7. 各种各样的语言特征

7.1. 局部(嵌套)例行程序（routines）

在大量的*例行程序（routine)*(函数,procedure(过程),方法)中,你可以定义一个帮助器(helper)例行程序(routine).
局部(嵌套)例行程序(routine)可以自由地访问(读和写)一个父类(parent)的所有参数,*和上述被声明父类(parent)的所有的局部变量*. 这是非常强大的.它通常允许拆分长的例行程序(routine)到几个小的例行程序中,而不需要很多努力 (因为你不必传递参数中所有的信息).
注意不要过度使用这个特征—如果一些嵌套的函数使用(甚至更改)父类(parent)的相同变量,代码可能很难理解(follow).
这两个示例是相等的:

```
procedure SumOfSquares(const N: Integer): Integer;

  function Square(const Value: Integer): Integer;
  begin
    Result := Value * Value;
  end;

var
  I: Integer;
begin
  Result := 0;
  for I := 0 to N do
    Result := Result + Square(I);
end;
```

另一个版本,在这里我们让局部(嵌套)例行程序(routine)Square 来直接访问 I:

```
procedure SumOfSquares(const N: Integer): Integer;
var
  I: Integer;

  function Square: Integer;
  begin
    Result := I * I;
  end;

begin
  Result := 0;
  for I := 0 to N do
    Result := Result + Square;
end;
```

局部(嵌套)例行程序(routine)可以转到任何深度—这意味着你可以在其它的局部(嵌套)例行程序(routine)内定义一个局部(嵌套)例行程序(routine).所以你可以转到偏远地区(但是请

不要转到*太偏远地区*,不然代码将成为不可读:).

7.2. Callbacks（回调）（亦称 events（事件），亦称 pointers to functions（指针到函数），亦称 procedural variables（过程变量））

它们允许间接地调用一个函数,通过一个变量.变量可以在运行时被分配到点到任何*带有匹配参数类型和返回类型的*函数.
callback(回调)可以是:

- 正常的,这意味着它可以点到任何正常的例行程序(routine)(不是一个方法,不是局部(嵌套)).

```
{$mode objfpc}{$H+}{$J-}

function Add(const A, B: Integer): Integer;
begin
  Result := A + B;
end;

function Multiply(const A, B: Integer): Integer;
begin
  Result := A * B;
end;

type
  TMyFunction = function (const A, B: Integer): Integer;

function ProcessTheList(const F: TMyFunction): Integer;
var
  I: Integer;
begin
  Result := 1;
  for I := 2 to 10 do
    Result := F(Result, I);
end;

var
  SomeFunction: TMyFunction;
begin
  SomeFunction := @Add;
  Writeln('1 + 2 + 3 ... + 10 = ', ProcessTheList(SomeFunction));

  SomeFunction := @Multiply;
  Writeln('1 * 2 * 3 ... * 10 = ', ProcessTheList(SomeFunction));
end.
```

- 一个方法: 在尾部带有 `of object` 的声明.

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils;

type
  TMyMethod = procedure (const A: Integer) of object;

  TMyClass = class
    CurrentValue: Integer;
    procedure Add(const A: Integer);
    procedure Multiply(const A: Integer);
    procedure ProcessTheList(const M: TMyMethod);
  end;

procedure TMyClass.Add(const A: Integer);
begin
  CurrentValue := CurrentValue + A;
end;

procedure TMyClass.Multiply(const A: Integer);
begin
  CurrentValue := CurrentValue * A;
end;

procedure TMyClass.ProcessTheList(const M: TMyMethod);
var
  I: Integer;
begin
  CurrentValue := 1;
  for I := 2 to 10 do
    M(I);
end;

var
  C: TMyClass;
begin
  C := TMyClass.Create;
  try
    C.CurrentValue := 1;
    C.ProcessTheList(@C.Add);
    Writeln('1 + 2 + 3 ... + 10 = ', C.CurrentValue);

    C.CurrentValue := 1;
    C.ProcessTheList(@C.Multiply);
    Writeln('1 * 2 * 3 ... * 10 = ', C.CurrentValue);
  finally FreeAndNil(C) end;
```

```
end.
```

注意你*不能* 像方法一样传递全局 procedures(过程)/函数. 它们是不兼容的. 如果你不得不提供一个 of object callback(回调), 除了不应该创建一个假程序类实例外, 你可以像方法一样传递类方法.

```
type
  TMyMethod = function (const A, B: Integer): Integer of object;

  TMyClass = class
    class function Add(const A, B: Integer): Integer
    class function Multiply(const A, B: Integer): Integer
  end;

var
  M: TMyMethod;
begin
  M := @TMyClass(nil).Add;
  M := @TMyClass(nil).Multiply;
end;
```

可惜, 你需要写怪模样的@TMyClass(nil).Add 代替合理的@TMyClass.Add.

- 一个(可能地)局部(嵌套)例行程序(routine):在尾部带有 is nested 声明, 并确保对代码使用{$modeswitch nestedprocvars}指令. 它们与 These go hand-in-hand with 局部 (嵌套)例行程序(routines)处于并进的状态.

7.3. 泛型(Generics)

任何的现代语言的一个强大的特征。一些事(通常, 属于一个类)的定义可以用其它的类型参数化。最典型的示例是，当你需要创建一个容器(container)(一个列表, 字典, 树, 图表…)时: 你可以定义*类型 T 的一个列表*, 然后*指定(specialize)*它来立即获得*整型的一个列表*, *字符串的一个列表*, *TMyRecord 的一个列表*, 等等.
在 Pascal 中的泛型更像在 C++中泛型的实现。这意味着在专门化（specialization）时它们是*"扩展的"*，有*一点*像宏(但是比宏更安全；例如，标识符在泛型定义事被分解，而不是在专门化（specialization）时，因此当专门化（specializing）泛型时，你不能"注入"任何的异常行为)。事实上, 这意味着它们是非常快的(对每个详细的（particular）类型可以被最优化)，并且与任何大小的类型一起工作。当当专门化（specializing）泛型时，你可以使用一个 基本的（primitive）类型 (整型，浮点)，一个记录，一个类。

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils;

type
  generic TMyCalculator<T> = class
    Value: T;
    procedure Add(const A: T);
  end;

procedure TMyCalculator.Add(const A: T);
```

```
begin
  Value := Value + A;
end;

type
  TMyFloatCalculator = specialize TMyCalculator<Single>;
  TMyStringCalculator = specialize TMyCalculator<string>;

var
  FloatCalc: TMyFloatCalculator;
  StringCalc: TMyStringCalculator;
begin
  FloatCalc := TMyFloatCalculator.Create;
  try
    FloatCalc.Add(3.14);
    FloatCalc.Add(1);
    Writeln('FloatCalc: ', FloatCalc.Value:1:2);
  finally FreeAndNil(FloatCalc) end;

  StringCalc := TMyStringCalculator.Create;
  try
    StringCalc.Add('something');
    StringCalc.Add(' more');
    Writeln('StringCalc: ', StringCalc.Value);
  finally FreeAndNil(StringCalc) end;
end.
```

泛型不是限定到类，你也可以有泛型函数和过程（procedures）：

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils;

{ Note: this example requires FPC 3.1.1 (will not compile with FPC 3.0.0 or older). }

generic function Min<T>(const A, B: T): T;
begin
  if A < B then
    Result := A else
    Result := B;
end;

begin
  Writeln('Min (1, 0): ', specialize Min<Integer>(1, 0));
  Writeln('Min (3.14, 5): ', specialize Min<Single>(3.14, 5):1:2);
  Writeln('Min (''a'', ''b''): ', specialize Min<string>('a', 'b'));
end.
```

关于重要的使用泛型的标准类型，参考容器(lists(列表),dictionaries(词典))使用泛型。

7.4. Overloading（重载）

方法(全局函数和 procedures（过程）)带有相同的名称是被允许的，只要它们有不同的参数。在编译时，编译器检查出你想使用的一个，知道你传递的参数。
在默认情况下，重载使用 FPC 途径（approach），这意味着在给定命名空间（一个类或一个单元）中的所有的方法是相等的，并隐藏在命名空间中其它的带有优先级较低的方法。例如，如果你定义一个带有方法 Foo(Integer)和 Foo(string)的类 ，它继承自一个带有方法 Foo(Float)的类，那么，你新类的用户将不能容易地(它们仍然能 --- 如果它们类型强制转换类为它的原型（ancestor）类型)访问方法。为克服这个问题，使用 overload 关键字。

7.5. 预处理程序

你可以使用简单的预处理程序指令，对于

- 条件编译(代码依赖于平台，或一些自定义开关（switches）)，
- 来包含在其它文件中一个文件，
- 你也可以使用无参数宏。

注意，带有参数的宏是不被允许的。通常，你应该避免使用预处理的原料（stuff）… 除非它真是合乎情理。预处理发生在语法分析前，这意味着你可以"打断"Pascal 语言的正常语法。这是一个强有力的，但也有些令人厌恶的特征。

```
{$mode objfpc}{$H+}{$J-}
unit PreprocessorStuff;
interface

{$ifdef FPC}
{ This is only defined when compiled by FPC, not other compilers (like Delphi). }
procedure Foo;
{$endif}

{ Define a NewLine constant. Here you can see how the normal syntax of Pascal
  is "broken" by preprocessor directives. When you compile on Unix
  (includes Linux, Android, Mac OS X), the compiler sees this:

    const NewLine = #10;

  When you compile on Windows, the compiler sees this:

    const NewLine = #13#10;

  On other operating systems, the code will fail to compile,
  because a compiler sees this:

    const NewLine = ;

  It's a *good* thing that the compilation fails in this case -- if you
```

```
  will have to port the program to an OS that is not Unix, not Windows,
  you will be reminded by a compiler to choose the newline convention
  on that system. }

const
  NewLine =
    {$ifdef UNIX} #10 {$endif}
    {$ifdef MSWINDOWS} #13#10 {$endif} ;

{$define MY_SYMBOL}

{$ifdef MY_SYMBOL}
procedure Bar;
{$endif}

{$define CallingConventionMacro := unknown}
{$ifdef UNIX}
  {$define CallingConventionMacro := cdecl}
{$endif}
{$ifdef MSWINDOWS}
  {$define CallingConventionMacro := stdcall}
{$endif}
procedure RealProcedureName; CallingConventionMacro; external 'some_external_library';

implementation

{$include some_file.inc}
// $I is just a shortcut for $include
{$I some_other_file.inc}

end.
```

Include 文件通常有 .inc 扩展名，并且被用于两个目的：

- Include 文件可能仅包含其它编译器指令，这"配置"你的源文件代码。例如，你可以创建一个带有这些内容的文件 myconfig.inc：

```
{$mode objfpc}
{$H+}
{$J-}
{$modeswitch advancedrecords}
{$ifndef VER3}
  {$error This code can only be compiled using FPC version at least 3.x.}
{$endif}
```

  现在你可以在你所有的源文件中使用{$I myconfig.inc}包含这个文件。
- 其它常见的用法是分裂一个大的单元到一些文件中，直到语言规则被过度使用为止，仍然保持它为一个单个的单元。不要过度使用这个技巧—你的第一个直觉应该是来分裂一个单个单元到多个单元，而不是分裂一个单个单元到多个 include（包含）文件。

永不减少，这是一个有用的技巧。

1. 它允许来避免"骤增"单元的数字，然而仍然保持你的源文件代码文件简短。例如，有一个带有 *"通常使用的 UI 控件"*的单个的单元比创建*一个用于每个 UI 控件类的单元*可能会更好，因为后一种方法可能使平常的"uses"分句长(尽管一个典型的 UI 代码将依赖于一组 UI 类的几个)。但是，放置所有的这些 UI 类在一个单个的 myunit.pas 文件中可能使它成为一个长文件，难于找到正确的方法，因此，分裂它到多个 include（包含）文件能够可以理解。
2. 它允许有一个带有平台独立容易实现（implementation）的跨平台单元接口（interface）。基本上你可以做

```
{$ifdef UNIX} {$I my_unix_implementation.inc} {$endif}
{$ifdef MSWINDOWS} {$I my_windows_implementation.inc} {$endif}
```

有时，比写一个带有很多{$ifdef UNIX}的长代码更好，{$ifdef MSWINDOWS}与一般的代码(变量声明，程序（routine）implementation（实施）)混合。这种方式代码更可读。 你甚至可以更积极地使用这个技巧，通过使用 FPC 的命令行选项-Fi 来包含一些仅对具体指定平台的子目录。然后你可以有很多 include（包含）文件{$I my platform_specific_implementation.inc}的版本，你简单的 include（包含）它们，让编译器找到正确的版本。

7.6.记录

Record 只是其它变量的一个容器。它像一个大量简化的*类*: 没有继承或虚拟方法。它像在类 C 语言中的一个*结构* 。

如果你使用{$modeswitch advancedrecords}指令，记录**能**有方法和可视化说明符（specifiers）。通常，可获得的类和*不打破一个记录的简单可预测的存储器布局*的语言特征是可能的。

```
{$mode objfpc}{$H+}{$J-}
{$modeswitch advancedrecords}
type
  TMyRecord = record
  public
    I, Square: Integer;
    procedure WriteInDescription;
  end;

procedure TMyRecord.WriteInDescription;
begin
  WriteIn('Square of ', I, ' is ', Square);
end;

var
  A: array [0..9] of TMyRecord;
  R: TMyRecord;
  I: Integer;
begin
  for I := 0 to 9 do
```

```
  begin
    A[I].I := I;
    A[I].Square := I * I;
  end;

  for R in A do
    R.WriteInDescription;
end.
```

在 modern Object Pascal 中，你的第一直觉应该是来设计一个类，而不是一个记录—因为类有大量的特征，像构造函数和继承。

但是，当你需要速度或可预见的存储器布局时，记录仍然非常有用：

- 记录没有一些构造函数或解析函数。你仅定义一个记录类型的一个变量。它在开始处(除自动管理类型外，像字符串；它们被保证为初始化为空，并且最终来释放参考（reference）计数)有未定义内容（存储器垃圾)。所以，当处理记录时，你不得不更小心，但是它给予你一些性能增加。
- 在存储器中，记录的数组是令人满意地线性的，所以，它们是缓存友好的。
- 在一些情况中，记录(大小，字段之间的填充)的存储器布局被清晰地定义：当你要求 *C 布局*，或当你使用 packed（包装）记录时。这是有用的：
    - 与用其它编程语言写的库通信，当它们阐述（expose）一个基于记录的 API 时，
    - 读和写二进制文件，
    - 做令人厌恶的（dirty）低级技巧(像不安全的定型（typecasting）一个类型到另一个，意识到它们的存储器表现形式)。
- 记录也可以有 case 部分，这像联合作业在 C 类语言中。它们允许来对待相同的存储器块作为一个不同的类型，依赖于你的需要。像这样，在一些情况下，这给予更高的存储器效率。并且它给予 *令人厌恶的（dirty）*，低级不安全技巧：）

7.7. 旧样式对象

在过去，Turbo Pascal 引入其它语法 ，像使用 object 关键字的类功能。它有点介于一个记录和一个现代类之间概念的融合。

- 旧样式对象可以被分配/释放，并且在这操作期间，你可以调用它们的构造函数/析构函数。
- 但是，它们也可以被简单地声明和使用，像记录（records）。一个简单的记录或对象类型不是一个到其它事情的引用(指针)，它不过是数据。这使它们对小数据舒适，在这里调用分配/释放可能会引起麻烦的。
- 旧样式对象提供继承和虚拟(virtual)方法，不过与来自现代的类相比有小的不同.注意—如果你尝试来使用一个对象,而不调用它的构造函数,并且该对象有虚拟(virtual)方法,*不好的事情 things* 将会发生.

在大多数情况下，反对使用旧样式对象。现代的*类*p 提供更多的功能。并且当需要时，记录(包括*高级记录*)能被用于执行（performance）.这些概念通常比旧样式对象更好

7.8. 指针

你可以创建一个*指针* 到任意类型. 指针到类型 TMyRecord 被声明为^TMyRecord,并按照惯例被称为 PMyRecord. 这是一个使用记录的整数链的一个传统示例:

```
type
  PMyRecord = ^TMyRecord;
  TMyRecord = record
    Value: Integer;
    Next: PMyRecord;
  end;
```

注意，这定义是递归的(recursive)(类型 PMyRecord 是使用类型 TMyRecord 定义的， 而 TMyRecord 是使用类型 PMyRecord 定义的).只要它在相同的类型语句块中将被解决，定义一个指针类型到一个*尚未定义的类型*是被允许的.

你可以使用 New / Dispose 方法来分配和释放指针,或者使用(更低级，非类型安全的)GetMem / FreeMem 方法. 你间接引用指针(来访问*指向的*原材料(stuff)，通过)你追加的^操作符.为执行逆运算，来*获取一个存在值的一个指针*,你使用@运算符前缀它.

也有一个未类型化的(untyped)指针类型,类似于在 C 类语言中的 void*.它是完全不安全的，可能被类型化(typecasted)到任意其它指针类型.

切记，事实上，一个*类实例*也是一个指针,尽管它不需要一些 ^ 或 @ 运算符来使用它. 一个使用类的链接表当然是合理的，它可以简单地是这样:

```
type
  TMyClass = class
    Value: Integer;
    Next: TMyClass;
  end;
```

7.9.运算符(Operator)重载

你可以重载很多语言运算符的意义。例如，来给予你自定义类型的加法和乘法。像这样:

```
{$mode objfpc}{$H+}{$J-}
uses StrUtils;

operator* (const S: string; const A: Integer): string;
begin
  Result := DupeString(S, A);
end;

begin
  Writeln('bla' * 10);
end.
```

你也可以重载关于类的运算符.因为你通常在运算符函数中创建你的类的新的实例,调用者必需记着释放结果.

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils;

type
  TMyClass = class
    MyInt: Integer;
  end;
```

```
operator* (const C1, C2: TMyClass): TMyClass;
begin
  Result := TMyClass.Create;
  Result.MyInt := C1.MyInt * C2.MyInt;
end;

var
  C1, C2: TMyClass;
begin
  C1 := TMyClass.Create;
  try
    C1.MyInt := 12;
    C2 := C1 * C1;
    try
      Writeln('12 * 12 = ', C2.MyInt);
    finally FreeAndNil(C2) end;
  finally FreeAndNil(C1) end;
end.
```

你也可以重载关于记录的运算符. 对于类来说,这通常比重载类它们更容易, 因为调用者不必须处理那时的存储器管理器.

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils;

type
  TMyRecord = record
    MyInt: Integer;
  end;

operator* (const C1, C2: TMyRecord): TMyRecord;
begin
  Result.MyInt := C1.MyInt * C2.MyInt;
end;

var
  R1, R2: TMyRecord;
begin
  R1.MyInt := 12;
  R2 := R1 * R1;
  Writeln('12 * 12 = ', R2.MyInt);
end.
```

对于记录,建议在记录内部中使用{$modeswitch advancedrecords} 和重载运算符作为类运算符. 这允许使用泛型(generic)类 ,依赖于带有这些记录的一些运算符的存在(像 TFPGList, 依赖于可用的相等的运算符).否则,一个运算符(operator) (而不是在记录中)的"全局"定义可能不被找到(因为在实施(implements)TFPGList 的代码处不是可找到的), 并且你不能专门

研究一个列表,像 specialize TFPGList<TMyRecord>.

```
{$mode objfpc}{$H+}{$J-}
{$modeswitch advancedrecords}
uses SysUtils, FGL;

type
  TMyRecord = record
    MyInt: Integer;
    class operator+ (const C1, C2: TMyRecord): TMyRecord;
    class operator= (const C1, C2: TMyRecord): boolean;
  end;

class operator TMyRecord.+ (const C1, C2: TMyRecord): TMyRecord;
begin
  Result.MyInt := C1.MyInt + C2.MyInt;
end;

class operator TMyRecord.= (const C1, C2: TMyRecord): boolean;
begin
  Result := C1.MyInt = C2.MyInt;
end;

type
  TMyRecordList = specialize TFPGList<TMyRecord>;

var
  R, ListItem: TMyRecord;
  L: TMyRecordList;
begin
  L := TMyRecordList.Create;
  try
    R.MyInt := 1;   L.Add(R);
    R.MyInt := 10;  L.Add(R);
    R.MyInt := 100; L.Add(R);

    R.MyInt := 0;
    for ListItem in L do
      R := ListItem + R;

    Writeln('1 + 10 + 100 = ', R.MyInt);
  finally FreeAndNil(L) end;
end.
```

山东世联环保科技开发有限公司

CNWUT Shandong World United Environmental Protection Technology Development Co.,Ltd.

世联环保官方网站：http://www.cnwut.com

感谢 **山东世联环保科技开发有限公司** 资助 Lazarus 书籍的中文化翻译。

希望大家帮助与支持 **山东世联环保科技开发有限公司** 的发展，

为 **世联环保** 提供业务信息，进而支持 Lazarus 中文化发展！