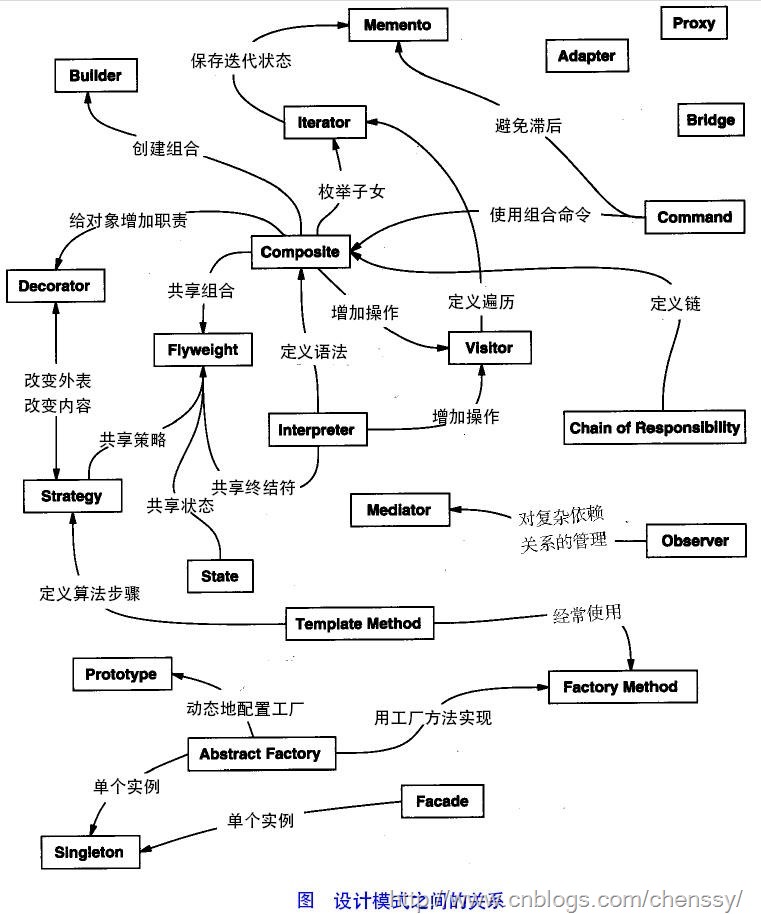
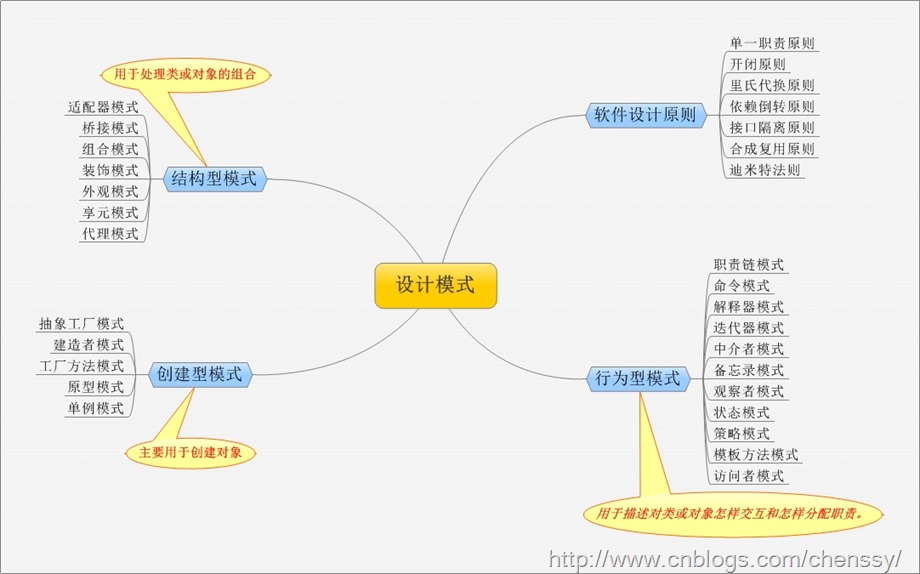
**设计模式总结**

[](http://images.cnitblog.com/blog/381060/201310/08191337-7d7eee98b3c94f4db2e22c35089286a8.jpg)

      设计模式总概况：

[](http://images.cnitblog.com/blog/381060/201310/08191338-0176aa5a338e4b31bf6ef5e83746ea8d.jpg)

**一、设计原则**

**1、单一职责原则**

      一个类，只有一个引起它变化的原因。应该只有一个职责。每一个职责都是变化的一个轴线，如果一个类有一个以上的职责，这些职责就耦合在了一起。这会导致脆弱的设计。当一个职责发生变化时，可能会影响其它的职责。另外，多个职责耦合在一起，会影响复用性。例如：要实现逻辑和界面的分离。from：百度百科

**2、开闭原则（Open Close Principle）**

      开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**3、里氏代换原则（Liskov Substitution Principle）**

      里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。from：百度百科

**4、依赖倒转原则（Dependence Inversion Principle）**

      所谓依赖倒置原则（Dependence Inversion Principle）就是要依赖于抽象，不要依赖于具体。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

      实现开闭原则的关键是抽象化，并且从抽象化导出具体化实现，如果说开闭原则是面向对象设计的目标的话，那么依赖倒转原则就是面向对象设计的主要手段。 from：百度百科

**5、接口隔离原则（Interface Segregation Principle）**

      这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

**6、合成复用原则（Composite Reuse Principle）**

      合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。简言之：要尽量使用组合/聚合关系，少用继承。

**7、迪米特法则（最少知道原则）（Demeter Principle）**

      为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。也就是说一个软件实体应当尽可能少的与其他实体发生相互作用。这样，当一个模块修改时，就会尽量少的影响其他的模块，扩展会相对容易，这是对软件实体之间通信的限制，它要求限制软件实体之间通信的宽度和深度。

**二、创建型模式**

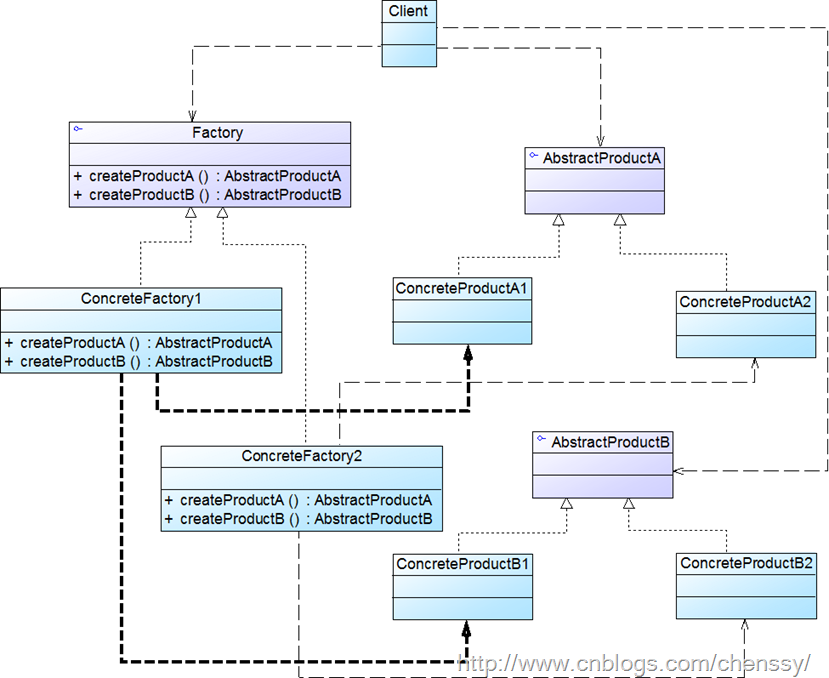
      在软件工程中，创建型模式是处理对象创建的设计模式，试图根据实际情况使用合适的方式创建对象。基本的对象创建方式可能会导致设计上的问题，或增加设计的复杂度。创建型模式通过以某种方式控制对象的创建来解决问题。

      创建型模式由两个主导思想构成。一是将系统使用的具体类封装起来，二是隐藏这些具体类的实例创建和结合的方式。

      创建型模式又分为对象创建型模式和类创建型模式。对象创建型模式处理对象的创建，类创建型模式处理类的创建。详细地说，对象创建型模式把对象创建的一部分推迟到另一个对象中，而类创建型模式将它对象的创建推迟到子类中。

**1、抽象工厂模式(Abstract Factory)**

      所谓抽象工厂模式就是她提供一个接口，用于创建相关或者依赖对象的家族，而不需要明确指定具体类。他允许客户端使用抽象的接口来创建一组相关的产品，而不需要关系实际产出的具体产品是什么。这样一来，客户就可以从具体的产品中被解耦。它的优点是隔离了具体类的生成，使得客户端不需要知道什么被创建了，而缺点就在于新增新的行为会比较麻烦，因为当添加一个新的产品对象时，需要更加需要更改接口及其下所有子类。其UML结构图如下：

[](http://images.cnitblog.com/blog/381060/201310/08191339-cd66eb20e87a4dacb57424106cc95899.png)

      参与者：

         AbstractFactory：抽象工厂。抽象工厂定义了一个接口，所有的具体工厂都必须实现此接口，这个接口包含了一组方法用来生产产品。

         ConcreteFactory：具体工厂。具体工厂是用于生产不同产品族。要创建一个产品，客户只需要使用其中一个工厂完全不需要实例化任何产品对象。

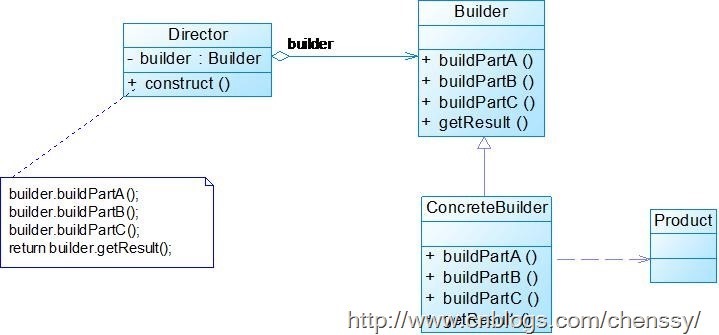
         AbstractProduct：抽象产品。这是一个产品家族，每一个具体工厂都能够生产一整组产品。

         Product：具体产品。

**2、建造者模式(Builder)**

      对于建造者模式而已，它主要是将一个复杂对象的构建与表示分离，使得同样的构建过程可以创建不同的表示。适用于那些产品对象的内部结构比较复杂。

      建造者模式将复杂产品的构建过程封装分解在不同的方法中，使得创建过程非常清晰，能够让我们更加精确的控制复杂产品对象的创建过程，同时它隔离了复杂产品对象的创建和使用，使得相同的创建过程能够创建不同的产品。但是如果某个产品的内部结构过于复杂，将会导致整个系统变得非常庞大，不利于控制，同时若几个产品之间存在较大的差异，则不适用建造者模式，毕竟这个世界上存在相同点大的两个产品并不是很多，所以它的使用范围有限。其UML结构图：

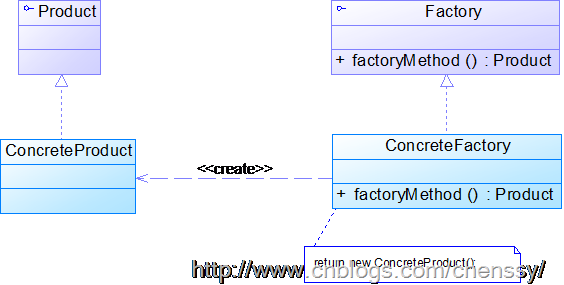
[](http://images.cnitblog.com/blog/381060/201310/08191341-9e989f409fce4896affe62c2fac36298.jpg)       参与者：

         Builder：抽象建造者。它声明为创建一个Product对象的各个部件指定的抽象接口。   
         ConcreteBuilder：具体建造者。实现抽象接口，构建和装配各个部件。   
         Director：指挥者。构建一个使用Builder接口的对象。它主要是用于创建一个复杂的对象，它主要有两个作用，一是：隔离了客户与对象的生产过程，二是：负责控制产品对象的生产过程。   
         Product：产品角色。一个具体的产品对象。

**3、工厂方法模式(Factory Method)**

      作为抽象工厂模式的孪生兄弟，工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个，也就是说工厂方法模式让实例化推迟到子类。

      工厂方法模式非常符合“开闭原则”，当需要增加一个新的产品时，我们只需要增加一个具体的产品类和与之对应的具体工厂即可，无须修改原有系统。同时在工厂方法模式中用户只需要知道生产产品的具体工厂即可，无须关系产品的创建过程，甚至连具体的产品类名称都不需要知道。虽然他很好的符合了“开闭原则”，但是由于每新增一个新产品时就需要增加两个类，这样势必会导致系统的复杂度增加。其UML结构图：

[](http://images.cnitblog.com/blog/381060/201310/08191344-c19aecfcfd1a4a63bb7a2aa698cb25c7.png)

      参与者：

          Product：抽象产品。所有的产品必须实现这个共同的接口，这样一来，使用这些产品的类既可以引用这个接口。而不是具体类 。

         ConcreteProduct：具体产品。

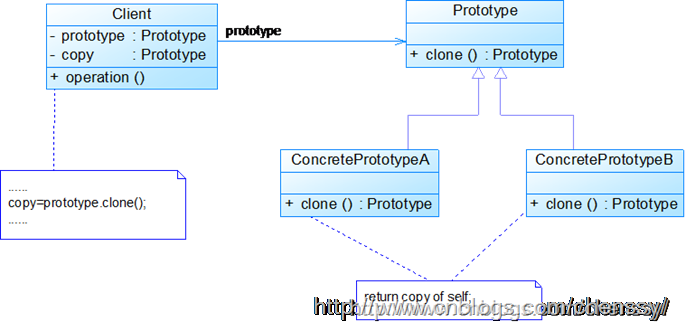
         Creator：抽象工厂。它实现了所有操纵产品的方法，但不实现工厂方法。Creator所有的子类都必须要实现factoryMethod()方法。

         ConcreteCreator：具体工厂。制造产品的实际工厂。它负责创建一个或者多个具体产品，只有ConcreteCreator类知道如何创建这些产品。

**4、原型模式(Prototype)**

      在我们应用程序可能有某些对象的结构比较复杂，但是我们又需要频繁的使用它们，如果这个时候我们来不断的新建这个对象势必会大大损耗系统内存的，这个时候我们需要使用原型模式来对这个结构复杂又要频繁使用的对象进行克隆。所以原型模式就是用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。

      它主要应用与那些创建新对象的成本过大时。它的主要优点就是简化了新对象的创建过程，提高了效率，同时原型模式提供了简化的创建结构。UML结构图：

[](http://images.cnitblog.com/blog/381060/201310/08191346-90bc8cd199e14f10add89470058b3a9e.png)

       参与者：

          Prototype：抽象原型类。声明克隆自身的接口。   
          ConcretePrototype：具体原型类。实现克隆的具体操作。   
          Client：客户类。让一个原型克隆自身，从而获得一个新的对象。

**5、单例模式(Singleton)**

      单例模式，从字面上看就是一个实例的意思。所以它的定义就是确保某一个类只有一个实例，并且提供一个全局访问点。

      单例模式具备如下几个特点：

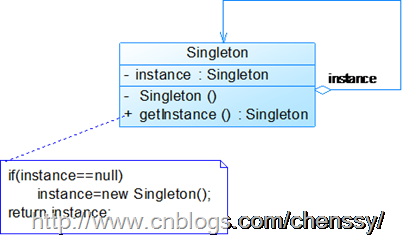
          1、只有一个实例。

          2、能够自我实例化。

          3、提供全局访问点。

      所以说当系统中只需要一个实例对象或者系统中只允许一个公共访问点，除了这个公共访问点外，不能通过其他访问点访问该实例时，可以使用单例模式。

       单例模式的主要优点就是节约系统资源、提高了系统效率，同时也能够严格控制客户对它的访问。也许就是因为系统中只有一个实例，这样就导致了单例类的职责过重，违背了“单一职责原则”，同时也没有抽象类，所以扩展起来有一定的困难。其UML结构图非常简单，就只有一个类：

[](http://images.cnitblog.com/blog/381060/201310/08191349-f6174c940aa44d86a765ac983778f699.png)

      参与者：

         Singleton：单例。

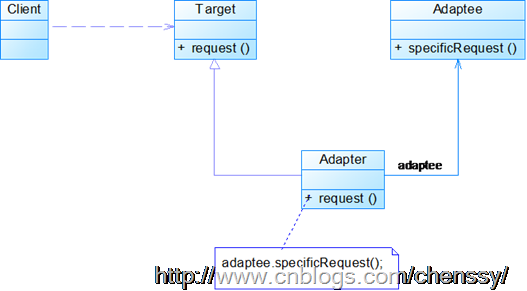
**三、结构型模式**

      结构型模式主要是用于处理类或者对象的组合，它描述了如何来类或者对象更好的组合起来，是从程序的结构上来解决模块之间的耦合问题。它主要包括适配器模式、桥接模式、组合模式、装饰模式、外观模式、享元模式、代理模式这个七个模式。

**1、适配器模式(Adapter)**

      在我们的应用程序中我们可能需要将两个不同接口的类来进行通信，在不修改这两个的前提下我们可能会需要某个中间件来完成这个衔接的过程。这个中间件就是适配器。所谓适配器模式就是将一个类的接口，转换成客户期望的另一个接口。它可以让原本两个不兼容的接口能够无缝完成对接。

      作为中间件的适配器将目标类和适配者解耦，增加了类的透明性和可复用性。

[](http://images.cnitblog.com/blog/381060/201310/08191352-3465244010d94b7ea6a5e3f769fb0a04.png)

      参与者：

         Target：目标抽象类 。

         Adapter：适配器类 。通过在内部包装一个Adaptee，将源接口转成目标接口。

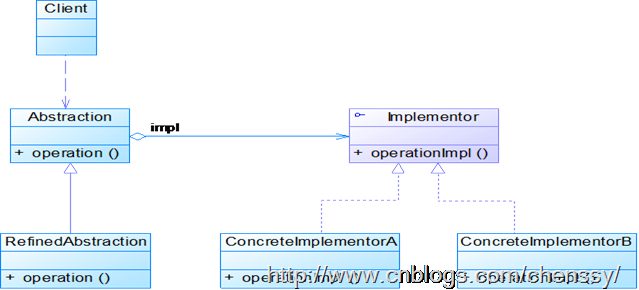
         Adaptee：适配者类 。需要适配的类。

         Client：客户类。

**2、桥接模式(Bridge)**

      如果说某个系统能够从多个角度来进行分类，且每一种分类都可能会变化，那么我们需要做的就是讲这多个角度分离出来，使得他们能独立变化，减少他们之间的耦合，这个分离过程就使用了桥接模式。所谓桥接模式就是讲抽象部分和实现部分隔离开来，使得他们能够独立变化。

      桥接模式将继承关系转化成关联关系，封装了变化，完成了解耦，减少了系统中类的数量，也减少了代码量。

[](http://images.cnitblog.com/blog/381060/201310/08191400-d1ce6d14e6cc4b679e24e2e38ee91643.png)

      参与者

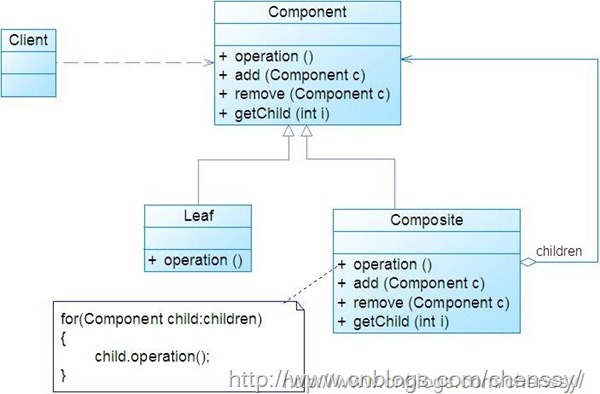
         Abstraction：抽象类。   
         RefinedAbstraction：扩充抽象类。   
         Implementor：实现类接口。   
         ConcreteImplementor：具体实现类 。

**3、组合模式(Composite)**

      组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次。它定义了如何将容器对象和叶子对象进行递归组合，使得客户在使用的过程中无须进行区分，可以对他们进行一致的处理。

      在使用组合模式中需要注意一点也是组合模式最关键的地方：叶子对象和组合对象实现相同的接口。这就是组合模式能够将叶子节点和对象节点进行一致处理的原因。

      虽然组合模式能够清晰地定义分层次的复杂对象，也使得增加新构件也更容易，但是这样就导致了系统的设计变得更加抽象，如果系统的业务规则比较复杂的话，使用组合模式就有一定的挑战了。

[](http://images.cnitblog.com/blog/381060/201310/08191402-ad0213120a1845b9b8564af6e00c1c1b.jpg)

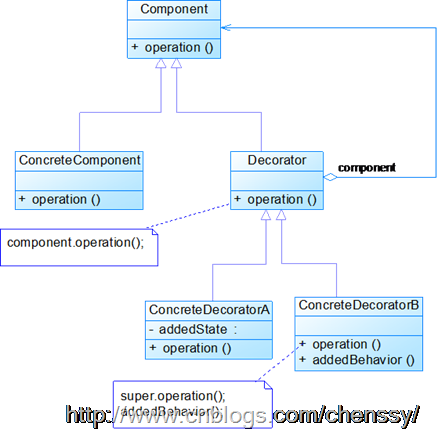
      参与者：

         Component ：组合中的对象声明接口，在适当的情况下，实现所有类共有接口的默认行为。声明一个接口用于访问和管理Component子部件。   
         Leaf：叶子对象。叶子结点没有子结点。   
         Composite：容器对象，定义有枝节点行为，用来存储子部件，在Component接口中实现与子部件有关操作，如增加(add)和删除(remove)等。

**4、装饰者模式(Decorator)**

      我们可以通过继承和组合的方式来给一个对象添加行为，虽然使用继承能够很好拥有父类的行为，但是它存在几个缺陷：一、对象之间的关系复杂的话，系统变得复杂不利于维护。二、容易产生“类爆炸”现象。三、是静态的。在这里我们可以通过使用装饰者模式来解决这个问题。

      装饰者模式，动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更加有弹性的替代方案。虽然装饰者模式能够动态将责任附加到对象上，但是他会产生许多的细小对象，增加了系统的复杂度。

[](http://images.cnitblog.com/blog/381060/201310/08191403-9fbb57e09b5a40e489a713f90b8f145a.png)

      参与者：

         Component: 抽象构件。是定义一个对象接口，可以给这些对象动态地添加职责。

         ConcreteComponent:具体构件。是定义了一个具体的对象，也可以给这个对象添加一些职责。

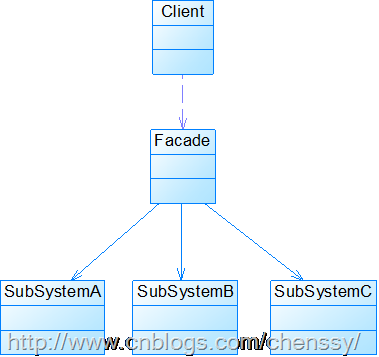
         Decorator: 抽象装饰类。是装饰抽象类，继承了Component,从外类来扩展Component类的功能，但对于Component来说，是无需知道Decorator存在的。

         ConcreteDecorator:具体装饰类，起到给Component添加职责的功能。

**5、外观模式(Facade)**

      我们都知道类与类之间的耦合越低，那么可复用性就越好，如果两个类不必彼此通信，那么就不要让这两个类发生直接的相互关系，如果需要调用里面的方法，可以通过第三者来转发调用。外观模式非常好的诠释了这段话。外观模式提供了一个统一的接口，用来访问子系统中的一群接口。它让一个应用程序中子系统间的相互依赖关系减少到了最少，它给子系统提供了一个简单、单一的屏障，客户通过这个屏障来与子系统进行通信。

      通过使用外观模式，使得客户对子系统的引用变得简单了，实现了客户与子系统之间的松耦合。但是它违背了“开闭原则”，因为增加新的子系统可能需要修改外观类或客户端的源代码。

[](http://images.cnitblog.com/blog/381060/201310/08191405-c300a2b1a31a4e579fcfca4be38b8113.png)

      参与者：

         Facade: 外观角色。知道哪些子系统类负责处理请求，将客户的请求代理给适合的子系统处理。

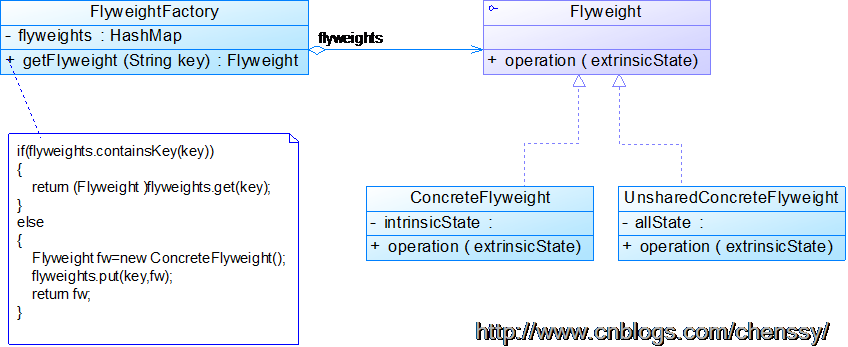
         SubSystem:子系统角色。实现子系统功能，处理Facade对象发来的请求。

**6、享元模式(Flyweight)**

      在一个系统中对象会使得内存占用过多，特别是那些大量重复的对象，这就是对系统资源的极大浪费。享元模式对对象的重用提供了一种解决方案，它使用共享技术对相同或者相似对象实现重用。

      享元模式就是运行共享技术有效地支持大量细粒度对象的复用。系统使用少量对象,而且这些都比较相似，状态变化小，可以实现对象的多次复用。这里有一点要注意：享元模式要求能够共享的对象必须是细粒度对象。

      享元模式通过共享技术使得系统中的对象个数大大减少了，同时享元模式使用了内部状态和外部状态，同时外部状态相对独立，不会影响到内部状态，所以享元模式能够使得享元对象在不同的环境下被共享。同时正是分为了内部状态和外部状态，享元模式会使得系统变得更加复杂，同时也会导致读取外部状态所消耗的时间过长。

[](http://images.cnitblog.com/blog/381060/201310/08191406-32f110ca782745c39c0de446985616aa.png)

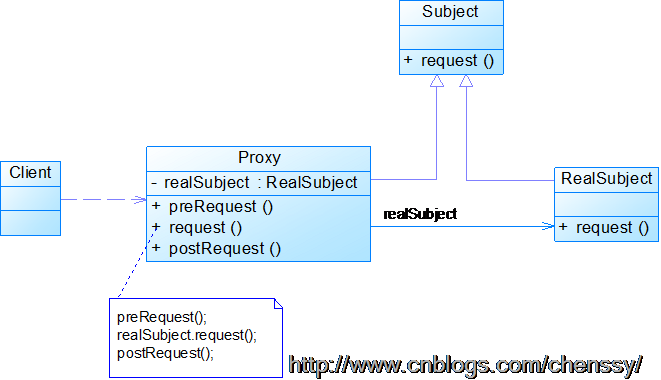
      参与者：

         Flyweight: 抽象享元类。所有具体享元类的超类或者接口，通过这个接口，Flyweight可以接受并作用于外部专题。   
         ConcreteFlyweight: 具体享元类。指定内部状态，为内部状态增加存储空间。   
         UnsharedConcreteFlyweight: 非共享具体享元类。指出那些不需要共享的Flyweight子类。   
         FlyweightFactory: 享元工厂类。用来创建并管理Flyweight对象，它主要用来确保合理地共享Flyweight，当用户请求一个Flyweight时，FlyweightFactory就会提供一个已经创建的Flyweight对象或者新建一个（如果不存在）。

**7、代理模式(Proxy)、**

      代理模式就是给一个对象提供一个代理，并由代理对象控制对原对象的引用。它使得客户不能直接与真正的目标对象通信。代理对象是目标对象的代表，其他需要与这个目标对象打交道的操作都是和这个代理对象在交涉。

      代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了的作用和保护了目标对象的，同时也在一定程度上面减少了系统的耦合度。

[](http://images.cnitblog.com/blog/381060/201310/08191407-2f55b888b2124d6cab9ea52cfe1850ec.png)

      参与者：

         Subject: 抽象角色。声明真实对象和代理对象的共同接口。

         Proxy: 代理角色。代理对象与真实对象实现相同的接口，所以它能够在任何时刻都能够代理真实对象。代理角色内部包含有对真实对象的引用，所以她可以操作真实对象，同时也可以附加其他的操作，相当于对真实对象进行封装。

         RealSubject: 真实角色。它代表着真实对象，是我们最终要引用的对象。

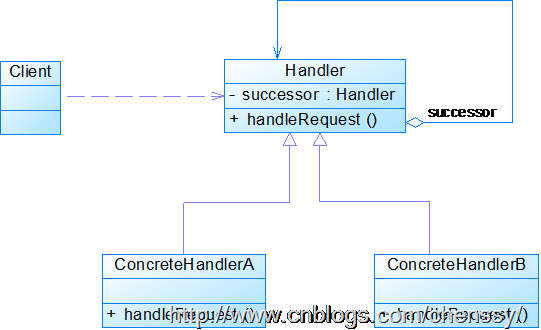
**四、行为型模式**

      行为型模式主要是用于描述类或者对象是怎样交互和怎样分配职责的。它涉及到算法和对象间的职责分配，不仅描述对象或者类的模式，还描述了他们之间的通信方式，它将你的注意力从控制流转移到了对象间的关系上来。行为型类模式采用继承机制在类间分派行为，而行为型对象模式使用对象复合而不是继承。它主要包括如何11中设计模式：职责链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式、访问者模式。

**1、职责链模式(Chain of Responsibility)**

       职责链模式描述的请求如何沿着对象所组成的链来传递的。它将对象组成一条链，发送者将请求发给链的第一个接收者，并且沿着这条链传递，直到有一个对象来处理它或者直到最后也没有对象处理而留在链末尾端。

      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收请求，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，这就是职责链模式。在职责链模式中，使得每一个对象都有可能来处理请求，从而实现了请求的发送者和接收者之间的解耦。同时职责链模式简化了对象的结构，它使得每个对象都只需要引用它的后继者即可，而不必了解整条链，这样既提高了系统的灵活性也使得增加新的请求处理类也比较方便。但是在职责链中我们不能保证所有的请求都能够被处理，而且不利于观察运行时特征。

[](http://images.cnitblog.com/blog/381060/201310/08191408-bf7f47bd6030450eb948b826fcde7733.png)

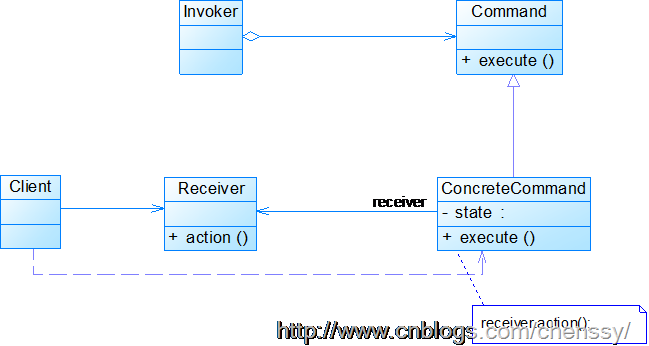
      参与者：

         Handler: 抽象处理者。定义了一个处理请求的方法。所有的处理者都必须实现该抽象类。   
         ConcreteHandler: 具体处理者。处理它所负责的请求，同时也可以访问它的后继者。如果它能够处理该请求则处理，否则将请求传递到它的后继者。   
         Client: 客户类。

**2、命令模式(Command)**

      有些时候我们想某个对象发送一个请求，但是我们并不知道该请求的具体接收者是谁，具体的处理过程是如何的，们只知道在程序运行中指定具体的请求接收者即可，对于这样将请求封装成对象的我们称之为命令模式。所以命令模式将请求封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。同时命令模式支持可撤销的操作。

      命令模式可以将请求的发送者和接收者之间实现完全的解耦，发送者和接收者之间没有直接的联系，发送者只需要知道如何发送请求命令即可，其余的可以一概不管，甚至命令是否成功都无需关心。同时我们可以非常方便的增加新的命令，但是可能就是因为方便和对请求的封装就会导致系统中会存在过多的具体命令类。

[](http://images.cnitblog.com/blog/381060/201310/08191409-2959f8b1fd534408ae3b786d7374ad6f.png)       参与者：

          Command: 抽象命令类。用来声明执行操作的接口。

          ConcreteCommand: 具体命令类。将一个接收者对象绑定于一个动作，调用接收者相应的操作，以实现Excute。

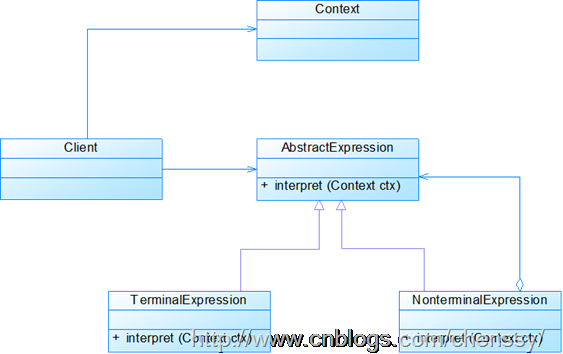
         Invoker: 调用者。要求该命令执行这个请求。

         Receiver: 接收者。知道如何实施与执行一个请求相关的操作，任何类都有可能成为一个接收者。

         Client:客户类。

**3、解释器模式(Interpreter)**

       所谓解释器模式就是定义语言的文法，并且建立一个解释器来解释该语言中的句子。解释器模式描述了如何构成一个简单的语言解释器，主要应用在使用面向对象语言开发的编译器中。它描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。

[](http://images.cnitblog.com/blog/381060/201310/08191409-2f771c9986024865aadbf6bfe561fa6b.png)

      参与者：

         AbstractExpression: 抽象表达式。声明一个抽象的解释操作，该接口为抽象语法树中所有的节点共享。

         TerminalExpression: 终结符表达式。实现与文法中的终结符相关的解释操作。实现抽象表达式中所要求的方法。文法中每一个终结符都有一个具体的终结表达式与之相对应。

         NonterminalExpression: 非终结符表达式。为文法中的非终结符相关的解释操作。

         Context: 环境类。包含解释器之外的一些全局信息。

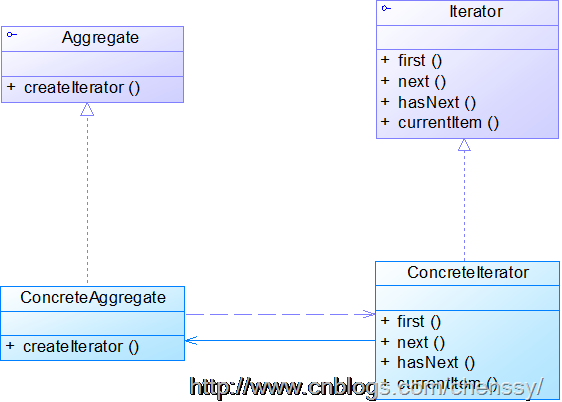
         Client: 客户类。

**4、迭代器模式(Iterator)**

     对于迭代在编程过程中我们经常用到，能够游走于聚合内的每一个元素，同时还可以提供多种不同的遍历方式，这就是迭代器模式的设计动机。在我们实际的开发过程中，我们可能会需要根据不同的需求以不同的方式来遍历整个对象，但是我们又不希望在聚合对象的抽象接口中充斥着各种不同的遍历操作，于是我们就希望有某个东西能够以多种不同的方式来遍历一个聚合对象，这时迭代器模式出现了。

      何为迭代器模式？所谓迭代器模式就是提供一种方法顺序访问一个聚合对象中的各个元素，而不是暴露其内部的表示。迭代器模式是将迭代元素的责任交给迭代器，而不是聚合对象，我们甚至在不需要知道该聚合对象的内部结构就可以实现该聚合对象的迭代。

      通过迭代器模式，使得聚合对象的结构更加简单，它不需要关注它元素的遍历，只需要专注它应该专注的事情，这样就更加符合单一职责原则了。

[](http://images.cnitblog.com/blog/381060/201310/08191411-dbc2a41a19314d2b8b9fb5d4e091d0ad.png)  
       参与者：

          Iterator: 抽象迭代器：所有迭代器都需要实现的接口，提供了游走聚合对象元素之间的方法。

          ConcreteIterator: 具体迭代器。利用这个具体的迭代器能够对具体的聚合对象进行遍历。每一个聚合对象都应该对应一个具体的迭代器。

          Aggregate: 抽象聚合类。

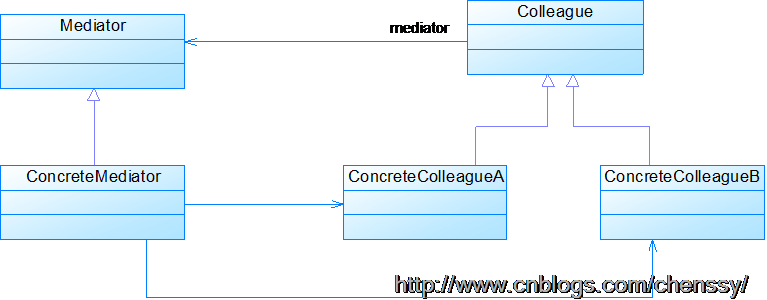
          ConcreteAggregate: 具体聚合类。实现creatorIterator()方法，返回该聚合对象的迭代器。

**5、中介者模式(Mediator)**

      租房各位都有过的经历吧！在这个过程中中介结构扮演着很重要的角色，它在这里起到一个中间者的作用，给我们和房主互相传递信息。在外面软件的世界里同样需要这样一个中间者。在我们的系统中有时候会存在着对象与对象之间存在着很强、复杂的关联关系，如果让他们之间有直接的联系的话，必定会导致整个系统变得非常复杂，而且可扩展性很差！在前面我们就知道如果两个类之间没有不必彼此通信，我们就不应该让他们有直接的关联关系，如果实在是需要通信的话，我们可以通过第三者来转发他们的请求。同样，这里我们利用中介者来解决这个问题。

      所谓中介者模式就是用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。在中介者模式中，中介对象用来封装对象之间的关系，各个对象可以不需要知道具体的信息通过中介者对象就可以实现相互通信。它减少了对象之间的互相关系，提供了系统可复用性，简化了系统的结构。

      在中介者模式中，各个对象不需要互相知道了解，他们只需要知道中介者对象即可，但是中介者对象就必须要知道所有的对象和他们之间的关联关系，正是因为这样就导致了中介者对象的结构过于复杂，承担了过多的职责，同时它也是整个系统的核心所在，它有问题将会导致整个系统的问题。所以如果在系统的设计过程中如果出现“多对多”的复杂关系群时，千万别急着使用中介者模式，而是要仔细思考是不是您设计的系统存在问题。

[](http://images.cnitblog.com/blog/381060/201310/08191412-6159f912f5284cc69d753717a8457e74.png)

      参与者：

         Mediator: 抽象中介者。定义了同事对象到中介者对象之间的接口。

         ConcreteMediator: 具体中介者。实现抽象中介者的方法，它需要知道所有的具体同事类，同时需要从具体的同事类那里接收信息，并且向具体的同事类发送信息。

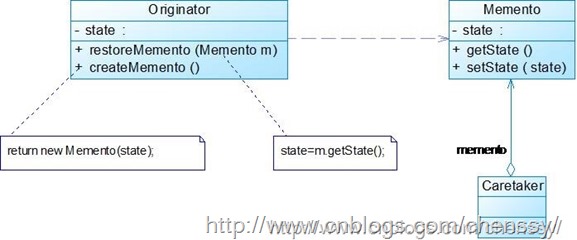
         Colleague: 抽象同事类。

         ConcreteColleague: 具体同事类。每个具体同事类都只需要知道自己的行为即可，但是他们都需要认识中介者。

**6、备忘录模式(Memento)**

     后悔药人人都想要，但是事实却是残酷的，根本就没有后悔药可买，但是也不仅如此，在软件的世界里就有后悔药！备忘录模式就是一种后悔药，它给我们的软件提供后悔药的机制，通过它可以使系统恢复到某一特定的历史状态。

      所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。它实现了对信息的封装，使得客户不需要关心状态保存的细节。保存就要消耗资源，所以备忘录模式的缺点就在于消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

[](http://images.cnitblog.com/blog/381060/201310/08191412-d02656c7a7024d9abbd98c0ac9dd31a5.jpg)

      参与者：

         Originator: 原发器。负责创建一个备忘录，用以记录当前对象的内部状态，通过也可以使用它来利用备忘录恢复内部状态。同时原发器还可以根据需要决定Memento存储Originator的那些内部状态。

         Memento: 备忘录。用于存储Originator的内部状态，并且可以防止Originator以外的对象访问Memento。在备忘录Memento中有两个接口，其中Caretaker只能看到备忘录中的窄接口，它只能将备忘录传递给其他对象。Originator可以看到宽接口，允许它访问返回到先前状态的所有数据。

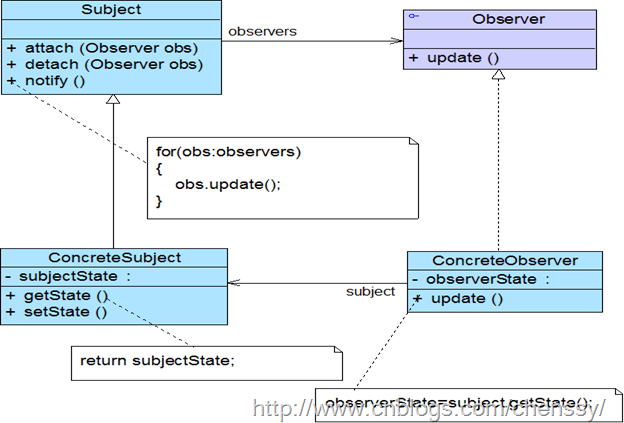
         Caretaker: 负责人。负责保存好备忘录，不能对备忘录的内容进行操作和访问，只能够将备忘录传递给其他对象。

**7、观察者模式(Observer)**

      何谓观察者模式？观察者模式定义了对象之间的一对多依赖关系，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。

      在这里，发生改变的对象称之为观察目标，而被通知的对象称之为观察者。一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，所以么可以根据需要增加和删除观察者，使得系统更易于扩展。

      所以观察者提供了一种对象设计，让主题和观察者之间以松耦合的方式结合。

[](http://images.cnitblog.com/blog/381060/201310/08191413-6c388d6717894b7cb6acad0c28030c2f.png)

      参与者：

          Subject：目标。他把所有对观察者对戏的引用保存在一个聚集里，每一个主题都可以有多个观察者。

          Observer：观察者。为所有的具体观察者定义一个接口，在得到主题的通知时能够及时的更新自己。

          ConcreteSubject：具体主题。将有关状态存入具体观察者对象。在具体主题发生改变时，给所有的观察者发出通知。

          ConcreteObserver：具体观察者。实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题状态相协调。

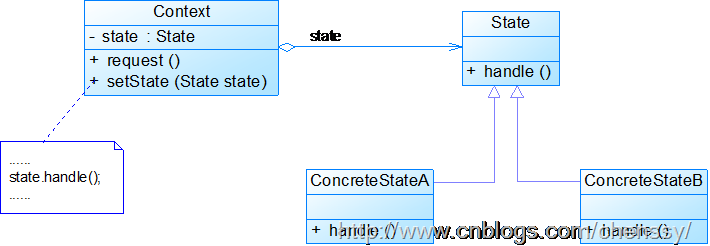
**8、状态模式(State)**

      在很多情况下我们对象的行为依赖于它的一个或者多个变化的属性，这些可变的属性我们称之为状态，也就是说行为依赖状态，即当该对象因为在外部的互动而导致他的状态发生变化，从而它的行为也会做出相应的变化。对于这种情况，我们是不能用行为来控制状态的变化，而应该站在状态的角度来思考行为，即是什么状态就要做出什么样的行为。这个就是状态模式。

      所以状态模式就是允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

      在状态模式中我们可以减少大块的if…else语句，它是允许态转换逻辑与状态对象合成一体，但是减少if…else语句的代价就是会换来大量的类，所以状态模式势必会增加系统中类或者对象的个数。

      同时状态模式是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。但是这样就会导致系统的结构和实现都会比较复杂，如果使用不当就会导致程序的结构和代码混乱，不利于维护。

[](http://images.cnitblog.com/blog/381060/201310/08191414-88eaa61f773147f4836b7a67b46bdfef.png)

      参与者：

         Context：环境类。可以包括一些内部状态。   
         State： 抽象状态类。State定义了一个所有具体状态的共同接口，任何状态都实现这个相同的接口，这样一来，状态之间就可以互相转换了。   
         ConcreteState：具体状态类。具体状态类，用于处理来自Context的请求，每一个ConcreteState都提供了它对自己请求的实现，所以，当Context改变状态时行为也会跟着改变。

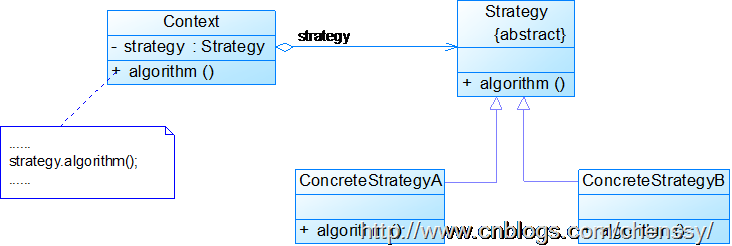
**9、策略模式(Strategy)**

      我们知道一件事可能会有很多种方式来实现它，但是其中总有一种最高效的方式，在软件开发的世界里面同样如此，我们也有很多中方法来实现一个功能，但是我们需要一种简单、高效的方式来实现它，使得系统能够非常灵活，这就是策略模式。

      所以策略模式就是定义了算法族，分别封装起来，让他们之前可以互相转换，此模式然该算法的变化独立于使用算法的客户。

      在策略模式中它将这些解决问题的方法定义成一个算法群，每一个方法都对应着一个具体的算法，这里的一个算法我就称之为一个策略。虽然策略模式定义了算法，但是它并不提供算法的选择，即什么算法对于什么问题最合适这是策略模式所不关心的，所以对于策略的选择还是要客户端来做。客户必须要清楚的知道每个算法之间的区别和在什么时候什么地方使用什么策略是最合适的，这样就增加客户端的负担。

      同时策略模式也非常完美的符合了“开闭原则”，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。但是一个策略对应一个类将会是系统产生很多的策略类。

[](http://images.cnitblog.com/blog/381060/201310/08191415-fc7cc4e987bd4f0a817a59a07a5dd0ad.png)

      参与者：

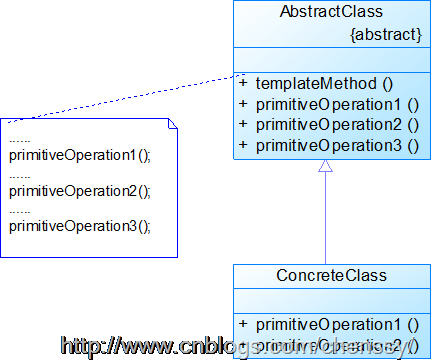
         Context: 环境类。维护一个Strategy对象的引用，用一个ConcreteStrategy来配置，可定义一个接口来让Strategy访问它的数据。   
         Strategy: 抽象策略类。定义所有支持算法的公共接口。Context使用这个接口来调用某个Concretestrategy定义的算法。   
         ConcreteStrategy: 具体策略类。封装了具体的算法实现。

**10、模板方法模式(Template Method)**

      有些时候我们做某几件事情的步骤都差不多，仅有那么一小点的不同，在软件开发的世界里同样如此，如果我们都将这些步骤都一一做的话，费时费力不讨好。所以我们可以将这些步骤分解、封装起来，然后利用继承的方式来继承即可，当然不同的可以自己重写实现嘛！这就是模板方法模式提供的解决方案。

      所谓模板方法模式就是在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

      模板方法模式就是基于继承的代码复用技术的。在模板方法模式中，我们可以将相同部分的代码放在父类中，而将不同的代码放入不同的子类中。也就是说我们需要声明一个抽象的父类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法让子类来实现剩余的逻辑，不同的子类可以以不同的方式来实现这些逻辑。所以模板方法的模板其实就是一个普通的方法，只不过这个方法是将算法实现的步骤封装起来的。

[](http://images.cnitblog.com/blog/381060/201310/08191416-4f16a50a631d4b9e95de6440b6cd7da9.png)

      参与者：

         AbstractClass: 抽象类。实现了一个模板，实现算法的基本骨架，具体子类将重定义primitiveOperation()方法以实现一个算法步骤。

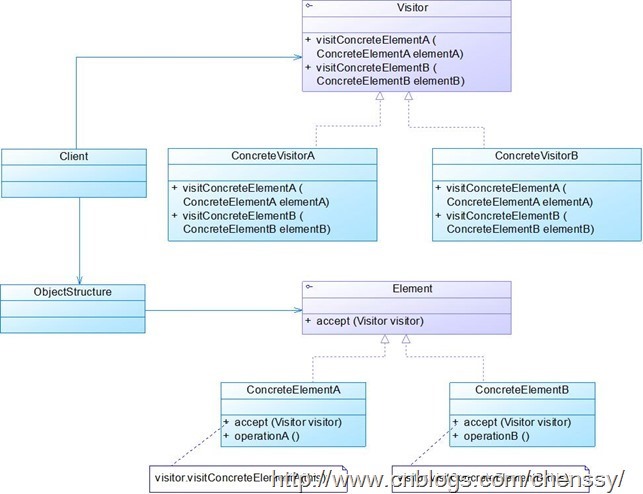
         ConcreteClass:  具体子类。实现primitiveOperation()方法以完成算法中与特定子类相关的步骤。

**11、访问者模式(Visitor)**

      访问者模式俗称23大设计模式中最难的一个。除了结构复杂外，理解也比较难。在我们软件开发中我们可能会对同一个对象有不同的处理，如果我们都做分别的处理，将会产生灾难性的错误。对于这种问题，访问者模式提供了比较好的解决方案。

     访问者模式即表示一个作用于某对象结构中的各元素的操作，它使我们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操作。

      访问者模式的目的是封装一些施加于某种数据结构元素之上的操作，一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。为不同类型的元素提供多种访问操作方式，且可以在不修改原有系统的情况下增加新的操作方式。同时我们还需要明确一点那就是访问者模式是适用于那些数据结构比较稳定的，因为他是将数据的操作与数据结构进行分离了，如果某个系统的数据结构相对稳定，但是操作算法易于变化的话，就比较适用适用访问者模式，因为访问者模式使得算法操作的增加变得比较简单了。

[](http://images.cnitblog.com/blog/381060/201310/08191417-ce61254e401b42548afb5a37e298ac44.jpg)

      参与者：

         Vistor: 抽象访问者。为该对象结构中的ConcreteElement的每一个类声明的一个操作。   
         ConcreteVisitor: 具体访问者。实现Visitor申明的每一个操作，每一个操作实现算法的一部分。   
         Element: 抽象元素。定义一个Accept操作，它以一个访问者为参数。   
         ConcreteElement: 具体元素 。实现Accept操作。   
         ObjectStructure: 对象结构。能够枚举它的元素，可以提供一个高层的接口来允许访问者访问它的元素。