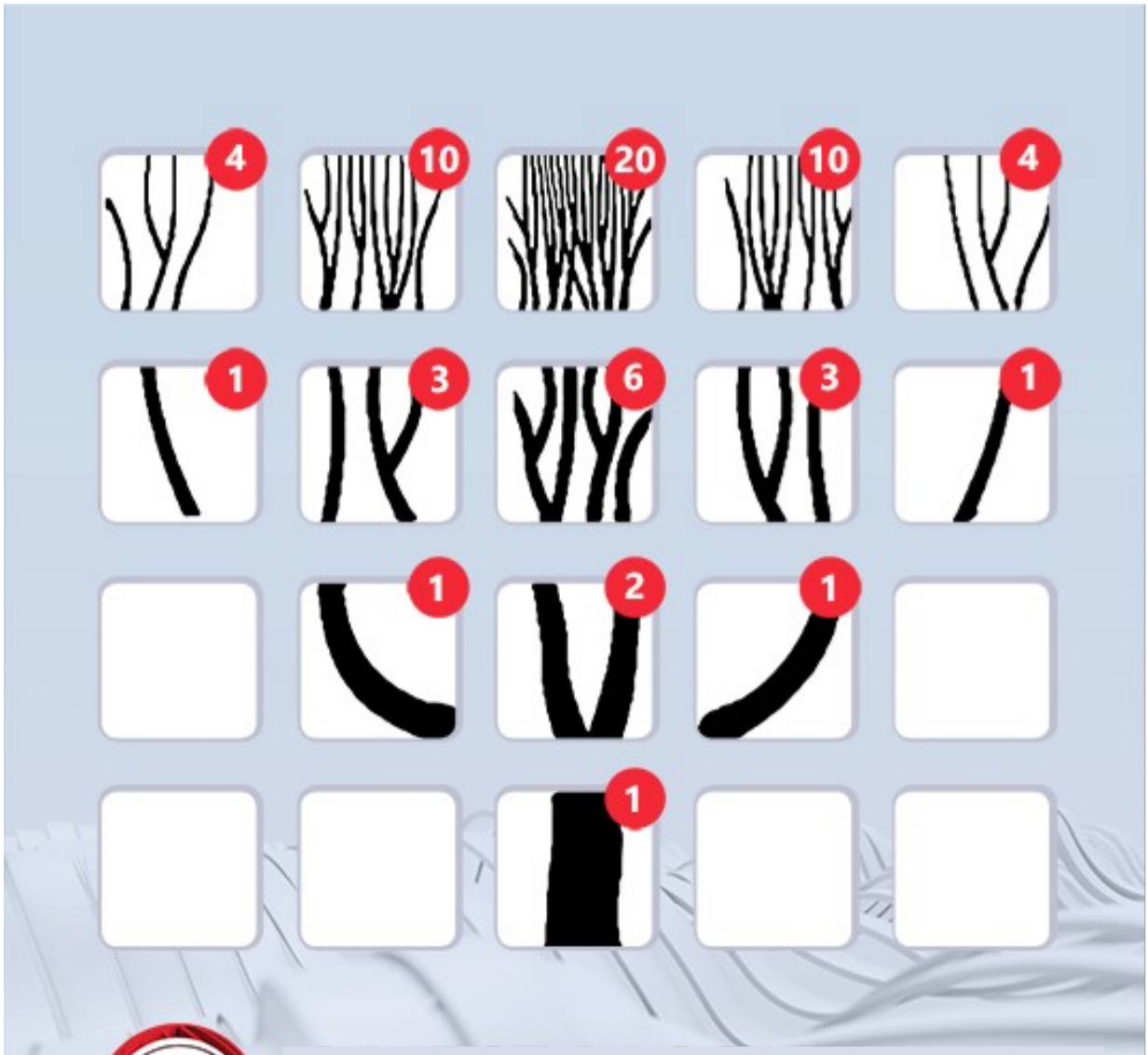


Object Pascal 编程语言手册

Delphi 11 Alexandria Edition

by Marco Cantú - November 2021

译者: 张子仁 – 2022 年一月

Marco Cantù

Object Pascal Handbook Delphi 11 Alexandria Edition

The Complete Guide to the Object Pascal

programming language for Delphi 11 Alexandria

Original edition: Piacenza (Italy), July 2015

Delphi 10.4 Edition: Piacenza (Italy), March 2021

Delphi 11 Alexandria Edition: Piacenza (Italy), November 2021

Author: Marco Cantù
Publisher: Marco Cantù
Editors: Peter W A Wood (first edition), Andreas Toth (second edition) Cover
Designer: Fabrizio Schiavi (www.fsd.it)

Object Pascal Handbook: Delphi 11 Alexandria Edition

This PDF-only version has been published on November, 2021

The last printed edition for Delphi 10.4 Sydney is ISBN-13: 9798554519963

The electronic edition of this book has been licensed to Embarcadero Technologies Inc. It's also sold directly by the author. *Do not distribute the PDF version of this book without permission from the author*. The printed edition is published through Kindle Direct Publishing and sold at several online outlets.

More information at http://www.marcocantu.com/objectpascal

begin

功能强大且简单、容易表达又具可读性、初学者跟专业人员都能快速熟悉，这些就是今日 Object Pascal 的部份特点。

Object Pascal 历久弥新，不断向前演进，而演进至今，更具备了多种不同面向，它结合了面向对象编程语言的强大、泛型程序的高阶功能，以及动态结构特性 (anonymous function 等结构化的程序区块)，却也没有舍弃许多程序化语言的传统特性。Object Pascal 适合所有行业，是一个在行动时代具备编译功能的开发工具，更是一个具备坚强历史，已经准备好面对未来挑战的编程语言。

Object Pascal 是为了开发什么特定功能而存在的？从单机桌面到主从式架构应用程序，从大量处理数据的 web 模块到中间件，从办公室自动化到最新的行动装置(手机与平板 app)的开发，从工厂自动化系统到网际/移动电话虚拟网络等等，这些都不是这个语言被设计准备来处理的领域，而是在今日世界中，Object Pascal 这个语言已经实际在服务的领域，这就是 Object Pascal 强大的地方，它不是预期可以做这些事，而是已经做到了。

我们今日使用的 Object Pascal 编程语言的核心，都是源自 1995 年，那个编程语言发展的黄金年代的定义，在同一个年代，JAVA 跟 JAVAScript 也刚被开发出来。虽然 Object Pascal 语言的根基是在那么久以前就已经奠定的，而且还有更久以前的 Pascal 语言，这个编程语言的发展并没有在 1995 年停下脚步，直至今日，它的核心功能仍旧不断的在增强当中，透过 Embarcadero 创造的桌机版与行动版的编译器，使用在 Delphi 跟 RAD Studio 的开发工具当中。

一本介绍今日编程语言的书

随着编程语言的角色不断变化，功能的延伸也逐年在发生，最终的反映就是吸引更多新的开发人员。我觉得写一本能够完整介绍今日的 Object Pascal 所涵盖的各个领域是很重要的。这个目标是为所有人提供一本手册，为新进的开发人员能够有编程语言的指导书，让对其他相似语言熟悉的开发人员也能快速入门，也让在过去的这些年里，对熟悉不同版本 Pascal 的开发人员能快速的知道目前 Object Pascal 究竟有了什么发展。

新进的开发人员当然需要一些基础，但随着功能的增加，即使是熟悉过去不同版本 Object Pascal 的开发人员，一定也能从本书的章节中得到一些新的信息。

我们会以一个篇幅不长的附录来简单介绍 Object Pascal 语言的历史，其余的篇幅则会着重在今日的 Object Pascal。从早期的 Delphi 开始，大多数的核心功能并没有很显著的变化，绝大多数很重要的核心功能，早在 1995 年第一版的 Delphi 当中就已经奠定了基础。

在我即将开始介绍的本书的内容当中，这个编程语言在这么多年的存在过程中，并不是停滞不前的，它在过去的这段期间中，以相当快的速度在进化着。

在我过去其他的作品当中，我的介绍手法比较传统，首先会介绍最传统的 Pascal 编程语言，接着或多或少介绍一些在发展过程中被加入的延伸功能。但在这本书里，我会直接切中主题，直接介绍今日我们怎么透过 Object Pascal 解决问题，或者怎么用它最好，而不会介绍它一路走来是怎么演进的了。

举例来说，最近被提出的原生的数据型别具备面向对象特性中 method 的功能，可溯及原始 Pascal 语言。这部份我在第二章里面就会直接介绍怎么使用这个功能，而不会试图先让读者们理解这当中的设计运作原理。

换句话说，这本书会着重在让读者们怎么在今时今日使用 Object Pascal，从基础引导、让读者们从做中学，只会提到最低限度的历史发展。即使读者们已经很熟悉这个语言了，应该还是会希望直接切入主题，至于历史的发展与脉络，我们只在最后一章介绍。

从做中学

这本书的意旨，是要介绍核心概念，并透过很短的范例让读者立即进入状况，透过这些范例，读者们可以试着执行、练习、并且自行摸索了解到概念，并对这些概念有更深切的体认。

这本书不会是一本参考手册，参考手册会解释这个编程语言在理论上该怎么做，并列举所有可能的案例作为说明。而为了精确的介绍概念，本书会

透过一步步的实战练习，用范例来让读者学会 Object Pascal 这个语言。这些范例都会很简单，因为我们的目标，是让读者一次专注在一个主题上面。

完整的范例源码都放在 GitHub 的链接库里面，您可以下载某一个特定的档案，或者复制整个链接库，或者直接在线浏览，并下载特定项目的某些源码，只要您能在线存取这个链接库，万一我对这些范例做了更新，或者新增了一些范例，您就可以很方便的随时取用或更新您的源码了。GitHub 的链接库跟前一版电子书所使用的网址是一样的：

```
http://github.com/MarcoDelphiBooks/ObjectPascalHandbook104
```

要编译或者测试这些范例程序，您会需要能够使用的 Object Pascal 编译器(至少需要 10.4 版才能让所有功能都正常运作，但大多数的范例在 10.x 跟 11 版也都能够正常运作)。

如果您还没有 Delphi 的授权，Embarcadero 有提供试用版的 Delphi，通常都会有 30 天的免费试用期，让您可以使用编译器与整合开发环境。或者您也可以申请免费的 Delphi 社群版(目前已经更新到 10.4 版了)，社群版是提供给所有人使用的，用途可以是商用或者个人使用，当中只有很低的限制，好让您可以快速进入软件开发的工作。

致谢

每一本书的完成，总是要感谢很多人，我要感谢的人可以列成一个很长很长的名单，首先，是为本书付出许多心血的编辑，Peter Wood。由于一再修改出书的时程，以及帮我使用的许许多多的科技用语顺成一般读者可以读的懂的语汇，感谢 Peter，协助我成就了本书。

从第一版推出之后，身为读者也是开发人员的 Andreas Toth，传给我对于前版的一些延伸的想法。后来在第二版之后，他也加入了编辑的行列，协助我对于内容的说明、英文文法的调整、整本书的延续性，以及 Object Pascal 程序风格上都提供了易于读者理解的修改，新的这一版很多地方都仰仗他的努力。

新版的内容也已经请一些 Delphi 的专家(大多数都已经是 Embarcadero MVP)，尤其是 François Piette，在最后一版修正之前，他提供了超过一百个修正跟建议给我。

由于我目前担任 Emarcadero 的产品经理，我也仰仗所有跟我一起工作的同仁，以及研发团队的成员，更要感谢在这段期间当中，所有曾经跟我谈话、开会或透过 Email 进行讨论的同仁们，让我能够对这个产品，以及当中的技术有更深层的了解。

要一一提到每位帮过我的人有点难，我不会真的一一回想，但如果只要选出对这本书第一版有莫大帮助的的人，我会说是:

负责开发人员维系的 David I、带领 RAD 产品管理的 John Thomas(JT)，以及 RAD 架构师 Allen Bauer.

最近我跟几位同事也非常密切，他们分别是 RAD Studio 的产品经理(Sarina DuPont 跟 David Millington)，我们现任的 Delphi 传道士 Jim McKeeth，以及我们非常杰出的研发团队的架构师们。

在 Embarcadero 之外的伙伴们也持续的提供直接的联系跟建议，从许多意大利的 Delphi 专家到数不清的使用者、Embarcadero 的销售跟技术伙伴、Delphi 社群的成员们、MVP 们、跟即使使用的是其他编程语言或开发工具的开发人员，都常会提供给我想法。

在整个团队中，跟我一起工作时间最久的，当数 Cary Jensen 了，在我加入 Embarcadero 之后，他和我一起在欧洲跟美国之间策划、建构了好几次 Delphi 开发者大会。

最后，我要深深的感谢我的家人们，因为我常变动的旅程、在夜间的会议，以及许多个用在写书的周末时间，谢谢 Lella, Benedetta 跟 Jacopo.

关于我自己 (Marco)

在过去的 25 年当中，花了大部分的时间来写作、教学，以及提供 Object Pascal 这个编程语言开发的软件顾问工作上。我过去曾写过 Master Delphi 系列的畅销作品，后来则是自己发行了几本关于开发工具的工具书，横跨了 Delphi 2007 到 Delphi XE 这几个不同的版本。在几个大洲的许多程序开发座谈会上，我都担任过主讲人，并协助数以百计的程序开发人员。

在从事了多年的独立软件顾问工作以及训练者之后，在 2013 年，我的职业生涯有了很突然的变化，我接受了 Embarcadero 的邀请，担任 Delphi 及 RAD

Studio 的产品经理。Embarcadero 开发、销售这些很棒的开发工具，同时还涵盖了最新的 Appmethod 产品。

为了不再多占篇幅，我只再多说一点，我目前定居在意大利，通勤到美国加州上班，有爱我的太太跟两个很棒的小孩，而且会尽我所能的享受回归程序设计的乐趣。

希望你阅读本书的时候能够乐在其中，就像我写这版新书的时候一样，如果你需要更多信息，可以透过以下任何一个方式跟我联系:
http://www.marcocantu.com/objectpascalhandbook
http://blog.marcocantu.com
http://twitter.com/marcocantu

Marco Cantù, Object Pascal 手册

{.译者的话}

这是译者的第五本书，和第四本书『Delphi/Kylix Indy 网络程序设计』间隔了十四年。写书从来是需要被鞭策的工作，也是极需要耐心的一项工作。在这十四年之间，译者的工作一直跟研发相关，从 Windows 应用程序到 iOS 应用程序，虽然编程语言不断的成长，我们身为开发人员也用生命陪着编程语言一起成长，其中的甘苦不足为外人道。

从 1997 年开始用 Delphi 来写论文，到 1999 年受学弟陈建良的影响，指导他写出了到目前都还很畅销的雷电 MAILD，到 2000 年在服役的时候写出了 Indy 当中的 TIdDNSServer，再商业化成雷电 DNSD。

在 2004 到 2007 之间专注在 Windows Media 播放程序、网络程序以及网络备份程序上，直到 2009 年转移到 iOS 的 Objective-C 领域，Delphi 跟 Object Pascal 一直是我念兹在兹的编程语言。

我常在许多场合被问到『Delphi 现在还有人用吗?』『那不是早就过气的语言吗?』这样的问题在 2010 年以前，听来相当尴尬，但在那段时间当中，我知道 Delphi 跟 C++ Builder 的使用者仍然很多，因为其他工具对数据库整合的程度不够好，使得 Delphi 跟 C++ Builder 成为硬件工具机台、生产机台跟数据库之间整合最好的工具，至今仍然少有能出其右者，从新竹科学园区使用这两个工具的公司仍占绝大多数的现象就能知道。

虽然我自己在 2009 年转为以 Objective-C 来撰写 iOS 程序，但 Objective-C 的概念跟许多地方和 Object Pascal 真的极为相似，所以我想忘也忘不掉，只是随着 Embarcadero 对这两个工具的用心加强，在 2012 至今，我还是能够用它来开发许多跨行动装置的程序与小工具。

在学界的教学当中，从 1996 到 2015 之间的 15 年，Java 可以说是完全制霸，但我正好最不想碰 Java，从 1995 年到 2015 年之间，我曾熟悉 Perl, PHP, Object Pascal, 甚至 C++跟 Python 都拿来开发过许多项目，唯独 Java 是我能不碰就不碰的，这纯粹是赌气。

在这 15 年当中，学界的主流是 Java，但在这 15 年当中，随着我在元智大学、长庚大学、中台科技大学(感谢李桂春老师的引介)等学校的兼任、演讲，让我保持与信息教学不间断的接触，也让我深感到这 15 年当中台湾在软件开

发的成长极为缓慢，甚至比 15 年前更为落后，同学们的热情不再，当然每一届都有很优秀的同学，只是比例越来越低，也让我在教学上的热忱逐年降低，都快熄灭了。不过我仍放不下 Delphi 的开发，所以从 2014 年开始，我时不时的把一些开发的心得写成部落格文章，跟大家分享，我的部落格网址是: http://firemonkeylessons.blogspot.tw/

在这 20 年左右的从业生涯中，我熟习的概念不少，最常接触、称得上精通的有两大部分，一个是面向对象程序的概念，另一个则是网络协议的订定与制作。对面向对象程序概念的熟习要感谢陈恭老师，陈恭老师目前(2015 年)仍任教于政治大学信息科学系，我在就读大同工学院(现改为大同大学)时，修习陈恭老师开设的『高等面向对象程序设计』，以及『编程语言结构』等课程，在课程中对面向对象的概念奠下完整的基础，当时并没有 Java 编译器，但唯其不透过编译器的协助，锻炼出来的对象概念更为深刻。

对研发的执着，要深深感谢鲁立忠老师的教导，鲁立忠老师目前服务于台积电，两位老师对于我在大学时代奠定对研发的基本功都有深切的影响。

另外也要感谢捷康科技跟 Embarcadero，让我不断有机会对 Delphi 跟 Object Pascal 有深入研究的机会。这几年我以 Delphi 开发的程序从 Windows 到 iOS 到 Android 都有，从 2007 年 Hinet SafeBox, 2009 年 FalconStor 的 FileSafe Express 到 2014 年 SGS 的云端质量查验系统，从数据库到 iPad 各项功能整合，从文件系统到网络协议，再到 2015 年以 iBeacon 为主体的物联网监控系统，都是以 Delphi 为主体进行开发的，涉猎范围不可谓不广，然而在台湾的软件开发成不了一个产业，这是事实，也是悲哀，期许哪一天在台湾写软件可以像在硅谷那样受到重视。

感谢捷康科技方总经理，让我有机会翻译我心目中偶像-Marco Cantù 的书，我在 2006 年间从 Macro 的 Mastering Delphi 2005, Mastering Delphi 2006 启发许多，早希望能有机会翻译 Marco 的著作，感谢方总让我圆梦。

最后，仅以此译作献给我的家人们，尤其这是我成家之后的第一本书，目前更新已经是第四个版本了，每一次在译作的期间，太太辛苦的帮我多分担了照顾女儿们的责任，感谢如馨，也希望两个女儿长大后，有机会看到这本书，但不要受到影响，以你们的喜好选择你们长大后的工作，颐荷，颐芊，愿你们平安长大，平安喜乐。

张子仁 2021 于台北

目录

begin ____ 3

一本介绍今日编程语言的书 ____ 3

从做中学 ____ 4

致谢 ____ 5

关于我自己 (Marco) ____ 6

{.译者的话} ____ 8

目录 ____ 10

第一部: 基础篇 ____ 22

第一部的章节列表: ____ 22

01:用 Pascal 写程序 ____ 23

我们开始来看源码吧 ____ 23

第一个文本模式的应用程序 ____ 23

第一个图形接口程序 ____ 25

语法和源码样式 ____ 28

程序批注 ____ 28

批注和 XML 文件 ____ 30

识别符号（Symbolic Identifiers） ____ 32

大小写视为相同与使用大写字母 ____ 33

使用空格符 ____ 34

源码内缩 ____ 35

强化语法标示 ____ 37

错误检知和源码检知 ____ 37

编程语言的关键词 ____ 38

程序结构 ____ 42

单元与程序名称 ____ 43

用.来为单元命名 ____ 44

更多关于单元文件的结构 ____ 45

Uses 条文 ____ 46

单元与界限 ____ 47

把单元文件当成命名空间来使用 ____ 48

程序档案 49

编译器设定 50

条件化定义(Conditional Defines) 50

编译器版本(Compiler Versions) 51

引入檔 (Include Files) 53

02:变量与数据型别 54

变量与指派数据(Assignments) 55

实际值（文字值、字面意义：Literal Value） 56

指派叙述句（Assignment Statements） 57

指派（Assignment）与转换（Conversion） 58

为全局变量进行初始化 58

为局部变量进行初始化 59

行内变量 (在源码中宣告的变量 Inline Variable) 59

为行内变数初始化 60

行内变数的型别推定 60

常数（常量，Constants) 61

行内常数 (在源码中宣告常数、常量 Inline Constants) 62

资源字符串常数（Resource String Constants） 63

变量的生命周期(Lifetime)与可视范围(Visibility) 63

资料型别 65

有序与数值类型别 65

整数类型型别的别名 66

整数型别, 64 位与 NativeInt 67

整数型别助手 67

标准有序型别函式 69

超过范围的运算 70

布尔值 71

字符 72

字符型别的运算 72

把 Char 视为有序型别 73

以 Chr 函式进行转换 74

32 位的字符 74

浮点数型别 75

为何浮点数的数值特别不同 76

浮点数类别助手与 Math 单元 77

简单的使用者自定型别 77

命名与非命名型别 78

型别别名 79
次范围型别 80
列举型别 81
范围列举 82
集合型别 83
集合型别的运算方法 84
表达式和运算方法 84
使用运算方法 85
显示表达式的结果 85
运算方法与其优先性 86
日期与时间 89
日期时间助手(DateTime Helper) 92
型别切换(Typecasting)与转型(Type conversions) 92
03:语言叙述句 95
简单与复合叙述句 95
IF 叙述句 96
Case 叙述句 98
For 循环 100
For-in 循环 104
While 和 Repeat 循环 105
循环的范例源码 107
用 Break 和 Continue 指令来中断流程 109
要介绍 Goto 指令了吗?绝不! 110
04:程序与函式 112
程序与函式 112
预先宣告 115
递归函数 116
方法(Method)是什么? 118
参数与回传值 118
附带回传值离开 119
引用参数(Reference Parameters) 121
常数参数 (Constant Parameters) 123
函式的多载 (Function Overloading) 124
多载与呼叫混淆 (Ambiguous call) 126
预设参数 (Default Parameters) 128

内嵌源码 (Inlining) 129
函式的进阶功能 133
Object Pascal 呼叫函式的约定(Conventions) 133
程序型别 (Procedural Types) 134
宣告外部函式 136

05:**数组与记录** 139
数组数据型别 139
静态数组 (Static Arrays) 139
数组的大小跟边界 141
多维度静态数组 142
动态数组 143
开放数组参数 148
记录数据型别 152
使用记录数组 155
变异记录 (Variant Records) 156
字段对齐 (Fields Alignments) 156
With 叙述句是什么? 158
带有方法的纪录 160
Self: 记录神奇的地方 163
为记录初始化 164
记录与建构者(Constructors) 164
运算符号的新纪元 (Operators Gain New Ground) 165
运算符号与自定受管理的记录 171
变动型别(Variants) 175
变动型别没有型别(Variants Have no Type) 176
深入探讨变动型别(Variants in Depth) 178
变异型别很慢(Variants Are Slow) 179
指标的两三事(What About Pointers) 180
档案型别，还有谁有提供?(File Types, Anyone?) 184

06:**关于字符串** 185
Unicode:**全世界的字母** 185
旧的文字编码:从 ASCII 到 ISO 编码 186
Unicode 字码跟字形 187
从字码到字节 (UTF) 188
位序列记号 (BOM) 189

看清楚 Unicode 190
再度介绍字符型别 193
使用 Character 单元来处理 Unicode 193
Unicode 字符常数 (Unicode Character Literals) 195
那单位元的字符呢? 197
字符串数据类型 (String Data Type) 198
把字符串作为参数传递 201
[]的使用，以及字符串中对字符计数的模式 203
字符串连接 205
字符串助手的处理程序 206
更多字符串的运行时间函式库(RTL) 210
格式化字符串 211
字符串的内部结构 213
观察在内存里面的字符串 215
字符串与编码 217
其他的字符串型别 220
UCS4String 型别 221
旧版的字符串型别 222
第二部: Object Pascal **中的** OOP 223
07:**物件** 225
介绍类别(Class)**与对象**(Object) 225
定义一个类别 226
在其他 OOP 语言的类别 227
类别方法 228
建立对象 229
对象参考模型(Object Reference Model) 230
释放对象 231
什么是"Nil" 232
在内存里面的记录与类别 233
私有、保护、公开 234
私有区数据字段的范例 235
封装与窗体 238
Self **关键词** 240
动态建立组件 241
建构函式 243

以建构函式跟解构函式来管理区域类别数据 245
多载方法以及建构函式 247
完整的 TDate 类别 249
嵌套类型与巢状常数 252
08:继承 255
从既存的型别中继承 255
常用的基础类别 258
保护区字段与封装 259
使用保护区来骇入 (Protected Hack) 259
从继承到多型 262
继承与型别兼容性 262
延迟绑定与多型 264
覆写(Override)、重新定义(Redefine)、重新介绍(Reintroduce)方法 266
继承和建构函式 269
虚拟与动态方法 270
把方法跟类别抽象化 271
抽象方法(Abstract Methods) 272
弥封类别(Sealed Classes)跟最终方法(Final Methods) 274
安全的型别转换指令 275
可视化窗体继承 277
从基础窗体进行继承 278
09:例外处理 282
Try-Except 区块 283
触发例外 288
例外与堆栈 289
Finally 区块 290
Finally 跟例外 291
透过 Finally 区块还原光标 292
透过受管理的纪录还原光标 293
真实世界的例外 294
全局例外处理 294
例外与建构函式 296
例外的进阶功能 298
巢状例外与内部例外机制 299

拦截例外 302

10:属性与事件 305

定义属性 305
和其他编程语言的属性相比较 307
属性实现了封装概念 308
在宣告属性时使用代码自动完成的功能 309
为窗体加入属性 310
为 TDate 类别加入属性 312
使用数组属性 314
以参考设定属性 316

发布(Published)存取关键词 317
设计时间属性 318
发布(published)与窗体 319
自动 RTTI 320

事件驱动程序作法 321
方法指标(Method Pointers) 323
委任的原理 325

建立 TDate 组件 331

在类别中实作对于列举功能的支持 333

结合 RAD 开发环境跟 OOP 的 15 个小提示 337
提示一:窗体是个类别 337
提示二:为组件命名 338
提示三:为事件命名 338
提示四:使用窗体的方法 338
提示五:建立窗体建构函式 339
提示六:避免使用全局变量 339
提示七:绝不要在类别实作的源码中使用特定实例变量名 339
提示八:尽量少用 Form 变量名 340
提示九:去除全局的 Form1 变数 340
提示十:加入窗体属性 340
提示十一:把组件属性披露 340
提示十二:适时的在需要使用数组属性 341
提示十三:使用属性的副作用 341
提示十四:隐藏组件 342
提示十五:使用 OOP 窗体精灵 342
提示结论 343

11:接口 344
使用接口 345
宣告接口 345
实作接口 346
接口与参考计数 348
混和参考的错误 350
弱化(weak)与不安全(unsafe)的接口参考 352
进阶接口科技 354
接口的属性 354
接口委任 356
多重接口以及方法别名 358
接口的多型 359
从接口参考解析出对象 361
透过接口来实作转换器模式(Adapter Pattern) 363
12:操作类别 366
类别方法跟类别数据 366
类别数据 367
类别静态方法 368
静态类别方法跟 Windows API callback 369
类别属性 371
具有实体个数计数器的类别 371
类别建构函式(以及解构函式) 373
在 RTL 里的类别建构函式 374
实作单一实体模式(Singleton Pattern) 375
类别参考 376
在 RTL 里面的类别参考 378
用类别参考来建立组件 378
类别与记录助手 381
类别助手 382
ListBox 的类别助手 384
类别助手跟继承 385
使用类别助手加入对控件列举的功能 385
现有型别的记录助手 389
型别别名的助手 391
13:对象与内存 393

全局数据、堆栈以及 Heap 394
全局内存 394
堆栈 395
Heap (堆) 395
对象参考模型 396
把对象当成参数来传递 397
内存管理小技巧 398
释放我们建立的对象 399
只能把对象释放一次 400
内存管理与接口 402
更深入 Weak 参考 403
Weak 参考是受到管理的 406
不安全的标注 (The unsafe Attribute) 407
追踪与检测内存 407
内存状态 408
FastMM4 408
追踪泄漏以及其他全局设定 409
在完整版 FastMM4 上的缓冲区溢出 411
在 Windows 以外的平台的内存管理 414
追踪每个类别的配置 414
撰写强健的应用程序 414
建构函式, 解构函式, 以及例外 415
巢状 Finally 区块 417
动态型别检查 418
这个指针是对象参考吗? 419
第三部: 进阶功能 423
14:泛型 424
通用的键-值对 (Key-Value Pairs) 424
行内变数与泛型型别推定 428
泛型的型别规则 428
Object Pascal **里面的泛型** 430
泛型型别兼容性规则 431
标准类别的泛型方法 432
泛型型别实体化 434
泛型型别函式 436
泛型类别的类别建构函式 439

泛型守则 (Generic Constraints) 441
类别守则 (Class Constraints) 441
特定的类别守则 444
接口守则 444
接口参考 v.s.泛型接口守则 447
预设建构函式守则 448
泛型守则的总整理以及组合应用 450

预先定义的泛型容器 451
对 TList<T>进行排序 453
以匿名方法进行排序 455
对象容器(Object Containers) 457
使用泛型字典(Dictionary) 457
Dictionary v.s. 字符串列表 462

泛型接口 464
预先定义的泛型接口 466

在 Object Pascal 里面的智能指标(Smart Pointer) 467
使用智能指针的记录 468
用泛型受管理的纪录来实现智能指针 469
用泛型记录与接口来实现智能指针 471
加入隐晦转换(Adding Implicit Conversion) 473
比较几个智能指标的方案 474

以泛型处理协同变异(Covariant)回传型别 475
关于 Animals, Dogs,跟 Cats 475
回传泛型结果的方法 477
回传不同类别的衍生对象 477

15:匿名方法 479

匿名方法的语法和语意 480
匿名方法变量 480
匿名方法的参数 481

使用局部变量 482
展延局部变量的生命周期 482

在背景里的匿名方法 484
(潜在的)漏掉括号 485
匿名方法实作 486
预先准备好的参考型别 487

实战上的匿名函式 488

匿名事件处理程序 489
为匿名方法计时 491
线程的同步 492
Object Pascal 里的 AJAX 495

16:镜射与标注 500

延伸的 RTTI 501
第一个范例 501
编译器产生的信息 502
强型别与弱型别连结 504

RTTI 单元文件 505
在 Rtti 单元文件里面的 RTTI 类别 507
RTTI 对象的生命周期管理以及 TRttiContext 记录 509
显示类别信息 510
套件中的 RTTI 512
TValue 结构 513
以 TValue 读取一个属性 516
呼叫方法 516

使用标注 (Using Attributes) 517
标注是什么？ 518
标注类别与标注宣告 519
浏览标注 521

虚拟方法拦截器 524

RTTI 个案研究 528
在 ID 跟描述上使用标注 528
XML 串流 534
其他以 RTTI 为基础的函式库 542

17:TObject 与 System 单元文件 544

TObject 类别 544
建立与毁灭 545
物件的二三事(Knowing About an Object) 545
更多 TObject 类别的方法 546
TObject 的虚拟方法 548
总结 TObject 类别 553
Unicode 跟类别名称 554

System 单元 555
被选上的系统型别 556

被选上的系统函式 557
预先定义的 RTTI 标注 558

18:其他核心 RTL 的类别 559

Classes 单元 559
在 Classes 单元中的类别 560
TPersistent 类别 561
TComponent 类别 562

现代档案存取 565
输入/输出工具单元 565
介绍串流 567
使用 Reader 跟 Writer 569

建立字符串跟字符串列表 572
TStringBuilder 类别 572
使用字符串列表 573

执行时期函式库是相当巨大的 574

写在最后 577

end. 578

a: Object Pascal 的演进 579

Wirth 的 Pascal 579
Turbo Pascal 580
早期的 Delphi 的 Object Pascal 581
Object Pascal 从 CodeGear 到 Embarcadero 582
朝向行动化 582
Delphi 10.x 时期 583
Delphi 11 发行 583

b: 词汇表 585

第一部: 基础篇

Object Pascal 是一个极为强大的编程语言，奠基在程序结构以及可延伸的数据结构的良好基础之上。这些基础有一部分是从传统的 Pascal 语言衍生而来，但核心的语言功能已经比早期的功能多了许多延伸功能。

在本书的第一部分里，我们会学到关于编程语言的语法，程序风格，以及程序结构，变量与数据型别的使用，基础的编程语言叙述据(像是条件判断与循环)，程序和函式的使用，以及核心型别的建构，例如数组、记录与字符串。

在本书的第二部跟第三部里面，我们会对许多进阶功能的基础，从类别到泛型型别一一介绍。

学习编程语言就像在盖房子，我们必须把基础打好，在不稳固的基础上所盖的任何结构，可能很亮丽，但也很不稳固。

第一部的章节列表:

第一章: 用 Pascal 写程序
第二章: 变量与数据型别
第三章: 语言叙述句
第四章: 程序与函式
第五章: 数组与记录
第六章: 关于字符串

01:用 Pascal 写程序

这个章节会从一些可以用来建立一个 Object Pascal 应用程序的程序片段开始，会涵盖到一些标准的程序写法，程序批注、介绍关键词与完整程序的架构。我会开始写一些简单的程序，试着用透过说明这些源码来介绍并带出接下来几个章节的关键概念。

我们开始来看源码吧

这个章节涵盖了 Object Pascal 语言的基础，但也会花我们几个章节的篇幅来带领读者们理解整个应用程序作业的一些细节。所以，我们先来快速的看两个入门的程序吧(它们的架构会有所不同)，我们不会看的太仔细。目前我只想介绍一下我会用来建立范例程序的架构，接着我们才能介绍其他不同的部分。所以，我希望读者能够尽快开始把书里提到的练习用相关信息取回，从最开始的练习范例开始看，会是个好主意。

笔记 如果您是 Object Pascal 这个语言的初学者，需要逐步操作的指引来协助您使用本书的范例程序，或者让您可以自己开始动手写程序的话，请参阅本书的附录 C.

Object Pascal 一开始就被设计成在透过 IDE 环境中可以实时上手，经由这个语言与 IDE 的坚强组合，Object Pascal 提供了对程序人员最友善的快速开发工具与语言，同时也是执行速度上对机器友善的编程语言。

在 IDE 里面，您可以设计用户接口、协助您撰写源码，执行写好的程序，还有更多的辅助功能。在本书中，就像我会介绍 Object Pascal 这个编程语言，我也会跟您分享我使用 IDE 的方法。

第一个文本模式的应用程序

在一开始，我要透过一个文本模式应用程序，只简单的显示 Hello, Word 这样一个字符串，来介绍 Pascal 程序语法的一些重要的部分。文本模式的应用程序，换句话说就是没有窗口画面的应用程序，执行时会以终端机窗口显示文字，接受用户透过键盘输入，也只以终端机窗口显示结果。文本模

式的应用程序在行动平台上面不常派上用场，但还是会用在 Windows 系统上(最近几年微软花了一些功夫改进 cmd.exe, PowerShell 以及终端机应用)，另外，文本模式应用程序在 Linux 上面也还是很常用到的。

我暂时不会对以下这些源码做太多说明，这也是本书前几章的用途，以下是 HelloConsole 这个程序项目的源码内容：

```
program HelloConsole;
{$APPTYPE CONSOLE}
var
    strMessage: string;
begin
    strMessage := 'Hello, World';
    writeln (strMessage);
    // 以下这个指令，是用来等待使用者输入，直到使用者按下 Enter 键为止
    readln;
end.
```

笔记 在一开始的介绍中，我们已经介绍过，本书所有完整的源码都可以在 GitHub 的档案库里面下载。这些范例的详细介绍则会在本书里面进行，在前文中，我已经提到项目名称(在本范例里面叫做 HelloConsole)，项目名称也会用来当做文件夹的名称，该文件夹里面还有许多跟这个项目相关的档案，由于我会把一个章节的范例放在同一个文件夹里面，所以上面这个项目的文件夹名称会是 01/HelloConsole.

您可以在范例程序第一行看到程序的名字，程序名之后会包含一些指示词(directives)：编译器的设定值(会以$这个符号开头，并且用大括号整个包起来)、变量宣告的区块（一个字符串变量，命名为 strMessage），以及被 begin 跟 end 所包起来的三行源码跟一行说明用的批注。

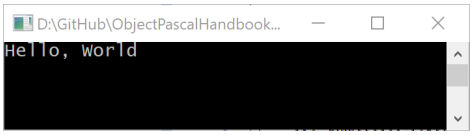
这三行源码会把一段文字复制到一个字符串变量里面去，呼叫一个系统函式来把这个字符串变量里面的内容输出在文本模式窗口里面，并且呼叫另一个系统函式，等待用户输入（在这个范例中，只是用来等待使用者按下 Enter 键）。我们接下来就可以自行定义我们需要的函式，不过 Object Pascal 已经帮我们附上了数百个常用的函式了。

再强调一下，我们很快就会开始介绍这些程序内容，在一开始的章节，我们只是给您一个简单的印象，让您大概知道 Pascal 完整的程序大概长什么

样子而已，当然您也可以直接打开、执行这个程序，程序执行的画面，会像图 1.1 的 DOS 窗口一样，窗口的内容文字如下：

```
Hello, World
```

图 1.1: HelloConsole 范例在 Windows 上面执行的结果：



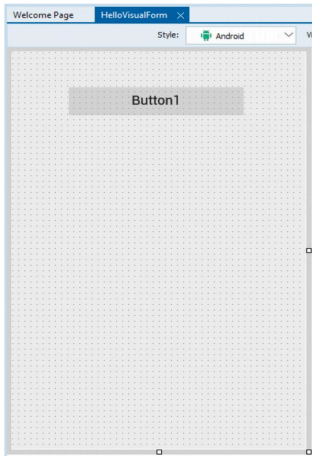
第一个图形接口程序

现代的应用程序，都已经不像上面这个范例，长得像很传统的文本模式窗口，通常会有许多可视化的元素（在Object Pascal里面我们称之为控制组件）显示在窗口画面中。在本书中，我们大部分的范例都会是用FireMonkey组件库(现在简称为FMX了)来制作的图形接口程序（即使大多数案例我都会把它简化到只显示简单的文字）。

笔记 在 Delphi 里面，视觉组件库已经区分成两个不同的类别了：一个是 VCL (专属 Windows 平台上面使用)，以及 FireMonkey（支持多种不同的平台、装置，包含桌面应用程序跟行动装置都支持）。要把这些范例改为 VCL 版本也都非常容易。

要了解一个图形接口程序结构的细节，您必须把本书大多数的篇幅都读过，例如一个form是一个特定类别的对象实体，它包含了许多方法、事件处理程序，以及属性。刚刚这句话所提到的每个部分，在本书中都会介绍到。但要建立一个应用程序，您不需要先成为专家，您只需要透过选单上面的选项，就能轻松的建立一个新的桌面应用程序或行动装置应用程序了。我在本书前面几个章节要介绍的，都会是以FireMonkey的范例为基础（两种IDE都支持），简单的介绍如何透过form的选单跟鼠标点击操作来完成这些动作。一开始，请您先建立任何一种form（桌面或行动应用程序均可，通常我会建立一个Multi-device的空白应用程序项目，这个项目在Windows环境也可以执行的），然后放一个button组件在上头，以及一个多行的文字组件(或者Memo也可以)来显示输出的结果。图1.2就是让您看一下在IDE里面，预设情形下，这个行动应用程序的form会在开发环境里面长什么样子。选择用安卓的样式预览(您可以看到图1.2里面右上角有个下拉选单)，然后新增一个按钮控件到这个form里面。

图 1.2: HelloVisual 范例在 IDE 环境中显示的画面：



您如果想要制作一个类似的应用程序，只需要先建立一个空白的行动应用程序，然后在空白的form上面加入一个button组件即可。现在，我们来加入源码吧。这也是目前我们要接着介绍的，请用鼠标左键双击form画面上的按钮，您就会看到以下的源码在画面上显示出来(也可能是很类似的其他源码)。

```
procedure TForm1.Button1Click (Sender: Tobject) begin
end;
```

即使您都还不知道类别的方法是什么(就是上面这段程序的Button1Click啦)，你也已经可以在上面的源码的begin跟end之间加入一些源码了，这些源码就会在项目执行时，当我们用鼠标左键点选按钮的时候被执行到。

我们的第一个图形接口程序，会有一些源码跟第一个文本模式程序完全相同，只是在图形接口程序里面，我们呼叫了不同函式库里面的不同函式，在这个范例程序中，我们呼叫的是ShowMessage。这个范例的源码，您可以在名为HelloVisual的文件夹里面找到，您可以试着直接编译它，就可以发现执行编译的动作真的是非常的简单：

```
procedure TForm1.Button1Click (Sender: Tobject)
var
    strMessage: string;
begin
    strMessage := 'Hello, World';
    ShowMessage (strMessage);
end;
```

请留意到 strMessage 这个字符串变量的宣告，是写在 begin 这个保留字之前，而真正执行的源码则是写在它之后，再说一次，如果对于任何部分觉得不太清楚，别担心，所有程序都会随着您读到越后面，而有更详细的说明。

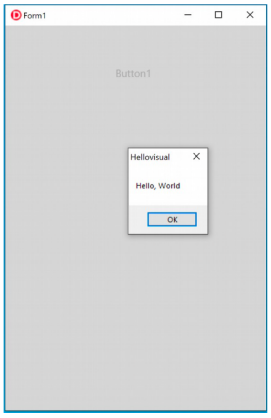
笔记 您可以在名为 01 的目录中找到本章所有范例的原始码，为了容易辨认，在这个范例的文件夹里面有一个名称跟项目档很相似的档案，我把这个文件名的前面加上了”Form”，以利区分，这也是我在本书当中用来为档案命名的标准规则，项目的结构将会在本章的后面篇幅介绍。

在图1.3里面，您可以看到这个简单程序的Windows上面透过启用 FMXMobilePreview 模式后执行的结果，但您也可以把这个范例程序拿到Android或者iOS，macOS上面执行，结果也会相同的(但这需要先在IDE里面做一些额外的设定才能正常运作喔)。

笔记 FireMonkey Mobile Preview 模式可以让 Windows 应用程序看起来有点像是行动装置 App, 在本书中的大多数范例中，我都启用了这个模式。要启用这个模式，只需要在项目原始码当中的 MobilePreview 单元档案里面加入一个 use 的指令即可。

现在我们已经介绍怎么撰写、测试一个范例程序了，让我们回头仔细看一下细节，一如我们在本章开始的时候我提到的顺序。我们要介绍的第一件事，就是如何阅读程序，各个不同部分的源码要如何撰写、以及我们刚建立这个项目是怎么组成的。（这个项目会包含有PAS档案跟DPR档案）

图 1.3: HelloVisual 范例只有一个简单的按钮执行的画面：



语法和源码样式

在我们开始介绍Object Pascal程序指令之前，我们要先来看一些Object Pascal源码的样式，我在这里想要点出的问题是：在程序语法之外（我们还没开始介绍），我们要怎么来撰写源码呢？这个问题并没有固定的标准答案，每个人都有自己习惯的写法，不同的习惯写法就会让源码看起来有不同的样式。然而，还是有一些固定的规范必须先介绍给大家知道，例如批注、大小写、空格符，以及多年以前曾经被称为*美观打印*的排列样式（这里指的美观当然是让人来阅读，跟计算机没有关系），而这个名词现在也已经很少听到人提起了。

通常源码样式是为了让人在阅读源码的时候可以更简洁、更快速的了解源码，这些样式跟格式就是您可以决定的一些源码的排列方法，好让程序看起来更整齐。而要让源码看起来整齐，就必须要坚持相同的源码样式，不管您选择了哪一种样式，记得要在整个项目的所有档案当中都用同样的源码样式，不然反而会让源码看起来更难懂。

提示 IDE(Integrated Development Environment，整合开发环境，可以简称为开发环境)已经支持自动格式化源码的功能（可以选择针对单一档案或者整个项目），您可以按下快捷键 Ctrl+D 要求 IDE 对目前的档案进行源码样式重新格式化，这个格式化的功能可以让我们自己对 40 几个源码样式的细节做设定。(请从 Options 选单当中找到这个设定画面)，您也可以把这西设定汇出，让同一团队的其他开发人员共享这些设定值，这样可以让整个团队的程序样式更为一致。但是，自动格式化的功能对于最新加入的部份编程语言功能并没有完全支持喔。

程序批注

虽然源码通常已经很容易被读懂，但如果加上一些批注的话，其他人就更容易看的懂（如果过了一段时间之后，我们又回头看一些自己也的程序的话，有批注也更容易看的懂）为什么当时这段程序要这样写，以及当时写这段程序的前提是什么。

传统的Pascal程序批注，是以两个大括号、或者小括号带星号来标注某段文字为批注的，而近期版本的Object Pascal编程语言则是把C++的批注语法，也就是用两个斜线来标注其后的文字为批注，所以，以下的三种写法，都是目前的Pascal语言可以辨识的批注写法：

```
// 从左边出现了两个斜线以后，到本行的末端都会被视为批注
{ 这里面的跨行文字都是批注 }
(* 这是另一种
多行文字批注的写法 *)
```

第一种批注的写法在目前流行的编程语言里面几乎都通用，但它并不是Pascal语言里面一开始就支持的，这语法是从C/C++语言借来的，C/C++语言也使用/* 批注文字 */ 来标注多行文字，这已经在C#, Objective-C, Java 跟JavaScript 里面都通用了。

第二种写法比第三种更普及，这个写法在欧洲最常见，因为大多数欧洲语系的键盘并没有小括号(或者小括号并不容易输入，需要多键组合才打的出来)。换句话说，最古老的语法有点过时了。

单行批注的语法很有用，常用来写短短的批注，或者把特定一行源码先暂时标注掉，这个语法也渐渐的成为在Object Pascal最常被使用的批注语法。

笔记 在 IDE 的编辑器里面，您可以按下 Ctrl+/这组快捷键，来把单一一行源码，或者选择多行源码进行批注或者解除批注，这组快捷键在英文键盘里面可以直接使用，但如果是其他语系的键盘或输入法，就要先确定一下/这个符号的位置，实际的按键，可以从编辑器的功能选单(按鼠标右键就会显示了)来清楚的看到。

我们介绍了三种不同语法的批注文字，这些语法可以帮助我们把单行或者多行的源码先变成批注文字。如果您希望把源码或程序档案里面的部份文字变成批注，则您可以套用前述三种语法的不同排列来达成批注文字当中还有其他批注文字这种设定，但多行批注如果要包含其他批注的话，两个批注文字的语法不能用同一种喔：

```
{ code...
        {comment, 这段批注底下的其他文字会不被当成批注喔}
code... }
```

上面这段源码会被编译器当成错误语法，因为第一个大括号在第二行遇到了结束的大括号，因此第三行已经不会再被视为批注文字，所以编译器就会判定语法有错，我们可以把它改写如下：

```
{ code...
```

```
        // this comment is OK
code... }
```

这样的写法，由于两种批注语法不互相冲突，所以我们如果要把整段文字或源码的批注状态取消，就只要把前后两个大括号拿掉就行了，第二行还是可以保持批注文字的状态。

笔记 在大括号之后如果出现一个钱字号$，就不再代表是批注，而是编译器的设定代码，例如在我们介绍的第一个范例程序当中，源码里面有一行写着{$APPTYPE CONSOLE}。编译器设定代码会让编译器去执行特定的一些动作，我们在本章后段加以介绍。

其实，编译器设定代码说到底也算是批注，例如{$X+ 这是一个批注}这样写也不会造成语法错误，这种写法同时扮演两个角色，不过大多数的程序人员都还是会把编译器跟批注给分开来写就是了。

批注和 XML 文件

批注当中有一个特别的种类，对其他编程语言也是通用的规则，这种特别的批注出现时，编译器会有特别的处理方法。这些特别的批注，会直接被 IDE 建立额外的文件到 IDE 协助功能中，编译器也会依此建立XML说明文件。

笔记 在 IDE 当中，协助功能会自动显示关于识别符号的相关信息(包含该识别符号的型别，以及它是在哪里被宣告的)。透过 XML 文件批注，我们可以对特定的识别符号写一些需要注意的信息，或者关于它的细节，而这些说明文件是可以直接写在原始码里面的喔。

XML说明文件可以透过设别的批注符号 /// 或者 {! 来启用。在这些批注中，我们可以用一般文字或者特定的XML卷标(用卷标更好)来对要加入批注的源码或者识别符号指派特定的信息，例如该识别符号所需的参数、回传值等等。以下我们用一般文字来做个简单的例子:

```
Public
  /// 这是一个自定方法
  procedure CustomMethod;
```

如果我们启用了XML文件产生功能，上面的源码会被编译器转译成底下的格式:

```
<procedure name="CustomMethod" visibility="public">
    <devnotes>
        这是一个自定方法
    </devnotes>
</procedure>
```

而这些信息在我们使用IDE的时候，当鼠标光标移动到该识别符号(CustomMethod)的时候，就会显示出来，如图1.4所示：

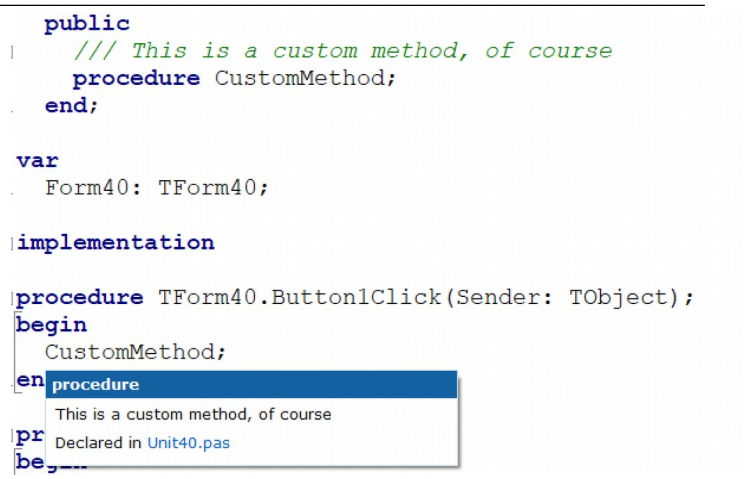


图 1.4: Delphi IDE 的帮助功能显示出以 /// 批注符号加入的 XML 说明信息：

如果我们在批注当中依照指引的建议，加入了 summary 区块，就会直接显示IDE的帮助窗口:

```
Public
  /// <summary> 这是一个自定方法 </summary>
  procedure CustomMethod;
```

这么做的优点，是还有许多其他的XML标签，可以让我们用来说明参数、回传值以及其他更细节的信息，相关的卷标说明，请参考这个网址:

```
http://docwiiki.embarcadero.com/RADStudio/en/XML_Documentation_Comments
```

识别符号（Symbolic Identifiers）

一个程序是由许多个不同的识别符号所构成的，我们会用这些识别符号来为不同的程序区段命名（例如数据型别、变量、函式、对象、类别等等）即使我们可以几乎可以使用我们想用的任何一个识别符号，仍然需要遵守一些规则：

- 识别符号不能包含空格符（空格符会用以区分不同的识别符号）
- 识别符号可以包含英文字母与数字，包含字符与所有Unicode的文字，所以我们已经完全可以用任何一种语言的文字（当然用中文也行）来做为识别符号的名称了。(有些时候也不建议这么做，例如IDE里面的工具部分，有可能并不完全支持)
- 在传统的ASCII符号之外，识别符号只能包含底线(_)，其他的ASCII符号则不可以用在识别符号上，不可以使用的符号，包含有+, -, *, /, = 以及所有标点符号跟括号、特殊字符(像是@ # $ % ^ & \ |)，我们只能用Unicode的字符，例如 ☼ ♪ 或 ∞ 都可以。
- 识别符号必须用底线或字符来开头，不可以用数字开头（所以数字当然可以当成识别符号的一部分，只是不能当成第一个字）我们在这里提到的数字是指0-9的阿拉伯数字，至于Unicode里面的其他语言数字，例如国字的一或壹，甚至是全型的 0 或 1 则都没有问题。

以下是很常见的一些识别符号命名方法，我把他们列在名为IdentifiersTest的范例程序中：

```
MyValue
 Value1
 My_Value
 _Value
 Val123
 _123
```

以下则是一些合法的Unicode识别符号：

```
Cantù (拉丁文)
结 (简体中文)
画像 (日文汉字)
☼ (Unicode 里面的太阳符号)
```

接下来我们也介绍一些不合法的识别符号：

```
123 (数字开头)
1Value (数字开头)
My Value (内含空格符)
My-Value (内含特殊符号)
My%Value (内含特殊符号)
```

提示 如果您在程序执行状态中想要检查识别符号是否合法(需要这样做的情境很少发生，除非您撰写的是帮助其他开发者的工具)，在运行时间函式库(Runtime Library)当中有提供这样的一个函式，名为 IsValidIdent。

大小写视为相同与使用大写字母

跟许多其他的编程语言不同，许多以C语言为基础的编程语言(像C++, Java, C#, 跟 JavaScript)都是把英文字母大小写视为相异，英文称为Case-Sensitive，Object Pascal是把英文字母大小写视为相同的，英文称为Case-Insensitive。

因此，对Object Pascal的编译器来说，Myname, MyName, myname, myName, 以及MYNAME这五个大小写各有不同的字符串，编译器会把他们全部当做完全一样的字符串。在我个人的观点中，把英文字母大小写视为相同，绝对是正面的功能，因为这样一来，因为拼字错误或者打字一时打错而造成语法错误的机会，会比其他语言发生的机会来的低。

如果我们把Unicode当成识别符号的情况也采计进来，事情就会变得更为复杂了，当我们把大小写视为相同时，只有一个字母大小写相异的关键词就会被视为完全相同，因此也就可以避免在语意上完全相同而只有大小写拼法不同的情形发生了。

```
cantu: Integer;
Cantu: Integer; // 错误: 因为与前一个名称重复
cantù: Integer; // 正确: 这是完全不同的名称了
```

警示 在 Object Pascal 的大小写视为相同的规则中，只有一个情形例外，就是组件函式库套件(component package)的 Register 函式，这个函式一定必须写成第一个字母大写的 Register, 因为必须兼容于 C++。

当然，当我们在引入由其他语言制作的函式库时，也必须随着该函式库制作时所使用的字符大小写，才能正确的呼叫、使用这些函式库。

在字母大小写的规则当中还是有一些问题的，所以我们首先要意识到不同大小写的识别符号实际上是完全相同的，所以一定要避免在不同的地方使用相同的识别符号，其次，我们也要适当的使用大写字母，让我们的源码更容易被阅读。

编译器并没有强制我们要在项目当中使用持续的规则，但使用持续的规则真的是个好习惯。通常大家会很习惯把第一个字母大写，当我们要用连续几个英文字作为识别符号的命名时，通常会让每个有意义的英文字的第一个字母大写，例如：

```
MyLongIdentifier

MyVeryLongAndAlmostStupidIdentifier
```

这样的习惯通常被称为Pascal-casing, 相对于Java以及其他以C语言语法为基础的编程语言所使用的Camel-casing：第一个英文字的首个字母小写，其他的英文前缀个字母均大写：

```
myLongIdentifier
```

实际上，目前也越来越常在Object Pascal的程序里面看到Camel-casing的规则了，目前仅剩Class宣告、参数宣告以及其他全局变量的范畴内还会使用Pascal-casing，在本书中，还是会尽量在所有的识别符号中都使用Pascal-casing的规则的。

使用空格符

空格符、tab跟换行符号在源码里面，几乎是完全被编译器所忽略掉的。这三个字符对于编译器来说都被当成空格符。空格符只用来让源码在阅读上更为舒服，在编译作业上完全没有作用。

跟传统的BASIC不同，Object Pascal允许我们把一个程序叙述句（statement）用很多行的源码来表达。允许用多行源码来表达一个程序叙述句的缺点，是我们自己必须记得在叙述句的最后加上一个分号，让编译器知道这是一个程序叙述句的完结点。Object Pascal的多行叙述句的唯一限制，是不能把字符串用多行来表示。

以下这几段程序虽然看起来奇怪，但他们都描述了同一件事情：

```
a := b + 10;
```

```
a := b
+ 10;
a :=
// 在源码当中夹一行批注, 也是可以的
 b + 10;
```

再次强调，在源码里面使用空格符跟换行符号并没有一定的规则，但有几个常用的写法：

- 编辑器画面通常在每行达到80个字符之后，就会切行，如果你的源码超过了这个长度，就会被推到下一行去，这是为了让你的源码看起来更容易阅读，因为这样就不用横向卷动，在比较小的屏幕上也能够很快的把源码阅读完毕。原本每行80个字符的用意，只是让源码打印出来的时候比较好看，而近年来也很少看到源码打印的需求了。

- 当函式或程序有多个复杂的参数时，我们通常会让每个参数用单独一行来表示。

- 我们也可在批注前后留下一行空白行，这样可以让我们的程序读起来更清楚易懂。

- 记得在呼叫函式的时候，在每个参数之间加入一个空白，甚至是在不需要参数时，也在括号当中留一个空格符，我也会在表达式里面，让每个运算符号前后多放一个空白，看起来好读多了。

源码内缩

关于空格符使用上的最后一个建议，是跟典型的Pascal语言格式相关的议题，原本是为了让打印出来的源码比较美观，但目前已经都统整为源码内缩来呈现了。

提示 源码内缩的规则通常是依照个人习惯的主观想法，我不想挑起『用 Tab 或空格符哪个比较好』的战争。我在这里介绍的只是 Object Pascal 世界里面”最常用”或者”标准”的源码格式。在 Pascal 世界的历史中，比较常被称为 pretty-printing，这个词目前已经不常见了。

这个规则非常简单：每当我们需要写一段复合的程序叙述句，就把整段源码内缩两个字符（不是tab字符，tab字符是C语言的开发人员常用的），如果这段源码当中还有其他复合的程序叙述，则该段源码再多内缩两个字符，依此类推：

```
if ... then
```

```
  statement;
if ... then
begin
  statement1;
  statement2;
end;
if ... then
begin
  if ... then
    statement1;
  statement2;
end;
```

重申一次，不同的程序人员在这个常用的规则上会有自己惯用的作法，有些程序人员会把begin到end之间的源码内缩，而有些程序人员则会把begin这个关键词放在前一段源码的最后（C语言大多都是这样做，译者也是习惯如此），这是因为个人习惯而有些许不同而已。

相似的规则也常用在变量列表跟数据型别的定义上，就是在下面这段范例源码当中 type 跟 var 保留字后面的源码：

```
type
  Letters = ('A', 'B', 'C');
  AnotherType = ...
var
   Name: string;
   I: Integer;
```

提示 在上面的源码中，您可能会觉得奇怪，为何 string 跟 Integer 这两个不同的型别要用在起始字符一个大写一个小写。在原始的 Object Pascal 格式指引中有提到『型别，例如 Integer 属于识别符号(identifier)时，第一个字符建议大写，但用保留字(reserved word)例如 string 宣告变量时，建议全部小写。』

在以往的作法，常常会在type宣告新的型别名称时，让所有的等号都放在同一个位置对齐，以及让变量宣告时的冒号都对齐起来，但现在已经很少见了，上述的源码如果用以往的规则来编排，就会变成：

```
type
  Letters     = ('A', 'B', 'C');
  AnotherType = ...
```

```
var
    Name : string;
    I       : Integer;
```

内缩的排列也常用在跨行的程序叙述句上面，通常第二行以后的源码，就会被内缩，而函式的参数如果长过一行，也会被用内缩的方式来显示：

```
MessageDlg ('This is a message',
    mtInformation, [mbOk], 0);
```

强化语法标示

为了让Object Pascal的程序更容易被阅读与编写，IDE的编辑器具备了一个名为强化语法标示的功能。根据我们所缮打的源码语法跟关键词，这些源码会被以不同的颜色跟字体加以标注。默认设定中，关键词会以粗体显示、字符串跟批注会以不同的颜色显示（而且通常会是斜体）等等规则。

保留字、批注跟字符串这三个类型的程序元素绝对是这个功能中对程序人员帮助最大的，透过这个功能，我们可以一眼看出源码是不是拼错字了？字符串是不是少打了一个引号？以及跨行的批注有没有把前后批注的标示符号正确的标示上去。

您可以很容易的透过Editor Colors这个设定来设定您自己喜欢的语法标示设定，如果您的开发工作只有自己需要看这些源码，则您可以直接设定，如果您需要跟其他程序开发人员一起工作，则请您还是先使用标准的颜色布景吧，我自己也发现一旦习惯了特定的布景颜色以后，一下看到其他颜色跟样式的画面的确很容易楞住一下，不知所措。

错误检知和源码检知

IDE编辑器有许多功能可以协助我们写出正确的源码，最直觉的应该就是错误检知这个功能了，当我们输入了错误的语法时，编辑器画面上就会立刻在第一个发生错误那一行标上一串红色的书名号，让我们知道编译器无法辨识该段源码，目前已经连文本编辑器跟文书软件也都有相同的功能了。

笔记 在您第一次试着撰写 Object Pascal 程序的时候，请务必记得也要先把适当的 unit 引入，这样可以避免您的程序档案最上方被标注许多的错误，

正确的引入其他的 unit, 可以解决掉不少错误标示。这个议题已经在 Delphi 10.4 当中透过新的语言服务器协议基础(LSD based, Language Server Protocol based)被明确的强调了。

其他功能，例如源码自动完成，会协助您显示当时我们输入的可能程序描述句，它会列出许多符合我们输入的函式名称或者属性名称，我们只要用下拉选单选择我们适用的function即可。又或者是一个函式的参数，也会被以下拉式选单的方式呈现。又或者我们可以按着Ctrl按键，用鼠标左键点选某一个源码里面的文字，就可以直接跳到该变量/型别被宣告的地方了。接下来我们就不再赘述关于IDE编辑器的功能了。我们还是把主要篇幅用来介绍Object Pascal语言本身吧。

编程语言的关键词

这里所指的关键词，就是由编程语言特别保留下来的识别符号。这些符号都是在编程语言里面已经预先赋予了功能，所以我们在整个程序的任何部分都不能拿来当成我们自己写的程序的符号（例如变量名、Class名等）。而基本上，指示词（directives）跟关键词是有所不同的:关键词是不能拿来当做变量、类别名称等标识符的，但指示词只要不被放在{$}这个符号组合中，就没有影响，所以指示词可以有其他的用途，但实务上，建议还是不要拿任何关键词（包含指示词）来做为标识符比较好。

如果您写了以下这样的源码（property这个字是关键词喔）：

```
var
    property: string
```

编译器就会给您这样的错误讯息：

```
E2029 Identifier expected but 'PROPERTY' found
```

警告您使用了关键词作为标识符，这是不被允许的。通常当您误用到关键词的时候，您会在编辑器或者编译器的时候得到错误讯息，当然用到不同的关键词会有不同的错误讯息出现。当编译器发现到关键词出现了，而觉得这个关键词出现的位置不对，就会依照错误关键词出现的地点回报错误。

在这里，我不打算把完整的关键词列表列出来，仅列出我们在写程序的时候比较常用到的关键词，并把他们依照功能分组列出，即使如此，要完整的涵盖到这些关键词仍旧需要用掉好几个章节来说明。您可以参考官方参考文件，网址是:

http://docwiki.embarcadero.com/RADStudio/en/Fundamental_Syntactic_Elements_(Delphi)#Reserved_Words

笔记 请注意，部分关键词会在不同的地方出现，在这里我谨列出最常见的部分，有些关键词可能会被列到两次。原因之一，是经过了这么多年以后，编译器的团队希望不要再导入新的关键词，以免旧有的程序反而失效了，所以他们把其中一些关键词再度回锅了。

那么，我们就开始关键词的旅程吧，这些关键词有一些可能在您过去写程序时或者在前面的章节当中有见过，他们就是构成整个应用程序项目的骨干：

program	标明应用程序项目的名称
library	标明函式库专案的名称
package	标明套件函式库专案的名称
unit	标明单元文件的名称，单元文件也就是源码的原始档
uses	指示当前这个单元文件会参考到哪些单元档案
interface	单元文件的区段，用来进行宣告
implementation	单元文件的区段，用来放置实作的源码
initialization	当程序启动时，要先被执行的源码区段
finalization	当程序结束前，最后要被执行的源码区段
begin	宣告一个源码区块的开始
end	宣告一个源码区块的结束

另一组关键词则是跟一些基础数据型别的宣告与变量相关的，兹列出如下:

type	标明开始进入数据类型声明区段
var	标明开始进入变量宣告区段
const	标明开始进入常数宣告区段
set	定义一个集合变量
string	定义一个字符串变量，或者自定的字符串型别
array	定义一个数组型别
record	定义一个复合数据型别
integer	定义一个整数变量
Real,single, double	定义一个浮点数型态的变量
file	定义一个档案变量
record	定义一个复合数据型别

笔记　在后面的章节中，我还会介绍更多 Object Pascal 的资料型别。

第三组关键词则是介绍 Object Pascal 编程语言的基础叙述句，例如条件判断式跟循环，也包含了函式(function)跟程序(Procedure):

if	标明一个条件判断式
then	将条件判断式与符合条件时执行的源码分隔的符号
else	标明条件判断式中，不符条件时要执行的源码
case	标明一个多重选项的条件判断式
of	把多重选项判断式的条件与各个选项分隔的符号
for	标明一个固定次数的循环开始
to	标明 for 循环将变量递增计算时的最终数值
downto	标明 for 循环将变量递减计算时的最终数值
in	标明在列举循环当中，用来表示要被列举的组合变量
while	标明一个条件化的循环开始
do	把 while 循环的条件式与要执行的源码做分隔的符号
repeat	标明一个具终止条件的循环开始
until	标明repeat循环的终止条件
with	标明要针对特定的数据结构进行处理
function	标明一个会回传执行结果的子程序（名为函式）
procedure	标明一个不会回传执行结果的子程序（名为程序）
inline	要求编译器对函式或程序进行优化
overload	允许同名的函式或程序被重复使用（称为多载）

以下则是跟类别、对象相关的关键词：

class	标明一个新的类别型别
object	用来标明一个就的类别型别（目前已不再使用）
abstract	标明一个抽象类，表示该类别还没有完全被定义
sealed	标明一个已封锁类别，该类别不能再被继承
interface	标明一个接口型别（这个关键词也在第一组当中出现过）
constructor	一个类别或对象的初始方法
destructor	一个类别或对象的清除方法

virtual	一个虚拟方法，在衍生类别中需要被实作出来
override	在衍生类别中，实作虚拟方法的关键词
inherited	直接呼叫、引用父类别的方法
private	宣告类别中不能被外界存取的属性、事件或方法
protected	宣告类别中有条件供外界存取的属性、事件或方法
public	宣告类别或记录中可以完全被外界存取的属性、事件或方法
published	宣告类别中特别为了用户建立的属性、事件或方法
strict	比private跟protected限制更为严格的类别区段
property	被对应到变量或方法的一个符号，称之为类别的属性
read	属性的数据源
write	属性的变更方法
nil	表示空对象，在许多有指针类型的语言当中也都有乡对应的特别符号，在C里面称为NULL

还有一小群跟例外处理(我们在第11章里面会介绍)有关的关键词：

try	标明例外处理区块开始
finally	表示不管例外发生与否，都要被执行的区块
except	表示当例外发生时，要被执行的源码区块
raise	用来触发一个例外事件

另外还有一小群关键词是用来作为运算用的，我们会在本章稍后的篇幅『算式与运算符』的部份介绍到(有一些进阶的运算符则会在后面的章节介绍)：

as	and	div
is	in	mod
not	or	shl
shr	xor	

最后，我们列出一些比较不常用的关键词，包含一些不建议使用的旧的关键词。我们快速的看一下这些关键词的意涵，在后面的章节我们也会尽可能介绍：

default	意指一个属性的默认值
dynamic	宣告虚拟方法的另一个被实现的源码

export	传统用来输出的关键词，已经被下一个关键词取代了
exports	在 DLL 项目中，列出要输入让其他程序使用的函式
external	指向我们要绑定的外部DLL中的函式名称
file	用来定义传统的 file 型别，这个型别近年来已经很少用到了
forward	表示一个函式预先宣告，实际的宣告在后面的源码才会完整出现
goto	让源码跳到程序中另一个标注的位置继续执行，强烈建议绝对不要使用这个功能。
index	用在当需要引入或者输出函式时，标明当中有索引的属性(现在已经很少用了)
label	定义一个特别的程序位置，让goto指令能直接跳到此一位置继续执行，强烈建议绝对不要使用这个功能。
message	虚拟函式的替代关键词，和不同平台的讯息相关
Name	用以对应外部函式
nodefault	表示该属性没有默认值
on	用来触发例外状况
out	var关键词的替代字，是用来表示一个引用参数(call by reference)，但没有被进行过初始化
packed	改变记录(record)或者数据结构在内存中的排列
reintroduce	允许重复使用一个虚拟函式的名称
requires	在制作套件时，用来宣告所需要的其他套件名称

请注意，近几年来Object Pascal的关键词已经很少有新增的了，因为任何新增的关键词都有可能会使得已存在的源码在使用新版的编译器进行编译时，导致旧有的程序发生编译错误，因为谁也不敢保证程序人员一定不会用到什么英文字。Object Pascal最近新增的功能都不需要透过关键词来达成，例如泛型（generics）与匿名方法（anonymous methods）。

程序结构

您可能曾经把所有的源码写在同一个档案里面，就像本章的第一个简单的文本模式应用程序一样。而当我们越常开发图形接口程序，就越有机会在项目档之外使用到第二个原始码档案。这『第二个档案』就被称为*单元文件*，通常它的扩展名会是PAS（Pascal 原始档的意思），项目档的扩展名则会用DPR(Delphi项目档的意思)，这两种档案都会内含有Object Pascal的原始码。

Object Pascal透过了单元文件或者程序模块的使用提供了延伸性。事实上，单元文件就提供了模块化以及数据封装的功能，即使没有使用到对象，实际上已经也被注记为命名空间 (namespace)了。Object Pascal的应用程序通常都是由好几个单元文件所建立的，包含用来储存画面与数据模块的单元文件。事实上，当我们加入一个可视化的画面窗体到项目里面，IDE就会帮我们加入一个单元文件，这个单元文件正是对应所加入的可视化画面的源码。

单元文件无需定义画面窗体，两者之间会自动被关联起来，两者之间的类别、属性、方法、事件处理程序，都已经被自动连结好，无需我们额外做什么处理了。如果您要加入一个新的空白单元文件到项目里面，这个空白单元文件只需要几个简单的关键词来宣告几个必要的区段即可，如下所示：

```
unit Unit1;

interface

implementation

end.
```

单元文件的结构极其简单，就像上面的范例一样：

- 首先，单元文件要有一个整个项目不能重复的名字为之命名，同时也当作主档名（所以上面这个例子存档时，档名就会是*Unit1.pas*）
- 其次，单元文件一定要有一个interface区段，用来宣告让其他单元文件可以使用、存取的资料。
- 第三，单元文件要有implementation区段，用来实作这个单元文件里面真正的源码，这里的源码也可以比interface区段所宣告得来的更多，只是在interface区段没有宣告的，就只有同一个单元文件的其他源码可以使用，不管是方法或属性都一样。

单元与程序名称

如同我提过的，单元的名称必须跟单元文件的文件名一致，程序名也一样，要为单元重新命名的话，我们应该使用Project Manager里面的重新命名功能，此时更名前后的单元使用在项目中都会保持同步，档案也会维持只有一份改名后的档案 (使用IDE里面的另存新文件功能，会在磁盘里面存在新旧两个单元档案了)。当然您也可以直接从档案总管把档案直接改名，但是如果改名后，您没有到单元文件里面把第一行的单元名称也一起做修改的话，在编

译器的时候，就会看到一个错误发生（或者只在加载项目的时候，IDE就会告诉你有错误了），以下的讯息就是当我们只改了档名，却没有同步修改单元名称时会发生的错误：

```
[DCC Error] E1038 Unit identifier 'Unit3' does not match file name
```

这表示单元名称也必须符合Pascal识别符号规则，以及文件系统的命名规则，例如像我们前面提到过的，不能包含空格符、不能有特殊符号（除了底线）。只要我们使用了符合规范的名称来为单元命名，就会自动被储存为合法的文件名，所以这一点我们不用太过担心。当然凡事都有例外，就是Unicode的符号，有些符号并不是文件系统允许我们拿来作为档名的，就别故意挑战文件系统了。

用.来为单元命名

单元名的规则还有一个延伸的规则，就是单元名可以包含. 所以以下列出的单元名称也都是合法的：

```
unit1

myproject.unit1

mycompany.myproject.unit1
```

依照这个常见的规则，这些单元需要也被储存为相同的档名，用.包含在檔名之中(例如假设有一个单元名为MyProject.Unit1，这个单元就要存放在MyProject.Unit1.pas档案之中了)

这个延伸规则的由来，是因为单元名称必须是唯一的，而随着Embarcadero跟第三方开发商所提供的单元文件越来越多，单元文件的名称就变复杂了，所以目前随着Delphi开发工具所预载提供的RTL单元以及各种不同功能的单元文件，都可以看到有许多用.来构成单元名称的情形，例如：

System	代表核心RTL的单元
Data	代表数据库存取与相关的单元
FMX	代表FireMonkey平台与跨装置组件的单元
VCL	代表Windows平台视觉组件库的单元

笔记 您经常都会在单元文件的全名里面使用到名称里面有.的单元，或者函式库的单元文件。不过也可以只在程序的参考单元当中使用到整个单元名称的最后部分（这也是为了让旧版的程序能够兼容新版的编译器），您只需要设定项目选项中的对应项目即可，这个设定选项的名称是”Unit scope

names”，它是一个以分号做项目区隔的列表。不过请注意，使用这个功能相对的会让编译的速度比使用完整单元名称的时候变慢许多。

更多关于单元文件的结构

除了 interface 跟 implementation 这两个区段之外，每个单元还可以有 initialization 跟 finalization 这两个非必要的区段。Initialization 是用来处理该单元被执行时最开始的源码，而 finalization 则是用来处理该单元在程序结束时要处理的源码。

笔记 您也可以在类别的建构方法(constructor)当中加入 initialization 源码，Object Pascal 的许多最新功能在第 12 章会介绍到，使用类别的建构方法可以帮助链接程序移除非必要的源码，这也是为什么建议大家使用类别的建构方法跟解构方法(destructor)，而比较不建议使用 initialization 跟 finalization 区段的原因。在过去的历史中，initialization 区段还是需要依靠 begin 这个关键词来进行宣告的，begin 的类似用法仍然是项目源码的标准。

换句话说，单元的结构，包含了所有可能的区段与一些简单的元素，应该长得像下面这个范例源码：

```
unit UnitName;
interface
// 其他我们在本单元会引入的单元名称，在 interface 这个区段中宣告
uses
    UnitA, UnitB, UnitC;
// 要公告周知的型别定义
type
    NewType = TypeDefinition;
// 要公告周知的常数
const
   Zero = 0;
// 全局变量
var
   Total: Integer;
// 要公告周知的函式与程序
procedure MyProc;
implementation
// 其他在 implementation 区段我们会引入的单元名称
```

```
uses
    UnitD, UnitE;
// 不对其它单元文件告知的全局变量
var
    PartialTotal: Integer;
// 所有被公告的函式都必须在此被实作
procedure MyProc;  begin
// ... MyProc  这个程序的源码
end;
 initialization
// 非必要的 initialization 区段的源码
 finalization
// 非必要的  finalization 区段源码
end.
```

一个单元的 interface 区段存在的意义，是为了告知其它单元，这个单元包含什么，能为其它单元或项目提供什么。而 implementation 区段则包含了所有其它单元都无法得知的源码。这也是Object Pascal之所以可以提供信息封装，而不需要靠类别或对象就能达成该功能的原因。

我们可以发现，单元的 interface 区段可以宣告不少型别各异的元素，包含程序、函式、全局变量，以及数据型别。数据型别当然是最常出现在这个程序区段中的。IDE环境会自动放一段新的类别宣告，当我们建立一个新的可视化画面窗体，宣告窗体定义并不是Object Pascal当中包含单元这个功能的唯一理由。我们也可以在单元当中只有源码，只有函式与程序（就像传统 Pascal 程序的作法），甚至在单元里面有新的类别，但不用参考其它的窗体的或者可视化组件。

Uses 条文

Uses条文位于interface区段的开头部分，是用来标示我们在该单元中需要参考的其它单元名称。是标明我们在这个单元当中，为了定义数据型别时，需要参考的其它单元，而参考的程序也限于那些单元的数据型别，例如我们在画面窗体当中定义的组件。

第二个uses条文是出现在implementation区段的开头，是标明我们只在源码实作阶段需要参考的单元文件。当您只需要参考其它单元的源码，例如子程序、方法，我们就得在implementation区段当中的uses来标明这些单元的

名称，这个作法可以减少对档案的依赖度并减少编译所耗费的时间。在uses条文中被标明要参考的单元文件，都必须位于项目目录当中，或者IDE环境的搜寻路径当中，这样编译才不会出问题。

提示 我们可以在项目设定里面设定搜寻路径(Search Path)。系统在编译作业中也会参考 Library Path 里面的单元文件，这个设定是 IDE 的全局设定。

C++的程序人员们请注意，uses叙述跟C++的include叙述句并不对等，uses叙述句的效应只会把预先编译的单元的interface部分引入。Implementation区段的源码会在该单元实际被编译器处理的时候才被考虑，被引入的单元文件，可以以原始码的格式(PAS檔)或者编译过的二进制文件(DCU文件)的形式存在前述的目录或路径中。

虽然在Object Pascal当中并不常用到，但Object Pascal的编译器设定当中也有一个类似C/C++ include的编译器设定，名为$INCLUDE。然而跟内嵌在原始码不同，这些特别的引入档会被部分需要共享编译器设定的函式库或者在许多单元文件要共享其它设定的时候才被用到的，而且通常会使用INC这个特别的扩展名，这个编译器设定会在本章的最后介绍。

警告 请注意，Object Pascal 的二进制编译文件(DCU)只会在使用相同版本的编译器与系统函式库时兼容。用旧版的编译器编译的二进制文件，通常无法与后来版本的编译器兼容，这一点不可不察。在同一个版本的更新套件一定会维持兼容性。换句话说，在 10.3.1 版里面的档案一定会跟 10.3.x 的所有版本兼容，举例来说，但就未必会跟 10.2 或 10.4 版的程序兼容了。在 Delphi 11 里面对版本的作法有了改变，在 11.x 的任何改版都会保持跟 11 的兼容性，但 12 版可能就会无法兼容。

单元与界限

在Object Pascal当中，单元正是数据封装与源码界限的关键，在这个规范下，单元能提供的源码界限甚至比类别中private或public关键词能提供的还更重要。一个标识符(例如变量、程序、函式或数据型别)的界限，是表示这个标识符能够被其它源码存取的范围，所以也被称为该标识符的可视范围。基本规则就是只有在该标识符的界限中，它才是有意义的，因此只有在这个单元当中的程序、函式才能使用这个标识符，我们无法在标识符的界限之外使用它。

笔记 请注意，Object Pascal 跟 C、C++都不一样，Object Pascal 在一般源码区块里面不允许变量、常数的宣告。当我们已经进入了 begin-end 的源码区块范围之后，在这里面就不能够再宣告任何变量了。

通常一个标识符只有在它被定义之后才能使用。但在Object Pascal当中，也有方法在一个标识符被完整定义之前先进行宣告，但我们在完整考虑了定义与宣告的规则后，应该可以发现其实他还是遵循着Object Pascal的基本规范的。

假设把整个程序的源码写在单一一个档案里面是有意义的，那么，这个规则会怎么修正好让我们在使用多个单元文件的时候能够遵循呢？简单的说，当我们透过uses条文把其它单元引入的时候，在被引入的单元中，interface区块所宣告的所有标识符也在新的单元文件里面变成可以被存取的了。

反之，如果在interface区段宣告了一个标识符（可能是型别、函式、类别、变量等等），所有引用现在建立的这个单元的其它所有源码也都可以看到这个刚宣告的标识符了。但如果是在implementation区段中宣告这个标识符的话，则这个标识符就只能在自己这个单元文件中被看见，其余引用这个单元的源码都无法看见了，我们可以把它理解成区域标识符，就像局部变量那样。

把单元文件当成命名空间来使用

我们已经看过了uses叙述句，它是让单元文件能够看见其它引入的单元文件相关标识符的标准技术。这时您可以存取这个单元中的所有定义，但有时在两个单元文件里面可能宣告了相同的标识符，例如您可能有两个类别，或者两个子程序使用了完全相同的名字。

在这种情形下，我们可以简单的把单元名称前置在这个被重复使用的标识符之前。举个例子来说，您可以引入在Calc这个单元当中名为ComputeTotal的程序，这时我们可以把它写成Calc.ComputeTotal，IDE不常要求我们一定要这么写，但当我们在两个不同单元中有相同的标识符时，这样写可以避免重复，以及让编译器不会误解。

然而如果您曾经深入看过系统或第三方组件的程序，您应该会发现许多函式跟类别的名称重复，最常见的例子就是在不同平台的可视化组件中，常

常有相同命名的组件，当您深入看到TForm或TControl的源码时，里面有很多类别或函式会依据您所引入的单元来决定要执行哪一个函式。

如果使用了相同名称的两个单元，正好都被您的单元文件引入了，最后被引入的那个单元会抢到该名称的使用权，编译器也就会把该名称直接对应到最后被引入的单元文件的标识符去，如果您无法避免这种情形的话，请一定在重复标识符名称的前面加入该单元文件的完整名字，这样编译器就不会弄错了。

笔记 Delphi 开发人员因为有名为 interposer classes 的技术，而享有让两个类别使用相同名称的方便性，关于这个技术，本书后面的章节会加以介绍。

程序档案

我们在前面的篇幅已经看过了，Delphi的应用程序文件会包含两种源码档案：一个是会出现多次的单元文件，而另外一个是只会出现一次的程序文件，或者我们也可以叫它项目档。单元文件可以看成是第二层的档案，所有单元文件都会被扮演主要阶层角色的项目档所引入。

理论上这是对的，在实务上，项目档通常是自动产生出来的档案，而且有其被局限的角色。项目文件只用来启动这个应用程序、通常会建立、执行主要的画面窗体（如果应用程序是可视化应用程序时）。项目档的内容也可以手动编辑，但是当我们修改项目设定的时候，这个档案的内容就会被自动修改（就跟应用程序当中的其它对象跟画面窗体一样）

项目文件的结构通常比其他单元档案来的简单，以下就是一个简单的项目档，这个内容会是由IDE帮我们自动产生：

```
program Project1;
uses
    FMX.Forms,
    Unit1 in 'Unit1.PAS' {Form1};
begin
    Application.Initialize;
    Application.CreateForm (TForm1, Form1);
    Application.Run;
end.
```

从上面的源码我们可以看到，它只有一个简单的uses区段，以及透过begin-end标明的应用程序主要源码，这个程序的uses叙述句非常重要，因为它们会被用来进行编译、链接成应用程序执行文件。

提示 在项目文件里面的单元列表对应了在 IDE 当中项目管理员画面的单元列表，当我们在 IDE 里面加入一个单元到这个项目里面的时候，这个单元的名称也会自动被加入到这个项目档的 uses 区段里面。当然，如果我们从 IDE 里面删除了一个单元文件，项目文件的 uses 区段也会立即有反应。
反之如果我们直接编辑项目档，从里面直接删除某个 uses 里面的单元文件，在 IDE 里面的设定画面也会同时立即有反应。

编译器设定

程序结构里面另一个特殊的部分（跟其他实际的源码相比），就是编译器设定了，我们稍早曾经提到过，这些特殊的指令是给编译器用的，会以以下的格式撰写：

```
{$X+}
```

有些编译器设定只是一个简单的字符，就像上面这个例子，用加号或减号来表示该设定是有效或者无效，大多数的设定会有一个长一点或者能够被判读的写法，使用ON与OFF来表示有效或者无效，而部分设定值则只有较长的写法，没有像上例这种简写。

编译器设定通常不是直接把源码编译成二进制，而是告诉编译器，这个设定值出现之后，要把编译器当中的部分设定进行调整以后再行编译之后的源码，大多数的时候，我们可以透过修改IDE当诸的项目设定值来调整这些设定，即使有些情况下，我们只需要对一个单元或一部分的源码进行编译器设定值的调整。
在后续的篇幅中，只要提到相关的编程语言功能，我也会介绍一下相关的编译器设定，在这个章节中，我只稍微介绍一些跟程序流程相关的编译器设定，例如条件化定义(Conditional Defines)与引入(includes)。

条件化定义(Conditional Defines)

条件化定义让我们可以告诉编译器使用哪一部分的源码，或者忽略它。通常我们会在定义标识符或者引入单元的时候用到这个功能。在Delphi中有

两种类型的条件化定义语法，传统的$IFDEF, $IFNDEF与新型较有弹性的语法 $IF

这些条件化定义可以透过定义的识别符号或者在 $IF 的写法中用常数值的定义来区分。定义的识别符号可能是系统预先定义的(像是编译器或是平台定义的识别符号)，也可能是定义在特定项目当中的设定选项，或者在源码当中以其他编译器定义值($DEFINE)宣告的。

传统的 $IFDEF 跟 $IFNDEF 写法格式如下:

```
{$DEFINE TEST} ...
{$IFDEF TEST}
// 这部份的源码会被编译
{$ENDIF}
{$IFNDEF TEST}
// 这部份的源码不会被编译
{$ENDIF}
```

我们可以加上$ELSE把一个条件的两种情形做出区隔。

刚刚提到的，使用$IF这个语法是比较有弹性的作法，这个新的语法可以让我们使用到在源码当中的变量或者判断式(举例来说，检查编译器的版本是否高于特定值)。$IF的语法必须以 $IFEND 结尾：

```
{$IF (ProgramVersion > 2.0) }
   … // 如果条件成立，这段程序会被执行
{$ELSE}
   … // 如果条件不成立，则执行这个程序区块
{$IFEND}
```

如果有多个条件的话，也可以加入 $ELSEIF 语法。

编译器版本(Compiler Versions)

每个版本的Delphi编译器都有一个特别的定义值，我们可以依此进行判断，检查我们的程序是否能使用特定版本的编译器。这有赖我们使用了后面介绍的一些功能，但希望在编译时先检查一下编译器是否能够处理这些功能的源码。

如果我们需要最近几版的Delphi来处理特定的源码，我们可以在$IFDEF后面判断以下的版本代号：

Delphi 2007	VER180
Delphi XE	VER220
Delphi XE2	VER230
Delphi XE4	VER250
Delphi XE5	VER260
Delphi XE6	VER270
Delphi XE7	VER280
Delphi XE8	VER290
Delphi 10 Seattle	VER 300
Delphi 10.1 Berlin	VER 310
Delphi 10.2 Tokyo	VER 320
Delphi 10.3 Rio	VER 330
Delphi 10.4 Sydney	VER 340
Delphi 11 Alexandria	VER 350

后面的数字是该版的编译器版本代号（例如26就是Delphi XE5），这个号码并不会只局限在Delphi，可以回推到第一版由Borland所推出的Turbo Pascal编译器(请参考附录A)。

我们也可以在$IF判断式里面使用这些内部的代号常数，这样我们就可以直接用>=来判断编译器是否符合特定版本的需求，版本的常数名称是CompilerVersion，在Delphi XE5里面，这个常数是一个浮点数，数值是26.0，所以范例如下：

```
{$IF CompilerVersion >= 26}}
  // 需要 Delphi XE5 或更新的版本编译器才能编译的源码
{$IFEND}
```

举一反三，我们也可以使用一些系统常数，例如用来判断是哪个操作系统平台，万一我们需要使用到该平台特定的程序功能：

Windows系统（32或64位都一样）	MSWINDOWS
Mac OS X	MACOS
iOS	IOS
Android	Android
Linux	LINUX

以下是简单的程序片段，使用了上述的操作系统定义，它们是HelloPlatform范例的部份程序：

```
{$IFDEF IOS}
  ShowMessage ('Running on iOS');
{$ENDIF}
{$IFDEF ANDROID}
  ShowMessage ('Running on Android');
{$ENDIF}
```

引入檔（Include Files）

我在此想介绍的另一个编译器设定指令，是$INCLUDE这个指令，我们在前面介绍uses的时候已经提到过了，这个指令让我们可以参照、引入特定程序档案中的一部分源码，通常这个用法会被用来在不同的档案中引入相同的源码，例如某段源码定义了一些编译器设定，而我们在使用一个单元时，只需要引入一部分的源码，当我们引入一个档案时，该档案所引入的所有单元都会一起被编译（这就是为什么我们应该避免在引入档里面加入新的识别符号的原因）

换句话说，我们应该不要在引入文件里面加入任何程序相关的元素与定义（这跟C语言的例子正好相反），相关的程序元素与定义都应该在单元文件里面来处理，所以我们到底该如何使用引入档呢？好的例子是在引入档里面写入一些我们希望在大多数的单元文件中都要用到的编译器设定，或者特殊的额外定义。

大型函式库通常会使用引入档来达成前述的目的，例如FireDAC函式库，这是已经成为系统默认函式库之一的用来处理数据库相关的函式库，另一个例子则是系统的运行时间函式库(RunTime Library, 又简写成RTL)也在各个操作系统中使用了独立的引入文件，而在编译器中会随着我们所选择的作业平台单独套用该平台的设定。

02:变量与数据型别

Object Pascal 是一个*强型别*的编程语言，在写程序的时候，变量被宣告时就需要标明其所属的资料型别（也可以是用户自定的数据型别）。所谓的资料型别，所指的是该变量当中可以储存什么样的资料内容，而我们又能对它做些什么运算。这同时让编译器可以用更快的速度处理我们的源码，也更容易发现哪些源码有错误。

这就是 Pascal 语言的型别理论强于其它语言（例如 C 或 C++）的地方，后来发展出的很多语言都在语法承袭了 C 语言，但却打破了与 C 语言的兼容性，例如 Java 跟 C#，就承袭了 C 语言，也学习了 Pascal 对数据型别强固的概念。举例来说，在 C 语言中，连算数型别都几乎可以互相交换。相对之下，原始的 BASIC 语言，就没有这样的概念，而今日的许多脚本式语言，（例如 Javascript 就是最好的例子）对型别的概念跟 Pascal 都是大相径庭的。

笔记

事实上，虽然 Object Pascal 是强型别语言，但仍然有一些技巧可以用来避开型别安全的规范，像是使用可变记录型别(Variant)，这个作法我们强烈建议大家别用，且今日已经很少被使用了。

一如我提过的，所有可变动的编程语言，从 JavaScript 以降，对于数据型别都没有很确切的描述，或是只有很概略性的描述。在这些编程语言当中，变量的型别大多取决于我们赋予什么数据进去。而该变量的型别可能在执行当中随着执行状态而改变。值得正视的是，数据型别在大型的应用程序的编译作业时，是可以用以判断程序正确性的关键，所以不能在运行时间才加以检查。数据型别需要更多的规范与结构，并在源码撰写前就应该要加以规划，这些地方很明显的都有其优缺点。

笔记

我比较喜爱强型别的编程语言，这一点已经不用再多说了，不过本书的目标还是在说明编程语言是怎么运作的，这比提倡我的想法里什么才是一个伟大的编程语言还要来的强烈。

变量与指派数据(Assignments)

跟任何一个强型别的编程语言相同，Object Pascal 也要求所有变量在被使用之前要先行宣告，每次我们宣告一个变量，都必须明白的说明它属于哪种数据型别，以下就是几个变量宣告的样子：

```
var
    Value: Integer;
    IsCorrect: Boolean;
    A, B: Char;
```

上面范例源码里面的 var 关键词，可能在一个源码的好多个地方会出现，例如在程序与函式开头的部份，我们都会宣告一些局部变量，好在程序、函式这些『子程序』当中使用，或者在一个单元文件中宣告整个单元的全局变量。

笔记　我们在后面一点的章节会介绍到，最近几版的 Delphi 已经加入了可以在源码当中随处宣告变量(inline)的功能。这跟古典传统的 Pascal 是很显著的不同。

在 var 这个关键词后面，我们可以宣告一连串的变量名称，以分号来区隔变量名称和其所属的资料型别。我们可以在一行里面输入多个变量名称，例如上面范例中最后的 A 跟 B，多个变量名称之间以逗点来做为区分(早期有一种程序写作风格是把每个变量放在单独一行里面: 用独立的一行也方便于阅读源码、做版本之间的比较，以及源码合并)。

我们定义好一个特定型别的变量之后，就可以用该型别的运算来处理这个变量了，例如我们可以用布尔值来检查条件是否符合，以及把整数值用在数字的运算上。我们不能在未加入特定型别转换函式的情形下把布尔值跟整数拿来混用，也不能把不兼容的各种变量值拿来互相处理（即使它们内部处理的数据兼容也不行，例如布尔值跟整数，它们的数据格式是兼容的，但还是不能把布尔值跟整数混在一起运算）。

最简单的数据指派指令，就是把一个实际的、特定的值指派进该型别的变量里面，例如我们现在有个变量名为 Value，我们希望在 Value 这个变量里面存放 10 这个数字，但是我们要怎么直接撰写实际的值？接下来就让我们来看一下这个实作的方法，以及当中所蕴含的细节。

实际值（文字值、字面意义：Literal Value）

Literal Value 就是在源码当中直接输入一个数据，例如我们需要一个整数 20，就可以在源码当中直接输入：

```
20
```

相同的数值也可以用十六进制来表示：

```
$14
```

从 Delphi 11 开始，我们也可以用二进制表示数值，例如可以写成:

```
%10100;
```

对于数值很大的数字，也是从 Delphi 11 开始，我们可以用底线来做为数值分隔的逗点(西方的数值分隔符, 是每三位数加一个逗点)，作为分隔数值符号的底线，会自动被编译器忽略，例如，我们可以把 2 千万写成:

```
20_000_000
```

这个写法表示是整数型态的 2 千万，如果我们需要表示浮点数的数值的话，就需要在数字结尾加上小数点，例如:

```
20.0
```

在源码的字面意义，不限于数字，我们也可以输入字符跟字符串，字符与字符串的字面意义，只要用单引号包起来即可：

```
//字符
'K'
#55
//字符串
'Marco'
```

如我们上面所介绍的，我们也可以用数值来表示字符（起初这个方式是用来表示 ASCII 字符，但目前已经可以用来表示 Unicode 了），只需在数字前面加一个井字号即可，例如#32 就代表空格符。透过这个方法，我们就可以不用实际输入字符，只要直接透过数值就可以处理许多控制用的字符了，例如 tab 或者删除键。

万一我们在这个字符串当中必须包含单引号这个字符，就直接输入*两个*单引号字符即可：

```
'Marco Cantu''';  //这样就代表了 Marco Cantu'
```

在上面的范例字符串当中，最后三个单引号当中，前面两个重复的单引号用来表示单引号字符，最后一个单引号则是用来表示字符串结束。另外也要注意到，字符串必须要在单行当中输入完毕，如果字符串太长，会超过一行，我们就必须要把它写成两行，然后两行之间用加号 + 来把两个字符串连接在一起。如果我们希望在字符串中加入换行符号，不要把他们写成两行，而是要在字符串当中加入一个换行符号：sLineBreak 这个系统常数：

```
'Marco'+ sLineBreak +' Cantu'''
```

指派叙述句（Assignment Statements）

在 Object Pascal 里面，要进行数据指派的符号，是冒号加上等号(:=)，这个符号对于只熟悉其他语言的开发人员来说，是一个很奇怪的符号。而单纯一个等号 =, 在其他编程语言当中大多用来作为指派数据的运算符，而在 Object Pascal 里面则是用来进行比对两个数据是否相同。

典故

:= 这个符号，是从 Pascal 的前身，Algol 语言来的，这个编程语言对现在的程序开发人员来说都已经是很少听到的，更遑论用过，大多数目前的编程语言都不再使用:=这个符号，而比较常用=了。

也因为用了不同的符号来处理数据指派跟数据比对，Pascal 的编译器(就像 C 的编译器一样)处理源码的速度就更快，因为无需花时间把整段源码判读过后才去判断等号在出现的时候是代表什么意义。而使用不同的符号也让程序人员在阅读源码的时候更容易读懂。Pascal 语言选择的这两个符号跟 C 相关的语言(从 C 取经而来的语言包含有 Java, C#, JavaScript)不同，C 相关的语言是用=作为指派数据的符号，而用==作为判断数据内容是否相同的运算符号。

笔记

为了完整性，我得再多提一下 JavaScript 还有个===符号，这个符号是用来判断两个变量或数据的数据型别与内容是否一样，因为除了判断型别是否相同，还会把两个数据先转成相同型别再去判断其内容，所以这个运算符号所花的时间会比较久，而且就连 JavaScript 的程序人员都常被搞混。

在指派叙述句当中的双方，我们通常称之为左值(lvalue)跟右值(rvalue)，所谓的左值，就是在叙述句的左方的家伙，它必须是个内容可以被改变的识别符号，例如变量、指针、对象，在这个叙述句当中，*右值*的内容会被复制一份到*左值*的内存空间当中。

另一个规则则是*左值*与*右值*的数据型别必须一致，或者两者之间能够有自动的型别转换，这我们会在接下来的篇幅中介绍。

指派（Assignment）与转换（Conversion）

单纯的指派动作中，我们可以写出以下的源码：(您可以在 variableTest 这个测试项目档中，找到这个章节里面的范例源码)

```
Value := 10;
Value := Value + 10;
IsCorrect := True;
```

透过适当的变量宣告，这段范例源码的三个叙述句都是正确的，而以下这个叙述句就是错的了：

```
Value := IsCorrect;
```

当我们输入这段源码的时候，Delphi 的编辑器会立刻在源码的变量下方显示一段红色的曲线，意指源码是有错的。如果我们试着编译上面这句程序叙述句，编译器会回报一个如下所述的错误：

```
[dcc32 Error]: E2010 Incompatible types: 'Integer' and 'Boolean'
```

编译器会告诉我们，在源码当中有错误，错误的描述则是不兼容的资料型别（Incompatible data types），当然，我们也可以把一个变量的型别转换成另一种型别。在某些情形下，这样的转换是自动的，例如我们如果把一个整数值指派给一个浮点数型别的变量，就会自动转换（当然了，反之要把浮点数指派给整数型别的变数则不行）。有时我们也需要呼叫一些特定的系统函式来做数据转换的动作。

为全局变量进行初始化

我们可以在宣告全局变量的时候顺便把初始值指派给它，用的是定义常数内容时候的符号(=)，而不是指派变量内容时所使用的(:=)，所以我们可以写成以下范例源码：

```
var
    Value: Integer = 10;
    Correct: Boolean = True;
```

为局部变量进行初始化

这样的初始化动作，只能对全局变量使用，在子程序，也就是程序或者函式当中的变量宣告，是不能够使用这个语法的，子程序的变量初始化，必须在子程序的源码当中进行：

```
var
    Value: Integer;
begin
    Value := 0; // 初始化
```

我要再强调个 100 次、10000 次，如果局部变量不经初始化就直接使用它，变量里面的内容是完全无法预料的，它将会是内存被变量取得的当下，内存里头的随机数据，在许多情形下，编译器都会警告说这会有潜在的错误发生。例如我们如果这么写：

```
var
    Value: Integer;
begin
    ShowMessage (Value.ToString); // Value 就没有初始化
```

编译器会警告说 Value 这个变量没有被初始化，而如果我们执行这个程序，会发现执行的结果中，显示的内容完全无法预料，因为显示的内容将会是内存被配置成 Value 这个变量时，该段内存里面的随机数据。

行内变量（在源码中宣告的变量 Inline Variable）

在最近几版的 Delphi 里面(是从 10.3 Rio 开始的)加入了一个新的概念，这个概念改变了早期 Pascal 跟 Turbo Pascal 编译器开始的变量宣告概念:在源码中宣告变量，我把它称之为行内变数 (inline variable declarations)。

这个新的行内变量语法，让我们可以直接在任意的源码区块当中宣告变量(可以用传统的变量宣告方式，在一行里面宣告多个变量):

```
procedure Test;
begin
    var I, J: Integer;
    I := 22;
    J := I + 20;
    ShowMessage(J.ToString);
end;
```

虽然表面上看起来，差别并不大，但是这个语法的影响所及，牵涉到初始化、型别推定(type inference)，以及变量的生命周期，本章的后段我会再详述这些部分。

为行内变数初始化

跟传统的宣告模式相比，第一个显著的改变就是在源码里面宣告跟为变量初始化可以用一个叙述句就完成。相较之下，这会比把变量都宣告在函式起始的区块容易阅读，我们接着用前一节的范例来看:

```
procedure Test;
begin
    var I : Integer := 22;
    ShowMessage(I.ToString);
end;
```

这么写的好处，就是当我们要连续宣告几个变量时，可以直接用上前一个变量，不用再写常数(例如 0 或者 nil)，也可以直接用上变量计算结果作为另一个变量的初始值了，例如底下这个例子里面的 K:

```
procedure Test1;
begin
    var I : Integer := 22;
    var J : Integer := 22 + I;
    var K : Integer := I + J;
    ShowMessage(K.ToString);
end;
```

从另一个角度来看，传统的宣告方式中，只要是在变量宣告区里面宣告了的变量，在整个函式里面都可以使用。但新的行内变量写法，是只有在该变量宣告的位置之后的源码，才可以使用该变量喔。用上面这个例子来对照，变量 K 在前面两行是不能使用的，在宣告前使用该变量，编译过程就会出错了。

行内变数的型别推定

另一个行内变量的明显好处，就是从现在起，编译器可以从几个不同的角度，例如初始值的型别，来推定这个行内变数的型别。底下是个简单的范例:

```
procedure Test;
begin
```

```
    var I := 22;
    ShowMessage(I.ToString);
end;
```

右值的算式型别(就是在 := 符号右边的内容)可以算出来决定该变量的型别。如果指派了一个字符串常数，该变量很单纯的，就会被定为字符串型别，但如果我们指派了函式的结果或者复杂的算式，这个型别推定就会由编译器来处理了。

要记得，这个变量仍旧是强型别，当编译器在编译阶段决定了该变量的型别，并且做了初始化，我们就不能在后面的源码里面试图指派不同型别的数据到该变量里面了。型别推定只是让我们偷个懒，可以不要写型别而已(例如复杂的变异数据型别，或者是泛型型别)，但并没有改变这个语言强型别的基础，也不会让它在运行时间因而变慢。

笔记 在进行型别推定的时候，有时数据型别会被『扩展』为比较大的型别，例如前面的例子里，数字型别的值 22 (就是一个 ShortInt)，会被推定为 Integer。这会作为通用规则，当右值是整数型别，并且小于 32 bits 的范围时，该变量会被推定宣告为 32 bit 整数，我们也可以自行指定特定的、范围较小的值类型。

常数(常量，Constants)

Object Pascal 也允许我们宣告常数，常数的存在，让我们可以把特定的数据赋予名称，在程序执行的过程中，常数的内容是不能被变更的。同时常数的存在，也会让我们的源码在编译之后的 Size 小一点。

要宣告常数，我们可以不用指定资料型别，只要直接指定初始值就行了，编译器会由我们指定的初始值内容，自动辨识该内容应该使用的数据型别，以下就是一些简单的常数宣告（这些源码仍然可以从 VariablesTest 项目中找到）：

```
const
Thousand = 1000;
Pi = 3.14;
AuthorName = 'Marco Cantu';
```

编译器会依照数据的内容自动判断常数的数据型别，上述的源码中 Thousand 这个常数的型别会被指派为 SmallInt，这是所有整数型别中占用最小空间就能储存 1000 这个数字的。如果我们要告诉编译器，我们要用特定的型别来储存该数值，那就可以改写为如下的源码：

```
const
Thousand: Integer = 1000;
```

警示 『应该会这样吧』的这种梦想，在程序里面通常都不会如你的意，尤其在编译器更新过几个版本之后，通常这种程序人员没来由的梦想都无法如意。所以如果你能设法让源码看起来更清楚，不要有假设、如果这种幻想参杂在其中的话，就那么做吧，让源码永远看起来能够明明白白，不要有任何幻想、灰色地带的机会！

当我们宣告了一个常数，编译器会决定要把这个数据储存在内存的哪个地方，或者在每次使用到这个常数的时候，把内容复制过去。第二个目标在处理简单的数据内容时尤其有用。

一旦我们宣告了常数，我们就可以把常数像其他变量一样使用，只是我们不能企图变更常数里面的数据，如果我们在指派叙述句里面把常数放在左值的位置，编译器就会回报错误。

笔记 够奇怪的了，Object Pascal 居然允许我们在运行时间变更常数的内容，等于把常数当成变量来用，只是我们必须先设定$J 这个编译器设定，或者透过适当的编译器选项，名为 Assigned typed constant。这个选项的规则是为了兼容于旧版的编译器，这作法很明显的，不是我们建议的程序风格，而我在这里介绍这个功能，也只是为了完整的呈现 Pascal 的历史，以及曾经有过这样的程序技术而已。

行内常数（在源码中宣告常数、常量 Inline Constants）

我们刚刚介绍过行内变量，我们也可以在源码里面宣告行内常数。用这个方式宣告的常数，可以宣告其型别或者省略类型声明，在省略型别的常数宣告时，其型别也是用推定的方式定义出来的(这个推定的作法在处理常数的宣告早已行之有年)。我们来看个简单的范例：

```
begin
    // 某些源码...
    const M: Integer = (L +H) div 2; //把型别透过标识符清楚定义
    // 又来一些源码
```

```
const M = (L + H) div 2;  // 没有写特定型别的常数宣告
```

资源字符串常数（Resource String Constants）

虽然这是一个稍微进阶的主题，当我们定义一个字符串常数，而不是写一个标准的常数宣告，我们可以使用特定宣告方法：资源字符串常数（resourcestring），它会告诉编译器和链接程序要对待该字符串常数如同对待 Windows 的资源一样。（或在非 Windows 平台上 Object Pascal 对等的数据结构）：

```
const
    sAuthorName = 'Marco';
resourcestring
    strAuthorName = 'Marco';
begin
    ShowMessage (strAuthorname);
```

在上述的两个宣告中，我们都是宣告一个常数，在程序执行过程中不能变更里面的内容，差别只在于程序内部是怎么实作而已。透过 resourcestring 区段定义的常数都会被储存在程序的资源字符串表里面，这个资源字符串表可以被一些外部程序编辑，例如作多国语言在地化的程序(可以把当中的字符串编辑成各个不同语系的内容)。

使用资源的好处就是在 Windows 应用程序处理内存的时候可以节省更多空间，在其他平台也有类似的机制，这个方法也是在处理多国语系应用程序时比较好的作法，源码的内容就不用做任何修改了。实际执行时，我们需要把所有在程序当中要显示的文字全部都以 resourcestring 来宣告，这样一来所有的文字都会被以 Windows 资源字符串表被储存起来，我们只需要用 Windows 资源编辑工具，把资源字符串表的内容进行翻译、另存成另一语系的资源文件，那么当使用者在不同语系的 Windows 上面执行这个程序的时候，就会加载该语系对应的资源字符串表，也就会用该语系的文字内容来显示程序接口跟讯息，应用程序的文件名也无须修改了。

提示 IDE 环境的编辑器有自动重构(refactoring)的功能，我们可以用它来把我们源码里面的字符串取代成对应的 resourcestring 宣告。只要把编辑光标移到我们要修改的字符串标识符上，然后按 Ctrl+Shift+L 就可以启动重构功能了。

变量的生命周期(Lifetime)与可视范围(Visibility)

依照我们定义一个变量的方式，变量使用内存的空间与时间都会不一样(通常这段时间我们称为变量的生命周期)，同时也会在源码当中不同的地方对该变量有可辨识与不可辨识的分别（这个界限我们称之为变量的可视范围）。

在这么前面的章节里，我们还没办法完整的把所有可能的选项都介绍完，但我们可以先试想最常用的几种案例：

- **全局变量**： 如果我们在一个单元的interface部分宣告了一个变量（或者其他识别符号，例如类别、常数），这个变量的可视范围将会扩及所有参照(或叫使用)了这个单元的所有单元源码。程序一开始执行后，就会尽快为这个变量配置内存空间，直到程序结束为止都维持这个内存空间的存在。我们可以在单元的initialization区段指派一个初始值给这个变数。

- **全局隐藏变数**：如果我们在一个单元的implementation区段宣告一个变量，则我们在整个程序的其他单元是无法看到这个变量的，但在这个宣告变量的单元中的任何一个函式、方法、程序当中则都可以使用这个变量，这样的变量使用了全局的内存，并且与刚介绍过的全局变量拥有相同的生命周期，相异者只有可视范围，初始化的方法也跟全局变量相同。

- **局部变量**：如果我们在任何一个函式、方法、程序当中宣告一个变量，我们只能在该段子程序当中使用这个变量，一离开该段子程序，就无法再存取这个变量了。这个变量的可视范围涵盖了该段子程序，以及其下的其他所有子程序（除非其下的子程序当中有用到相同名称作为其他识别符号之用）。在内存当中，这个变量是在该段子程序被呼叫的时候由操作系统配置空间给当时所在的内存堆栈里面，而该段子程序执行完成时，配置给这个变量的空间也会自动被回收给系统。

- **区域行内变量**：如果我们在函式、方法或程序的程序区块中宣告了一个行内变量，跟传统局部变量相较，行内变量只多了一个限制，就是该变量的生命周期是从它被宣告的那行开始，一直到该程序区块结束的地方为止。

- **程序区块内的行内变量**：如果我们是在特定程序区块(例如在程序区块中又另外写了 begin-end 这样的区块，例如 if-then 或者 while begin-end 这样的区块)，则该变量的生命周期就只局限在该区块里面。这个概念在很多早已能够在任何程序区块中宣告变量的编程语言来说早已是见怪不怪，但对于 Object Pascal 来说，这是最近几个版本才加入的概念。请特别注意，在多重 begin-end 程序区块中宣告的行内变量，只能在该阶层的区块中被使用，离开该层区块，到外层就不能使用了。

笔记 对程序区块内的行内变量来说，受限的不仅是该变数能否被看见，还有该变数的生命周期。受管理的数据型别(例如 interface, string 或 managed record)都会在该程序区块结束时就被自动释放，而不会等到该函式、程序或方法结束时才释放。这个原则也适用于暂存计算结果的暂时变量喔。

任何在单元的interface区段当中宣告的识别符号，在整个程序中都是可视的，只要该单元有被其他单元使用到。画面窗体类别的变量就是用这个方法宣告的，所以我们可以在整个程序的任何其他画面窗体中，随时取用任何一个画面窗体（当然也包含其下的public范围的方法、属性、以及组件）。把所有东西都宣告成全局可视，绝对是很糟的程序技巧。除了会占用内存空间之外，使用全局变量也会使一个程序项目很难维护、更新下去，换句话说，我们在使用全局变量的时候，应该要尽量精简也要避免使用Delphi建立的全局变量，例如窗体。

资料型别

在Pascal当中，有一些数据型别是预先定义好的，我们可以将之区分为三个组别：有序型别(ordinal types)、实数型别，以及字符串。我们会在接下来的篇幅当中讨论前两个，然后在第六章里面特别介绍字符串。

Delphi也包含了*未特定型别*的资料型别，称之为variant，还有另一些有弹性的数据型别，像是TValue (是进阶的RTTI功能的一部分，RTTI是RunTime Type Information，*运行时间型别信息*的缩写)。这类进阶的数据型别我们会在稍后第五章里面加以介绍。

有序与数值类型别

有序型别是根据顺序的概念而来的，我们不只能够算出两个资料之间的大小，而且还可以得知数据的下一个或前一个数据是什么，当然也能够算出该数据的最大值与最小值。

这组类别当中，最重要的三个型别分别是：Integer(整数), Boolean(布尔值), 以及Char(字符)。当然，还有其他相同意义的型别，与这三个相异者只在内部实作的方式，以及可表示的数值范围。

以下的表格列出了有序型别可表示的数字范围：

大小	有正负号之分	无正负号之分
8 bits	ShortInt: -128到127	Byte:0到255
16 bits	SmallInt:-32768到32767 (-32K到32K)	Word:0到65535 (0到64K)
32 bits	Integer: -2,147,483,648 到 2,147,483,647 (-2GB到正2GB)	Cardinal:0 到 4,294,967,295 (0到4GB)
64 bits	Int64:-9,223,372,036,854,775,808 到9,223,372,036,854,775,807	Uint64:0 到 18,446,744,073,709,551,615 (我已经无法读出这个数字了)

我们可以发现，这些数据型别各有其可表示的数字范围，端视其所使用的内存空间大小，以及其表示的数字是否包含正负号。要表示负数的型别，就会比使用同样空间但不表示负数的型别少了一倍的数字可以表示。

例如Int64这个型别可以表示十进制的18位数数字，函式库中对于这个型别也支持有序类别的一些子程序(例如High跟Low)，以及用来计算的子程序（像是Inc跟Dec），以及字符串转换的子程序(例如IntToStr)。

整数类型型别的别名

有时候我们会很难记住ShortInt跟SmartInt之间的差异，例如到底哪一个型别可储存的数字比较小，这时候我们就可以参考在System这个单元当中预先定义的类别别名：

```
type
Int8 = ShortInt;
Int16 = SmallInt;
Int32 = Integer;
UInt8 = Byte;
UInt16 = Word;
UInt32 = Cardinal;
```

请记得，这些型别并没有新增新的型别，而只是为了容易记忆，例如Int16当然比SmallInt更容易从字面上知道它比ShortInt(也就是Int8)来得大。这些型别别名也让从C或其他编程语言转换过来Object Pascal的程序开发人员更容易上手。

整数型别, 64 位与 NativeInt

在64位版本的Object Pascal里面，我们会觉得很惊讶，因为它的整数型别仍然还是32位。这么做的原因，是因为这样对于在CPU层级的数字运算最有效率。

指标型别（我们在稍后的篇幅当中会加以介绍）与其他与内存地址相关的数据型别都是64位，如果我们需要一个值类型转换成指标的大小，并想要符合CPU原生平台使用，我们可以使用两个特别的型别：NativeInt跟NativeInt的别名类别。这两个类别会在特定平台中保持相同的长度，在32位系统中，他们就是32位，在64位系统中，就会变成64位。

但有个很显著的特殊案例：LargeInt 型别，这个型别通常用来对应原生平台的 API 函式。它在Windows 32位版以及其他32位的平台中都是32位，而在64位ARM平台时就是64位。所以除非您需要特殊处理特定原生源码让您的程序能套用在该平台上，不然最好不要随便碰触这个议题啊。

整数型别助手

整数型别在Object Pascal里面是享有特殊待遇的型别，我们可以在处理整数变量，甚至于整数型别的常数时，在识别符号后面加上一个. 就会有跟其他类别相似的法可以使用喔。

笔记 从技术上来说，对原生数据型别的这些操作，都是被定义为要使用『内建的纪录助手』（intrinsic record helpers）。类别与记录的助手会在第 12 章介绍，简单的说，我们可以对核心的数据型别自定操作方法。有经验的开发人员会发现到，型别的操作方法会被定义为类别的固定方法，以符合内建记录助手。

我们可以在下面的范例当中看到许多从IntegersTest范例项目撷取出来的实例：

```
var
   N: Integer;
begin
    N := 10;
    Show (N.ToString);
    // display a constant
```

```
Show (33.ToString);
// type operation, show the bytes required to store the type
Show (Integer.Size.ToString);
```

笔记 上面的范例中，show 这个函式是用来把数据显示在一个 TMemo 组件里面的简单函式，使用这个函式，我们可以先避免太靠近 ShowMessage 对话框，而顺带的好处，则是我们可以把执行的结果从 TMemo 组件上直接复制下来，如下面的执行结果所示，在本书中的大多数范例也都会用这个方式来处理执行结果。

上面的源码执行结果如下：

```
10
33
4
```

这些操作方法的确很重要(比其他被列在运行时间函式库当中的更重要)，所以值得被我们列在这里：

ToString	把数值转成字符串，以十进制表示
ToBoolean	把数值转换成布尔型别
ToHexString	把数值转换成字符串，以十六进制表示
ToSingle	把数值转换成单精度浮点数型别
ToDouble	把数值转换成倍精度浮点数型别
ToExtended	把数值转成extend浮点数型别

第一和第三个操作方法会把数值转换成字符串，分别使用十进制与十六进制格式来显示，第二个操作方法则是转换为布尔型别，后面三个则是把数值转成浮点数型别，这个型别我们稍后会介绍。

整数型别还有一些操作方法我们可以使用的（大多数其他值类型也有），例如：

Size	该型别使用多少个字节来存放数据
Parse	把一个字符串转换成字面上显示的数值，如果该字符串并不是数值，就会抛出一个型别转换例外事件。

TryParse	试着把字符串转换成数值，如果字符串不是合法的数字，则会回传0

标准有序型别函式

除了由整数型别助手定义的方法，以及前列的这些方法，还有一些标准、古典的函式我们可以用来处理任何有序型别的（不一定只有数值类的型别）。最简单的例子，就是查询该型别的大小(SizeOf)、最大值(High)、最小值(Low)。SizeOf这个系统函式回传的值（我们可以在任何型别上使用这个函式），会是一个整数，告诉我们这个型别会使用多少个字节来储存数据（很像前面提到的Size这个方法所做的）

系统内建了一些可以用来处理有序型别的函式，我们以下表列出：

Dec	把参数一的数据递减，如果有参数二，则把参数一递减参数二所述的数值
Inc	把参数一的数据递增，如果有参数二，则把参数一递增参数二所述的数值
Odd	回传参数数据是否为奇数，如果要测试是否为偶数，请直接加个 not 在前面做反向运算即可：not Odd
Pred	回传参数数据的前一个数值，系统会依据参数型别来判定前一个数值应该是什么
Succ	回传参数数据的后一个数值，系统会依据参数型别来判定后一个数值应该是什么
Ord	回传参数的数值在该型别中的序列编号（通常用在非数字型的有序型别）
Low	回传参数数据的型别中的最小值
High	回传参数数据的型别中的最大值

笔记 C 和 C++语言的程序人员应该要注意，Inc 函式这里有两种版本，有一个参数的版本，也有两个参数的，分别对应 C 语言语法里面的++和+=(Dec 当然也有两种版本，分别对应--和-=)，Object Pascal 编译器会对这些函式进行优

化，就像 C 和 C++编译器所做的一样。然而，跟 C/C++的功能不同，Delphi 提供的这两个函式，只能提供处理前先计算(先加或者先减)的功能，无法提供处理后再计算的功能，并且也没有提供回传值。

请注意，这些函式当中，有些是由编译器自动计算好，然后直接替换掉其中的数值的，例如，如果我们呼叫了High(X)，这里的X是一个整数型别的变量，编译器会自动把该段源码直接替换成整数型别的最大值。

在IntegersTest范例中，我已经为一些有序型别函式加入了一个事件：

```
var
  N: UInt16;
begin
   N := Low (UInt16);
   Inc (N);
   Show (IntToStr (N));
   Inc (N, 10);
   Show (IntToStr (N));
   if Odd (N) then
      Show (IntToStr (N) + ' is odd');
```

执行结果如下：

```
1
11
11 is odd
```

您可以把上述源码里面的Uint16修改成Integer或者任何其他的有序型别，观察看看结果会有什么改变。

超过范围的运算

一个变量都有其有效范围的数值，就像上面这个范例里面的N一样。如果我们指派了太大的数值，或者一个负数给它，上面的执行结果就会是错误的，事实上一共有三种不同的错误可能会发生在这种超过范围的运算当中。

第一种是编译错误，会发生在我们指派一个超过范围的常数数值时，例如我们如果在上面的范例里面加入这行源码：

```
N := 100 + High(N)
```

编译器就会回报错误如下：

```
[dcc32 Error] E1012 Constant expression violates subrange bounds
```

第二种情况，则是编译器不会回报错误，因为还得看程序执行的时候会不会导致错误，假设我们在上面的程序片段中改写成：

```
Inc (n, High(n));
Show(IntToStr(n));
```

编译器不会在编译的时候告诉我们源码有错，因为编译器也无法在程序执行前预先知道错误可能发生（因为会不会有错，得看N这个变量的初始值才知道）。万一有两种可能性：假如我们编译了，并执行了这个程序，我们最后指派了不合逻辑数值给这个变量，则这行程序的动作将会是减法，这也是最糟的情形，我们不会得到错误，但这个程序也不正确。

我们能做的（也建议大家这么做）就是把编译器选项当中的Overflow checking（溢位侦测）这个选项设成打开({$Q+} 或者 {$OVERFLOWCHECKS ON})，这样一来编译器就会对类似的溢位错误进行防御并加以阻拦、抛出名为”Integer overflow”的错误。

布尔值

逻辑上的是(True)与非(False)被以布尔型别加以表示，这个型别也用来作为条件判断叙述句，我们在下一章就会介绍，布尔值只有两种可能的数值内容，也就是True或者False。

警示 为了跟微软的 COM 以及 OLE automation 兼容，ByteBool, WordBool, 以及 LongBool 都以-1 表示 True, 以 0 表示 False。然而我们应该可以忽略这三种特殊的型别，也避免直接用系统定义的数字来表示布尔值，除非绝对必要的情形。

跟C语言或其他C阵营的语言不同，布尔值在Object Pascal当中是以枚举数据类型来实作的，所以没有其他数值可以直接被用来表示布尔值。而且我们绝对要避免企图把布尔值转换为其他值类型，即使布尔值的型别助手当中也包含了ToInteger跟ToString这两个方法。我会在本章稍后的篇幅当中介绍枚举数据类型。

请注意，布尔值的ToString方法会把布尔值的内容以数值回传，就像我们用另一个函式BoolToStr的结果一样，当我们把第二个参数的内容设成True，

来表示要在输出中以字符串(True或者False)来回传(请见下面介绍字符型别操作的范例程序)

字符

字符变量是以字符型别来定义的，今日的字符型别已经是以双字节的Unicode字符(也可以用WideChar型别来表示)来储存资料了，和旧版的单字节是完全不同的。

笔记 Delphi 编译器仍然提供了单字节的字符型别，以 AnsiChar 这个型别来储存单位元的 ANSI 字符，而用 wideChar 来储存 Unicode 字符，Char 型别的定义则只是一个型别别名，我们建议只要是处理字符，都一律使用 wideChar，如果要处理单字节的数据，别再像以往很多人会用 Char 来处理，现在请改用 Bytes 这个类别吧。然而从 Delphi 10.4 开始，所有版本的编译器又都支持 AnsiChar 了，这是为了对目前已经存在的源码有更高的兼容性。

如果您需要关于Unicode字符的定义，包含字符代号与显示关系（当中有不少深入的主题），请阅读第六章，而在目前这个章节中，我将只聚焦于字符类型的核心概念。

一如稍早在介绍实际值的时候我所提到的，常数字元可以用文字符号直接来表示，像'k'这个符号，也可以用数值来表示，写作#78。字符也可以透过Chr这个系统函式来表示，例如Chr(78)，而相对的要把文字内容以数值来表示，则要透过Ord这个函式，通常直接用文字符号来表示字符、数字、标点符号是比较好的。

当我们在使用特殊字符，例如在#32以下的控制字符，我们也只能用数值来表示，以下的列表包含了最常使用的几个控制字符：

#8	删除键
#9	tab定位键
#10	换行符号
#13	回到行首
#27	esc键

字符型别的运算

跟其他的有序数据型别一样，字符型别也有几个原生的运算方法，可以让我们透过在该型别的变量之后加个.就能直接使用，也就是透过内建记录助手来处理。

然而，使用的情境已经相当不同，首先，我们必须在要使用这个功能的单元当中先use Character这个单元，跟其他的转换函式不同，字符型别的助手包含了一二十个跟Unicode专属的处理函式，例如IsLetter, IsNumber以及IsPunctuation这几个查验用的函式，以及ToUpper, ToLower这两个转换用的函式，以下是从CharsTest范例当中节录的一些源码：

```
uses
  Character;
...
var
    ch: Char;
begin
     ch := 'a';
     Show (BoolToStr(ch.IsLetter, True));
     Show (ch.ToUpper);
```

这段源码的的结果会是：

```
True
A
```

笔记 字符型别助手当中的 ToUpper 处理函式，已经完全支持 Unicode 了，这表示如果我们传了 Unicode 字符的 ù 就会回传大写的 Ù，部分传统的运行时间函式库当中的函式就没有这么聪明，而只能处理一般的 ASCII 字符而已。目前还没能支持这么多，因为对应的 Unicode 处理速度还很慢。

把 Char 视为有序型别

Char型别涵盖的范围相当大，但它仍旧是有序型别，所以我们就可以对Char型别使用Inc, Dec这样的函式来取得前一个、下一个字符，就像我们在前面介绍基础有序型别的章节所介绍的那样。也可以写个for循环，把Char当成里面用来记录顺序的元素。

底下是一个简单的范例，用for循环来显示一些字符，以开始值到结束值的数字作为要显示的字符数值：

```
var
ch: Char;
    str1: string;
begin
    ch := 'a';
    Show (ch);
    Inc (ch, 100);
    Show (ch);

    str1 := '';
    for ch := #32 to #1024 do
       str1 := str1 + ch;
    Show (str1)
```

在CharsTest范例的 for 循环当中，把很多文字加入了字符串中，使得执行结果相当的长，这个字符串是以底下的文字开头的：

```
a
Å
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abc
defghijklmnopqrstuvwxyz{|}~
```

以 Chr 函式进行转换

我们之前已经介绍过Ord这个函式，可以把字符的数值数据回传(更精确的说，是回传Unicode的字符码节点)，同样的我们也有另一个函式Chr可以把数值转换成字符。

32 位的字符

虽然目前默认的字符型别已经对应成WideChar，我们还是需要知道Delphi也同时提供了一种使用4个字节来定义字符数据的型别UCS4Char，这是Universal Character Set 4位的缩写。在System单元当中，定义如下：

```
type
    UCS4Char = type LongWord;
```

和UCS4Char对应的字符串型别是UCS4String(定义成UCS4Char的数组)，虽然这组型别不常用到，但一样定义在Character单元里，也是这个编程语言运行时间函式库的一部分。

浮点数型别

不同型态的整数能够表示有序的数据集合，而浮点数则是不具顺序的（虽然可以进行数值大小的比较，但却不具顺序，因为无从决定下一个数值与目前的数值要差距多少），浮点数型别可以用来表示一些数字的近似值，依据这些型别的表示范围，会多少有些误差。

浮点数依照不同型别使用不同长度记录数据，而有不同的格式，以下是在Object Pascal里面浮点数型别的列表：

Single	浮点数里面使用空间最小的就是Single，这个型别使用4个字节来储存数值，Single这个名字，正意指着它使用单精度浮点数，这个型别也正对应了其他语言里面的float型别
Double	使用Double的型别，就会使用8个字节来储存数据，Double这个名字指的就是它使用倍精度浮点数来储存数据，其他语言中也有使用Double这个型别，其他语言的Double也是使用8个字节来储存数据，在旧版的Pascal当中，这个型别则是被称为Real。
Extended	这个型别在原本的Delphi Win32版的编译器中，是使用10个字节来储存数据，但这个型别并不是所有操作系统都能使用的(在部分的操作系统上，例如Win64，就被转回了Double，在macOS上面它则是16个字节)。其他的语言则称这个数据型别为long Double

以上就是所有不同精确度的浮点数型别，分别对应了IEEE定义的标准浮点数表示方法，且CPU都可以直接支持，指令周期也最快（精确的说，应该是FPU，也就是浮点运算器才对）

另外还有两个很特别的不具顺序的数字型别，我们可以用来记录精确，而非近似值的型别：

Comp	使用8个字节来表示非常大的整数（可以记录到18位数的十进制数），由于是整数，透过这个型别来记录，就可以记录完全准确的数值，不像浮点数只能记录近似值。
Currency	同样使用8个字节来记录数字，但固定使用十进制的

四位数来记录小数点数值。跟型别的名字一样，Currency型别是用来处理货币数值的，必须精确到小数点以下四位数，而且不能有所误差。

笔记 Delphi 11 当中为 Currency 这个型别新增了新的记录助手。跟后面的篇幅中会介绍到的浮点数型别的助手类似，都提供了取整数的功能、转换为字符串，以及从字符串转换为浮点数的功能。

所有不具顺序的型别，都没有High, Low, Ord这些函数，因为实数型别在理论上，是可以用来表示不具上限的数值的，有序型别只能储存一定数量的数据，也就有了边界值。

为何浮点数的数值特别不同

我们来深入讨论一下，当我们观察整数23的时候，我们可以确切的知道下一个数字是多少，整数是有限的，因为整数是有序的数值，每次向前或向后的变化量都是整数1。

而浮点数则是用来表示连续的数字，即使只是表示很小的区段，请想象一下，浮点数，也就是实数，就像是一条线，是由无限个点组成的，例如在23到24之间到底有几个数？或者说23.46的下一个数值是多少？是23.47?23.461?或23.4601?由于实数是连续的，所以是完全无法得知的。

同样的道理，我们可以知道char型别数值中，w的下一个字符是什么，但完全无法得知7143.1562的下一个数字是什么？我们当然可以判别两个实数之间孰大孰小，但无法从一个实数推测知道它的下一个数值是什么。

另一个对于浮点数数值的关键概念，则是浮点数无法精确记录所有数字的数值，通常有些数字只能记录其近似值，例如从范例FloatTest摘录的以下源码：

```
var
   s1: Single;
begin
         s1 := 0.5 * 0.2;
         Show (s1.ToString);
```

以上的程序执行后，我们可能会预期屏幕上出现的结果是0.1, 但事实上它可能会出现0.100000001490116这个接近0.1的数字，但结果仍旧不是0.1，当然，如果我们对小数二位作四舍五入，结果就会是0.1，如果我们把范例中的s1型别改成Double,输出结果就会变成0.1，FloatTest就会这样显示。

笔记 目前我们还没有时间作对浮点数在计算机的处理进行数学上的深入讨论，所以我会再这里暂时不再作更多的说明，如果您对于 Object Pascal 语言的这方面处理有兴趣，我推荐给您 Rudy Velthuis 的精辟文章，您可以从这个网址看到：http://rvelthuis.de/articles/articles-floats.html.

浮点数类别助手与 Math 单元

从前一个程序范例可以看出，浮点数资料型别也提供了类别助手，我们可以透过助手直接对特定变量的转换作处理，就像把它当成一个类别实体一样。事实上，对浮点数型别的处理可是相当多种的。

以下是对Single型别的一些处理(部分函式可以从它们的名称看得出来是做什么处理，其他的函式可能就得看一下文件才知道是什么功能了)：

Exponent	Fraction	Mantissa
Sign	Exp	Frac
SpecialType	BuildUp	ToString
IsNan	IsInfinity	IsNegativeInfinity
IsPositiveInfinity	Bytes	Words

运行时间函式库也提供了一个名为Math的单元，在这里面定义了许多进阶的数学函式，包含了三角函数函式（例如ArcCosh），财务用的函式（例如InterestPayment），以及统计用的函式（例如MeanAndStdDev程序）。里面包含了许多数学函式，很多我们可能都没听过，例如MomentSkewKurtosis（我想让大家去搜寻一下这个函式的用途）

System.Math这个单元在功能上已经非常多元了，但我们还是可以找到许多额外的Object Pascal的数学函式库。

简单的使用者自定型别

随着类型的概念，由Nikalaus Wirth带入Pascal语言的伟大想法之一，就是在程序中定义新数据型别的能力(很多功能我们今天早已觉得理所当然，但在当时大家并还不这么想)。我们可以透过型别定义，在程序中引入我们自定

的数据型别，例如次范围型别、数组型别、记录型别、列举型别、指针型别，以及集合型别，最重要的使用者自定型别就是类别（Class），这是Pascal这个编程语言面向对象能力的一部分，将在本书的第二部分介绍。

您或许会觉得型别定义在许多编程语言里面都是很常见的，没有错，但Pascal是第一个正式以精确定义的方式把它作为一个基本功能的编程语言。Object Pascal仍有一些很独特的功能，例如次范围的定义、列举功能、集合等，将在接下来的章节里介绍。更复杂的数据型别（像数组或记录）则在第五章进行介绍。

命名与非命名型别

我们可以为使用者自定型别取个名字，好在后面的源码里面使用，或者直接用在变量上面。在Object Pascal的惯例中，是在任何数据型别的名称前面加个T开头，包含类别也是如此，但这规则并不只限于使用在类别上。我会强烈的建议您遵守这个规则，如果您原来是习惯写JAVA或C#的程序，或许您会不太习惯，但还是建议您遵守它。

当您为一个数据型别命名时，您必须在源码的type区段当中宣告（我们可以在每个单元当中加入数据型别，不限数量），以下则是一个简单的程序片段，示范宣告几个新的数据型别：

```
type
  // subrange definition
  TUppercase = 'A'..'Z';
  // enumerated type definition
  TMyColor = (Red, Yellow, Green, Cyan, Blue, Violet);
  // set definition
  TColorPalette = set of TMyColor;
```

有了这些数据型别，我们就可以宣告一些变量：

```
var
        UpSet: TUpperLetters;
        Color1: TMyColor;
```

在上述的情境中，我们使用了命名型别，在另一种情形下，也可以不为这个新的数据型别命名，如下面的源码所示：

```
var
        Palette: set of TMyColor;
```

在一般的情形下，我们应该避免使用非命名的数据型别，就像上面这个范例中所写的，因为这样写，我们没办法把它作为其他函式或者程序的参数。由于这个语言最后在进行型别比对的时候，会直接以型别名称来比对，而不会用型别当中的每个字段定义一一比对，所以为每个数据型别赋予一个名称就格外的重要。也要记得在一个单元文件的interface区段所做的型别定义，将会让所有使用该单元的其他源码都可以辨识它。

上面的型别定义是什么意思呢？我会用一些篇幅为不熟悉传统Pascal型别定义宣告的读者介绍，也会试着把Pascal的型别定义跟其他语言不同的地方特别点出来，所以大家不论如何，都可能对以下的篇幅感到有兴趣。

型别别名

如同我们所见的，Delphi语言在判别型别之间是否兼容时，是比对型别的名称是否相同，而非实际比对两个型别的真实定义是否相同。如果两个定义完全相同的型别用了两个不同的名称来命名的话，就会被视为不同的型别。

当我们使用型别别名的时候，这个情况也有部分是成立的，对系统来说，用了别名的新型别，就会被视为是全新的型别。令人迷惑的是两个相同的语法会产生相当不同的效应。让我们看一下在TypeAlias范例里面的源码:

```
type
    TNewInt = Integer;
    TNewInt2 = type Integer;
```

这两个新的型别都能跟原始的Integer型别兼容(透过自动型别转换功能)，然而 TNewInt2 型别却无法让编译器视为与 TNewInt 相同的型别，例如在函式当中的引用参数中，就无法把 TNewInt2 的变量传给 TNewInt 的字段：

```
procedure TForm40.Button1Click(Sender: TObject);
var
   I : Integer;
   NI: TNewInt;
   NI2: TNewInt2;
begin
    I := 10;
    NI := I;  //没问题
    NI2 := I;  //也没问题
```

```
Test(I);
Test(NI);
Test(NI2);  //会报错
```

最后一行的源码会报以下的错误讯息:

```
E2033 Types of actual and formal var parameters must be identical  (var 的参数域类型必须一致)
```

类似的状况也发生在型别助手，Integer型别助手可以用在 TNewInt 上，但不能用在 TNewInt2 上面，这情况我们稍后会在介绍记录助手的时候再深入介绍。

次范围型别

次范围型别是把特定范围的数据特别取个名字，所以称为次范围型别。举例来说，我们可以把一个小范围的整数，例如1到10，或者100到1000定为一个新的型别，或者也可以定义英文字母的大写部分成一个新的型别：

```
type
        TTen = 1..10;
        TOverHundred = 100..1000;
        TUppercase = 'A'..'Z';
```

在使用次范围定义新型别的时候，我们不用特别写出原始型别的名字，只需写出该型别的上下限即可，原始型别必须是有序型别，定义出来的新型别则会是另一个有序型别。当我们把变量宣告为任一个次范围型别时，就只能指派其范围中的任一数值给该变量，例如这样的写法就是正确的：

```
var
    UppLetter: TUpperCase;
begin
    UppLetter := 'F';
```

而以下的写法则是有误的：

```
var
    UppLetter: TUpperCase;
begin
    UppLetter := 'e'; //编译时期会发生错误
```

以上这段源码会在编译的时候，造成编译器错误，讯息则会是：Constant expression vio- lates subrange bounds." 但如果改成以下写法，编译器就会接受喔：

```
var
    UppLetter: TUpperCase;
    Letter: Char;
begin
    Letter := 'e';
    UppLetter := Letter;
```

如果在运行时间，我们有启用范围检查的编译器设定(在Project Option对话框的Compiler分页中)，就会得到*Range check error*的错误讯息。这也类似我们前面介绍过的整数型别溢位错误的问题。

我假设大家在开发程序的时候都会把这个编译器选项开启，这样一来在开发程序的时候就更容易及早发现问题，即使有些问题是在程序复合状况下才会出错。我们可以在最后要建立正式版本的源码时再把这个编译器选项关闭，这样会让最后发布出去的程序执行速度快一些。然而关闭这些选项后能让程序增加的速度也少到几乎可以忽略，所以我会建议把这些运行时间检查的选项都开启，即使已经到了最后要发布阶段的编译作业也一样。

列举型别

列举型别(通常被缩写成enums)可以构成另一个使用者自定有序型别。在列举型别中，我们不指定特定型别的范围，而是直接列出其中几个可能的数值，换句话说，我们只把我们需要的特定数值列出即可，举例如下：

```
type
    TColors = (Red, Yellow, Green, Cyan, Blue, Violet);
    TSuit = (Club, Diamond, Heart, Spade);
```

列表当中的每个值都对应一个序号，从0开始。当我们透过Ord函式来寻找列举型态中的特定数值，函式会回传该数值在列举型态中的顺序，以0为第一个元素的编号，因此Ord(Diamond)会回传1。

列举型态有不同的内部表示方法，预设情形下，Delphi使用8位来表示它，除非有超过256个数值被列在列举型别当中，才会使用16位来表示，当然也有32位的表示法，这方法会在需要跟C或C++的函式库兼容时使用。

笔记 我们可以改变列举型别的预设表示法，可以要求编译器给一个大一点的数值范围，只需使用$Z 这个编译器设定。这设定并不常被用到。

范围列举

列举型别的特殊常数值可以从其效应将之视为全局常数，有时在不同的列举型别当中，也有一些元素的名称会出现冲突，这就是为什么这个语言有支持范围列举，这个功能可以透过编译器设定$SCOPEENUMS来启用，而要使用特定列举型别的内容时，就需要把该列举型别的名称写在前头：

```
// classic enumerated value
s1 := Club;
// "scoped" enumerated value
s1 := TSuit.Club;
```

当这个功能被引入时，默认的规则仍沿用了传统的规则，以免使已经写好的源码无法执行。事实上范围列举已经改变了传统列举型别的规则，它使得列举型别必须强制使用一个型别名称作为前导描述。

给每个列举值一个绝对的名称，消除了名称重复的风险，有了型别名称作为列举值的前导描述，也可以让源码更容易阅读，不管源码写的多长，都不容易认错。

举例来说，IOUtils单元当中定义了这个型别：

```
{$SCOPEDENUMS ON}
type
  TSearchOption = (soTopDirectoryOnly, soAllDirectories);
```

这表示我们不能直接使用当中的第二个值soAllDirectories，需要使用它的时候，需要写出它的全名：

```
TSearchOption.soAllDirectories
```

在FireMonkey平台的函式库当中使用了许多范围列举型别，因此我们需要使用到这些型别的列举值时，就需要写出其完整的名称。旧版的VCL函式库通常还是使用比较传统的模型。而RTL则是两者的混合。

笔记 在 Object Pascal 的函式库里面，通常会在列举型别的列举值里面名称的前导字符(通常不成文的规定中会用小写)，例如在前一个例子里，用 so 代表 Search Option，但有了完整的型别名称作为前导，以字符作为名称前导字符就有些多余了，但以一般程序写作的习惯，这个作法短期内应该还不会全部消失。

集合型别

集合型别指的是一群数据的集合，这群数据会以其顺序作为整个型别的有序基础。这些有序型别通常是有限个数，而且通常会以列举型别或者次范围来表示。

如果我们以次范围1..3来表示一个集合型别，在Pascal的注记会写成1..3，这个集合当中可能的数值就只会包含单独的1或2或3、或者1和2, 2和3, 3和1, 或者1和2和3，或者完全不包含其中任何一个元素这几种组合，我们可以用数学里的组合来想象它，会是完全一致的。

集合型别的变量通常会储存一个集合当中任一个可能的数值组合，以上面的例子来说，这个变量可能储存的内容会是：没数据、1或2或3，或是前述的组合当中的任一种情形，也可能包含所有元素，以下是一个组合型别的范例：

```
type
  TSuit = (Club, Diamond, Heart, Spade);
  TSuits = set of TSuit;
```

现在我们可以定义一个这个型别的变量，然后存放一些数据到这个变量里面了。要定义一个集合型别里面的元素，我们得用逗点来分隔每个元素，做成一个列表，然后以方括号把这些元素括起来，以下的源码就示范如何把多个元素、一个元素、没有元素的这几种集合储存到一个集合型别的变量里：

```
var
   Cards1, Cards2, Cards3: TSuits;
begin
   Cards1 := [Club, Diamond, Heart];
   Cards2 := [Diamond];
   Cards3 := [];
```

在Object Pascal里面，一个集合通常用来储存几个不重复的数值，例如用来储存字体样式的变量，就是使用集合型别，这个变量可能包含粗体、斜体、

底线、删除线等样式。所以字体样式当然可以同时包含粗体、斜体，或者没有包含任何样式。因此这个变量就使用集合型别来储存。

我们可以在源码当中指派任何值给这个集合型别变量，举例如下：

```
Font.Style := []; // 没有特别样式
Font.Style := [fsBold]; // 只有粗体
Font.Style := [fsBold, fsItalic]; // 同时具备粗体与斜体
```

集合型别的运算方法

我们已经介绍了集合型别，这是只有Pascal特有的使用者自定型别，所以也必须介绍一下集合型别的运算方法。集合型别的运算方法有连集(+)、差集(-)以及交集(*)、是否属于该集合(in)，以及一些相关的运算方法。

要把一个元素加入集合中，我们可以使用连集(+)方法，把两个集合进行连接，以下是用来处理字体样式的相关范例：

```
// 加入粗体
 Style := Style + [fsBold];
// 加入粗体和斜体，移除底线（如果底线存在 Style 变量当中的话）
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

当然我们也可以使用Include跟Exclude这两个程序，这样会比较有效率（但无法使用在组件的集合型别属性上面）：

```
Include (Style, fsBold);
Exclude (Style, fsItalic);
```

表达式和运算方法

我们已经介绍过，可以把兼容型别的数据指派给变量、或把常数指配给变量，甚至把一个变量的内容指派给另一个变量。在许多情形里，我们也可以把一个表达式的结果指配给变量储存，包含一个或多个数据的运算，也可能是一个运算或多个运算的结果。表达式是Pascal这个语言的另一个核心元素。

使用运算方法

建立表达式并没有一定的规则，表达式只端赖运算方法的使用，在Object Pascal里面有许多运算方法，包含了逻辑运算、数学运算、布尔、实数、以及集合运算，以下是一些简单的例子：

```
//简单的表达式
20*5 // 乘法
30+n // 加法
a<b // 小于（比较判别式 ）
-4 // 负数
c=10 // 检验两个数是否相等 (等同于 C 语法的 == 符号)
```

表达式在绝大多数的编程语言里面都很常见，且大多数的运算方法与符号也大同小异。一个表达式可以是常数、变量、文字数据、运算符号、或者函式回传结果的任意合法组合，表达式可以用来决定要指派给变量的数据内容，计算函式或程序的参数，或者检查是否合于特定条件。我们只要有对任一个识别符号做处理，而不是单纯使用该符号，我们就是在使用表达式了。

笔记 表达式的结果通常都会储存为该型别的临时变量，这动作会由编译器自动处理好。我们可能会希望用特定的变量来储存这些结果，这样就不用一直重复同样的运算了。请记得，复合运算就会需要多个临时变量来存放计算结果，而这些动作已经都由编译器自动处理好了。

显示表达式的结果

如果您想对某些表达式进行实验，直接写个简单的程序最为直接了当，就相本书大多数的范例，就是以窗体画面程序建立简单的程序，然后透过自定的Show函式把某些执行结果显示给用户看。万一我们想显示的内容不是字符串，而是数字或者布尔逻辑值，我们就必须要做一些转换，例如呼叫IntToStr或者BoolToStr函式。

笔记 在 Object Pascal 里面，所有传给函式或者程序的参数都会以小括号括起来，有些其他的编程语言，例如 Rebol 与 Ruby，则会要求我们直接把参数写在函式或者程序的后面。而 Object Pascal 如果遇到重复呼叫程序或函式的情

形时，就只要把第二层或第三层的函数调用写在参数的字段即可，如以下的范例源码：

以下是从ExpressionsTest范例中节录的一些源码(在这当中我使用传统的IntToStr 语法让大家比较容易理解，当中我用表达式作为参数)：

```
Show (IntToStr (20 * 5));
Show (IntToStr (30 + 222));
Show (BoolToStr (3 < 30, True));
Show (BoolToStr (12 = 10, True));
```

执行结果如下：

```
100
252
True
False
```

我提供了这个范例作为架构，让您尝试不同型别的表达式与运算方法，并且可以看到对应的输出结果。

笔记 我们在 Object Pascal 里面所写的表达式，会被编译器处理，并产生汇编程序码，如果您想要变更其中一个表达式，就必须修改原始码并重新编译整个应用程序。然而系统函式库支持动态表达式，可以在运行时间进行计算，这个功能会在第 16 章进行介绍。

运算方法与其优先性

表达式是由运算符号和数值所组成的。前面曾提到过，在大多数的编程语言里面，大多数的运算符号都很相似，例如基本的比对符号。在这个章节里，我会就Object Pascal里面特有的运算符号做介绍。

以下是Object Pascal的运算符号列表，我们以运算优先级分组，并和C#，Java，Objective-C(以及大多数以C为基础的编程语言)的运算符号做一些比较。

关联与比较运算符号(优先级最低)

=	测试运算符号两边的识别符号内容是否相同(C语言是使用==)
<>	测试运算符号两边的识别符号内容是否不同(C语言是使用!=)
<	测试运算符号左方的内容是否小于右方的内容
>	测试运算符号左方的内容是否大于右方的内容
<=	测试运算符号左方的内容是否小于等于右方的内容，或左方的运算符号内容是否为右方的子集合。
>=	测试运算符号左方的内容是否大于等于右方的内容，或右方的运算符号内容是否为左方的子集合。
in	测试运算符号左方的内容是否为右方集合的元素之一
is	测试运算符号左方的内容是否为特定的型别（将在第8章里面介绍），或实作了特定的interface(将在第11章里面介绍)

相加或相减运算符号

+	数学运算的相加、集合的连集、字符串的连接、指针内容的相加
-	数学运算的相减、集合的差集、指针内容的相减
or	布尔或位数值之间的or运算(任一者成立即成立)，在C语言则是||或|
xor	布尔或位数值之间的xor运算(奇数个内容为true的时候即成立)，在C语言的位运算xor是使用^符号。

乘除以及位运算符号

*	数学运算的相乘、集合的交集
/	浮点数的相除
div	整数的相除(在C语言里这个运算也是用/符号)
mod	取余数(这个运算只对整数有效，在C语言是用%)
as	在运行时间进行型别确认的转换(第八章会介绍)
and	布尔或位运算的AND运算(在C语言则是用&&和&)

shl	位向左位移（在C语言是用<<)
shr	位向右位移（在C语言是用>>)

二元运算符号

@	变量或函式的内存地址，在C语言则是使用&
not	布尔或位计算的not(在C语言是使用!)

跟许多其他的编程语言不同，逻辑运算符号(包含and跟or运算符号)的优先级比比对符号来得高(例如大于跟小于符号)，所以如果我们这么写：

```
a < b and c < d
```

编译器会先执行b and c，所以通常会先出现编译器处理作业中显示型别兼容性错误。所以如果我们要进行两个比对，我们得用小括号把两个比对的表达式先包起来，写成这样：

```
(a < b) and (c < d)
```

在数学运算中，通则是先乘除后加减，所以前两个表达式是相同的，而第三个则不一样：

```
10 + 2 * 5 // 结果等于 20
10+(2*5) //结果等于 20
(10+2)*5 //结果等于 60
```

提示 虽然在很多情况下，算式里面加括号并不是必要的，但很多编程语言在处理这些计算的时候，并不是所有编程语言的优先处理顺序都是相同的，所以极力建议大家在算式里面尽量依照自己原始的想法把可以加的括号加上去，加了括号的算式，不仅是在源码阅读上更清楚，未来在其他人修改程序的时候，也可以减少被误解的机会。

有些运算符号在使用到不同数据型别变量的时候，会有不同的意义，例如运算符号+，可以把两个数字相加，可以把两个字符串相连，也可以把两个集合变成连集，甚至可以把两个指标的内容连在一起（如果特定的指标型别有启用指针数学功能的时候）：

```
10 + 2 + 11
10.3 + 3.4
```

```
'Hello' + ' ' + world'
```

但我们不能对两个字符做相加，在C语言里面也不行。

比较不常见的运算符号是div，在Object Pascal里面，我们可以把任意两个数字（实数或整数都行）相除，可以直接使用/这个符号，但用这个符号计算所得到的结果一定会是实数。如果我们希望两个整数相除，结果也得到整数的话，就得使用div这个运算符号。以下是两个简单的数据指派源码(等我们下一章介绍数据型别之后，这段源码就会更容易读懂了)：

```
realValue := 123 / 12;
integerValue := 123 div 12;
```

要确定整数除法是否有余数，我们可以使用mod计算，检查看看计算结果是不是0，如以下的源码：

```
(x mod 12) = 0
```

日期与时间

早期的Pascal语言并没有提供原生的日期与时间型别，Object Pascal则有提供原生的日期时间型别，是透过使用浮点数来记录日期与时间信息。准确的说，是在System单元里面，提供了TDateTime这个型别来处理日期与时间。

使用浮点数的原因，是需要足够的字段来记录年、月、日、时、分、秒，甚至精确到千分之一秒，这些都记录在一个Sinlge变量当中：

- TDateTime的日期部分是以一个整数数值来记录与1899-12-30的差别（如果是负数的话，则代表该日期是早于1899年的日期）
- TDateTime的时间部分则是以小数部分来记录当时是当天的哪个时间

典故 如果我们觉得该日期很奇怪，在这个表示法之后，有着跟 Excel 以及 Windows 应用程序用来处理日期的数据处理方式一个很长的故事。因为原本是以 1 来当做 1900 年一月一日，所以 1899 年的最后一天就是以 0 来表示。然而当时定义这个表示法的开发人员可能忘了 1900 年不是闰年，所以又把起始日向后调了一天，所以 1900 年的一月一日又变成了以 2 来表示。

刚刚提到过，TDateTime不是一个编译器中定义的预先定义型别，它是在System单元里面定义的：

```
type
  TDateTime = type Double;
```

笔记 System 单元已经几乎可以视为 Object Pascal 语言的一部分了，因为它几乎会自动被所有的单元文件引入，即使不写在 uses 区段里面（其实如果在 uses 区段里面写了 System，反而还会引起编译错误）。技术上来说，这个单元就是运行时间函式库的核心之一，我们会在第 17 章介绍它。

还有两个跟TDateTime一起用来处理时间与日期的型别，分别是TDate跟TTime，这两个型别只是TDateTime型别的别名，但他们常被系统函式用来去除掉日期或时间当中没有使用到的部份。

在Delphi里面使用日期或时间型别是相当容易的，因为Delphi为这个型别提供了许多处理的函式与方法。这些函式大多都放在SysUtils单元里面，或者是放在DateUtils单元里面(其中还有不少是用来处理时间的函式)。

以下列出一些常用的日期/时间函式：

NOW	回传内容为现在日期时间的值
Date	回传现在日期
Time	回传现在的时间
DateTimeToStr	把一个时间日期的值转换为字符串，使用默认的格式。如果需要更多控制选项，请使用 FormatDateTime 函式
DateToStr	把日期时间变量的日期部分转换为字符串
TimeToStr	把日期时间变量的时间部分转换为字符串
FormatDateTime	用特定的格式对日期时间进行限定格式显示，我们可以只选择我们想要显示的日期时间部分数据来显示，只需要透过格式字符串来设定即可
StrToDateTime	把一个字符串转换为日期时间值，如果字符串没有符合格式字符串的设定，系统会传出一个例外。对应的函式StrToDateTimeDef则会在遇到系统例外时，回传预设的值。
DayOfWeek	回传我们以参数传入的日期时间值是当周的星期几。
DecodeDate	从日期时间值取出年、月、日各个部分。

DecodeTime	从日期时间值取出时、分、秒各个部分。
EncodeDate	把年月日转换成日期时间值。
EncodeTime	把时分秒转换成时间值。

为了示范怎么使用这个数据型别以及其相关的一些函式，我建立了一个范例，名为TimeNow，当范例程序执行时，会自动把目前的时间与日期显示出来：

```
var
    StartTime: TDateTime;
begin
    StartTime := Now;
    Show ('Time is ' + TimeToStr (StartTime));
    Show ('Date is ' + DateToStr (StartTime));
```

第一行程序是呼叫了Now函式，然后把目前的日期与时间存放在StartTime变量里面。

笔记 当 Object Pascal 的函式被呼叫，而没有传递参数时，是不需要多打一组小括号在上面的，这一点跟 C 语言阵营的编程语言不同。

接下来的两行源码，则是把TDateTime的时间部分显示出来，当然要把它转换成字符串，我们才看的懂，然后是显示日期部分，执行结果如下：

```
Time is 6:33:14 PM
Date is 10/7/2014
```

要编译这个程序，我们需要引入SysUtils单元（是System Utilities的缩写），除了TimeToStr跟DateToStr这两个函式，我们也还可以用更强的FormatDateTime函式。

请留意，日期跟时间值在转换成字符串的时候，会根据系统的语言与地区设定(Windows Vista/Windows 7/Windows 8)。这个设定会从系统读取，而后储存为TFormatSettings这个型别的数据结构中。如果我们想要自己设定显示的样式，我们就得自己依照这个结构的规定，设定好要显示的样式，然后把它当成参数传给大多数的日期与时间函数。

笔记 TimeNow 范例程序当中，还有第二个按钮，我们可以透过它启动一个定时器。定时器组件可以依照我们的设定，每隔我们设定的时间间隔驱动一次事件。在这个范例中，如果您按了这个按钮，就会每隔一秒钟更新一次画面上的时间字符串。更有用的用户接口会是每秒钟更新时间的内容，我们也可以根据这个逻辑来做出一个时钟程序。

日期时间助手(DateTime Helper)

为了让程序人员方便处理TDateTime型别，Delphi 11提供了一个特殊的型别助手，跟本章前面的篇幅提过的内建数据型别类似。为TDateTime这个型别所提供的记录助手，名为TDateTimeHelper，它放在System.DateUtils单元里面。当中包含的操作功能，包含取得日期当中的年、月。或者转换为Unix的日期格式、检查AM/PM，检查该日期时间变量的内容是否为闰年等等。这个记录助手有超过150个方法，所以我们不在这里一一列出。

TDateTime助手型别也新增了一个新的方法名为NowUTC(目前的时间，但以UTC时区表示)，这个方法在传统RTL里面是没有的。我们用以下的源码来示范两个助手型别中的方法:Tomorrow跟 ToString:

```
uses
    DateUtils;
Procedure TForm1.Button1Click(Sender: TObject);
begin
    var MyDate: TDateTime := TDateTime.NowUTC;
    MyDate.Tomorrow.ToString;
end;
```

型别切换(Typecasting)与转型(Type conversions)

如我们介绍过的，在源码当中不能把一种型别的内容指派给另一种型别的变量去。原因是如果每次都要依照数据实际表示的方法来判断，我们可能会花上许多时间纠结在没有意义的环节上。

现在对每一种数据型别来说，这倒不一定正确。举例来说，数字型别就永远可以被标示为向上指派安全型别，因为我们永远可以把较短的数字型别指派给较长的型别，例如我们可以把Word指派给integer，或把integer指派给Int64。但另外一种指派，也就是把较长的数值指派给较短的数值时，编译器就会提出警告，因为我们可能只能把部分数值存到新的变量里面，举例来说，我们可以把整数数值指派到浮点数变量里面，但反之就不行了。

有些时候我们会想要改变数据型别，使得当时的数据处理合法。当我们需要这样做的时候，我们有两个选择，第一种是直接进行型别切换(Type casting)，这个作法会把数据复制一份，进行适当的转换，不依照该数据原始的型别与内容。当我们要进行型别切换的时候，等于是在告诉编译器说”我知道我自己在做什么，让我执行它吧”。如果我们使用型别切换，但并不确定自己正在做什么，这个处理会让我们失去编译器型别检查建构出来的安全网所提供的保护，当然，只有在源码初问题的时候才会出问题。

型别切换使用了简单的写法，像是在写呼叫数学函数一样，把要切换过去的数据型别当成这个数学函数的名称：

```
var
    I: Integer;
    C: Char;
    B: Boolean;
begin
    I := Integer ('X');
    C := Char (I);
    B := Boolean (I);
```

在使用同样长度数据储存数据的型别之间进行型别切换是安全的(因为数据会完整的被复制到被切换过去的型别数据空间，上面的范例源码就不是这样喔，那三个型别并不是使用相同长度的空间来储存数据的)。在有序型别之间的型别切换通常是安全的，但我们也可以在指标型别（当然，对象也可以）进行切换，只要我们自己真切的知道这些切换会有什么效应。

直接进行型别切换，对程序撰写来说是很危险的，因为这使得我们可以用另外一种表现方式来对数据进行处理。由于不同型别的内部数据储存常常是不一致的（且在不同平台上也有不同的处理方式），我们在这个动作上可能埋下日后难以找寻的问题，也因为如此，*建议大家应该尽量避免型别切换*！

第二种选择，是在把数据指派给另一种不同数据型别的变量之前，先透过型别转换函式处理。以下是我们在不同的几个基本型别上进行转换的常用函式（在本章的范例中，我也已经使用了其中的几个作为例子）：

Chr	把有序的数字转换为字符
Ord	把有序型别的数值转换为其序号
Round	把实数型别转换为整数型别，以四舍五入进行
Trunc	把实数型别转换为整数型别，以无条件舍去进行
Int	把浮点数的整数部分转换为整数表示
FloatToDecimal	把浮点数的内容转换为10进位表示
FloatToStr	以默认的数据格式，把浮点数转成字符串
StrToFloat	把一个字符串转换为浮点数

笔记 Round 这个函数的实作，是以 CPU 的内建功能来处理的。现代的处理器通常都会内建一个称为"Banker's Rounding"的功能，这个功能会把两个整数之间的数值(例如 5.5 或者 6.5)向上或向下取为整数，端看他们的整数部分是奇数还是偶数。或者我们可以使用 RoundTo 这个函数，这个函数就赋予我们控制权，看是要向上还是向下，不用由 CPU 决定了。

在本章前面的篇幅曾经提到，这些转换函式有些是直接在数据型别当中就提供了的（感谢型别助手这个功能）。当然有些从旧版Pascal语言就一直存在的转换程序，例如IntToStr，我们也已经可以透过大多数数值的型别助手的ToString来处理了。大多数的型别助手都提供了这些转换功能，但我们可以自己决定要用哪种方式来做转换，毕竟我们还是可以决定自己的源码风格在使用型别切换时要怎么写。

有些型别的转换函式我们会在后面的章节里面介绍，请注意上面的列表并没有把一些特殊型别包含进去，(例如TDateTime或者Variant)，也没有对一些对转换的功能提供许多延伸处理的函式做介绍，例如Format与FormatFloat这两个函式。

03:语言叙述句

对资料型别明确的概念(强型别的概念)是 Pascal 编程语言被发明时的创举之一。程序就是具备数据类型声明、加上用来处理这些数据型别变量的源码。

在 Pascal 编程语言被发明的时候，这两大支柱(数据型别与程序指令)的概念是由 Nicklaus Wirth 的巨作”Algorithm+Data Structures = Programs”(算法+数据结构=程序)所阐明的，这本书是 1976 年二月由 Prentice Hall 出版社所出版，是一本程序概念上的巨作，至今仍有再版。这本书比面向对象程序设计的概念早上许多年，可以被视为是现代程序概念的基础之一。以强型别为概念，并以此为理论基础，最后衍生出了面向对象编程语言的发展。

编程语言的叙述句是以关键词为基础，搭配其他元素，让我们能够让编译器得知我们要执行的一系列程序。叙述句通常会被以程序或函式的方式包装起来，这一点我们会在下个章节介绍，而目前我们只要先聚焦在我们可以用来撰写程序的一些基本的指令即可。我们在第一章里面介绍过的（在介绍使用空格符与源码样式的篇幅里）实际上撰写程序是很自由的，我们也介绍了批注与一些特别的元素，但还没有来得及完整的介绍其他核心概念，像是程序的叙述句。

简单与复合叙述句

程序指令通常会被称为*叙述句(Statements)*。一段程序区块可能会由好几个叙述句组成，叙述句可以分为两种：简单与复合叙述句。当一个叙述句没有包含其他子叙述句的时候，我们称之为简单叙述句，最简单的例子，就是指派叙述跟呼叫程序，在 Object Pascal 里面，简单叙述句式以分号来做分隔的：

```
X := Y + Z;  // 指派叙述
Randomize;  // 呼叫程序
```

要定义一段复合型的叙述句，我们可以在 begin 跟 end 之间写入一个或多个的叙述句。begin 跟 end 在此扮演着多个描述句的容器以及相似的角色，但

是跟 C 阵营语言当中的大括号并不完全一致。复合型叙述句可以出现在任何 Object Pascal 简单叙述句出现的位置：

```
begin
  A := B;
  C := A * 2;
end;
```

在复合叙述句里面的最后一个叙述句的分号不一定要写：

```
begin
  A := B;
  C := A * 2
end;
```

以上两段源码都是正确的，第一段的写法中，最后一句最后结尾的分号其实是没有用的（但也无伤大雅），这个分号事实上算是一个空的叙述句，也就是一句没有源码的叙述句，这一点跟其他编程语言是很不一样的，尤其是对 C 语言阵营的编程语言来说，在 C 语言阵营的编程语言来说，每个叙述句结尾的分号都是不可省略的。

请注意，很多时候，没有源码的叙述句，在内部的循环当中，有时候也是可以直接出现的，例如：

```
while condition_with_side_effect do
   ; // 没有源码的叙述句
```

虽然最后一个分号并没有特别的作用，大多数的使用者还是习惯写上去，而我也建议大家要写。因为程序总是需要修改的，常常我们写了一段时间的源码，后来又要在后头加上一些其他的源码，这时就不用老是在寻找最后一行了。但如果多加了一个分号，可是会导致编译器例外发生的，最常见的例子，就是在 else 前面加上一个分号。

IF 叙述句

条件叙述句是以一个条件来判断要执行特定区块的源码，或者不执行它（在不符合条件的时候），条件判断式的语法关键词有两个：if 跟 case。

If 叙述句是用来判断符合特定的一个条件，如果符合该条件，就执行该区块的源码 (if-then)，或者合于条件时执行一区块的源码，不符合时执行另一区块的源码(if-then-else)，条件判断需以布尔表达式定义。

我们提供了一个简单的范例:IfTest 来示范如何撰写条件叙述句，在这个程序中，我们使用了 checkbox 来取得使用者的输入值，透过 checkbox 的 IsChecked 属性（并把它储存在一个布尔变量当中，虽然这么做并不是必要的，我们可以直接使用该属性的值作为判断式）：

```
var
    isChecked: Boolean;
begin
    IsChecked := CheckBox1.IsChecked;
    if isChecked then
        Show ('Checkbox is checked');
```

如果该 checkbox 有被勾选，程序就会显示一个简单的讯息，不然的话就不会有任何事情发生，如果上面的这段程序以 C 的语法来写的话，就会长得像这样（C 语言的条件判断式一定要用小括号把它包起来）：

```
if (isChecked)
  Show ("Checkbox is checked");
```

还有一些编程语言会让我们用 endif 来作为判断叙述句的结尾，好让我们可以在程序区块当中使用多个叙述句，在 Object Pascal 当中，if 后面只能使用单一叙述句，所以如果我们需要使用多个叙述句的时候，就必须用 begin-end 来把带有多个叙述句的程序区块给包起来了。

如果我们希望能依照该条件的成立与否分别执行不同的源码，就可以使用 if-then-else 这样的语法(以下的范例中，我会直接把 checkbox 的属性当成条件判断式)：

```
// if-then-else statement
if CheckBox1.IsChecked then
    Show ('Checkbox is checked')
else
    Show ('Checkbox is not checked');
```

请留意，在 if 后面的叙述句，是不可以用分号结尾的，不然编译器会回报语法错误，这是因为 if-then-else 被当成一个单一叙述句，所以我们不能在当中使用分号把它切断。

if 叙述句可以很复杂，当中的条件判断式可以由一连串的条件组合而成（使用 and, or, not 运算符号）。而 if 叙述句里面也还可以在包含其他 if 叙述句，

我们也可以一连串的组合 if-then-else-if-then 这样的句子，我们可以任意组合多个 else-if 这样的条件叙述句。

在 IfTest 范例中的第三个按钮，就示范了这样的情形，透过第一个在 edit 组件里面输入的字符作为输入判断值：

```
var
    aChar: Char;
begin
     // multiple nested if statements
     if Edit1.Text.Length > 0 then begin
        aChar := Edit1.Text.Chars[0];
        // checks for a lowercase char (two conditions)
        if (aChar >= 'a') and (aChar <= 'z') then
            Show ('char is lowercase');

        // follow up conditions
        if aChar <= Char(47) then
            Show ('char is lower symbol')
        else if (aChar >= '0') and (aChar <= '9') then
            Show ('char is a number')
        else
            Show ('char is not a number or lower symbol');
      end;
```

要仔细看这段源码，然后执行范例程序，看看跟你预期的是否相同，以类似的程序做练习，我们的程序撰写能力进步的才快。我们也可以用这个范例当基础，加上多一点条件跟选项，增加它的复杂度，随我们所想的去进行程序的改写与测试。

Case 叙述句

如果我们要判断的条件很复杂，例如对同一个变量的数值要分成多个不同部分来分区处理的时候，用 IF 叙述句会变得很复杂，这时候我们可以改用 case 叙述句。Case 叙述句可以让我们判断同一个表达式的不同范围的数值，这些数值必须是常数，而且必须是有序型别的数值，不能重复。最后，我们也可以在所有列举的范围数值之外，使用 else 叙述句，让我们没能预想到的所有情形都在该段源码来处理。Case 叙述句并没有像 endcase 这样的结束关键词，它仍旧使用 end 来结尾。

笔记 建立 case 叙述句需要使用列举数值，在 case 叙述句中是不能使用字符串作为条件值的，如果要判别不同的字符串内容时，只能用 if 叙述句或者不同的数据结构，例如 dictionary。（我们在第 14 章会进行介绍）

以下是个简单的范例（是 CaseTest 项目的一部分），在这个范例当中，透过用户输入的整数值来作为 case 叙述句的判断资料：

```
var
    Number: Integer;
    AText: string;
begin
        Number := Trunc(NumberBox1.Value);
        case Number of
          1: AText := 'One';
          2: AText := 'Two';
          3: AText := 'Three';
        end;
        if AText <> '' then
          Show(AText);
```

另一个例子则是前面提到过的复杂的 if 叙述句的延伸，把输入的内容作为 case 叙述句的不同判断值：

```
case AChar of
  '+' : AText := 'Plus sign';
  '-' : AText := 'Minus sign';
  '*', '/': AText := 'Multiplication or division';
  '0'..'9': AText := 'Number';
  'a'..'z': AText := 'Lowercase character';
  'A'..'Z': AText := 'Uppercase character';
  #12032..#12255: AText := 'Kangxi Radical';
else
  AText := 'Other character: ' + aChar;
end;
```

笔记 在上面的范例程序当中，部分的数值范围使用了次范围数据型别的语法，反之，大多数的单一数值则使用了逗点作为分隔符。而超过英数字的字母，则使用 Kangxi Radical 作为显示文字，其中的侦测条件就直接使用数值，因为其中大多数的文字都无法在 IDE 编辑器显示，例如”一”，是这个群组的

第一个元素。（这部份在简体、繁体中文其实是可以正确显示的，只是大多数非中文用户的操作系统会无法正确显示）

在程序实务上，使用 else 来处理未被定义的条件是比较保险的。在 Object Pascal 里面，case 叙述句是用来判定要执行的路径，它并不会自己决定要如何选择。换句话说，它会执行符合的判断式的分号后面的叙述句或程序区块，而不是决定切入点。换句话说，它只会执行符合条件的判断式之后的叙述句，执行完以后就不会执行之后的其他源码了。

这跟 C 语言系列的编程语言很不一样，在 C 系列的语言里面，是用 switch 指令来达成同样的动作，但它是决定了切入点之后，就从该点开始执行下去，除非我们在希望中断的地方写入一个 break 叙述句（这个指令在 Java 跟 C# 跟字面上的指令实作是不太一样的），C 语言的写法如下：

```
switch (aChar) {
   case '+': aText = "plus sign"; break;
   case '-': aText = "minus sign"; break;
   ...  default: aText = "unknown"; break;
}
```

For 循环

Object Pascal 和其他编程语言一样，都有很传统的重复执行功能，称之为循环，在 Object Pascal 当中包含了三个循环指令：for, while 跟 repeat 这三个叙述句，而后来又加入了 for-in(或者称为 for-each)这个叙述句。如果您已经熟悉其他编程语言，那么这些循环您一定不会陌生，所以我会很快的大致上介绍一下这些循环（会点出跟其他编程语言的异同处）。

For 循环在 Object Pascal 里面是以计数器为基础，所以在 For 循环执行的时候，是对计数器做递增或递减处理。以下是一个简单的 For 循环范例，会把 1-10 的数字做加总（是 ForTest 范例的一部分）：

```
var
            Total, I: Integer;
begin
            Total := 0;
            for I := 1 to 10 do
               Total := Total + I;
            Show(Total.ToString);
```

结果当然不奇怪，一定会是 55。在介绍过循环之后，我们用另外一种写法，介绍使用行内变量的宣告方法来处理循环的计数变量(这语法有部分看起来像是结合了部分 C 语言阵营的语法，我们稍后再讨论)：

```
for var I: Integer := 1 to 10 do
  Total := Total + I;
```

从上面这个例子，我们可以使用到型别推定的好处，忽略一些型别的细节。我们用上面这个例子来写成完整的 Delphi 源码，就会变成:

```
var
  Total: Integer;
begin
    Total := 0;
    for var I := 1 to 10 do
        Total := Total + I;
    Show(Total.ToString);
```

使用行内变量作为循环计数变量的好处之一，就是这个变量的生命周期将会被限制在循环里面: 如果在 for 循环之后的源码里面还使用到这个变量的话，IDE 跟编译器都会显示有错误。用传统的变量宣告写法，这种情形顶多会显示警告而已。

Pascal 的 For 循环跟其他编程语言的 For 循环相比之下，弹性比较小(例如每次循环的计数变化量只能是一)，但这个现象也很容易理解，我们看看 C 语言的 For 循环语法就知道：

```
int total = 0;
for (int i = 1; i <= 10; i++) {
  total = total + i;
}
```

在其他的语言中，For 循环的变化量是每次执行过之后，由一个表达式来处理的，所以我们当然可以用我们希望的表达式放在里面来执行，要一次跳两个数字或其他处理法，也都没有问题。但坏处则是有时候会让源码变得比较不容易读懂：

```
int total = 0;
for (int i = 10; i > 0; total += i--) {
.. }
```

然而在 Object Pascal 里面，我们对 For 循环只能使用单步递增或递减。如果想要在每次循环执行过后做比较特别的设定或处理，我们可以改用 while 或是 repeat 指令。

在 For 循环当中唯一可以更换的部分，就是递增改成递减，或者称为逆向循环，关键词从 to 改用 downto：

```
var
        Total, I: Integer;
begin
        Total := 0;
        for I := 10 downto 1 do
          Total := Total + I;
```

笔记

逆向循环也是很有用的，例如当我们希望对一个列表型的数据结构进行内容处理，当删除其中的一些元素时，我们通常会逆向而行，以一个正向的循环，你会影响正在处理到的元素顺序（例如我们删除了列表中的第三个元素，原本的第四个元素就变成了第三个元素，而我们本来在第三个元素的位置，向后移动一个位置（现在的第四个），就到了第五个元素(中间直接跳过了原本的第四个元素)

在 Object Pascal 里面，for 循环的计数器不用非得是数字不可，只要是一个有序型别的数值就行了，例如字符，或者是列举型别都行。这也让我们写出来的源码更容易阅读，以下是使用字符型别来做 for 循环的一个范例：

```
var
AChar: Char;
begin
    for AChar := 'a' to 'z' do
        Show (AChar);
```

上面这段源码(是 ForTest 城市的一部分)会秀出所有英文字母，会以每行一个字母的方式显示在 Memo 组件里面。

笔记

我也提供了一个类似的范例，只是这个范例是使用数字作为计数器，它是第二章 CharTest 范例的一部分，在该范例中，所有输出的字符会被连接成一个字符串以后才一起输出。

以下是另一个代码段，用来示范如何使用自定列举型态作为 for 循环的计数器：

```
type
   TSuit = (Club, Diamond, Heart, Spade);
var
   ASuit: TSuit;
```

```
begin
    for ASuit := Club to Spade do
        ...
```

这个代码段当中的循环，会把该列举型别的所有数值都处理过一次，最好是能够对每个元素的型别都进行精确处理(在定义修改的时候会比较有弹性)，这样就不用写出该型别的第一个跟最后一个数值的名称了：

```
for ASuit := Low (TSuit) to High (TSuit) do
```

在类似的写法中，用 for 循环来让数据结构中的所有元素都跑一遍是很常见的，在这个案例中，我们可以用以下这段程序片段（它是 ForTest 项目的一部分）：

```
var
    S: string;
    I: Integer;
begin
    S := 'Hello world';
    for I := Low (S) to High (S) do
        Show(S[I]);
```

如果您比较不想带到数据结构里面的第一个跟最后一个元素，建议可以使用 for-in 循环。在接下来的章节里，我们就要来讨论 for 循环的各种特定用途了。

笔记 在 Object Pascal 里面，编译器是怎么透过[]符号来直接处理字符串当中的每个元素，并判断字符串的起始与结束值是一个很复杂的议题，虽然在所有平台中的作法现在几乎已经都一样了。这个主题在第六章里面我们会再进行介绍。

对于以 0 为起始值的数据结构而言，我们的起始索引值是从 0 到该数据结构的结尾。通常的写法会像这样:

```
    for I := 0 to Count – 1 do …
    for I := 0 to Pred(Count) do …
```

关于 for 循环要注意的最后一个问题，是『在循环结束之后，循环的计数变量会被怎么处理?』简单来说，这个变量的内容后续不会再被处理，如果我们在循环结束之后的源码当中又使用了这个循环的计数变量，编译器会发出警示。使用行内变量来宣告循环计数变量的好处之一，就是该变量的生

命周期只在循环源码当中，循环结束后该变量就无法再被使用，所以在编译阶段就会被报错了(这样的保护更周到)：

```
begin
    var Total := 0;
    for var I: Integer := 1 to 10 do
        Inc (Total, I);
    Show(Total.ToString);
    Show(I.ToString); // 编译器错误: 没有宣告过的识别符号 'I'
```

For-in 循环

Object Pascal 提供了特别的循环结构，可供一个 list 或者 collection 把里面的每个元素都列举一次，称为 for-in(在其他编程语言里面这功能常被称为 foreach)。在这个 for 的循环中会对于数组、list 或者字符串，或者其他具备容器功能的型别里的所有元素进行处理。跟 C#不同的是，Object Pascal 并不要求实作 IEnumerator 这个接口，但内部会以相似的作法来实现。

笔记 在第 10 章里面，您可以在类别中加入 for-in 循环来观察这个循环内部的技术细节。

我们用一个非常简单的容器型别：字符串来作为开始吧，我们可以把字符串看成是字符的集合(Collection)，在前一节的结尾，我们介绍了怎么用一个 for 循环来处理字符串中的所有元素。在接下来的范例中，我们也可以用 for-in 循环来达到相同的效果，在底下这个名为 Ch 的变量，会依序接收到字符串里每个元素的内容。

```
var
  S: string;
  Ch: Char;
begin
    S := 'Hello world';
    for Ch in S do
        Show(Ch);
```

上面这段程序是 ForTest 范例的一部分。For-in 循环比传统的 For 循环方便的地方，就是我们不用花心思去记录字符串的第一个位置、以及最后一个位置在哪里。因此这种循环更容易撰写与维护。

跟传统的 for 循环一样，在 for-in 循环里面如果使用行内变量也会有些优点。我们可以把上面的范例改用行内变量写成效果完全相同的源码:

```
var
  S: string;
begin
   S := 'Hello world';
   for var Ch: Char in S do
       Show(Ch);
```

For-in 循环可以用来读取多种不同数据结构里的元素：

- 字符串里面的字符(请参考上面的程序片段)
- 集合当中的每个元素
- 静态或动态数组里面的元素，也包含二维数组(将在第五章里面介绍)
- 支持 GetEnumerator 的对象类别，包含许多预先定义的类别，像是 StringList 里面的字符串、不同容器类别的元素，处理这些类别的方法将在第 10 章里面介绍。

目前要介绍一些进阶的模式还言之过早，所以我们稍后再回头来看本章节的这些范例吧。

笔记 在某些编程语言里面的 for-in 循环(例如 JavaScript)执行起来特别慢，已经成了负面口碑了。但在 Object Pascal 里则不然，在 Object Pascal 里面的 for-in 循环，效能跟 for 循环几乎一样好。为了证明这一点，我在 LoopsTest 范例程序中加了一些计时用的源码，在这个程序中，会先建立一个包含三千万个元素的字符串，然后再用两种循环来扫描内容。（循环中的每次作业都很简单，两种循环所造成的差异不到百分之十，在我的 Windows 机器上面执行起来，分别用了 62ms 跟 68ms）而已

While 和 Repeat 循环

While-do 和 repeat-until 的意义，是重复执行一个程序区块，直到特定条件达成。两种写法之间的差异，只在于一个是在执行前检查该条件，而一个是在执行源码之后进行检查而已。换句话说，repeat 循环永远会执行至少一次。

笔记 绝大多数其他的编程语言都只提供一种开放式循环叙述句，大多都像 while 循环。C 语言的语法和 Pascal 语法一样，提供了两种语法，分别写成 while

跟 do-while 两种语法。请注意，C 语法的 while 判断句是一样的含义，跟 Pascal 的 repeat-until 语法是不同的，until 是在条件成立时结束循环喔。

了解 repeat 循环为何至少执行一次是很容易的，请看以下这个简单的范例源码：

```
while (I <= 100) and (J <= 100) do
begin
    // use I and J to compute something...
    I := I + 1;
    J := J + 1;
end;
repeat
    // use I and J to compute something...
     I := I + 1;
     J := J + 1;
until (I > 100) or (J > 100);
```

笔记 请留意 while 跟 repeatd 当中我用括号括起来的子条件。在这个例子中是必要的，在编译器将要执行或进行比较之前(就像我在第二章里面提到关于运算符的章节里面提到的)

如果 I 或 J 的初始值比 100 大，while 循环就会直接结束，但 repeat 循环则会执行一次。

这两种循环之间另一个关键性的差异，则是 repeat-until 的条件是相反的，又称为反转条件(符合的时候就脱离循环)，循环的执行会在 until 语句后面的条件成立的时候，停止执行源码。而在 while-do 循环里面，则是在 while 后面的条件成立的时候才执行其中的源码。因此在上面的源码里面，两个循环的判别条件的写法正好是完全相反的。

笔记 反转条件在摩根定律当中已经广为周知 (请参考维基百科: http://en.wikipedia.org/wiki/De_Morgan%27s_laws)

循环的范例源码

要对循环有更多深入的了解，我们得看一些实际案例。在 LoopTest 范例程序里，点出了固定次数的循环跟开放式条件循环的差异，第一个循环是固定次数的循环，也就是 for 循环，用来依序显示数字：

```
var
I: Integer;
begin
    for I := 1 to 20 do
        Show ('Number ' + IntToStr (I));
end;
```

同样的输出结果，也可以用 while 循环来达成，透过一个内部变量，每次增加 1(记得，我们要在显示数字之后帮这个数字加一)。透过 while 循环，我们可以自由的设定递增的数值，例如每次加 2:

```
var
I: Integer;
begin
    I := 1;
    while I <= 20 do
    begin
        Show ('Number ' + IntToStr (I));
        Inc (I, 2)
    end;
end;
```

上述的范例程序，会显示从 1 到 19 的奇数。

上面这两个循环，作法上是相同的逻辑，会把程序区块执行固定次数。但执行次数并非在源码撰写的时候就能够预测的，有些源码的情形会根据运行时间中变量的变化或外部条件的变化而有不同。

笔记 在撰写 while 循环的时候，请一定要留意判别条件是不是有可能永远不会成立，例如以特定变量的大小作为判别条件的时候，要留意该变量是否记得在每次执行后有被进行递增或递减，以防变成无穷循环（无穷循环发生时，会把 CPU 完全占用，在多核的操作系统中，会看到特定的一个 CPU 使用率维持着 100%, 直到操作系统把该处理程序删除掉）

为了示范这种情形，我写了一个 while 循环，其判别条件是以计数器为依据的，但该计数器变量的数值则是随机递增。为了达到这个目的，我使用 Random 函式来建立小于 100 的整数。这个程序的执行结果，是会从 0 到 99 随机选取数字，而这一系列的随机数字则决定 while 循环执行的次数：

```
var
I: Integer;
begin
    Randomize;
    I := 1;
    while I < 500 do
    begin
        Show ('Random Number: ' + IntToStr (I));
        I := I + Random (100);
    end;
end;
```

如果你记得先呼叫 Randomize 这个子程序，它会重设每次随机数字产生时的依据，这样一来每次执行时，所产生的随机数字就会都不一样。以下是两次执行结果，我们把它并列在一起做对照:

```
Random Number: 1                Random Number: 1
Random Number: 40               Random Number: 47
Random Number: 60               Random Number: 104
Random Number: 89               Random Number: 201
Random Number: 146              Random Number: 223
Random Number: 198              Random Number: 258
Random Number: 223              Random Number: 322
Random Number: 251              Random Number: 349
Random Number: 263              Random Number: 444
Random Number: 303              Random Number: 466
Random Number: 349
Random Number: 366
Random Number: 443
Random Number: 489
```

请注意，不只是每次建立的随机数字都不一样，也因为 while 循环执行的次数也是透过随机数字产生而决定的，所以每次执行时，会产生几次随机数字也是每次都不同的，因此上面的执行结果连个数都不同。

用 Break 和 Continue 指令来中断流程

尽管循环在语法与执行上有些许不同，但所有的循环都是相同的意义：依照特定的条件，让程序区块执行许多次。然而有些时候，我们会希望在循环的逻辑里面加上一些额外的规则。举个例子，我们写了一个 for 循环用来侦测特定字符是否出现（这段源码是范例程序 FlowTest 的一部分）：

```
var
        S: string;
        I: Integer;
        Found: Boolean;
begin
   S := 'Hello World';
   Found := False;
   for I := Low (S) to High (S) do
      if (S[I]) = 'o' then
        Found := True;
```

在源码的后段，我们可以藉由检查 found 变量的值来得知特定的字符是否在该字符串里面出现过。这样的写法，程序会在发现要检查的特定字符发生后，仍然要把后面所有的字符一一查完才会停止（如果字符串很长，就会浪费掉许多运行时间）

传统的解法，是用 while 循环来同时检查这两个条件(循环的计数器跟 Found 这个变量的数值)

```
var
   S: string;
   I: Integer;
   Found: Boolean;
begin
   S := 'Hello World';
   Found := False;
   I := Low (S);
   while not Found and (I <= High(S)) do begin
      if (S[I]) = 'o' then
        Found := True;
        Inc (I);
   end;
```

上述的源码很合逻辑，也很容易读懂，我们还要在上头加一些条件，如果条件变得越来越多且越来越复杂，要把不同条件合并就会让源码变得更繁复。

这就是为何在 Object Pascal 的语言中，提供了系统层级的子程序让我们可以在标准的循环执行流程中进行更改的原因了，更改循环执行的指令有两个：

- Break 指令：让我们可以中断循环的执行，直接跳到循环外的下一个指令，中止原本循环还要执行的所有动作。
- Continue 指令：会从呼叫这个指令的点，跳过以下所有的循环指令，直接以循环条件的下一个数值从循环开头点继续执行（除非下一个数值已经超过了循环的终止条件，此情形就会跳出循环了）

使用 Break 指令，我们可以把原来检查特定字符是否出现的源码改成以下写法：

```
var
   S: string;
   I: Integer;
   Found: Boolean;
begin
    S := 'Hello World';
    Found := False;
    for I := Low (S) to High (S) do
      if (S[I]) = 'o' then
      begin
           Found := True;
           Break; // jumps out of the for loop
      end;
```

另外两个系统函式：Exit 跟 Halt，则提供了让我们从子程序执行中直接脱离，以及直接停止整个程序执行的功能。我们会在下个章节介绍 Exit 指令，通常我们不太会呼叫 Halt 让程序突然中断执行（所以这个指令在本书里面将不会加以介绍）

要介绍 Goto 指令了吗?绝不!

事实上除了上述四个指令之外，还有一些方法可以中断程序的执行流程，在最原始的 Pascal 语言里面，有提供了恶名昭彰的 goto 叙述句，让我们可

以直接以不同的标签标注在源码当中进行跳跃。跟条件判断式跟循环不同，条件判断式跟循环是让我们在连续的指令中进行特定的中止或排除部分条件。而 goto 这个指令提供的则是不稳定的源码跳跃，所以我们绝不建议在任何程序当中使用它。我有提过在 Object Pascal 里面支持这个指令吗？我可没这么说，但我不会提供任何程序范例，对我来说，goto 指令已经不存在了。

事实上，译者在教授程序写作的时候，也常对学员说，绝对不要使用 goto 这个指令，因为 goto 指令的跳跃是毫无判断的，直接用 goto 指令跳跃，可能会让许多变数在未进行初始化之前就被使用，或者造成更多损害，记得，千万别用这个指令。

笔记 还有一些程序叙述句是我们还没有介绍到的。例如 with 叙述句，这个叙述句是用来处理 record 结构的，所以我们会在第五章介绍，With 也是另一个有争议存在的程序功能，不过没有像 goto 这么令人深恶痛绝。

04:程序与函式

在 Object Pascal 语言里面强调的另一个重要观念（和 C 语言的概念极为相似），就是子程序的概念。基本上子程序的概念就是把一系列的指令集合起来，并赋与一个独特的名字，子程序可以被呼叫许多次。子程序（或者称为函式）是透过它们的名字进行呼叫的。透过子程序的设计，我们可以重复使用相同逻辑的源码，不用一再重复撰写相同的源码。这样一来就在整个程序中需要使用该段程序逻辑的地方使用同样版本的源码。从这个观点来看，我们可以把子程序看成是封装机制的基础。

程序与函式

在 Object Pascal 里面，子程序以两种形式存在：程序(procedure)和函式(function)。理论上，程序是我们要求计算机执行的一个动作，而函式是计算后要把数据回传的动作。两者之间的差异，是函式会回传结果，这个回传的结果可能是数值、型别，而程序不会回传任何数据。在 C 语言里面，只提供了单一语法，就是函式，当不回传数据的时候，C 语言就要求程序人员把回传数据的型别写成 void，C 语言的这个作法就完全等同于 Object Pascal 的程序。

两种写法的子程序都可以传递多个相同或不同型别的参数，我们稍后会介绍，程序跟函式都是类别(Class)当中用来实作方法(method)的基础，而在这个案例中，两种形式的区别仍旧存在。事实上，跟 C, C++, JAVA, C#或 JavaScript 都不同，我们在宣告函式或者方法的时候，仍旧会用到 procedure 或者 function 这两个关键词来宣告子程序或者方法。

实务上，除了宣告用的关键词不同，程序跟函式之间的差异微乎其微：我们可以宣告函式，然后最后不回传数据，或者完全不管回传值，这样就可以把函式当成程序来用（这可能会导致一些小错误发生）。也可以在程序中，透过传址（call by reference）的参数把最终结果回传给呼叫它的源码，这样也可以把程序当成函式来用（关于传址的参数，在本章稍后会介绍）。

以下是用 Pascal 语法定义一个程序的写法，会使用到 procedure 这个关键词，这段程序片段是范例程序 FunctionTest 项目的一部分：

```
procedure Hello;
begin
  Show ('Hello world!');
end;
```

比较一下，如果用 C 语言的语法来撰写上面这个子程序的功能，写起来应该会几乎完全相同。在这里面没有关键词，也不要求参数，且不用回传任何资料：

```
void Hello ()
{
    Show ("Hello world!");
};
```

事实上 C 语言的语法在撰写程序跟函式时完全相同。而在 Pascal 语言的语法中，函式还需要一个特殊的关键词与回传值(或者回传的型别)。

笔记 在 Object Pascal 的语法中，还有一个地方跟其他语言不同的，就是在函数声明的时候，结尾处必须用冒号来宣告函式要回传的数据型别。

在函式的撰写中，有两种写法可以用来定义最后的回传值，一个是直接把回传值指派给函数名称，另一种则是指派给 Result 这个关键词:

```
// 传统 Pascal 写法
function DoubleOld (Value: Integer) : Integer;
begin
    DoubleOld := Value * 2;
end;
// 现代写法
function DoubleIt (Value: Integer) : Integer;
begin
    Result := Value * 2;
end;
```

笔记 Object Pascal 的语法中，其实还有第三种写法可以用来指派回传值，我们会在本章的『附带回传值离开』这个小节进行讨论。

透过 Result 这个变量，而不使用函式名称来指派回传值，是比较常见的作法，同时也让源码比较容易阅读，直接把回传值指派给函数名称，是传统 Pascal 的写法，近代已经比较少用了。

我们再以 C 语言的语法作为对照，把上面的范例源码改写成 C 语言，可以写成这样：

```
int DoubleIt (int Value)
{
  return Value * 2;
};
```

笔记

在 C 阵营的编程语言里，return 这个描述是定义函式的回传值，并且结束函式的执行，把程序执行的权限交回原本呼叫该函式的点。在 Object Pascal 里面，指派数据给函式的回传值并不会终止该函式的运作。这也是为什么 Result 变量很常在 Object Pascal 里面被使用，例如一开始就指派预设的回传值，或者是在算法里面修改回传结果。另外，如果我们想要停止函式的执行，得要在流程控制中呼叫 Exit。相关的议题会在之后的章节『附带回传值离开』里面再深入介绍。

相对于这些子程序的定义撰写法，呼叫这些子程序的语法就相对直觉的多，我们只需要直接输入子程序的名字，如果子程序需要参数的话，则以小括号把这些参数括起来即可。如果子程序不需要参数，则在呼叫的时候连空的小括号都可以省略了（这当然也是跟 C 语言作为对照的，C 语言就算子程序没有要求参数，也得输入个空的小括号才行）。以下的程序片段都是本章范例项目 FunctionsTest 的一部分：

```
// 呼叫程序
Hello;
// 呼叫函式
X := DoubleIt (100);
Y := DoubleIt (X);
Show (Y.ToString);
```

我们示范一下源码封装的概念。当我们呼叫 DoubleIt 这个函式，我们不用知道里面使用的算法，也不用知道它是怎么被实作出来的，如果我们事后又找到了另一个把数字做 Double 的更好的方法，我们可以再写另一个 DoubleIt 函式来替换原本的 DoubleIt 函式，而呼叫 DoubleIt 函式的所有源码都不用再改动。

同样的原则也可以套用在 Hello 程序上：我们可以直接修改 Hello 这个程序来改善程序输出的作法，而原本呼叫 Hello 的源码则完全不用更动，只要修改完 Hello 函式，所有原本使用 Hello 函式的地方全部一起受益，以下就是我们修改程序实作的写法：

```
procedure Hello;
begin
    Show ('Hello world, again!');
end;
```

预先宣告

当我们需要使用一个识别符号(不论其型别是什么)，编译器必须预先得知这个符号，并且必须知道这个识别符号将会指向的参考地址。为了满足这个需求，我们得在使用任何识别符号之前先提供完整的定义。然而，有些情况下这个要求很难达成。如果程序 A 呼叫程序 B，而程序 B 又呼叫程序 A，当我们开始撰写这个源码的时候，我们等于是在呼叫一个编译器还没看到的程序。

在这个情形下(还有很多情形也会有类似的情况发生)，我们可以先宣告一个函式或者程序，把完整的名称、参数都宣告好，但不用提供完整的源码。要做到这一点，我们只要把完整的程序或函数名称宣告写好，最后加上一个 forward 关键词即可，例如：

```
procedure NewHello; forward;
```

在源码后段，我们再把完整的源码写好(这种写法，实作的源码跟预先宣告必须位于同一个单元文件里面)，这样一来，我们就可以在完整的源码还没出现之前，就直接呼叫它了，以下就是这样的例子：

```
procedure DoubleHello; forward;
procedure NewHello;
begin
    if MessageDlg ('Do you want a double message?',
        TMsgDlgType.mtConfirmation, [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo],  0) = mrYes then
        DoubleHello
    else
        ShowMessage ('Hello');
end;
```

```
procedure DoubleHello;
begin
    NewHello;
    NewHello;
end;
```

笔记 上面的程序片段中所用到的 MessageDlg，是 FireMonkey 框架里所提供的一个简单的方法，让我们可以透过对话框询问使用者进行确认(在 VCL 框架里面也有类似的方法，也很容易使用)，其参数是讯息、对话框的种类、我们要显示给用户看到的按钮，最后的回传值则是用户点击的按钮种类。

这样的功能(上面的程序片段也是范例项目 FunctionTest 的一部分)让我们可以写出互相呼叫的递归情形：DoubleHello 呼叫 Hello，但 Hello 也可以呼叫 DoubleHello。换句话说，如果用户一直点选 Yes 按钮，讯息就会一直被显示，而且每次点选 Yes 的时候，确认对话框就会再多问两次。在递归的源码里面，一定要有递归的终止条件，以避免相互呼叫直到堆栈溢出(stack overflow)的情形发生。

笔记 函式在呼叫的时候，是使用应用程序内存中的堆栈来记录参数、回传值、局部变量等。如果一个函数持续呼叫自己，成为了无穷循环，堆栈所使用的记忆空间(通常是预先定义好的固定大小，这会由 linker 或者项目设定值来决定)将会很快用完，使程序不正常停止，这种错误情形，就是大家所熟知的堆栈溢出(stack overflow)。而最近几年，也有个很热门的程序开发网站用了这个域名提供给程序人员一个问题交换讨论的平台，这我想应该就不用介绍了：http://www.stackoverflow.com/

虽然在 Object Pascal 里面已经不常使用预先宣告，但有个类似的案例还是很常见。当我们在一个单元文件的 interface 区段宣告程序或函式时，这个宣告就已经被视为预先宣告了，虽然我们宣告的时候并没有用上 forward 这个关键词。实际上我们本来就无法在 interface 区段撰写源码。请记得我们必须在宣告函式的单元文件里面把该函式的实作程序也完成。

递归函数

关于之前我提到的递归，我们先来看一个比较特别的例子(透过两个函式互相呼叫)，然后再观察一个比较传统的递归案例，也就是函数调用自己。使用递归也常用来当做另类的循环实作方法。

从传统的例子看起，假设我们要计算一个数字的多次方，而手边没有适当的函数(其实这个函式在 Object Pascal 的 RTL 里面就有了)。那我们就只能从数学上的定义来分析了，例如 2 的 3 次方，就是 2 乘自己乘三次，也就是 2x2x2.

所以实作这个函式的一个方法，可以是写一个 for 循环，执行三次(或者是几次方，就乘几次)，把乘法的计算结果再乘以要计算的基底数字：

```
function PowerL (Base, Exp: Integer): Integer;
var
    I: Integer;
begin
     Result := 1;
     for I := 1 to Exp do
        Result := Result * Base;
end;
```

另一个替代方案，则是直接乘以该函式的下一次方执行结果，直到下一次方为 0，因为任何数字的 0 次方都是 1，所以我们可以把 0 次方当成是递归调用的终止条件，把这个函式以递归方式实作：

```
function PowerR (Base, Exp: Integer): Integer;
var
    I: Integer;
begin
     if Exp = 0 then
        Result := 1
     else
          Result := Base * PowerR (Base, Exp - 1);
end;
```

这个程序的递归版本执行起来并没有比 for 循环的版本来的快，也没有比较容易读懂。然而像是在分析程序结构时(例如树状结构)，要处理的元素并不是固定的，因此要用一个 for 循环来处理也几乎是不可能的，因此递归在这种案例中，就显得格外有用了。

通常递归函式的源码功能都比较强，但也比较复杂。经过多年以后，递归几乎被遗忘了，和早期的程序相较，新的函式化语言，像是 Haskell, Erlang 和

Elixir 都大量使用递归，并使递归的观念渐渐回归主流。您可以在 FunctionTest 的范例程序中找到两种写法的次方函式。

笔记 范例程序里面的两种次方函式都没有处理负次方的能力，如果把次方的参数传个负数进去，就会造成无穷循环了。同时因为使用了整数型别，也会很快达到该型别的最大值，并造成数值溢位。我撰写这些源码时，也同时保留了这些先天的限制，好让源码能保持简洁。

方法(Method)是什么?

我们已经介绍了要怎么在单元文件的 interface 区段透过 forward 关键词进行预先宣告。在类别的型别当中宣告方法，也可以视为一种预先宣告。

但『方法』到底是什么？方法，是程序或函式与记录或类别相关的特别型态。在 Object Pascal 里面，每当我们为视觉组件处理事件的时候，我们就需要定义一个『方法』，通常是一个程序，但『方法』这个名词，是用来指属于特定类别(或记录)的函式或程序，所以同时包含两者。

以下是一个由 IDE 自动产生的空白方法源码，是属于一个 form 的(其实只要属于类别即可，我们在本书稍后的章节会介绍):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    {here goes your code}
end;
```

参数与回传值

当我们呼叫函式或者程序的时候，我们必须传递正确数目的参数，并且必须确认所有参数的型别都跟宣告的相同。要不然，编译器就会指出这里有错误，例如参数型别不符。以前面的定义的 DoubleIt 函式当例子，它是要求整数作为参数，所以如果我们这样呼叫：

```
DoubleIt(10.0);
```

编译器就会指出这个错误:

```
[dcc32 Error] E2010 Incompatible types: 'Integer' and 'Extended'
```

提示 编辑器在我们撰写程序的时候，会提供我们即将呼叫的函式或程序的所有可能参数清单，这个提示会以一个提示窗口的样式出现，当我们输入某个函式的名字，并输入了一个小括号，这个窗口就会出现了，这个功能称为源码参数(Code Parameters)，它算是 Code Insight 技术的一部分(在其他 IDE 环境中则称之为 IntelliSense)。从 Delphi 10.4 开始，Code Insight 已经是以 LSP(Language Server Protocol)服务器来提供的了。

有些时候，是允许限制型别转换的，例如进行指派，但通常我们应该试着使用特定型别的参数(这称之为强制性的参数参照，我们稍后会进行介绍)。

当我们呼叫函式的时候，我们可以把一个表达式作为参数，不用限制传递一个数值。这个表达式会在执行的时候被运算，运算结果则作为参数被传递到子程序里面去。在单纯的情形下，我们只把变量名字传递进去。在这个例子里，变数的数值则是被复制到该参数里(参数的名称通常和原来的变量名字不一样)。我强烈的建议绝不要把同名的变量传给函式当参数，因为光看上去就很容易让人混淆。

警示 在 Delphi 里，我们不能只检查传给函式的参数顺序，因为这取决于呼叫约定，最常见的案例是由右到左的检视。完整的相关信息请参考: https://docwiki.embarcadero.com/RADStudio/Sydney/en/Procedures_and_Functions_(Delphi)#Calling_Conventions

最后，请记得我们可以用同一个函式或程序的名字，但提供不同版本(这个技术被称之为多载，overloading)，也可以在函式或程序里面忽略一些参数，改以预设的数值来传递(这个技术则被称为预设参数)。这两个在函式跟程序上的重要技术，都会在本章稍后的篇幅加以介绍。

附带回传值离开

我们已经介绍过从函式回传结果的几种不同语法(和 C 语言或者从 C 衍生的其他语言对照)。不光是语法不同，其规则也相异。把回传数值指派给 Result 关键词(或者函式名称)，并不会像执行 return 指令一样结束函式的运行。

Object Pascal 开发者通常受惠于此功能，可以把 Result 当成一个暂时的储存空间，与其这样写：

```
function ComputeValue: Integer;
var
```

```
    value: Integer;
begin
     value := 0;
     while ...
        Inc (value);
     Result := value;
end;
```

我们可以省下暂时变量，直接使用 Result，不管 Result 的内容到函式结束的时候是什么，都会是用来回传该函式最后的数值：

```
function ComputeValue: Integer;
begin
     Result := 0;
     while ...
         Inc (Result);
end;
```

换句话说，也有一些情形，是我们会想要指派好回传值以后，立刻就。离开程序的。例如在特定的 if 判断式里面，如果我们想要指定函式的结果，并且直接中止当下执行中的源码。我们当然应该使用两个独立的函式：先指派结束的数据给 Result 关键词，而且立刻使用 Exit 函式脱离子程序。

假如您记得源码里面的 FlowTest(在”用 Break 和 Continue 指令来中断流程”章节里面介绍过)的项目源码，现在的作法可以视同为重写一些函式，用来替换掉呼叫 Break，然后接着呼叫 Exit。我已经把程序写在以下的程序片段里面了，它是 ParamsTest 这个范例程序的一部分。

```
function CharInString (S: string; Ch: Char): Boolean;
var
    I: Integer;
begin
    Result := False;
    for I := Low (S) to High (S) do
        if (S[I]) = Ch then
        begin
                Result := True;
                Exit;
        end;
end;
```

在 Object Pascal 里面，我们可以把 if 判断式成立时要执行的两个指令，用一个 Exit 指令来取代，Exit 指令可以直接把要当做回传值的数值当成参数，就像 C 语言里面的 return 指令一样。所以我们可以把上面的源码写的更精简，也就可以省下一个 begin-end 的区块了：

```
function CharInString2 (S: string; Ch: Char): Boolean;
var
    I: Integer;
begin
     Result := False;
     for I := Low (S) to High (S) do
         if (S[I]) = Ch then
             Exit (True);
end;
```

笔记 在 Object Pascal 里面，Exit 是一个函式，所以回传值必须括在小括号里面，C 语言的 return 是一个编译器的关键词，所以 return 不用把回传值用小括号当成参数括起。

引用参数(Reference Parameters)

在 Object Pascal 里面，程序和函式传递参数时，包含了传值和传址两种方式。预设的参数传递是以传值的方式进行的：作为参数传递进入子程序的数值或变量，会被复制一份放在堆栈里，子程序将会使用复制出来的这份数值在整个子程序里面使用，所以即使在子程序里面修改了这个复制的数值，呼叫前的参数变量或数值，也都不会有任何改变。（就像我们稍早在函式的参数与回传值章节所介绍的那样）

透过传址的方式来传递参数，则表示不会复制数据到子程序里面去，而是会把当做参数的变量内存地址传进子程序，让子程序直接使用该参数的内存地址，因此子程序只要对这个参数做了任何修改，就会同时修改到呼叫子程序时所传递的变量内容。要使用引用参数，在宣告参数的时候，只要在参数前面加个 var 关键词即可。

这个技术在大多数的编程语言里面都有提供，也因为不用复制变量内容，通常程序执行的速度会快上一些。这个作法在 C 语言里面就没有提供(我们可以使用指标来达成一样的效果)，但在 C++跟其他使用 C 语言语法的语言里

面则也有提供此功能，我们可以透过＆这个符号(意味着传址)来达成传址的目的。以下是使用 var 关键词提供引用参数的写法:

```
procedure DoubleIt (var Value: Integer);
begin
    Value := Value * 2;
 end;
```

在这个案例中，参数既扮演了传递数值让子程序进行计算的角色，也同时把计算完成的结果带回给呼叫它的程序，也就是回传值的角色。如果我们这么写：

```
var
X: Integer;
begin
    X := 10;
    DoubleIt (X);
    Show (X.ToString);
```

变量 X 的内容会在呼叫了 DoubleIt 函式之后变成 20，因为这个函式使用的是引用参数，会直接使用变量 X 的内存地址，直接影响变量 X 的内容。

跟传统的传值参数规则比较一下，引用参数所传递的不只是参数里面的数值，而是把整个参数传过去，所以引用参数不能接受把常数、表达式、函式的回传值，或是类别的属性当成引用参数传递。另一个规则则是不能传递不同型别的变量(必须要先经过转型)。变量的型别跟参数的型别必须完全一致，不然编译器又要抱怨有错了：

```
[dcc32 Error] E2033 Types of actual and formal var parameters must be identical
```

如果我们这么写，编译器就会抱怨而显示上面的错误讯息(下面的程序片段是范例程序 ParamsTest 的片段，不过我已经把它加上批注符号了)：

```
var
C: Cardinal;
begin
    C := 10;
    DoubleIt (C);
```

以引用参数来传递有序型别或记录型别(我们会在下一章介绍)的参数是很有用的。这些型别通常被称为值类型，因为它们的语法含义原本就是传值或者指派数值的作法。

Object Pascal 的字符串跟对象则是完全不同的规则，我们稍后会更深入讨论。对象变量原本就是内存地址，所以当我们把对象当做参数传递的时候，本来就会变动到该对象的原始内容。这些型别比较特殊，通常被称为『参考型别』。

除了标准的引用参数型别(var)之外，Object Pascal 还提供另外一种很特别的参数关键词: out。out 参数不用具备初始值，它只用来把数据回传给呼叫者，除了不用提供初始值以外，其他的规则，out 跟 var 几乎都一样。

笔记 out 参数是为了兼容 Windows 的 COM 模式而被创造出来的。除了在 Windows COM 模式与对象的源码。这可以被用来表示开发人员预期要回传但没有被初始化过的值。

常数参数 (Constant Parameters)

除了引用参数之外，参数还有一种称为常数参数，我们可以在参数前面加上 const 关键词。使用了 const 关键词的参数，在子程序里面会被当成常数，我们不能把值指派给常数参数，编译器可以对参数的传递进行优化。编译器会选择类似引用参数的作法(或者像 C++里面所提到的常数地址)，但这个规则会很像是传值参数，因为原始数值在子程序里面是不能被更动的。

事实上如果我们试着编译以下的源码(是可以的，但我已经在 ParamsTest 项目里面把它标注为批注了):

```
function DoubleTheValue (const Value: Integer): Integer;
begin
    Value := Value * 2; // compiler error
    Result := Value;
end;
```

这里面的错误我们很难一眼看出是怎么回事，错误讯息是：

```
[dcc32 Error] E2064 Left side cannot be assigned to
```

常数参数常被用在字符串型别上，因为在这种情形下，编译器会先暂时停止参考计算器制(Reference Counting mechanism)，以进行优化。使用常数参数最常见的原因，就是常数可以在使用有序型别时可以产生一定的界限。常数参数也不常被用在对象传递上，因为在 Object Pascal 里，如果我们用常数参数形式传递对象的话，被传递进去的会是该对象的参考(内存地址)，而非对象本身。换句话说，编译器不会让我们指派另一个对象到该参数中，但可以让我们呼叫该对象当中的方法来改变对象里面的数据。

笔记

const 参数有一个大家比较不知道的替代作法，就是加入一个 ref 属性，写起来像是"const [ref]"。这个属性会强制编译器常数参数用传址的方式来处理该参数，不然的话，编译器默认会以传值的方式来处理它，有时在不同的 CPU 跟操作系统上，则会依照该参数所占的内存空间而采用传址的方式来处理，

函式的多载 (Function Overloading)

我们常常会需要提供两个类似的函式，让其中的参数跟实作的源码各有些不同。在传统的写法里，我们只能乖乖的用另一个函式名称来制作第二个函式。但现代程序的写法，则可以用同一个函式名称，透过不同的参数，对同一个名称的函式进行***多载***。

多载的含义很简单：编译器允许我们使用同一个函式或程序的名称制作多个函式或程序，只要其中的参数不同即可。透过检查参数，编译器就可以判断我们要呼叫的是哪一个版本的函式或程序。

我们可以看一下以下这一些函式，它们是从 Object Pascal RTL (Run-Time Library, 运行时间函式库)当中的 System.Math 单元文件里面节录出来的：

```
function Min (A,B: Integer): Integer; overload;
function Min (A,B: Int64): Int64; overload;
function Min (A,B: Single): Single; overload;
function Min (A,B: Double): Double; overload;
function Min (A,B: Extended): Extended; overload;
```

当我们呼叫 Min(10, 20) 的时候，编译器会判断我们是呼叫第一个函式，所以回传值也会是整数。

多载有两个基本规则：

- 多载的每个函式(或程序)宣告，结尾都必须加上 overload 这个关键词。
- 在多载的每个函式(或程序)宣告中，参数的数量或型别必须要有不同，

参数名称则无所谓，因为名称在呼叫过程中不会被特别指出，而回传值并不会被拿来作为两个多载函式的判别，换句话说，多载函式(或程序)当然可以回传相同型别的数据。

笔记 规则上有一个例外状况，就是我们不能以回传值来做为多载函式(或程序)的区别依据，且用 Implicit 或 Explicit 转换运算符，这会在第五章里面介绍。

以下是 ShowMsg 程序的三个多载版本，我已经把他们放在 OverloadTest 范例程序里面了(这个应用程序用来展示多载与预设参数):

```
procedure ShowMsg (str: string); overload;
begin
    Show ('Message: ' + str);
end;
procedure ShowMsg (FormatStr: string; Params: array of const); overload;
begin
    Show ('Message: ' + Format (FormatStr, Params));
end;
procedure ShowMsg (I: Integer; Str: string); overload;
begin
    ShowMsg (I.ToString + ' ' + Str);
end;
```

这三个程序都会用讯息框来显示一个字符串，分别以不同的方式对字符串做格式化，以下就是三个不同呼叫这些程序的源码：

```
ShowMsg ('Hello');
ShowMsg ('Total = %d.', [100]);
ShowMsg (10, 'MBytes');
```

而结果分别如下:

```
Message: Hello
Message: Total = 100.
Message: 10 MBytes
```

提示 IDE 的 Code Parameters 技术对多载的程序与函式处理的非常好。当我们在编辑器上面输入了函式名称，且输入了小括号，所有同名的多载函式就会一起被自动列出。当我们输入参数的时候，Code Insight 技术则会依照我们

输入的参数型别自动判别符合的多载函式，隐藏其他不符型别的。所以在撰写源码的时候就轻松多了。

如果我们试着呼叫某个函式，然后传递完全不合其中任何一个多载版本的参数进去会怎样？当然，我们会得到编译器给的错误讯息，假设我们要呼叫：

```
ShowMsg(10.0, 'Hello');
```

我们就会得到以下这个很特别的错误讯息:

```
[dcc32 Error] E2250 There is no overloaded version of 'ShowMsg' that can be called with these
arguments
```

事实上，多载函式的每个多载版本都必须标注上 overload 的关键词，所以我们不能试着多载相同单元文件里，一个已存在但没有标注 overload 关键词的函式，如果我们试着这么做，就会得到以下的错误讯息：

```
Previous declaration of '<name>' was not marked with the 'overload' directive.
```

然而，我们可以在别的单元文件里面建立一个新的函式，然后把这个新单元文件当成命名空间来处理。在这个情形下，我们就不是新增一个多载版本函式，而是把原来的函式用一个完全新版的同名函式来取代了，原来的函式会被编译器视为被隐藏了(当然还是可以直接把该单元文件的名称作为前述字符串，来呼叫原始版本的函式)，这也就是为什么编译器不能只从参数来判断，就决定用哪一个多载版本，但编译器也会试着用唯一相符的多载版本函式来处理，但仍会在参数型别不相符的时候发出错误讯息。

多载与呼叫混淆（Ambiguous call）

当我们呼叫一个多载函式时，编译器通常会试着找到一个相符的多载版本，或者在找不到相符的多载版本时发出错误讯息(就像我们刚刚看过的例子一样)。

但还有第三种情况：假设编译器能够对函式的参数进行型别转换，则对单一个函数调用就有可能有不同的转换。当编译器发现它可以使用的多种多载版本函式，而其中并不包含完全相符的参数型别，则此时编译器发出的错误讯息就会是『此函数调用发生混淆』。

这种情形并不多见，我得建立一个不合理的范例来说明这个情形，但观察这个案例是值得的(虽然实际撰写程序的时候发生的机率非常之低)。假设我们决定要实作两个多载版本的函式来处理整数跟浮点数的加总：

```
function Add (N: Integer; S: Single): Single; overload;
begin
    Result := N + S;
end;
function Add (S: Single; N: Integer): Single; overload;
begin
    Result := N + S;
end;
```

这些函式都在 OverloadTest 范例项目里面。现在我们可以呼叫它们，指派两种不同的参数给它们：

```
Show (Add (10, 10.0).ToString);
Show (Add (10.0, 10).ToString);
```

然而，事实上通常一个函式可以在参数经过型别转换后接受该参数，例如一个函式要求浮点数型别，但可以在整数资料经过型别转换以后接受它，所以我们可以写成：

```
Show (Add (10, 10).ToString);
```

编译器会使用第一个多载版本，但第二种多载版本也会被呼叫。如果不知道我们的要求(或者知道呼叫该版本时，仍有其他多载版本可能导致这种问题时)，则以下错误讯息就会被回报出来：

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'Add'
    Related method: function Add(Integer; Single): Single;
    Related method: function Add(Single; Integer): Single;
```

提示 在 IDE 的错误讯息面板里，我们可以看到上述讯息的第一行，在该行错误讯息的左边有个加号，点开它，就可以看到其他的完整错误讯息了。

如果我们在实际上写程序的时候遇到上面这个情形，而我们真的需要呼叫这个函式，我们可以自己先做型别转换来解决这个问题，并让编译器可以判别我们要呼叫的是哪一个多载版本：

```
Show (Add (10, 10.ToSingle).ToString);
```

实际会发生混淆呼叫的情形，会是我们使用 variant 这种型别作为参数的时候，因为这个参数型别可以包含许多种不同特定型别，我们在本书稍后的篇幅会加以讨论。

预设参数 (Default Parameters)

另一个跟多载相关的技术，是在函式或程序的参数里面可能先设定了预设的参数值，所以在我们呼叫函式的时候，写不写该参数都可以被编译器接受。如果我们没有指定该参数，编译器就会以默认值传给该函式或程序。

我们来看一个例子(这个例子仍旧是 OverloadTest 范例项目的一部分)，我们可以定义以下的程序来封装对 Show 函式的呼叫，提供两个预设参数：

```
procedure NewMessage (Msg: string; Caption: string = 'Message'; Separator: string = ': ');
begin
    Show (Caption + Separator + Msg);
end;
```

藉由上述的定义，我们可以用以下的方式来呼叫这个函式：

```
NewMessage ('Something wrong here!');
NewMessage ('Something wrong here!', 'Attention');
NewMessage ('Hello', 'Message', '--');
```

会得到以下的输出结果:

```
Message: Something wrong here!
Attention: Something wrong here!
Message--Hello
```

请注意，编译器没有建立任何特别的源码来协助预设参数，也不用为这个函式制作任何多载版本。没有输入的参数，编译器就自动的加入到呼叫函式的源码里去了。只有一个简单的规则要遵循：我们不能跳过任何一个参数，例如我们不能只提供第一跟第三个参数，然后只跳过第二个。

当然定义上还是有一些其他的规则，呼叫上也有些要注意的地方:

- 在呼叫时，我们必须从最后一个参数开始跳过，如果我们要跳过参数，就得从最后面一个一个来。
- 在定义上，有默认值的参数必须放在整列参数的最后面。
- 默认值必须是常数，显然的，这也限制了预设参数能够使用的型别，像

是可变数组、interface 就不能以 nil 之外的值来当成其默认值，而 record 更是完全不能当成预设参数。

- 有默认值的参数必须以传值形式进行，或者当成常数(const)。传址(var)的参数是不能有默认值的。

同时使用预设参数和多载，我们就有更多机会可以让编译器昏头转向，例如发出混淆呼叫的错误提示，就像我们前一节介绍过的那样。举例来说，如果我在前一个范例里面加入以下的新版多载函式：

```
procedure NewMessage (Str: string; I: Integer = 0); overload;
begin
    Show (Str + ': ' + IntToStr (I))
end;
```

这时编译器并不会发出错误通知，因为这是个合法的定义。然而以下这个呼叫：

```
NewMessage ('Hello');
```

则会导致编译器提出错误讯息：

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'NewMessage'
    Related method: procedure NewMessage(string; string; string);
    Related method: procedure NewMessage(string; Integer);
```

请注意这个错误是发生在新的多载版本出现前，编译完全正确的一行指令上面。实务上我们应该不可能呼叫 NewMessage 这个程序，而只提供一行字符串作为参数，或者提供一个字符串参数与整数参数做为其默认值。当有类似的疑虑时，编译器就会要求程序人员把其意图表示的更明确一点。

内嵌源码（Inlining）

在 Object Pascal 当中内嵌函式和方法是低级语言的功能，透过这个作法，可以得到显著的优化。通常当我们呼叫一个方法，编译器会建立一些源码，让我们的程序进入一个新的执行点。这表示设立了一个堆栈框架，并开始处理一些程序，可能需要一些机器指令。然而，我们执行的方法可能非常简短，甚至呼叫一个方法可能只是设定或者回传一些私有字段。

在这种情形下，复制这些源码到实际呼叫它们的地方，就非常有用了，可以避免堆栈框架的设定或其他相关事情的衍生。去除了这些额外的动作之后，我们的程序当然可以执行更为快速，特别是当这个函式在循环当中被呼叫了

上千上万次的时候，每次省一点时间，整个执行过程所节省的时间就更为可观了。

对于一些很小的函式来说，回传值的源码甚至更小，这些源码可能比呼叫一个函式本身所耗用的空间来的更小。然而，请注意到，假如一个稍大的函式是内嵌的，而这个函式在我们的源码当中很多不同的地方都有呼叫到这个函式，我们的源码可能变得比较大，而这对于执行档来说，会是不必要的空间增加。

在 Object Pascal 里，我们可以要求编译器把一个函式(或方法)进行内嵌，内嵌的关键词是 inline，只要在宣告函式或方法的源码之后加上 inline 这个关键词即可。不用在定义函式的地方重复这个关键词。请牢记，inline 关键词对编译器来说只算是个提示，编译器仍旧可能判断该函式(或方法)以内嵌方式编译对整个程序来说并没有好处，而直接把这个关键词给忽略不管(不会提供任何警告讯息喔)。编译器也可能在完成分析所有源码之后，依照 $INLINE 开关的状态，把部分函式或方法给内嵌，但不会把所有函式完全都内嵌。这个开关可能是以下三种不同的设定值之一(请注意，这个功能是独立于编译器优化的设定之外的喔)：

- 默认值，{$INLINE ON}，对有标注要进行内嵌的函式或方法全部进行内嵌。
- {$INLINE OFF}，我们可以停止所有内嵌的处理，可以针对整个程序、部分程序或者特定的函式，除非有出现 inline 关键词的函式，否则其余全部内嵌的功能都会被停止。
- {$INLINE AUTO}，编译器通常会对我们标注 inline 关键词的函式进行内嵌，也会把一些很短的函式自动内嵌，使用这个设定的时候务必小心在意，因为可能让我们的执行程序档案变大。

在 Object Pascal 的运行时间函式库里就有许多被标注 inline 关键词的函式。举例来说，Math 单元里的 Max 函式就被这么定义的：

```
function Max(const A, B: Integer): Integer; overload; inline;
```

为了实地测试内嵌这个函式的效果，我在 InliningTest 项目里面，写了以下的循环：

```
var
    Sw: TStopWatch;
     I, J: Integer;
 begin
       J := 0;
```

```
Sw := TStopWatch.StartNew;
for I := 0 to LoopCount do
    J := Max (I, J);
Sw.Stop;
Show ('Max ' + J.ToString +
    ' [' + sw.ElapsedMilliseconds.ToString + '] ');
```

在这段源码里面，System.Diagnostics 单元的 TstopWatch 记录，这是一个会持续监控从 Start 指令(或者 StartNew 指令)被下达之后所耗费的时间(或者系统 CPU 的周期-ticks)，呼叫 Stop 之后，这个监控就会停止。

这个窗体有两个按钮，两个都会呼叫同一个函式，但一个在呼叫的时候停用了 inline 开关。请注意，我们必须使用 Release 组态来编译，才能看出两者之间的不同(因为内嵌是属于 Release 优化的一部分，Debug 组态不会对它有反应)。在我的计算机上面进行了 2 千万次(这个数字是 LoopCount 的常数)的互动以后，得到了以下的数字:

```
// on Windows (running in a VM)
Max on 20000000 [17]
Max off 20000000 [45]
// on Android (on device)
Max on 20000000 [280]
Max off 20000000 [376]
```

我们该怎么解读这些数据？在 Windows 系统上，内嵌程序快了将近一倍的速度，而在 Android 系统上面，内嵌程序也快了将近 35%。然而在 Android 装置上面执行的速度比计算机慢上许多(这个差距可是一个数量级的差距)，所以我们在 Windows 上面节省了 3 秒钟，而在 Android 装置上，这个内嵌的优化省下了将近 10 秒钟。

同样的程序还进行了第二个类似的测试，是透过 Length 函式来进行。这个函式有编译魔术在里面，会因为内嵌的优化而有很大的不同。再一次，进行内嵌的版本在 Windows 跟 Android 上面都有非常明显的变快：

```
// on Windows (running in a VM)
Length inlined 260000013 [11]
Length not inlined 260000013 [40]
// on Android (on device)
Length inlined 260000013 [401]
Length not inlined 260000013 [474]
```

这是用来作第二种测试的循环：

```
var
    Sw: TStopWatch;
    I, J: Integer;
    Sample: string;
begin
    J := 0;
    Sample:= 'sample string';
    Sw := TStopWatch.StartNew;
    for I := 0 to LoopCount do
       Inc (J, Length(Sample));
    Sw.Stop;
    Show ('Length not inlined ' + IntToStr (J) +
            ' [' + IntToStr (Sw.ElapsedMilliseconds) + '] ');
end;
```

Object Pascal 的编译器并不会清楚定义一个用内嵌或特定结构排除被内嵌的方式(for 或是 while 循环，条件式指令)建立的函式，编译出来的源码要限制多大的 Size。然而因为把一个大的函式进行内嵌，得到的优点跟让我们的程序暴露在风险的缺点相较之下，得到的少，失去的多，所以我们应该避免这么做。

其中一个限制，是这个函式或方法不能参考到任何定义在同一个单元文件的 implementation 区段的识别符号(例如型别、全局变量或函式)，而他们在呼叫的源码中也不能被使用到。然而，如果我们呼叫的是一个区域函式，则内嵌这个函式就不会有任何问题了，编译器会遵照我们的要求，把这个函式进行内嵌。

内嵌函式也有一个缺点，就是会使得单元文件要被编译的更频繁，当我们修改了一个内嵌函式，所有呼叫到这个函式的单元文件就需要重新编译才能使用修改后的源码。在同一个单元文件里面要呼叫内嵌函式的话，就必须要在内嵌函式的源码之后才能呼叫它，所以我们要使用内嵌函式的话，最好把它写在整个单元文件的 implementation 区段中，作为第一个函式。

笔记 Delphi 使用单次编译过程的编译器，所以我们无法使用还没编译源码的函式。

如果是在不同的单元文件，我们就必须把包含内嵌函式的单元文件加入到我们的 uses 区段，即使我们没有直接呼叫这些方法也一样。假设我们的 A 单元文件呼叫了 B 单元文件里面的一个内嵌函式，如果这个函式又呼叫了 C 单元文件里面的另一个内嵌函式的话，我们的 A 单元文件就必须也要引入 C 单元文件。不然的话，我们会在编译的过程看到编译器的警告讯息，提示无法内嵌该函式，因为缺少了一些被参考的单元。相关的作用就是如果单元之间相互参考，则这里面的所有内嵌函式都不会真的被内嵌。

函式的进阶功能

如果目前我们所介绍的已经包含函式相关的核心功能，那么还有几个进阶的功能也值得一提。如果您是软件开发的初学者，然而您可能已经想跳过本章剩下的篇幅，直接去看下一章了。

Object Pascal 呼叫函式的约定(Conventions)

不论何时，当我们的程序呼叫一个函式，两边都必须同意实际上参数从呼叫端被传递到被呼叫端的实务作法，这些我们称之为呼叫约定。通常，呼叫函式时，会把参数(如果有回传值的话也透过同样的方式传递)透过内存堆栈区进行传递。然而，参数或回传值在内存堆栈里面的顺序，会随着编程语言与操作系统的不同而有差异。多数编程语言都使用各自多种不同的呼叫约定。

很久以前，32 位版本的 Delphi 提出了一种传递参数的新方式，称之为 fastcall，在任何时候，三个以内的参数可以直接透过 CPU 的缓存器(Register)进行传递，这个方式会使得函数调用的速度快上许多。Object Pascal 预设使用 fastcall 这种方式作为呼叫约定，因此也需要使用到 register 这个关键词。

Fastcall 是预设的呼叫约定，而使用这个约定呼叫的函式就无法兼容于外部函式库，例如 Win32 版的 Windows API。Win32 API 的函式必须使用 stdcall(Standard call: Windows 的标准呼叫)作为呼叫约定，而原始的 Pascal 的呼叫约定则是 Win16 API 的 cdel，也就是 C 语言的呼叫约定。这些不同的呼叫约定在 Object Pascal 里面全都兼容，但是我们不常使用跟默认值不同的呼叫约定，除非我们需要使用到透过不同编程语言撰写的函式库，例如系统函式库。

我们需要使用到不同于预设呼叫约定的情形，就是当我们需要使用到一个操作系统的原生 API 时，这时我们会被要求使用该平台专属的呼叫约定。即

使是 Win64，使用的呼叫约定也跟 Win32 不同，所以 Object Pascal 支持很多不同的选项，在这里我们先不详述。而行动作业平台常常愿意揭露类别，而不揭露原生函式。即使在这种情形下，我们也必须考虑到尊重该平台的呼叫约定的问题。

程序型别（Procedural Types）

另一个 Object Pascal 独特的功能，则是程序型别的存在。这是很进阶的编程语言问题，只有很少数的程序人员会常用到它。然而，既然我们在后面的章节会讨论其他的议题(例如方法指标，在整体环境的技术上大量使用，用以定义事件处理程序、以及匿名方法等技术)，那我们就该花点时间在这里浏览一下这个主题了。

在 Object Pascal 里(不是在传统的 Pascal 语言里面喔)，存在着程序型别的概念(这个概念像极了 C 语言里面函式指针的概念-在 C#跟 JAVA 里面都舍弃了这个功能，因为这个功能会和全局变量紧密结合)。

程序型别的宣告会需要指定一连串的参数以及回传值来定义函式。举个例子，我们可以宣告一个新的程序型别，需要一个整数参数以传址形式传入：

```
type  TIntProc = procedure (var Num: Integer);
```

这个程序型别兼容于具备完全符合的参数的程序(用 C 语言的用语来说，就是函式特征相同)，以下就是一个兼容于前述型别的程序：

```
procedure DoubleIt(var Value: Integer);
begin
    Value := Value * 2;
end;
```

程序型别通常有两个要求：我们可以以程序类型声明变量，或者把程序型别当成参数型别传给另一个函式。假设前述的型别跟程序宣告属实，我们可以写下这些源码：

```
var
    IP: TIntProc;
    X: Integer;
begin
    IP := DoubleIt;
    X := 5;
```

```
    IP (X);
end;
```

这段源码跟较短版本的源码效果相同：

```
var
    X: Integer;
begin
    X := 5;
    DoubleIt(X);
 end;
```

第一个版本明显的比较复杂，所以我们什么时候，为什么要用它？某些情形下，能够决定要呼叫哪个函式，并实际上稍晚再呼叫它，是很有权力的。要找到能够达成相同目标的范例是完全可能的。然而，我希望我们能看一个比较单纯的例子，叫做 ProcType。

这个范例是以两个程序作为基础，一个函式用来把参数值算出两倍的值为何，这个函式我们已经看过很多次了。范例基于两个函式，其中之一用来把参数的值加倍。另一个函式则是用来把参数数值的数字变成三倍，所以这个名称就改为”TripleIt”:

```
procedure TripleIt(var Value: Integer);
begin
     Value := Value * 3;
end;
```

虽然没有直接呼叫这些函式，其中一些已经被存成程序型别的变量了。这个变量会在用户选择一个核取方框(checkbox)时有一些改变，目前的程序是以传统方式仿真用户点击按钮。这些程序使用了两个全局变量(被呼叫的函式，以及当前的现值)，所以这些数值都会被经年累月的保留下来。以下是完整的源码，请以真实程序的完整定义把这些数据封装起来吧：

```
var
    IntProc: TIntProc = DoubleIt;
    Value: Integer = 1;
procedure TForm1.CheckBox1Change(Sender: TObject);
begin
     if CheckBox1.IsChecked then
        IntProc := TripleIt
     else
          IntProc := DoubleIt;
      end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    IntProc (Value);
    Show (Value.ToString);
end;
```

当用户点击复选框(checkbox)，改变了组件的状态，所有按键点击时都会呼叫一个实际的程序，所以如果我们点了按钮两次，改变了选取状态，然后再点按钮两次，我们会先把参数成倍两次。然后再把现值先变成两倍，再变成三倍，如下所示：

```
2
4
12
36
```

另一个使用到程序型别的实例，是当我们要把一个函式传递给操作系统时，例如 Windows(通常在操作系统中都有所谓的 callback 函式)。在这个章节要开始的时候有提到过，Delphi 程序人员使用方法指针来处理程序化型别(我们会在第 10 章介绍)以及匿名方法(我们在第 15 章加以介绍)：

笔记 最常用的面向对象机制，就是取得一个稍后进行绑定的函式(是一个可以在运行时间进行变更的函式)，也就是虚拟方法。当虚拟方法在 Object Pascal 里面非常平常，Object Pascal 的程序化方法并不寻常。这些技术的基础，都相当的类似，虚拟函式跟多型将会在第 8 章进行说明。

宣告外部函式

系统程序的另一个重要元素，是由外部宣告的函式来承担的。最早使用到外部函式库连结的源码，是以汇编语言撰写的。外部连结函式变成非常普及，始于 Windows 程序里面呼叫动态链接函式库(DLL, Dynamic Link Library)。外部函式的宣告，表示具备能够呼叫一个函式，该函式未必由编译器或链接程序制作或掌控，但必须具备能够加载外部动态函式库，并执行当中函式的能力。

笔记 无论我们什么时候在 Object Pascal 源码里面呼叫某个平台专用的 API，我们都会失去把该程序编译成其他平台的能力。唯一的例外，是我们把呼叫特定平台的源码透过$IFDEF 编译开关包起来。

举例来说，我们在 Delphi 源码里面呼叫了 Windows API 的函式。如果打开 Winapi.Windows 这个单元文件来看，我们会发现里面有很多的宣告跟定义，长得像这样：

```
// forward declaration
function GetUserName(lpBuffer: LPWSTR; var nSize: DWORD): BOOL; stdcall;
// external declaration (instead of actual code)
function GetUserName; external advapi32
        name 'GetUserNameW';
```

我们很不常有机会需要写上面这样的源码，因为他们已经被列在 Windows 单元文件以及其他系统单元文件里面了。我们只有在需要使用到自定的 DLL 里面的函式时，或者需要呼叫一些系统没有宣告、转译成 Pascal 宣告语法的 Windows API 时，才会需要写这样的宣告。

上述的宣告，是意指在 advapi32 这个动态链接函式库里面，存在着 GetUserName 这个函式(advapi32 是一个宣告过的函式库名称，完整档名是 advaoi32.dll)。GetUserName 提供了 ASCII 与 WideString 两种版本，而在做外部函数声明的时候，我们可以自己指定让外部函式在我们的源码里面用另一个不同的函式名称。

延迟加载动态链接函式库的函式

在 Windows 操作系统里，提供了两种方式来呼叫 Windows SDK 与其他 API 的函式：我们可以让应用程序加载器把所有外部函式都先行解译，或者在我们需要的时候才自行加载特定的函式。

这样的源码并不难写(就像我们前一节看过的): 我们要做的事，就是做个外部函数声明而已。然而如果函式库，或者我们要呼叫的函式都没能找到，我们的程序就会无法在没有提供该函式库的操作系统上面执行。

动态加载提供了很大的弹性，我们得自己处理加载函式库的程序，使用的是 GetProcAddress API，来找到我们想呼叫的函式，然后把该函式的指标转换成适当的型别之后就能呼叫它了。这样的源码是相当繁复且容易出错的。

这就是为什么 Object Pascal 编译器跟链接程序支持 Windows 操作系统层级的功能，当需要使用该函式库的时候才延迟加载该函式，且已经用在部分 C++编译器上会大受欢迎了。这个宣告的目的并不是为了避免 DLL 的错误

载入，反正这情形总是会发生的，而是为了让 DLL 里面的特定函式得以被延迟载入。

基本上我们写程序的方法跟传统执行 DLL 的函式非常相似，但函式的地址早在程序第一次被呼叫的时候就被解析出来了，并不是在程序加载时就解析好的喔。这表示如果当时函式无法被使用，我们就会看到一个运行时间例外发生了(EExternalException)。然而，我们通常可以检查我们要加载的目前操作系统的版本，或者特定函式库的版本，然后决定是否要呼叫这个函式。

笔记 如果我们想要使用比例外处理更简单、明确的方式来处理这种情形，我们可以直接挂载(hook)到延迟加载函式的错误处理机制。Allen Bauer 在他的部落格上面有很精辟的说明:
https://blog.therealoracleatdelphi.com/2009/08/exceptional-procrastination_29.html

从 Object Pascal 语言来看，对于外部函数声明的唯一不同，就是宣告方法，以前我们可能会写成:

```
function MessageBox;
        external user32 name 'MessageBoxW';
```

但我们现在可以写成(这也是从 Windows 单元文件里面节录出来的):

```
function GetSystemMetricsForDpi(nIndex; Integer; dpi: UINT): Integer;
        stdcall; external user32 name 'GetSystemMetricsForDpi' delayed;
```

在运行时间，考虑到这个 API 是在 Windows 10, 1607 版之后的版本才有的，在首次加载的时候，我们可能会想把源码写成这样：

```
if (TOSVersion.Major >= 10) and (TOSVersion.Build >= 14393)    then
begin
     NMetric := GetSystemMetricsForDpi (SM_CXBORDER, 96);
```

比起旧版的 Windows 里，同样功能的程序写法，这段程序已经是少到不能再少了。(真的，译者也举双手双脚同意)

另一个相关的观察，则是我们可以在建立我们自己的 DLL，以及从 Object Pascal 里面呼叫这些 DLL 的函式时使用同样的机制。我们可以提供单一的执行档，让这个执行档能够和同一个 DLL 的不同版本进行绑定，这样我们就可以透过延迟加载函式的功能来使用该函式的新版本了。

05:数组与记录

我们在第二章介绍数据型别的时候，曾经提到过 Object Pascal 同时存在数据型别与型别建构子。型别建构子的简单例子是列举型别，会在本章介绍。

型别定义的力量，来自更多高深的机制，像是数组、记录与类别。在本章的篇幅中，我们先介绍前两个，他们的本质可以追溯到 Pascal 语言被定义出来的初期，但经过了这么多年以后，也已经有所改变(变得更强了)。现在的版本几乎已经跟原来的定义完全不同了，只剩名字还被留着。

到本章的结尾，我也会稍微提到一些 Object Pascal 进阶的数据型别，像是指标。然而自定数据型别的强大，我们要到第七章才能体会，在第七章里面，我们会开始深入类别与面向对象程序设计。

数组数据型别

数组型别定义了以清单列出特定型别的表示法。这些列表当中可以存放固定数量的元素(静态数组)，也可以存放变动的元素数量(动态数组)。我们通常以方括号括住一个索引值来存取数组中的特定元素。方括号也用来代表固定数量元素数组的数量。

Object Pascal 语言支持不同的数组型别，从传统的静态数组到动态数组。我们推荐使用动态数组，尤其是在使用行动版的编译器时。我们先介绍静态数组，然后稍后再来看动态数组吧。

静态数组 (Static Arrays)

传统的 Pascal 语言数组都是静态的，也就是其元素的数量在宣告的时候就确定了，底下的程序片段就是个范例，在这个范例中，定义了一个 24 个整数的清单，用来表示一天 24 小时的温度：

```
type
  TDayTemperatures = array [1..24] of Integer;
```

在这个传统的数组定义里，我们可以透过方括号来使用次范围型别，实际用两个有序型别的常数值来定义一个新的次范围型别。这个次范围指出数组中有效的索引值。既然我们定义了该数组索引的最大值与最小值，那么这个索引就不必非得从 0 开始了，就像在 C, C++, Java 以及其他大多数的语言一样(从 0 开始作为起始点的数组也已经在 Object Pascal 里面普及了)。注意，在 Object Pascal 里面，索引值可以是数字，或者其他有序型别的值，像是字符、列举型别等等。非整数型的索引目前还是很少见。

笔记

仍有许多编程语言，像 JavaScript，特别倚重关系型数组(associative arrays)。Object Pascal 的数组限制索引必须是有序型别，所以我们不能用字符串当成索引值。在 RTL 里面有提供了 Dictionary 跟其他类似的数据结构可以用，我会在本书的第三部分，介绍 Generic 的篇幅里面介绍他们。

既然数组索引值是以次范围为型别，编译器就可以检查他们的范围了。使用不合法的常数次范围值会导致编译错误，而超过范围的索引值则会导致运行时错误，不过，这也得要设定了对应的编译选项才会发生。

笔记

次范围检查的选项在设定页的 Compiling 页签里的 Runtime errors 群组里面。我们可以从 IDE 的 Project Options 对话框找到(在项目名称上面点选鼠标右键，就会出现 Project Options 选项了)。在第二章的『次范围型别』那一小节我们已经提过啰。

使用以上的数组定义，我们可以设定一个名为 DayTemp1 的变量，型别就用 TDayTemperatures(我已经写好在范例项目 ArrayTests 里面，以下就是该项目的源码节录):

```
type
   TDayTemperatures = array [1..24] of Integer;
var
   DayTemp1: TDayTemperatures;
begin
        DayTemp1 [1] := 54;
        DayTemp1 [2] := 52;
        ...
        DayTemp1 [24] := 66;
        // The following line causes:
        // E1012 Constant expression violates subrange bounds
       // DayTemp1 [25] := 67;
```

这么一来，我们当然可以用标准的方法来处理数组了，也就是 for 循环。这个范例是使用循环来显示一天当中的所有温度数字：

```
var
    I: Integer;
begin
    for I := 1 to 24 do
        Show (I.ToString + ': ' + DayTemp1[I].ToString);
```

当然，这段源码执行不会有问题，但因为它的数组范围是写死的(从 1 到 24)，这种写法还不尽理想。因为数组的定义会随时间改变，所以我们可能会想要使用动态数组。

数组的大小跟边界

当我们在处理数组的时候，记得！永远都要用标准函式 Low 跟 High 检查它的边界，这两个函式会回传上下两个边界值。(译者也这么认为，因为这几年越来越常碰到初学者，甚至已经写了几年的程序人员，完全不管数组边界值，直接就抓数据，然后就发生数组超出边界的错误，在手机就会闪退，在桌面应用程序就会发生不断出现错误) 请永远在存取数组内容之前，先检查该索引值是不是在数组的合法范围之内！尤其是在循环里面处理数组数据时，更需要逐一检查，因为循环可能从 0 到数组数量-1，或者从 1 到数组数量，或者任何次范围的定义，在执行过程中，什么时候会出问题很难预料，所以务必一一检查。

即使我们在程序写好之后，还需要更改数组索引的范围，Low 跟 High 这两个函式仍旧是有效的，但如果我们把范围写死，我们就得在每次数组范围有异动的时候，修改数组使用的源码了。Low 跟 High 这两个函式会让我们不用花太多时间在维护数组的源码，并使源码可靠性更高。

笔记 顺便一提，对静态数组使用 Low 跟 High 函式，在运行时间完全没有多花时间。因为它们是在编译阶段就已经完成了解析，所以不用任何额外的函式来处理。类似的编译阶段解决的作法，在其他系统函式里面也常出现喔。

另一个相关的函式是 Length，这个函式会回传数组的元素数量，我把上述三个函式整合在下面的范例源码里面，用来显示当天的平均温度：

```
var
    I: Integer;
```

```
    Total: Integer;
begin
     Total := 0;
    for I := Low(DayTemp1) to High(DayTemp1) do
        Inc (Total, DayTemp1[I]);
    Show ((Total / Length(DayTemp1)).ToString);
```

上面的源码也是 ArraysTest 范例项目的一部分。

多维度静态数组

数组可以包含超过一个以上的维度，以矩阵或者方块的形式来呈现，不一定是一维的线性列表。以下是两个简单的定义：

```
type
    TAllMonthTemps = array [1..24, 1..31] of Integer;
    TAllYearTemps = array [1..24, 1..31, 1..12] of Integer;
```

我们可以这样来存取里面的元素：

```
var
    AllMonth1: TAllMonthTemps;
    AllYear1: TAllYearTemps;
begin
    AllMonth1 [13, 30] := 55; // hour, day
    AllYear1 [13, 30, 8] := 55; // hour, day, month
```

笔记

静态数组会立即使用许多内存空间(例如使用掉堆栈里的内存)，这些浪费是可以被避免的。AllYear1 变量需要使用 8,928 个整数，以每个整数 4 bytes 来计算，就将近 35KB 了。不管在全局内存空间或者堆栈里面(如上例的程序，就会耗用堆栈的内存)，预先配置这么大的空间都是不对的。相反地，动态数组使用的是 heap 的记忆空间，而且在内存的配置跟管理上弹性也都更大。

假设这两个数组型别是以相同的核心型别建置，我们最好使用前例的资料型别，可以写成：

```
type
    TMonthTemps = array [1..31] of TDayTemperatures;
    TYearTemps = array [1..12] of TMonthTemps;
```

这样的宣告颠倒了前例的顺序，但也允许两个变量之间整个区块的指派动作。我们看一下可以怎么指派独立的数值：

```
Month1 [30][14] := 44;

Month1 [30, 13] := 55; // day, hour

Year1 [8, 30, 13] := 55; // month, day, hour
```

使用中介型别的重要性，只基于数组中元素相同时，元素型别可以兼容的特性而来(这当中的型别是完全相同的)。而当元素型别不同的时候，这个特性就不存在了。这个型别兼容的规则，对于 Object Pascal 的所有型别都成立，只有少数情况例外。

举例来说，以下的源码会复制一个月的温度到该年的第三个月份：

```
Year1[3] := Month1;
```

反之，一个类似的指令如果是套用到独立的数组时(这数组跟温度的数组型别并不相同):

```
AllYear1[3] := AllMonth1;
```

就会引发错误：

```
Error: Incompatible types: 'array[1..31] of array[1..12] of Integer' and 'TAllMonthTemps'
```

像我提过的，静态数组在内存管理的问题上做出了牺牲，特别是当我们想要把它作为参数进行传递，或者只配置一个大数组的一部分时。再者，我们无法在执行过程中，对仍处在合法生命周期的数组变量进行重新配置。这也是为什么使用动态数组占有比较多优势的原因，即使动态数组需要使用一些额外的管理机制，例如对内存的管理。

动态数组

在传统的 Pascal 语言里，数组是固定大小的，当我们宣告了数组的元素数量以后，这个数组可以储存的元素数量就被固定了。Object Pascal 也提供了一个原生、直接的动态数组功能。

笔记 『直接的动态数组功能』指的是相对于使用指针与易失存储器配置等技术来得到类似的效果，那样做程序会很复杂，而且容易出错。顺带一提，动态数组也是大多数现代编程语言唯一的构筑风格。

动态数组是动态配置，并且对参考进行计数(这个作法会使得传递参数的时候快上许多，且只有参考会被传递，而不是把完整的数组复制一份传过去)。当我们完成处理时，我们可以透过把数组变量指向 nil 或者把数组的长度设定为 0 这两种作法来把数组内容清除掉，而动态数组因为是使用参考计数在管理内存的，编译器就会帮我们把内存自动清除掉了。要记得数组的每个元素是有使用内存的：如果数组中掌控了其他内存空间的地址(例如对象的参考地址)，我们就得在释放数组之前，确保数组中每个元素指向的内存空间确实有被释放掉。

透过动态数组，我们宣告数组型别时，就不用指定元素数量了，等到我们需要使用的时候，再透过 SetLength 程序设定数组的长度即可：

```
var
    Array1: array of Integer;
begin
    // this would cause a runtime Range Check error
    // Array1 [1] := 100;
    SetLength (Array1, 10);
    Array1 [1] := 100; // this is OK
```

在还没有设定数组的长度之前，我们不能直接使用数组，透过 SetLength 程序，我们会在 heap 的内存里面配置数组需要的空间。如果我们没有这么做，要不会出现范围检查错误(如果对应的编译器设定没有设定为启动的话)，或者就会在 Windows 上面出现存取违规或者在其他作业平台上出现类似的内存存取错误。在上述源码里面，SetLength 函式会设定所有值先设为 0。初始源码会让这个数组立刻可以开始被读写，不用担心任何内存错误(当然，如果我们硬是存取数组范围之外的值，还是会错的)。

如果我们想要自己配置内存，我们也不用直接释放它，在上面的源码里面，当 Array1 变量的生命周期结束，而程序也结束时，编译器就会自己释放掉当中使用的内存了(以上面的源码为例，原本配置的 10 个整数的内存空间会被释放掉)。所以我们可以直接把 nil 指派给动态数组变量，或者呼叫 SetLength(0)，这通常都是不需要的(编译器会自己处理)。

请注意，SetLength 程序可以用来调整数组大小，如果我们把数组的长度加大，原有的元素并不会受到影响，而如果我们把数组的长度缩短，则被裁减到的元素也会被释放，但没有被裁减的元素则不会受影响。

在最初呼叫 SetLength 程序的时候，我们只指定了数组元素的数量，数组的起始索引仍旧是 0，最后一个索引则是指定的数量减一。换句话说，动态数组并不支持传统 Pascal 静态数组的两个功能：指定从非 0 的元素作为初始索引，以及以非整数作为索引值。同时，Object Pascal 动态数组的动作与行为也更像 C 语言或其他从 C 衍生而来的编程语言了。
要查询动态数组目前的长度，我们可以用跟静态数组相同的作法：透过 Length，也可以使用 High 跟 Low 函式。只是在动态数组使用 Low 函式，永远都会得到 0，使用 High 函式，则永远都会得到数组长度减一。也就是说，对一个长度为 0 的动态数组执行 High 函式，会得到-1(想一想，这是个奇怪的数值，对吧，因为 High 比 Low 传回的值还小！！)

所以，当成个例子吧，在 DynArray 范例项目里面，我透过可变循环来把信息放进，也从动态数组取出，这是型别跟变量定义：

```
type
    TIntegersArray = array of Integer;
var
    IntArray1: TIntegersArray;
```

数组被以索引值的数据进行配置与存放，使用以下的循环：

```
var
    I: Integer;
begin
       SetLength (IntArray1, 20);
       for I := Low (IntArray1) to High (IntArray1) do
           IntArray1 [I] := I;
end;
```

第二个按钮的程序则包办了显示每个数值与计算平均值的功能，类似前一个范例，不同的是只使用单一循环：

```
var
    I: Integer;
    total: Integer;
begin
    Total := 0;
    for I := Low(IntArray1) to High(IntArray1) do begin
       Inc (Total, IntArray1[I]);
       Show (I.ToString + ': ' + IntArray1[I].ToString);
    end;
```

```
    Show ('Average: ' + (Total / Length(IntArray1)).ToString);
end;
```

这段程序的输出则相当的浅显易懂(我省去了大部分):

```
0: 0
1: 1
2: 2
3: 3
...
17: 17
18: 18
19: 19
Average: 9.5
```

除了 Length, SetLength, Low, High 之外，我们还有一些常用的程序会跟数组搭配使用的，例如 Copy 函式，我们可以用它来复制数组的部份元素(也可以复制全部)。注意到，我们也可以把数组变量指派给另一个变量，但这么做并不是把所有元素都复制了一份过去喔，这么做只会把数组的指针复制过去，两个数组变量所使用的内存空间会是同一份，所以当我们修改其中一个数组的元素时，另一个也会一起变动。

唯一略为复杂的源码是 DynArray 程序的最后部分，用两种方法来把一个数组复制到另一个去：

- 使用 Copy 函式，可以把数组的数据复制到使用不同的内存空间的一个新的数组去。
- 使用指派运算符(:=)，这作法只会做出另一个变量的替身，储存的还是同一份内存空间。

在这时候，如果我们修改了其中一个数组的元素时，原来那数组的元素会不会跟着一起发生异动，就要看你用哪一种方法来做这个复制的动作了，以下是完整的源码：

```
var
    IntArray2: TIntegersArray;
    IntArray3: TIntegersArray;
  begin
    // Alias
    IntArray2 := IntArray1;
```

```
    // Separate copy
    IntArray3 := Copy (IntArray1, Low(IntArray1), Length(IntArray1));


  // Modify items
    IntArray2 [1] := 100;
    IntArray3 [2] := 100;
 // Check values for each array
    Show (Format ('[%d] %d -- %d -- %d',   [1, IntArray1 [1], IntArray2 [1], IntArray3 [1]]));
    Show (Format ('[%d] %d -- %d -- %d',   [2, IntArray1 [2], IntArray2 [2], IntArray3 [2]]));
```

执行结果，我们会看到如下的输出：

```
[1] 100 -- 100 -- 1
[2] 2 -- 2 -- 100
```

对 IntArray2 的变动也同步影响了 IntArray1，因为用 := 指派运算符，只是把 IntArray1 的地址存一份到 IntArray2 而已，内存空间是完全相同的一份，但在 IntArray3 则已经是独立的一份，所以对 IntArray3 的变动不会影响到 IntArray1。

对动态数组提供一些新的原生处理

动态数组已经支持把常数数组指派给动态数组，以及提供动态数组的连接。

笔记 这些对动态数组的功能延伸，是从 Delphi XE7 开始提供。

在实务上，我们可以写出像以下的源码，这比以往的源码都要来的更为精简：

```
var
    DI: array of Integer;
    I: Integer;
begin
    DI := [1, 2, 3]; // Initialization
    DI := DI + DI; // Concatenation
    DI := DI + [4, 5]; // Mixed concatenation
    for I in DI do
    begin
        Show (I.ToString);
    end;
```

请注意，在上面的源码里面的 for-in 循环会扫描整个数组里面的元素，这源码是范例项目 DynArrayConcat 的一部分。另外，这些数组的元素型别可以是任何一种，上例是 Integer，我们也可以改为 record 或类别。

还有第二个功能增强，就是可以在指派时直接进行连接，但这是 RTL 的部份，不是编程语言上有所改变。要把动态数组弄的像字符串那么容易操作是不太可能的，但我们可以用 Insert 跟 Delete 来处理元素的连接跟删除。

这表示我们可以写出以下这样的源码了(也是同一个项目的源码):

```
var
    DI: array of Integer;
    I: Integer;
begin
    di := [1, 2, 3, 4, 5, 6];
    Insert ([8, 9], di, 4);
    Delete (di, 2, 1); // remove the third (0-based)
```

开放数组参数

在数组的使用上有些情境非常特别，例如把一个不确定长度的清单作为参数传给函式。除了直接传一个数组作为参数，我们还会在这一节跟下一节介绍两个特殊的语法结构。具备这特性的函式之一，就是 Format 函式，我们在前面几个范例源码里面有使用到，它的第二个参数，就是以方括号建立起来的一个数组。

跟 C 语言不同(当然跟其他从 C 衍生的语言也不同)，在传统的 Pascal 语言里，函式跟程序的参数永远都是固定个数的。然而在 Object Pascal 里面，就有方法把不特定个数的参数透过数组参数来传递，从技术上来说，这就称为*开放数组参数*。

笔记 从历史纵轴来看，开放数组参数的出现早于动态数组，但今日看来，这两个技术从实际作用的方法来看非常相似，几乎越来越无法区别了。这也是为什么我把开放数组参数放在动态数组之后进行讨论的原因。

开放数组参数的基本定义跟已定义的动态数组型别是一样的，前面加上 const 叙述字。这表示我们可以定义参数的型别，但我们不用定义这个数组中一定要放几个元素不可。以下是这种定义的范例之一，也是从 OpenArray 范例项目中撷取而来：

```
function Sum (const A: array of Integer): Integer;
var
    I: Integer;
begin
     Result := 0;
     for I := Low(A) to High(A) do
         Result := Result + A[I];
end;
```

我们可以呼叫这个函式，传给它一个整数数组作为参数(当中当然可以参杂变量与常数，这个函式就会把其中每个数值拿来使用):

```
X := Sum ([10, Y, 27*I]);
```

假设有个整数型别的动态数组，我们可以把它直接传给一个要求开放数组参数的函式(假设这里就是要求整数数组)，以下就是这样一个范例，整个数组会被传进去作为参数：

```
var
    List: array of Integer;
    X, I: Integer;
begin
    // Initialize the array
    SetLength (List, 10);
    for I := Low (List) to High (List) do
        List [I] := I * 2;
    // obtain sum of list elements using our function
    X := Sum (List);
```

这是如果我们是用动态数组，假如我们是用相同元素型别的静态数组，就可以也把它传给要求开放数组参数的函式，或者我们也可以呼叫 Slice 函式，把数组的一部分传去当参数(当成其中的第二个参数)。以下的程序片段(也是 OpenArray 范例的一部分)示范了如何把一个静态数组的一部分传给 Sum 函式作为参数:

```
var
    List: array [1..10] of Integer;
    X, I: Integer;
begin
    // Initialize the array
    for I := Low (List) to High (List) do
```

```
    List [I] := I * 2;


// Obtain sum of list elements using our function
X := Sum (List);
Show (X.ToString);


// Sum portion of the array
X := Sum (Slice (List, 5));
Show (X.ToString);
```

变异型别的开放数组参数

除了这些指定型别的开放数组参数，Object Pascal 也允许我们定义变异型别(type-variant)或者不指定型别(untyped)的开放数组作为参数。这个特殊的数组会拥有不定数量的元素，而这些元素的数据型别也可能各自不同。这也正是 Object Pascal 的限制进入区域之一，因为这里已经不是完全型别安全的区域了。

技术上来说，我们可以定义一个参数，它的型别是 array of const，用以把一个不确定数量与元素型别的数组作为参数来传给函式。举例来说，以下是 Format 函式的定义(我们在第六章里面会讨论这个函式，在讨论字符串的时候，不过我已经在几个范例里面提过它了):

```
function Format (const Format: string;
    const Args: array of const): string;
```

第二个参数是一个开放数组，他会接受不确定数量的值，事实上，我们可以用以下方式呼叫这个函式：

```
N := 20;
S := 'Total:';
Show (Format ('Total: %d', [N]));
Show (Format ('Int: %d, Float: %f', [N, 12.4]));
Show (Format ('%s %d', [S, N * 2]));
```

请注意，我们可以把一个常数、变量内容，甚至是表达式当做参数传递。宣告这样的函式很容易，但我们要怎么实作它？我们怎么知道这些参数的型别？变异型别的开放数组参数是和 TVarRec 这个型别的元素完全兼容的。

笔记

别把 TVarRec 记录和 Variant(变异型别)使用的 TvarData 记录搞混了。这两个结构的作用不同，也完全不兼容。即使连列出的可能型别也不一样，因为 TVarRec 所处理的型别都是 Object Pascal 的数据型别，而 TVarData 处理的型别都是 Windows 的 OLE 数据型别。关于 Variant(变异型别)，我们会在本章后面的篇幅来加以介绍。

以下我们列出在变异型别开放数组当中可以支持的资料型别，当然 TVarRec 记录也就能支持：

```
vtInteger       vtExtended      vtPChar
vtWideChar      vtCurrency      vtWideString
vtBoolean       vtString        vtObject
vtPWideChar     vtVariant       vtInt64
vtChar          vtPointer       vtClass
vtAnsiString    vtInterface     vtUnicodeString
```

这个记录的结构包含型别(VType)，以及一个变异型别字段，我们可以透过这个字段来存取实际的资料(比一些页面的纪录更多，即使这是这些结构的进阶功能)

一般的作法是当我们的函式接收到这种型别的参数时，就透过 case 叙述句来对不同型别的数据进行处理。在范例源码函式 SumAll 里面，我想把不同型别的数据做加总，把字符串转换成整数，字符转成对应的有序数值，把布尔值的 True 转成 1。这个源码可以说相当的高端(里面还用到了指标引用：pointers dereference)，所以如果现在看不懂也不用担心：

```
function SumAll (const Args: array of const): Extended;
var
    I: Integer;
begin
    Result := 0;
    for I := Low(Args) to High (Args) do
      case Args [I].VType of
        vtInteger:
          Result := Result + Args [I].VInteger;
        vtBoolean:
          if Args [I].VBoolean then
            Result := Result + 1;
        vtExtended:
          Result := Result + Args [I].VExtended^;
```

```
        vtWideChar:
          Result := Result + Ord (Args [I].VWideChar);
        vtCurrency:
          Result := Result + Args [I].VCurrency^;
      end; // case
 end;
```

我已经把上面这个函式加入 OpenArray 范例程序里，可以用以下的源码来呼叫它：

```
var
   X: Extended;
   Y: Integer;
begin
            Y := 10;
            X := SumAll ([Y * Y, 'k', True, 10.34]);
            Show ('SumAll: ' + X.ToString);
end;
```

这个函式的输出加入了 Y 平方值，以及 K 的有序值(107)，以 1 表示布尔值的 True，以及浮点数，结果如下：

```
SumAll: 218.34
```

记录数据型别

数组以数字索引定义了一连串的元素，而记录则以名称定义对应的成群元素。换句话说，记录是一连串被赋予名称与对应字段的元素，每一组都元素都有其特定的资料型别。记录数据型别的定义会把这些字段全部列出，包含每个字段的名称，以及对应数据应有的型别。早期的 Pascal 语法中，记录只能储存数据，改进到现在，记录也可以包含方法跟运算方法了，我们在本章稍后会再介绍。

笔记　在大多数的编程语言里面都有记录这个型别，在 C 语言里面，被以 struct 这个关键词建构，在 C++里面则是用方法的延伸定义来处理，跟 Object Pascal 的作法很像。有些『纯』面向对象式的编程语言则只能在类别里面进行加注，就没有提供 record 或 struct 这样的功能了。

以下是段简单的源码(这段源码是从 RecordsDemo 范例项目节录的)，里面定义了一个记录型别，并用这个型别定义了一个变量，以及一些使用这个变量的源码：

```
type
      TMyDate = record
                Year: Integer;
                Month: Byte;
                Day: Byte;
      end;
var
   BirthDay: TMyDate;
begin
     BirthDay.Year := 1997;
     BirthDay.Month := 2;
     BirthDay.Day := 14;
     Show ('Born in year ' + BirthDay.Year.ToString);
```

笔记

『记录』这个词在 Object Pascal 里面可能代表两种意思，一个是『记录』型别的定义，另一个则是使用记录型别的变量(或者也可以叫『记录』变量)。所以『记录』这个词有两种意思，一定要认清我们用这个名词的时候所指的是什么。

在 Object Pascal 里面，记录用来代表数据结构的时候，多于代表简单的字段列表，在本章的剩余部分会试着说明，但我们先从记录的传统用法开始。记录所使用的内存通常会被配置在局部变量的堆栈，或者全局变量所使用的全局内存空间。我们用 SizeOf 这个函式来强调这个现象，SizeOf 会回传一个变量所使用的内存空间大小(单位为 byte)，可以写成以下指令这样:

```
Show ('Record size is ' + SizeOf (BirthDay).ToString);
```

上面这个指令在 Win32 平台上，使用预设的编译设定时会回传 8 (它回传 8, 而不是 6 或 4 bytes，是因为使用了一个整数，以及另外两个 byte 字段-我们等下会在”字段对齐”的小节加以说明)。

换句话说：记录是值类型，这表示如果我们把记录指派给另一个记录变量，我们就会做一份完整的数据复制。如果我们对新的变量内容做修改，原来的记录变量内容是不会变动的。以下的程序片段用源码来说明这个概念：

```
var
   BirthDay: TMyDate;
```

```
    ADay: TMyDate;
 begin
    BirthDay.Year := 1997;
    BirthDay.Month := 2;
    BirthDay.Day := 14;
    ADay := Birthday;
    ADay.Year := 2008;
    Show (MyDateToString (BirthDay));
    Show (MyDateToString (ADay));
```

输出结果(用国际日期格式):

```
 1997.2.14
 2008.2.14
```

对同样一份数据做异动的情况，会发生在当我们把记录当成函式的参数传入时，就像上面的源码里面我们呼叫的 MyDateToString 函式：

```
function MyDateToString (MyDate: TMyDate): string;
begin
    Result := MyDate.Year.ToString + '.' +
          MyDate.Month.ToString + '.' +
          MyDate.Day.ToString;
end;
```

呼叫这个函式的时候，都会完整复制一整份记录型别的数据。为了避免这个复制的情形发生，我们可以透过宣告该函式的参数是引用参数的作法，来达成对原始记录数据的变更。在下面这个范例源码里面，就特别以引用参数的作法来实作：

```
procedure IncreaseYear (var MyDate: TMyDate); begin
    Inc (MyDate.Year);
end;


var
    ADay: TMyDate;
begin
    ADay.Year := 2016;
    ADay.Month := 3;
    ADay.Day := 18;
    Increaseyear (ADay);
```

```
Show (MyDateToString (ADay));
```

假使原始记录变量的 Year 字段会因为呼叫这个程序而被增加，则呼叫完函式之后，原始变量的 Year 字段就会比呼叫前多一年：

```
2021.3.18
```

使用记录数组

就像之前提到过的，数组可以把同一个型别的数据重复建置出来，而记录是在单一结构里面包含不同的元素。假使这两个型别建构子的用途是不同的，那么把它们两个混着用就是很正常的想法，也就是定义一个数组，让该数组的元素型别是记录型别(另一种混用法就很少见: 在记录中使用数组)。

这个数组的源码还是跟其他任何的数组一样，每个数组的元素使用一个记录型别的空间。我稍晚会介绍如何使用更复杂的结构，像是 Collection 或者 Container 类别(用来储存多个元素)，在记录数组的使用上，我们可以学到更多数据管理的名词。

在 RecordTest 项目里面，我加进了一个 TMyDate 型别的数组，我们可以用以下的源码对这个数组进行配置、初始化并使用它：

```
var  DatesList: array of TMyDate; I: Integer;
begin
    // Allocate array elements
    SetLength (DatesList, 5);
    // Assign random array element values
    for I := Low(DatesList) to High(DatesList) do begin
        DatesList[I].Year := 2000 + Random (50);
        DatesList[I].Month := 1 + Random (12);
        DatesList[I].Day := 1 + Random (27);
    end;
    // Display the values
    for I := Low(DatesList) to High(DatesList) do
        Show (I.ToString + ': ' +
            MyDateToString (DatesList[I]));
```

假设 app 使用随机数据，则输出值就会每次都不一样，就像我以下撷取的结果:

```
0: 2014.11.8
```

```
1: 2005.9.14
2: 2037.9.21
3: 2029.3.12
4: 2012.7.2
```

笔记 在数组中的记录，会自动以受管理的记录被自动初始化，这是 Delphi 10.4 Sydney 的新功能，我们在本章后段会再介绍。

变异记录 (Variant Records)

在 Object Pascal 早期的版本中，记录型别也可以有变动的部份，也就是说，多个字段可能对应到相同的一个内存空间，即使这些字段的型别不同。(这个概念可以对应到 C 语言里面的 union)。另一个作法，我们可以使用这些变异字段或者群组来存取记录里面的同一个内存空间，但这些数据还必须考虑到不同的型别的观点。这种型别的主要用途，使用来储存大同小异的数据，或者用型别转换的方式来对同一块内存数据进行不同解译(透过强制转型，这在 Object Pascal 早期版本是可以的，今日的版本已经不允许直接做强制转型了)。变异记录的应用已经大幅被面向对象或其他更新的技术所取代了，目前只剩很少的系统函式库在很特别的情境下会内部使用到这个技术了。

变异记录型别的使用并不保证型别使用上的安全，在程序实务上也不建议使用。除非您已经是 Object Pascal 的专家，不然您不用特别去抓出这些情况。总之，这也是我决定不以实例说明这个观念的原因，当然也就不会特别介绍其用法了。如果您真的需要这方面的相关概念，请看一下我在 121 页中『变异型别的开放数组参数』那一节的范例程序中，对 TVarRec 的使用吧。

字段对齐 (Fields Alignments)

另一个跟记录相关的进阶议题，是这些记录用来进行对齐的方法，这个方法也让我们对于了解记录的实际大小更有帮助。如果我们深入观察函式库，我们会常常看到 packed 这个关键词在记录型别上出现：这表示记录必须尽可能使用最小的空间，即使这样的作法可能导致数据存取的动作变慢。

这个差异是从变异字段的对齐有 16bit 或 32bit 两种选项的时候开始出现的，所以在变异字段的内容储存的是整数时，即使这个整数只使用了 8bit，这字

段的下一个内存开始的地址，也会是在该整数起始位置后的 32bit 处。这样的内存对齐机制，会使得源码的执行更为快速。

笔记

字段的大小跟对齐作业会取决于型别的大小。所有大小不是 2 的 N 次方的型别，内存大小会自动配置为大于该大小的下一个 2 的 N 次方。举例来说，Extended 型别大小是 10 bytes，在记录中就会被配置 16 bytes(除非该记录在宣告时使用了 packed 关键词)

一般的字段对齐技术，都是被用在类似记录型别的数据结构，部分 CPU 架构在使用了这样的技术后，就能加速对独立字段的数据存取。我们可以透过 $ ALIGN 这个编译器开关来改变字段对齐的设定。

使用{$ALIGN 1}这个设定，编译器会尽量节省内存的使用，就像我们在记录型别上使用 packed 这个描述字的效果，另一种极端，则是使用{$ALIGN 16}，这个设定会使用最大量的内存进行对齐机制，此外还有 4 跟 8 两种对齐的参数可以设定。

如果我们回头拿 RecordsTest 项目当成个例子，在里面的记录定义加上 packed 关键词：

```
type
    TMyDate = packed record
        Year: Integer;
        Month: Byte;
        Day: Byte;
    end;
```

SizeOf 这个函式的回传值现在就会回传 6, 而不是回传 8 了.

用以下的例子作为进阶的范例，如果您对 Object Pascal 还不是很熟悉，可以先跳过这个例子。我们先看一下以下的结构(可以在 AlignTest 这个范例项目里面看到):

```
type
    TMyRecord = record
      C: Byte;
      W: Word;
      B: Boolean;
      I: Integer;
      D: Double;
```

```
end;
```

当设定了{$ALIGN 1}，上面的结构会使用 16 bytes(这是 SizeOf 的回传值)，而在相对的内存地址里面，内容会像以下这样：

```
C: 0 W: 1 B: 3 I: 4 D: 8
```

笔记 相对地址是以记录布局和以数值、字段配置之间的差异计算出来的。例如这样的算式：UIntPtr(@MyRec.w) – UIntPtr(@MyRec1)。指针跟地址符号(@)我们会在本章后面的部分介绍。

比较一下，如果我们把对齐的尺寸改为 4 (这会使数据存取的速度变快)，这个 size 就变成了 20 bytes，相对地址就变成了:

```
C: 0 W: 2 B: 4 I: 8 D: 12
```

如果我们用了很极端的选项{$ALIGN 16}，这个结构就需要 24 byte，对应的字段如下:

```
C: 0 W: 2 B: 4 I: 8 D: 16
```

With 叙述句是什么?

另一个旧版 Pascal 的语言叙述句，会被用来跟记录或类别一起使用，就是 With。这个关键词是比较奇特的 Pascal 叙述句，但它后来也在 JavaScript 跟 Visual Basic 里面出现了。这个关键词的使用，能让程序人员少打一些字，但也很危险，因为许多原本该写清楚的类别、记录的名称被省却了之后，整段源码就变得不容易读了。我们可以发现，一直以来关于 With 这个叙述句的争论不断，因此我也建议我们使用这个叙述句的时候，要持保守的态度。不管怎么说，我觉得一定要把这个叙述句在本书当中加以介绍（这跟稍早提到的 goto 叙述句是不同的）

笔记 对于是否要把 goto 叙述句从 Object Pascal 移除也一直有许多争论，对 with 叙述句是否要从行动版的编译器中移除也一样。虽然有些使用情境是合法的，但有些范围的问题会随着 with 叙述句的使用而发生，这就是不再继续支持这个功能的好理由了（或者像 C#一样用一个别名来提供该功能）

---

with 叙述句的功能就是让源码能够简写。当我们需要使用一个记录型别变量(或者需要使用对象)时，为了不用每次要用到该变量时就输入该变量的名字，我们就可以用 with 叙述句来省却输入这个变量的名字，举例来说，当需要使用到记录型别时，我把以下这段源码：

```
var
    ABirthDay: TMyDate;
begin
            ABirthDay.Year := 2008;
            ABirthDay.Month := 2;
            ABirthDay.Day := 14;
```

加上了 with 叙述句之后，可以把这段程序改写成：

```
with ABirthDay do
begin
       Year := 2008;
       Month := 2;
       Day := 14;
end;
```

这个作法可以在 Object Pascal 的源码当中套用在使用组件或其他类别的使用上。当我们的源码需要使用到对象或类别的时候，with 叙述句就允许我们可以简化源码，尤其是在巢状数据结构的使用上。

所以，我之所以不鼓励使用 with 叙述句的原因，就是它可能让微小的错误难以被找出来。这些难以发现的错误在本书的这个点当中很难说明。我们先来思考一个比较没有伤害性的情境，这已经会让我们白头搔更短，浑欲不胜簪。以下是一个记录以及使用这个记录的一些源码：

```
type
    TMyRecord = record
       MyName: string;
       MyValue: Integer;
    end;
procedure TForm1.Button2Click(Sender: TObject);
var
    Record1: TMyRecord;
begin
    with Record1 do
    begin
       MyName := 'Joe';
       MyValue := 22;
    end;
    with Record1 do
       Show (Name + ': ' + MyValue.ToString);
```

是吧?这个应用程序可以编译、执行，但它的执行结果却跟我们第一眼看完以后期盼的结果不一样：

```
Form1: 22
```

这个输出结果的字符串部分并不是前面的程序设定的记录值，原因是第二个 with 叙述句误用了 Name 字段，这不是记录里面定义的字段，而是另一个不在此程序范围中的变量(特别是使用到了 Button2Click 这个方法所在的 form 组件的 name 字段了)

如果我们这么写:

```
Show (Record1.Name + ': ' + Record1.MyValue.ToString);
```

编译器就会显示错误讯息，表示这个记录里面并没有定义一个名为 Name 的字段。

通常我们可以说从 with 叙述句出现的时候，就在当时的范围内定义了一个新的识别符号，我们可以隐藏既有的识别符号，在同一段源码的范围内使用到其他的识别符号。这也是一个用来警惕大家不要使用 with 叙述句的好理由，尤其我们更该避免使用多个 with 叙述句，例如:

```
with MyRecord1, MyDate1 do...
```

在上面这个 with 叙述句里面的源码会变得很难懂，因为每个在该程序片段当中的字段，我们都得想想到底是从哪个记录参考来的。

带有方法的纪录

在 Object Pascal 里面的纪录，比以前的 Pascal 语言的纪录或者 C 语言里面的 struct 都要更为强大。事实上，记录跟类别一样，可以包含有与之相关的方法(我们稍后会介绍)，例如程序或函式。这些方法甚至可以重新定义语言本身的运算方法(这个功能称之为运算符多载: operator overloading)，我们会在下个章节介绍。

一个带有方法的纪录，基本上已经跟类别非常接近了，我们稍后就会发现这一点。而这两者之间最大的不同，只在他们管理内存的方式不同而已。Object Pascal 的记录具备现代编程语言的两种基本功能：

- **方法**：也就是和记录数据结构直接链接的程序或函式，这些方法可以直接使用记录的数据字段。换句话说，方法就是在宣告记录型别的时候同时宣告的程序或函式(也可以只是预先宣告)。

- **封装**：透过封装，我们可以限制某些数据字段或方法不被其他源码直接使用。我们可以透过 private 这个存取描述字来提供封装的功能，好让其他源码无法看见位于 private 区段的数据或方法。而 public 区段的字段跟方法则可以被所有源码所使用。预设的存取描述字是 public。

现在我们对记录的延伸功能有了核心观念的认识，我们来看一个简单的记录定义，这是从 RecordMethods 范例项目所节录的：

```
type
    TMyRecord = record
    private
        FName: string;
        FValue: Integer;
        FSomeChar: Char;
    public
        procedure Print;
        procedure SetValue (NewString: string);
        procedure Init (NewValue: Integer);
end;
```

我们可以看到这个记录结构被分成两个部分:private 跟 public。我们可以宣告多个部分，public 跟 private 这些关键词在同一个记录宣告可以出现不只一次。但如果使用多次，就要留意到在不同区段的变量跟方法究竟是属于 private 还是属于 public，不然我们的源码以后会很难懂。其次，上述的程序片段中，里面的方法都只有宣告名称，并没有实作方法的源码在里面，所以这些是属于预先宣告。

我们要怎么撰写这些方法的源码，好让整个定义完整呢?方法几乎完全一样，我们得写个全局的函式或程序。唯一不同的是，我们得用记录的名字来帮这些方法冠名，这些方法是以记录型别的名称跟实际上的方法名称组合而成的。实际写作的时候，我们可以直接指名要存取的域名，甚至是直接使用该记录型别的其他方法，而不用再写一次该记录的名字：

```
procedure TMyRecord.SetValue (NewString: string);
begin
    FName := NewString;
end;
```

在这段源码里面，NewString 只是函式的参数，而 FName 则是记录的字段。

提示

先写好方法的定义，然后再一一写好完整的定义，这方法挺无聊的。我们可以直接在 IDE 里面按 Ctrl+Shift+C 这组快捷键，就能自动依照宣告的内容把实作所需要的程序样板全部自动产生出来了。我们也可以按 Ctrl+Shift+箭头键(上/下)在方法宣告与完整源码之间快速切换。

以下是这个记录型别的其他方法实作源码：

```
procedure TMyRecord.Init(NewValue: Integer);
begin
    FValue := NewValue;
    FSomeChar := 'A';
end;
function TMyRecord.ToString: string;
begin
    Result := FName + ' [' + FSomeChar + ']: ' + FValue.ToString;
end;
```

以下是介绍如何使用这个记录型别的简单程序范例：

```
var
  MyRec: TMyRecord;
begin
    MyRec.Init(10);
    MyRec.SetValue ('Hello');
    Show (MyRec.ToString);
```

您可能已经猜到，输出的结果是这样子的:

```
Hello [A]: 10
```

而如果我们想要直接使用这个记录里的字段，会发生什么事呢?

```
MyRec.FValue := 20;
```

这段程序不仅可以编译，也可以执行。这就跟我们在 private 区段里面需告了字段一样令人惊讶。因为一般应该只有记录方法可以存取这些字段。原因是因为 Object Pascal 的 private 存取描述字，只对不同 unit 有作用。

所以如果这段程序是放在不同 unit 里面，就会变成不合法的指令了。但在宣告该记录型别的同一 unit 里面，这是完全合法的。同样的规则，适用在记录型别以及类别上面。

Self: 记录神奇的地方

假设我们有两个记录，例如叫做 MyRec1 跟 MyRec2，这两个记录是相同的记录型别，当我们呼叫一个方法，并执行它的源码。我们怎么知道这两个记录的源码到底是哪一个会被执行？在幕后，当我们定义了一个方法，编译器会自动加一个隐藏参数进去，我们呼叫该方法时，就有一个记录的参考存在了。

换句话说，上述的源码里面会被编译器转换成类似以下这样：

```
// 您写的：  MyRec.SetValue ('Hello');
// 编译器产生：  SetValue (@MyRec, 'Hello');
```

在上述的伪码里面，@(*address of*)这个符号是用来表示内存所在的地址，通常用来取得一个记录型别的变量所配置的内存。

笔记 重申一次，@(*address of*)这个符号我们会在本章最后再介绍一次，章节标题会是”那关于指标呢?”

这是呼叫方法的源码被转译的方法，但实际上的源码呼叫是怎么参考到这个隐藏参数的？实际上是使用了一个特别的关键词”Self”.所以这个方法应该要写成：

```
procedure TMyRecord.SetValue (NewString: string);
begin
    Self.FName := NewString;
end;
```

在这段程序编译的过程中，使用 Self 这个关键词至为重要，除非我们需要把记录整个进行参考。例如，把记录当做参数传给另一个函式。这个情境在类别的使用上就蛮常见的，在同样的情形下，类别也会使用完全相同的关键词”Self”来处理。

使用了”Self”这个关键词作为参数，会让源码更容易懂(虽然多写 Self 并不是必要的)，其中一个情境，是我们需要用同一个记录型别复制完全相同的数据时。在这个情境下，我们如果要对另一个实体进行数值测试：

```
function TMyRecord.IsSameName (ARecord: TMyRecord): Boolean;
begin
    Result := (Self.FName = ARecord.FName);
end;
```

笔记 Object Pascal 隐藏的参数 Self, 在 C++跟 Java 里面叫做 this，在 objective-C 里面也一样用 self 喔。

为记录初始化

当我们定义了一个记录型别跟变量(或者称为记录的实体)，并把它宣告为全局变量时，记录里面的字段就已经被初始化了，但当我们在堆栈当中宣告的时候(可以想象这是个程序或函式内部的局部变量)，就不会自动初始化。所以当我们写了以下的源码(也是 RecordMethods 范例项目的一部分)：

```
var
  MyRec: TMyRecord;
begin
  Show (MyRec.ToString);
```

这段程序的输出值则会多少有些随机变化。当字符串初始值是个空字符串时，字符字段跟整数字段会直接从被配置的内存空间中取值来用(就跟一般局部变量初始化的动作一样)。通常这个值会随着每次执行的时候，从内存取得的随机地址有所不同，像是：

```
[□]: 1637580
```

这就是为什么在使用记录型别变量前，为记录执行初始化程序是很重要的(跟其他变量进行初始化一样重要)，避免读取非法的数据，因为那很容易导致整个应用程序挂点。

要处理这个情况，有两种不同的方式，第一种作法是为记录制作建构者，我们下一节就会谈到。第二种作法则是使用受管理的记录(managed record)，这是 Delphi 10.4 里面的新功能，本章后段我会再介绍。

记录与建构者(Constructors)

我们先从一般的建构者开始。记录也支持一种特别的方法，称为建构者，我们可以透过它来对记录进行初始化。跟其他方法不同的是，建构者可以建立一个新的变量实体(但仍旧可以对已存在的实体进行初始化)。以下示范怎么帮记录新增一个建构者：

```
type
  TMyNewRecord = record
  public
```

```
    constructor Create (NewString: string);
    function ToString: string;  ...
```

建构者是包含源码的方法:

```
constructor TMyNewRecord.Create (NewString: string);
begin
   Name := NewString;
   Init (0);
end;
```

这么一来，我们就可以用以下两种写法之一来对记录做初始化了：

```
var
  MyRec, MyRec2: TMyNewRecord;
begin
   MyRec := TMyNewRecord.Create ('Myself'); // class-like
   MyRec2.Create ('Myself'); // direct call
```

请注意，记录的建构者必须有参数：如果我们试着呼叫 Create，就会得到错误讯息：” Parameterless constructors not allowed on record types” (记录型别的建构者不能没有参数)。

笔记 我们也可以为建构者制作多载或者用不同的名称制作多个建构者。我们在后面的章节中讨论到类别的建构者时会进行介绍。我们接下来会快速介绍一下受管理的记录(managed record)，使用不同的语法，也没有无参数的建构者，跟类别方法 Initialize 不一样。

运算符号的新纪元 (Operators Gain New Ground)

Object Pascal 语言另一个跟记录相关的功能是对运算符号进行多载，这个功能可以让我们对我们定义的数据型别所需的系统标准的运算符号进行多载(例如加、减、乘等等运算)。这个意思就是说，我们可以自己定义一个加法(一个特别的 Add 方法)，然后透过+符号来呼叫它。要定义这样的运算符号，我们可以使用 class operator 这个关键词来实作。

笔记 透过重复使用已经存在的保留字，编程语言的设计者把这个功能对于旧有的源码冲击降到了 0。这个作法就是把常用的关键词做组合，例如 strict private, class operator, class var 等等。

class 这个名词，在这里是和类别方法(class method)做连结，我们之后会在第 12 章介绍这个概念。在这个关键词之后，我们可以宣告运算方法的名字，例如 Add:

```
type
  TPointRecord = record
  public
    class operator Add (a, b: TPointRecord): TPointRecord;
```

这里的 Add 方法，在我们使用 TPointRecord 这个记录型别的变量时，就可以用+符号来呼叫了，例如：

```
var
  A, B, C: TPointRecord;
begin ...
  C := A + B;
```

变量 C 的结果就会是 A 跟 B 这两个记录型别变量相加的结果。有哪些运算符号是可以透过这个方式来重新定义的呢?基本上，整个 Object Pascal 编程语言里面的运算符号都可以，但我们不能定义 Object Pascal 没有定义过的运算符号喔，以下的运算符号都可以重新定义：

- 型别转换运算符号(Cast Operators): Implicit, Explicit
- 一致性运算符号(Unary Operators): Positive, Negative, Inc, Dec, LogicalNot, BitwiseNot, Trunc, 以及 Round
- 比较运算符号(Comparison Operators): Equal, NotEqual, GreaterThan, GraterThanOrEqual, LessThan, 以及 LessThenOrEqual
- 二进制运算符号(Binary Operators): Add, Subtract, Multiply, Divide, IntDivide, Modulus, ShiftLeft, ShiftRight, LogicalAnd, LogicalOr, LogicalXor, BitwiseAnd, BitwiseOr, 以及 BitwiseXor.
- 受管理的纪录运算符号: Initialize, Finalize, Assign(请参阅下一节”运算符号与自定受管理的记录”，该节当中会介绍这三个在 Delphi 10.4 新增的运算符号)

在我们实际撰写的源码里面，不用特别呼叫这些方法，我们只需要在源码中写入对应的运算符号即可，只有在定义的时候才需要写出这些特殊方法的名字，记得要在这些特殊方法的宣告前面加上 class operator，以免跟其他方法的名称重复了。例如我们可以在一个记录型别里面重新制作 Add 运算符号，而这个记录型别里面也可以有个方法叫做 Add.

当我们在定义这些运算符号的时候，我们得把所需的参数完整列出，运算符号只会在参数完全符合的时候才被套用。例如要提供两个不同型别的值相加的功能，我们就得在这个新的运算符号宣告里面写清楚，第一个参数是哪一种型别的值，第二个参数是哪种型别的值。实务上，运算符号的定义并不会提供自动切换的功能喔。而且我们必须很精准的定义型别，因为型别自动转换的功能在这里不会启动。很多时候这表示运算符号以多载功能，透过不同型别的参数定义了多种不同版本。

另一个我们需要留意的重要元素，是在用来定义数据转换的两个特殊运算符号：Implicit 跟 Explicit。第一个是用来定义一个隐含的型别转换(或者称为宁静转换)透过这种转换，数据不会有任何遗失(因为只是把内存数据用另一种型别的格式加以解释)。第二个符号(Explicit)则会真的把一个变数转换成另一种型别的变数。这两种运算符号都可以用来处理『转换前』，以及『转换后』的数据型别。

我们要留意到，Implict 跟 Explict 都可以基于函式回传的型别进行多载，这通常不可能发生在多载的方法上。实务上，在型别转换发生时，编译器会知道要回传的型别是什么，因此就会自动先把该做的型别转换给做了。例如，在 OperatorsOver 这个范例项目中，我定义了一个记录跟一些运算符号：

```
type
    TPointRecord = record
    private
         X, Y: Integer;
    public
        procedure SetValue (X1, Y1: Integer);
        class operator Add (A, B: TPointRecord): TPointRecord;
        class operator Explicit (A: TPointRecord): string;
        class operator Implicit (X1: Integer): TPointRecord;
end;
```

以下则是这些运算符号方法的实作源码:

```
class operator TPointRecord.Add(A, B: TPointRecord): TPointRecord;
begin
     Result.x := A.X + B.X;
     Result.y := A.Y + B.Y;
end;


class operator TPointRecord.Explicit(A: TPointRecord): string;
```

```
begin
    Result := Format('(%d:%d)', [A.X, A.Y]);
end;
class operator TPointRecord.Implicit(X1: Integer): TPointRecord;
begin
    Result.X := X1;
    Result.Y := 10;
end;
```

这样的纪录在使用上就很直觉了，我们可以写出这样的源码：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  A, B, C: TPointRecord;
begin
  A.SetValue(10, 10);
  B := 30;
  C := A + B;
  Show (string(C));
end;
```

第二行的资料指派(B := 30)就是透过 implict 运算符号来达成的，由于在呼叫 Show 这个方法的时候，少了一个型别转换，所以我们自己在源码上面写清楚，用 explicit 型别转换处理它。另外也要思考一下，Add 运算符号不会更动传进来的参数，它只会回传一个新的数值。

笔记 为了配合运算符号只会回传新数值的这个规则，要思考覆写一个新的运算符号，其实难度就更高了。如果一个运算符号建立了一个新的随机对象，那谁要来释放这个对象?

在背景多载定义的运算符号

这一小节比较进阶，您可能在第一次阅读本书的时候会想先跳过。

透过鲜为人知的方法，其实是有方法可以透过撰写完整的内部方法名称来呼叫运算方法的(例如 &op_Addition)，在方法前面加上&符号，不要直接写出该运算方法对应的符号。举例来说，我们可以把这个记录的加总方法改写如下(请参照范例项目源码，就可以看到完整的列表):

```
C := TPointRecord.&&op_Addition(A, B);
```

不过我很少看到有什么情形需要我们这样写的就是了。(定义运算符号的用意，是让我们能够简化运算方法的名称，直接写算式当然比写成一般的函数调用更为直觉，对吧?)

实作出可交换性(Implementing Commutativity)

假如我们希望值做一个能把整数数值加到记录当中的功能。我们可以定义以下的运算方法(这些源码也在 OperatorsOver 范例项目里面找到，提供给两种显然不同的纪录型别使用):

```
class operator TPointRecord2.Add(A: TPointRecord2; B: Integer): TPointRecord2;
begin
    Result.x := A.X + B;
    Result.y := A.Y + B;
end;
```

笔记

之所以要把这个运算符号定义在一个新的型别，而没有在已经定义好的纪录里面加上它，是因为同样的结构已经定义了 Implicit 转换，把一个整数换乘该记录型别了。所以我就不用另外定义一个新的运算符号，就能直接把整数加到记录里面了。这个问题我们会在下一节里面好好说明。

现在，我们就可以把一个浮点数合法的加入到记录里面了：

```
var
   A: TPointRecord2;
begin
    A.SetValue(10, 20);
    A := A + 10;
```

然而如果我们把表达式子反过来写：

```
A := 30 + A;
```

编译器就会回报错误:

```
[dcc32 Error] E2015 Operator not applicable to this operand type
```

就像我们刚刚介绍过的，在实务上，运算符号定义后，是不会自动具备可交换性这个特质的，我们得自己重复制作，或者制作这个运算符号的另一种多载版本：

```
class operator TPointRecord2.Add(B: Integer; A: TPointRecord2): TPointRecord2;
```

```
begin
    Result := A + B; // 提供可交换性
end;
```

隐含转换与型别介绍(Implicit Cast and Type Promotions)

注意！这个重要：在呼叫运算符号时的解析，跟以往呼叫方法时的规则是完全不同的。当型别自动介绍发生时，是有可能发生单一一个表达式结束时，因为存在不同参数的多载，而导致函数调用产生混淆的情形的。这也是为何我们要更费心在 Implicit 运算符号的使用上。

以下面的表达式来思考前一个范例：

```
A := 50;
C := A + 30;
C := 50 + 30;
C := 50 + TPointRecord(30);
```

这些表达式都没有问题。在第一个式子中，转型会在指派动作之后发生。第二个式子，编译器会把 30 转换成适当的记录型别。第三个式子，explicit 型别转换会强迫把第一个数值用 Implicit 转型。然后加法才能在这些记录中执行。换句话说，第二个参数的结果跟另外两个都不一样，我们来看看这个延伸版本的的源码执行结果中的 X 跟 Y 值吧：

```
// Output
  (80:20)
  (80:10)
  (80:20)

// Expanded statements
C := A + TPointRecord(30);
// that is: (50:10) + (30:10)

C := TPointRecord (50 + 30);
// that is 80 converted into (80:10)

C := TPointRecord(50) + TpointRecord(30);
```

```
// that is: (50:10) + (30:10)
```

运算符号与自定受管理的记录

在 Delphi 当中，提供了一组特别的运算符号，让我们可以用来定义受管理的纪录(Managed record)。开始介绍之前，我们先回顾一下记录在内存初始化的规则，以及一般的记录与受管理记录之间的差别。

在 Delphi 的纪录里面，可以包含任何型别的数据字段。记录里面如果只包含了一般型别(并非受管理的)型别字段，像是数值或者其他列举型态的数据，编译器就没有什么要额外处理的，就是在建立与释放记录时，处理内存的取用与释放。(请记得 Delphi 为记录布局内存时，并不会把内存的内容用 0 预先填入喔，但配置数组跟新对象实体的内存时就会先填入 0)。

如果记录当中含有由编译器管理的型别(例如字符串、接口)，编译器则需要在初始化(initialization)与终止时(finalization)的源码当中额外注入程序来进行管理。举例来说，字符串型别是需要进行计数参考的，所以如果记录其他数据超过了起始时字符串配置的长度范围，则记录内的字符串就需要减少计数参考的值，这会导致需要把字符串使用的记忆空间释放掉。因此，当我们使用本节里面介绍到的源码，例如受管理的纪录，则编译器在这些源码的前后自动加上 try-finally 区块来确保即使在出现例外情形时，数据仍然不受影响。这个作法已经维持很长一段时间，受管理的记录已经是 Delphi 语言的一部分了。

从 Delphi 10.4 开始，记录型别支持自定的初始化与终止程序，自定的程序会凌驾在编译器默认对受管理记录的处理作业。我们可以宣告一个包含自定初始化与终止程序的记录型别，当中可能包含有我们需要的各种资料型别，此时我们可以自行撰写对应这些数据的初始化与终止源码。这样的记录会被认定为『自定受管理的纪录』。

开发人员可以把传统的纪录改写成自定的受管理记录，只需要加入以下列出的运算符号(一或多个都可以):

- 运算符号 Initialize 会在记忆空间被配置完成后触发，我们可以在当中写入对记录字段数据的初始程序。
- 运算符号 Finalize 会在记忆空间被释放之前触发，我们可以在这当中进行需要的资料清理。
- 运算符号 Assign 会在记录的数据需要被复制到另一个记录或者相

同型别变量的时候触发，我们可以把记录当中的数据复制到另一个记录，如果我们所定义的记录有需要做一些客制处理的话。

笔记

在受管理的记录终止程序被执行的时候，即使是在例外状况中(编译器会自动产生 try-finally 区块)，通常都会使用额外的方法来保护资源分配或实现清除数据的程序。我们会在第九章『透过受管理的纪录还原光标』那一小节里面来看个范例。

带有初始化与终止程序的记录

我们开始从以下这段简单的程序片段来介绍初始化与终止程序，：

```
type
    TMyRecord = record
        Value: Integer;
        class operator Initialize (out Dest: TMyRecord);
        class operator Finalize (var Dest: TMyRecord);
    end;
```

我们需要为这两个类别运算符号写出实作的源码，在以下范例中，我们在这两个函式中记录当时的值。这个范例(是 ManagedRecords_101 范例程序的一部分)中，我会为 Value 字段进行初始化，并记录的数据也包含当中的内存地址，好让我们能逐一检视程序中每个记录所使用的内存位置是否一致：

```
class operator TMyRecord.Initialze (out Dest: TMyRecord);
begin
    Dest.Value := 10;
    Log('Created' + IntToHec(Integer(Pointer(@Dest))));
end;


class operator TMyRecord.Finalize (var Dest: TMyRecord);
begin
    Log('Destoryed' + IntToHex(Integer(Pointer(@Dest))));
end;
```

这个新的建构机制跟传统方式的不同处，在于新的建构机制会自动被触发。例如我们写了以下这段源码，编译器会在该记录需要被初始化与被终止时自动建立 try-finally 区块来管理这个记录型别：

```
procedure LocalVarTest;
```

```
var
    My1: TMyRecord;
begin
    Log(My1.Value.ToString);
end;
```

在 Log 里面我们会看到以下的结果:

```
Created 0019F2A8
10
Destroyed 0019F2A8
```

即使我们用行内变量的方式来写:

```
begin
    var T: TMyRecord;
    Log(T.Value.ToString);
```

在 Log 里面也会看到相同顺序的数据，只是内存位置可能不同了。

指派(Assign)运算符号

一般来说 := (指派)这个运算符号会把范例当中所有的数据进行复制。在受管理的纪录中(例如字符串)的指派与复制程序，也会由编译器进行适当处理。

当我们在记录中宣告了自定的数据字段，以及自定的初始化源码，我们可能得改变一下预设的作法。因此在自定的受管理记录中，我们也可以自定指派(assign)运算符号。这个新的源码，会在源码中出现 := 的时候被触发，只是定义的时候关键词是用 Assign:

```
class operator Assign (var Dest: TMyRecord; const [ref] Src: TMyRecord);
```

运算符号源码的定义必须遵守很精准的规范，包含: 第一个参数必须是引用参数 (前面加上 var)，第二个参数则是常数参数，但必须也要用传址方式传递，所以必须用[ref]关键词或者 var 来达成。如果没这么写，编译器就会提出以下的错误讯息喔:

```
[dcc32 Error] E2617 First parameter of Assign operator must be a var parameter o the container type
[dcc32 Hint] H2618 Second parameter of Assign operator must be a const [ref] or var parameter of the container type.
```

以下是一个可以触发 Assign 运算符号作用的简单范例，

```
var
    My1, My2: TMyRecord;
begin
    My.Value := 22;
    My2 := My1;
```

以上这个程序会建立以下的 Log(在 Log 当中也包含了记录的序号):

```
Created 5 0019F2A0
Created 6 0019F298
5 copied to 6
Destroyed 6 0019F298
Destroyed 5 0019F2A0
```

请注意，在记录被释放的时候，序号的顺序跟建立的时候正好是颠倒过来的，也就是最后建立的记录会最先被释放掉。

把受管理的纪录当成参数

受管理的纪录跟一般记录不同，也可以作为函式的参数或者回传值的型别。以下是几种不同情境的源码宣告：

```
procedure ParByValue (Rec: TMyRecord);
procedure ParByConstVallue (const Rec: TMyRecord);
procedure ParByRef (var RecL TMyRecord);
procedure ParByConstRef (const [ref] Rec: TMyReord);
function ParRetured: TMyRecord;
```

我们接下来就不一一检视每个函式的 log 了(有兴趣的读者可以执行范例程序 ManagedRecords_101 范例)，我们直接汇整相关的信息给大家:

- ParByValue 会建立一个记录，并呼叫指派运算符号(如果有指派参数的话)把数据复制到新建的纪录，并在离开函式的时候把暂存用的纪录释放掉。
- ParByConstValue 不会复制数据，也不会呼叫指派的程序。
- ParByRef 不会复制数据，也不会呼叫指派的程序。
- ParByConstRef 不会复制数据，也不会呼叫指派的程序。
- ParReturned 会透过 Initialize 运算符号建立新的记录，并在回传时呼叫 Assign 程序把新建的记录回传，源码会像是 my1 := ParReturned，并且在回传后离开程序的时候删除暂存用的纪录。

例外处理与受管理的纪录

当程序执行时抛出例外事件，通常记录会被清除掉，即使没有特地写出 try-finally 区块加以处理，这跟对象的处理原则不太一样。这是从基础上就不同，而且这是受管理纪录确实有用的关键特性。

```
procedure ExceptionTest;
begin
    var A: TMRE;
    var B: TMRE;
    raise Exception.Create('Error Message);
end;
```

在这个程序里，会呼叫两次建构函式跟两次解构函式。再强调一次，这是从基础上就不同的，而且是受管理记录的关键特性。

受管理记录的数组

如果我们定义了一个静态的受管理记录数组，这些记录会在宣告的时候就先被呼叫 Initialize 运算符号进行初始化:

```
var
    A1: array [1..5] of TMyRecord; // 初始化时，会从此处呼叫
begin
    Log ('ArrOfRec');
```

这些记录会在离开函式的时候被释放。而如果我们宣告动态的受管理记录数组，初始化的程序将会在该数组的空间被定义时被呼叫(使用 SetLength):

```
var
    A2: array of TMyRecord;
begin
    Log ('ArrOfDyn');
    SetLength(A2, 5); // 在此处呼叫
```

变动型别(Variants)

为了要完整支持 Windows 的 OLE 跟 COM 技术，Object Pascal 在原生数据型别当中就有所谓的松散型别概念，这个型别就是 Variant。即使这个名称可能唤起你对变动记录(我们稍早介绍过)的印象，而且在实作层面上跟开放数组参数有点类似，Variant 是完全不同的功能，实作上也是用特殊的方法来完成的(Windows 开发世界中不常用的语言)。

在这一节里，我不会真的去参照 OLE 跟这个数据型别参照的其他情境(像 data set 介绍时要提及字段存取一样)。我只想从一般观点来讨论这个数据型别。

我会回头来看动态型别，RTTI(RunTim Type Information)并参照第 16 章，在那个章节里面，我也会提到一个相关的型别(但是是型别安全，而且执行速度快很多的喔)，称为 TValue。

变动型别没有型别(Variants Have no Type)

一般来说，我们可以用 Variant 变量来储存任何型别的数据，并且进行数字运算、型别转换。在 Variant 的使用上，自动型别转换会打破 Object Pascal 所遵循的型别安全的通则，而实现出动态型别的概念，这个概念在其他语言中有被介绍过，例如 SmallTalk 跟 Objective-C，在一些脚本语言里面也提供了这样的功能，像是 JavaScript, PHP, Python 跟 Ruby.

变动型别会在运行时间进行型别判别跟运算。编译器不会警告我们在源码里面可能有什么错误，因为它只能做一些延伸测试。整体来看，我们可以把使用了变动型别的源码视为直译源码，因为也只有直译器的源码才会在运行时间才能检查数据的正确性。实务上，这也直接冲击到执行的速度。

现在，我要提出警语，让大家尽量避免使用 Variant 这个型别，我们来看一下 Variant 能做什么。基本上，当我们宣告了一个 Variant 型别的变量:

```
var
    V: Variant;
```

我们就可以把很多不同型别的数据指派给它了：

```
V := 10;
V := 'Hello, World';
V := 45.55;
```

我们一有了 variant 数值之后，就可以把它复制到任何兼容或不兼容的数据型别去了。如果我们把一个数值指派到不兼容的数据型别变量去，编译器并不会在编译的时候指出错误，而会在运行时间进行可接受的型别转换。如果找不到可接受的型别转换，就会抛出一个运行时错误的讯息。技术上来看，Variant 会把型别信息随着数据做储存，允许一些可容许的运行时间型别转换，但会会牺牲速度，且比较不安全。

我们看一下以下的源码(这是 VariantTest 范例项目的一部分)，这是上面源码的一些延伸：

```
var
    V: Variant;
    S: string;
begin
    V := 10;
    S := V;
    V := V + S;
    Show (V);

    V := 'Hello, World';
    V := V + S;  Show (V);
    V := 45.55;
    V := V + S;
    Show (V);
```

有趣吧?以下是输出结果(不意外):

```
20
Hello, World10
55.55
```

除了把储存有字符串的 variant 变量指派给 S 变量，我们也可以把储存有整数、浮点数的 variant 变量指派给 S 变量试试看。再折磨它一下，我们可以用 variant 来计算数值，例如 V := V + S; 这个算式会以 variant 里面储存的数据格式来试着直译多种计算的方法。在以上的源码里面，同一行算式可能对整数、浮点数进行加总，也可能是做字符串连接。

在算式里面放了 variant 是有风险的。如果该字符串里面是储存数字，那么运算就 ok, 但如果不是，就会跑出运行时错误了。如果你不是要特别引人注

意这个情形，就不要使用 Variant 这个型别。还是使用标准的 Object Pascal 数据型别，以及型别检查的规则吧。

深入探讨变动型别(Variants in Depth)

深入了解 Varaint 也是很有意思的，我们再多加一些技术信息吧。Variant 是怎么运作的，我们又能怎么更深入控制一些?RTL(运行时间函式库)包含了变动记录型别:TVarData, 这个型别对内存的控制跟 Variant 一样。我们可以用它来存取 Variant 数据的实际型别。TvarData 结构包含了 Variant 的型别，用 VType、一些保留字段、以及实际值来标示。要记得 null 值的概念，我们可以使用 NULL 来指派给特定值(不是 nil)。

笔记

如果需要更详细的研究 TVarData 这个型别的定义，请参考 RTL 的原始码，在 System 单元文件中可以找到。它的定义跟一般简单的数据结构定义很不一样。我建议具备了足够经验之后再去看关于变动型别的定义比较好。

VType 这个字段的内容会随着我们在 OLE 变量里面所储存数据的型别而有不同。通常 OLE 变量的型别会被称为 OLE 型别或变动型别(variant types)，以下列出所有可用的变动型别：

varAny	varByte	varDate
varEmpty	varInteger	varOleStr
varSingle	varTypeMask	varUString
varArray	varByRef	varDispatch
varError	varLongWord	varRecord
varSmallint	varUInt64	varVariant
varBoolean	varCurrency	varDouble
varInt64	varNull	varShortInt
varString	varUnknown	varWord

以上的型别名称都很直觉，几乎都从字面上就可以知道它的用途。

有很多函式可以用来处理 variant，我们可以用来制作特定的型别转换，或者用来询问一个 variant 变量的真正数据型别(例如 VarType 函式)。这些型别转换跟指派的函式通常都会自动被使用，当我们撰写了使用到 variant 变量的表达式时就会自动使用到。其他支持 variant 的源码，实际上是使用了 variant 数组，再次重申，这个结构几乎只有 Windows 用在 OLE 整合的时候才会使用，其余平台、情形几乎不会有用到的机会。

变异型别很慢(Variants Are Slow)

使用了 Variant 型别的源码会很慢，不只是当我们在做数据型别转换的时候，即使是我们做很简单的两个数字相加也一样。它的速度几乎跟直译器的程序一样慢。为了比较使用 variant 跟使用一般整数的执行速度，我们用相同的算法来实验，我们从 VariantTest 范例项目的第二个按钮程序来看。

这个程序会跑一个循环，计时、然后更新进度列的状态。以下是两个相似循环中的第一个，以 Int64 跟 Variant 来比较：

```
const  MaxNo = 10_000_000; // 10 million
var
    Time1, Time2: TDateTime;
    N1, N2: Variant;
begin
     Time1 := Now;
     N1 := 0;
     N2 := 0;
     while N1 < MaxNo do
     begin
          N2 := N2 + N1;
          Inc (N1);
     end;

    // We must use the result
    Time2 := Now;
    Show (N2);
    Show ('Variants: ' + FormatDateTime ( 'ss.zzz', Time2-Time1) + ' seconds');
```

计时的源码值得看一下，因为未来我们可以用这个方法来做任何可能的效能测试。我们可以看到源码里面用 Now 函式来取得当时的时间值，然后用 FormatDateTime 函式来显示时间的经过，可以用 ss 来显示经过了几秒，或者用 zzz 来显示经过了几个 ms.在这个范例里面，速度的差异非常显著，即使不特别去计算也能注意到:

```
49999995000000          Variants: 01.169 seconds
49999995000000          Integers: 00.026 second
```

上面的数字是我在 Windows 虚拟机上面执行的结果，variant 的结果比直接用整数慢了 50 倍。当然，在每一台计算机上执行的结果都会不一样，但差

别不会太大，用越快的机器来跑，这个倍数说不定会增加的更为显著。在我的 Android 手机上面，执行结果如下(花的时间都更长，这是一定的，手机的运算能力本来就不如桌机):

```
49999995000000      Variants: 07.717 seconds

49999995000000      Integers: 00.157 second
```

在我的手机上，速度比 Windows 慢了 6 倍，事实上执行花了 7 秒多，这样的运行时间一定会让使用者很有感觉。但用整数实作的源码执行起来虽比桌机运行时间多 7 倍，但连 0.2 秒都不到，使用者仍旧很难意识到这个等待的时间。

指标的两三事(What About Pointers)

在 Object Pascal 里的另一个基础数据型别就是指针。有一些面向对象语言已经把这个强大却有些危险的程序结构给藏起来了，不过 Object Pascal 仍旧让程序人员在需要的时候使用它(不过已经不如以前那么常用到了)。

但什么是指标呢？这个名词又是哪来的？跟其他大多数的数据型别不同，指针不储存实际的数据，它只储存变量在内存里面的地址，透过指针，我们可以找到变量的实际使用内存空间，进而得到里面的数据。

笔记 在本书中，这部份属于进阶的章节，在此介绍指标是因为它是 Object Pascal 语言的一部分，而且是所有程序人员都需要知道的核心知识的一部分。虽然它并不是基础的主题，如果您是刚接触 Object Pascal 的话，您可以先跳过这个章节，未来再回来复习即可。如果您过去接触的编程语言都是没有指标的类型，这个短短的章节对您来说读起来也可能很有趣。

指针的定义并不是透过特定的关键词，它是透过特别的符号(^)来宣告的。举个例子，我们可以定义一个指针，指向整数变量：

```
type
    TPointerToInt = ^Integer;
```

我们定义好指针变量之后，就可以把另一个同型别的变量的地址指派进去了，只要透过@符号即可：

```
var
  P: ^Integer;
  X: Integer;
```

```
begin
    X := 10;
    P := @X;
    // change the value of X using the pointer

    P^ := 20;
    Show ('X: ' + X.ToString);
    Show ('P^: ' + P^.ToString);
    Show ('P: ' + Integer(P).ToHexString (8));
```

以上的源码是 PointerTest 范例项目的一部分。假设指针 P 指向了变量 X，我们就可以用 Pˆ 取得 X 的数据，可以读取也可以变更里面的内容。我们也可以独立显示 P 的内容，P 的内容就是变量 X 在内存里面的地址，我们可以把指针转型为数值(透过特殊型别 UIntPtr，请参考下面的提示)，就可以看到地址了。上面的范例程序不显示整数，而是以 16 进位来显示，这在内存的寻址里面比较常见，以下是程序的输出结果(指针的内容会随着编译与执行的结果有所不同)：

```
X: 20
P^: 20
P: 0018FC18
```

警语

要对指标的内容做整数转型，只有在 32 位的平台，并限制在 2GB 之内的地址才会正确，如果我们需要在更大的内存地址当中进行，则我们需要使用 Cardinal 型别。对 64 位平台而言，比较好的作法是使用 NativeUInt 这个型别。然而这是型别的别名，特别用来处理指标的，这型别叫做 UIntPtr，在处理指标的时候，最能够明确的让 Delphi 的编译器知道要做什么。

我来整理一下，当我们宣告了指针变量 P:

- 直接使用指标(P), 我们可以取得 P 所指向的内存地址。
- 使用指标指向(Pˆ), 我们可以取得 P 指向的内存内容。

指针不只可以指向已经配置好的内存地址，它也可以动态指向特定的内存区块，例如我们可以用 New 程序来配置一块新的内存空间，此时就可以用指针储存刚配置好的这块空间的地址。而当不需要再使用的时候，就可以用 Dispose 程序来释放它(行动平台则使用 DisposeOf)。

笔记

内存管理跟 Heap 作业的细节，我们会在第 13 章里面加以介绍。简单的说，Heap 是一大块由操作系统配置给应用程序随机取用/释放的内存空间。除了

New 跟 Dispose 之外，我们也可以用 GetMem 跟 FreeMem 来取用额外的内存空间，这些作法都需要开发人员提供要配置的内存大小(编译器在使用 New 跟 Dispose 的情境下，会自动配置所指定的空间)。在编译阶段还无法确定需要多大空间的时候，GetMem 跟 FreeMem 则会比较方便。

以下程序片段是用来示范如何使用随机配置内存：

```
var
    P: ^Integer;
begin
    // Initialization
    New (P);
    // Operations
    P^ := 20;
    Show (P^.ToString);
    // Termination
    Dispose (P);
```

如果在使用后没有把内存释放掉，我们的程序最后就会把可用的内存用尽，然后挂掉。没有把配置的内存释放掉的错误，通常称为内存泄漏(Memory leak)。

警示 要让上面的范例程序安全一点，我们可以用 try-finally 区块，这个主题我们会在第九章进行介绍。

如果一个指针变量没有值，我们可以指派 nil 给它。我们可以先检查指标的内容是否为 nil，如果不是 nil，我们才能够去读取其指向的内存空间内容。Object Pascal 提供了一个名为 Assigned 的函式，让我们可以方便进行这样的测试。

这样的测试很常用，因为存取未被配置的内存空间内容，会导致内存违规存取而使程序错误(不同操作系统对内存违规存取的处理各有不同的作法，Windows 会跳出一个错误窗口，iOS 跟 Android 则是直接关闭该程序)：

```
var
    P: ^Integer;
begin
     P := nil;
     Show (P^.ToString);
```

我们可以执行 PointerTest 范例项目来看看发生这个错误的反应，Windows 的错误窗口会显示如下的信息：

```
Access violation at address 0080B14E in module 'PointersTest.exe'. Read of address 00000000.
```

为了让存取指标更为安全，我们可以加入检查，看看指标内容是否为 nil:

```
if P <> nil then
  Show (P^.ToString);
```

另外，在 Object Pascal 里面提供的 Assigned 也可以用来检查这种情形:

```
if Assigned (P) then
  writeln (P^.ToString);
```

笔记 Assigned 倒不是一个真的函式，因为这个关键词会让编译器产生适当的检查源码。Assigned 也可以用在检查程序型别变量(或者方法参考)是否有被指派，不用真的执行看看该程序或方法是否存在。

Object Pascal 也定义了一个名为 Pointer 的数据型别，这个型别可以指向未定义型别的指针(就像 C 语言里面的 void*)如果我们使用了未定义型别的指标，我们就得用 GetMem，不能用 New 来配置内存空间。因为用 New 来配置，系统会以 New 所要配置的数据型别自动计算大小来配置，但为定义型别的指针无法自动由编译器判别需要多少空间。而我们使用 GetMem 来配置空间时，可以指定需要配置的内存空间大小。

事实上，在 Object Pascal 里面不常需要使用指标，但指标是这个编程语言里面一个有趣的进阶功能。能够提供这个功能，就能让我们实作出一些可以直接呼叫操作系统层级，高效率的 API 跟函式了。无论如何，了解指标对进阶程序设计以及完全了解 Delphi 的对象模型是很重要的(在实作层面中，就是透过指针来实现的，一般也会称为参考)。

警示 当某个变量内容储存的一个指向第二个变量的指针，而第二个变量已经脱离其生命周期范围(例如被动态释放了)，该指针所指向的数据可能是未被使用的内存空间，或者是已经被用来储存其他数据了。这可能会让侦错程序变得很难啊。

档案型别，还有谁有提供?(File Types, Anyone?)

本章最后的一节，我们来介绍 Object Pascal 数据型别建构的过程，就是档案(file)型别。档案型别是用来表示实体磁盘档案，这也是原始 Pascal 语言里面就具备的功能，相较起来，目前已经很少有旧的或现代的编程语言会把档案的概念作为基本的数据型别了。Object Pascal 语言也提供了 file 关键词，用来代表档案型别，就像 array 或 record 一样。我们可以用 file 来定义一个新的型别，然后用这个型别来宣告新的变量:

```
type
   IntFile = file of Integers;
var
    IntFile1: IntFile;
```

我们也可以直接用 file 这个关键词，不用指定该档案的内容型别，直接定义一个档案。当然我们也可以使用 TextFile 型别，这个型别定义在 RTL 里面的 System 单元，使用来宣告使用 ASCII 字符的档案(或者更精确一点，是定义 byte 组成的档案)。

最近这几年来，直接使用档案已经越来越不常见了，不过这个功能还是存在的。在 RTL 里面已经提供了很多种可以用来管理二进制跟文本文件的类别(例如，支持 Unicode 的文本文件)。

Delphi 应用程序一般来说都会使用 RTL 的 stream 类别(TStream 以及衍生的相关类别)来处理许多跟档案相关的读写动作。Stream 用来表示虚拟档案，可以对应到实体档案、内存区块、网络信道，以及任何其他连续的 byte 组合。

我们还是可以看到许多旧时代的档案管理函式，那些是提供给命令字符模式(DOS 命令字符)的程序开发之用，我们可以使用 write, writeln, read 等相关函式来对特定型态的档案进行读写的动作，这些都是使用标准输入、标准输出加以提供的(C 跟 C++也对输入输出有类似的支持，其他的编程语言也提供了类似的服务)。

06:关于字符串

字符、字符串已经是所有编程语言最常处理的数据型别了。在 Object Pascal 里，字符串的处理非常简单、快速，功能也非常强大。就算字符串基本上很容易掌握，且我们在前几章的范例中已经用了很多次字符串作为输出，但在幕后，字符串的处理还是比第一眼的印象来的复杂。文字处理还是使用了一些值得我们仔细了解的主题。要完整了解字符串处理，我们还需要知道关于 Unicode 的表示法、了解字符串怎么跟字符数组对应，还要学习一些在执行时期函式库(RTL)里面跟字符串处理相关的函式，包含把文字存放到档案里，以及从文本文件里面加载文字的方法。

Object Pascal 在文字处理上有一些选项，提供了不同的资料型别跟处理方法。本章的焦点会放在标准的字符串型别上，但我们还是会花一些时间来介绍目前桌机版操作系统仍在使用的旧版字符串型别(例如 AnsiString)。在开始主题之前，我们先从基本开始吧:Unicode 表示法。

Unicode:全世界的字母

Object Pascal 的字符串管理是以 Unicode 字符集为中心的，尤其是以 UTF-16 为主。在我们深入技术细节之前，值得先花一些时间来看一下 Unicode 的标准。

Unicode 的设计理念，是要用单一一个字符集把世界上所有的文字都包含在内，这样的作法把许多复杂的问题单纯化了，但要实现这个理想的过程却极为复杂。在 Unicode 的设计中，是要以同一个叙述方式，同时包含图形化的表示方法，以及独特的数值(称之为 Unicode 的字码)。

笔记 Unicode 联盟的网址是 http://www.unicode.org，在这个网站中，提供了非常多的文件，最具代表性的文件是 ”The Unicode Standard” 这本书，这本书可以在网上找到：http://www.unicode.org/book/aboutbook.html.

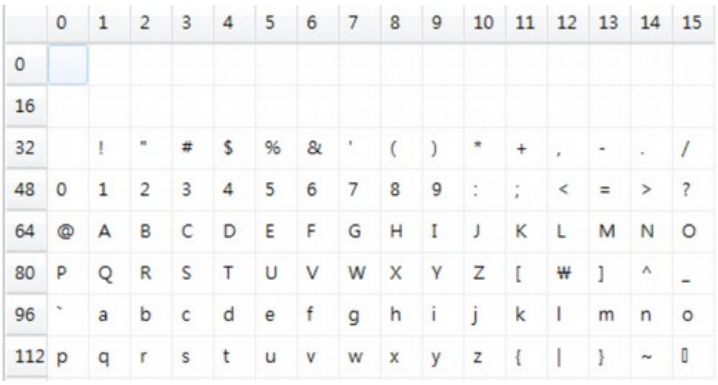
并不是所有程序人员对 Unicode 都熟悉，而且很多程序人员直到今日还在使用比较旧的文字编码方法，例如 ASCII (或者称之为 ISO 编码)。我们很快介绍一下这些旧标准，您就会对 Unicode 的特点(或是其复杂度)更为感激了。

旧的文字编码:从 ASCII 到 ISO 编码

字符编码始于美国标准信息交换编码(American Standard Code for Information Interchange – ASCII)，这个标准是在 1960 年代早期发展出来，作为计算机字符的编码标准。当时这个编码法只涵盖了 26 个英文字母 (包含大写、小写)、10 个数字、常用的标点符号，以及一些控制字符(我们到今天还有使用到)。

ASCII 使用 7 bit 来表示系统常用的 128 个不同字符。只有从#32(空格符)到#126(~：波浪号)是肉眼可见的字符，请见图 6.1 (从 Windows 平台的 Object Pascal 应用程序中解译出来的)。

图 6.1: ASCII 当中的可视字符集



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	₩	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	▯

ASCII 当然是所有字符编码的基础(这基本的 128 个字符目前仍旧是 Unicode 编码的核心部分)，后来很快的就以这 128 个字符做延伸，改以 8 bit 做为储存字符编码内容的数据长度，让 ASCII 多了另外 128 个字符。

目前所遭遇的问题，是全世界有太多种不同的语言，但并没有简单的方法让我们可以知道其他语言的文字要怎么被囊括在这个集合当中(就像过去的 ASCII 可以把英文字母跟常用符号包含在内一样)。为了让这个情形得到解决，Windows 里面内载了许多种不同的文字集合，我们后来称之为 code page，透过 code page 的定义，我们可以把不同语言的文字依照不同语系的 Windows 版本进行显示。除了 Windows 的 Code page 之外，还有许多种类似的分页标准，用来对不同语系的文字进行定义与编码，这些分页方式后来都成为了 ISO 国际标准的一部分。

目前仍旧最常被使用到的 ISO 文字编码标准仍然是 ISO 8859，在这个标准里面定义了好几个区域集合。最常用到的集合(大多数西方国家几乎都使用这个集合)是 Latin 这个集合，也被称为 ISO 8859-1。

笔记

即使 Windows 1252 code page 跟 ISO8859-1 已经极其接近，但仍然没有完全兼容。Windows 在里面加入了一些额外的字符，例如€符号等，这些额外的字符被添加在 128 到 150 区域中。除了跟 Latin 集合的所有值不同，Windows 的这些延伸值也跟 Unicode 的字码无法兼容。

Unicode 字码跟字形

如果要精确一点，我们应该再多介绍另一个概念，就是字码。事实上，有时多个字码可能指向同一个字形(可以辨识的文字)。通常这不一定是单一一个文字，可能是文字，或者文字跟符号的组合。举例来说，如果我们有一串字码代表拉丁文字的 a(#$0061)，尾随着重音符号(#$0300)，这样的组合应该会是代表一个音标符号。

在 Object Pascal 的编码名词中，如果我们写了以下的源码(这是 CodePoints 范例项目的一部分)，这个讯息就只会包含一个文字，如图 6.2 所示：

```
var
    str: String;
begin
    str := #$0061 + #$0300;
    ShowMessage (str);
```

图 6.2: 多个 Unicode 字码指向单一一个符号

在这个例子里，我们是写了两个字符，代表两个字码，但实际上两个字码的组合只指向一个单一符号。事实上，在拉丁字符集里面，我们可以用一个字码直接指向前述的这个字符(字符 a 加上重音符号的字码是$00E0)，在其他的字符组合，Unicode 的字码是唯一能够取得刚刚这个字符的方法(并且能够正确输出喔)。

即使显示的是一个重音音标，并没有可以自动把这个资料进行转换的方法(只能够适当的显示)，因此字符串内部处理对字符 à 仍旧各有不同的作法。

笔记 要正确显示由多个字码所组合描述的字符符号，必须依赖操作系统的特别处理，也必须使用文字绘制技术。所以我们应该有发现到，在操作系统上并不是所有的字都可以正确的被显示出来。

从字码到字节（UTF）

ASCII 使用直接而简单的方法处理字母跟其编码数值之间的对应，Unicode 则使用比较复杂的方式。就像我已经提过的，在 Unicode 的每个字符，都有一个关连的字码，但对应到要显示的字符，则通常要复杂的多。

在 Unicode 的原理背后，比较容易让人混淆的元素之一，是同一个字码 (或者说是 Unicode 字符的编码数值)在档案、内存、实际储存在媒体上的时候可能有多种方法加以表示。这个问题主要是因为所有只存在唯一对应的 Unicode 的字码实际上都是用了 4 bytes 来表示的。这个作法实现了固定长度表示字符的思路(所有字符都是以固定长度的数值加以表示)，但大多数的开发人员会觉得这样对内存用量和处理程序上都太昂贵了。

笔记 在 Object Pascal 里面，Unicode 字码是直接以 4 bytes 的 UCS4Char 数据结构来直接表示的。

这也是为何 Unicode 的标准里还定义了其他表示方法，会使用比较少的内存，但在这些表示方法里面，每个符号储存所使用的空间就会随着字码而有不同了。这个想法为最常用的元素提供了短一点的表示方法，而比较不常用的元素则会用比较长的储存空间。

Unicode 字码在实际储存时不同的表现格式，被称为Unicode Transformation Format (简称UTF)。这些格式是算法的对应，它们是Unicode标准的一部分，可以让每个字码(每个绝对的数值都对应到一个字符)对应到每个文字所拥有的代表数值。我们要注意到，这个对应关系是双向的，可以在不同的表现方式中双向转换。

Unicode标准定义了三种UTF格式，分别以不同的长度来为之命名：8, 16, 32。了解这三种格式最多需要4 bytes来为每个字码编码其实还蛮有趣的。

- UTF-8 把每个字符转换成变动长度来储存，每个字符的编码可能使用1到4个byte不等。UTF-8也是目前HTML跟类似的协议最常用的，因为UTF-8最为精简，所有ASCII的字符几乎完全兼容于UTF-8的格式。

- UTF-16则是很多操作系统(包含Windows跟Mac OS X)最常用的格式。这个格式可以把绝大多数的文字用两个byte来表示，相对来说很精简，而且处理上速度也很快。

- UTF-32在处理上速度最快，因为所有字符编码都是相同的长度，但是在内存用量上最浪费，也因此在实务应用上很少被使用。

UTF-16 最常被误解为可以用2 bytes直接对应到所有字码，但因为Unicode包含了超过 10万个 字码，我们直接心算一下，就知道总量超过了2 bytes 所能表示的元素上限 (64K)，所以UTF-16是不可能完整对应到所有 Unicode 字码 的。然而，开发人员常常只需要用到Unicode的一部分子集合，所以会把用到的字符塞到每两个byte定义一个字符，用固定长度来表示。在早期，这个Unicode的子集合被称为UCS-2，现在我们会常看到UCS-2对应到基础多语系平面(Basic Multilingual Plan, BMP)。然而这仍然只是Unicode的子集合(许多语系平面之一)。

笔记 有一个和多位表示法(UTF-16 跟 UTF-32)相关的问题，就是到底哪个位是起始位？根据 Unicode 标准的定义，怎么排列都可以，所以我们可以定义 UTF-16 BE(big endian)或者 LE(litte-endian)，UTF-32 也一样。Big-endian 位序列，是最高位优先(Most Significate Byte first，MSB 优先)。而 little-endian 则是最低位优先(Least Significant Byte first, LSB 优先)。位序列通常会标注在档案的最前面，称之为位序列记号(Byte Order Mark, BOM)。

位序列记号 (BOM)

当一个文本文件里面储存 Unicode 字符，有个方法可以注记这个档案使用哪一种 UTF 格式来储存 字码。这个信息会储存在档案的开头，称为位序列记号(Byte Order Mark, BOM)。这可以视为用以辨识Unicode使用何种序列格式(LE 或者 BE)的签章。以下的列表汇总了一些不同的BOM，包含了2, 3, 4个 bytes的长度:

00 00 FF FE	UTF-32, big-endian
FE FF 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF8

我们会在本章后段介绍 Object Pascal 如何用它的串流式类别管理 BOM。BOM 会出现在档案的最开头，Unicode 数据则会立刻接在 BOM 后头。所以，

一个内容为 AB 的 UTF-8 档案，它的前五个 Bytes 会是这样的(3 个是 BOM, 2 个是 AB 这两个字符):

```
EF BB BF 41 42
```

如果文本文件开头没有出现 BOM，这个档案就会被认为是 ASCII 文本文件，但这档案也可能包含其他种类编码的文字。

笔记 在另一种用法，当我们从 Web，或者从其他网络协议接收数据的时候，我们可能得从该协议的其他定义中来区分编码方式，就不能用 BOM 了，反而透过这些协议取数据的时候 BOM 有时还会让编码识别造成困扰。

看清楚 Unicode

我们有可能用像图 6.1 那样的列表来把所有的 Unicode 字符全部列出来吗? 我们可以用基础多语系平面(Basic Multilingual Plan, BMP)来显示字码，排除掉代理对应(surrogate pairs)。

笔记 并非所有数值都是真实的 UTF-16 字码，因为有些字符(称之为代理对应：surrogate pairs)的数值并不是合法的数值数据，这些数值是大于 65535 的数字，超过了 2 bytes 可以描述的范围。代理对应当中一个很好的例子，就是乐谱里面的低音部记号 𝄢 它的字码是 3 个 Bytes 长，内容是 1D122，以 UTF-16 表示的话，就需要以两个数值来表示:D834 然后接着 DD22。

要完整把所有这个 BMP 的元素显示出来，需要一个 256x256 的矩阵，在屏幕上面很难完整显示。这也是为何 ShowUnicode 这个范例项目里面在两页之间用了 tab 来做分隔的原因：第一个 tab 显示 256 个区块的主要索引值，第二个 tab 则显示真实 Unicode 的元素。每次显示一个区块。这个范例当中的用户接口比本书其他范例程序都多了一些，如果您只是对 Unicode 有兴趣的话，可以忽略它的源码，只看它的输出部分。

在这个程序的开头，在TabControl的第一个分页当中，我们放了一个TListView组件，这个ListView有256个项目，每个项目被点击的时候，都会显示一组256个Unicode字符。以下是在onCreate这个事件处理程序的源码，我们用了一个简单的程序来显示每一个元素，产出的内容请参考图6.3 :

```
// Helper function
function GetCharDescr (NChar: Integer): string;
```

```
begin
    if Char(NChar).IsControl then
       Result := 'Char #' + IntToStr (NChar) + ' [ ]'
    else
        Result := 'Char #' + IntToStr (NChar) +
                    ' [' + Char (NChar) + ']';
end;


procedure TForm2.FormCreate(Sender: TObject);
var
    I: Integer;
    ListItem: TListViewItem;
begin
     for I := 0 to 255 do // 256 pages * 256 characters each
     begin
         ListItem := ListView1.Items.Add;
         ListItem.Tag := I;
         if (I < 216) or (I > 223) then
            ListItem.Text :=  GetCharDescr(I*256) + '/' + GetCharDescr(I*256+255)
         else
              ListItem.Text := 'Surrogate code points';
     end;
end;
```

图 6.3: ShowUnicode 范例的第一页，程序画面列出了很长的 Unicode 字符区块

请留意源码里面是怎么储存这些分页的号码的，我们用了 ListView 项目中的 Tag 这个属性，这个属性储存了稍后我们用来显示该项目对应的区段所需

的信息。当用户点击任一个 ListView 的项目时，应用程序就会切换到 TabControl 的第二个分页，把该区段的 256 个字符显示在字符串矩阵上。

```
procedure TForm2.ListView1ItemClick(const Sender: TObject; const AItem: TListViewItem);
var
    I, NStart: Integer;
begin
    NStart := AItem.Tag * 256;
    for I := 0 to 255 do
    begin
        StringGrid1.Cells [I mod 16, I div 16] :=
            IfThen (not Char(I + NStart).IsControl, Char (I + NStart), '');
    end;
    TabControl1.ActiveTab := TabItem2;
```

这里的 IfThen 函式是用来作两种测试：如果传入的第一个参数为真(或说判别式成立)的话，这个函式会把第二个参数回传。反之，则以第三个参数回传。在这里的测试使用了 InControl 这个字符型别助手的方法，用来过滤掉无法显示的控制字符。

笔记

这里用到的 IfThen 函式，作用很像 C 语言阵营里面的?:运算符号。同样的函式，有一个用来测试字符串的版本。另一个则是用来测试整数的版本。测试字符串的版本放在 System.StrUtils 单元文件里面，测试整数的版本则是放在 System.SysUtils 里面。

由范例程序所输出的 Unicode 字符矩阵，我们用图 6.4 来做显示。请注意到输出的字符，会随不同操作系统当中的显示有所差异，即使选择的是同样的区块，Window 跟 Mac 的显示仍可能有差异。

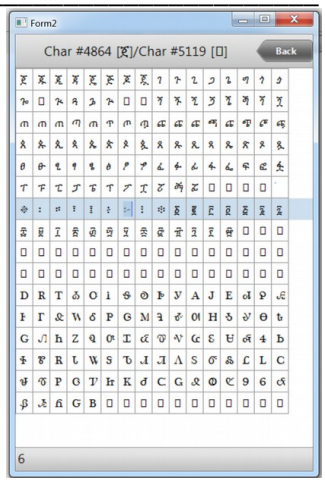


图 6.4: ShowUnicode 范例的第二页，程序画面显示了实际的 Unicode 字符

再度介绍字符型别

在简单介绍过 Unicode 之后，我们回头来看一下这个一章的主题，也就是 Object Pascal 式如何管理字符跟字符串的。我们在第二章已经介绍过了字符型别，也提到了型别助手的相关函式可以在 Character 单元当中引入来使用。现在，我们也对 Unicode 有了更完整的认识，是时候深入多看些细节了。

首先，字符型别并不是一成不变的记录 Unicode 的字码。事实上字符型别对每个元素都使用两个 bytes 来记录其数值。当字符以 Unicode 的基础多语系平面(Basic Multilingual Plan, BMP)来表示字码的时候，同时也可以透过代理对应(surrogate pairs)来显示字码。

技术上来说，我们可以使用另一个型别来直接记录任何一个 Unicode code，这就是 UCS4Char 型别，它使用了 4 bytes 来记录一个数值。这个型别不常被使用到，因为它需要使用更多的内存空间，我们可以看到 Character 单元里面提供了好几种不同的处理方法给这个型别使用。

回到字符型别，请记得它是一个有序型别(可能是该类型别当中相对比较大的)，所以这个型别本身有顺序性，并且我们也可以把 Ord, Inc, Dec, Hight, Low 这些函式套用在字符型别上面。绝大多数的延伸功能，都包含在型别助手里面，它们不是基本的系统 RTL 单元的一部分，所以我们得自己引入 Character 单元。

使用 Character 单元来处理 Unicode

大多数对于 Unicode 字符的特殊处理(当然也包含 Unicode 字符串)都被定义在 System.Character 这个单元当中。这个单元文件定义了 TCharHelper 这个型别助手，用来提供 Char 型别协助，让我们可以直接从 Char 型别的变量中直接呼叫这些处理方法。

笔记 Character 单元定义了一个名为 TCharacter 的纪录型别，在这个型别当中定义了一系列的静态类别函式，搭配有一些全局的子程序跟这些方法对应着。这些函式已经比较老，有些都已经被弃用了。在 Unicode 层级来处理 Char 型别，这些方法却仍然是很有帮助的，所以透过类别助手提供。

这个单元文件也定义了两个很有趣的列举型别。第一个称为 TUnicodeCategory，它对应了很多分类的字符，例如控制字符、空格符、大

写字符、小写字符、数字、标点符号、数学符号等等。第二个则是TUnicodeBreak，它定义了许多不同的空白、分隔符。如果我们惯用 ASCII 各种处理方法，那改用 Unicode 就将是个很大的改变。例如，在 Unicode 里面，数字不只是从 0 到 9 的字符，空白也不仅仅是字符#32，在其他许多不同语系的字符分页中，都可能也有相同意义的字符。

字符型别助手提供了超过 40 个不同的测试与处理方法，可以用来：

- 取得字符的数字表示值(GetNumericValue)
- 询问目录(GetUnicodeCategory)，或者检查字符是否属于不同目录当中的任何一种(IsLetterOrDigit, IsLetter, IsDigit, IsNumer, IsControl, IsWhiteSpace, IsPunctutation, IsSymbol, 以及 IsSeparator)。在前一个范例中，我就使用了其中的 IsControl 判断方法。
- 检查是否是大写或小写(IsLower, IsUpper)或者转换大小写(ToLower, ToUpper)
- 确认是否属于 UTF-16 代理对应(surrogate pairs)(IsSurrogate, IsLowSurrogate, IsHighSurrogate)，也可以用不同方式转换代理对应字符。
- 把字符转换成 UTF-32 或从 UTF-32 字符转换为字符型别(ConvertFromUtf32, ConverToUtf32)，或者转换成 UCS4Char 型别(ToUCS4Char)
- 确认是否为特定字符里面的成员(IsInArray)

请留意以上所有的动作，是对字符型别全体内容一体适用，不是只对特定变数有效。所以我们可以用字符型别当做起始关键词，来使用上述的方法，我们将在底下的程序范例里面介绍。

为了把这些方法对 Unicode 字符做点实验，我们提供了范例项目，名称是 CharTest。这个范例当中的一个实验，是对 Unicode 字符呼叫大写、小写操作方法的实验。事实上在 RTL 里面，传统的 UpCase 方法只能对基本的 26 个 ASCII 表示法的英文字母有效，对于 Unicode 字符来说就没有所谓的大小写之分。(因为并非全世界的文字都有大小写之分，所以这个方法无法一体适用于所有的 Unicode 字符)

为了测试这个情境，在 CharTest 这个范例项目里面，我加入了以下的程序片段，试着把一个音标符号转成大写:

```
var
   Ch1: Char;
begin
```

```
    Ch1 := 'ù';
    Show ('UpCase ù: ' + UpCase(Ch1));
    Show ('ToUpper ù: ' + Ch1.ToUpper);
```

传统的 UpCase 没办法把拉丁音标符号进行转换，而 ToUpper 方法可以做的很完美：

```
UpCase ù: ù
ToUpper ù: Ù
```

在字符型别助手里，包含了许多跟 Unicode 相关的功能，像是在接下来的范例源码里面所介绍的，这些源码定义了一个字符串，字符串里面的文字也涵盖了在 BMP(低于 Unicode 64K 以下的字码)以外的字符，以下的源码也是 CharTest 范例项目的一部分，我们对字符串中的不同字符做了一些测试：

```
var
   str1: string;
begin
    str1 := '1.' + #9 + Char.ConvertFromUtf32 (128) +
              Char.ConvertFromUtf32($1D11E);
    ShowBool (str1.Chars[0].IsNumber);
    ShowBool (str1.Chars[1].IsPunctuation);
    ShowBool (str1.Chars[2].IsWhiteSpace);
    ShowBool (str1.Chars[3].IsControl);
    ShowBool (str1.Chars[4].IsSurrogate);
end;
```

显示结果的函式在这个范例中是一个改写过的版本：

```
procedure TForm1.ShowBool(value: Boolean);
begin
    Show(BoolToStr (Value, True));
end;
```

笔记 Unicode 字码 $1D11E 是乐谱符号的 G 调符号。

Unicode 字符常数 (Unicode Character Literals)

在前几个范例中，我们已经知道可以直接把独立的字符常数或字符串常数指派到字符或字符串变量里面了。我们也可以直接透过数字把字符表示指派给

字符串或字符变量，只需在数字前加上一个井字号(#)，不过这样可能会发生意外。
为了兼容以前的源码，直接输入的字符常数内容会依照它们的内容来进行转换。以下的程序片段就是把数值 128 指派给字符串，这个字符串就会显示成欧元符号(€):

```
var
   Str1: string;
begin
    Str1 := #$80;
```

以上的源码并没有遵循 Unicode 规范，上面这个符号的字码是 8364。事实上，这个数值并不是从 ISO 官方的 codepage 定义而来，而是从微软为 Windows 实作的数据得来的。为了让已经存在的源码能够更轻松的转移到 Unicode 兼容的功能。Object Pascal 编译器会把 2 码的字符串数值直接当成 ASCII 字符来处理(当然这得视用户的计算机所使用的 code page 而定)。如果我们把数值转成字符，然后再加以显示，数值就会变成其代表的正确字符了，够惊讶吧？所以，以下的源码执行以后：

```
Show (Str1 + ' - ' + IntToStr (Ord (Str1[1])));
```

就会得到这样的输出：

```
€ - 8364
```

假如我们想要完整的把原有的源码全部转换为 Unicode, 把 ANSI 数据全部放弃，我们可以直接修改编译器设定，只要直接在源码里面加入这个设定: #HIGHCHARUNICODE 即可。这个设定会决定从 #$80 到 #$FF 的数值要被怎么处理？我们稍早曾介绍过默认的选项(关闭: OFF)，如果我们把这个设定开启了，同样的程序就会输出如下的结果：

```
€  - 128
```

这些数值会被直译为实际的 Unicode 字码，而输出值就会变成看不见的控制字符。另一个处理特定字码(或者任何#$FFFF 以下的 Unicode 字码)的方法，则是使用四个数字加以表示:

```
Str1 := #$0080;
```

这样也仍然不会显示欧元符号，除非我们把$HIGHCHARUNICODE 这行设定拿掉。

笔记 以上的源码只能在 US 或西欧字符的设定下正常运作。如果设定为其他语系，在 128 到 255 之间的字符会显示为不同的样子。

但我们可以用 4 个数值来表示远东的文字，例如以下的程序可以显示两个日文文字：

```
str1 := #$3042#$3044; Show (str1 + ' - ' + IntToStr (Ord (str1.Chars[0])) +
    ' - ' + IntToStr (Ord (str1.Chars[1])));
```

上面的文字可以显示出以下的结果：

```
あい- 12354 – 12356
```

笔记 あい可以译为"相遇"，但我不是完全确定我找到的这个翻译是不是正确，因为我不懂日文，所以或许会有错。

以上为原作者的文字。あい要看在日文裡面的前後文跟漢字的轉換，あい可以對應漢字的"愛"，或者逢い。
逢い：就像原作者找到的翻譯，就是相逢、遇見的意思。
愛: 這不用我多說了……

我们也可以用超过#$FFFF 的数值，这些数值会自动被转换为适合的代理对应(surrogate pairs)。

那单位元的字符呢?

一如我在前面的章节里面提过的，Object Pascal 语言已经把 Char 型别对应为 WideChar 型别，但还是保留有 AnsiChar 型别的定义，主要是为了跟已存在的源码兼容。建议的作法是使用 Byte 型别来处理单位元的数据结构，虽然 AnsiChar 型别对于处理单位元的字符也很方便。

在行动平台上，Delphi 已经有好几个版本不支持 AnsiChar 型别了，但从 10.4 开始，Delphi 编译器又开始让 AnsiChar 型别在各种平台都能使用。但在使用特定平台的 API，或者是要储存档案的时候，我们仍然应该避免使用单位元的字符型别，哪怕现在已经又提供支持了。虽然事实上使用单位元字符处理会比使用多位字符的处理耗用较少的内存，但使用 Unicode 编码才是比较安全也是比较建议的作法。

字符串数据类型 (String Data Type)

在 Object Pascal 里的字符串数据类型比简单的字符数组更为完整，而且其功能比大多数有提供字符串型别的编程语言的字符串功能更为强大。在这一小节里面，我也会介绍在这个型别当中的关键观念。接下来这个章节里，我们会更深入的介绍这些功能。

在以下的列表中，我会列出在 Object Pascal 里面字符串型别运作的关键概念(请记得，我们可以不用了解观念，也仍然可以把字符串使用的很好，且内部运作的规则也是非常透明的)：

- 字符串型别所使用的数据，是从 heap 里面*动态配置*来的。字符串变量就是实际数据的参考。但我们完全不用担心这些细节，因为编译器会帮我们把这些细节都处理好。就像使用动态数组一样，我们宣告一个新的字符串变量时，变量的内容是空的。
- 我们有很多方法可以把数据存放到字符串变量里面去，我们可以实际去*配置一块指定大小的内存空间*-透过呼叫 SetLength 函式，它所需的参数，是我们需要配置的字符数量，(当然，每个字符是 2 bytes)。当我们扩展一个字符串的时候，已经存在该字符串的数据会被保留的好好的(不过这些数据可能被搬移到不同的内存空间去)。当我们把字符串缩小的时候，原本的字符串内容就有部分可能会遗失。我们几乎不用去重新设定字符串的长度。仅有的情境是当我们要把一个字符串的缓冲区传递给操作系统特定的函式使用。
- 如果我们需要*增加*字符串在内存的空间(例如做两个字符串连接)，但有可能原来的内存地址已经无法再扩增(可能后面的空间被其他变量用到了)，此时连接字符串的动作就需要重新配置一段足以容纳连接之后的字符串数量的内存，然后把两段字符串都复制到新的这段内存空间，然后把旧的那段给释放掉。
- 要清除掉我们已经不再使用的字符串，我们可以直接指派一个''给字符串变量，或者我们可以用 Empty 这个常数，它也同样代表了''这个字符串。
- 根据 Object Pascal 的规则，*字符串的长度*(我们可以透过呼叫 Length 函式来取得)代表该变量里面合法字符的数量，而不是配置的数量。跟 C 语言不同的是，C 语言的字符串必须用#0 作为字符串结尾，而所有版本的 Pascal 都直接配置了字符串内容所需的内存大小。然而我们仍然可以在字符串的尾端找到字符串结尾符号。
- Object Pascal 的字符串使用了参考计数(Reference-Counting)机制，这可以持续追踪一共有多少个字符串变量在参考这段内存。参考计数会

在某个字符串变量再也没有被使用的时候释放内存。也就是说，当没有任何变量参考该段内存的时候，参考计数的数值就会变成 0.

- 字符串使用了”**写入时才复制**”的技术，这是很有效率的。当我们把一个字符串指派给另一个字符串，或者把字符串当成参数传递给函式时，没有任何数据被复制，只是参考计数会被增加而已。然而，如果其中一个参考到这个内存的变量试图改变内容，这个时候系统就会把该段内存复制到另一个空间，然后改变这个新空间的内容，原本的内存内容不会被改变。
- ***字符串连接***的作法，对已经存在的字符串是很快的，且没有明显的副作用。当有其他替代的要求时，字符串链接是很快且很强大的。但这对于目前已经存在的许多编程语言来说，就没这么简单。

我猜以上的描述可能会让许多人迷惑，所以我们来实际看看字符串的使用。我们待会来看一些以上提到的功能的示范，包含参考计数跟写入时才复制。在我们开始介绍前，我们先来回头看一下字符串型别助手的功能以及一些其他在基本 RTL 上面的字符串管理功能。

首先，我们先来说明前面列表里面的一些名词。因为字符串的处理已经相当无缝化，所以很难完整解析到底发生了什么事。除非我们开始深入观察字符串的内存结构，这一点我们在本章后续会持续深入，但这个主题目前有点太深奥。所以我们开始看一些简单的字符串处理吧，从 Strings101 应用程序项目开始:

```
var
   String1, String2: string;
 begin
   String1 := 'Hello world';
   String2 := String1;
   Show ('1: ' + String1);
   Show ('2: ' + String2);
   String2 := String2 + ', again';
   Show ('1: ' + String1);
   Show ('2: ' + String2);
end;
```

当上面这段源码执行，会让我们知道，当我们把同一个字符串指派到两个不同的字符串变量时，变更其中一个的内容，另一个并不会被影响到。在这个例子里，String1 并不会被 String2 的内容变动所影响到。

```
1: Hello world
```

```
2: Hello world
1: Hello world
2: Hello world, again
```

在我们开始下一个范例之前，先说明一下，一开始的字符串指派动作，并不会导致字符串的完整复制，复制的动作会被延迟，这个功能叫做”写入时才复制”。

另一个要介绍的重要功能，则是字符串长度是如何被管理的。如果我们查询一个字符串的长度，我们所得到的响应，会是字符串所使用的实际长度值(这个长度的数据是储存在字符串的 meta data 里面，会使得这个查询动作非常快就能完成)。但如果我们呼叫 SetLength 函式，这个函式会直接配置内存，但不会进行初始化。这常被用在把字符串当成呼叫外部系统函式的参数时。相反地，如果我们需要一个空字符串，我们可以使用虚拟建构函式 Create。最后，我们可以用 SetLength 来切断一个字符串。以上所提到的都包含在以下的源码里面：

```
var
    String1: string;
begin
     String1 := 'hello world';
     Show(String1);
     Show ('Length: ' + String1.Length.ToString);

     SetLength (String1, 100);
     Show(String1);
     Show ('Length: ' + String1.Length.ToString);
    String1 := 'Hello world';
    Show(String1);
    Show ('Length: ' + String1.Length.ToString);

    String1 := String1 + String.Create(' ', 100);
    SetLength (String1, 100);
    Show(String1);
    Show ('Length: ' + String1.Length.ToString);
```

输出的结果会像以下这样:

```
Hello world
Length: 11
```

```
Hello world
Length: 11
Hello world
Length: 100
```

我想在本节里面强调的第三个概念，是空字符串的概念。所谓的空字符串，是指字符串变量所存的文字内容是空的。不论是在指派或是比对的用途，我们都可以用两个单引号，或者用特定的函式来表示空字符串：

```
var
  String1: string;
begin
  String1 := 'Hello world';
  if String1 = '' then
    Show('Empty')
  else
    Show('Not empty');

  String1 := ''; // Or String1.Empty;
  if String1.IsEmpty then
    Show('Empty')
  else
    Show('Not empty');
```

输出的结果如下:

```
Not empty
Empty
```

把字符串作为参数传递

我们刚提到过，如果我们把字符串指派给另一个字符串，实际上的动作只是复制了内存地址，内存内所存放的字符串内容并不会被复制。然而如果我们的源码在字符串变量被指派后改变了字符串变量的内容，原始字符串的内容会被复制到另一个内存空间(也只有在这个时间点)，然后进行修改。

另一个会发生类似复制程序的时间点，是在把字符串作为参数传递给一个函式或程序的时候。默认的作法，系统会制作一个新的内存地址来存放作为参

数的字符串，这样一来，万一在子程序中改动了作为参数的字符串，原始的字符串就不会受到影响。假如我们要使用另一种规则，就是子程序当中的改动要直接套用在原始字符串变量的话，我们就得用传址呼叫的方式，只需在参数前面加上一个 var 关键词即可(同样的作法也可以套用在大多数其他简单的数据型别上面)。

但是万一我们不想改动这个作为参数被传递进子程序的字符串呢？在这个情形下，我们可以在参数前面直接加上个 const 描述字即可。加上了 const 描述字之后，编译器就不会让我们在子程序里面改动该字符串变量的内容了，但是同时也会优化参数传递的动作。事实上，const 字符串不会要求子程序进入时增加该字符串的参考计数，也不会要求离开时减少该字符串的参考计数，因为程序知道这个字符串是无法被修改的。

由于字符串管理程序的动作非常快，执行它们数千或数百万次，也只会对我们的程序增加很轻微的负担。这也是为何我们建议在没有改动参数需求时，直接在参数前加上 const 关键词的原因了。(这作法可能会有些潜在的问题，我们在底下的笔记来讨论一下)。

用程序的语汇来说，以下的三个程序宣告中，字符串参数的传递都是使用不同方法来达成的：

```
procedure ShowMsg1 (Str: string);
procedure ShowMsg2 (var Str: string);
procedure ShowMsg3 (const Str: string);
```

笔记

近几年，有个强烈的潮流，是要求除了在函式或者方法当中会异动到字符串内容的情形外，把所有字符串参数都以 const 方式宣告。但是，有个重大警讯。对于常数字符串参数，编译器会单纯的只取该字符串的内存地址，也只把它当成一个指向内存地址的指针，并不会『管理』该字符串(不会进行参考计数等管理)。编译器也只会检查该函式里面的源码确保没有对该字符串参数进行异动。然而编译器并不会管该段内存内容发生了什么事情。

对于字符串的异动可能影响内存的编排与位置，这些都是用一般方式传递字符串参数的时候会自动进行的管理(在一个字符串需要多重参考时，会自动触发写入时自动复制的程序)，但如果用 const 方法传递字符串参数，这些检查跟处理就会被忽略。换句话说，对于用 const 传入参数原始字符串的变动，可能会导致该字符串发生错误，更严重的可能会发生内存存取错误。

[]的使用，以及字符串中对字符计数的模式

一如我们所知的，我们在使用 Object Pascal 或者其他编程语言的时候，有个很关键的字符串处理方法，就是取用字符串当中的某一个元素(字符)，有时我们会透过方括号，也就是取用数组元素的相同方法。

在 Object Pascal 里，提供了两个明显不同的方式来达成这个功能：

- Chars[]字符串型别助手的处理，可以把字符串当成一个只读的字符数组，是 0 base 的索引值喔。
- 标准的[]字符串处理函式，支持读取与写入，也预设使用传统 Pascal 的以 1 起始的索引规则。这个设定可以透过编译器设定来修改。

我们用以下的笔记来厘清一些这当中的前因后果。把它列为笔记的原因，是让对这些历史没有兴趣的读者可以先跳过，而且如果没有先了解一下过往的作法，对于目前为什么要这样处理，很容易迷惑。

笔记 让我们先回顾一下历史，所谓鉴古而知今。在 Pascal 语言的早期，字符串是被以类似字符数组的方式来处理的。在这个字符数组的第一个元素(可以视之为数组的第 0 个元素)是用来记录字符串当中合法字符的数量的。在那个时期里，C 语言要计算字符串长度时，就得每次都重新计算一次，直到字符串结尾出现 NULL 字符为止。而 Pascal 的程序只需要直接检查第一个 Byte 即可，假设第 0 个 byte 是用来记载字符串的长度，字符串的第一个字符就是从数组的第 1 个元素开始了。

随着时间经过，其他编程语言都随着 C 语言的习惯，把字符串跟数组的第一个元素从索引值 0 开始记录。后来，Object Pascal 也开始让动态数组改为 0 base，渐渐的所有 RTL 跟组件库里面相关数据结构也都改为了 0-base，唯独字符串是一个很特别的例外。而在迈入了行动开发的纪元之后，Object Pascal 语言的设计群们决定提供字符串的 0-base 优先权，让开发人员万一有旧的源码需要移植到行动平台时可以选择，这可以透过编译器控制设定来调整。在 Delphi 10.4 当中，之前的决策被推翻了，这是为了让既有的原始码能更有一致性，不要因为对应布署的平台不同而有所影响。换句话说『单一原始码，多种平台』这个目标的优先性赢过了『更像现代语言』这个目标。

假如我们想要对索引值使用 0-base 跟 1-base 的不同做出一个比较详细的比较，可以思考一下在欧洲跟北美洲对于楼层的算法(我真的不知道世界上其他地区是怎么计算的)。在欧洲，地面楼层称为 0 楼，一楼则是高于地面楼

层的第一层楼。而在北美洲，地面楼层称为 1 楼，高于地面楼层的第一层则算是 2 楼。换句话说，北美洲使用 1 base 的楼层计算法，而欧洲是使用 0 base 的计算法。

而对字符串来说，绝大多数的编程语言都使用 0 base 的楼层算法，不管该语言是在美洲还是欧洲大陆发明的。Delphi 跟大多数从 Pascal 衍生的语言都使用 1 base 的标注方法。

我们再更深入一点来说明，以刚刚我们介绍过的，Char[]使用了 0 base 的索引法，所以如果我们写:

```
var
    String1: string;
begin
    String1 := 'Hello world';
    Show (String1.Chars[1]);
```

则输出的字符会是:

```
e
```

而如果我们直接使用方括号来存取字符串内容，把源码改写如下：

```
Show (String1[1]);
```

这个输出的预设结果是 H。但如果编译器定义了$ZEROBASESTRING 为 on，输出结果就会是 e。目前我们的建议作法(在 10.4 推出之后)，是在大家的源码当中对所有的字符串都使用 1-base，而不要因为过去的源码使用默认的 0-base 做字符串处理而进行调整。

但如果我们想要写段源码，而不受到$ZEROBASESTRING 设定的影响，该怎么做？我们可以把索引值抽象化，例如使用 Low(String)作为该字符串第一个索引值，而使用 High(String)作为最后一个索引值。这个写法不管编译器设定怎么改，都不会受到影响:

```
var
    S: string;
    I: Integer;
 begin
    S := 'Hello world';
    for I := Low (S) to High (S) do
        Show(S[I]);
```

换句话说，字符串毫无例外的拥有着从 Low 到 High 函数所回传的数值之间的所有元素。

笔记 字符串就是字符串，关于 0-base 的字符串概念，我讲的完全是错的。在内存的数据结构并没有不同，所以我们可以我们可以把任何字符串当做参数传递给任何函式，不管是编译器怎么设定，都不会出错。换句话说，如果我们有段程序，在编译器设定为 0-base 字符串时，需要把字符串传递给 1-base 的函式库，系统仍然是可以正确处理这种情形的。

字符串连接

我已经提过，Object Pascal 跟其他编程语言不一样，Object Pascal 完全支持直接进行字符串连接，而且处理速度相对比其他编程语言更快。在这一章里面，我已经示范过一些字符串连接的源码，当中也做了一些速度测试。在后面的章节里，在第 18 章，我会介绍关于 TStringBuilder 这个类别，这个类别遵循了.NET 组成字符串的表示法。因为使用 TStringBuilder 有一些原因，效能并不是最重要的一个(在以下的范例中，我会加以示范)：

```
var
    Str1, Str2: string;
begin
    Str1 := 'Hello,';
    Str2 := ' world';
    Str1 := Str1 + Str2;
```

请留意到我把 Str1 这个变量同时使用在一个指派动作的左边跟右边，把一一些新的内容加在已经存在的字符串后面，而不是指派给另一个全新的变数。当然两种作法都没问题，只是加在一个已存在的字符串的效能会比较好。

这种字符串连接的作法也可以用个循环来达成，就像以下从 LargeString 范例项目中撷取的部分源码：

```
uses
     Diagnostics;
const
    MaxLoop = 2_000_000; // two million
var
    Str1, Str2: string;
    I: Integer;
    T1: TStopwatch;
```

```
begin
    Str1 := 'Marco ';
    Str2 := 'Cantu ';

    T1 := TStopwatch.StartNew;
    for I := 1 to MaxLoop do
        Str1 := Str1 + Str2;
    T1.Stop;
    Show('Length: ' + Str1.Length.ToString);
    Show('Concatenation: ' + T1.ElapsedMilliseconds.ToString);
end;
```

这段源码执行后，我从 Windows 虚拟机跟 Android 装置分别得到以下的输出结果(桌机比较快一点):

```
Length: 12000006   // Windows (in a VM)
Concatenation: 59

Length: 12000006   // Android (Nexus 4)
Concatenation: 991
```

这个范例项目的源码跟 TStringBuilder 类别的作法有些类似。目前我还不想深入这个范例项目的源码(在第 18 章的时候我们再深入讨论)，我们只要先看实际上执行所花的时间，和直接连接的作法所花的时间做个比较

```
Length: 12000006   // (in a VM)
StringBuilder: 79

Length: 12000006   // Android (on device)
StringBuilder: 1057
```

我们可以看到，连结的动作可以说是最快的选项。

字符串助手的处理程序

假如字符串型别是重要的，那么这个型别的助手所提供的处理程序应该不少，这一点应该不会让人太惊讶。假如这些处理程序在大多数的应用程序中都很重要，且很常被使用到，那我就认为应该用个清单好好来介绍一下:

Delphi 传统的全局字符串处理函式跟字符串助手类别的方法有个关键上的差异：传统处理字符串时，会以 1-based 字符串处理，而字符串助手类别则会使用 0-based 的逻辑。

我已经把字符串型别助手的处理程序用逻辑先编组了(这些程序大多都拥有好几个多载版本)，简单的叙述一下它们的功能，大多数的功能名称都能很直觉的看出其功能：

- 复制整个字符串或复制部分字符串，像 Copy, CopyTo, Join 跟 SubString
- 字符串变更程序，像 Insert, Remove, Replace
- 把一些不同型别的数据转成字符串，我们可以用 Parse 或 Format
- 把字符串转成不同型别的数据，我们可以用 ToBoolean, ToInteger, ToSingle, ToDouble, ToExrtended, 如果要把字符串转成字符数组，我们也可以用 ToCharArray
- 把空格符塞到一个字符串里，我们可以用 PadLeft, PadRight, 也可以用 Create 的一个多载版本。相对的，我们也可以把空格符移除，透过 TrimRight, TrimLeft 跟 Trim.
- 字符串的比较跟相符测试(Compare, CompareOrdinal, CompareText, CompareTo 跟 Equals)- 请记得，我们也可以用等号来比较两个字符串是否内容相同。
- 转换大小写用 LowerCase, UpperCase, ToLower 跟 ToUpper 以及 ToUpperInvariant
- 检查字符串内容，我们可以用 Contains, StartsWith, EndWith。要搜寻特定字符是否存在该字符串中，也可以用 IndexOf 来达成(从字符串开头到特定位置之间的搜寻)。相似的函式还有 IndexOfAny(这可以搜寻一个字符数组中的任何一个字符是否存在该字符串中)，LastIndexOf 跟 LastIndexOfAny 可以从字符串结尾开始回头找特定字符最后一次出现在该字符串的位置。最特殊的处理程序，则是 IsDelimiter 跟 LastDelimiter，这是用来检查字符串的结尾是否是目录分隔符，以及最后一次目录分隔符出现的位置。
- 要存取字符串长度，我们可以用 Length 函式，这个函式会回传字符串中内含字符的数量。CountChars 则可以把代理对应(surrogate pair)一并计算，GetHashCode，可以回传该字符串的哈希值(Hash value)。另外还有一些检查的程序，例如 IsEmpty, IsNullOrEmpty 以及 IsNullOrWhiteSpace 都可以用来检查字符串的内容是否为空。
- 字符串的特殊处理程序，像是 Split, 可以帮我们把字符串依照特定分隔符(或分隔字符串)切成许多部分。要移除或加入引号，我们可以用 QuotedString 或者 DeQuoted.

- 最后，要存取单独字符，我们可以用 Char[]，方括号中需要填入数值索引。透过这个处理程序，我们可以读取特定的字符，但不能修改它，这个函式的索引值是 0-base 的，跟 C 语言的特性相同。

事实上，很重要的一点，所有字符串型别助手的处理程序都是依照 RTL 的字符串建立的，也就是说，它引入了 0-base 字符串的概念，字符串的开始元素从 0 开始，字符串的长度记录在 -1 的位置。换句话说，就像我之前提过的，也值得再说一次，所有字符串型别助手的处理程序都使用 0-base 的索引值作为参数与回传值。

笔记 Split 程序是在 Object Pascal RTL 里面较新的成员。之前比较常见的作法，是把字符串加载到 stringList 里面，然后设定一个特定的断行符号，接着就可以一一读取每一行的文字了。Split 处理程序则在效能跟弹性上都有了显著的提升。

假如我们需要对字符串处理程序进行大量的处理，我可以写几个项目来示范这些功能。或者也可以只提到几个相对简单的程序，或者常用的程序：

```
var
  Str1, Str2: string;
  I, NIndex: Integer;
begin
    Str1 := '';
    // create string
    for I := 1 to 10 do
        Str1 := Str1 + 'Object ';
    Str2:= string.Copy (Str1);
    Str1 := Str2 + 'Pascal ' + Str2.Substring (10, 30);
    Show(Str1);
```

请留意我使用 Copy 函式的方法，我是直接建立了一份独立的字符串来储存相同的内容，而不是只做了指派替身。即使在这个特定的范例中，也没有任何不同。最后一个处理程序时，SubString 的函数调用是用来解开字符串的内容，结果文字如下：

```
Object Object Object Object Object Object Object Object Object Object
Pascal ect Object Object Object Objec
```

在初始化完成之后，第一个按钮会处理搜寻部分字符串，并且重复这样的搜寻，透过不同的起始搜寻点，来计算目标字符串出现的次数(在范例中，是搜寻单一字符)：

```
// Find substring
Show('Pascal at: ' +
     Str1.IndexOf ('Pascal').ToString);
// Count occurrences
I := -1;
NCount := 0;
repeat
    I := Str1.IndexOf('O', I + 1);   // search from next element
    if I >= 0 then
      Inc (NCount);   // found one
until I < 0;
Show('O found: ' +
     NCount.ToString + ' times');
```

这个 repeat 循环不是最简单的一个：它从一个负值开始，之后找到吻合的字符后，就从该吻合的字符之后再继续进行搜寻，所以也会记录下吻合的字符出现的次数，如果没有任何吻合的字符，则回传-1。上述这段源码的输出值为：

```
Pascal at: 70
O found: 14 times
```

第二个按钮的功能则是扮演搜寻与取代的功能，可以搜寻特定字符串，并把搜寻到的字符串更改为指定字符串。在搜寻的功能中，会建立一个新字符串，复制初始与最后的部份，并加入一些新的文字在中间。取代的功能则是使用了 Replace 函式来处理出现多次的字符串，透过适当的参数传递即可(rfReplaceAll)，以下是范例源码:

```
// Single replace
NIndex := Str1.IndexOf ('Pascal');
Str1 := Str1.Substring(0, NIndex) + 'Object' +
            Str1.Substring(NIndex + ('Pascal').Length);
Show (Str1);


// Multi-replace
Str1 := Str1.Replace('O', 'o', [rfReplaceAll]);
Show (Str1);
```

由于输出的字符串相当长，而且不易阅读，我只列出当中的部分字符串：

```
...Object Pascal ect Object Object...
...Object Object ect Object Object...
...object object ect object object...
```

重申一次，这只是在众多丰富的字符串处理功能中，很小一部分的范例而已，在字符串型别助手当中还有许多功能等着我们去发现呢。

更多字符串的运行时间函式库(RTL)

决定沿用其他语言常用的处理程序名称来制作字符串型别助手的一个效应，就是这些名称会跟传统 Object Pascal 的处理程序名称不同(这些处理程序直到今天还是存在于全局函式当中)。以下的列表就可以看出一些名称不同的函式了：

Global	String type helper
Pos	IndexOf
IntToStr	Parse
StrToInt	ToInteger
CharsOf	Create
StringReplace	Replace

笔记 要记住，这些全局函式跟字符型别助手的处理程序之间的一个极大不同：第一组的函式使用的是 1-base 的索引方式来处理字符串，第二组的函式使用的都是 0-base 的。

只有在 RTL 当中最常用的字符串函式被改了名字，其他比较不常用的函式则还是沿用了旧名称，例如 UpperCase 或 QuotedString。在 System.SysUtils 单元里面还有许多这样的字符函式，这些函式很多并没有被并入到字符串型别助手当中。

这些函式中，值得一提的包含有:

- ResemblesText，这个函式实现了 Soundex 算法(这个算法可以找出不同拼法，但读音相同的英文字汇)
- DupeString，这个函式会回传特定函式被要求的份数。
- IfThen，这个函式会在判断式成立时回传第一个字符串，不成立时回传第二个字符串(我在本章前面的范例中有使用到这个函式)
- ReserveString，这个函式会把参数字符串顺序全部颠倒之后回传。

格式化字符串

透过加号来链接字符串、使用一些转换函式之后，我们已经可以产生出比较复杂的字符串了。但还有一个比较不同的、更强大的方法来把数字、金额以及其他数据进行格式化后转换成字符串的方法。复杂的字符串格式化可以透过呼叫 Format 函式来达成，这个函式很传统，但很常用，不只在 Object Pascal 里面有，在其他编程语言里面也有提供。

历史 『输出格式化字符串』这个功能的函式家族，或者称为 printf 的函式，在早期的编程语言里面都有提供，像是 FORTRAN 66, COBOL, 以及 ALGOL 68。这些特定的格式化字符串符号与结构到今日仍在使用(Object Pascal 里面也有使用到)，这些结构跟函式都很接近 C 语言的 printf 函式，需要对相关主题更深入研究的话，建议您可以参考 en.wikipedia.org/wiki/Printf_format_string 这个网址。

Format 函式要求一个包含基本输出的文字，以及特定参数格式符号(会以%符号作为开头字符，再加上各种代号表示不同型别)的字符串作为参数，以及一个数据数组，用来对应前面指定参数格式字符串的各个字段。例如，要把两个数字格式化输出到一个字符串中，我们可以写成：

```
Format ('First %d, Second %d', [N1, N2]);
```

上面的 N1 跟 N2 是两个整数数值。第一个参数格式符号(%d)会被 N1 的内容取代，第二个参数格式符号则会被 N2 的内容取代，如果有多个参数，则顺序依此类推。如果数组当中的变量或数据型别跟参数格式符号的顺序对不起来，就会发生运行时错误。没办法在编译阶段进行型别检查，是我们在使用 Format 函式的时候的最大风险。相似的是，如果我们在数组里面指定的数值个数跟参数不同，也会发生错误。

Format 函式使用了开放数组参数(我们在第五章介绍过，该参数可以有不特定的数量与型别作为参数内容)。除了使用%d, 我们也可以使用许多型别的参数格式符号，我们稍后会提供一个列表来说明各种型别的代号。这些参数格式符号都会为其代表的型别提供预设的输出值。然而我们也可以使用更多的格式表达方法来替代预设的输出值。例如，宽度的格式表达法可以决定使用固定长度的字符，好让输出的数字格式可以有固定的位数，例如：

```
Format ('%8d', [N1]);
```

这个范例会把 N1 的数值转换成一个 8 个字符的字符串，字符串靠右对齐(使用减号则可以让字符串靠左)如果不足 8 个字符，则会用空格符来补足，以下就是各个型别的参数格式符号列表：

d (十进制整数)　此符号对应以十进制表示法把整数转换成字符串

x (十六进制数)　此符号对应以十六进制表示法把整数转换成字符串

p (指针)　此符号对应以十六进制表示法把指针转换成字符串

s (字符串)　此符号对应字符串、字符或者 PChar(指向字符数组的指针)，该变量的内容会被复制到输出的字符串去。

e (指数)　此符号对应浮点数，会以科学计数表示法把浮点数显示为字符串。

f (浮点数)　此符号对应浮点数，会把浮点数数值以字符串显示。

g (通用)　此符号会以最短的十进制表示法把浮点数或指数以字符串显示。

n (数字)　此符号对应浮点数，会以十进制显示浮点数数值，但会每三位数就自动加上千位符号。

m (货币)　此符号对应货币数值，会以十进制显示浮点数，但以货币数值加以显示，因此该转换会依照地区货币的设定进行。

要仔细观察这些转换的最好方法，莫过于我们直接用 format 函式来实验了。为了让这个实验容易一点，我们提供了 FormatString 范例项目，在这里面开发人员可以自行输入格式化字符串来显示一些预先定义好的整数变量。

这个程序的窗体在按钮之上有一个输入文字框(TEdit)，初始化之后，内容是一个预先定义的格式字符串(‘%d - %d - %d’)。第一个按钮让我们显示较为复杂的范例格式化字符串(源码会把格式化字符串’Value %d, Align %4d, Fill %4.4’指派到文字框里面)。第二个按钮会把这个字符串应用在预先定义的变量中，透过以下的源码：

```
var
    StrFmt: string;
    N1, N2, N3: Integer;
begin
    StrFmt := Edit1.Text;
    N1 := 8;
    N2 := 16;
    N3 := 256;
```

```
    Show (Format ('Format string: %s', [StrFmt]));
    Show (Format ('Input data: [%d, %d, %d]', [N1, N2, N3]));
    Show (Format ('Output: %s', [Format (StrFmt, [N1, N2, N3])]));
    Show (''); // blank line
end;
```

如果我们先以初始的格式化字符串进行显示，然后再以复杂点的格式化字符串进行显示(依序点击第二个按钮，第一个按钮，然后再点第二个按钮)，我们应该就会看到如下的输出字符串：

```
Format string: %d - %d – %d
Input data: [8, 16, 256]
Output: 8 - 16 - 256

Format string: Value %d, Align %4d, Fill %4.4d
Input data: [8, 16, 256]
Output: Value 8, Align 16, Fill 0256
```

在这个程序后隐含的意义，是让我们可以直接输入格式化字符串，然后试着套用这些字符串来显示变量内容，我们可以对照看更多可以使用的格式化选项。

字符串的内部结构

我们可以使用字符串，而不需要了解字符串的内部作法，但仔细来看一下这个型别内部的实际数据结构也是挺有趣的。在早期的 Pascal 语言里面，字符串的长度限制是 255 个字符，并会使用第 0 个位来储存字符串的长度。从早期的发展到现在已经经过许久，但把额外信息储存为字符串信息的一部分，仍旧被保留下来，成为 Object Pascal 语言的特色(C 语言阵营的其他编程语言则还是使用字符串结尾符号的概念来实作字符串)。

笔记 传统 Pascal 字符串型别的型别名称是 ShortString，它是使用单一位字符或者 AnsiChar 为元素，长度限制为 255 个字符。ShortString 型别在桌面应用程序版的编译器仍然可以使用，但在行动版编译器已经不能使用了。我们仍然可以透过类似的数据结构，带领位的动态数组来达成这个功能，或者 Tbytes，或者 Byte 的静态数组也可以。

一如我们已经介绍过的，字符串变量只是个指向从 heap 配置而来的记忆空间的指针。事实上，储存在字符串变量的数据，并不是从这个数据的最前头

开始记录的，第一个字符是被储存在第 1 个位置，而不是第 0 个位置。字符串的相关信息则是储存在索引负值的内存空间当中。在内存当中对字符串型别的表示法如下:

-12	-10	-8	-4	字符串指针所指到的地址
Code page	元素大小	参考计数	字符串长度	字符串的第一个字符

第一个元素(从字符串指针开头的地方往回算起)是用来记录字符串长度的地方。第二个元素则是用来记录参考计数，接下来的字段(只用在桌面应用程序编译器)是每个元素占用几个位(目前不是 1 就是 2)，最后则是以 ANSI 为基础的字符串型别用来记录 code page 的字段。

很令人惊讶，除了我们熟知的 Length 之外，居然还有函式可以直接存取这些底层的字符串属性字段。

```
function StringElementSize(const S: string): Word;
function StringCodePage(const S: string): Word;
function StringRefCount(const S: string): Longint;
```

就像范例程序中所述，我们可以建立一个字符串，并且询问一些关于它的信息，我们用 StringMetaTest 这个范例来做示范:

```
var
    Str1: string;
begin
    Str1 := 'F' + string.Create ('o', 2);

    Show ('SizeOf: ' + SizeOf (Str1).ToString);
    Show ('Length: ' + Str1.Length.ToString);
    Show ('StringElementSize: ' +
          StringElementSize (Str1).ToString);
    Show ('StringRefCount: ' +
          StringRefCount (Str1).ToString);
    Show ('StringCodePage: ' +
          StringCodePage (Str1).ToString);

    if StringCodePage (Str1) = DefaultUnicodeCodePage then
       Show ('Is Unicode');
    Show ('Size in bytes: ' +
```

```
    (Length (Str1) * StringElementSize (Str1)).ToString);
Show ('ByteLength: ' +
        ByteLength (Str1).ToString);
```

笔记 范例程序中以动态的方式来建立出'Foo'这个字符串，而没有直接指派一个字符串常数是有特殊原因的，因为字符串常数的参考计数是没有作用的(或者设为-1)。在这个范例中，我想要详细的显示出参考计数的内容，所以使用了动态的方法来建立字符串。

这个范例的执行结果显示如下:

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: 1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

Code page 回传值是 Unicode 字符串，所以是 1200，这个数字会存在一个名为 DefaultUnicodeCodePage 的全局变量里面。在上面的源码中(以及执行结果)，我们可以很清楚的留意到两个字符串变量尺寸的不同(并不都是 4)，逻辑上的长度，或者实际上使用内存空间的长度都是如此。

这个结果可以用每个字符的长度(占用多少位)乘上整个字符串的长度得出，或者直接呼叫 ByteLength 函式也可以。ByteLength 这个函式在旧版的桌面应用程序编译器里面并不支持。

观察在内存里面的字符串

仔细观察字符串的 metadata 对于了解字符串的内存管理是很有帮助的，尤其是观察参考计数的变化。为了这个要求，我在 StringMetaTest 范例项目里面加了一些其他的源码。

这个程序有两个全局字符串：MyStr1 跟 MyStr2。源码仍使用动态方式为前面提到这两个变量产生出字符串内容(原因在稍早的"笔记"中已说明过)，然后把第二个变量指派给第一个：

```
MyStr1 := string.Create(['H', 'e', 'l', 'l', 'o']);
```

```
MyStr2 := MyStr1;
```

除了处理字符串之外，这个程序也显示字符串的内部状态，透过以下的 StringStatus 函式:

```
function StringStatus (const Str: string): string;
begin
    Result := 'Addr: ' + IntToStr (Integer (Str)) +
        ', Len: ' +  IntToStr (Length (Str)) +
        ', Ref: ' +  IntToStr (PInteger (Integer (Str) - 8)^) +
        ', Val: ' + Str;
end;
```

StringStatus 这个函式中很重要的一点，是它的参数是使用 const 描述来传递字符串的。如果我们不使用 const 来描述参数的话，作为参数的字符串会被复制一份、产生一些额外的副作用，例如每次把字符串传递给这个函式，参考计数就会被加一。比较一下，以 var 跟 const 传递字符串参数，则不会让字符串的参考计数有变动。在这个范例中我已经使用了 const 参数，在这个函式里并不会更动到字符串的内容。

要取得字符串的内存地址(对于搞清楚这个字符串到底是哪一份很有用，而且可以知道什么时候两个不同的字符串变量居然是指向同一个内存区块的)，我也把字符串型别做了一个强制型别转换，转成整数型别。字符串型别是以参考计数方式来实作的，字符串变量实际上是指针：实际内容是记录在字符串变量所指向的内存空间里面，而不是直接存在字符串变量中。以下的源码用来测试字符串变量的内部变化:

```
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
MyStr1 [1] := 'a';
Show ('Change 2nd char');
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
```

一开始，我们需要让两个字符串拥有同样的内容，同样的内存空间，我们可以看到计数参考的内容是 2。

```
MyStr1 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
MyStr2 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
```

接着程序改变了其中一个字符串的内容(改哪一个都没关系)，内容被更改的那个字符串变量所指向的内存位置就改变了。这就是"写入时才复制"的效果，这是第二部分的输出结果:

```
Change 2nd char
MyStr1 - Addr: 51848300, Len: 5, Ref: 1, Val: Hallo
MyStr2 - Addr: 51837036, Len: 5, Ref: 1, Val: Hello
```

您可以自由修改这个范例，并使用 StringStatus 这个函式来观察字符串的内容，并可以针对许多种不同情形下的长字符串处理方式来观察，例如对同一个字符串进行多重参考，当字符串被当成参数传递时、指派到局部变量的时候，这些情形都很值得留意。

字符串与编码

我们已经看过，在 Object Pascal 里面，字符串型别是对应到 UTF16 的格式，每个元素以 2 个 Byte 显示，并以代理对应(surrogate pair)来管理在基础多语系平面(Basic Multilingual Plan, BMP)之外的字码。然而在许多情形中，当我们需要存盘、读档、透过 socket 联机传送，或者从网络联机接收以不同表示法(例如 ANSI 或 UTF8)传送的文字数据时。

要在不同编码的档案跟内存空间里面进行格式转换(或者编码转换)时，Object Pascal RTL (运行时间函式库)提供了一个好用的类别 TEncoding 以及一些衍生类别，这些类别都定义在 System.SysUtils 单元里面。

笔记 在 Object Pascal RTL 里面还提供了一些其他好用的类别，让我们可以用来读写多种编码的文字数据。例如 TStreamReader 跟 TStreamWriter 类别就可以读写各种不同编码的文字数据，这些类别我们会在第 18 章里面介绍。

虽然我们还没介绍过类别跟继承，但这一系列的编码类别很简单，在全局对象中也已经都提供了每一种编码的对象，我们可以在需要时取用。换句话说，每个编码类别的对象都已经可以透过 TEncoding 类别来取用，就好像是类别的属性一样：

```
type
  TEncoding = class
    ...
  public
    class property ASCII: TEncoding read GetASCII;
    class property BigEndianUnicode: TEncoding read GetBigEndianUnicode;
```

```
    class property Default: TEncoding read GetDefault;
    class property Unicode: TEncoding read GetUnicode;
    class property UTF7: TEncoding read GetUTF7;
    class property UTF8: TEncoding read GetUTF8;
```

笔记 Unicode 的编码是基于 TUnicodeEncoding 类别，这个类别也使用 UTF-16 LE (Little Endian) 这个格式(字符串型别也是使用同样格式)。而 BigEndianUnicode 则用了比较少被使用到的 Big Endian 表示法。如果您对两种 Endian 表示法(这是用来表示数字数据以位排列时的顺序)不熟悉的话，在此赘述一次：Little Endian 是把最高位放最前面，也就是俗称的 MSB(Most Significate Byte)放最前面，而 Big Endian 则是把最高位放最后，也就是把 MSB 放最后面。更多的细节，建议您可以参考 en.wikipedia.org/wiki/Endianness

在目前这个章节要仔细观察这些类别还有点太难，我们先来看一些实际案例。Tencoding 类别提供了一些方法可以从位数组来读写 Unicode 字符串，会自动进行适当的编码/译码处理。

接下来我们用 TEncoding 系列类别来示范 UTF 格式转换，但保持范例源码简单、不让焦点被转移到文件系统去。在 EncodingsTest 范例项目我在内存里面用一些特定的数据建立了一个 UTF-8 的字符串，然后透过呼叫一个函式来把它转换成 UTF-16：

```
var
    Utf8string: TBytes;
    Utf16string: string;
begin
    // Process UTF-8 data
    SetLength (Utf8string, 3);
    Utf8string[0] := Ord ('a'); // Single byte ANSI char < 128
    Utf8string[1] := $c9; // Double byte, reversed Latin a
    Utf8string[2] := $90;

    Utf16string := TEncoding.UTF8.GetString(Utf8string);
    Show ('Unicode: ' + Utf16string);
```

输出结果会是:

```
Unicode: aɐ
```

为了对不同表示法的转换有更深入的了解，我加入了这些源码:

```
Show ('Utf8 bytes:');
for AByte in Utf8String do
    Show (AByte.ToString);

Show ('Utf16 bytes:');
UniBytes := TEncoding.Unicode.GetBytes (Utf16string);
for AByte in UniBytes do
    Show (AByte.ToString);
```

这段源码把内存做了倾印，以十进制表示法印出，把两种格式的字符串都倾印了，UTF8(以 1 个或 2 个 Byte 来表示一个字码)以及 UTF-16(所有字码都是 2 bytes):

```
Utf8 bytes:
 97
 201
 144
Utf16 bytes:
 97
 0
 80
 2
```

请注意，在 UTF-8 表示法中，字符直接转成 byte 的作法，只能对 ANSI-7 的字符有用，ANSI-7 的字符最大值只到 127。对更大的 ANSI 字符就没有所谓的直接对应了，我们必须要做适当的字符编码转换(这样的转换在多位的 UTF-8 元素进行时也可能出错)。所以以下的程序就跑出了错误的结果:

```
// Error: cannot use char > 128
Utf8string[0] := Ord ('à');
Utf16string := TEncoding.UTF8.GetString(Utf8string);
Show ('Wrong high ANSI: ' + Utf16string);
// Try different conversion
Utf16string := TEncoding.ANSI.GetString(Utf8string);
Show ('Wrong double byte: ' + Utf16string);
```

```
// Output
Wrong high ANSI:
Wrong double byte: àÉ
```

编码类别们让我们可以进行双向的编译码处理，所以我们接下来把 Unicode 转换为 UTF-8，然后做一些 UTF-8 的字符串处理(其中有些处理要很小心，因为这个格式的长度是变动的)，然后再转换回 UTF-16:

```
var
    Utf8string: TBytes;
    Utf16string: string;
    I: Integer;
begin
    Utf16string := 'This is my nice string with à and Æ';
    Show ('Initial: ' + Utf16string);

    Utf8string := TEncoding.UTF8.GetBytes(Utf16string);
    for I := 0 to High(Utf8string) do
        if Utf8string[I] = Ord('i') then
            Utf8string[I] := Ord('I');
    Utf16string := TEncoding.UTF8.GetString(Utf8string);
    Show ('Final: ' + Utf16string);
```

输出结果如下:

```
Initial: This is my nice string with à and Æ
Final: ThIs Is my nIce strIng wIth à and Æ
```

其他的字符串型别

因为字符串型别的内部单元格式并不能满足最常用且最大量被使用的字符串编码方法，Object Pascal 桌面应用程序编译器提供了许多种不同的字符串型别。其中部分型别也可以被使用在行动版应用程序，但建议一定要先做好适当的转码，或是使用 TBytes 来处理 1 byte 表示法的字符串。

过去惯用 Object Pascal 的程序人员可能已经积累了许多前 Unicode 时期的类别的源码(或者自己直接写程序来管理 UTF-8)，现在的编程语言几乎都是直

接完整支持 Unicode 的。另外，有一些型别，像是 UTF8String，在语言中都有支持，在 RTL 的支持已经有限制了。再次重申，我们可以用位数组来表示简单的数据型别、改写既存的源码来处理这些字符串，但还是建议尽快移转到标准的 Unicode 字符串型别吧。

笔记 一直以来，对于 Object Pascal 行动版编译器没有支持部分旧版当中的原生型别，像是 AnsiString 跟 UTF8String 都不断有很多讨论跟批评。在 Delphi 10.1 Berlin 中，已经对 UTF8String 跟低阶的 RawByteString 重新支持了(Delphi 10.0 Seattle 当中是把这些字符串型别都移除掉的)，到了 Delphi 10.4 之后，则是在桌面操作系统与行动装置系统上都支持了这些字符串型别。平心而论，几乎没有其他编程语言支持一种以上的原生字符串型别的。多种字符串型别太过复杂，很难以驾驭，而且可能会有一些我们不想遇到的副作用(像是自动型别转换的函式，会让程序的执行速度变慢)，而且会花很多时间来维护这些字符串管理跟处理的函式。所以，除非是很特殊的案例，强力推荐聚焦使用标准的字符串型别，或者 UnicodeString。

UCS4String 型别

UCS4String 是一个有趣但很少被用到的字符串型别，所有版本的编译器都有支持它。这是直接以 UTF32 表示字符串的型别，而且已经不是以 UTF32Char 字符数组来实作的，也不是用 4 bytes 字符数组来实作的。之前的篇幅中已经提过为什么这个型别会被保留下来，是因为它可以直接对应到所有 Unicode 字符。而明显的代价就是要花上近乎两倍的内存(跟 UTF-16 字符串相比，UTF-16 又比 ANSI 字符串的内存用量又多上一倍)来储存相同内容的字符串。

即使这个数据型别在特定的情形下会被使用，但它不适用于一般的情况。且这个型别没有支持”写入时才复制”的功能，也没有任何系统函式或程序是可以直接拿来处理它的。

笔记 UCS4String 型别可以保证 UTF32Char 与每一个 Unicode 字码直接对应，但不能保证每个 UTF32Char 的显示文字都是同一个图像。

旧版的字符串型别

我们前面已经提过，Object Pascal 编译器对旧版的传统字符串型别仍然提供支持(这些字符串型别从 Delphi 10.4 开始，在所有平台都提供支持)，。这些旧版字符串型别包含有:

- ShortString 型别，对应到传统的 Pascal 语言的字符串型别，这个型别的长度限制在 255 个字符，每个字符都是 ANSIChar 型别。
- ANSIString 型别，对应可变长度的字符串，这些字符串会是随机配置的，支持参考计数，并且也支持”写入时才复制”的技术，字符串的长度几乎没有限制(可以存到 20 亿个字符喔！)当然，这个字符串的字符也都是 ANSIChar 型别。行动版编译器也支持，即使 ANSI 显示是 Windows 专有，而部分特殊字符可能在特定平台上有不同的处理方式。
- WideString 型别：跟 2 byte 的 Unicode 字符串很相似，储存的内容是 Char 型别，但跟标准的字符串型别不同的是，WideString 并不支持”写入时才复制”的技术，所以在内存配置的效能上没有标准字符串型别来的好。对于为什么把这个型别加进 Object Pascal 好奇吗？原因是要兼容于微软的 COM 架构。
- UTF8String 型别：这个字符串型别奠基于可变字符长度的 UTF-8 格式，一如我们前面介绍的，在 RTL 里面对于这个型别的支持已经不多。
- RawByteString 型别：这是一个没有 code page 设定值的字符数组，这个型别也不会让系统进行任何字符的编码转换。(所以逻辑上这是重组了 TBytes 结构，但提供了一些数组没有提供的字符串处理方式)。
- 字符串结构机制允许我们定义一个与特定 ISO Code page 相关的字符串变量，但这是前 Unicode 时期的遗产。

最后再强调一次，以上这些字符串型别可以用在桌面版的编译器，但建议只在兼容旧版源码的时候使用。我们的目标是尽可能随时都使用 Unicode，TEncoding 以及其他现代字符串管理的技术。

第二部: Object Pascal 中的 OOP

许多现代编程语言都支持了某种程度上的面向对象程序设计(Object-oriented programming, OOP)的范例。这些语言当中大都会用到以下三个核心概念中的其中一个：

- 类别，具备公开接口与私有数据结构的数据型别，实作出封装概念，以类别制作出来的实体，通常称之为对象。
- 类别可以被延伸或继承，这是能够扩展数据型别功能，而不用动到原来源码的能力。
- 多型或者延迟绑定，这是让单一接口可以参考到不同类别对象的能力，并且仍然让对象能够以原来定义的方法运作。

笔记 其他编程语言，像是 IO, JavaScript, Lua 和 Rebol 使用了以原形为基础的面向对象概念，这个概念类似于类别，但没有继承的功能，动态型别可以用来实作出多型，但是是完全不同的作法。

在不用懂得太多面向对象程序的情形下，我们仍然可以用 Object Pascal 来撰写程序。当我们建立一个新的窗体、加入一个组件、处理一个事件的时候，IDE 几乎帮我们把该做的相关宣告跟源码都自动处理好了。但多了解一点编程语言，以及它是怎么实作出一些功能的，会帮助我们更精确的了解系统的运作，也让我们能完整掌握这个语言的强大功能。

对面向对象程序进行理解，也能帮助我们在程序中建立复杂的架构，甚至整个函式库，或者把组件依照我们的需求做功能的延伸，再整合到 IDE 里面来。

本书的第二部分，是要聚焦在面向对象程序设计(OOP)的核心技术。本书的这个部分是要介绍 OOP 的基础概念，以及 Object Pascal 是怎么实作出这些概念的，并把它跟类似的 OOP 编程语言做一些比较。

第二部的章节涵盖了：

第七章：物件
第八章：继承
第九章：例外处理
第十章：属性与事件
第十一章：接口
第十二章：操纵类别
第十三章：对象与内存

07:物件

即使我们对 OOP 的了解不深，这一章会介绍每一个关键概念。如果您对 OOP 已经能够熟极而流，您也可以从本章的介绍中快速的聚焦在 Object Pascal 是如何制作出这些您已经熟知的功能，并和其他面向对象编程语言做个对照或比较。

Object Pascal 对 OOP 的支持跟 C#与 Java 有很多的相似之处，同时还加入了一些 C++跟其他静态、强型别语言的特性。其他动态语言希望能够提供一些不同的 OOP 直译程序，一如它们在处理型别系统那样有弹性。

笔记

C#跟 Object Pascal 许多的概念极度相似，是因为这两个编程语言的设计者是同一个人: Anders Hejlsberg. 他是 Turbo Pascal 编译器、第一版 Delphi Object Pascal 的原始作者，后来他转职到微软，设计了 C#(近期更设计了从 JavaScript 延伸而出的 TypeScript)。您可以从附录 A 当中到更多关于 Object Pascal 语言的历史。

介绍类别(Class)与对象(Object)

*类别(Class)*跟*对象(Object)*这两个名词在 Object Pascal 跟其他 OOP 编程语言中都很常出现。然而也就是因为这两个名词太常被误用了，所以我们在一开始，要非常清楚的把它们的定义给厘清，子曰：必也正名乎：

- 类别，是用户定义的数据型别。类别当中包含状态以及定义了一些动作。用另外一个说法来描述，类别有一些内部数据，以及一些方法，以程序或函式的形式呈现。类别通常描述一些类似对象的特征跟行为，即使有些特别用途的类别所指的就是一个特别的对象。
- 对象则是类别的实体，例如是一个以特定类别作为数据型别而定义的变量。对象是实际存在的个体，当程序执行时，对象会使用一些内存空间来做内部储存之用。
- 类别与对象之间的关系，就跟任何一种数据型别跟变量之间的关系，只是在这个案例中，变量有特别的称呼(实体)而已。

译者乱入：讲述程序设计这么多年，上面的说法总觉得学生很不容易理解，所以我后来用另一个说法来说明，学生们就容易理解的多。类别，我们可以想象成『人类』，人类有许多定义、特征。而对象就是一个个特定的人，例如读者你就是一个『人类』这个类别的实体，译者我也是，作者也是。

历史

OOP 的这些术语是从早期的一些开始实作 OOP 概念的编程语言而来的，例如 Smalltalk。然而最原始的术语后来在演进的过程中被慢慢以程序化语言的术语给取代掉。所以像是类别、对象这些名词虽然还是很常用，我们也很常听到像呼叫某个方法，不像以往会用传递一个讯息给接收者(当然也是个对象)这种说法。完整的把 OOP 的术语列出来，并看一下它的演进是很有趣的，可惜本书的篇幅不够。

定义一个类别

在 Object Pascal 里面，我们可以用以下的语法来定义一个新的类别数据型别(TDate)，来定义一些内部数据字段(Month, Day, Year)跟一些方法(SetValue, LeapYear)：

```
type
    TDate = class
        FMonth, FDay, FYear: Integer;
        procedure SetValue (M, D, Y: Integer);
        function LeapYear: Boolean;
    end;
```

笔记

我们已经以记录来介绍过很类似的结构，记录在定义上跟类别非常相似。但在内存管理和其他领域则很不相同，在本章稍后的篇幅会进行介绍。但在历史意义上，Object Pascal 的这个语法是首先为了类别制作，稍后才为记录移植过来的。

在 Object Pascal 里面，对类别的约定是使用 T 这个字母作为所有类别的第一个字母，像是定义任何其他型别(事实上，T 是当做 Type 这个字的缩写)。这个约定，对编译器来说，T 跟其他字母都一样，但只是让我们的源码能够更容易被其他程序人员了解而已。

跟其他编程语言不同，在 Object Pascal 定义类别时，并不会同时把实作的源码写在一起。只会包含类别的定义(或者说是宣告)，这使得类别的源码更为精简，也更容易了解。

提示 我们已经以记录来介绍过很类似的结构，记录在定义上跟类别非常相似。但在内存管理和其他领域则很不相同，在本章稍后的篇幅会进行介绍。但在历史意义上，Object Pascal 的这个语法是首先为类别制作，稍后才为记录移植回来的。

也请记得除了撰写类别的定义(包含字段与方法)，我们还可以只写个宣告，也就是只写类别的名称:

```
type
    TMyDate = class;
```

这种宣告的用意，是万一我们写了两个类别，而这两个类别之间相互使用时，其中之一就需要先进行预先宣告。因为在 Object Pascal 里面，我们不能使用还没定义的代号，所以要参考到还没定义的类别时，就需要先进行宣告。以下的程序片段只是要介绍语法，它并没有任何实质作用喔：

```
type
  THusband = class;
  TWife = class
    FHusband: THusband;
  end;
  THusband = class
    FWife: TWife;
  end;
```

在实际的源码中，我们得自己处理类似的相互参考情形，因此这个语法很重要，请牢记。另外，请记得就像宣告方法一样，在单元文件里面宣告的类别，必须在同一个单元档案里面完整的定义宣告的所有方法。

在其他 OOP 语言的类别

做个比较，我们用 C#跟 Java 写一个完全相同的 TDate 类别(在这么简单的情形下也可能完全相同)使用一个更适合的命名规则以及对应的方法源码:

```
// C# and Java language
class Date
{
  int   month;
  int   day;
  int   year;
```

```
    void setValue (int m, int d, int y)
    {
      // code
    }
    bool leapYear()
    {
      // code
    }
}
```

在 Java 跟 C#里面，方法的源码会随着类别的定义一起写好，而在 Object Pascal 里面则是把类别的宣告跟实作部分分开来撰写的，但必须在同一个单元文件里面完成。换句话说，在 Object Pascal 里面，类别永远要在单一一个单元文件里面完成定义(当然，一个单元文件里面可以定义很多个类别)。相对的，C++的方法也是分开来撰写的，跟 Object Pascal 有一点点类似，但 C++包含类别定义的头文件案，则不用非跟宣告了方法的源码放在一起。

对应的 C++类别看起来会是这个样子：

```
// C++ language
class Date {
        int     month;
        int     day;
        int     year;

      void setValue (int m, int d, int y);
        BOOL leapYear();
      }
```

类别方法

跟记录类似，当我们定义好方法的时候，我们必须判别该方法是属于哪个类别的一部分(在这个范例中，是 TDate 类别)，在使用这个方法时，可以先输入该类别的名称，加上句点，就可以加以呼叫，用以下的源码作为范例：

```
procedure TDate.SetValue(M D, Y: Integer);
begin
    FMonth := M;
```

```
    FDay := D;
    FYear := Y;
end;

function TDate.LeapYear: Boolean;
begin
    // Call IsLeapYear in SysUtils.pas
    Result := IsLeapYear (FYear);
end;
```

大多数其他的 OOP 语言把方法定义成函式，Object Pascal 不同的是可以让方法定义成函式或者程序，只取决在是否需要回传值。这跟 C++的情况就不同了，C++把方法定义跟实作分开:

```
// C++ method
void Date::setValue(int m, int d, int y) {
        month = m;
        day = d;
        year = y;
};
```

建立对象

在跟其他热门的编程语言比较过后，我们回头来看 Object Pascal 当中要如何使用类别。

一旦类别定义好了，我们就可以以这个类别作为数据型别来建立对象，就像以下的范例(是从 Dates1 范例项目中取出的源码):

```
var
   ADay: TDate;
begin
 // Create
 ADay := TDate.Create;
 // Use
 ADay.SetValue (1, 1, 2016);
 if ADay.LeapYear then
   Show ('Leap year: ' + IntToStr (ADay.Year));
```

表示法没有什么特别，但非常有力。我们可以写很复杂的函式(像是LeapYear)，然后透过每个 TDate 的对象来存取它的内容，一如它是个直觉的数据型别。请注意，ADay.LeapYear 跟 ADay.Year 一样是对象可以存取的属性，然而前者是个函式，后者是直接存取数据。等我们进行到第十章的时候，Object Pascal 用来存取属性的表示法时，存取属性的方法也一样。

笔记

在 C 阵营的编程语言中，要呼叫不用参数的函式时，无论如何都还是得加上小括号，以上面的例子来看，就得写成 ADay.LeapYear()。这样的写法在 Object Pascal 也是合法的，但很少人使用这个写法。在 Object Pascal 里面呼叫不需要参数的函式时，通常是习惯不加小括号的。这也跟许多呼叫无需参数的方法时，会直接回传该函式地址的编程语言非常不同。我们曾经在第四章的"程序型别"当中介绍过，Object Pascal 在呼叫程序跟函式时是使用相同的呼叫方法，取决于该判断式的语意内涵。

这段源码的结果非常直觉：

```
Leap year: 2020
```

我们再来比较一下 Object Pascal 跟其他编程语言在对象的建立写法:

```
// C# and Java languages (object reference model)
Date aDay = new Date();
// C++ language (two alternative styles)
Date aDay; // local allocation
Date* aDay = new Date(); // "manual" reference
```

对象参考模型(Object Reference Model)

在一些 OOP 语言，像 C++，宣告一个类别型别的变量时就会自动出建立一个该类别的实体(多多少少有点像在 Object Pascal 里面的记录)。局部变量的对象所使用的内存会从 Stack 配置而来，然后当该函式结束时就会把这些内存释放掉。在大多数情形下，我们必须使用指针跟参考让管理对象生命周期的作业更有弹性，但也增加了额外的复杂度。

相反地，Object Pascal 语言是以对象参考模型为基础，完全像 Java 或 C#。这个概念是说，每个类别型别的变量并不直接储存对象的数据(例如以上例来说，不直接储存 day, month, year)。而是只储存一个内存参考，或者说指针，来指向一个真正用来储存对象数据的内存地址。

笔记

从我的观点来看，使用对象参考模型是早期 Object Pascal 编译器团队所做的最好的决定之一，在当时这种模型在编程语言当中已经不常被使用(事实上，当时 Java 还没完全发展出来，且 C#也还不存在)。

这也是为什么在这些编程语言中，我们需要额外建立一个对象，然后把它指派给一个变量，因为对象并不会自动被进行初始化。换句话说，当我们宣告一个变量，我们并没有在内存里面建立一个对象，而只是预留了一个内存空间来储存对象的参考地址而已。对象实体必须由我们撰写的程序来建立，至少由我们自己定义的类别必须如此。(在 Object Pascal 里面，我们从元键盘拖拉到窗体上面的各个组件的实体，就是由 RTL 自动建立出来的)

在 Object Pascal 当中，要建立一个对象的实体，我们必须呼叫该类别的特殊函式:Create，这个函式是一个建构函式，或者由类别自行定义的建构函式，写法像是这样:

```
ADay := TDate.Create;
```

如我们所见，建构函式是由类别(也就是一个型别)，而不是由对象(变量)来呼叫的。这是因为我们是要求类别建议一个新的实体，而结果会是一个新的实体，我们就可以把这个实体指派给一个变量了。

Create 这个方法是从何而来呢？它是 TObject 这个类别的建构函式，这个类别是 Object Pascal 所有对象的起源对象，所以在 Object Pascal 里面的所有类别全都依循相同规则。为我们定义的类别建立自行定义的建构函式也很常见，容我们在本章稍后再进行介绍。

释放对象

在使用对象参考模型的语言中，我们在使用对象前，必须先建立它，然后在我们不再需要使用该对象的时候，也要负责把该对象使用的所有内存空间释放掉。如果我们没有释放这些内存空间，这些残留在内存里面的空间将不会有机会被其他变量使用到，就会造成名为内存裂缝(memory leak)的问题。为了解决这个问题，有些语言，像是 C#跟 Java，就建立了虚拟执行环境(或称虚拟机)，让整个环境进行垃圾收集机制(garbage collection)。这个机制让开发人员的生活过的轻松了些，但这个功能要达到的要求是很复杂的，会牵涉到执行效能的问题，这又跟 Object Pascal 的关系不大了，所以我没办法在这里扯太多。

在 Object Pascal 里面，我们通常呼叫对象里面一个特殊的方法，名为 Free，来释放该对象所使用的内存空间(再强调一下，由于这是 TObject 的方法之一，所以任何一个类别都能使用)。Free 会在呼叫过该类别的解构函式(这里面可能有很特别的清除源码)之后，释放掉该对象所占用的内存空间，所以我们可以把上例当中的对象这样释放掉：

```
var
    ADay: TDate;
begin
    // Create
    ADay := TDate.Create;
    // Use
    ...
    // Free the memory
    ADay.Free;
end;
```

虽然这是标准的作法，但组件函式库在其中加入了类似对象归属的概念，以减少手动管理内存的冲击，让处理相关议题来说相对简单。

笔记

我们在后面的章节会介绍到，当接口参考到对象时，编译器会使用自动参考计数(ARC)的方式来做内存管理。这几年来，ARC 也被用 Delphi 行动装置版本的编译器用来管理一般类别型别的变量。从 Delphi 10.4 Sydney 开始，在所有作业平台上的内存管理，已经都统一为使用传统方式了。

在内存管理跟 ARC 这个主题上我们应该要更深入了解，但这是一个重要的课题，且并不简单，所以我决定在此提供一个简短的介绍，并且在第 13 章进行深入的探讨。在该章中，我会让各位看到在不同平台上所使用的这些不同技术的细节，以及他们是如何在所有平台上都能正常运作的。

什么是”Nil”

我们已经提过，变量可以指向特定类别所建立的对象，但变量并不能自己为对象做初始化，也不能把使用到的对象转成不再使用的状态。这时我们就可以使用 nil 了。这是一个特殊的常数值，用来表示变量已经不再指派给任何对象了。其他编程语言则使用另一个识别代号 null 来表示同样的概念。

当一个类别型别的变量没有值的时候，我们可以用这个方式帮它做初始化:

```
ADay := nil;
```

要确认一个对象是否已经指派给该变量，我们可以用以下任一种写法：

```
if ADay <> nil then ...
if Assigned (ADay) then ...
```

不要犯下想透过把 nil 指派给对象来释放内存的错。把一个对象设定为 nil 跟把它释放掉是两个完全不同的作业。所以我们通常既需要呼叫 free，也必须把该对象的变量指向 nil 或者呼叫一个特殊功能的程序以同时满足两种需求，这个程序就是 FreeAndNil。再预告一下，第 13 章里面就会有更多相关信息，以及实际的范例程序了。

在内存里面的记录与类别

一如我们前面所提，记录跟对象的主要不同，在于他们的内存模型。记录型别变量使用区域内存，它们被当成参数传递的时候，默认是用传值模式，也就是说被当成参数传递的变量，其内容会被复制一份之后，传递给函式使用。这跟类别型别变量的配置是从易失存储器 heap 而来，以及会自动以传址(call by reference)方式传递，并且在复制时还有”以参考复制”的特性恰好成为对比(会把该对象的指针复制一份，而不会实际上复制对象内容)。

笔记 在内存管理上的不同，是记录型别缺少了继承跟多型的机制，这两个功能我们会在下一章当中介绍到。

- private 这个存取分区描述关键词，宣告该类别位于此一区块内的字段跟方法，只能被宣告该类别的原始码档案使用。

举例来说，当我们在堆栈中宣告了一个记录型别的变量，我们在宣告完之后就可以立刻开始使用它了，不用呼叫它的建构函式(除非它是自定受管理的纪录)。这表示记录型别变量在内存管理上是比正规的对象更要来的精简(而且比较有效率)，因为记录型别的变量不需要进行内存配置管理。这也是在处理简单、单纯数据结构时，为何要建议使用记录型别，而不使用对象的关键之处。

另一个记录跟类别不同的地方是在当它们被作为参数传递的时候，比较看看把记录所使用的所有内存做一份完整的复制，跟把对象参考复制一份(数据没有复制)。当然我们也可以透过 var 跟 const 的参数来修改传递参数时的规则。

私有、保护、公开

- strict private 这个存取分区描述关键词，宣告该类别位于此一区块内的字段跟方法，只有该类别的方法可以取用，即使是同一个单元文件的其他类别也不可以，这个规则跟大多数面向对象编程语言的 private 关键词规则才是一样的。

一个类别可能拥有任何数量的数据字段跟任何数量的方法。然而，以好的面向对象设计内涵来说，数据应该被隐藏起来，或者说是『封装』在对象内部。例如当我们试着存取一个日期，直接改变当中的『日』内容是不对的。事实上，直接改变『日』这个字段可能导致整个日期变得不合逻辑，例如用户可能改出一个二月三十日这种不存在的日子。透过使用对象的方法，对象内部的表示法跟规则就可以尽量排除这种风险，例如透过对象的方法来设定日期，就可以判断使用者设定的日期有所错误，进而从对象本身直接拒绝新的错误日期被设定。适当的封装是相当重要的，因为它赋予了类别设计者在未来的版本中变更内部表示方法的权限。

其实封装的概念相当简单：就把类别想象成一个『黑盒子』，只露出一小部分可以看到里面。可以看到的部份，我们称之为*类别的接口*，这个部分允许其他源码透过属于这个类别的对象来存取、使用这个类别的功能。然而，当我们使用这个对象的时候，大多数的源码是被隐藏的。我们很少会知道一个对象当中有哪些内部数据，我们也无法直接存取这些数据，我们只能透过对象的方法来存取这些数据或使用它的方法。

封装使用了私有跟保护区的成员，这是面向对象解决方案提供给传统程序设计方法的目标，被称之为信息隐藏。

Object Pascal 有五种不同的基本存取分区描述关键词(有两个要搭配 strict 使用)，分别是:private (私有)、protected (保护)，以及 public(公开)。第六个特别的分区关键词则是 published(已发布)，这个我们在第十章里面讨论。以下是五个基本分区的特性，我们来仔细看一下:

- public 存取分区关键词，在这一区里面的所有字段，可以被整个程序的任何一部分存取，不限于在哪个单元文件里面。
- protected 跟 strict protected 存取分区关键词，在这一区里面的字段，有部分存取的限制。只有该类别的对象，以及从这个类别衍生的子类别可以存取这一区的字段，除非是相同类别或者使用了 strict 关键词的情况，

我们会在下一章的『保护区字段与封装』里面仔细讨论。

一般情形下，类别的字段应该放在 private 或者 strict private 区块中，方法则放在 public 区块里。然而，凡事并非绝对的。如果我们需要制作类别内部使用的方法来进行某些作业，这些方法也可以被放在 private 或者 protected 区块中。如果我们很肯定类别设计时某些字段的型别定义不会在被修改，也可以把字段或变量放在 protected 区块中，然后可以从衍生类别中直接使用这些变量(我们在下一章里面会说明)，但比较不建议使用这样的设计方式。

以一般的情形来说，我们应该不去直接把数据放在公开区，在公开区应该透过*属性*来提供对数据字段的存取，我们也会在第十章里面一起介绍。*属性*，是其他 OOP 编程语言当中用来提供数据封装机制的延伸，也是 Object Pascal 里面非常重要的机制。

如前文所述，存取分区关键词只用来限制其他单元文件的源码存取该类别在这个单元当中所宣告跟实作的类别成员。这表示如果两个类别在同一个单元文件里面，在类别当中私有区宣告的数据字段就没有任何保护的作用，除非使用 strict private 区块来加以保护，使用这个作法会让数据的隐密性比较高。

笔记 C++语言中有个概念叫做友善类别，这是让特定类别可以存取其他类别的私有数据。顺着这个概念，我们可以说在 Object Pascal 的类别中，所有在同一个单元文件里面宣告的类别都自动被当成友善类别了。

私有区数据字段的范例

我们可以用个例子来示范透过分区存取关键词如何实现数据封装的理念，以此实作一个 TDate 类别的新版本：

```
type
  TDate = class
  private
      Month, Day, Year: Integer;
  public
      procedure SetValue (M, D, Y: Integer);
      function LeapYear: Boolean;
      function GetText: string;  procedure Increase;
  end;
```

在这个版本中，数据字段现在被宣告在私有区内，并提供了其他方法。第一个方法是 GetText，这个方法是个会以字符串形式回传日期的函式。我们未来也可以加入其他方法，像是 GetDay，GetMonth，GetYear 可以用来回传各个在私有区的日期字段，只是类似的直接存取数据的函式并不会永远需要。提供可以直接存取每个字段的函式是可以简化封装的过程，但也使抽象化的概念被弱化了，同时也让未来要修改该类别的内部实作方式更为困难。

对数据字段存取数据的函式只宜在该类别有实际需要提供特定字段给其他源码存取时才提供，并不必凡数据字段必有存取函式。

第二个新增的方法是 Increase 程序，这个程序会在呼叫时为日期内容向后推一天。这并不像所描述的这么好做，因为我们得考虑每个月有不同的天数、以及该年是否为闰年。我们为了要让源码变得比较容易写而作的，是使用用 Object Pascal 的 TDateTime 型别作为这个类别内部实作的方法。所以实际上我们在 Dates2 项目类别里面所看到的类别源码会长的像这样：

```
type
    TDate = class
    private
        FDate: TDateTime;
    public
        procedure SetValue (M, D, Y: Integer);
        function LeapYear: Boolean;
        function GetText: string;
        procedure Increase;
    end;
```

请留意，这里的修改只有类别的私有区，我们不用因此修改任何使用到这个类别对象的源码，这也是信息封装设计的好处之一。

笔记 在这个类别的新版本中，唯一的数据字段的名称第一个字符是以 F 开始的。这是在 Object Pascal 语言里面的不成文规定，我也会在本书中沿用下去。

在这一节的结尾，我们把这个范例项目完成，我们把该类别所有方法的原始码都列出来，其中会使用一些系统函式来做日期跟内部结构的相互对照：

```
procedure TDate.SetValue (M, D, Y: Integer);
begin
    FDate := EncodeDate (Y, M, D);
end;
```

```
function TDate.GetText: string;
begin
    Result := DateToStr (FDate);
end;

procedure TDate.Increase;
begin
    FDate := FDate + 1;
end;

function TDate.LeapYear: Boolean;
begin
    // call IsLeapYear in SysUtils and YearOf in DateUtils
    Result := IsLeapYear (YearOf (FDate));
end;
```

也请注意到这个函式码是怎么使用这个类别的，它已经不再使用 Year 的资料内容，但可以从日期对象的回传信息中撷取出该项信息：

```
var
    ADay: TDate;
begin
    // Create
    ADay := TDate.Create;

    // Use
    ADay.SetValue (1, 1, 2016);
    ADay.Increase;

    if ADay.LeapYear then
     Show ('Leap year: ' + ADay.GetText);

    // Free the memory (for non ARC platforms)
    ADay.Free;
```

输出数据跟原来的不同：

```
Leap year: 1/2/2020
```

也请留意，您执行的结果可能跟这个字符串不同，因为日期格式会跟每台计算机的日期设定有关。

封装与窗体

封装技术的主要要求之一，是减少程序里面使用全局变量的数量。全局变量可以在一个程序的任何地方加以存取。为了这个原因，全局变量的改变会影响到整个程序。另一方面，当我们改变了类别中数据字段的设计，我们只需要修改该类别有使用到该数据字段的部分源码，其他部分不用动。所以我们可以说信息隐藏也引用了封装的改变。

让我们用个实例来厘清这个概念。当我们写了一个多重窗体的程序时，我们可能会为了让所有窗体可以存取同一份数据，而在全局变量上宣告它：

```
var
    Form1: TForm1;
    NClicks: Integer;
```

这样宣告可以正常运作，但有两个问题。第一，数据(NClicks)并不是跟窗体上的任何一个字段有所连结，而是跟整个程序链接。如果我们用同样的型别制作了两个窗体，预期让两个窗体之间互相共享数据，这是可行的。如果我们希望各个窗体都能有它自己的数据，唯一的方法是在窗体的类别中加入:

```
type
    TForm1 = class(TForm) public
        NClicks: Integer;
    end;
```

第二个问题则是，如果我们把该数据定义成全局变量或者一个位于公开区的字段，我们未来就不能在不影响使用到被修改字段相关源码的情形下，对这个类别的内部进行重新设计了。例如，如果我们只需要从其他窗体读取目前的数值，我们可以把该数据字段设计成私有，并且提供这样的读取方法：

```
type
  TForm1 = class(TForm)
      // Components and event handlers here
  public
      function GetClicks: Integer;
  private
      FNClicks: Integer;
end;


function TForm1.GetClicks: Integer;
begin
    Result := FNClicks;
```

```
end;
```

另一个比较好的解决方法，是帮窗体加入一个属性，我们在第十章再慢慢说。我们现在可以先用 ClicksCount 范例项目来做些实验。简单的说，这个项目的窗体上方有两个按钮、一个卷标，其他部分的窗体画面则留白让使用者可以点击(或触击)。在这个案例中，计数值会增加，接着该数值会被显示在卷标上：

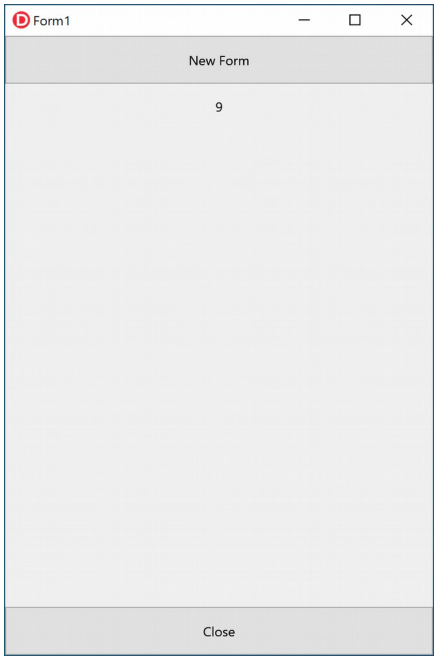
```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Single);
begin
  Inc (FNClicks);
  Label1.Text := FNClicks.ToString;
end;
```

在图 7.1 里面可以看到应用程序执行的画面。项目的窗体也有两个按钮，一个是用来建立一个相同型别的新窗体，第二个按钮则是关闭它(所以我们就可以把窗口焦点移回前一个窗体上)。这是为了强调每一个不同的实体被点击时，可以有自己的点击计数数值。以下是这两个方法的源码:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewForm: TForm1;
begin
  NewForm := TForm1.Create(Application);
  NewForm.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

图 7.1: ClicksCount 范例项目的窗体，显示该窗体实体被点击的次数(记录在该窗体的私有区)



Self 关键词

我们已经看过了很像程序跟函式的一些方法，实际的不同处，是方法会有一些额外的、预设的参数。该参数是一个指向触发该方法的对象参考。在方法之内，我们可以透过这个参数来存取正在执行中的对象，或使用 self 这个关键词(在第五章的『Self: 记录神奇的地方』当中我们也介绍过)。

这个额外的参数是有需要的，尤其是当我们建立相同类别的多个对象时。当我们每次使用该对象的任何一个方法时，这个方法只会对该对象实体有作用，对同一类别的其他实体则不会有任何作用。

笔记 实作出记录跟类别的 Self，在概念跟作法上都非常相似。Self 是在提出类别的时候先被提出的概念，后来才延伸到记录的，而当时方法则也已经加入了这个数据结构当中。

举例来说，在我们稍早提到的 TDate 类别里面的 SetValue 方法，我们很简单的直接用了 Month, Year, Day 来对应目前对象的字段，我们可以直接写出这样的源码:

```
Self.Month := m;

Self.Day := d;
```

事实上这也是 Object Pascal 编译器转换源码的方法，我们不用自己写出这些源码。Self 关键词是会由编译器建立的程序核心基础，但有时候程序人员也会自己写出来，可以让源码更容易被读懂，也比较不容易发生同名成员之间的混淆或误解。

笔记

C++, Java, C#, JavaScript 语言当中，也有相同功能的关键词，他们使用的是 this。然而 JavaScript 跟 C++, C#, Java 不一样，JavaScript 在方法中使用这个关键词来指定对象参考是必要条件。

我们真的需要知道的是 Self 是呼叫方法时的技术实作，跟呼叫一般子程序是不一样的。方法会有 Self 这个额外的隐藏参数存在。因为这一切都会在不知不觉中就由编译器处理好了，我们不需要知道 self 在这个情形下是怎么作业的。

第二重要的思路，是我们可以在整个源码当中，自己明确输入 self 这个关键词，来引用目前的对象，例如把目前的对象当成参数传给另一个函式。

动态建立组件

在我们刚刚提过的例子里，self 关键词在我们需要明确参考到目前这个窗体或对象时是很常用到的。常见的例子，就是当我们在运行时间的时候需要动态建立组件时，因为建立组件的时候需要指定该组件的拥有者作为呼叫建立组件建构函式(Create)的参数，并且把同一个值指派到新建组件实体的 Parent 属性。在上述的两个情形中，我们都需要提供当前窗体的对象参考作为参数，或者要被指派的数据，而完成这个作业最方便的方式就是使用 Self 关键词了。

笔记

组件的所有权会关系到它的生命周期以及两个对象内存管理的关联性。当一个组件的所有权者被释放掉了，该组件也会一起被释放掉。Parent(父代关系)所指的，则是在视觉层级上，这个组件要被贴在哪个组件上头。

为了进行这样的展示，我提供了 CreateComps 范例项目，这个应用程序只有一个干干净净的窗体，以及处理 OnMouseDown 事件的处理程序，这个处理程序也会把鼠标点击的坐标当成标准参数之一。因为我们需要这个坐标来当成新建立组件要放置的位置。

笔记

事件处理程序(event handler)是一种特殊的方法，我们在第十章里面会介绍，本章里面我们会先介绍这一系列里面的成员：按钮的 onClick 处理程序。

以下是这个方法的源码:

```
procedure TForm1.FormMouseDown (Sender: TObject;
      Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
    Btn: TButton;
begin
    Btn := TButton.Create (Self);
    Btn.Parent := Self;
    Btn.Position.X := X;
    Btn.Position.Y := Y;
    Btn.Height := 35;
    Btn.Width := 135;
    Btn.Text := Format ('At %d, %d', [X, Y]);
end;
```

请注意，我们需要在 uses 叙述句中，加入使用 StdCtrls 这个单元文件的描述，这样编译才会过喔。

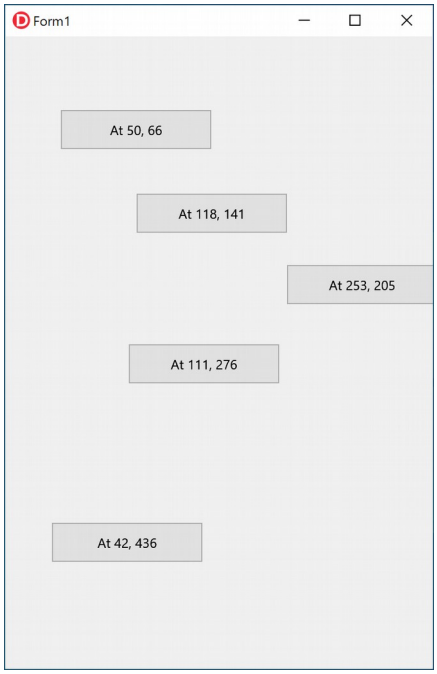
这段源码的作用，是会在鼠标点击的地方建立一个新的按钮，并且把该坐标当成按钮的卷标文字，就像图 7.2 所示(在这个项目中，我取消了 FMX Mobile Preview 的功能，所以显示出 Windows 原生样式的按钮，这样看起来会比较清楚)。在以上的源码当中，请特别留意 self 关键词的使用，尤其在我们呼叫 Create 方法跟指派 Parent 属性的时候。

当撰写一个跟上述范例相似的程序时，您可能会很想用 Form1 来取代上面源码里面的 Self。在上述这个范例中，如果您这么写，并没有造成任何实质的变化(即使它并不算一个好的源码范例)，但如果一个窗体会出现多个实体的时候，使用 Form1 来取代 self，就会出现很严重的错误喔。

事实上，如果 Form1 这个变量指向该窗体类别的一个实体(通常会是第一个被建立的实体)，而我们用相同窗体类别建立了第二个实体，那么，在点击任何一个窗体的画面时，都只有第一个窗体的画面上会出现新增的按钮，因为新建组件的 Parent 是 Form1，而不是被点击的窗体实体了。

通常，当我们在撰写一个类别的源码，需要使用到当前对象参考时，如果把它指向特定一个实体的变量值时，绝对是 OOP 的一个很糟很糟的作法。

图 7.2: CreateComps 范例的执行画面



建构函式

在上述的源码中，要建立一种类别的对象(您也可以想做要帮一个对象配置内存空间)时，我们会呼叫 Create 方法。它是建构函式(constructor)，是所有类别中一个预设的特别方法，让我们可以用来为类别的一个新建实体配置内存之用：

```
ADay := TDate.Create;
```

由建构函式回传的实体，我们可以把它指派给一个该类别为型别的变量，然后后续用这个变量就能使用这个新建的实体了。

当我们建立一个对象，它的内存已经完成了初始化，所有新对象的数据字段都该被设定为 0(或者 nil，或者空字符串，或者该数据型别的适当默认值)。

如果我们想要让我们的新建实体在建立之时就储存特定数据，那我们就需要写一个自定的建构函式来这么做。新的建构函式可以名为 Create，或者它也可以用任何一个名称。用来决定方法的角色的，并不是名称，而是特定的关键词: constructor。

笔记

换句话说，Object Pascal 支持可命名的建构函式，其他大多数的 OOP 编程语言的建构函式大多只能是该类别的名称。透过可命名的建构函式，我们可以拥有多个建构函式，让这些建构函式的参数是一样的(也可以直接对 Create 进行多载 – 我们在下一节介绍多载)。Object Pascal 另一个特殊的功能，也是在 OOP 编程语言中很独特的，就是建构函式也可以是虚构的(virtual)，下一章当中，我们在介绍过虚拟函式概念之后，就来花一些篇幅来探讨。

为一个类别建立自定建构函式的主要原因，是要进行数据的初始化，如果我们建立了一个对象，但没有为它做初始化，稍后在呼叫它的方法时，可能会得到很奇怪的结果，或者甚至得到执行程序错误。与其等到这些错误显示，我们应该用一些方法预先避免这种情形发生。这些方法的其中之一，就是持续使用建构函式来对对象数据进行初始化。举例来说，我们在建立 TDate 类别的对象之后，得先呼叫 SetValue 方法来设定一个日期给它。或者我们也可以提供一个自定的建构函式，在建立对象之时，同时就把预设日期给设定好：

```
constructor TDate.Create;
begin
    FDate := Date;    // Today
end;

constructor TDate.CreateFromValues (M, D, Y: Integer);
begin
    FDate := SetValue (M, D, Y);
end;
```

我们可以这样来使用这些建构函式，这些源码我已经在 Date3 范例项目中写好了，在这个范例项目中，以下的源码是分别由两个按钮来触发的：

```
ADay1 := TDate.Create;
ADay2 := TDate.CreateFromValues (12, 25, 2015);
```

即使我们通常可以使用任何一个名称来做为建构函式，要记得如果我们使用的名称不是 Create，Create 这个建构函式仍旧是有效的，因为它是从 TObject 一路继承下来的。所以如果我们正在开发要提供给其他程序人员使用的源码，其他程序人员直接呼叫 Create 建构函式时，就不会呼叫到你辛苦写的建构函式了。我们也可以直接新增一个名为 Create 带有完全相同参数的建构函式，这样就能取代掉原来的建构函式，确保使用上不会有失误了。

如同类别可以有自定的建构函式，类别也可以拥有自定的结构函式，以destructor 为名宣告的方法，呼叫时直接呼叫 Destroy 即可。这个解构函式可以在对象被从内存毁灭之前，先把使用过的资源清除掉，但大多数的情形下，并不需要自定的解构函式。

就像建构函式会为一个对象配置内存，解构函式则是会释放内存。自定解构函式只有在该类别的建构函式或者执行过程中要求了额外资源时才需要。

跟预设的 Create 建构函式不同，预设的 Destroy 解构函式是虚拟的(Virtual)，而且强烈建议开发人员覆写(override)这个虚拟的解构函式(虚拟函式会在下一章里面介绍)。

这是因为与其直接呼叫一个对象的解构函式，不如透过惯用的 Object Pascal 程序实务上的作法，呼叫名为 Free 的特殊 TObject 类别方法，这个方法会自动转为呼叫 Destroy 函式，如果该对象有定义它的话，也就是说该对象的 Destroy 如果不是 nil 的话，所以，如果我们定义了一个解构函式使用的名称不是 Destory 的话，这个解构函式就不会被 Free 呼叫了。再次重申，与此相关的更多主题我们会在第十三章里面的内存管理来介绍。

笔记 如下一章我们介绍的内容，Destroy 是一个虚拟方法，我们可以在继承特定类别，衍生新的次类别时取代原有的定义，透过 override 关键词。顺带一提，使用静态方法(static method)来呼叫一个虚拟方法是很常见的程序风格，称为模版模式(template pattern)。在解构函式中，我们通常只需要撰写清除资源的源码。试着避免太复杂的处理，例如发出例外，或者花上可观的时间，可以避免在对象清理时的问题，因为许多解构函式会在程序结束被呼叫，我们会希望让它执行速度尽可能快一些。

以建构函式跟解构函式来管理区域类别数据

虽然我们会在本书稍后的篇幅来介绍更复杂的情境，在此我想先用简单的案例来介绍以建构函式、解构函式来提供资源保护的功能。这也是使用解构函式最常见的情境。

假设我们有个如下宣告结构的类别(这也是 Date3 范例项目的一部分):

```
type
    TPerson = class
    private
```

```
        FName: string;
        FBirthDate: TDate;
    public
        constructor Create (name: string);
        destructor Destroy; override;
        // some actual methods
        function Info: string;
    end;
```

这个类别包含了一个内部对象的参考，名为 FBirthDate。当 TPerson 类别的实体被建立的时候，这个内部(或者子)对象应该也要同时被建立，而当这个实体被摧毁，内部这个对象也要一起被摧毁。

以下是我们可以用来撰写这个建构函式以及覆写后的解构函式源码，以及可以永远用来确保内部对象存在的内部函式。

```
constructor TPerson.Create (name: string);
begin
    FName := Name;
    FBirthDate := TDate.Create;
end;

destructor TPerson.Destroy;
begin
    FBirthDate.Free;
    inherited;
end;

function TPerson.Info: string;
begin
    Result := FName + ': ' + FBirthDate.GetText;
end;
```

笔记 要了解用来定义解构函式覆写(override)这个关键词，以及在函式中的 inherited 这个关键词，请等到下一章。目前我们先简单的说，第一个关键词是用来指出该类别有一个新的定义，要用来取代掉基础的 Destroy 解构函式。而第二个关键词则是用来呼叫该类别上一代的结构函式。也请记得，override 只用在方法的宣告，不会出现在实作的源码之中。

现在我们可以在以下的情境中使用外部类别的对象，而内部对象会适时的在 TPerson 对象被建立的时候一起被建立起来，也会在 TPerson 被摧毁时一起被处理掉：

```
var
    Person: TPerson;
begin
    Person := TPerson.Create ('John');
    // Use the class and its internal object
    Show (Person.Info);
    Person.Free;
end;
```

以上的源码，可以在 Date3 范例项目中找到。

多载方法以及建构函式

Object Pascal 支持对函式与方法的多载：我们可以用同一个名称建立许多方法，但这些方法的参数必须不一样。我们已经看过全局函式跟程序的多载是怎么运作的，同样的规则也适用于方法。透过检查参数，编译器可以判断出我们想要呼叫的方法是哪个版本。

重申一次，多载的基本规则有两个：

- 每一个版本的方法必须在宣告时最后都加上 overload 关键词。
- 每一个版本的参数型别或数量必须有点不同，然而回传值并不能拿来作为多载方法的判别条件。

如果多载被用在一个类别里面的所有方法上，这个功能也会跟建构函式有关，因为我们可以建立多个建构函式，并且都将之命名为 Create，这会让建构函式更容易被记忆。

笔记 来谈点历史，多载功能被加入到 C++就是特别为了让多个建构函式可以被使用，因为 C++的建构函式必须都是相同名称(类别的名字)。在 Object Pascal 里面，这个功能相对的不是很需要，因为多重建构函式本来在 Object Pascal 里面就可以有不同的名称，即使如此仍旧把这个功能加入，则是因为在其他情形底下，多载功能仍然很有用处。

我在 TDate 类别里面加了两个不同版本的 SetValue 方法，作为多载的范例：

```
type
    TDate = class
    public
        procedure SetValue (Month, Day, Year: Integer); overload;
        procedure SetValue (NewDate: TDateTime); overload;


procedure TDate.SetValue (Month, Day, Year: Integer);
begin
    FDate := EncodeDate (Year, Month, Day);
end;


procedure TDate.SetValue(NewDate: TDateTime);
begin
    FDate := NewDate;
end;
```

在这个简单的步骤之后，我也在这个类别里面加入了两个独立的 Create 建构函式，其中之一不用参数，这个版本把预设的建构函式直接取代掉了，另一个则会进行数据初始化。不用参数的建构函式会使用今天的日期作为默认的数据：

```
type
    TDate = class
    public
        constructor Create; overload;
        constructor Create (Month, Day, Year: Integer); overload;


constructor TDate.Create (Month, Day, Year: Integer);
begin
    FDate := EncodeDate (Year, Month, Day);
end;
constructor TDate.Create;
begin
    FDate := Date;    // Today
end;
```

有了这两个建构函式，我们就可以用以下两种方法来建立新的 TDate 对象了：

```
var
    Day1, Day2: TDate;
```

```
begin
    Day1 := TDate.Create (2020, 12, 25);
    Day2 := TDate.Create; // Today
```

这段源码是 Dates4 范例项目的一部分。

完整的 TDate 类别

综观这一节，我们已经在不同的程序片段里面看过了 TDate 类别。第一个版本是以三个整数来储存年月日三个部分的数值。第二个版本则是使用 TDateTime 型别作为内部数据，以下是完整的 TDate 类别定义:

```
unit Dates;
interface
type
  TDate = class
  private
       FDate: TDateTime;
  public
      constructor Create; overload;
      constructor Create (Month, Day, Year: Integer); overload;
      procedure SetValue (Month, Day, Year: Integer); overload;
      procedure SetValue (NewDate: TDateTime); overload;
      function LeapYear: Boolean;
      procedure Increase (NumberOfDays: Integer = 1);
      procedure Decrease (NumberOfDays: Integer = 1);  function GetText: string;
  end;
```

新增的两个方法：Increase 跟 Decrease(两个方法都有预设的参数值)，也都相当的容易理解。如果呼叫时不给参数，他们会直接把内部数值改成前一天或下一天，如果 NumberOfDays 参数有被指定，则会依据这个参数把内部数值改成往前或往后几天：

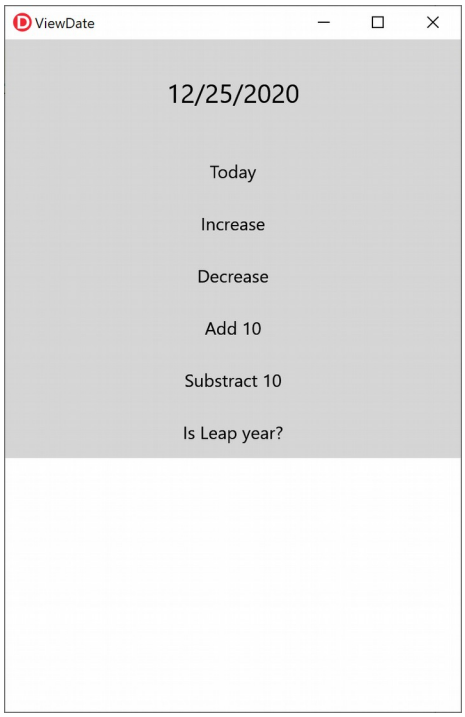
```
procedure TDate.Increase (NumberOfDays: Integer = 1);
begin
    FDate := FDate + NumberOfDays;
end;
```

GetText 方法则会以格式化的字符串回传内部数值的日期，我们透过 DateToStr 函式来做这个转换：

```
function TDate.GetText: string;
begin
    GetText := DateToStr (FDate);
end;
```

我们已经看过了前一节当中的大多数方法，所以我们不再做完整的列表了，您可以从 ViewDate 范例中找到这些源码。这个窗体的内容比书上其他的范例都来的复杂。它包含了一个用来显示日期的卷标，以及六个按钮。这六个按钮都可以用来变更对象内部的日期数据。您可以从图 7.3 里面看到该项目的窗体画面。为了让卷标组件好看，我用了比较大的字体，让它跟窗体一样宽，并且把对齐的属性(Alignment)设定成 Center，也把 AutoSize 设为了 False。

图 7.3: ViewDate 范例的执行画面



这个程序第一个被执行的源码，会是窗体的 OnCreate 这个事件处理程序。在对应的方法中，我建立了一个 TDate 类别的实体，然后把这个建立的对象初始化，最后把它的内容以字符串格式显示在窗体上的卷标文字字段，如上图(图 7.3)所示。

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
   ADay := TDate.Create;
```

```
    LabelDate.Text := ADay.GetText;
end;
```

ADay 是 TDateForm 这个窗体类别的私有字段。顺带一提，这个类别的名字是当我们把窗体的名称字段改为 DateForm 时，开发环境自动修改的。

指定的日期对象是在窗体被建立时自动建立的(就像我们刚刚看过的 person 类别跟它的日期子对象关系一样)，也会在窗体被摧毁时自动被摧毁：

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
    ADay.Free;
end;
```

当用户点选六个按钮的其中一个，我们需要使用 ADay 对象的对应方法，并把新的数值内容显示在卷标文字上：

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
    ADay.SetValue (Today);
    LabelDate.Text := ADay.GetText;
end;
```

另一个撰写最后一个方法源码的替代方案则是直接把目前的对象摧毁掉，然后建立一个新的：

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
    ADay.Free;
    ADay := TDate.Create;
    LabelDate.Text := ADay.GetText;
end;
```

在这个特殊的情形中，这个例子并不够好(因为建立一个新的对象，并把它摧毁掉，会花很多额外的时间，尤其是当我们只是想更改对象内容的时候)，但透过这个案例，我可以示范一些 Object Pascal 里面的技术。第一个值得一提的是我们在指派一个个新的对象前，会先摧毁前一个物件。事实上这个指派的动作是更换了参考值(reference)，把对象留在内存里面(即使可能并没有任何指标指向它)。当我们把一个对象指派给另一个，编译器只会把对象在内存里面的参考地址复制到新的对象参考地址。

另一个问题，是我们怎么把数据从一个对象复制到另一个。在上面这个源码的案例非常单纯，因为它只有一个数据字段跟对应的方法用来初始化。通常如果我们想要改变存在一个既存的对象中的数据，我们得复制每一个字段，或者提供一个特殊的方法来复制所有内部数据。部分类别会拥有一个名为 Assign 的方法，可以用来进行深度复制的作业。

笔记 更精确一点，在 RTL 里面，所有类别都是从 TPersistent 类别衍生而来，这个类别提供有 Assign 方法，但大多数从 TComponent 衍生的类别则没有实作它，所以呼叫组件的 Assign 会造成例外发生。原因是因为 RTL 支持的串流化机制以及对 TPersistent 型别的属性支持，但在这个章节说明这一点还太复杂。

嵌套类型与巢状常数

Object Pascal 允许我们在一个单元文件里面的 interface 区段宣告新的类别，在这里宣告的类别，可以被其他单元文件存取，也可以被 implementation 区段的程序存取。在 implementation 区段里面的程序与类别，则只能被存在同一个单元文件里面的源码或全局子程序(也只限于写在同一个单元文件里面)所存取。

最近比较新的一个作法，是在一个类别当中宣告另一个类别(或任何其他的数据型别)的可能性。而该类别的任何其他成员，嵌套类型都会有被限制的可见度(或者用我们已经介绍过的名词:私有区，或保护区)。嵌套类型的相关范例包含同一类别与其他在 implementation 区段支持的类别所使用的列举作业。

相关的语法允许我们定义一个巢状常数，这个常数可以跟一个类别相关(再提一次，如果在私有区宣告，就只允许内部使用，如果在公开区宣告，则在整个程序的任何一个部分都能使用)。举例来说，下面的源码就宣告了一个巢状类别(这是从 NestedTypes 范例项目中，NestedClass 单元文件所节录的):

```
type
    TOne = class
    private
        FSomeData: Integer;
    public
        // Nested constant
        const Foo = 12;
```

```
        // Nested type
        type
            TInside = class
            public
                procedure InsideHello;
            private
                FMsg: string;
            end;
       public
         procedure Hello;
    end;

procedure TOne.Hello;
var
    Ins: TInside;
begin
    Ins := TInside.Create;
    Ins.Msg := 'Hi';
    Ins.InsideHello;
    Show ('Constant is ' + IntToStr (Foo));
    Ins.Free;
end;

procedure TOne.TInside.InsideHello;
begin
    FMsg := 'New msg';
    Show (‘Internal call');
    if not Assigned (InsIns) then
       InsIns := TInsideInside.Create;
    InsIns.Two;
end;

procedure TOne.TInside.TInsideInside.Two;
begin
    Show ('This is a method of a nested/nested class');
end;
```

巢状类别可以在类别中直接使用(例如在前例的范例源码中的用法)，或者在类别之外使用(如果巢状类别宣告在公开区的话)，但必须完整把名称写上：TOne.TInside。完整的类别名称也在巢状类别的方法定义中使用。在这个例子里面，就得写 TOne.TInside 了。主类别可以在宣告巢状类别之后，立刻拥有一个以巢状类别为型别的数据字段(就像我们看到在 NestedClass 范例项目中的源码一样)。

拥有巢状类别的类别可以这样使用：

```
var
    One: TOne;
begin
    One := TOne.Create;
    One.Hello;
    One.Free;
```

这段源码的输出值如下:

```
Internal call
This is a method of a nested/nested class
Constant is 12
```

我们从使用 Object Pascal 的巢状类别到底得到了什么好处?这个概念在 Java 里面很常用来实作事件处理程序，也在 C#的类别中无法在同一档案中的其他类别隐藏信息时使用。在 Object Pascal 当中唯一使用到巢状类别的用途是当我们在类别的私有区里面宣告一个类别数据字段时，不想把这个类别加到全局的命名空间，也不让它被全局命名空间看见，此时，巢状类别就派上用场了。

如果内部类别只用在一个方法中，我们可以透过在单元文件的 implementation 区段宣告该类别来达成相同的效果。但如果这个内部类别会在同一单元文件的 interface 区段被参照使用的话(例如，我们要把它作为类别的字段或者函式的参数)，这个类别就必须在相同的 interface 区段中宣告，当然也就会暴露在其他源码之下了。在基础型别中宣告这样的字段，然后把它放在私有区实在是不如使用巢状类别来的干净利落。

笔记 在第十章，我们提供了一个实际的案例，对巢状类别做了一些练习，用了 for-in 循环做了一些实作。

08:继承

如果撰写类别的主要原因是为了封装，那么使用继承的主要理由就是为了弹性。把两个概念结合之后，我们就可以拥有一个强大的资料型别，我们可以拿来使用，而且以后不用为了改其中的功能而为这个数据型别来建立许多不同版本了，这个概念一开始被称为”开放式封闭概念(open-close principle)”:

> “软件设计的概念(类别、模块、函式等)应该要有能延伸的开放性，但对修改则要有封闭性” - Betrand Meyer, 面向对象软件建构, 1988

继承能够把一些源码很紧密的结合起来，也能够让程序人员发挥很强的功能(当然，也会带来更大的责任)。

并不是在这里开启对这个功能的争论，我们在这里要做的，是介绍型别继承是怎么作用的，尤其在 Object Pascal 中，型别继承是怎么作用的。

从既存的型别中继承

我们通常需要使用一些不同的功能，这些功能可能是从我们自己之前写过的源码，或者从别人写过的源码里面部分功能提供出来的。例如，我们可能会需要为已经写好的源码加上一些新的方法或者做一些改变。我们当然可以直接改原来的程序原始码，但有时候我们会同时需要这两种版本同时存在，在不同的情形中分别使用到。或者有个类别是别人写的(或者我们从函式库里面找到的)我们可能想要保留一个我们自己的独立版本。

传统在学校里面教授过的替代方法，是把原来的类别源码留一份下来，直接改一份新的源码，两份并存。这样做也可以，不过也可能创造问题:在重复的源码里面，我们可能也重复了原本的程序错误，当我们修正了其中一份源码的问题，也必须记得把这个修正套用到另外一份源码。如果您想要在源码中加入一个新功能，也必须把相同的程序在每份源码都做一次，这还得看你复制了几份，就得处理几次。就算你第一次写这程序的时候，这种复制多份的作法并没有拖慢你的速度，但未来维护的时候，这绝对是个大灾难。再者，这几个不同版本的类别源码最后会变成不同的独立型别，

编译器没办法告诉我们哪个版本才是最好的，也没办法告诉我们两个版本之间有哪些雷同之处。
如果我们要加一个新功能，我们可能得在两份源码里面都各做一次(可能得多做很多次也说不定，要看我们最后一共搞了多少个版本出来)。再者，这几个不同版本的类别源码最后会变成不同的独立型别，编译器没办法告诉我们哪个版本才是最好的，也没办法告诉我们两个版本之间有哪些雷同之处。

为了解决这些因为类别之间相似而导致的问题，Object Pascal 允许我们用既存的类别直接定义出一个新的类别。这个技术就称为*继承*(或者*次类别*、或*型别衍生*)，这个技术也是面向对象设计编程语言的基础元素之一。

要从一个既存的类别进行继承，我们只需要在定义类别的第一行，指出我们要从哪个既存类别衍生出新类别即可。这个动作实际上会在我们每次建立一个新窗体的时候由 IDE 自动帮我们做好:

```
type
  TForm1 = class(TForm)
  end;
```

这个简单的宣告表示 TForm1 类别会从 TForm 继承所有的方法、数据字段、属性、以及事件。我们可以从 TForm1 的对象来使用 TForm 类别的任何一个公开方法。TForm 类别依序继承了其上每一代的对象所有的方法，一直回溯到 TObject 类别(这是 Object Pascal 里所有类别的基础类别)。

跟 C++, C#, Java 做个比较，用这三个语言来宣告，则会写成:

```
class Form1 : TForm
{
...
}
```

作为继承关系的一个简单范例，我们可以用上一章的范例来改写成 ViewDate 范例项目，从 TDate 衍生一个新的类别，并且把它的函式 GetText 改写一下。您可以从 DerivedDates 范例项目的 DATES.pas 档案找到这个源码：

```
type
    TNewDate = class (TDate)
    public
```

```
    function GetText: string;
  end;
```

在这个范例里，TNewDate 是从 TDate 衍生而来。简单的说，TDate 是 TNewDate 的父代类别，基础类别或者祖先类别。而 TNewDate 则是 TDate 的子类别、衍生类别，或者次类别。

为了实作新版的 GetText 函式，我使用了 FormatDateTime 函式，这个函式中使用了预先定义的月份名称(还有其他功能)，以下是 GetText 方法，其中的'dddddd'代表了长日期格式：

```
function TNewDate.GetText: string;
begin
  Result := FormatDateTime ('dddddd', FDate);
end;
```

一旦我们定义了新的类别，我们就得在 DerivedDates 项目的窗体源码里面使用这个新的数据型别。直接定义一个以 TNewDate 为型别的对象 ADay，然后在 FormCreate 方法里面呼叫这个类别的建构函式：

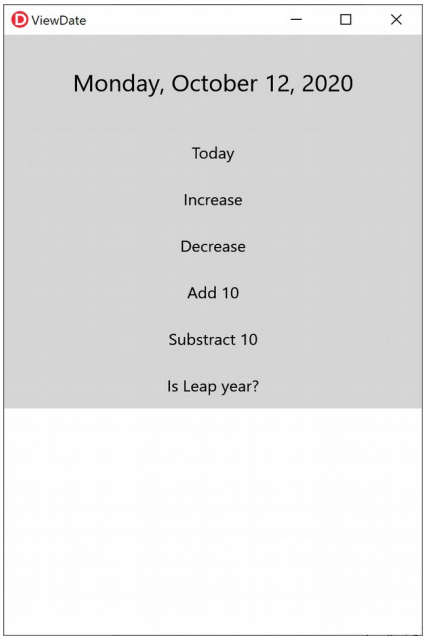
```
type
  TDateForm = class(TForm)
  ...
  private
    FDay: TNewDate; // updated declaration
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  FDay := TNewDate.Create; // updated line
  DateLabel.text := FDay.GetText;
end;
```

不用任何源码变更，这个新的范例项目就能执行的很顺畅。

新的类别 TNewDate 继承了所有 TDate 的方法，可以把日期设定向后延伸跟使用其他功能。换句话说，原来呼叫这些方法的源码仍旧可以正常运作。事实上，呼叫新版的 GetText 方法时，我们并不用改变任何源码！Object Pascal 的编译器会自动把呼叫这个方法跟新版的方法做绑定。所有事件处理程序的源码也都不用改变，虽然在新版的输出文字上，其意义已经跟旧版的事件有些不同(请见图 8.1)：

图 8.1: DerivedDates 程序的执行画面，月份跟日期的名字是依照 Windows 的地区与格式设定而显示的



常用的基础类别

我们已经介绍过，要从一个类别衍生新类别时，可以这样写：

```
type
  TNewDate = class (TDate)
    ...
  end;
```

但如果我们指定从基础类别衍生，写成以下这样的话，会发生什么事呢:

```
type
  TNewDate = class
    ...
  end;
```

在这种情形下，我们的类别就会从基础类别衍生，也就是 TObject。换句话说，Object Pascal 是单一根源的类别架构。在这种架构之下，所有类别直接或间接的都是从 TObject 衍生而来的。最常用的方法就是 TObject 的 Create, Free 跟 Destroy 方法，以及我在本书当中其他篇幅里面所用到的方法。关于这个基础类别的的完整介绍(可以把它视为在语言跟 RTL 两个层面都具备的基础)，我们会在第 17 章里面介绍。

笔记 共享基础类别的概念，除了 Object Pascal 以外，还有 C#跟 JavaScript 也是这样的架构，在这两个语言里面，基础类别称为 Object。C++则是完全不同的架构，C++是支持多重基础类别继承的架构。

保护区字段与封装

TNewDate 类别 GetText 方法的源码只有在跟 TDate 写在同个单元文件里面才会编译。事实上，它会存取父代类别的 FDate 私有区字段，如果我们想要在不同单元文件里面制作子类别的话，我们要不就得把 FDate 这个字段放在 protected(或者 strict protected)区域，或者在父代类别里加入一个简单的保护区方法来读取这个私有区的字段。

有些开发人员相信第一种作法是最好的，因为把大多数字段宣告在保护区，可以让类别比较有延伸性，要用它来撰写子类别也比较容易。然而，这就抵触了数据封装的原意。要在大架构的一群类别当中，想把其中基础类别的保护区字段做变更，就有如要修改部分全局数据结构一样困难。如果有 10 个衍生类别存取这个数据字段，对这个字段做变更，可能意味着有 10 个类别都必须要跟着做修改。

换句话说，弹性、延伸性，以及封装，这三个目标通常是有所冲突的。当冲突发生的时候，我们应该试着以封装这个目标为重。如果我们能够做到这一点，而不牺牲弹性，那就更棒。通常这个中间解决方案，可以透过使用虚拟方法来达成，虚拟方法我们会在稍后的”延迟绑定与多型”这一节里面来讨论。如果我们选择不以封装为重，而选择能让次类别的撰写比较容易、快速一点，那我们的设计很有可能就没办法依循面向对象的原则了。

也要记住，保护区的字段在存取规则上跟私有区是一样的，所以同一个单元文件里面的其他类别也随时可以存取任何一个同单元文件里面类别的保护区成员喔。如同前一章我们提过的，我们可以用更强的封装方式，只要使用分区存取关键词 srtrict protected 即可。

使用保护区来骇入 (Protected Hack)

如果您是 Object Pascal 的入门者，而且要迈入面向对象程序，那么这一节对您来说就会是相对进阶的章节，您或许会想先略过这一节，因为这一节可能会比较容易让您觉得难懂。

我们已经知道单元文件保护区的范围，即使是宣告在同一单元文件的类别所属的基础类别的保护区成员，也可以被同一单元文件的其他类别直接存取，除非我们使用 strict protected 关键词来加以保护。这种遵守 OOP 规则的漏洞，就称为『受保护的黑客』。这是能够定义一个衍生类别跟其父代类别完全相同，但只是为了取得其父代类别保护区成员的访问权限。以下是其工作的原理：

我们已经介绍过，一个类别的私有区、保护区，是可以被任何跟这个类别位于相同单元档案里面的其他类别存取的。举例来说，以下这个简单的类别(它是 Protection 范例项目的一部分):

```
type
    TTest = class
    protected
        FProtectedData: Integer;
    public
        PublicData: Integer;
        function GetValue: string;
    end;
```

GetValue 这个方法会简单的回传包含两个整数数值的字符串：

```
function TTest.GetValue: string;
begin
    Result := Format ('Public: %d, Protected: %d', [PublicData, FProtectedData]);
end;
```

一旦我们把这个类别放在独立的单元文件里面，我们就没办法从任何其他单元文件来存取它的保护区成员了，例如我们写了以下的源码:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Obj: TTest;
begin
    Obj := TTest.Create;
    Obj.PublicData := 10;
    Obj.FProtectedData := 20; // won't compile
    Show (Obj.GetValue);
    Obj.Free;
end;
```

编译器会直接提出以下的错误讯息:未经宣告的识别符号: “FProtectedData”。在此时我们或许会想到没有办法存取宣告在其他单元文件的类别的保护区数据。然而，还是有方法可以处理的。想想，如果我们建立了一个显然没有用途的衍生类别，像是:

```
type
    TTestAccess = class (TTest);
```

现在，在宣告它的同一个单元文件里面，我们就可以呼叫 TTestAccess 类别的所有保护区方法了。事实上我们可以呼叫同一单元文件里面所有类别的保护区方法。

但是，这样怎么帮助我们使用 TTest 类别的对象呢?想象一下，两个共享完全相同内存架构的类别，我们可以强迫编译器把一个类别的对象当做另一个类别的，通常可以透过不安全的型别转换来达成：

```
procedure TForm1.Button2Click(Sender: TObject);
var
    Obj: TTest;
begin
    Obj := TTest.Create;
    Obj.PublicData := 10;
    TTestAccess (Obj).FProtectedData := 20; // compiles!
    Show (Obj.GetValue);
    Obj.Free;
end;
```

这段程序可以编译，运作也正常，就像我们可以看到 Protection 范例项目执行一样。重申一次，原因是 TTestAccess 类别自动继承了 TTest 基础类别的保护区字段，且因为 TTestAccess 类别也在跟想要存取这些数据的源码位于同一个单元文件里面，所以保护区的数据是可以被存取的。

现在，我们来探讨为什么这可以行得通?我得先警告读者，这个可是违反类别保护机制的，这个方法可能会让你的程序发生错误(存取不该存取的数据)，且还遵循着 OOP 记录的规则执行着。然而有些时候，透过这个技术是最好的解决方法，就像您可以在阅读函式库的原始码时，发现很多组件的原始程序就是这样做的。

总体来说，这个技术更像是在骇入整个类别系统，且不管何时都应该尽量避免，但它可以跟其他任何效果一样，被视为是 Object Pascal 编程语言的规格，且在任何操作系统、任何版本的 Object Pascal 都有效。

从继承到多型

如同字面的意义，继承是个好技术，透过它，我们可以避免源码重复，并在许多不同的类别之间分享方法。然而，其内涵的力量来自让不同类别的对象使用同一规则的能力，像是常被在 OOP 语言里面以"多型"或者"延迟绑定"这两个名词界定的意思。

我们需要先探索一些技术：衍生类别之间的型别兼容性，虚拟方法，以及在接下来这一节里面要介绍的。

继承与型别兼容性

如同我们已经看过的一些主题，Object Pascal 是一个强型别的语言，举个实例，这表示我们不能把一个整数型别的值指派给一个布尔型别的变数。至少不能在没有经过明确写出要求型别转换的情形下是如此的。基本规则是，两个数据只有在属于相同型别时，才具有型别兼容性，更精确一点来说，他们所属的数据型别的名称必须完全相同，且定义在同一个单元文件里面才算。

这个规则里有一个重要的例外，就是类别型别。如果我们宣告了一个类别，姑且说它叫做 TAnimal，然后用它衍生出新的类别，就叫 TDog 好了，我们可以把 TDog 类别的对象指派给 TAnimal。这是因为 Dog 是 Animal 的一种(*狗也是一种动物*)。从编程语言的角度或许您会觉得讶异，以下呼叫建构函式的语法都是合法的：

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
   MyAnimal1 := TAnimal.Create;
   MyAnimal2 := TDog.Create;
```

用更精确的字眼来说，我们可以把子类别的对象当做父类别的变量来使用。然而反向的使用就不行了，我们不能把父类别的对象当成子类别的变量来使用，以下就是这个理论的源码体现：

```
MyAnimal := MyDog; // This is OK
```

```
MyDog := MyAnimal; // This is an error!!!
```

事实上，我们当然可以说狗是一种动物，但不能说动物一定就是狗吧。这在大多数时候说的通，但不是定律。这相当合逻辑，且编程语言的型别兼容性也遵循这个逻辑。

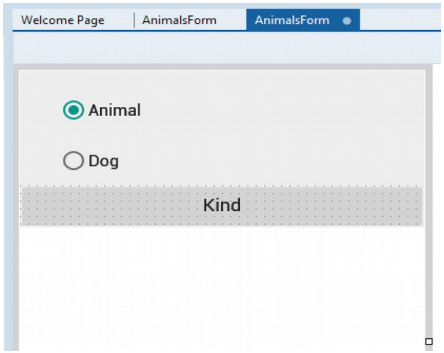
在我们仔细深入 Object Pascal 关于这个重要功能的意涵之前，请先试着执行 Animals1 范例项目，在这个项目里面定义了两个简单的类别，TAnimal 跟 TDog，TDog 是从 TAnimal 衍生而来的:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    FKind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

这两个建构函式 Create 都只是简单的设定 FKind 的值，这个值会由 GetKind 函式回传。

图 8.2: Animals1 范例项目的窗体在开发环境内的样子



这个范例的窗体，请参考图 8.2，有两个单选按钮(放在一个 Panel 上面)，可以选择属于何种类别的对象。这个对象储存在私有区，名为 FMyAnimal，属于 TAnimal 类别。这个类别的实体会在窗体每次建立或重建的时候依照

单选按钮的选择被建立且进行初始化(在此我们只显示第二个单选按钮对应的源码)：

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
    FMyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RadioButton2Change(Sender: TObject);
begin
    MyAnimal.Free;
    MyAnimal := TDog.Create;
end;
```

最后，Kind 按钮会呼叫目前这个 animal 的 GetKind 方法来把结果显示在窗体下方的 Memo 里面：

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
    Show(FMyAnimal.GetKind);
end;
```

延迟绑定与多型

Object Pascal 函式跟程序通常都是静态绑定，或者也称为早期绑定。这表示一个方法在程序撰写时，编译器或链接程序就已经进行解析，且源码会在编译结果的档案中，被置换成该程序呼叫所要对应到的内存位置。(这内存位置也被称为函式地址)。面向对象编程语言则同时允许另一种绑定，称之为延迟绑定，或者动态绑定。在这种情形下，要被呼叫的函式或方法的实际地址，得等到运行时间，由用来呼叫该函式的对象实体的型别来决定。

这个技术的好处也被称之为*多型*。多型意味着我们可以撰写程序来呼叫一个方法，当然是得从一个对象变量来发起，但 Delphi 会依照该对象的实际类别来决定要使用哪个方法。Delphi 在运行时间确定对象变量的实际型别之前，无法决定要使用哪个函式或方法，就因为前一节我们讨论过的型别兼容性。

笔记 Object Pascal 的方法预设是使用早期绑定式，跟 C++, C#一样。其中一个原因是因为这样比较有效率。而 Java 则不一样，Java 使用延迟绑定(并提供一些方法让编译器可以使用早期绑定来对方法进行优化)。

假设一个类别跟它的子类别(就用刚提到的 TAnimal 跟 TDog 做范例吧)，两者都定义了一个方法，这个方法就会是延迟绑定。现在我们可以用一个通用的变量，例如 FMyAnimal 来呼叫这个方法，在运行时间，这个变量可以是 TAnimal 或 TDog 型别。实际上在运行时间会用到哪个类别的方法，就得看当时该对象是属于哪个类别了。

Animals2 范例项目则延伸了 Animals1 项目来做为这个技术的范例。在新版本中，TAnimal 跟 TDog 类别都提供了一个新的方法:Voices，这个方法表示被选中的动物所发出的声音，同时以文字跟声音表示。这个方法在 TAnimal 类别中以虚拟方式定义(Virtual)，在稍后我们定义 TDog 类别时进行覆写(override)，透过 virtual 跟 override 这两个关键词:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

当然，这两个方法也需要被实作喔，以下是简单的实作源码:

```
function TAnimal.Voice: string;
begin
  Result := 'AnimalVoice';
end;

function TDog.Voice: string;
begin
  Result := 'ArfArf';
end;
```

那我们呼叫 FMyAnimal.Voice 会有什么反应呢?要看情况而定。如果 FMyAnimal 变量目前是储存 TAnimal 类别的对象，就会呼叫 TAnimal.Voice。如果储存的是 TDog 类别对象，就会呼叫 TDog.Voice。这个情形是因为该方法是虚拟的(Virtual)。

对所有 TAnimal 类别的任何子类别对象来呼叫 FMyAnimal.Voice 方法都会正常执行，即使该类别是在这个方法的源码之后才定义的，或者已经超出了它的可视范围。编译器不会需要一一知道衍生类别的族谱，就能让上面的这个函数调用兼容，编译器只需要知道被呼叫的对象所属类别的父代类别。换句话说，FMyAnimal.Voice 这个方法可以在所有 TAnimal 的子类别兼容。

这就是为什么 OOP 的编程语言在重复使用性这一点很强的关键性科技因素了。我们就算不用知道类别架构跟特定类别之间的关系，也可以直接写一段使用该类别的源码，。换句话说，该架构(或者说该程序)仍然具备可延展性，即使当我们用它写了数千行的源码。当然，有个前提条件，就是这个架构的父代类别得要仔细设计。

Animals2 项目示范了这些新类别的用法，并提供了类似前一个范例的窗体。以下的源码会在点击按钮的时候被执行，显示输出文字，并播放一些声音出来：

```
begin
  Show (FMyAnimal.Voice);
  MediaPlayer1.FileName := SoundsFolder + FMyAnimal.Voice + '.wav';
  MediaPlayer1.Play;
end;
```

笔记

这个项目使用了 MediaPlayer 组件来播放项目内嵌的两个声音档案的其中之一(声音档案是用实际的声音字符串来命名的，这个声音字符串会由 Voice 回传)。相对随机的噪音是提供给 TAnimal，而狗叫声则是提供给 TDog 来播放的。现在这段源码在 Windows 上面可以很容易执行了，只要声音档放在适当的文件夹里面，但如果在行动装置平台上开发，就需要一些额外的设定，好让这些档案能自动放到对应的文件夹去。

看一下实际的范例，看发布到行动装置平台时的文件夹是要怎么设定的。

覆写(Override)、重新定义(Redefine)、重新介绍(Reintroduce)方法

如我们刚刚看过的，在衍生类别里面要覆写一个延迟绑定的方法，我们得使用 override 这个关键词。请注意这个方法只能用在父代类别中，宣告为虚拟(virtual)的方法，但也可以被定义为动态(dynamic)，这个关键词我们稍后会再介绍。否则，如果该方法是静态方法，就无法进行延迟绑定，在子类别也不能更改源码了。

笔记

您可能还记得我们在前一章就使用了这个关键词来覆写 Destroy 这个从 TObject 继承而来的预设解构函式。

这个规则很简单: 只要一个方法被定义成静态方法，则这个方法在任何一代的子类别里面也都是静态方法，除非我们在某一代子类别里面特别去用虚拟的同名方法把它给替代掉。而定义成虚拟方法之后，这个方法在任何一代的子类别也都会是延迟绑定的，而且没有任何方法可以改变这一点，因为编译器为延迟绑定的方法建立源码的方式是不同的。

要重新定义一个静态方法，我们只需在子类别里面加入同样名称的方法，不管参数是否相同，也无须加入任何关键词。要覆写一个虚拟方法，我们则需要在子类别里面用相同名称、相同参数来定义这个方法，并且结尾要加上一个 override 关键词：

```
type
   TMyClass = class
      procedure One; virtual;
      procedure Two; // static method
   end;

   TMySubClass = class (MyClass)
      procedure One; override;
      procedure Two;
  end;
```

被重新定义的方法 Two，就不再使用延迟绑定。所以当我们透过基础类别的对象来呼叫这个方法，就会执行基础类别的方法了。(也就是说，如果该变量是指向衍生类别的对象，则透过该对象呼叫 Two 这个方法时，执行的结果跟基础类别对象执行的结果将会不同)。

有两种传统的方法可以覆写一个方法。第一是把父代类别的方法改写一个新版本，第二则是在已经存在的方法里面加入一些源码。以第二个作法实作时，可以透过使用 inherited 这个关键词来呼叫父代类别的同名方法。举例来说，我们可以这样写：

```
procedure TMySubClass.One;
begin
    // new code
    ...
    // call inherited procedure TMyClass.One
    inherited One;
end;
```

您可能会觉得有点讶异，为什么需要使用 orverride 关键词? 在其他编程语言里，当我们在子类别里面重新定义一个虚拟方法时，就已经会自动覆写原来的方法了。然而透过关键词的使用，编译器可以检查在父代类别跟子类别之间同名方法之间的名称与关系(重新定义方法时，把名称打错是很常见的错误，在其他 OOP 语言里面也一样)，检查看看该方法在父代类别是否是虚拟方法，以及其他项目。

笔记

另一个热门的 OOP 编程语言 C# 也使用相同的 override 关键词。这一点都不令人讶异，因为 C#跟 Object Pascal 的编程语言设计者是同一个人。Anders Hejlsberg 曾经写了不少文章来说明为何 override 这个关键词是设计函式库中基础可视化工具的原因，您可以从这里读读看：http://www.artima.com/intv/nonvirtual.html 更近期的语言，Apple 的 Swift 语言也是使用 override 关键词让衍生类别来修改方法的。

这个关键词的另一个优点，是如果我们在从函式库的类别中衍生任何类别时定义了静态方法，并不会发生问题，即使函式库已经用了一个新的虚拟方法来取代我们在子类别所定义的同名方法。因为我们定义的方法并没有使用 override 关键词，所以会被自动认为新的这个方法是一个新的独立方法，并不是被加入到函式库那个方法的新版本。(如果编译器认为是覆写版本的话，父代类别这样的程序修改可能会让我们在子类别中重新定义的程序无法执行)

支持多载(overloading)为这个功能蓝图添加了一些复杂度，子类别当中可以透过 overload 关键词来帮一个方法加入新的版本。如果这个方法的参数跟基础类别中同名方法的参数不同，就会很有效率的变成多载方法，如果参数相同，则会直接取代掉基础类别的同名方法，以下是个范例：

```
type
    TMyClass = class
        procedure One;
    end;
```

```
TMySubClass = class (TMyClass)
    procedure One (S: string); overload;
end;
```

请留意，这方法并不需要在基础类别中标注为 overload。然而，如果基础类别中的这个方法是个虚拟方法，编译器就会提出这样的警告讯息: *'One'这个方法会把基础类别'TMyClass'里面的虚拟类别隐藏掉*。

为了避免编译器提出这个警告讯息，并提供编译器更精确的指令，让它知道我们想要的结果，我们可以使用 reintroduce 这个关键词：

```
type
    TMyClass = class
      procedure One; virtual;
    end;
    TMySubClass = class (TMyClass)
     procedure One (S: string); reintroduce; overload;
    end;
```

我们可以在 ReintroduceTest 范例项目里面找到这个源码，并用它来做一些进阶的实验。

笔记 另一个我们常用到 reintroduce 的情形，就是我们想要为组件类别新增一个自定的建构函式 Create 之时。因为组件类别都已经有从 TComponent 这个基础类别继承而来的虚拟建构函式 Create 了。

继承和建构函式

一如我们看过的，我们可以用 inherited 这个关键词在子类别中呼叫父代类别的同名方法(当然也可以呼叫不同名称的方法)。对于建构函式也有一样的效果。在其他编程语言中，像是 C++, C#, 或者 Java，呼叫父代类别的建构函式是不用特别指名且会要求我们强制进行的(当我们必须把参数传给父代类别的建构函式之时)，而在 Object Pascal 里面，子类别并不一定要呼叫父代类别的建构函式。

然而，在大多数情形下，我们主动去呼叫父代类别的建构函式是很重要的。举个例子，以下就是这种情形，在任何一个组件类别中，当组件的初始化在 TComponent 类别层次已经完成：

```
constructor TMyComponent.Create (Owner: TComponent);
begin
```

```
  inherited Create (Owner);
  // specific code...
end;
```

这是非常重要的，因为组件的 Create 是虚拟方法。跟所有的类别相似，解构函式 Destroy 也是个虚拟方法，我们必须记得透过 inherited 来呼叫父代类别的同名方法。

还剩下一个问题：如果我们正在建立一个类别，它只继承了 TObject，在它的建构函式里面，我们还需要呼叫 TObject 的建构函式 Create 吗?从技术观点来看，是不需要的，假如该建构函式是空的，并没做什么事。然而，从养成好习惯的观点来看，无论如何，永远要记得呼叫父代类别的建构函式。然而如果您对效能有很高的要求，我得承认，这会拖慢速度，虽然影响层面只有小小的几个毫秒(microsecond, 不是 minisecond 耶)。

打屁结束，有几个好理由让我们使用以上两个技术，但特别是对入门者来说，我会建议一定要呼叫父代类别的建构函式，要养成这个好习惯。大家一起来推广安全的程序写法。

虚拟与动态方法

在 Object Pascal 里面，有两个方法可以启动延迟绑定。我们可以把一个方法宣告为虚拟方法，透过 virtual 关键词，或者也可以用我们在前面的篇幅介绍过的，把它宣告成动态方法，透过 dynamic 关键词。这两个关键词的用法都一样，直接把关键词放在方法宣告的最后面，它们的效果也一样。唯一的不同，是编译器在实作延迟绑定时的内部机制不同。

虚拟方法是以虚拟方法列表(Virtual method table，也可以简写为 vtable)来实作的。虚拟方法列表是一个用来储存方法地址的数组。要呼叫虚拟方法时，编译器会建立一个源码，把程序执行点跳到该对象的第 N 个虚拟方法列表的纪录去。

虚拟方法列表允许方法被快速呼叫。这个作法主要是把每个子类别所有虚拟方法的进入点收集起来，即使该方法并没有在子类别里面被覆写也一样。这样一来，虚拟方法列表里面就可以在整个架构中把每个子类别的虚拟方法快速的传播(即使对于没有被重新定义的方法也一样)。这方法可能会使用许多的内存，就只为了储存同一个方法的内存进入点地址。

另一方面，呼叫动态方法则是使用每个方法的唯一代号来进行识别与派发。搜寻特定函式通常是比单次从列表中寻找会花上更多的时间。动态方法的优点，是只会在子类别覆写了方法的时候，才会进行方法进入点的传播。对于庞大且多层的对象架构来说，使用动态方法，而不使用虚拟方法，可以省下很可观的内存，但速度上的落后并不严重。

从程序人员的观点来看，这两种技术的差异只在内部的表示法，以及不同的内存与速度而已，排除这几个差异，虚拟方法跟动态方法几乎是完全相同的技术。

现在我们已经说明了这两种模型的差异，让我们强调一下，在许多的案例中，应用程序开发人员大多使用虚拟方法，而非动态方法，这点也是不可不察的。

在 Windows 系统中的讯息处理程序

当我们在建立 Windows 应用程序时，有一个特殊要求的方法可以用来处理 Windows 系统讯息。为了这个要求，Object Pascal 提供了另一个关键词:message 用来定义讯息处理的方法，这个方法必须是带有一个适当型别的 var 参数的程序。Message 这个关键词之后必须跟着一个 Windows 讯息代号，也就是这个程序要处理的讯息代号。例如以下的源码，会让我们处理一个用户自定的讯息，讯息代号是以 Windows 的系统常数:wm_User 来定义的：

```
type
 TForm1 = class(TForm)
    ...
    procedure WmUser (var Msg: TMessage); message wm_User;
  end;
```

程序的名称跟实际的参数型别我们可以自己决定，只要实际的数据结构跟 Windows 讯息的结构相同即可。这个单元文件使用了 Windows API 来引入一些已经位多种 Windows 讯息定义好的记录型别。这个技术对于 Windows 应用程序的熟手尤其有用，我们都知道 Windows 讯息跟 API 函式，但这些跟其他操作系统真的无法兼容(像是 iOS, OSX 跟 Android)。

把方法跟类别抽象化

当我们建立一整个架构的类别时，通常很难决定用哪个类别当成基础类别，假如这个类别不会实际标示进入点，但只会用来表示一些共享的规则。我们之前介绍过的范例就是一个很好的例子，以 Animal 来做为基础类别，让 cat, dog 来做为子类别。我们并不期待为这样的类别建立对象，只让它用以表示一些特性，因此把它称为抽象类，因为这个类别不需要完整的实作源码。抽象类可以拥有抽象方法，这些方法也不用实作源码。

抽象方法(Abstract Methods)

关键词 abstract 是用来宣告只会在子类别当中定义的虚拟方法。透过这个关键词，已经完整定义了方法，这并不是预先宣告。如果我们企图为这个方法提供实质源码，编译器可是会发出警告的。

在 Object Pascal 里面，我们可以为拥有抽象方法的类别建立实体。然而当我们试着这样做，编译器还是会发出警告讯息的:"*企图为拥有抽象方法的<类别名称>建立实体*"。如果我们刚好在运行时间呼叫了一个抽象方法，Delphi 会抛出一个运行时间例外。

笔记 C++, Java 跟其他编程语言则有更多限制，在这些语言中，我们根本不被允许建立抽象类的实体。

您可能会觉得奇怪，为什么我们会想使用抽象方法。原因是使用了多型。如果 TAnimal 类别拥有一个名为 Voice 的虚拟抽象方法，每个子类别都可以重新定义它。

好处是我们可以透过共通的对象 FMyAnimal 来指向每一个动物型别的对象，然后呼叫这个方法。如果这个方法没有在 TAnimal 类别中出现，这样的呼叫方法可是不会被编译器允许的，因为编译器会执行静态型别检查。使用共享的 FMyAnimal 对象，我们可以只透过 TAnimal 定义的方法来呼叫。

我们不能呼叫子类别当中才有提供的方法，父代类别必须至少也得宣告过要呼叫的方法才行-也就是抽象方法的一个种类。下一个范例项目，Animals3，就将示范抽象方法跟抽象呼叫错误，以下是在新版范例中这个类别的定义：

```
type
    TAnimal = class
    public
        constructor Create;
        function GetKind: string;
```

```
    function Voice: string; virtual; abstract;
  private
    FKind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;

  TCat = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;
```

最有趣的部分是 TAnimal 类别的定义，它包含了一个虚拟抽象方法: Voice。而每个子类别都覆写了这个定义，并加入了一个新的虚拟方法:Eat 也很值得一看。这两个不同的功能是指什么呢?要呼叫 Voice 函式，我们可以用之前的范例程序中一样简单的写法:

```
Show (FMyAnimal.Voice);
```

那我们要怎么呼叫 Eat 方法呢?我们没办法从 TAnimal 类别的对象来呼叫它，这个指令:

```
Show (FMyAnimal.Eat);
```

会导致编译器发出错误讯息: *无法识别的字段 (Field identifier expected)*

要解决这个问题，我们可以使用动态且安全的型别转换来 TAnimal 对象当成 TCat 或当成 TDog 对象来对待，但这样的作法相对繁复，而且容易出错：

```
  begin
    if MyAnimal is TDog then
      Show (TDog(MyAnimal).Eat)
    else if MyAnimal is TCat then
      Show (TCat(MyAnimal).Eat);
```

这段程序会在稍后的『安全的型别转换指令』那个小节里面介绍。在 TAnimal 类别里面加入虚拟方法的定义，是解决这个问题的传统方法，所有动物都会有『eat』这个方法，而 abstract 关键词的使用强化了这个选择。上面的源码看起来很杂乱，为了避免这样杂乱的源码出现，正是使用多型这个技术的用意。

最后请注意，当一个类别拥有抽象方法时，这个类别通常会被当成抽象类对待。我们也可以用 abstract 关键词来把某个类别特地标注成抽象类，如果我们觉得有必要的话(当然，抽象类并不一定要拥有抽象方法)。重申一次，在 Object Pascal 里面，这个功能不是要阻止我们为该类别建立实体，所以这编程语言中，宣告没有用的抽象类是相当罕见的。

弥封类别(Sealed Classes)跟最终方法(Final Methods)

如同前面介绍过的，Java 预设对于延迟绑定(或说是虚拟方法)有着非常动态的功能。为了这个理由，Java 提出了一个概念，也就是无法再被继承的类别(弥封类别)，以及我们无法在子类别中加以覆写的方法(最终方法，或者称为非虚拟方法)。

我们无法再从弥封类别衍生出子类别。这对于我们要发布不带原始码的组件时非常需要，或者我们需要发布限制开发人员修改我们组件功能的函式库时，也是非常重要的。这功能的原始设计也是为了提升运行时间的安全性，但这些功能我们在完全编译的编程语言，像 Object Pascal 里面，就不需要。
*最终方法*是我们无法在衍生类别中加以覆写的方法。重申一次，这个概念在 Java(在 Java 里面所有方法预设都是虚拟方法，最终方法会很明显的得到优化)里面很有帮助，也被引入到 C#里面，因为 C#的虚拟函式需要精确标注，所以相对重要。因此同样的概念也被加入到跟 C#很相似的 Object Pascal 里面来了，但并不常被使用。

以下是弥封类别的范例源码，请大家留意其语法：

```
type
    TDeriv1 = class sealed (TBase)
        procedure A; override;
    end;
```

企图从这个类别衍生子类别的话，编译器会发出错误回报: "*无法延伸弥封类别 TDeriv1*"。以下则是最终方法的语法：

```
type
    TDeriv2 = class (TBase)
        procedure A; override; final;
    end;
```

从这个类别衍生子类别时，如果企图覆写 A 方法，也会得到编译器的错误回报:”无法覆写最终方法”。

安全的型别转换指令

一如稍早的篇幅所介绍的，Object Pascal 的子类别型别兼容性规则让我们可以把子类别当成父代类别来使用，然而不能把父代类别直接当成子类别来使用。

现在我们假设 TDog 类别拥有一个 Eat 方法，这个方法在 TAnimal 类别里面并没有出现过。如果 MyAnimal 变量指向了 TDog 类别的对象，那我们要怎么呼叫 Eat 方法呢？如果我们试着把变量直接转型成另一个类别，是会发生错误的。透过指令来进行的强制转型，我们可能会引发一个运行时错误(或者更糟，可能出现不预期的内存重复写入问题)，因为编译器无法判定该对象到底是哪种型别，且我们到底想呼叫哪个存在的方法。

为了解决这个问题，我们使用了运行时间型别信息(Run-Time Type Information, 简称 RTTI)。因为每个对象在运行时间一定都知道自己的型别，以及它自己的父代类别。我们就直接询问对象这项信息，可以透过 is 这个指令，或者使用 TObject 类别的一些方法。is 这个运算符号的参数是一个对象跟一个类别型别，会回传布尔值：

```
if FMyAnimal is TDog then
  ...
```

这个 is 判别式如果回传 true，就表示 FMyAnimal 对象目前是指向一个 TDog 类别的对象，或者是从与 TDog 兼容的类别衍生出的型别。这代表如果我们检查存在 TAnimal 变量里的对象是 TDog 时，回传值就会是正确的。换句话说，如果回传值是 true，我们就可以安全的把这个对象(MyAnimal)指派给 TDog 型别的变数了。

笔记

is 这个运算符号实际上的实作方式，是由 TObject 的 InheritsFrom 方法所提供的。所以我们可以用在 FMyAnimal 对象上使用同样的判别式，写成 FMyAnimal.InheritsFrom(TDog)。直接使用这个方法的原因，是因为这个方法也可以被类别参考跟其他用途的型别所使用，且这些型别有可能不支持 is 运算符号。

透过这个判断式，我们可以确定在 MyAnimal 里面的对象是 TDog 类别的对象了，所以我们可以直接用型别强制转换(通常直接进行转换会有安全疑虑，但我们已经确认了型别是相同的，所以可以这样做)，写成以下的范例源码:

```
if FMyAnimal is TDog then
begin
  MyDog := TDog (FMyAnimal);
  Text := MyDog.Eat;
end;
```

相同的动作也可以直接用另一个跟型别转换相关的运算符号 as 来达成。这个运算符号可以把对象进行转换，但只会在要求转换的对象兼容于想转换成为的类别时才转换，否则就会发生运行时间例外而出现错误。As 运算符号的要求的参数是一个对象，以及要转换成为的类别，回传的值则是转换成新型别之后的对象。我们可以简单的写成这样的源码：

```
MyDog := FMyAnimal as TDog;
Text := MyDog.Eat;
```

如果我们只想呼叫 Eat 函式，我们也可以使用一个简单的写法:

```
(FMyAnimal as TDog).Eat;
```

这个表达式的回传结果，会是一个 TDog 型别的对象，所以您可以透过这个对象来使用 TDog 类别的任何方法。在传统的型别转换跟 as 所呈现的型别转换之间的不同，是透过 as 进行型别转换的时候，会先进行该对象的型别兼容性检查，如果要转换的型别跟该对象不兼容，系统就会发出例外警告，这个例外是 EInvalidCast(我们在下一章里面会介绍)。

警告

对照一下，在 C#里面的 as 表达式在型别不兼容的时候，会传回 nil，如果进行直接强制转型，发生不兼容的时候，则会产生例外事件。所以 C#跟 Object Pascal 的 as 表达式在回传值上面刚好是相反的。

为了避免这个例外，请先使用 is 运算符号进行判别，如果型别是兼容的，就可以直接做强制转型(事实上好像没有必要把 is 跟 as 依序使用，因为这么一来相同的型别测试会做两次，所以通常不是使用 is，就是使用 as，很少两个并起来一起用的):

```
if FMyAnimal is TDog then
  TDog(FMyAnimal).Eat;
```

两种型别转换的运算符号在 Object Pascal 里面都很有用，因为我们通常会想要撰写通用的源码，可以套用在不同的组件中，来进行相同型别检查，甚至是不同型别之间的检查。举例来说，当一个组件被当做参数传递到一个处理事件的方法时，会是使用通用的数据类别(TObject)来传的。所以我们常常要先把型别转换回该组件的真正型别：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if Sender is TButton then
       ...
end;
```

这是一个很常见的技术，我也在很多之后的范例里面都会使用(事件会在第十章里面介绍到)。

这两个型别转换的运算符号，is 跟 as，功能是非常强大的，我们也可能会想要考虑在基本的程序撰写中用到它们。就因为它们真的很强大，所以在使用的时候要记得适当的在特定情形下规范使用的方法。当我们需要引入多个类别来解决复杂的问题时，记得先使用多型的技术。只有在特别的情形下，多型才无法被单独使用，此时我们就只能试着透过型别转换运算符号来解决它了。

笔记 型别转换运算符号在效能上会有明显的负面影响，因为它得把整个类别的族谱跑一次，才能确实判别型别转换是否合法。跟我们介绍过的技术对照一下，虚拟方法的呼叫也需要进行内存搜寻，但相对快的多了。

可视化窗体继承

继承这个技术并不只用在函式库的类别，或者我们自己写的类别，但在整个以 Object Pascal 为中心的 IDE 环境里面，这个技术是无所不在的。一如我们所见的，当我们在 IDE 里面建立一个窗体的时候，我们其实是建立了一个 TForm 类别的实体。所以任何可视化应用程序都是建立在继承这个技术之上的。因此我们在写程序的时候，大多是在写一些简单的事件处理程序。

然而，对于比较有经验的开发人员来说，还比较不知道的作法，是我们可以从我们已经写好的窗体再衍生出另一个窗体，这个功能通常被称为视觉窗体继承(这也是 Object Pascal 开发环境中相当奇特的一部分)。

这个功能有趣的地方，是你可以直接看到继承的威力，而且可以直接了解他的规则。这既有用又实用吧?也是，这大多时候会跟我们正在建立的应用程序类型有关。如果应用程序里面有多个窗体，其中部分窗体可能很相似(外表很相似、功能也很相似)，这样一来我们就可以把这些相似的窗体做成一个基础窗体，事件处理程序也放在基础窗体里面，然后每个确切的窗体再从这个基础窗体衍生出来制作。另一个类似的情景，是使用可视化窗体继承的技术来为特定客户进行客制窗体，但不复制任何源码(这也是第一时间使用继承这个技术的核心理由)。

我们也可以使用可视化窗体继承来为同一个应用程序在不同操作系统的外观进行客制化跟窗体的元素(例如从手机到平板)，不复制任何源码，或者窗体的定义，只从标准的窗体为客户衍生出一个特定的版本。

请记得，视觉继承的主要优点，是我们可以稍后再修改原始窗体，且自动更新所有的衍生窗体。这是在 OOP 语言里面广为人知，关于继承的优点。但也有一个正面的副作用：多型。您可以为基础窗体加入一个虚拟方法，然后在衍生的窗体之中把它覆写。然后我们可以引入两个窗体，然后呼叫每个窗体的这个方法。

笔记

以相同元素建立窗体的另一个方法是透过 frame。这作法是把一些原本放在窗体上的视觉组件改为放在 frame 上面。在设计时间，我们可以同时处理两个版本的窗体。然而在视觉窗体继承中，我们是定义了两个不同的类别，一个是父代类别，另一个是衍生类别。但使用 frame 十，我们则是新制作了一个 TFrame 的衍生类别，以及用来承载 TFrame 的窗体类别。

从基础窗体进行继承

一旦我们对继承的内涵有了清楚的认知，管理视觉窗体继承的规则其实就相当简单。基本上，衍生的子窗体跟父代表单会有同样的视觉组件。我们不能移除父代表单上面的组件，不过我们可以把该组件(如果该组件是个视觉组件)设定成隐形。重要的是我们可以很容易更改继承后的组件属性。

请注意，如果我们更改了衍生窗体上组件的属性，之前在父代表单中对相同属性的修改，在衍生窗体中都将失去作用。变更组件的其他属性，则仍会影

响衍生窗体中各组件的视觉表现。我们可以透过使用对象查看器上面的 Revert to Inherited 这个选单指令，让两个属性的内容重新一致化。这样的效果也可以透过手动把两个属性的内容修改成一致之后，再重新编译器来达成。改变多个属性之后，我们仍然可以透过 Revert to Inherited 这个选单指令来把被变更的值回复成跟父代表单中的属性设定相同。

除了继承组件，新的窗体也继承了基础窗体的所有方法，包含事件处理程序。我们可以在衍生窗体中新增，也可以覆写已经存在的事件处理程序。

为了示范视觉窗体继承的效果，我建立了一个简单的范例，名为 VisualInheritTest。我会逐步介绍如何编译这个项目。首先，建立一个新的多重装置应用程序项目、选择空白项目，然后在主窗体中加入两个按钮。然后选择 File->New->Others，接着在新增项目(New Items)的对话窗口中选择”Inheritable Items”分页(请参考图 8.3)。以下是我们可以从窗体来选择想要继承的项目。

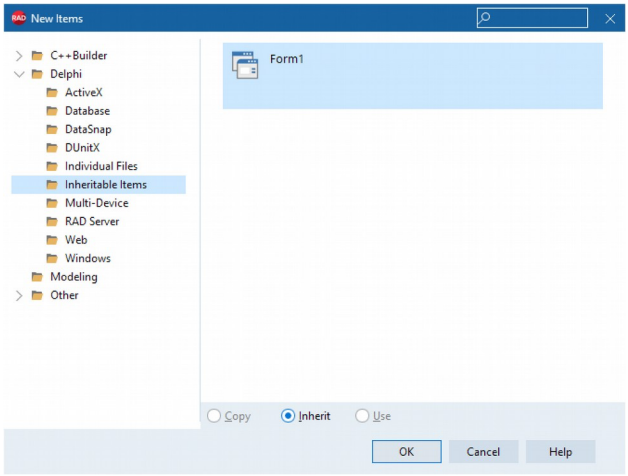


图 8.3: New 项目对话窗口让我们可以建立一个衍生窗体

新的窗体会同样有两个按钮，以下是该窗体最开头的文字描述：

```
inherited Form2: TForm2
  Caption = ‘Form2’
  ...
end
```

且以下是该类别刚建立时的宣告内容，我们可以看到其基础类别不是我们常见的 TForm，而是一个实际的基础类别窗体了：

```
type
 TForm2 = class(TForm1)
```

```
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

请注意 inherited 关键词在文字描述的使用，还有窗体实际上是拥有其他组件的，只是它们是被定义在基础类别里面而已。如果我们改掉了其中一个按钮的文字，并且加上一个新的按钮，窗体的文字描述就会变成这样：

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
  inherited Button1: TButton
    Text = 'Hide Form'
  end
  object Button3: TButton
      ...
      Text = 'New Button'
      OnClick = Button3Click
  end
end
```

只有属性的值不同的项目被列出了，因为其他的项目都是从基础窗体直接继承而来的。

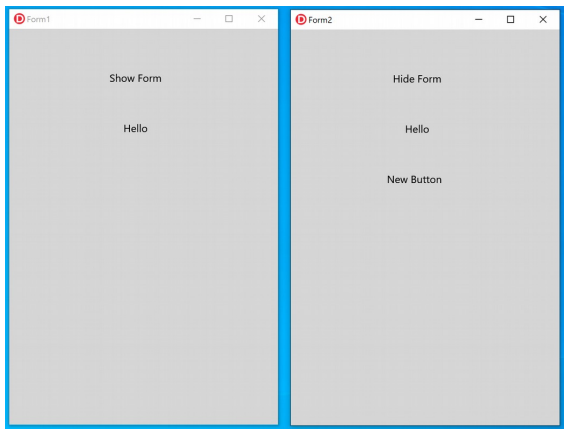


图 8.4: VirtualInheritTest 范例在运行时间的两个窗体

第一个窗体中的每一个按钮都有 onClick 事件处理程序，请参考范例程序。第一个按钮被点击时，会呼叫 Show 方法来显示出第二个窗体，第二个按钮被点击时则是显示简单的讯息。

在衍生窗体中发生了什么事？我们首先要改变 Show 按钮的规则，让它变成一个隐藏按钮。这样就不会执行基础窗体的事件处理程序了(所以我也把预设呼叫的 inherited 源码给批注掉了)。相反地，我在 Hello 按钮上加入了第二个讯息给基础类别建立出来的窗体，这个函式只留下了 inherited 呼叫：

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  // inherited;
  Hide;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    inherited;
    ShowMessage ('Hello from Form2');
end;
```

记得这跟衍生方法的差异，衍生方法可以使用 inherited 关键词来呼叫衍生类别中的同名方法，在事件处理程序里面使用 inherited 关键词则会呼叫基础类别中对应的同名方法。

当然，我们也该把基础窗体中的每个方法视为子窗体的方法，并且视需要自由呼叫它们。这个范例允许我们了解一些可视化窗体继承的功能，但要真正看到它的威力，我们需要多看一些复杂的，真实世界的范例，而不要被绑死在书上喔。

09:例外处理

在我们开始继续介绍 Object Pascal 类别的其他功能之前，我们需要先聚焦在处理错误条件的特定对象群组上，这个错误条件称之为例外。

例外处理的含意，是要透过加入一致性、简单的处理软件或硬件错误的能力，让程序更为自动化。一个程序可能在这样的错误发生后还继续执行，也可能安全的结束掉，让使用者能够在结束之前储存一些数据。例外事件允许我们把错误处理的源码从一般源码分离开来。这样我们就不用在源码之间交错一般程序功能跟错误处理的源码了。这样我们的源码可以更精简、不凌乱，把跟实际程序目标不相关的源码拢在一起。

另一个好处，是『例外』定义了一个统一、放诸四海阶准的错误回报机制，这个机制也同时用在组件函式库里面。在运行时间中，系统会在出错的时候发出例外事件。如果我们的源码已经适当处理了例外事件，我们的程序就会被通知，也会试着解决这个错误。

反之，这个例外会被传递到它所呼叫的源码，一层一层扩散。最终，如果我们的源码里面没有处理这个例外，系统通常会处理它，轮到系统处理时，画面上就会显示一个标准的错误讯息，并且试着继续执行程序。在大多数的情形下，我们的程序都是在例外处理区外面运作的，发出了例外讯息会使程序终止。

在整个 Object Pascal 的例外处理机制当中，是建构在五个关键词之上的:

- try 注册一个源码保护区的开始
- except 注册这个源码保护区的结束，并宣告开始进入例外处理的源码
- on 标示每个特定的例外处理叙述句，与特定的例外进行连结，每个 on 的叙述句语法都是 on 例外型别 do 叙述句
- finally 是用来标示不论如何都要被执行的源码，即使例外发生也一样。
- Raise 是用来触发例外的叙述句，它的参数则是一个例外类别的对象(在其他语言中，使用的语法则是 throw)

以下是一个简单的比较表，用来比较 Object Pascal 跟其他以 C++语法为基础的编程语言(像是 C#跟 Java)处理例外的语法:

try try

except on	catch
finally	finally
raise	throw

用 C++的说法，我们可以抛出一个例外对象，再依照例外所属型别来接住它。在 Object Pascal 里面也是一样的，我们可以透过 raise 指令传出一个例外对象，并在 except on 语法中以参数形式接收该对象。

Try-Except 区块

让我们从一个相对简单的 try-except 例子开始(这是 ExceptionsTest 范例项目的一部分)，以下则是一个通用的例外处理区块：

```
function DividePlusOne (A, B: Integer): Integer;
begin
    try
      // raises exception if B equals 0
      Result := A div B;
      Inc (Result);
    except
      Result := 0;
    end;
    //More code
end;
```

笔记 当我们在 Delphi 除错模式下执行一个程序时，除错程序会默认在例外发生的时候停下程序，即使我们已经在程序中写好了异常处理程序也一样。这也是我们通常乐见的，当然，因为我们想要知道例外会在什么时候发生，然后才能一行行的看着源码执行。如果我们想让程序在例外发生时适当的处理它，然后看看一般使用者会看到的画面，我们可以用 IDE 里面的”以非侦错模式执行”的指令来执行程序。

并不是说，『掩盖掉』例外事件，然后把回传值设定为 0 这样是对的(因为把使用者的问题遮盖掉通常不是好事)，但这个源码是用来帮助我们用简单的情况理解核心机制的：

```
var
    N: Integer;
```

```
begin
    N := DividePlusOne (10, Random(3));
    Show (N.ToString);
```

就像我们看到的，源码使用了一个随机产生的数值，因此当我们点击按钮的时候，我们可以处于合法的情形下(2 乘以 3 以外的值)或者处于错误的情形下。以下是可能发生的两种不同程序流程：

- 如果 B 不是 0，程序会进行除法计算，然后把结果加一，最后忽略例外处理区块。
- 如果 B 是 0，程序执行除法的时候会发生例外，在发生例外之后的其他源码会自动被跳过(在这个范例里面只有一行程序会被跳过)，直接跳到 try-except 区块的第一行继续执行。在进入例外程序区之后，程序就不会再回到原来的源码了，但会接着再把例外程序区里面的所有源码都执行完毕。

我们可以用一个方法来描述这个例外模型，就是它遵循的工作目标是不会再回头。在发生错误的情形下，试着处理错误情况，并回到导致这个错误发生的源码，是很危险的作法。在这个情形下，程序可能已经进入了未预期的状况。例外很明确的改变了程序执行的流程，跳过了接下来的源码，回复到错误未发生前的状态，直到适当的错误处理源码出现。

上面的源码包含了一个很简单的例外程序区，但没有包含 on 这个叙述句。当我们需要处理多种不同的例外情形时(或者多种例外类别型别)，或者我们想要存取被触发、传递到例外程序区的例外对象，我们就需要有一个或多个的叙述句：

```
function DividePlusOneBis (A, B: Integer): Integer;
begin
  try
     Result := A div B; // error if B equals 0
     Result := Result + 1;
   except
       on E: EDivByZero do
      begin
          Result := 0;
          ShowMessage (E.Message);
       end;
  end;
end;
```

在例外处理的叙述句中，我们拦截到 EDivByZero 例外，这是由 RTL(运行时间函式库)所定义的。在 RTL 里面定义了许多这样的例外型别(像这里所介绍的除以 0 的例外，或者错误的动态型别切换都是其中之一)，也有一部分例外型别是由系统所定义(例如内存不足的例外)，或者组件错误(像是索引值有误)。所有这些例外的类别都是从例外的基础类别 Exception 衍生而来的，在这里面提供了最低限度的功能，例如 Message 属性，在上面的范例程序中我们就用到了这个属性。这些类别也依循一些逻辑结构组成了一个不小的类别架构。

笔记 请注意，在 Object Pascal 里面一般只要是类别都会用 T 这个字母开头来命名，但例外类别是这个规则中的例外，它们是以 E 这个字母开头来命名的。

例外类别架构

以下的列表是 RTL 核心例外类别里面的一部分，这些例外类别都定义在 SysUtils 这个单元文件里面(大多数其他的系统函式库还另外会在新增自己的例外型别):

```
Exception
        EArgumentException
          EArgumentOutOfRangeException
          EArgumentNilException
        EPathTooLongException
        ENotSupportedException
        EDirectoryNotFoundException
        EFileNotFoundException
        EPathNotFoundException
        EListError
        EInvalidOpException
        ENoConstructException
        EAbort
        EHeapException
          EOutOfMemory
          EInvalidPointer
        EInOutError
        EExternal
          EExternalException
          EIntError
```

```
            EDivByZero
            ERangeError
            EIntOverflow
          EMathError
            EInvalidOp
            EZeroDivide
        EOverflow
        EUnderflow
    EAccessViolation
    EPrivilege
    EControlC
    EQuit
EInvalidCast
EConvertError
ECodesetConversion
EVariantError
EPropReadOnly
EPropWriteOnly
EAssertionFailed
EAbstractError
EIntfCastError
EInvalidContainer
EInvalidInsert
EPackageError
ECFError
EOSError
ESafecallException
EMonitor
    EMonitorLockException
    ENoMonitorSupportException
EProgrammerNotFound
ENotImplemented
EObjectDisposed
EJNIException
```

笔记 我不知道大多数人会怎么做，但我自己仍然会先把实际的使用情境中可能遇到的奇怪例外做个设想，像是 EProgrammerNotFound 这样的例外。在 Delphi 里面有一些隐藏彩蛋，这是其中一个。

现在我们已经介绍过了核心的例外类别架构，我们可以在先前的 except-on 叙述句上面多加一点信息了。这些叙述句会一一被比对，直到系统找到跟目前发生的例外相符的例外类别。目前所使用的比对规则是检查其型别兼容性，这个主题我们在前一章已经介绍过了：例外对象会跟其所属的类别以及所有父代类别兼容(就像 TDog 对象会兼容于 TAnimal 类别)。

这表示我们可以提供多个例外处理型别是跟目前发生的例外相符的。如果我们想要使用更精确相符的规则来处理该例外的话(在类别架构中比较末端的例外类别)，我们就需要更精确的列出要处理的例外类别(列的越末端越能精确命中特定的例外)。当然我们也可以写一个异常处理程序，直接使用 Exception 这个型别，这样所有的例外发生的时候都会由它来处理，所以这个区块也得放在整个 Exception 处理区的最后喔。以下就是在一个区块中包含两个处理程序的程序范例：

```
function DividePlusOne (A, B: Integer): Integer;
begin
    try
      Result := A div B; // Error if B equals 0 Result := Result + 1;
    except
      on EDivByZero do
      begin
          Result := 0;  MessageDlg ('Divide by zero error',
           mtError, [mbOK], 0);
      end;
      on E: Exception do
      begin
          Result := 0; MessageDlg (E.Message,
           mtError, [mbOK], 0);
      end;
   end; // End of except block
 end;
```

在上面这段源码里面，同一个 try 区块当中，包含有两个不同的异常处理程序。我们可以在同一个 except 区块当中写入任意数量的异常处理程序，在例外发生的时候，它们会被系统一一比对，就像前文所述。

也请记得，为每一种可能发生的例外都写一个处理程序通常并不是好主意。最好还是把一些我们不知道的例外留给系统。预设的异常处理程序通常会把该例外的类别名称用讯息窗口加以显示，然后回复该程序的正常运作。

提示 我们也可以透过实作 Application.OnException 这个事件的处理程序来把一般异常处理程序替换掉。例如把例外的讯息存到档案里面，而不要显示给用户看。

触发例外

我们在 Object Pascal 里面会遇到的绝大多数的例外，都是由系统所建立的，但当我们发现在运行时间当中有不合法或者不连续的资料时，我们也可以在我们自己的源码里面触发例外。

在大多数的情形下，我们需要先定义我们自己的例外型别，好让我们需要触发时可以用到。我们可以用以下的源码简单的建立一个新的例外型别，只需要从预设的例外基础型别 Exception 衍生一个即可：

```
type
  EArrayFull = class (Exception);
```

在绝大多数的案例中，我们不用在新的例外类别中加入任何方法，只需纯粹宣告一个新的例外类别即可。

用到这个例外型别的情境，可能会是在一个方法试着在数组中加入元素时，当数组已经满了，而触发这个例外。实际上的写法只需建立例外对象，然后把它用 raise 关键词传递即可：

```
if MyArray.Full then
 raise EArrayFull.Create ('Array full');
```

这个 Create 方法(从 Exception 类别继承而来的)要求一个字符串作为参数，来描述该例外的情形让使用者知道。

笔记 我们不用担心这个例外对象要怎么被释放，因为它会自动被例外处理机制给删除掉。

Raise 关键词还有一种情形下会使用到。就是在我们自己写的 except 区块当中，当我们所写的源码并没有真的抓到该例外，这时候该例外处理区块里面的源码应该都不会派上用场，此时我们就应该把这个例外再次触发，让系统

去处理，这时我们只需要执行 raise，不用参数了。这个动作我们称为*再次触发*例外。

例外与堆栈

当程序触发了一个例外，而目前的源码无法处理它，那么在我们方法或函式堆栈里面会发生什么事呢？程序会在已经触发的所有函式或方法堆栈当中一层一层的搜寻可以处理这个例外的处理程序。这表示源码会从当时正在执行的源码里面跳脱，不会再继续执行剩下的源码。要了解其中的工作原理，我们可以用侦错程序或者在目前执行的源码里面加一些简单的输出值。在下一个范例项目，ExceptionFlow 里面，我选择以第二种方式来做示范。

举例来说，当我们按下 ExceptionFlow 项目里的窗体上面的 Raise1 按钮，就会触发一个没有被处理的例外，所以源码的最后一部分并不会被执行到：

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
    // unguarded call
   AddToArray (24);
   Show ('Program never gets here');
end;
```

请注意，这个方法呼叫了 AddToArray 程序，这个程序将会触发例外。当这个例外有被处理时，程序的流程会从该处理程序之后继续下去，而不是从例外发生的源码继续下去。以下面这个方法来看:

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
    try
        // This procedure raises an exception
        AddToArray (24);
        Show ('Program never gets here');
    except
        on EArrayFull do
            Show ('Handle the exception');
    end;
    Show ('ButtonRaise1Click call completed');
end;
```

最后一行的 Show 会在例外被处理完毕以后继续执行下去，第一个 Show 则永远不会有被执行到的机会。我建议在执行这个程序之前，先把源码做些修改，做些对应的实验，好帮助您能够理解例外被触发时，程序流程是怎么运作的。

笔记

如果我们写来准备处理例外的源码跟该例外发生的位置不同，如果能够确切知道该例外是发生在哪个方法会是很有帮助的。虽然的确有方法可以在例外被触发的时候，取得函数调用堆栈的信息，但这是比较进阶的主题，我并不打算在此介绍。在绝大多数的案例中，Object Pascal 的开发者会依靠第三方组件跟函式库(像是 Jedi 组件函式库里面的 JclDebug、MadExcept，或者 EurekaLog)来取得这个信息。此外，我们也得先让编译器产生出 MAP 档，并把它引用到我们的项目里面，这个档案会列出我们应用程序中每个方法与函式的内存地址。

Finally 区块

例外处理还有第四个关键词，我们之前提到过，但还没有使用到，它就是 finally。Finally 区块是用来定义一些一定要被执行到源码(通常是用来做清理的程序)。事实上，在 finally 区块里面所撰写的源码，不管是否有例外事件被触发，都一定会被执行到。而一般的源码则是写在 try 跟 finally 之间，只有在没发生例外的时候或者发生了例外且被妥善处理了，这些源码才会被执行。换句话说，不管有没有发生例外事件，finally 区块的源码都会被执行到。

以下面这个方法为例(它是 ExceptFinally 范例项目的一部分)，它会执行一些耗时间的指令，并在窗体的标题上显示它当时的状态：

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
    I, J: Integer;
begin
    Caption := 'Calculating';

    J := 0;  // Long (and wrong) computation...
    for I := 1000 downto 0 do
        J := J + J div I;

    Caption := 'Finished';
```

```
    Show ('Total: ' + J.ToString);
 end;
```

因为在这个算法里面有一个错误(在我们可以改动的变量值当中，包含了 0，所以会发生除法除以 0 的错误)，程序会被中断，但并不会重设窗体的标题。这种情形就需要 try-finally 区块来解决了：

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
    I, J: Integer;
begin
    Caption := 'Calculating';
    J := 0;
    try
       // Long (and wrong) computation...
       for I := 1000 downto 0 do
          J := J + J div I;
       Show ('Total: ' + J.ToString);
    finally
       Caption := 'Finished';
    end;
end;
```

当这个函式被执行的时候，它永远都会记得把状态给重设，不管是否有发生任何一种例外。这个版本的函式唯一的缺点，应该就是它没有真的处理例外事件吧。

Finally 跟例外

够奇怪的吧，在 Object Pascal 里面，try 区块后面只能跟着 except 或者 finally 区块，但却不能两者一起出现。假设我们想同时处理两种区块，通常只能把两种 try 区块互相包含着使用，在里面那层 try 区块里面使用 finally，而在外层的 try 区块使用 except，或者反过来用，要视当时的情况而定。以下是 ExceptFinally 范例中的第三个按钮的事件处理程序:

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
var
    I, J: Integer;
begin
    Caption := 'Calculating';
```

```
    J := 0;
    try
        try
            // Long (and wrong) computation...
            for I := 1000 downto 0 do
                J := J + J div I;
            Show ('Total: ' + J.ToString);
        except
            on E: EDivByZero do
            begin
                // Re-raise the exception with a new message
                raise Exception.Create ('Error in Algorithm');
            end;
      end;
    finally
        Caption := 'Finished';
    end;
end;
```

透过 Finally 区块还原光标

Try-finally 区块很常用的一个情形，是用来处理资源的配置跟释放。另一种案例则是我们在特定作业完成后，需要把临时的设定重设，就算在这个案例中触发了例外事件，也要完成重设。

重置临时设定最容易理解的案例，就是把漏斗光标重设回箭头光标，通常我们会在可能执行很久的源码中，先把光标改成漏斗光标，等到该段程序执行完成再改回箭头光标。就算这源码再简单，也还是有发生例外事件的可能，所以我们一定都要使用 try-finally 区块把这源码包起来。

在范例 RestoreCursor 中(这是个 VCL 应用程序，要用 FireMonkey 管理光标有点复杂…)，在以下的源码里面，我把目前的光标状态先储存下来，暂时把光标改成漏斗光标，然后在执行结束时再改回执行前的光标:

```
Var CurrCur := Screen.Cursor;
Screen.Cursor := crHourGlass;
try
    // 某些执行会花比较久的源码
    Sleep(5000);
```

```
finally
    Screen.Cursor := CurrCur;
end;
```

透过受管理的纪录还原光标

要保护已配置的资源，或者还原临时暂存的设定，如果不想明确的使用 try-finally 区块，我们也可以使用受管理的纪录，因为编译器会自动在编译完成的执行码当中，为受管理的纪录加入 finally 区块。所以我们可以用这个效应，少写一些源码也能达到保护已配置的资源，或者还原暂存设定，但我们就得在定义这些记录的时候多写一些源码。

以下是一个受管理的纪录，它跟前一段的范例程序有一样的效果，会把目前的光标在 Initialize 方法中储存下来，然后在 Finalize 方法中还原:

```
type
    THourCursor = record
    private
        FCurrCur: TCursor;
    public
        class operator Initialize (out ADest: THourCursor);
        class operator Finalize (var ADest: THourCursor);
    end;
class operator THourCursor.Initialize (out ADest: THourCursor);
begin
    ADest.FCurrCur := Scree.Cursor;
    Screen.Cursor := crHourGlass;
end;
class operator THourCursor.Finalize (var ADest: THourCursor);
begin
    Screen.Cursor := ADest.FCurrCur;
end;
```

我们定义了这个受管理的纪录后，就可以把前段程序写成这样:

```
    var HC: THourglassCursor;
    // 某些执行会花比较久的源码
    Sleep(5000);
```

笔记 关于以受管理的纪录进行资源保护，您可以找到更多延伸的范例，是由 Erik van Bilsen 所发的部落格文章: https://blog.grijjy.com/2020/08/03/automate-restorable-operations-with-custom-managed-record/。这是一系列关于受管理的纪录有深入探讨的文章。

真实世界的例外

例外对于在大范围(也就是说并不是只针对单一程序，而是整个大架构的一部分)的程序设计中进行错误回报以及错误处理，是个很棒的机制。通常我们不应该完全依赖例外机制，而不仔细的进行区域错误条件检查(当然有些开发人员是喜欢这样用的)。举例来说，如果我们不确定特定档名是否可以使用，我们就应该在开启档案之前先检查一下档案是否存在，而不要依赖例外发生时才透过例外处理机制来处理档案不存在的情形。然而，如果在写档之前检查是否有足够的磁盘空间，就不是那么实用了，因为磁盘空间不足的情形并不常发生。

所以我们应该在程序撰写的时候，先自行处理常发生的问题，然后把不常发生的问题留给例外处理机制。当然常发生与不常发生之间的界线很模糊，端赖程序人员的判断，因此不同程序人员也会有不同的方法来加以界定。

我们应该持续的在不同的类别跟模块之中使用例外处理，让例外事件可以在不同的类别跟模块之间互相传递。回传错误码的作法相当乏味，且使用例外更容易比对错误情形。在组件或函式库的类别中触发例外，比在事件处理程序里面触发更为常见。

在日常的源码中越来越常见，且格外重要的是使用 finally 区块来预先保护程序的资源免于因例外发生而失控。我们应该永远在使用外部资源的源码当中使用 finally 来避免万一例外发生而导致资源发生内存泄漏。我们每当需要对档案做启闭、对网络进行联机、断线，在单一区块中建立、释放某些资源时，我们就应该派 finally 区块上场了。最后，finally 区块让我们在即使万一发生例外事件时，也能够使程序保持稳定，这样用户就可以继续使用，或者还能依序(万一有很明确的问题发生时)的关闭应用程序。

全局例外处理

如果例外是由一个事件处理程序所触发，停止了标准的执行流程，此时如果没有找到对应的例外处理程序，整个程序也会停止运作吗？在文本模式应用程序或者一些特别用途的程序结构里的确会这样。但大多数的窗口应用程序(包含使用 VCL 或 FireMonkey 架构的窗口应用程序)则有全局的讯息处理循环，会把整个源码包在一个 try-except 区块里面，所以如果有未被处理的例外被触发，就会在这里被捕捉到。

笔记

请记得，如果在程序启动的源码中，在讯息循环生效之前就已经有例外被触发了，这个例外事件就不会被程序或者任何函式接收、处理，程序就会直接因为发生错误而直接停止了。这个情况可以由我们自己在主程序里面加入 try-except 区块来解决一部分的情境。即使是使用这个等级的保护，在主程序执行前，还没有进入自定的 try-except 区块之前，函式库的初始化源码就会被触发，所以在还没进入例外控制之前，仍然有可能发生未被掌控的例外。

一般来说，执行中触发例外时要看函式库怎么处理它，在这里我们仍然可以写一个全局的异常处理程序来处理，或者显示例外事件的讯息。尽管细节处理规则可能稍有不同，但这个大原则在 VCL 跟 FireMonkey 则是都一样的。

在前一个范例中，您会在例外事件发生时看到一个简单的错误讯息显示。如果您想要改变这个规则，可以为全局的 Application 对象建立 OnException 事件的处理程序。虽然这个动作的归属更偏向视觉函式库以及应用程序的事件处理，它也还是跟例外处理相关，所以我们在这个章节也加以介绍。

我已经把前一个例子做成范例程序，名为 ErrorLog，现在我们再来它的主窗体上面加一个新的方法:

```
public
    procedure LogException (Sender: TObject; E: Exception);
```

在 OnCreate 事件处理程序中，我加入了下面这个源码，把这个方法跟全局的 OnException 事件做了绑定，在这里面我们就可以来实作全局的异常处理程序了:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Application.OnException := LogException;
end;
procedure TForm1.LogException(Sender: TObject; E: Exception);
```

```
begin
    Show('Exception ' + E.Message);
end;
```

笔记 我们会在下一章里面介绍怎么把方法的指标指派给一个事件(就像我们刚在前一个程序片段刚介绍过的)。

透过新的方法来处理全局例外事件之后，这个程序会在未被处理的例外发生时，直接显示错误讯息，不会把程序停止下来了。

例外与建构函式

在例外这个主题上还有一个明显比较进阶的问题，就是当例外发生在一个对象的建构函式执行时。大多数的 Object Pascal 程序人员都不知道在这个情形下，该对象的解构函式就会被自动呼叫。

这个作法非常重要，我们必须记住，因为这种情形表示解构函式会在对象初始化进行到一半的时候就被呼叫。假设解构函式中本来就已经预备着把内部对象释放掉，也是因为我们一开始已经假设在建构函式中已经把这些内部对象建立好了，此时就可能在解构函式中试图释放还没有建立的这些内部对象，这种情形着实是很危险的(这表示可能会在第一个例外事件还没被处理好之前，就又诱发了第二个例外事件)。

这也同时指出了在建立对象的时候，最好使用 try-finally 区块来确保安全，因为 try-finally 可以自动被编译器加以保护。所以如果建构函式执行中发生了例外，就可以不用释放该对象了。这是为什么标准的 Object Pascal 程序模式会用以下这种写法来保护一个对象：

```
AnObject := AClass.Create;
try
    // use the object...
finally
    AnObject.Free;
end;
```

笔记 类似的情况也会发生在 TObject 类别的两个方法：AfterDestruction 跟 BeforeConstruction 上面。这两个方法是为了与 C++兼容而提供的虚构建构函式与虚构解构函式(但在 Object Pascal 当中倒是很少被使用到)。请记得，在 AfterConstruction 方法中触发例外的话，BeforeDestruction 方法会被呼叫(然后正常的解构函式也会被呼叫)。

假设我们已经适当的在解构函式中适时的释放对象时显示了错误。我们可以更深入的用实例来说明可能的问题，包含如何处理，以及如何修复。假设我们有一个类别，内部包含一个字符串列表，而我们写这以下的程序来建立与释放这个类别(以下是 ConstructorExcept 范例项目的一部分):

```
type
 TObjectWithList = class
   private
      FStringList: TStringList;
   public
      constructor Create (Value: Integer);
      destructor Destroy; override;
   end;
constructor TObjectWithList.Create(Value: Integer);
begin
    if Value < 0 then
     raise Exception.Create('Negative value not allowed');
    FStringList := TStringList.Create;
    FStringList.Add('one');
end;
destructor TObjectWithList.Destroy;
begin
    FStringList.Clear;
    FStringList.Free;
    inherited;
end;
```

乍看之下，这两个源码没有什么不同。建构函式会配置子对象，而解构函式则会把它适当的释放掉。呼叫的源码则是写成如果在建构函式之后触发了例外事件，就会呼叫 Free 方法，但如果例外在建构函式中并没有发生:

```
var
    Obj: TObjectWithList;
```

```
begin
    Obj := TObjectWithList.Create (-10);
    try
      // Do something
    finally
      Show ('Freeing object');
      Obj.Free;
    end;
```

这样行吗?当然不行，当建构函式在建立字符串列表之前就发生例外的话，系统就会立刻呼叫解构函式，这时就会试着释放还没建立的字符串列表对象，而引发第二个存取错误，或是类似的错误了。

为什么会这样?如果我们把建构函式的顺序颠倒(先建立字符串列表，然后触发例外事件)，则一切都不会有问题，因为解构函式需要把该字符串列表对象给是放掉。但这不算真的解决，它只是一个暂时的解法。我们真的该做的，是先考虑怎么保护源码，让解构函式在执行时不要假设建构函式已经完整执行过。以下可以作为一个简单的范例：

```
destructor TObjectWithList.Destroy;
begin
    if Assigned (FStringList) then
    begin
      FStringList.Clear;
      FStringList.Free;
    end;
    inherited;
end;
```

例外的进阶功能

> *这一节也是本书当中您可能在第一次读到的时候想要直接跳过的，因为这一节可能有点太过复杂。除非您已经对 Object Pascal 很有经验，不然我也会建议您直接跳到下一章，以后再回来看这一节。*

在本章的最后一部分，我想介绍关于例外处理的一些进阶主题。我要来介绍巢状例外(RaiseOuterException)跟在类别中拦截例外的概念(RaisingException)。这些功能并不是早期版本 Delphi 所兼容的，而新版的 Delphi 也因此多了很强大的功能。

巢状例外与内部例外机制

如果我们触发了例外事件，而没有例外事件处理程序来处理，会怎么样?传统的答案是新的例外事件会取代已经存在的例外事件，这也是为何一般的作法至少都要把错误讯息结合，然后用类似下面范例的写法(没有实际的处理，只显示例外相关的讯息):

```
procedure TFormExceptions.ClassicReraise;
begin
    try
        // Do something...
        raise Exception.Create('Hello');
    except on E: Exception do
        // Try some fix...
        raise Exception.Create('Another: ' + E.Message);
    end;
end;
```

上面的源码是 AdvancedExcept 范例项目的一部分。当呼叫方法并处理例外时，我们会看到单一一个例外事件，其讯息是组合过的：

```
procedure TFormExceptions.btnTraditionalClick( Sender: TObject);
begin
    try
        ClassicReraise;
    except
      on E: Exception do
        Show ('Message: ' + E.Message);
    end;
end;
```

相当直觉的输出就是:

```
Message: Another: Hello
```

现在在 Object Pascal 里面，已经全系统都支持巢状例外了。透过异常处理程序我们可以建立、触发一个新的例外，并保留现存的例外对象，并把新旧两个例外结合起来。要达到这个要求，Exception 类别中的 InnerException 属性，它会参照前一个例外，而 BaseException 属性则让我们可以存取前一个例外

事件，因此例外事件得以透过递归方式存在。以下是 Exception 类别相关，用来管理巢状例外的元素：

```
type
    Exception = class(TObject)
    private
      FInnerException: Exception;
      FAcquireInnerException: Boolean;
    protected
      procedure SetInnerException;
    public
     function GetBaseException: Exception; virtual;
     property BaseException: Exception read GetBaseException;
      property InnerException: Exception read FInnerException;
     class procedure RaiseOuterException(E: Exception); static;
     class procedure ThrowOuterException(E: Exception); static;
end;
```

笔记 静态类别方法是类别方法中一个特别的形式。这些跟语言相关的功能我们会在第 12 章里面介绍。

从使用者的观点来透视一下，在保留已发生的例外时，触发一个新的例外，我们应该呼叫 RaiseOuterExcept 这个类别方法(或者使用 C++导向命名的星同方法 ThrowOuterException)。当我们正在处理一个类似的例外，我们可以使用这个新的属性来存取更多的信息。请注意我们在异常处理程序中呼叫 RaiseOuterException，只能像原始码的里面注明的文件所述：

> *使用这个函式从一个例外事件处理程序来触发一个例外实体，您可以获得运作中的例外，并把新的例外事件与之串连、保留着原有例外事件的内容。这会使得 FInnerException 字段被设定为正在运作的例外实体。*
>
> *您应该只从要处理这个新的例外事件的 except 区块来呼叫这个程序，其他任何情形都不该呼叫它。*

我们可以从 AdvancedExcept 范例项目来看一下实际的范例。在这个范例中，我加入了一个方法，会用新的方法触发一个巢状例外(跟前面的范例中的 ClassicReraise 方法做比较):

```
procedure TFormExceptions.MethodWithNestedException;
begin
```

```
  try
    raise Exception.Create ('Hello');
  except
    Exception.RaiseOuterException ( Exception.Create ('Another'));
  end;
end;
```

现在在异常处理程序中，我们就可以同时存取两个例外对象了(也可以看到呼叫新的 ToString 方法):

```
  try
     MethodWithNestedException;
  except
    on E: Exception do
    begin
       Show ('Message: ' + E.Message);
       Show ('ToString: ' + E.ToString);
       if Assigned (E.BaseException) then
          Show ('BaseException Message: ' + E.BaseException.Message);
       if Assigned (E.InnerException) then Show ('InnerException Message: ' +
          E.InnerException.Message);
    end;
  end;
```

这个程序的输出结果如下:

```
Message: Another
ToString: Another
Hello
BaseException Message: Hello
InnerException Message: Hello
```

有两个相关的元素值得留意。第一个是在单一巢状例外的案例中，例外对象的 BaseException 属性与 InnerException 属性，这两个属性都是指向同一个例外对象，也就是原来的例外对象。第二个则是新例外对象的讯息只包含了实际的讯息。而呼叫了 ToString 之后，我们就得到了所有被包含在巢状例外事件的对象的所有讯息组合在一起的字符串，每个讯息之间会以 sLineBreak 这个符号做区隔(您可以看一下 Exception.ToString 这个方法的源码)。使在这个例子里面使用换行符号作分隔，会让产生的结果看起来很奇怪，但我们是

可以用我们想显示的方法加以重新组合的，只要把换行符号换成我们想要的符号，或者把回传的字符串直接指派给字符串列表的 Text 属性即可。

我们接着来看个更深入的范例，看看当两个巢状例外被触发时会怎样？以下是新的方法：

```
procedure TFormExceptions.MethodWithTwoNestedExceptions;
begin
    try
      raise Exception.Create ('Hello');
    except
        begin
            try
                Exception.RaiseOuterException ( Exception.Create ('Another'));
            except
                Exception.RaiseOuterException ( Exception.Create ('A third'));
            end;
        end;
    end;
end;
```

这会呼叫一个方法，和前面我们看过的相同，并产生以下的输出:

```
    Message: A third
    ToString: A third
    Another
    Hello
    BaseException Message: Hello
    InnerException Message: Another
```

此时 BaseException 属性跟 InnerException 属性就会指向不同的对象，而 ToString 的输出值就会变成三行了。

拦截例外

另一个随着新版本被加到例外处理系统的进阶功能是这个方法:

```
procedure RaisingException(P: PExceptionRecord); virtual;
```

根据原始码里面的说明文件所述:

这个虚拟函式会在该例外将要被触发之前被呼叫。在外部例外的案例中，这个函式则会在例外对象建立完成，且触发的条件已经在处理时，尽快被呼叫。

在 Exception 类别中这个函式的实作会管理内部例外(透过呼叫内部的 SetInnerException)，这或许可以解释为何它会放在首位加以介绍，跟内部例外机制放在一起。

现在，我们已经可以在任何情形下透过使用这个功能来获益。透过覆写这个方法，事实上，我们在建立对象之后会有单一的函式被呼叫，不论是否透过建构函式来建立这个例外对象。换句话说，我们可以避免为我们的例外类别定义自定的建构函式，让使用者呼叫基础例外类别的任一个建构函式，然后再套用我们自定的规则。例如我们可以把特定类别的任何一个例外做记录(或者是例外的子类别)。

笔记 除了建构函式之外，透过初始化源码的另一个用意，是覆写 TObject 的虚拟方法 AfterConstruction.

以下是一个自定的例外类别(在 AdvancedExcept 范例项目中)，覆写了 RaisingException 方法：

```
type
  ECustomException = class (Exception)
  protected
      procedure RaisingException(  P: PExceptionRecord); override;
  end;


procedure ECustomException.RaisingException(P: PExceptionRecord);
begin
    // Log exception information
    FormExceptions.Show('Exception Addr: ' + IntToHex (
       Integer(P.ExceptionAddress), 8));
    FormExceptions.show('Exception Mess: ' + Message);
    // Modify the message  Message := Message + ' (filtered)';
    // Standard processing
    inherited;
end;
```

这个方法实作的源码所做的，就是记录下关于例外的一些信息，把例外的讯息做修改，然后呼叫例外基础类别的标准处理流程(需要巢状例外机制才能运作喔)。这个方法会在例外对象被建立之后、触发之前被呼叫。这一点也值得留意，因为由 Show 函式所建立的讯息通常在例外被侦错程序所捕捉之前就已经建立好了。相似的，如果我们在 RaisingException 方法里面放一个断点，侦错程序也会在捕捉到例外事件之前就停在该断点上面。

巢状例外跟这个拦截机制在应用程序的源码里面并不常用到，因为这个功能对于函式库跟组件的开发人员比较有用。

10:属性与事件

在过去三章里面，我们介绍了 Object Pascal 在 OOP 当中的基础，解释了当中的原理、并介绍了目前在大多数面向对象编程语言当中的功能是如何被实作出来的。从早期的 Delphi 开始，Object Pascal 就已经是一个完整的面向对象编程语言，但还包含了一些特别的作法。事实上，当时 Object Pascal 也被质疑只是一个以组件为主的视觉开发工具。

还有一些不相关的功能：这个开发模型的支持是基于一些核心的语言功能而来，例如属性与事件，这两个特性是 Object Pascal 最早提出的，之后才有部分功能被一些 OOP 的语言复制过去使用。像是属性，也可以在 Java 跟 C#里面看到，后来也普及在各种语言中，但这都是直接从 Object Pascal 承袭过去的，即使如此，我个人还是偏好最原始的实作版本，我稍后会简短的说明一下。

Object Pascal 能够支持快速应用程序开发工具(RAD)以及可视化程序设计的原因，就是属性、published 存取描述关键词、组件的原理，以及在这一章里面会介绍的一些其他的原理。这个开发模式时常被简写成 PME (Property-Method-Event)，这是达成快速开发目标的一种实现作法。

定义属性

什么是属性?属性可以被想成是让我们可以用来存取跟变更一个对象的状态，具有潜在的可能性可以透过其副作用来影响对象的规则。在 Object Pascal 里面，属性把对字段、方法的数据存取进行了抽象化与隐藏，使得他们达成了数据封装的主要实作。对属性的一个描述，可以说是『*最极致的数据封装*』。

从技术面来说，属性是可以说是一个具有型别的识别符号，而这个识别符号是透过 read 跟 write 叙述字来跟实际的数据或方法进行链接的。跟 Java, C#不同的是，在 Object Pascal 里面 read 跟 write 叙述字可以链接到一个取得数据/设定数据的方法，也可以直接连结到数据字段。

举个实例，以下是一个使用一般功能日期对象的属性(从数据字段读取，透过方法来写入):

```
private
    FMonth: Integer;
    procedure SetMonth(Value: Integer);
public
    property Month: Integer read FMonth write SetMonth;
```

要存取 Month 这个属性时，这段源码必须读取私有字段 FMonth，而要变更 Month 的内容时，则会去呼叫 SetMonth 这个方法。更改这个数据(只会排除负值而已)的方法，源码会长的像这样:

```
procedure TDate.SetMonth (Value: Integer);
begin
   if (Value <= 0) or (Value > 12) then
      FMonth := 1
   else
       FMonth := Value;
end;
```

笔记 当出现错误的输入时，例如月份的号码输入了负值，一般最好是直接显示错误(透过触发例外事件)，再从背景修正这个输入的数值，但为了维持范例的单纯，所以我把源码保留原样。

请注意，数据字段跟属性的数据型别必须完全一致(如果当中有不同点，我们可以使用简单的方法来进行转换)。设定数据的方法当中的唯一参数、取得数据的回传值，以及属性这三者的型别，必须完全一致才行。

属性的定义当中也可能出现不同的组合(例如我们也可以用一个方法来读取资料，或者直接把一个数据字段指派在 write 关键词之后直接让其他源码来修改)，但使用方法来修改属性的数值是最常见的。以下是一些定义属性的不同组合，仍以上面这个属性做例子:

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;
```

笔记 当我们写了源码要存取属性时，了解一个方法是怎么被呼叫的就很重要。问题是在于有些方法要花上一些时间来执行，这些方法可能会有一些副作用，通常会包含(延迟)更新屏幕上的视觉组件。即使这些副作用并不常被既

载下来，我们仍然应该要记得会有这些副作用，尤其当我们想要优化源码的时候。

一个属性的 write 关键词如果没有被写出来的话，就代表该属性是只读的：

```
property Month: Integer read GetMonth;
```

技术上来说，我们也可以不写 read 关键词，直接定义一个只能被更新的属性，但通常这样做没什么实质作用，所以也不常被这样用。

和其他编程语言的属性相比较

现在，我们如果把这一点跟 Java 或 C#做个比较，会发现这两个语言在属性都是直接对照到方法，但 Java 是隐晦的对照(属性基本上是进行转换)，而 C#比较像 Object Pascal 是要直接写明了如何对照，即使只是跟方法对照：

```
// properties in Java language
private int mMonth;
public int getMonth() { return mMonth; }
public void setMonth(int value) {
   if (value <= 0)
      mMonth = 1;
   else
        mMonth = value;
}
int s = date.getMonth ();
date.setMonth (s+1);

// properties in C# language
private int mMonth;
public int Month {
   get { return mMonth; }
   set {
       if (value <= 0)
          mMonth = 1;
       else
           mMonth = value;
   }
}
```

```
date.Month++;
```

并不是我想要深入讨论不同编程语言中对于属性的相关优点，不过就像我们在本章的介绍当中所述，在属性定义的时候务必要明确定义，另外透过把属性对应到数据字段提升抽象化的等级，而不造成方法的额外负担也是一个好的作法。

这也是为什么我在这么多种编程语言里面，特别偏爱 Object Pascal 对属性的实作方法了。

属性实现了封装概念

属性是非常 OOP 的机制，一个非常棒的作法，让应用程序能够达成数据封装的概念。尤其必要的是，我们必须提供个名称来隐藏一个类别实际上是如何存取信息的(不管是直接存取或是呼叫一个方法)。

事实上，当我们在一个完成的接口中使用了属性，就不可能再去改变它了。同时，如果我们只想让使用者存取到这个类别中的部分数据字段，我们可以把这些字段透过属性加以公布，而不用把数据字段直接搬移到公开区。使用属性，我们不用特别多写什么源码，而且我们仍旧可以更改类别的实作源码。就算我们把直接存取数据字段的作法改成了以方法来存取，我们也不用修改这些属性的原始码。我们只需重新编译器就行了。

笔记 您可能会觉得奇怪，如果我们在属性里面定义让该属性直接存取一个私有区的数据字段，难道不会因此失去数据封装的优点之一吗。用户不就可以直接动到私有区的变量内容，不用透过 getter 或 setter 方法，这样不会失去对数据的保护吗?然而假设用户会直接透过属性存取数据，设计这个类别的开发人员也可以随时更改数据型别，并加入新的 getter 或 setter 方法来因应，这样就算有其他源码使用到该类别，那些源码也不用因此需要修改。这就是我所谓『封装到极致』的意思。另一方面，这也展示了 Object Pascal 程序化的一面，开发人员可以选择任何一种较简单的方法(或者程序执行效率较高的方法)来处理当前的状况，并在有需要的时候，和缓的把程序转换成适当的 OOP 方法。

然而，在使用 Object Pascal 的属性时，也有一个限制。我们可以把一个数值指派给属性，也可以从属性读取数值，我们也可以自由的在判断式里面使用属性，但我们不能把属性当成传址型的参数来传递给函式或程序。这是因为

属性并不是一个实际的内存地址，而是一个抽象的概念，所以不能够用 var 加以传递。举例来说，我们不能把属性传给 Inc 函式，但 C#可以。

笔记 相关的功能，把属性以传址的方式传递，在本章稍后会加以介绍。然而这样的用法非常罕见，而且需要透过对编译器特别的设定，这并不是主流的用法。

在宣告属性时使用代码自动完成的功能

在类别里面加入属性是件很枯燥的工作，所以 IDE 的编辑器让我们可以简单的透过『*代码自动完成*』这个功能来处理。当我们在宣告属性的时候(当然是在类别宣告的时候)，只需要写好前面开头的部份，如以下的例子:

```
type
  TMyClass = class
  public
     property Month: Integer;
  end;
```

先把编辑的文字光标停在这个新增的属性上面，然后按下 Ctrl+Shift+C，编辑器就会自动帮我们把这个属性相应的数据字段、用来修改该属性的方法一起产生出来。不仅是宣告的源码，编辑器会连着实作的源码，例如用来修改该属性的方法以及当中基本的源码，需要宣告什么参数，里头该怎么写，都一起产生出来。换句话说，透过这个热键，上面的属性宣告立刻会变成:

```
type
   TMyClass = class
   private
      FMonth: Integer;
      procedure SetMonth(const Value: Integer);
   public
      property Month: Integer read FMonth write SetMonth;
   end;


{ TMyClass }
procedure TMyClass.SetMonth(const Value: Integer);
begin
   FMonth := Value;
end;
```

如果我们想要修改 getter 方法，只要把 read 部分的定义修改成 GetMonth，像是:

```
property Month: Integer read GetMonth write SetMonth;
```

然后再按一次 Ctrl+Shift+C，GetMonth 这个方法也又会自动被建立出来了，不过 GetMonth 这个方法的内容我们就得自己写了：

```
function TMyClass.GetMonth: Integer;
begin
end;
```

为窗体加入属性

我们来看一个使用属性进行封装的特例。这次我们不建立自定的类别，改成修改 IDE 建立的窗体类别，当然，也会享受到类别自动完成带来的便利。

当应用程序中有多个窗体时，我们通常为了方便，会让窗体之间可以互相存取彼此的信息。我们可以用暴力的作法，就是直接把数据字段宣告在公开区，但这样做在 OOP 的程序概念中其实不好。每当我们需要让窗体之间交换数据的时候，我们应该使用属性。

我们在窗体类别里面宣告简单的属性名称跟型别:

```
property Clicks: Integer;
```

然后按下 Ctrl+Shift+C，启动代码自动完成功能，我们就会看到以下的效果:

```
type
    TFormProp = class(TForm)
    private
        FClicks: Integer;
        procedure SetClicks(const Value: Integer);
    public
        property Clicks: Integer  read FClicks write SetClicks;
    end;
implementation
procedure TForm1.SetClicks(const Value: Integer);
begin
     FClicks := Value;
end;
```

不用说，这功能真的帮我们省下很多打字的时间(更别说有时候会打错字)。现在当用户点击窗体的时候，我们就可以透过下面这行程序把计数器的数字递增，我们可以从 FormProperties 范例项目里面的 OnMouseDown 事件处理程序看到:

```
Clicks := Clicks + 1;
```

您可能会觉得奇怪，怎么不直接对 FClicks 做递增呢？当然，在这个例子里面，直接做递增也是可行的，但我们可能会在 SetClicks 这个方法里面同时更新一些用户接口，然后把目前的数值做更新：如果我们不管属性，直接存取数据字段，setter 方法的副作用可能就不会被执行，例如数值的更新就不会同时被在接口上同步更新了。

这个封装的另一个优点，是我们可以从另一个窗体来读取这个窗体被点击的数字，当然，是透过适当的抽象方式。事实上，窗体类别的属性可以用来读取自定的数据，也可以封装窗体的组件存取。举个例子，如果我们有一个窗体，上面有个卷标用来显示一些信息，而我们想要透过另一个窗体来修改上头的文字，我们可能会用这种暴力写法:

```
Form1.StatusLabel.Text := 'new text';
```

这是通常的实用写法，但并不是个好作法，因为它并没有对窗体的结构或组件提供任何封装。如果我们在整个程序项目中有很多地方都这样写，而稍后我们决定要修改窗体上面的用户接口(或许可以让每次 Label 更新时都做些事情)，我们就得修改很多地方了。

另一个替代方案则是使用方法，或者使用属性来隐藏特定的控件，每个如下所示的宣告:

```
property StatusText: string read GetStatusText write SetStatusText;
```

如果我们输入了上面这行宣告，然后按下 Ctrl+Shift+C，让编辑器帮我们把这两个方法的宣告都先产生好:

```
function TFormProp.GetStatusText: string;
begin
    Result := LabelStatus.Text
end;
procedure TFormProp.SetStatusText(const Value: string);
begin
    LabelStatus.Text := Value;
```

```
end;
```

请注意，在这个案例中，属性并没有跟任何类别中的数据字段对应，而是跟一个子控件，也就是卷标对应(万一我们使用了代码自动完成功能，请记得先把编辑器帮我们产生的 FStatusText 数据字段给删掉喔)。

在程序中的另一个窗体，我们就可以直接参照这个窗体的 StatusText 属性，就算用户接口做了一些修改，到时候也只要改 Get 跟 Set 这两个方法。即使是在窗体当中的其他方法，我们只需要直接使用该属性，不用再去存取原本的控件了：

```
procedure TFormProp.SetClicks(const Value: Integer);
begin
    FClicks := Value;
    StatusText := FClicks.ToString + ' clicks';
end;
```

为 TDate 类别加入属性

我们在第七章里面建立了一个名为 TDate 的类别，现在我们来用属性帮这个类别做些延伸。这个新的范例项目名为 DateProperties，基本上是第七章的 ViewDate 范例项目的延伸。以下是这个类别的重新宣告，里面包含了一些新的方法(作为取得与设定属性的内容)以及四个属性:

```
type
    TDate = class
    private
        FDate: TDateTime;
        function GetYear: Integer;
        function GetDay: Integer;
        function GetMonth: Integer;
        procedure SetDay (const Value: Integer);
        procedure SetMonth (const Value: Integer);
        procedure SetYear (const Value: Integer);
    public
        constructor Create; overload;
        constructor Create (y, m, d: Integer); overload;
        procedure SetValue (y, m, d: Integer); overload;
        procedure SetValue (NewDate: TDateTime); overload;
        function LeapYear: Boolean;
```

```
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string; virtual;
    property Day: Integer read GetDay write SetDay;
    property Month: Integer read GetMonth write SetMonth;
    property Year: Integer read GetYear write SetYear;
    property Text: string read GetText;
  end;
```

Year, Day,跟 Month 属性的读写都是透过对应的方法，以下是其中两个跟 Month 属性相关的方法：

```
function TDate.GetMonth: Integer;
var
  Y, M, D: Word;
begin
   DecodeDate (FDate,Y, M, D);
   Result := M;
end;

procedure TDate.SetMonth(const Value: Integer);
begin
  if (Value < 1) or (Value > 12) then
    raise EDateOutOfRange.Create ('Invalid month');
  SetValue (Year, Value, Day);
end;
```

呼叫 SetValue 会对日期进行实际的编码，万一有错的时候也会触发例外。我定义了一个自定的例外类别，会当数值超过范围的时候被触发：

```
type
  EDateOutOfRange = class (Exception);
```

第四个属性，Text 只对应到一个读取的方法。这个方法被宣告为虚拟方法，因为它会被 TNewDate 子类别取代。而属性的 Get 或 Set 方法没有理由，不应该使用延迟绑定(这个技术在第八章里面已经用很长的篇幅介绍过)。

笔记 在这个范例里面值得一提的，是属性不要直接对应到数据字段。属性可以简单的从储存在不同型别的信息中被计算出来，并用不同的数据结构加以储存。

在类别里面加入了新的属性之后，我们现在可以把范例程序透过属性做一些适当的修改了。例如我们可以直接使用 Text 属性，我们也可以使用一些文字框让用户读写这三个主要属性的数值。我们可以让用户按下 BtnRead 按钮来读取数据:

```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
    EditYear.Text := TheDay.Year.ToString;
    EditMonth.Text := TheDay.Month.ToString;
    EditDay.Text := TheDay.Day.ToString;
end;
```

BtnWrite 按钮则是提供写入的动作，我们可以用以下两种方式来达成这个功能:

```
// Direct use of properties
 TheDay.Year := EditYear.Text.ToInteger;
 TheDay.Month := EditMonth.Text.ToInteger;
 TheDay.Day := EditDay.Text.ToInteger;
// update all values at once TheDay.SetValue (EditMonth.Text.ToInteger,
    EditDay.Text.ToInteger,
    EditYear.Text.ToInteger);
```

这两个方法之间的差异，是在使用者没有输入正确的日期时要怎么处理。当我们让每个数值分开设定的时候，程序可能只修改了年，然后触发例外，并跳过其他的源码，所以到时候日期就只有一部分被修改。当我们把所有的数值一起设定时，要不就格式正确，全部一起设定好，要不就格式错误，原来的日期数值都不变。其实，这就是为何这三个属性要设定为只读的主要原因：要维持数据的完整性。

使用数组属性

通常属性是让我们存取一个数据，甚至是一个复杂的数据型别。但 Object Pascal 也可以定义数组属性，在 C#里面这样的属性也被称为 indexer。数组型别透过进一步的参数，是可以包含任意数据型别的，只要该型别可以被当成索引或者实际数值的撷取者。

以下是定义一个数组属性的范例，在这个范例里面使用了整数作为索引值，储存的内容也是整数：

```
private
    function GetValue(I: Integer): Integer;
    procedure SetValue(I: Integer; const Value: Integer);
public
    property Value [I: Integer]: Integer read GetValue write SetValue;
```

数组属性必须有对应的读取与写入方法，这两个方法都得有个额外的参数用来作为索引值，我们可以透过代码自动完成的功能来定义一般的属性。

在 RTL 里面有一些类别很常使用数组属性，它们的索引值跟数据有很多种组合。例如，在 TStrings 类别里面就定义了以下五个数组属性：

```
property Names[Index: Integer]: string read GetName;
property Objects[Index: Integer]: TObject
        read GetObject write PutObject;
property Values[const Name: string]: string
        read GetValue write SetValue;
property ValueFromIndex[Index: Integer]: string
        read GetValueFromIndex write SetValueFromIndex;
property Strings[Index: Integer]: string
        read Get write Put; default;
```

这些数组属性当中，大多数都是用整数索引值作为列表的参数，少部分则是直接用字符串作为搜寻值(像是上面所列出的 Values 属性)。上面这五个定义当中的最后一个，使用了另一个重要的功能:default 关键词。这也是一个很强大的语法助手: 这个数组属性的名称就可以直接被省却，所以我们可以直接在对象后面加上方括号来存取该属性的内容。

换句话说，当我们有个 TStrings 型别对象 sList 时，以下两种写法都是相同的意义：

```
sList.Strings[1]
sList[1]
```

也就是说，默认的数组属性，可以让任何对象透过自定的方括号[ ]直接存取里面的数据。

以参考设定属性

这是一个相对进阶的主题(且这个功能也很少被用到)，所以如果您对 Object Pascal 还没有很精通的话，可能会想直接跳过这一节。

在 Object Pascal 编译器被延伸去支持 Windows COM 程序的时候，就被赋予了处理”以参考设定”(put by ref)属性的能力(用 COM 的术语来说)，或者说可以接受指针参考值的属性，这属性中储存的不是实际数据。

笔记 ”以参考设定”(put by ref)，是由 Chris Bensen 为这个功能命名的，它是在该功能被介绍之后，在 Chris 的部落格上介绍的: http://chrisbensen.blogspot.com/2008/04/delphi-put-by-ref-properties.html (Chris 当时是这个产品的研发工程师)

在它的设定方法(setter method)当中，是以 var 参数来描述的。假设这会导致相对尴尬的情况，这个功能就该被认为是例外，而不是规则了，这也是为何它为何预设是不使用的。

换句话说，如果我们要使用这个功能，得要自己写上这么一行指令来启用编译设定：

```
{$VARPROPSETTER ON}
```

没有写上这行指令的话，以下的源码就无法被编译，而且编译器会显示这个错误讯息: ”E2282 Property setters cannot take var parameters”:

```
type
 TMyIntegerClass = class
  private
      FNumber: Integer;
      function GetNumber: Integer;
      procedure SetNumber(var Value: Integer);
  public
      property Number: Integer read GetNumber write SetNumber;
  end;
```

这个类别是 VarProp 范例项目的一部分。现在我们会在这个属性的 setter 方法里面看到很奇怪的副作用了:

```
procedure TMyIntegerClass.SetNumber(var Value: Integer);
```

```
begin
    Inc (Value); // Side effect
    fNumber := Value;
end;
```

另一个非常不寻常的效应，是我们无法把常数数据指派给属性，不是变量喔(变量应该是可以的，就像把变量当成任何一个传址函式的参数一样传递)：

```
var
    mic: TMyIntegerClass;
    n: Integer;
begin
    ...
    mic.Number := 10; // Error: E2036 Variable required
    mic.Number := n;
```

这跟我们常用的功能不同，这是相对比较进阶的功能，让我们思考一下是要为一个属性初始化，或是为即将指派给它的数值资料提出警告。这可能导致很奇怪的源码：

```
    n := 10;
    mic.Number := n;
    mic.Number := n;
    Show(mic.Number.ToString);
```

这两种写法效果都一样，但看上去就很奇怪，但它们也都产生了副作用，就是把实际的数字变成了 12。这也许是取得运算结果的方法中，最让人无法理解、觉得荒谬的一种。

发布(Published)存取关键词

除了 public, protected, 以及 private 这几个存取关键词之外(当然还有比较不常用的 strict private 跟 strict protected)，在 Object Pascal 里面还有另一个奇特的存取关键词，叫做 published(发布)。一个发布区的属性(或数据字段、方法)，不只可以在运行时间具备公开属性的特性，同时也会建立出延伸的运行时间型别信息(Run-time Type Information, RTTI)以供查询。

事实上在一个编译后执行的语言里面，编译过的符号会被编译器处理过，在测试程序的时候，可以被侦错程序所使用，但在运行时间并不会被追踪。换句话说(至少在早期的 Object Pascal 是如此)，如果一个类别拥有一个名为

Name 的属性，我们就可以在源码里面跟该类别进行互动，但这个互动会透过被转译成数字化的内存地址进行，在运行时间当中，这些关键词将不复存在，所以我们无从得知一个类别是否有相符的关键词，例如 Name。

笔记 Java 跟 C#也都是编译后执行的语言，两者也都从复杂的虚拟执行环境而获得好处，也因为虚拟执行环境，在这两者当中都可以取得延伸的执行时期信息，通常称为 reflection。Object Pascal 从早期开始就具备有基本的 reflection 功能(透过 published 关键词)，比当时的其他编译后执行的编程语言更早，且之后也提供了更多的 reflection 或延伸 RTTI 功能。基本的 RTTI 信息是跟着 published 关键词，在本章里面被提到的，在第十六章里面会对 reflection 有更全面的介绍。

为什么类别会需要延伸的信息呢?它是 Object Pascal 函式库所依赖的组件模型与可视化程序模型的基础之一。这些信息当中，有一部分会在设计时间的开发环境中被使用到，例如当我们点击了视觉组件时，该组件的相关属性、事件就会被显示在对象查看器中。这可不是写死的程序行表，它们是由编译完成的源码在运行时间检测延伸 RTTI 信息而实时产生出来的。

另一个例子，或许现在来介绍有点太复杂了，则是在建立 FMX 跟 DFM 档案时背后使用到的串流机制。串流机制我们会在第十八章里面介绍，因为它比较属于运行时间函式库的一部分，跟编程语言的核心比较没那么相关。

为这个概念做个整理，published 关键词一般的重要使用时机，是当我们在写组件给自己的程序或者给其他人使用的时候。通常组件的 published 部分只会包含属性，而窗体类别的则还会把 published 用来宣告数据字段跟方法，我们稍后会介绍。

设计时间属性

我们在本章的前面已经介绍过，属性在类别的数据封装技术中扮演了重要的角色。它们也在视觉开发模型的功能中扮演了基础角色。事实上，我们可以写一个组件类别，让它能够在设计时间的窗体中，透过 IDE 的组件列表直接用拖拉的方式被加到一个视觉窗体或者视觉组件上面去，并且可以透过对象查看器(这个工具是 Delphi 提供来存取属性之用。)来跟它的属性做互动。并非所有的属性都可以透过这样的情形来使用，只有在组件类别中被标注为 published 的属性才可以。这也是 Object Pascal 的开发人员要把属性在***设计时间***跟***运行时间***之间做出区隔的原因。

设计时间的属性必须被宣告在类别的 published 区里面，这些属性可以在设计时间的 IDE 跟源码里面使用。而其他宣告在 public 区里面的属性，则不能在 IDE 里面使用，只能在源码里面使用，所以这些属性通常也被称为*运行时间专用*(Runtime only)。

换句话说，我们可以在 Object Pascal 设计时间的 IDE 中，透过对象查看器里面看到该属性的内容，也可以修改它。在运行时间，我们可以从源码里面用同样的方法来存取所有类别宣告在公开区或发布区的属性。

并不是所有的类别都有属性。属性只会在组件，以及 TPersistent 类别的衍生类别才会出现，因为属性可以被串流化，并且存到档案里面。像是窗体档案，就只是窗体上面所有组件的 published 属性的集合而已，也没有记录什么其他的了。
更精确一点，我们不需要从 TPersistent 衍生类别，也可以把属性宣告在 published 区，但我们需要在编译类别的时候设定$M 这个编译器开关。每个透过这个编译器开关进行编译的类别，或者从这样的类别所衍生的子类别，都可以支持 published 区。假设 TPersistent 类别就是包含着这个编译器开关做编译的，所以其下的所有衍生类别也都获得了支持。

笔记 接下来的两小节在默认程序可见区域跟自动化的 RTTI 中都加了额外的信息以支持$M 编译器开关跟 published 关键词。

发布(published)与窗体

当IDE建立了一个窗体，就会把它的组件跟方法放在初始的定义中，在public跟 private 关键词之前。这些放在初始部分的数据字段跟方法都会被视为该类别的 published 区块。

要记得，虽然这些都是全局可视的变量，真的在全局的程序中随意使用它们并不是好主意，真的要这么做，一定要打起十二分精神，在稍早介绍在窗体中加入属性的篇幅已经讨论过了。也要记得，当我们宣告一个 component 类别的元素，在前面明确显清楚存取的关键词时，默认的存取层级就是 published。

笔记 更精准一点来说，published 只有以$M 作为编译器开关，或者从这一类的类别衍生出的子类别中，才会是默认的关键词。就因为在 TPersistent 类别中

使用了这个开关，所以函式库里面大多数的类别，以及大多数的组件类别都默认是 publish 区。然而非组件的类别(像是 TStream 跟 TList)则是以$M-这个设定编译的，所以预设只有到 public 区段。

以下是一个例子:

```
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    btnTest: TButton;
```

要被指派给任何事件的方法，都必须是被发布的(published)方法，且在窗体中对应到组件的数据字段也该是被发布的，这样才能自动跟窗体档案中的对象链接起来，并且自动被建立在窗体里面。也只有被宣告在窗体中最初那区的组件跟方法才能在对象查看器里面显示出来(在窗体的组件列表或者当我们在对象查看器里面以下拉选单选取可以被对应到的事件处理程序时)。

为什么类别的组件应该要被宣告为被发布的呢?如果放在私有区，不是更依循 OOP 的封装规则吗?原因是因为这些组件会在读取串流化表示的数据时就被建立出来，但一旦建立出来，它们就需要被指派给对应的表单域。

这必须透过 RTTI 为被发布的字段来产生。

笔记 技术上来说，组件并没有义务要使用被发布的字段。我们可以把这些字段放在私有区，这样更符合 OOP 的精神。然而，这也需要更多额外的运行时间的源码。我会在本章的最后『RAD 跟 OOP』那个小节来做些说明。

自动 RTTI

另一个在 Object Pascal 编译器当中的特别规则，则是如果我们在类别当中加入 published 关键词，而这个类别并不是从 TPersistent 类别衍生而来，编译器就会自动启动 RTTI，加入{$M+}的规则。

假设我们建立了以下这个类别：

```
type
  TMyTestClass = class
  private
    FValue: Integer;
    procedure SetValue(const Value: Integer);
```

```
    published
        property Value: Integer read FValue write SetValue;
end;
```

编译器就会显示以下的警告讯息:

```
[dcc32 Warning] AutoRTTIForm.pas(27): W1055 PUBLISHED caused RTTI ($M+) to be added to type
'TMyTestClass'
```

编译器就自动的把{$M+}加入了源码。我们可以从范例项目 AutoRTTI 里面找到这段源码。在这个程序中，我们可以写成以下的源码，使用动态方式来存取一个属性(使用传统的 TypInfo 单元):

```
uses
    TypInfo;


procedure TFormAutoRtti.btnTetClick(Sender: TObject);
var
    test1: TMyTestClass;
begin
    test1 := TMyTestClass.Create;
    try
        test1.Value := 22;
        Memo1.Lines.Add (GetPropValue (test1, 'Value'));
    finally
       test1.Free;
    end;
end;
```

笔记 虽然我不常使用 TypInfo 单元跟其中的函式，像是 GetPropValue，RTTI 的真正力量是来自于更现代的 RTTI 单元文件，以及在其中对 reflection 的外延支持。由于这是比较复杂的主题，我觉得应该要用单独的一章来介绍它，并同时介绍 Object Pascal 支持的 RTTI 功能的两个效益。

事件驱动程序作法

在以组件为基础的函式库(当然还有其他很多情境也是)中，我们写的源码并不是把一连串的动作摊开来，而是许多动作在不同的时间点触发的动作累积而成的。以这个概念来看，我想我们在写程序时，都只是在定义程序遇到这些触发的动作时应该如何反应。而这个『动作』可能是使用者的操作，例如点击按钮，可能是操作系统的动作，例如某个传感器的状态改变，或者是远程联机的数据已经准备好了，可能是我们想得到或想不到的任何情形。

这些外部或内部的触发动作，都被通称为事件。事件最初是与讯息导向的操作系统对应的，像 Window，但已经跟原始的理论有很大的不同了。事实上在比较新的函式库里面，大多数的事件都是当用户在设定属性、呼叫方法，或者跟特定的组件互动时由内部驱动的(或者间接由其他类别驱动)。

事件跟事件驱动是怎么跟 OOP 产生关连的?这两个要求在建立新的衍生类别的方式以及时间点都是不同的。

在纯粹 OOP 的格式当中，当一个对象的行为(或方法)跟另一个对象不同时，他们就该被归类为不同的类别。我们已经看过了一些关于这些效应的范例。

让我们思考一下这个情形。一个有四颗按钮的窗体，让每个按钮在被点击时都需要有不同的反应。所以在纯粹的 OOP 名词里，我们应该为这个窗体上的四个按钮都单独做出一个不同的衍生类别，再单独赋予不同的 click 方法。这样的要求理论上是对的，但实际上会产生很多程序的工作要做，而且没有什么弹性。

事件驱动的程序写法，则考虑到相似的情境，并建议开发人员为按钮对象加上一些不同的规则，这些按钮对象仍然可以属于同一个类别。规则上变成了为可以为每个对象的状态进行扩充，而不用衍生不同的类别。这个模式称为委任(delegation)，因为一个对象的行为是委任给一个类别的方法，而不是由该对象所属的类别来处理的。

事件在不同的编程语言中可以用不同的方式加以实现，例如：

- 透过方法参考(在 Object Pascal 里面则称之为方法指标)，或者在内部方法中使用事件对象(像 C#的作法)。

- 把处理事件的源码委任给一个特定的类别，实作一个接口(像在 Java 里面最常见的作法)

- 使用程序区块，就像 JavaScript 里面最常使用的(在 Object Pascal 里面也

支持了匿名方法，我们在第 15 章里面介绍)。虽然在 JavaScript 里面所有的方法也都是程序区块，但在该语言里面，这两个概念就有点混淆了。

如果事件跟事件驱动的程序写作概念，在过去十几年里面已经变得很平常，且在大多数不同的编程语言里面都已经支持，那么 Object Pascal 实作事件的方法就是相当独特的，我们在接下来这一小节里面来谈谈。

方法指标(Method Pointers)

我们在第四章的最后一部分已经介绍过 Object Pascal 的函式指标概念。这是以一个变量来储存函式的内存地址，我们可以依此间接的使用这个函式。一个函式指标是以一个特别的方式来宣告的(以一系列的参数型别跟回传值加以宣告，如果该函式有参数跟回传值的话)。

跟函式指标很像，Object Pascal 里面还有方法指标。方法指针是记录属于一个类别的方法的内存地址。就像函式指标型别，方法指标型别也有一个特殊的宣告方法。然而，方法指针带有更多的信息，像是该方法指标是属于哪个对象的(或者这个方法里面如果需要使用到对象的内容时，应该透过 self 来作为参考)。

换句话说，方法(亦即该方法在内存中的进入地址，由该类别所有的个体共享之)指针是一个对象所属方法的参考(特定个体在内存空间中专属的空间，用以储存该个体的数据)。当我们把一个值指派给方法指标时，我们必须以特定的对象来作为该方法的拥有者，例如特定对象实体的方法。

笔记 如果能够看一下方法指针的定义与数据结构是如何建构出来的，一定会对了解方法指标的实作过程有更深入的了解，这个型别就称为 TMethod。这个记录型别有两个数据字段 Code 跟 Data，分别表示方法的地址跟它所属的对象。在其他的语言里面，程序参考是由委任类别(C#)或者由接口的方法(Java)来掌控的。

方法指针型别的宣告跟程序方法也很像，除了方法指标需要在结尾加上一个 object 关键词：

```
type
    IntProceduralType = procedure (Num: Integer);
    TStringEventType = procedure (const s: string) of object;
```

当我们宣告了一个方法指标，就像上面这样的宣告，我们可以宣告一个以这个方法指针作为型别的变量，并把任何对象中兼容的方法指派给这个变量。什么是兼容的方法?第一是需要跟方法指标拥有相同的参数，例如上述的例子中的方法指针是要求一个字符串为参数。任何一个对象所属方法的参考，只要兼容于方法指标型别，就可以被指派到以该方法指标为型别所宣告的变量里面。

现在，我们有了一个方法指标型别，就可以用这个类型声明一个变量，然后把一个兼容的方法指派给这个变量了：

```
type
    TEventTest = class
    public
        procedure ShowValue (const s: string);
        procedure UseMethod;
    end;
procedure TEventTest.ShowValue (const s: string);
begin
    Show (s);
end;


procedure TEventTest.UseMethod;
var
    StringEvent: TStringEventType;
begin
     StringEvent := ShowValue;
     StringEvent ('Hello');
end;
```

目前这简单的源码还无法真的解释事件的好用，因为上面这源码还聚焦在方法指标型别的低阶理论上。事件就是以此为基础建立起来的，并以储存一个对象的方法指针，指派给不同的对象的作法，超越了这个简单的概念(例如，我们把窗体的 OnClick 事件处理程序指派给按钮)。在大多数的情形下，事件也是透过属性加以实现的。

笔记

这作法不太常见，在 Object Pascal 我们则也可以使用匿名方法来定义一个事件处理程序。不太常见的原因或许是当这个功能在 Object Pascal 里面最近才支持，而许多函式库已经存在很久了。而且这功能的复杂度又更高一些。我们可以在第 15 章里面找到一些范例是关于这个公囊的。另一个可能

的延伸则是对单一一个事件定义了多种事件处理程序，像是 C#就支持这作法，虽然不是标准的功能，但我们可以自己实作出这样的功能。

委任的原理

猛一看，这个技术的目的不太清楚，但它的确是 Object Pascal 组件科技的奠基石之一。秘密就在于*委任*这个名词。如果我们建立了一个对象，在对象里面有一些方法指针，我们就可以很自由的把对象的规则改来改去，只需要把这些指标指派给新的方法就行了。这听起来很习惯吗?你应该习惯的。

当我们为按钮加入一个 OnClick 事件，开发环境就是帮我们做了这件事。该按钮有一个方法指针，名为 OnClick，我们可以直接或间接的把窗体的方法指派给它。当用户点击了这个按钮，这个方法就会被执行，即使我们把这方法定义在另一个类别里面(通常是在窗体中)。

以下我们列出在 Object Pascal 函式库里面的一些实战源码，用来把一个按钮的事件处理程序跟窗体的方法关连在一起:

```
type
    TNotifyEvent = procedure (Sender: TObject) of object;
    TMyButton = class
       OnClick: TNotifyEvent;
    end;

    TForm1 = class (TForm)
       procedure Button1Click (Sender: TObject);
       Button1: TMyButton;
    end;

var
  Form1: TForm1;
```

接下来是一个内部的方法，我们可以这样写：

```
MyButton.OnClick := Form1.Button1Click;
```

上面的源码跟实际上在函式库里面的只有一点点不同，不同的是在函式库里面是以 OnClick 作为属性的名称，而实际的数据则将之命名为 FOnClick。组件的事件会被列在对象查看器的事件分页里面。事实上，事件这种属性也就是方法指针。

这表示我们可以在设计时间动态的把事件处理程序改为与特定对象连接，或者在运行时间建立一个新的组件，然后把事件处理程序指派给新的对象当中的方法。DynamicEvents 范例项目中有这两种作法的案例。窗体中有一个按钮，使用标准的 OnClick 事件处理程序。然而我也在窗体中加入了第二个公开方法，这个方法的参数跟该事件处理常数一致(具备方法兼容性):

```
public
    procedure btnTest2Click(Sender: TObject);
```

当按钮被点击的时候，除了显示讯息，还会把事件处理程序指派到第二个风法去，把额外的动作加入到点击的处理程序中：

```
procedure TForm1.btnTestClick(Sender: TObject);
begin
    ShowMessage ('Test message');
    btnTest.OnClick := btnTest2Click;
end;

procedure TForm1.btnTest2Click(Sender: TObject);
begin
    ShowMessage ('Test message, again');
end;
```

这样一来，当用户第一次点击按钮的时候，原先的事件处理程序会被执行，点第二次的时候，我们就会看到第二个事件处理程序被执行了。

笔记

当我们要输入源码，把一个方法指派给一个事件的时候，代码自动完成的功能会建议我们可用的事件名称，并把它转成一个实际的程序呼叫写法，连小括号都会附上。这是不对的。我们是要把它指派给一个事件处理程序，不是要呼叫它。否则编译器就会试着把该方法执行的结果指派到事件处理程序的变量去(但如果是一个程序，没有回传值，就又会出错)，最后就会发生错误了。

项目源码的第二个部分，介绍了完整的动态事件关连。当我们点击了窗体的画面，就会随机建立出一个新的按钮出来，并把所属的按钮的文字显示出来(Sender 对象会把事件处理程序自动绑定到动态建立的按钮上，该按钮被点击的时候，就会呼叫对应的事件处理程序)：

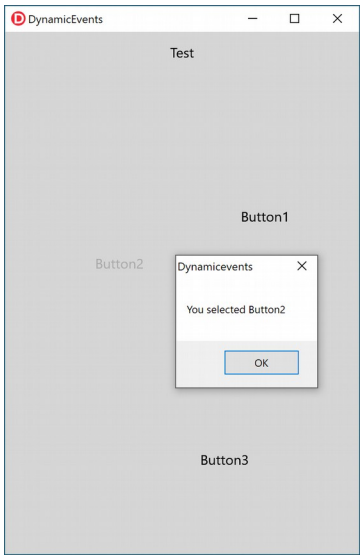
```
procedure TForm1.btnNewClick(Sender: TObject);
begin
```

```
    ShowMessage ('You selected ' + (Sender as TButton).Text);
end;


procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Single);
var
    AButton: TButton;
begin
    AButton := TButton.Create(Self);
    AButton.Parent := Self;
    AButton.SetBounds(X, Y, 100, 40);
    Inc (FCounter);
    AButton.Text := 'Button' + IntToStr (FCounter);
    AButton.OnClick := btnNewClick;
end;
```

透过这段源码，每一个随机建立出来的按钮都会在自己被点击的时候，把自己的按钮文字显示出来，即使使用的是共享的一个事件处理程序，感谢事件里面有使用到 sender 这个参数。这个程序的执行画面，如图 10.1 所示：

图 10.1: DynamicEvents 项目的执行画面，动态产生按钮与讯息



事件也是属性

在 Object Pascal 里面的事件，几乎都是使用方法指针型别的属性来实作的，这一点也很重要。这表示要处理一个组件的事件，我们只需要把方法指派给对应的事件属性即可。用程序的术语来说，这代表着我们可以把一个对象的方法指派作为一个事件处理程序，我们用前几节里面的源码作为范例来看一下:

```
Button1.OnClick := ButtonClickHandler;
```

再提一次，规则是该事件的方法指标型别必须跟我们想要指派的方法兼容，不然的话编译器会指出这个错误。

系统中原就为事件定义了几种方法指标型别，这些型别很常用到，我们从简单的开始看：

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;
```

这就是最常见的 OnClick 事件处理程序的型别，我们用以上这个型别做个简单的方法宣告:

```
procedure ButtonClickHandler (Sender: TObject);
```

这听起来如果有点疑惑，请回想一下在开发环境中的操作。点选任何一个按钮，例如 Button1，我们在按钮上面双击，OnClick 事件就会在开发环境的组件查看器当中被列出来了，并且会在该组件的类别需告终加入一个空白的方法实作源码：

```
procedure TForm1.Button1Click (Sender: TObject) begin
end;
```

我们要自己在上述的这个空的方法里面撰写源码，把任何我们想要的功能加进去。这是因为事件处理程序的方法已经在背景被指派给该事件了。我们在设计时间中，为任何其他的属性进行数值、方法的设定，也都会以完全相同的方法被指派给该组件。

从上面的描述里，我们可以了解到，在事件跟被指派给该事件的方法之间并没有一步一步的对应。相当奇怪。我们可以让很多个事件共享一个事件处理程序，这也说明了为何要很常用到 Sender 这个参数的原因，这个参数让我们知道是哪个对象驱动了这个事件。举例来说，如果我们把 OnClick 这个事件处理程序指派给两个按钮，Sender 这个值在一个按钮对象被点击的时候，就会包含被该按钮的指针。

笔记 我们可以透过源码把同一个方法指派给不同的事件，如上所述，但在设计时间也行。当我们从组件查看器的事件当中点选特定的一个事件，我们可以点选该事件右方的下拉选单钮，它会把目前类别中兼容于该事件的方法都列上来。我们就可以从当中选取一个要用来处理该事件的方法。透过这个方法，我们可以让不同的组件所属的事件用同一个方法来处理。

为 TDate 类别加入一个事件

我们刚刚已经为 TDate 类别加入了一些属性，现在我们再来加入一些方法吧。这个事件将会很简单，它的名字是 onChange，可以用来告诉用户组件的内容已经改变了。要定义一个事件，我们只需定义一个与之对应的属性，然后加入一些数据字段来储存对应的方法指标即可。以下就是 DateEvent 范例项目中，新增的一些定义：

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent read FonChange write FOnChange;
    ...
  end;
```

这些属性的定义实在相当简单。使用这个类别的开发人员可以指派一个新的值给它，然后这个值会被存放在私有区的数据字段 FOnChange 里面。在程序启动的时候，这个数据字段通常是未被指派的: 事件处理程序是给组件的用户用的，不是给组件的制作者用的。组件的制作者如果需要一些规则的话，则需要在组件方法中加入。

换句话说，TDate 类别单纯的接收到一个事件处理程序，当日期数据被改变成，就呼叫储存在 FOnChange 字段的方法。当然，这个方法只会在事件的属性有被指派的时候才会被呼叫。

DoChange 方法(以动态方法来宣告，是传统对事件触发方法的处理准则)会先进行检测，并呼叫该方法：

```
procedure TDate.DoChange;
begin
    if Assigned (FOnChange) then
      FOnChange (Self);
end;
```

笔记 您可能还有印象，在第八章里面，动态方法跟虚拟方法是类似的，但在实作上有些不同，动态方法节省了内存的用量，并节省了相当多的时间。

DoChange 方法会在每次数值改变的时候被呼叫，就像以下的程序所示：

```
procedure TDate.SetValue (y, m, d: Integer);
begin
    fDate := EncodeDate (y, m, d);
    // fire the event
    DoChange;
```

现在，如果我们深入看一下使用这个类别的城市，我们可以简化一下源码。首先我们为窗体类别加入一个自定的方法:

```
type
    TDateForm = class(TForm)
      ...
      procedure DateChange(Sender: TObject);
```

这个方法里面的源码会简单的把目前的卷标文字改为 TDate 对象的 Text 属性的文字内容。

```
procedure TDateForm.DateChange;
begin
    LabelDate.Text := TheDay.Text;
end;
```

而事件处理程序则会在 FormCreate 方法里面被指派:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
    TheDay := TDate.Init (7, 4, 1995);
    LabelDate.Text := TheDay.Text;
    // assign the event handler for future changes
    TheDay.OnChange := DateChange;
```

```
end;
```

这看起来动作很多，难道我们刚刚说事件处理程序可以省下很多源码是骗人的吗?并没有，我们在加入一些源码之后，现在我们可以完全不用再去管日期内容被修改的时候要怎么更新文字卷标了。以下是另一个简单的按钮的 OnClick 事件：

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
    TheDay.Increase;
end;
```

相同的简单源码也在很多其他的事件处理程序中出现。一旦我们指派了事件处理程序，就不用再一直去想着要更新文字卷标了。这把程序中一个很明显的潜在原始码错误给去除了。我们还得记得要在程序开始的地方写一些源码，因为这是个简单的类别，不是组件。如果我们处理的是组件，就只要在组件查看器的事件列表中建立一个事件处理程序，然后在里面写一行程序，更新文字卷标的内容，就打完收工了。

这也带出了另一个问题，到底在 Delphi 里面写个组件是有多困难?其实真的很简单，我们在接下来这几个小节来介绍一下。

笔记 本书不会讲解太多关于撰写自定组件的细节，但会用很简短的以属性、事件的角色来稍微提一下写组件的方法，对这些功能的基本理解，是对每个 Delphi 开发人员来说都很重要的。

建立 TDate 组件

现在，我们已经理解属性跟事件了，下一步我们要来看一下，什么是组件。我们透过把 TDate 类别转换为组件来简单的探索一下这个主题。首先，我们必须从 TComponent 类别来衍生出我们的新类别，这里不再从 TObject 来衍生了，以下是范例源码：

```
type
    TDate = class (TComponent)
      ...
    public
        constructor Create (AOwner: TComponent); overload; override;
```

```
    constructor Create (y, m, d: Integer); reintroduce; overload;
```

我们可以看到，第二步就是帮这个类别加入一个建构函式，我们把默认的组件建构函式覆写，以提供一个适当的数据初始化。因为有一个多载的版本，我们也需要为它用上 reintroduce 这个关键词，以避免编译器显示警告讯息。新的建构函式就简单的设定了今天的日期，在呼叫基础类别的建构函式之后:

```
constructor TDate.Create (AOwner: TComponent);
var
  Y, D, M: Word;
begin
   inherited Create (AOwner);
   FDate := Date; // Today
```

完成到这里之后，我们需要在撰写这个新的组件类别的单元档案里面，加入一个名为 Register 的程序(DateComp 范例项目里面的 Dates 单元文件)。(这个程序的第一个字母 R 务必大写，以免它被误认)。加入这个程序则是 IDE 要新增一个组件的规范。

单纯的宣告这个程序，它不需要任何参数，在这个单元文件的 interface 区段宣告，然后再于 implementation 区段里面撰写源码:

```
procedure Register;
begin
   RegisterComponents ('Sample', [TDate]);
end;
```

这源码把新的组件加入到工具盘的 Sample 分页当中，如果当时没有这个分页，系统就会自动新增这个分页。

最后一个步骤就是把这个写好的组件安装好。我们得建立一个套件(Package)，这是一种特殊的项目，专门用来安装组件用的。我们只需要:

- 点选 IDE 的 File->New->Other menu 选单项目，打开新增项目的对话窗口
- 选择 Package。

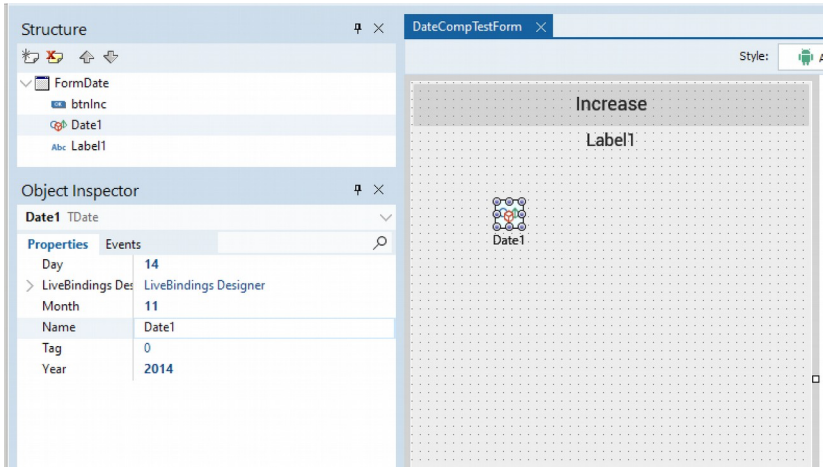
- 把这个套件取个名字存盘(可以在该组件的目录中直接把这个套件项目档存起来)。
- 在新建的套件项目中，请看 Project Manager 这个画面，在内容节点上点鼠标右键，然后为这个项目加入一个新的单元文件，当然就是把刚刚

制作好的 TDate 组件类别的档案加入啰。

- 在 Project Manager 画面上用鼠标右键点击套件项目，先选取里面的 Build 选单项目，建置整个套件，然后再选取鼠标右键选单中的 Install，就可以把我们刚刚写好的 TDate 当成一个组件安装到 IDE 的工具盘了。
- 如果您已经下载了本书的范例项目，那您开启套件项目之后就只需要做上述最后的两个步骤：建置、安装即可。

我们现在建立一个新的项目，然后把目光移到工具盘上面，应该就可以在 Sample 分页里面看到刚刚这个新的组件(TDate)了。此时我们就可以把该组件直接拖拉到窗体上面，也可以从组件查看器上面来处理它的属性了，就如图 10.2 所示。我们也可以更简单的处理 OnChange 这个事件了，很简单，对吧?

图 10.2: 在对象查看器当中显示我们刚做好的 TDate 组件属性



除了我们自己写一个新的项目来使用这个组件(我强烈建议您应该要试试看)，您也可以打开 DateComponent 范例项目，这是我们在上一节里面一步一步制作出来的组件的更新版。这是 DateEvent 范例项目的简化版，因为现在事件处理程序已经可以直接从对象查看器来处理了。

笔记 如果您在编译、安装好组件之前就打开 DateCompTest 范例项目的话，IDE 会认不得当中的组件，所以会有错误讯息，记得先把组件安装好喔。请注意到，在这个案例里面，开发环境提供给我们的错误讯息有两个选项：Cancel 会单纯的忽略未定义的组件，Ignore 则会把相关源码移除。

在类别中实作对于列举功能的支持

我们在第三章里面介绍了以 for-in 循环来作为传统的 for 循环的替代方案，在该节里面，我们介绍了如何对数组、字符串、集合以及其他的系统数据型别使用 for-in 循环。对任何类别都可以用这种循环来处理，只要该类别有支持列举功能。大多数的例子都是内部有一些列表的类别，从技术的观点来看，这个功能是相当具有扩充性的。

我们在 Object Pascal 中实作让类别支持元素列举的功能时，我们必须要加入一个名为 GetEnumerator 的方法，让它回传一个类别(真的会处理列举动作的类别)；为这个列举动作的类别定义 MoveNext 方法与 Current 属性。MoveNext方法提供在元素之间来回移动的功能，Current则回传实际的元素。一旦加好了(稍后我也会用实例来介绍)，编译器就能够解析 for-in 循环，在这循环里面，要处理的目标是我们的类别，当中的每个元素都必须跟列举功能的 Current 属性型别相同。

即使这并不是硬性规定，看起来为类别实作列举功能，把类别当做嵌套类型(我们在第七章已经对这个功能做了介绍)会是个好主意，因为硬要在特定的型别上面用到列举这个功能，其实也没什么意义。

以下的类别是 NumbersEnumerator 范例项目的一部分，会储存一个范围的数字(抽象集合的一种)，并允许从这些数字之中进行列举。这也就使得该类别可以定义列举功能，以嵌套类型进行宣告，并以 GetEnumerator 函式进行回传:

```
type
  TNumbersRange = class
  public
    type
      TNumbersRangeEnum = class
      private
          NPos: Integer;
          FRange: TNumbersRange;
      public
        constructor Create (aRange: TNumbersRange);
        function MoveNext: Boolean;
        function GetCurrent: Integer;
        property Current: Integer read GetCurrent;
      end;
    private
      FNStart: Integer;
```

```
        FNEnd: Integer;
    public
        function GetEnumerator: TNumbersRangeEnum;
        procedure set_NEnd(const Value: Integer);
        procedure set_NStart(const Value: Integer);
        property NStart: Integer  read FNStart write set_NStart;
        property NEnd: Integer read FNEnd write set_NEnd;
    end;
```

GetEnumerator 方法建立一个嵌套类型的对象用以储存列举出来的状态信息。请留意列举类别的建构函式是如何在该对象被列举时保留该对象的参考的(对象会以 self 这个参数被传递)，并在最初设定好起始对象的位置:

```
function TNumbersRange.GetEnumerator: TNumbersRangeEnum;
begin
    Result := TNumbersRangeEnum.Create (self);
end;


constructor TNumbersRange.TNumbersRangeEnum. Create(aRange: TNumbersRange);
begin
    inherited Create;
    fRange := aRange;
    nPos := fRange.nStart - 1;
end;
```

笔记

为何建构函式会把第一个数值设定成-1 呢，而不是如一般期望的设定成第一个数值呢?这披露了编译器为 for-in 循环建立的源码中，对应着建立出的列举功能，以及源码 while MoveNext 的确使用了 Current 属性。测试的动作会在取出第一个值之前就先进行，当整个列表里面还没有任何值存在时。这也披露了 MoveNext 会在第一个元素被使用前就先被呼叫过。这样的作法不需要太复杂的逻辑，上述的例子就简单的把初始值设定成-1，好让第一个值被使用前就会呼叫 MoveNext 方法的副作用消匿于无形。

最后，列举方法会提供一个能够存取到资料的途径，以及可以读取列表中下一个元素的方法(或者在范围中下一个元素):

```
function TNumbersRange.TNumbersRangeEnum. GetCurrent: Integer;
begin
    Result := nPos;
end;
```

```
function TNumbersRange.TNumbersRangeEnum. MoveNext: Boolean;
begin
    Inc (nPos);
    Result := nPos <= fRange.nEnd;
end;
```

我们在上述的源码里面可以看到，Next 方法满足了两种不同的需求，一到列表中的下一个元素，以及检查列表是否已经到了尽头，如果已经到了尽头，这方法就会回传 False。

完成了所有源码以后，我们现在就可以用 for-in 循环来列举出整个范围的对象了：

```
var
    aRange: TNumbersRange;
    I: Integer;
begin
    aRange := TNumbersRange.Create;
    aRange.nStart := 10;
    aRange.nEnd := 23;

    for I in aRange do
      Show (IntToStr (I));
```

执行的结果就很单纯的把列表中的数值列出来，从 10 到 23 之间:

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

笔记 在 RTL 跟 VCL 函式库中，有很多案例定义了列举功能，例如每一个 TComponent 都需要列举该组件所拥有的子组件(Component)。目前还缺的是子控件(Control)的列举功能。在第 12 章的『用类别帮手提供列举功能』中，我们会介绍建立该功能的方法。不在这里介绍的原因，是我们得先讨论类别帮手这个工具。

结合 RAD 开发环境跟 OOP 的 15 个小提示

在这一章里面，我们介绍了属性、事件、发布(published)关键词，这些功能把核心的语言功能跟快速开发环境，或者说视觉开发，也可以说是事件驱动程序写法(这三个名词在概念上几乎是相同的模式)关连在一起。这个模式非常强大，它是以非常坚固的 OOP 为基础的。RAD 的要求与功能常常让开发人员忘却了好的 OOP 实务作法。与此同时，回归纯粹的程序撰写，忘却 RAD 的要求与功能则常常失去效率。在本章的最后一节，我要列出一些结合这两个功能的提示跟建议。我们也可以把这一节称为"超越在 RAD 之上的 OOP"。

笔记 本章最后一小节的材料，是从 1999 年七月份发行的"The Delphi Magazine"的第 17 个议题而来的。当时该篇文章的标题是"用 Delphi 写 OOP 的 20 个规则"。现在我删去了部分规则，把其他的做了些整理，剩下来的这些，每个都很必要。

提示一:窗体是个类别

开发人员常常会把窗体当作对象，但事实上窗体是类别。之间的差异在于我们是否会拿相同的窗体类别来建立多个窗体对象。

让人搞不懂的原因，是 IDE 会预设建立出全局变量以及(要看我们怎么设定)会在程序启动时帮每个项目里面的窗体类别都先产生一个窗体对象出来。这对初学者来说很友善，很容易上手，但对于任何并不直觉的应用程序来说，并不是个好习惯。

当然，帮每个窗体(以及窗体型别)、单元文件取个有意义的名字是很重要的。不幸的，这两个名字必须不同，但我们可以自己定个不成文的规则让两者之间容易被辨识(例如 AboutForm 跟 About.pas)。

当我们在处理以下的步骤时，就会发现到”窗体是个类别”这个概念是很实用的了。

提示二:为组件命名

为组件取个容易一眼看出其功能的名字也是很重要的。最常用的命名方法，是用一些小写字母为类别型别，然后把该组件的角色放在前面，例如 BtnAddCustomer 或者 EditName。这些都是使用这类规则，很类似的注记法。很难说哪种命名方法最好，这要看你自己的喜好了。

提示三:为事件命名

为属性或者事件处理方法命名也一样的重要。如果我们把组件命名得宜，例如为该组件的 onClick 事件建立的处理程序，预设的名称就会是 BtnAddCustomerClick。即使我们可以从按钮的名字猜出那个方法的作用，但能从方法的名称一眼看出该方法的作用还是比较好的。例如，BtnAddCustomer 按钮的 OnClick 事件可能被命名为 AddCustomerToList，如果这是该方法要做的动作。

这会让源码更具可读性，特别是当我们需要从另一个类别的方法中呼叫事件处理程序时，也可以帮助开发人员把同一个方法指派给多个不同组件的事件，即使我得说使用 Actions 来把同一个事件指派给多个用户接口的元素是比较好的作法，尤其是对任何一个不直觉的程序来说。

笔记 Actions 跟 ActionList 组件在 VCL 跟 FireMonkey 这两个接口函式库当中都是很好的结构性功能，它们提供了让程序人员可以从概念上把使用者操作(与状态)跟接口控件分离的功能。启动该控件就可以执行对应的动作。如果我们把 Action 给 disable 了，对应的 UI 元素也会呈现为 disable 的状态。这个主题超越了本书的范围，但如果您需要使用这些框架的话，值得花时间了解一下。

提示四:使用窗体的方法

如果窗体是类别，它的源码就会被集结成方法。除了事件处理程序，事件处理程序扮演了一个特别的角色，但还是可以像其他方法一样被呼叫，它对于

在窗体类别中加入自定方法是很有用的。我们可以加入方法来执行动作或者存取该窗体各组件的状态。为窗体加入公开的方法，比让其他窗体直接存取这个窗体上头的组件来的好上许多。

提示五:建立窗体建构函式

在运行时间建立的第二个窗体，可以在默认的建构函式之外，在另外提供特定功能的建构函式(从 TComponent 类别衍生而来的窗体)。当我们需要特别的初始化程序时，我的建议是为 Create 方法进行多载，加入需要的初始化参数，就像以下的范例源码:

```
public
 constructor Create (Text: string); reintroduce; overload;


constructor TFormDialog.Create(Text: string);
begin
    inherited Create (Application);
    Edit1.Text := Text;
end;
```

提示六:避免使用全局变量

应该避免使用全局变量(在此是指宣告在单元文件的 interface 部分的变量)这样可以避免导致源码难以维护或者产生难以避免的错误。

如果您需要为窗体加入额外的储存数据，请在私有区加入一些数据字段。这样一来每个窗体实体就都可以有自己的一份数据了。

我们也可以使用单元文件变量(在单元文件的 implementation 区段宣告)，这种变量是期望在同一窗体类别的多个实体之间进行数据共享，但这个作法比使用类别的数据字段更好(我们会在第十二章说明)。

提示七:绝不要在类别实作的源码中使用特定实例变量名

我们绝不能在类别的方法中提及特定的对象。换句话说，在 TMyForm 类别的任何方法里面，绝对不要直接用 MyForm 这个变量。如果需要参考到目前的窗体对象，请用 self 关键词。请牢记在心，大多数的时间都不需要直接写

出变量名称，当我们希望能够参考到目前这个对象的方法或数据时。如果您不遵循这个规则，在建立多个窗体实体的时候，就会遇上大麻烦了。

提示八:尽量少用 Form 变量名

即使是在其他窗体的源码里面，也尽量不要直接参考到全局对象，例如 MyForm。最好还是宣告局部变量，或者私有区的数据字段来参考其他窗体，如 FMyForm。例如，一个程序的主窗体可能拥有一个参考到对话框的私有数据字段。显然的，如果我们计划要让第二个窗体类别建立多个实体时，这个规则就变得很必要了。我们可以在主窗体里面建立一个动态数组，或者直接用全局对象 Screen 的 Forms 变量来参考目前应用程序中的任何一个窗体。

提示九:去除全局的 Form1 变数

事实上，我的建议是，当我们把一个窗体加入项目的时候，拿掉 IDE 自动帮我们建立的全局 form 对象，像是 Form1。这个可能性只在我们取消自动建立该窗体的功能时。我建议的这个动作是让您可以摆脱 IDE 的某些副作用。而您可以删掉某些对应的源码，就是被放在项目档里面那些用来自动建立窗体实体的源码。

不用说，如果窗体没有自动被建立，我们就需要在应用程序中加入一些源码来建立它，可能也需要宣告一些变量来储存被建立的窗体。

我想，对于 Object Pascal 的初学者来说，去除全局的窗体对象是很有用的，因为初学者就不会再被类别跟全局对象搞混了。事实上，移除了全局对象之后，所有参考到这个对象的源码都会发生错误了。

提示十:加入窗体属性

犹如我在本章的”为窗体加入属性”小节所说的，当我们需要在窗体里储存数据的时候，就加一个私有的数据字段。如果我们需要从别的类别存取这个数据，并且为这个窗体加入属性。为了这个要求，我们可以改变窗体的源码跟它的资料(包含在它的用户接口中)而不改动任何其他窗体或类别的源码。我们应该也要使用属性跟方法来为第二个窗体或对话框做初始化，然后读取它的最终状态。初始化的动作也是使用建构函式，如我已经说过的那样。

提示十一:把组件属性披露

当我们需要存取另一个窗体的状态，我们无法直接参考它的对象。这样会把其他的窗体或类别的源码跟用户接口绑定在一起，这是应用程序的移植当中变化最大的，但也可规避这些组件管理逻辑中的封装逻辑。另外，宣告一个窗体的属性会对应到关联到组件的属性：这会以一个用来读取组件状态的 Get 方法，以及用来写入它的 Set 方法。假设我们现在修改了用户接口，把组件替代另一个。而我们需要做的，就是把 Get 跟 Set 方法改成与属性相关联。我们不用检查或更改任何一个使用到这个组件档案的方法。

提示十二:适时的在需要使用数组属性

如果我们需要在窗体里面处理一系列的值，可以宣告一个数组属性。假如这个信息对窗体来说很重要，我们也可以把它设定为窗体的默认属性，那我们就可以直接存取这些属性了，语法上可以直接写成 Self[3]或者用另一种写法 FDataForm[3]。这个作法已经在本章里面”使用数组属性”那一小节里面透过几个一般的案例介绍过了(不过该案例没有使用窗体类别就是了)。

提示十三:使用属性的副作用

记得使用属性，而不要直接存取全局数据的优点之一，就是在读写属性的时候，我们可以顺便呼叫某些方法或做一些我们想加入的处理。例如我们可以在设定很多属性的内容时，顺便把窗体的画面直接更新一下，或者呼叫其他方法，甚至可以一次把多个组件的状态更新，或者在需要的时候触发事件。

另一个相关的范例是使用属性的取得函式(getter)来实作延迟建立。在类别的建构函式里面不建立子对象，我们可以在该对象第一次被要求存取的时候才建立，可以写成像这样:

```
private
    FBitmap: TBitmap;
public
    property Bitmap: TBitmap read GetBitmap;


function TBitmap.GetBitmap: TBitmap;
begin
    if not Assigned (FBitmap) then
       FBitmap := ... // create it and initialize it
    Result := FBitmap;
end;
```

提示十四:隐藏组件

我很常听到 OOP 的狂热份子这么抱怨: 在窗体里面的发布区里面放了一堆组件。岂不是就不符合封装的精神了吗?他们的确只出了一个重要的问题，但大多数人似乎并没有意识到不用重写函式库或变更编程语言的解决方法就在眼前。窗体上面各个组件的参考可以搬到私有区，这样就不能直接被其他窗体存取，接着我们可以把组件宣告成属性(请见前面的几个小节)，这样就刀切豆腐两面光，既符合了封装的精神，又可以让组件存取很有弹性了。

如果 IDE 把所有的组件都放在发布区。这是因为要能够把窗体透过串流化储存到档案里面(DFM 檔或者 FMX 檔)。当我们为组件命名之后，VCL 会自动把该组件的对象与参考都加到窗体里面去。这也只有当该组件的参考是位于发布区才办的到，因为串流系统使用了传统的 RTTI 跟一些 TObject 的方法来完成这个功能。

所以如果我们直接把组件的参考从发布区搬到私有区，我们就会遗失这个自动建立的规则。要解决这个问题的话，只要把这个动作改成人工处理，把以下的源码加入到窗体的 OnCreate 事件处理程序即可：

```
Edit1 := FindComponent('Edit1') as TEdit;
```

我们得做的第二个动作则是在系统中注册这个组件类别，这样 RTTI 信息才会能够在编译后的程序中，以及 IDE 系统里面使用。这动作对每个组件类别都只需要做一次，且只有当我们把属于该型别所有组件的参考都搬到私有区的时候才需要。我们可以把这个函式加入到源码里面，就算我们不确定程序里需要用到它也没关系，例如呼叫 RegisterClasses 这个方法对系统没有负面影响。这个方法通常是在储存该类别的单元文件的初始区段(initialization section)里面呼叫的：

```
RegisterClasses([TEdit]);
```

提示十五:使用 OOP 窗体精灵

一直重复前面讲到的，对每个窗体里面每个组件做两个处理程序真的很无聊，也很花时间。为了免除这些沉重的负担，我写了一个简单的精灵，在一个小窗口里面把这些源码加到程序里面去。您只要简单的把这些产生好的源码剪贴到每个窗体里面就行了，因为这个精灵不会自动把原是码放到单元文件的初始区段里去。

要怎么拿到这个精灵?您可以在以下网址里面找一下”Cantools Wizards”:

```
http://www.marcocantu.com/tools
```

提示结论

以上只是一些提示的简单整理，以及让您在 RAD 跟 OOP 开发模式中取得更好平衡的一些建议。当然关于这个主题还有更多的东西要讲，已经超越本书要介绍的范畴了，本书主要是集中介绍语言本身，而不是如何找到最佳的应用程序架构。

笔记 有一些书介绍了关于 Delphi 的应用程序架构，但是没有一本比 Nick Hodges 写的一系列更好，这系列的书包含了”Coding in Delphi”, “More Coding in Delphi”, 以及”Dependency Injection in Delphi”。您可以在下面这个网址找到跟这些书相关的信息: http://www.codingindelphi.com/

11:接口

相异于 C++跟一些其他的编程语言，Object Pascal 的继承模式并不支持多重继承。这表示每个类别都只有一个相同的基础类别。

多重继承的功效一直都是被 OOP 专家们热烈讨论的议题。Object Pascal 不使用多重继承这个功能，一直都是两面评价，说这是缺点的人，就说这样一来功能就没有 C++强，说这是优点的人，就说这可以简化语言本身，并避免掉不少多重继承导致的麻烦。Object Pascal 用来弥补没有多重继承功能缺憾的作法，是使用了接口的概念，透过接口，我们可以宣告一个类别，同时实现多种抽象化定义出来的功能。

笔记 今天大多数的面向对象编程语言并不支持多重继承，而改为使用接口(interface)，包含 Java 跟 C#。接口提供了弹性、让类别能够宣告支持多个接口，并加以实现的能力，同时避免因多重继承而可能产生的问题。多重继承的支持目前大多受限于 C++语言。一些动态的面向对象编程语言则支持 mixins，这是另一个达到类似多重继承功能的方法。

为了避免陷入这个讨论的泥淖，我很简单的假设能够从多种面向来处理一个对象是有用的。但在我们用这个理论建立范例之前，我们得先介绍在 Object Pascal 里面，接口所扮演的角色，以及它的工作原理。

从一个宏观的角度来看，接口比类别支持了更多不同的面向对象程序设计的模式。实作接口的对象，可以视作它所实作的每个接口的多重型态，实际上以接口为基础的模式是很强的。跟类别相较，接口比较偏重在封装，并提供给类别之间一种比继承更宽松一点的连结(当实作没有被继承下去时)。

笔记 在本章所介绍的技术，是在 Object Pascal 语言发展之初就已经被支持，并用来实作 Windows COM 架构的。后来这个功能又被延伸在该情境之外，成为可以广泛使用的技术，但 COM 里面的一些元素，例如接口代号、参考计数这些功能，仍然留在了目前 Object Pascal 对接口的实作当中，也使得 Object Pascal 跟其他语言有明显的不同。

使用接口

除了宣告抽象类(拥有抽象方法的类别)，在 Object Pascal 里面我们也可以撰写***纯粹的抽象类***;也就是只包含虚拟抽象方法的类别。透过使用特别的关键词，interface 来定义一组作为接口(interfaces)的数据型别。

从技术面来看，接口不算是类别，虽然接口可以重组类别。因为类别可以建立实体，但接口不行。接口可以被一个或多个类别加以实作，所以这些实体就可以算是***支持***了或者***实作***了该接口。

在 Object Pascal 当中，接口有一些独特的功能:

- 接口型别的变量会使用参考计数，跟类别型别的变量不同，接口提供了一系列的自动内存管理机制
- 类别是从单一前代类别继承而来，但可以实作多个接口
- 就像所有类别都是自 TObject 衍生而来，所有的接口都是从 IInterface 衍生出来的，两者各归属于独立、正交的架构
- 接口的名称，会以大写字母 I 开头，跟类别的 T 大写字母是不同的。

笔记 起初在 Object Pascal 里面基础的接口型别被称为 IUnknown，这是 COM 所需求的。后来，IUnknown 接口被更名为 IInterface，为了强调这个事实，您仍然可以在没有使用 COM 的情形下使用接口，即使在没有 COM 存在的操作系统里面也一样。反正，IInterface 实际上的规则仍然跟 IUnknown 完全相同。

宣告接口

看过了核心理论之后，让我们来看看实际的例子吧，实例会帮助我们了解 Interface 在 Object Pascal 是如何作业的。在实务层面，接口的定义就像类别的定义写法。这个定义包含有一些方法，但这些方法不用实作，就像在一般类别里面的抽象方法一样。

以下就是一个接口的定义:

```
type
  ICanFly = interface
    function Fly: string;
  end;
```

假设每个接口都直接或间接的从基础接口型别继承而来，相对的写法就变成了:

```
type
    ICanFly = interface (IInterface)
        function Fly: string;
    end;
```

稍后我们会说明从 IInterface 继承的含义为何，并作一个比较表。现在我们就先把 IInterface 也有一些基础类别的方法吧(再提一下，跟 TObject 不一样)。

还有跟接口宣告相关最后一个重点。对接口来说，部分的型别检查是动态进行的，系统要求每个接口都要具备一个唯一的标识符，或者称为 GUID，我们可以在 IDE 里面按下 Ctrl+Shift+G，编辑环境就会自动帮我们产生一组。

以下是接口宣告的完整源码：

```
type
    ICanFly = interface
        ['{D7233EF2-B2DA-444A-9B49-09657417ADB7}']
        function Fly: string;
    end;
```

这个接口跟相关的实作都可以在 Intf101 范例项目里面找到。

笔记 虽然我们就算没指定 GUID 给接口，编译器也仍然可以编译成功，我们也会想要建立出 GUID 给它，因为在进行接口查询，或者透过 as 进行动态型别转换的时候都会用到 GUID。接口的重点，是因为它在运行时间具备了强大的扩充性，这些都是因为接口拥有 GUID。

实作接口

任何类别都可以实作接口(个数不限)，只要把它们列在基础类别后面，并且提供这些接口方法的执行源码即可：

```
type
    TAirplane = class (..., ICanFly)
        function Fly: string;
    end;
function TAirplane.Fly: string;
begin
```

```
    // actual code
end;
```

当类别实作接口的时候，它必须*提供*该接口所有方法(参数也必须要跟接口中该方法的宣告完全相同)的实作，所以在本例中的 TAirplane 类别就必须实作 Fly 方法，该方法必须回传一个字符串。假设这个接口也是从一个基础接口(IInterface)衍生而来，这个类别就必须实作出该接口，以及其基础接口的所有方法。

这就是为什么从已经实作了 IInterface 基础接口的基础类别衍生新的类别时，在新类别中实作该接口也相当常见的原因。Object Pascal 运行时间函式库已经提供了一些基础类别是已经实作好基本的方法的。最简单的一个就是 TInterfacedObject 类别，这个源码就会变成：

```
type
   TAirplane = class (TInterfacedObject, ICanFly)
      function Fly: string;
   end;
```

笔记

当我们实作接口时，我们可以选择以静态或虚拟方法来实作。如果我们准备在衍生类别里面覆写方法，使用虚拟方法就有意义。然而同一个功能也有替代作法，我们可以指定基础类别也继承同一个接口，然后覆写该接口的方法。我建议需要的时候，把接口的方法宣告为虚拟方法，这样可以保留未来的扩充弹性。

现在我们已经定义了一个接口，以及用来实作它的类别。我们可以为这个类别建立物件，就把它当成一般类别来对待，可以这么写:

```
var
  Airplane1: TAirplane;
begin
    Airplane1 := TAirplane.Create;
    try
       Airplane1.Fly;
    finally
        Airplane1.Free;
    end;
end;
```

在这个例子里面我们忽略了类别实现接口的事实。不同的是，现在我们也可以宣告以接口为型别的变量了。用接口作为型别来宣告变量，会自动启动参考内存模式，所以我们就可以省略掉 try-finally 区块：

```
var
  Flyer1: ICanFly;
begin
    Flyer1 := TAirplane.Create;
    Flyer1.Fly;
end;
```

这个简单的程序片段是从 Intf101 范例项目节录出来的，其中的第一行一定会引发许多联想。首先，当我们把一个对象指派给接口变量之后，透过 as 这个特殊指令，运行时间会自动检查该对象是否有实作这个接口。我们可以把整个指令这样写:

```
Flyer1 := TAirplane.Create as ICanFly;
```

其次，看我们是使用了直接指派的方式，或是透过 as 指令，运行时间会多做一件事: 它会呼叫对象的_AddRef 方法，增加它的参考计数。这是透过呼叫了从 TInterfacedObject 衍生而来的对象方法而完成的。

同时，当 Flyer1 变量离开了其使用范围(例如执行到 end 区块)，运行时间就会呼叫_Release 方法，接着该变量的参考计数就会自动减一，等到参考计数变成 0，如果需要，该对象就会自动被释放掉。因为有了这个功能，就没有必要在程序里面手动控制对象的释放了。

笔记 前述的范例中没有 try-finally 区块，编译器会自动在编译的时候，在呼叫_Release(有时可能是 Free)时自动加入 try-finally 区块。这在 Object Pascal 当中有很多情形都会发生: 基本上每当一个方法使用了一个或多个受管理的型别时(例如字符串、接口或者动态数组)，编译器就会在该方法的源码编译的时候自动在背景套用 try-finally 区块。

接口与参考计数

我们回头看一下上面的源码，Object Pascal 的对象被以接口变量进行参考的时候，会进行参考计数(除非接口型别的变量有被标注为 weak 或者 unsafe，在稍后的范例中会提到)。我们也看过它们会自动的在没有接口变量参考到时，会自动把对象释放掉。

值得注意的是，当一些编译器的背景动作涉入时(在背景呼叫的_AddRef 跟_Release)，实际上，参考计算器制还是取决于开发人员或者函式库在特定的源码当中如何实现。在最后一个例子当中，参考计数则真的作用了，
因为在 TInterfacedObject 类别的方法中，源码的内容触发了它: (这里仅列出简化过的版本):

```
function TInterfacedObject._AddRef: Integer;
begin
    Result := AtomicIncrement(FRefCount);
end;

function TInterfacedObject._Release: Integer;
begin
    Result := AtomicDecrement(FRefCount);
    if Result = 0 then
    begin
      Destroy;
    end;
end;
```

现在，我们来思考一下，实作 IInterface 在 RTL 里面的另一个基础类别：TNoRefCountObj，。这个类别会把实际的参考计算器制关闭:

```
function TNoRefCountObject._AddRef: Integer;
begin
    Result := -1;
end;
function TNoRefCountObject._Release: Integer;
begin
    Result := -1;
end;
```

笔记

在 Delphi 11 当中，提供了一个新的 TNoRefCountObject 类别，当中定义了让对象可以忽略参考技术的机制，这个类别取代了旧版 Delphi 中名为 TSingletonImplementation 类别(定义在 Generics.Defaults 单元里面)。旧的 TSingletonImplementation 类别仍旧以新的这个 TNoRefCountObject 类别的别名继续存在着。这两个类别的源码基本上是完全相同的。这个改变的主要理由，是因为原本的类别名字看起来名不符实，它当中并没有提供实质的 Singleton 功能。我们会在下一章里面来看一些这个模式的范例。

TNoRefCountObject 并不常被使用，系统中还有另一个类别，实作了可以忽略参考计算器制的接口，因为它有自己的内存管理模式，这就是 TComponent 类别。

如果我们想要建立一个自定组件，在里面实作接口，我们就不用担心参考计数跟内存管理了。我们会在本章的最后，”用接口来实作设计模式(Patterns)” 那一小节里面来看一个自定组件的例子，用它来实作一个接口。

混和参考的错误

使用对象的时候，我们应该只透过对象变量或只透过接口变量来存取对象。把两种作法混着用会打乱 Object Pascal 提供的参考计算器制，就可能导致非常难以发现的内存错误。实务上，如果我们决定要用接口，我们就应该只使用接口型别的变量。

以下是一个可能发生比较不一样错误的案例。假设我们建立了一个接口，用一个类别来实作它，并且用一个全局程序用这个接口作为参数：

```
type
    IMyInterface = interface
        ['{F7BEADFD-ED10-4048-BB0C-5B232CF3F272}']
        procedure Show;
    end;

    TMyIntfObject = class (TInterfacedObject, IMyInterface)
    public
        procedure Show;
    end;

procedure ShowThat (anIntf: IMyInterface);
begin
   anIntf.Show;
end;
```

这段程序看起来很直观，而且百分之百正确。可能发生的错误，只会来自于我们呼叫它的方法(这段程序是 IntfError 范例项目的一部分):

```
procedure TForm1.BtnMixClick(Sender: TObject);
var
   AnObj: TMyIntfObject;
```

```
begin
    AnObj := TMyIntfObject.Create;
    try
        ShowThat (AnObj);
   finally
       AnObj.Free;
   end;
end;
```

上面这段程序则是把一个一般的对象传给原本要求接口当做参数的函式。假设这个对象没有支持该接口，编译器也不会对这个函式的使用提出可能发生错误的警告讯息。在这里的问题，是内存管理的方式。

一开始，对象的参考计数会被设成 0，表示没有参数使用到它。在进入 ShowThat 程序的时候，参考计数会被加成 1。这没问题，程序的呼叫也会正常发生。等程序执行结束的时候，参考计数会被递减为 0，所以该对象就会被正常释放。换句话说，当程序结束，回到呼叫该程序的程序区段时，anObj 对象就会被释放，这实在相当尴尬。当我们执行这段程序时，则会发生内存错误。

有几个方法可以解决。我们可以手动增加参考计数的数字，并使用一些低阶的技巧。但实际的解法，就是不要把接口跟对象的参考混着用，并只使用接口来指向对象(以下这个源码也是 IntfError 范例项目的一部分):

```
procedure TForm1.BtnIntfOnlyClick(Sender: TObject);
var
   AnObj: IMyInterface;
begin
      AnObj := TMyIntfObject.Create;
      ShowThat (AnObj);
end;
```

在这个特例里，就使用了这个解决方案，但在许多其他的状况下，很难发现到正确的写法。再次强调，最终解决方案就是不要把不同型别的参考混在一起使用。

弱化(weak)与不安全(unsafe)的接口参考

从 Delphi 10.1 Berlin 开始，Object Pascal 语言提供了一种新的模型用以管理所有平台上的接口参考。Object Pascal 语言事实上提供了几种不同形式的参考:

- 一般参考会在对象被指派与释放的时候，对该对象的参考进行增减，最终在释放对象的时候，参考计数回归为 0
- 弱化参考 (weak references, 以[weak]关键词进行标注)不会在对象被参考或指派时增加参考计数。这些参考完全都是由系统管理的，所以在指向的对象被释放的时候，会自动被设定为 nil
- 不安全参考(unsafe references, 以[unsafe] 关键词进行标注) 不会在对象被参考或指派时增加参考计数。系统也不会管理这些参考-跟一般的指标没有什么不同。

笔记 弱化与不安全参考的运用，一开始是在行动装置平台中 ARC 内存管理的一部分。但随着 ARC 已经淡出，这个功能目前只剩在接口参考上还有用。

在一般情形中，参考计数会被启用，我们可以写出以下的源码，依靠参考计数的机制来释放临时配置的对象：

```
procedure TForm3.Button2Click(Sender: TObject);
var
    oneIntf: ISimpleInterface;
begin
    oneIntf := TObjectOne.Create;
    oneIntf.DoSomething;
 end;
```

要是这个对象有标准的参考计数实作，而我们想要建立一个接口参考，让这个接口参考完全脱离参考计数的总数，该怎么做呢?我们现在可以在接口变量的前面加上一个[unsafe]属性，就可以达到这个目的了，只要把前面的宣告改成这样:

```
procedure TForm3.Button2Click(Sender: TObject); var
    [unsafe] one: ISimpleInterface;
begin
    one := TObjectOne.Create;
    one.DoSomething;
 end;
```

这个作法并不好，上述的源码可能会造成内存泄漏。摆脱了参考计数，当变量离开了其有效范围，就不会有任何事情发生。有些情境是需要这样的『优点』的，我们可以使用接口，而不影响其他额外的参考。换句话说，不安全的参考可以被视为.....指标，而且不需要任何编译器的额外支持。

现在，在我们考虑使用 unsafe 属性来宣告一个不会增加参考计数的变量之前，要先想想在大多数情境中，我们还有另一个更好的选择: 使用弱化参考(weak reference)。弱化参考也可以避免增加参考计数的内容，但弱化参考还是在系统控制中的。这表示系统会持续监控弱化参考，当对象被释放的时候，指到该对象的所有参考都会自动被设为 nil。但如果使用 unsafe 参考，我们就无从得知该参考所指向的对象的情况了(这情形被称为 dangling 参考)。

在这种情境下，弱化参考是不是很有用呢？很常见的情境，就是两个对象交叉参考。在这样的案例中，事实上，对象会被我们的源码拖累，增加了对彼此的参考计数，这么一来，这两个对象就会永远被留在内存里面，(因为他们的参考计数永远是 1)即使这两个对象已经不会再被使用到。

用以下的程序来做个范例，以下的接口会同时接受一个指到另一个接口的参考，并以一个类别来实作该接口，当做内部参考：

```
type
    ISimpleInterface = interface
        procedure DoSomething;
        procedure AddObjectRef (simple: ISimpleInterface);
    end;
    TObjectOne = class (TInterfacedObject, ISimpleInterface)
    private
      AnotherObj: ISimpleInterface;
    public
      procedure DoSomething;
      procedure AddObjectRef (Simple: ISimpleInterface);
    end;
```

如果我们建立两个对象，并让他们彼此参考，下场就是内存泄漏：

```
var
    One, Two: ISimpleInterface;
begin
    One := TObjectOne.Create;
    Two := TObjectOne.Create;
```

```
One.AddObjectRef (Two);
Owo.AddObjectRef (One);
```

现在，在 Delphi 里面，有效的解法就是在 private 中的字段把 AnotherObj 标注成 weak:

```
private
  [weak] AnotherObj: ISimpleInterface;
```

这样一改，参考计数就不会在我们把一个对象当成参数丢给 AddObjectRef 的时候被增加了，这个数字会维持在 1，而当该变量脱离生命周期，参考计数就会自动回到 0，该变量使用的内存也就会被释放掉了。

现在有许多其他案例可以因为这个功能变得方便，但这个功能在实作上非常复杂，不容易解释。这是个非常强大的功能，但我们需要些时间才能运用自如。同时，它也会在执行时多花些时间，因为弱化参考是受管理的(而 unsafe 参考是没有受管理的)。

如果您对接口使用弱化参考有兴趣，您可以参考第 13 章『对象与内存』当中的章节:『内存管理与接口』。

进阶接口科技

要深入了解接口的能力，在我们探讨在实务上使用的可能情境之前，了解更进阶的功能是很重要的，像是如何实作多个接口，或是如何用不同的名称来帮接口的特定方法提供实作(万一有类别的方法名称重复时)。

另一个重要的功能，是接口也有属性。为了把这些进阶的接口功能做个示范，我们以 IntfDemo 范例项目来做实例。

接口的属性

本节的范例源码是以两个不同的接口为基础的: IWalker 跟 IJumper，两者都定义了一个新的方法跟一个属性。

接口的属性也只是对应了读取跟写入方法而已。跟类别不同的是，我们不能把接口的属性直接对应到数据字段，这就只是因为接口当中不能包含任何源码。

以下是实际的接口定义：

```
IWalker = interface
    ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
    function Walk: string;
    function Run: string;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
    property Position: Integer read GetPos write SetPos;
end;

IJumper = interface
    ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
    function Jump: string;
    function Walk: string;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
    property Position: Integer read GetPos write SetPos;
end;
```

当我们实作一个具有属性的接口，我们只需要实作其中的访问方法，因为接口的属性对类别来说是透明的，而且无法直接使用：

```
TRunner = class (TInterfacedObject, IWalker)
private
    FPos: Integer;
public
    function Walk: string;
    function Run: string;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
end;
```

这段实作源码看起来很简单、直觉(它是 IntfDemo 范例项目的一部分)，透过方法来计算新的位置，并显示要显示的的数据：

```
function TRunner.Run: string;
begin
    Inc (FPos, 2);
    Result := FPos.ToString + ': Run';
end;
```

范例源码中使用了 IWalker 接口，以及以下的 TRunner 实作源码：

```
var
    Intf: IWalker;
begin
    Intf := TRunner.Create;
    Intf.Position := 0;
    Show (Intf.Walk);
    Show (Intf.Run);
    Show (Intf.Run);
end;
```

输出的信息应该不让人惊讶:

```
1: Walk
3: Run
5: Run
```

接口委任

用类似的方法，我们可以定义一个简单的类别来实作 IJumper 接口:

```
TJumperImpl = class (TAggregatedObject, IJumper)
private
    FPos: Integer;
public
    function Jump: string;
    function Walk: string;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
end;
```

这次的实作跟前一版不同的地方，是使用了不同的基础类别: TAggregatedObject。它是一个用来定义内部支持接口的类别，其语法我们稍后介绍。

笔记 TAggregatedObject类别是在System单元里面定义的另一个有实作IInterface的类别。跟 TInterfacedObject 比较一下，TAggregatedObject 对参考计数的实作方法是不同的(基本上是把所有的参考计数委任给控制组件或容器组件)，在控制组件或容器组件中，则实作接口查询的功能。

我们会用不同的方式来使用它。在以下的类别: TMyJumper，我不想用类似的方法来重复实作出 IJumper 接口。反之，我想把这个接口的实作委任给已经实作出该方法的类别来处理。这个作法无法透过继承来达成(我们无法从两个类别作为基础类别来衍生一个新类别)，幸好我们可以透过 Object Pascal 语言的另一个功能：接口委任来达成这个需求。以下的类别透过把方法参考指向一个实作了该接口方法的对象，来达成实作该接口方法的需求。这样就不用真的实作该接口了:

```
TMyJumper = class (TInterfacedObject, IJumper)
private
    FJumpImpl: TJumperImpl;
public
    constructor Create;
    destructor Destroy; override;
    property Jumper: TJumperImpl read FJumpImpl implements IJumper;
end;
```

这个需告指出 IJumper 接口是由 TMyJumper 类别以 FJumpImpl 数据字段加以实作的。当然，这个字段必须实际上实作出该接口的所有方法。为了让这个作法行得通，我们需要在 TMyJumper 对象被建立的时候(建构函式的参数是由身为基础类别的 TAggregatedObject 类别所要求的)，为这个数据字段建立一个适当的对象:

```
constructor TMyJumper.Create;
begin
    FJumpImpl := TJumperImpl.Create (self);
end;
```

这个类别也有一个解构函式用来释放内部对象，它会参考到一个一般数据字段，而不会参考到接口(因为参考计数在这个情境下不会发生作用)。

这个范例虽简单，但并不常见，当我们开始修改一些方法，或者加入新的方法时，仍然需要对内部的 FJumpImpl 对象进行处理。此处的通则则是我们可以在许多类别中重复使用这个接口的实作。这个接口中所使用的源码是以间接的方式实作接口，跟标准的源码完全相同，可以写成:

```
procedure TForm1.Button2Click(Sender: TObject);
var
    Intf: IJumper;
begin
        Intf := TMyJumper.Create;
```

```
        Intf.Position := 0;
        Show (Intf.Walk);
        Show (Intf.Jump);
        Show (Intf.Walk);
end;
```

多重接口以及方法别名

接口的另一个很重要的功能，是为类别提供能够实作多于一个接口的能力。我们透过以下的 TAthlete 来做介绍，它同时实作了 IWalker 跟 IJumper 接口:

```
TAthlete = class (TInterfacedObject, IWalker, IJumper)
private
    FJumpImpl: TJumperImpl;
public
    constructor Create;
    destructor Destroy; override;
    function Run: string; virtual;
    function Walk1: string; virtual;
    function IWalker.Walk = Walk1;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
    property Jumper: TJumperImpl  read FJumpImpl implements IJumper;
end;
```

其中一个接口是直接被实作出来的，而其他的接口则是透过内部对象 FJumpImpl 进行委任，就像我们在前一个范例中所做的一样。

现在我们面临了一个问题。这两个我们想实作的接口，都有一个名为 Walk 的方法，它们除了名称相同，参数也完全一样，所以我们要怎么在一个类别里面同时实作它们呢?Object Pascal 语言里面是怎么处理多个接口中的方法名称重复的呢?解决方法就是把其中一个方法改名，可以透过以下这种写法:

```
function IWalker.Walk = Walk1;
```

这个宣告是指该类别会用 Walk1 这个方法来实作 IWalker 接口的 Walk 方法(代替用具有相同名称的方法)。最后，在这个类别当中对所有方法的实作里，我们需要参照到内部对象 FJimpImpl 的 Position 属性。透过为 Position 属性宣告一个新的实作方法，我们最后可以得到两个独立的 athlete 对象，这是一个奇怪的状况，以下就是这个范例的两个方法:

```
function TAthlete.GetPos: Integer;
```

```
begin
    Result := FJumpImpl.Position;
end;

function TAthlete.Run:string;
begin
    fJumpImpl.Position := FJumpImpl.Position + 2;
    Result := IntToStr (FJumpImpl.Position) + ': Run';
end;
```

我们要怎么为 TAthlete 对象建立一个接口，同时把它指向 IWalker 跟 IJumper 接口?好吧，我们的确不能那样作，因为并没有我们可以使用的基础接口。然而，接口允许更动态的型别检查与型别转换，所以我们可以把一个接口转换成另一个，只要我们所使用的对象的确同时支持两种接口，编译器就能在运行时间中找到让这转换兼容的蛛丝马迹。以下就是这种情形的源码:

```
procedure TForm1.Button3Click(Sender: TObject);
var
    Intf: IWalker;
begin
    Intf := TAthlete.Create;
    Intf.Position := 0;
    Show (Intf.Walk);
    Show (Intf.Run);
    Show ((Intf as IJumper).Jump);
end;
```

当然，我们可以挑选这两个接口的其中之一，然后把它转换成另外一个。透过 as 指令作转换是我们可以进行运行时间型别转换的方法之一，但我们还有更多方法可以用来处理接口，就像我们在下一节要介绍的那样:

接口的多型

在前一节，我们已经介绍过，我们可以定义多个接口，然后用一个类别来实作其中的两个接口。当然，这也可以延伸成多个，数量不拘。我们也可以建立一个接口架构，因为接口也可以从一个既存的接口衍生而来：

```
ITripleJumper = interface (IJumper)
    ['{0876F202-AAD3-11D2-8551-CCA30C584521}']
    function TripleJump: string;
```

```
end;
```

这个新的接口型别包含了其基础接口的所有方法，并新增了一个新的方法。当然，接口兼容性的规则也跟类别一样。

我们在这一节里面想要强调的是一个相当不同的主题，也就是以型别为基础的多型(polymorphism)。假设有一个一般的基础类别对象，我们可以建立一个虚拟方法，并确定正确的版本可以被呼叫。同样的作法也适用于接口。然而，透过接口，我们可以向前迈进一步并常可以透过接口的查询提供动态程序。假设对象跟接口的关系可以很复杂(一个对象可以实作多个接口，并间接的也实作这些接口的基础接口)，那么描绘出这种情境可以做到的愿景就显得更为重要。

首先，假设我们有一个通用的 IInterface 参考，我们怎么知道这个接口的参考有没有支持哪一个特定的接口呢？这里面包含多种技术，明显的跟类别的对应不太一样:

透过 is 指令进行测试(然后或许可以使用 as 进行接下来的型别转换)。这可以用来检查一个对象是否支持该接口，但如果一个对象所参考的接口也支持另一个接口就不行了(也就是说，我们不能用 is 来判断接口的从属关系)。请注意到，任何情况下，使用 as 来进行转换都是必要的:直接把接口型别进行转换，几乎不会有任何侥幸，都会发生错误。

呼叫全局函式 Support，使用它众多的多载版本之一，我们可以把要检查的对象(或者接口)以及要检查是否兼容的接口(透过 GUID 或者接口的型别名称)，当成参数传递给这个函式。如果这个检查过关了，我们就可以把这个接口变量传给实际上要用来储存该接口的变量了。

直接呼叫 IInterface 基础接口的 QueryInterface(这个作法则是比较低阶的作法)，永远要求一个接口变量作为额外的结果，并且会使用 HRESULT 数值作为回传值，不是布尔值喔。

以下是从 IntfDemo 范例项目节录的简单源码范例，用来介绍刚刚介绍的两种使用通用的 IInterface 变量的技术:

```
procedure TForm1.Button4Click(Sender: TObject);
var
    Intf: IInterface;
    WalkIntf: IWalker;
```

```
begin
    Intf := TAthlete.Create;
    if Supports (Intf, IWalker, WalkIntf) then
        Show (WalkIntf.Walk);
    if Intf.QueryInterface (IWalker, WalkIntf) = S_OK
        then Show (WalkIntf.Walk);
end;
```

把 Support 跟 QueryInterface 作比较的话，我建议一定要使用 Support 函式来做检查。因为 Support 提供了比较简单、高阶的选项。

另一个情况下，我们可能会想透过接口来使用多型，这个情形是当我们拥有一个用来储存高阶的接口型别的数组时(但也可能是一个用来储存对象的数组，其中可能有部分对象支持一些接口)。

从接口参考解析出对象

这情形可能发生在 Object Pascal 的很多个版本，当我们已经把一个对象指派给了一个接口变量，原来的对象就再也没办法被存取了。有时候，开发人员会在接口中加入一个 GetObject 方法来处理这种情形，但这是个很奇怪的设计。

在现代编程语言中，我们可以把接口参考转型回原来的对象，我们有三种不同的方法可以做到这件事:

我们可以写一个 is 测试，来验证一个原本为特定型别的对象是否可以被从接口参考解析出来:

```
IntfVar is TMyObject
```

也可以写一个 as 指令来进行型别转换，万一出现错误时可以触发一个例外:

```
IntfVar as TMyObject.
```

最后，还可以直接强制转型，万一出现错误的时候，就回传一个 nil 指标:

```
TMyObject(IntfVar)
```

笔记 在任何情形下，型别转换都只能用在该接口的确是从一个对象转换而来的时候，从 COM 服务器取得该接口的话就不行了。也要记得我们不只可以把

它转换为原始对象的型别，也可以转换为原始对象的任一基础型别(这当然是完全遵循着类别兼容性的基本规则喔)

我们把以下的简单接口跟实作类别当做个范例(它是从 ObjFormIntf 范例项目节录而来的):

```
type
    ITestIntf = interface (IInterface)
      ['{2A77A244-DC85-46BE-B98E-A9392EF2A7A7}']
      procedure DoSomething;
    end;

    TTestImpl = class (TInterfacedObject, ITestIntf)
    public
        procedure DoSomething;
        procedure DoSomethingElse; // not in the interface
        destructor Destroy; override;
    end;
```

有了这些定义，我们就可以定义一个接口变量，把这个对象指派到接口变量去，并且可以透过新的转型指令使用不存在该接口的方法：

```
var
    intf: ITestIntf;
begin
    intf := TTestImpl.Create;
    intf.DoSomething;
    (intf as TTestImpl).DoSomethingElse;
```

我们也可以用以下的方式来写，使用 is 作检测，然后来个直接转型，我们当然可以把该对象转型成其原来的类别，或该类别的任一基础类别:

```
var
    intf: ITestIntf;
    original: TObject;
begin
      intf := TTestImpl.Create;
      intf.DoSomething;
      if intf is TObject then
         original := TObject (intf);
      (original as TTestImpl).DoSomethingElse;
```

考虑到直接转性在失败时会回传 nil，所以我们也要多加上以下的源码以防万一(就不用 is 测试了):

```
original := TObject (intf);
if Assigned (original) then
  (original as TTestImpl).DoSomethingElse;
```

要注意，把对象从接口解析出来，然后储存到一个变量里，可能会导致参考计数的问题: 当接口变量被指派 nil 进去，或者离开执行范围时，该对象就会被删除，而该对象的参考指针就会变成非法的指标。我们可以从 btnRefCountIssueClick 这个事件处理程序的源码里面找到这个问题。

透过接口来实作转换器模式(Adapter Pattern)

我在这一章当中加入了一个小节来介绍转换器模式(Adapter Pattern)作为程序设计世界实战中使用接口的例子。您可以阅读附录 D 作为透过 Delphi 使用设计模式(Design Pattern)的基本介绍。

简单的说，转换器模式是用来把一个类别的接口转换成用户的类别中所预期的另一个。这方法让我们可以把一个既存的类别当成架构中要求的一个被定义的接口来使用。我们可以透过建立一个新的类别架构，让这个新的类别有对应或者从现存的类别衍生、实作需求的接口来完成这个模式的实作。我们可以用多重继承(在那些有支持这个功能的编程语言中就可以用这个方法)，或者透过接口来达到这个功能。在使用接口实作时(我们即将用这个方法作为范例)，一组新架构的类别就会实作这个特定的接口，并且把它的方法对应到现存的规则去。

在特定的情境中，转换器提供了通用的接口让多种组件可以进行查询，这会发生在有些接口没有延续性的时候(这就常发生在 UI 函式库里面)。这个接口称为 ITextAndValue，因为它允许存取组件的属性，可以透过文字化的描述或者数字化的内容:

```
type
  ITextAndValue = interface
    '[51018CF1-OD3C-488E-81B0-0470B09013EB]'
    procedure SetText(const Value: string);
```

```
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;

    property Text: string read GetText write SetText;
    property Value: Integer read GetValue write SetValue;
  end;
```

下一步是为每个我们想要使用这个接口的类别都建立一个新的子类别，例如我们可以这么写:

```
type
  TApadterLabel = class(TLabel, ITextAndValue)
  protected
      procedure SetText(const Value: string);
      procedure SetValue(const Value: Integer);
      function GetText: string;
      function GetValue: Integer;
  end;
```

这些方法的实作都很简单，因为他们可以被对应到 Text 属性，并透过一个几单的型别转换，看要提供的是文字还是数字，都能很简单的达成。然而现在我们有了一个新的组件，我们得要先安装它(如我们在前一章的最后所介绍过的)。把我们建立的每个组件都安装上去，这会花上一些时间。

另一个比较简单的作法，会是使用穿插类别的作法(interposer class idiom)，这个作法是定义一个跟基础类别相同名字的类别，但放在不同的单元文件里面。这会让编译器跟运行时间的串流系统可以适当的进行辨识，所以我们在运行时间就可以得到这个新的特定类别所建立的对象了。唯一的不同，是在设计时间中，我们只能跟基础组件类别的实体互动了。

笔记 穿插类别(Interposer class)，这个名词第一次被提到，是在许多年前的 Delphi Magazine 上面。这个技术多少是有些要骇入类别之中的意味，但有时也的确很有用。在我看来，穿插类别是在不同单元文件之中去定义跟某些类别的基础类别同名的类别，这也是众多 Delphi 常用名词之一。使用上要注意到，穿插类别所在的单元文件，在 uses 引入的时候，必须要列在我们要取代掉的类别所在的单元文件*之后*。换句话说，比较后来被引入的单元文件会取代掉顺序上先被引入的单元文件里面的同名标识符。当然我们可以透过把单元文件的名称明白的写出来作为识别，但大家一般都不会写的这么

明白，于是透过这个方法，我们就可以在全局的名称解析规则里面投机取巧了。

要定义一个穿插类别，我们通常要写一个新的单元文件，在新的单元文件里面会建立一个跟我们要取代的基础类别同名的类别。要参考基础类别，我们必须把它所在的单元文件的名称完整写上来(不然编译器会以为这是一个递归参考，然后就报出错误讯息了):

```
type
  TLabel = class(StdCtrls.TLabel, ITextAndValue)
  protected
      procedure SetText(const Value: string);
      procedure SetValue(const Value: Integer);
      function GetText: string;
      function GetValue: Integer;
  end;
```

在这个例子里，我们不用安装组件，也不用触及现存的程序，但要在 uses 引入单元文件的最后加上这个单元文件的名字。在两种案例中(我在穿插类别写的范例项目不行)，我们可以查询窗体中的组件是否支持这个转换器接口，然后当成练习，写个程序把所有值都设成 50，它会依次影响到不同组件的不同属性:

```
var
    Intf: ITextAndValue;
    I: integer;
begin
    for I := 0 to ComponentCount - 1 do
      if Supports (Components [i], ITextAndValue, intf) then
          Intf.Value := 50;
end;
```

在这个特例中，这个源码会影响进度列组件或者数字显示盒，以及卷标或文字编辑框的 Text 属性。但也有不少的组件会直接忽略它，例如我们没有定义转换器接口的组件就是如此。

这是非常特别的例子，如果我们多看一些其他的设计模式，就会更容易发现到，透过 Object Pascal 的类别以及充满弹性的接口，要实作这些设计模式相对简单多了(Java, C#以及其他透过接口来提供扩充性的热门的编程语言也跟 Object Pascal 一样)。

12:操作类别

在前几章里面，我们介绍了 Object Pascal 语言在对象方面的基础:类别、对象、方法、建构函式、继承、延迟绑定、接口，以及更多其他技术。现在我们要更进一步了，深入观察一些更进阶的主题，以及 Object Pascal 里面跟管理类别相关的特定功能。从类别参考到类别助手，这一章会介绍更多在其他 OOP 语言里面没有的功能，或明显不同的实作方式。

焦点会是类别，以及在运行时间操作类别的方法，这个主题我们也会在第 16 章介绍到 reflection 跟属性的时候更进一步的探讨。

类别方法跟类别数据

当我们在 Object Pascal 跟大多数其他 OOP 里面定义一个类别，我们是在为这个类别的对象(或叫实体) 定义它的数据结构，以及我们可以透过这个对象进行的动作。然而也有另一种可能，是要定义在这个类别里面所有对象之间分享的数据，以及任何一个从该类别中建立出的对象可以被呼叫的方法。

在 Object Pascal 里面需要一个类别方法，我们只要在该方法前面加上 class 关键词即可，不管该方法是程序或是函式:

```
type
    TMyClass = class
        class function ClassMeanValue: Integer;
```

假设我们建立了类别 TMyClass 的对象，名为MyObject，我们可以透过对象，也可以透过类别直接呼叫这个类别方法:

```
var
    MyObject: TMyClass;
begin
  …
  I := TMyClass.ClassMeanValue;
  J := MyObject.ClassMeanValue
```

从上面的源码，我们可以看出，类别方法可以随时使用，即使该类别的对象还没有被建立出来，也可以直接透过该类别去呼叫。有些情境下，类别会由类别方法建立，透过不特别写出的方法，我们就不用建立出该类别的对象了。(有时我们会需要宣告建构函式 Create 来处理)

笔记

通常类别方法的使用，以及只使用类别方法来建立对象是大多数不允许使用全局函式的 OOP 编程语言的常态。Object Pascal 还让我们宣告旧型的全局函式，但经过这几年，系统函式库和开发人员所写的源码已经迈向一致使用类别方法的途径。使用类别方法的优点是类别方法可以逻辑的链接一个类别，该类别为一整组相关的函式扮演了类似命名空间的角色。

类别数据

类别资料是可以在所有型别为同一类别中可以互相分享的数据，提供跨全局数据储存，但可以套用指定的类别存取限制。我们要怎么宣告类别数据呢?简单的定义一个区块然后用 class var 关键词来定义它即可:

```
type
    TMyData = class
    private
        class var CommonCount: Integer;
    public
        class function GetCommon: Integer;
```

class var 关键词会建立一个区块，可以宣告一个或多个类别的数据字段。我们也可以用 var 关键词(这是 var 关键词的新用法)来宣告同一个存取区块中的另一个区段(以下的 private):

```
type
    TMyData = class
    private
        class var CommonCount: Integer;
        var MoreObjectData: string;
    public
        class function GetCommon: Integer;
```

另外宣告类别数据，我们也可以定义类别属性，我们在后续的章节中会再介绍。

虚拟类别方法跟隐藏的 Self 参数

当类别方法的概念已经在许多编程语言里面变得普及，Object Pascal 的实作就变得有些独特的地方。首先，类别方法拥有一个隐晦(或者隐藏)的 self 参数，就像一般的实体方法。然而，这个隐藏的 self 参数就是指向类别本身的一个参考，而不是类别的实体。

猛一看，类别方法拥有这个隐藏的指向类别自己的 self 参数似乎没有什么用。编译器最终会认得类别方法。然而有一个奇怪的语言功能可以解释这一点: 跟大多数编程语言不同，在 Object Pascal 里，类别方法可以是虚拟的，在衍生类别中，我们可以覆写基础类别的类别方法，就像覆写一般的方法一样。

笔记 对虚拟类别方法的支持是和虚拟建构函式的支持(我们可以将之视为是一种特殊要求的类别方法)连结的。这两个功能在许多需要编译，支持强型别的 OOP 编程语言中都是不支持的。

类别静态方法

类别静态方法已经在 Object Pascal 对平台兼容性的章节中介绍过了，原始的类别方法跟类别静态方法之间的差异，在于类别静态方法并不需要参考到它的类别(也就是没有 self 参数指向类别本身)，而且不能是虚拟方法。

以下的范例源码里面包含了一些可能导致错误的源码，我们已经先把会导致错误的源码批注掉了，这些源码都是 ClassStatic 范例项目中所节录出来的：

```
type
    TBase = class
    private
        FTmp: Integer;
    public
        class procedure One;
        class procedure Two; static;
        ...
    end;
class procedure TBase.One;
begin
    // Error: Instance member FTmp inaccessible here
```

```
    // Show (FTmp);
    Show ('One');
    Show (self.ClassName);
end;

class procedure TBase.Two;
begin
    Show ('Two');
    // error: Undeclared identifier: 'self'
    // Show (self.ClassName);
    Show (ClassName);
    Two;
end;
```

在以上两个情形下，我们都可以直接呼叫这些类别方法，或者透过对象来呼叫也行:

```
  TBase.One;
  TBase.Two;

  Base := TBase.Create;
  Base.One;
  Base.Two;
```

有两个有趣的功能，使得 Object Pascal 的类别静态方法变得很有用。第一个是它可以用来定义类别的属性，就像我们在下一节里面会提到的，第二个则是类别静态方法是完全跟 C 语言兼容的。

静态类别方法跟 Windows API callback

因为没有 self 这个隐藏参数，所以静态类别方法可以被传给操作系统(例如 Windows)当做 callback 函式。在实务上，我们可以宣告一个静态类别方法，并加上 stdcall 这个呼叫转换关键词，就可以把它直接当成 Windows API 的 callback，我们把 StaticCallBack 范例项目中的 TimerCallBack 方法做为范例:

```
type
    TFormCallBack = class(TForm)
        ListBox1: TListBox;
        procedure FormCreate(Sender: TObject);
    private
```

```
        class var
            nTimerCount: Integer;
    public
        procedure AddToList(const AMessage: String);
        class procedure TimerCallBack (hwnd: THandle;
             uMsg, idEvent, dwTime: Cardinal); static; stdcall;
end;
```

类别的数据会被 callback 函式用来当做计数器。OnCreate 处理程序会呼叫 SetTimer API，把这个类别静态程序的地址传递过去。

```
procedure TFormCallBack.FormCreate(Sender: TObject);
var
    callback: TFNTimerProc;
begin
    nTimerCount := 0;
    callback := TFNTimerProc(@TFormCallBack.TimerCallBack);
    SetTimer(Handle, TIMERID, 1000, callback);
end;
```

笔记

TFNTimeProc 的参数是一个方法指标，因此我们需要在类别静态方法的名称之前加上一个@符号或者用 Addr 函式传递。因为我们需要取得该方法的地址，而非执行它。

现在，实际的 callback 函式会增加计数器的数值，并且更新窗体，把它当成全局变量(它会被忽略掉，但也会让范例明显的变得复杂)作为对应，把它视为一个类别方法，而不是视为窗体本身:

```
class procedure TFormCallBack.TimerCallBack(
     hwnd: THandle; uMsg, idEvent, dwTime: Cardinal);
begin
    try
      Inc (nTimerCount);
      FormCallBack.ListBox1.Items.Add (
        IntToStr (nTimerCount) + ' at ' + TimeToStr(Now));
    except on E: Exception do
      Application.HandleException(nil);
    end;
end;
```

这里使用的 try-except 区块，是用来避免由 Windows 所送回的任何例外事件。我们必须为 callback 或者 DLL 函式提供后续的程序控管。

类别属性

使用类别静态方法的原因之一，是要实作类别属性。什么是类别属性呢?就跟基本属性一样，它是包含有读写机制的符号。而跟基本属性不同的是，它是属于类别的而且必须透过类别数据或者类别静态方法来实作。这个 TBase 类别(再提一次，是从 ClassStatic 范例项目节录的)有两个类别方法，我们用两种定义的写法来做示范:

```
type
    TBase = class
    private
        class var
              FMyName: string;
    public
        class function GetMyName: string;    static;
        class procedure SetMyName (Value: string); static;
        class property MyName: string read GetMyName write SetMyName;
        class property DirectName: string read fMyName write fMyName;
end;
```

具有实体个数计数器的类别

类别数据跟类别方法可以用来储存类别中整体的信息。这一类的信息中，最好的范例就是这个类别的实体目前一共被建立了几个，也就是目前还存在的实体的数目。CountObj 范例项目就建立了这个情境。这个程序并没有好用到吓人，假设我们只先看几个特定的问题跟它的解法。换句话说，我们先透过很简单的类别作范例，让这个类别里面先只储存一个数字:

```
type
    TCountedObj = class (TObject)
    private
       FValue: Integer;
    private class var
        FTotal: Integer;
        FCurrent: Integer;
    public
```

```
        constructor Create;
        destructor Destroy; override;
        property Value: Integer read FValue write FValue;
    public
        class function GetTotal: Integer;
        class function GetCurrent: Integer;
end;
```

每当一个对象被建立，这个程序就会在呼叫基础类别的建构函式之后把上面源码的两个计数器。每次对象被释放，现有的对象数目也就会被减少：

```
constructor TCountedObj.Create (AOwner: TComponent);
begin
    inherited Create;
    Inc (FTotal);
    Inc (FCurrent);
end;

destructor TCountedObj.Destroy;
begin
    Dec (FCurrent);
    inherited Destroy;
end;
```

类别信息可以无需透过任何一个特定对象，直接就能存取。事实上，也可能是因为在该时间内，内存里面并没有对象存在:

```
class function TCountedObj.GetTotal: Integer;
begin
    Result := FTotal;
end;
```

我们可以用以下这段源码把目前的状态显示出来:

```
Label1.Text := TCountedObj.GetCurrent.ToString + '/' +
    TCountedObj.GetTotal.ToString;
```

这段程序会由一个定时器来执行，执行后会更新一个文字卷标，所以不用透过特定对象实体，也不用透过任何手动程序来触发。而是会由范例项目里面的按钮建立并释放这些对象，或者把它们留在内存里面(因为事实上这个范例项目也可能造成一些内存泄漏的状况)。

类别建构函式(以及解构函式)

类别建构函式提供了让我们为跟类别相关的数据进行初始化的方法，同时它也扮演了类别初始者的角色，但它实际上并没有建构任何东西。类别的建构函式在建构实体的时候，其实不需要作什么事情:它只用来在类别要被使用之前，先为类别本身进行初始化而已。举例来说，一个类别建构函式可以为该类别的资料设定初始值、加载设定或者为该类别加载支持的档案等等。

在 Object Pascal 里面，类别的建构函式算是单元文件初始区(initialization)源码的替代方案。如果在一个单元文件里面，初始区跟类别的建构函式都存在，类别的建构函式会先执行，然后单元文件的初始区才会被执行到。相对的，我们也可以定义一个类别结构函式，它会在单元文件的终止区(finalization)源码之后被执行。

然而，有个很明显的不同是，当单元文件被编译进某个程序，该单元文件的初始区跟终止区源码一定会被执行到，但类别的建构跟解构函式则要看该类别有没有被使用到，才有被执行到的机会。这表示类别的建构函式是比初始区、终止区程序跟链接程序(linker)更为亲近的。

笔记 换句话说，透过类别的建构跟解构程序，如果一个型别没有跟初始区源码连结，该型别就不会成为程序的一部分，也不会被执行。在传统的情形下，相反的逻辑也是成立的。初始区的程序也可能让链接程序带入部分类别的源码，即使这类别可能完全没有在任何地方被执行到。在实务上，这会跟手势架构一起介绍，该架构的源码很庞大，但在没有被用到的时候则不会被编译到执行档里面去。

用源码来做范例，我们可以这么写(节录自 ClassCtor 范例项目):

```
type
    TTestClass = class
        public
        class var
            StartTime: TDateTime;
            EndTime: TDateTime;
        public
            class constructor Create;
            class destructor Destroy;
```

```
    end;
```

这个类别有两个类别数据字段，当我们在初始区跟终结区里面加入以下源码以后，这两个数据字段就会在类别建构函式中被初始化并在类别解构函式中被修改:

```
class constructor TTestClass.Create;
begin
    StartTime := Now;
end;
class destructor TTestClass.Destroy;
begin
    EndTime := Now;
end;
initialization
     ShowMessage (TimeToStr (TTestClass.StartTime));
finalization
     ShowMessage (TimeToStr (TTestClass.EndTime));
```

单元文件被初始化时，程序就如我们预期的被执行，当我们显示出文字对话框时，类别的资料就已经准备好了。当结束程序时，另一个 ShowMessage 的文字框则会在类别解构函式指派数据之前被显示出来。

注意到，我们可以定义多个一般的建构函式跟解构函式，像是预设的 Create 跟 Destroy。但我们不能定义多个类别建构函式或类别解构函式。如果我们这么做，编译器会指出以下的错误讯息：

```
[DCC Error] ClassCtorMainForm.pas(34): E2359 Multiple class constructors in class TTestClass:
Create and Foo
```

在 RTL 里的类别建构函式

在 RTL 里面有些类别已经透过使用类别建构函式得到了一些优势，像是 Exception 类别，就定义了类别建构函式(如下面这个范例程序片段所示)，以及类别解构函式：

```
class constructor Exception.Create;
begin
    InitExceptions;
end;
```

InitExceptions 这个程序会在 SysUtils 单元文件的初始区程序里面先被呼叫。

通常，我认为使用类别建构函式跟类别解构函式会比使用单元文件的初始区跟终结区来的好。在大多数情形下，这只是语法上方便，所以我们可能不会想要回头修改现有的源码。然而如果我们面对从未使用过的数据结构初始化所可能遭遇的风险(因为这个型别的类别从未被建立过)，把初始化的工作搬到类别建构函式去，可以提供很显著的好处。这在一般的函式库当中显然比较常被使用到，当然我们并不可能在程序中用到函式库面的所有功能。

笔记 使用类别建构函式的一个特殊情况，会是在使用泛型类别(generic class)的时候。我们会在这一章里面提到泛型。

实作单一实体模式(Singleton Pattern)

有不少类别，是需要从头到尾只能建立一个实体的。单一实体模式(Singleton Pattern，另一个很常见的设计模式)，就有这样的要求，并建议提供一个可以对于该对象进行存取的全局指标。

笔记 我们在前一章里面已经提过，我们在附录 D 里面提供了一些透过 Object Pascal 的特点介绍设计模式(Design Pattern)的文章。

单一实体模式可以用很多种方式来实现，但一个类别的要求则是呼叫用来存取唯一实体的函式，来取得该类别的唯一实体。在很多情形下，这样的实作也使用了延迟初始化的规则，所以这个全局的实体会直到整个程序有用到它的时候，才会被建立出来。

在这个实作方法中，我使用了类别数据、类别方法，但也在最后要进行结束的时候使用了类别解构方法。以下是相关的范例源码:

```
type
    TSingleton = class(TObject)
        public
          class function Instance: TSingleton;
        private
          class var theInstance: TSingleton;
          class destructor Destroy;
    end;


class function TSingleton.Instance: TSingleton;
```

```
begin
    if theInstance = nil then
       theInstance := TSingleton.Create;
    Result := theInstance;
end;

class destructor TSingleton.Destroy;
begin
    FreeAndNil (theInstance);
end;
```

我们可以在源码里面以下面的写法取得该类别的实体(不管当时该类别的实体是否已经被建立出来):

```
var
  aSingle: TSingleton;
begin
    aSingle := TSingleton.Instance;
```

更进一步的说，我们可以把一般的建构函式给隐藏起来，把一般的建构函数声明在私有区，让任何人都无法违反单一实体模式的规则，不然就用不到这个类别。

类别参考

我们介绍过了好几个跟方法相关的主题，现在我们来看一下跟类别参考以及把我们的范例透过动态建立组件的方法延伸的结果吧。首先要一直提醒自己，类别参考并不是类别，不是对象，它只是一个指向对象的参考而已，它只是简单的一个类别型别的参考而已。换句话说，类别参考型别的变量，其内容就是类别。

类别参考型别决定了类别参考变量的型别。听起来很绕口吧?我们用几行源码来说明这一点。假设我们定义了一个名为 TMyClass 的类别，我们就可以定义一个新的类别参考型别来跟这个类别产生关连:

```
type
    TMyClassRef = class of TMyClass;
```

现在我们就可以定义这两种型别的变量了，第一个变量是指向一个对象，第二个则是指向一个类别:

```
var
    AClassRef: TMyClassRef;
    AnObject: TMyClass;
begin
    AClassRef := TMyClass;
    AnObject := TMyClass.Create;
```

您可能会觉得奇怪，类别参考到底有什么用?通常，类别参考让我们可以在运行时间操作一个类别资料型别。我们可以在表达式里面用类别参考来判断一个数据型别是否合法。事实上，这样的表达式并不多，但部分案例是挺有趣的。最简单的例子，是建立对象的时候。我们可以把上面两行重写一下:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

此时，我已经用了类别参考的建构函式，而没有使用特定实际类别的建构函式。所以是使用了类别参考来建立该类别的对象。

笔记 类别参考是跟 metaclass 的概念相关的，这个概念在许多其他的 OOP 语言里面也存在。然而在 Object Pascal 里面，类别参考并不是一个类别，只是用来定义类别数据的一个特殊的类别型别而已。因此，跟 metaclass(用来描述其他类别的类别)相比就很容易误导了想法。实际上 TMetaClass 也是在 C++ Builder 里面用到的一个名词。

当我们建立了类别参考，我们可以用它来呼叫其对应类别的类别方法。所以如果 TMyClass 有一个类别方法叫做 Foo，我们就可以用以下的任一写法:

```
TMyClass.Foo
AClassRef.Foo
```

如果类别参考不支持跟类别型别相同的型别兼容性规则的话，这看起来也不特别有用。当我们宣告了类别参考变量，例如上例中的 MyClassRef，我们可以把特定的类别或该类别的衍生类别指派给它。所以如果 MyNewClass 是 TMyClass 的子类别，我们就可以这么写:

```
AClassRef := MyNewClass;
```

现在我们来了解一下，这为什么这么有趣，我们必须记得我们可以从类别参考呼叫的类别方法可以是虚拟方法，所以特定的子类别可以覆写这些方法。使用类别参考跟虚拟类别方法，我们就可以在类别方法的层次实作出多型了，这可是只有少数静态 OOP 编程语言才支持的。同时考虑到每个类别都是从 TObject 衍生而来的，所以我们可以让每个类别参考上都带有 TObject 的方法，像是 InstanceSize, ClassName, ParentClass 跟 InheritsFrom。我们会在第 17 章里面介绍这些类别方法跟其他 TObject 类别的方法。

在 RTL 里面的类别参考

System 单元文件跟其他核心的 RTL 单元文件里面宣告了许多的类别参考，包含:

```
TClass = class of TObject;

ExceptClass = class of Exception;

TComponentClass = class of TComponent;

TControlClass = class of TControl;

TFormClass = class of TForm;
```

特别是 TClass 类别参考型别，可以用来储存任何我们在 Object Pascal 所写的类别的参考，因为任何类别都是从 TObject 所衍生出来的。而 TFormClass 参考则是在 FireMonkey 或 VCL 为架构的 Object Pascal 项目里面都有用到的。在两个架构的函式库中，Application 对象的 CreateForm 方法都会要求窗体的类别当成参数，这样才能用来建立窗体:

```
Application.CreateForm(TMyForm1, MyForm);
```

第一个参数是一个类别参考，第二个参数则是一个用来储存 Application 对象建立出来的窗体实体的变量。

用类别参考来建立组件

在 Object Pascal 里面，实务上到底是怎么使用类别参考的?要能在运行时间操作一个数据结构，是开发环境的基本功能。当我们从组件盘上面选了一个组件，把它加到窗体上，我们就是选择了一个数据型别，并且建立了该数据型别的一个组件。(实际上，这是开发环境帮我们在背景偷偷做好的)

为了帮助我们更了解类别参考的运作，我建立了一个名为 ClassRef 的范例项目。在这个范例中的窗体相当简单。上面只有三个 Radio 按钮，放在窗体上方的一个 Panel 里面。当我们点选了其中一个按钮，我们就可以建立

Radio 按钮文字卷标所描述的组件:Radio 按钮，一般按钮，跟文字编辑框。为了让这个程序能正常运作，需要适当为这三个组件命名，窗体必须也要有一个类别参考的字段:

```
private
    FControlType: TControlClass;
    FControlNo: Integer;
```

第一个字段储存每次用户点击按钮时的新数据型别，并修改它的状态。以下是其中一个按钮的方法:

```
procedure TForm1.RadioButtonRadioChange(Sender: TObject); begin
    FControlType := TRadioButton;
 end;
```

另外两个 Radio 按钮都有跟这个方法类似的处理程序，会指派 TEdit 或者 TButton 到 FControlType 这个数据字段。类似的指派也在窗体的 OnCreate 事件处理程序中出现，当做初始化方法来使用。这个源码有趣的部份，是当使用者点击了画面上大多数的 TLayout 组件时都会执行。我选择了窗体的 OnMouseDown 这个事件来取得鼠标的位置:

```
procedure TForm1.Layout1MouseDown(Sender: TObject;
      Button: TMouseButton; Shift: TShiftState; X, Y: Single);
var
    NewCtrl: TControl;
    NewName: String;
begin
    // Create the control
    NewCtrl := FControlType.Create (Self);

    // Hide it temporarily, to avoid flickering
    NewCtrl.Visible := False;

    // Set parent and position
    NewCtrl.Parent := Layout1;
    NewCtrl.Position.X := X;
    NewCtrl.Position.Y := Y;

    // Compute the unique name (and text)
    Inc (FControlNo);
    NewName := FControlType.ClassName + FControlNo.ToString;
```

```
    Delete (NewName, 1, 1);
    NewCtrl.Name := NewName;
    // Now show it
    NewCtrl.Visible := True;
end;
```

这段程序的第一行就是关键了。它会建立用储存在 FControlType 的类别参考来建立一个新的对象。我们简单的透过使用该类别参考的 Create 建构函式来完成。所以我们现在就可以设定 Parent 属性的值、设定它的位置、给他一个名字(名字当然是个字符串)，然后把它设定成可见。

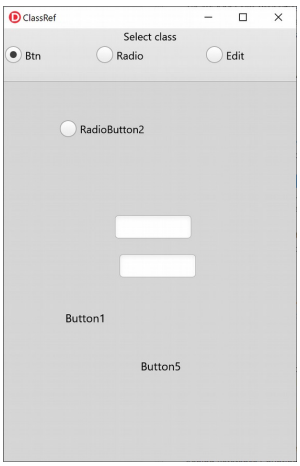
特别介绍源码中用来建立名称这部份，我们模仿了 Object Pascal 的默认命名规则，我用了 FControlType.ClassName，也就是储存在类别参考中的类别名称来当开头，这也是 TObject 类别的类别方法之一。然后加入该组件目前的序号作为结尾，并把最开头的字符串移除。

所以用第一个 Radio 按钮来做说明，假设 FControlNo 是 1，FControlType 是 TRadioButton，因此 FControlType.ClassName 回传 TRadioButton 这个字符串，FControlNo.AsString 会回传 1，我们复制第一个字符串(RadioButton)，然后在后面加上 1，这样就得到了 RadioButton1 这个字符串，很眼熟吧?

我们可以看一下图 12.1，这是程序执行的画面，要注意这个命名规则跟 IDE 的规则并不是完全一致的。IDE 里面对每个型别的组件有独立的计数。

而我们范例程序里面所有组件都用同一个计数。所以如果我们在画面上依序放了 Radio 按钮，一般按钮，以及文字编辑框，它们的名字会分别是：RadioButton1，Button2，以及 Edit3，就如图 12.1 所示(当然文字编辑框是不会显示名字的):

图 12.1: 在 Windows 环境下执行 ClassRef 的画面



记得这一点，跟窗体设计功能不一样，建立出来的 Edit 控件的名字是看不到的。

顺带一提，考虑到一旦我们建立了一般的组件，我们就可以用非常动态的方式来存取它的属性。例如Refection，这个主题我们会在第16章里面介绍。在那一章里面，我们也会介绍更多其他参考到型别的方法、以及透过类别方法引用类别信息等等。

类别与记录助手

我们在第八章里面介绍过，在类别当中继承的概念是用来帮类别提供新功能的一个方法，而且这个方法不会影响到原来的任何实作。这个实作方式也被称为开放的封闭原理:数据型别本身是完整定义了的(封闭)，但仍可以被修改(开放)。

虽然型别继承是一个很强大的机制，在某些情境底下它仍然还不完美。当我们的程序中使用了已经存在的复杂函式库，我们会想要能够对一个数据型别进行扩充，而不想继承类别。当对象是以某些自动化的方式建立时就可以这样做，而且替换掉建立的过程则会很复杂。

另一个对 Object Pascal 开发人员来说相对直觉的情境则是使用组件。如果我们想为一个组件类别加入新的方法，为它提供新的功能，我们就一定得要透过继承来做，但同时我们也就需要:建立新的衍生类别、建立新的套件并且安装它，把所有已经放在窗体上面的这个组件都替换掉、在项目里面所有将要使用到该组件的窗体画面上，也都要用新版本的组件(同时会影响到窗体跟原始文件的两个档案)。

的确很麻烦，另一个替代作法，则是使用类别或者记录助手。这些特殊用途的数据型别能够延伸现有的型别，为这些型别加上新的方法。即使使用类别助手这个作法有些限制，类别助手让我们可以处理像刚刚提到的，只是要简单的为现有组件加入新方法的这种需求，而且不用修改现有的组件型别喔。

笔记 我们其实已经看过一个用来延伸函式库类别，而不用更换掉相关参考的方法了，透过使用继承和相同名称的类别，也就是前一章里面提到的穿插类别(interposer class idiom)。类别助手提供了一个更简洁的模式，然而被耶助

手无法用来替换掉虚拟方法，也无法实作一些延伸的接口，这些我们在前一章都已经提到过了。

类别助手

类别助手是让我们为无法修改的类别(例如函式库类别)加入方法或属性的途径。使用类别助手来为我们自己写的类别做延伸并不常见，因为我们可以直接修改自己能掌握的类别的所有源码。

类别助手是 Delphi 的 Object Pascal 的特别创见，其他的编程语言则有像是扩充方法或者隐含类别来提供类似的功能。

类别助手无法加入实体数据，假设这个数据必须依赖实际的对象，而这些对象是以它们的原始类别定义的，或者碰触到原本定义在原始类别的实际架构中的虚拟方法，类别助手就帮不上忙了。

换句话说，助手类别只能加入或者换掉已存在的非虚拟方法。我们可以用这个方法在原始类别的对象上加入新的方法，即使该类别当中没有现存方法的线索也一样。

这么说或许不够清楚，我们来看个例子(这是 ClassHelperDemo 范例项目的一部分，这是一个我们不该做的范例-用类别助手来处理我们自己制作的类别):

```
type
  TMyObject = class
  private
    Value: Integer;
    Text: string;
  public
    procedure Increase;
  end;

  TMyObjectHelper = class helper for TMyObject
  public
    procedure Show;
  end;
```

以上的源码宣告了一个类别跟这个类别的助手。这表示当我们生成了一个 TMyObject 的对象时，我们也可以呼叫类别助手的方法:

```
Obj := TMyObject.Create;
Obj.Text := 'foo';
Obj.Show;
```

这个助手的类别方法会变成该类别的一部分，我们可以用 self 来呼叫助手类别的方法，就像我们呼叫该对象的任何一个方法一样(类别助手只是扮演助手的角色，因为它并没有实际被建立成一个实体)，如下面的范例所示:

```
procedure TMyObjectHelper.Show;
begin
    Show (Text + ' ' + IntToStr (Value) + ' -- ' +
        ClassName + ' -- ' + ToString);
end;
```

最后，请记得助手类别可以覆写原始的方法。在我们提供的范例源码里面，在类别跟着手类别当中都有名为 Show 的方法，但只有助手类别的方法会被呼叫到。

当然，宣告一个类别以及透过助手类别的语法来宣告同名类别的延伸也是有点用处的，这个作法适用于同一个单元文件，甚至同一个程序中。在范例程序中就是这么做的，但只是为了让大家更容易理解这个技术跟语法。

类别助手在开发应用程序结构的时候，不应该用来当做一般语言的的建构函式，但可以用来为我们没有原始码的函式库里面的类别，或者我们不想动到原始码的类别避免未来修改时发生冲突。

使用类别助手的时候还是有些规则的。类别助手的方法：

- 可以跟原始的类别方法位于不同的访问权限区。
- 可以是类别方法或者实体方法，类别变量与属性
- 可以是虚拟方法，未来可以在衍生类别中被覆写(即使我发现这个功能在实作时有些怪怪的)
- 可以提供额外的建构函式
- 可以在型别定义中加入巢状常数

类别助手中唯一缺少的功能就是常数数据字段。也要注意，类别助手在进入该类别的执行范围起就开始可以被使用了。我们必须在 uses 区段加入对

定义了类别助手的单元文件的引入宣告，才能看见该类别助手的方法，而不是只要在编译的过程中引入就行了。

笔记 某些情形下 Delphi 编译器当中的问题会让类别助手存取原始类别的私有区段，不论类别助手宣告在哪个单元文件都一样。这基本上算是打破了面向对象的封装规则。为了强化对象可视范围的语义，在最近几个版本的 Object Pascal 编译器(从 10.1 Berlin 开始)中，类别跟记录助手就无法存取他们延伸的原始类别私有区的属性跟方法了。这也的确让一些已经写好的项目无法被编译，当然，只有使用到类别助手，并使用到原始类别私有区属性与方法(这是错误，不能当做 Object Pascal 的新功能)的项目才会受影响。

ListBox 的类别助手

类别助手的实际应用，是提供函式库中的类别一些额外的方法。原因可能是我们不希望直接动到这些类别的原始码(即使我们可能拥有这些原始码，但我们不一定会想要动到这些核心函式库的原始码)或者从这些类别制作出衍生类别(使用这个作法，我们就得把窗体里面的组件一一替换了)。

当成做个练习，思考一下这个简单的案例:我们想要有一个简单的方法，能够拿到一个 list box 目前被选取的项目的文字。但不想要用传统写法:

```
ListBox1.Items [ListBox1.ItemIndex]
```

我们可以定义一个像这样的类别助手(节录自 ControlHelper 范例):

```
type
    TListboxHelper = class helper for TListBox
        function ItemIndexValue: string;
    end;


function TListboxHelper.ItemIndexValue: string;
begin
    Result := '';
    if ItemIndex >= 0 then
       Result := Items [ItemIndex];
 end;
```

定义完成之后，我们就可以直接改写成这样了:

```
Show (ListBox1.ItemIndexValue);
```

这只是个很简单的例子，但也用很实用的文字把内涵给表达出来了。

类别助手跟继承

类别助手最明显的限制，就是我们每次对一个类别只能使用一个助手。如果编译器发现了有两个助手类别，第二个助手类别就会把第一个给取代掉。所以也没有任何方法可以对类别助手进行连锁使用，也就是说我们不能对类别助手再制作另一个类别助手。

有个方法可以解决一部分的问题，就是我们可以为一个类别制作一个类别助手，然后当这个类别出现了衍生类别时，再为衍生类别制作另一个类别助手。这样做并不是从助手类别衍生另一个助手类别喔。我并不鼓励让类别助手的结构过于复杂，因为那可能让我们的源码变得很难懂。

当我们要加到一个函式库或者系统已定义完成的数据结构时，助手就很有用。TGUID 记录就是一个例子，它是一个 Windows 的数据结构，我们可以在 Object Pascal 里，在不同的操作系统中使用它，有个型别助手就帮它加了一些常用的功能:

```
type
  TGuidHelper = record helper for TGUID
    class function Create(const B: TBytes): TGUID; overload; static;
    class function Create(const S: string): TGUID; overload; static;
    // ... more Create overloads omitted
    class function NewGuid: TGUID; static;
    function ToByteArray: TBytes;
    function ToString: string;
  end;
```

使用类别助手加入对控件列举的功能

任何在函式库中的 Delphi 组件都自动定义了列举功能，我们可以透过这个功能来列举该组件所拥有、保管的组件或子组件。例如，在窗体的方法中，我们可以用以下的简单源码来列举出该窗体所拥有的所有组件:

```
for var AComp in self do
  … // 使用 AComp
```

另一种常见的作法，则是浏览子控件(control)，这只包含了以该窗体为直接 parent(第二或第三层的不包含)的视觉组件，例如直接放在窗体上面的 Panel

或者 Button (放在 Panel 里的已经算是第二层了)，非视觉组件，像是 TMainMeu，也就不包含在内。我们可以用另一个方法快速让所有控件都能被列举出来，这方法就是为 TWinControl 类别制作一个类别助手：

```
type
    TControlsEnumHelper = class helper for TWinControl
    type
      TControlsEnum = class
      private
          NPos: Integer;
          FControl: TWinControl;
      public
          constructor Create (aControl: TWinControl);
          function MoveNext: Boolean;
          function GetCutrrent: TControl;
          property Current: TControl read GetCurrent;
      end;
    public
        function GetEnumerator: TControlsEnum;
    end;
```

笔记 这个类别助手之所以是为 TWinControl 而写，而非 TControl 是因为只有具备 Windows handle 的控件才能当其他控件的 parent (上代控件)。所以图形控件很自然就被排除在外了。

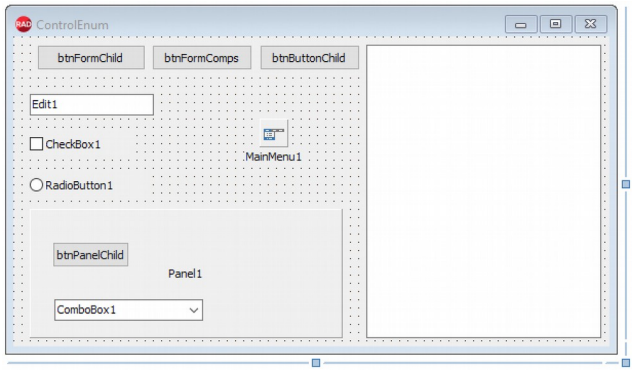
以下是上面这个范例中类别助手的完整源码，包含它自己的方法以及当中定义的类别 TControlsEnum:

```
{ TControlsEnumHelper }
function TControlsEnumHelper.GetEnumerator: TControlsEnum;
begin
    Result := TControlsEnum.Create (Self);
end;
{ TControlsEnumHelper.TControlsEnum }
constructor TcontrolsEnumHelper.TcontrolsEnum.Create (
    AControl: TWinControl);
begin
    FControl := AControl;
    NPos := -1;
end;
```

```
function TControlsEnumHelper.TControlsEnum.GetCurrent: TControl;
begin
    Result := FControl.Controls[nPos];
end;
function TControlsEnumHelper.TControlsEnum.MoveNext: Boolean;
begin
    Inc (NPos);
    Result := NPos < FControl.ControlCount;
end;
```

如果我们建立了一个像是图 12.2 的窗体，就可以测试一下不同的情境。我们是假设第一个情境来写出上面的源码的，也就是列举窗体上面的子控件:

图 12.2: 在 Windows 环境下执行 ClassRef 的画面




```
procedure TControlsEnumForm.BtnFormChildClick(Sender: TObject);
begin
    Memo1.Lines.Add ('Form Child');
    for var ACtrl in Self do
      Memo1.Lines.Add (ACtrl.Name);
    Memo1.Lines.Add ('');
end;
```

接下来则是执行后在 Memo 当中的输出结果，列出了窗体上所有的子控件，但放在 Panel 里面的组件或者控件则没有列出来了:

```
Form Child
Memo1
BtnFormChild
Edit1
CheckBox1
```

```
RadioButton1
Panel1
BtnFormComps
BtnButtonChild
```

如果我们完整的进行列举所有组件，显示的内容应该更多。但是在使用本节开始的源码时，会遇到一个问题：因为我们用类别助手里面的新版 GerNumerator 覆写了原本的版本，因此我们无法直接使用基础类别 TComponent 的列举功能。这个类别助手是为 TWinControl 定义的，所以我们可以投机取巧一下。如果我们把源码中的对象转型为 TComponent，就可以呼叫原本的、标准版的列举功能了:

```
procedure TControlsEnumForm.BtnFormChildClick(Sender: TObject);
begin
    Memo1.Lines.Add ('Form Components);
    for var ACtrl in TComponent(self) do
       Memo1.Lines.Add (AComp.Name);
    Memo1.Lines.Add ('');
end;
```

这段源码执行以后，就会列出比前一版本更多的组件了:

```
Form Child
Memo1
BtnFormChild
Edit1
CheckBox1
RadioButton1
Panel1
BtnPanelChild
ComboBox1
BtnFormComps
BtnButtonChild
MainMenu1
```

在范例 ControlsEnum 项目里面，我已经加入了可以列举出放在 Panel 里面所有子控件的源码了。

现有型别的记录助手

记录助手概念的延伸之一，就是为原生(或者说是编译器现有的型别)数据型别提供了加入方法的能力。即使我们用了”记录助手”这样的名词，但这写法并不是为了记录提供的，而是对一般的数据型别提供的。

笔记 记录助手目前已经被用来延伸、为原生资料型别提供像方法一样的处理，但这在未来应该会有所改变。今天的执行时期函式库定义了的一些原生助手，未来可能就会消失，保持我们目前以这些助手写程序的方法，但在定义这些助手的源码中打破了兼容性。这也是我们为什么不应该过度使用这个功能的原因，即使它用起来很方便，且功能强大。

现有的型别助手是怎么运作的?我们来看看以下对整数型别所定义的助手:

```
type
    TIntHelper = record helper for Integer
        function AsString: string;
    end;
```

现在假设有个整数变量 n，我们可以这么写:

```
n.AsString;
```

我们要怎么定义虚构的方法，而这个方法又是怎么让变量值能够参考到的呢?我们想象一下 self 这个关键词在该函式里被使用到：

```
function TIntHelper.AsString: string; begin
    Result := IntToStr (Self);
 end;
```

注意到，我们也可以在常数上面使用这个助手，可以这么写:

```
Caption := 400000.AsString;
```

然而我们不能对很小的数值作一样的动作，因为编译器会把太小的数值当成是其他的型别来处理。所以如果我们想要把 4 这个数字的常数用上面的方法来转换，就得用以下的第二个写法了:

```
Caption := 4.AsString; // Nope!
Caption := Integer(4).AsString; // OK
```

或者我们也可以用第一个写法，但我们就得定义一些不同的型别助手了:

```
type
    TByteHelper = record helper for Byte...
```

我们在第二章里面已经介绍过，我们用不着像上面这样写了很多的型别助手，像是整数跟 Byte，因为执行时期函式库已经定义了许多类别助手跟大多数核心型别的助手，以下这些都包含在 SysUtils 单元文件里面了:

```
TGUIDHelper = record helper for TGUID
TStringHelper = record helper for string
TSingleHelper = record helper for Single
TDoubleHelper = record helper for Double
TExtendedHelper = record helper for Extended
TByteHelper = record helper for Byte
TShortIntHelper = record helper for ShortInt
TWordHelper = record helper for Word
TSmallIntHelper = record helper for SmallInt
TCardinalHelper = record helper for Cardinal
TIntegerHelper = record helper for Integer
TUInt64Helper = record helper for UInt64
TInt64Helper = record helper for Int64
TNativeUIntHelper = record helper for NativeUInt
TNativeIntHelper = record helper for NativeInt
TBooleanHelper = record helper for Boolean
TByteBoolHelper = record helper for ByteBool
TWordBoolHelper = record helper for WordBool
TLongBoolHelper = record helper for LongBool
TCurrencyHelper = record helper for Currency // added in Delphi 11
```

还有一些现存的型别，它们的助手则被定义在其他单元文件里面:

```
// System.Character:
TCharHelper = record helper for Char
// System.Classes:
TUInt32Helper = record helper for UInt32
// System.DateUtils
TDateTimeHelper = record helper for TDateTime // added in Delphi 11
```

因为我们在本书前面的篇幅里面已经提供了许多范例，所以在这里就不赘述了。加入这一节，是为了让大家知道我们可以怎么为现存的型别定义一个型别助手而已。

型别别名的助手

如同刚刚介绍过的，我们不可能为一个型别定义两个助手，每个型别只会有一个助手。所以我们要怎么为一个原生型别，像是整数，加入一个直接的动作呢?这个需求并没有直接的解决方法,但有几个可以奏效的作法(简单的说，可以复制内部类别助手的原始码，然后用另一个类别助手方法来贴上这些源码)。

另一个我中意的解决方法，是定义一个型别别名。型别别名会被编译器当成一个全新的型别，所以它可以拥有它自己的型别助手，就不用改动原始型别的助手了。现在两个型别被当成不同型别了，我们仍然不能把两个型别助手透过同一个变量来使用，但我们可以透过型别转换来达成这个目的。我们用源码直接来说明吧。假设我们建立了这个型别别名:

```
type
    MyInt = type Integer;
```

现在我们就可以定义这个型别的助手了:

```
type
    TMyIntHelper = record helper for MyInt
        function AsString: string;
    end;


function MyIntHelper.AsString: string;
begin
    Result := IntToStr (self);
end;
```

如果我们定义了一个这个新型别的变量，我们就可以呼叫特定助手的方法来做预设的存取了，但是仍然呼叫整数助手的方法，只要透过转型-以上节录自 TypeAliasHelper 范例项目，您可以用它来做些延伸练习：

```
procedure TForm1.Button1Click(Sender: TObject);
var
    MI: MyInt;
begin
    MI := 10;
    Show (MI.AsString);
    // Show (MI.toString); // this doesn't work
```

```
    Show (Integer(MI).ToString)
end;
```

13:对象与内存

这一章的重点非常特别，但是是相当重要的主题，也就是在 Object Pascal 编程语言中的内存管理。这个编程语言与其执行时期环境提供了一个很独特的解决方法，它是介于 C++风格的人工内存管理，以及 Java 或 C#，由系统完全掌控之间的一个作法。

使用两者之间的作法，是因为它可以帮助我们避免掉内存管理的痛苦(但显然并非全是痛苦)，而避免掉因为垃圾清理所致的问题，从额外的内存配置到没有正确释放内存。

笔记 我并没有特别想要深入到垃圾清理策略的问题，或在不同平台上面，是怎么实作这个功能的。这已经是需要进入研究的议题了。在这里相关的是在局限性比较大的装置(例如行动装置)，垃圾清理目前还不够理想，但在每个平台上还是有些共通的问题。忽略 Windows 应用程序内存使用量的趋势，已经让很多小工具都占用很海量存储器的情况了。

然而，在 Object Pascal 里面有些额外的复杂情形，是需要因为不同的数据型别而使用不同的方式管理内存，有些型别要使用参考计数，而有些要使用传统的管理模式。VCL 组件为主的拥有权模式，以及一些其他的选项让内存管理成为一个复杂的议题。本章就要厘清这些问题，我们会从现代编程语言对内存管理的一些基础开始，接着会介绍在对象参考模型之后的概念。

笔记 这些年来，Delphi 行动装置编译器提供了一个不同的内存模式，称为自动参考计数(ARC，Automatic Reference Counting)。这个管理模式是由 Apple 开始从 Objective-C 语言推广的，ARC 让编译器支持对象参考在内存中被使用的次数，并在这个次数被设为 0 的时候(也就是该对象没有在任何地方被使用到的时候)，就把该对象自动释放掉。这个作法跟 Delphi 在所有平台上对接口参考所处理的作法很像。从 Delphi 10.4 开始，对 ARC 的支持已经从所有平台全数取消了。

全局数据、堆栈以及 Heap

在任何一个 Object Pascal 的应用程序里面所使用的内存可以分成两个领域:源码与数据。在一个程序的执行文件里面包含几个部分，包含资源(例如图片、窗体描述档案)，以及由程序加载到内存使用的函式库。这些内存区块是只读的(在几个平台上，例如 Windows)，且可以让多个处理程序共享。

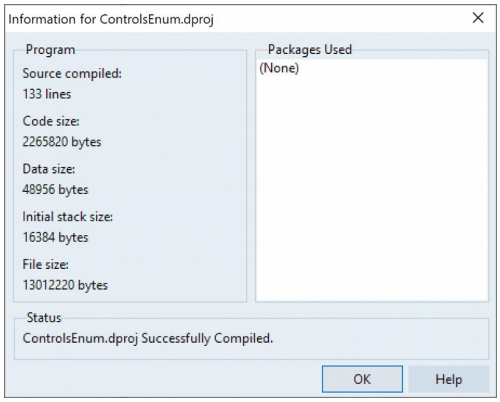
仔细研究资料部分也是很有趣的。Object Pascal 程序(像大多其他的编程语言的写法一样)的数据是分成三大块来储存的:全局内存、堆栈、Heap。

全局内存

在 Object Pascal 编译器建议执行档案的时候，就已经决定了用来储存生命周期从应用程序开始到结束为止的变量数据所需的内存空间。全局变量会在单元文件的 interface 区段或是 implementation 区段进行宣告，全局变量就会使用这一类的内存。要留意到，如果全局变量是一个类别型别(或是字符串、动态数组)，会使用到全局内存的空间就只有 4 byte 或 8 byte 的对象参考而已。

我们可以透过 Project|Information 这个选单项目来确定全局内存的大小，只需在编译完成之后，就可以看到这个数据所需的空间大小。在图 13.1 里面，显示了全局数据使用量约 50K(48,956 bytes)，这个用量是由我们的程序以及函式库所共同创造出来的。

图 13.1: 编译完成的程序相关信息



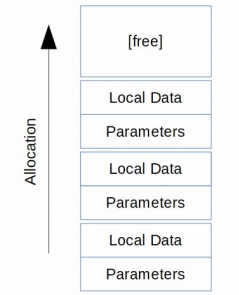
全局内存有时也被称为静态内存，因为一旦程序被加载了，这些变量所占用的空间就不会被从一开始的位置被搬走，也不会被释放，直到应用程序结束为止。

堆栈

堆栈是动态的内存区域，这个区域会依照后进先出的顺序来配置与释放。这意味着最后被配置的内存对象必须最先被删除掉。我们可以从图 13.2 看到内存堆栈的内容。

堆栈内存通常是由程序、函式以及方法使用到的，这些内存空间会用来储存传递的参数、回传的值，以及我们在函式或方法里面宣告的局部变量。一旦子程序执行完成，它在堆栈的内存空间就会被释放掉了。反正要记得，使用 Object Pascal 的默认缓存器呼叫转换功能，透过 CPU 缓存器来传递参数，是永远不可能的。

图 13.2: 内存堆栈区域的表示



也要注意到，堆栈内存一般是不会自动进行初始化也不会自动被清除的。这也是为什么如果我们在局部变量当中宣告了一个整数然后直接去读取这个变量的内容，我们不会知道里面会是什么数据。这也是为什么所有局部变量都必须先进行初始化，然后才能被使用。

堆栈的大小通常是固定的，并且在编译作业的过程就已经决定了。我们可以在项目选项的链接程序的设定选项中来设定这些参数。然而，默认值通常是 OK 的。如果我们看到了『stack overflow』这个错误讯息，这或许是因为我们的程序当中有不断呼叫自己的递归函式，而不是编译器指派的堆栈不够大。初始的堆栈大小是在 Project|Information 对话框里面的另一项信息。

Heap (堆)

Heap 是源码当中可以不依照顺序进行配置/释放作业的区域。这表示如果我们依序配置了三块内存，这三块内存可以在稍后用任何的顺序进行释放。Heap 管理程序会照看所有的细节，所以我们只要很简单的用低阶函式 GetMem 来要求新的内存，也可以透过呼叫建构函式来建立一个对象，然

后系统就会回传一个新的内存区块(也可能是从已经释放掉的内存当中回收使用的)。Object Pascal 使用 heap 来配置每个对象、字符串内容，动态数组，以及绝大多数数据结构的内存空间。

因为它是动态的，Heap 也是程序当中产生最大多数问题的内存区域:

- 每次一个对象只要被建立了，它就一定得被释放。没有被释放的内存空间被称为”Memory leak” (翻成内存泄漏)，这情形一般来说不会有太大的伤害，除非这个情形反复发生，而导致整个 Heap 都被用光了。
- 每次一个对象被释放了，我们必须确定被释放的对象已经没有在任何源码里面被使用了，当然，源码也不能对已经被释放的对象重复进行释放。
- 对任何动态建立的数据结构也完全一样，但编程语言的函式库会在大多数的时间自动在字符串与动态数组之后进行处理，所以我们不用担心其余的处理动作。

对象参考模型

一如我们在第七章里面介绍过的，Object Pascal 里面的对象是以参考来实作的。一个类别型别的变量里，只储存了指向该对象在 Heap 里面所使用的内存空间而已。当然还有一些额外的信息，像是类别参考，我们可以透过它来存取该对象的虚拟方法列表，但这不是本章的主题。(我会在第 13 章的『指针是对象参考吗?』那一节做个简短的介绍)

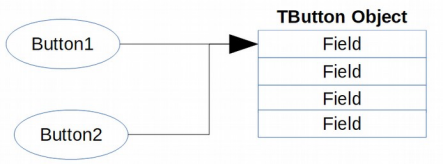
我们也已经介绍过如何把一个对象指派给另一个对象，只复制该对象的参考，所以我们会有两个参考指向同一个内存内的对象。如果要拥有两个完整的对象，我们必须要建立一个新的对象，然后把原本对象的内容复制到新对象去(这个复制的动作不会自动发生，因为实作的细节会依据不同的数据结构而需要不同的作法)。

用源码的思维来看，如果我们写出以下的源码，程序不会建立出新的对象，只会把内存里面的同一个对象的参考复制一份而已:

```
var
    Button2: TButton;
begin
     Button2 := Button1;
```

换句话说，内存里面只有一个对象，而有两个变量 Button1 跟 Button2 指向这个对象，请参考图 13.3 的示意。

图 13.3: 复制对象参考



把对象当成参数来传递

当我们把对象当成参数传递给一个函式或者方法的时候，其中的机制是很相似的。用一般的字眼来描述，我们只是把同一个对象的参考复制一份，然后在方法或函式中，我们就可以用这个对象来做任何处理了，当然也可以修改它，除非该参数是以 const 关键词来描述的。

举例来说，写一个像以下这段源码的程序，并呼叫它，我们可以变更 Button1 的文字卷标，如果我们喜欢用 AButton，也可以：

```
procedure ChangeCaption (AButton: TButton; Text: string);
begin
    AButton.Text := Text;
end;
// Call...
ChangeCaption (Button1, 'Hello')
```

假如我们需要一个新的对象呢，要怎么做?我们基本上得建立新对象，然后把每个属性的内容都复制过去。部分类别，只要是 TPesistent 类别的衍生类别，而不是从 TComponent 衍生出来的类别，就会定义有 Assign 这个方法可以用来复制一个对象的数据，举例来说，如果我们这么写:

```
ListBox1.Items.Assign (Memo1.Lines);
```

即使我们直接指派了这些属性，Object Pascal 会为我们执行类似的源码。事实上，SetItems 方法会把 items 属性跟 listbox 进行链接，靠的就是呼叫 TStringList 类别的 Assign 方法，TStringList 就是 ListBox 的实际项目。

所以，让我们试着回忆一下，当把对象当成参数传递时，不同的参数传递方法修饰词所做的:

- 如果参数上**没有修饰词**，我们就可以对这个对象以及指向这个对象的变量作任何处理。我们可以对原始对象作修改，但如果我们把一个新

的对象指派给参数，这个新的对象就无法作任何处理，当然指向它的变量也一样不行。

- 如果参数上面有 const **修饰词**，我们可以修改该对象里面的数据，也可以呼叫该对象的方法，但我们不能指派一个新的对象给这个参数。请注意，用 const 修饰词在效能上并没有比较好。
- 如果参数上面有 var **修饰词**，我们就可以修改该对象里面的任何数据，也可以用一个新的对象来取代原来传递进来的参数对象，就跟其他有 var 参数的型别一样。守则就是我们必须传递一个变量(不能传一个表达式)作为参数，且这个参数的变量型别必须跟宣告的型别一致。
- 最后，在把对象作为参数时有一个鲜为人知的选项，称为静态参考(const reference)，语法写为[ref] const. 当我们把一个对象用静态参考作为参数传递时，它的规则有点像是引用参数(var)，但在参数型别上有比较多的弹性，并没有要求完全一致的型别(参数可以是宣告的类别或者衍生类别都可以)。

内存管理小技巧

在 Object Pascal 里，内存管理是一个关于三个规则的议题:第一，我们必须建立每个对象，并配置所需的每一个内存区块。第二我们必须释放我们配置的每一个内存区块。第二，每个我们自己建立起来的对象跟内存区块都只能释放一次。Object Pascal 对动态元素支持三种形式的内存管理(也就是说，这些元素不在堆栈，也不在全局内存里面)，我们在本节其余的篇幅里面细细道来:

- 当我们建立了一个对象，我们就有负责把它释放掉的义务。如果我们没有做到，这个对象所使用的内存空间就不会被释放，也没有机会回收给其他对象使用，直到程序结束为止。
- 当我们建立一个组件，我们可以确立拥有这个组件的所有者，在建立组件的时候，把所有者当成该组件建构函式的参数。这个所有者对象(通常是窗体或是数据模块)就担负了对它所拥有的对象释放的责任，直到这个所有者对象被释放为止。换句话说，当我们释放该窗体(或是数据模块)，它所拥有的所有组件也会一起被释放掉。所以如果我们建立了一个新的组件，并且指派了该组件的所有者，我们就不用担心这个组件释放的问题了-但我们还是可以决定先释放该组件，这要感谢由 TComponent 类别提供的释放提醒机制。
- 当我们为字符串、动态数组跟被接口参考到的对象(请参考第 11 章的内容)配置了内存，Object Pascal 会在参考离开执行的范围时，自动把这些内存空间释放掉。我们不用释放字符串的空间，因为当字符串没

有再被使用到的时候，内存空间就被自动释放掉了。如果我们需要提前释放这些组件以取得多一点可用的内存，我们可以直接把字符串或者动态数组变量指派为 nil，或者指派空字符串到字符串变量即可。

释放我们建立的对象

在大多数单纯的情境里，我们得释放由我们建立的对象。任何非临时的对象都应该要有所有者，它可以是一个集合的成员，可以被一些数据结构存取，这些所有者在执行到要结束的时机，要担负释放这些对象的义务。

用来建立、释放临时对象的源码通常都会用 try-finally 区块来封装，这样可以保证就算在使用该对象的过程中发生问题，也确保可以把该对象给释放掉：

```
MyObj := TMyClass.Create;
try
    MyObj.DoSomething;
finally
     MyObj.Free;
end;
```

另一个常见的情境则是某个对象被另一个对象所使用，它就变成了这个对象的所有者：

```
constructor TMyOwner.Create;
begin
    FSubObj := TSubObject.Create;
end;
destructor TMyOnwer.Destroy;
begin
    FSubObj.Free;
end;
```

还有一些常见的情境复杂的多，万一对象在需要被使用的时候还没有被建立出来(延迟初始化)或比所有者更早被释放掉了(因为后面的源码里面不会再用到它)。

要实作延迟初始化的话，我们就不能在该对象的所有者的建构函式里面建立它，而得要在需要该对象的时候才来建立，如下面的程序片段所示:

```
function TMyOwner.GetSubObject: TSubObject
```

```
begin
    if not Assigned (FSubObj) then
        FSubObj := TSubObject.Create;
    Result := FSubObj;
end;
destructor TMyOnwer.Destroy;
begin
    FSubObj.Free;
end;
```

请留意，上述的情境只在一个情境下成立，就是该对象的内存空间有在使用前先进行初始化，执行后 FSubObj 会变成 nil。此时我们就不需要在释放之前检查该对象是否已经被建立了，因为这动作在 Free 方法里面就会做了，我们会在下一节看到这一点。

只能把对象释放一次

另一个问题就是我们如果呼叫了一个对象的解构函式两次，我们就会导致错误。*解构函式*是用来释放一个对象所使用的内存空间的方法。我们可以为解构函式写一些源码，通常是覆写预设的 Destroy 解构函式，好让对象在被释放掉之前还能执行这些源码。

Destroy 是一个属于 TObject 的虚拟解构函式。大多数需要在对象被释放前进行自定清理动作的类别，都会覆写这个虚拟方法。

我们之所以不用定义一个新的解构函式，是因为大硕的对象通常都会呼叫 Free 方法来释放对象，而 Free 这个方法会帮我们呼叫 Destroy 这个虚拟解构函式(也可能是覆写后的版本)。

刚刚我们有提到过，Free 是 TObject 类别的一个方法，几乎所有的类别都会继承它。Free 方法基本上会检查该对象(self)，确定它不是 nil，然后才会呼叫 Destroy 虚拟解构函式。

笔记 您可能会觉得奇怪，为什么当对象参考是 nil 的时候，还可以安全的呼叫它的 Free 方法，但我们却不能直接呼叫 Destroy 方法?理由是 Free 方法是位于已知的内存空间，而虚拟函式 Destroy 是在运行时间依照该对象的型别进行搜寻以后决定的，搜寻的结果有可能会是对象不存在，那这个动作就非常危险了。

以下是 Free 方法的虚构源码:

```
procedure TObject.Free;
 begin
    if Self <> nil then
Destroy;
end;
```

接下来，我们可以把注意力移到 Assigned 函式了。当我们把一个指标传递给这个函式的时候，这个函式会单纯的检查指标内容是不是 nil。所以以下两个叙述句是相同的(至少在大多数情形下是如此):

```
if Assigned (MyObj) then ...
if MyObj <> nil then ...
```

注意到这些叙述句只会在指标内容不是 nil 的时候进行检查，如果指标内容是不合法的，就不会进行检查的动作了。如果我们这么写:

```
MyObj.Free;
if MyObj <> nil then
   MyObj.DoSomething;
```

检查的动作会通过喔，也就是说 MyObj 的内容不会是 nil，所以接下来的那行指令就会出错了，因为我们呼叫了一个已经被释放的对象的方法。要知道，呼叫了 Free 方法之后，该对象的指针并不会被设为 nil 的。

要自动把一个对象设成 nil 是不可能的。我们可能会有很多个参考指向同一个对象。同时，我们可以在方法当中对一个对象进行处理，但我们并不知道该对象的参考-该变量的内存地址(我们会用来呼叫这个方法)。

换句话说，在 Free 方法的内部，或一个类别当中任何其他的方法，我们可以知道该对象(self)的内存地址，但我们不会知道参考到这个对象的变量的内存地址，例如 ToDestroy。因此，Free 方法不会动到 MyObj 变量的内存内容，只会释放 MyObj 所指向的对象。

然而，当我们呼叫一个外部程序把对象作为引用参数传递时，这个函式可以异动该参数的原始内容。这恰恰就是 FreeAndNil 程序所做的事，以下是 FreeAndNil 函式里面的源码：

```
procedure FreeAndNil(const [ref] Obj: TObject); inline;
var
```

```
    Temp: TObject;
begin
    Temp := Obj;
    TObject(Pointer(@(Obj)^) := nil;
    Temp.Free;
end;
```

过去，参数只是一个指标，但缺点就是我们也可以把一个完全不是对象的指针也传递给 FreeAndNil 程序，如果这指针指向的是一个接口，或者其他不兼容的数据结构，就可能导致内存冲突或者很难抓出来的程序问题。从 Delphi 10.4 开始，FreeAndNil 改成了刚刚范例程序的写法，使用 const 修饰词，这作法就可以限制传入的参数必须是 TObject 型别(或其衍生类别)。

笔记 部分 Delphi 的专家可能会争论说 FreeAndNil 几乎从来没有使用过，因为参考到对象的变量几乎跟它的生命周期一样。如果对象中还有一个对象，而释放的时间点也确定是在解构函式当中，那其实可以不用特地把指标设定为 nil，因为后面的源码已经确定不会再用到它了。同样的，在 try-finally 里面释放的局部变量，也无须再去把它设定为 nil，因为函式已经结束了，不会有机会再使用它了。

当做是个小笔记，除了 Free 方法之外，TObject 还有个 DisposeOf 方法可以用来释放对象，这是前几年为了支持 ARC 功能的时候所制作的。目前 DisposeOf 方法就是直接呼叫 Free 了。

整理一下，以下是几个准则:

- 一定要呼叫 Free 来释放对象，不要呼叫 Destroy 解构函式。
- 使用 FreeAndNil，或在呼叫 Free 之后要把该变量的内容设成 nil，除非参考在该函式里面是最后一行，Free 完以后就没有任何源码用到该对象了。

内存管理与接口

在第 11 章里面，我们介绍了接口内存管理的关键要素，它跟对象不同，是受系统管理，且使用参考计算器制管理的。就像之前介绍过的，接口参考会增加被参考对象的计数数字，但我们可以用 weak 宣告接口参考，这个作法可以让被参考的对象计数数字不被异动(但是仍然会要求编译器管理这个参考)，或者也可以使用 unsafe 修饰词来让编译器完全不管理这个被宣告

的参考变量。在这一节里，我们会稍微深入一点，用第 11 章里面提过的范例再做些延伸。

更深入 Weak 参考

Delphi 使用接口参考计数模式，当中有一个议题是: 如果两个对象互相参考，这情形称之为循环参考，此时它们的参考计数基本上永远不会变成 0。Weak 参考提供了一个机制来打破这样的循环，让我们可以定义一个参考，而不增加这个参考的计数。

假设两个对象互相参考，把对方储存在自己的数据字段中，并用一个外部变量把第一个对象储存起来。此时第一个对象的参考计数将会是 2(外部变量，以及另一个对象的数据字段各占 1):而第二个对象的参考计数将会是 1(第一个物件的数据字段)。图 13.4 描绘了这个情境。

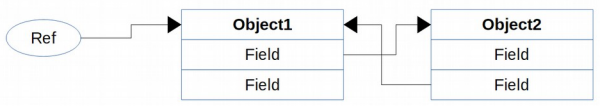


图 13.4: 对象之间的参考可能发生循环，weak 参考可以处理这个情形

现在，当外部变量离开执行范围时，两个对象的参考计数都是 1，因为主要对象 Object1 拥有 Object2，并没有外部的所有者，于是这两个对象就会永远留在内存里面了。为了解决这一类的问题，我们需要打破其中的循环参考，要达到这一点并不简单，因为我们并不知道何时可以进行这项作业(这项作业应该要在最后一个外部连结参考脱离执行范围时处理，但两个循环参考的对象则无从得知这个条件是否发生了)。解决这个情况以及类似情况的方法，就是使用 weak 参考。一如我们所描述的，weak 参考是一种当对象参考被指派进来时，不会对该对象的参考计数进行递增的特殊参考。技术上来说，我们只需要在该参考上面加上[weak]标注即可。

笔记 标注是进阶的 Object Pascal 语言功能之一，我们在第 16 章会介绍到这个功能，我只想说，这是对特定符号在运行时间当中加上一些信息的方法，这样一来，外部的源码就可以决定如何处理它。

接续前一个范例，如果第二个对象中指回第一个对象的参考是 weak 参考(请参阅图 13.5)，当外部变量离开了执行范围，两个对象就都可以被释放掉了。

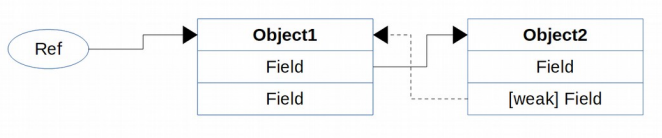


图 13.5: 在图 13.4 里面的循环参考，因为使用了 weak 参考而被打破了(虚线表示)

我们来看一下这个情形的实作源码。在 ArcExperiments 范例程序中宣告了两个接口，其中一个接口会参考到另一个:

```
type
    IMySimpleInterface = interface
        ['{B6AB548A-55A1-4D0F-A2C5-726733C33708}']
        procedure DoSomething(bRaise: Boolean = False);
        function RefCount: Integer;
    end;
    IMyComplexInterface = interface
      ['{5E8F7B29-3270-44FC-B0FC-A69498DA4C20}']
      function GetSimple: IMySimpleInterface;
      function RefCount: Integer;
    end;
```

源码则定义了两个不同的类别，各自实作了对应的接口。请留意当中互相参考的情况(FOwnedBy 跟 FSimple 都是接口型别的变量，其中一个我们标注为 weak):

```
type
    TMySimpleClass = class (TInterfacedObject, IMySimpleInterface)
    private
        [Weak] FOwnedBy: TMyComplexInterface;
    public
        constructor Create(Owner: IMyComplexInterface);
        destructor Destroy (); override;
        procedure DoSomething(bRaise: Boolean = False);
        function RefCount: Integer;
    end;
    TMyComplexClass = class (TInterfacedObject, IMyMyComplexInterfce)
    private
        FSimple: IMySimpleInterface;
    public
```

```
        constructor Create();
        destructor Destroy (); override;
        class procedure CreateSimple: IMySimpleInterface;
        function RefCount: Integer;
    end;
```

以下是 TMyComplexClass 类别的建构函式，用来建立另一个类别的对象:

```
constructor TMyComplexClass.Create;
begin
    inherited Create;
    FSimple := TMySimpleClass.Create(self);
end;
```

记得 FOwnedBy 字段是一个 weak 参考，所以它不会增加所指向的对象的参考计数，在这个例子中，就是指当前的对象 self。因为这样的类别结构，我们可以把源码这么写:

```
class procedure TMyComplexClass.CreateOnly;
var
    MyComplex: IMyComplexInterface;
begin
    MyComplex := TMyComplexClass.Create;
    MyComplex.FSimple.DoSomething;
end;
```

这么做不会造成内存泄漏，因为 weak 参考已经被适当的使用了。例如:

```
var
    MyComplex: IMyComplexInterface;
begin
    MyComplex := TMyComplexClass.Create;
    Log('Complex = ' + MyComplex.RefCount.ToString);
    MyComplex.FSimple.DoSomething (False);
end;
```

这么一来每个建构函式跟解构函式都会在执行时记录下来，我们会看到如下的 Log:

```
Complex class created
Simple class created
Complex = 1
Simple class doing something
```

```
Complex class destroyed
Simple class destroyed
```

如果我们把宣告那段程序中的 weak 属性拿掉，执行上面这段源码的时候就会发生内存泄漏，当中的参考计数值就会是 2，而不是像上面这个执行结果中的 1 了。

Weak 参考是受到管理的

有个重要的概念要记得，就是 weak 参考是受到管理的。换句话说，系统会在内存里面维护一份 weak 参考的列表，当对象被释放的时候，系统会检查是否有 weak 参考指向它，并且适当的进行标示。这表示 weak 参考在运行时间是有额外的代价的，但话说回来，我们自己在源码里面做这样的交叉检测，代价一定更高。

具备这种受管理的 weak 参考，跟传统的作法相较之下，好处是我们可以检测一个接口参考是否还是可用的(意指所指向的对象是否已经被释放了)。但也这表示，当我们使用 weak 参考，我们应该在使用它之前永远要先检查它是否已经有被指派内容了。

在 ArcExperiments 范例里面，窗体有一个型别为 IMySimpleInterface 的私有字段，宣告为 weak 参考:

```
  private
    [weak] MySimple: IMySimpleInterface;
```

里面有一个按钮点击事件会指派该字段，另一个按钮则会使用它，但使用前会确认该对象是否仍然是可用的:

```
procedure TForm3.BtnGetWeakClick(Sender: TObject);
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.GetSimple.DoSomething (False);
  MySimple := MyComplex.GetSimple;
end;
procedure TForm3.BtnUseWeakClick(Sender: TObject);
begin
  if Assigned (MySimple) then
```

```
        MySimple.DoSomething(False)
    else
        Log ('Nil weak reference');
end;
```

除非我们修改了上面的源码，否则当中的 if Assigned 检测结果一定是错误的，因为第一个按钮的事件处理程序建立了对象后就把对象释放掉了，所以 weak 参考的内容就变为 nil(在此案例中会是有错误的)。但如果它是受管理的状态，编译器就会帮我们追踪该对象的实际状态(跟对象的参考不一样)。

不安全的标注 (The unsafe Attribute)

有些情况是很特殊的(例如在一个实体建立的过程中)，在这些特别的情形中，可能是一个函式要把一个参考计数为 0 的对象刚建立出来，回传给另一段源码使用。在这个案例中，为了避免编译器马上把该对象删除掉(在源码还没来得及把建立出来的对象指派给变量之前，可能就得先把对象参考增加为 1 才行)，我们必须把这个对象标示为”unsafe”。

这里的含意是，该对象的参考计数必须先暂时被忽略，才能让源码安全(safe)。这个行为是透过使用新的特殊标注”[unsafe]”来达成的。这个功能我们应该指会在很特别的情形中才需要用到。以下是范例的语法:

```
var
    [Unsafe] Intf1:   IInterface;
    [Result: Unsafe] function GetIntf: IInterface;
```

这个标注的使用在实作建构模式的时候是很有用的。建构模式跟工厂模式一样，在一般用途的函式库当中都会用到。

笔记 为了支持目前已经弃用的 ARC 内存模式，System 单元文件里面使用了 unsafe 关键词，因为在定义宣告之前(在同一个单元文件比较后段的部份)它是无法被使用的。这并非假设要使用该单元文件之外的其他源码，而且 ARC 模式已经完全不再被使用了(我们可以看到 System 单元里面它已经被用 $IFDEF 做了限制)

追踪与检测内存

在这一章，我们已经看过了 Object Pascal 里面内存管理的基础。在大多数的案例中，只要遵循在本章所提到的规则，就已经足够让整个程序维持稳定、避免过度使用内存，并基本上让我们可以不用去管内存管理。在本章后段还有一些进阶的实例，可以用来撰写强健的应用程序。

在这一节里面，我们要专注在可以用来追踪内存使用的技术，监控异常状况，并且找出内存泄漏的情形。这对开发人员来说是很重要的，即使这并非编程语言的一部分，而比较属于运行时间函式库的支持。同时，实作内存管理也是跟目标平台与操作系统息息相关的，我们是可以在 Object Pascal 应用程序当中外挂一个自定的内存管理员的(这个作法目前已经很普遍了)。

留意到，我们目前所有的讨论都是关于追踪内存状态、内存管理、泄漏状况侦测则只跟 heap 内存有关。堆栈跟全局内存的管理是不同的，且我们没有办法干预，但这也是内存区域中不常发生问题的部分。

内存状态

要怎么追踪 heap 内存的状态?在 RTL 里面提供了一些有用的函式：GetMemoryManagerState 跟 GetMemoryMap。内存管理员的状态是指不同大小的配置区块数目，heap 地图则相当友善的在系统阶层描绘出应用程序的内存状态。

我们可以用以下的源码来检测每个内存区块的实际状态:

```
for I := low(aMemoryMap) to high(aMemoryMap) do
begin
    case AMemoryMap[I] of
        csUnallocated: ...
        csAllocated: ...
        csReserved: ...
        csSysAllocated: ...
        csSysReserved: ...
    end;
end;
```

这段源码在 ShowMemory 范例项目中用来建立应用程序当时使用内存状态的图形表示。

FastMM4

在 Windows 平台上，目前的 Object Pascal 内存管理名为 FastMM4，是主要由 Pierre La Riche 所领导的开放原始码项目开发出来的。在 Windows 之外的其他平台上，Delphi 则是使用各平台上原生的内存管理程序。

FastMM4 优化了内存的配置，加速、并节省了许多内存的使用量。FastMM4 可以透过许多精简的内存清除程序，来达成许多进阶的内存检测，包含对已经删除的对象进行不正确的使用，包含透过接口存取数据、在内存上覆写，以及在缓冲区溢出等。它也对遗留在内存中的对象提供了一些响应，帮助我们追踪内存泄漏的状况。

FastMM4 的一些进阶功能，只能在完整版的函式库中执行(我们在『在完整版 FastMM4 上的缓冲区溢出』加以介绍)，标准版的 RTL 里面并没有提供这些功能。这也是为什么我们会需要完整版的原因了，我们可以从这里下载完整版的原始码:

```
https://github.com/pleriche/FastMM4
```

笔记 这个函式库有个新版本叫做 FastMM5，这个版本中特别对多线程应用程序做了优化，在多核心的系统中执行起来表现也更好。这个新版的函式库是使用 GPL 授权(开源项目的授权)，也有付费的版本(真的值得花这个钱)，请参考该项目的介绍: https://github.com/pleriche/FastMM5

追踪泄漏以及其他全局设定

RTL 版本的 FastMM4 可以调整为使用 System 单元里面的全局设定。留意到，相关的全局设定是在 System 单元里，实际上的内存管理模块则是在 getmem.inc 这个 RTL 原始码档案里面。

更容易使用的设定是 ReportMemoryLeaksOnShutdown 全局变量，这个变量让我们更容易追踪内存泄漏的情形。我们需要在程序执行前就启用它，在程序结束之后，它会告诉我们在执行过的源码(或者我们使用的任何函式库)里面有没有发生任何内存泄漏的情况。

笔记 更进阶的内存管理模块设定，包含 NeverSleepOnMMThreadContention 全局变量，可以用来处理多线程的配置; GetMinimumBlockAlignment 跟 SetMinimumBlockAlignment 函式则可以加速一些 SSE 的处理，并降低使用更多内存的代价;全局程序 RegisterExpectedMemoryLeak 则提供了注册预期发生内存泄漏情形的能力。

为了展示标准的内存泄漏回报与注册功能，我写了简单的 LeakTest 范例项目。在这个项目中有一个按钮，它的事件处理程序源码如下:

```
var
    p: Pointer;
begin
    GetMem (p, 100); //Memory leak!
end;
```

这段源码会配置 100 bytes，然后这 100 bytes 的空间没有被释放，所以算是泄漏了。如果我们透过 IDE 来执行 LeakTest 项目程序，假设我们在第一个按钮上点击一次，结束程序的时候，就会看到 IDE 回报了如图 13.6 的错误讯息。

另一个由程序产生的内存泄漏，是由于建立了一个 TButton 并且把它遗留在内存而导致的，但这个对象包含了很多子元素，所以泄漏回报讯息变得很复杂，就像图 13.6 底部的窗口画面。同样的，我们能取得的关于泄漏本身的信息相当有限。

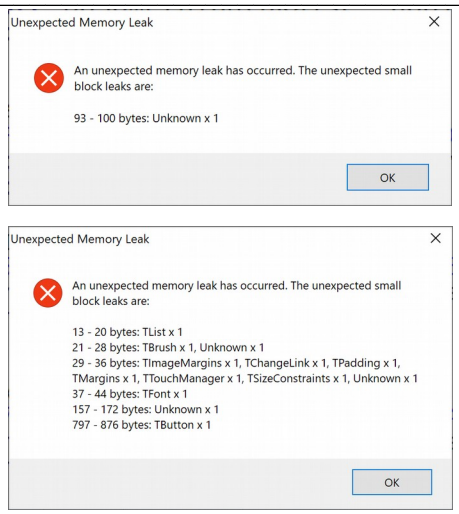


图 13.6: 在 Windows 平台上当程序结束时，由内存管理模块回报的内存泄漏警告

这个程序也为全局指针配置了一些内存空间，这些空间也不会被释放掉，但会把它注册成可预期的泄漏，所以不会得到系统回报警告讯息：

```
procedure TFormLeakTest.FormCreate(Sender: TObject);
begin
    GetMem (GlobalPointer, 200);
    RegisterExpectedMemoryLeak(GlobalPointer);
end;
```

再次强调，这个基本的内存泄漏回报机制，预设只有在 Windows 平台可以使用，因为只有在 Windows 平台上，预设是使用 FastMM4 这个内存管理模块的。

在完整版 FastMM4 上的缓冲区溢出

这个主题相对比较进阶，且特别限定只有 Windows 平台能用，所以我只建议非常有经验的开发人员阅读这一节。

如果我们想要对内存泄漏有更多的控制(像是启用以档案进行记录的功能)，调整配置策略以及使用 FastMM4 提供的内存检测功能，我们就需要下载完整版。包含了 FastMM4.pas 跟其配置文件 FastMM4Options.inc。

我们稍后会需要编辑后者(FastMM4Options.inc)来调整设定，只需简单的把一些设定批注掉，或者取消批注一些设定即可。依照惯例，我们只需要在 $DEFINE 指令前面加上一个.就能够把该设定批注掉，以下是我们从 inc 档案里面拿前两行来当成范例:

```
{.$DEFINE Align16Bytes} // comment
{$DEFINE UseCustomFixedSizeMoveRoutines} // Active setting
```

为了示范，我已经修改了下列相关的设定，在这里进行回报，让我们知道有哪些可用的定义：

```
{$DEFINE FullDebugMode}
{$DEFINE LogErrorsToFile}
{$DEFINE EnableMemoryLeakReporting}
{$DEFINE HideExpectedLeaksRegisteredByPointer}
{$DEFINE RequireDebuggerPresenceForLeakReporting}
```

测试程序(在 FastMMCode 文件夹里面，里面也包含了我有用到的完整版的 FastMM4，方便读者不用再去另外下载)启用了项目源码里面的客制化版本内存管理模块，只需把它设定成第一个单元文件即可：

```
program FastMMCode;
uses
  FastMM4 in 'FastMM4.pas',
  Forms,
  FastMMForm in 'FastMMForm.pas'; {Form1}
```

我们也需要把 FastMM_FullDebugMode.dll 复制一份到测试程序的目录，这样才能正常运作。测试程序会在缓冲区里面填入超过缓冲区大小的数据来导致一个缓冲区溢出的情况，因为 Length(Caption)大于提供的五个字符：

```
procedure TForm1.Button2Click(Sender: TObject);
var
    pch1: PChar;
begin
    GetMem (pch1, 5);
    GetWindowText(Handle, pch1, Length(Caption));
    ShowMessage (pch1);
    FreeMem (pch1);
end;
```

内存管理模块一开始会配置额外的空间，在每个内存区块的结尾会放特殊的值，在释放每个内存区块的时候也就会检查这个特殊值。这也是为何我们在呼叫 FreeMem 函式的时候会产生错误。当我们点击按钮(在侦错模式下)，我们会看到很长的一串错误讯息，这串讯息也会记录在以下这个档案里面:

```
FastMMCode_MemoryManager_EventLog.txt
```

以下是缓冲区溢出错误的讯息内容，包含了当时对堆栈的追踪，以及配置与释放的动作，再加上把当前堆栈内存直接倒出的内容(只列出一部分)：

```
FastMM has detected an error during a FreeMem operation. The block
footer has been corrupted.
The block size is: 5
Stack trace of when this block was allocated (return addresses):
40305E [System][System.@GetMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TWinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TWinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]
The block is currently used for an object of class: Unknown
```

```
The allocation number is: 381
Stack trace of when the block was previously freed (return addresses):
40307A [System][System.@FreeMem]
42DB8A [StdCtrls][StdCtrls.TButton.CreateWnd]
443863 [Controls][Controls.TWinControl.UpdateShowing]
44392B [Controls][Controls.TWinControl.UpdateControlState]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
44009F [Controls][Controls.TControl.Perform]
43ECDF [Controls][Controls.TControl.SetVisible]
45F770
76743833 [BaseThreadInitThunk]
The current stack trace leading to this error (return addresses):
40307A [System][System.@FreeMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TWinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TWinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]
Current memory dump of 256 bytes starting at pointer address 133DEF8:
46 61 73 74 4D 4D 43 6F 64 [... omitted...]
```

这看上去并不算很直觉，但它提供的信息应该已经足够让我们找出问题。

请注意，如果在内存管理模块里面没有这些设定，我们基本上是不会看到任何错误的喔，程序也会继续执行下去。虽然我们可能会发现不定时出现问题，也只会当缓冲区溢出的情形影响到了我们用来储存数据的内存区块，错误才会发生。此时，我们会看到程序执行的结果很奇怪，而且很难找到问题发生的原因。

举例来说，我曾经看到过原本用来储存类别参考的内存，当中有一部分是被初始化数据给覆写掉了。由于这种内存数据损坏，类别会变成未被定义，且每一次呼叫它的虚拟方法时，可能都会导致程序崩溃…这很难去联想到是在跟执行中的源码完全不同的区域对内存写入的动作有关。

在 Windows 以外的平台的内存管理

想象一下在 Object Pascal 的编译器是怎么进行内存管理的作业，就值得思考一下我们可以用的一些作法，来确定在 Windows 以外的操作系统中，所有事情都在控制下。在我们继续之前，很重要的一点，是在 Windows 以外的操作系统中，Delphi 并没有使用 FastMM4 内存管理模块，所以设定 ReportMemoryLeaksOnShutdown 这个全局选项，想要在程序关闭的时候侦测是否发生内存泄漏的情形是没用的，另一个理由是在行动装置上，关闭程序之后，程序并不是直接就被关闭了，因为 app 会在内存里面继续占用着，直到被用户或操作系统强制移除，内存空间才会被释放掉。

在 macOS, iOS 跟 Android 平台上，Object Pascal RTL 会直接呼叫原生 libc 函式库的 malloc 跟 free 函式。所以要在这些平台上监测内存的使用，就必须靠额外的平台工具了。例如在 iOS 跟 macOS 上面，我们得要用 Apple 提供，附在 Xcode 里面的 Instruments 工具程序，它会全面性的监测我们在实体装置上执行的应用程序。

追踪每个类别的配置

最后，Object Pascal 有一个特殊的功能可以用来追踪特定的类别，而不用对整个内存全部追踪。事实上，对单一对象的内存配置会发生在 NewInstance 这个虚拟类别方法被呼叫的时候，而会在 FreeInstance 虚拟方法被呼叫的时候进行清除。这是我们可以对特定类别进行覆写的虚拟方法来客制特定的内存配置策略。

好处是我们透过这个作法可以不用管建构函式(因为一个类别可能有多个建构函式)与解构函式，从标准的对象初始化跟结束化源码明确的把内存追踪的源码分离出来。

虽然这是一个比较极端的案例(可能只有在一些比较大型的内存结构中才值得这样做)，我们可以覆写这些方法，用以在建立后与释放后计算对象的数量或特定类别的数量，计算使用中的实体，并在最后确认对象的数量是否如预期的变成 0 了。

撰写强健的应用程序

在这一章里面，我们已经介绍了相当多的计数可以用来撰写强健的应用程序，以及适当的管理内存配置与释放。

在专心介绍内存管理这个章节的最后一节里，我决定要列出一些稍微进阶的主题，为之前讨论过的内容进行一些延伸。即使使用 try-finally 区块跟呼叫解构函式之前已经介绍过了，但在本章强调的情境里面显然更为复杂，并引入了更多编程语言的功能一起运作。

这并不全然是进阶的一节，但所有 Object Pascal 开发人员应该真正掌握它，才能写出强健的应用程序。只有上一节关于指针跟对象参考的部份算是真的很进阶的主题，因为它深入到一个对象跟类别参考的内部存储器结构。

建构函式，解构函式，以及例外

建构函式跟结构函式是正确内存管理的有利工具，但也可能是应用程序里面的问题来源。虚拟建构函式必须一定要先呼叫基础类别的建构函式(透过 inherit 来呼叫)。而解构函式应该在最后才呼叫基础类别的解构函式(也是透过 inherit 来呼叫的)。

笔记 要遵循好的程序实务，我们应该要在我们写的每一个 Object Pascal 源码的建构函式中记得要呼叫基础类别的建构函式，即使这并不是强迫性的，而且额外的呼叫可能不一定有用(像是呼叫 TObject.Create)

在这一节里面，我想要特别强调在类别的建构函式出错时的状况，像是:

```
MyObj := TMyClass.Create;
try
    MyObj.DoSomething;
finally
    MyObj.Free;
end;
```

如果这个对象被建立了，并被指派给 MyObj 变量，finally 区块就会处理释放对象的动作，但如果 Create 动作触发了例外，程序就不会进入 try-finally 区块了(这是对的!)。当一个建构函式触发例外时，此时对象只完成了部分初始化动作，对应的解构函式源码就会被自动执行。举例来说，如果在建构函式中建立了两个子对象，它们就需要透过呼叫相对的解构函式加以清

除。然而这可能导致一个潜在的问题，如果在解构函式里面我们先假设了所有子对象都有被完整初始化的话……

从理论上来看，这并不容易理解，所以我们来看一个实际的源码当例子。SafeCode 这个范例项目包含了一个类别，在这个类别中有建构函式跟解构函式，通常这两个函式都会正常运作，除非建构函式本身出了错：

```
type
    TUnsafeDesctructor = class
    private
        aList: TList;
    public
        constructor Create (positiveNumber: Integer);
        destructor Destroy; override;
end;
constructor TUnsafeDesctructor.Create(positiveNumber: Integer);
begin
     inherited Create;
     if positiveNumber <= 0 then
        raise Exception.Create ('Not a positive number');
     aList := TList.Create;
end;
destructor TUnsafeDesctructor.Destroy;
begin
    aList.Clear;
    aList.Free;
    inherited;
end;
```

在对象已经完全建立时，不会有什么问题，但当 FList 字段内容还是 nil 的时候，如果传递一个负数给建构函式就会发生问题了。在这个情形下，建构函式会发生错误，解构函式也会被触发，企图对 FList 呼叫 Clear 方法，FList 在建构函式被呼叫前，就已经被初始化成 nil 了。想要呼叫 nil 的 Clear 方法，当然会发生『违规存取』(Access Violation)的例外。

这个解构函式比较安全的写法如下:

```
destructor TUnsafeDesctructor.Destroy;
begin
    if assigned (aList) then
```

```
    aList.Clear;
  aList.Free;
  inherited;
end;
```

而这个例子的寓意，是我们永远不能在解构函式中假设对应的建构函式已经把所有子对象完整的建立完成了。我们可以在类别中的任何一个方法里面做这个前提假设，但就是不能在解构函式里面这样假设。

巢状 Finally 区块

Finally 区块是让我们的源码保持安全状态的技术中最重要，也最常见的。我也不觉得这是进阶的主题，但我们有把 finally 在该用的地方都用上吗?且我们们是否都在特定的案例中适当的使用了 finally 区块呢?例如巢状处理，或者我们是否有把多个完成源码组合在一个 finally 区块里面呢?这样可不是完美源码会做的事:

```
procedure TForm1.btnTryFClick(Sender: TObject);
var
    A1, A2: TAClass;
begin
    A1 := TAClass.Create;
    A2 := TAClass.Create;
    try
        A1.whatever := 'one';
        A2.whatever := 'two';
    finally
          A2.Free;
          A1.Free;
    end;
end;
```

以下才是比较保险也比较正确的写法(也是节录自 SafeCode 范例项目):

```
procedure TForm1.btnTryFClick(Sender: TObject);
var
  A1, A2: TAClass;
begin
  A1 := TAClass.Create;
  try
```

```
      A2 := TAClass.Create;
      try
        A1.whatever := 'One';
        A2.whatever := 'Two';
      finally
        A2.Free;
      end;
    finally
      A1.Free;
    end;
end;
```

动态型别检查

在型别之间进行动态的转换是常见的处理，且类别型别也可能是另一种错误的发生来源。特别是如果我们没有使用 is 或 as 运算符号，直接硬着进行型别转换的时候。事实上，每一个直接型别转换的动作就是潜在的原始码错误来源(除非进行转换之前有先做过 is 检查)。

从对象到指针的型别转换，转换成或转换自类别参考，从对象到接口，转换自字符串或者转换成字符串都是极度具有潜在危险性的，但在特别的情形下却难以避免。举例来说，我们可能希望把对象参考存放在一个组件的 Tag 属性里，Tag 属性是一个整数，所以我们可以依照我们自己的设计直接做型别转换。另一个例子则是当我们把对象存放在一个指针列表里面，一个旧型的 TList(并不是型别安全的泛型列表，我们在下一章里面会介绍)。

这是一个相当笨的范例：

```
procedure TForm1.btnCastClick(Sender: TObject);
var
   List: TList;
begin
  List := TList.Create;
  try
    List.Add(Pointer(Sender));
    List.Add(Pointer (23422));
    // direct cast
    TButton(List[0]).Caption := 'Ouch';
    TButton(List[1]).Caption := 'Ouch';
```

```
    finally
        List.Free;
    end;
end;
```

执行上面的源码通常会触发违规存取。

笔记

我写成通用的情形，因为当我们随机存取内存的时候，我们永远没有机会知道实际的效果。有时候程序会简单的重写内存，不会导致立即的错误……但我们接下来会很难搞清楚为什么其他数据会被破坏。

我们应该在任何时候都要避免类似的情形，但如果我们突然没有其他的替代方案，那要怎么修正这份源码?比较直觉的方法是使用 as 这种安全型别转换，或者透过 is 做型别检测，像以下的程序片段所做的:

```
// “as” cast
(TObject(list[0]) as TButton).Caption := 'Ouch';
(TObject(list[1]) as TButton).Caption := 'Ouch';

// “is” cast
if TObject(list[0]) is TButton then
   TButton(list[0]).Caption := 'Ouch';
if TObject(list[1]) is TButton then
   TButton(list[1]).Caption := 'Ouch';
```

然而，这并不是解决方法，我们还是会继续发生***违规存取***。问题在于 is 跟 as 最终还是会呼叫 TObject.InheritsFrom，这是另一个动作，会被执行好几次。

那解决方法呢?实际的解决方法是避免类似的情形一开始就发生(这种写法真的没什么意义)，使用 TObjectList 或者其他的安全机制(我们在下一章会介绍泛型的容器类别)。如果我们真的需要低阶对系统进行处理，像是跟指标深入处理，我们可以试着检查一个特定的『数值』是否是一个对象的参考。不过，这个动作并不直觉。它也有趣味的一面，这也是我拿来为范例说项的点，要为大家说明一个对象内部的结构，当然也适用在类别参考啦。

这个指针是对象参考吗?

这一小节说明了对象跟类别参考的内部结构，并超越了本书绝大部分的章节所涉及的程度。它可以提供一些有趣的内部观点给更多专业的读者，所以我决定保留这些内容，这些是过去我曾为内存管理所写的一篇进阶的文章的内容。也要留意到，以下的实作有部分是只能在 Windows 上面使用的，尤其是内存检测的部份。

有些时候，我们会很常使用指标(指标只是一个数值，这个数值代表储存某些数据的物理内存地址)。这些指针可能实际上是某些对象的参考，我们通常会知道指标指向的是什么，以及要把它们拿来做什么。但每当我们要做低阶的型别转换，我们可能随时位于搞砸整个程序的边缘。有些技术可以让这样的指标管理稍微安全一点，但不是百分之百保证安全。

我们可能要使用指标之前，考虑这个作法的出发点会是『它是否是一个合法的指标』?Assigned 函式只会检测一个指标是否为 nil，而在这样的情境里面并没有实质帮助。然而，很少人知道在 Object Pascal 里面有个名为 FindHInstance 的函式(在 System 单元文件里面，只有 Windows 平台可用)，会回传包含该函式的参数对象在 heap 当中的基础地址，如果该指针所指的是一个不合法的分页，就会回传 0(避免不常发生，但真的很难侦测的内存页面错误问题发生)。如果我们随机抓了一个数字，想要用这个数字当成内存地址来取得内存里面的数据，绝大多数的情形都不会抓到合法的内存的。

这是个很好的开始，但我们可以做的更好，因为如果这个值是一个字符串参考或任何有效的指针，而不是对象参考，就没有任何帮助。现在我们要怎么知道一个指针是否是一个对象的参考呢? 我已经提出以下的实证。每个对象为首的 4 个 bytes 是其类别的指标。如果我们想得知一个类别参考的内部数据结构，它是位于类别参考的 vmtSelfPtr 地址，当中是一个只到自己的指标。我们可以透过图 13.7 来表示：

图 13.7: 对象与类别参考的内部结构表示

换句话说，从类别参考指针的 vmtSelfPtr 位回溯到内存地址(这是在内存中向前寻找，是在比较前面的地址)，我们应该可以找到相同类别参考的指标才对。而且在类别参考的内部数据结构中，我们可以取得实体大小的信息

(在 vmtInstacneSize 的位置)以及看到是否有合理的数字。以下是实际的源码:

```
function IsPointerToObject (Address: Pointer): Boolean;
var
  ClassPointer, VmtPointer: PByte;
  InstSize: Integer;
begin
  Result := False;
  if (FindHInstance (Address) > 0) then
  begin
    Vmtpointer := PByte(Address^);
    Classpointer := Vmtpointer + VmtSelfPtr;
    if Assigned (Vmtpointer) and
      (FindHInstance (Vmtpointer) > 0) then
    begin
      instsize := (PInteger(
        Vmtpointer + VmtInstanceSize))^;
      // check self pointer and "reasonable" instance size
      if (Pointer(Pointer(classpointer)^) =
            Pointer(Vmtpointer)) and
          (InstSize > 0) and (InstSize < 10000) then
        Result := True;
    end;
  end;
end;
```

笔记 这个函式有很高的机会回传正确的值，但并非百分之百笃定。不幸的是，在以内存以随机数据通过测试时发生。

透过这个函式，在前一个 SafeCode 范例项目，我们可以在进行安全型别转换前加上一个指针-到-对象的检查：

```
if IsPointerToObject (List[0]) then
  (TObject(list[0]) as TButton).Caption := Ouch';
if IsPointerToObject (List[1]) then
  (TObject(list[1]) as TButton).Caption := 'Ouch';
```

相同的概念也可以直接应用在类别参考上，也可以用来实作类别参考之间的安全型别转换。而且，最好能够一开始就写好安全、明白的源码，试着避免类似的问题，但如果我们无法避免，IsPointerToObject 这个函式就有机会派上用场了。无论如何，这一节的用意应该是在对于这些系统数据结构的内容多做一些说明。

第三部: 进阶功能

我们已经深入研究了 Object Pascal 的基础功能，以及面向对象编程语言的范例，接下来我们可以来看看 Object Pascal 最新以及更进阶的功能了。泛型(Generics)、匿名方法(Anonymous methods)，以及镜像(reflection)都让我们使用最新的技术跟更明确的方法来延伸面向对象程序。

事实上，这些更进阶的编程语言功能，让开发人员在写程序上局限在特定的方法上，但透过更多型别与程序的抽象化、允许在源码里面使用更多动态的功能。

这一节的最后也会透过介绍核心的运行时间函式库的内容，来扩展这些编程语言的功能，这些都是 Object Pascal 开发模式的核心，也使得编程语言跟函式库本身的界线更为模糊。我们也会检视，例如我们在前面的篇幅里提到的 TObject，它是所有我们撰写的类别的最源头:它的影响至深至远，所有函式库的实作细节都与之相关。

第三部分章节涵盖了:

第十四章: 泛型
第十五章: 匿名方法
第十六章: 镜射与属性
第十七章: TObject 类别
第十八章: 执行时期函式库

14:泛型

Object Pascal 所提供的强型别检查对于提升源码的正确性是很有效的，这也是我在本书中不断提到的。强型别的检查有时候也很烦，就像有时候我们想要写一个程序，让这个程序可以处理不同数据型别或类别。这个议题已经被 Object Pascal 纳入成为功能之一，就像 C#, Java 这些相似的语言一样，我们称之为泛型(Generics)。

泛型的概念，或者用 C++的名词来说，叫做类别样板(template classes)。我在 1994 年所著，关于 C++的书里面有写道:

> *我们可以宣告一个类别，其中的一个或多个数据成员不特别指定使用哪种型别：这个型别的指定可以延迟到当我们需要定义该类别的变量时才来决定。我们在定义函式的时候也可以用相似的作法，不在撰写的函式的时候定义一个或多个参数的型别，直到该函式要被呼叫的时候才来指定。*

笔记 这本书的书名是”Borladn C++ 4.0 Object-Oriented Programming”，由 Marco Cantu 和 Steve Tendon 合着，在 1990 年代前期出版。

通用的键-值对（Key-Value Pairs)

本章的第一个范例就用一个很常见的类别，我实作了一个键-值对的数据结构。以下第一个程序片段，就是用传统的写法来建立这个数据结构，透过一个对象来储存数据的数值：

```
type
    TKeyValue = class
    private
        FKey: string;
        FValue: TObject;
        procedure SetKey(const Value: string);
        procedure SetValue(const Value: TObject);
    public
        property Key: string read FKey write SetKey;
        property Value: TObject read FValue write SetValue;
```

```
    end;
```

要使用这个类别，我们可以建立一件对象，设定它的键与值，然后使用它，就像下列的源码，节录自 KeyValueClassic 范例项目的主窗体中的几个方法：

```
// FormCreate
Kv := TKeyValue.Create;

// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender;

// Button2Click
Kv.Value := Self; // The form

// Button3Click
ShowMessage('[' + Kv.Key +',' + Kv.Value.ClassName + ']');
```

要是我们需要一个类似的类别，储存的是整数，而不是对象呢?好吧，我们也可以自己把整数作强制型别转换(这么做，出错的风险很高的)，或者建立一个新的类别，储存字符串键以及整数值。或许直接剪贴源码，然后修改里面的一部分也是种解决方法，最后我们就会有两段源码基本上完全一样，这样做，我们就抵触了良好程序设计的原则之一，而且我们以后如果发现源码里面有问题，就得同时改两次源码，如果复制三次，就要改三次，如果复制 20 次呢…… 那就烦死了。

泛型让我们可以对上例中的值的定义变得比较有弹性，而只要写一个泛型类别即可。一旦我们对键-值泛型类别进行了初始化，它就变成了特定的类别，直接跟特定的型别进行连结了。所以我们可以透过同样的作法，在我们的应用程序里面让这个类别宣告分别跟两种、三种，甚至 20 种型别进行连结，但类别的源码还是同一份，只是在变量或者属性宣告的时候，指定这个类别要跟哪一种数据型别进行链接。这样就不会在执行时期产生多余的动作了。让我们接着做，开始为我们成对的键-值，定义一个泛型类别：

```
type
    TKeyValue<T> = class
    private
        FKey: string;
        FValue: T;
        procedure SetKey(const Value: string);
        procedure SetValue(const Value: T);
```

```
  public
      property Key: string read FKey write SetKey;
      property Value: T read FValue write SetValue;
  end;
```

在上述的类别定义里，有一个尚未明确定义的型别，以字母 T 来代表。字母 T 在很多语言中都被使用来当做泛型类别中的符号，但我们可以在源码中用任何我们喜欢的字眼来代表泛型的类别或型别。使用 T 这个字母，是为了增加程序的可读性，泛型类别只是用这个符号来表示这个符号出现的地方，未来会以参数指定的型别来进行替换。如果类别中要进行泛型处理的地方有很多个，我们在实务上就会用它的角色来为这个代号进行命名，不用多个字母来代表(T, U, V)，这种方法是早期 C++里面的写法。

笔记

T 已经是通用的不成文写法，因为 C++语言在 1990 年代前期提出样板(template)这个概念的时候，就直接以 T 来表示样板。当时作者是要用 T 来表示样板(template)还是型别(Type)，目前已不可考。这样的转换在 Delphi 的世界里运作完美，型别只要以 T 字母开头即可，用 T 来表示 Type 是很有意义的作法。

泛型类别 TKeyValue<T>使用了尚未明确定义的型别作为它储存的两个数据字段之一、属性的值，以及设定方法(setter)的参数。方法的源码则很平常，但要注意到这当中也是使用了泛型型别，它们的定义包含了该类别的完整名称，包含泛型型别：

```
procedure TKeyValue<T>.SetKey(const Value: string);
begin
    FKey := Value;
end;
procedure TKeyValue<T>.SetValue(const Value: T);
begin
    FValue := Value;
end;
```

然而要使用这个类别时，我们就得完整的写出真正的型别了，必须提供实际的数据型别作为泛型型别。例如我们可以宣告一个键-值对象把按钮作为类别中储存的值，可以这样写:

```
var
    Kv: TKeyValue<TButton>;
```

在建立实体的时候，也需要写出完整的名字，因为这已经是确切的型别了(泛型型别中尚未实体化的型别名称看起来则像是一个型别建构机制)。

在键-值对当中使用尚未明确定义的型别来定义值的字段，会使得源码更能有弹性，我们现在只用了 TButton 来当做储存的值的字段，当然 TButton 的所有衍生类别对象也都可以存在这里面，我们也可以在解析出值字段的对象之后，使用该对象的各种方法。

以下的程序片段节录自 KeyValueGeneric 范例项目的主窗体：

```
// FormCreate
Kv := TKeyValue<TButton>.Create;


// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender as TButton;


// Button2Click
Kv.Value := Sender as TButton; // was "Self" but that is now invalid!
// Button3Click
ShowMessage ('[' + Kv.Key + ',' + Kv.Value.Name + ']');
```

在前一版的源码里面，要指派泛型对象的时候，我们可以指派一个按钮或者窗体，但现在只能储存按钮了，这也适用了编译器的强型别规则。但在泛型类别里面，输出字符串 kv.value.ClassName 则可以适用于所有型别，因为组件的 Name 或者 TButton 类别的任何方法或属性我们都可以在源码中使用。

当然我们也可以稍微改动一下原始程序中宣告键-值对的定义：

```
var
    Kvo: TKeyValue<TObject>
```

这样修改后，我们就可以把任何对象当成值的字段加以储存了。然而，我们就不能对解析出来的对象作太多动作，除非我们先把该对象转型成特定的型别。为了找到一个比较好的平衡点，我们可能会想要在特定的按钮跟任何对象之间找个介于中间的型别，例如说让储存的值只允许视觉组件:

```
var
    Kvc: TKeyValue<TComponent>;
```

对应的源码在 KeyValueGeneric 范例项目里面也有。最后，我们也可以建立一个储存一般数据的键-值对类别的实体，不让它储存任何对象，而是储存整数值:

```
var
    Kvi: TKeyValue<Integer>;
begin
    Kvi := TKeyValue<Integer>.Create;
    try
      Kvi.Key := 'Object';
      Kvi.Value := 100;
      Kvi.Value := Left;
      ShowMessage ('[' + Kvi.Key + ',' +
            IntToStr (Kvi.Value) + ']');
    finally
            Kvi.Free;
    end;
```

行内变数与泛型型别推定

当我们宣告了一个泛型型别的变量，这个宣告可能写很长。当我们用这个型别建立一个对象时，我们也必须重复相同的宣告，除非我们运用行内变量宣告与这功能的便利，让该变量的型别可以直接被推定，上一个范例源码的宣告就可以改写为:

```
begin
    var Kvi := TKeyValue<Integer>.Create;
    try
      …
```

在这段源码里面，我们就不用重复写两次完整的泛型类型声明，对我们需要使用容器类别的时候方便许多，我们稍后再介绍。

泛型的型别规则

当我们宣告了一个泛型型别的实体，这个型别就固定下来了，同时在后续的所有动作中，就会被编译器视为确定型别，所以如果我们有个泛型类别是像这样的:

```
type
```

```
TSimpleGeneric<T> = class
    Value: T;
end;
```

当我们为这个类型声明特定的对象时，我们在 Value 字段就不能指派跟宣告时不同型别的数据了。以下面两个对象为例，以下的部份指派就是不合法的(以下节录自 TypeCompRules 范例项目):

```
var
  Sg1: TSimpleGeneric<string>;
  Sg2: TSimpleGeneric<Integer>;
begin
  Sg1 := TSimpleGeneric<string>.Create;
  Sg2 := TSimpleGeneric<Integer>.Create;

  Sg1.Value := 'Foo';
  Sg1.Value := 10; // Error
  // E2010 Incompatible types: 'string' and 'Integer'

  Sg2.Value := 'foo'; // Error
  // E2010 Incompatible types: 'Integer' and 'string'
  Sg2.Value := 10;
```

一旦我们在泛型的宣告中确定了特定的型别，编译器就已经确定了要检查的型别，我们的源码就必须要遵循 Object Pascal 的强型别规则了。型别检查也会把泛型对象当成一个特殊的型别。当我们把泛型类别指定了特定的型别，我们就不能把指定另一个型别的泛型类别对象指派到不同泛型类别的变量了。只看文字说明很容易昏头，还是看一下源码，比较容易厘清内容:

```
Sg1 := TSimpleGeneric<Integer>.Create; // Error
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

我们接下来会在”泛型型别兼容性规则”这一节里面用明确的案例，指出型别兼容性规则是以结构检查，而不是以型别名称检查。我们不能在泛型型别已经确认了特定型别之后，又指派一个不同的、不相容的型别给一个泛型型别。

Object Pascal 里面的泛型

在前一个范例里，我们已经看过了如何在 Object Pascal 里面定义、使用泛型型别。我决定在深入到技术概念前，先用范例来介绍这个功能。因为泛型的技术概念相当复杂，也相当重要。从编程语言的观点看完泛型之后，我们再回头多看几个范例，包含了定义、使用泛型容器类别，这是 Object Pascal 里面主要使用这个技术的地方之一。

我们已经介绍过，当我们定义一个类别的时候，可以加入额外的参数，使用尖括号把特定的型别括起来即可:

```
type
    TMyClass<T> = class
        ...
    end;
```

泛型型别可以用来当做数据字段的型别(就像我们在前一个例子里面介绍的)、当做属性的型别、或者作为参数的型别，甚至是一个函式的回传型别，还可以用在更多地方。请注意，对一个区域字段(或者数组)使用这个型别并不是一种义务，有些情境下，泛型型别只用在作为回传值、参数，不能用在类别的宣告上，但可以用来定义类别的某些方法。

这种形式的延伸或泛型型别的宣告，不只是能对类别使用，也能用在记录上面(我们在第五章介绍过，记录也可以有方法、属性、覆写的运算方法)。泛型类别也可以有多个参数化的型别，我们用下面的源码作范例，让大家可以了解怎么在泛型的宣告中用上两个型别当成参数，一个当成方法的参数，另一个则是回传值的型别:

```
type
    TPWGeneric<TInput,TReturn> = class
    public
        function AnyFunction (Value: TInput): TReturn;
    end;
```

Object Pascal 里面对泛型的实作，跟其他静态编程语言一样，并不是基于执行时期的支持。这个功能是由编译器跟链接程序来处理的，在运行时间的机制几乎完全没有作什么事。跟虚拟函式的呼叫不同，虚拟函式是在执行时期进行绑定，而泛型类别的方法则是在我们定义出完整的泛型型别实体时就被建立了，而且是在编译时建立的！我们将会看到这个功能可能的缺

陷，但正面来看这个问题，它让我们知道泛型类别比一般类别的效率来的好，甚至连执行时期的检查所需的动作都省下了。在我们深入看到这些问题前，我们先来看些很明确的规则，这些规则打破了传统 Pascal 语言的型别兼容规则。

泛型型别兼容性规则

在传统的 Pascal 语言跟 Object Pascal 的核心型别兼容性规则都是基于型别名称是否一致。换句话说，两个变量的型别如果兼容，这两个型别的名称一定是一样的，不管两个型别里面实际上是怎么储存数据或定义方法的。

以下是一个静态数组间型别不兼容的传统例子(节录自 TypeCompRules 范例项目):

```
type
    TArrayOf10 = array [1..10] of Integer;


procedure TForm30.Button1Click(Sender: TObject);
var
    Array1: TArrayOf10;
    Array2: TArrayOf10
    Array3, array4: array [1..10] of Integer;
begin
    Array1 := Array2;
    Array2 := Array3; // Error
    // E2010 Incompatible types: 'TArrayOf10' and 'Array'


    Array3 := Array4;
    Array4 := Array1; // Error
    // E2010 Incompatible types: 'Array' and 'TArrayOf10'
end;
```

我们可以从上面的源码看到，四个数组在结构上是完全相同的。然而编译器只会让我们对型别兼容的数组进行指派。兼容的规则是必须型别名称完全相同(两个变量的型别都是 TArrayof10)，或者根本就是同一个型别(array3, array4 直接是宣告在同一行变量的宣告上)。

这个型别兼容性规则是很有局限性的例外，就像衍生类别的相关性。这个规则的另一个例外，而且是很明显的例外，就是泛型型别的型别兼容性，

这个情境也只会发生在编译器内部使用上，编译器会自行决定是否要位这个泛型型别建立一个新的内部型别，套用在该型别所有出现的地方。

新规则会界定泛型型别如果使用的是同样的泛别类别，且在定义时使用同样的实体型别，就视为相同的型别，不管在定义时，型别的名称是否相同。换句话说，泛型型别实体的全名会是泛型型别跟实体型别的组合。

在以下的范例中，四个变量就是完全型别兼容的：

```
type
    TGenericArray<T> = class
        AnArray: array [1..10] of T;
    end;

    TIntGenericArray = TGenericArray<Integer>;

procedure TForm30.Button2Click(Sender: TObject);
var
    Array1: TIntGenericArray;
    Array2: TIntGenericArray;
    Array3, array4: TGenericArray<Integer>;
begin
    Array1 := TIntGenericArray.Create;
    Array2 := Array1;
    Array3 := Array2;
    Array4 := Array3;
    Array1 := Array4;
end;
```

标准类别的泛型方法

使用泛型型别来定义类别是最常见的情境，泛型型别也可以用在非泛型的类别中。换句话说，一般的类别也可以有泛型方法。在这个情形下，我们不用在泛型符号出现的位置指定特定的型别，等到呼叫时再处理即可。以下就是一个简单的泛型方法的例子，节录自 GenericMethod 范例项目：

```
type
    TGenericFunction = class
    public
      function WithParam <T> (t1: T): string;
```

```
end;
```

笔记

当我第一次撰写上面这个源码的时候，或许是怀念以前 C++的日子吧，我把参数写成了(t: T)。不用特别说，Object Pascal 的语法是不分大小写的，在所有不分字母大小写的编程语言里面，这样的写法都是不合法的。编译器会指出上面两个 t 是无法辨识的……

我们没办法在类似的类别方法内部中作太多处理(至少在我们使用特定限制之前，我们会在下一章里面介绍)，所以我用特殊的泛型函式(稍后会介绍)写了一些源码来把这个型别转换成字符串，我们来看一下:

```
function TGenericFunction.WithParam<T>(T1: T): string;
begin
    Result := GetTypeName (TypeInfo (T));
end;
```

我们可以看到这个方法没有使用任何实际的值当做参数，只要求一些型别信息。在还不知道 T1 的型别之前，让写这段程序的时候相对变得很复杂。

我们可以用以下各种参数来呼叫这个『全局泛型函式』，可以使用的版本非常多:

```
var
    GF: TGenericFunction;
begin
    GF := TGenericFunction.Create;
    try
        Show (GF.WithParam<string>('Foo'));
        Show (GF.WithParam<Integer> (122));
        Show (GF.WithParam('Hello'));
        Show (GF.WithParam (122));
        Show (GF.WithParam(Button1));
        Show (GF.WithParam<TObject>(Button1));
    finally
            GF.Free;
    end;
```

以上所有呼叫这个函式的写法都是正确的，参数化的型别在这些呼叫的源码里面是暗中传递的。注意到泛型型别是被明确指定的，但参数的型别并没有被指定，所以我们会看到以下的执行结果:

```
string
Integer
string
ShortInt
TButton
TObject
```

如果我们在呼叫这个方法的时候没有在尖括号中指定型别，实际的型别就会从该参数的型别来指定。如果我们用一个型别、一个参数来呼叫这个方法，参数的型别就必须跟泛型型别的宣告一致。所以以下三行就无法被编译了:

```
Show (Gf.WithParam<Integer>('Foo'));
Show (Gf.WithParam<string> (122));
Show (Gf.WithParam<TButton>(Self));
```

泛型型别实体化

> ***请注意，这一节的主题是比较进阶的，我们会介绍泛型的内部运作机制跟它的潜在优化方式。如果您对泛型已经有研究过，第二次读到这一节，也会对您很有帮助。***

因为一些优化的例外，每次我们在为泛型型别实体化的时候，不管是泛型方法或是泛型类别，编译器都会产生一个新的型别。这个新的型别在同一个泛型型别的不同实体(或者是同一个方法的不同版本)之间是不会共享源码的。

我们来看一个范例(节录自 GenericCodeGen 范例项目)。在这个程序中定义了一个泛型类别，定义如下:

```
type
  TSampleClass <T> = class
  private
    FData: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (Value: T);
  end;
```

这三个方法的实作源码如下(请注意到 One 方法是绝对跟泛型型别独立的):

```
procedure TSampleClass<T>.One;
begin
    Form30.Show ('OneT');
end;
function TSampleClass<T>.ReadT: T;
begin
    Result := FData;
end;
procedure TSampleClass<T>.SetT(Value: T);
begin
    FData := Value;
end;
```

这个主程序当中使用了泛型型别要把它的方法被实体化时(这个动作会由编译器来处理)内存内的地址厘清。以下是它的源码:

```
procedure TForm30.Button1Click(Sender: TObject);
var
    T1: TSampleClass<Integer>;
    T2: TSampleClass<string>;
begin
    T1 := TSampleClass<Integer>.Create;
    T1.SetT (10);
    T1.One;

    T2 := TSampleClass<string>.Create;
    T2.SetT ('Hello');
    T2.One;

    Show ('T1.SetT: ' +
        IntToHex (PInteger(@TSampleClass<Integer>.SetT)^, 8));
    Show ('T2.SetT: ' +
        IntToHex (PInteger(@TSampleClass<string>.SetT)^, 8));
    Show ('T1.One: ' +
        IntToHex (PInteger(@TSampleClass<Integer>.One)^, 8));
    Show ('T2.One: ' +
        IntToHex (PInteger(@TSampleClass<string>.One)^, 8));
end;
```

执行结果如下(实际的数值每次都不会一样):

```
T1.SetT: C3045089
T2.SetT: 51EC8B55
T1.One: 4657F0BA
T2.One: 46581CBA
```

如我所预期的，编译器不只在内存里面为每一种数据型别的 SetT 方法都建立了不同的版本，就连 One 方法也是如此，尽管事实上各版本的 One 方法都是一模一样的。

不仅如此，如果我们重复宣告了一个完全相同的泛型型别，我们也会得到另一组新的实作函式。同样的，一个泛型型别的相同实作型别，在不同的单元文件里面被使用时，也会强迫编译器一再产生相同的源码，而可能会导致建立出的执行档明显变得很大。为了这个原因，如果我们要建立一个内含很多个方法的泛型类别，而这些方法跟泛型的型别没有什么关系的话，我会建议把这些方法定义在一个非泛型的类别里面，然后以这个非泛型的类别衍生一个需要使用到泛型型别的泛型类别：透过这个方法来实作，基础类别的方法就只会被编译器产出一次，执行档也就不会太大了。

笔记 在目前的编译器、链接程序、以及低阶 RTL 里面有些功能，用来缩减因为使用泛型而导致档案变大的状况，这些状况刚刚我们有提到过了。目前这些功能可以从一些网络上的文章阅读中稍微了解一下:http://delphisorcery.blogspot.it/2014/10/new-language-feature-in-xe7.html

---

泛型型别函式

目前我们在看泛型型别定义的最大问题，就是我们能对泛型类别型别的元素所做的非常少。我们可以用两个科技来降低这个限制。第一项就是使用在 RTL 里面特别用来支持泛型的特殊函式。第二项(也更有效)则是在泛型类别可以使用的类别上面作限制。

这一节我们将聚焦在第一项上面，下一节再来介绍使用上的限制。我们刚提到过，在 RTL 里面有一些函式是可以用来对泛型型别定义中的参数化型别进行处理的:

- ✧ Default(T)是一个和泛型功能一起出现的新函式，如果我们传入了任何一个非泛型的型别，它会回传空值，或 0，或 null，回传值可以是 0、

空字符串、nil 等等。以 0 清除过的内存对同一种型别的全局变量也会有一样的值(跟局部变量不同，事实上，全局变量会被编译器以 0 作初始化)

- ✧ TypeInfo(T)回传目前这个泛型型态在运行时间的指针，我们会在第十六章里面介绍更多关于型别信息的相关数据。
- ✧ SizeOf(T)会以 Byte 数回传该型别里面所使用的内存大小(如果 T 是对象或是字符串的话，这个回传值就是每个内存地址的大小，在 32 位系统中会回传 4 bytes, 64 位则回传 8 bytes)。
- ✧ IsManagedType(T)会告诉我们该型别在内存中是否是受管理的，如果是字符串或者动态数组的话，回传值就会是 true.
- ✧ HasWeakRef(T)，这个函式是跟支持 ARC 的编译器相关的，会回传一个内存参考是否为弱参考，需要特定的内存管理支持。
- ✧ GetTypeKind(T)：是让我们能够从型别信息中取得型别种类的快捷方式，这个回传值比 TypeInfo 所回传的型别定义来的更高阶。

笔记

以上这些方法所回传的，都是编译器处理过的常数，而不是在运行时间中实际呼叫什么函式所响应的值。重要的不是这些处理很快，而是让编译器跟链接程序可以对产出的源码进行优化，把没有使用到的源码移除。如果我们打算用这些函式的回传值来进行判断，编译器会发现判断式(if..else)其中的一段源码会被执行，而会自动移除掉没有被使用到的那一段。当相同的泛型方法被指定不同的型别，在编译时就会使用不同的判断状况，但要记得，源码都会先被产生，在优化的时候才有这些处理。

在 GenericTypeFunc 范例项目中有一个泛型类别，会显示三个泛型型别函式:

```
type
    TSampleClass <T> = class
    private
        FData: T;
    public
        procedure Zero;
        function GetDataSize: Integer;
        function GetDataName: string;
    end;


function TSampleClass<T>.GetDataSize: Integer;
begin
    Result := SizeOf (T);
end;
function TSampleClass<T>.GetDataName: string;
```

```
begin
    Result := GetTypeName (TypeInfo (T));
end;
procedure TSampleClass<T>.Zero;
begin
    FData := Default (T);
end;
```

在 GetDataName 方法中，我使用了 GetTypeName 函式(宣告在 TypeInfo 单元文件里面)，而不是直接存取数据结构，因为这样会从储存型别名称的已编码字符串进行适当的转换

因为有上述的宣告，我们可以把下列的测试源码进行编译，它会重复三次，使用三个不同的泛型型别实体。我就省略了重复的源码，只显示用来存取 data 字段的程序，它们会依照实际型别来进行内容变更：

```
var
    T1: TSampleClass<Integer>;
    T2: TSampleClass<string>;
    T3: TSampleClass<double>;
begin
    T1 := TSampleClass<Integer>.Create;
    T1.Zero;  Show ('TSampleClass<Integer>');
    Show ('Data: ' + IntToStr (T1.FData));
    Show ('Type: ' + T1.GetDataName);
    Show ('Size: ' + IntToStr (T1.GetDataSize));

    T2 := TSampleClass<string>.Create;
    ...
    Show ('Data: ' + T2.FData);

    T3 := TSampleClass<double>.Create;
    ...
    Show ('Data: ' + FloatToStr (T3.FData));
```

执行上面这段源码(节录自 GenericTypeFunc 范例项目)，结果如下:

```
TSampleClass<Integer>
Data: 0
Type: Integer
```

```
Size: 4
TSampleClass<string>
Data:
Type: string
Size: 4
TSampleClass<Double>
Data: 0
Type: Double
Size: 8
```

请注意，我们也可以对指定的型别使用泛型型别函式，在泛型类别的内容之外，例如可以这样写:

```
var
    I: Integer;
    s: string;
begin
    I := Default (Integer);
    Show ('Default Integer': + IntToStr (I));

    s := Default (string);
    Show ('Default String': + s);
    Show ('TypeInfo String': + GetTypeName (TypeInfo (string));
```

结果很直觉，当然就是:

```
Default Integer: 0
Default String:
TypeInfo String: string
```

笔记 我们不能直接对变量进行 TypeInfo 呼叫，例如上面的 TypeInfo(s)，这个呼叫的参数只能是一个型别。

泛型类别的类别建构函式

有个很有趣的例子，就是为泛型类别定义一个类别建构函式。事实上，这样的建构函式是由编译器所建立，并由每一个泛型类别的实体所呼叫的，也就是说，每一个实际的型别被定义时，都使用了泛型样板。这一点就相当有趣了，因为这会让我们的程序在不使用类别建构函式来建立每个泛型类别实体的时候，初始化程序的源码变得相当复杂。

举个例子，想象一下内含一些类别数据的泛型类别。我们只会取得一个类别数据的实体，共享于每一个泛型类别实体上。如果我们需要为这个类别数据进行初始化，指派一个初始值给它，我们就不能透过单元文件的初始区来做这个事情，因为我们并不知道这个类别实际上会使用哪个型别来宣告泛型类别。

以下是一个最简单的范例，范例中，泛型类别里包含有类别建构函式，用来为 DataSize 这个类别数据字段进行初始化，节录自 GEnericClassCtor 范例项目:

```
type
    TGenericWithClassCtor <T> = class
    private
        FData: T;
        procedure SetData(const Value: T);
    public
        class constructor Create;
        property Data: T read FData write SetData;
        class var
          DataSize: Integer;
    end;
```

接下来是这个泛型类别建构函式的源码，用了一个内部的字符串列表(请看完整的实作区源码)来追踪哪一个类别建构函式有被呼叫到:

```
class constructor TGenericWithClassCtor<T>.Create;
begin
    DataSize := SizeOf (T);
    ListSequence.Add(ClassName);
end;
```

这范例程序建立了，也使用了几个泛型类别的实体并宣告了第三个数据型别，这地三个资料型别则会被链接程序移除掉:

```
var
    GenInt: TGenericWithClassCtor <SmallInt>;
    GenStr: TGenericWithClassCtor <string>;
type
    TGenDouble = TGenericWithClassCtor <Double>;
```

如果我们要求程序显示 ListSequence 的内容，我们只会看到已经被初始化的型别名字:

```
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

然而，如果我们用相同的数据型别在不同的单元文件里面建立了泛型实体，链接程序就不会如预期中那样运作，我们会看到同一个型别有多个泛型类别建构函式。

笔记 类似的程序并不容易界定。为了避免重复进行初始化，我们可能会想要检查类别建构函式是否已经被执行过。通常这个问题对于泛型类别来说，是全面性的限制当中的一个，而链接程序并没有能力对这情形加以优化。

我在本范例的第二个单元文件当中加入了一个名为 Useless 的程序，当我们取消这些源码的批注状态时，编译器就会标明这些源码有问题，当初始的顺序如下:

```
TGenericWithClassCtor<System.string>
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

泛型守则 (Generic Constraints)

我们已经看过，在泛型类别中，我们能对泛型型别的值所做的处理很少。我们可以把它传过来传过去，或是对它进行我们已经介绍过的型别处理中的任何一项。

要能够对泛型类别的型态作更确切的动作，我们通常必须给它一些守则才行。例如如果我们规定泛型类别必须是一个类别，编译器就会让我们透过它来使用 TObject 的方法。我们也可以进一步规范该类别必须是类别架构中的特定部分，或者一定要实作特定的接口，这样我们才可能透过泛型类别的实体呼叫特定的方法。

类别守则 (Class Constraints)

最简单的守则就是我们要求该型别必须是一个类别，我们可以这么写，来达到这个守则：

```
type
```

```
TSampleClass <T: class> = class
```

透过特定的类别守则，我们可以指定只有哪些对象型别可以作为泛型型别。以下的源码就可以作为例子(节录自 ClassConstraint 范例项目):

```
type
    TSampleClass <T: class> = class
    private
        FData: T;
    public
        procedure One;
        function ReadT: T;
        procedure SetT (t: T);
    end;
```

我们可以用前两个指令成功建立出泛型实体，但第三个指令就不行了：

```
sample1: TSampleClass<TButton>;
sample2: TSampleClass<TStrings>;
sample3: TSampleClass<Integer>; // Error
```

编译器会指出第三个指令有错，错误讯息为:

```
E2511 Type parameter 'T' must be a class type
```

指定这个限制的好处是什么?在泛型类别方法中，我们现在已经可以使用 TObject 的任何一个方法了，包含虚拟方法喔！以下就是 TSampleClass 泛型类别的方法 One 的源码:

```
procedure TSampleClass<T>.One;
begin
    if Assigned (FData) then
    begin
        Form30.Show ('ClassName: ' + FData.ClassName);
        Form30.Show ('Size: ' + IntToStr (FData.InstanceSize));
        Form30.Show ('ToString: ' + FData.ToString);
    end;
end;
```

笔记 这里我们有两个建议。第一个是 InstanceSize 会回传该对象的实际大小，跟我们刚用过的 SizeOf 泛型函式不同，SizeOf 会回传参考型别的大小。其次，要留意到我们如何使用 TObject 类别的 ToString 方法。

我们可以执行个几次，玩玩看这个程序，看一下实际的效果，在源码中我们定义了几个不同的泛型型别实体，就像这段程序片段:

```
var
    Sample1: TSampleClass<TButton>;
begin
    Sample1 := TSampleClass<TButton>.Create;
    try
        Sample1.SetT (Sender as TButton);
        Sample1.One;
    finally
         Sample1.Free;
    end;
```

请注意到透过宣告一个具有客制化过的 ToString 方法的类别，这个版本会在数据对象的型别确定的时候被呼叫，不管实际上是什么型别被指派给泛型型别。换句话说，如果我们建立了一个以 TButton 为类别的泛型类别实体：

```
type
    TMyButton = class (TButton)
    public
      function ToString: string; override;
end;
```

我们可以把这个对象当成 TSampleClass<TButton>型别的对象，或者作为一个特定泛型型别的实体。在两种情形下，One 方法都会呼叫到 ToString 这个特别版本的方法:

```
var
    Sample1: TSampleClass<TButton>;
    Sample2: TSampleClass<TMyButton>;
    Mb: TMyButton;
begin
    ...
    Sample1.SetT (Mb);
    Sample1.One;
    Sample2.SetT (Mb);
    Sample2.One;
```

跟类别守则一样，我们也可以建立记录守则，例如这样宣告:

```
type
  TSampleRec <T: record> = class
```

然而，这跟大多数的记录并没有太多的不同(并没有共同的基础类别)，所以这个宣告相对的没有太多的新功能。

特定的类别守则

如果我们的泛型类别需要跟特定的类别子集一起运作(特定的类别继承结构树)，我们可能会想要以特定的基础类别进行泛型类别守则，例如我们可能会想要这样宣告:

```
type
  TCompClass <T: TComponent> = class
```

这个泛型类别的实体只容许组件类别，也就是 TComponent 的任一个衍生类别。这让我们的泛型类别变得很确切(对，听起来很奇怪，但它的确很奇怪)，而且编译器会让我们在这个泛型类别的实体当中使用 TComponent 的所有方法。

如果这看起来很棒，再多想一下。如果我们考虑到要达成继承的目标与型别兼容性规则，我们可能会指出使用传统面向对象技术的相同问题，而不会使用泛型类别了。我不是说特定的类别守则完全没用，但它显然不如高阶的类别守则或(我发现这个还蛮有趣的)以接口为基础的守则那么强大。

接口守则

不限制泛型类别只能使用特定的类别，改以限制只能使用有实作特定接口的类别，可以透过参数型别的开放性来达成泛型类别的通用性。这使得我们可以在泛型赢别的实体当中呼叫特定接口的方法。在泛型上使用接口守则在 C#编程语言是很常见的。我们用一个例子作为开场白(节录自 IntfConstraint 范例项目)。首先，我们得先宣告一个接口：

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    property Value: Integer read GetValue write SetValue;
  end;
```

接着我们可以定义一个实作它的类别:

```
type
    TGetValue = class (TSingletonImplementation, IGetValue)
    private
      fValue: Integer;
    public
      constructor Create (Value: Integer = 0);
      function GetValue: Integer;
      procedure SetValue (Value: Integer);
    end;
```

在定义限制使用有实作特定接口的泛型类别时，就开始有趣了：

```
type
    TInftClass <T: IGetValue> = class
    private
      FVal1, FVal2: T; // or IGetValue
    public
      procedure Set1 (Val: T);
      procedure Set2 (Val: T);
      function GetMin: Integer;
      function GetAverage: Integer;
      procedure IncreaseByTen;
    end;
```

请留意实作这个类别的泛型方法的源码：

```
function TInftClass<T>.GetMin: Integer;
begin
    Result := Min (FVal1.GetValue, FVal2.GetValue);
end;


procedure TInftClass<T>.IncreaseByTen;
begin
    FVal1.SetValue (FVal1.GetValue + 10);
    FVal2.Value := FVal2.Value + 10;
end;
```

有了这些定义，我们就可以这样来用这个泛型类别了:

```
procedure TFormIntfConstraint.btnValueClick( Sender: TObject);
```

```
var
    IClass: TInftClass<TGetValue>;
begin
    IClass := TInftClass<TGetValue>.Create;
    try
        IClass.Set1 (TGetValue.Create (5));
        IClass.Set2 (TGetValue.Create (25));
        Show ('Average: ' + IntToStr (IClass.GetAverage));
        IClass.IncreaseByTen;
        Show ('Min: ' + IntToStr (IClass.GetMin));
    finally
      IClass.val1.Free;
      IClass.val2.Free;
      IClass.Free;
    end;
end;
```

为了表现这个泛型类别的弹性，我为这个接口建立了另一个完全不同的实作方法:

```
TButtonValue = class (TButton, IGetValue)
public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
        Parent: TWinControl): TButtonValue;
end;
function TButtonValue.GetValue: Integer;
begin
    Result := Left;   // use base class property
end;
procedure TButtonValue.SetValue(Value: Integer);
begin
    Left := Value;    // use base class property
end;
```

我们在以下的源码里面，用类别函式(没有在本书中列出)为 Parent 控件建立了一个组件，其位置随机指定：

```
procedure TFormIntfConstraint.btnValueButtonClick( Sender: TObject);
```

```
var
    IClass: TInftClass<TButtonValue>;
begin
    IClass := TInftClass<TButtonValue>.Create;
    try
        IClass.Set1 (TButtonValue.MakeTButtonValue ( self, ScrollBox1));
        IClass.Set2 (TButtonValue.MakeTButtonValue ( self, ScrollBox1));
        Show ('Average: ' + IntToStr (IClass.GetAverage));
        Show ('Min: ' + IntToStr (IClass.GetMin));
        IClass.IncreaseByTen;
        Show ('New Average: ' + IntToStr (IClass.GetAverage));
    finally
      IClass.Free;
    end;
end;
```

接口参考 v.s.泛型接口守则

在刚刚最后一个范例中，我定义了一个泛型类别，可以接受有实作特定接口的任何类别作为其实作类别。我们也可以透过建立一个基于接口参考的标准类别(非泛型类别)来达到相似的效果。事实上，我们可以定义一个类别，像这样(节录自 IntfConstraint 范例项目):

```
type
    TPlainInftClass = class
    private
        FVal1, FVal2: IGetValue;
    public
        procedure Set1 (Val: IGetValue);
        procedure Set2 (Val: IGetValue);
        function GetMin: Integer;
        function GetAverage: Integer;
        procedure IncreaseByTen;
    end;
```

这两个作法之间的差别是什么?首先，在上述的类别里面我们可以传递两个不同型别的对象给设定方法(setter)，提供的类别都实作了特定的接口。但在泛型版本里面，我们只能够传递一种特定型别的对象(要看该泛型类别实

体要求哪一种型别)。所以泛型版本的程序在型别检查这一点，比较保守，限制也比较多。

从我的观点来看，主要的不同是使用接口为基础的版本表示当中有 Object Pascal 的参考计算器制在运作着，而使用泛型版本的时候，类别会直接以特定型别的一般对象家里处理，因此参考计算器制并没有介入。再者，泛型版本可能有多重的限制，像是建构函式的守则，以及使用不同的泛型函式(像是位泛型型别要求特定的型别)，有些动作我们就不能做(事实上，当我们透过接口来处理，我们就不能使用基础类别 TObject 的方法)。

换句话说，使用具有接口守则的泛型类别，可以让这个类别具有接口的好处，却没有接口的麻烦。但值得一提的是，这两种技术在大多数的情形来看都是完全一样的，且在其他情形下，这种作法则有更多的弹性。

预设建构函式守则

还有另一种可能的泛型型别守则，称为预设建构函式或者无需参数的建构函式。如果我们需要呼叫预设的建构函式来建立泛型型别的对象(例如要填满一个列表)，我们就可以使用这个守则。理论上(出处为官方说明文件)，编译器应该只让我们在具备预设建构函式的型别上使用它，如果没有预设的建构函式的话，编译器会跳过它，直接呼叫 TObject 的预设建构函式。

一个具备建构函式守则的泛型类别可以用以下的方式来写(节录自 IntfConstraint 范例项目)：

```
type
  TConstrClass <T: class, constructor> = class
  private
    FVal: T;
  public
    constructor Create;
    function Get: T;
  end;
```

笔记

我们也可以指定不具备类别守则的建构函式守则，在建构函式处理的型别会是一个类别。把两者都列出来，会让程序更具备可读性。

假设用以下的宣告，我们可以使用建构函式来建立一个泛型内部对象，而不用知道该类别实际上的运作方法：

```
constructor TConstrClass<T>.Create;
begin
    FVal := T.Create;
end;
```

我们要怎么使用这个泛型类别，且实际的规则是什么?在下一个范例里面，我们提供了两个类别。第一个是有预设建构函式的(无需参数)，第二个则是要求一个参数的建构函式：

```
type
     TSimpleConst = class
     public
         FValue: Integer;
         constructor Create; // Set Value to 10
     end;
     TParamConst = class
     public
        FValue: Integer;
        constructor Create (I: Integer); // Set Value to I
     end;
```

就像前面提过的，理论上我们应该只能使用第一个类别，但实务上我们却是两个都能使用:

```
var
    ConstructObj: TConstrClass<TSimpleCost>;
    ParamCostObj: TConstrClass<TParamCost>;
begin
    ConstructObj := TConstrClass<TSimpleCost>.Create;
    Show ('Value 1: ' + IntToStr (ConstructObj.Get.Value));

    ParamCostObj := TConstrClass<TParamCost>.Create;
    Show ('Value 2: ' + IntToStr (ParamCostObj.Get.Value));
```

执行结果是:

```
Value 1: 10
Value 2: 0
```

事实上，第二个对象从没有被初始化过，假如我们试着对这个范例项目侦错，并深入到源码，我们就会发现一个呼叫 TObject.Create 的源码(这里我认为是有错的)。注意，假如我们试着直接呼叫:

```
with TParamConst.Create do
```

编译器会指出以下的错误:

```
[DCC Error] E2035 Not enough actual parameters
```

笔记

即使直接呼叫 TParamConst.Create 在编译时期会犯错(如上例所示)。类似的呼叫如果使用类别参考或者其他间接方式呼叫就会成功，这或许也说明了建构函式守则效果的规则。

泛型守则的总整理以及组合应用

我们可以在泛型型别上应用许多中不同的守则，在此我们做个简单的整理，直接用源码来进行吧:

```
type
  TSampleClass <T: class> = class
  TSampleRec <T: record> = class
  TCompClass <T: TButton> = class
  TInftClass <T: IGetValue> = class
  TConstrClass <T: constructor> = class
```

看过这些守则(我也花了不少时间来熟悉它们)之后，没办法很直觉的知道我们可以把这些守则进行组合。例如我们可以定义一个泛型类别，让它局限在特定类别的子类别架构下并且要求它具备特定的接口，像是:

```
type
  TInftComp <T: TComponent, IGetValue> = class
    ...
  end;
```

并不是所有的组合都有意义：例如我们不可以同时要求使用类别跟记录，在使用类别守则时，合并使用特定的类别守则是多余的。最后，请注意方法守则则是一个特例，它可以达成单一方法接口守则的要求(然而比实际上体验的更复杂)。

预先定义的泛型容器

因为在 C++语言发展样板的早期，最常被使用的样板类别已经被定义在样板容器或者整个清单里面了，对这些已定义的类别，C++语言则是把他们定义为一个标准样板函式库(Standard Template Library, 简称 STL)。

当我们在定义一个对象的列表时，就像在 Object Pascal 里面提供的 TObjectList，我们就拥有了一个可以储存任何型别对象的列表。使用继承或者组合的方式，我们都需要定义一个特殊型别的自定容器，但这会是个乏味(其中也潜藏危机)的功能。

Object Pascal 编译器原本就内建了一些泛型容器的类别，我们可以在 Generics.Collections 单元文件里面找到。当中的四个核心容器类别都是独立实作的(彼此之间没有继承关系)，这些类别都是以类似的现代方法实作的(使用动态数组)，而且也都对应到相应的非泛型容器类别中，原来的容器类别都放在 Contnrs 单元文件里面:

```
type
    TList<T> = class
    TQueue<T> = class
    TStack<T> = class
    TDictionary<TKey,TValue> = class
    TObjectList<T: class> = class(TList<T>)
    TObjectQueue<T: class> = class(TQueue<T>)
    TObjectStack<T: class> = class(TStack<T>)
    TObjectDictionary<TKey,TValue> = class(TDictionary<TKey,TValue>)
```

这些类别逻辑上的差异，从它们的命名上应该可以看出明显的不同。要熟悉跟测试它们最好的方法，是在原本使用非泛型容器类别的源码跟使用不同数据型别的泛型版本，比较看看有多少差别。

笔记 接下来要讨论的这个程序，ListDemoMd2005，只使用了一些方法，所以在泛型与非泛型的类别当中，并没有对接口兼容性做太多的测试，不过我决定用一个已存在的程序，不要重新写一个。另一个要使用这个范例的原因，则是我们可能在原有的源码里面进行泛型修改的时候，可以透过泛型这个功能而获得额外的改进。

使用 TList<T>

名为 ListDemoMd2005 的这个程序里面，有一个用来定义 TDate 类别的单元文件，而主窗体里面就使用了一个 TList 来储存日期数据。在程序一开始的 uses 区段，我加入了 Generics.Collection 的引用，然后把主窗体里面的宣告改成了这样:

```
private
    FListDate: TList <TDate>;
```

当然,用来建立这个列表的主窗体 OnCreate 事件处理程序也需要做些修改，得改成这样:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    FListDate := TList<TDate>.Create;
end;
```

现在，我们可以直接编译看看，先不管其他的源码。此时，程序中还包含一些『待协寻』的问题，试着把 TButton 加入到这个列表里面。对应的源码原本是可以正常编译的，但现在则会出现错误了:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
    // Add a button to the list
    FListDate.Add (Sender); // Error:
    // E2010 Incompatible types: 'TDate' and 'TObject'
end;
```

新的日期列表在型别检查上比原本通用的指标列表更能自动的对型别进行检查。移除了上面这一行有问题的源码以后，程序就可以正常编译、正常运作了。不过，它还有改进的空间。

以下的源代码，会把列表中所有的日期数据显示在 ListBox 组件里头：

```
var
    I: Integer;
begin
    ListBox1.Clear;
    for I := 0 to ListDate.Count - 1 do
        Listbox1.Items.Add (  (TObject(FListDate [I]) as TDate).Text);
```

请注意当中的型别转换，因为这个程序使用了指针型的列表(TList)，而不是组件列表(TObjectList)。

我们可以把这个程序先改进一下，改写成：

```
for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add (ListDate [I].Text);
```

这个程序片段的另一个改进空间，可以是使用列举(预先定义的泛型列表中完全支持此一功能)，而不要使用 for 循环：

```
var
    aDate: TDate;
begin
    for aDate in ListDate do
    begin
        Listbox1.Items.Add (ADate.Text);
    end;
```

最后，这个程序还可以再改进一个地方，就是透过使用 TObjectList 来储存 TDate 组件，但这是下一节的主题了。

就像前面提到的 TList<T>这个泛型类别有较高的兼容性。它包含了所有原有的方法，像是 Add, Insert, Remove, 以及 IndexOf。同时也提供了 Capacity 跟 Count 属性。只是 Items 变成了 Item，而且是默认属性(可以直接用变量名称加上方括号来存取，不用透过属性名称)，过去我们不常直接用这种方式存取。

对 TList<T>进行排序

了解这当中排序的作法也是挺有趣的(我的目的是为 ListDemoMd2005 范例项目加入排序的功能)。Sort 方法是这样定义的:

```
procedure Sort; overload;
procedure Sort(const AComparer: IComparer<T>); overload;
```

这里的 IComparer<T>接口，是定义在 Generics.Defaults 单元文件里面的。如果我们执行了这个程序的第一个版本，它会使用预设的比较函式，由 TList<T>的预设建构函式进行初始化。在我们的案例中，这是没有用处的。

反之，我们要做的是为 IComparer<T>定义一个适当的实作方法。为了让型别能兼容，我们得定义能够针对 TDate 类别进行排序的实作源码。有很多个方法可以完成这个目标，包含使用匿名方法(我们虽然在下一章才会介绍到匿名方法，不过下一节会先介绍作法)。这是个有趣的技术，也是因为它让我们有机会可以介绍许多种泛型的设计模式。这是透过位于 Generics.Defaults 单元文件的部分结构化类别，名为 TComparer 来达成的。

笔记 我之所以把这个类别称为结构化类别，是因为它协助了源码结构化，而没有在实作上加入一大堆多余的名词。类似的类别通常也被称为框架类别，通常也用在较为复杂的设计上。

这个类别是定义成抽象化类别，并以泛型实作当中的接口：

```
type
  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  public
    class function Default: IComparer<T>;
    class function Construct(
        const Comparison: TComparison<T>): IComparer<T>;
    function Compare(
        const Left, Right: T): Integer; virtual; abstract;
  end;
```

我们所要做的，是先以特定的数据型别(例如本例中的 TDate)为泛型类别实体化，也继承一个实作了为这个数据型别处理的 Compare 方法的类别。这两个动作可以一次完成，我们透过以下的程序来说明一下:

```
type
  TDateComparer = class (TComparer<TDate>)
    function Compare(  const Left, Right: TDate): Integer; override;
  end;
```

如果你觉得这段程序看起来不太适应，那很正常。新类别继承自泛型类别的特定实体，我们可以从以下两个独立的步骤来体验:

```
type
  TAnyDateComparer = TComparer<TDate>;
  TMyDateComparer = class (TAnyDateComparer)
    function Compare(  const Left, Right: TDate): Integer; override;
  end;
```

笔记 这两个独立的宣告应该可以精简需要建立的源码，我们只要在宣告 TAnyDateComparer 型别的单元文件里面宣告它们即可。

我们可以在原始码里面找到 Compare 函式的实际源码，不过因为它不是这一节的关键，所以我就不在这里特别贴出来了。但是要记得，即使我们把列表排序过了，用 IndexOf 方法也不会比较有效率(这一点跟 TStringList 类别完全不同喔)

以匿名方法进行排序

在前一节出现的排序源码看起来相当复杂，实际上的确也是如此。把排序函式直接传递给 Sort 方法相对的简单，也清楚多了。在还没有支持泛型功能之前，这一直都是通用排序功能的作法。在 Object Pascal 里面，现在我们也可以透过匿名方法来达成这个功能了(匿名方法是一种方法指标，里面包含几种额外的功能，我们在下一章里面加以介绍)。

笔记 我建议大家看一下这一节，就算您对匿名方法所知不多，也可以先读一次，等读完下一章之后再回头重看一次。

TList<T>类别的 Sort 方法的参数 IComparer<T>，事实上可以被用来呼叫 TComparer<T>的 Construct 方法，会把一个匿名方法当做参数传递，可以写成:

```
type
  TComparison<T> = reference to function(
      const Left, Right: T): Integer;
```

实务上，我们可以写一个型别兼容的函式，然后把它当成参数传递:

```
function DoCompare (const Left, Right: TDate): Integer;
var
  Ldate, RDate: TDateTime;
begin
   LDate := EncodeDate(Left.Year, Left.Month, Left.Day);
   RDate := EncodeDate(Right.Year, Right.Month, Right.Day);
   if LDate = RDate then
      Result := 0
   else if LDate < RDate then
       Result := -1
```

```
    else
        Result := 1;
end;

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
    ListDate.Sort (TComparer<TDate>.Construct (DoCompare));
end;
```

笔记

上例中的 DoCompare 方法是以类似匿名方法的原理运作的，即使没有函式名称也能运作。我们稍后会以另一个程序片段来说明不需要函式名称的这个事实。我们到下一章会介绍 Object Pascal 这个新功能的更多信息。也请留意到，我们为 TDate 记录也定义了两个比较用的运算符号，可以让源码稍微简化，但即使有类别的存在，比对的源码也必须放在该类别的方法里面才行。

这看起来相当传统，我们在使用这种作法时可以省却独立函式的宣告，直接把源码当做参数传给 Construct 方法，如下:

```
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
    ListDate.Sort (TComparer<TDate>.Construct (
        function (const Left, Right: TDate): Integer
        var
            LDate, RDate: TDateTime;
        begin
            LDate := EncodeDate(Left.Year,
                Left.Month, Left.Day);
            RDate := EncodeDate(Right.Year,
                Right.Month, Right.Day);
            if LDate = RDate then
              Result := 0
            else if LDate < RDate then
              Result := -1
            else
              Result := 1;
        end));
end;
```

这个例子应该有激发你对于匿名方法学习的欲望吧!肯定的是，最后这个例子写起来比前一节里面的源码简单的多，虽然对许多 Object Pascal 开发人员来说建立一个衍生类别看起来已经很简洁易懂了(继承的版本会把逻辑区隔的很清楚，让未来可能重复使用源码时相对简单的多，但许多时候，我们仍然还是不会去使用这个功能)。

对象容器(Object Containers)

除了在这一节一开始介绍的泛型类别，还有四个从定义在 Generics.Collections 单元中的基础类别继承而来的泛型类别，可以模仿现存在 Contnrs(这是以容器类别的基础)单元里面的类别功能：

```
type
  TObjectList<T: class> = class(TList<T>)
  TObjectQueue<T: class> = class(TQueue<T>)
  TObjectStack<T: class> = class(TStack<T>)
```

和它们的基础类别相较之下，有两个关键性的差异。第一个是这些泛型类别只能用在对象上，第二则是他定义了一个客制化的 Notification 方法，万一对象要从列表当中(除了选择性的呼叫 OnNotify 事件处理常数)被移除了，就会释放掉该对象。

换句话说，TObjectList<T>类别的规则跟它的非泛型类别 OwnsObjects 属性被设定成 true 的时候规则是一样的。如果您觉得对于为什么这个属性不再是可以被设定的感到好奇，可以回想一下 TList<T>也是一样，可以直接使用对象型别，跟它的非泛型类别规则有些不同了。

还有第四个类别 TObjectDictionary<TKey, TValue>，这个类别则是以不同的方法定义的，它可以拥有 Key 对象，Value 对象，或者两者同时拥有。我们可以在类别建构方法的 TDictionaryOwnerships 属性中看到所有可能性。

使用泛型字典(Dictionary)

在所有预先定义的泛型容器类别中，Dictionary 或许是最值得我们花时间好好了解的了：TObjectDictionary<K, V>。

笔记 Dictionary 在这个案例中，意指一种元素的集合，在这个集合当中，可以透过唯一键作为查询与写入的依据，Dictionary 从其作用上，也可以把它想成是关系型数组。在古典的 Dictionary 当中，我们只能用字符串当成查询的关

键，但在程序实作的时候，键(Key)未必要是字符串(虽然字符串是比较常用的对象)。在旧版 Delphi 当中的通用集合中，Dictionary 的提示里使用了 TKey 跟 TValue 的词语。然而 TValue 是不具关联的运行时间数据型别(我们会在第 16 章里面介绍)，在 Delphi 11 里面这两个词语则被改为 K 跟 V，以避免语意混淆。本版当中也已经做了相应的更新，如大家在前面的范例中可以发现的。

其他的类别也很重要，但是它们似乎相对容易使用，也容易理解。为了示范 Dictionary 的使用，我写了一个范例，可以从数据表里面取出数据，为每个记录建立一个对象，然后使用综合索引值作为键值。这个作法的原因是相似的结构可以很容易就用来建立代理关系(proxy, 延迟初始化)，而这种关系里，键值可以当成从数据库加载数据时的轻量化数据版本。

以下是两个在 CustomerDictionary 范例项目中使用到的类别，用以透过键与值的对应储存数据。第一个只有两个对应数据来呼应数据库的数据表，第二个则有比较复杂的数据结构(我已经省略了私有数据字段，取得数据的方法，以及设定数据的方法)：

```
type
    TCustomerKey = class
    private
        ...
    published
        property CustNo: Double  read FCustNo write SetCustNo;
        property Company: string  read FCompany write SetCompany;
    end;


    TCustomer = class
    private
         ..
         procedure Init;
         procedure EnforceInit;
    public
        constructor Create (aCustKey: TCustomerKey);
        property CustKey: TCustomerKey
            read FCustKey write SetCustKey; published
        property CustNo: Double  read GetCustNo write SetCustNo;
        property Company: string  read GetCompany write SetCompany;
        property Addr1: string  read GetAddr1 write SetAddr1;
```

```
    property City: string  read GetCity write SetCity;
    property State: string  read GetState write SetState;
    property Zip: string  read GetZip write SetZip;
    property Country: string  read GetCountry write SetCountry;
    property Phone: string  read GetPhone write SetPhone;
    property FAX: string  read GetFAX write SetFAX;
    property Contact: string  read GetContact write SetContact;
  class var
    RefDataSet: TDataSet;
  end;
```

第一个类别非常单纯(每一个对象在被建立的时候就已经完成了初始化)，TCustomer类别使用了延迟初始化(或者代理)模式且把一个参考记录在原始码数据库中，以对象变量的(class var)形式分享给所有对象。

笔记

在这个范例中，我使用了一个衍生自 TDataSet 的对象来存取数据库的数据，好让它跟现实世界的情境比较贴近点。讨论在 Delphi 中存取数据库的确超越了本书的范围，所以我并不会提供额外的篇幅来描述 TDataSet 类别实际上是如何作用的。

当对象被建立的时候，它的参考就已经被指派为对应的 TCustomerKey，而类别数据则参照到源数据集合(dataset)。在每一个数据取得的方法中，该类别会检查对应的对象是否的确已经完成了初始化，然后才进行回传：

```
function TCustomer.GetCompany: string;
begin
  EnforceInit;
  Result := FCompany;
end;
```

EnforceInit 方法会检查一个区域旗标，最后会呼叫 Init 从数据库把数据加载到数据库内的内存对象：

```
procedure TCustomer.EnforceInit;
begin
  if not FInitDone then
    Init;
end;

procedure TCustomer.Init;
```

```
begin
    RefDataSet.Locate('custno', CustKey.CustNo, []);
    // could also load each published field via RTTI
    FCustNo := RefDataSet.FieldByName ('CustNo').AsFloat;
    FCompany := RefDataSet.FieldByName ('Company').AsString;
    FCountry := RefDataSet.FieldByName ('Country').AsString;
    ...
    FInitDone := True;
end;
```

透过这两个类别，我在这个应用程序中加入了一个特殊用途的 dictionary。这个客制化的 dictionary 类别继承自特定型别实体化的泛型类别，并加入一个特定的方法:

```
type
    TCustomerDictionary = class (
        TObjectDictionary <TCustomerKey, TCustomer>)
    public
        procedure LoadFromDataSet (Dataset: TDataSet);
    end;
```

加载方法则把数据填入了 dictionary、把数据复制到内存中，但只有键值的对象:

```
procedure TCustomerDictionary.LoadFromDataSet( Dataset: TDataSet);
var
    CustKey: TCustomerKey;
begin
    TCustomer.RefDataSet := dataset;
    Dataset.First;
    while not Dataset.EOF do  begin
       CustKey := TCustomerKey.Create;
       CustKey.CustNo := Dataset ['CustNo'];
       CustKey.Company := Dataset ['Company'];
       self.Add(custKey, TCustomer.Create (CustKey));
       Dataset.Next;
    end;
end;
```

范例程序中包含了一个主窗体，以及一个数据模块，里面放置了一个 ClientDataSet 组件。主窗体里面有一个 ListView 组件，当用户点选窗体上的按钮时，就会加载数据。

笔记

你可能会想要用一个实际的数据集合来换掉 ClientDateSet 组件，把这个范例扩展成真的可以使用的程序。这样就可以透过查询一个键值和每一个实际的 TCustomer 对象的数据进行关连了。我有类似的源码，但是在这里介绍的话有点离题太远，目前这个范例只是一个实验性的泛型 dictionary 类别而已。

把数据加载到 dictionary 之后，btnPopulateClick 方法会把 dictionary 的键值做列举:

```
procedure TFormCustomerDictionary.btnPopulateClick( Sender: TObject);
var
    Custkey: TCustomerKey;
    ListItem: TListItem;
begin
    DataModule1.ClientDataSet1.Active := True;
    CustDict.LoadFromDataSet(DataModule1.ClientDataSet1);

    for custkey in CustDict.Keys do
    begin
      ListItem := ListView1.Items.Add;
      ListItem.Caption := Custkey.Company;
      ListItem.SubItems.Add(FloatTOStr (custkey.CustNo));
      ListItem.Data := Custkey;
    end;
end;
```

这会把 ListView 组件的前两个字段填入数据，所有在 dictionary 的 key 当中的数据都会被显示出来。当用户点选 ListView 组件中的任何一个项目时，程序就会填入第三个字段:

```
procedure TFormCustomerDictionary.ListView1SelectItem(
     Sender: TObject; Item: TListItem; Selected: Boolean);
var
    ACustomer: TCustomer;
begin
    ACustomer := CustDict.Items [Item.data];
```

```
    Item.SubItems.Add(IfThen (
            ACustomer.State <> '',
            ACustomer.State + ', ' + ACustomer.Country,
            ACustomer.Country));
end;
```

上面的这个方法会取得该键值相对的对象，并使用对应的数据。在背景里，特定的对象第一次被使用到时，存取属性的方法就会把完整的数据加载到 TCustomer 对象。

Dictionary v.s. 字符串列表

经过了这么多年，许多 Object Pascal 的开发人员，包含我自己，都过度使用了 TStringList 类别(译者:我也是)。我们不只可以用 TStringList 来储存字符串，还可以用来储存成对的名称与内容。不仅如此，还可以用来储存成对的名称与对象，更可以透过搜寻名称来取得对应的对象。从简单的介绍来看，字符串列表看起来比泛型好理解多了，简直就像个瑞士万用刀一样了。

特定、聚焦的容器类别，的确相对是比较好的选择。举例来说，一般的 TDictionary 类别，以字符串当做键值，以对象当做内容，会比 TStringList 来的好，至少有两个优点:程序清晰明白、也比较安全。因为当中会牵涉到的型别转换比较少，且执行速度比较快，这当然是假设该 dictionary 是使用哈希表(Hash table)的。

要表现出这些不同点，我也写了一个相对简单的范例项目，叫做 StringListVs Dictionary。它的主窗体储存了两个相同的列表，宣告如下:

```
private
    FList: TStringList;
    FDict: TDictionary<string,TMyObject>;
```

这两个列表会随机的填入数据，但相同的节点则会循环填入，重复执行的源码内容如下:

```
FList.AddObject (AName, AnObject);
FDict.Add (AName, AnObject);
```

窗体上的两个按钮，分别是取得该列表中的每个元素，以及透过名称进行搜寻。两个方法都会搜寻整个字符串列表来找到对应的值，但是第一个方

法则是会找出字符串列表中的对象，第二个方法会使用 dictionary。请注意，使用第一个方法时，我们需要用到 as 型别转换来变成特定的型别，dictionary 则是已经跟类别绑定了。以下是这两个方法的主要循环:

```
TheTotal := 0;
for I := 0 to FList.Count -1 do
begin
    aName := FList[I];
    // now search for it
    anIndex := FList.IndexOf (aName);
    // get the object
    anObject := FList.Objects [anIndex] as TMyObject;
    Inc (TheTotal, anObject.Value);
end;

TheTotal := 0;
for I := 0 to FList.Count -1 do
begin
    aName := FList[I];
    // get the object
    AnObject := FDict.Items [aName];
    Inc (TheTotal, AnObject.Value);
end;
```

我不想依序存取字符串，但需要得知存取了几次以后才在已经排序过后的字符串列表中找到要找的内容(字符串列表的搜寻是以二元搜寻法进行)，来跟使用哈希键值的 dictionary 做个比较。毫无意外的，dictionary 当然比较快，以下是测试时用 ms 为单位得到的数字:

```
Total: 99493811
StringList: 2839
Total: 99493811
Dictionary: 686
```

这个结果很明显是相同的，但时间上差的可远了，同样处理一百万笔数据，使用 dictionary 搜寻 1 亿笔数据，大概只花了相同程序使用字符串列表四分之一的时间。

泛型接口

在『对 TList<T>进行排序』那一小节里，对预先定义的接口比较奇怪的用法，应该还有印象吧，当中有提到泛型的宣告。当中的技术细节值得我们仔细一谈，因为它制造了相当明确的机会。

第一个要提到的技术部分是它完美的合法定义了一个泛型接口，就像我们在 GenericInterface 范例项目中所做的:

```
type
  IGetValue<T> = interface
    function GetValue: T;
    procedure SetValue (Value: T);
  end;
```

笔记

这是 IntfCotraints 范例项目中，IGetValue 接口的泛型版本，我们在本章稍早的『接口守则』那一节里面介绍过。在该案例中，这个接口是记录一个整数的数值，而在此则改为泛型。

注意到其中和标准接口的差异，在泛型接口中，我们不需要定义明确的 GUID 作为该接口的代号(或称 IID)。编译器会在我们为泛型接口进行实体化的时候主动建立一个 IID，也算是隐藏式的宣告。事实上，我们不用为泛型接口建立特定的实体也可以实作它，只要定义一个泛型类别，让该类别实作这个泛型接口即可：

```
type
  TGetValue<T> = class (TInterfacedObject, IGetValue<T>)
  private
    FValue: T;
  public
    constructor Create (Value: T);
    destructor Destroy; override;
    function GetValue: T;
    procedure SetValue (Value: T);
  end;
```

在建构函式指派该对象的初始值的时候，解构函式的唯一要求是把该对象先记录下来注记成未来要释放的对象。我们要建立这个泛型类别的实体(在背景建立一个接口型别的特定实体)，可以这么写：

```
procedure TFormGenericInterface.btnValueClick( Sender: TObject);
```

```
var
    AVal: TGetValue<string>;
begin
    AVal := TGetValue<string>.Create (Caption);
    try
        Show ('TGetValue value: ' + AVal.GetValue);
    finally
      AVal.Free;
    end;
end;
```

我们在 IntfConstraint 范例项目里面看过一个替代的作法，是用一个特定型别的接口变量，然后明确的写明该接口型别的定义(不像一般我们在写型别时都是隐藏的写法):

```
procedure TFormGenericInterface.btnIValueClick( Sender: TObject);
var
    AVal: IGetValue<string>;
begin
    AVal := TGetValue<string>.Create (Caption);
    Show ('IGetValue value: ' + AVal.GetValue);
    // freed automatically, as it is reference counted
end;
```

当然，我们也可以定义一个特定的类别来实作这个泛型接口，就像以下这个案例(节录自 GenericInterface 范例项目):

```
type
    TButtonValue = class (TButton, IGetValue<Integer>)
    public
        function GetValue: Integer;
        procedure SetValue (Value: Integer);
        class function MakeTButtonValue (Owner: TComponent;
                Parent: TWinControl): TButtonValue;
    end;
```

请留意，TGetValue<T>泛型类别实作了 IGetValue<T>接口，TButtonValue 特定类别实作了 IGetValue<Integer>这个特定接口。在前面的例子当中，这个接口会被重新对应到组件的 Left 属性:

```
function TButtonValue.GetValue: Integer;
begin
    Result := Left;
```

```
end;
```

在上面的这个类别中，MakeTButtonValue 这个类别函式是一个已经准备好随时可以用来建立该类别对象的方法。这个方法会在主窗体的地三个按钮里面被用上：

```
procedure TFormGenericInterface.btnValueButtonClick( Sender: TObject);
var
    IVal: IGetValue<Integer>;
begin
    IVal := TButtonValue.MakeTButtonValue (self, ScrollBox1);
    Show ('Button value: ' + IntToStr (IVal.GetValue));
end;
```

虽然这跟泛型类别完全没关系，以下是 MakeTButtonValue 这个类别函式的源码:

```
class function TButtonValue.MakeTButtonValue(
        Owner: TComponent; Parent: TWinControl): TButtonValue;
begin
    Result := TButtonValue.Create(Owner);
    Result.Parent := Parent;
    Result.SetBounds(Random (Parent.Width),
          Random (Parent.Height), Result.Width, Result.Height);
    Result.Caption := 'btnv';
end;
```

预先定义的泛型接口

现在我们已经看过了如何定义泛型接口，以及如何把泛型接口跟泛型类别或一般类别结合的方法，我们再回头看一下 Generics.Default 单元文件。这个单元文件定义了两个泛型比较用的接口:

- IComparer<T>拥有一个 Compare 方法
- IEqualityComparer<T>则拥有 Equals 跟 GetHashCode 方法

这些接口则由一些泛型类别或者一般类别实作出来，兹列于下(实作的内容就不列了):

```
type
    TComparer<T> = class(TInterfacedObject, IComparer<T>)
    TEqualityComparer<T> = class(
      TInterfacedObject, IEqualityComparer<T>)
```

```
TCustomComparer<T> = class(TSingletonImplementation,
    IComparer<T>, IEqualityComparer<T>)
TStringComparer = class(TCustomComparer<string>)
```

在上面的列表中，我们可以看到用于实作泛型接口的基础类别要不是传统的参考计数功能的 TInterfacedObject 类别，就是 TSingletonImplementation 类别(从 Delphi 11 开始，它只是 TNoRefCountObject 类别的别名了)。这是一个名字很奇怪的类别，因为它只提供了 IInterface 的基本实作，而且没有用到参考计数。

笔记 singleton 这个名词一般是用来定义只能被建立出一个实体对象的类别，而不是不具备参考计数功能的类别。我觉得这个名字取得真的不恰当。

就像我们在本章『对 TList<T>进行排序』那一节所提到的，这些比较用的类别会被泛型容器使用到。为了让程序能不要太复杂，Generics.Default 单元文件里面大量使用了匿名方法，所以您或许需要先看一下下一章再回来重新阅读一次。

在 Object Pascal 里面的智能指标(Smart Pointer)

开始接触泛型的时候，我们一开始可能会误以为 Object Pascal 大多数是用它来处理集合对象的。当然处理集合对象对泛型类别来说是最基本的情境，在文件或者书籍里面的第一个范例也大多都是使用集合或者容器类别来说明泛型。在本章的最后一个范例里，我们会介绍非集合类的泛型类别，也就是定义一个智能指标(Smart Pointer)。

如果您原本就很习惯写 Object Pascal，您可能完全没听过智能指标，这个概念是从 C++编程语言而来。在 C++里面，我们可以有指向对象的指针，透过这个指标，我们可以直接进行人工内存管理，区域对象变量则会自动被管理，但有其他的限制(包含缺乏对多型的支持)。智能指针的概念是使用在区域中被管理的对象来处理指向我们会用到的实际对象指针的生命周期。如果这听起来很太复杂，我希望 Object Pascal 版可以协助厘清这个概念。

笔记 在 OOP 里面，多型(polymorphism)这个名词是用来标注以下情形：我们把一个衍生类别的对象指派给基础类别的变量，并且呼叫基础类别中的虚拟方法，最后可能会呼叫到特定子类别所重新实作的版本。

使用智能指针的记录

在 Object Pascal 当中的对象都是以参考进行管理，而记录的生命周期跟宣告了该记录变量的方法紧密结合着。当方法执行结束，该记录的内存空间也会自动被清除掉。所以我们可以透过记录来管理 Object Pascal 对象的生命周期。

在 Delphi 10.4 之前，Object Pascal 的纪录并没有提供在释放的时候执行自定源码的功能，这个功能是随着『受管理的记录』推出的新功能。旧式的机制中，是在记录中使用接口字段，随着接口字段也由系统进行管理，用来实作接口的对象也有了自己的参考计数管理。

另一个考虑则是，我们想要使用标准的记录或是泛型记录呢?使用 TObject 的标准记录的话，我们可以在需要的时候删除对象，所以一般情形下这已经够用了。使用泛型版本的话，我们则会有两个额外的好处：

- 泛型智能指针可以回传当中的对象参考，所以我们不用一直维护着两份参考
- 泛型智能指标可以透过无参数的建构函式，自动建立容器对象

在这里我们要介绍两个范例，使用泛型记录来实作智能指针，虽然有点复杂，最起始的城市还是一个包含对象守则的泛型记录:

```
type
  TSmartPointer<T: class> = record
  strict private
      FValue: T;
      function GetValue: T;
  public
      constructor Create(AValue: T);
      property Value: T read GetValue;
  end;
```

记录当中的 Create 跟 GetValue 方法就是单纯的指派、读回该值。以下这段源码的使用情境，是建立一个对象，然后建立一个智能指标来跟它交换，并允许使用智能指标指向该嵌入对象，并且呼叫它的方法(请看最后一行源码)：

```
var
  SL: TStringList;
begin
   SL := TStringList.Create;
   var SmartP:   TsmartPointer<TStringList>.Create (SL);
   SL.Add('foo');
   SmartP.Value.Add ('bar');
```

当我们已经让程序运作了，如果没有使用智能指标，上面这段源码可是会导致内存泄漏的情况发生喔。事实上，记录会自动在离开程序区块后被释放，但它没有释放内部对象。

用泛型受管理的纪录来实现智能指针

与智能指针记录最相关的行为就是它的终止过程(finalization)，在以下的源码中可以看到，我们也加入了起始过程，在当中把对象参考指向 nil，理想中，我们应该在起始过程中避免任何指派的作业(对内部对象进行多重链接会需要相对复杂的参考计算器制)，但这是不可能的，所以我在起始过程中加入了这个指派作业，以避免该程序一直行就触发例外事件。

以下是泛型受管理的纪录完整的宣告:

```
type
  TSmartPointer<T: class, constructor> = record
  strict private
      FValue: T;
      function GetValue: T;
  public
      class operator Initialize(out ARec: TSmartPointer<T>);
      class operator Finalize(var ARec: TSmartPointer<T>);
      class operator Assign(var ADest: TSmartPointer<T>; const [ref] ASrc: TSmartPointer <T>);
      constructor Create(AValue: T);
      property Value: T read GetValue;
  end;
```

请留意除了类别守则外，泛型记录也有建构函式守则，因为我想要建立泛型数据型别的对象。这会发生在 GetValue 方法被呼叫，且该字段还没有被初始化的时候，以下是所有方法的完整源码:

```
constructor TSmartPointer<T>.Create(AValue: T);
begin
    FValue := AValue;
end;
class operator TsmartPointer<T>.Initialize( out ARec: TSmartPointer <T>);
begin
    ARec.FValue := nil;
end;
class operator TsmartPointer<T>.Finalize( var ARec: TSmartPointer<T>);
begin
    ARec.FValue.Free;
end;
class operator TsmartPointer<T>.Assign( var ADest: TSmartPointer <T>;
    const [ref] ASrc: TSmartPointer <T>);
begin
    raise Exception.Create('Cannot copy or assign a TSmartPointer<T>');
end;
function TSmartPointer<T>.GetValue: T;
begin
    if not Assigned(FValue) then FValue := T.Create;
        Result := FValue;
end;
```

上面的源码是 SmartPointsMR 范例项目的一部分，项目中包含了如何使用智能指针的一些范例。第一个是把我们前几页的范例改写成比较严谨的作法:

```
procedure TFormSmartPointers.BtnSmartClick(Sender: TObject);
var
        SL: TStringList;
begin
        SL := TStringList.Create;
        var SmartP := TSmartPointer<TStringList>.Create (SL);
        SL.Add('foo');
        SmartP.Value.Add('bar');
        Log ('Count: ' + SL.Count.ToString);
```

```
end;
```

然而，有鉴于泛型智能指标支持对指定型别对象自动建构的功能，我们也可以透过明确宣告该变量指向字符串列表的作法，用以下的源码来建立它:

```
procedure TFormSmartPointers.BtnSmartShortClick(Sender: TObject);
var
        SmartP: TSmartPointer<TStringList>;
begin
        SmartP.Value.Add('Foo');
        SmartP.Value.Add('Bar');
        Log ('Count: ' + SmartP.Value.Count.ToString);
end;
```

在这个程序中，我们可以验证所有对象真的被释放，并且在初始程序中把全局变量 ReportMemoryLeaksOnShutdown 设定为 True 来确保没有内存泄漏的情况发生。为了凸显这个问题，在程序中我们特别做了一个按钮，点击它就会造成内存泄漏，程序执行就会异常结束。

用泛型记录与接口来实现智能指针

如我之前提过的，在 Delphi 10.4 推出受管理的记录这个功能之前，可以用来实现智能指标的作法是使用接口，因为记录会自动释放被接口字段参考的对象。虽然这个功能比较不受重视，但它的确提供了一些额外的功能，例如隐晦的转换功能。

也因为这主题还是有一些有趣的、复杂的范例，我决定还是保留它，但描述的篇幅短一点(请参考 SmartPointers 范例项目)。

要用接口来实作智能指针，我们可以写一个内部的支持类别、让它跟接口绑定，并使用参考计数的机制来决定何时要释放对象。这个内部类别的源码会像这样:

```
type
  TFreeTheValue = class (TInterfacedObject)
  private
      FObjectToFree: TObject;
  public
      constructor Create(AnObjectToFree: TObject);
      destructor Destroy; override;
```

```
  end;


constructor TFreeTheValue.Create( AnObjectToFree: TObject);
begin
    FObjectToFree := anObjectToFree;
end;
destructor TFreeTheValue.Destroy;
begin
    FObjectToFree.Free;
    inherited;
end;
```

我把这个类别当成泛型智能指标宣告成嵌套类型。我们需要在智能指标泛型型别上所做的，好让这个功能能够使用的，是新增一个接口参考，并用 TFreeTheValue 对象进行初始化，以参照到内部对象：

```
type
    TSmartPointer<T: class> = record
    strict private
        FValue: T;
        FFreeTheValue: IInterface;
        function GetValue: T;
    public
        constructor Create(AValue: T); overload;
        property Value: T read GetValue;
    end;
```

虚构的建构函式变成了:

```
constructor TSmartPointer<T>.Create(AValue: T);
begin
    FValue := AValue;
    FFreeTheValue := TFreeTheValue.Create(FValue);
end;
```

透过前述的源码，我们现在就可以在程序里面写以下的源码，而不会造成内存泄漏了(这段源码也跟我一开始提到过，并使用在受管理的纪录那一节介绍过的很像):

```
procedure TFormSmartPointers.btnSmartClick( Sender: TObject);
var
```

```
    SL: TStringList;
    SmartP: TSmartPointer<TStringList>;
begin
    SL := TStringList.Create;
    smartP.Create (SL);
    SL.Add('Foo');
    SL.Add('Bar');
    Show ('Count: ' + IntToStr (SL.Count));
end;
```

在方法的最后，smartP 记录会被释放掉，同时也会连带使得内部接口对象一起被清除，并释放掉 TStringList 对象。

笔记 虽然可能会发生例外，但这段程序还是可以执行的。事实上，当我们使用到受管理的型别时，编译器还是会自动在编译时在这些源码中加入 try-finally 区块。就像在刚刚这案例中，记录中使用了接口字段。

加入隐晦转换(Adding Implicit Conversion)

使用受管理的纪录方案，我们需要额外花些功夫来避免记录复制的功能，这会需要多加一些源码手动控制参考计算器制，并且把结构做的更复杂。然而这个功能在接口为主的作法中是内建的，我们在这个模型中加入转换的功能，这样就可以简化起始程序跟数据结构的建立。我要特别在指派功能中把目标对象加入用 Implicit 转换为智能指标的步骤:

```
class operator TSmartPointer<T>. Implicit(AValue: T): TSmartPointer<T>;
begin
     Result := TSmartPointer<T>.Create(AValue);
end;
```

透过这段源码(以及使用 Value 数据字段)我们现在可以让源码更为精简了，精简过的版本如下:

```
var
  SmartP: TSmartPointer<TStringList>;
begin
    SmartP := TStringList.Create;
    SmartP.Value.Add('foo');
    Show ('Count: ' + IntToStr (SmartP.Value.Count));
```

作为替代方案，我们可以使用一个 TStringList 变量，并使用一个比较复杂的建构函式来位智能指针记录进行初始化，即使没有特别写明其参考：

```
var
   SL: TStringList;
begin
    SL := TSmartPointer<TStringList>.
    Create(TStringList.Create).Value;
    SL.Add('Foo');
    SL.Add('Bar');
    Show ('Count: ' + IntToStr (SL.Count));
```

正如我们已经开始进行的，我们也可以定义完全相反的转换，并使用转换符号，而不要用 Value 属性：

```
class operator TSmartPointer<T>.
       Implicit(AValue: T): TSmartPointer<T>;
begin
    Result := TSmartPointer<T>.Create(AValue);
end;


var
    SmartP: TSmartPointer<TStringList>;
begin
    SmartP := TStringList.Create;
    TStringList(SmartP).Add('Bar');
```

现在，我们应该也要注意到上面这段源码里面使用了虚构建构函式，但这对记录并不需要。我们需要的是找到一个方法对内部对象进行初始化，可能是呼叫它的建构函式，当我们第一次使用到它的时候。

我们无法检查内部对象是否已经被指派，因为记录(跟类别不同)并不会被初始化为 0。然而我们可以从接口变量来测试，接口变量会被初始化。或者我们也可以用额外的建构函式源码，把该记录用 0 全部填满作初始化。

比较几个智能指标的方案

使用『受管理的记录』来实作智能指标的方案，相对来说是比较简单，也是比较有效率的，但『接口为基础』的实作版本提供了可以转换的功能。

尽管两个版本都有各自的优点，我个人比较喜欢『受管理的纪录』的实作版本。

笔记 如果大家想要在这个议题上有更清楚的分析，以及更深入的实作范例(超出本书的范围)，我推荐以下的部落格文章，由 Erik van Bilsen 所著: https://blog.grijjy.com/2020/08/12/custom-managed-records-fro-smart-pointers/

以泛型处理协同变异(Covariant)回传型别

在 Object Pascal 当中(以及大多数静态面向对象编程语言)，一般来说一个方法可以回传一个类别的对象，但我们不能把一个对象覆写成另一个衍生类别之后，当成该类别来回传。这个作法称为『协同变异回传型别』，在一些编程语言里面是有明确支持的，例如 C++。

关于 Animals, Dogs,跟 Cats

从源码的词语来看，如果 TDog 是继承自 TAnimal，我们就可以呼叫这些方法:

```
function TAnimal.Get (name: string): TAnimal;
function TDog.Get (name: string): TDog;
```

在 Object Pascal 里面我们不能让虚拟方法回传不同的值，我们也不能对回传型别进行多载，只能在参数上进行多载。我们来透过简单的范例介绍完整的源码吧，以下是三个我们会用到的类别:

```
type
  TAnimal = class
  private
    FName: string;
    procedure SetName(const Value: string);
  public
     property Name: string read FName write SetName;
  public
     class function Get (const aName: string): TAnimal; function ToString: string; override;
  end;
```

```
  TDog = class (TAnimal)
  end;

  TCat = class (TAnimal)
  end;
```

这两个方法的实作源码非常简单，我们注意到类别函式是实际上用来建立新组件的，它会从内部呼叫一个建构函式。

理由是我不想直接建立一个建构函式，这已经介绍过很多次了，建构函式是一个类别用来建立对象的方法(或者类别架构)。

源码如下:

```
class function TAnimal.Get(const aName: string): TAnimal;
begin
  Result := Create;
  Result.fName := aName;
end;

function TAnimal.ToString: string;
begin
  Result := 'This ' + Copy (ClassName, 2, maxint) +
    ' is called ' + FName;
end;
```

现在我们可以用以下的源码来使用这个类别了，这不是我喜欢的用法，假设我们要把回传来的结果转成适当型别:

```
var
  ACat: TCat;
begin
  ACat := TCat.Get('Matisse') as TCat;
  Memo1.Lines.Add (ACat.ToString);
  ACat.Free;
```

再次强调，我想做的是让 TCat.Get 的回传值能够被指派到 TCat 型别的参考，而不要多做一次转型，我们该怎么做呢?

回传泛型结果的方法

泛型可以协助我们解决这个问题。并不是泛型型别，泛型型别是最常用的泛型形式。但我们已经在本章前面的篇幅里面介绍过提供非泛型型别的泛型方法了。我们可以在 TAnimal 类别中加入的是一个以泛型型别作为参数的方法：

```
class function GetAs<T: class> (const aName: string): T;
```

这个方法会要求一个泛型型别作为参数，在这个例子里泛型型别是个类别(或实际的型别)，并回传该型别的物件。简单的实作源码如下:

```
class function TAnimal.GetAs<T>(const aName: string): T;
var
    res: TAnimal;
begin
    res := Get (aName);
    if res.inheritsFrom (T) then
       Result := T(res)
    else
       Result := nil;
end;
```

现在我们可以建立一个实际的案例了，透过 as 转型，不过我们还是可以把型别当成参数传进去:

```
var
    ADog: TDog;
begin
    ADog := TDog.GetAs<TDog>('Pluto');
    Memo1.Lines.Add (ADog.ToString);
    ADog.Free;
```

回传不同类别的衍生对象

当我们回传相同类别的对象时，我们可以直接改上面的源码来当做建构函式。但使用泛型来取得协同变异回传型别是更有弹性的。事实上我们可以用它来回传不同类别的对象，甚至是不同架构的类别：

```
type
    TAnimalShop = class
       class function GetAs<T: TAnimal, constructor> ( const aName: string): T;
```

```
    end;
```

笔记

像这样的类别，被用来为其他类别(或者不只一个类别，视其参数或者状态而定)建立对象，通常称为类别工厂(class factory)。

我们现在可以使用这个特定的类别守则(在该类别来说，也许不可能达成)，透过确立建构函式守则，好让泛型方法可以用来建立特定的类别对象：

```
class function TAnimalShop.GetAs<T>(const AName: string): T;
var
    Res: TAnimal;
begin
    Res := T.Create;
    Res.Name := AName;
    if Res.inheritsFrom (T) then
       Result := T(res)
    else
        Result := nil;
end;
```

要注意，现在在这个呼叫的写法中，我们不用重复该类别两次了:

```
ADog := TAnimalShop.GetAs<TDog>('Pluto');
```

15:匿名方法

Object Pascal 同时包含了程序型别(也就是可以宣告指向程序或函式的指针型别)以及方法指标(也就是可以宣告指向方法的指标)。

笔记 关于程序型别的相关知识，我们在第四章有详尽的介绍。而关于方法指标型别，则在第十章里面有详细的介绍。

虽然您可能不常直接使用匿名方法，但这是 Object Pascal 当中每个开发人员都会用到的关键功能喔。事实上，方法指针型别是控制组件与视觉组件中事件处理程序的基础：每当我们宣告了一个事件处理程序，即使只是简单的 Button1Click，事实上我们就宣告了一个方法，会用来跟一个事件连结(在这个例子里面是跟 OnClick 事件连结)，连结的关键就是方法指标。

匿名方法延伸了这个功能，让我们可以把一个方法的实际源码当成参数来传递，而不用先定义一个方法，然后把方法的名字当成参数了。然而这并不是唯一的差异。让匿名方法跟其他技术不同的地方，还有它对局部变量生命周期的管理。

上述的定义跟其他语言里面所称的 Closure 功能(例如 Javascript)一样。如果 Object Pascal 的匿名方法事实上就是这些编程语言里面所说的 Closure，为什么 Object Pascal 要用跟其他语言不一样的名词?原因是两个不同语言中使用了不同的名词。在 Embarcadero 所提供的 C++编译器所称的 closure 是用来称呼 Object Pascal 所称的事件处理程序。匿名方法在这几年里面已经用不同的形式、不同的名字，在一些编程语言里面出现在世界里，大多是标注式的动态语言。我自己也有用 JavaScript 的 closure 来开发的经验，主要是使用 jQuery 函式库跟 AJAX 来呼叫。对应的功能在 C#里面则是被称为匿名委任。

但在这里我不想花太多时间来比较 closure 跟不同语言当中的相关技术，我们只描述这个技术在 Object Pascal 里面的工作细节。

笔记 从比较高阶的观点来看，泛型允许源码可以用型别作为参数，而匿名方法则是允许源码可以用方法作为参数。

匿名方法的语法和语意

在 Object Pascal 里的匿名方法，是一种*在表达式的写法中建立方法内容*的机制。这个定义听起来很神秘，也相当精确的强调它与方法指标的关键差异，也就是*表达式写法的内容*。在我们介绍到内容之前，我们先从很简单的范例程序开始吧(本节的程序片段几乎都包含在 AnonymFirst 范例项目里面了)。以下是匿名方法型别的宣告，我们可以从这里看出 Object Pascal 的确是一个强型别的语言:

```
type
    TIntProc = reference to procedure (N: Integer);
```

这个宣告跟方法指标型别的差异只在用于宣告的关键词:

```
type
    TIntMethod = procedure (N: Integer) of object;
```

匿名方法变量

以最简单的案例来看，一旦我们宣告了匿名方法型别，当然就可以用这个型别来宣告变量，然后指派一个型别兼容的匿名方法给它，并且后过这个变量来呼叫该方法：

```
procedure TFormAnonymFirst.btnSimpleVarClick( Sender: TObject);
var
    anIntProc: TIntProc;
begin
    anIntProc :=
    procedure (N: Integer)
    begin
        Memo1.Lines.Add (IntToStr (N));
    end;
    anIntProc (22);
end;
```

请记得上例中，用来指派一个实际程序的语法，它是把内嵌的源码指派给变数喔。

匿名方法的参数

当成一个有趣的范例(或许用更让人惊讶的语法)，我们可以把一个匿名方法作为参数传给一个函式。假设我们有一个要以匿名方法当成参数的函式:

```
procedure CallTwice (Value: Integer; AnIntProc: TIntProc);
begin
    AnIntProc (Value);
    Inc (Value);
    AnIntProc (Value);
end;
```

这个函式里面会呼叫这个当成参数被传进来的匿名方法两次，把两个不同的整数数值传递进去，第一个数字会被当成第二个参数(匿名方法)的参数传递进去处理。我们呼叫这个函式的时候，只要把一个匿名方法当成第二个参数传进去即可，它的语法非常简单、直接：

```
procedure TFormAnonymFirst.btnProcParamClick( Sender: TObject);
begin
    CallTwice (48,
            procedure (N: Integer)
            begin
               Show (IntToHex (N, 4));
            end);

    CallTwice (100,
                procedure (N: Integer)
                begin
                   Show (FloatToStr(Sqrt(N)));
                end);
end;
```

从语法的观点来看，我们要注意到在小括号里面要把程序当成参数传进去时，最后不用加上分号喔。实际的源码里面，是呼叫了 IntToHex，以 48 跟 49 和 FloatToStr 来处理 100 跟 101 的平方根，得到了以下的结果:

```
0030
0031
10
10.0498756211209
```

使用局部变量

我们也可以透过方法指标来达成一样的功能，虽然语法比较不同，也比较不容易理解。匿名方法最明显的不同，是匿名方法当中在执行源码时使用局部变量的方法。先来一段范例：

```
procedure TFormAnonymFirst.btnLocalValClick( Sender: TObject);
var
    ANumber: Integer;
begin
    ANumber := 0;
    CallTwice (10,
        procedure (N: Integer)
        begin
            Inc (ANumber, N);
        end);
    Show (IntToStr (ANumber));
end;
```

这里所用的范例仍然呼叫 CallTwice 程序，使用了区域参数 n，也从被呼叫的程序当中使用了一个名为 ANumber 的局部变量。效果是什么?这两次呼叫匿名方法的过程中，都会改变局部变量，把传入的参数加进这个变量中，第一次执行的时候是 10，第二次执行的时候则是 11，所以两次执行完以后，ANumer 的值就变成了 21。

展延局部变量的生命周期

前例中显示了一个有趣的效应，但透过一连串巢状函式的呼叫，这个局部变量的变化也就不令人惊讶了。然而匿名方法的强大，是在于它可以使用局部变量，并且展延局部变量的生命周期直到匿名方法不再需要它。范例程序就可以证明，不用再多用文字说明了。

笔记 很明显的，从技术观点来看，匿名方法会把局部变量跟参数都复制到自己被呼叫的当时所使用的 heap，并维持这些变量的生命周期直到匿名方法执行结束为止。

我已经透过类别自动完整的功能，在 TFormAnonymFirst 窗体类别(属于 AnonymFirst 范例项目)中加入了一个属性，其类别是匿名方法指标型别(嗯…实际上我已经在该项目中使用了相同的匿名方法指标型别了):

```
  private
      FAnonMeth: TIntProc;
      procedure SetAnonMeth(const Value: TIntProc);
  public
      property AnonMeth: TIntProc
        read FAnonMeth write SetAnonMeth;
```

接着，我们在程序的窗体画面上加了两个按钮，第一个储存了使用局部变量的匿名方法属性，就像前一个 BtnLocalValclick 方法一样:

```
procedure TFormAnonymFirst.BtnStoreClick( Sender: TObject);
var
   ANumber: Integer;
begin
    ANumber := 3;
    AnonMeth :=
        procedure (N: Integer)
        begin
            Inc (ANumber, N);
            Show (IntToStr (ANumber));
        end;
end;
```

当这个方法执行时，匿名方法并没有被执行，只是被储存下来了。局部变量 aNumber 当时初始化之后，内容是 3，并没有被变更，接着这个方法执行完毕，然后被从内存移除了。至少这是我们从标准的 Object Pascal 源码的语法所得到的理解。

我们加在窗体面的第二个按钮，就会实质呼叫储存在 AnonMeth 属性里面的匿名方法了:

```
procedure TFormAnonymFirst.btnCallClick(Sender: TObject);
begin
   if Assigned (AnonMeth) then
   begin
       CallTwice (2, AnonMeth);
   end;
end;
```

这段源码被执行的时候，会呼叫使用到局部变量 ANumber 的匿名方法，但这个时候 ANumber 这个变量在内存当中已经不复存在了。然而，因为匿名方法的源码中有使用到 ANumber，这个情形会让 ANumber 的生命周期被延用到匿名方法结束。

我们再延伸举证，按下 Store 按钮一次，Call 按钮两次，我们就可以看到该变量的内容变化如下:

```
5
8
10
13
```

笔记

使用这个操作步骤的原因，是这个变量开始的时候内容是 3，每次点击时，第一次呼叫 CallTwice 来传递匿名方法(变量值会被加 2)，第二则会多加 1(所以第二次呼叫就会被加 3)。

现在，再按一次 Store，然后再按 Call。发生了什么事?为什么局部变量的内容被重设了?因为新的匿名方法实体被指派了，旧的匿名方法实体就被删除掉了(包含它的执行内容)，新的执行内容就被储存下来，连同新的局部变量实体一起。完整的程序 Store-Call-Call-Store-Call 程序：

```
5
8
10
13
5
8
```

这是匿名方法行为的含义，重组了一些其他编程语言所做的，这使得匿名方法成为一个很强大的编程语言功能，透过这个功能，我们可以实作出一些以前根本难以完成的东西。

在背景里的匿名方法

如果变量连带储存的功能是匿名方法的众多功能中，我们可能最常用到的功能之一，但其中还有许多技术是值得我们深入探讨的，在我们开始仔细深入到一些实际案例之前。

(潜在的)漏掉括号

请注意上面的源码，我们使用了 AnonMeth 这个属性来储存匿名方法，而没有直接呼叫它，要呼叫它的话，我们应该这么写:

```
AnonMeth (2)
```

差异之处很明显，我们需要传递一个参数给这个方法。如果匿名方法没有括号就更容易混淆了，如果我们这样宣告:

```
type
    TAnyProc = reference to procedure;
var
    AnyProc: TAnyProc;
```

呼叫 AnyProc 必须一定要写空括号了，不然的话编译器会以为我们是要取得该函式的参考，而不是要呼叫它：

```
AnyProc ();
```

当我们呼叫一个会回传匿名方法的含事实，类似的效应也会发生，就像以下的案例(节录自 AnonymFirst 范例项目)：

```
function GetShowMethod: TIntProc;
var
    X: Integer;
begin
     X := Random (100);
     ShowMessage ('New x is ' + IntToStr (X));
     Result :=
         procedure (N: Integer)
         begin
              X := X + N;
              ShowMessage (IntToStr (X));
         end;
end;
```

现在问题只剩我们是怎么呼叫它的?如果我们简单的直接呼叫:

```
GetShowMethod;
```

编译会通过，也可以执行，但实际上执行的动作只是进行了匿名方法指派的动作，把匿名方法从该函式回传回来而已。

我们要怎么真正呼叫到匿名方法，并将参数传递进去呢?选项之一，是使用暂存的匿名方法变量：

```
var
  Ip: TIntProc;
begin
   Ip := GetShowMethod();   // The parentheses are needed!!!
   Ip (3);
```

请留意这个情况下，在 GetShowMethod 函数调用后面的小括号。如果我们省略不输入这个小括号(标准 Pascal 语法是可以忽略小括号的)，就会出现以下的错误讯息:

```
E2010 Incompatible types: 'TIntProc' and 'Procedure'
```

把小括号省略掉，编译器会以为我们想要指派 GetShowMethod 函式，而不是呼叫它所储存的 IP 方法指标。同样的，使用暂存变量也可能是这个案例的最佳选项，同时也把程序不由自主的变得复杂了些，单纯呼叫:

```
GetShowMethod(3);
```

这样是不能通过编译的，因为我们不能把参数传递到方法里面，我们得在第一个呼叫上面先加上一个空的小括号，然后再把整数参数写在第二个小括号里面，实在够奇怪的，变成要这样写:

```
GetShowMethod()(3);
```

匿名方法实作

在匿名方法实作的背地里，到底发生了什么事?由编译器所建立的源码是基于接口，透过名为 Invoke 的单一呼叫函式(隐藏的)，加上常用的参考技术技术(这对于决定匿名方法以及其实质内容的生命周期相当有用)。

深入内部实作的细节可能会很复杂，且得到的知识相对有限。我只想说，这样的实作是很有效率的，而且每个匿名方法只需要大概多 500 个字符。

换句话说，在 Object Pascal 里面，方法参考是以特别的方法接口来实作的，透过编译器产生的方法具备了相同的特征，而让方法参考来使用。接口的好处则是具备了参考计数以及自动释放的功能。

笔记 虽然用来实作匿名方法的接口看起来跟其他接口没什么差别，但编译器仍然会把这两种接口自行做出区隔，所以我们可以在源码当中把两者混在一起使用。

除了隐藏式的接口，每一次呼叫匿名方法时，编译器就会建立一个隐藏的对象，储存匿名方法的源码与其所使用的相关内容。这也是为什么我们在每次呼叫匿名方法的时候，都会得到一组被保留的变量了。

预先准备好的参考型别

每当我们把一个匿名方法用作参数时，我们就需要定义一个相对的参考指针数据型别。避免这种区域型别不断增加，Object Pascal 提供了一些预先准备好的参考指标型别，就放在 SysUtils 单元文件里面。

我们可以看一下下面的程序片段，大多数的型别定义都使用了参数化的型别，所以透过以下任何一种通用的宣告，我们都可以用来定义任一种可能的数据型别的参考指针型别：

```
type
    TProc = reference to procedure;
    TProc<T> = reference to procedure (Arg1: T);
    TProc<T1,T2> = reference to procedure (
        Arg1: T1; Arg2: T2);
    TProc<T1,T2,T3> = reference to procedure (
        Arg1: T1; Arg2: T2; Arg3: T3);
    TProc<T1,T2,T3,T4> = reference to procedure (
        Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4);
```

使用这些宣告，我们可以定义以下的匿名方法参数：

```
procedure UseCode (Proc: TProc);
function DoThis (Proc: TProc): string;
function DoThat (ProcInt: TProc<Integer>): string;
```

在前两个案例中，我们是传递不需要参数的匿名方法，在第三个案例中，则是传递一个要求一个整数参数的匿名方法:

```
UseCode (
    procedure
```

```
    begin
        ...
    end);
strRes := DoThat (
    procedure (I: Integer)
    begin
        ...
    end);
```

同样的，在 SysUtils 单元文件里面还定义了一系列匿名方法的型别，这些型别会回传一个通用的回传值:

```
type
    TFunc<TResult> = reference to function: TResult;
    TFunc<T,TResult> = reference to function (
      Arg1: T): TResult;
    TFunc<T1,T2,TResult> = reference to function (
      Arg1: T1; Arg2: T2): TResult;
    TFunc<T1,T2,T3,TResult> = reference to function (
      Arg1: T1; Arg2: T2; Arg3: T3): TResult;
    TFunc<T1,T2,T3,T4,TResult> = reference to function (
      Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4): TResult;
    TPredicate<T> = reference to function (
                Arg1: T): Boolean;
```

这些定义很广泛，我们可以任选需要一到四个参数的匿名方法，而这个匿名方法还可以包含回传值。最后一个定义跟第二个很像，但它用来处理的情形是比较常见的一个特殊案例，是接受一个通用参数，并回传布尔值的函式。

实战上的匿名函式

第一眼看上去，不太容易看出匿名方法的强大威力以及在什么情形下使用匿名方法才是最适合的。这就是我决定把重点放在一些实务上的影响，并开始提供进一步说明，而不要再提更多涵盖编程语言功能的实例了。

匿名事件处理程序

Object Pascal 最明确的功能之一，就是它使用了方法指标来实作事件处理程序。匿名方法可以用来跟新的行为、事件进行绑定，而无需宣告一个独立的方法也无须实作该方法。这样一来就可以让窗体省下一个额外字段，不用把参数从一个方法传到另一个方法去。

我们再来看另一个范例(AnonButton)，我已经在范例中的按钮上加入了一个匿名的点击事件，宣告一个适当的方法指标型别，并为一个自定的按钮类别(使用中介类别定义)加入一个新的事件：

```
type
    TAnonNotif = reference to procedure (Sender: TObject);

    // Interceptor class  中介类别
    TButton = class (FMX.StdCtrls.TButton)
    private
        FAnonClick: TAnonNotif;
        procedure SetAnonClick(const Value: TAnonNotif);
    public
        procedure Click; override;
    public
        property AnonClick: TAnonNotif  read FAnonClick write SetAnonClick;
end;
```

笔记

中介类别是和其基础类别有着相同名称的衍生类别。让两个类别有相同的名称是可能的，因为只要这两个类别位于不同的单元文件里面，他们的完整名称(单元文件名.类别名称)就会不同。宣告中介类别是很容易的，就像我们把一个按钮组件放到窗体上，然后再加一些控制事件的逻辑上去那么简单，不用在 IDE 里面安装什么组件了。唯一要注意的事项只有一个，就是要记得如果使用了中介类别，它一定是宣告在另一个单元文件里面，记得要使用中介类别的单元文件必须 use 宣告该类别的单元文件才行。

这个类别的源码相对的简单，只有在 Click 方法被呼叫的时候在标准处理程序进行前，把新的指标存起来而已(也就是呼叫 OnClick 事件处理程序，如果它有被定义的话):

```
procedure TButton.SetAnonClick(const Value: TAnonNotif);
begin
    FAnonClick := Value;
```

```
end;
procedure TButton.Click;
begin
    if Assigned (FAnonClick) then
      FAnonClick (self)
      inherited;
end;
```

我们要怎么使用这个新的事件处理程序呢?基本上我们可以指派一个匿名方法给它:

```
procedure TFormAnonButton.btnAssignClick( Sender: TObject);
begin
  BtnInvoke.AnonClick :=
  procedure (Sender: TObject)
  begin
    Show ((Sender as TButton).Text);
  end;
end;
```

现在看起来这写法没有什么重点，因为相同的效果可以用标准的事件处理程序来达成。不过接下来就有点不同了，匿名方法储存一个组件的参考，然后把它传递给事件处理程序，透过参考 Sender 这个参数来达成。

这也可以在它暂时指派给一个局部变量来完成，因为匿名方法的 Sender 参数会把 btnKeepRefClick 方法的 Sender 参数隐藏起来:

```
procedure TFormAnonButton.btnKeepRefClick( Sender: TObject);
var
  ACompRef: TComponent;
begin
  ACompRef:= Sender as TComponent; BtnInvoke.AnonClick :=
  procedure (Sender: TObject)
  begin
      Show ((Sender as TButton).Text +
          ' assigned by ' + ACompRef.Name);
  end;
end;
```

当我们点击了 BtnInvoke 按钮，我们会看到被指派给匿名方法处理程序的组件名字被显示在窗口标题上。

为匿名方法计时

开发人员常常会在已开发好的源码里面加入计算时间的源码，用来比较执行速度。假设我们有两段源码，想要比较看看它们执行上百万次所花的时间，我们可以用以下的源码来做，这是第六章里面的 LargeString 范例项目里面节录出来的：

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Str1, Str2: string;
    I: Integer;
    T1: TStopwatch;
begin
    Str1 := 'Marco ';
    Str2 := 'Cantu ';

    T1 := TStopwatch.StartNew;
    for I := 1 to MaxLoop do
      Str1 := Str1 + Str2;

    T1.Stop;
    Show('Length: ' + Str1.Length.ToString);
    Show('Concatenation: ' + T1.ElapsedMilliseconds.ToString);
end;
```

第二个方法的源码也很类似，但使用了 TStringBuilder 类别，而不是直接使用字符串连接。

现在我们可以用匿名方法的长处来建立一个时间点，然后把特定的源码当成参数传递，就像我在 AnonLargeStrings 范例项目中已经改版的源码一样。不使用持续的计时源码一再计算，我们可以写一个方法，包含着计时的源码，让源码被呼叫的时候可以呼叫不用小括号的匿名方法:

```
function TimeCode (nLoops: Integer; proc: TProc): string;
var
    T1: TStopwatch;
    I: Integer;
```

```
begin
    T1 := TStopwatch.StartNew;
    for I := 1 to nLoops do
        proc;
    T1.Stop;
    Result := T1.ElapsedMilliseconds.toString;
end;
procedure TForm1.Button1Click(Sender: TObject);
var
    Str1, Str2: string;
begin
    Str1 := 'Marco ';
    Str2 := 'Cantu ';
    Show ('Concatenation: ' +
      TimeCode (MaxLoop,
        procedure ()
        begin
          Str1 := Str1 + Str2;
        end));
    Show('Length: ' + Str1.Length.ToString);
end;
```

请注意，如果我们执行了标准版的程序，以及使用匿名方法的版本，就会从输出的结果发现当中明显的差异，匿名方法的版本可以看出有大约 10% 左右的延迟。

原因是没有能够直接执行区域源码，这个程序必须透过虚呼叫匿名方法的实作方式。由于这个差异持续存在，只要这两个测试程序在源码中没有交换数据，测试程序的意义就达到了。

然而如果我们要追求较高的执行效率，使用匿名方法并不会比直接写个一般函式的效率来的好。使用非虚拟方法指标的效能可能会介于这两种作法之间。

线程的同步

在多线程的应用程序中，如果需要更新用户接口，我们不可以直接去存取视觉组件的属性(或者内存内对象的属性)，因为它们都是全局线程的一部分，

并不具备线程同步的机制。Object Pascal 的视觉组件函式库事实上并不是”线程安全(Thread-Safe)”(但大多数的用户接口函式库则是)，也就是说，如果两个线程同时企图存取同一个对象，会使该对象的状态被混淆。

传统的解决方法是由 Object Pascal 的 TThread 类别所提供，是透过呼叫一个特别的方法: Synchronize，我们把指向另一个方法的参考当成参数传给这个特别的方法，这个被作为参数传入的方法就会被安全的执行。被传入的参数不能要求参数，所以实务上我们通常会在线程类别中加入一个额外的字段当成传递信息的媒介。在 Mastering Delphi 2005 这本书当中，我曾写过一个叫做 WebFind 的范例项目当成例子(这个程序会透过 HTTP 搜寻 Google 的内容，然后把搜寻结果里面的网址从 HTML 里面解析出来)，用以下的线程类别来使用:

```
type
    TFindWebThread = class(TThread)
    protected
        Addr, Text, Status: string;
        procedure Execute; override;
        procedure AddToList;
        procedure ShowStatus;
        procedure GrabHtml;
        procedure HtmlToList;
        procedure HttpWork (Sender: TObject;
            AWorkMode: TWorkMode; AWorkCount: Int64);
    public
        FStrUrl: string;
        FStrRead: string;
end;
```

保护区的三个字符串字段跟一些额外的方法已经被介绍成支持用户接口同步的功能。举例来说，HttpWork 事件处理程序会跟内部的 THTTPClient 对象(这是 Delphi 的客户端函式库组件中的一个)的 OnReceiveData 事件绑定，源码如下，它会呼叫 ShowStatus 方法:

```
procedure TFindWebThread.HttpWork(const Sender: TObject;
    AcontentLength, AReadCount: Int64; var AAbort: Boolean);
begin
    FStatus := 'Received ' + IntToStr (AReadCount) + ' for ' + FStrUrl;
    Synchronize (ShowStatus);
 end;
```

```
procedure TFindWebThread.ShowStatus;
begin
    Form1.StatusBar1.SimpleText := FStatus;
end;
```

Object Pascal 的 Synchronize 方法有两个不同的多载定义：

```
type
    TThreadMethod = procedure of object;
    TThreadProcedure = reference to procedure;
    TThread = class
      ...
      procedure Synchronize(
          AMethod: TThreadMethod); overload;
      procedure Synchronize(
          AThreadProc: TThreadProcedure); overload;
```

为了这个原因，我们可以移除 FStatus 文字字段以及 ShowStatus 方法，然后用新版的 Synchronize 来重写 HTTPWork 事件处理程序跟一个匿名方法:

```
procedure TFindWebThreadAnon.HttpWork(const Sender: TObject;
    AcontentLength, AReadCount: Int64; var AAbort: Boolean);
begin
    Synchronize (
        procedure
            begin
                Form1.StatusBar1.SimpleText :=
                    'Received ' + IntToStr (AReadCount) + ' for ' + FStrUrl;
            end);
end;
```

使用相同的目标来思考这个类别的源码，则线程类别变成以下所示(我们可以在本书范例源码的 WebFind 范例项目中找到这两种版本的线程类别):

```
type
    TFindWebThreadAnon = class(TThread) protected
        procedure Execute; override;
        procedure GrabHtml;
        procedure HtmlToList;
        procedure HttpWork (const Sender: TObject; AContentLength: Int64;
```

```
                AReadCount: Int64; var AAbort: Boolean);
  public
    FStrUrl: string;
    FStrRead: string;
  end;
```

使用匿名方法简化了线程同步作业所需的源码。

笔记 匿名方法跟线程有许多关连性，因为线程是用来执行某些源码的，而匿名方法则是用来显示源码的。这也是为何 TThread 类别会支持匿名方法，在平行程序函式库(在 TParallel.For 当中，以及定义一个 TTask 的时候)当中也支持。在本章里面介绍多线程之后，我不会在这方面再提供其他的范例了。但在下一个范例中，我会使用另一个线程，因为在使用到 HTTP 联机的时候，使用线程来运作是很常见的要求。

Object Pascal 里的 AJAX

这一节的前一个例子，AnonAjax 范例项目，是我最中意的匿名方法范例(虽然这个范例有点难度)。理由是我学到使用 JavaScript 里面的 closure(或者我们也可以叫他匿名方法)来透过 jQuery 函式库写 AJAX 应用程序。

笔记 AJAX 是 Asychronous Javascript XML 的简写，通常用来在浏览器里面呼叫 Web 服务。这个技术近年来越来越受欢迎，也被广泛使用，Web 服务也逐渐往 REST 架构跟 JSON 格式发展，因此 AJAX 这个名词也开始慢慢失去光环，REST 成为了新一代的浪头。我决定在范例中保留旧的名字，如果您对这个范例的要求有兴趣，也想了解背后的历史，请参阅：https://en.wikipedia.org/wiki/Ajax_(programming)

AjaxCall 全局函式会生出一个线程，然后把这个线程传给一个匿名方法，让该匿名方法可以完整执行。函式只是一个用来包装线程建构函式的外壳而已：

```
type
  TAjaxCallback = reference to procedure (
      ResponseContent: TStringStream);
procedure AjaxCall (const StrUrl: string; AjaxCallback: TAjaxCallback);
begin
   TAjaxThread.Create (StrUrl, AjaxCallback);
end;
```

这些源码都在 TAjaxThread 类别中，这个线程包含一个内部的 HTTP 客户端组件(Delphi 内建的客户端函式库)用来浏览指定的 URL，当然是以异步形式进行:

```
type
    TAjaxThread = class (TThread)
    private
        FHttp: THTTPClient;
        FURL: string;
        FAjaxCallback: TAjaxCallback;
    protected
        procedure Execute; override;
    public
        constructor Create (const StrUrl: string;
                AjaxCallback: TAjaxCallback);
        destructor Destroy; override;
end;
```

这个建构函式做了一些初始化的动作，复制它的参数到线程类别中对应的字段，并建立 FHttp 对象。这个线程类别实际执行的源码，都在 Execute 方法里面，它会进行 HTTP 要求，把结果储存在一个串流结构中，稍后会进行重置，并把它传递给 callback 函式-一个匿名方法:

```
procedure TAjaxThread.Execute;
var
  AResponseContent: TStringStream;
begin
    AResponseContent := TStringStream.Create;
    try
      FHttp.Get (FURL, aResponseContent);
      AResponseContent.Position := 0;
      FAjaxCallback (AResponseContent);
    finally
      AResponseContent.Free;
    end;
end;
```

我们再拿 AnonAjax 范例项目来当一个例子，它有一个按钮用来把网页内容复制到一个 Memo 组件上(会在最开头加上要求的网址):

```
procedure TFormAnonAjax.btnReadClick(Sender: TObject);
begin
```

```
    AjaxCall (edUrl.Text,
        procedure (aResponseContent: TStringStream)
        begin
            Memo1.Lines.Text := aResponseContent.DataString;
            Memo1.Lines.Insert (
                0, 'From URL: ' + edUrl.Text);
        end);
end;
```

等 HTTP 要求完成，我们就可以在上面做我们想要进行的任何处理了。举例来说，就是把 HTML 档案里的连结解析出来(这也算是把稍早的 WebFind 范例执行的结果进行重组)。重申一下，要让这个函式更有弹性，它可以要求一个匿名方法当做参数，这样每个连结就有独立的源码来处理对应的数据了：

```
type
    TLinkCallback = reference to procedure (const StrLink: string);

    procedure ExtractLinks (StrData: string; ProcLink: TLinkCallback);
    var
        StrAddr: string;
        NBegin, NEnd: Integer;
    begin
        StrData := LowerCase (StrData);
        NBegin := 1;
        repeat
            NBegin := PosEx ('href="http', StrData, NBegin);
            if NBegin <> 0 then
            begin
                // find the end of the HTTP reference
                NBegin := NBegin + 6;
                NEnd := PosEx ("", StrData, NBegin);
                StrAddr := Copy (StrData, NBegin, NEnd - NBegin);
                // move on
                NBegin := NEnd + 1;
                // execute anon method
                ProcLink (StrAddr)
            end;
        until NBegin = 0;
```

```
    end;
```

如果我们拿这个函式来处理 AJAX 呼叫后的结果，然后提供另一个方法作为进阶的处理，我们实质上就会有两个巢状的匿名函数调用，就像 AnonAjax 范例项目的第二个按钮:

```
procedure TFormAnonAjax.btnLinksClick(Sender: TObject);
begin
    AjaxCall (edUrl.Text,
        procedure (aResponseContent: TStringStream)
        begin
            ExtractLinks(aResponseContent.DataString,
                procedure (const aUrl: string)
                begin
                    Memo1.Lines.Add (aUrl + ' in ' + edUrl.Text);
                end);
        end);
end;
```

在这个例子里，Memo 组件会接收到一连串的连结，而不是回传网页的 HTML 文字。上面这段源码所解析出来的连结也可以改成完全针对图片的网址进行解析。ExtractImages 函式就会从 HTML 原始码里面抓出 img 卷标的 src 属性，然后呼叫另一个 TLinkCallback 兼容的匿名方法。现在，我们可以在背景开启一个网页(透过 AjaxCall 函式)，解析出当中的图片网址，然后再呼叫一次 AjaxCall 把图片抓回来。这表示使用一个三层的巢状匿名方法的源码，对一些 Object Pascal 开发人员来说可能会变得很难懂(这会需要一些时间来习惯)，但这真的功能很强大，也很令人印象深刻:

```
procedure TFormAnonAjax.BtnImagesClick(Sender: TObject);
var
    NHit: Integer;
begin
    NHit := 0;
    AjaxCall (EdUrl.Text,
        procedure (AResponseContent: TStringStream)
        begin
            ExtractImages(AResponseContent.DataString,
                procedure (const AUrl: string)
                begin
                    Inc (NHit);
                    Memo1.Lines.Add (IntToStr (NHit) + '.' +
```

```
                    AUrl + ' in ' + EdUrl.Text);
                if NHit = 1 then // load the first
                begin
                    AjaxCall (AUrl,
                        procedure (AResponseContent: TStringStream)
                        begin
                          // load image of the current type only
                          Image1.Picture.Graphic.
                              LoadFromStream(AResponseContent);
                        end);
                end;
            end);
        end);
end;
```

笔记 这个程序片段是我在 2008 年九月的部落格文章中，标题为”Anonymous, Anonymous, Anonymous”，当中也有一些网友的想法交流，读者可以上网看原文: http://blog.marcocantu.com/blog/anonymous_3.html。

除了只能在图片组件中以同样的格式加载档案时，才能让图片显示这一点之外，上面这段源码跟执行的结果都令人印象深刻。请留意到编号的顺序是以被保留的局部变量 NHit 为基础的。如果我们快速点击了该按钮两次会如何?每一次匿名方法都会取得一份 NHit 的数值，然后被用来显示在列表中，当第二个线程开始提供输出结果时，第一个线程的结果可能还没出现呢。尽管有潜在的问题，最后一个按钮的执行结果，看起来会像图 15.1:

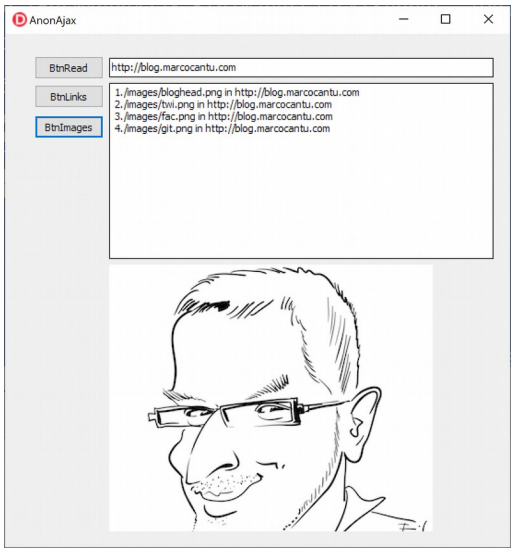


图 15.1: 透过三层巢式匿名方法下载网页内容的执行结果

16:镜射与标注

基本上，强型别的编译器，静态型别语言，例如 Pascal，在执行时期提供的可用型别信息很少，甚至没有。所有关于数据型别的信息只有在编译过程中可以被看的到。

Object Pascal 的第一版就打破了这个传统，它提供了属性与其他类别成员的执行时期信息，只需透过一个特别的区段宣告关键词:published。这个类别功能只要在编译器时有加上{$M+}这个编译设定即可，它同时也是 VCL 的 DFM 档案串流化机制的基础(FireMonkey 函式库的 FMX 档案也是使用此一机制)，在窗体与其他可视化组件的设计环境也是使用此一机制。

在 Delphi 第一版推出的时候，这功能算是整个程序设计工具业界最创新的想法，后来许多程序开发工具也都用了各种不同的方法提供这个功能，并把这个功能加以延伸。

首先，在型别系统中有延伸功能来为方法进行注记跟检测(此功能仅有 Object Pascal 语言提供)，以及在 COM 面动态进行呼叫。在 Object Pascal 里面这功能仍然以 dispatch ID 进行支持，在方法中使用变异变量(Variant)以及其他跟 COM 相关的功能。最后，在 Object Pascal 里面对 COM 的支持已经由编程语言内建的动态时期型别信息来外延支持了，但这已经超过本书作为介绍编程语言功能的书该涵盖的范围了。

后来系统管理环境的出现，像是 Java 跟.NET，提出了一个很有延伸形式的执行时期型别信息(Run Time Type Information，以后简称 RTTI)，透过由编译器把详细的 RTTI 跟执行模块进行绑定，程序中就可以对这些模块检测与使用了。这有让内部源码跟模块的大小增加的副作用，但它也透过新的程序模块，以及结合动态编程语言的一些弹性到固定结构、增加强型别速度的优点。

不管你喜欢与否(在该功能被加入的时候，的确引发了激烈的争论)，Object Pascal 已经慢慢朝向相同的方向前进，并且提供 RTTI 的延伸功能，使这个方向的发展越发明显了。正如我们看到的，我们可以选择不用 RTTI，但如果我们使用 RTTI，则可以在应用程序中整合更多的功能进去。

这个主题并不简单，所以我们会一步一步介绍。

- 首先我们会先介绍新的延伸 RTTI，这功能已经内建在编译器，我们可以透过使用 RTTI 单元文件来使用里面的新功能与类别。
- 其次，我们应该也要留意到新的 TValue 结构跟动态呼叫的技术。
- 第三，我会介绍自定标注，这个功能在.NET 上也有提供，并且可以让我们可以延伸由编译器产生的 RTTI 信息。

我们会在本章的最后部分来探讨在延伸的 RTTI 背后的理由，以及看一些使用 RTTI 使用的实际案例。

延伸的 RTTI

Object Pascal 编译器默认就会建立许多延伸的 RTTI 信息。执行时期信息包含所有型别、包含类别与所有其他使用者定义的型别，都会像是系统提供的核心数据型别一样提供相同层级的数据，并包含到私有区、保护区、公开区、发布区的所有元素。这需要深入到任何一个对象的所有内部结构。

第一个范例

在我们开始看到编译器产生的信息以及几种不同存取这些信息的技术之前，我们先直接跳去看结论，并看一下使用 RTTI 可以达到什么功能吧。具体的例子并不多，而且可能是以旧版的 RTTI 写成的，但这些例子可以让您理解我所要介绍的想法(也要考虑到并不是所有的 Object Pascal 开发人员都使用传统的 RTTI 来开发)。

假设我们有一个窗体上头放了一个按钮，就像 RttiIntro 范例项目。我们可以用以下这段源码来读取控制组件的 Text 属性:

```
uses Rtti;

procedure TFormRttiIntro.btnInfoClick(Sender: TObject);
var
    Context: TRttiContext;
begin
    Show (Context.
        GetType(TButton).
        GetProperty(Text).
        GetValue(Sender).ToString);
end;
```

这段源码使用了 TRttiContext 记录来参考 TButton 型别的信息，从这个型别的信息到 RTTI 关于属性的数据，以及这个属性的数据被用来参考到属性的内容值，它被转型成为了一个字符串。
如果您好奇这是怎么运作的，请继续读下去。在这里我的重点是，这种方法现在不只可以用来动态的存取属性，也可以读取数据字段的内容，包含私有区的字段。

我们还可以改变一个属性的值，就像 RttiIntro 范例项目中第二个按钮的动作一样：

```
procedure TFormRttiIntro.btnChangeClick(Sender: TObject);
var
    context: TRttiContext;
    AProp: TRttiProperty;
begin
    AProp := Context.GetType(TButton).GetProperty('Text');
    AProp.SetValue(btnChange, StringOfChar ( '*', random (10) + 1));
end;
```

这段源码以随机长度的＊符号来取代 Text 的内容。跟以上的源码的不同点是它用了暂存的局部变量来参考属性的 RTTI 信息。现在，您该对于我们要介绍的想法有些轮廓了，我们先从检查由编译器建立的延伸 RTTI 信息开始。

编译器产生的信息

我们不用特别作什么，编译器就会把这些额外信息加入到我们的执行档(不论它的形式为何: 应用程序、函式库、套件等)。就开启一个项目，然后编译它。编译器默认就会为所有数据字段(包含私有区的字段)以及公开区、发布区的所有方法与属性建立延伸 RTTI。您可能会觉得惊讶，RTTI 居然也涵盖到私有区，但这是由于动态处理的需求，像是二进制对象的串行化以及对在 heap 内存区的对象进行追踪。

我们可以用以下列表的设定值来控制延伸 RTTI 的建立:X 轴代表成员的种类，Y 轴代表不同访问权限区。以下的列表是系统默认值:

	数据字段	方法	属性

私有区	X		
保护区	X		
公开区	X	X	X
发布区	X	X	X

技术上来说，这四个访问权限设定是使用以下宣告在 System 单元文件的集合型别来进行对应的:

```
type
    TVisibilityClasses = set of (vcPrivate,
        vcProtected, vcPublic, vcPublished);
```

系统中提供了一些预先定义好的常数值，可以和预设的 RTTI 内容对应，这些访问权限的可视范围是跟 TObject 的可视设定关连，所有的类别也都继承了这些相同的权限：

```
const
    DefaultMethodRttiVisibility = [vcPublic, vcPublished];
    DefaultFieldRttiVisibility = [vcPrivate..vcPublished];
    DefaultPropertyRttiVisibility = [vcPublic, vcPublished];
```

编译器是否要建立这些信息，可以用一个新的编译器设定来处理：$RTTI，这当中包含了一个状态，用来指示特定型别或它的衍生类别(明确指示或是继承)，并以三个特定的关键词来标明方法、数据字段或者属性的可视状态。

在 System 单元文件中的默认值是:

```
{$RTTI INHERIT
    METHODS(DefaultMethodRttiVisibility)
    FIELDS(DefaultFieldRttiVisibility)
    PROPERTIES(DefaultPropertyRttiVisibility)}
```

要完整的关闭为所有类别的成员建立延伸 RTTI 的话，我们可以使用以下的设定:

```
{$RTTI EXPLICIT METHODS([]) FIELDS([]) PROPERTIES([])}
```

笔记

我们无法在单元文件的宣告之前设定 RTTI 的设定值，就跟其他编译器设定值一样，因为这些设定值的定义是在 System 单元文件里面宣告的。如果我们在单元文件最前面(uses 区段之前)就写好编译器设定值的话，就会出现内部错误，这看起来不太直觉。总之，就把它放在单元的指令之后吧。

使用这个设定的时候，要记得它只会被套用在我们自己写的源码，而且不可能完全移除，因为用于 RTL 跟其他函式库的类别的 RTTI 信息已经被编译到对应的 DCU 档案跟套件档案里面了。

同时也要记得，$RTTI 设定对于为已发布的型别所建立的传统 RTTI 的建立不会有任何影响：它还是会建立，跟是否设定$RTTI 无关。

笔记 RTTI 处理类别，是透过 RTTI 单元文件，我们稍后的章节会加以介绍，它是透过挂载在传统 RTTI 跟它的 PTypeInfo 结构来达成的。

我们可以透过这个设定值来停止为我们自定的类别建立延伸 RTTI。相对的，我们也可以增加 RTTI 要建立的范围，包含私有区、保护区的方法跟属性，只要有需求的话(虽然建立这些信息不一定都有实质用处)。

把延伸 RTTI 加到执行档，最明显的影响就是执行档的档案变大了(大的档案在进行散布的时候绝对算是一个缺点，因为额外的加载时间跟所使用的内存对于程序执行来说并不很相关)。如果我们决定不在源码里面使用 RTTI 的话，可以把 RTTI 从我们的源码当中的单元文件移除，重新编译后，就可以看出两者之间的差距了(如果我们决定不要使用 RTTI 这个技术的话)。RTTI 是个很强大的技术，从本章的介绍中就可以感觉的到，或者在大多数的状况下，它也值得多用一些额外的空间。

强型别与弱型别连结

我们还有什么方法可以让执行档的 Size 变小一点吗？有的，虽然影响不大，但绝对值得我们注意。

当我们可以从执行档案里面取得 RTTI 信息，要注意到编译器加进去的信息，有时候链接程序会这些信息给移除掉。预设情形下，没有被编译到执行档的类别跟方法就不会建立延伸 RTTI(因为在此一情形下，这些信息建了也没用)，因为也无法取得基础的 RTTI。

另一种情况下，如果我们想要让所有的延伸 RTTI 都被建立，并可以被使用，我们就需要把这些几乎没有用到的类别跟方法进行连结。

我们可以使用两个编译器设定值来控制与执行文件链接的信息。第一个是 $WeakLinkRTTI 设定值，它在文件里面有完整的叙述。设定这个值，在程

序里面没有被使用到的型别以及该型别所属的 RTTI 都会被从执行档当中移除。

或者我们也可以强制把所有型别，以及这些型别的延伸 RTTI 透过 $StrongLinkTypes 这个编译器设定值包进执行档里面。对很多程序来说，这个效果可是很戏剧化的，不过执行档的大小也大概会多出一倍来。

RTTI 单元文件

如果为所有型别建立延伸 RTTI 是 Object Pascal 镜射功能的第一根柱子，第二根柱子是能够用简单、高阶的方式来浏览这些型别的能力(多亏了 System.Rtti 单元文件)，第三根柱子我们待会会介绍，就是对于自定标注的支持，我们一个个来看吧。

传统的 Object Pascal 程序可以(现在还是可以喔)使用 TypeInfo 单元文件里面的函式来存取类别『发布区』的执行时期型别信息。这个单元文件定义了一些低阶的数据结构跟函式(都是以指针跟记录为基础)，并定义了一些高阶的子程序来简化程序的运作。

不同的是，Rtti 单元文件则让使用延伸 RTTI 变得很容易，当中提供了一整组的类别，包含了适当的方法跟属性。要存取不同的对象，进入点都是 TRttiContext 记录结构，当中有四个方法用来搜寻可用的型别:

```
function GetType (ATypeInfo: Pointer): TRttiType; overload;
function GetType (AClass: TClass): TRttiType; overload;
function GetTypes: TArray<TRttiType>;
function FindType (const AQualifiedName: string): TRttiType;
```

正如我们可以看到的，我们可以传递一个类别，以一个从型别所取得的 PTypeInfo 指标，一个完整的名称(包含从该类别被宣告的单元文件开始到型别的完整名称，像是"System.TObject")，或者取得型别的完整列表，被定义成一个 RTTI 型别的数组，或者更精确的说，是定义成 TArray<TRttiType>。

我在以下的列表当中也有使用到最后的函式，是 TypesList 范例项目的版本简化:

```
procedure TFormTypesList.btnTypesListClick(Sender: TObject);
var
    AContext: TRttiContext;
    TheTypes: TArray<TRttiType>;
```

```
    AType: TRttiType;
begin
     TheTypes := AContext.GetTypes;
     for AType in TheTypes do
         if AType.IsInstance then
            Show(AType.QualifiedName);
end;
```

GetTypes 方法回传了数据型别的完整列表，但程序会过滤掉内容，只留下型别所表示的相关类别而已。在这个单元文件里面大概有十几个类别所表示的型别。

笔记

RTTI 单元文件是把类别型别当成”实体”或者”实体型别”(就像在 TRttiInstanceType 里面)。这听起来有点容易混淆，就像我们平常把实际的对象称为实体一样。

在型别列表里面的独立对象是属于从 TRttiType 衍生而来的一些类别。我们可以特别来看一下 TRttiInstanceType 类别型别，把上面的源码重写成以下的程序片段:

```
for AType in TheTypes do
    if AType is TRttiInstanceType then
      Show(AType.QualifiedName);
```

这个范例的源码有些复杂，它会先建立一个字符串列表，把里面的元素进行排序，然后建立一个 ListView 组件，使用 BeginUpdate 跟 EndUpdate 来进行优化(并且会用一个 try finally 区块把这些源码包起来，确定最后的动作一定会被执行到)：

```
var
    AContext: TRttiContext;
    TheTypes: TArray<TRttiType>;
    SList: TStringList;
    AType: TRttiType;
    STypeName: string;
begin
    ListView1.ClearItems;
    SList := TStringList.Create;
    try
      TheTypes := AContext.GetTypes;
```

```
    for AType in TheTypes do
      if AType.IsInstance then
        SList.Add(AType.QualifiedName);
    SList.Sort;
    ListView1.BeginUpdate;
    try
      for STypeName in SList do
        (ListView1.Items.Add).Text := STypeName;
    finally
      ListView1.EndUpdate;
    end;
  finally
    SList.Free;
  end;
end;
```

这段源码会建立一个数以百计的数据型别列表，实际的数字要看编译器的版本跟其使用的平台了，我们可以看一下图 16.1。当中从 RTTI 单元文件列出了影像列表型别，我们会在下一节里面加以介绍。

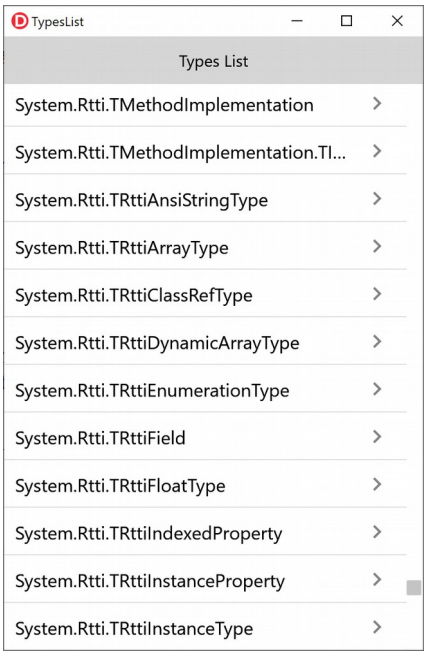


图 16.1: TypeList 范例项目的执行结果

在 Rtti 单元文件里面的 RTTI 类别

在以下的列表中，我们可以看到类别继承关系的完整图表，这些类别都是从 TRttiObject 衍生出来的，它们都定义在 Rtti 单元文件里面：

```
TRttiObject   //Abstract
        TRttiNamedObject
          TRttiType
            TRttiStructuredType   // Abstract
              TRttiRecordType
              TRttiInstanceType
              TRttiInterfaceType
            TRttiOrdinalType
              TRttiEnumerationType
            TRttiInt64Type
            TRttiInvokableType
              TRttiMethodType
              TRttiProcedureType
            TRttiClassRefType
            TRttiEnumerationType
            TRttiSetType
            TRttiStringType
              TRttiAnsiStringType
            TRttiFloatType
            TRttiArrayType
            TRttiDynamicArrayType
            TRttiPointerType
          TRttiMember
            TRttiField
          TRttiProperty
                TRttiInstanceProperty
              TRttiIndexedProperty
              TRttiMethod
            TRttiParameter
            TRttiPackage
          TRttiManagedField
```

这些类别当中的每一个都提供了该型别的专属信息。例如，只有 TRttiInterfaceType 提供了可以存取 GUID 的接口。

笔记 在 Rtti 单元文件的第一次实作中，并没有 RTTI 对象可以存取被索引过的属性(像 TStringList 类别的 Strings[]那样)。这是后来的版本才加上去的，现在仍旧可以使用，也让运行时间型别信息得以真正的完整。

RTTI 对象的生命周期管理以及 TRttiContext 记录

如果看过了刚刚列出的 BtnTypesListClick 方法的原始码，里头有些地方看起来应该是相当严重的错误。GetTypes 呼叫的函式回传了一个型别的数组，但那段程序并没有把内部的对象是放掉。原因是 TRttiContext 记录结构变成了所有被建立出来的 RTTI 对象的拥有者。当这个记录被释放(亦即程序执行脱离了该函式)，一个内部接口会明确的呼叫它自己的解构函式来是放掉被建立出来的所有 RTTI 对象。

TRttiContext 记录事实上扮演了两个角色。一方面它控制了 RTTI 对象的生命周期(我们刚刚介绍过)，另一方面它也对 RTTI 信息进行快取，因为每次对 RTTI 信息进行搜寻时所需要的重建程序是很昂贵的。这也是为什么我们会想要把 TRttiContext 记录的参考的生命周期做一个延伸，好让我们在存取 RTTI 信息的时候可以不用重新建立它们(重申一下，这些步骤是很花时间的)。

内部作业里，TRttiContext 记录使用了 TRttiPool 型别的一个全局缓冲区，它使用了临界区(Critical section, 这个名词我很习惯直接用英文，所以用这个翻译字眼不知道恰不恰当)来保证多线程的作业安全。

笔记 在 RTTI 的缓冲机制里面也会发生多线程同时存取的意外，相关的信息就直接被写在 Rtti 单元文件里面的批注文字。

所以，精确的说，RTTI 缓冲区是让所有的 TRttiContext 记录所共享的，因此被保留着的 RTTI 对象会一直被维持着，直到所有内存里面的 TRttiContext 记录都被释放掉为止。让我引用一下该单元文件里面的批注:

```
{... working with RTTI objects without at least one context being alive is an error. Keeping at least one
context alive should keep the Pool variable valid}
{译文: 想要不透过至少一个 context 来让 RTTI 运作，是会出错的。保持最少一个 context 存在可以让 Pool
变量能够被正常使用}
```

总而言之，我们必须避免在释放了 RTTI 内容之后还快取住，或是继续使用 RTTI 对象。以下是会导致内存违规存取的简单范例源码(它也是节录自 TypesList 范例项目):

```
function GetThisType (AClass: TClass): TRttiType;
```

```
var
    AContext: TRttiContext;
begin
     Result := AContext.GetType(aClass);
end;
procedure TFormTypesList.Button1Click(Sender: TObject);
var
    AType: TRttiType;
begin
     AType := GetThisType (TForm);
     Show (AType.QualifiedName);
end;
```

简单的做个整理，RTTI 对象是由内容所管理的，我们没办法让它随时保持可以使用的状态。这个内容是一个记录，所以会自动被编译器释放掉。我们可以看到一些源码是像这样来使用 TRttiContext 的:

```
AContext := TRttiContext.Create;
try
    // use the context
finally
  AContext.Free;
end;
```

虚构的建构函式跟虚构的解构函式会设定内部的接口，这些内部接口则会管理实际在背景使用到的数据结构，并把它设定成 nil 来清除这些缓冲区。然而，这个动作会对于区域型别，像是记录自动执行，这个动作不用特别处理，除非我们用了指标来参照这些记录。

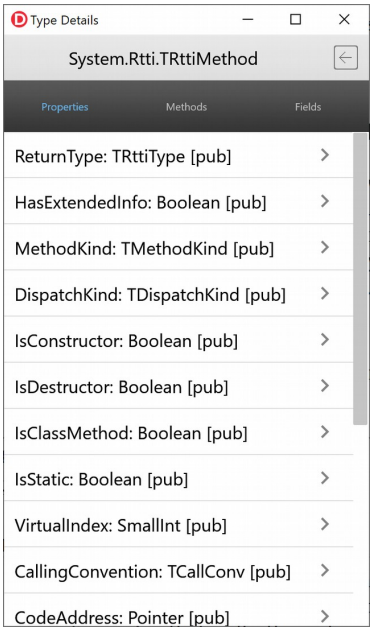
显示类别信息

我们在运行时间中最想要进行检查的相关型别，当然是结构化的型别，例如接口或记录。聚焦在这些实例上，我们可以参照这些类别之间的关系，透过以下可以用在实例型别上的 BaseType 信息。

存取型别当然是一个很有趣的开始，但相对的也是具备认识这些型别的内容的能力中相对比较新的，包含它们的成员。当我们点击这些型别中的任何一个(我们在范例中使用了 TPopup 组件类别)，程序就会显示一连串的属

性、方法，以及该型别的数据字段，我们透过三个分页加以呈现，请见图16.2。

图 16.2: 在 TypesList 范例项目中所显示的详细型别信息



第二个窗体的单元文件，可以用以延伸成为一个用在其他应用程序中通用的的型别浏览器，当中有一个名为 ShowTypeInformation 的方法，可以浏览特定型别当中的每一个属性、方法，以及数据字段，浏览的方式则是把它们加入到三个独立的列表，各自以不同的可视权限作为每个列表显示的内容(pri 会显示私有区，pro 会显示保护区，pub 则会显示公开区，pbl 则是显示发布区，这个显示的判别是写在 VisibilityToken 函式里面):

```
procedure TFormTypeInfo.ShowTypeDetails(Typename: string);
var
    AContext: TRttiContext;
    AType: TRttiType;
    AProperty: TRttiProperty;
    AMethod: TRttiMethod;
    AField: TRttiField;
begin
    AType := aContext.FindType(typename);
    if not Assigned(AType) then
       Exit;

    LabelType.Text := AType.QualifiedName;
    for AProperty in Atype.GetProperties do
        FormTypeInfo.LVProperties.Items.Add.Text := AProperty.Name +
```

```
            ': ' + AProperty.PropertyType.Name + ' ' +
            VisibilityToken (AProperty.Visibility);
    for AMethod in Atype.GetMethods do
        LVMethods.Items.Add.Text := AMethod.Name + ' ' +
            VisibilityToken (AMethod.Visibility);
    for AField in AType.GetFields do
        LVFields.Items.Add.Text := AField.Name + ': ' +
            AField.FieldType.Name + ' ' +
            VisibilityToken (AField.Visibility);
end;
```

我们可以继续解析这些属性所属的型别，来找出更深入的信息，取得该方法的参数列表，并检测回传型别等等。这个范例并不是要建立出完整的 RTTI 浏览器，只是要让我们知道，我们可以透过 RTTI 的功能获得哪些信息。

套件中的 RTTI

除了我们可以用来存取某个型别或者一系列型别的方法，TRttiContext 记录也提供了另外一个很有趣的方法”GetPackages”，这个方法可以回传目前应用程序有使用到的运行时间套件列表。如果我们在一个没有使用任何运行时间套件应用程序里面呼叫这个方法，回传值的内容就只会有目前的执行档而已。但如果我们在一个使用了许多运行时间套件的应用程序里面呼叫它，回传值的内容就会是这些套件的清单。从这个信息，我们就可以深入到每个套件里可以使用的型别。请留意到这个案例中，型别的清单可能会非常冗长，因为 RTL 跟视觉组件函式库当中没有被使用到的型别并不会被链接程序自动排除。

如果我们使用了运行时间套件，我们也可以取得每个套件当中所有型别的清单(也包含执行文件自己)，我们可以使用以下的源码:

```
var
    AContext: TRttiContext;
    APackage: TRttiPackage;
    AType: TRttiType;
begin
    for APackage in AContext.GetPackages do
    begin
        ListBox1.Items.Add('PACKAGE ' + APackage.Name);
```

```
    for AType in APackage.GetTypes do
      if AType.IsInstance then
      begin
        ListBox1.Items.Add('    - ' + AType.QualifiedName);
      end;
    end;
```

笔记 Object Pascal 的套件可以用来为开发环境加入视觉组件，就像我们在 11 章里面介绍过的。然而，套件也可以只在运行时间使用，把主要的执行档跟几个运行时间套件一起散发，不用把所有的源码组合成一个单一的执行档案。如果您对 Windows 开发还算熟悉，可以把套件想象成是 DLL 档案的角色(这些套件在技术上的确也是 DLL 档案)，或者我们也可以把它想成是.NET 的二进制文件。不过套件在 Windows 上面扮演了重要的角色，但在行动平台上却是不支持的(也是跟操作系统对应用程序的散发限制有关，例如 iOS 就不允许应用程序进行动态链接)。

TValue 结构

新的延伸 RTTI 不只让我们可以对程序内部结构进行浏览，还提供了特定的信息，包含属性和数据字段值。在 TypInfo 单元文件提供了名为 GetPropValue 的函式可以存取一般的属性，并以当中的值存取其对应的变异型别，新的 Rtti 单元文件则使用了不同的结构来处理这一类没有特别定义型别的元素，这个结构称为 TValue 记录。

这个记录几乎可以储存 Object Pascal 里面所有的数据型别，并且可以追踪它的原始数据表示方式，它的作法则是同时记录数据与其数据型别。因此我们就可以对该变量指定的型别进行读写。如果我们把整数写进 TValue，我们读出的就只能是整数型别了，如果写进去的是字符串的话，读出来的也就会是字符串。(在 XE6 之后，FireMonkey 里面的 TGrid 就已经使用 TValue 来设定每个 Cell 的内容，所以不像以往有分成 TStringGrid 等不同型别，TGrid 就涵盖了所有型别的 Grid)。

但 TValue 不能提供不同型别之间的格式转换，因此就算 TValue 有提供 AsString 与 AsInteger 方法，我们也只能在 TValue 的内容的确是字符串时使用 AsString，而在 TValue 内容的确是整数的时候才能使用 AsInteger。举例来说，在以下的案例里，我们可以使用 AsInteger 方法，如果我们呼叫 IsOrdinal 方法的话，回传值就会是 True:

```
var
```

```
    V1: TValue;
begin
    V1 := 100;
    if V1.IsOrdinal then
      Log (IntToStr (V1.AsInteger));
```

然而，我们不能使用 AsString 方法，强制使用的话，会导致一个 invalid typecast 例外:

```
var
    V1: TValue;
begin
     V1 := 100;
     Log (V1.AsString);
```

不过，如果我们需要使用字符串来表示 TValue 的内容，则可以用 ToString 方法，这个方法里面有一个 case 判别逻辑，可以把大多数数据型别的内容转换成字符串：

```
var
    V1: TValue;
begin
    V1 := 100;
    Log (V1.ToString);
```

我们可以阅读以下 Barry Kelly 所写的这段文字，增进对 RTTI 的了解，Barry 是前 Embarcadero 研发人员，他的工作主要就是 RTTI：

> *TValue 是个型别，可以用来存取以 RTTI 为基础的方法，并且读写相关的数据字段与属性。*
>
> *这个型别跟变异型别(Variant)类似，但其工作原理更偏向 Object Pascal 的型别系统。举例来说，实体可以被直接储存，像是集合(set)、类别参考等等。它也是更具局限性的型别，并且不会(例如)在背景进行字符串对数字的转换。*

现在我们对 TValue 的角色应该有更深一层的理解了，让我们来看一下 TValue 记录的实际能耐吧。它具备一组高阶的方法，可以指派、解析出实际的内容，也包含一组低阶的以指标为基础的方法。我们接下来会集中篇幅在高阶方法上。

在指派内容时，TValue 定义了许多个 Implict 运算方法，让我们可以直接在源码里面把特定的值指派给 TValue 型别的变数：

```
class operator Implicit(const Value: string): TValue;
class operator Implicit(Value: Integer): TValue;
class operator Implicit(Value: Extended): TValue;
class operator Implicit(Value: Int64): TValue;
class operator Implicit(Value: TObject): TValue;
class operator Implicit(Value: TClass): TValue;
class operator Implicit(Value: Boolean): TValue;
```

这些运算方法所做的，就只是呼叫泛型类别方法 From 而已:

```
class function From<T>(const Value: T): TValue; static;
```

当我们呼叫这些类别函式的时候，我们需要确定数据型别，并传递一个该型别的数据过去，像是以下的源码，会透过上述的这些类别方法，把整数值 100 进行指派:

```
V1 := TValue.From<Integer>(100);
```

这是一系列的统一技术，用来把所有型别的数据搬移到 TValue 型别的变数里去。一旦资料被指派完成，我们就可以用以下的几个方法来检测该数据的实际型别了:

```
property Kind: TTypeKind read GetTypeKind;
function IsObject: Boolean;
function IsClass: Boolean;
function IsOrdinal: Boolean;
function IsType<T>: Boolean; overload;
function IsArray: Boolean;
```

请注意这里面的 IsType 泛型方法，几乎可以用在任何数据型别上。
对于解析出来的数据，也有对应的方法，但我们只能对 TValue 里面储存的数据使用型别兼容的方法，因为当中并不会有任何转换的动作发生：

```
function AsObject: TObject;
function AsClass: TClass;
function AsOrdinal: Int64;
function AsType<T>: T;
function AsInteger: Integer;
function AsBoolean: Boolean;
function AsExtended: Extended;
function AsInt64: Int64;
```

```
function AsInterface: IInterface;
function AsString: string;
function AsVariant: Variant;
function AsCurrency: Currency;
```

这些方法大多会同时有 Try 的版本，会在执行时，如果发现型别不兼容时回传 False，而不会触发例外事件。当中也有一些限制转换的方法，最相关的是泛型函式 Cast 与 ToString 函式，我已经在范例中使用过它们了:

```
function Cast<T>: TValue; overload;
function ToString: string;
```

以 TValue 读取一个属性

TValue 的重要性，来自于这个结构会在存取属性或数据字段时，使用延伸的 RTTI 与 Rtti 单元文件。作为使用 TValue 的实际范例，我们可以用这个记录型别来存取 TButton 对象发布区的属性与私有区的数据字段，请参考以下的代码段(节录自 RttiAccess 范例项目):

```
var
    Context: TRttiContext;
    AType: TRttiType;
    AProperty: TRttiProperty;
    AValue: TValue;
    AField: TRttiField;
begin
    AType := Context.GetType(TButton);
    AProperty := AType.GetProperty('Text');
    AValue := AProperty.GetValue(Sender);
    Show (AValue.AsString);

    AField := AType.GetField('FDesignInfo');
    AValue := AField.GetValue(Sender);
    Show (AValue.AsInteger.ToString);
end;
```

呼叫方法

新的延伸 RTTI 并不只让我们存取值与数据字段，它也让我们对于呼叫方法有了更简化的方式。在这个例子中，我们必须为该方法的每个参数都定义

一个 TValue 元素。我们在呼叫这样的方法时，有个全局的函式叫做 Invoke 的可以使用:

```
function Invoke(CodeAddress: Pointer; const Args: TArray<TValue>;
    CallingConvention: TCallConv; AResultType: PTypeInfo): TValue;
```

作为更好的替代方案，在 TRttiMethod 类别中就定义了一个简化版的 Invoke 多载版本：

```
function Invoke(Instance: TObject;  const Args: array of TValue): TValue; overload;
```

我们在以下的范例中，提供了两个方式来呼叫使用简化版的方法(一个回传一个值，第二个则要求一个参数)，这些源码都是从 RttiAccess 范例项目里面节录出来的:

```
var
    Context: TRttiContext;
    AType: TRttiType;
    AMethod: TRttiMethod;
    TheValues: array of TValue;
    AValue: TValue;
begin
    AType := Context.GetType(TButton);
    AMethod := AType.GetMethod('ToString');
    TheValues := [];
    AValue := AMethod.Invoke(Sender, TheValues);
    Show(AValue.AsString);

    AType := Context.GetType(TForm1);
    AMethod := AType.GetMethod('Show');
    SetLength (TheValues, 1);
    TheValues[0] := AValue;
    AMethod.Invoke(self, TheValues);
end;
```

使用标注 (Using Attributes)

本章的第一个部分，是让我们能够对 Object Pascal 编译器建立的延伸 RTTI 能够有比较多的掌握，并透过介绍新的 Rtti 单元文件，能多认识 Rtti 的存取能力。第二部分，我们终于要来介绍整个架构的关键之一:定义自定标注，

以及透过特定方法来延伸编译器所建立的 RTTI 的可能性。我们接下来会从比较抽象的观点来看这个技术，然后说明这个技术对 Object Pascal 的重要性与其原因，而且我们会透过实例来看。

标注是什么？

标注，用 Object Pascal 或 C#的术语来说叫标注，用 Java 的习惯来看叫批注，是我们可以在源代码里面加入的说明或指示，我们可以把它加入到型别、数据字段、方法、或者属性(Property)，编译器会把它跟产出的执行二进制源码包在一起。这是通常用方括号进行的指示:

```
type
  [MyAttribute]
  TMyClass = class
    ...
```

在开发环境中以工具读取这些信息，或者在最后的应用程序从运行时间中读取这些信息，程序可以依照找到的信息来变更其规则。

通常标注不是用来改变类别或对象的实际核心功能，而是让这些类别能够在参与运作时能够有延伸特定功能的机制。把一个类别宣告为可串行化(serializable)，并不会影响到类别的任何功能，但会让串行化功能的源码得知这个类别需要被串行化处理，以及该如何处理(在这个例子里面，我们需要透过标注来提供更多信息，或者在类别的数据字段或属性中建立更进一步的标注)。

这完全就是受限版的 RTTI 一开始被 Object Pascal 内部使用的方式。被标示为发布区的属性(Property)，可以被显示在开发工具的对象查看器当中，也可以被串行化写成一个 DFM 档案，同时也可以在运行时间被存取。标注(Attribute)开启了这个机制，使它更有弹性、也更为强大。目前使用上很复杂，也很容易就误用，就像其他目前也很强大的编程语言功能一样。这个功能的存在，并不意味着要放弃原本我们已经理解的面向对象程序设计的优点，而是期望能从这些新的想法中跟原来的技术达到互补的效果。

举例来说，『员工』这个类别仍旧是从原有的架构上，一定还是从个人的类别中所衍生出来的，『员工』这个对象还是会对该对象需要提供一个标识符的字段，但我们可以”标注”这个『员工』类别需要跟数据库的特定数据表进行对应，或者跟特定的运行时间窗体进行对应。所以我们可以透过

继承(是特定类别)、拥有关系(属于)，以及标注(标示为)作为三种独立的机制，我们可以在设计应用程序的时候，这三种机制都可以是需求来用上。

当我们看过 Object Pascal 里面由编译器功能支持的自定标注，并看了几个案例以后，我刚介绍过的抽象想法应该就会变得比较容易懂了，至少这是我的希望。

标注类别与标注宣告

我们要如何定义一个新的标注类别(或者标注分类)呢?我们必须从 System 单元文件里面的一个新类别: TCustomAttribute 类别来衍生出新的类别：

```
type
   SimpleAttribute = class(TCustomAttribute)
   end;
```

我们为标注类别取的名字会变成在源码里面使用的标识符，我们可以选择把标注加在后面作为延伸。所以如果我们为类别命名为 SimpleAttribute，我们可以在源码里面用 Simple 或者 SimpleAttribute 来使用这个类别。因此 Object Pascal 里面不成文的规定，为每个类别都以 T 开头来命名的这个规定，通常在使用标注的时候，就不使用了，主要的例外会是 System.TCustomAttriute。

```
type
    [Simple]
    TMyClass = class(TObject)
    public
        [Simple]
        procedure One;
```

在上例中，我使用了 Simple 这个标注加在整个类别跟一个方法上面。除了名字，标注也支持一个或一个以上的参数。传递给标注的参数必须跟该标注类别的建构函数声明相同，如果该标注类别有建构函式的话:

```
type
  ValueAttribute = class(TCustomAttribute)
  private
      FValue: Integer;
  public
      constructor Create(N: Integer);
      property Value: Integer read FValue;
```

```
  end;
```

以下是把标注应用在一个参数的做法:

```
type
    [Value(22)]
    TMyClass = class(TObject)
    public
        [Value(0)]
        procedure Two;
```

要被传递给该类别的标注的值，必须是一个常数或表达式，因为常数或表达式的内容在编译阶段就可以被解译出来了。这也是只有少数的数据型别:有序内容、字符串、集合，以及类别参考可以使用的原因。从正面来看，我们可以用不同的参数来建立多种覆写版本的建构函式。注意一下，我们可以把许多个标注只在同一个标识符上面，就像在 RttiAtrrib 范例项目里面所做的，我们整理一下在这一节里面的代码段:

```
type
    [Simple][Value(22)]
    TMyClass = class(TObject)
    public
        [Simple]
        procedure One;
        [Value(0)]
        procedure Two;
    end;
```

要是我们试着使用一个还没有定义的标注呢(可能是因为在 uses 区段使用了别的单元文件所致)，我们就会得到下面这个错误讯息了:

```
[DCC Warning] RttiAttribMainForm.pas(44): W1025
    Unsupported language feature: 'custom attribute'
```

事实上这个讯息只是说该标注会被忽略掉，所以我们得要留意到类似的警告讯息里面，就像我们在其他情况下处理警告讯息一样，或者要更注意到是不是有”Unsupported language feature”这样的警告讯息出现，因为这个讯息已经跟错误没有两样了(我们可以在项目设定的画面中，Hint and Warning 设定页来调整要把这个情形看待成错误或是警告讯息):

```
[DCC Error] RttiAttribMainForm.pas(38):
    E1025 Unsupported language feature: 'custom attribute'
```

最后，跟其他类似的概念的实作方式做个比较，在标注的使用上目前机会没有方法可以限制使用的范围，像是在宣告区段，标注可以用在型别上，但不能用在方法上。然而在编辑器上面，标注在使用更名重购(rename refactoring)的功能上几乎是完全可以使用，没有任何限制的。我们不只可以更改标注类别的名字，系统还会自动在该标注被使用在完整的名字，以及没有使用最终完整标注的地方帮我们补正。

笔记 标注的重购，最初是由 Malcolm Groves 在他的部落格上所提出的: http://www.malcolmgroves.com/blog/?p=554。

浏览标注

现在如果没有方法可以找到哪些标注是有被定义过的，这些源码看起来就完全没有用了，而且可能因为这些标注，在对象当中注入不同的规则或行为。我们先来介绍第一部分吧。在 Rtti 单元中的类别，让我们可以厘清哪些标识符有相关的标注。

以下的源码是从 RttiAttrib 范例项目中节录出来的，可以显示出跟目前的类别有关连的标注清单：

```
procedure TMyClass.One;
var
    Context: TRttiContext;
    Attributes: TArray<TCustomAttribute>;
    Attrib: TCustomAttribute;
begin
    Attributes := context.GetType(ClassType).GetAttributes;
    for Attrib in Attributes do
        Form39.Log(Attrib.ClassName);
```

执行这个程序，输出结果如下：

```
SimpleAttribute
ValueAttribute
```

我们可以在源码的 for-in 循环里面做些修改，来解析出特定的标注型别：

```
if attrib is ValueAttribute then
  Form39.Show (' -' + IntToStr(ValueAttribute(attrib).Value));
```

要怎么得到透过特定的标注取得方法，或任何标注来取得?我们无法直接过滤方法，但可以一一取得这些方法，检查其标注，然后看看是不是我们要找的方法。为了协助这个作业，我已经写了这样的一个函式来检查每个方法是不是支持特定的标注：

```
type
    TCustomAttributeClass = class of TCustomAttribute;


function HasAttribute (AMethod: TRttiMethod;
     AttribClass: TCustomAttributeClass): Boolean;
var
    Attributes: TArray<TCustomAttribute>;
    Attrib: TCustomAttribute;
begin
    Result := False;
    Attributes := aMethod.GetAttributes;
    for Attrib in attributes do
        if attrib.InheritsFrom (attribClass) then
          Exit (True);
end;
```

HasAttribute 这个函式会被 RttiAttrib 程序用来检查特定的标注:

```
var
    Context: TRttiContext;
    AType: TRttiType;
    AMethod: TRttiMethod;
begin
    AType := Context.GetType(TMyClass);
    for AMethod in AType.GetMethods do
      if HasAttribute (AMethod, SimpleAttribute) then
        Show (AMethod.name);


    for AMethod in AType.GetMethods do
      if HasAttribute (AMethod, TCustomAttribute) then
        Show (AMethod.name);
```

这效果是列出特定标注的方法：

```
Methods marked with [Simple] attribute
One
```

```
Methods marked with any attribute
One
Two
```

我们一般所做的并不是简单的描述标注，而是对特定的标注类别加上一些独立的行为，跟它原本的源码不同。举个实例，我们可以在上面的源码里面注入特定行为: 目标是呼叫有特定标注类别的所有方法，先考虑它们都是不需要参数的方法:

```
procedure TForm39.BtnInvokeIfZeroClick(Sender: TObject);
var
    Context: TRttiContext;
    AType: TRttiType;
    AMethod: TRttiMethod;
    ATarget: TMyClass;
    ZeroParams: array of TValue;
begin
    ATarget := TMyClass.Create;
    try
      AType := Context.GetType(ATarget.ClassType);
      for AMethod in AType.GetMethods do
        if HasAttribute (AMethod, SimpleAttribute) then
          AMethod.Invoke(ATarget, zeroParams);
    finally
      ATarget.Free;
    end;
end;
```

这个程序片段所做的动作包含了建立一个对象、抓出该对象的型别，检测特定的标注，最后呼叫每个有标注为 Simple 的方法。这跟从类别继承、实作接口、或者写一些特定的源码来行使这个要求都不同，要取得新的动作，我们只需要以特定的标注来为一个或多个方法进行加注。这并不式说这个例子使得标注的使用异常明显: 我们可以在本章的最后参照一些常见的使用标注的模式，和一些实际的个案研究。

虚拟方法拦截器

这一节介绍的是 Object Pascal 当中非常高阶的功能，如果您目前才刚开始学习 Delphi 语言，您可能会想要先跳过它。这一节是为了对 Delphi 已经很专精的读者而写的。

有另一个相关的功能在延伸 RTTI 被加入 Object Pascal 之后也被加入了，这个功能是对一个现存的类别可以透过虚拟方法来加以拦截的能力，只要对现存的对象建立一个代理类别。换句话说，我们可以选一个已经存在的对象，然后修改它的虚拟方法(可以一次只修改其中一个，或一次把全部都处理掉)。

我们要怎么完成这一点?在标准的 Object Pascal 应用程序中，我们应该没办法使用这个功能。如果我们需要让一个对象拥有不同的行为，就只能先衍生一个子类别，然后在子类别里面修改源码来达成。而在函式库里面则是完全不同的，因为函式库必须用很通用的方式来撰写，对对象的了解不多，函式库可以操作，并且尽可能的降低对对象的负担。这就是在 Object Pascal 里面加入虚拟方法拦截器的情境。

笔记 我们可以从一则部落格的发文看到对于虚拟方法拦截器(这部份我是比较欠缺的): http://blog.barrkel.com/2010/09/virtual-method-interception.html。

在我们聚焦在可能的情境之前，我们先讨论技术本身吧。假设我们有一个已经存在的类别，它带有至少一个虚拟方法，像下面的例子这样:

```
type
    TPerson = class
    ...
    public
        property Name: string read FName write SetName;
        property BirthDate: TDate read FBirthDate write SetBirthDate;
        function Age: Integer; virtual;  function ToString: string; override;
    end;


function TPerson.Age: Integer;
begin
    Result := YearsBetween (Date, FBirthDate);
end;
```

```
function TPerson.ToString: string;
begin
    Result := FName + ' is ' + IntToStr (Age) + ' years old';
end;
```

假设我们在这个类别里有一个叫 FPerson1 的对象。现在，我们可以做的就是建立一个 TVirtualMethodInterceptor 对象(它是定义在 Rtti 单元文件里面的一个新类别)把它跟我们想要制作的新类别(TPerson)绑定起来，成为一个子类别的对象，把 FPerson1 对象的固定类别变成动态类别:

```
var
    FVmi: TVirtualMethodInterceptor;
begin
    FVmi := TVirtualMethodInterceptor.Create(TPerson);
    FVmi.Proxify(FPerson1);
```

一旦我们有了 FVmi 对象，我们就可以用匿名方法，来为它安装处理程序到特定的事件上(onBefore, onAfter, 以及 onException)。这些匿名方法会在任何虚拟方法被呼叫前、被呼叫之后，以及万一在任何虚拟方法触发了意外的时候被驱动。以下是这三个匿名方法的特征：

```
TInterceptBeforeNotify = reference to procedure( Instance: TObject; Method: TRttiMethod;
   const Args: TArray<TValue>; out DoInvoke: Boolean; out Result: TValue);
TInterceptAfterNotify = reference to procedure( Instance: TObject; Method: TRttiMethod;
   const Args: TArray<TValue>; var Result: TValue);
TInterceptExceptionNotify = reference to procedure( Instance: TObject; Method: TRttiMethod;
    const Args: TArray<TValue>; out RaiseException: Boolean; TheException: Exception;
    out Result: TValue);
```

在每个事件中，我们可以取得该对象，该方法的参考，参数，以及回传值(有时也可能没有回传值)。在 OnBefore 事件中，我们可以设定 DoInvoke 参数让标准的执行动作失效。在 OnExcept 事件中，我们可以取得意外事件的详细信息。

在 InterceptBaseClass 范例中，范例中使用了上面的 TPerson 类别，我拦截了类别的虚拟方法，加入了以下的纪录源码:

```
procedure TFormIntercept.BtnInterceptClick(Sender: TObject);
begin
    FVmi := TVirtualMethodInterceptor.Create(TPerson);
    FVmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
```

```
        const Args: TArray<TValue>; out DoInvoke: Boolean;
        out Result: TValue)
        begin
            Show('Before calling ' + Method.Name);
        end;
    FVmi.OnAfter := procedure(Instance: TObject; Method: TRttiMethod;
        const Args: TArray<TValue>; var Result: TValue)
        begin
            Show('After calling ' + Method.Name);
        end;
    FVmi.Proxify(FPerson1);
end;
```

注意到，FVmi 对象需要被维持到 FPerson1 对象被使用结束为止，不然我们就会使用一个动态类别，而该类别已经不能用了，最后我们就会呼叫到一个已经被释放掉的匿名方法。在范例中，我把它存放在一个窗体的数据字段中，就跟对象会参考到的数据(FPerson1)一样。

这个程序会透过呼叫它的方法以及检测基础类别的名称来使用对象:

```
Show ('Age: ' + IntToStr (FPerson1.Age));
Show ('Person: ' + FPerson1.ToString);
Show ('Class: ' + FPerson1.ClassName);
Show ('Base Class: ' + FPerson1.ClassParent.ClassName);
```

在我们安装拦截器之前，输出结果是:

```
Age: 26
Person: Mark is 26 years old
Class: TPerson
Base Class: TObject
```

安装了拦截器之后，输出结果变成了:

```
Before calling Age
After calling Age
Age: 26
Before calling ToString
Before calling Age
After calling Age
After calling ToString
Person: Mark is 26 years old
```

```
Class: TPerson
Base Class: TPerson
```

请注意这个类别跟基础类别的名称完全一样，但事实上却是完全不同的两个类别，它是用虚拟方法拦截器所制作的动态类别。

纵使并没有官方的作法可以把目标对象的类别恢复成原始的类别，该类别本身仍旧可以透过虚拟方法拦截器对象运作，也仍旧是该对象的基础类别。当然我们也可以硬把正确的类别参考硬塞给该对象(初始时是四个 Bytes)的类别数据:

```
PPointer(Person1)^ := FVmi.OriginalClass;
```

举个更难的例子，我们已经修改了 OnBefore 的源码，所以在这个案例中，如果我们呼叫了 Age，它会回传特定的值，而把原来类别源码中的动作给忽略掉:

```
FVmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
        const Args: TArray<TValue>; out DoInvoke: Boolean;
        out Result: TValue)
        begin
            Show ('Before calling ' + Method.Name);
            if Method.Name = 'Age' then
            begin
                Result := 33;
                DoInvoke := False;
            end;
         end;
```

输出结果跟原始版本的结果不同(请记得 Age 的呼叫跟相关的 OnAfter 事件都被忽略掉了):

```
Before calling Age
Age: 33
Before calling ToString
Before calling Age
After calling ToString
Person: Mark is 33 years old
Class: TPerson
Base Class: TPerson
```

现在我们已经看过了虚拟方法拦截器的技术细节，我们可以继续厘清在什么情境下我们会想用这个功能了。

再次重申，基本上在标准的应用程序中，我们没有理由使用这个功能。但在大多数开发进阶的函式库，且需要为了测试或处理对象的时候，会需要实作自定的行为。

举例来说，这个功能就可以拿来作为单元测试函式库的基础，不过它只能用在虚拟方法上面。我们也可以把这个功能和自定的标注一起使用，就可以用来实作一个源码样式，像是面向导向程序一样(Aspect Oriented Programming)。

RTTI 个案研究

现在我们已经介绍过了 RTTI 的基础，以及标注的使用，现在我们可以来看一些使用这些技术的实际案例，好证明这些技术是有用的。在许多情境中，更有弹性的 RTTI 以及能够使用自定标注都是相关连的，但是我们没有足够的篇幅一一列出这些情形。所以我们会用两个简单但经典的案例让大家一步步熟悉这样的开发方法。

第一个范例程序，会展现使用标注来识别在类别中特定的信息。具体来说，我们希望可以解析一个对象，该对象的类别是属于类别架构中的一员，并且拥有说明与独特的代号，这个代号可用来参照到该对象。这可以用来处理几种情形，像是描述储存在集合组件(可能是泛型集合对象或是传统的集合对象)当中的对象。

第二个范例则是串流的范例，特别是把对象串流到 XML 档案中。我会以使用发布区的 RTTI 作为古典的目标开始介绍，接着介绍新的延伸 RTTI，最后示范如何使用标注来自定源码，并让它变得更有弹性。

在 ID 跟描述上使用标注

如果我们想要有一些方法可以让许多对象共享，最传统的作法就是定义一个具备虚拟方法的类别，然后用这个类别作为基础类别来衍生几个不同的对象，并覆写这些虚拟方法。这个方法不错，但是在类别上还是有不少限制，这些限制也会由于类别结构而产生，因为我们让这些对象拥有同一个基础类别了。

完全不同的风格(当然有优点也有缺点)是使用标注来为特定的类别、方法货属性进行标示。这个作法比较有弹性，而且并没有使用到接口，而是基于

一个相对比较慢，且容易出错的执行时期信息搜寻，和编译时期的解决方法完全不同。这表示我们不崇尚这种以接口作为更高目标的程序风格，而只是作为一个可能值得评估，以及在某些情形下使用的话可能会很有趣的作法。

描述标注类别(The Description Attribute Class)

为了这个范例，我已经定义了会被应用到的标注，作为可以被搜寻的元素。我们可以使用三个不同的标注，但应该要避免污染命名空间的标注。以下是标注类别的定义：

```
type
    TDescriptionAttrKind = (dakClass, dakDescription, dakId);
    DescriptionAttribute = class (TCustomAttribute)
    private
        FDak: TDescriptionAttrKind;
    public
        constructor Create (ADak: TDescriptionAttrKind = dakClass);
        property Kind: TDescriptionAttrKind read FDak;
    end;
```

注意到，建构函式的使用透过他的唯一一个默认值的结构，让我们使用不带参数的标注。

简单的类别

接下来，我写了两个使用标注的简单类别。每个类别都用标注来标示，并且有两个方法以相同的标注，但不同的自定方式来标示。

第一个(TPerson)拥有对应到 GetName 函式的描述，并且使用了它的 TObject.GetHashCode 方法来提供了一个临时代号，把该方法重新宣告来套用这个标注(这个方法的源码我们简单的称之为继承版本):

```
type
    [Description]
    TPerson = class
    private
      FBirthDate: TDate;
      FName: string;
      FCountry: string;
```

```
    procedure SetBirthDate(const Value: TDate);
    procedure SetCountry(const Value: string);
    procedure SetName(const Value: string);
  public
    [Description (dakDescription)]
    function GetName: string;
    [Description (dakID)]
    function GetStringCode: Integer;
  published
    property Name: string read GetName write SetName;
    property BirthDate: TDate
        read FBirthDate write SetBirthDate;
    property Country: string read FCountry write SetCountry;
  end;
```

第二个类别(TCompany)相对的更简单，因为它有自己的代号跟描述:

```
type
  [Description]
  TCompany = class
  private
    FName: string;
    FCountry: string;
    FID: string;
    procedure SetName(const Value: string);
    procedure SetID(const Value: string);
  public
    [Description (dakDescription)]
    function GetName: string;
    [Description (dakID)]
    function GetID: string;
  published
    property Name: string read GetName write SetName;
    property Country: string read FCountry write FCountry;
    property ID: string read FID write SetID;
  end;
```

虽然这两个类别有点相似，但他们在类别架构上、一般接口、或是任何面向都是完全不相干的。它们只共享了相同名字的标注而已。

简单的专案与浏览标注

标注的分享是用来显示被加入一个列表中的对象的相关信息，这个列表宣告在程序的主窗体里面:

```
FObjectsList: TObjectList<TObject>;
```

这个列表会在程序启动时被建立，并进行初始化:

```
procedure TFormDescrAttr.FormCreate(Sender: TObject);
var
    APerson: TPerson;
    ACompany: TCompany;
begin
    FObjectsList := TObjectList<TObject>.Create;
    // add a person
    APerson := TPerson.Create;
    APerson.Name := 'Wiley';
    APerson.Country := 'Desert';
    APerson.BirthDate := Date - 1000;
    FObjectsList.Add(APerson);

    // add a company
    ACompany := TCompany.Create;
    ACompany.Name := 'ACME Inc.';
    ACompany.ID := IntToStr (GetTickCount);
    ACompany.Country := 'Worldwide';
    FObjectsList.Add(aCompany);

    // add an unrelated object
    FObjectsList.Add(TStringList.Create);
```

要显示跟这个对象相关的信息(通常是有被命名的代号跟描述，如果有定义的话)，在程序中使用了透过 RTTI 功能进行的标注寻找功能。首先，使用了助手函式来判断该类别是否有被以特定的标注进行标示：

```
function TypeHasDescription (aType: TRttiType): Boolean;
var
    Attrib: TCustomAttribute;
begin
    for Attrib in AType.GetAttributes do
```

```
    begin
        if (Attrib is DescriptionAttribute) then
          Exit (True);
    end;
    Result := False;
end;
```

笔记

在这个案例里面，我们需要检查完整的类别名称，DescriptionAttribute，则不只检查『描述』内容，它是当编译器透过方括号来辨识短名称时，我们要套用标注时可以使用的标识符。

如果这个案例符合条件，程序就会取得每个方法里面的每个标注，透过巢式循环，检查该标注是不是我们正在寻找的:

```
if TypeHasDescription (AType) then
begin
    for AMethod in AType.GetMethods do
        for Attrib in AMethod.GetAttributes do
            if Attrib is DescriptionAttribute then
              ...
```

在循环的核心中，有被标注的方法会被呼叫来读取作为回传值的两个暂时字符串(稍后会被显示在用户接口上):

```
if Attrib is DescriptionAttribute then
  case DescriptionAttribute(Attrib).Kind of
      dakClass: ; // ignore
      dakDescription:
          strDescr := AMethod.Invoke(anObject, []).ToString;
      dakId:
        strID := AMethod.Invoke(anObject, []).ToString;
```

这个程序错在不该检查标注是否重复了(因为如果多个方法被标示了同一个标注的话，我们可能会建立一个例外事件)。把前面提到的范例程序整理一下，以下是完整的 UpdateList 方法:

```
procedure TFormDescrAttr.UpdateList;
var
    AnObject: TObject;
    Context: TRttiContext;
    AType: TRttiType;
    ATtrib: TCustomAttribute;
```

```
    AMethod: TRttiMethod;
    StrDescr, StrID: string;
begin
    for AnObject in FObjectsList do
    begin
        AType := Context.GetType(AnObject.ClassInfo);
        if TypeHasDescription (AType) then
        begin
            for AMethod in AType.GetMethods do
                for Attrib in AMethod.GetAttributes do
                    if Attrib is DescriptionAttribute then
                        case DescriptionAttribute(Attrib).Kind of
                        dakClass: ; // ignore
                        dakDescription: // should check if duplicate attribute
                                StrDescr := AMethod.Invoke(
                                        AnObject, []).ToString;
                        dakId:
                            StrID := AMethod.Invoke(
                                AnObject, []).ToString;
                        end;
                        // done looking for attributes
                        // should check if we found anything
                        with ListView1.Items.Add do  begin
                            Text := STypeName;
                            Detail := StrDescr;
                        end;
                end;
            end;
// else ignore the object, could raise an exception
end;
```

如果这个程序产生了不太有趣的输出结果，那么执行的方法一定会是相关的，因为我把一些类别批注掉了，其中这些类别的两个方法又使用了同一个标注，且已经让这些类别可以用外部的算法来进行处理。

换句话说，这些类别并不需要特定的基础类别，不用实作接口，也不用任何类别架构中提供任何内部源码，只需要这些类别在宣告的时候记得使用标注就行了。而管理这些类别的完全责任，则落在了外部的源码身上。

XML 串流

有一个有趣且很有用的使用 RTTI 的案例，是为对象建立一个可携的外部呈现，用来把它的状态储存到档案，或者把它透过网络传递给另一个应用程序。传统上，Object Pascal 达到这个目标的作法，是为对象的发布区属性进行串流，同样的作法也用来建立 DFM 档案。选项之一可能是制作自定义的串流框架来处理对象的串行化和还原作业。

现在，RTTI 让我们可以把对象的实际数据进行储存，而不用透过外部接口。这功能更强大了，虽然它也带来了一些额外的复杂度，例如对于内部对象的数据管理。再次强调，这个范例只是为这个技术做一个简单的示范，并没有深入探讨它的含意。

这个范例包含有三个版本，而为了简化它，把它们放在同一个项目进行编译。第一个版本是传统的 Object Pascal 作法，基于发布区的属性，其次则是使用了延伸 RTTI 跟数据字段的版本，第三个则是使用标注来自定数据对照方式。

当然要有的 XML Writer 类别

为了协助建立这样的 XML，我已经以让 XmlPersist 范例以一个延伸版的 TTrivialXmlEriter 为基础，这个类别原本是我在撰写 Delphi 2009 Handbook 这本书的时候，为了介绍 TTextWriter 类别所做的范例。在这里我就不再对它多做赘述了。我只想说，这个类别可以持续追踪它所开启的 XML 节点(感谢字符串堆栈)，然后依照后进先出的顺序关闭 XML 节点。

笔记 TTrivialXmlWriter 类别的原始码，可以从 Delphi 2009 Handbook 的范例找到，网址是:
http://github.com/MarcoDelphiBooks/Delphi2009Handbook/tree/master/07/ReaderWriter

---

我在原始的类别中加入了一些限制格式的源码，以及用来储存对象的三个方法，基于我们在本节里面要介绍的三个不同作法，以下是类别的完整宣告内容:

```
type
  TTrivialXmlWriter = class
  private
```

```
    FWriter: TTextWriter;
    FNodes: TStack<string>;
    FOwnsTextWriter: Boolean;
  public
    constructor Create (AWriter: TTextWriter); overload;
    constructor Create (AStream: TStream); overload;
    destructor Destroy; override;
    procedure WriteStartElement (const SName: string);
    procedure WriteEndElement (FIndent: Boolean = False);
    procedure WriteString (const SValue: string);
    procedure WriteObjectPublished (AnObj: TObject);
    procedure WriteObjectRtti (AnObj: TObject);
    procedure WriteObjectAttrib (AnObj: TObject);
    function Indentation: string;
  end;
```

要了解这段源码的意义，WriteStartElement 方法，会使用到 Indentation 方法来为让该行的节点可以留下两个空白，好让整个内部堆栈可以显示的比较容易阅读:

```
procedure TTrivialXmlWriter.WriteStartElement( const SName: string);
begin
    FWriter.Write (Indentation + '<' + SName + '>');
    FNodes.Push (SName);
end;
```

在范例项目中可以找到完整的源码。

传统以 RTTI 为基础的串流

在介绍过涵盖了所支持的类别之后，我们从基础开始吧，也就是把一个对象用传统 RTTI 将发布区的属性以 XML 为基础格式进行储存。

WriteObjectPublished 方法的源码相当复杂，需要多一些说明。它是以 TypInfo 单元文件为基础，且用了旧版 RTTI 当中的低阶版本的方法来取得特定对象发布区的属性(参数 AnObj)，透过以下的源码:

```
NProps := GetTypeData(AnObj.ClassInfo)^.PropCount;
GetMem(PropList, NProps * SizeOf(Pointer));
GetPropInfos(AnObj.ClassInfo, PropList);
```

```
for I := 0 to NProps - 1 do
    ...
```

这段源码的功能，是要求一定数量的属性，配置适当大小的空间给数据结构使用，并把发布区属性的信息填入这些数据结构。如果我们想知道是否可以用低阶源码来做这些事情呢?发出这个问题的同时，我们就知道为什么新版的 RTTI 要被发展出来了：就是为了简化、隐藏旧版 RTTI 的复杂。

为了每一个字段，程序会把数字跟字符串型别的属性内容解析出来，如果解析出来的是任何子对象的话，也会进一步解析这个子对象:

```
StrPropName := UTF8ToString (PropList[i].Name);
case PropList[i].PropType^.Kind of
    tkInteger, tkEnumeration, tkString, tkUString, ...:
    begin
        WriteStartElement (StrPropName);
        WriteString (GetPropValue(AnObj, StrPropName));
        WriteEndElement;
    end;
    tkClass:
    begin
        internalObject := GetObjectProp(AnObj, StrPropName);
        // recurse in subclass
        WriteStartElement (StrPropName);
        WriteObjectPublished (internalObject as TPersistent);
         WriteEndElement (True);
    end;
end;
```

这里有点复杂，但为了范例，并让大家对传统方法有个概念，这应该是合理的。

为了示范这个程序的效果，我已经写了两个类别(TCompany 跟 TPerson)，这两个类别是从前一个范例借过来的。然而现在 Company 对象有了一个额外的 Person 型别属性，叫做 Boss。在实际的世界，这可能很复杂，但在这个范例中，这是很合理的假设。以下是这两个类别的发布区属性宣告:

```
type
    TPerson = class (TPersistent)
      ...
```

```
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
  end;

  TCompany = class (TPersistent)
    ...
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
    property ID: string read FID write FID;
    property Boss: TPerson read FPerson write FPerson;
end;
```

这个程序的主窗体有一个按钮，用来建立并连结这两个类别建立出来的两个对象，并且把它们储存到一个 XML 串流中，稍后我们会让它显示出来。以下是串流化的源码:

```
SS := TStringStream.Create;
XmlWri := TTrivialXmlWriter.Create (SS);
XmlWri.WriteStartElement('Company');
XmlWri.WriteObjectPublished(ACompany);
XmlWri.WriteEndElement;
```

所产生的 XML 内容如下:

```
<company>
  <Name>ACME Inc.</Name>
  <Country>Worldwide</Country>
  <ID>29088851</ID>
  <Boss>
    <Name>Wiley</Name>
    <Country>Desert</Country>
  </Boss>
</company>
```

以延伸版 RTTI 建立的串流数据字段

透过 Object Pascal 的高阶 RTTI 功能，我可以把旧的源码改写成使用延伸版 RTTI 来存取发布区的属性。而我所要做的，是使用它来把对象的内部表示

进行储存，也就是私有区的数据字段。我所做的不只是更为核心的事情，而且是使用更为高阶的功能来达成的，WriteObjectRtti 方法完整的源码如下：

```
procedure TTrivialXmlWriter.WriteObjectRtti(AnObj: TObject);
var
    AContext: TRttiContext;
    AType: TRttiType;
    AField: TRttiField;
begin
    AType := AContext.GetType (AnObj.ClassType);
    for AField in AType.GetFields do
    begin
        if AField.FieldType.IsInstance then
        begin
            WriteStartElement (AField.Name);
            WriteObjectRtti (AField.GetValue(AnObj).AsObject);
            WriteEndElement (True);
        end
        else
        begin
             WriteStartElement (AField.Name);
             WriteString (AField.GetValue(AnObj).ToString);
             WriteEndElement;
        end;
    end;
end;
```

产生的 XML 也很相似，只是相对的比较不清楚，因为数据字段的名称比属性的名称更不清楚：

```
<company>
    <FName>ACME Inc.</FName>
    <FCountry>Worldwide</FCountry>
    <FID>29470148</FID>
    <FPerson>
        <FName>Wiley</FName>
        <FCountry>Desert</FCountry>
    </FPerson>
</company>
```

然而另一个更大的不同，是在这个案例中，类别不需要从 TPersistent 类别衍生而来，也不需要有任何特别的限制。

使用标注来自定串流化

除了标签名称的问题以外，另外还有一个我还没提过的问题。使用 XML 的标签名会是很复杂的标识符，这不是个好主意。同时，在这段源码里面，无法从 XML 串流的数据来排除特定的属性。

笔记

Object Pascal 的属性串流可以透过 stored 设定来控制，我们可以透过使用 TypInfo 单元文件来读取它。再说一次，这个作法并不简单，也不明快，虽然 DFM 串流机制把它运用的相当有效率。

这些是我们可以用标注来加以厘清的问题，但它的缺点是在宣告类别的时候大量使用标注，这种风格我并不喜欢。对于新版的源码，我已经透过一个选用的参数定义了一个标注化的建构函式：

```
type
    XmlAttribute = class (TCustomAttribute)
    private
        FTag: string;
    public
        constructor Create (StrTag: string = '');
        property TagName: string read FTag;
    end;
```

标注化的串流源码则是一个以最后一版，以延伸 RTTI 为基础的演变。唯一的不同是现在的程序会呼叫 CheckXmlAttr 助手函式来确认该数据字段是否有 xml 标注，以及选用的标签名称加以注释:

```
procedure TTrivialXmlWriter.WriteObjectAttrib(AnObj: TObject);
var
    AContext: TRttiContext;
    AType: TRttiType;
    AField: TRttiField;
    StrTagName: string;
begin
     AType := AContext.GetType (AnObj.ClassType);
     for AField in AType.GetFields do
     begin
```

```
        if CheckXmlAttr (AField, StrTagName) then
        begin
            if AField.FieldType.IsInstance then
            begin
                WriteStartElement (StrTagName);
                WriteObjectAttrib (AField.GetValue(AnObj).AsObject);
                WriteEndElement (True);
            end
            else
            begin
                WriteStartElement (StrTagName);
                WriteString (AField.GetValue(AnObj).ToString);
                WriteEndElement;
            end;
        end;
    end;
end;
```

最相关的源码是在 CheckXmlAttr 助手函式里面的:

```
function CheckXmlAttr (aField: TRttiField; var strTag: string): Boolean;
var
    Attrib: TCustomAttribute;
begin
    Result := False;
    for Attrib in AField.GetAttributes do
        if Attrib is XmlAttribute then
        begin
            StrTag := XmlAttribute(attrib).TagName;
            if StrTag = '' then // default value
                StrTag := AField.Name;
            Exit (True);
        end;
end;
```

没有 XML 标注的数据字段会被忽略掉，在 XML 输出的标签是可以自定的。为了示范这一点，程序中有以下的类别(这一次我已经略过了发布区的属性，因为它们没相关):

```
type
    TAttrPerson = class
```

```
  private
    [xml ('Name')]
    FName: string;
    [xml]
    FCountry: string;
    ...
  TAttrCompany = class
  private
    [xml ('CompanyName')]
    FName: string;
    [xml ('Country')]
    FCountry: string;
    FID: string; // omitted
    [xml ('TheBoss')]
    FPerson: TAttrPerson;
    ...
```

透过这些宣告，XML 输出值看起来会像以下这些 XML 数据(请注意到卷标名称，ID 是被忽略掉了，而预设的名称则是 FCountry 字段):

```
<company>
 <CompanyName>ACME Inc.</CompanyName>
 <Country>Worldwide</Country>
 <TheBoss>
     <Name>Wiley</Name>
     <FCountry>Desert</FCountry>
 </TheBoss>
</company>
```

这里的不同是我们可以对哪些字段要被纳入 XML，以及如何在 XML 里面对它们进行命名很有弹性，而在前一版的作法则完全不能随意处理。

虽然这只是个骨架版本的实作，我还是希望读者们有机会能看一下如何一步步的以传统 RTTI 来建立出最后的版本，这会让我们对于几个不同的技术之间如何实作留下印象。

而很重要，必须一定要记住的是，事实上，使用标注并不一定是最好的解决方法。另一方面，很明显的 RTTI 跟标注在任何一种情境中都提供了许多

威力跟弹性，我们需要透过这个技术才能在运行时间对结构跟未知的对象进行了解。

其他以 RTTI 为基础的函式库

要为这一章做结论，我想指出一个事实，目前有一些函式库，包含内建在 Delphi 与第三方的函式库，都已经开始并入延伸 RTTI 了。一个很明显的例子是表达式绑定的机制已经跟背景的可视化绑定逐渐一致。我们可以建立绑定表达式，把它指派给一个表达式(例如 Text 里的一个字符串，可以进行处理，例如字符串连接)，或是让该表达式参考到一个额外对象与其数据字段。

即使我不想太深入这个主题，它是一个很特定的函式库，并且不是 Object Pascal 的一部分，也不是核心系统的一部分，我想用一个简单的列表来让大家有这个概念:

```
var
    BindExpr: TBindingExpression;
    Pers: TPerson;
begin
    Pers := TPerson.Create;
    Pers.Name := 'John';
    Pers.City := 'San Francisco';

    BindExpr := TBindingExpressionDefault.Create;
    BindExpr.Source := 'person.name + " lives is " + person.city');
    BindExpr.Compile([
      TBindingAssociation.Create(Pers, 'person')]);
    Show (BindExpr.Evaluate.GetValue.ToString);
    pers.Free;
    BindExpr.Free;
end;
```

注意到这段程序的优点来自于我们可以在运行时间改变表达式的内容(虽然在上面的范例源码里面，表达式的内容是用一个字符串常数写成的)。表达式的内容可以从一个 Edit 组件让用户输入，或者可以动态的从几个不同的表达式结合而来。它先被指派给 TBindingExpression 对象，接着在运行时间被透过呼叫 Compile 方法，加以分析、编译(这里所指的是字符串被转换

成标识符的格式，不是真的编译成汇编语言的执行码)。然后会在执行时使用 RTTI 来存取 TPerson 对象。

缺点是这个作法会使得表达式的执行明显的比预先编译完成的 Object Pascal 机器码来的慢。换句话说，我们得在效能跟弹性上取得一个平衡。也可以说可视化直接绑定模型的功能提供了功能强大、容易使用的开发者经验。

17:TObject 与 System 单元文件

在任何 Object Pascal 编程语言的应用程序核心里，都有类别的架构存在。在系统的每一个类别都一定是从 TObject 类别所衍生出来的，所以整个架构只有单一的根源。这使得我们可以使用 TObject 数据型别来做为系统所任何一个类别的数据型别的替代品。

TObject 类别是定义在核心的 RTL 当中，单元文件的名字是 System，因为这个单元文件非常重要，所以在任何的编译动作当中，都会自动引入这个单元文件。我们不会逐一介绍所有在 System 单元文件里面的类别跟函式，我们要把心力放在 TObject 这个最重要的主角之上。

笔记 我们可以花很长的篇幅来争论核心系统类别，像 TObject 是否算是 Object Pascal 编程语言的一部分，或者算是执行时期函式库的一部分。其他在 System 单元文件的功能也一样，一个单元文件重要到会自动在编译时被引入。(事实上如果我们把它加入到 uses 区块，反而会造成错误)。这样的争论没有意义，所以我们在这里不去谈他。

TObject 类别

正如我刚提到的，TObject 类别是非常特殊的类别，因为所有其他的类别都继承自 TObject。当我们宣告一个新的类别时，事实上，如果我们没有指定一个基础类别，该类别就会自动继承自 TObject 了。在编程语言的术语里，这样的型别系统被称为『单一根源类别架构』，这是 Object Pascal 的功能，C#，Java 跟一些现代的编程语言也都如此设计。值得一提的例外是 C++，它没有单一根源的基础类别，且允许我们定义多重完全分离的类别架构。

TObject 这个基础类别并不是一个让我们直接建立实体来用的类别。然而我们最后可以很容易的在很多地方使用它。每次我们需要一个可以用来储存对象或其任何其他型别变量时，我们就可以把它宣告为 TObject 型别。这个

用法有个很好的例子，就是在组件函式库当中的事件处理程序，所有的事件处理程序通常都使用 TObject 作为第一个参数的型别，通常称之为 Sender。这表示任何实际类别的对象都可以作为 Sender。很多泛型集合也是对象的集合，且有一些情境中，TObject 型别是直接被用上的。

在以下的情境当中，我会介绍到这个类别的一些功能，这些功能是所有位于 System 单元文件里面的类别都可以用的。

建立与毁灭

虽然直接建立 TObject 没有什么意义，不过这个类别的建构函式跟解构函式还是很重要的，因为它们会被所有其他类别自动继承。如果我们定义了一个没有建构函式的类别，我们还是可以呼叫它的 Create 方法，这会呼叫 TObject 的建构函式，这是个空的函式(因为在这个基础类别中没有什么要初始化的)。这个 Create 建构函式不是虚构的，且我们可以在自定的类别中把它整个替换掉，除非这个没做什么事的建构函式就已经够用了。呼叫基础类别的建构函式，对任何的子类别实作都是一个好习惯，即使直接呼叫 TObject.Create 没有什么特别的用处。

笔记 我已经强调这是一个非虚拟的建构函式，因为有另一个核心函式库类别，TComponent，是有定义虚拟建构函式的。TComponent 类别的虚拟建构函式是整个串流化系统当中的关键角色，我们将在下一章里面介绍。

为了毁灭一个对象，TObject 类别里有一个 Free 方法(这个方法最后会呼叫 Destroy 解构函式)，我们已经在第 13 章里面介绍过了，并在该章里面提出许多建议来端正内存使用的风气，所以在这里就不再赘述了。

物件的二三事(Knowing About an Object)

在 TObject 里面有一组有趣的方法，它们会回传关于型别的一些信息。最常用的就是 ClassType 跟 ClassName 方法。ClassName 方法会把类别的名称用字符串来回传。因为它是一个类别方法(就像大多数 TObject 的类别方法一样)，我们可以透过对象或类别来呼叫它。假设我们定义了一个对象名为 TButton，并以这个类别建立了一个名为 Button1 的对象。那么以下的两杭源码，回传的结果就会完全相同：

```
Text := Button1.ClassName;

Text := TButton.ClassName;
```

当然，我们也可以把这些应用在常见的 TObject 上面，但我们不会得到 TObject 的信息，而是关于该对象变量所属的类别的信息。例如在按钮的 OnClick 事件处理程序里，我们可以呼叫:

```
Text := Sender.ClassName;
```

这会回传跟前面两个指令一样的结果，回传值将会是”TButton”这个字符串。这是因为类别名称是在执行时期决定的(由特定的对象本身决定)，而不是由编译器来决定的(编译器只会认为它是一个 TObject 型别的对象)。当然，如果参考的内容没有被指派，也就是说变量内容是 nil，任何试图呼叫该类别方法都会造成例外。

取得类别名称对于侦错、记录跟一般显示类别信息都是很有用的，通常它也对于存取该类别的类别参考更为重要。举例来说，比较两个类别参考(会是两个数值，记录内存地址)比用两个类别的名称来做字符串比对更好。我们可以透过 ClassType 方法来取得类别参考，而 ClassParent 方法则会回传当前对象的基础类别的类别参考，允许浏览基础类别列表。唯一的例外是当该方法回传 TObject 是 nil 时(因为它没有父代类别)。一旦我们有了类别参考，我们就可以用它来呼叫任何类别方法，包含 ClassName 方法。

另一个也很有趣的方法会回传关于类别的信息，是名为 InstanceSize 的方法，这个方法会回传一个对象在执行时期的大小，回传的这个数字是指该对象的数据字段所要求的大小总和(以及继承自基础类别的数据)。这个功能是当系统需要为该类别的新实体进行配置时，内部使用的功能。

笔记 虽然我们可能会觉得 SizeOf 这个全局函式也会提供相同的信息，但这个函式实际上是回传对象参考的大小，也就是指标的大小，指标现在不是 4 bytes 就是 8 bytes，要看是在 32 位还是 64 位的操作系统上-跟对象本身所占用的空间完全无关。另一方面 InstanceSize 会回传该字段使用的空间大小，但并非该对象实际使用的内存空间总和，因为该字段可能是指向字符串或者其他对象的参考，这就可能使用了额外的内存。

更多 TObject 类别的方法

还有些 TObject 类别的方法我们可以用在任何对象上(也可以用在任何类别或类别参考，因为这些方法是类别方法)。以下是一部分的列表，搭配简单的描述：

- ClassName 以字符串回传该类别的名称，主要是用以显示。
- ClassNameIs 检查类别名称是否与参数值相同

- ClassParent 回传当前类别的父类别的类别参考或者对象的类别。我们可以从 ClassParent 浏览到 ClassParent，直到浏览的对象到达 TObject 为止，对 TObject 呼叫这个方法，回传值会是 nil。
- ClassInfo 回传类别所属的内部的，低阶的执行时期型别信息(RTTI)。这是早期在 TypInfo 单元文件里面使用的，但现在已经以 RTTI 单元文件的功能取代了它，我们在第 16 章已经介绍过了。目前仅剩内部使用，它还是类别取得 RTTI 信息的途径。
- ClassType 回传该对象所属类别的参考(这不是类别方法，所以只能透过对象来呼叫)。因为这是对象参考，两个在不同单元文件宣告的类别不会相同，所以在像 Delphi 这样的强型别语言里面不会有问题。
- InheritsFrom 检测一个类别是否继承自(直接或间接都算)特定的类别(这效果很像 is 运算符号，但 is 运算符号的实作比较全面)。
- InstanceSize 回传对象的数据大小，单位是 Bytes。这个数字会是所有字段的总和，加上一些额外的特定保留位(包含例如类别参考)。要留意到。再一次留意到，这是实体的大小，而指向实体的参考只是一个指标(4 或 8 bytes，要看操作系统而定)
- UnitName 回传定义该类别的单元文件的名字。对于描述一个类别是很有用的。事实上，类别名称在系统中并不是唯一的。当我们看完最后一章，就会知道只有完整连同单元文件一起列出的类别名称(包含单元文件名称跟类别名称的组合，以.加以连接)在应用程序中才是唯一的。
- QualifiedClassName 回传整个单元文件跟类别名称的连结，在执行中的系统里，这个值会是唯一的。

以上 TObject 的方法适用于每一个类别的对象，因为 TObject 是每个类别共通的祖先类别。

以下是我们透过这些方法来存取类别信息的例子:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
    Memo1.Lines.Add ('Class Name: ' + Sender.ClassName);
    if Sender.ClassParent <> nil then
        Memo1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);
    Memo1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));
```

这段源码会检测看看 ClassParent 的内容是不是 nil，万一我们正好用到 TObject 型别的实体，它就不会有基础类别了。我们可以用其他方法来进行检测。例如，我们可以用以下的源码来检验 Sender 对象是不是特定的型别：

```
if Sender.ClassType = TButton then ...
```

我们也可以检测 Sender 参数是否对应到特定的对象，用这个方式检测:

```
if Sender = Button1 then...
```

不检查特定的类别或对象，我们通常需要测试某个类别的对象型别是否兼容，也就是说，我们会需要检查一个类别的对象是否属于特定的类别或者该类别的子类别。这会让我们知道我们是否可以透过该对象呼叫定义在类别中的方法。这个检测可以透过呼叫 InheritsFrom 方法来完成，当我们使用到 is 运算符号的时候，这个运算符号的其中一个不同，是它也会处理 nil。以下两个测试是完全一样的：

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

显示类别信息

当我们取得类别参考时，我们可以把这个类别参考的所有基础类别的名称加入到一个列表中。在以下的程序片段中，MyClass 的所有基础类别都会被加到 ListBox 组件当中：

```
ListParent.Items.Clear;
while MyClass.ClassParent <> nil do
begin
    MyClass := MyClass.ClassParent;
    ListParent.Items Add (MyClass.ClassName);
end;
```

您应该注意到了，我们在 while 循环当中使用了类别参考，这个类别参考会用来检查该类别是否存在父代类别(万一目前的类别是 TObject, 就不会有父代类别)。或者我们也可以把 while 的判断条件写成这样:

```
while not MyClass.ClassNameIs ('TObject') do... // Slow, error prone
while MyClass <> TObject do... // Fast, and readable
```

TObject 的虚拟方法

从 Object Pascal 的早期，TObject 类别的结构中就已经相当稳定，我们可以从当中找到三个很有用的虚拟方法。这些方法可以被任何对象呼叫，就像 TObject 的其他方法一样，但相关的是，这些方法我们在自己建立的子类别中都应该覆写或重新写过。

笔记 如果您已经使用过.NET 架构，您可能会立刻发现这些方法是 C#基础类别函式库的 System.Object 类别的一部分。类似的方法也在 Java 当中被设计在基础类别当中，在 JavaScript 里面也很常见，其他语言当中也是。这些方法(例如 toString)的来源，可以追溯到 SmallTalk，它应该算是第一个 OOP 语言。

ToString 方法

ToString 这个虚拟方法是提示要回传以文字化表示(描述或者是把对象进行串行化的结果)一个特定对象。在 TObject 当中预设的实作源码是回传该类别的名字:

```
function TObject.ToString: string;
begin
    Result := ClassName;
end;
```

当然这样距离实用还远的很。理论上，每个类别都应该提供一个方式来把自己对使用者作介绍，例如当一个对象被加到可视化的列表时。在运行时间函式库中的部分类别有对这个函式进行覆写，像是 TStringBuilder, TStringWriter,以及 Exception 类别，Exception 类别会回传整个列表的例外讯息(我们已经在第九章的巢状例外与内部例外机制这一节当中介绍过)。

透过一个标准的方法为任何对象回传文字表现形式是相当有意思的想法。而且我推荐大家要好好利用 TObject 类别这个核心功能，把它当成编程语言内建功能一样。

笔记 要注意 ToString 方法在『语义上多载』了在 Classes 单元文件中定义的 toString 标识符，该定义的意思是『解析 token 字符串』。因此它参照了 Classes.toString。

Equals 方法

Equals 虚拟函式是提示要检测两个对象是否有相同的逻辑内容，这个检测跟检测两个变量是否在内存中指向同一个对象是不同的，如果只是要检测两个变量是否在内存当中指向同一个对象，我们可以用=这个运算符号。然

而，这样看上去的确挺让人迷惑的，预设的实作源码如下，我们该想想有没有更好的办法:

```
function TObject.Equals(Obj: TObject): Boolean;
begin
    Result := Obj = Self;
end;
```

举个使用这个方法的例子(透过适当的覆写)，在 TStrings 类别当中的 Equals 方法会把该类别的字符串列表的总数跟内容一一进行比对，如果完全相符的话，实际字符串的内容会被一一比对，直到有其中一个项目不相符或者到最后一项也相同，如果到最后一项都还相同，就代表整个字符串列表完全相符。

使用这个技术最显著的函式库是对泛型支持的功能，特别是在 Generics.Default 跟 Generics.Collections 单元。通常在函式库或架构中把对象概念定义为"内容是否相等"，而不是"是否是同一个对象"是很重要的。能透过一个标准的机制来比对对象的『内容』绝对是很大的好处。

GetHashCode 方法

GetHashCode 虚拟函式是从.NET 架构借来的另一个提示，让每一个类别可以为其对象计算哈希值(Hash)。预设的源码会回传一个整数值，看起来是该对象的地址：

```
function TObject.GetHashCode: Integer;
begin
    Result := Integer(Self);
end;
```

笔记

对象建立时的内存地址，通常都是在 heap 内存里面的限定区域，所以用这个数字来建立哈希值并不实际，对于 Hash 算法的使用会有负面的影响。强烈建议大家要自行重写这个方法，依照对象内部的逻辑数据与好的 Hash 算法来建立哈希值(Hash)，不要用对象的地址。这个作法在效能表现上会有显著的改善喔。

GetHash 虚拟方法在不少集合类别(只要有支持哈希表-Hash Table 的类别)当中都有用到，且这些类别是把 Hash 当成优化部分程序的方法，像是 TDictionary<T>。

使用 TObject 虚拟方法

以下是以 TObject 虚拟方法为基础的范例，这个范例中有一个类别，它覆写了这些虚拟方法当中的两个：

```
type
  TAnyObject = class
  private
      FValue: Integer;
      FName: string;
  public
       constructor Create (AName: string; AValue: Integer);
       function Equals(obj: TObject): Boolean; override;
       function ToString: string; override;
  end;
```

在这三个方法的实作中，我单纯的把呼叫 GetType 改成呼叫 ClassType:

```
constructor TAnyObject.Create(AName: string;
    AValue: Integer);
begin
    inherited Create;
    FName := AName;
    FValue := AValue;
end;
function TAnyObject.Equals(Obj: TObject): Boolean;
begin
    Result := (Obj.ClassType = self.ClassType) and
          ((Obj as TAnyObject).Value = self.Value);
end;
function TAnyObject.ToString: string;
begin
    Result := Name;
end;
```

注意到对象是否相同，是依照它们是否属于同一个类别，且当中的数据是否相同，当中判断的字符串表现内容，只包含了 FName 这个字段。

这个程序在启动时，以这个类别建立了一些对象:

```
procedure TFormSystemObject.FormCreate(Sender: TObject);
```

```
begin
  Ao1 := TAnyObject.Create ('Ao1', 10);
  Ao2 := TAnyObject.Create ('Ao2 or Ao3', 20);
  Ao3 := Ao2;
  Ao4 := TAnyObject.Create ('Ao4', 20);
  ...
```

注意到这两个参考(Ao2 跟 Ao3)是指向内存当中的同一个对象，且最后一个对象(Ao4)有着相同的数值内容。这个程序具备了用户接口，让用户可以选择任两者，并对被选上的对象进行比较，同时使用 Equals 跟直接透过参考来做比较。以下是比较的一些结果：

```
Comparing Ao1 and Ao4
Equals: False
Reference = False

Comparing Ao2 and Ao3
Equals: True
Reference = True

Comparing Ao3 and Ao4
Equals: True
Reference = False
```

这个程序有另一个按钮，用来检测该按钮的其他方法：

```
var
     Btn2: TButton;
begin
   Btn2 := BtnTest;
   Log ('Equals: ' +
      BoolToStr (BtnTest.Equals (Btn2), True));
   Log ('Reference = ' +
      BoolToStr (btnTest = Btn2, True));
   Log ('GetHashCode: ' +
      IntToStr (btnTest.GetHashCode));
   Log ('ToString: ' + BtnTest.ToString);
end;
```

执行结果如下(透过运行时间改变的哈希值):

```
Equals: True
Reference = True
GetHashCode: 28253904
ToString: TButton
```

总结 TObject 类别

总结一下，在最新版的编译器里，TObject 类别当中有完整的接口(已经把大多数的的 IFDEF 跟低阶的多载部分省略了，并略过私有区跟保护区):

```
type
    TObject = class
    public
      constructor Create;
      procedure Free;
      procedure DisposeOf;
      class function InitInstance(Instance: Pointer): TObject;
      procedure CleanupInstance;
      function ClassType: TClass; inline;
      class function ClassName: string;
      class function ClassNameIs(const Name: string): Boolean;
      class function ClassParent: TClass;
      class function ClassInfo: Pointer; inline;
      class function InstanceSize: Integer; inline;
      class function InheritsFrom(AClass: TClass): Boolean;
      class function MethodAddress(const Name: string): Pointer;
      class function MethodName(Address: Pointer): string;
      class function QualifiedClassName: string;
      function FieldAddress(const Name: string): Pointer;
      function GetInterface(const IID: TGUID; out Obj): Boolean;
      class function GetInterfaceEntry(
            const IID: TGUID): PInterfaceEntry;
            class function GetInterfaceTable: PInterfaceTable;
      class function UnitName: string;
      class function UnitScope: string;
    {$IFDEF AUTOREFCOUNT}
        function __ObjAddRef: Integer; virtual;
        function __ObjRelease: Integer; virtual;
    {$ENDIF}
```

```
    function Equals(Obj: TObject): Boolean; virtual;
    function GetHashCode: Integer; virtual;
    function ToString: string; virtual;
    function SafeCallException(ExceptObject: TObject;
          ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  public
    property RefCount: Integer read FRefCount;
    property Disposed: Boolean read GetDisposed;
  end;
```

Unicode 跟类别名称

MethodAddress 跟 FieldAddress 具备多载的版本，依照其参数的型别不同，我们可以使用 UnicodeString(通常是 UTF-16)或 ShortString 作为参数，如果使用 ShortString 参数，则输入的字符串会被认为是 UTF-8 字符串。事实上这些版本都会使用 Unicode 字符串，它们会自动呼叫 UTF8EncodeToShortString 来进行转换:

```
function TObject.FieldAddress(const Name: string): Pointer;
begin
  Result := FieldAddress(UTF8EncodeToShortString(Name));
end;
```

因为 Object Pascal 本身就已经支持 Unicode 了，在 Object Pascal 内部的类别名称是使用 ShortString(它是一个 1 位的字符数组)来表示的，但是是使用 UTF-8 的编码法，而不是传统的 ANSI 编码法喔。这个作法同时用在 TObject 阶层跟 RTTI 阶层。

举例来说 ClassName 方法是这样实作的(用了很丑的低阶源码):

```
class function TObject.ClassName: string;
begin
  Result := UTF8ToString (
```

```
            PShortString (PPointer (
                Integer(Self) + vmtClassName)^)^);
end;
```

跟 TypInfo 单元文件很相似，所有存取类别名称的函式都会在内部把 UTF-8 ShortString 表示转换成 UnicodeString。类似的动作也会发生在属性的名称处理上。

System 单元

TObject 很显然已经是 Object Pascal 语言的核心角色了，我们已经很难去区分它到底是编程语言的一部分，或是运行时间函式库的一部分，在 System 单元中，还有一些低阶的类别是构成基础且已经跟编译器支持整合在一起了。

这个单元的大多数内容，是为了低阶数据结构、简单记录结构、函式、程序以及一些类别而制作的。

在此我们要集中心力在这些类别上，但不可否认的，在 System 单元中的选多其他功能也是 Object Pascal 的关键。举例来说，System 单元里面定义了许多 Pascal 里面固有的函式，这些函式没有实际的源码，会由编译器直接加以解析。例如 SizeOf，编译器就会直接把这个函式换成该参数数据结构的实际大小。

我们可以阅读一下 System 单元档案里面的批注来了解它的独特地位(几乎解释了为什么在浏览系统变量的时候都会需要这个单元)：

```
{ Predefined constants, types, procedures, }
{ and functions (such as True, Integer, or }
{ Writeln) do not have actual declarations.}
{ Instead they are built into the compiler }
{ and are treated as if they were declared }
{ at the beginning of the System unit.       }
```

译文如下：

预先定义的常数、型别、程序与函式(像是 True, Integer 或 Writeln)并没有实际的宣告。反之，它们是内建在编译器里的，并且会被当成宣告在 System 单元文件的最开头一样。

阅读这个单元里面的原始码可能会很无聊，也是因为我们可能在这个单元文件里面发现一些执行时期函式库的低阶源码。所以我决定只挑里面很有限的内容来介绍。

被选上的系统型别

刚刚提过，System 单元定义了核心数据型别，以及许多不同值类型的型别别名、有序型别，以及字符串。还有许多系统在低阶处理会使用到的核心数据型别(包含列举型别、记录，以及强制型别别名)等，这些都值得一提:

- ✧ TVisiblityClasses 是用在 RTTI 可视范围设定的列举型别(详情请见第 16 章)。
- ✧ TGUID 是在 Windows 平台上用来表现 GUID 的纪录，但也可以用在其他支持 GUID 的操作系统上。
- ✧ TMethod 是一个核心的记录，用来表现事件处理程序要使用的结构，包含一个指向方法地址的指针，以及一个指向当前对象的指针(详情请见第 10 章)
- ✧ TMonitor 是一个记录，用来实作线程同步的机制(称为”Monitor”)，这是由 C.A.R Hoare 跟 Per Brinch Hansen 所发明的，在维基百科上面也有名为”Monitor synchronization”的详细介绍。这是 Object Pascal 编程语言中线程核心的功能，而 TMonitor 信息是在系统中的每个对象都存在的。
- ✧ TDateTime 是 Double 型别的强制型别别名，用来储存日期信息(使用其中的指数部分来储存)与时间信息(用其十进制的部份)。更进一步的别名包含 TDate 跟 TTime。这些型别在第二章里面有介绍。
- ✧ THandle 是值类型的别名，用来表示操作系统对象的参考，通常称为”Handle”(至少是在 Windows API 的范畴中)。
- ✧ TMemoryManagerEx 是用来储存核心内存处理的记录，可以把系统的内存管理模块换成自定的模块(这是比较新版的 TMemoryManager)，新的模块仍旧有向前支持。
- ✧ THeapStatus 是一个记录用来储存关于 heap 内存状态的信息，我们在第 13 章有提过。
- ✧ TTextLineBreakStyle 是一个列举型态，代表特定操作系统对文本文件的换行格式。DefaultTextLineBreakStyle 这种型别的全局变量保存了目前的信息，会在许多系统函式库里面使用到。同样的，sLineBreak 常

数则表达了字符串型别里相同的作用。

System 单元里面的接口

接口型别有好几种(而且有些类别在核心阶层实作了接口的功能)，它们是 System 单元的一部分，值得我们一看。以下是 System 单元里面跟接口相关的一些型别：

- ✧ IInterface 是所有接口的基础，所有接口都要从这个型别继承而来，就像 TObject 之于所有其他的类别一样。
- ✧ IInvokable 跟 IDispatch 是支持动态呼叫的接口(部分与 Windows COM 的实作绑在一起)。
- ✧ 列举的支持跟比较的动作是透过以下的接口来定义的：IEnumerator，IEnumerable, IEnumerator<T>, IENumerable<T>, IComparable, IComparable<T>跟 IEquatable<T>。

还有一些核心类别提供了接口的基础实作。我们可以直接从这些类别衍生出新的类别，就可以实作接口了，在第 11 章里面我们介绍过了：

- ✧ TInterfacedObject 是一个类别，在当中已经有对参考计数与对接口 ID 检查的基本实作。
- ✧ TAggregatedObject 跟 TContainedObject，这两个类别特别提供对 aggregate 对象跟实作的语法。

被选上的系统函式

在 System 单元里面，内建跟标准程序与函式非常多，但大多数都不常被用到。以下是我们选出来的核心程序跟函式，每个 Object Pascal 开发人员都应该知道这些：

- ✧ Move 是在系统中核心内存复制的程序，把特定数量的内存很单纯的从一个地址复制到另一个地址去(很强大、速度快，但有一点点的危险性)。
- ✧ ParamCount 跟 ParamStr 函式可以用来处理命令行指令的参数(在图形接口系统中，像 Windows 跟 Mac 也都可以运作的很好)
- ✧ Random 跟 Randomize 是两个传统的函式(似乎是从 BASIC 借来的概念)，提供我们产生随机数(虚拟的随机，但一定要记得呼叫 Randomize，产生的数目才会真的随机，不然每次执行程序你得到的随机数字都会一样)。
- ✧ 很大量的核心数学运算函式，在这里全略过。

- ✧ 许多字符串处理跟字符串转换函式(在 UTF-16，Unicode，UTF-8，ANSI，以及其他字符串格式之间转换)，其中包含一些是跟平台相关的。

笔记 这些函式中，包含一些间接的定义。换句话说，这些函式实际上是一个指标指向实际的函式。所以原始系统的行为可能会在运行时间被源码动态的替换掉。(当然，如果我们知道自己在做什么，这也是把整个系统搞死的好办法)

预先定义的 RTTI 标注

在本章的最后，我想介绍的最后一组数据型别是跟标注相关的，额外的 RTTI 信息，我们可以连接到编程语言的任何符号。这个主题我们已经在第 16 章里面介绍过，但当时我并没有提到过在系统中预先定义的标注。

以下是在 System 单元中定义的标注类别：

- ✧ TCustomAttribute 是所有自定标注的基础类别。这是我们可以当成所有需要使用到标注的类别的根源基础类别(且这是编译器可以识别类别的唯一方法，作为一个标注，因为没有特别宣告的语法)。
- ✧ WeakAttribute 也是用来在 ARC 环境中标明 weak 参考(详见第 13 章)
- ✧ UnsafeAttribute 也是在 ARC 环境中作为特别处理之用的(详见第 13 章)
- ✧ RefAttribute 很直觉是给参考值使用的。
- ✧ VolatileAttribute 标示会自动消灭的变量，这种变量可以从外部进行修改，而且无法被编译器优化。
- ✧ StoredAttribute 是可以表达属性中的 stored 旗目标另一种方法。
- ✧ HPPGENAttribute 控制 C++接口档案(HPP)的生成。
- ✧ HFAAttribute 可以用来优化 ARM 64 位 CPU 的参数传递，并可以控制同构型浮点数整合(Homogeneous Floating-point Aggregate, 缩写为 HFA)

在 System 单元中还有更多内容，但比较适合专家级的开发人员。我们先继续介绍吧，在最后一章里面，我们会触及 Classes 单元，以及其他 RTL 的功能。

18:其他核心 RTL 的类别

如果 TObject 类别跟 System 单元可以被认为是 Object Pascal 的结构化部分，其他编译器本身需要用来建置任何应用程序的功能，其他在执行时期函式库的功能，都会被认为是核心系统的选项延伸。

RTL 当中有许多的系统函式，包含最常用的标准动作，以及有些部分可以回溯到 Turbo Pascal 的时代，在前 Object Pascal 编程语言的时期。在 RTL 的许多单元文件都是函式跟子程序，包含核心的功能(SysUtils)，数学函式(Math)，字符串处理(StringUtils)，日期时间处理(DateUtils)等等。

在本书里，我并没有想要深入到 RTL 的传统部分，而是要专心介绍核心类别(这些是 Object Pascal(VCL 跟 FireMonkey)用于视觉组件的基础)，以及其他子系统。例如 TComponent 类别，定义了"以组件为基础"的概念架构。它也是内存管理跟其他基础功能的基石。TPersistent 类别则是组件串流化呈现的关键。

我们还会再看其他许多的类别，因为 RTL 涵盖的范围很大，包含了文件系统、核心线程的支持、平行程序函式库、字符串建立、许多不同型别的集合，以及容器类别、核心图形几何结构(像是点跟长方形)、核心数学结构(像是向量跟矩阵)，还有许许多多的功能。

因为本书的重点是 Object Pascal 编程语言，并不是函式库的手册，我们在此只会选择几个类别，选择的原因是因为它们的角色极为关键，或者是因为该类别是这几年才发表，而大多数的开发人员都没有留意到它们。

Classes 单元

Object Pascal RTL 类别函式库的基础(也是可视化函式库的基础)可以说就是 System.Classes。这个单元包含了许多最常用到的类别的集合，不包含特殊用途的类别。当中重要的类别值得一看，我们接下来就深入分析当中最重要的一些类别吧。

在 Classes 单元中的类别

以下是一个简单的列表(我们大致上列出了该单元当中定义类别的一半):

- ✧ TList 是一个核心的指标列表，它通常可以改编成未确立型别的列表，通常建议改用 TList<T>，详见第 14 章。
- ✧ TInterfaceList 是一个线程安全(thread-safe)的接口列表，它实作出 IInterfaceList，值得我们细细了解(在此暂不介绍)。
- ✧ TBits 是一个非常简单的类别，用来操作在数据里面的独立位。它比较高阶，执行位操作时可以透过位移(shift)或是二元运算符号 or 跟 and。
- ✧ TPersisitent 是一个很核心的类别(TComponent 的基础类别)，我们在下一节里面来介绍。
- ✧ TCollectionItem 跟 TCollection 是用来定义集合的属性，这个属性会包含一个内容的数组。对开发组件的人员来说(当间接的使用组件时也是)，这些是很重要的类别，但对于开发应用程序的人员来说就没有这么重要了。
- ✧ TStrings 是抽象的字符串列表，TStringList 才是实际实作出基础 TStrings 的类别，TStringList 类别提供对实际字符串的储存功能。每个项目还可以再跟一个对象连接，这也是使用字符串列表的一个标准作法，透过名称/内容的字符串成对。在本章后面有一小节『使用字符串列表』会介绍更多的信息。
- ✧ TStream 是一个抽象类，用来表现任何种类的连续字节，透过连续的存取，它可以包含许多不同的储存选项(内存、档案、字符串、网络 socket，二进制长数据(BLOB)等等)。在 Classes 单元定义了许多特定的串流类别，包含 THandleStream, TFileStream, TCustomMemoryStream, TMemoryStream, TBytesStream, TStringStream 以及 TResourceStream。其他特定串流则是在不同的 RTL 单元里宣告。我们可以在本章当中的『介绍串流』看到关于串流的介绍。
- ✧ 低阶组件的串流类别，像是 TFiler, TReader, TWriter 以及 TParser，大多是开发组件的人员会使用……但不是只有他们会用到。
- ✧ TThread 类别，定义了线程类别，可以支持跟操作系统无关的多线程应用程序。也有一个为异步操作设计的类别，名为 TBaseAsyncResult。
- ✧ 实作了 observer(检查者)模式的类别(像 Visual live binding 就有使用到)，包含 TObservers, TLinkObservers 以及 TObserverMapping。
- ✧ 为了自定标注而定义的类别，像是 DefaultAttribute, NoDefaultAttribute, StoredAttribute 以及 ObservableMemberAttribute。
- ✧ 基础的 TComponent 类别，它是所有 VCL 跟 FireMonkey 当中可视与不

可视组件的基础类别，我们在本章稍后来介绍。

- ✧ 为了支持 action 跟 action 列表的类别(action 是抽象化的"命令"，会由接口元素或者内部程序发起这些命令)，包含 TBasicAction 跟 TBasicActionLink。
- ✧ 用来镶嵌非可视化组件的容器类别，TDataModule。
- ✧ 对档案存取跟串流存取动作更高阶的接口，包含 TTextReader 跟 TTextWriter，TBinaryReader 跟 TBinaryWriter，TStringReader 跟 TStringWriter，TStreamReader 跟 TStreamWriter。这些类别也在本章稍后加以介绍。

TPersistent 类别

TObject 类别有个非常重要的子类别，它是整个函式库的基础之一，名为 TPersistent。如果我们仔细观察这个类别的方法，它的重要性将会让我们非常惊讶…因为这个类别所做的事非常少。TPersistent 的关键元素之一，是它定义了一个特别的编译器选项{M+}，它的角色是启用 published 关键词，我们在第十章里面介绍过了。

Published 关键词是属性串流化的一个基础角色，这一点也从类别的名字就能看得出来了。一开始，只有从 TPersistent 类别衍生出来的类别可以把数据字段当成发布区的属性。RTTI 的延伸在较新版的 Object Pascal 中可以允许不同的模型，VCL 跟 FMX 函式库的组件串流功能还是保留了基于 published 关键词的角色以及{$M+}编译器选项。

笔记 使用当代的编译器，如果我们在一个类别里加入了 published 关键词，而这个类别不是从 TPersistent 衍生而来，也没有设定{M+}这个编译器选项，系统会先加入适当的支持，然后也提出警告讯息。

TPersistent 在整个架构中的角色到底有什么特别的?首先，它是 TComponent 类别的基础类别，我们会在下一节里介绍 TComponent。其次，它被当成属性数据型别的基础类别，所以这些属性跟它们的内部结构才得以被适当的串流化。实例就是字符串列表、位图、字型跟其他对象的表示。

跟 TPersistent 最相关的功能是它启用了 published 关键词来处理子类别的 publishec 区段，不过这个类别里面还有一些有趣的方法值得我们花时间了解一下。首先，是 Assign 方法，这个方法可以用来把一个对象的所有内容复制到另一个对象实体去(是真的把数据完全复制，不是只把参考复制过去喔)。每个有使用到这个功能的 persistent 类别都需要自行实作对应的源码(编

程语言或编译器没有聪明到自己会判定每个字段，并加以复制喔)。其次是相反的功能，AssignTo，这个方法是被保护的。这两个方法跟其他在此一类别中的方法大都都是组件开发人员会使用到，应用程序开发人员不太有机会用到。

TComponent 类别

TComponent 类别是组件函式库的基石，它通常会被 Object Pascal 编译器直接拿来使用。组件的概念基本上就是在类别中具备一些额外的设计时间的规则、特定的串流化能力(所以在设计时间的设定可以被执行中的应用程序储存、重载)，我们已经在第十章里面介绍过 PME(Property-Method -Event，属性-方法-事件)的模式。

类别中定义了一些标准的规则跟功能，并以对象所有权的概念、跨组件的通知等概念为基础，把这些规则跟功能纳入了类别的内存模型当中。我们没有打算对所有的属性跟方法作完整的分析，当然 TComponent 类别当中的一些关键功能是值得我们重点关切，因为它在 RTL 是核心中的核心。

另一个 TComponent 类别的核心功能则是它提供了一个虚拟的 Create 建构函式，提供了从类别参考建立对象的能力，同时能够呼叫该类别的特定建构函式源码。我们在第 12 章里面对此进行过介绍，但这是 Object Pascal 比较奇特的功能，值得我们了解一下。

组件所有权

所有权的机制是 TComponent 类别的一个关键元素。如果一个组件在届例的时候就已经指派了拥有它的组件(会以参数传递到它的虚拟建构函式)，这个身为所有者的组件就要负责摧毁(或说释放)其拥有的所有组件。简单的说，每一个组件都有一个参考指向它的所有者(Owner 这个属性)，同时也有一个组件列表指向它所拥有的组件(Components 数组属性)以及所拥有的组件数目(ComponentCount 这个属性)。

在预设的情形下，当我们把一个组件放在设计接口中(窗体、Frame 或是 Data Module)，就等于是为这个组件指派了所有者。当我们用源码建立组件时，该组件的所有者可以随我们指派，指派 nil 也可以(这时我们就要自己负责把组件从内存当中释放掉了)。

我们可以透过 Components 跟 ComponetCount 属性来列出一个组件(在以下这个例子里，就是 aComp)所拥有的组件，程序片段如下:

```
var
    I: Integer;
begin
    for I := 0 to AComp.ComponentCount – 1 do
      AComp.Components[I].DoSomething;
```

或使用原生的列举功能，改成这么写:

```
var
  ChildComp: TComponent;
begin
  for ChildComp in AComp do
    ChildComp.DoSomething;
```

当组件被释放时，会把该组件从该组件所有者的对象列表中删除，并把自己所拥有的所有子组件释放掉。这个机制对于 Object Pascal 的内存管理非常关键: 因为没有回收清理机制,所有权机制可以解决掉大多数的内存管理议题，我们在第 13 章里面已经介绍了其中的部分。

一如我们介绍过的,通常在一个窗体或 Data Module 里所有的对象都是以窗体或 Data Module 作为其所有者。只要我们释放了该窗体或者该 Data Module，它们所拥有的组件也会一起被释放。这就是组件从串流中建立时所发生的。

组件属性

除了核心的所有权机制(这个机制包含了通知与我们在这里没有介绍的功能)，任何组件都有两个位于发布区的属性：

- ✧ Name 是用来储存组件名称的字符串。这个属性是用来动态搜寻特定组件(呼叫组件所有者的 FindComponent 方法)并把有参照到这个组件的窗体数据字段与之链接。所有被同一个所有者所拥有的组件的名称不可以相同，但这些组件的名称可以是空白字符串。这儿有两个简单的规则:为组件设定适当的名称，让我们的源码更具可读性。另外，则是不要在执行时期变更组件的名称(除非我们完全了解到当中会有什么潜在的效果)。
- ✧ Tag 是一个原生整数数值(过去是使用整数型别)，在函式库当中没有使

用到，但我们可以透过它来把组件跟额外的信息进行链接。这个型别是在储存大小上跟指针、对象参考兼容的，所以我们可以把指针跟对象参考除存在一个组件的 Tag 属性之内。(译者: 在撰写这么久的面向对象程序的过程中，Tag 是 Object Pascal 发前人之所未见，这个属性后来也在 Objective-C 里面被广泛使用，我在 Objective-C 的程序中，最常用的动态接口程序搜寻跟处理，就是 viewWithTag，但 Objective-C 里面后来还陆续扩充了这个属性，提供了字符串型别的 Tag)

组件串流

串流机制同时在 FireMonkey 跟 VCL 平台上都被用来建立 FMX 或 DFM 档案，这两个档案就是围绕着 TComponent 类别为基础的。Delphi 串流机制会把组件跟子组件位于发布区内的属性跟事件都储存下来。储存的结果就是我们看到的 DFM 跟 FMX 档案，也是当我们把一个组件复制到文本编辑器所看到的结果。

在运行时间，有些方法可以取得相同的信息，包含 TStream 类别的 WriteComponent, ReadComponent, ReadComponentRes, WriteComponentRes，以及用来处理组件串流功能的 TReader 与 TWriter 的 ReadRootComponent 与 WriteRootComponent 方法。这些操作通常是用来处理窗体串流化时二进制数据的转换: 我们可以透过全局程序 ObjectResourceToText 把二进制窗体格式转换为文字格式，也可以透过另一个程序 ObjectTextToResource 来把文字窗体格式转换为二进制。

其中一个关键因素是，串流并不是组件在发布区当中所有的属性的完整集合。串流内容会包含：

- ✧ 该组件在发布区当中内容并非默认值的所有属性(换句话说，内容是默认值的属性就不会被储存了，这样可以节省储存空间)
- ✧ 该组件在发布区当中有被标示为 stored 的属性(不管是不是默认值，该属性都会被储存)。Stored 被设定为 False 的属性则不会被储存。
- ✧ 还有另一个机制可以加入一个额外的『假』属性，用以将之进行串流以及把他们读回来。所以我们可以在串流当中储存跟发布区属性不同的内容。

当一个属性被从串流档案重建回来的时候，会发生以下的流程：

- ✧ 组件的虚拟建构函式 Create 会被呼叫(执行适当的初始化源码)
- ✧ 会从串流中把属性跟事件加载(加载属性时，会重新把方法的名称对应到该方法在内存当中实际的名称)

✧ Loaded 虚拟方法会在加载完成时被呼叫(且组件可以进行额外的自定程序，此时属性的内容已经完成加载了)。

现代档案存取

从前代的 Pascal 语言借来，Object Pascal 仍然有此一关键词，且在核心的语言机制当中也还是可以处理档案。当 Object Pascal 发表的时候，本来已经把档案存取的功能取消，而我也不打算在本书里面介绍这个功能的。反之，我本来要在下一节里面介绍一些现代处理档案的技术的，这些技术都在 IOUtils 单元中，包含串流类别、reader 跟 writer 类别。

输入/输出工具单元

System.IOUtils 单元是最近几版才被加入到执行时期函式库的。当中定义了三个包含大多数类别方法的记录: TDirectory, TPath, 以及 TFile。

TDirectory 很显然是用来浏览文件夹、搜寻当中的档案以及子文件夹，TFile 看起来跟 TPath 没有很明显的差异。首先，TPath 是用来处理文件名跟文件夹名称的，当中有可以用来分离出磁盘代号、不包含路径的文件名、扩展名，也用来操纵 UNC 路径。而 TFile 记录则让我们检查档案的时间戳跟文件属性，也可以用来操纵档案，包含写入档案跟复制档案。

我们来看些例子吧，IOFilesInFolder 范例项目会解析出特定目录里面的所有子目录，它也会抓出该目录里面所有特定扩展名的档案。

解析子目录

这个程序会把目录里面的文件夹名称找出来，填进 ListBox 的内容，使用的是 TDirectory 记录里面的 GetDirectories 方法，把 TSearchOption.soAll-Directories 当成参数传入。回传的结果会是一个字符串数组，我们可以用列举功能来处理它：

```
procedure TFormIoFiles.BtnSubfoldersClick(Sender: TObject);
var
    PathList: TStringDynArray;
    StrPath: string;
begin
    if TDirectory.Exists (EdBaseFolder.Text) then
```

```
  begin
    ListBox1.Items.Clear;
    PathList := TDirectory.GetDirectories(EdBaseFolder.Text,
                TSearchOption.soAllDirectories, nil);
    for StrPath in PathList do
     ListBox1.Items.Add (StrPath);
  end;
end;
```

搜寻档案

这个程序的第二个按钮，让我们可以取得所有子目录当中的所有档案，透过呼叫 GetFiles，以及特定的屏蔽，我们可以扫描每一个目录。我们可以把一个匿名方法，其型别是 TFilterPredicate，当成参数传给 GetFiles 的多载版本，这样就可以有更复杂的过滤条件了。

以下的范例使用的是比较简单的屏蔽型过滤方式，会建立内部的字符串列表。这个字符串列表里面的元素接着会先被去除掉路径，指留下文件名，然后复制到用户接口。

当我们呼叫 GetDirectories 方法的时候，回传的只有子目录，不会包含当前的目录。这也是为什么程序要先搜寻当前目录，然后再对子目录逐一搜寻的原因了:

```
procedure TFormIoFiles.BtnPasFilesClick(Sender: TObject);
var
  PathList, FilesList: TStringDynArray;
  StrPath, StrFile: string;
begin
  if TDirectory.Exists (EdBaseFolder.Text) then
  begin
    // Clean up
    ListBox1.Items.Clear;
    // Search in the given folder
    FilesList := TDirectory.GetFiles (EdBaseFolder.Text, '*.pas');
    for StrFile in FilesList do
      SFilesList.Add(StrFile);
    // Search in all subfolders
    PathList := TDirectory.GetDirectories(EdBaseFolder.Text,
```

```
        TSearchOption.soAllDirectories, nil);
    for StrPath in PathList do
    begin
      FilesList := TDirectory.GetFiles (StrPath, '*.pas');
      for StrFile in FilesList do
          SFilesList.Add(StrFile);
    end;
    // Now copy the file names only (no path) to a listbox
    for StrFile in SFilesList do
      ListBox1.Items.Add (TPath.GetFileName(StrFile));
  end;
end;
```

在最后几行里，TPath 的 GetFileName 函式会被用来从完整路径中解析出文件名。TPath 记录当中包含了一些有趣的方法，包含 GetTempFileName, GetRandomFileName，一个可以用来合并路径的方法，以及一些用来检查是否包含不合法字符的方法，还有更多其他的功能。

介绍串流

如果 IOUtils 单元是用来寻找档案跟操作档案，当我们想要读写档案的时候(或者任何类似依序存取的数据结构)，我们就可以用 TStream 类别跟它的几个衍生类别。TStream 抽象类中只有几个简单的属性(Size 跟 Position)，以及一些所有抽象类共享的基本接口，包含主要的 Read 跟 Write 方法。每当我们读写一些位数据时，目前的位置就会依照读写的数量移动。对大多数的串流来说，我们可以设定让目前位置向前移动，但也有一些串流类别是只能单向移动的。

常见的串流类别

一如稍早所提到的，Classes 单元定义了几个具体的串流类别，包含以下这一些:

- ✧ THandleStream 定义了磁盘档案串流，可以透过 Handle 指向一个档案。
- ✧ TFileStream 定义了磁盘档案串流，可以透过档名指向一个档案。
- ✧ TBufferedFileStream 是优化过的磁盘档案串流，当中使用了内存做缓冲区，因此效能也得到改善。这个串流类别在 Delphi 10.1 Berlin 当中开始提供。
- ✧ TMemoryStream 定义了一个在内存当中的数据串流，我们也可以使用

指标存取。

- ✧ TBytesStream 表示在内存当中的位串流，我们也可以把它当成像是位数组来存取。
- ✧ TStringStream 跟在内存当中的字符串串流关连。
- ✧ TResourceStream 定义了一个串流，可以读取跟应用程序执行文件链接的资源数据。
- ✧ TPointerStream 透过 TStream 的接口提供了对内存中的数据区块进行存取的功能，只要指定指针的地址与内存区块大小。这个类别是在 Delphi 11 当中新增的。

使用串流

建立并使用串流就像建立一个特定型别的变量，以及呼叫组件的方法来从档案加载内容一样简单。例如，我们要把串流数据加载到 Memo 组件中，我们可以这么写:

```
aStream := TFileStream.Create (FileName, fmOpenRead);
Memo1.Lines.LoadFromStream (aStream);
```

一如从这段源码中可以看出，档案串流的 Create 方法有两个参数:文件名，以及用来指定所需档案存取模式的设定旗标。我们提到过串流支持读与写的动作，但这些相对较为低阶，所以我建议使用下一节里面我们要介绍的 reader 跟 writer 类别。

直接使用串流，提供的是全面的作业，像是在上面的源码中加载完整的串流，或者把一个串流复制到另一个:

```
procedure CopyFile (SourceName, TargetName: String);
var
   Stream1, Stream2: TFileStream;
begin
   Stream1 := TFileStream.Create (SourceName, fmOpenRead);
   try
      Stream2 := TFileStream.Create (TargetName,
       fmOpenWrite or fmCreate);
      try
           Stream2.CopyFrom (Stream1, Stream1.Size);
      finally
          Stream2.Free;
      end
```

```
  finally
    Stream1.Free;
  end
end;
```

使用 Reader 跟 Writer

从串流进行读写的好处，是使用了身为RTL 一部分的 reader 跟 writer 类别。在 Classes 单元当中一共定义了六个读写的类别:

- ✧ TStringReader 跟 TStringWriter 用来处理内存中的字符串(直接或间接的使用了 TStringBuilder 类别)
- ✧ TStreamReader 跟 TStreamWriter 用来处理一般串流(例如档案串流、内存串流等等)
- ✧ TBinaryReader 跟 TBinaryWriter 用来处理二进制数据。

这些文字 reader 当中的每一个都实作了一些基本的读取技术:

```
function Read: Integer; overload;
function ReadLine: string;
function ReadToEnd: string;
```

这些文字 writer 当中的每一个都有两组多载的处理动作，不包含(Write)刚(WriteLine)，以及每行结尾的符号，以下是第一组:

```
procedure Write(Value: Boolean); overload;
procedure Write(Value: Char); overload;
procedure Write(const Value: TCharArray); overload;
procedure Write(Value: Double); overload;
procedure Write(Value: Integer); overload;
procedure Write(Value: Int64); overload;
procedure Write(Value: TObject); overload;
procedure Write(Value: Single); overload;
procedure Write(const Value: string); overload;
procedure Write(Value: Cardinal); overload;
procedure Write(Value: UInt64); overload;
procedure Write(const Format: string;
  Args: array of const); overload;
procedure Write(Value: TCharArray;
  Index, Count: Integer); overload;
```

文字 Readers 跟 Writers

为了写入串流，TStreamWriter 类别透过文件名使用了一个串流，或建立一个串流，一个建立或增加的属性，以及 Unicode 编码作为参数。
所以我们可以像我在 ReaderWriter 范例项目里面这么写:

```
var
    Sw: TStreamWriter;
begin
     Sw := TStreamWriter.Create('test.txt',
     False, TEncoding.UTF8);
     try
         Sw.WriteLine ('Hello, world');
         Sw.WriteLine ('Have a nice day');
         Sw.WriteLine (Left);
     finally
        Sw.Free;
end;
```

要从 TStreamReader 读取数据，我们可以再对串流或对档案作一次处理(在这个例子里，我用可以透过 UTF BOM 标注来侦测编码方式):

```
var
   SR: TStreamReader;
begin
   SR := TStreamReader.Create('test.txt', True);
   try
      while not SR.EndOfStream do
         Memo1.Lines.Add (SR.ReadLine);
   finally
      SR.Free;
end;
```

留意到我们是怎么检查 EndOfStream 状态的。跟直接使用文字符串流或者字符串不同，这些类别是便于使用的，且效能很好。

二进制 Reader 与 Writer

TBinaryReader 跟 TBinaryWriter 类别是用来管理二进制数据，而非文本文件。这些类别通常是封装起串流(可能是一个档案串流或内存内容串流，内容包含网络、数据库 BLOB 字段)，并覆写了 Read 跟 Write 方法。

我们提供了 BinaryFiles 范例项目作为例子。在这个程序当中的第一个部分会写一些二进制数据到档案中(一个字段的内容与目前的时间)然后把他们读出来，指派属性的内容：

```
procedure TFormBinary.BtnWriteClick(Sender: TObject);
var
  BW: TBinaryWriter;
begin
  BW := TBinaryWriter.Create('test.data', False);
  try
    BW.Write(Left);
    BW.Write(Now);
    Log ('File size: ' + IntToStr (BW.BaseStream.Size));
  finally
    BW.Free;
  end;
end;
procedure TFormBinary.BtnReadClick(Sender: TObject);
var
  BR: TBinaryReader;
  ATime: TDateTime;
begin
  BR := TBinaryReader.Create('test.data');
  try
    Left := BR.ReadInt32;
    Log ('Left read: ' + IntToStr (Left));
    ATime := BR.ReadDouble;
    Log ('Time read: ' + TimeToStr (ATime));
  finally
    BR.Free;
  end;
end;
```

使用这些 reader 跟 writer 类别的关键规则是我们必须依照写入的顺序来读取数据，不然我们就会把数据搞乱了。事实上，只有独立字段的二进制数据被储存，该字段的信息则不会被储存。没有可以阻止我们在档案中插入资料或者 metadata，例如把下一个数据结构的大小写在实际的数据之前，或者把跟该字段相关的 token 写入。

建立字符串跟字符串列表

在介绍过档案跟串流之后，我想花一些时间来介绍处理字符串跟字符串列表方法。这是很常用的作法，而且在 RTL 当中有很多强大的功能是提供给字符串跟字符串列表使用的。我只会介绍其中一部分。

TStringBuilder 类别

我在第六章里面曾经提到过，Object Pascal 跟其他语言不一样，完整了支持字符串直接链接的功能，这个功能的效能非常好。然而 Object Pascal 的 RTL 也包含一个特别的类别，可以用来把字符串跟不同型别的数据进行重组，这个类别就是 TStringBuilder。

我们用以下的程序片段作为 TStringBuilder 类别的范例:

```
var
  SBuilder: TStringBuilder;
  Str1: string;
begin
  SBuilder := TStringBuilder.Create;
  SBuilder.Append(12);
  SBuilder.Append('hello');
  Str1 := SBuilder.ToString;
  SBuilder.Free;
end;
```

留意到我们必须建立、释放这个 TStringBuilder 对象，另一个部分则是上面的源码当中，传给 Append 方法的参数中，包含了不同型别的资料。

TStringBuilder 里面有趣的方法还包含了 AppendFormat(在对象内部呼叫 Format 函式)，以及 AppendLine(在对象内部加入了 sLineBreak)。除了 Append

之外，还有一系列的多载方法，Insert，可以用来在字符串里面插入不同型别的数据，当然还有 Remove(删除部分字符串)跟 Replace(取代字符串)等常用的方法。

笔记 TStringBuilder 类别有一个非常好的接口，提供了良好的可用性。然而从效能的观点来看，使用标准的字符串链接跟格式化函式可以提供比较好的效能，跟其他编程语言不一样的地方是，Object Pascal 的字符串是可以修改内容的，且在纯粹字符串链接时效能也都比其他语言的效能来的好上许多。

在 StringBuilder 中的连续使用方法

TStringBuilder 类别里面有个很特别的功能，是大多数的方法都会回传当时被参考到的对象。

这种程序的形式开启了连续使用方法的可能性，这意思是呼叫对象的方法，会回传前者，原本的写法是如此:

```
SBuilder.Append(12);
SBuilder.AppendLine;
SBuilder.Append('hello');
```

我们可以改写成这样:

```
SBuilder.Append(12).AppendLine.Append('hello');
```

把源码格式化一下，可以变这样:

```
SBuilder.
    Append(12).
    AppendLine.
    Append('hello');
```

我觉得这个语法会比原先的语法来的好，但我觉得这只是在语法上面稍微好一点，但大多数的人会比较偏好原始版本的写法，每一行都确实的把对象名字写出来。不管怎么写，要记得不同的 Append 呼叫并不会回传新的对象(所以不会有内存泄漏的危险)，但永远都会是用同一个对象来连续呼叫这些方法。

使用字符串列表

字符串列表是在许多视觉组件中很常用的抽象概念，也是被用来处理多行文字的一个方法。有两个类别是用来处理字符串列表的：

- ✧ TStrings 是一个抽象类，用来表现所有形式的字符串列表，不管它们是以什么形式进行储存的。这个类别定义了字符串列表的抽象概念。因此，TStrings 的对象只被用来作为组件的属性，这些属性可以用来为组件储存文字数据。
- ✧ TStringList 是 TStrings 的子类别，定义了自行建立储存空间的字符串列表，我们可以用这个类别在程序中定义一个实际用来储存数据的字符串列表。

这两个字符串列表类别也有已经定义好的方法可以用来储存内容到档案，或者从档案读取内容了，分别是 SaveToFile 跟 LoadFromFile(这两个方法都已经完全支持 Unicode 了)。要把整个列表巡回一次，我们可以简单的用 for 指令，以列表的索引值为基础进行列举，直接把该字符串列表当成一个数组，或者使用 for-in 列举都可以。

执行时期函式库是相当巨大的

我们在使用 Object Pascal 编译器的时候，其中一大部分都是由 RTL 所提供的功能，当中包含了许多在不同操作系统上面的核心开发功能。如果我们要介绍整个 RTL 的话，随便写就会跟现在这一本一样厚了。

如果我们只考虑到函式库的主要功能，算是 System 这个命名空间，会包含以下的单元(我移除了其中一些不常用到的单元):

- ✧ System.Actions 包含了对 actions 架构的核心支持，它提供了跟用户者指令链接的方法，但是已经从用户接口层次抽象化了。
- ✧ System.Character 是为了支持 Unicode 字符而设的固有型别助手(协助 Char 型别)，我们已经在第三章里面介绍过。
- ✧ System.Classes 提供了核心系统类别，也是我们在本章第一部分所介绍的单元。
- ✧ System.Contnrs 提供了旧版，非泛型的容器类别，像是组件列表，Dictionary，队列(Queue)与堆栈(Stack)。我建议，可以的话尽量使用泛型版本的相同类别。
- ✧ System.ConvUtils 提供了不同测量单位之间的转换工具。
- ✧ System.DateUtils 提供了处理日期跟时间数值的函式。
- ✧ System.Devices 提供了与系统装置的接口(像是 GPS, 重力传感器等系统内建装置)。
- ✧ System.Diagnostics 定义了一个记录结构，用以精确测量在测试程序当中所花的时间，我在本书中几乎没有提到过。

- ✧ System.Generics 有两个分开的单元，其中一个是泛型的集合类别，另一个是泛型型别。这些单元我们已经在第 14 章里面介绍过了。
- ✧ System.Hash 提供了系统对定义哈希值的支持。
- ✧ System.ImageList 包含了抽象定义，实作出跟函式库无关，用以管理图片列表，以及把单一图片当成集合的元素。
- ✧ System.IniFiles 定义了一个接口来处理 INI 配置文件案，通常只会在 Windows 平台里面找到。
- ✧ System.IOUtils 定义了文件系统存取的记录(档案、文件夹、路径)，我们在本章前面的篇幅介绍过。
- ✧ System.JSON 包含了一些用来处理跟 JavaScript 对象记录方法，或称 JSON 的核心类别。
- ✧ System.Math 定义了数学运算的函式包含三角函数跟财务函数。在同一个命名空间的其他单元中也定义了向量跟矩阵的函数。
- ✧ System.Messaging 提供了在不同操作系统上进行讯息处理的共享源码。
- ✧ System.NetEncoding 包含了处理一些常见的因特网编码法的能力，像是 base64, HTML 跟 URL。
- ✧ System.RegularExpressions 定义了对常规表示法 (Regular Expression) 的支持。
- ✧ System.Rtti 有一整套的 RTTI 类别，我们在第 16 章里面介绍过。
- ✧ System.StrUtils 提供了核心与传统字符串的处理函式。
- ✧ System.SyncObjs 定义了一些类别用来支持同步与多线程应用程序。
- ✧ System.SysUtils 有系统工具的基本集合，当中包含一些最传统的函式，以兼容于早期的编译器。
- ✧ System.Threading 包含了最新的平行程序函式库所需的接口、记录以及类别。
- ✧ System.Types 包含了一些核心的延伸数据型别，像是 TPoint, TRectangle 以及 TSize 记录，TBitConverter 类别，以及许多在 RTL 里面有使用到的基本数据型别。
- ✧ System.TypInfo 定义了旧版的 RTTI 接口，我们也在第 16 章里面介绍过，但目前已经以 System.Rtti 取代掉它了。
- ✧ System.Variants 跟 System.VarUtils 有用来处理变异型别的函式(我们在第五章里面介绍过这个功能)。
- ✧ System.Zip 提供了一个文件压缩跟解压缩的接口与函式库。

RTL 当中还有许多部分是 System 命名空间的成员，因为每一部分都包含许多单元(通常当中的单元都很多，像是 System.Win 命名空间)包含了 HTTP 客户端(System.Net)，以及对物联网的支持(System.Beacon, System.Bluetooth,

System.Sensors 以及 System.Tether)。当然当中也有从支持的操作系统当中转译进来的接口跟 API 宣告。

再强调一次，直接使用 RTL 函式、型别、记录、接口，以及记录，可以让我们的程序立刻获得许多福利，这是我们需要去发现的，这样可以让我们得以发挥 Object Pascal 的强大功能。花一些时间，浏览一下系统文件吧，我们可以从中获益更多。

写在最后

第十八章是本书的结尾，留给接下来的附录。这是我之前第一本完全把重点集中在 Object Pascal 编程语言的书，我也尽了最大的努力更新当中的数据，并让本书的内容跟范例维持在最新的状态。之前我制作过一个只更新给 Delphi 10.1 Berlin 的 PDF 版本，目前您正在阅读的则是为 Delphi 10.4 Sydney 更新的版本。

再提一次，在一开始我们提到如何从 GitHub 取得本书的范例原始码(基于与 10.4 版相同的档案库)，您也可以随时关注我的网站以获得最新的信息跟程序的更新。

我希望您能享受阅读本书的过程，就跟我一样享受写作的过程，一如过去 25 年我用 Object Pascal 写作的时光。

end.

本书的最后一节，提供了一些附录，我们介绍一些特别值得一提的相关问题，但不适合在本书各个章节里面介绍的。包含 Pascal 跟 Object Pascal 的小历史，词汇表。

附录一览

附录 A: Object Pascal 的演进
附录 B:词汇表

a: Object Pascal 的演进

Object Pascal 是为不断延伸使用范围的计算装置而建置的编程语言，从智能型手机、平板、到桌面计算机以及服务器。它并不局限在特定的应用情境中。它是被以坚固的基础精心设计，为现代程序设计人员所锻造的工具。它提供了在开发速度与开发结果执行，以及在语法的清楚和表达式的功能几乎最为理想的平衡。

Object Pascal 的基础，是建立在 Pascal 家族的编程语言之上。就像 Google 的 Go 语言，或者 Apple 的 Objective-C 语言是从 C 语言衍生而来的一样。Object Pascal 是从 Pascal 产生出来的。我们从名字就可以猜的到了。

这个简单的附录包含了对这个编程语言家族的历史跟工具做了简短的摘要，包含 Pascal，Turbo Pascal，Delphi 的 Pascal，跟 Object Pascal。要学习这个编程语言虽然不用阅读历史，不过了解历史，也能了解这个语言演化至今这个态势的原因。

我们现在在 Embarcadero 开发工具里面所使用的 Object Pascal 编程语言，是在 Borland 于 1995 年发表的 Delphi 当中一起发明出来的，Delphi 是当时最新的视觉化开发环境。第一个 Object Pascal 语言是从已经在 Turbo Pascal 产品使用的语言版本延伸发展出来的，当时的 Pascal 语言就通常都被称为 Turbo Pascal。Borland 并没有发明 Pascal，而是协助 Pascal 变得更普及，并且扩展它的基础，克服许多原本跟 C 语言相比之下的一些局限。

以下的几个小节，涵盖了从 Wirth 发明出 Pascal 到最新版的 LLVM 基础的 Delphi 版 Object Pascal 为了 ARM 芯片与行动装置版制作的编译器。

Wirth 的 Pascal

Pascal 编程语言最初是在 1971 年由 Niklaus Wirth 设计出来的，他是瑞士苏黎士理工学院的教授。Wirth 教授最完整的传记，可以从下面这个网址看到: http://www.cs.inf.ethz.ch/~wirth。

Pascal 的设计，是为了教育用途，简化 Algol 语言。Algol 语言则是在 1960 年被创造出来的。当 Pascal 被发明出来时，同一时代就有许多编程语言并立，但只有其中几个有被广泛使用: FORTRAN、汇编语言、COBOL 与 BASIC。Pascal 的主要要求，是简洁、以强化型别的概念达到管理的要求、变量的宣告，以及结构化的程控结构。这个编程语言也是设计成为教育的工具，在接下来的十多年间，也的确成为学习程序设计最好的工具。

不用说，Wirth 的 Pascal 的核心概念对所有编程语言的历史都产生了重大的影响，远远的超越了原有的 Pascal 语法。作为教育用的编程语言，学校与大学往往都遵循其他标准(像是从工具提供者所提供的工作机会或捐赠)，而不是看哪个语言可以对学习编程语言的关键概念更有帮助。不过这已经是另一个故事了。

Turbo Pascal

Borland 闻名全球的 Pascal 编译器，称为 Turbo Pascal，是在 1983 发表的，是实作了 Jensen 和 Wirth 的”Pascal 用户指南和报告”而成的。Turbo Pascal 编译器已经是跨越时代最畅销系列的编译器之一，并且让 Pascal 语言在 PC 平台特别普及，感谢它在简单与强大之间的平衡。Turbo Pascal 最原始的作者是 Anders Hejlsberg，后来他也为微软创造了 C#语言。

Turbo Pascal 是以整合开发环境(IDE)著称，我们可以在这个环境中编辑源码(透过跟 WordStar 兼容的编辑程序)，执行编译器，看到程序的错误、并跳到出错的源码。现在听起来，这都很平常，但是在 1980 年代，我们得编辑好程序、跳出编辑程序，回到 DOS，用命令行执行编译器，写下哪一行有错，然后再开启编辑器去修改源码。

此外，Borland 把 Turbo Pascal 只卖 49 块美金一份，当时微软的 Pascal 编译器一份要卖数百块美金。Turbo Pascal 数年的热销，最后让微软放弃了它自己的 Pascal 编译器产品。

我们现在还是可以从 Embarcadero 的博物馆区下载 Borland 最初的 Turbo Pascal 编译器:

```
http://edn.embarcadero.com/museum/
```

笔记 在最初的 Pascal 语言之后，Nicklaus Wirth 设定了 Modula-2 语言，是以现在大多已经被遗忘的 Pascal 语法进行延伸的，在这个新的语言里面，发表

了模块化的概念，和早期的 Turbo Pascal 以及今日的 Object Pascal 在概念上非常相似。

Modula-2 的延伸版本还有 Modula-3，当中就有面向对象的概念了，和 Object Pascal 非常类似。Modula-3 的使用者比 Modula-2 的使用者更少，因为当时的程序人员多已经投向商业化的 Pascal 编译器和 Apple 的编译器了，直到 Apple 放弃 Object Pascal，转而支持 Objective-C，Borland 就成了这个编程语言独撑大局的角色。

早期的 Delphi 的 Object Pascal

历经九个版本的世代交替，从 Turbo Pascal 到 Borland Pascal 编译器，Pascal 终于从模块化编程语言蜕变进入到面向对象程序设计(OOP)的领域，Borland 在 1995 年发表了 Delphi，把 Pascal 引领朝向可视化编程语言的方向走去。Delphi 在好几个面向上延伸了 Pascal 语言，包含许多面向对象的延伸模块，这些跟 Object Pascal 的方向是不同的，包含 Borland 的 Pascal with Objects 编译器(这是 Turbo Pascal 最后的身影)。

笔记

您可以在网页上读到更多关于 Delphi 产品化的过程，网址是:https://delphi.embarcadero.com/
跟 https://www.marcocantu.com/delphibirth，在我每年大约 2 月 14 日所写的部落格文章里，也有一系列庆祝这个产品周年的内容

历史

1995 年真的是编程语言发展史上非常特别的一年，在同一年里面，同时有 Delphi 的 Object Pascal，Java，JavaScript 跟 PHP 登场了。这些都是目前全世界最普及的编程语言。事实上，大多数其他普及的语言(C，C++，Objective-C 跟 COBOL)也都已经历史悠久了，相对比较年轻的编程语言只剩下了 C#。我们可以从这个网址得到更多关于编程语言历史的资料:
http://en.wikipedia.org/wiki/History_of_programming_languages.

在 Delphi 2，Borland 就把 Pascal 编译器带到了 32 位的世界，也实际的把它重新设计后，把原本在 C++编译器中常用的一些作法带入了 Delphi。这也把许多本来只能在 C/C++编译器中独有的优化程序带入了 Pascal 编程语言。在 Delphi 3，Borland 在语言中加入了接口的概念，在类别跟他们的关系之间有了飞跃的进展。

在释出七个版本的 Delphi 之后，Borland 终于正式把 Object Pascal 语言称为 Delphi 语言，但并没有在任何时候修改编程语言。在当时，Borland 也创造了 Kylix，这是 Linux 版的 Delphi，后来又创造了 Delphi 的.NET 版本(该产品就是 Delphi 8)。这两个项目随后就被终止了，但 Delphi 8(最后在 2003 才发表)对编程语言本身做了很大幅度的修改，这些修改后来也被 Win32 版本的 Delphi 编译器跟所有后来版本的编译成所采纳。

Object Pascal 从 CodeGear 到 Embarcadero

随着 Borland 对于在开发工具上投资的迟疑，后续的版本，像是 Delphi 2007，就是以 CodeGear 的名义发表的，CodeGear 当时已经是 Borland 最主要的子公司了。这个子公司(或说是事业单位)，后来被卖给了 Embarcadero。在 Delphi 2007 发布之后，该公司的重心又回到了发展、延伸 Object Pascal 编程语言，在上面加入许多让全球盼望已久的功能，例如对 Unicode 的支持(Delphi 2009)，泛型，匿名方法(或称程序区块)，延伸的执行时期型别信息(或称镜射)，以及很多其他主要的编程语言功能(如本书第三部分所介绍的)。

同一时间，除了 Win32 编译器之外，该公司也发表了 Win64 的编译器(Delphi XE2)以及 Mac OSX 的编译器，渐渐走回多平台策略，这是继早期以 Kylix 企图跨足到 Linux 平台之后的另一次尝试。然而，这时候的版本已经成为了在单一 Windows 环境开发，并可以编译到其他平台上的功能支持了。对 Mac 的支持是该公司多装置策略的第一步，并同时涵盖桌上与行动平台，像是 iOS 跟 Android。这个策略也让新的图形接口架构应运而生，这个新的架构就称为 FireMonkey。

朝向行动化

朝向行动装置，以及 Object Pascal 对 ARM 芯片版本的编译器(所有之前版本的 Delphi 都只支持 Intel x86 的芯片)已经演变成为整体重新以开放式 LLVM 编译器架构来拟定编译器架构与相关工具了。

笔记 LLVM 是 LLVM 编译器架构的简写，或者『模块与可重复使用的工具组科技的集合』，您可以从 https://llvm.org/ 网站读到更详细的内容。

为支持 iOS 提出的 ARM 编译器在 Delphi XE4 的时候发表，当时是第一个以 LLVM 为基础的 Object Pascal 编译器，但也是第一次发表了像是自动参考计数(Automatic Reference Counting，简称 ARC)的功能。

稍后在同一年(2013)，Delphi XE5 加入了对 Android 平台的支持，它也是第二个以 LLVM 为基础的 ARM 编译器。总结一下，Delphi XE5 里面包含了 Object Pascal 语言的六种编译器(Win32, Win64, Mac OS X, iOS 仿真器，iOS ARM，以及 Android ARM)。这些编译器都支持单一语言的共通定义，当中的差异微乎其微，我已经在前面的篇幅加以介绍过了。

在 2014 年的前几个月，Embarcadero 发布了一个新的开发工具，以相同的核心行动科技为基础，称为 Appmethod。在 2014 年四月，Embarcadero 发布了 Delphi XE6，2014 年九月则发布了 Appmethod 与 Delphi XE7，随即在 2015 年春天发布 Delphi XE8，当中就包含为了支持 iOS 而发展的 ARM 64 位编译器。

Delphi 10.x 时期

在 Delphi 10 Seattle 之后，随着被 Idera 并购，Embarcadero 推出了 10.x 系列: Delphi 10.1 Berlin, Delphi 10.2 Tokyo, Delphi 10.3 Rio, 以及 Delphi 10.4 Sydney。在这些版本中，Embarcadero 加入了对新平台与新操作系统的支持: Linux 64 位, Android 64 位与 macOS 64 位。同时也为推出新的 VCL 来支持 Windows 10 操作系统，让焦点也回到 Windows 系统上。

在 10.x 的几个版本中，Embarcadero 持续让 Object Pascal 进化，加入了像是行内变量宣告、自定受管理的纪录，以及几个功能的强化，在本书中都有介绍到。

Delphi 11 发行

随着微软推出 Windows 11、苹果也把 macOS 从 10.x 的版号向前推进之后，Embarcadero 决定把版号从 10.x 系列向前推进，从这一版开始，让主版号改为 11 产品名为 Alexandria，即使目前 Delphi 已经能够进行跨装置的程序开发，Delphi 还是以 11 为版号，也宣示跟 Windows 操作系统有更深的关联。在 Delphi 11 当中，值得记上一笔的是开始支持 macOS ARM-64 版本的

操作系统，可以直接生成支持 Apple M1 CPU 的原生码，以及在 IDE 中可以支持 High-DPI (高分辨率)接口。

b: 词汇表

A

Abstract Class	抽象类	是没有被完整实作，只在接口区段宣告方法的类别，子类别则需要进行完整的实作。
Ambiguous call	不明确的呼叫	这个错误讯息会发生在编译器无法判别我们的源码到底要呼叫哪一个函式的时候。
Android	安卓系统	Google 为移动装置开发的操作系统，有数以百计的硬件制造厂商用以制造装置。 Andorid 是目前世界上最多装置使用的操作系统，已超越了微软窗口系统。
Anonymous Method	匿名方法	匿名方法，或者匿名函式是没有跟任何函式名称关连的函式。匿名方法可以被指派到一个变量，也可以当成参数传递给另一个函式，让另一个函式稍后可以执行它。我们可以把匿名方法想成是跟传统函式不同的一个小魔法。而它的确也是个魔法。这个实际存在的魔法是可以从区块中存取变量，即使程序已经执行到别的程序区块也可以喔。
API	应用程序编程接口	API 是由软件(像是操作系统)所提供，让应用程序开发人员可以用来制作某些功能。例如，当应用程序在屏幕上面显示一行文字的时候，通常他就是呼叫了在计算机中图形接口的函式。这一类由计算机的图形接口所提供的函式称为图形接口的 API。

		通常当软件为编程语言提供 API，该软件就是把一些功能写在里头了。举例来说，微软窗口操作系统提供了一个 API 给 C 跟 C++语言使用 笔记 Object Pascal 的 Windows 单元文件提供了 Object Pascal 语言呼叫微软窗口系统的功能，平息了直接呼叫以 C 或 C++写成的函式的争论。

B

Boolean Expression	布尔表达式	布尔表达式，就是结果会是 True 或 False 的表达式，例如(1=2)，这个表达式的结果就会是 False。布尔表达式的内容不用非得是传统的数学运算内容，它可以是一个单纯的布尔型别的变量，甚至也可以是一个回传布尔值的函数。

C

Cardinal	长整数	Cardinal 是 Object Pascal 里面用来表示自然数的一种型别，可以储存 0 或大于 0 的整数，其范围是 0 到 2^{32}-1。
Class	类别	类别是对象的定义，对象可以用有方法(method)、属性(Property)与数据字段(data field)，类别纯粹只是定义，并不能被当成变量使用。 笔记：并不是所有面向对象编程语言都需要先定义类别才能产生对象。像是 JavaScript、IO 跟 Rebol 这三种语言的对象，都可以直接定义，不用先定义类别。 笔记：记录的定义跟类别的定义方

		式，在 Object Pascal 里面是很像的。在记录里面也可以拥有函式，使用时必须完整的把记录名称与函式名称完整写上，这个特性与类别里面的方法几乎是完全相同的。
Code Point	字码	字码就是在 Unicode 里面用来表示每个字符的数字表示值。每个全世界文字的字符、符号都有一个 Unicode 的字码用以表示。
Compiler Directive	编译器设定	编译器设定是编译器的特殊指令，编译器有其标准规则。所有的编译器设定会是一个特别的字，前面以$符号开头，也可以透过项目选项来设定。
Components	组件	组件是预先建置、已经可以使用的程序对象，我们可以很容易的把应用程序跟组件组合起来，节省下来的开发时间，可是很惊人的。
COM	组件对象模型	COM 是微软窗口架构的一部分。
Control	控制组件	控制组件是图形接口(GUI)的一部分，例如按钮、文字输入框、图片容器等等，控制组件通常也是视觉组件。
CPU	中央处理单元	CPU 是 Central Processing Unit 的缩写，它是所有计算器的核心，也是实际上执行程序的零件。Object Pascal 的程序指令需要被转换成 CPU 能够理解的汇编语言。记住，我们在侦错程序中也有一个 CPU View 的窗口，它可不是新功能。CPU 通常会跟 FPU(浮点数处理单元)一起运作。

D

Data Type	资料型别	数据型别是数据的专属类型，例如整数。在 Object Pascal 里面，变量跟内容(Value)都会拥有数据。

		笔记：数据实际上是存在可执行程序当中，以二进制形式储存的，实际的储存方法则依照其数据型别而定。
Design Patterns	设计模式	从不同开发人员用来处理不同问题的软件架构来看，我们会注意到当中对共通的问题点处理上会有相似之处。设计模式就是对于共通的设计方式，只是大家给这些设计方式一个共通的名字。并对一些不同的情形整理出一些抽象的规则。软件界的设计模式运动，始于 1994 年，由 Erich Gamma, Richard Helm, Ralph Johnson 与 John Vlissides 所合着的”Design Patterns, Elements of Reusable Object-Oriented Software”一书 (Addison-Wesley, 1994, ISBN:0-201-633612)。作者通常会被简写成 Gamma 等人，或者 4 人帮 (Gang of Four)或 GoF。这本书也常被简称为 GoF book。 在该书中，作者描述了设计模式的概念，精确的指出了描述这些模式的方法，并且提出了 23 种模式，分为三大群组:生成模式、结构模式、行为模式。在台湾由台湾培生教育出版股份有限公司出版，叶秉哲译。
DLL	动态链接函式库	动态链接函式库是集结了许多没有被包含在应用程序执行档案的韩式。当应用程序执行的时候，应用程序会把函式库加载到内存当中，之后才能呼叫被包含在该函式库里面的函式。这些函式库通常是被设计给很多不同的应用程序使用的。在窗口程序系统之外，动态链接函

		式库则被称为共享对象(Shared Object，简称 SO 檔)

E

Event	事件	类别中一种特别的属性，可以用来链接对象中一个特别的”行为”，当特定的”事件”发生时，就可以执行被连结的行为。事件已经成为快速开发模式的一部分。

F

FireMonkey	FireMonkey	FireMonkey(现在官方也称之为 FM 平台)或 FMX，是一组视觉与非可视化组件，这个平台同时支持 Appmethod 和 Delphi，整组平台的组件是跨平台的，所以这些组件可以同时在 Windows, Android, OS X, iOS 这几个平台上面以同样的方式运作。
Form	窗体	窗体是在 VCL 跟 FireMonkey 平台上的窗口。
File System	文件系统	文件系统是计算机操作系统的一部分，这个系统会组织数据储存在计算机上的方式，并管理数据储存与取得的方法。
FPU	浮点数运算单元	FPU 是 Floating Point Unit 的缩写，这个单元的功能是纯粹在处理浮点数的运算，就像 CPU 纯粹在执行指令一样，由于浮点数运算由 FPU 专们执行，也就使 CPU 的执行速度能够得到大幅提升。
Function	函式	函式是源码的区块，在 Object Pascal 里面的函式会回传执行的结果，呼叫函式时，可以依照预先定义好的型别来进行参数的传递。

G

Global Memory	全局内存	全局内存是一个静态的内存区域，

		供应用程序的全局变量使用。这个区域的内存的生命周期是从应用程序从启动到结束为止，且这个区域无法扩充(跟 Heap 内存，以及动态配置内存相较)。全局内存在 Object Pascal 里面的使用是相当保守的。
GUI	用户图形接口	用户图形接口可以让用户在计算机、平板以及手机上透过图形化的图标以及其他可视化的指示讯息进行操作。大多数用户是透过使用鼠标或手指(或其他指针设备)来执行点击、触碰、按压、手指滑动以及其他手势进行互动操作的。

H

Heap Memory	Heap (有人翻成『堆』，但我习惯不翻译，直接念厂丨 P)	Heap 是让动态配置动作使用的内存区块，就如 Heap 这个英文字的字面意思，在这里面的内存没有结构或顺序。每当一个区块被需要的时候，系统就会从这一块内存里面找出还没有被使用到的区块提供源码使用。每个独立的区块的生命周期都不同，而配置与释放的动作之间也没有绝对必然成对的顺序。Heap 可以用来储存对象的数据、字符串、动态数组或其他参考型别(请见 Reference)，以及其他手动配置的区块(请见 Pointers)。Heap 的区块很大，但并不是无限的，且如果我们没有把用过的对象从内存里面释放掉，应用程序最后就会把所有内存都用光。

I

IDE	整合开发环境	IDE 是 Integrated Development Environment 的缩写，也就是整合开发环境的意思。通常是在一个应用程序中，提供给开发人员许多开发

		跟侦错的功能，让开发的过程能提高生产力。IDE 的最低限度，至少要提供一个源码编辑器、自动化编译工具以及侦错程序。IDE 的现代含义可以说是由 Borland Turbo Pascal 所发明的，它也是目前 Embarcadero 所提供的 Object Pascal IDE 的前身。 Object Pascal IDE 支持 Delphi 和 Appmethod，IDE 的功能非常丰富，并包含了图形接口设计、源码模版、源码重购(refactoring)以及整合的单元测试功能等。
Type Inheritance	型别继承	型别继承是面向对象程序设计的核心概念之一。原始的概念是让一个数据型别能够从一个既存的数据型别进行延伸，并为既存的数据型别进行功能扩充。这样的扩充就被称为型别继承，也常和基础类别、前代类别，以及祖先类别、子类别等名词一起被提及。
Interface	接口	通常是参考到一个软件模块能够具备的功能所进行的抽象宣告。在 Object Pascal 里面，接口是纯粹的抽象类定义(当中只宣告方法，不含到任何数据的定义)，像是 C#或 Java，请参考本书第 11 章。 然而，Object Pascal 也仍旧在单元文件里面引入了接口的概念，在单元文件的 interface 区段所宣告的类别跟方法可以被其他使用这个单元文件的源码所看到，interface 这个关键词也会在单元文件里面被用上。
iOS	iOS (不翻译)	这是由 Apple 提供的操作系统，在 Apple 的 iPod, iPhone, iPad, Apple

		TV 上面都有使用到。

M

macOS	macOS	苹果计算机的操作系统，之前被称为 OSX
Method	方法	方法就是与对象绑定的方法或程序。方法可以存取该对象当中的所有数据。

O

Object	物件	对象是许多数据项(属性或数据字段)与源码(方法)的组合。对象是类别的实体，类别则是对象的定义。
OOP	面向对象式程序设计	面向对象式程序设计是 Object Pascal 背后的概念性结构，奠基于类别、继承、多型(polymorphism)等概念。现代的 Object Pascal 也支持其他程序的概念，感谢泛型、匿名方法与镜设(reflection)。
Ordinal Type	有序型别	有序型别是指该型别面的的数据是有顺序概念的。我们可以想到整数是有顺序的，字符也是，甚至我们也可以自定列举型别。

P

Pointer	指标	指针是直接储存内存地址的一个变量。指针可以储存一个数据或者方法的地址。指标并不常用到，因为参考渐渐无法被直接使用，而受管理的指标也渐渐常被用到了，但指标仍旧很容易使用。
Polymorphism	多型	多型是指呼叫一个方法的时候，可以使用不同的形式加以呼叫(例如在不同的操作系统中)，端看该对象是否有提供此方法，这是 OOP 编程语言的标准特点之一。
Procedure	程序	程序跟函式一样，都是源码区块，

		在其他程序中也常被称为子程序，跟函式不同的地方，是程序不会回传任何执行结果。
Project Option	项目设定	一整组针对应用程序项目的设定值，也会影响编译器跟链接程序的行为。
Property	属性	属性是对象的抽象数据项，它可以直接对应数据字段，也可以透过方法来对属性的内容进行读写。

R

RAD	快速应用程序开发	快速应用程序开发，是开发环境的一个特征，透过这样的工具，会让我们开发应用程序的时候更快、更容易。RAD 工具通常具备视觉设计程序，虽然视觉设计工具已经相对的比较旧，且今日不一定还常被使用到。
Record	记录	简单的纪录就是许多数据项的集合，储存的方式具备结构化的特征。记录可以定义成型别，依序来显示记录中单独的数据项。 Object Pascal 也提供了进阶的纪录功能，让记录像对象一样可以包含方法。
Recursion	递归	递归或者递归调用是子程序写作的一种方式，让一个子程序可以呼叫自己，或者让两个子程序可以彼此呼叫。递归调用通常是循环的替代方案。
Reference	参考	参考是一个变量，可以直接餐靠内存里面的某些数据，而不直接储存这些数据。在 Object Pascal 中，某些型别，像是类别或字符串，以及接口、动态数组，都是参考。跟指标(请见 Pointer)不同的是，参考通常是由编译器跟运行时间函式库管

		理的，开发人员也需要一些低阶的相关知识跟直接存取内存的相关知识。
RTTI (或 Reflection)	执行时期型别信息，还是简称 RTTI	Run-Time Type Information 的缩写，是在实际的应用程序中存取型别信息的能力(这能力传统上只有编译器才拥有)。其他的程序开发环境则把这个功能称之为镜设(Reflection)
Run-Time Library (RTL)	执行时期函式库，仍简称 RTL	预先写好的源码的集合，编译器会自动把这些源码跟应用程序组合成执行文件。他包含了很多基础的处理，尤其当成是执行时需要与操作系统进行互动的程序(例如配置内存、读写数据、与文件系统的互动等)

S

SDK	软件开发工具包	软件开发工具包是一整组的软件工具，透过 SDK，我们可以在特定的环境中提供软件功能。一个很直觉的例子，是 Android SDK 提供了软件 API 函式库与开发工具，我们可以透过它来为 Android 应用程序提供建立、测试、侦错等程序。 笔记：Object Pascal 跟 FireMonkey 平台提供了许多单元文件，让我们可以透过 Android 函式库使用许多功能，而 IDE 则处理建置、测试、侦错等许多功能。
Search Path	搜寻路径	编译器会在该路径，以及其下的所有子目录寻找程序所使用到的外部单元文件。
Stack (memory)	内存堆栈	内存堆栈是动态、依序配置的内存区域。每当我们呼叫一个方法、程序或函式的时候，内存堆栈就会把它自己的内存区域(包含局部变量、

		暂时变量与参数等)进行保留。当子程序结束，这些内存就会被清除，这个作法非常有规范。会把内存堆栈用光的唯一情境，是一个方法进入了无穷递归的情形。 笔记 在大多数的情形下，在堆栈中配置的局部变量都没有被初始化成0：我们应该在使用前先对这些动态配置的变量进行初始化。

U

Unicode	万国码(本书中我仍称之为Unicode)	Unicode 已经是用来以二进制(一堆0与1的组合)标示每个独立文字的标准方法。文字在不同的程序之间进行交换的时候因此提升了可靠度。这个标准当中涵盖了超过110,000个不同的字符，包含了100种以上的手写文字。

V

VCL	视觉组件函式库	VCL是Visual Component Library的缩写，也就是视觉组件函式库的意思，这个架构支持 Delphi，也在Delphi 当中被大量使用。当中的用户图形接口组件都是原生的Windows用户图形接口组件。
Virtual Methods	虚拟方法	虚拟方法是当我们在宣告类别的时候，宣告的函式或者程序，这些含识货程序可以被该类别的子类别加以覆写(override)。基础类别中也可以包含该方法的一些实作，这些实作的源码也会在子类别当中被使用到。如果基础类别没有实作该方法的预设版本，任何子类别就都必须要定义虚拟方法的实作。

W

Window	窗口	窗口指的是屏幕上包含有用户图形接口元素的区域，用户可以跟这些元素互动。用户图形接口应用程序可以拥有多重窗口。在 VCL 跟 FireMonkey 当中，窗口就是透过窗体来定义的。
Windows	窗口系统	这是微软的操作系统名称，它是图形窗口概念的先驱(当时还有 Apple 的 Mac 操作系统，以及其他几个图形化接口的操作系统)之一。