ASP.NET Core

Service

www.bzMe.work

参考：

<https://docs.microsoft.com/zh-cn/aspnet/?view=aspnetcore-2.2#pivot=core>

ASP.NET Core: <https://dotnet.microsoft.com/download/dotnet-core>

.NET文档: <https://docs.microsoft.com/zh-cn/dotnet/>

.NET基金会：<https://www.dotnetfoundation.org/>

PowerShell：<https://github.com/PowerShell/PowerShell/releases>

Windows Management Framework 4.0：

<https://www.microsoft.com/zh-CN/download/details.aspx?id=40855>

**第一章 依赖注入和控制反转**

在讲解服务之前需要先理解依赖注入和控制反转。

**一、依赖注入（**Dependency Injection，DI**）**

ASP.NET Core 支持依赖关系注入(Dependency Injection，DI)软件设计模式，这是一种在类及其依赖关系之间实现控制反转(Inversion of Control，IoC)的技术。ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control(IoC) between classes and their dependencies.

来源：<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1>

后面我们会介绍，依赖关系注入(DI)设计模式在类的设计上必须遵循依赖关系反转原则(Dependency Inversion Principle,DIP)，在对象的管理上采用了反射(Reflection)等技术，对类的依赖反转和对对象的控制反转共同构成了控制反转技术(IoC)。如下图所示：



依赖关系注入的目的是降低代码之间的耦合，确切地说它将显式耦合转变成了隐形耦合，因为它将对象放入到容器中进行统一集中管理，当某个地方需要的时候就从容器中取出注入给使用者。然而受限于编程语言的规则：“要使用必须先声明”这一原则，因此它只能将强耦合转变成弱耦合，强依赖转变成弱依赖。因为只要存在依赖就有耦合，区别在于耦合的强弱和显隐程度。下图对依赖关系注入和正常实例化进行了对比：



依赖注入的优点：

1.降低了类之间的依赖关系和代码耦合的程度；

2.对象在容器中集中进行管理，对象的生命周期由容器来决定；

3.项目组中的开发人员可各自修改类而不相互影响(因为面向接口编程，消除了类之间直接依赖)

4.可采用测试驱动开发模式(TDD)(因为类被集中统一管理，采用注入的方式给使用者)

依赖注入的问题：

1.依赖注入容器除了构建类的依赖树，采用反射来实例化对象，还需要扫描整个源码标记需要注入的位置，显然一个大型的应用程序如果大量地使用依赖注入，在程序启动时必定对性能有影响。

2.依赖注入由于不必显式地实例化对象，这实际上已经打破了传统的面向编程规范(例如人们已经习惯于IMyWork work = new MyWork())，而大量地使用依赖注入，也不利于源码的阅读，例如注入时需要事先知道在什么地方注入，就需要事先在声明的地打上标记(注解)(例如:构造函数注入、属性注入、接口注入)，这些注解会对阅读代码带来不适的感觉，使得你总是担心某个类是否已经被实例化了，而事实上一旦某个类实例没有被注入给使用者，引发异常将很难追踪，因此微软默认只提供最基本的通过构造函数方式注入，充分意识到了注入的滥用不利于系统的后期维护。

**二、控制反转（**Inversion of Control，IoC**）**

参考：

<https://docs.microsoft.com/zh-cn/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion>

前面已经说过，依赖关系注入(DI)是一种软件设计模式，而控制反转(IoC)则是这种设计模式中应用到的控制类及其依赖关系之间的一种技术。控制反转(IoC)技术的核心思想来源于依赖关系反转原则(Dependency Inversion Principle,DIP)，在这种设计原则的思想指导之下，只要遵循依赖关系反转原则就可实现依赖关系注入。既然知道了依赖关系注入(DI)的思想本源是依赖关系反转原则(DIP)，那么让我们了解一下依赖关系反转原则的相关细节。

**1、依赖关系反转原则(DIP)**

**(1)定义**

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应当依赖于抽象。

**(2)含义**

依赖关系反转原则声明了我们应该将高层模块和低层模块解偶，在高层类和低层类之间引入一个抽象层。即应该要针对接口编程，而不是针对实现编程。

**(3)使用场景**

类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A即是高层模块，负责复杂的业务逻辑；类B和类C就是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。如果将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。例如：

|  |
| --- |
| // 定义一个接口  public interface IMy  {  public void DoSomething();  } |

|  |
| --- |
| public class ClassA  {  // 类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。  // 面对细节的多变性，会给程序带来风险。  // 此时就需要对依赖关系进行转换。  ClassB b;  public ClassA()  {  b = new ClassB();  b.DoSomething();  }  } |

|  |
| --- |
| public class ClassA  {  // 如果将类A修改为依赖接口I，类B和类C各自实现接口I。  // 类A通过接口I间接与类B或者类C发生联系，则降低了修改类A的频率。  // 以及与其它类的耦合度。  IMy b;  public ClassA()  {  b = new ClassB();  b.DoSomething();  }  }  public class ClassB : IMy  {  public void DoSomething()  {  throw new NotImplementedException();  }  }  public class ClassC : IMy  {  public void DoSomething()  {  throw new NotImplementedException();  }  } |

**(3)使用特点**

依赖关系反转原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。

以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。

在C#/Java等面向对象编程语言中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

  　 依赖关系反转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应当只实现接口或抽象类中声明过的方法，而不要给出多余的方法，否则将无法调用到在子类中增加的新方法。

在引入抽象层后，系统将具有很好的灵活性，在程序中尽量使用抽象层进行编程，而将具体类写在配置文件中，这样一来，如果系统行为发生变化，只需要对抽象层进行扩展，并修改配置文件，而无须修改原有系统的源代码，在不修改的情况下来扩展系统的功能，满足开闭原则的要求。

**(4)实现案例**

　　依赖关系反转原则的核心思想是面向接口编程，我们用一个绘图的例子来说明面向接口编程比相对于面向实现编程好在什么地方。代码如下：

|  |
| --- |
| // 画图  public class Paint  {  Rectangle p;  public Paint()  {  p = new Rectangle(); // 可以在构造函数中实例化Rectangle  }  public Paint(Rectangle obj)// 或者在构造函数传入Rectangle的对象  {  p = obj;  }  public Draw()  {  p.Draw();  }  }  // 长方形  public class Rectangle  {  public void Draw()  {  Console.WriteLine("画长方形");  }  }  // 三角形  public class Triangle  {  public void Draw()  {  Console.WriteLine("画三角形");  }  } |

假如有一天，要求不画长方形了，要求画三角形，我们只能进行如下的调整：

|  |
| --- |
| // 画图  public class Paint  {  Triangle p;  public Paint()  {  p = new Triangle (); // 可以在构造函数中实例化Triangle  }  public Paint(Triangle obj)// 或者在构造函数传入Triangle的对象  {  p = obj;  }  public Draw()  {  p.Draw();  }  } |

如果以后还要求画圆形、画椭圆形、画平行四边形等等，就要不断地修改Paint，显然这是不好的设计，因为Paint与Rectangle、Triangle之间的耦合性太高，必须降低他们之间的耦合度才行。此时可以抽象出一个IDraw接口对这种强耦合的依赖关系进行转换，如下所示：

|  |
| --- |
| // 绘图接口  public class IDraw  {  public void Draw();  } |
| // 画图  public class Paint  {  IDraw p;  public Paint(IDraw obj) // 现在只需要在构造函数传入IDraw接口  {  p = obj;    }  public Draw()  {  p.Draw();  }  } |
| // 长方形  public class Rectangle: IDraw  {  public void Draw()  {  Console.WriteLine("画长方形");  }  } |
| // 三角形  public class Triangle : IDraw  {  public void Draw()  {  Console.WriteLine("画三角形");  }  } |

我们可以看到这样修改后，无论以后以后画圆形、画椭圆形等等，都不需要再修改Paint类了，以后只需要传入不同的对象给Paint即可，Rectangle类和Triangle类也是各自独立互不影响的，应对变化只需要修改IDraw接口扩展功能即可。

通过这个例子可以知道，代表高层模块的Paint类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖转置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

**(5)注意事项**

在实际编程中，我们一般需要做到如下几点：

(1)低层模块尽量都要有抽象类或接口，或者两者都有。

(2)变量的类型声明尽量是抽象类或接口。

(3)使用继承时遵循里氏替换原则。

依赖转置原则的核心就是要面向接口编程，理解了面向接口编程，也就理解了依赖关系反转原则。

**2、实现依赖关系注入(DI)**

从上面的示例我们会发现，虽然依赖关系反转原则降低了类之间的耦合性，但是每次使用时都要将类的实例传递给使用它的类中，如下例所示：

|  |
| --- |
| // 画图  public class Paint  {  IDraw p;  public Paint(IDraw obj) // 现在只需要在构造函数传入IDraw接口  {  p = obj;  }  public Draw()  {  p.Draw();  }  }  // 调用  IDraw rect = new Rectangle(); // 实例化长方形  Paint p = new Paint(rect); // 将长方形对象传给Paint供其使用 |

能否有更好的方式不需要显示地实例化，而能将对象传递给使用它的类中呢？答案是通过依赖关系注入，这就催生出一种新的设计模式：依赖关系注入(Dependency Injection, DI)。依赖关系注入是指当一个对象要与其他对象发生依赖关系时，通过抽象来注入所依赖的对象，具体运用的技术被称为控制反转(Dependency Inversion Principle,DIP)。控制反转技术由两部分构成：

(1)控制反转要求在类的设计上必须遵循依赖关系反转原则(DIP)，以实现对类的控制反转。采用遵循依赖关系反转原则(DIP)来设计类之间的依赖关系，这种方式就叫类的控制反转。如果不遵守这一原则设计类，那么依赖关系注入设计模式就成了无源之水，没有了实际意义，在实际应用中也就变成了形式上的依赖注入了。

对类的控制反转就是依赖注入的方式，常用的注入方式有三种：

1.Constructor injection，构造注入，是指通过构造函数来传入具体类的对象。

2.Setter Injection，设值注入，是指通过Setter方法来传入具体类的对象。

3.Interface Injection，接口注入，是指通过在接口中声明的业务方法来传入具体类的对象。

(2)控制反转在对象的管理上采用了反射(Reflection)等技术，以实现对对象的控制反转。例如，通常情况下，当一个类实例化成对象以后(例如IMyWork work = new MyWork())，一旦不再需要这个对象时，我们将手动销毁(例如work = null)，而控制反转使得我们不必再关心这个对象的生死，控制权交给了Ioc容器(IoC容器又称DI容器或者服务容器)，由容器来决定并控制对象的生命周期(由用户创建和销毁变成由容器创建和销毁)，这种方式就叫对象的控制反转。

对对象的控制反转就是设置对象的生命周期，在Asp.Net Core中，可以设置三种生命周期：

1.Singleton，单例，程序级的生命周期，即整个应用程序(Application)生命周期内只存在一个实例。

2.Scoped，范围，请求级的生命周期，即每次请求(Request)都创建一个实例。

3.Transient，瞬态，类级的生命周期，即每个类(如控制器)都创建一个实例。



参考：<https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1#framework-provided-services>

以上两部分构成了控制反转技术的基础，以此技术诞生出一种新的软件设计模式：依赖关系注入(Dependency Injection, DI)。

**三、总结**

1.依赖关系注入(dependency injection，DI)是一种软件设计模式，

2.控制反转(Inversion of Control，IoC)是依赖关系注入设计模式运用的技术。

3.控制反转技术在类的设计上：采用了依赖关系反转原则(Dependency Inversion Principle，DIP)；

在对象的管理上：采用了反射(Reflection)等技术来管理对象的生命周期并进行依赖关系的注入。

在理论上，依赖关系注入(dependency injection，DI)设计模式是依赖关系反转原则(Dependency Inversion Principle，DIP)的扩展和延申。

在实际的软件开发中依赖关系注入设计模式给多人并行开发带来了极大的便利，它使得程序可以同时开工，互不影响，参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在测试驱动开发模式 (TDD)就是赖关系反转原则(DIP)或依赖关系注入（DI）成功的应用。

最后我们要注意，如果你准备采用依赖关系注入(DI)设计模式设计你的程序代码，你设计的类应该遵循依赖关系反转原则(DIP)。

**第二章 依赖注入在Asp.Net Core中的应用**

参考：<https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1>

**一、依赖关系注入概述**

首先定义一个类：

|  |
| --- |
| public class MyDependency  {  public MyDependency()  {  }  public Task WriteMessage(string message)  {  Console.WriteLine($"MyDependency.WriteMessage called. Message: {message}");  return Task.FromResult(0);  }  } |

可以创建MyDependency类的实例以使用其WriteMessage方法。如下例所示：

|  |
| --- |
| public class IndexModel : PageModel  {  MyDependency \_dependency = new MyDependency();  public async Task OnGetAsync()  {  await \_dependency.WriteMessage("IndexModel.OnGetAsync created this message.");  }  } |

该类创建并直接依赖于MyDependency实例，这将带来强依赖高耦合，应该避免使用，原因如下：

1.如果要用不同的实现替换MyDependency，必须修改类。

2.如果MyDependency具有依赖关系，则必须由类对其进行配置(例如实例化的时候需要传入依赖的对象)。在具有多个依赖于MyDependency的类的大型项目中，配置代码在整个应用中会变得分散。

3.这种实现很难进行单元测试。

依赖关系注入通过以下方式解决了这些问题：

1.使用接口或基类抽象化依赖关系实现。

2.注册服务容器中的依赖关系。ASP.NET Core提供了一个内置的服务容器[IServiceProvider](https://docs.microsoft.com/zh-cn/dotnet/api/system.iserviceprovider)。服务已在应用的Startup.ConfigureServices方法中注册。

3.将服务注入到使用它的类的构造函数中，框架负责创建依赖关系的实例，并在不再需要时对其进行销毁。

现在采用依赖关系反转原则(DIP)重新设计上面的类：

1.定义IMyDependency接口：

|  |
| --- |
| public interface IMyDependency  {  Task WriteMessage(string message);  } |

2.实现IMyDependency接口：

|  |
| --- |
| public class MyDependency : IMyDependency  {  private readonly ILogger<MyDependency> \_logger;  public MyDependency(ILogger<MyDependency> logger)  {  \_logger = logger;  }  public Task WriteMessage(string message)  {  \_logger.LogInformation("MyDependency.WriteMessage called. Message: {MESSAGE}", message);  return Task.FromResult(0);  }  } |

现在遵循DIP原则的类已经实现好了，需要将其放入服务容器中，以便于进行依赖注入。

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddRazorPages();  services.AddScoped<IMyDependency, MyDependency>();  services.AddTransient<IOperationTransient, Operation>();  services.AddScoped<IOperationScoped, Operation>();  services.AddSingleton<IOperationSingleton, Operation>();  services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));  // OperationService depends on each of the other Operation types.  services.AddTransient<OperationService, OperationService>();  } |

现在就可以在其它地方使用MyDependency类了，服务容器将自动将MyDependency类的实例注入给使用者。

|  |
| --- |
| public class IndexModel : PageModel  {  private readonly IMyDependency \_myDependency;  public IndexModel(  IMyDependency myDependency,  OperationService operationService,  IOperationTransient transientOperation,  IOperationScoped scopedOperation,  IOperationSingleton singletonOperation,  IOperationSingletonInstance singletonInstanceOperation)  {  \_myDependency = myDependency;  OperationService = operationService;  TransientOperation = transientOperation;  ScopedOperation = scopedOperation;  SingletonOperation = singletonOperation;  SingletonInstanceOperation = singletonInstanceOperation;  }  public OperationService OperationService { get; }  public IOperationTransient TransientOperation { get; }  public IOperationScoped ScopedOperation { get; }  public IOperationSingleton SingletonOperation { get; }  public IOperationSingletonInstance SingletonInstanceOperation { get; }  public async Task OnGetAsync()  {  await \_myDependency.WriteMessage("IndexModel.OnGetAsync created this message.");  }  } |

当IndexModel类实例化的时候，服务容器就会将MyDependency的实例自动注入，供\_myDependency使用。

**二、服务生存期**

Asp.Net Core采用服务的方式来提供依赖注入。可以使用以下生存期配置ASP.NET Core服务：

**1.暂时性(Transient)**

暂时生存期服务([AddTransient](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.dependencyinjection.servicecollectionserviceextensions.addtransient))是每次从服务容器进行请求时创建的。这种生存期适合轻量级、无状态的服务。

**2.范围内(Scoped)**

作用域生存期服务([AddScoped](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.dependencyinjection.servicecollectionserviceextensions.addscoped))以每个客户端请求（连接）一次的方式创建。

警告：在中间件内使用有作用域的服务时，请将该服务注入至Invoke或InvokeAsync方法。请不要通过构造函数注入进行注入，因为它会强制服务的行为与单一实例类似。有关详细信息，请参阅 [写入自定义 ASP.NET Core 中间件](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/middleware/write?view=aspnetcore-3.1#per-request-middleware-dependencies)。

**3.单例(Singleton)**

单一实例生存期服务([AddSingleton](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.dependencyinjection.servicecollectionserviceextensions.addsingleton))是在第一次请求时（或者在运行 Startup.ConfigureServices 并且使用服务注册指定实例时）创建的。每个后续请求都使用相同的实例。 如果应用需要单一实例行为，建议允许服务容器管理服务的生存期。不要实现单一实例设计模式并提供用户代码来管理对象在类中的生存期。

警告：从单一实例解析有作用域的服务很危险。 当处理后续请求时，它可能会导致服务处于不正确的状态。

在以下示例中，第一行向 IMyDependency 注册 MyDependency。

第二行没有任何作用，因为 IMyDependency 已有一个已注册的实现。

|  |
| --- |
| services.AddSingleton<IMyDependency, MyDependency>();  // The following line has no effect:  services.TryAddSingleton<IMyDependency, DifferentDependency>(); // Try方法仅当尚未注册实现时，注册该服务。 |

在以下示例中，第一行向 IMyDep1 注册 MyDep。 第二行向 IMyDep2 注册 MyDep。 第三行没有任何作用，因为 IMyDep1 已有一个 MyDep 的已注册的实现：

|  |
| --- |
| public interface IMyDep1 { }  public interface IMyDep2 { }  public class MyDep : IMyDep1, IMyDep2 { }  services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep1, MyDep>());  services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep2, MyDep>());  // Two registrations of MyDep for IMyDep1 is avoided by the following line:  services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep1, MyDep>()); |

**4.服务生存期对比**

首先按照DIP原则设计类，以实现接口注入，源码如下：

定义接口:

|  |
| --- |
| public interface IOperation  {  Guid OperationId { get; }  }  // 暂时性(Transient)  public interface IOperationTransient : IOperation  {  }  // 范围内(Scoped)  public interface IOperationScoped : IOperation  {  }  // 单例(Singleton)  public interface IOperationSingleton : IOperation  {  }  //单例实例(SingletonInstance)  public interface IOperationSingletonInstance : IOperation  {  } |

定义底层实现类

|  |
| --- |
| // 定义底层实现类(生产者)  public class Operation :  IOperationTransient,  IOperationScoped,  IOperationSingleton,  IOperationSingletonInstance  {  public Operation() : this(Guid.NewGuid())  {  }  public Operation(Guid id)  {  OperationId = id;  }  public Guid OperationId { get; private set; }  } |

定义高层实现类

|  |
| --- |
| // 定义高层业务类(消费者)(类)  public class OperationService  {  // 类实例化时，容器服务会进行依赖注入  public OperationService(  IOperationTransient transientOperation,  IOperationScoped scopedOperation,  IOperationSingleton singletonOperation,  IOperationSingletonInstance instanceOperation)  {  TransientOperation = transientOperation;  ScopedOperation = scopedOperation;  SingletonOperation = singletonOperation;  SingletonInstanceOperation = instanceOperation;  }  public IOperationTransient TransientOperation { get; }  public IOperationScoped ScopedOperation { get; }  public IOperationSingleton SingletonOperation { get; }  public IOperationSingletonInstance SingletonInstanceOperation { get; }  } |

控制器代码(HomeController.cs)

|  |
| --- |
| // 定义高层业务类(消费者)(控制器)  public class HomeController : Controller  {  private OperationService \_operationService;  private IOperationTransient \_transientOperation;  private IOperationScoped \_scopedOperation;  private IOperationSingleton \_singletonOperation;  private IOperationSingletonInstance \_singletonInstanceOperation;  // 控制器实例化时，容器服务会进行依赖注入  public HomeController(  OperationService operationService,  IOperationTransient transientOperation,  IOperationScoped scopedOperation,  IOperationSingleton singletonOperation,  IOperationSingletonInstance singletonInstanceOperation)  {  \_operationService = operationService;  \_transientOperation = transientOperation;  \_scopedOperation = scopedOperation;  \_singletonOperation = singletonOperation;  \_singletonInstanceOperation = singletonInstanceOperation;  }  public IActionResult Index()  {  ViewBag.OperationService = \_operationService;  ViewBag.TransientOperation = \_transientOperation.OperationId;  ViewBag.ScopedOperation = \_scopedOperation.OperationId;  ViewBag.SingletonOperation = \_singletonOperation.OperationId;  ViewBag.SingletonInstanceOperation = \_singletonInstanceOperation.OperationId;  return View();  }  } |

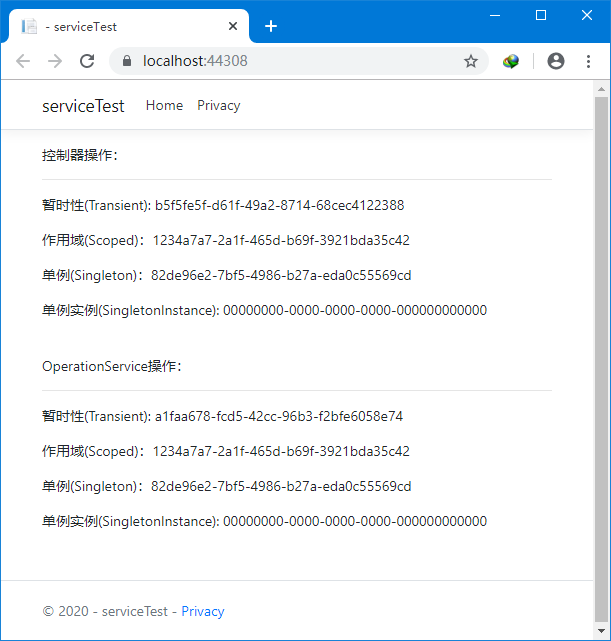
视图代码(Index.cshtml)

|  |
| --- |
| <p>控制器操作：</p>  <hr />  <div class="text-left">  <p>暂时性(Transient): @ViewBag.TransientOperation</p>  <p>作用域(Scoped)：@ViewBag.ScopedOperation</p>  <p>单例(Singleton)：@ViewBag.SingletonOperation</p>  <p>单例实例(SingletonInstance): @ViewBag.SingletonInstanceOperation</p>  </div>  <br />  <p>OperationService操作：</p>  <hr />  <div class="text-left">  <p>暂时性(Transient): @ViewBag.OperationService.TransientOperation.OperationId</p>  <p>作用域(Scoped)：@ViewBag.OperationService.ScopedOperation.OperationId</p>  <p>单例(Singleton)：@ViewBag.OperationService.SingletonOperation.OperationId</p>  <p>单例实例(SingletonInstance): @ViewBag.OperationService.SingletonInstanceOperation.OperationId</p>  </div> |

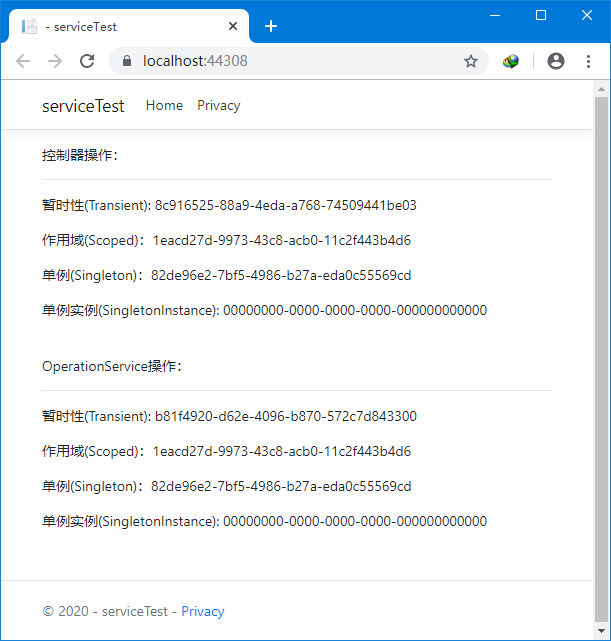
现在将服务添加到容器中，以便能够进行依赖注入：

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddTransient<IOperationTransient, Operation>();  services.AddScoped<IOperationScoped, Operation>();  services.AddSingleton<IOperationSingleton, Operation>();  // 单例实例，和单例一样，只是在注册的时候可以提供一个实例化对象  services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));  } |

F5运行



刷新页面，重新发起页面加载请求：



观察OperationId值在不同请求之间变化，你会发现：

暂时性(Transient)：

对象始终不同。第一个和第二个请求的OperationId值在控制器操作和OperationService操作中都是不同的，说明都提供了一个新实例。

作用域(Scoped)：

对象在一个客户端(即单个浏览器)请求中是相同的，但在多个客户端(即多个浏览器)请求中是不同的。

单例(Singleton):

对象在每个请求都是相同的(不管Startup.ConfigureServices中是否提供Operation实例，还是在不同的客户端请求中(即不同的浏览器中))。

**5.从Main调用服务**

使用[IServiceScopeFactory.CreateScope](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.dependencyinjection.iservicescopefactory.createscope)可以创建[IServiceScope](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.dependencyinjection.iservicescope)用以解析应用范围内的已设置范围(Scoped)的服务。此方法可以用于在启动时访问有作用域(Scoped)的服务以便运行初始化任务。 以下示例演示如何在Program.Main中获取MyScopedService的上下文：

|  |
| --- |
| using System;  using System.Threading.Tasks;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.AspNetCore.Hosting;  using Microsoft.Extensions.Hosting;  public class Program  {  public static async Task Main(string[] args)  {  var host = CreateHostBuilder(args).Build();  using (var serviceScope = host.Services.CreateScope())  {  var services = serviceScope.ServiceProvider;  try  {  var serviceContext = services.GetRequiredService<MyScopedService>();  // Use the context here  }  catch (Exception ex)  {  var logger = services.GetRequiredService<ILogger<Program>>();  logger.LogError(ex, "An error occurred.");  }  }  await host.RunAsync();  }  public static IHostBuilder CreateHostBuilder(string[] args) =>  Host.CreateDefaultBuilder(args)  .ConfigureWebHostDefaults(webBuilder =>  {  webBuilder.UseStartup<Startup>();  });  } |

**三、设计能够进行依赖关系注入的服务**

最佳做法是：

1.应该遵循依赖关系反转原则(DIP)设计服务以使用依赖关系注入来获取其依赖关系。

2.避免静态类和静态成员，将应用设计为改用单一实例服务，可避免创建全局状态。

3.避免在服务中直接实例化依赖类。

4.不在应用类中包含过多内容，确保设计规范，并易于测试。

如果一个类有过多的依赖关系注入，这通常表明该类拥有过多的责任并且违反了[单一责任原则 (SRP)](https://docs.microsoft.com/zh-cn/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#single-responsibility)。应该尝试通过将某些职责移动到一个新类来重构类。 请记住，Razor Pages 页、模型类和MVC控制器类应关注用户界面问题。业务规则和数据访问实现细节应保留在适用于这些[分离的关注点](https://docs.microsoft.com/zh-cn/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#separation-of-concerns)的类中。

**四、服务销毁**

容器只为其创建的且实现了 [IDisposable](https://docs.microsoft.com/zh-cn/dotnet/api/system.idisposable) 接口的类型调用 [Dispose](https://docs.microsoft.com/zh-cn/dotnet/api/system.idisposable.dispose)。

如果通过用户代码将实例添加到容器中，则不会自动销毁该实例。如下例所示：

|  |
| --- |
| // 实现了IDisposable接口的服务：  public class Service1 : IDisposable { }  public class Service2 : IDisposable { }  public class Service3 : IDisposable { }  public interface ISomeService { }  public class SomeServiceImplementation : ISomeService, IDisposable { }  public void ConfigureServices(IServiceCollection services)  {  // 容器创建下面的实例，并自动调用Dispose销毁它们  services.AddScoped<Service1>();  services.AddSingleton<Service2>();  services.AddSingleton<ISomeService>(sp => new SomeServiceImplementation());  // 如果用户通过代码将实例添加到容器中，则不会自动调用Dispose销毁它们。用户必须自己调用Dispose销毁  services.AddSingleton<Service3>(new Service3());  services.AddSingleton(new Service3());  } |

**五、替换默认的服务容器**

内置的服务容器旨在满足框架和大多数消费者应用的需求。我们建议使用内置容器，除非你需要的特定功能但内置容器不支持，例如：

1.属性注入(Property injection)

2.基于名称的注入(Injection based on name)

3.子容器(Child containers)

4.自定义生存期管理(Custom lifetime management)

5.对迟缓初始化的Func<T>支持(Func<T> support for lazy initialization)

6.基于约定的注册(Convention-based registration)

你可以选择以下第三方容器替代内置容器：

[Autofac](https://autofac.readthedocs.io/en/latest/integration/aspnetcore.html)

[DryIoc](https://www.nuget.org/packages/DryIoc.Microsoft.DependencyInjection)

[Grace](https://www.nuget.org/packages/Grace.DependencyInjection.Extensions)

[LightInject](https://github.com/seesharper/LightInject.Microsoft.DependencyInjection)

[Lamar](https://jasperfx.github.io/lamar/)

[Stashbox](https://github.com/z4kn4fein/stashbox-extensions-dependencyinjection)

[Unity](https://www.nuget.org/packages/Unity.Microsoft.DependencyInjection)