1. 反射与工厂设计模式

如果想要进行对象的实例化除了可以使用new，还可以利用反射机制来实现，于是此时一定会思考一个问题：

·为什么一定要提供反射的实例化？

·到底是使用new，还是反射机制实例化对象？

如果想要很好的理解此类问题，最好的解释方案就是工厂设计模式，工厂设计模式最大的特点就是:客户端的程序类不直接牵扯到对象实例化管理，只与接口发生关联，通过工厂类获取指定接口的实例化对象。

范例:传统的工厂设计模式

如果直接使用IMessage message = new NetMessage();则会一定会出现耦合问题，因为一个接口不可能只有一个子类，而new NetMessage（）则将实例化的对象写死了，也就是说改程序类与NetMessage（）的关系非常紧密。

在实际的开发中，接口的主要作用是为不同层之间提供一个操作的标准。此时如果我们直接将一个子类设置为接口的实例化操作，那么一定会有耦合问题，所以我们使用工厂设计模式解决耦合问题

|  |
| --- |
| **package com.company.反射机制;  import java.util.Optional;  */\*\*  \* 反射与工厂设计模式  \* @author JackWen  \*/* public class JavaDemo2 {  public static void main(String[] args) {  Optional<IMessage> message = Factory.*getInstance*("NetMessage");  if (message != null){  message.get().send();  }  } }  interface IMessage{  public void send(); } class NetMessage implements IMessage{  @Override  public void send() {  System.*out*.println("---【发送消息】---成功发送：我是超人");  } } class CloudMessage implements IMessage{   @Override  public void send() {  System.*out*.println("---【发送云消息】：有事请留言！");  } } class Factory{  private Factory(){};  public static Optional<IMessage> getInstance(String name){  if ("NetMessage".equalsIgnoreCase(name)){  return Optional.*of*(new NetMessage());  }else if ("CloudMessage".equalsIgnoreCase(name)){  return Optional.*of*(new CloudMessage());  }  return Optional.*of*(null) ;  } }** |

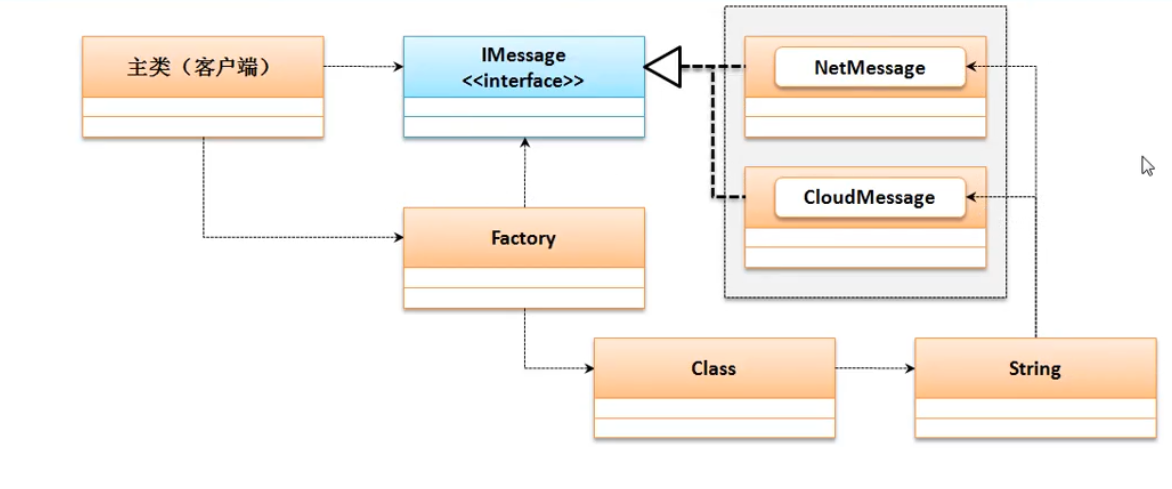
工厂设计模式最有效解决的是子类与客户端的耦合问题，但是解决的核心思想是在于提供有一个工厂类作为一个过渡端，可是随者项目的进行，IMessage可能会有更多的子类，而且随着时间的推移子类产生的可能越来越多，此时就意味着你的工厂类要一直修改（每增加一个子类就需要追加一个else if语句）。

此时最好的解决方案就是不使用关键字new ，因为关键字new 的使用必须要明确的知道这个类存在。

而newInstance()的方法只需要有一个明确表示类的字符串就可以使用。

|  |
| --- |
| **package com.company.反射机制;  import java.lang.reflect.InvocationTargetException; import java.util.Optional;  */\*\*  \* 反射与工厂设计模式  \* @author JackWen  \*/* public class JavaDemo2 {  public static void main(String[] args) {  Optional<IMessage> message = Factory.*getInstance*("com.company.反射机制.NetMessage");  if (message != null){  message.get().send();  }  } }  interface IMessage{  public void send(); } class NetMessage implements IMessage{  @Override  public void send() {  System.*out*.println("---【发送消息】---成功发送：我是超人");  } } class CloudMessage implements IMessage{   @Override  public void send() {  System.*out*.println("---【发送云消息】：有事请留言！");  } } class Factory{  private Factory(){};  public static Optional<IMessage> getInstance(String name){  IMessage instance = null ;  try {  instance = (IMessage) Class.*forName*(name).getDeclaredConstructor().newInstance();  return Optional.*of*(instance);  } catch (Exception e) {  e.printStackTrace();  }  return Optional.*of*(null);  } }** |

这个时候可以发现，利用反射机制实现的工厂设计模式，最大的优势在于接口子类的扩充将不再影响工厂的定义。

此时我们的Factory与子类之间没有任何的关系，也就没有了耦合问题。但是我们需要进一步思考，在实际项目开发之中存在大量的接口，并且这些接口都可能需要通过工厂类实例化，所以此时的工厂不应该只为一个IMessage接口服务，此时我们就需要用到我们的泛型。

|  |
| --- |
| **package com.company.反射机制;  import java.lang.reflect.InvocationTargetException; import java.util.Optional;  */\*\*  \* 反射与工厂设计模式  \* @author JackWen  \*/* public class JavaDemo2 {  public static void main(String[] args) {  Optional<IService> message = Factory.*getInstance*("com.company.反射机制.Service",IService.class);  if (message != null){  message.get().service();  }  } } interface IService{  public void service(); } class Service implements IService{  @Override  public void service() {  System.*out*.println("为你服务是我的荣幸！");  } } interface IMessage{  public void send(); } class NetMessage implements IMessage{  @Override  public void send() {  System.*out*.println("---【发送消息】---成功发送：我是超人");  } } class CloudMessage implements IMessage{   @Override  public void send() {  System.*out*.println("---【发送云消息】：有事请留言！");  } } class Factory{  private Factory(){};  */\*\*  \* 获取接口实例化对象  \* @param name 接口的子类  \* @param clazz 描述的是一个接口的类型  \* @return 如果子类存在则返回指定接口实例化对象  \*/* public static<T> Optional<T> getInstance(String name,Class<T> clazz){  T instance = null ;  try {  instance = (T) Class.*forName*(name).getDeclaredConstructor().newInstance();  return Optional.*of*(instance);  } catch (Exception e) {  e.printStackTrace();  }  return Optional.*of*(null);  } }** |

这样的工厂设计模式才是一个耦合度低可重用性高的工厂。