Delphi2010 语法手册

作者:wellsmart@live.cn

目　录

第一章 Delphi2010 入门

考虑到读者可能是第一次接触到 Delphi，我们在本章会介绍 Delphi2010 的一些极其基本的知识，包括：

- Delphi 中最常见的源文件的结构及相关的注意事项。
- 本书会用到的两种程序类型

可能看起来这些东西都非常浅显，但对于初学者而言，缺乏这些知识可能连在 Delphi 中运行本书上的示例都做不到。但是，若读者觉得自己这方面还可以的话，那就完全没有必要把时间花在第一章。根据我本人的经验，在阅读任何一本书时，第一章是最费时间的部分，若能用这种方法跳过第一章岂不是很好？

工程文件结构与语法

一个完整的可执行的 Delphi 程序由多个 unit 模块组成，这些模块被一个单一的源文件——工程文件所联系。不同于传统的 Pascal 程序，Delphi 中程序的主模块的源文件格式为.dpr，其它模块的格式为.pas。一个完整的程序 ＝ 一个.dpr 文件 + 0 或多个.pas 文件。

一个.dpr 文件的组成部分为：程序头、[uses 从句]、主程序块。（另外，在 IDE 中，一个工程中往往需要用到命名空间的限定语句。关于此我们后述。）一个典型的工程文件如下：

```
program Project2;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
end.
```

下面我们详细介绍这三个部分：

1.The Program Heading

程序头语句指示所编写的程序的名称。形式为：project 程序名; 在 RAD 中，此名称在任何时候均与.dpr 文件名相同。良好的程序中，.dpr 中的代码应当尽量少，尽量将代码分于.pas 中然后在.dpr 中调用。

2.uses 从句

在程序主文件中使用到的所有.pas 文件的名称必需列在 uses 从句中。其形式为：

```
uses  name1, name2, … ;
```

3.主程序块

Delphi 中用 begin 与 end 所包围起的一段代码称为一个代码块。工程文件含有一个主程序块用于存放主程序的代码。工程文件中的全局标识符必须声明于 begin 之前。

单元文件结构

程序的每个子模块称之为一个单元。单元中可包含任何元素包括类型、变量、常量、例程等。每个单元均存储于一个.pas 文件中。一个文件只能对应一个单元（就像一件衣服只能同时供一个人穿）。

单元文件的后缀名为.pas。每个.pas 文件包依次含以下部分：单元名称、Interface 关键字、implementation、 [initialization]、 [finalization]、关键字 end。一个完整的.pas 文件的结构如下：

```
unit Unit1;   //单元的名称
interface     //Interface 部分起始处
  uses …
……
implementation    //Implementation 部分起始处
  uses …
……
initialization       //initialization 部分起始处
……
finalization      //finalization 起始处
……
end.        //单元文件结尾（end 后必须以句号结尾）
```

1.单元名称

形式如 unit 名称;此处的名称与.pas 文件的名称也一致。注意一个.dpr 中不能引用两个名称相同的.pas。

2.Interface 部分

关键字 Interface 至关键字 implementation 之间的部分称为 Interface 部分。Interface 部分可定义变量、常量、例程等。单元中并非仅有此处才可定义标识符，但只有此部分中定义的标识符才可被其它文件访问。值得注意的是此部分定义的例程仅仅只需留下一个原型即可，具体的实现部分则放于接下来的 implementation 部分。此部分可以书写 uses 从句，其作用与.dpr 中相同。注意 uses 从句必须紧接于关键字 Interface 后，二者之间不允许有任何代码。

3.implementation

关键字 implementation 至关键字 initialization 之间的部分。此部分可书写例程的上体代码，但顺序上不必完全与声明时一致。此处实现时可以省略参数，但若未省略则必

须与声明时完全一致。与 interface 部分类似，此部分可以定义常量、变量、例程等，但只能在本单元内使用，本单元的 inteface 部分或其它单元无法使用此处定义的标识符。当然也可以合用从句，但只能紧跟关键字 implementation 之后。

4.initialization 部分

关键字 initialization 至关键字 finalization 之间的部分。此部分可选。 此部分中含有的代码在程序启动时运行，若多个单元中均含有，则按照工程文件 uses 从句中引用的顺序来运行。此关键字也可直接用 begin 代替，不过如此一为，将无法指定单元的 finalization，在此种情形下，可以用 EXITPROC 变量来模拟（将要执行的代码封装为一个过程然后将此过程赋予指针 ExitProc）。

5.finalization

finalization 至.pas 文件的末尾。此部分可有可无，仅当 initialization 部分存在时才可使用此部分。程序退出时运行此部分的代码，除非程序是由 Halt 例程强行退出。当多个单元均存在此部分时，执行顺序与 initialization 相反。

单元语法与 uses 从句

uses 从句中列出了所有使用到的单元的名称（不是源文件名称）。uses 从句可以出现（并非一定要出现）于 3 种场合，分别是：

- .dpr 文件中工程名称的后面
- .pas 文件 Interface 部分，必须紧跟于关键字 Interface
- .pas 文件 implementation 关键字之后

System 与 SysInit 单元被默认引用，故无需手动引用。在 IDE 中新建一个工程时，IDE 会自动引用一些常见单元以减轻代码编写量。

uses 从句中的单元寻找时按照一定的规则。

1.uses 从句的语法

uses 后接单元名称即可，单元名称间以逗号隔开，最后以分号结尾。如 uses until, unit2; 在.dpr 文件中，uses 后引用的单元名称后可加上源文件的名称如: unit1 in '…\unit1.pas'，in 用于指定单元所在的源文件。IDE 默认源文件为当前目录下的同名.pas 文件或是搜索路径中的同名.pas 文件，但有时并非如此，此时就需要 in 来限定。如下情形：

- 源文件不在当前目录及搜索目录中
- 不同的目录具有相同名称的源文件

在.dpr 中以 in 限定的源文件会被认为是工程的一部分。这不影响编译，但影响 IDE 对工程的管理。

在.pas 不能使用 in，这意味着此类型文件中引用的所有单元的源文件均在当前目录或搜索路径之中。

2.单元的多重引用及间接引用

单元在 uses 从句出现的顺序影响单元中 intialization 部分语句的执行顺序及编译器对标识符的定位。若多个单元中定义了相同的标识符，编译器取最后出现的单元，对于其它单元中的标识符，必须添加限定。

前面说过，uses 从句中列举出所有直接用到的单元名称。请注意这句话：直接用到。也就是说，若 unitA 用到了 unitB 中的标识符，则 unitB 必须出现于 unitA 的 uses 从句中，但若 unitB 使用了 unitC 中的标识符而 unitA 未用到 unitC 的标识符，此时 unitC 不必出现于 unitA 的 uses 从句中。

当单元的接口部分发生变体时所有依赖此单元单元的文件均需被重新编译，但若变体只是在实现部分，则不需要此过程。

3.单元的循环引用

当一个单元中引用了另一个单元时，称这两个单元为相互引用。Delphi 不允许循环引用。所谓循环引用，是指一个单元直接或间接地引用了自己，如：单元 A 引用了单元 B，单元 B 引用单元 C，而单元 C 又引用了单元 A。关于循环引用，最常见的错误是两个单元相互引用对方:A 引用 B，而 B 又引用 A。当然，无论是间接引用还是循环引用均是指 Interface 部分，对于 implementation 不存在这种说法。为避免循环引用，建议手动引用某个单元时，在 implementation 部分的 uses 从句中列出。

常见程序类型

1.命令行程序

启动 Delphi2010，依次点击：File—>New—>Other，此时弹出"New Item"对话框"，双击其中的"Console Application"，IDE 会自动新建一个包含一个.dpr 源文件的命令行程序，其代码为：

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

为了方便，我们将上述代码中粗体部分全部删除，将代码简化为如下形式：

```
program Project1;
```

```
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
end.
```

至此，我们得到了一个简单的程序。读者可按下键盘上的 F9 功能键或点击 Run 菜单中的 Run 命令来运行此程序。为了节省篇幅，在接下来的示例中，我们将省略程序名称及 uses 从句。也就是说，对于如下代码：

```
var
  x,y:integer;
begin
  x :=  9;
  y :=  7;
  writeln(x+y);
  readln;
end.
```

读者应当清楚，这段代码的完全形式为：

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;

var
  x,y:integer;
begin
  x :=  9;
  y :=  7;
  writeln(x+y);
  readln;
end.
```

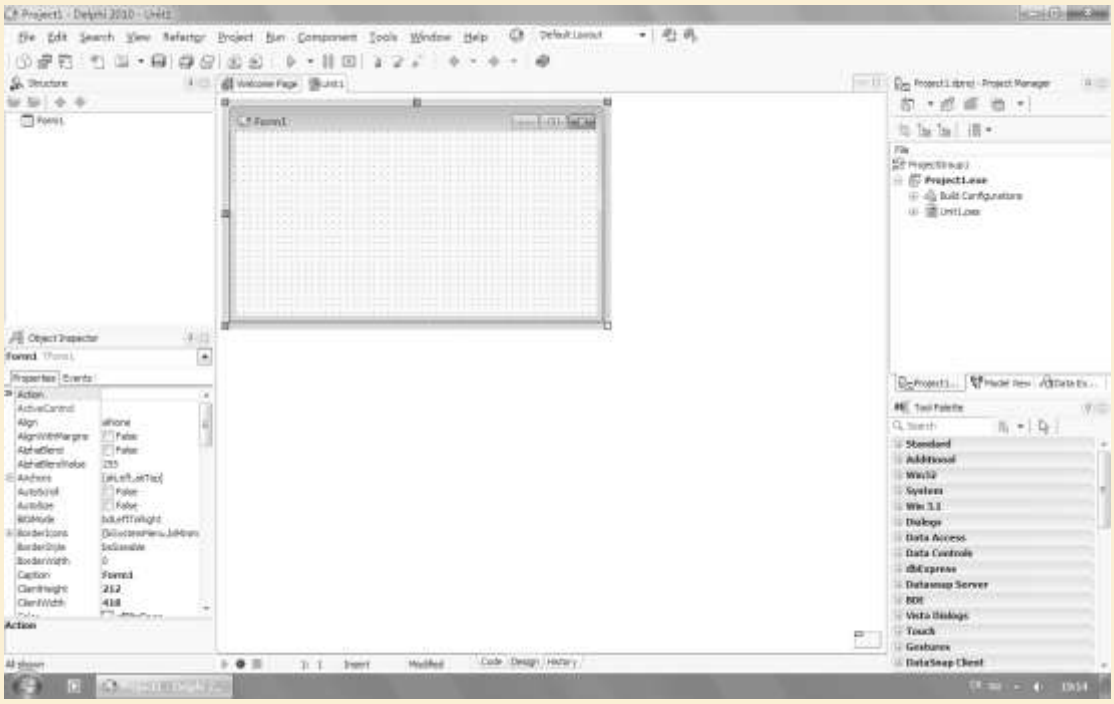
说明：

- 在命令行程序中我们使用传统的标准例程 writeln 在屏幕上显示内容。如上例中的 writeln(x+y)将在屏幕上显示 x+y 的值。同样，writeln('delphi')可在屏幕上显示 delphi。
- 标准例程 readln 用于使屏幕暂停，按回车键可结束暂停。读者可将此句代码去掉后运行程序以感受这句代码的作用。少数情形下 readln 可能无法暂停屏幕，此时可使用 read(V)代替。其中的 V 是在 var 中声明的变量。如上例中的 readln 也可替换成 read(y);不过在这种情形下，按下回车键并不能结束暂停，最保险的方法是用鼠标手动关闭命令行屏幕。

2.窗体程序

在 NewItem 对话框中双击"VCL Forms Application"，IDE 自动新建一个窗体程序。如

下图所示：



注意到新建的窗体周围有 6 个方形块，拖动右下角的空心方形可调节窗体大小。

自 IDE 右上方的 Project Manager 工具窗口可看到此工程包含一个 Unit1.pas 源文件。IDE 容器的底样有三个标签，分别是 Code/Design/History，单击 Code 标签可查看此窗体所对应的单元文件即 Unit1.pas 的代码：

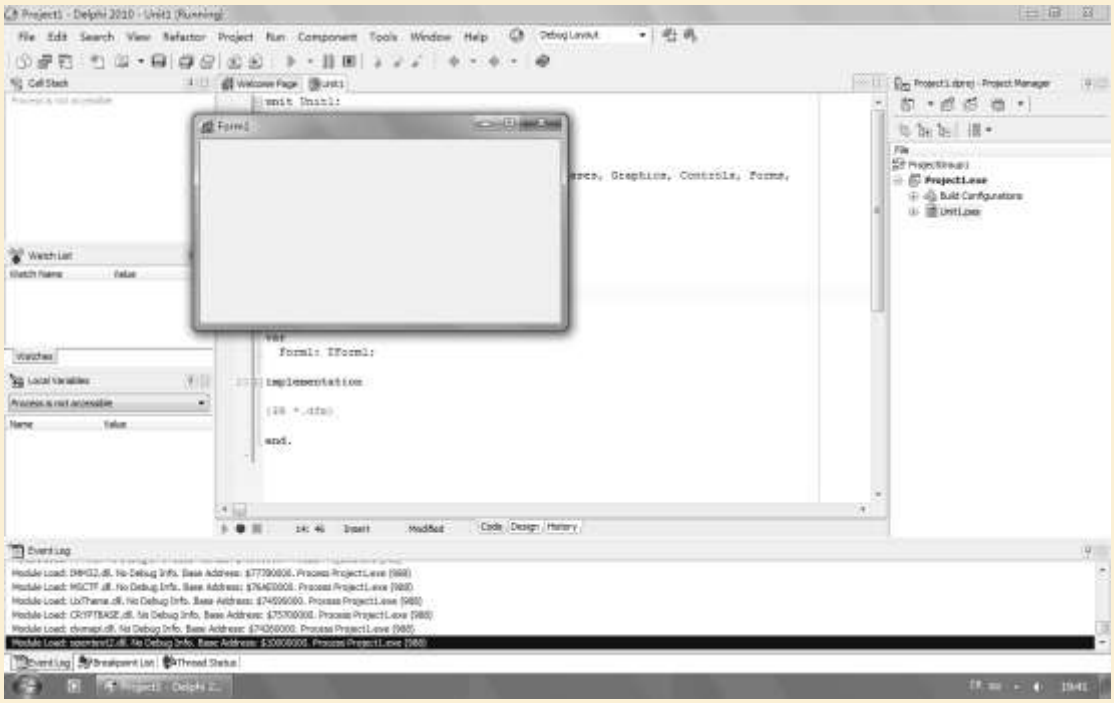
```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,   Forms,
Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
end.
```

若要查看此工程的.dpr 文件可用鼠标指向此窗口中的 Project1.exe，选择 View Source 命令即可。其源代码为：

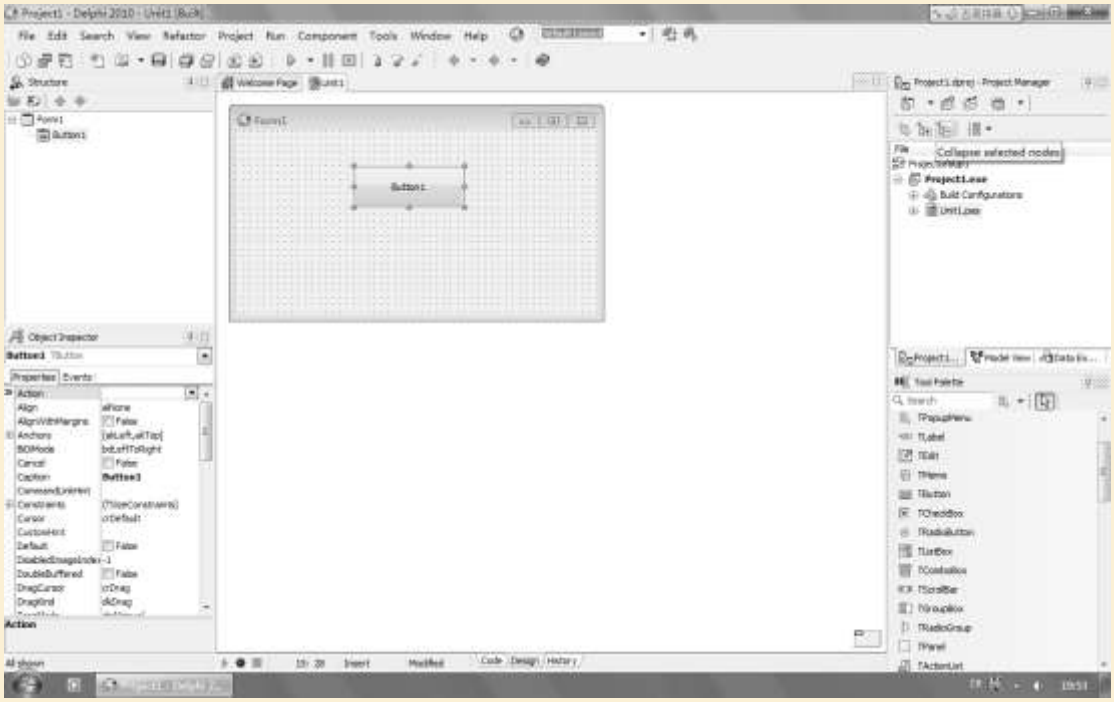
```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
  Application.Initialize;
  Application.MainFormOnTaskbar :=  True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

读者可将此两份代码与前面所介绍的.dpr 文件与.pas 文件的基本结构对照看一下是否符合。现阶段我们只使用工程的.pas 文件而不需要使用.dpr 文件。

同命令行程序一样，可以按下 F9 或选择 Run 菜单下的 Run 命令来运行这个窗体程序，得到的结果是一个空白的窗体如下图：



当工程处于 Design 状态时可在 IDE 右下方的 Tool Palette 中选择控件，然后在空白窗体中按下鼠标并拖动，如在 Tool Palette 的 Standard 类别中选择 TButton 控件然后拖动鼠标，可以在窗体上添加一个按钮，结果如图：



按钮周围的 8 个蓝色小圆珠有两个作用：一是表示当前焦点在此按钮上，二是通过拖动这些小圆珠可调节按钮大小。这与窗体周围的 6 个小方块的作用相同。

当焦点停留于窗体或窗体内的控件上时，IDE 左下方的 Object Inspector 工具窗口中会自动显示当前控件或窗体的属性，如标题、名称、大小等，可以手动设置这些值。

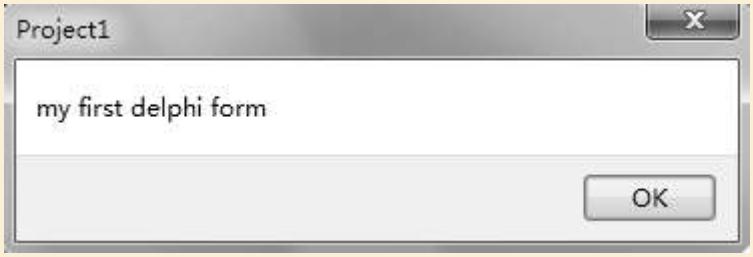
注意 Object Inspector 工具窗口亦有两个标签分别是 Properties 与 Events，默认情形为 Properties，表示此工具窗口中显示的是控件的属性；点击 Events，工具窗口中将显示控件的事件，其中有 OnClick、OnEnter 等，双击 OnClick 右侧的下拉列表，可在窗体对应的.pas 文件中产生此事件的框架：

```
procedure TForm1.FormClick(Sender: TObject);
begin


end;
```

当运行时点击按钮即可触发此事件。例如在其中加入显示对话框的代码：

```
procedure TForm1.FormClick(Sender: TObject);
begin
  showmessage('my first delphi form');
end;
```

按下 F9 运行并点击按钮，会弹出如下对话框：



第二章 Delphi 语法基础

2.1 标识符

简单来说，标识符就是一个合法的名称，用于替变量、常量、函数甚至是数据类型等命名。按种类可分为标准标识符及自定义标识符。

2.1.1 标准标识符

所谓标准标识符是指由 Delphi 预先定义的标识符，用于标识 Delphi 预先定义的函数，变量等。主要有以下几种：

1. 标准常量及变量名称，如 FALSE、TRUE 等。
2. 标准类型名称，如 Integer、Real 等。
3. 标准例程名称，如 Sin、IntToStr 等。
4. 标准文件名称，如标准 I/O 名称 Input、Output 等。
5. Delphi2010 的保留字及指令符，如 procedure、class 等。

在选择标识符时，应尽量与标准标识符相冲突，否则可能会造成想不到的结果。

2.1.2 自定义标识符

顾名思义，自定义标识符就是程序员根据需要所定义的名称，一个合法的自定义标识符需满足以下条件：

1. Delphi 语言不区分大小写，标识符亦然，如 PASCAL 与 pascal 将被视为同一标识符。
2. 自定义标识符不能和当前域中的其它标识符相同。
3. 标识符长度应小于或等于 255 个字符。若超出此长度，超出的部分将被舍弃，只取前 255 个字符。
4. 标识符由英文字母、下划线、数字组成，不包含空格，第一个字符不能是数字。

实际上，Delphi2010 采用 unicode 字符集，其标识符可由任何 unicode 中的字符组成，但有一个小小的例外：

Unicode 的前 256 字符对应于 ansi 字符集的 256 个基本字符，在这些字符的前 128 个字符中，只有英文字母和数字及下划线'_'可用于标识符，其它如英文标点、空格等字符均不能用于标识符。

如英文标点‘?’不可作为标识符，但中文中的‘？’却可以。因为中文的‘？’不属于英文字符。这也是为什么在有些早期版本中不支持中文作为文件名的原因。

之所以不能使用英文字符，是由于英文中的符号往往有其特殊用途，如‘:’用于声明变量，‘$’用于标识十六进制数字，这类系统预定义的符号称之为特殊符号。Delphi2010 中的特殊符号有：

```
# $ & ' ( ) * + , - . / : ; < = > @ [ ] ^ { }
```

以下的组合符号亦为特殊符号：

```
(*  (.  *)  .)  ..  //  :=  <=  >=  <>
```

其中，‘(.’组合等效于‘[’，‘.)’组合等效于‘]’，‘(*’组合等效于‘{’，‘*)’组合等效于‘}’。

除此之外，以下符号虽非特殊符号，但亦不能用于标识符：

```
%  ?  \  !  ″  |  ~
```

2.1.3 标识符的作用域

如同中国的法律只能使用于中国境内，每个标识符只在某个特定范围内有效。按照有效范围的大小可将标识符分为全局标识符及局部标识符。

局部标识符是指定义于例程(函数或过程)中的标识符，此种标识符只能用于定义它们的例程。举例子来说，局部标识符类似于北京市出台的政策，这些政策只能用于北京。

除局部标识符外的所有标识符均为全局标识符。定义于.pas 文件的全局标识符按照定义时所处位置分类，可分为公有及私有标识符。所谓公有标识符是指定义于.pas 文件的 Interface 部分的全局标识符，其它为私有标识符。二者在定义它们的.pas 文件中的有效范围均为定义时所处的位置到.pas 文件的末尾。不同的是，公有标识符不但在本.pas 文件中有效，在所有引用了本.pas 文件的其它文件如.dpr 文件中也有效，故称其为公有。

2.2 保留字与限定符

所谓保留字是指由 Delphi 预先规定只能由 Delphi 使用的单词，如最为常见的 begin、end 等等。理所当然地，这些保留字不能作为自定义标识符的名称。读者并不需要记住这些保留字，默认情况下，Delphi2010 的 IDE 会自动将这些单词以深蓝色显示，使用时稍加留意即可。

指令符不同于保留字，这些单词只在特定的位置上有特殊意义，如 virtual 只在定义虚函数时作使用。在其它位置，指令符与一般标识符没有区别。随意使用指令符作为标识符可能会导致意料之外的结果，故不建议使用指令符作为标识符。与保留字相同，当指令符位于其特定位置上时，IDE 将会以特定颜色显示。

为方便叙述，在本书我们将用关键字同时代表保留字与限定符。

Delphi2010 的保留字及指令符如下：

表 1　Delphi2010 保留字

and	else	inline	property	try
array	end	interface	raise	type
as	except	is	record	unit
at	exports	label	remove	until
asm	file	library	repeat	uses
begin	finalization	mod	resourcestring	var
case	finally	nil	set	while
class	for	not	shl	with

const	function	on	shr	xor
constructor	goto	of	strict private	
destructor	if	or	strict protected	
dispinterface	implementation	out	string	
div	in	packed	then	
do	inherited	procedure	threadvar	
downto	initialization	program	to	

表 2 Delphi2010 指令符

absolute	dispid	helper	near	private	reintroduce	stored
abstract	dynamic	implements	nodefault	protected	requires	unsafe
assembler	experimental	index	operator	public	resident	varargs
automated	export	inline	overload	published	safecall	virtual
cdecl	external	library	override	read	sealed	winapi
contains	far	local	package	readonly	static	write
default	final	message	pascal	reference	stdcall	writeonly
deprecated	forward	Name	platform	register	strict	delayed

注：指令符 private、protected、public、published 及 automated 在定义一个类时被视作保留字，但在 其它场合则被视为指令符。

2.3 常量

所谓常量就是在第一次赋值后不能改变其值的量。Delphi 支持两种类型的常量：一种是直接常量；另一种是声明常量。除此之外，Delphi 本身还预定义了一些常量，如常用的 False，True 以及 Nil 等。

2.3.1 直接常量

直接常量是指在程序中直接引用的量，这种常量不需要有标识符来表示，其本身就是一个确定值，如数字 123 只能代表这个数值，字符串‘abcd’也只能代表这个字符串一样。常用的直接常量类型及书写规则如下：

(1). 整型常量：即一个数学整数。如 123，43 等。其前可加负号‘-’表示负整数。整数加上符号‘$’可表示十六进制数，如$12 表示十六进制数 12，转化为十进制数为 18。

(2). 实型常量：即数学上所说的小数，由数字和小数点构成。注意，若无小数点，系统会将常量解析成整数，故小数点不可缺。实型常量还可用科学记数法表示，如 0.0023 用科学记数法表示为 2.3E-3，E 表示 10 的次方，如 1.2E3 表示 1200。

注意：所有带小数点的实数全部被推断为 Extented 类型

(3). 字符及字符串型常量：任意字符，若两边以单引号包围，即为字符或字符串。若只有一个字节，则系统将其定义为字符；若大于一个字节，则定义为字符串。如‘a’为字符，‘12345’为字符串。

(4). 布尔型常量：此类型常量常用的只有二个值：FALSE 和 TRUE，分别表示逻辑真和

逻辑假。

2.3.2 声明常量

与直接常量不同，声明常量必须用一个合法标识符表示，且常量在定义时必须同时赋值，一旦定义后代表， 此标识符的值不可再次被更改，否则会导致编译错误。声明常量可分为符号常量及类型常量。

1. 符号常量

符号常量亦称为真常量。此类型常量用一个标识符代表一个具体值，编译器在解析代码时，遇到此标识符直接将其替换成所代表的值，这有点类似于一个值不变的变量。声明符号常量可使用下列格式的语句：

```
Const
    〈标识符 1〉 = 〈常量值 1〉;
        …
     〈标识符 n〉 = 〈常量值 n〉;
```

Const 是 Delphi 的保留字，表示定义常量的开始，在书写时 Const 也可单独写成一行，不过为了代码的美观性，建议参照本书的代码书写风格。

常量值可以是直接常量或其它符号常量，也可以是二者组成的表达式。如下面的下面的声明均合法：

```
Const
    CONS_A = 123;
    CONS_B = CONS_A+234;
    CONS_C=CONS_A+CONS_B+ 89;
```

编译器在编译时将根据值的类型推测常量的类型。也就是说，上面声明的三个常量均属于数值型，若将其中的 CONS_B 的声明改成：

```
CONS_B = CONS_A + ' delphi ';
```

编译将会出现错误，因为编译器将 ' delphi ' 识别为字符串型常量，将 CONS_A 识别为是数值型常量，二者不能相加。

类似于直接常量，编译器在编译代码时会按照与直接常量相同的规则推断符号常量的类型。

2. 资源字符串

在声明字符串常量时可以使用 resourcestring 来声明：

```
resourcestring
  str = 'The Current Edition Is Delphi2010';
```

这种方式声明的字符串用于程序中时将被编译至资源文件中。这样做的好处是可以任意修改字符串而不需要重载编译程序。

3. 类型常量

符号常量虽然方便，但它有两个弱点：

其一，它只能表示一些简单的值，较复杂的类型如函数指针等很难用符号常量表示。

其二，不能精确地控制其类型。例如，若要定义一个能容纳 3 个字符的字符数组，按如下方式定义：

```
Const
    ChArray = 'abc';
```

编译器会将常量 ChArray 理解成是一个字符串而并非字符数组。

为了解决这些不足，可以使用类型常量。与符号常量不同，类型常量在声明时必须显式指定其类型并赋值。其声明格式如下：

```
Const
    <标识符 1> : <类型 1> = <常量值 1>;
        …
    <标识符 n> : <类型 n> = <常量值 n>;
```

其中的标识符的规定与符号常量一致，常量类型可以是内置类型，也可以是自定义类型。常量值必须与指定的类型一致或兼容于指定的类型，所谓兼容是指常量值的类型经类型转化能够转化至声明语句中指定的类型。注意，不能将常量声明为文件类型和变体类型。与符号类型不同，类型常量的常量值中不能含有其它常量。如：

```
Const
    Name: string = 'delphi2010';  //合法
    Caption: string = 'my'+name;  //不合法，name 是另一个常量，不可出现在表达式中
    MultiPi: real = 2*3.14;  //合法，MultiPi 的值为 6.28，‘*’代表乘号
```

若指定的类型是数组、记录、过程或指针，则另有特定的规则。

[注]：IDE 中开启{$J+}编译命令时，类型常量可被重新赋值，此时类型常量与普通变量没有太大区别。但在 Delphi2010 版本中，此开关默认关闭。若无意中开启，只需在程序中加入{$J-}命令即可关闭。

Delphi 中有些类型由于本身的缘故，其常量声明并非那么直观。接下来我们简要看一下几种特殊类型的常量的声明，这需要后面的数据类型的相关知识，所以若读者看的不是很明白也没关系。

4. 数组常量

数组常量的声明主要有三种：静态数组、字符串、多维数组。

声明静态数组常量时多个成员值之间以逗号隔开，所有的值以一对圆括号围起：

```
const
  Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

序数从 0 开始的字符数组与 Null 结尾的字符串兼容，将一个字符数组常量初始化为字符串时，可直接赋值：

```
const
  Digits: array[0..9] of Char = '0123456789';
```

声明多维数组常量就比上面要麻烦一点，如对于如下声明：

```
type
  TCube = array[0..1, 0..1, 0..1] of Integer;
```

在声明这个多维数组的常量时需要从最低维到最高维进行赋值。

首先，第一维有二个成员，所常量的表示形式为（X，Y）。第一维中的每个成员本身又包括两个成员，所以无论是 X 还是 Y，其形式都是（X1，Y1）。所以 TCube 的常量形式

又可写成（（X1，Y1），（X2，Y2））。TCube 是三维数组，所以 X1，Y1，X2，Y2 又有两个成员：（X11，Y11）。这样一来，整个的数组展开后的形式为：

（ （（X11，X12），（Y11，Y12）），　（（X21，X22），（Y21，Y22））　）。

所以 TCube 类型的常量 Maze 的声明为：

```
const
  Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

这个常量数组中各成员的值为：

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

注意：数组常量不能含有文件变量，数组的基类型也不可以。严格来说，数组常量不能以任何形式含有文件变量。

5. 记录常量

声明记录常量时需要以“字段名：字段值”的形式指定每个字段的常量值，各字段间以分号隔开。各字段的赋值顺序必须与声明时的顺序一致。

对于变体记录，若其中出现了一个 tag 值，则必须赋值；只有含有 tag 值时变体记录中的变体部分才能被赋值。

与数组一样，记录常量中不能含有任何形式的文件变量。

```
type
  TPoint = record
     X, Y: Single;
  end;
  TRec = record
    x:integer;
  case tag:integer of
    1:(i:integer);
    2:(n:integer);
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  VRec: TRec = (x:7; tag:1; i: 2;);
```

6. 指针常量

指针常量在声明时可以使用任意一个变量的地址作为常量值。对于 PChar 或 PWideChar，声明时可直接使用一个字符串序列作为常量值。

指针常量定义后，指针所指向的变量值可被改变，但指针不允许重载指向其它变量。如对于如下声明：

```
var
  i, n: integer;
```

```
const
  pi: ^integer = @i;
```
i 的值可随时改变，但指针 pi 只能指向 i，不能再指向其它变量。

2.4 变量

变量类似于符号常量，不同的是变量可以被重新赋值。根据存储方式可将变量分为动态变量与静态变量。静态变量在定义时就已经确定其所需内存的大小，而动态变量则在运行期间根据需求分配合适的内存。

1. 变量的声明

其声明格式如下：

```
Var
   <标识符 1> : <类型 1>;
       …
   <标识符 n> : <类型 n>;
```
声明多个类型相同的变量可采用如下简化形式：
```
Var
  <标识符 1>,…<标识符 2>：<类型>;
```

声明语句中，Var 保留字表示声明的开始。标识符为任意合法标识符，但习惯上选取含义直观的标识符，这样做可增加代码的可读性；类型可以是内置或自定义类型。

在某些时候我们可能需要声明一些变量或例程，其名称与 Delphi 中的某个关键词相同，在这种情况下，我们可以在这些变量名前加上符号"&"来区分这些名称，如：
```
var
  &begin:Integer;
```

2. 变量的初始化

声明变量后应对其进行初始化，最常见的初始化方式是对其赋值。变量的赋值可使用符号’ := ’其格式为：

变量名 := 变量值;

```
    如 v := 3;
```
赋值语句右边也可以是表达式。如：
```
    v := 3*4;
```
或
```
    v := function(3); //假设 function 为一函数，其返回 3 的平方
```

对变量进行赋值时，若所赋之值不在变量的值域之内，会出现两可能的情况：

(1)：编译器提示错误。此类错误可能是所赋之值与变量并非同一类型，也有可能是所赋之值太大或太小。

(2)：编译器不给出错误，但计算机会将所赋之值与变量能够表示的最大值进行求余运算，并将结果赋给变量。

变量手动赋值前会由系统赋值，对于不同类型的变量系统有不同的规则。对于全局变量，系统一律初始化为0，若是指针则初始化为nil；若是字符串则初始化为一个空字符串。对于局部变量系统会随机赋值，但这种赋值是很不可靠的，所以对于局部变量一定要手动初始化之后才能用。

在声明全局变量时也可直接手动进行初始化。其格式为：

```
变量名：类型 ＝ 初始值；
```

例如：

```
Var
  GlobalVar: integer = 100 ;
```

注意：GlobalVar必须为全局变量，且不能为变体或文件类型。采用简化形式在同一行声明多个同类型变量时不能手动初始化。

3. 共址变量(Absolute Addresses)

共址变量是这样一种变量：声明时可以将其与其它的某个变量相绑定，绑定后二个变量在内存中存储位置的起始地址相同。声明共址变量的格式为：

```
var
  v1:type1 absolute v2;
```

其中v2是一个声明过的变量。程序在编译时会将v1与v2的起始地址设置为相同位置。我们以一个例子来说明：

```
var
  c: AnsiChar;
  i: Byte absolute c;
begin
  c :=  'A';
  writeln(i); //屏幕显示65
  readln;
end.
```

上例中i被声明成变量c的共址变量，这样一来变量i与变量c将在内存中的起始位置相同。i在内存中占用一个字节，当攻取i的值时，系统会将变量c的第一个字节的值当成是i的值，而c中第一个字节的值是字符‘A’，故i的值为65。

同理，下面例子中变量leng的值为字符串str的第一个字节的值(即为字符串的长度)：

```
var
  str: ShortString;
  leng: Byte absolute str;
begin
  str :=  'Delphi';
  writeln(leng); //屏幕显示6
  readln;
end.
```

需要说明两点：

1. 共址变量具有相互性，若将上例中的声明改为如下形式结果将不会发生变化：

```
var
  i: Byte;
  c: AnsiChar absolute i;
```

2. 共址变量并非仅限于两个变量，可以声明多个变量为共址变量：

```
var
  i:Byte;
  c:AnsiChar absolute i;
  str: ShortString absolute i;
```

2.5 类型声明

除了变量及常量，在必要时我们还可以自己定义一个数据类型用于声明变量和常量。声明一个新的数据类型必须使用关键词 type，格式如下：

```
type
  类型名称 ＝ 类型表达式;
```

类型名称依然可以是任意一个不与其它标识符冲突的合法标识符，类型表达式可以是一个结构如 string[3]、array of char…，或者是另一个类型名称。下面声明了两个自定义类型：

```
type
  aSet = set of char;       //声明一个子界类型 aSet
  myInteger = Integer;  //声明一个整数类型，其名为 myInteger
```

若类型表达式是另一个类型名称，则在我们声明的新类型与这个类型完全等同，相当于给同一种数据类型指定了另一个名称。如上面声明的 myInteger 与 Integer 类型在任何时候都完全等同。举个例子，数据类型的名称就相当于是一个人的名字，myInteger 与 Integer 均是同一个人的名字。

Delphi 定义数据类型时还有另一种方式，这种方式与上面的方式有微妙的区别，其格式为：

```
type
  类型名称 ＝ type  类型表达式;
```

我们用这种方式重新声明 myInteger：

```
type
  myInteger = type Integer;
```

其中的 myInteger 与 Integer 不再是同一个人的名称，而两个人的名称。 只不过这两个人的各个方面均完全相同（除了名称）。一般情形下这种细小的差别不会被注意到，但在牵涉到诸如数据类型的动态判断之类的较高级的用法时，这种差别会导致类型的不兼容。

当类型表达式是一个结构时，任意两个以结构本身声明的数据类型间均有差别，如下面声明了两个子界类型：

```
type
  aSet = set of Char;
  bSet = set of Char;
```

从表面上看来，aSet 与 bSet 应当完全一致，但事实并非如此，bSet 与 aSet 之间相当于以下声明：

```
type
  aSet = type bSet;
```

或是

```
  bSet = type aSet;
```

若要 aSet 完全等同于 bSet，可使用如下的声明方式：

```
type
  aSet = set of Char;
  bSet = aSet;
```

2.5 注释

从作用上来说，注释用于提供代码的相关信息使用得读者阅读时能够更容易理解代码的含义。编译时编译器会忽略所有的注释部分。Delphi 中提供三种注释方式：//…、{…}、(*…*)，如下所示：

```
var
  I, J, V, W: Integer;
begin
  I := 4;   //I 的值为 4
  V := 4;
  J := DoubleByValue(I);    { J = 8, I = 4 (*DoubleByValue 的参数通过值传递*) }
  W := DoubleByRef(V);      (* W = 8,{DoubleByRef 的参数通过引用传递}
                               V = 8 *)
end;
```

其中//…只能用于单行注释，其余两种则可用于多行注释。

在注释中套嵌其它注释时必须用不同的注释方式。如上面例子中的{ J = 8, I = 4 (*DoubleByValue 的参数通过值传递*) }，在{…}中套嵌其它注释时只能选用(*…*)或//…方式。

注：当以 (*…*) 或 {…} 注释时，若注释内容的第一个字符为’$’，此注释表示编译器提示符，用于告知编译某些信息。如{$APPTYPE CONSOLE}表示当前程序是一个命令行程序

2.6 表达式

能够返回一个确定的值的合法语句均可以称之为表达式。如：

```
3*4         //数学表达式，其值为 12
78 < 90       //逻辑表达式，其值为 True
Char(65)   //数值转型，其值为字符’A’
```

常见的最简单的表达式为变量及常量。表达式必须有一个返回值，Delphi 的过程调用不返回任何值，故过程调用不属于表达式。同理，赋值语句：

i := 123;

此语句不返回任何值，故而也不是表达式。而逻辑判断语句：

I >= 7;

此语句返回逻辑值(TRUE 或 FALSE)，故属于表达式。

注意：Delphi 中的表达式不能单独存在，只能位于赋值符号“ := ”的右边。

2.7.语句

语句定义了程序中的一个演算行为。简单语句如赋值或函数调用可被包含进循环、条件或结构语句中。块、单元的 initialization 及 finalization 部分中的多条语句中用分号隔开。

Delphi 中的一条语句由一个或多个表达式、关键字或者运算符（符号）组成。一般来说，在编写代码时一条语句占一行，不过这并非绝对，在需要时也可以将两条或更多条语句也可以写在同一行上，每行语句应当以分别结束。

2.7.1 简单语句

简单是指不包含任何其它语句的语句，常见的简单语句有赋值语句、例程调用语句及 goto 语句。

1. 赋值语句

赋值语句的形式如下：

```
变量名 := 变量值;
```

句中的” := ”称为赋值符号。变量名可以为任何变量。变量值可以是一个常数值，可以是另一个变量值，也可以一个函数，但无论如何，赋值符号右边的部分必须能返回一个值。

2. 例程调用

对于不接受参数的过程和函数，在调用时可省略例程名后的括号。如：

```
procedure M1;
...
 M1;
...
```

对于函数，调用时既可以将其返回值赋给一个变量，也可单独调用：

```
procedure M1:integer;
...
var
  i:integer;
...
```

```
  i :=  m1;
  m1;
...
```

3. Goto 语句

Goto 语句可使程序离开当前位置直接跳至某个特定的语句执行，其形式为：

```
goto label;
```

后面的 label 表示语句标签，执行 goto 语句会直接导致程序跳到 lable 标记的位置。

大部分能够声明变量的地方都可声明一个标签，声明格式为:

```
label 标签;
```

标签可以是任何一个合法标识符或一个 0 至 9999 间的数值。理所当然，Goto 语句及其中的标签必须在当前范围内必须都有效，鉴于此，在窗体程序中不能定义一个全局标签，除非是在单元的 initialization 及 finalization 部分。

由于 Goto 语句不利于程序的调试，而且它会造成程序的某些不可捉摸的行为，所以不推荐使用 Goto 语句。

下面是 Goto 使用的一个例子：

```
var
  i:integer;
label 1,2;
begin
1:writeln('请输入 i 的值：');
  readln(i);
  if i <> 0 then
    goto 1;
  exit;
end.
```

这个程序会要求输入一个数字作为 i 的值，若输入的值为 0 则程序自动退出，若输入的值不为 0，则程序会要求再一次输入 i 的值。

2.7.2 结构语句

结构语句由多条简单语句组成。Delphi 中的结构语句包括复合语句、with 语句、条件语句、循环语句等。本部分只介绍复合语句和 with 语句。其它类型语句会在后续部分介绍。

1. 复合语句

多条简单语句使用 begin 和 end 包围后即为复合语句，如：

```
begin
  Z :=  X;
  X :=  Y;   //可以省略分号: X :=  Y
end;
```

如果读者比较懒，那么复合语句中最后一条简单语句后的分号可以省略。

一个复合语句中可以含有其它任何类型的结构语句。在必要时也可以定义一段空的复合语句：

```
begin
end;
```

2. 汇编语句

与很多其它的工具一样，Delphi 支持在其中嵌入汇编代码。Delphi 的汇编代码使用 asm 与 end 包围：

```
var
  a:word;
begin
  asm
    mov ax,43;
    add ax,54;
    mov a,ax;
  end;
  writeln(a);
  readln;
end.
```

3. With 语句

本部分较多的牵涉到后面的内容，所以如果读者不熟悉的话，建议看完后面的内容后再回过来看一下本部分的内容。

先来看一个例子。如对于如下声明：

```
type
  TDate = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;
var
  OrderDate: TDate;
```

你可以使用如下的方式来访问其中的字段：

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month :=  1;
  OrderDate.Year :=  OrderDate.Year + 1;
end
else
  OrderDate.Month :=  OrderDate.Month + 1;
```

这个例子中使用了 OrderDate 中的多个成员，每个成员均使用对象名来限定。虽然这样没

什么问题，但每次都要键入一启蒙对象名是不是有点烦呢？

使用 with 语句可以避免过多的输入。利用 with 语句我们可以将这些代码写成如下形式：

```
with OrderDate do
  if Month = 12 then
    begin
      Month :=  1;
      Year :=  Year + 1;
  end
  else
Month :=  Month + 1;
```

如何？读者喜欢哪种写法？

通过这个例子，可以发现使在引用同一标识符的多个成员时，使用 with 语句可以极大的方便编码，其声明方式为：

```
 with obj do 语句
 with obj1, obj2 do 语句
```

其中的 obj 可以是任何能引用字段的标识符，如记录名、对象名、接口名等，还可以是指针的解引用形式、引用等。其后的语句可以简单语句，也可骒复合语句。

对于 with 语句出现的所有的单独的标识符，编译器优先在 obj 中寻找其含义，换句话说对于如下编译：

```
type
  TRec = record
    x,y:integer;
  end;
var
  obj:TRec;
  x:integer;
begin
  with obj do
  begin
    x :=  3;
  end;
  writeln(obj.x);
end.
```

可以看到，在 with 语句中赋值的是 obj 中的 x 成员。若要在 with 中赋值给全局变量 x，可以使用限定形式上述的赋值代码改为如下形式：

```
  project1.x :=  3;
```

当 with 后有多个 obj 时相当于不断的套嵌，例如 with obj1, obj2, ..., objn do ...相当于：

```
with obj1 do
  with obj2 do
    ...
    with objn do
      ...
```

在这种情形下，编译器会从最内层的 objn 开始寻找标识符，若未寻找到则在外面一层的 obj 寻找，依次类推，一直到 obj1，若还未寻找到，则在 with 语句之外寻找。如下面的例子所示：

```
type
  TInner = record
    x,y:Integer;
  end;
  TRec = record
    x,y:integer;
    z:string;
  end;
var
  Rec:TRec;
  Inner:TInner;
  x:integer;
  z:string;
begin
  with rec,inner do
  begin
    x :=  3;
    z :=  'delphi';
  end;
  writeln(Inner.x);
  writeln(rec.z);
  readln;
end.
```

2.8 块和域

2.8.1 块（Block）

Delphi 中大部分的标识符声明及执行语句都被组织成块，然后组成整个源代码。使用块的最大好处就是可以使同一个名称可以在不同的地方表示不同的含义。每个程序、函数、过程均含有一个块。

一个块由声明部分与语句部分组成，语句部分使用 begin 与 end 围起。块的一般形式为：

```
    ...//声明部分
    begin
    ...//语句部分
    end;
```

块的声明部分可以声明任何标识符，包括变量、常量、函数、数据类型及标签等。这些声明原则上没有先后之分。在某些源文件中可能还会看到声明部分出现了若干个 exports 从句。

下面是一个函数的声明：

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  ...
end;
```

这个声明中的第一行称为函数首部，它声明了一个函数名称 UpperCase。这行语句与其后的 4 个变量的声明一起构成了块的声明部分。

块中声明的标识符具有局部性，它们只当前的块中。如上面的示例中声明的四个变量 Ch、L、Source、Dest 均只能用于声明它们的块当中。

2.8.2 域（Scope）

域指标识符的有效范围。声明于不同位置的标识符具有不同的有效域。

声明于块中的标识符只能用于这个块中，这些标识符称为局部标识符，若是变量则为局部变量，若是常量则为局部常量。

声明于单元的 Interface 部分的标识符的有效域为任何一个引用这个单元的源文件。这种标识符称为全局标识符。若这个标识符是一个变量名，这个变量为全局变量；若为常量名则为全局常量……。

我们使用下面这张表来总结标识符的域：

标识符的定义位置	**有效域**
程序、函数、过程的声明部分	从定义位置起始，到块的末尾，包括所有的子块
单元的 Interface 部分	从定义位置开始，到单元的末尾，也可以是任何引用了这个单元的其它单元
单元的 Implements 部分，但不在任何块内部	从定义位置开始，到单元的末尾，包括此部分中所有的块内部，以及单元的 initialization 和 finalization 部分。
记录的内部	从定义位置到记录末尾

2.8.3 名称冲突

当一个块包含了另一个块时，前者称为外部块，后者称为内部块。内部块的标识符会掩盖外部块的标识符，例如在下面的代码中，被赋值是声明于函数 M2 中的 s：

```
function M1:integer;
var
  s:string;
```

```
  function M2:integer;
  var
    s:string;
  begin
    s :=  'delphi';
  end;

begin
end;
```

在第一章，我们介绍了程序或单元可以通过 uses 从句来引用其它单元。这种特性使得编译器在确定标识符的含义时需要进行更为复杂的搜索：uses 从句中列出的每个外部单元都替当前单元或程序强加了一个新的域。换句话说，由于 uses 的作用，使得我们可以访问一些其它单元中的原本不能访问的标识符。

为解决这个问题，Delphi 规定：uses 从句中的第一个单元为最外层，其次为次外层...。但出乎你的意料，最内层的单元并不是 uses 从句最后出现的单元。所谓近水楼台先得月，正在使用这个标识符的的单元才是最里层的单元。这个规定有一个例外：system 和 sysinit 单元会在每个程序或单元中自动引用，它们永远是最外层单元。

若两个或更多的单元在 interface 部分中声明了同一标识符，则任何一个未限定的标识符都会被认为是最里层的标识符。限定标识符也很简单，直接使用“名称.标识符”的形式即可，如 Module.Fun 表示目前引用的是 Module 中的 Fun。其中的 Module 可以是单元名或工程名，也可以是结构类型的变量名。

2.9 类型兼容与赋值兼容

类型兼容性

当 A、B 两种数据类型满足以下条件时，可以将 B 类型的值赋给 A 类型的变量，这种情形称之为类型兼容。

1. 它们都是实数类型
2. 它们都是整数类型
3. B类型是A子界类型，例如A是Integer类型，而B是byte类型。
4. 两个都是同一种类型的子界类型，例如byte与smallint均为integer的子界类型，故byte类型的值可赋给smallint类型的变量，反之亦可，此种以情形下应当注意B类型的值不能超出A类型所能表示的最大值。例如若A是byte类型、B是word类型，在将B类型的值赋给A类型的变量时，B类型的值最好不要大于255，这会引起编译错误或数据丢失。
5. 两个都是集合类型，并且它们的基础类型是兼容的。例如以下两种类型兼容：

   ```
   var
     s1:set of byte;
     s2:set of byte;
   ```
6. A是字符串类型，B是字符串、packed-string 和Char 类型。
7. A是Variant 类型，B是整数、实数、字符串、字符或布尔类型。反之亦可。
8. 两个都是类、类引用或接口类型，并且B继承于（继承）A。

9. A是PChar 或PWideChar，B是0下标开始的字符数组（array[0..n] of Char）
10. A是Pointer（无类型指针）类型，B是任意指针类型。
11. 两个是同一种类型的指针，并且开启了编译器指示字{$T+}
12. 两个都是过程类型，它们有相同的返回类型，并且参数的个数、位置和类型都相同。

赋值兼容性

赋值兼容性不是一种对称关系。T1 是一个变量，T2 是一个表达式，若 T2 的值在 T1 的取值范围内，并且至少下面一个条件成立，则 T2 可以赋给 T1：

1. T1和T2是同一种类型，并且不是文件类型或包含文件类型的结构类型
2. T1和T2是兼容的有序类型
3. T1和T2都是实数类型
4. T1是实数类型，T2是整数类型
5. T1是PChar类型或任何字符串类型，而T2是字符串常量
6. T1和T2都是字符串类型
7. T1是字符串类型，T2是字符或packed-string类型
8. T1是一个长串类型，T2是一个PChar类型
9. T1和T2是兼容的packed-string类型
10. T1和T2是兼容的集合类型
11. T1和T2是兼容的指针类型
12. T1和T2都是类、类引用或接口类型，并且T2继承自T1
13. T1是一个接口类型，T2是实现T1的一个类
14. T1是PChar或PWideChar，T2是一个0下标开始的字符数组（array[0..n]ofChar）
15. T1和T2是兼容的过程类型（在一些赋值语句中，一个函数或过程的标志符被认为是过程类型）
16. T1是Variant类型，T2是整数、实数、字符串、字符、布尔或接口类型
17. T1是整数、实数、字符串、字符或布尔类型，T2是Variant
18. T1是IUnknown或IDispatch接口类型，T2是Variant类型（若T1是Iunknown，T2的类型编码必须是varEmpty、varUnknown或varDispatch；若T1是Idispatch，T2的类型编码必须是varEmpty或varDispatch。）

2.10 类型转换

所谓类型转换是指在编程时将某种类型转换成另一种类型，如 Integer('A')将字符 A 转换成整数型的值 65。类型转换的语法为：

类型标识符(表达式)

类型名称可以是任意一个已经定义过的类型的名称，这个名称必须是标识符（所以诸如 ^Integer 之类的类型表达式不能用于类型转换）。

根据表达式类型的不同，类型转换可以分为值及变量转换。二者语法完全相同，当转换规则上有一点小小的区别。

2.10.1 值转换

值转换中的表达式是一个直接常量值。如 Integer('A')中的字符'A'或 Char(65)中的整数 65。在默认的情形下，值转换中的类型标识符及表达式均不能是除指针类型及有序类型之外的其它类型。这句话的意思如下：

```
Integer(4.5)    //错误，表达式只能是有序类型或指针类型
Real(12) //错误，类型标识符只能是有序类型或指针类型
```

在某些时候转换后的值可能超出了变量所能表示表示的最大范围，如以下情形：

```
var
  ch:AnsiChar;
  …
  ch :=  AnsiChar(320);
  …
```

AnsiChar 类型的值域中只有 256 个值（最大序数为 255），所以将整数 320 转换成此类型型后，ch 无法容纳这个值，这就相当将一把 2 米长的剑塞进 1 米长的鞘的中，当然无法容纳。在这种情形下，编译器有两种选择，一是在编译期间提示错误，告诉你这把剑不能放进这把鞘中；二是对数据进行截取，直接把剑砍成一米长后再放进鞘中。

一般情形下截取都会将数据直接截成变量的最大值，如上面的 ch 的值为 255。但这里的截取却是将数据进行反绕后再赋给变量。具体的操作如下：

1. 将所赋之值除以变量的值域中值的数量，保留得到的余数
2. 将上一步中的余数与变量值域中的最小值相加，得到的值就是目标值

所以，对于上面的例子中的 ch := AnsiChar(320)，编译器进行如下操作：

1. 320 除以 256，余数为 64。
2. 将 64 与 AnsiChar 的最小值(0)相加，所得值为 64，即为目标值。

所以 ch 的值为 64。

但无论是反绕还是直接截取，均不会改变所赋之值的符号，也就是说，将一个负数赋给一个变量时，无论是截取还是反绕，变量得到的值一定是负数，但两者绝对值不一定相同。

2.10.2 变量转换

所谓变量转换是指将一个包括变量和常量在内的标识符而非一个值转换成其它类型。变量转换的第一规则：对于所占用的内存，目标类型永远不要小于源类型。若目标类型所占字节数大于源类型，编译器会进行反绕或截断。如下面的例子所示：

```
var
  ch:AnsiChar;
  i:Integer;
  b:Byte;
begin
  ch :=  AnsiChar(321);
  i :=  320;
  b :=  Byte(i);
  writeln(integer(ch));
```

```
  writeln(b);
  readln;
end.
```

对于实数的类型转换较为特殊。所有类型的实数在转换至其它类型的值时，首先被转换为实数中的 Extended 类型，然后再转换成其它类型。而 Extended 类型过大，导致大多数变量根本无法容纳这种类型的值，所以将实数转换成其它类型时应当使用 Delphi 提供的例程。常见的例程有 Int、Round、Trunc，它们均定义于 system 单元中。Int 以实数的形式返回一个实数的整数部分，Round 返回一个与指定的实数最为接近的整数，Trunc 以整数形式返回一个实数的整数部分。

```
var
  r:real;
begin
  r :=  3.84;
  writeln(Int(r));
  writeln(Round(r));
  writeln(Trunc(r));
  readln;
end.
```

无论是值转换还是变量转换，这些转换表达式均不能放在赋值语句的左边。也就是说代码中不能出现下列形式的语句：

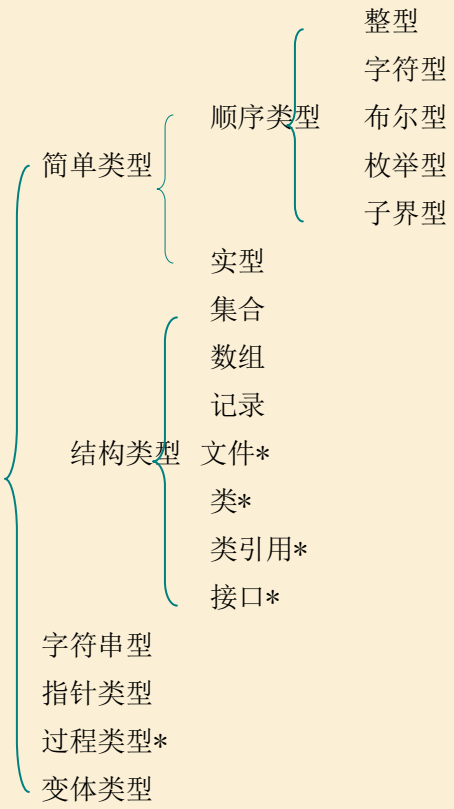
```
var
  ch:AnsiChar;
  …
  Integer(ch) :=  65;
  …
```

这点与 Delphi 的说明文档正好相反，可能在某情形下这种语句是合法的，但至少在 Delphi2010 的默认情形下不能这么做。

第三章 数据类型及运算符

Pascal 相比其它语言的突出优点是数据类型丰富，语法简单直观。由 Pascal 演化而来的 Delphi 语言继承了这一特点。具体说来，Delphi 中的数据类型包括整型、实型、字符型、字符串、布尔型、枚举型、子界型、数组、集合、指针、文件等。这些数据类型可分为基本类型与一般类型。基本类型包括大部分内置类型，其内部存储结构及范围不变在所平台上均保持一致；一般类型经过特殊优化，使用时可获得较好性能，但其缺点是在不同平台上其实现可能有差异，Delphi 的编译器会自动处理所有数据类型不同环境下数据实现的差异，故而读者大可不必考虑其间的不同。

较为系统的分法是将所有类型分为简单类型、字符串类型、结构类型、指针类型、过程类型及变体类型六类。如下：



上图中，类、类引用及接口属于面向对象部分，本章暂时不会接触。过程类型将在“函数与过程”章节中讲述。文件类型将于“文件输入及输出”相关内容中论述。

3.1 简单类型

简单类型共包括有六种类型：整型、字符型、布尔型、枚举类型、子界类型、实型。

除实型之外，其它五种类中每一种类型的值域(所有合法的值的集合)中值的数目有限且其中的值排列有序，其每个值在值域中均有一个序数 n，其前后两个值的序数分别为 n-1 和 n+1。这样的数据类型称之为顺序类型(ordinal type)，也有些书上翻译成有序类型。

3.1.1 整型类型

整型类型是由数学中的所有整数(包括正整数、负整数和 0)所组成的集合的一个子集。如其中的 byte 类型表示从 0 至 255 共计 256 个整数；shortint 表示-128 至 127 共 256 个整数。

Delphi2010 中的整型类型包括 Integer、Cardinal、Shortint、Smallint、Longint、Int64、Byte、 Word、Longword 以及 UInt64 共 10 种具体类型。其中最为常用的是 Integer 和 Cardinal，此二者为一般整型类型，分别等同于 longint 及 longword 类型，其性能经过特别优化处理，在多数环境下能获得较好的表现。另外六种为基本整型类型，一般不常用。下表给出上面 10 种类型各自的取值范围：

类型名称	取值范围	存储格式
Integer	-2147483648..2147483647	signed 32-bit
Cardinal	0..4294967295	unsigned 32-bit
Shortint	-128..127	signed 8-bit
Smallint	-32768..32767	signed 16-bit
Longint	-2147483648..2147483647	signed 32-bit
Int64	-2^63..2^63-1	signed 64-bit
Byte	0..255	unsigned 8-bit
Word	0..65535	unsigned 16-bit
Longword	0..4294967295	unsigned 32-bit
UInt64	0..2^64-1	unsigned 64-bit

[注]：上表中，signed 表示有符号，表明此类型可取正数也可取负数。Unsigned 表示无符号，表示此类型只能取正数和 0。如 shortint 存储格式为 signed 8-bit，表示此类型为 8 位有符号数，其在内在中占据个字节，所能表示的最大值为 =256，但此类型为有符号格式，故其取值为-128 至 127(包括 0)。

3.1.2 实型类型

类似于整型，实型为所有实数所组成的集合的子集。其各种具体类型及取值范围如下表：

类型名称	取值范围	有效位数	所占字节数
Real48	2.9 x 10^-39 .. 1.7 x 10^38	11-12	6
Single	1.5 x 10^-45 .. 3.4 x 10^38	7-8	4

Double	5.0 x 10^-324 .. 1.7 x 10^308	15-16	8
Extended	3.6 x 10^-4951 .. 1.1 x 10^4932	10-20	10
Comp	-2^63+1 .. 2^63-1	10-20	8
Currency	-922337203685477.5808 ... 922337203685477.5807	10-20	8
Real	-5.0 x 10^-324 ... 1.7 x 10^308	15-16	8

[注]：在早期的 Delphi 某些版本中，Real 等同于上表中的 Real48 类型。为了保持兼容性，用 Delphi2010 编译相应早期版本所编写的代码时，须手动改正，或者在 Delphi2010 中使用{$REALCOMPATIBILITY ON}编译指令使 Real 等同于 Real48；

类似于整型，实型亦有一般类型与基本类型之分，上表中的 Real 为一般类型，其余均为基本类型。但 Double 类型在实现上完全等同于 Real 类型。多数情况下，使用 Real 可获得最好的性能。

关于基本类型，有以下几点须注意：

1. Real48 是为兼容以前的代码而保留，因其不能于 Intel CPU 平台下优化，故运行稍慢。不推荐使用。
2. Extended 精度较高，但编写跨平台共享文件时须小心使用。
3. Comp 为 Intel CPU 的原生 64 位整数，之所以被归类为实型，是因为相比整型言，其内部实现类似于实数，例如此类型不能进行递增或递减运算。使用此类型时建议以 Int64 代替提高性能。
4. 任何计算结果若是 Currency 类型，不管这个结果原本有多少位小数，都将自动保留四位小数。

3.1.3 字符类型

字符类型用于描述一个单独的书面的文字和符号。Delphi 支持 AnsiChar 与 WideChar 两种基本字符类型。AnsiChar 类型变量使用单字节来表示一个字符，WideChar 则使用两个字节来表示一个字符。WideChar 与 AnsiChar 类型的变量之间不能相互赋值：

```
var
  wch:WideChar;
  ach:AnsiChar;
begin
  ach :=  '国';  //错误。ach 占用一个字节而汉字占两个字节
  ach :=  wch;  //错误
end.
```

Delphi 中最常用的字符类型是 Char 类型。它是上面两种字符类型其中一种的别名。在 Delphi2010 中 Char 被默认为 WideChar，但在之前的版本如 Delphi7 中，此类型代表 AnsiChar。

Delphi 中的字符与字符串没有明显区别，编译器将所有的只含有一个字符（对于 AnsiChar 则是一个字节）的字符串如'A'、'B'当成一个字符。

3.1.4 布尔类型

布尔型又称之为逻辑型，用于判断真假，其值只有 true 和 false 两种类型。Delphi 中有四种内置布尔类型：Boolean，ByteBool，WordBool，LongBool。

Boolean 是 Delphi 中最为常用的布尔类型，此类型变量在内存中占据 1 个字节，其值域中只有 true 和 false 两个枚举常量值。其中 true 的序数值为 1，false 的序数值为 0。

将其它类型的值赋给布尔型变量时，必须显式进行类型转化，非 0 值将被转化为 true，而 0 被转化为 false。例如：

```
begin
  if boolean(0) then
    writeln('False'); //不显示

  if boolean(3) then
    writeln('True'); //显示 True
end.
```

ByteBool，WordBool，LongBool 在内存中分别占据 1，2，4 个字节。这三种类型中，false 的序数为 0，但与 boolean 不同，true 的序数并非为 1，包括正数和负数在内的所有非 0 的值均被视为 true。

3.1.5 枚举类型

枚举类型表示一个有次序且数量有限的值的集合，其中的值用标识符表示。

1. 枚举类型的声明

枚举类型的声明格式如下：

Type

类型名＝(标识符 1，标识符 2...标识符 n)；

Type 为 Delphi 的保留字，用于声明一个新的数据类型。其后的(标识符，标识符 2...标识符 n)称为枚举类型的值域，表示此类型变量可取值的集合。

枚举类型中的每个标识符均具有一定的序数，在未替任何标识符指定序数的情形下，编译器将按照先后顺序从 0 开始依次给每个成员分配序数。如三原色用枚举表示为：

Type

BasicColor = (red，green，blue)；

此语句声明了一个枚举类型 BasicColor，其包含三个成员 red、green 及 blue，三者序数分别为 0，1，2，此类型的变量只能取这三个值的其中之一。

枚举类型允许使用成员的序数表示成员，其形式为：

enum(n)

上式中，enum 表示一个枚举类型名称，n 表示任一成员的序数，enum(n)表示枚举类型 enum 中序数为 n 的成员。如 BasicColor(0)、BasicColor(1)、BasicColor(2)分别表示 red、green、blue。

用户在声明枚举类型时可以自己指定枚举成员的序数，对于未指定序数的成员，默认其序数为前一个成员的序数加上 1。若第一个成员未指定序数，将会默认为 0。据此，如下

声明：

```
Type
    Myenum4 = (i1,i2,i3 = 4,i4,i5 = 8);
```

其中 i1 至 i5 的序数分别为：0，1，4，5，8。从表面看来，myenum4 有五个成员，分别是：i1,i2,i3,i4,i5。但事实情况是，Myenum4 共包括 9 个而非 5 个值。

造成上述情形是因为枚举类型的成员个数不是由此枚举类型中的标识符的数量决定，而由其标识符中最大序数与最小序数决定。如：Myenum4 中最大与最小序数分别为 8、0，故其有 9 个成员，只不过其它 4 个成员并未在声明时用标识符表示。使用未使用标识符标识的成员时，可利用序数索引表示，如 Myenum(6)表示其第 7 个成员，Myenum(4)表示其第 5 个成员。(注意序数是从 0 开始)

枚举类型的每个成员均为一个直接常量，就像英文字母 ABC 一样，它们不代表其它任何值，因其本身正是一个确定值。

在 Delphi 中枚举类型的每个成员的标识符均被理解成是一个符号常量。例如：

```
Type
    Myenum = (i1, i2, i3);
```

这个声明相当于定义了三个符号常量：

```
Const
  i1 = 0;
  i1 = 1;
  i2 = 2;
```

若再次将这些标识符声明用于其它声明，编译器会发现同一个标识符被用于了多个声明语句之中，会提示错误。如：

```
Var
    i1:string;
```

标识符 i1 既在 myenum 中被声明为符号常量，又于其后被声明为一个 string 类型的变量，所以此声明不能通过编译，编译器会提示错误：标识符 i1 不能被重新使用。

2. 枚举类型变量的声明

枚举类型的变量可以使用的枚举类型名称来声明，也可以直接使用枚举类型的值域声明。下面两个变量的声明均属合法：

```
Type
    Myenum = (i1, i2, i3);
Var
    v1:myenum;
    v2:(a,b,c);
```

当利用了某一枚举类型的名称声明一个变量后，在此变量有效的范围之内不能使用其枚举类型的本体声明其它变量。如上面的声明中，若 v2 的声明为

```
V2:(i1,i2,i3);
```

编译器将会提示错误：标识符被重新定义。因为枚举类型可以有未指定名称的成员，同理，亦可以存在未指定名称的枚举类型，上面 v2 的类型(i1,i2,i3)就是一个未指定名称的类型，而其中的标识符与 Myenum 中的标识符完全一样，这相当于同时定义了两种标识符相同的枚举类型，故而引起了名称的冲突。

3.1.6 子界类型

所谓子界，子为子集，界为界限。子界类型就是一种利用上界与下界在某种其它有序类型的值域中划定一个范围作为自己的值域的类型。例如 byte 可视作编译器内置的子界类型，它在 integer 类型的值域中插入了 0 与 255 两个边界，则其值域为 0 至 255 共计 256 个值。

1. 子界类型的声明

同枚举一样，子界类型的声明以 type 保留字作为开始：

```
Type
  子界类型名＝下界..上界
```

其中的上界与下界必须为其它有序类型中的直接常量，如整型数字、枚举成员等。上界及下界所属的类型称之为此子界类型的基类型。注意上界的序数不能小于下界的序数。

例如：若有枚举类型 myenum = (i1，i2，i3，i4，i5)；可以定义子界类型：

```
Type
  Mysub = i3..i5;
```

则此类型的取值值域为 i3，i4，i5，基类型为 myenum。

类似于枚举类型，子界类型也可以通过其上下界直接声明变量，如：

```
var
  Mycap:'a'..'z';
```

Mycap 表示一个子界类型的变量，其域为从'a'至'z'的 26 个小写英文字母，基类型为字符型。

[注]：在用 type 声明数据类型时，若‘＝’后第一个符号为‘（’，编译器将自动将此声明当成是枚举类型的声明。据此，若声明如下的子界类型：

```
Type
    Mysub = (2+3)*2..(6+4)*4;
```

由于‘＝’后第一个符号为‘（’，编译器会认为这是在声明一个枚举类型，故而给出一个错误。

2. 子界类型变量

声明子界类型变量有两种途径：其一是用类型名称来声明。其二是用下界..上界的形式直接声明。下面的例子中，v1 及 v2 的声明均为合法声明：

```
Type
  Myenum = 0..10;
Var
  V1:myenum;
  V2:1..10;
```

与枚举类型不同，V1 与 V2 并不会引起编译报错，可以认为 V1 与 V2 等效。因为子界类型只是在已有数据类型中截取一部分值域，并没有声明除类型名称以外的任何标识符，故而不存在标识符冲突的问题。

子界类型仅是在基类型的值域中加上上下界从而限定了子界类型的变量在基类型的值域中可取的值的范围而已，但其本质与基类型完全相同。

读者可如此理解：一滴水与一碗水之间除了份量有区别之外，其它完全一致。

替子界类型的变量赋值时，若所赋之值不在子界类型的值域内，则会提示错误，如对于上面声明的变量 v1，下面的赋值在编译器的默认设置下将不会通过编译：

```
V1 :=  15;
```

3.2 结构类型

3.2.1 集合及其运算

1. 集合类型的声明

集合类型表示某个有序类型的若干个值的集合。可以将集合类型理解为一个用于容纳数据的容器，只是其容纳的数据必须为有序类型。声明一个集合类型的方式如下：

```
Type
    Myset = set of BaseType;
```

Myset 为所声明的集合类型的名称。BaseType 为集合中所含成员的类型，称之为基类型。Delphi 规定：

- 一个集合类型的所有成员必须是同一有序类型且最多只能含 256 个成员；
- 集合类型成员具有唯一性，即同一集合类型中不允许含有相同的成员；
- 集合类型成员具有无序性，其中的成员没序数。无法像枚举型一样用序数表示集合成员；
- 集合类型的值域由 BaseType 的值域决定。

以下给出一些实例：

```
Type
    Set1 = set of byte;     //set1 的成员为 byte 型，值域为 0 至 255 共 255 个整型值
    Set2 = set of 1..9;     //set2 的成员为子界类型，值域为 1 至 9 共 9 个整型值
    Set3 = set of (red,green,blue);  //set3 的成员类型为枚举类型，值域为 3 个标识符
```

2. 集合类型变量的声明与赋值

声明集合类型的变量有两种方式：可使用集合类型的名称来声明，也直接利用集合类型的本体来声明：

```
Var
    Vset1:set1;
    Vset2:set of byte;
```

给集合类型的变量赋值时，所赋之值应使用方括号括起，括号内的成员以逗号隔开。如：

```
Vset1 = [1,3,5];
```
集合变量 Vset1 中所含的成员有三个，分别为：1、3、5。

3.2.2 数组

数组是由若干个同类型且具有序号的变量(即数组成员)所组成的队列。与普通变量不同，数组成员没有变量名，只能通过数组名和序号组成的索引来存取。如 A[2]、C[4]等。

在分配内存时，数组的所有成员将被安排在一段连续的区域中。

Delphi 的数组可分为静态数组和动态数组两种。

1. 静态数组

静态数组是在程序初始化时就被分配内存的数组。鉴于此，静态数组在定义之后其大小不能更改。定义一个 N 维静态数组的方法如下：

```
Type
  TypeName = Array [IndexTypeType1,IndexType2...IndexTypeN] of BaseType;
```
或
```
  TypeName = Array [IndexType1] of  ...Array [IndexTypeN] of BaseType;
```
其中 TypeName 代表所定义的数组类型名称。IndexType 代表数组各成员的标号类型。BaseType 代表数组成员的类型。事实上，虽然 Delphi 允许定义多维数组，但实际应用中一般只使用一维或二维数组。若定义一维的数组，声明语句则可简化成：

```
Type
  typeName = Array [IndexType] of BaseType;
```
如定义一个一维数组类型：
```
Type
  MyArray = Array [5..9] of integer;
```
MyArray 类型的数组变量中含有 5 个成员，序号分别为 5、6、7、8、9。

定义一个多维数组类型：
```
Type
  MultiArray = Array [1..3] of Array [8..9] of integer;
```
或
```
  MultiArray = Array [1..3, 8..9] of integer;
```
MultiArray 类型的数组变量中含有 3 个成员，其中每成员又是一个含 2 个成员的数组。这有点类似于教室中的座位，全班共分为若干组，每一组由有若干列组成。请读者思考下面的数组 MultiArrayEx 的构成：

```
Type
  SubArray = Array [8..9] of integer;
  MultiArrayEx = Array [1..3] of SubArray;
```

数组类型的变量可使用数组类型的名称来声明，也可直接使用数组类型的本体来声明。如：

```
Var
  A1:MyArray;
  A2:Array [5..9] of integer;
```

数组变量赋值时只能按其成员逐个逐个赋值。Delphi 在处理数组变量时，使用数组类型的名称来判断变量的类型。换而言之，若有数组 A1、A2 声明如下：

```
Type
  A1:Array [1..5] of integer;
  A2:Array [1..5] of integer;
```

编译器会将 A1、A2 当成两个不同的类型，因为 A1 与 A2 所属的数组类型均无名称。只有两个数组类型名称一致时编译器才会两个变量当成同一类型的变量。可将上面的声明语句改成如下：

```
Type
  A1,A2:Array [1..5] of integer;
```

2. 动态数组

与静态数组不同，动态数组在声明时只需声明成员的类型，并不需要指定成员的序号。也就是说，动态数组并未在声明时指定成员的数量，其成员数量可以在运行期间动态改变。

一个 N 维动态数组声明如下：

```
Type
    TypeName = Array of Array of ... BaseType;
```

与静态数组声明不同，声明动态数组不需要指定成员序号。根据上式，声明一个一维动态数组变量 dA1 的格式为：

```
Type
    dMyArray = array of char;
Var
    dA1:dMyArray;
```

当然也可以直接写成：

```
Var
    dA1:array of char;
```

不同于静态数组变量，动态数组变量在赋值前必须设置大小。此过程可使用函数 SetLength 完成，其使用方式为：

```
SetLength(A,n1,n2,...);
```

A 表示任一数组变量，n1 表示 A 的第一维的长度，n2 表示 A 的第二维的长度...依此类推。Delphi 并未要求一次性设置所有维的长度。故可先设置第一维的长度，必要时再去设置第二维的长度。注意：对于多维数组，只有第 N 维的长度确定后才能指定第 N+1 维的长度。

对于以下声明：

```
Var
    dA:Array of Array of integer;
```

此语句中 dA 为二维数组变量，调用 SerLength:

```
SetLength(dA1,4,2);
```

则 dA 共含 8 个成员：dA1[0][0]、dA1[0][1]、dA1[1][0]、dA1[1][1]、dA1[2][0]、dA1[2][1]、dA1[3][0]、dA1[3][1]。上述表示法也可简写成 dA1[0,0]、dA1[1,0]...

Delphi 提供 3 个标准函数：High()、Low()、Length()用于数组变量。其中 High()和

Low()用于返回数组成员的序号的最大值与最小值，而 Length()用于返回数组的成员数量即数组的长度。值得注意的是：动态数组只能设置数组的长度，不能设置成员的序号。而 Delphi 对于未指定成员序号的数组默认其序号从 0 开始。故而动态数组的第一个成员的标号为 0，其长度比其最大序号要大 1。

3.2.3 记录类型

Delphi 的新版本将 Record 类型作了扩展，但这种扩展涉及面向对象部分。为避免麻烦，此节我们只讨论未经扩展之前的 Record 类型，新版的 Record 将置于后续章节。从这个角度来说，本节的标题改为“传统记录类型”似乎更恰当一些。

闲话少说，我们开始。

1. 记录类型的定义

假设我们编写程序时需要使用某班同学的个人信息如身高、体重等，我们刻怎么表示这些数据？最简单的想法就是针对每个学生的每条信息设置一个变量。如 StdA_Height、StdB_Height 分别表示 A 同学、B 同学的身高等，但这样一来，若每个学生有 5 条信息，全班共有 60 个学生，我们就得设置 300 个变量，不用说，这是很麻烦的事。

如果有这样一种变量，其中含有 5 个成员，每个成员均用于表示每个学生的一条信息。这样一来我们只需要设置 60 个变量。显然这要轻松的多。

记录类型的变量符合上述要求。我们可以定义一个记录变量其中含有 height、high、ID、name、StdClass 共 5 个成员分别表示学生的体重、身高、学号、姓名、班级。获取 A 同学的信息时，A.height 表示其身高，A.name 表示其姓名，如此类推。可以看出此方式比起设置 300 个变量要好得多，最起码代码量要少得多。

一个记录类型声明如下：

```
Type
    TRec = record
      Member1:type1;
      Member2:type2;
      ...
      memberN:typeN;
    End;
```

此语句声明了一种记录类型，其中含有 N 个成员，member1 的数据类型是 type1，member2 的数据类型是 type2,依次类推。

- 成员类型可以是任何类型。
- 声明中的每一行用分号隔开，但 typeN 后的分号可以不写。

根据上述声明方式，本节前面所述的用于统计学生信息的记录类型可声明为：

```
Type
    Std = record
      Name:string;
      ID:string
      Height:integer;
      High:integer;
```

```
        StdClass:integer
      End;
```

数据类型相同的成员可以写在同一行，上式可简写成：

```
    Type
      Std = record
        Name,ID:string;
        Height,High,StdClass:integer;
      End;
```

2. 记录变量定义及赋值

记录类型变量的定义有两种方式：一是使用记录类型的名称定义；二是使用一个记录直接定义。不能将其中一种方式定义的变量直接赋给另一种方式定义的变量。无论记录变量是用何种方式声明，但对于记录成员而言，只要赋值兼容即可相互赋值。

```
program RecordRExample;
{$APPTYPE CONSOLE}
type
  TStd = record   //声明记录类型 TStd，用于描述每个学生的姓名和班级信息
    Name:string;
    Grade:integer;
  end;
var    //声明 A、B、C 三个变量
  A,C:TStd;
  B:record
      Name:string;
      Grade:integer;
    end;
begin
  B.Name := 'BName';   //将 B 的姓名设为 BName
  A := TStd(B);  //不能将 B 直接赋给 A，须经类型转化
  C.Name := B.Name;   //记录成员之间只要赋值兼容即可自由赋值
  writeln(A.Name);  //显示 A 的姓名
  writeln(C.Name);
  readln;  //使屏幕暂停滚动，按下任意键即可结束
end.
```

若要存取记录变量的成员，可使用 TypeName.Member 的形式来存取变量的成员。如 C.Name 表示 C 的姓名，则 C.Name := 'Bname'表示将 C 的姓名设为 Bname。

记录类型变量的赋值有两种方式：

一是利用另一个记录变量直接赋值。如上例中，对于 C 的赋值也可以直接定成：

```
C: =A;
```

编译器会自动将 A 中的所有成员的值逐一赋给 C 中的相应成员。若记录变量中有以引用类型的成员，则情况较为复杂，关于这点在后续章节中会有具体论述。

另一种方式是对记录变量的成员手动逐个赋值：

```
C.Name := 'Bname';
```

```
    C.Grade :=  2;
```
若记录变量中成员较多，可使用 with 语句简化赋值所需输入的代码：
```
    With C do
    begin
      Name :=  'Bname';
      Grade :=  2;
    end;
```

3. 变体记录

假如你是公司职员，现在要编写一个程序统计公司员工的收入情况。你可以定义一个记录类型来储存每个员工的薪水：

```
TSalary = record
  name:string;
  MonthWage:Currency;
end;
```

其中的成员用于记录员工的姓名及月薪。

现在有个问题，你可能是按月拿那一点可怜的月薪，但公司中某些领导干部却是拿年薪的。这样一来，用这个记录类型怎么来记录这些人的薪水呢？当然我们不允许定义另外一个记录。（其它的种种复杂情形也请读者吞到肚子里，这只是一个例子，经不起那么多的可能性来折腾。）

你可以向其中再添加一个 YearWage 用于记录员工的年薪，但这样一来，每个记录中都会有一个空闲的成员变量，是否有点浪费呢？

不知读者是否还记得前面介绍的共址变量，它可以很方便的解决这个问题。我们可以将 TSalary 声明成如下形式：

```
TSalary = record
  name:string;
  MonthWage:Currency;
  YearWage:Currency absolute MonthWage;
end;
```

这样，我们不管使用 MonthWage 还是 YearWage 都不会造成浪费，因为二者共用同一块内存。这就像两个轮流工作的人使用同一台计算机一样，虽然任何时候都有一个人在休息，但计算机却不会空闲而造成浪费。

但是不幸的很，记录成员根本不能声明共址变量。简单来讲，上面的声明无法编译。所以这种想法也只能束之高阁。

既然共址变量不行，我们就需要找一个与之类似的方法。变体记录就是这样的解决方法。

变体记录的实质是在记录中声明若干个共用同一块内存的共址变量（只共用开始的一部分内存，你不能指望一个 8 字节的字段与一个 2 字节的字段完全共用同一区域，但它们可以共用开始的 2 个字节）。但不同于普通的共址变量，变体记录有着完全不同的声明方式：

```
type
  记录名＝record
    字段 1：类型 1;
```

```
    字段 2：类型 2;
    ...
    字段 n：类型 n;
  case [tag：] 有序类型 of
    常量 1：（字段声明）;
    常量 2：（字段声明）;
    ...
    常量 n：（字段声明）[;]
  end;
```

其中的 case 之前的部分为普通记录声明。自 case 开始到记录声明的末尾均是变体记录的专有部分。

首先，这部分声明必须置于所有普通成员之后。

其次，其中所有以方括号“[]”围起的内容均可省略。

第三，其中的所有常量必须是指定的有序类型。

第四，其中的每个常量可包括多个常量值，中间以逗号隔开。

第五：括号中的字段声明也可包括多个字段的声明，中间以分号隔开。其中的字段不能被声明成长字符串、动态数组、变体类型、接口以及包含这些类型的其它结构类型如记录、数组等。

第六：括号中的最后一个字段可不接分号。

第七：声明中的 tag 以及常量对于用户来说，根本没有任何用，我们只需保证其规范性即可。

最后：虽然变体记录声明时使用了 case 关键词，但读者千万不要去想记录中的变体部分与真正的 case 语句是否有什么关联。很确切的讲，至少在用户角度，没有任何关联，只是用了同一个名字而已。使用变体记录时完全可将变体记录当成普通记录，编译器会处理所有的细节。

下面声明一个变体记录的声明：

```
type
  TRec = record
    s:string;
  case Integer of
    1:(f1:integer;
       f2:String[4]);
    2,6,8:(f3:string[8])
  end;
```

笔者当初曾经很好奇，如果将 TRec 中的 f1、f2、f3 全部赋值，会有怎样的结果？请读者也思考一下，对于如下示例，屏幕上会显示什么呢？

```
var
  Rec:TRec;
begin
  rec.s := '5';
  rec.f1 := 4;
  rec.f2 := 'ABCD';
  rec.f3 := 'Delphi32';
```

```
  writeln(rec.f2);
  readln;
end.
```

我们来分析一下，TRec 声明相当于声明了一个成员为 f1 和 f2 的记录类型的字段，然后再让这个字段与 f3 共用一段内存，且二者占用的内存大小相等，所以它们完全地共用相同的内存。明白这点，我们接着分析。

再替 f1 与 f2 赋值后，这段共用内存中 8 个字节的值已经确定为：0004ABCD，但 f3 被赋值后，这 8 个字节的值又被恰好全部被 f3 的值所取代而变成：Delphi32，再然后，我们在屏幕上显示 f2 的值时，它只能从这条字符串中选择属于自己的值来输出。由于 Integer 在内存中也占用 4 个字节，所以 f2 的值只能是共用内存的后 4 个字节，故而应该是显示字符串 hi32。事实也正是如此。

读者可以再试一下将 TRec 中的 f1 与 f2 的声明交换一下顺序，或是将代码中 f2 与 f3 的赋值语句交换顺序，看看这两种情形下屏幕上会显示什么。

3.3 字符串类型

字符串表示由字符所组成的序列，简单来说，若干个字符连在一起就可视为一个字符串。Delphi 中常用的字符串类型主要有四种：Shortstring、Ansistring、Widestring、Shortstring。三种字符串的简要信息如下：

类型	最大长度	所需内存	用途
ShortString	255 byte	2-256 bytes	容纳由 AnsiChar 组成的字符串
AnsiString	2 GB	4 bytes-2GB	容纳由 AnsiChar 组成的字符串
WideString	2 GB	4 bytes-2GB	容纳由 WideChar 组成的字符串
UnicodeString	2 GB	4 bytes-2GB	容纳 Unicode 编码的字符串

上表中，UnicodeString 与 WideString 基本无区别。

Delphi 编程中通常将字符串变量声明为 String 类型，此类型与 UnicodeString 类型完全等价，在较早的某些版本中 string 等价于 AnsiString 类型。

使用字符串时我们只需将其当成一个普通的简单变量，需要做的只是给其赋值并读取值。所有的细节均由 Delphi 自动进行管理。下面是一个简单的示例，在其中我们声明了字符串变量 STR，并给它赋值为‘Delphi2010’，然后在屏幕上显示。

```
var
  str:string;
begin
  str :=  'delphi2010';
  writeln(str);
end.
```

Delphi2010 支持多种编码类型的字符串，使得字符串的应用非常方便，但这也直接导致很多初学者甚至是有一定经验的 Delphi 用户迷惑不解。为了避免学习上的困难，此节我们只讨论了常用字符串类型的基本知识以使读者能够毫无困难地学习接下来的内容。在本

书的附录中我们详细介绍了各种字符串，若读者有需要可参考。

3.4 指针

指针类型是一类特殊的数据类型，此类型的变量专用于存储其它变量的地址(包括其它指针的地址）。通常将指针类型的变量简称为指针。

举个例子：“夏威夷”相当于一个变量的名字，而“西经 157°、北纬 21°”则是这个名字所代表的具体地址。指针就是一个专门用于表示某个变量在计算机中的具体地址的数据类型。当一个指针变量含有某个变量的地址时，我们称这个指针变量指向了这个变量。

看到这里读者不明白：既然可以使用变量的名称，为什么还要用指针呢？原因有两个：

- 某些情形下指针比变量名更方便，也更快捷
- 在调用其它语言编写的代码时可能需要指针

1. 指针变量的声明

声明一个指针变量与声明一个其它变量并无明显区别，基本格式为：

```
Var
  变量名：指针类型
```

其中的指针类型可以使用符号“^”加上一个合法的数据类型的名称表示。如声明一个字符串指针变量：

```
Var
  AnsiStr:^AnsiString;
```

也可以先声明指针类型，再声明指针变量：

```
Type
  PAnsiStr = ^AnsiString;
Var
  AnsiStr:PAnsiStr;
```

这两种声明方法完全等同，只不过第一种更为简洁而已。

看到这里，读者可能发现了一个问题：指针还分为字符串类型、整型类型等不同类型的指针？答案是 Yes。在未进行类型转换的情形下，Delphi 中的大部分指针类型必须和其所指的变量的类型一致。如字符串变量只能用字符串类型的指针表示，而记录指针只能指向记录类型的变量等。（只有极少数的指针才可指向多种类型的变量，这种指针称为通用指针）

2. 指针的赋值

可以使用两种方式给一个指针变量赋值。其一是将一个指针的值直接赋给另一个指针，赋值后二者将指向同一个变量；其二是接将某个变量的地址赋予一个指针变量。

Delphi 使用符号“@”加上变量名表示变量的地址。如@Var 表示变量 Var 的地址。符号“@”称为取地址符，表示其专用于获得变量地址。类似地，符号“^”也被称之解地址符。一个指针变量名后跟“^”时表示此指针所指变量的值。如

```
var
  P:^integer;
  V:integer;
begin
  V :=  89;
  P :=  @v;
end;
```

P 指向变量 V，P^表示 V 的变量值。

Delphi 中的取地址符“@”也可使用标准例程 Addr()代替。如@V 等效于 Addr(V)。唯一的区别在于 Addr() 不受编译指令$T 的影响。

注：当编译指令$T 处于$T-状态时，@所返回的指针为无类型指针，与其它所有类型的指针保持兼容。当处于$T+状态时，@返回与变量同类型的指针。默认状态为$T+。

3. 无类型指针

在进入本节正题前，我们先来看一下指针本身的结构。

以 sizeof 可知任何指针变量在内存中均占用 4 个字节，这 4 个字节分成两部分：一部分用于存储指针所指向的地址值，另一部分用于标识其指向的数据的具体类型，为方便起见我们将此部分称为类型码区。当我们运行下面语句时；

```
var
  i:^integer;
begin
  i^ :=  9;
end;
```

计算机实质上进行了两步操作：

1. 判断 i 的类型码是否与所赋之值一致，若不一致则无法通过编译，若一致则执行下一步
2. 将所赋之值赋予指针所指变量

在读取 i 所指向的值时同样要先验证类型码的值，不同的数据类型按照不同的方式读取，如一般情形下字符'A'不会被读取成数值 65。

现在我们进入正题。Delphi 中存在一种特殊的指针类型——Pointer，此类型指针结构中的类型码不为任何值，故也称为无类型指针。与普通指针不一样，在不进行类型转化时，Pointer 只支持两种操作：

- 将另一个指针或地址值赋给 pointer 指针
- 将 pointer 指针赋给另一个指针。

由于类型码为空值，使用时不能直接通过 P^的形式读写 P 所指向的变量的值，必须利用类型转化将 Pointer 指针转化为其它类型的指针（此过程实质上是设置 P 的类型码为某个特定值）：

```
var
  p:pointer;
  n:integer;
begin
  p :=  @n;       //令 P 指向变量 n
  n :=  98;
```

```
  writeln(pinteger(p)^);    //通过类型转化读取 n 的值
  pinteger(p)^  :=  78;     //设置 n 的值为 78
  writeln(pinteger(p)^);
  readln;
end.
```

4. 动态指针

指针有两种：一是静态指针，此类指针在声明时即可确定其所指向的变量需要多少内存]；二是动态指针，它不指向一个变量而指向某一块没有分配名称的内存（可以看成是无名变量），这种类型的指针是典型的“用多少，拿多少”。

编译器不会在声明时给动态指针分配内存，因为它不知道这个动态指针到底需要多少内存。编译器不知道，但我们肯定是知道的，所以动态指针需要用户在代码中手动给它分配内存。所谓善始善终，所有的由我们手动分配出来的内存都必须再由我们手动来进行销毁。

Delphi 提供了若干对用于分配及销毁动态指针的标准例程，其中最常用的有两组：

```
procedure New(var X: Pointer);
procedure Dispose(var P: Pointer);

procedure GetMem(var P: Pointer; Size: Integer);
procedure FreeMem(var P: Pointer);
```

New 及 Dispose 用于替某个指针分配内存，所分内存大小由指针所指变量的类型决定，如 integer 类型的指针将被分配 4 字节内存，Byte 类型的指针则被分配 1 个字节的内存。当这些内存不再被需要时应当使用 Dispose 手动销毁。这种分配方式算得上是典型的 0 或 1 型：要么分配固定大小的内存，要么不分配。　相对的 GetMem 则可以分配任意大小的内存，其中的 Size 参数指针指定了所分配的内存的大小，这块内存也必须使用 FreeMem 销毁。

下面的例子展示了两组标准例程的用法：

```
var
  p1: ^integer;
  p2: PChar;
begin
  New(p1);
  GetMem(p2, 40); //替 P2 分配 40 个字节的内存
  FreeMem(p2);
  Dispose(p1);
  readln;
End.
```

3.5 变体类型

1. 变体类型

在介绍变体类型前，我们先来看一下系统如何识别不同数据类型的数据。以最简单的整数为例，系统在存储整数时会根据不同的数据类型的名称而采用不同的内部结构存储来整数值。不同的数据类型有不同的内部结构，当我们以整数数据类型如 integer 声明一个变量时，系统在存储此变量值时就会以整数的内部结构来存储这个变量值。这个过程类似于我们登录电脑，当我们输入某个用户名及密码时，系统会启用此帐户的各项设置。在给变量赋值时若系统发现值不能被在存入此变量(如大小不符、结构不符等)时将会引发错误，例如若我们将一个带小数点的值赋予整数变量时会无法通过。(有些语言在此种情况会自动将值转换为对应的类型，但 Delphi 不支持这种转换)

变体类型是一种非常特殊的类型，说其特殊是因为它可以容纳多种不同类型的值。可以说变体类型有点类似于能够根据所用钥匙的不同可以自动变换魔法锁，其可根据变量值的类型而自动转换其内部存储结构以容纳所赋之值。将整数赋给它时，它的内部结构转换为整型结构，而将实数赋给它时，其内部则转换实型结构。当然，与其它转换一样，变体类型的这种变换也并非万能，其内部不能容纳以下类型的值：

记录、静态数组、集合、文件、类、类引用、指针

[注：]变体中的字符串另有规定，我们将在本节的末尾讨论。

计算机中，一个变体变量占据 16 个字节，这 16 个字节分为两个部分：变量值及变量值的类型码。

变量值可以是一个普通的变量值，也可以是一个指向变量值的指针。类型码用于标识当前的变量值的数据类型。Delphi 提供两种方式以获取变体变量中数据的实际类型：

- 使用 TVarData 结构，此结构相当于 Record 版本的变体类型：

```
var
  v:variant;
begin
  v :=  'Delphi';
  if TVarData(v).VType = varUString then
    writeln('v 中的实际类型为 UnicodeString');
end.
```

- 将标准函数 VarType 返回值与预定义常量 varTypeMask 进行 and 逻辑运算可返回变体变量值的确切类型，VarType 在接受变体类型的参数 V 并返回 TVarData(V).VType：

```
var
  v:variant;
begin
  v :=  2010;
  if VarType(V) = varDouble then
    writeln('v 中的实际类型为 Double 类型');
end.
```

若读者觉得以上两种方式不够方便，也可以直接获取相应类型所对应的常数值：

```
writeln(VarType(V)) 或 writeln(TVarData(v).VType);
```

它们将返回一个整数形式的类型码。System 单元中声明了每种类型所对应的类型码，使用时可自行查询。

定义后变体变量的初始值为预定义常量 Unassigned。常用的预定义常量 NULL 在变体类型中表示未知的值或由于某种错误而丢失的的值。 默认情形下 NULL 值小于包括 UnAssigned 在内的任何值，但这并非绝对。Delphi 中提供了两个预定义变量：NullEqualityRule 与 NullMagnitudeRule。

NullEqualityRule 的值决定了当 Null 与其它值进行“＝”比较时的行为，而 NullMagnitudeRule 的值则决定了当 Null 与其它值进行大小比较时的行为。

编写代码时可以通过改变二者的值来控制 Null 值的行为。下面展示了二者可取之值及各自所代表的意义：

NullEqualityRule 的值及其含义

常量值	含义
ncrLoose	Null 与其它任何值均不相等。NullEqualityRule 默认为此值。
ncrStrict	Null 与所有值均不相等，即使另一个的值也为 Null
ncrError	参与比较的两个变量值之一或全部为 Null 时会引发运行期错误

NullEqualityRule 的值及其含义

常量值	含义
ncrLoose	Null 小于其它任何值(包括 Null)。NullMagnitudeRule 默认为此值
ncrStrict	参与比较的两个变量值之一或全部为 Null 时，比较结果一定是 False
ncrError	参与大小比较的两个变量值之一或全部为 Null 时会引发运行期错误

2. 变体变量的赋值及使用

变体类型与普通类型间赋值兼容。我们可以直接将一个普通类型的值赋给变体变量，反过来也可。唯一需要注意的是：所谓的普通类型一定是变量可以容纳的类型。若在表达式滥用了变体类型及静态类型值，静态类型值将自动被转换成变体类型。如下面的示例：

```
var
   V1, V2, V3, V4, V5: Variant;
   I: Integer;
   D: Double;
   S: string;
 begin
    V1 :=  1;          { integer 值 }
    V2 :=  1234.5678; { real 值 }
    V3 :=  'Hello world!'; { string 值 }
    V4 :=  '1000';          { string 值 }
```

```
    V5 :=  V1 + V2 + I;   { real 值 2235.5678，I 转换成变体类型}
    I  :=  V1;             { I = 1 (integer 值) }
    D  :=  V2;              { D = 1234.5678 (real 值) }
    S  :=  V3;              { S = 'Hello world!' (string 值) }
    I  :=  V4;              { I = 1000 (integer 值) }
    S  :=  V5;              { S = '2235.5678' (string 值) }
end;
```

若所赋之值超出了变体所能容纳的最大值，编译器自动反绕；而对于变体变量的不正确操作如赋值、转型等均会引起编译错误。

操作符方面，除^、is、in 之外的所有操作符均接受变体类型的运算数。除比较运算总是返回一个逻辑型的值外，其它对变体变量执行的操作均将返回一个变体类型的值。这条规则有个例外：当赋值符号的右边存在 Null 时左边的变量值一定是 Null。例如下面的语句中 V 的值将为 Null：

```
var
  v:variant;
  ...
  V :=  Null + 3;
  ...
```

这段代码有一点需要注意：Delphi 中并不存在 Null，代码中的 Null 实为定义于 Variants 单元中的标准函数，其声明为：

```
function Null: Variant;
```

在程序的 uses 从名中添加 Variants 单元后，读者完全可以将 Null 当成一个预定义常量来使用。

3. 变体类型转换

变体变量的类型转换可分为两种情况：一是将变体变量转换为其它类型的变量；二是将变体变量的值转换为其它类型的值，如将变体变量值的类型由 varInteger 转换成 varSingle。

前者的操作形式与普通变量相同，均是以类型名进行强制转化，如 i := integer(v) 可将变体变量 v 转换成整数类型的变量并赋予整型变量 i。但转换的规则与普通变量间的转换稍有差异，详见附录 B。将接口作为值赋给变体变量后，试图对这个变体变量进行任何转换将导致编译器读取接口的默认属性值并将此值进行转换。若接口中未声明默认属性则会引发一个异常。

对于后者，必须使用 Delphi 提供的预定义例程 VarAsType 与 VarCast 进行转换。函数 VarAsType 声明于 Variants 单元，其声明式如下：

```
function VarAsType(const V: Variant; AVarType: TVarType): Variant;
```

此例程将变体变量 V 的值转换为 AVarType 指定的类型。

VarCast 的声明为如下：

```
procedure VarCast(var Dest: Variant; Source: Variant; VarType: Integer);
```

此例程也可将 Source 所代表的变体变量的值转换成所需类型并存储于 Dest 指定的变体变

量之中，varType 为预定义变量，与 VarAsType 中的 AvarType 类似。

下面是示例：

```
uses
  SysUtils, Variants;
var
  v1,v2:variant;
begin
  v1 :=  195;
  writeln(TVarData(v1).VType);  //显示 17，表示 V1 的值为 Byte 类型

  writeln(TVarData(v2).VType);  //显示 0，表示 V2 的值为 Unassigned
  v2 :=  VarAsType(v1,varInteger);
  writeln(TVarData(v2).VType); //显示 3，表示 V2 的值为 Integer 类型

  VarCast(v1,v2,varByte);
  writeln(TVarData(v1).VType); //显示 0，表示 V1 的值为 Byte 类型

  readln;
end.
```

4. 变体数组

Delphi 不允许将一个静态数组作为值赋给变体变量，但作为一个特例，我们可以将一个静态的变体数组赋给变体类型的变量。Delphi 提供了两个标准例程 VarArrayCreate 与 VarArrayOf 用于创建变体数组。varArrayOf 声明式为：

```
function VarArrayOf(const Values: array of Variant): Variant;
```

此例程将一个基类型为变体类型的普通数组转换为成员为 varVariant 类型的变体数组。如下例所示：

```
var
  v0:array[0..3] of variant;
  v1,v2:variant;
begin
  v1 :=  VarArrayOf(v0);    //v1 为变体数组变量，其值为一个含有 3 个成员的数组
  v2 :=  VarArrayCreate([0,3], varVariant); //v1 与 v2 完全相同
end.
```

VarArrayCreate 的声明式为：

```
function VarArrayCreate(const Bounds: array of Integer; AVarType: TVarType):
Variant;
```

如前文介绍，AVarType 标识目标数组的成员类型，在 System 单元中可查看它的值域。我们重点来看一下 Bounds 参数。

从声明中可知 Bounds 是一个以整型为基类型的动态数组。当我们所要创建的变体数组是多维数组时，Bounds 参数中从第 1 个成员起每两个成员代表了多维数组的其中一维的最小与最大成员序数值。由于 Bounds 的特殊用途，Delphi 规定了其成员数目必须是偶数且不能超过 128，否则会产生运行错误。示例：

```
v :=  VarArrayCreate([0,9,2,5], varInteger);
```

此行代码创建的变体变量 V 的值是一个二维数组，其中第一维有 9 个成员，序数分别为：0，1，2，3，4，5，6，7，8，9；第二维有 4 个成员，序数分别为：2，3，4，5。

我们声明一个普通的多维数组变量 V1：

```
var
  v1 : array[0..9] of array[2..5] of Integer;
```

读者可能发现 v1 与变体数组变量 V 很相似。事实不然，二者有本质上的不同：变体数组的成员并非一个单独的变量，无法获得其在内存中的单独地址，故而不能用指针指向变体数组变量的某个成员；同样，作为例程参数时也不可使用 var 或 out 的传递方式。

当然，普通数组与变体数组变量也有相同之处，如二者均可被索引，不过变体数组变量只能使用整数索引(普通数组的索引值可以是任一有序类型的值)。这也是 VarArrayCreate 的 Bounds 参数的基类型为 Integer 的原因。

在创建变体数组时，永远不要将 varString 传给 VarArrayCreate 以创建一个值为字符串的变体数组，取而代之，我们可使用 varOleStr。

类似于普通动态数组，在程序运行期间可以使用 Delphi 提供的标准例程 VarArrayRedim(动态数组使用 SetLength)动态改变变体数组变量的成员数目，此类例程还有：

```
VarArrayDimCount  VarArrayLowBound  VarArrayHighBound
VarArrayRef       VarArrayLock      VarArrayUnlock
```

关于这些例程的使用方法可参见 Delphi 的说明文档。

[注：]当某个变量的值为变体数组时，尽量不要将其赋值给另一个变量，因为这样会导致其中所含的所有的数组值被复制，而这种复制通常都会导致效率低下。这意味着将这样的变量作为例程参数时不适合选用默认的值传递及 const 传递方式，因为这两种传参方式传递参数时也会发生赋值行为。

附：变体变量能够容纳的值的类型及对应的类型码：

类型标识符	类型码数值	对应类型
varEmpty	0	Unassigned
varNull	1	Null
varSmallint	2	SmallInt
varInteger	3	Integer
varSingle	4	Single
varDouble	5	Double
varCurrency	6	Currency
varDate	7	Date
varOleStr	8	BSTR 或 WideString
varDispatch	9	IDispatch 接口
varError	10	Error
varBoolean	11	Boolean
varVariant	12	Variant
varUnknown	13	IUnknown 接口
varShortInt	16	ShortInt
varByte	17	Byte

varWord	18	Word
varLongWord	19	LongWord
varInt64	20	Int64
varUInt64	21	UInt64
varString	256	ShortString 或 AnsiString
varUString	258	UnicodeString

3.6 运算符

3.6.1 有序类型运算符

Delphi 提供了 5 个所有的有序类型共有的运算符 ord、pred、succ、hign、low。从外表看来，它们似乎有点像是标准函数。下面我们简要概括这些运算符各自的作用及用法。

1. ord：ord 可用于有序类型的表达式，主要用于变量，它返回指定值在值域中的序数值。如 ord('A') = 65，因为大写字母 A 在字符集中的序数为 65。注意 ord 不接受 Int64 类型的参数。
2. pred：与 ord 一样，用于有序类型的表达式。它返回指定值的前一个值，如 pred('B') = 'A',而 pred(8) = 7。
3. succ：与 pred 相反，它返回指定值的下一个值，如 succ(8) = 9;
4. high：high 与下面的 low 可用于有序类型表达式及有序类型本身。high 返回变量能够表示的的最大值或数据类型的上界，例如 high(byte) = 255。
5. low：与 hign 相反，它返回变量能够表示的的最小值或数据类型的下界。low 与 high 还可用于静态数组及短字符串。作用于短字符串时，它们分别用于得到数组的最大与最小序数值。短字符串实质上也是数组，故而雷同。

3.6.2 数学运算符

Delphi 中的数学运算符共有八个，其相关信息如下表所示：

运算符	运算功能	支持类型	返回值	示例
+	相加	integer, real	integer, real	X + Y
-	相减			Result -1
*	相乘			P * InterestRate
/	相除		real	X / 2
div	整除	integer	integer	34 div 3 = 10
mod	求余			34 mod 3 = 4

1. 上表中的“支持类型”表示运算符能够进行运算的类型，如 div 的支持类型仅限整数，表示 div 只能用于整数的相除，其返回值会自动舍弃小数部分，只保留整数。所以 29 div 10＝2。
2. x/y 的类型是 extended，无论 x 或 y 的类型是什么。对于其它的数学运算符，当运算数中有一个是实型时，结果自动转化成 extended；但运算数中至少有一个是 Int64 时，结果转化成 Int64。当运算数是 Integer 的子界类型如 Byte、word 时，编译器将其当成 Integer 类型。
3. Mod 表示求余，用于获得整数 X 除以整数 Y 后的余数，如 29 mod 10＝9；
4. 在 x mod y、x div y、x/y 中，y 不能等于 0。

除以上运算符外，Delphi 还提供了两一元运算符来进行正负运算：

运算符	运算功能	支持类型	返回值	示例
+	正运算	integer, real	integer, real	+7
-	负运算	integer, real	integer, real	-X

3.6.3 逻辑运算符

逻辑运算符有四种：not、and、or、xor。

1. not 用于得到指定逻辑的相反值，它只有两种情形：not(True) = False；not(False) = True；例如当集合 myset 中存在元素 c 时，not (c in myset)的值为 False。
2. and 类似于数学上的“且”运算，对于 A and B，只有 A 和 B 都为 True 时才能得到 True，其它情形全部得到 False。
3. or 类似于数学上“或”运算，对于 A or B，只要 A 和 B 中有至少有一个为 True，结果就为 True。当两者都为 False 时才会得到 False。
4. xor 表示“异或”运算，对于 A xor B，无论 A 与 B 的值是什么，当 A、B 值相同时，A xor B = False，当 A、B 的值不同时，A xor B＝True。

逻辑运算符只能运算逻辑类型的值，也就是说 if not 0 then...无法编译，因为 0 不是逻辑值，而 not 只能运算逻辑值。可以通过强制转型将 0 转化成逻辑值：if not boolean(0) then...。

在使用 or 进行运算时，若第一个运算数确定为 True，整个的表达一定返回 True，在这种情形下，第二个运算将不会被计算。这种机制实际上只执行了判断语句中的一部分代码，故而称为部分执行。

例如以下代码：

```
function B1:Boolean;
begin
  writeln('this is B1');
  Result :=  True;
end;

function B2:Boolean;
begin
  writeln('this is B2');
  Result :=  False;
end;
```

```
begin
  if B1 or B2 then Writeln('ok'); //B2 不会被执行
  readln;
end.
```

同样，在使用 and 进行运算时，若第一个运算数的值为 False，第二个运算数也不会被执行。读者可以将上例中的 if B1 or B2 then...改成 if B2 and B1 then...，然后运行并查看结果。

3.6.4 位运算符

每个字节都有 8 个位（bit）组成，位运算用于操作这些字节位。Delphi 提供了 6 个位运算符，分别为：not、and、or、xor、shl、shr。

1. not 用于将字节中所有的位的值变成取相反值。一个位的值只可能有两种 0 或 1。所以利用 not 运算时，它会将字节位中的 1 变成 0，0 变成 1。例如若某个值的二进制形式为：0111 1001，那么对此值进制 not 运算后其二进制形式变成：1000 0110。
2. and 将两个运算符的相同字节位进行比较，若两个位都是 1 则返回 1，否则返回 0。如对于 14 and 2＝2，14 在内存中的二进制表示：0000 1110，2 在内存中的二进制表示：0000 0010，两者进行运算的结果以二进制形式表现为：00000020，十进制表现为 2。
3. or 与 and 相反，当两个字节位的值都是 0 时返回 0，其它情况全返回 1。14 or 2＝14。
4. xor 在两个字节位相同时返回 0，不同时返回 1。所以 14 xor 2＝12。
5. shl 与 shr 较为麻烦：
   - 对于 X shl Y，编译器将 X 中的所有字节位整体左称 Y 的个字节位。如 byte 类型的值 14 在内存中的表现形式为：0000 1110，14 shl 2 后其表现为：0011 1000。可以发现，右端空出的字节位以 0 补全，左端称出的字节位被丢弃。
   - 在进行移动前编译器会将 Y 进行变化：假充 X 的类型在内存中占用 n 个字节位，则变换后的 Y 值为原来的 Y 值与 n 求余后的值 n。例如，若 X 为 Integer 类型，其在内存中占用 32 个字节，Y 的值为 40，则 X shl 40 会被变换成 X shl 8，因为 40 mod 32＝8。
   - 当 X 为有符号整数时，其最高位为符号位，这个符号位不参与字节移动。

   shr 在移动字节时会将字节整体右移，其它与 shl 完全相同。

所有的位运算符只能计算整数，其计算结果都是整数。在使用 shl 与 shr 例如 X shl Y 时，其中的 Y 也只能是整数。

以下是一些示例：

```
var
  n:Integer;
begin
  n :=  not 1.2; //错误
  n :=  4 shl 0.1;   //错误
  n :=  100 shl 2;   //正确
end;
```

3.6.5字符串运算符

字符串运算符只有一个：“+”。它将两个字符串或字符连接为一个字符串。如：'DEL'+'PHI' = 'DELPHI'。用于相加的两个字符串可以是任何的字符串类型，相加后得到的字符串可以赋给任何字符串变量。在将得到的字符串赋给短字符串变量时若字符串的长度过大，系统会自动截断，只保留前 255 个字节。

```
var
  s:shortstring;
  left:pchar;
  right:shortstring;
begin
  left := 'delphi';
  right := '2010';
  s := left+right;
  writeln(s);
  readln;
end.
```

这个例子中的 s、left、right 均可以是其它的任意类型，包括字符指针、长字符串、短字符串、字符数组甚至可以是字符类型。

3.6.6 集合运算符

Delphi 中的集合类型极其类似于数学的集合，其运算可分为三种类型：

- 对两个集合间进行合并、相减、相交从而得到另一个新集合。
- 判断一个集合中是否含有某个值。
- 判断两个集合的关系，如判断一个集合是否属于另一个集合。

约定 S1、S2 表示两个同类型的集合，X 为任一与集合同类型的值。则各集合类型的运算法则如下：

运算类型	书写方式	运算结果
并集运算	S1+S2	S1 与 S2 所有不重复成员组成的新集合。如：[1，2]+[2，3]＝[1，2，3]
交集运算	S1*S2	S1 与 S2 中所有相同的成员组成的新集合。如：[1，2]*[2，3]＝[2]
差集运算	S1-S2	去掉 S1 中所有与 S2 共有的成员后的新集合。如：[1，2]-[2，3]＝[1]
判断是否相等	S1＝S2	判断 S1 与 S2 中的成员是否完全相同，如相同则返回 True，否则返回 False
判断是否不等	S1<>S2	若 S1 与 S2 中的成员不是完全相同则返回 True，否则返回 False
判断是否包含	S1> = S2	若 S1 中含有 S2 的全部成员，则返回 True，否则返回 False

判断是否被包含	S1< = S2	若 S2 中含有 S1 的全部成员，则返回 True，否则返回 False
从属运算	X in S1	若 X 是 S1 的成员，则返回 True，否则返回 False

3.6.7 指针运算符

本部分我们介绍除了@、^之外的四个适用于指针的运算符。

“＝”运算符用于验算两个指针是否指向了同一个对象，若是则返回 True。

“<>”运算符用于验算两个指针是否指向了不同的对象，若是则返回 True。

“+”与“-”比较复杂一点，这两个运算符只能用于 PWideChar 及 PAnsiChar。

我们先定义两个字符指针：P1 和 P2，假设它所指向的对象在内存中占用 n 个字节。再定义一个整数变量 I。

“+”仅用于一个字符指针与一个整数相加的情形，如 P1+I 或 I+P1，表示 P1 的起始地址加上 n*I 个字节后得到的新位置。

“-”仅用于两个字符指针相减或一个字符指针减去一个整数：P1-P2 或 P1-I。

P1-I 与 P1+I 类似，它从 P1 起始的地址减去 n*I 个字节。

P1-P2 将 P1 的指针值减去 P2 的指针值，得到的值为两者的绝对数值除以 n 的商。例如若 P1、P2 为 WideChar 类型的指针，两者的指针值分别为 100、120，则得到的值是-10。

3.6.8 关系运算符

关系运算符用于计算两个运算数间的关系，包括等于、不等于、大于、小于、小于或等于、大于或等于，一共六种关系。下表归纳了这六种类型运算符的相关信息：

符号	关系	运算数类型	返回类型	示例
=	等于	简单类型、类、类引用、接口、字符串	逻辑值	I = Max
<>	不等于			X <> Y
<	小于	简单类型、字符串、PChar 指针		X < Y
>	大于			Len > 0
< =	小于或等于			Cnt < = I
> =	大于或等于			I > = 1

第四章 程序流程控制

Delphi 语言中能够控制流程的语句有三种：条件语句、选择语句、循环语句。下面逐一介绍：

4.1 条件语句

所谓条件语句即根据某个条件是否满足而决定下一步程序的运行方式。

完整的条件语句的书写格式为：

```
If <条件表达式> then
  <语句 1>
Else
  <语句 2>;
```

其中的语句可以是简单语句，也可以是复合语句。

此语句首先判断条件表达式的值，若为逻辑 True 则运行语句 1，若为 False 则运行语句 2。语句 1 和语句 2 可以是简单语句或复合语句，也可以是另一个条件语句。若某 If 语句的语句 1 及语句 2 中至少有一个是另一个条件语句，则称原来的条件语句为复合条件语句。

编译器将整个条件语句(if <条件表达式>Then<语句 1> else <语句 2>)看成是同一语句，else 之前的语句 1 不可以分号作为结尾。

下面我们编写一个小程序 CalcFees。此程序的功能非常简单：根据用户输入的货物重量计算所需的运费。运费的计算方法是：小于并等于 50kg 的部分运费为 0.25 元/kg；超过 50kg 但小于并等于 100kg 的部分运费为 0.35 元/kg；超过 100kg 的部分运费为 0.45 元/kg。

按这种方法，120kg 的货物所需运费为：50×0.25 + 50×0.35 + 20×0.45 ＝ 39 元。

```
uses
  SysUtils, Dialogs; //添加 Dialogs 单元

 var
   Money, Weight: Double;
   str:string;
 begin
   Write('请输入货物重量:');
   Read(Weight);
   if Weight < =  50 then
     Money :=  Weight * 0.25
   else
   begin
     if Weight < =  100 then
       Money :=  (Weight-50) * 0.35 + 50*0.25
     else
       if Weight > 100 then
         Money :=  (Weight-100)*0.45 + 50*0.35 + 50*0.25;
   end;
   Showmessage('运费的金额为:'+FloatToStr(Money)+'元');
 End.
```

上例中含有复合条件语句，读者可发现这样写法非常不直观。为了使代码更加有

条理，形如上例的复合条件语句通常写成如下形式：

```
if Weight < =  50 then
  Money :=  Weight * 0.25
else if Weight < =  100 then
  Money :=  (Weight - 50) * 0.35 + 50 * 0.25
else if Weight > 100 then
  Money :=  (Weight - 100) * 0.45 + 50 * 0.35 + 50 * 0.25;
writeln('运费的金额为:' + FloatToStr(Money));
```

这两种写法没有任何区别，但很明显第二种写法看起来更为直观。此种写法描述成一般形式为：

```
If 条件1 then
  语句1
Else if 条件2 then
  语句2
 …
Else if 条件N then
  语句N
Else
  其它语句;
```

系统执行时，将逐个判断每个条件，当遇到条件 X 的值为 True 时，将仅执行相应的语句 X，然后跳出整个 If 语句而运行接下来的代码。

上式中除了 If 条件 1 then 语句 1 外，其余语句若无必要均可省略。

4.2 选择语句

当我们需要从两种情形中选择其中一种时，条件语句非常适合。但若我们需要从多种情形中选择一种时，需要使用 if...else if ...end 复合条件语句。事实上，此种情况下我们有比复合条件语句更好的选择：选择语句：

选择语句的书写格式为：

```
Case 选择表达式 of
  常量1：语句1;
   ...
  常量N：语句N;
Else
  ...
End;
```

其中的语句可以简单语句或复合语句。与条件语句一样，else 及其后的语句可以省略，若未省略，则 else 语句必须放在最后。选择表达式的值必须为一个占用内存小于 32 字节的有序类型的值。

执行选择语句时，系统先判断选择器表达式的值并与其后的 N 个常量值比较，若其中的常量 X 值与选择器表达式的值相等则系统将仅执行语句 X(X 为 1 至 N 间的任一值)。若所有的常量值与选择器表达式的值均不相乘，则系统将仅执行 else 后的语句。

与上节一样，我们依然通过一个例子来演示选择语句的用法。我们将要编写这样一个程序：程序将根据读者输入的数目而在屏幕上显示相应数目的星号，但最多只显示 4

个。若读者输入的数目大于 4 将会出现一个提示。此程序代码如下：

```
program EchoStar;
{$APPTYPE CONSOLE}
uses
  SysUtils;

var
  N: integer;
begin
  write('请输入一个 1 至 4 间的数字:');
  read(n);
  case N of
    1:writeln('*');
    2:writeln('**');
    3:writeln('***');
    4:writeln('****');
  else
    writeln('所输的数字不在 1 至 4 之间');
  end;
  readln;
End.
```

选择语句中的选择器表达式可以是一个变量也可以是一个表达式(语法上也允许其为一个常量，但这样显然没有意义)，且其值的类型必须为有序类型。当选择器表达式的值与其后的常量的类型不同时，编译器将自动进行类型转换。但这种转换有时可导致未知错误，故而建议用户编程时手动显式进行类型转换。

4.3 循环语句

Delphi 中支持三种类型的循环：While，Repeat，For。

1.While 循环

While 语句的语法格式为：

```
While 条件表达式 do
  循环语句;
```

循环语句可以是任何合法的语句，也可以是另一个循环语句。条件表达式返回一个逻辑类型的值，当其为 True 时，系统执行其后的循环语句，每执行完一次循环语句时系统均会判断条件表达式的值，若为 True 则断续执行，若为 False 则终止执行当前的 while 循环而执行 while 循环后的语句。

以下程序将计算 1+2+3+…+N 的和，N 的值由用户指定(为降低复杂性，N 值不可大于 100)：

```
 program SumInt;
{$APPTYPE CONSOLE}
uses
```

```
  SysUtils;

var
  n,i:1..100;
  Sum:integer;
begin
  write('请输入一个 1 至 100 间的数字:');
  read(n);
  sum :=  0;     //将 sum 的值初始化为 0
  I :=  1;
  while i < =  n do      //当 i 的值小于 n 时执行循环体
  begin
    sum :=  sum+i;
    i :=  i+1;     //每执行完一次，i 的值增加 1
  end;
  writeln(inttostr(sum));      //显示 sum 的值
  readln;
End.
```

注意例子中的 i := i+1，若没有这句代码，i 的值永远为 1，程序将会一直执行循环至死机，即陷入死循环。

将 while 中的条件表达式省略时，也会致使此循环永不停止，这样的循环称之为空循环。

2.Repeat 循环

Repeat 循环与 while 循环非常类似，唯一的区别在于：while 循环第一次时先判断条件是否满足之后再执行循环休，而 repeat 循环第一次执行时先执行循环休再判断条件。

Repeat 语句的语法格式为：

```
Repeat
  [循环体]
Until <条件表达式>
```

将前面的 SumInt 改写成 repeat 循环，其代码为：

```
begin
  write('请输入一个 1 至 100 间的数字:');
  read(n);
  sum :=  1;     //将 sum 的值初始化为 1
  i :=  1;
  repeat
    i :=  i+1;
    sum :=  sum+i; //这两句语句的顺序不可调换
  until i = n;
  writeln(inttostr(sum));      //显示 sum 的值
  read(i);
End.
```

以下两个问题请读者自行思考：

- 为何 sum 的初始值由 0 变成了 1?
- Repeat 循环体中的两句语句的顺序能否调换?

3.For 循环

当我们知道将要循环的次数时，可用 for 循环。其语法格式为;

```
For 计数器 :=  初值 to 终值 do
  < 循环体 >
```

说明：

1.计数器必须为有序类型的变量，初值与终值必须为有序类型的常量或变量

2.计数器、初值、终值的数据类型必须相互兼容，最好全部相同

执行 For 语句时，系统先将初值赋予计数器(故而计数器不需要预先手动赋予初始值)，然后判断计数器与终值的关系，若计数器小于或等于终值，系统将执行一次循环，并将计数器的值加 1。当计数的值大于终值时，系统将停止循环，此时计数器的值等于终值加 1。当初值大于终值时不会发生循环。

For 语句还有另一种形式的格式：

```
For 计数器 :=  初值 downto 终值 do
  <循环体>
```

此种形式与前一种形式相反，系统每次执行完循环后，会将计数器的值减 1，直到计数器的小于终值为止。此种形式的 for 语句执行完毕后计数器值比终值小 1。显然，初值小于终值时将不会发生循环。

注意：对于任何形式的 for 语句而言，当初值与终值相等时，系统只执行一次循环。如下面的例子：

```
program ForSample;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  I,n:integer;
Begin
  write('输入 n 的值:');
  read(n);
  for i :=  1 to n do
    writeln('Delphi2010');   //系统执行 n 次循环
  writeln('i 的值为:'+inttostr(i));    //显示此时 i 的值
  read(n);
End.
```

上面的程序要求用户输入一个数字作为 n 的值，然后在屏幕上显示 n 行文字，当 for 执行完毕后 i 的值为 n+1。

当 n 的值为 1 时，只显示一行文字，且 for 语句执行完毕后 i 的值为 2。

注意，本例中 for 循环中的初值为 1，所以执行 n 次循环。当将初值改为其它值

时，系统将 X 次循环，其中 X 的值为：

```
X ＝ 终值 + 1 - 初值
```

若使用的是 for…downto…形式的循环，则 X 的值为：

```
X ＝ 初值 + 1 - 终值
```

4.For...in 语句

请读者思考一下，对于如下集合变量：

```
Var
  Myset:set of AnsiChar;
Begin
  Myset :=  ['A','B','C','D'];
End.
```

若现要求将其中的所有成员逐一在屏幕上显示出来，读者会怎么做？

可能读者会想到利用索引：

```
Var
  Myset:set of AnsiChar;
  I:integer;
Begin
  Myset :=  ['A','B','C','D'];
  For i :=  1 to 4 do
    Writeln(myset[i]);
End.
```

看起来很完美，但如果读者的记忆力还不错的话，就应该知道集合类型是不支持索引的。所以以上代码无法通过编译。这咱时候我们可以使用 For...in 循环语句。

For...in 循环是 Delphi 新版本中新加入的一种循环语句，习惯上称为遍历语句，关于其作用，笔者在此暂不介绍，若读者看完下面的例子后还不明白，那我说了也是白说。遍历语句的语法格式为：

```
For V in set do <循环体>
```

V 为任一合法变量。Set 为集合变量。V 与 set 的基本类型必须一致，至少相互兼容。

For...in 循环过程如下：先将集合中的第 1 个成员赋予变量 V 并执行循环体中的语句，然后将第 2 个成员赋予 V 并执行循环体中的语句…依次类推，直到最后一个成员，此过程称之为集合的遍历。注意：集合成员具有无序性，故系统执行时将根据赋值时各成员出现的先后顺序来运行。如下面的例子：

```
program ForSample;
{$APPTYPE CONSOLE}
uses
  SysUtils;

Var
  Myset:set of AnsiChar;
  ch:AnsiChar;
Begin
  Myset :=  ['D','B','C','A'];
```

```
  for ch in myset do
    Write(ch);
  readln;
End.
```

执行以上代码，屏幕显示：DBCA。

除 Set 类型外，for...in 还能遍历以下类型：

- 数组。一维二维多维均可，动态静态都行。遍历时将按序号由小到大进行遍历。
- 字符串。按照前后顺序从第一个字符开始遍历。
- 类、接口、记录。对于这三种类型的变量，实现遍历功能的过程较为复杂，一般很少使用，在此我们不再介绍。Delphi 预先提供了以下几个类用以支持对象的遍历：

```
Classes.TList
Classes.TCollection
Classes.TStrings
Classes.TInterfaceList
Classes.TComponent
Menus.TMenuItem
ActnList.TCustomActionList
DB.TFields
ComCtrls.TListItems
ComCtrls.TTreeNodes
ComCtrls.TToolBar
```

下面以 Classes.TStrings 给出一个相关的示例：

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils, Classes;

var
  StrArray:TStringList;
  Item:String;
begin
  StrArray :=  TStringList.Create;
  StrArray.Add('stringA');
  StrArray.Add('stringB');
  StrArray.Add('stringC');
  for Item in StrArray do
    writeln(Item);
  Readln;
end.
```

运行这个程序，屏幕上显示以下 5 条字符串：

```
stringA
stringB
stringC
```

可能读者会奇怪，这个例子中使用的类是 TStringList 而非 TStrings，为何？我们知道，类具有功能，TStrings 中的成员能够被遍历，也就意味着其所有的子类也可被遍历，而 TStringList 正是 TStrings 的子类。但在这个例子中我们不使用 TStrings 的原因并不仅限于此，还有另一个重要原因是：TStrings 是个抽象类，其中仅仅只是声明了相关的方法而并未具体这些功能，所以无法用于遍历。

以上例子同时也说明：对于任何一个类，只要这个类的某个祖先类支持遍历，这个类就可以被遍历。

5.Continue 与 Break 语句

continue 与 break 虽非循环语句，但此二者只用于前面介绍的四种循环语句，故在此一并介绍。continue 用于跳出当前正在执行的一次循环并重新开始新一次的循环。Break 用于停止循环而执行循环语句后面的代码。

下面我们将通过一个例子说明一下此二者在 for...in 循环中的使用，当用于其它循环语句时作用与 for...in 中一样。

```
program BreakSample;
{$APPTYPE CONSOLE}
uses
  SysUtils;

var
  ch:char;
  str:string;
begin
  str :=  'ABCDEFGHIJKLMN';
  for ch in str do
  begin
    if ch = 'H' then
      break;
    writeln(ch);
  end;
  readln;
End.
```

此程序遍历字符串 str 并显示其中的字符，但在循环体中设置了一条件：当遇到字符串中的'H'时停止遍历，直接执行循环语句后面的代码即'readln;'。故而屏幕上只显示'ABCDEFG'。

若将代码中的 break 改为 continue，循环体的条件则变为：当遇到'H'时不再断续执行此次循环而会重新开始一次新的循环，所以屏幕上不会显示'H'，只显示'ABCDEFGIJKLMN'。

4.4 程序中止例程

利用程序中止例程可以使得一个正常的运行的程序强行中止。程序中止的结果有两种，一是仅仅只退出当前正在运行的语句，但不一定会退出整个程序；二是直接退出整个

程序。

Delphi 常用的程序中止方法有三种：使用 Exit 例程、使用 Halt 例程、调用全局程序对象的 Terminate 方法。

1. Exit 例程

Exit 声明于 Delphi 的标准 System 单元，其声明原型为：

```
procedure Exit;
```

Exit 用于退出当前正在执行的程序块，当不会退出整个程序，除非将 Exit 用在了程序的主程序块（即.dpr 文件中的程序块）中。下面的程序演示了 Exit 例程的用法：

```
procedure M1;
begin
  exit;
  writeln('M1');
end;

begin
  writeln('Start Program');
  M1;
  writeln('Ending Program');
  //Exit;
  readln;
end.
```

运行后的结果为：

```
Start Program
Ending Program
```

若读者将例中的注释语句去掉注释符号后再运行程序时就会发现，程序执行后的窗口一闪而过，根本不会停留在屏幕上。

在使用 try...finally...类型的异常处理语句时，在 try...finally 部分中执行的 exit 会被当成一个异常来处理。详见”异常处理“部分。

自 Delphi2009 开始，Exit 后可接一个参数以传递函数退出时所返回的结果。这种应用必须满足两个条件：

1. 后接参数的 Exit 只能用于函数中
2. 参数的类型必须与函数的返回值的类型相同或兼容

下面的程序说明了这种用法：

```
function DoSomething(aInteger: integer): string;
begin
  if aInteger < 0 then
    Exit('Negative');
  else
    Result := 'Positive';
end;
```

```
var
  i: integer;
begin
  writeln(DoSomething(-1));
  readln;
end.
```

例中定义了函数 DoSomething，它接受一个 Integer 类型的参数。当传给 DoSomething 的参数的值小于 0 时，函数会返回一个字符串'Negative'并退出。

通过上面的例子我们可以发现，接参数的 Exit 事实相当于将两条语句简化成了一条。如上例中的 Exit('Negative')相当于以下的复合语句：

```
begin
  Result :=  'Negative';
  Exit;
end;
```

2. Halt 例程

与 Exit 不同，Halt 会引发一个异常中断并通过这个中断直接退出整个的程序。System 单元中 Halt 的声明原型为：

```
 procedure Halt([ExitCode:Integer]);
```

Halt 后可接一个 Integer 类型的标识码用于标识程序退出的相关信息。这个标识可以省略，省略时编译默认其值为 0。

Halt 会直接退出整个程序。这句话的意思是：不管在何种地方，以何种形式，只要调用了 Halt 就会退出整个的程序。请看下面的示例：

```
uses
  SysUtils, Dialogs;
procedure M1;
begin
  Halt;
end;

procedure M2;
begin
  Writeln('M2');
  M1;
end;

begin
  ShowMessage('Starting Program');
  M2;
  ShowMessage('Ending Program');
end.
```

程序中调用了 M2，这会导致程序的中断。

3. Terminate 方法

Delphi 中的窗体程序中含有一个全局程序对象(一般是 Application)，其中的 Terminate 方法可在窗体程序的任何地方使程序正常中断并退出：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.Terminate;
  showmessage('');
end;
```

第五章 函数与过程

实际应用中经常需要将一个完整的程序划分几个子程序，每个子程序用于实现某种特定的功能，多个子程序组合从而实现更复杂的功能。如同一个公司内部有多个部门，每个部门组合起来才能完成一个完整的任务一样。可以使用过程或函数来实现这些子程序。虽然名称不同，但二者在诸多方面完全等效，除了以下两方面：

- 二者声明时所使用的关键字不同，过程使用 procedure 声明，而函数则使用 function 声明。
- 声明时过程不能指定返回值，而函数必须指定。

为方便起见，Delphi 将过程与函数统称为例程(routine)。

类似于变量，例程亦分为预定义例程及自定义例程。使用 Delphi 的预定义例程前，必须直接或是间接引用(参见第一章)例程所在的单元。而使用自定义例程之前，必须声明此例程。(在.dpr 文件中我们可以将一个例程的声明与定义合在一起；但在.pas 文件中必须先声明，然后定义，除非是套嵌例程)

5.1 例程的声明

1. 普通声明

声明一个例程的语法格式如下：

```
procedure 过程名(形参列表);[指示字]; [调用约定;]          //用于声明一个自定义过程
function 函数名(形参列表):返回值类型;[指示字]; [调用约定;]//用于声明一个自定义函数
```

声明时的语句形式称为函数或过程的首部。过程名与函数名为任一合法标识符，只需避免名称冲突即可。

形参列表的一般形式为：

```
var 参数 1: 类型名 1;  var 参数 2: 类型名 2;  … var 参数 n: 类型名 n
```

我们将形参列表中诸如“var 参数 1: 类型 1; ”的结构单元称为一个形参项其中的

var 表示参数的传递方式，可用 out 或 const 代替。对于形参列表只需注意两点：

- 同一例程中不允许有同名的参数。
- 列表中的每个形参项后以分号结尾，但最后一个形参项后不允许有分号

返回值类型表示所返回的值的数据类型，不能是任何形式的文件类型。

指示字为某些特定的关键字，用 virtual、abstract 等。在声明例程可以不加任何指示字。

调用约定见于后续章节。

如同声明多个变量一样，当形参列表中有相同类型的参数时，我们可以将这些参数放在同一形参项中声明，其中的每个参数名称间以逗号相隔。如以下形参列表：

```
var p1:integer; var p2:integer
```

可简写成：

```
var p1, p2:integer
```

2. external 声明

有时我们可能需要从其它的编译单元如.obj 或.dll 文件中引用一个例程。声明此类例程时我们必须在指示字的位置指定 external 关键词。

声明一个源于.dll 文件中的例程的语法格式为：

```
procedure sample(var s:string); external 'SomeDLL.dll';
```

external 后应接上.dll 文件的完整名称(包括路径名和文件名)。当.dll 与当前编写的程序位于同一文件夹时可直接以文件名标识.dll 文件。

声明一个源于.obj 文件中的例程的语法格式为：

```
procedure sample(var s:string); external;
```

在声明之前必须在程序中引用相应的.obj 文件。其格式为：

```
{$L BLOCK.OBJ}
```

3. forward 声明

Forward 用于提前声明一个例程。它可使得例程在定义之前就可被使用。例如下面的程序：

```
program forward_sample;
{$APPTYPE CONSOLE}

uses
  SysUtils;
procedure fun1; forward;   //当此处的 forward 去掉时将无法通过编译

procedure fun2;
begin
  fun1;
end;
```

```
procedure fun1;
begin
  //
end;

begin
end.
```

5.2 例程参数

本节我们将先介绍默认参数，然后将介绍两种特殊类型的参数，最后将详述例程参数的 4 种传递方式。

5.2.1. 形参与实参

所谓形参，是指在声明一个例程时在例程的形参项中声明的变量或常量。例如对于如下的声明中：

```
    procedure sample(var s:string; var i,:integer);
```

s 与 i 在 sample 的形参项中被声明，故 s 和 i 均是形参。

```
    注意：例程参数不能是静态数组及文件类型。
```

所谓实参，是指在例程运行时所使用的实际参数。以 sample 为例，调用时我们使得,s 的值为'delphi'，i 的值为 101:

```
    sample('delphi', 101);
```

则 samplesample 的实参为'delphi'及 101，只不过在运行期间我们用 2 个标识符：s、i 作为二者的名称而已。

5.2.2. 参数传递

在介绍例程的参数项时，我们曾经提到其中的 var 表示参数的传递方式(传参方式)。Delphi 一共支持 4 种传参方式，分别为：传址方式，传值方式，常量方式，输出方式。下面逐一介绍。

(1). 传址方式:

例程中以关键词 var 声明的参数采用传址方式传递。这种类型的参数在例程运行时相当于将变量的地址传给例程。若例程运行期间，若形参数的值发生了改变，实参变量的值将会随之改变。请看下面示例：

```
procedure var_sample(var i:integer);  //定义一个过程
begin
  i :=  2*i;
  writeln(i);    //显示 i 的值
end;

var
```

```
  n:integer;
begin
  n :=  9;
  var_sample(n);
  writeln(n);    //输出 18
  readln;
end.
```

此例中我们先定义一个例程 var_sample，其以 var 方式接受一个参数并在运行期间将此参数的值加倍。当我们将变量 n 作为参数传递给它时，n 的值发生了改变。

调用例程时，传递给 var 参数的一定是能够被赋值的变量或表达式，绝不能是常量。读者可尝试一下：将上面程序中的 n 声明成一个整型常量，然后编译看一下结果。此条规定有 2 个例外：

- 当将一个对象类型的常量传入例程时，对象的属性可以被改变。
- 若开启了{$J+}开关，常量与变量无甚区别，此时亦可将常量传入。

下文中的 out 参数亦有此规定。

(2). 传值方式:

例程中声明参数时，若其前未加任何关键字，则此参数将采用传值方式传递。以此种方式传递的参数在变量运行期间其值也可发生改变，但 var 方式不同的是，这种改变不会影响到原来的变量。实际上，当遇到传值方式传递的参数时，系统会在参数传递前将参数复制一份，然后将复制得到的新变量传给例程，当例程运行完毕时，此新变量被销毁。例如当我们将上面的例子中 var_sample 的声明改成下列方式时：

```
  procedure  var_sample( i:integer);    //去掉 i 前的 var
```

我们可以看到 i 的值显示为 18，但 n 的值却并未发生变化，依然显示为 9。在传递过程中，系统将 n 的值复制一份()假设为 n2)并将复制后的值(n2)赋给了 i，故而原来的 n 根本没有变化。而等 var_sample 运行完毕后 n2 即被销毁。

(3). 常量方式

以常量方式传递的参数以 const 声明。可以说，const 方式是 4 种传递方式中最简单的一个。读者只需记住：此种方式传递的参数无认在何时都不会发生被改变，若调用时强制改变其值，只会导致编译错误。不过若将一个对象引用或是指针以 const 方式作为参数传入时，我们依然可以更改对象的属性值或字段值，也可以更改指针所指向的变量的变量值。

读者可尝试将上例中的 var_sample 中的参数 i 改为 const 声明后再编译：

```
  procedure  var_sample( const  i:integer);    //用 const 声明 i
```

由于 i 在运行过程中发生了值的改变，故以 const 声明 i 根本不会通过编译。

读者需要注意：当用 const 声明一个指针类型的参数 p 时，p 的值不会被改变(即 p 不可指向其它的变量)，但 p 所指向的变量的值却可以被改变。如下面的程序：

```
type
  pi = ^integer;
  //^integer 并非合法标识符，不能用于声明参数，故先声明新类型
procedure var_sample(const i:pi);
begin
  i^ :=  2*i^;
  writeln(i^);
```

```
end;

var
  n:integer;
begin
  n :=  9;
  var_sample(@n);
  writeln(n);
  readln;
end.
```

(4). 输出方式

以 out 关键字声明的参数将以输出而非输入方式传递。举例说明，以 out 声明的参数像是饮料生产线上的空瓶，经过不断的传递及加工，开始时传入的空瓶被装满饮料后传出生产线。很明显，我们需要的是饮料而非空瓶，空瓶仅仅只是为了装饮料而被传入生产线。

看一下例子：

```
function sample1(s:string):integer;
begin
  result :=  length(s);  //函数返回字符串 S 的长度
end;
```

此函数非常简单：传入一个字符串，函数将并返回此字符串的长度。利用 out 参数可将其改成以下形式：

```
procedure sample2(s:string; out leng:integer);
begin
  leng :=  length(s);
end;
```

分别调用 sample1 与 sample2:

```
var
  i,n:integer;
  s:string;
begin
  s :=  'delphi';
  i :=  sample1(s);
  sample2(s, n);
end;
```

运行完毕后，i 与 s 的值相等，均为 6。

看到这里，读者一定发现：即然可以使用函数返回特定值，为何还要使用 out 参数？理由：函数只能返回一个值，而有时需要同时返回多个值。而使用 out 参数就可以同时返回多个值。这种特性使得 out 参数在编写 COM 程序时非常方便。

就使用上而言，var 参数与 out 参数完全一样，var 参数也可存储返回值，二者唯一的区别在于：将一个变量作为 out 参数传入例程之前，系统会自动清空变量原来的值；而 var 方式则没有这种处理。例如下面的示例中，var_sample 没有进行任何操作，但将 str 作为参数调用 var_sample 后，str 却变成了空值：

```
procedure var_sample(out s:string);
begin
  //此例程不进行任何操作
end;

var
  str:string;
begin
  str :=  'delphi';
  writeln(str);   //显示'delphi'
  var_sample(str);
  writeln(str);  //不显示任何内容
  readln;
end.
```

这是由于 var_sample 以 out 方式使用了 str 作为实参，系统将 str 传入前预先将 str 的值设置为空，所以即使 var_sample 没有进行任何操作，str 依然变成了空值。正如装饮料的瓶子，在送入生产线之前肯定需要清洗干净才能使用。

5.2.3. 默认参数

大部分例程在声明时均被指定若干个参数(也可以没有任何参数)。在调用这些例程时需要给所有的参数赋值供例程执行期间使用。

有些时候我们可能会遇到这样的烦恼：某个例程在程序需要频繁使用，而此例程中的某些参数值在大部分的调用中均被赋予相同的值。在这种情况下，我们可在例程声明时替这些参数设置默认值，即默认参数。

设置例程的默认参数的格式为：

```
procedure 名称(…; var 默认参数名:类型 = 默认值);
```

设置默认参数后，若调用例程时未指定相应参数的值，编译器将自动将默认值赋予参数。如下列例程声明：

```
function myfun(var i:integer = 3;  var s:string = 'Delphi'):string;
```

若调用此例程时未指定参数 s 的值，编译器自动将字符串'Delphi'赋予 s。下列调用方式均返回相同结果：

```
myfun(3, 'Delphi');
myfun();  //i 与 s 均取默认值
myfun(3);     //s 为默认值
```

由上可以看出默认参数使用相当方便。不幸的是，并非所有类型的参数都能指定默认值。显然，只有直接常量值才可指定给参数作为默认值，而 Delphi 的某些类型根本就没有常量值，自然，这些类型的参数就不能指定一个常量值作为默认值。这此类型包括：

```
    记录、变体、文件、静态数组、对象类型
```

而以下类型的参数则仅能 nil 作为默认值：

```
    动态数组、例程类型、类、类引用、接口
```

除数据类型的限制外，Delphi 默认参数还有如下规定：

1. 当参数列表中某一个参数被指定了默认值时，此参数后的所有参数必须指定默认值：

```
function myfun(var a:integer; var b:string = 'delphi'):integer; //正确
procedure sample(var a:integer = 9; var b:string);
//错误，位于参数 a 后的所有参数均应指定默认值
```

类似地，在调用例程时，若某参数使用了默认值，则此参数后所有指定了默认值的参数均应使用默认值。

2. 当某个参数项中含多个参数名时，此参数项中所有参数均不可指定默认值：

```
procedure sample(var a, b: integer = 9);  //错误
procedure sample(var a:integer = 9; var b:integer = 9);  //正确
```

3. 当某个例程类型中指定了默认参数时，这些默认参数将掩盖具体函数中的默认值。这句话可能不太好　懂，读者只需记住默认参数与例程类型有关即可。我们将在“例程类型”部分详述这个问题。

若某例程有默认参数，则声明此例程时必须指定这些默认值，在定义时则例程时则可省略不写，若不省略则应保证函数首部与声明中的形式完全一样。

在遇到重载例程时，默认参数也很容易造成歧义。关于此问题我们同样在“重载例程”部分叙述。

5.2.4. 几种特殊类型的参数

有几类特殊的参数需要特别关注：无类型参数、短字符串、数组。

1. 无类型参数

所谓无类型参数即例程声明时没有指定所属类型的参数。如：

```
function Sample( var C):integer;
```

声明时无类型参数的传递方式可以指定为 const、out 或是 var 方式，但绝不能为默认的传值方式。

调用例程时，不可将整型直接常量或值为整数的无类型符号常量赋予无类型参数，但整型的类型常量可以。如下面的示例：

```
procedure fun(const c);
begin
  //此例程不执行任何操作
end;
const
  d = 987;//声明值为整型的无类型常量
  c:integer = 987;   //声明整型类型常量
begin
  fun('897'); //正确，
  fun(c);     //正确，c 为类型常量
  fun(d);     //错误，不可将值为整数的符号常量赋予无类型参数
  fun(876);       //错误，不可将整型直接常量赋予无类型参数
end.
```

在例程运行期间，无类型参数与其它任何类型均不兼容，故使用前必须利用类型转换将无类型参数显式转换成所需类型。不过编译器在编译时并不确保这种转换一定有效。

下面的例子源自于 Delphi2010 说明文档，其中定义了函数 Equal，当 Source 与 Dest 的前 Size 个字节值完全相等(即每个相对应的字节值均相等)时返回 True，否则返回 False。由于 Source 与 Dest 均被声明为无类型参数，故此函数可以接受多个类型的实参。但使用时参数将 source 和 dest 转换为相应类型后函数才能发挥作用，如下划线语句所示。

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N : Integer;
begin
  N :=  0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do  //参数类型转换
    Inc(N);
    Equal :=  N = Size;
end;
```

2. 短字符串参数

讨论完无类型参数，我们接着来看一下短字符串参数。我们知道，在声明一普通短字符串变量时，可以使用两种方式：

- 一是直接将变量声明为 shortstring 类型；
- 二是利用 string 后限定长度从而定义特定长度的字符串，如 string[9]。

但在声明例程的参数时不能使用第二种方法。也就是说类似于以下的声明将无法通过编译：

```
procedure Check1(var S: string[20]);
```

取而代之，可以先定义一个类型名，然后再定义参数：

```
type
  Mystring = string[20];
procedure Check2(var S:Mystring);
```

也可使用 Delphi 提供的 openstring 类型解决。此类型的参数接受任意长度的短字符串。若同时开启了{$H+}及{$P+}，则例程中 string 类型参数与 openstring 类型参数等价。

openstring 仅仅只是为了与其它版本代码兼容而保留(事实上在新版本中短字符串类型也很少使用)。

在 Delphi2010 中我们应当尽量使用长字符串如 AnsiString 或 UnicodeString 等。

实际上，短字符串参数具有这种限定的深层原因是因为短字符串是一个字符数组。接下来我们更进一步，直接讨论数组参数。

3. 数组参数

数组参数的规则：声明一个数组类型的参数时不可指定数组的索引值。例如我们声明以下的例程：

```
procedure sample(var a:array[0..9] of integer);
```
此声明无法通过编译。因为数组中指定了索引值：0..9。

解决这问题有两种方式：

- 先声明一个新的数组类型，再声明参数。如：

```
type
  myarray = array[0..9] of integer;
procedure sample(var a:myarray);
```

- 利用开放数组。如：

```
procedure sample(var a:array of integer);
```

由于开放数组参数的声明方式与动态数组一致，当我们需要声明动态数组变量时，我们只能先定义一动态数组类型，再用此类型去声明参数：

```
type
  darray = array of integer;
procedure sample(var a:darray);
```

读者可能发现开放数组和动态数组是如此的相像。事实上二者大部分特性也确实相同。二者唯一的区别在于：开放数组既可以容纳任意长度的静态数组，也可能容纳动态数组；而动态数组类型的参数只能接纳动态数组，无法接纳静态数组。下面是一个示例程序，读者可以尝试将其中 check 的参数 s 改为 darray 类型，再编译一下看看是什么结果。

```
program Sample;
{$APPTYPE CONSOLE}
uses SysUtils;
type
  darray = array of integer;  //声明动态数组类型 darray
procedure check(var s: array of integer);
begin
  //
end;
var
  s:array[0..9] of integer;
begin
  check(s);
end.
```

现在来看一下上面的 check1 无法通过编译的原因：由于 string[20]相当于一个字符串数组(参见附录 D)，check1 的声明相当于：

```
procedure Check1(var S: array[0..20] of ansichar);
```

所以 Check1 无法通过编译。

数组参数声明时不能指定索引，这其后事实上由另一个规则决定。

我们知道在声明例程时， 参数声明的基本格式为：

var 参数：类型名

其中的类型名只能是标识符或标识符的组合，而在在 check1 中，我们使用 string[20]来表示参数 s 的类型，很明显 string[20]并不是一个合法的标识符。同样，数组索引如[0..9]也并非一个合法的标识符。故而它们均不可用于参数的声明。开放数组 array of

type 虽非标识符，却是由三个合法的标准标识符组合而成，所以其可以用于声明参数。
根据此规则，我们同样不能以^type 的形式声明一个指针类型的参数。

至此，我们讨论了普通数组及开放数组类型的参数。事实上 Delphi 中还存在另一类数组参数：

Variant Open Array Parameters，直接翻译为变体开放数组参数。此参数类型为一个由变体组成的数组，故传递参数时，可以传递一个成员数据类型不一致的数组。

变体开放数组参数的声明一般的开放数组参数的声明方式非常相似，唯一的区别在于其参数类型为 array of const。例如我们可以定义如下的一个函数，它接受一个变体开放数组作为参数：

```
procedure M1(value:array of const);
```

array of const 实际上等效于 array of Variant。其它内容请查阅变体相关章节。下面给出一个例子：

```
function MakeStr(const Args: array of const): string;
var
  I: Integer;
begin
  Result :=  ;
  for I :=  0 to High(Args) do
     with Args[I] do
        case VType of
          vtInteger:  Result :=  Result + IntToStr(VInteger);
          vtBoolean:  Result :=  Result + BoolToStr(VBoolean);
          vtChar:     Result :=  Result + VChar;
          vtExtended: Result :=  Result + FloatToStr(VExtended^);
          vtString:   Result :=  Result + VString^;
          vtPChar:    Result :=  Result + VPChar;
          vtObject:   Result :=  Result + VObject.ClassName;
          vtClass:    Result :=  Result + VClass.ClassName;
          vtAnsiString:  Result :=  Result + string(VAnsiString);
          vtUnicodeString:  Result :=  Result + string(VUnicodeString);
          vtCurrency:    Result :=  Result + CurrToStr(VCurrency^);
          vtVariant:     Result :=  Result + string(VVariant^);
          vtInt64:       Result :=  Result + IntToStr(VInt64^);
  end;
end;
```

对于此函数，当我们以下列的方式调用它时：

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

程序会返回一个字符串：

```
'test100 T3.14159TForm'
```

5.3 例程的定义与使用

5.3.1 例程的定义

在本节开始之前，读者应当明白定义与声明间的区别。对于某个具体的例程而言，声明相当于通知编译器此例程在当前范围内有效，可以使用；而定义则相当于告诉编译器此例程处理问题的详细步骤。

定义一个例程的基本格式为：

```
function 名称(参数列表)：返回值类型;   //例程首部
局部声明区
begin
  //例程的实现代码
end;
```

或

```
procedure 名称(参数列表);   //例程首部
局部声明区
begin
  //例程的实现代码
end;
```

局部声明区域可声明数据类型、常量、变量甚至是另一个例程(即套嵌例程)。需要注意两点：

- 局部声明区域声明时所用的标识符不能与例程名称、本例程的参数名称相同，若例程是一个函数，则标识符 result 也不能被使用。
- 例程的局部声明区域所声明的所有标识符包括变量名、常量名、类型名、套嵌例程名等只在本例程内有效。(特殊情况如变量劫持等除外)

若例程预先声明过，在定义此例程时可以只保留声明关键字(function 或 procedure)与例程的名称而省略其中的参数列表、返回值等部分。如对于下列例程的声明：

```
function GetLength(s:string):integer;
```

在定义时可使用下列两种方式：

```
function GetLength;  //省略式
begin
  result :=  Length(s);
end;
```

```
function GetLength(s:string):integer;
//完全式
begin
  result :=  Length(s);
end;
```

从编程风格角度，推荐读者采用完全式的写法，在例程定义时保证例程的首部与声明时完全一致。

5.3.2 函数的返回值

1. Result 变量

result 是由 Delphi 声明的一个预定义变量，其作为局部变量被隐含地声明于例程。前面曾提到函数与过程的区别在于函数有返回值而过程没有，而 result 变量正是函数的返

回值。

更确切一点，函数运行完毕时会将变量 result 的值当成返回值。如此，若某个函数运行时没有给 result 变量赋值，则函数将返回一个未知值。相对地，若运行某个函数时，result 被多次赋值，函数的返回值将是最后一次赋给 result 的值。如下面例子所示：

```
function sample: integer;
begin
  result :=  98;
  result :=  56;
end;
```

此函数将返回 56。

若读者曾经学习 C 系语言如 C、C++等，可能会觉得 result 与 C 系语言中的 return 操作符有些相似。但事实并非如此。除了使函数返回一个值这点相同以外，二者有着本质上的不同：

- return 是操作符，其强制函数返回，即使函数并没有运行完毕。
- 而 result 是一个变量，只是函数在运行完毕时将其值作为返回值而已。使用 result 不会对函数的运行流程有任何影响，例程不会因 result 被赋值而终止运行。

2. 函数名作为返回值

函数运行时，我们可以使用 result 变量存储返回值，也可以直接使用函数的名称代替 result 存储。例如上面的 sample 函数也可以改成如下形式：

```
function sample:  integer;
begin
  sample :=  98;
  sample :=  56;
end;
```

对于预定义的 result 变量而言，只有编译开关{$X}处于{$X+}状态时，此变量才有效。若开关处于{$X-}时，将不会存在所谓的预定义 result 变量，在此种情况下，函数只能通过函数名来返回一个值。不过幸好，{$X+}是系统的默认设置，读者只要注意不要手动添加{$x-}将其关闭即可。

相比 result 而言，函数名作为返回值有其不便之处，如函数名不能作为变量进行类型转换等变量操作。另外有时使用函数名可能会被当成是循环调用函数。鉴于此，建议读者在任何时候均使用 result。

5.3.3 例程的调用约定

所谓调用约定是指调用例程时系统对于例程实参的一种处理方法，其影响参数的入栈及出栈顺序、传递参数时寄存器的使用、程序的错误与异常的处理。其中与用户最为相关的是其决定了调用例程时参数的传递顺序。例如下函数：

```
function sample(var s:string; var i:integer: ch:char):integer;
```

在调用此函数时需要向其传递三个参数，问题是如何决定哪个参数传递哪个后传递？最容易想到的方法是根据声明时的顺序来决定：s 最先被传递，i 其次，ch 最后。这也 Delphi 默认的方式。但是，在 C 语言中，参数的传递顺序却正好与此相反：ch 最先，i 其次，s 最后。通过指定调用具体的调用约定可以改变参数的这种处理方式。

Delphi 提供 5 种调用约定，按参数的传递顺序可分为两：一是从左至右传递，包括

pascal 与 register 两种方式，Delphi 默认采用 register 调用约定；二是从右至左的方式，包括 cdecl、stdcall、safecall 三种。

由于例程的调用约定有点超出 Delphi 语言的范畴，故而在明白调用约定的大致概念后我们只须知道以下几点：

- 通常，默认的 register 最为有效。若某个对象属性的访问权限为 published，此属性的读写方法必须使用 register 方式。
- 当调用某个使用 C/C++编写的共享库(如.dll 或.lib)中的例程时，必须使用 cdecl
- 当调用其它的外部代码时，尽量使用 stdcall 或 safecall。Windows API 函数大多使用这两种方式。

5.3.4 例程的内嵌(inline)

我们先来了解一下调用例程时系统的大致执行过程：当系统需要调用一个例程时，会暂停当前代码(我们假设此代码的位置为 CP)的执行而计算好例程所需的所有的参数，然后进入例程代码所在的区域并执行例程代码，执行完毕后系统会携带例程的返回值(如果有的话)返回 CP 继续运行调用例程之前正在运行的代码。

对于多数情形而言，按以上的方式调用一个例程并不会浪费多少时间。但在某些情况下需要频繁调用一些较为短小的例程时，这种浪费会相当的可观。兼于此，Delphi 提供了 inline 指示符用于内嵌一个例程。

所谓内嵌例程，顾名思义就是指编译器在编译时会将某个例程的定义代码直接拷贝至调用它的位置。如若有以下调用：

```
function add(x, y: integer):integer; inline;  //内嵌例程，用于返回两数之和
begin
  result :=  x + y ;
end;
var
  s : integer;
begin
  s :=  add(2,3);
  writeln(s);
end.
```

编译时编译器会直接将 add 中的代码拷入调用位置，以上的代码编译后相当于以下代码：

```
var
  s : integer;
begin
  s :=  2 + 3;    //直接将 x+y 拷贝至当前位置
  writeln(s);
end.
```

内嵌例程的定义相当简单，如例中所示：直接在普通函数的声明式之后加上关键字 inline 即可。内嵌例程可以使程序执行速度提高，但随之而来的是程序文件体积的增大，所以内嵌例程实质上是以空间抽取时间，这与后面介绍的动态方法有点相似。

并非所有的例程都可内嵌至目标代码，以下情形中使用的例程将不会被内嵌：

1. 任何迟绑定方法包括 virtual, dynamic, message 不能内嵌

2. 含有汇编代码的例程不能内嵌

3. 类的构造函数与析构函数不能内嵌

4. 主程序块、单元的 initialization 及 finalization 部分中的代码不可被内嵌

5. 单元中的内嵌例程先定义后使用，否则编译器将无法得知此例程的实现代码

6. 含开放数组参数的例程不可内嵌

7. 包中的代码可以内嵌，however，inlining never occurs across package boundaries.

8. 循环引用的单元间不存在 inline，但其中的单元可 inline 循环外的单元中的代码

9. 若某例程在 interface 部分声明但其代码中使用了定义于 implementation 部分的变量，则此例程不可加 inline 关键字

10. 若某内嵌例程使用了其它单元中的代码，则其所引用的单元必须全部在 uses 部分列出，否则此例程将不能内嵌

11. 若在 while-do 和 repeat-until 中使用的条件表达式含有例程，则此处的例程不会被内嵌，但在其它地方使用此例程则可被内嵌。如以下代码：

```
function add(x, y: integer): integer; inline;
begin
  result :=  x + y;
end;

var
  s,i: integer;
begin
  s :=  add(2, 3); // L1
  while s < 100 do
    if s > 78 then
      s :=  add(s, 1); // L2
end.
```

此段代码在编译时，L1 位置的 add 函数将变成 2+3，而 L2 位置的代码将保持不变。

[注：]对于以上的各种情形，编译器并非对每一种都会提出警告

一般情形下，当某单元的 interface 部分发生改变时，所有引用此单元的其它单元需要重新编译。涉及内嵌例程时这个规则稍微有点改变：只要一个内嵌例程的实现代码改变时，其所在的单元就会被重新编译，所有引用此单元的其它单元也将重新编译。

Delphi 提供了{$INLINE}编译开关使得用户可以手动控制编译器对内嵌例程的处理。

此开关有三种状态，各状态的含义如下：

{$INLINE ON}：　此状态下，当某例程后加上 inline 时将会被内嵌入目标代码

{$INLINE AUTO}：后加 inline 的例程被内嵌，若某例程的代码若不大于 32 字节，即使未加 inline　也会被嵌入目标代码

{$INLINE OFF}：　所有的例程均不会被嵌入目标代码，即使例程后加上了 inline

默认状态为{$INLINE ON}。

5.4 例程指针

例程指针是一种特殊的指针类型，其所指的内容并非为一个变量，而是由一段代码所构成的例程。Delphi 的说明文档中将例程指针称为过程类型(Procedural Types)，但笔者习惯上使用例程指针来指代这种类型。本书中将二者看成同义名词。

1. 例程变量的声明

[注：]本节只介绍普通例程指针，暂不介绍对象方法指针(见于第八章)

例程变量的声明与普通变量的声明区别不大，同样有两种方式：

- 先定义变量类型，再声明变量
- 直接声明变量

将一个完整的例程声明式中的名称去掉后剩下的部分即可用于声明例程变量类型或直接用于声明例程变量。如对于函数声明：

```
function AddData(x:integer):integer;
```

去掉名称 AddData 后的部分为：

```
function(x:integer):integer;
```

则可声明一个此例程变量(或例程指针)F 如下：

```
Var
  F:function(x:integer):integer;
```

只有当下列条件全部满足时，编译器才会认为两个例程变量等效：

- 参数的个数、类型、顺序完全一致
- 返回类型完全一致(仅针对函数)
- 声明时使用的关键字相同(全是 procedure 或 function)

换而言之，只有两者一字不差时，编译器才会认为两个例程变量等效。

如下列例程变量中，只有 F1 和 F6 相等：

```
Type
  MyData = integer;
Var
  F1:function(x:integer;y:double):integer;
  F2:function(y:double;x:integer):integer;
  F3:procedure(x:integer;y:double);
  F4:function(y:double;x:integer):string;
  F5:function(x:integer;y:double):MyData;
  F6:function(x:integer;y:double):integer;
```

2. 例程变量的赋值

例程变量赋值有两种种方式：将一个例程变量的值赋予另一个变量，或将一个定义好的例程名称赋给例程变量。如下面的例子：

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
function AddData(x:integer;y:real):integer;
begin
  result :=  2*x;
end;

var
  F,Q:function(x:integer;y:real):integer;
begin
  f :=  AddData;
  q :=  f;
End.
```

3. 例程变量的使用

例程变量的使用用普通函数没有任何区别。使用时我们完全可以将变量名当成函数名。如下面例子：

```
program Project1;
{$APPTYPE CONSOLE}

uses
  SysUtils;
function LinkStr(s:string): string;
begin
  result :=  '参数 S 的值为：'+s;
end;

var
  F:function(s:string): string;
  Str:String;
begin
  F :=  LinkStr;
  Str :=  F('this is Delphi2010');//S 的值为一条字符串
  writeln(LinkStr('this is Delphi2010'));
  writeln(F('this is Delphi2010'));
  readln;
end.
```

我们也可以将例程变量放于赋值表达式的右边，正如上面代码中的变量 S 和 Q。编译器会根据左边的表达式的类型自动赋值。故而在上段代码中，赋给 Q 的是函数地址，而赋给 S 的则是函数返回的字符串。当然，若将 S 的类型改整型，编译器提示错误。因为 LinkStr 返回的字符串不能赋给整型变量。

对于过程变量，无论赋值与否，使用取地址符@均无法取得其地址值，取而代之，应当使用符号@@：

```
program Project1;
{$APPTYPE CONSOLE}

uses
  SysUtils;
procedure LinkStr;
begin
end;

var
  F,Q:procedure;
  v2:String;
begin
  F :=  LinkStr;
  Writeln(integer(@@F));
  Writeln(integer(@F));
  Writeln(integer(@LinkStr));
  readln;
end.
```

运行后可观察到@F 与@LinkStr 的值相等，均表示例程 LinkStr 的地址。而@@F 则表示变量 F 本身的地址。

与大多数变量一样，例程变量的值也可为 nil。调用一个值为 nil 的例程变量将会导致运行期错误。Delphi 提供标准函数 Assigned()来判断一个指针是否与某块内存绑定，若是则证明此指针的值不为 nil，函数返回 True；当 P 的值为 nil 时函数返回 False。

5.5 匿名方法

1. 例程引用

习惯上将 Delphi 中的函数、过程、方法合称为例程。同样，可将函数引用、过程引用及方法引用合称为例程引用。匿名方法与例程引用间我关系非同寻常，故在正式学习匿名方法前必须熟悉例程引用。

声明一个例程引用类型的变量只有一种方式：先声明一个类型，再以此类型声明一个变量。例如，声明例程引用变量 RefFun 的格式如下：

```
Type
  TFun = reference to 本体;
Var
```

```
  RefFun:TFun;
```
其中的本体就是一个省略名称的普通例程声明。如：
```
Type
  TFun = reference to function(x:integer):integer;
```
此声明中包含以下信息：

1.TFun 是一个例程引用类型。

2.TFun 类型的变量所引用的例程是一个函数，此函数接受一个 integer 类型的参数并返回一个 integer 类型的返值。

这有点类似于例程指针：
```
Var
  PFun : function(x:integer):integer; //PFun 指向这样一个函数：接受一个整型参数并返回整型值
```
实际上，例程引用与例程指针除了内部细节有所不同，其它用法极其类似。例如，我们也可以将一个普通例程赋予例程引用类型的变量，如：
```
program RefSample_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  TFun = reference to function(x:integer):integer;

function fun(x:integer):integer;
begin
  writeln(x+100);      //显示计算结果
end;

var
  F:TFun;
begin
  F :=  Fun;     //将函数 Fun 赋予例程引用类型的变量 F
  F(12);        //调用 F
  readln;
end.
```
2. 例程引用类型的默认参数

若在声明例程引用类型时指定了默认参数，则所有此类型的变量中将含有此默认参数。

我们依然以用一个例子来讲述：
```
type
  TF = procedure(s:string = 'default');

procedure echo1(s : string = 'delphi');
begin
```

```
  writeln(s);
end;

var
  F : TF;
begin
  F :=  ECHO1;
  F;       //显示 default
  readln;
end.
```

其中的 F 相当于以下例程：

```
procedure F(s : string = 'delphi');
```

读者可能发现了：在调用 F 前已经将 echo1 作为变量值赋给 F，但调用 F 时 F 却并未采用 echo1 中的默认值而是采用了 TF 中的默认值。这正是 Delphi 中默认参数的使用规定之一：当某个例程类型中指定了默认参数时，这些默认参数将掩盖具体函数中的默认值。也就是说，例程类型 TF 中指定的默认值会掩盖具体例程 echo1 中的默认值。据此，若 TF 中未指定默认参数，无论 F 的值如何，调用时必须指定参数。

3. 匿名方法

所谓匿名方法是指一个未指定名称的例程。

类似于字符构成字符串直接常量，匿名方法是由一段代码所构成的直接常量。可以将这种常量作为值赋予函数引用类型的变量。同理，普通的例程相当于一个由代码构成的符号常量，只是这种常量能够执行特定的计算任务而已。

上一节，我们声明了一个函数引用类型的变量：

```
var
  F : TFun;
```

给 F 赋值时，除了将一个例程赋作为值赋予其之外，还可以将一个匿名方法赋给它。所谓匿名方法，就是一个没有名称的例程。

通常，可按下列方式定义一个普通的例程：

```
function myfun(x:integer):integer;
Begin
  //具体代码
End;
```

当我们将其名称省去时，这段代码就表示一个简单的匿名方法：

```
function (x:integer):integer;
Begin
  //具体代码
End;
```

与普通例程不同，匿名方法不存在“声明”这一说法，因为其本身就是一个直接常量值，只不过其值是一段能够进行运算的代码而并非一个确定的具体值。类似于将字串值赋予字符串变量，我们可以将一个匿名方法赋给例程引用类型的变量：

```
RefFun :=  function(x:integer):integer //此处结尾不可有分号，因为语句并未完结
            Begin
              //具体代码
             End;
```

匿名方法的调用相当简单：将其赋予一个例程引用类型的变量，然后按调用普通例程的方式调用此变量即可。如下列代码：

```
program Sample_2;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  TFun = reference to procedure(x:string);  //声明例程引用类型 TFun
var
  F:TFun;   //声明 TFun 类型的变量 F
begin
  F :=  procedure(x:string)   //将一个匿名方法赋予变量 F
       begin
         writeln(x);
       end;
  F('Delphi2010');     //调用 F 所指向的匿名方法
  readln;
End.
```

4. 匿名方法的使用

老实说，匿名方法只是一种语言上的障眼法。使用匿名方法可能会使代码的编写更加容易，但确并未夫语言增加任何新的功能。正做菜时用的味精，并不能增加任何营养，纯粹只是为了获得更好的口感。

就使用角度而言，Delphi 中的匿名方法完全可以被例程指针所取代。鉴于此，此节我们只讨论匿名方法作为一种常量时的使用问题。其它用法可参见例程指针的相关章节。

正如前面的章节所述，匿名方法是一种由可运行代码构成的特殊的直接常量。与普通直接常量不同，匿名方法被存储在堆中，其生存期管理受引用计数控制。

匿名方法不能作为常量值用于声明一个符号常量或于变量声明时作为变量值而初始化此变量，诸如以下形式的语句均不合法：

```
Const
  C = {某个匿名方法};
Var
  V: {例程引用类型} = {某个匿名方法};
```

匿名方法也可作为例程的参数。需要注意的是，由于匿名方法是一种直接常量，故其作为参数只能用 const 或 value 方式传递，不能使用 var 或 out 方式。如下面的程序：

```
program RefSample_3;
{$APPTYPE CONSOLE}
uses
  SysUtils;
```

```
type
  TFun = reference to procedure(s:string);

procedure Fa(s:string; Fun:TFun);
begin
  Fun(s);
end;

var
  F:TFun;
begin
  Fa('Delphi2010',procedure(s:string)
                    begin
                      writeln(s);
                    end);    //end 后不可加分号，为什么？
  readln;
End.
```

5. 变量劫持

我们已经学习过了匿名方法的声明、使用，对于普通的直接常量而言，这已经足够。但对于匿名方法这种特殊的直接常量而言，还差最后一点。

我们从一段代码开始：

```
program RefSample_4;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  TFun = reference to function(x: Integer): Integer;

function ReturnFun(y: Integer): TFun;
begin
  Result :=  function(x: Integer):integer
             begin
               Result :=  x + y;    //y 被匿名方法使用
             end;
end;

var
  fun: TFun;
begin
  fun :=  ReturnFun(20);  //语句 1
  Writeln(fun(22)); // 显示 42
  readln;
```

```
end.
```

不难发现，语句 1 将 ReturnFun 的参数 y 的值指定为 20，此后 fun 相当于以下匿名函数：

```
function(x:integer):integer;  //为方便起见，我们以标识符 F 代表此匿名函数
Begin
  Result :=  x+20;     //本当是 x+y,但此时 y 的值固定不变，为 20
End;
```

接下来的代码中，除非重新赋值给 Fun 或变量 Fun 不再有效(超出了 Fun 的有效域)，否则 y 的值将一直等于 20。

请读者思考，为什么 y 的值会保持在 20 而不是被销毁？

可能读者不明白这个问题的意思，不要紧，我们来展开一下。根据计算机的运行规律，当程序执行至语句 1 时，程序会依次执行如下操作：

1. 保存所有状态并离开当前区域而进入函数 ReturnFun 所在的区域以便运行此函数
2. 运行函数 ReturnFun 并获得一个 TFun 类型的值(即 ReturnFun 的返回值)
3. 携带 ReturnFun 的返回值回到原来的位置并将所携之值赋予变量 fun

经过以上步骤，fun 获得了一个值——匿名函数 F，且 F 中例用的 y 的值为 20。问题就在于：当程序的执行点离开 ReturnFun 所在的区域时，ReturnFun 中的所有局部变量及参数应当被销毁，而 y 作为 ReturnFun 的参数应当也不例外。如此，当程序携带 ReturnFun 的返回值回到原来的位置时，y 的值早就不存在。但现实是 y 的值保持 20 不变。这提示了一个重要信息：ReturnFun 中的某些局部变量并未在执行点退出 ReturnFun 所在区域时被销毁。

由此，我们得到一个结论：当某个局部变量被在其所在例程中声明的匿名方法使用时，此变量的生存期将受匿名方法的控制而超出原本的生存期。

以上述的 RefSample_4 为例，由于 ReturnFun 的参数 y 被其中的匿名方法(即前面所说的的 F)所使用，y 的生存期将变得不再是原来的生存期[注 1]，而是受匿名方法 F 影响。确切的讲，y 的生存期变得与 F 相等，只要 F 有效，y 就永远有效，即使超出了 ReturnFun 执行期。

以上情形下，我们称匿名方法 F 劫持了变量 y(参数也是一种特殊的局部变量),或变量 y 被 F 俘获。

注意：匿名方法只能俘获其父例程中定义的局部变量。例如若在例程 F 中定义匿名方法 AR，则 AR 只能俘　获(或绑架)声明于 F 中的局部变量。至于 AR 为何不能俘获其它例程中的局部变量或 F 中的其它变　　量，请读者自行思考

编译代码时，若编译器发现某个例程定义了一个匿名方法，其会自动产生一个与此例程相关联的数据结构，称之为框架对象（frame object，**以下简称 FO**），FO 其可用于存放变量及匿名方法。

当编译器发现例程 F1 中的某个变量 V 被例程 F2 俘获时，会将变量 V 记录在与 F1 相关联的 FO_1 上。同时，编译器也为 F2 创建一个框架对象 FO_2，并创建一个由 FO_2 指向 FO_1 的引用。若 V 同时被多个匿名方法俘获，系统将会为每个匿名方法创建一个 FO，及一个指向 FO_1 的引用。

在系统运行时，若出现以下任何一种情形时，引用会被销毁：

- 匿名方法超出的生存域期而被销毁。此时其所绑架的所有变量将被同时销毁。
- 例程引用类型的变量的值发生了改变，不再是原来的匿名方法。
- 匿名方法被析构

若所有指向 F0_1 的引用均已被销毁，则系统会销毁变量 V。更确切的说是 F0_1 及其所关联的匿名方法被销毁(匿名方法是由代码构成的直接常量，存储在堆中)，所有被俘获的变量全部随之销毁。

读者需注意两个问题：

- F1 可能是普通例程或匿名方法，但 F2 一定为匿名方法。原因：只有匿名方法才可能俘获变量。
- F2 一定是定义在 F1 当中。原因：F2 能使用 F1 中的局部变量。

[注 1]：y 原来的生存期为 ReturnFun 函数体执行期间，也就是说只有在函数 ReturnFun 执行期间 y 才　　有意义

5.6 重载例程

假设我们需要编写函数来查找计算机上的某个文件。为了达到这个目地，我们可以针对文件每条信息定义一个 FindByXXX 形式的例程，然后使用时分别调用。如利用文件的名称来查找，可以定义函数：

```
procedure FindByName(name:string);
```

同样，若使用文件的大小来查找文件则可以定义如下过程：

```
procedure FindBySize(size:integer);
```

通过文件的创建日期来查找：

```
procedure FindByDate(date:TDate);
```

……

这样当然可以达到目的。但用户在查找文件时，对于每个信息均必须知道对应的查找例程的名称，否则可出现诸如将文件大小传递入 FindByName 来查找文件的之类错误。

我们可以同时定义多个名为 Find 例程，每个例程接受不同的参数，使用时系统根据参数自动选择合适的例程。如定义以下例程：

```
procedure Find(name:string);overload;
procedure Find(size:integer);overload;
procedure Find(time:TDate);overload;
```

……

当我们以文件名作为参数调用 Find：

```
Find('d:\file\dest.txt');
```

系统会根据参数的类型自动判断出此处的 Find 是指 procedure Find(name:string)而非其它的 Find 例程。

在此例子中，我们实际已经重载了 Find 例程。“重载”中的“重”并非“重新”而是“多重”之意。所谓“重载”是指同一个有效范围内某个例程名称同时对应多个不同的例程的定义，这相当于同一有效域中的多个不同的例程恰巧拥有了相同的名称。

不同于普通变量，系统在识别一个函数或过程时并非仅仅根据其名称，而是根据“名称+参数个数+每个参数的数据类型”的组合来识别一个例程，此组合称为例程的特征集。当例程的名称相同时其特征集不一定相同，所以同一个例程名称可以代表不同的例程。注意：同一名称对应的多个例程必须全是过程或全是函数。

声明一个重载例程只需在声明时加上限定符 overload(定义例程时可省略限定符)。假设我们需要声明若干个 Find 例程，则每个 Find 例程(包括第一个)后必须加上 overload 以标识此例程是一个重载的例程。注意：例程的特征集不包括参数的传递方式，即以下的例

程重载将不会通过编译：

```
procedure find(const c:string);overload;
procedure find(var c:string);overload;
```

现在来看一下系统运行时是如何确定目标例程的。假设我们以文件大小和文件的创建时间作为参数查找一个文件，我们会作如下调用：

```
Find(size:integer; time:TDate);
```

系统将按以下步骤确定合适的 Find 例程：

1. 确定例程的名称：Find，系统会列出所有可用的且名称为 Find 的例程，这些例程的集合称之为候选集。
2. 确定调用时所使用的参数的数目及各自的类型，以上的调用中有两个参数，且第 1 个参数为 integer 类型，第 2 个参数为 TDate 类型。
3. 在候选集中寻找接受两个参数且第 1 个参数为 integer 类型，第 2 个参数为 TDate 类型的例程。
4. 调用所寻找到的例程

以上的步骤并非总是一帆风顺。默认参数及未指定数据类型的实参会影响重载例程的选择。如对于以下候选集：

```
procedure fun(var s:string; var size:integer = 1024);overload;   //R1
procedure fun(var s:string);overload;    //R2
procedure fun(var size:cardinal);overload;  //R3
procedure fun(var size:integer);overload;//R4
```

若有调用：fun('d:\file\dest.txt');编译器如何确定调用的是 R1 还是 R2？同样，对于 fun(1024);编译器如何确定调用的是 R3 还是 R4？

我们分别说明。首先，对于默认参数，Delphi 并未提供有效的方法，故而使用时应当注意不要在被重载的例程中定义默认参数。类似地，被重载的例程在定义时必须与声明式有完全相同的参数列表。

其次，若对于某个实参，候选集中有多个例程均符合，系统运行时将根据参数的数据类型选择值域最小者。如上面的 R3 与 R4 中，R4 的参数为 integer，其值域较 cardinal 小，故对于 fun(1024)系统将选择 R4。此时存在两种特殊情况：

- 若调用时传给例程的是一个带小数点的实数，则系统会直接将其转换成 Extended 类型。如对于调用 fun(3.4)，系统会将 3.4 看成 Extended 类型并在 fun 的候选集寻找接受 Extended 类型参数的例程，若寻找未果，系统将不会做另外的转换而是直接提示错误(除非有接受变体类型参数的例程)。
- 若传给例程的实参是未指定具体数据类型的字符、字符串或字符指针(包括值为三者之一的符号常量和直接常量)，系统会分别转换成 char、UnicodeString 及 pchar 类型。

根据以往的经验，凡是涉及到数据类型，变体类型一定是特殊的。此处亦然。

Delphi 规定：变体的优先级低于所有简单类型。根据此规则，系统先按上述一般规则进行配对，当无法确定目标例程时，系统会调用将参数看成是优先级最低的变体类型而调用接受变体参数的例程。下面的例子展示了这一规则。

```
procedure fun(s:byte);overload;
procedure fun(s:variant);overload;
```

对于调用：fun(256)，系统首先确定 256 是整数常量。整型类型包括多种类型，按值域从小到大排序为：

shortint < byte < smallint < word < integer < cardinal < int64 < uint64

根据值域排除 shortint 与 byte，由于 256 可能是剩下的 6 种类型中的任意一种，故而系统将按值域由小到大进行筛选：

1) 候选集中是否有例程接受 smallint 类型的参数？若是则调用此例程，若否则进行下一步
2) 候选集中是否有例程接受 word 类型的参数？若是则调用此例程，若否则进行下一步

……

3) 候选集中是否有例程接受 uint64 类型的参数？若是则调用调用此例程，若否，由于不存在比 uint64 更大的整数类型，系统会在候选集中寻找是否有例程接受 variant 类型的参数，若是则调用此例程，若否则引发编译错误。

通过以上步骤，最终系统确定目标例程为 procedure fun(s:variant)。

第八章 类与对象

本章开始，我们将进入面对象部分。读者可能是第一次听接触到这部分的内容，也可能因为某个机缘而学习过相关的知识，但不管怎样，笔者希望您在接触本章的内容时能够忘记以前所看过的那些内容，除非您已经精通了类和对象的概念。

6.1 面向对象初步

1. 现实世界中的对象？

任何事物皆可看成是一个对象。以哲学的观点而言，对象更多的是一种唯心而非唯物的概念 ，我们可以将一辆汽车看成是一个对象，也可以只将这辆车的某个车轮看成是一个对象，甚至可以将构成车轮的分子看成对象。只要方便我们解决问题，万物均可以看成对象。这个概念在计算机世界同样适用，但计算机世界中不存在无形与有形之分，其中的任何对象都会占用一定存储空间，小到一个常量、一个变量、大到一个函数甚至是一个完整的的程序，我们都可能将其看成是一个对象。

2. 在计算机中克隆现实世界的对象

现实中的任何对象，我们都能使用一些属性来描述这些对象，如重量、大小、颜色等。就算是无形之物如磁场、热量等，我们也可以用诸如温度、强度等属性来描述。

通常我们描述一个人时会使用这个人的各种信息如年龄、职业、性别等，通过这些描述，我们可以在未见到这个人的时候就能了解他的大概情况，而这个过程的实质是我们的大脑利用这些信息在我们的脑中构建了一个具有这些信息的虚拟人。同样，我们也可以在计算机中构造虚拟人。现在，我们在计算机中声明一个结构类型 TMan，其中的三个成员 name、age、sex 分别表示某个人的改名、年龄、性别三个信息：

```
type
  TMan = record
```

```
    name : string;
    age : integer;
    sex : string;
  end;
```

无可否认，所有人都具有这三种信息，所以我们可以使用 TMan 类型的变量去表示任意一个人。以下声明了 TMan 类型的变量 SomeBdoy 并赋值：

```
var
  SomeBody : TMan;
......
  SomeBody.name :=  'Mill';
  SomeBody.age :=  30;
  SomeBody.sex :=  '男';
```

这样一来，我们可以在计算机中用变量 SomeBody 来代表一个名为 Mill 的 30 岁的男性。

虽然我们不能把 Mill 塞进送往计算机世界，但取而代之我们可以在计算机以这些属性来构造出了一个虚拟的 Mill，读者完全可以把这个虚拟的 Mill 想像成是真实的 Mill 通过某种途径到达了计算机世界。

我们更进一步。现实中的人只要还活着就肯定能够执行一些行为（就算是植物人也会执行“呼吸”这项行为），所以若想将 SomeBody 变成 Mill 在计算机世界中的翻版，就必须让 SomeBody 能够执行能有 Mill 能够执行的行为。通过将例程作为变量的成员我们可以赋予 SomeBody 这项能力：

```
type
  TMan = record
    name: string;
    age: integer;
    sex: string;
    procedure Clean(var place: string); // place 表示何处需要打扫
  end;
```

但同时应当认识到，二者执行的结果肯定不一样。就像同样是踩下油门，飞机会往上飞，而汽车却往前跑一样，相同的行为在不同的世界肯定具有不同的表现。所以现实的 Mill 打扫卫生时会拿起工具将垃圾清扫一空，而 SomeBody 在打扫卫生时绝对不能打扫现实世界中的垃圾。至于它会做什么则由例程的编码决定。为了方便我们采用如下编码，当然读者可以使用任意的合法编码：

```
procedure TMan.Clean(var place: string);
begin
  showmessage(place + '已打扫完毕');
end;
```

当我们让 SomeBody 打扫客厅时，可以调用 SomeBody.Clean('客厅')，它会显示如下对话框：

通过这种方法我们可以真实地在计算机中以计算机世界的方式克隆任何现实世界的对象。

3. 面向对象编程中的对象

可以发现，要在计算机中克隆一个现实世界对象，我们需要使用类似于 record 的结构类型的变量，其成员可以是变量或例程，变量成员用于描述现实对象的相关信息如年龄、大小，而例程成员则用于模拟真实对象能够执行的某些行为。（当然变量可以只含有变量成员或只含有例程成员，甚至两者都没有。这不奇怪，就算你的档案也不会把你所有的信息都写进去）

在面向对象的编程体系中我们把这种可容纳变量与例程作为成员的结构类型的变量称之为对象。So- meBody 就是一个对象。

如无特殊说明，本书从现在开始所出现的对象一律为此含义。

4. 面向对象的编程方式

在开始本节前，我们先定义一个概念：基本代码。按照结构化设计方法，我们可能在 A 中调用 B，在 B 函数中再调用 C 函数，一直下去，但总有一个函数 X 中没有调用其它函数。我们将调用函数 X 中的代码（注意不是函数）称为基本代码。X 中的每句代码均为基本代码。

从结构上而言，基本代码有点类似于现实世界中组成物质的最基本粒子。所有的程序都由若干句基本代码组合而成。

计算机开始设计程序时，每次设计人员总是从头开始运用一句句的基本代码来组合。后来，人们逐渐发现，很多程序中都会用到一些相同的代码，于是将这些相同的代码抽出来组合成函数，使用时只需调用这些函数即可。很明显这种方法大大节省了时间。于是人们将这种方法加以完善形成了一种完整的理论，称为结构化程度设计。其核心思想在于将代码组合成函数，再将函数与基本代码组合形成更高一级的函数...，从而提供更为复杂的功能。

很明显，通过这种方法，有时为了完成一个非常复杂的功能，我们可能要从基本代码一层一层往上组合很多级函数。如此，万一程序出了问题，我们也需要从最低级的函数开始逐级查找问题的来源。这相当于医生诊断时要从组成人体的分子开始一点点的排除问题，可以想像其效率有多低。（就算后来出现了函数库，情形也未能好转。）

除此之外，这种方式写出的代码还比较混乱：假设一个程序调用了函数 FA，而 FA 又调用了 FB，FB 又调用了 FC...，对于代码的作者还稍微好一点。但万一这位仁兄因某个原因实然消失了，那么其他的人在阅读这份代码势必会很郁闷：当他想看 FA 的功能时，要先明白 FB 的功能，而明白 FB 的功能之前，又要先明白 FC 的功能...，等他好不容易看到了 FX 时，却又忘了 FA 的作用。这样看了后面又忘了前面，首尾不能兼顾。

所以说利用这种方式编写较大的软件时很难保证开发速度及开发质量，而当今时代的快节奏又要求在短期内开发出高质量的软件，要命的是这些软件还往往都比较复杂。

面对对象的编程方法可以很好解决这个问题。与结构化设计不同，面向对象的编程方法在实现时会在计算机中创造多个对象，让每个对象来完成特定的功能，然后将这些对象

组合以完成复杂功能。以计算机为例，我们不需要在亲自去敲击键盘键入文字，只需告诉键盘这个对象：喂，把我读的这句话记下来。当你念完任意一句句话后，计算机的键盘会自动将你所读的话输入计算机中。此时它不再是一个无生命的只能用于输入字符的器械，而是一全活的生命体。它的功能也可能超出传统之外。例如我们可以命令它将我们所说的内容输入硬盘上的某个文本文件中或是将这些内容翻译成其它文字后再输入某个文件。

所以说，面向对象实质就是在计算机中创造对友象并通过函数赋予其各种能力以实现对象的自动化。使用时我们通过驱使这些自动化的对象来完成特定的功能。例如当我们编辑一篇文档时，我们可以在计算机中创建一个 article 对象并使之管理这份文档：

```
type
  TDoc = Record
    filename: string;  //表示文档的文件名
    procedure save(var filename:string);
    procedure open(var filename:string);
    procedure insert(var text:string);
  End;

var
  Article:TDoc;
```

当我们需要执行某些操作如向其中插入一段文字时，我们可以直接调用其中的函数实现这项操作：

```
Article.insert('delphi');
```

问题：我们可以声明一个单独的函数：

```
procedrue insert(var filename, text : string);
```

通过这个函数我们同样可以向某个文件中插入一段文字，既是如此，为何那么麻烦的使用对象来操作？

答：面向对象编程方式优于结构化编程方式并非因为存在某些前者能实现而后者不能实现的功能。面向对象编程方式只是一件漂亮的外衣，它使得代码的阅读起来更加的清晰直观，维护时更加的快速，但人不可能因为换了件漂亮衣服就变得能上天入地，所以面向对象编程方式并没有程序语言增加新功能。利用面向对象能编写的程序，一定也能用结构化设计编写，差别只能代码没那么直观而已。以上面的例子来说，Aticle 对象中含有有三个函数，若使用结构化编程我们就必须编写三个独立的函数。很明显管理一个对象比管理三个函数要好得多。正如一包糖果有 20 粒，抓一整包糖果，亦或两手捧 20 粒散装的糖果，读者觉得哪个更方便？

6.2 类与对象

6.2.1 声明一个 class 类型

通过前面的内容，我们可知模拟一个对象需要两样东西：一是符合条件的结构化数据类型（例如前面使用的 record 类型）；二是这种类型的变量（对应于前面的 SomeBody）。

在支持面向对象编程的计算机语言如 Delphi、C#当中，统一将这种数据类型称为 class，中文翻译成“类”。下面的内容中我们把类等同于类类型。如声明一个类意味着声明一个新的 class 类型。很明显 class 类型的变量是一个对象。Delphi 中声明一个类及对象与普通的类型与变量声明无二致：

```
type
  MyClass = class
    //声明成员
  end;

var
  Obj:MyClass;
```

以上我们声明了一个类类型 MyClass，简称类。为了方便我们并没有在 MyClass 中添加任何的成员，读者可自行添加。

注意：类不能声明为局部类型，即类不能在函数或过程中声明。与此相似的还有 Interface 与 Object。

类中可以有多种声明，所有在单元中能够声明的元素在类中均能声明，如可以在类中利用 const 声明常量，利用 var 声明对象变量，利用 type 声明数据类型等。通常我们把类中声明的变量称为字段（Field），将类中声明的例程称为方法（Method）。

6.2.2 创建一个对象

类相当于对象的一个设计图，直接使用类并不能模拟现实的对象，只有计算机世界的对象才能模拟现实的对象。所以在使用对象前，我们需要根据设计图来建造一个对象。

Delphi 中使用类中隐含的构造函数来创建一个新的对象，其语句格式如下：

```
对象名称 :=  类名.create;
```

如下所示：

```
type
  T1 = class
    s:integer;
  end;
var
  obj:T1;
begin
  obj :=  T1.create; //若无此句代码将会导致运行错误，但可以正常编译
  obj.s :=  239;
end.
```

6.2.3 套嵌类的声明

在类的声明中除了声明任意一个变量或例程作为成员，我们还可以声明一种新的数据类型，包括另一个类。我们将在另一个类中声明的类称为套嵌类。接下来我们在 MyClass 中声明一个套嵌类 NestClass，它只能用于声明 MyClass 中其它的变量成员或例程成员的参数：

```
type
  MyClass = class
  type
    NestClass = class
   end;
  var
    NestObj:NestClass;
    procedure ShowName(var s:NestClass);
  end;

var
  procedure OtherFun(var s:NestClass);  //NestClass 只能用于 MyClass 当中
```

有效域规则在此处同样适用：NestClass 只能用于 MyClass 的成员，不能用于类之外的任何变量或例程中。故而当我们试图将 NestClass 用于一个 MyClass 之外的例程时编译器会提示错误。如 NestClass 就是 MyClass 的一个套嵌类。

当然，类中可声明任何数据类型如记录、数组等。关于类中声明的数据类型有以下规则需要遵守：

1. 类中声明的数据类型只能用于这个类当中。如 NestClass 不能用于 OtherFun 当中。
2. 与普通声明一样，类中的数据类型只能先声明，后使用。
3. 类中声明的标识符只能被本类及本类的套嵌类使用，但一个类却并不能使用套嵌类中声明的标识符。

我们来详细地看一下第 3 条规则。以下声明了一个存在着多重套嵌的类 T1：

```
type
  T1 = class
  type
    m2a = class
    type
      m3a = class
      end;
    end;

    m2b = class
    type
      m3b = class
      end;
    end;
  end;
```

根据上述的第 3 条规则，m3b 可以使用 m2b 与 T1 中声明的所有标识符，m2b 或 m2a 可使用 T1 中声明的所有标识符。m2a 不能使用 m3a 中声明的标识符，非但如此，m2a 还不能使用 m2b 中声明的标识符。关于此，有一个很形象的例子：

我们将 T1 看成是一个国家，按照套嵌关系，可以依次将 m2a、m2b 看成是这个国家中的两个省，而 m3a、m3b 又分别是这两个省中的两个市。很明显：国家的物资每个省、每个市、每个县均可使用。所以 T1 中声明的标识符可以被 m2b、m2a 所用，也可被 m3a、m3b 所用。然而某个省的物资却只能被本省的每个市每个县不能被其它省的市县使用，所以 m2a

中声明的标识符不能被 m2b 所用。按照这种比喻，我们也很容易解释为什么类中声明的标识符不能用于类外：国内的法律对国外当然没用。

注意：上述的规定也适用于除类之外的任何其它的结构类型如记录、数组等。

6.2.4 类的继承

顾名思义，类的继承是指通过某种方式可使一个类（子类）继承另一个类（父类）的字段与方法。Delphi 中实现类的继承只需在声明一个类时指定其父类，我们将此称为从父类的派生了一个子类。（父类与父类以上的类如爷爷类、太爷爷类等合称为祖先类）

```
type
  子类名称 ＝ class（父类名称）
  end;
```

如下面代码声明的 Tb 继承于 Ta：

```
type
  Ta = class(TObject)
    Fa:integer;
  end;

  Tb = class(Ta)
    Fb:integer;
  end;
```

Tb 继承于 Ta，其自动包括了 Ta 中所有的成员。下图展示了这种继承关系：

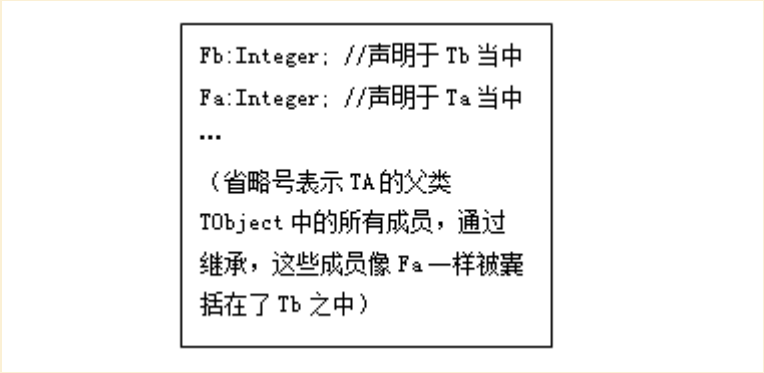


图 1 Tb 的内部结构(示意图)

注意：Tb 继承了 Ta 相当于 Tb 中含有了 Ta，但并不是真的含有。这点相当于古代的战争：A 国占领了 B 国，A 国就拥有了原本属于 B 国的一切，包括军队、人民、粮食、财富等，但这并不包括 B 国这个国家本身。

在 Delphi 中任何类在声明时均必须指定其父类。若代码中未显式指定，Delphi 会自动认为此类继承于 TObject。故而下面声明的 TA 与 T2 完全等同（不是“相当于”）：

```
type
  TA = class
  end;

  T2 = class(TObject)
  end;
```

在 Delphi 中当类可被提前引用，其方法类似于提前声明一个例程：

```
type
  TA = class;
```

但若在声明时加上了父类的名称则 IDE 认为是在定义而非提前声明一个类：

```
type
  TA = class(TObject);
```

这种写法等同于

```
type
  TA = class(TObject);  //也可直接写成 TA = class
  end;
```

在某些情况下，我们可能不希望某个类派生出子类，此时可使用关键词 sealed 替类做一个“绝育”手术。sealed 的用法为：

```
type
  类名称 ＝ class sealed (父类名称)
  end;
```

例如：

```
type
  TA = class sealed (TObject)
  end;
```

TA 将无法派生任何子类。

6.2.5 类成员的访问权限

一个人可能有很多朋友，根据亲密程度，这些朋友对这个人的了解程度也不一样。通常那种客户关系的朋友可能连这位仁兄住哪都不知道，而恋人之间可能连对方祖宗八代的血型是 A 型还是 B 型都知道。

作为对现实世界的模拟，对象如实模拟了这种关系。

我们可将对象的成员看成是对象的私有信息，这些信息有些较为重要，只能被一些关系较为亲密的对象所访问，而有些信息如名称却无关紧要，可以被所有的其它对象所访问。通过在类的声明中添加一些关键词来实现这种效果。

Delphi 共提供了 6 个关键词来用于限定访问权限：

```
public、private、protected、published、automated
strict private　　、　　strict protected
```

使用的格式为：

```
type
  类名称 = class(父类名称)
  private
    //private 区间
  protected
    //protected 区间
  ...
  end;
```

private 至其后的另一个关键词之间为 private 区间，protected 至其后的另一个关键词之间为 protected 区间...，依次类推。所有关键词均为可选，读者可全部使用，也可全部不使用或只使用其中的若干词。各词间无先后顺序。其各自的含义为：

1. strict private： 此区定义的字段或方法只能用于当前的类中。即 T1 中此区定义的成员只能在 T1 中使用。
2. strict protected：此区定义的成员除能用于当前类中，还可用于当前类的任何子类中。

以上两种成员，同一个类的不同对象间也不能互访问。

3. private：所限定的成员只能用于定义这个类的.pas 文件或.dpr 文件。
4. protected：相当于将 private 扩展到任意子类均可见。
5. public：对任何位置、任何类均可见。当某些成员未显式指定其访问权限时，Delphi 默认为此 public。
6. published：访问权限与 public 相同。
7. automated：用于 Win32 下的 COM 编程，在此不做介绍。

注意：strict private 与 strict protected 从 Delphi2007 开始引入，之前的版本不存在这两种 等级

在 published 区声明属性时应注意以下两点：

- published 属性的值只能是：有序类型、字符串、类、接口、变体、方法指针以及上下界在 0 到 31 之间的集合类型。实数类型不能是 Real48 类型。
- 不能以同一个名字公布两个或更多的重载方法。

6.3 对象字段及对象函数

在声明类类型时可以在其中声明字段与方法。按照所有者的不同，类中声明的字段和方法可分为两种。第一种成员可直接通过类访问，也可通过对象访问，这种成员称为类字段或类方法；第二种只能通过对象而不能通过类来访问，称为对象字段或对象方法。下面我们来定义一个类 T1，其中声明了一个类成员 i 以及一个对象成员 s。读者可看到，通过对象 obj 可同时访问 i 及 s，而通过 T1 只能访问 i，不能访问 s：

```
type
  T1 = class
  class var
    i:integer;
  var
    s:string;
  end;
var
  obj:T1;
begin
  obj :=  T1.Create;
  obj.i :=  90;
  obj.s :=  'this is obj';
  T1.i :=  100;
```

```
  T1.s :=  'this is T1';  //错误，s 只能通过对象 obj 来访问
end.
```

本节内容，我们重点介绍对象成员。按照数据类型的不同，对象成员可分为对象字段与对象方法。前者包括对象中的变量、常量等；后者则包括对象中的函数和过程。

我们先从对象字段开始。

6.3.1 对象字段

对象中的字段包括声明于对象中声明的变量、常量、数据类型。在类中声明对象成员与在单元中声明普通的标识符一模一样。例如，变量均以关键词 var 声明，常量使用 const，而数据类型则使用 type。类中于 var 之后声明的所有的字段将被认为是对象变量，除非遇到了以下三种情形：

1. 遇到其它的用于声明的关键词，如 const 或 type。
2. 遇到了访问权限的限定词，如 public、strict private 等。
3. 遇到了方法的声明

当在类中声明一个字段而未使用任何关键词时，编译器默认为对象变量。如下面声明的 T1 中的 i 就是一个对象变量字段：

```
type
  T1 = class
    i:Integer;
  end;
```

对象字段与对象静态绑定。所谓静态绑定是指在使用对象名称引用对象中的成员时，不管其实际类型如何，编译器都会以声明时的类型为准而调用相应的成员。例如以下代码：

```
type
  T1 = class
    i:integer;
  end;
  T2 = class(T1)
    s:string;
  end;
var
  O1:T1;
  O2:T2;
begin
  o1 :=  T2.Create;
  o2 :=  T2.Create;
  o1.i :=  90;
  o1.s :=  'this is O1';  //错误，o1 不能引用 s
  o2.i :=  100;
  o2.s :=  'this is O1';
end.
```

以上代码中，对于 T1 类型的对象 O1，虽然在赋值时我们将一个 T2 类型的对象赋给了 O1，但纵使如此，通过 O1 仍然无法引用 T2 中的成员 s。因为对象成员与对象名静态绑定，通过 O1 永远只能引用 T1 中的成员。

6.3.2 对象方法

对象方法的的声明与普通例程的声明区别不大。相对于普通的例程，方法后可接更多的限定词。但同一方法后需要接多个限定词时应当遵守一定的顺序。下表中按出现位置的先后列出了一些较为常用的限定词，多个限定词联用时应遵守此顺序。同一单元格的限定词不可能同时出现。

reintroduce
overload
virtual、dynamic、override
register、pascal、cdecl、stdcall、safecall
abstract

在类中声明了方法之后应在定义这个类的源文件中赋予这些方法以具体的定义。通常我们把类中的方法声明称为方法首部，而将方法具体的实现代码称为方法体。

在单元文件中，类可声明于任何地方，但不能声明为局部类型，即不能声明于函数或程序内部。对于在类中声明的方法，其具体的定义代码只能位于单元的 implementation 部分。

在工程文件中，类作为全局数据类型可声明于任何地方，但在声明后应当立即定义，除非使用了提前声明。

在 Delphi 完成了一个类的声明之后，只需将光标置于其中的任何位置并按下组合键 Ctrl+Shift+C，Delphi 就会自动产生类方法定义的的骨架，用户只需在其中添加代码即可。

按照首部与方法体的对应时间，可将方法分为 4 种：静态方法、消息方法、虚方法、动态方法。

1. 静态方法

方法后未加任何限定词时即为静态方法。此类型方法在编译时确定其所对应的方法体（即具体的实现代码）。本书到目前为止所举事例中的方法全是静态方法。

我们声明以下两个类，其中均含有 F1，然后再声明 T1 及 T2 的对象 O1、O2：

```
type
  T1 = class
    procedure F1;
  end;
  T2 = class(T1)
    procedure F1;
  end;
```

```
procedure T1.F1;
begin
  writeln('this is T1.F1');
end;

procedure T2.F1;
begin
  writeln('this is T2.F1');
end;

var
  O1:T1;
  O2:T2;
begin
  O1 :=  T1.Create;
  O2 :=  T2.Create;
  O1.F1;      //调用 T1.F1
  O2.F1;      //调用 T2.F1
  O1 :=  O2;
  O1.F1;      //调用 T1.F1
end.
```

与对象字段类似，通过 O1 我们永远只能调用 T1 中的静态方法，同样通过 O2 我们也只能调用 T2 中的静态方法，面向对象理论中将这种情形称之为方法的调用与方法实体之间的静态绑定，简称静态绑定。我们顺便定义一下绑定的概念：所谓绑定是指方法的调用与具体的方法代码（即方法实体）之间的对应。在这个例子中，当我们调用 O1.F1 时，系统就会自动执行代码：

```
   writeln('this is T1.F1');
```

换句话说，系统可根据方法的调用（在这里我们通过 O1 来调用 F1）而推断出应该执行的方法体（T1.F1），这就是绑定的作用。

2. 虚方法

在前面我们提到过静态绑定，有静态绑定自然也就有动态绑定，所谓的动态是指在根据调用形式的不同，系统会对同一名称的方法调用到不同的方法实体。最常见的情形是：通过不同的对象调用相同类中的某个方法时会产生不同的结果，如下例所示：

```
type
  T1 = class
    procedure F1;virtual; //变化 1
  end;
  T2 = class(T1)
    procedure F1;override;  //变化 2
  end;

procedure T1.F1;
```

```
begin
  writeln('this is T1.F1');
end;

procedure T2.F1;
begin
  writeln('this is T2.F1');
end;

var
  O1:T1;
  O2:T2;
begin
  O1 :=  T1.Create;
  O2 :=  T2.Create;
  O1.F1;      //调用 T1.F1
  O2.F1;      //调用 T2.F1
  O1 :=  O2;
  O1.F1;      //变化三，调用的不是 T1.F1，而是 T2.F1
  readln;
end.
```

与上一个例子相比，这个例子中仅有两处改动，我们特地使用注释来标识这两处的变化。托福于这两处变化，这个例子的运行结果比上一个例子也有了一点不同：最后一句调用 O1.F1 调用的不再是 T1.F1，而是 T2.F2。下面的内容中我们会慢慢来提示其中的奥秘。

通过观察可以了现前两处的变化（即变化一和变化二）均只是在方法后面加上了一个限定词（virtual 和 override），这两个限定词正是我们所要阐述的重点。

关键词 virtual 用于方法之后（与其它限定词混用时应遵守本节开始时所介绍的顺序），其用于标识某个方法为虚方法。当类中的某个方法被声明为虚方法后，所有的派生类中的同名方法均为虚方法。例如 TObject 中的 Destroy 被声明成了虚方法，这导致了所有的类中的 Destroy 方法均是虚方法。

学过生物学的人大概都知道一点：生物之间的遗传具有变异性。这种变异性表现在两个方面：

- 父代能做到的事情，子代不能完成。
- 父代能够完成的事情，子代可以完成，但完成这些事情的步骤与父代有一定的差别。

通过虚方法，我们可以在编写对象时模拟这两种特性：

- 对于第一种，我们可以在子类中隐藏父类中的某个方法，使得这个方法在子类的对象中无法使用；
- 对于第二种，我们可以在子类中改写父类中的方法，使得同一个方法在父类与子类中具有不同的执行代码（方法体）。

隐藏父类中的方法非常的简单：在子类中以一模一样的方法声明父类中的某个方法并赋予其不同的方法代码即可隐藏父类中的方法。接下来我们主要来介绍如何在子类中改写

父类中的方法。这个步骤也很简单：在子类中以完全相同的形式重新声明需要改写的方法，并将其中的 virtual（如果有的话）修改成 override。例如，在上面的例子中我们在 T2 中改写了其父类 T1 中的虚方法 F1，如此一来，T2 中的 F1 与 T1 中的 F1 就有了不同的执行代码。

但这个例子的奥秘在于结果：为何第三处的调用中调用的是 T2.F1 而不是 T1.F1？答案是动态绑定。与静态绑定不同，系统在遇到动态绑定时不会简单依照调用者来决定应该调用哪个方法体，它会根据对象的真正的数据类型来调用相应的方法体。我们注意到，在第三处调用前有这样一句代码：

```
  O1 := O2;
```

其中的 O2 是 T2 类型。我们知道，对象名称实质上代表了一个指向对象实体的指针，我们假设 O2 所指向的对象实体为 obj（这仅仅是为了方便，实际应用中并不区分对象指针及对象实体），在这句代码运行之后，O1 也指向了 obj，从前面可看出 obj 是 T2 类型，所以在其后的 O1.F1 中，系统会判断出 O1 当前所代表的对象是 T2 类型，故而调用了 T2.F1。

至此，读者是否已经明白什么是动态绑定了呢？动态绑定与静态绑定的唯一区别就在于前者是根据对象的真正类型来调用相应的方法，而后者是直接根据对象名称来调用相应的方法。举例来说，前者有点像侦探，看到一张学历证书时总要推测一下证书的持有人是否假货，而后者则相当于机关人员，只要看到这份证书就会办事，根本不管这份证书的背后到底有什么样的隐情。

接下来我们介绍动态虚方法表格（Virtual Method Tabel，简称 VMT）。若读者对这部分不是太有兴趣，可跳过本部分直接浏览动态方法相关的内容。

在介绍 VMT 之前我们先来看一下对象的存储结构。我们知道，类中的成员主要有两种，一种是普通的字段，相当于一个定义于类中的变量，这些变量用于描述对象的信息，所以它们的值与具体的对象有关（例如每个人都有自己的的姓名、学号、生日等不同于其它人的信息）；另一种是方法，用于描述对象能够执行的行为（例如本章开头所定义的 TMan 类的对象能够执行 Clean 行为）。在创建一个对象时，很容易想到的方法是依照类的定义在对象中构建包括方法与字段在内的所有成员，例如我们定义如下的 TMan 类：

```
TMan = class
  name: string;
  age: integer;
  sex: string;
  procedure Clean(place:String);
end;
```

然后我们以这个类为模板创建对象 SomeBody：

```
var
  SomeBody:TMan;
...
  SomeBody := TMan.Create;
...
```

按照上面的想法，对象 SomeBody 的结构如下：

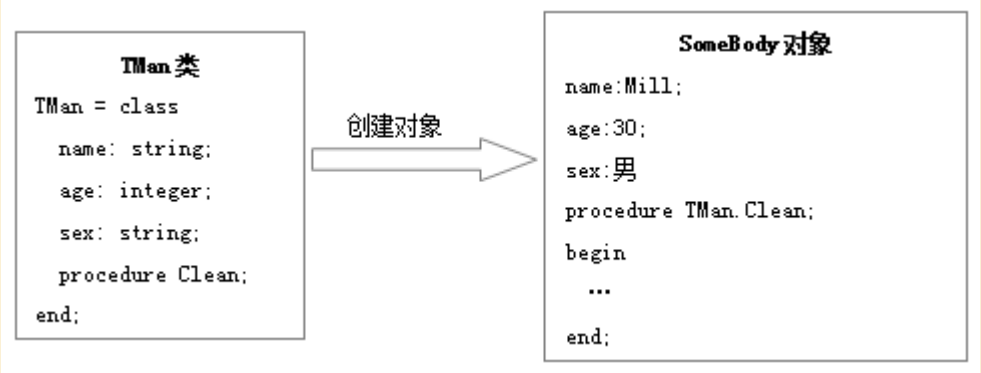


图 2

不难看出，SomeBody 中创建了包括方法和字段在内的所有的 TMan 成员。从理论上看来，这样似乎没有什么问题。果真如此吗？我们再来分析一下。

我们知道，对于任何一个 TMan 类型的对象而言，name、age、sex 均有所不同，但 Clean 肯定是相同的。假设我们现在创建了 100 个 TMan 类型的对象，这些对象中就会总共含有 100 个相同的 Clean 方法，无疑，这会造成内存的大量浪费。所以说，这样的设计不可取。

比较合理的方法是在每个对象只存储一些本对象所独有的、与其它对象不同的信息，把对象的共有信息放在一块公用的内存中并在每个对象中设置一个指针来指向这块内存，如下图所示：

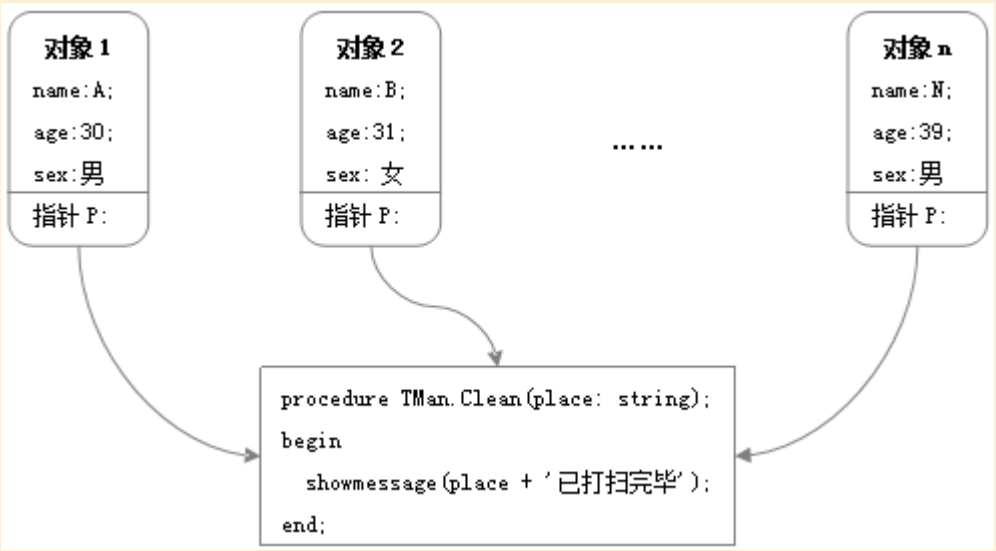


图 3

从各个角度看来，这种选择要前一种好的多，这也正是 Delphi 的选择。但这种机制在牵涉到虚方法时出现了一点小小的问题。

按照以上介绍的机制，虚方法应当与普通方法一样均被储存于一个公共的内存区域并在所有对象中设置一个指针来指向这块区域。但我们知道，类中的虚方法可以在派生类中被改写，这样一来，派生类就要为这样被改写的虚方法（并不是所有的虚方法，仅仅是那些被改写的虚方法）重新划分一块内存并将这些虚方法存储在其中。好了，既然子类终究都要重新划分内存来存储虚方法，这还不如一开始就将虚方法与其它的普通方法分开存储。Delhpi 就采用了这种机制。

Delphi 的编译器在编译时会将所有的动态绑定的方法与静态绑定的方法分开存储，对于静态成员只需为定义这些成员的类创建一块内存即可，与静态绑定的成员不同，动态绑定的成员在子类中可能被改变（如虚方法被改写），故而每个类都应当有自己的一块内存用于存储虚方法及其它的动态绑定成员。在执行完这个步骤后，编译器在每个类中插入一张表，其中列举这个类所拥有的所有虚方法的名称及每个虚方法所在的地址。您可能想到了，没错，这张表就称为虚方法表。请注意，表上列举出的是当前类所拥有的所有的虚方法，不仅仅是那些被改写过的方法，所若某个类中定义 100 个虚方法，这个类的子类又定义（不是改写）了 1 个自己的虚方法，那么这个子类中就含有 101 个虚方法。若子类改写了某个虚方法，则在子类的 VMT 中就会使用改写过的方法取代父类中的原方法。下图展示了这些机制：

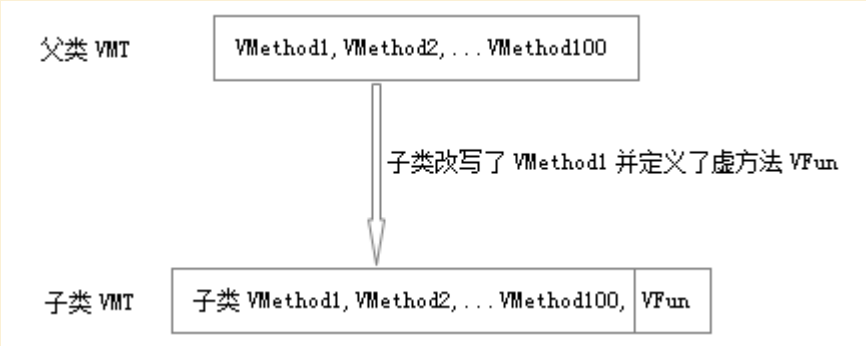


图 4 虚方法的内部存储

在调用虚方法时系统会根据对象的真正类型来查询相应的 VMT，从而调用到正确的虚方法。

3. 动态方法

以上我们介绍了虚方法及 VMT，通过 VMT，系统可以高效地调用虚方法，但这样做偶尔也会有点小困境。我们知道，根据 VMT 的机制，派生类中的虚方法数目永远大于其祖先类，这样就会遇到一个问题：我们假设现在定义了一个 TA 类，其中定义了 100 个虚方法，这些虚方法在其子类 TB 及 TB 的子类 TC 中均会使用到，但 TC 的子类 TD 只需要用到这 100 个方法中的极少数，为了这极少数的方法，TD 需要背负全部的 100 个方法。这就相当于要求年收入 2 万元的阶层与年收入 200 万元的阶层尽同等的社会责任，这对于前者很明显是不公平的。作为这种负担的恶果，TD 的对象在调用虚方法时会进行大量的无意义的查询，这使得调用效率受到极大的影响。

Delphi 使用了动态方法来应对这个问题。动态方法是一种特殊的虚方法，它使用 dynamic 而非 virtual 来声明，下面声明的 F1 就是一个动态方法：

```
type
  T1 = class
    procedure F1;dynamic;
  end;
```

作为特殊的虚方法，在使用方面动态方法与一般的虚方法毫无二致，在此不再赘述。二者的区别主要在于内部的存储机制上。

Delphi 在编译时将动态方法与普通的虚方法存储在不同的区域中，编译器会为每个类创建一张不同于 VMT 的动态方法表（Dynamic Method Table，DMT）。不同于 VMT，DMT 中只存储了当前类中新定义的动态方法及改写的动态方法的地址，至于那些定义于祖先类而未在当前类中被改写的动态方法，DMT 并没有存储它们的地址。我们使用下面的图来展示动态方法的内部存储机制：

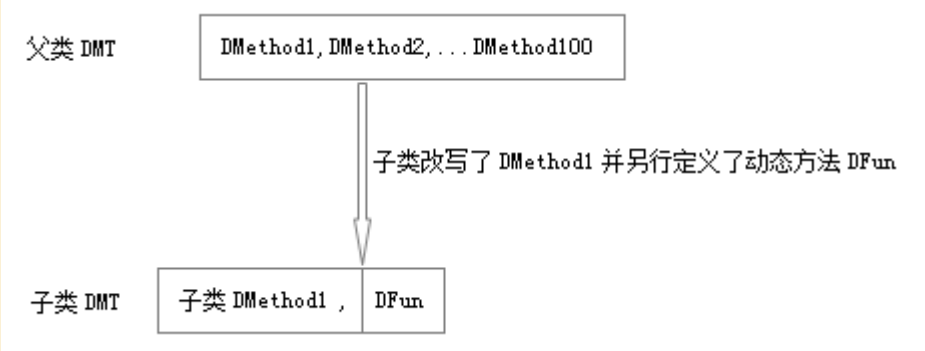


图 5 动态方法的内部存储

现在我们来解决开头的问题。在声明 TA 时可将那些最有可能被 TD 使用的方法声明成动态方法，再于 TD 中改写。这样一来，当 TD 调用这些方法时会直接进入 DMT 而 VMT 中寻找，如此，可避免大量的无用查询从而提高调用的效率。

读者可能会想，既然如此我们把 TA 中所有的方法都声明成动态方法不就万事大吉了吗？

在回答这个问题之前我们先来看一下程序在运行时如何根据动态方法的调用来确定其方法体。以 TD 为例，当我们使用如下语句调用其中的 DMethod2 时，程序会从当 TD 开始沿着继承关系向上查询，直到最终的 TObject，如下所示：

- 确定 DMethod2 是一个动态方法，并在 TD 的 DMT 中查询，若 TD 的 DMT 中存在这个方法的地址，则调用相应的方法，若不存在则进入下一步；
- 在 TC 的 DMT 中查询，若存在 DMehtod2 的地址则调用相应的 DMethod2，若不存在则进入下一步；

……

- 在 TObject 的 DMT 中查询，若存在 DMehtod2 的地址则调用相应的 DMethod2，若不存在则引发一个运行期错误；

根据以上步骤，若 TObject 类中的某个方法未经任何派生类改写，那么派生类在查询时不得不查询至 TObject 才能获得相应的方法地址。这种做法在一些小型的程序中倒也无所谓，但对象某个较为复杂的系统，其中的可能存在十几甚至几十代的派生类，这种情形下如果还使用这种查询方法将会极大的影响效率，其效率可能比不使用动态方法时还要来得低。所以说，把类中的方法全部定义成动态方法的做法，既不明智，也无必要。

实质上，动态方法与虚方法相当于天平的两端。动态方法可以减小程序占用的空间，但调用速度比虚方法稍慢。虚方法速度较快，但它会增加程序占用的空间。读者可根据实际需要自行选取。至于哪些场合下更加适合使用动态方法则是个见仁见智的问题，这需要

由读者自己去把握。不过在绝大多数场合，使用普通的虚方法都要比使用动态方法来得更为理想，故而 Delphi 建议使用虚方法，除非是在 D-Table 明显优于 V-Table 的场合（例如基本类有庞大数目的虚方法而其子类只需要改动其中的一小部分，此时就可将这些需要改动的方法声明为动态方法）。

4. 消息方法

Delphi 可使用消息方法直接响应 Windows 系统的消息。由于 Delphi 中需要直接使用消息方法的场合并不多，所以我们简要介绍。

消息的一般形式为：

```
procedure 方法名称(var 参数名：消息类型); message 消息 ID;
```

只要系统产生了方法后的 ID 所标识的消息，此方法就会被调用。例如下列方法会在窗口被关闭时被调用：

```
procedure WMClose(var MSG: TMESSAGE); message WM_CLOSE;
```

定义一个消息方法应注意以下三点：

1. 消息方法必须是一个过程，其参数只有一个且以 var 方式传址。
2. 参数的类型必须是 Delphi 封装的消息类型，这些类型定义于 Messages 单元中。
3. 方法后用关键字 message 加上消息 ID 限定。消息 ID 表示 Windows 消息的编号，读者同样可在

Message 单元中找到这些消息 ID 的声明。

6.3.3 抽象方法

Delphi 中还存在着一种独有的方法：抽象方法。抽象方法允许我们只声明而不定义。声明抽象方法只需在方法的 virtual 或 dynamic 后添加 abstract 即可：

```
普通方法声明; virtual 或 dynamic; abstract;
```

抽象方法首先应该是虚方法或动态方法。这是因为抽象方法可允许在当前类中不进行定义而留待子类中定义，这实际上是一种覆盖，或者说改写，所以能被声明抽象方法的只能是虚方法或动态方法。

任何类中只要含有了抽象方法，不管其数目是一个还是多个，均属于抽象类。抽象类的对象不能调用抽象方法，否则会引起运行错误，但调用非抽象方法不会引起任何问题。

下面是一个例子，展示了抽象方法及抽象类的用法。这个例子中我们没有使用命令行程序，取而代之，我们使用窗体程序。请读新建一个窗体程序，然后在窗体中添加两个按钮，并按下列程序编写两个按钮的 ButtonClick 事件。为也节省空间，我们省略了一些 IDE 自动编写的代码。

```
type
  TForm1 = class(TForm)
  ...  //省略
  T1 = class
    procedure M1;virtual;abstract;
```

```
  end;
  T2 = class(T1)
    procedure M1;override;
  end;

var
  Form1: TForm1;

implementation
{$R *.dfm}

procedure T2.M1;
begin
  showmessage('T2.M1');
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  obj:T1;
begin
  obj :=  T1.Create;
  obj.M1;  //引起运行错误
  FreeAndNil(obj);
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  obj:T2;
begin
  obj :=  T2.Create;
  obj.M1;
  FreeAndNil(obj);
end;
end.
```

最后声明一点：虽然我们将抽象方法放在的对象部分，但这并不代表 Delphi 不允许存在抽象的类方法。但问题在于，我们没有充足的理由来使用抽象的类方法。

6.4 类字段及类方法

6.4.1 类字段

现在我们来定义一个 TBall 类用于描述所有的球类。这个类中含有一个 shape 字段用于描述球的形状。很明显，球都是圆的。所以 TBall 的所有对象中的 shape 字段值均为

Round。为了避免我们繁的手动替每个 TBall 对象的 shape 字段赋以 Round，我们将 shape 定义成类字段。

类字段用于描述一个类的所有成员都拥有且值均相同的信息，例如我们可将国籍声明为一个类字段，如此一来，这个类的所有成员都会拥有相同的国籍。Delphi 中使用关键字 class var 声明类字段。

下面是 TBall 的定义：

```
type
  TBall = class(TObject)
  class var
    shape:string;
    name :string;
  end;
```

TBall 含有两个类字段 shape 与 name。在替这两个字段赋值后，所有的 T1 类型的对象的 name 及 shape 字段的值均为这两个值。如：

```
var
  obj:TBall;
begin
  TBall.shape :=  'Round';
  TBall.name :=  'Ball';
  obj :=  TBall.Create;
  writeln(obj.shape);
  readln;
end.
```

运行后显示结果：'Round'。

与单元中的 var 区域相似，class var 后也存在一个区域，声明于其中的所有字段均为类字段。此区域以 class var 开始，以下列 4 种情形的其中之一结束：

1. var 声明另一个 class var 声明
2. 包括对象方法类方法、构造或析构函数在内的任何一个例程的声明
3. 任意一个属性的声明
4. 任何一个访问权限限定词（如 public、private 等）

注意： 当 class var 区域出现了 const 声明的标识符时表示此标识符是一个类常量，并不表示 class var 区域的结束。

6.4.2 类方法

类方法有两种：普通类方法与静态类方法。为方便起见，我们将二者简称为类方法与静态方法。

1. 普通类方法

类方法的声明以 class 开始，其余与一般方法没任何区别。定义方法时不可省去 class。如：

```
type
  T1 = class
```

```
    class function F2(var s:string):integer; virtual;
  end;
  ...
  class function T1.F2(var s:string):integer; virtual;
  begin
    //方法代码
  end;
```

类方法与普通方法都可以通过对象来调用,但类方法还可以直接通过类来调用。所以通常将那些需要由类来调用的方法定义为类方法，如前面创建对象时所调用的 create 方法。

普通类方法中的 self 指向调用此方法的对象或类本身。当类调用类方法时，其中的 self 指向类本身，而通过对象来调用时则指向对象。

2. 静态类方法

静态类方法声明需在普通的类方法后加上限定词 static，如下所示:

```
type
  T1 = class
    class function F2(var s:string):integer; static;
  end;
```

静态类方法与普通的类方法非常相似，二者的唯一区别在于：类静态方法没有引含的 self 参数，所以类静态方法不能引用任何的对象成员，但可以引用类字段和类方法。事实上，Delphi 引入这种类型的方法完全是为了与微软的.net 框架相互兼容。就功能上而言，普通的类方法完全可以替代静态类方法。

6.4.3 构造与析构函数

每一个继承于 TObject 的类中都隐式或显式的声明了两个特殊的类方法：构造函数及析构函数。它们分别负责对象的创建及销毁。读者是否还记得，我们在前面介绍对象的创建时使用了如下的语句：

```
对象名称  :=   类名.create;
```

其中的 create 就是一个构造函数。

1. 构造函数

我们知道，对象的本质是一种特殊的结构类型的变量，我们把这种结构类型称为类。对于普通的结构类型如记录、数组等类型的变量，我们只需简单声明后就可以直接拿来使用，这点与简单类型的变量一样。直觉会告诉我们，当某人以一种非常确定的说法来说明一件事时，这其中一定有例外。类类型的变量就是这样一个意外。

作为一个常识，我们应当认识到：所有的变量均需在内存中有一处容身之地。对于简单类型的变量而言，在我们声明这些变量后，它们所需要的内存由编译器自动分配。对于数组、记录、枚举等其它结构类型亦是如此。但事情到了类类型的变量（对象）时发生了一点小小的变化：编译不能自动替这些变量分配内存，这需要由我们自己在代码中完成这项工作。

同前面介绍的指针一样，对象被声明后必须手动为其分配内存。在前面介绍动态变量中，我们可以通过一些内存分配函数来分配内存。在此处，我们并不需要这么麻烦。一切的问题依然交给编译器来处理，我们所要做的，就仅仅是调用类的构造函数。所谓构造函数就是专门用于创建对象的函数。与普通的例程不同，构造函数使用关键字 constructor 声明：

```
constructor create（参数列表）;
```

相比我们前面遇到的的函数，构造函数有三点奇特之处：

其一，构造函数不使用 function 或 procedure 声明，它必须使用 constructor 声明。

其二，构造函数的名称较为固定，统一使用 create（不分大小写）。Delphi 并没规定构造函数的名称，然而其内部统一使用了 create，为了保持编码的兼容性，我们最好不要使用其它名称。

其三，我们并不需要在每个类中都声明这样一个函数。TObject 中声明了构造函数，作为 TObject 的派生类，所有的类中其实都隐含了这样一个函数。所以当我们需要创建对象时，我们可以直接通过类的名称调用这个 create 函数来创建一个对象，如下例所示：

```
type
  T1 = class(TObject)
    i:Integer;
  end;

var
  O1:T1;
begin
  O1 :=  T1.create;
  O1.i :=  289;
end.
```

在这段代码中，我们定义了 T1， 其中未声明任何形式的名为 create 的方法，但我们却可通过 T1 来调用 create 方法，这个 create 方法正是从 TObject 继承而来。

大部分读者在此处都会怀疑：若一个类有很多的成员，从 TObject 中继承而来的构造函数是否能分配足够的内存？我是不是还要自己来调用 GetMem？我们回答是：不要有任何疑惑，Delphi 的编译器会处理一切的问题来创建一个对象，我们唯一需要确认的就只是保证自己没有把 create 与 crete 或其它的形式。

可能读者会问：如果每个对象都可以通过 TObject 中的 create 来创建，我们还有必要自己去定义一个构造函数么？答案是 Yes。我们先看一下 TObject 中 create 方法的声明：

```
constructor Create;
```

这个函数不接受任何的参数，它的作用仅仅是在堆中创建对象的实体并简单的将实体中的各个字段初始化成相应类型的初始值，如整数类型的字段被初始化为 0，字符串类型的字段被初始化成空字符串，而指针类型则被初始化成 nil。但我们需要改变这种初始化的值，或者，我们需要在对象的创建过程中执行一些其它的行为时，我们就需要定义一个自己的构造函数。

也就是说，如果你仅仅只想创建一个对象，那么，你完全没必要定义自己的构造函数。只有当我们想在创建对象时执行一些其它的行为时才需要这样做。

下面的代码中我们定义了类 T1，我们希望 T1 的对象在创建时能够自动显示一个对话框将一些信息告知用户，所以我们自定义了一个构造函数。注意，这个例子需要在 uses 从句中添加 Dialogs 单元：

```
uses
  SysUtils, Dialogs;
type
  T1 = class(TObject)
    i:Integer;
    constructor create;
  end;

constructor T1.create;
begin
  showmessage('创建了 T1 的对象');
end;

var
  O1:T1;
begin
  O1 :=  T1.create;
end.
```

本例中，只要我们创建了 T1 的对象，就会有一个对话框显示。从这个例子可以发现，通过自己定义一个构造函数，我们可以在对象的创建过程中执行各种各样的行为，这为我们编程带来的很大的方便。同时我们也应当注意到 TObject 中的 Create 方法采用了静态绑定，所以当我们在 TObject 的子代中定义了一个自己的 create 函数时，它会掩盖 TObject 中的 create 方法。换句话说，我们通过 T1 类型的变量调用到的只能是 T1 中的 create 而绝对不可能是 TObject 中的 create。

更多时候，自定义构造函数的目的是为了在对象的创建期间替不同的字段赋以不同的初值。面对这种问题，我们同样可以自定义构造函数。这次，我们需要在 T1 的对象被创建的过程中将 i 的值初始化成 289，为达到这个目的，我们需要重载定义一个构造函数：

```
uses
  SysUtils, Dialogs;
type
  T1 = class(TObject)
    i:Integer;
    constructor create(value:Integer);
  end;

var
  O1:T1;
constructor T1.create(value: Integer);
begin
  i :=  289;
end;

begin
  O1 :=  T1.create(289);
```

```
  //O1 :=  T1.create;
  writeln(O1.i);
  readln;
end.
```

读者若将例中注释取消然后运行，编译器会无法通过，显示如下错误：

```
[DCC Error] Project1.dpr(22): E2035 Not enough actual parameters
```

造成这种结果的原因是 T1 中的 create 掩盖了 TObject 中的 create，所以 T1 中不存在不接受参数的 Create 方法，也就是说，通过 T1 调用 create 方法时必须传递一个参数。

我们知道，构造函数的本质是一个普通的类方法，通过类名称调用时它会创建一个新的对象，那么当我们通过对象名称来调用它时，会发生什么结果？答案是对象的字段值会随着参数改变。我们再一次使用 T1 为例。我们将为 T1 定义两个构造函数，由于这两个函数同名，所以其后需要加上 overload 关键字，如下：

```
type
  T1 = class(TObject)
    i:Integer;
    constructor create;overload; //A
    constructor create(value:Integer);overload; //B
  end;
```

为了方便起见，我们将两个方法分别以 A、B 代称。二者的定义代码如下：

```
constructor T1.create(value: Integer);
begin
  i :=  289;
end;
constructor T1.create;
begin
end;
```

接下来的例子中，我们将使用 A 来构造一个对象，然后利用 B 来改变字段 i 的值：

```
var
  O1:T1;
begin
  O1 :=  T1.create;
  writeln(O1.i);
  O1.create(289);
  writeln(O1.i);
  readln;
end.
```

按下 F9 运行，结果为：

```
0
289
```

但这种改变并非任意，O1 能够能够构造函数 B 来改变字段值的前提是 B 的定义代码中明确的对字段 i 进行了赋值，若没有这种明确赋值，i 的值将不会被改变。根据这个前提，我们接下来将改变一下顺序，先用 B 来创建对象，然后用 A 来改变字段值，看看有何后果：

```
var
```

```
  O1:T1;
begin
  O1 :=  T1.create(289);
  writeln(O1.i);
  O1.create;
  writeln(O1.i);
  readln;
end.
```

运行结果为：

```
289
289
```

虽然使用构造函数 A 创建对象 O1 时，O1 的字段值为 0，但毕竟 A 中没有代码明确的改变字段值，所以 A 不像 B 一样可以用于改变对象的字段值。

到此，读者可能感觉在这些文字中没有看到什么有价值的东西。这也不奇怪，因为 Delphi 的编译器在后台为我们做了许多的工作，留给我们的只有一件事：通过重载构造函数来执行一些自定义的行为。所以，对于 Delphi 的程序员而言，只有需要在创建对象时执行一些其它的行为时才需要自己定义构造函数（重载），除此外的任何其它场合均可无视构造函数。

2. 析构函数

创建的对象在不需要时必须手动调用对象的析构函数来销毁。与构造函数相对，析构函数使用关键词 Destructor 来声明。声明形式如下：

```
 destructor Destroy;
```

和构造函数类似，析构函数也自动的存在于每个类中，使用时只需通过相应的类名称来调用即可。Delphi 的析构函数多使用 Destroy 作为名称。这同样也只是一个约定而并非规定。

由于 TObject 中将析构函数声明成虚方法，故当需要自定义析构函数时我们可使用 override 直接改写父类的析构函数即可，即使这个类不是直接继承于 TObject。

当某个类的构造函数在运行时出现了异常，系统会自动调用此类的析构函数。所以说，析构函数具有高度的智能化。类似于构造函数，默认的 Destroy 函数满足大部分场合的需求，所以我们一般不用费神去改写析构函数。不过若对象使用了外部的资源，而这此资源又未能在其它地方使用代码释放时，最好在改写析构函数中释放这些资源。下面的示例中声明了一个类 T1，在其构造函数中打开了一个外部文件，当 T1 的实例被销毁时，我们就需要改写其析构函数，在其中关闭这个打开的文件。

```
type
  T1 = class
    F:Integer;
    constructor create(path:string);
    destructor Destroy;override;
  end;

constructor T1.create(path: string);
begin
```

```
  F :=  FileOpen(path,fmopenreadwrite);
end;

destructor T1.Destroy;
begin
  inherited;
  FileClose(F);
end;

begin
end.
```

注意析构函数只是将对象占用的内存释放，并不没有将对象的引用设为空。这好比有关部门在没告知纳税人的情况下悄悄的搬进了其它的办公大楼，这自然是有问题的（最常见的问题可能是你跑腿的次数增加一次）。为避免这个问题，在调用析构函数后建议手动将对象的引用置为空：

```
O1.destroy;
O1 :=  nil;
```

Delphi 提供了标准函数 FreeAndNil 来统一上面的两个步骤：

```
FreeAndNil(O1);
```

当仅仅只需要销毁对象所占用的内存时，应该调用 Free 而不是 Destroy：

```
O1.Free;
```

二者间的区别是：Free 在销毁前会判断对象所占用的空间是否已经被销毁，然后再调用 Destroy。这样可防止将已经销毁过的内存再销毁一次（试图将已销毁的内存再销毁一次往往会导致系统崩溃）。

6.5 属性

对象中的字段用于储存对象的某些信息。字段相当于声明对象中的普通变量。当我们赋值时只需直接通过对象来引用字段。但有时候，字段的值必须符合一定的条件，如年龄不能小于 0，也不可能为 10000，人的身高不可大于 5 米。这种时候，我们却无法保证我们赋予的值一定在正确范围内。

为了解决这个问题，我们可以将字段声明为对象的 strict private 字段，然后在 public 区域声明两个函数分别用于读取及写入此字段的值，如下所示：

```
type
  M1 = class
  strict private
    Age:integer;
  public
    procedure SetAge(x:integer);
    function GetAge:integer;
  end;

function M1.GetAge: integer;
begin
```

```
  result :=  self.Age;
end;

procedure M1.SetAge(x: integer);
begin
  if (x<100)and(x>0) then //年龄的值只能在 0 至 100 之间
    self.Age :=  x
  else
    writeln('Age 不符合范围');
end;

var
  obj:M1;
begin
  obj :=  m1.Create;
  obj.SetAge(98);
  writeln(obj.GetAge);
  FreeAndNil(obj);
  readln;
end.
```

通过这种方法，我们可以手动控制字段的值的范围，使之不至于出现诸如年龄为 0.6 之类的不合理的值。但利用这种方法，每次读写字段值时均需显式的调用函数，显得不是那么直观。

Delphi 提供了属性封装了上面的步骤。一个完整的的属性的声明方法为：

```
property 名称：类型 read 读方法 write 写方法;
```

其中的类型表示属性的数据类型，读方法表示读取属性的方法，由 read 指定；写方法表示写入例程的方法，由 write 指定。“read 读方法”与“write 写方法”之间不存在先后顺序。类似于 read 或 write 的限定词还有：

```
read, write, stored, default, nodefault, implements
```

每个属性至少有一个 read 或 write 指定读或写方法。其它限定词的作用我们暂不介绍。

上面的例子中我们可以声明 pAge 属性：

```
property pAge:integer read GetAge write SetAge
```

这表示 pAge 属性由 SetAge 设置，由 GetAge 读取。对比上面的例子，我们发现 pAge 事实上相当于 Age 的另一个名称，但利用 pAge 我们可以更直观我读写 Age 的值：

```
type
  M1 = class
  strict private
    Age: integer; //对象的年龄
  public
    procedure SetAge(x: integer);
    function GetAge: integer;
    property pAge:integer read GetAge write SetAge;   //声明了一个属性 pAge
  end;
```

```
function M1.GetAge: integer;  //读取年龄值
begin
  result :=  self.Age;
  writeln(result);
end;

procedure M1.SetAge(x: integer);  //设置年龄值
begin
  if (x < 100) and (x > 0) then // 年龄的值只能在 0 至 100 之间
    self.Age :=  x
  else
    writeln('Age 不符合范围');
end;

var
  obj: M1;
begin
  obj :=  M1.Create;
  obj.pAge :=  98; //直接将值赋给 pAge
  writeln(obj.pAge);   //直接获取 pAge 的值
  FreeAndNil(obj);
  readln;
end.
```

pAge 属性相当于字段 Age 的助理：

- 当我们将值赋予 pAge 时，它会调用函数 SetAge 来审核这个值是否符合要求，若符合则将此值转赋给 Age，若不符合则拒绝此值。
- 同样我们读取 pAge 的值时，它也会调用 GetAge 来读取 Age 的值，然后将这个值返回。

站在用户角度，我们完全可以将 pAge 当成一个储存年龄的普通字段来进行赋值或读取。但使用归使用，实际上内存中根本不存在 pAge 这个变量，pAge 只是用于访问 Age 的一个中转站而已。所以无法用取地址符号“@”来获取属性的地址，同样，也不能用 var 将属性作为参数传递。

1. 属性访问符

属性访问符是用于读取或写入属性的方法或字段。如上面例子中用 read 指定的 SetAge 及用 write 指定的 GetAge。当属性的访问符是方法时，程序会调用相应的方法进行赋值或读取值。若属性没有指定 read 限定符，则无法读取此属性值，此时这个属性称为只写属性；若属性没有指定 write 访问符，则无法将任何值赋给这个属性，称其为只读属性。

当属性的访问符是一个字段时，程序会直接读取这个字段的值或将值赋给这个字段。

属性的访问符必须遵守以下规则：

1. 必须定义于当前类的祖先类的其中之一或当前类中，若定义于当前类中则须定义于属性之前；若定义于祖先类则必须保证访问符能够被当前类访问。

2. 若访问符是一个字段，则其与属性具有相同类型。
3. 若访问符是一个方法，则不能为动态方法或具有重载版本的虚方法。
4. 若属性定义于 published 区域，则其读方法与写方法必须采用 register 方调用式。
5. 属性的读方法必须是无参数的函数，且返回值必须与属性数据类型相同。属性的写方法必须是只有一个参数的过程，其参数只能使用 const 或 value 传递方式。

注意：索引属性及数组属性另有特殊的规定，参见下面内容。

2. 数组属性

首先声明：数组属性的意思是某个属性像数组一样由多个值组成，且每个值可通过索引读写。如我们定义一个类用于描述一幅图画时，可在其中添加一个属性 Pixel 用于描述图画上的某点的颜色信息。此时我们必须指定 X 坐标及 Y 坐标的值才能指定点的具体位置。

程序运行时只能操纵数组属性中的某个元素，如更改画面颜色时只能逐个像素进行更改。所以声明数组属性时必须指定一个索引用于指定要操纵的元素：

```
property 属性名称[索引名称：索引类型]：属性类型 访问符;
```

注意：数组属性的访问符只能为方法，不能为字段。

以下定义了前面提到的 Pixel 属性：

```
property Pixel[x,y:integer]:TColor read GetColor write SetColor;
```

其中的 TColor 用于表示点的颜色信息，GetColor 获取某个点的颜色，SetColor 用于设定某个点的颜色。x、y 为某一点的坐标值。下列调用表示将坐标为(128,128)的点的颜色设定为 Green:

```
Pixel[128,128] :=  Green;
```

数组属性的读方法必须是一个函数，其参数序列应当与属性声明中的索引参数完全一致，函数的返回值应当与属性的数据类型一致。例如我们可以这样定义 Pixel 属性的 GetColor 方法：

```
function GetColor(x,y:integer):TColor;
```

数组属性的写方法必须是一个过程，其参数分为两部分：第一部分与读方法的参数完全一致，第二部分是使用 const 或 value 方式传址方式声明的一个参数，用于储存赋给属性的值，其类型与属性一致：

```
procedure SetColor(x,y:integer; value:TColor);
```

同我们学过的其它的索引不一样，数组属性的索引可以是任何类型而不一定是有序类型。如下面的例子中就定义了一个索引类型为字符串的属性：

```
property Name[str:string]:string read GetName;
```

在声明数组属性时可以使用关键词 default 将属性声明为默认属性。当我们读写默认属性的值，可直接利用对象引用加上索引读写。如当我们将 Pixel 属性设定为默认属性时，以下两句调用等效：

```
property Pixel[x,y:integer]:TColor read GetColor write SetColor; default;
...
obj.Pixel[128,128] :=  Green;
obj[128,128] :=  Green;
```

3. Index 限定符

利用前面所学的知识，我们可较为完整的定义一个数组属性。在本部分，我们首先来利用前面所学的知识编写一个完整的实例。我们将定义一个 TTriangle 类（单词 triangle 表示三角形）用于描述任意一个三边形（三角形）。我们会在其中定义三个属性：Border1、Border2、Border3 分别表示三边形的三条边的长度。初步的定义如下：

```
type
  TTriangle = class
  published
    property Border1:Integer read GetLine write SetLine;
    property Border2:Integer read GetLine write SetLine;
    property Border3:Integer read GetLine write SetLine;
  end;
```

然后我们再分别为这三个属性定义三个字段及各自的读写方法:

```
type
  TTriangle = class
  strict private
    Line1,Line2,Line3:Integer;
    procedure SetLine1(vlaue:Integer);
    function GetLine1:Integer;
    procedure SetLine2(vlaue:Integer);
    function GetLine2:Integer;
    procedure SetLine3(vlaue:Integer);
    function GetLine3:Integer;
  published
    property Border1:Integer read GetLine write SetLine;
    property Border2:Integer read GetLine write SetLine;
    property Border3:Integer read GetLine write SetLine;
  end;
```

然后我们再为其中的每个方法定义代码，这个步骤由读者完成，在此不再余叙。

经过以上步骤，读者可能也感觉到了一个事实：Border1、Border2 及 Border3 的读写方法的代码惊人的相似，基本的格式均为：

```
procedure SetLineX(value:Integer);
begin
  LineX :=  value;
end;
function GetLine:Integer;
begin
  result :=  LineX;
end;
```

笔者向来认为，编写雷同的代码是一件非常的令人火大的事情。相信大部分读者亦有同样的感受。利用属性的 Index 限定，我们可以彻底的摆脱这种感受。具体说来，利用 Index 限定符我们可以将相似的属性存储在一个数组中，然后以此数组为对象编写一组读&写方法。

以上面的例子为例，我们可以声明一个数组 Lines 用于存储三条边的长度值：

```
Lines:array [1..3] of Integer;
```

然后再编写读&写方法用于读取或写入数组中的某个成员的值（即三边形的某条边的边长）：

```
procedure SetLine(index,vlaue:Integer);
function GetLine(index:Integer):Integer;
```

发现没有？我们在其中额外的指定了一个 index 参数，用于标识读取或设置长度值属于三边形的哪条边如 GetLine(1)表示读取的是三边形的第一条边的长度，GetLine(2)表示读取的是三边形的第二条边的长度。

然后，我们再来定义三个属性分别表示三条边。至此整个 TTriangle 类的定义完成，如下：

```
type
  TTriangle = class
  strict private
    Lines:Array[1..3] of Integer;
    procedure SetLine(index,value:Integer);
    function GetLine(index:Integer):Integer;
  published
    property Border1:Integer index 1 read GetLine write SetLine;
    property Border2:Integer index 2 read GetLine write SetLine;
    property Border3:Integer index 3 read GetLine write SetLine;
  end;
```

是否注意到属性中的 Index 限定符的用法？属性中的 Index 限定符紧接于属性的类型名称之后，其后接一个 Integer 类型的常数值（这意味着其后的常数值不能超出 Integer 类型的值域之外），这个常数值标识了当前属性指向了数组中的第几个成员，如 Border1 中的 Index 后的常数值为 1，表示 Border1 属性指向了 Lines[1]（当然了，具体指向哪个值取决于读写方法的编码，如果你愿意，你可以将 Border1 指向 Lines[3]，但我不推荐这样做）。接下来我们对读写方法进行编码：

```
procedure SetLine(index,value:Integer);
begin
  Lines[Index] :=  value;
end;

function GetLine(index:Integer):Integer;
begin
  result :=  Lines[Index];
end;
```

OK，现在我们可以代码中使用 TTriangle 类。下面的代码中定义了这个类的对象 obj:

```
var
  obj:TTriangle;
begin
  obj :=  TTriangle.Create;
  obj.Border1 :=  89;
```

```
  writeln(obj.Border1);
  readln;
end.
```

最后我们来总结一下。我们将属性存入数组的某个成员而非字段中（二者的实质相同，都是一个变量），利用这个成员取代普通的字段变量，但数组往往不只一个成员，所以我们需要利用参数来指定当前的属性值储存在哪一个成员中，在读写方法中利用 Index 限定符来标识这个成员。最后的问题是，这个数组是怎么来的？答案是：我们将几个基类型相同的属性（如三角形的三边长、三个角的角度等）合并起来声明成了这个数组。

4. 存储限定符

属性声明中可选的 stored、default、nondefault 称为存储限定符。

此限定词决定一个声明于 published 区域的属性的值是否被存储至窗体文件中。其使用方法为：

```
property Name: TComponentName read FName write SetName stored False;
```

stored 后可以接三种标识符：

- False 或 True
- 当前类或祖先类中所含有的 Boolean 类型的字段的名称
- 当前类或祖先类中所含有的函数名，此函数不接受任何参数且其返回值必须为 Boolean 类型的值

当属性声明中缺省 stored 时，编译器会理解为 stored True。

default 限定词后接一个与属性同类型的常数，用于标识一个属性的缺省值

属性的缺少值的作用是：将属性值存储至文件时，程序会将属性的当前值与默认值比较，当二者不相等且 stored 后的值为 True（或根本没加 stored）时，当前属性值会被存储至文件中。

当属性是一个数组属性时 default 限定词有其它的含义（事实上，存储限定符根本不能用于数组属性）。

当需要在子类中取消某个在父类中声明的属性的缺省值时可使用 nondefault。缺省值不能指定为 2147483648。

二者仅支持有序类型与集合类型，集合类型的子类型的上下界在 0..31 间。当一个属性没有指定二者中的任何一个时，默认为 nodefault。

当然，某个简单类型本身就有空值，如 0、nil 等。

5. 属性的改写与重定义

祖先类中声明的属性可以子类中被改写。具体表现为：子类中可改变祖先类中声明的属性的读写方法、访问权限、缺省值等。改写属性的语法如下：

```
property 属性名 限定符;
```

可以发现改写属性的语法与声明属性很相似，但改写属性时不需要声明属性的数据类型，且限定符只能为以下值：

```
read, write, stored, default, nodefault
```

关键词 property 后只有属性名时表示仅仅更改属性的访问权限。属性的访问权限只能扩大，如只能将 private 改为 public，反之不可。

下面的例子中声明了一个属性并在子类中改写：

```
type
  M1 = class
  ...
    property pName:string read GetName write SetName;
    property pAge:TColor read GetAge write SetAge;//声明 pAge 属性
  end;

  M2 = class(M1)
  ...
    property pName;    //更改访问权限
    property pAge read FAge write SetAge;  //改写 pAge 属性的读方法
  end;
```

在 M2 中改写了 pAge，将其读取手段从 GetAge 方法变成了字段 FAge。

改写属性时不能删除原有的限定符，只能更改原限定符、添加新限定符或更改原有属性的可见性。

在子类中声明一属性时，这个属性会隐藏祖先类中的同名属性。所有的属性与对象均是静态绑定:

```
type
  M1 = class
  strict private
    Age:integer;
    function GetAge:integer;
    procedure SetAge(value:integer);
  public
    property pAge:integer read GetAge write SetAge;
  end;

  M2 = class(M1)
  strict private
    Years:integer;
    function GetYears:integer;
    procedure SetYears(value:integer);
  public
    property pAge:integer read GetYears write SetYears;
  end;

var
  O1:M1;
begin
  O1 :=  M2.Create;//调用 M2 的构造函数
  O1.pAge :=  98;  //调用 M1.SetAge
  FreeAndNil(O1);
  Readln;
end.
```

6. 类属性

所谓类属性，当然就是指类的属性。接下来话读者闭着眼睛也应该知道：类属性表示这个类所有的对象共同的属性，它可以通过类直接使用，当然也可以通过对象来使用。

类属性使用 class property 来声明：

```
class property 名称：类型 限定符;
```

类属性的限定符不能 stored 或 default。其读写方法必须是当前类中声明的类静态方法，当然也可通过当前类中声明的类字段来访问。此外，类属性不能在 published 区域中声明。

6.6 辅助类(class helper)

Delphi 自 2007 起引入了辅助类(class helper)的概念。

辅助类是一种特殊的类。其存在就像共产党员一样，只为别人，从不为自己，就像是老板的助手一样。

首先，辅助类是一个类，所以声明时最好为其指定父类，否则会从 TObject 继承。其次，由于辅助类像一个助手，而助手只能辅助一个老板，所以声明时必须指定其属主类。总结，辅助的声明方式如：

```
type
  辅助类名称 = class helper (辅助类的父类) for 属主类
  end;
```

我们通过一个完整的示例来感受一下辅助类的作用：

```
type
  T1 = class
    procedure F1;
  end;

  T1_Hlp = class helper for T1
    procedure F1;
    procedure F2;
  end;

  T1_Helper = class helper for T1
  class var
    s:string;
    procedure F1;
    procedure F2;
  end;
...
var
  O1:T1;
begin
  O1 :=  T1.Create;
```

```
  O1.F1;  //调用 T1_Helper.F1
  O1.F2;  //调用 T1_Helper.F2
  FreeAndNil(O1);
end.
```

在这个例子中，M1 的辅助类 M1_Helper 将自己的能力无偿提供给了 M1，所以 M1 的对象 O1 可自由的使用 M1_Helper 中的各种方法及类字段,仿佛些方法或类字段都定义在 M1 中一般。这就是辅助类的作用。

关于以上的例子还有几点：

1. 由于辅助类不能单独使用，故其中不能定义对象字段，只能定义类字段。
2. 一个类可以定义多个辅助类。当使用辅助类中的成员时，所用到的辅助类为源代码中距离当前调用最近的一个辅助类。所以上面 O1.F2 调用的是 M1_Helper 中的 F2 而不是 M1_Hlp 中的 F2。
3. 当主类与辅助类中具有相同的成员时，优先使用辅助类的成员。对于某些事情，如果助手能搞定，老板就没必要亲自出马。

6.7 对象引用和类引用

6.7.1 类引用

所谓类引用，是一种数据类型，它代表一个类与其它类的不同部分，简单说来有点像是类的基因。类引用的英文称呼为“class reference”。结合其功能，笔者认为将“class reference”翻成“类特征”更为合适。

类引用类型的声明语法为：

```
type
  类引用名称 = class of 类名称;
```

例如下面的代码中声明了 TObject 类的类引用 TClass：

```
type
  TClass = class of TObject;
```

通过类引用可引用类中的所有的类方法与类字段。但不能通过类引用使用对象方法和对象字段，好比在没有特定对照时，我们只能够推测基因是否来源于人类而不能判定到底来自于哪个人。

概念上，类引用很像是类的基因。在功能上，若我们将类本身也看成一种特殊的对象的话，类引用则非常像是这种特殊对象的类。例如我们可以将父类引用类型的变量作为值赋给子类引用类型的变量。由于 TObject 是所有类的祖先类，故而对于 TObject 类型的类引用 TClass，可以将任何类的类引用变量作为值来赋给它：

```
type
  MyClass = class(TObject)
  end;
  MyClsRef = class of MyClass;
var
  o1:TClass;
  o2:MyClsRef;
begin
```

```
  o1 :=  o2;
end.
```

由于类引用特异性代表了某个类，所以它可用在某些需要将类而不是对象作为实际参数时的场合。若我们定义一个函数 AcceptAllObject，它需要接受所有类型的对象作为实际参数，此时我们应当将其参数的类型声明为 TObject 类型。若这个函数需要接受所有的类作为实际参数，我们就可以使用 TClass 作为参数类型。下面的例子中展示了这一用法：

```
procedure GetClassName(cls:TClass);
begin
  writeln(cls.ClassName);
end;

begin
  GetClassName(TObject);   //用于显示参数的类名称
  readln;
end.
```

这个例子中的 GetClassName 函数要求传入一个类，然后显示这个类的名称。此时，我们就可以将参数声明为一个类引用类型。

6.7.2 对象引用

1. 对象引用的概念

对象的名称仅仅只代表一个指向对象实体的引用而不代表对象实体。这一点和字符串非常相似。换句话说，我们使用一个标识符作为变量名来声明一个类类型的变量时，实际是声明了一个指针变量，但这个指针变量指向何方？答案是堆中的对象实体。

不同于 C++，Delphi 中的对象在分配内存时采用堆分配方式。声明一个对象变量仅仅只是在栈中留下了一个指针，必须在调用构造函数之后编译器才会在堆中分配一个内存并将栈中的指针指向这个内存块。堆与栈的基本概念故而以下以下语句：

```
  obj :=  M1.create;
```

实质上可分为两步：

- M1.create 在堆中创建一个 M1 类型的对象。
- 返回一个指向此对象的指针并将其赋予 obj。

堆中创建的对象不会自动销毁，所以对象在不被需要时应当由用户手动调用析构函数来销毁对象所占用的内存，否则会有可能导致内存泄漏。注意的是堆中的对象所占用的内存被销毁后，栈中的对象变量依然指向这个位置，所以最好手动将对象变量的值设为 nil：

```
obj.Free;     //销毁 obj 所指向的对象所占用的空间
obj :=  nil; //将指针 obj 的值设为 nil
```

也可以直接调用 Delphi 提供了标准例程 FreeAndNil：

```
FreeAndNil(obj);
```

前面我们为了方便没有手动销毁所创建的对象，希望读者在使用时不要省去这一步。

注意：一般情形下，我们不区分对象及对象引用。后面内容中，若无特别说明，对象就是

```
指对象的引用而非实体。
```

Delphi 提供了两个专用的操作符：as 及 is 操作符，它们分别用于对象的转型及对象的类型判断。

2. is 操作符

is 操作符用于在程序运行期间检查一个对象引用所指的对象实体的具体类型。其用法为：

对象名 is 类名称；

此表达式返回逻辑值，若对象名代表的对象派生于或等于类名称所代表的对象，表达式返回 True，否则返回 False：

```
if obj is TObject then
  writeln('obj 是 TObject 或其子类的对象');
```

参与运算的对象名必须在堆中有一个对应的对象实体。也就是说必须在调用构造函数创建对象之后才可将对象名用于 is 运算。或对象未创建，此表达式会一直返回 False：

```
var
  obj:TObject;
begin
  obj :=  TObject.Create;
  if obj is TObject then
    showmessage('ok')
  else
    showmessage('no');
  FreeAndNil(obj);
end.
```

此程序运行结果显示'ok'对话框。若读者将粗体部分的语句去掉便可发现程序显示'no'对话框。

试图将一个非对象的值进行 is 操作会无法通过编译，如下列代码无法编译：

```
var
  i:integer;
  ...
  if i is TObject then  //无法编译，i 不是对象
  ...
```

2. as 操作符

as 操作符用于将某个特定类型的对象引用转为其它类型的对象引用。简单说来，as 用于将某个类的对象转换成其它类的对象。其使用语法法为：

```
 对象名 as 类名称;
```

此表达式将对象名所标识的对象转换为类名称所标识的类类型，然后返回转换后的对象引用。如下面代码：

```
var
```

```
01:TObject;
obj:TButton;
...
01 :=  obj as TObject;
...
```

其中 TButton 类型的对象 obj 经 as 转型后被转换成了 TObject 类型。

接下来我们就来看一下 as 操作符到底执行了哪些行为。

as 操作的过程实际上是一种对象范围的强行截取。正常的对象的名称表示一个对象的引用，它指向堆中的一个对象实体。当我们通过对象名称来调用其中的成员时，编译器会通过这个引用找到对象实体，然后在其中搜寻我们要调用的成员。当我们使用 as 强行转换对象引用时，实质上是将引用指向的对象实体进行缩小化，如 obj as TObject 中，编译器会将 obj 的实体的范围进行缩小。至于要缩小到什么程度才算停止，这个答案取决于转换后的类型。需要注意的是：对象的缩小并不是真的缩小，只是将引用所指向的范围缩小而已。通俗来讲就是对象引用的领地缩小了一点。在需要时依然可通过转换来扩大对象引用的范围。整个的过程可以用下面的一张表来表示：

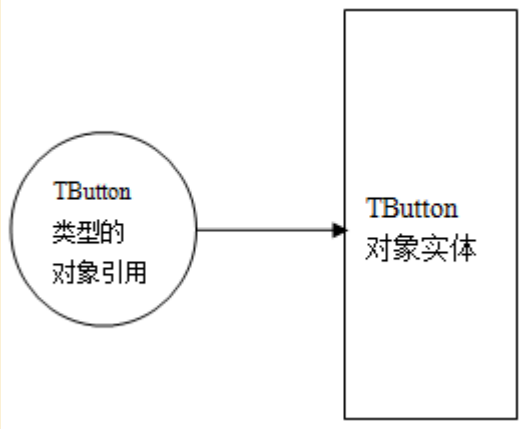


图 6 使用 as 之前的情形

使用 as 操作符转换后，对象引用指向了 TButton 中的 TObject 部分：

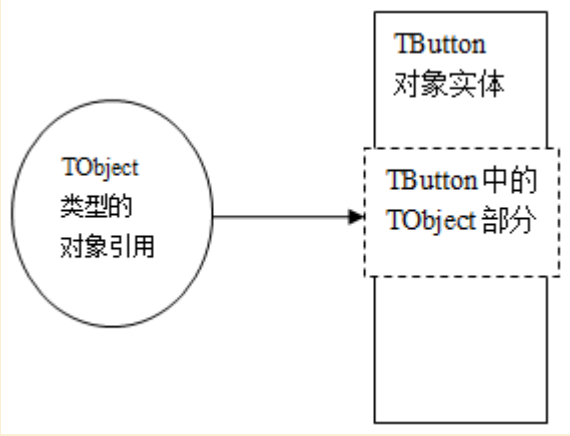


图 7 使用 as 之后的情形

上面语句中我们需要将 obj 转换成 TObject 类型引用，所以编译器会将 obj 的实体缩

小到只剩下那些从 TObject 中声明的成员，然后将这个引用赋给 o1。所以接下来通过 o1 只能调用 TObject 中的成员。这样还有个问题。若 TButton 中改写了 TObject 中的虚方法，那么 o1 引用的是 TObject 中的原有方法，还是被 TButton 改写的方法呢？答案是后者。如下面的例子所示：

```
type
  T1 = class(TObject)
    procedure M1;virtual;
  end;
  T2 = class(T1)
    procedure M1;override;
  end;

procedure T1.M1;
begin
  writeln('T1.M1');
end;

procedure T2.M1;
begin
  writeln('T2.M1');
end;
var
  o1:T1;
  O2:T2;
begin
  o2 :=  T2.Create;
  o1 :=  o2 as T1;
  O1.M1;      //显示 T2.M1
  readln;
  o2.Free;
  o1 :=  nil;
end.
```

个中的机制如下：当编译器从 TButton 中缩小实体范围时，它会在其中寻找所有的在 TObject 中声明的成员。但这种寻找是以名称为基准的。换句话说，在寻找一个名为 Free 的方法时，只需在 TButton 类型的对象的实体中找到这个方法即可，根本不会关注这个方法是由哪个对象实现的。只有当 TButton 类型的对象的实体中找不到这个方法时，才会进入祖先类中寻找名为 Free 的方法。所以，若 TButton 中覆盖了 TObject 中的虚方法，在编译器缩小对象实体时，会直接选用 TButton 中的方法。

6.7.3 Self 参数

类中声明的所有方法（静态的类方法除外），均有一个隐含的参数 self。类方法中的 self 隐含参数指向当前类本身。对象方法中的 self 隐含参数则指向了正在调用这个方法的对象实体。

某些场合下，当通过对象调用方法时，需要在方法中访问对象本身，此时可使用self，因为self指向调用当前方法的对象本身。如下列示例：

```
uses
  SysUtils, Classes;

 function TList.GetEnumerator: TListEnumerator;
 begin
   Result :=  TListEnumerator.Create(Self);
 end;

var
  obj:TList;
begin
  obj := TList.Create;
  obj.GetEnumerator;
end.
```

本段示例是预定义类TList的GetEnumerator方法的实现代码，此方法体中利用self作为参数调用构造函数来创建一个TListEnumerator对象。其中的self指向了调用当前方法GetEnumerator的TList类对象。我们在主程序块中以obj调用了GetEnumerator方法：

```
TListEnumerator.Create(Self);
```

其中的self是指obj。

6.8 其它的对象类型

1. 高级记录类型

在第三章学习记录类型时，我们遗留了高级记录类型未介绍。在本章开头时我们使用了高级记录类型作为范例，通过这些范例，读者可能对高级记录类型有了或多或少的猜想或者说是认识，但总归还是有一些迷惑。那么，本节我们就彻底解决这份迷惑。

奇怪的很，当我们正式接触这种类型时却发现没什么可讲。至少在了解了对象的相关概念后，所谓的高级记录类型充其量也只是一个低级类类型。读者在记住二者区别后，在一般场合完全可以将高级的记录类型当成类来使用。

高级的记录类型与类类型的区别如下：

1. 记录类型不支持继承机制。也就是说，这种类型就是天地所生，而且还没有后代。
2. 记录可以含有变体部分，类不能。
3. 记录类型是值类型，而类是引用类型。所以记录类型在传递时会通过复制来进行值传递。由于记录在栈中分配（除非是全局的记录类型或手动分配到堆中），所以记录类型的对象在声明后可直接使用，无需调用构造函数。
4. 类不支持运算符重载，而记录支持。
5. 记录类型可以声明一个有参数的构造函数，但不能声明任何析构函数。
6. 记录类型中不能有任何形式的动态绑定的方法，即不能含有虚方法、动态方法、消息方法。
7. 记录类型不能实现接口

2. object 类型

object 是另一种对象类型，其声明方式与 class 没有太大区别：

```
type
  类型名称 = object(父类名称)
  ...
  end;
```

其中的父类名称可有可无，但必须是另一个 object 类型。

object 仅保留用于与旧版代码兼容，所以不推荐使用。在功能上，不要说 class，就算是高级的记录类型也比 object 来得强大。所以对于这种类型，读者只需稍微了解即可。下面列出了 object 类型与 class 的区别如下：

1. 在省略父类名称时，class 默认继承自 TObject，而 object 则被认为是没有父类，其中不含有任何成员。
2. 同前面的 record 相似，object 类型的变量在声明后即可使用，无需调用构造函数，事实上也没有构造函数。也可使用 New 及 Dispose 来创建和销毁这类对象。
3. object 对象中不允许声明 published 成员。

6.9 多态

面向对象的三大特性：封装，继承，多态。

封装是将例程与变量封装成一个对象，它使得编程更为直观。继承指子类直接继承父类的所有代码，它减少了代码的编写量。现在我们来介绍最后一个特性。

所谓多态是指将不同子类的对象赋给父类对象后，父类对象会表现出相应的子类的行为。换句话说，利用多态可实现同一个对象在不同情形下表现出不同的行为。

很明显，要使得对象的行为发生变化，就必须使用动态绑定的方法。目前我们只能使用虚方法。在学习了接口后，我们同样会介绍如何利用接口来实现多态。

我们们先通过下面的例子来感性的认识一下多态到底是什么。请读者新建一个窗体程序，然后在窗体上添加三个按钮，如图所示：

在这个程序中，我们创建了三个对象，分别用三种语言说“早上好”，这三个对象所属的类都继承于 TGreed 类。

通过下面的源代码，读者可以发现虽然我们确实创建了这三个对象，但从头到尾我们

并没调用各个对象的 Greed 方法来实现这种问候，取而代之，我们全部调用了父类 TGreed 中的 Greed 方法。换句话说，同样是 TGreed 类的对象，在不同的场合，它使用了不同的语言来问候，这就是多态。

窗体对应的单元文件的源代码为：

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

  TGreed = class
    procedure Greed;virtual;abstract;
  end;
  TCHGreen = class(TGreed)
    procedure Greed;override;
  end;
  TENGreed = class(TGreed)
    procedure Greed;override;
  end;
  TJPGreed = class(TGreed)
    procedure Greed;override;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TCHGreen.Greed;
begin
  showmessage('早上好');
```

```
end;

procedure TENGreed.Greed;
begin
  ShowMessage('Good Morning');
end;

procedure TJPGreed.Greed;
begin
  showmessage('おはようございます');
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  obj:TGreed;
begin
  obj :=  TCHGreen.Create;
  obj.Greed;
  FreeAndNil(obj);
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  obj:TGreed;
begin
  obj :=  TJPGreed.Create;
  obj.Greed;
  FreeAndNil(obj);
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  obj:TGreed;
begin
  obj :=  TENGreed.Create;
  obj.Greed;
  FreeAndNil(obj);
end;
```

若读者熟悉了前面的内容，就觉得多态真是非常的简单。但在绝大多数场合，只要我们使用了虚方法，我们一定在不知不觉中使用了多态，只是读者未发觉而已。

第七章 接口

7.1 什么是接口

接口是一种结构类型，它声明了一组方法和属性，但并未替这些方法和属性实现具体的定义。

从所起作用来看，接口相当于计算机世界中的契约。现实世界中的契约规定了签订双方各自必须履行的义务，而接口则规定了所有实现这个接口的对象必须履行的义务，这种义务并不是用文字列出的若干条款，而是用程序语言声明的一组方法或属性，任何实现这个接口的对象必须替接口中的所有方法属性进行具体定义（即实现这个接口），就像任何一个履行合约的人都必须完成合同上所列出的所有条款（履行这份契约）一样。

我们将上面的比喻整理一下：

计算机世界	现实世界
接口	契约
当中声明了若干方法或属性	当中列举了若干条款
接口的使用者必须实现接口	契约的签订者必须履行契约
替接口中的方法或属性进行具体定义，即实现接口	完成契约中的每项条款，即履行契约

这样进行类比后，很容易发现接口与契约非常的相似。

那么，接口到底有什么用？总不会只是让对象去实现它所声明的方法或属性吧？当然不是。我们通过下面的例子来感受一下接口的作用。

假设我们现在要为 Word 开发一款小插件。这款的插件的功能非常简单：它在工具栏上添加一个按钮，当我们点击这个按钮时 Word 会自动保存当前正在编辑的文档。

要编写这款插件，我们首先需要在工具栏上添加按钮。问题是只有微软自己才知道调用哪个函数才能在工具栏上添加按钮该，所以问题变成了我们如何才能知道这个函数。很明显，我们不可能打电话去问微软公司。由于 Word 的源代码并不公开，所以也不可能在网上搜索。

很有可能，微软当初也想过这个问题。证据就是他们找到了解决之道：将这类的函数全部声明在一个接口中，并将接口暴露给开发者。如此一为，我们就可以调用相应接口的方法来实现这个功能。

有人可能会奇怪：接口不是没有提供方法的具体实现吗？这样调用方法还有什么用？

我的回答是：接口确实没有提供实现，但微软公司提供给开发者的并不是只有一个接口，还有一堆编译好的 DLL，而 DLL 中是可以储存代码的。当我们调用接口中的方法时，Word 会通过某种方法寻找到 DLL 中的代码并执行。

至于为什么不能直接从 DLL 中调用函数，而非要经过接口来调用，恐怕要等读者学习了 COM 编程后才会明白。

总而言之，通过接口提供只是众多解决方法的其中之一而不是唯一。就算这样，相信通过这个例子读者应该还是能感受到接口的作用的。我们再将这个例子用契约来描述一下：

我们把开发插件当成是开发者与微软公司之间的一种免费交易，为保证交易的顺利进行，二者签订了一份契约（接口），开发者通过契约要求微软公司实现契约中列举的所有

的方法及属性以提供开发所需的基本功能，而微软公司则如约实现。

至此，天下太平。

在上面的例子中，接口由提供者（微软公司）实现，这也是大部分接口的实现形式。但这不是全部，还有一部分的接口需要使用者来实现。

与提供者实现的接口相比，使用者提供实现的接口更像一份契约：提供者注明要实现的功能，使用者实现这些功能。这与上面的例子正好相反（微软提供功能，开发者从中选取自己所需的功能）

我们再次使用上面的例子。这次我们讨论点击按钮之后所发生的事情。

假设微软现在对全世界开发者宣布：在 Word 中保存文档必须使用 IDoc 接口中的 SaveDoc 方法，默认状态下这个函数会将文档保存至当前文件中。

但是我们现在有特殊的要求：我们希望当我们点击按钮后 Word 能将文档保存到 D:\dest.doc 当中，但我们又只能调用 SaveDoc。在这种情形下，微软会怎么办？

设想一下，如果我们可以自己提供 IDoc 的实现，当 Word 在保存时调用我们所提供的 IDoc，这样不就可以达到目的吗？

我们整理一下这个过程：提供者微软提供 IDoc 接口，并要求这个接口的 SaveDoc 用于保存当前文档。用户提供这个接口的实现，在保证 SaveDoc 能够保存当前文档的前提下，可以任意编写自己想要的代码。

这个例子与上个例子的不同之处就在于：上一个例子中，微软是干活的，开发者是老板。而这个例子中，微软改行当了老板，而用户则变成了员工。

上面两个例子展示了两种类型的接口：一是由提供者实现的接口，这接口是由于提供者在自己的软件中实现了许多的功能，为也方便开发者利用这些功能才用接口将这些功能包装；二是由使用者实现的接口，这种接口是由于提供者需要赋予开发者足够的自由进行自定义，所以提供接口并规定这些接口所必须提供的功能，对于开发者而言，无论通过什么步骤，只要实现这些功能即可。

在继续后续内容之前，请读者尽量感性的认知这两种接口的不同，它直接影响对接口的本质的理解。

7.2 声明一个接口

从本质来说，接口与类或记录没有多大区别，都只是一种结构数据类型，因而它的声明方式也极其类似于这些类型。声明一个接口的语法如下：

```
type
  接口名称 = Interface（父接口的名称）
  [GUID]
    //成员列表
  end;
```

注意：与 class 和 Object 一样，Interface 不能声明为局部类型。

接口可以像类一样进行继承，声明时必须指定父接口的名称。在没有明确指定的父接口时默认继承自 IInterface 接口（在 Delphi 的某些早期版本中接口默认继承自 IUnknown）。

GUID 为一个由十六进制的数字组成的字符串，作为接口的唯一标识。其形式为：

```
'{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}'
```

在接口声明中的 GUID 必须使用一对方括号括起，例如：

```
['{5E47549C-BA52-4315-83F8-DAC8BC590DC1}']
```

在 Delphi 的 IDE 中只要同时按下 Ctrl+Shift+G 即可自动产生一个完整的 GUID。从语法角度而言，接口声明中的 GUID 可省略不写，但是鉴于各种原因，建议不要省略。

与类相比，接口的声明有以下几点不同：

- 接口使用 Interface 声明，而类使用 class 来声明。
- 接口的成员不能是字段，只能是方法与属性，只也使得属性只能通过方法而不能通过字段来进行读写。
- 接口的所成员应该可以被任何使用者访问，故而接口中没必要也不允许指定任何访问权限。但一个类实现接口时，它可以定义接口中的方法的访问权限。
- 在声明属性时也不允许指定存储限定符（stored、default、nodefault）。但数组属性可指定 default。
- 由于不存在接口类型的对象，所以接口中不能含有构造与析构函数
- 由接口不能实例化，其中的方法不存在动态或静态绑定，故而方法后不能加关键词：virtual, dynamic, abstract,以及 override。

同类一样，接口也可以提前引用，语法和类的提前引用极为相似：

```
type
  接口名称 = Interface;
```

注意：接口的提前声明与这个接口的正式声明之间只能用于声明数据类型，不能声明其它的如常量或变量之类的标识符。例如下列的代码无法通过编译：

```
type
  r1 = interface;
var
  s:string;       //接口的提前声明与正式声明间只能是其它类型的声明
type
  r1 = Interface //默认继承自 IInterface
    procedure F1;
  end;
```

7.3 实现一个接口

接口仅仅声明了方法而并未实现这些方法，就使用而言，接口本身没有任何使用的价值。一个接口只有在被一个类继承之后才会有使用价值。类继承接口的语法如下：

```
type
  子类名称 = class (父类名称, 接口 1,接口 2, ..., 接口 n)
    //成员列表
  end;
```

这里有三点说明：

- 一个类可以同时继承多个接口。这种情形下，各个接口名称之间用逗号隔开。
- 类继承接口时，父类名称绝对不可省略，而且父类名称应当放在所有的接口名称之

前，写在第一位。

- 类继承接口时，必须在这个类中声明它所继承的所有接口中的所有的方法以及属性。如以下语句：

```
type
  I1 = Interface
    procedure F1;
  end;
  I2 = Interface
    procedure F2;
  end;
  T2 = class(TObject,I1,I2)//TObject 应当放在接口名称之前
    procedure F1;//必须重新声明所有接口中的方法及属性
    procedure F2;
  end;
```

在这个例子中，T2 是 TObject 的子类，它继承了接口 I1、I2 中的所有的方法及属性并在其中给予定义。I1 与 I2 之间必须以逗号隔开且只能写在类名称的后面，类名称绝对不能省略。

读者是否觉得在类中重新声明接口中的所有方法有点麻烦呢？反正笔者是这么想的。但这是没办法的事，因为接口不是为了减少代码的编写量而生，考虑到它的作用，这点麻烦根本就不算什么。

由于所有所有接口都直接或间接的继承自根接口 IInterface，而这个接口又声明了一些最基本的方法，所以任何一个实现了接口的对象都应当实现这些基本的方法。当然，如果某个类实现了这些方法，它的派生类就可直接继承这些实现。基于这种想法，Delphi 定义了一个 类：TInterfacedObject，其声明于 System.pas 中：

```
TInterfacedObject = class(TObject, IInterface)
...
end;
```

这个类中实现 IInterface 中所有的方法，所以当我们需要继承接口时，可使用 TInterfacedObject 来代替 TObject 作为父类。这样我们只需要实现我们自己定义的那些方法即可。

从上面的叙述可以看出，实现接口中的方法并不是一件多么困难的事。那么，对于接口中所声明的属性，我们该如何实现？其实这个也不困难。我们知道属性只是对方法以及字段的一种封装，所以若想实现接口中的属性，我们只需要在类中声明一个字段，并实现接口中的方法即可。

下面的代码中定义了一个 I1 接口，其中含有两个方法及一个属性，然后我们定义了 T1 类并在其中中实现了 I1 接口:

```
type
  I1 = Interface
    procedure SetX(value:integer);
    function GetX:integer;
    property P1:integer read GetX write SetX;
  end;

  T1 = class(TInterfacedObject, I1)
```

```
  strict private//定义接口中方法的访问权限
    i:integer;  //定义一个字段用于储存P1的值
    procedure SetX(value:integer);
    function GetX:integer;
  published
    property P1:integer read GetX write SetX;
  end;

function T1.GetX: integer;
begin
  Result :=  self.i;
end;

procedure T1.SetX(value: integer);
begin
  Self.i :=  value;
end;

var
  obj:T1;
begin
  obj :=  t1.Create;
  obj.P1 :=  78;
  writeln(obj.P1);
  FreeAndNil(obj);
  readln;
end.
```

最后，我们来看一下如何在子类中改变祖先类中实现的接口。一个类继承了某个接口之后，其所有派生类均会自动继承这个接口中的方法或属性。但有些时候我们需要在派生类中重载实现这些接口，我们就可以在这个派生类中重新接口。如下面的例子中，我们声明了接口I1并定义了一个类T1来实现这个接口。

```
type
  I1 = Interface
    procedure F1;
  End;
  T1 = class(TInterfacedObject,I1)
    procedure F1;
  end;
```

现在由于某些原因，T1的派生类T2需要以一种不同的方式来重载实现I1，此时我们只需直接使T2继承I1即可。

```

  T2 = class(T1,I1)
    procedure F1;
```

```
  end;
```

以下是本例的各个方法的实现及主程序块的代码：

```
procedure T1.F1;
begin
  writeln('this is T1.F1');
end;

procedure T2.F1;
begin
  writeln('this is T2.F1');
end;

var
  Interface1:I1;
begin
  Interface1 :=  T1.Create;
  Interface1.F1;
  Interface1 :=  T2.Create;
  Interface1.F1;
end.
```

注意：由于 T2 中的 I1 接口完全的隐藏了 T1 中的 I1 接口，若在 T1 中替 I1 接口的方法声明了别名，这个别名在 T2 中将不再有效。

7.4 方法别名

在一般情形下，一个类实现接口后，程序会在需要调用时按照名称来匹配。例如接口 I1 中声明了方法 F1，则当我们调用 I1.F1 时，程序会在实现 I1 的类中寻找名为 F1 的方法。

但这样存在一个问题，若类实现了多个接口，且这些接口中存在同名的方法。例如下面的类：

```
 type
   I1 = Interface
     procedure SetX(value:integer);
     function GetX:integer;
     property P1:integer read GetX write SetX;
   end;
   I2 = Interface
     procedure SetX(value:integer);
     function GetX:integer;
     property P2:integer read GetX write SetX;
   End;
   T1 = class(TInterfacedObject,I1,I2);
     procedure SetX;
     ...
```

```
  end;
```

当我们调用 T1 中的 SetX 时，我们调用的是 I1.SetX 还是 I2.SetX？可能我们知道，但计算机不知道，不过也不用担心，因为这种代码无法通过编译。

为解决这个问题，我们可以给这些同名的方法取一个不同的别名，这个别名称为方法别名。指定方法的别名可使用下列语法：

```
    procedure 接口.过程 = 方法别名;
```

若

```
    function 接口.函数 = 方法别名;
```

指定了方法别名后，在实现接口的类中应当完全使用别名代替方法的原名称。应用方法别名，上面的例子中 T1 应声明成如下方式：

```
T1 = class(TInterfacedObject, I1, I2)
strict private
  i,j:integer;
  procedure I1.SetX = set1;   //替 I1 中的 SetX 指定别名为 set1，以下同理
  function I1.GetX = get1;
  procedure I2.SetX = set2;
  function I2.GetX = get2;
public
  procedure Set1(value:integer); //定义 Set1
  function Get1:integer;
  procedure Set2(value:integer);
  function Get2:integer;
  property p1:integer read Get1 write Set1; //完全使用 set1 代替 SetX
  property p2:integer read Get2 write Set2;
end;
```

注意：别名只是方法的另一个名称，它并不能改变方法除名称之外的信息如参数、返回值等信息。所以例子中的 Get1、Set1 等方法在声明时只能有名称与原来的方法不同，其它应当完全一样。

前面我们介绍过在子类重载实现接口，如 T1 的子类 T2 重载实现了 I1 及 I2 接口，在这种情形下，T1 中为 I1、I2 中的方法指定的别名在 T2 中全部失效，需要在 T2 中再一次指定。

7.5 接口的代理

假设我们现在编写一个类 T1，它继承了接口 I1，按照规定我们应当在 T1 中实现 I1。但我们发现，在编写另一个类 T0 时，我们已经实现了接口 I1，且其中用于实现 I1 的代码正好可以拿来实现 T1 中的 I1。出于直觉，我们总是想直接把 T0 中的代码拿过来使用。最常见的做法就是将 T0 中的相关代码直接复制到 T1 中。但这并不是代码重用的方法，而且这种做法也不值得提倡。

在面向对象时代，我们有另一种方法，一种简单却有效的方法。我们在类中声明一个属性 P1 来实现 I1，当需要调用 I1 中的方法时，只需读取 P1 即可。具体过程如下：

```
type
  I1 = Interface
    procedure F1;
```

```
  End;
  T0 = class(TInterfacedObject,I1)   //t0＝字母 T+数字 0
    procedure F1;
  end;
  T1 = class(TInterfacedObject,I1)
  strict private
    FInterface:I1;
  public
    property P1:I1 read FInterface write FInterface Implements I1;
  end;

procedure T0.F1;
begin
  writeln('this is T0.F1');
end;

var
  Interface1:I1;
  obj:T1;
begin
  obj :=  T1.Create;
  obj.P1 :=  T0.Create;
  Interface1 :=  obj;
  Interface1.F1;
  readln;
end.
```

在这个例子中，我们使用了属性来实现接口。读者应该还记得属性后可接一个关键字 implements，其后拉一个接口名，表示在当前属性中实现了这个接口。我们在例子中用 P1 实现了接口 I1。可以看到，由于使用了 P1 实现接口，我们不用在 T1 类中实现 I1。

可能读者会问：既然 T1 中没有实现接口 I1，那么 T1 如何使用 I1 呢？

答案是在使用 T1 前必须将一个实现了 I1 的对象赋给 P1 属性，使得 T1 的对象能够借助他人的代码来使用 I1。本例中我们将 T0 的对象赋给 T1 的类型 obj 的 P1，这样 P1 就储存了一个 T0 的对象，利用这个对象我们就可以调用到 I1 的方法。可以看出，这个过程可以极大的增加代码的可重用性，减轻我们的代码编写量，从而节省大量的精力。这种方法就称之为接口代理。我们再来归纳一下上面的过程：

1. 在类中声明一个接口类型属性
2. 将其它实现了接口的对象赋给这个属性
3. 通过属性中储存的对象调用接口的功能

这种方法中使用了接口作为属性的类型，下面我们再来看一下如何使用类类型的属性取得同样的效果。

我们首先从一个例子开始。

```
...  //T0 与 I1 的代码与前面一样
```

```
  T1 = class(TInterfacedObject,I1)
  strict private
    FInterface:T0;  //从 I1 变成了 T0
  public
    property P1:T0 read FInterface write FInterface Implements I1;
    //P1 的类型由 I1 变成了 T0
  end;

procedure T0.F1;
begin
  writeln('this is T0.F1');
end;

var
  Interface1:I1;
  obj:T1;
begin
  obj :=  T1.Create;
  obj.P1 :=  T0.Create;
  Interface1 :=  obj;
  Interface1.F1;
  readln;
end.
```

这段代码与上段代码间看起来只有两处不同。obj 的属性也发生了一点不同，第一段代码中 obj 的 P1 属性是 I1 类型，而第二段代码中则变成了 T0 类型。这种方式称为对象代理。

其实以上的两种方法其实本质上没什么区别，如果读者理解了其中的一种就可以很轻松的学会另一种。

7.6 接口的赋值与转型

7.6.1 接口的赋值兼容

接口的赋值包括几种情况：将接口赋给其它非接口类型的变量、接口赋给另一个接口变量、将其它非接口变量赋给接口变量。接下来，我们将对这几种情况一一介绍。

1. 接口作为值赋给其它非接口变量

这种情况非常简单。因为非接口变量中，只有变体变量可接受接口类型的值。如果读者已经忘了变体变量的相关知识，请翻回到本书的第 3 章再温习一遍。

Delphi 规定，当将一个 IDispatch 类型的接口值赋给变体变量时，变体变量的类型码的值为 varDispatch。其它的任何接口值赋给变体变量时，类型码的值均为 varUnknown。

到此，没了。

2. 非接口变量值赋给接口变量

此情形可分为三种可能：变体变量值赋给接口变量、nil 赋给接口、类的对象赋给接口。

来而不往，那个啥。接口类型的值能赋给变体变量，变体类型的值也可以赋给接口类型的变量。Delphi 对此种可以有以下规定：

- 当变体变量的类型码为 varUnknown 时，它可以当成值赋给 IInterface(IUnknown)类型的变量。
- 当类型码是 varDispatch 或 varEmpty 时，它既可以赋给 IInterface(IUnknown)类型的接口变量，也可以赋给 IDispatch 类型的接口变量。
- 其它任何类型的变体值均不能赋给接口类型，任何类型的接口变量均不能接受变体值对于 nil，它可以作为值赋给任何接口类型的变量。

若将对象赋给某接口类型的变量，则此对象必须实现了此接口本身（即使实现了此接口的子代接口也不可以）。如下面的代码：

```
type
  I1 = Interface(IInterface)
  End;
  I2 = Interface(I1)
  End;
  T1 = class(TInterfacedObject,I2)
  end;

var
  obj:T1;
  Interface1:I1;
  Interface2:I2;
begin
  obj :=  T1.Create;
  Interface1 :=  obj;
  //此句错误。obj 实现了 I2 接口，它不能作为值赋给除 I2 以外的任何接口类型的变量
  Interface2 :=  obj;
  FreeAndNil(obj);
end.
```

3. 接口变量值赋给另一个接口变量

这种情形只有一种可能：派生的接口值赋给祖先接口变量。例如若要将接口 I1 赋给接口 I2，那么 I2 必须是 I1 的派生接口，这种赋值事实上是将子代接口转换成了父代接口。实际应用中这种转换往往没有什么太大的意义，但若子代接口指向了一个对象，那么将它赋给父代接口时相当于将这个对象直接赋给了父代接口。例如我们定义接口 I1 及其子接口 I2：

```
  I1 = Interface
    procedure M1;
```

```
  End;
  I2 = Interface(I1)
    procedure M2;
  End;
```
然后再定义一个实现了 I2 接口的类 T1：
```
  T1 = class(TInterfacedObject,I2)
    procedure M1;
    procedure M2;
  end;
```
下面是主程序块的代码：
```
var
  obj:T1;
  Interface1:I1;
  Interface2:I2;
begin
  obj :=  T1.Create;
  Interface2 :=  obj;
  Interface2.M2;
  Interface1 :=  Interface2;
  Interface1.M1;
  readln;
end.
```
在这个例子中，我们将 T1 的对象赋给 Interface2，再将 Interface2 作为值赋给 Interface1 ，如此一来，Interface1 其实也指向了 obj。对于 Interface1 而言，这个的过程相当于直接把 obj 作为值赋给它:
```
Interface1 :=  obj;
```

7.6.2 接口的转型

前面提到的接口赋值实际上也是一种接口的转型。在本部分内容，我们再详细讨论一下接口与对象之间的类型转换。我们先来声明一个接口及一个类：
```
type
  I1 = Interface
  end;
  T1 = class(TInterfacedObject,I1)
  end;
var
  obj:T1;
  Interface1:I1;
```
现在我们来讨论如何在 obj 与 Interface1 间进行相互的类型转换。按前面的介绍，我们可以直接将对象赋给接口：
```
Interface1 :=  obj;
```
这样我们可以通过 Interface1 来直接调用 obj 中的方法或属性，但只能调用 I1 中声明过

的方法和属性。在将对象赋给接口变量后，若我们想将接口变量再转回对象，我们可以直接进行强制转型：

```
obj :=  T1(Interface1);
```

以上的方法确实能够在对象引用与接口之间进行转换，但这种转换是一种隐匿转换，在某些场合它们并不能确保转换的安全性。为了规范起见，我们应当更多的使用明确的转换。

Delphi 提供了这样的方法：使用 as 操作符。使用 as 可将对象引用转换成接口引用。如下面所示：

```
type
  I1 = Interface
  ['{D8C36ABA-FCF9-46F1-A55B-8E69EEA75244}']
    procedure M1;
  End;
  T1 = class(TInterfacedObject,I1)
    procedure M1;
    procedure M2;
    procedure M3;virtual;
  end;

procedure T1.M1;
begin
  writeln('T1.M1');
end;

procedure T1.M2;
begin
  writeln('T1.M2');
end;

procedure T1.M3;
begin
  writeln('T1.M3');
end;

var
  Interface1:I1;
  obj:T1;
var
  Interface1:I1;
  obj:T1;
begin
  obj :=  T1.Create;
  Interface1 :=  obj as I1;
  Interface1.M1;
```

```
  Readln;
end.
```

注意：但接口与对象引用有点不同，在转型时我们需要动态查询接口中的成员，所以使用 as 转型的接口在声明时一定要标上 GUID，否则无法编译。

当然，利用 as 操作符也可把接口转回对象引用。我们保持其它部分不变，仅仅加入 3 行代码，这 3 行代码中先将 obj 置为 nil 然后利用 as 将 Interface1 转换成 T1 类型的引用并赋给 obj，然后通过 obj 调用 M3。代码如下：

```
  begin
    obj :=  T1.Create;
   Interface1 :=  obj as I1;
   //加入 3 行代码
    obj :=  nil;
    obj :=  Interface1 as T1;
   obj.M3;
   //至此
    Interface1.M1;
    Readln;
  end.
```

从这个过程我们可以清楚的感觉到接口被转换成了对象引用。可以看到这个方法被声明成了虚方法，这样可以解除 M3 与 obj 之间的静态绑定。若 M3 是静态绑定，则就算被置为 nil 它还是能够调用 M3，这样一来无论 obj := Interface1 as T1 这句代码有否起上作用，obj 都能够正常的调用 M3，而这样很明显不符合我们的需要。

最后，我们简要介绍一下接口转型的原理。

我们知道，对象名实际上代表了对象实体的一个引用，类似地，接口名也代表了一种接口类型的引用，但与普通对象引用不同的是，在堆中并没有与接口对应的对象实体。从这个角度来说，接口可以看成是一个只有引用没有实体的对象。当我们将一个对象名作为值赋给一个接口名时，实际上是将接口名所代表的接口引用指向了对象名所指向的对象实体。如 Interface1 := obj 实际上是将 Interface1 指向了 obj 代表的对象实体。我们在对象引用的 as 操作符中提到过的对象实体缩小化的过程在此处再一次应用，将对象引用赋给接口引用时，编译器会将对象实体中的成员缩小，只剩下接口中声明的成员。所以无论是将对象赋给接口，还是使用 as 进行转换，通过接口引用我们只能引用在这个接口中声明的成员。反过来，当我们将接口转换成对象时，实质上是将接口所指向的对象的实体范围进行扩大化，然后传给相应的对象。但我们知道，接口是没有实体的，所以只有先通过某种方式（如利用直接赋值或 as 操作符将对象引用转换成接口引用）使得接口引用指向一个对象实体之后才能将其转换成对象。

需要注意一点，由于接口的生存期受系统自动管理，当我们将某个对象赋给一个接口时，我们无需手动销毁这个对象，因为接口由系统自动管理，当接口被销毁时其所指的对象也会随之销毁。请读者不要小看这个特性，若运用恰当，这个特性可以极大的减轻我们管理对象的负担。

7.7 使用接口实现多态

在前面的内容中，我们介绍并使用虚方法实现了多态。本节我们再介绍一下使用接口

来实现多态。其实多态从实质来说，就是通过某种方法使得父类调用子类中的方法以达到同样的代码可以用于不同场合的目的。其中的关键是如何实现父类对象调用子类对象。到目前为止，我们最常用的是直接赋值，直接将子类对象赋给父类对象，其中的核心机制就是利用父类对象的引用指向子类对象的实体。

我们再次使用前面的例子。不同的是，这次我们不再定义父类 TGreed，而是定义一个接口 IGreed，其中声明了 Greed 方法。然后我们依然定义三个类并在其中实现 IGreed 接口。本例源代码如下：

```
unit Unit1;
interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  IGreed = Interface
    procedure Greed;
  End;
  TCHGreen = class(TInterfacedObject,IGreed)
    procedure Greed;
  end;
  TENGreed = class(TInterfacedObject,IGreed)
    procedure Greed;
  end;
  TJPGreed = class(TInterfacedObject,IGreed)
    procedure Greed;
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.dfm}

procedure TCHGreen.Greed;
begin
  showmessage('早上好');
end;

procedure TENGreed.Greed;
begin
  ShowMessage('Good Morning');
end;

procedure TJPGreed.Greed;
begin
  showmessage('おはようございます');
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  obj:IGreed;
begin
  obj :=  TCHGreen.Create;
  obj.Greed;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  obj:IGreed;
begin
  obj :=  TJPGreed.Create;
  obj.Greed;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  obj:IGreed;
begin
  obj :=  TENGreed.Create;
  obj.Greed;
end;
end.
```

- 与这个例子的原版本相比，这个版本有以下几点变化：

- 各个类没有继承于 TGreed，取而代之，它们实现了 IGreed 接口。
- 调用方法时不是通过 TGreed 类型的对象，而是通过 IGreed 接口。
- 所以的对象均没有手动销毁。

再次声明：由于接口的生存周期由系统自动管理，所以它指向的实体也会由系统自动销毁，无需我们手动销毁。

第八章 异常处理

所谓异常是指任何能够干扰程序正常运行的因素，例如内部运行错误、硬件错误等。异常处理是计算机世界的一种灾难防护措施，它通过某种手段使得程序在遇到意外时能够按照一套既定的规则来应对这些意外。以现实为例，异常处理有点像是地震演习，当地震发生时我们需要按照演习中的步骤来保护自己的人身安全，同样，计算机也需要按照预定的异常处理代码来保护程序的正常运行或退出。

现实中，对于地震、洪水之类的灾害，大家一般都比较熟悉，所以对于这些灾难的防护措施也比较详细。但对于某些千年一遇的灾难，从古籍中也很难知道这种灾难发生是到底是什么情形，应付这样的灾难时，由于缺乏了解，我们只能做一些最基本的防护如保证水源保持空气等。同样，在对程序进行异常处理时，对于某些较为常见的错误我们们可以较为详细地处理；但对于某些意想不到的错误，我们能做的事恐怕也只有让程序在尽量不影响系统运行的情形下顺利退出。

6.1 利用条件语句处理异常

计算机的异常处理说到底只是用户根据自己的经验对自己能想到的各种异常进行逐一的判断，然后处理。例如我们在程序中打开一个文件时，我们能想到常见的错误是：文件不存在、系统的 I/O 异常、文件发生错误无法打开、文件正在被其它程序使用等。我们可以使用条件语句进行逐个排除，下面是示例（注意此段代码不能编译）：

```
begin
  if file not exist then  //文件不存在
    DoSomeThing;
  if error with System I/O then  //系统 I/O 错误
    DoSomeThing;
  if error with file then //文件发生错误
    DoSomeThing;
  if file is using then   //文件正在被其它程序使用
    DoSomeThing;
  fileopen('d:\test.dat');//打开文件
end.
```

上面的代码处理了一些我们能设想到的情况，但若想使用这段代码处理我们没预料到的其它意外则显得力不从心。就算我们能够想到一万种可能，计算机也会用第 10001 种意外来告诉你一万与万一之间的区别。所以说这种异常处理方式根本不能适应现代编程的要求。除此以外，程序中使用大量的判断语句也不利于代码的阅读。

6.2 异常对象

在了解异常处理之前我们先来介绍异常对象。Delphi 将异常封装成类，触发异常实际上变成了创建特定类的对象并将这个对象提交给相关的的代码进行处理。这个对象中通常含有一个字段用于描述异常产生的原因，如前面提到的文件不存在、系统错误等。还含有一个字段用于描述异常标识符，利用这个标识符可以通过网络获悉异常的详细信息。其它还定义了各种方法用于通知异常的各种信息。

想想为什么要将异常封装成对象？答案是提交一个包含多条信息的独立的对象要比提交一大堆零散的信息方便得多，也直观得多。这就好比我们向某个人的信息交给其它人，你是选择交付十几张零散的纸条还是一份档案袋？异常对象就相当于一份档案袋，其中蕴含了很多具体的信息。

Delphi 中所有的异常对象均继承自定义于 System 单元中的 Exception 类。Exception 本身直接继承自 TObject 类:

```
Exception = class(TObject)
private
  FMessage: string;
  FHelpContext: Integer;
  ...//其它成员省略
end;
```

其中的 FMessage 用于异常产生的信息，FHelpContext 用于描述异常的唯一标识符我们可以从这个类派生我们自己的异常类。习惯上异常类以大写字母"E"开头。

Delphi 的 SysUtils 单元中定义了一些常见的异常类。基本上，这些异常类涵盖我们日常编程所能遇到的绝大部分异常，所以一般情况下读者并不需要自己自定义异常类。

Delhphi 在 SysUtils 单元中定义了大量的预定义异常类，Delphi 的帮助文档中详细描述了这些异常类，需要时查阅即可，在此我们不再赘述。

接下来我们简要介绍一下如何定义一个自己的异常类。

6.2.1 自定义异常类

虽然 VCL 提供的异常类基本能满足日常编程中的各个方面，但有时候相比这些预定义异常类，我们更喜欢自己来定义异常类。

定义一个异常类也是非常之简单，我们只需要从预定义的 Exception 类派生即可，例如下面的代码就定义了一个名为 EMyException 的异常类:

```
type
  EMyException = class(Exception)
  end;
```

当然，您可随着需要在其中加入任意的成员，例如，可以在其中添加一个构造函数及一个异常 ID：

```
type
  EMyException = class(Exception)
    FMessage:Integer;
    Constructor Create(ID:Integer);
  end;
```

再次声明，异常类中所有的成员没有任何的限制，只要有需要，您可以在其中添加任意成员。至于其中的原因，接下来自会揭晓。

我们前面说过，异常对象归根结底只是一个标志，说明某个地方出现了一个异常。事实上，任何一个对象均可以作为这个标志，但 Delphi 中约定：只使用 Exception 的派生类的对象标志一个错误。在下面的例子中我们就使用了一个 TObject 的对象作为异常对象：

```
uses
  SysUtils, Dialogs;
var
  i:Integer;
begin
  readln(i);
  try
    writeln(9/i);
  except
    on TObject do ShowMessage('产生了一个 TObject 对象');
  end;
  readln;
end.
```

虽然理论上我们确实可以使用作意一个对象代码异常对象，但出于种种原因我们强烈的不推荐这种做法。所以对于这种无意义的行为，还请读者不要浪费时间去尝试。

现在，读者该明白为什么异常对象的成员可以任意定义：既然异常对象仅仅只是一个标志，它最重要的就只是它的名称，至于其中有什么成员则完全无关紧要。但既然如此，为什么 Delphi 中的预定义异常类中还有那么多的成员？答案也很简单：您知道哪个地方出现了错误，您想不想知道出现错误的原因呢？作为一个对象，您又想不想定义一个构造函数以便创建这个对象时执行一些额外的行为呢？基于这些原因，异常对象必须要定义一些成员以方便应用。查阅 Exception 的源代码也可知，其中定义的成员大部分都是与构造这个对象及错误的具体信息有关。

下面的例子中完全实现了 EMyException 类并在代码中使用这个异常类：

```
program Project2;

{$APPTYPE CONSOLE}

uses
  SysUtils, Dialogs; //添加 Dialogs 单元
type
  EMyException = class(Exception)
    FMessage:Integer;
    Constructor Create(ID:Integer);
  end;

constructor EMyException.Create(ID: Integer);
begin
  ShowMessage('EMyException 被创建,其 ID 为'+Inttostr(ID));
end;
```

```
var
  i:Integer;
begin
  readln(i);
  try
    raise EMyException.Create(1001);
  except
    on EMyException do ShowMessage('产生的异常对象已经被处理');
  end;
  readln;
end.
```

6.3 异常处理语句

异常处理语句由两部分组成，一部分专门用于检测代码中是否存在异常，另一部分专门用于处理检测到的异常。Delphi 提供了两种专用于处理各类异常的语句，一是 try...except...end 语句，另一类是 try...finally...end 语句。

6.3.1. try...except...end 语句

Try...except...专用于处理各类异常。其形式为：

```
try  语句 1  except  语句 2  [else 语句 3]  end;
```

其中[else 语句 3]为可选部分。

这类语句的执行过程为：

1. 执行语句 1，若没有任何异常产生，则执行完语句 1 后，直接执行 end 后的语句；
2. 若执行语句 1 时产生了异常，则绕过语句 1 中剩下的还未执行的部分，直接跳至语句 2；
3. 跳至语句 2 后会在语句 2 中寻找合适的处理语句，若未寻找到则招待 else 部分
4. 若 else 部分缺省，则当前 try...except...无法处理这个异常，程序会将这个异常提交到上一层的 try...except...语句进行处理
5. 若上一层的 try...except...中还是不能处理这个异常，则将这个异常提交到更上一层的 try...except...，依此类推，直至程序的最外层的 try...except...
6. 若最外层还是不能处理，则提交给 Delphi 自动插入的异常处理语句，这样做的结果通常是程序强行中断并退出。

注意第 3 步，程序会在语句 2 中寻找合适的处理语句。问题是，究竟什么才是合适的处理语句？我们来看一下以上三部分语句。

首先，语句 1 是我们本来就要执行的普通代码，这个应该没什么问题。

现在，我们看一下语句 2。语句 2 有两种形式：

第一种形式就是普通的代码，其中调用若干个例程，这些例程通常是诸如 exit、break 之类负责程序退出或直接将异常提交到其它某个位置。当异常被提交给这种语句时，若第一句代码能够处理异常（退出程序或是将异常提交），则此段就算合适。若第一句代码不能搞定，异常会直接提交给 else 部分，若没有则直接跳到上一层的

try...except...。

第二种形式有点像选择语句，其形式为：

```
 except
   on obj1：type1    //obj 表示
     do 语句 1;
   on obj2：type2
     do 语句 2;
      ...
```

记住前面所说的：所有的异常均是类类型的对象。type1、type2 等为各异常类的名称。obj1、obj2 表示异常对象的名称，它可以省略，当然如果你省略了这个名称，请记着把后面的冒号也一并省略。

为了描述的方便，我们假设异常对象所属的类的名称为 TE。这种形式的语句执行过程为：

1. 依照出现顺序，程序先将 TE 与 type1 比较，若 TE 是 type1 的派生类或 type1 本身，则执行语句 1；若不是则进行下一步比较；
2. 将 TE 与 type2 比较，若 TE 是 type2 的派生类或 type2 本身，则执行语句 2；若不是则进行下一步比较；

   ...
3. 这种比较会在找到合适的 type 时停止。若直到最后一个语句也未能找到，则异常被提交给 else 部分。同样，若 else 部分缺省，异常会直接被白净净给上一层的 try...except...语句。

最后，我们再来看一下语句 3，else 部分中的语句在异常被提交到 else 部分时会被全部执行，除非其中夹杂了诸如 exit 或 break 之类的能使程序中止的语句。

下面我们通过一个例子来熟悉一下 try...except...语句：

```
program Project1;
{$APPTYPE CONSOLE}

uses
  SysUtils, Dialogs;
// 添加 Dialogs 单元
type
  T1 = class
  end;

var
  i: integer;
begin
  writeln('请输入数字 i 的值');
  read(i);
      try
        writeln(inttostr(9 div i));
        // 将 9 除以 i 并显示计算结果，当 i 等于 0 时会产生一个异常
      except
```

```
        on TObject do
          showmessage('this is TObject');
        on T1 do
          showmessage('this is Exception');
      else
        showmessage('this is else');
      end;
  readln;
end.
```

运行后请输入数字 0 并按下回车键。此时 i 等于 0，运行 writeln(inttostr(9 div i))时这个值会造成一个异常。（在按下 F9 后若出现一个了标题为"Debugger Exception Notification"的对话框，其中有三个按键，请读者务必按下其中的 Continue 按钮）。

异常产生后程序立即会执行 except...部分的语句，由于产生的异常对象是 Exception 的子类，而 Exception 继承自 TObject，所以 exception...中的第一个选项就符合条件，所以程序会直接执行其后的处理代码：显示一个对话框显示'this is TObject'。这时不管其它选项如何均不会执行。执行完这段执行代码后，程序会直接执行 end 后的代码：readln。

读者可去掉例中粗体部分的代码，然后再次输入 0 并回车，这一次，显示的对话框中显示的不再是'this is TObject'，而是'this is else'。这是因为粗体部分被去掉后剩下的 T1 与异常类型并不兼容，所以程序会执行 else 部分的语句。

6.3.2. try...finally...end 语句

try...finally...的作用是让程序在出现任何异常时都能像正常执行一样结尾。它与 try...except...语句的区别是：前者的 try...部分产生异常时会直接跳到 finally...部分，提前执行这部分代码，也就是说 finally...部分的代码一定会被执行，无论 try...部分有没有产生异常。try...finally...中无论是 try...部分还是 finally...部分的语句没有任何特殊要求，只要是正常语句即可。

在功能方面，try...finally...与其说是异常处理语句，不如说是程序保险语句更为恰当。当 try...部分发生异常时，except...部分会试图去解决这个异常，而 finally...部分则直接跳过这段代码。以包装礼品为例，try...except...在包装时若发现礼品出现了问题，它会启动一个预案来解决这个问题，然后才给这份礼品包上正常的包装；而 try...finally...则偷懒的多，当发现礼品有问题时，它立即放弃接下来的工作，直接给这份问题礼品包上正常的包装。这个例子用程序语言可描述如下：

try...except...	try...finally...
try 　对礼品进行清洗之类的准备工作 except 　尝试去解决礼品出现的问题 end; 包上包装	try 　对礼品进行清洗之类的准备工作 finally 　直接执行最后一步（包上包装） end;
示例	
try	try

obj := TObject.create;
//出现了异常
except
//解决异常
end;
FreeAndNil(obj);
	obj := TObject.create;
	//出现了异常
	finally
	FreeAndNil(obj);
	end;

我们再借助例子来熟悉一下 try...finally..语句：

```
uses
  SysUtils, Dialogs; // 添加 Dialogs 单元
var
  i:integer;
begin
  try
    writeln('请输入数字 i 的值');
    read(i);
    writeln(inttostr(9 div i));//若 i＝0，则发生异常
    ShowMessage('try section');
  finally
    ShowMessage('finally section');
  end;
end.
```

程序开始运行时会要求输入一个数字值为 i 的值，若输入一个不为 0 的数字，程序会依次显示两个对话框；而若读者者输入 0，try...finally 部分会发生一个异常，此时 try... 部分的最后一条语句不会执行，所以程序只会显示一个对话框显示'finally section'。

在使用 try...finally...类型的异常处理语句时，try...finally 部分中执行的 exit 例程也会被当成一个异常来处理。读者若不明白这句话是什么意思，可以看下面的例子：

```
uses
  SysUtils, Dialogs;  //必须添加 Dialogs 单元
begin
  try
    Exit;
    write('请输入 I 的值');
    readln(i);
    writeln(9 div i);
  finally
    showmessage('Delphi');
  end;
end.
```

程序执行后会直接运行 finally...部分中的语句，显示一个对话框，对话框上显示的文字为'Delphi'。可以看到，此例中执行的 exit 像普通异常一样，仅仅只退出了 try...finally 部分而并未退出整个的程序块。

6.4 手动触发异常

某此情形下，被检测的代码可能没有出现错误或出现的错误已经被处理，但出于种种原因我们需要在这些代码中强行触发一个异常，这种情形下可以使用特定的代码手动触发一个异常。Delphi 使用关键词 raise 来达到这个目的。

类似于 inherited，raise 可单独作为一条语句，也可在其后接一个对象。当 raise 单独作为语句时，它会将目前正在处理的异常重新提交给外层的异常处理语句；当 raise 后接一个对象时表示将这个对象提交给外层的异常处理语句。

```
begin
  writeln('请输入数字 i 的值');
  read(i);
  try
    try
      writeln(inttostr(9 div i));
    except
      showmessage('Inner Except');
      raise Exception.Create('error');
    end;
  except
    showmessage('Outer Except');
  end;
end.
```

运行后会以连续显示两个对话框，分别显示"Inner Except"及"Outer Except"。原因是内层的处理语句处理处理了异常后，raise 语句会将当前的异常提交给外层，所以外层的处理语句也得到了执行。将粗体部分的 raise 语句去掉再运行，可发现只有一个对话框会弹出，显示"Inner Except"。

实际应用中处理一个异常时，经常会由于各种原因而无法完整的处理这个异常，此时可先处理这个异常的一部分，然后利用 raise 语句将其提交给外层的处理语句以处理剩下的部分：

```
program Project1;
{$APPTYPE CONSOLE}

uses
  SysUtils, Dialogs;

procedure ReRaiseErr( i:Integer);
begin
  try
    writeln(5 div i);
  except
    ShowMessage('Run Error');
    raise;
  end;
```

```
end;

begin
  try
    ReRaiseErr(0);
  except
    ShowMessage('Error in ReRasieErr');
  end;
  readln;
end.
```

在这个例子中，将 0 作为参数值传递给 ReRaiseErr，这会导致其中出现一个异常，这个异常会被 ReRaiseErr 中的异常处理语句捕捉并处理，所以会弹出一个对话框显示'Run Error'，但这个处理语句中使用了 raise，它会将这个异常提交至上一层，所以在主程序中的异常处理语句也会捕捉这个异常并进行处理，处理的结果就是显示一个对话框，显示字符串'Error in ReRasieErr'。

最后请读者注意两句话：

- 在需要时，raise 语句可放置于程序的任何地方。
- raise 语句只能放在异常处理语句中。

请注意，这两句话并不矛盾。Delphi 的编译器会自动处理，将整个程序全部包括进一最外层的异常处理语句，换句话，程序中的任何一处皆被笼罩在一个大的异常处理语句中，所以 raise 可以置于程序中的任何一处。

6.5 Abort 语句

Delphi 提供了 Abort 例程，它可以在内部自动创建了一个 EAbort 异常类的对象并提交给上一层的异常处理语句。EAbort 直接继承于 Exception，它被称为静默异常类，因为程序在引发此类的异常时不会显示任何对话框来提示用户。换而言之，EAbort 类的异常在引发时不会对程序有任何干扰。

在需要使用 raise 手动触发一个 EAbort 类的异常时可直接使用 Abort 语句代替：

```
uses
  SysUtils, Dialogs; // 添加 Dialogs 单元

begin
  writeln('请输入数字 i 的值');
  read(i);
  try
    try
      writeln(inttostr(9 div i));
    except
      on Exception do
        Abort;
    end;
  except
```

```
    on EAbort do
      showmessage('EAbort Exception');
  end;

end.
```

6.6 套嵌的异常处理语句

异常处理语句可以套嵌。Delphi 中有两种例程处理请问，理论上应该有 4 种套嵌，但实际上只要读者掌握了每种异常处理语句的执行过程，任何套嵌均不是问题。签于此，我们不会浪费一大堆笔墨介绍一些很浅显的东西。本节我们只介绍两个 try...except...套嵌的情形。之所以单单介绍这种情形，一是因为这种情形最具代表性，二者我们打算通过这种情形介绍一个概念：异常对象的丢失。

其形如下：

```
try
  try...except... end;
except
...
end;
```

在这种套嵌语句中，只有内层的语句处理不了异常或者我们在其中手动重新触发了异常时，这个异常才会被交给外层的 try...except...语句进行处理。看下面的例子：

```
...//前面内容与上个例子完全一样
begin
  writeln('请输入数字 i 的值');
  read(i);
  try
    try
      writeln(inttostr(9 div i));
    except
      on T1 do//由 TObject 变成了 T1
        showmessage('this is TObject');
    //else
      //
    end;
  except
    showmessage('Outer Except');
  end;
  readln;
end.
```

请读者再一次输入 0，由于内层的 try...except...中的 except...部分中的 on 语句中的类型变成了 T1，这个类与内层 try...部分的语句所产生的异常所属的类不兼容，所以这条处理语句不能处理异常。异常将被提交到外层的 try...except...中进行处理。所以运行的结果是显示对话框，其显示的内容为"Outer Exception"。

现在请读者将 else 前的双斜杆去掉再运行，此时程序不会显示任何对话框，因为内层

的 try...except...无法处理的异常被提交给了 else 部分的语句，尽管这部分语句不能处理，异常还是没有提交到外层。这个例子中读者是否对 else 部分的作用有了更深一步的了解呢？

套嵌的异常处理并非这么简单。有时候这种套嵌可以会导致内层中本应提交的异常不能被提交到外层。请看下例：

```
type
  E1 = class(Exception)
  end;

var
  i: integer;

begin
  writeln('请输入数字 i 的值');
  read(i);
  try
    try
      writeln(inttostr(9 div i)); //语句 X
    except
      raise E1.Create('Error Message');
    end;
  except
    on E1 do  //注意此处是 Exception，这条语句如何接收 E1 类型的异常？
      showmessage('Outer Except');
  end;
end.
```

这个例子执行后，会显示对话框显示"Outer Except"。读者有没有觉得有什么奇怪之处？不觉得？没关系。我们慢慢来看。

首先，我们声明了一个异常类 E1，它没有声明任何成员。就算如此，E1 也算是 Exception 的子类，虽然可以将 E1 的对象赋给 Exception 的对象，但反过来赋值却不可能。了解这个前提我们继续往下。

在内层的 try...except...语句的 except...部分，我们仅仅只是重新触发了异常。理论上，外层的处理语句应该接受这个异常并进行处理，事实也正是如此。

但问题就出在这里。在不查阅资料的情况下，我们可能并不记得语句 X 中一个整数除以 0 时所引发的异常对象应该属于哪个类，但我们最起码知道两点：

第一，这个对象一定是 Exception 的派生类，因为它是一个异常对象；

第二，这个对象一定不是 E1 对象，因为我们并没有创建 E1 的任何对象；也不是 E1 的派生类的对象，因为我们并没有声明任何一个 E1 的派生类。

读者应当还记得 except 语句中 on type do...类型的语句应当具备的条件：异常对象所属的类应当是 Type 本身或 Type 子类。

我们手动触发了 E1 类型的异常，而语句 X 中产生的异常对象又不可能是 E1 或 E1 派生类的对象，那么外层 except 部分是如何利用 E1 类型接收这个对象的呢？

答案是当我们使用 raise 提交了一个 E1 类型的异常对象时，这个对象会被外层的 try...except...语句捕捉并处理，所以 on E1 do...部分才能捕捉到这个异常对象，因为

它正是 E1 类型的对象。而与此同时，语句 X 中产生的对象被自动销毁。

纵观整个过程，语句 X 中产生的异常在某个地方丢失，仿佛一开始产生的异常就是 E1 的对象。Delphi 中赋予了这种现象一个名称：异常对象的丢失。按英文翻译过来应该是异常的丢失。但中文中的异常本身就是一个形容词，如果读者一听到“异常的丢失”就认为其中的异常是名词而非形容词，那只能证明您的脑袋非常的异常，或是您已经洞悉了异常。

第九章 运算符重载

先来了解两个概念：操作符与操作数。操作符是用于完成某种功能的符号，如加号(+)、乘号(*)等，也称为运算符；操作数指操作符操作的对象，如表达式 4+5 中的 4、5 就是操作数。应当注意：某些操作符在使用时与函数非常相似，如 Round、Inc、Addr 等。

再来回顾一下函数的定义，所谓函数是能够完成一定功能并返回一个值的代码的集合。按照这个定义，我们常见的一些运算符如也可以被划到函数之列。以加号为例，它的功能是将两个操作数相加，并将结果作为返回值返回。

继续回顾一下函数的重载，所谓函数的重载是指为同一个函数名称定义多种不同的实现，调用时系统会根据特征集调用匹配的版本。

停止回顾。现在我们来对运算符重载作一个大概的定义。运算符重载是指对运算符进行重载使得其能够按我们定义的方式对本来不能运算的操作数进行运算。举例来说，加号(+)本来只能用于一些简单类型而不能用于 TObject 对象（即两个 TObject 类型的对象不能进行相加），但我们可以对加号(+)进行重载使得它可以用于此类对象。

9.1 认识运算符重载

Delphi 中所有的运算符都对应着一个函数名（每个运算符都是一个函数），加号(+)的名称是 Add，减号(-)的名称是 Subtract。当我们在源代码中调用这些运算符时实际上是在变相的调用这些函数。如表达式 2+3 相当于 Add(2,3)。但在源代码中我们只能使用这些运算符的符号而不能使用函数名。

运算符的重载实际上是在重载这些函数。但与一般函数不同，Delphi2010 目前只允许重载记录类型的操作符。同时，Delphi 也对可重载的操作符作了一些限制，使得只有一部分运算符能够被重载。下表列出了这些运算符：

操作符名称	种类	对应的函数原型	对应的符号
Implicit	转换操作符	Implicit(a:type):resultType;	隐匿转型
Explicit		Explicit(a:type):resultType;	显式转型
Negative	一元操作符	Negative(a:type):resultType;	-
Positive		Positive(a:type):resultType;	+
Inc		Inc(a:type):resultType;	Inc
Dec		Dec(a:type):resultType	Dec
LogicalNot		LogicalNot(a:type):resultType;	not
BitwiseNot		BitwiseNot(a:type):resultType;	not
Trunc		Trunc(a:type):resultType;	Trunc

Round		Round(a:type):resultType;	Round
Equal	比较操作符	Equal(a:type; b:type):Boolean;	=
NotEqual		NotEqual(a:type; b:type):Boolean;	<>
GreaterThan		GreaterThan(a:type; b:type) Boolean;	>
GreaterThanOrEqual		GreaterThanOrEqual(a:type; b:type):resultType;	> =
LessThan		LessThan(a:type; b:type):resultType;	<
LessThanOrEqual		LessThanOrEqual(a:type; b:type):resultType;	< =
Add	二元操作符	Add(a:type; b:type):resultType;	+
Subtract		Subtract(a:type; b:type):resultType;	-
Multiply		Multiply(a:type; b:type):resultType;	*
Divide		Divide(a:type; b:type):resultType;	/
IntDivide		IntDivide(a:type; b:type):resultType;	div
Modulus		Modulus(a:type; b:type):resultType;	mod
LeftShift		LeftShift(a:type; b:type):resultType;	shl
RightShift		RightShift(a:type; b:type):resultType;	shr
LogicalAnd		LogicalAnd(a:type; b:type):resultType;	and
LogicalOr		LogicalOr(a:type; b:type):resultType;	or
LogicalXor		LogicalXor(a:type; b:type):resultType;	xor
BitwiseAnd		BitwiseAnd(a:type; b:type):resultType;	and
BitwiseOr		BitwiseOr(a:type; b:type):resultType;	or
BitwiseXor		BitwiseXor(a:type; b:type):resultType;	xor

注：表中 BitwiseXXX 表示位运算中的运算符。LogicalXXX 表示逻辑运算符。

9.2 如何重载运算符

前面说过，重载运算符的实质是重载某记录类型的类方法。所以重载运算符可使用以下方法声明：

```
type
  RecName = record
      class operator conversionOp(a: type): resultType;
      //转换运算符
      class operator unaryOp(a: type): resultType;
      //一元运算符
      class operator comparisonOp(a: type; b: type): Boolean;
      //比较运算符
      class operator binaryOp(a: type; b: type): resultType;
      //二元运算符
  end;
```

重载运算符函数使用 class operator 声明，RecName 表示自定义的记录类型的名称。

以上四个成员函数分别重载了转换运算符、一元运算符、比较运算符、二元运算符。注意在以下四种类型的重载函数的参数中：

重载转换运算符时，参数 a 或返回类型有且只能有一个是 RecName 类型，另一个是其它类型。它表示将参数 a 的类型转换成 resultType 类型。

重载一元运算时，参数 a 必须是 RecName。返回值的类型任意。但对于 Inc 与 Dec，返回值类型必须也为 RecName 类型。

重载比较运算时，ab 两个参数至少有一个是 RecName 类型。返回值为 Boolean 类型。

重载二元运算时，ab 至少有一个是 RecName 类型，返回值类型任意。

在重载比较及二元运算符时，参数的顺序决定了在运算时操作数顺序。不同的操作数的顺序会调用不同的运算符函数版本。如我们重载 Add 操作符：

```
type
  T1 = record
class operator Add(a:T1; b:integer):integer;
  end;
var
  obj:T1;
  i:Integer;
begin
  i :=  obj + 8;  //合法
  i :=  8 + obj;  //不合法
end;
```

当编译器在计算 8+obj 时，它会寻找如下版本的函数：

```
  class operator Add(a:integer; b:T1):integer;
```

但我们并没有重载这个版本，所以编译器提示错误。

不同的返回值类型也会导致调用不同的版本：

```
var
  i:byte;
  obj:T1;
begin
  i :=  obj +8;
  obj :=  obj +8;//错误
end.
```

在计算 obj := obj+8 时，编译器会寻找如下版本：

```
  class operator Add(a:T1; b:integer):T1;
```

这个版本我们也没有定义，所以编译时会提示错误。

运算符重载使用非常的方便，适当使用可以使代码更为简洁而直观。但可惜的是 Delphi 中的运算符重载功能不是足够的强大，与 C++等其它语言相比依然存在一些差距。希望在下一个版本中有所改进。

最后我们通过一个完整的示例结束本章的内容。

```
 type
   T2  = record
     i:integer;
     s:string;
   end;
   T1 = record
     i:integer;
```

```
    s:string;
    class operator Implicit(a:integer):T1;
    class operator Negative(a:T1):string;
    class operator Equal(a:T1; b:String):Boolean;
    class operator Subtract(a:T1; b:integer):T1;
  end;

class operator T1.Equal(a: T1; b: String): Boolean;
begin
  if a.s = b then
    result :=  True;
end;

class operator T1.Implicit(a: integer): T1;
begin
  result.i :=  a;
  result.s :=  '';
end;

class operator T1.Negative(a: T1): string;
begin
  result :=  a.s;
end;

class operator T1.Subtract(a: T1; b: integer): T1;
begin
  result.i :=  a.i - b;
  result.s :=  a.s;
end;

var
  obj:T1;
begin
  obj.i :=  100;
  obj.s :=  'Delphi';

  if obj = 'Delphi' then writeln('True');

  obj :=  T1(78);
  writeln(obj.i);   //obj的成员值变为78,''

  obj.s :=  'Delphi';  //obj的成员值变为78,'Delphi'
  writeln(-obj);
```

```
  writeln((obj-47).i);

  readln;
end.
```

第十章 泛型

在正式开始本章的内容之前，我们先利用前面尝过的知识来解决一个小问题。我们知道，Delphi 中所有的字符串均可以被索引，现在由于某种原因，我们需要在程序中取出某个字符串中的所有奇数位的字符并将这些字符组成一个新串。例如'ABCDEFG'经过上述处理后会得到'ACEG'。整个过程可用如下代码表示：

```
var
  s1,s2:string;
  i:Integer;
begin
  s1 :=  'ABCDEFGHI';
  for i :=  1 to length(s1) do
    s2 :=  s2+s1[2*i-1];
  writeln(s2);
end.
```

这段代码特异性针对于字符串'ABCDEFG'而言，若是要求取'HIJKLMN'的奇数位，我们可写如下代码：

```
var
  s1,s2:string;
  i:Integer;
begin
  s1 :=  'HIJKLMN';
  for i :=  1 to length(s1) do
    s2 :=  s2+s1[2*i-1];
  writeln(s2);
end.
```

……

读者可能会疑惑，这些代码非常相似，为什么不将它们写成一个函数然后需要时将字符串当成参数传递呢？这样一来上面的两段代码可写成如下形式：

```
function GetStr(s:string):string;
var
  str:string;
```

```
  i:Integer;
begin
  for i :=  1 to length(s) do
    str :=  str+s[2*i-1];
  result :=  str;
end;

var
  s1,s2:string;
  i:Integer;
begin
  writeln(GetStr('ABCDEFG'));
  writeln(GetStr('HIJKLMN'));
  readln;
end.
```

这样一来，确实是方便了很多，从各个方面来看，这种方法都是非常好的选择。这种方法可用语言描述就是：提取不变的部分，将变化的部分当成一个变量。用上面的例子来说，除字符串之外的其它均没有变化，所以我们提取这些部分封装成一个函数，并将字符串当成变量传递给这个函数。说得更玄乎一点，这是一种以不变应万变的抽象方法，前面的重载函数以及多态就是这种思维的杰出代表。本章，我们将学习另一种典型代表。依照惯例，我们先从一个示例开始。

```
type
  TIntegerSample = record
  strict private
    F1:Integer;
    procedure SetF1(value:Integer);
    function GetF1:Integer;
  public
    property P1:Integer read GetF1 write SetF1;
  end;
```

以上我们定义一个 TIntegerSample 类，其中定义了属性 P1 及 P1 的读写方法 SetF1、GetF1 与存储字段 F1。为了省去创建及销毁对象的麻烦，我们直接使用了 record 代替 class。可以看到，TIntegerSample 中的属性类型是 Integer 类型，但现在由于某种原因，我们想定义一个 string 版本的 TStringSample，于是我们定义如下：

```
type
  TStringSample =  record
  strict private
    F1:String;
    procedure SetF1(value:String);
    function GetF1:String;
  public
    property P1:String read GetF1 write SetF1;
```

```
  end;
```

对于比 TIntegerSample 可以发现，TStringSample 中仅仅只是 Integer 变成了 String。就为了这小小的一点变化，我们需要重新编写整个的代码，然后还要将函数的实现代码中的 Integer 逐个手动变成 String。若我们还要定义 TRealSample，我们还要复制整个代码，然后逐个修改。

很明显，这个例子中我们编写了大量的相同的代码。这与面向对象的理念背道而驰，所以我们需要寻找一种更好的、可以免于编写大量相似代码的方法。

再来回顾一下这个例子，我们会发现不同版本的 TSample 间差别非常之小，仅仅只是数据类型不同而已。根据前面提到的抽象思维的方法，我们是否可以定义某种类似于函数的东西，然后将不同的数据类型名称当成参数传入以获取不同版本的 TSample 呢？这样在使用时我们通过传递不同的数据类型就可获得不同版本的 TXXXSample。

答案是 Yes。我们有这样一种技术，其名为——泛型，也称为通用类型。利用这种技术，我们可以重新声明 TSample，其中的数据类型统一用一个标识符（习惯上使用大写字母 T）来表示，TSample 声明如下:

```
type
  TSample<T> = record
  strict private
    F1:T;
    procedure SetF1(value:T);
    function GetF1:T;
  public
    property P1:T read GetF1 write SetF1;
  end;
```

可以看到 TSample 的名称之后加上了一对尖括号，其中放置了一个类型参数 T，在声明 TSample 成员时，我们完全可以将 T 当成一个正常的数据类型来使用。在使用 TSample 时，通过传递 T 的具体类型，我们可以得到不同版本的 TXXXSample。这里的 T 被称为类型参数，其具体值只能是数据类型的名称。通过这种方式，TSample<T>拥有了可随着 T 值的不同而表示不同的数据类型的特性，如 TRealSample、TStringSample 等，所以 TSample<T>被称为泛型（或通用类型），意欲其能表示多种类型。

现在我们对 TSample 中的两个方法进行定义：

```
function TSample<T>.GetF1: T;
begin
  Result :=  self.F1;
end;

procedure TSample<T>.SetF1(value: T);
begin
  self.F1 :=  i;
end;
```

这两个方法的定义与普通的定义没什么不同，我们只需将 T 当成是一种编译器知道而我们

不知道的数据类型即可。

下面再定义命令行程序的主程序块：

```
var
  IntegerObj:TSample<Integer>; //声明 TSample 的 Integer 版本
  StringObj:TSample<String>;   //声明 TSample 的 String 版本
begin
  IntegerObj.P1 :=  89;
  StringObj.P1 :=  'Delphi';
  Writeln(IntegerObj.P1);
  Writeln(StringObj.P1);
  Readln;
end.
```

感觉如何？是否感觉比手动逐个将 Integer 修改成 String 要强得多呢？

到了这里，读者对泛型编程是不是有了点感觉呢？所谓泛型編程是利用类型参数取代具体的类型进行编程，在使用时将一个具体的数据类型的名称作为参数传递。正如上例中使用 TSample 时将 Integer 与 String 作为参数传递一样。

10.1 声明泛型类型

所有使用了类型参数的数据类型（如 TSample<T>）称为泛型类型。从理论上来说，任何结构类型均可以使用泛型。这点有个例外，作为一种特殊的数据类型，普通的全局泛型例程或是例程指针中不能含有类型参数，但类方法可以。接下来，我们简要叙述一下如何在一些常见数据类型中声明类型参数。

首先我们从类类型开始，声明一个泛型类（或者说在类中声明类型参数）的语法为：

```
 type
   类名<类型参数 1，类型参数 2…> ＝ class(父类)
   ...
   end;
```

其中的类型参数可以使用任何有效的标识符，Delphi 中习惯使用大写字母 T。

如同 protected 成员一样，类的类型参数只能用于当前类或其派生类中。但反过来，类中使用的类型参数却不一定是类名称后指定的参数，因为类成员可以声明自己的类型参数。下面的例子中声明了 TSample 类，其成员全部声明了自己的类型参数：

```
type
  TSample<T> = class
  private
  type
    TInnerRec<R> = record
```

```
      i:R;
end;
    TQux<T> = class
      X: Integer;
    end;
  public
    FSample:T;
    function fun<F>(value:F):F;
  end;
```

这里先说明一下类型参数的声明与使用的区别。标识符后的尖括号中的类型参数表示声明，其它的表示使用，如此例中 TInnerRec<R>中的 R 表示声明，而其中的 i：R 表示使用，将 i 声明成 R 类型。显而易见，类型参数也必须先声明后使用。所以，类中的方法、数据类型等可以声明自己的类型参数，但普通的变量或常量却不能这样做，对于这两种成员而言，它们只能使用而不能声明类型参数。

类型参数是一个局部化的标识符，所以 TSample<T>中的 T 只能用于这个类中，TInnerRec<R>中的 R 也只能用于这个记录中，同理 F 参数也只能用于 TSample<T>.fun<F>当中。

在泛型类声明了其它的结构类型如记录和类时，这些结构类型中声明的类型参数只能用于自身的成员，不能用于自身以外。当这些参数与外界的参数名称相同时，它们会屏蔽外界参数。以 TSample<T>为例，其中声明了套嵌类 TQux<T>，这个 T 参数只能用于 TQux<T>中，由于这个参数的名称与 TSample<T>中的类型参数的名称相同，所以 TSample<T>中的类型参数在 TQux<T>被屏蔽。

总而言之，类型参数的有效范围等同于当前范围中声明的局部变量。

从某个泛型类派生它的子类时，这个父类可以有二种形式：带有形参的泛型类、带有实参泛型类。下面例子中，T2<T>的父类是一个带有形参的类：T1<T>，T3<T>的父类则是一个带有实参的类：T1<Integer>。

```
 Type
   T1<T> = class(TObject)
     function M1:T;
   end;

   T2<T> = class(T1<T>) //也可写成 T2<A> = class(T1<A>)
   end;

   T3<T> = class(T1<Integer>)
   end;
```

注意类型参数的标识符不作为类的识别依据，所以在第二种情形中的 class(T1<T>)中的 T 可以是任何标识符，只要它和子类中的类型参数采用了同一标识符即可,如注释中所示。根据此规则，T2<T>不能声明成如下形式：

```
Type
```

```
T1<T> = class(TObject)
    function M1:T;
  end;

  T2<A> = class(T1<T>)  //T2<A>与T1<T>中的类型参数必须使用同一标识
  end;
```

泛型接口与泛型记录的声明非常类似于泛型类的声明。在此只给出一些示例：

```
type
  R1<T> = record
    F:T;
    function M1<M>(v1:M; v2:T):M;
  end;

  I1<T> = Interface
  end;

  I2<T> = Interface(I1<T>)
  End;

  I2<T> = Interface(I1<Integer>)
  End;
```

泛型例程指针的声明：

```
type
  fun<T> = function(s:T):T;
```

其中的类型参数可以出现在方法或例程的参数列表中，也可以作为返回值的类型。泛型例程指针所指向的例程必须满足两个条件：

- 参数与返回值的类型的内部结构必须相同，当然，有返回值的函数指针不能指针无返回值的过程，反之亦然。如下面的例子所示：

```
type
  fun<T> = function(s: T): T;
  MyType = Integer;

function M1(value:Integer):MyType;
begin
end;

function M2(value:integer):Integer;
begin
```

```
end;

procedure M3(value:Integer);
begin
end;

var
  pF:fun<Integer>;
begin
  pF :=  M1;  //正确
  pF :=  M2;  //正确
  pF :=  M3;  //错误，M3 没有返回值
end.
```

- 参数列表必须与泛型例程指针声明时的形式一一对应，具体来说有二方面：一者相应参数的类型必须具有相同的内部结构，二者相应参数的传参方式必须相同。如下面例子所示：

```
type
   fun<T> = function(s1:T; s2:Integer): T;
   MyType = Integer;

function M1(v1:Integer; v2:Integer): Integer;
begin
end;

function M2(v1:Mytype; v2:Mytype): Integer;
begin
end;

function M3(var v1:Integer;v2:Integer): Integer;
begin
end;

var
   pF:fun<Integer>;
begin
  pF :=  M1;  //正确
  pF :=  M2;  //正确
  pF :=  M3;  //错误，M3 的 v1 采用了 var 传参数方式
end.
```

10.2 泛型的实例化

所谓泛型的实例化是指用一个具体的数据类型作为类型参数的值使得泛型表示一个具体类型。在本章开头，当我们向 TSample<T>中的 T 设置为 String 后我们就获得了 TStringSample 类型，这个过程称就是一个典型的实例化，将 TSample<T>实化成了 TStringSample。。

在前面我们曾经介绍过多态的概念。程序在运行时会动态的判断对象的实际类型，然后在调用相应版本的方法，由于这种多态依赖于运行期间的动态类型判断，故而称之为运行时多态。读者若有疑惑，可复习前面的内容。

与运行时多态不同，实例化是另一种形式的多态，我们称之为编译时多态。以前面的 TSample 为例，它可接受多种类型如 Integer、String 等类型名并产生相应的版本。不同于运行时多态，编译时多态并非在运行期间动态判断对象的实际类型。我们在前面说过，泛型省去了我们编写大量雷同代码的麻烦，但站在编译器角度，泛型非但没有减轻反而加重了编译器的负担，因为编译器必须代替我们去编写那些雷同的代码。从这个角度来看，我们使用泛型所得到的轻松是以加剧编译器的负担为代价的。

所以说，泛型实例化的实质是编译器在编译时根据用户的需要自动地编写了多个版本的代码，然后将这些代码编译并嵌入目标程序中。整个过程中，编译器替我们编写了代码，其它与一般程序没什么不同。从代码量的角度来看，运行时多态只有一份代码，这份代码相当于一个全才，无论什么领域的问题都能解决；而编译时多态则有多份代码，其中的每一份代码相当于一个专才，只能解决自己所在的领域的问题。

从前面一节的例子可以看出，泛型的实例化非常地简单，无论对于哪种泛型在实例化时只需提供合适的类型名称即可，所以花大量的笔墨来列举各泛型的实例化除了浪费纸张外没有任何其它意义。鉴于此，本节我们主要介绍的是泛型类在实例化时的一些特殊事项而并非如何实例化一个泛型。

泛型类的实例化共有四条较为特殊的规则，我们逐条介绍。

规则一，由于实例化发生于编译期间，运行时相当调用了不同的类，所以若在泛型类中定义一个类变量，不同版本的类变量会保存各自的值。来看一下下面的例子：

```
type
  TFoo<T> = class
  class var
    FCount: Integer;
    constructor Create;
  end;
constructor TFoo<T>.Create;
begin
  inherited Create;
  Inc(FCount);
end;

var
  FI: TFoo<Integer>;
  FS: TFoo<String>;
```

```
begin
  FI :=  TFoo<Integer>.Create;  FI.Free;
  FI :=  TFoo<Integer>.Create;  FI.Free;
  FS :=  TFoo<String>.Create;  FS.Free;
  WriteLn(TFoo<Integer>.FCount); // 显示 2
  WriteLn(TFoo<String>.FCount); // 显示 1
  readln;
end.
```

例中定义了一个泛型类，其中声明了一个字段 FCount，其值为类的 create 函数被调用的次数。我们重载的类的 create 函数，使得此函数每被调用一次，FCount 的值就增加 1。

在主程序中我们声明两个变量 FI 与 FS，它们分别是 Integer 及 String 版本的 TFoo 类型。这两个版本的 TFoo 相当两个同名的类。运行程序时，系统会连续两次创建并销毁 FI，创建并销毁 FS 一次。所以 Integer 及 String 版本的 TFoo 类中的类变量 FCount 的值分别为 2，1。

规则二，若一个泛型类中套嵌了另一个类，在访问这个类时需要实例化其母类：

```
type
  T1<T> = class
  type
    NestClass = class
      i:T;
    end;
  end;
var
  obj:T1<String>.NestClass;
```

规则三，在实例化泛型类的子泛型类时，其基类自动类型化：

```
type
  T1<Tk> = class
    i:Tk;
  end;
  T2<T> = class(T1<T>)
  end;
var
  obj:T2<String>;
begin
  obj.i :=  'Delphi';
end.
```

当我们使用了 String 实例化 T2 后，编译器自动产生了 String 版本的 T1。

再次说明一下：类型参数的标识符不作为类识别的依据。所以在声明 T2 时，我们将 T1 中的 Tk 变成了 T，这完全没有任何关系，只要基类与派生类的类型参数标识符相同即

可。读者可将 T2<T>中的 T 变成其它标识符后再编译，看一下能否编译。

规则四，类中的方法在未得到明确的类型值时，可根据实际的参数值进行推测，从而实例化整个类：

```
type
  T1 = class
    procedure test;
    procedure M<Y>(S:Y);
  end;

procedure T1.M<Y>(S: Y);
begin
end;

procedure T1.test;
begin
  self.M<String>('wagnin');//调用 string 版本 T1 的 M
  self.M('wagnin'); //调用 string 版本 T1 的 M
  self.M(23); //调用 Integer 版本 T1 的 M
end;
```

10.3 泛型方法重载

带有类型参数的方法可以像普通方法一样被重载，方法完全相同：在其后添加关键词 overload。

再次声明：使用 record 而非 class 仅仅是为了方便。

```
type
   T1<T> = record
     procedure M1<T>(A: T); overload;//版本 1
     procedure M1(A: String); overload;//版本 2
     procedure Test;
   end;

 procedure T1<T>.M1(A: String);
 begin
   writeln('T1<T>.M1(A: String)');
 end;

 procedure T1<T>.M1<T>(A: T);
 begin
   writeln('T1<T>.M1<T>(A: T)');
 end;
```

```
var
  obj:T1<string>;
begin
  obj.M1('delphi');
  readln;
end.
```

在这个例子中，我们将 obj 声明成 T1 的 string 版本，所以 obj.M1 调用时会将版本 1 的 M1 实例化为 string 版本：

```
procedure T1<String>.M1(A: String);
```

如何？看起来很眼熟是不是？没错，它和版本 2 的 M1 一模一样，如此一来，我们使用一个字符串作为参数调用 M1 时，编译器会选择哪个版本？

答案是版本 2。在重载方法的泛型版本与非泛型版本间，编译器优先选择非泛型版本。所以上例中编译器会优先选择版本 2。

这其中的机制很简单:若优先选择了泛型版本，编译器就需要自己编写代码，如同人一样，编译器也并不是那么勤奋，所以它会自动选择费力较少的方法。

10.4 泛型类型兼容

大学时代的好多人都喜欢考证书，笔者也未能免俗。但有一样证书笔者却从来都不考虑，这就是普通话等级证书。笔者一直觉得，普通话这种东西一张口就能辨高下，完全没必要看证书。但是不知何种原因，笔者当年的那些同学对此却热衷的很。

之所以提到上面的陈年之事并不是因为笔者看到了伟大的领袖导致大脑发热，而是因为这个这种怪现象在 Delphi 中同样存在。到目前为止，Delphi 中判断两个类型是否兼容都是按照类型名称来判断，无论这两个名称所代表的类型的实际结构是否一致。可能读者不明白这句话的意思，不要紧，看到下面的例子，相信你一定会有一种熟悉感：

```
type
  FourArray = array [0..3] of integer;
var
  v1:FourArray;
  v2:array [0..3] of integer;
begin
  v1 :=  v2;    //不能编译
end.
```

v1 与 v2 的类型除了名称之外，一切皆相同。但不幸的很，就像某些单位只看证书不看实际能力一样（从这个角度看，Delphi 还真是无比的适合中国国情），Delphi 根本不认实际结构，它只根据名称来判断类型是否一致。所 v1 与 v2 不能相互赋值。

随着泛型的出现，这个规则出现了一点小小的不和谐。可能因为泛型在国内用得比较

少而未能为天朝所同化，相对于类型名称，泛型更注重数据类型的实际结构并以此为据来判断两种类型是否兼容。为验证此点，我们使用泛型重写了上面的例子，如下：

```
type
  GArray<T> = array [0..3] of T;
  FourArray = GArray<Integer>;
var
  v1:GArray<Integer>;
  v2:FourArray;
begin
  v1 :=  v2;
end.
```

编译通过，没有任何问题。

可能是受泛型的影响，在实例化时编译器同样会检查类型参数所指向的最终类型是否相同。例如将这个例子改为如下形式依然可编译通过：

```
type
  GArray<T> = array [0..3] of T;
  Int = Integer; //新加入的一行代码
  FourArray = GArray<Int>; //FourArray 的类型参数值变成了 Int
var
  v1:GArray<Integer>;
  v2:FourArray;  //相当于 GArray<Int>
begin
  v1 :=  v2;
end.
```

10.5 泛型的限定

到目前为止，我们使用的泛型都是由编译器控制，我们唯一能做的仅仅只是声明一个泛型并在代码中使用它。关于它的实例化我们没有任何的方式可以干预，例如我们定义一个泛型类，可能我们希望它在实例化时只接受几种特定的数据类型作为参数，但迄今为止我们无法做到这点。所幸，Delphi 并没有忽视这点，本节我们学习的内容就能部分的解决这个问题。

可以读者会奇怪：为什么不能完全的解决这个问题？我的理解是：完全解决这个问题意味着在对泛型实例化时，我们可以自由的定义哪些类型用于实例化而哪些类型不能用于实例化，这样一来，我们使用泛型的理由又在哪里呢？当然实际理由可能并非如此，但这并非是我们该关注的东西，说不定 Delphi 的下个版本能够彻底的解决这个问题也未可知。

闲话不多说，本节我们开始学习泛型的限定，所谓泛型的限定是指通过某种方法可以限制泛型在实例化时能够接受的类型参数，例如通过添加一些关键词我们可以指定泛型实例化所用的类型必须是一个类类型或一个值类型。Delphi 提供了 5 种这样的方法来限制实例化时所用的数据类型。

1. 接口限定

在定义泛型类型时可在类型参数后加上冒号，后接一个或多个接口名称。如此一来，实例化时所用的类型必须是实现了这个接口，这也限定了这个类型必须是一个类类型。

注意，这个类实现了多少其它的接口无关紧要，关键是这个类，或者这个类的祖先类中一定要直接实现我们指定的所有接口的本身。这句话的意思是:若我们要求实现接口 I1，现在定义了一个类 T1，只有 T1 或其祖先类中直接实现了 I1 本身（不能是 I1 的子代接口），T1 才能被用于实例化：

```
type
  I1 = Interface
  End;
  I2 = Interface(I1)
  End;
  T1 = class(TInterfacedObject,I2)
  end;
  TCon<T:I1> = class //T 必须直接实现 I1
  end;
var
  obj:TCon<T1>;  //T1 中未直接实现 I1
begin
end.
```

上例无法编译通过。原因是 TCon 实例化时所用的类型必须支持 I1，而 T1 中并未直接实现 I1。

```
type
  I1 = Interface(IInterface)
  end;
  T1 = class(TInterfacedObject,I1)
  end;
  TCon<T:IInterface> = class
  end;
var
  obj:TCon<I1>;
begin
end.
```

这个例子中，虽然 T1 未直接实现 IInterface，但在 T1 的祖父类 TInterfacedObject 中直接实现了 IInterface 接口，所以 T1 可用于 TCon 的实例化。读者可将 T1 定义中的 I1 更换成其它的任何类或直接将它去掉，然后编译，会发现这个例子依然可运行。因为 T1 的祖先类中已经实现了 IInterface 接口，这样一来，T1 无论继承于哪个接口，甚至不继承任何接口也能用实例化。

2. 类名称限定

我们也可以使用一个类的名称来进行限定，表示所有用于实例化的类型与这个类之间必须赋值兼容，简单来讲，用于实例化的类型必须是这个类本身、nil 或这个类的派生类：

```
type
  T1 = class
  end;
  T2 = class(T1)
  end;
  TCon<T:T1> = class
  end;
var
  obj:TCon<T2>;
begin
end.
```

在这段代码中我们使用 T1 限制了 TCon 的类型参数，在实例化 TCon 时 T 的值只能是一个类或 nil，若是一个类则必须是 T1 或 T1 的派生类。

有一点很奇怪：Delphi 中好像是不允许直接使用 TObject 来进行限定，这让人很费解。不过这不影响，在需要用到 TObject 进行限定的场合，我们可以直接使用后面介绍的 class 进行限定，它等效于直接使用 TObject 进行限定。

3. constructor 关键词限定

constructor 限定一般较少使用。使用这种限定可以直接调用 T 的默认构造函数（及无参数的 create 函数）而无需知道 T 到底代表了哪种类型，当然了，T 所代表的类型必须是一个类，因为只有类才有默认构造函数。

```
type
  TCon<T:constructor> = class
    procedure test;
  end;

procedure TCon<T>.test;
var
  obj:T;
begin
  obj :=  T.create;  //此时并不知道 T 的具体类型
end;
```

官方文档中的描述是 T 必须有一个默认构造函数。事实上完全没必要，就算 T 中没有包含一个默认构造函数，编译器在编译时也会直接调用 TObject 的 create 函数。

4. class 关键词限定

与前面的类名称限定不同，此处的 class 限定是指直接使用关键词 class 来进行限定。这种类型的限定非常简单，它只要求用于实例化的类型是一个类类型即可，和这个类有什么成员、继承于哪个类等信息完全没有关系。此关键词可以与 constructor 进行联用，但这种联用似乎没有多大的意义。

```
type
  TCon<T:class> = class
  end;
var
  obj:TCon<TObject>;
```

5. record 关键词限定

record 限定了用于实例化的类型必须是值类型，即所有的简单类型加上记录类型。

```
type
  T1 = class
  end;
  TCon<T:record> = class
  end;
var
  o1:TCon<T1>;    //错误，T1 不是值类型
  o2:TCon<Integer>;
```

以上介绍的五种限定可以联合使用，但是注意：

- class 或 record 关键词不能与类名称联合限定。
- record 关键词也不能与 class 或 constructor 是的任何一个联用。

10.6 TList 类

Delphi 中预定义了一些泛型类，其中 TList 的历史可以说是最长。笔者并不知道 Delphi 从何时开始引入了这个类，从笔者第一次接触 Delphi 时，这就已经存在，所以有必要好好的介绍一下 TList 的具体用法。

基本上，从使用角度而言，TList 类似于一个智能化的动态数组，使用其构造函数创建一个 TList 对象后，它在内存中并不占用多少空间，在不断加入新的成员时 TList 对象所占用的空间不断的增大。看起来 TList 有点像是气球，在没有吹气时气球本身不会占用多少空间，吹进的气体越多，气球所占的空间就越大。不过很显然，吹大 TList 比吹气球要方便得多。

TList 的成员众多，但我们的目的是应用而非研究，所以下面只列举一些常用的方法。如有需要，读者可自行开启 Generics.Collections 单元查看这个泛型类的详细定义。

```
TList<T> = class(TEnumerable<T>)
  ...
  constructor Create; overload;
  destructor Destroy; override;

  function Add(const Value: T): Integer;
  procedure AddRange(const Values: array of T); overload;
  procedure Insert(Index: Integer; const Value: T);
  procedure InsertRange(Index: Integer; const Values: array of T); overload;

  function Remove(const Value: T): Integer;
  procedure Delete(Index: Integer);
  procedure DeleteRange(AIndex, ACount: Integer);
  function Extract(const Value: T): T;
  procedure Clear;

  procedure Exchange(Index1, Index2: Integer);
  procedure Move(CurIndex, NewIndex: Integer);
  procedure Sort; overload;

  function Contains(const Value: T): Boolean;
  function IndexOf(const Value: T): Integer;
  function LastIndexOf(const Value: T): Integer;

  property Capacity: Integer read GetCapacity write SetCapacity;
  property Count: Integer read FCount write SetCount;
  property Items[Index: Integer]: T read GetItem write SetItem; default;
  ...
end;
```

不要头晕，看起来可能确实是有点麻烦，但笔者相信这份代价是值得付出的。

首先，create 与 destroy 应该不用介绍。注意 TList 在刚创建后是一个空对象，不含有任何一个成员。不含成员的意思是没有成员而不是成员的值为 0，正如空房间指的是房间里没有东西而有一大堆的空椅子。所以说 TList 更类似于动态数组而不是静态数组（静态数组在声明之后其成员就已存在，只是其值不确定而已）。

创建一个 TList 对象后，我们总要往其中添加对象吧？从名字也看出来了，TList 在内存中表现为一个连续的队列。和现实世界不同，Delphi 是允许插队。所以添加成员就有了两种方法：到队尾排队、插队。将 Add 可将成员添加到队尾，Insert 则将成员插入特定的位置，如 List.Insert(3,78)表示将 78 插入 list 中作为第 4 个成员（序号为 3 的成员）。

有两点需要引起特别的注意：

- 任何情形下，TList 中成员序数绝对是从 0 开始。用户无法改变这点。
- 在添加成员时请务必保证这个成员在原来的队列中存在。如上面的 Insert(3,78)

中必须保证 list 中存在序数为 3 的成员，否则会引起运行期错误。

当需要添加大量成员数，逐个逐个手动添加是一件很麻烦的事，此时可以利用 AddRange 与 InsertRange 将数组添加到队列中。AddRange 将一个数组插入到队尾，而 InsertRange 则将数组插入至指定位置。例如：

```
uses
  SysUtils, Generics.Collections;//记得添加此单元
var
  ListArray:array[0..3]of integer;
  i:integer;
  list:TList<Integer>;
begin
  list :=  TList<Integer>.Create;
  for i :=  0 to 3 do
    ListArray[i] :=  i+101;   //成员值为 101，102，103，104
  for i :=  0 to 4 do  //在列表中添加 5 个成员，值为 1，2，3，4，5
    list.Add(i+1);

  list.InsertRange(3,ListArray);//请确保 list 中存在序数为 3 的成员
  for i :=  0 to list.Count - 1 do
    writeln(list[i]);  //输出 1，2，3，101，102，103，104，4，5
end.
```

若读者将例中的 list.InsertRange(3,ListArray)改为 list.AddRange(ListArray)，输出值为：

1，2，3，4，5，101，102，103，104

做为一个队列，不能只增不减，所以我们还需要从 list 中删除成员。Delphi 提供五种删除成员的方法：Remove、Delete、DeleteRange、Extract，Clear。

Remove 最为简单，它删除队列中值为 value 的成员，并返回这个成员的序数。若队列存在多个值相同的成员，remove 会删除最小者。如从 2，7，3，7，5 中用 remove 删除 7，则队列变为 2，3，7，5，由于删除的成员序数是 1，所以 remove 返回 1。若队列中不存在指定的值，remove 返回-1。

Extract 与 remove 非常相似，唯一的区别在于 Extract 返回的不是序数而是被删除的成员的值。若队列中不存在指定的值，Extract 返回 T 类型的默认值，如 0、nil 等。

Delete 直接返回指定序数的成员，DeleteRange 则删除从序数 AIndex 开始的 ACount 成员。如 list 的成员值为 1，2，3，4，5，那么运行 list.DeleteRange(1,2)后，list 值为 1，4，5。

Clear 的作用从名称上可以看出，它清空队列中所有的成员，将队列变为空队列。

除了删除及添加队列成员，Delphi 还提供了其它的方法来操纵成员：Exchange、Move、Sort。

Exchange 用于将 Index1 与 Index2 指定的成员的值互换。

Move 将 CurIndex 指定的值移动到 NewIndex 指定的位置，同样应注意在队列中应当存在这两个序数所指定的成员。其移动过程是：取出 CurIndex 位置的成员取出，然后将序数大于 CurIndex 且小于或等于 NewIndex 的所有成员往后退一格（即序数全部减一），空出序数为 NewIndex 的成员位置，然后将原来队列中序数为 CurIndex 的成员移动到空出的位置。

1. 例如对于队列 1，2，3，4，5 执行 Move(1,3)的具体操作为：
2. 将队列中序数为 1 的成员取出，则队列变为 1,_,3,4,5。("_"表示空位)
3. 将所有序数大于 1 且小于或等于 3 的成员后移一格：1，3，4，_,5
4. 最后将取出的成员置于其中的空位：1，3，4，2，5

sort 用于将队列中所有成员的值顺序排序。默认为由小到大排序。

```
uses
  SysUtils, Generics.Collections; //记得添加此单元
var
  ListArray:array[0..3]of integer;
  i:integer;
  list:TList<Integer>;
begin
  list :=  TList<Integer>.Create;
  for i :=  0 to 4 do  //在列表中添加 5 个成员
    list.Add(i+1);   //list 为 1，2，3，4，5

  list.Exchange(0,4);
  for I :=  0 to list.Count - 1 do
  begin
    write(list[i]);
    write(' ');
  end;
  writeln;
  //输出 5，2，3，4，1
  list.Sort;
  for I :=  0 to list.Count - 1 do
  begin
    write(list[i]);
    write(' ');
  end;
  writeln;
  //输出 1，2，3，4，5
  list.Move(1,3);
  for I :=  0 to list.Count - 1 do
  begin
    write(list[i]);
    write(' ');
  end;
  writeln;
```

```
  //输出 1，3，4，2，5
  readln;
end.
```

我们可查询队列中是否含有某个值，Delphi 提供三种方法：Contains，IndexOf，LastIndexOf。
Contains 用于查询队列中是否存在某个成员的值与指定值相同，当存在某个成员的值与指定值相同时，可使用 IndexOf 返回此成员的序数。若存在多个这样的成员，IndexOf 返回第一个成员的序数，而 LastIndexOf 则返回最后一个这样的成员的序数。当所查之值不存在于队列中时，IndexOf 与 LastIndexOf 均返回-1。

除以上介绍的方法外，TList 还定义了个有用的属性：Count、Item、Capacity。Count 表示当前 TList 对象中的成员数目。Capacity 表示当前对象的容量，也即能够储存的最大成员数量，设置这个属性值有点像设置动态数组的成员数量。

最后介绍 TList 的默认属性 Item。在引用 TList 对象的成员时，我们可使用此属性进行索引如 list.Item[2]，但 Item 是 TList 的默认属性，所以直接使用对象名加索引的形式如 list[2]。如果读者不明白，可以复习一下默认属性的相关内容。

附录 A ASCII 字符集

数值	字符	数值	字符	数值	字符	数值	字符
0	NULL	32	空格	64	@	96	`
1		33	!	65	A	97	a
2		34	″	66	B	98	b
3		35	#	67	C	99	c
4		36	$	68	D	100	d
5		37	%	69	E	101	e
6		38	&	70	F	102	f
7		39	’	71	G	103	g
8	退格	40	(	72	H	104	h
9	制表	41	)	73	I	105	i
10	换行	42	*	74	J	106	j
11		43	+	75	K	107	k
12		44	,	76	L	108	l
13	回车	45	-	77	M	109	m
14		46	.	78	N	110	n
15		47	/	79	O	111	o
16		48	0	80	P	112	p
17		49	1	81	Q	113	q
18		50	2	82	R	114	r
19		51	3	83	S	115	s
20		52	4	84	T	116	t
21		53	5	85	U	117	u
22		54	6	86	V	118	v
23		55	7	87	W	119	w
24		56	8	88	X	120	x
25		57	9	89	Y	121	y
26		58	:	90	Z	122	z
27		59	;	91	[	123	{
28		60	<	92	\	124	|
29		61	=	93	]	125	}
30		62	>	94	^	126	~
31		63	?	95	_	127	

注：空置的表格表示对应的这些字符都没有固定的图形表示，对于不同的应用程序，这些字 符可能会影响文本的显示效果。

附录 B 变体类型转换

本部分主要介绍了将变体变量值在不同类型间转换的规则。虽然是不同类型的变体变量值间的转换，但终究还是变体类型，只是将其中容纳的值由一种类型变成了另一种类型。

为了保证阅读的便利性，下面的表格中仅仅填写了一些数字，每一个数字代表一种类型 的转换。如 12 代表整数转换成实数。表格中的纵列表头表示原来的类型(Source，简称 S)，横列表头表示转换后的类型(Target，简称 T)。

S T	整型	实型	字符串	逻辑
整型		12	13	14
实型	21		23	24
字符串	31	32		34
字符	41	42	43	44
逻辑	51	52	53	
Unassigned	61	62	63	64
Null	71	72	73	74

12——将变体变量中的整数值转化为实型值：直接转化。如 1 会转成 1.000000。
13——变体变量中的整数值转化为字符串值：转化成以字符串形式描述的数字，如 34 被转化成'34'。
14——变体变量中的整数值转化为逻辑值：整数值为 0 时转化为 False，否则转化成 True。

21——变体变量中的实数值转化为整数值：转化成与原值接近的整数值，如 2.33 转化成 2，而 2.77 则转化成 3。

23——变体变量中的实数值转化为字符串：转化成字符串形式的实数值，如 2.33 转化成 '2.33'。

24——变体变量中的实数值转化为逻辑值：实数值为 0 时转化为 False，否则转化成 True。

31——变体变量中的字符串转化为整数值：只有字符串形式的数字（如'12'）才能被转化。若转化后的值超出整数范围或并非整数形式的字符串将引起异常。

32——变体变量中的字符串转化为实数：只有字符串形式的实数（如'0.9'）才能被转化。超出实数范围或并非实数形式的字符串将引起异常。

34——变体变量中的字符串转化成逻辑值：当字符串的值是'False'（不区分大小写，'false'、'FALSE'等值均可）或是由一串数值等于 0 的数字组成的字符串如'0.0'时，返回

False。当字符串的值是'True'（不区分大小写，'true'、'TRUE'等值均可）或是由一串数值不等于 0 的数字组成的字符串如'0.1'、'2'时，返回 True。

41~44. 由于 Delphi 中不区分字符及字符串，故而转换时将字符当成字符串。

51——将变体变量的逻辑值转化为整数值：若逻辑值为'False'则转化成 0；若为'True'时情形稍麻烦，假设得需要将逻辑值转化成整数中的 T 类型，V 为 T 类型的变量，则将 V 的所有字节的所有位(bit)都置为 1 时所得到的值就是我们需要的转化值，如若转换成 Integer 则得到-1，转换成 Byte 则得到 255。

52——将变体变量的逻辑值转化为实数值：若逻辑值为'False'则转化成 0；若为'True'则得到 1。

53——将变体变量的逻辑值转化成字符串：若为 False 则得到'False'；若为 True 则由定义于 Variants 单元中的 BooleanToStringRule 全局变量决定：当其值为 bsrAsls 时得到'True'，其值为 bsrUpper 时得到'TRUE'，其值为 bsrLower 时则得到'true'。默认为 bsAsls。

61——将变体变量中的 Unassigned 值转化成整数值：得到 0。
62——将变体变量中的 Unassigned 值转化成实数值：得到 0。
63——将变体变量中的 Unassigned 值转化成字符串：得到空字符串。
64——将变体变量中的 Unassigned 值转化成逻辑值：得到 False。

71~74——这四种情形的转换均取决于 Variants 单元中的逻辑类型的 NullStrictConvert 全局变量，当此值变量为为 True（默认情况下为 True）时，将变体变量的 Null 值转换成任何类型都会引了一个 EVariantTypeCastError 异常。当此变量值为 False 时，将变体变量的 Null 值转换成整数值或实数值时会得到 0，转换成逻辑值时得到 False，转换成字符串时将得到一个空串。

附录 C 常见字符集与字符编码方式

字符集是指囊括特定字符的一个集合，其中每个字符均有一个位置，如 A 在 ASCII 中排在 65 位，B 排在 66 位。理所当然，字符集也规定了每个字符在计算机中的书写方式，如‘＝’在计算机中的由两个相等的短线而不是由其它图形组成。

本节我们将介绍计算机中的字符集与字符编码。篇幅所限，我们只介绍目前国际上最为通用的三种字符集：ASCII 字符集，UCS 字符集，Unicode 字符集。在介绍这三种字符集后，我们会继续介绍一些常见的字符编码方式，以期读者在学习字符串相关内容时不至于漫步云中。

OK，我们开始。首先是字符集的鼻祖——ASCII 字符集。

C.1 常见字符集

1. ASCII 字符集

ASCII 的全称是美国信息交换标准码(American Standard Code for Information Interchange)，自 20 世纪 50 年代后期由美国国家标准局(ANSI)开始制定，于 1967 年完成，又被称为 ANSI 码。

此字符集中包含了 128 个字符，包括英文字母、阿拉伯数字以及诸如空格等其它字符，虽然数目较少，但足够西文如英文与法文的需要。ASCII 字符集中的每个字符在计算机中均只占一个字节，故而理论上一个字节最多可表示 2^8＝256 个值，但由于字节的最高位用于进行检查错误，所以 ASCII 中只容纳了 128 个字符。

后来随着计算机的发展，计算机的使用已不再局限于英语国家。此时，ASCII 字符集便显得不足，其无法在计算机上显示任何其它如汉字、日语文字等非英文的字符，而我们又不能指望所有的国家都改说英语。所以各国便根据自身的状况将标准 ASCII 字符集扩展成能容纳本国文字的字符集，其中最为常用的办法就是用多个特定的 ASCII 字符联合起来表示一个非英文字符。

以中文为例，我们使用 2 个值均大于 127 的字节来表示一个汉字：当计算机发现某个字节的值大于 127 时，它便会知道这个字节与其后的一个字节联合表示一个汉字。由于标准 ASCII 字符集未规定大于 127 的字符，故而这种方式编码的汉字在解析时不会和标准的任何 ASCII 字符集混淆。这种字符所形成的字符集称之为 GB2312。理论上 GB2312 中可容纳多达 32768 个字符。

但中国文字的数量实在过于庞大，这么大的数字只是所有中文字符的一小部分。所以后来规定，组成中文字符所需的两个字节中，不必两个字节值全部大于 127，只需要第一个大于 127 即可。这种方式形成的字符集就被称为 GBK 字符集。到按这种编码方式，当计算机遇到一个大于 127 的字节时，便知道此字节与其后的一个字节联合表示一个中文字符。后来为了照顾少数民族，又在 GBK 字符集中加入了大量少数民族所使用的的字符，形成了 GB18030 字符集。

顺便一提，由于中文标点与英文有一定的差异，所以中文字符集除了标准 ASCII 中的所有标点外，还特地为每个中文的标点也做了编排。所以我们在编辑文档时会有中文标点与英文标点之分。

2. UCS 字符集

UCS 的全称是通用字符集((Universal Character Set)，由 ISO 制定的 ISO-10646(或称 ISO/IEC-10646)标准所定义的字符编码方式。

我们前面讨论了中文对于 ASCII 的扩展，这种扩展很好的解决了各种语言中的字符在计算机上的使用问题，若非这种扩展，我们无法在电脑上看到汉字，更不用说输入汉字。但由于各国的扩展字符集不一致，导致同一个编码在不同语言中代表不同的文字。例如，中文两个字节 206、210 的组合在中文中表示汉字‘我’，但在其它文字诸如日语或是朝鲜文中却并不表示这个字。这样所造成的不便非常之多，其中之一便是计算机软件无法在不同语言的版本的操作系统中使用。抛开程序不说，如果使用这种编码方式，我们恐怕连国外的网页都无法浏览。很明显，这种现象非常不利于计算机与互联网的国际化。

为弥补 ASCII 字符集的的诸多不足，国际标准化组织(ISO)开始制定一个意在囊括全世界所有字符的字符集：ISO-10646。这是很伟大的想法，因为它不分国籍、不分民族的将全世界所有的字符全部收录进同一个字符集中，这样一来，我们在替字符编码时永远不会遇到同一编码对应多个字符的情形，乱码也将成为永远的传说。所以说，这确实是一个世界极的伟大计划。经过多年努力，这个计划终于产生了第一期的成果：UCS-2 与 UCS-4 两种版本的字符集。

3. Unicode 字符集

先驱者的眼光总是相同的。

就在 ISO 紧锣密鼓的制定 UCS 时，包括苹果、IBM、微软等数家计算机行业的巨头也于 1990 年联合成立了 Unicode.org 组织，意在创造出一套与 UCS 类似的囊括所有字符的字符集，这个组织在 1994 年发布了 Unicode1.0 字符集。不过很快地，这个组织和 ISO 均意识到：这个星球上并不需要两套类似的字符集。于是自 Unicode2.0 开始，Unicode.org 抛弃了 Unicode1.0 中的字库而采用了 ISO-10646-1 的字库和字码。也就是说，自 Unicode1.0 后的所有 Unicode 版本与 UCS 并无实质上的区别，同一个编码在 Unicode 字符集与 USC 字符集中一定对应同一个字符。所以在大部分场合不再区分 UCS 与 Unicode。在不是那么精确的场合，读者完全可以将两者当成一回事。如果实在要说区别，二者只有一点点：

UCS 字符集只是将字符收集起来并给每一个字符安排一个唯一的序号，但其确并未规定在计算机中如何存储或书写这个字符。Unicode 则包括所有实现细节。

打个比方，Unicode 相当于一家汽车生产商，这家厂商在过去生产汽车时所有的环节都由自己一手包办，连原料都由自己亲自加工。但后来因为某个原因，它决定从此以后不再自己生产原料，而改用另一家名为 USC 的厂家所提供的原料。这个例子中的原料就相当于字库，USC 只是提供了原料而已。

根据每个字符占用的空间大小的不同，UCS 字符集分为 UCS-2 与 UCS-4 两种。两者的区别：USC-2 对于其中的每个字符采用 2 个字节进行编码，而 USC-4 则采用 4 个字节。Unicode 正是采用了其中的 USC-2 字符集。也因为如此，现在一般将 UCS-2 等同于 UTF-16。这样并没任何不妥，毕竟两用的是同一字符集。

很明显，USC-4 能表示更多的字节。不过即使，USC-4 实际使用的很少。理由很简单：目前 UCS-2 已经满足基本上所有的需要，所以不需要使用 USC-4。这就像是天天有鱼吃的猫绝对不会去抓老鼠一样。

C.2 字符编码方式

在介绍完三种字符集之后，按照约定，我们来看一下以 Unicode 为基准的几种编码方式：UTF-16、UTF-8、UTF-32。可能读者有疑问：为什么只介绍 Unicode？ASCII 呢？USC 呢？

我的回答是：USC 没提供实现，所以不存在编码方式。ASCII 虽然有编码方式，但前面已经充分的介绍过，所以在此就不再重复。

在此之前我们先要明白我们为何需要字符集编码。字符编码主要有两个作用：

第一当然是给每个字符安排一个序号；

第二就是规定如何在计算机中储存这个序号，以使得计算机能够完整的不带歧义的识

别所有的序号。举个例子。假设某个汉字用 ASCII 码表示时其值为 065 066，那么计算机怎么知道这两个字节联合起来表示一个汉字？计算机完全有可能将它们当成是两个字节分别表示为 AB。为了防止出现这样的问题，设计之初就规定：当两个字节联合起来表示一个汉字时，第一个字节的值必须大于 127。这样一来，只要计算机发现某个字节值大于 127，它就会明白这个字节和后面的字节合起来表示一个汉字。

通过这个例子，读者是否对字符编码有一个大概的认识呢？如果你的答案是肯定的，那我们就正式开始下面的内容。

1. UTF-16

UTF-16 设计之初意图使用 2 个字节来表示字符，理论上用这种方式最多可表示 65536 个字符，但这么多字符还是不太够，所以后来又将 UTF-16 进行一些扩展使得其可以表示更多字符。

UTF-16 在给字符进行编码时以$ FFFF 将整个字符集分成两部分。自 0 至$FFFF 共计 65536 个码位可满足大部分需要，故将此区域称之为基本多文种平面(BMP)。下面我们就来具体看一下这种编码方式的具体算法。为方便起见我们用 U 表示 UTF-16 中每个字符的编码值。

对于 U ≤ $FFFE 的部分，不做任何变化，U 代表的值就是字符的编码。

对于 U ≥ $FFFF 的部分则较为复杂：

先计算 U-$10000 的值，然后将此值写成二进制形式：yyyy yyyy yyxx xxxx xxxx，若小于 20 位，可在高位补 0 凑成 20 位，如此，U 的 UTF-16 编码（二进制）就是：110110 + yyyyyyyyyy 110111 + xxxxxxxxxx。 例如计算编码为$10BCD 的字符在 UTF-16 中的编码可进行如计算：

首先，将其减去$10000 的值为$BCD，写成二进制并凑成 20 位：
00000000101111001101，则此值的 UTF-16 编码为：110110+0000000010
1101111111001101，写成十六进制为 D802 DFCD。

从上可以看出对于 Unicode 字符集中序号大于$FFFE 的字符，UTF-16 使用了四个字节来表示。读者现在应当明白不同实现方式中的编码与 Unicode 中编码的区别。不过所幸的是，一般使用中用到$FFFE 之后的字符的情形非常之少。

上面的计算还有一个问题，UTF-16 既有两字节的编码，也有四字节的编码。如$AB 表示一个字符，$ABCD 也表示一个字符，当计算机遇到$AB 时，它如何分辨这两个字节是单独表示表示一个字符还是作为$ABCD 的前两个字节与后面的$CD 联合来表示另一个字符？

UTF-16 使用了一个巧妙的方法：保留$D800 及$DFFF 之间的代码点用于标记。换句话说，在 UTF-16 的 2 字节字符中，并不存在位于编码位于$D800 及$DFFF 的字符。这个区间的代码点用于标记 4 字节编码。此区间称为标记区。

当计算机遭到一个编码值位于标记区 2 字节编码时，由于此编码区间并没有表示任何字符，故而计算机将其理解为一个 4 字节编码的标记。这个区间分为三个部分：

$D800－$DB7F $DB80－$DBFF $DC00－$DFFF

其中$D800-$DB7F 之间的编码为高位标识，当计算机遇到此区间的编码时，将所在的字节理解成是一个 4 字节编码中高位的 2 字节，计算机将这两个字节后的 2 字节作为低位的 2 个字节与其联合成 4 个字节来表示一个字符。

$DB80-$DBFF 之间的编码亦为高位标识，其处理与$D800-$DB7F 区间中的编码完全一样，只不过此区间能表示的字符在 Unicode 全部作为专用字符，故此区间又称为高位专用标记。

$DC00-$DFFF 之间的编码为低位标记，计算机遇到此区间的编码时，将其作为低位的 2 字节并与其前面的 2 个字节联合成 4 字节表示一个字符。

但计算机读取的字节中不含标记区的编码时会自动将当前字节与其后的一个字节联合表示一个字符。

有些读者可能疑惑：4 字节编码中加入了标记，如何能保证序号大于$FFFF 的字符在 UTF-16 中的编码一定是 110110 + yyyyyyyyyy 110111 + xxxxxxxxxx 形式？其中的 110110 和 110111 难道不会变吗？

答案是不会。至于为什么，笔者的算术不太好没法证明，若读者有兴趣可以自己去证明。

以这种方式编码，UTF-16 总共可编码的字符数为$7E0FFFF，转换成十进制：132186111，超过了一个亿。这个数量对于目前地球上的字符而言，显得相当的过剩，故而 UTF-16 中还保留余有大量空闲的码位以备将来扩展。

看到这里，相信读者的大脑一定有点晕，不过很遗憾，您还是要接着看。我们还有一个大问题。

还是以$AB 为例，我们看一下都知道 A 为高字节，B 为低字节，所这个字节代表 0010 0011。但不同计算机的 CPU 往往不一致，我们不能假设不同的 CPU 都是按我们的想法来理解这两个字节的顺序，例如有些 CPU 就会认为这两个字节代表 0011 0010。这个问题如何解决？总不能告诉 CPU 应该从前往后读或从后往前读吧？

设计者当初亦考虑了这个问题。他们采用了 BOM（Byte Order Mark）技术来解决这个问题。说起 BOM，笔者第一次见到这个名字时感觉好复杂，后来发现自己思维过度。所谓的 BOM 其实就是在文本开头做一个标记来告诉 CPU 高位的字节是放在前面还是放在后面，如对于$AB，存储于文件中时其存储方式究竟是 AB 还是 BA，若是 AB，此编码技术称之为 Litte Endian(LE)，表示高位的字节放在低位字节前面，若为 BA，则称之为 Big Endian(BE)表示高位的字节放在低位字节后面。简单来说，对于阿拉伯数字 98，若采用 LE 则在文件中的存储顺序为 98 高位字节在前，而 BE 则是 89，高位字节在后。系统会在 LE 编码的文件的开头放置一个标记：FFFE，在 BE 编码的文件的开头放置 FEFF。CPU 在读取字节时，若发现此文件开头的标记是 FFFE，会自动按读取顺处理字节；若为 FEFF，会将逆序处理字节。

最后提一下 Unicode，由于历史原因，此词身兼多义。首先，表示 Unicode 字符集。其次，由于 Unicode2.0 开始采用 UCS 的字库，故而 Unicode 又可表示 UCS 字符集。最后，由于 Windows 系统自 Win98 后使用 Unicode 字符集(相当于 UCS-2 字符集)且使用其中的 UTF-16 LE 编码方式来描述文本，故在 Windows 平台上 Unicod 有时又特指 UTF-16 LE 编码方式。

2. UTF-8

UTF-8 是一种变长的编码技术，采用 1 至 4 个字节编码。不同于 UTF-16，UTF-8 并不需要对原 ASCII 中字符做任何处理，UTF-8 直接采用 1 个字节来编码这些字符。其编码方式如下：

```
0000 0000-0000 007F | $xxxxxx
0000 0080-0000 07FF | 11$xxxx 1$xxxxx
0000 0800-0000 FFFF | 111$xxx 1$xxxxx 1$xxxxx
0001 0000-0010 FFFF | 1111$xx 1$xxxxx 1$xxxxx 1$xxxxx
```

CPU 在读取 UTF-8 编码的文件时，会自动将一个字节视为一个字符，除非其读到了某

字节的前三个位为 110、1110、11110。

若某字节的高三位为 110，表示此字节与其后的一个字节联合表示一个字符。

若某字节的高四位为 1110，表示此字节与其后的二个字节联合用于表示一个字符。

若某字节的高五位为 11110，表示此字节与其后的三个字节联合用于表示一个字符。

可以看出，此种编码方式比 UTF-16 的编码方式要巧妙的多。由于对于 ASCII 字符集中的字符直接保留原来的编码，使得 UTF-8 与 ASCII 编码保持兼容，有利于编码的迁移。

不过不知道为何，在此种编码方式中汉字基本占三个字节，用此种编码方式表示汉字会造成极大的浪费，所以综合来说，中文地区一般较多使用 Unicode 编码文本。同 UTF-16 一样，系统会在 UTF-8 编码的文本开头的三个字节添加标记 EFBBBF。

3. UTF-32

UTF-32 只是将 Unicode 中的字符进行了 32 位编码。将 UTF-16 转换为 UTF-32 只需在高位添加 0，而对于其它字符集，必须采取相应的方法。

C.3 汉字字符集

以上，我们介绍国际上流行的字符集及编码方式。接下来，我们简单介绍一下汉语相关字符集：

GB2312：于 1980 年颁布的简体字符集。此字符集实际上就是上文提过的 ASCII 扩展字符集的汉语版本，其采用 2 个连续的字节表示一个字符，其中这两个字节均只使用低 7 位。此字符集共囊括了 6763 个汉字、682 图形及其它字符。

GB13000：全称为《CJK 统一汉字编码字符集》——国家标准 GB13000.1。读者可以将其理解为是 ISO-10646 的另一个名字。在其初始版本也就是 1993 版中包括了 20902 个汉字。随着 ISO-10646 标准的不断完善，GB13000 也在不断发展。不出意外的话这套字符集将是未来的趋势。

GBK：全称为“汉字国标扩展码”，于 1995 年颁布。此字符集基本上采用了 GB2312 所有的汉字及码位，并涵盖原 Unicode 中所有的汉字及 CJK 中含有的汉字，当然，包括繁体字，兼容 GB2312 与 ISO10646。其中包含了日语字母、俄文字母等。注意，此标准只是一个参考标准而并非正式标准。目前在大部分场合已经被 GB18030 所替代。

GB18030：最初于 2000 年颁布，是目前中国大陆计算机行业的强制性标准，向前兼容 GBK 及 GB2312。事实上，此字符集正是 GB2312 的扩充。在功能上，GB18030 较之 GB13000 更适用于汉语编码，而 GB13000 则比 GB18030 更适用于国际编码。与 UTF-8 类似，GB18030 的编码也采用变长字符集。其中 1 字节的编码与标准 ASCII 字符集一样，其取值区间为 $00-$7F 也就是说，若计算机读到某个字节值位于此区间时就知道此字节表示一个 ASCII 字符，。2 字节的编码中高字节的取值区间为$81-$FE，尾字节的取值区间为$40-$8E。4 字节编码从左向右算起的第 1 个字节和第 3 个字节的取值区间均为$81-$FE，第 2 个字节与第 4 个字的取值区间均为$30-$39。以计算机的角度而言，当读到第 1 个字节的值位于$81-$FE 区间时，计算机便会知道此字节表示一个 2 字节编码或 4 字节编码的开头，接下来是第 2 个字节，如果第 2 个字节的值位$40-$8E 则表示这个字节与上一个字节联合表示一个 2 字节编码，若位于$30-$39 则表示这个字节是某个 4 字节编码中的一个字节。

BIG5：即俗称的“大五码”，是为目前最常用的繁体汉字字符集内码，系台湾部分企业组成的联盟于 1984 所分布，通行于台湾、香港等地区，其中共收录 13060 个中文字。经

过多年的普及发展，此编码现已成为世界上繁体中文显示的标准格式。BIG5 使用双字节编码，高字节取值区间为$81-$FE，低字节的取值区间为$40-$7E 和$A1-$FE。

附录 D Delphi2010 字符串详解

Delphi2010 中支持的字符串类型相当丰富，除了传统的 AnsiString、ShortString 及 WideString 之外，为了方便处理 Unicode 编码，还支持 UnicodeString 及 RawByteString 类型，除此之外，自 Delphi2009 开始，Delphi 新增 UCS4String 类型，不过此类型并不常用，可能是为了扩展而设置。

接下来，我们将着重讨论这些类型。由于字符串与字符串指针类型关系密切，为方便起见，我们也讨论字符串指针。

1.ShortString

ShortString 又称为短字符串（相对的，Ansistring、widestring、unicodestring 称为长字符串），其实质上是一个编译器内置的字符数组，类似（不是等效）于以下类型：

```
Type
  ShortString = array[0..255] of ansichar;
```

其容纳由 256 个 AnsiChar 组成的字符串，但第一个字符有特殊用途，所以 shortstring 类型的字符串长度不可超过 255byte。

ShortString 的第一个字节的值等于字符串的实际长度。故而获取 ShortString 字符串实际长度的方法有 2 种：一是利用标准函数 Length()；二是直接利用第 1 个字节的值。例如：

```
Var
  Str;shortString;
Begin
  Str :=  'abcdefg';
  Writeln(inttostr(integer(str[0])));  //显示 7
  Writeln(inttostr(length(str)));  //显示 7
End.
```

2.AnsiString

AnsiString 类型也被称为长字符串。是 Delphi 最常用的类型，此类型的字符串由 ASCII 扩展字符集(参见附录 C)组成。很显然，这种类型字符串不能用于不同语言版本的操作系统，除非用户不介意看到一堆乱码。

AnsiString 实质上是一个指针类型，与普通的指针不同，此类型的指针专用于指向字符串。利用标准函数 Sizeof()可知此类型变量在内存中占用 4 个字节。所以 AnsiString 最小值为 4byte 而非 0byte，也就是说，一个空字符串占用 4 个字节。（眼熟吧？想想你的手机月租费）

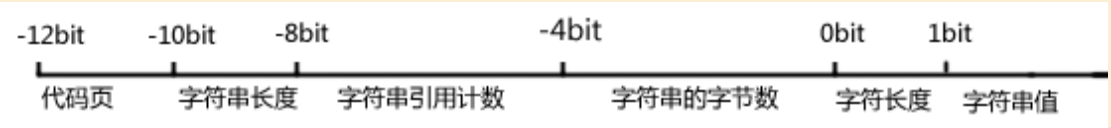
由于 AnsiString 类型的字符串占用的空间最大可达 2GB，为了节约内存，Delphi 将

AnsiString 字符串的字符串本身分配在堆中，而在栈中留下一指向字符串的指针，这个指针就是 AnsiString 类型的变量。不过就用户角度而言，完全可以将 AnsiString 类型的变量当成一个普通的简单变量，使用时可以像 ShortString 字符串一样进行读写或利用索引来获取 AnsiString 类型字符串中的任一字节的值。

使用字符串索引时有两点值得注意：索引应当从 1 开始，而不像数组从 0 开始，因为长字符串的第 1 个字节（即序数为 0 的字节）的值标识了这个字符串的每个字符由几个字节组成。

对于 AnsiString 类型字符进行索引时只能得到某个字节而不一定是某个字符的值，shortstring 类型亦如此。对其它类型如 widestring、unicodestring 等字符串索引时得到的是某个字符的值。这句话的意思可以通过下面的例子来明示：

```
Var
  strA,strB:AnsiString;
```




```
  Str_array:array[0..100] of ansichar;
Begin
  strA :=  'ABCDEFG';
  strB :=  '语言手册';
  Str_array :=  'ABCDEFG';
End.
```

Str_array[0]与 StrA[1]均表示字符'A'，而 strB[1]确并不表示'语'，为什么？因为 strB[1]只表示 strB 中第 1 个字节而非第 1 个字符的值，而不巧的很，汉字在 ASCII 扩展字符集由两个字节组成，所以 strB[1]的值只表示汉字'语'的第一个字节。

前面说过，Delphi 将字符串存储于堆中。我们就来看一下堆中的字符串实体的结构。其具体结构如下：

可以看到编译器在字符串的第一个字节之前插入了 12 个字节储存字符串的相关信息。

开始的的 2 个字节表示字符串所用的代码页，其值标识组成此字符串的字符集的版本。通俗一点，此区域标识这个字符串是由哪种字符集中的字符所组成。当然，作为通用语言，无论哪国文字，英文都是以照常显示的。

其次的 2 个字节的值为字符串中的字符数。

接下来的 4 个字节表示字符串的引用计数。

最后的 4 个字节的值为字符串的字节数，用于存储字符串的长度，32bit 能表示的最大值为2^{31} bytes = 2GB，恰好是此类型字符串的最大长度。当字符串全由英文字母组成时，组成字符串的字节数与组成字符串的字符数相等。

字符串的第一个字节标识了这个字符串的每个字符由几个字节组成。

现在，我们详细的看一下引用计数。再此之前先来看一段代码：

```
Var
  strA, strB:AnsiString;
Begin
  strA :=  'ABCDEFG';       //利用字符串常量'ABCDEFG'给 strA 赋值
  strB :=  strA;     //利用 strA 给 strB 赋值
  Writeln(strA); //显示 strA 的值
```

```
  Writeln(strB);
end.
```

运行完毕后，strA 与 strB 所代表的字符串均为'ABCDEFG'。我们主要看第 3 句：strB := strA;
问题：如下图，strB 与 strA 所指向的字符串究竟只是值一样，还是根本就是同一字符串？

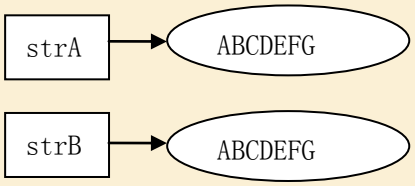


图 A

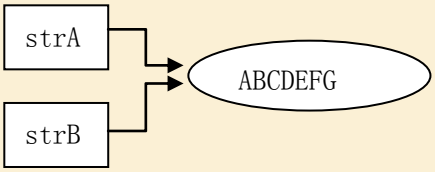


图 B

答案是后者。执行 strB := strA 时，系统并没有将 A 所指向的字符串复制一份后再用 strB 指向它，而是直接将 strB 指向了 strA 所指的字符串'ABCDEFG'。

其实这不奇怪：strA 与 strB 实质上均是指针，strA 的变量值为字符串的地址，而语句 strB := strA 的作用是将变量 strA 中的内容赋予 strB，故而执行完语句 strB := strA 后，变量 strB 与 strA 中的值完全相同，均为同一条字符串的地址，也就是说 strB 与 strA 指向同一个字符串。

虽然 Delphi 将字符串本身存储于堆中，但其所占用的内存由 Delphi 自动管理，用户完全不用费心字符串的内存管理。

对于字符串而言，当没有任何一个指针指向它时，此字符串就会被销毁，这与没人要的东西就会被当成垃圾扔掉是同一个道理。问题是，对于某个字符串而言，Delphi 如何知道是否有指针指向它？答案是引用计数。

引用计数表示指向某字符串的指针数量。一个字符串在堆中被创建时，一定有指针指向它(否则没必要创建这个字符串)，此时引用计数初始值为 1。当有其它指针指向此字符串时，其引用计数自动加 1；相反，当某个原本指向字符串的指针不再指向此字符串时，字符串的引用计数自动减 1。这样一来，当引用计数为 0 时，表示没有任何指针指向此字符串，此时可将字符串销毁。

接下来，我们还剩最后一个问题：对于有多个指针指向的某个字符串，若通过其中某些指针来改变字符串的值，其它的指针是否会断续指向改变后的字符串？

以图 B 为例，现通过语句：strA[1] := 'i'将字符串'ABCDEFG'的第一个字节的值'A'改为'i'，即 strA 所指向的字符串的值变为'iBCDEFG'，那么 strB 的值是多少？从理论上来说，strB 的值应当也随之变成'iBCDEFG'。但事实并非如此：strB 的值并未随 strA 而变化，依然是'ABCDEFG'。Delphi 处理此过程时执行了以下步骤：

- 复制字符串'ABCDEFG'；
- 将复制而来的新字符串的第一个字节的值改成'i'；
- 将 strA 指向修改后的新字符串：'iBCDEFG'；

也就是说，对于 strA 而言，当其改变字符串的值后，其所指的字符串已并非原来的字符串而经复制后形成的新字符串，而 strB 所指的原来的字符串并未受任何影响，仅仅只是引用计数减去 1 而已。这种处理机制称之为 Copy-on-Write 机制。当有多个指针指向同一字符串时，此机制会大加快处理速度。

疑问：AnsiString 可表示 MBCS，若超过 2 字节，其内部结构如何表示每个字符的长度呢？

3.WideString

WideString 与 AnsiString 基本一样，除以下几点：

- WideString 没有引用计数
- 其中容纳由 16bit unicode 字符组成的字符串
- 对于 WideString 类型的字符串索引会得到某个字符的值而非某个字节的值。
- 兼容于 COM 中的 BSTR 类型；实际上此类型最大的用途就在于此

4.UnicodeString

UnicodeString 与 AnsiString 基本相同，唯一的区别在于前者容纳的字符串由 Unicode 字符集中的字符组成。事实上 UnicodeString 的出现更多是为了弥补 WideString 在处理 Unicode 编码时的不足。在绝大多数场合，UnicodeString 比 WideString 更适合。但编写 COM 程序时，只能选择 WideString。注意对 unicodestring 字符串进行索引时得到的也是某个字符的值。

5.String

String 为 Delphi 的保留字，用作 UnicodeString 类型的一个别名。
在早期版本中如 Delphi7 当中包含编译指令{$H}，当其处于{$H+}状态时 string 等价于 Ansistring 类型；当其处于{$H-}状态时 string 等价于 shortstring 类型。Delphi2010 中此编译指令已被废弃，无论此指令处于何种状态 string 均等价于 UnicodeString。

作为 Delphi 的保留字，String 也用于声明自定义长度的 shortString 类型变量。默认情形下 shortString 类型的变量占用 256byte 的内存，我们可手动定义其占用的字节数：

```
Var
  Mystr:String[n];
```

Mystr 在内存中将只占 n+1 个字节。显然，N 的值不可超过 255。此声明相当于：

```
Var
  Mystr:array[0..n] of ansichar;
```

6.UCS4String

USC4String 类型容纳 USC4 字符集中的字符组成的字符串。其声明如下：

```
type
  UCS4String = array of UCS4Char;
```

此类型的字符串基本不用。在笔者看来，此类型现时只有两个用途：其一为方便

Delphi 在未来迁移时花费最小的代价，毕竟 UCS4 编码是将来的趋势；其二是用于各种编码的字符串的转换，其它编码的字符串转化至 UCS4String 不会有任何的数据丢失，故而此类型很适合作为字符串转换时的中间格式。

7.CodePaged AnsiString

从前面的叙述可知 AnsiString 用于容纳扩展 ASCII 字符集字符形成的字符串。事实上，这种说法并不精确。较为精确的说法是：AnsiString 能够容纳所有编码格式的字符串，只要计算机上安装了相应的字符集，但只有 AnsiString 字符串的编码为本地扩展 ASCII 字符集时才能被正确显示。

以笔者所用简体中文版操作系统，其本地扩展 ASCII 字符集为简体中文版。如此，只有 AnsiString 中容纳了简体中文字符串，计算机会正常显示。但如果笔者比较无聊非要将 Unicode 编码的字符串或是日本版 ASCII 扩展字符集编码字符串放入 AnsiString 中并意图在计算机上显示，那么我只能看到一堆乱码，除非字符串全由英文组成。

CodePaged AnsiString 部分的解决了此问题。通过指定一个代码页的值我们可以声明一个新的基于 AnsiString 的字符串类型，其格式为：

```
Type
  myString = type AnsiString(CODEPAGE);
```

CODEPAGE 为本地计算机已安装的代码页的值。当 MyString 类型的变量容纳指定代码页编码的字符串时可正常在计算机上显示。如 Delphi 中的 UTF8String 声明：

```
Type
  UTF8String = type AnsiString(CP_UTF8);
```

使用时读者完全可以将 UTF8String 当成是与 UnicodeString、AnsiString 一样的新的字符串类型。事实上，Delphi 内部也是这么做的。

从理念上来说，利用这种方法，只要在计算机上安装日文的代码页然后声明如下类型：

```
Type
  JPString = type AnsiString(CP_JAUTODETECT);
```

这个 JPString 应当完全与日文环境下的 AnsiString 等同。

8.RawByteString

RawByteString 的原型为：

```
Type
  RawByteString = type AnsiString($FFFF);
```

注意：$FFFF 不代表现有的任何编码方式。

在笔者看来，RawByteString 类型的变量与其称之为字符串，不如称之为字节容器更为贴切。此类型的变量可容纳任何编码方式编码的字符串且不进行任何的转换，其中的字符串不会有任何的变化，仿佛存储的不是字符串而是一堆字节一样。

使用 RawByteString 时注意：作为函数参数时，RawByteString 只能用作 const 或 value 方式传递的参数，不可用于 var 方式传递。

9.Nul 结尾字符串

Delphi 不存在 C 风格的 NULL 结尾的字符串。作为替代，可以使用以下 2 种方法来模拟：

- 利用序数从 0 开始的静态字符数组代替字符串。
- 使用字符指针 PChar 与 PWideChar。

注意：使用字符数组替代字符串时应保证开启了{$X+}指令，此指令默认为开启状态。

10.字符数组

利用静态字符数组可模拟字符串，这种形式的字符串与字符指针及长字符串相兼容，在使用时可以直接将任何类型的字符数组作为值赋给长字符串变量或字符指针 PChar、PWideChar。替字符数组赋值时不能将任何形式的字符串变量赋给字符数组，只能用字符串常量。

```
var
  a:array[0..5]of char;
  str:String;
  p:pchar;
begin
  a :=  'this is Delphi';
  p :=  @a;   //也可写成 p :=  @a[0]
  writeln(p);
  str :=  a;
  writeln(str);
  readln;
End.
```

将字符数组赋给字符指针时，实际上是将数组的地址赋给了指针变量，而数组名代表了数组的第一个成员，所以上面的赋值语句可改为：p := @a[0]，这两种形式完全等效。

10.Packed String

Delphi 中经常会用到 packed string 类型，其实质上是一个一维的静态 packed 字符数组。如：

```
var
  s1:packed array [1..5] of char
```

11.字符串指针

Delphi 中能作为字符串指针的只有两种：PChar 及 PWideChar。它们指向 Nul 结尾的字符串，与 C 字符串兼容。

得益于 Delphi 的优秀的编译器，在使用时我们并不感觉到字符串指针与一般的字符串变量的区别，（虽然一般的字符串也是一种特殊的指针）。对字符串指针初始化时我们可以直接使用字符串常量或是字符数组，但不能使用一般的字符串变量：

```
var
```

```
  a:array[0..255]of char;
  str:String;
  p:PChar;
begin
  p :=  a;
  p :=  str;
  p :=  'delphi';
  readln;
End.
```

鉴于这个特性，在例程需要一个 PChar 类型的参数时，我们可以直接传递一个字符串常量作为参数。

虽然不能将长字符串变量作为值赋给字符串指针，但反过来，我们可以将字符串指针作为值赋给任意类型的长字符串变量：

```
var
  str:String;
  p:PChar;
begin
  p :=  'delphi';
  str :=  p;
  writeln(str);
  readln;
End.
```

字符串指针也可以字符串变量一样进行索引从而得到字符串的某个字符的值：

```
var
  str:String;
  p:PChar;
begin
  str :=  'UK 英国';
  p :=  pchar(str);
  writeln(p[2]); //显示“英”
  readln;
End.
```

我们可以利用字符串连接符将字符指针与一个长字符串边成一条字符串，在这种情形下，编译器会先将字符指针转换成一条 UnicedeString 类型的字符串，然后再进行连接运算。事实上，在任何字符串的二元运算（如相加、赋值）中，字符指针均被转换成 UnicodeString 类型的字符串后才会进行具体运算。我们可以手动通过强制转型来实现这个过程。

强制转型可以将 PChar 类型的变量转换成 UnicodeString 类型的变量，也可将 UnicodeString 及 AnsiString 类型的变量转换成 C 风格的以 Null 结尾的字符串。具体的转换规则如下：

1. 用于强制转换的类型名称必须与字符串相对应，例如若 S 是一个 UnicodeString 类型的字符串，通过 PChar(S)或 PWideChar(S)可以将 S 转换成 Null 结尾的字符串并返回一个指针；若 S 是 AnsiString 类型则应当使用 PAnsiChar 来进行转换。

2. 利用 Pointer(S)可以将长字符串变量 S 转换成一个无类型指针，如果 S 是个空字符串，则转换为空指针 nil。类似的，PChar(S)将返回一个指向 S 的字符串值的指针，空字符串时同样返回 nil 指针。
3. 将字符串变量转换成指针后，通过指针来改变字符串的值会影响到字符串变量的值，而通过字符串变量来改变字符串的值却并不会影响指针所指向字符串的值。这句话比较绕口，我们看一个例子：

```
var
  str1,str2:String;
  p1,p2:PWideChar;
begin
  str1 :=  'ABCD';
  str2 :=  'ABCD';
  p1 :=  PChar(str1);
  p2 :=  PChar(str2);

  str1 :=  'DEFG';
  p2[0] :=  'X';

  writeln(str1);
  writeln(P1);
  writeln(str2);
  writeln(P2);
  readln;
End.
```

输出值为：

```
DEFG
ABCD
XBCD
XBCD
```

我们来分析一下，str1、str2 的初值为'ABCD'，P1 与 P2 分别指向 str1 与 str2。我们在例子中将 str1 的值改成'DEFG'，将 P2 指向的字符串的第一个字符改成了'X'，也就是说 P2 此时指向的字符串值为'XBCD'。根据输出值我们可以发现，在将 str1 及 P2[0]的值改变后，P1 指向的值没有随着 str1 的改变而变，而 str2 则随着 P2 的变化而产生了变化。

最后，我们稍微来看一下 PAnsiChar 类型的指针。虽然 AnsiString 在 Delphi 中普遍应用，但 PAnsiChar 却很少使用，因为它不能指向以 Nul 结尾的字符串。对 PAnsiChar，读者记住以下几点：

1. 同其它类型一样，可以使用一个字符串常量作为初值赋给 PAnsiChar 类型的变量。
2. 将一个非 AnsiString 的字符串变量 str 通过强制转型 p := PAnsiChar(str)后，得到的指针只能引用字符串中的第一个字节的值，例如：

```
var
  str:String;
  p:PAnsiChar;
begin
```

```
  str :=  '我';
  p :=  pANSIchar(str);
  writeln(p);
  readln;
End.
```

3. 只能将 AnsiString 类型的值赋给 PAnsiChar 类型的变量。
4. 对 PAnsiChar 指针索引会返回某个字节的值而不一定是某个字符的值。

附录 E 文件读写

文件读写指从一个外部文件中读取或向其中写入内容。在本节我们先介绍利用传统的利用文件变量来读写外部文件，在介绍完这部分内容后，我们会简要介绍如何使用流来操作文件。

我们先熟悉一下读写指针的概念。所谓读写指针是指读写文件、流等内容时，当前正在读或写的位置。新建一个文件并往其中写入内容时，其读写位置一定是文件的开始，除非你有本事在不写入内容的情况下跳过文件的开始直接将内容写在文件的末尾。当写入了所有的内容后当前文件的读写指针所处位置一定是文件的末尾。至于这个位置是如何计算的，各种情形下均有具体的规定。

E.1 利用文件变量进行读写

外部文件可以是磁盘文件，也可以是某个硬件设备如显示器。Delphi 中的文件可分为三类：类型文件、非类型文件、文本文件。

类型文件由多个 record 类型的值组成，其内部具有规律性，相当于将同一类型的多个变量值放在一起并写入文件，其中的每个变量值称为一个记录。

非类型文件以字节为单位组成整个文件，其内容没有规律性，这类文件纯粹就是多个字节堆积而成。理论上可以将任何数据都写入非类型文件。

文本文件严格上来说属于非类型文件，它由一连串连续的字符序列组成，每个字符序列以一个行末符号结尾，在.txt 文件中表现为多行文字。

E.1.1 类型文件

类型文件变量的声明格式如下：

```
var
  变量名：file of 类型名;
```

文件的类型由类型名决定。

Delphi 提供了各个方面的例程用于操作文件，包括打开文件、读取文件、写入文件等一系列例程。本节后面的列表中列出了所有的相关例程，此处不再列出。

Delphi 提供了三个用于打开文件的例程：Reset、Rewrite、Append。Reset 用于打开一个已经存在的文件；Rewrite 新建一个指定的文件并打开，若需要新建的文件已经存在，则 rewrite 会先删除它然后再新建一个同名文件；Append 也用于打开一个已经存在的文件，它与 reset 的区别在于：用 reset 打开的文件在写入内容时会从文件的开头写入，

即刚打开文件时读写指针处于文件的开头，而 Append 则会从文件的末尾写入，读写位置处于文件的末尾。

Delphi 提供 Write 及 Read 用于读写文件中的内容：

```
procedure Read(var F: file; V1[,V2,V3…]);
procedure Write(var F: file; V1[,V2,V3…]);
```

Read 用于从 F 中读取若干字节内容并将其存储入 V1 所代表的变量中。读取的字节数由 V1 所占用的字节数决定，如 V1 为 Char 类型时，系统会从 F 中读取 2 个字节的内容至 V1 中。若在 Read 指定了 V2 或更多的参数，系统会依次读取内容并存储入 V1，读取内容存储入 V2…，读取内容至 Vn 中。　Write 与 Read 相反，它将 V1、V2…Vn 中的内容写入 F 中。

Delphi 的文件变量实际上是一个指针，指向一个外部文件，在读写文件时所有的内容实质上均是在这个外部文件中读写，故而在使用文件变量前应当调用标准例程 AssignFile 将文件变量与一个外部文件相关联；在结束对文件的操作后也要调用 CloseFile 来断开变量与外部文件间的关联。

下面的例子展示了这些例程的用法：

```
Type
  TRec = record
    ch1:AnsiChar;
    ch2:AnsiChar;
  end;
var
  Rec1,Rec2:TRec;
  n:integer;
  path:string;
  F:file of TRec; //声明 TRec 类型的文件
begin
  Rec1.ch1 := 'A';
  Rec1.ch2 := 'B';

  writeln('请输入文件名：');
  Readln(path);

  AssignFile(F,path); //关联外部文件
  if FileExists(path) then //判断文件是否存在
    Reset(F)
  else   //若不存在则新建这个文件
    Rewrite(F);
  for n := 1 to 9 do
    Write(F,Rec1); //连续写入 9 个 Rec1 变量值
  Rec2.ch1 := '-';
  Rec2.ch2 := '-';
  Write(F,Rec2); //将 Rec2 的值写入 F 指向的文件
  CloseFile(F); //断开关联
  writeln('文件操作完毕');
```

```
  readln;
end.
```

现在，请读者打开 path 指向的文件，看一下其中的内容，有没有发现有什么变化？

为方便读者应用，我们再举一个例子，在这个例子中我们会实现一个程序，用于将一个文本中的内容复制至另一个文本中：

```
program CopyText;

{$APPTYPE CONSOLE}

uses
  SysUtils;
var
  Source,Target:String;
  SourceF,TargetF:TextFile;
  ch:AnsiChar;
begin
  Writeln('请输入源文件名称：');
  Readln(Source);
  Writeln('请输入目标文件名称：');
  Readln(Target);

  AssignFile(SourceF,Source);
  Reset(SourceF);
  AssignFile(TargetF,Target);
  Rewrite(TargetF);

  while not Eof(SourceF) do
  begin
    Read(SourceF,ch);
    Write(TargetF,ch);
  end;

  CloseFile(TargetF);
  CloseFile(SourceF);

  Writeln('操作完毕');
  readln;
end.
```

Delphi 的 system 单元中定义了两个全局变量：Input 及 Output。这两个变量使用于命令行程序中，分别代表计算机的标准输入出输出（一般是键盘与显示器）。当上面介绍的读写例程在使用时省略了文件变量，则默认的文件变量为 Input 及 Output。所以以下两组代码等效：

```
Readln(path);
```

```
 writeln('文件操作完毕');

 Readln(Input,path);
 writeln(Output,'文件操作完毕')
```

E.2.2 非类型文件

非类型文件直接使用 file 声明。非类型文件变量不存在文件类型，所以其后面不用接 of...。

除 write 与 read 外，适用于类型文件的所有例程均适用于非类型文件。Delphi 另外提供了两个例程供读写非类型文件：BlockRead、BlockWrite。二者的声明为：

```
 function BlockRead(var F:file;var buf;count:Integer):Integer;
 function BlockWrite(var F:file;const buf;count:Integer):Integer;
```

BlockRead 从 F 中读取数目为最多 count 字节的内容并存储至 Buf 所指向的内存中。
BlockWrite 将 Buf 中最多 Count 字节的内容写入 F 中。

```
var
  sourcename,targetname:string;
  FSource,FTarget:file;
  Buf:PAnsiChar;
begin
  try
    writeln('请分别输入文件名：');
    Readln(sourcename);
    Readln(targetname);
    AssignFile(FSource,sourcename);
    AssignFile(FTarget,targetname);
    Reset(FSource);
    Rewrite(FTarget);

    while eof(FSource) = False do
    begin
      BlockRead(FSource,Buf,1);
      BlockWrite(FTarget,Buf,1);
    end;

    CloseFile(FSource);
    CloseFile(FTarget);
  except
    writeln('error');
  end;
  writeln('文件操作完毕');
  readln;
end.
```

非类型文件属于低层次操作，一般不推荐使用。

E.1.3 文本文件

文本文件使用 TextFile 声明:

```
var
  变量名：TextFile;
```

首先声明：文本文件不等于后缀名为.txt 的文件，.txt 文件只是 Windows 系统上的一种文件类型，它与 Delphi 中的三种类型的文件没有任何关系，硬说有也只是同名而已。从文件结构方面来看，所有的一行一行的字节序列组成的文件均可以称为文本文件。适用于类型文件的例程均可用于文本文件，但是用于打开文件的例程在操作文本文件时效果稍微有点不同：reset 打开的文本文件为只读文件，只能从其中读取内容；而 rewrite 及 append 打开的文件为只写文件，只能向其中写入内容。

Delphi 还提供另外两个例程 WriteLn 和 ReadLn 用于操作文本文件，writeln 直接向文本文件中写入一行内容，readln 则直接从文本文件中读取一行内容。

以下为示例：

```
var
  str:string;
  n:integer;
  path:string;
  F:textfile;
begin
  writeln('请输入文件名：');
  Readln(path);

  AssignFile(F,path);
  if FileExists(path) then
    Reset(F)
  else
    Rewrite(F);

  str :=  'Delphi';
  for n :=  1 to 5 do
    write(F,str);

  writeln(f,'end');

  CloseFile(F);
  writeln('文件操作完毕');
  readln;
end.
```

打开文件后看一下其中的内容，有没有发现什么？

标准文件处理例程

Append	打开一个已经存在的文件
AssignFile	将文件变量与一个外部文件相关联
BlockRead	从非类型文件中读取一定数目的字节
BlockWrite	向非类型文件写入一定数目的字节
ChDir	改变当前目录
CloseFile	断开一个文件变量与外部文件的关联
Eof	判断读写指针是否在文件末尾
Eoln	判断读写指针是否在行末
Erase	通过文件变量删除一外部文件
FilePos	改变读写指针的当前位置
FileSize	返回文件所具有的记录数目
Flush	清洗输出文件的缓冲区
MkDir	创建一个目录
Read	从文件中读取内容
Readln	从文件中读取一行内容
Rename	通过文件变量重命名一个外部文件
Reset	打开一个外部文件
Rewrite	新建并打开一个外部文件
RmDir	删除一个空的子目录
Seek	将文件的读写指针移动到任意位置。此例程不能用于文本文件
SeekEof	与 seek 相同，仅用于文本文件
SeekEoln	与 Eoln 相同，仅用于文本文件
SetTextBuf	为文本文件设置一个缓冲区
Truncate	将一个非文本文件从当前位置截断
Write	向文件中写入内容
Writeln	将写入文件内容作为一行写入文件中

E.2 使用流操作文件

通过文件变量我们可以读写任何形式的文件，这种方式读写一个文件需要经过如下步骤：

- 利用 AssignFile 将外部文件与文件变量关联
- 利用 ReWrite、ReSet 或 Append 打开文件，以备读写
- 利用 write 写入内容或 read 读取内容
- 利用 CloseFile 关闭一个文件变量

利用以上步骤的确可以操作文件，但这种方式并不是面向对象思维的产物。在面向对象时代的编程应当一切以对象为主，接下来我们介绍的流就是这样一种对象。

如上面所言，流对象以面向对象的方式提供了所有与文件相关的操作，使用流对象我们可以将以上的四个步骤写成如下形式：

```
var
  Stream:TFileStream;
  str:AnsiString;
begin
```

```
  str :=  'handing files by Stream';
  Stream :=  TFileStream.Create('d:\ip.txt',fmOpenWrite); //关联并打开文件
  Stream.Write(PChar(str)^,length(str)); //将 str 写入文本
  Stream.Free;   //关闭文件
end.
```

Delphi 中所有的预定义的流均定义于 classes 单元中，所以在使用流之前请确保你的程序已经在 uses 从句中引用了这个单元。如果的回答是肯定的，那么我们开始。

Delphi 中所有的流均继承于 TStream。其中定义了所有的流都会用到的公共的方法和属性，我们功利一点，将那些我们无法使用或用得非常少的成员去掉，剩下的部分如下：

```
  function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
  function Write(const Buffer; Count: Longint): Longint; virtual; abstract;

  function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;

  procedure ReadBuffer(var Buffer; Count: Longint);
  procedure WriteBuffer(const Buffer; Count: Longint);

  function CopyFrom(Source: TStream; Count: Int64): Int64;
  property Position: Int64 read GetPosition write SetPosition;
  property Size: Int64 read GetSize write SetSize64;
```

看到没有？其中声明了抽象方法，所以 TStream 是个抽象类。这句话的潜台词是：在任何时候都不要直接创建 TStream 的对象。

由于这些成员是所有类中共用的成员，虽然其中的方法在子类中的实现可能有所不同，但具体的作用都是相同的。我们马上会简要介绍这些成员的作用。在接下来的内容中，如无必要我们不会再重复介绍子类中的这些成员。

Position 属性标识了读写指针的位置，而 Size 属性则标识了流所占用的内存的大小。当读写指针处于流的末端时，这两个属性值相等。

CopyFrom 方法用于将指定流中的所有内容复制到当前流中。seek 用于将读写指针移动到任意位置。这个方法在各个类中有多个重载版本，但作用没有太大变化。

其余两组方法用于读写流，它们在不同的子类中实现变化较大，我们会具体介绍。

由于本节内容重点在于介绍如何利用流读写文件而非流本身，所以我们只介绍 TStream 的三个子类，分别是用于读写文件的文件流 TFileStream、用于读写内存的内存流 TMemoryStream、用于读写字符串的字符串流 TStringStream。

E.2.1 文件流

TFileStream 继承于 TStream 的直接子类 THandleStream，所以其中的成员包括 THandleStream 的成员，实际上二者的成员并没有太大的差别。相对于 THandleStream，TFileStream 仅仅新加入了一个属性 FileName，用于标识正在读写的文件的名称。尽管如此，文件流中还是有不少的成员需要掌握。我们重点介绍以下 4 个成员：

```
TFileStream = class(TStream)
  ...
  constructor Create(const AFileName: string; Mode: Word);overload;
  function Read(var Buffer; Count: Longint): Longint;override;
```

```
  function Write(const Buffer; Count: Longint): Longint;override;
  function Seek(const Offset: Int64; Origin: TSeekOrigin):Int64;override;
  ...
end;
```

首先是第一个重载的构造函数。除非你对流已有足够理解，否则请尽量使用此函数来创建文件流对象。其中的 AFileName 表示目标文件，这个从名称上就可看出。重点在于 Mode 参数，它标识了文件流对象以体种方式与目标文件关联，其值有四，各自所代表的意义为：

值	含义
fmCreate	AFileName 指定的文件不存在时会创建这个文件并以只写方式打开，若文件已经存在则清空原有内容并以只写方式打开
fmOpenRead	将文件以只读方式打开，若文件不存在，你自己想想会怎样
fmOpenWrite	以只写方式打开文件，且写入的内容会取代文件原本的内容
fmOpenReadWrite	打开的文件可读也可写

另一组表示文件共享模式的值有五个，各自意义如下表所示：

值	含义
fmShareCompat	Sharing is compatible with the way FCBs are opened.
fmShareExclusive	独占当前文件，其它程序无法访问
fmShareDenyWrite	其它程序只能通过只读方式打开文件
fmShareDenyRead	其它程序只能通过只写方式打开文件
fmShareDenyNone	不对文件作任何限定

Read 用于读取 Count 数目字节的内容至 Buffer 所标识的缓冲区中。当读写指针到达流的末端时即使 Read 读取的字节数小于 Count 也会结束读取。若此函数返回一个大于非 0 值表示执行成功，返回读取的字节数目。

Write 用于从 Buffer 中写入 Count 数目字节的内容到流中，若成功执行则返回写入的字节数目。

Seek 的 Offset 参数表示读写指针移动的字节数，从休息开始移动由 TSeekOrigin 决定。TSeekOrigin 参数值有三种：soFromBegining、soFromCurrent、soFromEnd。不知读者是否发现，Offset 的值可以为负数，那么若这个参数值为负数时，读写指针会如何移动呢？答案是指针会后退。所以以下操作会使读写指针回到流的开始：

```
    obj.Seek(-obj.size,soFromEnd);
```

下面是 TFileStream 的使用示例，读者可比较一下文件流操作与文件变量操作的不同：

```
uses
  SysUtils, Classes;
var
  Stream:TFileStream;
  target:string;
  str:AnsiString;
begin
  writeln('请输入一个字符串：');
```

```
  Readln(str);
  writeln('请输入目标文件名：');
  readln(target);
  Stream :=  TFileStream.Create(target,fmCreate);
  Stream.Write(pchar(str)^,length(str));
  Stream.Free;
end.
```

E.2.2 内存流

内存流在运行时会将文件内容复制到内存中，对其内容所作的任何修改在手动保存前均不会影响到与其关联的文件的内容。所以读者最好不要使用内存流读取一个体积非常大的文件，这样做会有可能会导致计算机内存不足。

TMemoryStream 中需要掌握的成员如下：

```
function Write(const Buffer; Count: Longint): Longint; override;
function Read(var Buffer; Count: Longint): Longint; override;
function Seek(Offset: Longint; Origin: Word): Longint; override;
procedure Clear;
procedure LoadFromStream(Stream: TStream);
procedure LoadFromFile(const FileName: string);
procedure SaveToStream(Stream: TStream);
procedure SaveToFile(const FileName: string);
property Memory: Pointer read FMemory;
```

write、read、seek 与文件流的区别不是太大，在此不赘述。clear 方法从名称就可知道其用于清空内存流中的所有内容。Memory 为一个 pointer 类型的指针，指向内存流在内存中的位置。

LoadFromFile 用于从一个磁盘文件中载入数据至内存流，相对地，SaveToFile 则将内存流中的数据存入磁盘文件。与之类似，LoadFromStream 用于从另一个流中装载数据，而 SaveToStream 则将内存流中的数据存入另一个流中。注意在使用 LoadFromX 时，内存流中原来的内容会被丢弃；使用 SaveToX 时文件的原有内容也会被覆盖。

再次警告：不要用 LoadFromFile 装载体积非常大的磁盘文件。

下面的程序会将源文件的内容复制一份，然后将复制的内容与原有内容一道写入目标文件中：

```
uses
  SysUtils, Classes;

var
  source,target:String;
  o1,o2:TMemoryStream;
begin
  writeln('输入源文件名：');
  readln(source);
  writeln('输入目标文件名：');
  readln(target);
```

```
  o1 :=  TMemoryStream.Create;
  o1.LoadFromFile(source);
  o2 :=  TMemoryStream.Create;
  o2.LoadFromStream(o1);
  o1.WriteBuffer(o2.Memory^,o2.Size);//不能用 LoadFromStream
  o1.SaveToFile(target);
  FreeAndNil(o1);
  FreeAndNil(o2);
  writeln('ok');
  readln;
end.
```

E.2.3 字符串流

字符串流有点奇特，它在不同的系统环境下有不同的表现。在笔者的计算机上，它继承于内存流的子灰 TBytesStream，可能在读者的计算机有所不同。由于字符串流 TStringStream 继承于内存流，所以其中含有所有的内存流的方法及属性。相比内存流，字符串流中的新成员不是太多，其中需要注意的有：

```
    function ReadString(Count: Longint): string;
    procedure WriteString(const AString: string);
    property DataString: string read FDataString;
```

ReadString 用于从字符串流中读取并返回一个长度为 Count 的字符串，字符串中可包含换行符。WriteString 则将 AString 指定的字符串当前流中。DataString 则存储字符串流中的所有字符串，换句话说通过 writeln 显示这个属性值，系统会将当前流中的所有字符串都显示于窗口中。

下面是例子：

```
var
  Stream:TStringStream;
begin
  Stream :=  TStringStream.Create;
  Stream.WriteString('d');
  Stream.WriteString('e');
  Stream.WriteString('l');
  Stream.WriteString('p');
  Stream.WriteString('h');
  Stream.WriteString('i');
  writeln(Stream.DataString);
  FreeAndNil(Stream);
  readln;
end.
```

E.3 直接操作文件

Delphi 提供了一组不使用文件变量而是直接读写文件的例程：

```
function FileOpen(const FileName: string): Integer;
function FileCreate(const FileName: string): Integer; overload;
function FileRead(Handle: Integer; var Buffer; Count: Cardinal): Integer;
function FileSeek(Handle: Integer; Offset: Int64; Origin: Integer): Integer;
function FileWrite(Handle: Integer; const Buffer; Count: Cardinal): Integer;
procedure FileClose(Handle: Integer);
```

FileOpen 用于打开 FileName 指定的文件，Mode 表示以何种方式打开，这个参数的值与 TFileStream 中的 Mode 完全相同，在此不赘述。

成功执行时 FileOpen 返回所打开的文件的句柄。若返回-1 则表示执行期间出现了错误，这点与其余四个例程一致。FileOpen 返回的句柄是其它五种例程使用的根本。

FileCreate 用于创建一个新文件并返回其句柄，若 FileName 指定的文件已经存在则清空其所有的内容后打开并返回其句柄。

FileRead 及 FileWrite 用于读写文件，执行成功时返回所读取或写入的字节数。

FileSeek 将当前的读写指针移动到任意位置，其中的 origin 参数有 0、1、2 三个值，分别表示文件开头、当前位置、文件末尾。执行成功时此函数返回读写指针的新位置。这组例程未提供判断类似于 EOF 及 FileSize 的例程，在需要使用到类似功能时可以使用 FileSeek 代替。

最后一个 FileCreate 不用说，读者也知道它用于关闭文件。其中的 Handle 参数表示文件句柄，此过程只能关闭由 FileOpen 或 FileCreate 打开的文件。

```
var
  HSource,HTarget: Integer;
  source,target: String;
  text: array [0..99] of AnsiChar;
  i:integer;
begin
  writeln('输入源文件名：');
  readln(source);  //读取源文件名
  writeln('输入目标文件名：');
  readln(target);  //读取目标文件名

  HSource :=  FileOpen(Source,fmOpenRead);
  HTarget :=  FileCreate(Target,fmOpenWrite);

  i :=  FileSeek(HSource,0,2); //判断文件大小
  FileSeek(HSource,-i,2); //将读写指针移至文件开始
  while FileSeek(HSource,0,1) < i do //判断是否到达文件末尾
  begin
    FileRead(HSource,text,100);
    FileWrite(HTarget,text,100);
  end;
  FileClose(HSource);
  FileClose(HTarget);
  writeln('ok');
  Readln;
```

end.