# 代码学习方法

10分钟的英文文档阅读（jdk，oracle…）

<https://docs.oracle.com/javase/tutorial/index.html>

1. 整体认知：知道整体效果（直接copy代码过来运行）
2. 跟着代码敲一遍，解决报错问题（百度搜索||定性。定范围||java帮助文档）
3. 一行一行的分析，在看代码的同时，需要理解出现自己未知的、不了解的东西：理解过程：（百度搜索-博客1--5篇||定性。定范围||java帮助文档），加入自己的代码风格。在代码的达标水平上，加入自己的尝试。
4. 不要去管有多少还要学，有多少已经学了，有时间，就研究。----无我的进行（空）。

# OOP下核心操作-设计模式

## 设计模式

### 引用资源：

设计模式：

<https://www.runoob.com/design-pattern/design-pattern-intro.html>

[JAVA设计模式总结之23种设计模式](https://www.cnblogs.com/pony1223/p/7608955.html)原理

<https://www.cnblogs.com/pony1223/p/7608955.html>

### 了解作用环境：（怎么学）

定义？有什么用？应用在什么环境？代码风格（规范）？怎么来使用？

### 学习框架

|  |  |
| --- | --- |
| [创建型模式](#_创建型模式) | * [单例模式](#_单例模式)（Singleton Pattern） * [工厂模式](#_工厂模式)（Factory Pattern） * [抽象工厂模式](#_抽象工厂模式)（Abstract Factory Pattern） * [建造者模式](#_建造者模式)（Builder Pattern） * [原型模式](#_原型模式)（Prototype Pattern） |
| 结构型模式 | * 适配器模式（Adapter Pattern） * 桥接模式（Bridge Pattern） * 过滤器模式（Filter、Criteria 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） |
| 行为型模式 | * 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） |
| J2EE模式 | * MVC 模式（MVC Pattern） * 业务代表模式（Business Delegate Pattern） * 组合实体模式（Composite Entity Pattern） * 数据访问对象模式（Data Access Object Pattern） * 前端控制器模式（Front Controller Pattern） * 拦截过滤器模式（Intercepting Filter Pattern） * 服务定位器模式（Service Locator Pattern） * 传输对象模式（Transfer Object Pattern） |

## 创建型模式

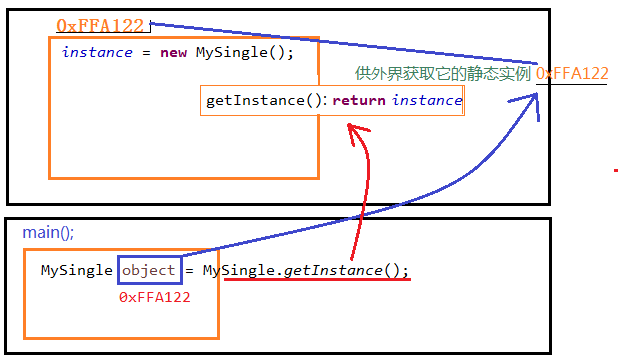
### 单例模式

生活实例场景应用：

回收站：

任务管理器：

参考：<https://www.cnblogs.com/whgw/archive/2011/10/05/2199535.html>



整体认知：

|  |
| --- |
| //饿汉式  //一开始就创建  **class** Singleton3 {  // 一开始就实例化静态对象  // 只要加载类 不管外界是否引用都会生成一个静态实例 浪费空间内存  // 创建 Singleton3 的一个对象,静态实例 instance=oxFFFAA123  **private** **static** **final** Singleton3 ***instance*** = **new** Singleton3();  // 让构造函数为 private，这样该类就不会被实例化  // 私有的构造函数：一个类只能有一个实例  **private** Singleton3() {  }  // 获取唯一可用的对象,通过return 供外界获取它的静态实例。new出来的 对象=oxFFFAA123  **public** **static** Singleton3 getInstance() {  **return** ***instance***;  }  **public** **void** show() {  System.***out***.println("饿汉式!!");  }  }Main(){  // 获取唯一可用的对象  Singleton3 object = Singleton3.*getInstance*();  } |

单例模式特点：

1、单例类只能有一个实例。

**private** MySingleObject () {}

2、单例类必须自己自己创建自己的唯一实例。

**private** **static** **final** MySingleObject ***instance*** = **new** MySingleObject();

3、单例类必须给所有其他对象提供这一实例。

**public** **static** MySingleObject getInstance(){

**return** *instance*;

}

**单例模式 实现方式1：**

|  |
| --- |
| /\*  \* 懒汉式1  \* 什么时候用就什么时候创建实例  \*/  **class** Singleton1 {  // 声明变量引用  **private** **static** Singleton1 *instance*;  // 私有的构造函数，这样该类就不会被实例化  // 该类只能有一个实例  **private** Singleton1() {};  // 单例类必须给所有其他对象提供这一实例。全局的访问点  **public** **static** Singleton1 getInstance() {  // 如果外界引用变量时为空，则实例出一个静态实例对象，否则直接返回对象  //问题：线程不安全：多个同学同时进入了一个面试房间，就造成了尴尬的局面  **if** (*instance* == **null**) {  //1 2 3 4 5 6....  *instance* = **new** Singleton1();  System.***out***.println("我是懒汉式1创建一次！！");  }  **return** *instance*;  }  **public** **void** show() {  System.***out***.println("我是懒汉式1");  }  } |

优点：

没有加锁，执行效率提高。

缺点：

线程不安全

**单例模式 实现方式2：（单同步锁）**

|  |
| --- |
| //懒汉式2  //基于解决懒汉式1的线程安全问题  **class** Singleton2 {  // 必须自己自己创建自己的唯一实例  // 注意：这里不能使用 final 修饰符，会报错  **private** **static** Singleton2 *instance*;  // 只能有一个实例  **private** Singleton2() {}  // 添加了一个同步锁 必须给所有其他对象提供这一实例  // 知识点：synchronized 特点：执行效率低，需要另一个线程创建完成之后才能进行下一个线程的开始  **public** **static** **synchronized** Singleton2 getInstance() {  // 判断是否需要创建实例  **if** (*instance* == **null**) {  // 创建实例  *instance* = **new** Singleton2();  System.***out***.println("我是懒汉式2-synchronized创建一次！！");  }  // 返回Singleton2的实例对象  **return** *instance*;  }  **public** **void** show() {  System.***out***.println("我是懒汉式2");  }  } |

**优点：**需要调用才初始化，避免内存浪费。

**缺点：**必须加锁 synchronized 才能保证单例，但加锁会影响效率。

|  |
| --- |
| //双重检查锁定  //基于解决懒汉式的加锁执行效率低的问题  **class** Singleton3 {  **private** **static** Singleton3 *instance* = **null**;  **private** Singleton3() {  }  **public** **static** Singleton3 getinstance() {  **if** (*instance* == **null**) {  **synchronized** (Singleton4.**class**) {  **if** (*instance* == **null**) {  *instance* = **new** Singleton3();  System.***out***.println("我是懒汉式2创建一次！！");  }  }  }  **return** *instance*;  }  } |

特点：延迟创建，线程安全，同步情况下效率高

|  |
| --- |
| //饿汉式  //一开始就创建  **class** Singleton4 {  // 一开始就实例化静态对象  // 只要加载类 不管外界是否引用都会生成一个静态实例 浪费空间内存  // 创建 Singleton3 的一个对象,静态实例 instance=oxFFFAA123  **private** **static** **final** Singleton4 ***instance*** = **new** Singleton4();  // 让构造函数为 private，这样该类就不会被实例化  // 私有的构造函数：一个类只能有一个实例  **private** Singleton4() {  }  // 获取唯一可用的对象,通过return 供外界获取它的静态实例。new出来的 对象=oxFFFAA123  **public** **static** Singleton4 getInstance() {  **return** ***instance***;  }  **public** **void** show() {  System.***out***.println("饿汉式!!");  }  } |

优点：

线程安全，没有加锁，执行效率提高。

缺点：

一开始就创建实例，浪费内存空间

|  |
| --- |
| //静态内部类  **class** Singleton5 {  **private** **static** **class** SingletonHolder {  **private** **static** **final** Singleton5 ***INSTANCE*** = **new** Singleton5();  }  **private** Singleton5() {  }  **public** **static** **final** Singleton5 getInstance() {  **return** SingletonHolder.***INSTANCE***;  }  } |

|  |
| --- |
| **public** **class** SingletonEnum {  //构造函数  **private** SingletonEnum() {    }  **public** **static** SingletonEnum getInstance() {  **return** Singleton.***INSTENCE***.getInstance();  }  //枚举  **private** **static** **enum** Singleton {  ***INSTENCE***;  **private** SingletonEnum instance;  **private** Singleton() {  instance = **new** SingletonEnum();  }  //共有的方法  **public** SingletonEnum getInstance() {  **return** instance;  }  }  } |

单例模式:

**意图：**保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**主要解决：**一个全局使用的类频繁地创建与销毁。

**何时使用：**当您想控制实例数目，节省系统资源的时候。

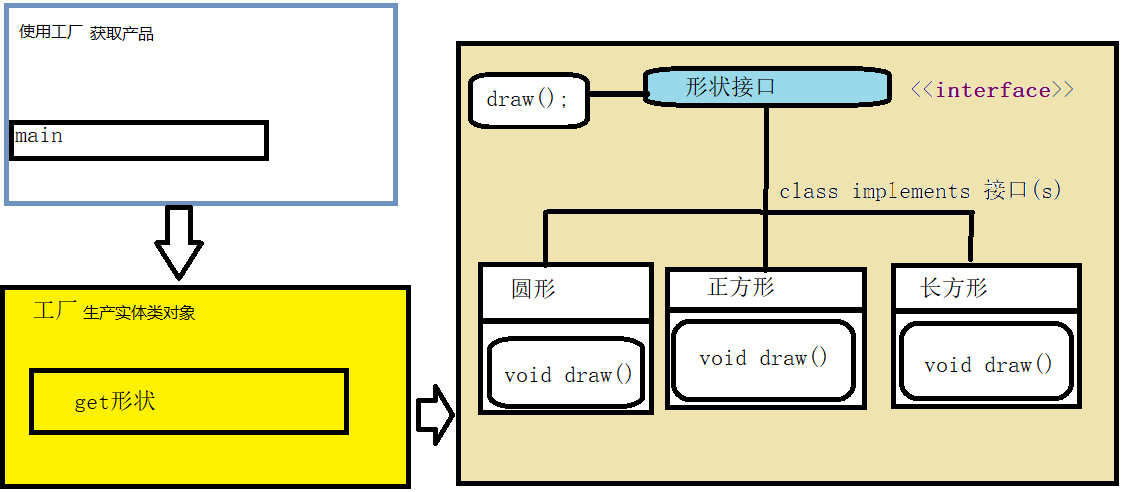
**如何解决：**判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

**关键代码：**构造函数是私有的。

同步锁线程：

### 工厂模式

1. 先创建一个接口（产品接口）
2. 创建子类（产品模型）
3. 创建一个工厂（虚拟的，）（产品生产车间）
4. 工厂实现（使用产品）



|  |
| --- |
| //形状接口类  **public** **interface** Shape {  **void** draw();  } |

|  |
| --- |
| //implements声明自己使用一个或者多个接口。 https://blog.csdn.net/tolcf/article/details/46135645  //创建实现接口的实体类。  //长方形  **public** **class** Rectangle **implements** Shape {  @Override  **public** **void** draw() {  System.***out***.println("Inside Rectangle::draw() method.");  }  } |
| //正方形  **public** **class** Square **implements** Shape {    @Override  **public** **void** draw() {  System.***out***.println("Inside Square::draw() method.");  }  } |
| //圆  **public** **class** Circle **implements** Shape {  @Override  **public** **void** draw() {  System.***out***.println("Inside Circle::draw() method.");  }  } |
| //椭圆》》圆  **public** **class** Ellipse **extends** Circle {  @Override  **public** **void** draw() {  System.***out***.println("Inside Ellipse::draw() method.");  }  } |

|  |
| --- |
| //创建一个工厂，生成基于给定信息的实体类的对象。  //产品的生产车间  **public** **class** ShapeFactory {  //设置形状的方法，  **public** Shape getShape(String shapeType) {  **if** (shapeType == **null**) {  **return** **null**;  }  //equalsIgnoreCase判断两字符串忽略大小写是否相等  **if** (shapeType.equalsIgnoreCase("CIRCLE")) {  //创建并返回一个形状对象  **return** **new** Circle();  } **else** **if** (shapeType.equalsIgnoreCase("RECTANGLE")) {  **return** **new** Rectangle();  } **else** **if** (shapeType.equalsIgnoreCase("SQUARE")) {  **return** **new** Square();  }**else** **if** (shapeType.equalsIgnoreCase("ELLIPSE")) {  **return** **new** Ellipse();  }  **return** **null**;  }  } |
| //使用该工厂，通过传递类型信息来获取实体类的对象。  **public** **class** FactoryPatternDemo {  **public** **static** **void** main(String[] args) {  //这里就像客户在餐厅点餐类似，只需要知道菜名就行了，不需要知道内部是这么做出来的，  //内部形成交由厨师(相当是工厂)完成。  /\*  \* 假如需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。  \*/  //实例一个工厂  ShapeFactory shapeFactory = **new** ShapeFactory();  // 获取 Circle 的对象，并调用它的 draw 方法  Shape shape1 = shapeFactory.getShape("CIRCLE");  // 调用 Circle 的 draw 方法  shape1.draw();  // 获取 Rectangle 的对象，并调用它的 draw 方法  Shape shape2 = shapeFactory.getShape("RECTANGLE");  // 调用 Rectangle 的 draw 方法  shape2.draw();  // 获取 Square 的对象，并调用它的 draw 方法  Shape shape3 = shapeFactory.getShape("SQUARE");  // 调用 Square 的 draw 方法  shape3.draw();  //  Shape shape4 = shapeFactory.getShape("ELLIPSE");  shape4.draw();  }  } |

特点：

1. 使用者只需要知道产品名称（产品）就行了，不需要知道具体生产方式
2. 可以扩展，
3. 屏蔽产品的具体实现，调用者只关心产品的接口

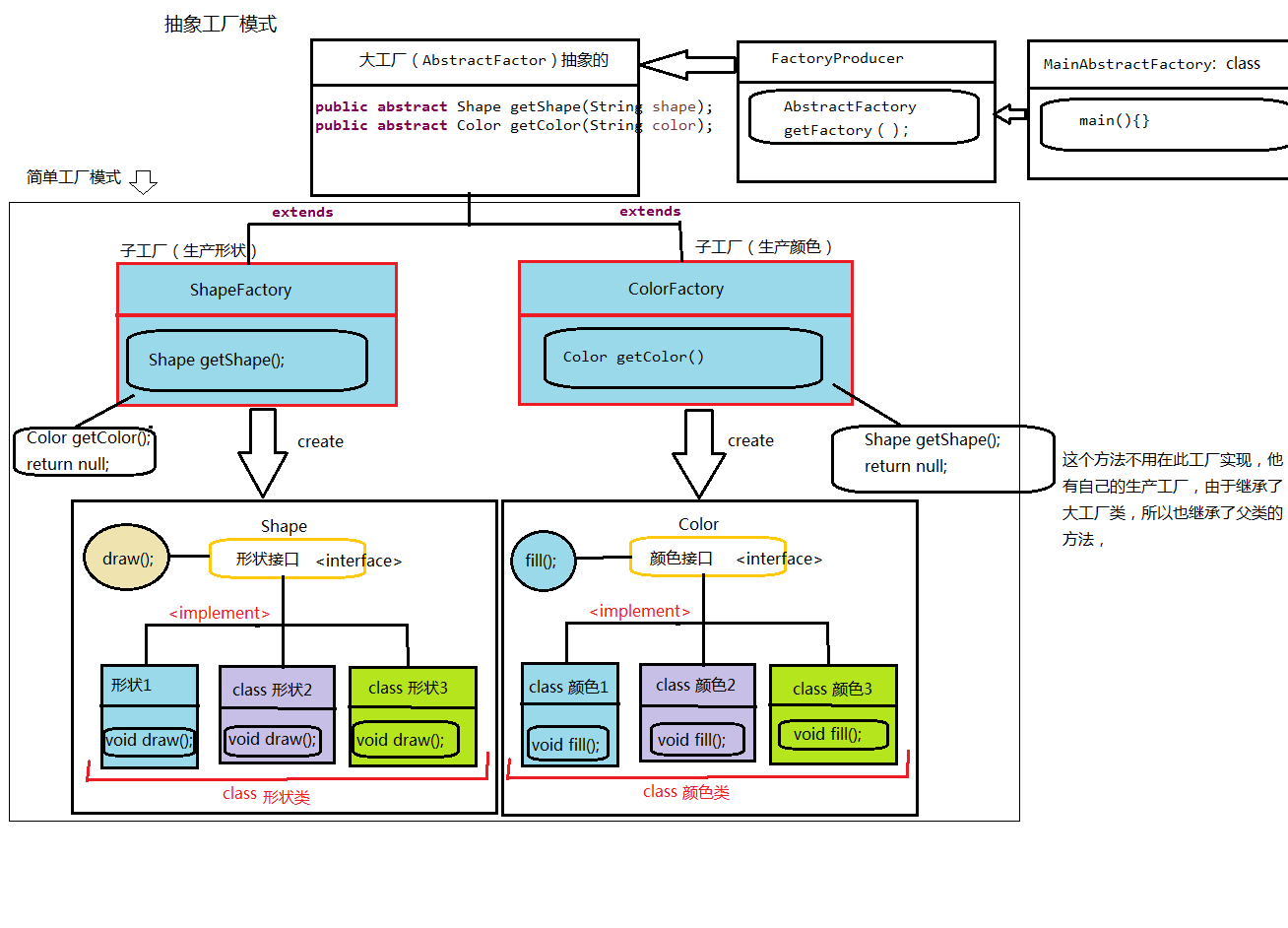
缺点：

每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。

使用场景：

使用者不需要知道产品的具体实现，只需要最终的产品。等等…

### 抽象工厂模式



特点：

扩展性繁琐。

抽象工厂：

|  |
| --- |
| //为 Color 和 Shape 对象创建抽象类来获取工厂。（大工厂）  **public** **abstract** **class** AbstractFactory {  **public** **abstract** Color getColor(String color);  **public** **abstract** Shape getShape(String shape);  } |
| //工厂生成器，把形状和颜色工厂实现  **public** **class** FactoryProducer {  **public** **static** AbstractFactory getFactory(String choice) {  **if** (choice.equalsIgnoreCase("SHAPE")) {  **return** **new** ShapeFactory();  } **else** **if** (choice.equalsIgnoreCase("COLOR")) {  **return** **new** ColorFactory();  }  **return** **null**;  }  } |
| **main()测试方法**  **public** **class** MainAbstractFactory {  **public** **static** **void** main(String[] args) {  // 通过FactoryProducer.getFactory()把相对应的工厂创建出来  // 实例形状工厂  // 获取形状工厂  AbstractFactory shapeFactory = FactoryProducer.*getFactory*("SHAPE");  // 获取形状为 Circle 的对象  Shape shape1 = shapeFactory.getShape("CIRCLE");  // 调用 Circle 的 draw 方法  shape1.draw();  // 获取形状为 Rectangle 的对象  Shape shape2 = shapeFactory.getShape("RECTANGLE");  // 调用 Rectangle 的 draw 方法  shape2.draw();  // 获取形状为 Square 的对象  Shape shape3 = shapeFactory.getShape("SQUARE");  // 调用 Square 的 draw 方法  shape3.draw();  System.***out***.println("###############");  // 实例颜色工厂  AbstractFactory colorFactory = FactoryProducer.*getFactory*("COLOR");  // 获取颜色为 Red 的对象  Color color1 = colorFactory.getColor("RED");  // 调用 Red 的 fill 方法  color1.fill();  // 获取颜色为 Green 的对象  Color color2 = colorFactory.getColor("Green");  // 调用 Green 的 fill 方法  color2.fill();  // 获取颜色为 Blue 的对象  Color color3 = colorFactory.getColor("BLUE");  // 调用 Blue 的 fill 方法  color3.fill();  }  } |

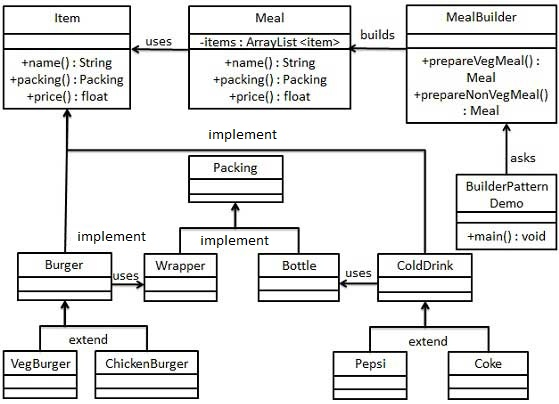
形状工厂：

|  |
| --- |
| //形状工厂（子工厂）  **public** **class** ShapeFactory **extends** AbstractFactory {  //工厂设置生产产品  @Override  **public** Shape getShape(String shapeType) {  **if**(shapeType==**null**) {  **return** **null**;  }  //条件成立则执行这条语句：  **if**(shapeType.contentEquals("CIRCLE")) {  //生产出这个产品Circle();  **return** **new** Circle();  }**else** **if**(shapeType.contentEquals("RECTANGLE")) {  **return** **new** Rectangle();  }**else** **if**(shapeType.contentEquals("SQUARE")) {  **return** **new** Square();  }  **return** **null**;  }  //这是工厂，所以颜色不是在这个工厂实现  @Override  **public** Color getColor(String color) {  **return** **null**;  }    } |
| 形状接口：  //创建一个形状接口。  **public** **interface** Shape {  **void** draw();  } |
| **形状1：**  **public** **class** Square **implements** Shape{  @Override  **public** **void** draw() {  System.***out***.println("Inside Square::draw() method.");  }  } |
| **形状2**  **public** **class** Rectangle **implements** Shape{  @Override  **public** **void** draw() {  System.***out***.println("Inside Rectangle::draw() method.");  }  } |
| **形状3**  **public** **class** Circle **implements** Shape{  @Override  **public** **void** draw() {  System.***out***.println("Inside Circle::draw() method.");  }  } |

颜色工厂：

|  |
| --- |
| //颜色（子工厂）  **public** **class** ColorFactory **extends** AbstractFactory {  // 这是颜色工厂，所以形状不属于在这个工厂实现  @Override  **public** Shape getShape(String shape) {  **return** **null**;  }  //颜色创建  @Override  **public** Color getColor(String color) {  **if** (color == **null**) {  **return** **null**;  }  **if** (color.equalsIgnoreCase("RED")) {  **return** **new** Red();  } **else** **if** (color.equalsIgnoreCase("GREEN")) {  **return** **new** Green();  } **else** **if** (color.equalsIgnoreCase("BLUE")) {  **return** **new** Blue();  }  **return** **null**;  }  } |
| 颜色接口：  //创建一个颜色接口  **public** **interface** Color {  **void** fill();  } |
| **颜色1：**  **public** **class** Green **implements** Color{  @Override  **public** **void** fill() {  System.***out***.println("Inside Greed::fill() method.");  }  } |
| **颜色2、**  **public** **class** Blue **implements** Color{  @Override  **public** **void** fill() {  System.***out***.println("Inside Blue::fill() method.");  }  } |
| **颜色3**  **public** **class** Red **implements** Color{  @Override  **public** **void** fill() {  System.***out***.println("Inside Red::fill() method.");  }  } |

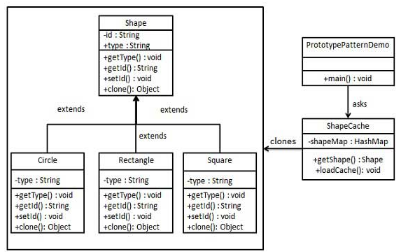
### 建造者模式





特点：套餐性

### 原型模式



抽象原型类

|  |
| --- |
| //创建一个实现了 Cloneable(可复制的，克隆的) 接口的抽象类。  //抽象原型类  **public** **abstract** **class** Shape **implements** Cloneable {  //id序号  **private** String id;  //型号  **protected** String type;    **abstract** **void** draw();  **public** String getType() {  **return** type;  }  **public** String getId() {  **return** id;  }  **public** **void** setId(String id) {  **this**.id = id;  }  //声明克隆自身  **public** Object clone() {  Object clone = **null**;  **try** {  clone = **super**.clone();  } **catch** (CloneNotSupportedException e) {  e.printStackTrace();  }  **return** clone;  }  } |

具体原型类

|  |
| --- |
| //具体原型类  **public** **class** Circle **extends** Shape {  //设置型号  **public** Circle() {  type = "Circle";  }  //打印信息  @Override  **void** draw() {  System.***out***.println("Inside Circle::draw() method.");  }  } |
| //具体原型类  **public** **class** Rectangle **extends** Shape {  //设置型号  **public** Rectangle() {  type = "Rectangle";  }  //打印信息  @Override  **void** draw() {  System.***out***.println("Inside Rectangle::draw() method.");  }  } |
| //具体原型类  **public** **class** Square **extends** Shape {  //设置型号  **public** Square() {  type = "Square";  }  //把型号打印出来  @Override  **void** draw() {  System.***out***.println("Inside Square::draw() method.");  }  } |

数据库获取实体类

|  |
| --- |
| //创建一个类，从数据库获取实体类，并把它们存储在一个 Hashtable 中。  **public** **class** ShapeCache {  //Hashtable<key, value>  **private** **static** Hashtable<String, Shape> *shapeMap* = **new** Hashtable<String, Shape>();  //获取形状方法  **public** **static** Shape getShape(String shapeId) {  //get(shapeId);获取数据的键key  Shape cachedShape = *shapeMap*.get(shapeId);  //  **return** (Shape) cachedShape.clone();  }  // 对每种形状都运行数据库查询，并创建该形状  // shapeMap.put(shapeKey, shape);  // 在数据库里添加三种形状  **public** **static** **void** loadCache() {  //实例一个形状  Circle circle = **new** Circle();  //key：setId("1");为Circle形状设置key：1  circle.setId("1");  //put(key, value);  *shapeMap*.put(circle.getId(), circle);    //实例一个形状  Square square = **new** Square();  //为Square形状设置key：2  square.setId("2");  *shapeMap*.put(square.getId(), square);    //实例一个形状  Rectangle rectangle = **new** Rectangle();  //为Rectangle形状设置key：3  rectangle.setId("3");  *shapeMap*.put(rectangle.getId(), rectangle);  }  } |

测试方法：

|  |
| --- |
| **public** **class** PrototypePatternDemo {  **public** **static** **void** main(String[] args) {  //数据库类调用数据库方法  ShapeCache.*loadCache*();    //获取到形状在数据库的键key，  //clonedShape.getType()：根据‘key’找到对应的值‘value’  Shape clonedShape = (Shape) ShapeCache.*getShape*("2");  System.***out***.println("Shape : " + clonedShape.getType());  // clonedShape.draw();    //获取到形状在数据库的键key，  //clonedShape.getType()：根据‘key’找到对应的值‘value’  Shape clonedShape2 = (Shape) ShapeCache.*getShape*("1");  System.***out***.println("Shape : " + clonedShape2.getType());  //获取到形状在数据库的键key，  //clonedShape.getType()：根据‘key’找到对应的值‘value’  Shape clonedShape3 = (Shape) ShapeCache.*getShape*("3");  System.***out***.println("Shape : " + clonedShape3.getType());  }  } |

## 结构型模式

### 适配器模式

参考：<https://www.cnblogs.com/V1haoge/p/6479118.html>

原理：通过继承来实现适配器功能。