17-21 模块 抽象数据类型 OOP 抽象数据类 ③

计算机科学1992 Vol. 19No. 6

模块、抽象数据类型和类

梅 宏 孙永强（200030 上海交通大学计算机系） TP312

摘 要

本文分析了面向对象程序设计(OOP)中类(class)和传统程序设计模块(module)及抽象数据类型(ADT)间的差异，以便加深对OOP中类及对象的理解，进而讨论了类间继承关系。

一、引 言

面向对象程序设计(OOP)以其支持模块化设计、代码复用及可扩展性等优点，特别有利于大型复杂软件系统的构造，近几年来已引起越来越多人的注目并逐步流行起来，很多人甚至预料OOP将成为九十年代程序设计方法的主流。

面向对象的思想来源于模块封装和抽象数据类型。最早的OOP语言可追溯到离散事件仿真语言SIMULA67，然而，只是发展自SIMULA67语言的Smalltalk语言问世以来，才逐步形成了现今广为流行的OOP风格。现在，已出现了许多声称支持OOP的语言，但其中最典型和杰出的代表仍是Smalltalk-80[1]，现行OOP的概念和准则大多源自Smalltalk-80。

经过十来年的发展，OOP及其支撑语言的研究已取得了较大成功，面向对象方法已在软件工程、数据库、人工智能等领域得到了较为广泛的应用，然而，和另两种新型风格程序设计——函数式程序设计和逻辑式程序设计相比，OOP缺乏形式的数学基础，有着相当复杂的语义，继承的存在更是增加了语义复杂性。显然，为了更精确地理解OOP，形式的数学模型是需要的，人们已在这方面付出了许多努力，并取得了一定成果[2][3][4]。本文的目的是通过分析OOP和传统模块程序设计间的差异，以加深对OOP的理解。

二、OOP的基本概念

在OOP中，一个系统是对象集合，对象可如下定义，

定义 对象::=〈IO, S, M, IO〉，其中ID为对象的标识，即对象名；M是对象所能承受的操作的集合，在OOP中称为方法集，$M=\{meth_1, \cdots, meth_m\}$；S是对象的实例变量集，构成对象的结构并存贮对象的状态，$S=\{Var_1, \cdots, Var_n\}$；IO是对象的对外接口，即该对象可接受的消息集，它向外界提

程序设计之间的关系；

(4) 使用线性逻辑对并行PROLOG目标求解公理化等。

参考文献

〔1〕 J. Y. Girard, Linear Logic, TCS, 50: 1-102, 1987

〔2〕 J. Y. Girard, Towards a geometry of interaction Contemporary Machs. Vol. 92, 1989

〔3〕 S. Cerrito, A linear axiomatization of negation as failure J, Logic programming 1992: 12:1-24

〔4〕 黄林鹏、孙永强，线性逻辑导论，计算机科学91.1

〔5〕 S. Cerrito, A linear semantics for allowed logic programs. In Proc. LICS, 1990.

供了对象的可用方法，IO={msg$_1$, …, msg$_n$}。在IO和M间应有一定对应和联系。

在纯的OOP语言中，如Smalltalk-80，所有数据均用对象表示。对象是动态的，可在程序执行中的任何时刻动态地创生，其创立者为另一对象。对象创立后，可在一个时间段内存在于系统中，这个时间段称为其生命期。在生命期内，对象的状态可以改变。对象中的变量不可为其他对象直接访问，它们是严格私有的。对象间唯一的交互方式是消息传递，一个消息是一个对象(消息发送者）向另一对象（消息接收者）发出的执行某方法的请求，消息传递的同时，可伴随一些消息参数，接收者根据消息内容决定被执行的方法，并向发送者返回方法执行结果。只有对象本身的方法才可对本身的变量进行访问，对其他对象变量的访问只能通过消息传递间接地进行。一个对象对外可用的方法集即构成了其对外接口，为达到消息多态性，该方法集可映射到一消息名集，即定义中的IO上，再将IO作为接口使用。由于对对象的访问只能通过消息传递进行，因此，提供了对对象的强有力的保护机制，以防止无控制的随意访问，也正是这个机制分开了对象的实现细节和其行为规范。毫无疑问，对象间的对外保护应该是OOP的基本的和实质的特征，它是ADT技术的精化，因为它不仅对对象类型进行保护，也对对象本身进行保护。

为了描述具有很多对象的系统，对象可以按类分组，即用类给出一组对象的描述，通常，类可定义如下：

定义 类::=〈ID, INH, SD, MD, IO〉，其中ID是类的标识，即类名，INH是继承描述，给出被继承的类的标识，SD是除继承内容之外的结构描述，MD是除继承内容的方法定义，IO是除继承内容的接口描述。

所有具有相似性质的对象可描述为一个类，对象称为类的实例。一个类的所有实例具有相同的变量名和类型，其方法具有相同代码，它们也具有相同的对外接口。类作为创立对象的蓝本。继承可作为类构造子用于类定义中，一个新类的定义可在旧类基础上加上新的变量、方法和接口描述而得到。对一个类的例化涉及对其父类的查询，因为它继承父类的方法和结构，本身所定义的方法和结构构成了对父类的扩充。用OOP语言进行程序设计实质上就是类的定义过程，每个类定义包含一系列变量和方法以及接口的说明。类定义提供了对象创生所需的所有信息。

三、模块、ADT和类

人们可以争辩说：上节中介绍的OOP仅仅是一些众所周知的原理的新提法，通过比较，可以发现对象和记录、消息传递和过程调用间的相似性。对软件系统的构造，已有了模块和抽象数据类型等概念。为了更好地理解OOP，讨论一下模块、ADT和类间的关系是必要的。

定义 模块::=〈D, IO〉，其中D是数据类型、变量、过程等的说明的集合，IO为对外接口，这个接口规定了这些说明中的哪些可以用于模块之外。

这个模块定义实质上就是Modula-2[5]中的模块和Ada[6]中的包（package)。在模块设计中，程序员具有较大的自由去选择模块的内容及模块的对外边界。将各种说明组成模块是一种有用的程序设计方法，但语言本身并不强求组合的方式，只要能够保证模块的对外接口是可观察的即可。

定义 ADT::=〈TD, IO〉，ADT也是一个模块，但它规定了模块中的内容只能是一个类型定义TD，该类型定义描述一个类型及其内部表示和类型值可承受的操作，即TD::=〈Type, Rep, op〉，Type是类型名，Rep是类型的内部表示和结构，op为操作集。IO为ADT的对外接口，是操作集中对外可用的操作名集和有关行为规范（主要是关于参数和结果的类型）组合。ADT的内部表示是外界不可访问的，对一个ADT的值的操作

只能是操作集中所描述的操作。

和模块相比，ADT对其对外边界的选择有更严的限制，但它比模块具有更为清晰的意义。在静态附类型语言中，模块和ADT可保证定义在程序单元中的设施是被正确使用的，从而提供高度的安全性。它们提供了清晰的程序结构，利于大型复杂系统的正确构造。

相对而言，OOP中的类则是ADT的精化，它对类定义的结构有着更严格的限制。如我们给出一个ADT定义A，则其中定义的操作可访问所有A类型元素的内部细节，而在OOP中，方法只能直接访问和其相联的对象中的变量，即只能访问对应消息的接收者中的变量。任何时刻，只有一个对象的内部细节是可访问的。

下面我们给出复数的ADT定义和类定义，以此说明二者的差异。为简便计，我们只考虑复数的加法操作，其他操作略去。

这是我们使用虚构的语法，有点类似Smalltalk-80的结构，但是是强附类型的。同时，我们还将类视为对象的类型。

例1 复数的ADT定义

```
ADT Complex                    ; ADT名
Var real, imag:Float           ; 复数的内部表示
Operation
add:Complex→Complex→Complex
add x y                        ; 操作使用方式
|z1, z2:Float|                 ; 暂时变量
z1←x. real+y. real             ; ←表示赋值
z2←x. imag+y. imag
↑ Complex. new(z1, z2)         ; ↑表示返回
TAD
```

其中new是对所有ADT适用的操作。Complex. new(z_1, z_2)产生一个新的复数，其实、虚部分别为z_1, z_2。在这个定义中，操作add可以同时直接对其两个自变量取实部和虚部。两个复数的加表示为add x y。

例2 复数的类定义

```
CLASS Complex
Var real, imag:Float           ; 实例变量
Method
add:Complex→Complex            ; 只给出参数及结果类型
add y                          ; 第一个参数为消息接收者
|z1, z2:Float|
z1←real+y getreal              ; 通过消息传递访问y
z2←imag+y getimag
↑ Complex new real:z1 imag:z2  ; 返回新建对象
getreal:→Float
getreal
|  |                           ; 没有暂时变量
↑ real
getimag:→Float
getimag
|  |
↑ imag
SSALS
```

这里方法add只能对消息接收者的实例变量real, imag直接访问，消息参数y的实例变量只能通过对其发送消息而间接访问。两个复数的加表示为x add y。

从这两个例子我们可以看到，模块和ADT提供的是语法级的保护，每个模块对外保护。而在OOP中，保护出现在语义级，即运行时存在的对象间的保护。不同的对象是相互防护的，即使两个对象均是同一类的例化也不例外。从这两个例子我们也可看到，类的定义耗费了比ADT更多的代码，似乎并未显示出OOP的优势，然而，例子中出现的现象，即一个操作符作用于几个同类型操作数上，是相当特别的，主要存在于数学计算中，而在其他领域则是比较少的。OOP风格的一个重要优势是其允许同类的不同实现，如复数类可用平面座标和极座标两种表示，上面例子中，消息add的接收者和消息参数均属于类Complex，但它们可具有不同表示，如我们可有如下定义：

例3

```
CLASS Complex1
Var polar, ceta:Float
Method
add:Complex1→Complex1
add y
|z1, z2, x1, x2:Float|
z1←polar*(ceta cos)+y getreal
z2←polar*(ceta SIN)+y getimag
x1=(z1*z1+z2*z2)SQR
x2=(y÷x)ATN
↑ Complex1new polar:x1 ceta:x2
getreal:→Float
```

```
getreal
|   |
↑ polar*(ceta cos)
getimag:→Float
getimag
|   |
↑ polar * (ceta SIN)
SSALC
```

例3中Complex1和例2中Complex具有相同对外接口，故表示同一类型，但具体表示是不同的。对表达式x add y，x和y可有不同表示；而对ADT定义，则不可能出现这种情况，在表达式add x y中，x和y的结构必须是相同的。

由上我们已清楚地看到了类和ADT的差异，显然，OOP是ADT程序设计的精化。OOP支持将对某种项合适的所有信息组合在一起并封装起来，并通过接口向外展示可用部分。对一个类的用户，与其相关的仅是可用方法及其行为规范的集合，对象内部的一切均是不可访问的。应该说，OOP是程序设计方法学发展的重要方向之一。

由OOP带来的关于软件的两重要性质是①修改局部性，当要修改某部分软件时，通常相关的部分均在一个类定义中而不是分散在整个程序中。如该类修改后接口不变，则程序的其余部分将不会受影响。②可复用性，一个已经测试和验证后的类可以多次用于不同程序中。为了使用一个类，用户只需了解其接口，其内部是无关的。

至此，我们已分析了模块、ADT和类间的关系，有了对类及对象的较深入了解。然而，我们知道，OOP中还存在另一重要特征，即类间的继承性，这通常不存在于ADT中，下节我们将讨论这个问题。

四、OOP中的继承

继承的基本思想是在定义新类时，从一个现存类开始，在其实例变量和方法基础上加入新的内容，从而构成新类。新类继承旧类的变量和方法。由此，多层的继承可以构成一个继承网层。我们甚至可以允许多路继承，即一个类可以继承多个类。由于继承，将带来类间的代码共享，从而大大减少代码量。另一方面，类间的继承也暗示了另一种关系，如类B继承类A，B的每个实例将至少具有A的实例的所有变量和方法，因此，在程序中需要A类对象出现的地方，B类对象也可出现，此即所谓包含多态性。在这个意义下，我们认为B是A的特例化，称B为A的子类，A为B的父类。

在传统OOP语言中，特别是Smalltalk-80中，基于描述对象内部结构的代码上的共享而形成的继承网层和涉及对对象的使用及对外可视行为的子父类网层是完全相同的。然而，越来越多的人认识到，区别对待这两种网层是必要的[7]，因为代码共享并不一定导致行为的特例化，这也不是导致行为特例化的唯一方式。一方面，新方法的加入可能导致与旧类功能不一致的新类，从而二者行为是不同的。另一方面，对同样的规范可有不同实现，因此，子父类间也可以有各自不同的内部表示，这样二者具有行为规范意义下的继承而没有代码共享。通过区别对待这两种继承网层，有些由继承，特别是多路继承所带来的问题[7]便可以得到解决。

为区分这两个概念，我们可考虑用继承关系描述代码共享，而用子类型关系表示行为特例化和规范继承。关于继承的解释已出现于大多数OOP语言中，它仅仅作为代码共享，即类的实现的继承手段，由继承而得到的新类可和原类是不同的，也是不相交的。子类型的概念则是比较复杂的，在无类型或弱类型语言中，没有显式的类型和子类型关系，而现行强类型语言又将子类型和继承视为同一。在我们的支持函数式程序设计和面向对象式程序设计的合成语言研究中，我们将对象的对外可视行为规范作为类型，该规范的实现称为类，类的例化产生的对象则满足该规范，即具有该类型。一个类型T_1是另一类型T_2的子类型是指任何具有类型T_1的对象也具有类型T_2，也即是说对任何对象，如它满足规范T_1则一定满足规范T_2，即规范T_1强于规范T_2。我们的思想类似于EATCS的

21-26 概念结构 知识表达方法

计算机科学1992 Vol.19 No.6

一种新的知识表达方法：概念结构

白振兴（空军工程学院，西安710038） TP311.12

摘　要

本文详细论述了概念结构中概念图的表示形式：类型理论、类型标号、类型层次和类型格；个体与名字、聚集与个别；正则图及构成规则(拷贝、限制、连接和化简)。概念图是一种最新的知识表达方法，语义表达能力强，可读性好，接近于自然语言，能加入背景知识，推理速度快，已被证明优于其它传统的知识表达方法，必将为解决AI知识获取瓶颈起到积极作用。

1. 前言

概念结构 (Conceptual Structure)是美国的John F. Sowa 提出的基于语言学、心理学和哲学的一种最新知识表达方法。较别的知识表达方法，例如产生式规则、谓词逻辑、语义网络、框架、概念从属、脚本等而言，概念结构对现有的知识表达方法给以了一个全面的描述，是非常一般的方法。概念结构不但可以表示一阶逻辑，而且可以表示高阶逻辑、模态逻辑等。概念结构用概念图(Conceptual Graph) 形式表达知识。概念图一看很简单，只有两个结点，一个是概念结点，另一个是关系结点，有点象语义网络，但又不一样，能够很自然地加入背景知识，更符合人类的思维方式。其推理速度快，表达接近于自然语言，能在语义方面弄

ESPRIT 3020计划[8]，所不同的是它将类型定义为对象的对外可视行为规范，即类型作为具有对外可视的固有共性的对象集合，这里的类型仍属于值的集合。我们的类型定义如Russell中的类型定义，是操作规范集合。关于这个问题将另做讨论。此外，本文的描述均是非形式化的，要想得到OOP的比较全面的形式语义模型，更多更深入的研究工作是需要的。

参考文献

[1] A.Goldborg & D.Robson, Smalltalk-80, The Language and its Implementation. Addison-Wesley, Reading, MA, 1983

[2] L.Cardell, A Semantics of Multiple Inheritance, in Semantics of Data Types, LNCS 173, 1984

[3] J.C.Mitchell,Polymorphic Type Inference and confainment,in Semantics of Data Types, LNCS 173, 1984

[4] S.Danforth and C.Tomlinson, Type Theories and Object-Oriented Programming, ACM Computing Surveys, Vol.20, No.1, 1988

[5] N.Wirth,Programming in Modula-2. Springer-Verlag, 1982

[6] ANSI, The Programming Language Ada Reference Manual, LNCS 155, Springer-Verlag, 1983

[7] A.Snyder, Encapsulation and Inheritance in Object-Oriented Programming Language, In Proceeding of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications,Portland,Oregon, 1986

[8] EATCS BULLETIN,NO.40,Feb.1990