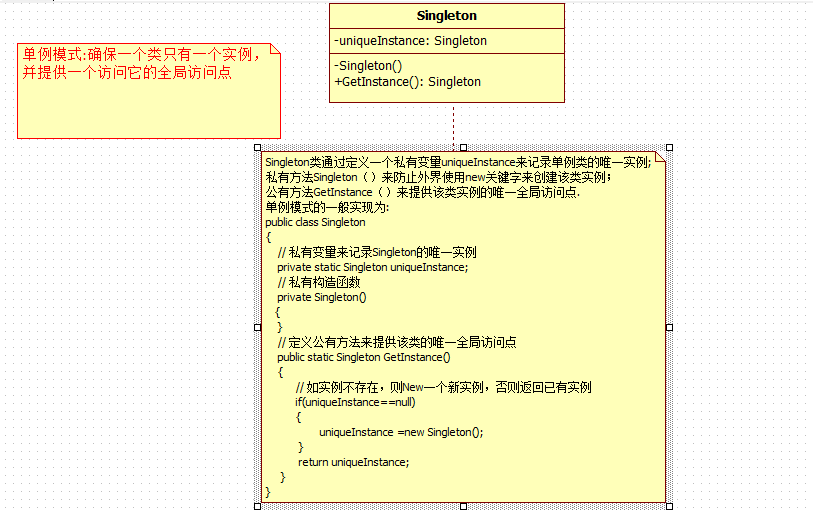


1. 单例模式

保证只有一个实例对象的存在

实现方法(单线程下)：采用私有变量的形式，在类中声明同一个类的实例变量。采用静态

****

原因：例如:操作系统中只能有一个任务管理器,操作文件时,同一时间内只允许一个实例对其操作等,既然现实生活中有这样的应用场景,自然在软件设计领域必须有这样的解决方案了(因为软件设计也是现实生活中的抽象)，

多线程模式下，使用 对象锁进行线程的锁定



对于每一个 线程都会有 lock锁，为了对特定uniqueInstance==null实现再一次的筛选，形成双层的锁机制。在多线程情况下，可能多个线程同时进入lock的判断，假设A,B，当A实例化完成后，B获得对象锁，若是不含有单例判定，那么，将再次进行实例化，浪费资源

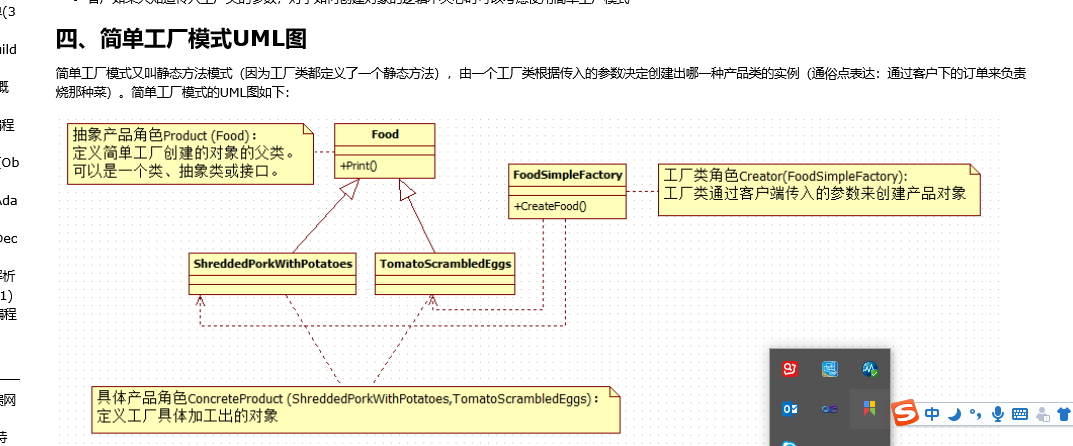


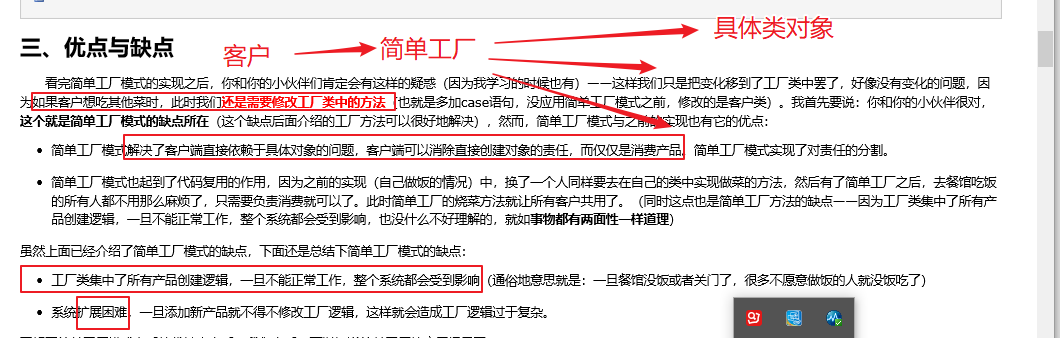
1. 工厂模式

用于生成对象实例

A简单工厂

采用门店的形式，集中处理生成所有类实例，客户只需要提供需要的实例信息既可以获得实例对象，简单工厂类依赖食物的类





B 工厂方法

指定 产品的规格， 给出具体工厂的接口，使用面向对象编程的特点，子类实例可用作父类声明变量的对象。

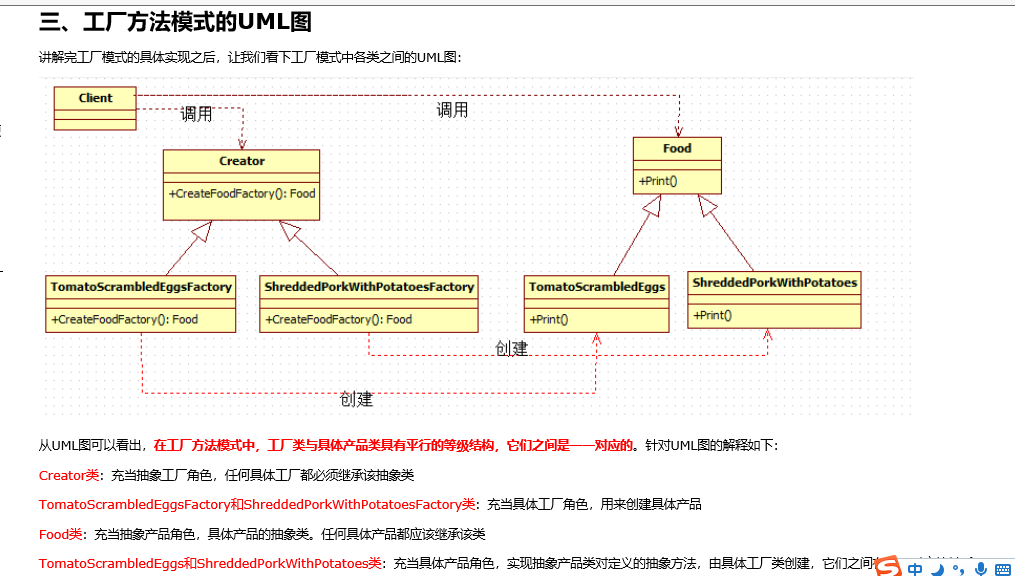
指定的接口， 流动的工厂实例， 因此，可实现面向对象编程的多态性





将简单工厂的实例化类，进行抽象编程，利用父类子类之间的泛化关系，生成实例

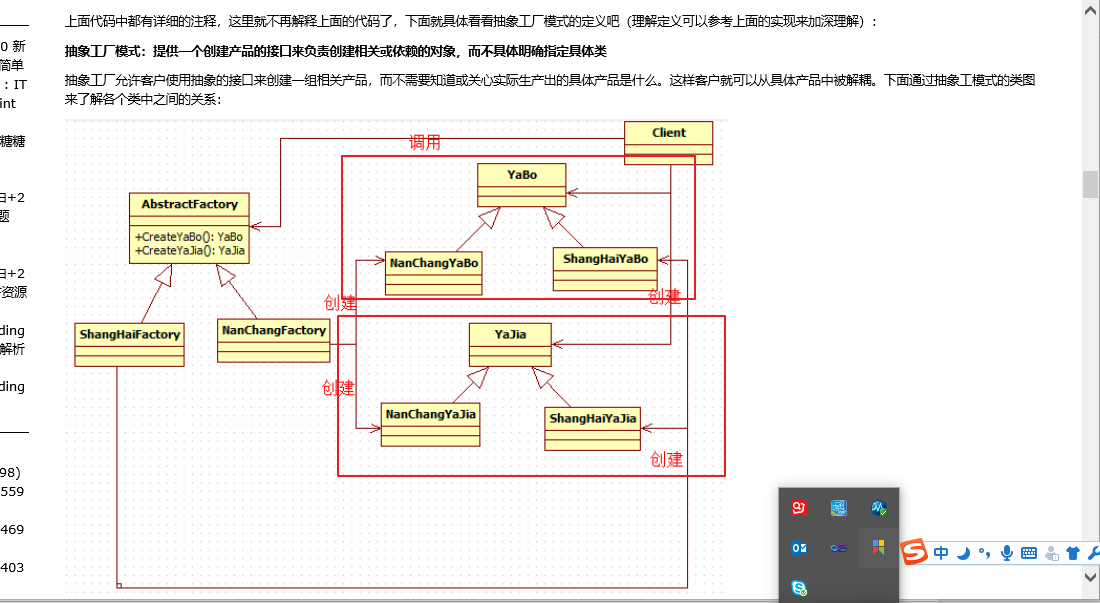
当后期需要添加更多的工厂时，至需要添加继承抽象接口的具体实现工厂类



抽象工厂：

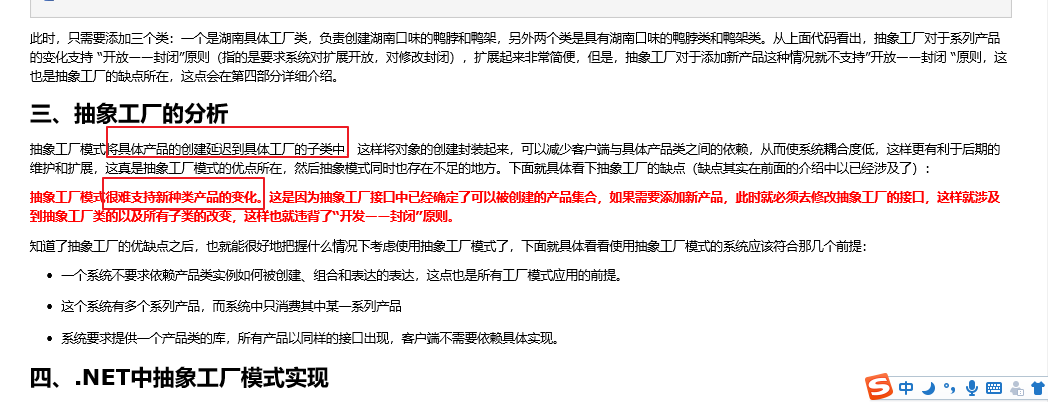
考虑到产品并非是唯一的，需要有多样性

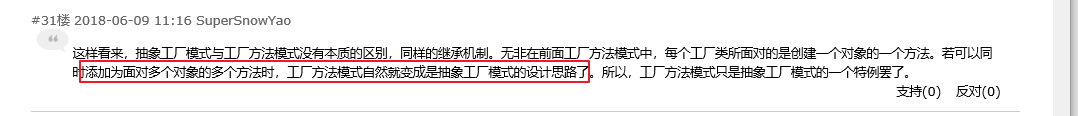
Food可能会有八大菜系的做法



我们需要抽象出不同地域的食物做法，对于不同的工厂，实例化不同的食物具体实例，

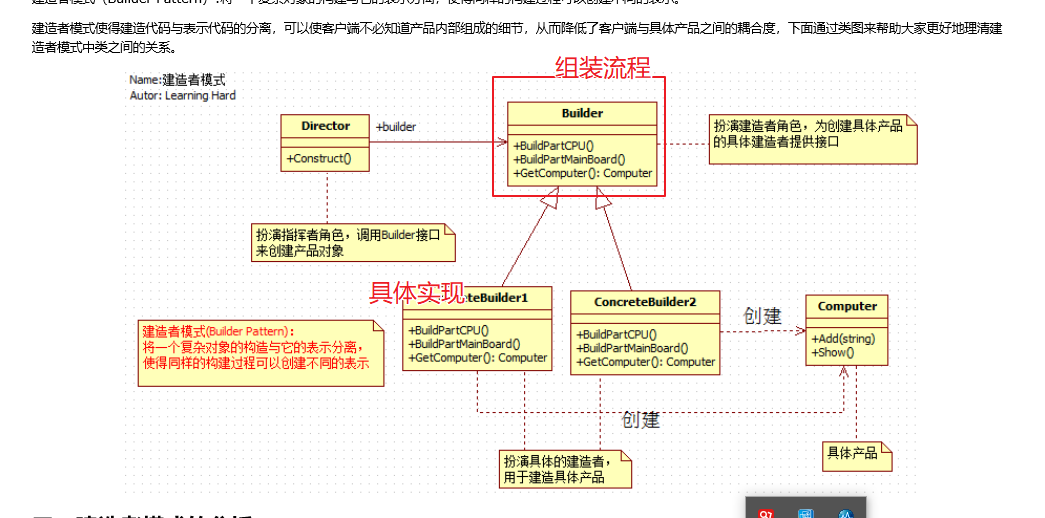
由于工厂的抽象接口，已经制定了产品的规格

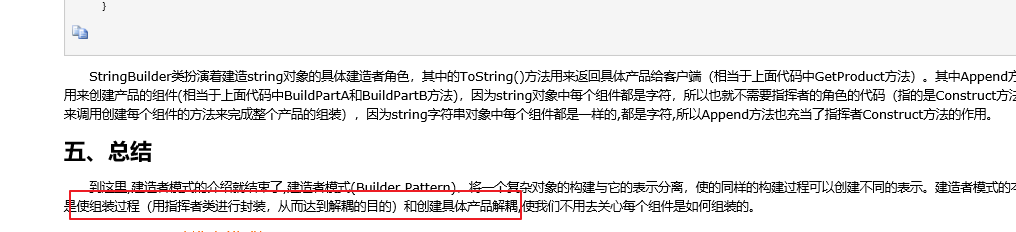




1. 建造者

Direcor进行执行发送 builder执行， builder 发送标准，交由下方具体builder建造



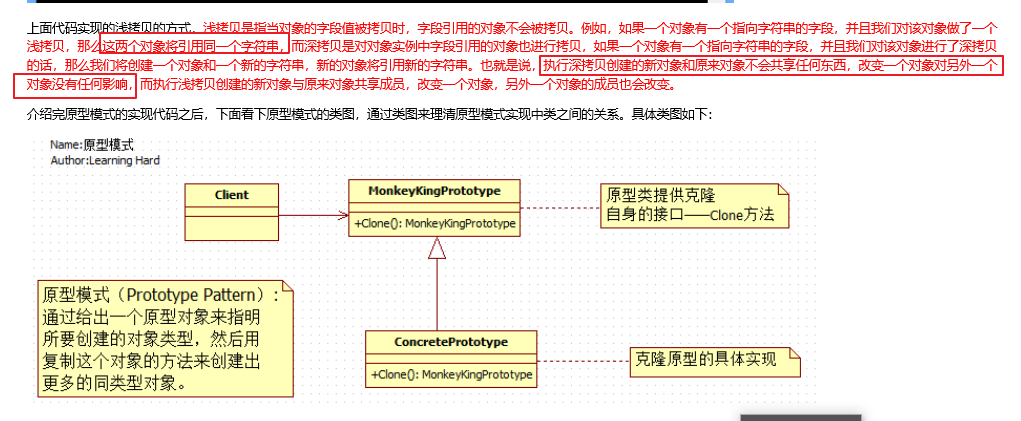


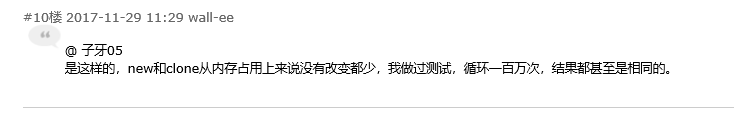
1. 原型

将类实例对象进行拷贝，形成对应的创建， 减少内存消耗及类实例的复用

涉及浅拷贝（地址指向相同），深拷贝（深度复制）

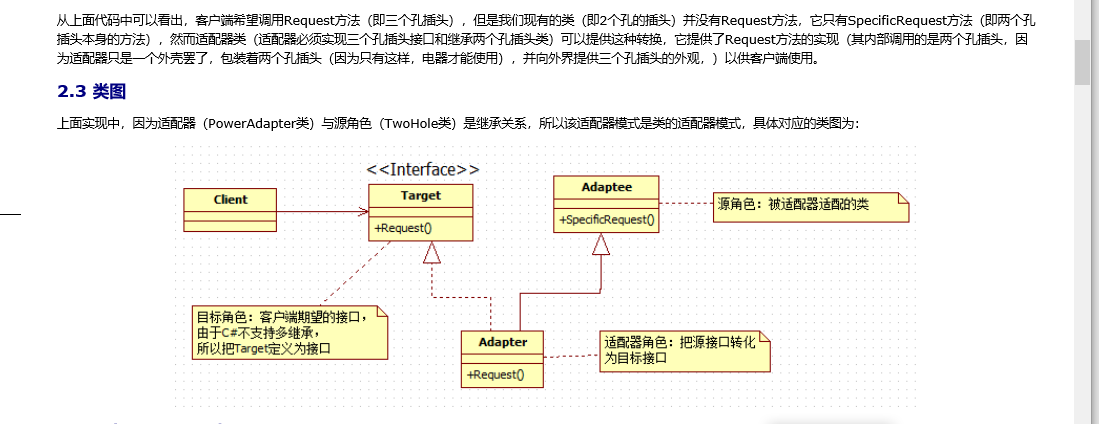






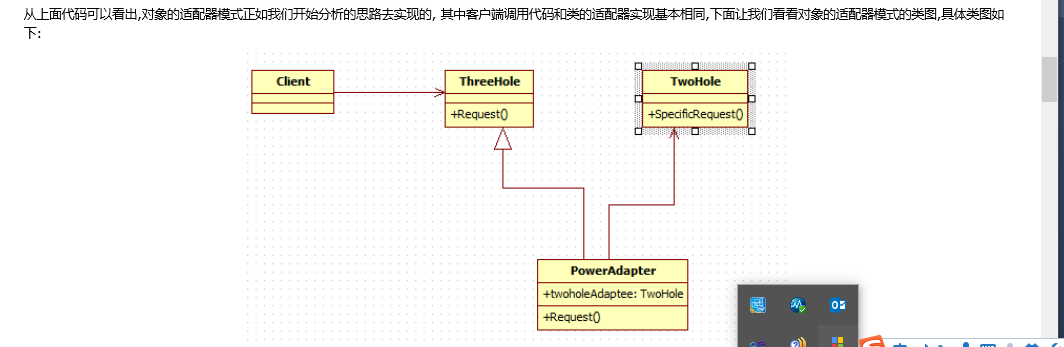
1. 适配器

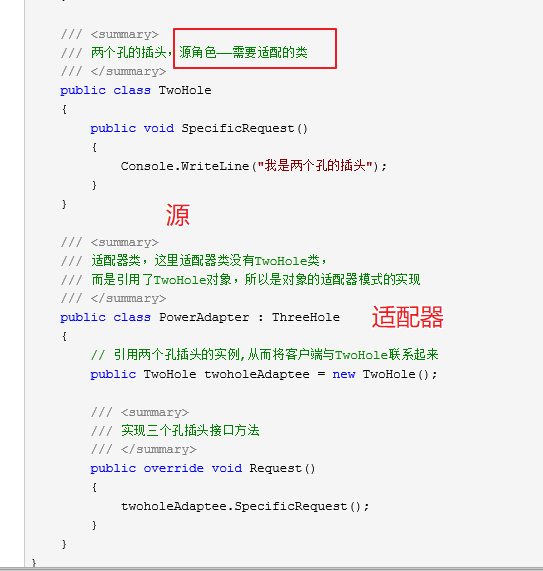
将类对外的接口，封装在适合标准的接口上

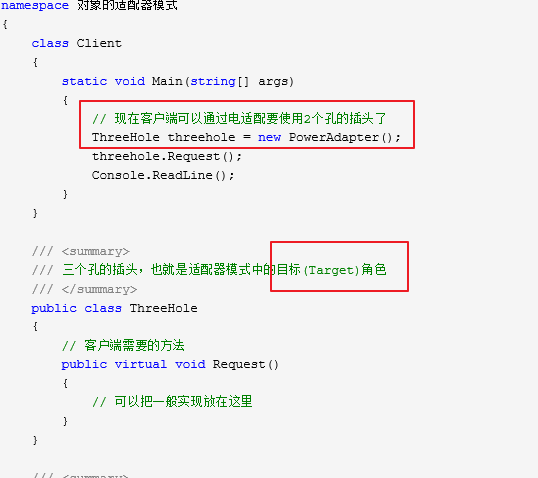




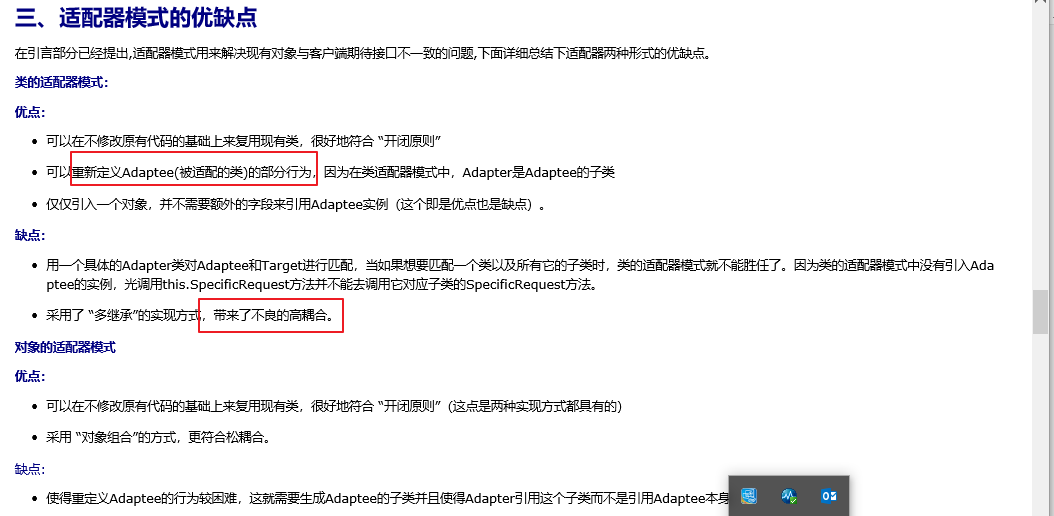
或者说，将对应的想要转换的接口，用对应的第三方类进行转化调用

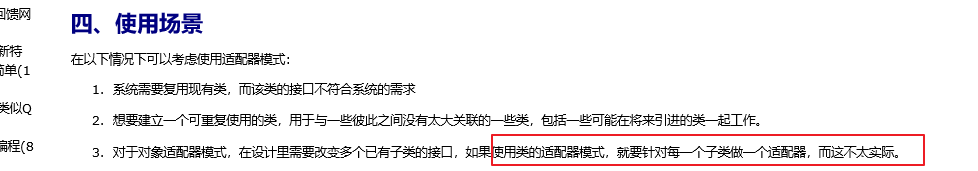








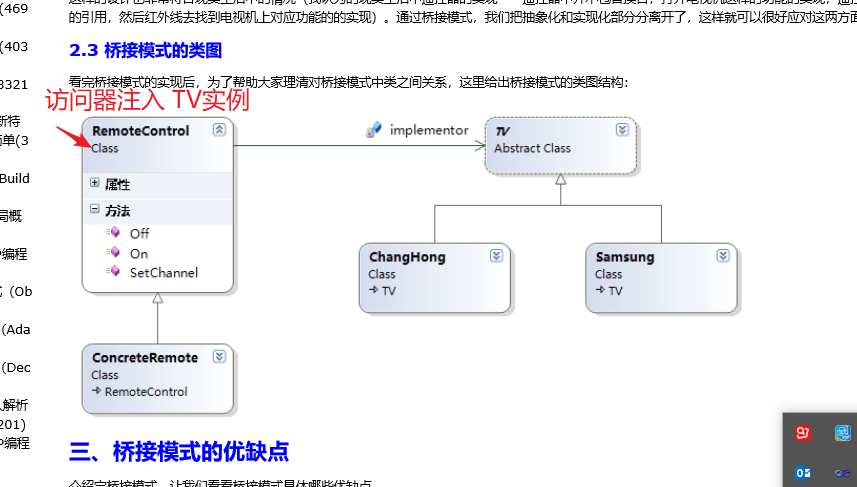


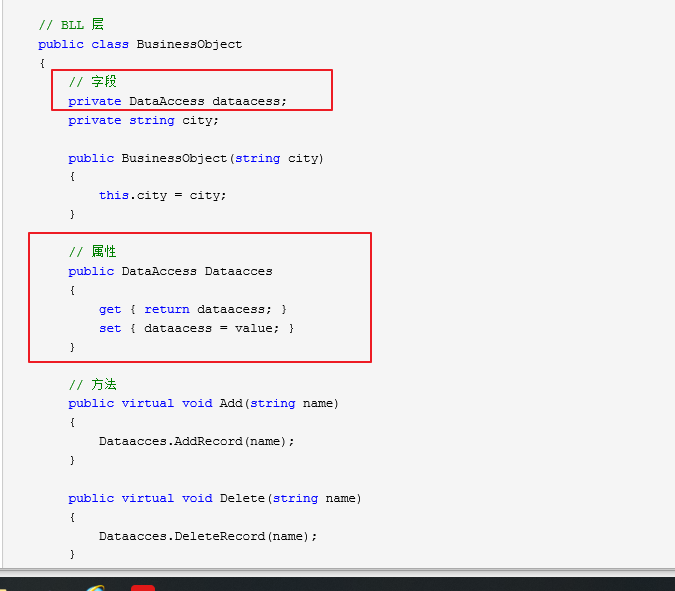


将适配器的适配对象，通过构造函数注入的方式注入，

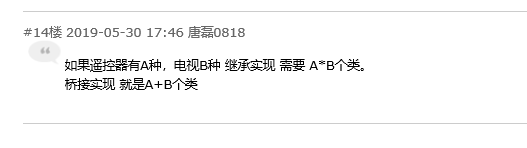
1. 桥接模式

抽象类的使用以及对应的操作注入对象





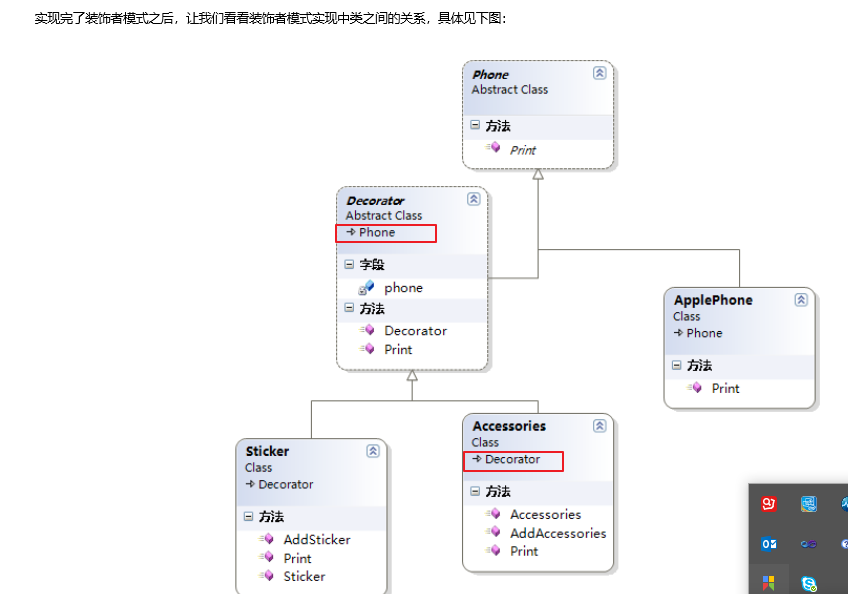


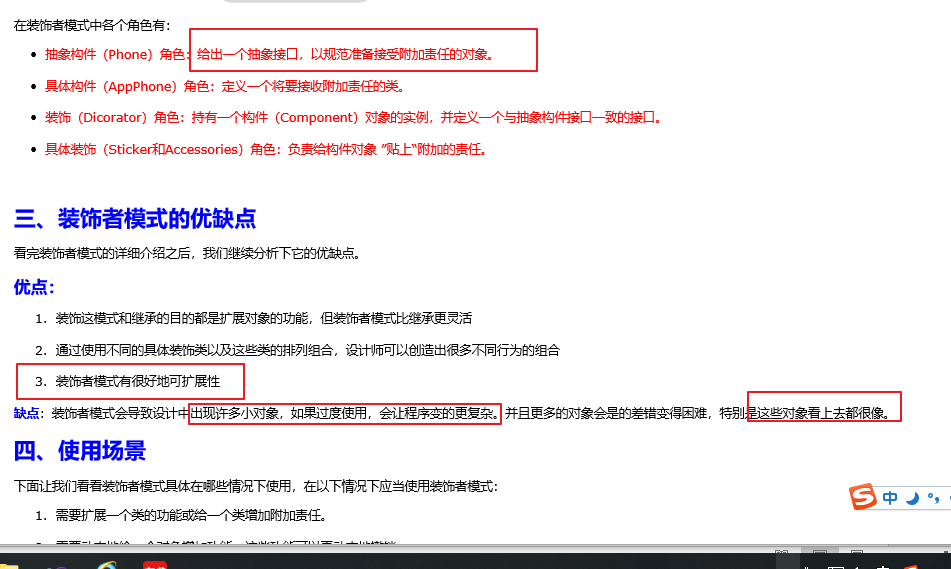


1. 装饰模式

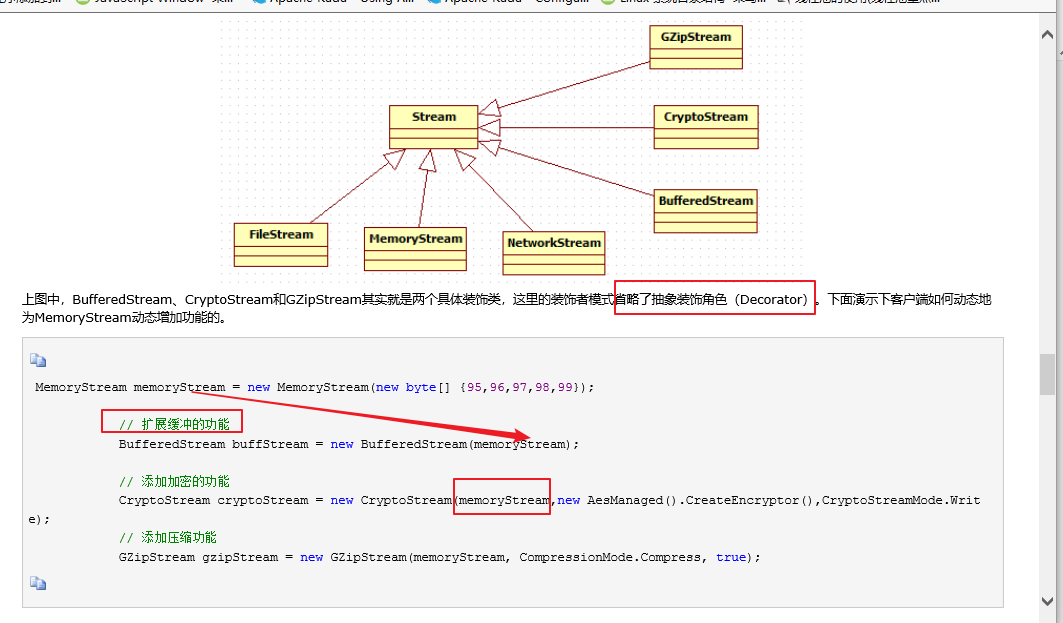
利用构造器的注入形式，对对象实例进行进一步的扩展







特殊的装饰者模式

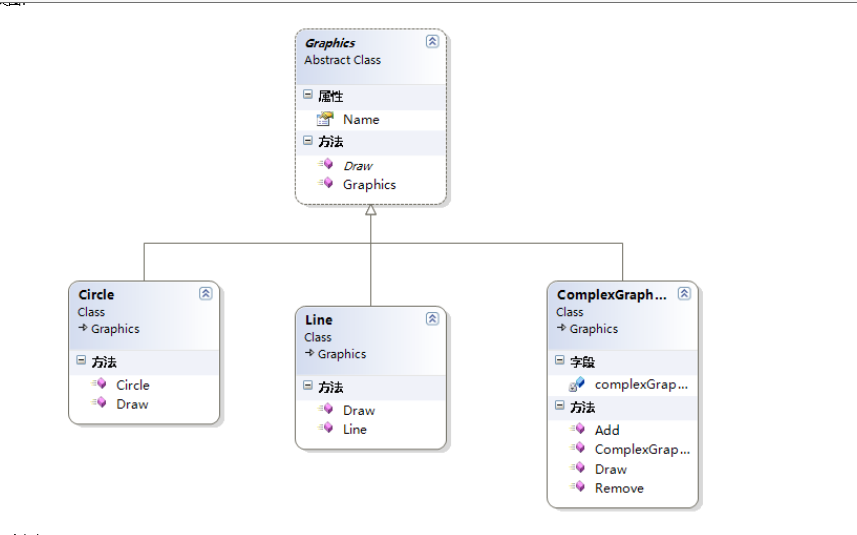


将对象传入下一级类，通过构造函数注入，形成功能的扩展

始终认定具体修饰的是 抽象的phone类型

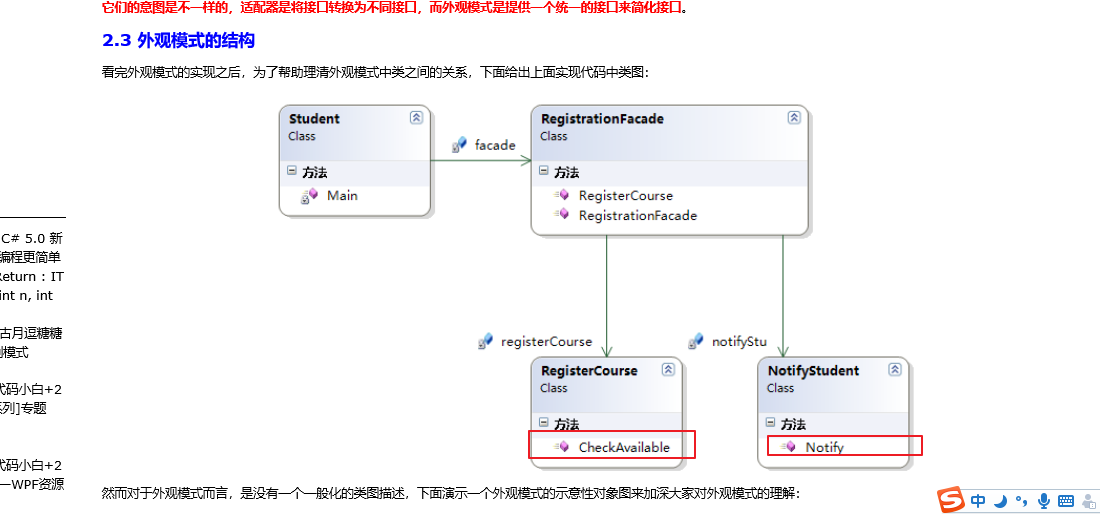
1. 组合

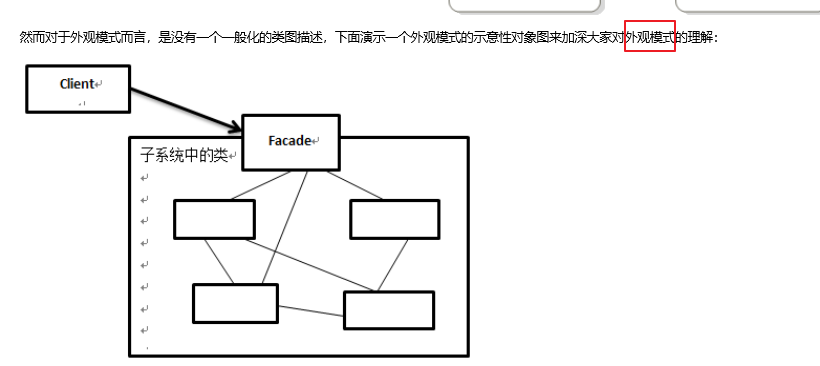
继承自同一个类，使用容器封装所有的实例对象，进行实例的统一处理



9．外观

使用一个外部接口类， 关联旧有的类实例，进行集中调用，较少客户端的实例化操作

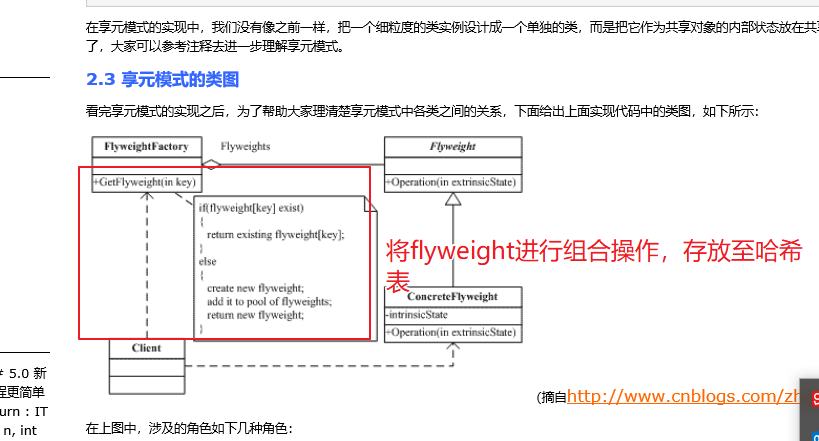




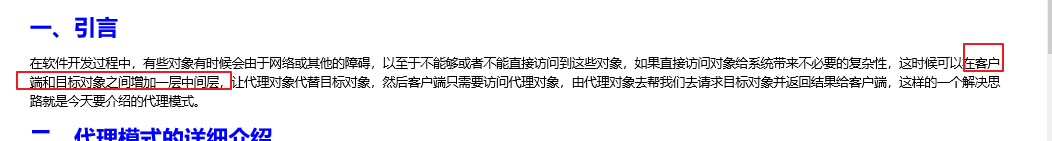
1. 享元模式

采用hash表集合，进行实例的存储，单需要进行使用是，将地址引用赋予对应的类型字段



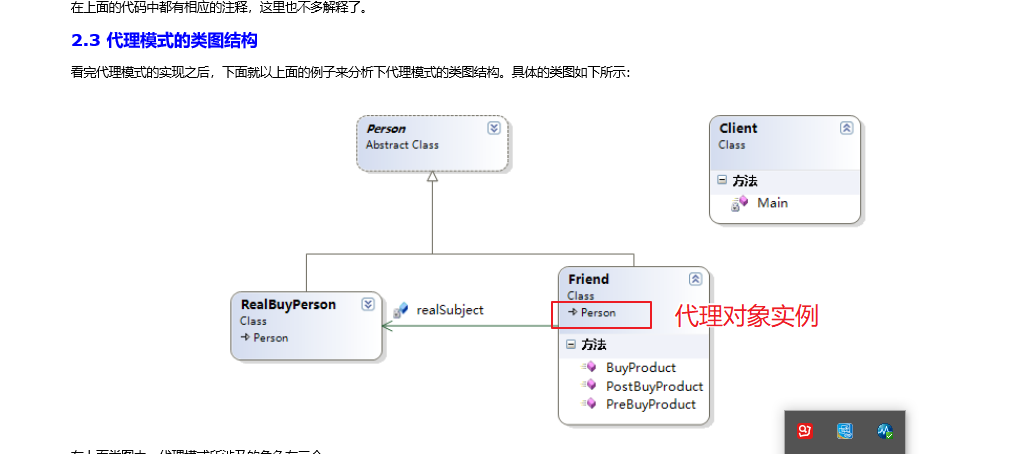


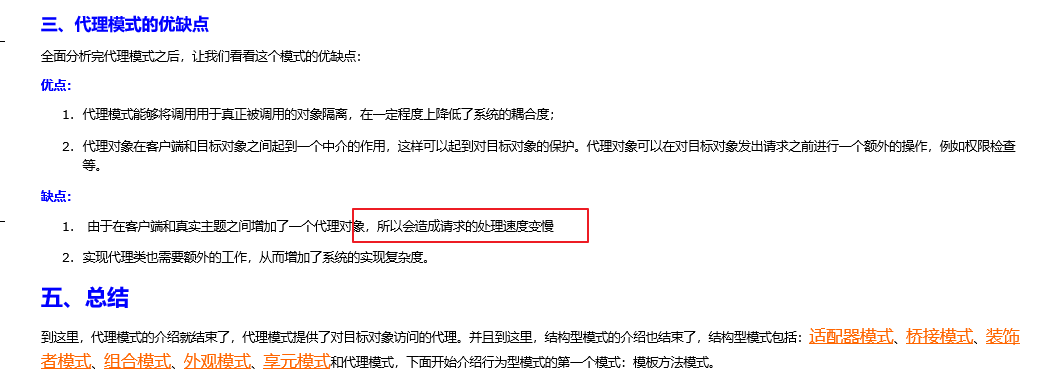
1. 代理



由代理对象控制对原有对象的使用，添加拓展的方法检查等







1. 模板

把相同的部分抽象出来到抽象类中去定义，具体子类来实现具体的不同部分，这个思路也正式模板方法的实现精髓所在

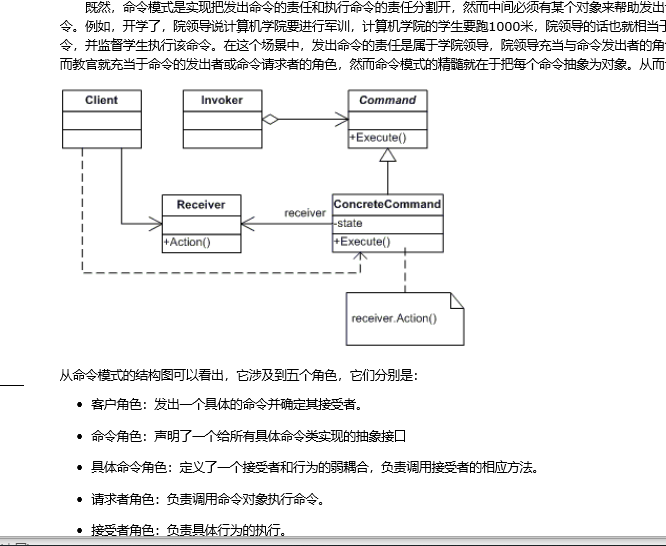


将变化的代码脱离重写，利用抽象类的特性。可实现对应的流程变化。



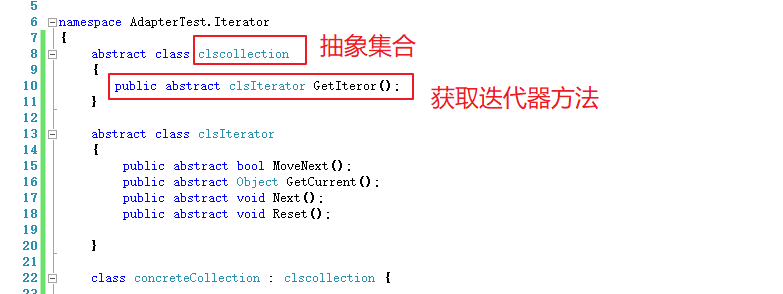
行为型模式

1. 命令模式

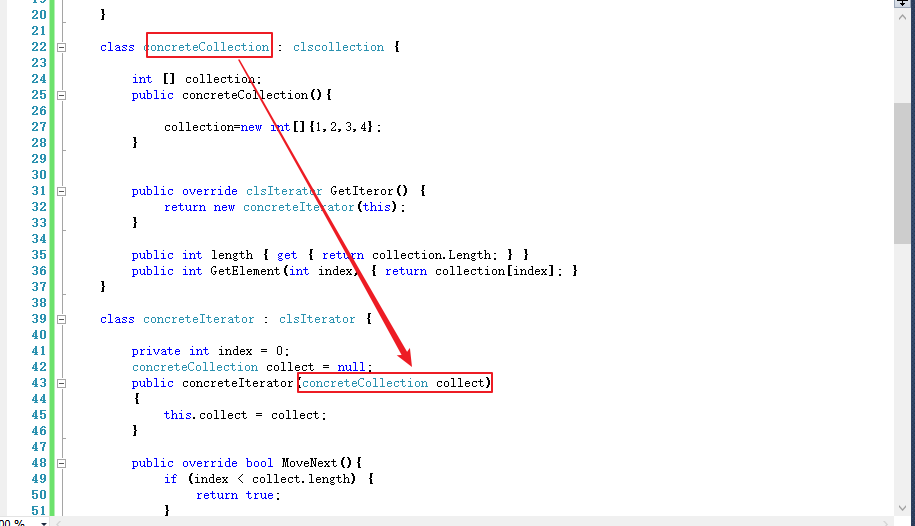


13．迭代器

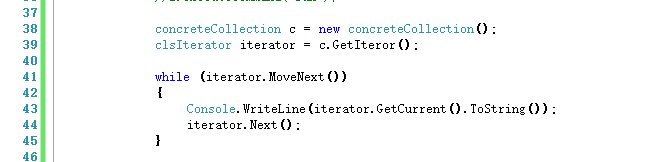
在集合中， 含有获取迭代器的方法定义



迭代器的实现，需要通过构造器传入对应的集合，在此基础上进行迭代

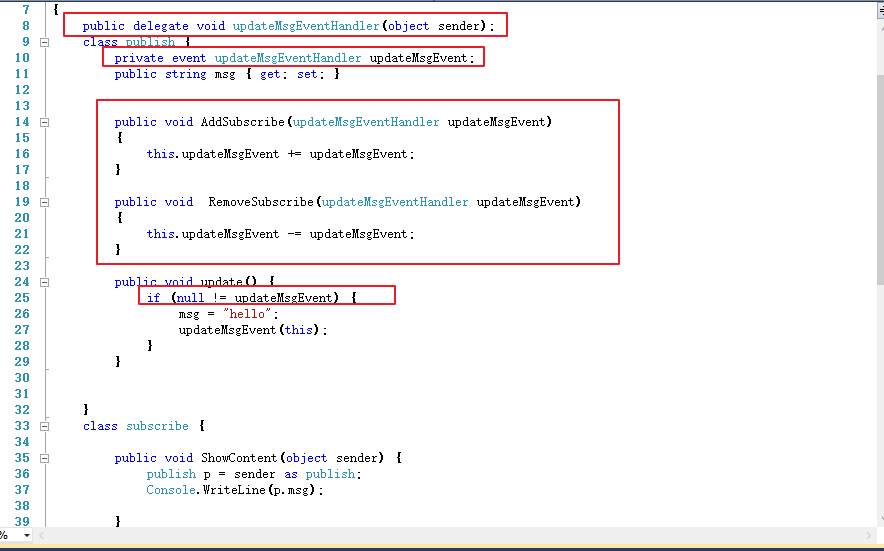


使用方式



1. 观察者

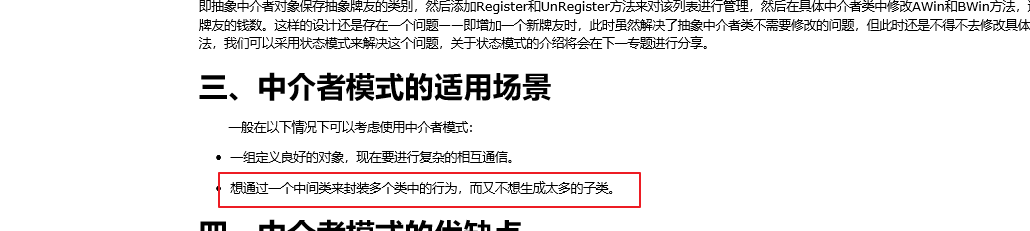
表示与逻辑分离。将具体的更新逻辑交由订阅者定义，而发布者仅仅提供消息。

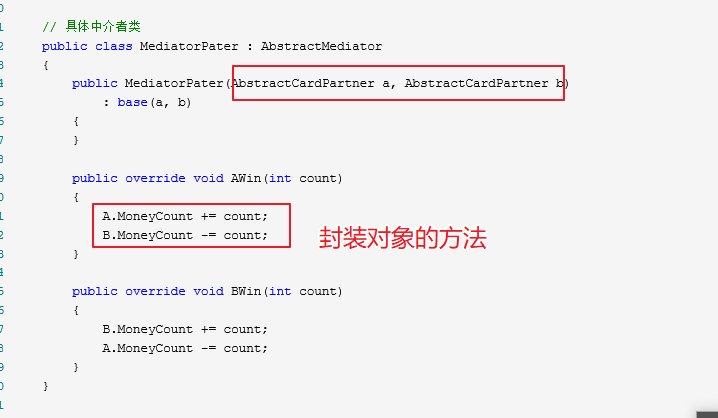




1. 中介者

使用中间类， 对特定的多个对象进行操作封装

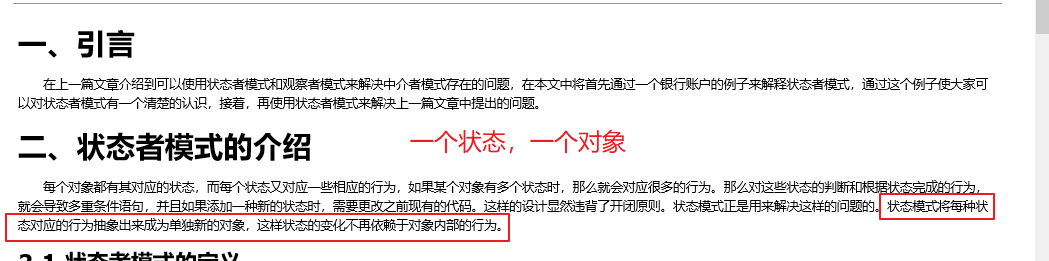


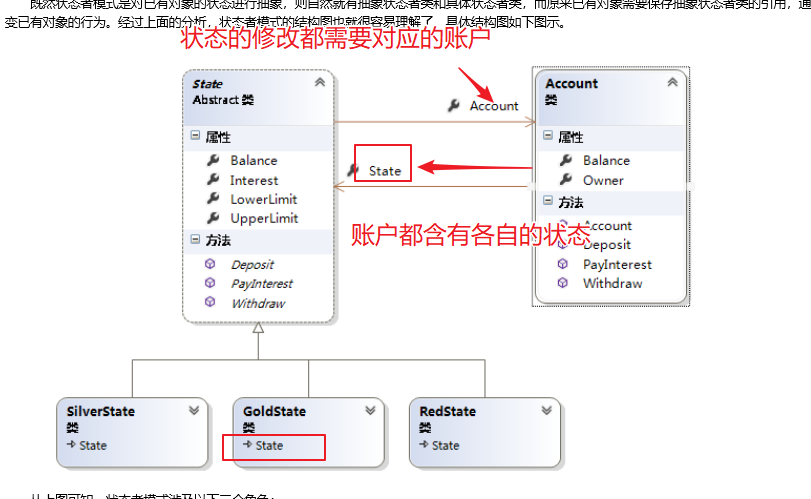


代理模式，外观模式，中介者模式，三者十分相像，应用场景不同

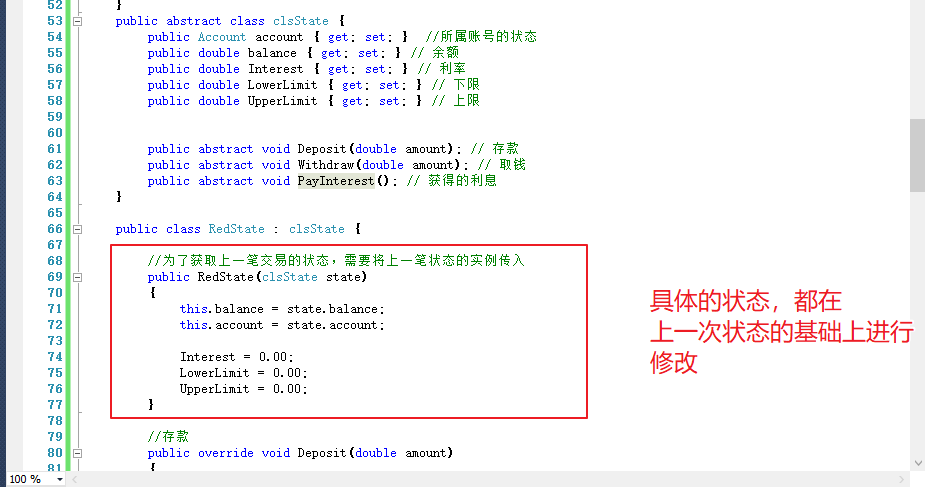
1. 状态

使用一个状态父类，对于某一个含有状态的的对象，利用多态在其中进行操作状态的变换。

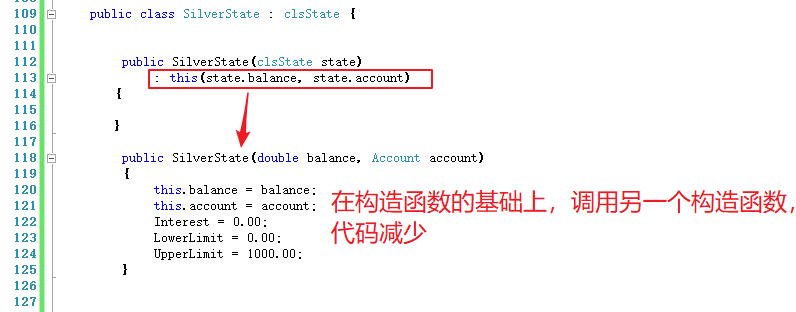






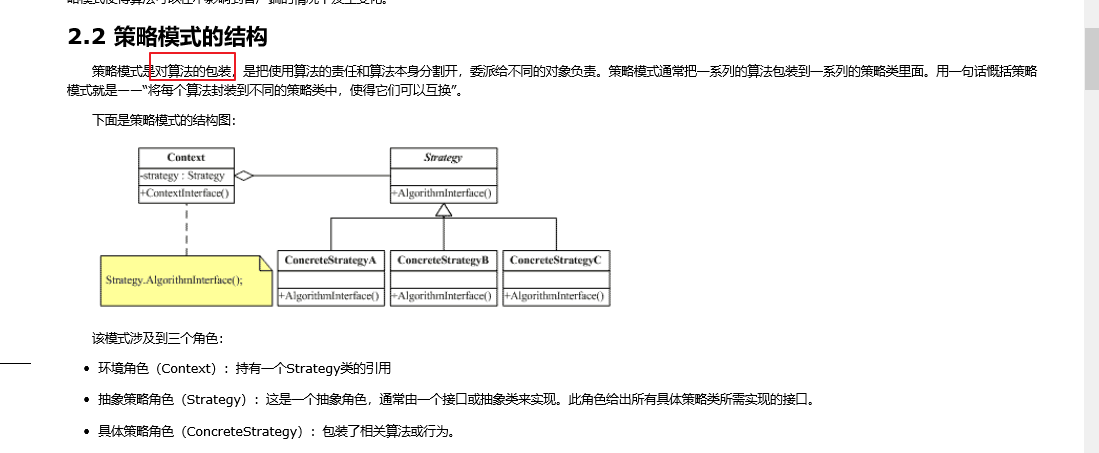


特殊语法：

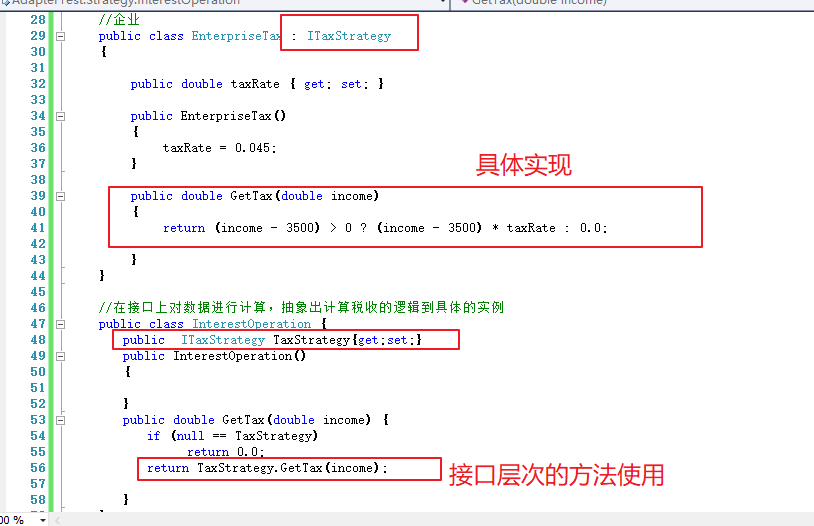


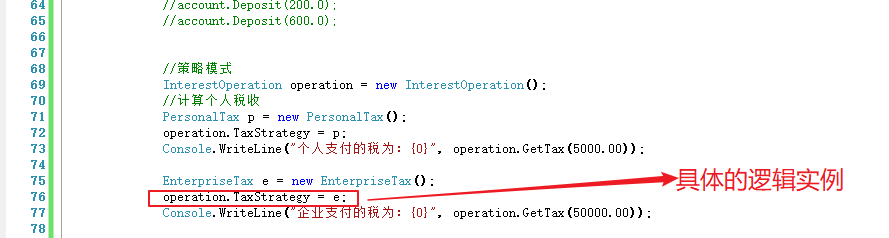
1. 策略模式

对策略（方法）进行抽象。



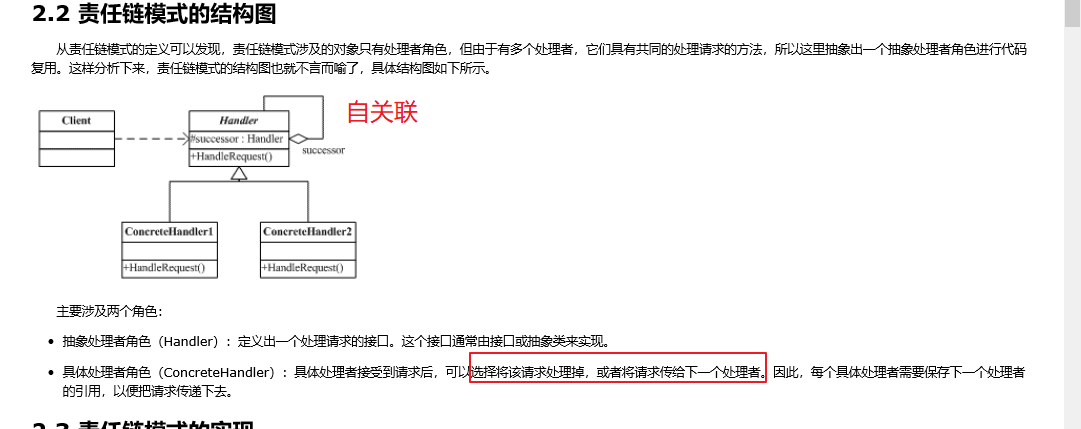
对于策略模式，实际上是在抽象层次（接口）上进行编程，在使用时，只需要传入对应的具体逻辑实例，即可灵活变换业务逻辑

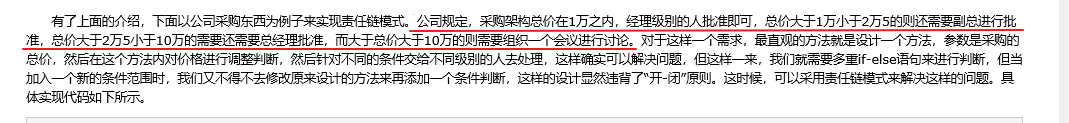




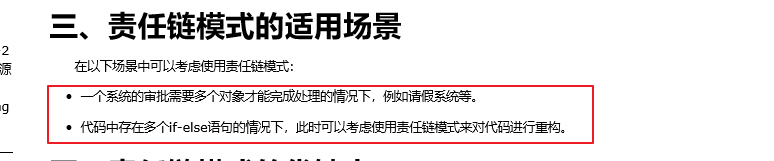
1. 责任链

将请求的发送者与接受者进行分割（解耦），请求在沿着链路发送时，可能经过的模块仅仅是处理验证。

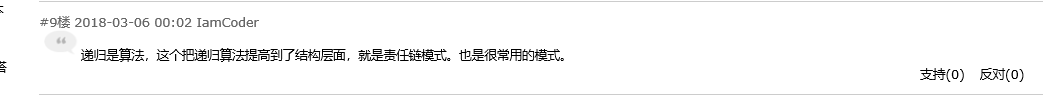




首先，我要知道 请求， 其次，判断是否移交下一个负责人处理，因此，需要知道更上层的负责人。 综上，请求， 下一级负责人

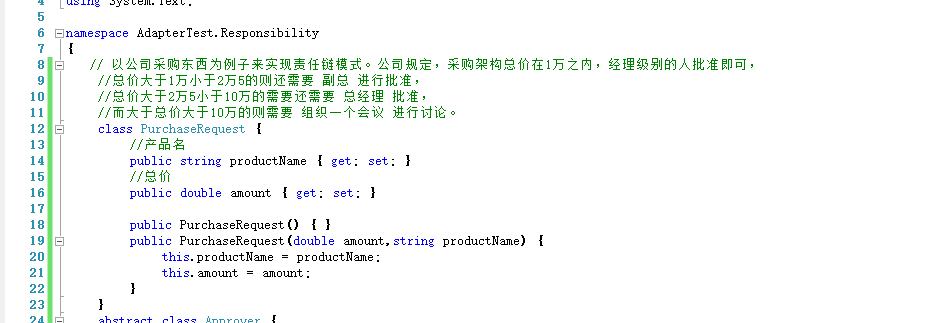


这是一种递归的算法？



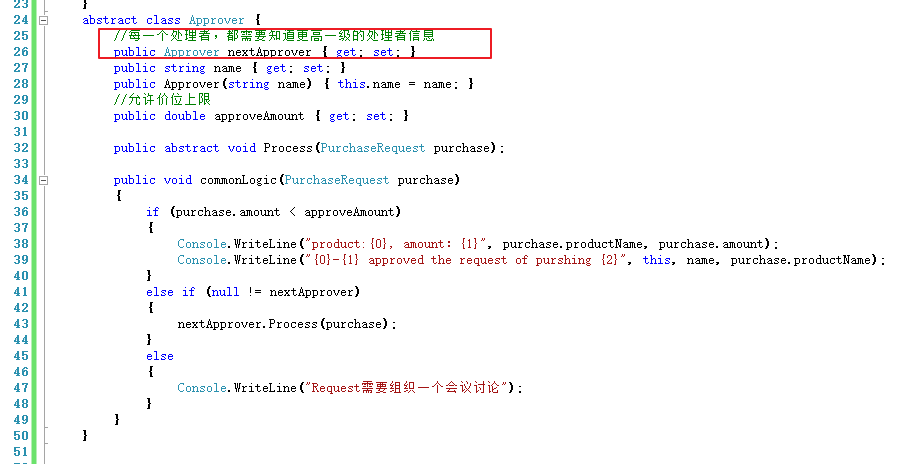
利用递归的形式(想象为都是同一类型的对象)，对请求进行处理

首先，确定请求的信息

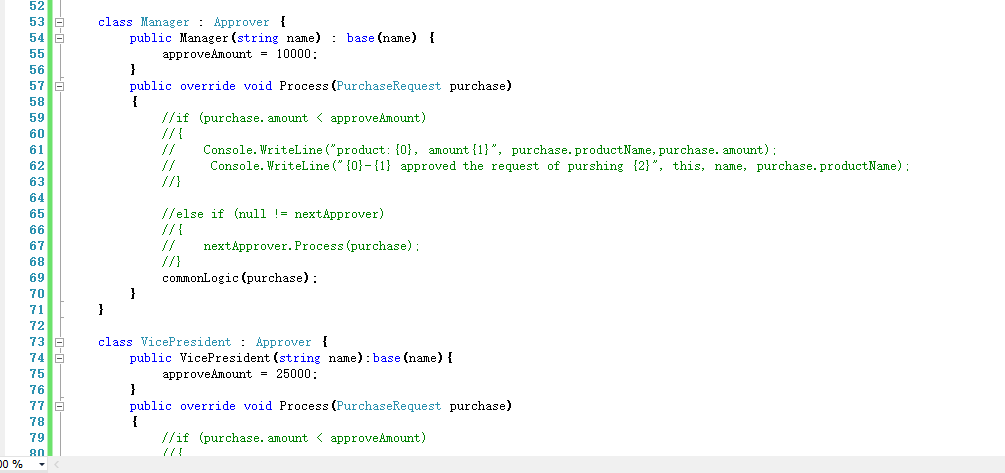


对责任链中的 对象进行抽象化，

因为每一个责任链的 对象，都需要知道下一个相同对象，因此，递归己身。



具体的对象实现





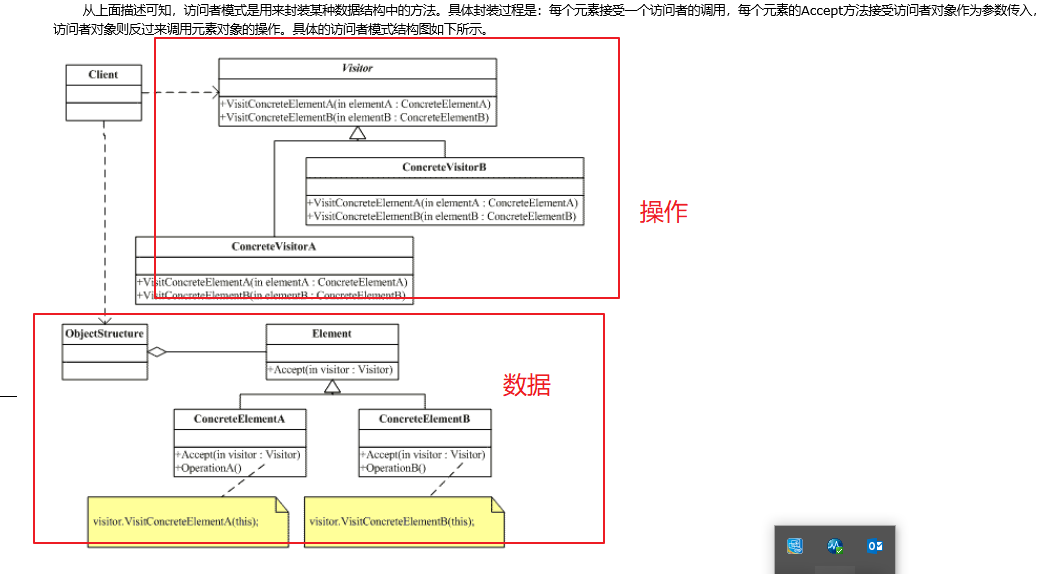
对于责任链的开始，从组装的第一个对象开始发送请求即可



此模式对于工厂等流程化作业 作用明显

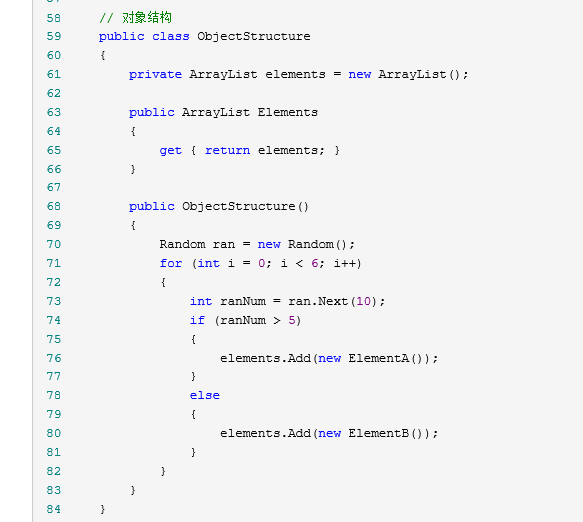
1. 访问者

将外部对象传入内部， 将自身信息传入访问者本身，进行逻辑的扩展

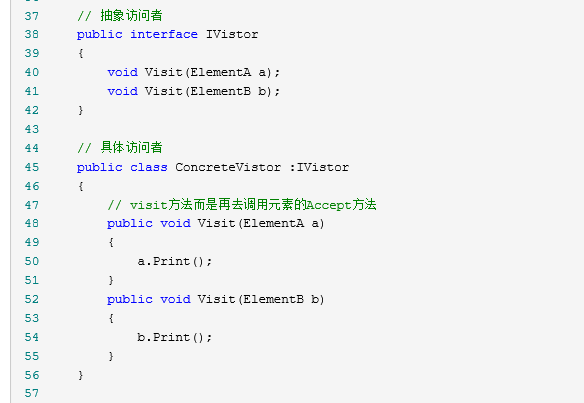


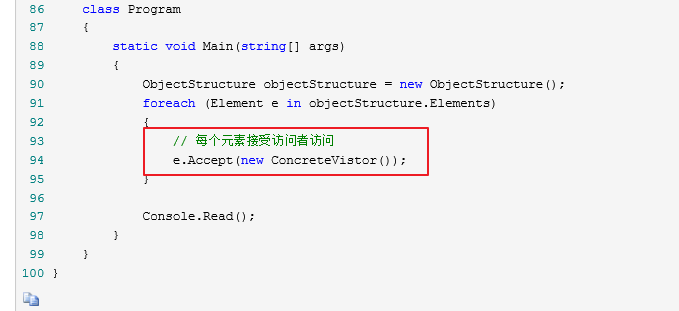
数据





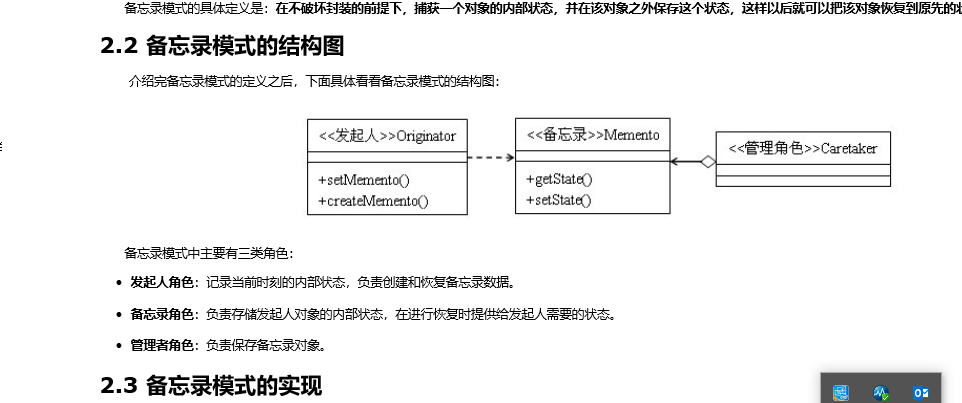
操作



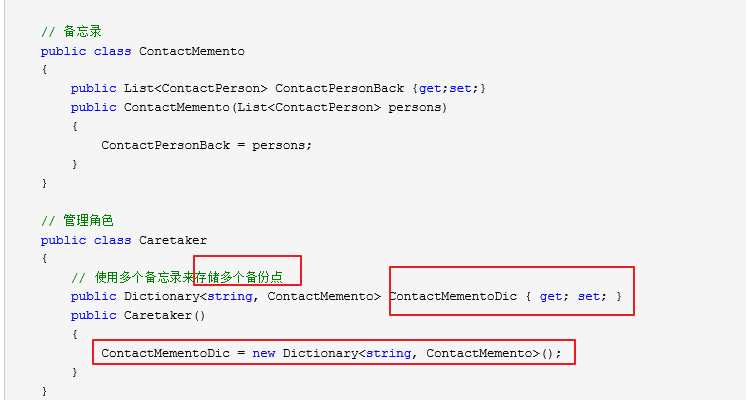


1. 备忘录

使用第三方管理者，进行对象的存储，备忘录充当介质









参考网站：https://www.cnblogs.com/KQNLL/p/9022497.html