

NopCommerce架构分析

zaxx 94027486 2015年8月27日

目录

[第一章 NOPCOMMERCE架构分析 1](#_Toc439083943)

[1 开篇 1](#_Toc439083944)

[2 数据库初试化及数据操作 2](#_Toc439083945)

[3 插件机制 2](#_Toc439083946)

[4 Model绑定Action参数 3](#_Toc439083947)

[5 自定义RazorViewEngine 3](#_Toc439083948)

[5.1 自定义RazorViewEngine 3](#_Toc439083949)

[5.2 自定义类WebViewPage<TModel> 4](#_Toc439083950)

[6 皮肤管理器 4](#_Toc439083951)

[7 多语言 5](#_Toc439083952)

[第二章 数据迁移 7](#_Toc439083953)

[1 开启迁移功能 7](#_Toc439083954)

[2 初次添加迁移文件 7](#_Toc439083955)

[3 初次更新数据库 8](#_Toc439083956)

[4 新增实体及实体映射后添加迁移文件 9](#_Toc439083957)

[4.1 新增实体Link 9](#_Toc439083958)

[4.2 新增实体映射LinkMap 10](#_Toc439083959)

[4.3 添加迁移文件 11](#_Toc439083960)

[4.4 更新数据库 12](#_Toc439083961)

[5 添加新增实体的示例数据 12](#_Toc439083962)

[第三章 映射 14](#_Toc439083963)

[1.1 14](#_Toc439083964)

[第四章 多语言 15](#_Toc439083965)

[1.1 获取实体数据的多语言 15](#_Toc439083966)

[1.2 前端页面获取资源的多语言 15](#_Toc439083967)

[1.3 后台文件获取资源的多语言 16](#_Toc439083968)

[1.4 将多语言资源添加到数据库中 16](#_Toc439083969)

[第五章 添加新功能 18](#_Toc439083970)

[1 添加实体实现新功能的一般步骤 18](#_Toc439083971)

[2 创建实体服务接口与实体服务类 18](#_Toc439083972)

[3 注册服务接口及服务类 25](#_Toc439083973)

[4 新增前端演示模型 25](#_Toc439083974)

[5 新增验证文件 27](#_Toc439083975)

[6 在资源文件中添加所需资源 29](#_Toc439083976)

[7 添加Web缓存处理 30](#_Toc439083977)

[7.1 指定缓存key格式 31](#_Toc439083978)

[7.2 通过缓存key获取缓存数据 32](#_Toc439083979)

[8 添加控制器及视图 32](#_Toc439083980)

[8.1 添加控制器 32](#_Toc439083981)

[8.2 添加视图 39](#_Toc439083982)

[9 注册路由 40](#_Toc439083983)

[9.1 全局路由 40](#_Toc439083984)

[9.2 插件中定义路由 40](#_Toc439083985)

[9.3 自定义路由处理 41](#_Toc439083986)

[第六章 Web层中的布局 44](#_Toc439083987)

[1 布局页 44](#_Toc439083988)

[2 菜单 45](#_Toc439083989)

[2.1 前台页面主菜单 45](#_Toc439083990)

[2.2 前台页面左侧菜单 46](#_Toc439083991)

[2.3 前台页面底部菜单 46](#_Toc439083992)

[2.4 用户中心左侧菜单 46](#_Toc439083993)

[2.5 管理平台主菜单 46](#_Toc439083994)

[3 脚本与样式 46](#_Toc439083995)

[第七章 路由 48](#_Toc439083996)

[1 路由解析流程图 48](#_Toc439083997)

[2 URL路由匹配规则 48](#_Toc439083998)

[3 GenericPathRoute 48](#_Toc439083999)

[第八章 权限 50](#_Toc439084000)

[第九章 插件 51](#_Toc439084001)

[1 新建插件类库 51](#_Toc439084002)

[1.1 插件项目位置 51](#_Toc439084003)

[1.2 插件项目名称 51](#_Toc439084004)

[2 设置项目编译文件DLL的输出位置 52](#_Toc439084005)

[3 建立插件描述文件 52](#_Toc439084006)

[4 实现接口Iplugin 53](#_Toc439084007)

[5 创建插件配置页面 54](#_Toc439084008)

[6 注册插件路由 56](#_Toc439084009)

[7 处理”安装“和”卸载“方法 57](#_Toc439084010)

[8 升级nopCommerce可能会让插件无法工作 59](#_Toc439084011)

[第十章 为管理平台添加功能：产品特有属性 60](#_Toc439084012)

[1 功能需求 60](#_Toc439084013)

[2 功能实现 60](#_Toc439084014)

[2.1 添加菜单项“产品属性” 60](#_Toc439084015)

[2.2 添加菜单项默认语言资源 61](#_Toc439084016)

[2.3 添加Control及Action 61](#_Toc439084017)

[2.4 注册接口及服务 63](#_Toc439084018)

[2.5 实现接口及服务 64](#_Toc439084019)

[2.6 实现实体类 67](#_Toc439084020)

[2.7 实现实体类的数据库映射类 68](#_Toc439084021)

[2.8 实现实体与模型的转换 70](#_Toc439084022)

[2.9 新建模型类及本地化模型类 70](#_Toc439084023)

[第十一章 联系信息 72](#_Toc439084024)

[参考资料 1](#_Toc439084025)

[1 ASP.NET MVC中使用AllowHtml属性 1](#_Toc439084026)

[2 @RenderBody、@RenderSection、@RenderPage、Html.RenderPartial、Html.RenderAction的作用和区别 2](#_Toc439084027)

[2.1 @RenderBody 2](#_Toc439084028)

[2.2 @RenderPage 2](#_Toc439084029)

[2.3 @RenderSection 3](#_Toc439084030)

[2.4 @Html.Partial 4](#_Toc439084031)

[3 ASP.NET MVC：自定义Route让Url更优雅 4](#_Toc439084032)

[3.1 目标 4](#_Toc439084033)

[3.2 基础知识：Route 是双向的 4](#_Toc439084034)

[3.3 自定义Route类 5](#_Toc439084035)

[3.4 自定义Route类必要的对应扩展方法 7](#_Toc439084036)

[3.5 使用自定义Route类 7](#_Toc439084037)

# NOPCOMMERCE架构分析

## 开篇

NopCommerce是.net开源项目中比较成熟的一款业务应用框架，也是电子商务系统中的典范。NopCommerce使用了Autofac这个IOC框架实现了松耦合的框架设计。

在IOC中，组件（在Autofac中对象又称为组件）首先需要在IOC中注册，可通过配置文件注册（如Spring.net）,也可通过特性注册的（如StructureMap）,也可通过代理来注册（如Autofac）。但是IOC讲究一个原则，就是接口和实现分离。所有IOC就是某个具体类实现了某个接口。然后在使用时，系统从IOC中获取接口的实现类，并创建对象。

NopCommerce的插件机制是通过Autofac来实现的。在Nop.Core.Infrastructure中充分体现了IOC的封装及灵活使用机制。

为了支持插件功能，以及支持一些自动注册的功能。NopCommerce提供了类型查找器。ITypeFinder以及实现类就是提供此功能。通过类型查找器可以查找本程序域中的类，也可以查找整个bin目录下所有动态链接库中类，并把它们注册到类型反转容器中。

在Nop中，容器管理类ContainerManager，管理通过Autofac生成的容器；引擎上下文EngineContext，根据配置文件生成单例Nop引擎，此引擎根据类型接口负责从容器中返回对象。Nop默认引擎是NopEngine，若没有配置有效的引擎，即用默认引擎，生成的引擎保存在单例容器中。

系统注册接口为：IDependencyRegistrar，并通过ITypeFinder类搜寻程序集里实现接口IDependencyRegistrar的类。代码如下：

namespace Nop.Core.Infrastructure.DependencyManagement

{

/// <summary>

/// 系统注册接口

/// <para>刘安全 2015年8月10日 10:20:58</para>

/// </summary>

public interface IDependencyRegistrar

{

/// <summary>

/// 通过ContainerBuilder注册依赖关系

/// <para>刘安全 2015年8月10日 10:20:58</para>

/// </summary>

/// <param name="builder">窗口生成器</param>

/// <param name="typeFinder">类型查找者接口</param>

void Register(ContainerBuilder builder, ITypeFinder typeFinder);

/// <summary>

/// 注册排序序号

/// <para>刘安全 2015年8月10日 10:20:58</para>

/// </summary>

int Order { get; }

}

}

Nop中单例类容器分为实体类、集合类和字典类的单例容器，即Singleton<T>，SingletonList<T>，SingletonDictionary<TKey, TValue>，单例类系列保存系统中与程序相同生命周期的单例对象，或者叫做单例类容器。EngineContext就是通过Singleton<T>类来管理引擎的。

## 数据库初试化及数据操作

系统启动时执行任务：IStartupTask,启动时执行的任务主要是数据库的初始化和加载。

SqlCeDataProvider：获取数据连接工厂，不同类型数据库，连接工厂不同。

在类型依赖注册类Nop.Web.Framework.DependencyRegistrar中实现数据库工厂的创建、数据库的加载。

## 插件机制

NopCommerce支持灵活的插件机制，所谓Web系统插件，其实也就是可以像原系统的一部分一样使用。

Web系统的使用方式就是客户端发送一个请求，服务端进行解析。在asp.net MVC中对客户请求的解析是通过路由的方式实现的。

所谓路由就是在客户端发生请求时，对请求路径的解析过程。

在Global.asax.cs中注册所有路由类：

var routePublisher = EngineContext.Current.Resolve<IRoutePublisher>();

routePublisher.RegisterRoutes(routes);

IRoutePublisher:路由发布者，以主动的方式向系统添加路由。实体类为:RoutePublisher。

IRouteProvider:路由提供者，每个插件都有一个路由提供者，向系统中注册路由。

路由发布者（IRoutePublisher）通过IOC在运行环境路径中查找路由提供者，即查找所有继承IRouteProvider的类。然后有各插件的路由提供者向系统中注册此插件的路由。

PluginManager：插件管理器，负责加载插件的DLL，并保存所有插件信息。此管理器在系统初始化时开始启动并通过Initialize方法加载所有插件。

IPluginFinder：插件查找者，在使用时从插件管理器中查找插件。

每一个插件都拥有自描述文件：Description.txt; 类PluginDescriptor负责保存解析后的插件描述或者说是此描述文件的解析后实体类。

每个插件都集成IPlugin接口，插件拥有安装和卸载的功能；

~/App\_Data/InstalledPlugins.txt：保存所有加载或安装了的插件；

~/Plugins：安装的插件所在目录；

~/Plugins/bin：每一个插件都会在此处建立一个备份；

## Model绑定Action参数

asp.net MVC中Action参数不只是一些基本类型，也支持实体参数。那么从客户端传来的数据如何映射或转换成实体对象呢？就是通过模型绑定类ModelBinder。此系列类在请求转化为后台Controller的Action方法前，捕获传递过来的数据，并对其进行解析和转换，最终为实体类对象。

NopModelBinder继承DefaultModelBinder承担系统的模型绑定类，但好像只是留一个借口，并没有使用。主要是继承父类的方法，稍有改变的地方是：方法BindModel添加了对NopModel的绑定支持。

BaseNopModel是nop所有Model的基类

## 自定义RazorViewEngine

系统中对Razor的支持包括两部分，其中之一就是自定义RazorViewEngine

### 自定义RazorViewEngine

在Global.asax.cs的Application\_Start方法中，注册了自定义视图引擎：

// 移除所有视图引擎

ViewEngines.Engines.Clear();

// 添加自定义的皮肤Razor视图引擎

ViewEngines.Engines.Add(new ThemeableRazorViewEngine());

ThemeableRazorViewEngine继承ThemeableBuildManagerViewEngine，

ThemeableBuildManagerViewEngine继承ThemeableVirtualPathProviderViewEngine

ThemeableVirtualPathProviderViewEngine继承VirtualPathProviderViewEngine，达到对虚拟路径解析的目的。

### 自定义类WebViewPage<TModel>

此类表示呈现使用ASP.NET Razor语法的视图所需的属性和方法。

所以每一个视图都应该继承此类。但是我们在项目中是看不到此继承的，默认情况下Razor会让视图继承自System.Web.Mvc.WebViewPage<TModel>基类。也可以通过修改视图目录（每一个asp.net mvc项目下面都有一个~/Views/目录）下的web.config文件来更改默认基类，NopCommerce就是使用此方法实现自定义WebViewPage类的。除此之外也可以在视图文件中引入命名空间，但这种方法比较繁琐，除非一个项目中有个别视图需要自定义WebViewPage。

## 皮肤管理器

IThemeContext：皮肤上下文，支持桌面皮肤和移动段皮肤。其实体类为:ThemeContext;

IThemeProvider:皮肤提供者，其实体类为ThemeProvider;皮肤提供者从皮肤配置文件中解析配置项，加载系统提供的所有皮肤方案，系统默认提供桌面干净皮肤（~/Themes/DefaultClean）和移动端皮肤(~/Themes/Mobile)。

ThemeConfiguration是皮肤配置类，它用于对theme.config数据进行解析。

IThemeContext从IThemeProvider中获取解析的配置方案。

上节讲到自定义视图引擎，从视图引擎的名字（ThemeableRazorViewEngine）上可以知晓，它实现了对皮肤的自动加载功能。

视图引擎从IThemeContext中获取当前的皮肤，并解析加载。

## 多语言

Language：语言，代表系统支持的语言

LocalizedProperty：本地化属性，即语言的多语言资源

IWorkContext及其实体类WebWorkContext为当前运行上下文；用户的登录信息以及一些上下文环境设置都保存在此类中。具体包括：当前用户信息，当前用户所用的货币、语言、税的类型，供应商等。

展现多语言资源的方式有几种：

一、在自定义类WebViewPage<TModel>中放置了方法：T()，通过此方法，网页在展现时获取对应语言的文字。

其实T只是一个代理，T的定义为：

public Localizer T

{

get

{

…

}

}

代理的定义为：

namespace Nop.Web.Framework.Localization

{

public delegate LocalizedString Localizer(string text, params object[] args);

}

此代理返回值类型为LocalizedString，此类继承接口IHtmlString，以保证能正确显示本地化的文字资源。

二、通过扩展HtmlHelper

类HtmlExtensions扩展了HtmlHelper类，主要是对一些控件的封装，并支持多语言。

方法 LocalizedEditor<T, TLocalizedModelLocal>是对Telerik的TabStrip控件的封装（也就是多页签控件---Tab控件），的。系统同时支持有多种语言时，多为每种语言显示一个页签，当然仅当需要时才这么做。这里面用到了接口ILocalizedModel和接口ILocalizedModelLocal。接口ILocalizedModel用来标示某Model类支持这种多语言显示，其中里面包括多种语言数据列表Locales，实现接口ILocalizedModelLocal的类就是特定一种语言的数据。LocalizedEditor方法就是根据这些接口的配合实现了支持多种语言页签了。Admin项目使用此方法，Web项目没有使用。

# 数据迁移

## 开启迁移功能

1. 打开Nop.Web项目中的web.config，添加连接字符串如下：

<connectionStrings>

<add name="NopDb" connectionString="Data Source=localhost;Initial Catalog=nopDemo82;Integrated Security=False;Persist Security Info=False;User ID=zaxx;Password=aq" providerName="System.Data.SqlClient" />

</connectionStrings>

**注意**：连接字符串是写在Nop.Web\web.config中。

1. 打开NopObjectContext.cs，添加默认构造函数：

public NopObjectContext():base("NopDb")

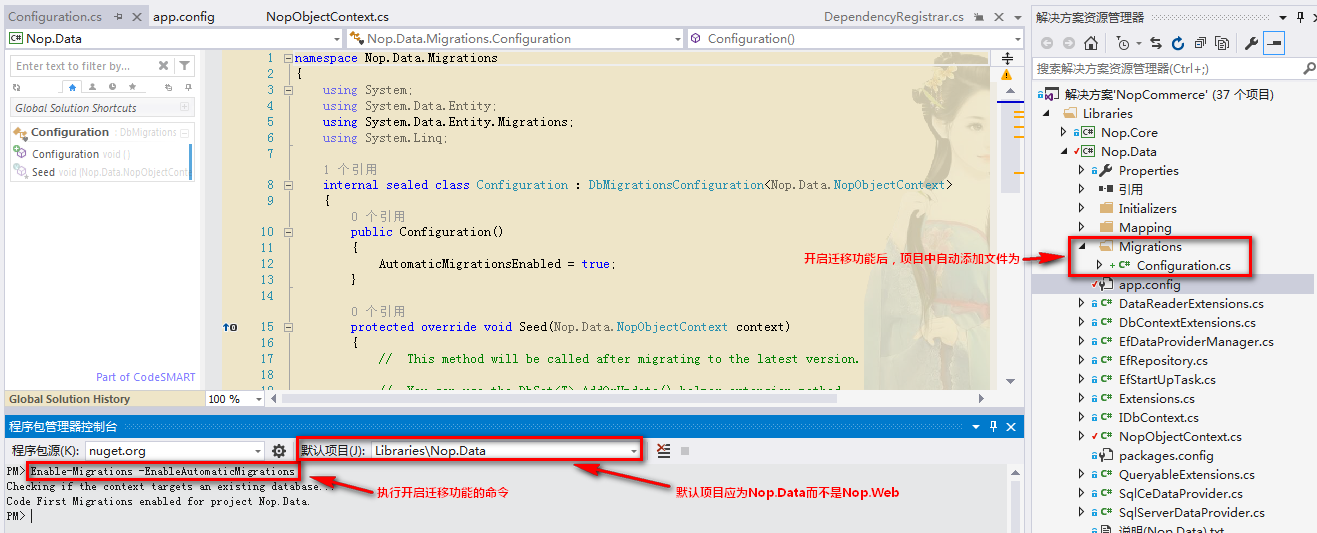
{

}

1. 打开程序包管理器控制台，使Nop.Data具备迁移功能：

Enable-Migrations –EnableAutomaticMigrations

执行开启迁移的命令后，项目变化如下图所示：

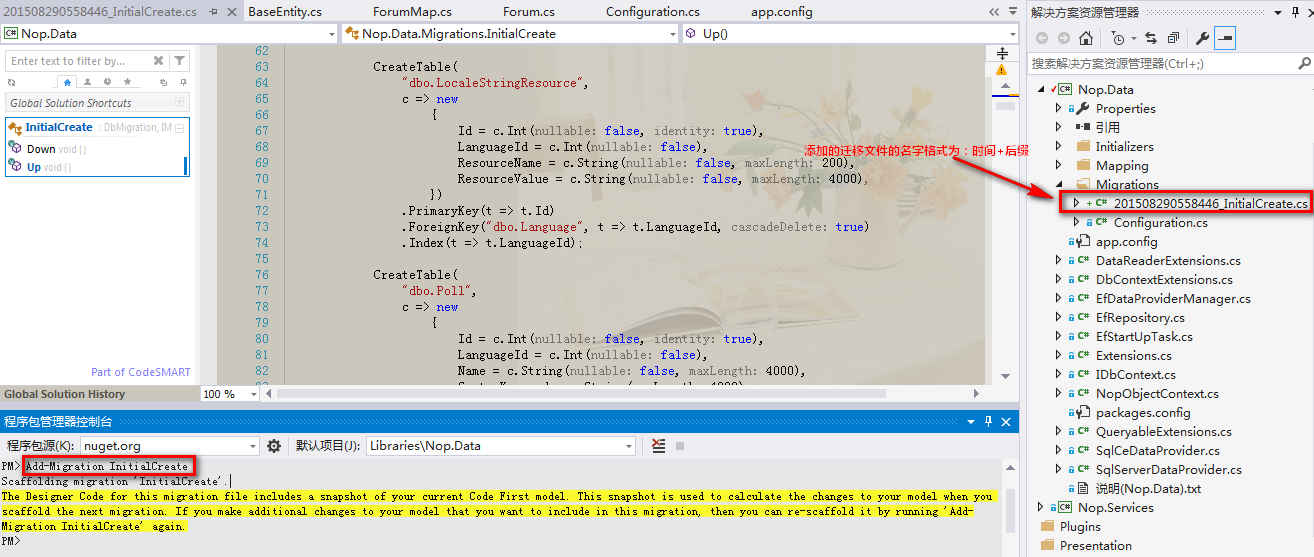


## 初次添加迁移文件

在程序包管理器控制台中，添加迁移文件。因之前未添加过迁移文件，故此次执行【添加迁移】的命令时，指定即将生成的迁移文件后缀为InitialCreate：

Add-Migration InitialCreate

执行添加迁移的命令后，项目变化如下图所示：



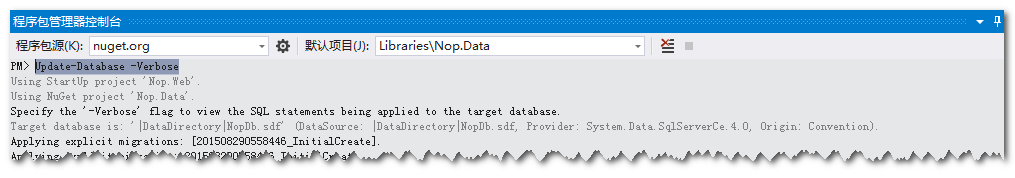
## 初次更新数据库

初次更新数据库即根据初次添加的迁移文件生成数据库。

在程序包管理器控制台中，更新数据库的命令为：Update-Database，若想看到执行更新操作时的详细信息，可以带上参数Verbose。

Update-Database -Verbose

执行更新数据库命令后程序包管理器控制台显示信息如下：

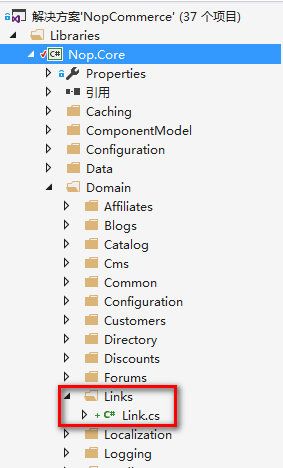


此时在数据库中，生成了数据库。

## 新增实体及实体映射后添加迁移文件

### 新增实体Link

新增实体Link，新增实体后项目结构图为：



新增的实体Link代码为：

namespace Nop.Core.Domain.Links

{

/// <summary>

/// 代表一个链接

/// <para>Represents a link</para>

/// <para>刘安全 2015年8月29日 14:20:01</para>

/// </summary>

public class Link : BaseEntity

{

public string Name { get; set; }

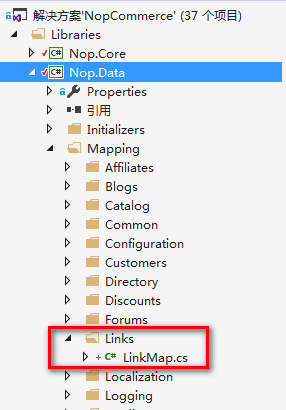
public string Value { get; set; }

}

}

### 新增实体映射LinkMap

新增实体映射LinkMap后，项目结构图为：



新增的实体映射LinkMap代码为：

using Nop.Core.Domain.Links;

namespace Nop.Data.Mapping.Links

{

/// <summary>

/// 链接映射类

/// <para>映射链接实体类Link与数据库中的数据表Links</para>

/// <para>刘安全 2015年8月29日 14:20:12</para>

/// </summary>

public class LinkMap : NopEntityTypeConfiguration<Link>

{

public LinkMap()

{

this.ToTable("Links");

this.HasKey(t => t.Id);

this.Property(l => l.Name).HasMaxLength(100);

this.Property(l => l.Value).HasMaxLength(300);

}

}

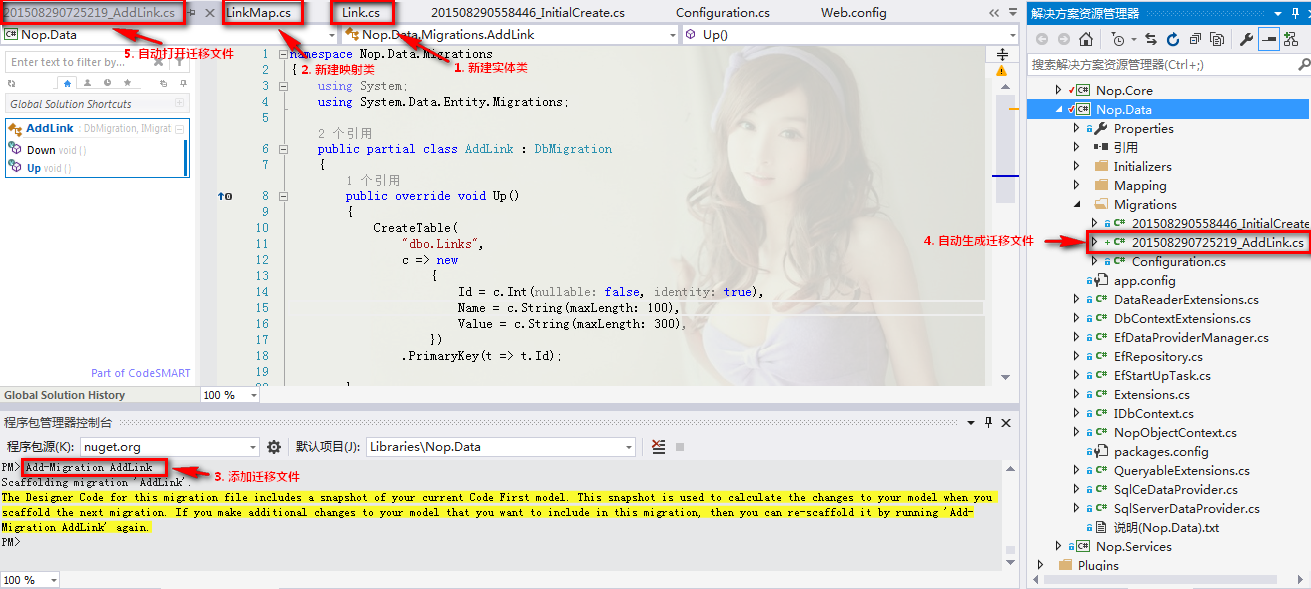
}

### 添加迁移文件

因新增了实体，为达到实体与数据库表的一一对应，需要添加迁移文件。在程序包管理器控制台中，执行【增加迁移】的命令如下：

Add-Migration AddLink

执行【增加迁移】命令后，界面截图如下：



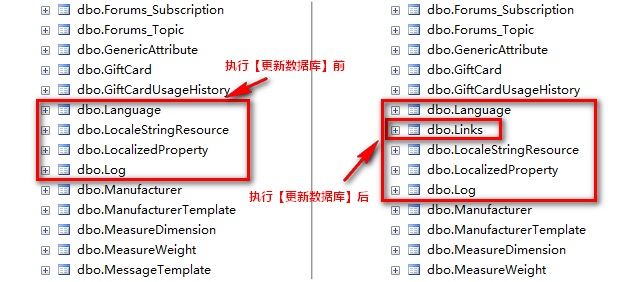
**注意**：增加实体类后，执行【增加迁移】命令后生成的迁移文件中的Up与Down应分别对应着创建及删除实体类对应的表的代码。如果Up、Down中没有创建、删除表的代码，则可【重新生成解决方案】，然后再执行【增加迁移】命令，应该就有相应代码了。

### 更新数据库

在程序包管理器控制台中，更新数据库的命令为：Update-Database，若想看到执行更新操作时的详细信息，可以带上参数Verbose。

Update-Database -Verbose

执行更新数据库命令后程序包管理器控制台显示信息如下：



## 添加新增实体的示例数据

在Libraries\Nop.Services\Installation\CodeFirstInstallationService.cs文件中添加服务成员变量：

private readonly IRepository<Link> \_linkRepository;

然后给CodeFirstInstallationService构造函数添加对应的参数IRepository<LinkModel> linkRepository,并给\_linkRepository赋值。

接着添加函数：

protected virtual void InstallLinks()

{

List<Link> links = new List<Link>()

{

new Link {Name="Zaxx",Value="最爱肖霄——刘安全" }

};

\_linkRepository.Insert(links);

}

最后在InstallData函数中添加对InstallLinks()函数的调用。

另外还要在Presentation\Nop.Web\App\_Data\Install\create\_sample\_data.sql文件中添加下面语句：

SET IDENTITY\_INSERT [dbo].[Link] ON

INSERT [dbo].[Link] ([Id], [Name], [Value]) VALUES (1, N'Zaxx', N'最爱肖霄——刘安全')

SET IDENTITY\_INSERT [dbo].[Link] OFF

GO

DependencyRegistrar的Register中会根据Web.config文件中UseFastInstallationService(使用快速安装服务)的配置值来判断是用SqlFileInstallationService的InstallData执行create\_sample\_data.sql文件向数据库中插入演示数据，还是使用CodeFirstInstallationService的InstallData向数据库中插入演示数据。Web.config文件中UseFastInstallationService为默认是false，将CodeFirstInstallationService的InstallData向数据库中插入演示数据(注意：若插入的中文数据是乱码，则用记事本打开CodeFirstInstallationService.cs，把它的编码改为UTF-8)。

**注意**：迁移时未向数据库中插入演示数据，因为在之前的代码中，未在Configuration.cs的Seed方法中给出演示数据。若有需要，可以自行给出。在数据库中已有演示数据时，按之前所述的迁移步骤对数据库进行更新时，已有演示数据是不会丢失的。

# 多语言

Nop中给出了很好的多语言处理。

### 获取实体数据的多语言

只要是实现了BaseEntity, ILocalizedEntity的实体类型数据，都会有**GetLocalized**方法，此方法可以获取实体数据指定属性的多语言信息。这个扩展方法**GetLocalized**是在Libraries\Nop.Services\Localization\LocalizationExtensions中定义的：

// 为所有实现了BaseEntity, ILocalizedEntity的实体类型扩展获取本地化方法

public static string GetLocalized<T>(this T entity,

Expression<Func<T, string>> keySelector)

where T : BaseEntity, ILocalizedEntity

{

var workContext = EngineContext.Current.Resolve<IWorkContext>();

return GetLocalized(entity, keySelector, workContext.WorkingLanguage.Id);

}

调用示例如：

var manufacturerName =manufacturer.GetLocalized(m => m.Name);

### 前端页面获取资源的多语言

在Web项目的前端页面文件中，若要获取指定资源的本地化语言，则只需要给出格式类似于如下的语句即可：

@T("Page.Breadcrumb.Home")

这里T的括号中给出的是一个资源名称，而这个T，实际是一个方法，它是在WebViewPage<TModel>中定义的一个属性，具体位置是在Presentation\Nop.Web.Framework\ViewEngines\Razor\WebViewPage.cs中定义：

public Localizer T

{

get

{

if (\_localizer == null)

{

//null localizer

//\_localizer = (format, args) => new LocalizedString((args == null || args.Length == 0) ? format : string.Format(format, args));

//default localizer

\_localizer = (format, args) =>

{

var resFormat = \_localizationService.GetResource(format);

if (string.IsNullOrEmpty(resFormat))

{

return new LocalizedString(format);

}

return

new LocalizedString((args == null || args.Length == 0)

? resFormat

: string.Format(resFormat, args));

};

}

return \_localizer;

}

}

### 后台文件获取资源的多语言

在项目的cs文件中，若要获取指定资源的本地化语言，则只需要给出格式类似于如下的语句即可：

\_localizationService.GetResource("Admin.Codes.AvaiableProperties.Now")

### 将多语言资源添加到数据库中

在SqlFileInstallationService文件的InstallLocaleResources方法中实现了：添加指定后缀的资源文件到给定语言，以做为此语言的本地化资源信息：

protected virtual void InstallLocaleResources()

{

//'English' language

var language = (\_languageRepository.Table2).Single(l => l.Name == "English");

//save resources

foreach (var filePath in System.IO.Directory.EnumerateFiles(\_webHelper.MapPath("~/App\_Data/Localization/"), "\*.nopres.xml", SearchOption.TopDirectoryOnly))

{

var localesXml = File.ReadAllText(filePath);

var localizationService = EngineContext.Current.Resolve<ILocalizationService>();

localizationService.ImportResourcesFromXml(language, localesXml);

}

}

代码中的English指的是App\_Data\Install\create\_required\_data.sql文件中添加的数据中的English：

SET IDENTITY\_INSERT [dbo].[Language] ON

INSERT [dbo].[Language] ([Id], [Name], [LanguageCulture], [UniqueSeoCode], [FlagImageFileName], [Rtl], [LimitedToStores], [DefaultCurrencyId], [Published], [DisplayOrder])

VALUES (1, N'English', N'en-US', N'en', N'us.png', 0, 0, 0, 1, 1)

SET IDENTITY\_INSERT [dbo].[Language] OFF

GO

# 添加新功能

## 添加实体实现新功能的一般步骤

1. 新增实体类

如第2章第4节4.1《新增实体Link》

1. 新增实体映射

如第2章第4节4.2《新增实体映射LinkMap》

1. 添加示例数据

如第2章第5节《添加新增实体的示例数据》

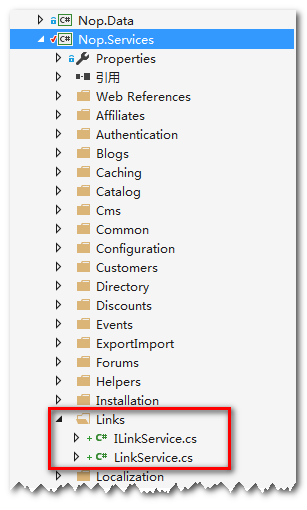
1. 创建实体服务接口与实体服务类
2. 注册服务接口及服务类
3. 新增前端演示模型
4. 新增验证文件
5. 在资源文件中添加所需资源
6. 添加Web缓存处理
7. 添加控制器及视图
8. 注册路由

在第二章第4节及第5节里，已经讲述了“添加新功能一般步骤”中的前3个步骤。本章将讲述剩余步骤。

## 创建实体服务接口与实体服务类

当给出实体类、实体映射类后，即实现了数据层，接着就要给出业务层。在Nop中，业务访问层（BAL）是在Nop.Services中实现的。Nop.Services项目包含一系列的核心服务、业务逻辑、验证，如果有数据的话还有数据的计算方法。

在Nop.Services项目根目录下新建Links目录，在该Links目录下创建服务ILinkService接口和LinkService类后项目结构如图：



ILinkService接口代码为：

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Assembly : Nop.Services

// Author : 刘安全

// Created : 08-29-2015

//

// Last Modified By : 刘安全

// Last Modified On : 2015-08-29 17:17:00

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// <copyright file="ILinkService.cs" company="Nop Solutions, Ltd">

// Copyright © Nop Solutions, Ltd 2010

// </copyright>

// <summary></summary>

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using System.Collections.Generic;

using Nop.Core.Domain.Links;

namespace Nop.Services.Links

{

/// <summary>

/// 链接服务接口

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

public interface ILinkService

{

/// <summary>

/// 删除链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="link">要删除的链接对象</param>

void DeleteLink(Link link);

/// <summary>

/// 通过ID获取链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="linkId">链接对象的ID字符串</param>

/// <returns>通过ID值获取到的链接.</returns>

Link GetLinkById(int linkId);

/// <summary>

/// 获取所有链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <returns>连接序列</returns>

IList<Link> GetAllLinks();

/// <summary>

/// 插入链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="link">欲插入数据库的链接对象</param>

void InsertLink(Link link);

/// <summary>

/// 更新链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="link">欲更新的链接对象</param>

void UpdateLink(Link link);

}

LinkService类的代码为：

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Assembly : Nop.Services

// Author : 刘安全

// Created : 08-29-2015

//

// Last Modified By : 刘安全

// Last Modified On : 2015-09-01 10:26:57

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// <copyright file="LinkService.cs" company="Nop Solutions, Ltd">

// Copyright © Nop Solutions, Ltd 2010

// </copyright>

// <summary></summary>

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using System;

using System.Collections.Generic;

using System.Linq;

using Nop.Core.Caching;

using Nop.Core.Data;

using Nop.Core.Domain.Links;

using Nop.Services.Events;

namespace Nop.Services.Links

{

/// <summary>

/// 链接服务

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

public class LinkService : ILinkService

{

/// <summary>

/// 获取或设置所有链接对象缓存时用到的键

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

private const string LinksAllKey = "Nop.links.all";

/// <summary>

/// 获取或设置链接对象缓存时用到的键，其格式为：Nop.links.id-{0}，其中{0}替换为链接的Id

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

private const string LinksByIdKey = "Nop.links.id-{0}";

/// <summary>

/// 清除缓存匹配Nop.links.模式的缓存

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

private const string LinksPatternKey = "Nop.links.";

/// <summary>

/// 缓存管理器

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

private readonly ICacheManager \_cacheManager;

/// <summary>

/// 事件公布器

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

private readonly IEventPublisher \_eventPublisher;

/// <summary>

/// 链接仓库

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

private readonly IRepository<Link> \_linkRepository;

/// <summary>

/// 初始化<see cref="LinkService" /> 类的一个新实例.

/// <para>刘安全 2015-08-29 17:44:27</para>

/// </summary>

/// <param name="linkRepository">链接仓库.</param>

/// <param name="eventPublisher">事件发布者 The event publisher.</param>

/// <param name="cacheManager">高速缓存管理器.</param>

public LinkService(IRepository<Link> linkRepository, IEventPublisher eventPublisher, ICacheManager cacheManager)

{

this.\_linkRepository = linkRepository;

this.\_eventPublisher = eventPublisher;

this.\_cacheManager = cacheManager;

}

/// <summary>

/// 删除链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="link">要删除的链接对象</param>

/// <exception cref="System.ArgumentNullException"></exception>

public virtual void DeleteLink(Link link)

{

// 如果参数为null则抛出参数为空的异常

if (link == null)

throw new ArgumentNullException(nameof(link));

// 如果链接对象不为null，则利用链接仓库把它从数据库中删除

\_linkRepository.Delete(link);

/\* 清除缓存中键匹配Nop.links.的缓存，即清除缓存中所有Link的缓存。之所以要清除匹配Nop.links.的缓存而不是

只清除这一个对象的缓存，是因为缓存中可能不是只有当前Link对象的缓存，还可能存在包含当前对象的链接列表。

当对象已删除时，若链接列表未从缓存中删除，则会因列表中包含已删除对象，而导致数据不准确 \*/

\_cacheManager.RemoveByPattern(LinksPatternKey);

\_eventPublisher.EntityDeleted(link);

}

/// <summary>

/// 通过ID获取链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="linkId">链接对象的ID字符串</param>

/// <returns>通过ID值获取到的链接.</returns>

public virtual Link GetLinkById(int linkId)

{

if (linkId == 0)

return null;

// 生成链接对象统一格式的缓存键。键的格式为：Nop.links.id-{0}

var key = string.Format(LinksByIdKey, linkId);

// 从缓存中获取key关联的链接对象。若不存在，则从链接仓库中获取ID为给定linkId值的链接对象(此对象将同时由\_cacheManager.Getw做缓存处理)。

return \_cacheManager.Get(key, () => \_linkRepository.GetById(linkId));

}

/// <summary>

/// 获取所有链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <returns>连接序列</returns>

public virtual IList<Link> GetAllLinks()

{

var key = string.Format(LinksAllKey);

/\* 从缓存中获取键为Nop.links.all的缓存对象（链接对象列表）。若缓存对象不存在，则从链接仓库中获取所有链接

并返回链接对象列表(此对象列表将同时由\_cacheManager.Getw做缓存处理)。\*/

return \_cacheManager.Get(key, () =>

{

var query = \_linkRepository.Table;

return query.ToList();

});

}

/// <summary>

/// 插入链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="link">欲插入数据库的链接对象</param>

/// <exception cref="System.ArgumentNullException">link</exception>

public virtual void InsertLink(Link link)

{

if (link == null)

throw new ArgumentNullException(nameof(link));

// 将链接对象添加到数据库中

\_linkRepository.Insert(link);

/\* 清除缓存中键匹配Nop.links.的缓存，即清除缓存中所有Link的缓存。之所以要清除匹配Nop.links.的缓存而不是

只增加这一个对象的缓存，是因为缓存中可能存在包含当前对象的链接列表。当增加一个Link对象时，若链接列表

未从缓存中删除，则会因列表中不包含新增的Link对象，而导致数据不准确 \*/

\_cacheManager.RemoveByPattern(LinksPatternKey);

\_eventPublisher.EntityInserted(link);

}

/// <summary>

/// 更新链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <param name="link">The link.</param>

/// <exception cref="System.ArgumentNullException">link</exception>

public virtual void UpdateLink(Link link)

{

if (link == null)

throw new ArgumentNullException(nameof(link));

\_linkRepository.Update(link);

/\* 清除缓存中键匹配Nop.links.的缓存，即清除缓存中所有Link的缓存。之所以要清除匹配Nop.links.的缓存而不是

只清除这一个对象的缓存，是因为缓存中可能存在包含当前对象的链接列表。当Link对象的信息已改变时，若链接

列表未从缓存中删除，则会因列表中当前Link对象的信息未改变，而导致数据不准确 \*/

\_cacheManager.RemoveByPattern(LinksPatternKey);

\_eventPublisher.EntityUpdated(link);

}

}

}

## 注册服务接口及服务类

Nop采用的是IOC机制，在前端中要调用业务层Nop.Service功能时，需要在presentation\nop.web.framework\DependencyRegistrar.cs文件的public virtual void Register(ContainerBuilder builder, ITypeFinder typeFinder)函数添加代码服务接口与服务类的注册功能，代码如下：

builder.RegisterType<LinkService>().As<ILinkService>().InstancePerLifetimeScope();

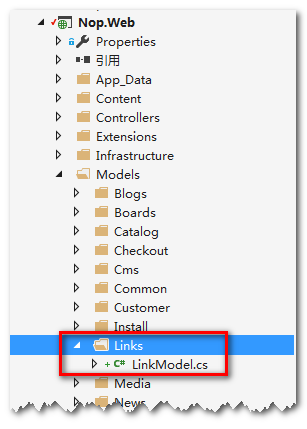
**注意**：如果功能是添加到插件或区域中，则在对应的插件或区域的依赖注册DependencyRegistrar中做服务接口与服务类的注册。

另注意：nop.web.framework\DependencyRegistrar.cs文件的Register主要是注册业务层Nop.Service的接口与实现类；nop.web\Infrastructure\DependencyRegistrar.cs文件的Register是注册控制器及控制器构造函数中参数的实参；插件的DependencyRegistrar.cs文件的Register是注册插件中所提供的服务接口与实现类。

## 新增前端演示模型

新增的实体Link如果要在页面上显示，需先建立实体对应的演示模型。

在Nop.Web\Models目录下新建Links目录，并在目录中新建links数据表相关的前端展示数据模型类LinkModel.cs，此时Nop.Web文件结构如下：



LinkModel代码如下：

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Assembly : Nop.Web

// Author : 刘安全

// Created : 09-01-2015

//

// Last Modified By : 刘安全

// Last Modified On : 2015-09-01 10:57:58

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// <copyright file="LinkModel.cs" company="Nop Solutions, Ltd">

// Copyright © Nop Solutions, Ltd 2010

// </copyright>

// <summary></summary>

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using System.Web.Mvc;

using FluentValidation.Attributes;

using Nop.Web.Framework.Mvc;

using Nop.Web.Validators.Links;

namespace Nop.Web.Models.Links

{

/// <summary>

/// 链接模型

/// <para> 刘安全 2015年09月01日 10:57:55</para>

/// </summary>

public class LinkModel : BaseNopEntityModel

{

/// <summary>

/// 获取或设置链接的名字

/// <para>刘安全 2015-09-01 10:57:58</para>

/// </summary>

/// <value>The name.</value>

[AllowHtml]

public string Name { get; set; }

/// <summary>

/// 获取或设置链接的值

/// <para>刘安全 2015-09-01 10:57:58</para>

/// </summary>

/// <value>The value.</value>

[AllowHtml]

public string Value { get; set; }

}

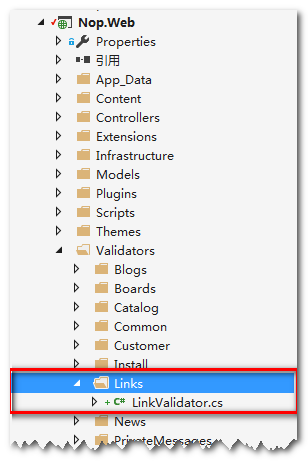
}

AllowHtml是禁用字段的验证，使该字段可以输入HTML标记，如<br/>。详见“参考资料”第一小节《[ASP.NET MVC中使用AllowHtml属性](#_ASP.NET_MVC中使用AllowHtml属性)》

## 新增验证文件

如果前端页面需要在输入时给出验证，那么可以为演示模型添加验证文件。若不需要，则不用添加。

为演示模型LinkModel新增验证文件是在Nop.Web\Validators目录中创建Links子目录，并在Links目录中创建LinkValidator.cs，此时Nop.Web文件结构如下：



LinkValidator.cs的代码为：

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Assembly : Nop.Web

// Author : 刘安全

// Created : 09-01-2015

//

// Last Modified By : 刘安全

// Last Modified On : 2015-09-01 11:47:58

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// <copyright file="LinkValidator.cs" company="Nop Solutions, Ltd">

// Copyright © Nop Solutions, Ltd 2010

// </copyright>

// <summary></summary>

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using FluentValidation;

using Nop.Services.Localization;

using Nop.Web.Framework.Validators;

using Nop.Web.Models.Links;

namespace Nop.Web.Validators.Links

{

/// <summary>

/// 链接验证器

/// <para> 刘安全 2015年09月01日 11:42:48</para>

/// </summary>

public class LinkValidator : BaseNopValidator<LinkModel>

{

/// <summary>

/// 初始化类链接验证器的实例.

/// <para>刘安全 2015-09-02 16:45:55</para>

/// </summary>

/// <param name="localizationService">本地化服务</param>

public LinkValidator(ILocalizationService localizationService)

{

// 当Name为空时，显示Link.Name所指定资源

RuleFor(x => x.Name).NotEmpty().WithMessage(localizationService.GetResource("Link.Name"));

RuleFor(x => x.Value).NotEmpty().WithMessage(localizationService.GetResource("Link.Value"));

}

}

}

新建模型验证器后，为演示模型类添加验证属性，指明演示模型所用验证器。重要代码如下：

[Validator(typeof (LinkValidator))]

public class LinkModel : BaseNopEntityModel

{

…

}

## 在资源文件中添加所需资源

在新增验证文件时，指明验证不通过时的提示信息，可以用字符串直接给出，也可以用资源字符串给出，如：

// 提示信息直接用字符串给出

RuleFor(x => x.Name).NotEmpty().WithMessage(@”名字不能为空！”);

// 提示信息用资源字符串给出

RuleFor(x => x.Name).NotEmpty().WithMessage(localizationService.GetResource("Link.Name"));

使用字符串的优点时编程简单明了，但缺点非常明显：后期若要更改提示，必须得改源代码。而使用资源字符串，程序发布后，用户也可以自己更改提示，且可以解决多语言时的验证提示问题。所以应该选择使用资源字符串。

前一小节中的LinkValidator在实现时就用到了资源字符串Link.Name、Link.Value。需要在Presentation\Nop.Web\App\_Data\Localization\defaultResources.nopres.xml文件中添加：

<LocaleResource Name="Link.Name">

<Value>Enter name</Value>

</LocaleResource>

<LocaleResource Name="Link.Value">

<Value>Enter value</Value>

</LocaleResource>

**注意**：

1. 系统在初次安装数据库时,会将defaultResources.nopres.xml中的资源字符串读取到数据库的LocaleStringResource表中。
2. defaultResources.nopres.xml中的默认语言环境是English。
3. 如果不是初次运行，新增的语言资源字符串配置是不会主动添加到LocaleStringResource表中，可以直接使用sql insert语句插入，也可以运行Nop在管理平台的Language配置界面添加。

## 添加Web缓存处理

在实体服务类中，获取实体或实体序列信息时，都有做缓存处理。即获取实体信息时，优先从缓存中获取，获取不到时再从数据库中获取（从数据库中获取到后，同时存储到缓存中以备下次使用）。如本章第2小节中实体服务类LinkService中获取所有Link实体的代码为：

/// <summary>

/// 获取所有链接.

/// <para>刘安全 2015-08-29 17:17:00</para>

/// </summary>

/// <returns>连接序列</returns>

public virtual IList<Link> GetAllLinks()

{

var key = string.Format(LinksAllKey);

/\* 从缓存中获取键为Nop.links.all的缓存对象（链接对象列表）。若缓存对象不存在，则从链接仓库中获取所有链接

并返回链接对象列表(此对象列表将同时由\_cacheManager.Getw做缓存处理)。\*/

return **\_cacheManager.Get(key, () =>**

**{**

**var query = \_linkRepository.Table;**

**return query.ToList();**

**});**

}

如上代码加粗部分是从缓存中获取指定key的缓存数据，若未获取到缓存数据，则从数据库中查询并使用查询到的数据，且这些数据还会存储到缓存中，且缓存的数据对应的缓存key为Get第一个参数给定的key。

### 指定缓存key格式

在Nop.Web中，若要把实体信息显示出来，需要在控制器Control里将获取的实体信息转换为演示模型，然后按规定布局显示在视图上。而在控制器中获取的演示模型时，若实体信息转演示模型时操作较复杂或计算量较大，则可进一步采用缓存处理。在对本章前面所讲到的新增实例做进一步缓存时，在Presentation\Nop.Web\Infrastructure\Cache\ModelCacheEventConsumer.cs文件中指定演示模型的缓存key格式：

/// <summary>

/// 链接模型缓存的Key

/// </summary>

/// <remarks>

/// {0} : 查询关键字

/// </remarks>

public const string LinkModelKey = "Nop.pres.link.info-{0}";

**注意**：

1. 这不是一个必须的操作：若实体转演示模型时只涉及到实体本身，则没有必要再做这个操作；若实体转演示模型时，还涉及其他信息，如演示模型是由实体及其他实体信息揉合而成，则可做此处理。
2. 如果功能是添加到插件或区域中，则在对应的插件或区域的模型缓存事件用户ModelCacheEventConsumer中做如上的缓存key格式处理。

### 通过缓存key获取缓存数据

在LinkController中获取演示模型数据时，可以从web缓存中获取，若web缓存中不存在，才通过服务层（即业务层）获取，并把获取到的数据存储到web缓存管理器中，以便于后期再取数据时能从缓存中直接获取。示例代码如下：

public ActionResult Search(string keyword)

{

// 获取规定格式的缓存键

var cacheKey = string.Format(ModelCacheEventConsumer.LinkModelKey, keyword);

// 获取给定key对应的缓存数据

**var cacheModel = \_cacheManager.Get(cacheKey, () =>**

**{**

**var links = \_linkService.GetAllLinks()**

**.Where(item=>item.Name == keyword) // 按查询关键字过滤**

**.Select(item=>new LinkModel() {Name = item.Name,Value = item.Value}) // 实体转演示模型**

**.ToList();**

**if (!links.Any())**

**return null;**

**return links;**

**});**

if (cacheModel == null)

return Content("");

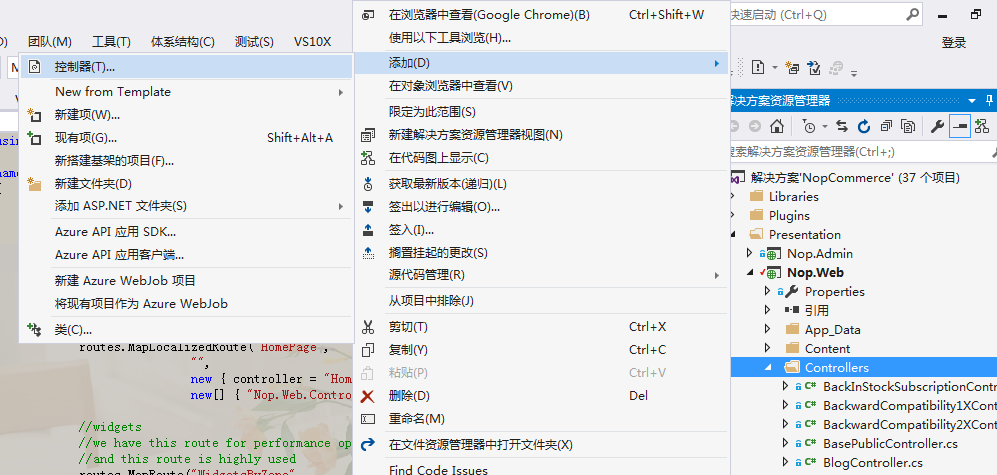
return View(cacheModel);

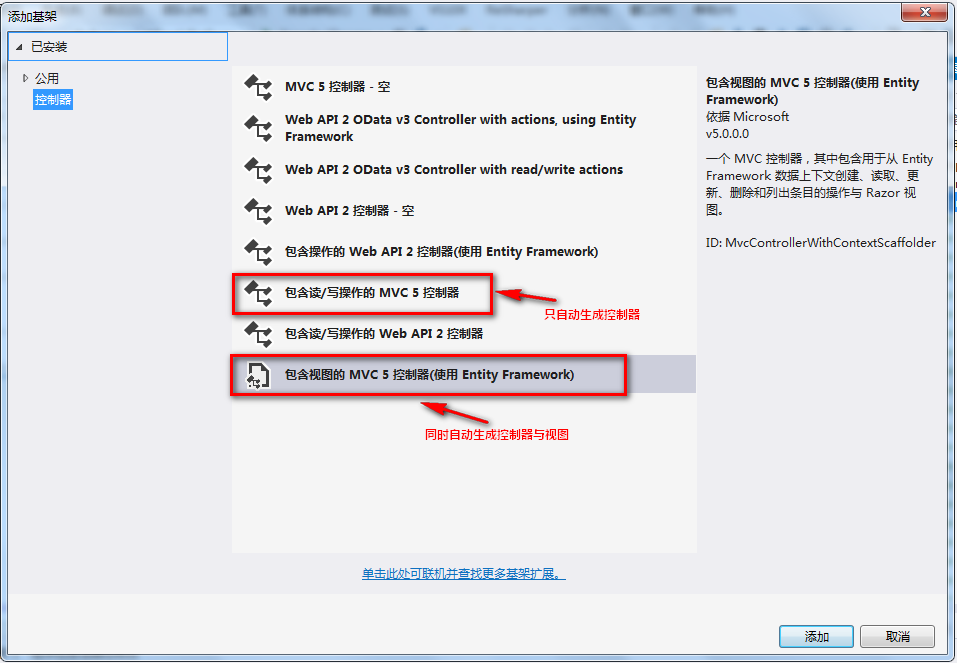
}

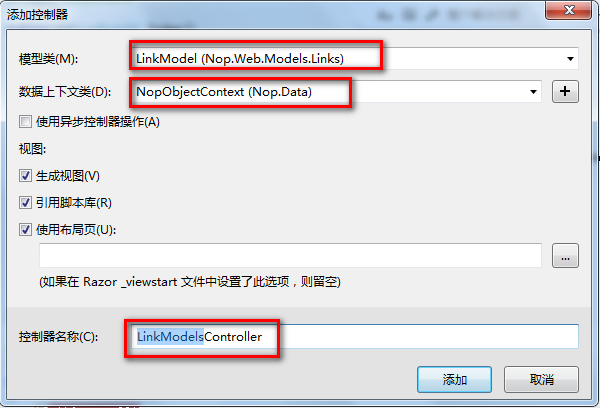
## 添加控制器及视图

### 添加控制器

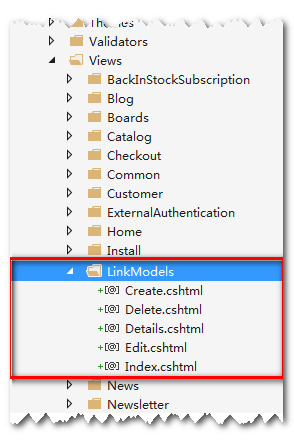
在Presentation\Nop.Web\Controllers上右击，选择“添加 -> 控制器”，如下3个截图：







这里选择的数据上下文类，只是因要用这个“自动生成控制器及视图”的模板而不得不选择的，在自动生成文件后，会看到控制器里自动添加了一个LinkModelsController文件，在View里自动添加了一个LinkModels目录，并自动生成了Create、Delete、Details、Edit、Index等5个视图文件。此时项目文件结构如下图所示：



删除NopObjectContext里由模板自动添加的语句，然后修改LinkModelsController文件，最终代码如下：

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Assembly : Nop.Web

// Author : 刘安全

// Created : 09-05-2015

//

// Last Modified By : 刘安全

// Last Modified On : 2015-09-05 17:01:00

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// <copyright file="LinkModelsController.cs" company="Nop Solutions, Ltd">

// Copyright © Nop Solutions, Ltd 2010

// </copyright>

// <summary></summary>

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using System;

using System.Linq;

using System.Net;

using System.Web.Mvc;

using Nop.Core.Caching;

using Nop.Core.Domain.Links;

using Nop.Services.Links;

using Nop.Web.Infrastructure.Cache;

using Nop.Web.Models.Links;

namespace Nop.Web.Controllers

{

/// <summary>

/// 连接模型控制器

/// <para>刘安全 2015-09-05 17:01:00</para>

/// </summary>

public class LinkModelsController : Controller

{

/// <summary>

/// 缓存管理器

/// <para>刘安全 2015-09-05 17:01:00</para>

/// </summary>

private readonly ICacheManager \_cacheManager;

/// <summary>

/// 链接服务

/// <para>刘安全 2015-09-05 17:01:00</para>

/// </summary>

private readonly ILinkService \_linkService;

/// <summary>

/// 创建链接模型控制器<see cref="LinkModelsController"/>实例.

/// <para>刘安全 2015-09-05 17:01:00</para>

/// </summary>

/// <param name="linkService">The link service.</param>

/// <param name="cacheManager">The cache manager.</param>

public LinkModelsController(ILinkService linkService,

ICacheManager cacheManager)

{

this.\_linkService = linkService;

this.\_cacheManager = cacheManager;

}

public ActionResult Index()

{

// 获取规定格式的缓存键

var cacheKey = ModelCacheEventConsumer.LinkModelAllKey;

// 获取给定key对应的缓存数据

var cacheModels = \_cacheManager.Get(cacheKey, () =>

{

var links = \_linkService.GetAllLinks()

.Select(item => new LinkModel {Id=item.Id, Name = item.Name, Value = item.Value}) // 实体转演示模型

.ToList();

if (!links.Any())

return null;

return links;

});

if (cacheModels == null)

return Content("");

return View(cacheModels);

}

// GET: LinkModels/Details/5

public ActionResult Details(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

var linkModel = PrepareLinkInfoModel(\_linkService.GetLinkById((int) id));

if (linkModel == null)

{

return HttpNotFound();

}

return View(linkModel);

}

// NonAction表示它不是一个真正的Action，而是一个普通方法；

[NonAction]

protected virtual LinkModel PrepareLinkInfoModel(Link link)

{

if (link == null)

throw new ArgumentNullException("link");

var linkModel = new LinkModel

{

Id = link.Id,

Name = link.Name,

Value = link.Value

};

return linkModel;

}

// GET: LinkModels/Create

public ActionResult Create()

{

return View();

}

// POST: LinkModels/Create

[HttpPost] // 指定只有POST请求才能调用这个Action方法

[ValidateAntiForgeryToken] // 和视图中的@Html.AntiForgeryToken(成对出现，用来阻止伪造的请求

public ActionResult Create([Bind(Include = "Id,Name,Value")] LinkModel linkModel)

{

if (ModelState.IsValid)

{

\_linkService.InsertLink(new Link {Name = linkModel.Name, Value = linkModel.Value});

return RedirectToAction("Index");

}

return View(linkModel);

}

// GET: LinkModels/Edit/5

public ActionResult Edit(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

var linkModel = PrepareLinkInfoModel(\_linkService.GetLinkById((int) id));

if (linkModel == null)

{

return HttpNotFound();

}

return View(linkModel);

}

// POST: LinkModels/Edit/5

// 为了防止“过多发布”攻击，请启用要绑定到的特定属性，有关

// 详细信息，请参阅 http://go.microsoft.com/fwlink/?LinkId=317598。

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Edit([Bind(Include = "Id,Name,Value")] LinkModel linkModel)

{

if (ModelState.IsValid)

{

\_linkService.UpdateLink(new Link {Id = linkModel.Id, Name = linkModel.Name, Value = linkModel.Value});

return RedirectToAction("Index");

}

return View(linkModel);

}

// GET: LinkModels/Delete/5

public ActionResult Delete(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

var linkModel = PrepareLinkInfoModel(\_linkService.GetLinkById((int) id));

if (linkModel == null)

{

return HttpNotFound();

}

return View(linkModel);

}

// POST: LinkModels/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public ActionResult DeleteConfirmed(int id)

{

var link = \_linkService.GetLinkById(id);

\_linkService.DeleteLink(link);

return RedirectToAction("Index");

}

}

}

### 添加视图

在添加控制器时，若选择模板时选择的是“包含视图的控制器”，则在添加控制器的同时，自动添加了相应的视图文件；如果选择的是“包含读/写操作的控制器”，则向导只在项目中添加控制器，并不生成视图文件，此时会发现各个Action的return View语句有红色下划线，标明Action对应的视图文件需要创建。

若控制器中Action需要添加对应视图，则可在Action方法体中右击并选择“添加视图”，并根据需要在向导里给出必要信息即可快速创建Action对应的视图。

## 注册路由

### 全局路由

在presentation\nop.web\Global.asax文件的Application\_Start方法中调用了RegisterRoutes方法，而在RegisterRoutes方法中指定了默认的url路由规则：

routes.MapRoute(

"Default", // 要映射的路由的名称

"{controller}/{action}/{id}", // 路由的 URL 模式

new { controller = "Home", action = "Index", id = UrlParameter.Optional }, // 一个包含默认路由值的对象

new[] { "Nop.Web.Controllers" } // 一组命名空间

);

在url只是给出<http://127.0.0.1:15536>时，系统会按如上给定路由规则自动解析当前url为<http://127.0.0.1:15336/Home/Index>；如果给出的url是<http://IP>:端口/A/B/C时，系统会自动解析到控制器A的B动作，并向名字为B的Action传递参数C。需要注意的是，这里给出的默认值是在没有给定A/B/C中任何一个时才会生效，如果想要在url给定{controller}/{action}/{id}的第一个或前两个时，未给出的都有默认值，那就必须要实现自定义HttpModule。

### 插件中定义路由

nop采用的是插件机制，在插件中也可针对当前插件制定路由规则，以防止插件被全局路由规则处理成其他url。插件中的路由是在RouteProvider.cs中指定：

routes.MapRoute("Plugin.DiscountRules.CustomerRoles.Configure",

"Plugins/DiscountRulesCustomerRoles/Configure",

new { controller = "DiscountRulesCustomerRoles", action = "Configure" },

new[] { "Nop.Plugin.DiscountRules.CustomerRoles.Controllers" }

);

nop.Web可根据需要在presentation\nop.web\infrastructure\RouteProvider.cs文件中注册路由。

### 自定义路由处理

1. 指定路由规则

RouteTable.Routes.Add("AqRoute",new Route("{A}/{B}/{C}"

,new RouteValueDictionary{ {"A","AAA"}, {"B","BBB"}, {"C","CCC"} }

,new AqRouteHandler()));

如上指定了路由模板、默认值及当前路由模板的路由处理器AqRouteHandler（自定义的RouteHandler）

1. 自定义RouteHandler

class AqRouteHandler:IRouteHandler

{

public IHttpHandler GetHttpHandler(RequestContext requestContext)

{

return new AqHttpHandler(requestContext);

}

}

1. 自定义HttpHandler

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Assembly : Zaxx.Abc.RouteHandlerAbc

// Author : 刘安全

// Created : 09-07-2015

//

// Last Modified By : 刘安全

// Last Modified On : 2015-09-07 10:03:26

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// <copyright file="AqHttpHandler.cs" company="Microsoft">

// Copyright © Microsoft 2015

// </copyright>

// <summary></summary>

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Web;

using System.Web.Routing;

namespace Zaxx.Abc.RouteHandlerAbc

{

/// <summary>

/// Class AqHttpHandler.

/// <para>刘安全 2015-09-07 10:03:26</para>

/// </summary>

class AqHttpHandler : IHttpHandler

{

/// <summary>

/// Initializes a new instance of the <see cref="AqHttpHandler"/> class.

/// <para>刘安全 2015-09-07 10:08:14</para>

/// </summary>

/// <param name="requestContext">The request context.</param>

public AqHttpHandler(RequestContext requestContext)

{

ProcessRequest(requestContext);

}

#region IHttpHandler成员

/// <summary>

/// Processes the request.

/// <para>刘安全 2015-09-07 10:08:14</para>

/// </summary>

/// <param name="context">The context.</param>

/// <exception cref="NotImplementedException"></exception>

public void ProcessRequest(HttpContext context)

{

}

/// <summary>

/// Gets a value indicating whether this instance is reusable.

/// <para>刘安全 2015-09-07 10:08:14</para>

/// </summary>

/// <value><c>true</c> if this instance is reusable; otherwise, <c>false</c>.</value>

public bool IsReusable { get; }

#endregion

public void ProcessRequest(RequestContext requestContext)

{

HttpResponseBase response = requestContext.HttpContext.Response;

var datas = requestContext.RouteData.Values;

/\* 运行可在页面上看到url有路由模板{A}/{B}/{C}中的对应部分时，key对应的value就是url中给出的，否则就是默认值

示例：

url A B C

http://ip:端口 AAA BBB CCC

http://ip:端口/Aq Aq BBB CCC

http://ip:端口/8/30 8 30 CCC

http://ip:端口/2015/8/30 2015 8 30 \*/

foreach (KeyValuePair<string, object> keyValuePair in datas)

{

response.Write($"Key:{keyValuePair.Key}, Value:{keyValuePair.Value} ");

}

response.End();

}

}

}

# 权限

用户访问Nop页面时，如果没有相应的权限，是无法正常访问的。此时通常看到的是转到登录页面。

匿名用户登录时，系统会自动添加一个用户信息

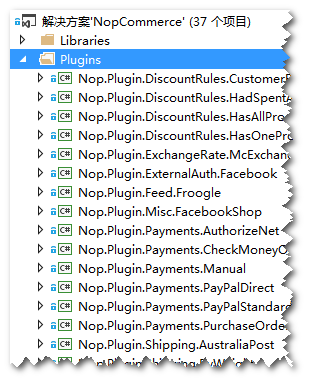
# 插件

插件是用来扩展NopCommerce功能的。NopCommerce拥有多种类型的插件。例如：支付方式（PayPal），税务机构，送货方式计算方法（UPS, USP, FedEx），小部件（如“在线聊天”块）等等。 nopCommerce本身也自带了很多不同的插件。还可以在nopCommerce官方网站搜索各种符合您要求的插件。如果没有，本章将引导完成创建自己的插件的过程。

## 新建插件类库

### 插件项目位置

创建自己的插件，第一件事情就是在解决方案中创建一个新的“类库”项目。将所有的插件放在解决方案的根目录下的\ Plugins目录中，这是一个很好的做法（不要与Nop.Web项目中的\ plugins子目录混淆了，因为这里是放置已发布生成的插件DLL文件的地方）。

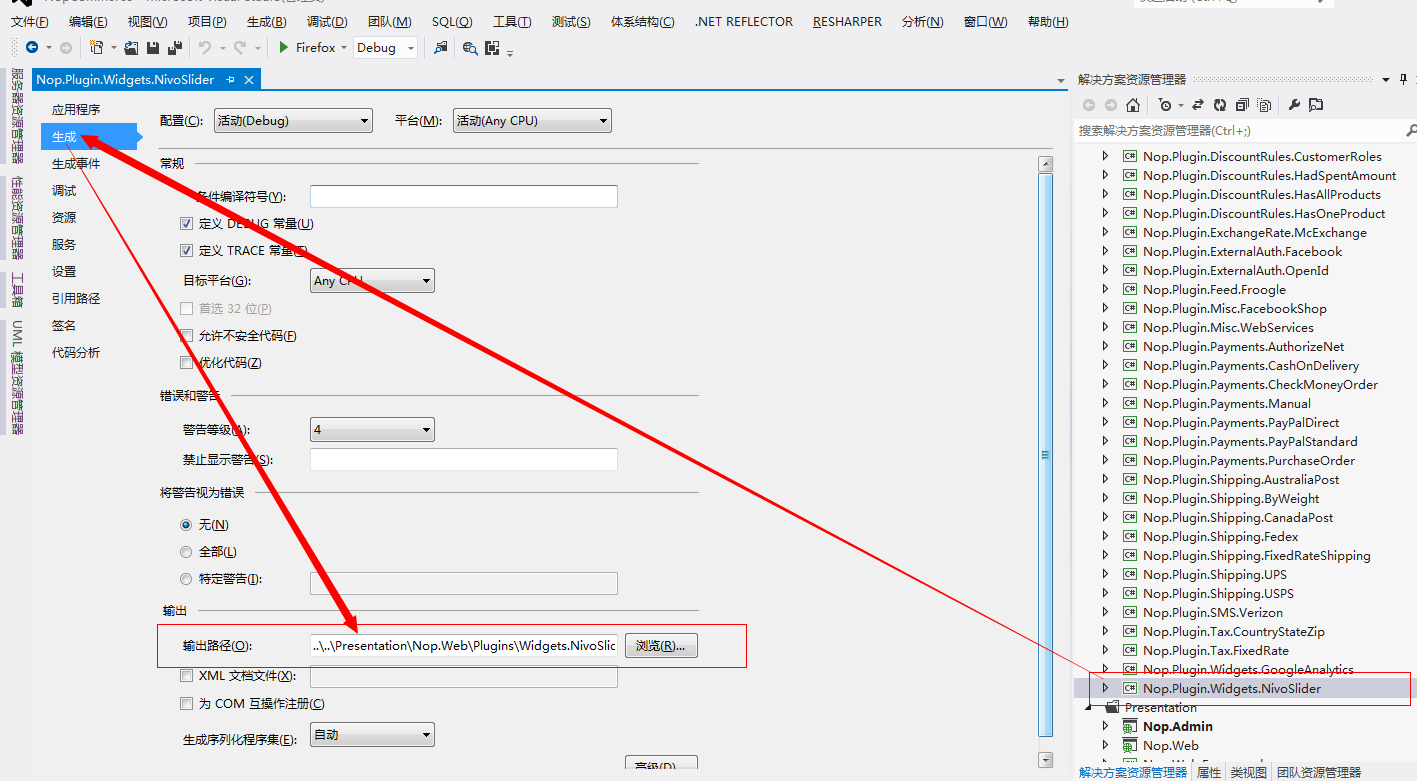


### 插件项目名称

最好以这种方法来命名：”Nop.Plugin.{Group}.{Name}”。{Group}是插件的分类（比如支付），{Name}是插件名 （比如”AuthorizeNet”），那么Authorize.NET的支付插件就会有这样的名 字：Nop.Plugin.Payments.AuthorizeNet。

## 设置项目编译文件DLL的输出位置

创建插件项目之后，需要设置插件项目编译文件DLL的输出位置为Nop.Web项目中的\ plugins子目录中："..\..\Presentation\Nop.Web\Plugins\{Group}.{Name}\"。比如 Authorize.NET 支付插件就会有这样的输入路径: “..\..\Presentation\Nop.Web\Plugins\Payments.AuthorizeNet\”。设置完成以后，对应的插件 DLL就会编译生成输出到 \Presentation\Nop.Web\Plugins\ 文件夹，nopCommerce内核会搜索此文件夹。



## 建立插件描述文件

为每一个插件建立一个Description.txt，此文件包含描述插件的信息。可以从其它插件目录中拷出来。比如图片切换插件Nop.Plugin.Widgets.NivoSlider插件的Description.txt就有如下内容：

Group: Widgets

FriendlyName: Nivo Slider

SystemName: Widgets.NivoSlider

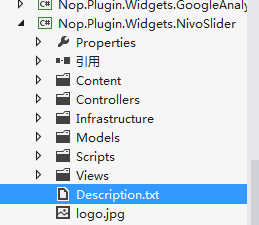
Version: 1.09

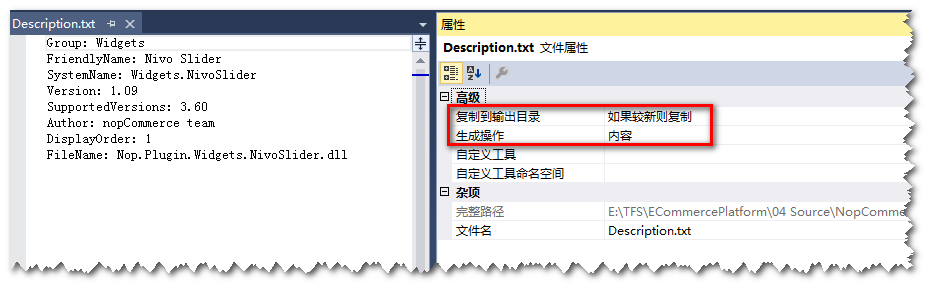
SupportedVersions: 3.60

Author: nopCommerce team

DisplayOrder: 1

FileName: Nop.Plugin.Widgets.NivoSlider.dll





## 实现接口Iplugin

最后一步就是创建一个实现接口的IPlugin （ Nop.Core.Plugins命名空间）的类。 NopCommerce的BasePlugin类已经实现了一些IPlugin方法，可以不需要再实现一遍了，这样就避免了功能代码重复。NopCommerce还提供一些从IPlugin派生的特定的接口。例如，用于创建新的付款方式插件“ IPaymentMethod ”接口。它包含了一些特定只有具体的付款功能的方法，如【支付流程】ProcessPayment()或【获取附件手续费】GetAdditionalHandlingFee () 。目前NopCommerce具有以下特定的插件接口：

* **IExternalAuthenticationMethod**:用于创建外部认证方法，如Facebook，Twitter的， OpenID的，等等
* **IWidgetPlugin**: 它允许你创建小部件。 Widget是呈现在您的网站的某些部分。例如，它可以在您的网站的左侧列中显示“在线聊天”块。
* **IExchangeRateProvider**：用于获取货币汇率。
* **IDiscountRequirementRule**:允许你创建新的折扣规则，如”帐单寄到的国家必须是……“
* **IPaymentMethod**:是用于支付处理的插件接口。
* **IShippingRateComputationMethod**：是用于获取可用的配送方法和正确的运费。例如，UPS，UPS，FEDEX等。
* **ITaxProvider**：税务提供商用于获取税率

如果这些接口不适合要创建的插件，可以使用“IMiscPlugin”接口。

**注意事项**：完成项目生成后，请清理解决方案，因为有些资源被缓存，在开发者继续编码的时候，可能出问题。

现在可以在Admin area > Configuration > Plugins看到新创建的这个插件了。但目前为止，这个插件什么都不做。甚至不需要为它配置用户界面。

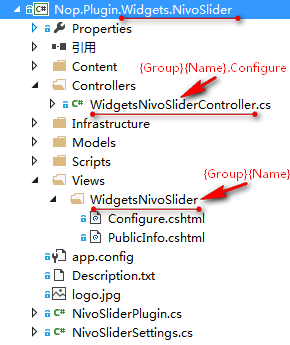
## 创建插件配置页面

创建一个插件配置页面。需要做的就是创建一个控制器、模型和视图。

* **控制器**负责响应一个ASP.NET MVC网站提出的请求。每个浏览器请求会映射到特定的控制器。
* **视图**包含被发送到浏览器的HTML标记和内容。一个视图是相当于一个ASP.NET MVC应用程序的页面。
* **模型**包含视图或控制器以外的所有应用程序逻辑。

现在可以开始插件的编写了：

* **创建模型**： 在新插件项目中添加一个名称为"Models"的文件夹, 然后在文件夹中添加一个你需要的模型类.
* **创建视图**： 在新插件项目中添加一个名称为"Views"的文件夹, 然后在文件夹中添加一个**{Group}{Name}**文件夹，此处{Group}{Name}是指插件名。然后再添加一个Configure.cshtml文件。注意：此视图应该要注明是嵌入资源。
* **创建控制器**： 在新插件项目中添加一个名称为"Controllers"的文件夹, 然后在文件夹中添加一个控制器类. 最好的命名办法是 **{Group}{Name}Controller**.cs. 例如, PaymentAuthorizeNetController. 然后在控制器中创建一个叫做"Configure"方法用于配置. 准备一个模型类并将其传给这个视图: "Nop.Plugin.{Group}.{Name}.Views. {Group}{Name}.Configure" (即那个嵌入视图). 例如, 查看在Authorize.NET支付插件中的PaymentAuthorizeNetController实现，就会比较清楚了.

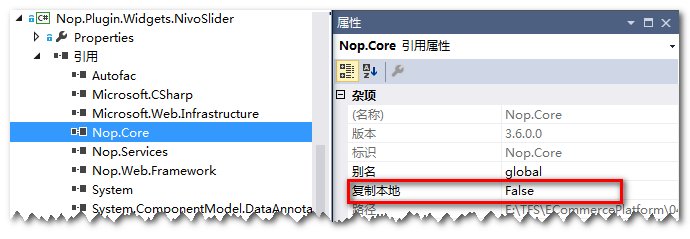


提示1：打开其他任何插件，并把其中的web.config复制到你的插件项目。该文件允许您在做视图的时候有智能感知。智能感知是微软实现自动完成的。

提示2：完成上述步骤最简单的方法就是打开其他任何插件，把它的文件拷贝到您的插件项目。然后，只需重命名类和文件夹就OK了。

提示3：如果你想限制只能是管理员（店主）访问控制器的操作方法，那么就用[AdminAuthorize]属性标记这个方法就OK了。

提示4：最后，确保所有第三方程序集引用的“复制本地”属性设置为“False”（不要复制）。这将减少部署的文件的大小。



## 注册插件路由

现在来注册插件的相应路由。ASP.NET路由用于把浏览器发送的请求映射成MVC控制器相应的action方法。

按照下面的步骤注册：

1. 一些特写的插件接口（如上所述）和“IMiscPlugin”接口有一个“GetConfigurationRoute”方法。它应该向控制器返回一个用于插件后台配置的路由。实现新建插件的“GetConfigurationRoute”方法，可以告知nopCommerce新建插件的后台路由配置是什么。如果新建插件没有后台配置，那么此方法将返回NULL，比如下边这样：

public void GetConfigurationRoute(out string actionName, out string controllerName, out RouteValueDictionary routeValues)

{

actionName = "Configure";

controllerName = "WidgetsNivoSlider";

routeValues = new RouteValueDictionary { { "Namespaces", "Nop.Plugin.Widgets.NivoSlider.Controllers" }, { "area", null } };

}

1. （可选）如果需要添加一些自定义的路由，可以创建RouteProvider.cs文件。它通知nopCommerce系统关于插件的路由。例如，下面的RouteProvider类添加一个新的路由，可以通过打开Web浏览器并导航到http://www.yourStore.com/Plugins/PaymentPayPalStandard/PDTHandler网址（使用PayPal插件）来访问一个新的路由：

public partial class RouteProvider : IRouteProvider

{

public void RegisterRoutes(RouteCollection routes)

{

//PDT

routes.MapRoute("Plugin.Payments.PayPalStandard.PDTHandler",

"Plugins/PaymentPayPalStandard/PDTHandler",

new { controller = "PaymentPayPalStandard", action = "PDTHandler" },

new[] { "Nop.Plugin.Payments.PayPalStandard.Controllers" }

);

}

public int Priority

{

get

{

return 0;

}

}

}

插件安装了以后，且添加了配置方法，就能在Admin > Configuration > Plugins找到一个配置链接。

## 处理”安装“和”卸载“方法

这一步是可选的。有些插件在安装过程中需要额外的逻辑。例如，一个插件可以添加本地资源。新建一个实现IPlugin接口的类（在大多数情况下，是从BasePlugin类派生），并重写下面的方法：

**Install**：此方法将插件安装过程中被调用。可以在这里初始化任何设置，添加本地资源，或者创建一些新的数据库表（如果有必要）。

**Uninstall**：这个方法将取消插件的调用.

**注意**：如果重写这些方法得其中一个，需要调用基方法而不能隐藏它。比如，重写“Install”方法，需要调用后面的方法“Base.Install()", Authorize.NET插件的“Install”的方法看起来像下面的代码:

public override void Install()

{

//settings

var settings = new AuthorizeNetPaymentSettings

{

UseSandbox = true,

TransactMode = TransactMode.Authorize,

TransactionKey = "123",

LoginId = "456"

};

\_settingService.SaveSetting(settings);

//locales

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Notes", "If you're using this gateway, ensure that your primary store currency is supported by Authorize.NET.");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.UseSandbox", "Use Sandbox");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.UseSandbox.Hint", "Check to enable Sandbox (testing environment).");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.TransactModeValues", "Transaction mode");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.TransactModeValues.Hint", "Choose transaction mode");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.TransactionKey", "Transaction key");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.TransactionKey.Hint", "Specify transaction key");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.LoginId", "Login ID");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.LoginId.Hint", "Specify login identifier.");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.AdditionalFee", "Additional fee");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.AdditionalFee.Hint", "Enter additional fee to charge your customers.");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.AdditionalFeePercentage", "Additional fee. Use percentage");

this.AddOrUpdatePluginLocaleResource("Plugins.Payments.AuthorizeNet.Fields.AdditionalFeePercentage.Hint", "Determines whether to apply a percentage additional fee to the order total. If not enabled, a fixed value is used.");

base.Install();

}

**备注**：已安装插件可以在”\ App\_Data\ InstalledPlugins.txt“中找到。这份清单是在安装过程中创建的（即在Admin > Configuration > Plugins > Local plugins中安装的）。

## 升级nopCommerce可能会让插件无法工作

nopCommerce升级后，有些插件可能会不兼容nopCommerce新版本。如果在升级到较新版本后有问题，删除插件，然后在nopCommerce官方网站，查看是否有兼容nopCommerce新的版本的插件。大部分插件开发者都会将升级他们的插件，以适应新的版本，但是，有些插件不会升级从而不兼容nopCommerce的新版本。但在大多数情况下，可以打开相应的 Description.txt文件并编辑SupportedVersions字段。

# 为管理平台添加功能：产品特有属性

## 功能需求

同一类型的产品，应有一些相同的属性。这些属性可以提取放在产品类型中去定义，这些属性即产品类型的特有属性。在添加某种类型的产品时，这些特有属性可自动带出。

考虑属性名称的多语言性，需先实现一个“产品属性”，并在其中实现多语言；再实现“产品特有属性”，实现这些属性是可以是哪些类型（文本框、多行文本域、下拉框还是……）、是否为必填项、是否为过滤项……

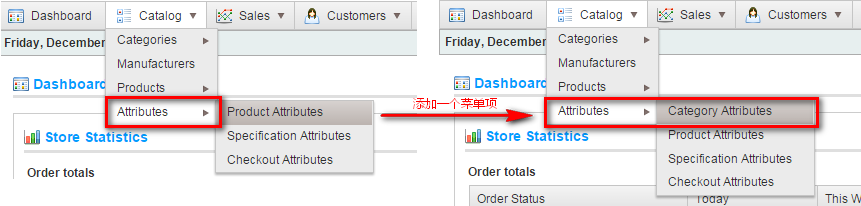
## 功能实现

以下是按照阅读的方式从上而下的来给出新增功能时的实现。在实际工作中，应按如下步骤实现新增功能：

* 添加并实现：实体类
* 添加并实现：实体映射类
* 添加并实现：实体对应的操作接口及服务
* 注册接口与服务的关系
* 添加并实现：控制器、动作、视图
* 添加资源信息
* 配置菜单

### 添加菜单项“产品属性”

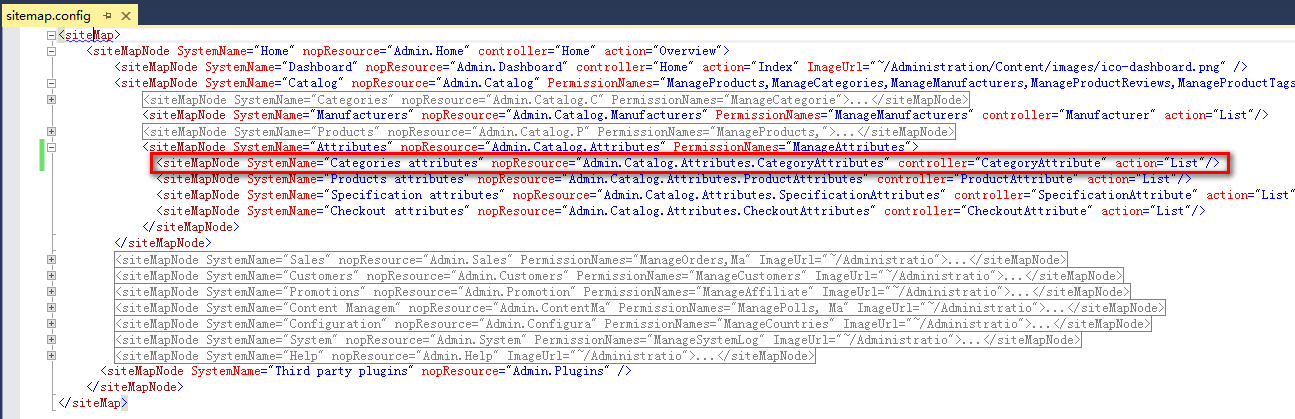
在Catalog -> Attribute 菜单下添加一个子菜单Category Attribute。管理平台菜单变化如下：



具体实现是在Presentation\Nop.Admin\sitemap.config中添加如下代码：

<siteMapNode SystemName="Categories attributes" nopResource="Admin.Catalog.Attributes.CategoryAttributes" controller="CategoryAttribute" action="List"/>

代码在sitemap.config中具体添加位置如图：



### 添加菜单项默认语言资源

为2.1所添加的菜单项Category Attributes所用到的语言资源nopResource="Admin.Catalog.Attributes.CategoryAttributes"，在App\_Data\Localization\ defaultResources.nopres.xml中添加对应资源：

<LocaleResource Name="Admin.Catalog.Attributes.CategoryAttributes">

<Value>Category Attributes</Value>

</LocaleResource>

### 添加访问菜单项的权限

为了能够访问前面所添加的菜单项“产品属性”，还需要为用户添加访问这个菜单项的权限。在Nop.Services\Security\StandardPermissionProvider.cs中（注意：如果框架结构改变，则Security\StandardPermissionProvider.cs可能不在Nop.Services中），添加如下3部分代码：

添加常量ManageCodes：

public static readonly PermissionRecord ManageCodes = new PermissionRecord {Name = "Admin area. Manage Codes", SystemName = "ManageCodes", Category = "Code"};

在GetPermissions中添加：

public virtual IEnumerable<PermissionRecord> GetPermissions()

{

return new[]

{

…,

ManageCodes,

…

}

}

在GetDefaultPermissions中添加：

public virtual IEnumerable<DefaultPermissionRecord> GetDefaultPermissions()

{

return new[]

{

new DefaultPermissionRecord

{

CustomerRoleSystemName = SystemCustomerRoleNames.Administrators,

PermissionRecords = new[]

{

…,

ManageCodes,

…

}

}

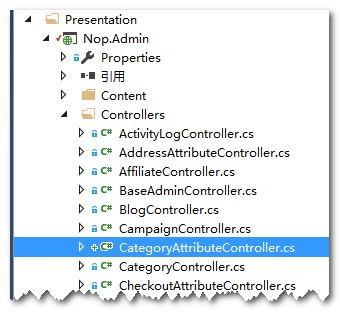
}

}

### 添加Control及Action

为2.1所添加的菜单项Category Attributes所用到的controller="CategoryAttribute" action="List"，在Presentation\Nop.Admin添加对应的控制器及动作：

添加控制器后，项目文件结构图如下：



控制器代码如下：

using System.Web.Mvc;

using Nop.Services.Security;

using Nop.Web.Framework.Kendoui;

namespace Nop.Admin.Controllers

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作　　者：刘安全

\* 创建时间：2015年12月11日 14:12:29

\* 功能描述：

\*

\* 更改历史：

\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// 类CategoryAttributeController：

/// <para> 刘安全 2015年12月11日 14:12:29</para>

/// </summary>

public class CategoryAttributeController : BaseAdminController

{

#region Field

private readonly IPermissionService \_permissionService;

private readonly ICategoryAttributeService \_categoryAttributeService;

#endregion

#region Ctor

public CategoryAttributeController(IPermissionService permissionService, ICategoryAttributeService categoryAttributeService)

{

this.\_permissionService = permissionService;

this.\_categoryAttributeService = categoryAttributeService;

}

#endregion

#region Attribute List

public ActionResult List()

{

if (!\_permissionService.Authorize(StandardPermissionProvider.ManageAttributes))

return AccessDeniedView();

return View();

}

[HttpPost]

public ActionResult List(DataSourceRequest command)

{

if (!\_permissionService.Authorize(StandardPermissionProvider.ManageAttributes))

return AccessDeniedView();

var categoryAttributes = \_categoryAttributeService.GetAllCategoryAttributes(command.Page - 1, command.PageSize);

var gridModel = new DataSourceResult

{

Data = categoryAttributes.Select(x => x.ToModel()),

Total = categoryAttributes.TotalCount

};

return Json(gridModel);

}

#endregion

}

}

### 注册接口及服务

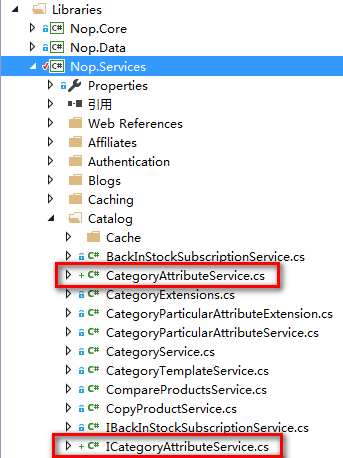
为2.3控制器中所用到的接口ICategoryAttributeService，在Presentation\Nop.Web.Framework\DependencyRegistrar.cs文件中配置接口ICategoryAttributeService与服务CategoryAttributeService的映射关系：

builder.RegisterType<CategoryAttributeService>().As<ICategoryAttributeService>().InstancePerLifetimeScope();

### 实现接口及服务

为2.3控制器中所用到的接口ICategoryAttributeService，在Libraries\Nop.Services\Catalog中添加并实现接口ICategoryAttributeService与服务CategoryAttributeService。

添加接口与服务后，项目文件结构图如下：



接口代码为：

using Nop.Core;

namespace Nop.Services.Catalog

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作　　者：刘安全

\* 创建时间：2015年12月11日 14:40:26

\* 功能描述：

\*

\* 更改历史：

\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// 接口ICategoryAttributeService：

/// <para> 刘安全 2015年12月11日 14:40:26</para>

/// </summary>

public interface ICategoryAttributeService

{

/// <summary>

/// Gets all category attributes.

/// <para>刘安全 2015-12-11 15:29:27</para>

/// </summary>

/// <param name="pageIndex">Index of the page.</param>

/// <param name="pageSize">Size of the page.</param>

/// <returns>IPagedList&lt;CategoryAttribute&gt;.</returns>

IPagedList<CategoryAttribute> GetAllCategoryAttributes(int pageIndex = 0, int pageSize = int.MaxValue);

}

}

服务代码为：

using System;

using System.Linq;

using Nop.Core;

using Nop.Core.Caching;

using Nop.Core.Data;

namespace Nop.Services.Catalog

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作　　者：刘安全

\* 创建时间：2015年12月11日 14:40:46

\* 功能描述：

\*

\* 更改历史：

\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// 类CategoryAttributeService：

/// <para> 刘安全 2015年12月11日 14:40:46</para>

/// </summary>

public class CategoryAttributeService : ICategoryAttributeService

{

#region Fields

private readonly IRepository<CategoryAttribute> \_categoryAttributeRepository;

private readonly ICacheManager \_cacheManager;

#endregion

#region Ctor

public CategoryAttributeService(IRepository<CategoryAttribute> categoryAttributeRepository, ICacheManager cacheManager)

{

\_categoryAttributeRepository = categoryAttributeRepository;

\_cacheManager = cacheManager;

}

#endregion

#region Constants

/// <summary>

/// Key for caching

/// </summary>

/// <remarks>

/// {0} : page index

/// {1} : page size

/// </remarks>

private const string CategoryAttributesAllKey = "Nop.CategoryAttribute.all-{0}-{1}";

#endregion

#region Methods

public IPagedList<CategoryAttribute> GetAllCategoryAttributes(int pageIndex = 0, int pageSize = Int32.MaxValue)

{

string key = string.Format(CategoryAttributesAllKey, pageIndex, pageSize);

return \_cacheManager.Get(key, () =>

{

var query = from pa in (IQueryable<RsWebServiceBll.NopWDSReference.CategoryAttribute>)\_categoryAttributeRepository.Table

orderby pa.Name

select pa;

var productAttributes = new PagedList<CategoryAttribute>(query, pageIndex, pageSize);

return productAttributes;

});

}

#endregion

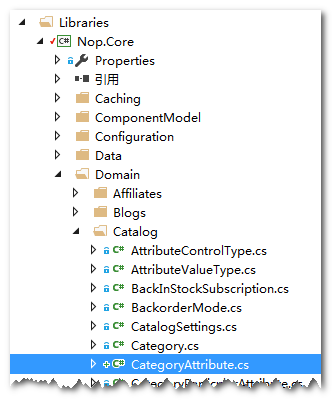
}

}

### 实现实体类

为2.5接口ICategoryAttributeService中所用到的实体CategoryAttribute，在Libraries\Nop.Core\Domain\Catalog中添加并实现实体类CategoryAttribute。

添加实体类后，Nop.Core项目文件结构图如下：



CategoryAttribute代码为：

namespace Nop.Core.Domain.Catalog

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作　　者：刘安全

\* 创建时间：2015年12月11日 15:42:03

\* 功能描述：

\*

\* 更改历史：

\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// 类CategoryAttribute：

/// <para> 刘安全 2015年12月11日 15:42:03</para>

/// </summary>

public class CategoryAttribute: BaseEntity, ILocalizedEntity

{

/// <summary>

/// Gets or sets the name.

/// <para>刘安全 2015-12-11 15:42:05</para>

/// </summary>

/// <value>The name.</value>

public string Name { get; set; }

/// <summary>

/// Gets or sets the description.

/// <para>刘安全 2015-12-11 15:42:05</para>

/// </summary>

/// <value>The description.</value>

public string Description { get; set; }

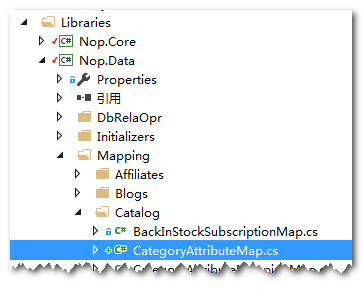
}

}

### 实现实体类的数据库映射类

在Libraries\Nop.Data\Mapping\Catalog中添加并实现2.6实体类CategoryAttribute对应的映射类CategoryAttributeMap。

添加映射类后，Nop.Data项目文件结构图如下：



CategoryAttributeMap的代码为：

using Nop.Core.Domain.Catalog;

namespace Nop.Data.Mapping.Catalog

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 作　　者：刘安全

\* 创建时间：2015年12月11日 15:48:30

\* 功能描述：

\*

\* 更改历史：

\*

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/// <summary>

/// 类CategoryAttributeMap：

/// <para> 刘安全 2015年12月11日 15:48:30</para>

/// </summary>

public class CategoryAttributeMap : NopEntityTypeConfiguration<CategoryAttribute>

{

public CategoryAttributeMap()

{

this.ToTable("CategoryAttribute");

this.HasKey(ca => ca.Id);

this.Property(ca => ca.Name).IsRequired();

}

}

}

### 实现实体与模型的转换

为2.3代码中的x.ToModel()，在Presentation\Nop.Admin\MappingExtensions中实现实体对象与模型对象的转换：

#region Category attributes

// 实现实体对象转模型对象：以给定实体信息，返回一个对应的新建模型对象

public static CategoryAttributeModel ToModel(this CategoryAttribute entity)

{

return entity.MapTo<CategoryAttribute, CategoryAttributeModel>();

}

// 实现模型对象转实体对象：以给定模型信息，返回一个对应的新建实体对象

public static CategoryAttribute ToEntity(this CategoryAttributeModel model)

{

return model.MapTo<CategoryAttributeModel, CategoryAttribute>();

}

// 实现模型对象转实体对象：以给定模型信息，更新并返回给定的实体对象

public static CategoryAttribute ToEntity(this CategoryAttributeModel model, CategoryAttribute destination)

{

return model.MapTo(destination);

}

#endregion

### 注册实体与模型的映射关系

在Infrastructure\AutoMapperStartupTask中，注册实体与模型之间的映射关系：

Mapper.CreateMap<CategoryAttribute, CategoryAttributeModel>();

Mapper.CreateMap<CategoryAttributeModel, CategoryAttribute>();

### 新建模型类及本地化模型类

为2.8所用到的CategoryAttributeModel，在中新建模型类CategoryAttributeModel及本地化模型类CategoryAttributeLocalizedModel。

CategoryAttributeModel中给出记录类型属性的名称及描述，代码为：

public class CategoryAttributeModel : BaseNopEntityModel, ILocalizedModel<CategoryAttributeLocalizedModel>

{

public CategoryAttributeModel()

{

Locales = new List<CategoryAttributeLocalizedModel>();

}

[NopResourceDisplayName("Admin.Catalog.Attributes.CategoryAttributes.Fields.Name")]

[AllowHtml]

public string Name { get; set; }

[NopResourceDisplayName("Admin.Catalog.Attributes.CategoryAttributes.Fields.Description")]

[AllowHtml]

public string Description { get; set; }

public IList<CategoryAttributeLocalizedModel> Locales { get; set; }

}

CategoryAttributeLocalizedModel中给出类型属性需要多语言处理的的名称及描述，代码为：

public class CategoryAttributeLocalizedModel : ILocalizedModelLocal

{

public int LanguageId { get; set; }

[NopResourceDisplayName("Admin.Catalog.Attributes.CategoryAttributes.Fields.Name")]

[AllowHtml]

public string Name { get; set; }

[NopResourceDisplayName("Admin.Catalog.Attributes.CategoryAttributes.Fields.Description")]

[AllowHtml]

public string Description { get; set; }

}

# 联系信息

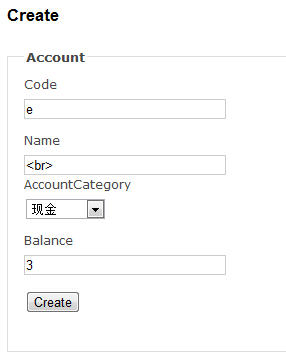
|  |  |  |
| --- | --- | --- |
| 武汉-Nc |  |  |
| QQ: 94027486 |  |  |
| https://gqrcode.alicdn.com/img?type=cs&shop_id=59005853&seller_id=262883289&w=140&h=140&el=q&v=1 |  |  |
| <https://wzdx.taobao.com/> |  |  |

因赶工作进度，后续文档有空再整理。

# 参考资料

## ASP.NET MVC中使用AllowHtml属性

创建一个包含类似有输入页面的ASP.NET MVC程序，输入框中需要可以输入HTML标记，假如没有使用某种方式禁用Request Validation,那么如果从客户端检测到有潜在危险的Request.Form值，例如Name输入框输入了<br>，



点击Create，则会出现如下错误提示：



Request Validation是防止有人通过注入脚本攻击我们的应用程序(XSS)。如果我们的应用程序需要用户可以输入HTML标记，我们可以禁用Request Validation.

ASP.NET MVC2中可以使用Action方法的ValidateInpute属性禁用Model的所有字段的验证，如果只是想禁用某个字段的验证，则可以使用MVC3中的AllowHtml属性实现。

public class Account  
{  
    [AllowHtml]  
    public string Name{get;set;}  
}

## @RenderBody、@RenderSection、@RenderPage、Html.RenderPartial、Html.RenderAction的作用和区别

### @RenderBody

在Razor引擎中没有了“母版页”，取而代之的是叫做“布局”的页面（\_Layout.cshtml）放在了共享视图文件夹中。在这个页面中，会看到标签里有这样一条语句：

@RenderBody()

其实它的作用和母版页中的服务器控件类似，当创建基于此布局页面的视图时，视图的内容会和布局页面合并，而新创建视图的内容会通过布局页面的@RenderBody()方法呈现在标签之间。

这个方法不需要参数，而且只能出现一次。

### @RenderPage

从名称可以猜出来这个方法是要呈现一个页面。比如网页中固定的头部可以单独放在一个共享的视图文件中，然后在布局页面中通过这个方法调用，用法如下：

@RenderPage(“~/Views/Shared/\_Header.cshtml”)

带参数

@RenderPage(“~/Views/Shared/\_Header.cshtml”,new{parm="my",parm2="you")

调用页面获取参数：

//获取 RenderPage() 传递过来的参数

@PageData["param"]

### @RenderSection

布局页面还有节（Section）的概念，也就是说，如果某个视图模板中定义了一个节，那么可以把它单独呈现出来，用法如下：

@RenderPage(“~/Views/Shared/\_Header.cshtml”)

@RenderBody()

//模板里添加了一个节

@RenderSection(“head”)

当然还要在视图中定义节，否则会出现异常：

@section head{

//do

}

为了防止因缺少节而出现异常，可以给RenderSection()提供第2个参数：

@RenderSection("SubMenu", false)

或

@if (IsSectionDefined("SubMenu"))

{

@RenderSection("SubMenu", false)

}

else

{

<p>SubMenu Section is not defined!</p>

}

### @Html.Partial

　Partial　每次都会创建自己的 TextWriter 实例并且把内容缓存在内存中. 最后把所有 writer输出的内容发送到一个 MvcString对象中

更多时候我们会使用 @{ Html.RenderPartial("Details"); } 而不是@Html.Partial

RenderPage()和RenderPartial()的区别：RenderPage()调用的页面只能使用其传递过去的数据。而RenderPartial()是可以使用viewdata,model等数据的。

Html.RenderPartial和Html.RenderAction的区别：

Html.RenderPartial适合用在重覆使用的UserControl，并且只需要透过Model来呈现内容，或是对于广告的UserControl也适合使用。 Html.RenderAction则会先去呼叫Controller的Action方法，如果此UserControl是需要透过资料库取得资料来呈现（透过Action来读取资料库），此时会比较适合使用此方式。

## ASP.NET MVC：自定义Route让Url更优雅

### 目标

如今，互联网越来越注重简单优雅的Url，对比下面两个：

~/Products/UpdateUnitPrice/5

~/products/update-unit-price/5

我相信大多数朋友会更喜欢第二种方式：小写，使用负（减）号作为连字符。

本文使用自定义 Route 来达到方式二的效果，只需增加几个类和简单修改下 global.asax 文件。

### 基础知识：Route 是双向的

Route 的基本概念在此不多说，这里重点强调一下 ASP.NET MVC 中 Route 是双向的，这可以从 RouteBase 类的定义可以看出：

public abstract class RouteBase{

public abstract RouteData GetRouteData(HttpContextBase httpContext);

public abstract VirtualPathData GetVirtualPath(RequestContext requestContext, RouteValueDictionary values);

}

GetRouteData 用于解析 Url，GetVirtualPath 用于生成 Url。

### 自定义Route类

在开始编写自定义的 Route 类之前，先定义一个将用在自定义的Route中的用于处理字符串的静态类：StringElegantHelper 类。StringElegantHelper 中有两个方法 Elegant 和 DeElegant，将分别用在 GetVirtualPath 和 GetRouteData 方法中，用于生成和解析优雅的 Url。

其代码为：

internal static class StringElegantHelper {

public static readonly char minus = '-';

public static string Elegant(string s) {

var builder = new StringBuilder();

var index = 0;

foreach (var c in s) {

if (c >= 'A' && c <= 'Z') {

if (index > 0) builder.Append(minus);

builder.Append(char.ToLower(c));

}

else if (c == minus) {

builder.Append(minus);

builder.Append(minus);

}

else

builder.Append(c);

index++;

}

return builder.ToString();

}

public static string DeElegant(string s) {

var builder = new StringBuilder();

var iterator = s.GetEnumerator();

while (iterator.MoveNext()) {

if (iterator.Current != minus) {

builder.Append(iterator.Current);

continue;

}

if (!iterator.MoveNext()) {

builder.Append(minus);

break;

}

if (iterator.Current == minus)

builder.Append(minus);

else

builder.Append(iterator.Current);

}

return builder.ToString();

}

}

StringElegantHelper 中有两个方法 Elegant 和 DeElegant，也可以使用正则表达式来实现，但效率不如上面这种最朴实的方式高。

有了 StringElegantHelper 类，写出 ElegantRoute 就简单多了，ElegantRoute 类的代码为：

public class ElegantRoute : Route {

public static readonly string[] ToElegant = new [] { "controller", "action" };

public ElegantRoute(string url, RouteValueDictionary defaults, RouteValueDictionary constraints, RouteValueDictionary dataTokens, IRouteHandler routeHandler)

: base(url, defaults, constraints, dataTokens, routeHandler) { }

public override RouteData GetRouteData(HttpContextBase httpContext) {

var result = base.GetRouteData(httpContext);

if (result == null) return null;

foreach (var key in ToElegant)

HandleItem(result.Values, key, StringElegantHelper.DeElegant);

return result;

}

public override VirtualPathData GetVirtualPath(RequestContext requestContext, RouteValueDictionary values) {

var elegantValues = new RouteValueDictionary(values);

foreach (var key in ToElegant)

HandleItem(elegantValues, key, StringElegantHelper.Elegant);

return base.GetVirtualPath(requestContext, elegantValues);

}

private void HandleItem(RouteValueDictionary dict, string key, Func<string, string> handler) {

if (!dict.ContainsKey(key)) return;

//

var value = dict[key];

if (!(value is string)) return;

//

dict[key] = handler(value as string);

}

}

注意，上面代码中只对 controller 和 action 的路由值进行了“优雅”处理，其它值并没有，为什么呢？

大家可以考虑以下网址：

~/customers/details/ANTON

全部处理的话会变成：

~/customers/details/a-n-t-o-n

代码其他部分比较简单，不多说了。

### 自定义Route类必要的对应扩展方法

实现自定义Route类后，还需要几个扩展方法，以方便在 global.asax 文件中使用：代码其他部分比较简单，核心代码为：

public static class ElegantRouteExtensions {

public static ElegantRoute MapElegantRoute(this RouteCollection routes, string name, string url, object defaults) {

var route = new ElegantRoute(url,

new RouteValueDictionary(defaults),

new RouteValueDictionary(), //constraints

new RouteValueDictionary(), //dataTokens

new MvcRouteHandler());

routes.Add(name, route);

return route;

}

}

这里只写出一个，可根据需要添加。

### 使用自定义Route类

借助上面的扩展方法，使用就很相当简单了。global.asax 文件中，把 routes.MapRoute 替换为 routes.MapElegantRoute 即可：

routes.MapElegantRoute( // MapRoute

"Default",

"{controller}/{action}/{id}",

new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);