维普资讯 http://www.cqvip.com

类型和数据抽象

蔡希尧 陈 平（西安电子科技大学）

摘要

Promoted by the progress of programming languages.two basic concepts in programming languages—data and types, have been extended, new mechanisms are introducted, and higher levels of abstraction are adopted. In this paper, a systematic review on the related important aspects is given, including the data types and their descriptions, the concepts of static typing and strong typing, the generic functions and procedures,the abstract data types, the mechanisms of class inheritance and delegation, and the semantic data model. A brief introduction of the object-oriented programming languages is also included.

类型是程序设计语言中的一个最基本的概念，数据、函数和过程，都有自己的类型，这是构造程序的基础。抽象在程序设计语言和设计方法学的发展过程中一直起着主导作用，整个软件工程的进展总是和抽象程度的提高紧密联系在一起，而数据抽象则是最基本的一个环节。深刻理解类型和抽象的概念及其机制，是掌握现代程序设计语言，例如面向基体（object）的语言的必备的基础。

本文讨论类型和数据抽象的几个主要问题。

一、数据类型及其描述

人们在生活和工作中，由于不同的目的，常把客观事物加以区别，加以组织。按照事物的特性、行为或用途，非形式地在某个域内把事物加以分类，便产生了类型，并逐渐地形成严格定义的类型系统。在科学研究中，这是我们熟悉的方法。

例如在计算机系统中，二进制序列是最基本的，也是非类型化的字符串，但它可以用来表示整数，也可以表示实数，表示整数的二进制序列与表示实数的二进制序列具有不同的特点，两者可以区分而加以分类。

所以，类型化是以共同的特性和统一的行为把一个要计算的非类型化的世界分解成多个子集，在同一子集中的每个成份有相同的特性和行为。这些子集是可以区别的，构成了不同的类型。

在程序设计语言中，数据类型就是按照以上的思路所形成的一个重要的概念。一个数据类型可定义为：

（1）一个值的集；

（2）一个作用于值集的操作集。

例：整数类型，它所包含的值集是

……−3，−2，−1，0，1，2，3，……

操作集则是：加、减、乘、除、取绝对值等。

把一个系统加以类型化，可以减少表示问题上的麻烦。类型化其实是给系统以某种限制，有助于正确性的加强，可以防止基本的逻辑单元之间交互作用的不一致性，这些

维普资讯 http://www.cqvip.com

好处，对程序设计来说都是非常重要的。

对于类型的描述，可以按照定义中的两个方面分别加以处理。首先是确定值的集，这比较简单。当值集是有限的，且元素的数目较少时，可以逐个加以明确，例如布尔量是元素有限（只有两个）的数据类型，可直接写明

boolean: {true, false}。

在值集的元素数量大，或是无穷的时候，根据实际的应用要求，可指明它们的一个子集，例如整数类型的值集是无穷的，可定义它的某个子集，如正整数集。有时候，也可以用构造的方法来定义值集，这就是先列出若干值，同时给出产生规则，以生成其余的值。例如用26个拉丁字母生成的字符串，可定义为：(1)所有的单个字母都是字符串；(2)一个字符串上加一个字符是一个字符串；(3)没有其他形式的字符串。

描述操作集比较复杂，一方面要有语法说明，另一方面要有语义说明。语法说明要指出操作名和被操作的数据类型。例如x和y都是整数，它们要进行加法操作("+")，结果仍是整数，语法说明可写成

function"+"(x, y: integer): integer

或者用数学的记法，写成

"+": integer×integer→integer

同样，x和y的减法操作("−")的语法说明可写成：

function "−" (x, y: integer): integer

或者

"−": integer×integer→integer

其余依此类推。

在语法正确的情况下，还要说明操作的意义，这就是语义说明。语义说明有好几种不同的方法。人们首先想到的是用自然语言来说明，这是一种非形式的方法，往往带有模糊性，所以使用时要注意可能发生的情况，以求严格。例如两个整数x和y，以y去除x，如果我们希望得到的商只是整数，那么要舍掉余数，如果y=0，不能进行操作。这样，"x÷y"的语义说明将是："x÷y"的值是x除以y以后的商的整数部分，舍去余数；当y≠0时操作无定义。

数据类型的形式化语义说明，在七十年代后期相继提出[1—3]，下面用字符串类型的定义做为例子来阐明。我们规定作用在字符串String上的操作有：

(1) NULL，生成一个空字符串；

(2) ADDCHAR，在字符串上加一个字符；

(3) CONCAT，两个字符串连接在一起；

(4) SUBSTR (s, i, j)，在字符串s的第i个字符开始，给出长度为j的子字符串；

(5) INDEX (s, t)，在字符串s中，子字符串t第1次出现的位置(如果t不是s的子字符串则给出0)。

此外，以LEN (s)表示s的长度，ISNULL(s)表示查问s是否为空。在以上的约定下，给出字符串String的形式说明如下[4]：

```
type String
declare NULL ( ) →String
    ISNULL (String) →Boolean
    LEN (String) →Integer
    ADDCHAR (String, Character) →String
    CONCAT (String, String) →String
    SUBSTR (String, Integer, Integer) →String
    INDEX (String, String) →Integer
  for all  s, t∈String, c, d∈Character, i, j∈Integer let
    ISNULL (NULL) =true
    ISNULL (ADDCHAR(s, c)) =false
    LEN (NULL) =0
    LEN (ADDCHAR(s, c)) =LEN(s)+1
    CONCAT (s, NULL) =s
    CONCAT (s, ADDCHAR (t, d)) =ADDCHAR (CONCAT(s, t), d)
    SUBSTR (NULL, i, j) =NULL
    SUBSTR (ADDCHAR(s, c), i, j) =
    if j=0
    then NULL
    else if j=LEN (s) −i+2
      then ADDCHAR (SUBSTR(s,i,j−1), c)
```

维普资讯 http://www.cqvip.com

```
    else SUBSTR (s,i,j)
  INDEX (s, NULL) =LEN(s)+1
  INDEX (NULL, ADDCHAR(t,d)) =0
  INDEX (ADDCHAR(s,c), ADDCHAR(t,
  d)) =
    if INDEX (s, ADDCHAR(t, d)) ≠0
    then INDEX (s, ADDCHAR(t, d))
    else if c=d and
  t=SUBSTR (s, LEN(s)－LEN(t)+1,
      LEN(t))
    then LEN(s)－LEN(t)+1
    else 0
 end
end string
```

在这个类型说明中，"for all"以后的部分是语义说明，前面的则是语法说明。

二、类型化的机制和特点

本节将讨论类型化的几个主要机制和特点。

1. 静态类型化和强类型化

类型化的主要目的是为了加强正确性，减少不一致性。当类型给定时，它的表示要适应它的特点，并易于完成类型所要求的操作。所以在程序中要设法防止类型受到破坏，为了做到这一点，要使程序有静态的类型结构。与类型联系在一起的有常数、变量、算符和函数等，在Pascal和Ada等类语言中，变量和函数的类型有附加的申述加以定义，编译程序可以检验类型的一致性。在缺乏显式的类型信息的情况下，则设置类型推理系统，利用局部的上下文对类型进行推断以保持其一致性。

每个表达式的类型可以通过程序的静态分析来决定的语言，叫做静态类型化的语言。静态类型化对于保持类型的一致性是很有利的，但要求所有的变量和表达式在编译时间内限定于一种类型，这是很严格的要求。有的语言在这方面的要求较弱，类型可以是静态未知的，但用运行中的类型检验来保证所有表达式的类型一致性。

一个语言，它的每个表达式都是类型一致的，叫做强类型化语言。Ada就是一个强类型化语言，这是它的主要技术特点之一[5]。强类型化语言的编译程序能够保证它所接受的程序在执行时没有类型错误。我们希望语言是强类型化的，如果可能的话，最好是静态类型化的。所有静态类型化的语言都是强类型化，但逆命题不一定成立。

2. 语言的多态性

我们常用的类型化语言（如Pascal），它的函数、过程以及它们的操作对象有唯一的类型，叫做单态语言。在单态语言中，每一个值和变量只能被解释为一种类型。

多态语言则与此不同，它们的值和变量可以有多于一种的类型；与之相应的函数和过程，因为它们的操作对象可以有多种类型，称之为多态函数和多态过程。同样，类型的操作可施加于多类型的操作对象时，称为多态类型。单态类型系统限制它们的对象只有一种行为，多态类型系统放松了这一限制。

多态化的问题实际上我们早就有所接触，例如符号"+"表示加法操作，是一个二元算符，当变元是整数或实数时，它都是有定义的，可见"+"是多态的操作。这种对于不同意义的变元使用同一操作名的多态，叫做"过载多态"。加法操作的两个变元还允许一个是整数，另一个是实数，但必须有约定的解释。用列表的方法[6]把以上的情况表示如下：

+	实数	整数
实数	实数	实数
整数	实数	整数

表中有四种不同的情况和相应的结果，因此，可以说"+"操作有四种过载的意义。但也可以认为它只有两种过载的意义，即整数相加和实数相加；变元分别为整数和实数的两种情况，则认为是把整数硬性地当作实数

维普资讯 http://www.cqvip.com

使用。甚至可以认为"+"只对实数有定义，其变元中出现整数，都当作实数看待。这种把语义上需要的变换加以省略的多态，叫做"强制多态"。同一表达式，看做是过载或强制，由实现来决定。强制情况下的类型转换，由编译程序来完成。

还有一种多态的形式也是我们熟悉的，这就是一个函数或过程的参量，在不同的使用场合，可以有不同的类型，这叫做"参量多态"。此外，[7]还定义了一种多态形式，叫做"包含多态"，这是从类的继承和子类型的概念导出的多态，在这种情况下，一个基体（object）可以看做是属于多个不同的类（Classes），它们并不一定是不重合的，也就是可以有类的包含。

3. 类属函数和过程

在常用的语言中，函数或过程的参量类型在编译时间内是固定的，例如一个对整数排序的过程：

```
Procedure sort (a: array of integer, n: integer);
{body of Procedure sort}
```

它只能对整数数组进行排序。如果要对实数或字符排序，要重新写相同的过程，而把数组a的元素换成实数或字符。这种作法显然是不方便的。如果我们把过程中的操作概念加以推广，不只是针对特殊的数据类型，而是通用于任何的数据类型Elem，于是得到一个通用的排序过程

```
Procedure Sort (a: array of Elem, n: integer);
{body of procedure sort}
```

这样的过程（函数），称为类属过程（函数）。

类属函数或过程不能直接使用，必须经过实例化，这就是把通用的类型用实际的类型加以替换。取Elem=integer, n=100, 得整数排序过程

```
Procedure sort 1 (a: array of integer, 100);
{body of procedure sort 1}
```

取Elem=real, n=50, 得实数排序过程

```
Procedure sort 2 (a: array of real, 50);
{body of Procedure Sort 2}
```

取Elem=Char, n=10, 得字符排序过程

```
Procedure sort 3 (a: array of char, 10);
{body of procedure sort 3}
```

这样就得到三个具体的排序过程。

可以看出，属类函数（过程）就是参量多态化的函数（过程）。参量是在编译时实例化而得到实际的函数（过程）。这种办法，提高了抽象程度，缩减了程序的规模，减少了编码的差错，为正确性的验证提供了方便。

类属的程序结构所允许的参量不仅是变量，也可以是或过程。在Ada中，类属机制允许子程序或包加以参量化。

三、抽象数据类型

一个抽象是对一个系统的简化描述，它强调了系统的某些特性而忽略了其他。对于用户来说，所关心的是程序能做什么，而不是它的实现细节，而抽象恰好可以用来对用户所关心的重要信息予以强调，而把不重要的信息予以忽略。整个程序设计语言的发展过程就是抽象层次不断提高的过程。

在程序中的抽象和其他领域中所用的分析思型很相似，所考虑的主要问题包括：（1）决定哪些问题在系统中是重要的；（2）应当包括哪些参量；（3）采用什么描述方式；（4）所用的模型如何确认。我们常见的有三种类型的抽象：

（1）控制抽象：

控制的效果用抽象的规范来表示，不问语句如何实现，例如用

```
if…then…else
```

和

```
for
```

等形式。

（2）过程抽象：

程序中的过程替代了语言中固有的算符（如加、减等等），把算符普遍化了。过程把它所包含的一组语句，忽略了其中细节。过程还起着作用域的作用，把某些语句集中在一个局部的范围之内，它可以作为一个整体被调用。

（3）数据类型抽象：

数据类型是一组操作和它们所作用的一组对象所组成。抽象的数据类型不计实现的细节，对象的行为可以用所定义的操作完全予以规定。

关于这些抽象的机制以及它们对现代程序设计语言所产生的巨大影响，已有许多论述，〔8〕就是其中比较精采的综述，〔9〕则对这方面的问题作了系统的总结，读者可以参考。这里，只对抽象数据类型的语言支持作一简要的介绍，以便与下面将要讨论的面向基体的语言相衔接。

抽象数据类型在程序中是一封闭的单元，它所包含的信息有：

（1）外部可见的：

·类型名，操作名

·程序单元的头标

·还可以给出类型变量值的形式规范和操作的特性。

（2）外部不可见的：

内部使用的类型，程序主体，以及只能在内部调用的隐蔽程序。

所要求的语言支持主要有：

（1）在命名方面，要按照范围规则（Scope rule），以保证名的可见度；设置保护机制以保证隐蔽信息的专用；一个数据不能用多于一种的办法来命名。

（2）要进行类型检验，这通常在编译时进行。当类型信息不能或很少可能显式地给出时，需要对表达式的类型进行推理以实现检验。

（3）要提供合适的规范的记法，使程序员能够从抽象数据类型的规范得到所需要的所有信息。规范和实现要一致，因此要求有一致性的检验方法。

（4）分别编译，这为编译的优化提供了更多的可能性。

除以上四点以外，还希望语言的构造能支持存贮分配和作用于数据结构元素之上的循环、同步等功能。

抽象数据类型在现代程序设计语言中占有很重要的地位，例如共享变量的并发程序设计语言中一个最重要的构造——管程，是在抽象数据类型的概念影响下形成的[10]。Ada的一个主要的技术特点就是使用抽象数据类型[5]，还提供了类属单元。目前正在蓬勃兴起的面向基体的语言，也是建立在抽象数据类型的基础之上的，在下一节中我们将讨论这一语言。

四、类型的继承性和面向基体的语言

基体（Object）的概念以及它被做为语言的基本逻辑结构，最早出现在Simula语言之中，以后虽然不断有所改进，但受到普遍的重视，是最近几年的事。

什么是一个基体呢？有许多相似的但字句上有所不同的定义，下面引用的是Booch的定义[11]，他对基体的特性给了最详细的描述。

定义 一个基体是一个实体，它具有以下的特性：

（1）有一个状态；

（2）为它所承受的和它所要求于其他基体的动作所表征；

（3）是某个类的实例；

（4）用一个名来表示；

（5）基体互相之间只有受限制的能见度；

（6）可以用它的规范或实现进行观察；

（7）基体之间用消息传递方式进行通信。

最后一条是作者所加的。

一个面向基体的语言是以基体作为一个基础逻辑构造而成的，〔7〕认为一个语言是面向基体的，当且仅当它满足以下三个条件：

（1）语言中的基体是数据抽象，后者有指明的操作接口和隐蔽的局部状态；

（2）基体有自己的类型；

（3）类型从超类型继承特性。

在这三个条件中，（1）和（2）在前面已进行了讨论，（3）中提出了继承性，是这类

维普资讯 http://www.cqvip.com

维普资讯 http://www.cqvip.com

语言所特有的机制。

一个或多个类型的特性在定义新类型时被再次使用，就形成了类型的继承性。所以继承性是类型的合成机制。继承性的前提是不同的基体有相同的特性，把有相同特性的基体组合在一起，在语言中称为"类"(Class)。于是一个类可以从其父类继承特性，而它的特性又可以被子类所继承。因此，我们也可以把这种机制称为类的继承。利用类及其继承机制，可以生成新的基体，增加了软件的重用性，便利了维护，这些显然都是很可贵的特点。

类是互斥的。有的语言规定了单一继承性，这就是每一个类只有唯一的一个父类。例如有以下三个类：

Class object =⟨age⟩

Class vehicle =⟨age, speed⟩

Class machine =⟨age, fuel⟩

vehicle和machine这两个类都是object的子类，它们从父类object继承了特性age。利用单一的继承性，容易形成树形的层次结构。

继承也可以是多重的，也就是一个类可以有多于一个父类。例如在前面的三个类再加一个类：

Class car=⟨age, speed, fuel⟩

那么，它既可以继承vehicle类的特性，又可以继承machine类的特性。多重继承性也能够形成层次结构，但这种结构不是树，而是格。用格表示信息之间的关系，比树更加丰富。

继承性是一种资源共享机制。在面向基体的语言中，还有另一种共享机制，叫做代理(delegation)，使用这一机制，一些子处理任务可以委托其他的基体代理。在继承机制中，一个基体可以从高一级的基体继承信息，控制仍归自己；在代理机制中，控制也将传送给代理者。当一个基体向代理者发出代理请求以后，自己可以接受其他的信息。这种机制也不是陌生的，在消息传递的并发系统中，需要调用本地以外的过程时，工作过程与此有类似之处。[12]给出一个利用代理机制于雷达ESM系统的例子，可资参考。

无论是继承或是代理，这些新的共享机制，给面向基体的语言增添了活力。面向基体的语言由于有许多其他语言所不具备的特点，所以这几年发展迅速。八十年代，程序设计语言的发展可以归结为三个流派[13]：

(1) 计算被看做是对一个输入施用一组函数的过程，称为函数式程序设计(FP)。

(2) 计算被看做是推演过程，给定输入和初始状态，在一组条件制约下，采用推理算法进行演算，称为逻辑程序设计(LP)。

(3) 计算被看做是一个系统的开发过程，系统包括若干基体，经历一连串的状态变换以完成计算，这就是面向基体的程序设计(OOP)。

这三种流派各有自己的特点和长处，但最近两三年的情况是：FP和LP都在扩充自己，使之具有OOP的特色，这是值得注意的新的动向。

五、 语义数据模型

数据库是信息系统的核心，随着信息系统的迅速发展，研究并开发更适合于应用要求的数据库，成为一个很重要的领域。七十年初提出来的关系数据模型，由于它是面向终端用户的，把数据逻辑表示和物理实现分离，设计了标准的非过程的查询语言等，已发展为目前应用最广泛的数据库系统。但关系数据库也有许多明显的弱点，不能满足一些应用要求。只要简单地把面向基体的模型和关系模型作一比较，就可以发现用前者的概念构成的数据库将在很多方面优于关系数据库。

语义数据模型是另一种在许多方面优于关系模型的数据模型。它也是在七十年代被提出来的，但比关系模型的提出稍晚。语义数据模型研究的主要成果是它具有表示数据结构方面的强大机制。它开始被提出来时主

维普资讯 http://www.cqvip.com

要是作为方案设计的工具。一个方案先用语义模型设计，然后转换成通用的模型并加以实现。当时强调的是为典型的数据库应用中的数据间的关系提供准确的模型。

语义数据模型的主要方面是对基体，基体的特征和基体间的关系，生成复杂类型的类型构造子，ISA关系以及导出的其他构造的显式表示。这种模型和面向基体的程序设计语言的主要不同在于，语义模型隔离了基体的结构方面，而面向基体语言隔离了基体的行为方面。此外，方法的继承和特征的继承也是不同的，在语义模型中，特征的继承只在类型之间，其中一个类型是另一类型的子集。方法的继承由于是行为性质而不是结构性质的，因此可存在于看来并不相似的基体之间。近年来有比较详细的语义数据模型方面的综述文章[14],[15]，可资参考。

Unisys公司已经为它的A系列机提供了一个语义数据库[16]，所采用的语义数据模型实现了的主要概念有：（1）实体，（2）类和子类，（3）通用等级，（4）特征继承性，（5）实体加值特征。设计者认为语义数据模型具有关系数据模型的所有优点，但克服了后者的缺点，模型更加自然地反映了现实世界。

参考文献

1. Guttag, J., Abstract Data Types and the Development of Data Structures, CACM, vol.20, No.6, June 1977, 396-404

2. Gries, D. and Gehani, N., Some Ideas on Data Types in High-Level Languages, CACM, vol. 20, No. 6, June 1977, 414-420

3. Liskov, B., Synder, A., Atkinson, R. and Schaffert, C., Abstraction Mechanisms in CLU, CACM, vol. 20, No. 8, Aug. 1977, 564-576

4. Guttag, J., Horowitz, E. and Musser, D, The Design of Data Type Specifications, in "Current Trends in Programming Methodology", vol. IV, Data Structuring, ed. by R. T. Yeh, Prentice-Hall, 1978

5. Barnes, J. G. P., An Overview of Ada, Software Practice and Experience, vol. 10, 851-887, 1980

6. Horowitz, E., Fundamentals of Programming Languages, Springer-Verlag, 1983

7. Cardelli, L. and Wegner, P., On Understanding Types, Data Abstraction, and Polymorphism, ACM Computing Surveys, vol. 17, No.4, Dec.1985, 471-522

8. Shaw, M., The Impact of Abstraction Concerns on Modern Programming Languages, PIEEE, vol. 68, No. 9, Sept. 1980, 1119--1130

9. Wulf, W. A., Shaw, M., Hilfinger, P. N., and Flon, L., Fundamental Structures of Computer Science, Addison-Wesley Pub, Co., 1981

10. 蔡希尧，多处理机系统的逻辑分析和设计，第二章，西电出版社，1987

11. Booch, G., Object-Oriented Development, IEEE Trans. vol. SE-12, No. 2, Feb. 1986, 211-221

12. Barry, B.M., Altoft, J.R., Thomas, D. A. and Wilson, M., Using Objects to Design and Build Radar ESM Systems, ACM SIGPLAN Notices, vol.22, No. 12, Dec. 1987, 192-201

13. Nygaard, K., Basic Concepts in Object-Oriented Programming, ACM SIGPLAN Notices, vol.21, No10, Oct, 1986, 128-132

14. Hull, R., and King, R., Semantic Database Modeling: Survey, Applications, and Research Issues, ACM Computing Surveys, vol.19, No.3, Sept.1987, 201-260

15. Peckham, J. and Maryanski, F., Semantic Data Models, ACM Computing Surveys, vol.20, No.3, Sept.1988, 153-189

16. Balfour Unisys Europe-Africa Division, The Practical Implementation of a Semautic Database, Database and Network Journal, vol.18, No.4, 1988. 2—5