ADA 语言基础教程（美国军方的专用计算机语言！）

PS：前天晚上应朋友之邀，过去喝茶聊天。其中有一位美籍台湾人。原来他还是 **IT** 界的老前辈，家住硅谷。**20** 多年前就在一家跨国大公司从事数据库管理，我们聊了很多。他说他以前做工程师时用的是 **Ada** 语言，我以前真没怎么听说过还有这种语言，回来查了一下，**Ada** 语言原来是美国军方的专用计算机语言！据说中国军方目前也在使用！**Ada** 是一种表现能力很强的通用程序设计语言，它是美国国防部为克服软件开发危机，耗费巨资，历时近 **20** 年研制成功的。它被誉为第四代计算机语言的成功代表。目前应用广泛度排 **16** 名。

Position May 2011	Position May 2010	Delta in Position	Programming Language	Ratings May 2011	Delta May 2010	Status
1	2	↑	Java	18.160%	+0.20%	A
2	1	↓	C	16.170%	-2.02%	A
3	3	=	C++	9.146%	-1.23%	A
4	6	↑↑	C#	7.539%	+2.76%	A
5	4	↓	PHP	6.508%	-2.57%	A
6	10	↑↑↑↑	Objective-C	5.010%	+2.65%	A
7	7	=	Python	4.583%	+0.49%	A
8	5	↓↓↓	(Visual) Basic	4.496%	-1.16%	A
9	8	↓	Perl	2.231%	-1.05%	A
10	11	↑	Ruby	1.421%	-0.67%	A
11	12	↑	JavaScript	1.394%	-0.69%	A
12	20	↑↑↑↑↑↑↑↑	Lua	1.102%	+0.61%	A
13	9	↓↓↓↓	Delphi	1.073%	-1.49%	A
14	-	=	Assembly	1.042%	-	A
15	16	↑	Lisp	0.953%	+0.30%	A
16	23	↑↑↑↑↑↑↑	Ada	0.747%	+0.32%	A
17	15	↓↓	Pascal	0.709%	-0.02%	A
18	21	↑↑↑	Transact-SQL	0.697%	+0.21%	B
19	-	=	Scheme	0.580%	-	B
20	25	↑↑↑↑↑	RPG (OS/400)	0.503%		

Ada 语言的中文资料极度缺乏，当当网里居然一本都没有。找到了一本 **80** 年底翻译的 **ada** 语言基础。下面这篇 **ADA** 语言文档还不错，有兴趣的同学可以了解一下，看了一下语法，跟 **C** 语言很像，应该属于 **C** 的高级改进版。

第 1 章 Ada 简介

1.1 概述(Overview)

Ada 语言最初设计是为了构建长周期的、高度可靠的软件系统。它提供了一系列功能来定义相关的数据类型(type)、对象(object)和操作(operation)的程序包(package)。程序包可以被参数化，数据类型可以被扩展以支持可重用库的构建。操作既可以使用方便的顺序控制结构，通过子程序(subprogram)来实现，也可以通过包含并发线程同步控制的入口(entry)来实现。Ada 也支持单独编译(separate compilation)，在物理层上支持模块性。

Ada 包含了很复杂的功能以支持实时(real-time),并发程序设计(concurrent programming)。错误可以作为异常(exception)来标示，并可以被明确地处理。Ada 也覆盖了系统编程(system programming)；这需要对数据表示和系统特性访问的精确控制。最后，提供了预定义的标准程序包，包括输入输出、字符串处理、数值计算的基本函数和随机数生成。

——译自《Ada Reference Manual》Section1:General

在本章里，我们将会初步介绍一下 Ada，以给读者留下大致的印象。

1.2 Ada 的历史(The History of Ada)

为了更好的理解 Ada，它的历史有必要简要地了解一下。

1974 年时，美国国防部(DoD)意识到开发和维护嵌入式系统(固化在硬件中的系统，如导弹弹导系统)耗费了过多的时间，精力和资金。

当时，在使用的计算机语言有 450 多种，这增加了开发新系统以及培训程序员使熟悉现有系统的时间和成本。维护工作也由于没有标准化的工具（编辑器，编译器等）而受阻。所有这些因素使 DoD 意识到它需要一门强大的语言，能被所有嵌入式电脑供应商使用。

开发工作始于 1975 年，当时 DoD 列举了一系列的语言需求；但没有一门现有语言指定了这些特性，因此在 1977 年，DoD 起草了一份建议，开发一门新的语言。不像 COBOL 这些语言由专门的委员会制定，新语言是一场竞争的主题，在产业界和学术界的评估中产生。

在众多竞争者当中，有 4 家被选中以完成进一步的工作。最终只剩下 Cii-Honeywell Bull 公司。这门语言被命名为 Ada。设计小组由 Jean Ichbiah 领导，对语言全权负责。

在 1983，Ada 成为了一个 ANSI 标准 ANSI/MIL-STD-1815A。此年成为一个 ISO 标准。在参考手册中定义的语言通常称为 LRM 或 ARM(Ada Reference Manual)。在

Ada 的相关书籍和编译器的报错信息中经常出现手册内容的引用。对于任何 Ada 站点，参考手册都是推荐的；虽然很难阅读，但它是所有 Ada 问题的最权威解释（一个小组正在澄清语言定义中已发现的语义不清的内容）。

Ada 也经过了一次修正，即 1995 的新的 ISO 标准。新标准修正了 Ada83 的很多缺陷，并进一步扩展了它的功能（在修正工作中，有个临时的标准，即 Ada9x，不少 Ada 文章是在 这段时间写的，因此有些内容在细节上可能与 Ada95 有所区别，但主要原理差不多）。

为了防止 Ada 编译器的不兼容版本的扩散，Ada Joint Program Office (控制 Ada 语言的执行部门，于 1998 年 10 月 1 日关闭，见 Ada Joint Program Office closed） 采取了不寻常的做法 – 他们注册 Ada 商标。除非通过他们的兼容性测试，编译器厂商不允许出售 'Ada' 编译器。这在不久以后放松了，保护协议变成了 `Validated Ada'。因而产生的 Ada 确认证书被限制在一定的时间内并有一个期满时间。当时间过期后，该编译器不能再被标记为`Validated Ada' 编译器。通过这种方式，AJPO 确保当前市场上的编译器与当前标准相一致。

目标是使所有的 Ada 程序能在所有系统上被编译-在这点上，AJPO 比其它语言小组做得好。

上述内容基本上是从 Quick Ada 翻译过来的(以前翻译该教材"半途而废"的残留品，直接引用一下了)，Ada 语言的详细历史细节见 The History of Ada

1.3 与 C 和 C++ 的比较(Contrast:Ada and C,C++)

由于 Ada 出生年月迟了一点，而且目前的操作系统基本上由 C，C++ 写成，导致 Ada 在"平民层"的推广比较糟糕，至今还不是很流行，Why Ada isn't Popular 一文对此有比较详细的解释。而 Ada 爱好者们为了显示 Ada 的优越性（这种心情相当能理解），将 Ada 与 C，C++ 做了一系列比较，其结果反正综和指数都是 Ada 高，这方面文章有不少，如 Comparing Development Costs of C and Ada，Contrasts: Ada 95 & C++。在这里，我们只初略地了解一下 Ada 的优势即可，在读者朋友接下去的学习中，应该是能从心里感受到 Ada 的优点。

1. 更高的安全性、可靠性。Ada 中对于访问内存、数值计算等很多方面有一些严格的规定，而没有 C 和 C++ 那么自由；程序的错误绝大部份能在编译和运行时检测到，以至于可以不需要编译器，另外，语言也包含异常特性，能方便地处理错误。
2. 更高的移植性。在 Unix 和 Windows 下有 C 编程经验的朋友应该对于兼容性深有体会，很多代码纯粹是为了适应不同的系统增添的，对于实际工作没多大用处。但 Ada 的初始语言环境中就有了异常（约等于 Unix 下的 Signal)、任务（线程）、

分布式计算、随机数产生、宽字符集很多特性的支持，而在现在的具体操作系统中，这些特性往往随系统而异。即使 Ada95 里缺少一些功能，也可以通过额外标准和函数库来弥补：GDI 库，可以用 GtkAda，在 Windows 和 X 下通用；Ada 也有一个 Posix 接口的标准，可以使用函数库 Florist 来调用 Posix 的函数.....用户层是大大省力—只要自己的操作系统上有所需的编译器和函数库即可。

3. 语法明确，基本上没有令人混淆的地方。Ada 程序的源代码远远比 C 或 C++ 的代码易懂。看程序的人是减轻了不少脑负担。
4. .....
5. .....

.....

总之，C 和 C++ 能做的 Ada 肯定能做，但 Ada 要省时方便的多。读者在学习 Ada 之后，无需多说也就明白了，笔者在初学 Ada 时就有耳目一新的感觉，唯一的遗憾是 Ada 不流行。

1.4 网络资源(Internet Resources)

Ada 虽然在国内不流行，但在国外还是有不少网站，下面是只列举一小部份，至于更多的资源，读者可至 VenusIC 查找:

1. Home of the Brave Ada Programmers (HBAP)，即 Ada Home，由 Magnus Kempe 维护，里面包含了不少 Ada 相关的文档、软件
2. Public Ada Library (PAL)。PAL 是 Ada 软件、文档的图书馆。主站点在 wuarchive.wustl.edu (WUARCHIVE), 以及法国的一个映像站点 mirror site，PAL 目前由 Richard Conn 维护。
3. AdaIC，由 AJPO 发起的一个站点，也包含了不少相关信息 。
4. SIGAda ，是 ACM 的一个 Special Interest Group。
5. AdaPower，很不错的一个站点，有相关教材、文档、Faq、Linux 等链接。
6. Ada Core Technology，Gnat,Glade 等软件的老家。
7. 新闻组 comp.lang.ada ，不用多介绍了

1.5 第一个程序(The First Program)

为了了解 Ada 程序的大致结构，举一个例子是难免的。大部份书籍一般都是用”hello world”程序来开始，我们下面就见识一个在终端屏幕上输出 “Hello World!”的简例。

```
000 -- filename:hello.adb;
001 with Ada.Text_IO;
002 procedure Hello is
003 begin
004    Ada.Text_IO.Put ("Hello World!");
005    Ada.Text_IO.New_Line;
006 end Hello;
```

先介绍一下在本教材中代码的一些问题：每行代码前的 000,001 等数字表示该代码是第几行，只为了讲解方便，在实际源代码中是不存在的；with，procedure 等保留字(reserved word)都用粗体表示，以示区别；有些字是用斜体表示，表示该字是用其它有效字符替换。

现在让我们分析上述的简单程序：

[000]标示该程序文件名为 hello.adb，在程序中并不需要；– 是注释符，表示从其所在位置至行尾都是注释，对程序没有什么影响，与 C 的 /* */类似。

[001] Ada.Text_IO 是预定义的一个程序包(package)；**with** *package_name* 和 C 的 include 功能差不多。

[002]-[006]是程序的主体部份。与 C 下的 main 函数类似，Ada 也需要一个主过程(main procedure)，在这个例子中是过程 Hello。过程的用法和上例一样，都是

```
procedure procedure_name is
   statements1;
begin
   statements2;
end procedure_name;
```

statement1 可以是变量、常量、函数、过程等的声明；statements2 是过程 procedure_name 要执行的语句，对象的声明不能在这部份；end 后的 *procedure_name* 不是必需的，但为了程序的可读性，应加上。

[003],[004] 分别输出”Hello World!” 和新行符。Put 和 New_Line 都是 Ada.Text_IO 里的过程。

但上 **7** 例调用过程 Put 和 New_Line 的方法比较罗嗦，因此我们也可以使用 use 语句:

000 — filename:hello.adb;
001 **with** Ada.Text_IO; **use** Ada.Text_IO;

002 **procedure** Hello **is**
003 **begin**
004 Put (“Hello World!”);

```
005    New_Line;
006 end Hello;
```

这样 Ada 编译器就能在程序包 Ada.Text_IO 中自动搜寻 Put，New_Line，而无需用户指定它们所在的准确位置。

我们在将上例略微改动以下，以使读者了解一下声明部份：

```
000 -- filename:hello.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 procedure Hello is
003    Str:String := "Hello World!";
004 begin
005    Put(Str);
006    New_Line;
007 end Hello;
```

在 is 和 begin 之间，声明了一个字符串变量 Str，它的初始值为”Hello World”。String 是预定义的字符串类型。上述的声明方式和 Pascal 差不多。 现在我们对 Ada 程序长的什么样已有了基本的认识，下面需要了解几个术语。一个 Ada 程序是由一个或多个程序单元组成(program unit)。一个程序单元可以为：

1. 子程序（subprogram）,定义一些可执行运算。过程(procedure)和函数(function)都是子程序。
2. 程序包（package）,定义一些实体（entity）。程序包是 Ada 中的主要分组机制，类似于 C 的函数库，Modula 的”module”。
3. 任务单元（task unit）,与线程类似，定义一些计算，然后并发执行。
4. 保护单元（protected unit）,在并发计算中协调数据共享，这在 Ada 83 中不存在。
5. 类属单元（generic unit）,帮助构建可重用组建，和 C++ 的模板类似。

后 3 类属于高级话题，在后面的章节中我们会依次介绍。程序单元又可为两部份：

1. 声明部份(declaration)。定义对其它程序的接口，有些可用资源可以被用户使用，与 C 下的’.h’文件相似。
2. 主体部份(body)。是声明部份的具体实现，等价与 C 的’.c’文件。

其中程序包（package）和子程序(subprogram)是用的最广泛的 2 个程序单元。大部份 Ada 程序基于一堆程序包，在以一个主过程(main proceudre) 来开始 Ada 程序 。

第 2 章 基本数据类型和表达式(Basic Types and Expressions) 2.1 概述 (Overview) 数据类型是一门计算机语言最基本的特性，表示一个对象的类型，是数字、字符或其它类型。由于 Ada 在数据类型上提供的强大处理能力，我们又不能很简

单地认为数据类型仅是定义一个对象的类型。在 Ada 里，数据类型可以自己创建，它的相关运算符也可以自己定义，同时又有数据类型属性这项特性，具有相当大的灵活性。学过其它的语言，特别是有 C 和 Pascal 背景的朋友初次接触时可能会感到有点新鲜。 创建新类型，是用户自己定义数据类型，包括该类型的名称、取值范围及相关操作；其中又包括了派生类型和创建子类型，即以现有数据类型为母类型创建新类型，继承母类型的一部份属性。 数据类型属性，如同我们玩 RPG 游戏时的人物属性：体力值、魔法值一样，是说明该类型固有的属性，包括最小取值范围、最大取值范围等等。 本章将会先介绍词法元素以及创建数据类型的基础性知识，然后是整型(integer)、实型(real)、字符型(character)、布尔型(boolean）、枚举类型(enumuration)这几种标量类型，最后则是相关的数据类型属性、类型限制和类型转换、表达式和运算符。 **2.2 词法元素(Lexical Element)** Ada 里的词法元素与其它语言的定义还是有不小差别，下面按照 RM95 关于词法元素的分类来逐项介绍，包括标识符、保留字、字符、数值文字等及它们的一些规则。 **2.2.1 基本字符集(Character Set)** Ada 95 规定的标准字符集是 Latin-1,支持 8 位(基于 ISO-8869)和 16 位(基于 ISO-10646)字符，在非标准模式下也可用本地字符集，具体情况取决于当前的系统。(一般来说，涉及字符时默认是指 Latin-1，程序几乎都是用 Latin-1 字符集写的) 字符在 RM 95 中是分成了三类：图形字符(graphic character)、格式控制符(format effector)、其它控制符(other control function),它们所表示的范围为： 图形字符包括字母(letter)、数字(digit)、空格符(space)、特殊字符(special character)(例如” # & ‘ ( ) * + , – . / : ; < = > _ | { } [ ]); 格式控制符包括水平制表符(HT)、垂直制表符(VT)、回车(CR)、换行(LF)、换页(FF); 其它控制符则是除了格式控制符以外的控制符。 更详细的内容参见 RM 95、ISO 8869,ISO 10646。 Ada 是大小写忽略的（除了字符和字符串中的实际内容，如字符’z' 和’Z'不相同，但标识符 z 和 Z 相同），但为了可读性，一般来说变量名或函数名首字母都会大写，其余小写，缩近格式也需要引起注意。根据实际情况尽量保证可读性。为了兼容性，Ada95 要求编译器最少支持一行字符串和一个词法元素的长度为 200 个字符（不包括行尾符）。 Ada 在字符集上并没有很严格要求编译器一定要怎样，但应该支持标准字符集。 **2.2.2 标识符(Identifier)** Ada 中不管是变量、函数还是其它对象都需要一个名称，这就叫做标识符。如 X、Count 、me 就是简单的标识符。 Ada 中的标识符有以下一些注意点： 1. 标识符一定要是字母开头，接下去可以是数字和下划线，结尾不能为下划线。如 Fig_、_Lik、1me 是不合法的。 2. 两个连续的下划线不能在一起，因为有些打印机可能会将两个下划线当作一个处理。 3. 虽然单个字符可作为标识符，但一般情况下不应该滥用这项特性（我不知道这除了少敲几个字外，还有什么意义，想想数字 0 和字母 O、数字 1 和字母 l 吧，绝对害死人的做法）。 4. 不能将保留字作为标识符。在 Ada 83 中，标识符包括了保留字，但在 Ada 95 中保留字从标识符中分离了出来。 5. .如上节所提及的，标识符是不分大小写的，但为了可读性，请注意你对标识符的命名。 **2.2.3 保留字(Reserved Word)** 保留字在程序语法中有特殊的含义，不属于标识符范围，这与 C 和 Pascal 等语言的定义有所不同。Ada 95 中的保留字如下： **abort abs abstract accept access aliased all and array at begin body case constant declare delay delta digits do else elsif end entry exception exit for function generic goto if in is limited loop mod**

new not null of or others out package pragma private procedure protected raise range record rem renames requeue return reverse select separate subtype tagged task terminate then type until use when while with xor 在以后的内容中，我们会逐个解释它们的含义及作用。Ada95 的保留字比原先添加了 6 个：**abstract**, **aliased**, **protected**, **requeue**, **tagged** 和 **until**，虽然这可能会牵连到兼容性问题，但通常也无须计较这点。 **2.2.4 分隔符(Separator and Delimiter)** Ada 程序中，各元素之间需要空格符、格式控制符或 EOF 隔开。RM 95 里将它们作为 separator,而 delimiter 则是指下列特殊字符(中文有点说不清楚）： & ‘ ( ) * + , -. / : ” < = > | 或是复合型的: => .. ** := /= >= <= << >> <>。当分隔符作为注释、字符串、字符、数值的一部分时，就不再是分隔符了。 **2.2.5 数值文字(Numeric Literal)** 数值文字就是指数字。Ada 和 Pascal 相同，将数字分为实数型(real literal)和整数型(integer literal)两大类。实数型是有小数点的数字，整数型则无小数点。如 1.2787,0.871,7.0 是实数型，而-882,5441,1 是整数型。Ada 在数字表示上有一个很好的特性就是可以明确指定使用何种基数（2 进制到 16 进制）表示，下面是数字的表示： **十进制数(Decimal Literal)** 不管是实型还是整型数，都可以在其间加上下划线，使长数字更加易读。如 56886515645125615,可写为 56_886_515_645_125_615 或 5_6886_5156_4512_5615,下划线并不改变数字的值。但两个下划线不能是连续的，下划线也不可以在数字首部和尾部,如 676__66 和 67_E4 都是非法的。 字母 E 作为数字的指数，同时适用于实型和整型。如 123_98E4、5.087E-5、 4.8E7 都是合法的，但负指数不能用于整型，指数也一定要是整数。E 大小写皆可以。 **基型数字(Based Literal)** 在大部分语言中，都使用 10 进制数字表示；Ada 里整数可以不用 10 进制的表示方法书写，而是直接使用 2 至 16 进制的表示法，格式为:*Base # Number #*,*Base* 表示所采用的进制，*Number* 为该进制下所表示的数字。 2#1001_1001#,表示 2 进制数 1001 1001,中间的下划线可取消，其 10 进值为 153; 10#153#,表示 10 进制数 153，等价与 153； 16#90#,表示 16 进制数 90，其 10 进值为 144； **2.2.6 字符文字(Character Literal)** 字符文字的表示是单个图形字符在单引号 ‘ ’中，如’ a’表示小写字母 a, ‘K’表示大写字母 K,’ ‘ ‘表示一个单引号,’ ‘表示一个空格。 **2.2.7 字符串文字(String Literal)** 字符串是双引号（” “)之间的有序图形字符。如” What I said.”就是一个字符串。表示空字符串时直接用” “。如果字符串中有双引号，一个双引号要用两个”来表示。如” He said,”" I am hungry.”" “，而” He said,” “I am hungry. “或” He said,” I am hungry.” “是不合法的。至于其它字符，如$ %之类可以直接出现在两个双引号间。与 C 语言不同，Ada 里没有与之相同的转义字符，并且 EOL 不会算到字符串中。 **2.2.8 注释(Comment)** 注释由两个连字号(hyphen)(–)开始，直到行尾。可以出现在程序的任一个地方，不影响程序本身。例如： — the comment; **end**;– processing of Line is complete. **2.2.9 Pragmas** Pragma 是编译指示(compile directive),给编译器指令如优化程序，列表控制等。它的作用往往不只影响一个编译单元，而是整个程序。 Pragma 是些预先定义的指令，如 pragma Page,pragma List(OFF) ,编译器也可扩展 RM 95 中 pragma。我们先接触 List，Page，Optimize 这 3 个 pragma。更多内容我们会在以后接触。 **pragma** List(identifier); **pragma** Page; **pragma** Optimize(identifier); **pragma** List 将 identifier On 或 Off

作为它的参数。它指定编译列表（listing of compilation) 是继续还是停止，直到在同一个编译单元内，一个 pragma List 使用了相反的参数。 **pragma** Page 指定在 pragma 后的程序正文在新页开始（如果编译器正在列表） **pragma** Optimize 有 Time，Space 或 Off 3 个参数,它的影响区域直到 pragama 所在编译单元的底部。 Time 和 Space 指示优化时间还是优化空间，Off 则关闭优化。 下面是简单的例子： **pragma** List(Off)； －－关闭列表生成 **pragma** Optimize(Off)； －－关闭可选的优化 不过，上述 3 个 pragma 是影响编译过程用的，基本上用户也用不着，以后涉及的 inline，Pure 等 pragma 使用频率倒挺高 **2.3 创建数据类型和子类型 (Creating Types and Subtypes)** 使用变量时，除了以某标识符作为变量的名称外，还要指定该变量的数据类型。一个数据类型定义了变量可接受的值以及所能执行的操作。比如说，一个数据类型为 Age 的变量 Bill，Age 的取值范围为 1..100,并只有 + – 这两种操作，在这里,对象(object)为名为 Bill 的变量，它的取值在 1..100 之间（包括 1，100），值的变化只能通过+ -这些基本运算符(primitive operation)来实现,而无法通过* /等其它运算符。Ada 中的数据类型，包括预定义类型，都是按照一定的格式在程序包中创建的。下面就介绍创建数据类型的一些基本内容，更多相关内容会在以后见到。 **2.3.1 创建新的数据类型** 创建一个新类型，需要使用保留字 **type，is,range**。格式如下： **type** *type_name* **is range** *range_specifcation*; *type_name* 为新类型的名称，是一个合法标识符；*range_specifcation* 表示该类型的取值范围，表示方式为 First .. Last,如 1..100 , -9 ..10 。 例如创建上面提及的一个新类型 Age： **type** Age **is range** 1 .. 100; 这样就有了一个数据类型 Age, 取值范围 1 .. 100。 有一点要注意：*range_specfication* 中 First 要大于 Last。 如 **type** months **is range**12 .. 0, 实际上 months 是一个空集(null)，而不是所期望的 0..12。 不同数据类型之间是不能进行混合运算的，即使取值范围和运算符一样,看以下的程序例子： 000 — filename: putwage.adb 001 **with** Ada.Text_IO; **use** Ada.Text_IO; 002 **with** Ada.Integer_Text_IO; **use** Ada.Integer_Text_IO; 003 **procedure** putwage **is** 004 **type** Age **is range** 1 .. 100; 005 **type** Wage **is range** 1 .. 100; 006 Bill_Age : Age := 56; 007 Bill_Wage: Wage := 56; 008 **begin** 009 Put (“Total wage is”); 010 Put (Bill_Wage * Bill_Age); 011 New_Line; 012 **end** putwage; [001]-[002]: 使用软件包 Ada.Text_IO,Ada.Integer_Text_IO;两个软件包分别处理字符类输出和整数输出。 [003] [008] [012] 定义一个过程 putwage。 [004]-[005]: 定义新的数据类型 Age,Wage,它们取值范围都为 1..100。 [006]-[007]: 声明两个变量 Bill_Age,Bill_Wage,类型分别为 Age 和 Wage, 并赋予相同初始值 56。 [009]-[011]:依次输出字符串”Total wage is”,整数 Bill_Wage 和 Bill_Age 的乘积，和一个新行符 (EOL)。 以上程序看上去毫无问题，但根本无法编译通过。首先，没有定义类型 Age 和 wage 的 * 操作，因此 Bill_Age 和 Bill_Wage 无法相乘；第二，两者数据类型不同，即使定义了*操作，还是无法相乘。 当然也可使用后面提到的类型转换 ，如果将[010] 改为 Put (Integer(Bill_wage) * Integer(Bill_Age)),将会输出所要的 3136;但如果改成 Put (Integer(Bill_wage * 56))，看上去也行的通，但实际结果却不是 3136。不同数据之间不能进行运算，要牢牢记住。(Integer 是预先定义的一个整型，Integer(Bill_Wage)是将 Bill_Wage 强制转换为整型）。 **2.3.2 派生类型** 大家可能会

发现，如果像上面一样创建一个截然不同的新类型，还需要定义它的运算符，使用很不方便。因此，往往是派生现有的类型，其格式为: **type** *type_name* **is new** *old_type* {**range** *range_specification*}; *type_name* 为新类型的名称，是一个合法标识符；**range** *range_specification* 表示该类型的取值范围，是可选的，没有的话表示新类型 *type_name* 的取值范围和 *old_type* 一样。如将上例改为： 000 — filename:putwage.adb 001 **with** Ada.Text_IO; **use** Ada.Text_IO; 002 **with** Ada.Integer_Text_IO; **use** Ada.Integer_Text_IO; 003 **procedure** putwage **is** 004 **type** Age **is new** Integer **range** 1 .. 100; 005 **type** wage **is new** Integer; 006 Bill_Age : Age := 56; 007 Bill_Wage: Wage := 56; 008 **begin** 009 Put ("Total wage is"); 010 Put (Bill_Wage * Bill_Age); 011 New_Line; 012 **end** putwage; 上例还是不能编译通过，因为派生类型只继承母类型的属性，如运算符，不同的派生类型即使母类型相同也还是属于不相同的类型。但将[10]改为 Put (Integer(Bill_wage * 56))则能输出正确的结果。但是派生类型使用还是麻烦了一点，不同类型之间即使都是数字类型也无法混合使用,只是自己不用创建运算符省力了点。 **2.3.3 创建子类型** 创建新类型和派生类型的麻烦从上文就可以感受的到，特别是在科学计算这些有很多种小类型的软件当中，上述两种方法实在过于繁杂。这时子类型(subtype)就相当有用，子类型的定义格式为: **subtype** *type_name* **is** *old_type* {**range** *range_specification*}; *type_name* 为新类型的名称，是一个合法标识符；**range** *range_specification* 表示该类型的取值范围，是可选的，没有的话表示新类型 *type_name* 的取值范围和 *old_type* 一样。再将先前的例子改一下： 000 — putwage.adb 001 **with** Ada.Text_IO; **use** Ada.Text_IO; 002 **with** Ada.Integer_Text_IO; **use** Ada.Integer_Text_IO; 003 **procedure** putwage **is** 004 **subtype** Age **is** Integer **range** 1 .. 100; 005 **subtype** Wage **is** Integer; 006 Bill_Age : Age := 56; 007 Bill_Wage: Wage := 56; 008 **begin** 009 Put ("Total wage is"); 010 Put (Bill_Wage * Bill_Age); 011 New_Line; 012 **end** putwage; 编译通过，输出值为 3136。子类型不仅继承母类型的属性，而且和母类型、其它同母类型的子类型可混合使用。 在前面的例子中的，我们都提到了取值范围，这也是 Ada 的一项“特色”：Ada 不同于 C 和 Pascal— 赋给一个变量超过其取值范围的值或进行不合法运算，会输出错误的值而不报错，与此相反，Ada 程序在编译时会提示错误，或在运行 Ada 程序时产生 Constraint_Error 异常（异常和 C 中的信号 Signal 差不多，详见**第 9 章 异常(Exception)**），挂起程序，来减少程序的调试时间。 **2.4 标量类型（Scalar Type)** 大部份语言，基本的数据类型如果按照该类型所表示的数据类型来分，一般来说可分为整型(integer)，实型(real)，布尔型(boolean)，字符型(character)这四类，并以它们为基础构成了数组，记录等其它更复杂的数据类型。在程序包 Standard 中预定义了一些简单数据类型，例如 Integer，Long_Integer，Float，Long_Float，Boolean，Character，Wide_Character，以及这些数据类型的运算符。下面我们除了学习上述的 4 种标量类型（Scalar Type）外，还要学习一下枚举类型(Enumration)。由于 Ada 中布尔型和字符型都是由枚举类型实现的，因此也可将这两种类型认为是枚举类型。 **2.4.1 整型(Integer) Integer 整型** 一个整型数据能存放一个整数。预定义的整型有 Integer，Short_Integer，Short_Short_Integer，Long_Integer，Long_Long_Integer 还有 Integer 的子类型 Positive ，Natural。

RM95 没有规定 Integer 及其它整型的具体取值范围及其位数，由编译器决定。只规定了没多大意思的最小取值范围，如要求一个 Integer 至少要为 16 位数，最小取值范围为-32767..32767（-2 ** 15+1 .. 2**15-1)。因此还有 Integer_8,Integer_16,Integer_32,Integer_64 这些指定了位数的整型,以方便用户。在 RM95 里，也就是编译器实现里，以上类型声明格式为: **type** Integer **is range** *implementation_defined*(Long_Integer 它们也一样） **subtype** Positive **is** Integer **range** 1..Integer'Last; **subtype** Natural **is** Integer **range** 0..Integer'Last; (Integer'Last 表示 Integer 的最后一个值，即上限，见 2.5 数据类型属性） 程序 System 里定义了整数的取值范围: Min_Int : **constant** := Long_Long_Integer'First; Max_Int : **constant** := Long_Long_Integer'Last; **Modular 整型** 还有一类整型是 Modular,异于上面的整型。如果将 Integer 整型与 C 中的 signed int 相类比，它们的取值范围可包括负数；那么 Modular 类型就是 unsigned int，不能包含负数。其声明格式为: **type** *tyep_name* **is mod** *range_specification*; 其中的 *range_specification* 应为一个正数; *type_name* 的取值范围为（0..*range_specification* – 1)。 如下面类型 Byte: **type** Byte **is mod** 256; 这里 Byte 的取值范围为 0 .. 255。 Modular 类型在程序包 System 也有常量限制，*range_specification* 如是 2 的幂则不能大于 Max_Binary_Modulus ，如不是幂的形式则不能大于 Max_Nonbinary_Modulus。这两个常量的声明一般如下： Max_Binary_Modulus : **constant** := 2 ** Long_Long_Integer'Size; Max_Nonbinary_Modulus : **constant** := Integer'Last; 细心的读者可能会发现上面两个常量的值实际上是不一样的，也就是说 Modular 类型实际上有两个不同的限制。RM95 关于这点的解释是,2 进制兼容机上，Max_Nonbinary_Modulus 的值大于 Max_int 很难实现。 **2.4.2 实型(Real)** 相对于整型表示整数，实型则表示浮点数。实型分为两大类: 浮点类型(floating point) 和定点类型 (fixed point)。它们之间的区别在于浮点类型有一个相对误差;定点类型则有一个界定误差，该误差的绝对值称为 delta。下面就分类介绍这两类数据类型。 **浮点类型(Floating Type)** 浮点类型预定义的有 Float，Short_Float，Short_Short_Float，Long_Float，Long_Long_Float 等,它们的声明格式入下: **type** *type_name* **is digits** *number* [**range** *range_specification*] ; **digits** *number* 表示这个浮点类型精度，即取 *number* 位有效数字，因此 *number* 要大于 0；**range** *range_specification* 是可选的，表示该类型的取值范围。下面是几个例子： **type** Real **is digits** 8; **type** Mass **is digits** 7 **range** 0.0 .. 1.0E35; **subtype** Probability **is** Real **range** 0.0 .. 1.0; Real 表示精度为 8 位的符点数类型，它的取值范围由于没给定，实际上由编译器来决定; RM 95 里关于这种情况是给出了安全范围(safe range), 取值范围是-10.0**(4*D) .. +10.0**(4*D), D 表示精度，此例中为 8,所以 Real 的安全取值范围一般来说应为 -10.0E32 .. +10.0E32。 Mass 是表示精度为 7 位的符点型，取值范围为 00.. 1.0E35; Probability 是 Real 的子类型，精度也是 8 位，取值范围 0.0..1.0; 程序包 System 定义了精度的两个上限：Max_Base_Digits 和 Max_Digits ,一般来说应为 Max_Base_Digits : **constant** := Long_Long_Float'digits;(即 Long_Long_Float 的精度） Max_Digits : **constant** := Long_Long_Float'digits; 当 *range_specification* 指定时，所定义类型的精度不能大于 Max_Base_Digits;当 *range_specification* 没有

指定时，所定义类型的精度不能大于 Max_Digits。 **定点类型** 定点类型主要是多了一个 delta，它表示该浮点类型的绝对误差。比方说美元精确到 0.01 元（美分），则表示美元的数据类型 Dollar 的 delta 为 0.01，不像浮点型是近似到 0.01。 定点型的声明格式有两种： 普通定点型：**type** *type_name* **is delta** *delta_number*_**range** *range_specification*； 十进制定点型：**type** *type_name* **is delta** *delta_number* **digits** digit_number [**range***range_specification]*； 除 **delta** *delta_number* 外，各部份意义与浮点型相同。 定点型中有一个 small 的概念。定点数由一个数字的整数倍组成，这个数字就称为该定点数类型的 small。如果是普通定点型，则 small 的值可以被用户指定（见下节 数据类型属性），但不能大于该类型的 delat；如果没有指定，small 值由具体实现决定，但不能大于 delta。如果是十进制定点型，则 small 值为 delta，delta 应为 10 的幂,如果指定了该定点型的取值范围，则范围应在 -(10**digits-1)*delta..+(10**digits-1)*delta 之间。看一下下例： **type** Volt **is delta** 0.125 **range** 0.0..255.0; **type** Fraction **is delta** System.Fine_Delta **range** -1.0..1.0; **type** Money **is delta** 0.01 **digits** 15; **subtype** Salary **is** Money **digits** 10; **2.4.3 布尔型 (Boolean)** 逻辑运算通常需要表示”是”和”非”这两个值，这时就需要使用布尔型。Ada 中的布尔型与 Pascal 中的类似，是 True 和 False 两个值。布尔型属于枚举数据类型，它在程序包 Standard 中定义如下： **type** Boolean **is** (True, False); 习惯于 C 语言的朋友在这里需要注意一下，Boolean 的两个值 True,False 和整型没有什么关系，而不是 C 语言中往往将 True 定义为值 1，False 为 2。 **2.4.4 字符型(Character)** Ada83 最初只支持 7 位字符. 这条限制在 Ada95 制订前已经放松了，但一些老编译器如 Meridian Ada 还是强制执行. 这导致在一台 PC 上显示图形字符时出现问题;因此，在一般情况下，是使用整型来显示 Ascii 127 以后的字符,并使用编译器厂商提供的特殊函数。 在 Ada95 里，基本字符集已由原来的 ISO 646 标准的 7 位字符变为 ISO 8859 标准的 8 位字符，基于 Latin-1 并且提供了 256 个字符位置。 Ada95 同样也支持宽字符 ISO 10646，有 2**16 个的字符位置。因此现代编译器能很好地处理 8 位字符和 16 位字符。 7 位字符在已经废弃的程序包 Standard.Ascii 内定义。在程序包 Standard 内预定义的字符型 Character 和 Wide_Character 分别表示 Latin-1 字符集和宽字符集，类型 Wide_Character 已经包含了类型 Character 并以它作为前 256 个字符。程序包 Ada.Characters.Latin_1 和 Ada.Characters.Wide_Latin_1 提供了 Latin-1 字符集的可用名称，Ada.Characters.Handling 则提供一些基本的字符处理函数。具体内容见 第 14 章 字符和字符串处理。 从下例可以了解一下字符型： 000 — filename: puta.adb 001 **with** Ada.Text_IO; **use** Ada.Text_IO; 002 **procedure** puta **is** 003 **subtype** Small_Character **is** {‘a’ ,’b','c', ‘d’}; 004 Level : Small_Character := ‘a’; 005 **begin** 006 Put (“You level is”); 007 Put (Level); 008 New_Line; 009 **end** puta; [003] 创建了一个字符类型 Small_Character,包含 a,b,c,d 四个字母；如 C 语言一样，使用字符时需加’ ‘。 [004]声明变量 Level,类型为 Small_Character,值为字母 a 。 上面这个例子主要还是说明一下字符类是怎样定义的，但 Character 和 Wide_Chracter 实际实现却不是这么简单。 **2.4.5 枚举类型 (Enumeration)** 有时候我们需要一个变量能表示一组特定值中的一个。如 today 这个变量，我们希望它的值是 Monday, Tuesday, Wednesday, Thursday, Friday,

Saturday, Sunday 其中的一个，这时枚举类型就相当有用，上述情况中就可以创建新类型 Day,如下： **type** Day **is** (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday); 然后声明变量 today 的数据类型为 Day: today : Day ; 这样 today 就能接受 Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday 这几个值中的任意一个。 前面提及的类型 Character ,Wide_Character,Boolean 都是枚举类型，也按照下面给出的格式声明: **type** *type_name* **is** (*elememt_list*)； *element_list* 需列举出该类型所有的可能值。 Ada 能自动检测上下文，因此大部份情况下能分辨不同枚举数据类型下的枚举元素，如再声明一个类型 Weekend: **type** Weekend **is** (Saturday, Sunday); 或 **subtype** Weekend **is range** Saturday .. Sunday; 赋给上例中的变量 Today 值为 Sunday 时，不会产生歧义；但在有些情况下，Ada 无法分辨枚举元素时则会产生问题，这时就要使用类型限制，详见 2.6 类型限制和类型转换 Ada 中的基本数据类型就讲到这里，实际上本节是基于上一节内容的扩展，说穿了还是创建数据类型。Ada 在数据类型处理上提供的强大功能在接下的章节里我们将会接触的更多，在这方面 Ada 的确比其它大部份语言做的好多了，熟悉 C ,Pascal 的朋友大概会感到相当有意思。 **2.5 数据类型属性 (Attributes)** 数据类型属性，表示某个数据类型的具体特征—取值范围，最小值，最大值，某数在该类型中的位置 ..... 应该说是相当有用的——起码不像 C 语言，还要翻翻系统手册才能知道某个数据类型的具体定义。这些属性的用法和调用函数一样，也可以认为它们就是预定义的函数—-虽然不怎么准确，关于函数，详见 第 6 章 子程序；有些返回值为通用类型(*universal type*)和字符串型。 数据类型的属性既可以使用预先定义的操作，也可以自己定义数据类型的属性，如 S'First 是返回 S 类型的下限，如果用户不满意默认的属性，也可自己指定 S'First（虽然没必要），如: **for** S'First **use** My_Version_First; My_Version_First 是用户自己的函数，以后当使用 S'First 时，实际上调用 My_Version_First；有些数据属性也可以直接用数字标示，如： **for** S'First **use** 0; 这样，S'First 的值就成了 0。在很多地方，如内存管理中，这种用法还是比较普遍的。下面简单地列出标量类型的属性，S 表示某个标量类型： **2.5.1 通用标量类型属性** S'First 返回 S 类型的下限，返回值为 S 型。 S'Last 返回 S 类型的上限，返回值为 S 型。 S'Range 返回 S 类型的取值范围，即 S'First .. S'Last。 S'Base 表示 S 类型的一个子类型，但没有范围限制（单纯从取值范围角度讲，"儿子"反而比"父母"大），称之为基类型(*base type*)。 S'Min 函数定义为: **function** (*Left,Right*:S'Base) **return** S'Base 。比较 *Left* 和 *Right* 的大小，返回较小值。如：Integer'Min (1 , 2) = 1 。 S'Max 函数定义为: **function** (*Left,Right*:S'Base) **return** S'Base 。比较 *Left* 和 *Right* 的大小，返回较大值。如：Integer'Max (1 , 2) =2 。 S'Succ 函数定义为：**function** S'Succ (*Arg* :S'Base) **return** S'Base 。返回 *Arg* 的后趋。 S'Pred 函数定义为: **function** S'Pred (*Arg* :S'Base) **return** S'Base 。返回 Arg 的前趋。 S'Wide_Image 函数定义为: **function** S'Wide_Image (*Arg* : S'Base) **return** Wide_String 。返回 *Arg* 的"像"，即可显示的字符串，这里返回宽字符串型 Wide_String。如：Float'Wide_Image (9.00) = "9.00" 。详见 第三章 数组。 S'Image 与 S'Wide_Image 一样，但返回字符串型 String 。 S'Wide_Width 表示 S'Wide_Image 返回的字符串的最大长度，返回值

为 *universal_integer*。 S'Width 表示 S'Image 返回的字符串的最大长度,返回值为 *universal_integer*。 S'Wide_Value 函数定义为: **function** S'Wide_Value (*Arg* : Wide_String) **return** S'Base。是 S'Wide_Image 的逆过程，返回与"像"*Arg* 相对应的 S 类型的值。如: Float'Wide_Value ("9.00") = 9.00 。 S'Value 与 S'Value 一样，但参数 *Arg* 是 String 类型。 **2.5.2 通用离散类型属性** 离散类型包括整型和枚举型，除了上述的属性外，还有： S'Pos 函数定义为: **function** S'Pos (*Arg* : S'Base) **return** *universal_integer*。返回 *Arg* 在 S 类型中的位置。 S'Val 函数定义为: **function** S'Pos (*Arg* : S'Base) **return** S'Base。返回 在 S 类型中位置为 *Arg* 的值。例如： type Color is (red, white, blue); Color'Pos (White) = 2 Color'Val (1) = red **2.5.3 浮点类型属性** S'Digits 返回 S 类型的精度，为 universal_integer 类型。 **2.5.4 定点类型属性** S'Small 返回 S 类型的 small 值，返回值为实型。 S'Delta 返回 S 类型的 delata，返回值为实型。 S'Fore 返回 S 类型所表示数的小数点前面的最小字符数量，返回值类型 universal_integer。 S'Alt 返回 S 类型所表示数的小数点后面的最小字符数量，返回值类型 universal_integer。 **十进制定点型属性** S'Digits 返回 S 类型的精度。 S'Scale 返回 S 类型的标度 N，使 S'Delta=10.0**(-N)。 S'Round 函数定义为 **function** S'Round(X:*universal_real*) **return** S'Base；返回 X 的舍入值。 **2.6 类型限制和类型转换（Type Qualification and Type Conversion)** 先让我们看一下下面的例子：

```
type primary is (red, green, blue);
type rainbow is (red, yellow, green, blue, violet);
...
for i in red..blue loop
...
```

这里明显有歧义，编译器也就无法自动判定 red，blue 到底是 primary 还是 rainbow 中的元素。因此我们就需要使用类型限制，精确指明元素：

```
for i in rainbow'(red)..rainbow'(blue) loop
for i in rainbow'(red)..blue loop -- 只需要一个限制
for i in primary'(red)..blue loop
```

类型限制并不改变一个值的类型，只是告诉编译器程序员所期望的数据类型。由于 Ada 中提供了重载等特性，变量、子程序等重名的现象相当普遍，以后我们会碰到，解决方法和现在所讲的类型限制差不多，都是指明资源的准确位置。

类型转换我们也常常在使用，比方说 modular 类型的数的输入输出需要特定的程序包，初学者便可以将 modular 数转换成 Integer 数来处理（虽然不怎么好）。下面是一个 Integer 和 Float 类型数互相转换的例子：

```
X : Integer:= 4;
Y : Float;
Y := float(X);
. . .
```

```
X := Integer(Y);
```

这导致编译器插入合适的代码来做类型转换，实际效果就取决于编译器，而且一些类型之间的转换是禁止的；像上述的强行转换一般也不推荐，除非意义很明确，建议是使用上节所讲的数据类型属性、以及以后会碰到的相关子程序等来进行类型转换。

还有一种不进行检查的类型转换，我们会在以后遇到。

2.7 表达式和运算符(Expressions and Operators)

我们在先前的简单例子中对于一些表达式和运算符已有所认识，现在具体的讲解一下。

2.7.1 变量、常量声明

变量声明

变量声名的格式很简单：

variable_name : type;

variable_name 是变量名称，只要是合法的标识符即可；*type* 为该变量的数据类型。声明时可以初始化变量值，也就只需在上述语句后添加 := *value*，:= 赋值运算符。如：

```
Var_1: Integer;
Var_2: Integer := 1;
```

也可以在声明某一变量时指定它的取值范围。如：

```
Var_1 : Integer range 1..10;
Var_2 : Integer range 1..20 := 2;
```

常量声明

常量声明格式如下：

variable_name **:constant** *type* := *value;*

这里需要注意一下，考虑下列两种情况：

Cons1 :constant Integer := 8;

Cons2 :constant := 8;

虽然 Cons1 和 Cons2 都是常量 8，但数据类型是不一样的。Cons1 为 Integer 型的常量；Cons2 是 universal_integer 的常量。

2.7.2 运算符

Ada 下的运算符有以下几类：

逻辑运算符： and、or、xor;

关系运算符： =、/=、<、>、<=、>=

加减运算符：+，-,&

乘除运算符：*,/,rem,mod

最高优先级运算符：**,not,abs

逻辑运算符

逻辑运算符适用于布尔型（即布尔型作为元素的数组）和 Modular 型的数。对于布尔型，and,or,xor 分别执行”与”、”或”、”异或”操作。对于 Mudular 类型的数，进行 2 进制逐位运算，该数的二进制位中 0 表示 False，1 表示 True。具体参见下面的表格：

关系运算符

运算符=、/=、<、>、<=、>=的使用和大部份语言以一样，只要看一下例子即可，下面的有些未讲到的内容如字符串、访问类型，读者可翻看后面的章节：

X /= Y
“” < “A” and “A” < “Aa” –True
“Aa” < “B” and “A” <”A ” — True
My_Car = null; –True，如果 My_Car 被设置为 null
My_Car = Your_Car –True，如果 My_Car 和 You_Car 所指的是同一个对象
My_Car.all = Your_Car.all –True，如果 My_Car 和 You_Car 所指的对象相同
N not in 1..10 –范围测试，如果 N 在 1..10 中，为 True
Today in Mon..Fri
Today in Weekday

加减运算符

+,-对于任何数值 T 都进行加法运算；&则是用于数组及字符串，如 “A”&”B” = “AB”，在第 3 章我们会近一部讲述。

乘除运算符

*,/不用多说，很简单的乘除运算符。注意的是 rem 和 mod。假如 N 表示一个自然数，A,B 有下列的关系：

A = (A/B)*B + (A rem B);

A = B*N + (A mod B);

因此:

最高优先级运算符

** 表示取某数的幂，如 2**2 = 4，3**7 = 2187；

not 则表示非，如 not True = Flase；

abs 为取绝对值，因此 abs(-34) =34;

第 3 章 数组(Array)

3.1 概述(Overview)

数组是一种复合数据类型(composite type)，包含多个同一种类型的数据元素。数组元素的名称用指定的下标表示，这些下标是离散数。数组的值则是一个由这些元素的值构成的合成值(composite value)。Ada 下的数组和其它语言很相似，只是多了一些“宽松”的规定，如无约束数组、动态数组，更加方便了用户。字符串类型 String，Wide_String 等则是数组元素为字符型的数组类型。

3.2 简单数组(Simple Array)

数组类型的一般声明格式如下：

type *array_name* **is array** (*index specification*) **of** *type*;

array_name 是该数组类型的名称；*index specification* 指明该数组类型的元素下标；*type* 是已经定义了的一个数据类型，表示每个元素的数据类型。

通常情况下，数组类型的使用方式和下例相同：

type Total **is range** 1 .. 100; — 声明一个整型 Total,取值范围 1..100。
type Wages **is array** (Total) **of** Integer;– 声明一个数组类型 Wages,该类型有 100 个

Integer 元素。
Unit1 : Wages; — 声明一个 Wages 类型的变量 Unit1，具有 100 个 Integer 元素，下标取值 1 .. 100。

Wages 的声明也可改为

type Wages **is array** (1 .. 100) **of** Integer;

效果是一样的，只是从维护性的角度来讲还是由 Total 来决定 Wages 的下标比较好。

Ada 数组的 *index specification* 是离散数，可以是一个范围，也可以是枚举类型，而不是单纯的一个表示数组大小的数值，这点和 C 、Pascal 有所区别 。数组元素的下标也不需要一定从 0 或从 1 开始，例如：

type First_Name **is**(Bill, Jack, Tom);
type Enrollment **is array** (First_Name) **of** Integer;
Var : Enrollment; — 数组 Var 有 3 个 Integer 元素，Var (Bill)、Var (Jack)、Var (Tom)。

type NO **is array** (-5 .. 100) **of** Integer;
X : NO; — 数组 X 有 105 个 Integer 元素，第一个元素是 X (-5)，最后一个是 X (100)。

3.3 匿名数组(Anonymous Array)

如果嫌上一节中的 Unit1 的声明过程麻烦，也可以直接声明：

Unit1 : **array** (1 .. 100) **of** Integer;

虽然更为精简了，但不推荐这样使用数组。这种数组没有明确的类型，被称为匿名数组(anonymous array),既不能作为子程序的参数，也无法同其它数组混用—-即使声明一样。通常情况下应避免出现这种情况

3.4 无约束数组(Unconstrained Array)

像上面的例子，数组有几个元素在声明数组类型时已经决定，而不是在声明该类型的数组变量时决定，当然这样的好处是明确的知道每个数组会占用多少空间，控制起来也方便。但如同我们先前提及的一样，字符串类型 String，Wide_String 是数组类型，而用户输入的字符串却是不定的，如果它们的长度也预先定好了，使用字符串时无疑会造成内存空间浪费或不够用，这时一般是使用无约束数组—其声明和一般数组类型相同，只是没有规定它的长度—其长度在声明该类型的数组变量时决定。

如 String 类型的声明:

subtype Positive **is** Integer **range** 1..Integer'Last;

type String **is array** (Positive **range** <>) **of** Character;

type Wide_String **is array** (Positive **range** <>) **of** Wide_Character;

< > 表示当声明该无约束数组类型的变量时，需要在( )中指定一个范围。如创建一个变量 Path_Name, 长度为 1024:

Path_Name: String (1 .. 1024);

如果没有指定该数组的大小，如:

Path_Name: String;

是不合法的。

当然范围也无须从 1 开始，Path_Name:String (7..1030) 和上例表示的字符串长度一样,只是下标不同而已。如果有一个函数 Get_Path (Path_Name :**in out** String),将当前路径名赋给参数 Path_Name，只要是 String 类型的参数就可以通用，长度也无须计较—函数出错或不处理不符合长度要求的字符串则是另一回事。

这里强调一下字符串类型的赋值问题，假如将 Path_Name 赋予"/root/",可以在一开始就赋值：

Path_Name :String := "/root/";

这样 Path_Name 的长度就自动成为 6。但如果

Path_Name :String(1..10) := "/root/";

则会引起错误（看上去很正确），因为还缺 4 个字符，应为

Path_Name :String(1..10) := "/root/　";

或采用从理论上讲麻烦点实际上根本不会这么做的方案：

```
Path_Name:String (1..10);
Path_Name(1) := '/';
Path_Name(2) := 'r';
...
Path_Name(6) :='/';
```

这点是和其它语言有所不同，也是很不方便的一点

3.5 动态数组 (Dynamic Array)

数组大小也可以在运行时动态决定，而不是在程序的源代码中就决定。如：

```
X : Positive := Y;
Var : array (1 .. X) of Integer;
```

这样 Var 的大小就由 X 来决定了，X 多大它也多大。只是这样做相当不妙，假设 Y 值是用户输入的，它的大小，甚止于输入的到底是什么，就根本无法确定，如果过大或负数或非 Positive 类型都会产生麻烦。一般情况下还是用在子程序中：

```
procedure Demo (Item :string) is
copy : String(Item'First..Item'Last) := Item;
double: String(1..2*Item'Length) := Item & Item;   --Item'First,Item'Last 等都是数组属性，& 是将两个字符串相连，参见下文。

begin
...
end Demo;
```

这样的话，对于动态数组的操作与用户没有多大关系，保证了安全性

3.6 多维数组(Multidimensional Array)

前面提及的数组用法对多维数组也适用，只是多维数组的声明和元素的表示略有不同。如定义一个矩阵:

```
type Marix is array (1 .. 100,1..100) of Integer;

Var :Marix;--Var 有 100×100 个元素，分别为
Var(1,1),Var(1,2)....Var(100,99),Var(100,100)。
```

上例的 Matrix 是二维数组，多维数组的 *index_specification* 由多个范围组成，每个范围由 ， 隔开。

3.7 访问和设置数组(Access and Set Arrays)

访问数组时，只需在添加在 *index specification* 范围内的下标，如 3.2 中的数组 Unit1:

Unit1 (1) := 190; — 赋予 Unit(1)值 190;
Unit1 (7) := 210;
Unit1 (7) > Unit (1) — 返回 True;

数组的值虽然可以单个元素分别赋予，但这样做明显是开玩笑的麻烦。以上例中的 Var:Enrollment 为例,一般情况下可以如下赋值:

Var := (15,14,13);

结果为 Var (Bill) = 15, Var (Jack) = 14, Var (Tom) =13，( )中的值，按顺序赋给数组里的元素，因此数量也不能少：

Var := (15,14) — 不合法

当数组元素有名称时,例如元素下标是用枚举类型表示，也可以这样赋值：

Var := (Bill => 15, Jack => 14, Tom => 13);

结果与上等同。但如果同时用以上两种表示法是不合法的，如：

Var := (Bill => 15, Jack => 14, 13); — 不合法

如希望相邻的一些元素都是相同值，如 Var (Bill)、 Var(Jack)、Var(Tom)都等于 15，则可以如下：

Var := (Bill .. Tom => 15);

注意是 Bill 和 Tom (包括它们本身）之间的所有元素值为 15。

而希望 Var (Bill) 等于 15，其余都等于 14 时，也可以如下：

Var := (Bill => 15, **others** => 14);

这样 Var (Jack) = Var(Tom) =14。**others** 在有很多元素都要赋予相同值时是相当有用。

如果将 Var 作为常量数组，则在声明该数组时就赋予初使值：

Var： **constant** Enrollment := (Bill => 15, **others** => 14);

最后看一下以下三种情况：

```
type Vector is array (Integer range <>) of Float;
V: Vector(1 .. 5) := (3 .. 5 => 1.0, 6 | 7 => 2.0);– [1]
V := (3 .. 5 => 1.0, others => 2.0); — [2]
V := (1.0, 1.0, 1.0, others => 2.0); — [3]
```

对于 [3]，我们已经学过：V(1)=V(2)=V(3)=1.0，V(4)=V(5)=2.0。 但 [1] 和 [2] 却很特殊，在 [1] 中：V(1)=V(2)=V(3)=1.0，V(4)=V(5)=2.0；在[2]中 V(3)=V(4)=V(5)=1.0，V(1)=V(2)=2.0。[1]和[2] 的情况是 Ada95 新增的。像 [1] 的话，Ada83 会认为它是超过了数组的下标，产生异常 Constraint_Error，而在 Ada95 则是合法的，并且( )里面的值按顺序赋给数组的元素；[2] 在 Ada83 里也是不允许的，Ada95 允许先赋值给 V(3),V(4),V(5)，others 就默认是指 V(1) 和 V(2)。这 3 种情况很容易引起混淆，请读者注意一下。

3.8 数组运算符(Array Operations)

赋值

整个数组的值可以赋给另一个数组，但这两个数组需要是同 1 种数组类型，如果是无约束数组，数组的元素的个数要相等。如：

```
My_Name   : String (1..10) := “Dale      “;
Your_Name : String (1..10) := “Russell  “;
Her_Name  : String (21..30) := “Liz      “;
His_Name  : String (1..5) := “Tim  “;
Your_Name := My_Name;
Your_Name := Her_Name; — 合法的,它们的元素个数和类型相同
His_name  := Your_name; — 会产生错误,类型虽然相同，但长度不同
```

等式与不等式

数组也可以使用运算符 = ， /=,<=,>=,<,> ，但意义与单个元素使用这些运算符不一样。如：

Your_Name = His_Name –返回值为 False

=,/= 比较两个数组中每个元素的值。如果两个数组相对应的元素值—包括数组长度都相等，则这两个数组相等，否则不相等。

Your_Name < His_Name –返回值为 True

<=,>=,>,< 实际上比较数组中第一个值的大小，再返回 True 或 Fasle，数组长度不一样也可比较。

连接符

连接符为 & ,表示将两个数组类型相“串联”。如：

His_Name & Your_Name –返回字符串 “Tim Russell”

```
type vector is array(positive range <>) of integer;
a : vector (1..10);
b : vector (1..5) := (1,2,3,4,5);
c : vector (1..5) := (6,7,8,9,10);
a := b & c;
```

则 a =(1,2,3,4,5,6,7,8,9,10);如果再添一句：

```
a := a (6..10) & a(1..5);
```

则 a =(6,7,8,9,10,1,2,3,4,5)。这种情况也有点特殊，虽然 a(1..5) 看上去接在 a(10) 后面，实际上 a(6..10)和 a(1..5)连接重新组合成一个 10 个元素的数组，再赋该值给 a。这种用法在 Ada83 里是认为不合法的，因为它认为 a(1..5) 是接在 a(10) 之后，而 a 只有 10 个元素，不是 15 个。

3.9 数组属性(Array Attributes)

数组属性有以下一些，A 表示该数组：

A’First 返回 A 的下标范围的下限。

A’Last 返回 A 的下标范围的上限。

A’Range 返回 A 的下标范围，等价于 A’First .. A’Last。

A’Length 返回 A 中元素的数目。

下面的是为多维数组准备的：

A’First(N) 返回 A 中第 N 个下标范围的下限。

A’Last(N) 返回 A 中第 N 个下标范围的上限。

A’Range(N) 返回 A 中第 N 个下标范围，等价于 A’First(N) .. A’Last(N)。

A’Length(N) 返回 A 中第 N 个下标范围所包含元素的数目。

如：

Rectange : Matrix (1..20,1..30);
Length : **array** (1..10) **of** Integer;

因此：

Rectange'First (1) = 1; Rectange'Last (1) = 20;
Rectange'First (2) = 1; Rectange'Last (1) = 30;
Rectange'Length (1) = 20; Rectange'Length (2) = 30;
Rectange'Range(1) =1.. 20; Rectange'Range (2) = 1..30;
Length'First = 1; Length'Last = 10;
Length'Range = 1..10; Length'Length = 10;

4.1 概述(Overview)

记录则是由命名分量(named component)组成的复合类型，即具有不同属性的数据对象的集合，和 C 下的结构(structure)、Pascal 下的记录(record) 类似。Ada 的记录比它们提供的功能更强，也就是限制更少。同时记录扩展(record extension)是 Ada95 中类型扩展（继承）机制的基础，使记录的地位更加突出，关于记录扩展详见 第 6 章 面向对象特性，为了避免重复，本章对此不作介绍。

4.2 简单记录(Simple Record)

记录类型的一般声明如下：

type *record_name* **is**
record
field name 1: type 1;
field name 2: type 2;
...
field name n: type N;
end record;

record_name 是记录类型的名称，一大堆 *filed name* 是记录的成员名称，紧跟其后的是该成员的数据类型。

如下面的例子：

type Id_Card **is**
record
Full_Name : String (1..15);
ID_Number : Positive;

```
Age : Positive;
Birthday : String (1..15);
Familiy_Address : String (1..15);
Family_telephone : Positive;
Job : String(1..10);
end record;
My_Card :Id_Card;
```

一个简单 ID 卡的记录，包含 Full_Name,ID_Number,Age,Birthday，Familiy_Address，Family_telephone，Job 这些成员。

4.3 访问和设置记录(Access and Set Records)

使用记录的成员时，只需在记录和其成员之间用 “.” 隔开即可。如赋予 My_Card 中的变量 Full_Name 值 Jack Werlch:

```
My_Card.Full_Name := “Jack Welch    “;
```

设置记录成员的值和设置数组给人感觉上有点类似,如：

```
My_Card := (“Jack Welch    “, 19830519，45，“Jan 1st 1976  “,
“China        “，8127271，”CEO      “);
```

将 ( )中的值依次赋给 My_Card 的成员。

相同的数据类型的成员一多，无疑会使人不大明了，因此也可以：

```
My_Card := ( Full_Name => “Jack Welch    “,
ID_Number => 19830519,
Age => 45,
Birthday => “Jan 1st 1976  “,
Familiy_Address => “China        “,
Family_telephone => 8127271;
Job => “CEO      “) ;
```

上面两种表示法可以混用，但按位值在有名的值前面：

```
My_Card := ( “Jack Welch    “,
19830519,
Age => 45,
Birthday => “Jan 1st 1976  “,
Familiy_Address => “China        “,
Family_telephone => 8127271;
Job => “CEO      “);
```

但如果为:

```
My_Card := ( Full_Name => “Jack Welch     “,
ID_Number => 19830519,
Age => 45,
Birthday => “Jan 1st 1976   “,
Familiy_Address => “China          “,
8127271;
“CEO       “);
```

则是非法的。

如果几个相同类型的成员，赋予同一数值，也可以：

```
My_Card := ( Full_Name => “Jack Welch     “,
ID_Number | Family_telephone => 19830519,
Age => 45,
Birthday => “Jan 1st 1976   “,
Familiy_Address => “China          “,
Job => “CEO       “);
```

上例我们假设 ID_Number 和 Family_telephone 值是一样的，为 19830519，不同成员间用 | 隔开。

记录类型有时在声明也需要默认值：

```
type Id_Card is
record
Full_Name : String (1..100) := “Jack Welch     “,
ID_Number : Positive := 19830519,
Age : Positive := 45,
Birthday: String (1..20) := “Jan 1st 1976   “,
Familiy_Address :String (1..100):= “China          “,
Family_telephone :Positive := 8127271;
Job : String(1..10) := “CEO       “);
end record;
My_Card :Id_Card;
```

将 Jack Welch 的资料当作了 Id_Card 类型的默认值，My_Card 无须赋值，在声明时已经有了前几个例子中所赋的值。

声明常量记录如下：

My_Card： **constant** Id_Card := (Full_Name => “Jack Welch “,
ID_Number => 19830519,
Age => 45,
Birthday => “Jan 1st 1976 “,
Familiy_Address => “China “,
Family_telephone => 8127271;
Job => “CEO “;)

和创建其它类型的常量类似，只需在该记录类型前添个 **constant**。

4.4 变体记录 (Variant Record)

在讲变体记录前，先介绍一下记录判别式(record discriminant)的概念。判别式（discriminant)以前没接触过，这里先简单提一下它的定义：一个复合类型（除了数组）可以拥有判别式，它用来确定该类型的参数（具体参见 RM95 3.7 Discriminant)。也就是说，一个复合类型创建时可以有一些参数，在接下去声明该类型的变量时，可以通过那些参数的值来改变变量初始化时所占用内存大小、成员数量、取值等等。这一节以及下一节的无约束记录（unconstrained record)的内容都在记录判别式的范围内，至于其它复合类型将在以后讲述。

变体记录，即它的一些成员存在与否取决于该记录的参数。如我们将 Id_Card 这个记录类型扩充一下：

type Id_Card (Age : Positive := 1) **is**
record
Full_Name : String(1..15）;
ID_Number : Positive;
Birthday : String(1..15);
Familiy_Address : String(1..15);
Family_telephone : Positive;
Job : String(1..10);
case Age is
when 1 .. 18 => School_Address : String(1..15);
when 19 .. 60 => Monthly_Income : Integer**;**
Working_Address: String(1..15);
when others => **null;** – 如果 Age 的值不属于 1..60,成员不改变
end case;
end record;
My_Card : Id_Card ;
Your_Card: Id_Card (Age => 20);

上例中，**case** Age ... **end case** 是变体部份，当 Age 值在 1..18 时，动态创建成员 School_Address；当 Age 值在 19..60 时，动态创建成员 Monthly_Income，

Working_Address；当 Age 不在 1..60 时，数据成员不改动。在声明判别式时一般应赋予一个默认值，如上例 Age 的默认值为 1 ，这样声明变量 My_Card 时不带参数也可以，默认参数为 1。但如果 Age 没默认值，上例中的 My_Card 声明是非法的。

因此，记录 My_Card 有 Full_Name，ID_Number，Birthday，Familiy_Address，Family_telephone, Job，School_Address 这些成员，因为 Age 默认为 1; 记录 Your_Card 中 Age 值为 20，因此有 Full_Name，ID_Number，Birthday，Familiy_Address，Family_telephone, Job， Monthly_Income，Working_Address 这些成员。

最后注意一下，变体部份要在记录类型声明的底部，不能在 Job 或其他成员前面—变体记录的变量大小是不定的

4.5 无约束记录(Unconstrained Record)

上面的记录都是受限定的，如 创建 My_Card 后，它的判别式无法再被改动，Monthly_Income，Working_Address 这些成员也无法拥有。但如果 ID_Card 的判别式有了初使值，则还有办法使记录动态改变。

如：

```
type Id_Card (Age : Positive := 1) is
record
Full_Name : String(1..15）;
ID_Number : Positive;
Birthday : String(1..15);
Familiy_Address : String(1..15);
Family_telephone : Positive;
Job : String(1..10);
case Age is
when 1 .. 18 => School_Address : String(1..15);
when 19 .. 60 => Monthly_Income : Integer;
Working_Address: String(1..15);
when others => null;
end case;
end record;
My_Card : Id_Card ； – 以上和上一节的例子一样
....
begin
...
My_Card := (17, “Jack Welch “, 19830519, “Jan 1st 1976 “, “China “,8127271, “CEO “,”Shanghai “);
```

end;

赋值的时候就有了点特殊。My_Card 在程序内部赋值，但与常规赋值不同，它的第一个值是判别式的值，后面才是成员的值—成员数量按照判别式的值动态改变，上例就多了一个 School_Address 成员。这种情况下，编译器会分配给 My_Card 可能使用的最大内存空间。因此将下句接在上句后面：

My_Card := (17, “Jack Welch “, 19830519, “Jan 1st 1976 “, “China “,8127271, “CEO “, 78112 ， ”Shanghai “);

也是可行的。

上面一些记录的例子并不好，成员的数据类型太简单（像生日的数据类型，一般是年月日做成员的一个记录），字符串类型太多，手工赋值的话还要数一下有几个字符，实际中也很少这样的用法，一般还是用函数来赋值。这点请注意一下。

4.6 判别式的其它用途

判别式的另一个用途是动态决定其成员长度，如：

type Buffer (Size:Integer) **is**
record
High_Buffer(1..Size);
Low_Buffer(1..Size);
end record;

这样 Buffer 两个成员的大小就取决于 Size 值，在文本处理中这种用法还是挺好的。

第 5 章 控制结构(Statement)

5.1 概述(Overview)

在 Ada 子程序的 “**is**”和“**end**”之间，是一组有序语句，每句用双引号；结束。这些语句大致可分成三种控制结构：顺序结构，选择结构，循环结构—-如果按照前辈们辛辛苦苦的证明：任何程序都可以只由这三种结构完成。以前我们见过的简单程序都是顺序结构，本章里会介绍一下 Ada 里选择结构的 if、case 语句和循环结构的 loop 语句及其变种，并介绍顺序结构中以前没讲过的 null 和块语句(block statement)，最后是比较有争议的 goto 语句—好像每本教科书上都骂它，说它打破了程序的良好结构。控制结构是一门老话题，Ada95 对它也没作多大改动，语法上和其它语言还是很接近的，但可读性好一点，所有控制结构后都以”**end** something”结束。

5.2 if 语句(if statement)

if 语句判断一个条件是否成立，如果成立，则执行特定的语句，否则跳过这些语句。一般格式如下：

if *condition* **then**
statements
end if;

当 *condition* 的值为 True 时,则执行 *statements*,否则跳过 *statements*，执行“end if”后面的语句。

如果当 *condition* 为 False 也要执行特定语句，则用下面的格式：

if *condition* **then**
statements
else
other statements
end if;

这样当条件不成立时，执行 *other statement*,而不是跳过 if 结构。

下面一种格式是为了多个条件判断而用，防止 if 语句过多:

if *condition* **then**
statements
elsif *condition* **then**
other statements
elsif *condition* **then**
more other statements
else
even more other statements
end if;

使用 elsif 的次数没有限制，注意 elsif 的拼写—-不是 elseif。在这里需要注意一下 *condition* 的值，一定要为布尔型，不像 C 里面，随便填个整数也没事。

下面以简单的一个例子来解释一下 if 语句：

```
000 — filename: ifInteger.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure testrange is
004       Var : Integer;
```

```
005 begin
006      Put (“Enter an Integer number to confirm its range:”);
007      Get (Var);

008  if Var in Integer’First .. -1 then
009           Put_Line (“It is a negative number”);
010  elsif Var in 1 .. Integer’Last then
011 Put_Line (“It is a positive number”);
012  else
013          Put_Line (“It is 0″);
014  end if;
015 end testrange;
```

[007] 输入值 Var;[008]-[014]的语句都是测试 Var 的范围，如是负数则输出”It is a negative number”,正数输出”It is a positive number”,为 0 则输出”It is 0″，以上 3 种情况如果都没产生，则是因为输入值非 Integer 类型或输入值过大，从而产生异常。

5.3 case 语句(case Statement)

如果所要判断的变量有多种可能，并且每种情况都要执行不同的操作，if 语句很显然繁了一点，这时就使用 case 语句，格式为：

```
case expression is
when choice1 => statements
when choice2 => statements
. . .
when others => statements
end case;
```

判断 expression 的值，如符合某项 choice，则执行后面的 statement,如果全都不符合时，就执行 **others** 后的语句。choice 的值不能相同。**when others** 也可以没有，但不推荐这样做，以免有没估计到的情况产生。因此上例也可改成：

```
000 — filename: ifInteger.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure testrange is
004    Var : Integer;
005 begin
006      Put (“Enter an Integer number to confirm its range:”);
007      Get(Var);
008  case Var is
009  when Integer’First .. -1 =>
```

```
010                         Put_Line (“It is a negative number”);
011    when 1 .. Integer’Last =>
012              Put_Line (“It is a positive number”);
013    when others =>
014              Put_Line (“It is 0″);
015  end case;
016 end testrange;
```

与前面的例子完全等效。

5.4 loop 语句(loop Statement)

很多情况下，我们要反复执行同一操作，无疑这时要使用循环结构。循环结构除了最简单的 loop 语句，还有其变种 for 和 while 语句。

最简单的 loop 语句格式为：

```
loop
statements
end loop;
```

当要退出该循环时，使用 exit 或 exit when 语句。exit 表示直接退出该循环，exit when 则在符合 when 后面的条件时再退出。再将 testrange 改动一下，来了解 loop 和 exit 语句。

```
000 — filename: ifInteger.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure testrange is
004    Var : Integer;
005 begin
006  loop
007       Put (“Enter an Integer number to confirm its range:”);
008       Get(Var);
009  case Var is
010  when Integer’First .. -1 =>
011                  Put_Line (“It is a negative number”);
012  when 1 .. Integer’Last =>
013                   Put_Line (“It is a positive number”);
014  when others =>
015                  Put_Line (“It is 0″);
016  end case;
017 exit when Var = 0;
```

```
018  end loop;
019 end testrange;
```

上例循环输出”Enter an Integer number to confirm its range:”，要求输入一个整数；当输入值为 0 时，输出”it is 0″,再退出。

5.5 for 循环(for loop)

for 循环只是 loop 的变种，格式如下：

for *index* **in [reverse]** *range* **loop**
statements;
end loop;
***reverse** 是可选的.

注意一下，*index* 是 for 循环中的局部变量，无需额外声明，只需填入一个合法的标识符即可，在 for 循环内，不能修改 *index* 的值。*index* 的值一般情况下是递增加 1，如 **for** i **in** 1..100,i 的初值为 1，每循环一次加 1，直至加到 100，循环 100 次结束；有时也需要倒过来，如 i 初值为 100,减到 1，则为 **for** i **in reverse** 1..100。但 *range* 中较大值在前则该循环不进行，如 **for** i **in [reverse]** 100..1,循环内语句会略过—即变成了空语句。

仍旧是通过修改 testrange 来了解 for:

```
000 — filename: ifInteger.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure testrange is
004    Var : Integer;
005 begin
006  for i in 1..10 loop
007       Put (“Enter an Integer number to confirm its range:”);
008       Get(Var);
009  case Var is
010  when Integer’First .. -1 =>
011                Put_Line (“It is a negative number”);
012  when 1 .. Integer’Last =>
013                Put_Line (“It is a positive number”);
014  when others =>
015                Put_Line (“It is 0″);
016  end case;
017 exit when Var = 0;
```

```
018  end loop;
019 end testrange;
```

如果不输入 0，在输入 10 次整数后，该程序会自动结束。

5.6 while 循环 (while loop)

while 循环则在某条件不成立时结束循环，其格式为：

while *condition* **loop**
statements
end loop;

condiotion 和 if 语句中的 *condition* 一样，都要求为布尔值，在其值为 False 时，循环结束。

还是老套的 testrange:

```
000 — filename: ifInteger.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure testrange is
004    Var : Integer;
005 begin
006  while Var /= 0 loop
007       Put (“Enter an Integer number to confirm its range:”);
008       Get(Var);
009  case Var is
010  when Integer’First .. -1 =>
011                Put_Line (“It is a negative number”);
012         when 1 .. Integer’Last =>
013                Put_Line (“It is a positive number”);
014  when others =>
015                Put_Line (“It is 0″);
016  end case;
017  end loop;
018 end testrange;
```

这里取消了 **exit when** 语句，由 while 语句来检测 Var 的值。当 Var 值为 0 时，循环结束。

5.7 null 语句(null Satement)

null 语句所做的事就是不做事，大部份情况下就等于没写；但在一些情况下，还是有其作用,如 **if** var > 0 **then null end if**,如果没有 **null**，则属于语法错误，缺少了语句。因此 **null** 用在语法上要求必须有语句，但又不想让程序干什么事的时候。

5.8 块语句(Block Statement)

块语句(block statement),就是以一组语句为单位，当作一个独立的块,也常用在循环中，格式为;

identifier:
[declare]
begin
statements
end *indentifier;*

declare 是可选的，如：

```
Swap:
declare
Temp :Integer;
begin
Temp := V; V:=U; U:=Temp;
end Swap;
```

其中的 Temp 为局部变量，Swap 外的语句无法访问它，Temp 也可写成 Swap.Temp,以此从形式上区分局部变量和全局变量。块语句的用法，还是通过实例来讲解方便：

```
000 — filename: swap.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Swap is
004    V:Integer := 1;
005    U:Integer := 2;
006 begin
007    PutVU1:
008 begin
009          Put(“V is:”); Put(V); New_Line;
010          Put(“U is:”); Put(U); New_Line;
011 end PutVU1;
012    Swap:
013  declare
014        Temp :Integer;
015 begin
```

```
016         Temp := V; V:=U; U:=Temp;
017  end Swap;
018     Put_Line ("After swap");
019     PutVU2:
020 begin
021           Put("V is:"); Put(V); New_Line;
022           Put("U is:"); Put(U); New_Line;
023 end PutVU3;
024 end Swap;
```

通过上面的例子，大家可能感觉没什么意思，块结构可有可无—反正还是按语句的先后顺序执行。但如果它用在循环结构中，则还有点用处：

```
Main_Circle:
begin
loop
statements;
loop
statements;
exit Main_Circle when Found;–* 如果 Found 为 True，则跳出 Main_Circle，而不是该句所在的小循环
statements;
end loop;
statements;
end loop;
end Main_Circlel;
```

这样就能跳出一堆嵌套循环,接下去执行的语句都在跳出的循环后面。

5.9 Goto 语句(Goto Statement)

goto 语句能直接跳到程序的某一处开始执行，使程序结构松散很多，有关编程的教材基本上将它作为碰都不能碰的东西。但在处理异常情况中，goto 还是很方便的—并被有些权威人士推荐；只要别滥用就可以了。Ada 里 goto 语句格式为：

```
<<Label>>
statements;
goto Label;
```

如将上例改为：

```
000 — filename: swap.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
```

```
003 procedure Swap is
004    V:Integer := 1;
005    U:Integer := 2;
006 begin
007   <<restart>>
008     PutVU1:
009 begin
010           Put("V is:"); Put(V); New_Line;
011           Put("U is:"); Put(U); New_Line;
012 end PutVU1;
013     Swap:
014  declare
015           Temp :Integer;
016 begin
017           Temp := V; V:=U; U:=Temp;
018       end Swap;
019     Put_Line ("After swap");
020     PutVU2:
021 begin
022           Put("V is:"); Put(V); New_Line;
023           Put("U is:"); Put(U); New_Line;
024 end PutVU2;
025  goto restart;
026 end swap;
```

快到程序结尾时，又返回到开头<<restart>>处，因此成了无限循环。goto 语句在 Ada 里的限制还是挺多的，如不能跳到 if,case,for,while 里面和其所在子程序外。

第 6 章 子程序(Subprogram)

6.1 概述 (Overview)

一个程序是由一个或更多的子程序组成，以一个主过程(main procedure)为根本，主过程类似与 C 下的 main 函数。子程序包括过程(proceudre)和函数(function)两类,两者区别在于，过程没有返回值，而函数有返回值。

子程序，包括函数和过程，以及下一章所讲述的程序包，是构成 Ada 程序的基础。Ada 提供了一些和 C、Pascal 不同的新特性，如重载、参数模式、分离程序等。

6.2 过程(Procedure)

过程我们以前已经见过了，但那些都是主过程(main procedure)，即程序的主体部体，作用和 C 下的 main 函数相似。一般情况下，Ada 程序的主程序名应当和该主程序所在的文件名相同。过程的声明格式如下：

procedure *procedure_name* (*parameter_specification*);

它的执行部份则为：

procedure *procedure_name* (*parameter_specification*) **is**
declarations;
begin
statements;
end *procedure_name* ;

procedure_name 为该过程的名称；*parameter_specification* 是这个过程所要使用的参数，是可选的；*declarations* 是声明一些局部的新类型、变量、函数、过程；*statements* 则是该过程要执行的语句。

下例创建一个比较两数大小，并输出较大值的过程：

```
procedure compare (A:Integer; B :Integer) is
begin
if A > B then
Put_Line (“A > B”);
elsif A = B then
Put_Line (“A = B”);
else
Put_ine (“A <B”);
end if;
end compare;
```

下例则是完整的程序：

```
000 — filename:comp.adb;
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure comp is
004  procedure compare (A:Integer; B :Integer) is
005  begin
006  if A > B then
007          Put_Line (“A > B.”);
008  elsif A = B then
009          Put_Line (“A = B.”);
```

```
010 else
011          Put_ine ("A < B.");
012 end if;
013  end compare;
014 X,Y:Integer;
015 begin
016     Put ("Enter A:"); Get (X);
017     Put ("Enter B:"); Get (Y);
018  compare (X,Y);
019 end comp;
```

通过上例，对过程的用法应该会有基本的了解了。因为 compare 的执行部分是在 comp 内部，所以我们无须给出单独的 compare 声明，如果要加一句**"procedure** compare (A:Integer; B :Integer)；"，程序还是老样子。声明部份和执行部份一般在使用程序包时分离。其中 Put_Line，Get 也都是预定义的过程

6.3 函数 (Function)

函数和过程也很像，只是它还要有返回值，和 C 很相似，也用 return 来返回函数值。声明格式为：

function *function_name* (*parameter_specification*) **return** *return_type;*

执行部份为：

function *function_name* (*parameter_specification*) **return** *return_type* **is**
declarations;
begin
statements;
return *return_value*;
end *function_name* ;

function_name 为该函数的名称；*parameter_specification* 是这个函数所要使用的参数，是可选的；*declarations* 是声明一些局部的新类型、变量、函数、过程；*statements* 则是该函数要执行的语句。**return** 返回一个数据类型为 *return_type* 的 *return_value*。

将上一节的 comp 程序改动一下：

```
000 — filename:comp.adb;
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
```

```
003 procedure comp is
004  function compare (A:Integer; B :Integer) return Integer is
005  begin
006  return (A – B);
007  end compare;
008 X,Y,Result:Integer;
009 begin
010     Put (“Enter A:”); Get (X);
011     Put (“Enter B:”); Get (Y);
012     Result := compare (X,Y);
013  case Result is
014  when Integer’First..-1 => Put_Line (” A < B.”);
015       when 0 => Put_Line (” A = B.”);
016  when 1..Integer’Last => Put_Line (” A > B.”);
017  when others => null;
018  end case;
019 end comp;
```

上例应该还能说明函数的特点。因为函数是返回一个值，所以在变量声明中赋予初始值时，也可用函数作为所要赋的值，如返回当前时间的 Clock 函数，可以直接在初始化某变量时作为要赋的值：Time_Now :Time:= Clock。与过程一样，在上述情况下，单独的函数声明可有可无。还有一点就是函数、过程的嵌套，上面两个例子就是过程包含过程，过程包含函数，可以无限制的嵌套下去—只要编译器别出错。

6.4 参数模式（Parameter Mode)

在上面的例子中，我们对函数或过程的参数并没做什么修饰，只是很简单的说明该参数的数据类型，但有时候，我们要设置参数的属性—函数和过程是否能修改该参数的值。一共有三种模式：**in**,**out**,**in out**。

in 模式

默认情况下，函数和过程的参数是 **in** 模式，它表示这个参数可能在子程序中被使用，但值不能被子程序改变。如我们写一个略微像样点的 swap 函数：

```
000 filename:swap.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Swap is
004  procedure Swap ( A: in Integer;B: in Integer) is
005       Temp :Integer;
006  begin
007       Temp := A; A:= B; B :=Temp;
```

```
008    end Swap;
009 X,Y:Integer;
010 begin
011    Put (“Enter A:”); Get (X);
012    Put (“Enter B:”); Get (Y);
013    Put (“After swap:”); New_Line;
014    swap(X,Y);
015    Put (“A is “); Put (X); New_Line;
016    Put (“B is “); Put (Y); New_Line;
017 end Swap;
```

上例的程序是无法编译通过的，因为 Swap 的两个参数 A 和 B 都是 **in** 模式，在这个子过程中无法修改 X,Y 的值，也就无法互换这两个值。这时就要使用 **in out** 模式的参数。

in out 模式

in out 模式表示该参数在这个子程序中既可修改又可使用。如只要将上例的[004]改为：**procedure** Swap (A: **in out** Integer,B: **in out** Integer) **is;**该程序便能编译通过，运行正常。

out 模式

单纯的 **out** 模式表示该参数在这个子程序中可以修改，但不能使用。如求和的 add 过程：

```
procedure Add (Left ,Right : Integer; Result : out Integer) is
begin
Result := Left + Right;
end Add;
```

这个过程没问题，但假如还有个输出过程为：

```
procedure PutResult ( Result : out Integer) is
Temp : Integer := Result;
begin
Put (Temp);
end PutResult;
```

则会产生问题，虽然编译可能通过，但结果是不定的，最起码不是所指望的结果，因此 **out** 模式的参数不能赋值给其它变量。单独的 out 模式一般也不会出现。

6.5 调用子程序(Calling Subprograms)

调用子程序最简单的方式就是按照子程序声明的格式调用，如前例的 procedure swap(A:Integer;B:Integer),只要填入的参数是 Integer 类型，便能直接使用 swap (A,B)。注意调用子程序时参数之间用“,”隔开;同类型的参数在声明时也可简化，如 procedure swap(A,B:Integer)。但使用参数时还有下列几种特殊情况.

有名参数（Named Parameter)

我们也可以不按照参数顺序调用子程序。如调用 swap 也可以这样: swap(B => Y, A => X),这时是使用有名参数，明确声明每个变量的值，可以不按照子程序声明中的参数顺序赋值。这样的做的话可读性是好了一点,比较适合参数较多的情况。

如果将有名参数和位置参数一起混用，只需遵守一条规则：位置参数在有名参数前面。因此 swap 的调用有以下几种情况：

swap(x , y);

swap(A => x , B => y);

swap(B => y , A => x);

swap(x, B => Y);

上述四种情况是一样的。下列两种情况是非法的：

swap(y, A => x);—不合法

swap(A => x , y); —不合法

默认参数值（Default Parameter Values)

在声明某个子程序时，我们也可以使参数具有默认值，如下例：

```
000 — filename:putlines.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure putlines is
004  procedure putline(lines: integer:=1) is<> 005 begin
006  for count in 1..lines loop
007         New_Line;
008  end loop;
009  end putline;
010   Line :Integer;
```

```
011 begin
012    Put (“Print Lines :”); Get (Line);
013    putline;
014 end putlines;
```

实际上[012]可有可无，因为调用输出行函数 putline 时，没用参数。而 putline 在声明时赋予了参数 lines 一个默认值 1，这样的话如果调用 putline 没用参数，就以 1 作为参数值。上例也就只输出一个空行。如果 putline 有参数，如 putline(Line),则输出的行数取决于 Line 的数值。

6.6 重载(Overload)

子程序重载

实际上通过先前的一些例子，细心的朋友可能发现，过程 Put 能处理不能类型的参数，不像 C 下的 printf 要选择输出类型，这就涉及到子程序重载：只要参数不一样，子程序可以有相同的名称。如在程序包 Ada.Text_IO 里的两个过程:

```
procedure Put (Item : in Character);

procedure Put (Item : in String);
```

编译器会自动选择合适的子程序，如果 Put 后面的参数为 Character 类型，则调用 **procedure**Put (Item : **in** Character);为 String 类型，则调用 **procedure** Put (Item : **in** String)。这样在用户层上使用子程序简便了许多，很多常见的子程序:Get,Put_Line,Line, Page 都是这样实现的—-虽然在预定义程序包内针对不同参数都有一个子程序与之相对应,用户却只要记住一个名称就可以了。

运算符重载

运算符重载应该说时时刻刻都在—-不同的数据类型都拥有相同的运算符:+,-,*,/等。它的原理和子程序重载是一样的，在 Ada 里，运算符也是通过子程序形式来实现。下面就给出一个“+”和 put 重载的例子：

```
000 — filename: overload.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure overload is
004  type Vector is array (1 .. 5 ) of Integer;
005    a, b, c :Vector;
```

```
006  function "+"(left,right:Vector) return Vector is
007       result : Vector ;
008  begin
009  for i in left'range loop
010          result(i) := left(i) + right(i);
011  end loop;
012  return result;
013  end "+";

014  procedure Put (Item : Vector) is
015  begin
016  for i in Item'range loop
017          Put (Item(i));
018  end loop;
019  end Put;
020 begin
021    a := (1,2,3,4,5);
022    b := (1,2,3,4,5);
023    c := a + b;
024    Put (c);
025 end overload;
```

上例为了简化问题，有些实际中应该有的代码去除了—-如检测所操作数的类型。但其它类型的运算符和重载子程序实现原理也和上例一样。

6.7 分离子程序(Separating Subprogram)

Ada 还允许子程序分成多个部份，而不是像以前的例子一样都塞在同一文件里，如将上例分成两个文件：

第一个文件：

```
000 — filename: overload.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure overload is
004  type Vector is array (1 .. 5 ) of Integer;
005    a, b, c :Vector;
006  function "+"(left,right:Vector) return Vector is
007       result : Vector ;
008  begin
009  for i in left'range loop
010          result(i) := left(i) + right(i);
```

```
011  end loop;
012  return result;
013  end "+";
014  procedure Put (Item : Vector) is separate;
015 begin
016    a := (1,2,3,4,5);
017    b := (1,2,3,4,5);
018    c := a + b;
019    Put (c);
020 end overload;
```

第二个文件：

```
000 –filename:overload-put.adb
001 separate (overload)     — *注意结尾没有分号；
002 procedure Put (Item : Vector) is
003 begin
004  for i in Item'range loop
005       Put (Item(i));
006  end loop;
007 end Put;
```

这个程序和先前那个完全一样，只是"分了家"而已。这样分离程序有时能更好的分解程序的任务，使程序结构更为清楚。注意一下 overload.adb 的[014] 和 overload-put.adb 的 [001]，这两句就是分离子程序的主要语句。

6.8 子程序的内嵌扩展(Inline Expansion of Subprograms)

子程序可以在调用地点被内嵌扩展，以提高程序效率，它的格式为：

pragma Inline(*name*);

如果 *name* 是一个可调用的实体，子程序或类属子程序（见第 1 1 章），那么 pragma Inline 指示在所有调用该实体的地方要求对该实体进行内嵌扩展。这在封装其它语言的接口时，使用的比较多，以提高效率。

第 7 章 程序包(Package)

7.1 概述(Overview)

多子程序封装在一个文件里过于庞大，且分类不清，这时就使用了程序包（package), 作为一种分组机制，将子程序归类封装成独立的单元。Ada 的程序包机制主要受 Pascal 及 70 年代时的软件工程技术影响。当时很主要的一项革新就是软件包的概念。

软件包使一个大程序分离成了多个单元，使软件维护更加方便，可重用性也更高，是结构化编程思想中必不可少的一部份。

软件包并不关心程序是如何运行的，而是在于理解程序是如何构成的以及维护性。Ada 的程序包是定义一堆相关实体（如数据类型、运算符、子程序）的最基本单元，也是使用最常用的编译单元之一。本章里我们介绍程序包的常见用法，更多的内容，如类属程序包，请参见后面的章节。

7.2 程序包的声明(Package Declaration)

程序包一般分为两部份，声明部分和执行部份。声明部份声明子程序、变量等实体，他们通常被称为资源；而执行部份则是声明部分中实体的实现。它们的扩展名一般情况下分别为 ads 和 adb。为了省力点，我们就直接将上一章的 overload 做个处理，将"+"和 Put 封装在程序包内,这样程序包说明如下：

```
000 –filename:overload.ads
001 package Overload is
002  type Vector is array (1 .. 5 ) of Integer;
003  function "+"(left,right:Vector) return Vector;
004  procedure Put (Item : Vector) ;
005 end Overload;
```

从这个例子，我们应该知道了程序包说明的格式：

package *packgae_name* **is**
statements;
end *package_name*;

statemets 就是数据类型、变量、常量、子程序的声明。

7.3 程序包的主体部份(Package Body)

仅仅有说明部份，程序包还没什么用处，还要有它的主体部份，即说明部份的具体实现。主体部份可以包含创建数据类型、子程序、变量、常量。它的格式为:

package body *packgae_name* **is**
statements1;
[begin]
statements2;
end *package_name*;

statements1 是创建子程序、数据类型、变量、常量的语句，一般情况下是说明部份的具体实现；**[begin]**是可选的，它后面的 *statement2* 是程序包的初始化语句，在主程序执行前开始执行。

所以上例 overload 的主体部分为：

```
000 — filename:overload.adb
001 with Ada.Integer_Text_IO; use Ada.Integer_text_IO;

002 package body Overload is
003  function "+"(left,right:Vector) return Vector is
004       result : Vector ;
005  begin
006  for i in left'range loop
007         result(i) := left(i) + right(i);
008  end loop;
009  return result;
010  end "+";
011  procedure Put (Item : Vector) is
012  begin
013  for i in Item'range loop
014         Put (Item(i));
015  end loop;
016  end Put;
017 end Overload;
```

这里我们没有使用可选的 [**begin**] *statement2*，因为没有必要做一些初始化操作。下面是一些注意点：

1.主体部分内的资源不能被其它程序包所引用，会引起编译错误。

2.假如好几个程序包都有初始化语句，执行顺序是不确定的。

3.所有说明部份内的资源可以被其主体部份直接使用，无须 **with** 和 **use** 语句。

7.4 程序包的使用(Using Package)

如同我们先前使用 Ada.Text_IO 一样，使用程序包要使用 **with** 和 **use**。**use** 可以不用，但这样的话使用程序包麻烦很多，如 Put ，就要使用 Ada.Text_IO.Put 这种详细写法；**use** 使编译器自动在软件包中寻找相匹配的子程序和其它资源。现在将 overload 的主程序给出：

```
000 — filename: main.adb
001 with Overload; use Overload;

002 procedure Main is
003    a,b,c:Vector;
004 begin
005    a := (1,2,3,4,5);
006    b := (1,2,3,4,5);
007    c := a + b;
008    Put (c);
009 end Main;
```

编译 main.adb,overload.adb,overload.ads 所得的程序和以前的效果也一样。

一般情况下，with 和 use 语句在程序的首部，但 use 语句也可以在程序内部（with 语句则不能），如：

```
000 — filename: main.adb
001 with Overload;

002 procedure Main is
003 use Overload;
004    a,b,c:Vector;
005 begin
006    a := (1,2,3,4,5);
007    b := (1,2,3,4,5);
008    c := a + b;
009    Put (c);
010 end Main;
```

这种用法很常见，特别是使用类属程序包时。以后我们会见到这方面的其它实例。

使用软件包时要注意一下变量、常量等名称的问题，如果有相同的名称，就需要详细写出所期望的资源名称，如程序包 Ada.Text_IO 的 Put 要写为 Ada.Text_IO.Put。如果在 Overload 的声明部份也加一个变量 a(2,3,5,6,8)，则声明部份为：

```
000 –filename:overload.ads
001 package Overload is
002 type Vector is array (1 .. 5 ) of Integer;
003     a: Vector := (2,3,5,6,8);
004 function “+”(left,right:Vector) return Vector;
005 procedure Put (Item : Vector) ;
006 end Overload;
```

同时我们希望主程序将两个不同的 a 分别与 b 相加，则 Overload 中的 a 要表示为 Overload.a,主程序变为：

```
00 — filename: main.adb
001 with Overload; use Overload;
002 with Ada.Text_IO; use Ada.Text_IO;

003 procedure Main is
004    a,b,c:Vector;
005 begin
006    a := (1,2,3,4,5);
007    b := (1,2,3,4,5);
008    c := a + b;
009    Put (c);
010    New_Line;
011    c := Overload.a + b; — Overload.a 表示程序包 Overload 中的变量 a
012    Put (c);
013    New_Line;
014 end Main;
```

明确资源的位置在很多地方都是需要的，都是为了防止相同名称引起混淆。

7.5 私有类型(Private Type)和有限专用类型（Limited Private Type)

私有类型

到目前为止，我们见过的在程序包内定义的数据类型，只要使用 with 语句，我们都能对它进行任意的处理，没有什么限制。这在有些情况下，会引起麻烦。比方说创建了一套函数库，如果在该函数库里的数据类型能被用户自由处理—-创建新类型，加减乘除运算.....用户又频繁使用的话，会使用户程序相当依赖于这些数据类型，而函数库的创建者为了提高效率或其它什么原因，需要改变这些数据类型—取消，重写或合并，这时用户所写的程序将会遇到很大的麻烦，要么就用旧版的函数库，要么就改写自己的程序——都是不怎么好的做法。在 Unix 下有 C 语言经验的朋友应该对这所谓的“兼容性”深有体会—-系统很无聊的包含了很多只为了兼容性考虑的数据类型、函数，为了移植性，开发稍大一点的软件就多了一些很无谓的“痛苦”。私有类型就是为这种情况产生的：在程序包外，对私有类型数据只能执行 := 和 = 操作，如有其它操作也是程序包内定义的。这样的好处是私有类型不会被滥用，相关的子程序都是程序包创建者定义的，而不是用户。考虑一下下面的账号管理的例子，具体函数实现略，只是象征性的说明一下问题：

```
000 — filename:account.ads
001 package Accounts is
```

```
002 type account is private; — 具体声明在后面
003    My_Account : constant account;
004  procedure withdraw(account:in out account; amount :in Positive);
005  procedure deposit (account:in out account; amount :in Positive);
006  function create(account:in out account;account_id :Positive) return Boolean;
007  function cancel(account:in out account;account_id :Positive) return Boolean;
008  function balance(account: in out account) return Integer;
009 private
010  type account is
011  record
012       account_id : positive;
013       balance : integer;
014  end record;
015 My_Account:constant account := (78781, 122);
016 end accounts;
```

过程 withdraw 和 deposit 对帐号进行取款和存款操作，create 和 cancle 对帐号进行创建和注销，balance 返回当前账号的存款额。实际应用中为了提高效率，这种类型的函数库很有可能需要随时升级，使用了私有类型，用户层的麻烦少了不少。

私有类型 account 先简略地声明为类型 private，它的具体声明跟在保留字 private 后，接下去就跟普通数据类型声明一样。account 类型的数据可以在该程序包外包括在主程序中创建，但对它的操作只能是赋值、相等比较及该函数包定义的操作；在该程序包内，则能对私有类型进行任意操作，就好像它不是私有类型一样。在这个例子里，我们还创建了一个 account 类型的常量 My_Account，注意它的声明方式：先是不完全的声明[3],再在 private 部份给出完整声明[15]。不管怎样，用户只能通过函数说明知道有这么个私有类型，却不能过多的使用它。

有限专用类型

如果嫌私有类型的 := 和 = 这两个默认操作也多余，则使用有限专用类型。如声明上例的 account 为有限专有类型：

type account is limited private;

其它方面与私有类型一样，只是声明略有不同。对这种数据类型的操作只能由该程序包完全定义，没有了其它默认操作。

有时类型定义中还会出现单独的 limited，没有 private，这表示该类型是限制类型，赋值等操作不能作用于该类型变量，但不是“私有”的。

7.6 子单元 (Child Unit)和私有子单元(Private Child Unit)

在一个比较大的软件系统开发中，往往会出现某个程序包变得很大这种情况。对于需要这些新添功能的用户来说，这是好事；而对于其它用户来说，却要花费更多的时间编译更大的程序包，无疑很让人不舒服。在这种情况下，就有了子单元这种处理方法：将逻辑上单一的程序包分割成几个实际上独立的程序包，子单元从逻辑上讲是源程序包的扩展，但却以独立的文件形式存在,如将上例分割：

```
000 — filename:account.ads
001 package Accounts is
002 type account is private;
003    My_Account : constant account;
004  procedure withdraw(account:in out account; amount :in Positive);
005  procedure deposit (account:in out account; amount :in Positive);
006  function balance(account: in out account) return Integer;
007 private
008  type account is
009  record
010       account_id : positive;
011       balance : integer;
012  end record;
013    My_Account:constant account := (78781, 122);
014 end accounts;
```

```
000 –filename:accounts_more_stuff.ads
001 package accounts.more_stuff is
002  function create(account:in out account;account_id :Positive) return Boolean;
003  function cancel(account:in out account;account_id :Positive) return Boolean;
004 end accounts.more_stuff;
```

程序包 accounts.more_stuff 是 accounts 的子程序包。这个例子的实际效果与上例一样，包括作用域等等，只是从形式上来说分成了两个不同程序包,编译时也是分别编译。对于用户来讲，使用上的区别还是有一点：

```
with accounts.more_stuff;
procedure test is
...
begin
...
accounts.more_stuff.create (his_account,7827);
accounts.deposit(his_account,7000);
...
end test;
```

上面虽然只 with 了 accounts.more_stuff，但能使用 accounts 里的资源，默认情况下已作了 with accounts 的工作，如果 accounts 还有其它子单元，则不会自动 with 那些

子单元。 use 和 with 不一样，它不会在 use accounts.more_stuff 时默认也 use accounts：

with accounts.more_stuff; use accounts; use more_stuff;

procedure test **is**
...
begin
create (his_account,7827);
deposit(his_account,7000);
...
end test;

子单元的使用应该说还是和原来差不多。另外注意一下，程序使用程序包时会将程序包内的所有子程序都载入内存，因此有内存浪费现象；如果一个程序包内一部份资源使用频率较高，另一部份资源较少使用，则可以将这两部份资源分成两个子单元，以提高效率，这方面的典型情况就是程序包 Characters.Latin_1，Characters.Handling，具体见 第 13 章。

私有子单元

私有子单元允许创建的子单元只能被源程序包所使用，如：

000 –filename:accounts_more_stuff.ads
001 **private package** accounts.more_stuff **is**
002 **function** create(account:in out account;account_id :Positive) return Boolean;
003 **function** cancel(account:in out account;account_id :Positive) return Boolean;
004 **end** accounts.more_stuff;

这样的话 create 和 cancle 对于用户来说是不可见的，只能被程序包 accounts 所使用。

子单元除了上述的使用方法外，也可直接在其它程序包内部：

```
package  parent_name is
        ...
    package child_name is
        ...
    end child_name;
        ...
end parent_name;
```

第 8 章 面向对象特性(Oriented Object Features)

8.1 概述(Overview)

很早以前，软件开发就要求简化开发的复杂度。其中很重要的一个编程思想就是结构化程序设计，这在 C 、Pascal 这些语言上得到了充分体现。按照这种思想方法，程序的主函数由一些函数构成，这些函数又由更小的函数构成，以此类推，直至将每个任务分解到最小函数单位。但在很复杂的程序中，一般来说是超过 50,000 行源代码的程序，结构化设计所造成的开发成本极高，以至于很难维护。虽然也有反例，如 Unix 以及近来流行的 Linux 在源代码相当庞大且内核基于 C 的情况下还是运行良好，但如果这些操作系统使用的是比 C 更加优秀的语言(不考虑效率问题，一般来说语言越强大执行效率越低），它们将会更加完美。大量的优秀程序员的努力暂时弥补了 C 的不足。

面向对象程序设计(OOP) 作为另一类编程思想，与结构化思想有很大的不同，它将一个系统看成是一个个对象组成，这些对象包含了数据和相关操作。虽然 OOP 提出时间较早，但人们对于它的兴趣，主要还是因为近几年来 C++、Java 这些语言的流行而产生。大部份人即使不很清楚什么叫面向对象，但却知道 OOP 比较适合软件开发，比传统的结构化程序更加容易维护和代码重用。

面向对象的技术实现较为困难，而且让广大用户接受也不那么容易—-起码在 Unix 领域 C 还占据统治地位。

Ada95 新增的一项主要功能就是对面向对象编程的直接支持，在这之前的 Ada83 也已经具有基于对像(object-based)的特性。两者区别在于基于对象没有继承和多态性这两项 OOP 的主要特性。在这一章中，我们就介绍 Ada 面向对象的特性。但很多内容实际上还是先前章节内容的延伸，只是概念上换个样而已，读者应该不会感到很陌生。如第 3 章所涉及的类型派生(type derivation)以及第 11 章的类属单元都属于面向对象的特性，只是在章节安排上分开了。由于本章是“过渡章节”，如果读者想看一下比较完整的面向对象内容讲述，可以参看 VenusIC 下的一些 Ada 文章，如 Object-Oriented Programming with Ada 9X，Using Inheritance in Ada。

8.2 基本概念 (Basic Concept)

面向对象程序设计(OOP)的具体定义很难下，也很容易因此而引起争论，在 Object-Oriented Frequently Asked Questions (OO FAQ) 中就有好几种不同的定义。这里就按照 Grady Booch [1994] 的定义：“面向对象程序设计是程序结构的一种实现方法，在这种方法下，程序由互相协作的对象组成，这些对象是某个类的实例，而这些类又是通过继承关系组成的类的分级结构的成员。”但不管具体的文字定义如何，面向对象的设计(object oriented design)都包含以下几个要素：

- 对象(Object)：包含一定的数据结构和状态的实体。
- 操作(Operation)：作用于对象的行为，如访问和处理对象的状态。

- 封装(Encapsulation)：定义对象和操作，只提供抽象的接口，并隐藏它们的具体实现。

Ada 83 已经支持上述 3 个特性，因此被称为基于对象(oriented-based)的语言；但面向对象程序设计经过十年的发展， Ada95 基于上述要素又增添了以下两个 Ada83 不支持的要素：

- 继承(Inheritance)：通过继承现有类型的性质，创建新的数据类型，而不影响原有数据类型。
- 多态性(Polymorphism)：判定数据类型集合中各类型的区别，使程序可以按照它们的共同特性来书写。

对新特性的需求是由三份报告[Dod 1990]指出的，具体见 Ada Rantinale 的 Part Two-4 Object Oriented Programming– 4.1 Background and Concepts，可以更好的理解继承和多态性。

OOP 的继承从理论上讲是模仿人的思考方法，将对象分类，如：car,bus 这两个数据类型是从数据类型 vehicle 继承而来的，它们作为 vehicle 的一类，自然继承了 vehicle 的特性，同时具有自身独有的特性；而 wheel 却不是 vehicle 的一类，只是 vehicle 的一个组成部份，因此不是从 vehicle 继承而来。同样，vehicle 有一些操作，如 start,reverse, car 和 bus 也继承下来，如果必要，也可加上自己独有的操作，如 drive_at_200mph。但在实际程序中，人们往往忽视了面向对象的内涵，甚止于 C++ 这些语言玩了好几年，也只是用上了面向对象的语法，而没有形成面向对象的思考方法。

OOP 的实现相对来说比较复杂，如果低层实现不好，用户的感受只能是难以使用。为了保证效率同时避免不必要的运行判定，有以下两个问题一定解决：

- 调度(Dispatch)：相对于静态判定，当调度和重新调度时，在程序执行时动态决定所要调用的子程序。
- 多继承(Multiple Inheritance)：从 2 个或更多的父类型中继承成员及操作。

具体的相关内容我们将会在后面章节学习，现在只要有一个大致框架即可。

8.3 继承(Inheritance)

OOP 一项主要特性就是继承（inheritance), 在 1995 年添加了一些功能使 Ada 能支持继承。继承包括两个方面的概念：1.操作的继承；2.数据类型的扩展(type extension)(注意类型扩展和类型派生(type derivation)的区别：前者是指扩展已有的数

据类型，在 Ada 中就是扩展标记记录；后者范围更广，还包括了第二章所讲的创建子类型这些内容）。在 Ada 83 中只支持操作的继承，而没有类型扩展。

继承就是让我们创建新的数据类型作为现有数据类型的扩展，同时这些新类型继承旧类型的所有操作。新的数据类型被称之为子类型或派生类型(derived type），旧的数据类型则称之为父类型(parent type)或基类型（base type)。派生类型可以忽略从基类型继承来的操作，也可以再添加新的操作，但这些新的操作不能适用于基类型。

考虑一下这种情况：假如我们要写一种软件显示很多种几何图形，如正方形 Square、圆 Circle 等等。毫无疑问，它们都有共同的特性：都是图形 Figure,且由无数个点 Point 构成,而点又能通过坐标 (X,Y) 来表示。对于 Figure 的操作，如 Draw,也是构成其它图形的 Draw 的基础（无数个点）。下面就让我们通过 Lovelace 里关于这种情况的一个实例来了解继承，这些数据结构的关系图为：

相对应的程序包说明：

```
000 package Figures is

001 type Point is
002 record
003         X, Y: Float;
004  end record;

005 type Figure is tagged
006 record
007         Start : Point;
008 end record;
009 function Area (F: Figure) return Float;
010 function Perimeter (F:Figure) return Float;
011 procedure Draw (F: Figure);

012 type Circle is new Figure with
013 record
014         Radius: Float;
015    end record;
016 function Area (C: Circle) return Float;
017 function Perimeter (C: Circle) return Float;
018 procedure Draw (C: Circle);

019 type Rectangle is new Figure with
020   record
021         Width: Float;
022         Height: Float;
023 end record;
```

024 **function** Area (R: Rectangle) **return** Float;
025 **function** Perimeter (R: Rectangle) **return** Float;
026 **procedure** Draw (R: Rectangle);

027 **type** Square **is new** Rectangle **with null record**;

028 **end** Figures;

上例已足够来说明继承的特性了，接下来让我们逐句分析：

[001]-[004] 创建一个记录类型 Point，它有两个浮点类型的成员:X，Y。

[005]-[008] 创建一个标记类型(tagged type) Figure，其成员为一个 Point 类型的 start。这里解释一下标记类型，一个数据类型如果要有父类型或子类型(即能被其它数据类型继承)，它就需要是一个标记类型，在它的定义中也需要有关键字 tagged，如上例我们所见的 Figure。标记类型通常的声明格式如下：

type *record_name* **is tagged**
record
field name 1: type 1;
field name 2: type 2;
...
field name n: type N;
end record;

跟一般的记录类型一样，只是多了 tagged。而且对于它的使用和 第四章 数组和记录 里介绍的一模一样，只是它可以作为基类型(base type)，使新类型能继承它的特性。

[009]-[011] 关于 Figure 的三个操作：Area，Perimeter 和 Draw。

[012]-[015] 创建新类型 Circle。注意[012] **type** Circle **is new** Figure **with**，这句表示 Circle 是 Figure 的派生类型，并且在 **with** 之后还包括新的成员。 本例中新成员是浮点型 Radius。因此 Circle 有两个成员：从 Figure 下继承来的 Start 和 新成员 Radius。同时还继承了 Figure 的操作，但上例中我们又重新创建了关于 Circle 的操作 [016]-[018]，因此 Figure 的操作被忽略。假如没有 [016]-[018]，则 Figure 的三个操作也能用于 Circle 类型。创建派生类型的格式一般如下：

type *new_record_name* **is new** *old_record_name* **with**
record
field name 1: type 1;
field name 2: type 2;
...

field name n: type N;
end record;

[019]-[026] 和 Circle 的情况一样。

[027] 创建派生类型 Square，不同于前面两个派生类型，它是从 Rectangle 派生的，并且没有新类型 **with null record**,一切操作也都从 Rectangle 继承。

通过上例大家关于继承应该有所了解了，这些数据类型的变量、操作的用法和前几章所讲的内容一样，没有不同的地方。还有一种是没有成员的记录，如果我们希望基类型没有成员只有操作供派生类型继承，则可以如下声明该类型：

type root **is tagged**
record
null;
end record;

Ada 提供了另一种特殊的格式来处理这种情况：

type root **is tagged null record**;

这点是比较特别。

8.4 多态性（Polymorphism）

Ada 95 里也有类(class)的概念，每个标记类型 T 都有相关联的类型 T'Class。T'Class 是以 T 为根的所有派生类型的集合，包括这些派生类型的派生。如上一节中的 Figure, 与之相对应的类为 Figure'Class,包括了数据类型 Figure，Circle, Rectangle, 和 Square。像 Figure'Class 这样的数据类型通常称为 class-wide type，跟 Class-wide type 有关的编程则常称为 Class Wide Programming(暂想不出一个中文译名来）。

我们在第二章提及 universal_integer，universal_real 能分别和所有整型数和实型数混合运算，因为 universla_integer 可以认为是类 root_integer'Class,universal_real 可以认为是类 root_real'Class,它们包括了所有整型和实型。与此类似，T'Class 出现的地方也能由它所包括的数据类型替换，如一个子程序的参数是 T'Class，则在调用该子程序时实际的参数可以是 T 所有派生类型中的任何一个。 考虑下面的过程 Put_Area，是上一节例子的延续：

procedure Print_Area(T : in Figure'Class) **is**
begin
Put("Area = ");
Put(Area(T));

```
New_Line;
end Print_Area;
```

这个过程的参数有点特别，是 Figure'Class，并且还调用了函数 Area (T)，而在上一节中没有 Area (T : in Figure'Class) 这样的函数。上述的情况称之为多态性(polymorphism)或动态调度(dynamic dispatch)。动态调度就是说在程序执行时决定要真正调用的子程序，并执行它。因此在上述情况中，当执行到 Put(Area(T))，如果参数类型为 Figure，则调用函数 Area (F : Figure)；如果参数类型为 Circle，则调用函数 Area(C : Circle)；如果参数类型为 Rectangle，则调用函数 Area (R : Rectangle); 如果参数类型为 Square，则调用函数 Area (R : Retangle)，因为 Square 是 Rectangle 的派生类型，但没有重新定义自己的操作。

这里就有个问题，Ada 如何动态决定要调用的子程序，并且保证不出错？如同我们上节所讲的，能有父类型和子类型的数据类型要声明为标记类型，之所以要声明为标记类型也正因为多态性的需要：标记类型的变量都有一个不可见的"标记"来告诉编译器它们实际的数据类型，这样当语句执行到类似于 Area (T) 这种情况时，就能判定实际上要调用的子程序。同时，由于 Ada 的编译规则，也保证了有子程序供它动态调用——如果编译器没有找到与所指定的参数的类型相对应的子程序，编译时会报错。

Figure'Class 在子程序中是运行好好的，但在创建数组时却不能使用，如：

```
Var : array (1..20) of Figure'Class;– 不合法
```

因为数组要求每个元素的大小相等，而 Figure'Class 包括的数据类型明显是大小不一的。这条规则也适用于其它需要定长的情况下。但有趣的是，指向 Figure'Class 的访问类型变量可以指向从 Figure 派生的任何一个数据类型(关于访问类型见 第 11 章 访问类型）：

```
type Figure_Access is access Figure'Class;
Var : Figure_Access;
```

这样，我们就能让 Var 指向 Figure'Class 所包括的所有数据类型:

```
Var := new Figure;
Var := new Circle;
Var := new Rectangle;
Var := new Square;
```

更有意思的是，可以创建成员长度不同的数组了：

```
type Figure_List is array(1..10) of Figure_Access;
Var : Figure_List;
```

```
...
Var(1) := new Figure;
Var(2) := new Circle;
Var(3) := new Rectangle;
...
```

这样的话，在创建数组时不必强求每个元素是相同类型，同时也导致使用时比较麻烦。

8.5 封装 (Encapsulation)

前面所见过的数据类型，如 Figure、Circle，它们对于用户来说是公开—用户知道这些数据类型的详细声明，并可随意更改这些类型的变量的值。但这在很多情况下这并不是好事，我们在第五章 子程序和程序包 关于私有类型的内容中已经介绍了这种情况。为了隐藏指定类型的实现细节，Ada 提构了一些机制使这些信息不能访问，称之为封装(encapsulation)。封装改善了程序的可维护性和可靠性。

私有类型要用到保留字 private，这对标记类型也同样适用。由于和私有类型的用法大致相同，我们这里不在罗嗦，就将前面的例子该装一下来说明问题：

```
000 package Figures is

001 type Point is private;
002 type Figure is tagged private;

003 function Area (F: Figure) return Float;
004 function Perimeter (F:Figure) return Float;
005 procedure Draw (F: Figure);

006 type Circle is new Figure with private;
007 function Area (C: Circle) return Float;
008 function Perimeter (C: Circle) return Float;
009 procedure Draw (C: Circle);

010 type Rectangle is new Figure with private;
011 function Area (R: Rectangle) return Float;
012 function Perimeter (R: Rectangle) return Float;
013 procedure Draw (R: Rectangle);

014 type Square is new Rectangle with private;

015 private
016 type Point is
017 record
```

```
018         X, Y: Float;
019  end record;

020 type Figure is tagged
021 record
022         Start : Point;
023 end record;012

024  type Circle is new Figure with
025 record
026         Radius: Float;
027    end record;

028 type Rectangle is new Figure with
029   record
030         Width: Float;
031        Height: Float;
032 end record;

033 type Square is new Rectangle with null record;

034 end Figures;
```

和以前所将的私有类型没有太多的区别。注意一下[002]**type** Figure **is tagged private**，在 private 前面还有 tagged，表示 Figure 是一个标记类型；[006],[010],[014] 是 Figure 的派生类型，并将新增的成员隐藏，需在尾部添一个 private。[015]-[033]private 部分关于这些数据类型的声明和平常一样。

8.6 抽象类型和子程序(Abstract Type and Subprogram)

有时我们希望一个标记类型不能直接用来创建一个对象，只能从它上面派生类型，这种数据类型称为抽象类型(abstract type)。与之相对应的是抽象子程序 (abstract subprogram)，它没有执行部份，因此调用该程序时会被派生的类型忽略。

抽象类型一定要为标记类型，因为只有标记类型才能被扩展。使用它相当简单，只需在“**tagged**” 前加入关键字“**abstract**”。抽象子程序也只需在程序声明后加上“**is abstract**” 。下面是来自 LRM 的例子:

```
package Sets is
subtype Element_Type is Natural;
type Set is abstract tagged null record;
function Empty return Set is abstract;
function Union(Left, Right : Set) return Set is abstract;
function Intersection(Left, Right : Set) return Set is abstract;
```

```
function Unit_Set(Element : Element_Type) return Set is abstract;
procedure Take(Element : out Element_Type; From : in out Set) is abstract;
end Sets;
```

由于这些类型和子程序只是抽象的，无实际含义，因此不能直接使用。如声明抽象类型的变量是不合法的:

```
Var : Set;   — 不合法
```

但派生类型既可以是抽象类型也可以不是：

```
with Sets;

package quick_sets is

type bit_vector is array(0..255) of boolean;
pragma pack(bit_vector);
type quick_set is new Sets.set with
record
bits:bit_vector := (others => false);
end record;

function Empty return quick_set;
function Empty(Element:quick_set) return boolean;
etc.
```

8.7 用户自定义初始化、终止过程（User-controlled Initialization and Finalization)

Ada 95 支持用户定义对象的初始化和终止过程，也就是说，用户可以准确控制对象在它生命周期内内存分配、复制、撤销等操作。这项功能是通过程序包 Ada.Finalization 来提供的：

```
package Ada.Finalization is
pragma Preelaborate (Finalization);

type Controlled is abstract tagged private;

procedure Initialize (Object : in out Controlled);
procedure Adjust (Object : in out Controlled);
procedure Finalize (Object : in out Controlled);

type Limited_Controlled is abstract tagged limited private;
```

procedure Initialize (Object : **in out** Limited_Controlled);
procedure Finalize (Object : **in out** Limited_Controlled);

private
……

end Ada.Finalization;

上面有两个抽象类型 Controlled 和 Limited_Controlled，前者有 3 个操作：Initialize，Adjust，Finalize ;后者则因为是限制类型，不能进行赋值等操作，也就不需要 Adjust。这两个抽象数据类型本身不会调用这些操作，但它们的派生类型却在特定情况下会调用这些过程：

1. 当创建一个受控类型的变量，即分配给该变量内存空间时，Ada 会自动调用用户定义的 Initialize；
2. 当销毁该变量，如局部变量随着子程序的结束而撤销、该变量被赋值时，会调用 Finalize；
3. 当对一个变量进行赋值时，如 A := B，在赋值结束后，调用 Adjust。

如果用户没有自己定义这些操作，Ada 只执行 null 语句，也就等于没执行初始化、终止操作。下面看一个例子：

```
000 --filename:contrlled_number.ads
001 with Ada.Finalization; use Ada.Finalization;
002 package Controlled_Number is
003    type Controlled_ID is private;
004    function Current_ID return Integer;
005 private
006    Number_ID : Integer := 0;
007    type Controlled_ID is new Controlled with record
008       ID : Integer;
009    end record;
010    procedure Initialize (Object : in out Controlled_ID);
011    procedure Adjust     (Object : in out Controlled_ID);
012    procedure Finalize   (Object : in out Controlled_ID);
013 end Controlled_Number;
000 --filename:controlled_number.adb
001 package body Controlled_Number is
002 function Current_ID return Integer is
003 begin
004    return Number_ID;
005 end Current_ID;
006 procedure Initialize(Object : in out Controlled_ID) is
007 begin
008    Number_ID := 1;
009 end Initialize;
010 procedure Adjust(Object : in out Controlled_ID) is
011 begin
```

```
012    Number_ID := 1;
013 end Adjust;
014 procedure Finalize(Object : in out Controlled_ID) is
015 begin
016    Number_ID := 0;
017 end;
018 end Controlled_Number;
000 --filename:controlled.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
003 with Controlled_Number; use Controlled_Number;
004 procedure controlled is
005 Var_1,Var_2: Controlled_ID;-- 创建 Controlled_ID 类型的变量，
                          调用 Initialize(Var_1)，Initialize(Var_2)，所以
Number_ID 两次被设置为 1;
006 begin
007    Put ("After initializing the Var_1 and Var_2, Number_ID is:");
008    Put (Current_ID);
009    New_Line;
010    Var_1 := Var_2;--先销毁 Var_1，所以调用 Finalize(Var_1)，赋值结束，再调用
Adjust(Var_1);
011    Put ("After first assignment,Number_ID is:");
012    Put (Current_ID);
013    New_Line;
014    Var_1 := Var_2;
015    Put ("After second assignment,Number_ID is:");
016    Put (Current_ID);
017    New_Line;
018 end controlled;--- 此时，调用 Finalize(Var_1)，Finalize(Var_2);
```

所以上例中的 3 次输出值均为 1。在与内存有关的访问类型中，这种初始化、终止更加有用，因为可以自动调节变量所占用的内存，但受控类型的效率较低，这在有些场合需要注意一下。

Gnat 还提供了关于链表的类似程序包，在 Ada.Finalization.List_Controller 中，有兴趣的朋友可以看一下。

第 9 章 异常(Exception)

9.1 概述(Overview)

实际程序中我们往往花费大量的精力在一些错误和异常情况的处理上，在 Ada 中，由于有了异常这个机制，相对来说处理异常情况比其它语言要方便一点。一个异常 在程序正常运行时引发，通常表示一个严重的错误，默认情况下会挂起程序，并输出异常的名称及在程序哪里出错，这取决于编译器，也可以手动忽略这个默认情况。如果不想程序在异常引发时按照默认情况处理，也可以设置异常处理，它的内容包含处理哪个异常和当异常引发时怎样处理。

异常 通常表示一个不可预测的错误，如果错误是预计会有的，最好还是别用异常机制。异常使程序效率降低不少，而且程序中过多的异常往往使人费解。因此，一般在子程序遇到严重的错误，使子程序不能顺利执行时使用异常。

9.2 异常声明（Exception Declaration)

在引发或处理异常前，需要声明一个异常，其形式就如同声明数据类型为 **exception** 的变量一样简单：

Singularity : **exception**;

Singularity 只要是一个合法的标识符即可，表示该异常的名称。

引发一个异常也很简单，只需在保留字 **raise** 后跟上要引发的异常名称：

raise *Singularity;*

9.3 预定义的异常(Predefined Exception)

有 5 种异常是 Ada 语言预先定义的（不是在一些预定义子程序中存在的异常，如 End_Error 是 Get 想在到达文件尾部时还要读取文件内容引发的），以下是这 5 种异常的分类介绍。

CONSTRAINT_ERROR

最常见的一个异常，当某变量值超过其类型的取值范围时引发。例如赋给该变量过大或过小的值，除以 0，或使用无效的数组下标。

```
000 –filename:Constraint.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Constraint is
004     x :Integer range 1..20;
005     y :Integer;
006 begin
007     Put(“Enter a number “); Get(y);
008     x := y;
009     Put(“OK”);
010 end Constraint;
```

当用户输入不在范围 1..20 的值时, CONSTRAINT_ERROR 会被引发。由于我们没有对该异常进行处理，程序会按默认处理方式退出，并向用户输出出错信息。其它错误情况也同样处理。

NUMERIC_ERROR

NUMERIC_ERROR 在数值运算无法产生正确结果时引发，如除以 0，无法传递准确的浮点数据时……在 Ada 95 里，重新定义了 NUMERIC_ERROR，和 CONSTRAINT_ERROR 一样。

PROGRAM_ERROR

当程序已运行到某函数尾部，却没有返回值时，引发该异常。

```
000 –filename:Constraint.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Program is
004     z :Integer;
005 function Y(x :Integer) return Integer is
006  begin
007  if x < 10 then
008  return x;
009  end if;
010  end Y;
011 begin
012    z:= y(30);
013 end Program;
```

由于 Y 的参数是 30，所以不返回任何值，就引发了 PROGRAM_ERROR。

STORAGE_ERROR

当内存空间耗尽时，不管是动态创建一个对象还是调用子程序(堆栈空间耗尽）都会产生 STORAGE_ERROR 。

TASKING_ERROR

在任务传递消息时产生，有关 任务 参见第 11 章 并发支持

9.4 异常处理(Exception Handling)

到目前为止，我们还没处理过异常，程序一遇到错误就自动退出，为了使异常有用，就需要写一些异常处理的代码。下面以先前的例子为例，处理异常 CONSTRAINT_ERROR :

```
000 –filename:Constraint.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Constraint is
004    x :Integer range 1..20;
005    y :Integer;
006 begin
007    Put("Enter a number "); Get(y);
008    x := y;
009    Put_Line("OK");
010 exception
011  when CONSTRAINT_ERROR =>
012       Put_Line ("The number should be between 1 and 20″);
013  when others =>
014       Put_Line ("Some other error occured");
015 end Constraint;
```

当输入值不在 1..20 时，不再是以前的直接输出出错信息，而是输出"The number should be between 1 and 20″；如果输入字符串之类的非 Integer 数据，会输出"Some other error occured"。others 表示发生其它错误时如何处理。

有一点要注意一下：异常处理的语句可以是在块结构、子程序及后面讲述的任务等执行部份里面，即在 begin 和 end 之间，但要在执行部份的结尾。

上例也可改为循环输入 Y 值,直至输入符合要求的值：

```
000 –filename:Constraint.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Constraint is
004 begin
005  loop
006  declare
007       x :Integer range 1..20;
008       y :Integer;
009  begin
010       Put("Enter a number "); Get(y);
011       x := y;
```

```
012         Put_Line(“OK”);
013  exit;
014  exception
015  when CONSTRAINT_ERROR =>
016               Put_Line (“The number should be between 1 and 20″);
017  when others =>
018  exit;
019 end;
020 end loop;
021 end Constraint;
```

虽然异常处理不能在普通的循环语句中，但在上例块语句中却可存在。输入值不在 1..20 时，程序直接跳到异常处理语句，又开始重复，当发生其它异常时，就退出循环。

再考虑一下下例：

```
000 –filename:Constraint.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Constraint is
004 x :Integer range 1..20;
005    y :Integer;
006  procedure Level (x,y:in out Integer) is
007  begin
008       Put(“Enter a number “); Get(y);
009       x := y;
010       Put_Line(“OK”);
011  exception
012         when CONSTRAINT_ERROR =>
013            Put_Line (“The number should be between 1 and 20—Level-1 Error”);
014  when others =>
015            Put_Line (“Some other error occured—Level-1 Error”);
016 end Level;
017 begin
018  Level(x,y);
019  exception
020  when CONSTRAINT_ERROR =>
021          Put_Line (“The number should be between 1 and 20″);
022       when others =>
023          Put_Line (“Some other error occured”);
024 end Constraint;
```

这个程序的结果就有点好笑了，如果输入值不在 1..20，先是输出”OK”，再是”The number should be between 1 and 20″。因为 Level 只是接受两个 Integer 数，并不管这两个数的取值范围，只要 Integer 数就可以了，这样在 level 内部程序是正确的；但在主程序中因为 x 的值过大，则认为是发生了错误。如果输入值不在 Integer 的范围内或非 Integer 数据，则在 Level 内部也发生错误，输出”Some other error occured—Level-1 Error”，再在主过程内引起异常，输出”The number should be between 1 and 20″。通过这个例子，需要注意一下，异常处理只处理自己所在执行部份的错误，而且处理范围是从小到大。

9.5 异常传播(Exception Propagation)

为了简化问题起见，考虑一下下面的简单程序：

```
000 –filename:Constraint.adb
001 with Ada.Text_IO; use Ada.Text_IO;

002 procedure Constraint is
003 procedure Level is
004 begin
005 raise constraint_error;
006  exception
007   when CONSTRAINT_ERROR =>
008          Put_Line (“The error in Level occured”) ;
009 end Level;
010 begin
011    Level;
012  exception
013 when CONSTRAINT_ERROR =>
014         Put_Line (“The error in main procedure occured”);
015 end Constraint;
```

虽然在 Level 里产生了异常 constraint_error，但输出只有”The error in Level occured”,主过程不会处理 constraint_error—因为异常只在子程序内部发生，不像前一个例子还波及到主过程的变量。如果想要主过程也处理该异常，则需要使用 raise 语句，后面不跟异常名，在[008]后添一句:

```
raise;
```

则会输出”The error in Level occured”，”The error in main procedure occured”。raise 语句使当前异常再度产生，就如同在子程序中的异常没被处理过一样，于是 Level 执行完以后，还是有个异常 CONSTRAINT_ERROR 等着主过程来处理。因此假如在[014]后也添一句 raise, 则按上述两种方式处理异常后还会按照默认方式来处理 CONSTRAINT_ERROR—输出出错信息，挂起程序。

传播异常实质上就是再度产生当前的异常，在需要多个子程序分别知道该异常并各自处理一次时，还是很有用的。

9.6 异常的作用域(Exception Scope)

用户定义的异常和预定义异常在处理上是一样的，但在作用域上却存在较大差别。考虑下例：

```
000 –filename:scope.adb
001 with Ada.Text_IO; use Ada.Text_IO;

002 procedure Scope is
003  procedure Level is
004       Level_Error:exception;
005  begin
006  raise Level_Error;
007  end Level;
008 begin
009    Level;
010  exception
011  when Level_Error =>
012           Put_Line (“The error in main procedure occured”);
013 end Scope;
```

这个程序看上去是正确的，实际上是不合法的。一个用户定义的异常作用域被局限于该过程内，因此上例中主过程访问 Level_Error 是不合法的；一些预定义程序包内的异常之所以能被用户所处理，因为这些异常在程序包的声明部份里定义了，用户是可见的。

一种解决方案是使用 others,如将上例[011]-[012]改成：

```
011 when others =>
012 Put_Line (“The error in main procedure occured”);
```

则程序会输出”The error in main procedure occured”,others 不管具体是哪个异常，只要有异常就处理，因此上例可行。

还有一种情况如下：

```
000 –filename:scope.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 procedure Scope is
003 Level_Error:exception;
```

```
004  procedure Level_1 is
005  begin
006  raise Level_Error;
007  end Level-1;
008  procedure Level_2 is
009       Level_Error:exception;
010  begin
011  level_1;
012  exception
013  when Level_Error =>
014            Put_Line (“The error in level_2 procedure occured”);
015  end Level_2;
016 begin
017    Level_2;
018  exception
019  when Level_Error =>
020         Put_Line (“The error in main procedure occured”);
021 end Scope;
```

上例的输出结果为”The error in main procedure occured”。Level_1 产生的 level_error 是主过程 Scope 里的 Level_Error,而 Level_2 里的 Level_Error 只是它的局部异常，与主过程里的异常只是同名而已，实际是不一样的。 为了区分起见，可以将[008]-[015]改为：

```
008 procedure Level_2 is
009    Level_Error:exception;
010 begin
011    level_1;
012  exception
013  when Level_Error =>
014          Put_Line (“The error in level_2 procedure occured,it’s
Level_2.Level_Error”);
015  when Scope.Level_Error =>
016          Put_Line (“The error in level_2 procedure occured,it’s
scope .Level_Error”);
017 end Level_2;
```

这样的话明显区分了全局异常和局部异常，输出”The error in level_2 procedure occured,it’s scope.Level_Error”。

9.7 消除异常检测(Suppress Exception Checks)

只要稍有些经验的朋友们都知道，像异常这种东西往往会使程序效率降低不少。一般来说，产生异常的代码是额外加入到程序中的，或者通过硬件检查机制如软件中断来实现。

为了提高效率，和满足有些用户需要，Ada 允许通过 pragma suppress 来取消一些检测；但在编译程序时，编译器可能就自动取消了一些异常检测，这取决于具体实现。

suppress 有很多选项，这取决于编译器实现。

CONSTRAINT_ERROR 有下列一些检测：

```
pragma suppress (access_check);
pragma suppress (discriminant_check);
pragma suppress (index_check);
pragma suppress (length_check);
pragma suppress (range_check);
pragma suppress (division_check);
pragma suppress (overflow_check);
```

PROGRAM_ERROR 只有一个检测：

```
pragma suppress (elaboration_check);
```

STORAGE_ERROR 也只有一个检测：

```
pragma suppress (storage_check);
```

pragam suppress 也可指定某个数据类型，如：

type employee_id **is new** integer;

pragma suppress (range_check, employee_id);

这样，employee_id 的范围检测会被忽略，其它数据类型则不受影响。

第 10 章 访问类型(Access type)

10.1 概述(Overview)

有时候需要一个对象不是存储一个数值，而是存储一个指向其它对象的引用，即该对象在内存中的地址，为此 Ada 提供了访问类型的机制，等价与 C 下的指针。访问类型也可用来创建动态大小的数据结构，如数、链表等。访问类型由于直接访问内存，与指针一样在安全性上令人担忧，使用时要多注意一下，虽然 Ada 在这方面比 C 要好多了，但仍有不尽人意的地方。

我们先介绍常见的访问类型的用法：创建、使用和回收内存，接下去则是高级点的话题：通用访问类型和指向子程序的访问类型。访问类型的基本用法比较简单，而略微复杂点的用法，如记录成员是访问类型变量等，也都基于基本的一些知识，在这本教材里就暂且忽略不提。

10.2 创建和使用访问类型（Create and Set Access Type)

创建访问类型

如同声明其它数据类型一样，声明访问类型按照下面的格式：

type *new_type_name* **is access [all]** *type_name*;

new_type_name 是该访问类型名称，*type_name* 是其它数据类型。**all** 为可选项，具体见下。

如：

```
type List_Node is
record
Data : Integer;
end record;
type List_Node_Access is access List_Node;
Current :List_Node_Access;
```

创建了访问类型 List_Node_Access，及其变量 Current。变量 Current 的初始值为 **null**，也就是 Current 不指向任何对象，访问值为 null。

访问类型的基本操作

有关访问类型最常见的一个操作是创建一个对象，返回一个访问值指向这个对象，如同 C 下的 malloc,Pascal 和 C++ 的 new，Ada 也有一个操作 new,如：

```
Current := new List_Node;

Root    := new List_Node;
```

表示 Current 访问一个 List_Node，Root 访问另一个 List_Node，它们都指向 List_Node 类型的数据在内存中的地址。这些新创建的对象在内存中所占用的空间，不会像整型这些变量一样随着子程序的退出自动消失，回收这些内存空间见 **10.3 无用存储单元收集** 。

如果要访问真实的对象，而不是上述的引用，可以用“ .” 访问所指向对象的实际值，如:

Current.Data := 1;

Root.Data := 2;

表示赋 1 给 Current 所指向的对象的成员 Data ，赋 2 给 Root 所指向的对象的成员 Data。

访问类型的变量也可以不指向任何对象，而是 **null**，只需赋值为 **null** 即可：

Current := **null**;

因此也可以用 = 、 /= 来比较变量值是否为 **null**，以此来得知某访问类型的变量是否指向一个对象。如果没有指定其它初始值，访问类型的变量被创建时的默认值也为 **null**。

访问类型变量的成员值也可以初始化,如：

Current := **new** List_Node’(1);

Root := **new** List_Node’(Data =>2);

这样的话 Current.Data 的值为 1，Root.Data 的值为 2。假如所指向对像有多个成员，初始化它的值时每个成员都要初始化。

访问类型的变量虽然很有用且高效，但也比较危险。为了提高安全性，Ada 有以下一些限制:

- 所有访问类型变量的初始值为 null（除非你作了其它一些初始化工作）。
- 对访问类型变量所指向的对象进行操作前，先检查该变量值是否为 null，如为 null，产生异常 Constraint_Error。
- 通常访问类型的变量所指向的对象是动态创建的，但也可以通过在访问类型声明中添加“all”，使该类型能指向所有给定类型的对象，这称之为通用访问类型 (general access type)。通用访问类型一个用途是连接 C 和 C++ 的程序，C 和 C++ 中的指针本质上和 Ada 的通用访问类型一样；另一个重要的用途是面向对象编程。
- 对访问类型变量的算术运算是禁止的，这点和 Pascal 和 Java 相同，但和 C 与 C++ 不一样。

Ada 编译器会在编译时决定有些检查是否去除，以优化程序。如果认为程序不会出问题，也可以手动去除这些检测。

最后，我们需要强调一下基本概念：访问类型的变量只是指向某个对象，即内存中某个空间，它的值为访问值(access value)。

10.3 自引用数据结构(Self Referencing Data Structures)

在其它语言涉及指针和引用时，往往用链表、树这些数据结构来讲解。我们接下去将会接触链表，至于其它相关数据结构原理上也差不多，不懂的朋友请自己翻阅有关数据结构的书。但不管怎样，这些数据结构的各节点之间按照一定的规则相互连接，每个节点既有要处理的数据部分，又包含了指向其它节点的访问值，可以认为是自引用数据结构。下面看一下一个简单的链表实现：

```
type List_Node; — 不完全的类型声明
type List_Node_Access is access List_Node;
type List_Node is
record
Data : Integer;
Next : List_Node_Access; — 链表中下个节点
end record;
Current, Root : List_Node_Access;
```

注意上例的 List_Node，List_Node_Access 这两个数据类型是相互依赖的。依照 Ada 的规定，声明某个对象后才能使用该对象，但在上述情况下这是不可能的。因此 List_Node 开始时要不完全声明，用来提示 Ada 编译器该类型的声明稍后出现。

在链表中有一个常令初学者头疼的问题就是如何使各节点串联起来构成一个更复杂的结构。我们在上例的 List_Node 中包含了一个元素 Next,用来指向另一个节点。假如使 Root 的 Next 指向 Current,可以：

```
Root.Next := Current;
```

就这么个简单的一句，令很多初学者感到头晕。Lovelace 这份教材里为了便于读者理解，认为上述赋值可这样理解：“改变 Root.Next,使 Root.Next 访问的东西正是 Current 当前所访问的。”现在将原著中的说明图“借用”一下,以说明问题：

还有两种赋值方式比较令人迷惑，即有没有”.all”,比较一下这两种方式：

```
Root.all := Current.all; — 语句(1).

Root := Current; — 语句(2).
```

语句(1) 表示将 Current 所指向的对象的所有值赋给 Root 所指向的对象;语句(2) 不改变底层的节点，改变 Root 的访问值，使 Root 和 Current 指向的对象相同。 下面是图解：

清楚以上的概念后，下面给出一个链表的简单例子：

```
000 — filename:link.adb
001 with Ada.Text_IO; use Ada.Text_IO;
002 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

003 procedure Link is
004  type List_Node;
005  type List_Node_Access is access List_Node;
006  type List_Node is
007  record
008       Data : Integer;
009       Next : List_Node_Access;
010  end record;
011   Current, Root : List_Node_Access;
012   input :Integer;
013 begin
014    Root := new List_Node'(Data =>0, Next=>null);
015   Current := new List_Node;
016   Root.Next := Current;
017  loop
018      Put ("Input Number:");
019      Get (Input);
020       Current.Data := input;
021  exit when input =0;
022      Current.Next := new List_Node;
023      Current := Current.Next;
024  end loop;
025   Current := Root;
026  while Current /= null loop
027      Put (Current.data);
028      Current := Current.Next;
029  end loop;
030 end Link;
```

————————————link.adb————————————————

该链表的第一个节点的 Data 值预先初始化为 0,[014]。[017]-[024]循环输入 Integer 数，并将输入值先赋予 input,再赋予 Current.Data；同时不断创建新的 Current 节点，并将 Current 指向下一个节点，直至 input 值为 0 循环结束或者内存空间耗尽产生异

常 STORAGE_ERROR，这时最后一个节点的 Data 值为 0。[025] 将 Root 赋予 Current,使 Current 指向节点的首部。[026]-[029]从第一个节点开始，输出 Current.Data 值，直至 Current 为 null 循环结束。输出结果的首尾都是 0,中间则是用户输入的非 0 值。

其它数据结构如树、双向链表以及它们的排序等等，还有更复杂的节点，如节点包含字符串等乱七八糟的数据类型等等，所涉及的语言要点也就是在上面了，对于它们的详细讲述超过了这本教材的范围，而且也没有多大必要—-反正是依样画葫芦。

最后强调一下，由于很容易将节点的连接搞错，一般情况下是用子程序来处理节点，将访问类型做为参数，如将上例的 [026]-[029]这段代码封装成过程 Put_List_Node：

```
procedure Put_List_Node(Current: in out List_Node_Access) is
begin
while Current /= null loop
Put (Current.data);
Current := Current.Next;
end loop;
end Put_List_Node;
```

这样再调用 Put_List_Node:

```
Put_List_Node (Current);
```

效果与原来的例子一样。但这样做，管理这些复杂的数据结构更加简单，问题也更容易找到。

10.4 无用存储单元收集(Garbage Collection)

访问类型的变量所占用内存空间能动态改变，但这样做的问题也相当明显：这些占用的内存空间不会被自动回收。假如将上例的[017]-[024]也封装成一个子程序，虽然也可行，但在子程序内创建节点所占用的内存空间在子程序退出后不会被自动回收。假如是稍大点的程序，如文件管理器(对于读取文件也一般采用链表结构），如果不注意内存的使用，一不小心内存耗尽都有可能。为此回收内存在实际程序中是必不可少的，Ada 中这称之为” 无用存储单元收集(garbage collection)”,和 C 下的函数 free(),C++下的 delete 做的事一样，但略微麻烦点。

Ada 提供了一个类属过程来释放一个对象所占用的内存空间，该过程在类属程序包 Ada.Unchecked_Deallocation 内定义：

```
generic
```

type Object (<>) **is limited private**;

type Name **is access** Object;

procedure Ada.Unchecked_Deallocation (X : **in out** Name);

需要传递参数有两个：一个数据类型 object,以及它的访问类型 Name。出于习惯，我们将回收内存的函数命名为 Free：

procedure Free **is new** Ada.Unchecked_Deallocation (List_Node, List_Node_Access);

或

procedure Free **is new** Ada.Unchecked_Deallocation (Object=>List_Node,Name =>List_Node_Access);

这样的话，我们就有了处理无用内存的过程 Free,将上一节中的节点释放，只需：

Free (Current);

该过程返回后，Current 值为 null,先前被 Current 所占用的内存空间也被释放。

至此，我们已知道了如何回收内存，但也要注意 Free (Current) 这看上去毫无问题的语句：假如 Current 和 Root 当时指向同一个对象，释放 Current 所访问的内存空间，也就释放了 Root 所访问的内存空间，再访问 Root 所指向的对象时，结果是不可预测的，视当前内存的变化而定—-如果释放 Current 后，紧接着马上访问 Root 所指向的对象，可能也不会出问题。

最后提一下自动回收内存的问题。像 Java , SmallTalk 这些语言回收内存空间是自动进行的，对于用户来收相当方便，但因为自动收集无用存储单元很难实现得很好，再加上开销也很大，有时可能导致结果无法预测。Ada 允许这项特性，但不是必需的。在用 Ada 编译器创建 Java 代码时，则会使用自动回收内存这项特性；编译器厂商也可自己决定使否使用该特性，如提供编译选项指定使用自动回收内存，这时上述的过程 Ada.Unchecked_Deallocation 在程序中没有效果。

10.5 通用访问类型(General Access Type)

我们先前所学的访问类型的变量所指向的对象是动态创建的，Ada 还提供了通用访问类型的机制，使访问类型的变量能指向任何一个给定类型的数据，如指向一个 Integer 类型的变量。在进一步讲解前，我们看一下下面的例子：

```
001   package Message_Services is
```

```
002      type Message_Code_Type is range 0..100;
003      subtype Message is String;
004      function Get_Message(Message_Code: Message_Code_Type)
005         return Message;
006      pragma Inline(Get_Message);
007   end Message_Services;
008   package body Message_Services is
009      type Message_Handle is access constant Message;
010      Message_0: aliased constant Message := "OK";
011      Message_1: aliased constant Message := "Up";
012      Message_2: aliased constant Message := "Shutdown";
013      Message_3: aliased constant Message := "Shutup";
014      ...
015      Message_Table: array (Message_Code_Type) of
016         Message_Handle :=
017           (0 => Message_0'Access,
018            1 => Message_1'Access,
019            2 => Message_2'Access,
020            3 => Message_3'Access,
021            -- etc.
022           );
023      function Get_Message(Message_Code: Message_Code_Type)
024         return Message is
025      begin
026         return Message_Table(Message_Code).all;
027      end Get_Message;
028   end Message_Services;
```

看到这个例子，大家先别头晕，让我们来仔细分析，因为上例的“含金量”较高，明白了上例，本节也就 over 了。Message_Services 的声明部份应当没什么问题，都是以前所学的语法；执行部份一开始就有了问题：

[009]**type** Message_Handle **is access constant** Message，这表示创建访问类型 Message_Handle，该类型的变量所指向的数据类型为 **constant** Message，也就是 Message 类型的常量。

[010]-[014]是创建 Message 类型的常量 Message_0 等，但多了 **aliased**，这是说这些对象有了属性 Access 和 Unchecked_Accesss，能标示它们在内存中的地址，同时也可以将它们当成普通字符串常量使用，如 Put(Message_0)。有了 **aliased** 后，Message_0 有了 Access 和 Unchecked_Access 两个属性，如 Message_0′Access 和 Message_0′Unchecked_Access，表示返回 Message_0 的访问值，即在内存中的地址。这样如果还有一个变量 Message_Ptr : Message_Handle，因为 Message_Ptr 是访问类型的变量，所以可以 Message_Ptr := Message_0′Access，这样 Message_Ptr 就指向了 Message_0，通过 Message_Ptr 就能间接操作 Message_0，如 Put(Message_Ptr.all) 是输出 Message_Ptr 所指向的对象的内容，也就输出了 Message_0 的内容，等价与 Put(Message_0)。Access 和 Unchecked_Access 的区别在于：Access 产生的访问值在编译时有些限制，并且该访问值的生存期不能长于所指

向的对象，这样防止全局的访问值在所指向对象释放后还占用一定内存空间。Unchecked_Access 则没有这些限制，这就是程序员的责任了。

[015]-[027] 都是 Message 们的使用，也就是由 **aliased** 产生的 Access 属性的用法。

上例是访问类型指向只读数据，如果不指向常量，则用“all”，如：

```
 type Int_Ptr is access all Integer;
IP: Int_Ptr;
I: aliased Integer;
  ...
IP := I'Access;
```

先创建访问类型 Int_Ptr，能指向 Integer 类型数据。接着创建 Int_Ptr 的变量 IP 和 aliased Integer 变量 I，最后将 I 的地址赋给 IP。

10.6 访问参数（Access Parameter)

当我们开发面向对象系统时，有时需要使访问值能像标记类型一样被动态调度。Ada95 提供了新的机制以解决这问题：提供新的参数模式，称为访问参数。以前我们讲参数模式时只讲了 in,out,in out 模式，子程序也可以有访问参数，如：

```
 procedure Get(Agent : access Occupant; Direct_Object : access
Occupant'Class);
```

- access 后面是一个标记类型时，输入参数（Agent)必须是给定数据类型（Occupant)的访问值。更主要的是，该过程可以被忽略，如果还有一个 Get 的参数是 Occupant 的派生类型，则被访问的对象动态决定要执行的子程序。
- access 后面跟着是类的话，则输入的参数(Direct_Object)是给定数据类型（Occupant’Class)或其派生类型的访问值。这种情况下，我们不需要在这参数上动态调度，派生类型会做这工作。这样，访问参数使我们能接受很大范围的数据类型，而不是指定的访问类型。

10.7 指向子程序的访问类型（Access Type to Subprogram)

访问类型的变量不仅能指向其它变量和常量，也能指向一个子程序。通过该子程序的 Access 属性，我们可以将其在内存中的地址赋与某访问类型变量，从而间接调用该子程序，如下例：

type Trig_Function **is access function**(F: Float) **return** Float;
T: Trig_Function;
X, Theta: Float;

上面的语法可能有点令人奇怪。我们可以将第一句理解为，Trig_Function 指向一个函数，该函数的参数为一个 Float 类型变量（函数参数在“( )”中列出），并且返回值的类型为 Float。这样 T 就是能指向这种函数的访问类型变量。接着：

T := Sin’Access;

Sin 函数具体见 第 16 章 数值计算，现在可认为它是 **function** Sin (X : Float) **return**Float。这样，T 就指向了 Sin 函数。我们就可以间接调用 Sin:

X := T(Theta);
或者：
X := T.all(Theta);

和访问类型的其它使用一样，”.all” 不是必需的，但没有参数时需要它。

通过访问类型变量指向一个子程序，我们还能实现一组操作按顺序先后完成,如：

type Action **is access** procedure;
Action_Sequence: **array**(1 .. N) **of** Action;

... — build the array

– and then obey it
for I in Action_Sequence’Range **loop**
Action_Sequence(I).all;
end loop;

这样在循环内就按顺序执行 Action_Sequence 的元素所指向的所有过程，这在加工工业实现顺序控制时比较常见。

按照上面的原理，记录的成员等也可以指向某一子程序，由于道理一样，我们这里就省略不提了。

第 11 章 类属单元(Generic)

11.1 概述(Overview)

代码重用是多年来软件开发一直强调的重点，也是程序员们的一个希望。但在 C 这些语言中，由于语言的先天不足，代码重用是有限的。Ada 里提供了类属单元(Generic unit)的功能（和 C++的模板 template 差不多),使我们有可能创建更为通用的子程序或程序包。

一个类属单元可以是程序包或子程序，允许执行的运算不依赖特定数据类型。比方说一个是类属单元的 Swap 函数，它可以接受 Integer，Float 等各种数据类型的参数，而无需为不同数据类型的参数各写一个 Swap。使用一个类属单元需要设置它的特定数据类型，这个过程称之为实例化(instantiation)，如使用上面所说的 Swap 函数时，要配置它将要处理的数据类型。

11.2 类属子程序 (Generic Subprogram)

类属子程序即子程序为类属单元。下面通过一个 Swap 的例子来讲解：

```
000 — filename:Swap.adb
001 with Ada.Integer_Text_IO;use Ada.Integer_Text_IO;

002 procedure Swap is
003 generic
004 type Element_Type is private;
005 procedure Generic_Swap(Left, Right : in out Element_Type);
006 procedure Generic_Swap(Left,Right:in out Element_Type) is
007       Temporary : Element_Type;
008  begin
009       Temporary := Left;
010       Left := Right;
012       Right := Temporary;
013 end Generic_Swap;
014 procedure Swap is new Generic_Swap(Integer);
015    A, B : Integer;
016 begin
017    A := 5;
018    B := 7;
019    Swap(A, B);
020 end Swap;
```

[003]-[004] 声明一个类属数据类型 Element_Type,[005]声明一个类属子程序 Generic_Swap，它的参数是类属类型(generic type)，[006]-[013] 是 Generic_Swap 函数的具体实现。但这些还不够，Element_Type 并不是什么整型、浮点型等具体数据类型，只是抽象意义上的类属类型，为形式参数。因此还需要[014] 创建实际的子程序 Swap，这里的 Integer 就是一个实际参数,表示 Swap 能接受 Integer 类型的参数，这个步骤称之为实例化。当然为了可以交换浮点等其它数据类型，也可以再添加：

```
procedure Swap is new Generic_Swap(Float);

procedure Long_Swap is new Generic_Swap(Elementary => Long_Integer);
```

...

基于类属子程序的实际程序的数目从理论上讲没有限制。重名的子程序在使用时由编译器自动区分。从这个例子，大家应该明白了类属单元的好处–实际应用中省了很多重复的工作。

11.3 类属程序包(Generic Package)

如果一个程序包内的子程序都需要成为类属子程序，并且实例化时一次性使整个程序包的子程序能处理所指定的数据类型，使用类属程序包就比较适合。将上例 swap 改动一下：

```
000 — filename:generic_swap.ads
001 generic
002  type Element_Type is private;
003 package Generic_Swap is
004  procedure Generic_Swap(Left, Right : in out Element_Type);
005 end Generic_swap;

000 — filename:generic_swap.adb
001 package body Generic_Swap is
002  procedure Generic_Swap(Left, Right : in out Element_Type) is
003       Temporary : Element_Type;
004  begin
005       Temporary := Left;
006       Left := Right;
007       Right := Temporary;
008  end Generic_Swap;
009 end Generic_Swap;

000 — filename:integer_swap.ads
001 with generic_swap;
002 package Integer_Swap is new Generic_Swap(Integer);

000 — filename:swap.adb
001 with Integer_Swap;use Integer_Swap;
002 procedure swap is
003     A, B : Integer;
004 begin
005     A := 5;
006     B := 7;
007     Integer_Swap(A, B);
008 end swap;
```

上例的确是有点小题大做了，但说明问题还可以。注意一下 generic_swap.ads 的[001]-[002],在程序包声明前面的位置声明类属类型，其后的子程序使用这个类属类型，其它方面和普通程序包一样。integer_swap.ads 是新接触的，[002] 创建程序包 Integer_Swap,是 Generic_Swap 的 Integer 版，即 Integer_Swap 是 Generic_Swap 的一份拷贝，但类型 Element_Type 由 Integer 替换。Element_Type 也可以由其它数据类型替换，只需按照上面的格式创建程序包即可。需要注意一下 integer_swap.ads 的第一句 with generic_swap,不能再加上 use generic_swap，因为类属程序包是不可用的。

上例中还无法见出类属程序包的好处，因为只有一个 generic_swap 过程，但假如程序包内有很多子程序的代码需要共用，则相当方便。预定义的很多程序包就是这样实现的，例如我们先前遇到的 Ada.Integer_Text_IO 之类的程序包，如果也是每个子程序重写一遍，工作量简直不敢想象，而它们的实现仅是：

```
with Ada.Text_IO;
package Ada.Integer_Text_IO is
new Ada.Text_IO.Integer_IO (Integer);
```

其它整型输出如 Ada.Short_Integer_Text_IO 也都类似，真正的需要实现的只是 Ada.Text_IO.Integer_IO。其它的浮点型输出、模数输出也都建立在各自的类属程序包上。

11.4 类属参数(Generic parameter)

类属参数有 3 类：类型参数(type parameter)，值参数(value parameter)，子程序参数(subprogram parameter)，下面我们就分类介绍。

11.4.1 类型参数(Type Parameter)

类属类型吸引人的地方大家都有所体会了，但有些朋友可能在上面的例子中就想到假如 Swap 的参数是数组或其它什么来着会怎样—不是说类属类型可以由其它类型替换吗？但毫无疑问，仅凭上面这么简单的实现，就想接受所有的数据类型是不可能的。为了防止实例化时出现不合适的数据类型，我们可以指定类型参数的类别；当然编译器也会自动找出错误，不允许我们进行非法操作。

下面是类型参数的一些分类，各自有不同的限制条件：

type T is private — 限制最少，先前例子所采用的；

type T is limited private — 比前者略多一点限制；

type T is (<>) — T 一定要是离散类型；

type T is range <> — T 一定要是整型；

type T is digits <> — T 一定要为符点型；

type T is delta <> — T 一定要定点型；

type T is array(index_type) of element_type –实际数组的成员类型一定要和形式数组的类型匹配；

type T is access X — T 指向类型 X，X 可以预先定义的类属参数；

具体使用和先前类似，这里就略过了。

11.4.2 值参数(Value Parameter)

先前我们在类属部份都声明类型参数，供后面的程序使用，但是也可以直接声明某一变量：

```
generic
  type element is private;
  size: positive := 200;
package stack is
  procedure push...
  procedure pop...
  function empty return boolean;
end stack;
package body stack is
  size:integer;
  theStack :array(1..size) of element;
  . . .
```

这样，实例化新的程序包：

package fred **is new** stack(element => integer, size => 50);
或
package fred **is new** stack(integer,1000);
或
package fred **is new** stack(integer);

注意，如果值参数没有默认值，实例化时一定要提供一个值。值参数也可以为字符串类型：

```
generic
  type element is private;
```

```
  file_name        :string;
package ....
```

由于 file_name 长度没有限制，相当于子程序有一个参数为 string 类型。

11.4.3 子程序参数(Subprogram parameter);

我们也可以将一个子程序作为一个类属参数，见下：

```
generic
  type element is limited private;
  with function "="(e1,e2:element) return boolean;
  with procedure assign(e1,e2:element);
package stuff is . . .
```

实例化一个程序包：

```
package things is new stuff(person,text."=",text.assign);
```

也可以有一个默认子程序：

```
with procedure assign(e1,e2:element) is myAssign(e1,e2:person);
```

或者

```
with function "="(e1,e2:element ) return boolean is <>;
```

这样实例化时，如果”=”函数没有函数提供，则会根据 element 的类型，自动选择一个”=”函数，如 element 是 Integer 类型，则会使用 Integer 的 “=” 函数。

第 12 章 并发支持(Concurrency Support)

12.1 概述(Overview)

Ada 里内建了对并行运算的支持，也就是任务(task)机制。Ada 程序可以由 1 个或多个任务组成，既可以运行在多 CPU 机上也可运行在单 CPU 机器上，这些任务通过一些通信机制并发执行，好像每个任务是在一台单独机器上运行一样。Ada 下的 Task 也就是我们通常所说的线程(thread)或轻量级进程(light-wight process)，有关这方面的内容读者可参阅《操作系统设计与实现》、《Unix Internal》（中文译名忘了）等关于操作系统的书籍。这里就简单的解释一下：单线程的进程就是我们通常所见的程序，从内存中的一个执行点按顺序执行指令；多线程的进程则将要执行的指令序列分成多个执行片断，每个线程从一个执行点开始执行。

多线程在服务器程序、桌面系统等方面使用相当广泛。比方说 IE 这些浏览器，它们从服务器上下载数据的同时还要显示所下载的信息，这就至少要两个线程；实际程序

中通常也将下载数据分成多个线程，如该网页上有多幅图片，就创建多个线程分别下载这些图片。这就给人一种同时发生的感觉。

在分步式计算中，如果操作系统和编译器都支持的话，不同的线程还可以运行在不同的机器上，比方说像 QNX 这样的分布式实时操作系统。

从本质上讲，Ada 中的任务和上述的线程一样。任务可以创建和终止，在被触发的情况下，多任务间通过以下一些方法进行通信：

- 一个任务可以等待其它任务结束。
- 任务之间可以传递消息，这称之为会合（rendezvous）。
- 任务可以使用保护类型(protected type)，保护类型提供了对数据读写访问的控制。
- 任务也可以通过全局变量来通信。这虽然比较高效，但也比较危险，Ada 允促这项特性主要是为了满足一些实时系统开发人员的需要。

同样，也有一些注意点：

- Ada 不能创建原本没有的东西。在单 CPU 机上，Ada 模拟多台机器，使任务看上去在单独机器上运行一样。但这中间有很大的系统开销，即从一个任务切换到另一个任务，这个过程称为“上下文切换(context swtich)”。不管是多进程还是多线程，这方面的开销都比较大，尤其是内核线程(kernel thread)，还涉及到系统调用，要在内核态和用户态间进行切换；数据库软件如 Oracle 它们采用的用户线程(user thread)则要高效的多，《Unix Internal》关于线程有很详细的讲述。编译器有关任务的具体实现直接影响到任务的效率。
- 任务可能被过多使用。操作系统对于任务数一般都有限制，而且随着任务数的增多，程序效率会降低很多。特别是在 Linux，FreeBSD 这些对线程支持不佳的操作系统上，要谨慎使用任务；在嵌入式系统中这还好点，它们一般在实时调度、线程上做得很不错。
- 如果操作系统支持线程，则一般会对 Ada 的任务机制提供支持；像 Windows NT，Mach，Solaris，OS/2 这些现代化的操作系统。而 Windows 3.1 和老的 Unix 则没提供支持，会造成一些限制。至于 MS-DOS，虽然根本不支持线程，但由于 MS-DOS 相当初级，很容易通过接管整个系统来创建线程。

Ada 下的任务机制在很多方面的确比传统操作系统下的线程强：没有像 Unix 一样每个操作系统几乎都有自己的一套线程库，也比 Posix 规定的线程库使用简单；提供了保护类型，信号灯之类的实现更加高效、简单。

12.2 任务创建和通信（Creating and Communicating with Tasks）

就让我们通过一个小程序 Noise 来了解任务。在这个程序中会有两个任务 Quarreler1 和 Quarreler2，它们并发执行输出一些字符串。多说无益，现让我们看了例子再说：

```
000 — filename :quarrel.ads
001 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

002 package Quarrel is
003  task type Quarreler is
004  entry Start(Message : String; Count : Natural);
005  end Quarreler;
006 end Quarrel;
```

在这里，我们创建了一个任务类型 Quarreller，还有一个 entry 为 Start，将它们封装在程序包 Quarrel 内。Entry 的声明和过程一样，它的参数模式也可为 in,out,in out，并可以有默认参数值，当然也可以没有参数。Entry 表示任务被触发时要执行的指令，如赋值给任务内的局部变量等，一般情况下是具有服务性质的任务传递一些信息给被调用任务。

和子程序有声明部份和主体部份一样，任务类型 Quarreler 也有主体部份：

```
000 –filename quarrel.adb
001 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
002 with Ada.Text_IO;use Ada.Text_IO;

003 package body Quarrel is
004    task body Quarreler is
005       Quarreler_Words : Unbounded_String;
006       Maximum_Count   : Natural;
007    begin
008       accept Start(Message : String; Count : Natural) do
009          Quarreler_Words := To_unbounded_String(Message); — Copy the
rendezvous data to
010          Maximum_Count := Count; — local variables.
011       end Start;
012       for I in 1 .. Maximum_Count loop
013          Put_Line(To_String(Quarreler_Words));
014          delay 0.5;
015       end loop;
016 end Quarreler;
017 end Quarrel;
```

任务的执行部份定义任务要执行的指令。上例中，先声明了两个局部变量 Quarreler_Words 和 Maximum_Count。“accept”语句等待其它任务对该任务发送与 entry 相匹配的请求，本例中是 start；如果接收任务没接受到 entry Start，则会一直等待下去；当其它任务发送与 Start 相匹配的请求时，接收任务(accepting task)会执行在“do”和“end”之间的 accept 语句，这时它就在与其它任务会合（rendezvous）。此时请求任务(requesting task)不会执行任何指令，直至 accept 语句执行完毕，接着两个任务都开始运行。一般来说，在会合处传递给接收任务以后要用到的数据。当然，如果将[012]-[15] 加入 accept 语句之间也可以，在该例中效果差不多。

下面是使用上面任务的例子：

```
000 –filename:noise.adb
001 with Quarrel, Ada.Strings.Unbounded;
002 use Quarrel, Ada.Strings.Unbounded;

003 procedure Noise is
004     Quarreler_1 : Quarreler; — Create a task.
005     Quarreler_2 : Quarreler; — Create another task.
006 begin
007     Quarreler_1.Start(“You must do apologize for causing my jet fighter crash!”, 10);
008     Quarreler_2.Start(“Do apologize?    Are you kidding?”, 10);
009 end Noise;
```

过程 Noise 先创建两个任务 Quarreler_1，Quarreler_2;然后调用它们的 Entry，这时在进行会合。这样两个任务就并发执行了，在屏幕上输出各自的字符串，至于 delay 语句表示该任务要暂停 n 秒。

如果不创建任务类型，只和声明一个变量一样创建一个单一的任务，也可以使用如下格式：

```
task task_name is -- 如果该任务没有 entry，只有一句 task task_name 也是合法的；
       ...
end task_name;
task body task_name is
begin
  ...
end task_name;
```

这里随便提一下两个概念性的问题：

1. 每个 Ada 程序至少有一个任务，称之为环境任务（*environment* task），像上例的 Noise 就在环境任务中运行。

2. 生成其它任务实例的过程称之为 Master，上例中是 Noise。在 Master 的所有任务结束以前，Master 不能结束。

像上面创建任务的方法，在动态产生任务的场合不大适用，如服务器程序。假如我们循环调用 Quarreler_1.Start，希望能产生多个这样的任务：

```
for no in 1..1000 loop
Quarreler_1.Start（”Silly! Stupid! Foolish! Idiots! Weak!....”，1);
end loop;
```

与所期望相反，产生的是异常 Task_Error。这时，调用任务要使用访问类型,如：

```
type Quarreler_Access is access Quarreler;
Quarreler_One: Quarreler_Access;
```

这样创建了指向任务 Quarreler 的访问类型 Quarreler_Access，和该类型变量 Quarreler_One。调用时也很简单：

```
Quarreler_One := new Quarreler;
Quarreler_One.Start(“Silly! Stupid! Foolish! Idiots! Weak!....”,10);
```

如果要产生指定数量的任务，则：

```
for no in 1..1000 loop
Quarreler_One := new Quarreler;
Quarreler_One.Start”Silly! Stupid! Foolish! Idiots! Weak!....”，1)
end loop;
```

这里有个比较有意思的现象，笔者在 FreeBSD 4.2 上做实验时(当时内核的 maxuser 设为 64)，同时间能有 14309 个任务，在 9000 以后，由于内存耗尽，动用了虚存，创建任务的速度大降；在 Windows 2000 Professional 里只能产生 251 个任务。而且余下的任务不能再创建，程序也不会出错。如果实际开发应用程序，对于目标系统还是有必要做一下实际测试。

如果希望任务在执行完指令后不自动结束，也可以在任务的主体部分加入 loop 语句：

```
000 –filename quarrel.adb
001 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
002 with Ada.Text_IO;use Ada.Text_IO;

003 package body Quarrel is
004     task body Quarreler is
005         Quarreler_Words : Unbounded_String;
```

```
006       Maximum_Count   : Natural;
007    begin
008       loop
009          accept Start(Message : String; Count : Natural) do
010             Quarreler_Words := To_unbounded_String(Message); — Copy the
rendezvous data to
011             Maximum_Count := Count; — local variables.
012          end Start;
013          for I in 1 .. Maximum_Count loop
014             Put_Line(To_String(Quarreler_Words));
015             delay 0.5;
016          end loop;
017       end loop;
018 end Quarreler;
019 end Quarrel;
```

这样任务在执行完[009]-016]这段语句后，不会退出，重新等待其它任务调用它的 entry Start。这样做以后，由于该任务不会自然终止，主程序也就不终止，因此还要额外增添退出的代码，比方说通过改变一个全局保护类型变量，来通知任务退出，相关内容见下几节。

还有就是多个 entry 的问题，使用上和一个 entry 一样，更具体的内容见 12.6 Select 语句。

12.3 保护类型 (Protected Type)

上一节我们介绍了以会合方式在任务间进行通信，但这需要一个额外任务传递消息，效率比较低下。为了解决共享数据访问的效率问题，Ada 95 提供了保护类型。保护类型的效率相当高，可以当作高级的信号灯。我们先介绍一下保护类型的基本内容：

保护类型提供了对一个数据对象的同步访问。它包括一些由程序员提供的私有操作，只有这些操作能改动指定数据。和程序包、任务一样，保护类型也有一个说明部份和主体部分。它的操作有以下 3 类：

1. 保护函数(protected function)，它提供了对内部数据的只读访问，一般返回内部数据的值，多个任务可以同时调用一个保护函数。
2. 保护过程(protected procedure)，对内部数据提供排斥性的读写操作，一般是改变内部数据的值。当一个任务调用一个保护过程时，其它任务不能与该访问类型交互作用。
3. 保护入口（protected entry)，和保护过程差不多，但它有一个屏障(barrier)。一个屏障是布尔表达式，在调用程序接下去执行前，它的值必须为 True。屏障通常

依赖于内部数据。当调用程序发送请求时，如果屏障值不是 True，则调用程序在队列(queue)中等待直至屏障值为 True（也就是被阻塞了，和操作系统书上所讲的阻塞(block)、挂起(suspend)差不多）。

保护类型的实现相当高效，一般采用中断封锁(interrupt disable)、自旋锁(spin lock）这些技术。更主要的是，避免了会合所造成的上下文切换，系统开销省了很多。下面就让我们看一下实例：

```
protected type Counting_Semaphore(Start_Count: Integer := 1) is
    entry Secure;
    procedure Release;
    function Count return Integer;
private
   Current_Count: Integer := Start_Count;
end Counting_Semaphore;
protected body Counting_Semaphore is
   entry Secure when Current_Count > 0 is
   begin
      Current_Count := Current_Count - 1;
   end Secure;
   procedure Release is
   begin
      Current_Count := Current_Count + 1;
   end Release;
   function Count return Integer is
   begin
      return Current_Count;
   end Count;
end Counting_Semaphore;
```

上例中的保护类型 Counting_Semaphore 有一个初始值 Start_Count，默认值为 1。因此创建该保护类型的数据只需：

Var : Counting_Semaphore;

或 Var: Counting_Semaphore(2);

但 Start_Count 这样的参数不是必需的。

里面还有个私有类型 Current_Count，它的初始值就是 Start_Count。由于 Current_Count 是保护类型内部操作要用的变量，所以为私有类型。当然，私有部份也可以有其它保护函数、保护过程和保护入口，这样做的话它们只能被保护类型内部的其它操作所使用，而不能被调用程序所用；保护类型内的操作也可以使用全局变量，而不是内部的私有类型，但这样做并不推荐。

函数 Count 较为简单，返回当前的 Current_Count，其它任务可以同时调用它。

过程 Release 增加 Current_Count 的值，同时间只有一个任务能调用它，这时候其它调用它的任务都会被阻塞，直至上一个任务调用它结束。

虽然 Entry Secure 的只是将 Current_Count 减 1，注意 entry Secure 后面的 when Current_Count>0，这就是所谓的 barrier，如果 Current_Count >0 ，则调用它的任务顺利执行后面的语句；如果 Current_Count = 0，则调用它的任务就暂停，直至其它任务调用了 Release，使 Current_Count >0，被阻塞的任务接下去执行。

因此

Var.Secure； — 如果条件符合，则顺利执行，否则，被阻塞。由于 Current_Count 实际上是在 1,0 之间变动，所以顺利调用它的任务将 Current_Count 又设为 0，来阻塞其它调用 Var.Secure 的任务。

doing something else

Var.Release — 由于对共享资源的操作已完成，该任务就释放所占用的资源—将 Current_Count 加 1，这样，刚才被阻塞的任务接下去占用该资源，重复上述操作。这些阻塞的任务按先进先出的顺序排队，就防止了竞态状态（race condition）和轮询(polling)。

除了上述的方法外，我们也可以使用重新排队(requeue)。格式为：

requeue *entry_name* [**with abort**];

entry_name 为其它的 Entry ，可以是同一保护类型内部的 entry，也可以是外部的 entry，requeue 以后，调用程序就重新在 *entry_name* 上排队了。因此假设上述例子中还有一个 entry Continue，并在 secure 中添一句 requeue Continue，则调用 Var.Secure 的程序在执行到这一句时，就重新在 Continue 上排队。这对比较复杂的保护类型还是比较有用的。另外，requeue 也可以出现在任务的 entry 中，这时产生的效果也差不多，不是被阻塞就接下去执行语句。如果有了 with abort，则原来在排队的任务被终止或超时后，这个 requeue 也就取消了；如果没有 with abort，则要退出的任务还会在排队，直至不用排队，这时结束任务。关于 abort 见下一节。

12.4 结束任务(Abort Task)

通常情况下，任务会随着指定语句的完成而结束，或者使用 12.6 Select 语句中所介绍的 Terminate；但有时我们需要强行结束任务，这就要使用 abort 语句。前者退出任务比较“温和”，后者则是强制的。abort 语句格式如下：

abort *task_name*;

task_name 是一个任务的名称，也就是所要退出的任务类型。比方说，我们在 12.1 讲的任务类型 Quarreler，在它的执行部份可以加入：

abort Quarreler;

这样 Quarreler_1，Quarreler_2 执行到这一句，会进入异常状态，马上结束任务。但有几种情况例外，它们在完成前不受 abort 影响，而是到一个特定的完成点(completion point)结束：

- 一个保护操作(protected action)；
- 等待一个 entry 呼叫结束；
- 等待相关联的任务结束；
- 过程 Initialize 作为一个受控对象(controlled object)的最后一步初始化工作执行时；
- 过程 Finalize 作为一个受控对象的部份结束工作执行时；
- 对一个有受控部份(controlled part)的对象赋值时；

如果 Master 退出了，依赖于这个 Master 的任务都会退出。一般来说，abort 会顺利完成，即使任务延迟处理 abort，由于这个延迟时间相当短，如果延迟部份不是凑巧处理全局变量之类，其结果都和所希望的一样；如果遇到比较不明不白的情况时，请参阅 RM95 9.8 Abort of a Task。

12.5 Delay 语句,Duration,Time（Delay Statements,Duration,Time)

12.2 里我们已见过了 delay 语句，从它的名字我们也可推断是暂停时间用的，它的使用格式为：

delay *delay_expression*;

delay until *delay_expression;*

dealy 语句中的 *delay_expression* 是预定义的 Duration 类型的定点小数(fixed point)，Duration 表示一个时间间隔，用秒来表示；*delay_expression* 可以在小数点后有很多位数字，但最终都会转换成 Duration 类型。因此，执行该句的任务暂停 Duraion((*delay_expression)* 秒。

delay until 语句中的 *delay_expression* 可以是任何一个无限制类型(nonlimited type)，一般情况下，是 Time 类型的数。表示暂停直到 *delay_expression* 所表示的时间到达。与 delay 有关的是程序包 Ada.Calendar:

```
package Ada.Calendar is
   type Time is private;
   subtype Year_Number  is Integer range 1901 .. 2099;
   subtype Month_Number is Integer range 1 .. 12;
   subtype Day_Number   is Integer range 1 .. 31;
   subtype Day_Duration is Duration range 0.0 .. 86_400.0;
   function Clock return Time;
   function Year    (Date : Time) return Year_Number;
   function Month   (Date : Time) return Month_Number;
   function Day     (Date : Time) return Day_Number;
   function Seconds (Date : Time) return Day_Duration;
   procedure Split
     (Date    : Time;
      Year    : out Year_Number;
      Month   : out Month_Number;
      Day     : out Day_Number;
      Seconds : out Day_Duration);
   function Time_Of
     (Year    : Year_Number;
      Month   : Month_Number;
      Day     : Day_Number;
      Seconds : Day_Duration := 0.0)
      return    Time;
   function "+" (Left : Time;     Right : Duration) return Time;
   function "+" (Left : Duration; Right : Time)     return Time;
   function "-" (Left : Time;     Right : Duration) return Time;
   function "-" (Left : Time;     Right : Time)     return Duration;
   function "<" (Left, Right : Time) return Boolean; function "<=" (Left,
Right : Time) return Boolean;
   function ">"  (Left, Right : Time) return Boolean;
   function ">=" (Left, Right : Time) return Boolean;
   Time_Error : exception;
private
   type Time is new Duration;
end Ada.Calendar;
```

我们需要注意的是 Clock 函数，它返回的 Time 类型实际上是 Duration 的派生类型，返回从一个基点到现在所经历的秒数，这个基点一般情况下是 1969 年 12 月 31 日 24:00，或者说是 1970 年 1 月 1 日 0:00。但如 Year_Number 所提示的，Ada 表示的年份范围为 1901..2099，所以 Time 类型的数也可以是负值，具体实现可以进一部扩展所能表示的年份范围。

函数 Year，Month，Day，Seconds 返回给定 Time 值的相对应值；过程 Split 返回 4 个相对应值。函数 Time_Of 根据它的 4 个参数，返回相对应的 Time 值。”+”，”-”这些运算符则对 Time 值进行运算。

Time_Of 参数中的 Seconds 如果为 86_400.0，则返回值和下一天的 seconds 值为 0.0 一样。函数 Seconds 返回值和 Split 中的 Seconds 总是小于 86_400.0。

最后要注意的是 Time_Error，有以下 3 种情况会产生 Time_Error（具体实现可以再添加）：

- 如果函数 Time_Of 的参数无法形成合适的 Time 值；
- “+””-”运算产生的结果不能表示；
- 函数 Year 返回值和 Split 中的 Year 超出了 Year_Number 的范围；

讲了这么多，让我们返回来再看 delay 和 delay until 语句。delay 语句我们见过了，下面给出 delay until 语句的例子:

```
...
Next.Time : Time := Clock + Period;–Period 是 Time 类型值；
begin
loop
delay until Next_Time;
...
Next_Time := Next_Time + Period;
end loop;
end;
```

这样，程序暂停直至经过了 Period 所表示的时间。应该说，它们的使用还是很简单的。RM95 里建议，如果 delay 后跟的时间超过现在时间 90 天，也产生 Time_Error，编译器一般情况下会照做。

Gnat3.13p 中还提供了以下一些和 Ada.Calendar 相关的程序包：

- Ada.Calendar.Conv：提供了 Time 和 System.Task_Clock.Stimespec 类型的转换函数（Stimespec 一般精确到纳秒 10E-9)
- Ada.Calendar.Delays: 提供了过程 Delay_For，Delay_Until 和 To_Duration；
- GNAT.Calendar：功能和 Ada.Calendar 差不多（函数名，数据类型略有区别），但精确度更高，可到 0.1 秒；
- GNAT.Calendar.Time_IO ：用于输出日期，既可以自定义输出格式，也可以按照预定义的 ISO,欧洲和美国这 3 种输出格式；

感兴趣的朋友可以自行察看它们的程序包说明，在此就不提了。

时间控制这块内容比较多，在实时系统中我们还会进一步介绍 delay，时间精度这些问题，现在暂时先告一段落。

12.6 Select 语句(Select Statements)

在讲 Select 语句前，我们先将 12.2 的 Quarreler 改动一下：

```
000 — filename :quarrel.ads
001 package Quarrel is
002    task type Quarreler is
003       entry Start_1 (Message : String; Count : Natural);
004       entry Start_2 (Message : String);
005    end Quarreler;
006 end Quarrel;
```

```
000 –filename quarrel.adb
001 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
002 with Ada.Text_IO;use Ada.Text_IO;

003 package body Quarrel is
004   task body Quarreler is
005      Quarreler_Words : Unbounded_String;
006      Maximum_Count   : Natural;
007   begin
008     loop
009       accept Start_1(Message : String; Count : Natural) do
010          Quarreler_Words := To_Unbounded_String(Message);
011          Maximum_Count := Count;
012       end Start_1;
013       for I in 1 .. Maximum_Count loop
014          Put_Line(To_String(Quarreler_Words));
015         delay 0.5;
016       end loop;
017       accept Start_2(Message : String) do
018          Quarreler_Words := To_unbounded_String(Message);
019       end Start_2;
020        Put_Line(To_String(Quarreler_Words));
021       delay 0.5;
022     end loop;
023   end Quarreler;
024 end Quarrel;
```

这样它有两个 entry 了，如果有该任务类型的变量 Var，我们可以 Var.Start_1 或 Var.Start_2 来调用该任务；但假如单个 Var.Start_1 连续两次以上出现，则程序会进入很尴尬的状态：一方面请求任务要执行第二个 Var.Start_1，则 accept Start_2 必须执行后，循环才能重新返回执行 accept Start_1，于是接受任务就会卡在 accept Start_2 上，直至其它任务调用 Var.Start_2 请求任务才能继续，而请求任务又执行了 Var.Start_1 同时又没有其它任务，整个程序就停止了。单个 Var.Start_2 出现 2 次以上也如此。

为了防止上述现象，我们可以采用 select 语句：

```
loop
select
accept Start_1(Message : String; Count : Natural) do
Quarreler_Words := To_Unbounded_String(Message);
Maximum_Count := Count;
end Start;
for I in 1 .. Maximum_Count loop
Put_Line(To_String(Quarreler_Words));
delay 0.5;
end loop;
or
accept Start_2(Message : String) do
Quarreler_Words := To_unbounded_String(Message);
end Start_2;
Put_Line(To_String(Quarreler_Words));
delay 0.5;
end select;
end loop;
```

这样我们连续调用 Var.Start_1 就不会出现上述的问题了。select 语句顾名思义，在多个 entry 中做出选择，因此 Start_1 和 Start_2 的地位是并列的，不存在谁先执行谁后执行的问题。但上例中有 loop 语句，因此该任务正常情况下不会结束，它的 master 也就伴随着不终止。这里我们可以采用 terminate 来终止它，在 select 语句中添一句：

```
or terminate;
```

这时，不管 Var 做不做事，调用它的 master 也可以退出。terminate 语句只有满足下列条件才会被选择，即退出任务：

- 该任务依赖于已完成的 master；
- 该任务依赖的 master 已被终止或阻塞在有 terminate 语句的 select 语句上；

因此创建 Var 的 master 已执行完自己的语句，只等待 Var 结束这种情况下，terminate 会被选择，Var 和它的 master 就都正常结束了。

select 语句也可用于定时控制：

```
select
    do_something;
or
    delay  3.0;
    do_otherthing;
```

```
end select;
```

当有这种语句的任务被创建后，如果 3.0 秒内没有被调用执行 *do_something*，则自动执行 *do_otherthing* 。

上面的例子都是有条件选择（conditional alternative)，下例是无条件选择(unconditional alternative):

select
do_something_1;
or
do_something_2;
else
do_otherthing;
end select;

这样，任务被创建时，只是很简单地检查是否有其它任务和它会合，没有，则执行 *do_otherthing*。

最后一种情况是异步控制转移(asynchronous transfer of control)：

```
select
    trigging_statement;
then abort
    abortable_statement;
end select;
```

trigging_statement 里有 delay 语句。假如 *abortable_statement* 不能在 delay 语句指定的时间内完成，则执行 *trigging_statement*；如果提前完成，则取消 *trigging_statement*。如：

```
select
    delay 5.0;
    Put_Line("Calculation does not converge.");
then  abort
     -- This calculation should finish in 5.0 seconds;
      -- if not,it is assumed to diverge.
    Horribly_Complicated_Recursive_Function(x,y);
end select;
```

第 14 章 字符与字符串处理

14.1 概述(Overview)

很多程序都要处理文本文件或用户输入等，这就涉及到下章要介绍的输入输出和本章的字符与字符串处理。这两大块内容几乎是每门计算机语言或每个操作系统的“必带

品”，虽然实现原理简单，却对整个系统和程序有很重要的影响，同时又往往牵连到效率和安全性的问题。比方说，由于 C 中指针和字符串的特殊关系，导致缓冲区溢出成了多年来编程的冤大头；输入输出的效率对整个系统的效率起了决定性的作用，而伴随而来的文件缓存机制也多种多样。

本章重点介绍 Ada 标准库所列出的相关子程序，由于字符串相对来说比较简单，读者也可直接翻看 RM 来代替本章节。

14.2 字符和字符串处理的基本知识(Basis of Character and String)

字符

我们在第二章已介绍过了 Ada 里的字符元素，Ada95 已支持 ISO 8859 Latin_1 字符集，字符类型 Character 中有 256 个位置，相对应的 Ada83 只支持 ISO 646，仅有 128 个位置，即通常所说的 ASCII 。同时 Ada95 还支持宽字符集 Wide_Character。在这里，我们简要讲一下字符处理程序包的组成结构。

Characters 只是作为一个象征意义的程序包存在，它的声明只有：

```
package Ada.Characters is
pragma Pure (Characters);
end Ada.Characters;
```

Charcaters.Latin_1 是作为 Charcaters 的子单元存在，定义了 Latin_1 中 256 个字符的名称，也就是字符类型 Character 里每个元素的名称，以方便用户能控制一些不可见字符。Ada 也允许具体实现支持本地字符集，比方说一些东欧国家可以使类型 Character 基于 ISO 8859, Latin-2，但这样的话和标准有很大不同，并且导致 Characters.Handling 也要随之变化。还有一点就是 Ada83 里的 Standard.ASCII，它和 Ada95 里的 Charcaters.Latin_1 作用相同，但只有 128 个字符名称，现在已是“古董”了，如果碰见使用这个软件包的程序，知道它是什么东西就可以了。
Character.Latin_1 的实现一般是这样的：

```
package Ada.Characters.Latin_1 is
pragma Pure (Latin_1);
NUL             : constant Character := Character'Val (0);
SOH             : constant Character := Character'Val (1);
STX             : constant Character := Character'Val (2);
ETX             : constant Character := Character'Val (3);
EOT             : constant Character := Character'Val (4);
ENQ             : constant Character := Character'Val (5);
...
LC_U_Grave      : constant Character := Character'Val (249);
```

```
LC_U_Acute         : constant Character := Character'Val (250);
LC_U_Circumflex    : constant Character := Character'Val (251);
LC_U_Diaeresis     : constant Character := Character'Val (252);
LC_Y_Acute         : constant Character := Character'Val (253);
LC_Icelandic_Thorn : constant Character := Character'Val (254);
LC_Y_Diaeresis     : constant Character := Character'Val (255);
end Ada.Characters.Latin_1;
```

具体见附录 A，这样输出一些控制字符很方便，如在终端屏幕上换行，可以 Put(PM)（PM 表示换行符），来取代 New_Line。有关这方面的内容见下章 标准输入与输出。

由于 Character.Latin_1 的使用频率较高，但用户程序不一定使用 Character.Handling 里的子程序，为了避免无谓的代码空间造成的性能损失（如果一个程序包被绑定到一个应用程序，该程序包内的子程序会被载入内存，即使不调用任何子程序，有 Windows 下有编程经验的朋友应该清楚 dll 吃内存的"贪"），因此 Charactes.Latin_1，Character.Handling 被作为 Characters 的子单元。

字符串

一组有序的字符就组成了字符串，Ada83 里只提构了一个内建的字符串类型 String；为了满足需要，Ada95 提构了其它字符串类型，它们都是为了某特定目的而产生，而且和 String 之间可以互相转换：

String：最常见的字符串类型，它只是一个 Chraracter 类型的数组，并且 String 类型的变量创建要指定长度，这个长度是固定的，因此 String 也被称为"定长字符串 (fixed length string)"。当然也可以采用和 C 差不多的技术，使访问类型的变量指向一个字符串，以此来动态改变字符串长度。如果真要动态改变数组长度，下面两种字符串类型应当更值得推荐。

Bounded_String: 子符串的长度可以动态改变，但最大长度不能超过创建字符串时指定的最大值，因此该类型比较适合可以预知字符串最大长度的情况。

Unbounded_String：从字面来看，是没有限制长度的字符串，它的最大长度到 Natural'Last，一般 32 位机中，Natural'Last 是 20 亿多，所以很适合根本不知道字符串会多长的情况。

其它字符串类型： C.Strings.chars_ptr, COBOL.Alphanumeric, and Fortran_Character。主要是在创建其它语言的接口时使用，不管怎样，Ada 的字符串和其它语言的字符串是不一样的，这些字符串类型在封装其它语言的函数库时就相当有用。

字符串处理的主程序包是 Strings，和 Characters 一样的无聊：

```
package Ada.Strings is
Space : constant Character := ' ';
Wide_Space : constant Wide_Character := ' ';
Length_Error, Pattern_Error, Index_Error, Translation_Error : exception;
type Alignment is (Left, Right, Center);
type Truncation is (Left, Right, Error);
type Membership is (Inside, Outside);
type Direction is (Forward, Backward);
type Trim_End is (Left, Right, Both);
end Ada.Strings;
```

这些数据类型和异常会在 Strings 子单元中使用。在后面的章节中我们会介绍 Ada.Strings.Bounded，Ada.Strings.Fixed，Ada.Strings.Unbounded，Ada.Strings.Maps，除了 Ada.Strings.Maps 还是对字符集进行操作外，其它 3 个程序包都针对某一个字符串类型提供一些处理子程序。Characters 有 Characters.Latin_1，Strings 也有 Ada.Strings.Maps.Constants，该程序包定义了一些 Character_Set 和 Character_Mapping 常量，它们和 Characters.Handling 里的分类和转换函数相对应，具体内容下节字符处理：

```
package Ada.Strings.Maps.Constants is
Control_Set          : constant Character_Set;
Graphic_Set          : constant Character_Set;
Letter_Set           : constant Character_Set;
Lower_Set            : constant Character_Set;
Upper_Set            : constant Character_Set;
Basic_Set            : constant Character_Set;
Decimal_Digit_Set    : constant Character_Set;
Hexadecimal_Digit_Set  : constant Character_Set;
Alphanumeric_Set     : constant Character_Set;
Special_Set          : constant Character_Set;
ISO_646_Set          : constant Character_Set;
Lower_Case_Map       : constant Character_Mapping;
Upper_Case_Map       : constant Character_Mapping;
Basic_Map            : constant Character_Mapping;
private
...
end Ada.Strings.Maps.Constants
```

私有部份是上述常量的具体定义。数据类型 Character_Set 和 Character_Mapping 的定义在 Strings.Maps 的私有部份：

```
type Character_Set is array (Character) of Boolean;

type Character_Mapping is array (Character) of Character;
```

可想而知 Control_Set 它们的定义了，如:

Control_Set : **constant** Character_Set := （

Ada.Characters.Latin_1.NUL .. Ada.Characters.Latin_1.US =>True,

Ada.Characters.Latin_1.DEL .. Ada.Characters.Latin_1.APC => True，

others => False);

表示 Control_Set 的 255 个元素中，下标是控制字符的元素值都为 True，其它元素值都为 False(在 GNAT 中，为了方便，将 Ada.Characters.Latin_1 重命名为 L，所以上面的 Ada.Characters.Latin_1 应当用 L 代替），布尔值就表示该字符是否是控制字符。其它字符集也类似。

Character_Mapping 的常量比较复杂，如 Lower_Case_Mapping：

```
Lower_Case_Map : constant Character_Mapping := (
L.NUL  &       — NUL      0
L.SOH  &       — SOH      1
L.STX  &       — STX      2
...
L.LC_A &       — ‘a’      65
L.LC_B &       — ‘b’      66
L.LC_C &       — ‘c’      67
...
L.LC_A &       — ‘a’      97
L.LC_B &       — ‘b’      98
L.LC_C &       — ‘c’      99
... );
```

“–”的后面是该元素的值和在 Latin_1 中的位置。大家大概注意到了，有两个 L.LC_A，本来该大写的字母变成了小写，所以常量 Lower_Case_Map 实际上就是 Latin_1 中的 255 个字符按顺序排成的字符串，其中大写字母都变成了小写。其它 Character_Mapping 类型的变量以此类推。

Character_Set 和 Character_Mapping 这两个数据类型还是比较重要，子程序不少是用着它的。但注意一下，Character_Set 的实现虽然如上，但从语义上讲则是一个集合，因此如果说某一个字符在某字符集中，实际上就是 Character_Set 的变量中以该字符为下标的值为 True；Character_Mapping 的变量中如果字符下标为 S ，该元素值为 T，则从字符 S 映射到字符 T。

14.3 字符处理（Characters.Handling 和 Strings.Maps)

Characters.Handling

我们在第二章的时候将字符分为图形字符、控制字符、其它控制字符。在 Characters.Handling 里，字符分类更为复杂。下列是一些分类函数的说明：

function Is_Control (Item : in Character) **return** Boolean; 如果 Item 是控制字符，返回 True，控制字符的位置在 0..31 和 127..159;
function Is_Graphic (Item : in Character) **return** Boolean; 如果 Item 是图形字符，返回 True，图形字符的位置在 32..126 和 160..255;
function Is_Letter (Item : in Character) **return** Boolean; 如果 Item 是字母，返回 True，包括 'A'..'Z','a'..'z',字母的位置在 192..214，216..246，248..255;
function Is_Lower (Item : in Character) **return** Boolean; 如果 Item 是小写字母，返回 True,包括'a'..'z'，小写字母的位置在 223..246，248..255;
function Is_Upper (Item : in Character) **return** Boolean; 如果 Item 是大写字母，返回 True,包括'A'..'Z'，大写字母的位置在 192..214，216..222;
function Is_Basic (Item : in Character) **return** Boolean; 如果 Item 是基本字母，返回 True,包括上面所讲的字母和'Æ' 'æ' 'Ð' 'ð' 'þ' 'Þ' 'ß';
function Is_Digit (Item : in Character) **return** Boolean; 如果 Item 是十进制（'0'..'9'）的数字，返回 True;
function Is_Decimal_Digit (Item : in Character) **return** Boolean **renames** Is_Digit; Is_Digit 的重命名;
function Is_Hexadecimal_Digit (Item : in Character) **return** Boolean; 如果 Item 是 16 进制数字，返回 True，16 进制数字包括 10 进制数字和'A..F'， 'a'..'f';
function Is_Alphanumeric (Item : in Character) **return** Boolean; 如果 Item 是文字数字，返回 True，文字数字是一个字母或是一个十进制数字。
function Is_Special (Item : in Character) **return** Boolean; 如果 Item 是特殊图形字符，返回 True，特殊图形字符是非文字数字的图形字符。

这些函数使用上很简单，这里就不举例了。它们的实现相对来说也很简单，我们只需了解一下即可：

type Character_Flags **is mod** 256;

Control : **constant** Character_Flags := 1;
Lower : **constant** Character_Flags := 2;
Upper : **constant** Character_Flags := 4;
Basic : **constant** Character_Flags := 8;

```
Hex_Digit :  constant Character_Flags := 16;
Digit    :  constant Character_Flags := 32;
Special  :  constant Character_Flags := 64;
Letter   :  constant Character_Flags := Lower or Upper;
Alphanum :  constant Character_Flags := Letter or Digit;
Graphic  :  constant Character_Flags := Alphanum or Special;
```

首先是将字符分成 10 类，用 Modular 类型 Character_Flags 来标记它们。然后创建一个数组 Char_Map：

```
Char_Map : constant array (Character) of Character_Flags :=
(
NUL => Control,
SOH => Control,
STX => Control,
ETX => Control,
...
LC_Y_Acute => Lower,
LC_Icelandic_Thorn => Lower + Basic,
LC_Y_Diaeresis => Lower
);
```

这样 255 个字符每个都被标明了类型，上述的函数也就很容易实现了，如 Is_Basic:

```
function Is_Basic (Item : in Character) return Boolean is
begin
return (Char_Map (Item) and Basic) /= 0;
end Is_Basic;
```

Item 如果是 Basic 型，则 Char_Map(Item) = Basic，而 Basic and Basic = Basic，所以 (Char_Map (Item) and Basic) /= 0 值为 True。其它函数实现也都类似。

除了上述的 Latin_1 中的字符分类外，还有 4 个分类函数：

subtype ISO_646 **is** Character **range** Character'Val (0) .. Character'Val (127);
function Is_ISO_646 (Item : **in** Character) **return** Boolean; 如果 Item 属于子类型 ISO_646,返回 True;
function Is_ISO_646 (Item : **in** String) **return** Boolean; 如果 Is_ISO_646(Item(I))（for I in Item'range) 都是 True,返回 True;
function Is_Character (Item : **in** Wide_Character) **return** Boolean; 如果 Wide_Character'Pos(Item) >= Character'Pos(Character'Last), 返回 True;
function Is_String (Item : **in** Wide_String) **return** Boolean;

如果 Is_Character(Item(I))(For I in Item'range) 都是 True,返回 True;

它们的实现更为简单，只是很简单的判定 Item 是否在所指定类型的取值范围内。如 Is_ISO_646(Item : Character)，判定 Item 的位置是否在 0 和 127 之间，即 ISO_646 内；判定字符串也只是一个循环，将每个字符比较过去。

在 Characters.Handling 里还有以下一些转换函数：

function To_Lower (Item : **in** Character) **return** Character; 如果 Is_Upper(Item)为 True，返回 Item 相对应的小写字母，否则返回 Item 自身。
function To_Upper (Item : **in** Character) **return** Character; 如果 Is_Lower(Item)为 True，返回 Item 相对应的大写字母，否则返回 Item 自身。
function To_Basic (Item : **in** Character) **return** Character; 如果 Item 是一个字母但不是基本字母，返回和 Item 相对应的字符不包括附加符，否则返回 Item 自身；
function To_Lower (Item : **in** String) **return** String; To_Lower(Item : in Character) 的字符串版本;
function To_Upper (Item : **in** String) **return** String; To_Upper(Item : in Character)的字符串版本;
function To_Basic (Item : **in** String) **return** String; To_Basic(Item : in Character)的字符串版本;

以及：

function To_ISO_646 (Item : **in** Character; Substitute : **in** ISO_646 := ' ') **return** ISO_646; 如果 Is_ISO_646(Item) 为 True，返回 Item，否则返回 Substitute（默认为 ' ');
function To_ISO_646 (Item : **in** String; Substitute : **in** ISO_646 := ' ') **return** String; To_ISO_646 (Item : **in** Character; Substitute : **in** ISO_646 := ' ')的字符串版本；
function To_Character (Item : in Wide_Character; Substitute : in Character := ' ') **return** Character; 如果 Is_Character(Item)是 True，返回 Item 相对应的 Character；否则返回 Substitute;
function To_String (Item : in Wide_String; Substitute : in Character := ' ') **return** String; To_Character 的字符串版本；
function To_Wide_Character (Item : in Character) **return** Wide_Character; 返回 Wide_Character X，使 Character'Pos(Item)=Wide_Character'Pos(X);
function To_Wide_String (Item : in String) **return** Wide_String; To_Wide_Character 的字符串版本；

后 5 个函数没什么，前 6 个函数则涉及到下半节的内容，如：

```
function To_Lower (Item : in Character) return Character is
begin
return Value (Lower_Case_Map, Item);
end To_Lower;
```

这里有个 Value 函数，它在 Strings.Maps 里面。

Strings.Maps

Strings.Maps 里的数据结构除了上一节所讲的 Character_Set 和 Character_Mapping，还有以下一些：

```
Null_Set : constant Character_Set;
type Character_Range is
record
Low : Character;
High : Character;
end record;
type Character_Ranges is array (Positive range <>) of Character_Range;
subtype Character_Sequence is String;
Identity : constant Character_Mapping
```

Null_Set 就是没有字符的空字符集；Character_Range 类型的变量 Obj 表示某个字符集合 Obj'Low..Obj'High；Character_Ranges 就表示多个 Character_Range 了。Character_Sequence 只不过是名字换了一下的 String。Identity 的每个元素值和它的字符下标相同。

下面是 Strings.Maps 的一些子程序：

function To_Set (Ranges : **in** Character_Ranges) **return** Character_Set; 如果 Ranges'Length = 0，则返回 Null_Set；否则返回和 Ranges 相对应的字符集；
function To_Set (Span : **in** Character_Range) **return** Character_Set; 返回的字符集包括了 Span 里的每一个字符；
function To_Ranges (Set : **in** Character_Set) **return** Character_Ranges; 如果 Set = Null_Set,返回空 Character_ranges；否则 Set 里的字符值用 Character_Ranges 来表示；
function "=" (Left, Right : **in** Character_Set) **return** Boolean; 如果 Left 和 Right 表示的是相同的字符集，返回 True，否则为 False;
function "not" (Right : **in** Character_Set) **return** Character_Set; **function** "and" (Left, Right : **in** Character_Set) **return** Character_Set; **function** "or" (Left, Right : **in** Character_Set) **return** Character_Set;

function “xor” (Left, Right : **in** Character_Set) **return** Character_Set; **function** “-” (Left, Right : **in** Character_Set) **return** Character_Set; 返回值是进行逻辑运算后的值。“-”等价于 “and”(Left，“not”(Right))，提供“-”主要是为了效率；
function Is_In (Element : **in** Character; Set : **in** Character_Set) **return** Boolean; 如果 Element 在 Set 中，则返回 True，否则为 False；
function Is_Subset (Elements : **in** Character_Set; Set : **in** Character_Set) **return** Boolean; 如果 Elements 是 Set 的子集，则返回 True，否则为 False;
function To_Set (Sequence : **in** Character_Sequence) **return** Character_Set; **function** To_Set (Singleton : **in** Character) **return** Character_Set; 前者返回的字符集包括了 Sequence 中的所有字符，后者返回的字符集只包含了 Singleton；
function To_Sequence (Set : **in** Character_Set) **return** Character_Sequence; 返回的字符序列包含了 Set 中的所有字符，以升序排列，没有重复字符；
function Value (Map : **in** Character_Mapping; Element : **in** Character) **return** Character; 返回 Element 在 Map 中映射的字符；
function To_Mapping (From, To : **in** Character_Sequence) **return** Character_Mapping; From 的每个元素都映射到 To 中相对应的元素，如果 From’Length /= To’Length，或 From 中有字符重复，产生异常 Translation_Error；
function To_Domain (Map : **in** Character_Mapping) **return** Character_Sequence; 返回字符串 D，D 由 Map 中不映射到自身的字符组成；
function To_Range (Map : in Character_Mapping) **return** Character_Sequence; 返回字符串 D，D 由 Map 中不映射到自身的字符所映射的字符组成；
type Character_Mapping_Function **is access function** (From : in Character) **return** Character;

看了上面这么多函数，大家可能对它们是干什么用的还是不清不楚；起码笔者是如此，越翻 Reference Manual 越糊涂，还是看了它们的源代码才明白。

前三个函数从原理上讲是差不多的，就让我们看一下 **function** To_Set (Ranges : **in**Character_Ranges) **return** Character_Set：

```
function To_Set (Ranges : in Character_Ranges) return Character_Set is
Result : Character_Set;
begin
for C in Result’Range loop
Result (C) := False;
end loop;

for R in Ranges’Range loop
for C in Ranges (R).Low .. Ranges (R).High loop
Result (C) := True;
```

```
end loop;
end loop;
return Result;
end To_Set;
```

第一个循环将 Result 变成空字符集，和 Null_Set 一样；第二个嵌套循环，如果在 Ranges (R).Low..Ranges(R).High 中有某个字符，则 Result 中以该字符为下标的元素值为 True，也就是字符集 Result 包括的字符是 Ranges 所包含的字符。To_Set (Span : **in** Character_Range) 和 To_Ranges (Set : **in** Character_Set) 的实现采用的方法也差不多。假如 Range1 : Character_Range = ('A','C')，则 To_Set(Range1) 的返回值为 (L.'A'..L.'C' => True,others=> False);

Is_In (Element : **in** Character; Set : **in** Character_Set) 和 Is_Subset (Elements :**in** Character_Set; Set : **in** Character_Set)应当没什么问题，函数名就说明了它的用途。

To_Set (Sequence : **in** Character_Sequence) **return** Character_Set ， To_Set (Singleton : **in** Character) **return** Character_Set 和 To_Sequence (Set : **in**Character_Set) **return** Character_Sequence 的实现很简单，看一下就可以了，如:

```
function To_Set (Sequence : Character_Sequence) return Character_Set is
Result : Character_Set := Null_Set;
begin
for J in Sequence'Range loop
Result (Sequence (J)) := True;
end loop;
return Result;
end To_Set;
```

Value (Map : **in** Character_Mapping; Element : **in** Character) ，其执行部份只有一句 return Map(Element)，因此 Value(Lower_Case_Map,'A') ='a'。

To_Mapping (From, To : **in** Character_Sequence) ，To_Domain (Map : **in**Character_Mapping) 和 To_Range (Map : in Character_Mapping) 可能使人比较糊涂，看一下例子：

假设 To_Mapping("ABCD","1234″) 的返回值为 R，则 R('A') = '1′,R('B')='2′,R('C') = '3′,R('D')='4′，再 To_Domain(R) ="ABCD"，To_Range(R) = "1234″。

它们的源代码为：

```
———————-
– To_Mapping –
———————-

function To_Mapping (From, To : in Character_Sequence) return
Character_Mapping
is
Result : Character_Mapping;
Inserted : Character_Set := Null_Set;
From_Len : constant Natural := From'Length;
To_Len : constant Natural := To'Length;
begin
if From_Len /= To_Len then
raise Strings.Translation_Error;
end if;

for Char in Character loop
Result (Char) := Char;
end loop;

for J in From'Range loop
if Inserted (From (J)) then
raise Strings.Translation_Error;
end if;
Result (From (J)) := To (J – From'First + To'First);
Inserted (From (J)) := True;
end loop;

return Result;
end To_Mapping;

———————
– To_Domain –
———————

function To_Domain (Map : in Character_Mapping) return Character_Sequence
is
Result : String (1 .. Map'Length);
J : Natural;
begin
J := 0;
for C in Map'Range loop
if Map (C) /= C then
J := J + 1;
Result (J) := C;
```

```
end if;
end loop;
return Result (1 .. J);
end To_Domain;

————–
– To_Range –
————–

function To_Range (Map : in Character_Mapping) return Character_Sequence
is
Result : String (1 .. Map'Length);
J : Natural;
begin
J := 0;
for C in Map'Range loop
if Map (C) /= C then
J := J + 1;
Result (J) := Map (C);
end if;
end loop;
return Result (1 .. J);
end To_Range;
```

字符处理应该说还是比较简单的，实在不明白看一下源代码问题就解决了。

最后提一下 Character_Mapping_Function，它表示将字符 C 映射到字符 F.all(C)，该函数由用户定义。

14.4 定长字符串处理(Strings.Fixed)

字符串复制

```
procedure Move  (Source : in String;
Target : out String;
Drop : in Truncation := Error;
Justify : in Alignment := Left;
Pad : in Character := Space);
```

过程 Move 将字符串 Source 复制到字符串 Target。如果 Source 和 Target 长度相等，产生的效果等于将 Source 赋值给 Target。

如果 Source 比 Target 长度短：

- 如果 Justify=Left，则将 Source 复制到 Target 开头的 Source'Length 个字符。
- 如果 Justify=Right，则将 Source 复制到 Target 末尾的 Source'Length 个字符。

- 如果 Justify=Centre，则将 Source 复制到 Target 中间的 Source'Length 个字符；如果 Source 和 Target 的长度差为奇数，则额外的 Pad 在右面。
- Pad 复制到 Target 中所有没有被赋值的字符。

如果 Source 比 Target 长度长，产生的效果取决于 Drop：

- 如果 Drop=Left，则将 Source 最右边的 Target'Length 个字符复制到 Target。
- 如果 Drop=Right，则将 Source 最右边的 Target'Length 个字符复制到 Target。
- 如果 Drop=Error，则产生的效果取决于 Justify 的值，
  - 如果 Justify=Left，并且 Source 中最右边的 Source'Length-Target'Length 个字符都是 Pad，则 Source 最左边的 Target'Length 个字符复制到 Target。
  - 如果 Justify=Right，并且 Source 中最左边的 Source'Length-Target'Length 个字符都是 Pad，则 Source 最右边的 Target'Length 个字符复制到 Target。
  - 否则，产生异常 Length_Error;

Move 的用法和实现都比较繁琐，但实现原理很简单，这里就不讲了。如果嫌它使用麻烦，后面有些相关子程序是用它来实现的，能满足一些常见的需要。

字符串查找

```
function Index  (Source : in String;
Pattern : in String;
Going : in Direction := Forward;
Mapping : in Maps.Character_Mapping := Maps.Identity)
return Natural;

function Index (Source : in String;
Pattern : in String;
Going : in Direction := Forward;
Mapping : in Maps.Character_Mapping_Function)
return Natural;
```

每个 Index 函数都在 Source 中搜寻通过 Mapping 映射的字符串 Pattern ； 参数 Going 指明查找方向。如果 Going=Forward，Index 返回 Source 中与 Pattern 匹配的片断的起始位置的最小索引值；如果 Going=Backward，Index 返回 Source 中与 Pattern 匹配的片断的起始位置的最大索引值；如果没有匹配的字符串片断，返回 0；如果 Pattern 是空字符串，产生 Pattern_Error。

```
function Index (Source : in String;
          Set    : in Maps.Character_Set;
          Test   : in Membership := Inside;
          Going  : in Direction  := Forward)
          return Natural;
```

Index 查找字符集 Set（当 Test=Inside)或它的补集(当 Test=Outside)中的任一字符最先出现或最后出现的位置。它返回最小的索引值（如果 Going=Forward) 或最大的索引值（如果 Going=Backward)；当 Source 中没有这样的字符时，返回 0。

```
function Index_Non_Blank (Source : in String;
Going : in Direction := Forward)
return Natural;
```

返回 Index(Source,Maps.To_SetI(Space),Outside,Going)。

```
function Count (Source : in String;
Pattern : in String;
Mapping : in Maps.Character_Mapping := Maps.Identity)
return Natural;
function Count (Source : in String;
Pattern : in String;
Mapping : in Maps.Character_Mapping_Function)
return Natural;
```

返回在 Source 中与字符串 Pattern 相匹配的非重叠片断的最大数目。如 Source 为 “ABABABA”，Pattern 为”ABA”，则返回 2。

```
function Count (Source : in String;
Set : in Maps.Character_Set)
return Natural;
```

返回 Set 中的字符在 Source 中出现的次数。

```
procedure Find_Token (Source : in String;
Set : in Maps.Character_Set;
Test : in Membership;
First : out Positive;
Last : out Natural);
```

返回 Source 中的每个元素都满足 Test 条件的第一个片断的索引值 First 和 Last。如果没有这样的片断，则 Last 的值为 0，First 的值为 Source’First。

字符串转换

```
function Translate (Source : in String;
Mapping : in Maps.Character_Mapping)
return String;
function Translate (Source : in String;
Mapping : in Maps.Character_Mapping_Function)
return String;
```

返回字符串 S，它的长度为 Source’Length，因此 S(I) 是经 Mapping 映射的 Source 中的相对应元素，for I in 1..Source’Length。

```
procedure Translate (Source : in out String;
Mapping : in Maps.Character_Mapping);
procedure Translate (Source : in out String;
Mapping : in Maps.Character_Mapping_Function);
```

等价于 Source := Translate(Source,Mapping)。

字符串变换

```
function Replace_Slice (Source : in String;
```

Low : **in** Positive;
High : **in** Natural;
By : **in** String)
return String;

如果 Low > Source'Length，或 High < Source'First-1，产生 Index_Error；如果 High >= Low，返回字符串 Source(Source'First..Low-1)&By&Source(High+1..Source'Last)；如果 High < Low，返回字符串 Insert(Source,Before=>Low,New_Item=>By)。

procedure Replace_Slice (Source : **in out** String;
Low : **in** Positive;
High : **in** Natural;
By : **in** String;
Drop : **in** Truncation := Error;
Justify : **in** Alignment := Left;
Pad : **in** Character := Space);

等价于 Move(Replace_Slice(Source,Low,High,By),Source,Drop,Justify,Pad);

function Insert (Source : **in** String;
Before : **in** Positive;
New_Item : **in** String)
return String;

如果 Before 不在 Source'First..Source'Last+1 中，产生 Index_Error；否则返回 Source(Source'First..Before-1)&New_Item&Source(Before..Source'Last)，下标下限为 1。

procedure Insert (Source : **in out** String;
Before : **in** Positive;
New_Item : **in** String;
Drop : **in** Truncation := Error);

等价于 Move(Insert(Source,Before,New_Item),Source,Drop)。

function Overwrite (Source : **in** String;
Position : **in** Positive;
New_Item : **in** String)
return String;

如果 Position 不在 Source'First..Source'Last 中，产生 Index_Error；否则返回字符串 S，S 从 Source 取到，Position 开始的字符由 New_Item 替换；如果在 New_Item 用完之前，已到达 Source 的底部，则 New_Item 的剩余字符追加在 S 的尾部。

function Delete (Source : **in** String;
From : **in** Positive;
Through : **in** Natural)
return String;

如果 From <= Through，返回字符串 Replace_Slice(Source,From,Through,""），否则返回 Source。

procedure Delete (Source : **in out** String;
From : **in** Positive;
Through : **in** Natural;
Justify : **in** Alignment := Left;
Pad : **in** Character := Space);

等价于 Move(delete(Source,From,Through),Source,Justify=>Justify,Pad=>Pad)。

字符串选择

function Trim (Source : **in** String; Side : **in** Trim_End) **return** String; 返回字符串 S，S 为删除 Source 开头所有的空格符（Side = Left)，或末尾所有的空格符(Side=Right)或两边所有的空格符(Side = Both) 所得的字符串。
procedure Trim (Source : **in out** String; Side : **in** Trim_End; Justify : **in** Alignment := Left; Pad : **in** Character := Space); 等价于 Move(Trim(Source,Side),Source,Justify=>Justify,Pad=>Pad).
function Trim (Source : **in** String; Left : **in** Maps.Character_Set; Right : **in** Maps.Character_Set) **return** String; 返回字符串 S， S 为删除在 Left 中的所有起始字符和在 Right 中的所有末尾字符所得的字符串。
procedure Trim (Source : **in out** String; Left : **in** Maps.Character_Set; Right : **in** Maps.Character_Set; Justify : **in** Alignment := Strings.Left; Pad : **in** Character := Space); 等价于 Move(Trim(Source,Left,Right),Source,Justify=>Justify,Pad=>Pad)。
function Head (Source : **in** String; Count : **in** Natural; Pad : **in** Character := Space) **return** String; 返回长度为 Count 的字符串。如果 Count <= Source'Length，该字符由 Source 的最先 Count 个字符组成；否则由 Source 和 Count-Source'Length 个 Pad 字符相连接。
procedure Head (Source : **in out** String; Count : **in** Natural; Justify : **in** Alignment := Left; Pad : **in** Character := Space); 等价于 Move(Head(Source,Count,Pad),Source,Drop=>Error,Justify=>Justify,Pad=>Pad)。
function Tail (Source : **in** String; Count : **in** Natural; Pad : **in** Character := Space) **return** String; 返回长度为 Count 的字符串。如果 Count <= Source'Length，该字符由 Source 的最末 Count 个字符组成；否则由 Source 和 Count-Source'Length 个 Pad 字符相连接。
procedure Tail (Source : **in out** String; Count : **in** Natural; Justify : **in** Alignment := Left; Pad : **in Character** := Space);

等价于 Move(Tail(Source,Count,Pad),Source,Drop=>Error,Justify=>Justify,Pad=>Pad)。

14.5 有界字符串处理(Ada.Strings.Bounded)

Bounded_String 看上去比较可爱：一个字符串有一个最大长度限制，这样一些有限制的字符串，如文件路径名，都可以声明为 Bounded_String，比普通的 String 方便很多。但事实上并非如此，很多应用程序并不使用 Bounded_String，让我们先看一下 Ada.Strings.Bounded 的声明部份：

package Ada.Strings.Bounded **is**

generic Max : Positive; — Maximum length of a Bounded_String

package Generic_Bounded_Length **is**

Max_Length : **constant** Positive := Max;

……

end Generic_Bounded_Length;

end Ada.Strings.Bounded;

在 Ada.Strings.Bounded 的内部还有一个类属程序包 Generic_Bounded_Length 来提供 Bounded_String 的处理能力。因此如果要声明一个有界字符串类型，先要进行实例化。Ada Rationale 里关于为什么这样设计 Ada.Strings.Bounded 有很详细的解释。主要还是因为 Ada 的本身原因：如果和普通程序包一样设计，赋值操作就无法实现；而不相同长度的字符串之间无法互相赋值。

Ada.Strings.Bounded 提供的字符串处理子程序和 Ada.Strings.Bounded 差不多，略微有点变化，具体可见 附录 A。主要增添以下几个子程序：

function To_Bounded_String (Source : **in** String; Drop : **in** Truncation := Error) **return** Bounded_String;

如果 Source'Length <= Max_Length，返回一个表示 Source 的 Bounded_String。否则效果取决于 Drop 的值：

- 如果 Drop = Left，那么结果是一个 Bounded_String 由 Source 的最右边 Max_Length 个字符组成；
- 如果 Drop = Right，那么结果是一个 Bounded_String 由 Source 的最左边 Max_Length 个字符组成；
- 如果 Drop=Error，产生 Strings.Length_Error；

function To_String (Source : **in** Bounded_String) **return** String;

返回 Source 所表示的 String。如果 B 是一个 Bounded_String，则 B=To_Bounded_String(To_String(B))。

function Length (Source **: in** Bounded_string) **return** Length_Range；

返回 Source 的长度，Length_Range 是 Natural 的子类型。

function Element(Source: **in** Bounded_String;Index: **in** Positive) **return** Character；

返回 Source 中第 Index 个元素值，如果 Index > Length(Source),产生 Index_Error。

procedure Replace_Element(Source: **in out** Bounded_String; Index: **in** Positive； By: **in** Character);

更新 Source，使 Source(Index) 替换为 By；如果 Index>Length(Source)，产生 Index_Error。

function Slice (Source : **in** Bounded_String； Low**: in** Positive;High: **in** Natural) **return** String;

返回 Source 中从 Low 到 High 的字符串片断；如果 Low>Length(Source)+1,产生 Index_Error。

还有几个 Append 函数、过程，作用是在 Bounded_String 末尾增添字符、字符串。

14.6 无界限字符串处理(Ada.Strings.Unbounded)

相对于 String 的固定和 Bounded_String 的使用麻烦，Unbounded_String 对于大多数用户来说可能是最好的选择：Unbounded_String 的实现是指向 String 的访问类型，因此从某种意义上讲和 C 的字符串指针差不多，长度可自由改变，且为无限长度，同时又不会造成缓冲区溢出等问题。

Ada.Strings.Unbounded 提供的处理程序和 Ada.Strings.Bounded 差不多，具体见 附录 A。GNAT 还有两个 Ada.Strings.Unbounded 的子单元：Ada.Strings.Unbounded.Aux 和 Ada.Strings.Unbounded.Text_IO。

Ada.Strings.Unboundex.Aux 提供了下列 3 个转换程序：

```
function Get_String (U : Unbounded_String) return String_Access; );

procedure Set_String (UP : in out Unbounded_String; S : String);

procedure Set_String (UP : in out Unbounded_String; S : String_Access);
```

它们都是将 Unbounded_String 转换为 String，但相对于 To_String ，它们更加高效。

Ada.Strings.Unbounded.Text_IO 提供 Unbounded_String 的输入输出，因为标准库里没有提供该项功能：

```
function Get_Line return Unbounded_String;
```

```
function Get_Line (File : Ada.Text_IO.File_Type) return Unbounded_String;

procedure Put (U : Unbounded_String);

procedure Put (File : Ada.Text_IO.File_Type; U : Unbounded_String);

procedure Put_Line (U : Unbounded_String);

procedure Put_Line (File : Ada.Text_IO.File_Type; U : Unbounded_String);
```

输入输出我们会在下章涉及，不过这么简单的子程序使用应该不成问题。

14.7 宽字符和宽字符串处理(Wide Character and Wide String Handling)

宽字符和宽字符串处理的内容和前面大致相似，只是在原来程序包的名称上多了一个 Wide，如 Ada.Characters.Latin_1，Ada.Strings.Fixed，宽字符版本相对应的名称为 Ada.Characters.Wide_Latin_1，Ada.Wide_Strings.Fixd；子程序的名称基本上一样，如 Index，Count，但 To_Bounded_String 之类的子程序变为了 To_Wide_Bounded_String。具体内容读者可参见 附录 A