[FluentValidation](http://fluentvalidation.codeplex.com/) 是 [.NET](http://www.haogongju.net/tag/.NET) 下的模型验证组件，和 [ASP.NET](http://www.haogongju.net/tag/ASP.NET) M[VC](http://www.haogongju.net/tag/VC) 基于Attribute 声明式验证的不同处，其利用表达式语法链式[编程](http://www.haogongju.net/tag/%E7%BC%96%E7%A8%8B)，使得验证组件与实体分开。正如 [FluentValidation](http://fluentvalidation.codeplex.com/) 的 介绍：

A small validation library for .[NET](http://www.haogongju.net/tag/net) that uses a fluent interface and lambda expressions for building validation rules for your business objects.

使用后，只能用一句话来形容：真乃神器也！

[项目](http://www.haogongju.net/tag/%E9%A1%B9%E7%9B%AE)地址：<http://fluentvalidation.codeplex.com/>

想体验 Lambda Expression 流畅的感觉吗，下面 let's go!

首先，你需要通过 NuGet 获取 FluentValidation、FluentValidation.[MVC](http://www.haogongju.net/tag/MVC)3 包，我当前使用的版本如下：

<?xml version="1.0" encoding="utf-8"?>

<packages>

<package id="FluentValidation" version="3.3.1.0" />

<package id="FluentValidation.MVC3" version="3.3.1.0" />

</packages>

***快速入门***

**1. 建立模型类**

为了演示，我这里建了一个 Person 类，并且假设有下面这些 Property（属性）。

/// <summary>

/// 个人

/// </summary>

public class Person

{

/// <summary>

/// 姓

/// </summary>

public string Surname { get; set; }

/// <summary>

/// 名

/// </summary>

public string Forename { get; set; }

/// <summary>

/// 公司

/// </summary>

public string Company { get; set; }

/// <summary>

/// 地址

/// </summary>

public string Address { get; set; }

/// <summary>

/// 邮政编码

/// </summary>

public string Postcode { get; set; }

/// <summary>

/// 个人空间的地址的别名，比如：bruce-liu-cnblogs、cnblogs\_bruce\_liu

/// </summary>

public string UserZoneUrl { get; set; }

}

根据 FluentValidation 的使用方法，我们直接可以在 Person 类上面直接标记对应的 Validator，比如： [Validator(typeof(PersonValidator))]。但如果我们的模型层（Model Layer）不允许修改（假设），并且你像我一样喜欢干净的模型层，不想要标记太多业务型的 Attribute 时，我们就使用继承的方式来标记，在派生类上标记。下面我们建一个 Customer 类，继承自 Person 类，并且再增加 2 个 Property（属性），最后标记 Validator Attribute。

[Validator(typeof(CustomerValidator))]

public class Customer : Person

{

/// <summary>

/// 是否有折扣

/// </summary>

public bool HasDiscount { get; set; }

/// <summary>

/// 折扣

/// </summary>

public float Discount { get; set; }

}

**2. 建立模型类相应的 FluentValidation 验证类**

public class CustomerValidator : AbstractValidator<Customer>

{

public CustomerValidator()

{

// 在这里写验证规则，比如：

// Cascade(FluentValidation.CascadeMode.StopOnFirstFailure) 可以指定当前 CustomerValidator 的验证模式，可重写全局验证模式

RuleFor(customer => customer.Surname).Cascade(FluentValidation.CascadeMode.StopOnFirstFailure).NotEmpty().Length(3, int.MaxValue).WithLocalizedName(() => "姓").WithLocalizedMessage(() => "亲，{PropertyName}不能为空字符串，并且长度大于{0}！！！");

// 更多...

// 更多...

}

}

**3. 在 Global.asax 里面的**[**App**](http://www.haogongju.net/tag/App)**lication\_Start 中配置 FluentValidation**

默认情况下，FluentValidation 使用的验证错误消息是英文的，且官方自带的语言包中没有中文，于是我自己就手动翻译，建立了一个[资源](http://www.haogongju.net/tag/%E8%B5%84%E6%BA%90)文件 FluentValidationResource.resx，并且在 Global.asax 中配置。

protected void Application\_Start()

{

ConfigureFluentValidation();

}

protected void ConfigureFluentValidation()

{

// 设置 FluentValidation 默认的资源文件提供程序 - 中文资源

ValidatorOptions.ResourceProviderType = typeof(FluentValidationResource);

/\* 比如验证用户名 not null、not empty、length(2,int.MaxValue) 时，链式验证时，如果第一个验证失败，则停止验证 \*/

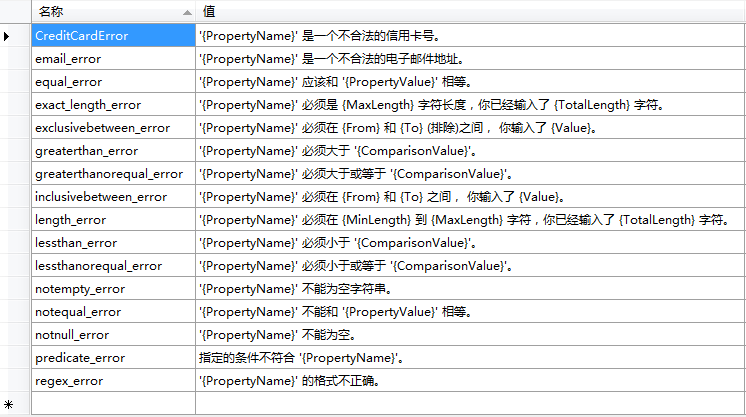
ValidatorOptions.CascadeMode = CascadeMode.StopOnFirstFailure; // ValidatorOptions.CascadeMode 默认值为：CascadeMode.Continue

// 配置 FluentValidation 模型验证为默认的 ASP.NET MVC 模型验证

FluentValidationModelValidatorProvider.Configure();

}

FluentValidationResource 代码中的 Key-Value 如下（PS：由于不知道怎么贴 Resource 文件中的代码，我就用截图了）：



翻译得不好，请多多包涵！

**4.**[**客户端**](http://www.haogongju.net/tag/%E5%AE%A2%E6%88%B7%E7%AB%AF)**调用**

本来用控制台程序就可以调用的，由于笔者建立的项目是 [ASP.NET MVC](http://www.haogongju.net/tag/ASP.NET%20MVC) 项目，本文的重点也是 FluentValidation 在 ASP.NET MVC 中使用，于是就在 Action 里面验证了。在 HomeController 的 Index 方法里面的代码如下：

public ActionResult Index()

{

/\* 下面的例子验证 FluentValidation 在 .net 中的使用，非特定与 ASP.NET MVC \*/

Customer customer = new Customer();

// 我们这里直接 new 了一个 Customer 类，看看模型验证能否通过

CustomerValidator validator = new CustomerValidator();

ValidationResult results = validator.Validate(customer);

// 或者抛出异常 validator.ValidateAndThrow(customer);

bool validationSucceeded = results.IsValid;

IList<ValidationFailure> failures = results.Errors;

StringBuilder textAppender = new StringBuilder();

if (!results.IsValid)

{

foreach (var failureItem in failures)

{

textAppender.Append("<br/>==========================================<br/>");

textAppender.AppendFormat("引起失败的属性值为：{0}<br/>", failureItem.AttemptedValue);

textAppender.AppendFormat("被关联的失败状态为：{0}<br/>", failureItem.CustomState);

textAppender.AppendFormat("错误消息为：{0}<br/>", failureItem.ErrorMessage);

textAppender.AppendFormat("Property（属性）为：{0}<br/>", failureItem.PropertyName);

textAppender.Append("<br/>==========================================<br/>");

}

}

ViewBag.Message = textAppender.ToString();

return View();

}

最后，运行就能看到效果！

***进阶篇***

**1. 属性类（Property Class）的验证**

既然是顾客，那么顾客就可能会有订单，我们建立一个 Order 类，把 Customer 类作为 Order 类的一个 Property（属性）。

/// <summary>

/// 订单

/// </summary>

[Validator(typeof(OrderValidator))]

public class Order

{

public Customer Customer { get; set; }

/// <summary>

/// 价格

/// </summary>

public decimal Price { get; set; }

}

相应的，我们还需要建立一个验证类 OrderValidator。为了共用 CustomerValidator 类，我们需要在 OrderValidator 类的构造函数中，为 Order 类的 Customer 属性指定 Validator。

/// <summary>

/// 订单验证类

/// </summary>

public class OrderValidator : AbstractValidator<Order>

{

public OrderValidator()

{

RuleFor(order => order.Price).NotNull().GreaterThanOrEqualTo(0m).WithLocalizedName(() => "价格");

// 重用 CustomerValidator

RuleFor(order => order.Customer).SetValidator(new CustomerValidator());

}

}

在 ASP.NET MVC 中使用时，在 Action 方法的参数上，可以像使用 Bind Attribute 一样：

public ActionResult AddCustomer([Bind(Include = "Company", Exclude = "Address")]Customer customer)

使用 CustomizeValidator Attribute，来指定要验证的 Property（属性）：

[HttpGet]

public ActionResult AddCustomer()

{

return View(new Customer());

}

[HttpPost]

public ActionResult AddCustomer([CustomizeValidator(Properties="Surname,Forename")] Customer customer)

{

/\*

在 Action 的参数上标记 CustomizeValidator 可以指定 Interceptor（拦截器）、Properties（要验证的属性，以逗号分隔）。

如果指定了 Properties （要验证的属性，以逗号分隔），请注意是否别的属性有客户端验证，导致客户端提交不了，而服务器端

又可以不用验证。

\*/

if (!ModelState.IsValid)

{

return View(customer);

}

return Content("验证通过");

}

由此可见，FluentValidation 真是用心良苦，这都想到了，不容易啊！

***扩展篇***

**1. 完善 CustomerValidator**

接下来，我们继续 完善 CustomerValidator ，增加更多的验证规则。

public class CustomerValidator : AbstractValidator<Customer>

{

public CustomerValidator()

{

// CascadeMode = CascadeMode.StopOnFirstFailure; 可以指定当前 CustomerValidator 的验证模式，可重写全局验证模式

RuleFor(customer => customer.Surname).Cascade(FluentValidation.CascadeMode.StopOnFirstFailure).NotEmpty().Length(3, int.MaxValue).WithLocalizedName(() => "姓").WithLocalizedMessage(() => "亲，{PropertyName}不能为空字符串，并且长度大于{0}！！！");

// 注意：调用 Cascade(FluentValidation.CascadeMode.StopOnFirstFailure) 表示当一个验证条件失败后，不再继续验证

RuleFor(customer => customer.Forename).NotEmpty().WithLocalizedName(() => "名").WithLocalizedMessage(() => "{PropertyName} 一定要不为空，Do you know ?");

RuleFor(customer => customer.Company).NotNull().WithLocalizedName(() => "公司名称").WithMessage(string.Format("{{PropertyName}} 不能 \"{0}\"，下次记住哦，{1}！", "为空", "呵呵"));

RuleFor(customer => customer.Discount).NotEqual(0).WithLocalizedName(() => "折扣").When(customer => customer.HasDiscount);

RuleFor(customer => customer.Address).Length(20, 250).WithLocalizedName(() => "地址").Matches("^[a-zA-Z]+$").WithLocalizedMessage(() => "地址的长度必须在 20 到 250 个字符之间，并且只能是英文字符！");

RuleFor(customer => customer.Postcode).Must(BeAValidPostcode).WithLocalizedName(() => "邮政编码").WithMessage("请指定一个合法的邮政编码");

// 注意：如果用了 Must 验证方法，则没有客户端验证。

Custom((customer, validationContext) =>

{

bool flag1 = customer.HasDiscount;

bool flag2 = !validationContext.IsChildContext;

return flag1 && flag2 && customer.Discount > 0 ? null : new ValidationFailure("Discount", "折扣错误", customer.Discount);

});

}

/// <summary>

/// 检查是否是合法的邮政编码

/// </summary>

/// <param name="postcode"></param>

/// <returns></returns>

private bool BeAValidPostcode(string postcode)

{

if (!string.IsNullOrEmpty(postcode) && postcode.Length == 6)

{

return true;

}

return false;

}

}

当我想要给 Customer.UserZoneUrl（个人[空间](http://www.haogongju.net/tag/%E7%A9%BA%E9%97%B4)的地址的别名） 写验证规则的时候，我发现它的验证规则可以提取出来，方便下次有类似的功能需要用到。那能不能像调用 NotNull() 、NoEmpty() 方法那样，调用我们写的 EntryName() 呢？答案：当然可以！

这样调用怎么样？

RuleFor(customer => customer.UserZoneUrl).EntryName();

其中 EntryName() 是一个扩展方法。

using FluentValidation;

public static class FluentValidatorExtensions

{

public static IRuleBuilderOptions<T, string> EntryName<T>(this IRuleBuilder<T, string> ruleBuilder)

{

return ruleBuilder.SetValidator(new EntryNameValidator());

}

}

我们看到，调用 EntryName 扩展方法其实是调用另外一个 Validator - EntryNameValidator。

public class EntryNameValidator : PropertyValidator, IRegularExpressionValidator

{

private readonly Regex regex;

const string expression = @"^[a-zA-Z0-9][\w-\_]{1,149}$";

public EntryNameValidator()

: base(() => ExtensionResource.EntryName\_Error)

{

regex = new Regex(expression, RegexOptions.IgnoreCase);

}

protected override bool IsValid(PropertyValidatorContext context)

{

if (context.PropertyValue == null) return true;

if (!regex.IsMatch((string)context.PropertyValue))

{

return false;

}

return true;

}

public string Expression

{

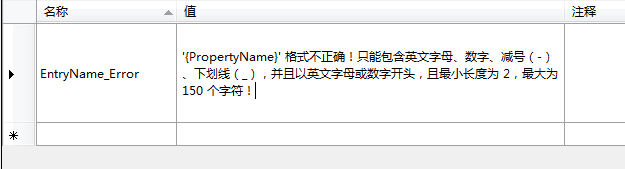
get { return expression; }

}

}

这里我们的 EntryNameValidator 除了继承自 PropertyValidator，还实现了 IRegularExpressionValidator 接口。为什么要实现 IRegularExpressionValidator 接口 呢？是因为可以共享由 FluentValidation 带来的好处，比如：客户端验证等等。

其中 ExtensionResource 是一个资源文件，我用来扩展 FluentValidation 时使用的资源文件。



**2. 复杂验证**

下面我们再建立一个 Pet（宠物）类，为 Customer 类增加一个 public List<Pet> Pets { get; set; } 属性。

/// <summary>

/// 顾客类

/// </summary>

[Validator(typeof(CustomerValidator))]

public class Customer : Person

{

/// <summary>

/// 是否有折扣

/// </summary>

public bool HasDiscount { get; set; }

/// <summary>

/// 折扣

/// </summary>

public float Discount { get; set; }

/// <summary>

/// 一个或多个宠物

/// </summary>

public List<Pet> Pets { get; set; }

}

/// <summary>

/// 宠物类

/// </summary>

public class Pet

{

public string Name { get; set; }

}

那 FluentValidation 对集合的验证，该如何验证呢？下面我们要求顾客的宠物不能超过 10 个。你一定想到了用下面的代码实现：

Custom(customer =>

{

return customer.Pets.Count >= 10

? new ValidationFailure("Pets", "不能操作 10 个元素")

: null;

});

或者我们写一个自定义的 Property（属性）验证器 ListMustContainFewerThanTenItemsValidator<T>，让它继承自 PropertyValidator

public class ListMustContainFewerThanTenItemsValidator<T> : PropertyValidator

{

public ListMustContainFewerThanTenItemsValidator()

: base("属性 {PropertyName} 不能超过 10 个元素!")

{

// 注意：这里的错误消息也可以用资源文件

}

protected override bool IsValid(PropertyValidatorContext context)

{

var list = context.PropertyValue as IList<T>;

if (list != null && list.Count >= 10)

{

return false;

}

return true;

}

}

应用这个属性验证器就很容易了，在 Customer 的构造函数中：

RuleFor(customer => customer.Pets).SetValidator(new ListMustContainFewerThanTenItemsValidator<Pet>());

再或者为了公用，写一个扩展方法，扩展 IRuleBuilder<T, IList<TElement>> 类

/// <summary>

/// 定义扩展方法，是为了方便调用。

/// </summary>

public static class MyValidatorExtensions

{

public static IRuleBuilderOptions<T, IList<TElement>> MustContainFewerThanTenItems<T, TElement>(this IRuleBuilder<T, IList<TElement>> ruleBuilder)

{

return ruleBuilder.SetValidator(new ListMustContainFewerThanTenItemsValidator<TElement>());

}

}

调用也像上面调用 EntryName() 一样，直接调用：

RuleFor(customer => customer.Pets).MustContainFewerThanTenItems();

至此，完毕！

参考自：<http://fluentvalidation.codeplex.com/>