1.创建模式： 工厂方法、简单工厂、抽象工厂、单例、建造、模型；

1.1.FACTORY METHOD—请 MM 去麦当劳吃汉堡，不同的 MM 有不同的口味，要每个都记住是一件烦人的事情，我一般采用 Factory Method 模式，带着 MM 到服务员那儿，说"要一个汉堡"，具体要什么样的汉堡呢，让 MM 直接跟服务员说就行了。

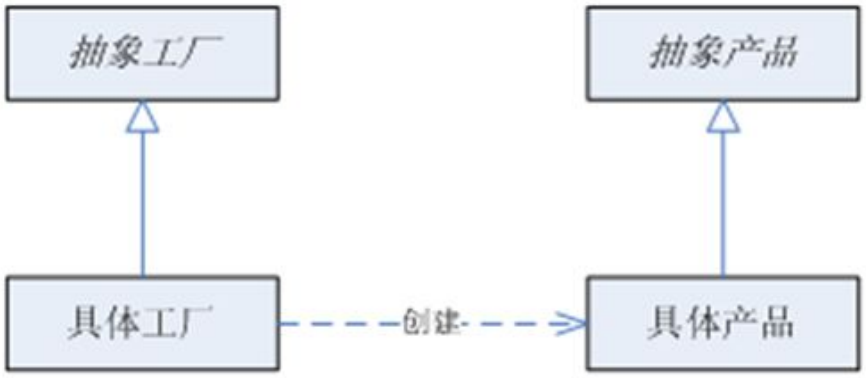
工厂方法模式是类的创建模式，又叫虚拟构造子（Virtual Constructor）模式或者多态性工厂（Polymorphic Factory）模式。 工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际工作推迟到子类中。

工厂方法解决问题：

工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

工厂方法模式：核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

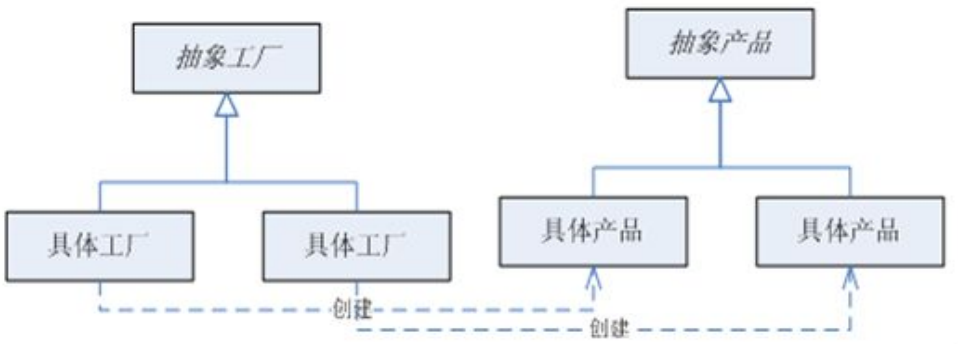
工厂方法缩略图



该模式的优点：

这种抽象的结果，使这种工厂方法模式可以用来允许系统不修改具体工厂角色的情况下引进新产品，这一特点无疑使得工厂模式具有超过简单工厂模式的优越性。

在工厂方法模式中，一般都有一个平行的等级结构，也就是说工厂和产品是对应的的。抽象工厂对应抽象产品，具体工厂对应具体产品。简单的示意图如下：



1.2 简单工厂模式是创建型模式，用于对象的创建。简单工厂模式（Simple Factory Pattern）属于类的创新型模式，又叫静态工厂方法模式（Static FactoryMethod Pattern）,是通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。它是工厂模式家族中最简单实用的模式，可以理解为是不同工厂模式的一个特殊实现。

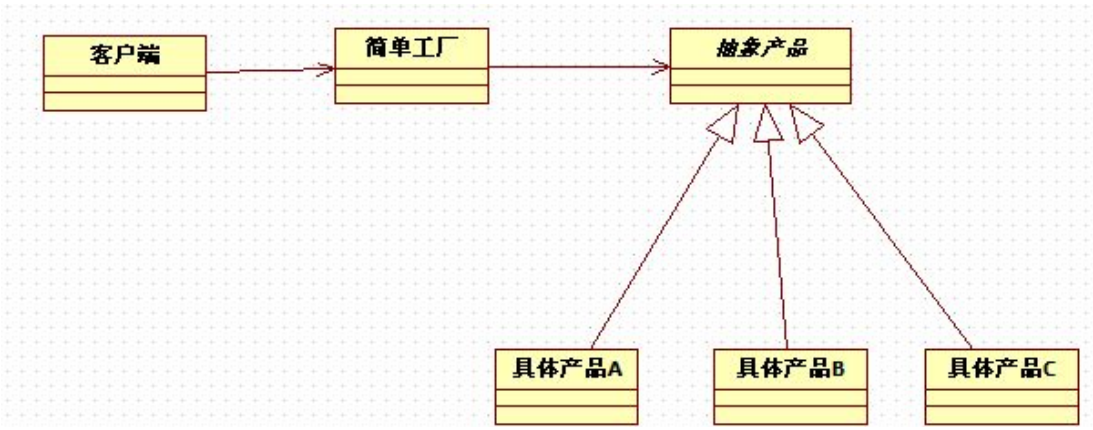
简单工厂模式的 UML 图：

简单工厂模式中包含的角色及其相应的职责如下：

工厂角色（Creator）：这是简单工厂模式的核心，由它负责创建所有的类的内部逻辑。当然工厂类必须能够被外界调用，创建所需要的产品对象。

抽象（Product）产品角色：简单工厂模式所创建的所有对象的父类，注意，这里的父类可以是接口也可以是抽象类，它负责描述所有实例所共有的公共接口。

具体产品（Concrete Product）角色：简单工厂所创建的具体实例对象，这些具体的产品往往都拥有共同的父类。



简单工厂模式的优缺点分析：

优点：工厂类是整个模式的关键所在。它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，而无需了解这些对象是如何创建以及如何组织的。有利于整个软件体系结构的优化。

缺点：由于工厂类集中了所有实例的创建逻辑，这就直接导致一旦这个工厂出了问题，所有的客户端都会受到牵连；而且由于简单工厂模式的产品室基于一个共同的抽象类或者接口，这样一来，但产品的种类增加的时候，即有不同的产品接口或者抽象类的时候，工厂类就需要判断何时创建何种种类的产品，这就和创建何种种类产品的产品相互混淆在了一起，违背了单一职责，导致系统丧失灵活性和可维护性。而且更重要的是，简单工厂模式违背了“开放封闭原则”，就是违背了“系统对扩展开放，对修改关闭”的原则，因为当我新增加一个产品的时候必须修改工厂类，相应的工厂类就需要重新编译一遍。

总结一下：简单工厂模式分离产品的创建者和消费者，有利于软件系统结构的优化；但是由于一切逻辑都集中在一个工厂类中，导致了没有很高的内聚性，同时也违背了“开放封闭原则”。另外，简单工厂模式的方法一般都是静态的，而静态工厂方法是无法让子类继承的，因此，简单工厂模式无法形成基于基类的继承树结构。

2.*结构模式：* 适配器、缺省适配、合成、装饰、代理、享元、门面、桥梁；

2.1ADAPTER pattern 在朋友聚会上碰到了一个美女 Sarah，从香港来的，可我不会说粤语，她不会说普通话，只好求助于我的朋友 kent 了，他作为我和 Sarah 之间的 Adapter，让我和 Sarah 可以相互交谈了(也不知道他会不会耍我)

适配器（变压器）模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作。适配类可以根据参数返还一个合适的实例给客户端。

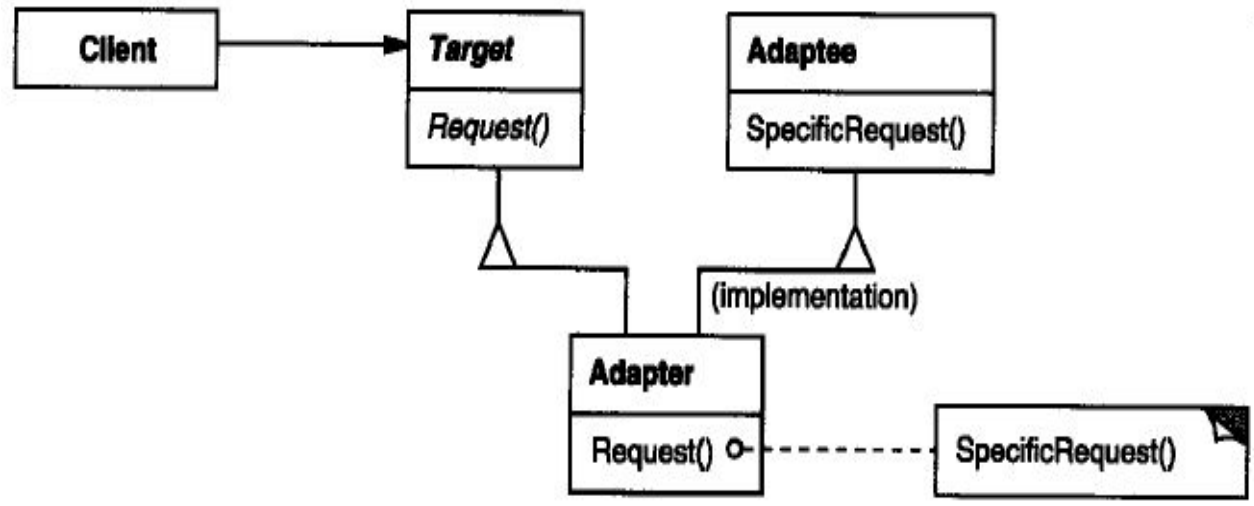
适用性

以下情况使用 A d a p t e r 模式

你想使用一个已经存在的类，而它的接口不符合你的需求。

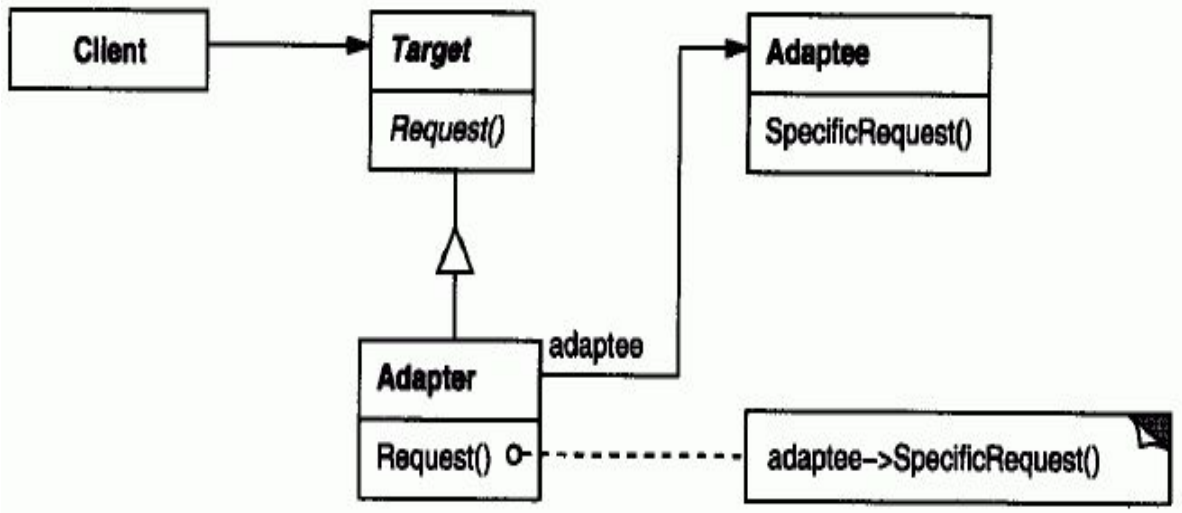
你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。

结构（类版本）



基于类的 Adapter 模式的一般结构如下：Adaptee 类为 Adapter 的父类，Adaptee 类为适配源，适配目标（接口）也是 Adapter 的父类；基于类的 Adapter 模式比较适合应用于 Adapter 想修改 Adaptee 的部分方法的情况

结构（对象版本）



Adaptee 类对象为 Adapter 所依赖，适配目标（接口）是 Adapter 的父类；基于对象的 Adapter 模式比较适合应用于 Adapter 想为 Adaptee 添加新的方法的情况。

Adapter 模式

- 参与者
  - Target
    - Client使用的与特定领域相关的“接口”。
  - Client
    - 与符合Target接口的对象协同的专业系统。
  - Adaptee
    - 一个已经存在的“接口”，它具有Client要求的功能但不符合Client的接口要求。这个接口需要适配。
  - Adapter
    - 对Adaptee的接口与Target接口进行适配

- 协作
  - Client在Adapter实例上调用一些操作（请求）。接着适配器调用Adaptee的操作实现这个请求。

- 效果（类适配器和对象适配器有不同的权衡）
  - 类适配器
    - 用一个具体的Adapter类对Adaptee和Target进行匹配。结果是当我们想要匹配一个类以及所有它的子类时，类Adapter将不能胜任工作。
    - 使得Adapter可以重定义Adaptee的部分行为，因为Adapter是Adaptee的一个子类。
    - 仅仅引入了一个对象，并不需要额外的指针以间接得到adaptee。

- 对象适配器则
  - 允许一个Adapter与多个Adaptee—即Adaptee本身以及它的所有子类（如果有子类的话）同时工作。Adapter也可以一次给所有的Adaptee添加功能。
  - 使得重定义Adaptee的行为比较困难。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。

2.2 合成模式：COMPOSITE—Mary 今天过生日。"我过生日，你要送我一件礼物。""嗯，好吧，去商店，你自己挑。""这件 T 恤挺漂亮，买，这条裙子好看，买，这个包也不错，买。""喂，买了三件了呀，我只答应送一件礼物的哦。""什么呀，T 恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。".....", MM 都会用 Composite 模式了，你会了没有？

合成模式：合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式就是一个处理对象的树结构的模式。合成模式把部分与整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由他们复合而成的合成对象同等看待。

要点：如何设计单个对象和组合对象，使他们具有一致的操作接口。

Composite 模式把部分和整体关系用树结构表示，是属于对象的结构模式。 Composite 模式要对 Composite 的对象进行管理，所以在一定位置给予对象的相关管理方法，如：add(),remove()等.

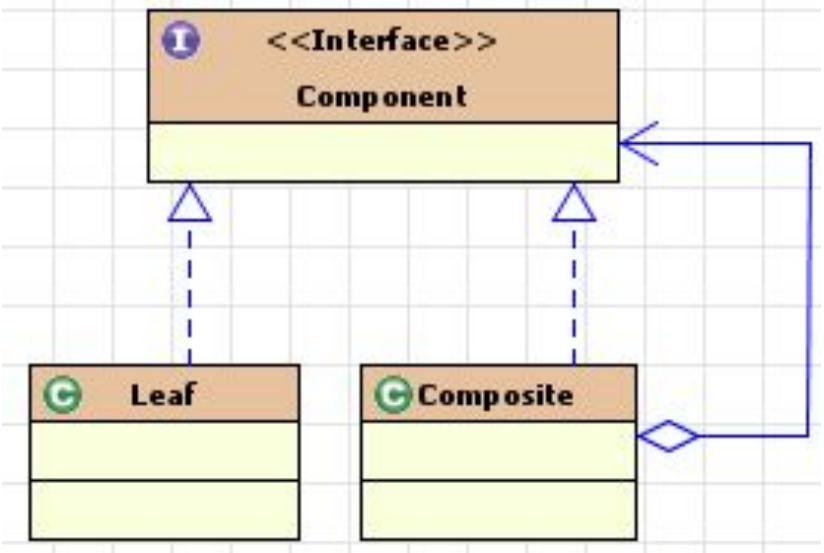
Composite 模式中对象的管理有两种方案。

安全方式：此方式只允许树枝构件有对象的管理方法。
透明方式：此方式只允许树枝和树叶都有对象的管理方法，但树叶对象中的管理方法无实际意义。安全式特点：错误地调用会导致编译时出错

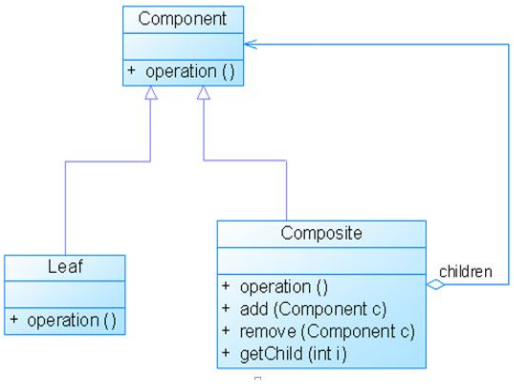
透明式特点：编译时不会出错，错误地调用会导致抛出异常

组成部分：

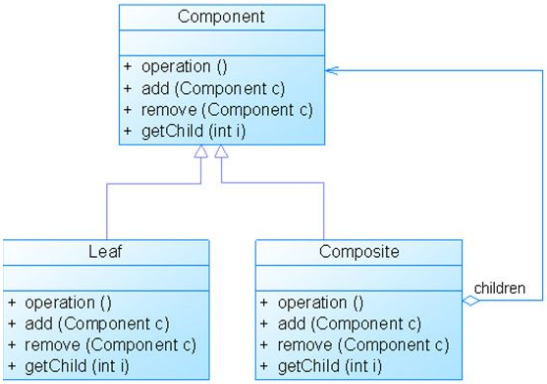
- component构件：抽象组合对象的公共行为接口
- leaf构件：树叶对象，没有下级子对象
- composite构件：树枝对象，树枝对象可以包含一个或多个其他树枝或树叶对象



安全方式：



透明方式：



适用环境:

需要表示一个对象整体或部分层次，在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，可以一致地对待它们。

让客户能够忽略不同对象层次的变化，客户端可以针对抽象构件编程，无需关心对象层次结构的细节。

对象的结构是动态的并且复杂程度不一样，但客户需要一致地处理它们。

- 优点:
  - 可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，使得增加新构件也更容易。
  - 客户端调用简单，客户端可以一致的使用组合结构或其中单个对象。
  - 定义了包含叶子对象和容器对象的类层次结构，叶子对象可以被组合成更复杂的容器对象，而这个容器对象又可以被组合，这样不断递归下去，可以形成复杂的树形结构。
  - 很容易在组合体内加入对象构件，客户端不必因为加入了新的对象构件而更改原有代码。
- 缺点：
  - 设计变得更加抽象，对象的业务规则如果很复杂，则实现组合模式具有很大挑战性，而且不是所有的方法都与叶子对象子类都有关联。
  - 增加新构件时可能会产生一些问题，很难对容器中的构件类型进行限制。

2.3装饰模式：

动态地给一个对象添加一些额外的职责。

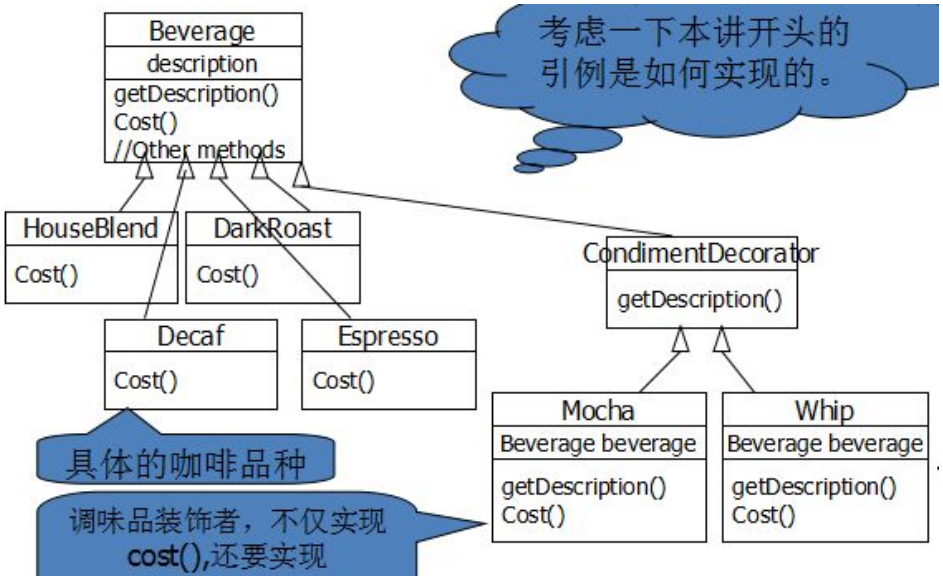
优点：把类中的装饰功能从类中搬移出去，这样可以简化原有的类。有效地把类的核心功能和装饰功能区分开了。

❖ 何谓装饰？如何装饰？

- 所谓装饰就是将一个对象包装起来。
- 在程序上也就是让一个对象a包含另一个对象b。
- a对应的类A是"装饰类"，b对应的类B是被装饰类。
- 如果要让装饰可以重复，且不必考虑次序，那么装饰的要点是装饰者和被装饰者具有相同的类型（有共同的父类）。

已经开发完毕的对象，后期由于业务需要，对旧的对象需要扩展特别多的功能，这时候使用给对象动态地添加新的状态或者行为（即装饰模式）方法，而不是使用子类静态继承。

例子：



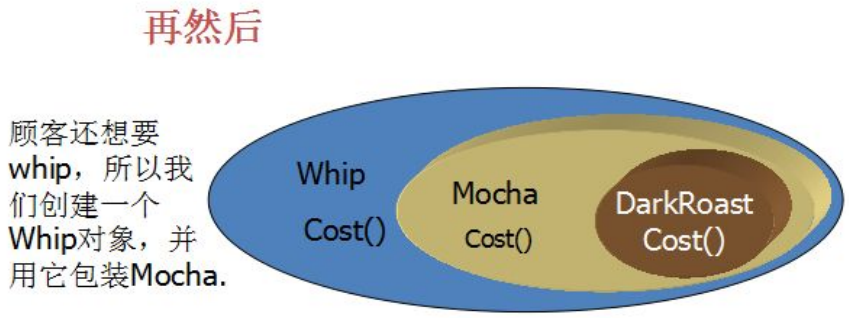
首先生成一个DarkRoast对象

然后

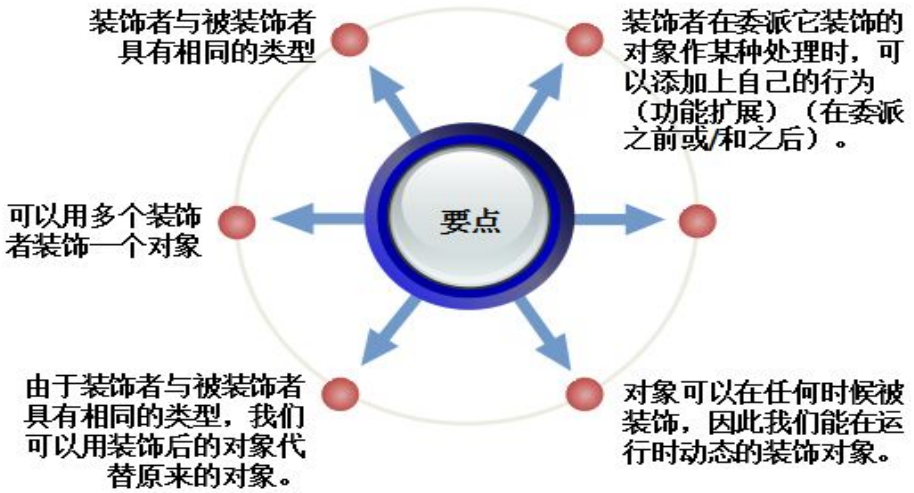
顾客想要mocha，所以我们创建一个Mocha对象，并用它包装DarkRoast.

DarkRoast继承了Beverage，拥有一个计算饮料价格的方法cost()。

Mocha对象是装饰者，他与被它装饰的对象DarkRoast具有相同的类型（是Beverage的子类），也有一个cost()方法。



所谓委派就是一个对象将工作（或工作的一步分）交给另一个对象来完成。在装饰模式中，委派是指装饰对象将任务交给被装饰对象来完成。委派可以传递，最终必须要有一个干实事的对象。

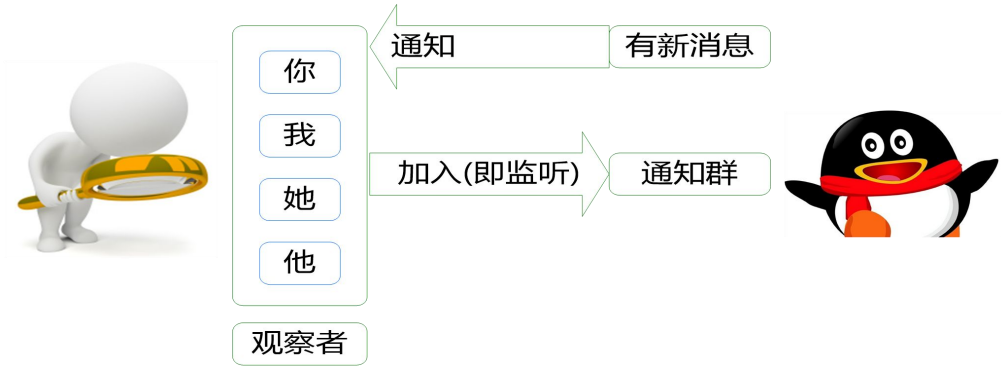


3.行为模式： 不变、策略、模板方法、观察者、迭代子、责任链、命令

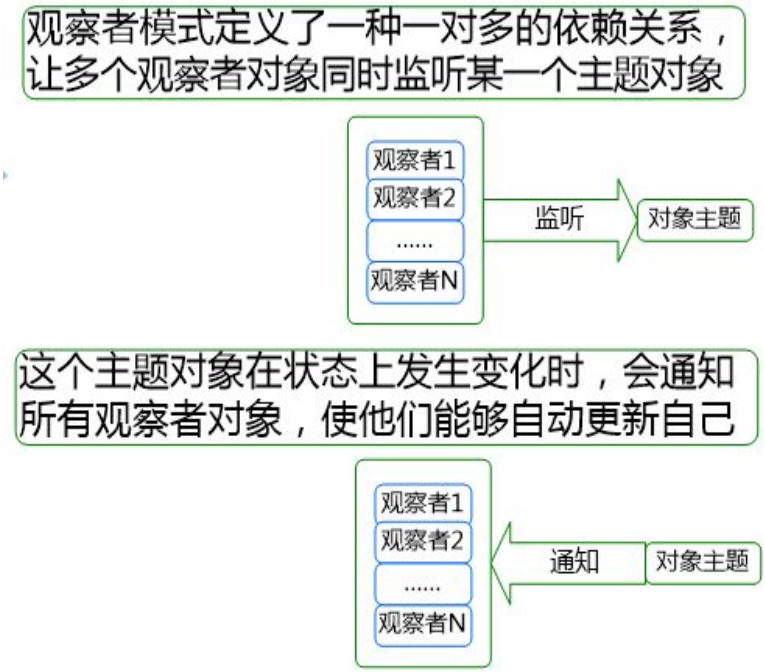
备忘录、状态、访问者、解释器、调停者

3.1 观察者模式：

例子：



特点：



优点：

1、被观察者和观察者之间建立一个抽象的耦合。

被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。
被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

2、观察者模式支持广播通讯。

被观察者会向所有的登记过的观察者发出通知

缺点：

1 、如果一个被观察者对象有很多的直接和间接的观察者，将所有的观察者都通知到会花费很多时间

2 、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。

3 、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的

4 、可以随时使观察者知道观察的对象发生了变化，但是没有相应的机制使观察者知道观察的对象是怎么发生变化的

2.2 责任链模式：

CHAIN OF RESPONSIBLEITY—晚上去上英语课，为了好开溜坐到了最后一排，哇，前面坐了好几个漂亮的 MM 哎，找张纸条，写上"Hi,可以做我的女朋友吗？如果不愿意请向前传"，纸条就一个接一个的传上去了，糟糕，传到第一排的 MM 把纸条传给老师了，听说是个老处女呀，快跑!

责任链模式：在责任链模式中，很多对象由每一个对象对其下家的引用而接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。客户并不知道链上的哪一个对象最终处理这个请求，系统可以在不影响客户端的情况下动态的重新组织链和分配责任。处理者有两个选择：承担责任或者把责任推给下家。一个请求可以最终不被任何接收端对象所接受。

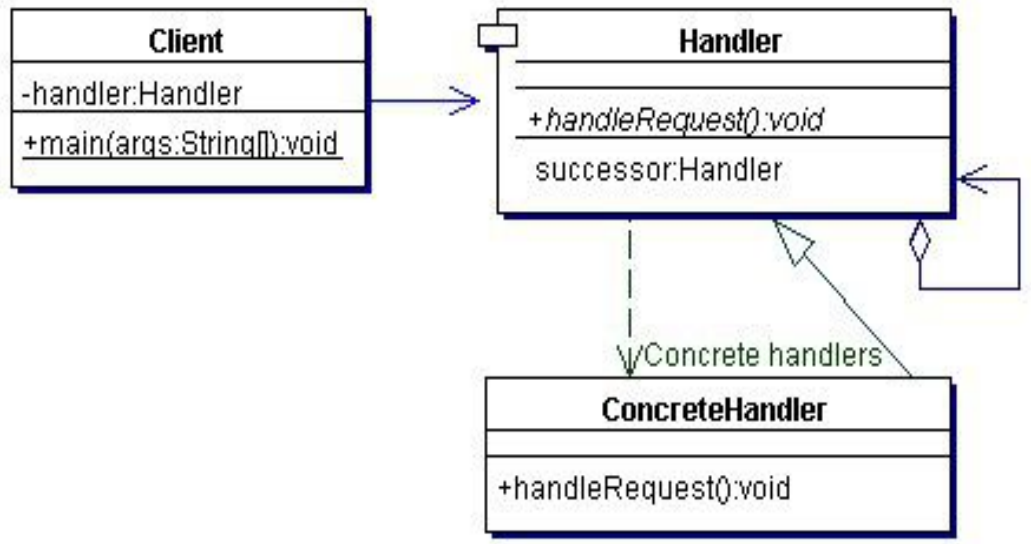
责任链模式是使用多个对象处理用户请求的成熟模式，责任链模式的关键是将用户的请求分派给许多对象，这些对象被组织成一个责任链，即每个对象含有后继对象的引用，并要求责任链上的每个对象，如果能处理用户的请求，就做出处理，不能处理就必须将用户的请求传递给责任链上的下一个对象。

责任链模式的结构

第一、抽象处理者（Handler）角色

定义出一个处理请求的接口；如果需要，接口可以定义出一个方法，以返回对下家的引用。下图给出了一个示意性的类图：

第二、具体处理者（ConcreteHandler）角色、处理接到请求后，可以选择将请求处理掉，或者将请 求传给下家。下图给出了一个示意性的类图。



责任链模式优点：责任链模式减低了发出命令的对象和处理命令的对象之间的耦合，它允许多与一个的处理者对象根据自己的逻辑来决定哪一个处理者最终处理这个命令。换言之，发出命令的对象只是把命令传给链结构的起始者，而不需要知道到底是链上的哪一个节点处理了这个命令。这意味着在处理命令上，允许系统有更多的灵活性。哪一个对象最终处理一个命令可以因为由那些对象参加责任链、以及这些对象在责任链上的位置不同而有所不同。

缺点：责任链模式可能会带来一些额外的性能损耗，因为它要从链子开头开始遍历。

适用情况：

第一、系统已经有一个由处理者对象组成的链。当有多于一个的处理者对象会处理一个请求，而且在事先并不知道到底由哪一个处理者对象处理一个请求。这个处理者对象是动态确定的。

第二、当系统想发出一个请求给多个处理者对象中的某一个，但是不明显指定是哪一个处理者对象会处理此请求。

第三、当处理一个请求的处理者对象集合需要动态地指定时。