# MVC基本工具

一般常用的工具有DI容器（依懒性注入：DI，Dependency Injection）、单元测试框架和模仿工具

## DI容器

**依懒性注入——DI（Dependency Injection）**：

也成为**控制反转**（IoC，Inversion of Control），它是一种实现松散耦合的设计模式，可以解决一个类对其他类的依赖性问题，从而解决类之间的耦合问题。一般有两种方式，一是构造器注入（即通过构造函数注入依赖性）；另一个是设置器注入（即通过public属性注入）。

但是，依赖性注入不能解决这样的问题：我们该如何实例化接口的具体实现，而不需要在程序的某个其他地方创建依赖性？（即我们总是需要在其他地方去实例化该接口的具体实现，然后将其传递给该接口。）所以，我们就需要一个依赖性注入容器，一般常用的DI容器是Ninject，另外，微软也有自己的DI容器——Unity。

我们这里将以Ninject为例进行介绍。

在项目中添加Ninject可以有两种方式：一是通过“工具”—>“库包管理器”—>“管理解决方案的NuGet包”，打开NuGet包对话框在线搜索，然后安装。另一种方式是从官网（www.ninject.org）直接下载最新版，然后手动安装。由于手动安装会让我们失去更新和包依赖性管理的好处，所以，通常会选择在线自动安装的方式。

1. **使用Ninject的基本步骤：**

* 创建一个Ninject内核实例
* 在程序中建立接口和想要使用的实现类之间的关系
* 通过Get方法使用Ninject

1. **建立MVC依赖性注入**

**创建依赖解析器：**

MVC 框架需要使用依赖性解析器来创建类的实例，因此，需要创建一个自定义的解析器，这里，以名为 NinjectDependencyResolver 的解析器举例，解析器需要继承 IDependencyResolver 接口，代码如下：

/// <summary>

/// 依赖性解析器

/// </summary>

public class NinjectDependencyResolver : IDependencyResolver

{

private IKernel \_kernel;

public NinjectDependencyResolver()

{

\_kernel = new StandardKernel();

AddBindings();

}

#region IDependencyResolver 的实现

public object GetService(Type serviceType)

{

return \_kernel.TryGet(serviceType);

}

public IEnumerable<object> GetServices(Type serviceType)

{

return \_kernel.GetAll(serviceType);

}

#endregion End IDependencyResolver 的实现

private void AddBindings()

{

\_kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();

}

**注册依赖解析器：**

在具体的使用时，需要告知 MVC 框架要使用的自定义的依赖解析器，这就需要修改 Global.asax.cs文件来完成了，具体修改件下面粗体字部分：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http;

using System.Web.Mvc;

using System.Web.Routing;

**using EssentialTools.Infrastructure;**

namespace EssentialTools

{

// 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，

// 请访问 http://go.microsoft.com/?LinkId=9394801

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

**//注册依赖解析器**

**//下面语句将 DI 放在了该程序的内核中，它实现了让 Ninject 创建 MVC 框架所需的任何对象的实例**

**DependencyResolver.SetResolver(new NinjectDependencyResolver());**

WebApiConfig.Register(GlobalConfiguration.Configuration);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

}

}

}

最后，重置 Home 控制器，以使其能够利用上面建立的解析器。下面是添加的代码：

// 通过创建全局私有的接口对象，并通过添加的控制器构造函数的参数

// （该接口对象的接口类型参数）注入具体的实现来进行赋值的方式来

// 进行对 HomeController 对该接口的具体实现的解耦

// ----------------------------------基本工作流程-----------------------------------------------

// 1、浏览器项 MVC 发送一个请求 Home 的 URL，MVC 框架猜出该请求意指 Home 控制器，于是

// 会创建 HomeController 类实例

// 2、MVC 框架在创建 HomeController 类实例的过程中会发现其构造器有一个对 IValueCalculator

// 接口的依赖项，便会要求依赖性解析器对此依赖项进行解析，将该接口指定为依赖性解析器中的

// GetService 方法所使用的类型参数

// 3、依赖性解析器将传递来的类型参数交给 TryGet 方法，要求 Ninject 创建一个新的 IValueCalculator

// 接口类实例

// 4、Ninject 会检测到该接口与其实现类 LinqValueCalculator 具有绑定关系，于是将为该接口创建

// 一个 LinqValueCalculator 类实例，并返回给依赖性解析器

// 5、依赖性解析器将 Ninject 返回的 LinqValueCalculator 类作为 IValueCalculator 接口实现类

// 实例回递给 MVC 框架

// 6、MVC 框架礼仪依赖性解析器返回的接口类实例创建 HomeController 控制器实例，并使用该控制器

// 实例队请求进行服务

// --------------------------------------------------------------------------------------------

private IValueCalculator \_calc;

public HomeController(IValueCalculator calcParam)

{

\_calc = calcParam;

}

**依赖性链：**

当Ninject创建一个类型时，它会检测该类型与其他类型之间的耦合。如果有额外的依赖性，则会自动解析这些依赖性，并创建所需要的所有类的实例。可以通过下面的代码示例进行了解：

如有一个类文件：Discount.cs，其中定义了一个接口和类型，如下所示：

namespace EssentialTools.Models

{

public interface IDiscountHelper

{

decimal ApplyDiscount(decimal totalParam);

}

public class DefaultDiscountHelper : IDiscountHelper

{

public decimal ApplyDiscount(decimal totalParam)

{

return (totalParam - (10m / 100 \* totalParam));

}

}

}

在类 LinqValueCalculator 中进行了引用，代码如下：

public class LinqValueCalculator : IValueCalculator

{

**private IDiscountHelper \_discounter;**

**public LinqValueCalculator(IDiscountHelper discountParam)**

**{**

**\_discounter = discountParam;**

**}**

/// <summary>

/// 计算并返回 Product 集合的所有 Product 对象的价格（Price）总和

/// </summary>

/// <param name="products"></param>

/// <returns></returns>

public decimal ValueProducts(IEnumerable<Product> products)

{

//使用 IDiscountHelper 接口定义的方法计算

return **\_discounter.ApplyDiscount**(products.Sum(p => p.Price));

}

}

最后在依赖性解析器中进行绑定（粗体部分）：

/// <summary>

/// 依赖解析器

/// </summary>

public class NinjectDependencyResolver : IDependencyResolver

{

private IKernel \_kernel;

public NinjectDependencyResolver()

{

\_kernel = new StandardKernel();

AddBindings();

}

#region IDependencyResolver 的实现

public object GetService(Type serviceType)

{

return \_kernel.TryGet(serviceType);

}

public IEnumerable<object> GetServices(Type serviceType)

{

return \_kernel.GetAll(serviceType);

}

#endregion End IDependencyResolver 的实现

private void AddBindings()

{

\_kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();

**\_kernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>();**

}

}

**Ninject对上述这种方式的解析实现机制大致如下：**

Ninject首先会检测到接口 IValueCalculator ,并创建其实现类 LinqValueCalculator，此时，将发现 LinqValueCalculator 对接口 IDiscountHelper具有引用关系，因此，Ninject 将会查看其绑定，并创建一个 DefaultDiscountHelper 对象，然后将其传递给 LinqValueCalculator 对象的构造函数，进一步将 LinqValueCalculator 对象传递给 HomeController 的构造函数，最终将得到预期结果。Ninject 会使用这种方式检查它将要实例化的每一个依赖类，不论这种依赖关系有多复杂。

1. **指定属性与构造器参数值**

在为接口绑定其具体实现时，可以提供要有运用的属性上的一些属性细节对 Ninject 创建的类进行配置，或者通过构造函数的参数设置相关信息。如下分别举例如何使用属性和构造函数的参数进行相关设置：

* **属性设置方式：**

public class DefaultDiscountHelper : IDiscountHelper

{

/// <summary>

/// 用于计算折扣量

/// </summary>

**public decimal DiscountSize { get; set; }**

public decimal ApplyDiscount(decimal totalParam)

{

return (totalParam - (**DiscountSize** / 100 \* totalParam));

}

}

依赖解析器 NinjectDependencyResolver中通过 WithPropertyValue 方法在绑定后进行设置：

…

private void AddBindings()

{

…

//使用 WithPropertyValue 方法设置 DefaultDiscountHelper 类中的 DiscountSize 属性的值

//通过这种方式可以不修改绑定或通过 Get 方法获取具体实现类的实例的方式

\_kernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>().**WithPropertyValue("DiscountSize", 50M)**;

}

…

* **构造函数参数方式：**

public class DefaultDiscountHelper : IDiscountHelper

{

/// <summary>

/// 用于计算折扣量

/// </summary>

**public decimal \_discountSize;**

**public DefaultDiscountHelper(decimal discountParam)**

**{**

**\_discountSize = discountParam;**

**}**

public decimal ApplyDiscount(decimal totalParam)

{

return (totalParam - (**\_discountSize** / 100 \* totalParam));

}

}

将使用 WithConstructorArgument 方法绑定：

…

private void AddBindings()

{

…

//使用 WithPropertyValue 方法设置 DefaultDiscountHelper 类中的 DiscountSize 属性的值

//通过这种方式可以不修改绑定或通过 Get 方法获取具体实现类的实例的方式

\_kernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>().**WithConstructorArgument("discountParam ", 50M)**;

}

…

上述两种方式均可以使用链接调用的方式涵盖要设置的所有信息，如：**WithPropertyValue("A", A). WithPropertyValue("B", B)…**的形式，并以此类推

1. **使用条件绑定**

Ninject使用多个条件绑定的方法可以知道用哪一个类对某一特定的请求进行响应。

可以通过添加一个名为 FlexibleDiscountHelper 的类做一个示例进行阐述：

public class FlexibleDiscountHelper : IDiscountHelper

{

public decimal ApplyDiscount(decimal totalParam)

{

decimal discount = totalParam > 100 ? 70 : 50;

return (totalParam - (discount / 100m \* totalParam));

}

}

然后，通过下面的方式通知 Ninject 何时使用 FlexibleDiscountHelper，何时使用 DefaultDiscountHelper：

…

private void AddBindings()

{

…

\_kernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>().WithPropertyValue("DiscountSize", 50M);

//通知 Ninject 何时使用 FlexibleDiscountHelper，何时使用 //DefaultDiscountHelper

**\_kernel.Bind<IDiscountHelper>().To<FlexibleDiscountHelper>().WhenInjectedInto<LinqValueCalculator>();**

}

…

上面示例保留了对 IDiscountHelper 接口的原有默认的绑定，这时 Ninject会自动匹配最佳实现方式。这样也有助于对同一个类或接口采用一个默认绑定，以便在条件得不到满足时自动回滚。

下表列出了一些常用的条件绑定方法：

|  |  |
| --- | --- |
| **方法** | **效果** |
| When(predicate) | 当“predicate（谓词）”（一个lambda表达式），结果为true时，实施绑定 |
| WhenClassHas<T>() | 当被注入的类以注解属性（特性，以方括号[]形式设置）进行注释，且其类型为T时，实时绑定 |
| WhenInjectedInto<T>() | 当要被注入的类是类型T时，实时绑定 |

## 单元测试

目前在开发工作中，尤其是在Web开发中，有两种自动化测试方式。一是单元测试（Unit Testing，这是一种以与应用程序其他部分相隔离的方式，指定并检验单个类（或其他小型代码单元）行为的方法），一是集成测试（Integration Testing，这是指定并检验多个组件，乃至包括整个Web应用程序，协同工作行为的方法）。

这里主要了解单元测试。在单元测试中，常常会借助测试驱动开发（TDD）的方式进行。不明而寓，这是要先写测试，通过测试用例驱动代码的开发，其主要实现方式如下：

* 确定需要添加的新特性或方法
* 编写测试，用以检验新特性的行为
* 运行测试并得到一个红色信号——异常通知
* 编写实现新特性的代码
* 再次运行测试并修改代码，直到最终得到一个绿色信号——测试通过信息
* 必要时重构该代码，例如重组语句、重命名变量等
* 运行该测试，以确认修改已经不会改变这个新增特性的行为

注意，在测试中一定要保证测试的全面性。

在编写单元测试的过程中，一般分为三个步骤：准备、动作、断言，在断言部分，常用的一个类是Assert。在单元测试时不用担心单元测试之间的相互影响，因为，他们每一个都是相互独立的。

## Moq

Moq是一种模仿对象的方式，它相比Fakes更简单易用，且免费（Fakes是在收费版的Visual Studio中才被提供的一项功能）。在项目中经常在测试的时候遇到所测试的类或方法对其他类有依赖性，此时就很有可能自动测试到依赖的类，而如果只希望测试到目标对象或方法，我们就需要使用模仿对象的方式了，通过这种方式，能够实现一种以特殊而受控的方式，来模拟项目中实际对象的功能。模仿对象能够缩小测试的侧重点，以达到只检查所感兴趣的功能的目的。

例如，我们现在有下面这样的一个单元测试类，它实现了对LinqValueCalculator类的测试，但这里有个问题就是LinqValueCalculator依赖于IDiscountHelper接口的实现（如下面示例代码中的MinimumDiscountHelper类）。这变会有如下两个不同的问题：

* 单元测试变得复杂和脆弱。在创建单元测试的时候就需要考虑到IDiscountHelper实现中的折扣逻辑，而脆弱性表现在：如果该实现中的折扣逻辑发生改变，测试便会失败。
* 该测试范围被延展到了MinimumDiscountHelper类，当测试失败时就很难判断出是LinqValueCalculator还是MinimumDiscountHelper出的问题。

如下代码所示：

[TestClass]

public class UnitTest2

{

private Product[] \_products = {

new Product{Name="Kayak",Category="Watersports",Price=275M},

new Product{Name="Lifejacket",Category="Watersports",Price =48.95M},

new Product{Name="Soccer ball",Category="Soccer",Price=19.50M},

new Product{Name="Corner flag",Category="Soccer",Price=34.95M}

};

[TestMethod]

public void Sum\_Products\_Correctly()

{

//准备

var dicounter = new **MinimumDiscountHelper**();

var target = new **LinqValueCalculator**(dicounter);

var goalTotal = \_products.Sum(e => e.Price);

//动作

var result = target.ValueProducts(\_products);

//断言

Assert.AreEqual(goalTotal, result);

}

}

* **在项目中使用Moq**

对于Moq的添加引用的方法可参考DI容器部分关于Ninject的添加介绍。这里需要注意的是Moq是要添加到单元测试的项目中，而不是MVC项目中。

在用 Moq 创建模仿都系的整个过程包含了以下几步：

1. 用 Mock 创建模仿对象
2. 用 Setup 方法建立对象的行为
3. 用 It 类设置行为的参数
4. 用 Return 方法指定行为的返回类型
5. 用 lambda 表达式在 Return 方法中建立具体行为
6. 使用模仿对象

具体代码示例如下：

// 1、创建模仿对象

Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();

// 2、选择方法:Setup，并通过 It 类设置需要的参数信息

// 3、定义结果：Returns，并用 lambda 表达式在 Return 方法中建立具体行为

mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);

// 4、使用模仿对象：mock.Object，通过获取 Object 属性的值来获取具体实现，如这里的实现是 IDiscountHelper 接口的实现

var target = new LinqValueCalculator(mock.Object);

使用 Moq 的主要有这样的优点：*使用 Moq 会使单元测试只检查 LinqValueCalculator 对象的行为，并不依赖任何 Models 文件夹中 IDiscountHelper 的真实实现。*

* **复杂模仿对象的创建**

对于复杂对象的创建如下示例所示：

…

private Product[] createProduct(decimal value)

{

return new[] { new Product { Price = value } };

}

/// <summary>

/// 创建复杂的模仿对象——模拟 MinimumDiscountHelper 类的行为

/// </summary>

[TestMethod]

[ExpectedException(typeof(System.ArgumentOutOfRangeException))]

public void Pass\_Through\_Variable\_Discounts()

{

// 准备

Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();

// 全匹配

mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);

// 模仿特定值，并抛出异常

mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v == 0))).Throws<System.ArgumentOutOfRangeException>();

// 模仿特定值

mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v > 100))).Returns<decimal>(total => (total \* 0.9M));

// 模仿值的范围

mock.Setup(m => m.ApplyDiscount(It.IsInRange<decimal>(10, 100, Range.Inclusive))).Returns<decimal>(total => total - 5);

// 上面的写法等同于下面这种方式（这种方式通常更灵活）：

//mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v >= 10 && v <= 100))).Returns<decimal>(total => total - 5);

// \*\*\*\*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\*\*\*\*

// 上述的模仿顺序是不能颠倒的，原因如下：

// Moq 会以相反的顺序评估所给定的行为，因此会考虑调用最后一个 Setup 方法。也就是说我们在使用时必须遵循先从最一般开始向最特殊的情绪顺序就能行。

// 即：Moq 会先去评定最后的，然后上逐步执行。按照此例，如果最普通的情况先执行，就意味着所有符合 decimal 类型参数的情形均可满足条件，那后续的如

// 大于 100 时将不会返回预期结果，而是错误的模仿结果了。

// \*\*\*\*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\*\*\* 注意 \*\*\*\*\*\*\*\*\*\*

var target = new LinqValueCalculator(mock.Object);

// 动作

decimal FiveDollarDiscount = target.ValueProducts(createProduct(5));

decimal TenDollarDiscount = target.ValueProducts(createProduct(10));

decimal FiftyDollarDiscount = target.ValueProducts(createProduct(50));

decimal HundredDollarDiscount = target.ValueProducts(createProduct(100));

decimal FiveHundredDollarDiscount = target.ValueProducts(createProduct(5000));

// 断言

Assert.AreEqual(5, FiveDollarDiscount, "$5 Fail");

Assert.AreEqual(5, TenDollarDiscount, "$10 Fail");

Assert.AreEqual(45, FiftyDollarDiscount, "$50 Fail");

Assert.AreEqual(95, HundredDollarDiscount, "$100 Fail");

Assert.AreEqual(450, FiveHundredDollarDiscount, "$500 Fail");

target.ValueProducts(createProduct(0));

}

…

# 附表1：Assert 类（单元测试的断言类）

|  |  |
| --- | --- |
| **方法** | **描述** |
| AreEqual<T>(T,T)  AreEqual<T>(T,T,string) | 断言两个类型T的对象有相同的值 |
| AreNotEqual<T>(T,T)  AreNotEqual<T>(T,T,string) | 断言两个类型T的对象的值不相等 |
| AreSame<T>(T,T)  AreSame<T>(T,T, string) | 断言两个变量指向相同的对象 |
| AreNotSame<T>(T,T)  AreNotSame<T>(T,T, string) | 断言两个变量指向不同的对象 |
| Fail()  Fail(string) | 失败断言——无条件检查 |
| Inconclusive()  Inconclusive(string) | 指示最终不能建立单元测试的结果 |
| IsTrue(bool)  IsTrue(bool,string) | 断言一个布尔值为true——最常用于评估一个返回布尔结果的表达式 |
| IsFalse(bool)  IsFalse(bool,string) | 断言一个布尔值为false |
| IsNull(bool)  IsNull(bool,string) | 断言一个变量未被分配一个对象引用 |
| IsNotNull(bool)  IsNotNull(bool,string) | 断言一个变量被分配了一个对象引用 |
| IsInstanceOfType(object,Type)  IsInstance OfType(object,Type,string) | 断言一个对象是指定的类型，或是派生于指定的类型 |
| IsNotInstanceOfType(object,Type)  IsNotInstanceOfType(object,Type,string) | 断言一个对象不是指定的类型 |

# 附表2：It 类（Moq中相关技术）

|  |  |
| --- | --- |
| **方法** | **描述** |
| Is<T>(predicate) | 指定类型T的值，该值使“predicate”返回true |
| IsAny<T>() | 指定类型T的值为任意值 |
| IsInRange<T>(min,max,kind) | 如果参数介于定义值之间，而且是T类型的，则匹配。最后一个参数是该范围的一个枚举值，可以是Inclusive（包括）或Exclusive（排除） |
| IsRegex(expr) | 如果参数是一个字符串，而且符合指定的正则表达式，则匹配 |