**2017.5.1**

**1.单例模式：**

单例模式是指，单例类在所有地方使用的都是同一个对象。即：只能通过静态方法获取对象，不能通过构造方法new对象。

要实现主要有两种方式：

1. 饿汉式（简单粗暴）：

|  |
| --- |
| **public class SingleModel** {  */\* \* 单例模式1 饿汉式 \* 缺陷是：不管是否使用这个类，都会创建出这样一个对象。有一定内存消耗。 \* 好处是，线程安全。 \* \*/* **private static SingleModel** *singleModel*=**new** SingleModel();  **private** SingleModel(){}  **public static SingleModel** getSingleModel(){  **return** *singleModel*;  } } |

2）懒汉式(静态类实现)

|  |
| --- |
| /\*\* 静态内部类 \* 静态成员，只有被创建对象或者被显示调用的时候，才会加载，且只加载一次。  \* Created by horse on 2017/4/30.  \*/  public class SingleModel3 {  public SingleModel3(){  System.out.println("single 初始化");  }  //static成员是位于一个类的外部级别。  private static class loadLazy{  static {  System.out.println("loadlazy jingtai");  }  //final 修饰的变量，不能再被初始化了。  public loadLazy(){ //当 类.静态成员 的时候，并不会执行构造方法，之后实例化才会。  System.out.println("loadlazy");  }  private static final SingleModel3 SINGLE\_MODEL=new SingleModel3();  }  static{  System.out.println("执行静态代码快");  }  public static SingleModel3 getSingleModel(){  //真正调用了getSingleModel方法时，才会加载loadLazy。  return loadLazy.SINGLE\_MODEL;  }  **任何一个类，都只在其被用到的时候才会加载。所以静态内部类，在使用的时候才被加载。即：**  **\* 调用 getSingleModel方法的时候，才会被使用。**  **\* 这样就避免了饿汉式，一开始就创立一个对象。**  **\* 当实例化 或者 类.静态成员 的时候**  **\* 1.获取 类的 .class字节码文件。**  **\* 2. 执行静态代码块 和 初始化静态成员。 静态内部类无任何影响。**  **\* 3. 如果要实例化，那么在堆中的新生代中创建一个对象，并把基本类型初始化。**  **\* 4.调用构造函数。** |

**2.代理模式之动态代理：**

1)代理模式：

代理模式就是为了隐藏被代理类，并且起到控制代理类行为的作用。

要点：

1. 被代理类和代理类都要实现同一接口。
2. 代理类中要持有被代理类的对象。

|  |
| --- |
| 三要素：抽象接口，被代理类，代理类。 |
| 抽象接口：  public interface KindWomen {  public void throwEye();  public void doSomething()  }起来不 |
| 被代理类：  public class PanJinlian implements KindWomen {  public void throwEye(){  System.out.println("抛媚眼");  }  public void doSomething(){  System.out.println("doSomething");  }  } |
| 代理类：  public class WangPo implements KindWomen {  private KindWomen kindWomen; //持有被代理类的对象。  public WangPo(KindWomen kindWomen){  this.kindWomen=kindWomen;  }  @Override  public void throwEye() {  if(true) //控制突出了代理的功能。  kindWomen.throwEye();  System.out.println("传递给武松"); //加强功能突出了装饰的功能。  }  @Override  public void doSomething() {  kindWomen.doSomething();  }  } |

代理模式的缺陷：

一个被代理类就必须有一个相应的代理类，代码复杂。那么是否存在多个被代理类，一个代理类呢？

2）动态代理：

两元素： 被代理类，抽象接口。

|  |
| --- |
| 抽象接口：  public interface Person {  void say(String string);  } |
| 被代理类：  public void say(String string) {  System.out.println("lalalalla"+string);  } |
| 动态代理的实现：  public class ProxyDemo {  public static void main(String[] args) {  final Student student=new Student();  \* Person person= (Person) Proxy.newProxyInstance(student.getClass().getClassLoader(), student.getClass().getInterfaces(),  new InvocationHandler() {  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  boolean flag=false;  Object object=null;  System.out.println("abc");  if (flag)  object=method.invoke(student,args);  System.out.println("zzzzzzzzzzzzzzzz");  return object;  }  });  \* person.say("zzzz");  }  } |

蓝\*处，就相当于调用了invoke方法

Person person 表示通过Proxy类的方法，创建了一个代理对象，这个代理对象用接口接受（实际上，他们是虚拟的因为并没有代理类）。

当person.say即：调用了invoke方法，执行红颜色的代码。