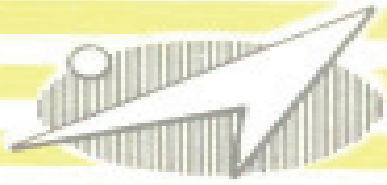
Ada语言简明教程

麦中凡 译 谢竹建 校

[美] M.J.STRATFORD-COLLINS 著



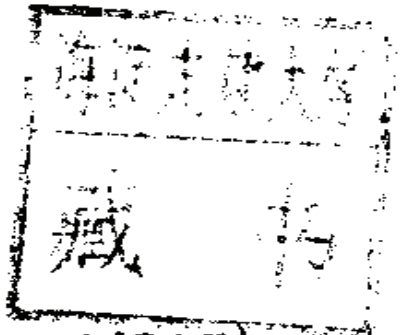
北京航空學院出版社

Ada语言简明教程

M.J.S 斯特拉特福-柯林斯
[美] M.J.STRATFORD-COLLINS 著

麦中凡 译

谢竹虚 校



北京航空学院出版社

内容简介

*Ada*程序设计语言是第四代计算机有代表性的语言之一。主要应用于大型软件系统、各种军事装备上的嵌入式计算机系统、实时系统，而且支持系统开发和分布式应用。尤其是*Ada*能在整个软件生存期中支持软件工程的各项目标，从而有效地降低软件费用。近期内，*Ada*可能成为软件产业中的主导语言，甚至可能取代*FORTRAN*和*COBOL*。

《*Ada*程序设计语言简明教程》是为已经熟悉一门高级程序设计语言的读者编写的*Ada*入门教材。本书内容简明扼要，文字通俗，例题浅显，是一本良好的自学教材。

本书适用范围，计算机软件工作者、计算机专业研究生、高年级大学生、以及从事计算的各学科科技人员、信息工作者。

*Ada*语言简明教程

[美]*M.J.STRATFORD-COLLINS* 著
秀中凡 譯　謝竹虛 校

责任編輯　肖之中
北京航空学院出版社出版
新华书店北京发行所发行　各地新华书店经售
北京航空学院印刷厂印装
*
787×1092 1/32 印张：7.25　字数：162 千字
1985年11月第一版 1985年11月第一次印刷 印数：1—10000册
统一书号：15432·001　定价：1.45 元

译 者 的 话

*Ada*程序设计语言最初是美国国防部为摆脱软件费用急骤增长而研制的大型通用的计算机语言。自75年开始研究以来，前后经历约十年时间至83年2月，*Ada*语言文本才正式定型。84年已有第一批商品的系统软件问世。*Ada*是第四代计算机有代表性的语言之一。主要应用于大型软件系统、嵌入式计算机系统、实时系统，而且还支持系统设计应用和分布式应用。尤其是*Ada*能在整个软件生存期（*Lifecycle*）中有效地降低软件费用。这是当今现有高级程序设计语言无法与之比拟的。因此，有人预言从现在到公元2000年将是*Ada*鼎盛时期，她会成为软件产业中的主导语言，甚至有可能在科技计算和数据处理的领域中代替*FORTRAN*和*COBOL*。当然，一个语言的成功与否不仅与它本身性能有关，而且更重要的是看她能否为使用者接受，以及权势的支持。为此，美国国防部这个软件业界最大的用户声称，自84年2月份以后，所有军用软件一律用*Ada*语言作为开发工具，否则不予承认。我们知道，60年代初期*COBOL*就是在美国国防部全力支持下才形成今日统一的通用语言的。因此，可以预期，*Ada*必将得到广泛的应用。

我国计算机科学界和军事科学界对*Ada*的研制一开始就十分关注，并进行了广泛的评论和深入的研究。目前已有几个*Ada*实验性子集问世，正酝酿开发我国正式的*Ada*。因而，一个学习，研制、应用*Ada*语言的高潮即将到来。在国

外这两年高潮已经出现。有关*Ada*的各种教程已有50余种。为了有助于我国关心并渴望及早了解*Ada*的读者，结合当前教学需要，我们选择了本书。

《*Ada*简明教程》原名为“*Ada*：*A Programmers Conversion Course*（*Ada*：程序员的转换教程）”。为美国加州计算机高级程序设计员*M.J.Stratford-Colins*所撰写。顾名思义，本书是为已熟悉一门高级程序设计语言的人学习*Ada*而写的。内容简明扼要，篇幅仅及*Ada*语言文本之半。文字通俗、例题浅显、清晰地写出了*Ada*语言的各主要特征和*Ada*的思想风格。是一本自学入门的良好教材。

作为软件开发工具的第四代计算机语言，以与软件环境紧密相连为其主要特征。*Ada*是第一个提出环境与语言同时开发的新语言（美国国防部提出的语言需求计划是*STEELMAN*，环境需求计划是*STONEMAN*）。然而，本书未涉及这方面的内容。

翻译过程中，对*Ada*新概念的译名我们主要参照了中国科学院数学研究所袁崇义、徐译同同志的译法。有些地方，在行文中感到译名有些不切，因而还参照了南大徐家福、船舶公司709所王振宇等先生在有关文献上的译法。也有个别杜撰之处。由于本人水平有限，不仅译名，译文之中定有许多不妥之处，欢迎读者批评指正。

本书出版过程中，得到北京航空学院出版社大方支持。北航计算机系付主任金茂忠同志百忙中审阅了全稿，在此表示感谢。

麦中凡

1985.2.1于北航

作 者 序

*Ada*程序设计语言是竭尽全力工作的硕果。它最初是由合众国国防部高级语言工作小组于1975年为该部嵌入式系统提供一个单一的程序设计语言而研制的。

本书的目的在于给专业程序员提供一个学习*Ada*基础的方便工具。因此，假定读者已具有至少一种高级程序设计语言的知识(最好是一种象*Pascal*的块结构语言)。本书不是程序设计的启蒙教本，也不象参考手册那样涉及语言的每一细节。而是集中论述多数程序员最常使用的一些特征。

头五章包括基本的语言元素，这和多数现代过程语言没有什么不同。第一章是导论，第二章涉及整个程序流程的控制，第三和第四章是数据类型和数据结构，第五章是过程和函数。第六、七、八、九章分别包括程序包概念、类属概念、对异常处理的支援以及支持任务的语言结构。第十章专门讨论程序结构问题、名字的作用域和可见性。最后一章涉及*Ada*提供的输入和输出设施。为便于参考，附录*A*提供了语言的语法定义(按字母字符序*)，附录*B*列出了*Ada*的保留字表。

我乐于藉此机会向我的家人和朋友们致谢，感谢他们在我整理手稿和成稿期间对我的鼓励和指导。我特别要感谢我的妻子，没有她热诚的支持，本书是不可能写成的。

---

* 为便于我国读者查询，按汉字笔画序—译者

目　　录

第五章　子程序

第六章　程序包

第七章　类属子程序和类属程序包

第一章　基　础

用任何一种语言写的程序，除了最不重要的以外，都是用来处理数据的。于是，如何把那些可能遇见到的各种不同数据对象表示出来并告诉计算机就是十分重要的问题。本章中的材料将给读者提供能被*Ada*程序处理的基本元素（标识符、简单数据类型、字面量和常量）的知识。在结尾的一节中将介绍“作用域”和“可见性”的概念，以及它们和程序结构的关系。

1.1　起　点

我们从分析一个简单的程序段开始，该程序段的目的是（从普通的输入装置）读入一个用华氏温标表示的读数，把它转换为摄氏温标，并（在普通的输出装置上）写出转换结果。

```
1    declare
2          Celsius, Fahrenheit : float;
3    begin
4          get(Fahrenheit);
5          Celsius:=(Fahrenheit-32) * 5/9;
6          put(Celsius);
7    end;--程序段完
```

第一行的'**declare**'（声明）标志着这个程序段或程序块的开始，其目的在于通知编译本程序块中使用的各个变量声明将在下面出现。第三行'*begin*'（开始）标志着各个可执行语

句的开始，并与第七行终止它的 '**end**'（结束）成对出现。**begin-end** 可以看作是括着语句序列的一对括号。

第二行是"声明"语句。它的功能是建立名为"*Celsius*"和"*Fahrenheit*"的变量。它们的值当前都是不知道的。它们的"类型"包含在每一变量声明之中。类型的概念和许多现代语言(如*Algol*和*Pascal*)是一样的，并且是*Ada*语言最重要的特征之一。此外，它允许编译检查程序员是否给变量赋了一个不同类型的值，比如把一个'整型'数赋给了'字符'变量。

第四、五和六行是进行实际计算的行。第四行引用一个称之为 '*get*' 的通用系统模块，它读入一个值到变量*Fahrenheit*之中，*Fahrenheit*读到一个值之后，我们就按照某个公式将它转换为摄氏温标的一个数，再将其结果放在名为 *Celsius* 的变量之中。这是靠第五行简单"赋值"语句得到的。这条语句先求出赋值号（即":="）右边的表达式的值并"赋给"左边的变量，在此处是*Celsius*。新的值将冲掉原先存放在此处的任何值。

现在剩下的事是在第六行打印*Celsius*的值。第七行标志着由第一行'**declare**'所开始的程序段到此结束。注意，此行有一个注解。注解字符串从一对字符"--"开始，随本行结束而终止。注解被编译略去。

关于此例还有一个基本要点值得重复。*Ada*要求程序员用声明语句定义他将要使用的每一个常量和变量。所要求的信息是名字，或更正确地说是'标识符'，以及'类型'。类型指示编译如何去表示这种新的数据对象。*Ada*不支持也不允许象 *Fortran* 中所用的那种缺省声明(*default declaration*)。在使用常量和变量之前一定要程序员去定义它们所有

的属性，这样可以消除许多产生错误的潜在可能性，同时，编译时刻(*compile-time*)和运行时刻(*run-time*) 的核查也可以在较大范围内进行。

1.2 常 量

在第二、三行我们建立了可以对它们进行读、写的变量。在某些程序中，我们可能希望定义一个数据对象，它的值是知道的，而且在任何情况下该值都不改变。然而，一旦需要；在以后的日子里我们又可以很容易地改变它。库存品管理程序中就有这样的例子，该程序当库存品少于10件时自动打印订购一定数量货物的订货单。在此程序中，我们可以定义一个常量如下：

reorder_elephants : **constant** *integer*:=10;

此后就把再订货的数量叫做"*reorder_elephants*."如果将来某个时侯，我们希望手头至少保有15件，则我们只需改变一项，即在声明中再指定该常量的值，并把程序重新编译一次。与此对比，若从几千行程序正文的扫描中找数字"10"，并判定它们是否与再订货限量有关，然后将有关的改为"15"，这样，你就会看到使用常量的好处。要注意变量和常量的一个主要的差别，即一旦常量得到定义则永不再改写它（它们从不出现在赋值号的左边)。这再一次消除了出错的一个可能来源，因为若把一个语句写成：

reorder_elephants:=*first_of_month*;

而程序员的本意是写：

reorder_elephants_date:=*first_of_month*;

则编译就指出语句出错，因为被定义的常量其值是不能更改的。

尽管我们讲了许多使用常量的好处，然而也存在别的情况，在那种情况下我们绝对无误地知道一个值不会改变。比如，1公里就是1000米，从华氏转为摄氏的转换因子总是个定数。因而，将它们声明为常量就没有什么特别的优越性，我们只要在需用它们的地方把它们直接写到程序中就可以了，我们的常识会确保我们不会再去改变它们。按这种方式使用的值称为'字面'量，比如，在我们的例子中"32"，"5"、"9"都是字面量。在随后的章节中讨论各种数据类型时，我们将会更详细地看到如何使用每种类型的字面量。

1.3 字符集

对于一个特定的*Ada*编译，其允许使用的字符集要取决于实现，不过语言定义要求所有的实现至少都要提供如下字符集：

```
A B C..X Y Z
0 1 2..7 8 9
'#&'()* +, - ./ : ; <=>_|
空白字符
```

我们假定本书中*ASCII*字符集的其它字符也可用，则在我们的集合中还要加上：

```
a b c..x y z
! $ %? @ [\] ^'{ }~
```

最后，我们还假定字符集的顺序是：

```
0<1..<9<A<B..<Z<a<b..<z
```

虽然此处给出的顺序是最常用的一种，但现存系统各有差异。如果你有怀疑，請参考你们那里的编译实现指南，查看所使用的字符顺序。同样，如果你对你们那里能用的字符集不清楚，請参阅编译文档（在名叫*STANDARD*的程序包之中）。

1.4 标识符

标识符是一个字符序列，用以表示被声明的程序元素（如常量、变量、子程序等等）。标识符按以下规则生成：

(*a*) 标识符必须从字母开始。

(*b*) 后续字符可以是字母、十进制数字或下横线符。

(*c*) 标识符可由任意多个字符组成。

(*d*) 若允许用小写字母，则大小写字母没有什么区别，作为同一字符对待。

为了描述这些规则，用一种巴库斯_瑙(*Backus-Naur*)范式的变体作为速记的方便形式，如下所示：

```
标识符::=字母{[下横綫]字母_数字}
字母_数字::=字母|数字
字母::=大写_字母|小写_字母
```

小写字母所写的词（可能包括下横线符）表示的语法项目和大写_字母的一样。花括号{ }表示括在其内的部份可以重复任意次，包括零次。方括号[ ]表示括在其内的实体是可选的。下面的例子用来说明这些规则：

today ——正确

tomorrow ——也正确

next_thursday	--还是正确的
todays%	--不正确，'%'不是字母、数字或'_'
week51	--正确
4_next_week	--不正确，未从字母开头
next-week	--不正确，'-'不是字母、数字或'_'

有两种信息没有包括在标识符的定义之中，然而它们又是极端重要的，应该很好地了解，否则会导致混乱。第一，当编译检查标识符时，对大小写字母都是同样看待，*PI*，*Pi*，*pi*和*pI*都是同一标识符，因而都指同一数据对象。第二，*Ada*编译要检查标识符中的**每一**字符，以判断两个标识符是否一致，而不象某些语言的编译，只检查头六个或八个字符。

*Ada*语言中有一些具有标识符形式的词，也符合标识符的生成则规，但它被"保留"给编译自用。在开始的程序段中已经遇见了一些例子，比如，'**begin**'、'**end**'、'**declare**'。由于它们有特定的语义，程序员就不能把它们当做标识符用。在所有的示例中，保留字用黑体字印刷以区别于标识符。完整的保留字表列于附录*B*，以备参考。

1.5 简单数据类型

为了正确地操作数据对象，程序就必须明确数据对象在运行该程序的机器上是如何表示的？应该如何访问它？此对象的合法取值在什么范围内？对它可施行什么样的运算？

*Ada*中每一种'类型' 都和唯一定义该类型对象的一组属性相联系。如实型数用什么精度存贮；定义数组长度或合法取值范围的范围规格说明等都是属性的例子。两个串*'us'*和*'them'*可看成是同一串类型，但其长度属性不同，一个为2，一个为4。要注意，同一类型的所有对象都有相同的属性集合，但对某种特定的属性，不同的对象可以有不同的值。声明语句的目的就是把这些信息传达给编译。

至此，我们介绍了'类型'的概念，现在再看如何用它去生成数据对象将是有益的。生成一个给定类型的对象，并按给定类型的属性集合给对象的属性定值，然后将新建立的对象与程序员提供的标识符相结合，这些都是由'对象声用'作出的。这就叫做声明的'确立'(*'elaboration'*)(注：为了和*Ada* 参考手册所用的术语一致，我们按惯例，声明是'被确立'，表达式是'被求值'，语句是'被执行')。声明的一些例子是：

```
Period    : constant character := '.';
I_ten     : integer := 10;
R_ten     : float := 10;
count     : integer;
```

*Ada*程序根据特定应用的要求，可能需要表示许多不同的实体。例如，在科学问题中我们可能要表示向量，而在工程控制问题中我们可能要表示可用的资源。为了辅助程序员，*Ada*提供了一个内定义(*built_in*)的集合，表示了比较常见的实体或'类型'。而且还允许我们生成自己的更复杂的类型，这些类型能更好地表示我们要处理的数据。这种可以建立面

向用户类型的能力正是*Ada*最强力的特征之一。本章先讨论预定义类型：'布尔'、'字符'、'整数'、'浮点'、'定点'型。用这五种类型生成的类型以及用户自定义类型将在以后的章节中讨论。

1.6 枚举类型

有一些仅取离散值的对象，例如，表示星期中日子的变量（星期一、星期二等等），还有一年中的月份（一月、二月,等等），以及人的性别（男、女）等。为了使编译知道这种对象的哪些值是合法的，它们出现的顺序怎样，当这种类型被定义的时候(类型声明的细节請参考第3章)，它的值必须'枚举'出来。要求按这种方式定义的类型称为'枚举类型'。*Ada*语言对布尔及字符的枚举类型提供了内部定义。

1.6.1 布尔类型

布尔型数据对象用于表示真、假逻辑值。这种类型对象也只能取这两种值。*Ada*提供的布尔类型其功能和*Fortran*中的*LOGICAL*和*Algol*及*Pascal*中布尔类型是一样的。布尔类型仅有两种字面量：*'true'*和*'false'*，它的顺序是 *false*<*true*。布尔型对象的预定义运算符是：

and or xor = /=

每一运算符都要放在一对操作数之间运算(它们是'二元'运标符)，而运算符

not

是在单一的操作数上运算(它是'单目'运算符)。下面是操作

数取不同值($f=false, t=true$)时，用这些运算符得出结果的真值表：

p	**not** p
f	t
t	f

p	q	p**and**q	p**or**q	p**xor**q	$p=q$	$p/=q$
f	f	f	f	f	t	f
f	t	f	t	t	f	t
t	f	f	t	t	f	t
t	t	t	t	f	t	f

注意，在以上表格中p和q并不一定是被声明的布尔对象。它们可以是表达式，只要这些表达式求值的结果是布尔型就可以了。于是，我们可以写：

if($qty\_on\_hand<min\_qty$)**or** $special\_case$ **then** ... **endif**;

括号内的项其求值结果是布尔值，所以此处用**or**运算符是合法的。布尔变量的典型用法是控制程序的流程，如下所示：

```
end_of_file:boolean:=false;   --对象声明
if end_of_file then....endif;   --对象使用
```

枚举类型是离散类型集合中的一个成员，正如名字所指出，它们是由只能取离散值的那些类型构成的。离散类型有五种属性询问函数可用，这些函数使程序员得以判定容许值的范围和顺序（尽管对于这种类型的结果值并不特别感兴趣）：

```
boolean 'first          --容许范围内最小值(false)
boolean 'last           --容许范围内最大值(true)
boolean 'succ(false)    --序列中下一个值
boolean 'pred(true)     --序列中前一个值
boolean 'pos(x)         --布尔对象的合法范围
                        --内的x所处位置（对false
                        --是0,true是1）
```

若试图询问 *true* 的后继值或 *false* 的前导值则将会产生 *RANGE_ERROR*异常。

其他两个询问函数（*'first*和*'last*）可应用于任何标量类型（布尔、字符、整型、浮点、定点型以及其他任何枚举类型）。

1.6.2 字符类型

许多应用场合要求程序有处理字符数据的能力（例如编译，字词处理程序等）。和许多老语言诸如 *Fortran_Ⅳ* 和 *Algol_60* 不同，*Ada* 为了表示单个字符提供了内定义类型的字符。同布尔型一样，这也是枚举类型，只不过它有一个更有意思的枚举范围。作为说明，字符类型的类型声明是：

```
type character is
  (NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS, HT, LF, VT, FF, CR, SO, SI,
   DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
   CAN, EM, SUB, ESC, FS, GS, RS, US,
   ' ', '!', '"', '#', '$', '%', '&', ''',
   '(', ')', '*', '+', ',', '-', '.', '/',
```

```
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', DEL;
```

注意此处使用*ASCII*码控制字符的名字仅仅是为了清楚，它们不是标识符也不应按标识符那样用。

枚举类型的顺序按枚举时的指定严格地排好了。于是，字符类型中*'A'*小于*'B'*，*'B'*又小于*'C'*等等，从而给出了对照序列（对于熟悉*Cobol*和其他语言的读者是清楚的）。

和布尔变量一样，字符变量也有五个属性询问函数可用，下面的例子说明了它们的用法：

```
character 'first          --NUL
character 'last           --DEL
character 'pred('A')      --'@'
character 'succ('A')      --'B'
character 'pos'(BEL)      -- 7  (注：函数
                  --character'pos(character'first)
                  --的值是0)
```

现在我们再看如何利用属性询问函数来判断系统是否有小写字母字符：

if *character'last*<*lower_case_A* **then** ....**endif**;

1.7 数

为了在计算机上进行数值计算，我们需要表示数的能力。在*Ada*中有两种不同的表示数的方法，即精确的(整数)和近似的(浮点和定点数)。其区别在于任何整数（只要机器有一个充分大的二进制位数*bit*)都可以精确地表示，相反，实数只能在某个给定的精度下近似地表示（比如，在有限的位数下你如何表示分数1/3呢？)。下面三小节讨论与数值表示有关的简单数据类型。

1.7.1 整 数

整数集合中的数（如整数25,1024,0和-3)用整数类型的对象表示。如上面提到的，只要给出恰当的位数我们就能表示任何整型数。然而，在实际上是不可能的。所以*Ada*提供了属性询问函数*integer'first*和*integer'last*使程序员可以得知表示为整类型时的取值范围。例如，在用2进制补码的机器上，以16位表示整数时，

integer=-32768　　--最小整数
integer=32767　　--最大整数

重要的是要注意，若给一对象赋值，而该类型又不能表示这个过大（或过小）的值时，则将引发异常*CONSTRAINT_ERROR*,如下所示：

```
a、b、c: integer   --设16位二进制补码表示
a:=10_000;
b:=10_000;
c:=a*6;   --此处引发CONSTRAINT_ERROR
```

这种类型定义有通常的运算符，相应的操作数及其结果都是整型。它们是：

x ** y	--乘幂
x * y	--乘法
x / y	--整除
x **rem** y	--取余
x **mod** y	--求模
x + y	--加法
x − y	--减法
+ x	--单目加
− x	--单目减

运算符的优先级按下降的次序为：

**			
*	/	rem mod	
+	−		--单目的
+	−		--加法类

我们可以利用这些运算符将整型字面量、常量和变量组成整型表达式，如：

weeks * 7
years * 52 + *days*
feet * 12
length * *breadth*

并且我们能将这些表达式用于判断和赋值语句。如有以下声明：

area, *length*, *breadth* : *integer*;
hours, *days*, *overtime*, *vacation* : *integer*;

我们就可以写出如下语句形式：

```
area:=length*breadth;
```
或
```
hours:=days*8+overtime;
if hours+vaction>=max_time then....endif;
```

整型数通常不仅用于计算，程序员还把它们用作计数器和序标(*index*)：

```
for count in 1..5 loop
   put(zip_code(count));
end loop;
while x<10 loop
   put (x*x);
   x:=x+1;
end loop;
```

还要进一步说明整除运算符'／'。用上述所有运算符其结果和操作数应取同一类型(整型)。于是，表达式'4/5'将得到值为零的整型结果，而表达式'5/5'结果值为1。这对粗心大意的程序员是一个很容易上当的圈套。特别是像 *Ada* 这种语言，整除和实数除法的运算符号是一样的。与*Algol_60*和 *Algol_68* 比较起来，后者谨慎地选用了不同的符号。然而，下例可用到整除的优点，该例是计算格林威治时间和给定经度地点时间差的程序：

```
declare
     longitude,time_difference:integer;
                    direction:character;
begin
     get(longitude);   --0-180度
     get(direction); --'W'=west,'E'=east
```

```
        time_difference := longitude/15;--整除!
        if direction='W' then
          time_difference := -time_difference;
        endif;
        put(longitude);
        put(direction);
        put(time_difference);
    end;
```

1.7.2 浮点数

浮点和定点类型使我们能在计算机上近似地表示实数。二者的不同在于如何处理近似。浮点型和 *Fortran* 语言中 *REAL* 类型是等价的。浮点类型在表示指出的对象时，设置一个指定精度的相对误差界。当数如下表示时：

```
.ddddddEee        --位数=6
```

精度则根据此尾数所要求的十进制位数来定义。*Ada* 语言指明，可表示数的范围以及这些数能保持的精度，两者都和具体的实现有关。它们可用属性询问函数求出：

```
float'digits   --十进制位数的精度(是一个整型对象)
float'small    --可表示的最小正值
float'large    --可表示的最大正值
```

由于精度是按尾数的长度定义的，为方便起见提供了函数

```
float'epsilon
```

使程序员能求出表示的‘离散差’*granularity*。它是 1.0 和大于 1.0 的下一个数之差的绝对值。注意它与 *float'small* 是有区别的，因为 *float'small* 要取决于尾数和指数

二者的大小。

为浮点型对象定义的运算符是：

+　　-　　*　　/

各符运算的结果都是浮点型。还有幂运算符

**

它用于浮点对象乘一个整数幂，其结果仍为浮点型。注意，要使一个数乘一个非整数幂在 *Ada* 中是不合法的。下面的例子用来说明浮点变量的使用：

```
declare
   height,time_to_fall : float;
                    g : constant float := 32;
                          --呎/秒²
begin
   get(height);
   time_to_fall := sqrt(2 * height/g);
                          -- sqrt = 平方根
   put(time_to_fall);
end;
declare
   radius,area,circumference : float;
   pi : constant float := 3.14159;
begin
   get(radius);
   area := pi * radius * radius;
   circumference := 2 * pi * radius;
   put(area);
   put(circumference);
end;
```

注意，使用以下语句也能计算面积：

```
area := pi * radius ** 2;
```

并有完全一样的结果。

*Ada*是强类型语言，这意味着禁止程序员将不同类型的对象混合用于表达式或赋值语句。至今我们遇见的例子都设法避避这个问题。这对于任何一个在现实世界里工作的程序员是一种毫无道理的约束，因为使用不同类型的操作数或参数进行计算是极其普通的事。*Ada*中有一种机制许可这种计算，即'类型转换'，它将某种类型的表达式转换为另一种类型，这种转换是合法的。注意，它不象在*Algol*和*pascal*中那样，*Ada*要求所有类型转换以显式写出。这样做有双重的作用，一来可使需要转换的地方看起来显眼。再者可以避免编译做出程序员并不希望做的某些事。这里有几个类型转换的例子：

```
declare
    gross_pay  : float;
    hourly_pay : constant float := 3.12;
    empl_no, no_of_empl, hours, I : integer;
begin
    get(no_of_empl);
    for i in 1 .. no_of_empl loop
       get(empl_no);
       get(hours);
       gross_pay := hourly_pay * float(hours);
       put(empl_no);
       put(gross_pay);
    end loop;
```

```
end;
```

通常一个对象的值由一种类型转换为另一种类型可写成：

目标_类型（表达式）

来得到，它使表达式作为一个目标_类型的对象产生出结果。注意，类型转换仅允许在密切有关的类型之间进行。

```
declare
    degrees : integer;
    radians : float;
        pi : constant float := 3.14159;
begin
    get(degrees);
    radians := pi/180 * float(degrees);
    put(radians);
end;
```

1.7.3 定点数

除了浮点类型而外，实数还能用一个定点类型的对象来近似表示。然而，在这种情况下，设置的是绝对误差界，这不同于浮点类型的相对误差界。在某些小型计算机的应用中，实数的定点表示能使计算快速执行，而浮点计算若无附加的硬件（价值昂贵）是无法与之相比的。

定点类型的对象可应用与浮点类型相同的运算符，其属性询问函数

```
fixed'delta
```

当类型被定义后（看第3章)就会按规定返回本类型之‘离散差’。询问函数

fixed'actual_delta

则返回实现的增量（*delta*）值。定点类型的增量值取决于实现。它可以用询问函数求出，也可以从叫做 *STANDARD* 的程序包中查到。后者每个实现都提供。

1.8 作用域简介

在我们迄今所遇到的例题中，我们按*Ada*的要求，在使用之前‘声明’了我们所需用的所有的标识符。然而，并未涉及某个标识符在整个程序正文中可以访问的范围（亦即标识符何时‘可见’？何时不行？）。下面的例子可以很好地说明此问题：

```
A : declare                              --外部块A开始
      alpha,beta : character;
    begin
      get(alpha);                        --取得一字符
      B : declare                        --内部块B开始
            alpha : integer;
          begin
            get(alpha);                  --取得一整数
            put(alpha);                  --送出一整数
          end B;
      put(alpha);                        --送出一字符
    end A;
```

如果我们在*B*块中写*alpha*，此标识符指的是哪一个对象呢？这要由‘块结构’和‘可见性规则’来回答。标明

'B'的块被包容在外面的块'A'之中，因而'A'中所有的声明在'B'中是可见的，除非发生了冲突。在发生冲突的情况下则最内层的声明直接可见（除非我们采取了特殊的措施）。反之则不然。在'B'中声明的对象在'A'中均不可见。在上面的例子中如若不考虑'掩蔽'我们就只能写成：

```
A: declare
     alpha:character;
   begin
     get(alpha);
     B : declare
           alpha : integer;
         begin
           get(alpha);        --整数
           put(alpha);        --整数
           put(A.alpha);      --字符
         end B;
         put(alpha);          --字符
         put(B.alpha);        --非法，整数已不
                              --复存在
end A;
```

作用域和可见性规则是复杂的，本节仅触及皮毛。这个题目以后还要详细讨论。

第二章　流程控制

本章中我们将研究 *Ada* 程序员可以使用的基本控制结构：**loop**，**while**，**for**，**exit**，**if**，**case**，**goto**及**raise**.它们用来改变逐行执行的正常语句顺序。

2.1　循环语句

循环语句产生一个重复执行的语句序列直到指定条件满足为止。循环语句的语法定义是：

```
循环_語句::=[循环_标识符:][迭代_子句] 基本_循环
                [循环_标识符];
基本_循环::=loop
                語句_序列
            end loop
迭代_子句::=for循环_参数in[reverse]离散_范围
            |while条件
循环_参数::=标识符
```

对定义粗略地研究就可看出：我们有几种不同风格的循环，可以为某一特定的应用选择与之最相适应的一种。基本循环对同一语句序列无休止地执行。**for** 循环执行循环体到一个规定的次数。**while** 循环重复地执行循环体直到满足给定条件为止。

在我们考查各种不同循环的细节之前，关于语法定义还

有一点需要进一步澄清。尽管从定义中看尾部的循环_标识符是可选的，但如果起始的 循环_标识符 被指定了，则它是必需的。换句话说，若循环开始时给出了 循环_标识符，在其结束处若遗漏了与之匹配的标识符，则将产生编译错误。

2.1.1 基本循环

基本循环将无休止地执行循环体，如下例：

```
automatic_teller:
    loop
        get_next_transaction(transaction_type);
        case transaction_type is
            when withdrawal=>subtract;
            when deposit=> add;
            when transfer=> shuffle;
            when balance=>compute;
            when others=> null;
        end case;
    end loop automatic_teller;
```

这类循环最主要的特征是没有迭代子句。于是，一旦进入了循环就没有出路，组成循环体的语句序列将执行不止（然而，并非绝对如此。因为我们可以在循环体内设置一个 exit 或raise语句，这在以后的章节中可以看到）。

2.1.2 while 循环

我们已经介绍了基本循环，再看看用一个迭代子句来修改它会怎么样，这就产生了 while 循环：

```
find_last_item:
    while list_entry(i)/=last loop
        i :=i+1;
    end loop find_last_item;
    next;
```

在基本循环的顶端放上一个迭代子句，其作用是在每次执行循环体之前都要按 **while** 子句指定的条件求一次值。如果条件求得的是布尔'真'值，则组成循环体的语句组将执行，并且条件再一次求值。当条件值为'真'时"测试...执行，测试...执行，..."序列一直继续下去。然而，一旦求值为'假'则放弃循环去执行循环终结后的下一个语句。即示例中的"*next*"。

由于我们可以指定一个循环终止条件，**while** 循环提供给我们的是较之基本循环更为有力的工具。如下例所示，它取自应用控制系统的一个简单的查找：

```
    read(actual_position);
valve_position:
    while abs(desired_position -
        actual_position)>deadband
    loop
        error:=desired_position -
            actual_position;
        correction:=correction_calc(error);
        output_signal(correction);
        delay 5.0;
        read(actual_position);
    end loop valve_position;
    i:=1;
find_me:
```

```
while my_rec(i).name/=my_name and i
                                <=imax loop
    i:=i+1;
end loop find_me;
```

特别要注意的是有了迭代子句并不保证循环终止。设计出一个不休止的循环其可能性依然存在，这很容易证实：

```
state:=true;                --布尔变量初始值
while state=true loop       --恒为真
    null;                   --什么事也不作
end loop;
next;                       --永不执行
```

总之，**while** 循环终止于条件值最初出现‘假’值时，或是条件的求值以某种方式取决于循环体的执行，从而使得最终能获得‘假’值以退出循环。

2.1.3 for 循环

循环家族中最后一个成员是 **for** 循环。在这类循环中，迭代子句指定了一组离散值的集合，这组值于每次迭代时逐次地赋给循环_参数：

```
for i in 1..10 loop
    put(i*i);
end loop;
```

它的作用是置“i”为 1，测试 $i<=10$，若其结果为‘真’，则打印 i^2 的值。只要每次赋给的值都在 1..10 范围之内，这个运算序列一直重复地进行，直至测试 $i<=10$值为‘假’，此时打印结果是：

```
1   4   9   16   25   36   49   64   81   100
```

Ada 并不要求离散范围从 1 开始，且不限于正数。于是，我们就可以写：

```
for i in -20..-15 loop
    put(i * i);
end loop;
```

它将打印序列：

```
400   361   324   289   256   225
```

现在再看

```
for i in young..old loop
    premium(i):=i * factor(i)÷base_rate;
end loop;
```

在本例中，用变量代替字面量来指定循环借以迭代的值域。当域的一个或两个端点的值由计算决定的时候，这种写法特别有用。因为这时 **for** 循环不要求程序员在此处显式地测试而会自动地处理空域的情况（当下界超出上界时，即 *young* > *old* 时，就说该域是空域）。最后还有一句与域有关的话：在进入循环后，**for** 循环的域只计算一次，所以在循环体内不能修改它。

循环_参数'*i*'按其作用域被认为是局部于循环的变量（即它仅'存在'于循环之中），并且在循环体内它的值不能改换。因而，它不能出现在赋值语句的左端，也不能作为可修改的参数（**out** 或 **inout**）进行过程调用。循环_参数的类型派生于循环域的类型，于是，**for** 循环可用步长来通过整数域、字符域或任何其他任何类型的域，甚至是用户自定义类型的域。

常常要求以相反的顺序通过循环域，比如，从 10..1 而不是 1..10，如果我们写：

```
for i in 10..1 loop
    put(i * i);
end loop;
```

由于 i 的初值大于终值，而这是空域的定义，故本循环将不执行。因而，有一个保留字 **reverse** 用来使循环按循环_参数递减的次序取代通常的递增次序来执行循环。于是：

```
for i in reverse 1..10 loop
    put(i * i);
end loop;
```

产生的序列为：

```
100   81   64   49   36   25   16   9   4   1
```

某些说明循环的例子是：

```
work_hours:
    for i in monday..friday loop
        total_hours:=total_hours+
            daily_hours(i);
    end loop work_hours;
countdown:                          --“十..九..八..”
    for i in reverse 0..10 loop
        speak_int(i);
       delay 1.0;
    end loop countdown;
scan_again;
    while refs_satisfied_this_pass>0 loop
```

```
        rescan(library_list);
    end loop scan_again;
```

2.2 出口语句

基本循环对其自身是没有约束的，而 **for** 和 **while** 循环仅在循环开始处进行一些测试。需要的是对循环控制和测试的一般形式。*Ada* 将出口语句同前述循环结构组合起来，从而提供了这种一般形式：

出口_语句::=
exit [循环_名] [**when** 条件];

出口的用意是双重的。首先，它给程序员以自由，可在循环体内任何地方设置一个或多个终止循环的条件。第二，它显式地标出了循环的出口点。这就大大地改善了某些循环的可读性并易于编写代码。

```
get(my_record);
xfer:
    while my_record/=end_of_file loop
        put(my_record);
        get(my_record);
    end loop xfer;
```

这里主要应说明，开始的“*get*”之所以需要，仅仅是因为对这种特定情况采用 **while** 结构有其内在缺陷；它不是原来问题的一部分。现在让我们看看同样的问题如何用基本循环和 **exit** 来重写代码：

```
xfer:
    loop
```

```
        get(my_record);
        exit xfer when my_record = end_of_file;
        put(my_record);
    end loop xfer;
```

用这种结构没有多余的语句，它和 **while** 写法一样好读，可读性胜过 **if...goto**组合。注意，循环仅当 **when** 子句求值为‘真’时才终止。

exit 可选定一个循环_名。如果名字省写了，则 **exit** 语句在词法解释上是离开所属循环。如果名字加上了，则 **exit** 将转到指名循环的末端。使用 **exit** 语句从外循环跳到内循环是非法的，跳出过程或程序包也是非法的。

2.3 条件语句

条件语句根据一个或多个条件的逻辑结果从语句序列集合中选择执行的语句序列。

```
条件_語句::=
    if 条件 then
        語句_序列
    {elsif 条件 then
        語句_序列}
    [else
        語句_序列]
    end if;
```

粗略地研究上述语法定义就可以看出，我们可以生成以下语句：

```
if...then...end if;
if...then...else...end if;
```

if...**then**...**elsif**...**then**...**end if**;
if...**then**...**elsif**...**then**...**else**...**end if**;

注意，*Ada* 条件语句的语法解决了“多意**else**”问题。这在 *Algol_60* 及其派生的 *Pascal* 之中都是生来就有的。下面给出的例子中可说明这个问题（它们在*Ada*的语法中是非法的）：

if *a* **then if** *b* **then** *put*(“*B*”)**else** *put*(“*C*”);

由于两个 **if** 语句都隐含地终止于最后的分号，本例有两种解释：

```
if a then
    if b then put( "B" )
    else put ( "C" );        --要 a 为真时打印'C'
  或
if a then
    if b then put ( "B" )
else put( "C" );             --要 a 为假时打印'C'
```

看来在 *Ada* 语法中解决这个问题要比 *Pascal* 用语句括号限制路径以及 *Fortran 77* 中 *IF* 块的隐含解决办法都要好。

从上面可以看出，执行 **if** 语句时先求条件值，如果其结果为‘真’，则跟在 **then** 后面的语句序列被执行。这个序列的最后一个语句被执行之后，则执行跟着 **end if** 语句的下一语句。

如果初始条件求值为‘假’，则控制跳到 **elsif** 或 **else** 子句，且相应的语句被执行。这些语句序列的最后一个语句被执行之后，和前面一样，就转去执行跟在 **end if** 后面的

语句。

几个 *Ada* **if** 语句的合法例子是：

```
if deposits<withdrawals then
    put( "You are overdrawn" );
end if;
if aircraft.trajectory=inbound then
    if aircraft.id/=ours then
        launch_interceptors;
    else
        notify_operator;
    end if;
end if;
if trip_length<=short then
    pack(toothbrush);
elsif trip_length<=medium then
    pack(small_bag);
else
    pack(trunk);
end if;
```

2.4 情况语句

情况语句据据一个表达式的值在一组相互独立的可选语句序列中挑选要执行的语句序列：

```
情况_語句::=
    case 表达式 is
        { when 选择 { |选择 } =>語句_序列 }
    end case;
选择::=簡单_表达式|离散_范围|others
```

它的功能类似于 *Algol-60* 的 *SWITCH* 语句，*Fortran* 的计算转(*Computed-GOTO*)语句，以及*Pascal* 和 *Algol-68* 的 *CASE* 语句。将给定表达式求值（此表达式取值必须是离散类型之一），并测试其结果和哪种选择匹配。如果匹配找到了，则相应的语句序列被执行，随后控制转移到跟在 **end case**后面的语句。

由于它给程序员在实现多路选择时比多重 **if** 语句更为方便的方法，所以它是一个特别有用的结构。作为一个例子，我们看看用 **if** 语句实现的一个功能：

```
get(operand_type);
if operand_type=single then
    single_mult;
elsif operand_type=double then
    double_mult;
elsif operand_type=complex then
    complex_mult;
elsif operand_type=vector then
    vector_mult;
else
    put( "Operand Error" );
end if;
```

同样的功能，下面用 **case** 语句实现：

```
get(operand_type);
case operand_type is
    when single  =>single_mult;
    when double  =>double_mult;
    when complex =>complex_mult;
```

```
        when vector     =>vector_mult;
        when others     =>put( "Operand Error" );
    end case;
```

这个功能用 **case** 语句实现较之多重 **if** 语句更易读和更易懂，从而减少了程序设计易犯的错误。

when 子句不限于单一选择，我们可用‘或’将几个可能合在一个选择里。甚至对我们希望的一个或几个要发生的特定动作定义一个值域，如下面两个例子：

```
case keyboard_input is
    when                "0"..“9” =>
                                numeric_convert;
    when  “A”..“Z” | “a”..“z” =>
                                alpha_code;
    when “ ”..“/” | “<”..“@”
         |“[”..“‘” | “{”..“~”=>
                                special_code;
    when               | others  =>
                                control_code;
end case;
case weather is
    when    sunny =>shorts;
    when   cloudy =>jacket;
    when  raining =>coat;
                    umbrella;
    when  snowing=>coat;
                    umbrella;
                    hat;
                    boots;
end case;
```

注意 **others** 子句，它提供了一个简单的机制，允许程序员指定当 **case** 表达式为所有其他值时要做的动作，而无需将那些值一一列出。但这并不意味着程序员可以忽视对表达式的**每一个**可能的值指定执行的动作。更正确地说，它是列出未指明量的一种简写。这点和某些其他语言大为不同，比如 *Pascal* 就要求对表达式每一个可能值显式地列出动作表，同时它也优于 *Algol-68* 的“*out*”。

2.5 转移语句

*Ada*中转移语句的功能和*Algol*，*Fortran*及*Pascal* 中 *GOTO* 语句完全一样，它允许程序员实现一个无条件的跳跃，即从它的程序的某个地方跳到另一个地方。其语法是：

转移_語句::=**goto** 标号_名
标号::=<<标识符>>

标识 **goto** 目标的方法是对所需求的语句作‘标号’。标号是括在一对双尖括号内的标识符。这一点和必须是数字的 *Fortran* 或 *Pascal* 的标号明显不同。一个示例性程序片断是这样的：

```
get(x);
get(y);
if x=y then
    goto finish;
else
    process(x,y);
end if;
...
```

```
...
<<finish>>put("AllDone.");
```

既然标号是标识符，很自然地就想知道它的声明及其作用域。问题在于标号是隐式声明，它在最内层声明所包含的体内。这就定义了它的作用域，也就定义了它的可见性，因而转移就限制在这部分程序之中。

注意，程序员并不限于在一个语句上标以唯一的标号，他（或她）可以随心所欲地使用许多必需的标号。于是，我们就可以有许多不同的转移标号都转到同一个下一步要执行的语句上。

```
                ...
                goto finish;
                ...
<<done>>
<<finish>>
<<way_out>>
                next;
                ...
                goto way_out;
                ...
                goto finish;
```

goto 语句的使用是当前有争议的问题，关于它对良好程序设计实践会有什么影响，有许多不同的观点。某些结构化程序设计方法学的提倡者建议在语言中完全取消它，因为任何用 **goto** 形成的结构都能用 **if**，**while**等语句改写。从而将它取消。

如果仔细地使用 goto，则结构程序方法学的应用和使用 goto 并非天生来就不相容。它们在某些場合还是强有力的工具，比如紧急出口或是执行速度要求特别高的地方，诸如中断处理程序等等。总的说来，使用 goto 所蒙受的损失要比从中受益大得多，而少数例外的受益事例还在日趋减少，因而要尽可能避免使用 goto 语句，除非设有替代的办法。goto 是一把双刃剑，要一个熟练的高手才能使它得到最好的应用。

2.6 引发语句

引发语句用于指示一个错误条件产生或指出一个要求由'异常处理段'来处理的异常发生。它的作用类似于执行一个 goto 语句，用 raise 代替 goto，其区别在于启动异常处理段。引发语句的语法是：

引发_語句::=raise [异常_名]

异常处理和引发语句的全貌将在后面的章节中讲述，这里提到它是因为它能改变程序的流程。

第三章　类型与简单数据

由于软件和软件工程领域的发展，语言研究的重点已从问题的算法描述（说明什么样的操作要被实行）转向面向数据的观点（说明要控制什么样的对象）。这就导致在数据表示和数据描述方面更多的研究，这方面的一些劳动成果现在已经在许多语言，如 *Ada* 中体现了。

本章的主题是数据的表示和描述，着重 *Ada* 的‘类型’概念。我们将会看到程序员如何定义自己的类型，以及如何从简单的类型组成复杂的类型。它给程序员以变换他的数据描述来适应自己的需要的能力，这也是 *Ada* 最有力的特征之一。

3.1　类型声明

在上面几章中，我们已知道如何声明一个给定内定义类型的对象，随后又如何使之取得那种对象的值。然而，在有些情况下，内定义类型不能或不足以满足手中作业的需要。比如，在一个程序中大陆用整数表示（1＝北美洲，2＝欧洲，3＝亚洲等等）程序员就不得不记住它（或记在文挡上）。这是很麻烦的。此时实际需要的是一种使程序员可以建立他自己类型的手段，随后他就可以很方便地表示数据。*Ada* 中称之为‘类型声明’的机制可以使你做到这点，它为编译引入一个新类型。一旦一个新类型被声明，以后它就可以和内

定义类型一样用于该类型的对象声明。用戶定义的类型在功能上和內定义类型沒有什么区别。

类型声明语句的语法定义是：

```
类型_声明::=
    type 标识符[判别式_部分]is 类型_定义
  | 不完全_类型_声明
类型_定义::=
    枚举_类型_定义 | 整_类型_定义
  | 实 _类型_定义 | 数組_类型_定义
  | 記录_类型_定义 | 訪問_类型_定义
  | 派生_类型_定义 | 私有_类型_定义
```

程序员定义一个新类型时，他就指明了一个特定的属性集合而提供给该类型的对象。这些属性包括该类型对象合法的取值范围以及可在其上或两对象间进行的操作。本章以下部分及整个的下一章都用来讨论各种不同的 类型_定义 以及它们內在特性的细节。

3.2 标量类型和离散类型

在我们考察单个 类型_定义 之前先研究一下‘标量’和‘离散’类型的概念是有好处的。标量类型有以下特征：

1. 它的值沒有分量。
2. 它的值的集合是有序的。
3. 值的集合有一个确定的域。

Ada 允许程序员用属性询问函数来求某一标量类型的合法值域。对于任何标量类型（例如，*my_type*）为找出这种类

型在实现上的最小值和最大值，可写成：

my_type'first　　*my_type* 类型的最小值
my_type'last　　*my_type* 类型的最大值

离散类型是标量类型的子集。离散类型由枚举类型和整数类型组成，由于离散类型值的集合是离散量组成的，所以每个值都有一个‘位置数’，该数规定了那个值在序列中的位置。同样，一个特定的离散值总有一个前导值和一个后继值（端点显然是例外）。一个值的前导值和后继值可以用询问函数求出：

my_type'pos(x)　　*x* 的位置数
my_type'pred(x)　　*x* 的前导值
my_type'succ(x)　　*x* 的后继值

注意，在法定范围内的‘最小值’的位置数是零。
(*my_type'pos(my_type'first)*=0)

在本章的后面我们还会看到这些函数如何使用。

为了说明上述论点，我们小结为：预定义类型的‘定点型’、‘浮点型’、‘整型’和‘字符型’都是标量类型，但只有‘整型’和‘字符型’是离散型。

3.3 枚举类型

枚举类型的值为离散量的有序集合所组成。我们在第1章讨论布尔型和字符型时已经见过枚举型的例子。枚举类型的语法定义是：

枚举_类型_定义::=
　　(枚举_字面量{,枚举_字面量})

枚举_字面量::=标识符|字符_字面量

类型_定义 给出的字面量其顺序是重要的，因为它定义了一个值和下一个值的关系。于是，如果我们想定义一个表示罗盘方位的类型如：

```
type cardinal is (north, east, south, west);
```

那么它们彼此的关系是：

$north < east < south < west$

由于这个关系要影响到条件判断的结果，故其次序是重要的。很容易设想出两个算法相同的程序代码片段，其结果不同：

```
declare
    type cardinal is(north,east,south,west);
    direction:cardinal;--定义一个'cardinal'变量
begin
    ...
    direction:=south;
    ...
    if direction<east then
        correct_drift;--由于 south>east，不执行
    end if;
end;
declare
    typecardinal is (north,south,east,west);
    direction:cardinal;--定义一个'cardinal'变量
begin
    ...
```

```
        direction:=south;
        ...
        if direction<east then
            correct_drift;--由于 south<east，执行
        end if;
    end;
```

掌握定义自己的枚举类型的能力是非常有用的。例如，在正文处理的应用中可能要求我们为旧式排字机表示出希腊字母。我们就将这些字母定义为我们自己的类型，而不是把每个希腊字母设成一个数（如 1=α，2=β 等等）。然后生成并操作这种类型的对象。在本例中：

```
declare
    type greek is
        (alpha,   beta,   gamma,  delta,
         epsilon, zeta,   eta,    theta,
         iota,    kappa,  lambda, mu,
         nu,      xi,     omicron, pi,
         rho,     sigma,  tau,     upsilon,
         phi,     chi,    psi,     omega);
    g_char : greek;--定义一个'greek'类型的变量
    alt_set : boolean;
begin
    ...
    if alt_set then
        read_greek(g_char);
        typeset(g_char);
    end if;
    ...
end;
```

使用用户_定义的类型而不用 1,2,3... 数字系统还有一额外的好处，它使编译得以检查类型的兼容性。记住，在沒有显式的类型_转换之时，*Ada* 不允许在一个表达式中混用类型。于是，编译就能检查出可能的错误，如以下情况：

```
declare
    line_number : integer;
    gchar       : greek;
begin
    ...
    line_number := alpha; --类型混用，不允许
    ...
    line_number := 1;     --如果'1'是行数，合
                          --法。如果'1'表示
                          --'alpha',语法上合法，
                          --逻辑上不正确
end;
```

下面的例子说明属性询问函数的使用，本例用枚举类型：

```
declare
    type hand is (thumb,index,middle,ring,
                                  little);
    this_finger     :hand;  --定义一个'hand'对象
    this_finger_no  :integer;
    max_fingers     :constant integer :=
                        hand'pos(hand'last);--4
begin
```

```
    ...
    read_hand(this_finger);--'thumb','index',
                                --等等
    this_finger_no  :=hand'pos(this_finger);
    put(this_finger_no); --'0','1',等等
    ...
end;
```

如果你去复习一下第 1 章讲述布尔型和字符型的有关章节，它们的那些表示法也适用于枚举类型，这点现在是再清楚不过了。

现在再考察一个情况，程序员需要定义两个（或多个）新的枚举类型，但他发现某些字面量在不同的类型中都出现，如以下片段：

```
type body is(head,arms,legs,trunk);
    type table is(top,drawers,legs);
```

这种情况下，一个字面量有多种解释，即所谓的‘重载’。*Ada* 中用重载字面量去定义类型完全可以接受，然而，在使用它们时要特别小心。在使用重载字面量之时，其类型必须能由上下文确定或是用下述形式的‘限制表达式’将所希望的类型显式地标出：

```
类型'(表达式)
```

为了说明，看例中注解：

```
declare
    type desk is(top,drawers,legs);
    type table is(top,leaves,legs);
```

```
begin
    ...
    write(desk'(legs));    --由于'legs'重载，有
                           --限定要求
    ...
    write(leaves);         --无限定要求
    ...
    for i in lable'(top)   --legs loop
                           --或是限定'legs'，或
                           --是限定'top'
    ...
    ...                    --按限定的类型决定本
                           --范围类型
    end loop;
end;
```

3.4 整类型

第 1 章我们已经介绍了整类型。然而，在许多程序设计的实践中，整变量仅仅只用到预定义‘整’类型值域的很小子集。考虑以下例子：

```
declare
    day_number,page_number     :integer;
    number_of_grains           :integer;
begin
    ...
    read(number_of_grains);    --比如取10,000
    day_number :=number_of_grains - 3;
    page_number := - 273;
    ...
end;
```

在这种情况下使用整型变量有严重的毛病。由于所有的变量和所有的字面量都是整型，没有类型不兼容的问题，也不存在上溢或下溢的错误，因而无论在编译时或是运行时都无法查出负的页数或一年的天数达 9997 天之类的错误。此时就要求有一种机制去定义一个新的、有限范围的整数类型，以便在执行中进行合理性检查。整_类型_定义就可达到此种功能，它允许程序员声明一个从内定义整类型中派生出来的新整类型，其语法定义是：

整_类型_定义::=约束_范围

现在我们改写以上的例子，这次利用这一新的特征：

```
declare
    type days is range  1..366;     -- 包括闰年
    type pages is range 1..5000;    -- 一个恰当的数
    day_number          :days;
    page_number         :pages;
    number_of_grains    :integer;
begin
    ...
    read(number_of_grains);         -- 比如取10,000
    day_number:=number_of_grains-3;
        --这里将产生一个错误，因为9,997不在类型
        --'days'对象的合法范围内
    page_number:=-273;
        --又一个错误，因为'pages'的对象不能取
        --负值
    ...
end;
```

在改写的例子中要注意到：在编译时或运行时对单个变量作值域检查要有足够的信息。

语言并不要求我们把新类型的值域限制在正数范围。如果情况需要，我们可以定义为任何所需的范围，只要限制在非空域或不超出实现内定义的整类型范围就可以了，新类型由内定义类型派生。记住这几点，则以下整_类型_声明都是合法的：

```
type       tolerance is -20..20;
type        sub_zero  is -50..-1;
type  near_absolute  is -273..-230;
type  serial_number is1..1000;
```

由于新声明的类型派生于原来的整类型，它们‘继承’了为整型对象定义的功能及其算符。因而，我们即使定义了新类型也勿需对其功能和算符重新定义。

特别要注意的是，无论何时当程序员使用了类型定义，他就引入了一个新类型。新类型不同于其他类型，因而在表达式中不能和别的类型混用，即使标识了相同的特性也不行，除非符合类型兼容规则：

```
declare
    type little_int   is range 1..5;
    type small_int    is range 1..5;
    little_var              :little_int;
    small_var,tiny_var :small_int;
```

```
begin
    ...
    little_var    :=3;
    small_var     :=little_var;--非法，类型混用
    small_var     :=small_int(little_var);
                                    --许可，因为有一
                                    --显式的类型转换
    tiny_var      :=small_var       --不需要转换，因
                                    --为 small_var 和
                                    --tiny_var 是用
                                    --同一类型_定义来
                                    --声明的，因而它
                                    --们的类型相同
end;
```

要注意，类型转换只允许在数值对象或表达式（包括实类型）之间、在数组类型之间、以及在派生类型（看 3.6 节）之间进行（只有这样才有意义）。

3.5 实类型

实类型由浮点类型和定点类型组成。我们在第 1 章中已碰到过内定义的浮点类型和定点类型。

在定义一个新实型时，我们主要关心的是规定一个所要表示的实数的精度要求(及其可能的值域)。因而，看到下述实_类型_定义就不足为奇了：

```
实_类型_定义::=精度_約束
精度_約束::=浮点_約束|定点_約束
```

3.5.1 浮点类型

浮点类型的精度要求是根据该数的尾数（当以10为基数表示该数时）所要求的精确十进位制的位数来规定的。

浮点_约束::=
    digits 静态_简单_表达式[范围_约束]

定义的位数为表示和处理新类型对象设置了一个精度下限。一个具体的实现可以选用比实际要求更高的精度。但是如果你想写一个可移植的程序时，则只需依据你所要求的精度而勿需别的。注意，在保留字 **digits** 后面的表达式是静态的，即必须有可能在编译时刻求出其值。显然，你不能要求高于预定义'浮点'（实现时叫'长浮点'）类型的精度约束。浮点类型_声明的几个例子是：

```
type small_float is digits    5;
type   mid_float is digits    7 range 1.0..1.0E6;
type   big_float is digits   10 range 0.0..1.0E3;
```

类型 *small_float* 的对象可以表示精度为5位的数，如2.7183，2.7183*E*5，－27.183*E*4 等。类型 *mid_float* 的对象可表示3.141593，314159.3等。但不能是－3.141593，因为它超出了范围约束。

对于任何浮点类型（例中是 *my_float*）程序员都可以用以下询问函数了解它们在实现上的情况：

my_float'digits	本类型对象合法精度的最大十进制数字的位数（本身是整数）

my_float'mantissa	尾数的二进制位数（整数）
my_float'emax	指数“域”，是 ±*my_float'emax*
my_float'small	类型 *my_float* 的最小正数
my_float'large	类型 *my_float* 的最大正数
my_float'epsilon	这种表示法的‘离散差’的大小

如果你想使用这些函数，则须好好地读一下 *Ada* 的标准以及你那个地方机器的参考手册中关于浮点表示的细节。

3.5.2 定点类型

定点类型的精度约束其定义为：

定点_約束::=

delta 静态_簡单_表达式[范围_約束]

有一点很重要，尽管此定义表示的 范围_約束 是可选的，但它只对子类型指示可选（看第 3.6 节）。定点类型必须定义它，因为让编译去决定它所必需的表示是不可能的，除非 *delta*（增量）和范围_约束两者皆知。

如第 1 章所指出，定点类型和浮点类型的差别仅在于精度约束。浮点类型的精度是指保留有效值的位数，而不管它的指数。于是，浮点类型若定义为：

type *s_float* **is digit** 3;

则用它生成的对象可取值为 1.23，123000，0.000123，因为其中每一个数都可改写为 $0.123Ex$，此处 $x=1,6,-3$。精度约束只关系到数的长短，即有效数字数位。然而，定点

型的精度约束是绝对的，是十进制位数。于是，如果我们要用以下类型_声明来定义一个定点_类型：

```
type s_fixed is delta 0.1 range 0.0..2000.0;
```

则它可以取值为 12.3,123.0,1230.0 等，但决不可能取1.23的值，因为后者要求的精度是两个十进制位，大于本类型定义的位数。

用定点操作数作乘除法时，其中间结果的精度是任意(高)的。因此，在将它们赋给定点对象之前，必须执行一个隐式的类型_转换。为了说明这点：

```
declare
    type s_fixed is delta 0.1 range -100.0..100.0
    small   :s_fixed;
begin
    ...
    small:=2.1;
    small:=small+3.2; --small现在是 5.3
    small:=small*small;
                --计算的中间值是 28.09 当转换到与
                --'small' 一致时，不知不觉地出现了
                --误差
    ...
end;
```

当考虑使用定点算法时，必须记住舍入和截断误差的可能性。

定点类型的属性询问函数是：

```
my_type'delta          本类型精度的增量
```

my_type'actual_delta	被实现的增量
my_type'large	*my_type*的最大数

总之，除非有硬性的理由才使用定点算法，用浮点运算可能比较容易而且可靠。

3.6 子类型和派生类型

我们已经讨论了标量类型，现在我们再到‘子类型’和‘派生类型’的园地里浏览一下。

在 *Ada* 中，我们不仅能声明‘类型’，还能声明‘子类型’。顾名思意，子类型是一个现存类型的子集，其语法为：

```
子类型_声明::=
    subtype 标识符 is 子类型_指示
子类型_指示::=类型_标記[約束]
类型_标記::=类型_名|子类型_名
約束::= 范围_約束|精度_約束
        |序标_約束|判别式_約束
```

声明一个子类型时，要算出它的约束（如果有的话），然后将‘基’类型及约束与子类型的名字相结合。以后就可用于生成新的‘子类型’对象了。这似乎只是在复制类型声明的功能。那要它做什么用呢？问题在于一个给定子类型的对象，当它用于包含属于其父类型或‘基’类型对象的表达式时，勿需类型转换，于是，如果我们想生成一个从 0 到 100 范围内的整数计数器，我们就可以写：

```
declare
    subtype small_int is integer range 0..100;
    counter :small_int;
    start   :integer;
begin
    ...
    counter:=start; --不需要类型转换!
    ...
end;
```

用子类型去实现现存类型的子集，它的好处是程序员用起来容易。由于子类型声明并未引进一个新类型，所以子类型和‘基’类型有相同的属性（子类型声明强加的任何约束除外）。

注意，子类型声明中的任何约束必须和基类型兼容。例如，你不能写成下面的样子：

```
type a is integer range 1..10;
    subtype b is a range 5..12;--约束错,12>10
```

派生类型类似于子类型，不同的是派生类型引进一个从现存类型派生出的新类型，派生类型继承了父类型的性质。派生类型的定义是：

派生_类型_定义::=new 子类型_指示

定义了派生类型，程序员就为编译引进一个其性质和父类型一致的类型，但是它仍然是一个不同的类型。因而，如果没有显式的类型转换，它就不能和其父类型混用。我们以前讨论的整类型声明，实际上是声明了一个新的派生类型，于是：

```
type my_int is range 1..10;
```

等价于声明

```
type my_int is new integer range 1..10;
```

子类型和派生类型声明的几个例子是：

```
type revision is range 1..20;
type old_revs is new revision;              --派生
type new_revs is new revision range 15..20
                                            --派生
subtype a_rev is revision range 15..20      --子类型
```

这些例子中，'revision' 是 'old_revs' 和 'new_revs' 的父类型，是 'a_rev' 的基类型。注意，'revision'，'old_revs' 及 'new_revs' 三者是不同的类型，因而不能在一个表达式内混用，而'a_rev'是 'revision' 的子类型，所以它可以和 'revision' 的对象混用。但不能和其他两个混用。

要注意子类型和派生类型并不限于在标量类型中使用，我们在下章将会看到它们也能用于结构类型。

第四章　结构型数据

4.1　数组类型

我们现在研究第一种结构类型——数组类型。数组对象可以看作是同一类型子对象的序列，每一成员都可用序标(*index*)从序列中引用。因而，串对象"*abcdef*"可以看作是'字符'类型对象的数组，'*a*'由序标1引用，'*b*'由序标2引用等等。数组的概念和多数语言是一样的，这包括*Fortran*，*Algol*，*Pascal*等等，对于多数读者都没有什么新奇的东西。要知道如何生成一个数组对象首先要研究数组类型定义：

数组_类型_定义::=
　　array（序标{，序标}）**of**成分_子类型_指示
　　array序标_约束**of**成分_子类型_指示
序标::=类型_标记**range**<>
序标_约束::=（离散_范围{，离散_范围}）
离散_范围::=类型_标记[范围_约束]|范围

从定义中看出，我们可以产生两种数组类型，上边一种仅用序标，下边一种用序标约束。*Ada*中，按上边那种形式定义的数组称为'无约束数组类型定义'，按下边那种形式定义的是'约束数组类型定义'。

无约束数组类型定义用来生成其序标范围事先不定的数

组类型（序标范围决定了数组的长度），并且我们希望先不对此做出决定，直到我们实际生成这种类型对象时再定。于是，我们可以写：

```
type vector is array (integer range<>) of float;
```

它将引入类型'vector'，由此类型我们可以生成好几种不同长度和不同序标范围的'vector'对象，但它们都是一维的，且每个都由'浮点'对象组成。

```
type vector is array (integer range<>) of float;
a,vector( -1..10);--序标为整数在 - 1 到10的范围内
b,vector(  0..10);--同一类型，但序标界不同
c,vector( 20..50);--同上
```

注意，序标类型的规格说明是在数组类型声明中完成的，但序标值的范围直到对象声明前均未定义。任何离散类型都可用作序标，这就允许我们写：

```
type continent is (Africa, Asia, America,
     Europe, Australia, Antarctica);
type population is array (continent range<>)
                                  of integer;
a, population (Africa..Australia);
```

显然，类型声明中指出的序标类型和对象声明中序标约束的类型如果不一致，将会导致错误。

我们并不限于单个序标，如果需要可以增加，序标的个数就叫做数组"维数"。于是，a，b和c都是一维数组，二维数组的实例是：

```
type vector_2d is array (integer range<>,
      integer range<>) of float;
d:vector_2d(1..20,1..50);
```

要特别注意，序标的次序是很重要的。在上例中 $d(1,2)$ 和 $d(2,1)$ 并不是引用同一个对象，同样，第一个序标的上界是20，第二个是50。换句话说，序标与所据位置有关。

我们再来考察约束数组类型定义。同只能用于类型声明的无约束数组类型定义相反，约束数组类型定义在类型和对象声明中均可使用。其实，所有上述例子都可用约束数组类型在其对象声明中定义。约束定义使我们能把序标的类型和范围作为类型的一部分或在对象声明中确定下来，如下所示：

```
type matrix_2d is array
          (integer1..20, integer1..50) of float;
e:matrix_2d;--上下界已在类型定义中确定!
```

我们也可以不用过渡声明'matrix_2d'同样可声明'e'，如：

```
e:array (integer 1..20, integer 1..50) of float;
    --上下界在对象声明中确定
f: array (1..20, 1..50) of float;
    --序标域的类型缺省了'整型'
```

类型或对象的上下界并未要求用字面量给出，用变量也是完全合法的：

```
g:array (m..n, o..p, q..r) of float;
```

此处序标的类型如果没有指明，则假定为整型。此种序标界

是柔性的数组，称之为动态数组。要注意，序标的类型必须是离散类型之一，因为如果一个数组的序标值为0.75，那是讲不通的。

当数组声明被确立时，数组的界就决定了。因而，如果我们有一动态数组，当确立发生时，定义它上下界的变量必须从声明中可见，并有合法的值。

如同我们见过的其他类型一样，数组类型也有属性询问函数。例如，对于某一数组类型'*vec*'，这些函数是：

vec'first	第一个序标的下界
vec'last	第一个序标的上界
vec'length	第一序标值的个数
vec'range	用 *vec'first..vec'last* 范围定义的子类型
vec'first(n)	第*n*个序标的下界
vec'last(n)	第*n*个序标的上界
vec'length(n)	第*n*序标值的个数
vec'range(n)	用*vec'first(n)..vec'last(n)*范围定义的子类型

我们考察计算 2 维矩阵各元素总和的功能是如何完成的，借以说明询问函数的使用：

```
function sum(matrix:vector)return float is
    temp:float:=0.0;
begin
    for i in matrix'first..matrix'last loop
        for j in matrix'first(2)..
                              matrix'last(2) loop
```

```
            temp:=temp+matrix(i,j);
        end loop;
    end loop;
    return temp;
end sum;
```

有一些操作可以在整个数组对象或部分数组对象上实施，没有必要只对数组包含的单个成分进行操作。例如，我们可以初始化或复制数组的全部或一部分，如下所示：

```
declare
    type my_vec is array(1..10)of integer;
    x,y:my_vec;
begin
    ...
    x(1..5) :=0;--赋给x的头5个元素
    x(6..10):=1;--赋给x的后5个元素
                --x=0,0,0,0,0,1,1,1,1,1
    y:=x;       --y现在和x的值一样
    x(1..3) :=y(7..9);
                --x=1,1,1,0,0,1,1,1,1,1
    x(4..5) :=x(1..2);
                --x=1,1,1,1,1,1,1,1,1,1
end;
```

还有另一个方法给数组赋值——使用聚集值。聚集是一种描述由成分建立值的方法，它的语法是：

聚集::=（成分_結合{，成分_結合}）
成分_結合::=［选择{|选择}=>］表达式

聚集赋值可用三种方式中任何一种来指定聚集成分和个别数

组元素之间的对应关系：即按位置，按范围，按序标值。下面几例表示用不同的聚集值给数组赋值：

```
declare
    type my_array is array(1..01)of integer;
    type vector_3 is array(1..10,1..20,1..30)of
                                        integer;
    a:my_array:=(1..10=>0);--所有元素置零
    b:vector_3;--未初始化
degin
    ...
    a:=(1..5=>1,6..10=>3);
        --同上面一样按域找对应
        --a=1,1,1,1,1,3,3,3,3,3
    ...
    a:=(1,1,2,2,3,3,4,4,5,5,);
        --按序列中的位置找对应
    ...
    a:=(1|3|5|7|9=>1,others=>0);
        --按序标找对应
        --a=1,0,1,0,1,0,1,0,1,0
    ...
    b:=(1..10=>(1..20=>(1..30=>0)));
        --三维数组,因此聚集是按二维成分(它又作
        --为单个成分的一维数组)的一维聚集给出
    ...
end;
```

以下规则适用于给数组赋聚集值：

1. 在聚集中使用选择时，每个选择能且只能出现一次。
2. 在聚集中使用域是明确指明包在域的上下界之间所

有中间序标值的较省事的办法。

3. **others**术语仅能出现在成分表的最后一项。

在4.3节中我们将要看到使用聚集给记录赋值和在数组中使用它们极其相似。

在我们对数组的所有讨论之中，我们不是说整个数组就是说它的一个成分。*Ada*为访问数组提供了第三种机制，即数组片。数组片是表示数组子集的一维数组。作为说明，我们若有以下数组定义：

```
type vector is array(1..20)of float;
my_vector:vector;
```

则以下都是*my_vector*的数组片：

```
my_vector(1..5)
my_vector(1..19)
my_vector(19..20)
my_vector(20..19)--空片，由于序标域为空
                  --（下界应为上界之前导值）
```

注意，数组片的类型是命名数组的基类型。

4.2 串类型

为了程序员的方便，预定义有'串'类型。这种类型等价于类型声明：

```
subtype natural is integer range 1..integer'last;
type string is array(natural range<>)of character;
```

'串'对象是'字符'对象的一维数组，它的长度或按数组通常的办法来定义，或由赋初值来隐含地定义。

```
line_buffer :string(1..132);
blank_buffer:string(1..132):=(1..132=>' ');
header      :constant string:="My Sample
                               String";
   --
   --注：字符字面量括在撇号'和'之中
   --串字面量是括在引号"和"之中
   --
```

顺便插一句，要注意，给串赋初值是给数组赋聚集值的一个特例。

有一种被定义在两个'串'对象之间使用的运算符 &，即并运算符。用这种运算符我们可以写：

```
help     :constant string:="Help";
me       :constant string:="Me";
helpme   :constant string: = help & me;
                          --"HelpMe"
```

为串还定义了关系运算符<=,<,>,>=,其运算结果取决于字典顺序。例如，"a"小于"b"，同样，"aa"也小于"b"。和预定义'字符'类型一样，这个顺序定义了一个核对次序。

4.3 记录类型

记录和数组相似，因为它们都是由"较小的"对象组成的结构型数据。然而，记录又和数组不同，因为它可以由不同类型的对象组成，而且这些成分对象要用名字引用。它类似于*Cobol*中记录的概念。记录类型的语法定义是：

记录_类型_定义::=

```
    record
        成分_表
    end record;
成分_表::=｛成分_声明｝[变体_部分]|null;
成分_声明::=
    标识符_表:子类型_指示[:=表达式];
  |标识符_表:数組_类型_定义[:=表达式];
```

在此，有几个例子能说明记录类型的定义：

```
type person is
    record
        surname :string(1..20=>'');--缺省值
                                    --为空白
        age     :integer range 0..130;
        gender  :sex;     --男，女
        married :boolean;--已婚、未婚
    end record;
type automobile is
    record
        maker   :manufacturer;
        colour  :paints;
        import  :boolean;
        year    :integer range 1900..2000;
    end record;
type address is
    record
        number  :integer:=-1;--缺省值是-1
        streel  :string(1..80=>''); --所有串的
        town    :string(1..80=>''); --缺省值都
        county  :string(1..80=>''); --是空白
    end rceord;
```

照例，成分类型和该类型对象的缺省初值的类型必须兼容，否则将产生一个错误。

我们现在已经知道如何生成某些记录类型了，但我们还必需学会如何生成记录对象，以及如何访问和运用它们。*Ada*中取接记录的一部分是由选定一个"成分"来实现，它的语法是：

选择：成分::=名字.标识符
　　|名字.**all**|名字.运算符_符号

我们暂时回避名字 **all**和名字.运算符_符号的选择，因为它们不适用于记录。这样，在访问一个记录成分时，就剩下简单的'名字.标识符'选择，其中'名字'就是记录的名字，'标识符'就是该成分的标识符。下一个例子可以看到如何使用上面给出的类型定义去声明某个记录对象，以及如何访问新声明对象的成分：

```
declare
    mary,john :person;     --注：姓的缺省值是空白
    home,base :address;    --街道、城镇、县的缺省
                           --值也是空白
    car       :automobile;--本类型无缺省值
begin
    mary.surname    := "jones";
    john.surname    :=mary.surname;
    john.gender     :=male;
    car             :=(ford,red,false,1981);
        --这是使用位置成分对应的记录聚集的例子
```

```
    car                 :(import=>false,
                          colour=>red,
                          year  =>1981,
                          maker =>ford);
        --这是使用指名成分的记录聚集的例子
    home.town       := "perth";
    base            :=home;--整个记录赋值
end;
```

注意这里如何使用聚集记录值初始化整个记录。为了求得相应的值上述两种示例的结果是一样的，但实现的机制不同。

4.3.1 记录判别式

到目前为止的例题中，我们所遇到的记录其成分的长度都是固定的。这对于程序员并不方便，因为定义一个在各种情况下工作的记录之前，他必须知道数组或串的最大尺寸。实际上需要的是将参数传送给记录类型声明的一种手段，然后在声明确立之时，使诸成分按要求的尺寸精确地生成。*Ada* 中这种机制是判别式_部分。要注意，它是类型声明的语法特征，因而不出现在记录类型定义的语法中。判别式的语法是：

```
判别式_部分::=
    (判别式_声明{ ,判别式_声明})
判别式_声明::=
    标识符_表:子类型_指示[:=表达式]
```

由于判别式是用来定义数组或串的长度的，所以它们必须是

某种离散类型。记录中用判别式的一些例子是：

```
type family(size:integer)is
    record
        members:array(1..size)of person;
    end record;
type library(shelves,books_per_shelf:integer)is
    record
        volumes:array(1..shelves,
                1..books_per_shelf)of book;
    end record;
```

当我们给整个记录对象赋值时，不同于对单个记录成分赋值，我们要将记录值赋给记录对象。对于固定尺寸的记录（没有判别式），记录值是由要赋给记录的各个成分的值所组成。若记录有判别式部分，则其尺寸（甚至成分的个数）是可变的。因而，用于带判别式部分的记录的记录值要求包括各个判别式的值，为了说明：

```
declare
    type heading(length:integer range 1..80
                                    :=80)is
        record
            text:string(1..length);
        end record;
    title:heading;--初始长度是80
begin
    ...
    title:=(length=>5,text=> "Title" );
            --聚集赋值，注意判别式的值来自前者
    ...
end;
```

本例还可以说明，我们如何用和给变量赋值相同的方法来给记录的判别式置缺省初值。要记住的是，若要对一个判别式置缺省值，则对所有的判别式都要做。

4.3.2 判别式约束

和我们在数组域上置放一个约束大体相似的方式，我们也可以约束判别式的值域。例如，假定我们希望定义一个记录来暂时存放信的内容。设一般信只有 3 页，我们就可用它来作缺省值，如下示：

```
type letter(no_pages:integer range
                    1..max_pages:=3); is
    record
        content:array(1..no_pages)of page;
    end record;
my_letter:letter;--长度缺省值为3页
```

我们可以用给记录赋值的办法来改变对页数的约束，但这样做比较麻烦。我们可以在对象声明中置约束：

```
his _letter :letter;          --还是3页长
your_letter :letter(15);      --现在是15页了
their_letter:letter(no_pages=>10);
                              --约束的变换形式
```

注意，若在类型声明中无判别式约束的缺省值，则我们必须在每一对象声明中指定这个约束。如果有缺省值，则它们可被对象声明中新指定的约束值所复盖。判别式约束的语法是：

判别式_约束::=

(判别式_规格说明{ ,判别式_规格说明 })

判别式_规格说明::=

[判别式_名{ |判别式_名 }=>]表达式

如果一个对象，如*my_letter*，是来自有判别式的类型，则有一属性询问函数：

my_letter'constrained

它返回一个布尔值。该值告诉我们在对象声明中是否用了判别式约束。因而，在上述诸例中，*his_letter'constrained* 将为假，而 *your_letter'constrained* 和 *their_letter'constrained* 将为真。

4.3.3 记录变体

讨论判别式时，我们已学会如何变换记录成分的尺寸。如果你再往前查看一下记录类型的语法定义，就会注意到有一个可选的变体部分。这是 *Ada* 记录中最有用的特征之一，因为它使程序员有可能去改变组成记录的成分，而不仅仅只改变某个成分的尺寸。其语法是：

变体_部分::=

case判别式_名 **is**

{ **when**选择{ |选择 }=>成分_表 }

end case;

选择::=简单_表达式|离散_范围|**null**

注意下面的例子，看我们如何使用变体记录：

type *vehicle* **is**(*bicycle*,*car*,*truck*);

```
...
type transport(vehicle_style:vehicle)is
    record
            owner   :string(1..80);
        description:string(1..80);
        case vehicle_style is
            when car   =>licence:string(1..80);
            when truck=>licence:string(1..80);
                        trailer:boolean;
            when others=>null;
        end case;
    end record;
```

这个记录类型将有不同个数的成分，这些成分取决于参数 *vehicle_style*。我们可用以上类型声明生成并初始化以下三个记录对象：

```
declare
    my_truck   :transport(truck);
    my_car     :transport(car);
    my_bicycle:transport(bicycle);
begin
    ...
    my_truck   :=("John Smith","Truck",
                  "1ALC261",false);
    my_car     :=( owner   =>"John Smith",
                  description =>"Car",
                  licence     =>"MA3785");
    my_bicycle:=("John Smith","Bicycle");
    ...
end;
```

注意，如果我们愿意，则可由这个类型声明定义一些子类型：

```
subtype bike is transport(bicycle);
subtype auto is transport(car.);
subtype camion is transport(truck);
```

它们随后就可用于对象声明而无需再指定变体部分。这种对象都自动地继承由父类型'transport'传给子类型的属性。

4.4 访问类型和分配算符

所有以上处理的对象声明，都隐含了一个假设：无论何时确立对象声明，都要分配给该类型对象一个适宜的内存空间，同时声明中的标识符就与内存的这部分相结合。用这种方式生成的对象从任何地方（只要是从程序的另一部位对其可见）都是可以访问的，但还要取决于程序的行文结构。这种程序正文的结构是非动态的，不以任何方式依赖于结果代码的执行。因而，我们讨论过的对象也不能是动态的，它们的'存在'仅取决于程序正文的静态性质。

有这样的情况，程序员事先并不知道如何决定标识符与对象之间的对应关系，例如，他要在几个缓冲区中的一个取接值。这时实际上要求有一种给对象定义名字的机制，让实际的对象与名字结合的问题直到程序运行之前悬而不定。*Ada* 中这些功能由访问类型和分配算符来执行。

和整型对象能从其整值范围取任何值一样，访问对象也能从访问值的集合中取值。当分配算符执行时，它生成一个所要求类型的对象，给它分配存贮，并返回一个访问值。这

就可以给访问对象赋值。因而，为动态建立的对象和它们的引用之间建立了联系。

访问类型和分配算符的语法定义是：

访问_类型_定义::＝**access**子类型_指示
不完全_类型_声明::＝**type**标识符[判别式_部分]
分配_算符::＝**new**类型_标记[(表达式)]
　　|**new**类型_标记　聚集
　　|**new**类型_标记　判别式_约束
　　|**new**类型_标记　序标_约束

让我们跟着下例中事件的顺序，看看它们是怎样工作的：

```
declare
    type pool_space is array(1..5)of integer;
        --
        --首先定义我们要动态建立的
        --对象的类型
    type pool_space_pointer is access pool_space;
        --
        --这就建立了一个访问类型pool_space_
        --pointer
        --它现在可用来为pool_space类型对象
        --建立访问对象
    spare:pool_space_pointer;
        --
        --我们现在有了一个访问对象，其缺
        --省初值为空，即其意为无值。它现
```

```
        --在可用来访问pool_space的对象
    pool_element : pool_space_pointer :: = new
                    pool_space
        --执行分配算符'new  pool_space',
        --生成一个pool_space对象并为该对
        --象返回一个访问值，随后就赋给了
        --访问对象pool_element
        --
```

我们现在能为一个特定的类型生成访问对象，我们能用分配算符去生成这些对象，并能给一个访问对象赋一个访问值。我们还必须确定如何得到访问对象的值。在这样做之前，必须了解访问对象所具有的值和它所引用的值的区别。对于非访问对象，当程序员在程序中写一个标识符时，编译将确保他引用的是指明对象所期望的值。例如，你若在程序中写“$x:=y$”，就被解释为“用y的当前值来替换由标识符x表示的值”。这是直接见效的。因为x和y都直接引用了由x和y的声明所生成的实际数据对象。

对于访问类型，事情就要复杂一点。如果x和y被声明为访问对象，则它们的值并不直接和数据对象相联系。对于访问对象，标识符是用来取得它们所表示的访问值，进而再用它去得到我们用于运算的数据。因而，当“$x:=y$”时，显然还是将y的内容替换x的内容，但结果与前不同。如果x和y是访问对象，各自当前都访问一个数据对象，“$x:=y$”语句执行以后，得到的结果是x和y两者都访问“y”所访问的数据对象。谁也不去访问“x”原先访问的对象，它现在已经消失。

为了取得我们要求的结果，亦即将“y”访问的数据内容转而为由“x”访问的数据对象，*Ada* 有成分选择符 **all**（看第4.3节）。下面的例子概括了 **all** 使用的要点，并说明了访问值和它们所表示的数据的差别：

```
declare
    type get_at_integers is access integer;
            --访问类型
    x:get_at_integers:=new integer;--访问变量
    y:get_at_integers:=new integer;--访 问变量
    a:integer;
begin
    ...
    a:=1;           --给普通变量赋值
    x.all:=5;       --给访问变量赋值
    y.all:=a;       --同上
    x:=y;           --现在x引用y引用的对象
    y.all:=-1;
    a:=x.all;       --a现在是-1;
```

初读起来，上述几段文字似乎有些笨拙，可为此辨解的理由是：有意识地避在 *Pascal* 和 *PL/1* 语言中都用过的“指针”一词。指针带有它们实现的细节（如地址范围等等），而这些细节又不属于访问变量的讨论范围。*Ada* 语言并不指明访问机制是如何实现的，用术语“指针”来解释反倒会引入歧途。

访问类型是本章介绍的最后一个类型。私有类型放在第6章，当我们讨论程序包的时候再介绍。

第五章　子程序

本章专门研究过程和函数，*Ada*中叫子程序，子程序的概念和几乎所有当前应用的现代语言，如*Fortran*，*Algol*，*Pascal*，*Basic*等完全一样。

5.1　过程和函数

过程和函数可以看作是一些方法，即将程序代码的某个特定部分封装起来，使之在运行时刻从程序的许多地方都能用到它，而无需多次复制已有的代码。这可直接和传统的宏代码的概念进行对比，每当宏调用（在编译时刻）被扩展时，产生一个宏正文的付本去代替宏调用。尽管*Ada*语言并不支持宏，但程序员有要求在每次调用之时展开特定子程序体的选择权，可用以下设施：

pragma *inline*（子程序_名{，子程序_名}）；

为了得到子程序通常的好处，过程和函数在每次使用，或“调用”时，可有选择地得到传递来的参数，告诉它要处理什么数据，以及在什么地方送出结果。

在我们使用子程序或函数之前，我们必须知道调用的规则，即参数的个数，它们的类型，是过程还是函数。定义一个过程的例子是：

procedure *double_an_integer*(*i*:**in out** *integer*) **is**
begin

```
    i := i + i;
end double_an_integer;
```

它声明了一个叫做'*double_an_integer*'的过程，要求一个整类型参数，该参数值可在过程中修改。它还定义了此过程调用时所执行的运算。调用上述过程的一个例子如下：

```
count := 3;                          --假设 count 是 '整型'
double_an_integer(count);--  '调用' 本过程
put(count);                          --此处count = 6
```

过程和函数在几个方面有差别。最主要的是函数返回一个值到调用它的地方，而过程则不是；函数还可以作为语句的一部分来调用，反之，过程调用要一整条语句；最后，过程可以改变它的参数（只要程序员允许它改变），而函数只能用返回值影响调用它的程序。函数和过程满足类似的需求，其差别在于它们对调用者环境影响的作用域。

要注意，上述例子不一定要求使用过程。我们用函数代替过程也能履行同样的计算功能（从程序风格的观点来看这样更好些）。如下例：

```
function double_an_integer(x:integer)return
            integer is
begin
    return x + x;
end double_an_integer;
```

调用是这样的：

```
count := 3;                  --假定count是 '整型'
                             --
count := double_an_integer(count);
```

```
put(count);                    --此处count = 6
```

特别要注意的是过两种调用结果的差别。在过程调用语句中，

```
double_an_integer(count);
```

在调用的时候，'count' 的值于计算执行时在过程“内部”直接就修改了。与此对照的语句是：

```
count := double_an_integer(count);
```

此处函数自己并未修改 'count' 的值，但函数返回的值赋给变量 'count' 之后，把它修改了。

这个区别是非常重要的。编译程序把函数的参数限制为in模式就保证了函数只能用返回值去影响调用它的环境。

5.2 定义子程序—子程序体

当程序员定义一个子程序的时候，他必须向编译指明两件事：用于实施实际计算的代码和看起来和调用程序一样的子程序定义。'子程序_体' 可同时指明这两方面的信息片，它的语法是：

```
子程序_体 ::=
    子程序_规格说明 is
        声明_部分
    begin
        語句_序列
    [exception
```

```
        {异常_处理段}]
    end[指明符];
子程序_规格说明::=
    procedure标识符[形参_部分]
    |function指明符[形参_部分]return子类型_指示
形参_部分::=(参数_声明{;参数_声明})
参数_声明::=
    标识符_表:模式  子类型_指示[:=表达式]
模式::=[in]|out|in out
指明符::=标识符|运算符_符号
```

子程序_规格说明 从调用程序的观点是为编译 定义了过程或函数的界面，而从子程序的观点是定义了调用它的界面。于是，它就是子程序和系统的其余部分之间界面的定义。子程序体的其余部分指明要实施的计算机制，以及为异常处理而设的任何特殊要求。这些在语法定义 中都是括在 **begin……end**之内的部分。让我们看一下子程 序参数的 定义，以此作为研究的开始。

5.2.1. 形式参数

如果你注意看子程序体的语法定义，你就会发现子程序规格说明中“形参_部分”的功能是定义这个子程序的参数。在子程序体内或子程序声明内定义的参数称为‘形式_参数’。而那些在调用子程序时给出的参数则叫做‘实在_参数’。形参_部分指定参数的名字，靠这些名字形式参数才会在子程序体内成为已知；形参_部分还指定它们的类型 和约 束；还指出某个特定参数是只读(in)参数(如无显式的模式 说明，缺省模

式是in模式）还是只写(out)参数，或者是读_写(in out)参数；以及任何缺省初值。下面是几个子程序规格说明的例子，可以看到形式参数的各种模式：

```
procedure bubble_sort(matrix:in out array)
   --
   --分类一个数组
procedure date(day:out day_of_week;
               month:out month_of_year ;
               year:out years_ad      )
   --
   --返回今天的日期
procedure enqueue(     item     :in     queue_item;
                  this_queue    :in out queue;
                  queue_full    :   out boolean
   --
   --置'queue_item'对象到队列中
procedure dequeue(     item     :   out queue_item ;
                  this_queue    :in out queue;
                  queue_empty   :   out boolean      )
   --
   --从队列中移去一项
function queue_empty(this_queue:queue)return
         boolean
   --
   --测试一空队列
   --注意，由于函数只能有in模式的参数，
   --故不需指定模式
function random_int(seed:in integer:=31415)
         return integer
   --
```

```
        --随机数函数
        --注意，不要求写模式说明，但写了无
        --害处，如果有助于清晰和可读性，它
        --还有好处
```

我们试写一简单的过程或函数求整型数组中的最大值：

```
procedure max_int(list:in vector;
                  max:out integer)is
    --
    --定义一个叫做'max_int'的过程，带有
    --两个形式参数，'list'是vector'类型，
    --只读，'max'是'integer'类型，只写
    --
begin
    --
    --此处开始写进行计算 的 代码
    --前面的'begin'标志着声明部
    --分的结束
    --
    max:=list(list'first);
    for i in list'range loop
        if list'(i)>max
        then max:=list(i);
        end if;
    end for;
    --
    --此处程序代码与过程体均结束
    --
end max_int;
```

本过程的调用程序看来是这样：

```
declare
    type vector is array(1..10)of integer;
    my_vec    :vector;
    maximum :integer;
begin
    …
    my_vec:=(1,2,3,4,5,6,7,8,9,0);
    …
    max_int(my_vec,maximum);
    put(maximum); -- "9"
    …
end;
```

5.3 实在参数和子程序调用

上面所述的情况，都隐含了一个假设，即调用程序中的实在参数和子程序定义中的形式参数相结合的机制是两个参数表一对一地对应，一个表的第一个参数和另一个表的第一个参数匹配，第二个和第二个匹配等等。这在*Ada*中自然是合法的，同时*Ada*语言还允许程序员在调用时使用显式的指名参数结合，即用形参表中的名字指明实在参数。于是，当给出以下过程定义时：

```
procedure delete_file
        (file_name      :file;
        volume_name     :volume;
          password      :key;
          owner         :user      )is…
```

以下调用都是等价的：

```
delete_file(my_file,my_volume,my_key,me);
```

```
delete_file(file_name  =>my_file,
            volume_name =>my_volume,
               password =>my_key,
               owner    =>me       );
delete_file(my_file,my_volume,owner=>me,
                              password=>my_key);
```

上述所有信息，在过程或函数调用的语法定义中都有规定：

过程_调用::=过程_名[实在_参数_部分]
函数_调用::=
　　函数_名　实在_参数_部分|函数_名（）
实在_参数_部分::=（参数_結合{，参数_結合}）
参数_結合::=[形式_参数=>]实在_参数
形式_参数::=标识符
实在_参数::=表达式

注意到参数_結合中形式_参数的语法和使用指名的记录聚集成分的语法极其相似是很有意思的。和记录聚集一样，子程序参数可纯粹地按位置结合，纯粹地按指名结合，以及这两者的混合使用。如果将它们混用，则所有按位置结合的参数必须出现在指名参数之前，一旦使用了头一个指名参数则要求所有后续序列都按指名结合。

5.4 参数模式和缺省初值

如果你回头看前述'*random_int*'函数的例题，就会看到它的参数带有初值。这是*Ada*的一个有用特征，程序员就可以对in模式的任何形参指定缺省值。一旦提供了缺省值，

调用时相应的实在参数就可以省略。此时缺省值就用在该参数的位置上。如果程序员在调用时给出了实在参数值，则该值将复盖缺省值。作为一个缺省实际参数的例子，下例使用了缺省设施：

```
procedure standard_out put(  data:in line;
                             unit:in device:=tty;
                           status :out boolean   )is
begin
    ...
end;
standard_out put(line_buffer,
          status=>fail_flag);
    --
    --使用了缺省，unit=tty
    --
    ...
standard_out put(line_buffer,printer,fail_flag);
    --
    --复盖了缺省值，unit=printer
    --
    ...
standard_out put(data   =>line_buffer,
                 unit   =>mag_tape,
                 status =>fail_flag   );
    --
    --复盖了缺省值，unit=mag_tape
    --
```

关于缺省参数有两个要点要注意。第一，如果在过程或函数

调用中省略了一个实在参数，允许缺省值占据它的位置，则所有后续的实在参数都必须是指名结合（否则编译要区分哪些是合法的参数省略，哪些是程序设计的疏漏）。第二，只有in模式参数才允许使用缺省值。如果试图在out或in out模式参数置缺省值，则编译将指出错误。

当我们研究参数模式这个题目时，研究一下不同模式对传递参数的影响是有价值的。子程序调用的语法允许将表达式传给过程或函数。如果表达式和in参数结合，则其值在调用之先求出。如果形式参数被指定为out或in out模式，则表达式只限变量或变量的类型转换。于是，我们就可象下面那样写：

```
procedure x(y:in out float)is…
declare
    i:integer;
begin
    …
    x(float(i));
    …
end;
```

其中整变量'i'的值先转换成'浮点'类型对象，然后浮点化的对象作为一个参数传递到被调用的过程。从子程序返回后，在复制回实际参数之前，又执行从'浮点'到'整型'的反向类型转换。

关于参数的最后几句话：调用程序中的实在参数和子程序规格说明中的形式参数之间，按常规要作类型兼容的检查。同样，还有参数约束的检查。以确保任何传到子程序的

实在参数都遵循对形式参数的要求。同时，任何返回的参数或返回的值也应符合调用程序强行规定的约束。

5.5 子程序声明

按照*Ada*关于语法实体在使用之前都要声明的概念，*Ada*给程序员提供了声明子程序定义但不必同时给出子程序体的能力。这有几方面的好处，如增加了可读性；有利于自动文挡抽取系统*；特别是，使其他用户也可以方便地应用程序包内的过程或函数。它允许在一个地方声明成组的相关子程序，以减少疏忽错误的机会，使之很容易找到某个特定的声明。子程序声明的语法是：

子程序_声明 :: = 子程序_规格说明
　　|类属_子程序_声明|类属_子程序_設置
子程序_规格说明 :: =
　　procedure　标识符[形参_部分]
　　|**function**　指明符[形参_部分]**return**子类型_指示
形参_部分 :: =(参数_声明{ ,参数_声明 })
参数_声明 :: = 标识符_表:模式子类型_指示[:=表达式]
模式 :: =[**in**]|**out**|**in out**
指明符 :: = 标识符|运算符_符号
运算符_符号 :: = 字符_串

我们以一个处理矩阵计算的程序包作为例子。我们可以在程序包的可见部分中用子程序声明去声明所有的函数和过程，

* 自动文挡抽取系统*Automatic Document Extraction System*是*Ada*语言支持环境的子系统，其功能是根据编译需要抽取相关的过程、模块、文件重新编译。——译者注

而将相应的子程序体留在程序包的掩蔽部分(*hidden part*)。可见部分中的声明使程序可从程序包外部决定对子程序调用的规则，但防止它们对实现任何具体特征作出何任决定。于是，我们可有一系列声明：

```
package matrix_calculations is
    procedure mat_add(x,y:in matrix;
                      z:out  matrix);
    procedure mad_sub(a,b:in matrix;
                      c:out  matrix);
end;
package body matrix_calculations is
    ...
begin
    ...
end;
```

此例声明一个作矩阵加减法的程序包。调用程序只知道此程序包的两个过程的规格说明。过程的实际实现掩蔽在程序包体中。要使用这个程序包，要求调用程序有如下的使用(*use*)子句：

```
declare
    use matrix_calculations;
    a,b,c:matrix;
begin
    ...
    mat_add(a,b,c);
    ...
end;
```

使用子句使程序包的可见部分中的实体成为可访问的。在下

章程序包中我们还要进一步讨论。

最后，要记住，如果你使用了子程序声明，并随后又定义了子程序体，则有一显见的要求。即子程序声明和子程序规格说明必须协调一致。

5.6 重载子程序

枚举型字面量按其上下文可能有多个解释。与此大致相仿，子程序也能重载。当程序员有几个不同的对象，而其中每一对象都执行同一逻辑功能时，这将是一个有用的特性。作为一个例子，函数'*next_item*'可以返回一个队列_项(*queue_item*)，也可以返回表中的下一项(*list_item*)或者是终端屏幕上光标位置的下一项(*screen_item*)。我们可以设想出一系列的声明：

```
function next_item(q:queue )return queue_item is
                                    ...
function next_item(l:list   )return  list_item is
                                    ...
function next_item(c:cursor)return screen_item is
                                    ...
```

同时要确定在调用时用哪个函数求解。这个问题的解决有赖于在调用时有某些可分辨的特性，这些特性能唯一确定一个函数的定义。用于决断的因素是调用时参数的个数、顺序和类型。在函数的情况下还有返回值的类型。如果调用的不是一个或不止一个子程序定义，也就是说它含混不清，因而是非法的。

在外层和内层的声明部分中使用同一子程序名并不一定意味着外层子程序声明被'掩蔽'于内层声明。掩蔽仅发生

在两个声明引入的子程序参数的个数一样，参数的名字也相同，类型相同，且按同样的顺序，以及同样的缺省值。在函数的情况下，结果的类型也必须相同。如果所有这些条件都满足了，就可以说这两个子程序是‘等价的’，那么外层的那个就被掩蔽了。注意，参数的任何约束值、初值以及参数模式在判定等价性时都不起作用。

```
declare
    procedure a(x:integer);
    function b(y:co_ordinate)return real;
    procedure c(z:in boolean);
begin
    ...
declare
    procedure a(x:integer);
    --
    --掩蔽了外层的'a'，因为名字
    --和类型都没有差别
    --
    function b(y:co_ordinate)return co_ordinate;
    --
    --没有掩蔽外层的'b'，因为结果参数
    --的类型有差别
    --
    procedure c(z:in out boolean);
    --
    --掩蔽了外层的'c'，因为在判定掩
    --蔽时并不考虑参数的模式
    --
    end;
end;
```

如果问题有了多余的掩蔽，通常是用限定词更加明显地标定参数或子程序的名字来解决。

5.7 重载运算符

*Ada*强有力的特征之一是程序员可以为他自己的类型定义运算符，这和*Algol_68*语言是相同的。这是靠定义一个起运算符作用的函数来做到的。*Ada*允许你定义一个函数，其名字和预定义语言运算符的指明符相同（运算符“/=”是例外，如下注）。

```
逻辑_运算符 :: = and | or | not
关系_运算符 :: = < | < = | = | / = | > = | >
加法类_运算符 :: = + | - | &
单目_运算符 :: = + | - | not
乘法类_运算符 :: = * | / | mod | rem
乘幂_运算符 :: = * *
```

对于一元或‘单目’运算符，函数只能有一个参数。于是，我们对颜色定义如下：

```
function “+” (c1,c2:colour)return colour is…
function “-” (c1,c2:colour)return colour is…
function “=” (c1,c2:colour)return boolean is…
```

按下述方式使用它们：

```
x:= “+” (red,blue)        --紫色，'red'对应于形
                          --式参数'c1'，'blue'对
                          --应于c2'
                          --
```

```
y:= "-" (x,red);          --兰色
if "/=" (y,blue) then…
```

注意，运算符“=”必须返回一个布尔型结果值，并且此运算符的定义也就定下了运算符“/=”的定义，因而“/=”不能重载。该运算符还是只能取受限私有类型参数的专用运算符。在程序包一章中我们还将更多地谈到这些。

第六章　程序包

本章专门研究*Ada*语言最有力的特征之一——程序包，以及在可见性规则之下它们的作用。

6.1　概　述

计算机科学中把程序的一部分封装起来的概念并不是新概念。早期语言中最明显的例子是*Fortran*的子程序和*Algol*或*Pascal*的过程。这种结构将要履行的逻辑功能（比如，求一个整数的平方根）和它的实现细节（实际算法的使用以及它的精度等等）截然分开。通过使用子程序库或过程库也能得到这种分离（统计库和数学库都是很好的例子）。遗憾的是没有一种方法提出了为程序员掩蔽不必要的信息这个问题，也不能有效地防止程序员去利用实现细节而牺牲可移植性，可能还会牺牲可靠性。作为一个例子，我们考察一个用*Fortran*语言写的通用终端驱动器的子程序。我们令其中一个参数*ITERM*的值去标识特定终端类型，0＝*TTY*，1＝*CRT*，2＝图形*CRT*等等。要使用这个程序就必须使用这些信息，所以我们无法向程序员隐蔽终端的特性是由整数来表示的这一事实。

*Ada*中，原始的封装思想推广为‘程序包’的概念。其特点是把数据、类型、算法以及这三者的组合给封装起来。语言提供了适宜的机制，以确保程序仅能访问由程序员显式

地指明的程序包的某些部分，防止访问其余部分。这样，程序员就可以控制他的程序包，规定其中哪些信息片是程序包的用户可以使用的。程序包的概念在提供与实现无关的代码的尝试上迈出了重要的一步，因而也是朝可移植性的目标方向迈出重要的一步。

Ada 的程序包由两部分组成：程序包规格说明（它定义了外部世界可以访问的实体）和程序包体（它包括了为实现预期功能的代码、数据、类型等，但不为外部世界所知）。如果所有的信息都能在规格说明中给出（参看 6.2.1 节），则程序包可以不要包体；也不要求在规格说明的同一位置上给出程序包体（这两部分甚至可作为分开的单元编译）。

6.2 定义程序包

和其他 *Ada* 对象一样，在使用程序包之前必须用程序包_声明来定义它，其语法如下：

```
程序包_声明::=程序包_规格说明
    |类属_程序包_声明
    |类属_程序包_被置
程序包_规格说明::=
    package  标识符 is
        {声明_项}
     [private
        {声明_项}
        {表示_规格说明}]
    end [标识符] ;
```

类似地，程序包体也要定义，其语法是：

```
程序包_体::=
    package body  标识符 is
        声明_部分
    [begin
        語句_序列
    [exception
        { 异常_处理段 } ] ]
     end  [标识符] ;
```

注意，程序包体中的程序包标识符必须与相应的程序包规格说明中的标识符相同。同样，在 end 后的可选标识符也必须和规格说明及包体中给出的相匹配。

6.2.1 程序包规格说明

如上所述，程序包中能为外界所识别的部分只有那些在程序包规格说明中给出的内容。在*Ada*中，它称为程序包的‘可见’部分。程序包规格说明的可见部分中声明的任何对象都可以被包括在规格说明之内的程序单元访问。程序包规格说明的一些可能的例子是：

```
package complex_arithmetic is
    type complex is
        record
            real_part :float:=0.0;
            imag_part:float:=0.0;
        endrecord;
    function "+" (a,b:complex)return complex;
```

```
        function "-" (a,b:complex) return complex;
        function "*" (a,b:complex) return complex;
        function "/" (a,b:complex) return complex;
        function sqr (a,b:complex) return complex;
    end complex_arithmetic;

    package keyboard is
        type key is private;
        function get_key return key;
        function convert_key(a:key)return key;
    private
        type key is integer range 0..255;
    end keyboard;
```

第一个例子是*complex_arithmetic*程序包，它由类型声明和重载运算符 +、-、*、/ 的规格说明所组成，类型声明用以生成‘复型’对象，重载运算符用以运算‘复型’对象。程序包提供给用户实施复型对象计算所必需的信息（本例稍粗糙，因为它缺乏作标量运算的运算符）。

第二个例子与此类似，它也包括了类型声明和许多函数声明。然而它和*complex_arithmetic*程序包又有差别，其差别就是通过**private**子句(看6.3节)对类型'*key*'所加的限制。

程序包的完整概念是：在使用它的时候可以不考虑隐藏在包体内的实现细节。在我们研究程序包体定义的细节之前，先研究下例看来是适宜的。该例演示如何从用户程序去访问*complex_arithmetic*程序包。我们先给出例子然后讨论它的特征。

```
declare
    x,y,z:complex_arithmetic.complex;
```

```
begin
    ...
    z:=complex_arithmetic.sqr(x,y);
    ...
end;
```

我们假定从本程序的外层块可以见到程序包的规格说明，所以内层块也可见到它。本例中，用类型声明'complex'生成了三个复型变量x、y、z。随后它们可用在程序包规格说明中声明过的（重载）运算符或函数进行操作。要记住，没有理由要程序员知道这些类型和函数实现的细节。例如，算术运算符"+"，可以由一个软件的子程序来实现，也可以由硬件的一系列指令来实现。甚至可由某个向量处理硬件的一条指令来实现。

6.2.2 使用子句

到目前为止，有关在程序包规格说明中定义的对象如何由用户程序访问的规则，我们几乎没有谈到，甚至根本没有触及。为了使用在程序包中命名的对象，我们只好用选择成分（和记录成分选择是同样的办法），写出象这样的语句：

```
x:complex_arithmetic.complex;
```

它声明对象'x'是'complex'类型，而类型的定义可在程序包'complex_arithmetic'中找到。事实上，任何在程序包规格说明中声明的项都要用在对象名前加上程序包名和圆点的办法来选定。这很快就会成为一个麻烦的事，而且是不必要的负担。所以Ada提供了使用子句，其语法是：

使用_子句::=use程序包_名{，程序包_名}；

作为一种便利设施，它使程序包规格说明中所声明的项成为直接可见的，就好象程序包规格说明中给出的定义都写在使用的地方一样。我们现在可利用这个新特征来改写前面的例子：

```
declare
    use complex_arithmetic;
    x:y,z:complex;
begin
    ...
    z:=sqr(x,y);
    ...
end
```

有了使用子句之后，不需要程序员对它希望访问的程序包中每个对象冠以程序包名。当程序员不愿意写或键入长名时，首先受害的将是程序的可读性以及文挡管理，这时这个特征就极其有用了。

通过举例，试比较以下两个程序片段，两者实施同一功能，第一个繁琐：

```
declare
    use complex_arithmetic;
    x,y,z:complex;
begin
    ...
    z:=spr(x,y);
    complex_arithmetic_input_output. put(z);
    ...
end;
```

第二个比较明智：

```
declare
    use complex_arithmetic,
        complex_arithmetic_input_output;
    x,y,z:complex;
begin
    ...
    z:=sqr(x,y);
    put(z);
    ...
end;
```

使用这个特征要小心一些，因为利用使用子句可能会导致混淆，而这只有用成分选择才能解决。这很容易用两个程序包来说明，二者都包含了同一标识符的声明：

```
package person is
    ...
    age:integer;
end person;
package antique is
    ...
    age: integer;
end antique;
...
declare
    use person,antique;
    --现在'age'有两种版本供程序员使用
    --
    temp:integer;
```

```
begin
    ...
    temp:=age;              --不合法,要求进一步定义
    temp:=person.age;       --好的，没有什么含混
    antique.age:=temp;      --好的，不含混
    ...
end;
```

6.2.3 程序包体

我们现在讨论程序包体。它是程序包的一部分，它包含了程序包规格说明中定义的实体实现的细节。公共的程序包名将规格说明和包体联系起来。作为一个例子，程序包*complex-arithmetic*的包体可以是这样：

```
package body complex_arithmetic is
    function "+" (a,b:complex) return complex is
        c:complex;
    begin
        c.real_part:=a.real_part+b.real_part;
        c.imag_part:=a.imag_part
                        +b.imag_part;
        return c;
    end "+";
    function "-" (a,b:complex)return complex is
        ... end;
    function "*" (a,b:complex)return complex is
        ... end;
    function "/" (a,b:complex)return complex is
        ... end;
    function sqr (a;b:complex)return complex is
```

```
        ··· end
end complex_arithmetic;
```

用户程序只能经过程序包规格说明中的可见部分才能访问定义在程序包体中的对象。这就看出程序包的概念提供了所要求的机制以阻止程序员去了解程序包内部工作的任何细节。为了说明，我们考察带有以下程序包规格说明的程序包：

```
package trig is
    ···
    function sine(theta:radians)return float;
    ···
end trig;
```

对本程序包用户可用的有关'*sine*'子程序的信息只有调用序列和每个参数的类型。程序员只知道*sine*返回参数的值域和精度（可用类型的属性询问函数求出）。他不能确定求*sine*值的算法，也不能确定，比如说，子程序使用了中间存贮没有？从程序员的角度看，只要他能取得*sine*值，他不需要知道到底*sine*值是由级数逼近算出，还是靠查表找出，甚至靠变魔术变出来的。

就象在引论中提到的那样，并非每个程序包规格说明都要有一个包体。如果一个规格说明中仅包含类型或数据以及在规格说明中能得到完整定义的对象，则并不需要提供一个相应的程序包体。例如，用这种办法可以定义全局常量和全局性数据区，实现象*Fortran*中*COMMON*那样的概念。

```
package common is
        real_common :array(1..100) of float;
    integer_common  :array(1..100) of integer;
```

```
end common;
package terminals is
    type term is private;
private
    type term is (tty,crt,graphic_crt,printer);
    for term use(      tty=>1,
                       crt=>2,
              graphic_crt=>3,
                printer   =>4);
end terminals;
package document_printer_defaults is
    lines_per_page   :constant integer:=66;
    columns_per_line :constant integer:=80;
    paper_width      :constant float:=8.5;   --吋
    paper_length     :constant float:=11.0;  --吋
    header_margin    :constant integer:=3;   --行
    footer_margin    :constant integer:=8;   --行
end document_printer_defaults;
```

以上每个程序包都是独立的，不要求对应的程序包体。

我们现在研究程序包初始化的概念。在程序包的语法定义中有执行语句序列的条款。但在我们仔细研究它们之前，程序包体在确立期间会发生什么事是必须知道的。当程序包体确立时，引入了程序包标识符，随后声明部分确立。此时引入并初始化每个变量，一旦这些发生，包体中的语句序列就被执行。若在声明部分不能实施某些初始化时，执行这些语句使之实施初始化，这种能力被证实是极其有用的。很容易设想到一个要求使用一个或多个队列的作业。最初将一个队列_项联到队列_空项上仅靠变量初始化的机制是难

于实行的，但这很容易用执行语句实现：

```
package body queue_manager is
    free_queue_size:constant:=100;
    free_item_pool:array(1..free_queue_size) of
                                    queue_item;
    procedure enqueue(x: in queue_item)is...end;
    procedure dequeue(x:out queue_item)is...end;
begin
    --写建立初始队列指针的代码
    --将项缓冲池插入到空队列中
end queue_manager;
```

在声明部分给变量赋值的组合，加上在“一次_通过”的基础上执行语句的能力，提供了一个强有力而又灵活的初始化机制。

6.3 私有类型和受限私有类型

在前节的*complex_arithmetic*例题中，我们声明了一个具有‘浮点’类型的实部和虚部的记录*'complex'*。程序员完全可以访问这个类型声明，因为程序包的规格说明中并没有明显的动作来阻止这种访问。这恰是规格说明可见部分的新问题。然而，有时我们需要声明一个能为外部世界使用的类型又要掩蔽该类型的实际实现。这可用 类型 **私有** 来完成，像前述键盘和终端程序包用过的那样。生成一个**私有**类型其结果是：除了类型名字、用以赋值的预定义函数、以及测试相等不相等的功能而外，所有属性均消失。举例来说，如果我们希望掩蔽类型“*complex*”的实现细节，我们就改写*complex_arithmetic*程序包的规格说明如下：

```
package complex_arithmetic is
    tyqe complex is private;
    function "+" (a,b:complex)return complex;
    function "-" (a,b:complex)return complex;
    function "*" (a,b:complex)return complex;
    function "/" (a,b:complex)return complex;
    function sqr (a,b:complex)return complex;
private
    type complex is
        record
            real_part :float:=0.0;
            imag_part:float:=0.0;
        end record;
end complex_arithmetic;
```

这样，就使程序包的使用者无法判定这种类型的结构。

现在我们来改虑关于文件管理的一个具体问题。我们希望用戶能访问某个有密码的文件。该密码与用戶自己的代码对上号就可访问，但我们并不希望他能判定密码本身的特性，否则他就可能破译它们，并得以访问其他用戶的文件。我们构造一个程序包，它提供一个'access_code'类型，由此类型用戶可生成一个对象(my_code)去保存他的密码。我们再加上三个函数。第一个是将用戶自己的密码返回给用戶，第二个是求出某个文件的密码，第三个是根据用戶提供的密码试着打开现存的文件。

```
package file_security is
    type access_code is private;
    type file_name is array(1..20)of character;
    function get_my_code return access_code;
```

```
    function get_file_code(x:file_name)return
                                        access_code;
    function open_file(x:file_name;
                       y:access_code)return boolean;
private
    type access_code is integer;
end file_security;
declare
    use file_security;
    my_code:access_code;
    my_file,any_file:flei_name;
begin
    ...
    my_code:=get_my_code();
    if open_file(my_file,my_code)--打开成功吗?
    then
        --是，就按正常处理
    else
        --否，按“开启失败”处理
    end if;
    ...
    if my_code=get_file_code(any_file)
        --为了写入，我能打开这个文件吗?
    then
        --可以，处理它罢
    else
        --不行，放弃这个雅兴
    end if;
end;
```

如果在程序片段的声明部分里已声明了所需的对象，用户就

可以调用返回他自己访问密码的函数。要注意，尽管此时他已得到了自己的密码，但他并不知道它是什么样的，也不知道它是如何表示（它可能是个整数，可能是串或是两个数字等）。却可以使用这个代码去调用 *open_file* 并用以测试 *get_file_code* 返回的值。

有这样的情况，除了把类型功能特意地提供给用户而外，要阻止对某个类型全部对象的访问。那就要用到**受限私有**类型，用它之后连私有类型的少数权利（如赋值、测试相等/不相等）也都取消了。使用**受限私有**类型有许多严格的限制，主要是由缺少赋值功能而引起的。首先，如果没有赋值，这种类型的变量在声明之时就不能使之初始化。同理，受限私有类型的子程序参数也不能有缺省值。最后，受限私有类型的常量也不能在定义这种类型的程序包之外去声明（如若不能测试他们，你又何以知道他们对不对呢？你能对它们怎么办？）。

受限私有类型使程序员得以最大限度地控制那些使用者可用的信息。作为一个例子，让我们重建保密文件的程序包：

```
package secure_files is
    type access _code is limited private;
    type file_name is array (1..20) of character;
    function get_my_code return access_code;
    function test_write_permission(x:file_name;
                                   y:access_code)
                                  return boolean;
    function open (x:file_name;
                   y:access_code)return boolean;
private
    type access_code is integer;
```

```
end secure_files;
declare
    use secure_files;
    my_code:access_code;
    any_file,my_file:file_name;
begin
    my_code:=get_my_code();
    ...
    if open(my_file,my_code)--‘开启’成功吗?
    then
        --是，按正常处理
    else
        --否，按“开启失败”处理
    end if;
    ...
    if test_write_permission(any_file,my_code)
        --为了写入，我能打开这个文件吗?
    then
        --可以，处理它罢
    else
        --不行，放弃这个雅兴
    end if
end;
```

总之，程序包给程序员以一个强有力的封装机制，而定义**私有**或**受限私有**类型则提供了从用戶程序对程序包结构内部的可见度的控制。

第七章　类属子程序和类属程序包

本章我们将研究类属程序单元（类属子程序和类属程序包）的性质及其用途。

7.1　类属概念

在软件工程中有许多需要程序员把同样的逻辑功能应用于多种不同数据对象上的事例。作为一个例子，我们来考虑插入队列/取出队元(*enqueue/dequeue*)的操作。纵观全局，被插入队列的对象的类型和插入队列的功能没有什么关系。我们可能正在处理一个即将运行的进程，一个要打印的文件，一个字符加工或者是将一个磁盘块回复到空表。在这些情况中没有一种情况的对象类型影响到我们对“插入队列”功能的理解。

基于上述论点，似乎有理由问一问，在设计各种“插入队列”的版本时，各操作之间是否有逻辑相似性？能不能找出一种办法来保存这些相似性？*Ada* 提供的类属子程序和类属程序包就是在源程序代码这一级上保存这些相似的手段。例如，类属子程序的概念允许程序员设计一个插入队列过程的样板，编译时刻从样板中导出几种不同的插入队列子程序的示例，每种要处理的对象各对应一个子程序。

照例，子程序样板由类属子程序声明给出，而那个样板

的特例（多半要具体到特定类型的对象）由类属子程序设置给出。显而易见，在类属设置的过程中必须有一种传递参数的手段，以便我们确定如何建立这个子程序特例，*Ada* 则提供了能进行这些参数传递的机制（尽管有某些限制）。

作为一种完全的补充，我们可将类属概念用于程序包。这就给予程序员一种手段，使他能将这个新技术用于更广义的抽象。我们可以想像一个程序员写了一个类属队列操作的程序包，它由若干队列处理类属子程序组成。而类属程序包可用来生成几个队列处理程序包，每个都具体到特定类型的对象。

最后要记住的一点是：类属子程序和类属程序包两者都不能直接执行，只有'设置'的特例可以执行。

7.2 类属子程序

我们首先考虑类属子程序。如上指出，类属子程序通过类属子程序声明引入，其语法是：

```
类属_子程序_声明::=
    类属_部分  子程序_规格說明;
类属_部分::=generic{类属_形式_参数}
类属_形式_参数::=参数_声明
        |type标识符[判别式_部分]is类属_类型_定义;
        |with子程序_规格說明 [is 名字];
        |with子程序_规格說明 is <>;
类属_类型_定义::=
    (<>) |range<> |delta<> |digits<>
    |数組_类型_定义
```

| 訪問_类型_定义
| 私有_类型_定义

用来说明类属子程序内在原理的标准例子是“*swap*”过程，它用于交换同类型的两个对象。我们能够设计这样一个类属子程序，它在类属设置之后，就可以建立能交换同类型两个变量内容的可执行过程。这个类属 子程序 声明 可以是这样的：

```
generic
    --定义类属参数，一个类型，一个子程序
    type object_type is private;
    procedure swap_objects
        (a,b:in out object_type);
```

带有如下类属体：

```
procedure swap_objects(a,b:in out object_type)is
    temp:object_type;
begin
    temp:=a;
        a:=b;
        b:=temp;
end swap_objects;
```

对于枚举类型'*day*'的类属设置是：

```
procedure swap_day_objects is new
        swap_objects(day);
```

以及这个新子程序使用的例子：

```
declare
    type day is(sun,mon,tue,wed,thu,fri,sat);
    x,y:day;
begin
    ...
    swap_day_objects(x,y);
    ...
end;
```

让我们顺序跟踪一下从类属声明到被设置过程的使用。当一个类属子程序（在编译处理期间）确立时，子程序名（即子程序标识符）就被引入，随后，确立‘类属部分’。

随着类属形式参数的确立，类属部分也就开始确立了。其方式类似于非类属形式参数的确立。当所有类属形式参数确立完之后，就引入子程序_规格 说明。在这里，常规 声明确立期间的操作和类属子程序声明确立期间的操作有一个重要的差别。在非类属情况下，一遇见子程序规格说明，该规格说明就被确立了。然而，类属子程序声明确立时，遇见了子程序规格说明它并不确立。而是把这个规格说明作为子程序的‘样板’记下来，直到所有的类属实在参数替换了类属形式参数，确立才在设置的示例上实际发生。

类属子程序体的确立为今后使用它引入了一个样板。和类属子程序规格说明一样，它的确立要延迟到类属设置的时候。

我们现在再看设置，即用样板建立一个可执行子程序的一个实例。类属子程序设置的语法是：

类属_子程序_設置::=

procedure 标识符 **is** 类属_設置

```
    |function 指明符 is 类属_設置
类属_設置::=
    new 名字 [(类属_結合{，类属_結合})]
类属_結合::=[形式_参数=>]类属_实参
类属_实参::=表达式
    |子程序_名|子类型_指示
```

类属实在参数和类属形式参数的结合跟其他参数替换的办法一样，可以按位置、按指名或用缺省值。但有一点不同，即类属参数结合的模式只能是 in 或 in out。如果参数模式为in则作常量看待，而in out 模式则作为一个对象的名字看待，该对象等价于在设置中给出的相应类属实在参数。同非类属形式参数一样，若无显式的模式则其结果是用缺省模式 in。

一旦实、形参数完成了结合（包括各缺省值的替换），设置就继续进行，建立样板的示例，在整个新子程序的规格说明和体中将实在参数替换形式参数。这些都做完之后，类属子程序产生的示例到那时候就按通常的办法确立。我们这时就有一个可用的子程序了，好像我们用手写的而不是'自动'生成的一样。

7.3 类属程序包

类属程序包在概念上和类属子程序非常相似。类属程序包的语法是：

```
类属_程序包_声明::=
    类属_部分  程序包_规格說明;
```

类属_部分的语法和在类属子程序声明的定义中给出的完全相同。类属程序包设置的语法是：

类属_程序包_設置::=

package 标識符 **is** 类属_設置

类属_設置的语法和7.2节中给出的完全相同。

现在让我们跟读一个定义、设置、使用类属程序包的例子，该包中有几个类属子程序。本例中，我们将熟悉为实现一个循环队列方案去写类属程序包的问题。队的长度和列队对象的多少由参数给定。这样的程序包其定义可以是：

```
generic
    queue_size:integer range 2..integer'last;
    type object is private;
package queue_handler is
    procedure enqueue(a: in object);
    procedure dequeue(a:out object);
end queue_handler;
package body queue_handler is
    subtype pointer_type is integer
                            range 1..queue_size+1;
        queue      :array(1..queue_size)of object;
     in_pointer    :pointer_type:=1;
    out_pointer    :pointer_type:=1;
procedure enqueue(a:in object)is
    begin
        if not queue_full(in_pointer,
                          out_pointer)then
            queue(in_pointer):=a;
            in_pointer:=increment(in_pointer);
        else
            put("queue_handler,enqueue:" &
                "You are now out of room");
```

```
            end if;
        end enqueue;
    procedure dequeue(a:out object) is
        begin
            if not queue_empty(in_pointer,
                                out_pointer)then
                a:=queue(out_pointer);
                out_pointer:=increment
                                (out_pointer);
            else
                put("queue_handler,dequeue:" &
                    "The queue is empty");
            end if;
        end dequeue;
    function increment(p:in pointer_type)
            return pointer_type is
        begin
            if p=queue_size then
                return (1);
            else
                return (p+1);
            end if;
        end increment;
    function queue_full(inp,outp:in pointer_type)
            return boolean is
        begin
            return(increment(inp)=outp);
        end queue_full;
    function queue_empty(inp,outp:in pointer_type)
            return boolean is
        begin
```

```
            return(inp=outp);
        end queue_empty;
    end queue_handler;
```

我们可以利用这个类属程序包的定义去建立两个程序包示例。其中一个将处理鍵盘输入，另一个处理磁盘传输的請求：

```
    declare
        type key_type is new character;
        package key_queue_handler is new queue_handler
            (queue_size=>10,
                object=>key_type);
        use key_queue_handler;
        new_key,old_key:key_type;
    begin
        ...
        enqueue(new_key);
        ...
        dequeue(old_key);
        ...
    enb;
declare
        type disc_transfer_request is
            (read,write,seek,verify,format);
        package disc_transfer_handler is new
            queue_handler
            (queue_size=>50,
                object=>disc_transfer_request);
        use disc_transfer_handler;
        next_request:disc_transfer_request;
```

```
begin
    ...
    find(next_request);
    enqueue(next_request);
    ...
end;
```

现在我们已经看到如何声明、设置、使用类属子程序和类属程序包的几个例子。我们再回过头来研究控制类属参数的规则和机制的细节。首先是类属类型定义。

7.4 类属类型定义

类属类型定义允许程序员把要在被设置的程序包或子程序内操作的对象类型传送到设置的程序中。其语法重写如下：

类属_类型_定义::=
(<>)|**range**<>|**digits**<>|**delta**<>
|数组_类型_定义
|访问_类型_定义
|私有_类型_定义

“方框”符号<>用来定义标量类型，用法如下：

(<>) 离散类型，如枚举、布尔和字符型
range <> 整型
digits <> 浮点类型
delta <> 定点类型

方框中的“值”直到设置时才填入。

可用于操作给定类型对象的操作集合由类属形式参数来定义。例如，若类属类型定义包括了保留字 **digits**，编译此

刻就知道按浮点类型来处理，而不必等待设置来决定这一事实。因而，这类对象所许可的操作也就知道了。由于使类属子程序作为类属形式参数（参看7.5节)去实施附加的功能，充实了类属类型，因而，这些操作集合大为加强了。注意，如果程序员想使用$IMAGE$和$VALUE$属性，而标量类型的类属类型定义又不许可这样做（尽管其他属性可用），这种特征就很需要了。下面给出几个类属类型定义的例子：

```
type fp_type is digits<>;
type hues is(<>);
type counter is range<>;
type vector is array(counter)of fp_type;
```

7.5 类属形式子程序

*Ada*的类属特征还包括将一个或多个子程序规格说明当作类属形式参数的一部分来传递的能力。这些“类属形式子程序”就像他们的称呼那样，可以有自己的参数，尽管带到类属形式子程序中去的参数可以不是类属的。举几个例子：

```
with function "+" (a,b:complex) return complex;
with procedure swap(a,b:item);
```

类属形式子程序的典型用法是传递一个特定目的的子程序和一个类型。例如，一个资本贬值的程序包可以传递一个子程序，该子程序计算相应类型对象的专用贬值细目表。这个程序包的类属程序包声明是：

```
generic
    type capital_object is (<>);
    with function depreciation_routine
            (a:capital_object)return float;
package depreciation is
    ...
end depreciation;
```

上面给出的例题是充分的，但要求程序员为每种对象提供计算贬值细目单的子程序。我们所期望的就是任选子程序的能力。在 *Ada* 中，将子程序规格说明的末尾加上“ **is** 名字”或“ **is** <> ”就可做到这点。头一种情况的例子，（命名为缺省细目单）是：

```
generic
    type capital_object is (<>);
    with function depreciation_routine
                (a:capital_object) return float
        is default_schedule;
package depreciation is
    ...
end depreciation;
```

使用保留字 **is** 表明出现在那个规格说明中的子程序是可选参数。当缺少与类属形式子程序参数相应的类属实在子程序时，则用指名类属形式子程序的参数结合。例如，给出了以上类属程序包声明，我们可用以下设置的二者之一：

```
--使用专用的贬值细目表
--
```

```
package my_depreciation is new depreciation
        (capital_object=>buildings,
    depreciation_routine=>
                    inventory_depreciation);
--使用缺省的贬值细目单default_schedule
--
package my_depreciation is new depreciation
    (capital_object=>buildings);
```

还有另一类的缺省类属形式子程序，当省略了类属实在参数时就引起编译找遍当前可见的子程序规格说明，试图找到一个在个数、次序、模式以及参数类型都与之匹配的子程序规格说明（如果形式参数是函数，则返回值的类型以及可能的约束也要匹配）。用“is<>”来指明这种类型的缺省参数：

```
generic
    type capital_object is(<>);
    with function depreciation_routine
            (a:capital_object)return float is<>;
package depreciation is
    ...
end depreciation;
```

它允许我们生成像下面给出的程序包设置：

```
--使用专用的贬值细目单
--
package my_depreciation is new depreciation
        (capital_object=>buildings,
    depreciation_routine=>
                    inventory_depreciation);
--缺省值用任意的贬值细目单都行
--
```

```
package my_depreciation is new depreciation
    (capital_object => buildings);
```

7.6 形、实参数匹配规则

完整的类属概念还包括在类属设置过程中传递参数的基础上，将若干程序单元剪辑为一个特定作业的能力。类属形式参数与类属实在参数的匹配则是中心问题。本章余下部分专门讨论实参与形参匹配的规则。

7.6.1 标量类型匹配

标量类型匹配规则最简单。如7.4节谈到的，一个类属形式参数中出现了(<>)就与任何枚举类型的一个实在参数相匹配。同样，**range** <> 与任何整型相匹配，**digits** <> 与任何浮点类型以及 **delta** <> 与任何定点类型相匹配。只允许有这些对应。

7.6.2 数组类型匹配

一个实在数组参数要和形式数组参数相匹配，二者的序标个数必须相同。显而易见，除非序标或成分的类型（或者是两者）是形式类型才可匹配。在这些情况下就执行实际类型名替换形式类型名，然后试着进行实在数组和形式参数的匹配。如果成分类型(以及强加于其上的各种约束)相同，相应的序标子类型也相同，以及相应的序标约束也相同的话，那就是说发生匹配了。只有这些条件都满足才能匹配，否则在试图设置的时候将会引发 *CONSTRAINT_ERROR*(约束_错误）。例子有助于我们阐明这个规则。首先，我们假

定一个以下形式的类属程序包声明：

```
generic
    type item is private;
    type index is (<>);
    type unconstrained table is array
        (index ) of item;
    type constrained_table is array(index
        range <>)of item;
package examqle is
    ...
end example;
```

对于这样的声明，我们可以看到一个形式为：

```
type work_hours is array(day range<>) of hours;
```

的类型将与类属参数“*constrained_table*”匹配，而不与“*unconstrained_table*”匹配。下述类型则相反：

```
type all_hours is array(day)of hours;
```

它仅与“*unconstrained_table*”匹配。

7.6.3 形式子程序的匹配

形式子程序的匹配规则非常类似于7.5节中找缺省子程序的规则。正如前节程序包的情况一样，各形式类型在匹配之前就被替换掉。为使形式的和实在的子程序得以匹配，它们的参数个数必须相同、次序也相同、有同样的类型和模式，并且有相同的约束。如果所研究的子程序刚巧是函数，则其返回值的类型和约束也必须匹配。要注意，像确定掩蔽（参看第5章，第5.6节）一样，参数名字和缺省值的使用均不

影响匹配的确定。为了说明这一点，考虑类属子程序规格说明：

```
generic
    type item is private;
    with function swap(a,b:item)
        return boolean is<>;
function greater(a,b:item)return item;
```

给出了这个规范说明，我们就可生成以下新函数示例：

```
function bigger is new greater(float);
function larger is new greater(character,
                               reverse_collation);
```

注意，第二个设置中要求函数“reverse_collation”可见。

7.6.4 访问类型匹配

如同上节，任何匹配进行之前，要先履行类属类型的替换。在形式访问类型的情况下，如果由访问值取接的“被访问”对象的类型，其形式的和实在的访问类型都相同的话，就可以说实在参数与形式参数匹配了。作为一个例子，给出以下类属程序包声明：

```
generic
    type item is private;
    type item_pointer is access item;
package a is
    ...
end a;
```

还有这些类型：

```
type queue is ...;
type queue_point is access queue;
```

则我们就能使类属程序包的一个示例得以生成，如下：

```
package q is new a(item_pointer=>queue_pointer,
                   item=>queue);
```

7.6.5 私有类型匹配

私有类属形式类型除无约束数组类型而外，可以和任何类型匹配，问题在于约束，即：如果形式类型不是受限的，则实在类型的赋值功能以及测试相等不相等的功能必须有效。如果形式类型有某个判别式部分，则实在类型必须有相同的判别式，带有相同的名字，子类型及缺省初值，还要有相同的次序。

注意，在前面有关论述形式类型和实在类型匹配的各节中，出现任何错误的匹配时，将由设置处理引发 *COSTRAINT-ERROR* 异常。

第八章　异常处理

Ada 最令人感兴趣的特征之一是异常处理概念。*Ada* 语言提供了一种内部机制，当一个干扰条件（用*Ada* 的术语就是一个“异常”）发生时，允许程序员规定某些要作的动作。本章研究*Ada* 处理异常的方法。

8.1　概　述

在程序执行期间的任何时刻都有出错的可能。*Ada* 为一般性的错误处理进程定义了三个阶段。首先是出现了称之为“异常”的错误本身。其次发出一个通知报告一个异常发生了，称为“引发”异常。最后是对付这个异常的动作，称为“处理”异常。作为一个例子，考虑以下程序片段：

```
declare
    x,y:float:=0.0;
begin
    ...
    x:=1.0;
    x:=x/y;    --注：被零除，它得到一个错误的结果
    ...
end;
```

当我们试图被零除时就引起了异常。只要运算处理器（不管是硬件或是软件）通知了问题的环境，异常就被引发，此时由某个标准的系统错误处理程序（本例为 *NUMERIC_*

ERROR）来理处这个异常，它可能打印错误信息并终止本程序。

在许多广为使用的语言中，如*Fortran*, *Basic*, *Cobol*等，查错是程序在其中运行的软件或／和硬件环境的职责。同样，为响应错误的各个动作通常不受用戶控制。例如，若上述程序片段是用*Fortran*写的，就沒有一种机制允许程序员表示出他已经意识到他的程序可能被零除。也不许可他规定一旦发生这种情况时他希望执行的特定代码。结果只能是程序夭折。他只得在使用变量之前艰苦地、遍及程序一一对它们测试。

*Ada*解决这个问题的办法是允许程序员去声明一个异常（给它一个名字），给他以引发这个异常的手段，并许可他提出为响应异常引发而执行的异常处理程序段。

8.2 异常声明

异常声明的目的是引入一个异常的名字，它能随后用于异常处理段和引发语句中（参看第2章，第2.6节）。异常声明的语法是：

异常_声明::= 标识符_表:**exception**;

用于上述被零除的例子，我们可以写出允许我们自己引发异常的子程序：

```
function percent(a, b:integer) return float is
    zero_divide: exception
begin
```

```
        if b=0 then
            raise zero_divide;
        else
            return(float(a)/float(b) * 100.0);
        end if;
    end percent;
```

注意，这里已遵循了在（引发语句）使用一个实体（异常）之前先声明它的一般概念，但我们并未指出异常处理段在哪里，也没有说出它有什么后果。

还要注意语言提供了以下预定义异常：

CONSTRAINT_ERROR	当一个域、序标或判别式违约时引发。
NUMERIC_ERROR	当一个预定义的数值操作给出的结果不在目标对象能被实现的范围之内时引发。
SELECT_ERROR	当一个选择语句所有的备选行不通而又没有*else*部分时引发。
STORAGE_ERROR	为满足一个分配算符的执行而存贮不够时，或任务运行超出了动态存贮时引发。
TASKING_ERROR	若在内部任务通讯期间发生了异常时引发。

这些预定义异常无需声明。

8.3 引发语句

引发语句用来指示程序员希望以显示引发的一个异常（在第2章第2.6节中我们已经见过引发语句）。它的语法如下：

引发_語句::=**raise**[异常_名];

从这个语法定义中看到异常名是可选的。但要省略异常名，只有**raise**语句在异常处理段内部时才可以，这从语法中是不能确定的。在这种情况下，其作用是把引起进入处理段的异常再次引发。

8.4 异常处理段

异常处理段提供了在引发语句(或预定义异常之一)中的异常名和实际的语句之间的联接，这些语句是为响应异常的引发而要被执行的。异常处理段只能出现在块的末尾，或是子程序、程序包、任务体的末尾。它的语法是：

异常_处理段::=

exception

when异常_选择{|异常_选择}=>語句_序列

异常_选择::=异常_名|**others**

我们已经看到把一个异常处理段设置在什么地方的例子，例如，在子程序的语法定义中（参看第5章、第5.2节）。

为了探究当一个异常被引发随后被处理所发生的一系列事件，让我们回到被零除的例子。这次我们提出自己的异常

处理段：

```
function percent (a,b:integer)return float is
    zero_divide:exception;
begin
    if b=0 then
        raise zero_divide;
    else
        return(float(a)/float(b)*100.0);
    end if;
exception
    when zero_divide=>
      put("percent:Attempt to divide by zero");
        return 0.0;
end percent;
```

我们现在已能完全由我们自己来处理被零除的情况了。当被零除这个条件被查出（$y=0$），我们就引发这个"*zero_divide*"异常。引发异常使得程序单元(块、子程序体、程序包体、任务体）的执行被悬置起来，并去执行一次搜索，找程序员是否为此异常指定了一个处理程序段。如果有，继续执行带有这个名字的处理段，否则程序单元的执行被终止，异常则向前"传播"（看第8.5节）。本例中，我们为*zero-divide*找到了一个异常处理段，因此，继续执行异常处理段中的"*put*"语句。注意，出现在引发语句和异常处理段之间的任何语句都被跳过。

此例要注意两个要点。第一，当进入异常处理段时，子程序的执行并没有废止，只不过是控制转移了。这点很重要，尤其是异常在主程序或任务中出现的时候，如若找不到

用户提供的处理段就会产生程序或任务终止的后果。如果提供了异常处理段，则用户保有控制程序终止的权益。第二，因为异常处理段是程序单元的一部分，所以它可以访问程序单元的所有资源。于是，上例中，我们就能执行为函数而设的**return**语句。同样，再试图作原来的操作之前有机会去执行某个修正的动作（当然，要仔细避免无限循环和死锁）。

如果我们不得不测试所有被零除的地方，那么我们比上面提到的可怜的*Fortran*程序员好不了多少。然而，我们可以为预定义异常指定一个异常处理段，该段使我们得以改写上例如下：

```
function percent(a,b:integer)return float is
begin
    return(float(a)/float(b) * 100.0);
exception
    when NUMERIC_ERROR=>
      if b=0 then
            put("percent:Attempt to divide by
            zero");
        else
            put("percent:NUMERIC_
            ERROR");
        end if;
        return 0.0;
end percent;
```

如果“*b*”凑巧为零，则预定义*NUMERIC_ERROR*异常将按正常事件的发展引发。然而，在本例中，我们提供了当场处理这个错误的自己的异常处理段，它允许我们执行**eturn**并继续执行调用程序。注意，这里没有要求程序员去

做显式的测试和引发，所有的处理都是自动的。还要注意异常处理段中附加的代码，它们是用来判定异常是否真正由被零除而引起。之所以要这样做是由于我们并不知道*NUMERIC_ERROR*产生的真正原因，它可能由我们计算中的另外事件所引起。

最后，没有什么理由禁止我们显式地引发一个预定义的异常。我们也可以用*NUMERIC_ERROR*来写被零除的例子：

```
function percent(a,b:integer)return float is
begin
    if b=0 then
        raise NUMERIC_ERROR;
    else
        return(float(a)/float(b) * 100.0);
    end if;
exception
    when NUMERIC_ERROR=>
        put("percent: Attempt to divide by
        zero");
        teturn 0.0;
end percent;
```

它和我们声明过并在其中使用“*Zero_divide*”异常的例子效果是一样的。

剩下最后一个还要研究的结构，即异常处理段中的**others**子句。它作为一个“安全网”去处理所有没有显式处理段的异常，如以下情况：

```
function percent(a,b:integer)return float is
begin
```

```
        return(float(a)/float(b) * 100.0;
    exception
        when NUMERIC_ERROR=>
            put("percent:NUMERIC_ERROR");
            return 0.0;
        when others=>
            put("percent:Unknown Exception");
            return 0.0;
    end percent;
```

8.5 异常到处理段的连接

我们现在转而研究一旦引发了异常，用来查找为处理异常的异常处理段的机制，以及控制异常传播的规则，还有程序单元终止的规则。这些就构成了*Ada*异常处理系统的核心。

有两类规则要研究，一是声明确立期间引发异常，一是语句执行期间引发异常。将它们分开研究要方便一些。

8.5.1 确立期间引发异常

为确立期间出现的异常所采取的动作要看是什么样的确立。我们以子程序为例，其声明确立时能产生的异常引发是：

```
declare
    function search(a:character;
                   b:string    )return integer;
        index:integer:=b'last+1;--超出了串
                                 --末尾
        char:character:=b(index);----个约束错误
```

```
    begin
        ...
    end search;
```

如果在子程序声明部分确立期间引发一个异常，如本例，则确立就被废弃了。同时在导致确立的程序单元中引发异常。为了说明，我们将上面的例子用于更大的环境中：

```
declare
    start:integer:=search('i',"Hi There");
begin
    ...
end
```

如果我们想要处理在声明部分确立 "search" 时引发的异常 CONSTRAINT_ERROR，我们必须在本块内提出它，而不在子程序中。于是，我们就得写成这样：

```
function search(a:character;
                b:string)return integer;
    index:integer:=b'last+1;--超出串末尾
    char:character:=b(index);--当这些被确立时，
                             --将产生约束错误
begin
    ...
end search;
    declare
        start:integer;
    begin
        ...
        start:=search('i',"Hi There");
```

```
        ...
    exception
        when CONSTRAINT_ERROR=>
            block_handler;
    end;
end;
```

于是我们就将子程序声明部分确立期间引发异常的传播规则陈述为：异常使声明部分的确立废弃，并在调用它的单元内引发异常。若子程序本身就是主程序，则程序的执行将终止。因此，调用程序单元有责任提供所必需的异常处理段，如果它想那样做的话。

块中声明部分引发的异常其规则和上述非常类似：声明部分的确立被废弃，异常被传送到包括这个块的程序单元。这可以用以下例子说明。将这个例子和前面的例子比较，要注意本例的异常是由“*something_handler*”来处理，而不是“*block_handler*”。

```
procedure something is;
    function search(a:character;
                    b:string)return integer;
        index:integer:=b'last+1;
        char:character:=b(index);
    begin
        ...
    end search;
begin
    declare
        start:integer:=search('i',"Hi There");
    begin
```

```
    ...
exception
    when CONSTRAINT_ERROR=>
        block_handler;
end;
exception
    when CONSTRAINT_ERROR =>
        something_handler;
end something;
```

在程序包体声明部分确立期间引发的异常，也会使确立废弃，并使异常传播到包容这个包体的程序单元。如果这程序包刚巧是库单元，则此程序废弃。

若在任务体声明部分确立期间引发异常，则确立废弃，异常传播到激话任务的程序单元。

若异常在子程序声明、程序包声明、任务声明的确立期间发生（不同于声明部分确立期间引发异常），则确立废弃，异常传播到包含这个声明的程序单元。和前面一样，若子程序声明或程序包声明是库单元声明，则程序将终止。

8.5.2 语句执行期间引发异常

当语句执行期间出现异常时，准确地引发出什么样的动作序列取决于包括这个语句的程序单元的情况。在我们讨论异常和异常处理段语法（第8.1节和8.4节）的过程中，我们已经有了一些直觉的接触。现在再提供一些细节。

如果一个块在执行期间引发了一个异常，该块又没有为此异常供提一个处理段，则块的执行被终止，同时在包含此块的程序单元内引发同样的异常。这可用下面的例子来说明：

```
outer.
declare
    ...
begin
    ...
    inner:
    declare
        type small_int is range 0..10;
        a:small_int;
    begin
        ...
        a:=10;
        a:=a*a; --此处引发
                --CONSTRAINT_ERROR
        ...
    end inner;
    ...
exception
    when CONSTRAINT_ERROR=>
        ...
end outer;
```

"inner"(内层)块中 CONSTRAINT_ERROR 被引发，"inner" 中其余的语句均被跳过。由于 "inner" 无处理段，"outer"(外层)块又一次引发这个异常。由于"outer"中有一个为此特定异常的异常处理段，则继续执行这个处理段。

下一步要考察的情况是子程序体中的语句执行时引发了异常，而体中又没有提供相应的异常处理段。在这种情况下，子程序的执行就废止了。同一异常在调用它的程序中再次被引发（如果这个体是主程序体。此时，程序就终止）。

```
decclare
    procedure xyz; --总是引发
                   --NUMERIC_ERROR
    begin
        ...
    end;
    procedure abc;  --总是引发
                    --CONSTRAINT_ERROR
    begin
      ...
    end;
begin
    ...
    xyz;  --如有异常由outer_handler处理
    ...
    declare
        ...
    begin
        ...
        abc;  --由general_handler处理
        xyz;  --由inner_handler处理
    exception
        when NUMERIC_ERROR=>
            inner_handler;
    end;
exception
    when NUMERIC_ERROR=>
        outer_handler;
    when others=>
    general_handler;
end;
```

如果我们现在注意程序包体里的异常引发，就会发现若程序包中缺少应提供的处理段，程序包的确立就废弃了（要记住，一个程序包的确立是由它的声明部分确立所组成，随后才是一些语句的执行，所以说它是确立的废弃是正确的），同时异常在包容程序包体的程序单元中再度引发。如第8.5.1节所说，若程序包是库单元则主程序被终止。

对于任务体执行期间引发的异常，若无为此而设的处理段，此任务被终止。没有异常传播的问题。

我们必须处理的最后一种情况是在异常处理段的语句执行中又产生了进一步的异常。如果发生这种情况，程序单元的执行被废止，同时异常按前述方式传播。

8.5.3 任务通讯期间引发异常

*Ada*为支持任务提供了同步和通讯原语，它们是**entry**和**accept**语句(详见第9章'任务')。一个任务发出一个**entry**调用到另一个任务（它和前者同步执行），而被调用任务则发出一个**accept**语句。在执行与**accept**有关的语句之前，它一直等待**entry**的调用。这就会以下述两种方式中的一种产生异常的引发。第一种是：被调用任务可能已经终止，因而不能发出**accept**语句。此时调用任务在调用点处引发异*TASKING_ERROR*异常，如下例：

```
task body a is
    ...
begin
    ...
x.read(char);  --若任务x已死,引发
               --TASKING_ERROR
```

```
    ...
exception
    when TASKING_ERROR=>
        ...
end a;
```

注意，由于被调用任务已不执行，这种异常的形式仅能影响到调用任务。

异常能出现的第二种形式是：当与被调用任务的 **accept** 语句相结合的语句执行时产生了异常，而对此异常又没有局部的处理段（“局部”意味着在包含**accept** 的程序单元内）。如果发生这种情况，异常就按8.5.2节给出的规则传播，同时还要传播到调用任务的调用点。

还有另外一种与任务有关的异常，这就是属性 $FAILURE$（失败）。任务也是类型，像类型一样有他们的属性，其中就有属性 $FAILURE$。*Ada* 允许任何任务引发在其他任务中的这个异常，这时可写：

```
raise his'FAILURE
```

这里“*his*”是要在其内引发异常的任务名。这对发出 **raise** 的任务没有什么影响，但对目标任务（“*his*”）有“中断”的作用。如果目标任务被悬置（睡着的），它就可以被唤醒，因而可用这种干扰去对付它。

8.6 检查的抑制

为了使预定义异常能被引发，就有理由假设一些检查，它们用来判定相应的异常（例如，检查数组的界）是否要引发。*Ada* 允许程序员有选择的“关闭”一些检查。方法是借

助于pragma（编用），其语法是：

pragma *SUPPRESS*(检查_名[,[*ON*=>]名字]);

这就允许程序员去抑制一个特定的检查（在一个块、子程序体、程序包、任务之内），其类型是可选的。下面几个例子有助于说明这一点：

```
pragma SUPPRESS(OVERFLOW_CHECK);
pragma SUPPRESS(INDEX_CHECK,
                ON=>vector);
```

可以执行也可以关闭的检查是：

ACCESS_CHECK	当试图用一个访问值去访问一个对象时，检查访问值是否为空**null**。
DISCRIMINANT_CHECK	检查一个记录成分是否存在，以及是否满足判别式约束。
INDEX_CHECK	检查序标值是否满足对它的约束，或是否和序标类型相容。
LENGTH_CHECK	检查序标成员是否是所要求的个数。
RANGE_CHECK	检查一个值是否满足范围约束，或约束是否与类型标记相容。
DIVISION_CHECK	检查是否被零除。
OVERFLOW_CHECK	检查一个数值结果是否溢出。
STORAGE_CHECK	检查存贮空间够不够。

注意，*STORAGE_ERROR*是由*STORAGE_CHECK*产生的；*NUMERIC_ERROR*由*DIVISION_CHECK*或*OVERFLOW_CHECK*产生的；其余情况则产生*CONSTRAINT_ERROR*。

在抑制检查方面要说的最后一句话是：记住一个要点，*SUPPRESS*编用仅仅是向编译推荐可以省去运行时刻的相应检查。但由于传播规则的作用，异常仍有可能发生。通常，除非有十分充足的理由才那样做，一般不应当抑制检查。因为它是一个安全措施，不应当轻易地扔掉。

第九章　任　务

本章我们将涉及*Ada*支持任务（其他语言曾经称之为进程）生成和维护的结构，以及任务的同步和通讯机制。

9.1　概　述

*Ada*中的任务可以设想为一个在时间上与其他任务独立执行的程序单元。这种异步操作要求相互合作的任务有一种可用的结构，这个结构支持同步和它们之间的通讯。作为一个例子，我们考虑编写一个行编号或叫伪脱机任务的问题。它的功能是接收来自用户任务的数据，加上行号和页次，将改造了的数据传送到第二个任务——打印机驱动器。这就要求用户任务能把信息发送到行编号任务，行编号任务应能等待一段时间直到他接收到数据为止，还要求行编号任务能把信息发送到驱动器任务。行编号任务的规格说明可以是：

```
task line_numberer is
    entry get_data(a:in item);
end x;
```

则相应的任务体是这样：

```
task body line_numberer is
    temp:item:
begin
    loop
```

```
            accept get_data(a:in item)do
                temp:=transform(m);
                printer_driver.put_data(temp);
            end;
        end loop;
    end x;
```

本例有几点要注意。第一，和其他*Ada*对象一样，有一个规格说明和一个相应的体。第二，规格说明定义了本任务和任何与之合作的任务之间的界面，这些外面的任务经由**entry**(入口)语句和本任务交换信息。最后，注意打印机任务要有一个**entry**语句定义 *put_data*代码，像“*line_numberer*”中定义*get_data*一样。

我们并没有对任何任务规定它应在何时运行，也没有指明它应多快，了解这些是很重要的。它们是独立的且能并行运行，甚至于可在不同的机器上运行。我们只能肯定“*line_numberer*”要等待从用户任务中输入，以及向打印机任务发送输出（按推测打印机任务也要等待“*line_numberer* 的输出）。但是这样叙述也有点毛病，因为如果行编号任务产生的输出快于打印机任务处理的能力，则过剩的就要排队(理所当然)。不过，按这种办法时间上的依赖性最小。

我们已经尝到了任务的风味，现在让我们把注意力转向任务的定义并考察它更多的细节。

9.2 任务定义

和子程序和程序包一样，在使用任务之前必须先声明它，并且用术语‘任务规格说明’和‘任务体’来定义任务。任

务声明的语法是：

```
任务_声明::=任务_规格說明
任务_规格說明::=
    task[type]标识符[is
        { 入口_声明 }
        { 表示法_规格說明 }
    end[标识符]];
任务_体::=
    task body标识符 is
        [声明_部分]
    begin
        語句_序列
    [exception
        { 异常_处理段 }]
    end[标识符];
```

任务规格说明定义了任务和系统其他部分的界面。它履行和子程序规格说明同类的功能（见第5章，5.2节）。它为任务对外设置了“入口点”，这些入口点的标识符由系统可见，并能被其他任务调用。此外，任务规格说明可以包括一个为特定类型而设的表示法规格说明，该规格说明给出下述项目的信息，如任务所需要的存储量以及对机器的依赖性，如外围设备的物理地址等。

与其他各对规格说明——体一样，体和规格说明的标识符必须相互匹配，最后end后面的可选标识符也要匹配。对于任务规格说明还有一个附加的约束，即规格说明必须和任

务体出现在同一声明部分，且行文上必须先于体出现。

*Ada*中一个任务就是一个对象。和*Ada*其他对象一样，任务对象也有与之结合的类型，由任务规范说明定义。一旦任务类型被定义，就可用它来生成相应的任务对象。例如，假设在行编号的例子中有几台打印机可供选择。这就必须有好几个打印机任务来驱动它们，它们各不相同但履行同一功能。我们可以定义一个任务类型如下：

```
task type printer_driver is
    entry put_data(a:item);
    entry reset;
end printer_driver;
task body printer_driver is
    ...
begin
    ...
end printer_driver;
```

用上述任务类型定义，我们就可用它来声明所需要的打印机驱动任务：

```
declare
    printer_1, printer_2;
    printer_3: printer_driver;
    x:item;
    ...
begin
    printer_2.reset;
    ...
    printer_2.put_data(x);
    ...
end;
```

这里，我们声明了三个任务对象，只用其中之一“*printer_2*”来说明它们的使用。尤其要注意，为了指明我们所希望访问的特定任务的入口，我们不得不用成分选择。

注意，任务类型定义了一组相应任务对象所拥有的入口。还要注意由于任务也是对象，所以它们也可用于数组和记录。只要遵守一个限制，即任务对象没有赋值和相等——不相等测试的功能（从这一点说，任务对象可以看作是由受限私有类型生成的，对象的值已在对象声明确立时定义了，或经由一个分配算符在它们建立后定义）。从上面的叙述可得出如下结论，当形式参数是 **inout** 或**out**模式时，任务对象不能作为实在参数。然而，当模式是 **in**时，任务对象的传递是允许的。

最后还要注意一点，如果你回头看看任务规格说明的语法定义，你就会发现**type** 字是可选的。如果省略了，规格说明及相应的体就引入了一个单一的任务对象（它由给出的标识符得知）。而不是引入任务类型，若是任务类型，随后要求显式的对象声明。

9.3 任务属性

任务类型和任务对象和*Ada*其他的类型和对象一样，有与之结合的属性。这些属性的值可用下述属性询问函数在任务对象或任务类型（例如，叫做“*a*”）上求出：

a'PRIORITY　　任务“*a*”的优先数，要事先定义。它的值是子类型*PRIORITY*的一个整数（参见9.

10节）。

$a'STORAGE\_SIZE$	分配给任务“a”的存贮单元数，整数。
$a'TERMINATED$	一个布尔值，当任务“a”被声明时置假值，当“a”终止时置真值（参见9.5节）。

除了与任务本身相结合的属性以外，还有一个附加的询问函数$COUNT$。它允许程序员确定对某了入口，比如“a”，的等待队列中入口调用的个数：

$a'COUNT$	与入口“a”相结合的等待队列中入口调用的个数，一个整数。

9.4 任务的生成和初始化

我们已经讨论了任务定义的语法，但还需详细阐述任务对象声明后所发生的一系列事件。我们也没有准确地说过任务在什么时候开始执行。

和其他声明一样，在程序单元的声明部分确立期间任务对象就被声明了。然而，对于任务，除了引入对象本身而外还有一附加的步骤。对这一步骤“激活”的响应，就开始了任务的执行。任务体声明部分的确立；随后，构成任务体其余部分的语句序列开始执行就组成了“激活”。注意，这就意味着任务要先于本程序单元中跟在声明部分后的任何语句而执

行。为了说明，看下例：

```
declare
    ...
    printer_1:printer_driver; --printer_1 确立
    ...
begin--执行到这里时printer_1被激活
    First_statement;
    ...
end;
```

在同一个声明部分有多于一个任务被声明时，它们被激活的次序并没有定义：

```
declare
    ...
    printer_1,printer_2:printer_driver;
    printer_3:printer_driver;
    ...
begin
    --printer_1,_2,_3被激活，但未定义次序！
    ...
end;
```

如果一个任务对象由分配算符生成，它马上被激活并开始执行。分配算符的执行直到任务被激活后才算完成。这就意味着我们现在可以对任务激活的次序作某种控制。于是，我们有下例：

```
declare
    ...
    type dynamic_printer is access printer_driver;
    printer_1,printer_2:printer_driver;
```

```
        spare_printer:dynamic_printer:=new
                                        printer_driver;
        --spare_printer 在此处就被激活了！！
        …
    begin
        --printer_1,_2 在此处被激活，三个任务现在
        --都处于执行状态
        …
    end;
```

注意，由于入口调用在进程中排好了队，所以，甚至任务还没有被激活，任务的入口就可调用。

我们现在不得不涉及在任务激活期间异常引发的可能性。当一个异常发生时，实际上只要遵循四条规则：

1. 任何早先被激活的任务不受影响。

2. 任何后续的任务被认为是“终止”了（见下文）。

3. 异常就当作在跟着声明部分后面的语句中发生的那样看待。

4. 如果异常在声明部分中发生，则到此为止，所有被确立过的任务均予终止。

下两个例子应能阐明上述规则：

```
    declare
        p_1,p_2:printer_driver;
        spare:printer_driver; --此处引起的异常
        p_3:printer_driver;
    begin
        --p_1,p_2 执行，spare和p_3 终止
        …
    end;
```

```
declare
    p_1,p_2:printer_driver;
    x:integer range 1..10:=y; --产生异常
    spare:printer_driver;
    p_3:printer_driver;
begin
    --p_1,p_2,p_3 和 spare 都终止了
    ...
end;
```

9.5 任务终止

现在我们已经有可运行的任务了，此时再看它是如何停，或“终止”的。

上面已经说过，任务是在块、子程序等的声明部分中声明的。拿下面的例子来说：

```
declare
    ...
    p_1:printer_driver;
    ...
begin
    --p_1在此处激活
    ...
end;
```

考查上面的例子，我们有理由问一问，当执行到了块（即声明任务的单元）末尾任务和块都会生发什么事情呢？为了回答这点我们要引入“任务依从性”的概念。*Ada* 中，每个任务都“依附”在一个块、子程序体、任务体、库程序包之中。换

句话说，对给定的程序单元，若任务在单元（包括内层的程序包）内声明，但不在内层单元之中；或由访问类型值指示的任务且其访问类型已在单元（包括内层的程序包）内声明，但不在内层单元之中，则称此任务为“依附”于该单元的任务。

我们现在可为上段开头提出的问题提供一个解答。语言定义指出：直到依附任务终止之前程序单元（我们的情况是块）不会结束。于是，“*p_1*”在我们从块中出来之前必须结束。注意，这个规则甚至在没有异常处理段而使之强行出口的异常情况下也是适用的。

从任务的观点看，当任务体执行到终点同时它所有的依附任务（如果有的话）都已终止，则本任务终止。任务也可以执行选择语句（看下文）中的**terminate**后而终止。下例说明依从性：

```
task type timer is …end timer;
declare
    …
    type at_timer is access timer;
    a:timer;
    …
begin
    -- “a”在此处激活
    …
    declare
        …
        b:timer;
        c:at_timer:=new timer;
```

```
        -- "c.all" 在此处激活
        ...
    begin
        -- "b" 在此激活
        ...
    end;-- "b" 在此等待终止
    ...
end;      --"a"和"c.all"在此等待终止
```

9.6 会合概念

为了使两个任务同步，两个任务必须同意"相会"，即早到达的任务要等待晚到达的。这就是"会合(*rendezvous*)"的概念，在*Ada*中是用**entry**（入口）和**accept**（接受）语句来实现的，它们的语法定义如下：

```
入口_声明::=
    entry标识符[（离散_范围）][形参_部分];
入口_调用::=入口_名[实参_部分];
接受_语句::=
    accept入口_名[形参_部分] do
        語句_序列
    end[标识符];
```

任务的入口声明定义了任务与系统其余部分的界面。任务规格说明中的每个入口声明在任务体中都有一个相应的接受语句。有时一个特定的**entry**可有多个**accept**语句，这种情况下调用任务必须使用一个索引来规定它所希望的调用。用任务规格说明和任务体来说明这一点：

```
task type printer_driver is
    entry write(speed) (a:item) ;
end printer_driver;
task body printer_driver is
    ...
begin
    ...
    accept write(slow)(a:item)do
        ...
    end;
        ...
    accept write(fast)(a:item)do
        ...
    end;
    ...
end printer_driver;
```

调用任务为了访问入口“write”必须使用以下方式：

```
declare
    a:printer_driver;
    x,y:item;
begin
    ...
    a.write(slow)(x);
    a.write(fast)(y);
    ...
end;
```

在这类情况下，一个单一的entry语句有一系列的accept，就是说有所谓的入口“族”。由于索引用来从族中唯一确定所要求的accept语句。所以如果索引范围违约，就有引发

*CONSTRAINT_ERROR*异常的可能。

我们现在进而研究会合系统的核心。问题的讨论逐渐深入到与accept语句相结合的语句序列的执行上来了。当一个入口被调用时，调用任务被挂起来，直到与accept相结合的语句（如果有的话）被执行完。现在有两种可能性：被调用任务已经在accept语句处等待，此时语句序列就立即执行；也可能有一个不确定的时间间隔要求被调用任务过一会儿到达accept。不管哪种情况，我们让调用任务等待到与accept相结合的语句执行完为止，我们就能使两个任务得到有效的同步。这就是*Ada*的“会合”，一旦被联接到accept的语句得以执行，则调用任务和被调用任务就平行地自由执行。下面的例子示出我们刚刚讨论的原理：

```
task a is
    entry sync;
end a;
task body a is
degin
    loop
        ...
        accept sync do
            null;
        end sync;
        ...
    end loop;
end a;
declare
    ...
begin
```

```
        ...
        a.sync;
        next;   --在任务"a"的"end sync;"之后执行本句
        ...
    end;
```

在上段的陈述中，有可能在被调用任务被激活之前调用一个入口。这种可能性是因为每个**entry**都有单独与之结合的队列。当入口被调用时，它们就被置入队列。随后被调用的任务请他们出队，一次一个，每次都执行相应的**accept**语句。注意，*Ada*语言中定义的队列是按"先进先出"原则来处理的。如前所述，任何想调用已终止任务内的入口的企图都将引发*TASKING_ERROR*异常。

9.7 延迟语句

在程序设计中有很多情况需要能将一个任务的执行悬挂一段时间。操作系统的程序设计尤其这样。为了完善*Ada*的任务概念，延迟语句作为语言的一部分提供了这种功能。它的语法是：

延迟_语句::=**delay**简单_表达式；

简单表达式解释为按秒的时间延迟。它的求值精度必须同强加于其上的约束一致，该约束即与实现密切相关的、预定义定点类型*DURATION*约束，在程序包*STANDARD*中可找到。然而，语言定义还要求它可表示的值达到86400，这正好是一天的秒数。延迟语句应用可如下例：

```
declare
    tick:constant DURATION:=1.0; --时间间隔
                                 --按秒
```

```
    counter:integer:=0;
begin
    loop
        delay tick; --等待约一秒钟
        counter:=counter+1;
    end loop;
end;
```

要知道，delay语句并不能精确地保证所指出的延迟时间，只是延迟至少那样长。了解这一点是非常重要的。实际的延迟很可能要更长一点。这是由于像系统加载、任务调度延迟等等因素造成的。

延迟语句中使用负值在逻辑上无意义，但不影响在任务中的执行。

9.8 选择语句

入口调用执行时都隐含地规定：调用任务要停止执行一段不确定的时间。这对于那些不许可延迟的任务尤其不利，而调用之先，调用程序又不能直接访问用来测试是否要等待的参数。accept语句也存在同样的问题。select（选择）语句的设置就是为了解决这些问题，它给程序员提供了一些方法去控制接受语句或入口调用是否执行。select 语句的语法如下：

```
选择_语句::=
    选择_等待
    |条件_入口_调用
    |定时_入口_调用
```

9.8.1 条件入口调用

条件入口调用用于这种情况，即程序员希望测试能否在

调用点立即和被调用任务会合。它的语法是：

```
条件_入口_調用::=
select
    入口_調用[語句_序列]
else
    語句_序列
end select;
```

如果调用任务不能立即与被调用任务会合，则执行 else 部份的语句组。如果会合立即满足，则入口调用就发生了。为了说明，我们考虑一个轮询键盘任务的问题，该任务是查看是否按下了键，如果找到了就将该键入符置于队列。若不能将键入符作排队处理，则给本系统其他任务置以标识。我们的任务可以是这样：

```
task keyboard is
    entry reset;
end keyboard;
task body keyboard is
    ...
    flag:boolean:=false;
    key:character;
    ...
begin
    loop
        read_keys(key);
        select
            a.queue(key);
            flag:=false;
```

```
            else
                flag:=true;
            end select;
        end loop;
    end keyboard;
```

9.8.2 定时入口调用

定时入口调用类似于条件入口调用，只是在这种情况下，对其进行的测试不是判定是否立即满足会合，而是判定是否能在给定时间内满足会合的要求。定时入口调用的语法是：

```
定时_入口_調用::=
    select
        入口_調用[語句_序列]
    or
        延迟_語句[語句_序列]
    end select;
```

对于定时入口调用，仅当能在延迟语句所指定的时间间隔内开始会合指名的入口时才能被调用，否则就执行延迟语句。我们再看看键盘任务的例子，这次不同的是若不能将键入符排队，我们就等待，如连试三次以后还不能排上队，则放弃这个键入符。

```
task keyboard is
    entry reset;
end keyboard
task body keyboard is
    ...
```

```
        counter:integer:=0;
        limit:constant integer:=3;
        flag:boolean:=false;
        key:character;
    begin
        loop
            read_keys(key);
            select
                a.queue(key);
                counter:=0;    --若列队成功计数器复
                               --位;
            or
                delay 0.5;
                counter:=counter+1;
            end select;
            exit when counter>3;
        end loop;
    end keyboard;
```

9·8.3 选择等待语句

我们现在转而研究最后一个选择语句，即选择等待。上节曾涉及到 entry 语句的使用者可用的选择，现在选择等待语句又对其作了补充，它提供了对accept 语句的控制。选择等待语句的语法是：

```
选择_等待::=
    select
        [when条件=>]
            选择_备选
```

```
        [or[when条件==>] ]
            选择_备选
        [else
            語句_序列]
        end select;
选择_备选::=
    接受_語句[語句_序列]
  |延迟_語句[語句_序列]
  |terminate;
```

下面的例子示出选择等待语句的一个可能的用途：

```
task body space_manager is
    ...
begin
    loop
        ...
        select
            when space_available==>
                accept allocate_space do
                    ...
                end;
        or
            accept return_space do
                ...
            end;
        end select;
    end loop;
end space_manager;
```

当选择语句执行时，就测试哪个备选是“开启”的，即选择没有when子句或when子句求值为真的备选。到底执行哪个开启的备选还要取决于备选本身，按以下约定。

delay 如果某个开启的备选从delay语句开始，则在when条件求值后马上求延迟的值。如在指定的延时时间消逝前，没有其他的备选可选，则选delay备选。

accept 如果某个开启的备选从accept语句开始，则当其他任务向本accept发出入口调用时，则选本备选（看下文）。若本备选选中，则accept语句及其相应的语句序列被执行。

terminate 如个某个开启的备选用了terminate语句，则仅当该任务依附于某程序单元而该单元又到达结束（*end*）语句时（即它在等待依附任务的终止），则选本备选。此外，若此程序单元所有的依附任务不是终止就是在terminate备选上等待，则选本备选。注意，若此任务依附的程序单元本身又是一个任务，则结束的定义应包括一个terminate备选。terminate语句的执行使任务得以正常终止。

若无立即可选的备选，则当存在else部分时它被执行，否则任务将等待，直到某个开启的任务可选为止。若无开启的备选，则当存在else部分时它就被执行，否则将引发$SELECT_ERROR$异常。下面几个例子可以阐明这些规则：

```
...
open:=true;
```

```
select
    when open=>
        accept xyz do
            ...
        end;
or
    when not open=>
        accept abc do
            ...
        end;
end select;
```

本例中，第一个备选是开启的(因为条件为真)。因而第二个未开。如果另一任务产生了对“xyz”的入口调用，则由于accept是仅有的开着的备选，故该语句被选。如果没有发出这个入口调用，由于没有else部分可执行，则此任务将等待调用。

```
...
open:=true;
select
    when open=>
        accept xyz do
            ...
        end;
or
    accept abc do
        ...
    end;
end select;
```

本例中，两个备选都是开启的。如果对“abc”或“xyz”都没有

发出入口调用，则任务将在第一个那里等待。然而，如果对两者同时发出入口调用则备选的选定是随意的。这就是为什么前面叙述accept备选时只规定开启的备选是可选的。如给出了几个接受备选而且都是开启的，并可立即实行会合，则不能保证实际会选上哪一个备选。

9.9 任务夭折

abort（夭折）语句是非正常地终止一个或多个任务的手段，其语法是：

夭折_语句::＝**abort**任务_名{，任务_名}；

因为是非正常终止，所以不仅指名的任务终止，而且依附于其上的任何任务以及由它调用的程序单元均终止。和你预想的一样，如果指名任务已激活，则它被终止，若任务已终止则本句无作用。依据**abort**语句执行对会合的影响，采用以下规则：

1. 若在调用一个入口的进程中任务夭折，则此入口调用从被调用任务的等待队列中取消。
2. 若在执行会合的进程中任务夭折，则调用任务夭折，但还允许被调用任务完成其正常的接受功能。
3. 若尚有未处理的入口而任务夭折，则所有发出入口调用的任务在其调用点引发异常 *TASKING_ERROR*。若此调用为定时入口调用，则延迟被删去。

非正常终止任务的办法是在别的办法都失败了的情况下作为最后的手段而设置的。终止一个任务的推荐方式是在任务中

引发$FAILURE$异常，执行$FAILURE$的异常处理段可使任务收拾好选择项目后再终止。而 abort 语句甚至连这点也做不到。

9.10 任务优先级

多数的任务系统包括了任务优先级的概念。这是指示某个任务当它排列到处理器的队列之时相对于其他任务的重要性。*Ada* 允许程序员用编用（*pragma*）为他的任务指定一个他所希望的确定的优先数：

pragma $PRIORITY$（静态_表达式）

将它置于任务规格说明中（如果我们要在主程序中设置优先级，则将它放在最外层声明部分中）。注意，这个优先级是在编译时刻确定的。小的值表示比大些的值优先级低。优先数的值就是预定义子类型$PRIORITY$ 的整数值，其值域要取决于实现。

最后要注意优先级还涉及到会合执行时的优先问题。若会合的任务双方或二者之一定义了优先级，则按高的优先级执行。若任务双方都没有定义优先级，此时会合执行时的优先级如同没有定义一样。

第十章　程序结构、作用域和可见性

本章我们把前面几章串在一起讲述*Ada*程序单元的结构以及单元之间相互关系，还有标识符的作用域和可见性。

10.1　程序结构

在前面章节中，我们已经涉及到“程序单元“和”库单元”等。现在就用它们去定义一个*Ada*程序，再看看我们如何把这些都已见识过的程序单元捏在一起形成一个程序。

*Ada*中，程序被定义为几个（一个或多个）“编译单元”，每个编译单元的成员是同时编译的。编译的语法如下：

```
编译　::＝{ 编译_单元 }
编译_单元::＝
      上下文_规格說明　子程序_声明
    | 上下文_规格說明　子程序_体
    | 上下文_规格說明　程序包_声明
    | 上下文_规格說明　程序包_体
    | 上下文_规格說明　子单元
上下文_规格說明　::＝{ 带有_子句 [使用_子句] }
带有_子句　::＝with单元_名{，单元_名}；
```

它的最简单形式是一个程序可由单一的编译单元组成，

如下面的例子：

```
with text_io;
procedure temperature_conversion is
    use text_io;
    package real_io is new float_io(float);
    use real_io;
    Fahrenheit,Celsius:float;
begin
    get(Fahrenheit);
    Celsius:=(Fahrenheit - 32.0)*5.0/9.0;
    put(Celsius);
    put(newline);
end temperature_conversion;
```

注意，在每个编译单元的最外层都隐含地声明了*STANDARD*程序包，所以*STANDARD*可见部分中的标识符在所有的编译单元中均可用。通过这种机制我们就可使用在*STANDARD*中声明了的标识符。例如，“*float*”类型。

不幸的事是实际问题并不总像上面给出的例题那么简单，所以我们在考虑比较实际的问题之前先定义几个术语：

程序库　用来组成程序的若干编译单元都隶属于程序库。

库单元　一个编译单元若非另一程序单元的子单元则该编译单元为库单元。

子单元　子程序体、程序包体或任务体在某编译单元内声明，但从该单元中分离出来编译则称它是声明单元的子单元。

我们现在考查一个编译的示例，看看整块代码如何分解为编译单元以及编译单元和上述三个定义有么什关系：

```
procedure screen_io is
    package cursor is
        x_coord:integer range 0..79;
        y_coord:integer range 0..23;
        procedure home;
        ...
    end cursor;
    package body cursor is
        procedure home is
        begin
            x_coord:=0;
            y_coord:=0;
            ...
        end home;
    begin
        home;
    end cursor;
    procedure move(x,y:integer)is
        use cursor;
    begin
        x_coord:=x;
        y_coord:=y;
    end move;
begin
    cursor.home;
    ...
end screen_io;
```

上面这个例子是一个完整的程序，由单一的编译单元组成。

按前述定义它也是一个程序库和一个库单元。我们现在把同一程序分解为三个分别编译的单元，如下例：

```
with cursor;
procedure screen_io is
    procedure move(x,y:integer) is
        use cursor;
    begin
        x_coord:=x;
        y_coord:=y;
        ...
    end move;
begin
    cursor.home;
    ...
end screen;
```

为“主程序”，

```
package cursor is
    x_coord:integer range 0..79;
    y_coord:integer range 0..23;
    procedure home;
    ...
end cursor;
```

为程序包规格说明，

```
package body cursor is
    procedure home is
    begin
        x_coord:=0;
        y_coord:=0;
        ...
    end home;
```

```
begin
    home;
end cursor;
```

为程序包体。这样，现在就有分别编译的编译单元了。它们一起组成程序库。“主程序”和程序包规格说明是库单元，至于程序包体，如果分别编译则是声明它的程序单元（在这种情况下，此单元包括程序包规范说明）的子单元。注意，若要分别编译程序包规格说明和程序包体现在的这个形式还要作进一步的修改（见10.2节）。

10.1.1 带有子句

with（带有）子句的确立，使由它指名的库单元在*STANDARD*程序包声明之后立即隐含地得到声明。这样，被指名的单元就直接可见，除非他们有某种掩蔽机制。当我们将原有程序分解为单独的编译单元时，这样就克服了主要的障碍。亦即某个单元内的标识符能被其他单元可见的问题。我们若注意例中的主程序，就可看出：为了在使用子句中引用程序包“*cursor*”及标识符“*home*”(它在“*cursor*”中定义），加上**with**子句是必需的。

注意，任何给定的程序单元仅仅指**with**子句中直接要求的单元。这样，“*screen_io*”只在它的**with**子句中指明“*cursor*”而不是指“*cursor*”所要求的其他单元。

10.2 子单元

在*Ada*中，可将子程序、程序包、任务的体从声明它的单元中分离出来。这就使声明单元产生一个“子单元”，它

的语法是：

子单元 ::=**separate**（单元_名）体

为了完善子单元和声明单元之间的联接，将“体存根”（*body stub*）置于声明单元中体通常要出现的地方。其语法是：

体_存根 ::=
　　子程序_规格说明 **is separate**;
　|**package body** 标识符 **is separate**;
　|**task body** 标识符 **is separate**;

为了说明这点，我们改写原例题，将程序包“*cursor*”的程序包体做成子单元，如下：

```
procedure screen_io is
    package cursor is
        x_coord:integer range 0..79;
        y_coord:integer range 0..23;
        procedure home;
        ...
    end cursor;
    package body cursor is separate;
    procedure move(x,y:integer) is
        use cursor;
    begin
        x_coord:=x;
        y_coord:=y;
        ...
    end move;
begin
    cursor.home;
```

```
    ...
end screen;
```

为主程序，以及

```
separate (screen_io);
package body cursor is
    procedure home is
    begin
        x_coord:=0;
        y_coord:=0;
        ...
    end home;
begin
    home;
end cursor;
```

关于这个例子有几点要注意，父单元必须在体存根中提及子单元，而子单元也要类似地提及它所从属的父单元。此外，如果父单元本身也是子单元，则在子单元中必须给出它的全名（象选择成分一样）。

和你预期的一样，为了避免混淆，*Ada* 要求单个库单元中所有子单元的名字要有区别。然而，同一程序库中不同库单元的子单元有同样的名字都是完全可以接受的，因为全名不同。例如，我们可以有两个子单元都叫“*page_up*”，一个子单元是库单元“*crt_io*”的，另一个是库单元“*printer_io*”的。它们的全名“*crt_io.page_up*”和“*printer_io.page_up*”是有区别的。

从可见性观点，由声明单元的体存根中可见的任何实体对子单元也是可见的。于是，声明单元可见的任何库单元，

经由with子句对子单元也可见。与此类似，若子单元的上下文规格说明中提到任何库单元，则它们在子单元内部均直接可见。

10.3 标识符的作用域和可见性

我们已经研究到编译单元之间标识符可见性的问题了，现在把注意力集中到这些单元内部标识符可见性的控制规则。

10.3.1 标识符的作用域

声明的目的是向*Ada*编译程序传递有关被声明实体（对象、类型、子程序等等)的信息。这些信息中的一部分是标识符，靠了这些标识符程序正文的另一部分才可以引用给定的实体。本章剩下部分我们将考察声明的“作用域”和“可见性”等概念。

声明的“作用域”定义为声明在程序正文的有效范围。反过来说，如果在某一特定点能使用标识符来引用该实体，则说声明从该点可见。为了说明这些，考虑如下例子：

```
outer:
    declare
        a,b,c:integer;
    begin
        ...
        a:=b;
        ...
inner:
        declare
```

```
        x,y,z:integer;
    begin
        ...
        x:=y;
        z:=a;
        ...
    end inner;
    ...
    c:=z; -- 非法
    ...
end outer;
```

"a"、"b"和"c"声明的作用域是从他们被声明那点到"outer"块结束，而"x"、"y"和"z"是从他们被声明的那点到"inner"块结束。因此，可以看出赋值语句 c:=z 是非法的，因为对象"z"不在作用域之内。

为了给出比较正规的规则，摘录Ada程序设计语言参考手册定义的作用域规则如下：

（a） 块的声明部分或子程序、程序包、任务的体的声明部分给出的声明，其作用域从声明处伸展到块、子程序、程序包、任务的结束处。

（b） 程序包可见部分给出的声明其作用域从声明处伸展至程序包声明本身作用域的末端。因此，它包括相应的程序包体。

（c） 程序包私有部分给出的声明其作用域从声明处伸展至程序包规格说明结束处；它还能伸展到整个相应的程序包体。

（d） 在任务规格说明中给出的入口声明，其作用域从声明处伸展到任务声明的作用域的末端。因而它包括相应的

任务体。

（*e*） 分别编译的子程序或程序包，除子单元外，其作用域包括本编译单元，它的子单元（如果有的话），任何在带有子句中提到本子程序或程序包名字的其他编译单元，以及这些子程序或程序包的体。

（*f*） 记录成分的作用域从成分声明起，伸展到记录类型声明的作用域的末端。

（*g*） 判别式的作用域从判别式声明起，伸展到相应类型声明的作用域的末端。

（*h*） 子程序、入口、类属程序单元的形式参数其作用域从参数声明起，伸展到子程序、入口、类属程序单元本身声明的作用域的末端。因此，它包括相应的子程序或类属程序单元的体，对于入口则包括相应的接受语句。

（*i*） 循环参数的作用域从它在迭代子句产生处，伸展到相应循环的结束处。

（*j*） 枚举字面量的作用域从它在相应的枚举类型声明中产生起，到枚举类型声明本身作用域的末端。

为了说明其中的某些规则，考虑下面给出的例子：

```
outer:
    declare
        a,b,c:integer;          --a,b,c 作用域开始
    begin
        ...
        a:=b;
        ...
inner:
        declare
```

```
        x,y,z:integer;      --x,y,z作用域开始
    begin
        ...
        x:=y;
        z:=a;
        c:=z;
        ...
    end inner;              --x,y,z作用域结束
    ...
    c:=a;
    ...
end outer;                  --a,b,c作用域结束
procedure x(a:in out integer)is--a 作用域开始
    temp:integer:           --temp作用域开始
    ...
begin
    ...
    for i in 1..10 loop     --i 作用域开始
        ...
    end loop;               --i 作用域结束
    ...
end x;                      --a,temp作用域结束
package complex_addition is
    type complex is private; --complex 作用
                             --域开始
    function "+"  (a,b:complex)
                  return complex;
    ...
private
    type complex is
        record
```

```
            re:float:=0.0;  --re作用域开始
            im:float:=0.0;  --im作用域开始
        end record;
end complex_addition;
package body complex_addition is
    ...
end complex_addition;           --re, im 作用域
                                --结束
procedure xyz is
    use complex_addition;       --complex 作用
                                --域开始,注意re
                                --和 im 不在此
                                --作用域内
begin
    ...
end xyz                         --complex 作用
                                --域结束
```

作用域和可见性主要的差别在于：标识符在其作用域以内的任何点都潜在地可见，但在某个特定点上它是否实际可见要取决于其他的几个因素，这是在下节中要研究的。

10.3.2 标识符的可见性

现在把我们的注意力转到讨论可见性及 *Ada*的掩蔽规则上。在作用域内给定一个标识符，这些规则可决定在程序正文的某个点上是否可见，或它是否被掩蔽。

介绍掩蔽概念最好的办法是举例：

```
outer:
    declare
```

```
        a,b,c:integer;
    begin
        ...
        a:=1;
        ...
inner:
        declare
            x,y,z:integer;
            a:integer:=-1;
        begin
            ...
            z:=a;
            ...
        end inner;
        ...
    end outer;
```

从程序正文观察，不能清楚地看出语句z:=a; 执行后是什么结果，问题在于它使用叫做“a”的两个实体中的哪一个，赋值语句之后 z 的值是+1还是-1。

这个问题的解决要用到标识符的“掩蔽”概念。当标识符出现冲突时，如象我们例题中的情况，掩蔽规则规定：内层块中的对象声明若与外层块中的对象有同一标识符时，就直接可见而论内层标识符“掩蔽”了外层标识符。于是，本例中执行z:=a; 之后， z 值为-1。

再次引用*Ada*程序设计语言参考手册，可见性规则按手册上的规定是：

(a) 一个标识符如在块的声明部分或子程序、程序包、任务的体的声明部分中声明，则在块或体的内部直接可

见。

(*b*) 一个标识符在程序包的可见部分中声明，则程序包规格说明和包体内均直接可见。

若一标识符在程序包外，但在其作用域之内，可由前缀写上程序包名的选择成分使其成为可见。通过使用子句该标识符也成为直接可见。

(*c*) 一个标识符在程序包的私有部分中声明，则在程序包私有部分和包体内部直接可见。

(*d*) 一个（入口）标识符在任务规格说明中声明，则在任务规格说明和任务体内部直接可见。

在任务之外，但在其作用域之内，此标识符可由一选择成分使其成为可见，该选择成分的前缀给出了任务或任务类型的任务对象的名字。

(*e*) 分别编译的子程序或程序包的标识符，在各自的编译单元和它的子单元内部直接可见。用带有子句提及本标识符的其他编译单元的内部也直接可见。

(*f*) 记录成分的标识符在声明这个成分的记录类型定义中直接可见。并在此记录类型的表示法规格说明中直接可见。

在记录类型定义之外，但在该定义的作用域之内记录成分可由一个选择成分使其成为可见，该选择成分的前缀给出了成分类型的记录名。在记录类型聚集的成分结合中作为一个选择也是可见的。

(*g*) 判别式的标识符在声明本判别式的判别式部分之内以及被结合的记录类型类型定义之内直接可见。

在不能直接可见的地方，但如在该类型的作用域之内，

将判别式置于一选择成分或聚集之内（如同对其他任何记录成分所做的那样），则判别式成为可见。在判别式约束中以指名判别式规格说明所指定的判别式名在该约束中也是可见的。

（*h*） 子程序形式参数的标识符在声明这些参数的形参部分中，以及在子程序体之内是直接可见的。入口形式参数的标识符在声明这些参数的形参部分中，以及与此入口对应的接受语句之中是直接可见的。类属形式参数的标识符在声明这些参数的类属部分中，以及类属子程序或程序包的规格说明或体中直接可见。

（*i*） 循环参数的标识符在声明它的循环中直接可见。

（*j*） 在声明枚举字面量的枚举类型作用域之中，枚举字面量直接可见。

在某些情况下，希望能够访问掩蔽了的标识符。要注意，上述指明的规则仅适用于直接的可见性，亦即单独使用标识符去访问所希望的对象。掩蔽的标识符通常用限定词来访问，多半是选择成分或判别式名。

下面几个例子用来说明上述规则的某些规定。

```
procedure matrix_invert is
    a,b:matrix;
    procedure matrix_multiply is
        b,c:matrix;
    begin
        ...
        b:=c;
        ...
```

```
        c:=matrix_invert.b; --使用选择成分访问
                            --掩蔽的标识符
        ...
    end matrix_multiply;
begin
    ...
end matrix_invert;
procedure a is
    package short_vectors is
        type short_vec is...;
        ...
    end short_vectors;
    package long_vectors is
        type long_vec is…;
        ...
    end long_vectors;
    use short_vectors;          --使short_vec直接
                                --可见
    x:short_vec;                --现在可以只用这个
                                --标识符
    y:long_vectors.long_vec;    --这里，标识符不是
                                --直接可见的，故必
                                --须用选择成分
begin
    ...
end a;
--第一编译单元：
package misc is
    procedure init;
    ...
end misc;
```

```
package body misc is
    procedure init is...end init;
    ...
end misc;
--第二编译单元:
with misc;                          --使misc的标识符
                                    --直接可见
procedure main_prog is
begin
    init;                           --现在可 单个地用
                                    --misc中的标识符
    ...
end main_prog;
```

10.3.3 标识符重载

到目前为止在所有课文和例子中，我们都没有考虑标识符使用重载的可能性。标识符的重载隐喻着同一个标识符可以引用好几个实体。这似乎和可见性规则有矛盾。

对于子程序，仅当参数的次序、名字、类型都相同（对于函数，其结果值的类型也应相同），有缺省值的参数也相同（见第5章，5.6节），则说此子程序声明掩蔽了先前的相同标识符的子程序声明。若这些准则不能满足，则第二个声明重载，但不能掩蔽第一个。以类似的方式，当枚举字面量的作用域覆盖时它们也可以重载。随后使用限定表达式来解决这种混淆。

如果由于使用use子句造成表达式中出现重载，又无别的办法来唯一确定所要引用的实体时，则只考虑由use子句引入的标识符来解决这个冲突。

10.4 换名声明

换名声明是用来为一个实体引入一个附加的名字。老的名字不受新名字的影响继续有效。换名声明的语法是：

```
换名_声明 ::=
      标识符：类型_标记 renames 名字；
    | 标识符： exception renames 名字；
    | package 标识符 renames 名字；
    | task 标识符 renames 名字；
    | 子程序_规格说明 renames 名字；
```

在上面给出的所有不同类型的换名中，换名声明和原有声明的类型必须兼容。例如，下面的例子就是不合法的：

```
declare
    a_very_long_name_to_type:integer;
    avlntt:float renames
        a_very_long_name_to_type;
    --非法，其类型不一致
begin
    ...
end;
```

注意，对于给子程序换名的换名声明有一个明显的约束，即必须从名字和规格说明能唯一地决定某个子程序。如果有多于一个可见的子程序其规格说明和标识符能与换名声明相匹配，则此换名声明是非法的。

第十一章　输入—输出

在最后一章里，我们将讨论*Ada*的输入—输出设施，与*Algol*-60不同，它是语言定义的组成部分。

11.1　概　述

*Ada*提供了两个程序包以协助实施输入—输出功能。*INPUT_OUTPUT*程序包用于对文件的高级输入／输出。*TEXT-IO*程序包用于较为基本的操作。

11.2　文件和文件名

*Ada*有两类文件，内部文件和外部文件。内部文件是一个被声明的对象，它是一个名字以及与名字相结合的文件_类型。文件类型*IN_FILE*，*INOUT_FILE*或*OUT_FILE*用于为指名文件规定数据传递的类别（和子程序参数模式大致相同的方法），提供常规的*READ*、*READ／WRITE*和*WRITE*访问机制。外部文件是信息实际驻留的所在，它是由象设备名、部件号等等之类与实现有关的信息来指明的。这些信息按实现要求以正文串形式纳于外部文件名下。内部文件和外部文件之间的联系由对*CREATE*或*OPEN*过程的调用来建立。

元素的序列组成文件，所有元素都是同一类型。由于各种不同类型的存在，文件的输入／输出要靠叫做

*INPUT_OUTPUT*的类属程序包的支持，对于每种要用的元素类型都要求一个程序包的类属设置。一旦设置发生，则用户就可用该元素类型来声明内部文件，同时能使用设置提供的子程序去处理这些文件。为了说明：

```
declare
    package integer_io is new
        INPUT_OUTPUT (integer) ;
    package   float_io is new
        INPUT_OUTPUT (   float) ;
    from_file:integer_io.in_file;     --注意文件
                                      --类型
        to_file:float_io.out_file;
    i : integer;
    f : float;
begin
    ...
    integer_io.OPEN(file=>from_file,
                name=>"SY:OLD.DAT;1");
    float_io.CREAT(file=>to_file,
                name=>"SY:NEW.DAT;1");
    ...
    integer_io.READ (from_file, i) ;
    f:=float(i);
    float_io.WRITE(to_file, f) ;
    ...
        float_io.CLOSE(to_file);
    integer_io.CLOSE(from_file);
end;
```

11.2.1 文件的开启、关闭和测试

上例中，对外部文件访问之前，必须建立内部文件名和它之间的联系，这种联系是靠调用*OPEN* 或 *CREATE*两个过程之一来实现的。倒底用哪一个要看外部文件是否已经存在。这两个过程的规格说明是：

```
procedure CREATE (file:in out out_file;
                  name:in      string);
procedure CREATE (file:in out inout_file;
                  name:in      string );
procedure OPEN(file:in out in_file;
               name:in     string);
procedure OPEN( file:in out out_file;
                name:in      string);
procedure OPEN( file:in out inout_file;
                name:in      string) ;
```

对*CREATE*过程的调用则导致一个新的外部文件建立，同时产生了新外部文件和指名的内部文件之间的联接。调用成功地返回之后，则外部文件就和内部文件接上了。当这个联接存在时就说该内部文件“开启”了。如果由于某种原因外部文件不能建立，则 *NAME_ERROR* 异常将被引发。如果内部文件已经开启则 *STATUS_ERROR* 异常将被引发。注意用于调用*CREATE*的内部文件的文件_类型，显而易见不是 *OUT_FILE* 就是 *INOUT_FILE*。因为试图从一个没有数据的文件中去读数是毫无意义的。

对*OPEN*的调用则产生对一个现存外部文件的联接（不是建立新文件），于是内部文件就“开启”了。如果外

部文件不存在，则 $NAME\_ERROR$ 异常将被引发。若内部文件已经开启则 $STATUS\_ERROR$ 异常被引发。这种调用，对内部文件的文件_类型没有任何限制。

如果在程序员试图用 $CREATE$ 或 $OPEN$ 去打开一个文件之前，查看一下一个给定的内部文件是否已经打开，这将会很有用。函数 IS_OPEN 可以做到这点，它的规格说明是：

```
function IS_OPEN(file:in  in_file)return
    boolean;
function IS_OPEN(file:in out_file)return
    boolean;
function IS_OPEN(file:in inout_file)return
    boolean;
```

用一个内部文件名作参数来调用本函数，若内部文件是开启的（即已与外部文件结合了）将返回一个布尔 $TRUE$ 值，否则是 $FALSE$ 值。于是，我们可以写为：

```
declare
    package x_io is new input_output(x);
    use x_io;
    my_file:x_io.in_file;
    ...
begin
    ...
    if is_open(my_file) then
    ...
    else
        open(my_file, "MT:" );
```

```
        end if;
        ...
    end;
```

当程序员处理完一个文件，内部和外部文件之间的联接就可以断开。这可由以内部文件名作参数调用 *CLOSE* 过程来完成。*CLOSE*的规格说明是：

```
procedure CLOSE(file:in out     in_file);
procedure CLOSE(file:in out    out_file);
procedure CLOSE(file:in out inout_file);
```

如果内部文件已经关闭 则 *STATUS_ERROR* 异常被引发。

在*INPUT_OUTPUT*程序包中还有两个处理外部文件的子程序，*NAME*和*DELETE*。函数 *NAME* 的作用是返回一个表示外部文件的字符串。该文件已被联接在作为参数给出的内部文件之上了。*NAME*的规范说明是：

```
function NAME(file:in     in_file)return
                                  string;
function NAME(file:in    out_file)return
                                  string;
function NAME(file:in inout_file)return
                                  string;
```

注意，返回的字符串显然要取决于实现，如果作为参数传递的内部文件已经关闭，则本函数将引发异常 *STATUS_ERROR*。

*DELETE*过程用于删除一个外部文件。它的规格说明是：

procedure *DELETE*(*name*:**in** *string*);

调用*DELETE*不能保证指名文件立即被删除，而是不能再对此文件实施*OPEN*操作。这种情况是不难设想的：一个单一的外部文件可以联接到几个内部文件上。在这些情况下，在该文件实际被删除之前所有的联接均应断开。如果文件因某种理由（如，它不存在，它被保护等等）不能删除，则引发*NAME_ERROR*异常。

11.3 用文件输入——输出

上节我们介绍了一些机制，用这些机制我们可以命名文件，建立语言和实际存贮介质之间的必要联系。本节我们将研究从文件中读入和写出的设施，以及在文件内部定位。注意本节中的过程和函数规格说明仅给出文件类型参数*IN_FILE*和*OUT_FILE*，因为对于每个能取*IN_FILE*参数的子程序都是能取*INOUT_FILE*参数的子程序，同样*OUT_FILE*也如是。

按上面讲述，文件可看成是元素的序列，每一元素都有隶属于依赖实现的*FILE_INDEX*类型的顺序号与之结合（文件中所有的位置都可用隶属于该类型的整数来指示）。注意，并非所有元素都要定义顺序号。比如，在索引文件中有没有顺序号对它说来不是关键。

每个开启的内部文件都有一个可供读/写（要取决于文件_类型）的“当前位置”，它指示的元素就是下次读或写操作要用到的元素。当文件第一次打开时，当前位置置于文件的“起始”位置。对于读入就意味着其当前位置置1（或者是，如果没有已定义元素与此顺序号相应，则当前位置就是第一

个被定义元素的位置）；对于写出，当前位置也置 1 。

除了当前位置而外，文件还有两种性质，即“当前尺寸”和“结束位置”。当前尺寸就是文件中已定义元素的个数，而结束位置是最后被定义元素的顺序号。如果文件沒有被定义的元素，二者皆为零。

有两个过程 *READ*和*WRITE* 使向或自文件传递信息生效：

```
procedure READ( file: in      in-file ;
                item:out element_type);
procedure WRITE( file:in      out_file ;
                 item:in element_type);
```

READ 过程在“*item*”参数中返回一个在当前读位置上的元素值，并改置当前读位置为下一被定义元素的位置。如果沒有那个元素，位置仍然增加 1 。若试图去读一个未定义的元素，则*DATA_ERROR*异常将引发。同样，试图去读文件结束处以外的元素（当前读位置大于结束位置）将引发 *END_ERROR*异常。

与其相伴的*WRITE* 过程将参数“*item*”的值写在文件的当前写位置上，而后当前写位置增值。如果写是写在未定义的元素上，于是文件被定义元素的个数就增加，则当前文件尺寸也增值。若当前写位置在写之前就大于结束位置，则用此当前写位置的值来更新结束位置。

我们用一个例子将前几段的内容连贯在一起，该例读一个整数文件，读数平方后将其结果写入一个新文件：

```
declare
    package my_io is new input_output(integer);
```

```
    use my_io;
    destination:out_file;
        source :in_file;
        temp   :integer;
begin
    open(source,"/usr/source/integers");
    create(destination,"/usr/dest/integers");
    read(source,temp);
    temp:=temp*temp;
    write(destination,temp);
    close(source);
    close(destination);
end;
```

如例中所写，本例仅用于对源文件的第一个元素作平方。要使它真正有用，我们必须能对所有被定义元素作循环。有几个函数在这方面有助于我们。第一个是测试文件_结束的函数：

```
function END_OF_FILE(file:in in_file)
        return boolean;
```

如果给定文件的当前读位置大于它的结束位置，则本函数返回布尔“真”值，我们改写上例如下：

```
declare
    package my_io is new input_output(integer);
    use my_io;
    destination:out_file;
        source :in_file;
          temp:integer;
begin
    open(source,"/usr/source/integers");
```

```
        create(destination,"/usr/dest/integers");
        while not end_of_file(source)loop
            read(source,temp);
            temp:=temp*temp;
            write(destination,temp);
        end loop;
        close(source);
        close(destination);
    end;
```

其余的函数允许程序员询问有关文件当前的状态，它们是：

```
function SIZE(file:in in_file)return file_index;
function SIZE(file:in out_file)return
                                    file_index;
```

这两个函数可以使我们求出文件中被定义元素的个数。

```
function LAST(file:in in_file)return file_index;
function LAST(file:in out_file)return
                                    file_index;
```

这两个函数返回结束位置的顺序号。最后两个函数是*NEXT_READ*和*NEXT_WRITE*：

```
function NEXT_READ(file:in in_file)return
                                    file_index;
function NEXT_WRITE(file:in out_file) return
                                    file_index;
```

它们返回当前读或写位置的顺序号。

*Ada*提供了通过改变一个文件当前位置和尺寸以及结束位置的值去操纵文件的能力。以下过程实施这些动作，它们

是：

```
procedure SET_READ  (file:in in_file;
                     to:in file_index);
procedure SET_WRITE(file:in out_file;
                    to:in file_index) ;
```

由指定的“*to*”参数来设置当前读位置或当前写位置的值。将值设置在文件起始位置的特定情况（类似于*Fortran*的*REWIND*），可由以下过程处理：

```
procedure RESET_READ(file:in in_file);
```

它使当前读位置从第一个被定义的元素开始，若无被定义元素则从1开始，此外

```
procedure RESET_WRITE(file:in out_file);
```

使当前写位置从1开始。最后一个操纵件文的过程是：

```
procedure TRUNCATE(file:in out_file;
                   to:in file_index);
```

它将件文结束位置置为一个指定值。如果必要，文件的尺寸也可以修改。注意，本过程不能用来扩充文件，这是因为若“*to*”参数值大于当前结束位置的值，则将引发*USE_ERROR*异常。

在转入讨论正文_输入/出之前要提醒一句。程序员应该意识到一个输入/出操作有可能不能完成，或许是因为物理设备不支持这种操作。比如，试图写入键盘或是访问一个脱机设备。在这些情况下将引发*DEVICE_ERROR*异常。

11.4 程序包TEXT_IO

上几节研究了文件的输入/出，并没有提及数据的外部表示。然而，程序包*TEXT_IO*支持可读形式的输入/出，履行内部表示到外部表示之间必要的转换。*INPUT_OUTPUT*程序包提供的功能和*TEXT_IO*提供的功能，其差别类似于*Fortran*中“无格式的”和“格式的”输入/出之间的差别。

*TEXT_IO*程序包由四个类属程序包和几个子程序组成。一个整型的类属程序包*INTEGER_IO*，一个浮点输入/出的类属程序包*FLOAT_IO*，一个定点输入/出的类属程序包*FIXED_IO*，以及枚举类型输入/出的类属程序包*ENUMERATION_IO*。此外，还有几个涉及字符处理、串处理、布尔型输入/出、格式化功能以及文件操作的子程序。程序包中大量的动作是靠重载子程序*GET*和*PUT*来完成的。

11.4.1 标准输入和标准输出

*INPUT_OUTPUT*程序包要求每个子程序都作为参数传递到内部文件。*TEXT_IO*程序包则与此不同，它应用了缺省的概念，输入是一个缺省的源文件，输出是一个缺省的目的文件。程序一开始就把这两个与实现有关的文件联接上了，并处于开启的初态。这就允许用不需要显式声明文件的短格式的*GET*和*PUT*过程，当然，长格式的也可以使用。因此，对于给定类型的“*x*”，*GET*过程有两种形式：

procedure *GET*(*item*:**out** *x*);

```
procedure GET(file :in in_file;
              item:out     x);
```

*PUT*过程也有类似的两种形式，长格式带有文件参数，而短格式没有。本章的后文中，各子程序规格说明只给出短格式，若用长格式则假定只取合适的文件类型（对*GET*取*IN_FILE*，*PUT*取*OUT_FILE*等等）作文件参数。

两种程序包还有一个差别，即*INPUT_OUTPUT*程序包所有的文件类型都可用，相反*TEXT_IO*只能用*IN_FILE*和*OUT_FILE*两种文件类型。将*INOUT_FILE*文件类型用于*TEXT_IO*的子程序，其结果无定义。

为了支持缺省文件的设施，*TEXT_IO*提供以下子程序，以便缺省文件环境的操作：

```
function STANDARD_INPUT return in_file;
function STANDARD_OUTPUT return
                                  out_file;
```

它们分别返回对缺省的输入或缺省输出文件指派的初值，下面两个函数：

```
function CURRENT_INPUT return in_file;
function CURRENT_OUTPUT return out_file;
```

则返回缺省输入和缺省输出文件的当前背景情况，而过程：

```
procedure SET_INPUT  (file:in in_file);
procedure SET_OUTPUT(file:in out_file);
```

可用来改变缺省输入和缺省输入的背景情况。

11.4.2 正文文件格式

在我们深入讨论不同型式输入/出设施之前，有必要了解*Ada*有关正文文件格式的规则。

在文件输入/出那节里，我们读到，*READ*和*WRITE*都有一个它们动作的当前位置，*GET*和*PUT*也一样，有它们动作的当前位置。对于正文文件，当前位置是根据“当前行号”和“当前列号”来指明的。二者的值域都是从1往上数。至于行的长度并没有什么限制（除了它必须小于或等于文件总长而外），也不需要把整个文件的行固定为某个长度。程序员在一个文件里可以自由混用长短不同的行。下面的子程序得以支持格式控制：

```
function    COL return natural;
procedure SET_COL(to:in natural);
```

它们可以得到或设置当前列号，

```
function LINE return natural;
```

它返回当前行号，

```
function LINE_LENGTH return integer;
procedure SET_LINE_LENGTH(n:in integer);
```

它们返回或设置当前行长度（行长度为零是指未设置行长度，即文件开启时的初始条件），

```
procedure NEW_LINE(spacing:in natural:=1);
```

它终止当前行（如果必要将当前行充以空白），改置当前列号为1并加几个“*spacing*”（空）行（注意，其缺省值

为 1)。此过程仅对缺省的输出文行或类型为OUT_FILE的文件是合法的。对于缺省输入文件(或称IN_FILE文件)还有一补充过程：

```
procedure SKIP_FILE(spacing:in natural:=1);
```

最后，对缺省输入文件（或IN_FILE文件）还有：

```
function END_OF_LINE return boolean;
```

如果当前行上已无要读的字符，则返回一个布尔“真”值。用来说明这些子程序使用的几个例子是：

```
procedure main is
    use text_io;
    old_input, new_input:in_file;
    ...
begin
    ...
    old_input:=standard_input ( ) ;
    set_input(new_input);
    ...
end main;
procedure test_page(n:in natural)is
    use text_io;
    page_size:constant natural:=52;
begin
    if(line()+n)>page_size
    then
        -- 页起头，打印题头并重置计数器
    end if;
end test_page;
```

11.4.3 整型数输入／出

整型数的输入和输出是由类属程序包*INTEGER_IO*为此类型建立的示例来完成的，如果我们希望用“自然数(*natural*)”类型来工作，则设置为：

```
package nat_io is new integer_io(natural);
```

它就为我们提供了“自然数”类型的*GET*和*PUT*过程。这两个过程短格式的子程序规格说明是：

```
procedure GET(item:out i_type);
procedure PUT(item:in i_type;
              width:in integer:=0;
              base:in integer range 2..16:=10);
```

*GET*过程将跳过前导的格式字符诸如制表符、空格、行_标记等，然后就从输入文件读入(对于短格式是缺省输入文件，对长格式是*in_file*参数的文件)，执行从正文到所要求类型的转换，并将转换的结果值置于“*item*”参数。如所预期，单个的正负号字符＋或－先于数字处理。如果项(*item*)参数放不下最后转换的值，则*CONSTRAINT_ERROR*异常将被引发。

*PUT*过程实施相反的功能，将“*item*”参数的内部值转换成字符序列并将它们写入输出文件。

可选的“*width*（宽度）”参数用来指出程序员所希望的此子程序生成的最小字符个数。如果宽度参数太小而不能表示“*item*”中的值，则跳过此项。若宽度大于需要，则前面加以空格。

“*base*（基）”参数也是可选的，其缺省值为10。如果用

了基则按语法（见附录A）输出带基的数，其形式为：

基#数#幂

输出转换的一些例子是：

```
put(5);                    -- "5"
put(item=>5,
    width=>3);             -- "5"
put(item=>5,
    width=>7,
    base=>2);              -- "2#101#"
put(-5);                   -- "-5"（此例对"自然
                           --数"类型是非法的）
```

11.4.4 浮点数输入／出

浮点数输入/出靠类属程序包$FLOAT_IO$的示例来支持。过程的工作方式非常类似用于整型的过程。它们的规格说明是：

```
procedure GET(item:out fl_type);
procedure PUT  (item     :in fl_type;
                width    :in integer:=0;
                mantissa :in integer:=
                          fl_type'digits;
                exponent :in integer:=2);
```

$item$和$width$参数实施如前述同样的功能。浮点数按"-1.23456E-78"的形式输出，以可选的"$mantissa$（尾数）"参数值来控制该数的尾数，即整个数字的位数（不包括可能有的前导"-"号和"."）。"$exponent$（幂）"

参数提供对该数乘幂位数的类似的控制（不包括“E”和可能的“-”号）。浮点数输入/出的一些例子是：

```
declare
    type fl_type is digits 8;
    package fl_io is new float_io(fl_type);
    use fl_io;
    pi:fl_type:=3.141593;
begin
    put(pi);                    --"3.1415930E00"
    pi:=pi/10.0;
    put(item=>pi,
        mantissa=>4,            --注意省略了一个参数
        exponent=>2),           -- "3.142E-01" 注意
                                --舍入
    put(item=>pi,
        width=>10,
        mantissa=>4);           -- "  3.142E-01"
    ...
end;
```

此例有两点值得研究。第一，要注意，若指定的“尾数”参数少于用以表示此类型的位数，则将施行圆整化。第二，若使用“幂”参数且它提供的位数不足，则不考虑给出的值而用实际需要的位数。

11.4.5 定点数输入／出

最后一个数值输入/出程序包是支持定点类型输入/出的类属程序包$FIXED_IO$。类属设置仍将提供子程序：

```
procedure GET(item :out fi_type);
procedure PUT(item :in  fi_type;
              width:in  integer:=0;
              fract :in  integer:=
                  default_decimals);
```

用“*fract*（小数）”参数来指定小数点后的位数。如果必需，输入和输出两者都作四舍五入。定点数输入/出的一些例子是：

```
declare
    type fi_type is delta 0.001;
    package fi_io is new fixed_io(fi_type);
    use fi_io;
    score:fi_type:=1.248;
begin
    put(score);                          -- "1.248"
    put(item=>score,
        width=>6);                       -- "  1.248"
    put(item=>score,
        width=>  6,
        fract=>  2);                     -- "  1.25"
    put(item=>score,
        width=>  8,
        fract=>  5);                     -- "  1.24800"
end;
```

11.4.6 枚举型输入／出

类属程序包*ENUMERATION_IO*的类属设置提供了为此类型输入/出设施常用的一对子程序。它们短格式的

子程序规格说明是：

```
procedure GET(item:out e_type);
procedure PUT(item:in e_type;
              width:in integer:=0;
              lower_case:in boolean:=false);
```

输入时，*GET*对大写和小写字符不加区分。如果有一个枚举字面量没有匹配，则将引发*DATA_ERROR*异常。输出时，*PUT*用参数“*lower_case*（小写）”去指定枚举字面量应写成哪种字样。如果用了“*width*”参数，且比需要提供了更多的字符位值，则尾部充空格。枚举型输入/出的一些例子是：

```
declar
    type valve_position is (closed,
                            travelling, open);
    package valve_io is new enumeration_io
                           (valve_position);
    use valve_io;
    var:valve_position:=open;
begin
    put(var);                          -- "OPEN"
    put(item=>var,
        lower_case=>true);             -- "open"
    put(item=>var,
        width=>5);                     -- "OPEN  "
end;
```

11.4.7 布尔型输入／出

布尔型实则是枚举型的特例，所以它的输入/出子程序规格说明没有什么新花样，与枚举类型的规格说明不同之处仅仅是“*item*”参数的类型：

```
procedure GET(item:out boolean);
procedure PUT(item:in boolean;
              width:in integer:=0;
              lower_case:in boolean:=false);
```

然而，要注意，它不能进行类属设置。布尔型输入／出作为枚举型输入/出看待，只不过用了字面量*TRUE*和*FALSE*。

11.4.8 字符型输入／出

字符型输入/出利用了当前行号和当前列号的概念，在前面11.4.2节中已介绍。短格式过程的子程序规格说明是：

```
procedure GET(item:out character);
procedure PUT(item:in character);
```

调用*GET*将返回当前位置上的字符值，同时，位置向后增值（如果文件的行长度是固定的，则将当前位置设置到下一行的开始）。

调用*PUT*过程将把“*item*”参数中的字符写到当前位置上，当前位置则向后增值。若文件有固定长度的行，则增值处理还包括在当前行末端附加一个行标记，同时置当前位置到新的一行开始。

注意，和布尔型的*GET*和*PUT*一样，这些过程不是类属程序包的一部分。

11.4.9 串型输入／出

*Ada*除了提供字符型输入/出的基本功能之外，还提供串和行的输入/出功能。下述两个（计及长格式是四个）过程：

```
procedure GET(item:out string);
procedure PUT(item:in string);
```

读入或写出“*item*”参数。对于*GET*，跳过前导的格式字符读入字符串，直到（但不包括）下一个空白字符。

*GET*的另一种形式是以函数形式提供的：

```
function GET_STRING return string;
```

这样就允许直接和输入串联系而无需使用中间变量。*Ada*还提供了*GET_STRING*的特例：

```
function GET_LINE return string;
```

这个函数返回以串表示的从当前位置直到（但不包括）下一个行_标记的所有字符。调用之后，当前位置在下一行的起始位置。

还有一个补充的操作是写一整行，这由下述子程序提供：

```
procedure PUT_LINE(item:in string);
```

它从“*item*”参数中取出串将它写在输出文件上，并于末端加上行标记。

附录A　Ada语法定义

（按汉字笔划序）

十进制_数::＝整数［，整数］［冪］

入口_声明::＝

　　entry标识符［（离散_范围）］［形参_部分］；

入口_调用::＝入口_名［实参_部分］；

上下文_规格說明::＝{ 协同_子句［使用_子句］ }

子类型_指示::＝类型_标記［約束］

子程序_声明::＝子程序_规格說明；

　　| 类属_子程序_声明；

　　| 类属_子程序_設置；

子程序_体::＝子程序_规格說明 **is**

　　声明_部分

　　begin

　　語句_序列

　　[exception

　　{ 异常_处理段 }]

　　end［指明符］；

子程序_规格說明::＝

　　procedure 标识符［形参_部分］

　| **funetion**指名符［形参_部分］**return**子类型_指示

夭折_語句::＝**abort** 任务_名{，任务_名}；

引发_語句::=**raise** [异常_名] ;
分配_算符::=**new**类型_标記 [(表达式)]
| **new**类型_标記 聚集
| **new**类型_标記 判别式_約束
| **new**类型_标記 序标_約束
不完全_类型_声明::=
type 标识符 [判别式_部分] ;
长度_规格說明::=**for**属性**use**表达式;
出口_語句::=**exit** [循环名] [**when**条件] ;
代碼_語句::=限定_表达式;
对象_声明::=
标识符_表 : [**constant**]子类型_指示[:=表达式];
| 标识符_表 : [**constant**]数組_类型_定义
[:=表达式] ;
記录_类型_表示法::=
for 类型_名 **use**
record [准綫_子句;]
{ 成分_名 定位; }
end record;
記录_类型_定义::=
record
成分_表
end record;
字母::=大写_字母|小写_字母
字面量::=数值_字面量|枚举_字面量
|字符_字面量|**null**

```
字符_串::= "{字符}"
片::=名字（离散_范围）
加法类_运算符::=+ | — |&
扩展_数字::=数字 | 字母
名字::=标识符      | 序标_成分
      |片          | 选择_成分
      |属性        | 函数_调用
      |运算符_符号
自变量::= [标识符=>] 名字
    | [标识符=>] 静态_表达式
任务_体::=
    task body标识符 is
        [声明_部分]
    begin
        語句_序列
   [exception
        {异常_处理段}]
    end [标识符];
任务_声明::=任务_规格說明
任务_规格說明::=
    task [type] 标识符 is
          {入口_声明}
          {表示法_规格說明}
    end [标识符];
异常_处理段::=
    when异常_选择{|异常_选择}=>
```

語句_序列
异常_声明::=标识符_表：**exception**;
异常_选择::=异常_名 | **others**
訪問_类型_定义::=**access** 子类型_指示
地址_规格說明::=**for** 名字 **use at**
　　静态_簡单_表达式;
約束::=范围_約束 | 精度_約束
　　| 序标_約束 | 判别式_約束
延迟_語句::=**delay** 簡单_表达式;
关系::=簡单_表达式 [关系_运算符　簡单_表达式]
　　| 簡单_表达式 [**not**] **in** 范围
　　| 簡单_表达式 [**not**] **in** 子类型_指示
关系_运算符::= = | /= | < | <= | > | >=
成分_声明::=
　　标识符_表：子类型_指示 [：=表达式] ;
　　|标识符_表：数組_类型_定义 [：=表达式] ;
成分_表::={ 成分_声明 } [变体_部分] |**null**;
成分_結合::= [选择{|选择}=>] 表达式
因子::=初等量 [* * 初等量]
过程_調用::=过程_名 [实参_部分] ;
体::=子程序_体 | 程序包_体 | 任务_体
体_存根::=子程序_规格說明 **is separate**;
　　| **package body** 标识符 **is separate**;
　　| **task body** 标识符 **is separate**;
声明::=对象_声明 | 数_声明
　　|类型_声明 | 子类型_声明

| 子程序_声明 | 程序包_声明
| 任务_声明 | 异常_声明
| 换名_声明

声明_项::=声明 | 使用_子句

声明_部分::=
{ 声明_项 } { 表示法_规格說明 } { 程序_成分 }

形式_参数::=标识符

形参_部分::= (参数_声明{ ; 参数_声明 })

运算符_符号::=字符_串

判别式_約束::=
(判别式_规格說明{ , 判别式_规格說明 })

判别式_声明::=
标识符_表 : 子类型_指示 [: = 表达式]

判别式_部分::=
(判别式_声明{ , 判别式_声明 })

判别式_规格說明::=
[判别式_名{ |判别式_名 } =>] 表达式

序标::=类型_标記 **range** <>

序标_成分::=名字 (表达式{ , 表达式 })

序标_約束::=离散_范围{ , 离散_范围 }

返回_語句::=**return** [表达式] ;

块::= [块_标识符]
[**declare**
声明_部分]
begin
語句_序列

[exception
{ 异常_处理段 }]
end [块_标识符] ;
条件::=布尔_表达式
条件_入口_調用::=
select
入口_調用 [語句_序列]
else
語句_序列
endselect;
条件_語句::=
if 条件 then
語句_序列
{ elsif 条件then
語句_序列 }
[else
語句_序列]
end if;
私有_类型_定义::= [limited] private
初等量::=字面量 | 聚集 | 名字 | 类型_转换
| 分配_算符 | 函数_調用 | 限定_表达式|(表达式)
語句::={ 标号 } 簡单_語句 | { 标号 } 复合_語句
語句_序列::=語句{ , 語句 }
表示法_規格說明::=
长度_規格說明 | 枚举_类型_表示法
|記录_类型_表示法 | 地址_規格說明

表达式::= 关系{**and** 关系}
|关系{**or** 关系}
|关系{**xor** 关系}
|关系{**and than** 关系}
|关系{**or else** 关系}

使用_子句::=**use** 程序包_名{，程序包_名}；

枚举_字面量::=标识符 | 字符_字面量

枚举_类型_表示法::=**for** 类型_名 **use** 聚集；

枚举_类型_定义::=(枚举_字面量{，枚举_字面量})

函数_調用::=
函数_名 实参_部分 | 函数_名 ( )

范围::=簡单_表达式‥簡单_表达式

范围_約束::=**range** 范围

空_語句::=**null**；

转移_語句::=**goto** 标号_名；

实在_参数::=表达式

实参_部分::=（参数_結合{，参数_結合}）

实_类型_定义::=精度_約束

限定_表达式::= 类型_标記（表达式）
|类型_标記′聚集

定位::=**at** 静态_簡单_表达式 **range** 范围

定时_入口_調用::=
select
入口_調用［語句_序列］
or
延迟_語句［語句_序列］

```
    end select;
定点_約束::=
    delta 静态_簡单_表达式 [范围_約束]
单目_运算符::= + | — | not
迭代_子句::=while 条件
    | for 循环_参数 in [反向] 离散_范围
变体_部分::=
    case 判别式_名
        when 选择{ |选择 } =>
            成分_表
        end case
带有_子句::=with 单元_名{ , 单元_名 };
带基_数::基#基_整数 [。基_整数] #幂
参数_声明::=
    标识符_表：模式  子类型_指示 [：=表达式]
参数_結合::= [形式_参数=>] 实在_参数
类型_声明::=
    type 标识符 [判别式_部分] is 类型_定义
    |不完全_类型_声明
类型_定义::=
    枚举_类型_定义 |  整_类型_定义
    |  实_类型_定义 | 数組_类型_定义
    |記录_类型_定义 | 訪問_类型_定义
    |派生_类型_定义 | 私有_类型_定义
类型_转换::=类型_标記（表达式）
类型_标記::=类型_名 | 子类型_名
```

类属_子程序_声明::=
　　类属_部分　子程序_规格說明
类属_設置::=**new** 名字[(类属_結合{,类属_結合})]
类属_形参::=参数_声明;
　　|**type** 标识符[判别式_部分] **is** 类属_类型_定义;
　　|**with** 子程序_规格說明 [**is** 名字];
　　|**with** 子程序_规格說明 **is**〈　〉;
类属_实参::=表达式
　　|子程序_名|子类型_指示
类属_部分::=**generic**{类属_形参}
类属_結合::=[形式_参数=>]类属_实参
类属_类型_定义::=
　　(〈　〉)|**range**〈　〉|**delta**〈　〉|**digits**〈　〉
　　|数組_类型_定义|訪問_类型_定义
　　|私有_类型_定义
类属_程序包_声明::=
　　类属_部分　程序包_规格說明;
类属_程序包_設置::=
　　procedure 标识符 **is** 类属_設置;
　　|**funetion** 指明符 **is** 类属_設置;
标号::=<<标識符>>
标识符::=字母{[下横綫]字母|数字}
标识符_表::=标识符{,标识符}
指明符::=标识符|运算符_符号
复合_語句::=
　　条件_語句|情况_語句|循环_語句

```
    |块          |接受_語句|选择_語句
选择::=簡单_表达式|离散_范围|others;
选择_成分::=名字.标识符
    |名字.all|名字.运算符_符号
选择_备选::=
     接受_語句〔語句_序列〕
    |延迟_語句〔語句_序列〕
    |terminate;
选择_語句::=选择_等待
    |条件_入口_調用|定时_入口_調用
选择_等待::=
    select
        〔when 条件=>〕
            选择_备选
    {or〔when 条件=>〕
        选择_备选      }
     〔else
        語句_序列      〕
    end select;
浮点_約束::=digit 静态_簡单_表达式
項::=因子{乘法类_运算符  因子}
派生_类型_定义::=new 子类型_指示
离散_范围::=类型_标記〔范围_約束〕|范围
准綫_子句::=at mod 静态_簡单_表达式
簡单_表达式::=
    〔单目_运算符〕項{加法类_运算符  項}
```

简单_語句::=空_語句 | 入口_調用
| 赋值_語句 | 出口_語句 | 返回_語句
| 转移_語句 | 过程_調用 | 夭折_語句
| 延迟_語句 | 引发_語句 | 代碼_語句
乘法类_运算符::= * | / | **mod** | **rem**
換名_声明::=标识符:类型_标記 **rename** 名字;
| 标识符:**exception rename** 名字;
|**package** 标识符 **rename** 名字;
|**task** 标识符 **rename** 名字;
| 子程序_规格說明 **rename** 名字;
基::=整数
基本_循环::=
loop
語句_序列
end loop;
基_整数::=扩展_数字{ [下横綫] 扩展_数字 }
情况_語句::=
case 表达式 **is**
{选择{ |选择 }=>語句_序列 }
end case;
接受_語句::=
accept 入口_名{ 形参_部分 } [**do**
語句_序列
end [标识符]] ;
冪::=*E* [+] 整数 | *E* - 整数
程序_成分::=体_存根

```
    |体 | 程序包_声明 | 任务_声明
程序包_声明::=程序包_规格說明;
    |类属_程序包_声明; | 类属_程序包_設置;
程序包_体::=
    package body 标识符 is
        声明_部分
     [bagin
        語句_序列
     [exception
        { 异常_处理段 }]]
    end [标识符] ;
程序包_规格說明::=
    package 标识符
        { 声明_项 }
     [private
        { 声明_项 }
        { 表示法_规格說明 }]
    end [标识符] ;
赋值_語句::=变量_名 := 表达式
属性::=名字' 标识符
循环_参数::=标识符
循环_語句::=
     [循环_标识符: ]
     [迭代_子句]
        基本_循环
     [循环_标识符] ;
```

最高_优先級_运算符::= ** | **abs** | **not**
編譯::= { 編譯_单元 }
編譯_单元::=
　　上下文_規格說明　子程序_声明
　| 上下文_規格說明　子程序_体
　| 上下文_規格說明　程序包_声明
　| 上下文_規格說明　程序包_体
　| 上下文_規格說明　子单元
数值_字面量::= 十进制_數 | 带基_数
数_声明::=
　　标识符_表：**constant**：= 字面量_表达式;
数組_类型_定义::=
　　array（序标{，序标}）**of** 成分_子类型_指示
　| **array** 序标_約束 **of** 成分_子类型_指示
模式::= [**in**] | **out** | **in out**
精度_約束::= 浮点_約束 | 定点_約束
聚集::=（成分_結合{，成分_結合}）
整_类型_定义::= 范圍_約束
整数::= 数字{ [下横綫] 数字}

附录B　Ada保留字

下表中的字是语言保留字，他们对编译有特定的含义，因此不可用作标识符。

abort	declare	generic	of	select
accept	delay	goto	or	separate
access	delta		others	subtype
all	digits	if	out	
and	do	in		task
array		is	package	termin
at			pragma	then
	else		private	type
	elsif	limited	procedure	
	end	loop		
begin	entry		raise	use
body	exception		range	
	exit	mod	record	when
			rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

词 汇 表

abort,	夭折
accept *statement*	接受语句
access type,	访问类型
accuracy constraint,	精度约束
actual parameter,	实在参数
aggregate	聚集
array,	数组聚集
record,	记录聚集
all	成分选择符
allocator	分配算符
array	数组
dynamic,	动态数组
initialisation,	数组初始化
slice,	数组片
type definition	数组类型定义
constrained	约束数组
unconstrained	无约束数组
attribute enquiries	属性询问
base type,	基类型
body stub,	体存根
boolean type,	布尔类型

input_output, 浮点输入输出
for *loop* *for* 循环
formal parameter 形式参数
function 函数
generic, 类属
formal subprogram, 类属形式子程序
instantiation, 类属设置
package, 类属程序包
parameter, 类属参数
subprogram, 类属子程序
type definition, 类属类型定义
go to *statement*, 转移语句

identifier, 标识符
overloading, 重载标识符
if *statement*, 条件语句
input_output, 输入_输出
file, 输入_输出文件
INPUT_OUTPUT *INPUT_OUTPUT*
package 程序包
text, 正文输入_输出
character, 字符正文输入_输出
enumeration, 枚举正文输入_输出
fixed_point, 定点正文输入_输出
floating_point, 浮点正文输入_输出