# 设计模式

###### 设计模式种类

设计模式分为三种类型，共23种：

1. 创建型模式：单例模式、抽象工厂模式、原型模式、建造者模式、工厂模式。
2. 结构型模式：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。
3. 行为型模式：模版方法模式、命令模式、访问者模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式（Interpreter模式）、状态模式、策略模式、职责链模式(责任链模式)。

###### 单例模式

单例模式有八种方式：

1) 饿汉式(静态常量) 可以使用

2) 饿汉式(静态代码块) 可以使用

3) 懒汉式(线程不安全) 不可使用

4) 懒汉式(线程安全，同步方法) 不推荐使用

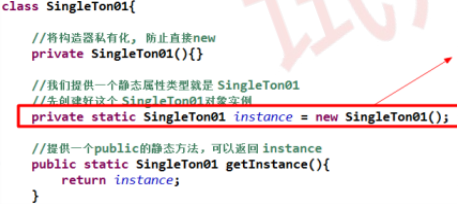
5) 懒汉式(线程安全，同步代码块) 不可使用

6) 双重检查 推荐使用

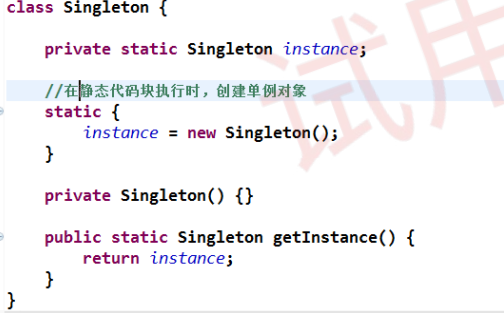
7) 静态内部类 推荐使用

8) 枚举 推荐使用

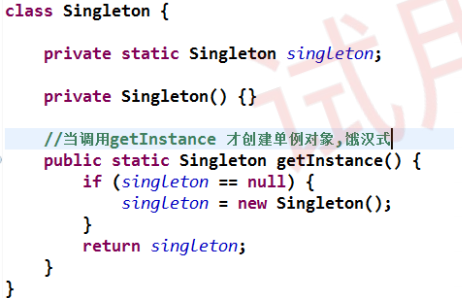
1. 饿汉式（静态常量）



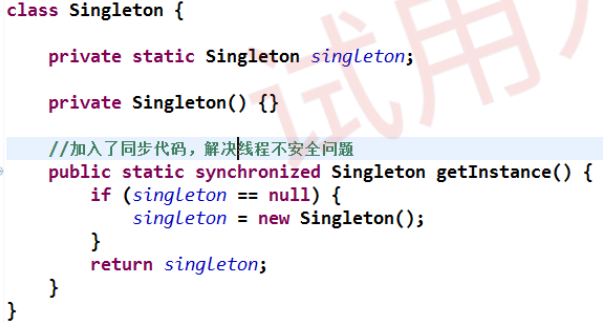
1. 饿汉式（静态代码块）



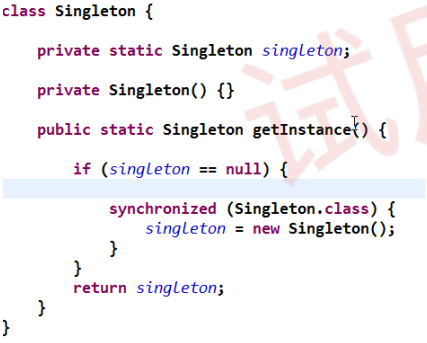
1. 懒汉式（线程不安全）



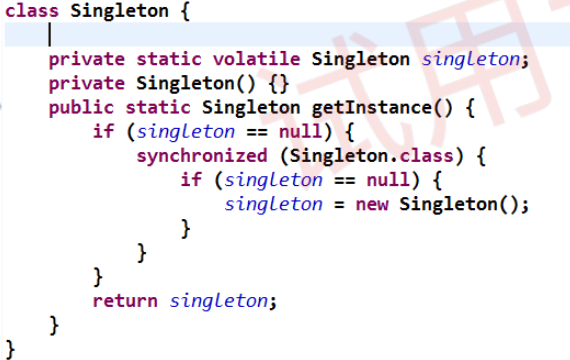
1. 懒汉式(线程安全，同步方法)



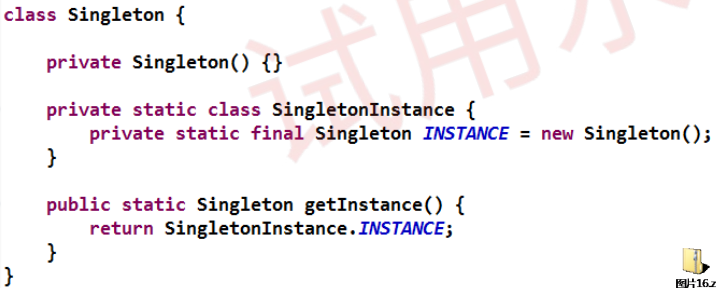
1. 懒汉式(线程安全，同步代码块)



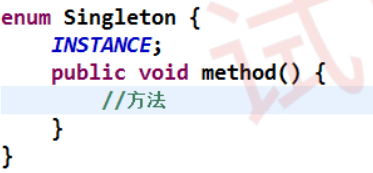
1. 双重检查



1. 静态内部类



1. 枚举



###### 工厂模式

1. 工厂方法模式
   1. **定义**：工厂方法模式定义了一个创建对象的接口（抽象工厂类或接口），但让子类决定实例化哪一个具体类（具体工厂类）。工厂方法把实例化推迟到子类。
   2. **优点**：符合开闭原则（对扩展开放，对修改封闭）。如果需要添加新的产品类，只需要增加一个新的具体工厂类，而不需要修改现有的代码。
   3. **缺点**：每增加一个产品类，就需要增加一个对应的工厂类，类的数量会增加。
   4. **使用方法**：
2. 定义一个interface或abstract的工厂类，给定一个create抽象方法
3. 定义多个实现工厂类implement该抽象类，每个实现工厂类实现create方法

此时每一个实现工厂类可以create一种产品类，但是若要在原有业务上面将产品类进行细分分类，则会导致抽象类和所有实现工厂类都改变代码。

* 1. **代码示例**：

// 先定义产品类  
public abstract class Animal {}

public class Cat extends Animal {}

public class Dog extends Animal {}

// 定义interface或abstract工厂类

public abstract class AnimalFactory {  
 public abstract Animal createAnimal();  
}

// 定义各种产品专属工厂

public class CatAnimalFactory extends AnimalFactory {  
 @Override  
 public Animal createAnimal() {  
 return new Cat();  
 }  
}

public class DogAnimalFactory extends AnimalFactory {  
 @Override  
 public Animal createAnimal() {  
 return new Dog();  
 }  
}

1. 简单工厂模式
   1. **定义**：简单工厂模式不属于经典的23种设计模式之一，但它是一个非常实用的模式，用于简化对象的创建过程。
   2. **优点**：客户端不需要知道具体的产品类，只需要知道工厂类和参数即可。
   3. **缺点**：工厂类集中了所有产品的创建逻辑，违反了单一职责原则（SRP）。每增加一个产品类，都需要修改工厂类的代码。
   4. **使用方法**：
2. 定义一个工厂类，并提供create方法，该方法要根据用户需求的不同产品返回不同产品类
3. 此时只需要一个工厂即可，但是每当多一种产品都需要对工厂进行修改。
   1. **代码示例**：

// 先定义产品类  
public abstract class Animal {}

public class Cat extends Animal {}

public class Dog extends Animal {}

// 定义工厂类

public class AnimalFactory {  
 public Animal createAnimal(String animalType) {  
 if(animalType.equals("Cat")) {  
 return new Cat();  
 } else if (animalType.equals("Dog")) {  
 return new Dog();  
 } else {  
 return null;  
 }  
 }  
}

1. 抽象工厂模式
   1. **定义**：抽象工厂模式提供了一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
   2. **优点**：符合开闭原则，可以轻松添加新的产品族（即一组相关的产品），而不需要修改现有的代码。
   3. **缺点**：如果需要添加一个新的产品类，需要修改所有工厂类，违反了开闭原则。
   4. **使用方法**：
2. 定义一个interface或abstract的工厂类，给定一套create抽象方法
3. 定义多个实现工厂类implement该抽象类，每个实现工厂类实现这一套create方法
4. 此时每一个实现工厂类可以create一套产品类，不过不同实现工厂类产生的一套产品类都有某种等级或属性差异（如男/女、高级/中级/初级），若要在原有业务上面添加一类细分分类，只需要增加一个实现工厂类即可。但如果要添加一种产品，则需要将所有工厂类进行修改。
   1. **代码示例**：

// 先定义产品类  
public abstract class Animal {}

public class Cat extends Animal {}

public class Dog extends Animal {}

public abstract class AnimalFactory {  
 public abstract Animal createDog();  
 public abstract Animal createCat();  
}

// 定义interface或abstract工厂类，提供一套create抽象方法

public class FemaleAnimalFactory extends AnimalFactory {  
 @Override  
 public Animal createDog() {  
 return null;  
 }  
 @Override  
 public Animal createCat() {  
 return null;  
 }  
}

// 根据种类或属性定义多个工厂类

public class MaleAnimalFactory extends AnimalFactory {  
 @Override  
 public Animal createDog() {  
 return new Dog();  
 }  
 @Override  
 public Animal createCat() {  
 return new Cat();  
 }  
}

###### **装饰器模式**

1. **定义**：装饰器模式是一种结构型设计模式，允许你通过将对象放入装饰类中来动态地给对象添加功能或职责。这种模式提供了一种比继承更加灵活的机制来扩展对象的功能。
2. **优点**：动态地给对象添加功能，而不影响其他对象。避免了通过继承方式进行扩展的复杂性，符合开闭原则。
3. **缺点**：装饰类可能会过多，增加系统复杂度。装饰后的对象的类型发生了变化，可能需要强制类型转换。
4. **使用方法**：

Component：定义一个接口或抽象类，表示被装饰的对象。

ConcreteComponent：实现 Component 接口的具体对象，是被装饰的具体组件。

Decorator：装饰类，也实现 Component 接口，并包含一个对 Component 的引用。Decorator 类定义了装饰类的接口，使得可以在运行时动态地给对象添加功能。

ConcreteDecorator：具体的装饰类，实现 Decorator 接口，并在其方法中添加具体的功能。

1. **代码示例**：

// Component 接口

public interface Component {

void operation();

}

// ConcreteComponent 具体组件

public class ConcreteComponent implements Component {

@Override

public void operation() {

System.out.println("ConcreteComponent operation");

}

}

// Decorator 装饰类

public abstract class Decorator implements Component {

protected Component component;

public Decorator(Component component) {

this.component = component;

}

@Override

public void operation() {

component.operation();

}

}

// ConcreteDecorator 具体装饰类

public class ConcreteDecoratorA extends Decorator {

public ConcreteDecoratorA(Component component) {

super(component);

}

public void addedBehavior() {

System.out.println("Added behavior in ConcreteDecoratorA");

}

@Override

public void operation() {

super.operation();

addedBehavior();

}

}

// 客户端代码

public class Client {

public static void main(String[] args) {

Component component = new ConcreteComponent();

component = new ConcreteDecoratorA(component);

component.operation();

}

}

###### **适配器模式**

1. **定义**：适配器模式是一种结构型设计模式，它允许你将一个类的接口转换成客户端所期望的另一种接口。这种模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
2. **优点**：提高了代码的复用性和扩展性。客户端可以透明地使用适配后的对象。
3. **缺点**：类适配器可能会导致继承层次结构变得复杂。对象适配器可能会引入额外的性能开销。
4. **使用方式**：

Target：目标接口，客户端所期望的接口。

Client：客户端，与符合 Target 接口的对象协同工作。

Adaptee：被适配的类，需要被适配的现有类，它可能有一个不同的接口。

Adapter：适配器类，它负责将 Adaptee 的接口转换成 Target 接口。

1. **分类**：

类适配器：通过多重继承的方式实现适配。

对象适配器：通过组合的方式实现适配。

1. **代码示例**：
2. 类适配器(以SpringMVC的HandlerAdapter源码为示例)

// Target 接口

public interface HandlerAdapter{

void handle(HttpServletRequest request, HttpServletResponse response, Object handler);

}

// Adaptee 被适配的类

public class HttpRequestHandler {

public void handleRequest(request, response) {

...

}

}

// Adapter 适配器类

public class HttpRequestHandlerAdapter implements HandlerAdapter{

public HttpRequestHandlerAdapter() {}

@Override

void handle(HttpServletRequest request, HttpServletResponse response, Object handler) {

(HttpRequestHandler handler).handleRequest(request, response);

}

}

// 客户端代码

public class Client {

public static void main(String[] args) {

HttpRequestHandler htttpRequestHandler = new HttpRequestHandler();

HandlerAdapter handlerAdapter= new HttpRequestHandlerAdapter();

handlerAdapter.handle(new Request(), new Response(), httpRequestHandler);

}

}

1. 对象适配器

// Target 接口

public interface Target {

void request();

}

// Adaptee 被适配的类

public class Adaptee {

public void specificRequest() {

System.out.println("Adaptee specificRequest");

}

}

// Adapter 适配器类

public class Adapter implements Target {

private Adaptee adaptee;

public Adapter(Adaptee adaptee) {

this.adaptee = adaptee;

}

@Override

public void request() {

adaptee.specificRequest();

}

}

// 客户端代码

public class Client {

public static void main(String[] args) {

Adaptee adaptee = new Adaptee();

Target target = new Adapter(adaptee);

target.request();

}

}

###### **代理模式**

1. **静态代理**
   1. **定义**：静态代理模式是一种通过静态方式创建代理对象的设计模式。在静态代理模式中，代理类和被代理类（真实主题）都实现了同一个接口，代理类在转发请求给被代理类之前或之后可以执行一些额外的操作。
   2. **优点**：可以在不修改被代理类代码的情况下，增加额外的功能。符合开闭原则（对扩展开放，对修改封闭）。
   3. **缺点**：每增加一个新的被代理类，都需要创建对应的代理类，导致类的数量成倍增加，系统复杂度提高。代理类与被代理类之间是静态绑定关系，灵活性较差。
   4. **使用方法**：
2. 定义一个接口，声明被代理类和代理类的共同方法。
3. 实现被代理类（真实主题），实现该接口。
4. 实现代理类，内部维护一个被代理类的引用，并在转发请求前后执行额外操作。
   1. **代码示例**：

// 定义接口

public interface Subject {

void request();

}

// 被代理类（真实主题）

public class RealSubject implements Subject {

@Override

public void request() {

System.out.println("RealSubject: Handling request.");

}

}

// 代理类

public class StaticProxy implements Subject {

private RealSubject realSubject;

public StaticProxy(RealSubject realSubject) {

this.realSubject = realSubject;

}

@Override

public void request() {

preRequest();

realSubject.request();

postRequest();

}

private void preRequest() {

System.out.println("StaticProxy: Do something before request.");

}

private void postRequest() {

System.out.println("StaticProxy: Do something after request.");

}

}

// 客户端代码

public class Client {

public static void main(String[] args) {

RealSubject realSubject = new RealSubject();

StaticProxy proxy = new StaticProxy(realSubject);

proxy.request();

}

}

1. **动态代理**
   1. **定义：**动态代理模式允许在运行时动态地创建代理对象，而无需在编译时指定具体的代理类。Java提供了内置的动态代理机制，通过java.lang.reflect.Proxy类和InvocationHandler接口实现。
   2. **优点：**可以在运行时动态地创建代理对象，灵活性高。无需为每个被代理类创建对应的代理类，减少了类的数量，降低了系统复杂度。
   3. **缺点：**动态代理的实现相对复杂，需要理解反射机制。动态代理在每次方法调用时都会有一定的性能开销。
   4. **使用方法：**
2. 定义一个接口，声明被代理类的方法。
3. 实现被代理类（真实主题），实现该接口。
4. 实现动态代理处理器InvocationHandler接口，重写invoke方法，在方法中实现对被代理类方法的调用以及额外操作。
5. 使用Proxy.newProxyInstance方法动态创建代理对象。

Public static Object newProxyInstance (ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

throws IllegalArgumentException { ... }

loader :类加载器，用于加载代理对象。

interfaces : 被代理类实现的一些接口。

h : 实现了 InvocationHandler 接口的对象。

* 1. **代码示例：**

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

// 定义接口

public interface Subject {

void request();

}

// 被代理类（真实主题）

public class RealSubject implements Subject {

@Override

public void request() {

System.out.println("RealSubject: Handling request.");

}

}

// 动态代理处理器

public class DynamicInvocationHandler implements InvocationHandler {

private Object realSubject;

public DynamicInvocationHandler(Object realSubject) {

this.realSubject = realSubject;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("DynamicProxy: Do something before request.");

Object result = method.invoke(realSubject, args);

System.out.println("DynamicProxy: Do something after request.");

return result;

}

}

// 客户端代码

public class Client {

public static void main(String[] args) {

RealSubject realSubject = new RealSubject();

Subject proxyInstance = (Subject) Proxy.newProxyInstance(

Subject.class.getClassLoader(),

new Class[]{Subject.class},

new DynamicInvocationHandler(realSubject)

);

proxyInstance.request();

}

}

###### **模板方法模式**

1. **定义**

模板方法模式是一种行为设计模式，它在方法中定义了算法的框架，将一些步骤延迟到子类中实现。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

1. **优点**

封装不变部分：将算法的不变部分封装在父类中，将可变部分留给子类实现，符合开闭原则。

便于维护和扩展：算法的结构在父类中定义，子类只需实现具体的步骤，便于维护和扩展。

避免重复代码：减少重复代码，提高代码复用性。

1. **缺点**

灵活性受限：子类的执行步骤必须符合父类定义的模板方法的顺序，灵活性受到一定限制。

可能增加类的数量：如果算法的步骤较多，可能需要定义多个抽象方法，导致子类数量增加。

1. **使用方法**
2. 定义一个抽象类（模板类），声明模板方法和抽象方法。
3. 在模板方法中定义算法的框架，调用抽象方法。
4. 创建具体子类，实现抽象方法，完成具体的算法步骤。
5. **代码示例**

// 模板类

public class Thread implement Runnable{

// 模板方法，定义算法框架

public synchronized void start() {  
 if (threadStatus != 0)  
 throw new IllegalThreadStateException();  
 group.add(this);  
 boolean started = false;  
 try {  
 start0(); //start0这个JNI方法底层调用了run()  
 started = true;  
 } finally {  
 try {  
 if (!started) {  
 group.threadStartFailed(this);  
 }  
 } catch (Throwable ignore) {  
 /\* do nothing. If start0 threw a Throwable then  
 it will be passed up the call stack \*/  
 }  
 }  
}

// 假设Thread用无参构造器，此时target为NULL，因此run()为空方法，待子类实现

@Override  
 public void run() {  
 if (target != null) {  
 target.run();  
 }  
 }

}

// 具体子类

public class MyThread1 extends Thread {

@Override

protected void step1() {

System.out.println("ConcreteClassA: Step 1");

}

@Override

protected void step2() {

System.out.println("ConcreteClassA: Step 2");

}

@Override

protected void step3() {

System.out.println("ConcreteClassA: Step 3");

}

}

public class MyThread2 extends Thread{

@Override

protected void step1() {

System.out.println("ConcreteClassB: Step 1");

}

@Override

protected void step2() {

System.out.println("ConcreteClassB: Step 2");

}

@Override

protected void step3() {

System.out.println("ConcreteClassB: Step 3");

}

}

// 客户端代码

public class Client {

public static void main(String[] args) {

TemplateClass classA = new ConcreteClassA();

classA.templateMethod();

TemplateClass classB = new ConcreteClassB();

classB.templateMethod();

}

}

###### **策略模式**

1. **定义**

策略模式是一种行为设计模式，它定义了一系列算法，并将每个算法封装起来，使它们可以互换。策略模式让算法的变化独立于使用算法的客户。

1. **优点**

算法可互换：算法可以独立于客户端进行修改和扩展，客户端无需修改代码。

符合开闭原则：新增算法时，只需新增策略类，无需修改现有代码。

易于理解和实现：策略模式结构清晰，易于理解和实现。

1. **缺点**

类的数量可能增加：每种算法都需要一个策略类，可能导致类的数量过多。

客户端需要了解策略类：客户端需要知道所有策略类的细节，增加了客户端的复杂性。

1. **使用方法**
2. 定义一个策略接口，声明算法的公共方法。
3. 实现具体的策略类，实现策略接口。
4. 定义上下文类，维护一个策略接口的引用，并提供方法供客户端调用。
5. 客户端通过上下文类使用策略。
6. **代码示例**

// 策略接口

public interface Strategy {

void execute(int a, int b);

}

// 具体策略类

public class AddStrategy implements Strategy {

@Override

public void execute(int a, int b) {

System.out.println(a + b);

}

}

public class SubtractStrategy implements Strategy {

@Override

public void execute(int a, int b) {

System.out.println(a - b);

}

}

// 上下文类

public class Context {

private Strategy strategy;

public Context(Strategy strategy) {

this.strategy = strategy;

}

public void setStrategy(Strategy strategy) {

this.strategy = strategy;

}

public void executeStrategy(int a, int b) {

strategy.execute(a, b);

}

}

// 客户端代码

public class Client {

public static void main(String[] args) {

Context context = new Context(new AddStrategy());

context.executeStrategy(10, 5);

context.setStrategy(new SubtractStrategy());

context.executeStrategy(10, 5);

}

}