# .net Core 使用Ioc注入服务

## Ioc原理介绍

### 1、Ioc简介

在学习控制反转（Inversion of Control，以下由Ioc简称）的时候总是伴随着依赖倒置（Dependence Inversion Principle，以下由DIP简称）、依赖注入（Dependency Injection，以下由DI简称）、Ioc容器这些专业名词让人望而生畏。然而抛开这些复杂的理论其实Ioc只是一种降低项目中模块之间的依赖（也就是耦合度）的设计模式。

### 2、依赖（Dependence）

既然Ioc是降低项目中的依赖，那什么是依赖？很简单，当在一个类A中创建了类B的实例时，就称类A依赖于类B。

public class A

{

private B b = new B();

public void TestA()

{

b.TestB();

}

}

public class B

{

public void TestB()

{

Console.Write("Test");

}

}

如果仅仅是几个类产生了依赖并没有什么问题，但一个项目中往往有几十个类（甚至上百个），此时依赖若未得到很好地控制，就会发生当一个类需要修改时（比如方法增加参数，修改返回值类型等等）与之相关联的类都得跟着调整，这就大幅度降低了开发的效率和维护的难度。

如何处理项目中的依赖？在设计模式《Design Patterns: Elements of Reusable Object-Oriented Software》一书中几位大神提出了20多种设计模式（这里不多赘述），而需要关注的是设计模式中六大原则：

* **开闭原则（Open Close Principle）**：对扩展开放，对修改关闭；
* **里氏代换原则（Liskov Substitution Principle）：**任何基类可以出现的地方，子类一定可以出现。
* **依赖倒置原则（Dependence Inversion Principle）：**针对接口编程，依赖于抽象而不依赖于具体。
* **接口隔离原则（Interface Segregation Principle）：**使用多个隔离的接口，比使用单个接口要好。
* **迪米特法则，又称最少知道原则（Demeter Principle）：**一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。
* **合成复用原则（Composite Reuse Principle）：**尽量使用合成/聚合的方式，而不是使用继承。

### 3、依赖倒置（DIP）

设计模式中六大原则中**依赖倒置明确的告诉我们处理**依赖的方法，其原始定义为：High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions。

翻译过来就是：

* 高层模块不应该依赖于底层模块，两者都应依赖抽象
* 抽象不应该依赖细节
* 细节应该依赖于抽象

在日常开发中则表现为：

* 模块间的依赖是通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类；
* 接口或抽象类不依赖于实现类；
* 实现类依赖接口或抽象类。

那什么又是高低层模块，当A依赖于B，而B却相对独立如下图，此时A也被称为高层模块，B也被成为低层模块。

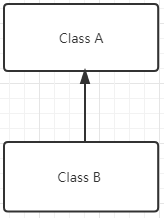
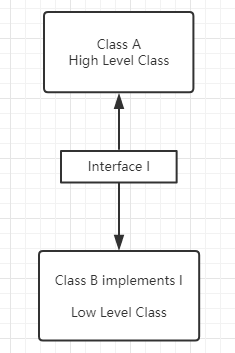


图 3-1

而之所以要依赖于抽象（接口和抽象类），是因为在项目中抽象（接口和抽象类）往往是规则（法名、参数、属性等信息）的定义，定好后就不会轻易发生改变（除非项目重构），如下图中当B发生变化时并不会让A中对Interface的调用发生改变（同理A发生变化也不会对B造成影响），这也就是面向接口编程（优秀的团队往往时间花得最多的是接口的定义，以及开发文档的编写，编程只要按照定好的规则来占较少的比例）。



### 5、控制反转（Ioc）

我们先写一段代码来实现DIP面向接口设计原则如下：

public class A

{

private I impl = new B();

public void testA(string param)

{

Console.WriteLine(impl.test(param));

}

}

public interface I

{

string test(string param);

}

public class B : I

{

public string test(string param)

{

return "class B: " + param;

}

}

当高层模块A中需要使用低层模块时，只需要实例化相应的低层模块即可。乍一看好像没什么问题，但是看看private I impl = new B()这句代码，虽然使用了面向接口的编程方式并没有解决A对B的依赖。而Ioc正好解决了这个问题。

IOC的全名Inverse of Control，翻译成中文就是“控制反转”。然而它真正意指的是控制权的转移，即原来控制权（private I impl = new B()）在A手中，现在转移给第三方来接管（反转这个词纠结我了很长时间一直不理解哪反转了，然而其实是转移），也就是A中只包含对抽象的定义，而对低层模块的构造将被剥离出去，此时A、B都只依赖于抽象而真正地独立开来。

public class A

{

private I impl;

public void testA(string param)

{

Console.WriteLine(impl.test(param));

}

}

public interface I

{

string test(string param);

}

public class B : I

{

public string test(string param)

{

return "class B: " + param;

}

}

### 6、依赖注入（DI）

Ioc该如何实现，其关键是如何在第三方中将A对B的依赖注入到A中。依赖注入（DI）就是一种Ioc的实现机制，常见的有两种：构造函数注入和属性注入。下面使用main函数模拟第三方Ioc模块。

第一种：构造函数注入

public class A

{

private I \_impl;

public A(I impl)

{

this.\_impl = impl;

}

public void testA(string param)

{

Console.WriteLine(\_impl.test(param));

}

}

public interface I

{

string test(string param);

}

public class B : I

{

public string test(string param)

{

return "class B: " + param;

}

}

对应的Main函数：

class Program

{

static void Main(string[] args)

{

B b = new B(); // 创建依赖

A a = new A(b); // 注入依赖

a.testA("构造函数注入");

Console.Read();

}

}

第二种：属性注入

public class A

{

private I \_impl;

public I Ipml

{

get { return this.\_impl; }

set { this.\_impl = value; }

}

public void testA(string param)

{

Console.WriteLine(\_impl.test(param));

}

}

public interface I

{

string test(string param);

}

public class B : I

{

public string test(string param)

{

return "class B: " + param;

}

}

对应的main函数：

class Program

{

static void Main(string[] args)

{

B b = new B(); // 创建依赖

A a = new A();

a.Ipml = b; // 注入依赖

a.testA("属性注入");

Console.Read();

}

}

### 7、IoC容器

在项目中如果全用手动的方式，自己来创建和注入依赖的话，显然效率很低，而且往往还会出现不可控的场面。正因如此，IoC容器诞生了。IoC容器实际上是一个DI框架，它能简化我们的工作量。它包含以下几个功能：

* 动态创建、注入依赖对象。
* 管理对象生命周期。
* 映射依赖关系。

## 二、.net core原生Ioc使用

## 三、.net core使用AutoFac第三方Ioc框架