[**深入理解DIP、IoC、DI以及IoC容器**](http://www.cnblogs.com/liuhaorain/p/3747470.html)

**摘要**

面向对象设计（OOD）有助于我们开发出高性能、易扩展以及易复用的程序。其中，OOD有一个重要的思想那就是依赖倒置原则（DIP），并由此引申出IoC、DI以及Ioc容器等概念。通过本文我们将一起学习这些概念，并理清他们之间微妙的关系。

**目录**

* [前言](http://www.cnblogs.com/liuhaorain/p/3747470.html#title_1)
* [依赖倒置原则（DIP)](http://www.cnblogs.com/liuhaorain/p/3747470.html#title_2)
* [控制反转（IoC）](http://www.cnblogs.com/liuhaorain/p/3747470.html#title_3)
* [依赖注入（DI）](http://www.cnblogs.com/liuhaorain/p/3747470.html#title_4)
* [IoC容器](http://www.cnblogs.com/liuhaorain/p/3747470.html#title_5)
* [总结](http://www.cnblogs.com/liuhaorain/p/3747470.html#title_6)

**前言**

对于大部分小菜来说，当听到大牛们高谈DIP、IoC、DI以及IoC容器等名词时，有没有瞬间石化的感觉？其实，这些“高大上”的名词，理解起来也并不是那么的难，关键在于入门。只要我们入门了，然后循序渐进，假以时日，自然水到渠成。

好吧，我们先初略了解一下这些概念。

**依赖倒置原则（DIP）：**一种软件架构设计的原则（抽象概念）。

**控制反转（IoC）：**一种反转流、依赖和接口的方式（DIP的具体实现方式）。

**依赖注入（DI）：**IoC的一种实现方式，用来反转依赖（IoC的具体实现方式）。

**IoC容器：**依赖注入的**框架**，用来映射依赖，管理对象创建和生存周期（DI框架）。

哦！也许你正为这些陌生的概念而伤透脑筋。不过没关系，接下来我将为你一一道破这其中的玄机。

**依赖倒置原则（DIP）**

在讲概念之前，我们先看生活中的一个例子。



                                                            图1   ATM与银行卡

相信大部分取过钱的朋友都深有感触，只要有一张卡，随便到哪一家银行的ATM都能取钱。在这个场景中，ATM相当于高层模块，而银行卡相当于低层模块。ATM定义了一个插口（接口），供所有的银行卡插入使用。也就是说，ATM不依赖于具体的哪种银行卡。它只需定义好银行卡的规格参数（接口），所有实现了这种规格参数的银行卡都能在ATM上使用。现实生活如此，软件开发更是如此。**依赖倒置原则，它转换了依赖，高层模块不依赖于低层模块的实现，而低层模块依赖于高层模块定义的接口**。通俗的讲，就是高层模块定义接口，低层模块负责实现。

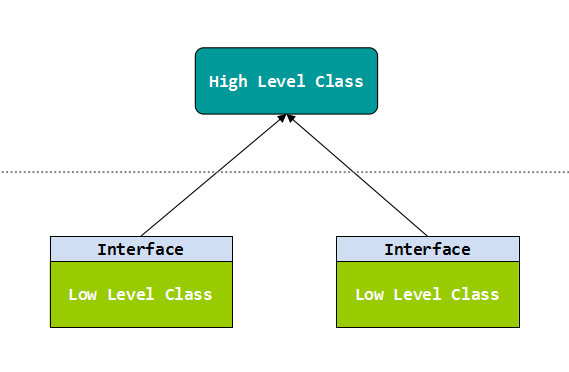
Bob Martins对DIP的定义：

*高层模块不应依赖于低层模块，两者应该依赖于抽象。*

*抽象不不应该依赖于实现，实现应该依赖于抽象。*

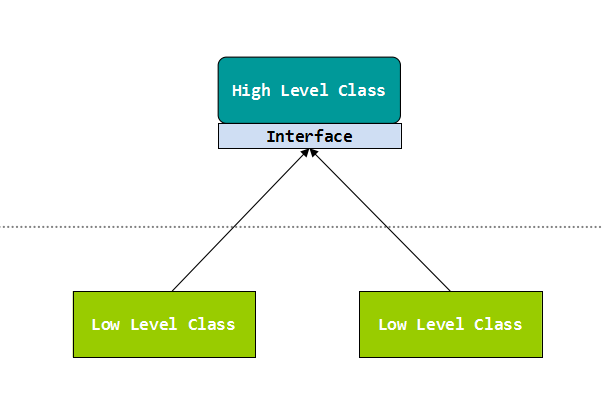
如果生活中的实例不足以说明依赖倒置原则的重要性，那下面我们将通过软件开发的场景来理解为什么要使用依赖倒置原则。

**场景一  依赖无倒置（低层模块定义接口，高层模块负责实现）**



从上图中，我们发现高层模块的类依赖于低层模块的接口。因此，低层模块需要考虑到所有的接口。如果有新的低层模块类出现时，高层模块需要修改代码，来实现新的低层模块的接口。这样，就破坏了开放封闭原则。

**场景二 依赖倒置（高层模块定义接口，低层模块负责实现）**



在这个图中，我们发现高层模块定义了接口，将不再直接依赖于低层模块，低层模块负责实现高层模块定义的接口。这样，当有新的低层模块实现时，不需要修改高层模块的代码。

由此，我们可以总结出使用DIP的优点：

**系统更柔韧：**可以修改一部分代码而不影响其他模块。

**系统更健壮：**可以修改一部分代码而不会让系统崩溃。

**系统更高效：**组件松耦合，且可复用，提高开发效率。

**控制反转（IoC）**

DIP是一种 **软件设计原则**，它仅仅告诉你两个模块之间应该如何依赖，但是它并没有告诉如何做。IoC则是一种 **软件设计模式**，它告诉你应该如何做，来解除相互依赖模块的耦合。**控制反转（IoC），它为相互依赖的组件提供抽象，将依赖（低层模块）对象的获得交给第三方（系统）来控制**，**即依赖对象不在被依赖模块的类中直接通过new来获取**。在图1的例子我们可以看到，ATM它自身并没有插入具体的银行卡（工行卡、农行卡等等），而是将插卡工作交给人来控制，即我们来决定将插入什么样的银行卡来取钱。同样我们也通过软件开发过程中场景来加深理解。

**软件设计原则：**原则为我们提供指南，它告诉我们什么是对的，什么是错的。它不会告诉我们如何解决问题。它仅仅给出一些准则，以便我们可以设计好的软件，避免不良的设计。一些常见的原则，比如DRY、OCP、DIP等。

**软件设计模式：**模式是在软件开发过程中总结得出的一些可重用的解决方案，它能解决一些实际的问题。一些常见的模式，比如工厂模式、单例模式等等。

做过电商网站的朋友都会面临这样一个问题：**订单入库**。假设系统设计初期，用的是SQL Server数据库。通常我们会定义一个SqlServerDal类，用于数据库的读写。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class SqlServerDal  {       public void Add()      {          Console.WriteLine("在数据库中添加一条订单!");      }  } |

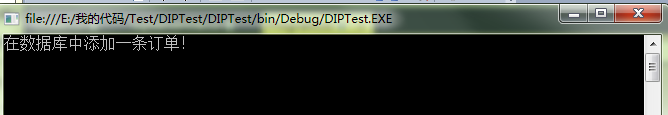
 然后我们定义一个Order类，负责订单的逻辑处理。由于订单要入库，需要依赖于数据库的操作。因此在Order类中，我们需要定义SqlServerDal类的变量并初始化。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Order  {          private readonly SqlServerDal dal = new SqlServerDal();//添加一个私有变量保存数据库操作的对象             public void Add()         {             dal.Add();         }  } |

最后，我们写一个控制台程序来检验成果。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;    namespace DIPTest  {      class Program      {          static void Main(string[] args)          {              Order order = new Order();              order.Add();                Console.Read();          }      }  } |

 输出结果：



  OK，结果看起来挺不错的！正当你沾沾自喜的时候，这时BOSS过来了。“小刘啊，刚客户那边打电话过来说数据库要改成Access”，“对你来说，应当小CASE啦！”BOSS又补充道。带着自豪而又纠结的情绪，我们思考着修改代码的思路。

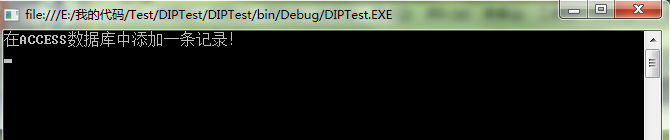
 由于换成了Access数据库，SqlServerDal类肯定用不了了。因此，我们需要新定义一个AccessDal类，负责Access数据库的操作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class AccessDal  {      public void Add()     {         Console.WriteLine("在ACCESS数据库中添加一条记录！");     }  } |

 然后，再看Order类中的代码。由于，Order类中直接引用了SqlServerDal类的对象。所以还需要修改引用，换成AccessDal对象。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Order  {          private readonly AccessDal dal = new AccessDal();//添加一个私有变量保存数据库操作的对象             public void Add()         {             dal.Add();         }  } |

输出结果：



费了九牛二虎之力，程序终于跑起来了！试想一下，如果下次客户要换成MySql数据库，那我们是不是还得重新修改代码？

显然，这不是一个良好的设计，组件之间高度耦合，可扩展性较差，它违背了DIP原则。高层模块Order类不应该依赖于低层模块SqlServerDal，AccessDal，两者应该依赖于抽象。那么我们是否可以通过IoC来优化代码呢？答案是肯定的。IoC有2种常见的实现方式：依赖注入和服务定位。其中，依赖注入使用最为广泛。下面我们将深入理解依赖注入（DI），并学会使用。

**依赖注入（DI）**

控制反转（IoC）一种重要的方式，**就是将依赖对象的创建和绑定转移到被依赖对象类的外部来实现**。在上述的实例中，Order类所依赖的对象SqlServerDal的创建和绑定是在Order类内部进行的。事实证明，这种方法并不可取。既然，不能在Order类内部直接绑定依赖关系，那么如何将SqlServerDal对象的引用传递给Order类使用呢？



依赖注入（DI），**它提供一种机制，将需要依赖（低层模块）对象的引用传递给被依赖（高层模块）对象。**通过DI，我们可以在Order类的外部将SqlServerDal对象的引用传递给Order类对象。那么具体是如何实现呢？

**方法一 构造函数注入**

构造函数函数注入，毫无疑问通过构造函数传递依赖。因此，构造函数的参数必然用来接收一个依赖对象。那么参数的类型是什么呢？具体依赖对象的类型？还是一个抽象类型？根据DIP原则，我们知道高层模块不应该依赖于低层模块，两者应该依赖于抽象。那么构造函数的参数应该是一个抽象类型。我们再回到上面那个问题，**如何将SqlServerDal对象的引用传递给Order类使用呢**？

首选，我们需要定义SqlServerDal的抽象类型IDataAccess，并在IDataAccess接口中声明一个Add方法。

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IDataAccess  {          void Add();  } |

 然后在SqlServerDal类中，实现IDataAccess接口。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class SqlServerDal:IDataAccess  {         public void Add()         {             Console.WriteLine("在数据库中添加一条订单！");         }  } |

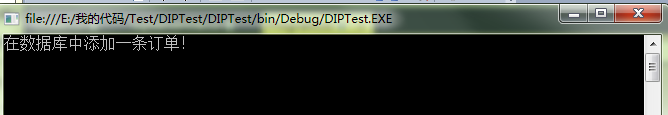
 接下来，我们还需要修改Order类。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Order    {          private IDataAccess \_ida;//定义一个私有变量保存抽象            //构造函数注入          public Order(IDataAccess ida)          {              \_ida = ida;//传递依赖        }            public void Add()          {              \_ida.Add();          }  } |

 OK，我们再来编写一个控制台程序。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;    namespace DIPTest  {      class Program      {          static void Main(string[] args)          {              SqlServerDal dal = new SqlServerDal();//在外部创建依赖对象              Order order = new Order(dal);//通过构造函数注入依赖                order.Add();                Console.Read();          }      }  } |

 输出结果：



从上面我们可以看出，我们将依赖对象SqlServerDal对象的创建和绑定转移到Order类外部来实现，这样就解除了SqlServerDal和Order类的耦合关系。当我们数据库换成Access数据库时，只需定义一个AccessDal类，然后外部重新绑定依赖，不需要修改Order类内部代码，则可实现Access数据库的操作。

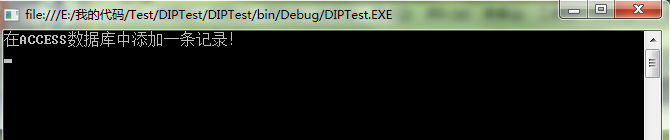
定义AccessDal类：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class AccessDal:IDataAccess  {          public void Add()          {              Console.WriteLine("在ACCESS数据库中添加一条记录！");          }  } |

然后在控制台程序中重新绑定依赖关系：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;    namespace DIPTest  {      class Program      {          static void Main(string[] args)          {               AccessDal dal = new AccessDal();//在外部创建依赖对象                 Order order = new Order(dal);//通过构造函数注入依赖                   order.Add();                Console.Read();          }      }  } |

输出结果：



显然，我们不需要修改Order类的代码，就完成了Access数据库的移植，这无疑体现了IoC的精妙。

**方法二 属性注入**

顾名思义，属性注入是通过属性来传递依赖。因此，我们首先需要在依赖类Order中定义一个属性：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Order  {        private IDataAccess \_ida;//定义一个私有变量保存抽象            //属性，接受依赖          public IDataAccess Ida         {             set { \_ida = value; }             get { return \_ida; }         }           public void Add()         {             \_ida.Add();         }  } |

 然后在控制台程序中，给属性赋值，从而传递依赖：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;    namespace DIPTest  {      class Program      {          static void Main(string[] args)          {              AccessDal dal = new AccessDal();//在外部创建依赖对象              Order order = new Order();              order.Ida = dal;//给属性赋值                order.Add();                Console.Read();          }      }  } |

我们可以得到上述同样的结果。

**方法三 接口注入**

相比构造函数注入和属性注入，接口注入显得有些复杂，使用也不常见。具体思路是先定义一个接口，包含一个设置依赖的方法。然后依赖类，继承并实现这个接口。

首先定义一个接口：

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IDependent  {             void SetDependence(IDataAccess ida);//设置依赖项  } |

依赖类实现这个接口：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Order : IDependent   {       private IDataAccess \_ida;//定义一个私有变量保存抽象         //实现接口       public void SetDependence(IDataAccess ida)       {           \_ida = ida;       }         public void Add()       {           \_ida.Add();       }     } |

  控制台程序通过SetDependence方法传递依赖：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;    namespace DIPTest  {      class Program      {          static void Main(string[] args)          {              AccessDal dal = new AccessDal();//在外部创建依赖对象            Order order = new Order();                order.SetDependence(dal);//传递依赖                order.Add();                Console.Read();          }      }  } |

我们同样能得到上述的输出结果。

**IoC容器**

前面所有的例子中，我们都是通过**手动**的方式来创建依赖对象，并将引用传递给被依赖模块。比如：

|  |  |
| --- | --- |
| 1  2 | SqlServerDal dal = new SqlServerDal();//在外部创建依赖对象  Order order = new Order(dal);//通过构造函数注入依赖 |

 对于大型项目来说，相互依赖的组件比较多。如果还用手动的方式，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。

目前，比较流行的Ioc容器有以下几种：

1.**Ninject**:  <http://www.ninject.org/>

2. **Castle Windsor**:  <http://www.castleproject.org/container/index.html>

3.**Autofac**:  <http://code.google.com/p/autofac/>

4. **StructureMap**： <http://docs.structuremap.net/>

5. **Unity**：  <http://unity.codeplex.com/>

**注：**根据园友 [徐少侠](http://www.cnblogs.com/Chinese-xu/) 的提醒，MEF不应该是IoC容器。我又查阅了一些资料，觉得MEF作为IoC容器是有点勉强，它的主要作用还是用于应用程序扩展，避免生成脆弱的硬依赖项。

~~6.~~**~~MEF~~**~~:~~[~~http://msdn.microsoft.com/zh-cn/library/dd460648.aspx~~](http://msdn.microsoft.com/zh-cn/library/dd460648.aspx)

另外，园友[aixuexi](http://www.cnblogs.com/LiuJourney/)提出Spring.NET也是比较流行的IoC容器。

7.**Spring.NET**： <http://www.springframework.net/>

园友 *wdwwtzy* 也推荐了一个不错的IoC容器：

8. **LightInject**:  <http://www.lightinject.net/> （推荐使用Chrome浏览器访问）

 以Ninject为例，我们同样来实现[**[方法一 构造函数注入]**](http://www.cnblogs.com/liuhaorain/p/3747470.html#title_7) 的功能。

首先在项目添加Ninject程序集，同时使用using指令引入。

|  |  |
| --- | --- |
| 1 | using Ninject; |

然后，Ioc容器注册绑定依赖：

|  |  |
| --- | --- |
| 1  2  3 | StandardKernel kernel = new StandardKernel();    kernel.Bind<IDataAccess>().To<SqlServerDal>();//注册依赖 |

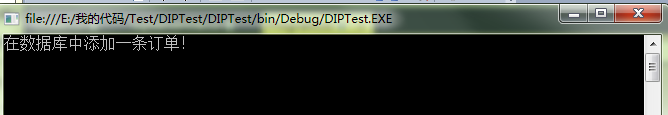
 接下来，我们获取需要的Order对象（注入了依赖对象）：

|  |  |
| --- | --- |
| 1 | Order order = kernel.Get<Order>(); |

 下面，我们写一个完整的控制台程序

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using Ninject;    namespace DIPTest  {      class Program      {          static void Main(string[] args)          {             StandardKernel kernel = new StandardKernel();//创建Ioc容器             kernel.Bind<IDataAccess>().To<SqlServerDal>();//注册依赖                 Order order = kernel.Get<Order>();//获取目标对象                 order.Add();             Console.Read();          }      }  } |

 输出结果：



使用IoC容器，我们同样实现了该功能。

**总结**

在本文中，我试图以最通俗的方式讲解，希望能帮助大家理解这些概念。下面我们一起来总结一下：DIP是软件设计的一种思想，IoC则是基于DIP衍生出的一种软件设计模式。DI是IoC的具体实现方式之一，使用最为广泛。IoC容器是DI构造函注入的框架，它管理着依赖项的生命周期以及映射关系。

作者：[可米小子](http://liuhaorain.cnblogs.com/)