**阅读目录**

* [1.基本介绍](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_label0)
* [2.创建验证器](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_label1)
* [3.针对相同属性的链式编程验证](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_label2)
* [4.验证结果](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_label3)
* [5.异常与复杂验证](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_label4)
* [6.灵活的验证规则组设置](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_label5)

    在文章：[这些.NET开源项目你知道吗？让.NET开源来得更加猛烈些吧!（第二辑）](http://www.cnblogs.com/asxinyu/p/dotnet_opensource_project_2.html)中，给大家初步介绍了一下[FluentValidation](https://github.com/JeremySkinner/FluentValidation)验证组件。那里只是概述了一下，并没有对其使用和强大功能做深入研究，所以今天以及接下去的几篇文章就专门介绍这个组件。不仅仅是它小，轻量级，优雅，而且一直在持续更新中。本人对这个感触很深是源于4年前自己在做一个数据过滤软件时，自己也设计了一套验证过滤的东西，虽然勉强能用，但太复杂了，复杂到我看到就想吐。。。指导我遇到了FluentValidation，彻底颠覆了我的看法，原来代码是可以很优雅的。。。这就是所谓的架构技术吧，虽然我不是很懂，但一直追求中。

    为了保持内容的完整性，大部分内容我都是参考FluentValidation提供的帮助文档，自己经过翻译和理解加工。更好的呈现给大家。

.NET开源目录：[【目录】本博客其他.NET开源项目文章目录](http://www.cnblogs.com/asxinyu/p/Bolg_Category_For_DotNet_OpenSource.html)

本文原文地址：[.NET平台开源项目速览(6)FluentValidation验证组件介绍与入门(一)](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html)

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_labelTop)

**1.基本介绍**

[FluentValidation](https://github.com/JeremySkinner/FluentValidation)是一个使用Linq表达式，非常流畅的小型业务对象验证组件。流畅也可以说优雅。类似链式操作。易于理解，功能完善。还可以配合MVC使用直接在页面进行验证，当你看到它的语法时，非常优雅，非常令人心动。不仅可以使用Linq的操作，还能自带验证返回信息。更重要的是，组件内部已经封装好了10几种验证器。当然可以自定义一个复杂的哦。核心dll文件也不大，130多k。如果好用，可以自己移植到自己的系统哦。直接更好。目前一直在更新中，主要是bug修复。

官方网站：<https://github.com/JeremySkinner/FluentValidation>

NuGet Packages：Install-Package FluentValidation

ASP.NET MVC集成包：Install-Package FluentValidation.MVC5

    下面我们将从一个简单的验证器的创建以及使用来介绍它的基本功能。

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_labelTop)

**2.创建验证器**

    使用之前，要引用FluentValidation.dll，太简单，就省略了吧。为了给特定的对象定义一组验证规则，必须创建一个继承AbstractValidator<T>的类，T是需要验证的类型。例如，假设我们有一个Customer类，如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Customer {    public int Id { get; set; }    public string Surname { get; set; }    public string Forename { get; set; }    public decimal Discount { get; set; }    public string Address { get; set; }  } |

    所以按照上面要求，我们要给Customer定义一组验证规则，就要继承AbstractValidator<Customer>，如下面代码：

|  |  |
| --- | --- |
| 1  2  3  4  5 | using FluentValidation;    public class CustomerValidator : AbstractValidator<Customer>  {  } |

    验证规则本身在验证器的构造函数中定义。为了给特定属性指定一个验证规则，需要调用RuleFor方法,通过属性的lambda表达式来进行你想要的验证。例如,确保Surname属性不是null,验证器应该这样写:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using FluentValidation;  //Customer验证器  public class CustomerValidator : AbstractValidator<Customer>  {    public CustomerValidator()    {       RuleFor(customer => customer.Surname).NotNull();    }  } |

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_labelTop)

**3.针对相同属性的链式编程验证**

    针对同一个属性编写验证代码时，我们可以使用链式的方式进行，非常方便，也容易理解。如下面的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using FluentValidation;  public class CustomerValidator : AbstractValidator<Customer>  {    public CustomerValidator()    {  //Surname不为空，且不等于foo       RuleFor(customer => customer.Surname).NotNull().NotEqual("foo");    }  } |

    上面的注释已经很明显了，针对Surname属性，直接进行判断和编写代码，同样可以写其他的，一直写下去。。。为了执行验证器，我们首先要创建一个验证器的实例对象，然后将要验证的对象传递给Validate方法，即可进行验证。如下面代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //要验证的对象实例  Customer customer = new Customer();  //验证器实例  CustomerValidator validator = new CustomerValidator();  //进行验证操作，获取验证结果  ValidationResult results = validator.Validate(customer); |

     至于结果的内容，请接着往下看。

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_labelTop)

**4.验证结果**

     在使用验证器的Validate进行验证后，获取的ValidationResult对象里面提供了2个主要信息：

1.IsValid: 标记是否验证成功

2.Errors ：错误情况，验证失败的对象集合，包括所有验证失败对象的细节。

    例如下面的代码将验证失败的信息打印出来：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Customer customer = new Customer();  CustomerValidator validator = new CustomerValidator();  ValidationResult results = validator.Validate(customer);  //如果验证失败  if(! results.IsValid) {    //遍历所有的失败对象    foreach(var failure in results.Errors) {    //失败的属性名称，如错误信息      Console.WriteLine("Property " + failure.PropertyName + " failed validation. Error was: " + failure.ErrorMessage);    }  } |

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_labelTop)

**5.异常与复杂验证**

**5.1 如何抛出异常**

    上一节中，我们看到了验证结果的处理情况，但是在很多情况下，如果验证失败的情况下，要及时抛出异常给用户，所以在验证的时候就要注意抛出异常，FluentValidation也提供了这样的机制，它为验证器提供了一个ValidateAndThrow 的扩展方法。使用这个方法后，如果碰到失败的情况，会及时抛出一个ValidationException 异常，让业务验证过程使用。主要代码如下：

|  |  |
| --- | --- |
| 1  2  3 | Customer customer = new Customer();  CustomerValidator validator = new CustomerValidator();  validator.ValidateAndThrow(customer); |

**5.2 复杂验证的处理**

    我们前面处理的都是单个验证器的使用，构成也基本清楚了，但如果在当前验证的实体类中还有其他对象，也需要对这个对象进行各种属性验证，该怎么办。举个例子，假设我们有一个 客户类 Customer 以及地址类Address，在Customer 中包括了一个Address类型的属性，用来存储当前客户的地址信息，如下面所示代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | //客户类  public class Customer  {      public string Name { get; set; }      public Address Address { get; set; }  }  //地址类  public class Address  {      public string Line1 { get; set; }      public string Line2 { get; set; }      public string Town { get; set; }      public string County { get; set; }      public string Postcode { get; set; }  } |

    由于我们也需要对Address类进行，验证，所以先类似的构造一个Address验证器：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class AddressValidator : AbstractValidator<Address>  {      public AddressValidator()      {          RuleFor(address => address.Postcode).NotNull();      }  } |

    然后我们同理要构造CustomerValidator验证器，在验证Address的时候，及可以复用上面的AddressValidator，如下面代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class CustomerValidator : AbstractValidator<Customer>  {      public CustomerValidator()      {          RuleFor(customer => customer.Name).NotNull();          //对Address使用验证器直接进行验证，可以重复使用代码          RuleFor(customer => customer.Address).SetValidator(new AddressValidator())      }  } |

    过程比较简单，更加复杂的处理也类似，应该没啥问题了。值得注意的是，如果Address 是集合类型，如List<Address> ,则要使用SetCollectionValidator来进行验证，和SetValidator的使用类似。

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html#_labelTop)

**6.灵活的验证规则组设置**

    我们在编写验证过程中，可以编写大量的验证方法，针对不同使用场景，不同字段。但随着业务的复杂，并不是每一个验证器的使用的时候都要执行所有的验证规则。有的时候可能只需要执行一部分就可以了，否则需要针对同一个类型，编写很多个不同业务场景的验证器，显然则是非常残酷的。而[FluentValidation](https://github.com/JeremySkinner/FluentValidation)则提供了处理这种问题的灵活性，将规则组集合在一起，并赋予一个名称，在执行的时候，可以只执行指定名称规则组的规则。看看下面的例子：

    假设一个Person类有3个属性(Id,Surname,Forename)，我们写一个验证器分别对几个属性进行验证，并将其中2个的验证规则放在一个规则集合里面RuleSet，名称为:Names，如下面代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class PersonValidator : AbstractValidator<Person>  {      public PersonValidator()      {   //名称为 Names 的规则组          RuleSet("Names", () =>              {                  RuleFor(x => x.Surname).NotNull();                  RuleFor(x => x.Forename).NotNull();              });          //其他没有集合的组名称，默认为：default          RuleFor(x => x.Id).NotEqual(0);      }  } |

    然后我们在验证的时候，就可以灵活指定规则集的名称了，一次可以指定单个或者多个，值得注意的是，没有放在集合中的规则，默认在 default 组中。如下面的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var validator = new PersonValidator();  var person = new Person();  //只执行 Names 规则集  var result = validator.Validate(person, ruleSet: "Names");  //执行 Names,MyRuleSet,SomeOtherRuleSet 规则集  validator.Validate(person, ruleSet: "Names,MyRuleSet,SomeOtherRuleSet")  //执行默认的规则(不在集合中的) 和 MyRuleSet  validator.Validate(person, ruleSet: "default,MyRuleSet") |

    上面就是今天的内容，我们对验证器的完整过程和使用细节进行了很深入的研究，相信自己构造一个强大的验证器已经很容易了吧。接下来的内容，我们将继续介绍内置的一些验证方法和规则。对于该组件的源码可以直接从github获取，本文将在后面的文章中发布自己制作的CHM帮助文档。敬请关注！

# 进阶

**阅读目录**

* [1.FluentValidation内置验证器](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html#_label0)
* [2.验证器错误信息配置](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html#_label1)
* [3.其他内容](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html#_label2)
* [4.帮助文档](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html#_label3)

    在上一篇文章：[.NET平台开源项目速览(6)FluentValidation验证组件介绍与入门(一)](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html) 中，给大家初步介绍了一下[FluentValidation](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html)验证组件的使用情况。文章从构建间的验证器开始，到最后的结果，以及复杂验证等都做了比较深入的讲解和使用。但其实一个完整的组件是麻雀虽小五脏俱全的，一篇文章还不能说明问题，对优秀的组件，我将分开尽量多一点的进行很完整的介绍。在查阅了官方提供的帮助文件中，还有一些组件内部以及个性化的东西，所以今天一起看看看这方面的内容。

    为了保持内容的完整性，大部分内容我都是参考[FluentValidation](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html)提供的帮助文档，自己经过翻译和理解加工。更好的呈现给大家。

.NET开源目录：[【目录】本博客其他.NET开源项目文章目录](http://www.cnblogs.com/asxinyu/p/Bolg_Category_For_DotNet_OpenSource.html)

本文原文地址：[.NET平台开源项目速览(10)FluentValidation验证组件深入使用(二)](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html)

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html#_labelTop)

**1.FluentValidation内置验证器**

[FluentValidation](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html)在使用的时候，我们也可以看到例子中，有一些验证方法，例如NotNull等，其实是其内部已经实现的几个常规的验证。所以先来看看他们的使用以及作用。FluentValidation本身附带了一个内置的验证器，每个验证器都可以输出固定格式的错误信息。这些内置的验证器是大家自定义验证器的基础。

**1.1 空值验证:NotNull,NotEmpty**

1.NotNull：确保属性的值不是Null，例如代码：

|  |  |
| --- | --- |
| 1 | RuleFor(customer => customer.Surname).NotNull(); |

2.NotEmpty：确保属性值不是Null，不为空，或者空白字符串，相当于String的IsNullOrWhiteSpace方法

|  |  |
| --- | --- |
| 1 | RuleFor(customer => customer.Surname).NotEmpty(); |

    上面应该是最简单也是最常用的吧，毕竟验证主要是空值相关的。内置验证器有错误信息的格式，例如NotEmpty的信息就是这个样子：

|  |  |
| --- | --- |
| 1  2 | {PropertyName} = The name of the property being validated  {PropertyValue} = The current value of the property |

**1.2 相等验证:NotEqual,Equal**

1.NotEqual：不相等验证，可以验证固定字符串，也可以验证和其他属性相对，比如验证密码和用户名不能相同：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.Surname).NotEqual("Foo");  RuleFor(customer => customer.Surname).NotEqual(customer => customer.Forename); |

2.Equal：去报指定的字段和其他固定信息或者属性的值是相等的，例如：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.Surname).Equal("Foo");  RuleFor(customer => customer.Password).Equal(customer => customer.PasswordConfirmation); |

**1.3 大小比较验证:Length,Less Than，Greater Than等**

1.Length：长度比较，主要是确定字符串的长度范围，相当于Range，例如：

|  |  |
| --- | --- |
| 1  2 | //Surname的长度要在1-250个字符之间  RuleFor(customer => customer.Surname).Length(1, 250); |

2.Less Than：主要是进行值比较，确定值＜某个给定值或者属性，例如：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.CreditLimit).LessThan(100);  RuleFor(customer => customer.CreditLimit).LessThan(customer => customer.MaxCreditLimit); |

3.Less Than Or Equal：比较值小于或者等于某个值或者属性，例如：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.CreditLimit).LessThanOrEqual(100);  RuleFor(customer => customer.CreditLimit).LessThanOrEqual(customer => customer.MaxCreditLimit); |

4.Greater Than：比较值大于某个值或者属性，例如：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.CreditLimit).GreaterThan(0);  RuleFor(customer => customer.CreditLimit).GreaterThan(customer => customer.MinimumCreditLimit); |

5.Greater Than Or Equal：比较值大于或者等于某个值或者属性，例如：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.CreditLimit).GreaterThanOrEqual(1);  RuleFor(customer => customer.CreditLimit).GreaterThanOrEqual(customer => customer.MinimumCreditLimit); |

**1.4 其他验证:正则验证,Email**

     同时[FluentValidation](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_1.html)也支持Email验证以及正则表达式验证。正则表达式是比较强大的，直接支持应该可以给不少人提供方便，虽然我不太懂这玩意。但了解了解也是好的。看看例子：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.Surname).Matches("some regex here");  RuleFor(customer => customer.Email).EmailAddress(); |

**1.5 高级委托验证：Predicate**

     这个单词不知道咋翻译，就是这个验证和其他的不一样，可以允许使用委托来进行验证。使用Must方法，也可以接受Lambda表达式，它的使用方法比较多，可以直接指定属性，或者指定类型，或者只给指定的值去验证都是可以的，例如下面这几种用法：

1.单独验证属性：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.Identification).Must(BeUniqueId);  public bool BeUniqueId(Customer customer, CustomerId id) {  if (id == ... |

2.单独验证类:

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer).Must(HaveUniqueId);  public bool HaveUniqueId(Customer customer) { if (customer.Id == ... |

3.Lambda表达式:

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.Id).Must(id => IsUniqueId(id));  public bool IsUniqueId(ID id) { if (id == ... |

    这些内置的验证器就介绍到这里。总的来说，用了这些方法，构建自定义的验证器就更加方便快捷了。下面我们简单介绍一下错误信息的配置以及本地化的内容。

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html#_labelTop)

**2.验证器错误信息配置**

    在进行验证的时候，如果不重载错误信息方法，会提示类似默认验证器的一些验证错误信息，但也可以进行自定义。例如，下面2个例子：

|  |  |
| --- | --- |
| 1  2 | RuleFor(customer => customer.Surname).NotNull()                                       .WithMessage("Please ensure that you have entered your Surname"); |

     当然也可以搞更复杂的，格式化的信息，例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | RuleFor(customer => customer.Surname)    .NotNull()    .WithMesasge("This message references some other properties: Forename: {0} Discount: {1}",      customer => customer.Forename,      customer => customer.Discount    ); |

    当然还有一些简单的用法，大家可以参考官方文档：

<https://github.com/JeremySkinner/FluentValidation/wiki/d.-Configuring-a-Validator>

[回到目录](http://www.cnblogs.com/asxinyu/p/dotnet_Opensource_project_FluentValidation_2.html#_labelTop)

**3.其他内容**

    其他内容，例如一些特殊的使用方法，限于篇幅和复杂，暂时不翻译了。这2篇文章已经足够全面的介绍该组件的功能了。其他内容可以参考下面页面：

<https://github.com/JeremySkinner/FluentValidation/wiki/f.-Localization>

<https://github.com/JeremySkinner/FluentValidation/wiki/e.-Custom-Validators>