OOP思想回顾

OOP中的内容

- 类 ——》对象
- 接口
- 抽象类

抽象类是做什么用的?

- 包容不变与变的

类是做什么的?

- 模拟现实，封装数据与代码

接口是做什么用的？（包含interface，抽象类等）

- 类之间的交互规范

- 定义功能使用者和功能提供者间的接口
- 为什么要有接口？隔离变化

OOP三大特性

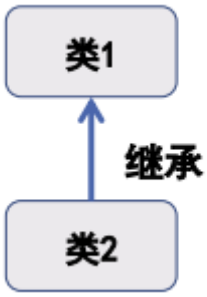
OO的三大特性

- 封装
- 继承
- 多态
    - 多态为我们提供了什么?
        - 一种实现变化的方式

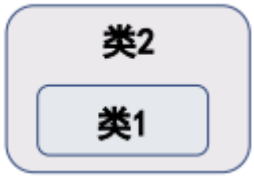
OOP中复用的形式

类与类之间的关系有哪些?

- 继承



- 组合



组合，类2持有类1的实例

类2使用类1

为什么要用设计模式?

天天加班赶项目，开发项目具体都做的是什么?

编写代码，写接口、写类、写方法

用设计模式或做设计的作用是什么?

指导、规定该如何撸代码，如何来写接口、写类、写方法

为什么要做设计、用设计模式?

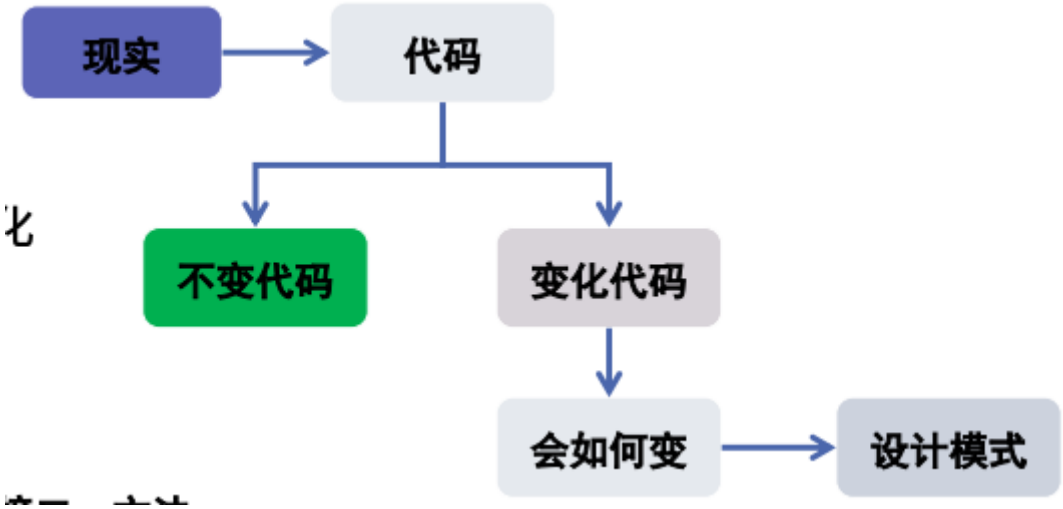
代码会变，为应对变化，为了以后方便扩展。

做到以不变应万变，做一个会偷懒的程序员!

不变的是变化！软件界永恒的真理

如何着手使用设计模式

- 理清现实
- 区分变与不变
- 搞清楚会如何变
- 使用者如何隔绝这种变化



设计的体现：如何来定义类、接口、方法

不同的变化对应不同的设计模式

- 着手：找出变化，分开变化和不变的

  隔离，封装变化的部分，让其它部分不受它的影响

面向对象7大设计原则

- 单一职责原则 SRP

  一个类只有一个引起修改变化的原因，也就是只负责一个职责。核心思想：高内聚，低耦合

- 开闭原则 OCP

  对模块类的操作，对扩展开放，对修改关闭。经过多年验证的代码，改变后容易出现bug

- 里氏替换原则 LSP

  超类可以被替换为其子类来使用。这样就能做到代码不用修改，能够任意切换不同的实现子类，少修改复用。

- 依赖倒置原则DIP

  程序依赖于抽象接口，不依赖于具体实现，降低实现模块间的耦合度。你不要来找（new）我，我会来找（set）你。

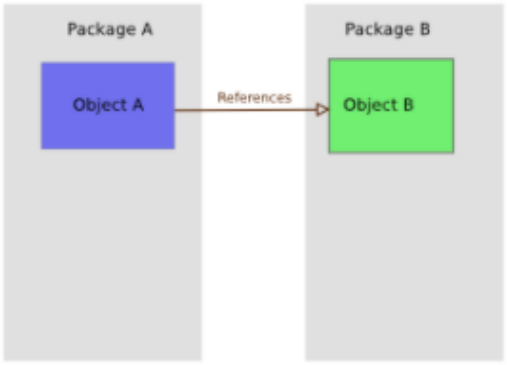


Figure 1

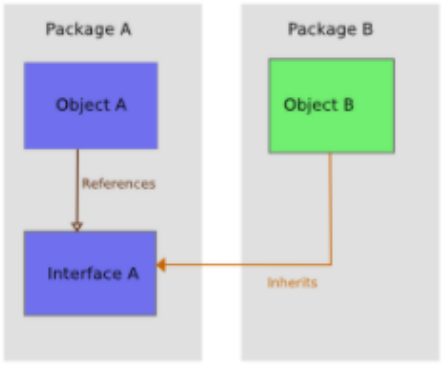


Figure 2

该原则规定：

  - 高层次的模块不应该依赖于低层次的模块，两者都应该依赖于抽象接口
  - 抽象接口不应该依赖于具体实现。具体实现应该依赖于抽象接口

    例如：台灯和按钮的例子https://flylib.com/books/en/4.444.1.71/1/

    可以通过按钮控制台灯的开关。

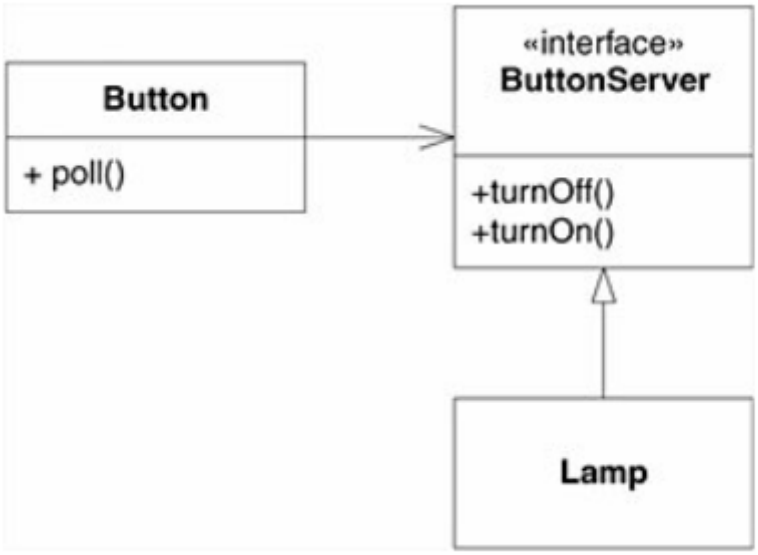
- 传统写法：

代码如下：

```
public class Button {   private Lamp lamp;   public void Poll()   {
if (/*some condition*/)       lamp.TurnOn();   } }
```

问题：高层依赖于底层实现。Button直接控制Lamp，且只能控制Lamp。

- 依赖倒置的做法



Button和Lamp都依赖于抽象层ButtonServer。

- 接口隔离原则ISP

  不该强迫客户程序依赖不需要使用的方法，一个接口只提供对外的功能，不是把所有功能封装进去减少依赖范围

- 组合复用原则CRP

  尽量使用组合，而不是继承来达到复用的目的，继承强耦合，组合低耦合，组合还能运行时动态替换

- 迪米特法则 LOD

  一个对象应当对其他对象有尽可能少的了解，不和陌生人说话，降低各个对象之间的耦合，提高系统的可维护性性

设计模式

再次强调：应用设计模式的目的

- 易扩展，易维护
- 少改代码，不改代码

策略模式

示例：

京东、天猫双十一促销，各种商品有多种不同的促销活动：

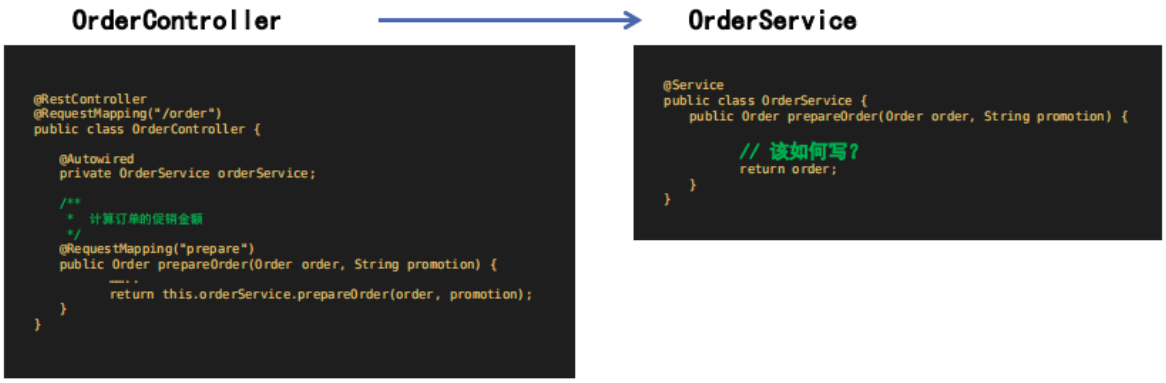
- 满减：满400减50
- 每满减：每满100减20
- 数量折扣：买两件8折、三件7折
- 数量减：满三件减最低价的一件
- 。。。

顾客下单时可选择多种其中一种来下单

后端代码中如何来灵活应对订单金额的计算?

以后换会有很多的促销活动出现!

该如何来实现订单金额的计算?



这样写可以吗?

```
@Service
public class OrderService {
    public Order prepareOrder(Order order, String promotion) {
            switch (promotion) {
            case "promotion-1":
                    // 促销1的算法
                    ...
                    break;
            case "promotion-2":
                    // 促销2的算法
                    ...
                    break;
            case "promotion-3":
                    // 促销3的算法
                    ...
                    break;
            ...
            }
            return order;
    }
}
```

营销活动会有很多，这个switch会不会很庞大？生怕出错！！！

改写一下，这样会好很多

```
@Service
public class OrderService {
    public Order prepareOrder(Order order, String promotion) {
            switch (promotion) {
            case "promotion-1":
                    // 促销1的算法
                    return calPromotion1(order);
            case "promotion-2":
                    // 促销2的算法
                    return calPromotion2(order);
            case "promotion-3":
                    // 促销3的算法
                    return calPromotion3(order);
            ...
            }
            return order;
    }

    private Order calPromotion1(Order order) {
            System.out.println("促销1计
算.............................");
            return order;
    }
    ....
}
```

方法设计原则：单一职责原则

营销活动经常变，这个switch就得经常改，还得不断加促销的算法方法。。。

改代码是bug的源泉

我们希望少改动OrderService

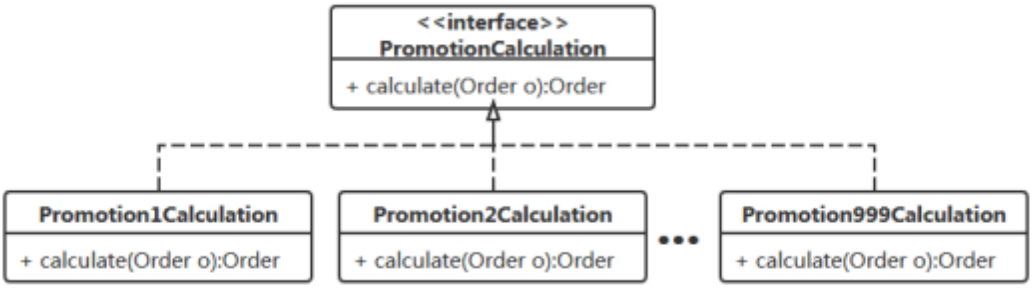
分析：这里变的是什么?

促销的金额的算法！同一行为的不同算法！我们不希望OrderService被算法代码爆炸！

再次改进

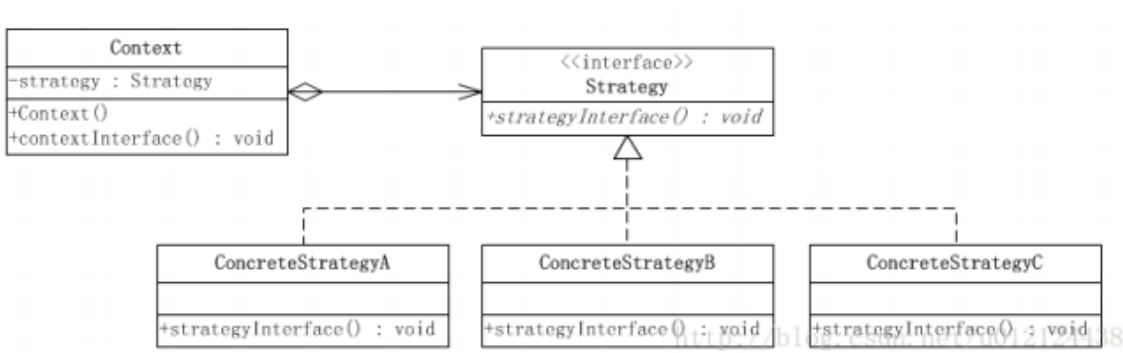
同一行为的不同算法实现，我们可以用接口来定义行为，不同的算法分别去实现接口

设计原则：对修改关闭，对扩展开发



这就是策略模式的应用

定义：策略模式定义了一系列的算法，并将每一个算法封装起来，而且使得他们可以相互替换，让算法独立于使用它的用户而独立变化。



改进后的OrderService

```
@Service
public class OrderService {
    public Order prepareOrder(Order order, String promotion) {
            switch (promotion) {
            case "promotion-1":
                  // 促销1的算法
                  return new Promotion1Calculation().calculate(order);
            case "promotion-2":
                  // 促销2的算法
                  return new Promotion2Calculation().calculate(order);
            case "promotion-3":
                  // 促销3的算法
                  return new Promotion3Calculation().calculate(order);
            ......
            }
    }
}
```

但是switch中的代码还是会不断的变！！！switch中需要知道所有的实现！如何让OrderService的代码不要改变？把变的部分移出去！如何移？

工厂模式

通过一个工厂来专门负责创建各种促销计算实现，就把变化移出来了！

```
@Component
public class PromotionCalculationFactory {

    public PromotionCalculation getPromotionCalculation(String promotion) {
          switch (promotion) {
          case "promotion-1":
                // 促销1的算法
                return new Promotion1Calculation();
          case "promotion-2":
                // 促销2的算法
                return new Promotion2Calculation();
          case "promotion-3":
                // 促销3的算法
                return new Promotion3Calculation();
          ......
          }
    }
}
```

```
@Service
public class OrderService {
   @Autowired
   private PromotionCalculationFactory promotionCalculationFactory;

   public Order prepareOrder(Order order, String promotion) {
         return promotionCalculationFactory
               .getPromotionCalculation(promotion).calculate(order);
   }
}
```

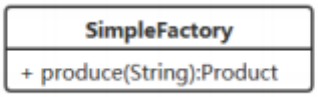
想要工厂中的代码也不要随促销的变化而变化，该如何处理?

- 方式一：promotion=spring beanName
- 方式二：配置promotion与实现类的对应关系

工厂模式

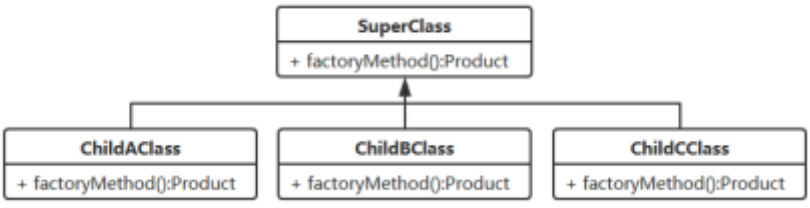
- 简单工厂模式

  一个工厂负责创建所有的实例



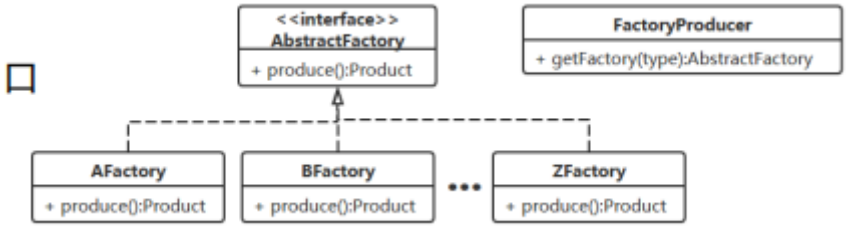
- 工厂方法模式

  父类中定义工厂方法，各子类实现具体的实例创建



- 抽象工厂模式

  定义一个工厂接口，所有具体工厂实现接口



装饰者模式

示例：促销活动可多重叠加，该如何灵活实现订单金额计算?

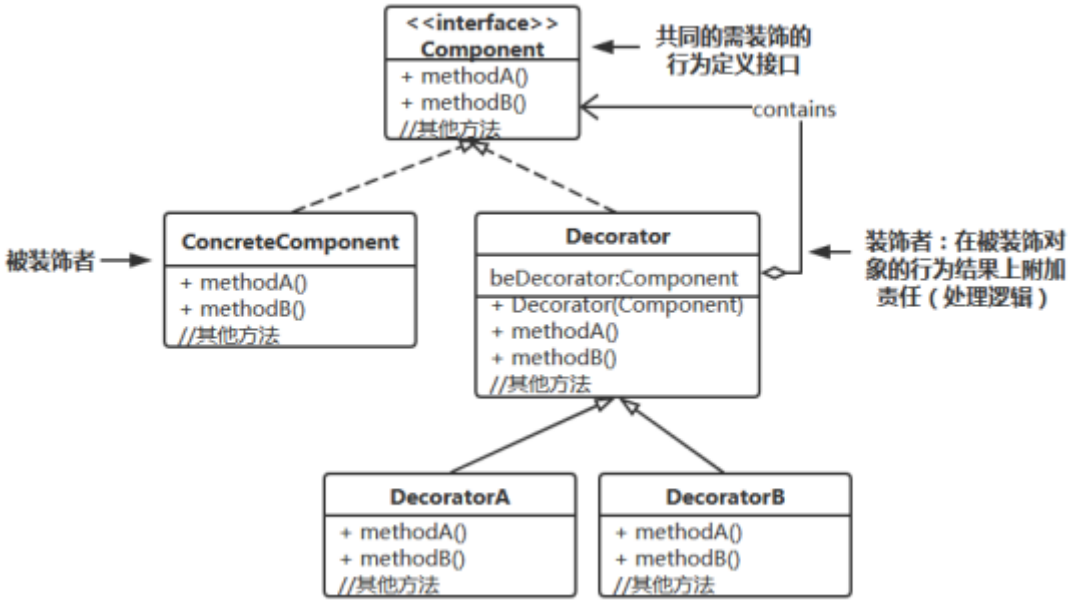
OrderController → OrderService

```java
@RestController
@RequestMapping("/order")
public class OrderController {

    @Autowired
    private OrderService orderService;

    /**
     * 计算订单的促销金额，促销按给入的顺从叠加
     */
    @RequestMapping("prepare")
    public Order prepareOrder(Order order, String... promotion) {
        return this.orderService.prepareOrder(order, promotion);
    }
}
```

```java
@Service
public class OrderService {
    @Autowired
    private PromotionCalculationFactory promotionCalculationFactory;

    public Order prepareOrder(Order order, String... promotion) {

        // 该如何写
    }
}
```

```java
public Order prepareOrder(Order order, String... promotion) {

    for (String p : promotion) {
        order = promotionCalculationFactory.
        getPromotionCalculation(p).calculate(order);
    }
    return order;
}
```

定义：以装饰的方式，动态地将责任附加到对象上



不改变具体类代码，动态叠加增强行为功能。

若要扩展功能，装饰者提供了比继承更弹性的替代方案

相较于前面的for循环，有何区别?

考虑：当需要对一个类的多个方法进行增强，使用者随意使用被增强时，for循环就不够灵活了

责任链和装饰者模式完成的是相同的事情

代码示例

```
public interface Component {
    String methodA();
    int methodB();
}
```

```
public class ConcreteComponent implements Component {
    public String methodA() {
           return "concrete-object";
    }

    public int methodB() {
           return 100;
    }
}
```

```
public class DecoratorSample {
    public static void main(String[] args) {
           Component cc = new ConcreteComponent();
           cc = new DecoratorA(cc);
           System.out.println(cc.methodA());
           System.out.println(cc.methodB());
    }
}
```

```
concrete-object + A
110}
```

```
public class Decorator implements Component {
   protected Component component;

   public Decorator(Component component) {
          super();
          this.component = component;
   }
   public String methodA() {
          return this.component.methodA();
   }
   public int methodB() {
          return this.component.methodB();
   }
}
```

```
public class DecoratorA extends Decorator {

   public DecoratorA(Component component) {
          super(component);
   }
   public String methodA() {
          return this.component.methodA() + " + A";
   }
   public int methodB() {
          return this.component.methodB() + 10;
   }
}
```

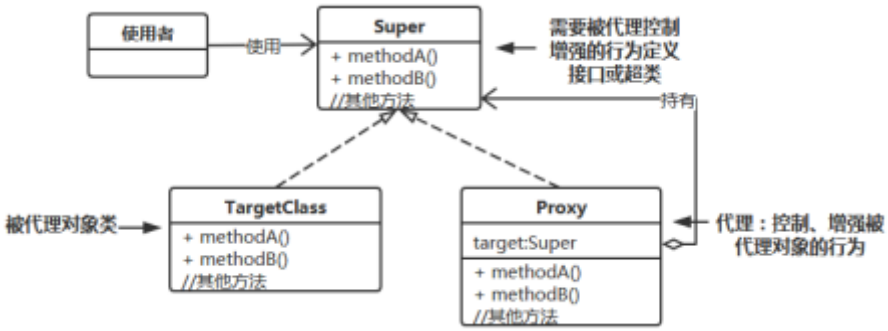
代理模式

定义：为其他对象提供一种代理以控制这个对象的访问

有些情况下，一个对象不适合或者不能直接引用另外对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

作用：不改变原类的代码，而增强原类对象的功能可选择前置、后置、环绕、异常处理增强

类图



类图与装饰者模式一样

与装饰者模式的区别

意图的不同：代理模式意在在代理中控制使用者对目标对象的访问，以及进行功能增强。

代理模式-实现方式

静态代理

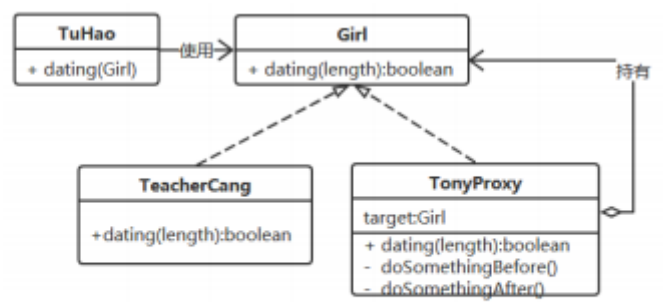
由程序员创建或由特定工具自动生成代理类源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。

动态代理

代理类在程序运行时，运用反射机制动态创建而成。

静态代理事先要知道代理的是什么，而动态代理不知道要代理什么东西，只有在运行时才知道。

代理模式-静态代理示例




```
public interface Girl {
   boolean dating(float length);
}
```

```
public class TeacherCang implements Girl {
   public boolean dating(float length) {
         if (length >= 1.7F) {
               System.out.println("身高可以，可以约！");
               return true;
         }
         System.out.println("身高不可以，不可约！");
         return false;
   }
}
```

```
老板，这个我试过了，很不错，推荐给你！
身高可以，可以约！
老板，你觉得怎样，欢迎下次再约！
```

```
public class Tony implements Girl {
   private Girl girl;
   public Girl getGirl() {
         return girl;
   }
   public void setGirl(Girl girl) {
         this.girl = girl;
   }
   public boolean dating(float length) {
         // 前置增强
         doSomethingBefore();
         boolean res = this.girl.dating(length);
         // 后置增强
         doSomethingAfter();
         return res;
   }
   private void doSomethingBefore() {
         System.out.println("老板，这个我试过了，很不错，推荐给你！");
   }
   private void doSomethingAfter() {
         System.out.println("老板，你觉得怎样，欢迎下次再约！");
   }
}
```

```
public class TuHao {
   private float length;
   public TuHao(float length) {
         this.length = length;
   }
   public void dating(Girl g) {
         g.dating(length);
   }
}
```

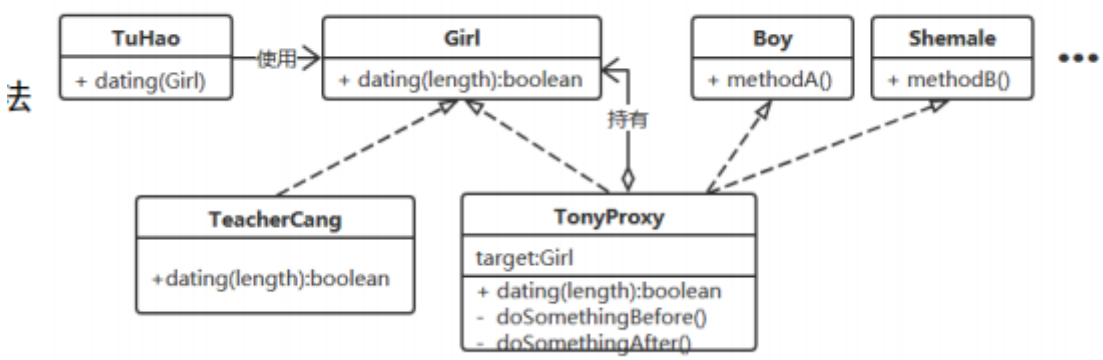
```
public class PlayGame {
   public static void main(String[] args)
{
         TuHao th = new TuHao(1.7F);
         Girl tc = new TeacherCang();
         Tony tony = new Tony();
         tony.setGirl(tc);
         th.dating(tony);
   }
}
```

代理模式-静态代理的缺点

扩展能力差

- 横向扩展：代理更多的类
- 纵向扩展：增强更多的方法

可维护性差



代理模式-动态代理

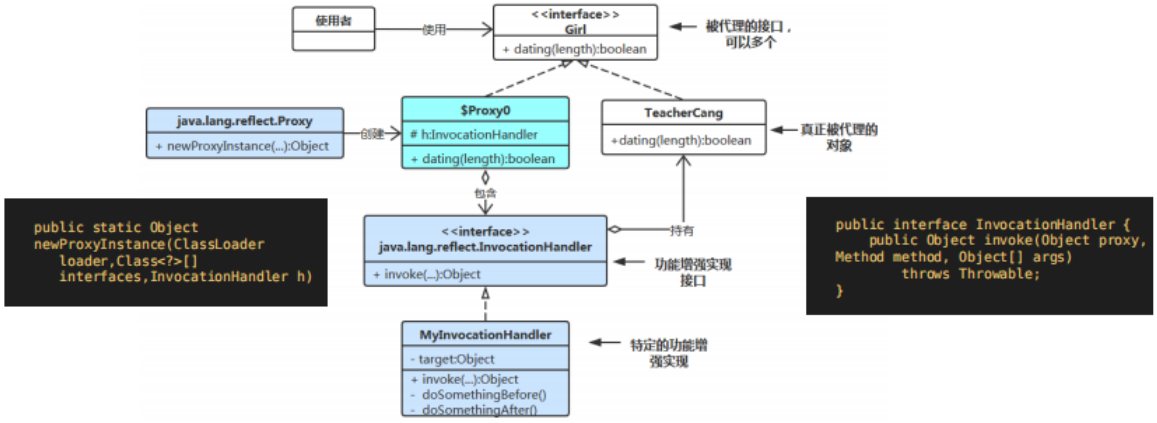
在运行时，动态为不同类的对象创建代理，增强功能。灵活扩展，易维护！

实现方式：

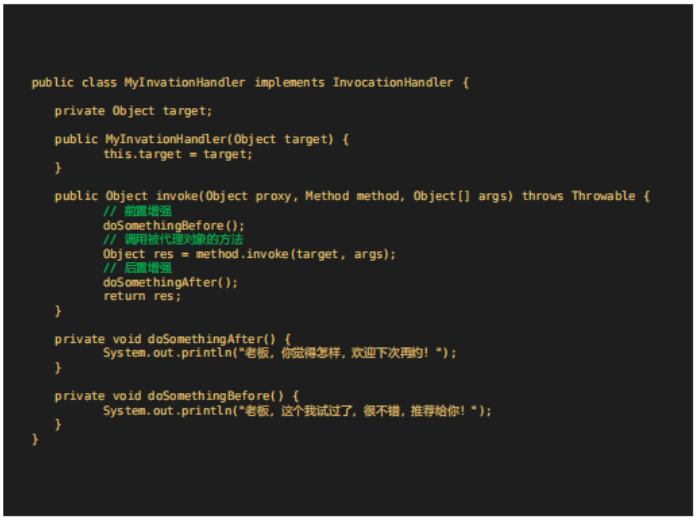
- JDK动态代理：只可对接口创建代理
- CGLIB动态代理：可对接口、类创建代理

代理模式-JDK动态代理

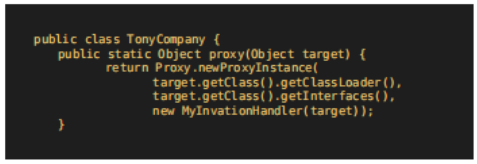
在运行时，对接口创建代理对象



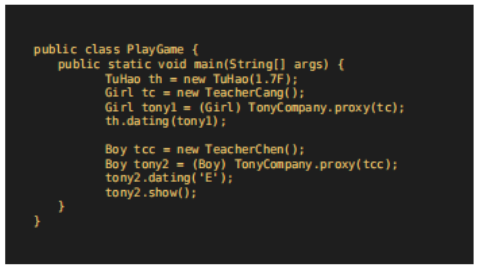
代理模式-JDK动态代理-代码示例



```
public class MyInvationHandler implements InvocationHandler {

    private Object target;

    public MyInvationHandler(Object target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 前置增强
        doSomethingBefore();
        // 调用被代理对象的方法
        Object res = method.invoke(target, args);
        // 后置增强
        doSomethingAfter();
        return res;
    }

    private void doSomethingAfter() {
        System.out.println("老板，你觉得怎样，欢迎下次再约！");
    }

    private void doSomethingBefore() {
        System.out.println("老板，这个我试过了，很不错，推荐给你！");
    }
}
```



```
public class TonyCompany {
    public static Object proxy(Object target) {
        return Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                new MyInvationHandler(target));
    }
```



```
public class PlayGame {
    public static void main(String[] args) {
        TuHao th = new TuHao(1.7F);
        Girl tc = new TeacherCang();
        Girl tony1 = (Girl) TonyCompany.proxy(tc);
        th.dating(tony1);

        Boy tcc = new TeacherChen();
        Boy tony2 = (Boy) TonyCompany.proxy(tcc);
        tony2.dating('E');
        tony2.show();
    }
}
```

通过jdk动态代理生成大的类如下

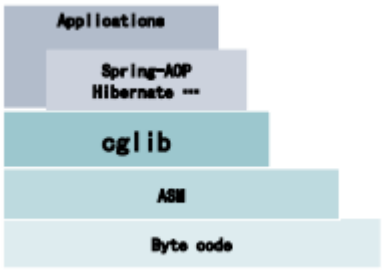
```
public final class $Proxy1 extends Proxy implements Boy {
    private static Method m1;
    private static Method m3;
    private static Method m2;
    private static Method m4;
    private static Method m0;

    public $Proxy1(InvocationHandler var1) throws  {
        super(var1);
    }

    public final boolean equals(Object var1) throws  {
        try {
            return (Boolean)super.h.invoke(this, m1, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
            throw new UndeclaredThrowableException(var4);
        }
    }

    public final void show() throws  {
        try {
            super.h.invoke(this, m3, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final String toString() throws  {
        try {
            return (String)super.h.invoke(this, m2, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
```

```
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final boolean dating(char var1) throws  {
        try {
            return (Boolean)super.h.invoke(this, m4, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
            throw new UndeclaredThrowableException(var4);
        }
    }

    public final int hashCode() throws  {
        try {
            return (Integer)super.h.invoke(this, m0, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    static {
        try {
            m1 = Class.forName("java.lang.Object").getMethod("equals",
Class.forName("java.lang.Object"));
            m3 =
Class.forName("com.example.designpattern.sample.proxy.Boy").getMethod("show");
            m2 = Class.forName("java.lang.Object").getMethod("toString");
            m4 =
Class.forName("com.example.designpattern.sample.proxy.Boy").getMethod("dating",
Character.TYPE);
            m0 = Class.forName("java.lang.Object").getMethod("hashCode");
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```

代理模式-cglib代理模式

cglib是什么?

一个高层次的java字节码生成和转换的api库



ASM:一个低层次的字节码操作库

它的用途

在运行期为类、接口生成动态代理对象。以达到不改动原类代码而实现功能增强的目的

常使用在哪里?

常在AOP、test、orm框架中用来生成动态代理对象、拦截属性访问

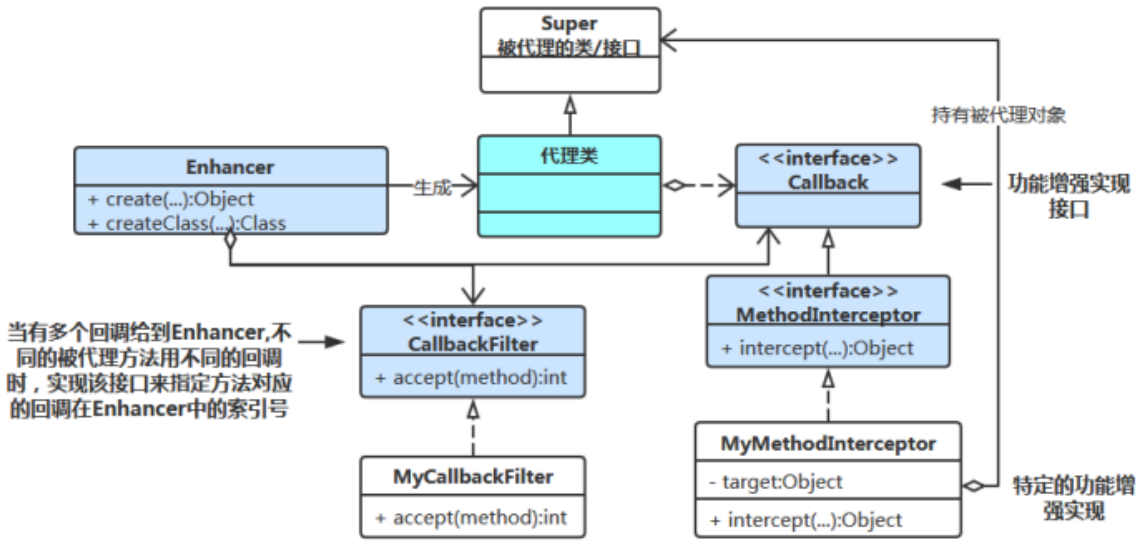
https://github.com/cglib/cglib/wiki

如何使用它?

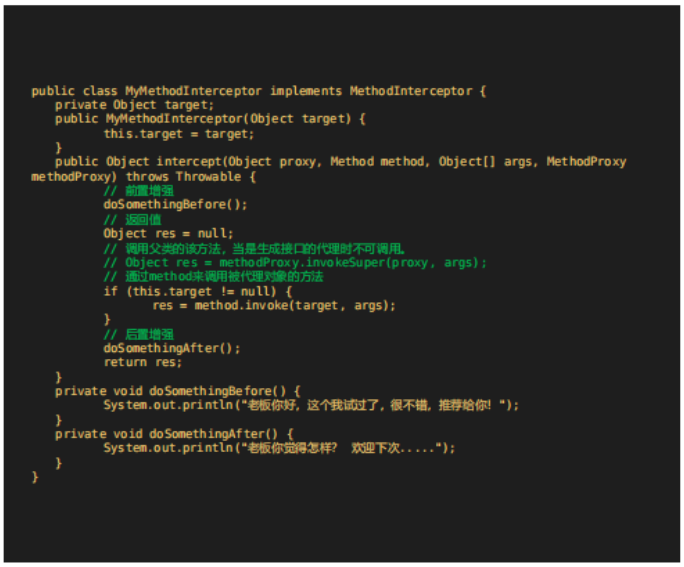
- 引入它的jar
- 学习它的API

```
<!-- https://mvnrepository.com/artifact/cglib/cglib -->
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.2.6</version>
</dependency>
```

代理模式-cglib动态代理-类图&API



代理模式-cglib动态代理-代码示例



```
public class MyMethodInterceptor implements MethodInterceptor {
    private Object target;
    public MyMethodInterceptor(Object target) {
        this.target = target;
    }
    public Object intercept(Object proxy, Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {
        // 前置增强
        doSomethingBefore();
        // 返回值
        Object res = null;
        // 调用父类的该方法，当是生成接口的代理时不可调用。
        // Object res = methodProxy.invokeSuper(proxy, args);
        // 通过method来调用被代理对象的方法
        if (this.target != null) {
            res = method.invoke(target, args);
        }
        // 后置增强
        doSomethingAfter();
        return res;
    }
    private void doSomethingBefore() {
        System.out.println("老板你好，这个我试过了，很不错，推荐给你！");
    }
    private void doSomethingAfter() {
        System.out.println("老板你觉得怎样？ 欢迎下次.....");
    }
}
```

```
public class CglibDemo {
    public static void main(String[] args) {
        Enhancer e = new Enhancer();
        TeacherCang tc = new TeacherCang();
        // 设置增强回调
        e.setCallback(new MyMethodInterceptor(tc));

        // 获得接口代理对象
        e.setInterfaces(new Class[] { Girl.class });
        Girl g = (Girl) e.create();
        g.dating(1.8f);

        // 对类生成代理对象
        e.setSuperclass(TeacherCang.class);
        e.setInterfaces(null);
        //当有多个callback时，需要通过callbackFilter来指定被代理方
法使用第几个callback
        /* e.setCallbackFilter(new CallbackFilter() {
            @Override
            public int accept(Method method) {
                return 0;
            }
        });*/
        TeacherCang proxy = (TeacherCang) e.create();
        proxy.dating(1.8f);
    }
}
```

cglib代理类的生成类

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//

package com.example.designpattern.sample.proxy.staticproxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;

public final class TeacherCang$$EnhancerByCGLIB$$95d275e0 extends Proxy
implements TeacherCang {
    private static Method m1;
    private static Method m8;
    private static Method m2;
    private static Method m3;
    private static Method m6;
    private static Method m5;
    private static Method m7;
    private static Method m9;
    private static Method m0;
    private static Method m4;

    public TeacherCang$$EnhancerByCGLIB$$95d275e0(InvocationHandler var1) throws
{
        super(var1);
    }

    public final boolean equals(Object var1) throws  {
        try {
            return (Boolean)super.h.invoke(this, m1, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
```

```
            throw new UndeclaredThrowableException(var4);
        }
    }

    public final void notify() throws  {
        try {
            super.h.invoke(this, m8, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final String toString() throws  {
        try {
            return (String)super.h.invoke(this, m2, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final boolean dating(float var1) throws  {
        try {
            return (Boolean)super.h.invoke(this, m3, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
            throw new UndeclaredThrowableException(var4);
        }
    }

    public final void wait(long var1) throws InterruptedException {
        try {
            super.h.invoke(this, m6, new Object[]{var1});
        } catch (RuntimeException | InterruptedException | Error var4) {
            throw var4;
        } catch (Throwable var5) {
            throw new UndeclaredThrowableException(var5);
        }
    }

    public final void wait(long var1, int var3) throws InterruptedException {
        try {
            super.h.invoke(this, m5, new Object[]{var1, var3});
        } catch (RuntimeException | InterruptedException | Error var5) {
            throw var5;
        } catch (Throwable var6) {
            throw new UndeclaredThrowableException(var6);
        }
    }

    public final Class getClass() throws  {
        try {
            return (Class)super.h.invoke(this, m7, (Object[])null);
        } catch (RuntimeException | Error var2) {
```

```
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final void notifyAll() throws  {
        try {
            super.h.invoke(this, m9, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final int hashCode() throws  {
        try {
            return (Integer)super.h.invoke(this, m0, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final void wait() throws InterruptedException {
        try {
            super.h.invoke(this, m4, (Object[])null);
        } catch (RuntimeException | InterruptedException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    static {
        try {
            m1 = Class.forName("java.lang.Object").getMethod("equals",
Class.forName("java.lang.Object"));
            m8 =
Class.forName("com.example.designpattern.sample.proxy.staticproxy.TeacherCang").
getMethod("notify");
            m2 = Class.forName("java.lang.Object").getMethod("toString");
            m3 =
Class.forName("com.example.designpattern.sample.proxy.staticproxy.TeacherCang").
getMethod("dating", Float.TYPE);
            m6 =
Class.forName("com.example.designpattern.sample.proxy.staticproxy.TeacherCang").
getMethod("wait", Long.TYPE);
            m5 =
Class.forName("com.example.designpattern.sample.proxy.staticproxy.TeacherCang").
getMethod("wait", Long.TYPE, Integer.TYPE);
            m7 =
Class.forName("com.example.designpattern.sample.proxy.staticproxy.TeacherCang").
getMethod("getClass");
```

```
            m9 =
Class.forName("com.example.designpattern.sample.proxy.staticproxy.TeacherCang").
getMethod("notifyAll");
            m0 = Class.forName("java.lang.Object").getMethod("hashCode");
            m4 =
Class.forName("com.example.designpattern.sample.proxy.staticproxy.TeacherCang").
getMethod("wait");
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```

cglib代理接口的生成类

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//

package com.example.designpattern.sample.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;

public final class Girl$$EnhancerByCGLIB$$c440ee8f extends Proxy implements Girl
{
    private static Method m1;
    private static Method m2;
    private static Method m3;
    private static Method m0;

    public Girl$$EnhancerByCGLIB$$c440ee8f(InvocationHandler var1) throws  {
        super(var1);
    }

    public final boolean equals(Object var1) throws  {
        try {
            return (Boolean)super.h.invoke(this, m1, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
            throw new UndeclaredThrowableException(var4);
        }
    }

    public final String toString() throws  {
        try {
            return (String)super.h.invoke(this, m2, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
```

```
        }
    }

    public final boolean dating(float var1) throws  {
        try {
            return (Boolean)super.h.invoke(this, m3, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
            throw new UndeclaredThrowableException(var4);
        }
    }

    public final int hashCode() throws  {
        try {
            return (Integer)super.h.invoke(this, m0, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    static {
        try {
            m1 = Class.forName("java.lang.Object").getMethod("equals",
Class.forName("java.lang.Object"));
            m2 = Class.forName("java.lang.Object").getMethod("toString");
            m3 =
Class.forName("com.example.designpattern.sample.proxy.Girl").getMethod("dating",
Float.TYPE);
            m0 = Class.forName("java.lang.Object").getMethod("hashCode");
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```

责任链模式

应用场景：

http web请求处理，请求过来后将经过转码、解析、参数封装、鉴权。。。一系列的处理（责任），而且要经过多少处理是可以灵活调整的。

将所有的处理都写在一个类中可否?

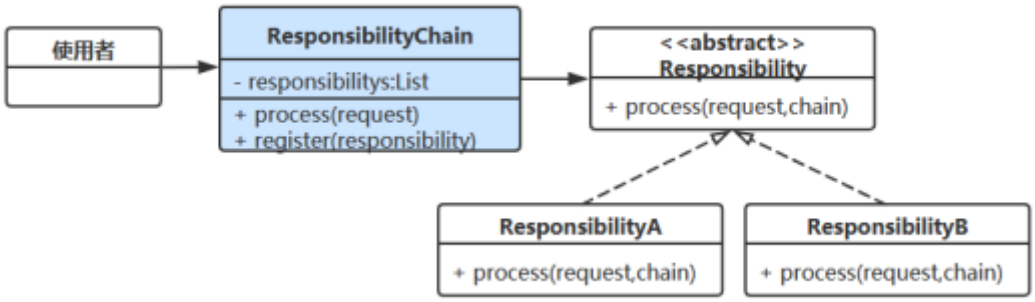
分成多个类如何灵活组合在一起?

责任链：所有的处理者，都加入这个链式，一个处理完，转给下一个

- 抽象责任接口，具体责任逻辑实现接口
- 根据处理过程需要，将具体实现组合成链
- 使用者使用链

典型代表：Filter、Interceptor

责任链模式-类图



和装饰者的区别在哪里?

责任链模式-代码示例

```
public interface Responsibility {
    void process(Request request, ResponsibilityChain chain);
}
```

```
public class ResponsibilityA implements Responsibility {
    @Override
    public void process(Request request, ResponsibilityChain chain) {
        System.out.println("Responsibility-A done something...");
        chain.process(request);
    }
}
```

```
public class ResponsibilityB implements Responsibility {
    @Override
    public void process(Request request, ResponsibilityChain chain) {
        System.out.println("Responsibility-B done something...");
        chain.process(request);
    }
}
```

```
public interface Request {
    // 定义你想要的
}
```

```
public class ResponsibilityChain {

    private List<Responsibility> responsibilitys;
    private int index = 0;

    public ResponsibilityChain() {
        this.responsibilitys = new ArrayList<>();
    }

    public void process(Request request) {
        if (this.index < this.responsibilitys.size()) {
            this.responsibilitys.get(index++).process(request, this);
        }
    }

    public void register(Responsibility res) {
        this.responsibilitys.add(res);
    }
}
```

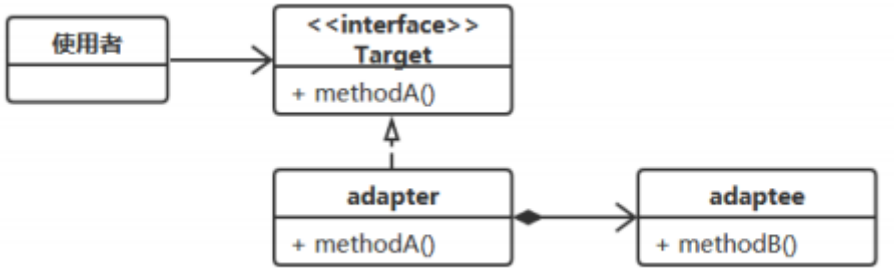
```
public class PlayGame {
    public static void main(String[] args) {
        ResponsibilityChain chain = new ResponsibilityChain();
        chain.register(new ResponsibilityA());
        chain.register(new ResponsibilityB());

        chain.process(new Request() {
        });
    }
}
```

```
Responsibility-A done something...
Responsibility-B done something...
```

适配器模式

应用场景：

使用者依赖的接口与提供者的接口不匹配时，就加一层适配，而不改两端的代码。

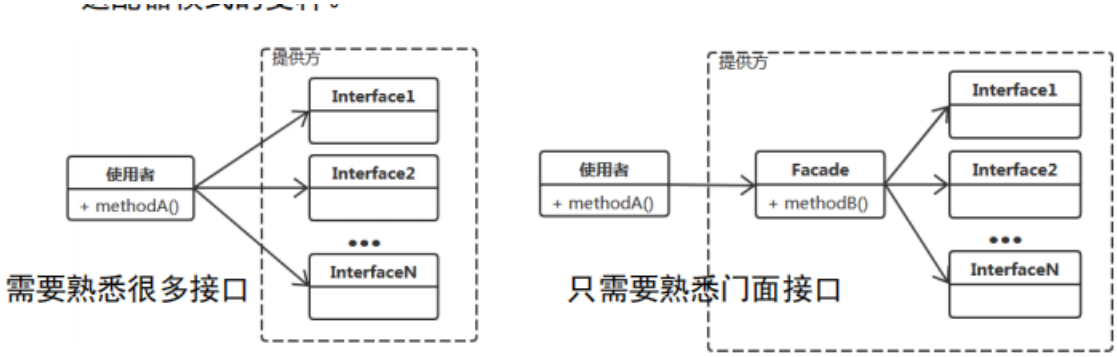


和代理、装饰的区别在哪里？

外观（门面）模式

应用场景：

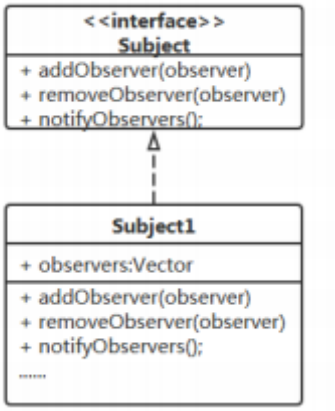
使用方要完成一个功能，需要调用提供的多个接口、方法，调用过程复杂时，我们可以再提供一个高层接口（新的外观），将复杂的调用过程向使用方隐藏。

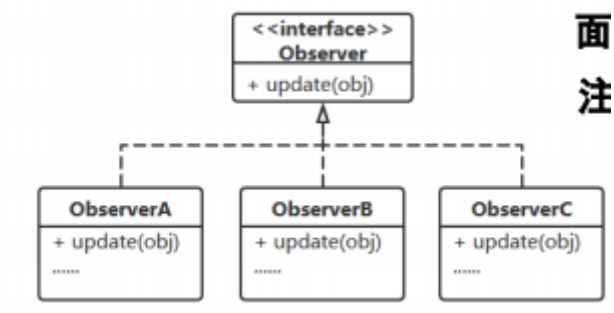


设计原则：最少知道原则

观察者模式

示例：微信公众号，关注就可以收到消息，取消关注，就不会再收到。





面向接口编程

注册、回调机制

变化之处：观察者会变，观察者的数量会变。

不变：主题的代码要不受观察者变化的影响

定义：

定义类对象之间一对多的依赖关系，当一端对象改变状态时，它的所有依赖者都会收到通知并自动更新（被调用更新方法）。

也称为：监听模式、发布订阅模式。提供一种对象之间松耦合的设计方式。

设计模式：为了交互对象之间的松耦合设计而努力！

java中为我们提供了观察者模式的通用实现

java.util.Observable 可被观察的（主题），具体主题扩展它。

java.util.Observer 观察者接口，具体观察者实现该接口

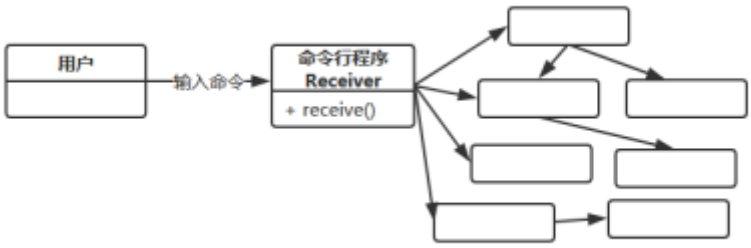
使用实例

```
public class ObserverSample {

    public static void main(String[] args) {
        Observable subject1 = new Observable() {
            public synchronized void notifyObservers(Object data) {
                setChanged();
                super.notifyObservers(data);
            }
        };

        subject1.addObserver(new Observer() {
            @Override
            public void update(Observable o, Object arg) {
                System.out.println("观察者1收到通知被更新了..." + arg);
            }
        });

        subject1.addObserver(new Observer() {
            @Override
            public void update(Observable o, Object arg) {
                System.out.println("观察者2收到通知被更新了..." + arg);
            }
        });

        subject1.notifyObservers("change1");
        subject1.notifyObservers("change2");
    }
}
```

命令模式

示例

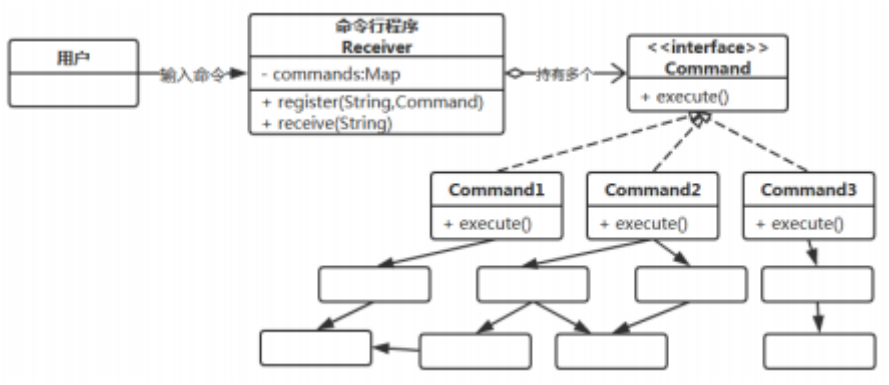
请你为系统设计一个命令行界面，用户可输入命令来执行某项功能。系统的功能会不断添加，命令也会不断增加。如何将一项一项的功能加入到这个命令行界面?



如何让我们的命令行程序写好后，不因功能的添加后修改，又可灵活加入命令、功能。请为此做设计

```
public class Receiver{
    public void receive(String command) {
        switch (command) {
        case "command-1":
            // 执行功能1
            ......
            break;
        case "command-2":
            // 执行功能2
            ......
            break;
        case "command-3":
            // 执行功能3
            ......
            break;
        ......
        }
        System.out.println("不支持此命令" + command);
    }
}
```

命令模式-类图




```
public class Receiver{
    private Map<String,Command> commands;

    public void register(String strComm,Command command){
            commands.put(strComm,command);
    }

    public void receive(String command) {
            Command commandObj = commands.get(command);
            if(commandObj != null){
                    commandObj.execute();
                    return;
            }
            System.out.println("不支持此命令" + command);
    }
}
```

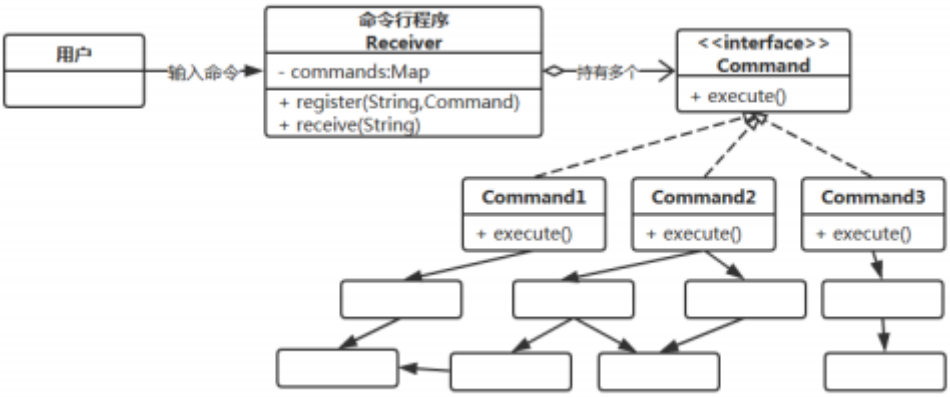
命令模式

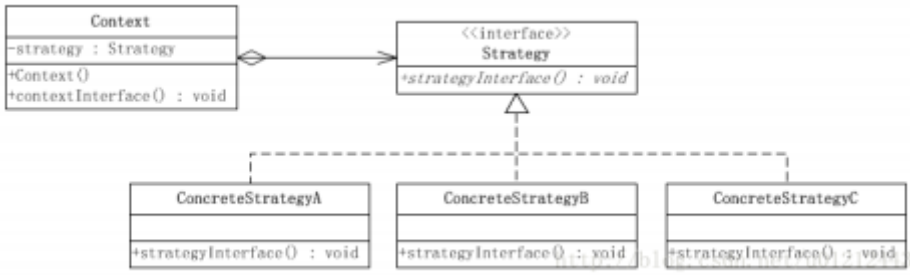
以命令的方式，解耦调用者与功能的具体实现者，降低系统耦合度，提供了灵活性。

适用场景：交互场景

实例：Servlet　　Controller　　线程池

命令模式-策略模式的区别



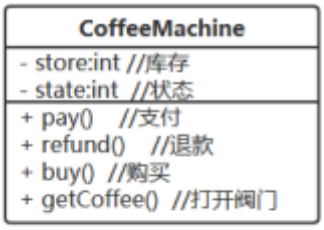


- 策略模式侧重的是一个行为的多个算法实现，可互换算法
- 命令模式侧重的是为多个行为提供灵活的执行方式

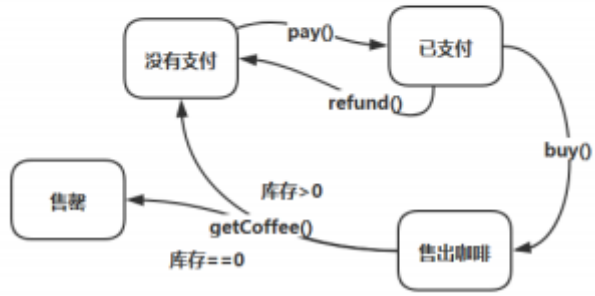
状态模式

示例

一个类对外提供多个行为，同时该类对象有多种状态，不同状态下对外行为的表现不同，我们该如何来设计该类让它对状态可以灵活扩展？



咖啡机状态转换图



- 用户可在咖啡机上进行支付、退款、购买、取咖啡操作
- 不同的状态下，这四种操作将有不同的表现

  如在没有支付状态下，用户在咖啡机上点退款、购买、取咖啡，和在已支付的状态下做这三个操作。

代码示例

```
public class CoffeeMachine {

    final static int NO_PAY = 0;
    final static int PAY = 1;
    final static int SOLD = 2;
    final static int SOLD_OUT = 4;

    private int state = SOLD_OUT;
    private int store;

    public CoffeeMachine(int store) {
        this.store = store;
        if (this.store > 0) {
            this.state = NO_PAY;
        }
    }

    public void pay() {...}
    public void refund() {...}
    public void buy() {...}
    public void getCoffee() {...}
    ...
}
```

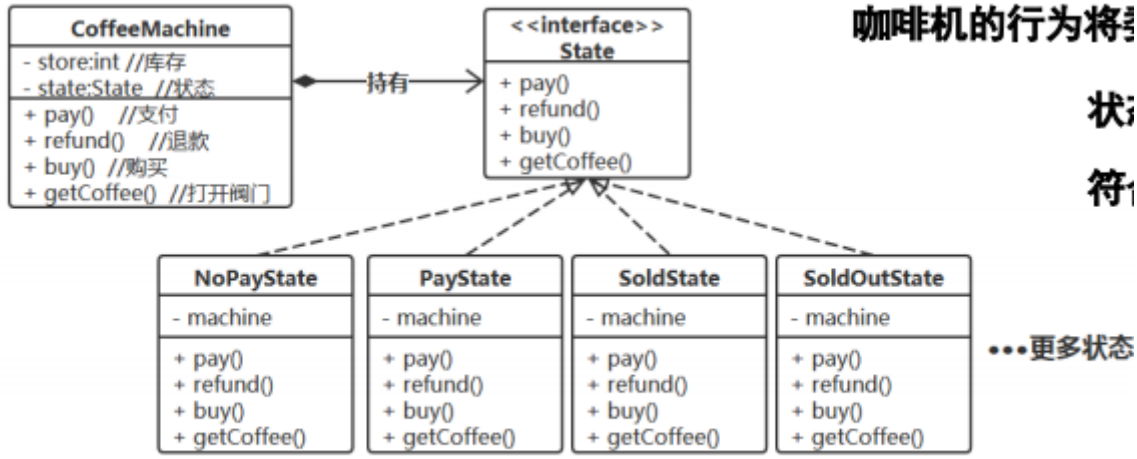
```
public void pay() {
    switch (this.state) {
    case NO_PAY:
        System.out.println("支付成功，请确定购买咖啡。");
        this.state = PAY;
        break;
    case PAY:
        System.out.println("已支付成功，请确定购买咖啡。");
        break;
    case SOLD:
        System.out.println("不可支付，已购买请取用咖啡！");
        break;
    case SOLD_OUT:
        System.out.println("咖啡已售罄，不可购买！");
    }
}
```

```
public void refund() {
    switch (this.state) {
    case NO_PAY:
        System.out.println("你尚未支付，请不要乱按！");
        break;
    case PAY:
        System.out.println("退款成功！");
        this.state = NO_PAY;
        break;
    case SOLD:
        System.out.println("已购买，请取用！");
        break;
    case SOLD_OUT:
        System.out.println("咖啡已售罄，不可购买！");
    }
}
```

如何让状态可以灵活扩展（加状态）？

如何做到开闭原则?

转态模式-类图



咖啡机的行为将委托给当前的状态实例状态可以灵活扩展否？符合开闭原则否？

新代码示例

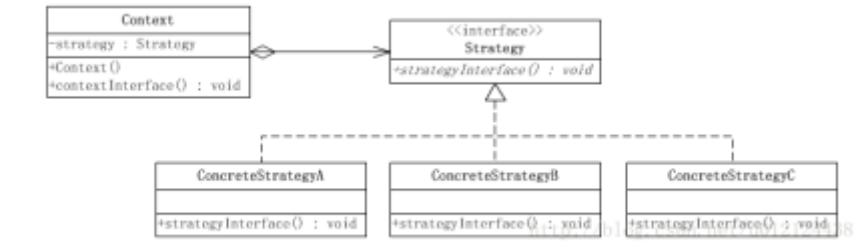
```
public interface State {
    void pay();
    void refund();
    void buy();
    void getCoffee();
}
```

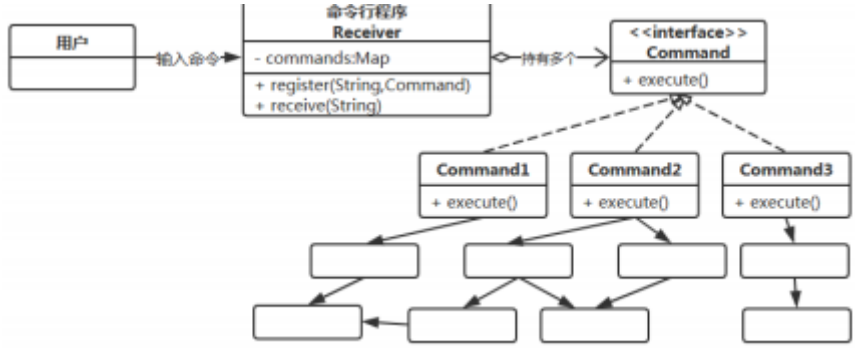
```
public class NoPayState implements State {
    private NewCoffeeMachine machine;

    public NoPayState(NewCoffeeMachine machine) {
        this.machine = machine;
    }

    public void pay() {
        System.out.println("支付成功，请去确定购买咖啡。");
        this.machine.state = this.machine.PAY;
    }
    public void refund() {
        System.out.println("你尚未支付，请不要乱按！");
    }
    public void buy() {
        System.out.println("你尚未支付，请不要乱按！");
    }
    public void getCoffee() {
        System.out.println("你尚未支付，请不要乱按！");
    }
}
```

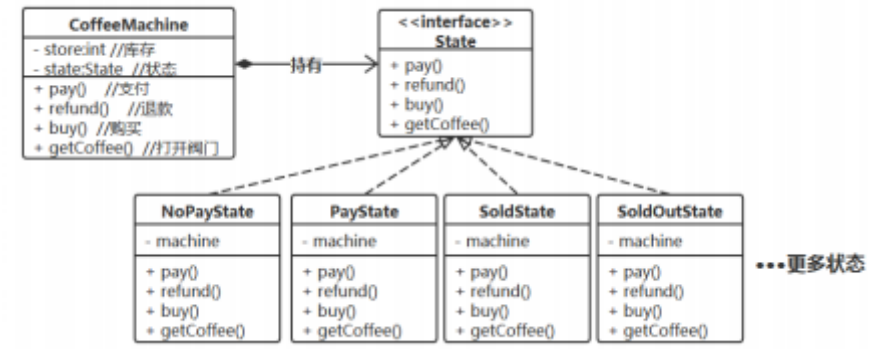
```
public class PayState implements State {
    private NewCoffeeMachine machine;

    public PayState(NewCoffeeMachine machine) {
        this.machine = machine;
    }
    public void pay() {
        System.out.println("您已支付，请去确定购买！");
    }
    public void refund() {
        System.out.println("退款成功，请收好！");
        this.machine.state = this.machine.NO_PAY;
    }
    public void buy() {
        System.out.println("购买成功，请取用");
        this.machine.state = this.machine.SOLD;
    }
    public void getCoffee() {
        System.out.println("请先确定购买！");
    }
}
```

```
public class NewCoffeeMachine {
    final State NO_PAY, PAY, SOLD, SOLD_OUT;
    State state;
    int store;

    public NewCoffeeMachine(int store) {
        NO_PAY = new NoPayState(this);
        PAY = new PayState(this);
        SOLD = new SoldState(this);
        SOLD_OUT = new SoldOutState(this);

        this.store = store;
        if (this.store > 0) {
            this.state = NO_PAY;
        }
    }

    public void pay() {
        this.state.pay();
    }
    public void refund() {
        this.state.refund();
    }
    public void buy() {
        this.state.buy();
    }
    public void getCoffee() {
        this.state.getCoffee();
    }
}
```

状态模式-命令模式-策略模式







- 策略模式侧重的是一个行为的多个算法实现，可互换算法

- 命令模式侧重的是多个行为提供灵活的执行方式
- 状态模式，应用于状态机的情况

设计原则：区分变与不变，隔离变化

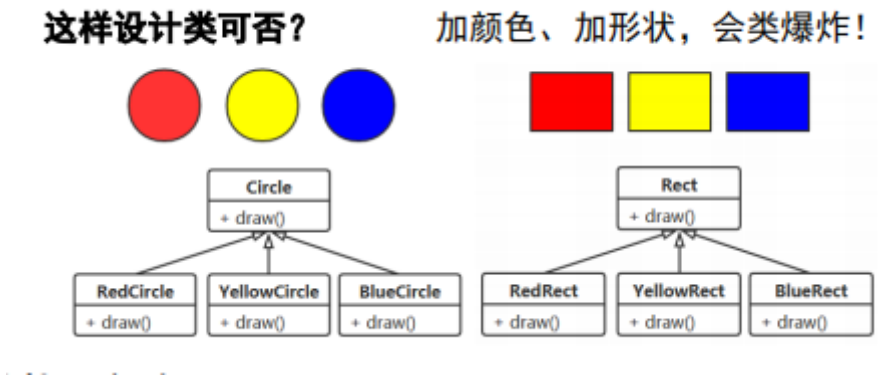
设计原则：面向接口编程

设计原则：多用组合，少用继承

桥接模式

示例

请开发一个画图程序，可以画各种不同形状的图形，请用面向对象的思想设计图形。



分析

- 比如有红、黄、蓝三种颜色
- 形状有方形、圆、三角形
- 圆可以是红圆、黄圆、蓝圆

变化

会从两个维度发生变化：形状、颜色

任其在两个微幅各自变化，为这两个维度搭个桥，让他们可以融合在一起：桥接模式

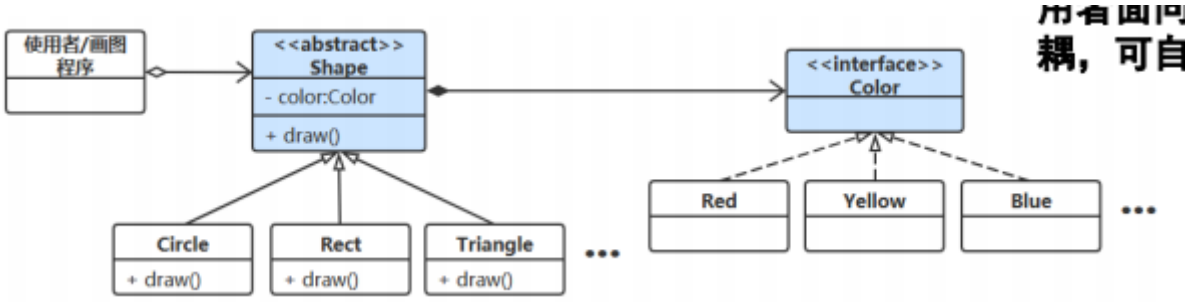
如何搭?

抽象

分别对各个维度进行抽象，将共同部分提取出来

组合

将抽象组合在一起（桥接）



桥接：将多个维度的变化以抽象的方式组合在一起。使用者面向抽象。各维度间解耦，可自由变化。

单例模式

饥汉式（可用）

```
public class Singleton {

    private final static Singleton INSTANCE = new
Singleton();

    private Singleton(){}

    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

```
public class Singleton {

    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

懒汉式

- 懒汉式1

```
public class Singleton {                    【不可用】

    private static Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

- 懒汉式2

```
public class Singleton {                    【线程安全，但不推荐】

    private static Singleton singleton;

    private Singleton() {}

    public static synchronized Singleton
getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

缺点：实例化后就不应该再同步了，效率低。

- 懒汉式3

```
public class Singleton {                    【不可用】

    private static Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                singleton = new Singleton();
            }
        }
        return singleton;
    }
}
```

做不到单例

- 懒汉式4：双层检查

```
public class Singleton {                                    【推荐使用】
    private static volatile Singleton singleton;
    private Singleton() {}
    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

注意：volatile关键字修饰很关键

优点：线程安全；延迟加载；效率高

- 懒汉式5 静态内部类方式

```
public class Singleton {                                    【推荐使用】
    private Singleton() {}
    private static class SingletonInstance {
        private static final Singleton INSTANCE =
new Singleton();
    }
    public static Singleton getInstance() {
        return SingletonInstance.INSTANCE;
    }
}
```

优点：避免了线程不安全，延迟加载，效率高

原理：类的静态属性只会在第一次加载类的时候初始化。在这里，JVM帮助我们保证了线程的安全，在类进行初始化时，别的线程是无法进入的。

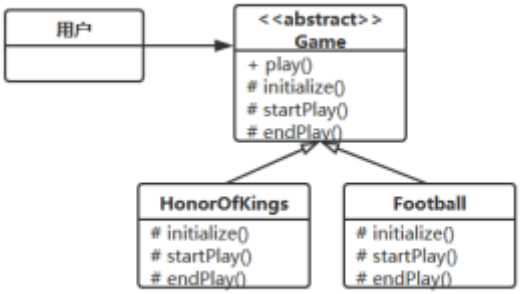
- 懒汉式6 用枚举

```
public enum Singleton {                                     【推荐使用】
    INSTANCE;
    public void whateverMethod() {
    }
}
```

模板方法模式

示例：当我们设计一个类时，我们能明确它对外提供的某个方法的内部执行步骤，但一些步骤，不同的子类有不同的行为时，我们该如何来设计该类?

可以用模板方法模式




```
public abstract class Game {
    protected abstract void initialize();
    protected abstract void startPlay();
    protected abstract void endPlay();
    //模板方法
    public final void play(){
        //初始化游戏
        initialize();
        //开始游戏
        startPlay();
        //结束游戏
        endPlay();
    }
}
```

优点

- 封装不变部分，扩展可变部分
- 提取公共代码，便于维护
- 行为由父类控制，子类实现

使用场景

- 有多个子类共有的方法，且逻辑相同
- 重要的、复杂的方法，可以考虑作为模板方法

设计模式总结

序号	模式 & 描述	包括
1	**创建型模式** 这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。	•工厂模式（Factory Pattern） •抽象工厂模式（Abstract Factory Pattern） •单例模式（Singleton Pattern） •建造者模式（Builder Pattern） •原型模式（Prototype Pattern）
2	**结构型模式** 这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。	•适配器模式（Adapter Pattern） •桥接模式（Bridge Pattern） •组合模式（Composite Pattern） •装饰器模式（Decorator Pattern） •外观模式（Facade Pattern） •享元模式（Flyweight Pattern） •代理模式（Proxy Pattern）
3	**行为型模式** 这些设计模式特别关注对象之间的通信。	•责任链模式（Chain of Responsibility Pattern） •命令模式（Command Pattern） •解释器模式（Interpreter Pattern） •迭代器模式（Iterator Pattern） •中介者模式（Mediator Pattern） •备忘录模式（Memento Pattern） •观察者模式（Observer Pattern） •状态模式（State Pattern） •空对象模式（Null Object Pattern） •策略模式（Strategy Pattern） •模板模式（Template Pattern） •访问者模式（Visitor Pattern）

变化隔离原则

找出变化，分开变化与不变的

隔离，封装变化的部分，让其他部分不受它的影响。

面向接口编程 依赖倒置 隔离变化的方式

使用者使用接口，提供者实现接口。“接口”可以是超类！

开闭原则

对修改闭合，对扩展开放 隔离变化的方式

多用组合，少用继承

灵活变化的方式

最少知道原则 又称迪米特法则

单一职责原则

方式设计的原则

最后，如果都忘记了，请一定要记住这三