    }
17 }
```

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public Apartment(ApartmentId id, UserId ownerId)
{
    Id = id;
    Owner = ownerId;
}

1 个引用 | 0 项更改 | 0 名作者, 0 项更改
public ApartmentId Id { get; internal set; }

0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public Area Areas { get; internal set; }
```

同时修改Apartment, 将Areas的类型修改为刚才定义的Area

添加业务

我们的实体里还有两个金额：租金（Rent）和押金（Deposit），我们考虑这两个金额都不能小于0。由于金额（Money）在真实世界中是有负数的，所以先要建立一个Money的基类，在Domain中新建一个Money类型。

```
1 namespace RJRentalOfHousing.Domain
2 {
3     1 个引用 | 0 项更改 | 0 名作者, 0 项更改
4     public record Money
5     {
6         1 个引用 | 0 项更改 | 0 名作者, 0 项更改
7         public decimal Amount { get; internal set; }
8
9         0 个引用 | 0 项更改 | 0 名作者, 0 项更改
10        public Money(decimal amount)
11        {
12            Amount = amount;
13        }
14    }
15 }
```


添加业务

在Domain里新增一个Price类，让它继承Money类并且加入限制。

```
JRentalOfHousing.Domain RJRentalOfHousing.Domain.Pr  
1 namespace RJRentalOfHousing.Domain  
2 {  
3     1 个引用 | 0 项更改 | 0 名作者, 0 项更改  
4     public record Price : Money  
5     {  
6         0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
7         public Price(decimal amount) : base(amount)  
8         {  
9             if (amount < 0)  
10                throw new ArgumentException("价格不能小于0", nameof(amount));  
11         }  
12     }  
13 }
```

添加业务

将Apartment中的decimal类型替换成Price类型

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
public Price Rent { get; internal set; }  
  
0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
public Price Deposit { get; internal set; }
```

现在这两个金额类型已经被我们加上了业务限制，确保不会处于无效的状态

值对象

现在我们在大部分字段上加了业务，比如上面的两个金额类型的字段。但是在现实生活中，金额应该是能直接相加的，可实际上在我们的系统里需要做比较复杂的处理才行，让我们看看如何去改进。修改Money类，新增一些代码。

1 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public Money Add(Money m) => new Money(Amount + m.Amount);
```

1 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public Money Subtract(Money m) => new Money(Amount - m.Amount);
```

0 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public static Money operator +(Money m1, Money m2) => m1.Add(m2);
```

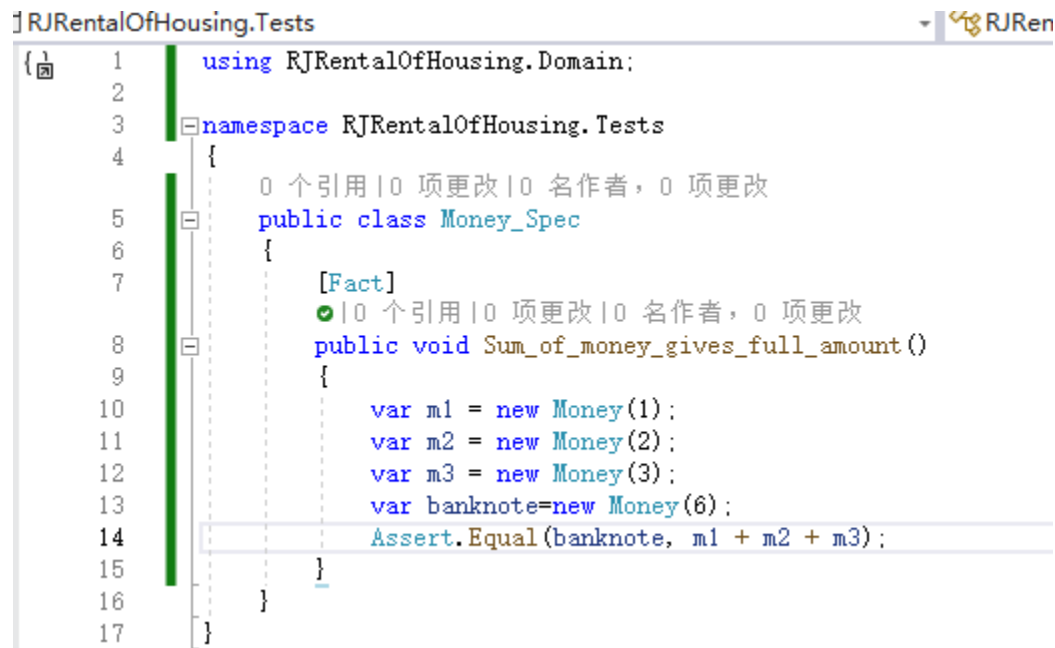
0 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public static Money operator -(Money m1, Money m2) => m1.Subtract(m2);
```

在这里又看到了一个陌生的关键字：operator，这个关键字的意思是运算符重载，和前面提到过的实现值对象相等是同一个东西，这里不具体展开讲解。有兴趣的可以看微软文档：[运算符重载 - C# 引用 | Microsoft Learn](#)

值对象

让我们来测试一下刚才的代码，在Tests项目中新建一个单元测试，命名为：Money_Spec



```
1  using RJRentalOfHousing.Domain;
2
3  namespace RJRentalOfHousing.Tests
4  {
5      public class Money_Spec
6      {
7          [Fact]
8          public void Sum_of_money_gives_full_amount()
9          {
10             var m1 = new Money(1);
11             var m2 = new Money(2);
12             var m3 = new Money(3);
13             var banknote=new Money(6);
14             Assert.Equal(banknote, m1 + m2 + m3);
15         }
16     }
17 }
```

值对象

测试结果符合我们的预期:

测试资源管理器

测试运行已完成: 410 毫秒 中运行 2 个测试(2 个已通过, 0 个失败, 0 个已跳过)

测试

测试	持续时间	特征	错误消息
✓ RJRentalOfHousing.Tests (2)	10 毫秒		
✓ RJRentalOfHousing.Tests (2)	10 毫秒		
✓ ApartmentIdTest (1)	5 毫秒		
✓ Money_Spec (1)	5 毫秒		
✓ Sum_of_money_gives_full_amount	5 毫秒		

组摘要

RJRentalOfHousing.Tests

组中的测试: 2

⌚ 总时长: 10 毫秒

结果

✓ 2 已通过

CodeLens 程序包管理器控制台 测试资源管理器 错误列表 ... 输出

工厂

回顾一下我们实际开发的时候，如果有一个字段的类型是Decimal或者DateTime或者其他数值类型。但是从边缘（英文：Edge，我们最常见的边缘是HTTP，不怎么常见的还有gRpc，WCF等）传过来的值往往不是这个类型，而是string或者其他，这时候就需要做一次数据转换而且要判断数据的有效性，这样的代码遍布整个系统。现在我们看看如何在值对象中，利用工厂解决这个问题。

还是以Money为例，修改代码。

工厂

```
namespace RJRentalOfHousing.Domain
{
    23 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public record Money
    {
        5 个引用 | HCJ, 1 天前 | 1 名作者, 1 项更改
        public decimal Amount { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static Money FromString(string amount) => new Money(decimal.Parse(amount));

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static Money FromDecamal(decimal amonut) => new Money(amonut);

        9 个引用 | 1/1 通过 | HCJ, 1 天前 | 1 名作者, 1 项更改
        protected Money(decimal amount)
        {
            if (decimal.Round(amount, 2) != amount)
                throw new ArgumentOutOfRangeException(nameof(amount), "金额不能超过两位小数");
            Amount = amount;
        }

        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Money Add(Money m) => new Money(Amount + m.Amount);

        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Money Subtract(Money m) => new Money(Amount - m.Amount);

        2 个引用 | 1/1 通过 | 0 项更改 | 0 名作者, 0 项更改
        public static Money operator +(Money m1, Money m2) => m1.Add(m2);

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static Money operator -(Money m1, Money m2) => m1.Subtract(m2);
    }
}
```

现在构造函数已经被修改成protected，除了子类无法在其他地方直接实例化这个实体。实例化的代码被移到了两个静态方法FromString和FromDecimal中。这样做能使得数据转换都在统一的地方处理，而且能根据不同情况添加一些限制。

现在在构造函数中，我们加入了一个业务限制：金额不能超过两位小数。这看上去很合理，如果我们没有海外业务的话。

这时候我们公司的租房业务要出海了，需要对接其他币种，想想看有哪些货币没有两位小数的？

接下来我们会用领域服务解决这个问题。

Tips：把构造函数设为Protected之后，单元测试会报错，注释掉就行。图里的FromDecamal打错了，应该是FromDecimal，之后的几页里同样。

领域服务

我们先用在值对象里加入币种的方式来解决这个需求，首先改造Money类，加上一个币种字段，并且给一个默认币种，设为人民币。

```
private const string DefaultCurrency = "CNY";
```

5 个引用 | HCJ, 1 天前 | 1 名作者, 1 项更改

```
public decimal Amount { get; internal set; }
```

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
public string CurrencyCode { get; internal set; }
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public static Money FromString(string amount) => new Money(decimal.Parse(amount));
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public static Money FromDecamal(decimal amonut) => new Money(amonut);
```

5 个引用 | HCJ, 1 天前 | 1 名作者, 1 项更改

```
protected Money(decimal amount)
{
    if (decimal.Round(amount, 2) != amount)
        throw new ArgumentOutOfRangeException(nameof(amount), "金额不能超过两位小数");
    Amount = amount;
}
```

人民币的代号不是RMB哟，不会有人不知道吧？不会吧？不会吧？

领域服务

接下来我们要对Money类的工厂方法和构造函数进行改造。

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public static Money FromString(string amount, string currency = DefaultCurrency) => new Money(decimal.Parse(amount), currency);
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public static Money FromDecamal(decimal amonut, string currency = DefaultCurrency) => new Money(amonut, currency);
```

5 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
protected Money(decimal amount, string currencyCode="CNY")
{
    if (decimal.Round(amount, 2) != amount)
        throw new ArgumentOutOfRangeException(nameof(amount), "金额不能超过两位小数");
    Amount = amount;
    CurrencyCode = currencyCode;
}
```

领域服务

因为我们需要在相加或者相减的时候判断是否为同一个货币，所以这里我们定义一个有业务含义的Exception，在Money.cs里面添加一个Exception。

```
32 1 个引用 | 0 项更改 | 0 名作者, 0 项更改  
33 public class CurrencyMismatchException : Exception  
34 {  
35     0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
36     public CurrencyMismatchException(string? message) : base(message)  
37     {  
38     }  
39 }
```

领域服务

最后，对Money.cs里面的一些操作方法进行修改。

1 个引用 | HCJ, 不到 5 分钟之前 | 1 名作者, 1 项更改

```
public Money Add(Money m)
{
    if (CurrencyCode != m.CurrencyCode)
        throw new CurrencyMismatchException("不能对两个不同币种的面值进行相加");
    return new Money(Amount + m.Amount, CurrencyCode);
}
```

1 个引用 | HCJ, 不到 5 分钟之前 | 1 名作者, 1 项更改

```
public Money Subtract(Money m)
{
    if (CurrencyCode != m.CurrencyCode)
        throw new CurrencyMismatchException("不能对两个不同币种的面值进行相减");
    return new Money(Amount - m.Amount, CurrencyCode);
}
```

领域服务

目前我们已经能兼容多种币种，但是我们还有几个细节没处理：不能验证哪些货币是有效的，哪些货币面值有多少位小数。接下来我们用领域服务来处理这些细节。在Domain里新建一个CurrencyDetail类。

```
namespace RJRentalOfHousing.Domain
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public interface ICurrencyLookup
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        CurrencyDetails FindCurrency(string currencyCode);
    }

    3 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public record CurrencyDetails
    {
        0 个引用 | HCJ, 1 天前 | 1 名作者, 1 项更改
        public string CurrencyCode { get; set; }
        1 个引用 | HCJ, 1 天前 | 1 名作者, 1 项更改
        public bool InUse { get; set; }
        0 个引用 | HCJ, 1 天前 | 1 名作者, 1 项更改
        public int DecimalPlaces { get; set; }
        public static CurrencyDetails None = new CurrencyDetails { InUse = false };
    }
}
```

在这里我们定义了三个字段，币种（CurrencyCode），是否启用（InUse），小数位（DecimalPlaces），并且定义了一个接口。我们接下来要用这些来实现一个简单的领域服务（Domain Service）

领域服务

继续修改（折腾）我们的Money类。首先把字符串类型的CurrencyCode修改为Currency，然后修改构造函数和工厂方法。

6 个引用 | HCJ, 3 天前 | 1 名作者, 1 项更改

```
public decimal Amount { get; internal set; }
```

8 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public CurrencyDetails Currency { get; internal set; }
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public static Money FromString(string amount, string currency, ICurrencyLookup currencyLookup) => new Money(decimal.Parse(amount), currency, currencyLookup);
```

3 个引用 | 1/2 通过 | 0 项更改 | 0 名作者, 0 项更改

```
public static Money FromDecimal(decimal amonut, string currency, ICurrencyLookup currencyLookup) => new Money(amonut, currency, currencyLookup);
```

3 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
protected Money(decimal amount, string currencyCode, ICurrencyLookup currencyLookup)
{
    if (string.IsNullOrEmpty(currencyCode))
        throw new ArgumentNullException(nameof(currencyCode), "币种编码不能为空");
    var currency = currencyLookup.FindCurrency(currencyCode);
    if (!currency.InUse)
        throw new ArgumentException($"非法的币种 {currency.CurrencyCode}");
    if (decimal.Round(amount, currency.DecimalPlaces) != amount)
        throw new ArgumentOutOfRangeException(nameof(amount), $"币种 {currency.CurrencyCode} 最多只能有 {currency.DecimalPlaces} 位小数");
    Amount = amount;
    Currency = currency;
}
```

领域服务

接下来修改两个操作方法，Add和Subtract。此外我们还需要一个私有的构造方法，里面不用验证，直接赋值。

```
protected Money(decimal amount, CurrencyDetails currency)
```

```
{  
    Amount = amount;  
    Currency = currency;  
}
```

1 个引用 | HCJ, 2 天前 | 1 名作者, 1 项更改

```
public Money Add(Money m)
```

```
{  
    if (Currency != m.Currency)  
        throw new CurrencyMismatchException("不能对两个不同币种的面值进行相加");  
    return new Money(Amount + m.Amount, Currency);  
}
```

1 个引用 | HCJ, 2 天前 | 1 名作者, 1 项更改

```
public Money Subtract(Money m)
```

```
{  
    if (Currency != m.Currency)  
        throw new CurrencyMismatchException("不能对两个不同币种的面值进行相减");  
    return new Money(Amount - m.Amount, Currency);  
}
```

接下来我们会对这个花了洪荒之力写的Money对象进行测试

领域服务（单元测试）

首先我们需要新建一个用于测试的领域服务，在Tests新建一个FakeCurrencyLookup类。

```
using RJRentalOfHousing.Domain;

namespace RJRentalOfHousing.Tests
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class FakeCurrencyLookup : ICurrencyLookup
    {
        private readonly IEnumerable<CurrencyDetails> _currencies =
            new[]
            {
                new CurrencyDetails() {CurrencyCode="CNY",DecimalPlaces=2,InUse=true},
                new CurrencyDetails() {CurrencyCode="JPY",DecimalPlaces=0,InUse=true},
                new CurrencyDetails() {CurrencyCode="USD",DecimalPlaces=2,InUse=true},
                new CurrencyDetails() {CurrencyCode="EUR",DecimalPlaces=2,InUse=true},
                new CurrencyDetails() {CurrencyCode="DEM",DecimalPlaces=2,InUse=false},
            };

        2 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public CurrencyDetails FindCurrency(string currencyCode)
        {
            var currency = _currencies.FirstOrDefault(x=>x.CurrencyCode == currencyCode);
            return currency ?? CurrencyDetails.None;
        }
    }
}
```

领域服务（单元测试）

在Money_Spec里面新增一些测试代码，在此之前可以把这个类里之前的代码删除。

```
public class Money_Spec
{
    private static readonly ICurrencyLookup CurrencyLookup = new FakeCurrencyLookup();

    [Fact]
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public void Two_of_same_amount_should_be_equal()
    {
        var m1 = Money.FromDecimal(5, "CNY", CurrencyLookup);
        var m2 = Money.FromDecimal(5, "CNY", CurrencyLookup);
        Assert.Equal(m1, m2);
    }

    [Fact]
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public void Two_of_same_amount_but_differentCurrencier_should_not_be_equal()
    {
        var m1 = Money.FromDecimal(5, "CNY", CurrencyLookup);
        var m2 = Money.FromDecimal(5, "USD", CurrencyLookup);
        Assert.NotEqual(m1, m2);
    }
}
```


领域服务（单元测试）

[Fact]

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void FromString_and_FromDecimal_should_be_equal()
{
    var m1 = Money.FromDecimal(5, "CNY", CurrencyLookup);
    var m2 = Money.FromString("5.00", "CNY", CurrencyLookup);
    Assert.Equal(m1, m2);
}
```

[Fact]

0 个引用 | HCJ, 2 天前 | 1 名作者, 1 项更改

```
public void Sum_of_money_gives_full_amount()
{
    var coin1 = Money.FromDecimal(1, "CNY", CurrencyLookup);
    var coin2 = Money.FromDecimal(2, "CNY", CurrencyLookup);
    var coin3 = Money.FromDecimal(3, "CNY", CurrencyLookup);
    var banknote = Money.FromDecimal(6, "CNY", CurrencyLookup);
    Assert.Equal(banknote, coin1 + coin2 + coin3);
}
```

[Fact]

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void Unused_currency_should_not_be_allowed()
{
    Assert.Throws<ArgumentException>(() => Money.FromDecimal(100, "DEM", CurrencyLookup));
}
```

[Fact]

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void Unknown_currency_should_not_be_allowed()
{
    Assert.Throws<ArgumentException>(() => Money.FromDecimal(100, "ABC", CurrencyLookup));
}
```

领域服务（单元测试）

[Fact]

0 个引用 10 项更改 10 名作者, 0 项更改

```
public void Throw_when_too_many_decimal_places()
{
    Assert.Throws<ArgumentOutOfRangeException>(() => Money.FromDecimal(100.111m, "CNY", CurrencyLookup));
}
```

[Fact]

0 个引用 10 项更改 10 名作者, 0 项更改

```
public void Throws_on_adding_different_currencies()
{
    Money m1 = Money.FromDecimal(5, "CNY", CurrencyLookup);
    Money m2 = Money.FromDecimal(5, "USD", CurrencyLookup);
    Assert.Throws<CurrencyMismatchException>(() => m1 + m2);
}
```

[Fact]

0 个引用 10 项更改 10 名作者, 0 项更改

```
public void Throws_on_subtracting_different_currencies()
{
    Money m1 = Money.FromDecimal(5, "CNY", CurrencyLookup);
    Money m2 = Money.FromDecimal(5, "USD", CurrencyLookup);
    Assert.Throws<CurrencyMismatchException>(() => m1 + m2);
}
```

领域服务（单元测试）

在启动测试之前需要对Price类进行修改，因为我们前面改动了Money类的构造函数。

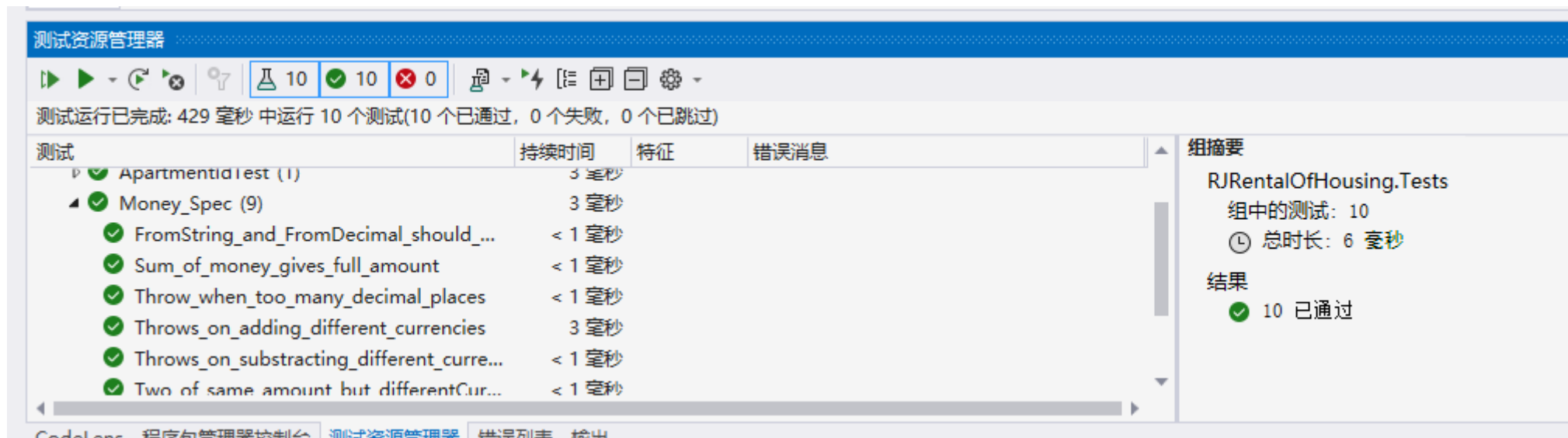
```
6 个引用 | 0 项更改 | 0 名作者, 0 项更改
public record Price : Money
{
    1 个引用 | 0 项更改 | 0 名作者, 0 项更改
    private Price(decimal amount, string currencyCode, ICurrencyLookup currencyLookup) : base(amount, currencyCode, currencyLookup)
    {
        if (amount < 0)
            throw new ArgumentException("价格不能为负数", nameof(amount));
    }

    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    internal Price(decimal amount, string currencyCode) : base(amount, new CurrencyDetails { CurrencyCode=currencyCode })
    {
    }

    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public static Price FromDecimal(decimal amount, string currency, ICurrencyLookup currencyLookup)
        => new Price(amount, currency, currencyLookup);
}
```

领域服务（单元测试）

在测试资源管理器里运行全部测试



测试全部通过

值对象（我保证这是最后一个）

现在我们的实体看上去很完美，但是仍有一些业务被遗漏，比如地址（Address）不能为空。我们打算将这个限制外包到领域之外的地方，因此我们将新建一个值对象。在Domain中新建一个Address类。

```
namespace RJRentalOfHousing.Domain
{
    1 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public record Address
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public string Value { get; internal set; }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Address(string value)
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException("地址不能为空");
            Value = value;
        }
    }
}
```

```
1 个引用 | HCJ, 3 天前 | 1 名作者, 1 项更改
public ApartmentId Id { get; internal set; }

1 个引用 | HCJ, 3 天前 | 1 名作者, 1 项更改
public Area Areas { get; internal set; }

0 个引用 | HCJ, 3 天前 | 1 名作者, 1 项更改
public Address Address { get; internal set; }

0 个引用 | HCJ, 2 天前 | 1 名作者, 2 项更改
public Price Rent { get; internal set; }
```

新增行为

到目前为止我们都是在对模型添加业务限制，实体现在并没有真正的动起来。现在我们来添加一些行为，为实体注入生命。在Apartment类中添加一些代码。

```
public void SetArea(Area area) => Area = area;
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void SetAddress(Address address) => Address = address;
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void SetRent(Price rent) => Rent = rent;
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void SetDeposit(Price deposit) => Deposit = deposit;
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void SetOwner(UserId owner) => Owner = owner;
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void SetRemark(string remark) => Remark = remark;
```

这些代码看上去很简单，甚至能用Public的Set方法代替。但是这是我们后面实现领域事件（Domain Event）的基础。

实体状态

前面说到，实体是有状态的，我们现在考虑一下我们的房子应该有什么状态？现在假设我们有已发布、待审核、招租中、已出租四个状态。修改Apartment中的代码，添加一个枚举，然后新增一个状态字段。

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
public void SetOwner(UserId owner) => Owner = owner;  
  
0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
public void SetRemark(string remark) => Remark = remark;  
  
0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
public enum ApartmentState  
{  
    PendingReview,  
    Active,  
    Renting,  
    Rented  
}
```

```
1 个引用 | HCJ, 3 天前 | 1 名作者, 1 项更改  
public string Remark { get; internal set; }  
  
0 个引用 | HCJ, 3 天前 | 1 名作者, 1 项更改  
public UserId ApprovedBy { get; internal set; }  
  
0 个引用 | 0 项更改 | 0 名作者, 0 项更改  
public ApartmentState State { get; internal set; }  
  
0 个引用 | HCJ, 不到 5 分钟之前 | 1 名作者, 1 项更改  
public void SetArea(Area area) => Areas = area;  
  
0 个引用 | HCJ, 不到 5 分钟之前 | 1 名作者, 1 项更改  
public void SetAddress(Address address) => Address = address;  
  
0 个引用 | HCJ, 不到 5 分钟之前 | 1 名作者, 1 项更改
```

实体状态

接下来我们要改变实体的状态，在此之前我们要新增一个Exception的类型，以表示这是业务的Exception。在Domain中新建一个类，命名为InvalidEntityStateException。

```
RentalOfHousing.Domain RJRentalOfHousing.Domain.Inva
1 namespace RJRentalOfHousing.Domain
2 {
3     0 个引用 | 0 项更改 | 0 名作者, 0 项更改
4     public class InvalidEntityStateException : Exception
5     {
6         0 个引用 | 0 项更改 | 0 名作者, 0 项更改
7         public InvalidEntityStateException(object entity, string? message)
8             : base($"实体 {entity.GetType().Name} 状态改变时发生错误, {message}")
9         {
10         }
11     }
```


实体状态

现在在Apartment中新建一个方法，来改变实体状态。然后在构造函数中给一个初始状态。

0 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public void RequestToPublish()
{
    if (Areas == null)
        throw new InvalidEntityStateException(this, "面积不能为空");
    if (Address == null)
        throw new InvalidEntityStateException(this, "地址不能为空");
    if (Rent.Amount > 0)
        throw new InvalidEntityStateException(this, "租金不能小于0");
    if (Deposit.Amount > 0)
        throw new InvalidEntityStateException(this, "押金不能小于0");
    State = ApartmentState.Renting;
}
```

```
public Apartment(ApartmentId id, UserId ownerId)
{
    Id = id;
    Owner = ownerId;
    State = ApartmentState.Created;
}
```

实体状态

现在RequestToPublish方法里面不仅有改变实体状态的能力，也有检查实体状态的能力。看起来很完美，可是如果在提交审核之后再吧面积或者租金、押金改为空就会破坏实体的有效性。简单粗暴的做法是把这些检查状态的代码到处复制粘贴几遍，这样就破坏了DRY原则，因此我们不会这么做。我们在Apartment里新建一个方法，在那里检查实体的有效性。

实体状态

在Apartment中新建一个方法。

```
protected void EnsureValidState()
{
    var valid =
        Id != null &&
        Owner != null &&
        (
            State switch
            {
                ApartmentState.PendingReview =>
                    Areas != null &&
                    Address != null &&
                    Rent.Amount > 0 &&
                    Deposit.Amount > 0,
                ApartmentState.Renting =>
                    Areas != null &&
                    Address != null &&
                    Rent.Amount > 0 &&
                    Deposit.Amount > 0 &&
                    ApprovedBy != null,
                _ => true
            }
        );
    if (!valid)
        throw new InvalidEntityStateException(this, $"实体提交状态 {State} 检查失败");
}
```

又见到了一个平时比较少见的东西，模式匹配。这个语法后面会经常用到，但是在这里还是不展开来讲。有兴趣了解的可以去看微软文档[模式匹配概述 — C# 指南 | Microsoft Learn](https://learn.microsoft.com/zh-cn/csharp/whats-new/csharp-9.0#pattern-matching)

实体状态

接下来，在改变实体状态的地方都加上这个方法的调用。

```
public void SetAddress(Address address)
{
    Address = address;
    EnsureValidState();
}
```

0 个引用 | HCJ, 40 分钟前 | 1 名作者, 1 项更改

```
public void SetRent(Price rent)
{
    Rent = rent;
    EnsureValidState();
}
```

0 个引用 | HCJ, 40 分钟前 | 1 名作者, 1 项更改

```
public void SetDeposit(Price deposit)
{
    Deposit = deposit;
    EnsureValidState();
}
```

0 个引用 | HCJ, 41 分钟前 | 1 名作者, 1 项更改

```
public void SetOwner(UserId owner)
{
    Owner = owner;
    EnsureValidState();
}
```

0 个引用 | HCJ, 41 分钟前 | 1 名作者, 1 项更改

```
public void SetRemark(string remark)
{
    Remark = remark;
    EnsureValidState();
}
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void RequestToPublish()
{
    State = ApartmentState.Renting;
    EnsureValidState();
}
```

实体状态（单元测试）