.NET 设计模式开篇

——.NET 设计模式系列之一

Terrylee，2005 年 12 月 06 日

前言

加入 Design & Pattern 团队有几个月的时间了，惭愧的是从没有写过关于设计模式的随笔，得到 wayfarer 的同意，把企业库系列的随笔放在了团队的首页上。不是不想去写这样的随笔，也不是没有时间，自己初学设计模式，去写设计模式的文章，有点班门弄斧的味道。园子里吕震宇老师的《设计模式系列》和 wayfarer 的《设计之道》堪称设计模式里的经典之作。可是正如 wafarer 所说的那样，受到发表欲的蛊惑，本着交流就是进步的想法，思考再三，还是决定写这样的随笔，来对设计模式做一些探索和总结，起名曰“探索设计模式”，有些言过其实，就当是记录自己学习设计模式的历程吧，不过还是希望能得到各位前辈的指点！

设计模式

设计模式是规则吗？

地上本没有路，走得人多了也就成了路。设计模式如同此理，它是经验的传承，并非体系；是被前人发现，经过总结形成了一套某一类问题的一般性解决方案，而不是被设计出来的定性规则；它不像算法那样可以照搬照用。

设计模式是架构吗？

架构和模式应该是一个属于相互涵盖的过程，但是总体来说架构更加关注的是所谓的 High-Level Design,而模式关注的重点在于通过经验提取的“准则或指导方案”在设计中的应用，因此在不同层面考虑问题的时候就形成了不同问题域上的模式。模式的目标是，把共通问题中的不变部分和变化部分分离出来。不变的部分，就构成了模式，因此，模式是一个经验提取的“准则”，

并且在一次一次的实践中得到验证，在不同的层次有不同的模式，小到语言实现，大到架构。在不同的层面上，模式提供不同层面的指导。

设计模式，软件的永恒之道？

这个问题没有答案，有的只是讨论，看一下一位前辈结合建筑学得出的几点心得吧：

和建筑结构一样，软件中亦有诸多的"内力"。和建筑设计一样，软件设计也应该努力疏解系统中的内力，使系统趋于稳定、有生气。一切的软件设计都应该由此出发。

任何系统都需要有变化，任何系统都会走向死亡。作为设计者，应该拥抱变化、利用变化，而不是逃避变化。

好的软件只能"产生"而不能"创造"，我们所能做的只是用一个相对好的过程，尽量使软件朝向好的方向发展。

需要设计模式吗？

答案是肯定的，但你需要确定的是模式的应用是否过度？我得承认，世界上有很多天才的程序员，他可以在一段代码中包含 6 种设计模式，也可以不用模式而把设计做得很好。但我们的目标是追求有效的设计，而设计模式可以为这个目标提供某种参考模型、设计方法。

我们不需要奉 GOF 的设计模式为圭臬，但合理的运用设计模式，才是正确的抉择。很多人看过 GOF 的《Design Patterns》，对这 23 种模式也背得滚瓜烂熟。但重要的不是你熟记了多少个模式的名称，关键还在于付诸实践的运用。为了有效地设计，而去熟悉某种模式所花费的代价是值得的，因为很快你会在设计中发现这种模式真的很好，很多时候它令得你的设计更加简单了。

其实在软件设计人员中，唾弃设计模式的可能很少，盲目夸大设计模式功用的反而更多。言必谈“模式”，并不能使你成为优秀的架构师。真正出色的设计师，懂得判断运用模式的时机。

还有一个问题是，很多才踏入软件设计领域的人员，往往对设计模式很困惑。对于他们来说，由于没有项目的实际经验，00 的思想也还未曾建立，设计模式未免过于高深了。其实，即使是非常有经验的程序员，也不敢夸口对各种模式都能合理应用。[--摘自 wayfare 的设计之道]

后记

关于设计模式的理论性的文章，已经写了很多了，我不想再继续重复抄写下去，仅记录下上面几段话，用它来作探索设计模式系列的一个开篇吧。[现已更名为.NET 设计模式]

单件模式（Singleton Pattern）

——.NET 设计模式系列之二

Terrylee，2005 年 12 月 07 日

概述

Singleton 模式要求一个类有且仅有一个实例，并且提供了一个全局的访问点。这就提出了一个问题：如何绕过常规的构造器，提供一种机制来保证一个类只有一个实例？客户程序在调用某一个类时，它是不会考虑这个类是否只能有一个实例等问题的，所以，这应该是类设计者的责任，而不是类使用者的责任。

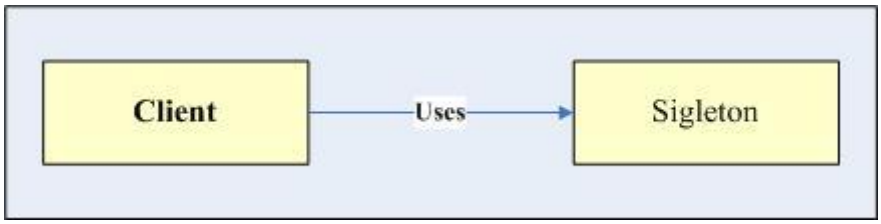
从另一个角度来说，Singleton 模式其实也是一种职责型模式。因为我们创建了一个对象，这个对象扮演了独一无二的角色，在这个单独的对象实例中，它集中了它所属类的所有权力，同时它也肩负了行使这种权力的职责！

意图

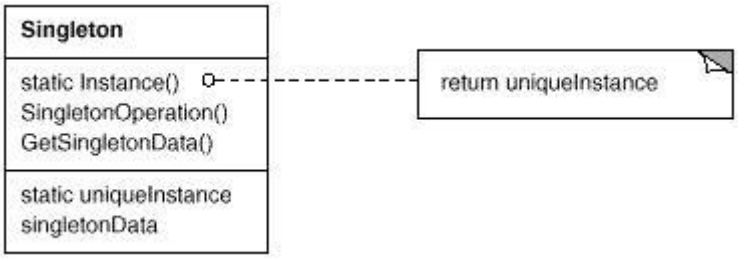
保证一个类仅有一个实例，并提供一个访问它的全局访问点。

模型图

逻辑模型图：

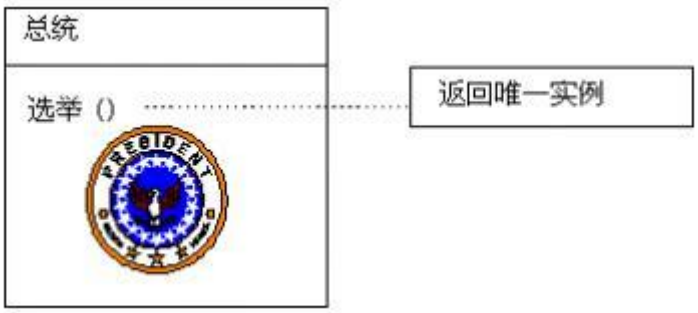


物理模型图：



生活中的例子

美国总统的职位是 Singleton，美国宪法规定了总统的选举，任期以及继任的顺序。这样，在任何时刻只能由一个现任的总统。无论现任总统的身份为何，其头衔"美利坚合众国总统"是访问这个职位的人的一个全局的访问点。



五种实现

1．简单实现

```
1  public sealed class Singleton
2  {
3      static Singleton instance=null;
4
5      Singleton()
6      {
7      }
8
9      public static Singleton Instance
10     {
11         get
12         {
13             if (instance==null)
14             {
15                 instance = new Singleton();
16             }
17             return instance;
18         }
19     }
20 }
```

这种方式的实现对于线程来说并不是安全的，因为在多线程的环境下有可能得到 Singleton 类的多个实例。如果同时有两个线程去判断（instance == null），并且得到的结果为真，这时两个线程都会创建类 Singleton 的实例，这样就违背了 Singleton 模式的原则。实际上在上述代码中，有可能在计算出表达式的值之前，对象实例已经被创建，但是内存模型并不能保证对象实例在第二个线程创建之前被发现。

该实现方式主要有两个优点：

- 由于实例是在 Instance 属性方法内部创建的，因此类可以使用附加功能（例如，对子类进行实例化），即使它可能引入不想要的依赖性。
- 直到对象要求产生一个实例才执行实例化；这种方法称为“惰性实例化”。惰性实例化避免了在应用程序启动时实例化不必要的 singleton。

2．安全的线程

```
1  public sealed class Singleton
2  {
3      static Singleton instance=null;
4      static readonly object padlock = new object();
5
6      Singleton()
7      {
8      }
9
10     public static Singleton Instance
11     {
```

```
12        get
13        {
14          lock (padlock)
15          {
16            if (instance==null)
17            {
18              instance = new Singleton();
19            }
20            return instance;
21          }
22        }
23    }
24  }
25
26
```

这种方式的实现对于线程来说是安全的。我们首先创建了一个进程辅助对象，线程在进入时先对辅助对象加锁然后再检测对象是否被创建，这样可以确保只有一个实例被创建，因为在同一个时刻加了锁的那部分程序只有一个线程可以进入。这种情况下，对象实例由最先进入的那个线程创建，后来的线程在进入时（instence == null）为假，不会再去创建对象实例了。但是这种实现方式增加了额外的开销，损失了性能。

3．双重锁定

```
1  public sealed class Singleton
2  {
3      static Singleton instance=null;
4      static readonly object padlock = new object();
```

```
5
6     Singleton()
7     {
8     }
9
10    public static Singleton Instance
11    {
12      get
13      {
14        if (instance==null)
15        {
16          lock (padlock)
17          {
18            if (instance==null)
19            {
20              instance = new Singleton();
21            }
22          }
23        }
24        return instance;
25      }
26    }
27  }
28
```

这种实现方式对多线程来说是安全的，同时线程不是每次都加锁，只有判断对象实例没有被创建时它才加锁，有了我们上面第一部分的里面的分析，我们知道，加锁后还得再进行对象是否已被创建的判断。它解决了线程并发问题，同时避免在每个 Instance 属性方法的调用中都出现独占锁定。它还允许您将实例化延迟到第一次访问对象时发生。实际上，应用程序很少需要

这种类型的实现。大多数情况下我们会用静态初始化。这种方式仍然有很多缺点：无法实现延迟初始化。

4．静态初始化

```
1  public sealed class Singleton
2  {
3      static readonly Singleton instance=new Singleton();
4
5      static Singleton()
6      {
7      }
8
9      Singleton()
10     {
11     }
12
13     public static Singleton Instance
14     {
15         get
16         {
17             return instance;
18         }
19     }
20 }
21
```

看到上面这段富有戏剧性的代码，我们可能会产生怀疑，这还是 Singleton 模式吗？在此实现中，将在第一次引用类的任何成员时创建实例。公共语言运行库负责处理变量初始化。该类

标记为 sealed 以阻止发生派生，而派生可能会增加实例。此外，变量标记为 readonly，这意味着只能在静态初始化期间（此处显示的示例）或在类构造函数中分配变量。

该实现与前面的示例类似，不同之处在于它依赖公共语言运行库来初始化变量。它仍然可以用来解决 **Singleton** 模式试图解决的两个基本问题：全局访问和实例化控制。公共静态属性为访问实例提供了一个全局访问点。此外，由于构造函数是私有的，因此不能在类本身以外实例化 **Singleton** 类；因此，变量引用的是可以在系统中存在的唯一的实例。

由于 **Singleton** 实例被私有静态成员变量引用，因此在类首次被对 **Instance** 属性的调用所引用之前，不会发生实例化。

这种方法唯一的潜在缺点是，您对实例化机制的控制权较少。在 **Design Patterns** 形式中，您能够在实例化之前使用非默认的构造函数或执行其他任务。由于在此解决方案中由 **.NET Framework** 负责执行初始化，因此您没有这些选项。在大多数情况下，静态初始化是在 **.NET** 中实现 **Singleton** 的首选方法。

5．延迟初始化

```
1  public sealed class Singleton
2  {
3      Singleton()
4      {
5      }
6
7      public static Singleton Instance
8      {
9          get
```

```
10        {
11            return Nested.instance;
12        }
13    }
14
15    class Nested
16    {
17        static Nested()
18        {
19        }
20
21        internal static readonly Singleton instance = new Singleton();
22    }
23}
24
```

这里，初始化工作有 Nested 类的一个静态成员来完成，这样就实现了延迟初始化，并具有很多的优势，是值得推荐的一种实

现方式。

实现要点

- Singleton 模式是限制而不是改进类的创建。

- Singleton 类中的实例构造器可以设置为 Protected 以允许子类派生。
- Singleton 模式一般不要支持 Icloneable 接口，因为这可能导致多个对象实例，与 Singleton 模式的初衷违背。
- Singleton 模式一般不要支持序列化，这也有可能导致多个对象实例，这也与 Singleton 模式的初衷违背。
- Singleton 只考虑了对象创建的管理，没有考虑到销毁的管理，就支持垃圾回收的平台和对象的开销来讲，我们一般没必要对其销毁进行特殊的管理。
- 理解和扩展 Singleton 模式的核心是“如何控制用户使用 new 对一个类的构造器的任意调用”。

- 可以很简单的修改一个 Singleton，使它有少数几个实例，这样做是允许的而且是有意义的。

优点

- 实例控制：Singleton 会阻止其他对象实例化其自己的 Singleton 对象的副本，从而确保所有对象都访问唯一实例
- 灵活性：因为类控制了实例化过程，所以类可以更加灵活修改实例化过程

缺点

- 开销：虽然数量很少，但如果每次对象请求引用时都要检查是否存在类的实例，将仍然需要一些开销。可以通过使用静态初始化解决此问题，上面的五种实现方式中已经说过了。

- 可能的开发混淆：使用 **singleton** 对象（尤其在类库中定义的对象）时，开发人员必须记住自己不能使用 **new** 关键字实例化对象。因为可能无法访问库源代码，因此应用程序开发人员可能会意外发现自己无法直接实例化此类。

- 对象的生存期：Singleton 不能解决删除单个对象的问题。在提供内存管理的语言中（例如基于 .NET Framework 的语言），只有 Singleton 类能够导致实例被取消分配，因为它包含对该实例的私有引用。在某些语言中（如 C++），其他类可以删除
对象实例，但这样会导致 Singleton 类中出现悬浮引用。

适用性

- 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

- 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

应用场景

- 每台计算机可以有若干个打印机，但只能有一个 Printer Spooler，避免两个打印作业同时输出到打印机。
（摘自吕震宇的 C#设计模式（7）－Singleton Pattern）

- PC 机中可能有几个串口，但只能有一个 COM1 口的实例。

- 系统中只能有一个窗口管理器。

- .NET Remoting 中服务器激活对象中的 Sigleton 对象，确保所有的客户程序的请求都只有一个实例来处理。

完整示例

这是一个简单的计数器例子，四个线程同时进行计数。

```
1  using System;
2  using System.Threading;
3
4  namespace SigletonPattern.SigletonCounter
5  {
6      /// <summary>
7      /// 功能：简单计数器的单件模式
8      /// 编写：Terrylee
9      /// 日期：2005 年 12 月 06 日
10     /// </summary>
11     public class CountSigleton
12     {
13         ///存储唯一的实例
14         static CountSigleton uniCounter = new CountSigleton();
15
16         ///存储计数值
17         private int totNum = 0;
18
19         private CountSigleton()
20
21         {
```

```
22            ///线程延迟 2000 毫秒
23            Thread.Sleep(2000);
24        }
25
26        static public CountSigleton Instance()
27
28        {
29
30            return uniCounter;
31
32        }
33
34        ///计数加 1
35        public void Add()
36        {
37            totNum ++;
38        }
39
40        ///获得当前计数值
41        public int GetCounter()
42        {
43            return totNum;
```

```
44        }
45
46    }
47}
48
```

```
1  using System;
2  using System.Threading;
3  using System.Text;
4
5  namespace SigletonPattern.SigletonCounter
6  {
7      /// <summary>
8      /// 功能：创建一个多线程计数的类
9      /// 编写：Terrylee
10     /// 日期：2005 年 12 月 06 日
11     /// </summary>
12     public class CountMutilThread
13     {
14         public CountMutilThread()
15         {
16
```

```
17      }
18
19      /// <summary>
20      /// 线程工作
21      /// </summary>
22      public static void DoSomeWork()
23      {
24          ///构造显示字符串
25          string results = "";
26
27          ///创建一个 Sigleton 实例
28          CountSigleton MyCounter = CountSigleton.Instance();
29
30          ///循环调用四次
31          for(int i=1;i<5;i++)
32          {
33              ///开始计数
34              MyCounter.Add();
35
36              results +="线程";
37              results += Thread.CurrentThread.Name.ToString() + "——〉";
38              results += "当前的计数：";
```

```
39              results += MyCounter.GetCounter().ToString();
40              results += "\n";
41
42              Console.WriteLine(results);
43
44              ///清空显示字符串
45              results = "";
46          }
47      }
48
49      public void StartMain()
50      {
51
52          Thread thread0 = Thread.CurrentThread;
53
54          thread0.Name = "Thread 0";
55
56          Thread thread1 =new Thread(new ThreadStart(DoSomeWork));
57
58          thread1.Name = "Thread 1";
59
60          Thread thread2 =new Thread(new ThreadStart(DoSomeWork));
```

```
61
62          thread2.Name = "Thread 2";
63
64          Thread thread3 =new Thread(new ThreadStart(DoSomeWork));
65
66          thread3.Name = "Thread 3";
67
68          thread1.Start();
69
70          thread2.Start();
71
72          thread3.Start();
73
74          ///线程 0 也只执行和其他线程相同的工作
75          DoSomeWork();
76      }
77  }
78 }
79
```

```
1 using System;
2 using System.Text;
```

```
3  using System.Threading;
4
5  namespace SigletonPattern.SigletonCounter
6  {
7      /// <summary>
8      /// 功能：实现多线程计数器的客户端
9      /// 编写：Terrylee
10     /// 日期：2005 年 12 月 06 日
11     /// </summary>
12     public class CountClient
13     {
14         public static void Main(string[] args)
15         {
16         CountMutilThread cmt = new CountMutilThread();
17
18             cmt.StartMain();
19
20             Console.ReadLine();
21         }
22     }
23 }
24
```

总结

Singleton 设计模式是一个非常有用的机制，可用于在面向对象的应用程序中提供单个访问点。文中通过五种实现方式的比较和一个完整的示例，完成了对 Singleton 模式的一个总结和探索。用一句广告词来概括 Singleton 模式就是“简约而不简单”。

抽象工厂模式（Abstract Factory）

——探索设计模式系列之三

Terrylee，2005 年 12 月 12 日

概述

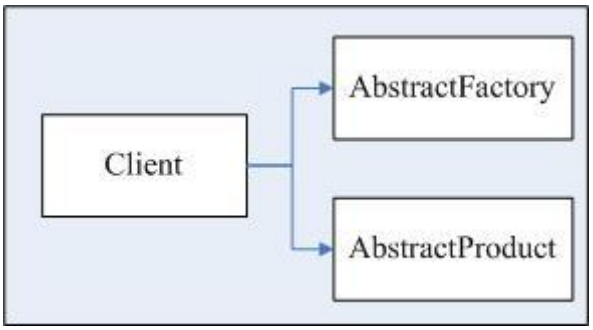
在软件系统中，经常面临着“一系列相互依赖的对象”的创建工作；同时由于需求的变化，往往存在着更多系列对象的创建工作。如何应对这种变化？如何绕过常规的对象的创建方法（*new*），提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合？这就是我们要说的抽象工厂模式。

意图

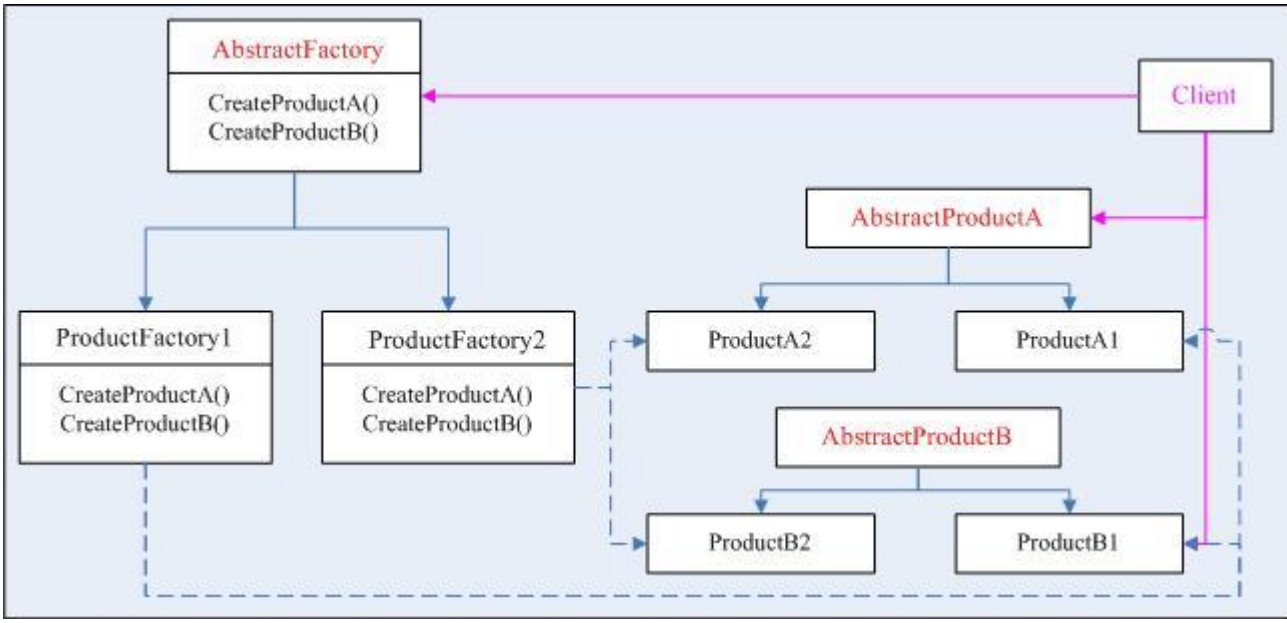
提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

模型图

逻辑模型：

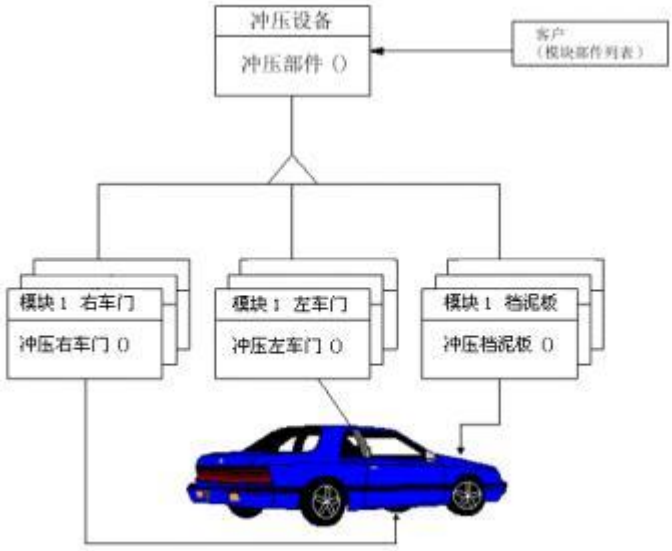


物理模型：



生活中的例子

抽象工厂的目的是要提供一个创建一系列相关或相互依赖对象的接口，而不需要指定它们具体的类。这种模式可以汽车制造厂所使用的金属冲压设备中找到。这种冲压设备可以制造汽车车身部件。同样的机械用于冲压不同的车型的右边车门、左边车门、右前挡泥板、左前挡泥板和引擎罩等等。通过使用转轮来改变冲压盘，这个机械产生的具体类可以在三分钟内改变。



抽象工厂之新解

虚拟案例

中国企业需要一项简单的财务计算：每月月底，财务人员要计算员工的工资。

员工的工资 = (基本工资 + 奖金 - 个人所得税)。这是一个放之四海皆准的运算法则。

为了简化系统，我们假设员工基本工资总是 4000 美金。

中国企业奖金和个人所得税的计算规则是:

奖金 = 基本工资(4000) * 10%

个人所得税 = (基本工资 + 奖金) * 40%

我们现在要为此构建一个软件系统（代号叫 Softo），满足中国企业的需求。

案例分析

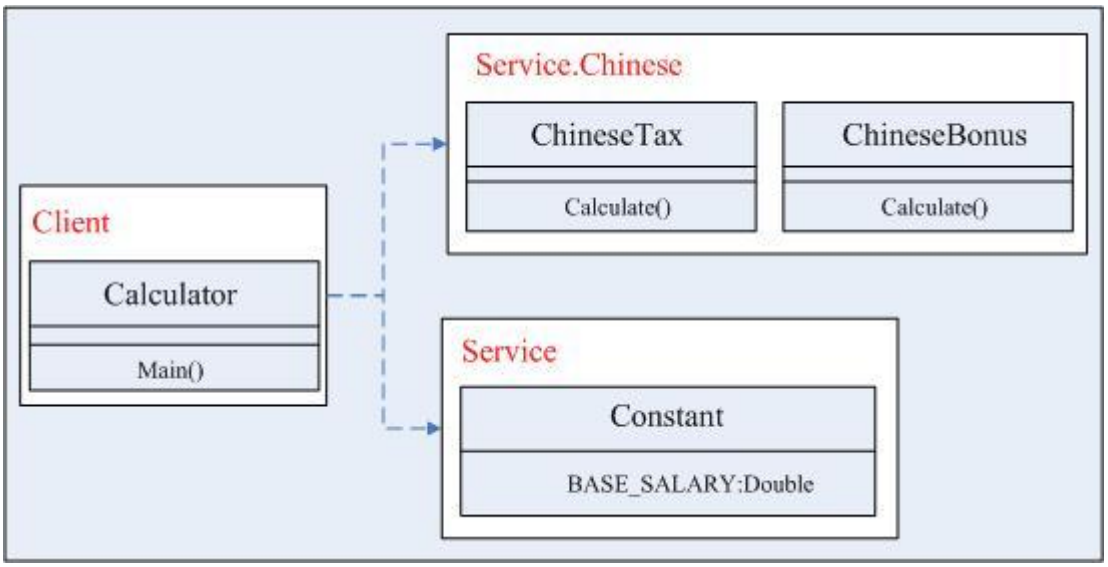
奖金(Bonus)、个人所得税(Tax)的计算是 Softo 系统的业务规则(Service)。

工资的计算(Calculator)则调用业务规则(Service)来计算员工的实际工资。

工资的计算作为业务规则的前端(或者客户端 Client)将提供给最终使用该系统的用户(财务人员)使用。

针对中国企业为系统建模

根据上面的分析，为 Softo 系统建模如下：



则业务规则 Service 类的代码如下：

```
1  using System;
2
3  namespace ChineseSalary
4  {
5      /// <summary>
6      /// 公用的常量
7      /// </summary>
8      public class Constant
```

```
9    {
10       public static double BASE_SALARY = 4000;
11   }
12 }
```

```
1  using System;
2
3  namespace ChineseSalary
4  {
5      /// <summary>
6      /// 计算中国个人奖金
7      /// </summary>
8      public class ChineseBonus
9      {
10         public double Calculate()
11         {
12             return Constant.BASE_SALARY * 0.1;
13         }
14     }
15 }
16
```

客户端的调用代码：

```
using System;

namespace ChineseSalary
{
    /// <summary>
    /// 计算中国个人所得税
    /// </summary>
    public class ChineseTax
    {
        public double Calculate()
        {
            return (Constant.BASE_SALARY + (Constant.BASE_SALARY * 0.1)) * 0.4;
        }
    }
}

```

运行程序，输入的结果如下：

```
Chinese Salary is: 2640
```

针对美国企业为系统建模

为了拓展国际市场，我们要把该系统移植给美国公司使用。

美国企业的工资计算同样是：员工的工资 = 基本工资 + 奖金 - 个人所得税。

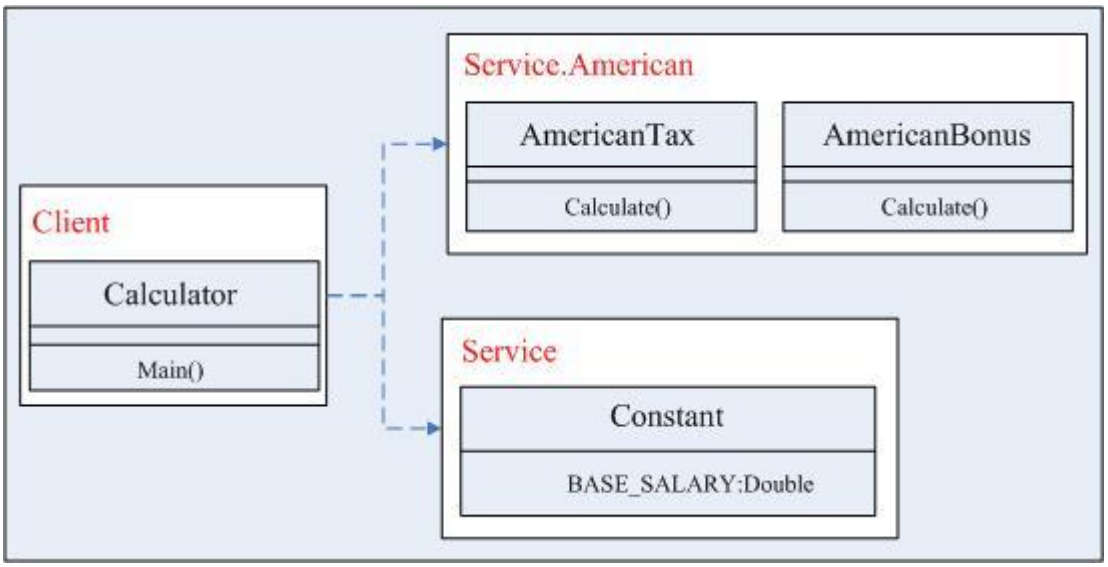
但是他们的奖金和个人所得税的计算规则不同于中国企业：

美国企业奖金和个人所得税的计算规则是：

奖金 = 基本工资 * 15 %

个人所得税 =（基本工资 * 5% + 奖金 * 25%）

根据前面为中国企业建模经验，我们仅仅将 ChineseTax、ChineseBonus 修改为 AmericanTax、AmericanBonus。 修改后的模型如下：



则业务规则 Service 类的代码如下：

```
1 using System;
2
3 namespace AmericanSalary
4 {
```

```
5    /// <summary>
6    /// 公用的常量
7    /// </summary>
8    public class Constant
9    {
10       public static double BASE_SALARY = 4000;
11   }
12 }
13
```

```
1  using System;
2
3  namespace AmericanSalary
4  {
5      /// <summary>
6      /// 计算美国个人奖金
7      /// </summary>
8      public class AmericanBonus
9      {
10         public double Calculate()
11         {
12             return Constant.BASE_SALARY * 0.1;
```

```
13        }
14    }
15}
16
```

```
1  using System;
2
3  namespace AmericanSalary
4{
5    /// <summary>
6    /// 计算美国个人所得税
7    /// </summary>
8    public class AmericanTax
9    {
10       public double Calculate()
11       {
12           return (Constant.BASE_SALARY + (Constant.BASE_SALARY * 0.1)) * 0.4;
13       }
14   }
15}
16
```

客户端的调用代码：

```
1
2  using System;
3
4  namespace AmericanSalary
5  {
6      /// <summary>
7      /// 客户端程序调用
8      /// </summary>
9      public class Calculator
10     {
11         public static void Main(string[] args)
12         {
13             AmericanBonus bonus = new AmericanBonus();
14             double bonusValue  = bonus.Calculate();
15
16             AmericanTax tax = new AmericanTax();
17             double taxValue = tax.Calculate();
18
19             double salary = 4000 + bonusValue - taxValue;
20
21             Console.WriteLine("American Salary is：" + salary);
22             Console.ReadLine();
```

```
23        }
24      }
25  }
26
```

运行程序，输入的结果如下：

American Salary is：2640

整合成通用系统

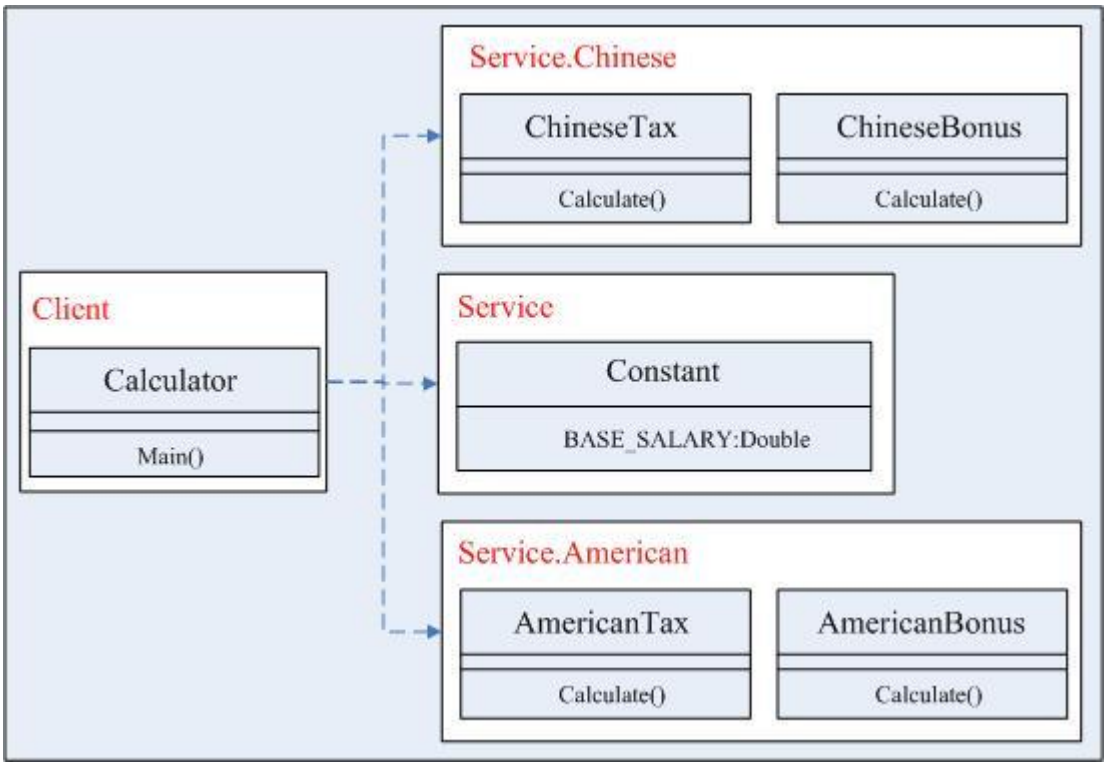
让我们回顾一下该系统的发展历程：

最初，我们只考虑将 Softo 系统运行于中国企业。但随着 MaxDO 公司业务向海外拓展， MaxDO 需要将该系统移植给美国使用。

移植时，MaxDO 不得不抛弃中国企业的业务规则类 ChineseTax 和 ChineseBonus， 然后为美国企业新建两个业务规则类: AmericanTax,AmericanBonus。最后修改了业务规则调用 Calculator 类。

结果我们发现：每当 Softo 系统移植的时候，就抛弃原来的类。现在，如果中国联想集团要购买该系统，我们不得不再次抛弃 AmericanTax,AmericanBonus，修改回原来的业务规则。

一个可以立即想到的做法就是在系统中保留所有业务规则模型，即保留中国和美国企业工资运算规则。



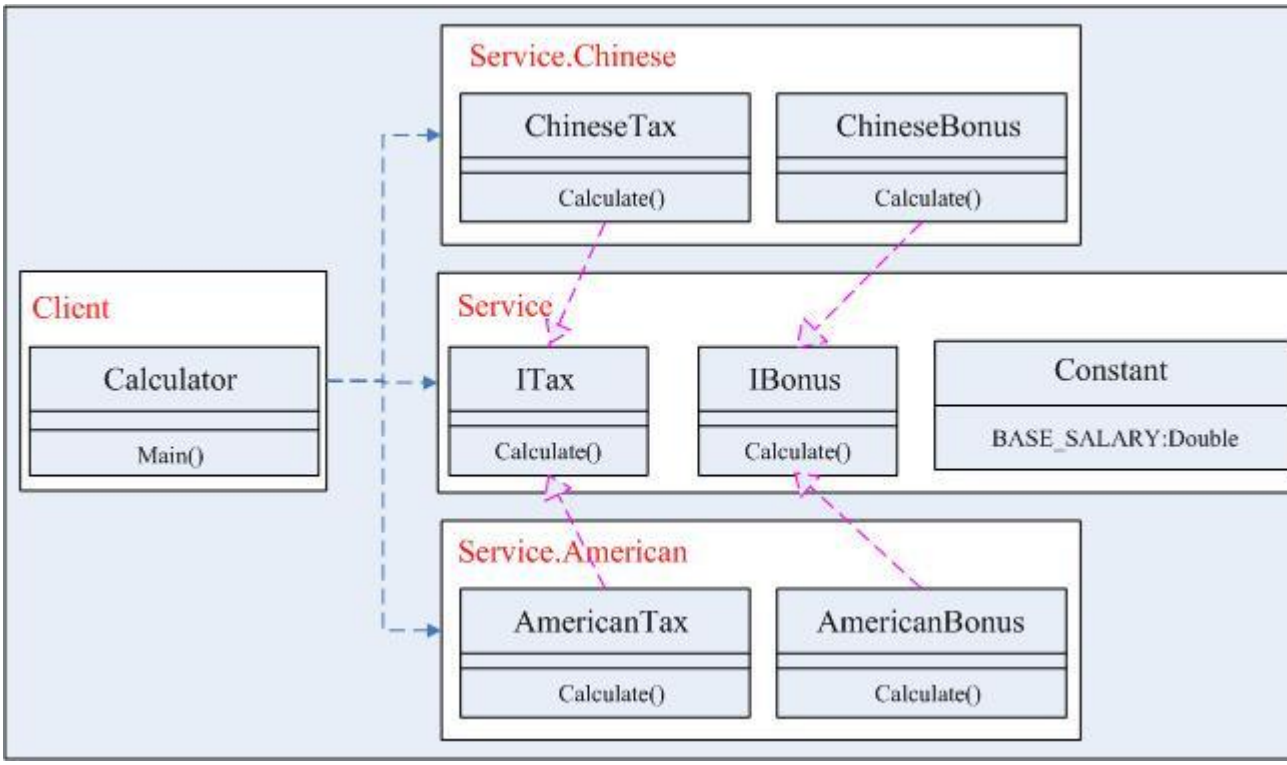
通过保留中国企业和美国企业的业务规则模型，如果该系统在美国企业和中国企业之间切换时，我们仅仅需要修改 Caculator 类即可。

让移植工作更简单

前面系统的整合问题在于：当系统在客户在美国和中国企业间切换时仍然需要修改 Caculator 代码。

一个维护性良好的系统应该遵循“开闭原则”。即：封闭对原来代码的修改，开放对原来代码的扩展（如类的继承，接口的实现）

我们发现不论是中国企业还是美国企业，他们的业务运规则都采用同样的计算接口。 于是很自然地想到建立两个业务接口类 Tax，Bonus，然后让 AmericanTax、AmericanBonus 和 ChineseTax、ChineseBonus 分别实现这两个接口， 据此修正后的模型如下：



此时客户端代码如下：

```
1
2  using System;
3
4  namespace InterfaceSalary
5  {
6      /// <summary>
7      /// 客户端程序调用
8      /// </summary>
9      public class Calculator
10     {
```

```
11        public static void Main(string[] args)
12        {
13            Bonus bonus = new ChineseBonus();
14            double bonusValue  = bonus.Calculate();
15
16            Tax tax = new ChineseTax();
17            double taxValue = tax.Calculate();
18
19            double salary = 4000 + bonusValue - taxValue;
20
21            Console.WriteLine("Chinaese Salary is：" + salary);
22            Console.ReadLine();
23        }
24    }
25}
26
```

为业务规则增加工厂方法

然而，上面增加的接口几乎没有解决任何问题，因为当系统的客户在美国和中国企业间切换时 Caculator 代码仍然需要修改。

只不过修改少了两处，但是仍然需要修改 ChineseBonus,ChineseTax 部分。致命的问题是：我们需要将这个移植工作转包给一个叫 Hippo 的软件公司。 由于版权问题，我们并未提供 Soft

o 系统的源码给 Hippo 公司，因此 Hippo 公司根本无法修改 Calculator，导致实际上移植工作无法进行。

为此，我们考虑增加一个工具类(命名为 Factory)，代码如下：

```
1  using System;
2
3  namespace FactorySalary
4  {
5      /// <summary>
6      /// Factory 类
7      /// </summary>
8      public class Factory
9      {
10         public Tax CreateTax()
11         {
12             return new ChineseTax();
13         }
14
15         public Bonus CreateBonus()
16         {
17             return new ChineseBonus();
18         }
19     }
```

```
}

```

修改后的客户端代码：

```
using System;

namespace FactorySalary
{
    /// <summary>
    /// 客户端程序调用
    /// </summary>
    public class Calculator
    {
        public static void Main(string[] args)
        {
            Bonus bonus = new Factory().CreateBonus();
            double bonusValue  = bonus.Calculate();

            Tax tax = new Factory().CreateTax();
            double taxValue = tax.Calculate();

            double salary = 4000 + bonusValue - taxValue;
```

```
20
21          Console.WriteLine("Chinaese Salary is：" + salary);
22          Console.ReadLine();
23      }
24  }
25 }
26
```

不错，我们解决了一个大问题，设想一下：当该系统从中国企业移植到美国企业时，我们现在需要做什么？

答案是：对于 Caculator 类我们什么也不用做。我们需要做的是修改 Factory 类，修改结果如下：

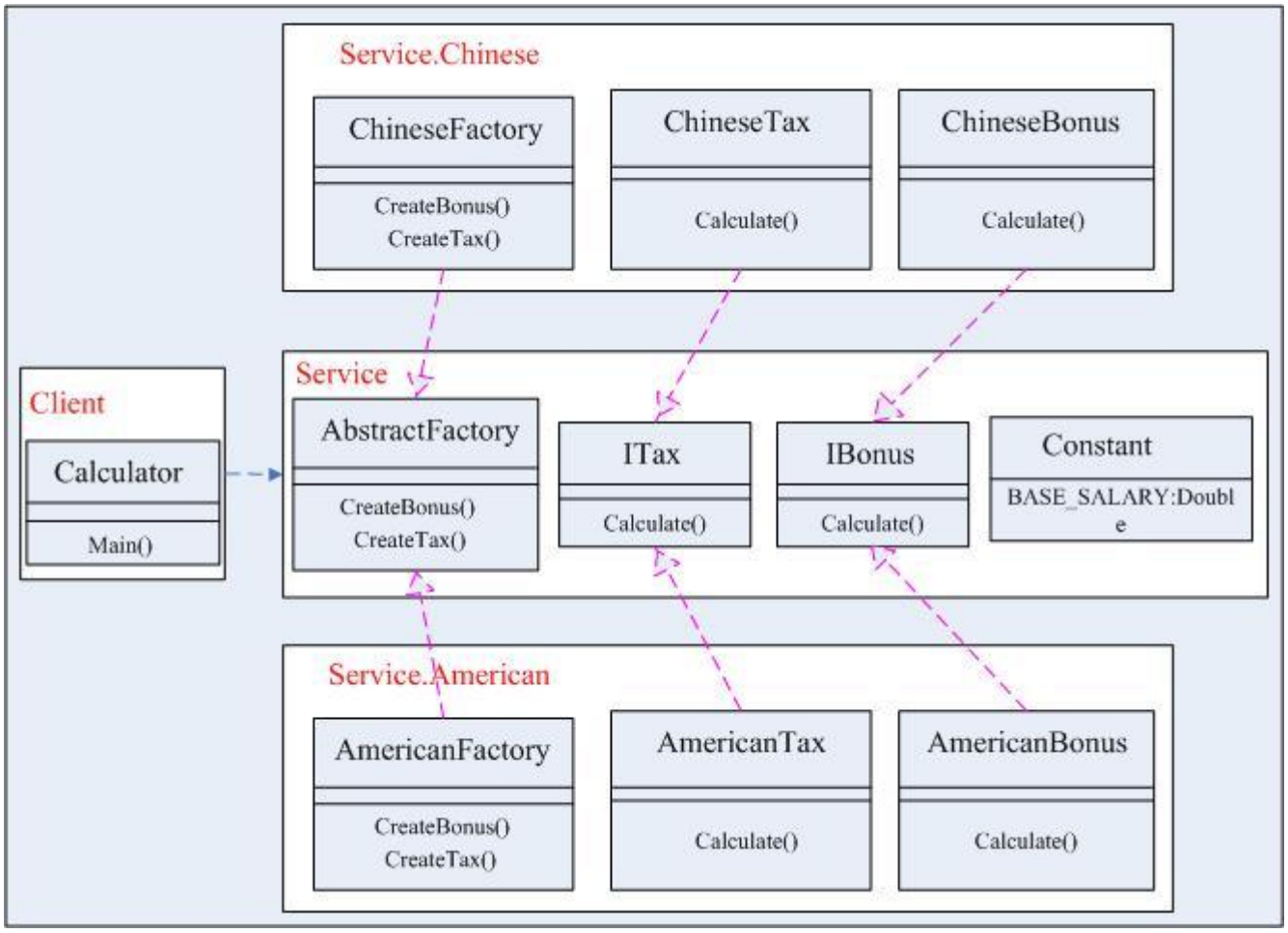
```
1  using System;
2
3  namespace FactorySalary
4 {
5     /// <summary>
6     /// Factory 类
7     /// </summary>
8     public class Factory
9     {
10        public Tax CreateTax()
11        {
```

```
12          return new AmericanTax();
13      }
14
15      public Bonus CreateBonus()
16      {
17          return new AmericanBonus();
18      }
19  }
20 }
21
```

为系统增加抽象工厂方法

很显然，前面的解决方案带来了一个副作用：就是系统不但增加了新的类 Factory，而且当系统移植时，移植工作仅仅是转移到 Factory 类上，工作量并没有任何缩减，而且还是要修改系统的源码。 从 Factory 类在系统移植时修改的内容我们可以看出：实际上它是专属于美国企业或者中国企业的。名称上应该叫 AmericanFactory,ChineseFactory 更合适.

解决方案是增加一个抽象工厂类 AbstractFactory，增加一个静态方法，该方法根据一个配置文件(App.config 或者 Web.config) 一个项(比如 factoryName)动态地判断应该实例化哪个工厂类，这样，我们就把移植工作转移到了对配置文件的修改。修改后的模型和代码：



抽象工厂类的代码如下：

```
1  using System;
2  using System.Reflection;
3
4  namespace AbstractFactory
5  {
6      /// <summary>
7      /// AbstractFactory 类
8      /// </summary>
```

```
9   public abstract class AbstractFactory
10  {
11      public static AbstractFactory GetInstance()
12      {
13          string factoryName = Constant.STR_FACTORYNAME.ToString();
14
15          AbstractFactory instance;
16
17          if(factoryName == "ChineseFactory")
18              instance = new ChineseFactory();
19          else if(factoryName == "AmericanFactory")
20              instance = new AmericanFactory();
21          else
22              instance = null;
23
24          return instance;
25      }
26
27      public abstract Tax CreateTax();
28
29      public abstract Bonus CreateBonus();
```

```
30     }
31 }
```

配置文件：

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <configuration>
3      <appSettings>
4          <add key="factoryName" value="AmericanFactory"></add>
5      </appSettings>
6  </configuration>
7
```

采用上面的解决方案，当系统在美国企业和中国企业之间切换时，我们需要做什么移植工作？

答案是：我们仅仅需要修改配置文件，将 factoryName 的值改为 American。

修改配置文件的工作很简单，只要写一篇幅配置文档说明书提供给移植该系统的团队(比如 Hippo 公司）就可以方便地切换使该系统运行在美国或中国企业。

最后的修正（不是最终方案）

前面的解决方案几乎很完美，但是还有一点瑕疵，瑕疵虽小，但可能是致命的。

考虑一下，现在日本 NEC 公司决定购买该系统，NEC 公司的工资的运算规则遵守的是日本的法律。如果采用上面的系统构架，这个移植我们要做哪些工作呢?

1. 增加新的业务规则类 JapaneseTax, JapaneseBonus 分别实现 Tax 和 Bonus 接口。

2. 修改 AbstractFactory 的 getInstance 方法，增加 else if(factoryName.equals("Japanese")){....

注意：系统中增加业务规则类不是模式所能解决的，无论采用什么设计模式，JapaneseTax, JapaneseBonus 总是少不了的。（即增加了新系列产品）

我们真正不能接受的是：我们仍然修要修改系统中原来的类(AbstractFactory)。前面提到过该系统的移植工作，我们可能转包给一个叫 Hippo 的软件公司。 为了维护版权，未将该系统的源码提供给 Hippo 公司，那么 Hippo 公司根本无法修改 AbstractFactory，所以系统移植其实无从谈起，或者说系统移植总要开发人员亲自参与。

解决方案是将抽象工厂类中的条件判断语句，用.NET 中发射机制代替，修改如下：

```
1  using System;
2  using System.Reflection;
3
4  namespace AbstractFactory
5  {
6      /// <summary>
7      /// AbstractFactory 类
8      /// </summary>
9      public abstract class AbstractFactory
10     {
11         public static AbstractFactory GetInstance()
12         {
```

```
13          string factoryName = Constant.STR_FACTORYNAME.ToString();
14
15          AbstractFactory instance;
16
17          if(factoryName != "")
18              instance = (AbstractFactory)Assembly.Load(factoryName).CreateInstance(factoryNam
e);
19          else
20              instance = null;
21
22          return instance;
23      }
24
25      public abstract Tax CreateTax();
26
27      public abstract Bonus CreateBonus();
28  }
29 }
30
```

这样，在我们编写的代码中就不会出现 Chinese，American，Japanese 等这样的字眼了。

小结

最后那幅图是最终版的系统模型图。我们发现作为客户端角色的 Calculator 仅仅依赖抽象类， 它不必去理解中国和美国企业具体的业务规则如何实现，Calculator 面对的仅仅是业务规则接口 Tax 和 Bonus。

Softo 系统的实际开发的分工可能是一个团队专门做业务规则，另一个团队专门做前端的业务规则组装。 抽象工厂模式有助于这样的团队的分工：两个团队通讯的约定是业务接口，由抽象工厂作为纽带粘合业务规则和前段调用，大大降低了模块间的耦合性，提高了团队开发效率。

完完全全地理解抽象工厂模式的意义非常重大，可以说对它的理解是你对 OOP 理解上升到一个新的里程碑的重要标志。 学会了用抽象工厂模式编写框架类，你将理解 OOP 的精华:面向接口编程. 。

应对“新对象”

抽象工厂模式主要在于应对“新系列”的需求变化。其缺点在于难于应付“新对象”的需求变动。如果在开发中出现了新对象，该如何去解决呢？这个问题并没有一个好的答案，下面我们看一下李建忠老师的回答：

“GOF《设计模式》中提出过一种解决方法，即给创建对象的操作增加参数，但这种做法并不能令人满意。事实上，对于新系列加新对象，就我所知，目前还没有完美的做法，只有一些演化的思路，这种变化实在是太剧烈了，因为系统对于新的对象是完全陌生的。”

实现要点

- 抽象工厂将产品对象的创建延迟到它的具体工厂的子类。
- 如果没有应对“多系列对象创建”的需求变化，则没有必要使用抽象工厂模式，这时候使用简单的静态工厂完全可以。

- 系列对象指的是这些对象之间有相互依赖、或作用的关系，例如游戏开发场景中的“道路”与“房屋”的依赖，“道路”与“地道”的依赖。

- 抽象工厂模式经常和工厂方法模式共同组合来应对“对象创建”的需求变化。

- 通常在运行时刻创建一个具体工厂类的实例，这一具体工厂的创建具有特定实现的产品对象，为创建不同的产品对象，客户应使用不同的具体工厂。

- 把工厂作为单件，一个应用中一般每个产品系列只需一个具体工厂的实例，因此，工厂通常最好实现为一个单件模式。

- 创建产品，抽象工厂仅声明一个创建产品的接口，真正创建产品是由具体产品类创建的，最通常的一个办法是为每一个产品定义一个工厂方法，一个具体的工厂将为每个产品重定义该工厂方法以指定产品，虽然这样的实现很简单，但它确要求每个产品系列都要有一个新的具体工厂子类，即使这些产品系列的差别很小。

优点

- 分离了具体的类。抽象工厂模式帮助你控制一个应用创建的对象的类，因为一个工厂封装创建产品对象的责任和过程。它将客户和类的实现分离，客户通过他们的抽象接口操纵实例，产品的类名也在具体工厂的实现中被分离，它们不出现在客户代码中。

- 它使得易于交换产品系列。一个具体工厂类在一个应用中仅出现一次——即在它初始化的时候。这使得改变一个应用的具体工厂变得很容易。它只需改变具体的工厂即可使用不同的产品配置，这是因为一个抽象工厂创建了一个完整的产品系列，所以整个产品系列会立刻改变。

- 它有利于产品的一致性。当一个系列的产品对象被设计成一起工作时，一个应用一次只能使用同一个系列中的对象，这一点很重要，而抽象工厂很容易实现这一点。

缺点

- 难以支持新种类的产品。难以扩展抽象工厂以生产新种类的产品。这是因为抽象工厂几口确定了可以被创建的产品集合，支持新种类的产品就需要扩展该工厂接口，这将涉及抽象工厂类及其所有子类的改变。

适用性

在以下情况下应当考虑使用抽象工厂模式：

- 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。
- 这个系统有多于一个的产品族，而系统只消费其中某一产品族。
- 同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。
- 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

应用场景

- 支持多种观感标准的用户界面工具箱（Kit）。
- 游戏开发中的多风格系列场景，比如道路，房屋，管道等。
- ……

总结

总之，抽象工厂模式提供了一个创建一系列相关或相互依赖对象的接口，运用抽象工厂模式的关键点在于应对“多系列对象创建”的需求变化。一句话，学会了抽象工厂模式，你将理解 OOP 的精华：面向接口编程。

建造者模式（Builder Pattern）

——.NET 设计模式系列之四

Terrylee，2005 年 12 月 17 日

概述

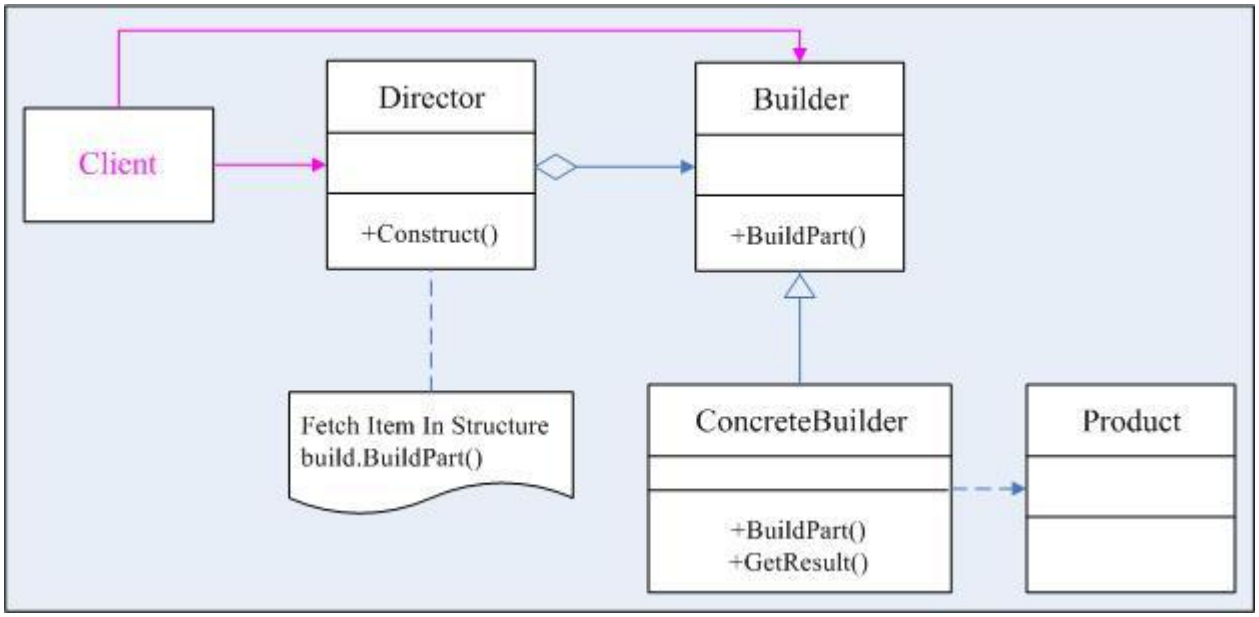
在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法确相对稳定。如何应对这种变化？如何提供一种“封装机制”来隔离出“复杂对象的各个部分”的变化，从而保持系统中的“稳定构建算法”不随着需求改变而改变？这就是要说的建造者模式。

本文通过现实生活中的买 KFC 的例子，用图解的方式来诠释建造者模式。

意图

将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

模型图

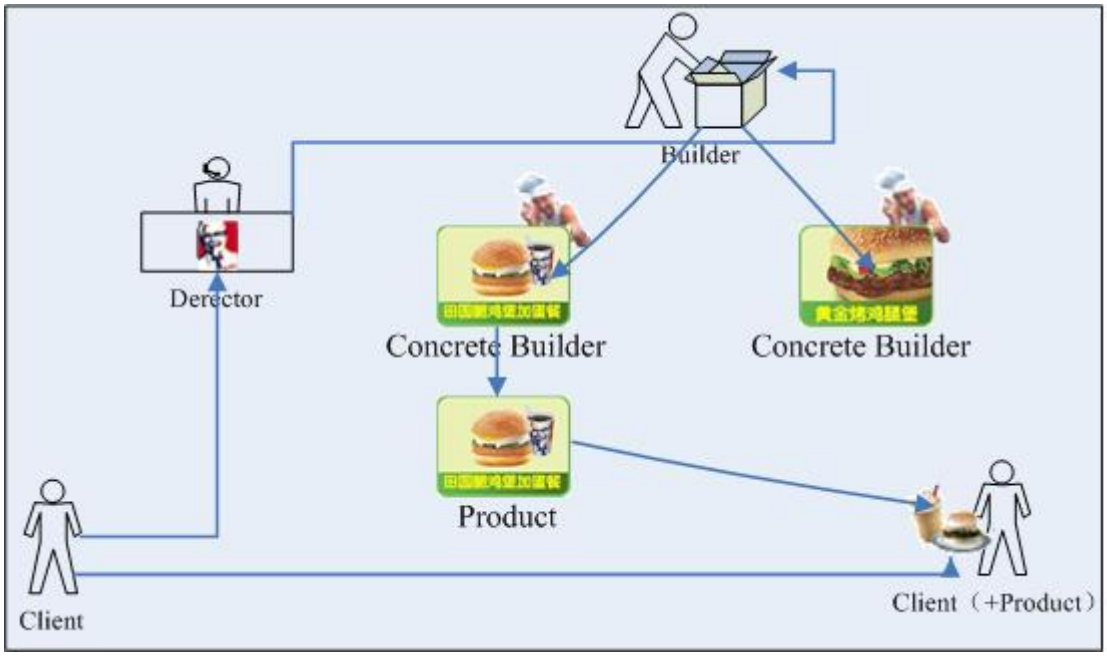


生活中的例子

生成器模式将复杂对象的构建与对象的表现分离开来，这样使得同样的构建过程可以创建出不同的表现。这种模式用于快餐店制作儿童餐。典型的儿童餐包括一个主食，一个辅食，一杯饮料和一个玩具（例如汉堡、炸鸡、可乐和玩具车）。这些在不同的儿童餐中可以是不同的，但是组合成儿童餐的过程是相同的。无论顾客点的是汉堡，三名治还是鸡肉，过程都是一样的。柜台的员工直接把主食，辅食和玩具放在一起。这些是放在一个袋子中的。饮料被倒入杯中，放在袋子外边。这些过程在相互竞争的餐馆中是同样的。

实现过程图解

在这里我们还是以去 KFC 店买套餐为例子，示意图如下：



客户端：顾客。想去买一套套餐（这里面包括汉堡，可乐，薯条），可以有 1 号和 2 号两种套餐供顾客选择。

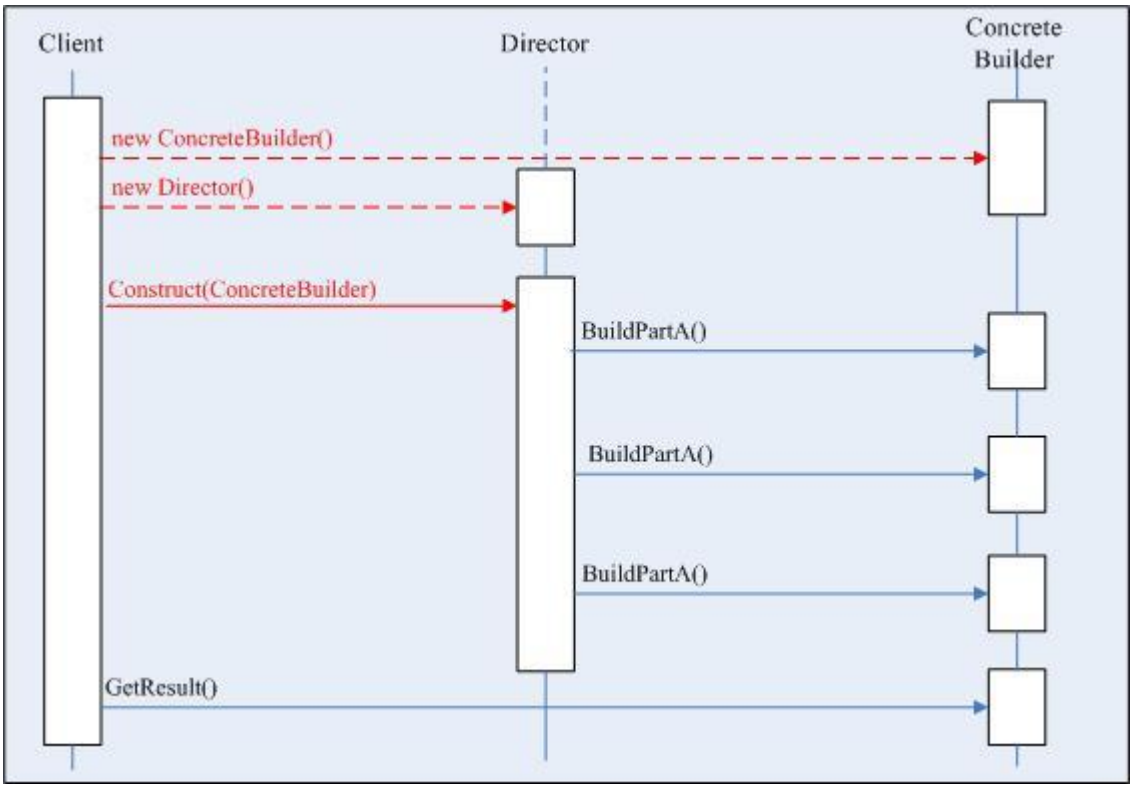
指导者角色：收银员。知道顾客想要买什么样的套餐，并告诉餐馆员工去准备套餐。

建造者角色：餐馆员工。按照收银员的要求去准备具体的套餐，分别放入汉堡，可乐，薯条等。

产品角色：最后的套餐，所有的东西放在同一个盘子里面。

下面开始我们的买套餐过程。

1．客户创建 Derector 对象，并用它所想要的 Builder 对象进行配置。顾客进入 KFC 店要买套餐，先找到一个收银员，相当于创建了一个指导者对象。这位收银员给出两种套餐供顾客选择：1 普通套餐，2 黄金套餐。完成的工作如时序图中红色部分所示。



程序实现：

```
1  using System;
2  using System.Configuration;
3  using System.Reflection;
4
5  namespace KFC
6  {
7      /// <summary>
8      /// Client 类
9      /// </summary>
10     public class Client
11     {
```

```
12        public static void Main(string[] args)
13        {
14            FoodManager foodmanager = new FoodManager();
15
16            Builder instance;
17
18            Console.WriteLine("Please Enter Food No:");
19
20            string No = Console.ReadLine();
21
22            string foodType = ConfigurationSettings.AppSettings["No"+No];
23
24            instance = (Builder)Assembly.Load("KFC").CreateInstance("KFC." + f
oodType);
25
26            foodmanager.Construct(instance);
27        }
28    }
29}
30
```

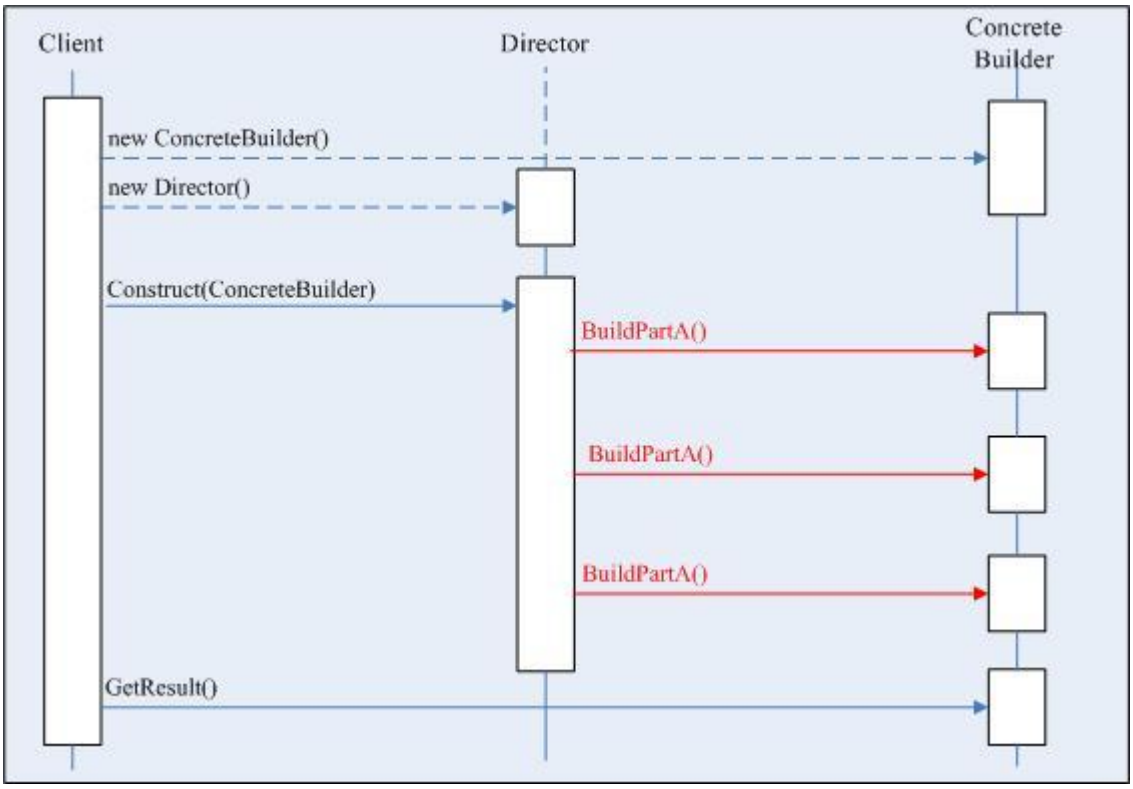
产品（套餐）类：

```
1  using System;
2  using System.Collections;
3
4  namespace KFC
5  {
6      /// <summary>
7      /// Food 类，即产品类
8      /// </summary>
9      public class Food
10     {
11         Hashtable food = new Hashtable();
12
13         /// <summary>
14         /// 添加食品
15         /// </summary>
16         /// <param name="strName">食品名称</param>
17         /// <param name="Price">价格</param>
18         public void Add(string strName,string Price)
19         {
20             food.Add(strName,Price);
21         }
22
```

```
23        /// <summary>
24        /// 显示食品清单
25        /// </summary>
26        public void Show()
27        {
28            IDictionaryEnumerator myEnumerator  = food.GetEnumerator();
29            Console.WriteLine("Food List:");
30            Console.WriteLine("----------------------------");
31            string strfoodlist = "";
32            while(myEnumerator.MoveNext())
33            {
34                strfoodlist = strfoodlist + "\n\n" + myEnumerator.Key.ToString();
35                strfoodlist = strfoodlist + ":\t" +myEnumerator.Value.ToString();
36            }
37            Console.WriteLine(strfoodlist);
38            Console.WriteLine("\n----------------------------");
39        }
40    }
41 }
42
```

2．指导者通知建造器。收银员（指导者）告知餐馆员工准备套餐。这里我们准备套餐的顺序是：放入汉堡，可乐倒入杯中，薯条放入盒中，并把这些东西都放在盘子上。这个过程对于普

通套餐和黄金套餐来说都是一样的，不同的是它们的汉堡，可乐，薯条价格不同而已。如时序图红色部分所示：



程序实现：

```
1  using System;
2
3  namespace KFC
4  {
5      /// <summary>
6      /// FoodManager 类，即指导者
7      /// </summary>
8      public class FoodManager
9      {
```

```
10        public void Construct(Builder builder)
11        {
12            builder.BuildHamb();
13
14            builder.BuildCoke();
15
16            builder.BuildChip();
17        }
18    }
19}
20
```

3．建造者处理指导者的要求，并将部件添加到产品中。餐馆员工（建造者）按照收银员要求的把对应的汉堡，可乐，薯条放入盘子中。这部分是建造者模式里面富于变化的部分，因为顾客选择的套餐不同，套餐的组装过程也不同，这步完成产品对象的创建工作。

程序实现：

```
1  using System;
2
3  namespace KFC
4  {
5      /// <summary>
6      /// Builder 类，即抽象建造者类，构造套餐
7      /// </summary>
```

```
8   public abstract class Builder
9   {
10      /// <summary>
11      /// 添加汉堡
12      /// </summary>
13      public abstract void BuildHamb();
14
15      /// <summary>
16      /// 添加可乐
17      /// </summary>
18      public abstract void BuildCoke();
19
20      /// <summary>
21      /// 添加薯条
22      /// </summary>
23      public abstract void BuildChip();
24
25      /// <summary>
26      /// 返回结果
27      /// </summary>
28      /// <returns>食品对象</returns>
29      public abstract Food GetFood();
```

```
30    }
31 }
32
```

```
1  using System;
2
3  namespace KFC
4  {
5      /// <summary>
6      /// NormalBuilder 类，具体构造者，普通套餐
7      /// </summary>
8      public class NormalBuilder:Builder
9      {
10         private Food NormalFood = new Food();
11
12         public override void BuildHamb()
13         {
14             NormalFood.Add("NormalHamb","￥10.50");
15         }
16
17         public override void BuildCoke()
18         {
```

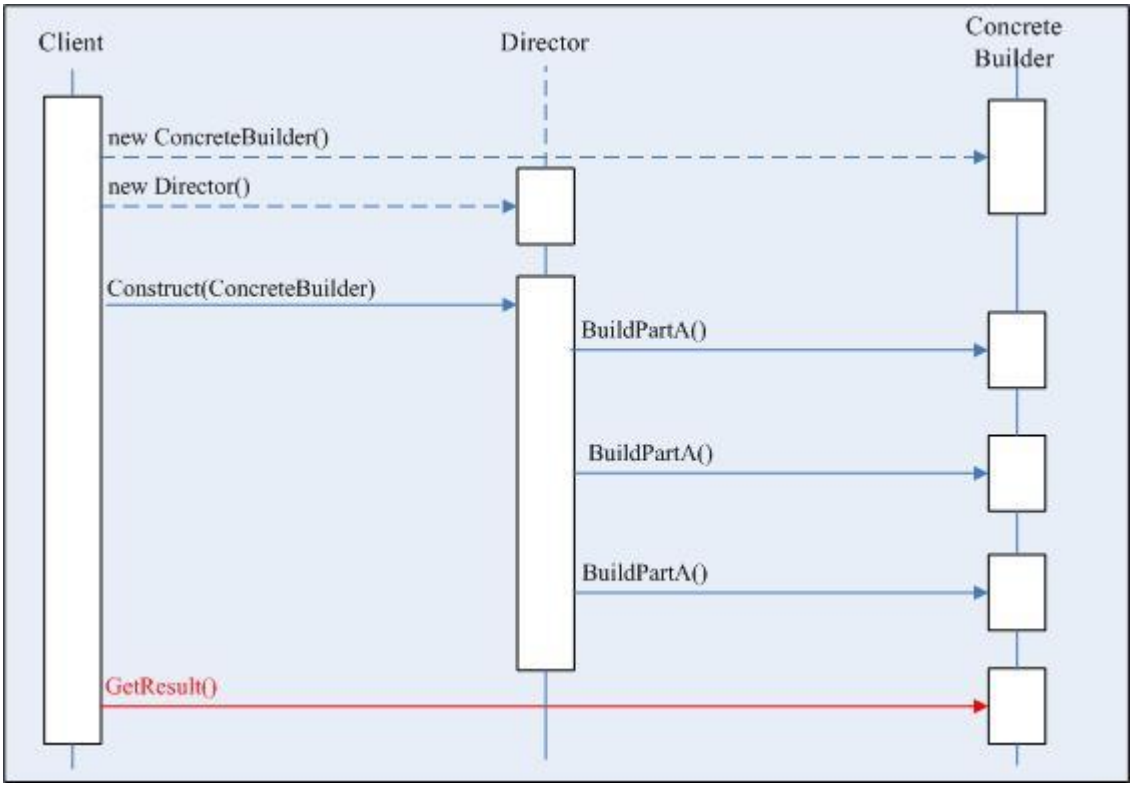
```
19            NormalFood.Add("CokeCole","￥4.50");
20        }
21
22        public override void BuildChip()
23        {
24            NormalFood.Add("FireChips","￥2.00");
25        }
26
27        public override Food GetFood()
28        {
29            return NormalFood;
30        }
31
32    }
33 }
34
```

```
1  using System;
2
3  namespace KFC
4  {
5      /// <summary>
```

```
6   /// GoldBuilder 类，具体构造者，黄金套餐
7   /// </summary>
8   public class GoldBuilder:Builder
9   {
10      private Food GoldFood = new Food();
11
12      public override void BuildHamb()
13      {
14          GoldFood.Add("GoldHamb","￥13.50");
15      }
16
17      public override void BuildCoke()
18      {
19          GoldFood.Add("CokeCole","￥4.50");
20      }
21
22      public override void BuildChip()
23      {
24          GoldFood.Add("FireChips","￥3.50");
25      }
26
27      public override Food GetFood()
```

```
28        {
29            return GoldFood;
30        }
31
32    }
33}
34
```

4．客户从建造者检索产品。从餐馆员工准备好套餐后，顾客再从餐馆员工那儿拿回套餐。这步客户程序要做的仅仅是取回已经生成的产品对象，如时序图中红色部分所示。



完整的客户程序：

```
1  using System;
2  using System.Configuration;
3  using System.Reflection;
4
5  namespace KFC
6  {
7      /// <summary>
8      /// Client 类
9      /// </summary>
10     public class Client
11     {
12         public static void Main(string[] args)
13         {
14             FoodManager foodmanager = new FoodManager();
15
16             Builder instance;
17
18             Console.WriteLine("Please Enter Food No:");
19
20             string No = Console.ReadLine();
21
22             string foodType = ConfigurationSettings.AppSettings["No"+No];
```

```
23
24          instance = (Builder)Assembly.Load("KFC").CreateInstance("KFC." + f
oodType);
25
26          foodmanager.Construct(instance);
27
28          Food food = instance.GetFood();
29          food.Show();
30
31          Console.ReadLine();
32      }
33  }
34 }
35
```

通过分析不难看出，在这个例子中，在准备套餐的过程是稳定的，即按照一定的步骤去做，而套餐的组成部分则是变化的，有可能是普通套餐或黄金套餐等。这个变化就是建造者模式中的“变化点“，就是我们要封装的部分。

另外一个例子

在这里我们再给出另外一个关于建造房子的例子。客户程序通过调用指导者（CDirector class)的 BuildHouse()方法来创建一个房子。该方法有一个布尔型的参数 blnBackyard，当 blnBackyard 为假时指导者将创建一个 Apartment（Concrete Builder），当它为真时将创建一个 Single Family Home（Concrete Builder）。这两种房子都实现了接口 Ihouse。

程序实现：

```
1  //关于建造房屋的例子
2  using System;
3  using System.Collections;
4
5  /// <summary>
6  /// 抽象建造者
7  /// </summary>
8  public interface IHouse
9  {
10     bool GetBackyard();
11     long NoOfRooms();
12     string  Description();
13 }
14
15 /// <summary>
16 /// 具体建造者
17 /// </summary>
18  public class CApt:IHouse
19 {
20     private bool mblnBackyard;
21     private Hashtable Rooms;
```

```
22    public CApt()
23    {
24        CRoom room;
25        Rooms = new Hashtable();
26        room = new CRoom();
27        room.RoomName = "Master Bedroom";
28        Rooms.Add ("room1",room);
29
30        room = new CRoom();
31        room.RoomName = "Second Bedroom";
32        Rooms.Add ("room2",room);
33
34        room = new CRoom();
35        room.RoomName = "Living Room";
36        Rooms.Add ("room3",room);
37
38        mblnBackyard = false;
39    }
40
41    public bool GetBackyard()
42    {
43        return mblnBackyard;
```

```
44    }
45    public long NoOfRooms()
46    {
47        return Rooms.Count;
48    }
49    public string  Description()
50    {
51        IDictionaryEnumerator myEnumerator  = Rooms.GetEnumerator();
52        string strDescription;
53        strDescription = "This is an Apartment with " + Rooms.Count + " Room
s \n";
54        strDescription = strDescription + "This Apartment doesn't have a backy
ard \n";
55        while (myEnumerator.MoveNext())
56        {
57            strDescription = strDescription + "\n" + myEnumerator.Key + "\t
" + ((CRoom)myEnumerator.Value).RoomName;
58        }
59        return strDescription;
60    }
61 }
62
```

```
63 /// <summary>
64 /// 具体建造者
65 /// </summary>
66  public class CSFH:IHouse
67 {
68     private bool mblnBackyard;
69     private Hashtable Rooms;
70     public CSFH()
71     {
72         CRoom room;
73         Rooms = new Hashtable();
74 
75         room = new CRoom();
76         room.RoomName = "Master Bedroom";
77         Rooms.Add ("room1",room);
78 
79         room = new CRoom();
80         room.RoomName = "Second Bedroom";
81         Rooms.Add ("room2",room);
82 
83         room = new CRoom();
84         room.RoomName = "Third Room";
```

```
85          Rooms.Add ("room3",room);
86
87          room = new CRoom();
88          room.RoomName = "Living Room";
89          Rooms.Add ("room4",room);
90
91          room = new CRoom();
92          room.RoomName = "Guest Room";
93          Rooms.Add ("room5",room);
94
95          mblnBackyard = true;
96
97      }
98
99      public bool GetBackyard()
100     {
101         return mblnBackyard;
102     }
103     public long NoOfRooms()
104     {
105         return Rooms.Count;
106     }
```

```
107     public string  Description()
108     {
109         IDictionaryEnumerator myEnumerator  = Rooms.GetEnumerator();
110         string strDescription;
111         strDescription = "This is an Single Family Home with " + Rooms.Coun
t + " Rooms \n";
112         strDescription = strDescription + "This house has a backyard \n";
113         while (myEnumerator.MoveNext())
114         {
115             strDescription = strDescription + "\n" + myEnumerator.Key + "\t
" + ((CRoom)myEnumerator.Value).RoomName;
116         }
117         return strDescription;
118     }
119 }
120
121  public interface IRoom
122 {
123     string RoomName{get;set;}
124 }
125
126  public class CRoom:IRoom
```

```
127 {
128     private string mstrRoomName;
129     public string RoomName
130     {
131         get
132         {
133             return mstrRoomName;
134         }
135         set
136         {
137             mstrRoomName = value;
138         }
139     }
140 }
141
142 /// <summary>
143 /// 指导者
144 /// </summary>
145  public class CDirector
146 {
147     public IHouse BuildHouse(bool blnBackyard)
148     {
```

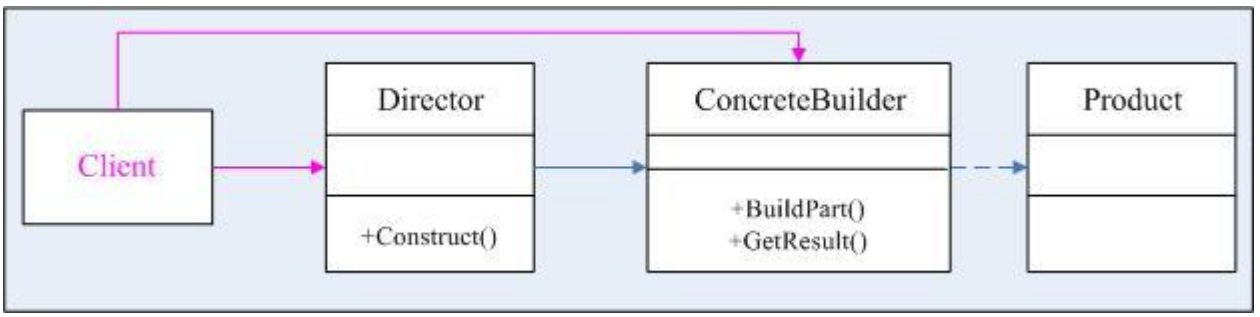
```
149         if (blnBackyard)
150         {
151             return new CSFH();
152         }
153         else
154         {
155             return new CApt();
156         }
157     }
158 }
159
160 /// <summary>
161 /// 客户程序
162 /// </summary>
163  public class Client
164 {
165     static void Main(string[] args)
166     {
167         CDirector objDirector = new CDirector();
168         IHouse objHouse;
169
170         string Input = Console.ReadLine();
```

```
171         objHouse = objDirector.BuildHouse(bool.Parse(Input));
172
173         Console.WriteLine(objHouse.Description());
174         Console.ReadLine();
175     }
176 }
177
178
```

建造者模式的几种演化

省略抽象建造者角色

系统中只需要一个具体建造者，省略掉抽象建造者，结构图如下：



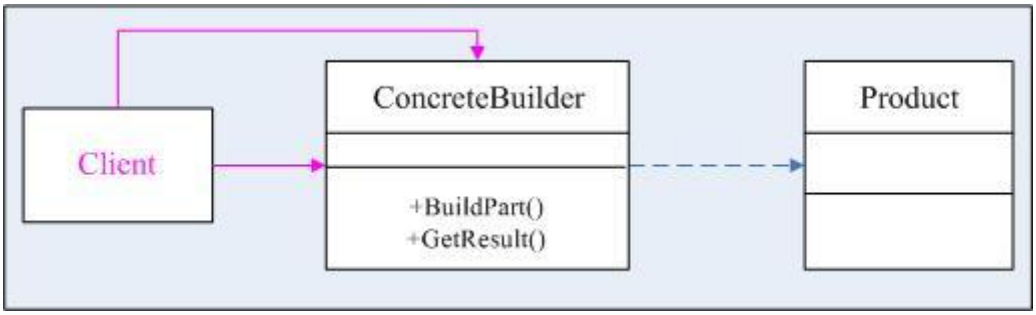
指导者代码如下：

```
1  class Director
2  {
3      private ConcreteBuilder builder;
4
5      public void Construct()
```

```
6    {
7        builder.BuildPartA();
8        builder.BuildPartB();
9    }
10 }
```

省略指导者角色

抽象建造者角色已经被省略掉，还可以省略掉指导者角色。让 Builder 角色自己扮演指导者与建造者双重角色。结构图如下：



建造者角色代码如下：

```
1  public class Builder
2  {
3      private Product product = new Product();
4
5      public void BuildPartA()
6      {
7          //……
8      }
```

```
9
10    public void BuildPartB()
11    {
12      //……
13    }
14
15    public Product GetResult()
16    {
17      return product;
18    }
19
20    public void Construct()
21    {
22      BuildPartA();
23      BuildPartB();
24    }
25  }
```

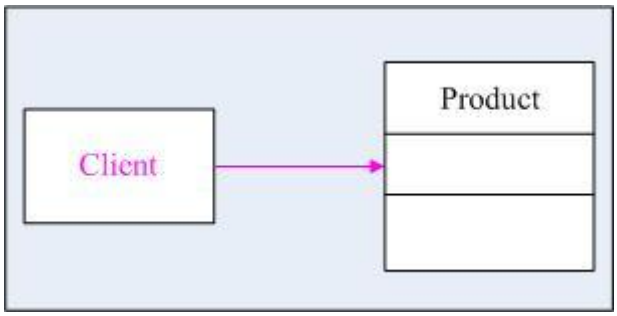
客户程序：

```
1   public class Client
2   {
3     private static Builder builder;
```

```
4
5    public static void Main()
6    {
7        builder = new Builder();
8        builder.Construct();
9        Product product = builder.GetResult();
10   }
11 }
```

合并建造者角色和产品角色

建造模式失去抽象建造者角色和指导者角色后，可以进一步退化，从而失去具体建造者角色，此时具体建造者角色和产品角色合并，从而使得产品自己就是自己的建造者。这样做混淆了对象的建造者和对象本身，但是有时候一个产品对象有着固定的几个零件，而且永远只有这几个零件，此时将产品类和建造类合并，可以使系统简单易读。结构图如下：



实现要点

1、建造者模式主要用于“分步骤构建一个复杂的对象”，在这其中“分步骤”是一个稳定的算法，而复杂对象的各个部分则经常变化。

2、产品不需要抽象类，特别是由于创建对象的算法复杂而导致使用此模式的情况下或者此模式应用于产品的生成过程，其最终结果可能差异很大，不大可能提炼出一个抽象产品类。

3、创建者中的创建子部件的接口方法不是抽象方法而是空方法，不进行任何操作，具体的创建者只需要覆盖需要的方法就可以，但是这也不是绝对的，特别是类似文本转换这种情况下，缺省的方法将输入原封不动的输出是合理的缺省操作。

4、前面我们说过的抽象工厂模式（Abtract Factory）解决“系列对象”的需求变化，Builder 模式解决“对象部分”的需求变化，建造者模式常和组合模式（Composite Pattern）结合使用。

效果

1、建造者模式的使用使得产品的内部表象可以独立的变化。使用建造者模式可以使客户端不必知道产品内部组成的细节。

2、每一个 Builder 都相对独立，而与其它的 Builder 无关。

3、可使对构造过程更加精细控制。

4、将构建代码和表示代码分开。

5、建造者模式的缺点在于难于应付“分步骤构建算法”的需求变动。

适用性

以下情况应当使用建造者模式：

1、需要生成的产品对象有复杂的内部结构。

2、需要生成的产品对象的属性相互依赖，建造者模式可以强迫生成顺序。

3、在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到。

应用场景

1、 RTF 文档交换格式阅读器。

2、 .NET 环境下的字符串处理 StringBuilder，这是一种简化了的建造者模式。

3、 ……

总结

建造者模式的实质是解耦组装过程和创建具体部件，使得我们不用去关心每个部件是如何组装的。

工厂方法模式（Factory Method）

——.NET 设计模式系列之五

Terrylee，2004 年 1 月 2 日

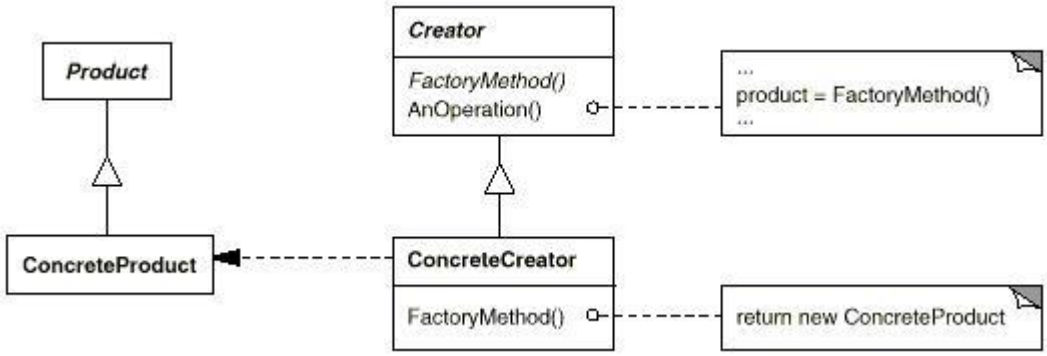
概述

在软件系统中，经常面临着“某个对象”的创建工作，由于需求的变化，这个对象的具体实现经常面临着剧烈的变化，但是它却拥有比较稳定的接口。如何应对这种变化？提供一种封装机制来隔离出“这个易变对象”的变化，从而保持系统中“其它依赖该对象的对象”不随着需求的改变而改变？这就是要说的 Factory Method 模式了。

意图

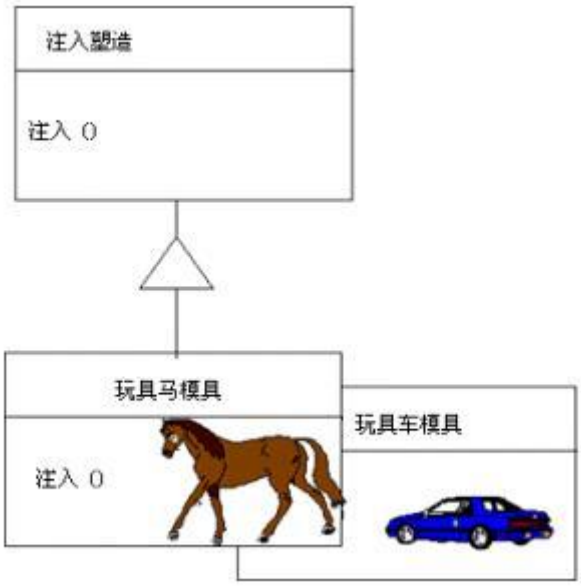
定义一个用户创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。

结构图



生活中的例子

工厂方法定义一个用于创建对象的接口，但是让子类决定实例化哪个类。压注成型演示了这种模式。塑料玩具制造商加工塑料粉，将塑料注入到希望形状的模具中。玩具的类别（车，人物等等）是由模具决定的。



工厂方法解说

在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不接触哪一个产品类被实例化这种细节。这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。在 Factory Method 模式中，工厂类与产品类往往具有平行的等级结构，它们之间一一对应。

现在我们考虑一个日志记录的例子（这里我们只是为了说明 Factory Method 模式，实际项目中的日志记录不会这么去做，也要比这复杂一些）。假定我们要设计日志记录的类，支持记

录的方法有 FileLog 和 EventLog 两种方式。在这里我们先不谈设计模式，那么这个日志记录的类就很好实现了：

```
/// <summary>
/// 日志记录类
/// </summary>
 public class Log
    {

        public void WriteEvent()
        {
            Console.WriteLine("EventLog Success!");
        }

        public void WriteFile()
        {
            Console.WriteLine("FileLog Success!");
        }

        public void Write(string LogType)
        {
            switch(LogType.ToLower())
            {
                case "event":
                    WriteEvent();
                    break;

                case "file":
                    WriteFile();
                    break;
```

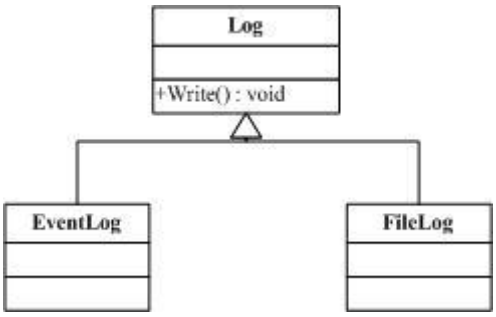
```
28
29              default:
30                  break;
31          }
32      }
33  }
34
```

这样的程序结构显然不能符合我们的要求，如果我们增加一种新的日志记录的方式 DataBaseLog，那就要修改 Log 类，随着记录方式的变化，switch 语句在不断的变化，这样就引起了整个应用程序的不稳定，进一步分析上面的代码，发现对于 EventLog 和 FileLog 是两种完全不同的记录方式，它们之间不应该存在必然的联系，而应该把它们分别作为单独的对象来对待。

```
1 /// <summary>
2 /// EventLog 类
3 /// </summary>
4  public class EventLog
5 {
6     public void Write()
7     {
8         Console.WriteLine("EventLog Write Success!");
9     }
10 }
11
12 /// <summary>
13 /// FileLog 类
14 /// </summary>
15  public class FileLog
16 {
17     public void Write()
18     {
```

```
19        Console.WriteLine("FileLog Write Success!");
20    }
21 }
22
```

进一步抽象，为它们抽象出一个共同的父类，结构图如下：



实现代码：

```
1 /// <summary>
2 /// Log 类
3 /// </summary>
4  public abstract class Log
5 {
6     public abstract void Write();
7 }
8
```

此时 EventLog 和 FileLog 类的代码应该如下：

```
1 /// <summary>
2 /// EventLog 类
3 /// </summary>
4  public class EventLog:Log
5 {
6     public override void Write()
7     {
8         Console.WriteLine("EventLog Write Success!");
```

```
9     }
10 }
11 /// <summary>
12 /// FileLog 类
13 /// </summary>
14  public class FileLog:Log
15 {
16     public override void Write()
17     {
18         Console.WriteLine("FileLog Write Success!");
19     }
20 }
21
```

此时我们再看增加新的记录日志方式 DatabaseLog 的时候，需要做哪些事情？只需要增加一个继承父类 Log 的子类来实现，而无需再去修改 EventLog 和 FileLog 类，这样的设计满足了类之间的层次关系，又很好的符合了面向对象设计中的单一职责原则，每一个类都只负责一件具体的事情。到这里似乎我们的设计很完美了，事实上我们还没有看客户程序如何去调用。在应用程序中，我们要使用某一种日志记录方式，也许会用到如下这样的语句：

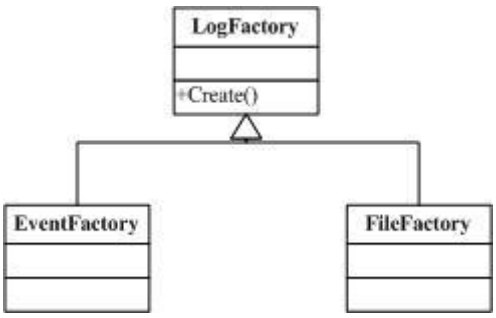
```
EventLog eventlog = new EventLog();
eventlog.Write();
```

当日志记录的方式从 EventLog 变化为 FileLog，我们就得修改所有程序代码中出现上面语句的部分，这样的工作量是可想而知的。此时就需要解耦具体的日志记录方式和应用程序。这就要引入 Factory Method 模式了，每一个日志记录的对象就是工厂所生成的产品，既然有两种记录方式，那就需要两个不同的工厂去生产了，代码如下：

```
1 /// <summary>
2 /// EventFactory 类
3 /// </summary>
4  public class EventFactory
```

```
{
    public EventLog Create()
    {
        return new EventLog();
    }
}
/// <summary>
/// FileFactory 类
/// </summary>
public class FileFactory
{
    public FileLog Create()
    {
        return new FileLog();
    }
}

```

这两个工厂和具体的产品之间是平行的结构，并一一对应，并在它们的基础上抽象出一个公用的接口，结构图如下：



实现代码如下：

```
/// <summary>
/// LogFactory 类
/// </summary>
```

```
4  public abstract class LogFactory
5 {
6      public abstract Log Create();
7 }
8
```

此时两个具体工厂的代码应该如下：

```
1 /// <summary>
2 /// EventFactory 类
3 /// </summary>
4  public class EventFactory:LogFactory
5 {
6     public override EventLog Create()
7     {
8         return new EventLog();
9     }
10 }
11 /// <summary>
12 /// FileFactory 类
13 /// </summary>
14  public class FileFactory:LogFactory
15 {
16     public override FileLog Create()
17     {
18         return new FileLog();
19     }
20 }
21
```

这样通过工厂方法模式我们把上面那对象创建工作封装在了工厂中，此时我们似乎完成了整个 Factory Method 的过程。这样达到了我们应用程序和具体日志记录对象之间解耦的目的了吗？看一下此时客户端程序代码：

```
/// <summary>
/// App 类
/// </summary>
 public class App
{
    public static void Main(string[] args)
    {
        LogFactory factory = new EventFactory();

        Log log = factory.Create();

        log.Write();
    }
}

```

在客户程序中，我们有效地避免了具体产品对象和应用程序之间的耦合，可是我们也看到，增加了具体工厂对象和应用程序之间的耦合。那这样究竟带来什么好处呢？我们知道，在应用程序中，Log 对象的创建是频繁的，在这里我们可以把

```
LogFactory factory = new EventFactory();
```

这句话放在一个类模块中，任何需要用到 Log 对象的地方仍然不变。要是换一种日志记录方式，只要修改一处为：

```
LogFactory factory = new FileFactory();
```

其余的任何地方我们都不需要去修改。有人会说那还是修改代码，其实在开发中我们很难避免修改，但是我们可以尽量做到只修改一处。

其实利用.NET 的特性，我们可以避免这种不必要的修改。下面我们利用.NET 中的反射机制来进一步修改我们的程序，这时就要用到配置文件了，如果我们想使用哪一种日志记录方式，则在相应的配置文件中设置如下：

```
<appSettings>
    <add key="factoryName" value="EventFactory"></add>
</appSettings>

```

此时客户端代码如下：

```
/// <summary>
/// App 类
/// </summary>
public class App
{
    public static void Main(string[] args)
    {
        string strfactoryName = ConfigurationSettings.AppSettings["factoryName"];

        LogFactory factory;
        factory = (LogFactory)Assembly.Load("FactoryMethod").CreateInstance("FactoryMethod." + strfactoryName);

        Log log = factory.Create();
        log.Write();
    }
}

```

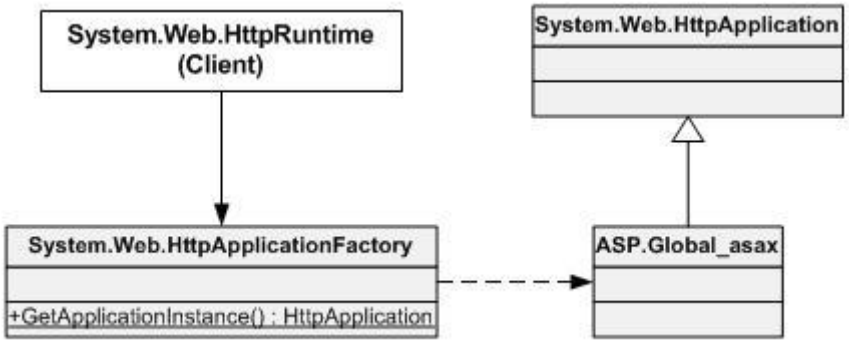
现在我们看到，在引进新产品（日志记录方式）的情况下，我们并不需要去修改工厂类，而只是增加新的产品类和新的工厂类（注意：这是任何时候都不能避免的），这样很好的符合了开放封闭原则。

ASP.NET HTTP 通道中的应用

Factory Method 模式在 ASP.NET HTTP 通道中我们可以找到很多的例子。ASP.NET HTTP 通道是 System.Web 命名空间下的一个类，WEB Server 使用该类处理接收到的 HTTP 请求，并给客户端发送响应。HTTP 通道主要的工作有 Session 管理，应用程序池管理，缓存管理，安全等。

System.Web.HttpApplicationFactory

HttpRuntime 是 HTTP 通道的入口点，它根据每一个具体的请求创建一个 HttpContext 实例， HttpRuntime 并没有确定它将要处理请求的 HttpApplication 对象的类型，它调用了一个静态的工厂方法 HttpApplicationFactory.GetApplicationInstance，通过它来创建 HttpContext 实例。GetApplicationInstance 使用 HttpContext 实例来确定针对这个请求该响应哪个虚拟路径，如果这个虚拟路径以前请求过，HttpApplication（或者一个继承于 ASP.Global_asax 的类的实例）将直接从应用程序池中返回，否则针对该虚拟路径将创建一个新的 HttpApplication 对象并返回。如下图所示：



HttpApplicationFactory.GetApplicationInstance 带有一个类型为 HttpContext 的参数，创建的所有对象（产品）都是 HttpApplication 的类型，通过反编译，来看一下 GetApplicationInstance 的实现：

```
1 internal static IHttpHandler GetApplicationInstance(HttpContext c
ontext)
2 {
```

```
3       if (HttpApplicationFactory._customApplication != null)
4       {
5           return HttpApplicationFactory._customApplication;
6       }
7       if (HttpDebugHandler.IsDebuggingRequest(context))
8       {
9           return new HttpDebugHandler();
10      }
11      if (!HttpApplicationFactory._theApplicationFactory._inited)
12      {
13          lock (HttpApplicationFactory._theApplicationFactory)
14          {
15              if (!HttpApplicationFactory._theApplicationFact
ory._inited)
16              {
17                  HttpApplicationFactory._theApplicationFac
tory.Init(context);
18                  HttpApplicationFactory._theApplicationFac
tory._inited = true;
19              }
20          }
21      }
22      return HttpApplicationFactory._theApplicationFactory.GetNor
malApplicationInstance(context);
23  }
24
```

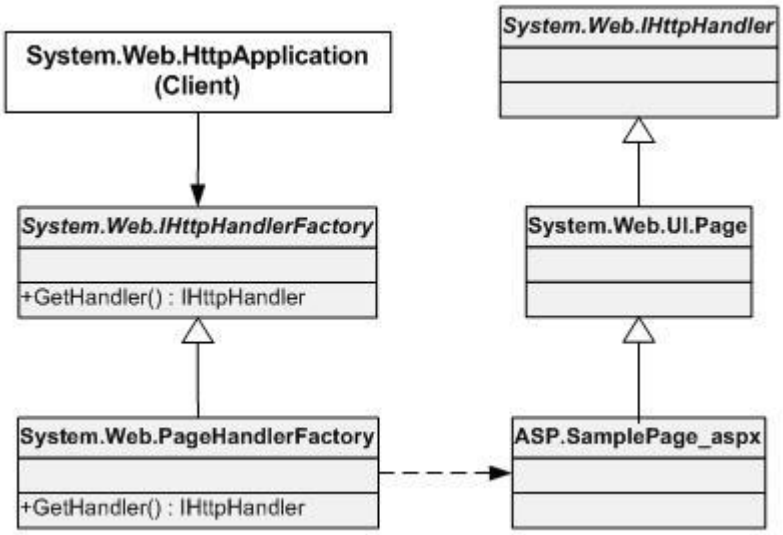
System.Web.IHttpHandlerFactory

我们来做进一步的探索，HttpApplication 实例需要一个 Handler 对象来处理资源请求， HttpApplication 的主要任务就是找到真正处理请求的类。HttpApplication 首先确

定了一个创建 Handler 对象的工厂，来看一下在 Machine.config 文件中的配置区<httphandlers>，在配置文件注册了应用程序的具体处理类。例如在 Machine.config 中对*.aspx 的处理将映射到 System.Web.UI.PageHandlerFactory 类，而对*.ashx 的处理将映射到 System.Web.UI.SimpleHandlerFactory 类，这两个类都是继承于 IhttpHandlerFactory 接口的具体类：

```
<httpHandlers>

<add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory" />

<add verb="*" path="*.ashx" type="System.Web.UI.SimpleHandlerFactory" />

……

</httpHandlers>
```

这个配置区建立了资源请求的类型和处理请求的类之间的一个映射集。如果一个.aspx 页面发出了请求，将会调用 System.Web.UI.PageHandlerFactory 类，HttpApplication 调用接口 IHttpHandlerFactory 中的工厂方法 GetHandler 来创建一个 Handler 对象。当一个名为 sample.aspx 的页面发出请求时，通过 PageHandlerFactory 将返回一个 ASP.SamplePage_aspx 对象（具体产品），如下图：



IHttpHandlerFactory 工厂：

```
1  public interface IHttpHandlerFactory
2 {
3         // Methods
4         IHttpHandler GetHandler(HttpContext context, string requestT
ype, string url, string pathTranslated);
5         void ReleaseHandler(IHttpHandler handler);
6 }
7
```

IHttpHandlerFactory.GetHandler是一个工厂方法模式的典型例子，在这个应用中，各个角色的设置如下：

抽象工厂角色：IHttpHandlerFactory

具体工厂角色：PageHandlerFactory

抽象产品角色：IHttpHandler

具体产品角色：ASP.SamplePage_aspx

进一步去理解

理解上面所说的之后，我们就可以去自定义工厂类来对特定的资源类型进行处理。第一步我们需要创建两个类去分别实现 IHttpHandlerFactory 和 IHttpHandler 这两个接口。

```
public class HttpHandlerFactoryImpl:IHttpHandlerFactory {

    IHttpHandler IHttpHandlerFactory.GetHandler(
        HttpContext context, String requestType,
          String url, String pathTranslated ) {

            return new HttpHandlerImpl();

    }//IHttpHandlerFactory.GetHandler

    void IHttpHandlerFactory.ReleaseHandler(
         IHttpHandler handler) { /*no-op*/ }

}//HttpHandlerFactoryImpl

public class HttpHandlerImpl:IHttpHandler {

     void IHttpHandler.ProcessRequest(HttpContext context) {

        context.Response.Write("sample handler invoked…");

    }//ProcessRequest

     bool IHttpHandler.IsReusable { get { return false; } }

}//HttpHandlerImpl

```

第二步需要在配置文件中建立资源请求类型和处理程序之间的映射。我们希望当请求的类型为*.sample 时进入我们自定义的处理程序，如下：

```
<httpHandlers>

    <add verb="*" path="*.sample"

        type="HttpHandlerFactoryImpl,SampleHandler" />

</httpHandlers>
```

最后一步我们需要把文件扩展*.sample 映射到 ASP.NET ISAPI 扩展 DLL（aspnet_isapi.dll）上。由于我们已经建立了用于处理新扩展文件的处理程序了，我们还需要把这个扩展名告诉 IIS 并把它映射到 ASP.NET。如果你不执行这个步骤而试图访问*.sample 文件，IIS 将简单地返回该文件而不是把它传递给 ASP.NET 运行时。其结果是该 HTTP 处理程序不会被调用。

运行 Internet 服务管理器，右键点击默认 Web 站点，选择属性，移动到主目录选项页，并点击配置按钮。应用程序配置对话框弹出来了。点击添加按钮并在可执行字段输入 aspnet_isapi.dll 文件路径，在扩展字段输入.sample。其它字段不用处理；该对话框如下所示：

在.NET Framework 中，关于工厂模式的使用有很多的例子，例如 IEnumerable 和 IEnumerator 就是一个 Creator 和一个 Product；System.Security.Cryptography 中关于加密

算法的选择，SymmetricAlgorithm，AsymmetricAlgorithm，和 HashAlgorithm 分别是三个工厂，他们各有一个静态的工厂方法 Create；System.Net.WebRequest 是 .NET Framework 的用于访问 Internet 数据的请求/响应模型的抽象基类。使用该请求/响应模型的应用程序可以用协议不可知的方式从 Internet 请求数据。在这种方式下，应用程序处理 WebRequest 类的实例，而协议特定的子类则执行请求的具体细节。请求从应用程序发送到某个特定的 URI，如服务器上的 Web 页。URI 从一个为应用程序注册的 WebRequest 子代列表中确定要创建的适当子类。注册 WebRequest 子代通常是为了处理某个特定的协议（如 HTTP 或 FTP），但是也可以注册它以处理对特定服务器或服务器上的路径的请求。有时间我会就.NET Framework 中工厂模式的使用作一个专题总结。

实现要点

1. Factory Method 模式的两种情况：一是 Creator 类是一个抽象类且它不提供它所声明的工厂方法的实现；二是 Creator 是一个具体的类且它提供一个工厂方法的缺省实现。
2. 工厂方法是可以带参数的。
3. 工厂的作用并不仅仅只是创建一个对象，它还可以做对象的初始化，参数的设置等。

效果

1. 用工厂方法在一个类的内部创建对象通常比直接创建对象更灵活。
2. Factory Method 模式通过面向对象的手法，将所要创建的具体对象的创建工作延迟到了子类，从而提供了一种扩展的策略，较好的解决了这种紧耦合的关系。

适用性

在以下情况下，适用于工厂方法模式：

1. 当一个类不知道它所必须创建的对象的类的时候。
2. 当一个类希望由它的子类来指定它所创建的对象的时候。
3. 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

总结

Factory Method 模式是设计模式中应用最为广泛的模式，通过本文，相信读者已经对它有了一定的认识。然而我们要明确的是：在面向对象的编程中，对象的创建工作非常简单，对象的创建时机却很重要。Factory Method 要解决的就是对象的创建时机问题，它提供了一种扩展的策略，很好地符合了开放封闭原则。_____

_____

原型模式（Prototype Pattern）

——.NET 设计模式系列之六

Terrylee，2006 年 1 月

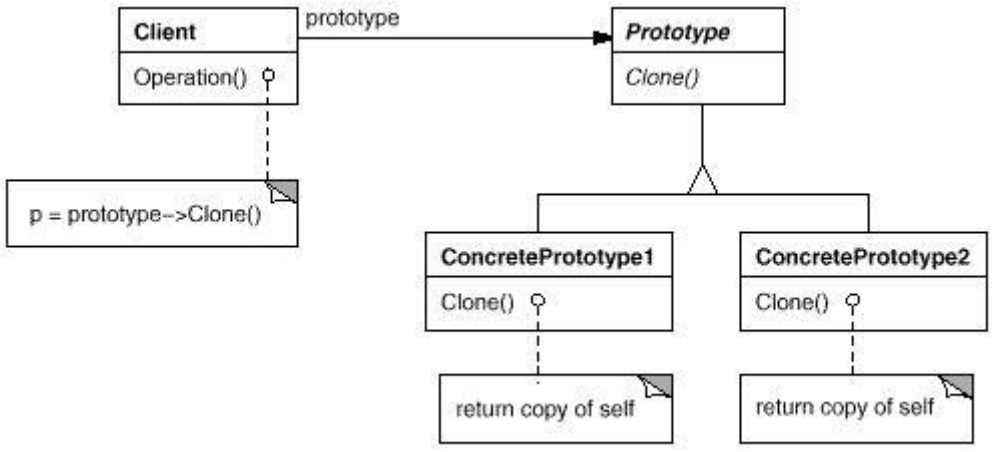
概述

在软件系统中，有时候面临的产品类是动态变化的，而且这个产品类具有一定的等级结构。这时如果用工厂模式，则与产品类等级结构平行的工厂方法类也要随着这种变化而变化，显然不大合适。那么如何封装这种动态的变化？从而使依赖于这些易变对象的客户程序不随着产品类变化？

意图

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

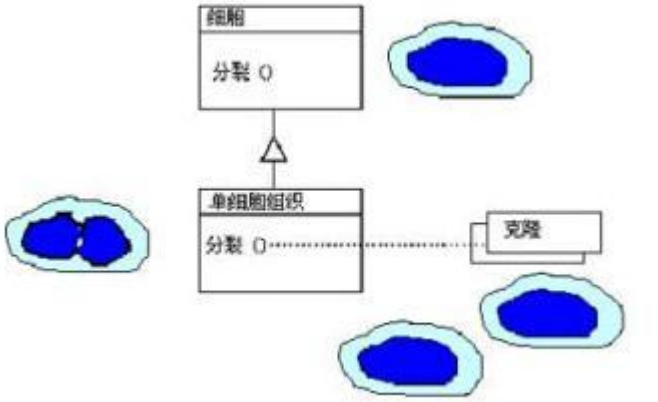
结构图



Prototype 模式结构图

生活中的例子

Prototype 模式使用原型实例指定创建对象的种类。新产品的原型通常是先于全部产品建立的，这样的原型是被动的，并不参与复制它自己。一个细胞的有丝分裂，产生两个同样的细胞，是一个扮演主动角色复制自己原型的例子，这演示了原型模式。一个细胞分裂，产生两个同样基因型的细胞。换句话说，细胞克隆了自己。

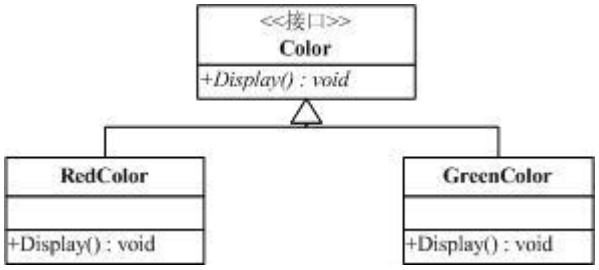


使用细胞分裂例子的 Prototype 模式对象图

原型模式解说

我们考虑这样一个场景，假定我们要开发一个调色板，用户单击调色板上任一个方块，将会返回一个对应的颜色的实例，下面我们看看如何通过原型模式来达到系统动态加载具体产品的目的。

很自然，我们利用 OO 的思想，把每一种颜色作为一个对象，并为他们抽象出一个公用的父类，如下图：

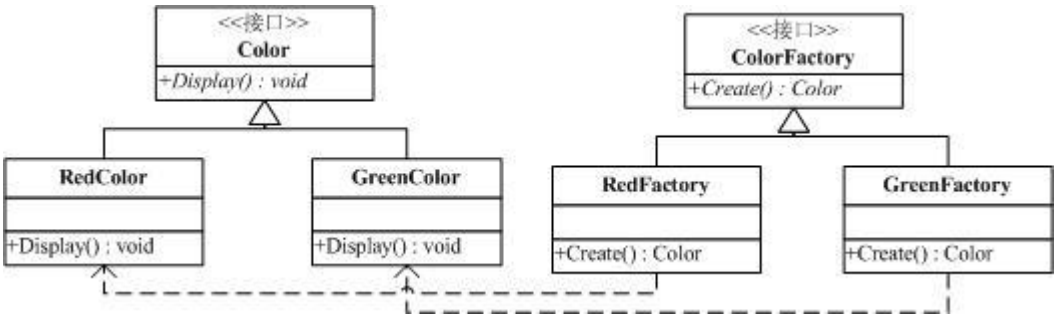


实现代码：

```
public abstract class Color

{
```

```
        public abstract void Display();
}
public class RedColor:Color
{
    public override void Display()
    {
        Console.WriteLine("Red's RGB Values are:255,0,0");
    }
}
public class GreenColor:Color
{
    public override void Display()
    {
        Console.WriteLine("Green's RGB Values are:0,255,0");
    }
}
```

客户程序需要某一种颜色的时候，只需要创建对应的具体类的实例就可以了。但是这样我们并没有达到封装变化点的目的，也许你会说，可以使用工厂方法模式，为每一个具体子类定义一个与其等级平行的工厂类，那么好，看一下实现：



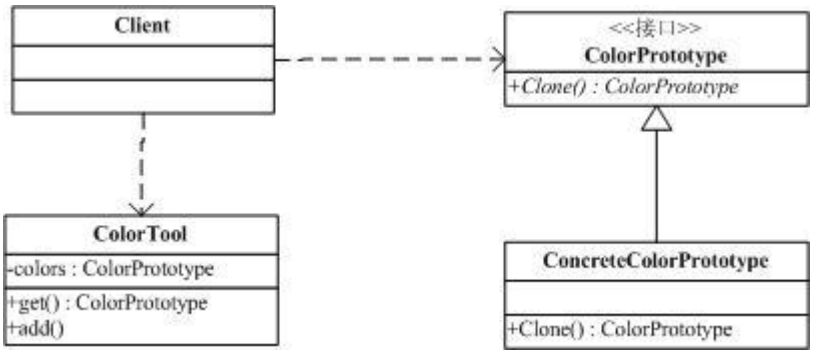
实现代码：

```
public abstract class ColorFactory
{
    public abstract Color Create();
}
public class RedFactory:ColorFactory
{
    public override Color Create()
    {
        return new RedColor();
    }
}
public class GreenFactory:ColorFactory
{
    public override Color Create()
    {
        return new GreenColor();
```

```
    }
}
```

实现了这一步之后，可以看到，客户程序只要调用工厂方法就可以了。似乎我们用工厂方法模式来解决是没有问题的。但是，我们考虑的仅仅是封装了 new 变化，而没有考虑颜色的数量是不断变化的，甚至可能是在程序运行的过程中动态增加和减少的，那么用这种方法实现，随着颜色数量的不断增加，子类的数量会迅速膨大，导致子类过多，显然用工厂方法模式有些不大合适。

进一步思考，这些 Color 子类仅仅在初始化的颜色对象类别上有所不同。添加一个 ColorTool 这样的类，来参数化的它的实例，而这些实例是由 Color 支持和创建的。我们让 ColorTool 通过克隆或者拷贝一个 Color 子类的实例来创建新的 Color，这个实例就是一个原型。如下图所示：



实现代码：

```
abstract class ColorPrototype
{
    public abstract ColorPrototype Clone();
}
class ConcteteColorPrototype : ColorPrototype
{
    private int _red, _green, _blue;
```

```
    public ConcteteColorPrototype(int red, int green, int blue)
    {
        this._red = red;
        this._green = green;
        this._blue = blue;
    }
    public override ColorPrototype Clone()
    {
        //实现浅拷贝
        return (ColorPrototype) this.MemberwiseClone();
    }
    public void Display(string _colorname)
    {
        Console.WriteLine("{0}'s RGB Values are: {1},{2},{3}",
            _colorname,_red, _green, _blue );
    }
}
class ColorManager
{
    Hashtable colors = new Hashtable();

    public ColorPrototype this[string name]
```

```
        {
            get
            {
                return (ColorPrototype)colors[name];
            }
            set
            {
                colors.Add(name,value);
            }
        }
    }
```

现在我们分析一下，这样带来了什么好处？首先从子类的数目上大大减少了，不需要再为每一种具体的颜色产品而定一个类和与它等级平行的工厂方法类，而 ColorTool 则扮演了原型管理器的角色。再看一下为客户程序的实现：

```
class App
{
    public static void Main(string[] args)
    {
        ColorManager colormanager = new ColorManager();
        //初始化颜色
        colormanager["red"] = new ConcteteColorPrototype(255, 0, 0);
        colormanager["green"] = new ConcteteColorPrototype(0, 255, 0);
```

```
            colormanager["blue"] = new ConcteteColorPrototype(0, 0, 255);
            colormanager["angry"] = new ConcteteColorPrototype(255, 54, 0);
            colormanager["peace"] = new ConcteteColorPrototype(128, 211, 128);
            colormanager["flame"] = new ConcteteColorPrototype(211, 34, 20);
            //使用颜色
            string colorName = "red";
            ConcteteColorPrototype c1 = (ConcteteColorPrototype)colormanager[color
Name].Clone();
            c1.Display(colorName);
            colorName = "peace";
            ConcteteColorPrototype c2 = (ConcteteColorPrototype)colormanager[color
Name].Clone();
            c2.Display(colorName);
            colorName = "flame";
            ConcteteColorPrototype c3 = (ConcteteColorPrototype)colormanager[color
Name].Clone();
            c3.Display(colorName);
            Console.ReadLine();
        }
    }
```

可以看到，客户程序通过注册原型实例就可以将一个具体产品类并入到系统中，在运行时刻，可以动态的建立和删除原型。最后还要注意一点，在上面的例子中，用的是浅表复制。如果想做深复制，需要通过序列化的方式来实现。经过了上面的分析之后，我们再来思考下面的问题：

1．为什么需要 Prototype 模式？

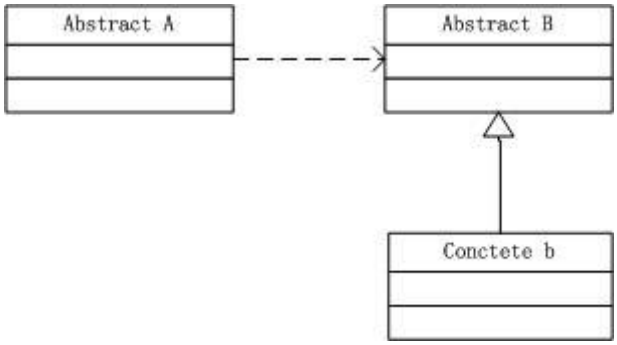
引入原型模式的本质在于利用已有的一个原型对象，快速的生成和原型对象一样的实例。你有一个 A 的实例 a:A a = new A();现在你想生成和 car1 一样的一个实例 b，按照原型模式，应该是这样：A b = a.Clone();而不是重新再 new 一个 A 对象。通过上面这句话就可以得到一个和 a 一样的实例，确切的说，应该是它们的数据成员是一样的。Prototype 模式同样是返回了一个 A 对象而没有使用 new 操作。

2．引入 Prototype 模式带来了什么好处？

可以看到，引入 Prototype 模式后我们不再需要一个与具体产品等级结构平行的工厂方法类，减少了类的构造，同时客户程序可以在运行时刻建立和删除原型。

3．Prototype 模式满足了哪些面向对象的设计原则？

依赖倒置原则：上面的例子，原型管理器（ColorManager）仅仅依赖于抽象部分（ColorPrototype），而具体实现细节（ConcteteColorPrototype）则依赖与抽象部分（ColorPrototype），所以 Prototype 很好的满足了依赖倒置原则。



通过序列化实现深拷贝

要实现深拷贝，可以通过序列化的方式。抽象类及具体类都必须标注为可序列化的[Serializable]，上面的例子加上深拷贝之后的完整程序如下：

```
using System;

using System.Collections;

using System.IO;
```

```
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
abstract class ColorPrototype
{
    public abstract ColorPrototype Clone(bool Deep);
}
[Serializable]
class ConcteteColorPrototype : ColorPrototype
{
    private int _red, _green, _blue;
    public ConcteteColorPrototype(int red, int green, int blue)
    {
        this._red = red;
        this._green = green;
        this._blue = blue;
    }
    public override ColorPrototype Clone(bool Deep)
    {
        if(Deep)
            return CreateDeepCopy();
        else
```

```
        return (ColorPrototype) this.MemberwiseClone();
}

//实现深拷贝
public ColorPrototype CreateDeepCopy()
{
    ColorPrototype colorPrototype;
    MemoryStream memoryStream = new MemoryStream();
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(memoryStream, this);
    memoryStream.Position = 0;

    colorPrototype = (ColorPrototype)formatter.Deserialize(memoryStream);
    return colorPrototype;
}

public ConcteteColorPrototype Create(int red,int green,int blue)
{
    return new ConcteteColorPrototype(red,green,blue);
}
public void Display(string _colorname)
{
```

```
        Console.WriteLine("{0}'s RGB Values are: {1},{2},{3}",
            _colorname,_red, _green, _blue );
    }
}
class ColorManager
{
    Hashtable colors = new Hashtable();
    public ColorPrototype this[string name]
    {
        get
        {
            return (ColorPrototype)colors[name];
        }
        set
        {
            colors.Add(name,value);
        }
    }
}
class App
{
    public static void Main(string[] args)
```

```
        {
            ColorManager colormanager = new ColorManager();
            //初始化颜色
            colormanager["red"] = new ConcteteColorPrototype(255, 0, 0);
            colormanager["green"] = new ConcteteColorPrototype(0, 255, 0);
            colormanager["blue"] = new ConcteteColorPrototype(0, 0, 255);
            colormanager["angry"] = new ConcteteColorPrototype(255, 54, 0);
            colormanager["peace"] = new ConcteteColorPrototype(128, 211, 128);
            colormanager["flame"] = new ConcteteColorPrototype(211, 34, 20);
            //使用颜色
            string colorName = "red";
            ConcteteColorPrototype c1 = (ConcteteColorPrototype)colormanager[color
Name].Clone(false);
            c1.Display(colorName);
            colorName = "peace";
            ConcteteColorPrototype c2 = (ConcteteColorPrototype)colormanager[color
Name].Clone(true);
            c2.Display(colorName);
            colorName = "flame";
            ConcteteColorPrototype c3 = (ConcteteColorPrototype)colormanager[color
Name].Clone(true);
            c3.Display(colorName);
            Console.ReadLine();
```

```
    }
}
```

实现要点

1．使用原型管理器，体现在一个系统中原型数目不固定时，可以动态的创建和销毁，如上面的举的调色板的例子。

2．实现克隆操作，在.NET 中可以使用 Object 类的 MemberwiseClone()方法来实现对象的浅表拷贝或通过序列化的方式来实现深拷贝。

3．Prototype 模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些“易变类”拥有稳定的接口。

效果

1．它对客户隐藏了具体的产品类，因此减少了客户知道的名字的数目。

2．Prototype 模式允许客户只通过注册原型实例就可以将一个具体产品类并入到系统中，客户可以在运行时刻建立和删除原型。

3．减少了子类构造，Prototype 模式是克隆一个原型而不是请求工厂方法创建一个，所以它不需要一个与具体产品类平行的 Creater 类层次。

4．Portotype 模式具有给一个应用软件动态加载新功能的能力。由于 Prototype 的独立性较高，可以很容易动态加载新功能而不影响老系统。

5．产品类不需要非得有任何事先确定的等级结构，因为 Prototype 模式适用于任何的等级结构

6．Prototype 模式的最主要缺点就是每一个类必须配备一个克隆方法。而且这个克隆方法需要对类的功能进行通盘考虑，这对全新的类来说不是很难，但对已有的类进行改造时，不一定是件容易的事。

适用性

在下列情况下，应当使用 Prototype 模式：

1．当一个系统应该独立于它的产品创建，构成和表示时；

2．当要实例化的类是在运行时刻指定时，例如，通过动态装载；

3．为了避免创建一个与产品类层次平行的工厂类层次时；

4．当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

总结

Prototype 模式同工厂模式，同样对客户隐藏了对象的创建工作，但是，与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的，达到了“隔离类对象的使用者和具体类型（易变类）之间的耦合关系”的目的。

创建型模式专题总结（Creational Pattern）

——.NET 设计模式系列之七

Terrylee，2006 年 1 月

概述

创建型模式，就是用来创建对象的模式，抽象了实例化的过程。它帮助一个系统独立于如何创建、组合和表示它的那些对象。本文对五种常用创建型模式进行了比较，通过一个游戏开发场景的例子来说该如何使用创建型模式。

为什么需要创建型模式

所有的创建型模式都有两个永恒的主旋律：第一，它们都将系统使用哪些具体类的信息封装起来；第二，它们隐藏了这些类的实例是如何被创建和组织的。外界对于这些对象只知道它们共同的接口，而不清楚其具体的实现细节。正因如此，创建型模式在创建什么（what），由谁（who）来创建，以及何时（when）创建这些方面，都为软件设计者提供了尽可能大的灵活性。

假定在一个游戏开发场景中，会用到一个现代风格房屋的对象，按照我们的一般想法，既然需要对象就创建一个：

```
ModernRoom room = new ModernRoom();
```

好了，现在现代风格房屋的对象已经有了，如果这时房屋的风格变化了，需要的是古典风格的房屋，修改一下：

```
ClassicalRoom room = new ClassicalRoom();
```

试想一下，在我们的程序中有多少处地方用到了这样的创建逻辑，而这里仅仅是房屋的风格变化了，就需要修改程序中所有的这样的语句。现在我们封装对象创建的逻辑，把对象的创建放在一个工厂方法中：

```
ModernFactory factory = new ModernFactory();
ModernRoom room = factory.Create();
```

当房屋的风格变化时，只需要修改

```
ClassicalFactory factory = new ClassicalFactory();
ClassicalRoom room = factory.Create();
```

而其它的用到 room 的地方仍然不变。这就是为什么需要创建型模式了。创建者模式作用可以概括为如下两点：

1. 封装创建逻辑，绝不仅仅是 new 一个对象那么简单。

2. 封装创建逻辑变化，客户代码尽量不修改，或尽量少修改。

常见的五种创建型模式

单件模式（Singleton Pattern）解决的是实体对象的个数问题，其他的都是解决 new 所带来的耦合关系问题。

工厂方法模式（Factory Pattern）在工厂方法中，工厂类成为了抽象类，其实际的创建工作将由其具体子类来完成。工厂方法的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中去，强调的是“单个对象”的变化。

抽象工厂模式（Abstract Factory）抽象工厂是所有工厂模式中最为抽象和最具有一般性的一种形态。抽象工厂可以向客户提供一个接口，使得客户可以在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象，强调的是“系列对象”的变化。

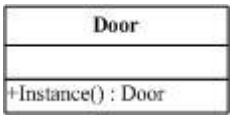
生成器模式（Builder Pattern）把构造对象实例的逻辑移到了类的外部，在这个类的外部定义了这个类的构造逻辑。他把一个复杂对象的构造过程从对象的表示中分离出来。其直接效果是将一个复杂的对象简化为一个比较简单的目标对象。他强调的是产品的构造过程。

原型模式（Prototype Pattern）和工厂模式一样，同样对客户隐藏了对象创建工作，但是，与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。

如何选择使用创建型模式

继续考虑上面提到的游戏开发场景，假定在这个游戏场景中我们使用到的有墙（Wall），屋子（Room），门（Door）几个部件。在这个过程中，同样是对象的创建问题，但是会根据所要解决的问题不同而使用不同的创建型模式。

如果在游戏中，一个屋子只允许有一个门存在，那么这就是一个使用 Signleton 模式的例子，确保只有一个 Door 类的实例被创建。解决的是对象创建个数的问题。

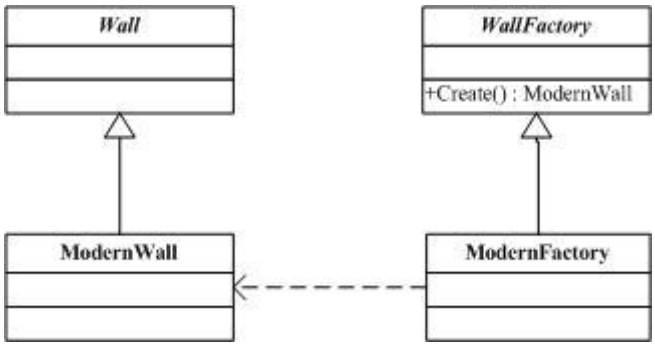


示例代码：

```
using System;
public sealed class SigletonDoor
{
    static readonly SigletonDoor instance=new SigletonDoor();
    static SigletonDoor()
    {
    }
    public static SigletonDoor Instance
    {
        get
        {
            return instance;
```

```
        }
    }
}
```

在游戏中需要创建墙，屋子的实例时，为了避免直接对构造器的调用而实例化类，这时就是工厂方法模式了，每一个部件都有它自己的工厂类。解决的是“单个对象”的需求变化问题。

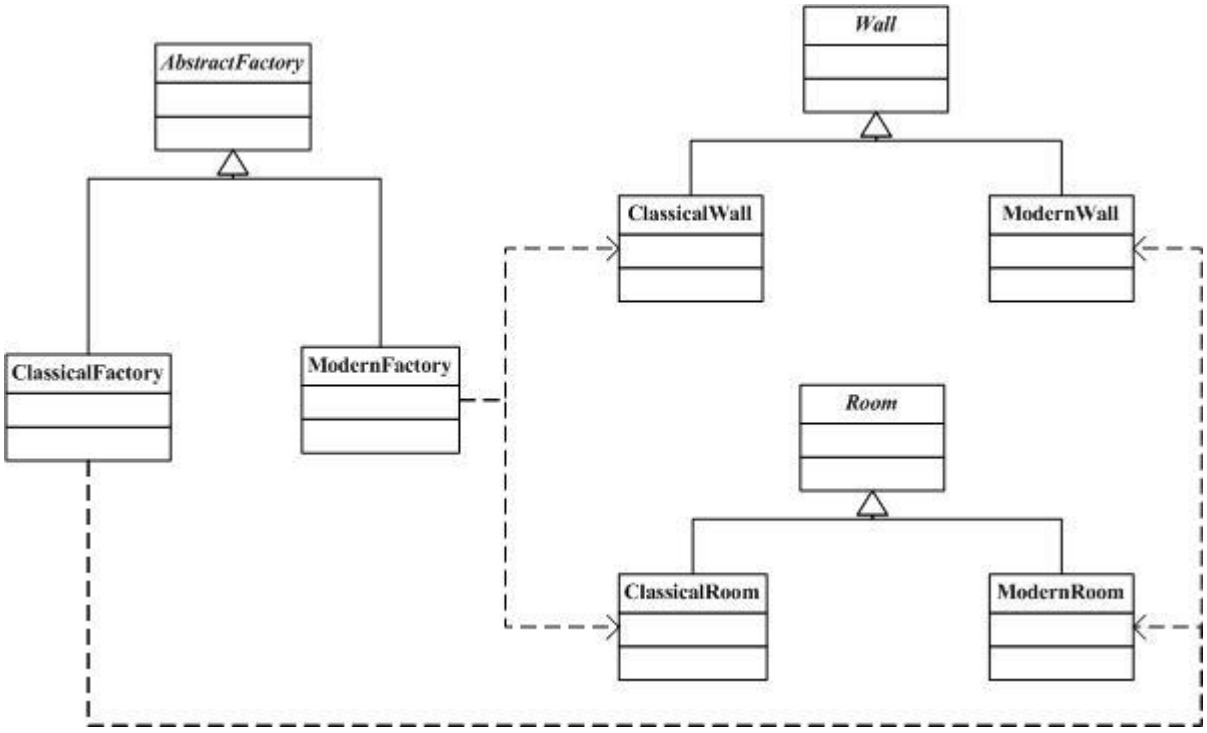


示例代码：

```
using System;
public abstract class Wall
{
    public abstract void Display();
}
public class ModernWall:Wall
{
    public override void Display()
    {
        Console.WriteLine("ModernWall Builded");
    }
}
```

```
public abstract class WallFactory
{
    public abstract Wall Create();
}
public class ModernFactory:WallFactory
{
    public override Wall Create()
    {
        return new ModernWall();;
    }
}
```

在游戏场景中，不可能只有一种墙或屋子，有可能有现代风格（Modern），古典风格（Classical）等多系列风格的部件。这时就是一系列对象的创建问题了，是一个抽象工厂的例子。解决的是“系列对象”的需求变化问题。



示例代码：

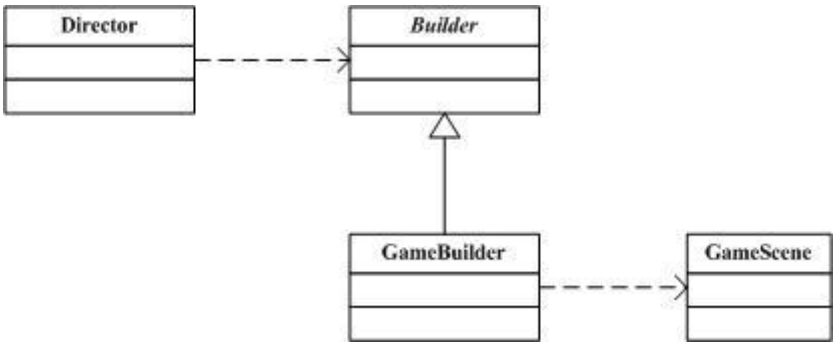
```
using System;


public abstract class Wall
{
    public abstract void Display();
}


public class ModernWall:Wall
{
    public override void Display()
    {
        Console.WriteLine("ModernWall Builded");
```

```
    }
}

public class ClassicalWall:Wall
{
    public override void Display()
    {
        Console.WriteLine("ClassicalWall Builded");
    }
}

public abstract class Room
{
    public abstract void Display();
}

public class ModernRoom:Room
{
    public override void Display()
    {
        Console.WriteLine("ModernRoom Builded");
    }
```

```
}

public class ClassicalRoom:Room
{
    public override void Display()
    {
        Console.WriteLine("ClassicalRoom Builded");
    }
}

public abstract class AbstractFactory
{
    public abstract Wall CreateWall();
    public abstract Room CreateRoom();
}

public class ModernFactory:AbstractFactory
{
    public override Wall CreateWall()
    {
        return new ModernWall();
    }
```

```
    public override Room CreateRoom()
    {
        return new ModernRoom();
    }
}

public class ClassicalFactory:AbstractFactory
{
    public override Wall CreateWall()
    {
        return new ClassicalWall();
    }
    public override Room CreateRoom()
    {
        return new ClassicalRoom();
    }
}
```

如果在游戏场景中，构成某一个场景的算法比较稳定，例如：这个场景就是用四堵墙，一个屋子，一扇门来构成的，但具体是用什么风格的墙、屋子和门则是不停的变化的，这就是一个生成器模式的例子。解决的是“对象部分”的需求变化问题。



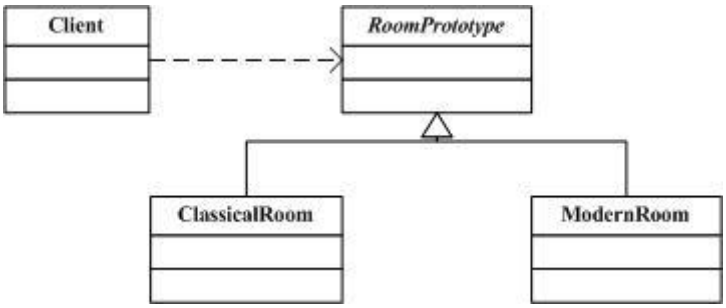
示例代码：

```
using System;
using System.Collections;


public class Director
{
    public void Construct( Builder builder )
    {
        builder.BuildWall();
        builder.BuildRoom();
        builder.BuildDoor();
    }
}


public abstract class Builder
{
    public abstract void BuildWall();
```

```
    public abstract void BuildRoom();
    public abstract void BuildDoor();
    public abstract GameScene GetResult();
}

public class GameBuilder : Builder
{
    private GameScene g;

    public override void BuildWall()
    {
        g = new GameScene();
        g.Add( "Wall" );
    }
    public override void BuildRoom()
    {
        g.Add( "Room" );
    }
    public override void BuildDoor()
    {
        g.Add( "Door" );
    }
```

```
    public override GameScene GetResult()
    {
        return g;
    }
}

public class GameScene
{
    ArrayList parts = new ArrayList();
    public void Add( string part )
    {
        parts.Add( part );
    }
    public void Display()
    {
        Console.WriteLine( " GameScene Parts: " );
        foreach( string part in parts )
            Console.WriteLine( part );
    }
}
```

如果在游戏中，需要大量的古典风格或现代风格的墙或屋子，这时可以通过拷贝一个已有的原型对象来生成新对象，就是一个原型模式的例子了。通过克隆来解决“易变对象”的创建问题。



示例代码：

```
using System;


public abstract class RoomPrototype
{
    public abstract RoomPrototype Clone();
}


public class ModernPrototype:RoomPrototype
{
    public override RoomPrototype Clone()
    {
        return (RoomPrototype)this.MemberwiseClone();
    }
}


public class ClassicalPrototype:RoomPrototype
{
```

```
        public override RoomPrototype Clone()
        {
            return (RoomPrototype)this.MemberwiseClone();
        }
    }
```

究竟选用哪一种模式最好取决于很多的因素。使用 Abstract Factory、Prototype Pattern 或 Builder Pattern 的设计比使用 Factory Method 的设计更加灵活，但是也更加复杂，尤其 Abstract Factory 需要庞大的工厂类来支持。通常，设计以使用 Factory Method 开始，并且当设计者发现需要更大的灵活性时，设计便会向其他设计模式演化，当你在多个设计模式之间进行权衡的时候，了解多个设计模式可以给你提供更多的选择余地。

总结

使用创建者模式是为了提高系统的可维护性和可扩展性，提高应对需求变化的能力！