

# 解除具体依赖的技术

张逸

www.agiledon.com

一个外部具体对象的引入，必然会给一个模块带来与外部模块之间的依赖。而具体对象的创建始终是我们无法规避的。即使我们可以利用设计模式的工厂方法模式或抽象工厂封装具体对象创建的逻辑，但却又再次引入了具体工厂对象的创建依赖。虽然在设计上有所改进，但没有彻底解除具体依赖，仍让我心有戚戚焉。

以一个电子商务网站的设计为例。在该项目中要求对客户的订单进行管理，例如插入订单。考虑到访问量的关系，系统为订单管理提供了同步和异步的方式。显然，在实际应用中，我们需要根据具体的应用环境，决定使用这两种方式的其中一种。由于变化非常频繁，因而我们采取了“封装变化”的设计思想。譬如，考虑应用 Strategy 模式，因为插入订单的行为，实则就是一种插入订单的策略。我们可以为此策略建立抽象对象，如 `IOrderStrategy` 接口。

```
public interface IOrderStrategy {  
    void Insert(OrderInfo order);  
}
```

然后分别定义两个类 `OrderSynchronous` 和 `OrderAsynchronous` 实现 `IOrderStrategy` 接口。类结构如图 1 所示。

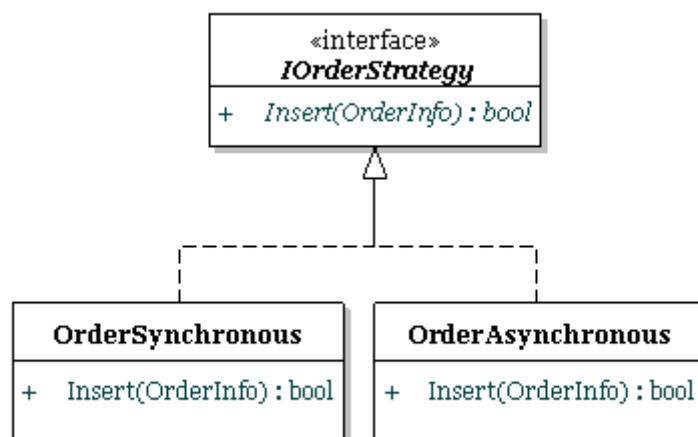


图 1 订单策略的设计

当领域对象 `Order` 类需要插入订单时，将根据 `IOrderStrategy` 接口的运行期类型，执

行相关的订单插入策略，如下代码所示。

```
public class Order {
    private IOrderStrategy m_orderStrategy;
    public Order(IOrderStrategy orderStrategy) {
        m_orderStrategy = orderStrategy;
    }
    public void Insert(OrderInfo order) {
        m_orderStrategy.Insert(order);
    }
}
```

由于用户随时都可能会改变插入订单的策略，因此对于业务层的订单领域对象而言，绝不能与具体的订单策略对象产生耦合关系。也就是说，在领域对象 Order 类中，不能 new 一个具体的订单策略对象，如下面的代码：

```
IOrderStrategy orderStrategy = new OrderSynchronous();
```

虽然在前面的实现中，我们通过领域对象的构造函数传递了 IOrderStrategy 接口对象。但这样的实现仅仅是将具体订单策略对象的创建推迟到了领域对象的调用者那里而已，调用者无法避免具体订单策略对象的创建。显然，这是一种“治标不治本”的做法。我们当然也期望能够有一种理想的状态，就是具体对象的创建永远都不要在代码中出现。事实上，模块与模块间之所以产生依赖关系，正是因为有具体对象的存在。一旦在一个模块中创建了另一个模块中的具体对象，依赖就产生了。现在，我们的目的就是要将这些依赖消除。

## 1、配置文件与反射技术

使用硬编码方式创建一个对象，必然会带来对象之间的具体依赖。一种最简单的方式是将反射技术与配置文件相结合，在具体对象拥有共同抽象的前提下，通过配置文件获得具体对象的类型信息，然后利用反射创建相应的对象。例如，在领域对象 Order 类中，可以如此实现：

```
public class Order {
    private static readonly IOrderStrategy orderInsertStrategy =
        LoadInsertStrategy();
    private static IOrderStrategy LoadInsertStrategy() {
        //通过配置文件找到具体的订单策略对象
        string path = ConfigurationManager.
```

```

        AppSettings["OrderStrategyAssembly"];
        string className = ConfigurationManager.
            AppSettings["OrderStrategyClass"];

        //通过反射创建对象实例

        return (IOrderStrategy)Assembly.Load(path).
            CreateInstance(className);
    }
}

```

在配置文件 web.config 中，配置如下的节：

```

<add key="OrderStrategyAssembly" value="AgileDon.BLL"/>
<add key="OrderStrategyClass" value="BLL.OrderSynchronous"/>

```

通过引入泛型，我们可以对前面的逻辑进行有效的封装，例如定义如下的工厂辅助类。

```

public static class FactoryHelper<T> where T : class {
    private static T instance = null;
    public static T Create(string typeNameKey,
        string nameSpace,
        string assemblyPath) {
        if (instance == null) {
            string typeName =
                ConfigurationManager.AppSettings[typeNameKey];
            string className = nameSpace + "." + typeName;
            instance = (T)Assembly.Load(assemblyPath).
                CreateInstance(className);
        }
        return instance;
    }
}

```

注意，Create()辅助方法中的 typeNameKey，是指向具体对象类型的键值。通常建议将其键值赋值为具体对象类型的抽象接口类型名，而对应的值则是目标创建对象的类型名。

例如：

```
<add key="IOrderStrategy" value="OrderSynchronous" />
```

然后，我们可以为属于相同命名空间的类统一定义工厂类，并在其中调用工厂辅助类 `FactoryHelper` 的 `Create()` 辅助方法。例如，为业务逻辑层的对象定义工厂类 `BLLFactory`。

```
public static class BLLFactory<T> where T : class {
    public static T Create(string typeNameKey) {
        string nameSpace =
            ConfigurationManager.AppSettings["BLLAssembly"];
        string assemblyPath =
            ConfigurationManager.AppSettings["BLLPath"];
        return BaseFactory<T>.CreateT(
            typeNameKey, nameSpace, assemblyPath);
    }
}
```

针对订单策略对象，对应的配置文件为：

```
<add key="BLLAssembly" value="AgileDon.BLL" />
<add key="BLLPath" value="AgileDon.BLL" />
<add key="IOrderStrategy" value="OrderSynchronous" />
```

现在，我们就可以调用 `BLLFactory` 类的 `Create()` 方法，传入类型名以获得具体的对象。例如：

```
IOrderStrategy orderInsertStrategy = BLLFactory<IOrderStrategy>.
    Create("IOrderStrategy");
```

如果需要将订单插入策略从同步修改为异步方式，只需将配置文件中 `IOrderStrategy` 键对应的值修改为 `"OrderAsynchronous"` 即可。

## 2、表驱动法

借鉴表驱动法【注：参见 Steve McConnell 著作《代码大全》第 18 章】的思想，我们可以利用一个 `Dictionary` 集合来维护目标对象与键值之间的映射关系。当我们需要获得对象时，可以利用键值对表进行查询，这样就可以有效地消除 `if` 语句。例如，可以在 `Strategy` 模式中使用表驱动法，将其作为模式的上下文对象，而不必执行对策略对象类型

的逻辑判断。利用表驱动法，我们也可以解除对象之间的具体依赖。

仍然以订单的管理为例。我为订单的管理专门定义了一个 `OrderManager` 类，它负责初始化并维持对象表。

```
public static class OrderManager {
    private static IDictionary<string, IOrderStrategy>
        m_strategyTable;
    static OrderManager() {
        Init();
    }
    private static void Init() {
        m_strategyTable = new Dictionary<string, IOrderStrategy>();
        m_strategyTable.Add("sync", new OrderSynchronous());
        m_strategyTable.Add("async", new OrderAsynchronous());
    }
    public static IOrderStrategy
        GetOrderStrategy(string strategyKey) {
        IOrderStrategy strategy;
        if (m_strategyTable.TryGetValue(strategyKey, out strategy))
        {
            return strategy;
        } else {
            throw new Exception("无法找到正确的订单策略对象");
        }
    }
}
```

在调用 `OrderManager` 的 `GetOrderStrategy()` 方法时，为提供更好的灵活性，寻找策略对象的键值应该放在配置文件中，以避免修改源代码。

```
string strategyKey =
    ConfigurationManager.AppSettings["StrategyKey"];
IOrderStrategy strategy =
    OrderManager.GetOrderStrategy(strategyKey);
```

我们甚至可以提供一个注册方法 `RegisterStrategy()`，用以应对未来可能的扩展。

```
public static class OrderManager {  
    //其余实现略  
  
    public static void RegisterStrategy(  
        string strategyKey,  
        IOrderStrategy strategy) {  
        if (String.IsNullOrEmpty(strategyKey)) {  
            throw new ArgumentNullException(strategyKey);  
        }  
        if (strategy == null) {  
            throw new ArgumentNullException("策略对象不能为 null");  
        }  
        if (m_strategyTable.ContainsKey(strategyKey)) {  
            throw new ArgumentException("已经存在键值" + strategyKey);  
        }  
        m_strategyTable.Add(strategyKey, strategy);  
    }  
}
```

### 3、依赖注入

依赖注入(Dependency Injection)是一个漂亮的隐喻。依赖关系就像是被注入的液体，我们可以在任何时候将依赖关系注入到模块中，而不只限于在编译时绑定。既然这种依赖关系是通过注入的方式完成，就意味着我们可以随时更新，因为注入的液体与模块并无直接关联。实现依赖注入的前提是面向接口编程，而辅助的技术则是利用反射技术。

依赖注入是目前大多数轻量级 IoC (控制反转, Inversion of Control) 容器用于解除外部服务与容器服务之间依赖关系的一把利刃。首先，容器服务包含了外部服务接口的定义。然后，依赖注入通过使用一个专门的装配器对象，提供外部服务的具体实现，并将其赋值给对应的容器服务对象。Martin Fowler 将依赖注入的形式分为三种：构造函数注入 (Constructor Injection)、设置方法注入 (Setter Injection) 和接口注入 (Interface Injection)。其中，接口注入通过定义接口约束的方式实现依赖注入，会给容器带来设计的限制。而构造函数注入与设置方式注入则表现了产生依赖的两个连接点：构造函数与属性。如果构造函数参数或属性对象的类型为抽象的接口类型，则产生具体依赖的源头在于具体对象的创建。将创建具体对象的职责转移到 IoC 容器，就可以在运行时为

构造函数参数或属性对象传递依赖。

目前,实现了依赖注入的轻量级容器已经应用在许多框架产品中,如 Java 平台下的 Spring、PicoContainer 等。在 .NET 平台下,常见的依赖注入框架包括 Autofac, Ninject, Spring.NET, StructureMap 和 Windsor 等。

以 Ninject 框架为例,我们可以定义这样的 Order 类:

```
public class Order {
    private IOrderStrategy m_orderStrategy;
    public Order(IOrderStrategy orderStrategy) {
        m_orderStrategy = orderStrategy;
    }
    public void Insert(OrderInfo order) {
        m_orderStrategy.Insert(order);
    }
}
```

然后,我们需要自定义一个 OrderModule 类,它派生自 Ninject.Core.StandardModule 类。这是 Ninject 实现依赖注入的一个特色,它抛弃了传统的 xml 映射文件,而是利用类型绑定的方式,并根据要创建的具体对象分组建立对应的 Module 类。注意,它不同于之前的解耦方法,因为它对业务逻辑代码没有造成任何侵略与污染。如上定义的 Order 类,保留了领域对象的本来面貌。使得领域层的开发人员可以专心致志着力于业务逻辑的实现。OrderModule 类的定义如下所示:

```
using Ninject.Core;

public class OrderModule:StandardModule {
    public override void Load() {
        Bind<IOrderStrategy>().To<OrderSynchronous>();
    }
}
```

客户端调用的代码可以通过 Ninject 提供的 IKernel 对象获得 Order 对象:

```
OrderModule module = new OrderModule();
IKernel kernal = new StandardKernel(module);
```

```
Order order = kernal.Get<Order>();  
order.Insert(new OrderInfo());
```

#### 4、惯例优于配置

使用配置文件固然可以解除与具体对象之间的依赖，然而，它带来的良好可扩展性，却是以牺牲系统的可维护性乃至至于可靠性为代价的。配置文件很难管理，尤其是在配置信息相对较多的情况下。不管是集中管理还是分散管理，都存在一些与生俱来的缺陷。如果采用集中管理，则配置文件过大，既影响性能，也不能很好地展现配置信息的分类与层次。在.NET中，虽然可以利用<section></section>对配置文件进行分节，但终究不够直观。采用分散管理，则不同大小的配置文件千头万绪，既会给维护者带来管理的障碍，也不利于部署与使用。使用配置文件尤其不便于调试。开发环境提供的编译期检查，对于配置文件只能是“望洋兴叹”。所谓“差之毫厘，谬以千里”，小小的一个配置项错误，可能会造成难以弥补的巨大损失。为了弥补这些缺陷，许多产品或框架都提供了专门的配置或管理工具，使用直观的UI界面对配置文件进行操作，但繁杂的配置项仍然有可能让使用者望而却步。

惯例优于配置 (Convention over Configuration) 来源于 Ruby On Rails 框架的设计理念，也被认为是 Rails 大获成功的关键因素之一。这里所谓的惯例，可以理解为框架对编程的一些约束，我们可以根据实现制订的默认规则，通过反射技术完成对象的创建，对象的协作，甚至是应用程序的组装。例如在 Rails 中对 MVC 模式的实现中，就事先确立了 Model、View 和 Controller 的目录结构与命名规范。在这种情况下，我们不需要对元数据进行任何配置。ASP.NET MVC 框架同样采纳了惯例优于配置的思想。采用惯例，虽然在一定程度上损失了系统的灵活性，带来的却是良好的可维护性。同时，它仍然可以解除系统与具体对象之间的强耦合关系。

惯例优于配置的技术并不是非常适合于本文中的订单策略示例。不过，在.NET框架中，有关 WebRequest 对象的创建，却可以改用惯例优于配置的思想来实现。图 2 是 WebRequest 对象的继承体系：

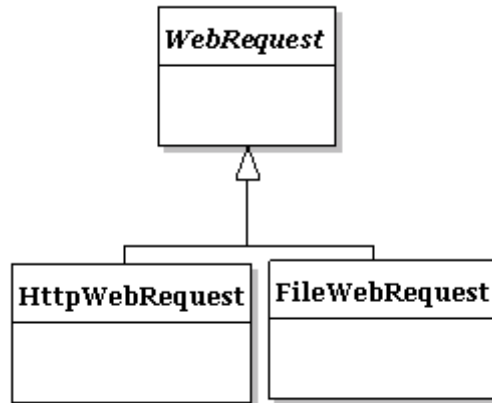


图 2 WebRequest 的类结构

在 .NET 框架中，创建一个 `WebRequest` 实例的方法是调用 `WebRequest` 的静态方法 `Create()`：

```
WebRequest myRequest =  
    WebRequest.Create("http://www.agiledon.com");
```

由于传入的 `Uri` 地址其前缀为 "http"，因此创建的 `myRequest` 对象应该为 `HttpWebRequest` 具体对象。如果需要根据不同的 `Request` 协议，扩展不同的 `WebRequest` 对象，就需要引入一些设计技巧，来解除与具体对象创建的依赖。 .NET 框架的实现能够达到这样的目的，但非常复杂，这里不提。我想要介绍的是如何利用惯例优于配置来实现 `WebRequest` 对象的扩展。利用“惯例优于配置”的思想有一个前提，就是我们要对 `WebRequest` 对象的命名规范进行惯例约束。例如，我们规定所有的 `WebRequest` 子类对象均由协议名加上“`WebRequest`”后缀构成。通过解析传入的 `Uri`，可以获得传输协议的名称，之后将它与“`WebRequest`”连接起来，获得 `WebRequest` 子类对象的类名，再利用反射技术创建该对象。在 `WebRequest` 类中定义如下的 `Create()` 静态方法：

```
public static WebRequest Create(Uri requestUri) {  
    if (requestUri == null) {  
        throw new ArgumentNullException("requestUri");  
    }  
    string prefix = requestUri.Scheme.ToLower();  
    if (prefix == null) {  
        throw new ArgumentNullException("requestUri");  
    }  
    if (prefix.Contains(".")) {
```

```
        prefix = prefix.Replace(".", "");
    }
    StringBuilder typeName = new StringBuilder();
    typeName.Append("System.Net.");
    typeName.Append(prefix.Substring(0, 1).ToUpper());
    typeName.Append(prefix.ToLower().Substring(1, prefix.Length -
1));
    typeName.Append("WebRequest");

    return (WebRequest)Activator.CreateInstance(
        System.Type.GetType(typeName));
}
```

只要 `WebRequest` 的子类对象能够遵循我们的惯例，即该类的类型名符合事先制订的规范，改进后的 `Create()` 方法就能够运行良好。以新增 `Tcp` 协议的 `WebRequest` 对象为例。该协议的 `Schema` 为 “`net.tcp`”，因此其类名必须为 “`NettcpWebRequest`”，并放在 “`System.Net`” 命名空间下。如果客户端调用 `WebRequest.Create()` 方法，并传入 “`net.tcp://www.agiledon.com`” 值，则 `Create()` 方法就会对该 `Uri` 地址进行解析，获得完整的类型名为 “`System.Net.NettcpWebRequest`”，然后，利用反射技术创建该对象。采用 “惯例优于配置” 的方式，可以极大地简化工厂方法的实现代码，抛弃了繁琐的设计理念，具有非常灵活的扩展性以及良好的代码可读性。或许，唯一的遗憾是由于反射技术带来的性能损耗。

利用抽象的方式封装变化，固然是应对需求变化的王道，但它也仅仅能解除调用者与被调用者之间的耦合关系。只要还涉及具体对象的创建，即使引入了创建型模式，例如 `Factory Method` 模式，具体工厂对象的创建依然是必不可少的。不要小看这一点点麻烦，需知 “千里之堤，溃于蚁穴”，牵一发而动全身，小麻烦可能会酿成大灾难。对于那些业已被封装变化的对象，我们还应该学会利用诸如 “依赖注入”、“表驱动法” 等技术，彻底解除两者之间的耦合；至于选择何种技术，则需要根据具体的应用场景做出判断。当然，模块或对象解耦的重要前提，则源于封装变化，要求我们针对接口编程，而不是实现。这也是 `GOF` 提出的面向对象设计原则。