7.1 属性和操作

属性（attribute）是用来描述对象静态特征的一个数据项。

实例属性（**instance attribute**）和类属性（**class attribute**）的区别

例如：仪表类

输入电压、功率及各种规定的质量指标——类属性
编号、出厂日期、精度等实际性能参数——实例属性

操作（operation）是用来描述对象动态特征（行为）的一个动作序列 。

近义词：方法（method），服务（service）

被动操作（**passive operation**）：

只有接收到消息才能执行的操作

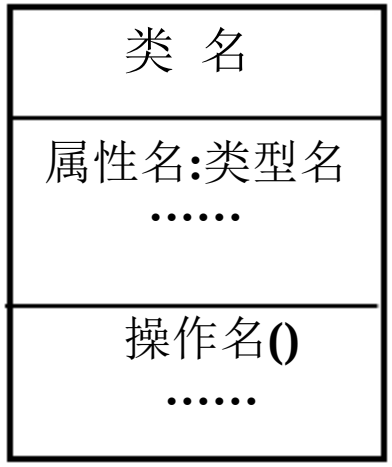
编程语言中的函数、过程等被动成分

主动操作（**active operation**）：

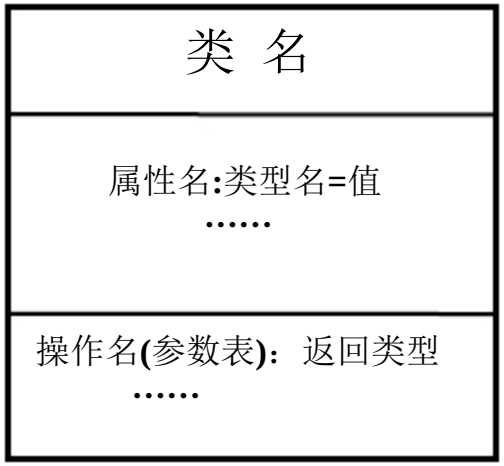
不需要接收消息就能主动执行的操作

编程语言中的进程、线程等主动成分

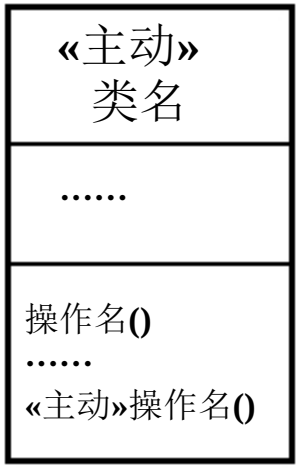
7.2 属性和操作的表示法

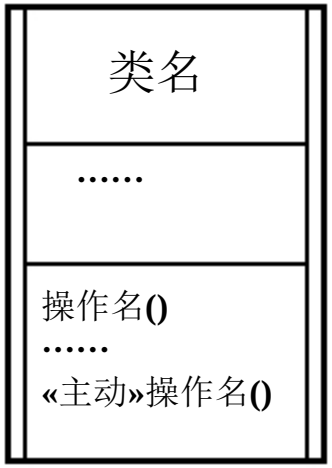


分析级细节方式



实现级细节方式





用衍型表示主动操作

7.3 定义属性

（1）策略与启发

按常识这个对象应该有哪些属性？

人→姓名、地址、出生年月

在当前的问题域中，对象应该有哪些属性？

商品→条形码

根据系统责任，这个对象应具有哪些属性？

乘客→手机号码

建立这个对象是为了保存和管理哪些信息？

物资→型号、规格、库存量

为实现操作的功能，需要增设哪些属性？

传感器（信号采集功能）→时间间隔

是否需要增加描述对象状态的属性？

设备→状态

用什么属性表示关联和聚合？

课程→任课教师，汽车→发动机

（2）审查与筛选

是否体现了以系统责任为目标的抽象

例：书→重量？

是否描述对象本身的特征

例：课程→电话号码？

是否可通过继承得到？

是否可从其他属性直接导出？

（3）推迟到OOD考虑的问题

规范化问题

对象标识

性能问题

（4）属性的命名与定位

命名：原则与类的命名相同

定位：针对所描述的对象，适合全部对象实例

7.4 定义操作

（1）对象行为分类

系统行为——不需要定义

例：创建、删除、复制、转存

封装行为引起的附加行为——不需要定义

例：读、写属性值

对象自身的行为——努力发现

计算或监控

（2）策略与启发

考虑系统责任

有哪些功能要求在本对象提供？

考虑问题域

对象在问题域对应的事物有哪些行为？

分析对象状态

对象状态的转换是由哪些操作引起的？

追踪操作的执行路线

模拟操作的执行，并在整个系统中跟踪

（3）审查与调整

审查对象的每个操作是否真正有用

是否直接提供系统责任所要求的某项功能？

或者 响应其它操作的请求间接地完成这种功能的某些局部操作？

调整——取消无用的操作

审查操作是不是高内聚的

一个操作应该只完成一项单一的、完整的功能

调整——拆分 或 合并

（4）认识对象的主动行为

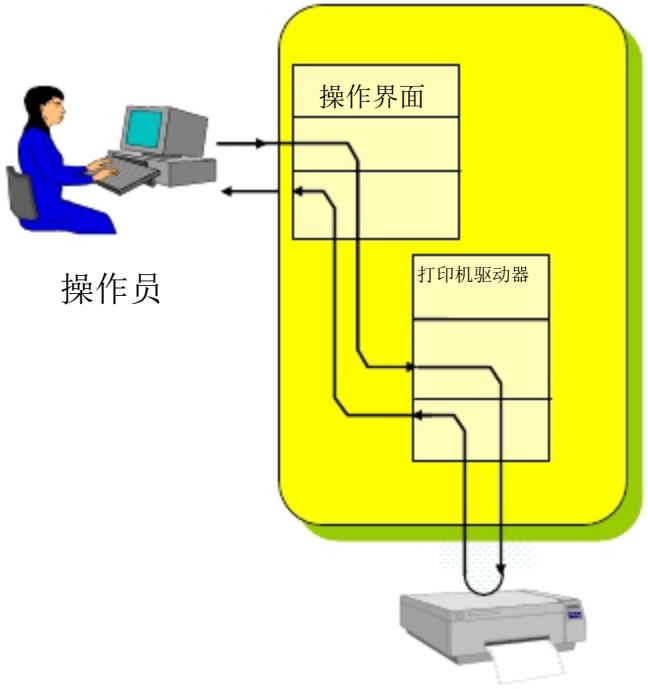
考虑问题域

对象行为是被引发的，

还是主动呈现的？

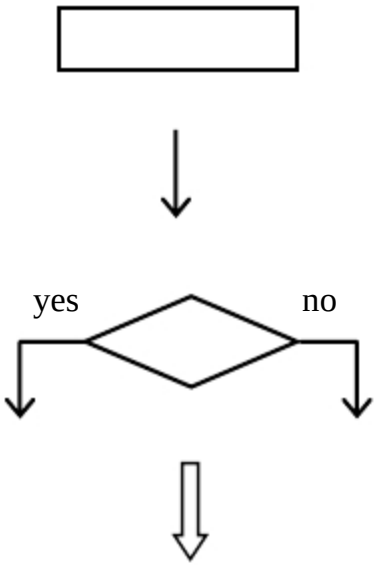
与参与者直接交互的对象操作

操作执行路线逆向追踪



（5）操作过程描述
——可采用流程图或活动图

流程图：



动作陈述框，在框内填写要执行的动作。

转接，用于连接各个框，表示它们之间的转接关系。

条件判断框，给出一个判断条件。

入口/出口标记，指出操作的开始或结束。

活动图：在流程图基础上进行了一些扩展，有更强的描述能力（第**9**章介绍）

问题：分析阶段为什么要给出操作流程？
关于**OOA/OOD**分工的两种不同观点

（6）操作的命名和定位

命名：动词或动宾结构

定位：

与实际事物一致

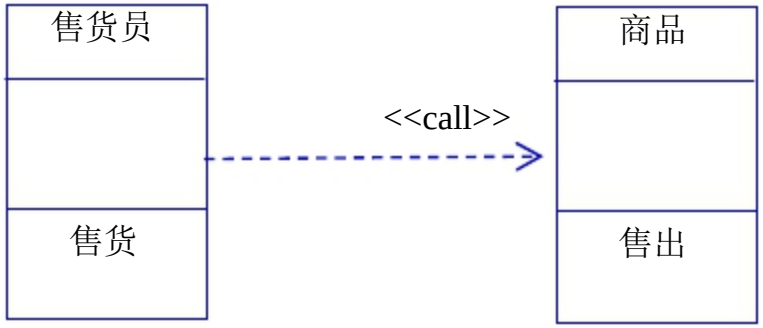
例：售货员——售货，商品——售出

在一般-特殊结构中的位置

——适合类的全部对象实例

从主语-谓语-宾语结构看对象操作的设置

“售货员销售商品”——操作应该放在哪里？



7.5 接口的概念及用途

早期的面向对象方法并没有把接口作为正式的**OO**概念和系统成分，只是用来解释**OO**概念

“操作是对象（类）对外提供的访问接口”

20世纪**90**年代中后期，接口才作为一种系统成分出现在**OOPL**中，并且被**UML**作为一种模型元素

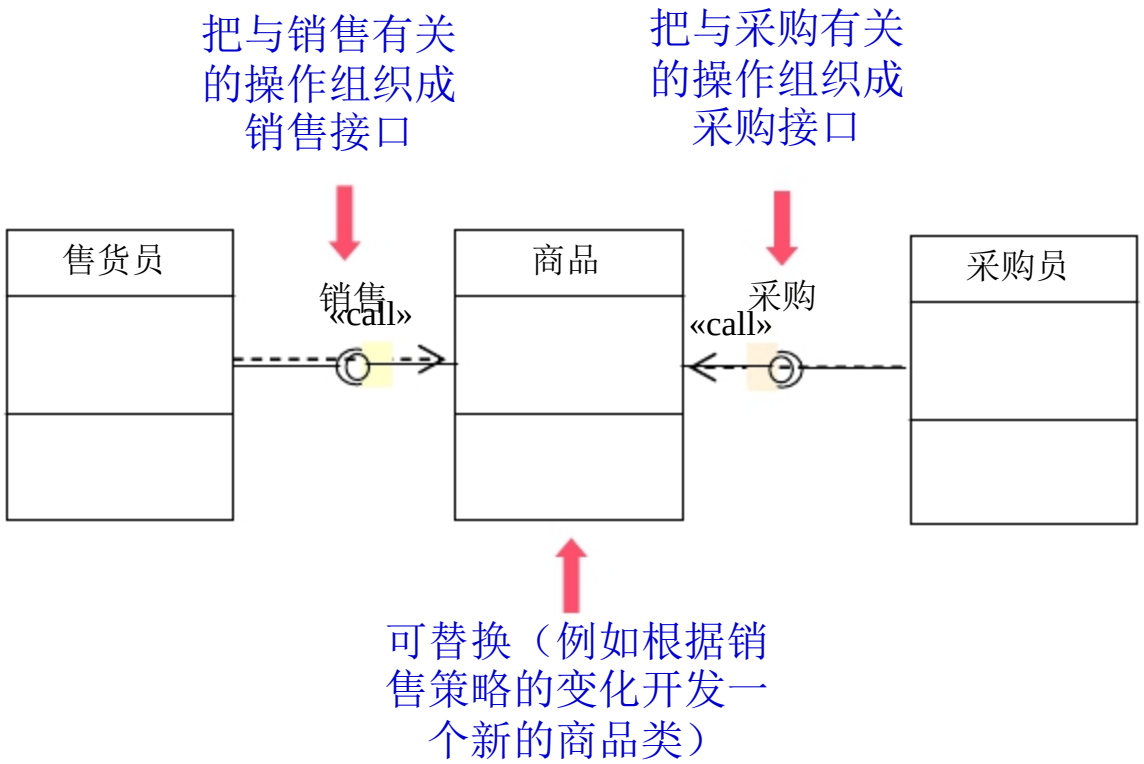
UML对接口的定义及解释：

“接口（**interface**）是一种类目（**classifier**），它表示对一组紧凑的公共特征和职责的声明。一个接口说明了一个合约；实现接口的任何类目的实例必须履行这个合约。”

“一个给定的类目可以实现多个接口，而一个接口可以由多个不同的类目来实现。”

为什么引入接口的概念

针对不同的应用场合组织对象的操作



接口提供了更灵活的衔接机制

接口（**interface**）

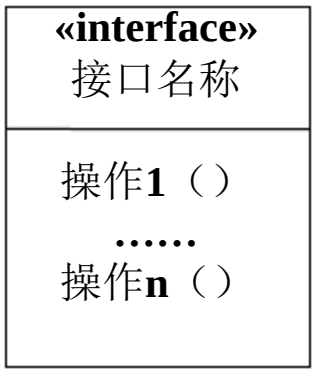
是由一组操作所形成的一个集合，它由一个名字和代表其中每个操作的特征标记构成。

特征标记（**signature**）

代表了一个操作，但并不具体地定义操作的实现

特征标记 **::=** <操作名>（**[**<参数>**:**<类型>**]** {**,**<参数>**:**<类型>}）**[:**<返回类型>**]**

表示法（详细方式）：

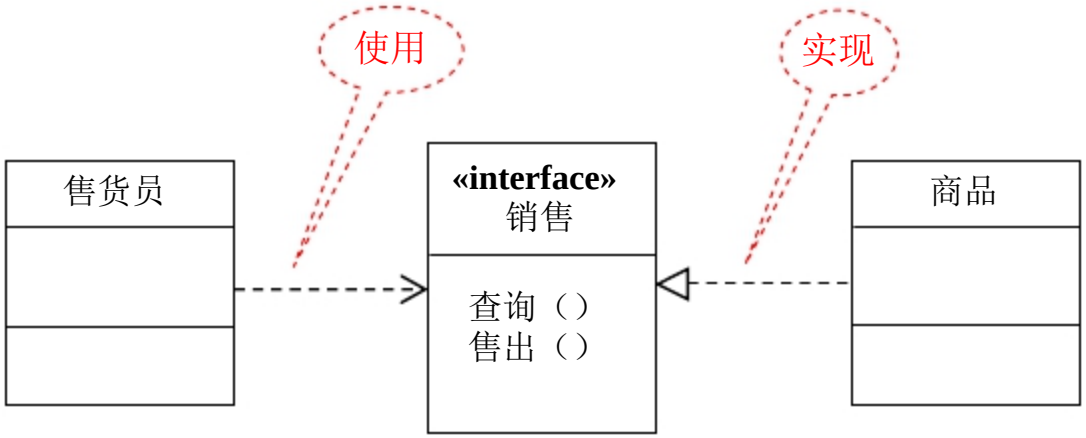


接口与类的关系

接口由某些类实现（提供），由另外某些类使用（需要）

前者与接口的关系称为实现（**realization**）

后者与接口的关系称为使用（**use**）



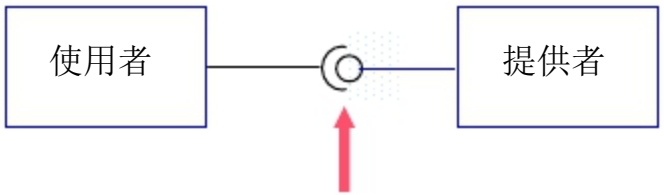
同一个接口

对实现者而言是供接口（**provided interface**）

对使用者而言是需接口（**required interface**）

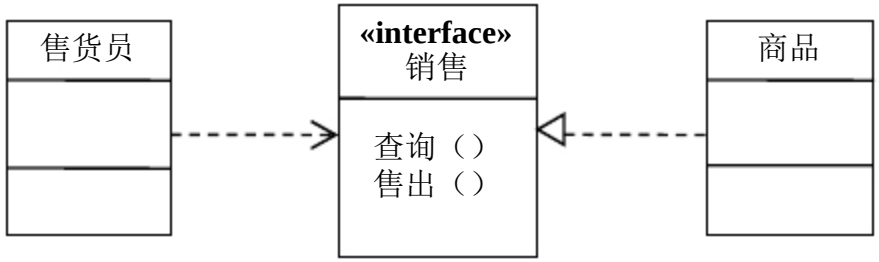
表示法（简略方式）：

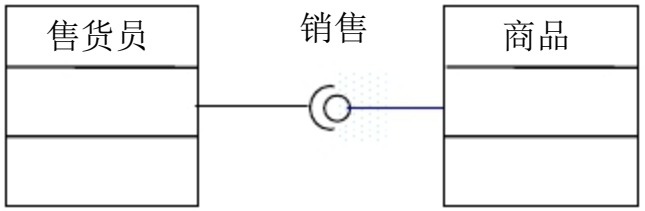
——托球-托座



提供者的供接口（托球）
使用者的需接口（托座）

例：





接口与类的区别

类既有属性又有操作；
接口只是声明了一组操作，没有属性。

在一个类中定义了一个操作，就要在这个类中真正地实现它；
接口中的操作只是一个声明，不需要在接口中加以实现。

类可以创建对象实例；
接口则没有任何实例。

引入接口概念的好处

在接口的使用者和提供者之间建立了一种灵活的衔接机制，有利于对类、构件等软件成分进行灵活的组装和复用。

将操作的声明与实现相分离，隔离了接口的使用者和提供者的相互影响。使用者只需关注接口的声明，不必关心它的实现；提供者不必关心哪些类将使用这个接口，只是根据接口的声明中所承诺的功能来实现它，并且可以有多种不同的实现。

接口概念对描述构件之间的关系具有更重要的意义