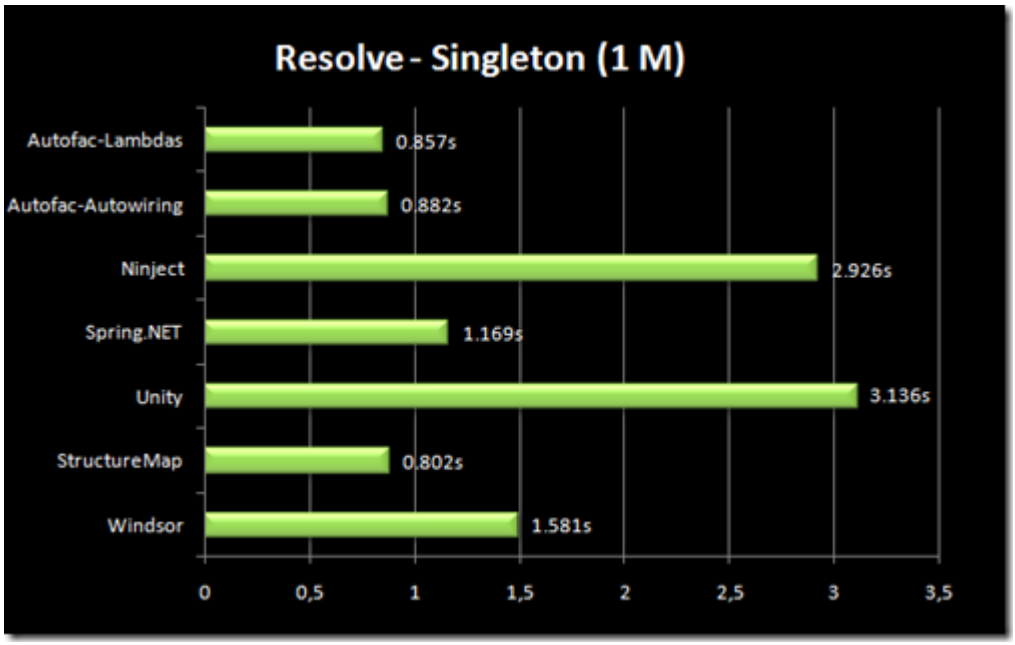
轻量级 IoC 容器之 AutoFac

目录

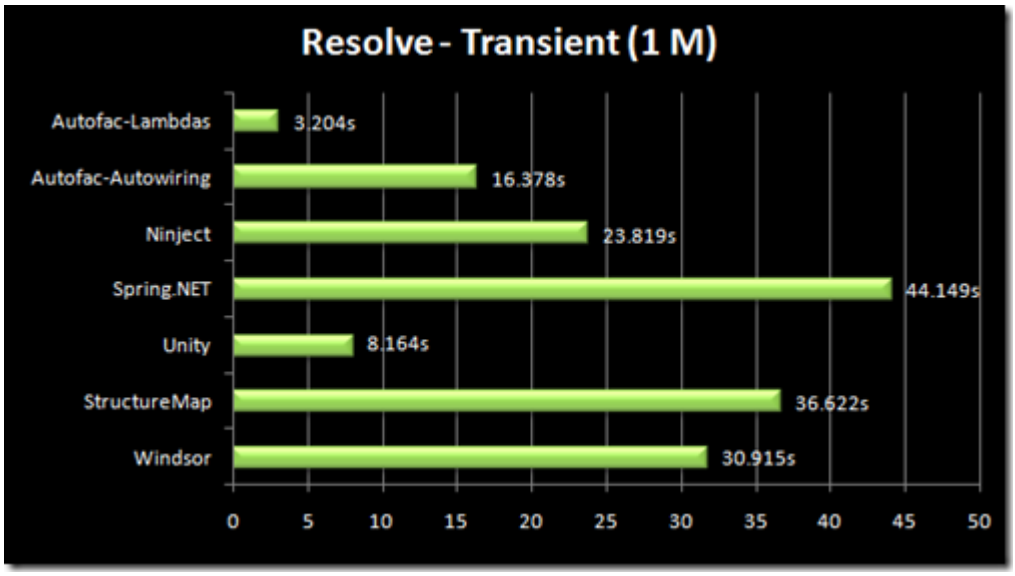
IOC 组件众多，代表性的有 spring.net,castle，autofac，objectbuilder，Unity 等

Autofac 以轻量，便捷高性能，并上手容易的特点，在多个知名的开源项目中得到应用。

性能对比如下
单例模式下获取服务性能对比



实时获取服务实例性能对比(非单例)



表述说明：
服务类型指用于接收服务实例的变量声明类型，如 IService iservice =new StudentSerevice(),

服务类型为 IService。

服务实例则指通过 container.Resolve<T>()返回的实例对象。

组件，代指定义的类型，Interface，class 或者 abstract class 类型

基本用法

一般使用示例如下

```
[Test]
public void TestIOC()
{
    var builder = new ContainerBuilder();
    builder.RegisterType<StudentService>().AsSelf().As<IStudentService>();
    builder.RegisterType<ApplyAuditService>().AsSelf().As<IApplyAuditService>();
    container = builder.Build();
    using (var scope = container.BeginLifetimeScope())
    {
        var iStudentService = container.Resolve<StudentService>();

    }
}
```

类型注册

服务类型的注册，可以通过 lamda 表达式，反射，或者传入一个已经存在的实例，来注册 IOC 管理的服务类型。
注册服务类型的主要方法是 ContainerBuilder 类型提供的三个 RegisterType 的重载方法

```
        public static IRegistrationBuilder<TImplementer, ConcreteReflectionActivatorData, SingleRegistrationStyle> RegisterType<TImplementer>(this
ContainerBuilder builder);
        public static IRegistrationBuilder<object, ConcreteReflectionActivatorData, SingleRegistrationStyle> RegisterType(this ContainerBuilder builder, Type
implementationType);
        public static IRegistrationBuilder<object, ScanningActivatorData, DynamicRegistrationStyle> RegisterTypes(this ContainerBuilder builder, params Type[]
types);
```

基本用法

服务类型注册如下

```
var builder = new ContainerBuilder();

builder.RegisterType<ConsoleLogger>().As<ILogger>();

builder.RegisterType<NHPersonRepository>()
      .As<IFindPerson, IRepository<Person>>();
```

RegisterType<T>泛型方法指定注册的默认服务类型，即通过 Autofac 的 IOC container 获取的服务类型为 T，如果使用 as，则可更改返回的服务类型

如上图，注册的默认服务类型为 ConsoleLogger，使用 as 覆盖已注册 ConsoleLogger 类型为 ILogger 接口类型。

此时如果使用 builder.Resolve<ConsoleLogger>()，运行会报错，因为 ConsoleLogger 类型的注册已经被 ILogger 类型覆盖，则意味着 ConsoleLogger 类型未注册

正确的使用方式如下

```
container.Resolve<ILogger>();
```

以上方法返回的服务实例声明类型是 ILogger，而此服务的真实类型的 ConsoleLogger

通过 RegisterType 给所需服务指定了多个默认类型，又改如何，如下

```
builder.Register<X1>().As<IX>();
builder.Register<X2>().As<IX>();
```

向 IOC 注册了两个服务类型 X1，X2，而且 X1，X2 返回的服务类型都更改 IX 接口类型，那如果通过 container.Resolver<IX>()方法，返回一个服务实例，他真实类型是 X1 还是 X2 呢

以上代码返回的服务实例的真实类型是 X2;

如果有多个类型通过注册来暴露同一个服务，autofac 的处理是，后注册的覆盖之前注册的规则，这就是上面返回服务实例的真实类型为 X2 的原因。

如果要变更此种默认行为，如下

```
builder.Register<X1>().As<IX>();
builder.Register<X2>().As<IX>().PreserveExistingDefaults();
```

此时,container.Resolve<IX>()返回的服务真实类型为 X1

服务注册具体如下：
无参构造器服务注册

```
builder.RegisterType<A>();   // Create A using reflection
builder.RegisterType(typeof(B)); // Non-generic version
```

带参构造器服务注册

```
builder.Register(c => new A(c.Resolve<B>()));
```

类型 A 的构造函数参数是一个类型 B 的实例

亦可如下

```
builder.Register(c => new UserSession(DateTime.Now.AddMinutes(25)));
```

属性注入

```
builder.Register(c => new A(){ MyB = c.ResolveOptional<B>() });
```

```
builder.RegisterType<X>().PropertiesAutowired();
```

以上两种用法一样。
Lamda 表达式中的 c 为 IContainer 类型的实例

如果是属性注入，A 类型和 B 类型服务存在循环依赖，可如下

```
builder.Register(c => new A()).OnActivated(e => e.Instance.B = e.Context.Resolve<B>());
```

先用不带参的构造器构造出 a，然后 OnActivated 事件中为属性 B 注入实例。

如果属性名和属性值已知，可如下

```
builder.WithProperty("propertyName", propertyValue).
```

构造函数注入

如果是带参的构造器类型，需要 IOC 容器根据参数的具体值来返回不同类型的服务实例，此种需求的服务注册如下

```
builder.Register<CreditCard>((c, p) => {
      var accountId = p.Named<string>("accountId");
      if (accountId.StartsWith("9"))
        return new GoldCard(accountId);
      else
        return new StandardCard(accountId);
   });
```

而获取服务实例的方式如下

```
var card = container.Resolve<CreditCard>(new NamedParameter("accountId", "12345"));
```

在以上的示例中注册类型 CreditCard，在 IOC 进行 CreditCard 实例的创建，会执行 lamda 表达式。根据条件，构造出对应的实例。

从此处看出，这种通过 lamda 表达式来延迟创建实体的做法，灵活度很高，并且，比常规的 RegisterType<T>性能要高很多。

为什么呢？常规的 RegisterType 注册，IOC 容器在 Resolve 的时候，需要通过反射来获取实体的参数信息和 ctor 信息，再根据配置(有无指定参数)，来创建一个实例。

而 lamda 表达式里是直接 new 出一个，没有反射的过程。性能自然会高了。

所以就有本文开头的，autofac 两种实体创建方式的自身性能对比。

Autofac 和 spring.net 实体的反射创建是一个非常复杂的过程，都是直接写 Emit，不用深究

方法注入

通过调用方法，完成类型依赖项的注入工作，有两种方式，第一种是 lamda 表达式，另一一种是在 Activating 事件中完成注入，两种方法注入的方式具体如下

使用委托，此方式不直接通过 RegisterType 进行注册，而是在委托中直接注册实体，并完成实体依赖项的注入工作

```
builder.Register(c => {
   var result = new MyObjectType();
   var dep = c.Resolve<TheDependency>();
   result.SetTheDependency(dep);
   return result;
});
```

在激活事件 Activating 中完成注入处理

```
builder
   .Register<MyObjectType>()
   .OnActivating(e => {
      var dep = e.Context.Resolve<TheDependency>();
      e.Instance.SetTheDependency(dep);
   });
```

先按照类型注册，在回调事件 OnActivating 时间完成类型实例的依赖项的注入工作。

实例注入

将单例模式的服务纳入 IOC 的管理，传入一个已存在的实体即可

```
builder.RegisterInstance(MySingleton.Instance).ExternallyOwned();
```

此操作执行后，原来的静态单例服务实例会被移除，取而代之的是由 IOC 容器管理的一个原类型的单例服务实例。而此单例服务的声明实例类型，将会是服务实例的实际类型，而非其他类型，如接口类型或其父类型

注入泛型类型

```
builder.RegisterGeneric(typeof(NHibernateRepository<>))
.As(typeof(IRepository<>))
.InstancePerLifetimeScope();
```

获取泛型服务实例的方法如下

```
var tasks = container.Resolve<IRepository<Task>>();
//将返回类型为NHibernateRepository<Task>的实例
```

IOC 容器会返回一个具体类型服务实例。

亦可明确指定泛型类型

```
builder.RegisterGeneric(typeof(UnivarsityRepository<>))
.AS(typeof(IRepository<Student>))
.InstancePerLefeTimeScope()
```

返回的实例类型为 IRepository<Student>类型，使用 AS 并明确指定泛型类型后，将覆盖 RegisterGeneric 注册的默认类型，实际返回的是 IRepository<Student>类型的实例

委托工厂

Autofac 的委托工厂是一个很牛逼的特性。不好理解。放码

声明式委托工厂的注入

```
16 个引用
public class A<T>
{
    5 个引用
    public T P { get; private set; }

    public delegate A<T> Factory(T p);

    0 个引用
    public A(T p)
    {
        P = p;
    }
}
```

```
    [Test]
    0 个引用
    public void CreateGenericFromFactoryDelegate()
    {
        var cb = new ContainerBuilder();

        cb.RegisterType<A<string>>();
        //new TypedService(typeof(T))
        //将返回类型包装成RegisterGeneratedFactory方法可用的参数
        cb.RegisterGeneratedFactory<A<string>.Factory>(new TypedService(typeof(A<string>)));

        var container = cb.Build();

        var factory = container.Resolve<A<string>.Factory>();
        Assert.IsNotNull(factory);

        var s = "Hello!";
        var a = factory(s);
        Assert.IsNotNull(a);
        Assert.AreEqual(s, a.P);
    }
```

通用委托工厂注入

```
public void CanSetParmeterMappingToPositional()
{
    var builder = new ContainerBuilder();

    int i0 = 32, i0Actual = 0, i1 = 67, i1Actual = 0;

    //注册一object类型，当object被激活器构造产生的时候，执行lamd表达式中的代码
    builder.Register<object>((c, p) =>
    {
        i0Actual = p.Positional<int>(0);
        i1Actual = p.Positional<int>(1);
        return new object();
    });
    //注册委托工厂，指定两个参数类型为int
    //指定返回类型为object
    builder.RegisterGeneratedFactory<Func<int, int, object>>()
        .PositionalParameterMapping();
    var container = builder.Build();
    //获取委托实例
    var generated = container.Resolve<Func<int, int, object>>();
    //返回object结果，IOC处理，执行lamda表达式。
    generated(i0, i1);
    Assert.AreEqual(i0, i0Actual);
    Assert.AreEqual(i1, i1Actual);
}
```

循环依赖

关于循环依赖，有必要说明。
循环依赖存在两种，一种是属性造成的循环依赖，另一种是构造方法造成的循环依赖

属性循环依赖

```
class A
{
   public B BInstance { get; set; }
}

class B
{
   public A AInsttance { get; set; }
}

var cb = new ContainerBuilder();
cb.RegisterType<A>()
   .InstancePerLifetimeScope()
   .PropertiesAutowired(PropertyWiringOptions.AllowCircularDependencies);
cb.RegisterType<B>()
   .InstancePerLifetimeScope()
   .PropertiesAutowired(PropertyWiringOptions.AllowCircularDependencies);
```

Tips：

1. 必须设置依赖类型的属性为可写 set
2. 注册类型时使用 PropertiesAutowired()自动进行属性注入

1.两个相互依赖的类型均不能被注册为 InstancePerDependency，在对象管理作用域中提及，此种管理规则，无论是因依赖项还是因 Resolve 方法的调用，所有产生的实体都将是一个新的并且唯一的实体。

如果如此，a 中依赖 b，b 中依赖 a，在实体创建的过程中，构成死循环，最终方法无法跳出，内存泄露而报异常。

构造器循环依赖

```
class DependsByCtor
{
   public DependsByCtor(DependsByProp dependency) { }
}

class DependsByProp
{
   public DependsByCtor Dependency { get; set; }
}


var cb = new ContainerBuilder();
cb.RegisterType<DependsByCtor>()
   .InstancePerLifetimeScope();
cb.RegisterType<DependsByProperty>()
   .InstancePerLifetimeScope()
   .PropertiesAutowired(PropertyWiringOptions.AllowCircularDependencies);
```

类型 DependsByctor 的构造函数依赖于类型 DependsByProp
而类型 DependsByProp 内又存在对 DependsByctor 的属性依赖

构造器循环依赖，在注册时注意 3 点
2.属性依赖，属性必须可读可写
3.注册类型时，属性依赖需启用 PropertiesAutowired，此设置确保允许循环依赖的存在，避免报错
4.存在循环依赖的两个类型，在注册的时候都不应该设置为 InstancePerDependency，在对象管理作用域中提及，此种管理规则，无论是因依赖项还是因 Resolve 方法的调用，所有产生的实体都将是一个新的并且唯一的实体。
如果如此，a 中依赖 b，b 中依赖 a，在实体创建的过程中，构成死循环，最终方法无法跳出，内存耗尽而报异常。

参数传递

参数类型
在 autofac 传参时支持三种参数类型
NamedParameter 按命名进行参数匹配传递
TypedParameter 按类型进行参数匹配传递，严格的类型匹配
ResolvedParameter 动态的参数传递

NamedParameter 和 TypedParameter 只支持非可变个数的参数
ResolveParameter 支持可变个数的参数

服务注册和服务获取时候，均支持参数传递
服务获取时候，参数传递使用如下

```
public class Student
{
   private string _name
   public Student()
   {
   }

   public Student(string name)
   {
        _name=name;
   }
}
var fred = container.Resolve<Person>(new NamedParameter("name", "yangshuang"));
```

类型定义 Student，有两个构造器，带参构造器 Student(string name)
传递参数，使用键值对的形式传入。

“name”为构造函数的参数名，”yangshuang”为 name 参数的参数值

通过表达式进行服务注册，参数传递使用如下

```
builder.Register((c, p) => new Person(p.Named<string>("name")));
```

此种形式的服务注册前面已经提及，有两个委托参数，c 代表 container 容器，而 p 代表参数集合，类型为 IEnumerable<Parameter>:

以上为命名匹配的参数传递，autofac 在通用委托工厂创建示例的时候，采用的则是按照位置和参数类型进行参数匹配

```
public void CanSetParmeterMappingToPositional()
{
    var builder = new ContainerBuilder();

    int i0 = 32, i0Actual = 0, i1 = 67, i1Actual = 0;

    //注册一object类型，当object被激活器构造产生的时候，执行lamd表达式中的代码
    builder.Register<object>((c, p) =>
    {
        i0Actual = p.Positional<int>(0);
        i1Actual = p.Positional<int>(1);
        return new object();
    });
    //注册委托工厂，指定两个参数类型为int
    //指定返回类型为object
    builder.RegisterGeneratedFactory<Func<int, int, object>>()
        .PositionalParameterMapping();
    var container = builder.Build();
    //获取委托实例
    var generated = container.Resolve<Func<int, int, object>>();
    //返回object结果，IOC处理，执行lamda表达式。
    generated(i0, i1);
    Assert.AreEqual(i0, i0Actual);
    Assert.AreEqual(i1, i1Actual);
}
```

在构造 object 的时候，有一个 positional 按照参数位置的处理过程，这里的 p 是通用委托传过来的参数
主义后面的 return new object(),在这里你可以把参数按位置按取出来，根据自我需要灵活的处理。完全碉堡了。

服务的唯一性标识

Autofac 提供三种方式来标识服务的唯一性，按类型标识，按名称标志，按健值标识

默认的服务标识是按类型区分，如下

```
builder.Register<OnlineState>().As<IDeviceState>();
var r = container.Resolve<IDeviceState>();
```

按名称区分服务

```
builder.Register<OnlineState>().Named<IDeviceState>("online");
var r = container.ResolveNamed<IDeviceState>("online");
```

按键区分

```
public enum DeviceState { Online, Offline }
public class OnlineState : IDeviceState { }
var builder = new ContainerBuilder();
builder.RegisterType<OnlineState>().Keyed<IDeviceState>(DeviceState.Online);
builder.RegisterType<OfflineState>().Keyed<IDeviceState>(DeviceState.Offline);
```

```
var r = container.ResolveKeyed<IDeviceState>(DeviceState.Online);
```

服务的自动装配

毫无疑问，autofac 也提供了自动装配特性，这是 IOC 组件共有的特性。
自动装配，是通过反射技术来实现的，既然用到了反射。那么构造器的选择和处理就会成为一个问题，
Autofac 处理是，优先选择无参或者选择一个参数都能从 container 容器中产生的构造函数来创建服务实例。

比如 container 可以选择无参的构造器，可以一个带参的构造函数，而这个构造函数的参数是可以从 container 中产生的，比如

```
builder.Register(c => new A(c.Resolve<B>()));
```

A 实例的产生，构造函数中有一个 B 实例的参数，而 b 实例可以直接通过 IOC 来获取

亦可通过指定参数的参数类型来通知 Container 通过那个构造函数来产生实例

```
builder.RegisterType(typeof(MyFoo)).UsingConstructor(typeof(int));
```

以上 MyFoo 实例产生， 将会使用 MyFoo(int param)这个构造函数来构造实例

指定带参构造函数创建服务实例，可在 Register 时指定，亦可在 Resolve 时指定，如下

Register 指定，可如下

```
    builder.RegisterType<MyFoo>()
.WithParameters(
    new NamedParameter("message", "Hello!"),
    new NamedParameter("meaning", 42));
```

Resolve 时指定如下

```
var iStudentService = container.Resolve<StudentService>(new NamedParameter("message", "Hello!"),
new NamedParameter("meaning", 42));
```

```
...public static TService Resolve<TService>(this IComponentContext context);
...public static TService Resolve<TService>(this IComponentContext context, IEnumerable<Parameter> parameters);
...public static TService Resolve<TService>(this IComponentContext context, params Parameter[] parameters);
```

Tips：Resolve 时指定的参数值，会覆盖所有同名的参数值
通过 Register 注册时指定的带参构造函数，会覆盖 container 中已注册的并且具有相同构造器的服务，不太好理解。示例如下

```
builder.RegisterType<MyFoo>()
.WithParameters(
    new NamedParameter("message", "Hello!"),
    new NamedParameter("meaning", 42));

    builder.RegisterType<MyFoo>()
.WithParameters(
    new NamedParameter("message", "message new value!"),
    new NamedParameter("meaning", 1200);
```

以上代码的注册，如果从 container 获取一个服务实例。
所使用的构造器是 MyFoo(string a,string b)或者 MyFoo(string a,int c);而后配置的参数 message 和 hello 参数值均覆盖，之前配置的同名参数值，带参的构造函数在使用构造器构造对象的时候，使用的参数值分别为 message new value 和 1200

当有很多类型需要注册为服务时候，可以通过扫描程序集的方式一次性注册所有的服务类型

```
var builder = new ContainerBuilder();
builder.RegisterAssemblyTypes(Assembly.Load("Commands"))
    .As<ICommand>();
```

Autofac 对服务的注册管理可谓强大，采用了跟 Linq 一样优雅的语法，可通过 C#语言直接编码配置。

相比 spring.net 而言，这一点
是不错的。Spring.net 中常见的配置形式是通过 xml 配置文件来管理服务的依赖和加载。

华通项目中，采用的服务配置形式就是 xml，如下

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net"
         xmlns:aop = "http://www.springframework.net/aop"
         xmlns:db="http://www.springframework.net/database"
         xmlns:tx="http://www.springframework.net/tx">

  <object id="UserService" type="Services.Admin.UserControl.UserService,Services" />
  <object id="RoleService" type="Services.Admin.RoleControl.RoleService,Services" />
  <object id="GroupService" type="Services.Admin.GroupControl.GroupService,Services" />
  <object id="RightService" type="Services.Admin.RightControl.RightService,Services" />
  <object id="OrganizationService" type="Services.Admin.OrganizationControl.OrganizationService,Services" />
  <object id="ModuleService" type="Services.Admin.ModuleControl.ModuleService,Services" />
  <object id="CarTypeService" type="Services.Admin.CarTypeControl.CarTypeService,Services" />
  <object id="StationService" type="Services.Admin.StationControl.StationService,Services" />
  <object id="MachineService" type="Services.Admin.MachineControl.MachineService,Services"/>
  <object id="JobService" type="Services.Admin.JobControl.JobService,Services"/>
  <object id="EmployeeService" type="Services.Admin.EmployeeControl.EmployeeService,Services" />
  <object id="MemberService" type="Services.Admin.MemberControl.MemberService,Services" />
  <object id="PayRecordService" type="Services.Admin.MemberControl.PayRecordService,Services" />
  <object id="ConsumerRecordService" type="Services.Admin.MemberControl.ConsumerRecordService,Services" />
  <object id="CarHireCompanyService" type="Services.Admin.CarHireCompanyControl.CarHireCompanyService,Services" />
  <object id="CarService" type="Services.Admin.CarControl.CarService,Services" />
  <object id="ApiService" type="Services.Admin.ApiControl.ApiService,Services" />
  <object id="TicketInfoForUserService" type="Services.Admin.MemberControl.TicketInfoForUserService,Services" />
  <object id="CollectionPayRecordService" type="Services.Admin.MemberControl.CollectionPayRecordService,Services" />
  <object id="CustomerInfoService" type="Services.Admin.CustomerInfoControl.CustomerInfoService,Services" />
  <object id="ScheduleService" type="Services.Admin.ScheduleControl.ScheduleService,Services"/>
  <object id="LogService" type="Services.Log.LogService,Services"/>
</objects>
```

上手曲线略陡，并且易出错。

当然这种配置形式，对发布而言是比较容器管理的，但是，这些配置文件是作为嵌入文件嵌入到程序集中，在 springIOC 容器初始化时，会一次创建所有的配置服务实例。

默认采用单例模式，所有的服务纳入到一个字典里管理。一旦有那个 xml 配置，存在语法错误，或者在调整 xml 的时候，误操作没有正确配置，整个服务都会崩溃。

相比而言，autofac 这一点做的要优。

扫描程序集并注册服务类型

Autofac 可以根据约定来查找和注册程序集中的服务类型,
如下

```
var dataAccess = Assembly.GetExecutingAssembly();
builder.RegisterAssemblyTypes(dataAccess)
.Where(t => t.Name.EndsWith("Repository"))
.AsImplementedInterfaces();
```

程序集中以 Repository 结尾命名的组件(类)，均注册为服务，服务的声明类型为该组件所实现的接口类型

RegisterAssemblyTypes 方法参数接收一个程序集名称数组，默认情况下，会将所有的组件(类)都注册为服务，显然这不是我想要的。

那么需要在注册的时候进行过滤，可以使用 where 扩展方法进行筛选。也有其他的扩展方法提供筛选过滤，如 Namespace 过滤，key 过滤，name 过滤等。

亦可使用 Except 排出特定的类型

```
var dataAccess = Assembly.GetExecutingAssembly();
builder.RegisterAssemblyTypes(dataAccess)
.Where(t => t.Name.EndsWith("Repository"))
.Except<MyUnwantedType>()
```

以上配置代码。在从 Assembly 中扫描的时候，会跳过 MyUnwantedType 类型，亦不会注册此服务类型。

亦可自定义服务的注册规则

```
public void TestRegister()
{
    var builder = new ContainerBuilder();
    var assamlys = Assembly.GetExecutingAssembly();
    builder.RegisterAssemblyTypes(assamlys)
        .Where(p => p.Name.EndsWith("Service"))
        .Except<StudentService>(ct => ct.As<IStudentService>().SingleInstance())
        .AsImplementedInterfaces();
    container = builder.Build();
    using (var scope = container.BeginLifetimeScope())
    {
        var iStudentService = container.Resolve<StudentService>(new NamedParameter("message", "Hello!"),
new NamedParameter("meaning", 42));

    }
}
```

注册以名称以 Service 结尾组件(类)为服务，指定其类型(声明类型)为所实现的接口类型
此注册规则不适用于 StudentService，StudentService 注册为 IStudentService 类型，并且单例

通过程序集扫描来注册服务，有三个方法比较常用

AsImplementedInterfaces()

注册服务类型为其实现的公共接口类型(不包括 IDisposable)，前提是实现了公共接口

AsClosedTypesOf(open)

注册实现了指定接口的类，如果未实现 open 接口，或者未继承 open 类，那么这些类是不会被注册的。

```
Assembly[] assemblies = GetYourAssemblies();

builder.RegisterAssemblyTypes(assemblies)
    .AsClosedTypesOf(typeof(IHandler<>));
```

关于此方法，实现了 IHander 接口的类才会被注册

AsSelf() 默认的服务注册行为，注册所有类，并且服务类型为其本身的真实类型

如果项目部署在 IIS 上，那么再 application_Start 的时候，所有程序集会加载，而后 appdomain 中的程序集会由 IIS 来管理，有程序集的回收和按需加载，

此时，如果有其他 client 进入访问，需要服务时候，如果服务所在的程序集已经被卸载，会有一个 appdomain 临时装载程序集的过程。

也就是意味着 autofac 创建服务的过程中有一个等待

GetExecutingAssembly 获取当前在执行的程序集，但是服务注册的类型有可能是 IService 或者是 Service，如果这些类型都在其他程序集中而不再当前执行的程序集中，则需要临时装入

为解决这个问题，如下

```
var builder2 = new ContainerBuilder();
var assemblies = System.Web.Compilation.BuildManager.GetReferencedAssemblies().Cast<Assembly>();
builder2.RegisterAssemblyTypes(assemblies)
    .AsImplementedInterfaces<>();
container = builder.Build();
```

获取所有引用的程序集，进行一次注册。

服务注册模块化

在实际生产过程中，根据需要变更注册类的构造函数参数或者变更类型的属性值或者属性类型，是很常见的。

为了更好的管理注册服务的变更，需要将服务注册的处理进行模块化

所需的处理是，集成 Module 类或者实现 IModule 接口，如下

```
class ObjectModule : Module
{
    public bool ConfigureCalled { get; private set; }

    protected override void Load(ContainerBuilder builder)
    {
        if (builder == null) throw new ArgumentNullException("builder");
        ConfigureCalled = true;
        builder.RegisterType<object>().SingleInstance();
    }
}
```

关键处理就是重写 Load 方法，在 load 里进行服务注册的处理

使用时如下使用

```
[Test]
public void RegisterAssemblyModulesChainedToRegisterModule()
{
    var assembly = typeof(AComponent).Assembly;
    var builder = new ContainerBuilder();
    builder.RegisterAssemblyModules(assembly).RegisterModule<ObjectModule>();
    var container = builder.Build();

    Assert.IsTrue(container.IsRegistered<AComponent>());
    Assert.IsTrue(container.IsRegistered<BComponent>());
    Assert.IsTrue(container.IsRegistered<object>());
}
```

通过 Module 来管理服务注册有何好处，当项目模块化后，有大量的服务类型需要注册，
此时可根据模块，来划分每一个 Modul 内需要注册的服务类型，创建多个 Module，并在各自的 Module 中实现 Register，然后再 application_start 中统一对各个 Module 进行注册，

如此，各个模块注册的服务管理清晰，已修改和变动

比如定义俩 module

```
public class AModule : IModule
{

}

public class BModule : IModule
{

}
```

```
var assembly = typeof(AComponent).Assembly;
var builder = new ContainerBuilder();
builder.RegisterAssemblyModules(assembly);
```

以上代码会注册所有实现 IModule 接口的类型，AModule 和 BModule 都会被注册
而 Amodule 中实现了所有 A 模块的服务注册
BModule 中实现了所有的 B 模块下的服务注册
如此一来，清晰明了

带泛型参数的RegisterAssemblyModules注册

```
var assembly = typeof(AComponent).Assembly;
var builder = new ContainerBuilder();
builder.RegisterAssemblyModules<AModule>(assembly);
```

以上配置代码，只有 AModule 会被注册
泛型参数 T 严格要求了扫描过程中注册的类型必须是 AModule 类型，因此 BModule 是不会被注册的

如此，A 和 B 模块下的注册服务实现灵活插拔。

另一个类似的重载 Module 注册方法

```
var assembly = typeof(AComponent).Assembly;
var builder = new ContainerBuilder();
builder.RegisterAssemblyModules(typeof(AModule), assembly);
```

从程序集中筛选选出基类型为 AModule 类型的类进行注册。

归根结底，autofac 中的 Module，实际上一个类，这个类实现了对模块内所有所需的注册类型和其依赖类型进行集中打包管理，类似于外观模式 Facade 对 Service 的包装一样

再看一个 Module 注册的使用示例

```
public class CarTransportModule : Module
{
  public bool ObeySpeedLimit { get; set; }

  protected override void Load(ContainerBuilder builder)
  {
    builder.Register(c => new Car(c.Resolve<IDriver>())).As<IVehicle>();

    if (ObeySpeedLimit)
      builder.Register(c => new SaneDriver()).As<IDriver>();
    else
      builder.Register(c => new CrazyDriver()).As<IDriver>();
  }
}

builder.RegisterModule(new CarTransportModule() {
  ObeySpeedLimit = true
});
```

实体的作用域

实体的作用域 scope 决定了发生服务请求时，IOC 容器内各个服务实体的共享形式。(服务实体指的是 IOC 容器通过 Resolve 产生实体或者产生实体时，因依赖项而自动构造的实体)

服务实体作用域允许嵌套

```
        var builder = new ContainerBuilder();
        builder.RegisterType<StudentService>().AsSelf().As<IStudentService>();
        builder.RegisterType<ApplyAuditService>().AsSelf().As<IApplyAuditService>();
        container = builder.Build();
        using (var scope = container.BeginLifetimeScope())
        {
            var iStudentService = container.Resolve<StudentService>(new NamedParameter("message", "Hello!"),
new NamedParameter("meaning", 42));
            using (var sonScope = scope.BeginLifetimeScope())
            {
                //在这里获取son所需的service实体
                using(var grandSonScope = sonScope.BeginLifetimeScope())
                {
                    //在这里获取孙子所需的service实体
                }
            }
```

Autofac IOC 对对象管理的作用域有 7 种，分别为

1.InstancePerDependency()

配置组件创建规则为，每一个依赖项或者通过 Resolve 获取的一个服务实体，都将是一个新的，并且唯一的实体。**没有特殊指定的话这种实体管理方式为 autofac IOC container 对实体默认的管理形式**

以下两种服务实体的获取方式相同

```
builder.RegisterType<X>();

builder.RegisterType<X>().InstancePerDependency();
```

2.InstancePerLifetimeScope()

配置组件创建规则为，在一个 ILifetimeScope 作用范围内，每一个依赖项或者通过 Resolve 获取的实体都将获取一个相同的实体，这个服务实体在 scope 范围内，唯一并且共享。无论通过 Resolve 多少次，只要在这个 scope 内，获取的都将是这个共享的服务实体。

本实体管理规则使用于嵌套的使用周期，scope 内可以嵌套多个子 scope。在一个作用域内，不包含子作用域内，实体呈单例，为所在作用域内共享

对于有依赖项的服务类型而言，在不同的 scope 范围内，Resolve 获取的将是不同的服务实体

此种 IOC 实体管理方式，是因作用域 scope 允许嵌套产生的。

```
builder.RegisterType<X>().InstancePerLifetimeScope();
```

3.InstancePerMatchingLifeTimeScope(params object[] lifetimescopetags)

跟 lifetime scope 类似，Matching life timeScope 提供了对嵌套作用域内的实体更精确的共享控制

配置组件创建规则为，在一个命名并且提供命名名称的 ILifeTimeScope 范围内，无论是因依赖项还是因 Resolve 方法调用产生的实体，都将获取一个相同的服务实体，这个服务实体在此命名 scope 范围内共享

一张图说明一切

```
/// <summary>
/// 命名作用域
/// InstancePerMatchingLifetimeScope使用
/// </summary>
[Test]
public void NamedScopeAndInstancePerMatchingLifetimeScopeTest()
{
    var builder = new ContainerBuilder();
    //本处配置服务实体为为命名作用域内共享
    builder.RegisterType<StudentService>().AsSelf()
        .InstancePerMatchingLifetimeScope("scope");
    container = builder.Build();
    using (var scope = container.BeginLifetimeScope("scope"))
    {
        var iStudentService1 = scope.Resolve<StudentService>();
        using (var sonScope = scope.BeginLifetimeScope())
        {
            var iStudentService2 = sonScope.Resolve<StudentService>();
            Assert.IsTrue(iStudentService1.Equals(iStudentService2));
        }
    }
}
```

如上代码配置，声明共享域的名城为 scope
第一个 using 开启共享域，在这个共享域内，StudentService 实例单例。
iStudentService1 和 iStudentService2 两个实例指向同一个实体。

在 grandsonScope 范围内，无论是因依赖项还是因 Resolve 使用获取的实体都将是 instance。此实例在 grandsonScope 内唯一且共享

在此命名 scope 范围内的其他子 scope，所有因依赖项而产生的服务实体将共享父 scope 内的同类型实体，如果在 scope 层次范围内，并没有找到一个合适的命名 scope，会有一个异常报错。

换句话说如果未找到 grandSonScope，autofac 会抛异常

```
/// <summary>
/// 命名作用域
/// </summary>
[Test]
public void NamedScopeExceptionTest()
{
    var builder = new ContainerBuilder();
    //本处配置服务实体为为命名作用域内共享
    builder.RegisterType<StudentService>().AsSelf().InstancePerMatchingLifetimeScope();
    container = builder.Build();
    using (var scope = container.BeginLifetimeScope())
    {
        var iStudentService = container.Resolve<StudentService>();
        using (var sonScope = scope.BeginLifetimeScope("sonScope"))
        {

        }
    }
}
```

用户代码未处理 DependencyResolutionException

No scope with a Tag matching '' is visible from the scope in which the instance was requested. This generally indicates that a component registered as per-HTTP request is being requested by a SingleInstance() component (or a similar scenario.) Under the web integration always request dependencies from the DependencyResolver.Current or ILifetimeScopeProvider.RequestLifetime, never from the container itself.

疑难解答提示:

获取异常的常规帮助。

以上配置 StudentService 实体管理规则为命名作用域内共享

显然，顶层的作用域仅仅开启了一个匿名作用域，未做命名，在此匿名作用域内使用 container.Resolve<StudentService>()，抛出了未找到命名的作用域异常。

这种服务实体的管理方式，针对一些工作单元特别有用，比如一次 Request，每当一个 Request 产生的时候，可以创建一个作用域，给他提供命名，这样，在这个命名的作用域内，所有服务实体均为单例。

Autofac 与 webform，mvc ，wcf 的集成，其服务实体的管理方式就是采用的此种形式、

4.InstancePerOwned(object serviceKey,Type serviceType)

```
builder.RegisterType<MessageHandler>();
builder.RegisterType<ServiceForHandler>().InstancePerOwned<MessageHandler>();
```

Instance 管理规则，一类服务实体的共享行为依赖于另一种实体类型，相当于为"它"所有。如上在一个 MessageHandle 的生命周期内，所有 SeviceHandle 均为单例共享。ServiceHandle 随着 MessageHandle 消亡而消亡。

在 MessageHandle 的生命周期内，无论因依赖项还是 Resolve 产生的 SeviceHandle 实例，都将获取的是一个相同的实例，此实体为此作用域内所有实体共享

在 Instanceownedscope 范围内的其他子 scope 范围内，因类型依赖要创建实体，并不会创建一个新的实体，而且共享父 scope 范围内的同类型实体

5.InstancePerOwned<T>(object serviceKey)

第四种类型的泛型方法。管理规则同上。

6.InstancePerRequest

一个 web request 内，实例为所有类型共享，无论因依赖还是因 Resolve 的使用，获取的都将是一个相同的实体(相同类型)，仅在 webapi ，webform，mvc 项目中适用

7.Single Instance

实体单例，为所有实体共享

示例

```
//按Module进行服务的注册
var builder4 = new ContainerBuilder();
builder4.RegisterType<StudentService>()
    .InstancePerLifetimeScope()
var container4 = builder4.Build();
```

关于作用域内资源的管理。使用 using 开启一个作用域后，
一旦 using 块结束，服务 Register 未指定 InstanceScope 类型，这些采用默认实体共享规则的服务实体，在此作用域内所有 Resolve 的资源，如果其实现了 Idispose 接口，那么资源会自动 Dispose

如果不适用 using 块，那么你需要手动的对 scope 内的资源进行 Dispose。

作用域 LifetimeScope 自身是实现了 Idispose 接口的，但是 using 结束的时候，不能释放自身。子作用域可由于父作用域使用 using 块来自动管理，或者由开发者手动管理。

Autofac 中的层次管理结构是很灵活的。**但是最简单并且最被推荐的是使用两层管理策略。**Autofac 在容器 build 的时候，会创建一个 root 容器，而我们的工作单元都是从顶层的 root 容器进行一个作用域的创建。

顶层容器的作用域是全局的，生命周期伴随着整个 Application 的生命周期，无需管理

```
using(var scope=container.beginlifetimeScope())
{}
```

如上，这个是第二级的作用域。通常情况下，我们只需要这两层作用域管理即可。

当然 autofac 也支持创建更复杂的作用域层次结构。

1. 给 Scope 指定 tag
2. 注册服务类型时候，声明为 InstancePerMatchingLifetTime

然后实体的生命周期将和指定 tag 的作用域一致， 而 using 的自动管理了。

以上的 IOC 实体的管理策略和 spring.net 是一致的。称呼不一致而已。

服务实体的使用

获取服务后的使用，官方推荐使用 using 来控制 container 的一次生命周期
避免需要 dispose 的资源能及时释放以免资源的过度占用，如下

```
using (var scope = container.BeginLifetimeScope())
{
    var iStudentService = container.Resolve<StudentService>();

}
```

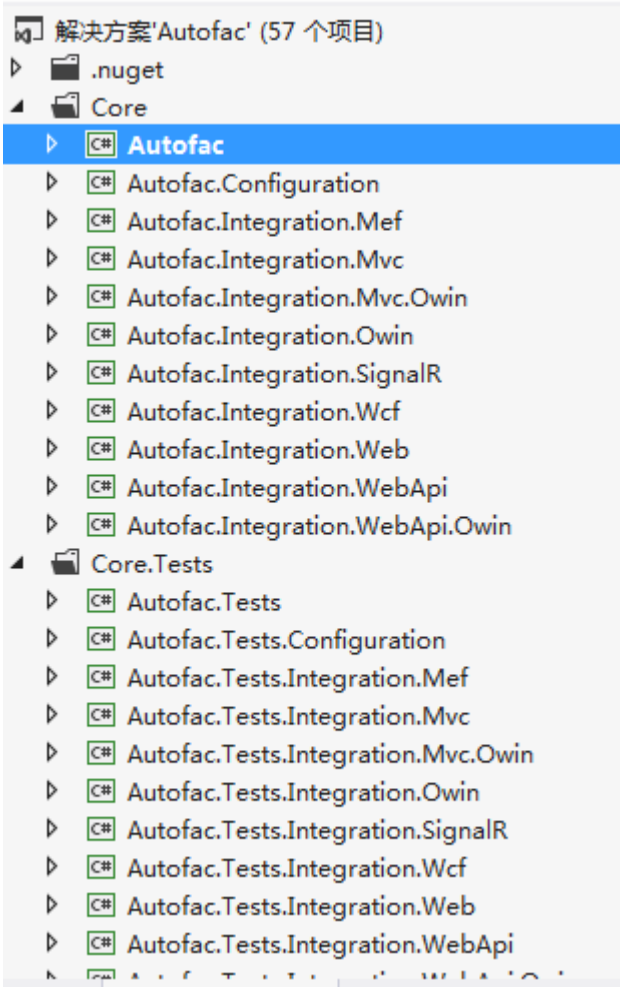
在 using 结束以后，iStudentService 实例所有的依赖项包括其本身都将得到释放或者销毁。

Autofac 项目的源码使用

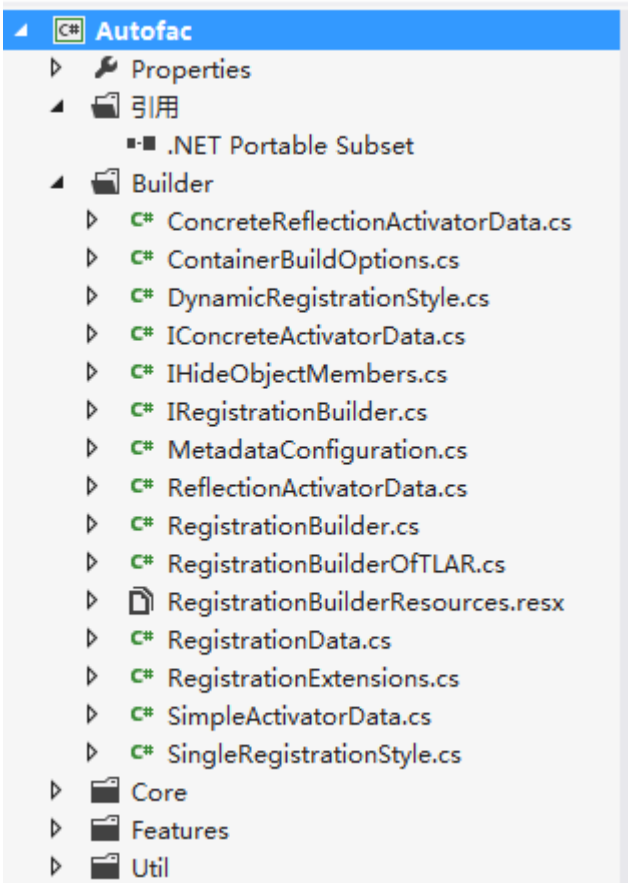
Autofac 文档齐全，但是仍然只介绍了一部分，窥 autofac 全貌还得看源码。学习阶段，对照源码中的对应项目的单元测试和 example 学习。

从 github 上下载 autofac 的所有源码，使用 Visual Studio 2013Premium/Ultimate 版本打开，使用 2010 打开解决方案，会有多个项目无法加载，包括核心库 core 中的 autofac 项目都将无法加载

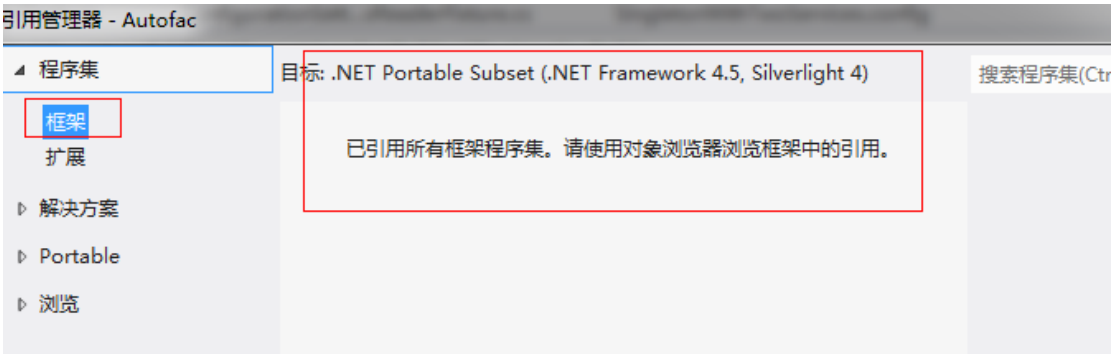
如果用 Vs2012 打开解决方案。直接编译报几百个错误。单元测试无法运行，原因是因为缺少一些库文件。



打开 core 下的 autofac 项目

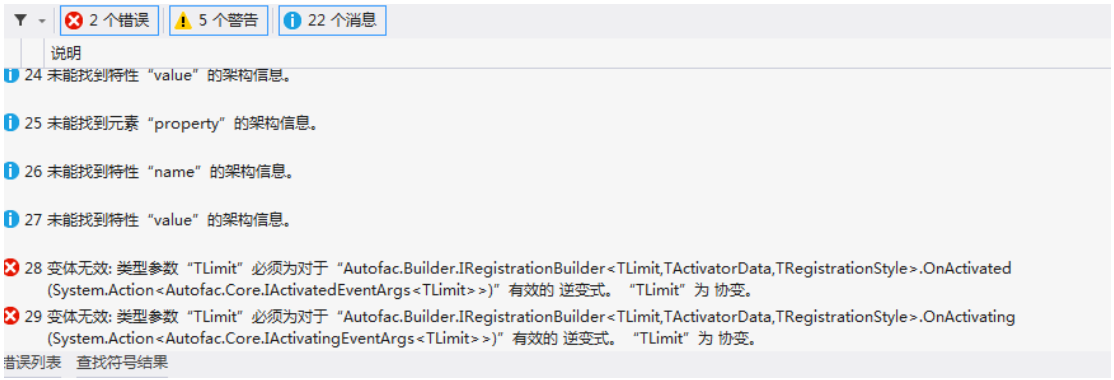


引用有 portable subset，这个是支持跨技术平台的需要，采用 portable subset 管理引用程序集。右键添加引用，勾上几个平台



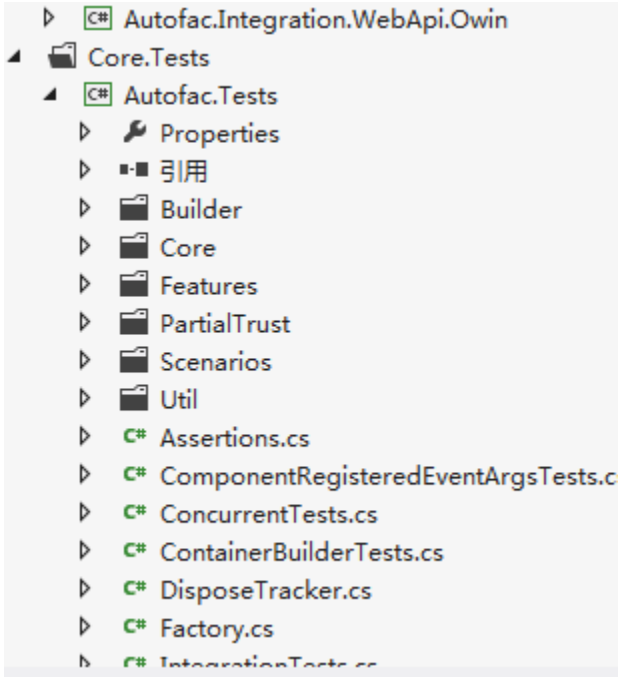
把所有的各个框架的所有程序集勾上

在编译 autofac，错误减少到 3 个

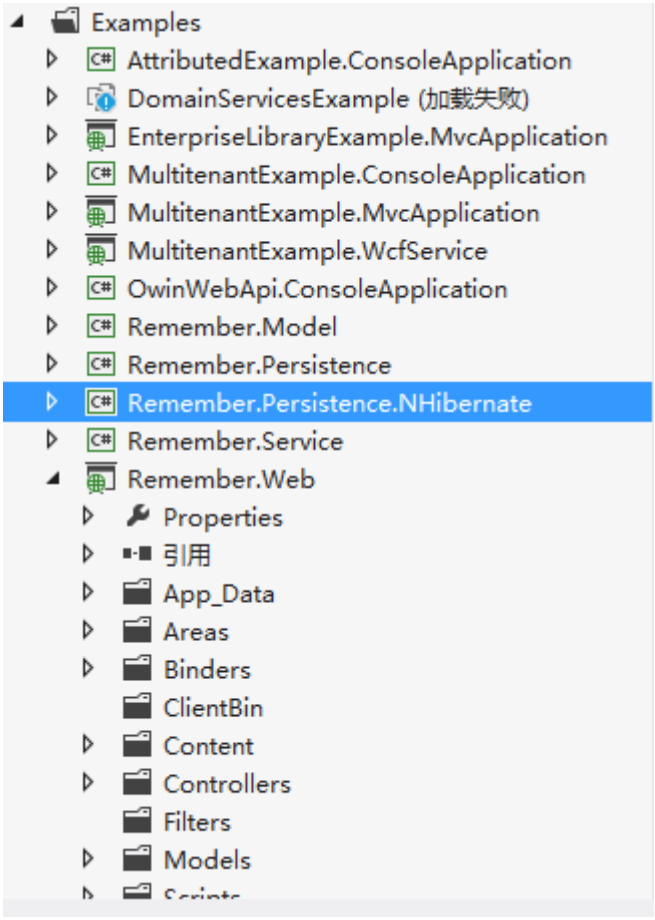


此源码中的逆变错误，尚与开发者联系解决中。原因不明也不敢随意改动

重点看 core.test 下的 Autofac.tests



和



以上是 Autofac 如何与其他项目集成的示例

经沟通咨询，目前 github 上的 autofac 的所有项目均在 VS2013 下开发，并且需要以下 SDK 和工具的支持

1..NET 4.5

2.WCF RIA Services

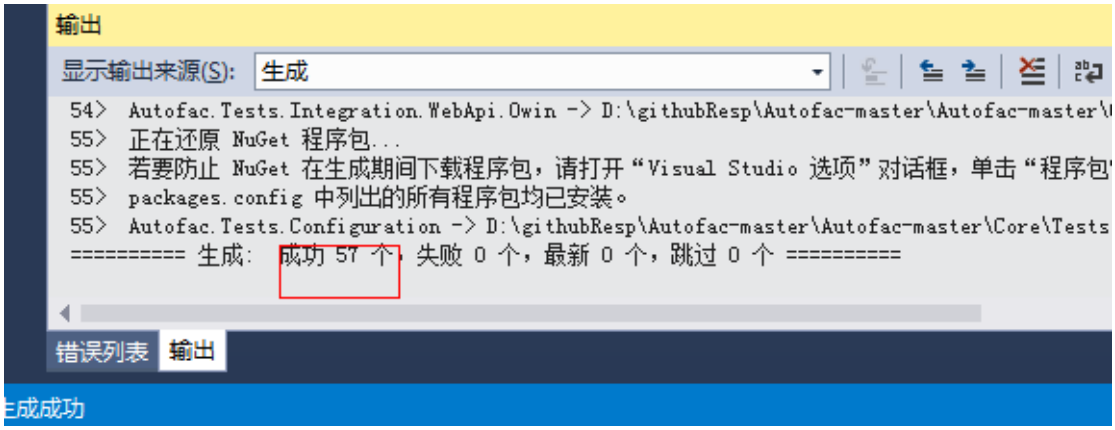
3.Portable Class Library tooling

4.FxCop

5.SQL Server Express

安装 VS2013 ，在安装 sdk 的时候，除了 Office 开发不用沟选，其他包括 lightSwitch 都要勾选。否则有 1 个演示 WCF，Silverlight 和 Autofac 集成的项目无法打开，并且有数个项目编译报错。

编译整个解决方案，全部通过，所有的服务集成和核心库的单元测试均可以正常运行了



关于服务的注册就这些，官网的文档很齐全，可以上官网的 wiki 查看

Wiki 地址

https://github.com/autofac/Autofac/wiki

项目源码地址

https://github.com/autofac/Autofac

2014-7-11