## 23种设计模式：http://c.biancheng.net/design\_pattern/

## 1：简单工厂模式

定义: 由一个工厂对象决定创建出哪一种产品类的实例。（不属于23种设计模式之一）  
  
 #适用场景：1.工厂类负责创建的对象比较少  
 2.客户端（应用层）只知道传入工厂类的参数，对于如何创建对象（逻辑）不关心。  
 #注意事项：  
  
 #优点：只需要传入一个正确的参数，就可以获取你所需要的对象，而无需知道其创建的细节。  
  
 #缺点：工厂类的职责相对过重，增加新的产品，需要修改工厂类的逻辑判断，违背开闭原则  
  
 举例：

### Code：

### UML：

### 源码解析：

## 2：工厂模式

定义: 定义一个创建对象的接口，但让实现这个接口的类来决定实例化哪个类，工厂方法让类的实例化推迟到子类中进行。  
  
 #类型：创建型  
 #适用场景：1.创建对象需要大量的代码。  
 2.客户端（应用层）不依赖于产品类实例如何被创建，实现等细节。  
 3.一个类通过其子类来指定创建哪个对象。  
 #注意事项：  
  
 #优点：1.用户只需要关心所需产品对应的工厂，无需关心创建细节；2.加入新产品符合开闭原则，提高可扩展性。  
  
 #缺点：1.类的个数容易过多，增加复杂度；2.增加了系统的抽象性和理解难度。

### Code：

### UML：

### 源码解析：

## 3：抽象工厂模式

### Code：

### UML：

### 源码解析：

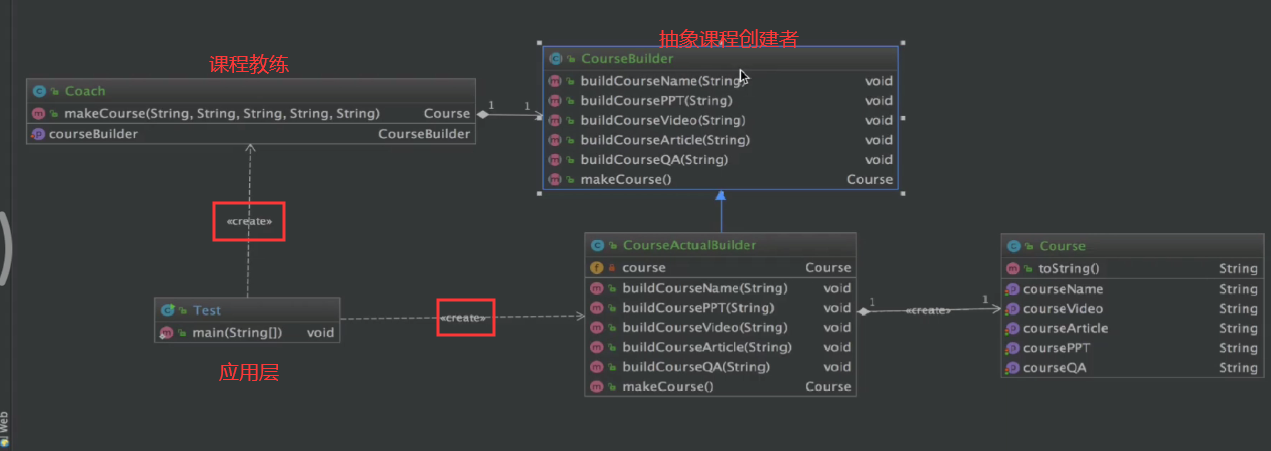
## 4：建造者模式

定义: 将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。（用户只需指定需要建造的类型就可以得到他们，建造过程以及细节不需要知道。）  
  
 #类型：创建型  
 #适用场景：  
 1.如果一个对象有非常复杂的内部结构（很多属性）。  
 2.想把复杂对象的创建和使用分离。  
 #注意事项：  
  
 #优点：1.封装性好，创建和使用分离  
 2.扩展性好，"建造类"之间独立，一定程度上解耦。  
  
 #缺点：1.产生多余的Builder对象；  
 2.产品内部发生变化，建造者都要修改，成本较大。  
  
 区别：1.和工厂模式比较类似，但是建造者模式更注重方法的调用顺序，而工厂模式注重创建产品；  
 2.创建力度不同：建造者模式可以创建一些复杂的产品，由各种复杂的部件组成，而工厂模式创建出来的都是一个样子的产品。  
 3.关注点：工厂模式注重只要把这个对象创建出来就ok，而建造者模式不仅要创建出这个产品，还要知道这个产品由哪些部件组成。  
 举例：

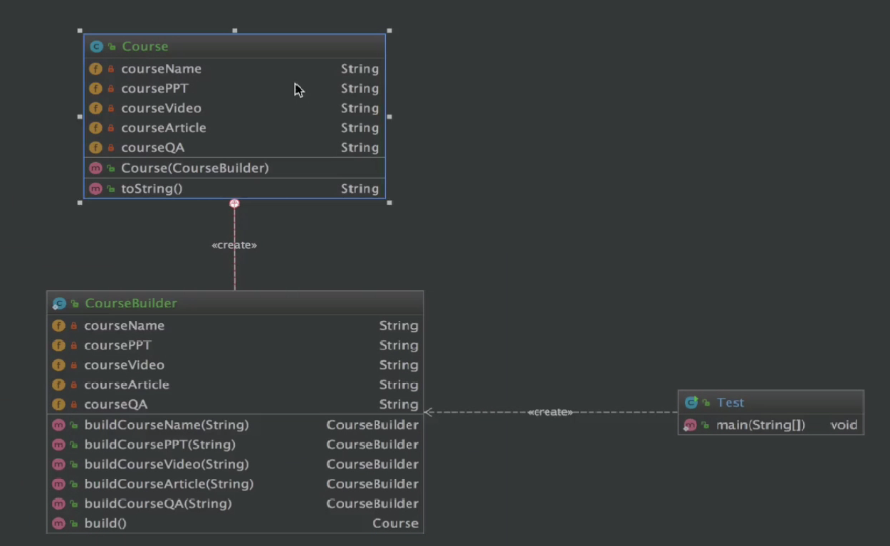
### Code：

### UML：

#### v1：



#### v2:



### 源码解析：

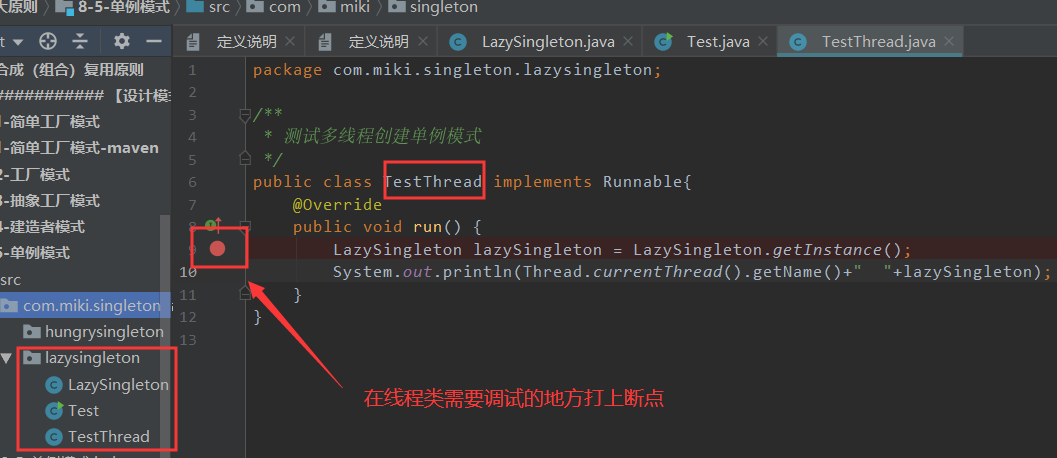
## 5：单例模式

定义: 保证一个类仅有一个实例，并提供一个全局访问点。  
  
 #类型：创建型  
 #适用场景：  
 1.想确保任何情况下都绝对只有一个实例。  
  
 #注意事项：1.必须要有私有构造器，保证不能外部new出对象  
 2.线程安全  
 3.延迟加载  
 4.序列号化和反序列化的安全问题  
 5.注意防御反射攻击  
  
 #优点：1.在内存中只有一个实例，减少了内存开销；  
 2.可以避免对资源的多重占用；  
 3.设置全局访问点，严格控制访问点；  
  
 #缺点：1.没有接口，扩展困难；

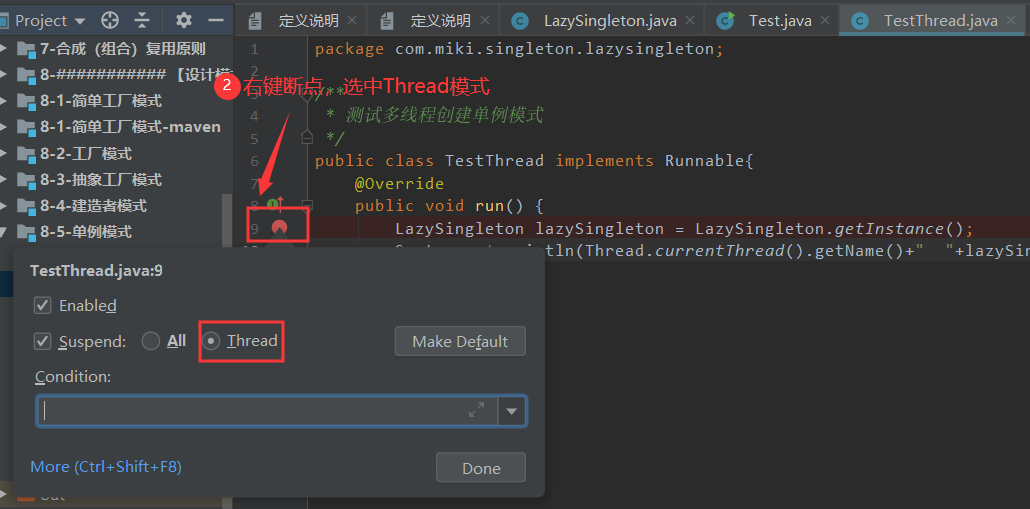
### Code：

#### 使用debug模式调试多线程（多线程下单例对象被重复创建）

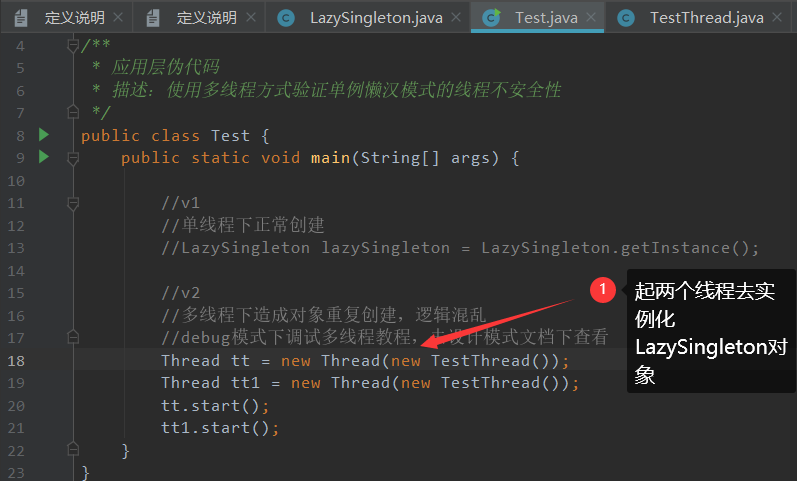
1. 使用debug模式调试多线程



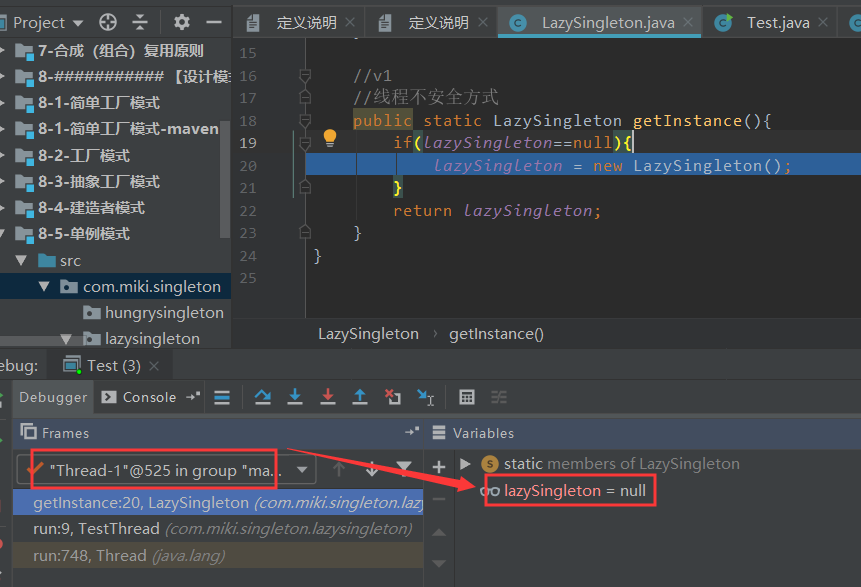
* 1. 右键debug断点设置为多线程模式



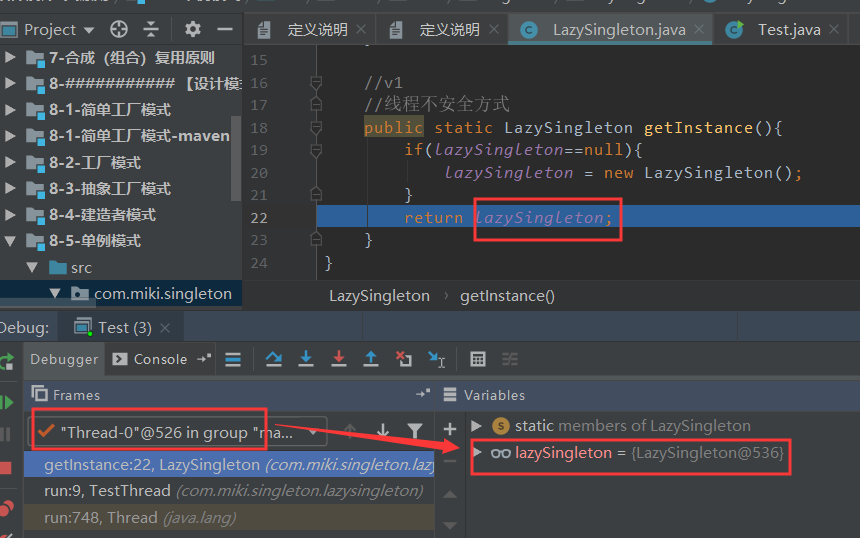
* 1. 运行程序，开始调试



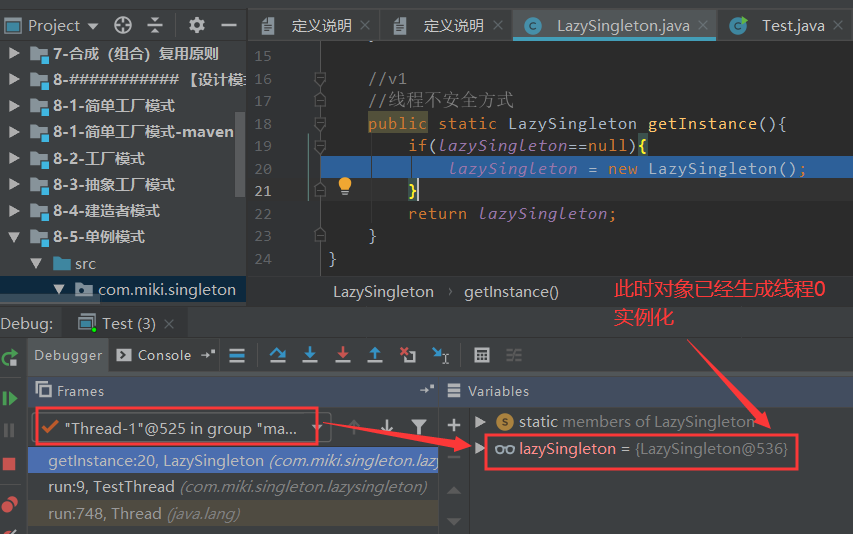
* 1. 进入线程1



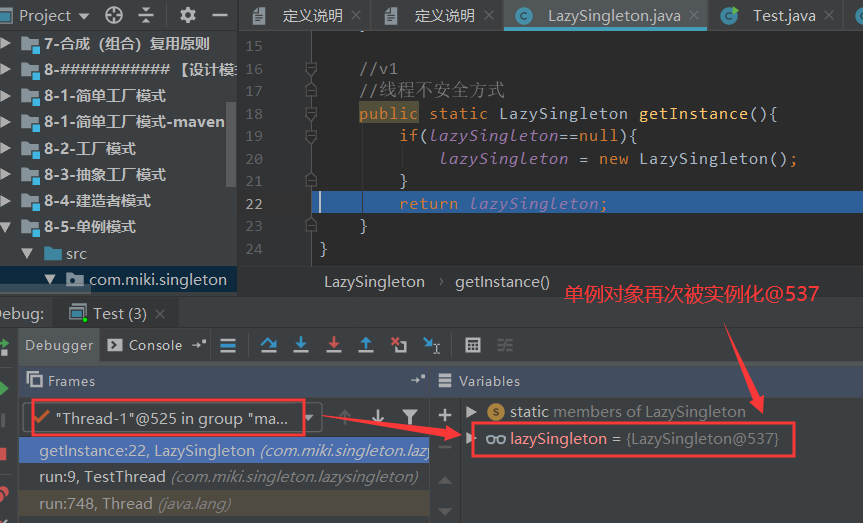
* 1. 进入线程2：并向下调试，生成对象lazySingleton@536



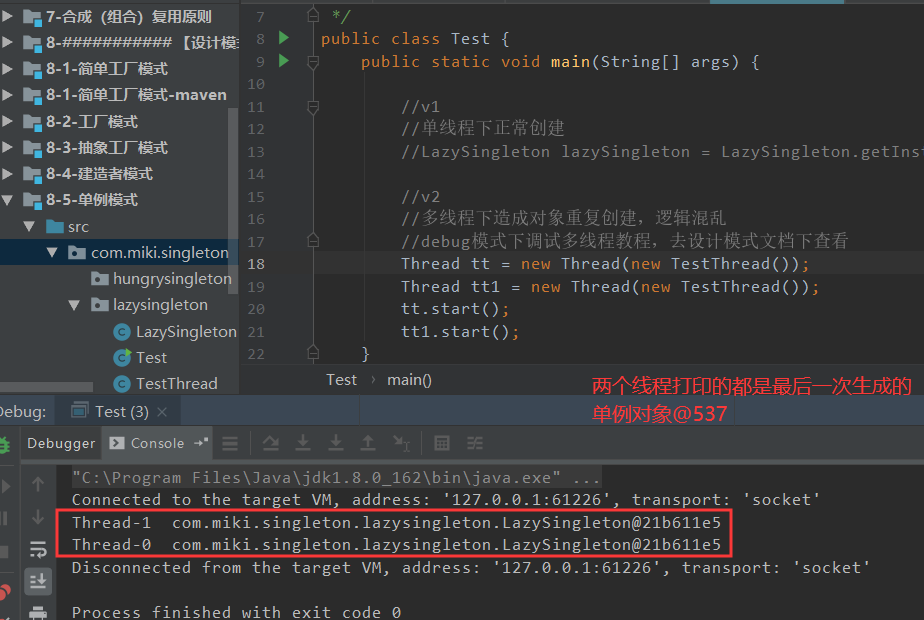
* 1. 再回到线程1：此时单例对象已经被线程0实例化



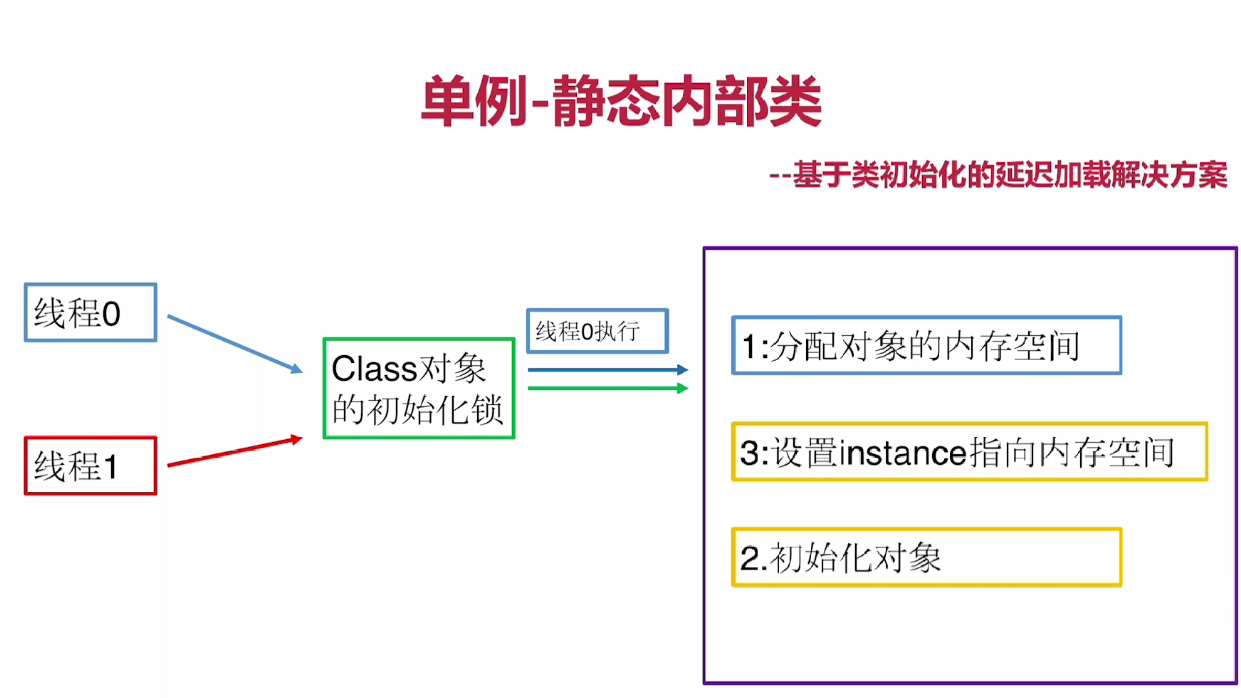
* 1. 向下调试，单例对象再次被实例化



1.7. 执行完毕，单例对象被实例化两次，切且最后打印出来的就是最后一次调试生成的单例对象



#### 延迟加载解决方案：



### UML：

### 源码解析：

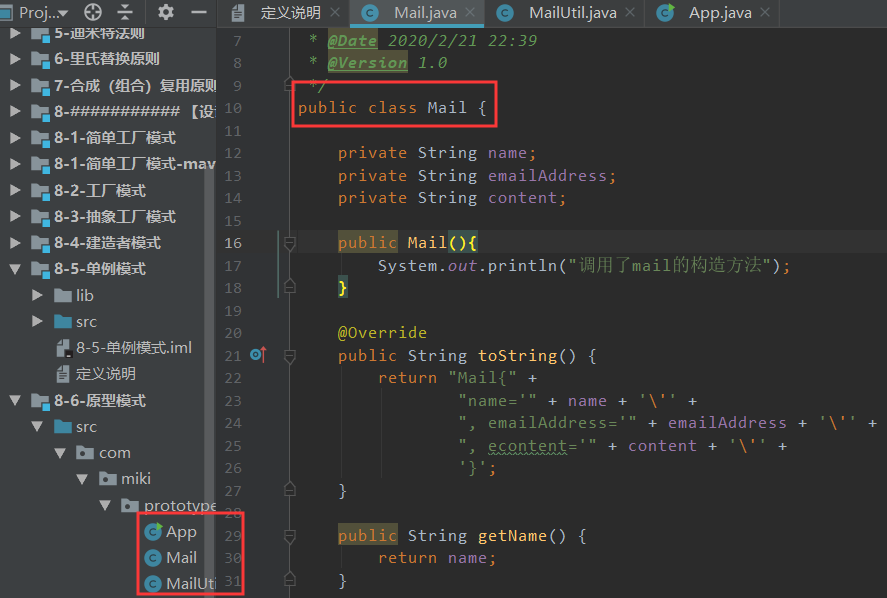
## 6：原型模式

定义: 指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。（不需要知道任何创建细节，不调用构造函数）  
  
 #类型：创建型  
 #适用场景：  
 1.类初始化消耗较多资源。  
 2.new产生的一个对象需要非常繁琐的过程（数据准备，访问权限）。  
 3.构造函数比较复杂。  
 4.循环体中生产大量对象时。  
  
  
 #注意事项：1.  
  
 #优点：1.原型模式比直接new一个对象的性能高；  
 2.简化创建过程；  
  
 #缺点：1.必须配备克隆方法，（如果不重写Clone方法，无法使用）;  
 2.对克隆复杂对象或克隆出的对象进行复杂改造时，容易引入风险；  
 3.深拷贝，浅拷贝要运用得当；  
  
 举例： 深克隆，浅克隆

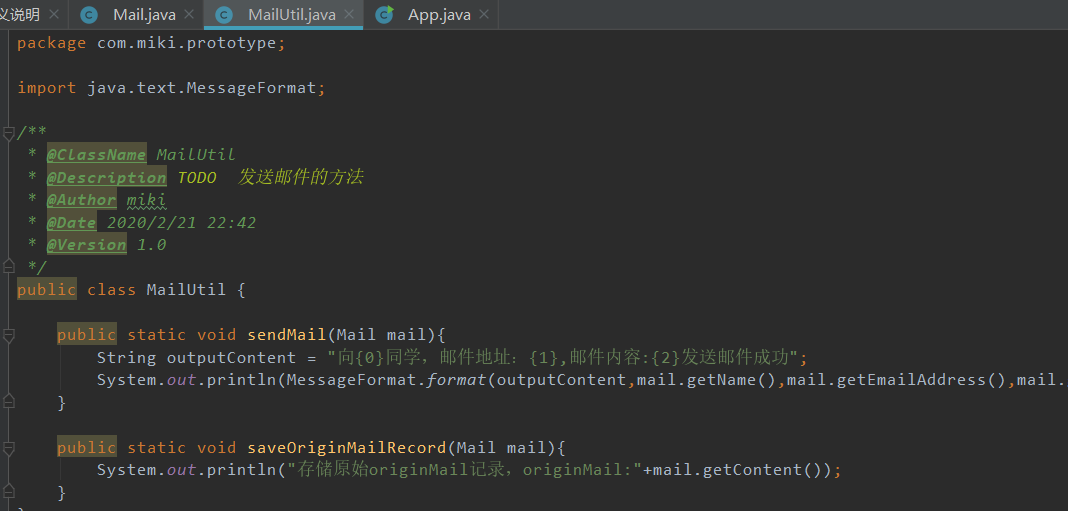
### Code：

**V1：不采用原型模式出现的问题：**

创建邮件实体类，用作存储信息实体



1. 定义一个mail工具类，具备发送邮件，并能够保存原始邮件模板；



1. 指定一个for循环，向10000人中中奖的人发送邮件信息：



**V2：采用原型模式解决该需求：**

### UML：

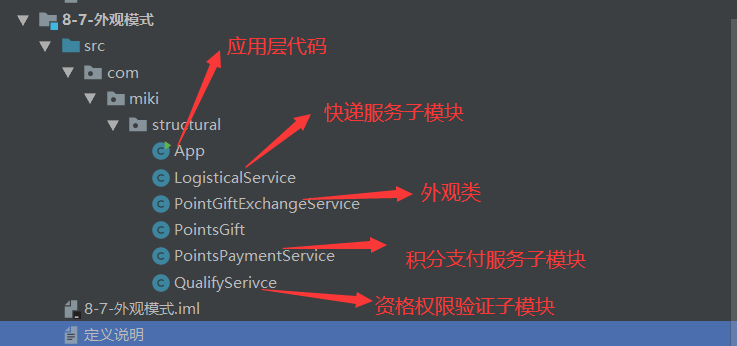
### 源码解析：

Jdk源码中：HashMap，ArrayList等源码都使用了Cloneable接口

## 7：外观模式

定义: 又叫门面模式，提供了一个统一的接口，用来访问子系统中的一群接口;  
 外观模式定义了一个高层接口，让子系统更容易利用;  
  
 #类型：结构型  
  
 #适用场景：  
 1.子系统越来越复杂，增加外观模式，提供简单调用接口；  
 2.构建多层系统结构，利用外观对象作为每层的入口，简化层间调用；  
  
 #注意事项：  
  
 #优点：1.简化了调用过程，无需了解深入子系统，防止带来风险；  
 2.减少了系统依赖，松散耦合；  
  
 #缺点：1.增加子系统，扩展子系统行为容易引入风险;  
 2.不符合开闭原则；  
  
 举例：  
 外观模式经常配合配置使用的设计模式：  
 1.外观模式 && 中介者模式 ：外观模式关注的是外界和子系统之间的交互，中介者模式关注 的是子系统内部之间的交互。  
  
 2.外观模式 && 单例模式：通常可以将外观模式中的外观对象做成单例模式使用  
  
 3.外观模式 && 抽象工厂模式：外观类可以通过抽象工厂获取子系统的实例，这样子系统可以在内部对外观类进行屏蔽

### Code：



### UML：

又叫门面模式，提供了一个统一的接口，用来访问子系统中的一群接口;  
 外观模式定义了一个高层接口，让子系统更容易利用;让应用层代码只和外观类交互。



### 源码解析：

## 8：装饰者模式

### UML：

