## 资料地址

<https://www.cnblogs.com/lspz/p/6859991.html>

https://www.cnblogs.com/maowang1991/archive/2013/04/15/3023236.html

## 7大原则

### 单一职责

一个类一个职责

### 开闭原则

一个软件实体如类、模块和函数对扩展开放，对修改关闭。

### 里氏替换原则

继承和派生类（尽量不重写父类已实现的方法，可以用接口等其他方法绕过）

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。

### 依赖倒置原则

以开闭原则为基础，对接口编程，不要针对实现编程。

高层模块不依赖于底层模块，二者都应依赖其抽象细节；细节应该依赖于抽象不依赖于具体。

### 接口隔离原则

建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。降低类之间的耦合度，降低依赖。便于维护和升级。

### 迪米特法则（最少知道原则）

一个类应尽量少的与其他类之间发生相互作用，使得系统功能模块相对独立。就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地将逻辑封装在内部，对外除了提供public方法，不对外泄露任何信息。

### 合成（组合/聚合）复用原则

尽量使用合成/聚合的方式，而不是使用继承的关系来达到复用的原则。

## 24种设计模式

### 创建者模式

#### 单例模式（Singleton）

确保一个类只有一个实例，并提供全局访问点。

使用synchronized关键字锁定对象，一定要恰当使用。

使用类的静态方法实现单例模式效果相同，但两者有区别。

|  |
| --- |
| /\*\*  \* 单例模式  \* 单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，  \* 该对象只有一个实例存在。这样的模式有几个好处：  1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。  2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。  3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。  （比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。  \*/  public class SingletonModel {  /\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/  private static SingletonModel model = null;  /\* 私有构造方法，防止被实例化 \*/  private SingletonModel(){}  public static SingletonModel getInstance(){  if (model == null) syncInit();  return model;  }  /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/  public Object readResolve() {  return model;  }  private synchronized static void syncInit(){  if (model == null) model = new SingletonModel();  }  public void run(){  System.out.println("This is a singleton.");  }  public static void main(String[] args) {  SingletonModel model = SingletonModel.getInstance();  model.run();  }  } |

#### 工厂模式（Factory）

定义一个创建对象的接口，由子类决定要实例化的类是哪一个。工厂方法让类实例化推迟到子类。

普通工厂：建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

多工厂：对普通工厂方法模式进行改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

静态工厂：将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

#### 抽象工厂模式（Abstract Factory）

提供一个接口，用于创建相关或依赖对象的家族，而不需要指定具体类。

创建者模式（Builder）

将各种产品集中起进行管理，用来创建复合对象（某个类具有不同的属性）

即使用创建者模式封装一个产品的构造过程，并允许按步骤构造。将一个复杂对象的构建与它表示分离，使得同样的构建过程可以创建不同的表示。

|  |
| --- |
| /\*\*  \* 工厂方法模式  \*/  public class FactoryMethod {  public static void main(String[] args) {  /\*\*  \* 普通工厂模式  \* 建立一个工厂类，对实现了同一接口的一些类进行实例的创建  \* 如：发送信息，可以短信/邮件/qq/微信等  \*/  SenderFactory factory = new SenderFactory();  factory.produce("sms").send();  /\*\*  \* 多工厂模式  \* 对普通工厂方法模式的改进，在普通工厂方法模式中，  \* 如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，  \* 分别创建对象  \*/  new MultiSenderFactory().produceQQ().send();  /\*\*  \* 静态工厂方法模式  \* 多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用  \*/  StaticSenderFactory.produceQQ().send();  /\*\*  \* 抽象工厂模式  \* 工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则。  \* 所以，从设计角度考虑，有一定的问题，如何解决？  \* 就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，  \* 不需要修改之前的代码。  \*/  Sender sender = new QQProduce().produce();  sender.send();  }  }  interface Sender{  void send();  }  class SmsSender implements Sender{  public void send() {  System.out.println("this is sms sender");  }  }  class QQSender implements Sender{  public void send() {  System.out.println("this is QQ sender");  }  }  class SenderFactory{  public Sender produce(String type){  if ("qq".equals(type)) return new QQSender();  else if ("sms".equals(type)) return new SmsSender();  return null;  }  }  class MultiSenderFactory{  public Sender produceQQ(){  return new QQSender();  }  public Sender produceSms(){  return new SmsSender();  }  }  class StaticSenderFactory{  public static Sender produceQQ(){  return new QQSender();  }  public static Sender produceSms(){  return new SmsSender();  }  }  interface Provider{  Sender produce();  }  class QQProduce implements Provider{  public Sender produce() {  return new QQSender();  }  }  class SmsProduce implements Provider{  public Sender produce() {  return new SmsSender();  }  } |

#### 原型模式（Prototype）

当创建给定类的实例过程很复杂时，就使用原形模式。

与工厂模式无关，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。

|  |
| --- |
| /\*\*  \* 原型模式  \* 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  \* 1、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。  2、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。（  要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象。）  \*/  @Data  public class PrototypeMethod implements Cloneable,Serializable {  private String key;  private String value;  /\*\*浅复制\*/  @Override  protected Object clone() throws CloneNotSupportedException {  return super.clone();  }  public Object deepClone() throws ClassNotFoundException,IOException{  /\*写入当前对象的二进制流\*/  ByteArrayOutputStream bos = new ByteArrayOutputStream();  ObjectOutputStream oos = new ObjectOutputStream(bos);  oos.writeObject(this);  /\*读出二进制流产生新对象\*/  ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());  ObjectInputStream ois = new ObjectInputStream(bis);  return ois.readObject();  }  } |

#### 多例模式（Multitude）

在一个解决方案中结合两个或多个模式，以解决一般或重复发生的问题

定义：作为对象的创建模式，多例模式中的多例类可以有多个实例，而且多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

特点：

1、多例类可有多个实例

2、多例类必须自己创建、管理自己的实例，并向外界提供自己的实例

3、根据是否有实例上限分为：有上限多例类和无上限多例类。

### 结构模式

#### 适配器模式（Adapter）

将一个类的接口，转换成客户期望的另一个接口。适配器让原来不兼容的类可以合作无间。

类的适配器：多重继承

对象的适配器：组合

接口的适配器

#### 装饰模式（Decorator）

对象的适配器

动态地将责任附加到对象上，若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

#### 代理模式（Proxy）

对象的适配器

为另一个对象提供一个替身或占位符以控制对这个对象的访问。

#### 外观模式（Facade）

对象的适配器

提供一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

#### 桥接模式（Bridge）

使用桥接模式通过将实现和抽象放在两个不同的类层次中而使它们可以独立改变。

#### 组合模式（Composite）

允许你将对象组合成树形结构来表现"整体/部分"层次结构. 组合能让客户以一致的方式处理个别对象以及对象组合.

#### 享元模式（Flyweight）

如想让某个类的一个实例能用来提供许多的“虚拟实例”，就使用蝇量模式。

行为模式