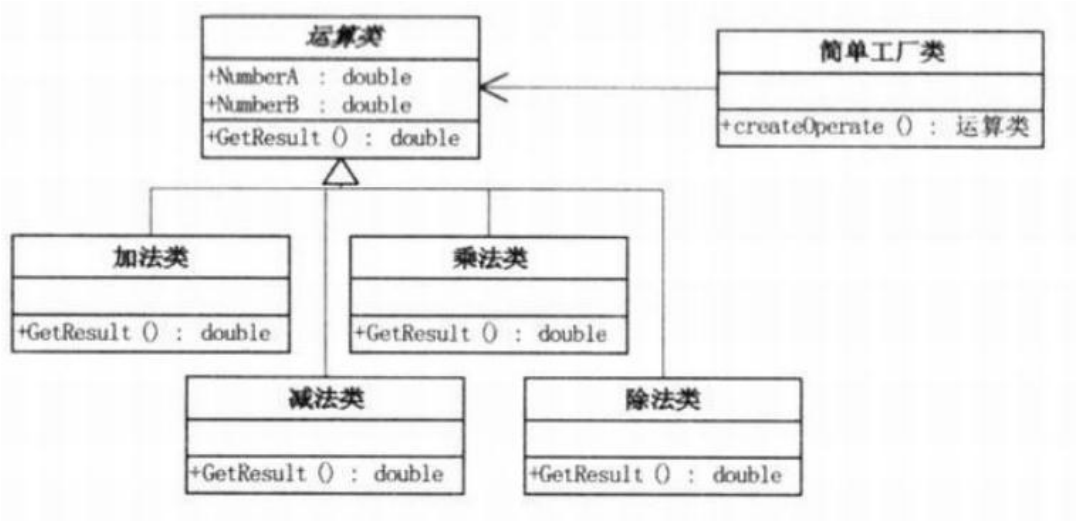
一、创建型模式

工厂方法



工厂模式概念：

实例化对象，用工厂方法代替new操作。

工厂模式包括工厂方法模式和抽象工厂模式。

抽象工厂模式是工厂方法模式的扩展。

什么情况下适合工厂模式？

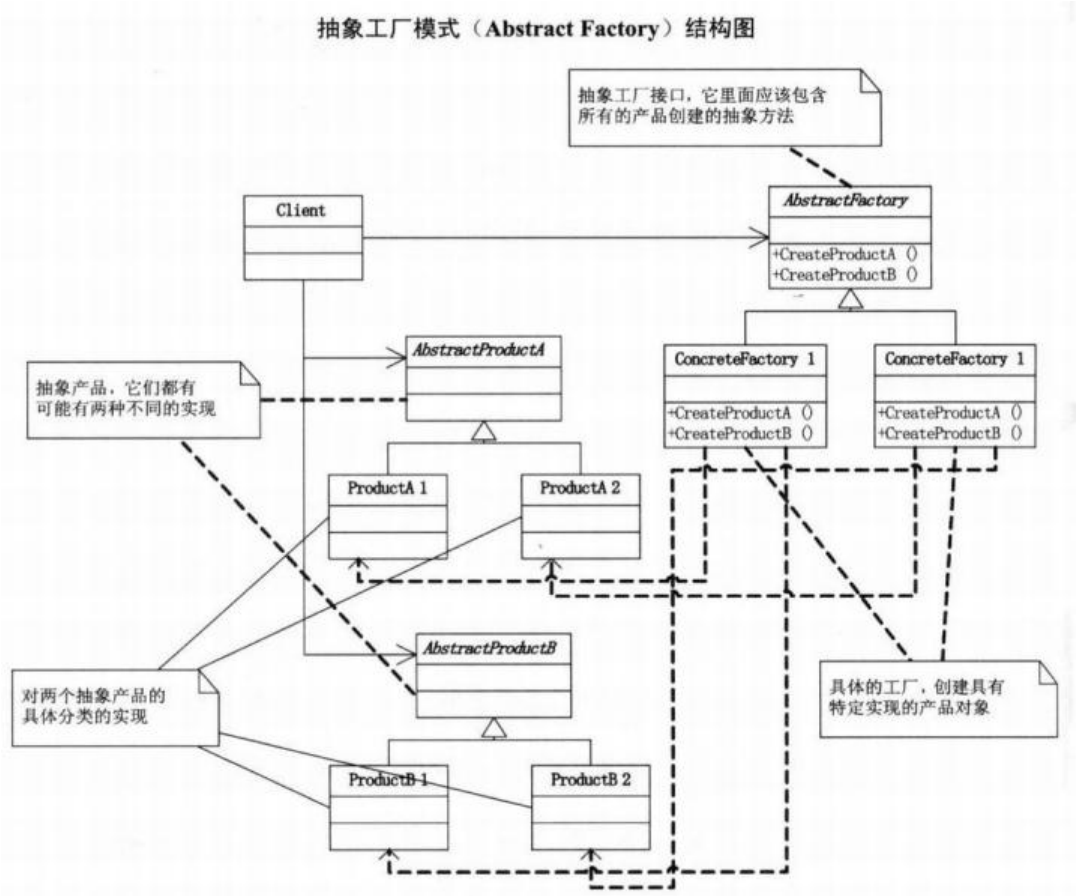
- 有一组类似的对象需要创建。
- 在编码时不能预见需要创建哪种类的实例。
- 系统需要考虑扩展性，不应依赖于产品类实例如何被创建、组合和表达的细节。

```
/**
 * 根据类的名称来生产对象
 * @param className
 * @return
 */
public HairInterface getHairByClassKey(String key){

    try {
        Map<String, String> map = new PropertiesReader().getProperties();

        HairInterface hair = (HairInterface) Class.forName(map.get(key)).newInsta
        return hair;
    } catch (InstantiationException e) {
```

抽像工厂



抽象工厂： 存在不同实现的接口，则以不同接口的集合为一个工厂(即由不同的实现接口组成了一个产品系列)，由抽像接口实例出具体的工厂来。
但抽象工厂比简单工厂多了抽像工厂接口及具体实现的工厂(不同于简单工厂只用一个)，所以如果用简单工厂实现，则是一个具体工厂的静态方法，实例出不同的对象出来。

工厂方法模式和抽象工厂模式对比

- **工厂模式是一种极端情况的抽象工厂模式，而抽象工厂模式可以看成是工厂模式的推广**
- **工厂模式用来创建一个产品的等级结构，而抽象工厂模式是用来创建多个产品的等级结构**
- **工厂模式只有一个抽象产品类，而抽象工厂模式有多个抽象产品类**

工厂模式的实现帮助我们

- 系统可以在不修改具体工厂角色的情况下引进新的产品
- 客户端不必关心对象如果创建，明确了职责
- 更好的理解面向对象的原则 面向接口编程，而不要面向实现编程

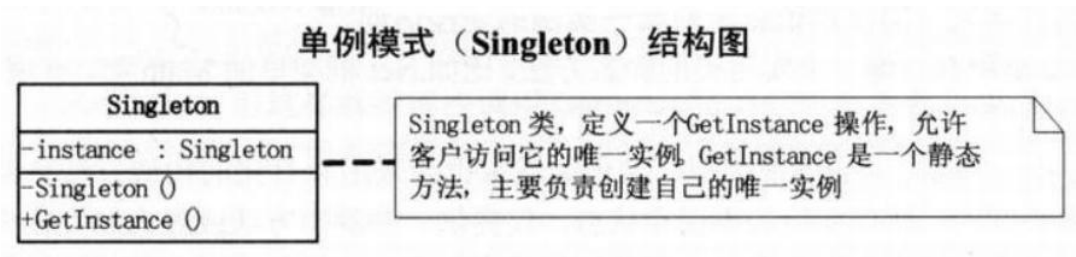
工厂模式适用与哪些场景

- 一个系统应当不依赖于产品类实例被创立，组成，和表示的细节。这对于所有形态的工厂模式都是重要的
- 这个系统的产品有至少一个的产品族
- 同属于同一个产品族的产品是设计成在一起使用的。这一约束必须得在系统的设计中体现出来
- 不同的产品以一系列的接口的面貌出现，从而使系统不依赖于接口实现的细节

建造者

原型

单例



单例模式

懒汉模式　　饿汉模式

饿汉模式

```
public class Singleton {
    //1.将构造方法私有化，不允许外部直接创建对象
    private Singleton(){
    }

    //2.创建类的唯一实例，使用private static修饰
    private static Singleton instance=new Singleton();

    //3.提供一个用于获取实例的方法，使用public static修饰
    public static Singleton getInstance(){
        return instance;
    }
}
```

懒汉模式

```java
public class Singleton2 {
    //1.将构造方式私有化，不允许外边直接创建对象
    private Singleton2(){
    }

    //2.声明类的唯一实例，使用private static修饰
    private static Singleton2 instance;

    //3.提供一个用于获取实例的方法，使用public static修饰
    public static Singleton2 getInstance(){
        if(instance==null){
            instance=new Singleton2();
        }
        return instance;
```

```java
/*
 * 懒汉模式
 * 区别：饿汉模式的特点是加载类时比较慢，但运行时获取对象的速度比较快，线程安全
 *      懒汉模式的特点是加载类时比较快，但运行时获取对象的速度比较慢，线程不安全
 */
```

二、结构型模式

适配器

装饰

桥接

组合

享元

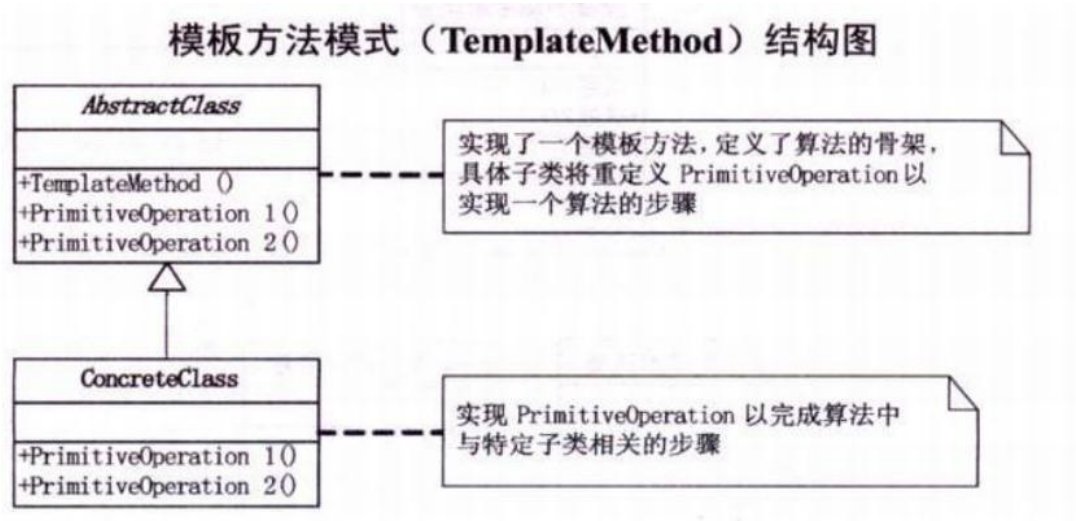
代理

外观

三、行为型模式

观察者

模板方法



模板方法模式的实现要素

准备一个抽象类，将部分逻辑以具体方法的形式实现，然后声明一些抽象方法交由子类实现剩余逻辑，用钩子方法给予子类更大的灵活性。最后将方法汇总构成一个不可改变的模板方法。

模板方法模式的适用场景

(1)算法或操作遵循相似的逻辑

```
/*
 * 具体子类，提供了制备茶的具体实现
 */
public class Tea extends RefreshBeverage {

    protected void brew() {

    protected void addCondiments() {

    @Override
    /*
     * 子类通过覆盖的形式选择挂载钩子函数
     * @see com.imooc.pattern.template.RefreshBeverage#is
     */
    protected boolean isCustomerWantsCondiments(){
        return false;
    }

}
```

```
/*
 * 具体子类，提供了咖啡制备的具体实现
 */
public class Coffee extends RefreshBeverage {

    @Override
    protected void brew() {
        System.out.println("用沸水冲泡咖啡");

    }

    @Override
    protected void addCondiments() {
        System.out.println("加入糖和牛奶");
    }

}
```

(2)重构时(把相同的代码抽取到父类中)

(3)重要、复杂的算法，核心算法设计为模板算法

缺点 是类常已被继承，如果像JAVA单继承会行不通

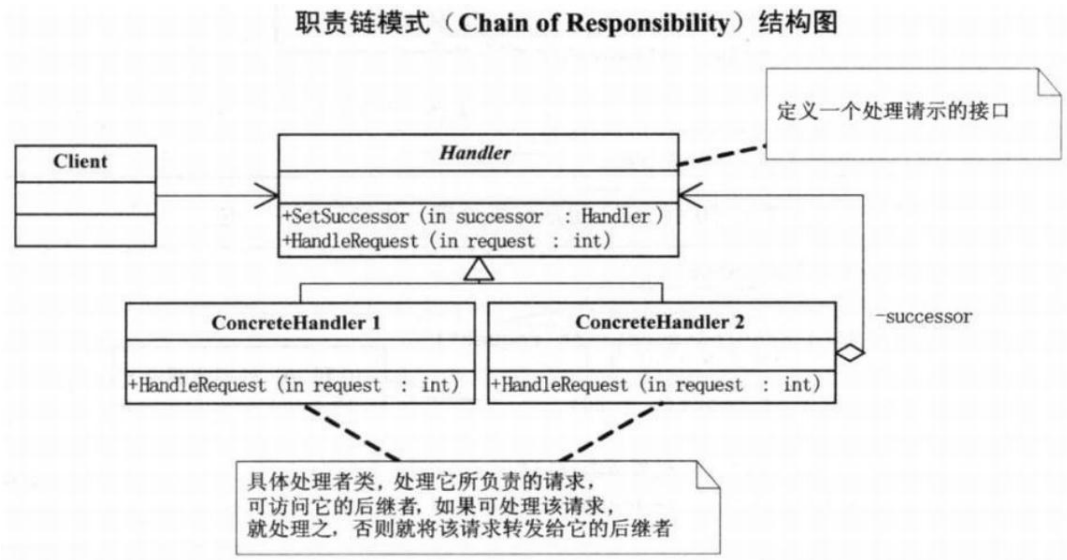
模板方法模式的缺点

继承

命令

状态

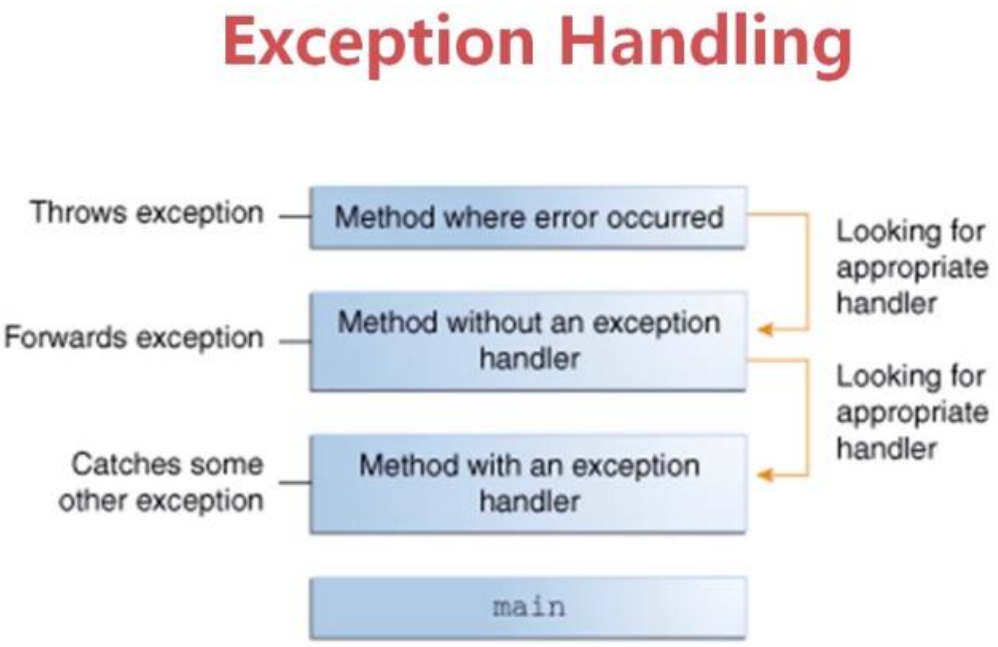
职责链

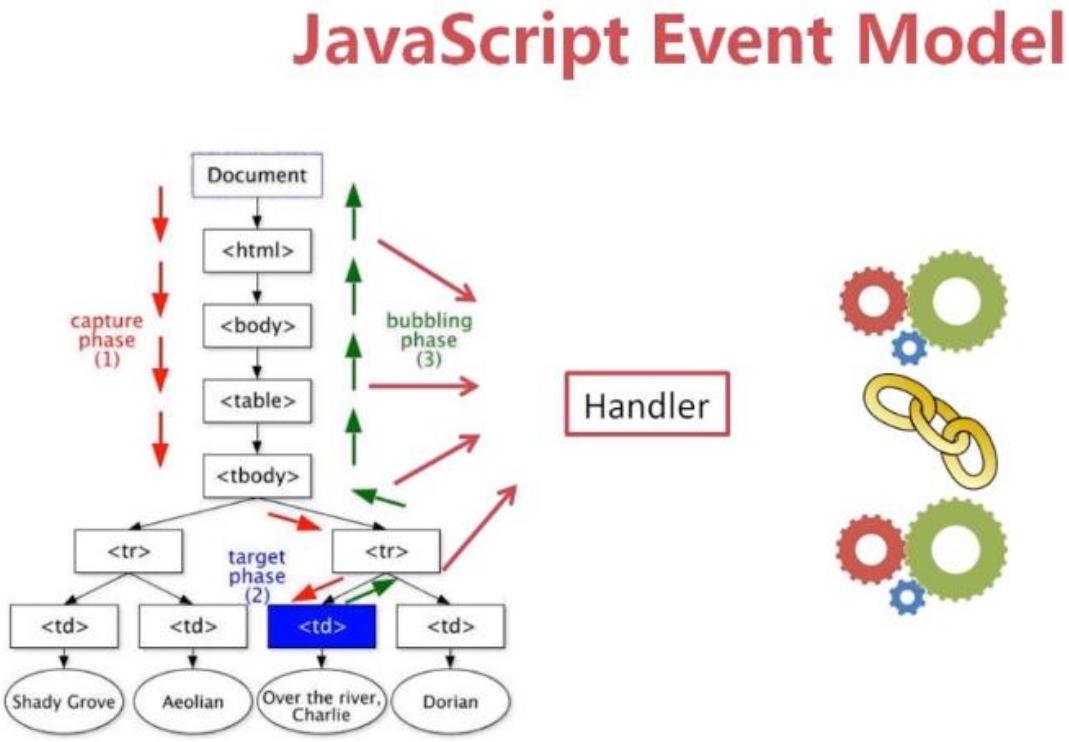


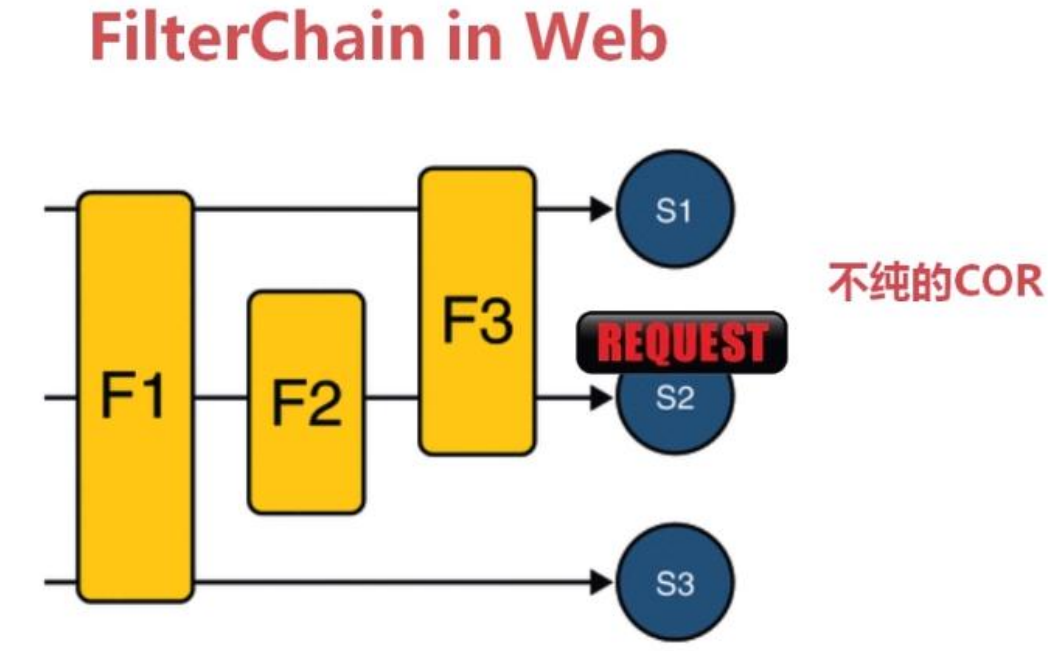
将接收者对象连成一条链，并在该链上传递请求，直到有一个接收者对象处理它。通过让更多对象有机会处理请求， 避免了请求发送者和接收者之间的耦合。

在责任链模式中，作为请求接收者的多个对象通过对其后继的引用而连接起来形成一条链。请求在这条链上传递，直到链上某一个接收者处理这个请求。每个接收者都可以选择自行处理请求或是向后继传递请求。

发出请求的客户端并不知道链上的哪一个接收者会处理这个请求，从而实现了客户端和接收者这件的解耦。







职责链的缺点： 时间复杂度高，内存开销大

解释器

中介者

访问者

策略

备忘录

迭代器