## 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。

|  |
| --- |
| /\*\*  \* 多例模式  \* 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  \* @author yaozou  \* @create 2018-04-26 21:36  \*\*/  public class MultitudeMethod {  private static int num = 3; //实例个数  private static Map<Integer,MultitudeMethod> multitude = new HashMap<Integer, MultitudeMethod>(); //存放实例  private static int index; //当前实例所在位置  static{ //当类加载时创建指定个数的实例  for(int i = 0;i < num;i++){  multitude.put(i,new MultitudeMethod());  }  }  private MultitudeMethod(){}  public static MultitudeMethod getInstance(){  Random random = new Random();  index = random.nextInt(num); //随机获取实例  return multitude.get(index);  }  public int getIndex() {  return index;  }  public static void main(String[] args) {  System.out.println("当前实例是第"+MultitudeMethod.getInstance().getIndex());  System.out.println("当前实例是第"+MultitudeMethod.getInstance().getIndex());  System.out.println("当前实例是第"+MultitudeMethod.getInstance().getIndex());  }  } |

### 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

|  |
| --- |
| /\*\*  \* 适配器模式  \* 要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。  \*/  public class AdapterMethod {  public static void main(String[] args){  //类的适配器模式  ClassAdapter classAdapter = new ClassAdapter();  classAdapter.run();  //对象的适配器模式  ObjectAdapter objectAdapter = new ObjectAdapter();  objectAdapter.run();  //接口的适配器模式  InterfaceAdapter interfaceAdapter = new InterfaceAdapter();  interfaceAdapter.run();  }  }  interface Targetable{  void methodFirst();  void methodSecond();  }  class Source{  public void methodFirst(){  System.out.println("This a first method.");  }  }  /\*\*  \* 类的适配器模式  \* 核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里  \* 应用场景：将一个接口转换成满足另一个新接口的类时，可以使用类的适配器模式，创新一个新类，继承原有的类，实现新的接口。  \*/  class ClassAdapter{  class Adapter extends Source implements Targetable{  public void methodSecond() {  System.out.println("This a second method.");  }  }  public void run(){  Targetable targetable = new Adapter();  targetable.methodFirst();  targetable.methodSecond();  }  }  /\*\*  \* 对象的适配器模式  \* 核心思想：基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。  \* 应用场景：将一个对象转换成满足另一个新接口的对象时，可以创建一个Adapter类，持有原类的一个实例，在Adapter类的方法中，调用实例的方法  \*/  class ObjectAdapter{  @Data  class Adapter implements Targetable{  private Source source;  public Adapter(Source source){  super();  this.source = source;  }  public void methodFirst() {  source.methodFirst();  }  public void methodSecond() {  System.out.println("This a second method.");  }  }  public void run(){  Targetable targetable = new Adapter(new Source());  targetable.methodFirst();  targetable.methodSecond();  }  }  /\*\*  \* 接口的适配器模式  \* 核心思想：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，  \* 因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，  \* 该抽象类实现了该接口，实现了所有的方法，  \* 而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。  \* 应用场景：实现一个接口中所有的方法时，可以创建一个抽象类AbstractClass，实现所有方法，写别类时继承抽象类即可  \*/  class InterfaceAdapter{  abstract class AbstractClass implements Targetable{  public void methodFirst() { }  public void methodSecond(){ }  }  class SourceSubFirst extends AbstractClass{  @Override  public void methodFirst() {  System.out.println("This a first method.");  }  }  class SourceSubSecond extends AbstractClass{  @Override  public void methodSecond() {  System.out.println("This a second method.");  }  }  public void run(){  Targetable targetable1 = new SourceSubFirst();  Targetable targetable2 = new SourceSubSecond();  targetable1.methodFirst();  targetable1.methodSecond();  targetable2.methodFirst();  targetable2.methodSecond();  }  } |

#### 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。

|  |
| --- |
| /\*\*  \* 装饰模式  \* @author yaozou  \* @create 2018-04-20 14:30  \*\*/  public class DecoratorMethod {  public static void main(String[] args) {  // 基本思想：装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例  // Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能  // 应用场景：1、需要扩展一个类的功能。  // 2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）  // 缺点：产生过多相似的对象，不易排错！  DecoratorMethod method = new DecoratorMethod();  method.run();  }  interface Sourceable{  void method();  }  class Source implements Sourceable{  public void method() {  System.out.println("This is a method");  }  }  class Decorator implements Sourceable{  private Sourceable source;  public Decorator(Sourceable source){  this.source = source;  }  public void method() {  System.out.println("Before Decorator......");  source.method();  System.out.println("After Decorator.......");  }  }  public void run(){  Sourceable source = new Source();  Decorator decorator = new Decorator(source);  decorator.method();  }  } |

#### 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。

|  |
| --- |
| /\*\*  \* 代理模式  \* @author yaozou  \* @create 2018-04-20 15:40  \*\*/  public class ProxyMethod {  public static void main(String[] args) {  // 多一个代理类出来，替原对象进行一些操作。比如：有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。  /\*\*  \* 应用场景：  \* 如果已有的方法在使用的时候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(开闭原则)。  \* 2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。  \* 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！  \*/  ProxyMethod method = new ProxyMethod();  method.run();  }  interface Sourceable{  void method();  }  class Source implements Sourceable{  public void method() {  System.out.println("this is original method");  }  }  class Proxy implements Sourceable{  private Source source;  public Proxy(){  super();  this.source = new Source();  }  public void method() {  System.out.println("after proxy!");  source.method();  System.out.println("before proxy!");  }  }  public void run(){  Proxy proxy = new Proxy();  proxy.method();  }  } |

#### 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。

|  |
| --- |
| /\*\*  \* 外观模式  \* @author yaozou  \* @create 2018-04-20 16:12  \*\*/  public class FacadeMethod {  public static void main(String[] args) {  /\*\*  \* 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  \*/  // 例如打开电脑  FacadeMethod method = new FacadeMethod();  method.open();  method.close();  }  class CPUClass{  public void startup(){  System.out.println("CPU startup......");  }  public void stopdown(){  System.out.println("CPU stopdown......");  }  }  class MemoryClass{  public void startup(){  System.out.println("Memory startup......");  }  public void stopdown(){  System.out.println("Memory stopdown......");  }  }  class DiskClass{  public void startup(){  System.out.println("Disk startup......");  }  public void stopdown(){  System.out.println("Disk stopdown......");  }  }  class Computer{  private CPUClass cpu;  private MemoryClass memory;  private DiskClass disk;  public Computer(){  this.cpu = new CPUClass();  this.memory = new MemoryClass();  this.disk = new DiskClass();  }  public void startup(){  cpu.startup();  memory.startup();  disk.startup();  }  public void stopdown(){  cpu.stopdown();  memory.stopdown();  disk.stopdown();  }  }  public void open(){  Computer computer = new Computer();  computer.startup();  }  public void close(){  Computer computer = new Computer();  computer.stopdown();  }  } |

#### 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。

|  |
| --- |
| /\*\*  \* 桥接模式  \*/  public class BridgeMethod {  public static void main(String[] args) {  // 基本思路：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，  /\*\*  \* 例如：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  \*/  BridgeMethod method = new BridgeMethod();  method.run();  }  interface Driver{  void connection();  }  class MysqlDriver implements Driver{  public void connection() {  System.out.println("mysql is connecting......");  }  }  class OracleDriver implements Driver{  public void connection() {  System.out.println("oracle is connecting.......");  }  }  @Data  abstract class DriverManager{  private Driver driver;  public void connection(){  driver.connection();  }  }  class Client extends DriverManager{  public void connection(){  getDriver().connection();  }  }  public void run(){  Client client = new Client();  /\*调用第一个对象\*/  Driver mysqlDriver = new MysqlDriver();  client.setDriver(mysqlDriver);  client.connection();  /\*调用第二个对象\*/  Driver oracleDriver = new OracleDriver();  client.setDriver(oracleDriver);  client.connection();  }  } |

#### 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.

|  |
| --- |
| /\*\*  \* 组合模式  \* 部分-整体模式在处理类似树形结构的问题时比较方便  \*/  public class CompositeMethod {  public static void main(String[] args) {  }  @Data  class TreeNode{  private String name;  private TreeNode parent;  private Vector<TreeNode> children = new Vector<TreeNode>();  public TreeNode(String name){  this.name = name;  }  /\*添加孩子节点\*/  public void add(TreeNode node){  children.add(node);  }  /\*删除孩子节点\*/  public void remove(TreeNode node){  children.remove(node);  }  /\*取得孩子节点\*/  public Enumeration<TreeNode> getChildren(){  return children.elements();  }  }  class Tree{  TreeNode root = null;  public Tree(String name){  root = new TreeNode(name);  }  }  public void run(){  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!");  }  } |

#### 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

|  |
| --- |
| /\*\*  \* 享元模式  \*/  public class FlyweightMethod {  public static void main(String[] args) {  /\*\*  \* 主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。  \* FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，  \* 如果有，就返回已经存在的对象，如果没有，则创建一个新对象，FlyWeight是超类。  \* 一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我们不难总结出：  \* 适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password及dbname，  \* 这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，  \* 其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。  \*/  //通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能  ConnectionPool connectionPool = ConnectionPool.getInstance();  connectionPool.getConnection();  connectionPool.release();  }  }  class ConnectionPool{  private Vector<Connection> pool;  /\*公有属性\*/  private String url = "jdbc:mysql://localhost:3306/test";  private String username = "root";  private String password = "123456";  private String driverClassName = "com.mysql.jdbc.Driver";  private int poolSize = 100;  private static ConnectionPool instance = null;  Connection conn = null;  private ConnectionPool(){  pool = new Vector<Connection>(poolSize);  for (int i=0;i<poolSize;i++) {  try{  Class.forName(driverClassName);  conn = DriverManager.getConnection(url,username,password);  pool.add(conn);  } catch (ClassNotFoundException e) {  e.printStackTrace();  }catch (SQLException e){  e.printStackTrace();  }  }  }  private static synchronized ConnectionPool createPool(){  if (instance == null) instance = new ConnectionPool();  return instance;  }  public static ConnectionPool getInstance(){  if (instance == null) return createPool();  return instance;  }  /\*返回连接到连接池\*/  public synchronized void release(){  pool.add(conn);  }  /\* 返回连接池中的一个数据库连接 \*/  public synchronized Connection getConnection(){  if (pool.size() > 0){  Connection connection = pool.get(0);  pool.remove(connection);  return connection;  }else return null;  }  } |