# 设计原则



1. **封装变化** 找出应用中可能需要变化之处，把他们独立出来，不要和那些不需要变化的代码混在一起。换句话说，把会变化的部分取出并封装起来，以便以后可以轻易的改动或扩充此部分，而不影响不需要变化的其他部分。
2. **针对接口编程，而不是针对实现编程**。
3. **多用组合，少用继承**。
4. **为了交互对象之间的松耦合设计而努力**
5. **开闭原则**：类应该对扩展开放，对修改关闭。
6. **依赖倒置原则**：要依赖抽象，不要依赖具体类，这个原则说明：不能让高层组件依赖底层组件，而且，不管高层或低层组件，都应该依赖于抽象。
7. **最少知识原则**：要减少对象之间的交互，只和你的密友谈话。不要让太多的类耦合在一起，免得修改系统中一部分，会影响到其他部分。(外观模式)
8. **好莱坞原则** 别调用(打电话给)我们，我们会调用（打电话给）你。我们允许底层组件将自己挂钩到系统上，但是高层组件会决定什么时候和怎样使用这些底层组件。换句话说，只允许高层组件调用低层组件，反之则禁止。
9. **一个类应该只有一个引起变化的原因。**

# 策略模式

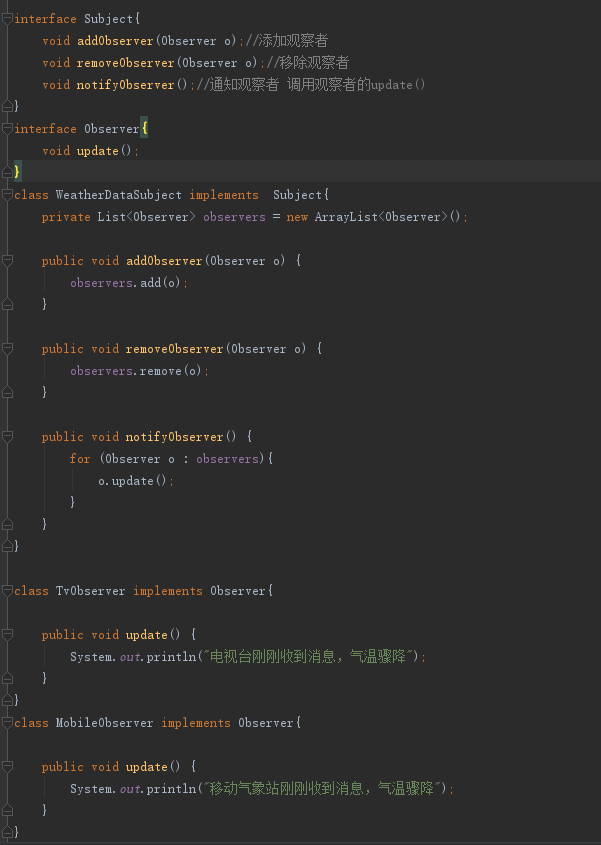
定义 策略模式定义了算法族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。可以在运行中更改类的算法。

使用场景



# 观察者模式

该模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，他的所有依赖者都会收到通知并自动更新



# 装饰者模式

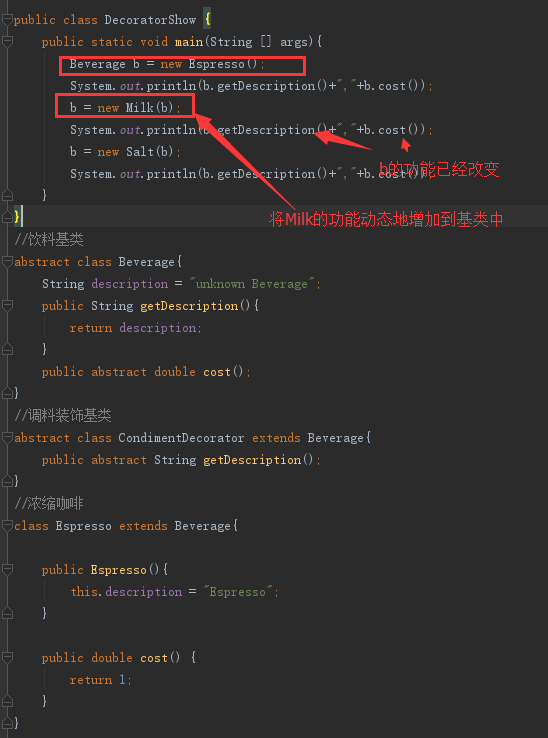
**定义：装饰者模式动态地将责任附加到对象上而不改变对象的类型。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。**

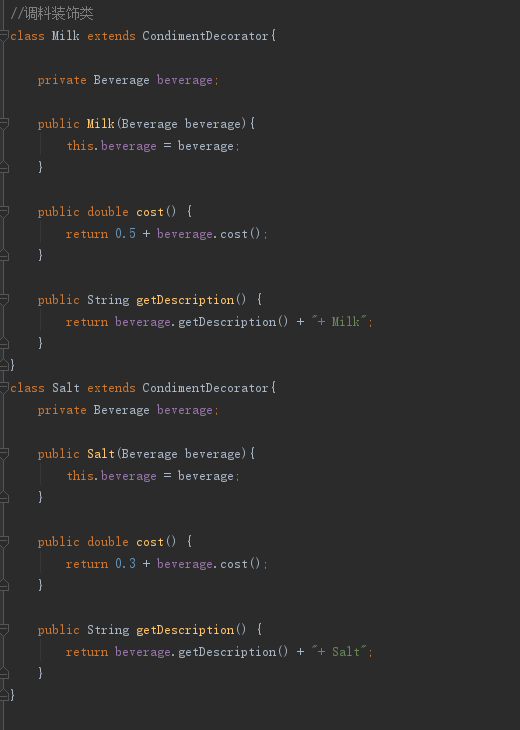
**特点：**

1. 装饰者和被装饰对象有相同的超类型
2. 可以用一个或多个装饰者包装一个对象
3. 既然装饰者和被装饰对象有相同的超类型，所以在任何需要原始对象(被包装的)的场合，可以用装饰过的对象代替它
4. 装饰者可以在被装饰者的行为之前与/或之后，加上自己的行为，以达到特定的目的
5. 对象可以在任何时候被装饰，所以可以在运行时动态地，不限量地用你喜欢的装饰者来装饰对象

缺点：

1. 小类会比较多，不太容易理解
2. 通常可以透明地插入装饰者，客户程序可能不知道是和装饰者打交道，但是有些代码可能会依赖特定的类型，而这样的代码一导入装饰者，就会出问题，尽量避免依赖特定类型的编码



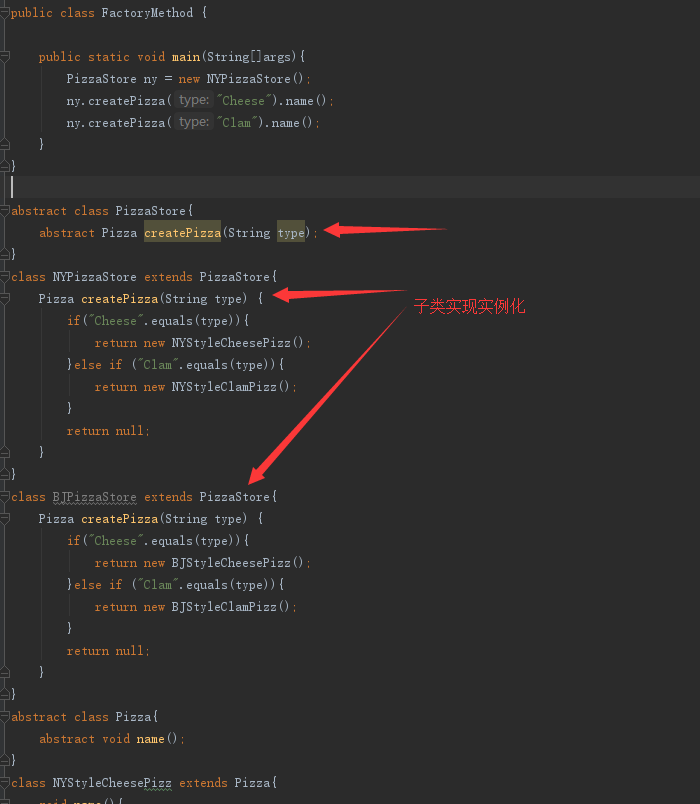
装饰类基类CondimentDecorator继承自Beverage，Milk和Salt继承自CondimentDecorator所以Mike和Salt仍是Beverage的子类，当不停地在装饰某个对象的时候，只是动态地给其增加功能而不改变类型。

# 工厂模式

## 简单工厂

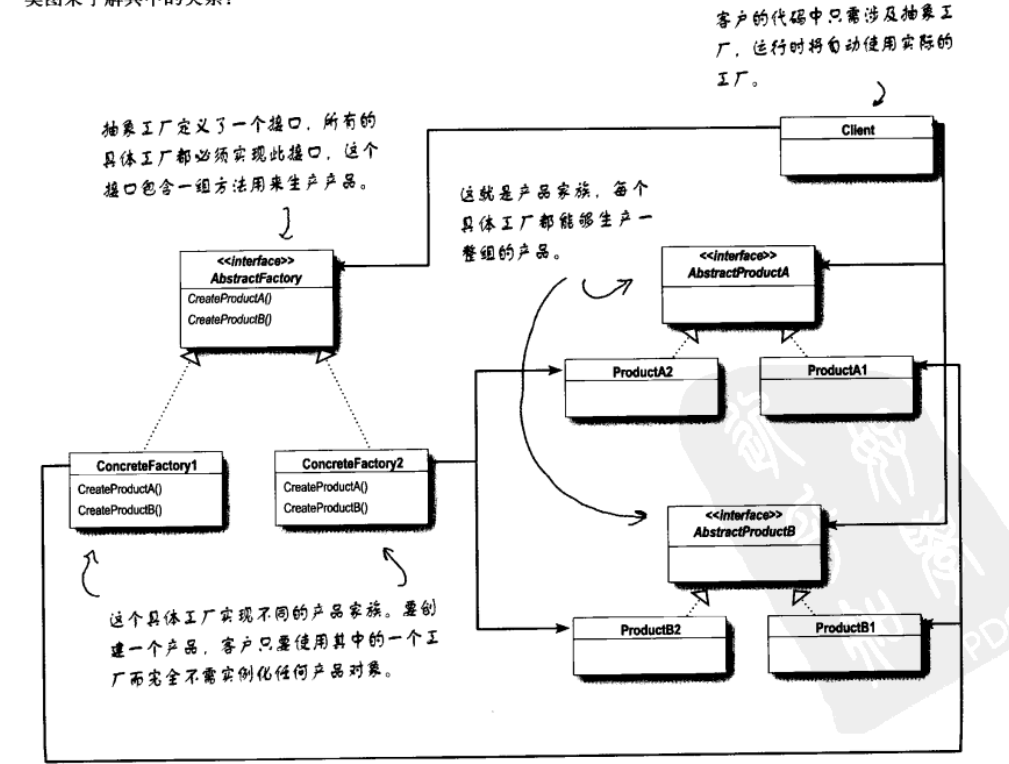
## 工厂方法模式

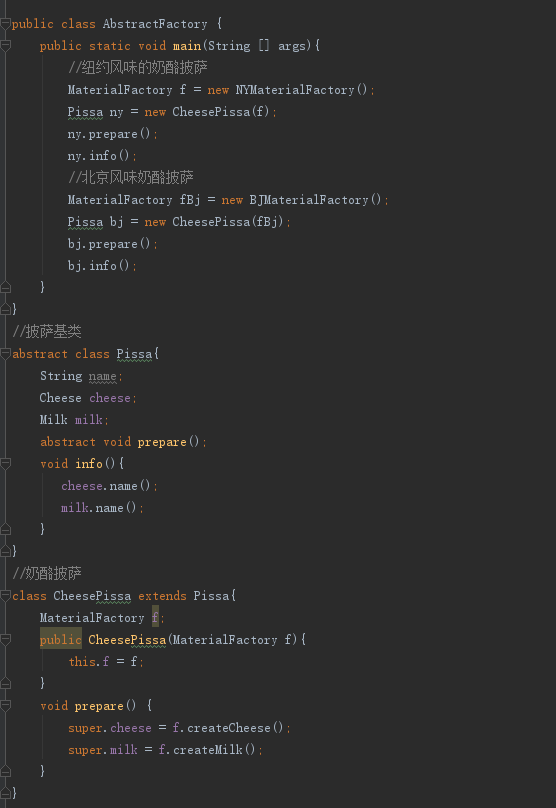
工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。



## 抽象工厂模式

抽象工厂提供一个接口，***用于创建相关或依赖对象的家族***，而不需要明确指定具体类。这样客户就从具体的产品中被解耦。



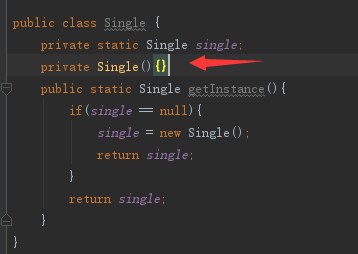




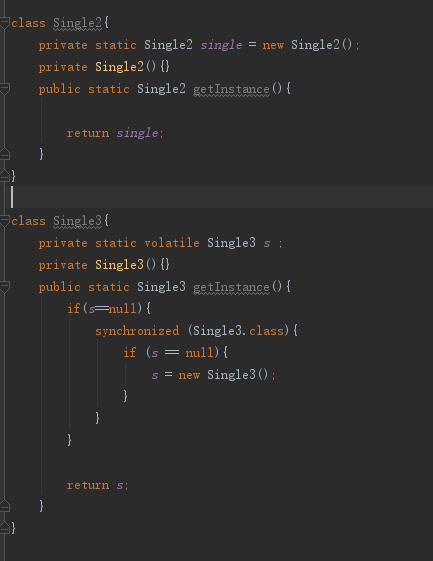
# 单例模式

确保一个类只有一个实例，并提供一个全局访问点。

经典的单例模式 但是多线程会出现问题



解决同步问题



# 命令模式

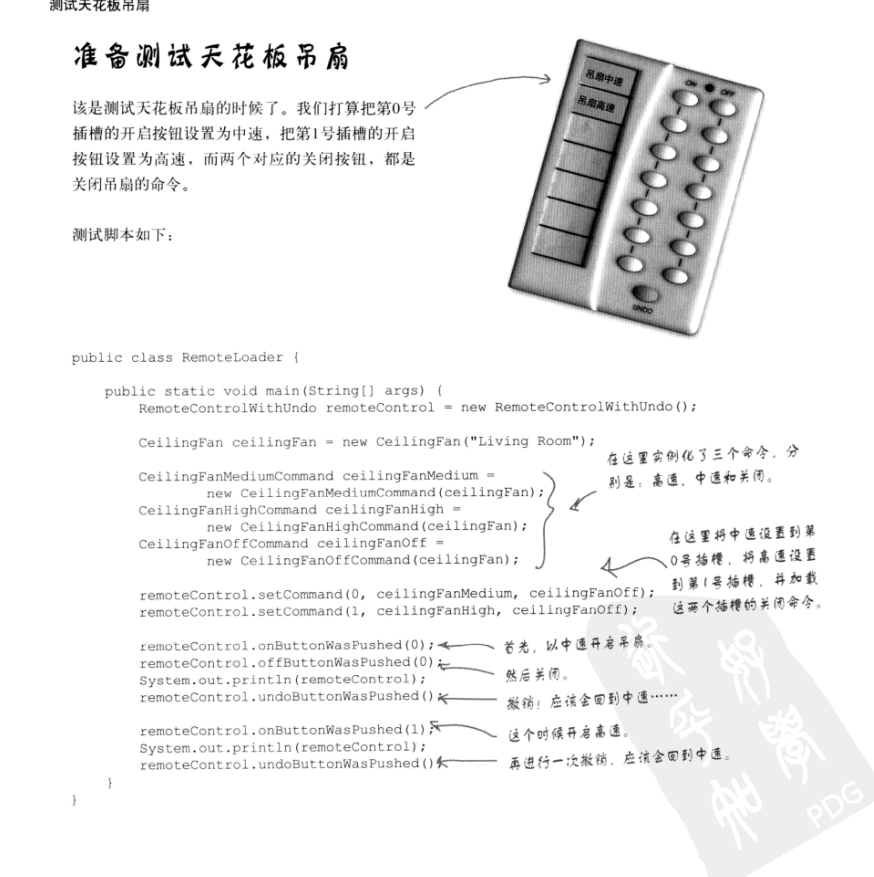
## 定义

将请求(动作)封装成对象，以便使用不同的请求、队列或日志来参数化其他对象。命令模式也支持可撤销的操作。

## 要点

1. 命令模式将发出请求的对象和执行请求的对象解耦
2. 在被解耦的两者之间是通过命令对象进行沟通的命令对象封装了接收者和一个或一组动作
3. 调用者通过调用命令对象的execute()发出请求，这会使得接收者的动作被调用
4. 调用者可以接受命令当做参数，甚至在运行时动态地进行
5. 命令可以支持撤销，做法是实现一个undo()方法来回到excute()执行前的状态
6. 宏命令是命令的一种简单的延伸，允许调用多个命令。宏方法也可以支持撤销
7. 实际操作时，很常见使用 “聪明”命令对象，他就是直接实现了请求，而不是将工作委托给接收者
8. 命令也可以用来实现日志和事务系统

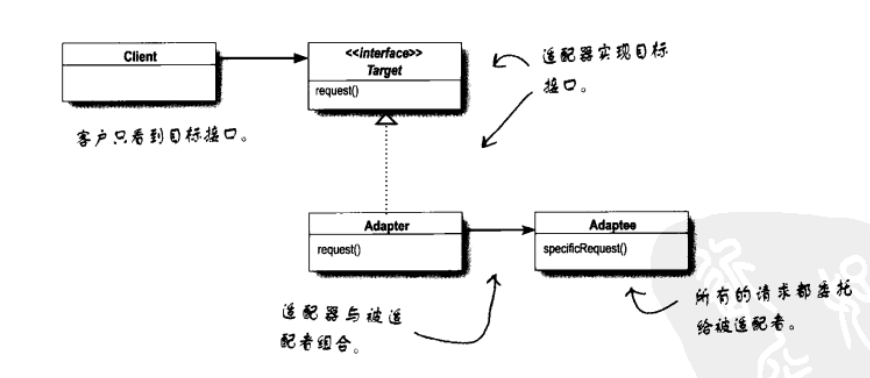


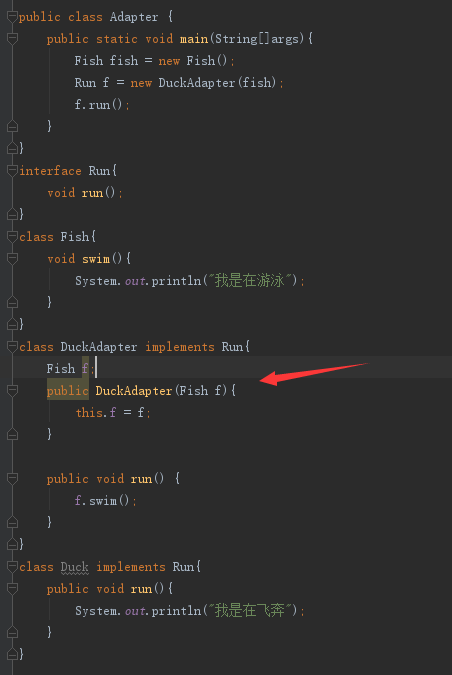


# 适配器

## 定义

将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类可以合作无间。





# 外观模式

## 定义

提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

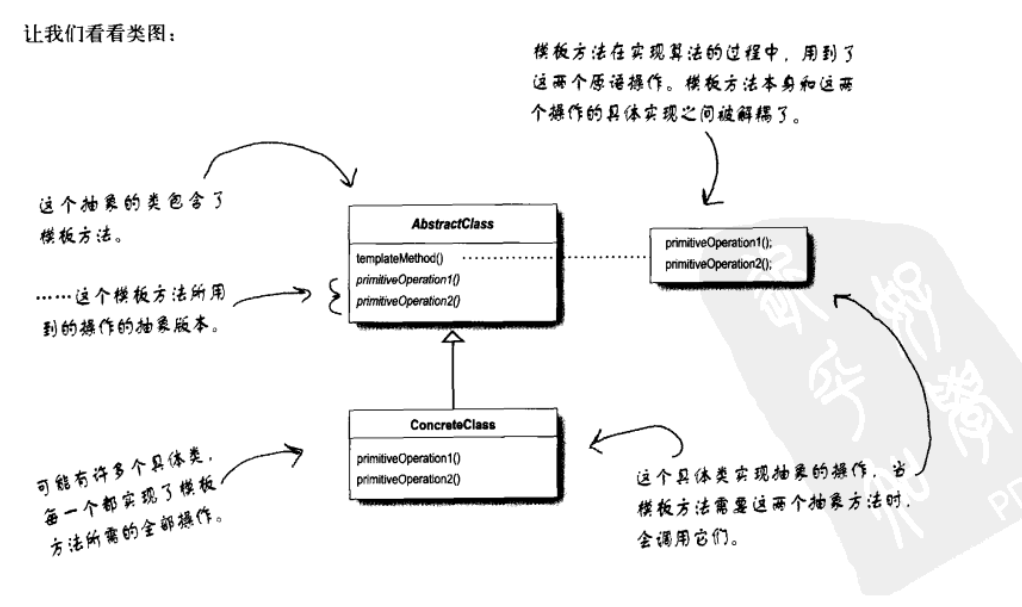
意图是提供子系统的简化接口，适配器的意图是将接口转换成不同的接口

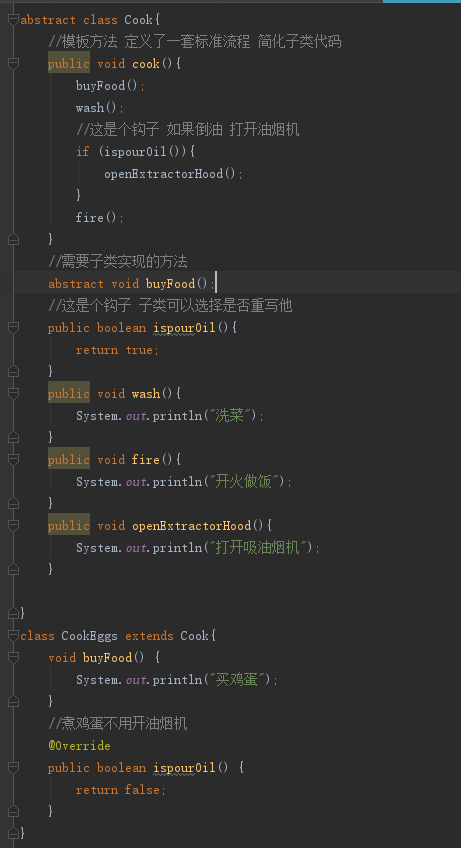
# 模板方法模式

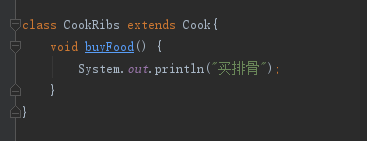
## 定义

**模板方法**在一个方法中定义了一个算法的骨架，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。

**钩子**的定义 在父类中提供一个空的或者默认实现的方法，子类可以根据需要选择是否重写该方法。





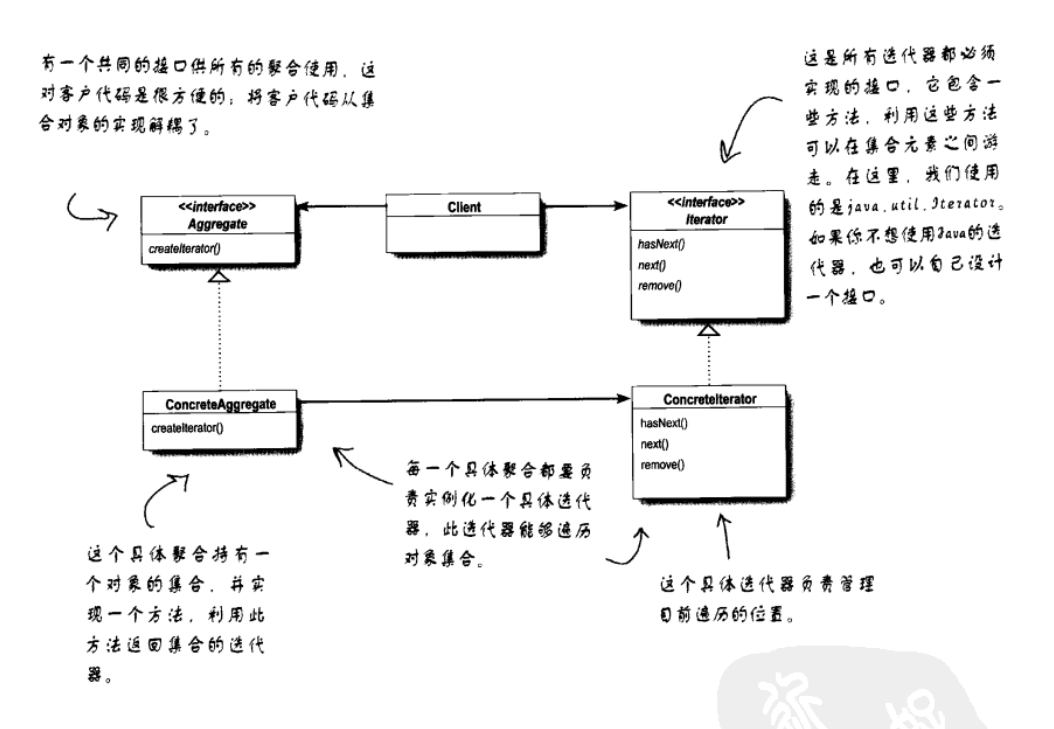


# 迭代器模式

## 定义

提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

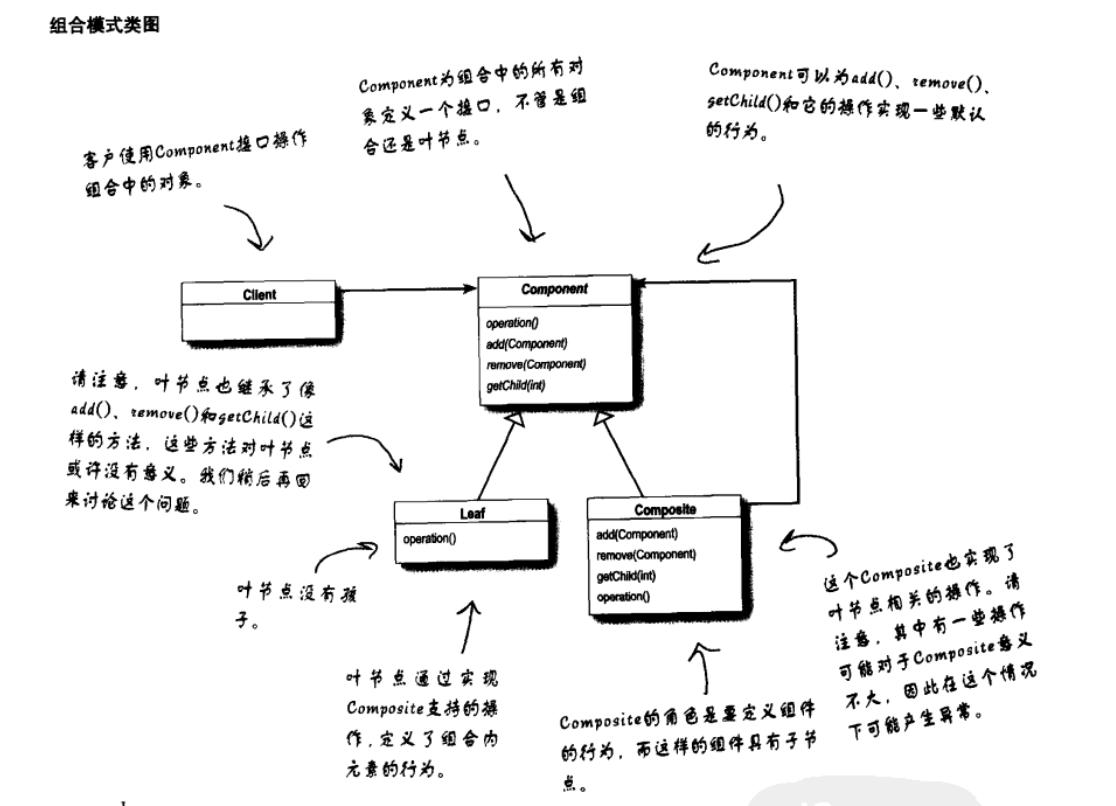
java.utils.Iterator

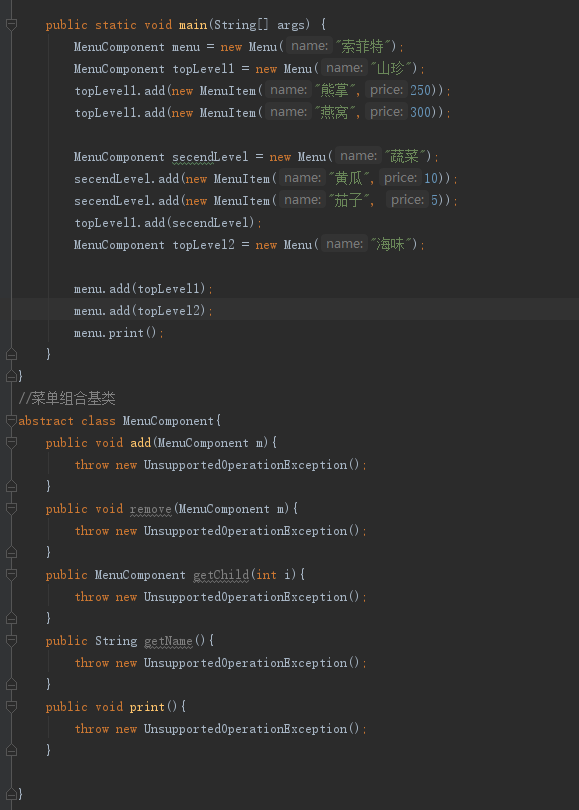


# 组合模式

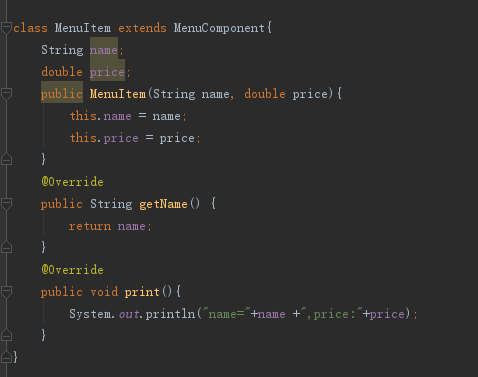
## 定义

允许你将对象组合成**树形结构**来表现“**整体/部分**”层次结构。组合能让客户以一致的方式处理个别对象一级对象组合





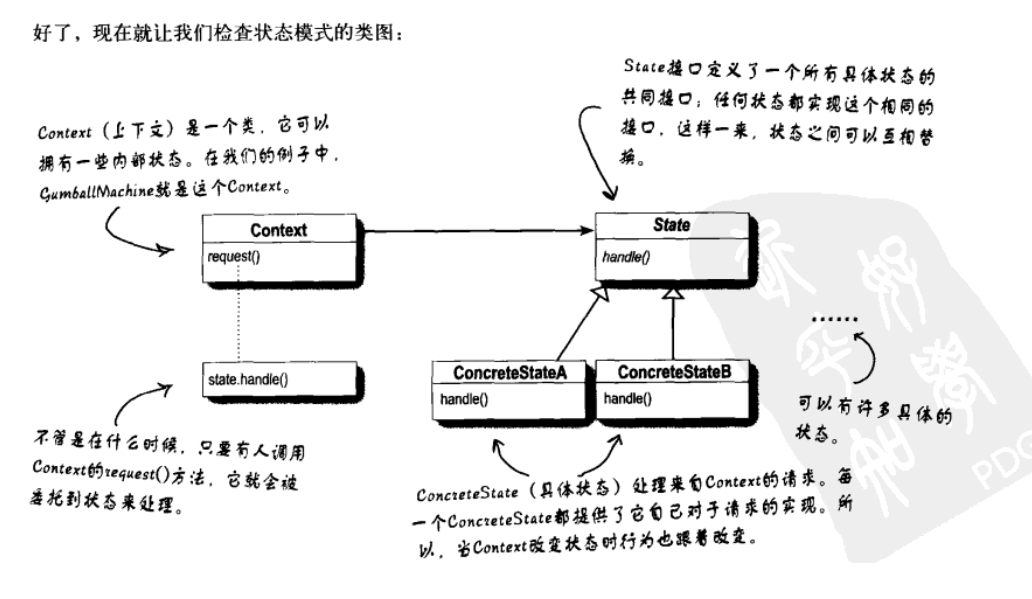


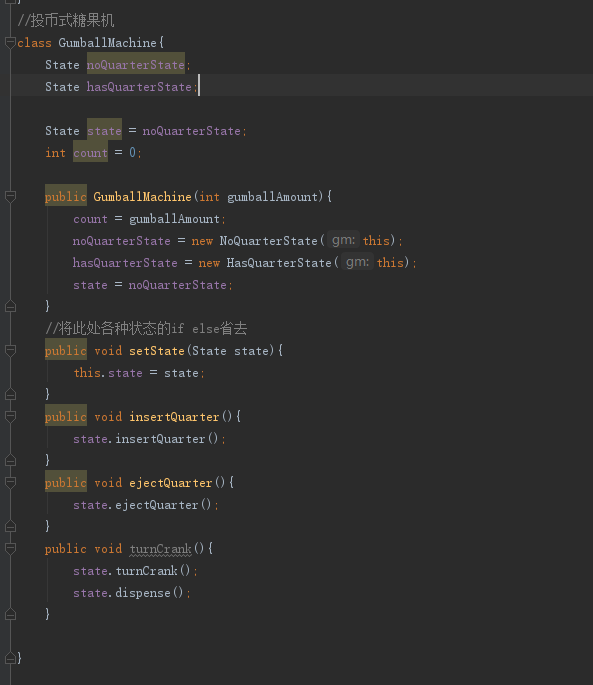


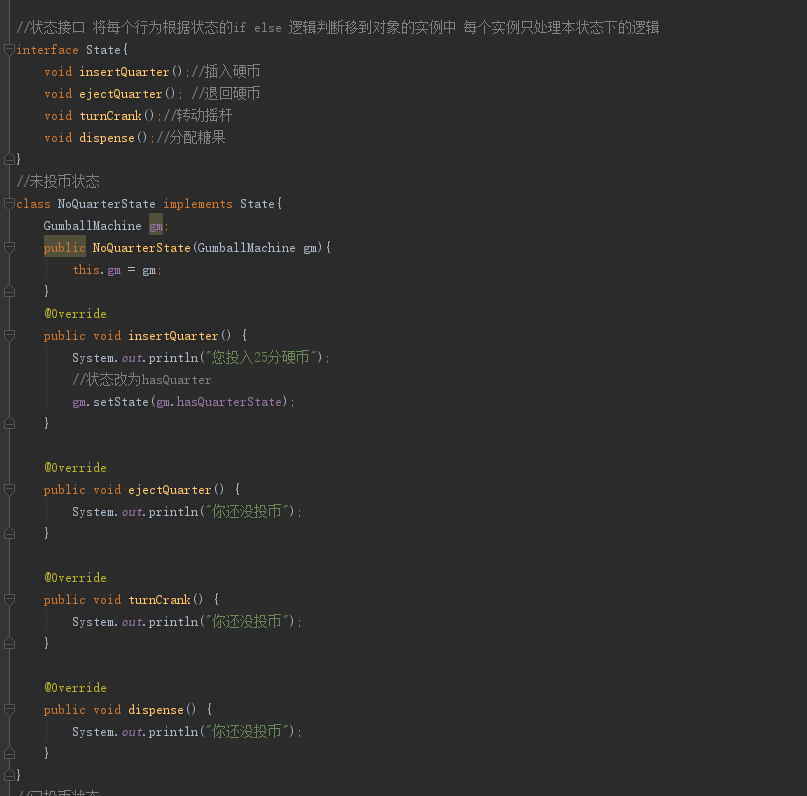
# 状态模式

## 定义

允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。





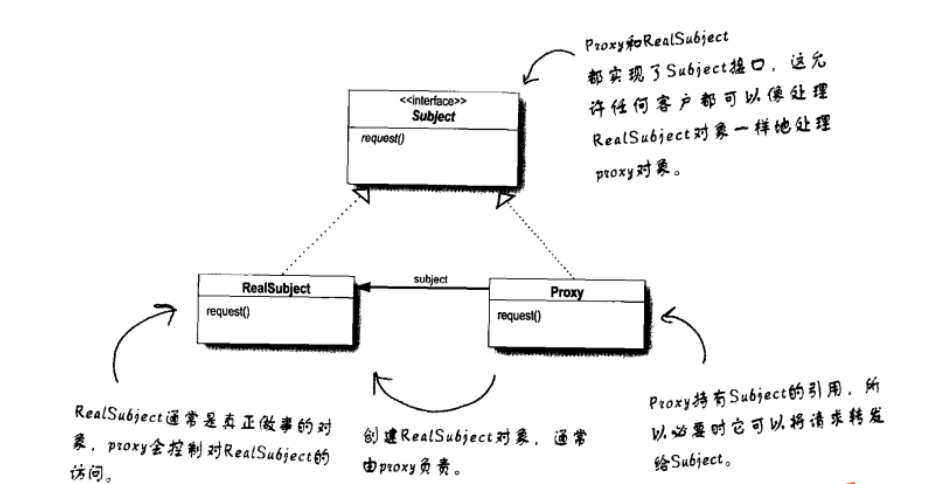




# 代理模式

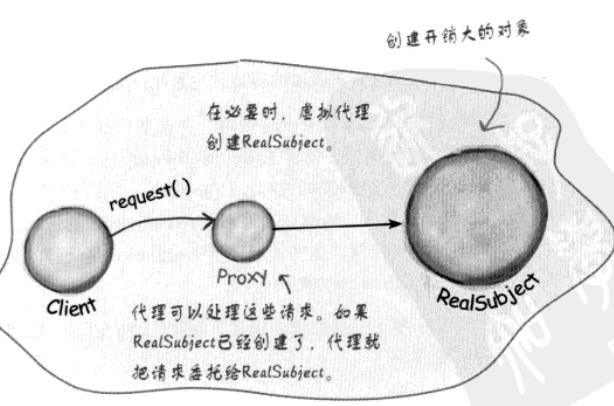
## 定义

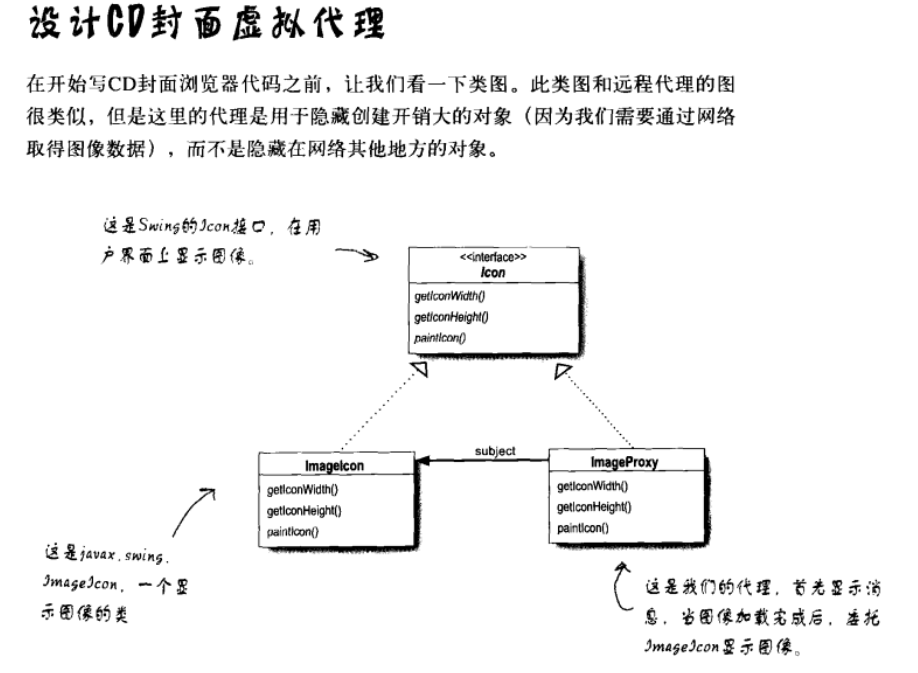
代理模式为另一个对象提供一个替身或占位符以控制对这个对象的访问。被代理的对象可以是远程的对象、创建开销大的对象或需要安全控制的对象。



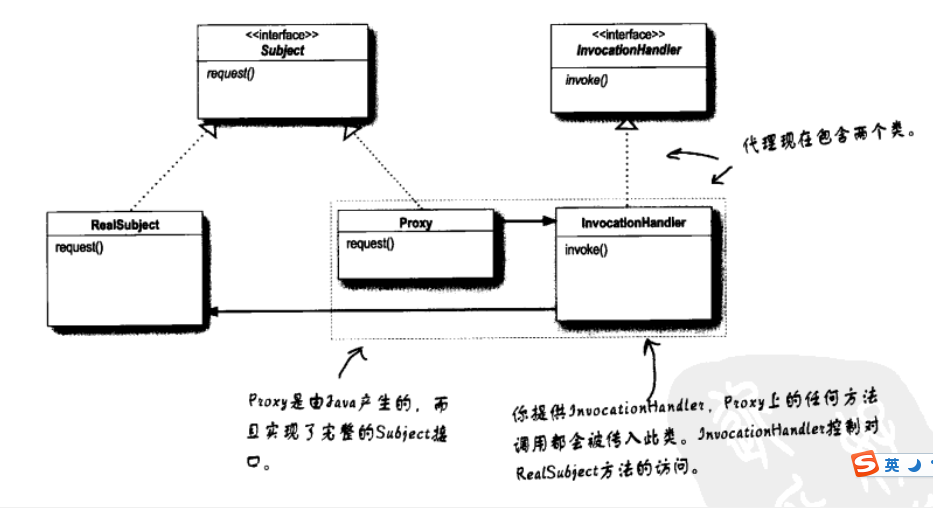
## 虚拟代理

作为创建开销大的对象的代表。虚拟代理经常直到我们真正需要一个对象的时候才创建它。当对象在创建前和创建中时，由虚拟代理来扮演对象的替身。对象创建后，代理就会将请求直接委托给对象。

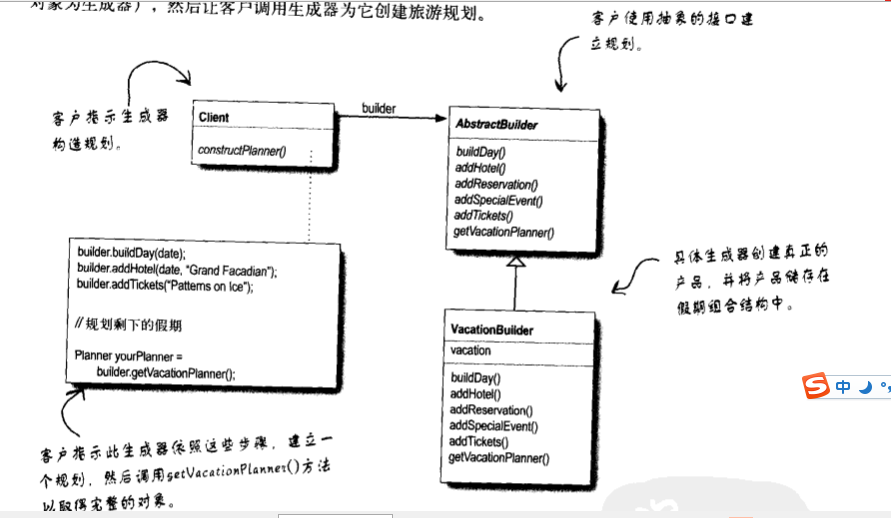


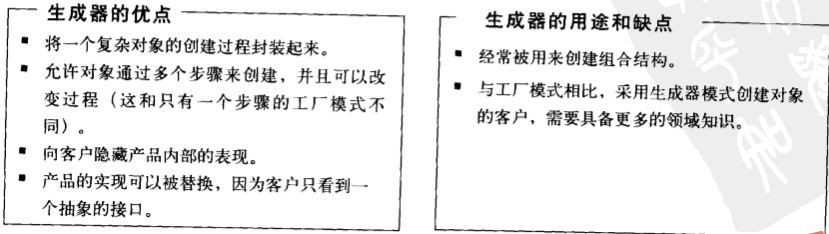


## 保护代理(动态代理)



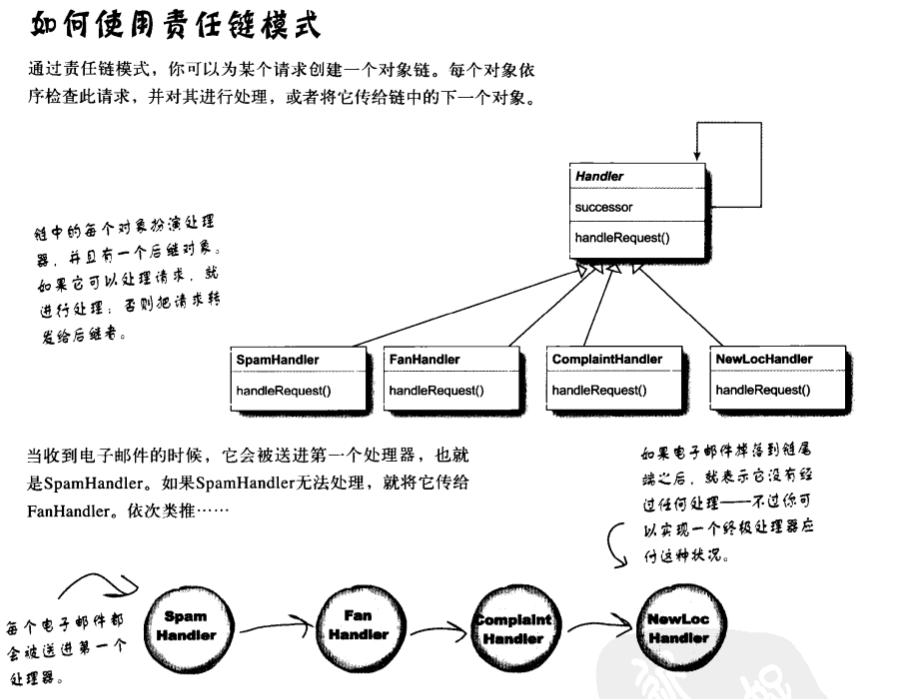
# 生成器模式

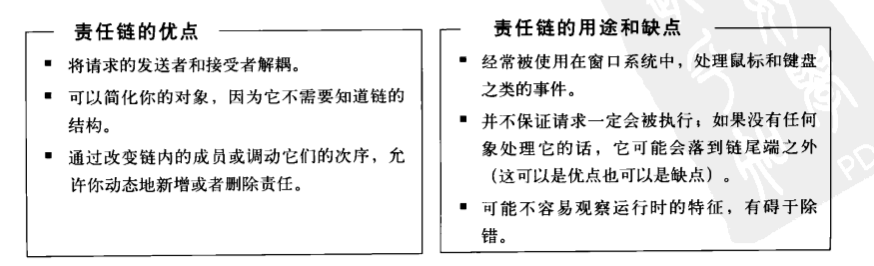




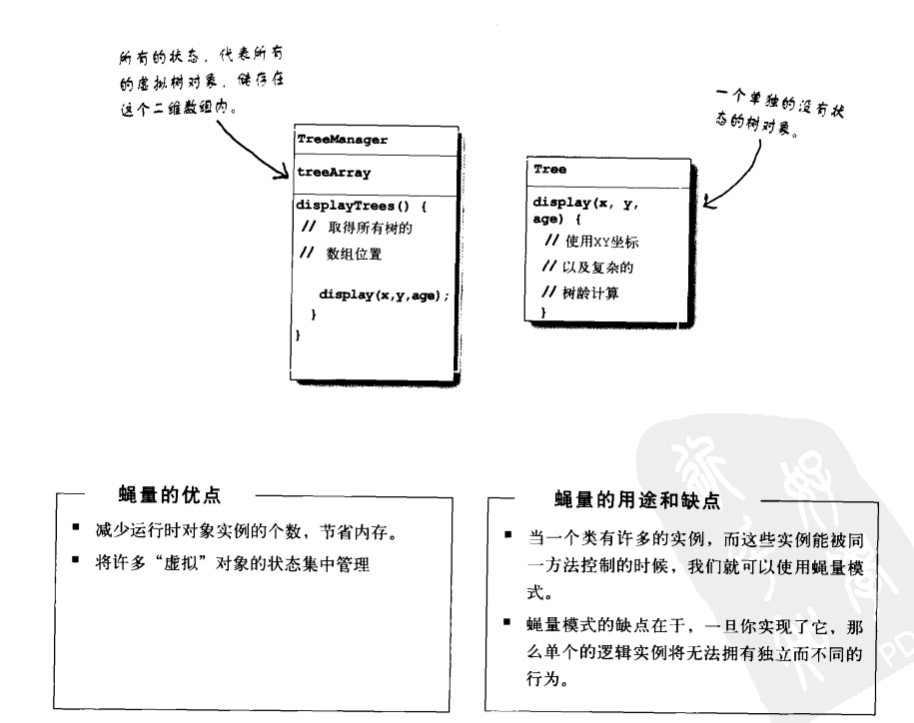
# 责任链

当你想要让一个以上的对象有机会能够处理某个请求时候，可以使用责任链模式。

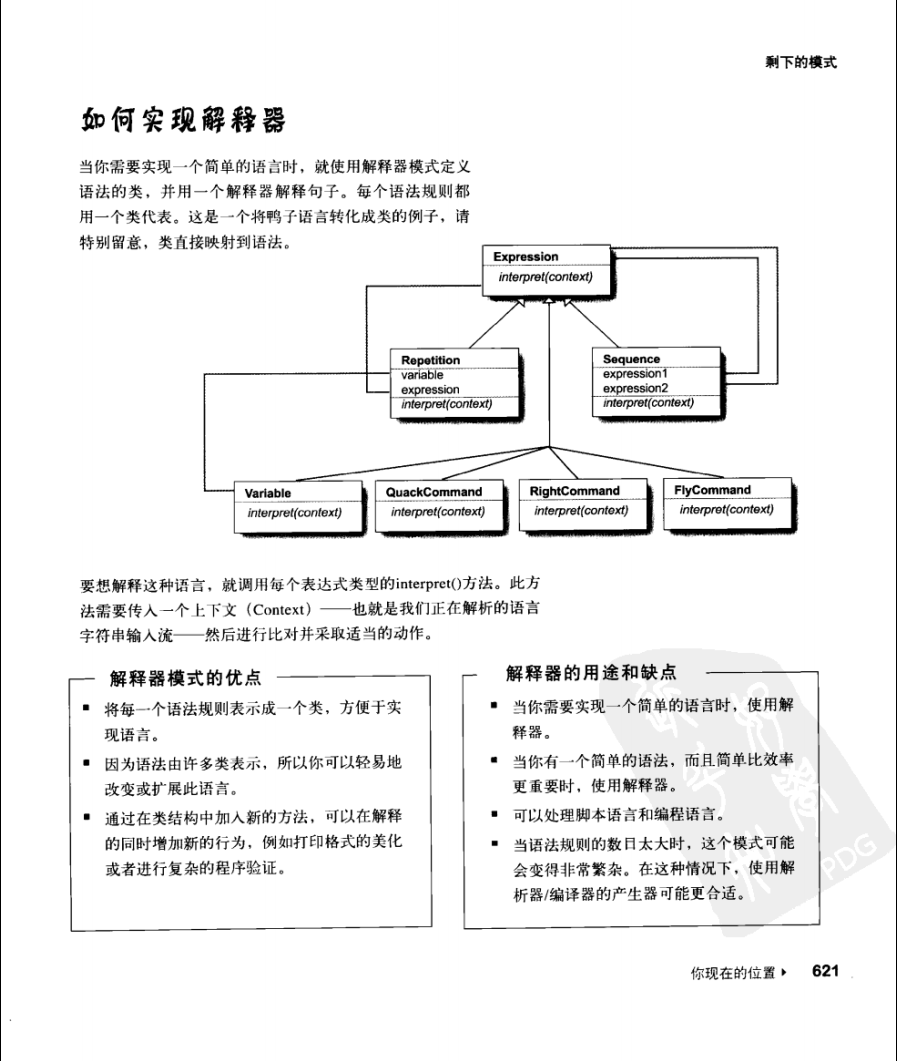




# 蝇量模式

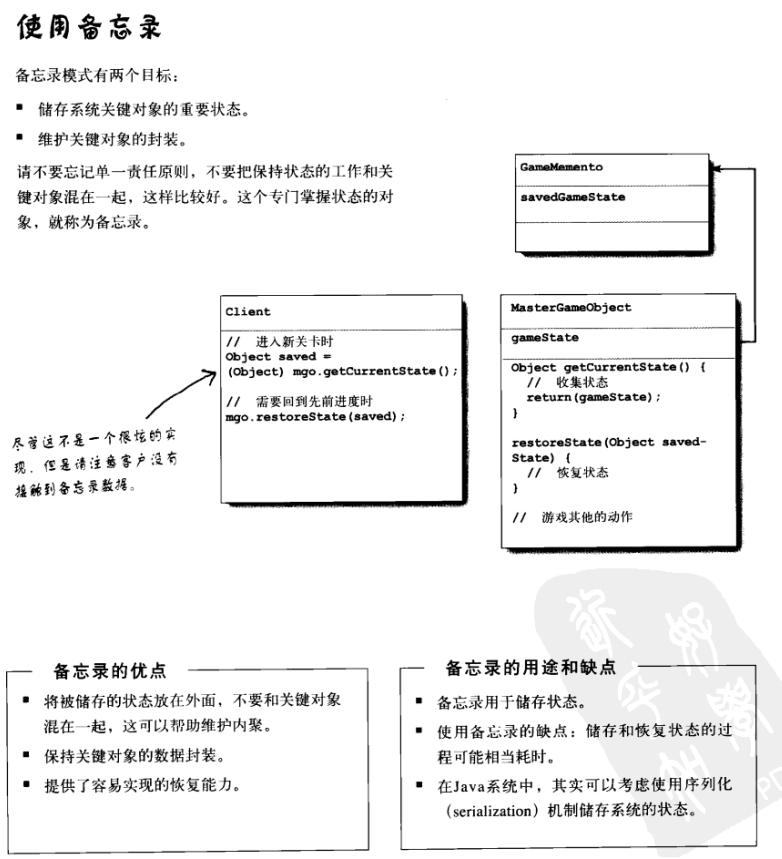


# 解释器模式



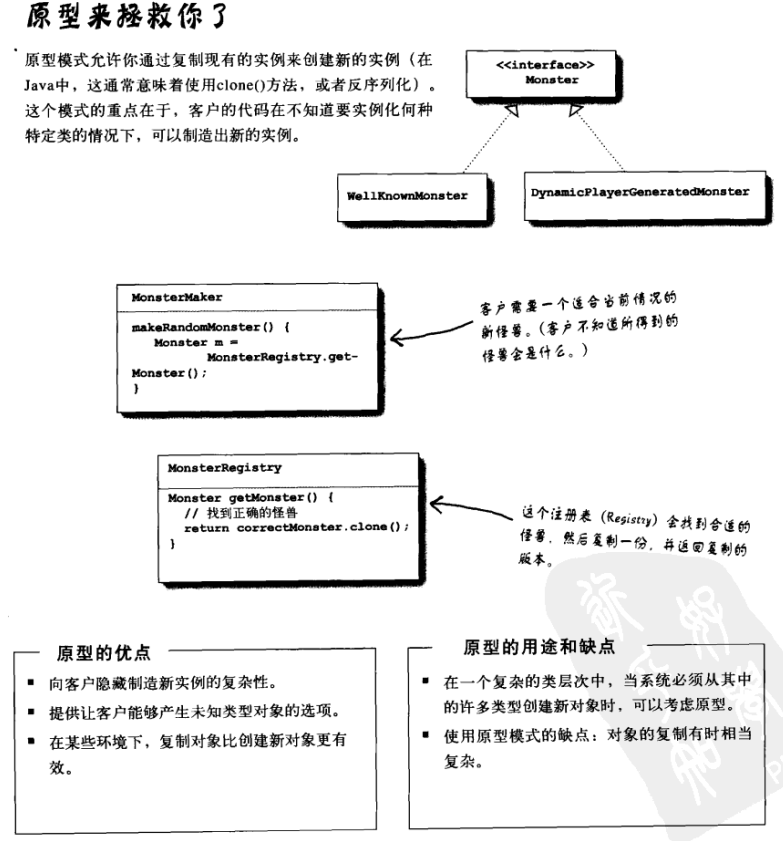
# 备忘录模式

当你需要让对象返回之前的状态 例如撤回 就是用该模式。



# 原型模式

当创建给定类的实例过程很昂贵或很复杂时，适用。



# 访问者

想要为一个对象的组合增加新的能力，且封装并不重要时，适用。

