# 学习地址

http://www.runoob.com/design-pattern/design-pattern-tutorial.html

# 创建型模式（5种）

## 工厂方法模式

分为：

普通工程模式，根据传递的字符串判断返回什么样的对象。

多个工厂方法模式，提供多个方法，每个方法返回不同的对象。是普通的改进版，防止 传递的字符串不存在。

静态工厂方法模式，可以直接通过类名.方法进行获取对象。

首先创建一个接口，Shape

public interface Shape {

void draw();

}

创建实现类，Rectangle，Square，Circle

创建工厂

|  |
| --- |
| public class ShapeFactory {    //使用 getShape 方法获取形状类型的对象  public Shape getShape(String shapeType){  if(shapeType == null){  return null;  }  if(shapeType.equalsIgnoreCase("CIRCLE")){  return new Circle();  } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  return new Rectangle();  } else if(shapeType.equalsIgnoreCase("SQUARE")){  return new Square();  }  return null;  }  } |

例如想要得到Square对象，

Shape shape = shapeFactory.getShape("Square");

## 抽象工厂模式

工厂模式中的工厂定义的是Shape工厂，所以这里只能生产Shape类型的对象，现在还想要Color工厂。所以这里又向上抽取一层，由抽象工厂生产工厂，然后再在相应的工厂中获取对象。

抽象工厂就像工厂模式中的工厂，工厂模式中生产的是产品（对象），抽象工厂生产的就是工厂，工厂具体生产产品。

## 单例模式

负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不用实例化改类的对象。

|  |
| --- |
| **public** **class** SingleObject {  **private** **static** SingleObject *instance* = **new** SingleObject();    /\*\*  \* 让构造函数为private，这样该类就不会被实例化  \*/  **private** SingleObject(){    }    **public** **static** SingleObject getInstance(){  **return** *instance*;  }    **public** **void** showMessage(){  System.***out***.println("HelloWord！");  }  } |

单例模式的几种实现方式

1. 懒汉式并且线程不安全

这种方式最大的问题就是不支持多线程。

|  |
| --- |
| **public** **class** SingleObject {  **private** **static** SingleObject *instance* ;    /\*\*  \* 让构造函数为private，这样该类就不会被实例化  \*/  **private** SingleObject(){    }    **public** **static** SingleObject getInstance(){  **if**(*instance*== **null**){  *instance* = **new** SingleObject();  }  **return** *instance*;  }    } |

以下的几种方式都支持多线程

1. 懒汉式并且线程安全

这种方式就是加了一个锁**synchronized**

|  |
| --- |
| **public** **class** SingleObject {  **private** **static** SingleObject *instance* ;    /\*\*  \* 让构造函数为private，这样该类就不会被实例化  \*/  **private** SingleObject(){    }    **public** **static** **synchronized** SingleObject getInstance(){  **if**(*instance*== **null**){  *instance* = **new** SingleObject();  }  **return** *instance*;  }    } |

1. 饿汉式

这种方式在类的装载时就会创建对象。但是容易产生垃圾对象。

|  |
| --- |
| **public** **class** SingleObject {  **private** **static** SingleObject *instance* = **new** SingleObject();    /\*\*  \* 让构造函数为private，这样该类就不会被实例化  \*/  **private** SingleObject(){    }    **public** **static** SingleObject getInstance(){  **return** *instance*;  }    **public** **void** showMessage(){  System.***out***.println("HelloWord！");  }  } |

1. 双检锁/双重校验锁（DCL，double-checked locking）

|  |
| --- |
| **public** **class** SingleObject {  **private** **volatile** **static** SingleObject *instance* ;    /\*\*  \* 让构造函数为private，这样该类就不会被实例化  \*/  **private** SingleObject(){    }    **public** **static** SingleObject getInstance(){    **if**(*instance* == **null**){  **synchronized** (SingleObject.**class**) {  **if**(*instance* == **null**){  *instance* = **new** SingleObject();  }  }    }  **return** *instance*;  }    } |

1. 登记式/静态内部类

SingleObject 被装载了，但是instance不一定被实例化，因为SingObjectHode还没被装载，只有调用getInstance()方法时才会装载SingObjectHode，并创建instance对象。

做到了延迟初始化。

|  |
| --- |
| **public** **class** SingleObject {  /\*\*  \* 这个是匿名内部类  \* **@author** hzl  \*  \*/  **private** **static** **class** SingleObjectHode{  **private** **static** **final** SingleObject ***INSTANCE*** = **new** SingleObject();  }    /\*\*  \* 让构造函数为private，这样该类就不会被实例化  \*/  **private** SingleObject(){    }    **public** **static** **final** SingleObject getInstance(){    **return** SingleObjectHode.***INSTANCE***;  }    } |

## 建造者模式

使用多个简单的对象一步一步构建成一个复杂的对象。应用实例：StringBuilder。

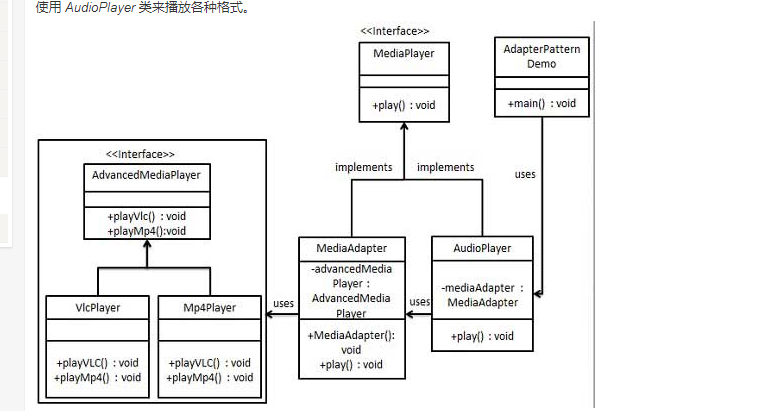
工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理。

## 原型模式

# 结构型模式（7种）

## 适配器模式

将一个接口转换成客户希望的另外一个接口，例如现在有一个AudioPlayer只能支持mp3格式文件的播放，还有一个AdvancedMediaPlayer支持mp4，vlc等多种格式，现在想让AudioPlayer也支持mp4和vlc格式的文件播放，就用用到适配器模式。







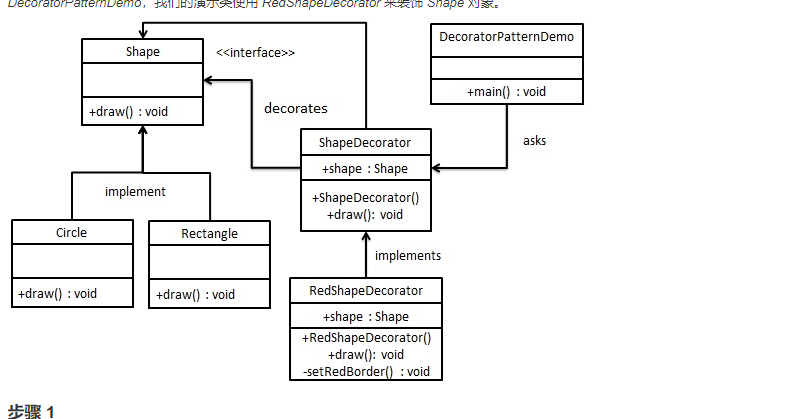


## 装饰器模式

就是向现有的对象添加新的功能，同时又不改变其结构。

一般的，为了扩展一个类经常使用的是继承的方式实现的，由于继承为类引入了静态特征，并且随着扩展功能的增多，子类会很膨胀。

优点：装饰类和被装饰类可以独立的发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。



## 代理模式

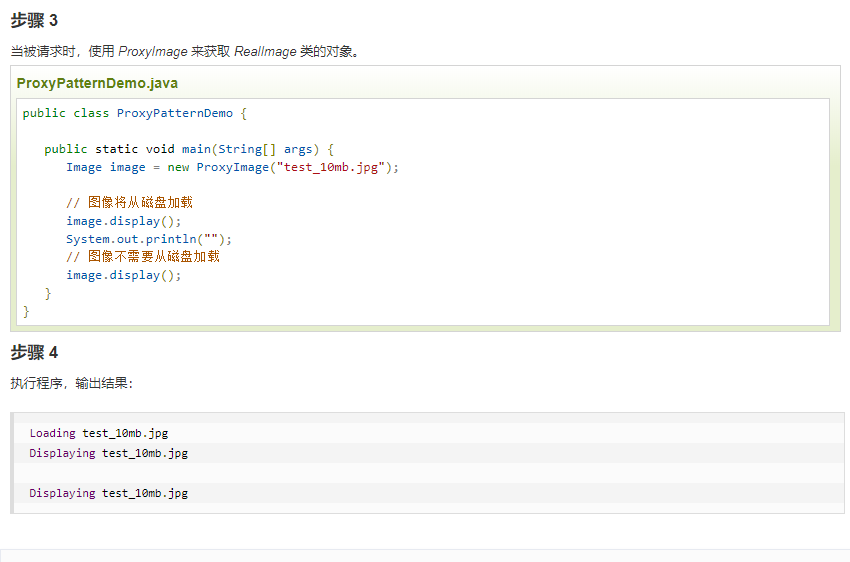
Spring AOP

优点：职责清晰，高扩展性，智能化。

和适配器的区别：适配器主要改变所考虑对象的接口，而代理不能改变所代理类的接口。

和装饰器的区别：装饰是为了增强功能，而代理是为了加以控制。



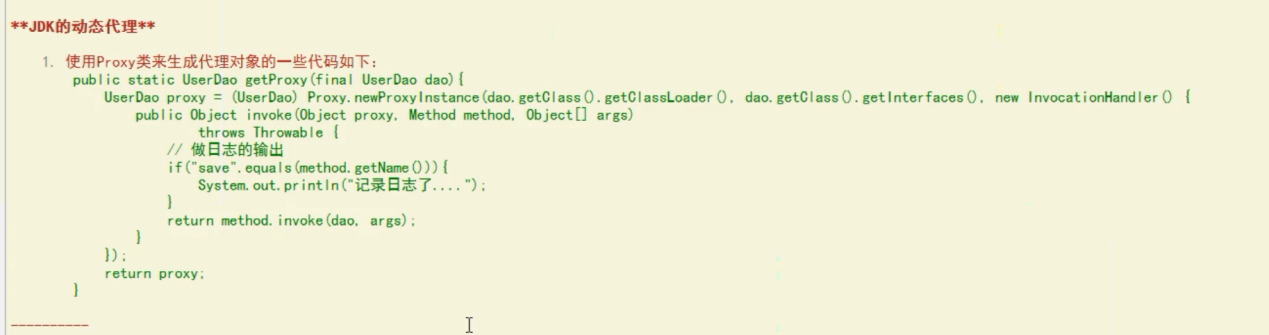


### 静态代理

### 动态代理

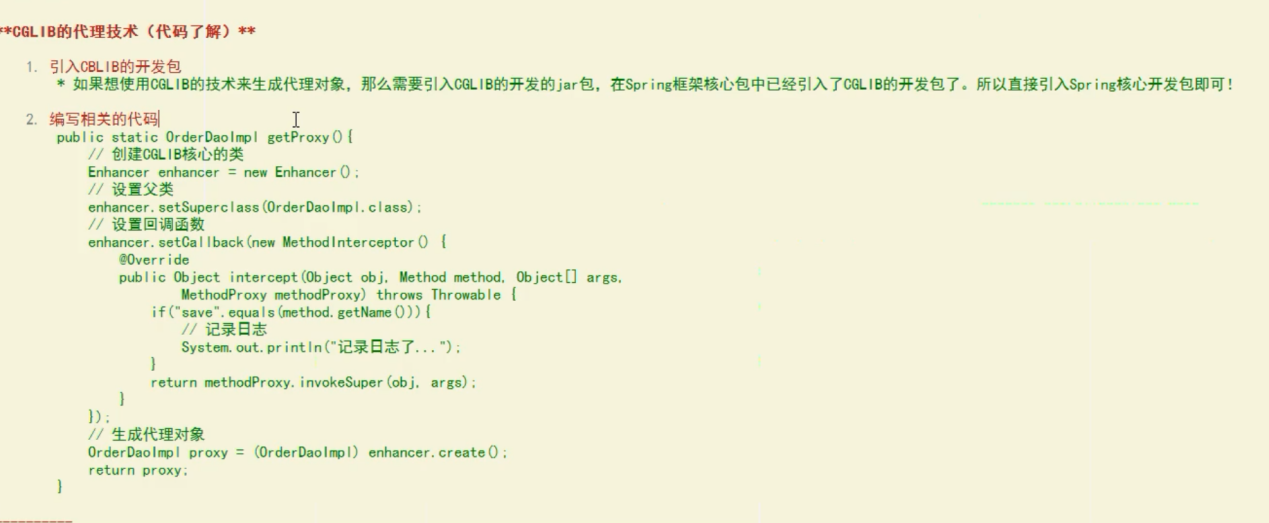
基于JDK的动态代理

必须面向接口，只有生成了具体接口的类才能生成代理对象



基于CGLIB动态代理

对于没有实现接口的类，也可以产生代理，产生这个类的子类的 方式。



## 外观模式

## 桥接模式

## 组合模式

## 享元模式

# 行为型模式（11种）

## 策略模式

## 模板方法模式

## 观察者模式

对象之间的一种一对多关系，每当一个对象改变状态时，则所有依赖他的对象都会得到通知并执行某些操作。也就是一种监听。

### 观察者模式和发布订阅模式

观察者模式是Subject(被观察者)直接通知Observers(观察者),面向接口的方式，实现松耦合。

观察者都要实现统一的接口。

发布订阅模式不是这样直接交流的，而是通过第三者。也就是消息队列里面常说的经纪人(Broker),发布者和订阅者之间是完全解耦的。

## 迭代子模式

## 责任链模式

## 命令模式

## 备忘录模式

## 状态模式

## 访问者模式

## 中介者模式

## 解释器模式

# 设计模式遵循的原则有6个

## 开闭原则（Open Close Principle）

对扩展开放，对修改关闭。

## 里氏代换原则（Liskov Substitution Principle）

　　只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。

## 依赖倒转原则（Dependence Inversion Principle）

　　这个是开闭原则的基础，**对接口编程**，依赖于抽象而不依赖于具体。

## 接口隔离原则（Interface Segregation Principle）

　　使用多个隔离的借口来降低耦合度。

## 迪米特法则（最少知道原则）（Demeter Principle）

　　一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

## 合成复用原则（Composite Reuse Principle）

　　原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改。