# <http://blog.csdn.net/WuLex/article/details/78995697>

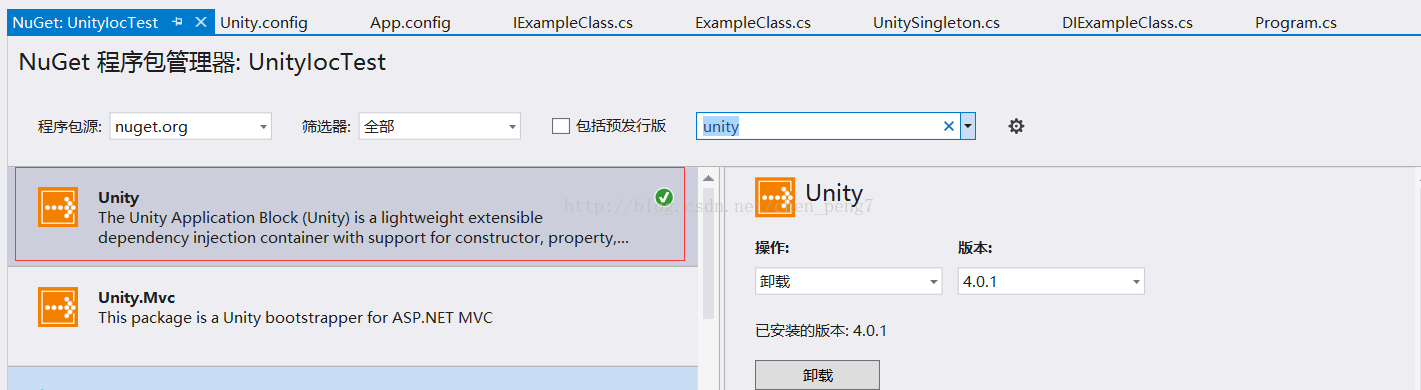
# 1.IOC简介

IOC(Inversion of Control)， 控制反转

DI (Dependency Injection)，依赖注入

IOC的基本概念是：不创建对象，但是描述创建它们的方式。在代码中不直接与对象和服务连接，但在配置文件中描述哪一个组件需要哪一项服务。容器负责将这些联系在一起。

# 2.Unity引入



# 3.创建单例模式容器类

using Microsoft.Practices.Unity;

using Microsoft.Practices.Unity.Configuration;

using System;

using System.Collections.Generic;

using System.Configuration;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace UnityIocTest

{

/// <summary>

/// Unity IOC单例模式

/// </summary>

public class UnitySingleton

{

//单例

private static UnitySingleton instance;

//ioc容器

public IUnityContainer container;

//获取单例

public static UnitySingleton getInstance()

{

if (instance == null || instance.container == null)

{

string configFile = "Unity.config";

var fileMap = new ExeConfigurationFileMap { ExeConfigFilename = configFile };

//从config文件中读取配置信息

Configuration = ConfigurationManager.OpenMappedExeConfiguration(fileMap, ConfigurationUserLevel.None);

//获取指定名称的配置节

UnityConfigurationSection section = (UnityConfigurationSection)configuration.GetSection("unity");

instance = new UnitySingleton()

{

//container = new UnityContainer().LoadConfiguration((UnityConfigurationSection)ConfigurationManager.GetSection("unity"), "MyContainer")

container = new UnityContainer().LoadConfiguration(section, "MyContainer")

//container = new UnityContainer()

};

//instance.container.RegisterType<IExampleClass, ExampleClass>();

}

return instance;

}

//IOC注入实体

public static T GetInstanceDAL<T>()

{

return getInstance().container.Resolve<T>();

}

}

}

此处是将Unity config配置单独成一个文件，方便管理。

# 4.添加IOC相关类

接口类

public interface IExampleClass

{

void DoHelloWord();

}

具体实现类

public class ExampleClass : IExampleClass

{

public void DoHelloWord()

{

Console.WriteLine("Hello Word!");

}

}

实现类扩展

public class DIExampleClass

{

//属性注入

[Dependency]

public IExampleClass example { get; set; }

private IExampleClass testInject;

public void DoWork()

{

example.DoHelloWord();

testInject.DoHelloWord();

}

//方法注入

[InjectionMethod]

public void Initialize(IExampleClass instance)

{

testInject = instance;

}

}

# 5.添加配置文件Unity.config

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

<configuration>

<configSections>

<section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,Microsoft.Practices.Unity.Configuration" />

</configSections>

<unity>

<!--定义类型别名-->

<aliases>

<add alias="IExampleClass" type="UnityIocTest.IExampleClass,UnityIocTest" />

<add alias="ExampleClass" type="UnityIocTest.ExampleClass,UnityIocTest" />

</aliases>

<!--容器-->

<container name="MyContainer">

<!--映射关系-->

<!--<register type="IExampleClass" mapTo="ExampleClass"></register>-->

<register type="UnityIocTest.IExampleClass,UnityIocTest" mapTo="UnityIocTest.ExampleClass,UnityIocTest"></register>

</container>

</unity>

</configuration>

# 6.函数调用

class Program

{

static void Main(string[] args)

{

//var dao = UnitySingleton.GetInstanceDAL<IExampleClass>();

var dao = UnitySingleton.GetInstanceDAL<DIExampleClass>();

dao.DoWork();

Console.ReadLine();

}

}

# 7.结果显示

Hello Word!

Hello Word!

# 8.带参数的[[有参构造注入]](https://www.cnblogs.com/wujy/p/3321478.html)

public interface iOperation

{

string save(string a);

string update(string a);

}

public class ORA : iOperation

{

public string notes { get; set; }

private string name { get; set; }

public ORA()

{

this.notes = "ORA()";

Console.WriteLine("ORA Default Constructor()");

}

public ORA(string msg, int level)

{

this.notes = $"ORA({msg}, {level})";

Console.WriteLine($"ORA Constructor({msg}, {level})");

}

public string save(string a)

{

this.notes += a;

string nstr = "ORA Save: " + this.notes;

Console.WriteLine(nstr);

return nstr;

}

public string update(string a)

{

this.notes += a;

string nstr = "ORA Update: " + this.notes;

Console.WriteLine(nstr);

return nstr;

}

# }

UnityContainer ucon = new UnityContainer();

ucon.RegisterType<iOperation, ORA>("oracle");

ParameterOverrides **parms** = new ParameterOverrides(); - 构造函数的参数

**parms**.Add("msg", "HelloParams"); - key 是参数的名称， value是object类型

**parms**.Add("level", 104);

iOperation ir = ucon.Resolve<iOperation>("oracle", **parms**);

ir.save("Good");

# 

一旦定义了有参数的构造函数， 默认的构造函数则不会成为一种选择：

iOperation ir = ucon.Resolve<iOperation>("oracle"); - 不带参数

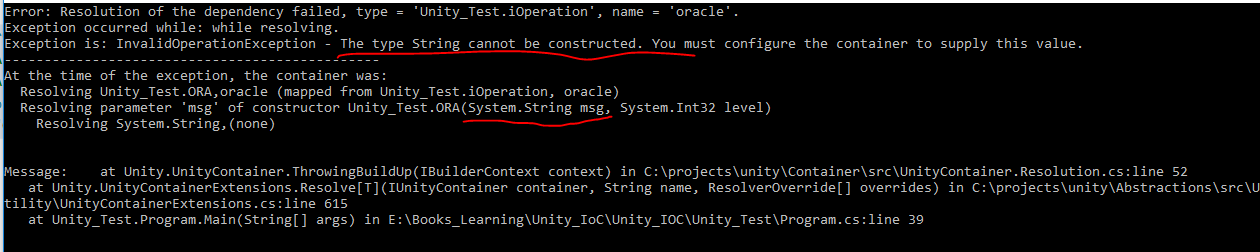
ir.save("Good");

ParameterOverrides parms = new ParameterOverrides();

iOperation ir = ucon.Resolve<iOperation>("oracle", parms); - 或者带空参数

ir.save("Good");

都会出现以下错误：



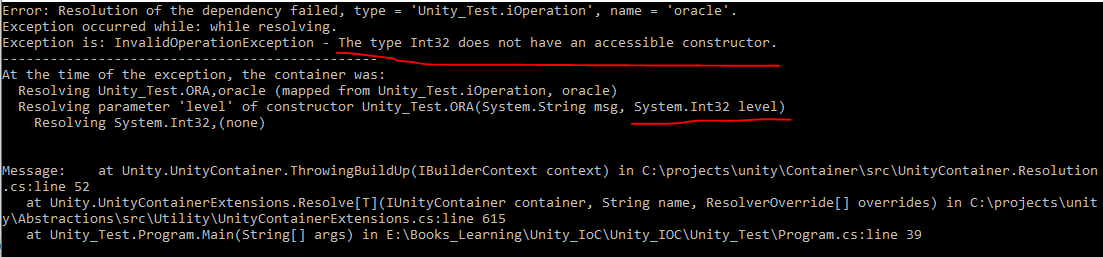
ParameterOverrides parms = new ParameterOverrides();

parms.Add("msg", "HelloParams");

//parms.Add("level", 104); - 参数 level 不定义

iOperation ir = ucon.Resolve<iOperation>("oracle", parms); - 遗漏另外一个参数

ir.save("Good");



<http://blog.csdn.net/WuLex/article/details/78996850>

目录[(?)[-]](http://blog.csdn.net/WuLex/article/details/78996850)

1. [一 我的第一个Unity Demo](http://blog.csdn.net/WuLex/article/details/78996850#t0)
2. [二 Unity的构造函数注入](http://blog.csdn.net/WuLex/article/details/78996850#t1)
3. [三属性注入](http://blog.csdn.net/WuLex/article/details/78996850#t2)
4. [四 初使化器注入自已起的名字](http://blog.csdn.net/WuLex/article/details/78996850#t3)

Unity是Unity是微软patterns& practices组用C#实现的轻量级，可扩展的依赖注入容器，它为方便开发者建立松散耦合的应用程序，有以下优点：

1.简化了对象的创建，特别是针对分层对象结构和依赖关系；

2.需求的抽象，允许开发人员在运行时或配置文件中指定依赖关系，简化横切关注点的管理；

3.推迟为容器配置组件的时机，增加了灵活性；

4.服务定位能力，这使客户能够存储或缓存容器；

5.实例和类型拦截

下载地址: <http://unity.codeplex.com/>

目前最新的版本为 unity 3.0 for .net

以下程序用的是2.1版本

下面开始Unity之旅

## (一) 我的第一个Unity Demo

新建一个控制台应用程序,引用Microsoft.Practices.Unity.dll文件；

新建一个鸟类的接口,定义一个鸟叫的方法;

/// <summary>

/// 鸟类接口

/// </summary>

public interface IBird

{

/// <summary>

/// 讲话

/// </summary>

void Say( );

}

对这个接口进行实现:

/// <summary>

/// 燕子

/// </summary>

public class Swallow : IBird

{

public void Say( )

{

Console.WriteLine("燕子在叫...");

}

}

在Main方法中通过Unity实现IOC反转控制;

static void Main( string[] args )

{

//实例化一个控制器

IUnityContainer unityContainer = new UnityContainer();

//实现注入

unityContainer.RegisterType<IBird, Swallow>();

IBird bird = unityContainer.Resolve<IBird>();

bird.Say();

Console.Read();

}

运行结果:   
这里写图片描述

这个小实例已经实现了简单的IOC控制反转.

当一个接口有两个实现怎么办呢?是不是在加一个类似于下边的代码就行了呢? 下面试一下.

unityContainer.RegisterType<IBird, Swallow>();我们在原有的程序中加一个Sparrow类,实现IBird接口:

public class Sparrow : IBird

{

public void Say()

{

Console.WriteLine("麻雀在叫....");

}

}

Main方法代码:

//实例化一个控制器

IUnityContainer unityContainer = new UnityContainer();

//实现注入

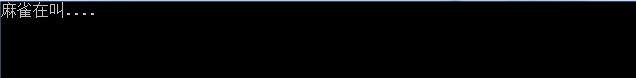
unityContainer.RegisterType<IBird, Swallow>();

unityContainer.RegisterType<IBird, Sparrow>();

IBird bird = unityContainer.Resolve<IBird>();

bird.Say();

Console.Read();

运行一下,结果:   


嗯?这是什么情况,为什么是麻雀在叫..而不是燕子叫呢? 原来

**当一个接口有多个实现,而且没有用别名区分时,就会选择最后一个注入的实现;**

下边给每个注入都加上别名:

//实例化一个控制器

IUnityContainer unityContainer = new UnityContainer();

//实现注入,用别名区分实现

unityContainer.RegisterType<IBird, Swallow>("Swallow");

unityContainer.RegisterType<IBird, Sparrow>("Sparrow");

IBird swallow = unityContainer.Resolve<IBird>("Swallow");

IBird sparrow = unityContainer.Resolve<IBird>("Sparrow");

swallow.Say();

sparrow.Say();

Console.Read();

运行结果:   
这里写图片描述

这才是我们想要的结果,哈哈…..

**当一个接口有多个实现时,需要通过别名区分。**

## (二) Unity的构造函数注入

新建 一个IBirdHome 接口,并对接口进行实现:

/// <summary>

/// 小鸟的家

/// </summary>

public interface IBirdHome

{

IBird Swallow { get; set; }

}

/// <summary>

/// 小鸟的家

/// </summary>

public class BirdHome : IBirdHome

{

public IBird Swallow { get; set; }

public BirdHome(IBird bird)

{

this.Swallow = bird;

}

}

main方法如下:

//实例化一个控制器

IUnityContainer unityContainer = new UnityContainer();

//实现注入

unityContainer.RegisterType<IBird, Swallow>();

unityContainer.RegisterType<IBirdHome, BirdHome>();

IBirdHome birdHome = unityContainer.Resolve<IBirdHome>();

birdHome.Swallow.Say();

Console.Read();

我们只是通过Unity得到了一个IBirdHome实现,但并没有对 IBird Swallow { get; set; } 进行实例化,但结果已经运行出来,并没有报错.

原来这些工作Unity已帮我们做了,我们可以偷个懒了.嘿嘿…….

## (三)属性注入

把BirdHome类的中构造函数去掉,在属性上加上[Dependency]特性

/// <summary>

/// 小鸟的家

/// </summary>

public class BirdHome : IBirdHome

{

/// <summary>

/// 属性注入

/// </summary>

[Dependency]

public IBird Swallow { get; set; }

}

运行结果与上结果一样

## (四) 初使化器注入(自已起的名字)

初使化器注入与构造函数注入相似,但不用写到构造函数里边,而是在初使化方法上加上[InjectionMethod]特性

/// <summary>

/// 初始化器注入

/// </summary>

/// <param name="bird"></param>

[InjectionMethod]

public void Initialize(IBird bird)

{

this.Swallow = bird;

}

运行结果依然是:   
这里写图片描述

<http://blog.csdn.net/WuLex/article/details/78997080>

1.引言

高内聚，低耦合成为一个OO架构设计的一个参考标准。高内聚是一个模块或者一个类中成员跟这个模块或者类的关系尽量高，低耦合是不同模块或者不同类之间关系尽量简单。

拿咱国家举例来说，假如你是中国人。

高内聚：就是说你跟中国亲，关系好，国家会保护你。

低内聚：就是说你跟中国的关系不好，那还怎么混，改天就要批斗你，你就是个问题源。

低耦合：就是说你跟日本的关系非常正常，非常简单，这样的话，就不会被骂汉奸了。

高耦合：就是说你跟日本亲，活该被砸，被游街。

上面例子虚构的，不太贴切，但从中可以看出来高内聚和低耦合的处境是最好的。高内聚是对内部成员跟本身这个模块的关系的描述，低耦合是对成员跟外部模块之间关系的描述。对内对外也是一个相对范围，一个模块里面的小模块之间是耦合，对大模块是聚合，所以说耦合无处不在，我们都要低耦合，Untity就可以帮助我们。

## 2.Unity

Unity是一个IoC容器，用来实现依赖注入（**Dependency Injection，DI**），减少耦合的，Unity出自于伟大的微软。对依赖注入和IoC模式可以查看之前一篇文章IoC模式。   
unity组件网址：<http://unity.codeplex.com/>   
网址中有源码有文档有示例，大家可以下载。我是用的是2.1版本。

看看Unity能做些什么，列举部分如下：

1.Unity支持简单对象创建，特别是分层对象结构和依赖，以简化程序代码。其包含一个编译那些可能存在依赖于其他对象的对象实例机制。

2.Unity支持必要的抽象，其允许开发者在运行时或配置去指定依赖关系同时可以简单的管理横切点（AOP）。

3.Unity增加了推迟到容器组件配置的灵活性。其同样支持一个容器层次的结构。

4.Unity拥有服务定位能力，对于一个程序在许多情况下重复使用组件来分离和集中功能是非常有用的。

5.Unity允许客户端储存或缓存容器。对于在ASP.NET Web applications中开发者将容器持久化于ASP.NET中的session或application中特别有效。

6.Unity拥有拦截能力，其允许开发者通过创建并执行handlers（在方法或属性被调用到达之前）来为已存在的组件增加一个函数，并再次为返回调用结果。

7.Unity可以从标准配置系统中读取配置信息，例如：XML文件，同时使用配置文件来配置容器。

8.Unity支持开发者实现自定义容器扩展，例如：你可以实现方法来允许额外的对象构造和容器特征，例如缓存。

9.Unity允许架构师和开发者在现代化的程序中更简单的实现通用设计模式。

…..

我们项目中什么时候要使用到Unity呢，如下情况：

1.所构建的系统依赖于健全的面向对象原则，但是大量不同的代码交织在一起而难以维护。   
2.构建的对象和类需要依赖其他对象或类。   
3.依赖于复杂的或需要抽象的对象。   
4.希望利用构造函数、方法或属性的调用注入优势。   
5.希望管理对象实例的生命周期。   
6.希望能够在运行时管理并改变依赖关系。   
7.希望在拦截方法或属性调用的时候生成一个策略链或管道处理容器来实现横切（AOP）任务。   
8.希望在Web Application中的回发操作时能够缓存或持久化依赖关系。

…………..

先看看Unity容器IUnityContainer 接口的定义：

//Interface defining the behavior of the Unity dependency injection container.

public interface IUnityContainer : IDisposable

{

//The parent of this container.

IUnityContainer Parent { get; }

//Get a sequence of Microsoft.Practices.Unity.ContainerRegistration that describe

//the current state of the container.

IEnumerable<ContainerRegistration> Registrations { get; }

//Add an extension object to the container.

IUnityContainer AddExtension(UnityContainerExtension extension);

//Run an existing object through the container and perform injection on it.

object BuildUp(Type t, object existing, string name, params ResolverOverride[] resolverOverrides);

//Resolve access to a configuration interface exposed by an extension.

object Configure(Type configurationInterface);

//Create a child container.

IUnityContainer CreateChildContainer();

//Register an instance with the container.

IUnityContainer RegisterInstance(Type t, string name, object instance, LifetimeManager lifetime);

//Register a type mapping with the container, where the created instances will

//use the given Microsoft.Practices.Unity.LifetimeManager.

IUnityContainer RegisterType(Type from, Type to, string name, LifetimeManager, params InjectionMember[] injectionMembers);

//Remove all installed extensions from this container.

IUnityContainer RemoveAllExtensions();

//Resolve an instance of the requested type with the given name from the container.

object Resolve(Type t, string name, params ResolverOverride[] resolverOverrides);

//Return instances of all registered types requested.

IEnumerable<object> ResolveAll(Type t, params ResolverOverride[] resolverOverrides);

//Run an existing object through the container, and clean it up.

void Teardown(object o);

}

IUnityContainer 中有几个方法经常会使用到，如：RegisterInstance，RegisterType，Resolve等等。

Unity为了方便操作，专门为IUnityContainer 提供了许多的扩展方法，部分方法声明如：

/// <summary>

/// 扩展方法 v2.0.50727

/// </summary>

public static class UnityContainerExtensions

{

public static IUnityContainer AddNewExtension<TExtension>(this IUnityContainer container) where TExtension : UnityContainerExtension;

public static T BuildUp<T>(this IUnityContainer container, T existing, params ResolverOverride[] resolverOverrides);

public static T BuildUp<T>(this IUnityContainer container, T existing, string name, params ResolverOverride[] resolverOverrides);

public static object BuildUp(this IUnityContainer container, Type t, object existing, params ResolverOverride[] resolverOverrides);

public static TConfigurator Configure<TConfigurator>(this IUnityContainer container) where TConfigurator : IUnityContainerExtensionConfigurator;

public static bool IsRegistered<T>(this IUnityContainer container);

public static bool IsRegistered<T>(this IUnityContainer container, string nameToCheck);

public static bool IsRegistered(this IUnityContainer container, Type typeToCheck);

public static bool IsRegistered(this IUnityContainer container, Type typeToCheck, string nameToCheck);

public static IUnityContainer RegisterInstance<TInterface>(this IUnityContainer container, TInterface instance);

public static IUnityContainer RegisterInstance<TInterface>(this IUnityContainer container, string name, TInterface instance);

public static IUnityContainer RegisterInstance<TInterface>(this IUnityContainer container, TInterface instance, LifetimeManager lifetimeManager);

public static IUnityContainer RegisterInstance(this IUnityContainer container, Type t, object instance);

public static IUnityContainer RegisterInstance<TInterface>(this IUnityContainer container, string name, TInterface instance, LifetimeManager lifetimeManager);

public static IUnityContainer RegisterInstance(this IUnityContainer container, Type t, object instance, LifetimeManager lifetimeManager);

public static IUnityContainer RegisterInstance(this IUnityContainer container, Type t, string name, object instance);

public static IUnityContainer RegisterType<T>(this IUnityContainer container, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType<TFrom, TTo>(this IUnityContainer container, params InjectionMember[] injectionMembers) where TTo : TFrom;

public static IUnityContainer RegisterType<T>(this IUnityContainer container, LifetimeManager, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType<TFrom, TTo>(this IUnityContainer container, LifetimeManager, params InjectionMember[] injectionMembers) where TTo : TFrom;

public static IUnityContainer RegisterType<TFrom, TTo>(this IUnityContainer container, string name, params InjectionMember[] injectionMembers) where TTo : TFrom;

public static IUnityContainer RegisterType<T>(this IUnityContainer container, string name, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType(this IUnityContainer container, Type t, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType<T>(this IUnityContainer container, string name, LifetimeManager, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType<TFrom, TTo>(this IUnityContainer container, string name, LifetimeManager, params InjectionMember[] injectionMembers) where TTo : TFrom;

public static IUnityContainer RegisterType(this IUnityContainer container, Type t, LifetimeManager, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType(this IUnityContainer container, Type t, string name, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType(this IUnityContainer container, Type from, Type to, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType(this IUnityContainer container, Type t, string name, LifetimeManager, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType(this IUnityContainer container, Type from, Type to, LifetimeManager, params InjectionMember[] injectionMembers);

public static IUnityContainer RegisterType(this IUnityContainer container, Type from, Type to, string name, params InjectionMember[] injectionMembers);

public static T Resolve<T>(this IUnityContainer container, params ResolverOverride[] overrides);

public static T Resolve<T>(this IUnityContainer container, string name, params ResolverOverride[] overrides);

public static object Resolve(this IUnityContainer container, Type t, params ResolverOverride[] overrides);

public static IEnumerable<T> ResolveAll<T>(this IUnityContainer container, params ResolverOverride[] resolverOverrides);

public static IUnityContainer LoadConfiguration(this IUnityContainer container);

public static IUnityContainer LoadConfiguration(this IUnityContainer container, string containerName);

public static IUnityContainer LoadConfiguration(this IUnityContainer container, UnityConfigurationSection section);

public static IUnityContainer LoadConfiguration(this IUnityContainer container, UnityConfigurationSection section, string containerName);

}

下面来个简单的例子，现在项目中添加Microsoft.Practices.Unity.dll和Microsoft.Practices.Unity.Configuration.dll的引用，准备几个类，如下：

/// <summary>

/// 班级接口

/// </summary>

public interface IClass

{

string ClassName { get; set; }

void ShowInfo();

}

/// <summary>

/// 计科班

/// </summary>

public class CbClass : IClass

{

public string ClassName { get; set; }

public void ShowInfo()

{

Console.WriteLine("计科班：{0}", ClassName);

}

}

/// <summary>

/// 电商班

/// </summary>

public class EcClass : IClass

{

public string ClassName { get; set; }

public void ShowInfo()

{

Console.WriteLine("电商班：{0}", ClassName);

}

}

用编程方式实现注入：

public static void ContainerCodeTest1()

{

IUnityContainer container = new UnityContainer();

//默认注册（无命名）,如果后面还有默认注册会覆盖前面的

container.RegisterType<IClass, CbClass>();

//命名注册

container.RegisterType<IClass, EcClass>("ec");

//解析默认对象

IClass cbClass = container.Resolve<IClass>();

cbClass.ShowInfo();

//指定命名解析对象

IClass ecClass = container.Resolve<IClass>("ec");

ecClass.ShowInfo();

//获取容器中所有IClass的注册的已命名对象

IEnumerable<IClass> classList = container.ResolveAll<IClass>();

foreach (var item in classList)

{

item.ShowInfo();

}

}

配置文件方式：

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

<configuration>

<configSections>

<section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,Microsoft.Practices.Unity.Configuration"/>

</configSections>

<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

<!--定义类型别名-->

<aliases>

<add alias="IClass" type="ConsoleApplication1.UnityDemo.IClass,ConsoleApplication1" />

<add alias="CbClass" type="ConsoleApplication1.UnityDemo.CbClass,ConsoleApplication1" />

<add alias="EcClass" type="ConsoleApplication1.UnityDemo.EcClass,ConsoleApplication1" />

</aliases>

<!--容器-->

<container name="FirstClass">

<!--映射关系-->

<register type="IClass" mapTo="CbClass"></register>

<register type="IClass" name="ec" mapTo="EcClass"></register>

</container>

</unity>

</configuration>

public static void ContainerConfigurationTest1()

{

IUnityContainer container = new UnityContainer();

string configFile = "http://www.cnblogs.com/UnityDemo/Constructor/Unity.config";

var fileMap = new ExeConfigurationFileMap { ExeConfigFilename = configFile };

//从config文件中读取配置信息

Configuration =

ConfigurationManager.OpenMappedExeConfiguration(fileMap, ConfigurationUserLevel.None);

//获取指定名称的配置节

UnityConfigurationSection section = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");

//载入名称为FirstClass 的container节点

container.LoadConfiguration(section, "FirstClass");

//解析默认对象

IClass cbClass = container.Resolve<IClass>();

cbClass.ShowInfo();

//指定命名解析对象

IClass ecClass = container.Resolve<IClass>("ec");

ecClass.ShowInfo();

//获取容器中所有IClass的注册的已命名对象

IEnumerable<IClass> classList = container.ResolveAll<IClass>();

foreach (var item in classList)

{

item.ShowInfo();

}

}

效果跟代码方式一样

Unity使用方式基本分为三步：

1. 声明容器IUnityContainer
2. Register注册类型
3. Resolve解析类型对象

## 3.小结

看到IUnityContainer接口声明还是比较简单的，主要用到的方法也不多，另外提供了扩展方法，看起来代码挺多，都是一些方法重载，不要对Unity组件感到恐惧，我们只是简单的使用它，主要是使用IUnityContainer。

<http://blog.csdn.net/WuLex/article/details/78997294>

## 1.引言

Unity的生命周期是注册的类型对象的生命周期，而Unity默认情况下会自动帮我们维护好这些对象的生命周期，我们也可以显示配置对象的生命周期，Unity将按照配置自动管理，非常方便，下面就介绍一下 Unity中内置的生命周期管理器。

## 2.Lifetime Managers生命周期管理

准备以下类关系，用于演示

有2个接口类：IClass（班级接口）和ISubject（科目接口），其分别有2个实现类。看下面一个示例

public static void RegisterInstance()

{

IClass myClass = new MyClass();

IClass yourClass = new YourClass();

//为myClass实例注册默认实例

container.RegisterInstance<IClass>(myClass);

//为yourClass实例注册命名实例，同RegisterType

container.RegisterInstance<IClass>("yourInstance", yourClass);

container.Resolve<IClass>().ShowInfo();

container.Resolve<IClass>("yourInstance").ShowInfo();

}

这段代码很简单，就是通过RegisterInstance方法为已存在的对象进行注册，这样可以通过UnityContainer来管理这些对象实例的生命周期。   
**需要注意的是，使用RegisterInstance来将已存在的 实例注册到UnityContainer中，默认情况下其使用的是ContainerControlledLifetimeManager，这个生命周期 是由UnityContainer来进行管理，UnityContainer会维护一个对象实例的强引用，当你将已存在的实例注册到 UnityContainer后，每次通过Resolve方法获取对象都是同一对象，也就是单件实例（singleton instance），具体有关生命周期相关信息在下面进行介绍。**   
由于RegisterInstance是对已存在的实例进行注册，所以无法通过配置文件来进行配置。有关RegisterInstance方法的其他重载我这边就不详细介绍了，可以点此[查看详细](https://msdn.microsoft.com/en-us/library/ff660894%28PandP.20%29.aspx#registerinstance_methods_tbl)的重载方法介绍。

下面将详细介绍各个Unity的生命周期管理

## 2.1 TransientLifetimeManager

瞬态生命周期，默认情况下，在使用RegisterType进行对象关系注册时如果没有指定生命周期管理器则默认使用这个生命周期管理器，这个生命周期管理器就如同其名字一样，当使用这种管理器的时候，每次通过Resolve或ResolveAll调用对象的时候都会重新创建一个新的对象。

**需要注意的是，使用RegisterInstance对已存在的对象进行关系注册的时候无法指定这个生命周期，否则会报异常。**

代码如下：

public static void TransientLifetimeManagerCode()

{

//以下2种注册效果是一样的

container.RegisterType<IClass, MyClass>();

container.RegisterType<IClass, MyClass>(new TransientLifetimeManager());

Console.WriteLine("-------TransientLifetimeManager Begin------");

Console.WriteLine("第一次调用RegisterType注册的对象HashCode:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("第二次调用RegisterType注册的对象HashCode:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("-------TransientLifetimeManager End------");

}

配置文件如下：

<register type="IClass" mapTo="MyClass">

<lifetime type="transient" />

<!--<lifetime type="SessionLifetimeManager"

value="Session#1" typeConverter="SessionLifetimeConverter" />-->

</register>

如果想在配置文件中在在注册关系的时候更改一个生命周期管理器只需在<register>配置节下新增<lifetime>既可（如果不新增则默认使用TransientLifetimeManager）。

其中<lifetime>有3个参数：

1）type，生命期周期管理器的类型，这边可以选择Unity内置的，也可以使用自定义的，其中内置的生命周期管理器会有智能提示。

2）typeConverter，生命周期管理器转换类，用户自定义一个生命周期管理器的时候所创建一个转换器。

3）value，初始化生命周期管理器的值。

配置文件读取代码：

public static void TransientLifetimeManagerConfiguration()

{

//获取指定名称的配置节

UnityConfigurationSection section = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");

container.LoadConfiguration(section, "First");

Console.WriteLine("-------TransientLifetimeManager Begin------");

Console.WriteLine("第一次调用RegisterType注册的对象HashCode:" +

container.Resolve<IClass>("transient").GetHashCode());

Console.WriteLine("第二次调用RegisterType注册的对象HashCode:" +

container.Resolve<IClass>("transient").GetHashCode());

Console.WriteLine("-------TransientLifetimeManager End------");

}

效果图如下，可以看出每次产生的对象都是不同的：   


## 2.2 ContainerControlledLifetimeManager

容器控制生命周期管理，这个生命周期管理器是RegisterInstance默认使用的生命周期管理器，也就是单件实例，UnityContainer会维护一个对象实例的强引用，每次调用的时候都会返回同一对象，示例代码如下：

public static void ContainerControlledLifetimeManagerCode()

{

IClass myClass = new MyClass();

//以下2种注册效果是一样的

container.RegisterInstance<IClass>("ccl", myClass);

container.RegisterInstance<IClass>("ccl", myClass, new ContainerControlledLifetimeManager());

container.RegisterType<IClass, MyClass>(new ContainerControlledLifetimeManager());

Console.WriteLine("-------ContainerControlledLifetimeManager Begin------");

Console.WriteLine("第一次调用RegisterType注册的对象HashCode:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("第二次调用RegisterType注册的对象HashCode:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("第一次调用RegisterInstance注册的对象HashCode:" +

container.Resolve<IClass>("ccl").GetHashCode());

Console.WriteLine("第二次调用RegisterInstance注册的对象HashCode:" +

container.Resolve<IClass>("ccl").GetHashCode());

Console.WriteLine("-------ContainerControlledLifetimeManager End------");

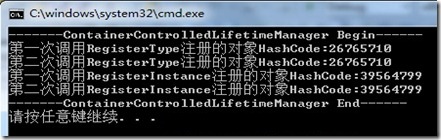
}

配置文件如下：

<register type="IClass" mapTo="MyClass" name="ccl">

<lifetime type="singleton" />

</register>

效果图如下，可以看出每次获取的对象都是同一对象：   


## 2.3 HierarchicalLifetimeManager

分层生命周期管理器，这个管理器类似于ContainerControlledLifetimeManager，也是由UnityContainer来管理，也就是单件实例。不过与ContainerControlledLifetimeManager不 同的是，这个生命周期管理器是分层的，因为Unity的容器时可以嵌套的，所以这个生命周期管理器就是针对这种情况，当使用了这种生命周期管理器，父容器 和子容器所维护的对象的生命周期是由各自的容器来管理，代码如下（RegisterInstance情况也类似，这边就不展示了）

public static void HierarchicalLifetimeManagerCode()

{

container.RegisterType<IClass, MyClass>(new HierarchicalLifetimeManager());

//创建子容器

var childContainer = container.CreateChildContainer();

childContainer.RegisterType<IClass, MyClass>(new HierarchicalLifetimeManager());

Console.WriteLine("-------ContainerControlledLifetimeManager Begin------");

Console.WriteLine("第一次调用父容器注册的对象HashCode:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("第二次调用父容器注册的对象HashCode:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("第一次调用子容器注册的对象HashCode:" +

childContainer.Resolve<IClass>().GetHashCode());

Console.WriteLine("第二次调用子容器注册的对象HashCode:" +

childContainer.Resolve<IClass>().GetHashCode());

Console.WriteLine("-------ContainerControlledLifetimeManager End------");

}

由于配置文件不能配置这种层级效果，所以配置这种生命周期时只需要更改下生命周期名称：

<register type="IClass" mapTo="MyClass" name="hl">

<lifetime type="hierarchical" />

</register>

具体的效果图如下，可以看出父级和子级维护不同对象实例：   


**这边需要提一下的就是，Unity这种分级容器的好处就在于我们可以对于有不同生命周期的对象放在不同的容器中，如果一个子容器被释放，不会影响到其它子容器中的对象，但是如果根节点处父容器释放后，所有的子容器都将被释放。**

## 2.4 PerResolveLifetimeManager

这个生命周期是为了解决循环引用而重复引用的生命周期，先看一下微软官方给出的实例：

public interface IPresenter

{ }

public class MockPresenter : IPresenter

{

public IView View { get; set; }

public MockPresenter(IView view)

{

View = view;

}

}

public interface IView

{

IPresenter Presenter { get; set; }

}

public class View : IView

{

[Dependency]

public IPresenter Presenter { get; set; }

}

从这个例子中可以看出，有2个接口IPresenter和IView，还有2个类MockPresenter和View分别实现这2个接口，同时这2个类 中都包含了对另外一个类的对象属性，这个就是一个循环引用，而对应的这个生命周期管理就是针对这种情况而新增的，其类似于 TransientLifetimeManager，但是其不同在于，如果应用了这种生命周期管理器，则在第一调用的时候会创建一个新的对象，而再次通过 循环引用访问到的时候就会返回先前创建的对象实例（单件实例），代码如下：

public static void PerResolveLifetimeManagerCode()

{

var container = new UnityContainer()

.RegisterType<IPresenter, MockPresenter>()

.RegisterType<IView, View>(new PerResolveLifetimeManager());

var view = container.Resolve<IView>();

var tempPresenter = container.Resolve<IPresenter>();

var realPresenter = (MockPresenter)view.Presenter;

Console.WriteLine("-------PerResolveLifetimeManager Begin------");

Console.WriteLine("使用了PerResolveLifetimeManager的对象 Begin");

Console.WriteLine("通过Resolve方法获取的View对象：" +

view.GetHashCode());

Console.WriteLine("View对象中的Presenter对象所包含的View对象：" +

realPresenter.View.GetHashCode());

Console.WriteLine("使用了PerResolveLifetimeManager的对象 End");

Console.WriteLine("");

Console.WriteLine("未使用PerResolveLifetimeManager的对象 Begin");

Console.WriteLine("View对象中的Presenter对象：" +

realPresenter.GetHashCode());

Console.WriteLine("通过Resolve方法获取的View对象：" +

tempPresenter.GetHashCode());

Console.WriteLine("未使用PerResolveLifetimeManager的对象 End");

Console.WriteLine("-------PerResolveLifetimeManager Begin------");

}

从代码中可以看出，在注册对象的时候，仅对IView和View应用了PerResolveLifetimeManager，所以第二次访问View对象会返回同一实例。

具体配置文件如下，**有关构造函数注入和属性注入的内容在下一篇文章中进行介绍**：

<alias alias="IPresenter" type="UnityStudyConsole.IPresenter, UnityStudyConsole" />

<alias alias="IView" type="UnityStudyConsole.IView, UnityStudyConsole" />

<alias alias="MockPresenter" type="UnityStudyConsole.MockPresenter, UnityStudyConsole" />

<alias alias="View" type="UnityStudyConsole.View, UnityStudyConsole" />

<container name="Second">

<register type="IPresenter" mapTo="MockPresenter">

<constructor>

<param name ="view" type="IView">

</param>

</constructor>

</register>

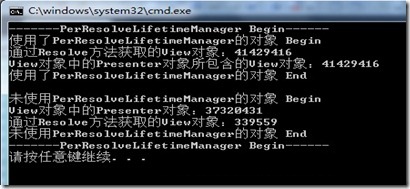
<register type="IView" mapTo="View" >

<lifetime type="perresolve"/>

<property name="Presenter" dependencyType="IPresenter"></property>

</register>

</container>

读取配置文件代码类似于前面其他配置文件读取代码，这里就不展示，具体请看示例代码。   
具体的效果图如下：   


可以看出2次调用View对象的HashCode都是一样的，而Presenter对象的HashCode不同。

## 2.5 PerThreadLifetimeManager

每线程生命周期管理器，就是保证每个线程返回同一实例，具体代码如下：

public static void PerThreadLifetimeManagerCode()

{

container.RegisterType<IClass, MyClass>(new PerThreadLifetimeManager());

var thread = new Thread(new ParameterizedThreadStart(Thread1));

Console.WriteLine("-------PerResolveLifetimeManager Begin------");

Console.WriteLine("默认线程 Begin");

Console.WriteLine("第一调用:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("第二调用:" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("默认线程 End");

thread.Start(container);

}

public static void Thread1(object obj)

{

var tmpContainer = obj as UnityContainer;

Console.WriteLine("新建线程 Begin");

Console.WriteLine("第一调用:" +

tmpContainer.Resolve<IClass>().GetHashCode());

Console.WriteLine("第二调用:" +

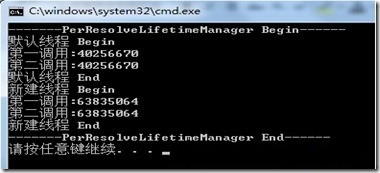
tmpContainer.Resolve<IClass>().GetHashCode());

Console.WriteLine("新建线程 End");

Console.WriteLine("-------PerResolveLifetimeManager End------");

}

有关配置相关的代码与前面的生命周期管理器差不多，这边就不贴代码了，请看示例代码。

具体效果图如下：   


**同时需要注意的是，一般来说不建议在使用RegisterInstance对已存在的对象注册关系时使用PerThreadLifetimeManager，因为此时的对象已经在一个线程内创建了，如果再使用这个生命周期管理器，将无法保证其正确调用。**

## 2.6 ExternallyControlledLifetimeManager

外部控制生命周期管理器，这个 生命周期管理允许你使用RegisterType和RegisterInstance来注册对象之间的关系，但是其只会对对象保留一个弱引用，其生命周期 交由外部控制，也就是意味着你可以将这个对象缓存或者销毁而不用在意UnityContainer，而当其他地方没有强引用这个对象时，其会被GC给销毁 掉。

在默认情况下，使用这个生命周期管理器，每次调用Resolve都会返回同一对象（单件实例），如果被GC回收后再次调用Resolve方法将会重新创建新的对象，示例代码如下：

public static void ExternallyControlledLifetimeManagerCode()

{

container.RegisterType<IClass, MyClass>(new ExternallyControlledLifetimeManager());

var myClass1 = container.Resolve<IClass>();

var myClass2 = container.Resolve<IClass>();

Console.WriteLine("-------ExternallyControlledLifetimeManager Begin------");

Console.WriteLine("第一次调用：" +

myClass1.GetHashCode());

Console.WriteLine("第二次调用：" +

myClass2.GetHashCode());

myClass1 = myClass2 = null;

GC.Collect();

Console.WriteLine("\*\*\*\*GC回收过后\*\*\*\*");

Console.WriteLine("第一次调用：" +

container.Resolve<IClass>().GetHashCode());

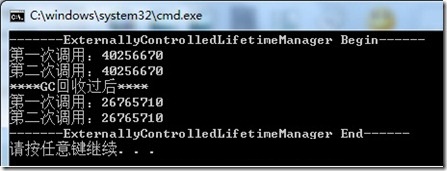
Console.WriteLine("第二次调用：" +

container.Resolve<IClass>().GetHashCode());

Console.WriteLine("-------ExternallyControlledLifetimeManager End------");

}

有关配置相关的代码与前面的生命周期管理器差不多，这边就不贴代码了，请看示例代码。

效果图如下：   


## 3.小结

由于自己在学习Unity，在查找相关资料，在园子里找到这篇，看了一下还是很不错的，于是转载了

<http://blog.csdn.net/WuLex/article/details/78997564>

## 1.引言

上节介绍了，Unity的Lifetime Managers生命周期，Unity具体实现依赖注入包含**构造函数注入、属性注入、方法注入**，所谓注入相当赋值，下面一个一个来介绍。

## 2.构造函数注入

Unity利用Resolve方法解析一个对象，都是调用注册类型的构造函数来初始化的，初始化时，Unity能够控制初始化的值，当然，我们要给Unity提供足够的原料，要不然也是巧妇难无米之炊，下面看一些简单的示例。

先准备几个类如下：

/// <summary>

/// 班级接口

/// </summary>

public interface IClass

{

string ClassName { get; set; }

void ShowInfo();

}

/// <summary>

/// 计科班

/// </summary>

public class CbClass : IClass

{

public string ClassName { get; set; }

public void ShowInfo()

{

Console.WriteLine("计科班：{0}", ClassName);

}

}

/// <summary>

/// 电商班

/// </summary>

public class EcClass : IClass

{

public string ClassName { get; set; }

public void ShowInfo()

{

Console.WriteLine("电商班：{0}", ClassName);

}

}

/// <summary>

/// 学生接口

/// </summary>

public interface IStudent

{

string Name { get; set; }

//就读班级

void ShowInfo();

}

/// <summary>

/// 学生

/// </summary>

public class QlinStudent : IStudent

{

public string Name { get; set; }

private IClass ToClass { get; set; }

public QlinStudent(IClass \_class)

{

ToClass = \_class;

}

public void ShowInfo()

{

Console.WriteLine("{0}就读班级：{1}", Name, ToClass.ClassName);

}

}

是一个班级和学生的结构，现在我们要解析一个学生IStudent，我们看到具体学生类QlinStudent的构造函数需要一个班级接口，当然要给IUnityContainer容器提供这个班级映射还有学生自己的映射，就你要什么东东，首先要提供IUnityContainer什么东东。

## 2.1 默认方式

默认方式跟new一个对象，它会根据你提供的材料，选择一个构造函数，即要有构造器要能访问权限，用Public修饰，构造函数的参数也要提供，即IClass也要能解析，不然就报错了，编程注入方式如下：

public static void ConStructorCodeTest1()

{

IUnityContainer container = new UnityContainer();

//默认注册（无命名）,如果后面还有默认注册会覆盖前面的

container.RegisterType<IClass, CbClass>();

container.RegisterType<IStudent, QlinStudent>();

//解析默认对象

IStudent splitClass = container.Resolve<IStudent>();

splitClass.ShowInfo();

}

配置文件方式 如下：

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

<configuration>

<configSections>

<section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,Microsoft.Practices.Unity.Configuration"/>

</configSections>

<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

<!--引用命名空间-->

<namespace name="ConsoleApplication1.UnityDemo.Constructor" />

<!--引用程序集-->

<assembly name="ConsoleApplication1" />

<!--容器-->

<container name="FirstClass">

<!--映射关系-->

<register type="IClass" mapTo="CbClass"></register>

<register type="IClass" name="ec" mapTo="EcClass"></register>

<register type="IStudent" mapTo="QlinStudent">

</register>

</container>

</unity>

</configuration>

以下是加载配置文件

public static void ConStructorConfigTest1()

{

IUnityContainer container = new UnityContainer();

string configFile = "http://www.cnblogs.com/UnityDemo/Constructor/Unity.config";

var fileMap = new ExeConfigurationFileMap { ExeConfigFilename = configFile };

//从config文件中读取配置信息

Configuration =

ConfigurationManager.OpenMappedExeConfiguration(fileMap, ConfigurationUserLevel.None);

//获取指定名称的配置节

UnityConfigurationSection section = (UnityConfigurationSection)configuration.GetSection("unity");

//载入名称为FirstClass 的container节点

container.LoadConfiguration(section, "FirstClass");

IStudent splitClass = container.Resolve<IStudent>();

splitClass.ShowInfo();

}

## 2.2 指定构造函数

如果构造函数有多个，它也会按照上面那样来初始化一个对象，我们还可以显示用InjectionConstructor特性来指定一个构造函数来解析对象，如下声明：

public class QlinStudent : IStudent

{

private string Name { get; set; }

private IClass ToClass { get; set; }

public QlinStudent()

{

}

[InjectionConstructor]

public QlinStudent(IClass \_class,string name)

{

ToClass = \_class;

Name = name;

}

public void ShowInfo()

{

Console.WriteLine("{0}就读班级：{1}", Name, ToClass.ClassName);

}

}

## 2.3 指定参数依赖的注册名称

构造函数中IClass参数，如果IUnityContainer注册了多个，默认是使用无名称的那个注册，也可以通过Dependency依赖哪个名称来指定哪个来注册，代码，指定ec名称如下：

[InjectionConstructor]

public QlinStudent([Dependency("ec")]IClass \_class)

{

ToClass = \_class;

}

下面注册一个名称为ec的映射，如果没有名称ec的映射将报错

public static void ConStructorCodeTest1()

{

IUnityContainer container = new UnityContainer();

//默认注册（无命名）,如果后面还有默认注册会覆盖前面的

container.RegisterType<IClass, CbClass>();

//命名注册

container.RegisterType<IClass, EcClass>("ec");

container.RegisterType<IStudent, QlinStudent>();

//解析默认对象

IStudent splitClass = container.Resolve<IStudent>();

splitClass.ShowInfo();

}

配置文件方式，代码不变，配置中添加一个 name属性就行，如下：

<container name="FirstClass">

<!--映射关系-->

<register type="IClass" mapTo="CbClass"></register>

<register type="IClass" name="ec" mapTo="EcClass"></register>

<register type="IStudent" mapTo="QlinStudent">

</register>

</container>

## 2.4 指定参数值

构造器中的参数也可以依赖一个指定的类型值，如下代码依赖于EcClass类型，可以让构造函数中可以传入一个具体的类型，这也是构造函数传参数，如下：

public static void ConStructorCodeTest1()

{

IUnityContainer container = new UnityContainer();

//默认注册（无命名）,如果后面还有默认注册会覆盖前面的

container.RegisterType<IClass, CbClass>();

//命名注册

container.RegisterType<IClass, EcClass>("ec");

container.RegisterType<IStudent, QlinStudent>(new InjectionConstructor(new CbClass()));

IStudent splitClass = container.Resolve<IStudent>();

splitClass.ShowInfo();

}

或者注册一个实例对象，如下：

public static void ConStructorCodeTest1()

{

IUnityContainer container = new UnityContainer();

IClass cbClass = new CbClass { ClassName="计科051班" };

//实例注册命名实例

container.RegisterInstance<IClass>("ec", cbClass);

container.RegisterType<IStudent, QlinStudent>();

IStudent splitClass = container.Resolve<IStudent>();

splitClass.ShowInfo();

}

配置文件也可以指定类型依赖，如下，指定EcClass：

<register type="IStudent" mapTo="QlinStudent">

<constructor>

<param name="\_class" type="IClass">

<dependency type="EcClass"/>

</param>

</constructor>

</register>

上面已经介绍了传参数，是用InjectionConstructor类型，现在构造函数，多一个参数，如下：

[InjectionConstructor]

public QlinStudent([Dependency("ec")]IClass \_class, string name)

{

ToClass = \_class;

Name = name;

}

多了一个name参数，那必须为容器IUnityContainer提供这个参数，没有这个原材料，它无法构造，就会报错，如下代码：

public static void ConStructorCodeTest1()

{

IUnityContainer container = new UnityContainer();

container.RegisterType<IStudent, QlinStudent>(new InjectionConstructor(new CbClass() { ClassName = "计科051" }, "Qlin"));

IStudent splitClass = container.Resolve<IStudent>();

splitClass.ShowInfo();

}

注入参数后，也可以下次解析的时候，通过ParameterOverrides类来覆盖原来的参数，改变参数值，如下：

public static void ConStructorCodeTest1()

{

IUnityContainer container = new UnityContainer();

container.RegisterType<IStudent, QlinStudent>(new InjectionConstructor(new CbClass() { ClassName = "计科051" }, "Qlin"));

IStudent student = container.Resolve<IStudent>();

student.ShowInfo();

//覆盖参数解析

IStudent student1 = container.Resolve<IStudent>(new ParameterOverrides()

{

{"\_class",new EcClass(){ ClassName="电商051"}},

{"name","linq"}

});

student1.ShowInfo();

}

## 3.属性注入

就是Unity容器解析对象时，为属性赋值，有操作权限要Public修饰属性。属性注入方式和构造函数注入类似，只需在需要注入的属性上增加一个Dependency特性，Dependency指定一个注册名称name参数用来指定注入对象的名称，属性注入也是伴随着类型初始化时注入的，在解析时自动注入，所以解析时跟以前一样。代码修改如下，在ToClass属性上增加了Dependency特性，来表示这个属性需要注入：

public class QlinStudent : IStudent

{

public string Name { get; set; }

[Dependency("ec")]

public IClass ToClass { get; set; }

public void ShowInfo()

{

Console.WriteLine("{0}就读班级：{1}", Name, ToClass.ClassName);

}

}

代码方式如下：

IUnityContainer container = new UnityContainer();

container.RegisterType<IClass, EcClass>("ec");

container.RegisterType<IStudent, QlinStudent>();

IStudent splitClass = container.Resolve<IStudent>();

splitClass.ShowInfo();

配置文件方式，依赖的<dependency name="ec1" name值 可指定注册时注册的名称：

<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

<!--引用命名空间-->

<namespace name="ConsoleApplication1.UnityDemo.Constructor4" />

<!--引用程序集-->

<assembly name="ConsoleApplication1" />

<!--容器-->

<container name="FirstClass">

<!--映射关系-->

<register type="IClass" mapTo="CbClass">

</register>

<register type="IClass" name="ec1" mapTo="EcClass">

<property name="ClassName" propertyType="System.String" value="电商051" />

</register>

<register type="IStudent" mapTo="QlinStudent">

<property name="ToClass">

<dependency name="ec1" type="EcClass"/>

</property>

</register>

</container>

</unity>

调用效果图：

## 4.方法注入

用public修饰方法，方法注入也是跟构造函数类似代码修改如下

public class QlinStudent : IStudent

{

public string Name { get; set; }

private IClass ToClass { get; set; }

[InjectionMethod]

public void InitClass(IClass \_class)

{

ToClass = \_class;

}

public void ShowInfo()

{

Console.WriteLine("{0}就读班级：{1}", Name, ToClass.ClassName);

}

}

编程方式注入不变，就是初始化时，注入值，如下：

IUnityContainer container = new UnityContainer();

container.RegisterType<IClass, EcClass>();

container.RegisterType<IStudent, QlinStudent>();

IStudent student = container.Resolve<IStudent>();

student.ShowInfo();

配置文件方式：

<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

<!--引用命名空间-->

<namespace name="ConsoleApplication1.UnityDemo.Constructor5" />

<!--引用程序集-->

<assembly name="ConsoleApplication1" />

<!--容器-->

<container name="FirstClass">

<!--映射关系-->

<register type="IClass" mapTo="CbClass">

</register>

<register type="IClass" name="ec1" mapTo="EcClass">

<property name="ClassName" propertyType="System.String" value="电商051" />

</register>

<register type="IStudent" mapTo="QlinStudent">

<property name="Name" propertyType="System.String" value="Qlin" />

<method name="InitClass">

<param name="\_class" type="IClass">

<dependency name="ec1" type="EcClass"/>

</param>

</method>

</register>

</container>

</unity>

## 5.小结

介绍了3种依赖注入方式，平时主要也就用到这么几种，其它还有复杂的像扩展容器等，通过本节，基本知道Unity的使用了。

<http://blog.csdn.net/WuLex/article/details/78997739>

public class UnityIocHelper : IServiceProvider

{

private readonly IUnityContainer \_container;

private static readonly UnityIocHelper dbinstance = new UnityIocHelper("DBcontainer");

private UnityIocHelper(string containerName)

{

UnityConfigurationSection section = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");

\_container = new UnityContainer();

section.Configure(\_container, containerName);

//section.Configure(\_container);

}

public static string GetmapToByName(string containerName,string itype, string name = "")

{

try

{

UnityConfigurationSection section = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");

var \_Containers = section.Containers;

foreach (var \_Container in \_Containers)

{

if (\_Container.Name == containerName)

{

var \_Registrations = \_Container.Registrations;

foreach (var \_Registration in \_Registrations)

{

if (name == "" && string.IsNullOrEmpty(\_Registration.Name) && \_Registration.TypeName == itype)

{

return \_Registration.MapToName;

}

}

break;

}

}

return "";

}

catch

{

throw;

}

}

public static UnityIocHelper DBInstance

{

get { return dbinstance; }

}

public object GetService(Type serviceType)

{

return \_container.Resolve(serviceType);

}

public T GetService<T>()

{

return \_container.Resolve<T>();

}

public T GetService<T>(params ParameterOverride[] obj)

{

return \_container.Resolve<T>(obj);

}

public T GetService<T>(string name, params ParameterOverride[] obj)

{

return \_container.Resolve<T>(name,obj);

}

}

* 66

调用方法：

/// <summary>

/// 连接数据库

/// </summary>

/// <param name="connString">连接字符串</param>

/// <param name="DbType">数据库类型</param>

/// <returns></returns>

public static IDatabase Base(string connString, DatabaseType DbType)

{

DbHelper.DbType = DbType;

return UnityIocHelper.DBInstance.GetService<IDatabase>(new ParameterOverride(

"connString", connString), new ParameterOverride(

"DbType", DbType.ToString()));

}