### 单例模式

**保证一个类仅有一个实例，并提供一个访问他的全局访问点**。

实现步骤：

（1）将构造方法设为私有。

（2）提供一个公有静态方法，返回对象实例。

饿汉模式：

|  |
| --- |
| public class BasicSingleTon {  private static BasicSingleTon instance = new BasicSingleTon();    private BasicSingleTon() {}    public static BasicSingleTon getInstance() {  return instance;  }  } |

缺点：在虚拟机加载该类时会初始化类的静态变量，即创建类的实例，影响性能。

懒汉模式：

|  |
| --- |
| public class LazySingleTon {  private static LazySingleTon instance;    private LazySingleTon() {}    public static LazySingleTon getInstance() {  if (instance == null) {  instance = new LazySingleTon();  }  return instance;  }  } |

优点：在调用静态方法时，才创建实例对象，提高系统性能。

缺点：线程不安全。

---线程安全----

线程安全的单例类

|  |
| --- |
| public class SyncSingleTon {  private static SyncSingleTon instance;    private SyncSingleTon() {}    public static synchronized SyncSingleTon getInstance() {  if (instance == null) {  instance = new SyncSingleTon();  }  return instance;  }  } |

缺点：锁的粒度太粗导致性能低，改造为如下

双重非空判断（并非正真安全）

|  |
| --- |
| public class SyncSingleTon {  private static SyncSingleTon instance;    private SyncSingleTon() {}    public static SyncSingleTon getInstance() {  if (instance == null) {  synchronized(SyncSingleTon.class) {  if (instance == null) {  instance = new SyncSingleTon();  }  }  }  return instance;  }  } |

缺点：其实这并不是线程安全的单例类，可能会在instance == new SyncSingleTon时发生指令重排，需要在instance加上volatile字段，防止初始化instance是jvm发生指令重排。

**指令重排**：jvm并不一定总是按照我们的编码顺序执行代码，它会在“不改变”原有代码语句含义下进行代码指令的重排序。（生成机器指令和字节码指令顺序不同）。

|  |
| --- |
| instance = new SyncSingleTon()，代码执行分为三步   1. 分配内存空间 2. 对象初始化 3. instance指向对象的内存地址   在指令重排的情况下顺序可能变为（1）（3）（2），这样在上面的双重非空判断就可能在多个线程并发时出现错误。  此时的线程B将会返回Null。可以在静态变量前加上volatile关键字，避免在初始化instance对象时发生指令重排。 |

双重非空线程安全类

|  |
| --- |
| public class SyncSingleTon {  // 使用volatile关键字保证这个对象创建时不会发生指令重排  private static volatile SyncSingleTon instance;    private SyncSingleTon() {}    public static SyncSingleTon getInstance() {  if (instance == null) {  synchronized(SyncSingleTon.class) {  if (instance == null) {  instance = new SyncSingleTon();  }  }  }  return instance;  }  } |

Enum就是单例的

应用场景：

1. 创建对象耗时过多或消耗资源过多，但又经常使用的，比如数据库连接池。
2. 需要生成唯一序列的环境，比如网站的计数器。
3. 频繁创建和销毁的对象，可以提高系统性能。

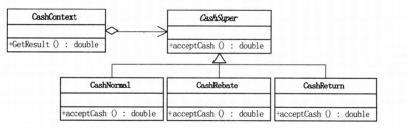
### 策略模式

策略模式：定义了算法家族，它们之间可以相互替换，此模式让算法的变化，不会影响使用算法的客户。

步骤：

1. 创建算法接口
2. 具体算法类实现抽象算法接口
3. 创建算法的上下文类，属性变量指向抽象算法对象

uml图：



CashSuper算法抽象类：

|  |
| --- |
| public abstract class CashSuper {  abstract double accept(double money);  } |

CashNormal普通算法：

|  |
| --- |
| public class CashNormal extends CashSuper {  @Override  public double accept(double money) {  return money;  }  } |

CashRebate回扣算法：

|  |
| --- |
| public class CashRebate extends CashSuper {  private double rebate;  public CashRebate(double rebate) {  this.rebate = rebate;  }  @Override  public double accept(double money) {  return money \* rebate;  }  } |

CashContext上下文类：

|  |
| --- |
| public class CashContext {  private CashSuper cashSuper;  public CashContext(CashSuper cashSuper) {  this.cashSuper = cashSuper;  }    public double accept(double money) {  return cashSuper.accept(money);  }  } |

客户端：

|  |
| --- |
| public static void main(String[] args) {  CashNormal cashNormal = new CashNormal();  CashContext normalContext = new CashContext(cashNormal);  System.out.println(normalContext.accept(100));    CashRebate cashRebate = new CashRebate(0.6);  CashContext rebateContext = new CashContext(cashRebate);  System.out.println(rebateContext.accept(100));  } |

改进：还可以结合简单工厂模式，在策略的上下文类的构造方法中生成具体的策略类，来减轻客户端的复杂度。

应用场景：

1、系统容错。比如记录日志，首先尝试将日志内容存入数据库，如果失败就保存到文件中。

### 代理模式

静态代理：



|  |
| --- |
| // 代理类和被代理类的公共接口  public abstract class Subject {  public abstract void request();  }  // 委托类  public class RealSubject extends Subject {  @Override  public void request() {  System.out.println("被代理对象执行的方法");  }  }  // 代理类  public class Proxy extends Subject {  private RealSubject realSubject;  public Proxy(RealSubject realSubject) {  this.realSubject = realSubject;  }  @Override  public void request() {  realSubject.request();  }  }  // 客户端  public static void main(String[] args) {  Proxy proxy = new Proxy(new RealSubject());  proxy.request();  } |

使用场景：

（1）远程代理

特点：被代理对象固定，在编译期生成代理对象，适用于被代理对象较少的场景。

**动态代理**：运行时创建代理对象

实现方式：

1. jdk反射针对接口实现代理；

缺陷：委托类必须实现接口；委托类增加了接口不存在的方法，这个方法不能被代理

1. cglib采用底层字节码技术，针对类实现代理

通过**反射方式**实现动态代理步骤：

1. 实现InvocationHandler接口创建调用处理器类。
2. 指定ClassLoader对象和一组interface创建动态代理类。
3. 通过反射机制获得动态代理类的构造函数，其唯一参数类型是调用处理器接口类型。
4. 通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数。

|  |
| --- |
| // 代理类接口  public interface Subject {  void request();  }  // 代理类  public class RealSubject implements Subject {  @Override  public void request() {  System.out.println("被代理对象执行的方法");  }  }  // 调用处理器实现类  public class DynamicProxy implements InvocationHandler {    private Object source;    public DynamicProxy(Object source) {  this.source = source;  }  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  return method.invoke(source, args);  }  // 获取动态代理类实例  public Object getProxy() {  return Proxy.newProxyInstance(source.getClass().getClassLoader(), source.getClass().getInterfaces(), this);  }  }  // 客户端  public static void main(String[] args) {  Subject proxy = (Subject) new DynamicProxy(new RealSubject()).getProxy();  proxy.request();  } |

**cglib**：高性能、高质量的Code生成库，可以在运行期扩展java类和实现java接口，通俗的说cglib可以在运行时动态生成字节码。

### 观察者模式

一个对象的状态改变可以被其观察者知道，俗称发布-订阅模式。



|  |
| --- |
| // 主题（通知者）接口  public abstract class Subject {  private List<Observer> observers = new ArrayList<>();  public void add(Observer observer) {  observers.add(observer);  }  public void remove(Observer observer) {  observers.remove(observer);  }  public void notice() {  for (Observer observer : observers) {  observer.update();  }  }  }  // 观察者接口  public abstract class Observer {  public abstract void update();  }  // 主题  public class ConcreteSubject extends Subject {  private String subjectState;  public String getSubjectState() {  return subjectState;  }  public void setSubjectState(String subjectState) {  this.subjectState = subjectState;  }  }  // 观察者  public class ConcreteObserver extends Observer {  private ConcreteSubject concreteSubject;  public ConcreteObserver(ConcreteSubject concreteSubject) {  this.concreteSubject = concreteSubject;  }  @Override  public void update() {  System.out.println("具体观察者，被观察者状态发生改变：" + concreteSubject.getSubjectState());  }  }  // 客户端  public static void main(String[] args) {  ConcreteSubject subject = new ConcreteSubject();  Observer observer = new ConcreteObserver(subject);  subject.add(observer);  subject.setSubjectState("主题1");  subject.notice();  } |

应用场景：idea点击运行按钮时，窗口（观察者）发生了改变。

不足：如上idea的窗口是不同厂家编写的，窗口没有实现同一个接口，而且同个运行按钮触发的事件，窗口的行为方式不同。所以在此种状况下可以使用**事件委托**（需自己实现）：不需要主题接口和观察者接口，只需要在客户端调用更新方法。

### 装饰模式

装饰模式：动态的给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更为灵活。



|  |
| --- |
| // 要增加职责对象的接口  public abstract class Component {  public abstract void operation();  }  // 具体对象  public class ConcreteComponent extends Component {  @Override  public void operation() {  System.out.println("具体操作");  }  }  // 装饰器  public abstract class Decorator extends Component {  protected Component component;  public Decorator(Component component) {  this.component = component;  }  }  // 具体装饰器A  public class ConcreteDecoratorA extends Decorator {  public ConcreteDecoratorA(Component component) {  super(component);  }  public void methodA() {  System.out.println("装饰器A提供的方法");  }  @Override  public void operation() {  System.out.println("装饰器A修饰操作");  component.operation();  System.out.println("装饰器A修饰完毕");  }  }  // 具体装饰器B  public class ConcreteDecoratorB extends Decorator {  public ConcreteDecoratorB(Component component) {  super(component);  }  public void methodB() {  System.out.println("装饰器B提供的方法");  }  @Override  public void operation() {  System.out.println("装饰器B修饰");  component.operation();  System.out.println("装饰器B修饰完毕");  }  }  //客户端  public static void main(String[] args) {  Component component = new ConcreteComponent();  component.operation();  ConcreteDecoratorA concreteDecoratorA = new ConcreteDecoratorA(component);  concreteDecoratorA.operation();  ConcreteDecoratorB concreteDecoratorB = new ConcreteDecoratorB(component);  concreteDecoratorB.operation();  // 装饰器也可以被修饰  concreteDecoratorB = new ConcreteDecoratorB(concreteDecoratorA);  concreteDecoratorB.operation();  } |

装饰器类也可以简化：

1. Component接口可以不用，让Decorator直接继承ConcreteComponent
2. 只有一个装饰器可以删除Decorator

应用：

InputStream就是使用装饰器



### 适配器模式

适配器模式：兼容老的接口，复用代码。一般用在后期补救

分为两种：

1. 类适配器。采用类继承的方式，但是java是单继承，推荐使用对象适配器。
2. 对象适配器。采用对象组合的方式。

类适配器：



|  |
| --- |
| // Target目标接口  public interface Target {  void request();  }  // 被适配的类  public class Adaptee {  public void special() {  System.out.println("特别的方法");  }  }  // 适配器  public class Adapter extends Adaptee implements Target {  @Override  public void request() {  super.special();  }  }  // 客户端  public static void main(String[] args) {  Target target = new Adapter();  target.request();  } |

对象适配器



|  |
| --- |
| // 目标接口  public interface Target {  void request();  }  // 被适配类  public class Adaptee {  public void special() {  System.out.println("特别的方法");  }  }  // 适配类  public class Adapter extends Adaptee implements Target {  @Override  public void request() {  super.special();  }  }  // 客户端  public static void main(String[] args) {  Target target = new Adapter();  target.request();  } |

### 命令模式

**命令模式**：将一个请求封装为一个对象，从而可以对不同请求进行参数化；支持请求排队、记录请求日志和撤销请求。

解耦命令调用者和命令执行者



|  |
| --- |
| // 请求调用者  public class Invoker {  private Command command;  public void setCommand(Command command) {  this.command = command;  }  public void executeCommand() {  command.execute();  }  }  // 命令抽象  public abstract class Command {  protected Receiver receiver;  public Command(Receiver receiver) {  this.receiver = receiver;  }  public abstract void execute();  }  // 具体命令  public class ConcreteCommand extends Command {  public ConcreteCommand(Receiver receiver) {  super(receiver);  }  @Override  public void execute() {  receiver.action();  }  }  // 命令执行者  public class Receiver {  public void action() {  System.out.println("执行请求");  }  }  // 客户端  public static void main(String[] args) {  Invoker invoker = new Invoker();  Command command = new ConcreteCommand(new Receiver());  invoker.setCommand(command);  invoker.executeCommand();  } |

### 组合模式

组合模式，将对象组合成树形结构以表示“部分-整体”的层次结构。使得用户对单个对象和组合对象的使用具有一致性。



|  |
| --- |
| // 组合接口  public abstract class Component {  protected String name;  public abstract void add(Component component);  public abstract void remove(Component component);  public abstract void display(int depth);  }  // 单对象  public class Leaf extends Component {    public Leaf(String name) {  this.name = name;  }    @Override  public void add(Component component) {  throw new UnsupportedOperationException();  }  @Override  public void remove(Component component) {  throw new UnsupportedOperationException();  }  @Override  public void display(int depth) {  for (int i = 0; i < depth; i++) {  System.out.print("-");  }  System.out.println(name);  }  }  // 组合对象  public class Composite extends Component {    private List<Component> list;    public Composite(String name) {  this.name = name;  list = new ArrayList<>();  }  @Override  public void add(Component component) {  list.add(component);  }  @Override  public void remove(Component component) {  for (Component item : list) {  if (component.name.equals(item.name)) {  list.remove(item);  }  }  }  @Override  public void display(int depth) {  System.out.println(name);  for (Component component : list) {  component.display(depth + 2);  }  }  }  // 客户端  public static void main(String[] args) {  Leaf leaf = new Leaf("b");  Composite composite = new Composite("a");  composite.add(leaf);  composite.display(0);  } |

安全性VS可见性

可见性：单个对象和组合对象的行为都是一样的，尽管单个对象有的行为是禁止的。

安全性：单个对象不具备组合对象的某些特定行为，所以在接口中不应该声明出来，这不利于安全性。

这两者是矛盾的，现实中更倾向于可见性。

### 简单工厂模式



|  |
| --- |
| // 产品抽象  public abstract class Product {  public abstract void method();  }  // 具体产品A  public class ProductA extends Product {  @Override  public void method() {  System.out.println("产品A");  }  }  // 具体产品B  public class ProductB extends Product {  @Override  public void method() {  System.out.println("产品B");  }  }  // 工厂类  public class Factory {  public static Product createProduct(String name) {  switch (name) {  case "a": return new ProductA();  case "b": return new ProductB();  default:  return null;  }  }  }  // 客户端  public static void main(String[] args) {  Product productA = Factory.createProduct("a");  productA.method();  } |

优点：减少客户端与实现类之间的耦合。

缺点：增加一个操作类需要在工厂类中添加判断，违背了开闭原则，所以出现了工厂方法模式。

**工厂方法模式**

工厂方法模式：定义了用于创建对象接口，让其子类决定实例化哪个类。



|  |
| --- |
| // 抽象产品类，定义同上  public abstract class Product  // 产品A  public class ProductA extends Product  // 产品B  public class ProductB extends Product  // 工厂抽象类  public abstract class Factory {  public abstract Product createProduct();  }  // 产品A工厂类  public class ProductAFactory extends Factory {  @Override  public Product createProduct() {  return new ProductA();  }  }  // 产品B工厂  public class ProductBFactory extends Factory {  @Override  public Product createProduct() {  return new ProductB();  }  }  // 客户端  public static void main(String[] args) {  Product productA = new ProductAFactory().createProduct();  productA.method();  } |

优点：相比简单工厂模式，增加操作类，不需要去在工厂类中增加判断逻辑，符合开闭原则。

缺点：增加操作类，就要同时增加操作工厂类；同时增加了客户端的逻辑。

工厂方法模板只能创建一系列的类，如果想创建多个系列的类，可以选择抽象工厂模式。

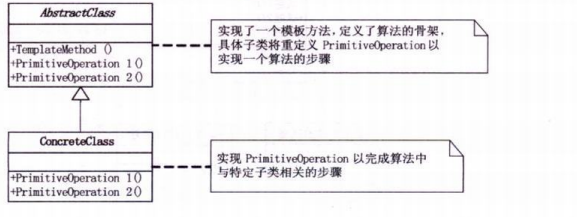
**抽象工厂模式**



|  |
| --- |
| // 抽象产品1  public abstract class Product1  // 抽象产品1实现类A  public class Product1A extends Product1  // 抽象产品1实现类B  public class Product1B extends Product1  // 抽象产品2  public abstract class Product2  // 抽象产品2实现类A  public class Product2A extends Product2  // 抽象产品2实现类B  public class Product2B extends Product2  // 工厂抽象类  public abstract class Factory {  public abstract Product1 product1();  public abstract Product2 product2();  }  // 产品A工厂类  public class ProductAFactory extends Factory {  @Override  public Product1 product1() {  return new Product1A();  }  @Override  public Product2 product2() {  return new Product2A();  }  }  // 产品B工厂类  public class ProductBFactory extends Factory {  @Override  public Product1 product1() {  return new Product1B();  }  @Override  public Product2 product2() {  return new Product2B();  }  }  // 客户端  public static void main(String[] args) {  Factory aFactory = new ProductAFactory();  aFactory.product1();  } |

### 模板方法模式

模板方法模式：定义一个操作中的算法骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。





AbstractClass父类，定义了算法骨架

|  |
| --- |
| public abstract class AbstractClass {  public void templateMethod() {  System.out.println("父类中的逻辑");  primitiveMethod();  System.out.println("父类中的逻辑");  }    abstract void primitiveMethod();  } |

ConcreteClass1子类，实现算法中的特定步骤

|  |
| --- |
| public class ConcreteClass1 extends AbstractClass {  @Override  void primitiveMethod() {  System.out.println("子类1实现的逻辑");  }  } |

ConcreteClass2，同上

|  |
| --- |
| public class ConcreteClass2 extends AbstractClass {  @Override  void primitiveMethod() {  System.out.println("子类2实现的逻辑");  }  } |

应用场景：

1. 各个子类的公共行为提取到父类中，避免代码重复