设计模式

**软一 郭怀江 1917700121**

目录

[简单工厂模式 2](#_Toc75287246)

[策略模式 3](#_Toc75287247)

[代理模式 3](#_Toc75287248)

[装饰模式 4](#_Toc75287249)

[工厂方法模式 5](#_Toc75287250)

[原型模式 6](#_Toc75287251)

[外观模式 7](#_Toc75287252)

[模板方法模式 8](#_Toc75287253)

[建造者模式 8](#_Toc75287254)

[观察者模式 8](#_Toc75287255)

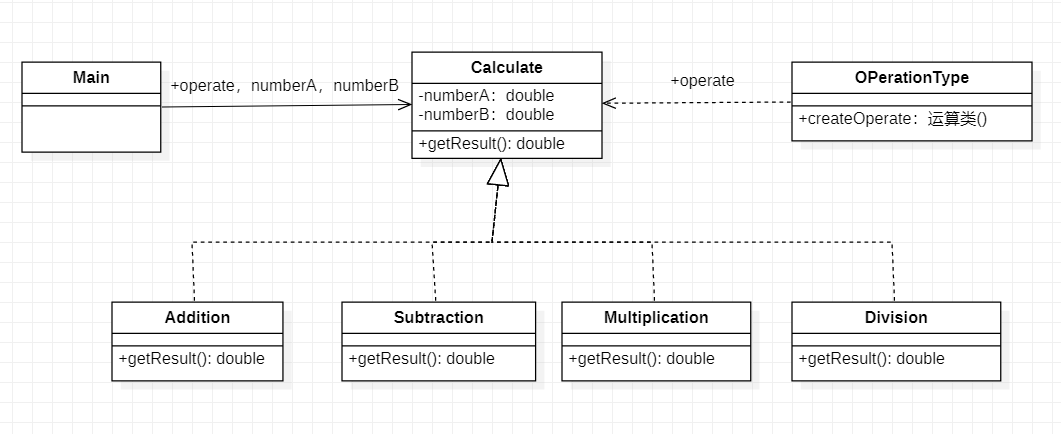
简单工厂模式

1. **定义：**又叫静态工厂方法模式，它定义一个具体的工厂类负责创建一些类的实例。提供一个创建对象实例的功能，而无需关心其具体实现。被创建实例的类型可以使用接口、抽象类或者具体的类。简而言之，就是外界不知道一个工厂里面到底要生成什么样的实例对象，只需要传入一个标识，工厂内部自己就能生产对应的产品，工厂内部是知道具体产品的实现的，最后只需要返回给客户端一个遵循协议的对象即可，客户端拿到这个对象就能直接访问接口协议方法，而不需要关心具体的实现。
2. **结构和使用**

①Factory（Operation Type）：工厂类，简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类的创建产品类的方法可以被外界直接调用，创建所需的产品对象。

②Calculate：抽象产品类，简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

③Method：具体产品类，是简单工厂模式的创建目标。



1. **优缺点：**

①优点**：** 具有封装性，很好地实现了组件的封装，组件外部实现了面向接口编程。解耦性，实现了客户端和具体实现类的解耦合。

②可能会增加客户端的复杂度，因为客户端需要向工厂传递参数，就必得理解所传参数的含义。不方便扩展子工厂。

1. **使用场景：**

①工厂类负责创建的对象比较少：由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。

②客户端只知道传入工厂类的参数，对于如何创建对象不关心：客户端既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数。

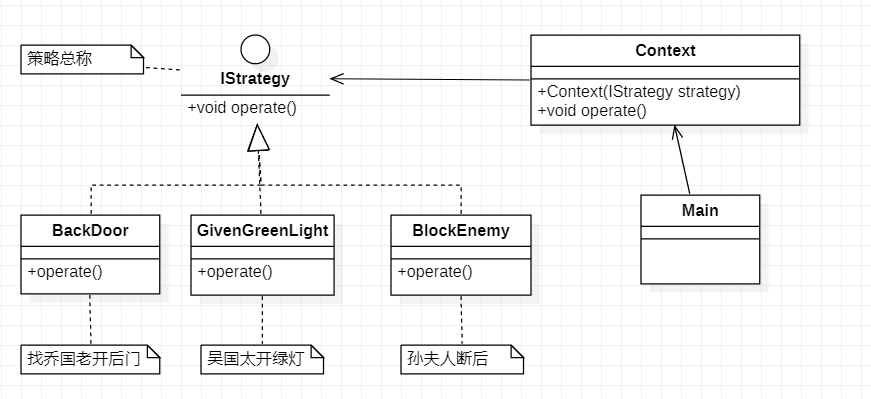
策略模式

1. **定义：**定义一组算法，将每个算法封装起来，并且使他们之间可以互换。策略模式是一个简单的模式也叫做政策模式。
2. **结构和使用**

环境类(Context):用来操作策略的上下文环境，也就是赵云。

抽象策略类(IStrategy):策略的抽象，出行方式的抽象。

具体策略类(Concrete Strategy):具体的策略实现，每一种策略方式的具体实现。



1. **优缺点：**

①优点：1）算法可以自由切换，这是策略模式本身定义的，只要实现抽象策略，他就成为策略家族的一个成员，通过封装角色对其进行封装，保证对外提供“可自由切换”的策略。

2）避免使用多重条件判断，我们之前在选择策略的时候，往往会使用if-else语句，也就是用户不选择A那么就选择B这样的一种情况。这种情况耦合性太高了，而且代码臃肿，有了策略模式我们就可以避免这种现象 。

3）扩展性良好，策略模式遵循开闭原则，实现代码的解耦合。扩展新的方法时也比较方便，只需要继承策略接口就好了。

②缺点：

1. 策略类数量增多，每一个策略都是一个类，复用性小，类数量增多。
2. 所有策略类都需对外暴露，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。策略模式会出现很多的策略类。Context在使用这些策略类的时候，这些策略类由于继承了策略接口，所以有些数据可能用不到，但是依然初始化了。
3. **使用场景：**

①多个类只有在算法或行为上稍有不同的场景。

②算法需要自由切换的场景。

③需要屏蔽算法规则的场景。

代理模式

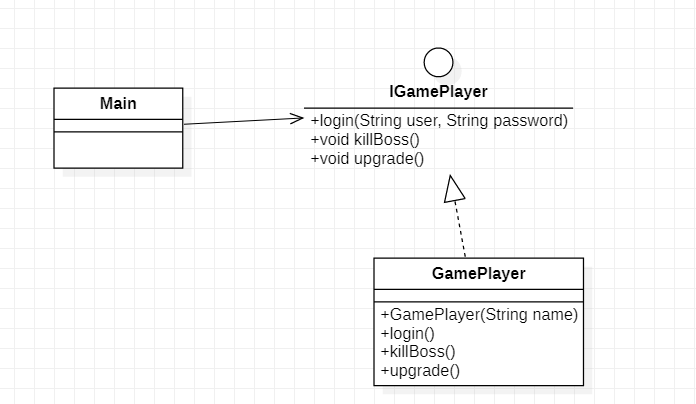
1. **定义：**为其他对象提供一种代理以控制对这个对象的访问。
2. **结构和使用**

Subject: 抽象主题角色 ，可以使抽象类也可以是接口。

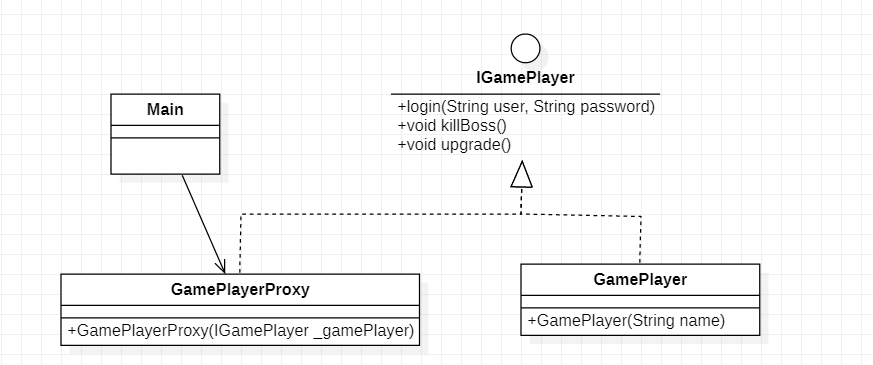
RealSubject：具体主题角色，是业务逻辑的具体执行者。

Proxy：代理主题角色，也叫委托类、代理类。负责对真是角色的应用。

1. 未使用代理模式



1. 使用代理模式



1. **优缺点**

①优点：职责清晰，真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件事务，附带的结果就是编程简洁清晰。

高扩展性，具体的主题角色随时都会发生变化的，只要它实现了接口，不管他如何变化，都离不开接口，那我们的代理类完全就可以在不做任何修改的情况下使用。

装饰模式

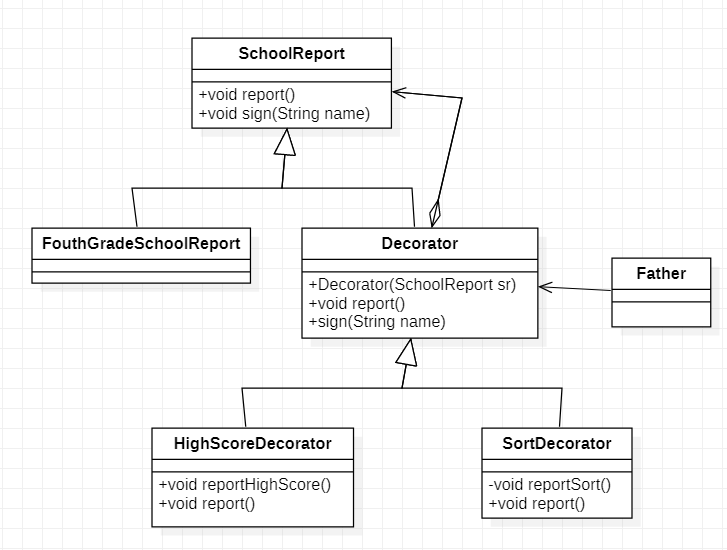
1. **定义：**动态的给对象添加一些额外的职责。装饰模式相比生成子类更为灵活。
2. **结构和使用：**

Component: 抽象构件。

Concrete Component: 具体构件，最核心的，就装饰它。

Decorator: 抽象装饰类。

Concrete Decorator: 具体装饰类，两个具体的装饰类。



1. **优缺点：**

①优点：1)装饰类和被装饰类可以独立发展，而不会互相耦合。

2）装饰模式是继承关系的一个替代方案。

3）装饰模式可以动态的扩展一个实现类的功能。

②缺点：多层的装饰是比较复杂的。

1. **使用场景：**

①需要扩展一个类的功能，或给一个类增加附加功能。

②需要动态的给一个对象增加功能，这些功能可以动态的取消。

③需要为一批的兄弟类进行改装或加装功能，首选装饰模式。

工厂方法模式

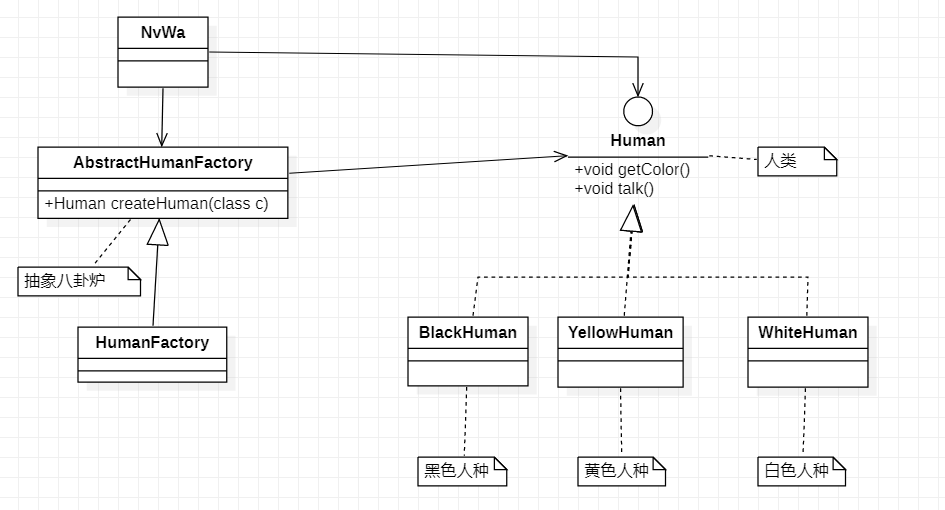
1. **定义：**定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到子类。
2. **结构和使用：**

Product：抽象产品

Concrete Product：具体产品

Factory：抽象工厂

Concrete Factory：具体工厂



1. **优缺点：**

①优点：1）良好的封装性，代码结构清晰。2) 扩展性高，屏蔽了产品类。3）它是典型的解耦框架。

②缺点：在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。

1. **使用场景：**

①需要灵活的、可扩展的框架时使用。②可以使用在测试驱动开发的框架下。

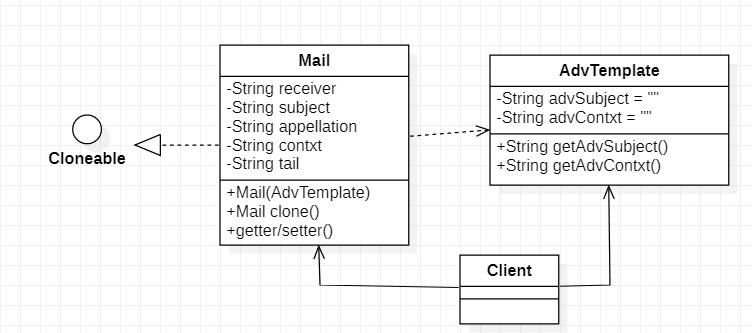
原型模式

1. **定义：**原型模式是一种对象创建型模式，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。原型模式允许一个对象再创建另外一个可定制的对象，无须知道任何创建的细节。
2. **结构和使用：**

Prototype：抽象原型类

Concrete Prototype：具体原型类

Client：客户类



**3、优缺点：**

①优点：1）当创建新的对象实例较为复杂时，使用原型模式可以简化对象的创建过程，通过一个已有实例可以提高新实例的创建效率。

2）可以动态增加或减少产品类。

3）原型模式提供了简化的创建结构。

4）可以使用深克隆的方式保存对象的状态。

②缺点：1）需要为每一个类配备一个克隆方法，而且这个克隆方法需要对类的功能进行通盘考虑，这对全新的类来说不是很难，但对已有的类进行改造时，不一定是件容易的事，必须修改其源代码，违背了“开闭原则”。

1. 在实现深克隆时需要编写较为复杂的代码。

**4、使用场景：**

①资源优化场景。

②性能和安全要求的场景。

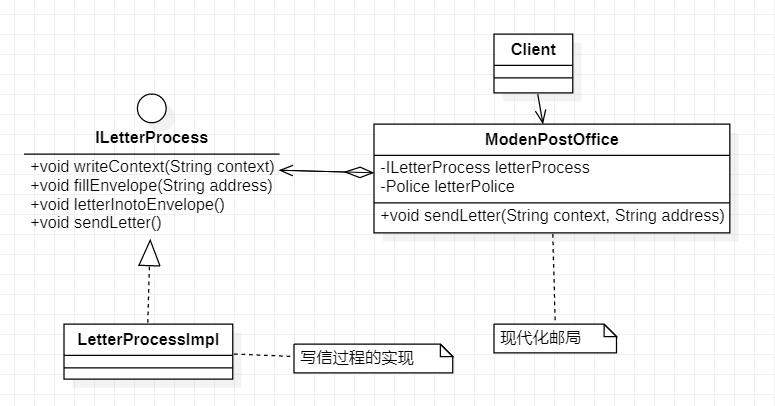
③一个对象多个修改者的场景。

外观模式

1. **定义：**外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统中的一组接口提供一个一致的界面，外观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。外观模式又称为门面模式，它是一种对象结构型模式。
2. **结构和使用：**

Facade: 外观角色

Sub System:子系统角色



1. **优缺点：**

①优点：1）减少系统的相互依赖。2）提高了灵活性。3）提高安全性。

②缺点：1）不能很好地限制客户使用子系统类，如果对客户访问子系统类做太多的限制则减少了可变性和灵活性。

2）在不引入抽象外观类的情况下，增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。

1. **使用场景：**

①当要为一个复杂子系统提供一个简单接口时可以使用外观模式。该接口可以满足大多数用户的需求，而且用户也可以越过外观类直接访问子系统。

②客户程序与多个子系统之间存在很大的依赖性。引入外观类将子系统与客户以及其他子系统解耦，可以提高子系统的独立性和可移植性。

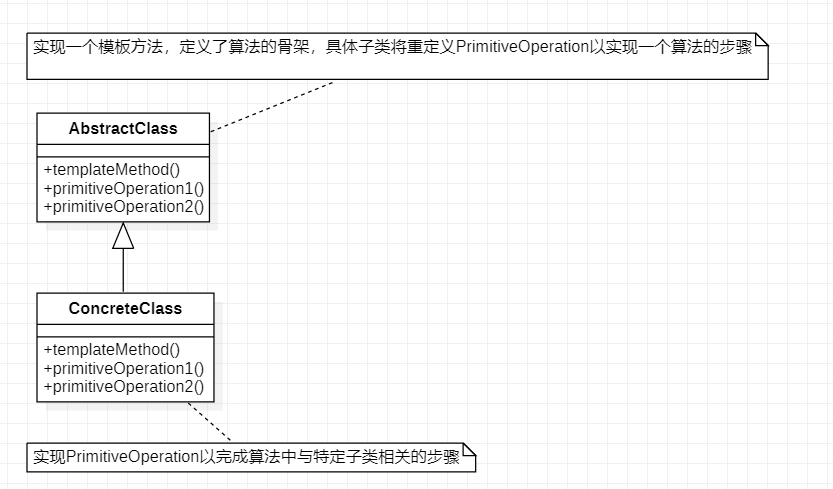
③在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

模板方法模式

1. **定义：**定义一个操作中算法的骨架，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。模板方法是一种类行为型模式。
2. **结构和使用：**

Abstract Class: 抽象类

Concrete Class: 具体子类



1. **优缺点：**

①优点：1）模板方法模式在一个类中形式化地定义算法，而由它的子类实现细节的处理。

2）模板方法模式是一种代码复用的基本技术。

3）模板方法模式导致一种反向的控制结构，通过一个父类调用其子类的操作，通过对子类的扩展增加新的行为，符合“开闭原则”。

②缺点：每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象，但是更加符合“单一职责原则”，使得类的内聚性得以提高。

1. **使用场景：**

①多个子类有公有的方法，并且逻辑基本相同时。

②重要、复杂的算法，可以把核心算法设计为模板方法。

③重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数约束其行为。

建造者模式

**1、定义：将一个复杂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。**

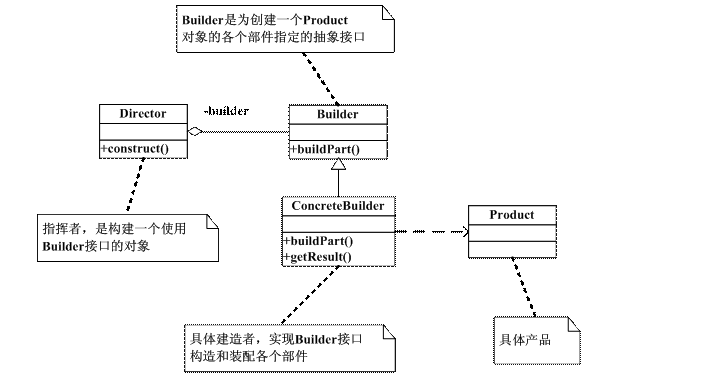
**2、结构和使用：**

Product产品类

Builder抽象建造者

Concrete Builder具体建造者

Director导演类



**3、优缺点：**

①优点：1）封装性，使用建造者模式，可以使客户端不必知道产品内部组成的细节。

2）建造者独立，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。

**4、使用场景：**

①相同的方法，不同的执行顺序，产生不同的事件，结果是可以采用建造者模式。

②多个部件或零件都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用建造者模式。

观察者模式

**1、定义：**观察者模式也叫做发布订阅模式，定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于他的对象都会得到通知，并被自动更新。

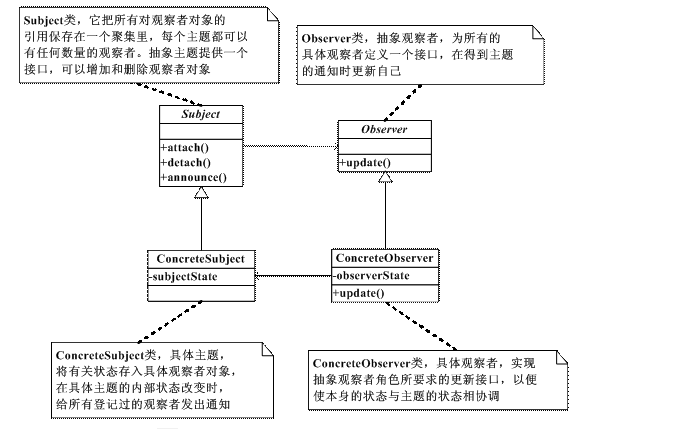
**2、结构和使用：**

Subject被观察者

Observer观察者

Concrete Subject具体的被观察者

Concrete Observer具体的观察者



**3、优缺点：**

①优点：1) 观察者网观察者和被观察者之间是抽象耦合。2）建立一套触发机制。

②缺点：观察这种模式需要考虑一下开发效率和运行效率问题，一个被观察者多个观察者开发和调试就会比较复杂，而且在Java中消息的通知默认是顺序执行，一个观察着卡壳会影响整体的执行效率，在这种情况下，一般考虑采用异步的方式。

**4、使用场景：**

**①**关联行为场景。

②事件多及触发场景。

③跨系统的消息交换场景，如消息队列的处理机制。