**[IoC容器Autofac之实例引入(一)](http://www.cnblogs.com/sylone/p/6080912.html)**

 先不必尝试理解IOC,先来看段代码。

**一、一个没有使用IoC的例子**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class MPGMovieLister     {         public Movie[] GetMPG()         {             var finder = new ListMovieFinder();             var allMovies = finder.FindAll();               return allMovies.Where(m => m.Name.EndsWith(".MPG")).ToArray();         }     }      public class ListMovieFinder     {         public List<Movie> FindAll()         {             return new List<Movie>                        {                            new Movie                                {                                    Name = "Die Hard.wmv"                                },                            new Movie                                {                                    Name = "My Name is John.MPG"                                }                        };         }     } |

类MPGMovieLister的作用是列出所有的mpg类型的电影，其中调用了类ListMovieFinder类的方法FindAll()来获取所有的电影。

这段代码看起来还不错，已经符合当前的需求了。

**二、当需求发生变动时，非IoC遭遇到的困境**

 假如这个时候，movie的列表获取不是直接创建一个list获取，而要求从某个文本文件读取，或者是数据库获取，又或者从web service中获取，我们怎么办？

第一步，再实现一个类, 比如FileMovieFinder，来实现从文本文件中读取Movie列表，再把MPGMovieLister中的这行代码，

     var finder = new ListMovieFinder();   
替换成

    var finder = new FileMovieFinder();   
那么这行代码就又能够符合要求了。

如果底层获取数据的方式不确定，或者经常更改，MPGMovieLister的代码岂不是要频繁改动?

**三、使用IoC彻底解决问题:**

**上面的问题是什么？MPGMovieLister的功能都是依赖着具体的类，ListMovieFinder，FileMovieFinder。**

当需求发生变化的时候，就会导致MPGMovieLister的代码也要做相应的改动。

跳出来看，MPGMovieLister的功能只是负责从列表中找出MPG类型的movie, 至于movie从什么地方来的，不是MPGMovieLister的职责，它也不需要关心。

解耦合的方法就是”依赖于抽象，而不是依赖于具体”.(非常类似于持久层(数据层)和业务逻辑层，业务逻辑层也不关心数据是如何提供的，所以业务逻辑层也应当与持久层解耦合。)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | 实际解决之后的代码：  public class MPGMovieLister     {         public Movie[] GetMPG()        {             var finder = MovieFinderFactory.GetFinder();             var allMovies = finder.FindAll();             return allMovies.Where(m => m.Name.EndsWith(".MPG")).ToArray();         }     }    public class MovieFinderFactory  {       public static IMovieFinder GetFinder()       {           return new FileMovieFinder();       }  }    public interface IMovieFinder  {      List<Movie> FindAll()  } |

　　这里的MovieFinderFactory就已经是一个简陋的IoC容器功能了

**四、总结**

我相信到这里你已经猜到IoC的功能了--解耦，即解决类之间的强依赖关系。

真正的IoC容器比上面的MovieFinderFactory自然要好用和适用的多。

# [IoC容器Autofac之IOC/DI基本概念(二)](http://www.cnblogs.com/sylone/p/6080913.html)

原文：<http://www.cnblogs.com/xdp-gacl/p/4249939.html>

### 1.1、IoC是什么

　　Ioc—Inversion of Control，即“控制反转”，一种设计思想**。**如何理解IOC呢？

　　●传统程序设计直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象(正转)；

        而反转则是将本由自身控制的对象初始化交由外部IoC容器进行；即由Ioc容器来控制依赖对象的创建和注入；

### 1.2、IoC和DI

　　DI—Dependency Injection，即“依赖注入”：**动态的向某个对象提供它所需要的其他对象。**

　　理解DI的关键是下面几点：

　　●**谁注入谁：IoC容器注入应用程序某个对象**；

**●注入了什么：注入某个对象所需要的外部资源（包括对象、资源、常量数据）**。

**IoC和DI**由什么**关系**呢？其实它们**是同一个概念的不同角度描述。**

**1.3  总结**

**控制反转IoC是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方，比如转移交给了IoC容器**

**DI(依赖注入)其实就是IOC的另外一种说法**

回顾之前的代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | //这个类的作用是筛选出MPG类型的电影  public class MPGMovieLister  {    public Movie[] GetMPG()    {         var finder = MovieFinderFactory.GetFinder();//这里调用工厂类获取具体的实例，得到一个电影列表         var allMovies = finder.FindAll();         return allMovies.Where(m => m.Name.EndsWith(".MPG")).ToArray();     }  }    public class MovieFinderFactory  {       public static IMovieFinder GetFinder()       {           return new ListMovieFinder();       }  }    public class ListMovieFinder :IMovieFinder  {     public List<Movie> FindAll()     {         return new List<Movie>                    {                        new Movie                            {                                Name = "Die Hard.wmv"                            },                        new Movie                        {                            Name = "My Name is John.MPG"                        }                   };     }  }    public interface IMovieFinder { List<Movie> FindAll() } |

**二、改造代码,去除MovieFinderFactory**

在应用Autofac替换MovieFinderFactory之前，我们先从代码中去掉MovieFinderFactory, 改动之后的代码是这样：

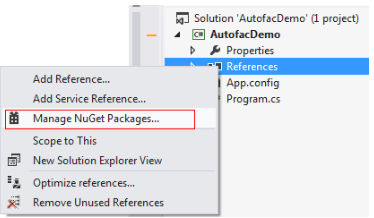
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class MPGMovieLister  {      private readonly IMovieFinder \_movieFinder;      //增加了构造函数，参数是IMovieFinder对象      public MPGMovieLister(IMovieFinder movieFinder)      {           \_movieFinder = movieFinder;      }        public Movie[] GetMPG()      {       var allMovies = \_movieFinder.FindAll();       return allMovies.Where(m => m.Name.EndsWith(".MPG")).ToArray();      }  }    public interface IMovieFinder  {      List<Movie> FindAll()  } |

　　我们去掉了工厂类MovieFinderFactory， 改造了MPGMovieLister， 添加了一个构造函数， 构造函数要求使用MPGMovieLister时，需要提供一个IMovieFinder的实例。

**三、应用Autofac替代工厂类**

应用Autofac改造上面的代码。

**第一步: 从Nuget中添加Autofac引用**



**第二步:**

**\* 创建一个ContainerBuilder对象(ContainerBuilder从字面的意思就是用来创建Container(容器)的，而Conainter就是我们从中取各种我们需要对象的地方)**

**\* 注册我们后面将从容器中取出对象的类型。**

代码是这样：

var builder = new ContainerBuilder();

builder.RegisterType<ListMovieFinder>().As<IMovieFinder>();//注册ListMovieFinder类型

builder.RegisterType<MPGMovieLister>();//注册MPGMovieLister类型

**\* 创建容器**

\_container = builder.Build();

**第三步: 在程序中使用 \_container容器：**

[复制代码](javascript:void(0);)

复制代码

复制代码

var lister = \_container.Resolve<MPGMovieLister>();

foreach (var movie in lister.GetMPG())

{

Console.WriteLine(movie.Name);

}

复制代码

复制代码

[复制代码](javascript:void(0);)

**理解一下Autofac为我们在背后做了什么：**

首先我们注册了类型ListMovieFinder和MPGMovieLister，这样容器就能够知道如何创建这两种类型的实例了。(类其实是创建对象的模板，当我们把模板注册给Autofac, 它就会遵循这个模板为我们提供实例)

后面的代码中，我们调用Resolve方法，取出一个MPGMovieLister的实例。

\_container.Resolve<MPGMovieLister>();

    这里还有一个需要解释的，对于MPGMovieLister类型，我们为Autofac提供了类型, 但是当Autofac创建MPGMovieLister的实例, 调用它的构造函数的时候，却遇到了问题：

它的构造函数需要提供一个IMovieFinder的实例作为参数, 聪明的Autofac要在自己的容器里找找，看看没有有办法提供一个IMovieFinder的实例。

这个时候Autofac会发现我们注册过ListMovieFinder且通过AsImplementedInterfaces()方法，指明了就是为接口IMovieFinder提供实例的。

所以Autofac会创建一个ListMovieFinder的实例，作为创建MPGMovieLister时，提供给构造函数的参数。

builder.RegisterType<ListMovieFinder>().AsImplementedInterfaces();

**四、当需求发生变动Autofac如何应对?**

上面的例子中类ListMovieFinder实现了IMovieFinder接口， 实际运行中由它来提供数据。

假如这个时候，我们要从数据库中获取数据，怎么办?

非常简单，创建一个类DBMovieFinder继承IMovieFinder接口， 然后注册给Autofac就可以了。

|  |  |
| --- | --- |
| 1  2  3  4  5 | var builder = new ContainerBuilder();  builder.RegisterType<ListMovieFinder>().AsImplementedInterfaces();  //这里注册了DBMovieFinder， 这个类继承IMovieFinder接口。因为它也使用了AsImplementedInterfaces，它会覆盖ListMovieFinder的注册。  builder.RegisterType<DBMovieFinder>().AsImplementedInterfaces(); <br>builder.RegisterType<MPGMovieLister>();  \_container = builder.Build(); |

**五、Autofac对程序架构的影响**

常见的程序架构大概是: UI层， 业务逻辑层， 持久层(数据层)。

我们可用Autofac作为不同层之间的中间人，让UI层依赖于业务逻辑层的抽象接口，业务逻辑层依赖于持久层的接口，而实际运行过程中的实例都由Auotfac来提供。

这样就能够解除不同层之间的依赖，将所有的注册类型的操作在一个核心函数或者核心类中实现，那么只要修改这个函数或者类就能够非常方便的让它们之间的依赖关系发生变化。

比如 在一个大的项目中，持久层和业务逻辑层是并行开发的，而且是不同团队开发，这个时候业务逻辑开发团队的人在没有持久层代码的情况下，如何开始呢？

我们只要定义好持久层的接口， 业务逻辑团队再写一些Stub类(桩类)来实现这些接口，让这些Stub类来替换真正的持久层，所要做的就只是简单的把这些Stub类型注册到Autofac中就可以了。同时做业务逻辑层的单元测试也非常容易了。

**六、 总结**

从上面的例子可以看出，使用IoC对于复杂的项目来说，非常有意义，能够为我们搭建一个好的开发层次。

同时，在使用过程中，还能够发现Autofac有以下优点：

1. 可以使用C#代码来完成注册配置，非常方便而且便于调试。(使用xml配置，往往容易出现格式不对，或者其它问题，非常难于调试和排错)

2. 非常聪明，能够自动装配(发现构造函数需要的必须参数的时候，会自己想办法解决)

[**IoC容器Autofac之实例优化(三)**](http://www.cnblogs.com/sylone/p/6080917.html)

回顾之前的代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | //这个类的作用是筛选出MPG类型的电影  public class MPGMovieLister  {    public Movie[] GetMPG()    {         var finder = MovieFinderFactory.GetFinder();//这里调用工厂类获取具体的实例，得到一个电影列表         var allMovies = finder.FindAll();         return allMovies.Where(m => m.Name.EndsWith(".MPG")).ToArray();     }  }    public class MovieFinderFactory  {       public static IMovieFinder GetFinder()       {           return new ListMovieFinder();       }  }    public class ListMovieFinder :IMovieFinder  {     public List<Movie> FindAll()     {         return new List<Movie>                    {                        new Movie                            {                                Name = "Die Hard.wmv"                            },                        new Movie                        {                            Name = "My Name is John.MPG"                        }                   };     }  }    public interface IMovieFinder { List<Movie> FindAll() } |

**二、改造代码,去除MovieFinderFactory**

在应用Autofac替换MovieFinderFactory之前，我们先从代码中去掉MovieFinderFactory, 改动之后的代码是这样：

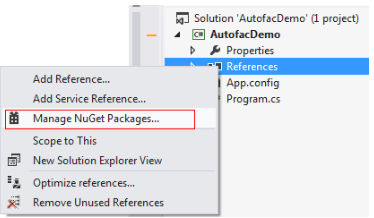
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class MPGMovieLister  {      private readonly IMovieFinder \_movieFinder;      //增加了构造函数，参数是IMovieFinder对象      public MPGMovieLister(IMovieFinder movieFinder)      {           \_movieFinder = movieFinder;      }        public Movie[] GetMPG()      {       var allMovies = \_movieFinder.FindAll();       return allMovies.Where(m => m.Name.EndsWith(".MPG")).ToArray();      }  }    public interface IMovieFinder  {      List<Movie> FindAll()  } |

　　我们去掉了工厂类MovieFinderFactory， 改造了MPGMovieLister， 添加了一个构造函数， 构造函数要求使用MPGMovieLister时，需要提供一个IMovieFinder的实例。

**三、应用Autofac替代工厂类**

应用Autofac改造上面的代码。

**第一步: 从Nuget中添加Autofac引用**



**第二步:**

**\* 创建一个ContainerBuilder对象(ContainerBuilder从字面的意思就是用来创建Container(容器)的，而Conainter就是我们从中取各种我们需要对象的地方)**

**\* 注册我们后面将从容器中取出对象的类型。**

代码是这样：

var builder = new ContainerBuilder();

builder.RegisterType<ListMovieFinder>().As<IMovieFinder>();//注册ListMovieFinder类型

builder.RegisterType<MPGMovieLister>();//注册MPGMovieLister类型

**\* 创建容器**

\_container = builder.Build();

**第三步: 在程序中使用 \_container容器：**

[复制代码](javascript:void(0);)

复制代码

复制代码

var lister = \_container.Resolve<MPGMovieLister>();

foreach (var movie in lister.GetMPG())

{

Console.WriteLine(movie.Name);

}

复制代码

复制代码

[复制代码](javascript:void(0);)

**理解一下Autofac为我们在背后做了什么：**

首先我们注册了类型ListMovieFinder和MPGMovieLister，这样容器就能够知道如何创建这两种类型的实例了。(类其实是创建对象的模板，当我们把模板注册给Autofac, 它就会遵循这个模板为我们提供实例)

后面的代码中，我们调用Resolve方法，取出一个MPGMovieLister的实例。

\_container.Resolve<MPGMovieLister>();

    这里还有一个需要解释的，对于MPGMovieLister类型，我们为Autofac提供了类型, 但是当Autofac创建MPGMovieLister的实例, 调用它的构造函数的时候，却遇到了问题：

它的构造函数需要提供一个IMovieFinder的实例作为参数, 聪明的Autofac要在自己的容器里找找，看看没有有办法提供一个IMovieFinder的实例。

这个时候Autofac会发现我们注册过ListMovieFinder且通过AsImplementedInterfaces()方法，指明了就是为接口IMovieFinder提供实例的。

所以Autofac会创建一个ListMovieFinder的实例，作为创建MPGMovieLister时，提供给构造函数的参数。

builder.RegisterType<ListMovieFinder>().AsImplementedInterfaces();

**四、当需求发生变动Autofac如何应对?**

上面的例子中类ListMovieFinder实现了IMovieFinder接口， 实际运行中由它来提供数据。

假如这个时候，我们要从数据库中获取数据，怎么办?

非常简单，创建一个类DBMovieFinder继承IMovieFinder接口， 然后注册给Autofac就可以了。

|  |  |
| --- | --- |
| 1  2  3  4  5 | var builder = new ContainerBuilder();  builder.RegisterType<ListMovieFinder>().AsImplementedInterfaces();  //这里注册了DBMovieFinder， 这个类继承IMovieFinder接口。因为它也使用了AsImplementedInterfaces，它会覆盖ListMovieFinder的注册。  builder.RegisterType<DBMovieFinder>().AsImplementedInterfaces(); <br>builder.RegisterType<MPGMovieLister>();  \_container = builder.Build(); |

**五、Autofac对程序架构的影响**

常见的程序架构大概是: UI层， 业务逻辑层， 持久层(数据层)。

我们可用Autofac作为不同层之间的中间人，让UI层依赖于业务逻辑层的抽象接口，业务逻辑层依赖于持久层的接口，而实际运行过程中的实例都由Auotfac来提供。

这样就能够解除不同层之间的依赖，将所有的注册类型的操作在一个核心函数或者核心类中实现，那么只要修改这个函数或者类就能够非常方便的让它们之间的依赖关系发生变化。

比如 在一个大的项目中，持久层和业务逻辑层是并行开发的，而且是不同团队开发，这个时候业务逻辑开发团队的人在没有持久层代码的情况下，如何开始呢？

我们只要定义好持久层的接口， 业务逻辑团队再写一些Stub类(桩类)来实现这些接口，让这些Stub类来替换真正的持久层，所要做的就只是简单的把这些Stub类型注册到Autofac中就可以了。同时做业务逻辑层的单元测试也非常容易了。

**六、 总结**

从上面的例子可以看出，使用IoC对于复杂的项目来说，非常有意义，能够为我们搭建一个好的开发层次。

同时，在使用过程中，还能够发现Autofac有以下优点：

1. 可以使用C#代码来完成注册配置，非常方便而且便于调试。(使用xml配置，往往容易出现格式不对，或者其它问题，非常难于调试和排错)

2. 非常聪明，能够自动装配(发现构造函数需要的必须参数的时候，会自己想办法解决)

[**IoC容器Autofac正篇之简单实例（四）**](http://www.cnblogs.com/sylone/p/6080920.html)

先上一段代码。

[复制代码](javascript:void(0);)

复制代码

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

ContainerBuilder builder = new ContainerBuilder();//容器构造器 组件中的类型通过此对象注册到容器中

builder.RegisterType<AutoFacManager>();//注册类型

builder.RegisterType<Worker>().As<IPerson>();//注册类型且用as方法指定此类型是IPerson接口

using (IContainer container = builder.Build())//build方法创建容器

{

AutoFacManager manager = container.Resolve<AutoFacManager>();//通过resolve方法取得对象

manager.Say();

}

Console.ReadKey();

}

}

public interface IPerson

{

void Say();

}

public class Worker : IPerson

{

public void Say()

{

Console.WriteLine("我是一个工人！");

}

}

public class Student : IPerson

{

public void Say()

{

Console.WriteLine("我是一个学生！");

}

}

public class AutoFacManager

{

IPerson person;

public AutoFacManager(IPerson MyPerson)

{

person = MyPerson;

}

public void Say()

{

person.Say();

}

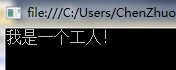
}

}

复制代码

[复制代码](javascript:void(0);)

以上例子，是一个最简单的例子，输出如下：



**总结：**

   1.明确autofac的作用：简单说，自动创建程序所依赖对象的实例，解除程序中的强依赖。（如Program类和AutoFacManager类）

   2.思考：它何时帮你创建？在你需要使用这个对象的时候也就是获取这个对象即[resolve时](http://www.cnblogs.com/tiantianle/p/5154801.html)。

             创建的前提是什么？你已经向autofac注册了这个对象所依赖的资源([类型注册](http://www.cnblogs.com/tiantianle/p/5154742.html))和([类型关联](http://www.cnblogs.com/tiantianle/p/5194027.html))，autofac会进行匹配。

可学习的实例：<http://www.cnblogs.com/liping13599168/archive/2011/07/16/2108209.html>  引入配置文件

# [IoC容器Autofac正篇之类型注册(五)](http://www.cnblogs.com/sylone/p/6080926.html)

## Autofac类型注册

     类型注册简单的从字面去理解就可以了，不必复杂化，只是注册的手段比较丰富。

#### ****(一)类型/泛型注册****

builder.RegisterType<Class1>();

这种简单较常用，但缺点是注册的类型必须在当前项目或被当前项目引用，因为使用泛型，必须类型明确。

针对这点，还有一种通过Type对象进行注册的方式：

//字符串为类型完全名称

builder.RegisterType(Type.GetType("AutofacBlog.Class\_1"));

使用这种方式进行类型注册，被注册的类型可以不是被直接引用，但类型所在的程序集必须被加载。

这种注册方式在有插件或类似需要动态加载程序集的情况下使用，通过扫描程序集，获取一些满足指定条件的类型，来进行注册。

#### (二)程序集批量注册

类型注册中提到了通过扫描程序集来获取部分类型进行注册。Autofac对此提供了一个方便的方式，可以直接通过程序集来筛选类型注册：

//获取当前应用程序加载程序集（C/S应用中使用）

var assembly = Assembly.GetExecutingAssembly();

var builder = new ContainerBuilder();

builder.RegisterAssemblyTypes(assembly); //注册所有程序集类定义的非静态类型

代码通过RegisterAssemblyTypes方法，将assembly中所有自定义的非静态类型都注册到Autofac中，后面可以使用IContainer对象获取所有该程序集中自定义的类型对象。

这种方式达到了批量的效果，但是通常，我们并不需要把所有的类型都进行注册，所以Autofac提供了几种过滤方式：

builder.RegisterAssemblyTypes(assembly)

.Where(type => type.Namespace.Equals("IocAutofac.Example")); //条件判断

这种方式相信大家都比较熟悉，Where+lambda的方式来进行条件判断，lambda参数为Type类型，也就是程序集中的type。

builder.RegisterAssemblyTypes(assembly)

.Except<Program>(); //排除Program类型

这种方式用来排除指定类型的注册，当排除的个例比较少时，会比较适用。Except还有另一种用法，但是用法比较复杂，在此不进行介绍。

builder.RegisterAssemblyTypes(assembly)

.InNamespace("IocAutofac.Example"); //类型在IocAutofac.Example命名空间中

被注册的类型需要在指定命名空间中。

builder.RegisterAssemblyTypes(assembly)

.InNamespaceOf<Program>(); //类型在Program所在的命名空间中\*/

这种方式与上面一种方式比较相似，也是用来判断命名空间的，这种方式是根据类型来获取命名空间。

通过这种方式，我们可以对程序集中的类型进行批量注册，类型/泛型方式在被注册类型较少的情况下还是不错的，但当被注册类型很多的时候，一个一个的手写注册会显得很无力，这时候就是程序集批量注册显威的时候了。

#### (三)Lambda注册

上面讲到的两种方式都是通过类型进行直接注册的，这种注册方式，在获取时，会直接通过构造函数new出对象，不会做更多的操作。

有时，我们希望能够在获取对象时能够自动的做更多的事情时，我们可以通过Lambda注册来解决：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | builder.Register(cc =>  {      var clas1 = new Class\_1();      while (clas1.Id.ToString().Contains("a"))      {          clas1.Id = Guid.NewGuid();      }      return clas1;  }); |

上述代码，实际注册了Class\_1类型，因为最后返回的对象类型为Class\_1。

Register方法接收了一个lambda表达式作为参数，在lambda表达式中，我们可以做很多事，包括一些属性注入（后续说明）、方法注入（后续说明）、条件判断等等。

#### (四)实例注册

var clas1 = new Class\_1();

clas1.Id = Guid.Empty;

builder.RegisterInstance(clas1);

通过RegisterInstance进行实例注册，进行实例注册时，我们需要注意，实例注册可以作为一种单例注册的方式，也就是在后面通过Autofac获取Class\_1对象时，获取到的是注册时的那个对象。并且，如果一个在某处修改了该对象，其他地方再获取时，获取到的就是修改后的对象。

#### (五)Module注册

在日常开发中，可能不同开发会负责不同的模块进行单独开发。在开发过程中，不同模块可能都有自己的类型需要注册到autofac中，但是如果每个人在注册时，都去修改一个指定地方的代码，这在进行代码合并时，是令人痛苦的。更好的方式是每个开发不同的模块都有自己指定的类型注册区域，这样，在代码合并时，会减少很多代码冲突。

对于这种方式，Autofac已经为我们提供了：

[复制代码](javascript:void(0);)

复制代码

复制代码

class Program

{

static void Main(string[] args)

{

var builder = new ContainerBuilder();

builder.RegisterModule<ModuleA>();

//这两种注册方式达到的效果都一样

builder.RegisterModule(new ModuleB());

IContainer container = builder.Build();

Class\_1 clas1 = container.Resolve<Class\_1>();

Class\_2 clas2 = container.Resolve<Class\_2>();

Console.WriteLine(clas1.Id);

Console.WriteLine(clas2.ToString());

Console.Write("Press any key to continue...");

Console.ReadKey();

}

}

class ModuleA : Module

{

protected override void Load(ContainerBuilder builder)

{

builder.RegisterType<Class\_1>();

}

}

class ModuleB : Module

{

protected override void Load(ContainerBuilder builder)

{

builder.RegisterType<Class\_2>();

}

}

复制代码

复制代码

[复制代码](javascript:void(0);)

两个继承自Module类的类：ModuleA、ModuleB重写了父类的Load方法并在load方法中注册了Class\_1与Class\_2类型，然后在主程序中，通过RegisterModule对Module进行注册。不同的开发就可以各自创建一个类继承自Module，然后重写Load方法，在Load方法进行自己的类型注册，最后再进行Module的统一注册。

##### Module注意说明

实际上，RegisterModule需要的参数，并不是继承自Module类的，而是实现了IModule接口的类，而Module也是实现了IModule接口的。也就是我们也可以写一个实现了IModule接口的类型，然后在RegisterModule时传入。但是一般我们直接去继承Module就好了，这种方式比较简单方便，实现IModule的方式更为复杂，当然，功能也更多，在此就不进行介绍了。

#### 程序集Module注册

Module注册，为多人开发提供了一种方便的注册方式，但这种方式还是会需要手动注册Module，如果Module过多，Module注册代码也会显得多而杂，当然，可以通过人工管理来控制Module的量。但是Autofac还提供了一种更方便的方式且对于类似Orchard的模块开发（子模块与主模块无引用关系，通过程序集加载方式来加载子模块）或是插件开发，我们没办法通过Registerodule来注册无直接引用关系的Module。

对于上述的情况，Autofac提供了很好的方式来解决：

var builder = new ContainerBuilder();

var assembly = Assembly.GetExecutingAssembly();

builder.RegisterAssemblyModules(assembly);

代码注册assembly程序集中所有实现了IModule接口的类型，这样只需要取出所有程序集通过RegisterAssemblyModules进行一次性注册，就可以自动注册所有Module了。

RegisterAssemblyModule还可以指定一个泛型类型：

builder.RegisterAssemblyModules<ModuleA>(assembly);

这样注册是指定只注册assembly程序集中继承自ModuleA的Module。

文章来源： <http://www.cnblogs.com/ancupofcoffee/p/5007649.html>

备注：类型注册只是一部分，个人认为一般不会注册一个类型然后直接去取这个类型的实例，因为没有那个必要。autofac常用的应用场景应该是像简单实例篇中所描述的场景，所以有了类型关联这一篇。

# [IoC容器Autofac正篇之解析获取(六)](http://www.cnblogs.com/sylone/p/6080930.html)

#### 解析获取的方式有如下几种：

#### Resolve

[复制代码](javascript:void(0);)

复制代码

复制代码

class Program

{

static void Main(string[] args)

{

var builder = new ContainerBuilder();

builder.RegisterType<Class\_1>(); //如果注释掉这句，下面Resolve时将会抛出异常

IContainer container = builder.Build();

Class\_1 clas1 = container.Resolve<Class\_1>();

Console.WriteLine(clas1.Id);

Console.Write("Press any key to continue...");

Console.ReadKey();

}

}

复制代码

复制代码

[复制代码](javascript:void(0);)

这种方式在类型已经注册的情况下使用时没问题的，能够获取到注册类型的实例对象，但是如果类型没有经过注册，直接Resolve解析获取，便会抛出异常。

#### ResolveOptional

[复制代码](javascript:void(0);)

复制代码

复制代码

class Program

{

static void Main(string[] args)

{

var builder = new ContainerBuilder();

//builder.RegisterType<Class\_1>(); //这里注释掉类型注册的代码

IContainer container = builder.Build();

Class\_1 clas1 = container.ResolveOptional<Class\_1>();

Console.WriteLine(clas1 == null ? "null" : clas1.Id.ToString()); //这里将会输出null

Console.Write("Press any key to continue...");

Console.ReadKey();

}

}

复制代码

复制代码

[复制代码](javascript:void(0);)

我们可以使用ResolveOptional进行解析获取，当类型没有经过注册时，ResolveOptional方法将会返回null作为结果。

#### TryResolve

[复制代码](javascript:void(0);)

复制代码

复制代码

class Program

{

static void Main(string[] args)

{

var builder = new ContainerBuilder();

//builder.RegisterType<Class\_1>(); //这里注释掉类型注册的代码

IContainer container = builder.Build();

Class\_1 clas1 = null;

if (container.TryResolve<Class\_1>(out clas1))

{

Console.WriteLine(clas1.Id);

}

else

{//这里将会被执行

Console.WriteLine("null");

}

Console.Write("Press any key to continue...");

Console.ReadKey();

}

}

复制代码

复制代码

[复制代码](javascript:void(0);)

这种方式与我们常用的Int32.TryParse相同。使用out参数且返回一个bool类型表示是否成功获取到类型实例。

## 其他相关内容

#### Resolve对象构造方法选择原则

    当我们注册的类型拥有多个构造方法，那么在Resolve时，将会以哪个构造方法为准呢？答案是——尽可能最多参数，下面我们以实例来分析：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | class Program  {      static void Main(string[] args)      {          var builder = new ContainerBuilder();          builder.RegisterType<ConstructorClass>();          builder.RegisterType<Class2>();          builder.RegisterType<Class3>();          var container = builder.Build();          var obj = container.Resolve<ConstructorClass>();          Console.WriteLine(obj);          Console.Write("Press any key to continue...");          Console.ReadKey();      }  }    //构造方法测试类  class ConstructorClass  {      private Class1 \_clas1;      private Class2 \_clas2;      private Class3 \_clas3 = null;        public ConstructorClass()      {          \_clas1 = null; \_clas2 = new Class2 { Id = -1 };      }        public ConstructorClass(Class1 clas1, Class2 clas2)      {          \_clas1 = clas1; \_clas2 = clas2;      }        public ConstructorClass(Class2 clas2, Class3 clas3)      {          \_clas2 = clas2; \_clas3 = clas3;      }        public ConstructorClass(Class2 clas2, Class3 clas3, Guid guid)      {          \_clas1 = new Class1 { Id = guid }; \_clas2 = clas2; \_clas3 = clas3;      }        public ConstructorClass(Class1 clas1, Class2 clas2, Class3 clas3)      {          \_clas1 = clas1; \_clas2 = clas2; \_clas3 = clas3;      }        public override string ToString()      {          return string.Format(              "{{Class1.Id: {0}, Class2.Id: {1}, Class3: {2}}}",              \_clas1 == null ? "null" : \_clas1.Id.ToString(),              \_clas2 == null ? "null" : \_clas2.Id.ToString(),              \_clas3 == null ? "null" : "not null");      }  }    //构造方法参数类型  class Class1  {      public Guid Id { get; set; }  }    //构造方法参数类型  class Class2  {      public int Id { get; set; }  }    //构造方法参数类型  class Class3  {    } |

最终输出结果为 {Class1.Id: null, Class2.Id: 0, Class3: not null} ，最终执行的是第三个构造方法（参数为 Class2, Class3 的）。

    按照字面上里说明”最多参数“，那么理应执行的是最后一个构造方法或倒数第二个构造方法，但是为什么却是第三个？。

     先抛开为什么执行的第三个构造方法，我们还是会有疑问”如果执行的是第三个构造方法，那么Class2和Class3参数分别赋的是什么值？值又是从哪儿来？“，这里就涉及到了后面会讲到的构造注入。我们可以看到，在进行类型注册时，我们是对Class2和Class3进行了注册的，而ConstructorClass又是通过Autofac进行获取的，所以Class2和Class3参数的值是由Autofac进行初始化赋值的，Class2和Class3没有自定义构造方法，所以调用的是默认的空构造方法。

     在知道Class2和Class3参数的初始化与赋值缘由后，我们再来看看之前的那个问题，其实现在就好明白了，因为最后两个的构造方法，一个需要额外的Guid类型参数，另一个需要Class1类型参数，而这两个类型又没有经过注册，如果调用这两个构造方法，那么Auotofac将不知道应该赋何值给这两个参数，所以Autofac最终选择了第三个构造方法。

我们还需要注意一点，如果倒数第二个构造方法的Guid参数给上默认值，那么最后选择的构造方法将会是这个构造方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public ConstructorClass(Class2 clas2, Class3 clas3, Guid guid = default(Guid))  {      \_clas1 = new Class1 { Id = guid };      \_clas2 = clas2;      \_clas3 = clas3;  } |

　　如果在上面改造了倒数第二个构造方法的基础上继续改造最后一个构造方法，将Class1参数也默认赋值为null，那么最后在Resolve获取ConstructorClass实例时，将会抛出异常。因为在尽可能最多的原则上，出现了两个最多参数的构造方法，Autofac不知道应该选择哪个进行执行。异常信息告诉我们可以使用UsingConstructor来解决这个问题（关于UsingConstructor的用法，将在后续博文中进行说明）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public ConstructorClass(Class2 clas2, Class3 clas3, Class1 clas1 = null)  {      \_clas1 = clas1;      \_clas2 = clas2;      \_clas3 = clas3;  } |

#### 解析获取传参

我们明白了Autofac在Resolve时对构造方法选择的原则，尽可能最多的参数中的参数，可以是已经注册的类型，或是赋给默认值，除了这两种方式，还有一种方式是在Resolve时指定参数。我们可以通过在Resolve时传参来选择更多参数的构造方法：

var obj = container.Resolve<ConstructorClass>(new NamedParameter("guid", Guid.NewGuid()));

在Resolve时传入了一个NamedParameter，NamedParameter表示按名字匹配参数，上面的代码表示，为参数名为guid的构造参数传入了Guid.NewGuid值。这段代码最后执行的是第四个构造方法。因为第四个构造方法是能够匹配的最多参数的构造方法。

Resolve的方法签名为：Resolve<T>(this IComponmentContext context, params Parameter[] parameters)

第一个参数也就是我们使用的container，我们主要关注第二个参数——一个可变的Parameter数组，Parameter是一个抽象类，其中NamedParameter为Parameter的一个子类，除了NamedParameter，还有以下几种子类拱Resolve时使用：

|  |  |
| --- | --- |
| 参数类型 | 参数说明 |
| NamedParameter | 根据名称进行匹配 |
| PositionalParameter | 根据索引进行匹配，注意：起始索引为0 |
| TypedParameter | 根据类型进行匹配，注意：传入多个相同类型的TypedParameter，所有该类型的参数都将采用第一个TypedParameter的值 |
| ResolvedParameter | 接收两个Func参数，两个Func签名都接收两个相同的参数ParameterInfo和IComponmentContext，第一个参数为参数的信息（常使用放射的朋友应该熟悉），第二个参数还是当做IContainer使用就好了。第一个Func的返回值为bool，表明当前这个ResolvedParameter是否使用当前匹配到的参数，如果返回true，则会执行第二个Func；第二个Func返回一个object对象，用于填充构造参数值。 |

下面有一个这些Parameter使用的示例供参考：

[复制代码](javascript:void(0);)

复制代码

复制代码

class Program

{

static void Main(string[] args)

{

var builder = new ContainerBuilder();

builder.RegisterType<ParameterClass>();

var container = builder.Build();

container.Resolve<ParameterClass>(

new NamedParameter("value", "namedParameter"), //匹配名字为value的参数

new TypedParameter(typeof (int), 1), //匹配类型为int的参数

new PositionalParameter(4, "positionalParameter"), //匹配第五个参数（注意，索引位置从0开始）

new TypedParameter(typeof (int), -1), //这个将被抛弃，因为前面已经有一个类型为int的TypedParameter

new ResolvedParameter(

//第一个Func参数用于返回参数是否符合要求，这里要求参数是类，且命名空间不是System开头，所以第四个参数将会匹配上

(pi, cc) => pi.ParameterType.IsClass && !pi.ParameterType.Namespace.StartsWith("System"),

//第二个Func参数在第一个Func执行结果为true时执行，用于给参数赋值，也就是第四个参数的值为这个Func的执行结果

(pi, cc) => new Temp {Name = "resolveParameter"})

);

// 最后的输出结果为： {x:1, y:1, value:'namedParameter', temp.Name:'resolveParameter', obj:'positionalParameter'}

Console.Write("Press any key to continue...");

Console.ReadKey();

}

}

class ParameterClass

{

public ParameterClass(int x, int y, string value, Temp temp, object obj)

{

Console.WriteLine("{{x:{0}, y:{1}, value:'{2}', temp.Name:'{3}', obj:'{4}'}}", x, y, value, temp.Name, obj);

}

}

class Temp

{

public string Name { get; set; }

}

[复制代码](javascript:void(0);)

# [IoC容器Autofac正篇之依赖注入(七)](http://www.cnblogs.com/sylone/p/6080936.html)

依赖注入，这个专业词我们可以分为两个部分来理解：

   依赖，也就是UML中描述事物之间关系的依赖关系，依赖关系描述了事物A在某些情况下会使用到事物B，事物B的变化会影响到事物A；

   注入，医生通过针头将药物注入到病人体内。注入也就是由外向内注入、灌输一些东西。

   综合上面的说明，依赖注入就是A类依赖B类，B类的实例由外部向A注入而不是由A自己进行实例化或初始化。

## 三种注入方式

 一.**构造器注入**

     类A依赖于类B，类A的构造方法中，有一个参数为类B，在new 类A时会从外部为类B传入实例就是构造注入

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class Program  {      static void Main(string[] args)      {          var b = new B();          var a = new A(b);      }  }    class A  {      private B \_b;      public A(B b)      {          this.\_b = b;      }  }    class B { } |

**构造器注入是默认行为，不需要设置，autofac自动完成了构造注入的工作。**

**二.属性注入**

    修改上面的A类，将变量\_b通过属性暴露出来且删掉有参构造方法，然后看看我们平常写代码时怎么实现属性注入的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Program  {      static void Main(string[] args)      {          var a = new A();          var b = new B();          a.B = b;    //通过属性来注入具有依赖关系的B      }  } |

来看看autofac是怎么进行属性注入的：

      属性注入的所有注入方式都是在注册时定义的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Program  {      static void Main(string[] args)      {          var builder = new ContainerBuilder();          // 通过PropertiesAutowired制定类型A在获取时会自动注入A的属性          builder.RegisterType<A>().PropertiesAutowired();          builder.RegisterType<B>();            var container = builder.Build();          var a = container.Resolve<A>();            Console.Write("Press any key to continue...");          Console.ReadKey();      }  } |

　　使用PropertiesAutowired也只是能指定某个类会自动进行属性注入。PropertiesAutowired方式会自动注入所有可以注入的属性，但是如果只想注入指定几个属性，可以使用除PropertiesAutowired以外的几种注入方式，WithProperty就是其中一种：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Program  {      static void Main(string[] args)      {          var builder = new ContainerBuilder();          builder.RegisterType<A>().WithProperty(new NamedPropertyParameter("B", new B()));          // builder.RegisterType<A>().WithProperty("B", new B());    //效果与上面相同<br>        var container = builder.Build();          var a = container.Resolve<A>();            Console.Write("Press any key to continue...");          Console.ReadKey();      }  } |

　　在注册篇里面有讲到一种[lambda注册](http://www.cnblogs.com/ancupofcoffee/p/5007649.html#autoid-3-2-0)方式，lambda注册时，因为是写lambda表达式进行注册，其lambda内容可以写很多，其中就可以进行属性注入：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var builder = new ContainerBuilder();  builder.Register(c =>  {      var \_a = new A();      \_a.B = new B(); //手动注入      return \_a;  }); |

**三 方法注入**

方法注入有两种方式，也就是属性注入的后两种方式：lambda以及事件

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var builder = new ContainerBuilder();    // lambda  builder.Register(cc =>  {      var \_a = new A();      \_a.MethodInjection(new B());      return \_a;  });    // 事件  builder.RegisterType<A>().OnActivated(e =>  {      e.Instance.MethodInjection(new B());  }); |

　　MethodInjection为A的一个方法且它需要一个类型为B的参数，我们在外部通过方法的方式将B传入，这就是方法注入

#### 尾述

     个人还是推荐使用默认最简单的构造注入，属性注入推荐设置自动属性注入；方法注入还是不怎么推荐的。其实这里的推荐原则是这样的，需要在注册时进行指定注入的方式实际是不太好的，因为后来的人可能不太清楚每个类型的注入规则，还需要到注册的地方进行查看，而且不同人员写的不同，这样容易混乱。而在获取时进行注入，实际也是不太妥的，因为在实际的用法中，我们会将注册类型与接口进行关联，在获取是直接获取接口类型。也正因为我们获取时获取的是接口类型，我们无法保证接口的实际实现是不是具有我们预期的参数

# [IoC容器Autofac正篇之类型关联（服务暴露）(八)](http://www.cnblogs.com/sylone/p/6080938.html)

## 类型关联

**类型关联就是将类挂载到接口(一个或多个)上去，以方便外部以统一的方式进行调用(看下例)。**

#### 一.As关联

    我们在进行手动关联时，基本都是使用As进行关联的.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class Program  {      static void Main(string[] args)      {          var builder = new ContainerBuilder();            builder.RegisterType<Class1>().As<IInterface>();            var container = builder.Build();            IInterface inter = container.Resolve<IInterface>();            Console.WriteLine(inter.Id);            Console.Write("Press any key to continue...");          Console.ReadKey();      }  }    interface IInterface  {      Guid Id { get; }  }    class Class1 : IInterface  {      private Guid \_id;        public Guid Id      {          get { return \_id; }      }  } |

从代码中可以看到，我们直接在类型注册后加了一句As<…>()，然后在使用Resolve时也是用的As的类型。

之前是直接RegisterType注册的类型是Class1，Resolve时也是直接用Class1。

而现在注册的虽然还是Class1，但是获取时却是用IInterface且最后获取到的实例类型是Class1。

这样看有点绕，但我们可以理解为注册了IInterface，但是为IInterface指定了实现。好处在于我们如果希望更改IInterface的实现，只需要修改注册的代码，而不需要修改获取处以及后续使用IInterface实例的代码。

**多关联**一个类可能实现多个接口，如果我们希望Resolve多个接口时获取到的都是那个类型，应该怎么做呢？如果希望多个接口或类型都与同一个类型进行关联，我们可以直接再表达式后面继续As：builder.RegisterType<C1>().As<I1>().As<I2>().As<I3>，如此，Resolve<I1>、Resolve<I2>、Resolve<I3>获取到的都是C1类型实例。

#### 二.自关联AsSelf

#### 不使用As时，RegisterType注册的是什么类型，Resolve就使用什么类型获取，但使用As后就只能使用As的类型进行Resolve。比如前面的例子中，只能够Resolve<IInterface>，而不能Resolve<Class1>，否则抛出异常。但如想在Resolve<Class1>时能够获取到Class1而不抛出异常，应该怎么办呢？

|  |  |
| --- | --- |
| 1  2  3 | // 这两句代码效果相同  builder.RegisterType<Class1>().As<IInterface>().As<Class1>();  builder.RegisterType<Class1>().As<IInterface>().AsSelf(); |

**三 批量关联AsImplementedInerfaces**

AsImplementedInterfaces也就是直接与类型实现的接口进行类型关联

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | interface I1 { }  interface I2 { }  interface I3 { }  interface I4 { }  interface I5 { }  interface I6 { }  interface I7 { }    class C1 : I1, I2, I3, I4, I5, I6, I7  {  } |

　　一个类型实现了7个接口，然后我们希望Resolve这七个中任何一个接口，都能获取到C1实例，多关联代码可以简写为：

|  |  |
| --- | --- |
| 1  2  3  4  5 | builder.RegisterType<C1>()          .As<I1>().As<I2>()          .As<I3>().As<I4>()          .As<I5>().As<I6>()          .As<I7>(); |

　　然而在有AsImplementInterfaces方法后，就能够非常简单：

|  |  |
| --- | --- |
| 1 | builder.RegisterType<C1>().AsImplementedInterfaces(); |

　　在[程序集注册](http://www.cnblogs.com/ancupofcoffee/p/5007649.html#autoid-3-1-0)这种方式中，AsImplementInterfaces更能显示出威力，程序集注册这种方式中，都是批量注册类型的，批量注册的这些类型，它们可能都实现了不同的接口，这样我们没办法为它们一一关联接口，但是通过AsImplementInterfaces方法，可以让所有注册类型自动与实现的接口进行关联。当然，也正因为这点，在使用AsImplementInterfaces时需要注意，是否真的希望与所有接口都进行关联。

**四 总结(3中关联方式)**

#### a.使用AS关联时一个接口只能与一个类型进行关联

 前面将一个类型与多个接口进行关联，让多个接口Resolve结果都是同一个类型实例。这个是不可逆的，也就是一个接口不能与多个类型进行关联，因为这样autofac不知道应该返回哪个类型实例。代码实例：

|  |  |
| --- | --- |
| 1  2 | builder.RegisterType<C1>().As<I>(); // class C1 : I  builder.RegisterType<C2>().As<I>(); // class C2 : I |

　　按上面的代码，最后实际与I接口关联的类型是C2，autofac中按注册顺序，后面注册的会覆盖前面注册的。如果想要阻止这种默认行为，可以在As方法调用用继续调用PreserveExistingDefaults方法，这样，如果之前该接口/类型已经进行关联注册，则此次关联无效：

|  |  |
| --- | --- |
| 1  2 | builder.RegisterType<C1>().As<I>(); // class C1 : I  builder.RegisterType<C2>().As<I>().PreserveExistingDefaults(); // class C2 : I |

上面代码通过Resolve<I>()获取到C1类型实例。注意c1需要实现或继承I

这里说的一个接口不能关联多个类型，是针对这种常用的注册及关联。其实可以通过Named<>、Meta<>这种方式进行多关联

**b.一个接口与多个类型关联**

builder.RegisterType<**Worker**>().Named<**IPerson**>("worker");

builder.RegisterType<Student>().Named<**IPerson**>("student");

使用Name可以检索服务创建实例：

**IPerson** p = container.ResolveNamed<**IPerson**>("worker");

　　ResolveNamed()只是Resolve()的简单重载，指定名字的服务其实是指定键的服务的简单版本。

**c.键**

有Name的方式很方便，但是值支持字符串，但有时候我们可能需要通过其他类型作键。

　　例如，使用枚举作为key：

　　public enum **DeviceState** { Worker, Student }

　　使用key注册服务，通过Keyed<T>()方法：

　　builder.RegisterType<**Student**>().Keyed<**IPerson**>(**DeviceState**.Student);

**IIndex索引**

　Autofac.Features.Indexed.IIndex<K,V>是Autofac自动实现的一个关联类型。component可以使用IIndex<K,V>作为参数的构造函数从基于键的服务中选择需要的实现。

[复制代码](javascript:void(0);)

复制代码

builder.RegisterType<**Student**>().Keyed<**IPerson**>(**DeviceState**.Student);

using (**IContainer** container = builder.Build())

{

**IIndex**<**DeviceState**, **IPerson**> IIndex = container.Resolve<IIndex<**DeviceState**, **IPerson**>>();

**IPerson** p = IIndex[**DeviceState**.Student];

p.Say(); //输出我是一个学生

}

复制代码

[复制代码](javascript:void(0);)

　　IIndex中第一个泛型参数要跟注册时一致，在例子中是DeviceState枚举。其他两种注册方法没有这样的索引查找功能，这也是为什么设计者推荐Keyed注册的原因之一

# [Autofac 组件、服务、自动装配(2)](http://www.cnblogs.com/sylone/p/6094323.html)

# 一、组件

　　创建出来的对象需要从组件中来获取，组件的创建有如下4种(延续第一篇的Demo，仅仅变动所贴出的代码)方式：

**1、类型创建RegisterType**

　　AutoFac能够通过反射检查一个类型,选择一个合适的构造函数,创造这个对象的实例。主要通过RegisterType<T>() 和 RegisterType(Type) 两个方法以这种方式建立。

　　ContainerBuilder使用 As() 方法将Component封装成了服务使用。

builder.RegisterType<**AutoFacManager**>();

builder.RegisterType<**Worker**>().As<**IPerson**>();

**2、实例创建**

　　builder.RegisterInstance<**AutoFacManager**>(new **AutoFacManager**(new **Worker**()));

**单例**

　　提供示例的方式，还有一个功能，就是不影响系统中原有的单例：

　　builder.RegisterInstance(MySingleton.GetInstance()).ExternallyOwned();　　//将自己系统中原有的单例注册为容器托管的单例

　　这种方法会确保系统中的单例实例最终转化为由容器托管的单例实例。

**3、Lambda表达式创建**

　　Lambda的方式也是Autofac通过反射的方式实现

builder.Register(c => new **AutoFacManager**(c.Resolve<**IPerson**>()));

builder.RegisterType<**Worker**>().As<**IPerson**>();

**4、程序集创建**

　　程序集的创建主要通过RegisterAssemblyTypes()方法实现，Autofac会自动在程序集中查找匹配的类型用于创建实例。

builder.RegisterAssemblyTypes(**Assembly**.GetExecutingAssembly()); //在当前正在运行的程序集中找

builder.RegisterType<**Worker**>().As<**IPerson**>();

**5、泛型注册**

　　泛型注册通过RegisterGeneric() 这个方法实现，在容易中可以创建出泛型的具体对象。

//泛型注册,可以通过容器返回List<T> 如:List<string>,List<int>等等

builder.RegisterGeneric(typeof(List<>)).As(typeof(IList<>)).InstancePerLifetimeScope();

using (IContainer container = builder.Build())

{

IList<string> ListString = container.Resolve<IList<string>>();

}

**6、默认的注册**

　　如果一个类型被多次注册,以最后注册的为准。通过使用PreserveExistingDefaults() 修饰符，可以指定某个注册为非默认值。

[复制代码](javascript:void(0);)

复制代码

ContainerBuilder builder = new ContainerBuilder();

builder.RegisterType<AutoFacManager>();

builder.RegisterType<Worker>().As<IPerson>();

builder.RegisterType<Student>().As<IPerson>().PreserveExistingDefaults(); //指定Student为非默认值

using (IContainer container = builder.Build())

{

AutoFacManager manager = container.Resolve<AutoFacManager>();

manager.Say(); //输出我是一个工人

}

复制代码

[复制代码](javascript:void(0);)

　　如果不使用PreserveExistingDefaults()，那么将输出“我是一个学生”。

# 二、服务

　　Autofac有三种典型的方式区分服务，同一个服务的不同实现可以由类型，名称和键区分。

**1、类型**

　　类型是描述服务的基本方法

　　builder.RegisterType<**Worker**>().As<**IPerson**>(); //IPerson类型的服务和Worker的组件连接起来，这个服务可以创建Worker类的实例

　　并且上面的服务在自动装备中也有效

**AutoFacManager** manager = container.Resolve<**AutoFacManager**>();

**2、名字**

　　服务可以进一步按名字识别。使用这种方式时，用 Named()注册方法代替As()以指定名字:

　　builder.RegisterType<**Worker**>().Named<**IPerson**>("worker");

　　使用Name可以检索服务创建实例：

**IPerson** p = container.ResolveNamed<**IPerson**>("worker");

　　ResolveNamed()只是Resolve()的简单重载，指定名字的服务其实是指定键的服务的简单版本。

**3、键**

　　有Name的方式很方便，但是值支持字符串，但有时候我们可能需要通过其他类型作键。

　　例如，使用枚举作为key：

　　public enum **DeviceState** { Worker, Student }

　　使用key注册服务，通过Keyed<T>()方法：

　　builder.RegisterType<**Student**>().Keyed<**IPerson**>(**DeviceState**.Student);

**显式检索**

　　使用key检索服务以创建实例，通过ResolveKeyd()方法：

**IPerson** p = container.ResolveKeyed<**IPerson**>(**DeviceState**.Student);

 　　ResolveKeyd()会导致容器被当做 Service Locator使用，这是不被推荐的。应该使用IIndex type替代。

**IIndex索引**

　　Autofac.Features.Indexed.IIndex<K,V>是Autofac自动实现的一个关联类型。component可以使用IIndex<K,V>作为参数的构造函数从基于键的服务中选择需要的实现。

[复制代码](javascript:void(0);)

复制代码

builder.RegisterType<**Student**>().Keyed<**IPerson**>(**DeviceState**.Student);

using (**IContainer** container = builder.Build())

{

**IIndex**<**DeviceState**, **IPerson**> IIndex = container.Resolve<IIndex<**DeviceState**, **IPerson**>>();

**IPerson** p = IIndex[**DeviceState**.Student];

p.Say(); //输出我是一个学生

}

复制代码

[复制代码](javascript:void(0);)

　　IIndex中第一个泛型参数要跟注册时一致，在例子中是DeviceState枚举。其他两种注册方法没有这样的索引查找功能，这也是为什么设计者推荐Keyed注册的原因之一。

# 三、自动装配

　　从容器中的可用服务中选择一个构造函数来创造对象，这个过程叫做自动装配。这个过程是通过反射实现的，所以实际上容器创造对象的行为比较适合用在配置环境中。

**1、选择构造函数**

　　Autofac**默认从容器中选择参数最多的构造函数**。如果想要选择一个不同的构造函数，就需要在注册的时候就指定它。

　　builder.RegisterType(typeof(**Worker**)).UsingConstructor(typeof(int));

　　这种写法将指定调用Worker(int)构造函数，如该构造函数不存在则报错。

**2、额外的构造函数参数**

　　有两种方式可以添加额外的构造函数参数，在注册的时候和在检索的时候。在使用自动装配实例的时候这两种都会用到。

**注册时添加参数**

　　使用WithParameters()方法在每一次创建对象的时候将组件和参数关联起来。

**List**<**NamedParameter**> ListNamedParameter = new **List**<**NamedParameter**>() { new **NamedParameter**("Id", 1), new **NamedParameter**("Name", "张三") };

builder.RegisterType<**Worker**>().WithParameters(ListNamedParameter).As<**IPerson**>();

**在检索阶段添加参数**  
　　在Resolve()的时候提供的参数会覆盖所有名字相同的参数，在注册阶段提供的参数会覆盖容器中所有可能的服务。

**3、自动装配**

　　至今为止，自动装配最大的作用就是减少重复配置。许多相似的component无论在哪里注册，都可以通过扫描使用自动装配。

　　builder.RegisterAssemblyTypes(**Assembly**.GetExecutingAssembly()).As<**IPerson**>();

　　在需要的时候，依然可以创建指定的构造函数创建指定的类。

　　builder.Register(c => new Worker(2,"关羽"));

# 四、程序集扫描

**1、扫描**

　　Autofac可以使用约定在程序集中注册或者寻找组件。

　　Autofac可以根据用户指定的规则在程序集中注册一系列的类型，这种方法叫做convention-driven registration或者扫描。

　　builder.RegisterAssemblyTypes(**Assembly**.GetExecutingAssembly()).Where(t => t.Name.EndsWith("Manager"));

　　每个RegisterAssemblyTypes方法只能应用一套规则。如果有多套不同的集合要注册，那就有必要多次调用RegisterAssemblyTypes。

**2、选择类型**

　　RegisterAssemblyTypes接受程序集的集合。默认情况下，程序集中所有公共具体的类都会被注册。

　　如果想要过滤注册的类型，可以使用Where.向下面这样：

　　Where(t => t.Name.EndsWith("Manager"))

　　如果想要排除某些类型，使用Except()：

　　Except<AutoFacManager)>()

　　或者，自定义那些已经排除的类型的注册：

　　Except<**Worker**>(ct =>ct.As<**IPerson**>().SingleInstance())

　　多个过滤器可以同时使用，这时他们之间是AND的关系。

**3、指定服务**

　　RegisterAssemblyTypes这个注册方法是注册单个方法的超集，所以类似As的方法也可以用在程序集中，例如

　　As<IPerson>();

　　As和Named这两个方法额外的重载方法接受lambda表达式来决定服务会提供什么样的类型。

# 五、事件

**1、激活事件**

　　在component生命周期的不同阶段使用事件。

　　Autofac暴露五个事件接口供实例的按如下顺序调用

1. OnRegistered
2. OnPreparing
3. OnActivated
4. OnActivating
5. OnRelease

　　这些事件会在注册的时候被订阅，或者被附加到IComponentRegistration 的时候。

builder.RegisterType<Worker>().As<IPerson>()

.OnRegistered(e => Console.WriteLine("在注册的时候调用!"))

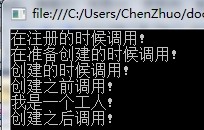
.OnPreparing(e => Console.WriteLine("在准备创建的时候调用!"))

.OnActivating(e => Console.WriteLine("在创建之前调用!"))

.OnActivated(e => Console.WriteLine("创建之后调用!"))

.OnRelease(e => Console.WriteLine("在释放占用的资源之前调用!"));

　　以上示例输出如下：



**OnActivating**

　　组件被创建之前调用，在这里你可以：

1. 将实例转向另外一个或者使用代理封装它
2. 进行属性注入
3. 执行其他初始化工作

**OnActivated**

　　在component被完全创建的时候调用一次。在这个时候你可以执行程序级别的一些工作（这些工作依赖于对象被完全创建）-这种情况很罕见。

**OnRelease**

　　替代component的标准清理方法。实现了IDisposable 接口的标准清理方法（没有标记为ExternallyOwned） 通过调用Dispose 方法。没有实现IDisposable或者被标记为ExternallyOwned的清理方法是一个空函数-不执行任何操作。OnRelease 就是用来覆盖默认的清理行为的。

# 六、属性注入

　　属性注入使用可写属性而不是构造函数参数实现注入。

　　示例：

builder.Register(c => new **AutoFacManager** { person = c.Resolve<**IPerson**>() });

builder.RegisterType<**Worker**>().As<**IPerson**>();

　　为了提供循环依赖（就是当A使用B的时候B已经初始化），需要使用OnActivated事件接口：

builder.Register(c => new **AutoFacManager**()).OnActivated(e => e.Instance.person = e.Context.Resolve<**IPerson**>());

builder.RegisterType<**Worker**>().As<**IPerson**>();

　　通过反射，使用PropertiesAutowired()修饰符注入属性:

builder.RegisterType<**AutoFacManager**>().PropertiesAutowired();

builder.RegisterType<**Worker**>().As<**IPerson**>();

　　如果你预先知道属性的名字和值，你可以使用:

builder.RegisterType<**AutoFacManager**>().WithProperty("person", new **Worker**());

builder.RegisterType<**Worker**>().As<**IPerson**>();

# 七、方法注入

　　可以实现方法注入的方式有两种。

**1、使用Activator**

　　如果你使用委托来激活，只要调用这个方法在激活中

builder.Register(c =>

{

var result = new **AutoFacManager**();

result.SetDependency(c.Resolve<**IPerson**>());

return result;

});

　　注意，使用这种方法，AutoFacManager类里必须要有这个方法：

public void SetDependency(**IPerson** MyPerson)

{

person = MyPerson;

}

**2、使用Activating Handler**

　　如果你使用另外一种激活，比如反射激活，创建激活的事件接口OnActivating，这种方式仅需一行代码：

**builder.Register<AutoFacManager>(c** => new **AutoFacManager**()).OnActivating(e => e.Instance.SetDependency(new **Worker**()));

# 八、Resolve的参数

　　当注册或者检索component的时候可以使用参数。

**1、传递参数给Resolve**

　　Resolve接受可变参数或IEnumerable<T>传入多个值

using (**IContainer** container = builder.Build())

{

**AutoFacManager** manager = container.Resolve<**AutoFacManager**>(new **NamedParameter**("name", "刘备"));

**Console**.WriteLine(manager.Name); //输出 刘备

manager.Say();

}

　　此时，AutoFacManager下必须添加如下构造函数

public AutoFacManager(string name,**IPerson** MyPerson)

{

Name = name;

person = MyPerson;

}

**2、可用的参数类型**

　　Autofac提供几种不同的参数对应策略：

1. NamedParameter ：像上面那样对应的参数名字
2. TypedParameter：对应到参数的类型（必须是具体的类型）
3. ResolvedParameter：灵活的参数匹配
4. NamedParameter 和TypedParameter：只能提供常量参数

**3、从表达式中使用参数**

　　如果使用表达式注册的方式，可以使用第二个可用的委托参数来获得参数。

[复制代码](javascript:void(0);)

复制代码

builder.Register((c, p) => new **AutoFacManager**(p.Named<string>("name"), c.Resolve<**IPerson**>()));

builder.RegisterType<**Worker**>().As<**IPerson**>();

using (**IContainer** container = builder.Build())

{

**AutoFacManager** manager = container.Resolve<**AutoFacManager**>(new **NamedParameter**("name", "刘备"));

**Console**.WriteLine(manager.Name); //输出刘备

manager.Say();

}

复制代码

[复制代码](javascript:void(0);)

# 九、元数据

　　Autofac提供一些机制去创建和使用component的元数据。元数据是存储component中的关于这个component的信息，不需要创建实例也能访问。

**1、在注册的时候添加元数据**

　　值描述的元数据在注册阶段和component联系起来，每个元数据都是一个键值对：

builder.RegisterType<**AutoFacManager**>();

builder.Register(c => new **Worker**()).As<**IPerson**>().WithMetadata("大将", "赵云");

　　用XML文件可以表示为:

[复制代码](javascript:void(0);)

复制代码

　　<component

　　type="ConsoleApplication3.Program.Worker, ConsoleApplication3"

　　service="ConsoleApplication3.Program.IPerson, ConsoleApplication3" >

　　<metadata>

　　<item name="大将" value="赵云" type="System.String" />

　　</metadata>

　　</component>

复制代码

[复制代码](javascript:void(0);)

**2、使用元数据**

　　不用于一般的属性，元数据和component本身是相互独立额度。

　　这使得在运行条件下从很多component中选择一个时非常有用，或者元数据不是component实例的固有属性时。元数据可以表述ITask 执行的时间，或者实现了ICommand的按钮标题。

　　另外一些component可以通过Meta 使用元数据。

# 十、循环依赖

　　循环依赖是指运行期间对象之间的相互依赖

　　目前，Autofac仅仅由处理构造函数/属性依赖的方法。

**1、构造函数/属性依赖**

 　　使用含有属性依赖的类时,使用Activated事件的InjectUnsetProperties。

[复制代码](javascript:void(0);)

复制代码

　　public class **Student**

　　{

　　public Student(**Worker** worker) { }

　　}

　　public class **Worker**

　　{

　　public Student Student { get; set; }

　　}

**ContainerBuilder** cb = new **ContainerBuilder**();

　　cb.Register<**Student**>();

　　cb.Register<**Worker**>().OnActivated(ActivatedHandler.InjectUnsetProperties);

复制代码

[复制代码](javascript:void(0);)

# 十一、泛型

　　给定一个开放的泛型，Autofac会提供一个具体的实现。

　　开放的泛型类型使用泛型服务注册需要给定一个服务类型和一个实现类型。

builder.RegisterGeneric(typeof(**List**<>)).As(typeof(**IList**<>));

using (**IContainer** container = builder.Build())

{

var tt = container.Resolve<**IList**<int>>();

Console.WriteLine(tt.GetType().FullName);

}

　　Autofac关心泛型约束。如果一个有约束的实现类型对服务不可用，那么这个实现类型将被忽略。

# 十二、适配器和装饰器

　　Autofac提供一些机制来实现适配器模式和装饰器模式。

**1、适配器**

　　一个适配器使用一个服务并且适配另外一个。  
　　如果一个适配器在Autofac中被注册，Autofac会为每一个适配服务的实现创建单独的适配器。  
　　这个介绍性的文章描述了适配器在Autofac中是如何实现的。

**2、装饰器**

　　装饰器像适配器一样，在其中封装了一个具体的服务的实现，但是和适配器相比，装饰器暴露出的服务和它封装的一样。

# 十三、实例生命周期

　　实例生命周期决定在同一个服务的每个请求的实例是如何共享的。

　　当请求一个服务的时候，Autofac会返回一个单例 (single instance作用域), 一个新的对象 (per lifetime作用域) 或者在某种上下文环境中的单例。比如 一个线程 或者一个HTTP请求 (per lifetime 作用域)。

　　这条规则适用于显式调用Resolve从容器中检索对象或者满足依赖而隐式实现的对象。

**1、Per Dependency**

　　在其他容器中也称作瞬态或者工厂，使用Per Dependency作用域，服务对于每次请求都会返回互补影响实例。

　　在没有指定其他参数的情况下，这是默认是作用域。

　　builder.RegisterType<**Worker**>();

　　// or

　　builder.RegisterType<**Worker**>().InstancePerDependency();

**2、Single Instance**

　　使用Single Instance作用域，所有对父容器或者嵌套容器的请求都会返回同一个实例。

　　builder.RegisterType<**Worker**>().SingleInstance();

**3、Per Lifetime Scope**

　　这个作用域适用于嵌套的生命周期。一个使用Per Lifetime 作用域的component在一个 nested lifetime scope内最多有一个实例。

　　当对象特定于一个工作单元时，这个非常有用。比如，一个HTTP请求，每一个工作单元都会创建一个nested lifetime，如果在每一次HTTP请求中创建一个nested lifetime，那么其他使用 per-lifetime 的component在每次HTTP请求中只会拥有一个实例。

　　这种配置模型在其他容器中等价于per-HTTP-request, per-thread等。

　　builder.RegisterType<**Worker**>().InstancePerLifetimeScope();

　　ASP.NET和WCF集成中，每一次web请求或者方法调用，InstancePerLifetimeScope会被默认附加到component上。

**4、上下文**

　　上下文作用域和per-lifetime作用域类似，但是对可见性提供更多显示的控制。

　　在大多数程序中，同一层次的容器嵌套代表一个工作单元，如果需要多层嵌套（例如global->request->transation），可以使用标签确保component在多层结构中的某一层共享。

　　builder.RegisterType<**XWorker**>().InstancePerMatchingLifetimeScope(**MyContextHierarchy**.UserSession);

　　提供的标签和生命周期作用域是对应的

　　var userSessionLifetime = container.BeginLifetimeScope();

　　userSessionLifetime.Tag = MyContextHierarchy.UserSession;