# 编程原则规范

## 单一职责原则

每个类（类级别单一职责）或方法（方法级别单一职责）应该只有一个职责，对外只提供一种功能，引起类或方法变化的原因应该只有一个。

只有逻辑足够简单，才可以在类级别上违反单一职责原则；只有类中方法数量足够少，才可以在方法级别上违反单一职责原则；

单一职责原则优点：

1、降低复杂度；

2、提高可读性，提高系统可维护性；

3、降低变更引起的风险。

## 里氏代换原则

里氏代换原则（Liskov Substitution Principle ）：任何父类可以出现的地方，子类一定可以出现。

里氏代换原则是继承复用的基石，只有当子类可以替换掉父类，软件单位的功能不受到影响时，父类才能真正被复用，而衍生类也能够在父类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化。而父类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：

1、子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。

2、子类中可以增加自己特有的方法。

3、当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。

4、当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

## 依赖倒转原则

依赖倒置原则是（Dependence Inversion Principle ）：开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

## 接口隔离原则

接口隔离原则（Interface Segregation Principle ）：一个接口不需要提供过多的行为，最好只提供一种对外的功能，不要把所有的操作都封装到一个接口中。

使用多个隔离的接口，比使用单个接口要好。降低类之间的耦合度，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以要降低依赖，降低耦合。

## 迪米特法则

迪米特法则（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

合成复用原则（Composite Reuse Principle ）：使用合成/聚合而不是使用继承。

## 开闭原则

开闭原则（Open Close Principle ）：对扩展开放，对修改关闭。

在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

# 设计模式（Java）

## 设计模式概览

### 创建型设计模式

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **图稿** | **业务场景** | **实现要点** |
| 工厂方法  Factory | https://bugstack.cn/assets/images/2020/design/11.png | 多种类型商品不同接口，统一发奖服务搭建场景 | 定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。 |
| 抽象工厂  Abstract  Factory | https://bugstack.cn/assets/images/2020/design/12.png | 替换Redis双集群升级，代理类抽象场景 | 提供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。 |
| 生成器 | https://bugstack.cn/assets/images/2020/design/13.png | 各项装修物料组合套餐选配场景 | 将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。 |
| 原型 | https://bugstack.cn/assets/images/2020/design/14.png | 上机考试多套试，每人题目和答案乱序排列场景 | 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。 |
| 单例 | https://bugstack.cn/assets/images/2020/design/15.png | 7种单例模式案例，Effective Java 作者推荐枚举单例模式 | 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。 |

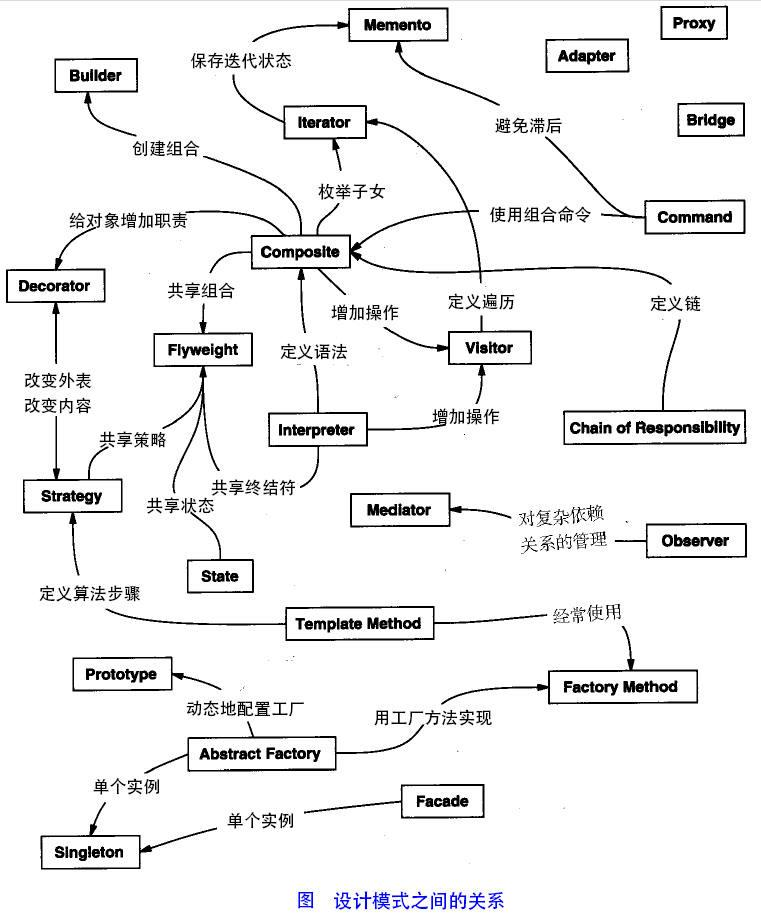
### 结构型设计模式

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **图稿** | **业务场景** | **实现要点** |
| 适配器 | https://bugstack.cn/assets/images/2020/design/21.png | 从多个MQ消息体中，抽取指定字段值场景 | 将一个类的接口转换成客户希望的另外一个接口。适配器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。 |
| 桥接 | https://bugstack.cn/assets/images/2020/design/22.png | 多支付渠道(微信、支付宝)与多支付模式(刷脸、指纹)场景 | 将抽象部分与实现部分分离，使它们都可以独立的变化。 |
| 组合 | https://bugstack.cn/assets/images/2020/design/23.png | 营销差异化人群发券，决策树引擎搭建场景 | 将对象组合成树形结构以表示”部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。 |
| 装饰 | https://bugstack.cn/assets/images/2020/design/24.png | SSO单点登录功能扩展，增加拦截用户访问方法范围场景 | 动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。 |
| 外观 | https://bugstack.cn/assets/images/2020/design/25.png | 基于SpringBoot开发门面模式中间件，统一控制接口白名单场景 | 为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。 |
| 享元 | https://bugstack.cn/assets/images/2020/design/26.png | 基于Redis秒杀，提供活动与库存信息查询场景 | 运用共享技术有效地支持大量细粒度的对象。 |
| 代理 | https://bugstack.cn/assets/images/2020/design/27.png | 模拟mybatis-spring中定义DAO接口，使用代理类方式操作数据库原理实现场景 | 为其他对象提供一种代理以控制对这个对象的访问。 |

### 行为型设计模式

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **图稿** | **业务场景** | **实现要点** |
| 责任链 | https://bugstack.cn/assets/images/2020/design/31.png | 模拟618电商大促期间，项目上线流程多级负责人审批场景 | 避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。 |
| 命令 | https://bugstack.cn/assets/images/2020/design/32.png | 模拟高档餐厅八大菜系，小二点单厨师烹饪场景 | 将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。 |
| 迭代器 | https://bugstack.cn/assets/images/2020/design/33.png | 模拟公司组织架构树结构关系，深度迭代遍历人员信息输出场景 | 提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。 |
| 中介者 | https://bugstack.cn/assets/images/2020/design/34.png | 按照Mybatis原理手写ORM框架，给JDBC方式操作数据库增加中介者场景 | 用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。 |
| 备忘录 | https://bugstack.cn/assets/images/2020/design/35.png | 模拟互联网系统上线过程中，配置文件回滚场景 | 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。 |
| 观察者 | https://bugstack.cn/assets/images/2020/design/36.png | 模拟类似小客车指标摇号过程，监听消息通知用户中签场景 | 定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。 |
| 状态 | https://bugstack.cn/assets/images/2020/design/37.png | 模拟系统营销活动，状态流程审核发布上线场景 | 允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。 |
| 策略 | https://bugstack.cn/assets/images/2020/design/38.png | 模拟多种营销类型优惠券，折扣金额计算策略场景 | 定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。 |
| 模板方法 | https://bugstack.cn/assets/images/2020/design/39.png | 模拟爬虫各类电商商品，生成营销推广海报场景 | 定义一个操作中的算法的骨架，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。 |
| 访问者 | https://bugstack.cn/assets/images/2020/design/310.png | 模拟家长与校长，对学生和老师的不同视角信息的访问场景 | 主要将数据结构与数据操作分离。 |

**参考**

* <https://refactoringguru.cn/design-patterns>
* <https://www.runoob.com/design-pattern/design-pattern-tutorial.html>
* 

## 创建型设计模式

### 工厂方法模式(Factory Method)

工厂方法模式避免创建者与具体的产品逻辑耦合、满足单一职责，每一个业务逻辑实现都在所属自己的类中完成、满足开闭原则，无需更改使用调用方就可以在程序中引入新的产品类型。

该模式的扩展性较高，增加新的子类实现较简单，且隐蔽了具体的子类实现过程。

缺点在于增加了系统的复杂度，增加了子类的个数。

|  |
| --- |
| public interface ISender {  void send(); } |

|  |
| --- |
| public class MailSender implements ISender {  @Override  public void send() {  System.*err*.println("this is mailSender");  } } |

|  |
| --- |
| public class SmsSender implements ISender {  @Override  public void send() {  System.*err*.println("this is mailSender");  } } |

|  |
| --- |
| public class SendFactory<T extends ISender> {  public static String *MAIL* = "mail";  public static String *SMS* = "sms";  public static <T extends ISender> SendFactory<T> build() {  return new SendFactory<>();  }  ... } |

#### if-else分支结构

通过if-else分支结构来对输入的内容进行判断，返回不同的类的对象，适用于通过参数来决定创建的对象。

|  |
| --- |
| public ISender produce(String type) {  if (*MAIL*.equals(type)) {  return new MailSender();  } else if (*SMS*.equals(type)) {  return new SmsSender();  } else {  System.*err*.println("please input correct type!");  return null;  }  } |

#### 普通方法

通过多个方法来返回不同类的对象，各个方法之间不会冲突，从而创建多个类的对象。

|  |
| --- |
| public ISender produceMail() {  return new MailSender();  }   public ISender produceSms() {  return new SmsSender();  } |

#### 静态方法

通过多个静态方法来返回不同类的对象。

|  |
| --- |
| public static ISender produceStaticMail() {  return new MailSender();  }   public static ISender produceStaticSms() {  return new SmsSender();  } |

### 抽象工厂模式(Abstract Factory Pattern)

工厂方法模式类的创建依赖工厂类，要拓展程序，必须对工厂类进行修改，违背了闭包原则。

使用抽象工厂模式，创建多个工厂类，需要增加新的功能时，直接增加新的工厂类而不需要修改之前的代码。

例如发送短信功能，此时使用抽象工厂模式，便可以更方便的新增或切换短信服务。

该模式的优点在于新增产品时便于扩展，不用修改现有的产品类，符合单一职责和开闭原则；缺点在于新增产品时需要编写的代码量较多，既要实现该产品的工厂类，又要实现该产品。

抽象工厂模式还可以和接口的适配器模式结合，通过抽象类的方式来作为中间层，避免实现接口的所有方法。

|  |
| --- |
| */\*\* \* 创建接口，使工厂类实现该接口*  *\* 需要增加新的产品时，新增该产品的工厂类实现该接口即可，即可在产品族中获取不同产品*  *\*/*  public interface Provider {  public Sender produce(); }  public class SendSmsFactory implements Provider{  public static SendSmsFactory build() {  return new SendSmsFactory();  }  @Override  public ISender produce() {  return new SmsSender();  } }  public class SendMailFactory implements Provider{  public static SendMailFactory build() {  return new SendMailFactory();  }  @Override  public ISender produce() {  return new MailSender();  } } |

### 单例模式(Singleton)

单例模式要私有化构造器，在类中创建私有实例并提供公有方法给外部类获取该实例对象，确保只会存在唯一的一个实例。

单例模式优点在于在内存里只有一个实例，减少了内存的开销，尤其是在频繁的创建和销毁实例场景下，避免对资源的多重占用；缺点在于没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

单例模式分为**懒汉式**（线程不安全，调用效率高，但是不能延时加载）、**饿汉式**（线程安全，调用效率不高，可以延时加载）、**双重检测锁式**（由于JVM底层内部模型原因，偶尔会出问题，不建议使用）**、静态内部类式**（线程安全，调用效率高，可以延时加载）**和枚举单例**（线程安全，调用效率高，不能延时加载）。

#### 懒汉式

线程不安全的懒汉式

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

这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。

线程安全的懒汉式

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

这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。

优点：第一次调用才初始化，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。

#### 饿汉式

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

这种方式比较常用，但容易产生垃圾对象。

优点：没有加锁，执行效率会提高。

缺点：类加载时就初始化，浪费内存。

它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

#### 双检锁式

|  |
| --- |
| */\*\*  \* 双检锁懒汉式单例模式  \* 由于JVM底层内部模型原因，偶尔会出问题，不建议使用  \*/*  public class Singleton {private static Singleton *instance* = null;private Singleton() {}  */\*\*  \* 静态工程方法，创建实例，此时毫无线程安全保护，因此要添加synchronized  \** ***@return*** *单例对象  \*/*  public static Singleton getInstance() {  if(*instance* == null) {  synchronized(*instance*) {  if(*instance* == null) {  *instance* = new Singleton();  }  }  }  return *instance*;  } } |

这种方式采用双检锁机制（DCL，即 double-checked locking），安全且在多线程情况下能保持高性能。

#### 静态内部类式

实际情况是，单例模式使用内部类来维护单例的实现，JVM 内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用getInstance的时候，JVM 能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕， 这样我们就不用担心上面的问题。 同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。

|  |
| --- |
| */\*\*  \* 通过静态内部类的方式来实现单例模式  \* 线程安全，调用效率高，可以延时加载  \*/*  public class Singleton {  private Singleton () {  }   */\*\*  \* 使用一个内部类来维护单例  \*/* private static class SingletonFactory{  private static Singleton *instance* = new Singleton();  }  public static Singleton getInstance() {  return SingletonFactory.*instance*;  } } |

这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。

这种方式同样利用了 classloader 机制来保证初始化 instance 时只有一个线程，它跟饿汉式不同的是：饿汉式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有通过显式调用 getInstance 方法时，才会显式装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比饿汉式就显得很合理。

如果在构造函数中抛出异常， 实例将永远得不到创建， 也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加synchronized关键字，也是可以的。考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。

|  |
| --- |
| public class Singleton {  private static Singleton *instance* = null;  private Singleton() {  }  public static synchronized void syncInit() {  if(*instance* == null) {  *instance* = new Singleton();  }  }  public static Singleton getInstance() {  if(*instance* == null) {  *syncInit*();  }  return *instance*;  } } |

类的静态方法也能实现单例模式，两者的不同处：

静态类不能实现接口，因为接口中不允许有static修饰的方法

单例可以被延迟（比较庞大的类延迟加载有助于提升性能）初始化，静态类一般在第一次加载时初始化。

单例类比较灵活，可以实现一些其它功能，静态类不行。

前面的单例模式的实现，内部就是一个静态类实现的，两者思想的结合，才能造就出完美的解决方案。

#### 枚举式

|  |
| --- |
| public enum Singleton {  *INSTANCE*; } |

这种方式是实现单例模式的最佳方法。它更简洁，不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象。

### 建造者模式(Builder)

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

**意图**：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

**主要解决**：主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

**何时使用**：一些基本部件不会变，而其组合经常变化的时候。

**如何解决**：将变与不变分离开。

**关键代码**：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

**优点**：

1、建造者独立，易扩展。

2、便于控制细节风险。

**缺点**：

1、产品必须有共同点，范围有限制。

2、如内部变化复杂，会有很多的建造类。

**使用场景**：

1、需要生成的对象具有复杂的内部结构。

2、需要生成的对象内部属性本身相互依赖。

**注意事项**：

与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。

|  |
| --- |
| public interface Item {  String name();  Packing packing();  float price(); }  public interface Packing {  String pack(); } |

|  |
| --- |
| public abstract class BaseBurger implements Item {  @Override  public Packing packing() { return new Wrapper(); }  @Override  public abstract float price(); }  public abstract class BaseColdDrink implements Item {  @Override  public Packing packing() { return new Bottle(); }  @Override  public abstract float price(); }  public class Bottle implements Packing {  @Override  public String pack() { return "Bottle"; } }  public class Wrapper implements Packing {  @Override  public String pack() { return "Wrapper"; } } |

|  |
| --- |
| public class ChickenBurger extends BaseBurger {  @Override  public float price() { return 50.5f; }  @Override  public String name() { return "Chicken Burger"; } }  public class VegBurger extends BaseBurger {  @Override  public float price() { return 25.0f; }  @Override  public String name() { return "Veg Burger"; } }  public class Coke extends ColdDrink {  @Override  public float price() { return 30.0f; }  @Override  public String name() { return "Coke"; } }  public class Pepsi extends BaseColdDrink {  @Override  public float price() { return 35.0f; }  @Override  public String name() { return "Pepsi"; } } |

|  |
| --- |
| public class Meal {  private List<Item> items = new ArrayList<>();  public void addItem(Item item){  items.add(item);  }  public float getCost(){  float cost = 0.0f;  for (Item item : items) {  cost += item.price();  }  return cost;  }  public void showItems(){  for (Item item : items) {  System.*out*.print("Item : "+item.name());  System.*out*.print(", Packing : "+item.packing().pack());  System.*out*.println(", Price : "+item.price());  }  } } |

|  |
| --- |
| public class MealBuilder {  public Meal prepareVegMeal (){  Meal meal = new Meal();  meal.addItem(new VegBurger());  meal.addItem(new Coke());  return meal;  }  public Meal prepareNonVegMeal (){  Meal meal = new Meal();  meal.addItem(new ChickenBurger());  meal.addItem(new Pepsi());  return meal;  }   public static void main(String[] args) {  MealBuilder mealBuilder = new MealBuilder();   Meal vegMeal = mealBuilder.prepareVegMeal();  System.*out*.println("Veg Meal");  vegMeal.showItems();  System.*out*.println("Total Cost: " +vegMeal.getCost());   Meal nonVegMeal = mealBuilder.prepareNonVegMeal();  System.*out*.println("Non-Veg Meal");  nonVegMeal.showItems();  System.*out*.println("Total Cost: " +nonVegMeal.getCost());  } } |

从这点看出，建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工程模式的区别就是： 工厂模式关注的是创建单个产品， 而建造者模式则关注创建复合对象，多个部分。因此，是选择工厂模式还是建造者模式，依实际情况而定。

### 原型模式(Prototype)

用原型实例指定创建对象的种类，并且通过拷贝这些原型来创建新的对象。

原型模式虽然是创建型的模式，但是与工程模式没有关系，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。

一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为 Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者 cloneB,因为此处的重点是 super.clone()这句话， super.clone()调用的是 Object的 clone()方法，而在 Object 类中，clone()是 native 的。

对象深、浅复制的概念：

浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。

简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

**意图：**

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

**主要解决：**

在运行期建立和删除原型。

**何时使用：**

1、当一个系统应该独立于它的产品创建，构成和表示时。

2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。

3、为了避免创建一个与产品类层次平行的工厂类层次时。

4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

**如何解决：**

利用已有的一个原型对象，快速地生成和原型对象一样的实例。

**关键代码：**

1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()，在 .NET 中可以使用 Object 类的 MemberwiseClone() 方法来实现对象的浅拷贝或通过序列化的方式来实现深拷贝。 2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有稳定的接口。

**应用实例：**

1、细胞分裂。

2、JAVA 中的 Object clone() 方法。

**优点：**

1、性能提高。

2、逃避构造函数的约束。

**缺点：**

1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

2、必须实现 Cloneable 接口。

**使用场景：**

1、资源优化场景。

2、类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。

3、性能和安全要求的场景。

4、通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

5、一个对象多个修改者的场景。

6、一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

7、在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与 Java 融为浑然一体，大家可以随手拿来使用。

**注意事项：**

与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

|  |
| --- |
| public class PrototypePattern implements Cloneable, Serializable {  */\*\*  \* 不论是浅复制还是深复制，基本数据类型的变量都会重新创建  \*/* private String str = "hello";   public static void main(String[] args) throws Exception {  PrototypePattern p = new PrototypePattern();  // 浅复制的引用类型指向的还是原对象所指向的，因此和原先的一样  PrototypePattern cp1 = (PrototypePattern) p.clone();  System.*out*.println(cp1.str == p.str);  // 深复制的引用类型是重新创建的，因此和原先的不一样  PrototypePattern cp2 = (PrototypePattern) p.deepClone();  System.*out*.println(cp2.str == p.str);  }   */\*\*  \* 浅复制  \*/* @Override  public Object clone() throws CloneNotSupportedException {  return super.clone();  }   */\*\*  \* 深复制  \*/* private Object deepClone() throws IOException, ClassNotFoundException {  // 写入当前对象的二进制流  ByteArrayOutputStream bos = new ByteArrayOutputStream();  ObjectOutputStream oos = new ObjectOutputStream(bos);  oos.writeObject(this);  // 写出当前对象的二进制流  ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());  ObjectInputStream ois = new ObjectInputStream(bis);  return ois.readObject();  } } |

## 结构型设计模式

### 代理模式(Proxy)

开发中已有的方法在使用时需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应，这样违反了“**对扩展开放，对修改关闭**”的原则。

2、使用一个代理将对象包装起来, 然后用该代理对象取代原始对象，任何对原始对象的调用都要通过代理，代理对象决定是否以及何时将方法调用转到原始对象上，且对产生的结果进行控制。这种方法就是代理模式。

代理模式提供了对目标对象的间接访问方式，即通过代理对象访问目标对象。便于在目标实现的基础上增加额外的功能操作，如前拦截，后拦截等，以满足自身的业务需求，代理模式便于扩展目标对象功能。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

代理模式可以分为静态代理和动态代理：

静态代理的特征是代理类和目标对象的类都是在编译期间确定下来，不利于程序的扩展。同时，每一个代理类只能为一个接口服务，这样一来开发中必然产生过多的代理。

动态代理是指通过代理类来调用其它对象的方法，并且是在程序运行时根据需要动态创建目标类的代理对象。动态代理一般用于调试和远程方法调用。

**优点：**

1、职责清晰。

2、高扩展性。

3、智能化。

**缺点：**

1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。

2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

**使用场景：**

1、远程代理。

2、虚拟代理。

3、Copy-on-Write 代理。

4、保护（Protect or Access）代理。

5、Cache代理。

6、防火墙（Firewall）代理。

7、同步化（Synchronization）代理。

8、智能引用（Smart Reference）代理。

#### 静态代理

静态代理比较简单，代理类通过实现与目标对象相同的接口，并在类中定义一个代理对象。通过构造器塞入目标对象，赋值给代理对象，进而执行代理对象实现的接口方法，并实现前拦截，后拦截等所需的业务功能。

|  |
| --- |
| public class CalculatorProxy implements Calculate{  private Calculate calculate;  public CalculatorProxy(Calculate calculate){  this.calculate = calculate;  }  @Override  public int add(int num1, int num2) {  System.*out*.println("before");  int sum = calculate.add(num1,num2);  System.*out*.println("after");  //返回值和代理对象的该方法的返回值类型一致  return sum;  } } class Calculator implements Calculate {  public int add(int num1, int num2) {  int result = num1 + num2;  return result;  } } interface Calculate {  public int add(int num1, int num2); } |

静态代理总结：

优点：可以做到不对目标对象进行修改的前提下，对目标对象进行功能的扩展和拦截。

缺点：因为代理对象需要实现与目标对象一样的接口，会导致代理类十分繁多，不易维护，同时一旦接口增加方法，则目标对象和代理类都需要维护。静态代理这种方式每次修改和添加都需要修改大量的代码，工作量过于繁杂。

#### 动态代理

动态代理是指动态的在内存中构建代理对象（需要我们制定要代理的目标对象实现的接口类型），根据代理对象是否实现了接口而分为jdk动态代理（需要实现接口）和cglib动态代理（不需要实现接口）。

##### JDK动态代理

在Java中java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口，通过使用这个类和接口就可以生成动态代理对象。JDK提供的代理只能针对接口做代理，而cglib可以不需要接口就能完成代理的功能。

jdk动态代理，是根据目标对象的接口生成一个实现类，这个实现类实现了目标对象的所有接口。

Proxy.newProxyInstance创建的代理对象是在jvm运行时动态生成的一个对象，它并不是我们的InvocationHandler类型，也不是我们定义的那组接口的类型，而是在运行是动态生成的一个对象，并且命名方式都是这样的形式，以$开头，proxy为中，最后一个数字表示对象的标号。

每一个动态代理类都必须要实现InvocationHandler这个接口，并且每个代理类的实例都关联到了一个handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的invoke 方法来进行调用。

###### 实现步骤

1、实现接口InvocationHandler，重写invoke方法，以完成代理的具体操作。invoke三个参数分别为被代理对象、要调用的方法和方法调用时所需要的参数。

2、通过Proxy的静态方法newProxyInstance创建代理实例，其三个参数分别为被代理对象的类加载器，被代理对象实现的所有类的接口和InvovationHandler的实例。

3、通过代理实例调用被代理对象的方法。

###### 代码实例

|  |
| --- |
| public class Calculator implements Calculate {  @Override  public int add(int num1, int num2) {  int result = 0;  result = num1 + num2;  System.*out*.println("this is executed");  return result;  }  } |

|  |
| --- |
| public class JdkProxyFactory {  public static Object createJdkProxy(Object target) {  /\* newProxyInstance方法用于创建一个代理实例对象，该对象必须用接口来接收  通过jdk动态代理返回的对象，是实现了所有目标对象实现的所有接口的一个对象实例。 \*/  return Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(),  new InvocationHandler() {  /\* invoke是代理对象调用之后就会进入的方法，需要调用目标方法并在这里添加增强代码  \* invoke方法的返回值是目标方法的返回的值。\*/  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  Object result = null;  try {  LogUtils.*logBefore*(method.getName(), args);//前置扩展代码  result = method.invoke(target, args);  LogUtils.*logAfter*(method.getName(), args);//返回扩展代码  } catch (Exception e) {  e.printStackTrace();  LogUtils.*logAfterThrowing*(method.getName(), e);//异常扩展代码  }  return result;  }  });  }  public static void main(String[] args) {  Calculate calculate = new Calculator();  Calculate calculateProxy = (Calculate) *createJdkProxy*(calculate);  System.*out*.println(calculateProxy.add(66, 66));  } } |

优点：代理对象无需实现接口，免去了编写很多代理类的烦恼，同时接口增加方法也无需再维护目标对象和代理对象，只需在事件处理器中添加对方法的判断即可。

　　 缺点：代理对象不需要实现接口，但是目标对象一定要实现接口，否则无法使用JDK动态代理。

**存疑：calculateProxy的值为null**

##### Cglib动态代理

Cglib代理技术，是通过对某个具体的类型，生成一个子类。然后添加增强。

优点：在没有接口的情况下，同样可以实现代理的效果。

缺点：需要自己编码实现代理全部过程。

###### 实现步骤

1、创建增强器Enhancer对象并调用该对象的setSuperclass方法设置代理的对象

2、调用Enhancer对象的setCallback方法，通过匿名内部类的方式设置代理类MethodInterceptor

3、重写MethodInterceptor类的intercept达到代理的效果

4、调用Enhancer对象的create方法创建代理对象

###### 代码实例

|  |
| --- |
| public class CglibProxyFactory {  public static void main(String[] args) {  Calculator calculator = new Calculator();  // 调用cglib代理对象  Calculator calculatorProxy = (Calculator) *createCglibProxy*(calculator);  System.*out*.println(calculatorProxy instanceof Calculate);  System.*out*.println(calculatorProxy instanceof Calculator);  System.*out*.println(calculatorProxy.add(100, 100));  }  public static Object createCglibProxy(Object target) {  // 创建一个增强器对象==帮助我们生成Cglib代理对象  Enhancer enhancer = new Enhancer();  // 设置你对哪个类进行增强，即父类，因为Cglib是通过类进行代理，而不是接口  enhancer.setSuperclass(target.getClass());  // 设置拦截的代理方法  enhancer.setCallback(new MethodInterceptor() {  /\* intercept这个方法，就是当你调用代理对象方法的时候，会被拦截到，而调用的方法<br/>  \* 第一个参数是代理对象<br/>  \* 第二个参数是调用的方法的返回对象<br/>  \* 第三个参数是调用方法的参数<br/>  \* 第四个参数是调用方法的代理方法的反射对象<br/> \*/  @Override  public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  Object result = null;  try {  LogUtils.*logBefore*(method.getName(), args);// 前置扩展代码  result = methodProxy.invokeSuper(proxy, args); // 调用目标方法  LogUtils.*logAfter*(method.getName(), args); // 返回扩展代码  } catch (Exception e) {  LogUtils.*logAfterThrowing*(method.getName(), e); // 异常扩展代码  throw e;  }  return result;  }  });  return enhancer.create();// 创建代理对象  } } |

##### 扩展代码位置

动态代理中扩展代码根据其作用的不同可以分为四类，前置、后置、异常和返回扩展代码，其中前置和后置扩展代码不论被代理对象的方法是否发生异常都会执行，而异常代码只有当被代理对象的方法发生异常并被捕获到才会执行，返回代码只有当被代理对象的方法顺利执行完毕才会执行。

###### 前置扩展代码

前置扩展代码的位置应位于try代码块中，调用被代理对象的方法之前。

###### 后置扩展代码

后置扩展代码的位置应位于finally代码块中，在代理中，后置扩展代码不管程序运行情况如何都会执行。

###### 异常扩展代码

异常扩展代码的位置应位于catch代码块中，只有抛出异常并被catch捕获到，异常扩展代码才会执行。

###### 返回扩展代码

返回扩展代码的位置应位于try代码块中，调用被代理对象的方法之后，如果发生异常，返回扩展代码就不会执行。

### 外观模式(Facade)

为子系统中的一组接口提供一致的界面，本模式提供了一高层接口，这个接口使得子系统更容易使用。

外观模式是为了解决类与类之家的依赖关系的，像 spring 一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个 Facade 类中，降低了类类之间的耦合度，该模式中没有涉及到接口。

**意图：**

为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

**主要解决：**

降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。

**何时使用：**

1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。

2、定义系统的入口。

**如何解决：**

客户端不与系统耦合，外观类与系统耦合。

**关键代码：**

在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。

**应用实例：**

1、去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。

2、JAVA 的三层开发模式。

**优点：**

1、减少系统相互依赖。

2、提高灵活性。

3、提高了安全性。

**缺点：**

不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。

**使用场景：**

1、为复杂的模块或子系统提供外界访问的模块。

2、子系统相对独立。

3、预防低水平人员带来的风险。

**注意事项：**

在层次化结构中，可以使用外观模式定义系统中每一层的入口。

🡺 FacadePattern

|  |
| --- |
| public class FacadePattern {  public static void main(String[] args) {  Computer com = new Computer();  com.startup();  com.shutdown();  } } |

🡺 Computer

|  |
| --- |
| public class Computer {  private CPU cpu;  private Memory memory;  private Disk disk;   Computer() {  cpu = new CPU();  memory = new Memory();  disk = new Disk();  }   void startup() {  System.*out*.println("start the computer!");  cpu.startup();  memory.startup();  disk.startup();  System.*out*.println("start computer finished!");  }   void shutdown() {  System.*out*.println("begin to close the computer!");  cpu.shutdown();  memory.shutdown();  disk.shutdown();  System.*out*.println("computer closed!");  } } |

🡺 CPU

|  |
| --- |
| public class CPU{  void startup() {  System.*out*.println("cpu startup!");  }  void shutdown() {  System.*out*.println("cpu shutdown!");  } } |

🡺 Disk

|  |
| --- |
| public class Disk{  void startup() {  System.*out*.println("disk startup!");  }  void shutdown() {  System.*out*.println("disk shutdown!");  } } |

🡺 Memory

|  |
| --- |
| public class Memory{  void startup() {  System.*out*.println("memory startup!");  }  void shutdown() {  System.*out*.println("memory shutdown!");  } } |

没有 Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改。有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这就是外观模式！

### 桥接模式(Bridge)

将抽象部分与它的实现部分相分离，使他们可以独立的变化。

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是： 将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。

**意图：**

将抽象部分与实现部分分离，使它们都可以独立的变化。

**主要解决：**

在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。

**何时使用：**

实现系统可能有多个角度分类，每一种角度都可能变化。

**如何解决：**

把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。

**关键代码：**

抽象类依赖实现类。

**应用实例：**

1、猪八戒从天蓬元帅转世投胎到猪，转世投胎的机制将尘世划分为两个等级，即：灵魂和肉体，前者相当于抽象化，后者相当于实现化。生灵通过功能的委派，调用肉体对象的功能，使得生灵可以动态地选择。

2、墙上的开关，可以看到的开关是抽象的，不用管里面具体怎么实现的。

**优点：**

1、抽象和实现的分离。

2、优秀的扩展能力。

3、实现细节对客户透明。

**缺点：**

桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

**使用场景：**

1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。

2、对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。 3、一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

**注意事项：**

对于两个独立变化的维度，使用桥接模式再适合不过了。

🡺 BridgePattern

|  |
| --- |
| public class BridgePattern {  public static void main(String[] args) {  AbstractShape redCircle = new Circle(100,100, 10, new RedCircle());  AbstractShape greenCircle = new Circle(100,100, 10, new GreenCircle());  redCircle.draw();  greenCircle.draw();  } } |

🡺 DrawApi

|  |
| --- |
| public interface DrawApi {  void drawCircle(int radius, int x, int y); } |

🡺 GreenCircle

|  |
| --- |
| public class GreenCircle implements DrawApi {  @Override  public void drawCircle(int radius, int x, int y) {  System.*out*.println("Drawing Circle[ color: green, radius: "  + radius +", x: " +x+", "+ y +"]");  } } |

🡺 RedCircle

|  |
| --- |
| public class RedCircle implements DrawApi {  @Override  public void drawCircle(int radius, int x, int y) {  System.*out*.println("Drawing Circle[ color: red, radius: "  + radius +", x: " +x+", "+ y +"]");  } } |

🡺 AbstractShape

|  |
| --- |
| public abstract class AbstractShape {  DrawApi drawApi;  AbstractShape(DrawApi drawApi){  this.drawApi = drawApi;  }  public abstract void draw(); } |

🡺 Circle

|  |
| --- |
| public class Circle extends AbstractShape {  private int x, y, radius;   Circle(int x, int y, int radius, DrawApi drawAPI) {  super(drawAPI);  this.x = x;  this.y = y;  this.radius = radius;  }   @Override  public void draw() {  drawApi.drawCircle(radius,x,y);  } } |

### 组合模式(Composite)

将对象组合成树形结构以表示部分整体的关系，本模式使用户对单个对象和组合对象的使用具有一致性。

**意图：**

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

**主要解决：**

它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

**何时使用：**

1、您想表示对象的部分-整体层次结构（树形结构）。

2、您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

**如何解决：**

树枝和叶子实现统一接口，树枝内部组合该接口。

**关键代码：**

树枝内部组合该接口，并且含有内部属性 List，里面放 Component。

**应用实例：**

1、算术表达式包括操作数、操作符和另一个操作数，其中，另一个操作符也可以是操作数、操作符和另一个操作数。

2、在 JAVA AWT 和 SWING 中，对于 Button 和 Checkbox 是树叶，Container 是树枝。

**优点：**

1、高层模块调用简单。

2、节点自由增加。

**缺点：**

在使用组合模式时，其叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则。

**使用场景：**

部分、整体场景，如树形菜单，文件、文件夹的管理。

**注意事项：**

定义时为具体类。

🡺 CompositePattern

|  |
| --- |
| public class CompositePattern {  public static void main(String[] args) {  TreeNode root = new TreeNode("root");  TreeNode nodeA = new TreeNode("A");  TreeNode nodeB = new TreeNode("B");  TreeNode nodeC = new TreeNode("C");  nodeB.add(nodeC);  nodeA.add(nodeB);  root.add(nodeA);  root.getChildren();  } } |

🡺 TreeNode

|  |
| --- |
| public class TreeNode{  private String name;  private List<TreeNode> children = new ArrayList<>();   TreeNode(String name) {  this.name = name;  }   public void add(TreeNode node) {  children.add(node);  }   void getChildren(){  for (TreeNode child : children) {  System.*out*.println(child);  child.getChildren();  }  }   @Override  public String toString() {  return name;  } } |

|  |
| --- |
| **public** **class** Tree {  TreeNode root = **null**;  **public** Tree(String name) {  root = **new** TreeNode(name);  }  **public** **static** **void** main(String[] args) {  Tree tree = **new** Tree("A");  TreeNode nodeB = **new** TreeNode("B");  TreeNode nodeC = **new** TreeNode("C");  nodeB.add(nodeC);  tree.root.add(nodeB);  System.***out***.println("build the tree finished");  }  }  **class** TreeNode{  **private** String name;  **private** TreeNode parent;  **private** Vector<TreeNode> children = **new** Vector<TreeNode>();    **public** TreeNode(String name) {  **this**.name = name;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** TreeNode getParent(){  **return** parent;  }  **public** **void** setParent(TreeNode parent) {  **this**.parent = parent;  }  **public** **void** add(TreeNode node) {  children.add(node);  }  **public** **void** remove(TreeNode node) {  children.remove(node);  }  **public** Enumeration<TreeNode> getChildren(){  **return** children.elements();  }  } |

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

### 享元模式(Flyweight)

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。我们将通过创建 5 个对象来画出 20 个分布于不同位置的圆来演示这种模式。由于只有 5 种可用的颜色，所以 color 属性被用来检查现有的 Circle 对象。

**意图：**

运用共享技术有效地支持大量细粒度的对象。

**主要解决：**

在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。

**何时使用：**

1、系统中有大量对象。

2、这些对象消耗大量内存。

3、这些对象的状态大部分可以外部化。

4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。

5、系统不依赖于这些对象身份，这些对象是不可分辨的。

**如何解决：**

用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。

**关键代码：**

用 HashMap 存储这些对象。

**应用实例：**

1、JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面。

2、数据库的数据池。

**优点：**

大大减少对象的创建，降低系统的内存，使效率提高。

**缺点：**

提高了系统的复杂度，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱。

**使用场景：**

1、系统有大量相似对象。

2、需要缓冲池的场景。

**注意事项：**

1、注意划分外部状态和内部状态，否则可能会引起线程安全问题。

2、这些类必须有一个工厂对象加以控制。

🡺 FlyweightPattern

|  |
| --- |
| public class FlyweightPattern {  private static final String[] *COLORS* = {"Red", "Green", "Blue", "White", "Black"};   public static void main(String[] args) {  for (int i = 0; i < *COLORS*.length; ++i) {  Circle circle =   ShapeFactory.*getCircle*(*COLORS*[new Random().nextInt(*COLORS*.length)]);  circle.setX(new Random().nextInt(100));  circle.setY(new Random().nextInt(100));  circle.setRadius(new Random().nextInt(100));  circle.draw();  }  } } |

🡺 Shape

|  |
| --- |
| public interface Shape {  void draw(); } |

🡺 Circle

|  |
| --- |
| @Data public class Circle implements Shape {  private String color;  private int x;  private int y;  private int radius;   @Override  public void draw() {  System.*out*.println("Circle: Draw() [Color : " + color +  ", x : " + x +", y :" + y +", radius :" + radius);  } } |

🡺 ShapeFactory

|  |
| --- |
| public class ShapeFactory {  private static final HashMap<String, Shape> *CIRCLE\_MAP* = new HashMap<>();  static Circle getCircle(String color) {  Circle circle = (Circle) *CIRCLE\_MAP*.get(color);  if(circle == null) {  circle = new Circle();  circle.setColor(color);  *CIRCLE\_MAP*.put(color, circle);  System.*out*.println("Creating circle of color : " + color);  System.*out*.println(*CIRCLE\_MAP*.size());  }  return circle;  } } |

## 行为型设计模式

|  |  |  |  |
| --- | --- | --- | --- |
| **父类和子类** | **两个类之间** | **类的状态** | **通过中间类** |
| 策略模式 | 观察者模式 | 备忘录模式 | 访问者模式 |
| 模板方法模式 | 迭代子模式 | 状态模式 | 中介者模式 |
|  | 责任链模式 |  | 解释器模式 |
|  | 命令模式 |  |  |

### 策略模式(Strategy)

定义一系列算法，将它们封装起来，并使他们可以互相替换，本模式使得算法可以独立于使用他们的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口。

策略模式的决定权在用户，系统本身提供不同算法的实现，对各种算法做封装。策略模式多用在算法决策系统中，外部只需要决定用哪个算法即可。

**优点：**

1、算法可以自由切换。

2、避免使用多重条件判断。

3、扩展性良好。

**缺点：**

1、策略类会增多。

2、所有策略类都需要对外暴露。

**使用场景：**

1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。

2、一个系统需要动态地在几种算法中选择一种。

3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

**注意事项：**

如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

🡺 StrategyPattern

|  |
| --- |
| public class StrategyPattern {  public static void main(String[] args) {  String message = "this is a message";   // 通过获取不同的实现类来调用不同的算法进行操作  IStrategyPrintExecutor normalPrint = new NormalStrategyPrint();  normalPrint.print(message);   IStrategyPrintExecutor errorPrint = new ErrorStrategyPrint();  errorPrint.print(message);  } } |

🡺 IStrategyPrintExecutor

|  |
| --- |
| public interface IStrategyPrintExecutor {  */\*\*  \* print method  \*  \** ***@param*** *message message which is waiting to be print  \*/* void print(String message); } |

🡺 NormalStrategyPring

|  |
| --- |
| public class NormalStrategyPrint implements IStrategyPrintExecutor {  @Override  public void print(String message) {  System.*out*.println(message);  } } |

🡺 ErrorStrategyPrint

|  |
| --- |
| public class ErrorStrategyPrint implements IStrategyPrintExecutor {  @Override  public void print(String message) {  System.*err*.println(message);  } } |

### 模板模式(Template)

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，本模式使得子类可以不改变一个算法的结构而重定义该算法的某些特定步骤。

一个抽象类中，有一个模板方法，再定义多个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用。

**优点：**

1、封装不变部分，扩展可变部分。

2、提取公共代码，便于维护。

3、行为由父类控制，子类实现。

**缺点：**

每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

**使用场景：**

1、有多个子类共有的方法，且逻辑相同。

2、重要的、复杂的方法，可以考虑作为模板方法。

**注意事项：**

为防止恶意操作，一般模板方法都加上 final 关键词。

🡺 TemplateMethodPattern

|  |
| --- |
| public class TemplateMethodPattern {  public static void main(String[] args) {  AbstractTemplateMethodShoppingExecutor executor = new MenConsumer();  executor.goShopping();  } } |

🡺 AbstractShoppingExecutor

|  |
| --- |
| public abstract class AbstractShoppingExecutor {final void goShopping() {  shopping();  order();  pay();  } abstract void order();   private void shopping() {  System.*out*.println("开始购物");  }   private void pay() {  System.*out*.println("开始支付");  } } |

🡺 MenConsumer

|  |
| --- |
| public class MenConsumer extends AbstractTemplateMethodShoppingExecutor {  @Override  public void order() {  System.*out*.println("选择商品电脑");  System.*out*.println("选择商品笔记本");  } } |

### 观察者模式(Observer)

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

**优点：**

1、观察者和被观察者是抽象耦合的。

2、建立一套触发机制。

**缺点：**

1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

**使用场景：**

一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。

一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。

一个对象必须通知其他对象，而并不知道这些对象是谁。

需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

**注意事项：**

1、JAVA 中已经有了对观察者模式的支持类**Observer**和**Observable**。

2、避免循环引用。

3、如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。

🡺 ObserverPattern

|  |
| --- |
| public class ObserverPattern {  public static void main(String[] args) {  Subject sub = new MySubject();  sub.add(new Observer1());  sub.add(new Observer2());  sub.operation();  } } |

🡺 Observer

|  |
| --- |
| public interface Observer {void update(); } |

🡺 Observer1

|  |
| --- |
| public class Observer1 implements Observer {  @Override  public void update() {  System.*out*.println("observer1 has received and update");  } } |

🡺 Observer2

|  |
| --- |
| public class Observer2 implements Observer {  @Override  public void update() {  System.*out*.println("observer2 has received and update");  } } |

🡺 Subject

|  |
| --- |
| public interface Subject {void add(Observer observer);void operation(); } |

🡺 AbstractSubject

|  |
| --- |
| public abstract class AbstractSubject implements Subject {  private List<Observer> observers = new ArrayList<>();   @Override  public void add(Observer observer) {  observers.add(observer);  }   void notifyObservers() {  // 唤醒所有的观察者并同步更新  for (Observer observer : observers) {  observer.update();  }  } } |

🡺MySubject

|  |
| --- |
| public class MySubject extends AbstractSubject {  @Override  public void operation() {  System.*out*.println("update self!");  notifyObservers();  } } |

### 迭代子模式(Iterator)

提供一个方法顺序访问一个聚合对象的各个元素，而由不要暴露该对象的内部表示。

迭代器模式就是顺序访问聚集中的对象，集合中非常常见。这句话包含两层意思：一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。

**优点：**

1、它支持以不同的方式遍历一个聚合对象。

2、迭代器简化了聚合类。

3、在同一个聚合上可以有多个遍历。

4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

**缺点：**

由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

**使用场景：**

1、访问一个聚合对象的内容而无须暴露它的内部表示。

2、需要为聚合对象提供多种遍历方式。

3、为遍历不同的聚合结构提供一个统一的接口。

**注意事项：**

迭代器模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

🡺 IteratorPattern

|  |
| --- |
| public class IteratorPattern {  public static void main(String[] args) {  Collection collection = new CollectionImpl();  Iterator it = collection.iterator();  while(it.hasNext()) {  System.*out*.println(it.next());  }  } } |

🡺 Collection

|  |
| --- |
| public interface Collection<T> {  */\*\*  \* 获取迭代器  \*/* Iterator iterator();   */\*\*  \* 获取指定位置的元素  \*/* T get(int i);   */\*\*  \* 获取元素数量  \*/* int size(); } |

🡺 CollectionImpl

|  |
| --- |
| public class CollectionImpl<T> implements Collection{  private T[] elements;   @Override  public Iterator iterator() {  return new IteratorImpl<T>(this);  }   @Override  public T get(int i) {  return elements[i];  }   @Override  public int size() {  return elements.length;  } } |

🡺 Iterator

|  |
| --- |
| public interface Iterator<T> {T previous();T next();boolean hasNext();T first(); } |

🡺 IteratorImpl

|  |
| --- |
| public class IteratorImpl<T> implements Iterator{  private Collection<T> collection;  private int pos = -1;   IteratorImpl(Collection<T> collection) {  this.collection = collection;  }   @Override  public T previous() {  if(pos > 0) {  pos--;  }  return collection.get(pos);  }   @Override  public T next() {  if(pos < collection.size() - 1) {  pos++;  }  return collection.get(pos);  }   @Override  public boolean hasNext() {  return pos < collection.size() - 1;  }   @Override  public T first() {  pos = 0;  return collection.get(pos);  } } |

### 责任链模式(Chain of Responsibility)

责任链模式有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。

但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现在隐瞒客户端的情况下，对系统进行动态的调整。

链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要我们自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象。

**优点：**

1、降低耦合度。它将请求的发送者和接收者解耦。

2、简化了对象。使得对象不需要知道链的结构。

3、增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任。

4、增加新的请求处理类很方便。

**缺点：**

1、不能保证请求一定被接收。

2、系统性能将受到一定影响，而且在进行代码调试时不太方便，可能会造成循环调用。

3、可能不容易观察运行时的特征，有碍于除错。

**使用场景：**

1、有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。

2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

3、可动态指定一组对象处理请求。

**注意事项：**

在 JAVA WEB 中遇到很多应用。

🡺 ChainOfResponsibilityPattern

|  |
| --- |
| public class ChainOfResponsibilityPattern {  public static void main(String[] args) {  // 创建责任链上的所有的执行对象，每个责任链的构造参数可以不同，也可以是同一个参数  Handler shoppingHandler = new ShoppingHandler("shopping");  Handler orderHandler = new OrderHandler("order");   // 设置闭环责任链的循环次数，如果超过指定次数，则断开责任链  Handler payHandler = new PayHandler("pay", 3);   // 设置下一步需要执行的处理器对象  shoppingHandler.setHandler(orderHandler);  orderHandler.setHandler(payHandler);   // 如果责任链的尾部处理器设置的下一步处理器为头部处理器，则形成了一个闭环  payHandler.setHandler(shoppingHandler);   // 首先执行的处理器调用执行方法  shoppingHandler.operator();  } } |

🡺 Handler

|  |
| --- |
| public interface Handler {void operator();void setHandler(Handler handler); } |

🡺 AbstractHandler

|  |
| --- |
| public abstract class AbstractHandler implements Handler {  private Handler handler;   Handler getHandler() {  return handler;  }   @Override  public void setHandler(Handler handler) {  this.handler = handler;  } } |

🡺 ShoppingHandler

|  |
| --- |
| public class ShoppingHandler extends AbstractHandler {  private String name;   ShoppingHandler(String name) {  this.name = name;  }   @Override  public void operator() {  System.*out*.println(name + " started!");  if (getHandler() != null) {  getHandler().operator();  }  } } |

🡺 OrderHandler

|  |
| --- |
| public class OrderHandler extends AbstractHandler {  private String name;   OrderHandler(String name) {  this.name = name;  }   @Override  public void operator() {  System.*out*.println(name + " started!");  if (getHandler() != null) {  getHandler().operator();  }  } } |

🡺 PayHandler

|  |
| --- |
| public class PayHandler extends AbstractHandler {  private String name;  private Integer money;   PayHandler(String name, Integer money) {  this.name = name;  this.money = money;  }   @Override  public void operator() {  this.money = this.money - 1;  if(this.money > 0) {  System.*out*.println(this.name + " started!");  if (getHandler() != null) {  getHandler().operator();  }  } else {  System.*out*.println("you don't have enough money");  }  } } |

### 命令模式(command)

命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

将一个请求封装为一个对象，从而用不同的请求对客户进行参数化，对请求排队和记录请求日志，以及支持可撤销的操作。

命令模式很好理解，举个例子，司令员下令让士兵去干件事情，从整个事情的角度来考虑，司令员的作用是，发出口令，口令经过传递，传到了士兵耳朵里，士兵去执行。这个过程好在，三者相互解耦，任何一方都不用去依赖其他人，只需要做好自己的事儿就行，司令员要的是结果，不会去关注到底士兵是怎么实现的。我们看看关系图：

**意图：**

将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。

**主要解决：**

在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适。

**何时使用：**

在某些场合，比如要对行为进行"记录、撤销/重做、事务"等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将"行为请求者"与"行为实现者"解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

**如何解决：**

通过调用者调用接受者执行命令，顺序：调用者→接受者→命令。

**关键代码：**

定义三个角色：1、received 真正的命令执行对象 2、Command 3、invoker 使用命令对象的入口

**应用实例：**

struts 1 中的 action 核心控制器 ActionServlet 只有一个，相当于 Invoker，而模型层的类会随着不同的应用有不同的模型类，相当于具体的 Command。

**优点：**

1、降低了系统耦合度。

2、新的命令可以很容易添加到系统中去。

**缺点：**

使用命令模式可能会导致某些系统有过多的具体命令类。

**使用场景：**

认为是命令的地方都可以使用命令模式，比如： 1、GUI 中每一个按钮都是一条命令。 2、模拟 CMD。

**注意事项：**

系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作，也可以考虑使用命令模式，见命令模式的扩展。

|  |
| --- |
| public class CommandPattern {  public static void main(String[] args) {  // 执行正常打印命令  Receiver normalPrintReceiver = new NormalPrintReceiver();  Command cmd = new PrintCommand(normalPrintReceiver);  Invoker invoker = new Invoker(cmd);  invoker.action();    // 执行错误打印命令  Receiver errorPrintReceiver = new ErrorPrintReceiver();  cmd.setReceiver(errorPrintReceiver);  invoker.action();  } } |

|  |
| --- |
| public interface Command {void print();  void setReceiver(Receiver receiver); } |

|  |
| --- |
| public class PrintCommand implements Command {  private Receiver receiver;  PrintCommand(Receiver receiver) {  this.receiver = receiver;  }   @Override  public void setReceiver(Receiver receiver) {  this.receiver = receiver;  }   @Override  public void print() {  receiver.action();  } } |

|  |
| --- |
| public interface Receiver {void action(); } |

|  |
| --- |
| public class ErrorPrintReceiver implements Receiver {  @Override  public void action() {  System.*err*.println("errorPrintReceiver: command received!");  } } |

|  |
| --- |
| public class NormalPrintReceiver implements Receiver {  @Override  public void action() {  System.*out*.println("normalPrintReceiver: command received!");  } } |

|  |
| --- |
| public class Invoker{  private Command command;  Invoker(Command command) {  this.command = command;  }  void action() {  command.print();  } } |

命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开。

### 备忘录模式(Memento)

在不破坏对象的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类 A，A 中有各种属性，A 可以决定需要备份的属性，备忘录类 B 是用来存储 A 的一些内部状态，类 C 呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。

**意图：**

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

**主要解决：**

所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。

**何时使用：**

很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有"后悔药"可吃。

**如何解决：**

通过一个备忘录类专门存储对象状态。

**关键代码：**

客户不与备忘录类耦合，与备忘录管理类耦合。

**应用实例：**

返回上一步、存档等场景

**优点：**

1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。

2、实现了信息的封装，使得用户不需要关心状态的保存细节。

**缺点：**

消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

**使用场景：**

1、需要保存/恢复数据的相关状态场景。

2、提供一个可回滚的操作。

**注意事项：**

1、为了符合迪米特原则，还要增加一个管理备忘录的类。

2、为了节约内存，可使用原型模式+备忘录模式。

Original类是原始类，里面有需要保存的属性value及创建一个备忘录类，用来保存value值。Memento 类是备忘录类，Storage 类是存储备忘录的类，持有Memento类的实例，该模式很好理解。

🡺 MementoPattern

|  |
| --- |
| public class MementoPattern {  public static void main(String[] args) {  Original origin = new Original("old");  Storage<String> storage = new Storage<>(origin.createMemento());  System.*out*.println("before:"+origin.getValue());  origin.setValue("new");  System.*out*.println("after:"+origin.getValue());  origin.restoreMemento(storage.getMemento());  System.*out*.println("recover:"+origin.getValue());  } } |

🡺 Original

|  |
| --- |
| @Data public class Original{  private String value;   Original(String value) {  this.value = value;  }   Memento<String> createMemento() {  return new Memento<>(value);  }   void restoreMemento(Memento<String> memento) {  this.value = memento.getValue();  } } |

🡺 Storage

|  |
| --- |
| @Data public class Storage<T>{  */\*\*  \* 将memento的类型改为集合并添加获取对应memento对象的方法，则可以存储多个版本的属性  \*/* private Memento<T> memento;   Storage(Memento<T> memento) {  this.memento = memento;  } } |

🡺 Memento

|  |
| --- |
| @Data public class Memento<T>{  private T value;   Memento(T value) {  this.value = value;  } } |

### 状态模式(State)

允许对象在其内部状态改变时改变它的行为，对象看起来似乎改变了它的类。

当对象的状态改变时，同时改变其行为。状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。

**意图：**

允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

**主要解决：**

对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

**何时使用：**

代码中包含大量与对象状态有关的条件语句。

**如何解决：**

将各种具体的状态类抽象出来。

**关键代码：**

通常命令模式的接口中只有一个方法。而状态模式的接口中有一个或者多个方法。而且，状态模式的实现类的方法，一般返回值，或者是改变实例变量的值。也就是说，状态模式一般和对象的状态有关。实现类的方法有不同的功能，覆盖接口中的方法。状态模式和命令模式一样，也可以用于消除 if...else 等条件选择语句。

**应用实例：**

1、打篮球的时候运动员可以有正常状态、不正常状态和超常状态。

2、曾侯乙编钟中，'钟是抽象接口','钟A'等是具体状态，'曾侯乙编钟'是具体环境（Context）。

**优点：**

1、封装了转换规则。

2、枚举可能的状态，在枚举状态之前需要确定状态种类。

3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。

4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。

5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

**缺点：**

1、状态模式的使用必然会增加系统类和对象的个数。

2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。

3、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

**使用场景：**

1、行为随状态改变而改变的场景。

2、条件、分支语句的代替者。

**注意事项：**在行为受状态约束的时候使用状态模式，而且状态不超过 5 个。

State 类是个状态类，Context 类可以实现切换

🡺 StatePattern

|  |
| --- |
| public class StatePattern {  public static void main(String[] args) {  Context context = new Context();  IState firstState = new FirstState();  context.setState(firstState);  context.run();  IState secondState = new SecondState();  context.setState(secondState);  context.run();  } } |

🡺 IState

|  |
| --- |
| public interface IState {  void execute(); } |

🡺 FirstState

|  |
| --- |
| public class FirstState implements IState {  @Override  public void execute() {  System.*out*.println("this is first state");  } } |

🡺 SecondState

|  |
| --- |
| public class SecondState implements IState{  @Override  public void execute() {  System.*out*.println("this is second state");  } } |

🡺 Context

|  |
| --- |
| @Data public class Context{  private IState state;  void run() {  state.execute();  } } |

根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等。

#### 状态模式扩展

**共享状态**

在有些情况下多个环境对象需要共享同一个状态，如果希望在系统中实现多个环境对象实例共享一个或多个状态对象，那么需要将这些状态对象定义为环境的静态成员对象。

**简单状态模式与可切换状态的状态模式**

简单状态模式：简单状态模式是指状态都相互独立，状态之间无须进行转换的状态模式，这是最简单的一种状态模式。对于这种状态模式，每个状态类都封装与状态相关的操作，而无须关心状态的切换，可以在客户端直接实例化状态类，然后将状态对象设置到环境类中。如果是这种简单的状态模式，它遵循“开闭原则”，在客户端可以针对抽象状态类进行编程，而将具体状态类写到配置文件中，同时增加新的状态类对原有系统也不造成任何影响。

可切换状态的状态模式：大多数的状态模式都是可以切换状态的状态模式，在实现状态切换时，在具体状态类内部需要调用环境类Context的setState()方法进行状态的转换操作，在具体状态类中可以调用到环境类的方法，因此状态类与环境类之间通常还存在关联关系或者依赖关系。通过在状态类中引用环境类的对象来回调环境类的setState()方法实现状态的切换。在这种可以切换状态的状态模式中，增加新的状态类可能需要修改其他某些状态类甚至环境类的源代码，否则系统无法切换到新增状态。

### 访问者模式(Visitor)

表示一个作用于某对象结构中的各元素的操作，它能在不改变各元素类的前提下定义作用于这个元素的新操作。

访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。若系统数据结构对象易于变化，经常有新的数据对象增加进来，则不适合使用访问者模式。访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者。访问者模式将有关行为集中到一个访问者对象中，其改变不影响系统数据结构。其缺点就是增加新的数据结构很困难。

访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。

**意图：**

主要将数据结构与数据操作分离。

**主要解决：**

稳定的数据结构和易变的操作耦合问题。

**何时使用：**

需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。

**如何解决：**

在被访问的类里面加一个对外提供接待访问者的接口。

**关键代码：**

在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。

**应用实例：**

您在朋友家做客，您是访问者，朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式。

**优点：**

1、符合单一职责原则。

2、优秀的扩展性。

3、灵活性。

**缺点：**

1、具体元素对访问者公布细节，违反了迪米特原则。

2、具体元素变更比较困难。

3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象。

**使用场景：**

1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。

2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

**注意事项：**访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

🡺 VisitorPattern

|  |
| --- |
| public class VisitorPattern {  public static void main(String[] args) {  Visitor visitor = new MyVisitor();  Subject sub = new MySubject();  sub.accept(visitor);  } } |

🡺 Subject

|  |
| --- |
| interface Subject{  */\*\*  \* 核心方法，接受Visitor的访问  \*/* void accept(Visitor visitor);  String getSubject(); } |

🡺 MySubject

|  |
| --- |
| public class MySubject implements Subject{  String title = "mySubject";   @Override  public void accept(Visitor visitor) {  visitor.visit(this);  }   @Override  public String getSubject() {  // 不同的Subject实现类可以暴露不同的属性/元素  return "attributes";  } } |

🡺 Visitor

|  |
| --- |
| interface Visitor{  void visit(MySubject sub) ; } |

🡺 MyVisitor

|  |
| --- |
| public class MyVisitor implements Visitor{  @Override  public void visit(MySubject sub) {  // 不同的Visitor可以对不同的Subject采取不同的实现  System.*out*.println("visit the subject : " + sub.title + " : " + sub.getSubject());  } } |

该模式适用场景：如果我们想为一个现有的类增加新功能，不得不考虑几个事情：1、新功能会不会与现有功能出现兼容性问题？2、以后会不会再需要添加？3、如果类不允许修改代码怎么办？面对这些问题， 最好的解决方法就是使用访问者模式，访问者模式适用于数据结构相对稳定的系统，把数据结构和算法解耦。

### 中介者模式(Mediator)

用一个中介对象封装一些列的对象交互。

中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。如果使用中介者模式，只需关心和 Mediator 类的关系，具体类类之间的关系及调度交给 Mediator 就行，这有点像 spring 容器的作用。

User类统一接口，User1 和 User2 分别是不同的对象，二者之间有关联，如果不采用中介者模式，则需要二者相互持有引用，这样二者的耦合度很高，为了解耦，引入了 Mediator类，提供统一接口，MyMediator 为其实现类，里面持有 User1 和 User2 的实例，用来实现对 User1 和 User2 的控制。这样 User1 和 User2 两个对象相互独立，他们只需要保持好和Mediator 之间的关系就行，剩下的全由 MyMediator 类来维护！

**意图：**

用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

**主要解决：**

对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。

**何时使用：**

多个类相互耦合，形成了网状结构。

**如何解决：**

将上述网状结构分离为星型结构。

**关键代码：**

对象之间的通信封装到一个类中单独处理。

**应用实例：**

1、中国加入 WTO 之前是各个国家相互贸易，结构复杂，现在是各个国家通过 WTO 来互相贸易。

2、机场调度系统。

3、MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者。

**优点：**

1、降低了类的复杂度，将一对多转化成了一对一。

2、各个类之间的解耦。

3、符合迪米特原则。

**缺点：**

中介者会庞大，变得复杂难以维护。

**使用场景：**

1、系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。 2、想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

**注意事项：**不应当在职责混乱的时候使用。

🡺 MediatorPattern

|  |
| --- |
| public class MediatorPattern {  public static void main(String[] args) {  Mediator mediator = new MyMediator();  mediator.createMediator();  mediator.trade();  } } |

🡺 Mediator

|  |
| --- |
| public interface Mediator{  void createMediator();  void trade(); } |

🡺 MyMediator

|  |
| --- |
| public class MyMediator implements Mediator{  private AbstractBusiness provider;  private AbstractBusiness consumer;   @Override  public void createMediator() {  provider = new Provider(this);  consumer = new Consumer(this);  }   @Override  public void trade() {  provider.doBusiness();  System.*out*.println("中间商赚差价");  consumer.doBusiness();  } } |

🡺 AbstractBusiness

|  |
| --- |
| public abstract class AbstractBusiness {  private Mediator mediator;  AbstractBusiness(Mediator mediator) {  this.mediator = mediator;  }  public abstract void doBusiness(); } |

🡺 Consumer

|  |
| --- |
| public class Consumer extends AbstractBusiness {  public Consumer(Mediator mediator) {  super(mediator);  }   @Override  public void doBusiness() {  System.*out*.println("consumer buy goods!");  } } |

🡺 Provider

|  |
| --- |
| public class Provider extends AbstractBusiness {  Provider(Mediator mediator) {  super(mediator);  }   @Override  public void doBusiness() {  System.*out*.println("provider sell goods!");  } } |

### 解释器模式(Interpreter)

给定一个语言，定义它的文法含义，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。解释器模式用来做各种各样的解释器，如Java编译器、正则表达式和Spring EL表达式的解释器！

解释器模式一般需要构建语法树和环境类，语法树包含终结符与非终结符，环境类包含解释器之外的一些全局信息，一般是 HashMap。

**优点：**

1、可扩展性比较好，灵活。

2、增加了新的解释表达式的方式。

3、易于实现简单文法。

**缺点：**

1、可利用场景比较少，JAVA 中如果碰到可以用 expression4J 代替。

2、对于复杂的文法比较难维护。

3、解释器模式会引起类膨胀。

4、解释器模式采用递归调用方法。

**使用场景：**

1、可以将一个需要解释执行的语言中的句子表示为一个抽象语法树。

2、一些重复出现的问题可以用一种简单的语言来进行表达。

3、一个简单语法需要解释的场景。

🡺 InterpreterPattern

|  |
| --- |
| public class InterpreterPattern {  public static void main(String[] args) {  int firstResult = new Plus().interpret(new Context(9, 2));  System.*out*.println(firstResult);  int finalResult = new Minus().interpret(new Context(firstResult, 8));  System.*out*.println(finalResult);  } } |

🡺 Expression

|  |
| --- |
| public interface Expression {  int interpret(Context context); } |

🡺 Context

|  |
| --- |
| @Data public class Context {  private int firstNumber;  private int secondNumber;   Context(int firstNumber, int secondNumber) {  this.firstNumber = firstNumber;  this.secondNumber = secondNumber;  } } |

🡺 Minus

|  |
| --- |
| public class Minus implements Expression {  @Override  public int interpret(Context context) {  return context.getFirstNumber() - context.getSecondNumber();  } } |

🡺 Plus

|  |
| --- |
| public class Plus implements Expression {  @Override  public int interpret(Context context) {  return context.getFirstNumber() + context.getSecondNumber();  } } |

# 编程技巧

1、if及时返回