2、具体内容

当可以灵活的使用抽象类和接口进行设计的时候，那么基本上就表示面向对象的概念理解了。这一步是需要大量的代码累积的。

■接口的基本定义

抽象类与普通类相比最大的优势在于：可以实现对子类覆写方法的控制，但是在抽象类里面可能依然会保留有一些普通方法，而普通方法里面可能会涉及到一些安全或者隐私的操作问题，那么这样在进行开发的过程之中，如果想要对外部隐藏全部的实现的细节，则就可以通过接口来进行描述。

接口可以理解为一个纯粹的抽象类（最原始的定义接口之中是只包含有抽象方法与全局常量的），但是从JDK1.8开始由于引入了Lambda表达式的概念，所以接口的定义也得到了加强，除了抽象方法与全局常量之外，还可以定义普通方法或静态方法。如果从设计本身的角度来讲，接口之中的组成还是应该以抽象方法和全局变量为主。

在Java中接口主要使用interface关键字来进行定义。

范例：定义一个接口

```
// 由于类名称与接口名称的定义要求相同，所以为了区分出接口接口名称前往往会加入字母
I（interface简写）
interface IMessage {       // 定义了一个接口
        public static final String INFO = "www.mldn.cn" ; // 全局常量
        public abstract String getInfo() ; // 抽象方法
}
```

但是现在很明显的问题出现了，此时的接口肯定无法直接产生实例化对象，所以对于接口的使用原则如下：

- 接口需要被子类实现（implements），一个子类可以实现多个父接口；
- 子类（如果不是抽象类）那么一定要覆写接口之中的全部抽象方法；

- 接口对象可以利用子类对象的向上转型进行实例化；

范例：定义接口子类

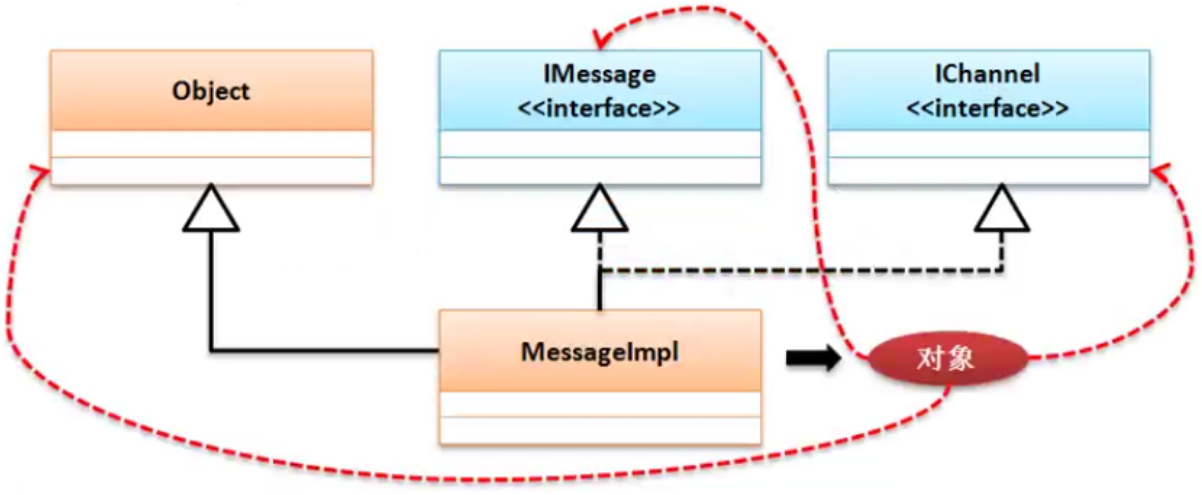
```
// 由于类名称与接口名称的定义要求相同，所以为了区分出接口接口名称前往往会加入字母
I（interface简写）
interface IMessage {       // 定义了一个接口
    public static final String INFO = "www.mldn.cn" ; // 全局常量
    public abstract String getInfo() ; // 抽象方法
}
class MessageImpl implements IMessage {  // 实现了接口
    public String getInfo() {
        return "得到一个消息，秘密的消息，有人胖了（不是我）。" ;
    }
}
public class JavaDemo {
    public static void main(String args[]) {
        IMessage msg = new MessageImpl() ;
        System.out.println(msg.getInfo()) ;
        System.out.println(IMessage.INFO) ;      // 直接输出全局常量
    }
}
```

以上是接口的基本使用，但是在Java里面之所有使用接口主要的目的是一个子类可以实现多个接口，利用接口可以实现多继承的概念。

范例：观察子类实现多个父接口

```
interface IMessage {       // 定义了一个接口
    public static final String INFO = "www.mldn.cn" ; // 全局常量
    public abstract String getInfo() ; // 抽象方法
}
interface IChannel {
    public abstract boolean connect() ;       // 定义抽象方法
}
class MessageImpl implements IMessage,IChannel { // 实现了接口
    public String getInfo() {
        if (this.connect()) {
            return "得到一个消息，秘密的消息，有人胖了（不是我）。" ;
        }
        return "通道创建失败，无法获取消息。" ;
    }
    public boolean connect() {
        System.out.println("消息发送通道已经成功建立。") ;
        return true ;
    }
}
public class JavaDemo {
    public static void main(String args[]) {
        IMessage msg = new MessageImpl() ;
        System.out.println(msg.getInfo()) ;
    }
}
```

但是这个时候就需要考虑一个世纪的情况了，关于对象的转型问题了。



此时MessageImpl子类的对象可以任意的实现父接口的转换。

范例：观察转换

```
public class JavaDemo {
	public static void main(String args[]) {
		IMessage msg = new MessageImpl() ;
		IChannel chl = (IChannel) msg ;
		System.out.println(chl.connect()) ;
	}
}
```

由于MessageImple子类实现了IMessage与IChannel两个接口，所以这个子类可以是这两个接口任意一个接口的实例，那么就表示此时这两个接口实例之间是可以转换的。

在java程序里面接口是不允许去继承父类的，所以接口绝对不会是Object的子类，但是根据之前的分析可以发现，MessageImpl是不是Object的子类，所以接口一定可以通过Object接口。

范例：观察Object与接口转换

```
interface IMessage {	// 定义了一个接口
	public static final String INFO = "www.mldn.cn" ; // 全局常量
	public abstract String getInfo() ; // 抽象方法
}
interface IChannel {
	public abstract boolean connect() ;	// 定义抽象方法
}
class MessageImpl implements IMessage,IChannel {	// 实现了接口
	public String getInfo() {
		if (this.connect()) {
			return "得到一个消息，秘密的消息，有人胖了（不是我）。" ;
		}
		return "通道创建失败，无法获取消息。" ;
	}
	public boolean connect() {
		System.out.println("消息发送通道已经成功建立。") ;
		return true ;
```

```
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		IMessage msg = new MessageImpl() ;
		Object obj = msg ; // 向上转型
		IChannel chan = (IChannel) obj ;
		System.out.println(chan.connect()) ;
	}
}
```

Object类对象可以接收所有数据类型，包括基本数据类型、类对象、接口对象、数组。

由于接口描述的是一个公共的定义标准，所以在接口之中所有的抽象方法的访问权限都为public，也就是说写不写都是一样的，例如：下面两个接口本质是完全相同的；

完整定义：	**简化定义：**

interface IMessage {　　// 定义了一个接口
　　public static final String INFO = "www.mldn.cn" ;　　// 全局常量
　　public abstract String getInfo() ; // 抽象方法
}
	interface IMessage {　　// 定义了一个接口
	　　final String INFO = "www.mldn.cn" ;// 全局常量
	　　String getInfo() ; // 抽象方法
	}

方法不写访问权限也是public，不是default，所以覆写的时候只能够使用public。

接口虽然已经可以成功的进行了定义，但是千万不要忽略，在实际的开发过程之中，实现接口的有可能是抽象类，一个抽象类可以实现多个接口，而一个普通类只能够继承一个抽象类并且可以实现多个父类接口，但是要求先继承后实现。

范例：子类继承抽象类并且实现接口

```
interface IMessage {　　// 定义了一个接口
	public static final String INFO = "www.mldn.cn" ;
	public abstract String getInfo() ;
}
interface IChannel {
	public abstract boolean connect() ;　　// 定义抽象方法
}
abstract class DatabaseAbstract {  // 定义一个抽象类
	// 接口中的abstract可以省略，抽象类中不允许省略
	public abstract boolean getDatabaseConnection() ;
}
class MessageImpl extends DatabaseAbstract implements IMessage,IChannel { // 实现了接口
	public String getInfo() {
		if (this.connect()) {
			if (this.getDatabaseConnection()) {
				return "数据库中得到一个消息，秘密的消息，有人胖了（不是我）。" ;
			} else {
				return "数据库消息无法访问。" ;
			}
		}
```

```
            return "通道创建失败，无法获取消息。" ;
        }
        public boolean connect() {
            System.out.println("消息发送通道已经成功建立。") ;
            return true ;
        }
        public boolean getDatabaseConnection() {
            return true ;
        }
}
public class JavaDemo {
        public static void main(String args[]) {
            IMessage msg = new MessageImpl() ;
            System.out.println(msg.getInfo()) ;
        }
}
```

虽然接口无法去继承一个父类，但是一个接口却可以通过extends继承若干个父接口，此时称为接口多继承。

范例：实现接口多继承

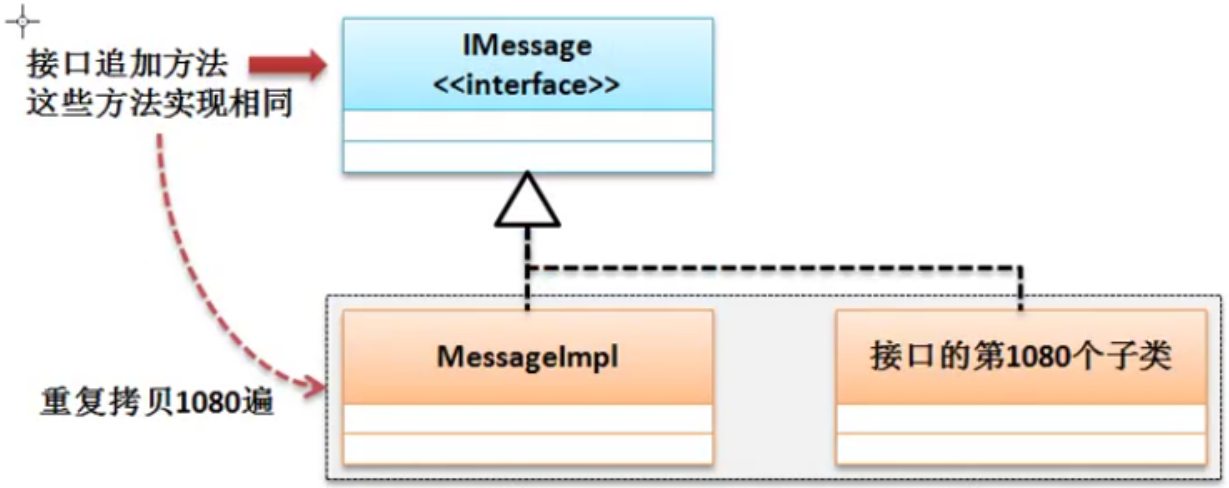
```
interface IMessage {
        public abstract String getInfo() ;
}
interface IChannel {
        public boolean connect() ;
}
// extends在类继承上只能够继承一个父类，但是接口上可以继承多个
interface IService extends IMessage,IChannel {     // 接口多继承
        public String service() ;
}
class MessageService implements IService {
        public String getInfo() {
            return null ;
        }
        public boolean connect() {
            return true ;
        }
        public String service() {
            return "获取消息服务。" ;
        }
}
```

在实际的开发之中，接口的使用往往有三种形式：

- 进行标准设置；
- 表示一种操作的能力；
- 暴露远程方法视图，这个一般都在RPC分布式方法之中使用。

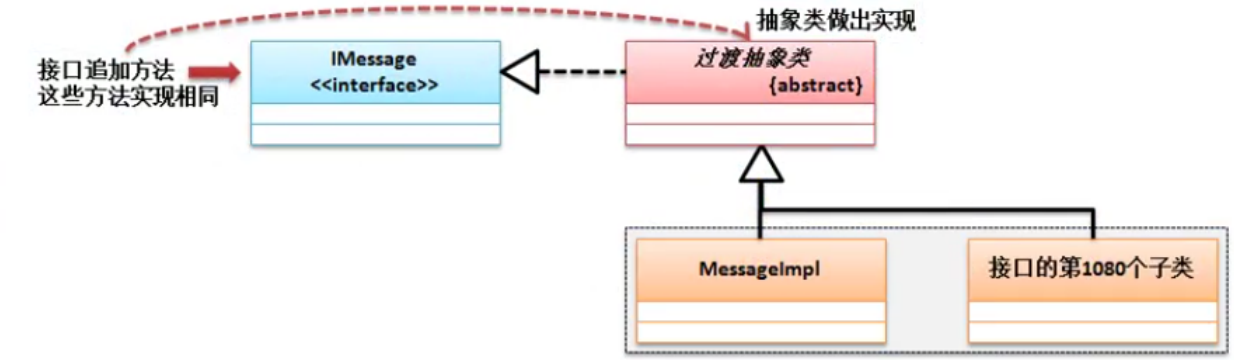
■接口定义加强

接口最早的主要特点是全部由抽象方法和全局常量所组成，但是如果你的项目设计不当，就有可能出现一种非常严重的问题。



一直在强调该操作是属于结构设计不当的结果，那么在最初的时候任何人都不敢保证接口设计的足够完善，所以在这样的情况下，为了方便子类的修改，往往不会让子类直接实现接口，而是中间追加一个过渡的抽象类。

接口不当设计



但是从JDK1.8之后开始，为了解决接口设计的缺陷，所以在接口之中允许开发者定义普通方法。

范例：观察普通方法定义

```
interface IMessage {
	public String message() ;
	public default boolean connect() {	// 方法是一个公共方法，都具备
		System.out.println("建立消息的发送通道。") ;
		return true ;
	}
}
class MessageImpl implements IMessage {
	public String message() {
		return "www.mldn.cn" ;
	}
}
public class JavaDemo {
```

```
    public static void main(String args[]) {
        IMessage msg = new MessageImpl() ;
        if (msg.connect()) {
            System.out.println(msg.message()) ;
        }
    }
}
```

接口中的普通方法必须追加default的声明，但是需要提醒的是，该操作属于挽救功能，所以如果不是必须的情况下，不应该作为你设计的首选。

除了可以追加普通方法之外，接口里面也可以定义static方法了，而static方法就可以通过接口直接调用。

范例：在接口中定义static方法

```
interface IMessage {
    public String message() ;
    public default boolean connect() { // 方法是一个公共方法，都具备
        System.out.println("建立消息的发送通道。") ;
        return true ;
    }
    public static IMessage getInstance() {
        return new MessageImpl() ;   // 获得子类对象
    }
}
class MessageImpl implements IMessage {
    public String message() {
        if (this.connect()) {
            return "www.mldn.cn" ;
        }
        return "没有消息发送。" ;
    }
}
public class JavaDemo {
    public static void main(String args[]) {
        IMessage msg = IMessage.getInstance() ;
        System.out.println(msg.message()) ;
    }
}
```

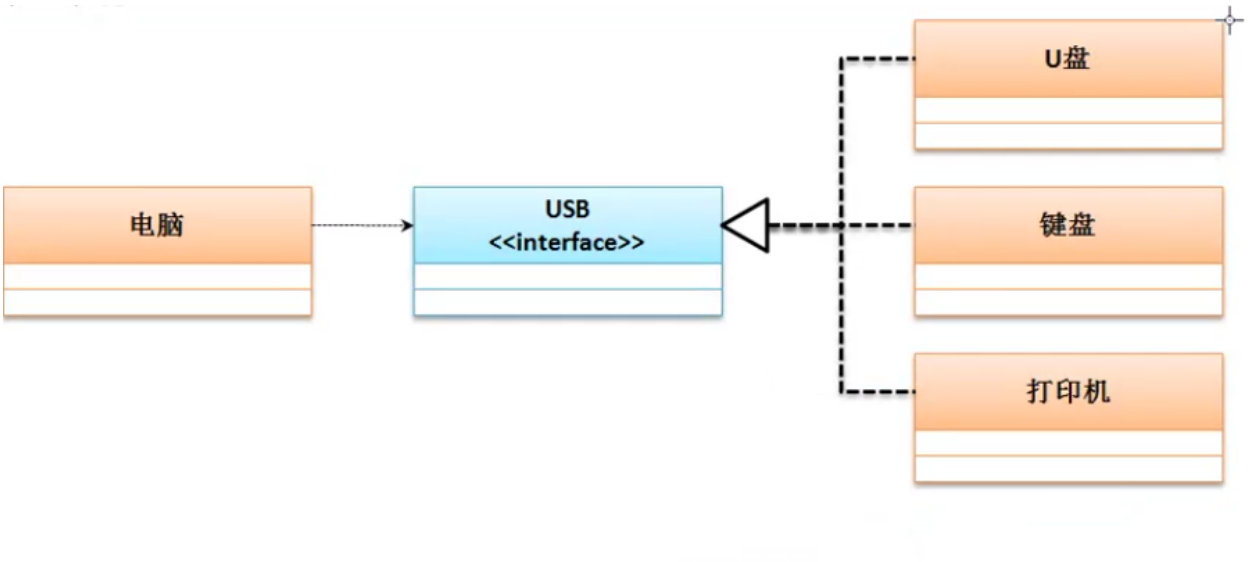
如果现在真的可以在接口里面定义普通方法或static方法，那么这个功能就已经可以却带抽象类了，但是不应该将这两个组成作为接口的主要设计原则。所写的代码里面还是应该奉行：接口就是抽象方法。

■使用接口定义标准

对于接口而言在开发之中最为重要的应用就是进行标准的制定，实际上在日常的生活之中也会听见许多关于接口的名词；

如：USB接口、PCI接口、鼠标接口等等，那么这些实际上都是属于标准的应用。

以USB的程序为例，电脑上可以插入各种USB的设备，所以电脑上认的是USB标准，而不关心这个标准的具体实现类。

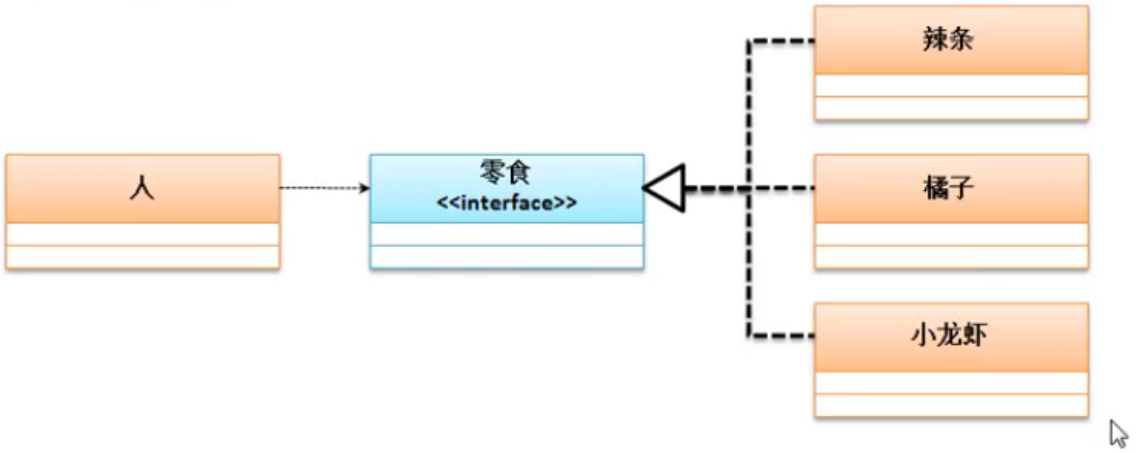



```
interface IUSB //定义USB标准
{
	public boolean check();  //检查通过可以工作
	public void work();
}
class Computer
{
	public void plugin(IUSB usb){
		if(usb.check()){
			usb.work();//开始工作
		}else{
			System.out.println("设备硬件安装出现了问题，无法使用！");
		}
	}
}
class Keyboard implements IUSB
{
	public boolean check(){
		return true;
	}
	public void work(){
		System.out.println("开始进行码字任务");
	}
}
class Print implements IUSB
{
	public boolean check(){
		return false;
	}
	public void work(){
		System.out.println("开始进行照片打印");
	}
```

```
}
public class JavaDemo {
	public static void main(String args[]) {
		Computer computer = new Computer();
		computer.plugin(new Keyboard());//插入键盘
		computer.plugin(new Print());//插入打印机
	}
}
```

而在现实的开发之中，对于标准的概念无处不在。

现实中的标准



■工厂设计模式(Factory)

对于接口而言，已经可以明确的清楚，必须有子类，并且子类可以通过对象的向上转型来获取接口的实例化对象。但是在进行对象实例化的过程之中也可能存在有设计问题。

范例：观察如下一个程序

```
interface IFood {	// 定义一个食物标准
	public void eat() ; // 吃
}
class Bread implements IFood { // 定义一种食物
	public void eat() {
		System.out.println("吃面包。") ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		IFood food = new Bread() ;
		food.eat() ;// 吃面包
	}
}
```

在本程序之中根据接口进行子类的定义，并且利用对象的向上转型进行接口对象实例化处理，而此时的程序的结构如下：

程序结构设计

客户端需要明确的知道具体的那一个子类，如果说现在面包吃腻了，需要喝牛奶了，那么客户端就要做出修改。

范例：扩展一类食物

```
interface IFood {     // 定义一个食物标准
	public void eat() ; // 吃
}
class Bread implements IFood { // 定义一种食物
	public void eat() {
		System.out.println("吃面包。") ;
	}
}
class Milk implements IFood { // 定义一种食物
	public void eat() {
		System.out.println("喝牛奶。") ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		IFood food = new Milk() ;
		food.eat() ;// 吃面包
	}
}
```

所以此时的程序就表示出现有耦合的问题，而造成耦合最直接的元凶：“关键字new”。以JVM的设计为例，Java实现可移植性的关键在于：JVM，而JVM的核心原理：利用一个虚拟机来运行Java程序，所有的程序并不与具体的操作系统有任何的关联，而是由JVM来进行匹配，所以得出结论：良好的设计应该避免耦合。

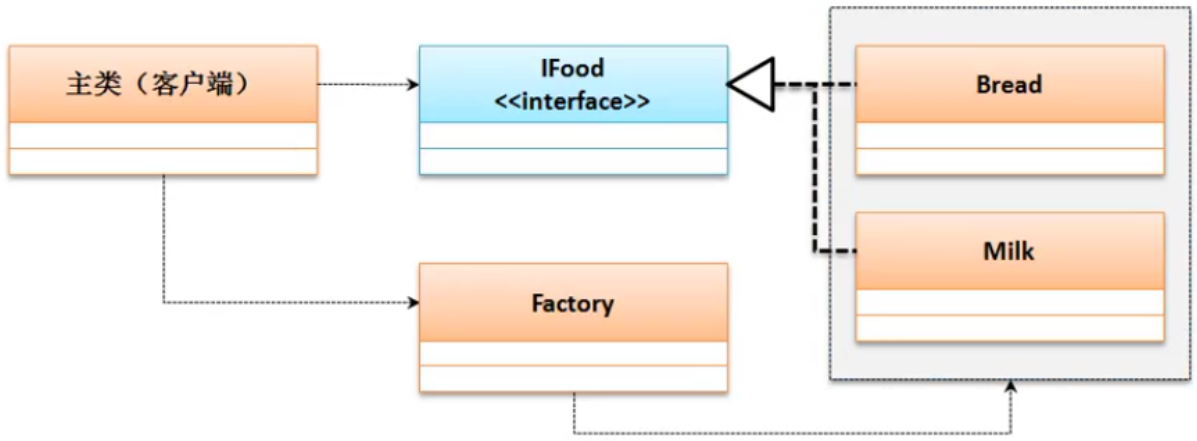
范例：工厂设计实现

```
interface IFood {     // 定义一个食物标准
	public void eat() ; // 吃
}
class Bread implements IFood { // 定义一种食物
	public void eat() {
		System.out.println("吃面包。") ;
	}
}
class Milk implements IFood { // 定义一种食物
```

```
        public void eat() {
            System.out.println("喝牛奶。") ;
        }
}
class Factory {
        public static IFood getInstance(String className) {
            if ("bread".equals(className)) {
                return new Bread() ;
            } else if ("milk".equals(className)) {
                return new Milk() ;
            } else {
                return null ;
            }
        }
}
public class JavaDemo {
        public static void main(String args[]) {
            IFood food = Factory.getInstance(args[0]) ;
            food.eat() ;// 吃面包
        }
}
```

在本程序之中，客户端程序类与IFood接口的子类没有任何的关联，所有的关联都是通过Factory类完成的，而在程序运行的时候可以通过初始化参数进行要使用的子类定义：

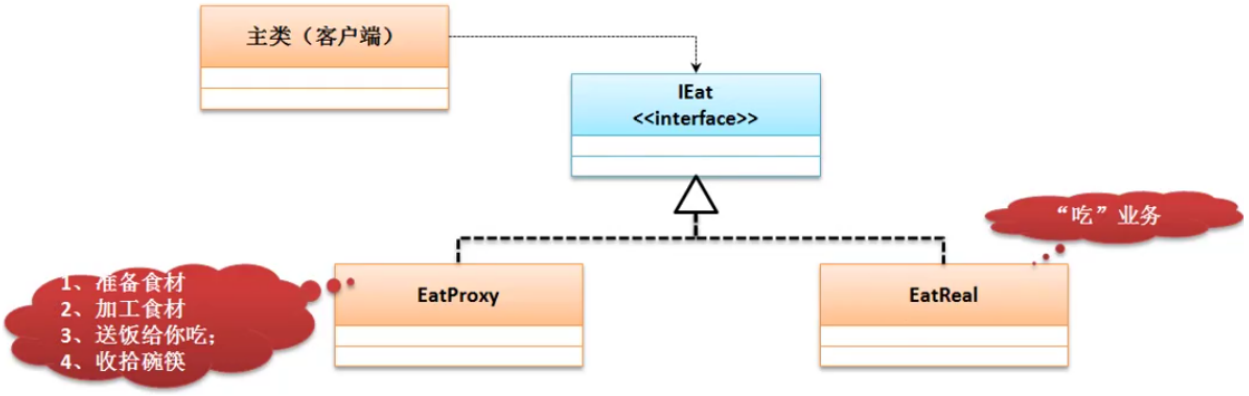
- java JavaDemo bread
- java JavaDemo milk



如果在日后进行子类扩充的时候只需要修改Factory程序类即可实现。

■代理设计模式（Proxy）

代理设计模式的主要功能是可以帮助用户将所有的开发注意力只集中在核心业务功能的处理上，例如：肚子饿了，思考的是如何可以吃到东西。



范例：实现代理设计

```
interface IEat {
	public void get() ;
}
class EatReal implements IEat {
	public void get() {
		System.out.println("【真实主题】得到一份食物，而后开始品尝美味。") ;
	}
}
class EatProxy implements IEat {	// 服务代理
	private IEat eat ; // 为吃而服务
	public EatProxy(IEat eat) {	// 一定要有一个代理项
		this.eat = eat ;
	}
	public void get() {
		this.prepare() ;
		this.eat.get() ;
		this.clear() ;
	}
	public void prepare() {	// 准备过程
		System.out.println("【代理主题】1、精心购买食材。") ;
		System.out.println("【代理主题】2、小心的处理食材。") ;
	}
	public void clear() {
		System.out.println("【代理主题】3、收拾碗筷。");
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		IEat eat = new EatProxy(new EatReal()) ;
		eat.get() ;
	}
}
```

代理设计模式的主要特点是：一个接口提供有两个子类，其中一个子类是真实业务操作类，另外一个主题是代理业务操作类，没有代理业务操作，真实业务无法进行。

■抽象类与接口的区别

在实际的开发之中可以发现抽象类和接口的定义形式是非常相似的，这一点从JDK1.8开始实际上就特别的明显了，因为在JDK1.8里面接口也可以定义default或static方法了，但是这两者依然是有着明显的定义与使用区别的。

NO	区别	抽象类	接口
01	定义关键字	abstract class 抽象类名称{}	interface 接口mingcheng{}
02	组成	构造、普通方法、静态方法、全局常量、成员、static方法	抽象方法、全局常量、普通方法、static方法
03	权限	可以使用各种权限定义	只能够使用public
04	子类使用	子类通过extends关键字可以继承一个抽象类	子类使用implements关键字可以实现多个接口
05	两者关系	抽象类可以实现若干个接口	接口不允许继承抽象类，但是允许继承多个父接口
06	使用	1、抽象类或接口必须定义子类； 2、子类一定要覆写抽象类或接口中的全部方法 3、通过子类的向上转型实现抽象类或接口对象实例化	

当抽象类和接口都可以使用的情况下优先要考虑接口，因为接口可以避免子类的单继承局限。

另外从一个正常的设计角度而言，也需要先从接口来进行项目的整体设计。