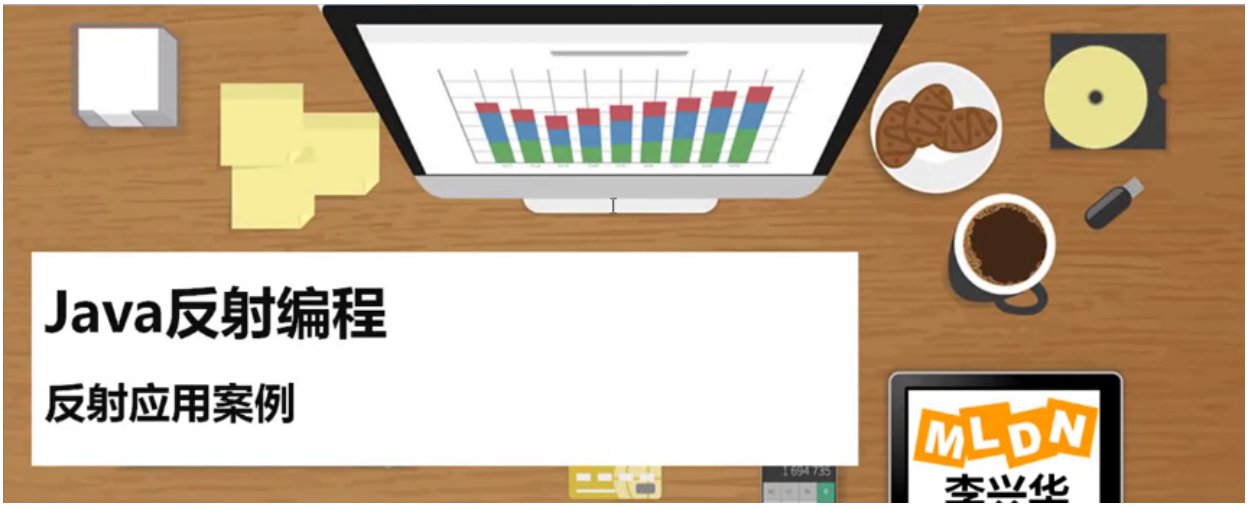


博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

经过一系列的分析之后可以发现虽然获得了Class类的实例化对象，但是依然觉得这个对象获取的意义不是很大，所以为了进一步帮助大家理解反射的核心意义所在，下面将通过几个案例进行程序的说明（都是在实际开发之中一定会使用到的）。

■反射实例化对象

获取Class对象之后最大的意义实际上并不是在于只是一个对象的实例化操作形式，更重要的是Class类里面提供有一个对象的反射实例化方法（代替了关键字new）：

·在JDK1.9以前的实例化：public T newInstance() throws InstantiationException,IllegalAccess Exception;

·在JDK1.9之后的实例化： clazz.getDeclaredConstructor().newInstance();

范例：通过newInstance()方法实例化Person类对象


```
package cn.mldn.vo;
public class Person {
    // 任何情况下如果要实例化类对象则一定要调用类中的构造方法
    public Person() {     // 无参构造方法
        System.out.println("********** Person类构造方法 **********");
    }
    @Override
    public String toString() {
        return "我是一个人，一个脱离了低级趣味的好人！";
    }
```

```
}
```

```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Class.forName("cn.mldn.vo.Person") ;
		Object obj = cls.newInstance() ;	// 实例化对象，JDK 1.9后被废除了
		System.out.println(obj); // 输出对象调用toString()方法
	}
}
```

程序执行结果：	【cls.newInstance()、**关键字new**】********** Person类构造方法 ********** 【System.out.println(obj)】我是一个人，一个脱离了低级趣味的好人！

现在通过反射实现的对象实例化处理，依然要调用类中的无参构造方法，其本质等价于“类 对象 = new 类()”，也就是说相当于隐含了关键字new，而直接使用字符串进行了替代。

范例：从JDK1.9之后newInstance()被替代了

```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Class<?> cls = Class.forName("cn.mldn.vo.Person") ;
		Object obj = cls.getDeclaredConstructor().newInstance() ;
		System.out.println(obj); // 输出对象调用toString()方法
	}
}
```

因为默认的Class类中的newInstance()方法只能够调用无参构造，所以很多开发者会认为其描述的不准确，于是将其变换了形式（构造方法会讲解）。

■反射与工厂设计模式

如果要想进行对象的实例化处理除了可以使用关键字new之外，还可以使用反射机制来完成，于是此时一定会思考一个问题：为什么要提供有一个反射的实例化？那么到底是使用关键字new还是使用反射呢？

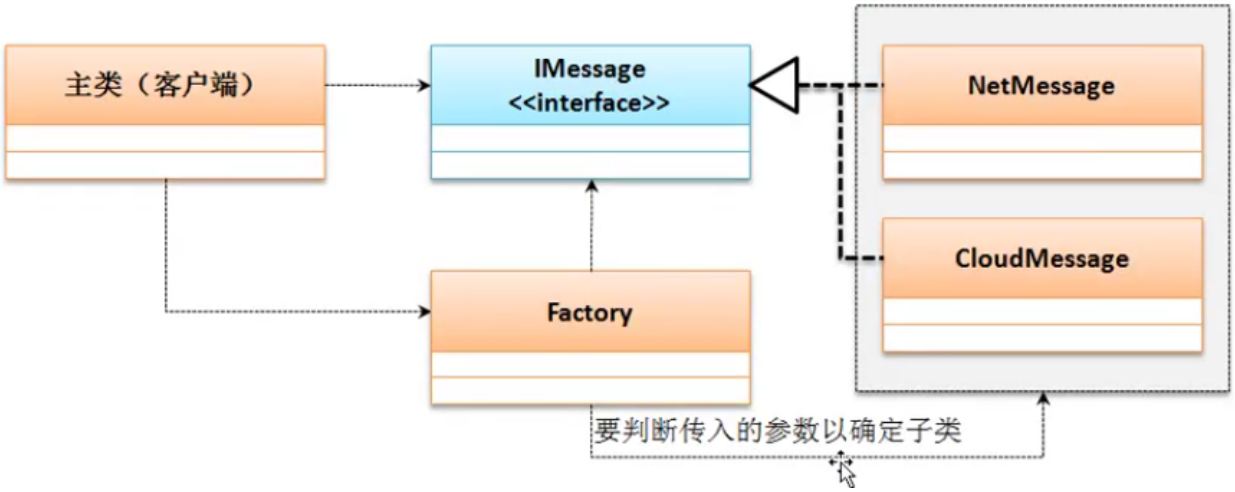
如果要想更好的理解此类问题，最好的解释方案就是通过工厂设计模式来解决。工厂设计模式的最大特点：客户端的程序类不直接牵扯到对象的实例化管理，只与接口发生关联，通过工厂类获取指定接口的实例化对象。

范例：传统工厂设计模式

```
package cn.mldn.demo;
interface IMessage {
	public void send() ; // 消息发送
}
class NetMessage implements IMessage {
	public void send() {
```

```
            System.out.println("【网络消息发送】www.mldn.cn");
        }
}
public class JavaAPIDemo {
        public static void main(String[] args) throws Exception {
            IMessage msg = new NetMessage() ;    // 如果直接实例化则一定会有耦合问题
        }
}
```

在实际的开发之中，接口的主要作用是为不同的层提供有一个操作的标准。但是如果此时直接将一个子类设置为接口实例化操作，那么一定会有耦合问题，所以使用了工厂设计模式来解决此问题。



范例：利用工厂设计模式解决

```
package cn.mldn.demo;
public class JavaAPIDemo {
        public static void main(String[] args) throws Exception {
            IMessage msg = Factory.getInstance("netmessage") ;
            msg.send();
        }
}
interface IMessage {
        public void send() ; // 消息发送
}
class NetMessage implements IMessage {
        public void send() {
            System.out.println("【网络消息发送】www.mldn.cn");
        }
}
class Factory {
        private Factory() {}  // 没有产生实例化对象的意义，所以构造方法私有化
        public static IMessage getInstance(String className) {
            if ("netmessage".equalsIgnoreCase(className)) {
                return new NetMessage() ;
            }
            return null ;
        }
}
```

此种工厂设计模式属于静态工厂设计模式，也就是说如果现在要追加一个子类，则意味着工厂类一定要做出修改，因为如果不追加这种判断是无法获取指定接口对象的。

范例：为IMessage追加一个子类

```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		IMessage msg = Factory.getInstance("cloudmessage") ;
		msg.send();
	}
}
interface IMessage {
	public void send() ; // 消息发送
}
class CloudMessage implements IMessage {
	@Override
	public void send() {
		System.out.println("【云消息】www.mldnjava.cn");
	}
}
class NetMessage implements IMessage {
	public void send() {
		System.out.println("【网络消息发送】www.mldn.cn");
	}
}
class Factory {
	private Factory() {}  // 没有产生实例化对象的意义，所以构造方法私有化
	public static IMessage getInstance(String className) {
		if ("netmessage".equalsIgnoreCase(className)) {
			return new NetMessage() ;
		} else if ("cloudmessage".equalsIgnoreCase(className)) {
			return new CloudMessage() ;
		}
		return null ;
	}
}
```

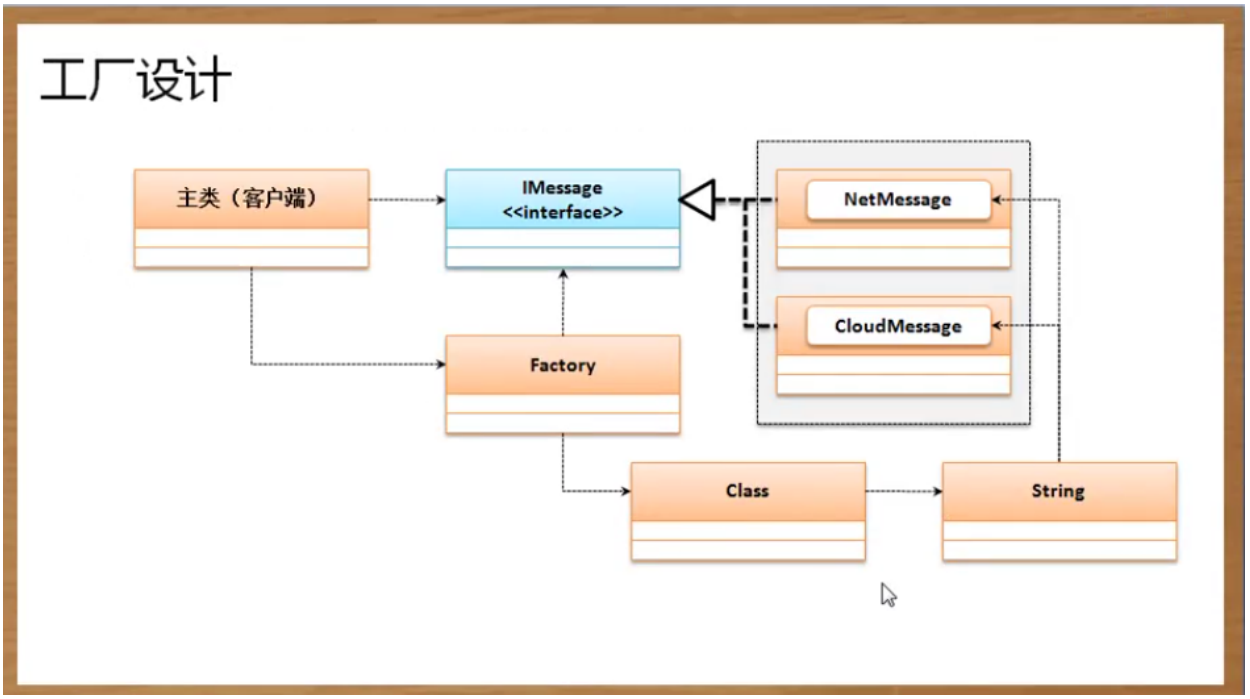
工厂设计模式最有效解决的是子类与客户端的耦合问题，但是解决的核心思想是在于提供有一个工厂类作为过渡端，但是随着项目的进行，你的IMessage接口有可能会有更多子类，而且随着时间的随意子类产生的可能越来越多，那么此时就意味着，你的工厂类永远都要进行修改，并且永无停止之日。

那么这个时候最好的解决方案就是不适用关键字new来完成，因为关键字new在使用的时候需要有一个明确的类存在。而newInstance()方法只需要有一个明确表示类名称的字符串即可应用。

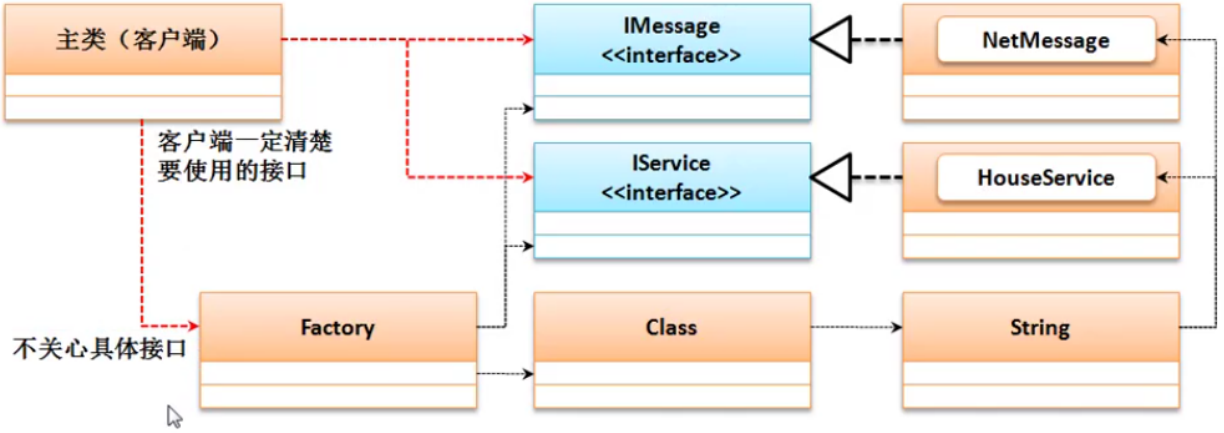
```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		IMessage msg = Factory.getInstance("cn.mldn.demo.NetMessage") ;
```

```
            msg.send();
        }
}
class Factory {
        private Factory() {}  // 没有产生实例化对象的意义，所以构造方法私有化
        public static IMessage getInstance(String className) {
            IMessage instance = null ;
            try {
                instance = (IMessage)
Class.forName(className).getDeclaredConstructor().newInstance() ;
            } catch (Exception e) {
                e.printStackTrace();
            }
            return instance ;
        }
}
interface IMessage {
        public void send() ; // 消息发送
}
class CloudMessage implements IMessage {
        @Override
        public void send() {
            System.out.println("【云消息】www.mldnjava.cn");
        }
}
class NetMessage implements IMessage {
        public void send() {
            System.out.println("【网络消息发送】www.mldn.cn");
        }
}
```

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



但是现在依然需要进一步思考，因为在实际的项目开发过程之中有可能会存在有大量的接口，并且这些接口都可能需要通过工厂类实例化，所以此时的工厂设计模式不应该只为一个IMessage接口服务，而应该变为为所有的接口服务。




```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		IMessage msg =
Factory.getInstance("cn.mldn.demo.NetMessage",IMessage.class) ;
		msg.send();
		IService service =
Factory.getInstance("cn.mldn.demo.HouseService",IService.class) ;
		service.service();
	}
}
class Factory {
	private Factory() {}  // 没有产生实例化对象的意义，所以构造方法私有化
	/**
	 * 获取接口实例化对象
	 * @param className 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
	 */
	@SuppressWarnings("unchecked")
	public static <T> T getInstance(String className,Class<T> clazz) {
		T instance = null ;
		try {
			instance = (T)
Class.forName(className).getDeclaredConstructor().newInstance() ;
		} catch (Exception e) {
			e.printStackTrace();
		}
		return instance ;
	}
}
interface IService {
	public void service() ;
}
class HouseService implements IService {
```

```
    @Override
    public void service() {
        System.out.println("【服务】为您的住宿提供服务。");
    }
}
interface IMessage {
    public void send() ; // 消息发送
}
class NetMessage implements IMessage {
    public void send() {
        System.out.println("【网络消息发送】www.mldn.cn");
    }
}
```

此时的将不再受限于指定的接口，可以为所有的接口提供实例化服务。

■反射与单例设计模式

单例设计模式的核心本质在于：类内部的构造方法私有化，在类的内部产生实例化对象之后通过static方法获取实例化对象，进行类中的结构调用，单例设计模式一共有两类：懒汉式、饿汉式，饿汉式的单例是不再本次讨论范围之内的，主要来讨论懒汉式单例设计模式。

范例：观察懒汉式单例设计模式的问题

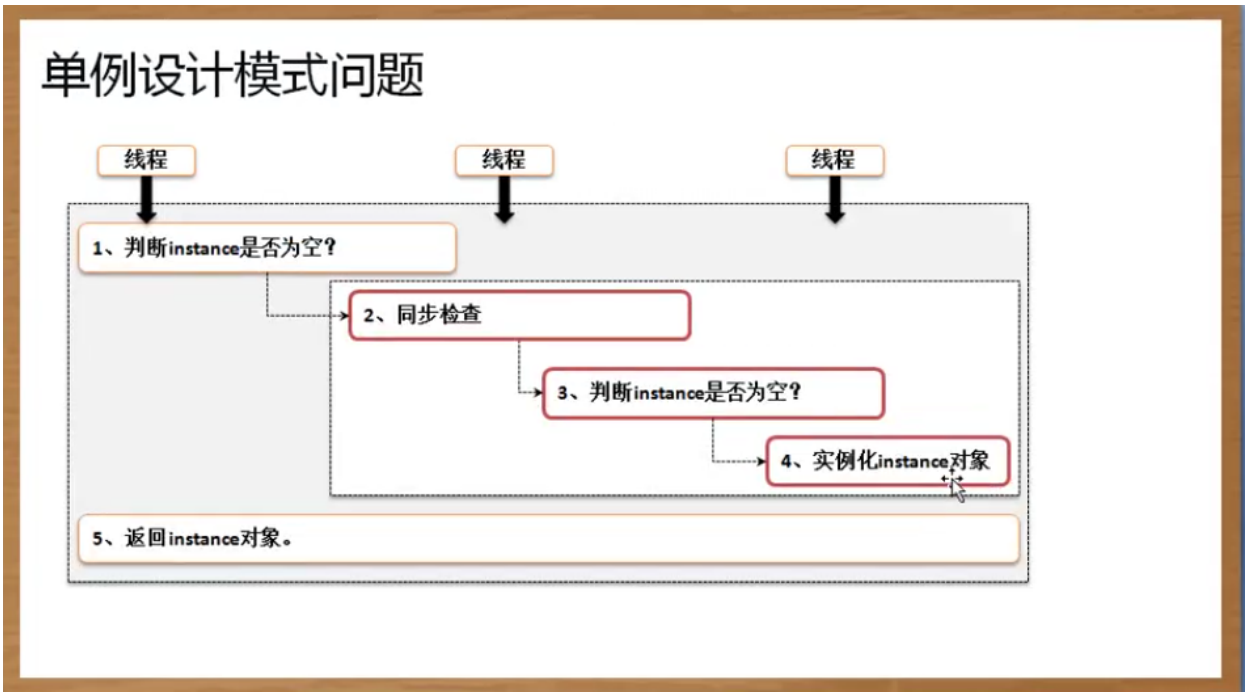
```
package cn.mldn.demo;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        Singleton sinA = Singleton.getInstance() ;
        sinA.print();
    }
}
class Singleton {
    private static Singleton instance = null ;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton() ;
        }
        return instance ;
    }
    public void print() {
        System.out.println("www.mldn.cn");
    }
}
```

```
//观察问题所在的程序
package cn.mldn.demo;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        for (int x = 0 ; x < 3 ; x ++) {
```

```
                new Thread(()->{
                    Singleton.getInstance().print();
                },"单例消费端-" + x).start();
            }
        }
}
class Singleton {
    private static Singleton instance = null ;
    private Singleton() {
        System.out.println("【" + Thread.currentThread().getName() + "】****** 实例化
Singleton类对象 ******");
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton() ;
        }
        return instance ;
    }
    public void print() {
        System.out.println("www.mldn.cn");
    }
}
```

程序运行结果：	【单例消费端-1】****** 实例化Singleton类对象 ****** www.mldn.cn 【单例消费端-0】****** 实例化Singleton类对象 ****** www.mldn.cn 【单例消费端-2】****** 实例化Singleton类对象 ****** www.mldn.cn

单例设计模式的最大特点是在整体的运行过程之中只允许产生一个实例化对象，这个时候会发现当有了若干线程之后实际上当前的程序就可以产生多个实例化对象了，那么此时就不是单例设计模式。此时问题造成的关键在于代码本身出现了不同步的情况，而要想解决的关键核心就在于需要进行同步处理，同步自然可以想到synchronized关键字。



范例：修改getInstance()方法进行同步处理

```
public static synchronized Singleton getInstance() {
	if (instance == null) {
		instance = new Singleton() ;
	}
	return instance ;
}
```

这个时候的确是进行了同步处理，但是这个同步的代价有些大，因为效率会低。因为整体代码块里面实际上只有一块部分需要进行同步处理，instance对象的实例化处理部分，在这样的情况下会发现同步加的有点草率了。

范例：更加合理的进行同步处理

```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		for (int x = 0 ; x < 3 ; x ++) {
			new Thread(()->{
				Singleton.getInstance().print();
			},"单例消费端-" + x).start();
		}
	}
}
class Singleton {
	private static volatile Singleton instance = null ;
	private Singleton() {
		System.out.println("【" + Thread.currentThread().getName() + "】****** 实例化
Singleton类对象 ******");
	}
	public static Singleton getInstance() {
		if (instance == null) {
			synchronized (Singleton.class) {
				if (instance == null) {
```

```
                        instance = new Singleton() ;
                    }
                }
            }
            return instance ;
        }
        public void print() {
            System.out.println("www.mldn.cn");
        }
}
```

面试题：请编写单例设计模式

- 【100%】直接编写一个饿汉式的单例设计模式，并且实现实现构造方法私有化；
- 【120%】在Java中哪里使用到单例设计模式了？runtime类、Spring框架；
- 【200%】懒汉式单例设计模式的问题？