...

[核心概念 2](#_Toc482695676)

[第1章 asp.net 简介 2](#_Toc482695677)

[第2章 Visual Studio 3](#_Toc482695678)

[第3章 Web窗体 5](#_Toc482695679)

[第4章服务器控件 9](#_Toc482695680)

[第5章 Asp.Net应用程序 14](#_Toc482695681)

[第6章状态管理 18](#_Toc482695682)

[数据访问 23](#_Toc482695683)

[第7章 Ado.net基础 23](#_Toc482695684)

[第8章数据组件和DataSet 27](#_Toc482695685)

[第9章数据绑定 31](#_Toc482695686)

[第10章富数据控件 37](#_Toc482695687)

[第11章缓存和异步界面 43](#_Toc482695688)

[第12章文件和流 49](#_Toc482695689)

[第13章 Linq 54](#_Toc482695690)

[第14章 XML 58](#_Toc482695691)

[构建Asp.net网站 63](#_Toc482695692)

[第15章用户控件 63](#_Toc482695693)

[第16章主题和母版页 65](#_Toc482695694)

[第17章网站导航 69](#_Toc482695695)

[第18章网站部署 76](#_Toc482695696)

[安全 82](#_Toc482695697)

[第19章 Asp.Net安全模型 82](#_Toc482695698)

[第20章表单验证 84](#_Toc482695699)

[第21章成员资格 84](#_Toc482695700)

[第22章 Windows 验证 86](#_Toc482695701)

[第23章授权和角色 87](#_Toc482695702)

[第24章用户配置 88](#_Toc482695703)

[第25章加密 89](#_Toc482695704)

[第26章自定义成员资格提供程序 91](#_Toc482695705)

[高级用户界面 91](#_Toc482695706)

[第27章自定义服务器控件 91](#_Toc482695707)

[第28章设计时支持 100](#_Toc482695708)

[第29章动态图形和GDI+ 104](#_Toc482695709)

[第30章使用Web部件页面的门户 105](#_Toc482695710)

[客户端程序设计 106](#_Toc482695711)

[第31章 JavaScript 和 Ajax 技术 106](#_Toc482695712)

[第32章 Asp.Net Ajax 108](#_Toc482695713)

[第33章 SilverLight 109](#_Toc482695714)

[技术积累 111](#_Toc482695715)

[第34章正则表达式 111](#_Toc482695716)

[.NET扩展知识学习 112](#_Toc482695717)

[35 反射学习 112](#_Toc482695718)

[36 WCF 113](#_Toc482695719)

[37 Silverlight 113](#_Toc482695720)

[38 Linq 113](#_Toc482695721)

[第三方框架学习 113](#_Toc482695722)

[39 AOP 113](#_Toc482695723)

[40 IOC 113](#_Toc482695724)

[41 Spring.Net 113](#_Toc482695725)

[42 Nhibernate 113](#_Toc482695726)

[43 ASP.NET MVC 113](#_Toc482695727)

[44 Castle 113](#_Toc482695728)

[45 SOA 113](#_Toc482695729)

[C#扩展知识学习 113](#_Toc482695730)

[46委托与事件 113](#_Toc482695731)

[47 异步编程 120](#_Toc482695732)

[48 C# DateTime格式化 121](#_Toc482695733)

# C#扩展知识学习

### 46 委托与事件

##### 46.1委托

###### 46.1.1定义:

委托是一个类，它定义了方法的类型，使得可以将方法当作另一个方法的参数来进行传递，这种将方法动态地赋给参数的做法，可以避免在程序中大量使用If-Else(Switch)语句，同时使得程序具有更好的可扩展性。

###### 46.1.2使用场景:

将方法作为参数传递，委托类型，委托是一个类

###### 46.1.3定义代码:

public delegate void GreetingDelegate(string name);

public static void EnglishGreeting(string name)

{

Console.WriteLine("Morning,{0}", name);

}

public static void ChineseGreeting(string name)

{

Console.WriteLine("早上好,{0}", name);

}

public static void GreetPeople(string name, GreetingDelegate makingGreet)

{

makingGreet(name);

}

static void Main(string[] args)

{

GreetPeople("Jimmy Zhang", EnglishGreeting);

GreetPeople("张子阳", ChineseGreeting);

Console.ReadLine();

}

###### 46.1.4备注:

1. 委托是一个类，使用之前必须实例化或者是进行赋值
2. 注册到委托的方法，每一次执行委托的时候，其注册过且未注销的方法都会执行一遍，注销过的不再执行
3. 使用委托可以将多个方法绑定到同一个委托变量，当调用此变量时(这里用“调用”这个词，是因为此变量代表一个方法)，可以依次调用所有绑定的方法。

##### 46.2事件

###### 46.2.1定义:

* 1. Event封装了委托类型的变量，使得：在类的内部，不管你声明它是public还是protected，它总是private的。在类的外部，注册“+=”和注销“-=”的访问限定符与你在声明事件时使用的访问符相同。
  2. 声明一个事件不过类似于声明一个进行了封装的委托类型的变量而已

###### 46.2.2使用场景:

为什么在类型中使用事件向外部提供方法注册，而不是直接使用委托变量的原因

* + 1. 封装性和易用性
    2. 事件应该由事件发布者触发，而不应该由客户端（客户程序）来触发

###### 46.2.3定义代码:

public class GreetingManager

{  
//这一次我们在这里声明一个事件  
public event GreetingDelegate MakeGreet;  
  
 public void GreetPeople(string name)

{  
        MakeGreet(name);  
    }

}

###### 46.2.4备注:

对事件的声明实际是声明一个私有的委托变量

##### 46.3Observer设计模式

###### 46.3.1定义:

Observer设计模式是为了定义对象间的一种一对多的依赖关系，以便于当一个对象的状态改变时，其他依赖于它的对象会被自动告知并更新。Observer模式是一种松耦合的设计模式。

###### 46.3.2使用场景:

* 1. 假设热水器由三部分组成：热水器、警报器、显示器，它们来自于不同厂商并进行了组装。那么，应该是**热水器**仅仅负责烧水，它不能发出警报也不能显示水温；在水烧开时由**警报器**发出警报、**显示器**显示提示和水温。
  2. 问题：如何在水烧开的时候通知报警器和显示器？Observer设计模式中主要包括如下两类对象：

1. Subject：监视对象，它往往包含着其他对象所感兴趣的内容。在本范例中，热水器就是一个监视对象，它包含的其他对象所感兴趣的内容，就是temprature字段，当这个字段的值快到100时，会不断把数据发给监视它的对象。
2. Observer：监视者，它监视Subject，当Subject中的某件事发生的时候，会告知Observer，而Observer则会采取相应的行动。在本范例中，Observer有警报器和显示器，它们采取的行动分别是发出警报和显示水温。
   1. 在本例中，事情发生的顺序应该是这样的：
3. 警报器和显示器告诉热水器，它对它的温度比较感兴趣(注册)。
4. 热水器知道后保留对警报器和显示器的引用。
5. 热水器进行烧水这一动作，当水温超过95度时，通过对警报器和显示器的引用，自动调用警报器的MakeAlert()方法、显示器的ShowMsg()方法。

###### 46.3.3定义代码:

###### 46.3.4 备注:

###### 46.3.5 .Net Framework中的委托与事件

46.3.5.1 .Net Framework的编码规范

1. 委托类型的名称都应该以EventHandler结束。
2. 委托的原型定义：有一个void返回值，并接受两个输入参数：一个Object 类型，一个 EventArgs类型(或继承自EventArgs)。
3. 事件的命名为 委托去掉 EventHandler之后剩余的部分。
4. 继承自EventArgs的类型应该以EventArgs结尾。

46.3.5.2 说明

* 1. 委托声明原型中的Object类型的参数代表了Subject，也就是监视对象，在本例中是 Heater(热水器)。回调函数(比如Alarm的MakeAlert)可以通过它访问触发事件的对象(Heater)。
  2. EventArgs 对象包含了Observer所感兴趣的数据，在本例中是temperature。
  3. 上面这些其实不仅仅是为了编码规范而已，这样也使得程序有更大的灵活性。比如说，如果我们不光想获得热水器的温度，还想在Observer端(警报器或者显示器)方法中获得它的生产日期、型号、价格，那么委托和方法的声明都会变得很麻烦，而如果我们将热水器的引用传给警报器的方法，就可以在方法中直接访问热水器了。

##### 46.4 总结

###### 46.4.1事件与Observer

1. 单独谈论事件，我们说发布者(publisher)、订阅者(subscriber)、客户端(client)
2. 讨论Observer模式，我们说主题(subject)和观察者(observer)。客户端通常是包含Main()方法的Program类

###### 46.4.2事件与委托的使用场合

当使用委托变量时，客户端可以直接通过委托变量触发事件，也就是直接调用pub.NumberChanged(100)，这将会影响到所有注册了该委托的订阅者。而事件的本意应该为在事件发布者在其本身的某个行为中触发，比如说在方法DoSomething()中满足某个条件后触发。通过添加event关键字来发布事件，事件发布者的封装性会更好，事件仅仅是供其他类型订阅，而客户端不能直接触发事件（语句pub.NumberChanged(100)无法通过编译），事件只能在事件发布者Publisher类的内部触发（比如在方法pub.DoSomething()中），换言之，就是NumberChanged(100)语句只能在Publisher内部被调用。

###### 46.4.3为什么委托定义的返回值通常都为void？

委托变量可以供多个订阅者注册，如果定义了返回值，那么多个订阅者的方法都会向发布者返回数值，结果就是后面一个返回的方法值将前面的返回值覆盖掉了，因此，实际上只能获得最后一个方法调用的返回值

###### 46.4.4如何让事件只允许一个客户订阅？

* 1. 将事件声明为private的，然后提供两个方法来进行注册和取消注册

代码如下：

// 定义事件发布者  
public class Publishser {  
    private event GeneralEventHandler NumberChanged;    // 声明一个私有事件  
    // 注册事件  
    public void Register(GeneralEventHandler method) {  
        NumberChanged = method;  
    }  
    // 取消注册  
    public void UnRegister(GeneralEventHandler method) {  
        NumberChanged -= method;  
    }  
  
    public void DoSomething() {  
        // 做某些其余的事情  
        if (NumberChanged != null) {    // 触发事件  
            string rtn = NumberChanged();  
            Console.WriteLine("Return: {0}", rtn);      // 打印返回的字符串，输出为Subscriber3  
        }  
    }  
}

* 1. 事件访问器（Event Accessor）的东西，它用来封装委托变量

代码如下：

class Program {  
    static void Main(string[] args) {  
        Publishser pub = new Publishser();  
        Subscriber1 sub1 = new Subscriber1();  
        Subscriber2 sub2 = new Subscriber2();  
  
        pub.NumberChanged -= sub1.OnNumberChanged;  // 不会有任何反应  
        pub.NumberChanged += sub2.OnNumberChanged;  // 注册了sub2  
        pub.NumberChanged += sub1.OnNumberChanged;  // sub1将sub2的覆盖掉了  
          
        pub.DoSomething();          // 触发事件  
    }  
}  
  
// 定义委托  
public delegate string GeneralEventHandler();  
  
// 定义事件发布者  
public class Publishser {  
    // 声明一个委托变量  
    private GeneralEventHandler numberChanged;  
    // 事件访问器的定义  
    public event GeneralEventHandler NumberChanged {  
        add {  
            numberChanged = value;  
        }  
        remove {  
            numberChanged -= value;  
        }  
    }  
      
    public void DoSomething() {  
        // 做某些其他的事情  
        if (numberChanged != null) {    // 通过委托变量触发事件  
            string rtn = numberChanged();  
            Console.WriteLine("Return: {0}", rtn);      // 打印返回的字符串  
        }  
    }  
}  
  
// 定义事件订阅者  
public class Subscriber1 {  
    public string OnNumberChanged() {  
        Console.WriteLine("Subscriber1 Invoked!");  
        return "Subscriber1";  
    }  
}

###### 46.4.5获得多个返回值与异常处理

* + 1. 获得多个订阅者的返回值:
       1. 委托定义在编译时会生成一个继承自MulticastDelegate的类，而这个MulticastDelegate又继承自Delegate，在Delegate内部，维护了一个委托链表，链表上的每一个元素，为一个只包含一个目标方法的委托对象。而通过Delegate基类的GetInvocationList()静态方法，可以获得这个委托链表。
    2. 异常处理:
       1. 在异常处理中, 很有可能在触发事件时，订阅者的方法会抛出异常，而这一异常会直接影响到发布者，使得发布者程序中止，而后面订阅者的方法将不会被执行。
       2. 代码：

// 触发某个事件，以列表形式返回所有方法的返回值  
public static object[] FireEvent(Delegate del, params object[] args){  
  
    List<object> objList = new List<object>();  
  
    if (del != null) {  
        Delegate[] delArray = del.GetInvocationList();  
        foreach (Delegate method in delArray) {  
            try {  
                // 使用DynamicInvoke方法触发事件  
                object obj = method.DynamicInvoke(args);  
                if (obj != null)  
                    objList.Add(obj);  
            } catch { }  
        }  
    }  
    return objList.ToArray();  
}

public void DoSomething() {  
    // 做某些其他的事情  
    Program5.FireEvent(MyEvent, this, EventArgs.Empty);  
}

###### 46.4.6委托中订阅者方法超时的处理

* 1. 超时：方法运行所需要的时间超过一定的时间，导致服务器端与客户端之间的连接断开
  2. 超时和异常的区别：就是超时并不会影响事件的正确触发和程序的正常运行，却会导致事件触发后需要很长才能够结束。
  3. 委托的定义会生成继承自MulticastDelegate的完整的类，其中包含Invoke()、BeginInvoke()和EndInvoke()方法
  4. 直接调用委托时，实际上是调用了Invoke()方法，它会中断调用它的客户端，然后在客户端线程上执行所有订阅者的方法（客户端无法继续执行后面代码），最后将控制权返回客户端。用于方法的异步执行，即是在调用BeginInvoke()之后，客户端从线程池中抓取一个闲置线程，然后交由这个线程去执行订阅者的方法，而客户端线程则可以继续执行下面的代码。
  5. BeginInvoke()接受“动态”的参数个数和类型，其中前面参数的个数和类型与委托定义中接受的参数个数和类型相同，最后两个参数分别是AsyncCallback和Object类型
  6. 注意：
     1. 在委托类型上调用BeginInvoke()时，此委托对象只能包含一个目标方法，所以对于多个订阅者注册的情况，必须使用GetInvocationList()获得所有委托对象，然后遍历它们，分别在其上调用BeginInvoke()方法。如果直接在委托上调用BeginInvoke()，会抛出异常，提示“委托只能包含一个目标方法”。
     2. 如果订阅者的方法抛出异常，.NET会捕捉到它，但是只有在调用EndInvoke()的时候，才会将异常重新抛出。而在本例中，我们不使用EndInvoke()（因为我们不关心订阅者的执行情况），所以我们无需处理异常，因为即使抛出异常，也是在另一个线程上，不会影响到客户端线程（客户端甚至不知道订阅者发生了异常，这有时是好事有时是坏事）。
     3. BeginInvoke()方法属于委托定义所生成的类，它既不属于MulticastDelegate也不属于Delegate基类，所以无法继续使用可重用的FireEvent()方法，我们需要进行一个向下转换，来获取到实际的委托类型。

###### 46.4.7委托和方法的异步调用

对于每一个异步调用都通过创建线程来进行操作显然会对性能产生一定的影响，同时操作也相对繁琐一些

.Net中可以通过委托进行方法的异步调用，就是说客户端在异步调用方法时，本身并不会因为方法的调用而中断，而是从线程池中抓取一个线程去执行该方法，自身线程（主线程）在完成抓取线程这一过程之后，继续执行下面的代码，这样就实现了代码的并行执行。

使用线程池的好处就是避免了频繁进行异步调用时创建、销毁线程的开销。

当客户端调用EndInvoke()时，如果异步调用的方法没有执行完毕，则会中断当前线程而去等待该方法，只有当异步方法执行完毕后才会继续执行后面的代码。所以在调用完BeginInvoke()后立即执行EndInvoke()是没有任何意义的。我们通常在尽可能早的时候调用BeginInvoke()，然后在需要方法的返回值的时候再去调用EndInvoke()，或者是根据情况在晚些时候调用。

AsyncResult可以唯一的获取到与它相关的调用了的方法（或者也可以理解成委托对象）

我们在调用BeginInvoke()后不再需要保存IAysncResult了，因为AysncCallback委托将该对象定义在了回调方法的参数列表中；

### 队列Queue

##### What

###### 引言

在。

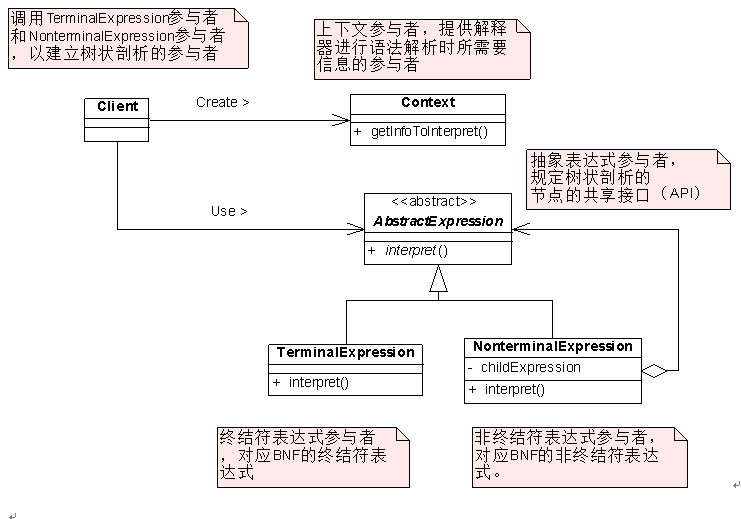
###### 动机（Motivate）

1. 在生活当中存在存在着一种按照一定顺序排列，并且依照顺序进和出的这样一种结构，在C#中，和List一样属于数据结果的一种。比如：蚂蚁给蚁后进食，是严格按照先后顺序进行的，排在前面的先进先出，排在后面的后进后出，再比如：阅兵的方队

###### 意图（Intent）

1. 队列是一种特殊的[线性表](https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E8%A1%A8/3228081" \t "_blank)，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。
2. First In First Out

###### 结构图（Structure）



###### 组成

1. **抽象表达式(AbstractExpression)**：定义解释器的接口，约定解释器的解释操作。其中的Interpret接口，正如其名字那样，它是专门用来解释该解释器所要实现的功能。
2. **终结符表达式(Terminal Expression)**：实现了抽象表达式角色所要求的接口，主要是一个interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应。比如有一个简单的公式R=R1+R2，在里面R1和R2就是终结符，对应的解析R1和R2的解释器就是终结符表达式。
3. **非终结符表达式(Nonterminal Expression)**：文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式R=R1+R2中，“+”就是非终结符，解析“+”的解释器就是一个非终结符表达式。
4. **环境角色(Context)**：这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如R=R1+R2，我们给R1赋值100，给R2赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了。
5. **客户端（Client）：**指的是使用解释器的客户端，通常在这里将按照语言的语法做的表达式转换成使用解释器对象描述的抽象语法树，然后调用解释操作。

##### How

###### 实现方法

1. 在很多场合都需要把数字转换成中文，我们就可以使用解释器来实现该功能，把给定的数字解释成符合语法规范的汉字表示法。
2. **Component**
   1. ）

###### 实现要点

1. 要点
   1. 使用Interpreter模式来表示文法规则，从而可以使用面向对象技巧方便地“扩展”文法。
   2. Interpreter模式比较适合简单的文法表示，对于复杂的文法表示，Interpreter模式会产生比较大的类层次结构，需要求助于语法分析生成器这样的标准工具。
2. 优点
   1. 易于改变和扩展文法。
   2. 每一条文法规则都可以表示为一个类，因此可以方便地实现一个简单的语言。
   3. 实现文法较为容易。在抽象语法树中每一个表达式节点类的实现方式都是相似的，这些类的代码编写都不会特别复杂，还可以通过一些工具自动生成节点类代码。
   4. 增加新的解释表达式较为方便。如果用户需要增加新的解释表达式只需要对应增加一个新的终结符表达式或非终结符表达式类，原有表达式类代码无须修改，符合“开闭原则”。
3. 缺点
   1. 对于复杂文法难以维护。在解释器模式中，每一条规则至少需要定义一个类，因此如果一个语言包含太多文法规则，类的个数将会急剧增加，导致系统难以管理和维护，此时可以考虑使用语法分析程序等方式来取代解释器模式。
   2. 执行效率较低。由于在解释器模式中使用了大量的循环和递归调用，因此在解释较为复杂的句子时其速度很慢，而且代码的调试过程也比较麻烦
4. 场景
   1. Interpreter模式的应用场合是Interpreter模式应用中的难点，只有满足“业务规则频繁变化，且类似的模式不断重复出现，并且容易抽象为语法规则的问题”才适合使用Interpreter模式。
   2. 当一个语言需要解释执行，并可以将该语言中的句子表示为一个抽象语法树的时候，可以考虑使用解释器模式（如XML文档解释、正则表达式等领域）
   3. 一些重复出现的问题可以用一种简单的语言来进行表达。
   4. 一个语言的文法较为简单.
   5. 当执行效率不是关键和主要关心的问题时可考虑解释器模式（注：高效的解释器通常不是通过直接解释抽象语法树来实现的，而是需要将它们转换成其他形式，使用解释器模式的执行效率并不高。

##### Refer

###### .NET 场景：

1. 正则表达式就是一个典型的解释器。ASP.NET中，把aspx文件转化为dll时，会对html语言进行处理，这个处理过程也包含了解释器的模式在里面。Interpreter模式其实有Composite模式的影子，但它们解决的问题是不一样的。

##### Summary

对

# Asp.Net Core

### 会话和应用状态

HTTP 是无状态的协议。不采取其他步骤的情况下，HTTP 请求是不保留用户值或应用状态的独立消息。

| **存储方法** | **存储机制** |
| --- | --- |
| Cookie | HTTPCookie（可能包括使用服务端应用代码存储的数据） |
| 会话状态 | HTTPCookie和服务端应用代码 |
| TempData | HTTP Cookie 或会话状态 |
| 查询字符串 | HTTP查询字符串 |
| 隐藏字段 | HTTP窗体字段 |
| HttpContext.Items | 服务器端应用代码 |
| 缓存 | 服务器端应用代码 |
| 依赖关系注入 | 服务器端应用代码 |

##### Cookie

###### Cookie 存储所有请求的数据。因为 Cookie 是随每个请求发送的，所以它们的大小应该保持在最低限度。

1. 理想情况下，仅标识符应存储在 Cookie 中，而数据则由应用存储。
2. 大多数浏览器 Cookie 大小限制为 4096 个字节。每个域仅有有限数量的 Cookie 可用。

###### 由于 Cookie 易被篡改，因此它们必须由服务器进行验证。

* + - 1. 客户端上的 Cookie 可能被用户删除或者过期。但是，Cookie 通常是客户端上最持久的数据暂留形式。

###### Cookie 通常用于个性化设置，其中的内容是为已知用户定制的。

* + - 1. 大多数情况下，仅标识用户，但不对其进行身份验证。
      2. Cookie 可以存储用户名、帐户名或唯一的用户 ID（例如 GUID）。然后，可以使用 Cookie 访问用户的个性化设置，例如首选的网站背景色。

##### 会话状态

会话状态是在用户浏览 Web 应用时用来存储用户数据的 ASP.NET Core 方案。

###### 会话状态机制

* + - 1. 会话状态使用应用维护的存储来配置会话状态保存客户端所有请求的数据。
      2. 会话数据由缓存支持并被视为临时数据 - 站点应在没有会话数据的情况下继续运行。
      3. 关键应用程序数据应存储在用户数据库中，并仅作为性能优化缓存在会话中。
      4. ASP.NET Core 通过向客户端提供包含会话 ID 的 Cookie 来维护会话状态，该会话 ID 与每个请求一起发送到应用。应用使用会话 ID 来获取会话数据。

###### 会话状态行为：

* + - 1. 由于会话 Cookie 是特定于浏览器的，因此不能跨浏览器共享会话。
      2. 浏览器会话结束时删除会话 Cookie。
      3. 如果收到过期的会话 Cookie，则创建使用相同会话 Cookie 的新会话。
      4. 不会保留空会话 - 会话中必须设置了至少一个值以保存所有请求的会话。会话未保留时，为每个新的请求生成新会话 ID。
      5. 应用在上次请求后保留会话的时间有限。应用设置会话超时，或者使用 20 分钟的默认值。会话状态适用于存储特定于特定会话的用户数据，但该数据无需永久的会话存储。
      6. 调用 ISession.Clear 实现或者会话过期时，会删除会话数据。
      7. 没有默认机制告知客户端浏览器已关闭或者客户端上的会话 Cookie 被删除或过期的应用代码。
      8. ASP.NET Core MVC 和 Razor Pages 模板包括对一般数据保护条例 (GDPR) 的支持。

###### 在内存中缓存会话状态数据（服务器方案）

1. 使用粘性会话将每个会话加入到单独服务器上的特定应用实例。默认情况下 Azune应用服务使用应用程序请求路由（ARR）强制实施粘性会话。
   * + - 1. 影响程序伸缩性，使web应用更新变得复杂
         2. 推荐使用Redis或 SqlServer 分布式缓存。他们不需要粘性会话。
2. 通过IDataProtector加密会话 Cookie。
3. 必须正确配置数据保护，以确定在每台计算机上读取会话 Cookie

###### 配置会话状态

1. 关系类库：
2. Microsoft.AspNetCore.App metapackage 中包含的 Microsoft.AspNetCore.Session 包提供中间件来管理会话状态。
3. 启用会话状态中间件
   1. 任一 IDistributedCache内存缓存。 IDistributedCache实现会话后备存储
   2. 对 ConfigureServices中 AddSession的调用
   3. 对 Configure中 UseSession 的调用



1. 中间件的顺序很重要。配置会话状态后，HttpContext.Session可用。调用UseSession以前无法访问HttpContext.Session
2. 在应用已经开始写入到响应流之后，不能创建有新会话Cookie的新会话。

###### 以异步方式加载会话状态

1. 只有在TryGetValue、Set或者 Remove方法显示调用 ISeesion.LoadAsync方法，Asp.Net Core 默认会话提供程序才会从基础的 IDistrubutedCache后备存储中以异步形式加载会话记录。
2. 如果未先调用 LoadAsync，则会同步加载基础会话记录，可能对性能产生大规模影响。

###### 会话选项SessionOptions

1. Cookie用于创建Cookie的设置
   1. 名称默认为 SessionDefaults.CookieName(.AspNetCore.Session)
   2. 路径默认为SessionDefaults.CookiePath(/)
   3. SameSite 默认为SameSiteMode.Lax(1)
   4. HttpOnly默认为 true
   5. IsEsstensial默认为 false
2. IdelTimeout超时时间
   1. 显示放弃内容前，内容可以空闲多长时间。每个会话访问都会重置超时。
   2. 此设置仅适用于会话内容，不适用于Cookie。默认为20分钟。
3. IOTimeout从存储加载会话或者提交回存储的超时时间
   1. 此设置仅适用于异步操作，可使用 InfiniteTimeSpan禁用超时，默认1分钟。

###### 设置或获会话状态

1. 使用HttpContext.Session获取会话状态。此属性是ISession的实现。
2. 相关包：
   1. Microsoft.AspNetCore.App
   2. Microsoft.AspNetCore.Http
   3. Microsoft.AspNetCore.Http.Extensions
3. 相关扩展方法
   1. Get(ISession,String)
   2. GetInt32(Session,String)
   3. GetString(ISession,string)
   4. SetInt32(ISession,String,Int32)
   5. SetString(ISession,String,String)

###### 注意

1. 请勿将敏感数据存储在会话状态中。用户可能不会关闭浏览器并清除会话 Cookie。某些浏览器会保留所有浏览器窗口中的有效会话 Cookie。会话可能不限于单个用户 - 下一个用户可能继续使用同一会话 Cookie 浏览应用。
2. 必须对所有的会话数据进行序列化以启用分布式缓存方案，即使是在使用内存中缓存的时候。提供最小的字符串和数字序列化程序

##### TempData

此属性存储未读取的数据。 Keep 和 Peek 方法可用于检查数据，而不执行删除。多个请求需要数据时，TempData 非常有助于进行重定向。使用 Cookie 或会话状态通过 TempData 提供程序实现 TempData。

###### TempData 提供程序

1. 基于 cookie 的 TempData 提供程序默认用于存储 cookie 中的 TempData。
   1. 使用由 Base64UrlTextEncoder 编码的 IDataProtector 对 Cookie 数据进行加密，然后进行分块。
   2. 未压缩 Cookie 数据，因为压缩加密的数据会导致安全问题，如 CRIME 和 BREACH 攻击。
   3. 提供程序CookieTempDataProvider

###### 选择 TempData 提供程序

1. 应用是否已使用会话状态？如果是，使用会话状态 TempData 提供程序对应用没有额外的成本（除了数据的大小）。
2. 应用是否只对相对较小的数据量（最多 500 个字节）使用 TempData？如果是，Cookie TempData 提供程序将为每个携带 TempData 的请求增加较小的成本。如果不是，会话状态 TempData 提供程序有助于在使用TempData 前，避免在每个请求中来回切换大量数据。
3. 应用是否在多个服务器上的服务器场中运行？如果是，无需其他任何配置，即可在数据保护外使用 CookieTempData 提供程序
4. 大多数 Web 客户端（如 Web 浏览器）针对每个 Cookie 的最大大小和/或 Cookie 总数强制实施限制。使用 Cookie TempData提供程序时，请验证应用未超过这些限制。考虑数据的总大小。解释加密和分块导致的 Cookie 大小增加。

###### 配置 TempData 提供程序

### Docker

##### 安装（针对WIN10）

###### 开启Hyper-V

1. 在Windows 功能添加Hyper-V功能
2. 有可能需要重启

###### 安装Toolbox

* + - 1. <https://www.docker.com/get-docker>
      2. 点击Download Desktop and Take a Tutorial
      3. 运行Installer.exe 一路Next，点击Finish完成安装

###### 运行Docker

* + - 1. Docker version 查看版本。
      2. Docker run hello-world 载入测试镜像

###### 镜像加速

* + - 1. 新版的 Docker 使用 /etc/docker/daemon.json（Linux） 或者 %programdata%\docker\config\daemon.json（Windows） 来配置 Daemon。
      2. 请在该配置文件中加入（没有该文件的话，请先建一个）：

{"registry-mirrors": ["http://hub-mirror.c.163.com"]}

##### 使用

###### 在容器内运行应用程序

* + - 1. docker run ubuntu:15.10 /bin/echo/ “Hello world”
         1. docker：Docker主命令，二进制执行文件
         2. run：运行一个容器
         3. ubuntu：容器类型
         4. :15.10 容器类型的版本。Docker首先从本地本地主机上查找镜像是否存在，如果不存在，Docker则从镜像镜仓库Docker Hub上下载公共镜像
         5. /bin/echo “Hello world” 在启动的容器里执行的命令
         6. 以上完整意思就是：Docker以Ubuntu15.10镜像创建一个新容器，然后在容器里执行bin/echo “Hello world”,然后输出结果
      2. docker run -I -t ubuntu:15.10 /bin/bash
         1. -t：在新容器内指定一个伪终端或终端
         2. -i：允许你对容器内的标准（STDIN）进行交互
      3. 退出容器
         1. exit
         2. CTRL+D
      4. 启动容器
      5. 停止容器
         1. docker stop docker\_name
      6. 设置镜像

###### 常用命令：

* + - 1. 拉取镜像
         1. docker pull postgres
      2. 安装镜像
         1. docker run postgres
      3. docker run 命令大全
         1. -a stdin:指定标准输入输出内容类型，可选STDIN/STDOUT/STDERR三项
         2. **-d:** 后台运行容器，并返回容器ID
         3. **-i:** 以交互模式运行容器，通常与 -t 同时使用
         4. **-P:** 随机端口映射，容器内部端口**随机**映射到主机的高端口
         5. **-p:** 指定端口映射，格式为：**主机(宿主)端口:容器端口**
         6. **-t:** 为容器重新分配一个伪输入终端，通常与 -i 同时使用
         7. **--name="nginx-lb":** 为容器指定一个名称
         8. **--dns 8.8.8.8:** 指定容器使用的DNS服务器，默认和宿主一致
         9. **--dns-search example.com:** 指定容器DNS搜索域名，默认和宿主一致
         10. **-h "mars":** 指定容器的hostname
         11. **-e username="ritchie":** 设置环境变量
         12. **--env-file=[]:** 从指定文件读入环境变量
         13. **--cpuset="0-2" or --cpuset="0,1,2":** 绑定容器到指定CPU运行
         14. **-m :**设置容器使用内存最大值
         15. **--net="bridge":** 指定容器的网络连接类型，支持 bridge/host/none/container: 四种类型
         16. **--link=[]:** 添加链接到另一个容器
         17. **--expose=[]:** 开放一个端口或一组端口
         18. **--volume , -v:** 绑定一个卷

# Design &&Pattern

### Design Pattern

##### 定义

###### 定义

1. 设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。

##### 原则

###### 单一职责原则（Single Responsibility Principle，SRP）

1. 类的职责要单一。就一个类而言，应该仅有一个引起它变化的原因。
2. 如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其它职责的能力。
3. 举个简单的例子，我们现在很多时候都直接使用手机拍照，但新闻、时尚杂志等的拍照使用的是相机。手机其实就相当于把相机耦合到手机中去了，而相机却只有拍照这一个职责。大多数时候，一件产品简单一些，职责单一一些，或许是更好的选择，这就是单一职责原则。

###### 开闭原则（Open Close Principle，OCP）

1. 软件实体可扩展，但不可修改。
2. 就好比“一国两制”，一个国家有两种制度，没有修改原有的管理制度，而是增加了一种新的制度。正所谓对于扩展的开放的，对于更改是封闭的。

###### 依赖倒转原则（Dependence Inversion Principle，DIP）

1. 高层模块不应该依赖底层模块，两个都应该依赖抽象；抽象不应该依赖细节，细节应该依赖抽象。后面这句话换个方式说就是针对接口编程，不要对实现编程。
2. 举个例子，就好比你的电脑，如果CPU、内存、硬盘都需要依赖具体的主板，主板坏了，所有的部件都不能用了，这显然是不合理的。
3. 换句话说，即谁也不要依赖谁，除了约定的接口，大家都可以灵活自如。
4. 建议：
   1. 没有变量引用具体的类(可已使用工厂代替创建这个具体的类)
   2. 没有类派生于具体的类(派生于它就依赖于它)
   3. 不去重写(override)其任一父类的已实现方法(如果重写了, 那么这个类并不适合作为起始的抽象类, 因为基类里面的方法本应该是共享与所有子类的)

###### 里氏代换原则（Liskov Substitution Principle，LSP）

1. 子类型必须能够替换掉它们的父类型。
2. 举个例子，假如现在父类是鸟类，有个属性是可以飞，子类有个企鹅类，那么很明显企鹅类不能继承鸟类，因为鸟类有会飞的属性，如果子类企鹅类继承父类鸟类，则企鹅必须会飞。

###### 接口隔离原则（Interface Segregation Principle，ISP）

1. 接口尽量细化，同时接口中的方法尽量少。
2. 举个设计模式之禅中的例子，星探找美女，星探要去找美女，他需要通过一个标准来找，比如可以分为外形美女和气质美女（即两个接口），这样不管以后是要气质美女还是外形美女，都可以保持接口的稳定。

###### 迪米特法则（Law of Demeter，LoD）/最少知识原则（Least Knowledge Principle，LKP）

1. 一个类对自己依赖的类知道的越少越好；一个对象应该对其他对象有最少的了解。
2. 比如体育老师上课前清点人数，他让体育委员来替他清点人数，之后再由体育委员告知结果即可，老师本身并没有接触到其他学生。
3. 换句话说，每个类都应该尽量降低成员的访问权限，强调了类之间的松耦合。

###### 合成/聚合复用原则（Composite/Aggregate Reuse Principle，CARP）

1. 尽量使用合成/聚合，尽量不要使用类继承。
2. 合成表示一种强的“拥有”关系，体现了严格的部分与整体的关系，他们的生命周期相同；而聚合表示一种弱的“拥有”关系，体现的是A对象可以包含B对象，但B对象不是A对象的一部分。
3. 这里再举个简单的例子，大雁与翅膀是合成关系；而每只大雁都属于一个雁群，即一个雁群可以有多只大雁，所以大雁和雁群是聚合关系。

##### 设计模式类型

###### 创建型模式

1. [单例模式](http://www.cnblogs.com/adamjwh/p/9033554.html)
2. [工厂方法模式](http://www.cnblogs.com/adamjwh/p/9033553.html)（工厂模式）
3. [抽象工厂模式](http://www.cnblogs.com/adamjwh/p/9033552.html)
4. [原型模式](http://www.cnblogs.com/adamjwh/p/9033550.html)
5. [建造者模式](http://www.cnblogs.com/adamjwh/p/9033551.html)（生成器模式）

###### 结构型模式

1. [适配器模式](http://www.cnblogs.com/adamjwh/p/9033549.html)（变压器模式/包装模式）
2. [桥接模式](http://www.cnblogs.com/adamjwh/p/9033548.html)（桥梁模式）
3. [组合模式](http://www.cnblogs.com/adamjwh/p/9033547.html)（合成模式/部分-整体模式）
4. [装饰模式](http://www.cnblogs.com/adamjwh/p/9036358.html)（装饰器模式）
5. [外观模式](http://www.cnblogs.com/adamjwh/p/9048594.html)（门面模式）
6. [享元模式](http://www.cnblogs.com/adamjwh/p/9070107.html)
7. [代理模式](https://www.cnblogs.com/adamjwh/p/9102037.html)（委托模式）

###### 行为型模式

1. [观察者模式](https://www.cnblogs.com/adamjwh/p/10913660.html)（发布订阅模式）
2. [模板方法模式](https://www.cnblogs.com/adamjwh/p/10919149.html)
3. [命令模式](https://www.cnblogs.com/adamjwh/p/10923122.html)
4. [状态模式](https://www.cnblogs.com/adamjwh/p/10926952.html)
5. [职责链模式](https://www.cnblogs.com/adamjwh/p/10932216.html)（责任链模式）
6. [解释器模式](https://www.cnblogs.com/adamjwh/p/10938852.html)
7. [中介者模式](https://www.cnblogs.com/adamjwh/p/10959987.html)
8. [访问者模式](https://www.cnblogs.com/adamjwh/p/10968634.html)
9. [策略模式](https://www.cnblogs.com/adamjwh/p/11011095.html)（政策模式）
10. [备忘录模式](https://www.cnblogs.com/adamjwh/p/11018268.html)
11. [迭代器模式](https://www.cnblogs.com/adamjwh/p/11024311.html)

### Inverse of Control（IoC）

##### WHAT

“控制反转”或者“控制倒置”, 它体现的意思是控制权的转移，即原来控制权在A手中，现在需要B来接管。

###### 概念

1. Inversion of Control 控制反转
2. 谁控制，控制谁？
   1. 程序提供的容器来控制程序调用组件的创建
3. 谁反转，反转谁？
   1. 从程序创建要调用的组件的实例，反转为程序提供的容器为程序推送要调用的组件的实例
4. 好莱坞原则 Don’t call us,we’ll call you.
   1. 程序不关心要调用的是什么，即don’t call us
   2. 程序容器推送给程序的组件是什么，就调用什么，即 we’ll call you.

##### WHERE

###### 说明

1. IoC一方面通过流程控制从应用程序向框架的反转实现了针对流程自身的重用，另一方面采用“好莱坞原则”使得这个被重用的流程可能自由地被定制，这两个因素实际上决定了框架自身的价值。
2. 重用让框架不仅仅是为应用程序提供实现单一功能的API，而是提供一整套可执行的解决方案；可定制则使我们可以为不同的应用程序对框架进行定制，这无疑让框架可以使用到更多的应用之中。

##### WHY

###### 说明

1. 可以有效的解决程序和被调用组件之间的耦合。被调用组件通过容器可以被定制，程序而不用去关心组件的实现

##### HOW

1. 程序

### Git用法

##### 客户端操作

###### 设置账户

需要和github账户设置一致

* + - * 1. git config –global user.name xxx
        2. git config –global user.email xxx@foxmail.com

###### 查看设置

1. git config --list

###### 创建git本地库

* + - 1. git init (在当前目录下创建本地仓库)

###### 查看git状态

* + - * 1. git status(显示需要提交的文件和未追踪的文件)

uncommited已有的，修改过但未提交的文件

untracked原先没有的，新建的

###### 添加git文件到暂存区(需要和版本库区分)

* + - 1. git add <name>

###### 提交文件

1. git commit -m “add a commit remark”
2. -m表示注释

###### 删除文件（夹），并没有提交到服务器

* + - 1. git rm test.txt // 删除文件
      2. git rm -r filebook //删除文件夹

###### 删除文件（夹），并提交到服务器

* + - 1. git rm test.txt => git commit -m “delete a file”
      2. rm test.txt => git commit -am “delete a file”
      3. 命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

###### 操作日志

* + - 1. **git log --decorate --graph --oneline --all** #显示当前及之前的版本号
      2. **git log --pretty=oneline** #将版本历史显示为一行，历史版本号全部显示
      3. **git log --pretty=oneline --abbrev-commit** #将版本历史显示为一行，历史版本号部分显示
      4. **git log --graph**  #查看分支合并图

###### 版本回退

执行版本退回后，本地工作区的内容会自动和回退到的版本库版本的内容保持同步

* + - * 1. **git reset --hard HEAD^**回退到上一个版本
        2. **git reset --hard HEAD^^**回退到上上个版本，以此类推，一次提交即为一个版本
        3. **git reset --hard *e9efa77***回退到***e9efa77***版本

###### 还原操作

丢弃工作区的操作，但不会丢失暂存区的操作(add操作能将更改添加到暂存区)，实际上就是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”

* + - * 1. **git checkout -- readme.txt**

###### 暂存区撤销操作

工作区修改了文件，而且执行了add，没执行commit操作，暂存区还是可以撤销的

* + - * 1. **git reset HEAD readme.txt**
        2. 备注：git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

##### 操作服务器

###### 配置远程仓库免密登陆

1. 在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有**id\_rsa**和**id\_rsa.pub**这两个文件，如果已经有了，可直接跳到下一步。
2. 如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：ssh-keygen -t rsa -C "xxx@foxmail.com" 一路回车，执行生成 id\_rsa 私钥和 id\_rsa.pub 公钥，Windows用户在git bash中输入上述指令
3. 获得key的内容，复制下来，添加到gitHub的SSH key中
   1. **windows位置：C:\Users\用户名\.ssh\id\_rsa.pub**‬‬
   2. **Linux位置：cat ~/.ssh/id\_rsa.pub**
4. **ssh -T git@github.com** #验证key，根据提示输入yes，添加为信任主机

###### 添加远程仓库

1. **git remote add origin https://github.com/xxx/LearnGit.git（https方式）**
2. 此处可以为https地址也可以是ssh地址，orign为设置的远程仓库的别名，强烈建议使用ssh方式，因为https方式每次都要输入用户名和密码
3. 修改传输协议
   1. **git remote rm <远程主机名>**（删除远程仓库）
   2. **设置传输方式和目标远程仓库**
   3. **git push -u <远程主机名><本地分支名>**
4. 因为git本身是分布式版本控制系统，可以同步到另外一个远程库，当然也可以同步到另外两个远程库，所以**一个本地库可以既关联GitHub，又关联码云**！
   1. **git remote rm origin**
   2. 先关联GitHub的远程库：

**git remote add github** [**git@github.com:xxx/LearnGit.git**](mailto:git@github.com:xxx/LearnGit.git)

远程库的名称叫github，不叫origin了

* 1. 再关联码云的远程库

**git remote add gitee git@gitee.com:xxx/LearnGit.git**

* 1. 推送到GitHub，使用命令

**git push github master**

* 1. 推送到码云，使用命令

**git push gitee master**

###### 查看远程仓库及传输协议

1. **git remote**
2. **git remote -v 查看名称和详细地址**

###### 删除远程仓库

1. **git remote remove <远程主机名>**

###### 推送本地分支到远程仓库

1. **git push <远程主机名><本地分支名>:<远程分支名>(**如果省略远程分支名，则表示将本地分支推送与之存在“追踪关系”的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。**)**
2. **git push origin <本地分支名>**
3. **git push origin master**
4. 如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样以后就可以不加任何参数使用git push。
5. **git push -u <远程主机名><本地分支名> 例如：git push -u origin master**

###### 将远程仓库克隆为本地仓库

1. **git clone** [**git@github.com:xxx/LearnGit.git**](mailto:git@github.com:xxx/LearnGit.git)
   1. 不能使用别名
   2. 默认情况下，从远程clone到本地的库只能看到master分支
2. 将远程的分支同步到本地

**git checkout -b <本地分支名><远程主机名>/<远程分支名>**

前提是远程<远程主机名>必须存在名为<远程分支名>的分支，而且<本地分支名>和<远程分支名>最好一致。

###### 本地仓库更新

将远程存储库中的更改合并到当前分支中。在默认模式下，git pull是git fetch后跟git merge FETCH\_HEAD的缩写。更准确地说，git pull使用给定的参数运行git fetch，并调用git merge将检索到的分支头合并到当前分支中。使用--rebase，它运行git rebase而不是git merge。

1. 拉取分支
   1. **git pull <远程主机名><远程分支名>:<本地分支名>**
   2. 比如，要取回origin主机的next分支，与本地的master分支合并

**git pull origin next:master**

* 1. 如果远程分支(next)要与当前分支合并，则冒号后面的部分可以省略

**git pull origin next**

* 1. 表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再执行git merge。

**git fetch origin** =>**git merge origin/next**

1. 手动建立追踪关系

在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系(tracking)。比如，在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是说，本地的master分支自动“追踪”origin/master分支。

* 1. **git branch --set-upstream-to=远程主机名/<远程分支名>  <本地分支名>**
  2. 比如git branch --set-upstream-to=origin/next master，指定master分支追踪origin/next分支

1. 分支合并
   1. 本地当前分支自动与对应的origin主机”追踪分支”(remote-tracking branch)进行合并

**git pull origin**

* 1. 如果当前分支只有一个追踪分支，连远程主机名都可以省略

**git pull**

表示，当前分支自动与唯一一个追踪分支进行合并。

1. 采用rebase模式合并
   1. **git pull --rebase <远程主机名><远程分支名>:<本地分支名>**
2. **git fetch和git pull的区别**
   1. git fetch：相当于是从远程获取最新版本到本地，不会自动合并。
      1. **git fetch origin mastergit log -p master..origin/mastergit merge origin/master**
      2. 以上命令的含义：

首先从远程的origin的master主分支下载最新的版本到origin/master分支上

然后比较本地的master分支和origin/master分支的差别

最后进行合并

* + 1. 上述过程其实可以用以下更清晰的方式来进行：

**git fetch origin master:tmpgit diff tmp git merge tmp**

* 1. git pull：相当于是从远程获取最新版本并merge到本地
     1. **git pull origin master**
     2. 上述命令其实相当于git fetch 和 git merge
     3. **在实际使用中，git fetch更安全一些，因为在merge前，可以查看更新情况，然后再决定是否合并。**

###### 查看分支

1. git branch

###### 创建分支

1. **git branch <name>**

###### 创建并切换到分支

1. **git checkout -b <name>**
2. git checkout命令加上-b参数表示创建并切换，相当于以下两条命令
   1. git branch <name>git checkout <name>

###### 切换分支

1. **git checkout <name>**

切换分支后，在git bash中显示为绿色

###### 删除分支

1. **git branch -d <name>**
   1. 如果分支没有合并，删除分支就表示会丢失修改，此时git无法使用-d删除，可使用-D强行删除
   2. **git branch -D <name>**

###### 合并分支

git合并默认使用Fast forward模式，一旦删除分支，会丢掉分支信息，也就看不出来曾经做过合并

1. **git merge <name>** #基于当前分支，合并另外一个分支，前提需要保证分支之间不冲突
2. 如果强制禁用Fast forward模式，即普通模式，Git就会在merge时生成一个新的commit
   1. **git merge --no-ff -m "there is a comment" <name>**

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

* 1. **工作中，肯定需要不管有没有分支被删除，都要从分支历史上就查看所有的历史分支信息，所以要使用普通模式合并。**

###### 创建tag

1. **git tag <tagname>**#默认在HEAD版本
2. 对指定的commit版本创建tag
   1. 需要先找到历史commit的id

**git log --pretty=oneline --abbrev-commit**

* 1. 然后对指定的commit创建tag：

**git tag <tagname><commitid>**

1. 创建带有说明的tag，用-a指定标签名，-m指定说明文字

**git tag -a <tagname> -m "there is a tag description" [<commitid>]**

1. 通过-s用私钥签名一个标签，签名采用PGP签名

**git tag -s <tagname> -m "there is a tag description" [<commitid>]**

必须首先安装gpg（GnuPG），如果没有找到gpg，或者没有gpg密钥对，就会报错，参考GnuPG帮助文档配置Key。

###### 查看tag

1. **查看tag**

**git tag** #显示的tag不是按时间顺序排列，而是按字母顺序排列

1. 如果想查看tag和commit的对应关系

**git log --pretty=oneline --abbrev-commit**

1. 如果想查看tag的的详细情况

**git show <tagname>**

###### 删除tag

创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除：

1. 删除本地tag

**git tag -d <tagname>**

1. 删除服务器 tag
   1. 先本地删除：**git tag -d <tagname>**
   2. 再远程删除：**git push origin :refs/tags/<tagname>**

###### 推送标签至远程

1. 推送标签至远程

**git push origin <tagname>**

1. 一次性推送全部尚未推送到远程的本地标签

**git push origin --tags**

###### 现场的保存与恢复

1. 将目前的工作现场保存

**git stash**

1. 查看所有保存的工作现场

**git stash list**

1. 工作现场存在，恢复stash内容
   1. **git stash apply stash@{0}**

恢复后，stash内容并不删除，你需要用**git stash drop stash@{0}**来删除；

* 1. **git stash pop**

恢复的同时把stash内容也删了，这种方式省时省力

1. **注意点：**
   1. 如果在分支下新建文件，而尚未执行add操作，stash无法将新文件纳入保存的现场，因为stash只对被修改的被追踪的文件和暂存的变更有效，对于新文件必须先执行add。
   2. 如果修改分支下的已被追踪的文件，不管有没有对修改的文件进行add操作，如果执行stash，所有修改会被纳入保存的现场，而文件会恢复成修改前的状态。恢复现场后，文件又呈现被修改后的状态。特别的是，如果修改的文件在stash前已经被add了，恢复现场后，暂存区的内容就会清空，相当于这个文件从未被add一样。

###### 设置Git UI颜色

1. 让Git显示颜色，会让命令输出看起来更醒目

**git config --global color.ui true**

###### 忽略特殊文件

1. 在Git工作区的根目录下创建一个特殊的**.gitignore**文件，然后把要忽略的文件名填进去
2. <https://github.com/github/gitignore>
3. 忽略文件的原则是
   1. 忽略操作系统自动生成的文件，比如缩略图等；
   2. 忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
   3. 忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。
4. 把.gitignore也提交到Git
   1. **git add .gitignore**
   2. **git commit -m "there is a description"**
   3. 检验.gitignore的标准是git status命令是不是显示working tree clean
   4. **使用Windows的注意**：如果在资源管理器里新建一个.gitignore文件，系统会非常弱智地提示必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。
5. 如果确实想要添加已经被.gitignore忽略的文件，可以用-f强制添加到Git

**git add -f test.class**

1. 怀疑.gitignore写的有问题，需要查找哪个规则写错了，可以用git check-ignore命令检查

**git check-ignore -v App.class**.gitignore:3:\*.class App.class

表示.gitignore的第3行规则忽略了App.class这个文件，于是我们就可以知道应该修订哪个规则。

###### 为命令配置别名

1. 命令可以简写，用git st表示git status，再比如用co表示checkout、ci表示commit、br表示branch：
   1. **git config --global alias.co checkoutgit config --global alias.ci commitgit config --global alias.br branch**
   2. 以后提交就可以简写成：**git ci -m "there is a description"**
   3. --global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。
2. 命令git reset HEAD <filename>可以撤销暂存区的修改（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：
   1. **git config --global alias.unstage 'reset HEAD'**可以简化命令
   2. **git unstage test.py** = **git reset HEAD test.py**
3. 配置一个git last，让其显示最后一次提交信息：
   1. **git config --global alias.last 'log -1'**
   2. 用git last就能显示最近一次的提交
   3. git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

###### 修改配置文件

1. 配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。
2. 每个仓库的Git配置文件都放在.git/config文件中
3. 而当前用户的Git配置文件放在**用户主目录下的一个隐藏文件.gitconfig**中

##### 多人协作

###### 首先将远程仓库克隆为本地仓库

* + - * 1. **git clone git@github.com:xxx/LearnGit.git**

###### 在本地创建和远程分支对应的分支

* + - * 1. **git checkout -b <本地分支名> origin/<远程分支名>**

###### 本地和远程分支的名称最好一致；

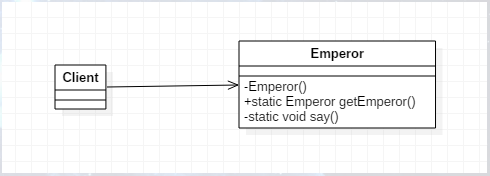
1. 在本地分支完成任务后，可以试图用**git push <远程主机名><本地分支名>**推送自己的修改；
2. 如果推送失败，则表明远程分支比本地更新，需要先用**git pull**试图合并；
3. 如果pull失败并提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令**git branch --set-upstream-to=<远程主机名>/<远程分支名>  <本地分支名>**创建链接；
4. 如果合并有冲突，则解决冲突，并在本地提交（add => commit）；
5. 没有冲突或者解决掉冲突后，再用**git push <远程主机名><本地分支名>**推送就能成功。

### 单例模式Singleton

##### What

###### 概念

1. 单例模式的目标就是只创建一个类的实例
2. 实际中有很多对象我们只需要他们的一个实例，例如：线程池、缓存、弹出的对话框，用户保存的设置的类，用于Loggin的类，硬件设备驱动对象等等，对于创建和销毁实例需要消耗大量的资源的对象
3. UML



###### 分析

1. 定义一个类

public class MyClass{}

1. 如何创建（实例化）一个对象

MyClass mclass = new MyClass();

1. 如何创建（实例化）多个对象

MyClass mclass 1= new MyClass();

MyClass mclass2 = new MyClass();

1. 如何可以不让类被创建（实例化）,得不到其实例

public class PrivateClass

{

private PrivateClass(){}

}

1. 如何既可以不实例化类，又可以得到其实例

Public class PrivateClass()

{

Private PrivateClass();

Public static PrivateClass Instance()

{

Return new PrivateClass();

}

}

1. 但是5在每一次调用的时候都会得到一个新的实例，因为每次在类的内部都会New 一次，解决办法：

Public class PrivateClass()

{

Private static PrivateClass instance;

Private PrivateClass(){ };

Public static PrivateClass Instance()

{

Return new PrivateClass();

}

}

###### 使用

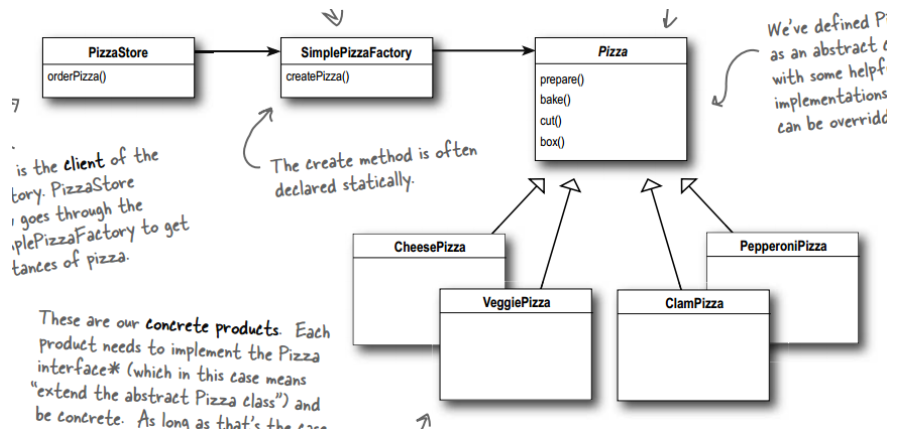
1. 要点
   1. **Singleton模式中的实例构造器可以设置为protected以允许子类派生。**
   2. **Singleton模式一般不要支持ICloneable接口，因为这可能会导致多个对象实例，与Singleton模式的初衷违背**
   3. **Singleton模式一般不要支持序列化，因为这也有可能导致多个对象实例，同样与Singleton模式的初衷违背。**
   4. **Singletom模式只考虑到了对象创建的工作，没有考虑对象销毁的工作。为什么这样做呢，因为Net平台是支持垃圾回收的，所以我们一般没有必要对其进行销毁处理。**
   5. **不能应对多线程环境：在多线程环境下，使用Singleton模式仍然有可能得到Singleton类的多个实例对象**。
2. 优点
3. 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
4. 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
5. 缺点
6. 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。
7. 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。
8. 对象的生存期：Singleton 不能解决删除单个对象的问题。因为它包含对该静态的私有字段的引用，静态字段是不能被CLR回收内存的，该实例会和应用程序生命周期一样长，一直存在。
9. 场景
10. 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。
11. 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。
12. .NET 实例
13. **HttpContext.Current就是一个单例，他们是通过Singleton的扩展方式实现的，他们的单例也并不是覆盖所有领域，只是针对某些局部领域中，是单例的，不同的领域中还是会有不同的实例。**

### 工厂模式Factory Method

##### 简单工厂

###### 概念

1. 简单工厂就是：我们去商店买开水瓶，商店就联系工厂将开水瓶做出来，给商店，商店再给买家，买开水瓶，商店就联系工厂做开水瓶，给商店，给买家
2. 概念图：



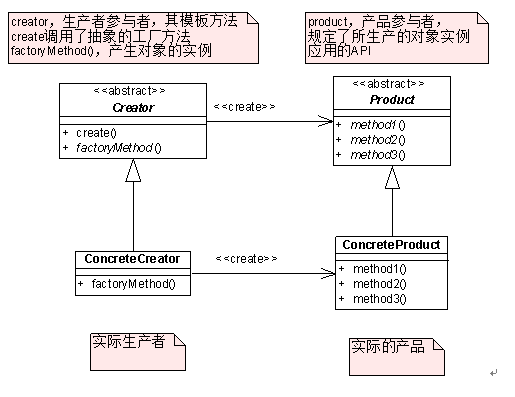
###### 使用

1. 何时使用
   1. 根据不同的参数获取不同的实例时。方法是让实际对象实例类继承自实例抽象类，根据调用方传入要获取实例的标识到工厂，工厂根据标识返回不同的真实实例。
2. 优点
3. 良好的封装性、代码结构清晰。如一个调用者想创建一个对象，只需知道其名称即可，降低了模块间的耦合
4. 工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对与客户端来说，去除了与具体产品的依赖。
5. 缺点
6. 每增加一个产品，就需要增加一个产品工厂的类，增加了系统的复杂度
7. 每增加一个产品，就需要修改工厂类，使其能够传递出增加的产品实例
8. 耦合严重，违反了DIP原则 和 开闭原则
9. 使用场景
10. 需要生成对象的地方
11. 需要灵活的、可扩展的框架时
12. 数据库访问，数据库可能变化时
13. 应用实例
14. 需要一辆汽车，直接从工厂里面提货，不用去管这辆车是怎么做出来的
15. Hibernate换数据库只需换方言和驱动即可
16. 简单计算器的实现
17. .NET 实例
18. NET中System.Text.Encoding类就实现了简单工厂模式，该类中的**GetEncoding(int codepage)**就是工厂方法

##### 工厂模式

###### 概念

1. 定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使得一个类的实例化延迟到子类。
2. 工厂二字想必大家都不陌生，工厂就是用来建造东西的，我们市面上买的东西比如水杯、玩具、汽车等等都是从工厂生产的，那我们需不需要知道它们是如何生产出来的呢？当然不需要，商家从工厂中直接提货，我们就可以购买了，完全不知道它是如何生产的，这就是工厂方法模式。
3. 在软件系统的构建过程中，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？如何提供一种“封装机制”来隔离出“这个易变对象”的变化，从而保持系统中“其他依赖对象的对象”不随着需求改变而改变？
4. UML



1. 组成：
   1. **抽象工厂角色（Creator）:**充当抽象工厂角色，定义工厂类所具有的基本的操作，任何具体工厂都必须继承该抽象类。
   2. **具体工厂角色（ConcreteCreator）**：充当具体工厂角色，该类必须继承抽象工厂角色，实现抽象工厂定义的方法，用来创建具体产品。
   3. **抽象产品角色（Product）：**充当抽象产品角色，定义了产品类型所有具有的基本操作，具体产品必须继承该抽象类。
   4. **具体产品角色（ConcreteProduct）**：充当具体产品角色，实现抽象产品类对定义的抽象方法，由具体工厂类创建，它们之间有一一对应的关系。

###### 使用

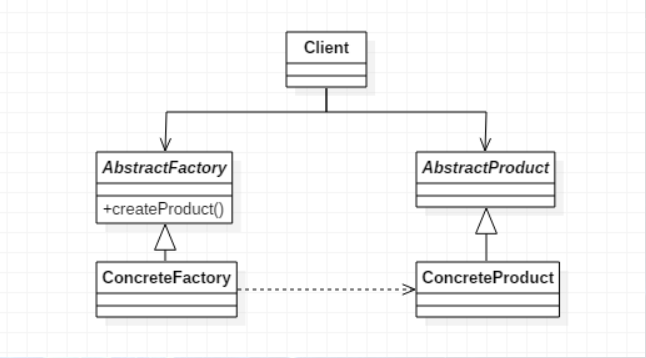
1. 何时使用
   1. 不同条件下创建不用实例时。方法是让子类实现工厂接口。
   2. Factory Method模式主要用于隔离类对象的使用者和具体类型之间的耦合关系。面对一个经常变化的具体类型，紧耦合关系会导致软件的脆弱。
   3. Factory Method模式通过面向对象的手法，将所要创建的具体对象工作延迟到子类，从而实现一种扩展（而非更改）的策略，较好地解决了这种紧耦合关系。
2. 优点
3. 良好的封装性、代码结构清晰。如一个调用者想创建一个对象，只需知道其名称即可，降低了模块间的耦合
4. 扩展性好，如果想增加一个产品，只需扩展一个工厂类即可
5. 屏蔽产品类。调用者只关心产品的接口
6. 典型的解耦框架
7. 缺点
8. 每增加一个产品，就需要增加一个产品工厂的类，增加了系统的复杂度
9. 使用场景
10. 需要生成对象的地方
11. 需要灵活的、可扩展的框架时
12. 数据库访问，数据库可能变化时
13. **Factory Method模式**解决“**单个对象**”的需求变化；
14. **AbstractFactory模式**解决“**系列对象**”的需求变化；
15. **Builder模式**解决“**对象部分**”的需求变化；
16. 应用实例
17. 需要一辆汽车，直接从工厂里面提货，不用去管这辆车是怎么做出来的
18. Hibernate换数据库只需换方言和驱动即可
19. 简单计算器的实现
20. .NET 实例
    1. 当我们请求一个\*.aspx的文件时，此时会映射到System.Web.UI.PageHandlerFactory类上进行处理，而对\*.ashx的请求将映射到System.Web.UI.SimpleHandlerFactory类中（这两个类都是继承于IHttpHandlerFactory接口的）

### 抽象工厂Abstract Factory

##### What

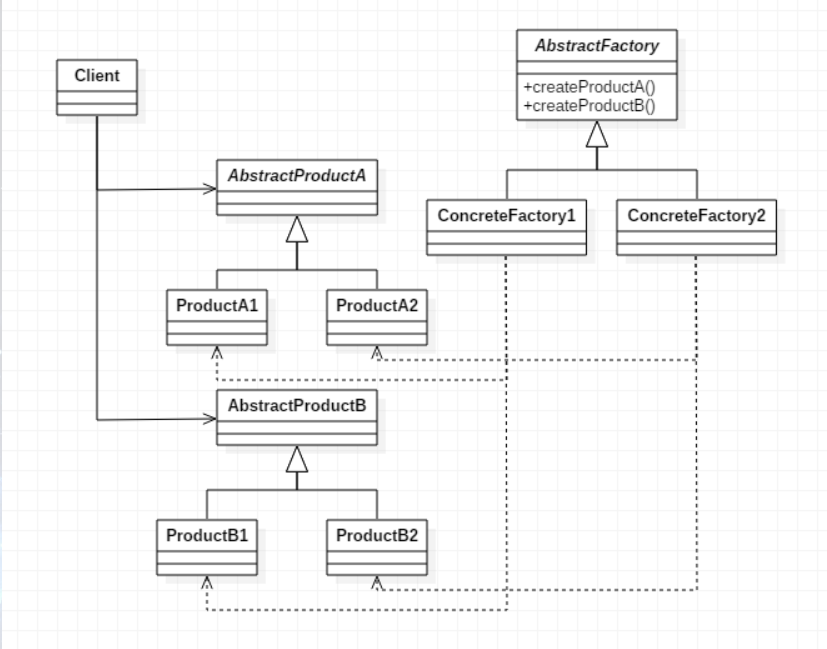
###### 概念

1. **抽象工厂模式（Abstract Factory）**，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
2. 抽象工厂模式其实就是多个[工厂方法模式](http://www.cnblogs.com/adamjwh/p/9033553.html" \t "_blank)，比如前面工厂方法模式中，我们创建多个不同类型的数据库，有MySQL、SQLServer等等，就是用工厂方法模式来实现的，但此时我们只能实现一个表（具体内容见下方工厂模式的实现），我们数据库中当然不可能只有一个表呀，所以抽象工厂模式就来了。
3. UML



其中，AbstractFactory是抽象工厂接口，里面包含所有的产品创建的抽象方法；ConcreteFactory则是具体的工厂，创建具有特定实现的产品对象；AbstractProduct是抽象产品，有可能由两种不同的实现；ConcreteProduct则是对于抽象产品的具体分类的实现。

1. 抽象工厂模式是工厂方法模式的升级版本，在有多个业务品种、业务分类时，通过抽象工厂模式产生需要的对象是一种非常好的解决方式。



1. 组成：
   1. **抽象产品类角色（AbstractProduct）**：为抽象工厂中相互依赖的每种产品定义抽象接口对象，也可以这样说，有几种产品，就要声明几个抽象角色，每一个抽象产品角色和一种具体的产品相匹配。
   2. **具体产品类（ConcreteProduct）**：具体产品类实现了抽象产品类，是针对某个具体产品的实现的类型。
   3. **抽象工厂类角色（Abstract Factory）**：定义了创建一组相互依赖的产品对象的接口操作，每种操作和每种产品一一对应。
   4. **具体工厂类角色（ConcreteFactory）**：实现抽象类里面的所有抽象接口操作，可以创建某系列具体的产品，这些具体的产品是“抽象产品类角色”的子类。

###### 使用

1. 何时使用
   1. 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AbstractFactory模式，这时候使用简单的静态工厂完全可以。
   2. "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“道路”与“房屋”的依赖，“道路”与“地道”的依赖。
   3. AbstractFactory模式主要在于应对“新系列”的需求变动。其缺点在于难以应对“新对象”的需求变动。
   4. AbstractFactory模式经常和FactoryMethod模式共同组合来应对“对象创建”的需求变化。
2. 优点
   1. 封装性，易于产品交换。由于具体工厂类在一个应用中只需在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，只需改变具体工程即可使用不同的产品配置。
   2. 创建实例与客户端分离
3. 缺点
   1. 很难支持增加新产品的变化，这是因为抽象工厂接口中已经确定了可以被创建的产品集合，如果需要添加新产品，此时就必须去修改抽象工厂的接口，这样就涉及到抽象工厂类的以及所有子类的改变，这样也就违背了“开发——封闭”原则。
4. 使用场景
   1. 如果系统需要多套的代码解决方案，并且每套的代码方案中又有很多相互关联的产品类型，并且在系统中我们可以相互替换的使用一套产品的时候可以使用该模式，客户端不需要依赖具体实现。
5. 应用实例
   1. 生成不同的操作系统的程序
   2. QQ 换皮肤，一整套一起换
   3. 更换数据库
6. .NET 实例
   1. 这个类就是System.Data.Common.DbProviderFactory，这个类位于System.Data.dll程序集中。该类扮演抽象工厂模式中抽象工厂的角色

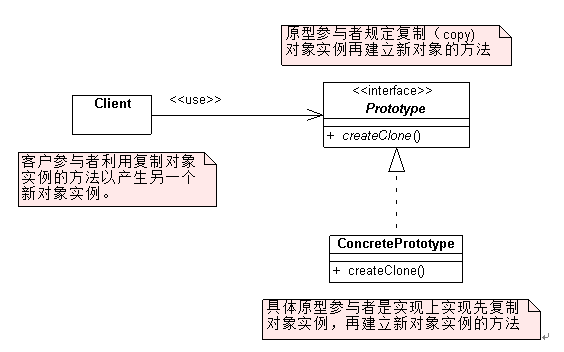
###### 分析

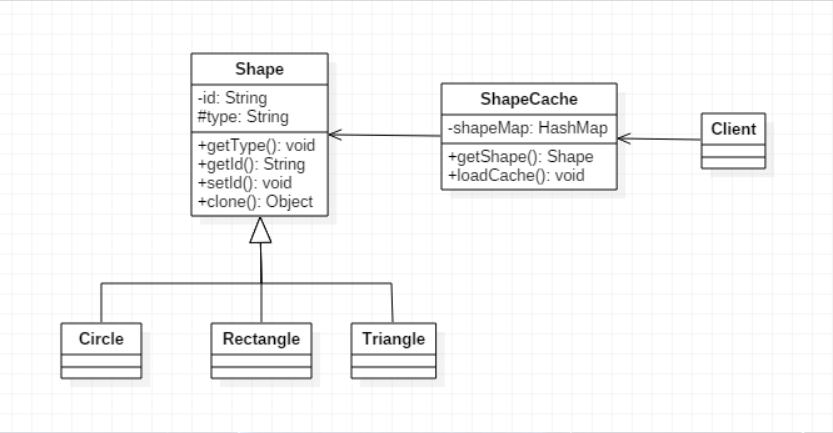
### 原型模式Prototype

##### What

###### 概念

1. **原型模式（Prototype）**，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
2. UML





Prototype为原型类，声明一个克隆自身的接口；ConcretePrototype为具体实现类，实现一个克隆自身的操作；而客户端Client只需让一个原型克隆自身，从而创建一个新的对象。

1. 组成
   1. **原型类（Prototype）**：原型类，声明一个Clone自身的接口；
   2. **具体原型类（ConcretePrototype）**：实现一个Clone自身的操作。
   3. 在原型模式中，Prototype通常提供一个包含Clone方法的接口，具体的原型ConcretePrototype使用Clone方法完成对象的创建。

###### 使用

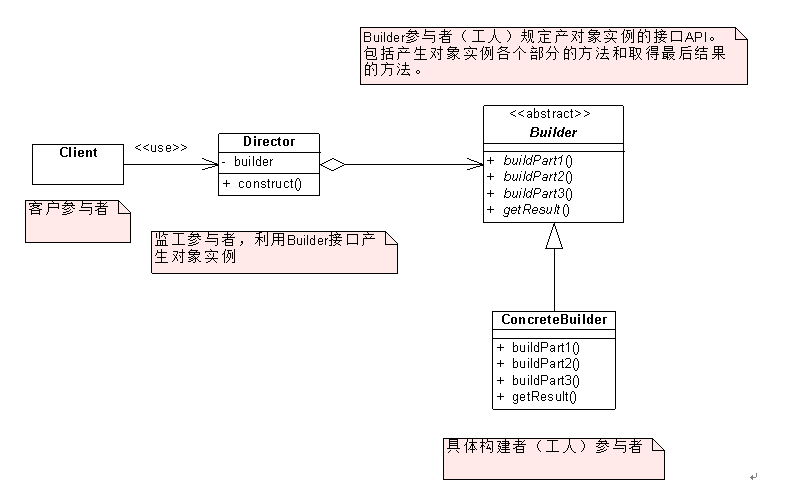
1. 何时使用
   1. 当创建一个类的实例的过程很昂贵或很复杂，并且我们需要创建多个这样类的实例时，如果我们用new操作符去创建这样的类实例，这就会增加创建类的复杂度和创建过程与客户代码复杂的耦合度。
   2. 当一个系统应该独立于它的产品创建、构成和表示时
   3. 当要实例化的类是在运行时刻指定时（如动态装载）
   4. 为了避免创建一个与产品类层次平行的工厂类层次时
   5. 当一个类的实例只能有几种几个不同状态组合中的一种时
   6. 如果采用工厂模式来创建这样的实例对象的话，随着产品类的不断增加，导致子类的数量不断增多，也导致了相应工厂类的增加，维护的代码维度增加了，因为有产品和工厂两个维度了，反而增加了系统复杂程度，所以在这里使用工厂模式来封装类创建过程并不合适。由于每个类实例都是相同的，这个相同指的是类型相同，但是每个实例的状态参数会有不同，如果状态数值也相同就没意义了，有一个这样的对象就可以了。当我们需要多个相同的类实例时，可以通过对原来对象拷贝一份来完成创建，这个思路正是原型模式的实现方式。
2. 优点
   1. 性能优良。原型模式向客户隐藏了创建新实例的复杂性
   2. 原型模式允许动态增加或较少产品类。
   3. 原型模式简化了实例的创建结构，工厂方法模式需要有一个与产品类等级结构相同的等级结构，而原型模式不需要这样。
   4. 逃避构造函数的约束
   5. 产品类不需要事先确定产品的等级结构，因为原型模式适用于任何的等级结构
3. 缺点
   1. 每个类必须配备一个克隆方法
   2. 配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。
4. 场景
   1. 资源优化场景。类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。
   2. 性能和安全要求的场景。通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。
   3. 一个对象多个修改者的场景。一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。
   4. 一般与工厂方法模式一起出现，通过clone方法创建一个对象，然后由工厂方法提供给调用者。
5. 应用实例
   1. 细胞分裂
   2. Java中的Object.Clone（）方法
   3. 复印
6. .NET 实例

### 建造者模式Builder

##### What

###### 概念

1. **建造者模式（Builder）**，将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示
2. 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却相对稳定。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？
3. 在现实生活中，我们经常会遇到一些构成比较复杂的物品，比如：电脑，它就是一个复杂的物品，它主要是由CPU、主板、硬盘、显卡、机箱等组装而成的。手机当然也是复杂物品，由主板，各种芯片，RAM 和ROM  摄像头之类的东西组成。但是无论是电脑还是手机，他们的组装过程是固定的，就拿手机来说，组装流水线是固定的，不变的，但是把不同的主板和其他组件组装在一起就会生产出不同型号的手机。那么在软件系统中是不是也会存在这样的对象呢？
4. UML



其中，Director为指挥者/导演类，负责安排已有模块的顺序，然后告诉Builder开始建造；Builder是抽象建造者，规范产品的组建，一般由子类实现；ConcreteBuilder是具体建造者，实现抽象类定义的所有方法，并且返回一个组建好的对象；Product是产品类，通常实现了模板方法模式。

1. 组成
   1. **抽象建造者角色（Builder）**：为创建一个Product对象的各个部件指定抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此角色规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。
   2. **具体建造者（ConcreteBuilder）**
      1. 实现Builder的接口以构造和装配该产品的各个部件。即实现抽象建造者角色Builder的方法。
      2. 定义并明确它所创建的表示，即针对不同的商业逻辑，具体化复杂对象的各部分的创建
      3. 提供一个检索产品的接口
      4. 构造一个使用Builder接口的对象即在指导者的调用下创建产品实例
   3. **指导者（Director）**：调用具体建造者角色以创建产品对象的各个部分。指导者并没有涉及具体产品类的信息，真正拥有具体产品的信息是具体建造者对象。它只负责保证对象各部分完整创建或按某种顺序创建。
   4. **产品角色（Product）**：建造中的复杂对象。它要包含那些定义组件的类，包括将这些组件装配成产品的接口。

###### 使用

1. 何时使用
   1. 一个基本部件不会变，而且组合经常变化的时候
2. 优点
   1. 封装性，是客户端不必知道产品内部的组成的细节。
   2. 具体的建造者类之间是相互独立的，容易扩展。
   3. 便于控制细节风险。由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响。
3. 缺点
   1. 产品必须有共同点，范围有限制
   2. 如果内部变化复杂，会有很多建造类
4. 使用场景
   1. 当创建复杂对象的算法应该独立于该对象的组成部分
   2. 相同的方法，不同的执行顺序，产生不同的事件结果时。
   3. 多个部件或者零件，都可以装配到一个对象中，但产生的结果又不相同时
   4. 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能。
   5. 创建一些复杂的对象时，这些对象的内部组成构件间的建造顺序是稳定的，但是对象的内部组成构件面临着复杂的变化。
5. 与工厂模式的区别
   1. 建造者模式更关注零件的装配顺序
6. 应用实例
   1. KFC的食品制作流程，原料多少克、加热几分钟等都有严格的规定，我们只需点餐即可，无论在哪里点的都是一样的。
   2. 去KFC吃汉堡、薯条、炸鸡等，这些单品是不变的，其组合是经常改变的，也就是所谓的“套餐”。
   3. Java中的StringBuilder/StringBuffer。
7. .NET 实例
   1. 该类型的名字就是System.Text.StringBuilder(存在mscorlib.dll程序集中)，它就是一个建造者模式的实现，从名称也可以看出来。不过它的实现属于建造者模式的演化，此时的建造者模式没有指挥者角色和抽象建造者角色，StringBuilder类即扮演着具体建造者的角色，也同时扮演了指挥者和抽象建造者的角色，StringBuilder类扮演着建造string对象的具体建造者角色，其中的ToString()方法用来返回具体产品给客户端（相当于上面代码中GetProduct方法）。其中Append方法用来创建产品的组件(相当于上面代码中BuildPartA和BuildPartB方法)，因为string对象中每个组件都是字符，所以也就不需要指挥者的角色的代码（指的是Construct方法,用来调用创建每个组件的方法来完成整个产品的组装），因为string字符串对象中每个组件都是一样的,都是字符,所以Append方法也充当了指挥者Construct方法的作用。

### 适配器模式Adapter

##### What

###### 动机（Motivate）

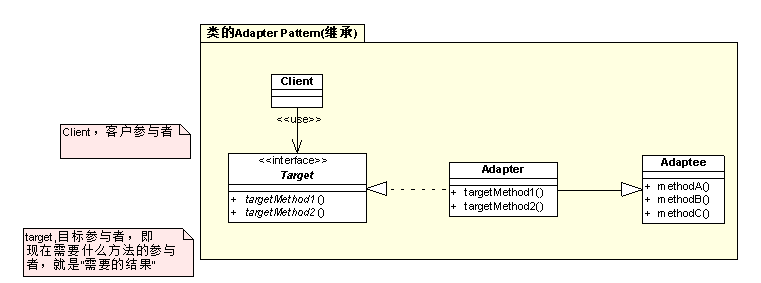
1. 在软件系统中，由于应用环境的变化，常常需要将“一些现存的对象”放在新的环境中应用，但是新环境要求的接口是这些现存对象所不满足的。如何应对这种“迁移的变化”？如何既能利用现有对象的良好实现，同时又能满足新的应用环境所要求的接口？
2. 适配器的功能就是介入“既有内容”和“需要的结果”之间，作为沟通桥梁。有时Adapter Pattern也称为Wrapper Pattern(包装模式)。

###### 意图（Intent）

1. 将一个类的接口转换成客户希望的另一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

###### 结构图（Structure）

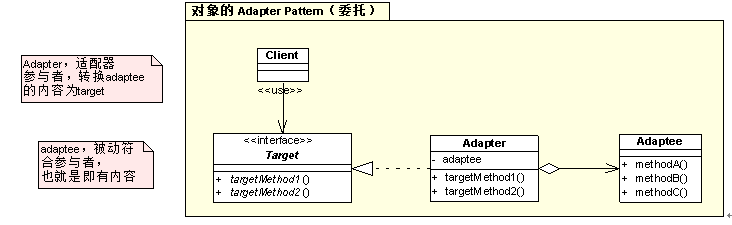
1. **对象适配器（更常用）**



对象适配器使用的是对象组合的方案，它的Adapter核Adaptee的关系是组合关系。

OO中优先使用组合模式，组合模式不适用再考虑继承。因为组合模式更加松耦合，而继承是紧耦合的，父类的任何改动都要导致子类的改动。

1. **类适配器**



###### 组成

1. **目标角色（Target）**：定义Client使用的与特定领域相关的接口。
2. **客户角色（Client）**：与符合Target接口的对象协同。
3. **被适配角色（Adaptee)**：定义一个已经存在并已经使用的接口，这个接口需要适配。
4. **适配器角色（Adapte)** ：适配器模式的核心。它将对被适配Adaptee角色已有的接口转换为目标角色Target匹配的接口。对Adaptee的接口与Target接口进行适配.

##### How

###### 实现方法

1. Adapter模式主要应用于“希望复用一些现存的类，但是接口又与复用环境要求不一致的情况”，在遗留代码复用、类库迁移等方面非常有用。
2. 对象适配器
   1. 对象适配器采用“对象组合”的方式，更符合松耦合精神，对适配的对象也没限制，可以一个，也可以多个，但是，使得重定义Adaptee的行为较困难，这就需要生成Adaptee的子类并且使得Adapter引用这个子类而不是引用Adaptee本身。
3. 类适配器
   1. 类适配器采用“多继承”的实现方式，在C#语言中，如果被适配角色是类，Target的实现只能是接口，因为C#语言只支持接口的多继承的特性。在C#语言中类适配器也很难支持适配多个对象的情况，同时也会带来了不良的高耦合和违反类的职责单一的原则，所以一般不推荐使用。
4. Adapter模式可以实现的非常灵活，不必拘泥于GoF23中定义的两种结构。例如，完全可以将Adapter模式中的“现存对象”作为新的接口方法参数，来达到适配的目的。
5. Adapter模式本身要求我们尽可能地使用“面向接口的编程”风格，这样才能在后期很方便地适配。

###### 实现要点

1. **优点：**
   1. **类的适配器模式：**
      1. 可以在不修改原有代码的基础上来复用现有类，很好地符合 “开闭原则”
      2. 可以重新定义Adaptee(被适配的类)的部分行为，因为在类适配器模式中，Adapter是Adaptee的子类
      3. 仅仅引入一个对象，并不需要额外的字段来引用Adaptee实例（这个即是优点也是缺点）。
   2. **对象的适配器模式：**
      1. 可以在不修改原有代码的基础上来复用现有类，很好地符合 “开闭原则”（这点是两种实现方式都具有的）
      2. 采用 “对象组合”的方式，更符合松耦合。
2. **缺点：**
   1. **类的适配器模式：**
      1. 用一个具体的Adapter类对Adaptee和Target进行匹配，当如果想要匹配一个类以及所有它的子类时，类的适配器模式就不能胜任了。因为类的适配器模式中没有引入Adaptee的实例，光调用this.SpecificRequest方法并不能去调用它对应子类的SpecificRequest方法。
      2. 采用了 “多继承”的实现方式，带来了不良的高耦合。
   2. **对象的适配器模式：**
      1. 使得重定义Adaptee的行为较困难，这就需要生成Adaptee的子类并且使得Adapter引用这个子类而不是引用Adaptee本身。
3. 场景：
   1. 系统需要复用现有类，而该类的接口不符合系统的需求
   2. 想要建立一个可重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。
   3. 对于对象适配器模式，在设计里需要改变多个已有子类的接口，如果使用类的适配器模式，就要针对每一个子类做一个适配器，而这不太实际。

##### Refer

###### .NET 场景：

* 1. System.IO里面的很多类都有适配器的影子，当我们操作文件的时候，其实里面调用了COM的接口实现
  2. 在.NET中复用COM对象：COM对象不符合.NET对象的接口，使用tlbimp.exe来创建一个Runtime Callable Wrapper（RCW）以使其符合.NET对象的接口，COM Interop就好像是COM和.NET之间的一座桥梁。
  3. .NET数据访问类（Adapter变体）：各种数据库并没有提供DataSet接口，使用DbDataAdapter可以将任何个数据库访问/存取适配到一个DataSet对象上，DbDataAdapter在数据库和DataSet之间做了很好的适配。当然还有SqlDataAdapter类型了，针对微软SqlServer类型的数据库在和DataSet之间进行适配。

##### Summary

每种设计模式都有自己的适用场景，它是为了解决一类问题，没有所谓的缺点，没有一种设计模式可以解决所有情况的。我们使用设计模式的态度是通过不断地重构来使用模式，不要一上来就使用设计模式，为了模式而模式。如果软件没有需求的变化，我们不使用模式都没有问题。遇到问题，我们就按着常规来写，有了需求变化，然后我们去抽象，了解使用的场景，然后在选择合适的设计模式。

### 桥接模式Bridge

##### What

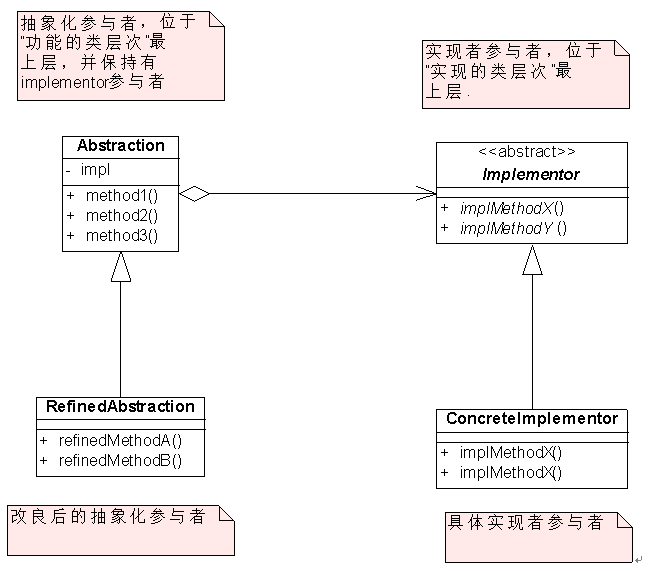
###### 动机（Motivate）

1. 在设计模式中的【桥模式】也有类似的概念，是连接了两个不同维度的东西，而且这两个维度又有强烈的变化，什么叫强烈呢，经常变化，什么是经常呢？哈哈，自己理解吧。
2. 在很多游戏场景中，会有这样的情况：【装备】本身会有的自己固有的逻辑，比如枪支，会有型号的问题，同时现在很多的游戏又在不同的介质平台上运行和使用，这样就使得游戏的【装备】具有了两个变化的维度——一个变化的维度为“平台的变化”，另一个变化的维度为“型号的变化”。如果我们要写代码实现这款游戏，难道我们针对每种平台都实现一套独立的【装备】吗？复用在哪里？如何应对这种“多维度的变化”？如何利用面向对象技术来使得【装备】可以轻松地沿着“平台”和“型号”两个方向变化，而不引入额外的复杂度？

###### 意图（Intent）

1. 将抽象部分与实现部分分离，使它们都可以独立地变化。
2. Bridge Pattern沟通着“功能的类层次”和“实现的类层次”
3. 桥模式不能只是认为是抽象和实现的分离，它其实并不仅限于此。其实两个都是抽象的部分，更确切的理解，应该是将一个事物中多个维度的变化分离。

###### 结构图（Structure）

1. 

###### 组成

1. 桥接模式的结构包括Abstraction、RefinedAbstraction、Implementor、ConcreteImplementorA和ConcreteImplementorB五个部分
2. **抽象化角色(Abstraction)**：抽象化给出的定义，并保存一个对**实现化对象（Implementor）**的引用。
3. **修正抽象化角色(Refined Abstraction)**：扩展抽象化角色，改变和修正父类对抽象化的定义。
4. **实现化角色(Implementor)**：这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非常不一样。实现化角色应当只给出底层操作，而抽象化角色应当只给出基于底层操作的更高一层的操作。
5. **具体实现化角色(Concrete Implementor)：**这个角色给出实现化角色接口的具体实现。
6. 在桥接模式中，两个类Abstraction和Implementor分别定义了抽象与行为类型的接口，通过调用两接口的子类实现抽象与行为的动态组合。

##### How

###### 实现方法

1. Abstraction
   1. 抽象接口的定义，即对过桥的对象进行抽象
   2. 通过组合方式引用平台接口，即桥梁
   3. 通过构造注入，初始化桥梁的实现
   4. 定义Operation方法
2. RefinedAbstraction
   1. 继承自Abstraction，具有Operation的真实实现，具体谁过桥
3. Implementor
   1. 抽象实现化对象（角色），即桥梁
4. ConcreteImplementor
   1. 实现或者定制自身桥梁的操作行为以达到过桥的意图，即过什么样的桥梁

###### 实现要点

1. 要点
   1. Bridge模式使用“对象间的组合关系”解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。
   2. 所谓抽象和实现沿着各自维度的变化，即“子类化”它们，得到各个子类之后，便可以任意组合它们，从而获得不同平台上的不同型号。
   3. Bridge模式有时候类似于多继承方案，但是多继承方案往往违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差。Bridge模式是比多继承方案更好的解决方法。
   4. Bridge模式的应用一般在“两个非常强的变化维度”，有时候即使有两个变化的维度，但是某个方向的变化维度并不剧烈——换言之两个变化不会导致纵横交错的结果，并不一定要使用Bridge模式。
2. 优点
   1. 把抽象接口与其实现解耦。
   2. 抽象和实现可以独立扩展，不会影响到对方。
   3. 实现细节对客户透明，对用于隐藏了具体实现细节。
3. 缺点
   1. 增加了系统的复杂度
4. 场景
   1. 如果一个系统需要在构件的抽象化角色和具体化角色之间添加更多的灵活性，避免在两个层次之间建立静态的联系。
   2. 设计要求实现化角色的任何改变不应当影响客户端，或者实现化角色的改变对客户端是完全透明的。
   3. 需要跨越多个平台的图形和窗口系统上。
   4. 一个类存在两个独立变化的维度，且两个维度都需要进行扩展。

##### Refer

###### .NET 场景：

##### Summary

桥接模式它是连接客户端代码和具体实现代码的一座桥梁，同时它也隔离了实现代码的改变对客户代码的影响。在【意图】中所说的抽象和实现，这两个部分其实都是高度抽象的，前面“抽象”是指定义的针对客户端的接口，客户端其实使用的是Abstract类型或者是RefinedAbstract类型，这两个类型只是接口，具体的实现委托给了Implementor类型了，Abstract类型子类化的扩展也演变成Implementor子类化的变化。我个人的理解，Abstract类型和其子类型在客户端代码和真正实现的代码之间起到了桥梁的作用，隔离了Implementor实现代码的变化，让客户端更稳定，所以【意图】才说是讲抽象部分和它的实现部分隔离。

### 装饰模式Decorator

##### What

###### 引言

【装饰模式】，英文名称：Decorator Pattern。我第一次看到这个名称想到的是另外一个词语“装修”，我就说说我对“装修”的理解吧，大家一定要看清楚，是“装修”,不是“装饰”。我们长大了，就要结婚，要结婚就涉及到要买房子，买的精装修或者简单装修就可以住的，暂时不谈。我们就谈谈我们购买的是毛坯房。如果我想要房子的内饰是大理石风格的，我们只要在毛坯房的基础之上用大理石风格的材料装修就可以，我们当然不可能为了要一个装修风格，就把刚刚盖好的房子拆了在重新来过。房子装修好了，我们就住了进来，很开心。过了段时间，我们发现我们的房子在冬季比较冷，于是我就想给我们的房子增加保暖的功能，装修好的房子我们可以继续居住，我们只是在房子外面加一层保护层就可以了。又过了一段时间，总是有陌生人光顾，所以我们想让房子更安全，于是我们在外墙和房顶加装安全摄像头，同时门窗也增加安全系统。随着时间的流逝，我们可能会根据我们的需求增加相应的功能，期间，我们的房子可以正常使用，加上什么设施就有了相应的功能。从这一方面来讲，“装修”和“装饰”有类似的概念

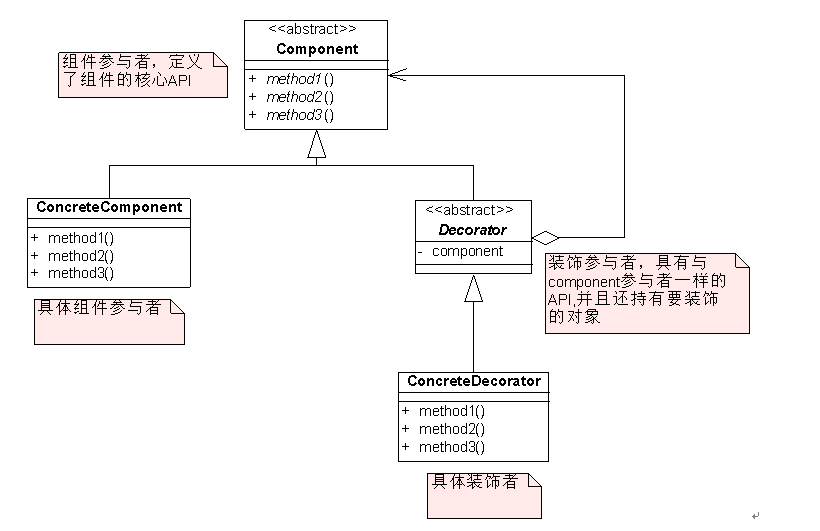
###### 动机（Motivate）

1. 在房子装修的过程中，各种功能可以相互组合，来增加房子的功用。类似的，如果我们在软件系统中，要给某个类型或者对象增加功能，如果使用“继承”的方案来写代码，就会出现子类暴涨的情况。比如：IMarbleStyle是大理石风格的一个功能，IKeepWarm是保温的一个接口定义，IHouseSecurity是房子安全的一个接口，就三个接口来说，House是我们房子，我们的房子要什么功能就实现什么接口，如果房子要的是复合功能，接口不同的组合就有不同的结果，这样就导致我们子类膨胀严重，如果需要在增加功能，子类会成指数增长。
2. 这个问题的根源在于我们“过度地使用了继承来扩展对象的功能”，由于继承为类型引入的静态特质（所谓静态特质，就是说如果想要某种功能，我们必须在编译的时候就要定义这个类，这也是强类型语言的特点。静态，就是指在编译的时候要确定的东西；动态，是指运行时确定的东西），使得这种扩展方式缺乏灵活性；并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能的组合）会导致更多子类的膨胀（多继承）。
3. 如何使“对象功能的扩展”能够根据需要来动态（即运行时）地实现？同时避免“扩展功能的增多”带来的子类膨胀问题？从而使得任何“功能扩展变化”所导致的影响降为最低？

###### 意图（Intent）

1. **装饰模式（Decorator）**，动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更灵活
2. 在核心对象的基础上，加上层层装饰，使之符合要求。并且装饰与核心组件一视同仁，具有相同的API。

###### 结构图（Structure）



###### 组成

1. **抽象构件角色（Component）**：给出一个抽象接口，以规范准备接收附加责任的对象。
2. **具体构件角色（Concrete Component）**：定义一个将要接收附加责任的类。
3. **装饰角色（Decorator）**：持有一个构件（Component）对象的实例，并实现一个与抽象构件接口一致的接口。
4. **具体装饰角色（Concrete Decorator）：**负责给构件对象添加上附加的责任。

##### How

###### 实现方法

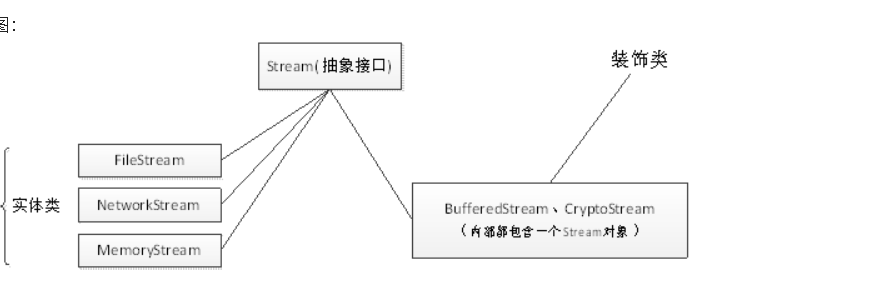
1. **Component**
   1. 定义抽象装饰部件，部件必须要符合一定的规范（中式装修规范，你不可能加一个壁炉）
2. **Concrete Component**
   1. 定义具体的要装饰部件，实现装饰规范和自己的特性（打造中式家具）
3. **Decorator**
   1. 定义抽象装饰行为，行为也要符合一定的规范（插入墙内还是可以自由摆放）
   2. 注入要装饰的目标，按照装修标准进行装修
4. **Concrete Decorator**
   1. 实现具体的装饰行为（是打造家具还是挂一个壁炉）

###### 实现要点

1. 要点
   1. 通过采用组合、而非继承的手法，Decorator模式实现了在运行时动态地扩展对象功能的能力，而且可以根据需要扩展多个功能。避免了单独使用继承带来的“灵活性差”和“多子类衍生问题”。
   2. Component类在Decorator模式中充当抽象接口的角色，不应该去实现具体的行为。而且Decorator类对于Component类应该透明——换言之Component类无需知道Decorator类，Decorator类是从外部来扩展Component类的功能。
   3. Decorator类在接口上表现为is-a Component的继承关系，即Decorator类继承了Component类所具有的接口。但在实现上又表现为has-a Component的组合关系，即Decorator类又使用了另外一个Component类。我们可以使用一个或者多个Decorator对象来“装饰”一个Component对象，且装饰后的对象仍然是一个Component对象。
   4. Decorator模式并非解决“多子类衍生的多继承”问题，Decorator模式应用的要点在于解决“主体类在多个方向上的扩展功能”——是为“装饰”的含义。
2. 优点
   1. 把抽象接口与其实现解耦。
   2. 抽象和实现可以独立扩展，不会影响到对方。
   3. 实现细节对客户透明，对用于隐藏了具体实现细节。
3. 缺点
   1. 增加了系统的复杂度。
4. 场景
   1. 如果一个系统需要在构件的抽象化角色和具体化角色之间添加更多的灵活性，避免在两个层次之间建立静态的联系。
   2. 设计要求实现化角色的任何改变不应当影响客户端，或者实现化角色的改变对客户端是完全透明的。
   3. 需要跨越多个平台的图形和窗口系统上。
   4. 一个类存在两个独立变化的维度，且两个维度都需要进行扩展。

##### Refer

###### .NET 场景：

1. 在Net框架中，有一个类型很明显的使用了“装饰模式”，这个类型就是Stream。Stream类型是一个抽象接口，它在System.IO命名空间里面，它其实就是Component。FileStream、NetworkStream、MemoryStream都是实体类ConcreteComponent。
2. 右边的BufferedStream、CryptoStream是装饰对象，它们都是继承了Stream接口的。
3. 
4. Stream就相当于Component，定义装饰的对象，FileStream就是要装饰的对象，BufferedStream是装饰对象。

##### Summary

对这个模式的看法，这个模式有点像包饺子，ConcreteComponent其实是饺子馅，Decorator就像饺子皮一样，包什么皮就有什么的样子，皮和皮也可以嵌套，当然我们生活中的饺子只是包一层。其实手机也是一个装饰模式使用的好例子，以前我们的手机只是接打电话，然后可以发短信和彩信，我在装饰一个就可以拍照了。我们现在的手机功能很丰富，其结果也类似装饰的结果。随着社会的进步，技术发展，模块化的手机也出现了，其设计原理也和“装饰模式”就更接近了。不光手机，还有我们身边其他很多家用电器也有类似的发展经历

### 组合模式Composite

##### What

###### 引言

【组合模式】，英文名称是：Composite Pattern。当我们谈到这个模式的时候，有一个物件和这个模式很像，也符合这个模式要表达的意思，那就是“俄罗斯套娃”。“俄罗斯套娃”就是大的瓷器娃娃里面装着一个小的瓷器娃娃，小的瓷器娃娃里面再装着更小的瓷器娃娃，直到最后一个不能再装更小的瓷器娃娃的那个瓷器娃娃为止（有点绕，下面我会配图，一看就明白）。在我们的操作系统中有文件夹的概念，文件夹可以包含文件夹，可以嵌套多层，最里面包含的是文件，这个概念和“俄罗斯套娃”很像。当然还有很多的例子，例如我们使用系统的时候，会使用到“系统菜单”，这个东西是树形结构。这些例子包含的这些东西或者说是对象，可以分为两类，一类是：容器对象，可以包含其他的子对象；另一类是：叶子对象，这类对象是不能在包含其他对象的对象了。在软件设计中，我们该怎么处理这种情况呢？是每类对象分别对待，还是提供一个统一的操作方式呢。

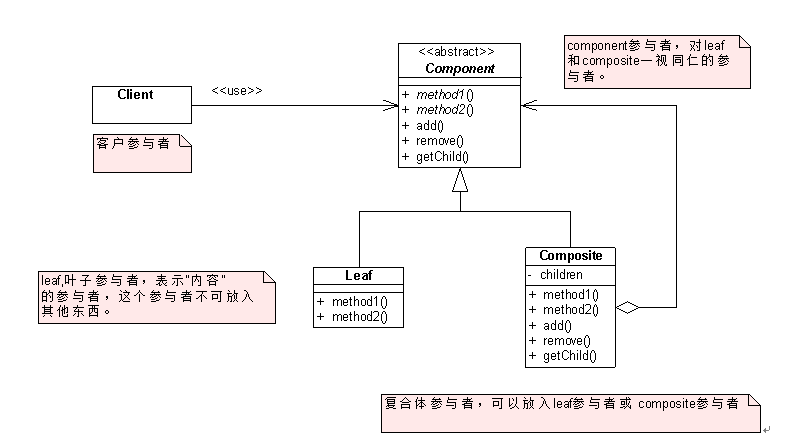
###### 动机（Motivate）

1. 客户代码过多地依赖于对象容器（对象容器是对象的容器，细细评味）复杂的内部实现结构，对象容器内部实现结构（而非抽象接口）的变化将引起客户代码的频繁变化，带来了代码的维护性、扩展性等方面的弊端。如何将“客户代码与复杂的对象容器结构”解耦？如何让对象容器自己来实现自身的复杂结构，从而使得客户代码就像处理简单对象一样来处理复杂的对象容器？
2. 像把目录和文件合起来看成为一个目录进入点一样，有时候把容器和内容当作同类处理会比较好下手。容器里面可以是内容，也可以是更小一号的容器；而这个小一号的容器里还可以再放更小一号的容器，可以建立出像这样大套小的结构和递归结构。

###### 意图（Intent）

1. 将对象组合成树形结构以表示“部分-整体”的层次结构。Composite使得用户对单个对象和组合对象的使用具有一致性。

###### 结构图（Structure）



###### 组成

1. **抽象构件角色（Component）**：这是一个抽象角色，它给参加组合的对象定义出了公共的接口及默认行为，可以用来管理所有的子对象（在透明式的组合模式是这样的）。在安全式的组合模式里，构件角色并不定义出管理子对象的方法，这一定义由树枝结构对象给出。
2. **树叶构件角色（Leaf）**：树叶对象是没有下级子对象的对象，定义出参加组合的原始对象的行为。（原始对象的行为可以理解为没有容器对象管理子对象的方法，或者 【原始对象行为】+【管理子对象的行为（Add，Remove等）】=面对客户代码的接口行为集合）
3. **树枝构件角色（Composite）**：代表参加组合的有下级子对象的对象，树枝对象给出所有管理子对象的方法实现，如Add、Remove等。

##### How

###### 实现方法

1. 组合模式实现的最关键的地方是——简单对象和复合对象必须实现相同的接口。这就是组合模式能够将组合对象和简单对象进行一致处理的原因
2. **透明式的组合模式**
   1. 指“抽象构件角色”定义的接口行为集合包含两个部分，一部分是叶子对象本身所包含的行为（比如Operation），另外一部分是容器对象本身所包含的管理子对象的行为(Add,Remove)。
   2. 这个抽象构件必须同时包含这两类对象所有的行为，客户端代码才会透明的使用，无论调用容器对象还是叶子对象，接口方法都是一样的，这就是透明
   3. 针对客户端代码的透明，但是也有他自己的问题，叶子对象不会包含自己的子对象，为什么要有Add,Remove等类似方法呢，调用叶子对象这样的方法可能（注意：我这里说的是可能，因为有些人会把这些方法实现为空，不做任何动作，当然也不会有异常抛出了，不要抬杠）会抛出异常，这样就不安全了，然后人们就提出了“安全式的组合模式”。
3. **安全式的组合模式**
   1. 所谓安全式是指“抽象构件角色”只定义叶子对象的方法，确切的说这个抽象构件只定义两类对象共有的行为，然后容器对象的方法定义在“树枝构件角色”上，这样叶子对象有叶子对象的方法，容器对象有容器对象的方法，这样责任很明确，当然调用肯定不会抛出异常了。

###### 实现要点

1. 要点
   1. Composite模式采用树形结构来实现普遍存在的对象容器，从而将“一对多”的关系转化为“一对一”的关系，使得客户代码可以一致地处理对象和对象容器，无需关心处理的是单个的对象，还是组合的对象容器。
   2. 将“客户代码与复杂的对象容器结构”解耦是Composite模式的核心思想，解耦之后，客户代码将与纯粹的抽象接口——而非对象容器的复杂内部实现结构——发生依赖关系，从而更能“应对变化”。
   3. Composite模式中，是将“Add和Remove等和对象容器相关的方法”定义在“表示抽象对象的Component类”中，还是将其定义在“表示对象容器的Composite类”中，是一个关乎“透明性”和“安全性”的两难问题，需要仔细权衡。这里有可能违背面向对象的“单一职责原则”，但是对于这种特殊结构，这又是必须付出的代价。ASP.Net控件的实现在这方面为我们提供了一个很好的示范。
   4. Composite模式在具体实现中，可以让父对象中的子对象反向追朔；如果父对象有频繁的遍历需求，可使用缓存技巧来改善效率。
2. 优点
   1. 组合模式使得客户端代码可以一致地处理对象和对象容器，无需关系处理的单个对象，还是组合的对象容器。
   2. 将”客户代码与复杂的对象容器结构“解耦。
   3. 可以更容易地往组合对象中加入新的构件。
3. 缺点
   1. 使得设计更加复杂。客户端需要花更多时间理清类之间的层次关系。（这个是几乎所有设计模式所面临的问题）。
4. 场景
   1. 需要表示一个对象整体或部分的层次结构。
   2. 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

##### Refer

###### .NET 场景：

1. 其实组合模式在FCL里面运用还是很多的，不知道大家是不是有所感觉，这个模式大多数是运用在控件上或者是和界面操作、展示相关的操作上。这个模式在.NET 中最典型的应用就是应用与WinForms和Web的开发中，在.NET类库中，都为这两个平台提供了很多现有的控件，然而System.Windows.Forms.dll中System.Windows.Forms.Control类就应用了组合模式，因为控件包括Label、TextBox等这样的简单控件，这些控件可以理解为叶子对象，同时也包括GroupBox、DataGrid这样复合的控件或者叫容器控件，每个控件都需要调用OnPaint方法来进行控件显示，为了表示这种对象之间整体与部分的层次结构，微软把Control类的实现应用了组合模式（确切地说应用了透明式的组合模式）。

##### Summary

### 外观模式Facade

##### What

###### 引言

【外观模式】，英文名称是：Facade Pattern。我们先从名字上来理解一下“外观模式”。我看到了“外观”这个词语，就想到了“外表”这个词语，两者有着很相近的意思。就拿谈恋爱来说，“外表”很重要，如果第一眼看着很舒服、有眼缘，那就有交往下去的可能。如果长的“三寸钉、枯树皮”，估计就够呛了。在这方面，“外观”和“外表”有着相同的作用。在软件系统中，要完成一个功能，需要很多接口调用，不仅增加了开发难度，也增加了调试成本和维护的复杂度。不如我们把这些接口再封装一次，给一个很好的“外观”，让使用者使用更方便，只需调用一个接口，就可以完成以前调用多个接口的来完成任务，这就方便了。这个模式很简单，大家很容易理解，可能大家在编码的过程中已经不止一次使用过该模式了，只是不知道名字罢了。

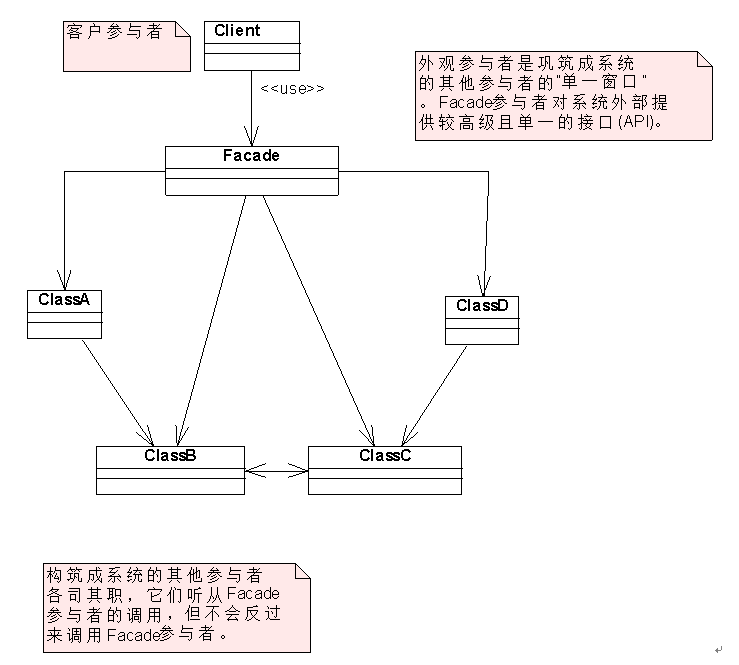
###### 动机（Motivate）

1. 在软件系统开发的过程中，当组件的客户（即外部接口，或客户程序）和组件中各种复杂的子系统有了过多的耦合，随着外部客户程序和各子系统的演化，这种过多的耦合面临很多变化的挑战。如何简化外部客户程序和系统间的交互接口？如何将外部客户程序的演化和内部子系统的变化之间的依赖相互解耦？
2. 程序这个东西往往越做越大，许多类彼此间的影响让关系更加错综复杂。因此在使用类时，要确实了解类之间的关系，正确依次调用方法。也就是说，利用大型程序进行数据处理时，必须精确控制相关的类，既然如此，就干脆设个处理专用的“窗口”，如此一来，不需要个别控制类，只要把要求留给“窗口”即可。这个“窗口”就是Façade Pattern。

###### 意图（Intent）

1. **装饰模式（Decorator）**，动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更灵活。
2. 为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
3. **Façade Pattern能整理错综复杂的来龙去脉，提供较高级的接口（API）。Façade参与者让系统外部看到较简单的接口（API），而且Façade参与者还会兼顾系统内部各类功能和互动关系，以最正确的顺序利用类。**

###### 结构图（Structure）



###### 组成

1. **外观角色（Facade）**：在客户端可以调用它的方法，在外观角色中可以知道相关的（一个或者多个）子系统的功能和责任；在正常情况下，它将所有从客户端发来的请求委派到相应的子系统去，传递给相应的子系统对象处理。
2. **子系统角色（SubSystem）**：在软件系统中可以有一个或者多个子系统角色，每一个子系统可以不是一个单独的类，而是一个类的集合，它实现子系统的功能；每一个子系统都可以被客户端直接调用，或者被外观角色调用，它处理由外观类传过来的请求；子系统并不知道外观的存在，对于子系统而言，外观角色仅仅是另外一个客户端而已。

##### How

###### 实现方法

1. **定义操作类 ClassA**
2. **定义操作类 ClassB**
3. **定义外观类 Façade**
4. **在客户端中实例化Façade ，然后用实例调用各自操作类的方法**

###### 实现要点

1. 要点
   1. 一个系统可以有几个门面类
      1. 在门面模式中，通常只需要一个门面类，并且此门面类只有一个实例，换言之它是一个单例类。当然这并不意味着在整个系统里只有一个门面类，而仅仅是说对每一个子系统只有一个门面类。或者说，如果一个系统有好几个子系统的话，每一个子系统都有一个门面类，整个系统可以有数个门面类。
   2. 为子系统增加新行为
      1. 初学者往往以为通过继承一个门面类便可在子系统中加入新的行为，这是错误的。门面模式的用意是为子系统提供一个集中化和简化的沟通管道，而不能向子系统加入新的行为。比如医院中的接待员并不是医护人员，接待员并不能为病人提供医疗服务。
   3. Facade有助于建立层次结构的系统，实现了子系统与客户之间的松耦合关系，子系统内部的功能组件往往是紧耦合的。松耦合关系使得子系统的组件变化不会影响到它的客户。Facade消除了复杂的循环依赖关系。这一点在客户程序与子系统分别实现的时候格外重要。
   4. 从客户程序的角度来看，Facade模式不仅简化了整个组件系统的接口，同时对于组件内部与外部客户程序来说，从某种程度上也达到了一种“解耦”的效果——内部子系统的任何变化不会影响到Facade接口的变化。
2. 优点
   1. 外观模式对客户屏蔽了子系统组件，从而简化了接口，减少了客户处理的对象数目并使子系统的使用更加简单。
   2. 外观模式实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件是紧耦合的。松耦合使得子系统的组件变化不会影响到它的客户。
3. 缺点
   1. 如果增加新的子系统可能需要修改外观类或客户端的源代码，这样就违背了”开——闭原则“（不过这点也是不可避免）。
4. 场景
   1. 外一个复杂的子系统提供一个简单的接口
   2. 提供子系统的独立性
   3. 在层次化结构中，可以使用外观模式定义系统中每一层的入口。其中三层架构就是这样的一个例子。

##### Refer

###### .NET 场景：

1. 外观模式在FCL里面运用还是很多的，多数情况是单个类的情况，在Asp.Net里面，有很多复合控件，比如：Login控件，可以登录，可以认证，可以保存登录用户信息。其实，外观模式更多的是应用在业务系统当中，效果更好。

##### Summary

Facade设计模式更注重从架构的层次去看整个系统，而不是单个类的层次。Facade很多时候更是一种架构设计模式。注意区分Facade模式、Adapter模式、Bridge模式与Decorator模式：

* + - 1. Facade模式注重简化接口
      2. Adapter模式注重转换接口
      3. Bridge模式注重分离接口（抽象）与其实现
      4. Decorator模式注重稳定接口的前提下为对象扩展功能

### 享元模式Flyweight

##### What

###### 引言

【享元模式】，英文名称是：Flyweight Pattern。还是老套路，先从名字上来看看。“享元”是不是可以这样理解，共享“单元”，单元是什么呢，举例说明，对于图形而言就是图元，对于英文来说就只26个英文字母，对于汉语来说就是每个汉字，也可以这样理解“元”，构成事物的最小单元，这些单元如果大量、且重复出现，可以缓存重复出现的单元，达到节省内存的目的，换句说法就是享元是为了节省空间，对于计算机而言就是内存。面向对象很好地解决了系统抽象性的问题（系统抽象性指把系统里面的事物写成类，类可以实例化成为对象，用对象和对象之间的关系来设计系统），在大多数情况下，这样做是不会损及系统的性能的。但是，在某些特殊的应用中，由于对象的数量太大，并且这些大量的对象中有很多是重复的，如果每个对象都单独的创建（C#的语法是new）出来，会给系统带来难以承受的内存开销。比如图形应用中的图元等对象、字处理应用中的字符对象等。

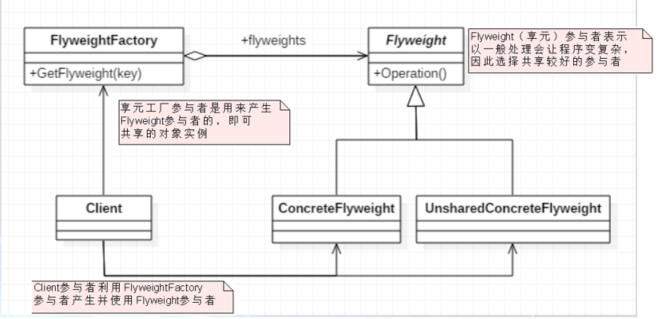
###### 动机（Motivate）

1. 在软件系统中，采用纯粹对象方案的问题在于大量细粒度的对象会很快充斥在系统中，从而带来很高的运行时代价——主要指内存需求方面的代价。如何在避免大量细粒度对象问题的同时，让外部客户程序仍然能够透明地使用面向对象的方式来进行操作？
2. flyweight是“蝇量级”的意思，它是拳击赛中体重最轻的等级。这个Design Patterns就是用来“减轻”对象。（说明：计算机世界里的对象是虚拟的，其“重”与“轻”是用来描述“内存使用量”的相对概念）。
3. Flyweight Pattern的基本理念是：尽量共享对象实例，不做无谓的new，也就是说：不是一需要对象就马上去new，如果可以利用其他现有的对象实例，就让它们共享吧，这就是Flyweight Pattern的核心概念。

###### 意图（Intent）

1. 运用共享技术有效地支持大量细粒度的对象。

###### 结构图（Structure）



###### 组成

1. **抽象享元角色（Flyweight）**:此角色是所有的具体享元类的基类，为这些类规定出需要实现的公共接口。那些需要外部状态的操作可以通过调用方法以参数形式传入。
2. **具体享元角色（ConcreteFlyweight）**：实现抽象享元角色所规定的接口。如果有内部状态的话，可以在类内部定义。
3. **享元工厂角色（FlyweightFactory）**：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享，当一个客户端对象调用一个享元对象的时候，享元工厂角色检查系统中是否已经有一个符合要求的享元对象，如果已经存在，享元工厂角色就提供已存在的享元对象，如果系统中没有一个符合的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。
4. **客户端角色（Client）**：本角色需要存储所有享元对象的外部状态。

##### How

###### 实现方法

1. **定义抽象的享元类 Flyweight**
2. **定义真正的享元类 ConcreteFlyweight**
3. **定义享元的工厂角色类**
   1. **在类中定义静态的享元类集合**
   2. **定义GetFlyweight(key)方法中，从集合类中获取已存在的角色，读取出来**

###### 实现要点

1. 要点
   1. 面向对象很好地解决了抽象性的问题，但是作为一个运行在机器中的程序实体，我们需要考虑对象的代价问题。Flyweight设计模式主要解决面向对象的代价问题，一般不触及面向对象的抽象性问题。
   2. Flyweight采用对象共享的做法来降低系统中对象的个数，从而降低细粒度对象给系统带来的内存压力。在具体实现方面，要注意对象状态的处理。
   3. 对象的数量太大从而导致对象内存开销加大——什么样的数量才算大？这需要我们仔细的根据具体应用情况进行评估，而不能凭空臆断。
2. 优点
   1. 享元模式的优点在于它能够极大的减少系统中对象的个数。
   2. 享元模式由于使用了外部状态，外部状态相对独立，不会影响到内部状态，所以享元模式使得享元对象能够在不同的环境被共享。
3. 缺点
   1. 由于享元模式需要区分外部状态和内部状态，使得应用程序在某种程度上来说更加复杂化了。
   2. 为了使对象可以共享，享元模式需要将享元对象的状态外部化，而读取外部状态使得运行时间变
4. 场景
   1. **在下面所有条件都满足时，可以考虑使用享元模式：**
      1. 一个系统中有大量的对象；
      2. 这些对象耗费大量的内存；
      3. 这些对象中的状态大部分都可以被外部化
      4. 这些对象可以按照内部状态分成很多的组，当把外部对象从对象中剔除时，每一个组都可以仅用一个对象代替软件系统不依赖这些对象的身份，
   2. 满足上面的条件的系统可以使用享元模式。但是使用享元模式需要额外维护一个记录子系统已有的所有享元的表，而这也需要耗费资源，所以，应当在有足够多的享元实例可共享时才值得使用享元模式。

##### Refer

###### .NET 场景：

1. .NET在C#中有一个Code Behind机制，它表面有一个aspx文件，背后又有一个cs文件，它的编译过程实际上会把aspx文件解析成C#文件，然后编译成dll，在这个过程中，我们在aspx中写的任何html代码都会转化为literal control，literal control是一个一般的文本控件，它就表示html标记。当这些标记有一样的时候，构建控件树的时候就会用到Flyweight模式.

##### Summary

任何设计模式都有他特定的使用场景，小心误用。这个模式在业务系统中相对而言使用的并不多，在类似游戏场景中、字符处理等系统用的比较多。还是老话，通过迭代来使用模式，别为了模式而模式。

### 代理模式Proxy

##### What

###### 引言

【代理模式】，英文名称是：Proxy Pattern。还是老套路，先从名字上来看看。“代理”可以理解为“代替”，代替“主人”做一些事情，为什么需要“代理”，是因为某些原因（比如：安全方面的原因），不想让“主人”直接面对这些繁琐、复杂的问题，但是这些事情是经“主人”同意或者授意的，如同“主人”亲自完成的一样。这个模式很简单，生活中的例子也很多。举例说明，歌星、影星的经纪人就是现实生活中一个代理模式的很好例子，还有操作系统中的防火墙，也是代理的例子，要访问系统，先过防火墙这关，否则免谈。

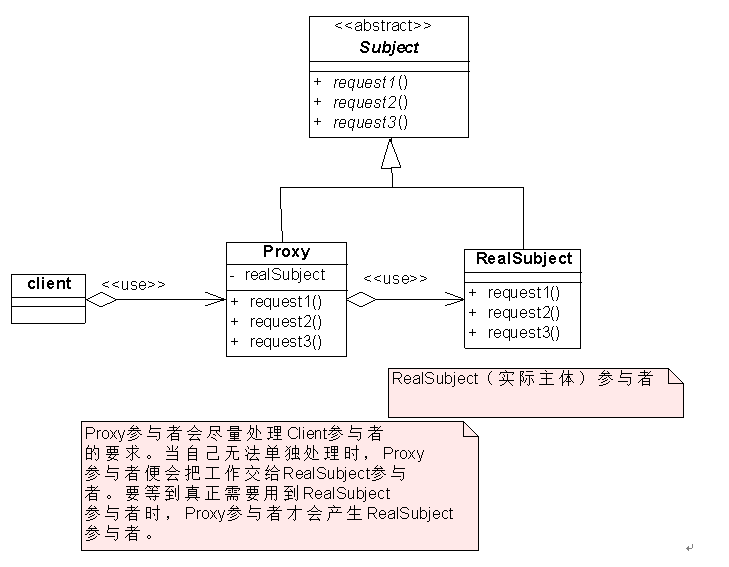
###### 动机（Motivate）

1. 在面向对象系统中，有些对象由于某种原因（比如对象创建的开销很大，或者某些操作需要安全控制，或者需要进程外的访问等），直接访问会给使用者、或者系统结构带来很多麻烦。如何在不失去透明操作对象的同时来管理/控制这些对象特有的复杂性？增加一层间接层是软件开发中常见的解决方式。

###### 意图（Intent）

1. 为其他对象提供一种代理以控制对这个对象的访问。
2. 用轻量级对象代替重量级对象工作，只有真正需要重量级对象时，才创建之。

###### 结构图（Structure）



###### 组成

1. **抽象主题角色（Subject）**：声明了真实主题和代理主题的公共接口，这样一来在使用真实主题的任何地方都可以使用代理主题。
2. **代理主题角色（Proxy）**：代理主题角色内部含有对真实主题的引用，从而可以操作真实主题对象；代理主题角色负责在需要的时候创建真实主题对象；代理角色通常在将客户端调用传递到真实主题之前或之后，都要执行一些其他的操作，而不是单纯地将调用传递给真实主题对象。
3. **真实主题角色（RealSubject）**：定义了代理角色所代表的真实对象。
4. **分类：**
   1. 远程（Remote）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。这个不同的地址空间可以是本电脑中，也可以在另一台电脑中。最典型的例子就是——客户端调用Web服务或WCF服务。
   2. 虚拟（Virtual）代理：根据需要创建一个资源消耗较大的对象，使得对象只在需要时才会被真正创建。
   3. Copy-on-Write代理：虚拟代理的一种，把复制（或者叫克隆）拖延到只有在客户端需要时，才真正采取行动。
   4. 保护（Protect or Access）代理：控制一个对象的访问，可以给不同的用户提供不同级别的使用权限。
   5. 防火墙（Firewall）代理：保护目标不让恶意用户接近。
   6. 智能引用（Smart Reference）代理：当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等。
   7. Cache代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以这些结果。

在WCF或者WebService的开发过程中，我们在客户端添加服务引用的时候，在客户程序中会添加一些额外的类，在客户端生成的类扮演着代理主题角色，我们客户端也是直接调用这些代理角色来访问远程服务提供的操作。这个是远程代理的一个典型例子。

##### How

###### 实现方法

1. **定义Subject角色，定义代理角色和真实主体角色共有的接口方法**
2. **定义RealSubject，实现RealSubject的接口方法实现**
3. **定义 代理对象 Proxy**
   1. **传入要代理的真实对象，即RealSubject**
   2. **实现代理对象的方法**

###### 实现要点

1. 要点
   1. “增加一层间接层”是软件系统中对许多复杂问题的一种常见解决方法。在面向对象系统中，直接使用某些对象会来带很多问题，作为间接层的Proxy对象便是解决这一问题的常用手段。具体Proxy设计模式的实现方法、实现粒度都相差很大，有些可能对单个对象做细粒度的控制，如copy-on-write技术，有些可能对组件模块提供抽象代理层，在架构层次对对象做Proxy
2. 优点
   1. 代理模式能够将调用用于真正被调用的对象隔离，在一定程度上降低了系统的耦合度；
   2. 代理对象在客户端和目标对象之间起到一个中介的作用，这样可以起到对目标对象的保护。代理对象可以在对目标对象发出请求之前进行一个额外的操作，例如权限检查等
   3. **远程代理**为位于两个不同地址空间对象的访问提供了一种实现机制，可以将一些消耗资源较多的对象和操作移至性能更好的计算机上，提高系统的整体运行效率。
   4. **虚拟代理**通过一个消耗资源较少的对象来代表一个消耗资源较多的对象，可以在一定程度上节省系统的运行开销。
   5. **缓冲代理**为某一个操作的结果提供临时的缓存存储空间，以便在后续使用中能够共享这些结果，优化系统性能，缩短执行时间。
   6. **保护代理**可以控制对一个对象的访问权限，为不同用户提供不同级别的使用权限。
3. 缺点
   1. 由于在客户端和真实主题之间增加了一个代理对象，所以会造成请求的处理速度变慢
   2. 实现代理类也需要额外的工作，从而增加了系统的实现复杂度。
4. 场景
   1. 当客户端对象需要访问远程主机中的对象时可以使用**远程代理**。
   2. 当需要用一个消耗资源较少的对象来代表一个消耗资源较多的对象，从而降低系统开销、缩短运行时间时可以使用**虚拟代理**，例如一个对象需要很长时间才能完成加载时。
   3. 当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问这些结果时可以使用**缓冲代理**。通过使用缓冲代理，系统无须在客户端每一次访问时都重新执行操作，只需直接从临时缓冲区获取操作结果即可。
   4. 当需要控制对一个对象的访问，为不同用户提供不同级别的访问权限时可以使用**保护代理**。
   5. 当需要为一个对象的访问（引用）提供一些额外的操作时可以使用**智能引用代理**。

##### Refer

###### .NET 场景：

1. 代理模式在Net的FCL中的实现也不少，框架级别的有，类级别的也有。框架级别的有WCF,Remoting,他们都需要生成本地的代理，然后通过代理访问进程外或者机器外的对象。类级别的有StringBuilder类型，StringBuilder其实就是一种代理，我们本意是想访问字符串的，StringBuilder就是一种可变字符串的代理，而且StringBuilder也没有和String保持接口的一致性。

##### Summary

第一个部分是“创建型”的设计模式，解决对象创建的问题，对对象创建的解耦。第二部分就是“结构型”的设计模式，所谓结构型设计模式模式，顾名思义讨论的是类和对象的结构 ，主要用来处理类或对象的组合。它包括两种类型，一是类结构型模式，指的是采用继承机制来组合接口或实现；二是对象结构型模式，指的是通过组合对象的方式来实现新的功能。它包括适配器模式、桥接模式、装饰者模式、组合模式、外观模式、享元模式和代理模式。设计模式到现在也说了不少了，但是看起来很多模式都很类似，之间好像很容转换，有时候条件不同了，的确模式也可以转换，但是不能肆意的转换。为了避免思想的混乱，我们把“结构型”这个几个设计模式，再总结一次，把握核心，理解使用场景。

  适配器模式注重转换接口，将不吻合的接口适配对接  
  
    桥接模式注重分离接口与其实现，支持多维度变化  
  
    组合模式注重统一接口，将“一对多”的关系转化为“一对一”的关系  
  
    装饰者模式注重稳定接口，在此前提下为对象扩展功能  
  
    外观模式注重简化接口，简化组件系统与外部客户程序的依赖关系  
  
    享元模式注重保留接口，在内部使用共享技术对对象存储进行优化  
  
    代理模式注重假借接口，增加间接层来实现灵活控制

### 模版方法模式Template Method

##### What

###### 引言

【模板方法】，英文名称是：Template Method Pattern。还是老套路，先从名字上来看看。“模板方法”我第一次看到这个名称，我的理解是，有一个方法的名字叫“模板方法”，后来深入学习之后，感觉最初的理解还没错，也可以换个理解方法，有一个方法包含了一个模板，这个模板是一个算法。在我们的现实生活中有很多例子可以拿来说明这个模式，就拿吃饺子这个事情来说，要想吃到饺子必须经过三步，第一步是“和面”，第二步是“包馅”，第三步是“煮饺子”，这三步就是一个算法，我们要想吃到不同的面和馅的饺子，对这三步中的任意一步就行操作就可以，也可以完全定义这三步

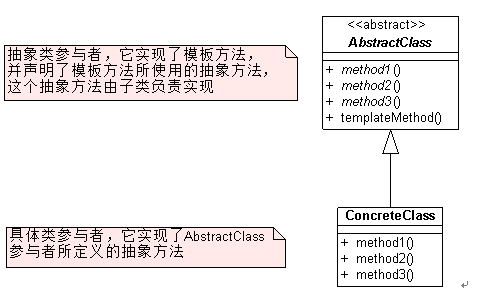
###### 动机（Motivate）

1. 在软件构建过程中，对于某一项任务，它常常有稳定的整体操作结构，但各个子步骤却有很多改变的需求，或者由于固有的原因（比如框架与应用之间的关系）而无法和任务的整体结构同时实现。如何在确定稳定操作结构的前提下，来灵活应对各个子步骤的变化或者晚期实现需求？
2. Template Method Pattern 是一种具有模板功能的pattern。作为模板的方法要定义在父类，并且模板方法中使用了类的抽象方法，完成了程序流程。实现实现抽象方法的是子类。理论上，如果在不同的子类执行不同的实现，应该就能发展出不同的处理内容。不过无论在哪个子类执行任何一种实现，处理的大致流程都还是要依照父类所制定的方式。
3. 像这样在父类指定处理大纲，在子类规定具体内容的Design Pattern就称为Template Mehtod Pattern。

###### 意图（Intent）

1. 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
2. 在父类建立处理逻辑的大纲骨架，而在子类补充具体的处理内容。

###### 结构图（Structure）



###### 组成

1. 抽象类角色（AbstractClass）：定义一个模板方法（TemplateMethod），在该方法中包含着一个算法的骨架，具体的算法步骤是PrimitiveOperation1方法和PrimitiveOperation2方法，该抽象类的子类将重定义PrimitiveOperation1和PrimitiveOperation2操作。
2. 具体类角色（ConcreteClass）：实现PrimitiveOperation1方法和PrimitiveOperation2方法以完成算法中与特定子类（Client）相关的内容。
3. 在模板方法模式中，AbstractClass中的TemplateMethod提供了一个标准模板，该模板包含PrimitiveOperation1和PrimitiveOperation2两个方法，这两个方法的内容Client可以根据自己的需要重写。

##### How

###### 实现方法

1. **定义抽象类角色（AbstractClass）和其模版方法**
2. **定义模版方法里面的算法方法结构，Step1,Step2,可以有默认实现，供子类重写**
3. **定义实现类（ConcreteClass），继承自抽象类角色，根据子类逻辑实现父类算法方法**

###### 实现要点

1. 要点
   1. Template Method模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制（虚函数的多态性）为很多应用程序框架提供了灵活的扩展，是代码复用方面的基本实现结构。除了可以灵活应对子步骤的变化外，“Don't call me, let me call you（不要调用我，让我来调用你）”的反向控制结构是Template Method的典型应用。
2. 优点
   1. TemplateMethod模式是一种非常基础性的设计模式，在面向对象系统中大量应用。它用最简洁的机制（基础、多态）为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。
   2. 在具体实现方面，被TemplateMethod调用的虚方法可以具有实现，也可以没有任何实现（抽象方法或虚方法）。但一般推荐将它们设置为protected方法使得只有子类可以访问它们。
   3. 模板方法模式通过对子类的扩展增加新的行为，符合“开闭原则”。
3. 缺点
   1. 无
4. 场景
   1. 一次性实现一个算法的不变部分，并将可变的行为留给子类来实现。
   2. 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。
   3. 控制子类扩展。模板方法只允许在特定点进行扩展，而模板部分则是稳定的。

##### Refer

###### .NET 场景：

1. 比如：控件有自己的生命周期，Page对象也有自己的生命周期，Application应用对象也有自己的生命周期，这个生命周期里面的每个阶段其实就是模板方法里面包含的每个步骤，这些阶段步骤会被一个方法包含着，这个方法就是“模板方法”。让我们再说说控件吧，因为写好的控件，可能需要被开发人员自定义，那么在控件里我们已经定义好了控件呈现、动作的骨架，但是有些自定义的需求，需要延迟到扩展控件的开发人员来决定。

##### Summary

### 命令模式Command

##### What

###### 引言

【命令模式】，又称为行动（Action）模式或交易（Transaction）模式，英文名称是：Command Pattern。还是老套路，先从名字上来看看。“命令模式”我第一次看到这个名称，我的理解是，可能是一种行为或者一个操作就是一个命令。“命令”这个词语在军队里面用的最多，比如：下达作战命令，接下来就是上战场玩命了。基于这些，我感觉“命令”就是任务，执行了命令就完成了一个任务。或者说，命令是任务，我们再从这个名字上并不知道命令的发出者和接受者分别是谁，为什么呢？因为我们并不关心他们是谁，发出命令的人发出命令，可以继续做其他的事情，接受命令的人执行任务就可以，不需要你发出命令，还要监督我们完成，只要我们完成任务是合格的就行。这种行为也就是“解耦”。在我们的现实生活中有很多例子可以拿来说明这个模式，我们还拿吃饺子这个事情来说。我的奶奶说了，今天想吃饺子，发出了命令，然后我奶奶就去看电视去了。我们夫妻俩收到命令就开始和面，做饺子馅，包饺子。饺子包好了，我们就休息一会，等下午5点就开始烧水煮饺子了，晚饭的时间到了，我奶奶按时吃上了饺子。

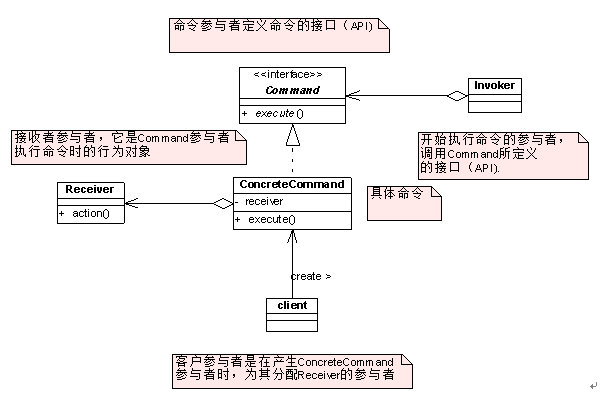
###### 动机（Motivate）

1. 在软件构建过程中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合——比如需要对行为进行“记录、撤销/重做（undo/redo）、事务”等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将“行为请求者”与“行为实现者”解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。
2. 当类在执行操作时，会调用自己类或其他类的方法。调用方法之后的结果会反映在对象状态上，但却不会留下任何操作记录。遇到这咱情形时，要是有一个类能表现“请执行这项作业”的“命令”就方便多了。因为如此一来便可用一个“表示命令的类的对象实例”来代表欲执行的操作，而不需要采用“调用方法”的方式处理。如欲管理相关记录，只管理该对象实例的集合即可。而且若预先将命令的集合存储起来，还可再执行同一命令：或者是把多个命令结合成一个新命令供再利用。
3. command有时也称为Event。一旦发生事件时，则先将该事件变成对象实例，按照发生顺序排入等候序列内。接着，再依次处理所有排列等待事件。
4. 这种模式在我们现实的编码中用的不是很多，可能针对特定领域的软件或者系统需求更大，比如：文档编辑等。

###### 意图（Intent）

1. 将一个请求封装为一个对象，从而使你可用不同的请求对客户（客户程序，也是行为的请求者）进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

###### 结构图（Structure）



###### 组成

1. **客户角色（Client）**：创建具体的命令对象，并且设置命令对象的接收者。注意这个不是我们常规意义上的客户端，而是在组装命令对象和接收者，或许，把这个Client称为装配者会更好理解，因为真正使用命令的客户端是从Invoker来触发执行。
2. **命令角色（Command）**：声明了一个给所有具体命令类实现的抽象接口。
3. **具体命令角色（ConcreteCommand）**：命令接口实现对象，是“虚”的实现；通常会持有接收者，并调用接收者的功能来完成命令要执行的操作。
4. **请求者角色（Invoker）：**要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。这个是客户端真正触发命令并要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口。
5. **接受者角色（Receiver）**：接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。

##### How

###### 实现方法

1. **定义抽象命令（Command）属性，传入命令要执行的人员**
2. **定义具体的命令（ConcreteCommand），定义要执行命令的人（Receiver），实现具体的命令，在命令里面使用执行人来执行命令（Execute）**
3. **定义命令的执行人（Receiver），定义自身的行为**
4. **定义命令的传达人（Invoker）， 传入要传达的命令(Command) ，定义传达方法(Send) ，在传达方法体内执行命令执行方法(Execute)**

###### 实现要点

1. 要点
   1. Command模式的根本目的在于将“行为请求者”与“行为实现者”解耦，在面向对象语言中，常见的实现手段是“将行为抽象为对象”。
   2. 实现Command接口的具体命令对象ConcreteCommand有时候根据需要可能会保存一些额外的状态信息。
   3. 通过使用Composite组合模式，可以将多个命令封装为一个“复合命令”MacroCommand。
   4. Command模式与C#中的Delegate有些类似。但两者定义行为接口的规范有所区别：Command以面向对象中的“接口-实现”来定义行为接口规范，更严格，更符合抽象原则；Delegate以函数签名来定义行为接口规范，更灵活，但抽象能力比较弱。
   5. 使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。
2. 优点
   1. 命令模式使得新的命令很容易被加入到系统里。
   2. 可以设计一个命令队列来实现对请求的Undo和Redo操作。
   3. 可以较容易地将命令写入日志。
   4. 可以把命令对象聚合在一起，合成为合成命令。合成命令式合成模式的应用。
3. 缺点
   1. 使用命令模式可能会导致系统有过多的具体命令类。这会使得命令模式在这样的系统里变得不实际。
4. 场景
   1. 系统需要支持命令的撤销（undo）。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用undo方法把命令所产生的效果撤销掉。命令对象还可以提供redo方法，以供客户端在需要时，再重新实现命令效果。
   2. 系统需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命周期。意思为：原来请求的发出者可能已经不存在了，而命令对象本身可能仍是活动的。这时命令的接受者可以在本地，也可以在网络的另一个地址。命令对象可以串行地传送到接受者上去。
   3. 如果一个系统要将系统中所有的数据消息更新到日志里，以便在系统崩溃时，可以根据日志里读回所有数据的更新命令，重新调用方法来一条一条地执行这些命令，从而恢复系统在崩溃前所做的数据更新。
   4. 系统需要使用命令模式作为“CallBack(回调)”在面向对象系统中的替代。Callback即是先将一个方法注册上，然后再以后调用该方法。

##### Refer

###### .NET 场景：

1. 由于.NET有了Delegate，它很少很少用到Command。它只要需要用到行为抽象，它都用Delegate去做。因为这是Framework，这是和业务领域相关度不大的基础建设层面，它是不太需要用到OO的层面。对于我们来说，我们建议更多地用Command去实现。

##### Summary

命令模式是把一个操作或者行为抽象为一个独立的对象，通过对命令的抽象化来使得发出命令的责任和执行命令的责任分隔开，也可以对独立的命令对象进行特殊操作，比如可以实现命令的撤销和恢复功能。

### 迭代器模式Iterator

##### What

###### 引言

【迭代器模式】，英文名称是：Iterator Pattern。还是老套路，先从名字上来看看。“迭代器模式”我第一次看到这个名称，我的理解是，迭代是遍历的意思，迭代器可以理解为是遍历某某的工具，遍历什么呢？在软件设计中，当然遍历的是集合对象，所以说迭代器模式是遍历集合的一种通用的算法。如果集合只有一种类型，那这个模式就没用了，就是因为集合对象包含数组、列表，字典和哈希表等各种对象，如果为每一种集合对象都实现一套遍历算法，也不太现实，因此为了解决遍历集合有一个统一的接口这个事情，所以就提出了“迭代器”这个模式。

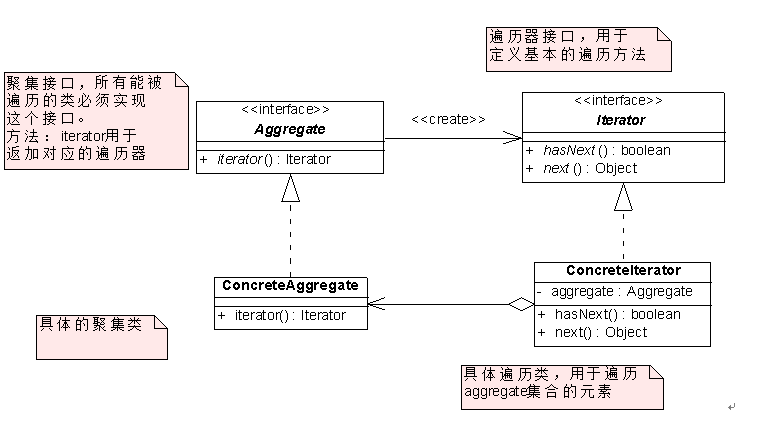
###### 动机（Motivate）

1. 在软件构建过程中，集合对象内部结构常常变化各异。但对于这些集合对象，我们希望在不暴露其内部结构的同时，可以让外部客户代码透明地访问其中包含的元素；同时这种“透明遍历”也为“同一种算法在多种集合对象上进行操作”提供了可能。
2. 使用面向对象技术将这种遍历机制抽象为“迭代器对象”为“应对变化中的集合对象”提供了一种优雅的方式。

###### 意图（Intent）

1. 提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露该对象的内部表示。

###### 结构图（Structure）



###### 组成

1. **抽象迭代器(Iterator)**：抽象迭代器定义了访问和遍历元素的接口，一般声明如下方法：用于获取第一个元素的first()，用于访问下一个元素的next()，用于判断是否还有下一个元素的hasNext()，用于获取当前元素的currentItem()，在其子类中将实现这些方法。
2. **具体迭代器(ConcreteIterator)**：具体迭代器实现了抽象迭代器接口，完成对集合对象的遍历，同时在对聚合进行遍历时跟踪其当前位置。
3. **抽象聚合类(Aggregate)**：抽象聚合类用于存储对象，并定义创建相应迭代器对象的接口，声明一个createIterator()方法用于创建一个迭代器对象。
4. **具体聚合类(ConcreteAggregate)**：具体聚合类实现了创建相应迭代器的接口，实现了在抽象聚合类中声明的createIterator()方法，并返回一个与该具体聚合相对应的具体迭代器ConcreteIterator实例。

##### How

###### 实现方法

1. **定义抽象迭代器（Iterator），定义基本行为，First(),Next(),HasNext(),GetCurrent()**
2. **定义抽象的聚合类（Aggregate），定义基本行为CreateIterator(),创建该聚合的迭代器对象**
3. **定义迭代器实现类（ConcreteIterator），实现基本行为，并且注入聚合和迭代器索引，来实现聚合的迭代器功能**
4. **定义聚合实现类（ConcreteAggregate），定义基本行为，并且实现接口为返回为迭代器实现类的实例的方法**

###### 实现要点

1. 要点
   1. 迭代抽象：访问一个聚合对象的内容而无需暴露它的内部表示。
   2. 迭代多态：为遍历不同的集合结构提供一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作。
   3. 迭代器的健壮性考虑：遍历的同时更改迭代器所在的集合结构，会导致问题。
2. 优点
   1. 迭代器模式使得访问一个聚合对象的内容而无需暴露它的内部表示，即迭代抽象。
   2. 迭代器模式为遍历不同的集合结构提供了一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作
3. 缺点
   1. 迭代器模式在遍历的同时更改迭代器所在的集合结构会导致出现异常。所以使用foreach语句只能在对集合进行遍历，不能在遍历的同时更改集合中的元素。
4. 场景
   1. 访问一个聚合对象的内容而无需暴露它的内部表示。
   2. 支持对聚合对象的多种遍历。
   3. 为遍历不同的聚合结构提供一个统一的接口(即, 支持多态迭代)。

##### Refer

###### .NET 场景：

1. System.Collections，在该命名空间里面早已有了迭代器模式的实现。对于聚集接口和迭代器接口已经存在了，**其中IEnumerator扮演的就是迭代器的角色**
2. public interface IEnumerator{object Current{get;}bool MoveNext();void Reset();}
3. public interface IEnumable{ IEnumerator GetEnumerator();}
4. 抽象聚合角色（Aggregate）和抽象迭代器角色（Iterator）分别是IEnumerable接口和IEnumerator接口
5. 具体聚合角色（ConcreteAggregate）有Queue类型， BitArray等类型

##### Summary

### 观察者模式Observer

##### What

###### 引言

【观察者模式】，英文名称是：Observer Pattern。还是老套路，先从名字上来看看。“观察者模式”我第一次看到这个名称，我的理解是，既然有“观察者”，那肯定就有“被观察者”了，“观察者”监视着“被观察者”，如果“被观察者”有所行动，“观察者”就会做出相应的动作来回应。“观察者模式”在现实生活中，实例其实是很多的，比如：八九十年代我们订阅的报纸，我们会定期收到报纸，因为我们订阅了。银行可以给储户发手机短信，也是“观察者模式”很好的使用的例子，因为我们订阅了银行的短信业务，当我们账户余额发生变化就会收到通知

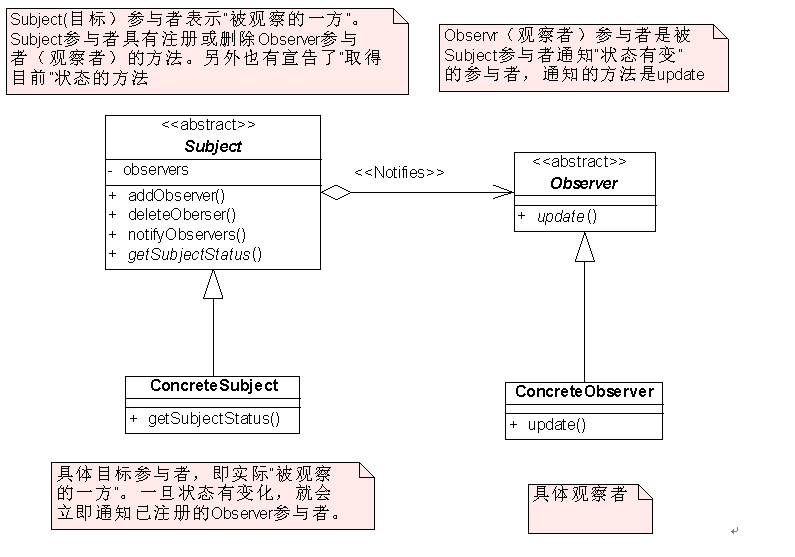
###### 动机（Motivate）

1. 在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系”——一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。
2. 使用面向对象技术，可以将这种依赖关系弱化，并形成一种稳定的依赖关系。从而实现软件体系结构的松耦合。

###### 意图（Intent）

1. 定义对象间的一种一对多的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。
2. 当被Observer Pattern列入观察名单的状态发生变化，就会通知观察者，在写一些跟状态变化有关的处理时，Observer Pattern是很好用的工具。

###### 结构图（Structure）



###### 组成

1. 抽象主题角色（Subject）：抽象主题把所有观察者对象的引用保存在一个列表中，并提供增加和删除观察者对象的操作，抽象主题角色又叫做抽象被观察者角色，一般由抽象类或接口实现。
2. 抽象观察者角色（Observer）：为所有具体观察者定义一个接口，在得到主题通知时更新自己，一般由抽象类或接口实现。
3. 具体主题角色（ConcreteSubject）：实现抽象主题接口，具体主题角色又叫做具体被观察者角色。
4. 具体观察者角色（ConcreteObserver）：实现抽象观察者角色所要求的接口，以便使自身状态与主题的状态相协调。

##### How

###### 实现方法

1. **定义观察者抽象对象（Subject），定义Add(Observer obs)、Remove(Observer obs)、Notify()[用来监控状态]**
2. **定义观察者对象实例（ConcreteSubject），实现抽象方法 ，实现Notify具体细节**
3. **定义被观察者抽象对象（Observer），定义一些属性，和关联状态的属性如Temperature，IsChanged，定义更新数据的抽象方法Update(int temperature)**
4. **定义被观察者对象实例（ConcreteObserver），实现Update()方法**

###### 实现要点

1. 要点
   1. 使用面向对象的抽象，Observer模式使得我们可以独立地改变目标与观察者（面向对象中的改变不是指改代码，而是指扩展、子类化、实现接口），从而使二者之间的依赖关系达致松耦合。
   2. 目标发送通知时，无需指定观察者，通知（可以携带通知信息作为参数）会自动传播。观察者自己决定是否需要订阅通知，目标对象对此一无所知。
   3. 在C#的event中，委托充当了抽象的Observer接口，而提供事件的对象充当了目标对象。委托是比抽象Observer接口更为松耦合的设计。
2. 优点
   1. 观察者模式实现了表示层和数据逻辑层的分离，并定义了稳定的更新消息传递机制，并抽象了更新接口，使得可以有各种各样不同的表示层，即观察者。
   2. 观察者模式在被观察者和观察者之间建立了一个抽象的耦合，被观察者并不知道任何一个具体的观察者，只是保存着抽象观察者的列表，每个具体观察者都符合一个抽象观察者的接口。
   3. 观察者模式支持广播通信。被观察者会向所有的注册过的观察者发出通知。
3. 缺点
   1. 如果一个被观察者有很多直接和间接的观察者时，将所有的观察者都通知到会花费很多时间。
   2. 虽然观察者模式可以随时使观察者知道所观察的对象发送了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎样发生变化的。
   3. 如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃，在使用观察者模式应特别注意这点。
4. 场景
   1. 对对象的属性数据需要进行监控的时候

##### Refer

###### .NET 场景：

1. “在C#的event中，委托充当了抽象的Observer接口，而提供事件的对象充当了目标对象。委托是比抽象Observer接口更为松耦合的设计。”，其实在Net里面实现的观察者模式做了一些改变，用委托或者说是事件来实现观察者模式。事件我们都很明白，我们可以注册控件的事件，当触发控件的动作时候，相应的事件就会执行，在事件的执行过程中我们就可以做相关的提醒业务

##### Summary

该模式不是很难，结构也不是很复杂，唯一让我们多多注意的是状态的管理。

### 中介模式Mediator

##### What

###### 引言

【中介者模式】，英文名称是：Mediator Pattern。还是老套路，先从名字上来看看。“中介者模式”我第一次看到这个名称，我的理解是，在两个对象中间增加一个“中间对象”，“中间对象”协调两个对象的关系，但是随着理解的深入，“中间对象”处理关系的对象可能不只是两个对象，可能是三个对象，或者更多对象。中介者模式在现实生活中的例子很多，比如：A对象和B对象做生意，如果A对象和B对象是一次性买卖，没有讨价还价的过程，A对象和B对象直接见面更好。但是A对象或者B对象的想法经常变，每次想法的改变都通知对方，就会使对方很反感，而且也不利于生意的顺利进行。如果在A对象和B对象之间增加一个C对象，在最终确定之前不要告诉C对象，对方也就不知道（隔离了耦合，对方可以更具需求变化），等一方最终确定想法后，把最后决定告诉C对象，C对象再转告对方，这样就简化了A对象和B对象的交易过程，而且双方都很满意。在软件构建过程中，因为有了变化，才有增加中介者的需要，如果没有变化，可以一次搞定，直接硬编码也没关系，所以说“变化”是模式的前提，无论是什么模式，就因为有变化，我们需要抵御变化，才要使用相应的模式来解决问题。

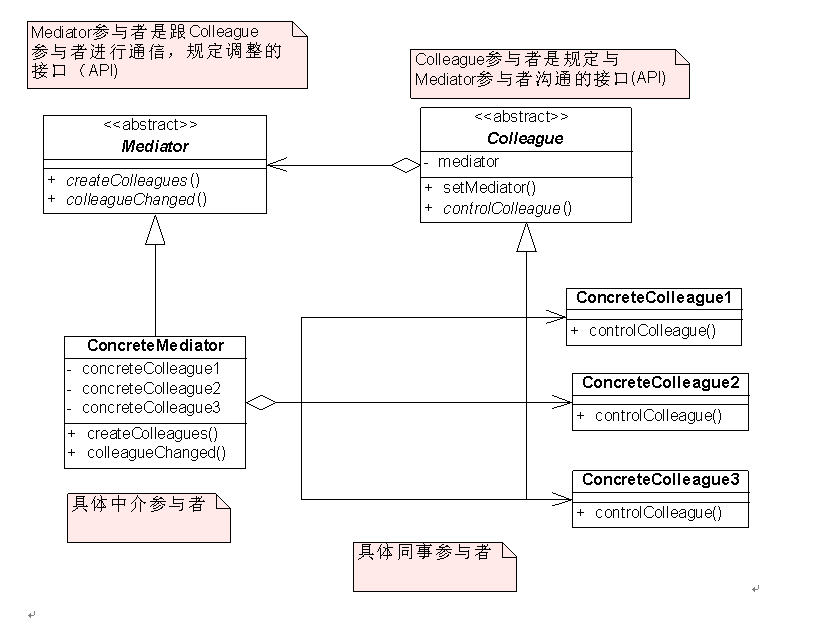
###### 动机（Motivate）

1. 在软件构建过程中，经常会出现多个对象互相关联交互的情况，对象之间常常会维持一种复杂的引用关系，如果遇到一些需求的更改，这种直接的引用关系将面临不断地变化。
2. 在这种情况下，我们可使用一个“中介对象”来管理对象间的关联关系，避免相互交互的对象之间的紧耦合引用关系，从而更好地抵御变化。

###### 意图（Intent）

1. 定义了一个中介对象来封装一系列对象之间的交互关系。中介者使各个对象之间不需要显式地相互引用，从而使耦合性降低，而且可以独立地改变它们之间的交互行为。
2. 在Mediator Pattern中，“顾问”称为mediator(中介者)、“各个成员”称为colleague(同事)。每个成员都只对顾问提出报告，也只有顾问会发出指示给各个成员；成员们彼此也不会去探问目前状况如何，或乱发指示给其他成员。

###### 结构图（Structure）



###### 组成

1. **抽象中介者角色（Mediator）**：在里面定义各个同事之间交互需要的方法，可以是公共的通信方法，也可以是小范围的交互方法。
2. **具体中介者角色（ConcreteMediator）**：它需要了解并维护各个同事对象，并负责具体的协调各同事对象的交互关系。
3. **抽象同事类（Colleague）**：通常为抽象类，主要约束同事对象的类型，并实现一些具体同事类之间的公共功能，比如，每个具体同事类都应该知道中介者对象，也就是具体同事类都会持有中介者对象，都可以到这个类里面。
4. **具体同事类（ConcreteColleague）**：实现自己的业务，需要与其他同事通信时候，就与持有的中介者通信，中介者会负责与其他同事类交互。

##### How

###### 实现方法

1. **Component**
   1. ）

###### 实现要点

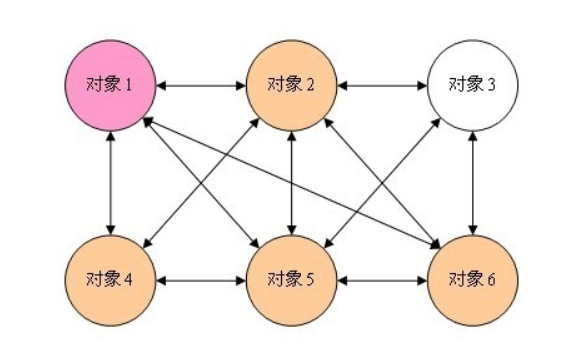
1. 要点
   1. 将多个对象间复杂的关联关系解耦，Mediator模式将多个对象间的控制逻辑进行集中管理，变“多个对象互相关联”为“多个对象和一个中介者关联”，简化了系统的维护，抵御了可能的变化。随着控制逻辑的复杂化，Mediator具体对象的实现可能相当复杂。这时候可以对Mediator对象进行分解处理。
   2. Facade模式是解耦系统外到系统内（单向）的对相关联关系
   3. Mediator模式是解耦系统内各个对象之间（双向）的关联关系
2. 优点
   1. **松散耦合**
      1. 中介者模式通过把多个同事对象之间的交互封装到中介对象里面，从而使得对象之间松散耦合，基本上可以做到互不依赖。这样一来，同时对象就可以独立的变化和复用，不再“牵一发动全身”
   2. **集中控制交互**
      1. 多个同事对象的交互，被封装在中介者对象里面集中管理，使得这些交互行为发生变化的时候，只需要修改中介者就可以了。
   3. **多对多变为一对多**
      1. 没有中介者模式的时候，同事对象之间的关系通常是多对多，引入中介者对象后，中介者和同事对象的关系通常变为双向的一对多，这会让对象的关系更容易理解和实现。
3. 缺点
   1. **过多集中化**
      1. 如果同事对象之间的交互非常多，而且比较复杂，当这些复杂性全都集中到中介者的时候，会导致中介者对象变的十分复杂，而且难于维护和管理。
4. 场景

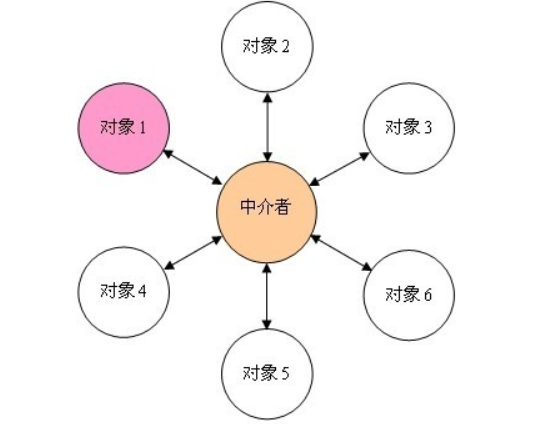
##### Refer

###### .NET 场景：

1. 微软的ASP.NET MVC开发模式就是一个中介者模式的很好的实现，其中C就是Controller，也就是中文所说的控制器，控制器就是一个中介者，M和V和它打交道

##### Summary

为什么要使用中介者模式呢？如果不使用中介者模式的话，各个同事对象将会相互进行引用，如果每个对象都与多个对象进行交互时，将会形成如下图所示的网状结构。  


如果不使用中介者模式的话，每个对象之间过度耦合，这样的既不利于类的复用也不利于扩展。如果引入了中介者模式，那么对象之间的关系将变成星型结构，采用中介者模式之后会形成如下图所示的结构：  
  

使用中介者模式之后，任何一个类的变化，只会影响中介者和类本身，不像之前的设计，任何一个类的变化都会引起其关联所有类的变化。这样的设计大大减少了系统的耦合度。

### 状态模式State

##### What

###### 引言

【状态模式】，英文名称是：State Pattern。无论是现实世界，还是面向对象的OO世界，里面都有一个东西，那就是对象。有对象当然就有状态了。每个对象都有其对应的状态，而每个状态又有对应一些相应的行为，在不同状态下，行为的的方式也是不一样。如果某个对象有多个状态时，那么就会有很多对应的行为。那么对这些状态的判断和根据状态完成的行为，就会导致多重条件语句交织在一起，并且如果添加一种新的状态时，需要更改之前现有的代码。这样的设计显然违背了开闭原则，状态模式正是用来解决这样的问题的。

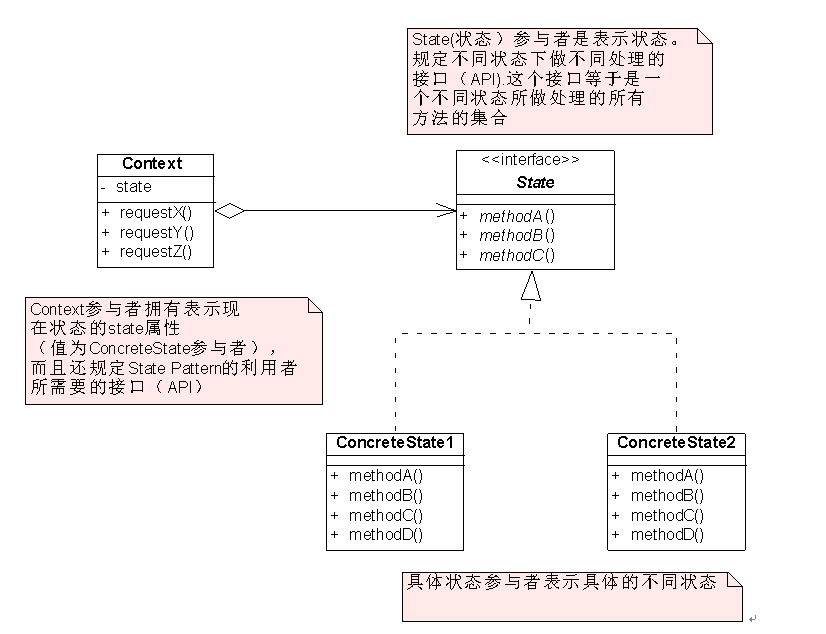
###### 动机（Motivate）

1. 在软件构建过程中，某些对象的状态如果改变，其行为也会随之而发生变化，比如文档处于只读状态，其支持的行为和读写状态支持的行为就可能完全不同。
2. 如何在运行时根据对象的状态来透明地更改对象的行为？而不会为对象操作和状态转化之间引入紧耦合？

###### 意图（Intent）

1. 允许一个对象在其内部状态改变时改变它的行为。从而使对象看起来似乎修改了其行为。
2. 把状态抽象为类，也就是说，以类来表现状态

###### 结构图（Structure）



###### 组成

1. **环境角色（Context）：也称上下文，定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境对象的现有状态。**
2. **抽象状态角色（State）：定义一个接口，用以封装环境对象的一个特定的状态所对应的行为。**
3. **具体状态角色（ConcreteState）：每一个具体状态类都实现了环境（Context）的一个状态所对应的行为。**
4. 在状态模式结构中需要理解环境类与抽象状态类的作用：
   1. 环境类实际上就是拥有状态的对象，环境类有时候可以充当状态管理器(State Manager)的角色，可以在环境类中对状态进行切换操作。
   2. 抽象状态类可以是抽象类，也可以是接口，不同状态类就是继承这个父类的不同子类，状态类的产生是由于环境类存在多个状态，同时还满足两个条件：这些状态经常需要切换，在不同的状态下对象的行为不同。因此可以将不同对象下的行为单独提取出来封装在具体的状态类中，使得环境类对象在其内部状态改变时可以改变它的行为，对象看起来似乎修改了它的类，而实际上是由于切换到不同的具体状态类实现的。由于环境类可以设置为任一具体状态类，因此它针对抽象状态类进行编程，在程序运行时可以将任一具体状态类的对象设置到环境类中，从而使得环境类可以改变内部状态，并且改变行为。

##### How

###### 实现方法

1. 状态模式在我们的现实生活中也有类似的例子，例如：在我们上网购买商品的过程中，就可以查看订单的随时状态。对于商家来说，订单的状态不同，也会允许客户有不同的动作要求，比如：订单在已经处于发货状态，此订单是不能退货的。如果订单在备货阶段，客户是可以换货或者退货的。如果我们的订单已经发货了，您就等着接收货物吧，如果货物有质量问题，可以拒签，或者顺利完成交易，今天我们就以订单为例来说明状态模式的实现。
2. **Component**
   1. ）

###### 实现要点

1. 要点
   1. State模式将所有与一个特定状态相关的行为都放入一个State的子类对象中，在对象状态切换时，切换相应的对象；但同时维持State的接口，这样实现了具体操作与状态转换之间的解耦。
   2. 为不同的状态引入不同的对象使得状态转换变得更加明确，而且可以保证不会出现状态不一致的情况，因为转换是原子性的——即要么彻底转换过来，要么不转换。
   3. 如果State对象没有实例变量，那么各个上下文可以共享同一个State对象，从而节省对象开销。
2. 优点
   1. 封装了转换规则。
   2. 枚举可能的状态，在枚举状态之前需要确定状态种类。
   3. 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
   4. 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。
   5. 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。
3. 缺点
   1. 状态模式的使用必然会增加系统类和对象的个数。
   2. 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
   3. 状态模式对“开闭原则”的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态；而且修改某个状态类的行为也需修改对应类的源代码。
4. 场景
   1. 对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为。
   2. 代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。在这些条件语句中包含了对象的行为，而且这些条件对应于对象的各种状态

##### Refer

###### .NET 场景：

1. 。

##### Summary

对

### 策略模式Stragety

##### What

###### 引言

【策略模式】，英文名称是：Stragety Pattern。在现实生活中，策略模式的例子也非常常见，例如，在一个公司中，会有各种工作人员，比如：有的是普通员工，有的是软件架构师，有的是部门经理，当然也会有公司的CEO。这些工作人员负责的工作不同，担负的责任不同，自然得到的报酬也就不同了。每种工作人员都有自己的工资，但是每个工种的工作人员的工资的计算方法又是不一样的。如果所有人的工资都一样，肯定会天下大乱的。如果不采用策略模式来实现这个需求的话，我们可能会这样来做，我们会定义一个工资类，该类有一个属性来标识工作人员的类型，并且有一个计算工资的CalculateSalary()方法，在该方法体内需要对工作人员类型进行判断，通过if-else语句来针对不同的工作人员类型来计算其所得工资。这样的实现确实可以解决这个场景，但是这样的设计不利于扩展，如果系统后期需要增加一种新的工种时，此时不得不回去修改CalculateSalary方法来多添加一个判断语句，这样明显违背了“开放——封闭”原则。此时，我们可以考虑使用策略模式来解决这个问题，既然工资计算方法是这个场景中的变化部分，此时自然可以想到对工资算法进行抽象，不同工种的工资可以用不用的策略算法具体实现，想要得到某个工作人员的工资，用其相应的工资算法策略来计算就可以了。

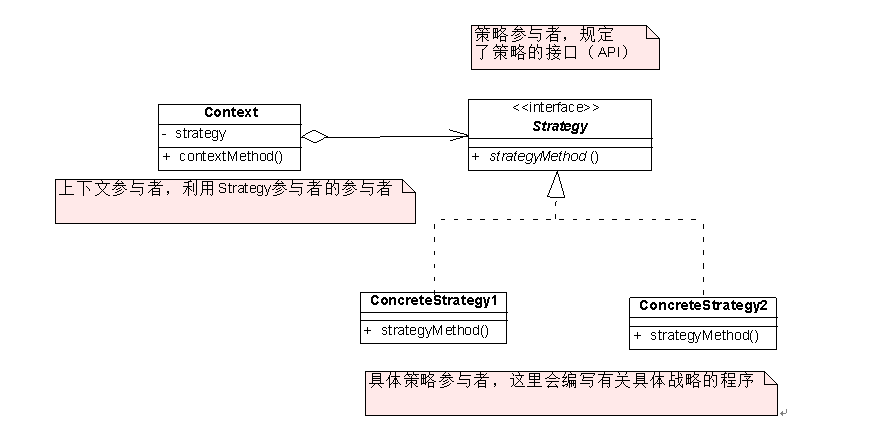
###### 动机（Motivate）

1. 在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使对象变得异常复杂；而且有时候支持不使用的算法也是一个性能负担。如何在运行时根据需要透明地更改对象的算法？将算法与对象本身解耦，从而避免上述问题？

###### 意图（Intent）

1. 定义一系列算法，把它们一个个封装起来，并且使它们可互相替换。该模式使得算法可独立于使用它的客户而变化
2. strategy是“战略”的意思，在程序设计里，不妨把它视为“算法”。任何一种程序都是为了解决问题而撰写出来的，解决问题时需要实现一些特定的运算法则。在Strategy Pattern下，可以更换实现算法的部分而不留痕迹。切换整个算法，简化改为采用其他方法来解决同样的问题，这就是Strategy Pattern。

###### 结构图（Structure）



###### 组成

1. **环境角色（Context）：**持有一个Strategy类的引用。
   1. 需要使用ConcreteStrategy提供的算法。
   2. 内部维护一个Strategy的实例。
   3. 负责动态设置运行时Strategy具体的实现算法。
   4. 负责跟Strategy之间的交互和数据传递
2. **抽象策略角色（Strategy）**：定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，Context使用这个接口调用不同的算法，一般使用接口或抽象类实现。
3. **具体策略角色（ConcreteStrategy）**：实现了Strategy定义的接口，提供具体的算法实现。

##### How

###### 实现方法

1. 在现实生活中，策略模式的例子也是很多的，例如：一个公司会有很多工作种类，每个工作种类负责的工作不同，自然每个工种的工资计算方法也会有千差万别，我们今天就以工资的计算为例来说明策略模式的使用
   1. ）

###### 实现要点

1. 要点
   1. Strategy及其子类为组件提供了一系列可重用的算法，从而可以使得类型在运行时方便地根据需要在各个算法之间进行切换，所谓封装算法，支持算法的变化。Strategy模式提供了用条件判断语句以外的另一种选择，消除条件判断语句，就是在解耦合。含有许多条件判断语句的代码通常都需要Strategy模式。
   2. 与State类似，如果Strategy对象没有实例变量，那么各个上下文可以共享一个Strategy对象，从而节省对象开销。Strategy模式适用的是算法结构中整个算法的改变，而不是算法中某个部分的改变。
   3. Template Method方法：执行算法的步骤协议是本身放在抽象类里面的，允许一个通用的算法操作多个可能实现
   4. Strategy模式：执行算法的协议是在具体类，每个具体实现有不同通用算法来做。
2. 优点
   1. 策略类之间可以自由切换。由于策略类都实现同一个接口，所以使它们之间可以自由切换。
   2. 易于扩展。增加一个新的策略只需要添加一个具体的策略类即可，基本不需要改变原有的代码。
   3. 避免使用多重条件选择语句，充分体现面向对象设计思想。
3. 缺点
   1. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这点可以考虑使用IOC容器和依赖注入的方式来解决，关于IOC容器和依赖注入（Dependency Inject）的文章可以参考：IoC 容器和Dependency Injection 模式。
   2. 策略模式会造成很多的策略类。
4. 场景
   1. 一个系统需要动态地在几种算法中选择一种的情况下。那么这些算法可以包装到一个个具体的算法类里面，并为这些具体的算法类提供一个统一的接口。
   2. 如果一个对象有很多的行为，如果不使用合适的模式，这些行为就只好使用多重的if-else语句来实现，此时，可以使用策略模式，把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现面向对象涉及的概念。

##### Refer

###### .NET 场景：

1. 在.NET中，为集合类型ArrayList和List<T>提供的排序功能，其中实现就利用了策略模式，定义了IComparer接口来对比较算法进行封装，实现IComparer接口的类可以是顺序，或逆序地比较两个对象的大小，具体.NET中的实现可以使用反编译工具查看List<T>.Sort(IComparer<T>)的实现。其中List<T>就是承担着环境角色，而IComparer<T>接口承担着抽象策略角色，具体的策略角色就是实现了IComparer<T>接口的类，List<T>类本身实现了存在实现了该接口的类，我们可以自定义继承与该接口的具体策略类。

##### Summary

对

### 职责链模式Chain Of Responsibility

##### What

###### 引言

【职责链模式】，英文名称是：Chain of Responsibility Pattern。让我们看看现实生活中的例子吧，理解起来可能更容易。我们看看某公司的采购流程吧。某公司的规章制度规定，采购原材料的总价在5万之内，只需要经理级别的人批准即可，采购总价大于5万小于10万的则需要财务经理进行批准，总价大于10万小于30万的需要总经理批准，而总价大于30万的则需要通过董事会会议讨论决定。对于这样一个需求，最直接的方法就是设计一个方法，该方法接受的参数是采购的总价，然后在这个方法内对价格进行判断，然后针对不同的条件交给不同级别的角色去处理，如果情况就是这样，不变了，这样做很好，没问题。如果我们又有新的条件要增加该怎么办呢？我们不得不去修改原来设计的方法来再添加一个条件判断，让本已多重if-else判断语句更多了，这样的设计显然违背了“开放-关闭”原则。这时候，我们可以采用职责链模式来解决这样的问题。

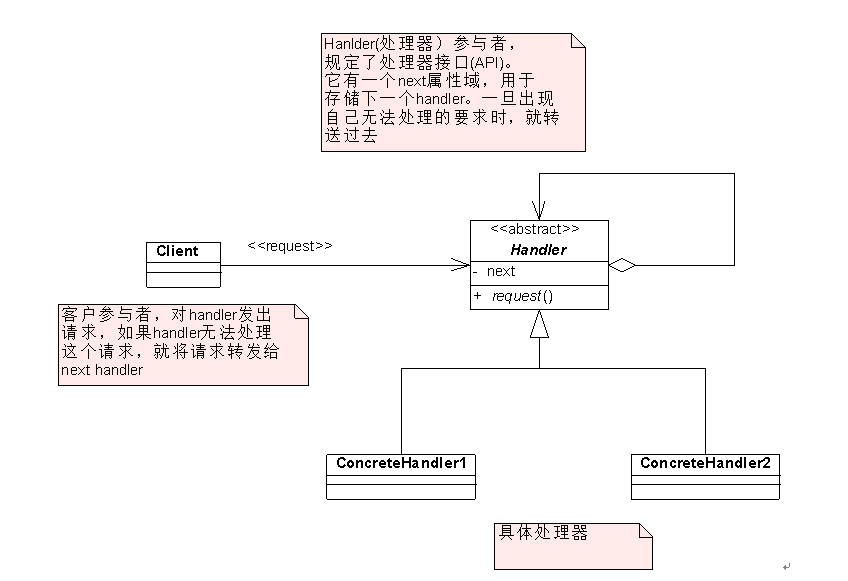
###### 动机（Motivate）

1. 在软件构建过程中，一个请求可能被多个对象处理，但是每个请求在运行时只能有一个接受者，如果显示指定，将必不可少地带来请求发送者与接受者的紧耦合。如何使请求的发送者不需要指定具体的接受者，让请求的接受者自己在运行时决定来处理请求，从而使两者解耦。

###### 意图（Intent）

1. 避免请求发送者与接收者耦合在一起，让多个对象都有可能接受请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。
2. 先对人产生一个要求，如果这个人有处理的能力就处理掉；如果不能处理的话，就把要求转送给“第二个人”。同样的，如果第二个人有处理的能力时就处理掉，不能处理的话，继续转达给“第三个人”，以此类推，就是Chain of Responsibility Pattern

###### 结构图（Structure）



###### 组成

1. **抽象处理者角色（Handler）**：抽象处理者定义了一个处理请求的接口，它一般设计为抽象类，由于不同的具体处理者处理请求的方式不同，因此在其中定义了抽象请求处理方法。因为每一个处理者的下家还是一个处理者，因此在抽象处理者中定义了一个自类型的对象，作为其对下家的引用。通过该引用，处理者可以连成一条链。
2. **具体处理者角色（ConcreteHandler）**：具体处理者是抽象处理者的子类，它可以处理用户请求，在具体处理者类中实现了抽象处理者中定义的抽象处理方法，在处理请求之前需要进行判断，看是否有相应的处理权限，如果可以处理请求就处理它，否则将请求转发给后继者；在具体处理者中可以访问链中下一个对象，以便请求的转发。

##### How

###### 实现方法

1. 在现实生活中，职责链模式的例子也是很多的，例如：公司的请假流程就是一个很好的职责链模式的例子，如果请假半天，只要告诉本部门经理就可以了；如果请假7天或者以上必须人事总监批准；如果请假15天以上，那就要经过总裁批准了。还有类似的例子就是采购的流程，其流程也是职责链模式很好的体现，采购的金额不同，需要批准的人员也不同，比如：部门采购1万元的纸品，只要部门领导签批就可以，如果要采购大于1万小于5万的物品，那就需要财务经理签批了，如果采购30万的原材料或者物品，那就需要总裁或者类似角色才能审批了。
   1. ）

###### 实现要点

1. 要点
   1. Chain of Responsibility模式的应用场合在于“一个请求可能有多个接受者，但是最后真正的接受者只有一个”，只有这时候请求发送者与接受者的耦合才有可能出现“变化脆弱”的症状，职责链的目的就是将二者解耦，从而更好地应对变化。
   2. 应用了Chain of Responsibility模式后，对象的职责分派将更具灵活性。我们可以在运行时动态添加/修改请求的处理职责。
   3. 当我们要新增一个DHandler处理请求，就不需再改原来的代码了，遵从了开放封闭原则。这样我们的程序就更赋予变化，更有变化的抵抗力。Handler类本身继承自BaseHandler类型，又包含了一个BaseHandler类型的对象，这点类似Decorator模式。
   4. 如果请求传递到职责链的末尾仍得不到处理，应该有一个合理的缺省机制。这也是每一个接受对象的责任，而不是发出请求的对象的责任。
2. 优点
   1. 降低耦合度：职责链模式使得一个对象无需知道是其他哪一个对象处理其请求。对象仅需知道该请求会被处理即可，接受者和发送者都没有对方的明确信息，且链中的对象不需要知道链的结构，有客户端负责链的创建。
   2. 可简化对象的相互连接：接受者对象仅需维持一个指向其后继者的引用，而不需维持它对所有的候选处理者的引用。
   3. 增强给对象指派职责的灵活性：在给对象分派职责时，职责链可以给我们带来更多的灵活性。可以通过在运行时对该连进行动态的增加或修改处理一个请求的职责。
   4. 增加新的请求处理类很方便：在系统中增加一个新的请求处理者无需修改原有系统的代码，只需要在客户端重新建链即可，从这一点看来是符合“开闭原则”的。
3. 缺点
   1. 在找到正确的处理对象之前，所有的条件判定都要执行一遍，当责任链过长时，可能会引起性能的问题。
   2. 可能导致某个请求不被处理。
   3. 客户端需要组装这个链条，耦合了客户端和链条的组成结构，可以把这个在客户端的组合动作提到外面，通过配置来做，会更好点。
4. 场景
   1. 一个系统的审批需要多个对象才能完成处理的情况下，例如请假系统等。
   2. 代码中存在多个if-else语句的情况下，此时可以考虑使用责任链模式来对代码进行重构
   3. 有多个对象可以处理同一个请求，具体哪个对象处理该请求有运行时刻自动确定。客户端只需将请求提交到链上，无须关心请求的处理对象是谁以及它是如何处理的。
   4. 不明确指定接受者的情况下，向多个对象中的一个提交一个请求。请求的发送者与请求者解耦，请求将沿着链进行传递，寻求响应的处理者。
   5. 可动态指定一组对象处理请求。客户端可以动态创建职责链来处理请求，还可以动态改变链中处理者之间的先后次序

##### Refer

###### .NET 场景：

1. 这种模式在处理UI的消息时很常用，但实际上Windows消息循环还是硬编码的结构。因为效率上的考虑，Windows消息循环是哪个对象有一个请求，则直接到达处理函数的地址。如果链条上的对象多了，而真正处理的函数在链条后部分，效率会很低下。因此我们在使用这种模式的时候更适合业务流程，即对性能要求不是特别高的情况更加常用。

##### Summary

这个模式也是为了解耦，解耦请求的发送者和接受者，当有新的需求的时候更容易变化，让我们的代码更符合面向对象OO的设计。

### 访问者模式Visitor

##### What

###### 引言

【访问者模式】，英文名称是：Visitor Pattern。如果按老规矩，先从名称上来看看这个模式，我根本不能获得任何对理解该模式有用的信息，而且这个模式在我们的编码生活中使用的并不是很多。该模式的意图定义很抽象，第一次看了这个定义其实和没看没有什么区别，一头雾水，为了让大家更好的理解该模式的初衷，我们举个例子来说明模式。比如：当我们为了解决一个新的软件需求的时候，经过多个日以继夜的努力，最终通过一个完美（自己认为的）的软件设计解决了客户提出的新的需求，而且这个设计有完美的类层次结构，并且是符合OO的设计原则的，我们很开心，对自己设计的东西很有成就感。又过了一段时间，客户突然又有了一个新的需求，需要为现有的类层次结构里面的类增加一个新的操作（其实就是一个方法），怎么办？好办，在面向OO设计模式中有一个模式就是为了解决这个问题的，那就是“访问者模式”，可以为现有的类层次结构中的类轻松增加新的操作

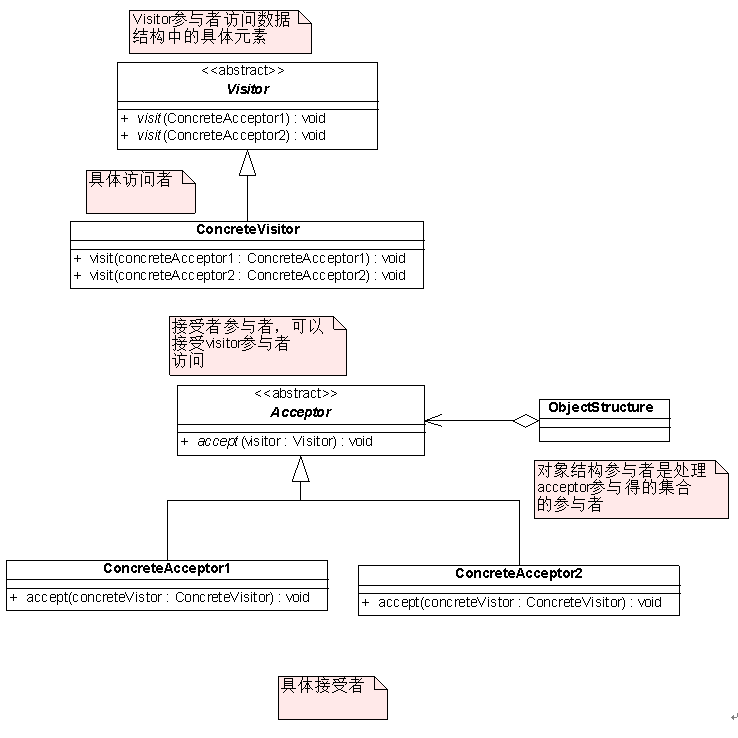
###### 动机（Motivate）

1. 在软件构建过程中，由于需求的改变，某些类层次结构中常常需要增加新的行为（方法），如果直接在基类中做这样的更改，将会给子类带来很繁重的变更负担，甚至破坏原有设计。如何在不更改类层次结构的前提下，在运行时根据需要透明地为类层次结构上的各个类动态添加新的操作，从而避免上述问题？

###### 意图（Intent）

1. 表示一个作用于某对象结构中的各个元素的操作。它可以在不改变各元素的类的前提下定义作用于这些元素的新的操作。
2. 数据结构里存储了很多个元素，假设现在要对所有元素进行一项“处理”。那么这项处理的程序代码应该写在哪里？以常理来判断，应该写在表示数据结构的类里面， 不过如果这项“处理”的行为不只一个的话，该怎么办？每次要做新处理的时候，就必须修改数据结构的类。
3. Visitor Pattern把数据结构和处理两者分开，另外写一个表示在数据内穿梭来去的主体“访客”类，然后把处理交给这个类来进行。如此一业，如果想追加新的处理行为时，只要再建立一个新的“访客”即可，而在数据结构这边，也只要能接受来敲门的“访客”就能完成动作。

###### 结构图（Structure）



###### 组成

1. **抽象访问者角色（Vistor）**: 声明一个包括多个访问操作，多个操作针对多个具体节点角色（可以说有多少个具体节点角色就有多少访问操作），使得所有具体访问者必须实现的接口。
2. **具体访问者角色（ConcreteVistor）**：实现抽象访问者角色中所有声明的接口，也可以说是实现对每个具体节点角色的新的操作。
3. **抽象节点角色（Element）**：声明一个接受操作，接受一个访问者对象作为参数，如果有其他参数，可以在这个“接受操作”里在定义相关的参数。
4. **具体节点角色（ConcreteElement）**：实现抽象元素所规定的接受操作。
5. **结构对象角色（ObjectStructure）**：节点的容器，可以包含多个不同类或接口的容器。

##### How

###### 实现方法

1. **Component**
   1. ）

###### 实现要点

1. 要点
   1. Visitor模式通过所谓双重分发（double dispatch）来实现在不更改Element类层次结构的前提下，在运行时透明地为类层次结构上的各个类动态添加新的操作。所谓双重分发即Visitor模式中间包括了两个多态分发（注意其中的多态机制）：第一个为accept方法的多态辨析；第二个为visit方法的多态辨析。
   2. 设计模式其实是一种堵漏洞的方式，但是没有一种设计模式能够堵完所有的漏洞，即使是组合各种设计模式也是一样。每个设计模式都有漏洞，都有它们解决不了的情况或者变化。每一种设计模式都假定了某种变化，也假定了某种不变化。Visitor模式假定的就是操作变化，而Element类层次结构稳定。
2. 优点
   1. 访问者模式使得添加新的操作变得容易。如果一些操作依赖于一个复杂的结构对象的话，那么一般而言，添加新的操作会变得很复杂。而使用访问者模式，增加新的操作就意味着添加一个新的访问者类。因此，使得添加新的操作变得容易。
   2. 访问者模式使得有关的行为操作集中到一个访问者对象中，而不是分散到一个个的元素类中。这点类似与”中介者模式”。
   3. 访问者模式可以访问属于不同的等级结构的成员对象，而迭代只能访问属于同一个等级结构的成员对象。
3. 缺点
   1. 增加新的元素类变得困难。每增加一个新的元素意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个具体访问者类中添加相应的具体操作。具体来说，Visitor模式的最大缺点在于扩展类层次结构（增添新的Element子类），会导致Visitor类的改变。因此Visitor模式适用于“Element类层次结构稳定，而其中的操作却经常面临频繁改动”。
4. 场景
   1. 如果系统有比较稳定的数据结构，而又有易于变化的算法时，此时可以考虑使用访问者模式。因为访问者模式使得算法操作的添加比较容易。
   2. 如果一组类中，存在着相似的操作，为了避免出现大量重复的代码，可以考虑把重复的操作封装到访问者中。（当然也可以考虑使用抽象类了）
   3. 如果一个对象存在着一些与本身对象不相干，或关系比较弱的操作时，为了避免操作污染这个对象，则可以考虑把这些操作封装到访问者对象中。

##### Refer

###### .NET 场景：

1. 。

##### Summary

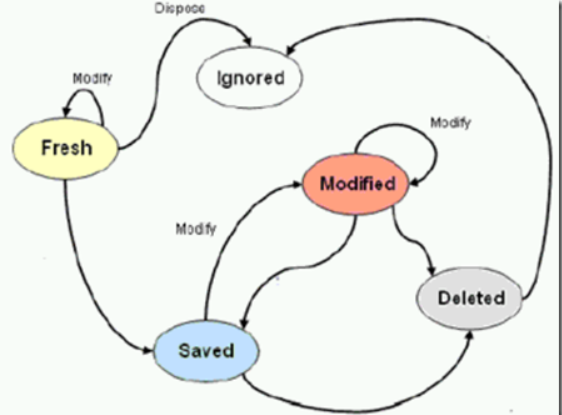
对

### 备忘录模式Memento

##### What

###### 引言

【备忘录模式】，英文名称是：Memento Pattern。按老规矩，先从名称上来看看这个模式，个人的最初理解就是对某个对象的状态进行保存，等到需要恢复的时候，可以从备忘录中进行恢复。生活中这样的例子也能经常看到，如备份电话通讯录，备份操作操作系统，备份数据库等。如果我们想恢复对象的状态，那么我们可能首先想到的是把对象保存下来，但是这样会破坏对象的封装性。因为对象有状态有操作，如果我们为了保存对象而留着原来的对象，做一个深拷贝，那么其他对象也能通过这个对象的接口访问这个对象状态，这并不是我们所希望的。而我们需要它的职责只是保存和恢复对象状态，而不应在上面支持对对象状态访问的接口，这就产生了Memento模式。



上图，一个对象肯定会有很多状态，这些状态肯定会相互转变而促进对象的发展，如果要想在某一时刻把当前对象回复到以前某一时刻的状态，这个情况用“备忘录模式”就能很好解决该问题。

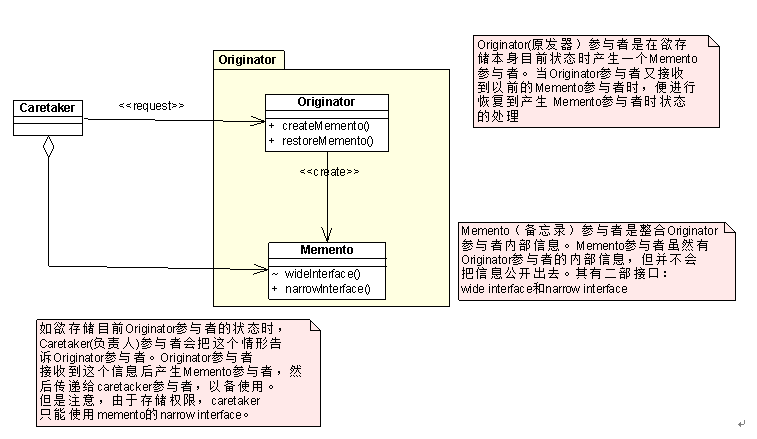
###### 动机（Motivate）

1. 在软件构建过程中，某些对象的状态在转换的过程中，可能由于某种需要，要求程序能够回溯到对象之前处于某个点时的状态。如果使用一些公有接口来让其他对象得到对象的状态，便会暴露对象的细节实现。
2. 如何实现对象状态的良好保存与恢复，但同时又不会因此而破坏对象本身的封装性？

###### 意图（Intent）

1. 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态（如果没有这个关键点，其实深拷贝就可以解决问题）。这样以后就可以将该对象恢复到原先保存的状态。
2. 在文本软件的操作环境下，即使不小心误删某些文字内容，只要利用恢复（undo）功能就能救顺删除前的内容。有些文本软件甚至支持一次以上的恢复行为。面向对象程序如果要执行复原，必须要能自由存取对象实例内部的状态，才能还原对象实例。不过如果对存取行为毫不设限，又会让高度依赖类内部结构的程序代码分散到程序各处，增加修改类时的困难。这称为封装性被破坏。
3. 加入表示对象实例状态的功能，而能在执行存储和恢复时不破坏封装性，这就是Memento Pattern的目的。利用Memento Pattern可以程序进行以下操作：undo（恢复）、redo(重复)、history(产生操作记录)、snapshot(存储目前状态)
4. Memento Pattern就是这样的Design Pattern。它会把某个时间点的对象实例状态记录存储起来，等到以后再让对象实例复原到当时的状态。

###### 结构图（Structure）



###### 组成

1. 任。

##### How

###### 实现方法

1. **发起人角色（Originator）：**记录当前时刻的内部状态，负责创建和恢复备忘录数据。负责创建一个备忘录Memento，用以记录当前时刻自身的内部状态，并可使用备忘录恢复内部状态。Originator【发起人】可以根据需要决定Memento【备忘录】存储自己的哪些内部状态。
2. **备忘录角色（Memento）：**负责存储发起人对象的内部状态，在进行恢复时提供给发起人需要的状态，并可以防止Originator以外的其他对象访问备忘录。备忘录有两个接口：Caretaker【管理角色】只能看到备忘录的窄接口，他只能将备忘录传递给其他对象。Originator【发起人】却可看到备忘录的宽接口，允许它访问返回到先前状态所需要的所有数据。
3. **管理者角色（Caretaker）：**负责保存备忘录对象。负责备忘录Memento，不能对Memento的内容进行访问或者操作。

###### 实现要点

1. 要点
   1. 备忘录（Memento）存储原发器（Originator）对象的内部状态，在需要时恢复原发器状态。Memento模式适用于“由原发器管理，却又必须存储在原发器之外的信息”。
   2. 在实现Memento模式中，要防止原发器以外的对象访问备忘录对象。备忘录对象有两个接口，一个为原发器使用的宽接口；一个为其他对象使用的窄接口。在实现Memento模式时，要考虑拷贝对象状态的效率问题，如果对象开销比较大，可以采用某种增量式改变（即只记住改变的状态）来改进Memento模式。
   3. 我们也可以用序列化的方式实现备忘录。序列化之后，我们可以把它临时性保存到数据库、文件、进程内、进程外等地方。
2. 优点
   1. 如果某个操作错误地破坏了数据的完整性，此时可以使用备忘录模式将数据恢复成原来正确的数据。
   2. 备份的状态数据保存在发起人角色之外，这样发起人就不需要对各个备份的状态进行管理。而是由备忘录角色进行管理，而备忘录角色又是由管理者角色管理，符合单一职责原则。
   3. 提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤，当新的状态无效或者存在问题时，可以使用先前存储起来的备忘录将状态复原。
   4. 实现了信息的封装，一个备忘录对象是一种原发器对象的表示，不会被其他代码改动，这种模式简化了原发器对象，备忘录只保存原发器的状态，采用堆栈来存储备忘录对象可以实现多次撤销操作，可以通过在负责人中定义集合对象来存储多个备忘录。
   5. 本模式简化了发起人类。发起人不再需要管理和保存其内部状态的一个个版本，客户端可以自行管理他们所需要的这些状态的版本。
   6. 当发起人角色的状态改变的时候，有可能这个状态无效，这时候就可以使用暂时存储起来的备忘录将状态复原。
3. 缺点
   1. 在实际的系统中，可能需要维护多个备份，需要额外的资源，这样对资源的消耗比较严重。资源消耗过大，如果类的成员变量太多，就不可避免占用大量的内存，而且每保存一次对象的状态都需要消耗内存资源，如果知道这一点大家就容易理解为什么一些提供了撤销功能的软件在运行时所需的内存和硬盘空间比较大了。
   2. 如果发起人角色的状态需要完整地存储到备忘录对象中，那么在资源消耗上面备忘录对象会很昂贵。
   3. 当负责人角色将一个备忘录 存储起来的时候，负责人可能并不知道这个状态会占用多大的存储空间，从而无法提醒用户一个操作是否很昂贵。
   4. 当发起人角色的状态改变的时候，有可能这个协议无效。如果状态改变的成功率不高的话，不如采取“假如”协议模式。
4. 场景
   1. 如果系统需要提供回滚操作时，使用备忘录模式非常合适。例如文本编辑器的Ctrl+Z撤销操作的实现，数据库中事务操作。
   2. 保存一个对象在某一个时刻的状态或部分状态，这样以后需要时它能够恢复到先前的状态。
   3. 如果用一个接口来让其他对象得到这些状态，将会暴露对象的实现细节并破坏对象的封装性，一个对象不希望外界直接访问其内部状态，通过负责人可以间接访问其内部状态。
   4. 有时一些发起人对象的内部信息必须保存在发起人对象以外的地方，但是必须要由发起人对象自己读取，这时，使用备忘录模式可以把复杂的发起人内部信息对其他的对象屏蔽起来，从而可以恰当地保持封装的边界。
5. 特性
   1. 为了确保备忘录的封装性，除了原发器外，其他类是不能也不应该访问备忘录类的，在实际开发中，原发器与备忘录之间的关系是非常特殊的，它们要分享信息而不让其他类知道，实现的方法因编程语言的不同而不同。
   2. **多备份实现**
      1. 在负责人中定义一个集合对象来存储多个状态，而且可以方便地返回到某一历史状态。
      2. 在备份对象时可以做一些记号，这些记号称为检查点(Check Point)。在使用HashMap等实现时可以使用Key来设置检查点。

##### Refer

###### .NET 场景：

##### Summary

我们是不是感觉Memento模式和Command模式有些类似，我们要仔细把握模式之间的异同，否则使用模式的时候就会出现张冠李戴的情况或者不能确定使用哪个模式好。Memento备忘录模式和Command命令模式其实还是有些细微的差别的，那就让我们来看看他们的异同吧。虽然两者都支持Undo操作，但是Command是对行为的封装，Memento是对对象状态的保留，这是目的上的不同。它们支持的也是Undo操作的不同层面，Command是对行为序列的操作，Memento是对行为状态的操作。命令模式保存的是发起人的具体命令（命令对应的是行为），而备忘录模式保存的是发起人的状态（而状态对应的数据结构，如属性）。

### 解释器模式Interpreter

##### What

###### 引言

在业务系统中写一个解释器的机会并不是很多，实践比较少，理解和应用该模式就有些困难，所以就放在最后来说。该模式就是【解释器模式】，英文名称是：Interpreter Pattern。按老规矩，先从名称上来看看这个模式，个人的最初理解“解释器”和Google的中英翻译功能类似。如果有一天你去国外旅游去了，比如去美国吧，美国人是讲英语的，我们是讲汉语的，如果英语听不懂，讲不好，估计沟通就完蛋了，不能沟通，估计玩的就很难尽兴了，因为有很多景点的解说你可能不明白（没有中文翻译的情况下，一般情况会有的）。所以我们需要一个软件，可以把中英文互译，那彼此就可以更好的理解对方的意思，我感觉翻译软件也可以称得上是解释器，把你不懂的解释成你能理解的。我们写代码，需要编译器把我们写的代码编译成机器可以理解的机器语言，从这方面来讲，C#的编译器也是一种解释器。

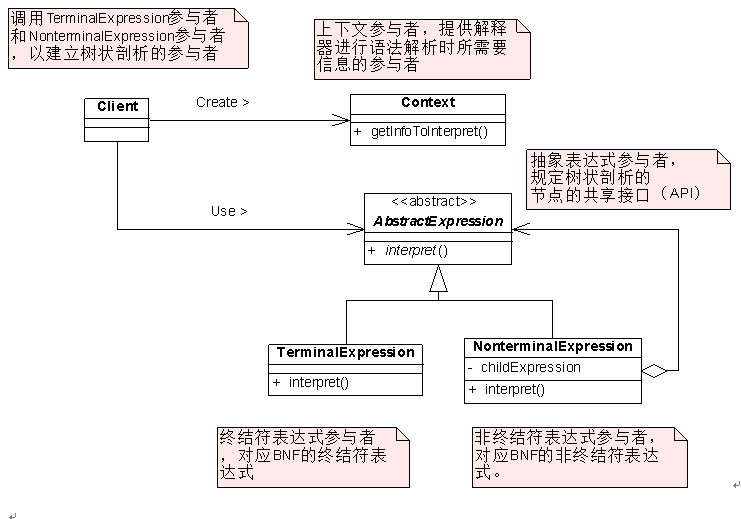
###### 动机（Motivate）

1. 在软件构建过程中，如果某一特定领域的问题比较复杂，类似的模式不断重复出现，如果使用普通的编程方式来实现将面临非常频繁的变化。在这种情况下，将特定领域的问题表达为某种语法规则下的句子，然后构建一个解释器来解释这样的句子，从而达到解决问题的目的。

###### 意图（Intent）

1. 给定一个语言，定义它的文法的一种表示，并定义一种解释器，这个解释器使用该表示来解释语言中的句子。
2. 创建解释性语言

###### 结构图（Structure）



###### 组成

1. **抽象表达式(AbstractExpression)**：定义解释器的接口，约定解释器的解释操作。其中的Interpret接口，正如其名字那样，它是专门用来解释该解释器所要实现的功能。
2. **终结符表达式(Terminal Expression)**：实现了抽象表达式角色所要求的接口，主要是一个interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应。比如有一个简单的公式R=R1+R2，在里面R1和R2就是终结符，对应的解析R1和R2的解释器就是终结符表达式。
3. **非终结符表达式(Nonterminal Expression)**：文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式R=R1+R2中，“+”就是非终结符，解析“+”的解释器就是一个非终结符表达式。
4. **环境角色(Context)**：这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如R=R1+R2，我们给R1赋值100，给R2赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了。
5. **客户端（Client）：**指的是使用解释器的客户端，通常在这里将按照语言的语法做的表达式转换成使用解释器对象描述的抽象语法树，然后调用解释操作。

##### How

###### 实现方法

1. 在很多场合都需要把数字转换成中文，我们就可以使用解释器来实现该功能，把给定的数字解释成符合语法规范的汉字表示法。
2. **Component**
   1. ）

###### 实现要点

1. 要点
   1. 使用Interpreter模式来表示文法规则，从而可以使用面向对象技巧方便地“扩展”文法。
   2. Interpreter模式比较适合简单的文法表示，对于复杂的文法表示，Interpreter模式会产生比较大的类层次结构，需要求助于语法分析生成器这样的标准工具。
2. 优点
   1. 易于改变和扩展文法。
   2. 每一条文法规则都可以表示为一个类，因此可以方便地实现一个简单的语言。
   3. 实现文法较为容易。在抽象语法树中每一个表达式节点类的实现方式都是相似的，这些类的代码编写都不会特别复杂，还可以通过一些工具自动生成节点类代码。
   4. 增加新的解释表达式较为方便。如果用户需要增加新的解释表达式只需要对应增加一个新的终结符表达式或非终结符表达式类，原有表达式类代码无须修改，符合“开闭原则”。
3. 缺点
   1. 对于复杂文法难以维护。在解释器模式中，每一条规则至少需要定义一个类，因此如果一个语言包含太多文法规则，类的个数将会急剧增加，导致系统难以管理和维护，此时可以考虑使用语法分析程序等方式来取代解释器模式。
   2. 执行效率较低。由于在解释器模式中使用了大量的循环和递归调用，因此在解释较为复杂的句子时其速度很慢，而且代码的调试过程也比较麻烦
4. 场景
   1. Interpreter模式的应用场合是Interpreter模式应用中的难点，只有满足“业务规则频繁变化，且类似的模式不断重复出现，并且容易抽象为语法规则的问题”才适合使用Interpreter模式。
   2. 当一个语言需要解释执行，并可以将该语言中的句子表示为一个抽象语法树的时候，可以考虑使用解释器模式（如XML文档解释、正则表达式等领域）
   3. 一些重复出现的问题可以用一种简单的语言来进行表达。
   4. 一个语言的文法较为简单.
   5. 当执行效率不是关键和主要关心的问题时可考虑解释器模式（注：高效的解释器通常不是通过直接解释抽象语法树来实现的，而是需要将它们转换成其他形式，使用解释器模式的执行效率并不高。

##### Refer

###### .NET 场景：

1. 正则表达式就是一个典型的解释器。ASP.NET中，把aspx文件转化为dll时，会对html语言进行处理，这个处理过程也包含了解释器的模式在里面。Interpreter模式其实有Composite模式的影子，但它们解决的问题是不一样的。

##### Summary

对

### 设计模式总结：

##### 创建型

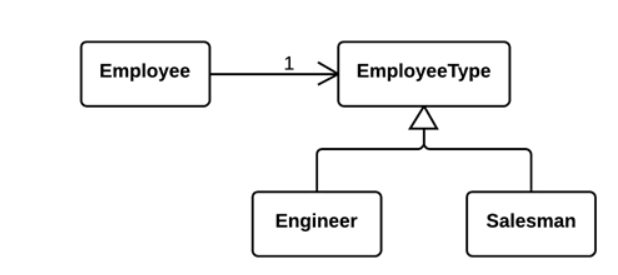
###### 定义

1. Singleton单件模式解决的是实体对象个数的问题。
2. 除了Singleton之外，其他创建型模式解决的都是new所带来的耦合关系。
3. Factory Method，Abstract Factory，Builder都需要一个额外的工厂类来负责实例化“易变对象”
4. 而Prototype则是通过原型（一个特殊的工厂类）来克隆“易变对象”。 （其实原型就是一个特殊的工厂类，它只是把工厂和实体对象耦合在一起了）
5. 如果遇到“易变类”，起初的设计通常从Factory Method开始，当遇到更多的复杂变化时，再考虑重构为其他三种工厂模式（Abstract Factory，Builder，Prototype）。
6. 一般来说，如果可以使用Factory Method，那么一定可以使用Prototype。但是Prototype的使用情况一般是在类比较容易克隆的条件之上，如果是每个类实现比较简单，都可以只用实现MemberwiseClone，没有引用类型的深拷贝，那么就更适合了。

##### 问题解析和修正

###### Switch Case 和 多重 If elseif语句

1. 如果是根据类型码进行选择，要的是“与该类型码相关的函数或类”
   1. 运用  提炼函数(Extract Method)  将  switch  语句提炼到一个独立函数中，再以  搬移函数(Move Method)  将它搬移到需要多态性的那个类里。
2. 基于类型码来识别分支
   1. 以子类取代类型码(Replace Type Code with Subclass)  或  以状态/策略模式取代类型码(Replace Type Code with State/Strategy)



1. 分支并不多并且它们使用不同参数调用相同的函数，多态就没必要了。在这种情况下，你可以运用  以明确函数取代参数(Replace Parameter with Explicit Methods)  。
2. 选择条件之一是 null，可以运用  引入 Null 对象(Introduce Null Object)
3. 收益
   1. 提升代码组织性。
4. 忽略
   1. 如果一个  switch  操作只是执行简单的行为，就没有重构的必要了。
   2. switch  常被工厂设计模式族（ 工厂方法模式(Factory Method) 和 抽象工厂模式(Abstract Factory) ）所使用，这种情况下也没必要重构

###### Switch Case 和 多重 If elseif语句

###### Switch Case 和 多重 If elseif语句

# Algorithm && Data Structure

### Design Pattern

##### 定义

###### 定义

1. 设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。

##### 原则

###### 单一职责原则（Single Responsibility Principle，SRP）

1. 类的职责要单一。就一个类而言，应该仅有一个引起它变化的原因。
2. 如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其它职责的能力。
3. 举个简单的例子，我们现在很多时候都直接使用手机拍照，但新闻、时尚杂志等的拍照使用的是相机。手机其实就相当于把相机耦合到手机中去了，而相机却只有拍照这一个职责。大多数时候，一件产品简单一些，职责单一一些，或许是更好的选择，这就是单一职责原则。

###### 开闭原则（Open Close Principle，OCP）

1. 软件实体可扩展，但不可修改。
2. 就好比“一国两制”，一个国家有两种制度，没有修改原有的管理制度，而是增加了一种新的制度。正所谓对于扩展的开放的，对于更改是封闭的。

###### 依赖倒转原则（Dependence Inversion Principle，DIP）

1. 高层模块不应该依赖底层模块，两个都应该依赖抽象；抽象不应该依赖细节，细节应该依赖抽象。后面这句话换个方式说就是针对接口编程，不要对实现编程。
2. 举个例子，就好比你的电脑，如果CPU、内存、硬盘都需要依赖具体的主板，主板坏了，所有的部件都不能用了，这显然是不合理的。
3. 换句话说，即谁也不要依赖谁，除了约定的接口，大家都可以灵活自如。
4. 建议：
   1. 没有变量引用具体的类(可已使用工厂代替创建这个具体的类)