计算机科学1990№5

面向对象程序设计语言BETA

曹 华 赵致琢

（中国科学院数学研究所）

摘 要

本文介绍一个新的面向对象的程序设计语言BETA。BETA语言在发展了十几年后，虽然国内对它了解甚少，但BETA的许多思想和风格确实在语言理论研究中占有重要地位，而且可以肯定，BETA将影响国内的一些研究工作。本文根据参考文献编译，限于篇幅，不可能完整而又详细地介绍。有兴趣的读者，请参考有关的文献。本文引用的参考文献系指B.B. Kristensen等人的文章中列出的参考文献。

1.抽象机制

1.1 引言

BETA是一个具有SIMULA 67传统的现代语言，它支持程序设计中面向对象的透明性，并具有过程式和函数式程序设计的综合功能。BETA使用一种称之为模式的抽象机制取代类、过程、函数和类型，它将虚拟过程推广到虚模式和诸如嵌套、块结构等流行的语言学概念，提供了顺序、共行和并发执行的一个统一的架框。BETA语言比SIMULA要小，但表示能力很强。

在BETA中，模式的实例称为对象，可作为变量、数据结构、过程/函数激活、共行程序和并发系统使用。模式可根据子模式按分类层次组织。用子模式组合成虚模式使得有可能推迟说明模式中的某个属性，这样，属性在不同的子模式中可能会有不同的约束，此对应于Smalltalk中方法的动态约束。然而，同Smalltalk相比，虚模式的使用可以在编译时检测，即使该约束是在运行时完成也无妨。

下面我们简要地介绍BETA语言的基本结构。

1.2 基本结构

1.2.1 **对象与模式** 一个BETA程序的执行是一个对象的集合。在一个BETA程序中，对象由对象描述符描述，它们具有下列形式：

```
(#                          {开始对象描述}
   D1; D2; …;Dn             {说明表}
enter In                    {输入参数}
do Imp                      {待执行的命令}
exit Out                    {输出参数}
#)                          {结束对象描述}
```

$D_1, D_2, \cdots, D_n$是属性说明，I_n是输入参数表，Imp说明当对象作为过程、共行子程序或进程执行时待执行动作的命令，Out是作为对象执行的结果输出的输出参数表。enter部分对应于值参，而exit部分对应于结果参数。

对象描述符可用于描述对象模式。一个模式由一个有名的对象描述符构成：

```
C: (#D1; D2; …; Dn
enter In
do Imp
exit Out
#)
```

模式充当产生对象（实例）的模板，作

为C的实例产生的对象都具有相同的结构，这就是说它们具有相同的说明集、enter部分、do部分和exit部分。

说明用于描述属性，这些属性可以是对象和/或模式。一个模式的模式属性与Smalltalk中类的接口运算相对应，对象属性与实例变量相对应。当对象作为一个过程，共行子程序或进程执行时使用对象的do部分，它没有Smalltalk中的副本。do部分通过引用一个模式名被调用，就象用过程调用一样。对象的do部分表示了对象概念的一个非常重要的扩展，它允许模式作为过程被执行，以及用于模拟正在执行的进程和系统。模式可借助求值实例化。

1.2.2 **求值** 说明对象执行步骤的基本机制叫做求值。求值是一个命令，在其执行时可能引起状态的改变并输出一个值。求值概念为赋值、函数调用和过程调用提供了一种统一的方法。

求值的一般形式是：

$E_1 \rightarrow E_2 \rightarrow \cdots \rightarrow E_n \quad n>0$

每个E_i或者是一个对象名称（对象或模式），或者是一个求值表（F_1, F_2, …, F_n），其中F_i是一个求值。假如E_i是对象名称，那么E_i的执行就意味着执行E_i的do部分，若E_i是一个求值表，那么它是一个空的动作。一个值从E_i传给E_{i+1}是指将E_i exit表的元素赋给E_{i+1} enter表对应的元素。如果E_i, E_{i+1}其中一个或两者是求值表，那么这些表就取enter/exit表的位置。一个值从E_i传给E_{i+1}是合法的，是指它们enter/exit表中对应的元素都是可赋值的。注意，赋值的递归定义意味着在值传递过程中还要执行被赋值的对象的do部分。

BETA的模式统一了过程和函数。可以用类似于过程和函数的方式使用模式。至于求值运算，针对通常使用的诸如Integer、Boolean、Char和Real这类数据类型以及它们的运算，BETA包括了若干预先定义的模式。例如，+、-、*等常用整数运算和一些别的模式。下面是一个简单的例子。

```
(#
Power: {计算X^n, n>0}
{#X, Y: @Real; n: @Integer;
            {三个对象属性说明}
enter (X, n)
do 1->Y;
 (for inx: n repeat Y*X->Y for)
exit Y
#)
Reciproc: {计算 (Q, 1/Q)}
 (#Q, R: @Real;
enter Q
do (if(Q=0)
'// True then 0->R
// False then 1/Q->R
if)
exit (Q, R)
#);
A, B: @Real;
do(3.14, 2)->Power->Peciproc->(A, B)
          {A=3.14^2, B=3.14^-2}
#}
```

1.2.3 **控制结构** BETA的迭代控制结构称为for命令，其形式如下：

```
(for Index: Range repeat Imperative-list for)
```

这里，Index是一个Integer对象名，Range是一个Integer求值，Range的求值优先于for命令的执行并确定Imperative-list的执行次数，Index的值从1, 2, …, 到Range。Index的范围是Imperative-list，不能对Index赋值。

选择控制结构称为if命令，形式如下：

```
(if E0
 // E1 then I1
 // E2 then I2
  ...
 // En then In
 if)
```

其中，E_i(i=0, …, n)是求值。首先求出E_0的值，然后以任意次序用E_1, E_2, …, E_n

同这个值进行相等测试，如果$E_0=E_j$，那么执行I_j。倘若有一个或多个选择是可选的，那就随机地选择其中的一个。如果不存在可选的选择，那么就继续在if命令之后执行。依求值的定义，可知这隐含了支持用户自己定义的相等。

此外，BETA还支持另外一种控制结构，即标号语句。对象描述符do部分内的一条命令可以标上标号：

L：命令 (Imperative)

L是标号名，L只能在Imperative内引用。通过在命令中执行一条leave或restart命令可以终止一个带标号的命令的执行。执行leave L，则在标以L的命令之后继续执行，而执行restart意味着在标以L的这条命令处继续执行。

1.2.4 对象种类和构造方式 BETA有三种对象：系统、部件和项。一个对象的种类说明了如何使用这个对象。

- 一个系统对象可以与其它系统对象一样并发执行；
- 一个部件对象（共行子程序）可以和其它部件一起交替执行；
- 一个项对象是包含在系统、部件或项之中的一个从属动作序列。

项对象可以通过静态说明、直接插入和动态执行"新"命令生成。生成对象的方法称为构造方式，由此引出的三种项分别称为静态项、插入项和动态项。例如

```
E: @P      {说明E为一个静态项}
E→C→A      {说明C为一个插入项}
```

动态项稍稍复杂一些：

```
X: ↑P
```

P是引用的限制条件，它说明X可以表示(引用）一个P项、P的一个子项或NONE，NONE意指空项。动态P项可以通过执行一条"新"命令&P产生。用这种方法产生的对象称为动态项。

用下列方法可以给动态引用一个值：

```
&P□→X□; {产生一个P项并将它的引用赋给X}
```

&P□说明返回引用新近生成的项。相比之下，象&P一样没有方框的求值是指执行这个项。

下面这个例子描述一个计算阶乘的递归函数，其它模式描述了一个链表。

```
(#
  Factorial:
   (#N, fac:@Integer
  enterN
  do (if (N≤1)
  // True then 1→fac
  // False then((N-1) →& Factorial)•N→
    facif)
  exit fac
  #)
  Link:{Link描述一个链表}
   (# succ↑Link; {this; Link的尾}
  elm:@Integer{head element of this Link}
  Insert:{将一个元素插在this Link之前}
   (#E:@Integer; R:↑Link
  enter E
  do &Link□→R□; {R表示Link的一个新
  实例}
  E→R. elm; {E⇒R. elm}
  succ□→R. succ□; {this Link的尾=R的
  尾}
  R□→succ□ {R=this Link的尾}
 #)
#)
 head:@Link; {一个静态Link项}
do 0→head. elm;
   (for inx:4 repeat
      inx→&Factorial→head. Insert
   for); {head=(0 24 6 2 1)}
 #)
```

概括地说，项类的对象可用三种方式产生：

- 作为一个静态分配的属性，称为静态项；
- 作为一个静态分配的动作，称为插入项；

• 作为一个动态分配的属性和动作，称为动态项。

1.2.5 **奇异对象** 在说明一个对象时，有可能不引用模式而直接描述对象的性质。一个直接被描述的对象称为奇异对象，一个作为模式实例被描述的对象称为模式规定的对象。

1.2.6 **块结构和作用域法则** 就块结构、域法则和类型检查而论，BETA 属于ALGOL家族。在ALGOL和SIMULA 中，一个过程可以有局部过程和/或块。在BETA中，对象描述符文体上可以是嵌套的。块结构是构造大型程序各部分的一种重要机制，块结构和子模式是构造对象和模式的补充机制。[BETA 83a]介绍了BETA 组织大型程序的模块化机制，其思想是由BETA 文法的非终结符产生的任何句型可以是一个模块。这种模块化机制还支持将BETA 程序分成接口模块和实现模块。

信息隐藏/保护机制是许多现代语言所追求的目标之一。BETA没有这种机制，因为落在对象描述体内的所有名字都是可见的，但BETA可通过模块化机制来实现这种保护机制。

1.3 分类层次

模式可按分类层次组织，一个对象描述可以包括一个上模式（常叫做前缀模式或简称前缀）。这说明根据描述产生的对象具有上模式刻划的一切性质。我们用下列方式刻划一拥有一个模式的模式：

```
C1:C(# D1′; D2′; …; Dm′
enter In′
do Imp′
exit Out′
#)
```

其中，C是C_1的上模式，C_1也叫做 C 的子模式。这样，除了（#…#）之间称为C_1的主要部分所说明的那些以外，任何C_1对象都具有同C对象一样的性质。

C_1对象具有与说明$D_1,\cdots,D_n$和$D_1',\cdots$ D_m'相对应的属性。C_1对象的输入部分是In和In′的并，C_1对象的输出部分是Out和Out′的并。C_1对象的动作部分是Imp和Imp′的组合，这种组合靠inner命令描述。一个 C_1 对象通过执行C中的命令Imp开始执行。在执行Imp的过程中，每执行一条inner命令意味着Imp′的一次执行。

BETA与Smalltalk 的区别之一是"变量定型"的问题。变量在Smalltalk 中没有类型而可以指任何对象，而BETA 中引用则由模式名限定。限定说明引用只能联系由那个模式指定的对象。限定概念定义如下：称模式C_1由模式A限定，如果

• A为C_1，或者

• C_1的上模式由A限定。

类似地，一个对象由模式A限定，如果它是A 的一个实例，或者该对象的描述中一个可能使用的上模式是由A 限定的。

1.4 虚模式

BETA 中引进了虚模式的概念。可以将模式属性作虚拟说明，这意味着虚模式只有某些性质是已知的，而它的完整的说明可以延迟。已知的性质与虚模式的一个前缀相对应，这个前缀在虚模式的说明中具体说明。因此，虚模式是外层模式的"模式参数"。一个虚模式的说明如下：

```
V:<A
```

V被说明为具有限定模式A的虚模式。模式V可能受到A的任意子模式的约束。当虚模式不受约束时，它有一个缺省约束，即它的限定模式。

虚概念是广义的，它对所有种类的对象可用于整个模式概念。利用模式和子模式使我们有可能构造出色罗万象的抽象。然而，这种方法存在着一些限制和缺点，为构造控制抽象而使用 inner 和虚模式之间有一种不对称性。为了处理这些问题，[BETA 83]给出了一种虚模式准则，[BETA 87]中又拓展了虚模式的概念。

KeyMax:

```
(#Rec:<Record;
M:@Rec
enter (M)
do (if (M.Key>MaxKey) // True then
        M.Key->MaxKey;
        inner
    if) #)
```

KeyMax中有一个虚模式属性Rec，已知为带有Record前缀。所以，KeyMax的任意Rec对象拥有Record对象的一切性质。如果任意Record对象有一个Key属性，则M.Key是合法的。其效果是Rec可用作KeyMax的一个"模式参数"。

2. 多顺序执行机制

BETA语言支持多种不同的动作顺序结构，包括顺序动作序列、交替和并发。各并发部件之间的通讯通过项的同步执行实现，这种Ada中的会合（rendezvous）相似，BETA没有类似于CSP和Ada的警戒输入/输出的结构，而是使用交替和块结构的一种组合。

2.1 引言

程序设计语言中动作排列顺序最简单的是顺序执行。大多数情况下，将程序执行组织成几个顺序进程比较合理。这种方式称为多顺序执行。众所周知，共行程序对于支持多动作序列的程序执行非常有用。

BETA语言引进了一种说明混合对象的一般方法。交替和混合对象是非常有用的构造机制，二者一起使用，可代替警戒命令。

系统对象以并发形式支持非确定性多动作排列顺序，依据并发命令可执行系统。

系统可以通过项的同步执行进行通讯。系统之间的同步类似于CSP中的信号交换或Ada中的会合。并发和同步将在2.2节中介绍。

我们可以采用正文嵌套方法说明混合系统。一个系统可说明一个或多个内层对象的并发或交替执行，这样的混合系统将具有几个正在执行的动作序列。外层系统可直接与不和内含系统同步的内层系统通讯，内层系统可以隶属于封闭系统的项，这些项一次执行一个。据此，用几个并发动作序列组成的混合系统使我们有可能模拟很多活动。

部件的交替执行可用交替命令来说明。一系列部件的交替执行说明一次只有一个部件执行其动作部分，没有执行动作的部件延迟到严格定义的点（时刻）。

2.2 系统的并发执行

这一节介绍系统对象的生成和系统的并发执行。下面的例子说明了一个包含系统Slave1，Slave2和Master的BETA程序。程序的动作部分由并发命令组成：

```
(||Master ||Slave1 ||Slave2||)
```

并发命令说明系统的并发执行。仅当所包含的全部系统都挂起了动作执行时并发命令才结束。一个系统在它执行了挂起命令或执行完动作部分后挂起。

```
#)
  Slave:(#…#);
  Slave1:@||Slave;
  Slave2:@||Slave;
  Master:@||(#…#);
  do(||Master||Slave1||Slave2||)
#)
```

系统之间的通讯通过项的同步执行实现。下面是两个正在通讯的系统：

```
S:@||(#…
        do…E1->R>? M->E2…
        #);
R:@||(#M:@(#…with S do…#);
        …
        do…; <?M; …
        #);
```

E_1->R>? M->M_2意思是请求系统R执行项M。M的with部分说明M可以和S同步执行。<? M的意思是R执行接收命令。

在BETA中，一个系统所执行的动作可分布于几个内层系统中。内层系统之间可以相互通讯，也可以与外层系统通信或者控制

外层系统与内含系统之间的通讯。

2.3 部件的交替执行

BETA用命令(|C1 |C2 |C3 |)来说明部件之间的交替执行。交替执行是指一次至多一个部件执行其动作部分。切换可发生在：①部件动作部分的起点；②部件试图进行通讯时；③部件挂起其动作序列时。

交替不是并发的替代方式，而是并发的补充。许多活动本质上是交替进行的，非确定的，这些活动不应当用并发系统，共行程序或警戒命令来模拟。然而，有必要进一步探讨一下，BETA中的交替是否是一种有效的通用的顺序机制。

3. 结论

作为BETA计划的一部分，现已开发了BETA语言。这项研究始于1976年，该语言各个时期的情况记载在〔BETA76—85〕中。

BETA的应用领域是大规模软件系统的程序设计，包括嵌入式和分布式计算机系统。因此，我们试图开发一种具有能够有效实现的结构，并且非常通用的语言。

BETA遵循SIMULA传统的现代语言，它较之Smalltalk等语言在"类型"上有进步。"类型"的优点是众所周知的，它使编译程序能够检查错误，否则将在运行时发现一个又一个的错误，同时程序也更易于理解，能生成更有效的代码。目前尚未有任何一种语言能够成功地将类/子类结构与处理并发和交替的结构结合在一起。并发PASCAL作了这种尝试，但没有子类和虚拟(类)。

SIMULA是一种系统描述和编程语言。根据SIMULA的概念名（Platform）开发的DELTA语言〔DELTA〕仅仅是一种系统描述语言，它允许描述完全并发、连续变化和部件交互。BETA从DELTA的系统概念出发，发展成一种程序设计语言，对七十年代程序设计的研究作出了巨大贡献。

BETA支持面向对象的程序设计方法。BETA计划的主要部分是发展基于面向对象的程序设计的基本概念。本文仅介绍了BETA程序设计语言，有许多基本概念尚未涉及。

模式结构是一种非常通用的机制。实际上，大多数模式被用作类、过程、函数和类型，几乎没有模式能够具有一个以上的上述用途。对于抽象的上模式，经常未明确说明它们的子模式的用途。语言中只采用一种抽象机制的优点是可以统一处理抽象机制及其实例，而且层次分类对用作过程、函数或类型的模式仍然有效，虚概念仅对过程无效。虚模式可以模拟类SIMULA的虚过程和Smalltalk的方法，也可以模拟在SIMULA和Smalltalk中无效的虚类。虚模式还可模拟规范过程的规范类型。

BETA已发展了多年，并且还在继续发展之中。下面介绍一些BETA语言进一步研究的课题：

- BETA的值概念需要进一步发展，有关的一些思想在〔BETA 83b〕中作了论述。BETA应支持PASCAL中的枚举类型的概念。考虑

  color=(red, green, white)

  在BETA中，用模式来模拟color，red，green和white。red，green和white是color的子模式。把red这样的模式具体说明为别的子模式也应是可能的，这样就有可能描述类型的层次和值。

- BETA不支持多路继承。现在尚没有可采纳的关于支持多路继承的建议。问题在于取代组合动作部分的内部机制，〔Thomsen 86〕提出的方法是一种很有前途的方法。

- 除了面向对象的程序设计之外，BETA还支持过程性程序设计和限定范围的函数式程序设计。目前正在做的工作改进了对函数式程序设计的支持。此外，还应包括对逻辑程序设计的支持。当然，BETA不可能等同地支持所有这些程序设计风格，重点是支持面向对象的程序设计，函数式和逻辑程序设计的主要目的在于描述对象的可度量的性质和有意义状态之间的转换。

- 前面提到了内层模式和虚模式使用中的不对称性，〔BETA 83b〕中提出了改进的意见。

- BETA没有用于数据表示模块化和保护的结构，现已开发了基于语言的上下文无关文法的〔BETA

计算机科学1990№.5

JSP & JSD

蔡建明　林钧海

（南京航空学院）

摘要：本文全面介绍了Jackson 方法的主要思想及其形成过程，并对该方法作了简要评述。同时，针对国内研究中的有关提法进行商榷。最后，谈到了支持Jackson方法的软件开发环境。

一、概　述

Jackson 方法是以数据结构为基础开发软件的代表性方法，它由英国的M．A．Jackson 提出，经过若干人十几年的努力，目前已成为一种有影响的软件开发方法。它的发展分为二个阶段，前期（70年代）主要研究结构化程序设计，称为JSP （Jackson Structured Programming），后期（80年代）集中研究软件系统的开发，称为 JSD （Jackson System Development)[3]。国内不少软件工程专著称JSP为Jackson方法的提法值得商榷，时至今日，Jackson 提出的方法已不仅仅用来设计程序，而且是开发软件系统的完整方法。

与传统的功能分割，结构化分析和设计的观点不同，Jackson强调对问题解的组合而不是分解。Jackson坚持认为[2]，自顶向下逐步求精的方法只适合描述问题的解，并

83a]，以取代通常独立于语言的机制。其思想就是程序可分解成任意的片段(模块)，一个片段可以是由文法的非终结符产生的符号和文法的终结符所组成的字符串，通过定义片段间的偏序，可以进行独立的上下文有关（静态语义）分析。这种方法可将一个模块分成接口部分和实现部分，从而提供对数据机制的保护。最后，这种机制可作为构造处理不同程序版本/变体的系统的基础。

• 对一大类系统例如操作系统和数据库系统来说，在系统的寿命（执行）期内交换部件的能力是系统固有的和自然的特性。为了做到这一点，必须有一个包括程序执行和完成该程序执行的处理器的机器模型。目前，BETA正在研究这种模型，机器将被看作是按照某种语言描述的剧本（程序）实现（程序执行）的一个整体。

• 实现一个给定应用领域中的概念和现象的一组属性（模式和引用）可看作为描述属于该应用领域的系统的语言。更为常见的要求是使用一种语法而不是BETA来描述这样的系统，因此，BETA程序设计系统应该包括一个工具，给定一组属性及其语法描述，该工具将产生一个新的BETA 编译器的前端。

尽管上面引出了一些BETA 有待改进之处，但我们认为BETA 已达到可用来编制计算机系统程序的水平。BETA 的一个实验版本已经在各种机器（包括Macintosh，Sun和Syswar SUS）上实现。目前，有关设计和实现一个BETA程序设计环境〔Mjølner〕，〔Scala〕的工作正在继续。

参考文献

1. B．B．Kristensen et al,The BETA Programming Language, Research Directions on Object-oriented Programming, The MIT Press, 1987．pp．8-48．