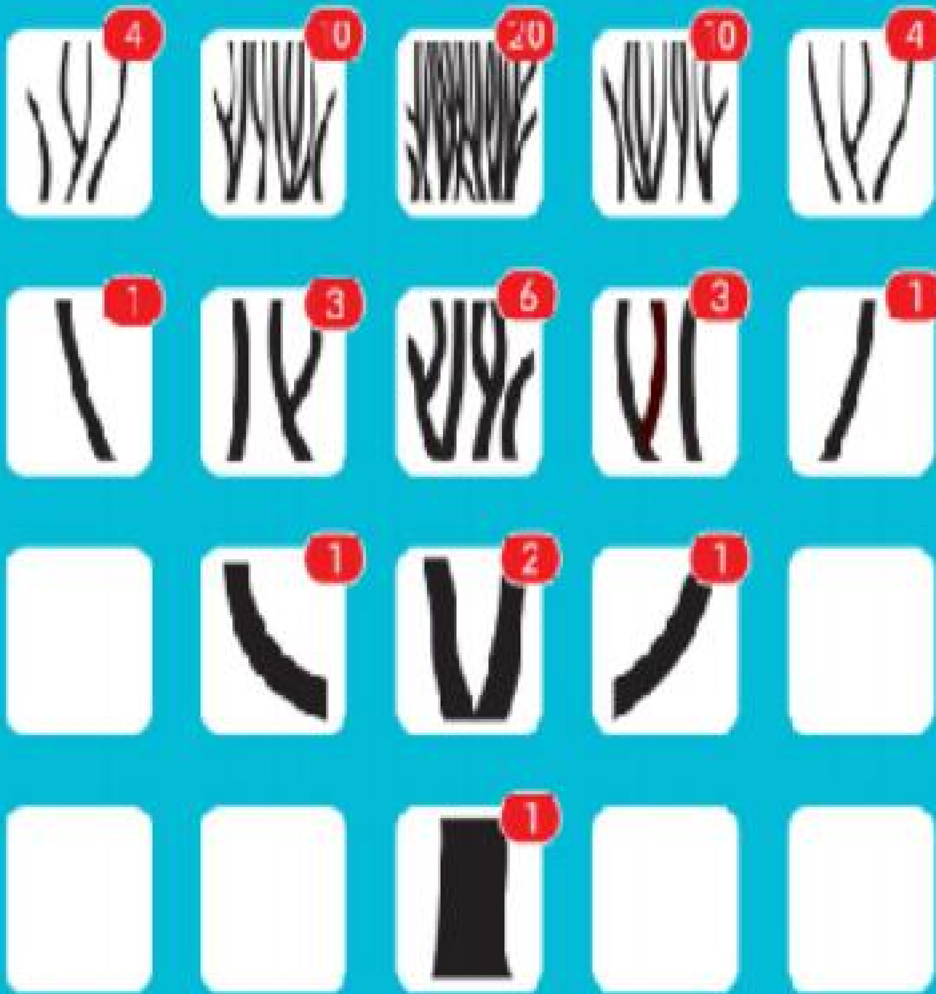

MARCO CANTÙ

OBJECT PASCAL HANDBOOK

DELPHI 10.4 SYDNEY EDITION

Marco Cantù

Object Pascal Handbook
Delphi 10.4 Sydney Edition

Delphi 10.4 Sydney 的 Object Pascal

编程语言完整指南

原版：皮亚琴察（意大利），**2015** 年 **7** 月

Delphi 10.4 版：**Piacenza**（意大利），**2020** 年 **11** 月草案

翻译：**pole**

2020.12.20

作者：MarcoCantù
发行人：MarcoCantù
第一版编辑：Peter W A Wood
封面设计师：Fabrizio Schiavi（www.fsd.it）

版权所有 1995-2020 MarcoCantù，意大利皮亚琴察。保留世界权利。

作者在此出版物中明确创建了示例代码，供读者免费使用。
本书的源代码是受版权保护的免费软件，可通过该书和该书网站上列出的 GitHub 项目分发。版权防止您未经许可而在印刷或电子媒体中重新发布代码。只要没有将代码本身作为独立产品进行分发，出售或商业开发，就授予读者在其应用程序中使用此代码的有限权限。
除了与源代码有关的特定例外，本出版物的任何部分均不得以原始或翻译语言（包括但不限于影印，照片，磁性，或其他记录，未经出版商事先同意和书面许可）。
Delphi 是 Embarcadero Technologies（Idera，Inc.的子公司）的商标。 其他商标是各自所有者的权利，如本文中所述。尽管作者和出版商已尽最大努力编写本书，但他们对本文内容的完整性或准确性不做任何形式的陈述或保证，也不承担任何形式的责任，包括但不限于性能，适销性，是否适合任何特定目的，或直接或间接地导致或间接导致此书的任何形式的损失或损害。

Object Pascal Handbook for Delphi 10.4 Sydney
此 PDF 版本是该书的 2020 年 11 月 30 日草案

ISBN-10：待分配
ISBN-13：待分配

本书的电子版已获得 EmbarcaderoTechnologiesInc.的许可。作者也直接将其出售。未经许可，请勿分发本书的 PDF 版本。
印刷版通过 KindleDirectPublishing 出版，并在多个商店销售。

有关更多信息，请访问 **http://www.marcocantu.com/objectpascal**

Begin

对于我的家人 Raffaella，Benny 和 Jacopo，
以我的全部爱心，并非常感谢您为使
我的生活超出我的期望所做的一切

强大，简单，表现力和可读性强，对于学习和专业发展均是如此，这些都是当今的 Object Pascal 的一些特征，Object Pascal 是一门源远流长的语言，活泼的现在，以及灿烂的未来。

Object Pascal 是一种多方面的语言。它结合了面向对象编程的功能，对通用编程的高级支持以及诸如属性之类的动态构造，但是却没有删除对更传统风格的过程编程的支持。一种适用于所有行业的工具，其编译器和开发工具已涵盖了移动时代。一种为未来做好准备的语言，但扎根于过去。

Object Pascal 语言有什么用？从编写桌面应用程序到客户端服务器应用程序，从大型 Web 服务器模块到中间件，从办公自动化到用于最新手机和平板电脑的应用程序，从工业自动化系统到 Internet 虚拟电话网络……这不是该语言可以用于现实世界中，但目前的用途是什么。

今天我们使用的 Object Pascal 语言的核心来自 1995 年的定义，这是编程语言的辉煌之年，因为这也是 Java 和 JavaScript 发明的一年。虽然该语言的起源可以追溯到其 Pascal 祖先，但它的发展并没有在 1995 年停止，直到今天，核心增强功能仍在继续，由 Embarcadero Technologies 构建并在 Delphi 和 RAD Studio 中找到的桌面和移动编译器。

一本关于当今语言的书

鉴于该语言的角色变化，其多年来的扩展以及它现在吸引着新开发人员的事实，我认为写一本书以全面覆盖当今的 Object Pascal 语言非常重要。目的是为新开发人员，其他相似语言的开发人员提供语言手册，也为希望了解更多有关最新语言更改的不同 Pascal 方言的旧计时器提供语言手册。

新手当然需要一些基础，但是鉴于变化无处不在，即使是老手也将从最初的章节中找到新的东西。

除了涵盖 Object Pascal 语言的简短历史的简短附录之外，本书的编写还涵盖了当今的语言。自从 Delphi 的早期版本（1995 年现代 Object Pascal 的首次实现）以来，该语言的大部分核心功能都没有发生太大变化。

正如我将在整本书中所暗示的那样，这些年来，该语言一直没有停滞不前，它的发展速度非常快。在我过去写过的其他书籍中，我遵循了更按时间顺序的方法，首先涵盖了经典的 Pascal，随后随着时间的推移或多或少地遵循了扩展。但是，在本书中，其想法是使用一种更合乎逻辑的方法，逐步讨论各个主题，并介绍当今语言的工作方式以及如何最好地使用它，而不是随着时间的推移而发展。

例如，可以追溯到原始 Pascal 语言的本机数据类型具有最近引入的类似于方

法的功能（由于内在类型帮助程序）。因此，在第二章中，我将介绍如何使用此功能，尽管直到您了解如何自己制作此类自定义类型扩展之后，它才会出现。

换句话说，这本书涵盖了当今的对象帕斯卡语言，只是从非常有限的历史角度对它进行了全面的教学。即使您过去曾经使用过该语言，您可能想略过整个文本以寻找更新的功能，而不是只关注最后几章。

通过实践学习

该书的目的是解释核心概念，并立即提供简短的演示，鼓励读者尝试执行，试验和扩展以理解概念并更好地吸收它们。这本书不是参考手册，它解释了该语言在理论上应该做什么，并列出了所有可能的极端情况。在尝试精确的同时，重点更多地放在提供实用的分步指南的语言教学上。示例通常非常简单，因为目标是使它们一次专注于一项功能。

完整的源代码可在 GitHub 上的在线代码存储库中找到。 您可以将其作为单个文件下载，克隆存储库，或者仅在线浏览并仅下载特定项目的代码。如果您是从存储库中获得的，则在我发布任何更改或其他演示的情况下，您可以轻松地更新代码。GitHub 上的位置是：

https://github.com/MarcoDelphiBooks/ObjectPascalHandbook104

要编译和测试演示代码，您需要使用最新版本的 Delphi（至少需要 10.4 才能运行所有内容，但是大多数演示都可以在 10.x 版本上运行）。

如果您没有 Delphi 许可证，则可以使用一个试用版，该版本通常允许您 30 天免费使用编译器和 IDE。还有一个免费的 Delphi 社区版（当前更新为 10.3 版），可供没有软件开发工作收入或收入有限的任何人使用。

致谢

就像任何一本书一样，这本书欠许多人很多，太多了，无法一一列举。

分享本书第一版大部分工作的人是我的编辑 PeterWood，他不断适应我日新月异的日程，并能够提高我的技术英语水平，从而使这本书成为真正的书。

在第一版之后，读者 Andreas Toth 向我发送了对上一版的广泛反馈，他的建议有助于改进本文。此新版本还由其他一些 Delphi 专家（其中大多数是 Embarcadero MVP 中的专家）进行了审查，其中包括 FrançoisPiette，他给了我许多纠正和建议。

鉴于我目前在 Embarcadero Technologies 担任产品经理的工作，我非常感谢我的同事和研发团队的成员，因为在公司期间，我对产品及其技术的了解更加深入，这要归功于我在无数次对话、会议和电子邮件中获得的洞察力。考虑到要确保每个人都被提及是多么困难，我不会真的尝试，但只挑选出三个人作为他们的角色在这本书的第一版中有直接的投入：开发人员关系部的 David I，负责 RAD 产品管理的 John Thomas（JT）和 RAD 的架构师 Allen Bauer。

最近，我与 RADStudio 的其他两个产品经理 SarinaDuPont 和 DavidMillington，我们现任的传教士 JimMcKeeth 以及当前的杰出 R&D 建筑师进行了广泛的合作。

Embarcadero 以外的其他人继续保持重要联系，有时会提供直接的投入，从许多意大利 Delphi 专家到无数客户，Embarcadero 销售和技术合作伙伴，Delphi 社区成员，MVP 甚至是使用我经常会见的其他语言和工具的开发人员。

如果在这个小组中只有一个人，在加入 Embarcadero 之前，我已经花了很多时间，那就是 CaryJensen，我与他一起在欧洲和美国组织了几轮 Delphi 开发者日。最后，非常感谢您家人与我的旅行计划，会议之夜以及周末写的一些额外书籍有关。再次感谢 Lella，Benny 和 Jacopo。

关于我自己，作者

在过去的 25 年中，我大部分时间都在使用 Object Pascal 语言进行软件开发方面的写作，教学和咨询。我写了 MasteringDelphi 畅销书系列，后来又自行出版了有关开发工具的几本手册（关于从 Delphi2007 到 DelphiXE 的不同版本）。

我在大多数大洲的许多编程会议上都发表过演讲，并在会议，Delphi 开发人员活动，公司举办的课程，在线网络研讨会和 CodeRage 会议上向成千上万的开发人员授课。

经过多年的独立顾问和培训师工作，2013 年我的职业突然发生了变化：我担任了 Delphi 的职位，现在是 EmbarcaderoTechnologies 的 RADStudio 产品经理，该公司负责构建和销售这些开发工具。为了避免进一步困扰您，我只补充说，我目前居住在意大利，通勤到加利福尼亚（最近有所减少），有一个可爱的妻子和两个漂亮的孩子，并尽我所能重新享受编程的乐趣。。

我希望您喜欢阅读本书，就像我喜欢撰写新版本一样。

有关更多信息，请使用以下网站和社交媒体渠道：

http://www.marcocantu.com/objectpascalhandbook

http://blog.marcocantu.com

http://twitter.com/marcocantu

译者标注：

- ❖ ——注意
- ✧ ——提示
- ➢ ——警告
- ✓ ——历史

Begin......2
一本关于当今语言的书......2
通过实践学习......3
致谢......3
关于我自己，作者......4
第一部分：基础......15
第 1 章：Pascal 中的编码......16
1.1 让我们从代码开始......16
1.1.1 第一个控制台应用程序......16
1.1.2 第一个视觉应用......17
1.2 语法和编码风格......19
1.2.1 注释和 XML 文档......20
1.2.2 符号标识符......21
不区分大小写和大写的使用......22
1.2.3 空白空间......22
1.2.4 Indentation（缩进）......23
1.2.5 语法高亮......24
Error Insight and Code Insights（错误见解和代码见解）......24
1.3 语言关键词......25
1.4 程序的结构......27
1.4.1 单元和程序名称......28
Dotted Unit Names（带点“.”的单元名称）......28
More on the Structure of a Unit（有关单元结构的更多信息）......29
The Uses Clause（uses 部分）......30
1.4.2 单元和范围......31
Using Units Like Namespaces（使用像命名空间一样的单元）......31
1.4.3 程序文件......32
1.5 编译器指令......32
1.5.1 条件定义......33
1.5.2 Compiler Versions（编译器版本）......33
1.5.3 Include Files（包含文件）......2
第二章 变量和数据类型......3
2.1 变量和赋值......3
2.1.1 文字值......4
2.1.2 赋值语句......4
2.1.3 分配和转换......5
2.1.4 初始化全局变量......5
2.1.5 初始化局部变量......5
2.1.6 内联变量......6
初始化内联变量......6
2.1.7 内联变量的类型推断......7
2.1.8 Constants（常数）......7
内联常数......8
Resource（资源）字符串常量......8

2.1.9 变量的生命周期和可见性......9
2.2 Data Types（数据类型）......10
2.2.1 序数和数字类型......10
Aliased Integer Types（别名整数类型）......10
整数类型，64 位，NativeInt 和 LargeInt......11
整数类型 Helper（助手）......11
标准序数例程......12
超出范围的操作......13
2.2.2 Boolean......13
2.2.3 Characters......14
Char Type Operations（字符类型操作）......14
Char as an Ordinal Type（字符作为序数类型）......15
Converting with Chr（用 Chr 转换）......15
32-bit Characters（32 位字符）......15
2.2.4 Floating Point Types（浮点类型）......15
为什么浮点值不同......16
浮动助手和 Math 单元......17
2.3 简单的用户定义数据类型......17
2.3.1 命名与未命名类型......17
2.3.2 类型别名......18
2.3.3 子范围类型......19
2.3.4 枚举类型......19
Scoped Enumerators（范围枚举器）......20
2.3.5 集合类型......20
集合运算符......21
2.4 表达式和运算符......21
2.4.1 使用运算符......21
显示表达式的结果......22
2.4.2 运算符和优先级......22
关系和比较运算符（最低优先级）......23
加法运算符......23
乘法和按位运算符......23
一元运算符（优先级最高）......23
2.5 日期和时间......24
2.6 类型转换......26
第三章 语言语句......28
3.1 简单和复合语句......28
3.2 if 语句......29
3.3 case 语句......30
3.4 for 循环语句......31
3.4.1 for-in 循环......33
3.5 While 和 Repeat 语句......34
3.5.1 循环示例......35
3.5.2 Break 和 Continue......36

Goto 来了吗？ 没门！......37
第四章 例程和函数......38
4.1 例程和函数......38
4.1.1 Forward 声明......39
4.1.2 递归函数......40
4.1.3 什么是方法？......41
4.2 参数和返回值......41
4.2.1 Exit with a Result（结果退出）......42
4.2.2 Reference Parameters（参考参数）......43
4.2.3 Constant Parameters（常数参数）......44
4.2.4 Function Overloading（函数重载）......45
4.2.5 重载和不明确的调用......46
4.2.6 默认参数......47
4.3 Inlining（内联）......48
4.4 函数的高级功能......50
4.4.1 Object Pascal 调用约定......50
4.4.2 程序类型......51
4.4.3 外部函数声明......53
延迟加载 DLL 函数......53
第五章 arrays（数组）和 records（记录）......55
5.1 数组数据类型......55
5.1.1 Static Arrays（静态数组）......55
5.1.2 数组大小和边界......56
5.1.3 多维静态数组......57
5.1.4 Dynamic Arrays（动态数组）......58
动态阵列的本机运算......60
5.1.5 开放数组参数......60
变体类型开放数组参数......61
5.2 记录数据类型......63
5.2.1 使用记录数组......65
5.2.2 Variant（变体）记录......65
5.2.3 Fields Alignments（字段对齐）......66
5.2.4 那么 With 语句呢？......67
5.3 带方法的记录......68
5.3.1 Self：记录背后的魔力......69
5.3.2 初始化记录......70
5.3.3 记录和 Constructors（构造函数）......70
5.3.4 Operators（操作符）Gain（增添）New Ground（新天地）......71
Operators Overloading（重载）Behind the Scenes（幕后）......73
Implementing Commutativity（实现交换性）......73
Implicit Cast and Type Promotions（隐式类型转换）......74
5.3.5 Operators 和自定义托管 Record（记录）......75
带有初始化和终结运算符的记录......75
将托管记录作为参数传递......77

Exceptions（例外）和托管记录 Managed Records....77
托管记录数组....77
5.4 Variants（变体）....77
5.4.1 变体没有类型....78
5.4.2 深度的变体....79
5.4.3 Variants Are Slow（变体变慢）....79
5.5 指针呢？....80
5.6 File Types, Anyone（文件类型，有人吗）?....83
第六章 关于 Strings 的全部....84
6.1 Unicode：整个世界的字母....84
6.1.1 过去的字符：从 ASCII 到 ISO 编码....84
6.1.2 Unicode 代码点和字素....85
6.1.3 从代码点到字节（UTF）....85
6.1.4 The Byte Order Mark（字节顺序标记）....86
6.1.5 看一下 Unicode....87
6.2 字符类型再探....89
6.2.1 字符单元的 Unicode 操作....89
6.2.2 Unicode 字符文字....91
6.2.3 1 字节字符呢？....91
6.3 字符串数据类型....92
6.3.1 将字符串作为参数传递....94
6.3.2 []和字符串字符计数模式的使用....95
6.3.3 连接字符串....96
6.3.4 字符串 Helper（助手）操作....97
6.3.5 更多字符串 RTL 函数....100
6.3.6 格式化字符串....100
6.3.7 字符串的内部结构....102
6.3.8 查看内存中的字符串....103
6.4 字符串和 Encoding（编码）....104
6.5 其他类型的字符串....106
6.5.1 UCS4String 类型....107
6.5.2 较旧的字符串类型....107
第二部分 Object pascal 的 oop....108
第七章 Objects（对象）....109
7.1 介绍类和对象....109
7.1.1 类的定义....109
7.1.2 其他 OOP 语言的类....110
7.1.3 类方法....111
7.1.4 创建一个对象....111
7.2 对象参考模型....112
7.2.1 处理 Objects....113
7.2.2 什么是“Nil”?....113
7.2.3 内存中的记录与类....114
7.3 Private（私有）, Protected（受保护）,和 Public 公共....114

7.3.1 private（私有的）数据的例子......115
7.3.2 Encapsulation（封装）和 Forms（表单）......117
7.4 Self 关键字......118
7.4.1 动态创建组件......119
7.5 Constructors（构造）......120
7.5.1 使用构造函数和析构函数管理本地类数据......121
7.5.2 Overload（重载）方法和构造函数......122
7.5.3 完整的 TDate 类......123
7.6 嵌套类和嵌套常量......125
第八章 Inheritance（继承）......128
8.1 从现有类继承......128
8.2 普通基类......130
8.3 Protected（受保护的）字段和封装......130
8.3.1 使用“Protected Hack（受保护的黑客）”......130
8.4 从 Inheritance（继承）到 Polymorphism（多态）......132
8.4.1 Inheritance（继承）和类 Compatibility（兼容性）......132
8.4.2 Late Binding（后期绑定）和 Polymorphism（多态）......133
8.4.3 Override(覆盖),Redefine(重新定义)和 Reintroduce(重新引入)方法......135
8.4.4 Inheritance（继承）和 Constructor（构造函数）......136
8.4.5 Virtual（虚拟）与 Dynamic（动态）方法......137
Windows 上的消息处理程序......137
8.5 Abstract（抽象）方法和类......138
8.5.1 Abstract（抽象）方法......138
8.5.2 Sealed（密封）类和 Final（最终）方法......139
8.6 Safe（安全）类型转换运算符......140
8.7 Visual Form Inheritance（可视表单继承）......141
8.7.1 从基本表单继承......142
第九章 异常处理......144
9.1 Try-Except 块......144
9.1.1 异常层次结构......146
9.1.2 引发异常......147
9.1.3 异常与 Stack（堆栈）......148
9.2 Finally 块......149
9.2.1 Finally 和 Except......150
9.2.2 在 Finally 块中恢复游标......150
9.2.3 使用托管记录还原游标......150
9.3 现实世界中的异常......151
9.4 Global（全局）异常处理......152
9.5 异常和构造函数......153
9.6 异常的高级功能......154
9.6.1 嵌套异常和 InnerException 机制......154
9.6.2 Intercepting（拦截）异常......157
第十章 属性和事件......158
10.1 定义 property（属性）......158

10.1.1 与其他编程语言相比的属性......159
10.1.2 给属性编码......160
10.1.3 向表单添加属性......161
10.1.4 将属性添加到 TDate 类......162
10.1.5 使用数组属性......164
10.1.6 通过引用设置属性......164
10.2 Published（发布的）访问说明符......165
10.2.1 Design-Time（设计时）属性......166
10.2.2 Published（发布的） and Forms（表单）......166
10.2.3 自动 RTTI......167
10.3 事件驱动编程......168
10.3.1 方法指针......169
10.3.2 Delegation（委托）的概念......170
10.3.3 事件就是属性......171
10.3.4 将事件添加到 TDate 类......172
10.4 创建一个 TDate 组件......174
10.5 在类中实现枚举支持......175
10.6 关于混合 RAD 和 OOP 的 15 个技巧......177
技巧 1：Form（表单）就是类......178
技巧 2：Component（组件）名称......178
技巧 3：Envent（事件）名称......178
技巧 4：使用表单方法......178
技巧 5：添加表单构造函数......179
技巧 6：避免全局变量......179
技巧 7：永远不要在 TForm1 方法中使用 Form1......179
技巧 8：很少在其他表单中使用 Form1......179
技巧 9：删除全局 Form1 变量......179
技巧 10：添加表单属性......180
技巧 11：公开组件属性......180
技巧 12：在需要时使用数组属性......180
技巧 13：在属性中开始操作......180
技巧 14：隐藏组件......181
技巧 15：使用 OOP 表单向导......181
技巧结论......181
第十一章 Interfaces（接口）......182
11.1 使用 Interfaces（接口）......182
11.1.1 声明接口......183
11.1.2 Implementing（实施）接口......183
11.1.3 接口和引用计数......185
11.1.4 混合参考中的错误......185
11.1.5 Weak（弱）和 Unsafe（不安全）的接口参考......187
11.2 先进的接口技术......188
11.2.1 接口属性......188
11.2.2 接口 Delegation（委托）......189

11.2.3 多重接口和方法别名......190
11.2.4 接口（Polymorphism）多态......192
11.2.5 从接口引用中提取对象......193
11.3 用接口实现 Adapter Pattern（适配器模式）......194
第十二章 Manipulating（操纵）类......196
12.1 类方法和类数据......196
12.1.1 类数据......196
12.1.2 虚拟类方法和隐藏的 self 参数......197
12.1.3 类 Static（静态）方法......197
静态类方法和 Windows API Callbacks（回调）......198
12.1.4 类属性......199
12.1.5 具有实例计数器的类......199
12.2 类构造函数（和析构函数）......200
12.2.1 RTL 中的类构造函数......201
12.2.2 实现 Singleton Pattern（单例模式）......202
12.3 类参考......202
12.3.1 RTL 中的类引用......203
12.3.2 使用类引用创建组件......204
12.4 类和记录助手......205
12.4.1 类 Helper（助手）......205
列表框的类助手器......207
12.4.2 类助手和继承......207
12.4.3 使用类助手添加控件枚举......208
12.4.4 记录内在类型的助手......210
12.4.5 类型别名的助手......211
第十三章 对象和内存......213
13.1 全局数据，Stack（栈）和 Heap（堆）......213
13.1.1 Global（全局）Memory......213
13.1.2 Stack（栈）......214
13.1.3 Heap（堆）......214
13.2 对象参考模型......215
13.2.1 将对象作为参数传递......215
13.3 内存管理技巧......216
13.3.1 销毁您创建的对象......216
13.3.2 销毁对象一次......217
13.4 内存管理和接口......218
13.4.1 有关 Weak（弱）References（引用）的更多信息......219
Weak References Are Managed（弱引用被管理）......220
13.4.2 The Unsafe Attribute（不安全属性）......221
13.5 Tracking（跟踪）和检查内存......222
13.5.1 Memory Status（内存状态）......222
13.5.2 FastMM4......222
13.5.3 跟踪泄漏和其他全局设置......223
13.5.4 Full FastMM4 中的缓冲区溢出......224

13.5.5 Windows 以外的平台上的内存管理 226
13.5.6 跟踪每个类 Allocations（分配） 226
13.6 编写健壮的应用程序 226
13.6.1 构造函数，析构函数和异常 227
13.6.2 嵌套的 Finally 块 228
13.6.3 Dynamic（动态）类检查 228
13.6.4 该指针是对象引用吗？ 229
第三部分 Advanced features（高级功能） 232
第十四章 generics（泛型） 233
14.1 Generic Key-Value Pairs（泛型键值对） 233
14.1.1 内联变量和泛型类型推断 235
14.1.2 泛型的类型规则 235
14.2 Object Pascal 中的泛型 236
14.2.1 泛型类型兼容性规则 237
14.2.2 标准类的泛型方法 238
14.2.3 泛型类型实例化 239
14.2.4 泛型类型函数 240
14.2.5 泛型类的类构造函数 242
14.3 Generic Constraints（泛型约束） 243
14.3.1 Class Constraints（类约束） 244
14.3.2 特定类约束 245
14.3.3 接口约束 245
14.3.4 接口 References（参考）与泛型接口约束 247
14.3.5 默认构造函数约束 248
14.3.6 约束摘要和组合 249
14.4 预定义的泛型容器 250
14.4.1 使用 TList<T> 250
14.4.2 对 TList <T>排序 251
14.4.3 用匿名方法排序 252
14.4.4 对象容器 253
14.4.5 使用泛型 Dictionary（字典） 254
14.4.6 字典与字符串 Lists（列表） 256
14.5 泛型接口 257
14.5.1 预定义的泛型接口 259
14.6 Object Pascal 中的智能指针 259
14.6.1 将记录用于智能指针 260
14.6.2 使用泛型托管记录实现智能指针 261
14.6.3 使用泛型记录和接口实现智能指针 262
14.6.4 添加隐式转换 263
14.6.5 Comparing（比较）智能指针解决方案 264
14.7 具有泛型的 Covariant（协变）返回类型 264
14.7.1 Animals（动物），Dog（狗）和 Cat（猫） 265
14.7.2 具有泛型返回值的方法 266
14.7.3 返回不同类的派生对象 266

第十五章 Anonymous（匿名）方法......268
15.1 匿名方法的语法和语义......268
15.1.1 匿名方法变量......268
15.1.2 匿名方法参数......269
15.2 使用局部变量......269
15.2.1 延长局部变量的寿命......270
15.3 幕后的匿名方法......271
15.3.1 （可能）缺少括号......271
15.3.2 匿名方法实现......272
15.3.3 准备使用参考类型......273
15.4 现实世界中的匿名方法......273
15.4.1 匿名事件处理程序......274
15.4.2 计时匿名方法......275
15.4.3 Threads Synchronization（线程同步）......276
15.4.4 Object Pascal 中的 AJAX......278
第十六章 reflection（反射）与 attributes（属性）......281
16.1 扩展 RTTI......281
16.1.1 第一个例子......281
16.1.2 编译器生成的信息......282
16.1.3 弱类型和强类型链接......283
16.2 RTTI 单元......284
16.2.1 Rtti 单元中的 RTTI 类......285
16.2.2 RTTI 对象生命周期管理和 TRttiContext 记录......286
16.2.3 显示类信息......287
16.2.4 RTTI for Packages（程序包）......288
16.3 TValue 结构......289
16.3.1 使用 TValue 读取属性......290
16.3.2 Invoking（调用）方法......291
16.4 使用属性......291
16.4.1 什么是属性？......291
16.4.2 属性类和属性声明......292
16.4.3 Browsing Attributes（浏览属性）......294
16.5 虚拟方法 Interceptors（拦截器）......295
16.6 RTTI 案例研究......298
16.6.1 ID 和说明的属性......298
描述属性类......299
样本类......299
示例项目和属性导航......300
16.6.2 XML Streaming（XML 流）......302
普通的 XML Writer 类......302
基于经典 RTTI 的流......303
具有扩展 RTTI 的流字段......304
使用属性来自定义流......305
16.6.3 其他基于 RTTI 的库......307

第十七章 TObject 和 system 单元 308
17.1 TObject 类 308
17.1.1 构造函数和析构函数 308
17.1.2 了解对象 309
17.1.3 TObject 类的更多方法 309
显示类信息 310
17.1.4 TObject 的虚拟方法 310
ToString 方法 311
Equals（等于）方法 311
GetHashCode 方法 311
使用 TObject 虚拟方法 312
17.1.5 TObject 类摘要 313
17.1.6 Unicode 和类名 314
17.2 系统单元 314
17.2.1 选定的系统类型 315
17.2.2 系统单元中的接口 315
17.2.3 选定的系统 Routines（例程） 316
17.2.4 预定义的 RTTI 属性 316
第十八章 其他核心 RTL 类 318
18.1 Class Unit（类单元） 318
18.1.1 类单元中的类 318
18.1.2 TPersistent（持久）类 319
18.1.3 TComponent 类 320
Components Ownership（所有权） 320
组件属性 321
组件流 321
18.2 现代文件访问 321
18.2.1 IOUtils（输入/输出实用程序）单元 322
提取子文件夹 322
搜索文件 322
18.2.2 Streams（流）介绍 323
普通 Stream（流）类 323
使用 Streams（流） 323
18.2.3 使用 Readers 和 Writers 324
Readers（文本阅读器）和 writers（文本编写器） 324
二进制读写器 325
18.3 构建字符串和字符串列表 326
18.3.1 TStringBuilder 类 326
StringBuilder 中的方法链接 326
18.3.2 使用字符串列表 327
18.4 运行时库很大 327
in closing 330
end 330

第一部分：基础

Object Pascal 是一种非常强大的语言，基于良好的程序结构和可扩展的数据类型等核心基础。这些基础部分源于传统的 Pascal 语言，但即使是核心语言功能，从早期开始也有许多扩展。

在本书的第一部分中，您将学习语言语法，编码风格，程序的结构，变量和数据类型的使用，基本语言语句（如条件和循环），过程的使用以及函数和核心类型构造函数，例如数组，记录和字符串。

这些是从类到泛型类型的更高级功能的基础，我们将在本书的第二部分和第三部分中进行探讨。学习语言就像盖房子一样，您需要以坚实的基础和良好的基础开始，否则上方和上方的其他所有事物都将闪闪发光……但不稳定。

摘要

第 1 章：Pascal 中的编码
第 2 章：变量和数据类型
第 3 章：语言语句
第 4 章：过程和函数
第 5 章：数组和记录
第 6 章：关于字符串的全部

第 1 章：Pascal 中的编码

本章从 Object Pascal 应用程序的一些构建模块开始，涵盖了编写代码和相关注释的标准方法，介绍关键字以及程序的结构。我将开始编写一些简单的应用程序，尝试解释它们的作用，然后在接下来的章节中介绍一些其他关键概念，这些概念将在更多细节中介绍。

1.1 让我们从代码开始

本章介绍了语言的基础，但将需要我一些章节来指导您完成完整的工作应用程序的细节。因此，现在让我们先看两个程序（它们的结构不同），而不必真正涉及太多细节。在这里，我只想向您展示程序的结构，我将使用它们来构建演示以解释特定的语言构造，然后才能涵盖所有各种要素。鉴于我希望您能够尽快将书中的信息付诸实践，因此从头开始看演示示例将是一个好主意。

Object Pascal 被设计为与它的集成开发环境紧密结合。通过这种强大的组合，Object Pascal 可以与程序员友好的语言的轻松开发速度相匹配，并与机器友好的语言的运行速度相匹配。

通过 IDE，您可以设计用户界面，帮助您编写代码，运行程序等等。在向您介绍 Object Pascal 语言时，我将在整本书中使用 IDE。

1.1.1 第一个控制台应用程序

首先，我将向您展示一个简单的 HelloWorld 控制台应用程序的代码，该应用程序显示了 Object Pascal 程序的一些结构元素。控制台应用程序是没有图形用户界面，显示文本并接受键盘输入的程序，通常从操作系统控制台或命令提示符处执行。控制台应用程序在移动平台上通常没有什么意义，但仍在 Windows 上使用（Microsoft 最近已在其中进行了 cmd.exe 改进，PowerShell 和终端访问方面的努力），而在 Linux 上却相当受欢迎。

我还不会解释下面代码的不同元素的含义，因为这是本书前几章的目的。这是 HelloConsole 应用程序项目中的代码：

```
program HelloConsole;
{$APPTYPE CONSOLE}
var
  StrMessage: string;
begin
  StrMessage := 'Hello, World';
  writeln (StrMessage);
  // 等到按下 Enter 键
  readln;
end.
```

❖ 如引言中所述，本书涵盖的所有演示的完整源代码都可以在 GitHub 的在线存储库中找到。有关如何获取这些演示的更多详细信息，请参见本书简介。在文本中，我指的是项目名称（在本例中为 HelloConsole），它也是包含演示的各种文件的文件夹的名称。项目文件夹按章节分组，因此您可以在 01/HelloConsole 下找到第一个演示。

您可以在特定声明，编译器指令（以$符号为前缀并用花括号括起来）后的第一行中看到程序名称，一个变量声明（具有给定名称的字符串）和三行代码在

开始的主要内容中加上注释。这三行代码将值复制到字符串中，调用一个系统函数将该行文本写入控制台，并调用另一个系统函数以读取一行用户输入（或者在这种情况下，要等到用户按下回车键）。正如我们将看到的，您可以定义自己的函数，但是 Object Pascal 具有数百个预定义的函数。

再次，我们将很快了解所有这些元素，因为此初始部分仅用于使您了解小型但完整的 Pascal 程序的外观。当然，您可以打开并运行此应用程序，它将产生如下所示的输出（实际的 Windows 版本显示在图 1.1 中）。

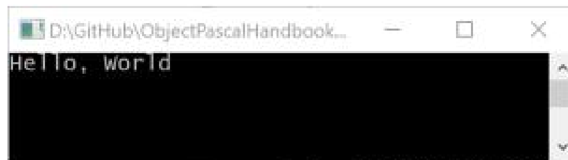

图 1.1：在 Windows 上运行的 HelloConsole 示例的输出

1.1.2 第一个视觉应用

虽然，现代应用程序很少看起来像这个老式的控制台程序，但是通常由显示在窗口（称为表单）中的可视元素（称为控件）组成。在本书的大多数情况下，我将使用 FireMonkey 库（也称为 FMX）来构建视觉演示（即使在大多数情况下，它们会简化为显示简单文本）。

❖ 在 Delphi 中，视觉控件有两种形式：VCL（用于 Windows 的视觉组件库）和 FireMonkey（用于所有受支持平台，台式机和移动设备的多设备库）。无论如何，使演示适应 Windows 特定的 VCL 库应该相当简单。

要理解可视化应用程序的确切结构，您必须阅读本书的大部分内容，因为表单是给定类的对象，并且具有方法，事件处理程序和属性……所有这些功能将需要一段时间才能完成。但是能够创建这些应用程序并不需要成为专家，因为您要做的就是使用菜单命令来创建新的桌面或移动应用程序。在本书的第一部分中，我要做的就是将演示基于 FireMonkey 平台，并仅使用表单和按钮单击操作的上下文。首先，您可以创建任何类型的表单（台式机或移动设备，我通常会选择一个多设备“空白”应用程序，因为它也将在 Windows 上运行），然后在其上放置一个按钮控件，它下方的多行文本控件（或备注）以显示输出。

图 1.2 显示了在选择 Android 样式预览（请参见设计图上方的组合框）并添加单个控件（即按钮）之后，您的应用程序表单在 DelphiIDE 中查找移动应用程序的方式。

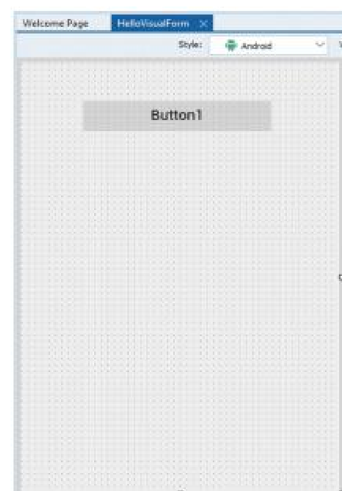

图 1.2：HelloVisual 演示使用的带有一个按钮的简单移动应用程序

创建类似应用程序所要做的就是将按钮添加到空白表格中。现在添加实际的代码，这是我们目前唯一感兴趣的内容，双击按钮，将为您提供以下代码框架（或类似的内容）：

```
procedure TForm1.Button1Click (Sender: TObject)
begin

end;
```

即使您不知道类的方法是什么（Button1Click 是什么），也可以在该代码段中键入某些内容（这意味着在 begin 和 end 关键字内），并且在按下按钮时该代码将执行。

我们的第一个“可视”程序的代码与第一个控制台应用程序的代码匹配，仅在不同的上下文中并且调用不同的库函数，即 ShowMessage 全局函数，该函数用于在消息框中显示某些字符串。这是您可以在 HelloVisual 应用程序项目中找到的代码，您可以尝试非常轻松地从头开始重建它：

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

如何将 strMessage 变量的声明放置在 begin 语句之前，并将实际代码放置在它之后。 同样，不要担心如果不清楚，一切都会在适当的时候得到详尽的解释。

- 您可以在本章的 01 容器下的文件夹中找到此演示的源代码。但是，在这种情况下，有一个项目文件名（如演示），还有一个次级单位文件，在项目名后添加了单词“Form”。那是我在书中要遵循的标准。项目的结构在本章的最后介绍。

在图 1.3 中，您可以看到此简单程序的输出，该程序在启用了 FMXMobilePreview 模式的 Windows 上运行（您也可以在 Android，iOS 和 macOS 上运行此演示，但需要在 IDE 中进行一些额外配置）。

- FireMonkey Mobile Preview 使 Windows 应用程序看起来有点像移动应用程序。 我在本书的大多数演示中都启用了此模式。 这是通过在项目源代码中为 MobilePreview 单元添加一个 use 语句来完成的。

现在，我们已经有了编写和测试演示程序的方法，让我们回到本章的最后，就像我在本章开始时所承诺的那样，通过覆盖应用程序前几个构建块的所有细节。您需要了解的第一件事是如何读取程序，如何编写各种元素以及我们刚刚构建的应用程序的结构（具有 PAS 文件和 DPR 文件）。

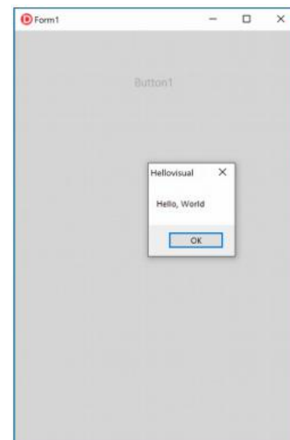

图 1.3：HelloVisual 演示使用的带有一个按钮的简单移动应用程序

1.2 语法和编码风格

在继续讨论编写实际的 Object Pascal 语言语句这一主题之前，重要的一点是要强调 Object Pascal 编码风格的一些要素。我在这里要解决的问题是：除了语法规则（我们仍然没有研究过）之外，您应该如何编写代码？这个问题没有一个唯一的答案，因为个人品味可以决定不同的风格。但是，您需要了解一些有关注释，大写字母，空格以及许多年前被称为漂亮打印（对于我们人类来说是漂亮的东西，而不是计算机）的原则，而现在这个术语已经过时了。

通常，任何编码风格的目标都是清晰。您做出的样式和格式决定是一种简写形式，表示给定代码段的用途。

保持一致性的一种重要工具是一致性-无论您选择哪种样式，请务必在整个项目中以及跨项目遵循它。

✧ IDE（集成开发环境）支持自动代码格式设置（在单位或项目级别）：您可以要求编辑器使用 Ctrl+D 键重新设置代码格式，遵循一组规则，您可以通过调整约 40 种不同的格式设置元素（在 IDE 选项中找到）来进行更改，甚至与团队中的其他开发人员共享这些设置以使格式保持一致。但是，自动格式化不支持某些最新的语言功能。

评论

尽管代码通常是不言自明的，但还是有必要在程序的源代码中添加大量注释，以便进一步向其他人（以及将来在很长一段时间内查看代码时向您自己）解释。代码是以给定的方式编写的，并且有什么假设。

在传统的 Pascal 中，注释用大括号或括号括起来，后面加星号。该语言的现代版本还接受 C ++样式的单行注释，双斜杠，该注释跨越行的结尾，并且不需要任何符号来表示注释的结尾：

```
// 这是直到行尾的注释
{ 这是多行注释 }
(* 这是另一条
多行注释 *)
```

迄今为止，第一种评论形式是最常见的形式，但它最初并不是 Pascal 的一部分。它是从 C/C++借来的，C/C++也将/*comment*/语法用于多行注释，以及 C＃，Objective-C，Java 和 JavaScript。

第二种形式比第三种形式更常用，第三种形式在欧洲经常被使用，因为许多欧洲键盘缺少花括号符号（或通过多键组合使其难以使用）。换句话说，最古老的语法有点过时。

直到行尾的注释对于简短的注释和注释掉一行代码非常有帮助。在现代 Object Pascal 语言中，它们是迄今为止最常见的注释形式。

✧ 在 IDE 编辑器中，您可以通过直接击键注释或取消注释当前行（或一组选定行）。 这是美式键盘上的 Ctrl + /，而其他键盘上的组合是不同的（带有物理/键）：实际键在编辑器的弹出菜单中列出。

拥有三种不同形式的注释有助于标记嵌套的注释。如果要注释掉几行源代码以禁用它们，并且这些行包含一些实际注释，则不能使用相同的注释标识符：

```
{
  code...
  {嵌套评论，造成问题}
  code...
```

```
}
```

上面的代码会导致编译器错误，因为第一个封闭的大括号表示整个注释部分的结尾。 使用第二个注释标识符，可以编写以下正确的代码：

```
{
  code...
  // 这个评论还可以
  code...
}
```

一种替代方法是如上所述注释掉一组行，因为它将在注释行中添加第二条注释，您可以通过取消注释相同的块来轻松删除（保留原始注释）。

❖ 如果在括号或星号后加美元符号（$），则不再是注释，而是变成编译器指令，正如我们在{$APPTYPECONSOLE}行的第一个演示中所看到的那样。编译器指令指示编译器执行一些特殊的操作，并在本章末简要说明。
实际上，编译器指令仍然是注释。例如，{$X+这是一条评论}是合法的。
尽管大多数理智的程序员可能会倾向于将指令和注释分开，但它既是有效的指令又是注释。

1.2.1 注释和 XML 文档

有一个注释的特殊版本，也是其他编程语言所共有的，编译器以特殊方式对其进行处理。这些特殊注释会生成其他文档，这些文档可直接用于 IDE Help Insight 和编译器生成的 XML 文件中。

❖ 在 Delphi IDE 中，“帮助”见解会自动显示有关符号的信息（包括符号的类型和定义位置）。使用 XML Doc 注释，您可以使用源代码本身中编写的特定详细信息来扩充此信息。

通过使用///注释或{! 来启用 XML Doc。评论。在这些注释中，您可以使用常规文本或（更好）特定的 XML 标记来指示有关注释符号，其参数和返回值等的信息。这是自由格式文本的非常简单的情况：

```
public
  /// 当然，这是一种自定义方法
  procedure CustomMethod;
```

如果启用 XML Doc 生成，则将信息添加到由编译器生成的 XML 输出中，如下所示：

```
<procedure name="CustomMethod" visibility="public">
  <devnotes>
  当然，这是一种自定义方法
  </devnotes>
</procedure>
```

将鼠标悬停在符号上时，IDE 中会显示相同的信息，如图 1.4 所示。

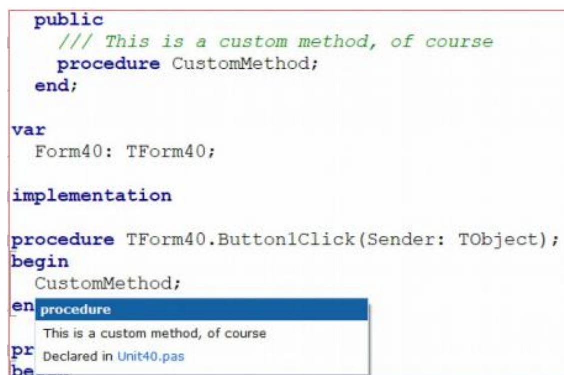

图 1.4：Delphi IDE 中的 Help Insight 显示以///注释编写的 XML Doc 信息

如果您在注释中提供了摘要部分，则遵循建议的准则，这将同样显示在“Help Insight”窗口中：

```
public
  /// <summary>当然，这是一种自定义方法</summary>
  procedure CustomMethod;
```

优点是，还有许多其他 XML 标记可用于参数，返回值和更详细的信息。可用标签在以下位置列出：

http://docwiki.embarcadero.com/RADStudio/en/XML_Documentation_Comments

1.2.2 符号标识符

程序由许多不同的符号组成，可以引入这些符号来命名各种元素（数据类型，变量，函数，对象，类等）。尽管您几乎可以使用所需的任何标识符，但必须遵循一些规则：

- 标识符不能包含空格（因为空格会将标识符与其他语言元素分开）
- 标识符可以使用字母和数字，包括整个 Unicode 字母中的字母； 因此，您可以根据需要使用自己的语言命名符号（不建议这样做，因为 IDE 的某些工具部分可能无法提供相同的支持）
- 在传统的 ASCII 符号中，标识符只能使用下划线符号（_）;不允许使用字母和数字旁边的所有其他 ASCII 符号。标识符中的非法符号包括匹配符号（+，-，*，/，=），所有括号和花括号，标点符号，特殊字符（包括@，＃，$，％，^，＆，\，|）。但是，您可以使用 Unicode 符号，例如☼♪或∞。
- 标识符必须以字母或下划线开头，不允许以数字开头（换句话说，您可以使用数字，但不能用作第一个符号）。在这里，数字是指 0 到 9 的 ASCII 数字，而其他数字的 Unicode 表示形式也是允许的。

以下是 IdentifiersTest 应用程序中列出的经典标识符的示例：

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

这些是合法的 Unicode 标识符的示例（最后一个有点极端）：

```
Cantù (拉丁字母重音)
结 (简体中文现金余额)
画像 (日文图片)
☼ (Sun Unicode 符号)
```

这些是无效标识符的一些示例：

```
123
1Value
My Value
My-Value
My%Value
```

✧ 如果您想在运行时检查有效的标识符（除非您正在编写用于帮助其他开发人

员的工具，否则很少需要这样做），则可以在运行时库中使用一个称为 IsValidIdent 的函数。

不区分大小写和大写的使用

与许多其他语言（包括所有基于 C 语法的语言（例如 C ++，Java，C＃和 JavaScript）不同，Object Pascal 编译器将忽略标识符的大小写或大写。因此，标识符 Myname，MyName，myname，myName 和 MYNAME 都完全相同。我个人认为，不区分大小写绝对是一个积极的功能，因为语法错误和其他细微的错误可能是由于大小写敏感的语言中不正确的大写字母引起的。

但是，如果考虑到可以将 Unicode 用作标识符的事实，事情就会变得更加复杂，因为字母的大写版本被视为相同的元素，而同一字母的重音版本则被视为一个单独的符号。换一种说法：

```
cantu: Integer;
Cantu: Integer; //  错误：重复的标识符
cantù: Integer; //  正确：不同的标识符
```

- 语言的不区分大小写规则只有一个例外：由于 C ++兼容性问题，组件包的 Register 过程必须以大写 R 开头。当然，当您引用其他语言导出的标识符时（如本机操作系统功能），则可能必须使用适当的大小写。

但是，有一些细微的缺点。首先，您必须意识到这些标识符实际上是相同的，因此必须避免将它们用作不同的元素。其次，您应尝试在使用大写字母时保持一致，以提高代码的可读性。

编译器不会强制使用大小写一致，但是这是一个很好的习惯。一种常见的方法是仅大写每个标识符的首字母。

当标识符由几个连续的单词组成（您不能在标识符中插入空格）时，单词的每个首字母应大写：

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

通常将其称为“Pascal-casing”，以与 Java 和其他基于 C 语法的语言的“Camel-casing”形成对比，C 语法将内部单词的首字母大写，例如：

```
myLongIdentifier
```

实际上，越来越常见的情况是使用 Object Pascal 代码，其中局部变量使用驼峰式大小写（小写的首字母），而类元素，参数和其他更全局的元素使用 Pascal 大小写。无论如何，在本书的源代码片段中，我都尝试对所有符号始终使用 Pascal 外壳。

1.2.3 空白空间

编译器完全忽略的其他元素是添加到源代码中的空格，换行和制表符。所有这些元素统称为空白。空白仅用于提高代码的可读性。它不会以任何方式影响编译。

与传统的 BASIC 不同，Object Pascal 允许您在几行代码中编写一条语句，将一条长指令分成两行或更多行。允许在多行中使用语句的缺点是，您必须记住添加一个分号来表示一条语句的结尾，或更准确地说，是将一个语句与下一个语句分开。在不同行上拆分编程语句的唯一限制是字符串文字不能跨越多行。

尽管很奇怪，但以下几行都表示相同的语句：

```
A := B + 10;
A :=
  B
  +
  10;
A
:=
// 这是一个中间陈述
B + 10;
```

同样，对于空格和多行语句的使用没有固定的规则，只有一些经验法则：

- 编辑器有一条竖线，您可以在 80 个左右的字符后放置竖线。如果您使用此行并尝试避免超过此限制，则源代码看起来会更好，并且无需在较小屏幕的计算机上水平滚动即可阅读源代码。80 个字符背后的最初想法是使代码在打印时看起来更好，这在当今并不常见（但仍然很有价值）。
- 当一个函数或过程具有多个复杂的参数时，通常将参数放在不同的行上。
- 您可以在注释之前将行完全留白（空白），也可以将一长段代码分成较小的部分。即使这个简单的想法也可以提高代码的可读性。
- 使用空格分隔函数调用的参数，甚至可能在初始打开括号之前使用空格。我也喜欢将表达式的操作数分开，尽管这是优先选择的问题。

1.2.4 Indentation（缩进）

关于使用空白的最后建议与典型的 Pascal 语言格式样式和缩进有关。

❖ 缩进规则取决于个人喜好，我不想参与制表符对空格的争斗。在这里，我只是指出这是 Object Pascal 世界中“最常见”或“标准”的格式化样式，这是 Delphi 库源代码中使用的样式。从历史上看，在 Pascal 世界中，这套规则被认为是精美印刷的，这个术语现在已经很少见了。

这条规则很简单：每次需要编写复合语句时，都要在当前语句的右边缩进两个空格（不是制表符，就像 C 程序员通常那样）。另一个复合语句内部的复合语句缩进了四个空格，依此类推：

```
if ... then
  statement;

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

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

同样，程序员对此通用规则有不同的解释。一些程序员将 begin 和 end 语句缩进内部代码的级别，另一些程序员将 begin 放在前一个语句行的末尾（类似于

C 的方式）。这主要是个人喜好问题。

在 type 和 var 关键字之后，变量或数据类型的列表通常使用类似的缩进格式：

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

❖ 在上面的代码中，您可能想知道为什么两个不同的类型（字符串和整数）以不同的大小写写成首字母。正如最初的《Object Pascal 样式指南》所言，“诸如 Integer 之类的类型只是标识符，并带有第一个大写字母；但是，使用保留字串声明字串，该字串应全部为小写。”

过去，在声明自定义类型和变量时，通常使用基于列的分隔符缩进。在这种情况下，上面的代码将如下所示：

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

缩进通常还用于从上一行继续的语句或函数的参数（如果您未将每个参数放在单独的行中）：

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

1.2.5 语法高亮

为了使读取和编写 Object Pascal 代码更容易，IDE 编辑器具有一项称为语法突出显示的功能。根据所键入单词的语言含义，它们以不同的颜色和字体样式显示。默认情况下，关键字以粗体显示，字符串和注释以颜色显示（通常用斜体显示），依此类推。

保留字，注释和字符串可能是受益于此功能的三个元素。您可以一眼看出拼写错误的关键字，未正确终止的字符串以及多行注释的长度。

您可以使用 IDE 的“选项”对话框的“Editor Colors（编辑器颜色）”页面轻松地自定义语法突出显示设置。如果您是唯一使用计算机查看 Object Pascal 源代码的人，请选择所需的颜色。如果您与其他程序员紧密合作，则应该都同意标准的配色方案。我经常发现，在使用与我通常使用的语法颜色不同的语法的计算机上工作确实令人困惑。

Error Insight and Code Insights（错误见解和代码见解）

IDE 编辑器具有更多功能，可帮助您编写正确的代码。最明显的是 Error Insight，它以一个字形处理器标记拼写错误的方式，在它无法理解的源代码元素下放置了一个红色的花体。

❖ 有时，您需要第一次编译程序，以避免针对完全合法的代码显示 ErrorInsight 指示。同样，保存文件（例如表单）可能会强制包含当前组件所需的正确单位，从而解决错误的 InsightInsight 指示。这些问题已在 Delphi10.4 中通过新的基于 LSD（基于语言服务器协议）的 CodeInsight 得以解决。

其他功能（如代码完成）可通过在您编写的位置提供合法符号列表来帮助您

编写代码。当函数或方法具有参数时，您将在键入时看到列出的参数。您还可以将鼠标悬停在符号上以查看其定义。但是，这些是编辑器特有的功能，我不想详细研究，因为我想继续专注于语言，而不是详细讨论 IDE 编辑器（即使它是迄今为止用于编写的最常用工具）Object Pascal 代码）。

1.3 语言关键词

关键字是该语言保留的所有标识符。这些是具有预定义含义和作用的符号，您不能在其他上下文中使用它们。

保留字与伪指令在形式上是有区别的：保留字不能用作标识符，伪指令具有特殊含义，但也可以在不同的上下文中使用（尽管建议不要这样做）。实际上，您不应使用任何关键字作为标识符。

如果您编写类似以下的代码（其中 property 实际上是关键字）：

```
var
  property: string
```

您会看到以下错误消息：

E2029 Identifier expected but 'PROPERTY' found

（E2029 应该有标识符，但找到了“属性”）

通常，当您滥用关键字时，由于编译器可以识别关键字，但是会因其在代码中的位置或以下元素而感到困惑，因此根据情况，您会收到不同的错误消息。

在这里，我不想向您显示完整的关键字列表，因为其中一些关键字比较模糊且很少使用，而仅列出其中的一部分，并按其角色进行分组。我将花费几章来探讨所有这些关键字，而我在此列表中跳过的其他关键字。您可以在以下位置找到官方参考：

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

❖ 请注意，某些关键字可以在不同的上下文中使用，这里我通常只指最常见的上下文（尽管有两次列出了两个关键字）。原因之一是多年来，编译器团队希望避免引入新的关键字，因为这可能会破坏现有的应用程序，因此他们回收了一些现有的应用程序。

因此，让我们从在初始演示源代码中已经看到的，用于定义 structure of an application project（应用程序项目结构）的一些关键字开始探索关键字：

Program	指示应用程序项目的名称
Library	指示库项目的名称
Package	表示软件包库项目的名称
Unit	表示一个单元的名称，一个源代码文件
Uses	指代码所依赖的其他单位
Interface	带有声明的单元的一部分
Implementation	带有实际代码的单元部分
Initialization	程序启动时执行的代码
Finalization	程序终止时执行的代码
Begin	代码块的开始
End	代码块的结尾

另一组关键字涉及不同基本数据类型的声明以及此类数据类型的变量：

Type	引入了一个类型声明块
Var	引入了一个变量声明块

Const　　　　　　引入了一个常量声明块
Set　　　　　　　定义功率设定数据类型
String　　　　　　定义字符串变量或自定义字符串类型
Array　　　　　　定义数组类型
Record　　　　　　定义记录类型
Integer　　　　　定义一个整数变量
real, single, double　定义浮点变量
File　　　　　　　定义一个文件

第三组包括用于基本语言语句的关键字，例如条件和循环，还包括函数和过程：

if　　　　引入条件语句
then　　　将条件与代码分开以执行
else　　　指示可能的替代代码
case　　　介绍带有多个选项的条件语句
of　　　　将条件与选项分开
for　　　 引入了修复重复周期
to　　　　指示 for 循环的最终上限值
downto　　指示 for 周期的最终较低值
in　　　　指示要循环迭代的集合
while　　 引入条件重复周期
do　　　　将循环条件与代码分开
repeat　　引入具有最终条件的重复循环
until　　 指示循环的最终条件
with　　　指示要使用的数据结构
function　返回结果的子例程或语句组
procedure 不返回结果的子例程或语句组
inline　　请求编译器用函数的实际代码替换函数调用,以加快执行速度
Overload　允许重用函数或过程的名称

许多其他关键字与类和对象有关：

class　　　　表示 class 类型
object　　　 用于指示较旧的类类型（现已弃用）
abstract　　 未完全定义的类
sealed　　　 其他类无法继承的类
interface　　指示接口类型（也在第一组中列出）
constructor　对象或类的初始化方法
destructor　 对象或类的清理方法
virtual　　　虚拟方法
override　　 修改后的虚拟方法版本
inherited　　引用基类的方法
private　　　无法从外部访问的课程或记录的一部分
protected　　从外部访问受限的部分课程
public　　　 可以从外部完全访问的课程或记录的一部分
published　　专供用户使用的课程的一部分
strict　　　 对私有和受保护部分的更严格限制

property　映射到值或方法的符号
read　用于获取属性值的映射器
write　用于设置属性值的映射器
nil　零对象的值（也用于其他实体）

较小的关键字组用于异常处理（请参见第 11 章）：

try　异常处理块的开始
finally　引入与异常无关的代码
except　介绍异常情况下要执行的代码
raise　用于触发异常

另一组关键字用于运算符，并在本章后面的“表达式和运算符”部分中进行了介绍（除了一些仅在后面的章节中介绍的高级运算符）：

as	and	div
is	in	mod
not	or	shl
shr	xor	

最后，这是其他一些较不常用的关键字的部分列表，其中包括一些您应该避免使用的旧关键字。同样，这只是它们使用的位置的快速指示，在本书中稍后或将予以介绍：

default　指示属性的默认值
dynamic　具有不同实现方式的虚拟方法
export　用于导出的旧关键字，由下面的关键字代替
exports　在 DLL 项目中，列出要导出的函数
external　引用要绑定到的外部 DLL 函数
file　用于旧文件类型，目前很少使用。
forward　指示前向函数声明
goto　跳转到代码特定位置的标签，强烈建议避免 goto。
index　用于索引属性，以及（很少）在导入或导出函数时使用
label　定义 goto 语句跳转到的语句。 强烈建议避免 goto。
message　与平台消息关联的虚拟功能的备用关键字
name　用于映射外部功能
nodefault　指示属性没有默认值
on　用于触发异常
out　var 的替代方法，指示通过引用传递但未初始化的参数
packed　更改记录或数据结构的内存布局
reintroduce　允许重用虚拟函数的名称
requires　在软件包中，表示从属软件包

请注意，近年来，Object Pascal 语言关键字列表几乎没有增加，因为任何其他关键字都暗示着可能在一些现有程序中引入编译错误，从而防止碰巧使用新关键字之一作为符号。最近对该语言的大多数添加都不需要新的关键字，例如泛型和匿名方法。

1.4 程序的结构

尽管我在本章早些时候展示的第一个简单控制台应用程序就是这种情况，但

您几乎不会将所有代码都写在一个文件中。创建可视化应用程序后，您将在项目文件旁边至少获得一个辅助源代码文件。该辅助文件称为单元，由 PAS 扩展名指示（对于 Pascal 源单元），而主项目文件使用 DPR 扩展名（对于 DelphiProject 文件）。这两个文件都包含 Object Pascal 源代码。

Object Pascal 大量使用单元或程序模块。实际上，即使不使用对象，单元也可以用于提供模块化和封装，并且它们充当名称空间。Object Pascal 应用程序通常由几个单元组成，包括托管表单和数据模块的单元。实际上，当您向项目中添加新表单时，IDE 实际上会添加一个新单元，该单元定义了新表单的代码。

单元不需要定义表单；他们可以简单地定义例程并使其可用，或者提供一种或多种数据类型（包括类）。如果将新的空白单元添加到项目中，则它将仅包含用于分隔单元划分的节的关键字：

```
unit Unit1;
interface
implementation
End.
```

上面显示的简单单元的结构包括以下元素：

- 首先，一个单元具有与其文件名相对应的唯一名称（即，上面的样本单元必须存储在文件 Unit1.pas 中）。
- 其次，该单元具有一个接口部分，用于声明其他单元可见的内容。
- 第三，该单元具有一个实现部分，其中包含实现详细信息，实际代码以及可能的其他本地声明，这些在单元外部不可见。

1.4.1 单元和程序名称

如前所述，单元名称必须与该单元文件的名称相对应。

对于程序也是如此。要重命名单元，应使用项目管理器中的“重命名”选项，并且两者将保持同步（在 IDE 中执行“另存为”操作时，您仍保持单元名和文件名同步，但最终得到两个副本）磁盘上文件的大小）。当然，您也可以在文件系统级别更改文件名，但是如果您也没有在单元开头更改声明，则在编译单元时（甚至是在在 IDE 中加载）。如果您在不更改文件名的情况下更改了单元的声明，也会收到以下错误消息示例：

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

（DCC 错误]E1038 单元标识符“Unit3”与文件名不匹配）

该规则的含义是，单元或程序名称必须是有效的 Pascal 标识符，而且必须是文件系统中的有效文件名。例如，在下划线（_）旁边不能包含空格，不能包含特殊字符，如本章前面的标识符部分所述。给定的单元和程序必须使用 Object Pascal 标识符命名，它们会自动产生有效的文件名，因此您不必担心。当然，例外是使用在文件系统级别上不是有效文件名的 Unicode 符号。

Dotted Unit Names（带点“.”的单元名称）

对单元标识符的基本规则进行了扩展：单元名称可以使用点分符号。 因此，以下所有都是有效的单位名称：

```
unit1
myproject.unit1
```

```
mycompany.myproject.unit1
```

按照一般规则，这些单元需要以相同的点名保存在文件中（即，一个名为 MyProject.Unit1 的单元必须存储在 MyProject.Unit1.pas 文件中）。

进行此扩展的原因是单元名称必须唯一，并且随着 Embarcadero 和第三方供应商提供越来越多的单元，这变得越来越复杂。现在，作为产品库一部分提供的所有 RTL 单元和其他各种单元都遵循点分单元名称规则，并带有表示该区域的特定前缀，例如：

- 用于核心 RTL 的系统
- 用于数据库访问等的数据
- 用于 FireMonkey 平台的 FMX，用于台式机和移动设备的单源多设备体系结构
- 用于 Windows 的 Visual Component Library 的 VCL

❖ 通常，您会使用全名来引用带点划线的单元名称，包括库单元。通过在项目选项中设置相应的规则，也可以仅在引用中使用名称的最后一部分（允许与旧代码向后兼容）。此设置称为“单元作用域名称”，它是一个用分号分隔的列表。但是请注意，与使用标准单元名称相比，使用此功能会减慢编译速度。

More on the Structure of a Unit（有关单元结构的更多信息）

除了接口和实现部分之外，一个单元还可以具有一个可选的初始化部分，其中包含一些启动代码，这些程序将在程序首次加载到内存中时执行。如果有初始化部分，则还可以有一个终结部分，在程序终止时执行。

❖ 您还可以在类构造函数中添加初始化代码，这是第 12 章介绍的最新语言功能。使用类构造函数有助于链接程序删除不需要的代码，这就是为什么建议使用类构造函数和类析构函数，而不是使用旧的初始化和构造函数的原因。完成部分。作为历史记录，编译器仍支持使用 begin 关键字代替初始化关键字。在项目源代码中，begin 的类似用法仍然是标准的。

换句话说，一个单元的一般结构及其所有可能的部分和一些示例元素如下所示：

```
unit unitName;
interface
// 我们在接口部分提到的其他单位
uses
  unitA, unitB, unitC;
// 导出的类型定义
type
 newType = TypeDefinition;
//导出常量
const
 Zero = 0;
// 全局变量
var
 Total: Integer;
// 导出函数和程序列表
```

```
procedure MyProc;
implementation
// 我们在实现中提到的其他单位
uses
  unitD, unitE;
// 隐藏的全局变量
var
  PartialTotal: Integer;
// 所有导出的函数必须进行编码
procedure MyProc;
begin
  // ... 程序代码 MyProc
end;
initialization
  // 可选的初始化代码
finalization
  // 可选的清理代码
end.
```

单元接口部分的目的是详细说明该单元包含的内容，并且主程序和将使用该单元的其他单元可以使用该单元。而实现部分包含单元的螺母和螺栓，这些螺母和螺栓对外部观察者是隐藏的。这就是 Object Pascal 甚至不使用类和对象也可以提供所谓的封装的方式。

如您所见，单元的接口可以声明许多不同的元素，包括过程，函数，全局变量和数据类型。数据类型通常使用最多。每次创建可视表单时，IDE 都会自动在单元中放置新的类数据类型。但是，包含表单定义当然不是 Object Pascal 中单位的唯一用途。您可以只有代码单元，具有功能和过程（以传统方式）以及不引用表单或其他可视元素的类。

在接口或实现部分中，类型，变量，常量等的声明可以按任何顺序编写，并且可以重复多次。您可以有一些常量，某些类型，然后有更多常量，其他变量和其他类型部分。唯一的规则是，要引用一个符号，必须在引用该符号之前对其进行声明，这就是您经常需要具有多个节的原因。

The Uses Clause（uses 部分）

接口部分开头的 uses 子句指示我们需要在该单元的接口部分中访问哪些其他单元。这包括定义我们在该单元的数据类型的定义中引用的数据类型的单元，例如在我们定义的表单中使用的组件。

在实现部分的开头，第二个 uses 子句指示我们仅需要在实现代码中访问的其他单元。当您需要从例程和方法的代码中引用其他单元时，应在第二个 use 子句而不是第一个 use 子句中添加元素，因为这会减少依赖关系并增加编译时间。您引用的所有单位必须存在于项目目录或搜索路径的目录中。

✧ 您可以在项目选项中设置项目的搜索路径。系统还考虑“库”路径中的单元，这是 IDE 的全局设置。

C++程序员应注意，uses 语句与 include 指令不对应。use 语句的作用是仅导入列出单元的预编译接口部分。仅当编译该单元时，才考虑该单元的实现部分。

您所指的单位可以是源代码格式（PAS）或编译格式（DCU）。

尽管很少使用，但是 Object Pascal 还具有$INCLUDE 编译器指令，该指令的工作方式与 C/C++include 相似。一些库使用这些特殊的包含文件在多个单元之间共享编译器指令或其他设置，并且通常具有 INC 文件扩展名。该指令将在本章末尾介绍。

- 注意，仅当使用相同版本的编译器和系统库构建对象时，Object Pascal 中的编译单元才兼容。在产品的较早版本中编译的单元通常与编译器的更高版本不兼容。同一发行版中的更新保持兼容性。换句话说，内置于版本 10.3.1 的单元与所有 10.3.x 版本兼容，但与 10.2 或 10.4 版本不兼容。

1.4.2 单元和范围

在 Object Pascal 中，单元是封装和可见性的关键，从这个意义上讲，它们可能比类的 private 和 public 关键字更为重要。标识符的范围（例如变量，过程，函数或数据类型）是代码中可访问或可见标识符的部分。基本规则是，标识符仅在其范围内（即，仅在声明它的单元，函数或过程中才有意义）。您不能在其范围之外使用标识符。

- 直到最近，Object Pascal 才将范围的概念限制为可以包含声明的通用代码块。从 Delphi10.3 开始，您可以在 begin-end 块中声明内联变量，以将变量的范围限制为特定的块，就像 C 或 C++中那样。有关更多详细信息，请参见第 2 章的“变量的生命周期和可见性”。

通常，标识符只有在定义后才可见。该语言中有一些技术允许在其完整定义之前声明一个标识符，但是如果同时考虑定义和声明，则通用规则仍然适用。

但是，考虑到将整个程序写在一个文件中几乎没有什么意义，那么当您使用多个单元时，上述规则将如何改变？简而言之，通过使用 use 语句引用另一个单元，该单元的接口部分中的标识符对于新单元将变为可见。

相反，如果您在单元的接口部分中声明了标识符（类型，函数，类，变量等），则其他引用该单元的模块将可以看到该标识符。如果在一个单元的实现部分中声明了一个标识符，则只能在该单元中使用它（通常称为本地标识符）。

Using Units Like Namespaces（使用像命名空间一样的单元）

我们已经看到，uses 语句是访问在另一个单元范围内声明的标识符的标准技术。此时，您可以访问单元的定义。但是，您引用的两个单元可能会声明相同的标识符；也就是说，您可能有两个具有相同名称的类或两个例程。

在这种情况下，您可以简单地使用单元名称为单元中定义的类型或例程的名称添加前缀。例如，您可以将在给定的 Calc 单位中定义的 ComputeTotal 过程称为 Calc.ComputeTotal。经常不需要这样做，因为如果可以避免的话，强烈建议您不要对同一程序的两个不同元素使用相同的标识符。

但是，如果您查看系统或第三方库，则会发现具有相同名称的函数和类。一个很好的例子是不同用户界面框架的可视控件。当您看到对 TForm 或 TControl 的引用时，它可能表示不同的类，具体取决于所引用的实际单位。

如果您的 uses 语句中的两个单元公开了相同的标识符，则最后一个正在使用的单元中的那个将覆盖该符号，并将成为编译器使用的那个。换句话说，列表中最后一个单元中定义的符号将获胜。如果您根本无法避免这种情况，建议在符

号前加上单位名称，以免您的编码取决于单位的列出顺序。

- ❖ Delphi 开发人员确实利用了具有两个名称相同的类的功能，并使用了一种称为中介程序类的技术，本书稍后将对此进行介绍。

1.4.3 程序文件

如我们所见，Delphi 应用程序由两种源代码文件组成：一个或多个单元以及一个，只有一个程序文件（保存在 DPR 文件中）。可以将这些单位视为辅助文件，这些文件由应用程序的主要部分程序引用。从理论上讲，这是正确的。实际上，程序文件通常是角色有限的自动生成的文件。在可视化应用程序的情况下，只需要启动程序，通常就可以创建并运行主要表单。程序文件的代码可以手动编辑，但也可以通过使用 IDE 的某些“项目选项”（例如与应用程序对象和表单相关的选项）自动进行修改。

程序文件的结构通常比单元的结构简单得多。这是由 IDE 为您自动创建的示例程序文件（省略了一些可选的标准单位）的源代码：

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

如您所见，只有一个 uses 部分和应用程序的主要代码，由 begin 和 end 关键字括起来。该程序的 use 语句特别重要，因为它用于管理应用程序的编译和链接。

- ❖ 程序文件中的单元列表对应于 IDE 项目管理器中属于项目的单元列表。在 IDE 中将单元添加到项目中时，该单元会自动添加到程序文件源的列表中。如果将其从项目中删除，则会发生相反的情况。无论如何，如果您编辑程序文件的源代码，则项目管理器中的单元列表将相应更新。

1.5 编译器指令

如前所述，程序结构的另一个特殊元素（而不是其实际代码）是编译器指令。这些是针对编译器的特殊指令，格式为：

```
{$X+}
```

如上所述，某些编译器指令具有单个字符，带有加号或减号，指示该指令是已激活还是未激活。大多数指令还具有更长的可读版本，并使用 ON 和 OFF 标记它们是否处于活动状态。一些指令仅具有较长的描述性格式。

编译器指令不会直接生成已编译的代码，而是会在遇到该指令后影响编译器如何生成代码。在许多情况下，尽管在某些情况下您只想将特定的编译器设置应用于单元或代码片段，但使用编译器指令可以替代在 IDE 项目选项中更改编译器设置之一。

在讨论它们可能影响的语言功能时，我将介绍相关的特定编译器指令。在本

节中，我只想提及与程序代码流有关的几个指令：条件定义和包含。

1.5.1 条件定义

条件定义使您可以指示编译器包括一部分源代码或将其忽略。在 Delphi 中有两种条件定义。传统的$ IFDEF 和$ IFNDEF 以及更新且更灵活的$ IF。

这些条件定义可以取决于定义的符号，或者对于$IF 版本，取决于常量值。定义的符号可以由系统进行预定义（如编译器和平台符号），可以在特定的项目选项中定义它们，也可以使用另一个编译器指令$DEFINE 将其引入代码中。

传统的$IFDEF 和$IFNDEF 具有以下格式：

```
{$IFDEF TEST}
  // 这将被编译
{$ENDIF}
{$IFNDEF TEST}
  // 这将不会被编译
{$ENDIF}
```

您还可以有两种选择，使用$ ELSE 指令将它们分开。

前面提到过，更新的$ IF 指令更灵活，允许对条件使用常量表达式，例如可以引用代码中任何常量值的比较函数（例如，检查编译器版本是否高于给定值）。$ IF 指令由$ IFEND 指令关闭：

```
{$IF (ProgramVersion > 2.0) }
  ... // 如果条件为真，则执行此代码
{$ELSE}
  ... // 如果条件为假，则执行此代码
{$IFEND}
```

如果您有多个条件，也可以使用$ ELSEIF 指令。

1.5.2 Compiler Versions（编译器版本）

每个版本的 Delphi 编译器都有一个特定的定义，您可以使用该定义来检查是否要针对该产品的特定版本进行编译。如果您使用的是稍后介绍的功能，但要确保代码仍可针对较早版本进行编译，则可能需要执行此操作。

如果需要某些最新版本的 Delphi 的特定代码，则可以将$IFDEF 语句基于以下定义：

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 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	VER300
Delphi 10.1 Berlin	VER310
Delphi 10.2 Tokyo	VER320
Delphi 10.3 Rio	VER330

Delphi 10.4 Sydney	VER340

这些版本号的十进制数字表示实际的编译器版本(例如 Delphi XE5 中的 26)。数字序列不是特定于 Delphi 的，而是可以追溯到 Borland 发布的第一个 Turbo Pascal 编译器（有关更多信息，请参见附录 A）。

您还可以在$ IF 语句中使用内部版本控制常量，其优点是可以使用比较运算符(> =)而不是特定版本的匹配项。版本控制常量称为 CompilerVersion，在 Delphi XE5 中，其分配的浮点值为 26.0。

因此，例如：

```
{$IF CompilerVersion >= 26)}
  // code to compile in Delphi XE5 or later
{$IFEND}
```

类似地，如果您需要一些特定于平台的代码，则可以针对不同的平台使用系统定义（通常是 Object Pascal 中的一个例外，不是常见做法）：

Windows (both 32 and 64 bit)	MSWINDOWS
macOS	MACOS
iOS	IOS
Android	ANDROID
Linux	LINUX

下面是一个代码片段，其中包含基于上面定义的平台的测试，这是 HelloPlatform 项目的一部分：

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

1.5.3 Include Files（包含文件）

我想在这里讨论的另一个指令是$ INCLUDE 指令，在讨论 uses 语句时已经提到过。该指令使您可以在源代码文件的给定位置引用源代码片段并将其包括在内。有时，在代码片段定义编译器指令和编译器直接使用的其他元素的情况下，这通常可以在不同的单元中包含相同的片段。

使用单元时，它只会被编译一次。当您包含文件时，该代码会在添加到其的每个单元中进行编译(这就是为什么通常应该避免在包含文件中声明任何新标识符的原因，如果相同的包含文件嵌入在相同的不同单元中项目。那么如何使用包含文件？

一个很好的例子是您希望在大多数单元中启用的一组编译器指令或一些额外的特殊定义。

大型库通常为此目的使用包含文件，例如 FireDAC 库，该数据库现在是系统库的一部分。

系统 RTL 单元展示的另一个示例是，每个平台使用单个包含，而$IFDEF 用于有条件地仅包含其中一个。

第二章 变量和数据类型

Object Pascal 是一种强类型语言。Object Pascal 中的变量声明为数据类型（或用户定义的数据类型）。变量的类型决定了变量可以保存的值以及可以对其执行的操作。这样，编译器既可以识别代码中的错误，又可以为您生成更快的程序。

这就是为什么 Pascal 中的类型概念比 C 或 C++之类的语言更强的原因。

后来的基于相同语法但破坏了与 C 的兼容性的语言（例如 C＃和 Java）从 C 转向并采用了 Pascal 的强数据类型概念。例如，在 C 语言中，算术数据类型几乎可以互换。相比之下，BASIC 的原始版本没有类似的概念，并且在当今的许多脚本语言（JavaScript 是一个明显的例子）中，数据类型的概念非常不同。

❖ 实际上，有一些技巧可以绕过类型安全性，例如使用变体记录类型。强烈建议不要使用这些技巧，而今天使用的技巧并不多。

正如我提到的那样，从 JavaScript 开始，所有动态语言都没有相同的数据类型概念，或者（至少）对类型的概念非常宽松。在这些语言中的某些语言中，类型是根据您分配给变量的值来推断的，并且变量类型可以随时间变化。需要指出的是，数据类型是在编译时增强大型应用程序正确性的关键元素，而不是依赖于运行时检查。数据类型需要更多的顺序和结构，并对您要编写的代码进行一些计划，这显然有其优点和缺点。

❖ 不用说我更喜欢强类型语言，但是无论如何，我在本书中的目标是解释语言的工作原理，而不是倡导为什么我认为它是一种很棒的编程语言。尽管我相信您在阅读文字时会得到那种印象。

2.1 变量和赋值

像其他强类型语言一样，Object Pascal 要求在使用所有变量之前声明所有变量。 每次声明变量时，都必须指定一种数据类型。以下是一些变量声明：

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

var 关键字可以在程序中的多个位置使用，例如在函数或过程开始时，以声明该代码部分的局部变量，或者在一个单元内声明全局变量。

❖ 稍后我们将看到，Delphi 的最新版本增加了声明变量“内联”的功能，该变量被混合到编程语句中。这与经典的 Pascal 有很大的不同。

var 关键字之后是变量名列表，后跟冒号和数据类型名称。您可以在一行上写多个变量名，如上一个代码段的最后一条语句中的 A 和 B（与将声明分为两行相比，今天的编码风格现在不那么普遍了：使用独立的行有助于（具有可读性，版本比较和合并）。

一旦定义了给定类型的变量，就只能在其数据类型上执行支持的操作。例如，您可以在测试中使用布尔值，而在数值表达式中使用整数值。例如，您不能在不调用特定转换函数的情况下混合布尔值和整数，也不能混合任何不兼容的数据类型（即使内部表示形式在物理上是兼容的，布尔值和整数也是如此）。

最简单的分配是实际的特定值。例如，您可能希望 Value 变量保存数字值 10。但是，如何表达文字值呢？虽然这个概念可能很明显，但是值得对其进行一些详细研究。

2.1.1 文字值

文字值是您直接在程序源代码中键入的值。 如果您需要一个值为 20 的数字，则可以简单地写：

```
20
```

还有一个基于十六进制值的相同数值的替代表示形式，例如：

```
$14
```

这些将是整数（或可用的各种整数类型之一）的文字值。如果要使用相同的数值，但要使用浮点文字，则通常在其后添加一个十进制小数：

```
2.0
```

文字值不限于数字。您也可以使用字符和字符串。

两者都使用单引号（如果许多其他编程语言都将双引号或字符使用单引号，而字符串则使用双引号）：

```
// 文字字符
'K'
#55
// 文字字符串
'Marco'
```

如上所示，您还可以通过字符的相应数字值（原来是 ASCII 码，现在是 Unicode 代码点值）来指示字符，并在数字前加上＃符号，如＃32（用于空格）。这对于在源代码中没有文本表示形式的控制字符（例如，退格或制表符）很有用。

如果需要在字符串中加引号，则必须将其加倍。因此，如果我想使用自己的名字和姓氏（用最后引号而不是重音拼写），我可以写：

```
'My name is Marco Cantu'''
```

两个引号代表字符串中的引号，而第三个连续引号则标记字符串的结尾。还要注意，字符串文字必须写在一行上，但是您可以使用+号将多个字符串文字串联起来。如果要在字符串中使用换行符或 line break（换行），请不要在两行上写它，而要用 sLineBreak 系统常数（特定于平台）将这两个元素连接起来，如下所示：

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

2.1.2 赋值语句

Object Pascal 中的分配使用冒号等于运算符（:=），这是习惯于其他语言的程序员的奇怪表示法。Object Pascal 中的=运算符（用于许多其他语言的赋值）用于测试是否相等。

- ✓ <u>:=运算符来自 Pascal 的前身 Algol，这是当今很少有开发人员听说过的语言（甚至使用过）。当今大多数语言都避免使用:=表示法，而赞成=赋值表示法。</u>

通过使用不同的符号进行赋值和相等性测试，Pascal 编译器（如 C 编译器）可以更快地翻译源代码，因为它不需要检查使用运算符确定其含义的上下文。使

用不同的运算符还使人们更易于阅读代码。TrulyPascal 选择了两个不同于 C 的运算符（以及 Java，C＃，JavaScript 之类的语法派生），它们使用=进行赋值，并使用==进行相等性测试。

❖ 为了完整起见，我应该提到 JavaScript 还有一个===运算符（执行严格的类型和值相等性测试），但是即使是大多数 JavaScript 程序员也对此感到困惑。

赋值的两个元素通常称为左值（lvalue）和右值（rvalue），左值（要分配给变量或内存的位置）和右值（指的是表达式的值）。尽管右值可以是表达式，但左值必须（直接或间接）引用您可以修改的存储位置。有些数据类型具有特定的分配行为，我将在适当的时候进行介绍。

另一个规则是，左值和右值的类型必须匹配，或者两者之间必须有自动转换，如下一节所述。

2.1.3 分配和转换

使用简单的分配，我们可以编写以下代码（您可以在 VariablesTest 项目的本节中找到许多其他代码段）：

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

给定前面的变量声明，这三个分配是正确的。相反，下一条语句是不正确的，因为两个变量具有不同的数据类型：

```
value := IsCorrect; //  错误
```

当您键入此代码时，Delphi 编辑器会立即显示一个红色的花体，指示错误并带有正确的描述。如果尝试对其进行编译，则编译器会发出相同的错误，并给出如下描述：

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

（[dcc32 错误]：E2010 不兼容的类型：“整数”和“布尔”）

编译器会通知您代码中存在错误，即两种不兼容的数据类型。当然，通常可以将变量的值从一种类型转换为另一种类型。在某些情况下，转换是自动的，例如，如果您将整数值分配给浮点变量（当然不是相反）。通常，您需要调用一个特定的系统函数来更改数据的内部表示。

2.1.4 初始化全局变量

对于全局变量，可以在声明变量时使用下面的常量赋值表示法（=）代替赋值运算符（:=）来赋初始值。例如，您可以编写：

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

此初始化技术仅适用于全局变量，而全局变量无论如何都被初始化为其默认值（例如对于数字为零）。

2.1.5 初始化局部变量

相反，在过程或函数开头声明的变量不会初始化为默认值，并且没有赋值语法。 对于这些变量，通常值得在代码的开头添加显式的初始化代码：

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

同样，如果您不初始化局部变量而是按原样使用它，则该变量将具有完全随机的值（取决于存在于该内存位置的字节）。在某些情况下，编译器会警告您潜在的错误，但并非总是如此。

换句话说，如果您写：

```
var
  Value: Integer;
begin
  ShowMessage (Value.ToString); // 值未定义
```

输出将是一个完全随机的值，无论 Value 变量的存储位置上被视为 Integer 的字节是多少。

2.1.6 内联变量

Delphi 的最新版本（从 10.3 Rio 开始）具有一个附加概念，该概念改变了自早期 Pascal 和 Turbo Pascal 编译器以来使用的变量声明方法：内联变量声明。

新的内联变量声明语法使您可以直接在代码块中声明变量（与传统变量声明一样，也允许使用多个符号）：

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

尽管这可能看起来有限，但此更改有一些副作用，与初始化，类型推断和变量的生存期有关。 我将在本章的后续部分中介绍这些内容。

初始化内联变量

与旧的声明模型相比，第一个重大变化是可以在单个语句中完成内联声明和变量的初始化。与在函数开头初始化多个变量相比，这使事情更具可读性，如上一节所述：

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

这里的主要优点是，如果变量的值仅在代码块的后面才可用，而不是设置初始值（如 0 或 nil），然后再分配实际值，则可以将变量声明延迟到该点 您可以

在其中计算适当的初始值，例如下面的 K：

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

换句话说，在过去，所有局部变量在整个代码块中都可见，而现在，内联变量仅在其声明位置到代码块末尾才可见。换句话说，在为其分配适当的值之前，不能在代码块的前两行中使用像 K 这样的变量。

2.1.7 内联变量的类型推断

内联变量的另一个显着优点是，编译器现在可以在几种情况下通过查看表达式的类型或分配给它的值来推断内联变量的类型。这是一个非常简单的示例：

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

分析右值表达式的类型（即，:=之后的值）以确定变量的类型。通过分配字符串常量，变量将是字符串类型，但是如果分配函数或复杂表达式的结果，则会进行相同的分析。

请注意，变量仍然是强类型，因为编译器确定数据类型并在编译时分配它，并且不能通过分配新值来对其进行修改。类型推断只是一种便利，它可以减少代码的键入（某些与复杂数据类型相关的事情，例如复杂的泛型类型），但不会更改语言的静态和强类型化性质，也不会导致运行时速度变慢。

推断类型时，某些数据类型将“扩展”为较大的类型，如上述将数值 22（ShortInt）扩展为 Integer 的情况一样。通常，如果右侧表达式类型是整数类型且小于 32 位，则该变量将声明为 32 位 Integer。如果需要特定的，较小的数字类型，则可以使用显式类型。

2.1.8 Constants（常数）

Object Pascal 还允许声明常量。这使您可以为程序执行期间不会更改的值赋予有意义的名称（并可能通过不在已编译代码中复制常量值来减小大小）。

要声明一个常量，您无需指定数据类型，而只需分配一个初始值。编译器将查看该值并自动推断正确的数据类型。以下是一些示例声明（也来自 VariablesTest 应用程序项目）：

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

编译器根据其值确定常量数据类型。在上面的示例中，千常量被假定为 SmallInt 类型，这是可以容纳它的最小整数类型。如果要告诉编译器使用特定类型，则可以简单地将类型名称添加到声明中，如下所示：

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

- 程序中的假设通常是不好的，并且编译器可能会随着时间的变化而与开发人员的假设不符。如果您可以在没有假设的情况下更清晰地表达代码，那就去做吧！

当声明一个常量时，编译器可以选择是为该常量分配一个内存位置并在其中保存其值，还是每次使用该常量时都复制实际值。第二种方法特别适用于简单常量。

声明常量后，就可以像使用其他变量一样使用它了，但是不能为它赋一个新值。如果尝试尝试，则会出现编译器错误。

- 奇怪的是，Object Pascal 确实允许您在运行时更改类型常量的值，就好像它是一个变量一样，但是仅当您启用$J 编译器指令或使用相应的 Assignable 类型类型常量编译器选项时。包含此可选行为是为了向后兼容使用旧编译器编写的代码。显然，这不是建议的编码风格，我在本说明中将其作为有关此类编程技术的历史轶事加以介绍。

内联常数

正如我们之前在变量中看到的那样，现在还可以内联一个常量值声明。可以将其应用于类型常量或非类型常量，在这种情况下，可以推断类型（很长时间以来对常量可用的功能）。

下面是一个简单的示例：

```
begin
  // 一些代码
  const M: Integer = (L + H) div 2; // 具有类型说明符的标识符
  // 一些代码
  const M = (L + H) div 2; // 没有类型说明符的标识符
```

请注意，虽然常规常量声明只允许您分配常量值，但是对于内联常量声明，您可以使用任何表达式。

Resource（资源）字符串常量

尽管这是一个稍微高级的主题，但是当您定义字符串常量时，您可以使用特定的指令 resourcestring 代替编写标准常量声明，该指令向编译器和 Linker（链接器）指示将字符串视为 Windows 资源（或 Windows 资源）。 非 Windows 平台上 Object Pascal 支持的等效数据结构）：

```
const
  AuthorName = 'Marco';
resourcestring
  StrAuthorName = 'Marco';
begin
  ShowMessage (StrAuthorname);
```

在这两种情况下，您都定义了一个常数。也就是说，您在程序执行期间不会

更改的值。区别仅在于内部实现。用 resourcestring 伪指令定义的字符串常量存储在程序的资源中的字符串表中。

简而言之，使用资源的优点是 Windows 执行更有效的内存处理，用于其他平台的相应 Delphi 实现以及无需修改源代码即可对程序进行本地化（将字符串转换为其他语言）的更好方法。根据经验，对于显示给用户并且可能需要翻译的任何文本，都应使用 resourcestring，对于其他内部程序字符串，例如固定配置文件名，则应使用内部常量。

✧ IDE 编辑器具有自动重构功能，可用于将代码中的字符串常量替换为相应的 resourcestring 声明。将编辑光标放在字符串文字中，然后按 Ctrl + Shift + L 激活此重构。

2.1.9 变量的生命周期和可见性

根据定义变量的方式，它将使用不同的内存位置并在不同的时间段内保持可用状态（通常称为变量生存期），并且将在代码的不同部分中使用（此功能称为“可见性”）。

现在，我们无法在本书的开头部分就所有选项进行完整的描述，但是我们当然可以考虑最相关的情况：

- 全局变量：如果在单元的接口部分中声明变量（或任何其他标识符），则其范围将扩展到使用声明该变量的任何其他单元。该变量的内存在程序启动后立即分配，并一直存在直到终止。您可以为其分配默认值，也可以使用单元的初始化部分，以防以更复杂的方式计算初始值。
- 全局隐藏变量：如果在单元的实现部分中声明变量，则不能在该单元之外使用该变量，但可以从声明位置开始在该单元中定义的任何代码块和过程中使用它。这样的变量使用全局内存，并且与第一组具有相同的生存期。唯一的区别在于可见性。初始化与全局变量的初始化相同。
- 局部变量：如果在定义函数，过程或方法的块内声明变量，则不能在该代码块外使用此变量。标识符的范围涵盖了整个函数或方法，包括嵌套例程（除非嵌套例程中具有相同名称的标识符隐藏了外部定义）。当程序执行定义该变量的例程时，将在堆栈上分配该变量的内存。一旦例程终止，堆栈中的内存就会自动释放。
- 局部内联变量：如果在定义函数，过程或方法的块中声明了内联变量，则与传统局部变量相比，另一个限制是可见性从声明该变量的那一行开始，一直到函数或方法结束为止。
- 块作用域内联变量：如果在代码块（即嵌套的 begin-end 语句）中声明内联变量，则变量的可见性将限于该嵌套块。这与大多数其他编程语言中发生的情况相匹配，但是在 Object Pascal 中仅使用最新的内联变量声明语法进行了介绍。请注意，在块中声明的内联变量不能使用与在父代码块中声明的变量相同的标识符。

❖ 对于块作用域内联变量，不仅可见性而且变量生存期都限于该块。托管数据类型（如接口，字符串或托管记录）将被放置在子块的末尾，而不是过程或方法的末尾。对于创建用于保存表达式结果的临时变量也是如此。

单元接口部分中的任何声明都可以从在 using 子句中包含该单元的程序的任何部分访问。表单类的变量以相同的方式声明，因此您可以从任何其他表单的代

码中引用表单（及其公共字段，方法，属性和组件）。当然，将所有内容都声明为全局是很差的编程习惯。除了明显的内存消耗问题外，使用全局变量还会使程序难以维护和更新。简而言之，应使用尽可能少的全局变量。

2.2 Data Types（数据类型）

在 Pascal 中，有几种预定义的数据类型，可以将其分为三类：序数类型，实数类型和字符串。我们将在以下各节中讨论序数和实数类型，而字符串将在第 6 章中专门介绍。

Delphi 还包括非类型化的数据类型，称为变量，以及其他“灵活”类型，例如 TValue（增强的 RTTI 支持的一部分）。其中一些更高级的数据类型将在第 5 章中讨论。

2.2.1 序数和数字类型

顺序类型基于顺序或序列的概念。您不仅可以比较两个值以查看哪个值更高，还可以要求任何值的下一个或上一个值，并计算数据类型可以表示的最低和最高值。

三种最重要的预定义序数类型是 Integer，Boolean 和 Char（字符）。但是，还有其他一些具有相同含义但内部表示形式不同且支持不同值范围的相关类型。

下表列出了用于表示数字的序数数据类型：

Size	Signed（有符号）	Unsigned（无符号）
8 bits	ShortInt: -128 to 127	Byte: 0 to 255
16 bits	SmallInt: -32768 to 32767 (-32K to 32K)	Word: 0 to 65,535 (0 to 64K)
32 bits	Integer: -2,147,483,648 to 2,147,483,647 (-2GB to +2GB)	Cardinal: 0 to 4,294,967,295 (0 to 4 GB)
64 bits	Int64: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	UInt64: 0 to 18,446,744,073,709,551,615 (if you can read it!)

如您所见，这些类型对应于数字的不同表示形式，具体取决于用于表示该值的位数，以及符号位的存在与否。有符号的值可以是正数或负数，但取值范围较小（对应的无符号值的一半），因为少了一位可用于存储值本身。

Int64 类型表示最多 18 位的整数。运行时库的某些序数类型例程（例如 High 和 Low），数字例程（例如 Inc 和 Dec）以及字符串转换例程（例如 IntToStr）完全支持此类型。

Aliased Integer Types（别名整数类型）

如果您很难记住 ShortInt 和 SmallInt 之间的区别（包括实际上更小的那个），而不是实际类型，则可以使用在 System 单元中声明的预定义别名之一：

```
type
  Int8 = ShortInt;
  Int16 = SmallInt;
  Int32 = Integer;
```

```
UInt8 = Byte;
UInt16 = Word;
UInt32 = Cardinal;
```

同样，这些类型没有添加任何新内容，但可能更易于使用，因为记住 Int16 而不是 SmallInt 的实际实现很容易。对于来自 C 和其他使用类似类型名称的语言的开发人员，这些类型别名也更易于使用。

整数类型，64 位，NativeInt 和 LargeInt

在 Object Pascal 的 64 位版本中，您可能会惊讶地发现 Integer 类型仍然是 32 位。之所以如此，是因为这是在 CPU 级别进行数字处理的最有效的类型。

它是 Pointer 类型（稍后将详细介绍指针）和其他相关的 64 位引用类型。如果需要适合指针大小和本机 CPU 平台的数字类型，则可以使用两种特殊的 NativeInt 和 NativeUInt 别名类型。它们与特定平台上的指针大小相同（即，在 32 位平台上为 32 位，在 64 位平台上为 64 位）。

对于 LargeInt 类型，情况略有不同，该类型通常用于映射到本机平台 API 函数。在 32 位平台和 Windows32 位上为 32 位，而在 64 位 ARM 平台上为 64 位。除非您特别需要本机代码以使其适合基础操作系统的方式，否则最好不要使用它。

整数类型 Helper（助手）

尽管 Integer 类型与 Object Pascal 语言中的对象分开对待，但是可以使用您使用“点符号”应用的操作对这些类型的变量（和常量值）进行操作。这是通常用于将方法应用于对象的表示法。

从技术上讲，对本机数据类型的这些操作是使用“内部记录助手”定义的。

第 12 章介绍了类和记录助手。简而言之，您可以自定义适用于核心数据类型的操作。专家开发人员会注意到，类型操作在匹配的内部记录帮助器中定义为类静态方法。

您可以从 IntegersTest 演示摘录的以下代码中看到几个示例：

```
var
  N: Integer;
begin
  N := 10;
  Show (N.ToString);
  // 显示一个常数
  Show (33.ToString);
  // 类型操作，显示存储类型所需的字节
  Show (Integer.Size.ToString);
```

❖ 此代码段中使用的 Show 函数是一个简单的过程，用于显示备忘录控件中的某些字符串输出，以避免必须关闭多个 ShowMessage 对话框。另一个好处是，这种方法使复制输出和粘贴文本更加容易（就像我在下面所做的那样）。您会在本书的大多数演示中看到使用这种方法。

该程序的输出如下：

```
10
33
4
```

鉴于这些操作非常重要（比运行时库中的其他操作更重要），因此值得在此

处列出：

ToString	使用十进制格式将数字转换为字符串
ToBoolean	转换为布尔型
ToHexString	使用十六进制格式转换为字符串
ToSingle	转换为单浮点数据类型
ToDouble	转换为双浮点数据类型
ToExtended	转换为扩展浮点数据类型

第一个和第三个运算使用十进制或十六进制运算将数字转换为字符串。第二个是到布尔的转换，而后三个是到稍后描述的浮点类型的转换。

您可以将其他操作应用于 Integer 类型（以及大多数其他数字类型），例如：

Size	存储此类型变量所需的字节数
Parse	将字符串转换为它表示的数字值，如果该字符串不表示数字，则会引发异常
TryParse	尝试将字符串转换为数字

标准序数例程

除了上面列出的 Integer 类型助手定义的操作之外，还有一些标准和“classic（经典）”函数可应用于任何序数类型（不仅是数字类型）。一个典型的示例是使用 SizeOf，High 和 Low 函数来询问有关类型本身的信息。SizeOf 系统函数的结果（可以应用于语言的任何数据类型）是一个整数，指示表示给定类型的值所需的字节数（就像上面显示的 Sizehelper 函数一样）。下表显示了对序数类型的运算：

Dec	将作为参数传递的变量减 1 或可选的第二个参数的值
Inc	将作为参数传递的变量加 1 或指定值
Odd	如果参数为奇数，则返回 True。要测试偶数，应使用一个 not 表达式（not Odd）
Pred	按照数据类型（前身）确定的顺序返回参数之前的值
Succ	返回参数后的值，后继
Ord	返回一个数字，该数字指示参数在数据类型的值集中的顺序（用于非数字序数类型）
Low	返回作为参数传递的序数类型范围内的最小值
High	返回序数数据类型范围内的最大值

❖ C 和 C++程序员应注意，带有一个或两个参数的 Inc 过程的两个版本分别对应于++和+=运算符（Dec 过程对应于-和-=运算符）。与 C 和 C++编译器相似，Object Pascal 编译器优化了这些递增和递减操作。但是，Delphi 仅提供递增前和递减前的版本，而不提供递后和递减后的版本，并且该操作不会返回任何值。

请注意，其中一些例程由编译器自动评估并替换为其值。例如，如果调用 High(X)，其中 X 定义为 Integer，则编译器将表达式替换为 Integer 数据类型的最大可能值。

在 IntegersTest 应用程序项目中，我添加了一个包含以下一些序数类型函数的事件：

```
var
  N UInt16;
begin
  N := Low (UInt16);
  Inc (N);
  Show (N.ToString);
```

```
Inc (N, 10);
Show (N.ToString);
if Odd (N) then
Show (N.ToString + ' is odd');
```

这是您应该看到的输出：

```
1
11
11 is odd
```

您可以将数据类型从 UInt16 更改为 Integer 或其他顺序类型，以查看输出如何变化。

超出范围的操作

像上面的 n 这样的变量只有有限范围的有效值。如果您为其分配的值是负数或太大，则会导致错误。超出范围的操作实际上会遇到三种不同类型的错误。

第一种错误类型是编译器错误，如果您分配的常量值（或常量表达式）超出范围，则会发生该错误。例如，如果您添加到上面的代码：

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

编译器将发出错误：

[dcc32 Error] E1012 Constant expression violates subrange bounds

（[dcc32 错误] E1012 常数表达式违反了子范围限制）

第二种情况发生在编译器无法预测错误情况时，因为它取决于程序流。假设我们用同一段代码编写：

```
Inc (N, High (N));
Show (N.ToString
```

编译器不会触发错误，因为有函数调用，并且编译器事先也不知道其作用（并且错误也取决于 n 的初始值）。在这种情况下，有两种可能性。默认情况下，如果编译并运行该应用程序，则变量中的值将完全不合逻辑（在这种情况下，该操作将导致减 1！）。这是最糟糕的情况，因为没有错误，但是程序不正确。

您可以做的（强烈建议这样做）是打开一个名为“溢出检查”（{$ Q +}或{$ OVERFLOWCHECKS ON}）的编译器选项，它将防止类似的溢出操作并引发错误，在这种情况下为“整数溢出”。

2.2.2 Boolean

逻辑“True”和“False”值使用 Boolean 类型表示。这也是条件语句中条件的类型，我们将在下一章中看到。

Boolean 类型只能具有两个可能的值 True 和 False 之一。

- 为了与 Microsoft 的 COM 和 OLE 自动化兼容，数据类型 ByteBool，WordBool 和 LongBool 用-1 表示值 True，而值 False 仍为 0。同样，通常应忽略这些类型并避免所有低级 布尔运算和数字映射，除非绝对必要。

与 C 语言及其某些派生语言不同，布尔值是 Object Pascal 中的枚举类型，没有直接转换为表示布尔值变量的数值，并且您不应通过尝试将布尔值转换为布尔值来滥用直接类型强制转换 一个数值。但是，布尔类型帮助器确实包含函数 ToInteger 和 ToString。我将在本章后面介绍枚举类型。

请注意，使用 ToString 返回具有布尔变量数字值的字符串。或者，您可以使用 BoolToStr 全局函数，将第二个参数设置为 True，以指示对输出使用布尔字符

串（“True”和“False”）。（有关示例，请参见下面的“字符类型操作”部分。）

2.2.3 Characters

字符变量是使用 Char 类型定义的。与旧版本不同，今天的语言使用 Char 类型表示双字节 Unicode 字符（也由 WideChar 类型表示）。

❖ Delphi 编译器仍然提供了一个字节的 ANSI 字符的 AnsiChar 和 Unicode 字符的 WideChar 之间的区别，其中 Char 类型定义为后者的别名。建议重点是 WideChar，并将 Byte 数据类型用于单字节元素。但是，的确，从 Delphi 10.4 开始，AnsiChar 类型已在所有编译器和平台上提供，以更好地与现有代码兼容。

有关 Unicode 字符的介绍，包括代码点的定义和代理对的定义（以及其他高级主题），您可以阅读第 6 章。在本节中，我将仅关注 Char 类型的核心概念。

正如我在前面提到的字面值时所提到的，常量字符可以用其符号表示法（如“k”）或数字表示法（如＃78）来表示。后者也可以使用 Chr 系统函数表示，如 Chr(78)所示。

可以使用 Ord 函数完成相反的转换。通常，在表示字母，数字或符号时，最好使用符号符号。

当引用特殊字符时，例如＃32 下的控制字符，通常会使用数字符号。以下列表包括一些最常用的特殊字符：

#8	backspace（退格键）
#9	tab（制表符）
#10	new line（换行）
#13	carriage return（回车键）
#27	escape（退出键）

Char Type Operations（字符类型操作）

与其他序数类型一样，Char 类型具有一些预定义的操作，您可以使用点符号将其应用于该类型的变量。再次使用内部记录助手来定义这些操作。

但是，使用情况完全不同。首先，要使用此功能，您需要通过在 uses 语句中引用字符单元来启用它。其次，Char 类型的帮助程序包括几个十二个 Unicode 特定的操作，而不是几个转换函数，这些操作包括 IsLetter，IsNumber 和 IsPunctuation 等测试，以及 ToUpper 和 ToLower 等转换。这是从 CharsTest 应用程序项目中获取的示例：

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

此代码的输出是：

```
True
A
```

❖ Char 类型帮助器的 ToUpper 操作完全启用了 Unicode。这意味着，如果传递

带重音的字母（如ù），则结果将为Ù。一些传统的 RTL 功能不是很聪明，只能用于纯 ASCII 字符。到目前为止，尚未对其进行修改，因为相应的 Unicode 操作要慢得多。

Char as an Ordinal Type（字符作为序数类型）

Char 数据类型非常大，但仍然是序数类型，因此您可以在其上使用 Inc 和 Dec 函数（以获取下一个或上一个字符或以给定数量的元素向前移动，如我们在并在“标准序数类型例程”一节中编写并使用 Char 计数器编写 for 循环（下一章中的 for 循环更多信息）。

这是一个用于显示一些字符的简单片段，可通过从起点开始增加值来获得：

```
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)
```

CharsTest 应用程序项目的 for 循环向字符串添加了很多文本，从而使输出相当长。它从以下几行开始：

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

Converting with Chr（用 Chr 转换）

我们已经看到，有一个 Ord 函数可以返回字符的数字值（更具体地说是 Unicode 代码点）。还有一个相反的函数可用于获取与代码点相对应的字符，即 Chr 特殊函数。

32-bit Characters（32 位字符）

尽管默认的 Char 类型现在已映射到 WideChar，但值得注意的是 Delphi 还定义了 4 字节字符类型 UCS4Char，在 System 单元中定义为：

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

此类型定义和对应的 UCS4String（定义为 UCS4Char 数组）很少使用，但它们是语言运行时的一部分，并在 Character 单元的某些功能中使用。

2.2.4 Floating Point Types（浮点类型）

尽管可以用一组序数值来表示各种整数，但浮点数不是序数（它们具有顺序的概念，但没有元素序列的概念），并由某个近似值表示，但存在一些误差在他们的代表。

浮点数有多种格式，具体取决于用于表示浮点数的字节数和近似值的质量。以下是 Object Pascal 中的浮点数据类型的列表：

Single	最小的存储大小由单个数字给出，单个数字以 4 字节的值实现。该名称表示单个精度浮点值，而其他语言中的名称 float 表示相同的类型。
Double	这些是用 8 个字节实现的浮点数。该名称表示双精度浮点值，并且由许多语言共享。Double 精度是最常用的浮点数据类型，也是旧的 Pascal 类型的别名，称为 Real。
Extended	这些数字是在原始 DelphiWin32 编译器中以 10 个字节实现的，但是这种类型并非在所有平台上都可用（在某些平台上，例如 Win64，它会还原为 Double，而在 macOS 上是 16 字节）。其他编程语言将此数据类型称为 long double。

这些都是精度不同的浮点数据类型，它们对应于 IEEE 标准浮点表示形式，并由 CPU（或更确切地说，由 FPU 浮点单元）直接支持，以实现最大速度。

您还可以使用两种特殊的非普通数字数据类型来表示数字，而不是近似值：

Comp	描述使用 8 个字节或 64 位（可以容纳 18 个十进制数字的数字）的非常大的整数。与相应的浮点值不同，此想法是在不损失精度的情况下表示大量数字。
Currency	指示具有四个十进制数字的定点十进制值，并且与 Comp 类型相同的 64 位表示形式。顾名思义，已添加了 Currency 数据类型以处理非常精确的货币值，并保留小数点后四位（同样，在计算中不会损失精度）。

所有这些非常规数据类型都不具有 High，Low 或 Ord 函数的概念。实类型在理论上表示无穷多个数字。序数类型表示一组固定的值。

为什么浮点值不同

让我进一步解释。当您拥有整数 23 时，您可以确定以下值。整数是有限的（它们具有确定的范围并且具有阶数）。浮点数即使在很小的范围内也是无限的，并且没有顺序：实际上，在 23 到 24 之间有多少个值？哪个数字跟随 23.46？是 23.47、23.461 还是 23.4601？这真的是不可能知道的！

因此，尽管在 Char 数据类型的范围内询问字符“w”的顺序位置是有意义的，但在浮点范围内询问有关 7143.1562 的相同问题根本没有任何意义。数据类型。尽管您确实可以知道一个实数是否具有比另一个更高的值，但是询问在给定数之前存在多少个实数毫无意义（这是 Ord 函数的含义）。

浮点值背后的另一个关键概念是它们的实现不能精确表示所有数字。通常情况下，您希望计算结果是一个特定数字（有时是整数），实际上可能是它的近似值。考虑以下代码，该代码来自 FloatTest 应用程序项目：

```
var
  S1: Single;
begin
  S1 := 0.5 * 0.2;
  Show (S1.ToString);
```

您可能希望结果为 0.1，而实际上您会得到类似 0.100000001490116 的信息。这接近期望值，但不完全是期望值。不用说，如果对结果进行四舍五入，您将获得期望值。如果使用 Double 变量，则输出将为 0.1，如 FloatTest 应用程序项目所示。

❖ <u>现在，我没有时间在计算机上深入讨论浮点数学，因此，我将讨论范围缩短</u>

了一些，但是如果您从 Object Pascal 语言的角度对这个主题感兴趣，我可以向您推荐已故的鲁迪•韦尔特胡斯（Rudy Velthuis）的出色文章，网址为 http://rvelthuis.de/articles/articles-floats.html。

浮动助手和 Math 单元

从上面的代码片段中可以看到，浮点数据类型还具有记录助手，使您可以将操作直接应用于变量，就好像它们是对象一样。实际上，浮点数的操作列表实际上很长。

这是 Single 类型实例上的操作列表（一些操作从它们的名称中可以明显看出，而其他一些则更隐蔽，您可以在文档中查找）：

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

运行时库还具有一个 Math 单元，该单元定义了高级数学例程，涵盖了三角函数（例如 ArcCosh 函数），财务（例如 InterestPayment 函数）和统计信息（例如 MeanAndStdDev 过程）。这些例程很多，其中一些对我来说听起来很陌生，例如 MomentSkewKurtosis 函数（我会让您知道这是什么）。

System.Math 单元功能非常丰富，但是您还会发现许多用于 Object Pascal 的数学函数的外部集合。

2.3 简单的用户定义数据类型

与类型的概念一起，尼克劳斯•沃思（NiklausWirth）在最初的 Pascal 语言中引入的一个好主意是能够在程序中定义新的数据类型(我们今天已经习以为常了，但当时还不明显）。您可以通过类型定义来定义自己的数据类型，例如子范围类型，数组类型，记录类型，枚举类型，指针类型和集合类型。用户定义的最重要的数据类型是类，这是该语言的面向对象功能的一部分，将在本书的第二部分中介绍。

如果您认为类型构造函数在许多编程语言中都很常见，那么您是对的，但是 Pascal 是第一种以正式且非常精确的方式介绍该思想的语言。Object Pascal 仍然具有一些相当独特的功能，例如子范围的定义，枚举和集合，将在以下各节中介绍。

第 5 章介绍了更复杂的数据类型构造函数（如数组和记录）。

2.3.1 命名与未命名类型

可以为用户定义的数据类型命名，以备后用或直接应用于变量。Object Pascal 中的约定是使用字母 T 前缀来表示任何数据类型，包括类但不限于它们。我强烈建议您坚持这一规则，即使您一开始来自 Java 或 C# 背景，即使一开始可能感觉不自然。

为类型命名时，必须在程序的“类型”部分中进行命名（您可以在每个单元中添加任意数量的类型）。以下是一些类型声明的简单示例：

```
type
  // 子范围定义
  TUppercase = 'A'..'Z';
  // 枚举类型定义
  TMyColor = (Red, Yellow, Green, Cyan, Blue, Violet);
  // 集合定义
  TColorPalette = set of TMyColor;
```

使用这些类型，您现在可以定义一些变量：

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

在上述情况下，我使用的是命名类型。或者，可以使用类型定义直接定义变量，而无需使用显式的类型名称，如以下代码所示：

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

通常，应该避免像上面的代码那样使用未命名的类型，因为您不能将它们作为参数传递给例程或声明相同类型的其他变量。考虑到该语言最终诉诸于类型名称对等而不是结构类型对等，因此对每种类型使用单一定义确实很重要。还要记住，单元的接口部分中的类型定义可以通过 use 语句在任何其他单元的代码中看到。

上面的类型定义是什么意思？我将为不熟悉传统 Pascal 类型构造的用户提供一些说明。我还将尝试强调与其他编程语言中相同构造的区别，因此无论如何您可能有兴趣阅读以下部分。

2.3.2 类型别名

如我们所见，Delphi 语言在检查类型兼容性时使用类型名称（而不是其实际定义）。具有不同名称的两个相同定义的类型是两种不同类型。

当您定义类型别名（基于现有类型的新类型名称）时，也是如此。令人困惑的是，同一语法有两种变体，它们产生的效果略有不同。在 TypeAlias 示例中查看以下代码：

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

这两个新类型都保持与 Integer 类型（通过自动转换）兼容的分配，但是 TNewInt2 类型将不完全匹配，例如，不能将其作为引用参数传递给需要别名类型的函数：

```
procedure Test (var N: Integer);
begin
end;

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

```
    Test (NI);
    Test (NI2); // error
```

最后一行产生错误：

E2033 Types of actual and formal var parameters must be identical

（E2033 实际和形式 var 参数的类型必须相同）

类型助手会发生类似的情况，因为 Integer 类型助手可以用于 newint，但不能用于 TNewInt2。在讨论记录助手时，这将在后面的部分中专门介绍。

2.3.3 子范围类型

子范围类型定义了另一个类型范围内的值范围（因此称为子范围）。例如，您可以定义 Integer 类型的子范围，从 1 到 10 或从 100 到 1000，或者您可以定义 Char 类型的子范围（仅用英文大写字符），如下所示：

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

在子范围的定义中，您无需指定基本类型的名称。

您只需要提供该类型的两个常量即可。原始类型必须是序数类型，而结果类型将是另一序数类型。将变量定义为子范围后，可以在该范围内为它分配任何值。

此代码有效：

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

但这不是：

```
var
  UppLetter: TUpperCase;
begin
  UppLetter := 'e'; //编译时错误
```

编写上面的代码会导致编译时错误，“Constant expression violates subrange bounds（常量表达式违反了子范围的边界）”如果您改为编写以下代码，则编译器将接受它：

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

在运行时，如果启用了“范围检查”编译器选项（在“项目选项”对话框的“编译器”页面中），则会收到到预期的 Range check error（范围检查错误）消息。这类似于我先前描述的整数类型溢出错误。

我建议您在开发程序时打开此编译器选项，这样它会更健壮且更容易调试，因为在发生错误的情况下，您将得到明确的消息，而不是不确定的行为。您最终可以在程序的最终版本中禁用此选项，以使其运行更快。但是，速度的提高几乎可以忽略不计，因此我建议即使在运输程序中也要打开所有这些运行时检查。

2.3.4 枚举类型

枚举类型（通常称为“枚举”）构成另一种用户定义的序数类型。在枚举中，

您没有列出现有类型的范围，而是列出了该类型的所有可能值。换句话说，枚举是（常量）值的列表。这里有些例子：

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

列表中的每个值都有一个关联的序数，从零开始。当您将 Ord 函数应用于枚举类型的值时，将获得此“从零开始”的值。例如，Ord（Diamond）返回 1。

枚举类型可以具有不同的内部表示形式。默认情况下，Delphi 使用 8 位表示形式，除非存在超过 256 个不同的值，在这种情况下，它将使用 16 位表示形式。还有一个 32 位表示形式，有时对于与 C 或 C++库的兼容性很有用。

您可以使用$ Z 编译器指令来更改枚举类型的默认表示，无论枚举中的元素数量如何，都要求使用更大的表示形式。这是相当罕见的设置。

Scoped Enumerators（范围枚举器）

枚举类型的特定常量值可以视为所有效果都视为全局常量，并且在不同枚举值之间存在名称冲突的情况。这就是为什么该语言支持作用域枚举的原因，可以使用特定的编译器指令$ SCOPEDENUMS 激活该功能，并且要求您使用类型名称作为前缀来引用枚举值：

```
// classic（经典）枚举值
S1 := Club;
// "scoped"（“范围”）枚举值
S1 := TSuit.Club;
```

引入此功能后，默认的编码样式仍然是传统行为，以避免破坏现有代码。实际上，作用域枚举器会更改枚举的行为，使得必须使用完全限定的类型前缀来引用它们。

使用绝对名称来引用枚举值可以消除发生冲突的风险，可以避免使用枚举值的初始前缀作为与其他枚举区别的方式，并使代码更易读，即使编写时间更长。

例如，System.IOUtils 单元定义了以下类型：

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

这意味着您不能将第二个值称为 soAllDirectories，但必须使用其完整名称来引用枚举值：

```
TSearchOption.soAllDirectories
```

FireMonkey 平台库也使用了很多作用域枚举器，要求类型作为实际值的前缀，而较旧的 VCL 库通常基于更传统的模型。RTL 是两者的结合。

❖ Object Pascal 库中的枚举值通常在值的开头使用两个或三个该类型的首字母，按照惯例使用小写字母，例如上例中的“搜索选项”为“so”。当使用类型作为前缀时，这似乎有点多余，但是鉴于该方法的通用性，我认为它不会很快消失。

2.3.5 集合类型

集合类型表示一组值，其中可用值列表由集合所基于的序数类型表示。这些序数类型通常是有限的，并且通常由枚举或子范围表示。

如果采用子范围 1..3，则基于它的集合的可能值仅包括 1，仅 2，仅 3、1 和 2、1 和 3、2 和 3，所有三个值，或没有一个。

变量通常包含其类型范围的可能值之一。相反，集合类型变量可以包含范围的任何一个，一个，两个，三个或更多值。它甚至可以包含所有值。

这是集合的示例：

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

现在，我可以定义这种类型的变量，并为其分配一些原始类型的值。为了表示集合中的某些值，请编写一个用逗号分隔的列表，并用方括号括起来。以下代码显示了对具有多个值，一个值和一个空值的变量的分配：

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

在 Object Pascal 中，通常使用集合来指示几个非排他的标志。例如，基于设置类型的值是字体样式。可能的值表示粗体，斜体，下划线和删除线字体。当然，相同的字体可以是斜体和粗体，没有属性或全部具有。因此，它被声明为集合。您可以在程序代码中为该集合分配值，如下所示：

```
Font.Style := []; // 没有风格
Font.Style := [fsBold]; // 仅大胆的风格
Font.Style := [fsBold, fsItalic]; // 两种风格活跃
```

集合运算符

我们已经看到，集合是一个非常特定于 Pascal 的用户定义数据类型。这就是为什么集合运算符值得特定的覆盖范围。它们包括并集（+），差（-），交集（*），隶属度检验（in），以及一些关系运算符。

要将元素添加到集合中，可以将该集合与仅包含所需元素的另一个集合进行合并。这是一个与字体样式有关的示例：

```
// 加粗体
Style := Style + [fsBold];
// 添加粗体和斜体，但删除下划线（如果存在）
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

或者，您可以使用标准的“包含”和“排除”过程，它们效率更高（但不能与 set 类型的组件属性一起使用）：

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

2.4 表达式和运算符

我们已经看到可以为变量分配类型兼容的文字值，常量值或另一个变量的值。在许多情况下，分配给变量的是表达式的结果，涉及一个或多个值和一个或多个运算符。表达式是语言的另一个核心元素。

2.4.1 使用运算符

构建表达式没有通用的规则，因为它们主要取决于所使用的运算符，并且 Object Pascal 具有许多运算符。有逻辑，算术，布尔，关系和集合运算符，以及

其他一些特殊的运算符：

```
// 样本表达
20 * 5 // 乘法
30 + n // 加法
a < b // 小于比较
-4 // 负值
c = 10 // 测试是否相等（如 C 语法中的==）
```

表达式是大多数编程语言所共有的，并且大多数运算符是相同的。表达式是常量，变量，文字值，运算符和函数结果的任何有效组合。表达式可用于确定要分配给变量的值，计算函数或过程的参数或测试条件。每次对标识符的值执行操作时，不是在单独使用标识符，而是在使用表达式。

- ❖ 表达式的结果通常存储在编译器代表您自动生成的具有适当数据类型的临时变量中。当您需要在同一代码片段中多次计算同一表达式时，可能需要使用显式变量。注意，复杂的表达式可能需要多个临时变量来存储中间结果，这也是编译器为您处理的，您通常可以忽略。

显示表达式的结果

如果您想对表达式做一些实验，没有什么比编写一个简单的程序更好的了。对于本书的大多数初始演示，请基于表单创建一个简单的程序，然后使用自定义的 Show 函数向用户显示内容。如果要显示的信息不是字符串消息而是数字或布尔逻辑值，则需要对其进行转换，例如调用 IntToStr 或 BoolToStr 函数。

- ❖ 在 Object Pascal 中，传递给函数或过程的参数用括号括起来。其他一些语言（尤其是 Rebol 和某种程度上为 Ruby）使您可以通过在函数或过程名称之后编写参数来传递参数。回到 Object Pascal，嵌套函数调用使用嵌套括号，如下面的代码所示。

这是 ExpressionsTest 应用程序项目的示例代码片段（为了清楚起见，在其中使用经典的 IntToStr 语法，假设参数是一个表达式：

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

该代码段的输出非常简单：

```
100
252
True
False
```

我已将此演示作为框架提供给您，您可以尝试不同类型的表达式和运算符，并查看相应的输出。

您在 Object Pascal 中编写的表达式将由编译器解析并生成汇编代码。如果要更改这些表达式之一，则需要更改源代码并重新编译应用程序。但是，系统库支持在运行时计算的动态表达式，该功能与反射相关，在第 16 章中进行了介绍。

2.4.2 运算符和优先级

表达式由应用于值的运算符组成。正如我提到的，大多数运算符在各种编程语言之间共享都相当直观，例如基本的匹配和比较运算符。在本节中，我将仅强调 Object Pascal 运算符的特定元素。

您可以在下面看到该语言的运算符列表，按优先级分组，并与 C＃，Java 和 Objective-C（以及大多数基于 C 语言语法的语言）进行比较。

关系和比较运算符（最低优先级）

=	测试是否相等（在 C 中为==）
<>	测试是否不相等（在 C 中为!=）
<	测试是否小于
>	测试是否大于
<=	测试是否小于或等于集合的子集
>=	测试是否大于或等于集合或集合的超集
in	测试项目是否是集合的成员
is	测试对象是否与给定类型兼容（在第 8 章中介绍）或实现给定接口（在第 11 章中介绍）

加法运算符

<table>
<tr><td>+</td><td>算术加法，集合并集，字符串串联，指针偏移加法</td></tr>
<tr><td>-</td><td>算术减，设定差，指针偏移减</td></tr>
<tr><td>or</td><td>布尔值或按位或（在 C 中为||或|）</td></tr>
<tr><td>xor</td><td>布尔值或按位互斥或（在 C 中按位互斥或为^）</td></tr>
</table>

乘法和按位运算符

*	算术乘法或集合相交
/	浮点除法
div	整数除法（在 C 中也使用/）
mod	模（整数除法的余数）（在 C 中为％）
as	允许在运行时进行类型检查的转换（在第 8 章中介绍了）
and	布尔值或按位运算符（在 C 中为&&或＆）
shl	按位左移（在 C 中为<<）
shr	按位右移（在 C 中为>>）

一元运算符（优先级最高）

@	变量或函数的内存地址（返回指针，在 C 中为＆）
not	布尔或按位非（在 C 中为！）

与许多其他编程语言不同，逻辑运算符（包括 and, or,和 not）的优先级高于比较运算符（包括<和>）。因此，如果您写：

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

编译器将首先执行 and 操作，通常会在表达式中导致类型兼容性编译器错误。 如果要测试两个比较，则应将每个小于号括在括号中：

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

相反，对于数学运算，将应用通用规则，其中乘法和除法优先于加法和减法。下面的前两个表达式是等效的，而第三个是不同的：

```
10 + 2 * 5 // result is 20
10 + (2 * 5) // result is 20
(10 + 2) * 5 // result is 60
```

- 尽管在某些情况下不需要括号，但鉴于您可以依靠语言运算符优先级规则，因此强烈建议您添加它们，因为这些规则根据编程语言而有所不同，对于任何阅读或修改该规则的人来说，最好保持清楚将来的代码。

当与不同的数据类型一起使用时，某些运算符具有不同的含义。

例如，+运算符可用于将两个数字相加，连接两个字符串，将两个集合进行并集，甚至为指针添加偏移量（如果特定的指针类型启用了指针数学运算）：

```
10 + 2 + 11
10.3 + 3.4
'Hello' + ' ' + 'world'
```

但是，不能像在 C 语言中那样添加两个字符。

div 是一个不寻常的运算符。在 Object Pascal 中，您可以使用/运算符将任意两个数字（实数或整数）相除，并且始终会得到实数结果。如果需要除以两个整数并想要一个整数结果，请改用 div 运算符。这是两个示例分配（随着我们在下一章介绍数据类型，此代码将变得更加清晰）：

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

为了确保整数除法没有余数，可以使用 mod 运算符并检查结果是否为零，如以下布尔表达式所示：

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

2.5 日期和时间

虽然在 Pascal 语言的早期版本中没有日期和时间的本机类型，但是 Object Pascal 具有日期和时间的本机类型。它使用浮点表示法来处理日期和时间信息。更具体地说，系统单元为此目的定义了特定的 TDateTime 数据类型。

这是浮点类型，因为它必须足够宽以在单个变量中存储年，月，日，小时，分钟和秒，低至毫秒的分辨率：

- 日期以 TDateTime 值的整数部分存储为自 1899 年 12 月 30 日以来的天数（负值表示 1899 年之前的日期）
- 时间以天的小数部分存储在值的小数部分中

✓ 如果您想知道奇怪的日期是从哪里来的，那么它背后还有一段很长的故事要与 Excel 和 Windows 应用程序中的日期表示联系在一起。当时的想法是将第 1 天视为 1900 年 1 月的第一天，因此 1899 年的除夕将是第 0 天。但是，该日期表示法的原始开发者适当地忘记了 1900 年不是飞跃。年，因此后来将计算结果调整了 1 天，从而将 1900 年 1 月 1 日转换为第 2 天。

如前所述，TDateTime 不是编译器可以理解的预定义类型，而是在系统单元中定义为：

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

❖ 考虑到系统单元总是自动包含在每个编译中，而没有使用语句（实际上将系统单元添加到使用部分中会导致编译错误），因此可以将系统单元几乎视为核心语言的一部分。但是，从技术上讲，该单元被视为运行时库（RTL）的核心部分，它将在第 17 章中介绍。

还有两种用于处理 TDateTime 结构的时间和日期部分的相关类型，分别定义为 TDate 和 TTime。这些特定类型是完整 TDateTime 的别名，但是系统函数会通过修剪数据的未使用部分来对待它们。

使用日期和时间数据类型非常容易，因为 Delphi 包含许多对此类型进行操作的函数。System.SysUtils 单元中有几个核心函数，而 System.DateUtils 单元中有许多特定函数（尽管名称也包括操纵时间的函数）。

在这里，您可以找到常用日期/时间操作功能的简短列表：

Now	将当前日期和时间返回为日期/时间值。
Date	仅返回当前日期。
Time	仅返回当前时间。
DateTimeToStr	使用默认格式将日期和时间值转换为字符串；若要对转换有更多控制，请改用 FormatDateTime 函数。
DateToStr	将日期/时间值的日期部分转换为字符串。
TimeToStr	将日期/时间值的时间部分转换为字符串。
FormatDateTime	使用指定的格式格式化日期和时间；您可以通过提供复杂的格式字符串来指定要查看的值以及要使用的格式。
StrToDateTime	将具有日期和时间信息的字符串转换为日期/时间值，如果字符串格式出错，则会引发异常。它的伴随函数 StrToDateTimeDef 在发生错误的情况下返回默认值，而不是引发异常。
DayOfWeek	返回与作为参数传递的日期/时间值的星期几相对应的数字（使用语言环境配置）。
DecodeDate	从日期值中检索年，月和日值。
DecodeTime	从日期值中检索小时，分钟，秒和毫秒。
EncodeDate	将年，月和日值转换为日期/时间值。
EncodeTime	将小时，分钟，秒和毫秒值转换为日期/时间值。

为了向您展示如何使用此数据类型及其一些相关例程，我建立了一个名为 TimeNow 的简单应用程序项目。程序启动时，它将自动计算并显示当前时间和日期。

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

第一条语句是对 Now 函数的调用，该函数返回当前日期和时间。此值存储在 StartTime 变量中。

❖ 在不带参数的情况下调用 Object Pascal 函数时，与 C 样式语言不同，无需键入空括号。

接下来的两个语句显示 TDateTime 值的时间部分（转换为字符串）和相同值的日期部分。这是程序的输出（将取决于您的语言环境配置）：

```
Time is 6:33:14 PM
Date is 10/7/2020
```

要编译该程序，您需要引用 System.SysUtils 单元（“system utilities（系统实用程序）”的简称）中的函数。除了调用 TimeToStr 和 DateToStr，您还可以使用功能更强大的 FormatDateTime 函数。

请注意，时间和日期值会根据系统的国际设置转换为字符串。根据操作系统和语言环境从系统中读取日期和时间格式设置信息，并填充 TFormatSettings 数据结构。如果需要自定义格式，则可以创建该类型的自定义结构，并将其作为参数传递给大多数日期时间格式函数。

❖ TimeNow 项目还有另一个按钮，可用于启用计时器。这是一个随时间自动执行事件处理程序的组件（您指定间隔）。在演示中，如果启用计时器，您将看到当前时间每秒钟添加到列表中。一个更有用的用户界面将是每秒用当前时间更新标签，基本上建立一个时钟。

2.6 类型转换

如我们所见，您不能将一种数据类型的变量分配给另一种数据类型。原因是，根据数据的实际表示，您可能最终得到的是毫无意义的东西。

现在，对于每种数据类型而言并非如此。例如，始终可以安全地推广数字类型。这里的“提升”表示您始终可以安全地将值分配给具有较大表示形式的类型。因此，您可以将单词分配给整数，将整数分配给 Int64 值。编译器允许使用相反的操作，称为“降级”，但会发出警告，因为您可能会得到部分数据。其他自动转换只是一种方式：例如，您可以将整数分配给浮点数，但是相反的操作是非法的。

在某些情况下，您想更改值的类型，并且该操作很有意义。当您需要执行此操作时，有两种选择。一种是执行直接类型转换，这将复制物理数据，并可能导致正确的转换或不取决于类型。当您执行类型转换时，您是在告诉编译器“我知道我在做什么，让我努力吧”。如果您使用类型强制转换但不确定自己在做什么，则可能会因为失去编译器类型检查安全网而遇到麻烦。

类型转换使用简单的功能符号，并将目标数据类型的名称用作函数：

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

您可以安全地在大小相同的数据类型之间进行类型转换（即表示数据的字节数相同-与上面的代码段不同！）。在顺序类型之间进行类型转换通常是安全的，但是只要知道自己在做什么，就可以在指针类型（以及对象）之间进行类型转换。

直接类型转换是一种危险的编程习惯，因为它使您可以像访问其他值一样访问值。由于数据类型的内部表示形式通常不匹配（甚至可能因目标平台而异），因此您可能会意外地产生难以跟踪的错误。因此，通常应避免键入强制转换。

将变量分配给其他类型之一的第二种选择是使用类型转换函数。下面总结了允许您在各种基本类型之间进行转换的函数列表（并且我已经在本章的演示中使用了其中一些函数）：

Chr	将序数转换为字符。
Ord	将序数类型的值转换为表示其顺序的数字。
Round	将实型值转换为整数型值，并将其舍入（也请参见以下注释）。
Trunc	将实型值转换为整数型值，将其值截断。
Int	返回浮点值参数的 Integer 部分。
FloatToDecimal	将浮点值转换为记录，包括其十进制表示形式（指数，数字，符号）。
FloatToStr	使用默认格式将浮点值转换为其字符串表示形式。
StrToFloat	将字符串转换为浮点值。

❖ Round 函数的实现基于 CPU 提供的本机实现。现代处理器通常采用所谓的“Banker's Rounding”，它根据中间值（奇数或偶数）上下舍入中间值（例如 5.5 或 6.5）。 还有其他舍入功能，例如 RoundTo，可为您提供对实际操

作的更多控制。

如本章前面所述，其中一些转换函数还可以作为对数据类型的直接操作（由于类型帮助器机制）。尽管有类似 IntToStr 的经典转换，但是您可以将 ToString 操作应用于大多数数字类型，以将它们转换为字符串表示形式。您可以使用类型帮助程序直接将许多转换直接应用于变量，这应该是您首选的编码样式。

其中一些例程适用于我们将在以下各节中讨论的数据类型。请注意，该表不包含特殊类型的例程（例如 TDateTime 或 Variant）或专门用于格式化而不是转换的例程，例如功能强大的 Format 和 FormatFloat 例程。

第三章 语言语句

如果数据类型的概念在 Pascal 编程语言首次发明时是其突破之一，则另一侧由代码或编程语句表示。那时，尼克劳斯 • 沃思（Nicklaus Wirth）的杰出著作“算法+数据结构=程序”（1976 年 2 月，普伦蒂斯 • 霍尔（Prentice Hall）出版）澄清了这一想法（经典著作，目前仍在转载和提供）。尽管本书比面向对象编程要早很多年，但它基于强大的数据类型概念，可以被认为是现代编程的基础之一，因此，它是导致面向对象编程语言的概念的基础。

编程语言的语句基于关键字和其他元素，使您可以向编译器指示要执行的一系列操作。语句通常包含在过程或函数中，我们将在下一章开始详细介绍。现在，我们仅关注可用来创建程序的基本指令类型。正如我们在第 1 章（在有关空白和代码格式的部分）所看到的那样，可以非常自由地编写实际的程序代码。我还介绍了注释和其他一些特殊元素，但从未完全介绍一些核心概念，例如编程语句。

3.1 简单和复合语句

编程指令通常称为语句。一个程序块可以由多个语句组成。语句有两种类型，简单和复合。

一条不包含任何其他子语句的语句称为简单语句。

简单语句的示例是赋值语句和过程调用。在 Object Pascal 中，简单的语句用分号分隔：

```
X := Y + Z; //  分配
Randomize; //  程序调用
...
```

要定义复合语句，可以在关键字 begin 和 end 中包含多个语句之一，这些关键字充当多个语句的容器，并且具有与 C 衍生语言中的花括号类似但不相同的作用。复合语句可以出现在简单的对象 Pascal 语句可以出现的任何地方。这是一个例子：

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

不需要在复合语句的最后一条语句之后（即末尾之前）的分号，如下所示：

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

两种版本都是正确的。第一个版本的最终分号没有用（但无害）。实际上，该分号是一个空语句或一个空语句。也就是说，没有代码的语句。这与许多其他编程语言（例如基于 C 语法的编程语言）有显着不同，在其他编程语言中，分号是语句终止符（而不是分隔符），并且始终在语句末尾使用。

请注意，有时，空语句可以在循环内或在其他特定情况下专门用于代替实际语句，例如：

```
while condition_with_side_effect do
  ; //  空或空语句
```

尽管这些最后的分号毫无用处，但大多数开发人员倾向于使用它们，我建议您也这样做。有时，在写了几行之后，您可能想要再添加一条语句。如果缺少最

后一个分号，则必须记住添加它，因此通常最好将其添加到第一位。我们马上就会看到，添加额外的分号的规则是个例外，那就是当下一个元素是条件中的 else 语句时。

3.2 if 语句

根据特定的测试（或条件），条件语句用于执行它包含的任何一个语句，也可以不执行任何一个。条件语句有两种基本类型：if 语句和 case 语句。

如果满足特定条件（if-then），则 if 语句可用于执行语句；或者在两个不同的选择之间进行选择（if-then-else）。条件用布尔表达式定义。

一个简单的 Object Pascal 示例，称为 IfTest，将演示如何编写条件语句。 在此程序中，我们将使用一个复选框，通过读取其 IsChecked 属性（并将其存储到一个临时变量中，尽管不是严格要求的，因为您可以直接在条件表达式中检查该属性值）来获取用户输入：

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

如果选中此复选框，程序将显示一条简单消息。否则什么也不会发生。相比之下，使用 C 语言语法的同一条语句如下所示（其中条件表达式必须用括号括起来）：

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

其他一些语言使用 endif 元素的概念来允许您编写多个语句，在 Object Pascal 语法中，条件语句默认为单个语句。作为相同条件的一部分，您可以使用 begin-end 块执行多个语句。

如果要根据条件执行不同的操作，可以使用 if-then-else 语句（在这种情况下，我使用直接表达式读取复选框状态）：

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

请注意，在第一条语句之后和 else 关键字之前，不能有分号，否则编译器将发出语法错误。 原因是 if-then-else 语句是单个语句，因此您不能在其中间放置分号。

if 语句可能非常复杂。 可以将条件转换为一系列条件（使用 and，or 或 not 布尔运算符），或者 if 语句可以嵌套第二条 if 语句。除了嵌套 if 语句外，当存在多个不同的条件时，通常有连续的 if-then-else-if-then 语句。您可以根据需要保持链接尽可能多的其他条件。

IfTest 应用程序项目的第三个按钮使用输入框的第一个字符（可能会丢失，因此可能是外部测试）演示了这些情况：

```
var
  AChar: Char;
begin
  // 多个嵌套的 if 语句
```

```
if Edit1.Text.Length > 0 then
begin
  AChar := Edit1.Text.Chars[0];
  // 检查小写字符（两个条件）
  if (AChar >= 'a') and (AChar <= 'z') then
    Show ('char is lowercase');
  // 后续条件
  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;
```

仔细查看代码，然后运行程序以了解是否理解（并尝试编写类似的程序以了解更多信息）。您可以考虑更多选项和布尔表达式，并增加此小示例的复杂性，进行您喜欢的任何测试。

3.3 case 语句

如果您的 if 语句变得非常复杂，并且它们基于序数值的测试，则可以考虑将它们替换为 case 语句。case 语句由用于选择值的表达式和可能值或值范围的列表组成。 这些值是常量，并且必须是唯一的并且是序数类型。最终，如果您指定的值都不与选择器的值相对应，则可以执行 else 语句。尽管没有特定的 endcase 语句，但 case 总是以结尾终止（在这种情况下，它不是块终止符，因为没有匹配的开始）。

❖ 创建一个 case 语句需要一个序数值。当前不允许基于字符串值的 case 语句。在这种情况下，您需要使用嵌套的 if 语句或不同的数据结构，例如字典（如我在第 14 章的书中稍后所述）。

这是一个示例（CaseTest 项目的一部分），该示例使用在 NumberBox 控件（数字输入控件）中输入的数字的整数部分作为输入：

```
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);
```

另一个示例是对先前复杂的 if 语句的扩展，它变成了案例测试的许多不同条件：

```
case AChar of
  '+' : AText := '加法';
  '-' : AText := '减法';
  '*', '/': AText := '乘法或除法';
  '0'..'9': AText := '数字';
  'a'..'z': AText := '小写字母';
  'A'..'Z': AText := '大写字母';
```

```
    #12032..#12255: AText := '康熙字根';
  else
    AText := '其他字符: ' + aChar;
  end;
```

❖ 如您在前面的代码片段中所看到的，使用子范围数据类型的相同语法定义了一个值范围。相反，单个分支的多个值用逗号分隔。对于 Kangxi Radical 部分，我使用的是数字值而不是实际的字符，因为 IDE 编辑器使用的大多数固定大小的字体都无法正确显示符号。

包括 else 部分以表示未定义或意外的情况被认为是一种好习惯。Object Pascal 中的 case 语句选择一个执行路径，它不会将自己定位在入口点。换句话说，它将在所选值的冒号之后执行该语句或块，并且将跳至大小写之后的下一条语句。

这与 C 语言（及其某些派生语言）非常不同，后者将 switch 语句的分支视为入口点，并且将执行以下所有语句，除非您专门使用 break 请求（尽管这是 Java 和 Java 的特定情况）C#实际上在实现上有所不同）。C 语言语法如下所示：

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

3.4 for 循环语句

Object Pascal 语言具有大多数编程语言的典型重复或循环语句，包括 for，while 和 repeat 语句，以及更现代的 for-in（或 for-each）循环。如果您使用了其他编程语言，那么大多数这些循环将是熟悉的，因此，我仅简要介绍它们（表明与其他语言的主要区别）。

Object Pascal 中的 for 循环严格基于一个计数器，每次执行循环时，该计数器可以增加或减少。这是一个用于添加前十个数字的 For 循环的简单示例（ForTest 演示的一部分）。

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

对于那些好奇的人，输出为 55。在内联变量引入之后，您可以编写 for 循环的另一种方法是在声明中声明循环计数器变量（其语法在某种程度上类似于 C 语言中的循环以及稍后讨论的衍生语言）：

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

在这种情况下，您还可以利用类型推断的优势并忽略类型规范。上面的完整代码段变为：

```
var
  Total: Integer;
begin
  Total := 0;
  for var I := 1 to 10 do
```

```
      Total := Total + I;
    Show(Total.ToString);
```

使用内联循环计数器的一个优点是该变量的范围将限于循环：在 for 语句之后使用它会导致错误，而通常在循环外使用循环计数器时只会收到警告。

Pascal 中的 for 循环不如其他语言灵活（不可能指定一个不同的增量），但它简单易懂。作为比较，这是用 C 语言语法编写的 for 循环相同：

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

在这些语言中，增量是可以指定任何类型序列的表达式，这可以导致许多人认为不可读的代码，如下所示：

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

相反，在 Object Pascal 中，您只能使用单步增量。如果要测试更复杂的条件，或者要提供自定义计数器，则需要使用 while 或 repeat 语句，而不是 for 循环。

单增量的唯一替代方法是单减，或者使用 downto 关键字反向循环：

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

❖ 反向计数很有用，例如，当您影响要遍历的基于列表的数据结构时。删除某些元素时，您通常会向后移动，就像使用正向循环一样，您可能会影响正在操作的顺序（即，如果删除列表的第三个元素，则第四个元素将变为第三个元素：现在您位于第三，移至下一个（第四个），但实际上您正在操作第五个元素，跳过一个）。

在 Object Pascal 中，for 循环的计数器不必为数字。它可以是任何序数类型的值，例如字符或枚举类型。这可以帮助您编写更具可读性的代码。这是一个基于 Char 类型的 for 循环示例：

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

此代码（ForTest 程序的一部分）显示了英文字母的所有字母，每个字母都位于输出 Memo 控件的单独一行中。

❖ 作为第 2 章 CharsTest 示例的一部分，我已经展示了一个类似的演示，但是基于整数计数器，但是，在这种情况下，char 被串联在单个输出字符串中。

这是另一个代码片段，显示基于自定义枚举的 for 循环：

```
type
  TSuit = (Club, Diamond, Heart, Spade);
var
  ASuit: TSuit;
begin
  for ASuit := Club to Spade do
    ...
```

最后一个循环遍历数据类型的所有元素。最好写成对类型的每个元素进行显

式操作（使其更灵活地更改定义），而不是通过写以下内容来具体指出第一个和最后一个元素：

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

以类似的方式，在数据结构的所有元素（例如字符串）上编写 for 循环是很常见的。在这种情况下，您可以使用以下代码（来自 ForTest 项目）：

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

如果您不想指示数据结构的第一个元素和最后一个元素，则可以使用 for-in 循环，这是下一部分中讨论的一种特殊用途的 for 循环。

❖ 编译器如何使用[]运算符处理直接读取字符串数据并确定字符串的上下边界，这在 Object Pascal 中仍然是一个相当复杂的主题，即使现在所有平台的默认值都相同。这将在第 6 章中介绍。

对于使用基于 zer0 的索引的数据结构，您希望从索引 0 循环到数据结构大小或长度之前的索引。编写此代码的常见方法是：

```
for I := 0 to Count-1 do ...
for I := 0 to Pred(Count) do...
```

关于 for 循环的最后说明是循环后循环计数器发生的情况。简而言之，该值未指定，如果在循环终止后尝试使用 for 循环计数器，则编译器将发出警告。使用内联变量作为循环计数器的一个优点是，该变量仅在循环本身内定义，并且在 end 语句之后将无法访问，从而导致编译器错误（这是一种更强大的保护）：

```
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：未声明的标识符“I”
```

3.4.1 for-in 循环

Object Pascal 具有特定的循环构造，用于循环遍历列表或集合的所有元素，这称为 for-in（其他编程语言均要求此功能）。在此 for 循环中，循环对数组，列表，字符串或其他某种类型的容器的每个元素进行操作。与 C＃不同，Object Pascal 不需要实现 IEnumerator 接口，但是内部实现有些相似。

❖ 您可以在第 10 章中找到有关如何在类中支持 for-in 循环，添加自定义枚举支持的技术详细信息。

让我们从一个非常简单的容器（一个字符串）开始，它可以看作是字符的集合。在上一节的结尾，我们已经看到了如何使用 for 循环对字符串的所有元素进行操作。通过以下基于字符串的 for-in 循环，可以获得相同的精确效果，其中 Ch 变量依次接收每个字符串元素作为值：

```
var
  S: string;
  Ch: Char;
begin
  S := 'Hello world';
```

```
for Ch in S do
  Show(Ch);
```

此代码段也是 ForTest 应用程序项目的一部分。与使用传统的 for 循环相比，优点是您无需记住字符串的第一个元素以及如何提取最后一个元素的位置。该循环更易于编写和维护，并具有类似的效率。

像传统的 for 循环一样，for-in 循环也可以受益于使用内联变量声明。我们可以使用以下等效代码重写上面的代码：

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

for-in 循环可用于访问几种不同数据结构的元素：

- 字符串中的字符（请参见前面的代码片段）
- 集合中的活动值
- 静态或动态数组中的项目，包括二维数组（在第 5 章中介绍）
- 具有 GetEnumerator 支持的类引用的对象，包括许多预定义的对象，例如字符串列表中的字符串，各种容器类的元素，表单所拥有的组件以及许多其他对象。如何实现这一点将在第 10 章中讨论。

现在，在本书中要覆盖这些高级用法模式有些困难，因此我将在本书后面的示例中再次介绍该循环的示例。

❖ 某些语言（例如 JavaScript）中的 for-in 循环由于运行速度很慢而声誉不佳。在对象 Pascal 中不是这种情况，在此过程中，大约需要花费与相应循环标准相同的时间。为了证明这一点，我在 LoopsTest 应用程序项目中添加了一些计时代码，该计时代码首先创建一个包含 3000 万个元素的字符串，然后使用两种类型的循环对其进行扫描（每次迭代都做一个非常简单的操作。速度的差异是约 10％的人赞成经典的 for 循环（62 毫秒，而 Windows 机器上为 68 毫秒）。

3.5 While 和 Repeat 语句

while-do 和 repeat-until 循环背后的想法是一遍又一遍地重复执行代码块，直到满足给定条件为止。这两个循环之间的区别是在循环的开始或结束时检查条件。换句话说，repeat 语句的代码块始终至少执行一次。

❖ 大多数其他编程语言只有一种类型的开放式循环语句，通常被称为行为类似于 while 循环。C 语言语法具有与 Pascal 语法相同的两个选项，同时具有 while 和 do-while 循环。注意，请注意，它们使用相同的逻辑条件，不同于具有相反条件的 repeat-until 循环。

通过看一个简单的代码示例，您可以轻松理解为什么重复循环总是至少执行一次：

```
while (I <= 100) and (J <= 100) do
begin
  // 用 I 和 J 计算一些东西...
  I := I + 1;
  J := J + 1;
end;
```

```
repeat
  // 用 I 和 J 计算一些东西...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);
```

- 您将注意到，在 while 和重复条件中，我都将“子条件”括在括号中。在这种情况下，有必要这样做，因为编译器将在执行比较之前或在执行比较之前（如我在第 2 章的运算符部分中所述）。

如果 I 或 J 的初始值大于 100，则 while 循环将被完全跳过，而 repeat 循环内的 while 语句将始终执行一次。

这两个循环之间的另一个主要区别是重复直到循环的条件相反。只要不满足条件，就会执行此循环。

当满足条件时，循环终止。这与 while-do 循环相反，while-do 循环在条件为 true 时执行。因此，我必须反转上面代码中的条件才能获得类似的效果。

- “逆向条件”被正式称为“De Morgan's 法则”（例如，在 Wikipedia 上进行了描述，网址为 http://en.wikipedia.org/wiki/De_Morgan%27s_laws）。

3.5.1 循环示例

为了探索更多的循环细节，让我们看一个小的实际例子。LoopsTest 程序突出显示了具有固定计数器的循环和具有开放计数器的循环之间的区别。第一个固定计数器循环，一个 for 循环，按顺序显示数字：

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

使用 while 循环也可以得到相同的结果，其内部增量为 1（请注意，在使用当前循环后，您要递增该值）。但是，使用 while 循环时，您可以自由设置自定义增量，例如按 2：

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

此代码显示从 1 到 19 的所有奇数。这些具有固定增量的循环在逻辑上是等效的，并且执行预定义的次数。这并非总是如此。有些循环的执行不确定，例如取决于外部条件。

在编写 while 循环时，必须始终考虑永远不满足条件的情况。

- 例如，如果您在上面编写了循环，但是忘记增加循环计数器，则会导致无限循环（该循环将永远使程序停止运行，使 CPU 消耗 100%，直到操作系统将其杀死为止）。

为了显示确定性较低的循环的示例，我仍然基于计数器编写了 while 循环，

但是该循环是随机增加的。为此，我使用范围值为 100 的 Random 函数进行了调用。此函数的结果是一个 0 到 99 之间的数字，是随机选择的。一系列随机数控制执行 while 循环的次数：

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

如果您记得添加一个调用 Randomize 过程，该过程会在每次执行程序的不同时间重置随机数生成器，每次运行该程序时，数字都会不同。以下是两个单独执行的输出，并排显示：

```
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，不仅每次生成的数字都不同，而且项数也不同。此 while 循环执行随机次数。如果您连续执行几次程序，您将看到输出具有不同的行数。

3.5.2 Break 和 Continue

尽管有所不同，但每个循环仍允许您根据某些规则多次执行一个语句块。但是，在某些情况下，您可能需要添加一些其他行为。举例来说，假设您有一个 for 循环，在其中搜索给定字母的出现（此代码是 FlowTest 应用程序项目的一部分）：

```
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;
```

最后，您可以检查 found 的值，以查看给定字母是否为字符串的一部分。问题在于，即使发现了某个字符，该程序仍会继续重复循环并检查给定字符（这可能是一个很长的字符串的问题）。

一个经典的替代方法是将其转换为 while 循环并检查两个条件（循环计数器和 Found 的值）：

```
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;
```

尽管此代码是逻辑性和可读性的，但仍有更多代码要编写，并且如果条件变得多重且更加复杂，则将所有各种选项组合在一起将使代码非常复杂。

这就是该语言（或更确切地说，其运行时支持）具有系统过程的系统过程，可让您更改循环执行的标准流程：

- Break 过程中断循环，直接跳到其后的第一条语句，跳过任何进一步的执行
- Continue 过程跳至循环测试或计数器增量，继续循环的下一次迭代（除非条件不再成立） 或计数器已达到最大值）

使用 Break 操作，我们可以修改原始循环以匹配字符，如下所示：

```
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; //跳出 for 循环
    end;
```

另外两个系统过程，退出和停止，使您可以立即从当前函数或过程中返回或终止程序。我将在下一章介绍退出，而基本上没有理由调用 Halt（因此我不会在本书中进行讨论）。

Goto 来了吗？ 没门！

实际上，打破流程比上述四个系统过程要多得多。

最初的 Pascal 语言是臭名昭著的 goto 语句的功能之一，它使您可以在源代码的任何行上附加标签，然后从另一个位置跳转到该行。与条件语句和循环语句不同，后者揭示了为什么要从顺序代码流中分离出来，而 goto 语句通常看起来像是不稳定的跳转，并且实际上是完全不鼓励的。我是否提到 Object Pascal 不支持它们？不，我没有，也不会向您展示代码示例。对我而言，goto 早已不复存在。

❖ 到目前为止，我还没有涉及其他语言陈述，但它们是语言定义的一部分。其中之一是 with 语句，它与记录特别相关，因此我将在第 5 章中介绍它。With 是另一个“debated（辩论）”语言功能，但不如 goto 讨厌。

第四章 例程和函数

Object Pascal 语言（以及 C 语言的类似功能）中强调的另一个重要思想是例程的概念，该例程基本上是一系列具有唯一名称的语句，可以多次激活。例程（或函数）以其名称调用。这样，您不必一遍又一遍地编写相同的代码，并且在程序的许多地方都将使用单一版本的代码。从这个角度来看，您可以将例程视为基本的代码封装机制。

4.1 例程和函数

在 Object Pascal 中，例程可以采用两种形式：过程和函数。从理论上讲，例程是您要求计算机执行的操作，函数是返回值的计算。函数具有结果，返回值或类型，而过程则没有结果，这一事实凸显了这种差异。C 语言语法提供了一种单一的机制，函数，并且在 C 中，过程是具有 void（或 null）结果的函数。

两种例程都可以具有指定数据类型的多个参数。正如我们稍后将看到的，过程和函数也是类方法的基础，在这种情况下，两种形式之间的区别仍然存在。实际上，与 C，C++，Java，C＃或 JavaScript 不同，在声明函数或方法时，您需要使用这两个关键字之一。

实际上，即使有两个单独的关键字，函数和过程之间的差异也非常有限：您可以调用一个函数来执行一些工作，然后忽略结果（可能是可选的错误代码或类似的代码）或者，您可以调用一个将结果传回参数之一的过程（在本章后面的参考参数中有更多介绍）。

这是使用 Object Pascal 语言语法的过程定义，该语法使用特定的过程关键字，并且是 FunctionTest 项目的一部分：

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

作为比较，这将是使用 C 语言语法编写的相同函数，该函数没有关键字，即使没有参数，也需要带括号，并且返回值是空值或空值以指示没有结果：

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

实际上，在 C 语言语法中，过程和函数之间没有区别。相反，在 Pascal 语言语法中，函数具有特定的关键字，并且必须具有返回值（或返回类型）。

❖ Object Pascal 与其他语言之间还有另一个非常特殊的语法差异，即在 begin 关键字之前的定义中，函数或过程签名的末尾有分号。

有两种方法可以指示函数调用的结果，将值分配给函数名称或使用 Result 关键字：

```
// classic（经典的）Pascal 风格
function DoubleOld (Value: Integer) : Integer;
begin
  DoubleOld := Value * 2;
end;

//  现代替代
function Double (Value: Integer) : Integer;
```

```
begin
  Result := Value * 2;
end;
```

与经典的 Pascal 语言语法不同，现代的 Object Pascal 实际上具有三种指示函数结果的方式，包括本章“带有结果的退出”一节中讨论的 Exit 机制。

使用 Result 而不是函数名称来分配函数的返回值是最常见的语法，并且倾向于使代码更具可读性。函数名称的使用是经典的 Pascal 表示法，现在很少使用，但仍受支持。

同样，通过比较，可以使用 C 语言语法编写相同的函数，如下所示：

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

❖ 基于 C 语法的语言的 return 语句指示函数的结果，但也终止执行，将控制权返回给被调用者。相反，在 Object Pascal 中，向函数结果分配值不会终止它。这就是为什么结果经常用作常规变量的原因，例如分配初始默认值，甚至在算法中修改其结果。同时，如果需要停止执行，则还需要使用其他一些流控制语句的 Exit。以下部分“以结果退出”中将更详细地介绍所有这些。

如果这是可以定义这些例程的方式，则调用语法相对简单，因为您键入标识符，然后在括号内键入参数。

在没有参数的情况下，可以省略空括号（与基于 C 语法的语言不同）。此代码段及以下几个代码段是本章的 FunctionsTest 项目的一部分：

```
// 调用例程
Hello;
// 调用函数
X := Double (100);
Y := Double (X);
Show (Y.ToString);
```

这是我介绍的代码概念的封装。当您调用 Double 函数时，您无需了解实现它的算法。如果您以后找到了一种更好的将数字加倍的方法，则可以轻松更改函数的代码，但是调用代码将保持不变（尽管执行起来可能会更快）。

可以将相同的原理应用于 Hello 程序：我们可以通过更改该程序的代码来修改程序输出，并且主程序代码将自动更改其效果。这是我们可以更改过程实现代码的方法：

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

4.1.1 Forward 声明

当您需要使用（任何类型的）标识符时，编译器必须已经看过它，才能知道标识符所指的是什么。因此，通常在使用任何例程之前都提供完整的定义。但是，在某些情况下这是不可能的。如果过程 A 调用过程 B，而过程 B 调用过程 A，则当您开始编写代码时，您将需要调用一个编译器仍未定义的例程。

在这种情况下（以及许多其他情况），您可以使用特定名称和给定参数声明过程或函数的存在，而无需提供其实际代码。声明过程或函数而不进行定义的一种方法是编写其名称和参数（称为函数签名），后跟 forward 关键字：

```
procedure NewHello; forward;
```

稍后，代码应提供该过程的完整定义（必须在同一单元中），但是现在可以在完全定义该过程之前调用该过程。

这里有一个例子，只是为了给你这个主意：

```
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;
```

❖ 上一个代码片段中调用的 MessageDlg 函数是在 FireMonkey 框架中要求用户确认的相对简单的方法（VCL 框架中也存在类似的函数）。参数是消息，对话框的类型和要显示的按钮。结果是所选按钮的标识符。

这种方法（也是 FunctionTest 应用程序项目的一部分）使您可以编写相互递归：DoubleHello 调用 Hello，但 Hello 也可以调用 DoubleHello。换句话说，如果您继续选择“是”按钮，程序将继续显示该消息，并为每个“是”显示两次。在递归代码中，必须有一个终止递归的条件，以避免被称为堆栈溢出的条件。

❖ 函数调用将应用程序存储器的堆栈部分用于参数，返回值，局部变量等。如果某个函数一直在无休止的循环中调用自身，则堆栈的存储区（通常具有固定的和预定义的大小，该大小由 Linker（链接器）确定，并在项目选项中配置）将通过称为堆栈溢出的错误终止。不用说，流行的开发人员支持网站（www.stackoverflow.com）的名称来自此编程错误。

尽管在 Object Pascal 中，前向过程声明不是很常见，但是有一种类似的情况更常见。在单元的接口部分中声明过程或函数时，即使不存在 forward 关键字，也会自动将其视为前向声明。实际上，您不能在单元的接口部分中编写例程的主体。 注意，您必须提供在同一单元中声明的每个例程的实际实现。

4.1.2 递归函数

鉴于我提到了递归并给出了一个相当特殊的示例（两个过程互相调用），让我也向您展示一个递归函数调用自身的经典示例。使用递归通常是编码循环的另一种方法。

为了坚持经典的演示，假设您要计算数字的幂，并且缺少适当的功能（当然，在运行时库中可用）。您可能从数学上还记得，2 乘以 3 的幂对应于将 2 乘以 3 倍，即 2*2*2。

用代码表示这种情况的一种方法是编写一个 for 循环，该循环执行 3 次（或指数的值），然后将 2（或底数的值）乘以当前总数，从 1 开始：

```
function PowerL (Base, Exp: Integer): Integer;
```

```
var
   I: Integer;
begin
   Result := 1;
   for I := 1 to Exp do
      Result := Result * Base;
end;
```

另一种方法是用相同的幂次幂乘以底数，然后将其乘以递减的指数，直到指数为 0，在这种情况下，结果始终是 1。以递归的方式：

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

该程序的递归版本可能不会比基于 for 循环的版本快，也不会更易读。但是，在诸如解析代码结构（例如树形结构）之类的场景中，没有固定数量的元素要处理，因此编写循环几乎是不可能的，而递归函数使自己适应该角色。

通常，尽管递归代码功能强大，但往往更加复杂。与编程的早期相比，在经过了近乎忘了递归的多年之后，诸如 Haskell，Erlang 和 Elixir 之类的新功能语言大量使用了递归，并将这种想法重新流行起来。

无论如何，您都可以在 FunctionTest 应用程序项目的代码中找到这两个幂函数。

❖ 该演示的两个幂函数不能处理负指数的情况。在这种情况下，递归版本将永远循环（直到程序遇到物理约束为止）。同样，通过使用整数，达到最大数据类型大小并溢出它相对较快。我写这些函数时有这些固有的局限性，试图使它们的代码保持简单。

4.1.3 什么是方法？

我们已经看到了如何通过使用 forward 关键字在单元的接口部分中编写前向声明。类类型内部的方法的声明也被视为前向声明。

但是到底是什么方法？方法是一种特殊的功能或过程，与两种数据类型（记录或类）之一有关。 在 Object Pascal 中，每次处理视觉组件的事件时，我们都需要定义一个方法，通常是一个过程，但是术语“方法”用于表示绑定到类或记录的函数和过程。

这是一个自动添加到表单源代码中的空方法（它确实是一个类，我们将在本书后面详细介绍）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   // 这是您的代码
end;
```

4.2 参数和返回值

调用函数或过程时，需要传递正确数量的参数，并确保它们与期望的类型匹

配。否则，编译器将发出错误消息，类似于为变量分配错误类型的值时的类型不匹配。给定 Double 函数的先前定义，如果调用，则采用 Integer 参数：

```
Double (10.0);
```

编译器将显示错误：

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

（[dcc32 错误] E2010 不兼容的类型："整数"和"扩展"）

- ✧ 当您键入函数名称或过程的名称和打开的括号后，编辑器会通过带提示的提示来建议函数或过程的参数列表，从而为您提供帮助。此功能称为"代码参数"，是 Code Insight 技术（在其他 IDE 中称为 IntelliSense）的一部分。从 Delphi 10.4 开始的 CodeInsight 由 LSP 服务器（语言服务器协议）提供支持。

在某些情况下，允许进行有限的类型转换，这与赋值类似，但通常应尝试使用特定类型的参数（这对于参考参数是强制性的，我们将在一段时间内看到）。

调用函数时，可以将表达式作为参数而不是值。计算表达式并将其结果分配给参数。 在更简单的情况下，您只需传递变量的名称。在这种情况下，变量的值将复制到参数（通常具有不同的名称）。我强烈建议您不要对参数和作为该参数值传递的变量使用相同的名称，因为这可能会造成混乱。

- ➢ 使用 Delphi，通常不应该依赖传递给函数的参数的求值顺序，因为这取决于调用约定，并且在相同情况下它是不确定的，尽管最常见的情况是从右到左求值。有关更多信息，请访问：

http://docwiki.embarcadero.com/RADStudio/en/Procedures_and_Functions_(Delphi)#Calling_Conventions

最后，请注意，您可以拥有一个具有不同版本的函数或过程（一种称为重载的功能）和一些参数，您可以跳过这些函数或过程以使它们使用预定义的值（一种称为默认参数的功能）。本章稍后的特定部分将详细介绍功能和过程的这两个关键功能。

4.2.1 Exit with a Result（结果退出）

我们已经看到，与 C 语言家族相比，从函数返回结果使用的语法完全不同。不仅语法不同，而且行为也不同。向 Result（或函数名称）分配值不会像 return 语句那样终止函数。Object Pascal 开发人员通常通过将 Result 用作临时存储来利用此功能。而不是写：

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

您可以省略临时变量，而使用 Result。函数终止时 Result 的值是该函数返回的值：

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

```
end;
```

另一方面，在某些情况下，您想分配一个值并立即退出过程，例如在特定的 if 分支中。如果需要分配函数结果并停止当前执行，则必须使用两个单独的语句，分配 Result，然后使用 Exit 关键字。

如果您记得上一章 FlowTest 应用程序项目的代码(在"使用 BreakandContinue 中断流程"一节中介绍），则可以将其重写为函数，将 Break 的调用替换为 Exit 的调用。我在 ParamsTest 应用程序项目的以下代码段中进行了此更改：

```
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 中，您可以使用对 Exit 的特殊调用来替换 if 块的两个语句，该调用将类似于函数 C 的 return 语句的函数返回值传递给该函数。因此，您可以以更紧凑的方式编写上述代码（也因为使用一条语句可以避免 begin-end 块）：

```
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;
```

Object Pascal 中的 Exit 是一个函数，因此您必须将要返回的值括在括号中，而使用 C 样式语言的 return 则是不需要括号的编译器关键字。

4.2.2 Reference Parameters（参考参数）

在 Object Pascal 中，过程和函数允许按值和按引用传递参数。按值传递参数是默认值：将值复制到堆栈上，并且例程使用并操纵该数据副本，而不是原始值（正如我之前在"函数参数和返回值"部分中所述）。通过引用传递参数意味着它的值不会在例程的形式参数中复制到堆栈上。相反，程序也在例程代码中引用原始值。这允许过程或函数更改作为参数传递的变量的实际值。通过引用传递的参数由 var 关键字表示。

在大多数编程语言中也可以使用此技术，因为避免复制通常意味着程序执行速度更快。它在 C 中不存在（您只能使用指针），但是它是在 C++和其他基于 C 语法的语言中引入的，您可以在其中使用＆（按引用传递）符号。这是使用 var 关键字通过引用传递参数的示例：

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

在这种情况下，该参数既可用于将值传递给过程，也可用于将新值返回给调用代码。当你写：

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

X 变量的值变为 20，因为该函数使用对 X 原始存储位置的引用，从而影响了 X 的原始值。

与常规参数传递规则相比，将值传递给参考参数受制于更严格的规则，因为传递的不是值，而是实际变量。您不能将常量值作为参考参数，表达式，函数的结果或属性传递。另一个规则是，您不能传递类型稍有不同的变量（要求自动转换）。变量和参数的类型必须完全匹配，或者如编译器错误消息所述：

[dcc32 Error] E2033 Types of actual and formal var parameters must be identical（[dcc32 错误] E2033 实际和形式 var 参数的类型必须相同）

例如，这是您编写时将收到的错误消息（这也是 ParamsTest 应用程序项目的一部分，但已注释掉）：

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

通过引用传递参数对于顺序类型和记录是有意义的（我们将在下一章中看到）。这些类型通常称为值类型，因为它们默认情况下具有按值传递和按值分配的语义。

Object Pascal 对象和字符串的行为稍有不同，我们将在后面详细研究。对象变量是引用，因此您可以修改作为参数传递的对象的实际数据。这些类型是不同组的一部分，通常表示为参考类型。

除了标准和参考（var）参数类型外，Object Pascal 还有一种非常不寻常的参数说明符。out 参数没有初始值，仅用于返回值。除了没有初始值外，out 参数的行为类似于 var 参数。

❖ 引入了 out 参数以支持 Windows 的 Component Object 模型（或 COM）中的相应概念。它们可用于表达开发人员期望未初始化值的意图。

4.2.3 Constant Parameters（常数参数）

作为参考参数的替代方法，可以使用 const 参数。由于您不能在例程中为常数参数分配新值，因此编译器可以优化参数传递。编译器可以选择一种与引用参数（或 C++术语中的 const 引用）相似的方法，但是行为将与值参数相似，因为该函数无法修改原始值。

实际上，如果您尝试编译以下代码（可用，但已在 ParamsTest 项目中注释掉），则系统将发出错误：

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2; //  编译器错误
  Result := Value;
end;
```

您将看到的错误消息可能不是立即直观的，因为它说：

[dcc32 Error] E2064 Left side cannot be assigned to
（[dcc32 错误] E2064 左侧无法分配给）

常量参数对于字符串来说非常常见，因为在这种情况下，编译器可以禁用引用计数机制，从而进行一些优化。

这是使用常量参数的最常见原因，该功能对于序数和标量类型意义有限。

❖ 除了传递 const 参数外，还有另一种鲜为人知的替代方法，即向其添加 ref 属性，例如在“const[ref]”中。该属性强制编译器通过引用传递常量参数，默认情况下，编译器将根据参数的大小选择按值或引用传递常量参数，结果取决于目标 CPU 和平台。

4.2.4 Function Overloading（函数重载）

有时，您可能希望拥有两个非常相似的函数，它们具有不同的参数和不同的实现。传统上，每种名称都必须略有不同，但现代的编程语言却允许您使用多个定义重载符号。

重载的想法很简单：只要参数不同，编译器就可以使用相同的名称定义两个或多个函数或过程。实际上，通过检查参数，编译器可以确定要调用的函数版本。

考虑从运行时库的 System.Math 单元提取的以下一系列功能：

```
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;
```

当您调用 Min（10，20）时，编译器确定您正在调用该组的第一个函数，因此返回值也将是 Integer。

有两个基本的重载规则：

- 每个版本的重载函数（或过程）后均必须带有重载关键字（包括第一个）。
- 在重载的功能中，参数的数量或类型必须有所不同。不考虑参数名称，因为在调用过程中未指出参数名称。同样，返回类型不能用于区分两个重载函数。

❖ 您不能区分返回值上的函数的规则是一个例外，它适用于第 5 章中介绍的隐式和显式转换运算符。

这是我已添加到 OverloadTest 示例（展示重载和默认参数）中的 ShowMsg 过程的三个重载版本：

```
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
  Show (I.ToString + ' ' + Str);
end;
```

这三个函数会显示一个带有字符串的消息框，可以选择以其他方式设置字符

串格式。这是程序的三个调用：

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

这就是他们的效果：

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

✧ IDE 的代码参数技术可以很好地与重载的过程和函数配合使用。当您在例程名称后键入打开的括号时，将列出所有可用的替代项。输入参数时，Code Insight 技术将使用其类型来确定哪些替代方法仍然可用。

如果您尝试使用与任何可用的重载版本都不匹配的参数来调用函数，该怎么办？ 当然，您会收到一条错误消息。假设您要呼叫：

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

在这种情况下，您将看到的错误是一个非常具体的错误：

[dcc32 Error] E2250 There is no overloaded version of 'ShowMsg' that can be called with these arguments（[dcc32 错误] E2250 没有可通过这些参数调用的'ShowMsg'重载版本）

必须正确标记每个版本的重载例程这一事实意味着您不能重载未使用重载关键字标记的同一单元的现有例程。

尝试时收到的错误消息是：

Previous declaration of '<name>' was not marked with the 'overload' directive.（先前的“ <名称>”声明未使用“过载”指令标记。）

但是，您可以创建一个例程，该例程的名称与在另一个单元中声明的例程的名称相同，因为该例程充当名称空间。在这种情况下，您不是要用新版本重载某个功能，而是要用新版本替换该功能，而将原始版本隐藏起来（可以使用前缀单元名称来引用）。这就是为什么编译器无法根据参数选择版本，但是它将尝试匹配唯一的版本，如果参数类型不匹配，则会发出错误。

4.2.5 重载和不明确的调用

当您调用重载函数时，如果没有一个重载版本具有正确的参数（如我们刚刚看到的），则编译器通常会找到匹配项并正常工作或发出错误。

但是，还有第三种情况：如果编译器可以对函数的参数进行一些类型转换，则单个调用可能会有不同的可能转换。当编译器找到一个可以调用的函数的多个版本，并且没有一个完美的类型匹配（将被选中）时，它会发出一条错误消息，指出该函数调用不明确。

这不是常见的情况，我不得不建立一个不合逻辑的示例来向您展示它，但是值得考虑这种情况（因为它确实在现实世界中偶尔发生）。

假设您决定实现两个重载函数以添加整数和浮点数：

```
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;
```

这些功能在 OverloadTest 应用程序项目中。现在，您可以调用它们以任意顺序传递两个参数：

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

但是事实是，通常，函数在进行转换时可以接受不同类型的参数，就像当函数需要浮点类型的参数时接受整数一样。那么，如果您调用：

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

编译器可以调用重载函数的第一个版本，但也可以调用第二个版本。如果不知道你在要求什么（并且知道调用一个函数或另一个函数会产生相同的效果），它会发出一个错误

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'Add'
  Related method: function Add(Integer; Single): Single;
  Related method: function Add(Single; Integer): Single;
（[dcc32 错误] E2251 对“Add”的歧义重载调用。
  相关方法：函数 Add(Integer; Single): Single;
  相关方法：函数 Add(Single; Integer): Single;）
```

✧ 在 IDE 的错误窗格中，您将看到一条错误消息，上面带有第一行，并且在侧面可以看到加号，可以展开以查看以下两行，并详细说明编译器正在考虑哪些重载函数是模棱两可的。

如果这是现实情况，并且您需要进行调用，则可以添加一个手动类型转换调用来解决该问题，并向编译器指示您要调用该函数的哪个版本：

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

如果使用 variants（变体），则可能会发生歧义调用的特殊情况，这种变体是一种相当奇怪的数据类型，我将在本书后面再介绍。

4.2.6 默认参数

与重载有关的另一个功能是可以为函数的某些参数提供默认值，以便您可以在有或没有这些参数的情况下调用该函数。如果调用中缺少该参数，它将采用默认值。

让我展示一个示例（仍然是 OverloadTest 应用程序项目的一部分）。我们可以定义以下 Show 调用的封装，提供两个默认参数：

```
procedure NewMessage (Msg: string; Caption: string = '信息';
  Separator: string = ': ');
begin
  Show (Caption + Separator + Msg);
end;
```

使用此定义，我们可以通过以下每种方式调用该过程：

```
NewMessage ('这里出问题了！ ');
NewMessage ('这里出了点问题！ '，'注意');
NewMessage ('你好', '消息', '--');
```

这是输出：

```
信息: 这里出问题了!
注意: 这里出了点问题!
消息--你好
```

注意，编译器不会生成任何特殊代码来支持默认参数。它也不会创建函数或过程的多个（重载）副本。缺少的参数仅由编译器添加到调用代码中。有一个重

要的限制会影响默认参数的使用：您不能“跳过”参数。例如，在省略第二个参数后，您无法将第三个参数传递给函数。

对于具有默认参数的函数和过程（以及方法）的定义和调用，还有其他一些规则：

- 在通话中，只能从上一个开始省略参数。换句话说，如果省略参数，则还必须省略以下参数。
- 在定义中，具有默认值的参数必须在参数列表的末尾。
- 默认值必须是常量。显然，这限制了您可以使用默认参数的类型。例如，动态数组或接口类型不能具有除 nil 之外的默认参数；默认值为 0。记录根本无法使用。
- 具有默认值的参数必须通过值或 const 传递。引用（var）参数不能具有默认值。

如上一节所述，同时使用默认参数和重载使您更有可能陷入使编译器感到困惑的境地，从而引发模棱两可的调用错误。例如，如果我将以下新版本的 NewMessage 过程添加到前面的示例中：

```
procedure NewMessage (Str: string; I: Integer = 0); overload;
begin
  Show (Str + ': ' + IntToStr (I))
end;
```

那么编译器就不会抱怨，因为这是一个合法的定义。但是，如果您编写调用：

```
NewMessage ('你好');
```

编译器将其标记为：

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'NewMessage'
  Related method: procedure NewMessage(string; string; string);
  Related method: procedure NewMessage(string; Integer);
（[dcc32 错误] E2251 对'NewMessage'的歧义重载调用
  相关方法：过程 NewMessage(string; string; string);
  相关方法：过程 NewMessage(string; Integer);）
```

请注意，在新的重载定义之前正确编译的一行代码中显示此错误。在实践中，我们无法用一个字符串参数调用 NewMessage 过程，因为编译器不知道我们是否只想用字符串参数调用版本，还是想用字符串参数和整数参数调用默认值的版本。 如果有类似的疑问，编译器将停止并要求程序员更清楚地陈述其意图。

4.3 Inlining（内联）

内联 Object Pascal 函数和方法是一种低级语言功能，可以导致重大的优化。通常，当您调用方法时，编译器会生成一些代码，以使您的程序跳至新的执行点。这意味着要设置堆栈框架并执行更多操作，并且可能需要十几个机器指令。但是，您执行的方法可能很短，甚至可能只是简单地设置或返回某些私有字段的访问方法。

在这种情况下，将实际代码复制到调用位置非常有意义，避免了堆栈框架设置和其他所有操作。通过消除这些开销，您的程序将运行得更快，尤其是在执行数千次的紧密循环中进行调用时。

对于某些非常小的函数，生成的代码甚至可能更小，因为粘贴到位的代码可能比函数调用所需的代码小。但是，请注意，如果内联一个较长的函数，并且在程序中的许多不同位置调用了此函数，则可能会遇到代码膨胀，这是可执行文件大小不必要的增加。

在 Object Pascal 中，您可以要求编译器使用 inline 指令将函数（或方法）内联，该指令位于函数（或方法）声明之后。 不必在定义中重复此指令。 始终要记住，内联指令仅是对编译器的提示，编译器可以确定该函数不是内联的理想选择，并跳过您的请求（不以任何方式警告您）。

编译器还可以在分析调用代码后并根据$ INLINE 指令在调用位置的状态，内联该函数的一些（但不一定是全部）调用。 该伪指令可以采用三个不同的值（请注意，此功能独立于优化编译器开关）：

- 使用{$ INLINE OFF}可以禁止在程序，程序的一部分或特定的调用站点中进行内联，而不管被调用函数中是否存在内联指令。
- 使用默认值{$ INLINE ON}，启用由 inline 指令标记的功能的内联。
- 使用{$ INLINE AUTO}时，编译器通常会内联用指令标记的函数，以及自动内嵌非常短的函数。注意，因为此指令可能导致代码膨胀。

Object Pascal 运行时库中有许多功能已标记为嵌入式候选。例如，System.Math 单元的 Max 函数具有如下定义：

```
function Max(const A, B: Integer): Integer; overload; inline;
```

为了测试内联此功能的实际效果，我在 InliningTest 应用程序项目中编写了以下循环：

```
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 + '] ');
```

在此代码中，System.Diagnostics 单元的 TStopWatch 记录是一种结构，用于跟踪 Start（或 StartNew）和 Stop 调用之间经过的时间（或系统滴答）。

表单有两个按钮，两个按钮都调用相同的确切代码，但是其中一个按钮已在调用站点上禁用内联。请注意，您需要使用 Release 配置进行编译才能看到任何差异（因为内联是 Release 优化）。通过两千万次交互（LoopCount 常数的值），在我的计算机上得到以下结果：

```
// 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]
```

我们如何读取这些数据？在 Windows 上，内联使执行速度提高了一倍以上，而在 Android 上，内联使程序速度提高了约 35％。但是，在设备上，程序运行速度要慢得多（一个数量级），因此，在 Windows 上，我们在 Android 设备上节省了 30 毫秒的时间，而这种优化节省了大约 100 毫秒的时间。

同一程序使用 Length 函数进行第二次类似测试，Length 函数是专门修改为内联的编译器魔术函数。同样，内联版本在 Windows 和 Android 上都明显更快：

```
// 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]
```

这是第二个测试循环使用的代码：

```
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 编译器没有为可以内联的函数的大小或防止内联的特定结构列表（for 或 while 循环，条件语句）定义明确的限制。但是，由于内联大型函数几乎没有优势，但是却使您面临某些实际劣势的风险（就代码膨胀而言），因此应避免这样做。

一种限制是方法或函数不能引用在单元的实现部分中定义的标识符（例如类型，全局变量或函数），因为在调用位置将无法访问它们。但是，如果您正在调用也恰好是内联的局部函数，则编译器将接受您的内联例程的请求。

缺点是内联需要对单元进行更频繁的重新编译，因为当您修改内联函数时，每个调用站点的代码也将需要重新编译。在一个单元中，您可以在调用内联函数之前编写它们的代码，但最好将它们放在实现部分的开头。

❖ Delphi 具有单程编译器，因此它无法引用尚未见过的函数的代码。

在不同的单元中，即使您不直接调用这些方法，也需要将其他具有内联函数的单元专门添加到 uses 语句中。 假设您的单元 A 调用了在单元 B 中定义的内联函数。如果此函数又调用了单元 C 中的另一个内联函数，则您的单元 A 也需要引用 C。否则，您会看到一个编译器警告，指示由于缺少单元引用，因此未内联该调用。一个相关的效果是，当存在循环单元引用时（通过其实现部分），函数永远不会内联。

4.4 函数的高级功能

如果到目前为止，我所介绍的内容包括与函数相关的核心功能，那么有几项值得探讨的高级功能。但是，如果您真的是软件开发方面的新手，则可能需要暂时跳过本章的其余部分，然后转到下一章。

4.4.1 Object Pascal 调用约定

每当您的代码调用函数时，双方都需要就将参数从调用方传递到被调用方的实际可行方式达成共识，这称为调用约定。通常，通过堆栈存储区传递参数（并期望返回值）来进行函数调用。但是，参数和返回值在堆栈上的放置顺序会根据编程语言和平台而变化，大多数语言都可以使用多种不同的调用约定。

很久以前，Delphi 的 32 位版本引入了一种新的传递参数的方法，称为“fastcall

（快速调用）”：只要有可能，最多可以在 CPU 寄存器中传递三个参数，从而使函数调用快得多。默认情况下，Object Pascal 使用此快速调用约定，尽管也可以使用 register 关键字来请求它。

Fastcall 是默认的调用约定，使用它的函数与外部库不兼容，例如 Win32 中的 WindowsAPI 函数。必须使用 stdcall（标准调用）调用约定，Win16API 的原始 pascal 调用约定和 C 语言的 cdecl 调用约定的组合来声明 Win32API 的功能。所有这些调用约定在 Object Pascal 中都受支持，但是除非您需要调用以其他语言编写的库（例如系统库），否则很少使用不同于默认值的东西。

您需要脱离默认的快速调用约定的典型情况是，当您需要调用平台的本机 API 时，该平台需要不同的调用约定，具体取决于操作系统。甚至 Win64 也使用与 Win32 不同的模型，因此 Object Pascal 支持许多不同的选项，在这里真的不值得详细介绍。相反，移动操作系统倾向于公开类而不是本机函数，尽管即使在这种情况下也必须考虑遵守给定调用约定的问题。

4.4.2 程序类型

Object Pascal 的另一个功能是过程类型的存在。这些确实是一个高级语言主题，只有很少的程序员会定期使用。

但是，由于我们将在后面的章节中讨论相关主题（特别是方法指针，环境大量用于定义事件处理程序的技术以及匿名方法），因此值得在此快速进行介绍。

在 Object Pascal 中（但不是在更传统的 Pascal 语言中），存在过程类型的概念（类似于 C 语言的函数指针的概念 - 诸如 C# 和 Java 之类的语言已被删除，因为它与全局变量相关联）函数和指针）。

程序类型的声明指示参数列表，对于函数，则指示返回类型。例如，您可以使用以下代码声明新的过程类型，该类具有通过引用传递的 Integer 参数：

```
type
  TIntProc = procedure (var Num: Integer);
```

此过程类型与具有完全相同的参数（或使用 C 行话的相同函数签名）的任何例程兼容。这是一个兼容例程的示例：

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

程序类型可以用于两个不同的目的：可以声明程序类型的变量或将程序类型（即函数指针）作为参数传递给另一个例程。给定前面的类型和过程声明，您可以编写以下代码：

```
var
  IP: TIntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

此代码与以下较短的版本具有相同的作用：

```
var
  X: Integer;
begin
```

```
    X := 5;
    DoubleTheValue (X);
  end;
```

第一个版本显然更复杂，那么为什么以及何时应该使用它呢？在某些情况下，能够决定要调用哪个函数并在以后实际调用它会非常强大。可以构建一个显示此方法的复杂示例。但是，我更喜欢让您探索一个相当简单的应用程序项目，称为 ProcType。

本示例基于两个过程。像我已经展示的那样，使用一种过程将参数的值加倍。第二个过程用于将参数的值增加三倍，因此命名为 TripleTheValue：

```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
end;
```

而不是直接调用这些函数，一个或另一个保存在过程类型变量中。当用户选择一个复选框时，将修改该变量，并在用户单击按钮时以这种通用方式调用当前过程。该程序使用两个初始化的全局变量（要调用的过程和当前值），以便随着时间的推移保留这些值。这是完整的代码，除了上面已经显示的实际过程的定义之外：

```
var
  IntProc: TIntProc = DoubleTheValue;
  Value: Integer = 1;

procedure TForm1.CheckBox1Change(Sender: TObject);
begin
 if CheckBox1.IsChecked then
    IntProc := TripleTheValue
 else
    IntProc := DoubleTheValue;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  IntProc (Value);
  Show (Value.ToString);
end;
```

当用户更改复选框状态时，所有随后的按钮单击都将调用活动函数。 因此，如果按两次按钮，更改选择，然后再次按两次按钮，则将首先使当前值翻倍两次，然后将当前值翻三倍，从而产生以下输出：

```
2
4
12
36
```

使用过程类型的另一个实际示例是，当您需要将函数传递给 Windows 之类的操作系统时（通常将它们称为“回调函数”）。如本节开头所述，除对象类型外，Object Pascal 开发人员还使用方法指针（在第 10 章中介绍）和匿名方法（在第 15 章中介绍）。

- ❖ <u>获得后期绑定函数调用（即可以在运行时更改的函数调用）的最常见的面向对象机制是使用虚拟方法。虽然虚拟方法在 Object Pascal 中非常普遍，但很少使用过程类型。但是，技术基础在某种程度上是相似的。虚函数和多态性在第 8 章中介绍。</u>

4.4.3 外部函数声明

系统编程的另一个重要元素是外部声明。最初用于将代码链接到用汇编语言编写的外部函数，外部声明已成为 Windows 编程中常见的从 DLL（动态链接库）中调用函数的程序。外部函数声明意味着能够调用对编译器或链接器不完全可用的函数，但是需要加载外部动态库并调用其函数之一的能力。

❖ 每当您在 Object Pascal 代码中为给定平台调用 API 时，就会失去针对特定平台以外的任何其他平台重新编译应用程序的能力。如果调用被平台特定的 $ IFDEF 编译器指令包围，则为例外。

例如，这就是如何从 Delphi 应用程序调用 Windows API 函数。如果打开 Winapi.Windows 单元，则会发现许多函数声明和定义，例如：

```
// forward declaration（前瞻性声明）
function GetUserName(lpBuffer: LPWSTR; var nSize: DWORD): BOOL; stdcall;
// external declaration (instead of actual code（而不是实际的代码）)
function GetUserName; external advapi32 name 'GetUserNameW';
```

您很少需要像刚刚说明的那样编写声明，因为它们已经在 Windows 单元和许多其他系统单元中列出。您可能需要编写此外部声明代码的唯一原因是从自定义 DLL 调用函数，或调用未在平台 API 中转换的 Windows 函数。

此声明意味着函数 GetUserName 的代码存储在 advapi32 动态库中（advapi32 是与 DLL 全名“advapi32.dll”相关联的常数），其名称为 GetUserNameW，因为此 API 函数同时具有 ASCII 码和 WideString 版本。实际上，在外部声明中，我们可以指定函数引用原始名称不同的 DLL 函数。

延迟加载 DLL 函数

在 Windows 操作系统中，有两种方法可以调用 Windows SDK（或任何其他 DLL）的 API 函数：您可以让应用程序加载程序解析对外部函数的所有引用，也可以编写用于查找函数的特定代码，如果可用，执行它。

前一个代码更容易编写（正如我们在上一节中看到的）：因为您只需要外部函数声明。但是，如果库或您要调用的函数中的一个不可用，则您的程序将无法在不提供该函数的操作系统版本上启动。

动态加载可提供更大的灵活性，但意味着使用 GetProcAddress API 查找要调用的函数并在将指针转换为正确的类型后调用它，从而手动加载库。这种代码非常麻烦且容易出错。

这就是为什么 Object Pascal 编译器和链接器特别支持现在在 Windows 操作系统级别可用并且已被某些 C ++编译器使用的功能（将功能延迟加载到调用它们之前）的特定支持。声明的目的不是要避免 DLL 的隐式加载（无论如何都会发生），而是允许 DLL 中该特定功能的延迟绑定。

您基本上以与 DLL 函数的经典执行非常类似的方式编写代码，但是函数地址是在第一次调用该函数时而不是在加载时被解析的。这意味着，如果该功能不可用，您将获得运行时异常 EExternalException。但是，通常，您可以验证操作系统的当前版本或要调用的特定库的版本，并事先确定是否要进行调用。

❖ 如果您想在全局层面上比异常更具体，更容易处理，您可以加入延迟加载调用的错误机制，如 Allen Bauer 在他的博客文章中所述：
https://blog.therealoracleatdelphi.com/2009/08/exceptional-procrastination_29.html

从 Object Pascal 语言的角度来看，唯一的区别在于外部函数的声明。而不是写：

```
function MessageBox; external user32 name 'MessageBoxW';
```

您现在可以编写（同样，从 Windows 单元中的实际示例中）：

```
function GetSystemMetricsForDpi(nIndex: Integer; dpi: UINT): Integer;
  stdcall; external user32 name 'GetSystemMetricsForDpi' delayed;
```

在运行时，考虑到该 API 是第一次添加到 Windows 10 版本 1607 中，您可能需要编写如下代码：

```
if (TOSVersion.Major >= 10) and (TOSVersion.Build >= 14393)
begin
  NMetric := GetSystemMetricsForDpi (SM_CXBORDER, 96);
```

这比无需延迟加载就可以在较旧版本的 Windows 上运行相同程序的代码要少得多。

另一个相关的观察结果是，在构建自己的 DLL 并在 Object Pascal 中调用它们时，可以使用相同的机制，只要您对新函数使用延迟加载，就可以提供可以绑定到同一 DLL 多个版本的单个可执行文件。

第五章 arrays（数组）和 records（记录）

在第 2 章中介绍数据类型时，我提到了一个事实，即 Object Pascal 中既有内置数据类型又有类型构造函数。类型构造函数的一个简单示例是本章介绍的枚举类型。

类型定义的真正力量在于更高级的机制，例如数组，记录和类。在本章中，我将介绍前两个，它们的本质可以追溯到 Pascal 的早期定义，但是多年来已经进行了很大的更改（并且功能如此强大），以至于它们几乎都与祖先的类型构造函数相似名称。

在本章结束时，我还将简要介绍一些高级的 Object Pascal 数据类型作为指针。但是，自定义数据类型的真正功能将在第 7 章中介绍，我们将开始研究类和面向对象的编程。

5.1 数组数据类型

数组类型使用特定类型的元素定义列表。这些列表可以具有固定数量的元素（静态数组）或可变数量的元素（动态数组）。通常，您可以使用方括号内的索引来访问数组的元素之一。方括号也用于指定固定大小数组的值的数量。

Object Pascal 语言支持不同的数组类型，从传统的静态数组到动态数组。建议使用动态数组，尤其是在编译器的移动版本中。我将首先介绍静态数组，然后再介绍动态数组。

5.1.1 Static Arrays（静态数组）

传统的 Pascal 语言数组定义为静态或固定大小。以下代码段中的一个示例定义了一个由 24 个整数组成的列表，这些代码段显示了一天 24 小时的温度：

```
type
  TDayTemperatures = array [1..24] of Integer;
```

在这种经典的数组定义中，可以在方括号内使用子范围类型，实际上是使用两个序数类型的常量来定义新的特定子范围类型。此子范围指示数组的有效索引。由于您同时指定了数组的上下索引，因此索引不必从零开始，这在 C，C ++，Java 和大多数其他语言中都是这样（尽管从 0 开始的数组也是如此）在 Pascal 对象中非常常见）。还要注意，Object Pascal 中的静态数组索引可以是数字，也可以是其他序数类型，例如字符，枚举类型等。但是，非整数索引非常少见。

❖ 有些语言（例如 JavaScript）大量使用关联数组。Object Pascal 数组仅限于顺序索引，因此您不能直接使用字符串作为索引。RTL 中已经可以使用实现字典的数据结构以及提供此类功能的其他类似数据结构。我将在本书的第三部分中有关泛型的一章中介绍它们。。

由于数组索引基于子范围，因此编译器可以检查其范围。

无效的常量子范围会导致编译时错误；并且在运行时使用超出范围的索引会导致运行时错误，但前提是启用了相应的编译器选项。

❖ 这是 IDE 的“Project Options（项目选项）”对话框的“Compiling（编译）”页面的“Runtime errors（运行时错误）”组的“Range checking（范围检查）”

选项。我已经在第 2 章“子范围类型”部分中提到了此选项。

使用上面的数组定义，您可以按如下方式设置 TDayTemperatures 类型的 DayTemp1 变量的值（就像我在 ArraysTest 应用程序项目中所做的那样，从中提取了以下代码片段）：

```
type
  TDayTemperatures = array [1..24] of Integer;
var
  DayTemp1: TDayTemperatures;
begin
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;
  //  以下行会导致：
  // E1012 常量表达式违反了子范围的界限
  // DayTemp1 [25] := 67;
```

现在，考虑到数组的性质，对数组进行操作的标准方法是用于循环。

这是一个循环的示例，用于显示一天的所有温度：

```
var
  I: Integer;
begin
  for I := 1 to 24 do
    Show (I.ToString + ': ' + DayTemp1[I].ToString);
```

尽管此代码有效，但对数组边界（1 和 24）进行硬编码并不理想，因为数组定义本身可能会随着时间而改变，并且您可能希望使用动态数组。

5.1.2 数组大小和边界

使用数组时，您始终可以使用标准的 Low 和 High 函数来测试其边界，这些函数会返回上下边界。强烈建议在对数组进行操作时使用“Low”和“High”，尤其是在循环中，因为它使代码独立于数组的当前范围（它可能从 0 到数组的长度减去 1，可能从 1 开始到达数组的长度，或者有任何其他子范围定义）。如果以后要更改数组索引的声明范围，则使用 Low 和 High 的代码仍然可以使用。如果您编写一个对数组范围进行硬编码的循环，则必须在阵列大小发生变化时更新循环代码。Low（低）和 High（高）使您的代码更易于维护和更可靠。。

❖ 顺便说一下，将 Low 和 High 与静态数组一起使用没有运行时开销。它们在编译时解析为常量表达式，而不是实际的函数调用。表达式和函数调用的这种编译时解析也发生在许多其他系统函数中。

另一个相关的函数是 Length，它返回数组元素的数量。我在下面的代码中结合了这三个功能，这些代码计算并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);
```

此代码也是 ArraysTest 应用程序项目的一部分。

5.1.3 多维静态数组

数组可以具有多个维，表示矩阵或立方体而不是列表。这是两个示例定义：

```
type
  TAllMonthTemps = array [1..24, 1..31] of Integer;
  TAllYearTemps = array [1..24, 1..31, 1..12] of Integer;
```

您可以通过以下方式访问元素：

```
var
  AllMonth1: TAllMonthTemps;
  AllYear1: TAllYearTemps;
begin
  AllMonth1 [13, 30] := 55; // hour, day
  AllYear1 [13, 30, 8] := 55; // hour, day, month
```

- 静态数组会立即占用大量内存（在上面的堆栈中），这应该避免。 AllYear1 变量需要 8,928 个整数，每个整数占用 4 个字节，几乎为 35KB。在全局内存或堆栈中分配如此大的块（如在演示代码中）确实是一个错误。相反，动态数组使用堆内存，并在内存分配和管理方面提供了更大的灵活性。

鉴于这两种数组类型是基于相同的核心类型构建的，因此您最好使用前面的数据类型声明它们，如以下代码所示：

```
TMonthTemps = array [1..31] of TDayTemperatures;
TYearTemps = array [1..12] of TMonthTemps;
```

该声明如上所述反转了索引的顺序，但是它也允许在变量之间分配整个块。让我们看看如何分配单个值：

```
Month1 [30][14] := 44;
Month1 [30, 13] := 55; // day, hour
Year1 [8, 30, 13] := 55; // month, day, hour
```

理论上，您应该使用第一行，选择一个数组的数组，然后选择结果数组的元素。但是，也可以使用两个索引放在方括号内的版本。或在“多维数据集”示例中使用三个索引。

使用中间类型的重要性在于，只有在数组引用相同的确切类型名称（即完全相同的类型定义）的情况下，数组才是类型兼容的，而不是在它们的类型定义碰巧引用相同的实现时才是兼容的。

该类型兼容性规则对于 Object Pascal 中的所有类型都是相同的，只有一些特定的例外。

例如，以下语句将一个月的温度复制到一年中的第三个月：

```
Year1[3] := Month1;
```

相反，基于独立数组定义（类型不兼容）的类似语句：

```
AllYear1[3] := AllMonth
```

会导致错误：

Error: Incompatible types: 'array[1..31] of array[1..12] of Integer' and 'TAllMonthTemps（错误：不兼容的类型：“整数[1..12]的数组[1..31]”和“TAllMonthTemps”）

如前所述，静态数组会遇到内存管理问题，特别是当您要将它们作为参数传递或仅分配大数组的一部分时。

此外，您无法在数组变量的生存期内调整它们的大小。这就是为什么最好使用动态数组，即使它们需要一点额外的管理，例如关于内存分配。

5.1.4 Dynamic Arrays（动态数组）

在传统的 Pascal 语言中，数组具有固定大小的数组，并且您在声明数据类型时指定了数组的元素数。Object Pascal 还支持动态数组的直接和本地实现。

❖ 这里的“动态数组的直接实现”与使用指针和动态内存分配来获得类似的效果相反……使用非常复杂且容易出错的代码。顺便说一下，在大多数现代编程语言中，动态数组是该构造的唯一形式。

动态数组是动态分配和引用计数的（使参数传递更快，因为仅传递引用，而不是完整数组的副本）。完成后，可以通过将变量的变量设置为 nil 或将其长度设置为零来清除数组，但是在大多数情况下，鉴于对它们的引用计数，编译器将自动为您释放内存。

对于动态数组，可以在不指定元素数量的情况下声明数组类型，然后使用 SetLength 过程以给定的大小分配它：

```
var
  Array1: array of Integer;
begin
  //  这会导致运行时范围检查错误
  // Array1 [1] := 100;
  SetLength (Array1, 10);
  Array1 [1] := 100; // this is OK
```

您必须先分配数组的长度，然后在堆上分配所需的内存，然后才能使用该数组。如果这样做，您将看到范围检查错误（如果相应的编译器选项处于活动状态），或者在其他平台上出现访问冲突（在 Windows 上）或类似的内存访问错误。SetLength 调用将所有值设置为零。初始化代码可以立即开始读取和写入数组的值，而不必担心内存错误（除非您违反了数组边界）。

如果确实需要显式分配内存，则无需直接释放它。在上面的代码段中，随着代码终止和 Array1 变量超出范围，编译器将自动释放其内存（在这种情况下，将分配十个整数）。因此，尽管您可以将动态数组变量分配给 nil 或使用 0 值调用 SetLength，但通常不需要（而且很少这样做）。

请注意，SetLength 过程也可以用于调整数组的大小，而不会丢失其当前内容（如果正在增长），但是会丢失一些元素（如果正在收缩）。就像在最初的 SetLength 调用中一样，您仅指示数组的元素数，动态数组的索引始终从 0 开始，一直到元素数减 1。换句话说，动态数组不支持两个功能。经典静态 Pascal 数组，非零低界和非整数索引。同时，它们更紧密地匹配基于 C 语法的数组在大多数语言中的工作方式。

与静态数组一样，要了解动态数组的当前大小，可以使用 Length，High 和 Low 函数。但是，对于动态数组，Low 始终返回 0，High 始终返回长度减去 1。这意味着对于一个空数组，High 返回-1（当您考虑它时，这是一个奇怪的值，因为它比 Low 返回的值低）。

因此，作为一个示例，在 DynArray 应用程序项目中，我已使用自适应循环从动态数组中填充并提取了信息。这是类型和变量定义：

```
type
  TIntegersArray = array of Integer;
var
  IntArray1: TIntegersArray;
```

使用以下循环为数组分配和填充与索引匹配的值：

```
var
```

```
    I: Integer;
begin
    SetLength (IntArray1, 20);
    for I := Low (IntArray1) to High (IntArray1) do
        IntArray1 [I] := I;
end;
```

第二个按钮具有用于显示每个值和计算平均值的代码，类似于上一个示例，但在一个循环中：

```
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 ('平均: ' + (Total / Length(IntArray1)).ToString);
end;
```

这段代码的输出非常明显（并且几乎被省略了）：

```
0: 0
1: 1
2: 2
3: 3
...
17: 17
18: 18
19: 19
平均: 9.5
```

除了“Length（长度）”，“SetLength（设置长度）”，“Low（低）”和“High（高）”外，还可以在数组上使用其他常见过程，例如“copy”函数，可以用来复制数组的一部分（或全部）。请注意，您还可以将一个数组从一个变量分配给另一个数组，但是在这种情况下，您不是在进行完整复制，而是在内存中有两个引用同一数组的变量。

唯一稍微复杂的代码是 DynArray 应用程序项目的最后一部分，该代码以两种不同的方式将一个数组复制到另一个数组：

- 使用复制功能，该功能使用单独的存储区在新数据结构中复制阵列数据。
- 使用赋值运算符，该赋值运算符有效地创建别名，即引用内存中相同数组的新变量。

此时，如果您修改新数组的元素之一，则将取决于复制的方式而影响原始版本或不影响原始版本。这是完整的代码：

```
var
    IntArray2: TIntegersArray;
    IntArray3: TIntegersArray;
begin
    // 别名
    IntArray2 := IntArray1;
    // separate copy（单独的副本）
    IntArray3 := Copy (IntArray1, Low(IntArray1), Length(IntArray1));
    // modify items（修改栏目）
    IntArray2 [1] := 100;
    IntArray3 [2] := 100;
```

```
// 检查每个数组的值
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]]));
```

您将获得的输出如下所示：

```
[1] 100 -- 100 -- 1
[2] 2 -- 2 -- 100
```

对 IntArray2 所做的更改将传播到 IntArray1，因为它们只是对同一物理数组的两个引用；因此，对它们的更改不会生效。IntArray3 的更改是单独的，因为它具有单独的数据副本。

动态阵列的本机运算

动态数组支持分配常量数组和连接。

- ❖ 这些对动态数组的扩展是在 Delphi XE7 中添加的，有助于使开发人员感到此功能很有价值，并且是显而易见的选择（而不必考虑它们的静态对应项）。

实际上，您可以编写类似以下的代码，与以前的代码段相比，该代码已大大简化：

```
var
   DI: array of Integer;
   I: Integer;
begin
   DI := [1, 2, 3]; // 初始化
   DI := DI + DI; // concatenation（级联）
   DI := DI + [4, 5]; // mixed concatenation（混合级联）
   for I in DI do
   begin
      Show (I.ToString);
   end;
```

注意，在此代码中使用了 for-in 语句来扫描数组元素，这是 DynArrayConcat 应用程序项目的一部分。请注意，这些数组可以基于任何数据类型，从上面的代码中的简单整数到记录和类，都可以。

除了赋值和串联之外，还有第二个附加功能，但这是 RTL 的一部分，而不是语言。现在可以在动态数组中使用字符串常用的函数，例如插入和删除。

这意味着您现在可以编写如下代码（属于同一项目）：

```
var
   DI: array of Integer;
   I: Integer;
begin
   DI := [1, 2, 3, 4, 5, 6];
   Insert ([8, 9], DI, 4);
   Delete (DI, 2, 1); // 删除第三个（从 0 开始）
```

5.1.5 开放数组参数

使用数组有一种非常特殊的情况，它将一个灵活的参数列表传递给一个函数。除了直接传递数组外，本节和下一节还将介绍两种特殊的语法结构。顺便说一句，此类函数的一个示例是我在上一个代码段中调用的 Format 函数，该函数在方括号中具有一个值数组作为第二个参数。

与 C 语言（以及其他一些基于 C 语法的语言）不同，在传统的 Pascal 语言中，函数或过程始终具有固定数量的参数。但是，在 Object Pascal 中，存在一种使用

数组作为参数将可变数量的参数传递给例程的方法，该技术称为开放数组参数。

❖ 从历史上看，开放数组参数早于动态数组，但是今天这两个功能在工作方式上看起来非常相似，以至于如今几乎无法区分。这就是为什么我仅在讨论动态数组之后才介绍开放数组参数。

开放数组参数的基本定义与类型化动态数组类型的定义相同，并以 const 说明符为前缀。这意味着您可以指示参数的类型，但无需指示数组将具有多少个该类型的元素。这是从 OpenArray 应用程序项目中提取的此类定义的示例：

```
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;
```

您可以通过传递一个整数数组常量表达式(也可以将变量作为用于计算各个值的表达式的一部分包括在内）来调用此函数：

```
X := Sum ([10, Y, 27*I]);
```

给定一个动态的 Integer 数组，您可以将其直接传递到需要一个具有相同基本类型的开放数组参数（在这种情况下为 Integers）的例程。这是一个示例，其中完整的数组作为参数传递：

```
var
  List: array of Integer;
  X, I: Integer;
begin
  // 初始化数组
  SetLength (List, 10);
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // 调用
  X := Sum (List);
```

这是如果您有一个动态数组。如果您具有匹配基本类型的静态数组，则还可以将其传递给需要开放数组参数的函数，或者可以调用 Slice（片段）函数仅传递现有数组的一部分（如第二个参数所示）。以下代码段（也是 OpenArray 应用程序项目的一部分）显示了如何将静态数组或静态数组的一部分传递给 Sum 函数：

```
var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // 初始化数组
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // 调用
  X := Sum (List);
  Show (X.ToString);
  // 通过数组的一部分
  X := Sum (Slice (List, 5));
  Show (X.ToString);
```

变体类型开放数组参数

除了这些类型化的开放数组参数外，Object Pascal 语言还允许您定义类型变

量或非类型化开放数组。这种特殊类型的数组具有不确定数量的元素，但是对于那些元素也具有不确定的数据类型，以及传递不同类型的元素的可能性。这是语言类型不完全安全的有限区域之一。

从技术上讲，您可以定义 const 类型数组的参数，以将具有未定义数量的不同类型元素的数组传递给函数。例如，这是 Format 函数的定义（我们将在第 6 章中介绍如何使用此函数，同时涵盖字符串，但我已经使用了它作为一些演示）：

```
function Format (const Format: string;
  const Args: array of const): string;
```

第二个参数是一个开放数组，它接收不确定数量的值。实际上，您可以通过以下方式调用此函数：

```
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，您可以将参数传递为常数，变量或表达式的值。声明这种函数很简单，但是如何编写它呢？您如何知道参数的类型？类型变量开放数组参数的值与 TVarRec 类型元素兼容。。

❖ 不要将 TVarRec 记录与 Variant 类型使用的 TVarData 记录混淆。这两种结构具有不同的目的并且不兼容。甚至可能的类型列表也不一样，因为 TVarRec 可以保存 Object Pascal 数据类型，而 TVarData 可以保存 Windows OLE 数据类型。变体将在本章后面介绍。

以下是类型变量开放数组值和 TVarRec 记录支持的数据类型：

vtInteger	vtBoolean	vtChar
vtExtended	vtString	vtPointer
vtPChar	vtObject	vtClass
vtWideChar	vtPWideChar	vtAnsiString
vtCurrency	vtVariant	vtInterface
vtWideString	vtInt64	vtUnicodeString

记录结构包含一个具有类型（VType）和变体字段的字段，您可以使用该字段来访问实际数据（有关这几页中的记录的更多信息，即使这是该构造的高级用法）。

一种典型的方法是使用 case 语句对可在此类调用中接收的不同类型的参数进行操作。在 SumAll 函数示例中，我希望能够求和不同类型的值，将字符串转换为整数，将字符转换为相应的序数值，并为 True 布尔值加 1。该代码肯定是相当高级的（并且使用了指针取消引用），所以如果您现在还不完全了解它，请不要担心：

```
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;
```

我已将此函数添加到 OpenArray 应用程序项目中，该项目的调用方式如下：

```
  var
    X: Extended;
    Y: Integer;
  begin
    Y := 10;
    X := SumAll ([Y * Y, 'k', True, 10.34]);
    Show ('SumAll: ' + X.ToString);
  end;
```

该调用的输出将 Y 的平方，K 的序数值（即 107），布尔值 1 和扩展数相加，结果为：

```
SumAll: 218.34
```

5.2 记录数据类型

数组定义由数字索引引用的相同项目的列表，而记录定义由名称引用的不同类型的元素组。换句话说，记录是命名项或字段的列表，每个项或字段都具有特定的数据类型。记录类型的定义列出了所有这些字段，并为每个字段指定了用于引用它的名称。

在 Pascal 初期，记录只包含字段，现在它们也可以包含方法和运算符，这将在本章中介绍。

❖ 记录可用大多数编程语言提供。 它们用 C 语言中的 struct 关键字定义，而 C++具有扩展的定义，其中包括方法，就像 Object Pascal 一样。一些更“纯”的面向对象语言仅具有类的概念，而没有记录或结构的概念，但是 C# 最近重新引入了该概念。

这是一个小代码片段（来自 RecordsDemo 应用程序项目），其中包含记录类型的定义，该类型的变量的声明以及使用此变量的一些语句：

```
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);
```

记录一词有时以一种相当宽松的方式使用，用以指代该语言的两个不同元素：记录类型定义和记录类型变量（或记录实例）。Record 被用作记录类型和记

录实例的同义词，与类类型不同，在这种情况下，实例称为对象。

正如本章的其余部分将说明的那样，Object Pascal 中的这种数据结构除了简单的字段列表之外还有更多的方法，但是让我们从这种传统的记录方法开始。记录的内存通常在堆栈中分配给局部变量，在全局存储器中分配给全局变量。这通过调用 SizeOf 突出显示，该调用返回变量或类型所需的字节数，如以下语句所示：

```
Show ('Record size is ' + SizeOf (BirthDay).ToString);
```

它返回 8（为什么不返回 8 而不是 6，所以 Integer 为 4 个字节，每个字节为两个字节，我将在“字段对齐”一节中讨论）。

换句话说，记录是值类型。这意味着，如果您将一条记录分配给另一个，则表示您正在制作完整副本。如果您更改副本，原始记录将不会受到影响。此代码段用代码术语解释了这一概念：

```
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（日语或国际日期格式）为：

```
1997.2.14
2008.2.14
```

当您将记录作为参数传递给函数时，会发生相同的复制操作，例如我在上面使用的 MyDateToString 中：

```
function MyDateToString (MyDate: TMyDate): string;
begin
  Result := MyDate.Year.ToString + '.' +
    MyDate.Month.ToString + '.' +
    MyDate.Day.ToString;
end;
```

每次对该函数的调用都涉及记录数据的完整副本。为了避免复制，并可能更改原始记录，您必须显式使用引用参数。以下过程突出显示了这一点，该过程对作为参数传递的记录进行了一些更改：

```
procedure IncreaseYear (var MyDate: TMyDate);
begin
  Inc (MyDate.Year);
end;
var
  ADay: TMyDate;
begin
  ADay.Year := 2020;
  ADay.Month := 3;
  ADay.Day := 18;
  Increaseyear (ADay);
  Show (MyDateToString (ADay));
```

通过过程调用增加了原始记录值的 Year 字段，最终输出比输入晚一年：

```
2021.3.18
```

5.2.1 使用记录数组

正如我提到的，数组代表重复多次的数据结构，而记录具有不同元素的单个结构。鉴于这两种类型的构造函数是正交的，将它们一起使用以定义记录的数组是很常见的（虽然可能但不常见的是看到数组的记录）。

数组代码与任何其他数组一样，每个数组元素占用特定记录类型的大小。虽然我们稍后将看到如何使用更复杂的集合或容器类（用于元素列表），但是可以通过记录数组来实现很多数据管理。

在 RecordsTest 应用程序项目中，我添加了 TMyDate 类型的数组，可以将其分配，初始化并与以下代码一起使用：

```
var
  DatesList: array of TMyDate;
  I: Integer;
begin
  //  分配数组元素
  SetLength (DatesList, 5);
  //分配随机值
  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;
  //  显示值
  for I := Low(DatesList) to High(DatesList) do
    Show (I.ToString + ': ' + MyDateToString (DatesList[I]));
```

假设该应用使用随机数据，则每次输出都会有所不同，就像我捕获的以下内容一样：

```
0: 2014.11.8
1: 2005.9.14
2: 2037.9.21
3: 2029.3.12
4: 2012.7.2
```

❖ 如果是托管记录，则可以自动初始化数组中的记录，这是 Delphi 10.4 Sydney 中引入的功能，我将在本章稍后介绍。

5.2.2 Variant（变体）记录

从该语言的早期版本开始，记录类型也可以包含变体部分。也就是说，即使它们具有不同的数据类型，也可以将多个字段映射到同一存储区。（这对应于 C 语言中的并集。）或者，您可以使用这些变体字段或字段组来访问记录中的相同存储位置，但是要从不同角度考虑这些值（就数据类型而言）。这种类型的主要用途是存储类似但不同的数据，并获得与类型转换（在语言的早期使用的东西，不允许直接类型转换）类似的效果。尽管某些系统库在特殊情况下在内部使用了变体记录类型，但是它们已被面向对象和其他现代技术所取代。

Variant（变体）记录类型的使用不是类型安全的，也不建议使用编程方法，特别是对于初学者。无论如何，直到您真正成为 Object Pascal 专家之前，您都不需要解决这些问题。这就是为什么我决定避免向您显示实际示例并更详细地介绍此功能。如果您真的想要一个提示，请查看我在“TypeVariant 开放数组参数”部

分的演示中使用的 TvarRec。

5.2.3 Fields Alignments（字段对齐）

与记录相关的另一个高级主题是字段的对齐方式，这也有助于了解记录的实际大小。如果查看库，通常会看到在记录中使用了 packed 关键字：这意味着记录应使用尽可能少的字节数，即使这样做会导致数据访问操作变慢。

传统上，差异与各个字段的 16 位或 32 位对齐有关，因此，即使仅使用 8 位，一个字节后跟一个整数也可能最终占用 32 位。这是因为在 32 位边界上访问以下整数值会使代码执行起来更快。

❖ 字段的大小和对齐方式取决于类型的大小。对于任何大小不是 2 的幂（或 2 ^ N）的类型，其大小都是下一个 2 的更高幂。因此，例如，扩展类型（使用 10 个字节）在一条记录中占用 16 个字节（除非该记录是 Packed（打包））。

通常，诸如记录之类的数据结构使用字段对齐来提高某些 CPU 体系结构对各个字段的访问速度。您可以将不同的参数应用于$ALIGN 编译器指令以对其进行更改。

使用{$ALIGN1}时，编译器将通过使用所有可能的字节来节省内存使用，例如当您使用打包的说明符记录时。在另一个极端，{$ALIGN16}将使用最大的对齐方式。其他选项使用 4 和 8 对齐方式。

举例来说，如果我回到 RecordsTest 项目并将关键字打包添加到记录定义：

```
type
  TMyDate = packed record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;
```

调用 SizeOf 的输出现在将返回 6 而不是 8。

作为更高级的示例，如果您还不是流利的 Object Pascal 开发人员，可以跳过此示例，让我们考虑以下结构（可在 AlignTest 应用程序项目中使用）：

```
type
  TMyRecord = record
    c: Byte;
    w: Word;
    b: Boolean;
    I: Integer;
    d: Double;
  end;
```

使用{$ ALIGN 1}时，该结构占用 16 个字节（由 SizeOf 返回的值），并且这些字段将位于以下相对内存地址中：

```
c: 0   w: 1    b: 3    i: 4    d: 8
```

❖ 相对地址是通过分配记录并计算指向结构的指针的数值与指向给定字段的指针的数值之间的差来计算的，表达式为：UIntPtr（@MyRec.w） - UintPtr（@MyRec1）。指针的概念和（@）运算符的地址将在本章后面介绍。

相反，如果将 alignment（对齐方式）更改为 4（这可以导致优化的数据访问），则大小将为 20 个字节，并且相对地址为：

```
c: 0   w: 2    b: 4    i: 8    d: 12
```

如果转到极端选项并使用{$ ALIGN 16}，则该结构需要 24 个字节，并将字段

映射如下：

c: 0　w: 2　　b: 4　　i: 8　　d: 16

5.2.4 那么 With 语句呢？

用于记录或类的另一种传统语言语句是 with 语句。该关键字以前是 Pascal 语法特有的，但后来在 JavaScript 和 VisualBasic 中引入。这是一个很容易编写较少代码的关键字，但是由于它使代码的可读性大大降低，因此也可能变得非常危险。

关于 with 语句，您会发现很多争论，而且我倾向于同意应该尽量少使用它。无论如何，我都认为无论如何都要将其包含在本书中（与 goto 语句不同）很重要。

❖ 关于从 Object Pascal 语言中删除 goto 语句是否有意义，存在一些争论，并且还讨论了是否从语言的移动版本中删除 goto 语句。尽管有一些合理的用法，但考虑到语句的范围界定问题可能会导致这种情况，但有充分的理由停止使用此功能（或对其进行更改，以便像 C# 中那样需要别名）。

with 语句不过是简写。当您需要引用记录类型变量（或对象）时，可以使用 with 语句来代替每次重复其名称。

例如，在介绍记录类型时，我编写了以下代码：

```
var
  BirthDay: TMyDate;
begin
  BirthDay.Year := 2008;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

使用 with 语句，我可以修改此代码的最后部分，如下所示：

```
  with BirthDay do
  begin
    Year := 2008;
    Month := 2;
    Day := 14;
  end;
```

可以在 Object Pascal 程序中使用此方法来引用组件和其他类。通常，当您使用组件或类时，with 语句使您可以跳过编写某些代码，特别是对于嵌套数据结构。

那么，为什么我不鼓励使用 with 语句呢？原因是它至少可以消除很难捕获的细微错误。尽管本书中有些难以理解的错误难以解释，但让我们考虑一个温和的情况，它仍然可能导致您挠头。这是一种记录类型以及一些使用它的代码：

```
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);
```

对？该应用程序会编译并运行，但是其输出不是您期望的（至少乍一看）：

```
Form1: 22
```

输出的字符串部分不是先前设置的记录值。原因是第二个 with 语句错误地使用了 Name 字段，它不是记录字段，而是另一个范围内的字段（特别是 Button2Click 方法所属于的表单对象的名称）。

如果您写过：

```
Show (Record1.Name + ': ' + Record1.MyValue.ToString);
```

编译器将显示一条错误消息，指示给定的记录结构没有名称字段。

通常，我们可以说，由于 with 语句在当前范围内引入了新的标识符，因此我们可能会隐藏现有的标识符，或者错误地访问同一范围内的另一个标识符。这是不鼓励使用 with 语句的一个很好的理由。 您甚至应该避免使用多个 with 语句，例如：

```
with MyRecord1, MyDate1 do...
```

紧随其后的代码可能极难读取，因为对于块中使用的每个字段，您都需要考虑其引用的记录。

5.3 带方法的记录

在对象中，Pascal 记录比原始 Pascal 语言或 C 语言中的结构更强大。实际上，记录可以具有与之关联的过程和功能（称为方法）。他们甚至可以自定义方式（一种称为运算符重载的功能）来重新定义语言运算符，这将在下一节中看到。

带有方法的记录在某种程度上类似于类，稍后我们将发现，最重要的区别是这两种结构管理内存的方式。

Object Pascal 中的记录具有现代编程语言的两个基本特征：

- **Methods**（方法），是与记录数据结构相关并可以直接访问记录字段的函数和过程。换句话说，方法是在记录类型定义中声明（或具有前向声明）的函数和过程。
- **Encapsulation**（封装），这是对其余代码隐藏对数据结构的某些字段（或方法）的直接访问的能力。您可以使用私有访问说明符获得封装，而外部可见的字段和方法被标记为 public。记录的默认说明符是 public。

现在，您已经拥有有关扩展记录的核心概念，让我们看一下来自 RecordMethods 应用程序项目的示例记录的定义：

```
type
  TMyRecord = record
  private
    Name: string;
    Value: Integer;
    SomeChar: Char;
  public
    procedure Print;
    procedure SetValue (NewString: string);
    procedure Init (NewValue: Integer);
  end;
```

您可以看到记录结构分为两个部分，私有和公共。您可以有多个部分，因为

可以根据需要将 private 和 public 关键字重复多次，但是将这两个部分明确划分无疑有助于提高可读性。这些方法在记录定义中列出（例如在类定义中），没有完整的代码。换句话说，它具有方法的前向声明。

您如何编写方法的实际代码及其完整定义？几乎以与您编写全局函数或过程相同的方式进行编码。区别在于编写方法名称的方式，该方法名称是记录类型名称和实际记录名称的组合，并且事实上，您可以直接引用记录的字段和其他方法，而无需写出记录的名字：

```
procedure TMyRecord.SetValue (NewString: string);
begin
  Name := NewString;
end;
```

✧ 虽然必须先编写方法的定义，然后再编写其完整的声明，这似乎很繁琐，但是您可以在 IDE 编辑器中使用 Ctrl + Shift + C 组合键自动从另一个生成一个。您也可以使用 Ctrl + Shift +上/下箭头键从方法声明移至相应的定义，反之亦然。

这是此记录类型的其他方法的代码：

```
procedure TMyRecord.Init(NewValue: Integer);
begin
  Value := NewValue;
  SomeChar := 'A';
end;

function TMyRecord.ToString: string;
begin
  Result := Name + ' [' + SomeChar + ']: ' + Value.ToString;
end;
```

以下是如何使用此记录的样本片段：

```
var
  MyRec: TMyRecord;
begin
  MyRec.Init(10);
  MyRec.SetValue ('hello');
  Show (MyRec.ToString);
```

您可能已经猜到了，输出将是：

```
hello [A]: 10
```

现在，如果您想使用使用记录的代码中的字段，如上面的代码片段所示：

```
  MyRec.Value := 20;
```

这实际上可以编译和工作，当我们在私有部分中声明该字段时，这可能令人惊讶，因此只有记录方法可以访问它。

事实是，在 Object Pascal 中，私有访问说明符实际上仅在不同的单元之间启用，因此该行在其他单元中不是合法的，但可以在最初定义数据类型的单元中使用。正如我们将看到的，对于类也是如此。

5.3.1 Self：记录背后的魔力

假设您有两个记录，例如相同记录类型的 myrec1 和 myrec2。

当您调用一个方法并执行其代码时，该方法如何知道它必须使用的记录的两个副本中的哪一个？在幕后，当您定义方法时，编译器会向该方法添加一个隐藏参数，即对已应用该方法的记录的引用。

换句话说，对上述方法的调用由编译器以类似以下方式转换：

```
// 你这样写
MyRec.SetValue ('hello');
// 编译器生成
SetValue (@MyRec, 'hello');
```

在此伪代码中，@是运算符的地址，用于获取记录实例的内存位置。

❖ 同样，本章末尾（高级）标题为“指针的作用是什么”的内容很快涵盖了运算符的地址。

这就是调用代码的翻译方式，但是实际的方法调用如何引用并使用此隐藏参数？通过隐式使用称为 self 的特殊关键字。

因此，该方法的代码可以写为：

```
procedure TMyRecord.SetValue (NewString: string);
begin
  self.Name := NewString;
end;
```

编译此代码时，除非明确引用整个记录，例如将记录作为参数传递给另一个函数，否则显式使用 self 几乎没有意义。对于类来说，这种情况会更频繁地发生，这些类对于方法具有完全相同的隐藏参数，并且具有相同的 self 关键字。

使用显式的 self 参数可以使代码更具可读性（即使不是必需的）的一种情况是，当您处理相同类型的第二个数据结构时，例如在测试来自另一个实例的值时：

```
function TMyRecord.IsSameName (ARecord: TMyRecord): Boolean;
begin
  Result := (self.Name = ARecord.Name);
end;
```

❖ “隐藏”的 self 参数在 C ++和 Java 中称为 this，但在 Objective-C（当然还有 Object Pascal）中称为 self。

5.3.2 初始化记录

当您将记录类型的变量（或记录实例）定义为全局变量时，其字段将被初始化，但在堆栈上定义一个变量时（作为函数或过程的局部变量，则不会）。因此，如果您编写这样的代码（也是 RecordMethods 项目的一部分）：

```
var
  MyRec: TMyRecord;
begin
  Show (MyRec.ToString);
```

它的输出或多或少是随机的。当字符串初始化为空字符串时，字符字段和整数字段将具有恰好位于给定存储位置的数据（就像堆栈中的字符或整数变量通常发生的情况一样）。通常，根据实际的编译或执行，您将获得不同的输出，例如：

```
[]: 1637580
```

这就是为什么在使用记录之前初始化记录（与大多数其他变量一样）很重要，以避免读取不合逻辑的数据的风险，甚至可能导致应用程序崩溃。

有两种截然不同的方法来处理这种情况。首先是使用构造函数进行记录，如下一节所述。第二个是托管记录的使用，这是 Delphi10.4 中的一项新功能，我将在本章稍后介绍。

5.3.3 记录和 Constructors（构造函数）

让我们从常规构造函数开始。记录支持一种特殊的方法类型，称为构造函数，可用于初始化记录数据。与其他方法不同，构造函数还可以应用于记录类型以定义新实例（但仍可以将其应用于现有实例）。

这是向记录添加构造函数的方法：

```
type
  TMyNewRecord = record
  private
    ...
  public
    constructor Create (NewString: string);
    function ToString: string;
    ...
```

构造函数是带有代码的方法：

```
constructor TMyNewRecord.Create (NewString: string);
begin
  Name := NewString;
  Init (0);
end;
```

现在，您可以使用以下两种编码方式之一来初始化记录：

```
var
  MyRec, MyRec2: TMyNewRecord;
begin
  MyRec := TMyNewRecord.Create ('Myself'); // 类似于类的调用
  MyRec2.Create ('Myself'); // 直接调用
```

请注意，记录构造函数必须具有参数：如果尝试使用 Create()，则会收到错误消息“记录类型不允许使用无参数构造函数”。

❖ 根据文档，用于记录的无参数构造函数的定义是为系统保留的（它具有初始化某些记录字段（例如字符串和接口）的方式）。这就是为什么任何用户定义的构造函数都必须至少具有一个参数的原因。当然，您也可以具有多个重载的构造函数或具有不同名称的多个构造函数。在讨论类的构造函数时，我将更详细地介绍这一点。我们将很快看到，托管记录使用不同的语法，并且没有引入无参数构造函数，而是引入了 Initialize 类方法。

5.3.4 Operators（操作符）Gain（增添）New Ground（新天地）

与记录相关的另一个 Object Pascal 语言功能是运算符重载。也就是说，您可以针对数据类型的标准操作（加，乘，比较等）定义自己的实现。这个想法是，您可以实现一个 add 运算符（一种特殊的 Add 方法），然后使用+号对其进行调用。要定义运算符，请使用类运算符关键字组合。

❖ 通过重新使用现有的保留字，语言设计人员设法对现有代码没有影响。他们最近经常在关键字组合（例如严格的 private，类运算符和 class var）中完成此操作。

这里的“class”一词与类方法有关，我们将在以后的第 12 章中探讨这个概念。 在指令之后，您可以输入 operator 操作的名称，例如“Add”：

```
type
  TPointRecord = record
  public
    class operator Add (A, B: TPointRecord): TPointRecord;
```

如您所愿，然后用+符号调用操作符 Add：

```
var
```

```
    A, B, B: TPointRecord;
begin
    ...
    B := A + B;
```

那么哪些是可用的运算符？基本上是语言的整个运算符集，因为您无法定义全新的语言运算符：

- **强制转换运算符**：Implicit（隐式）和 Explicit（显式）
- **一元运算符**：Positive（正）， Negative（负），Inc，Dec，LogicalNot，BitwiseNot，Trunc 和 Round
- **比较运算符**：Equal，NotEqual，GreaterThan，GraterThanOrEqual，LessThan 和 LessThenOrEqual
- **二进制运算符**：Add（加），Subtract（减），Multiply（乘），Divide（除），IntDivide（整数除法），Modulus（模量），ShiftLeft，ShiftRight，LogicalAnd，LogicalOr，LogicalXor，BitwiseAnd，BitwiseOr 和 BitwiseXor。
- **托管记录运算符**：Initialize（初始化），Finalize（完成），Assign（分配）【有关在 Delphi 10.4 中添加的这 3 个运算符的特定信息，请参见下面的“运算符和自定义托管记录”部分。】

在调用运算符的代码中，您无需使用这些名称，而是使用相应的符号。您只能在定义中使用这些特殊名称，并使用类运算符前缀以避免任何命名冲突。例如，您可以使用 Add 方法创建一条记录，并向其添加一个 Add 运算符。

定义这些运算符时，您将拼出参数，并且仅在参数完全匹配时才应用运算符。要添加两个不同类型的值，您必须指定两个不同的“添加”操作，因为每个操作数都可以是表达式的第一项或第二项。实际上，运算符的定义不提供自动可交换性。此外，您必须非常准确地指明类型，因为自动类型转换不适用。很多时候，这意味着使用不同类型的参数为操作员定义多个重载版本。

要注意的另一个重要因素是，您可以为数据转换定义两个特殊的运算符：Implicit 和 Explicit。第一个用于定义隐式类型转换（或静默转换），该类型应该是完美的，并且没有损耗。第二种，显式，只能使用从类型变量到另一个给定类型的显式类型调用。这两个运算符共同定义了允许与给定数据类型之间的强制类型转换。

注意，隐式和显式运算符都可以基于函数返回类型进行重载，这对于重载方法通常是不可能的。

实际上，在类型转换的情况下，编译器知道预期的结果类型，并且可以找出要应用的类型转换操作。作为示例，我编写了 OperatorsOver 应用程序项目，该项目定义了一个包含几个运算符的记录：

```
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;
```

这是记录方法的实现：

```
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;
```

使用这样的记录非常简单，因为您可以编写如下代码：

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    A, B, C: TPointRecord;
  begin
    A.SetValue(10, 10);
    B := 30;
    C := A + B;
    Show (string(C));
  end;
```

由于缺少转换，第二次分配（B := 30;）使用隐式运算符完成，而 Show 调用使用转换符号来激活显式类型转换。还请考虑运算符 Add 不会修改其参数；而是返回了全新的值。

❖ 运算符返回新值的事实使我们更难想到类的运算符重载。如果 operator 创建了一个新的临时对象，谁来处置它？

Operators Overloading（重载）Behind the Scenes（幕后）

这是一个相当高级的简短部分，您可能希望在初读时略过。

一个鲜为人知的事实是，从技术上讲，可以使用其完全限定的内部名称（例如&& op_Addition）调用操作员，并在其前面加上双精度＆，而不是使用操作符号。例如，您可以按以下方式重写记录总和（有关完整列表，请参见演示）：

```
  C := TPointRecord.&&op_Addition(A, B);
```

尽管我看不到您可能要这样做的少数情况。（定义运算符的全部目的是能够使用比方法名称更友好的表示法，而不是在前面的直接调用中使用较丑陋的符号。）

Implementing Commutativity（实现交换性）

假设您想要实现将整数添加到其中一条记录的功能。您可以定义以下运算符（对于稍有不同的记录类型，可以在 OperatorsOver 应用程序项目的代码中找到）：

```
  class operator TPointRecord2.Add (A: TPointRecord2;
    B: Integer): TPointRecord2;
  begin
    Result.X := A.X + B;
    Result.Y := A.Y + B;
  end;
```

❖ 之所以在新类型上而不是现有类型上定义此运算符，是因为相同的结构已经

定义了将整数隐式转换为记录类型，因此我可以在不定义特定运算符的情况下添加整数和记录。下一节将对此问题进行更好的解释。

现在，您可以合法地将浮点值添加到记录中：

```
var
  A: TPointRecord2;
begin
  A.SetValue(10, 20);
  A := A + 10;
```

但是，如果尝试编写相反的加法：

```
  A := 30 + A;
```

这将失败并显示以下错误：

[dcc32 Error] E2015 Operator not applicable to this operand type

（[dcc32 错误] E2015 运算符不适用于此操作数类型）

实际上，正如我所提到的，对应用到不同类型变量的运算符来说，可交换性不是自动的，但是必须重复执行调用或调用另一种运算符（如下所示）来具体实现：

```
class operator TPointRecord2.Add(B: Integer;
  A: TPointRecord2): TPointRecord2;
begin
  Result := A + B; //  实现可交换性
end;
```

Implicit Cast and Type Promotions（隐式类型转换）

重要的是要注意，与涉及运营商的呼叫解决相关的规则不同于涉及方法的传统规则。借助自动类型提升，单个表达式有可能最终调用不同版本的重载运算符并导致模棱两可的调用。这就是为什么在编写隐式运算符时需要多加注意的原因。

考虑上一个示例中的这些表达式：

```
A := 50;
C := A + 30;
C := 50 + 30;
C := 50 + TPointRecord(30);
```

他们都是合法的！在第一种情况下，编译器将 30 转换为正确的记录类型，在第二种情况下，转换在赋值之后进行，在第三种情况下，显式强制将隐式变量强制转换为第一个值，因此要执行的加法运算是在记录中自定义一个。换句话说，第二个操作的结果与其他两个操作的结果不同，如以下语句的输出和扩展版本中突出显示的那样：

```
// output（输出）
(80:20)
(80:10)
(80:20)
// expanded statements（扩展陈述）
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)
```

5.3.5 Operators 和自定义托管 Record（记录）

您可以使用一组特殊的运算符来用 Delphi 语言记录，以定义自定义的托管记录。在到达那里之前，让我回顾一下记录内存初始化的规则以及纯记录和托管记录之间的区别。

Delphi 中的记录可以具有任何数据类型的字段。当记录具有纯（非托管）字段时，像数字值或其他枚举值一样，编译器没有太多工作要做。创建和处理记录包括分配内存或摆脱内存位置。（请注意，默认情况下，Delphi 不会对记录进行零初始化。）

如果记录具有由编译器托管的类型的字段（例如字符串或接口），则编译器需要注入额外的代码来管理初始化或终结处理。例如，对字符串进行引用计数，因此当记录超出范围时，记录内的字符串需要减少其引用计数，这可能导致为该字符串分配内存。因此，当您在代码的一部分中使用这种托管记录时，编译器会自动在该代码周围添加一个 try-finally 块，并确保即使出现异常也清除了数据。长期以来就是这种情况。换句话说，托管记录已成为 Delphi 语言的一部分。

现在，从 10.4 开始，Delphi 记录类型支持自定义初始化和终结处理，这超出了编译器对托管记录所做的默认操作。

您可以使用自定义初始化和完成代码声明记录，而不管其字段的数据类型如何，并且可以编写此类自定义 initialization（初始化）和 finalization（终止）代码。这些记录称为“自定义托管记录”。

通过将特定的新运算符之一添加到记录类型中，或多个，可以使开发人员将记录变成定制的托管记录：

- 在分配了用于记录的内存之后，调用了 Operator Initialize，使您可以编写代码来为字段设置初始值
- 在取消分配记录的内存之前调用 Operator Finalize，并允许您执行任何必需的清除操作
- 将记录数据复制到相同类型的另一记录时，将调用“操作员分配”，因此您可以通过特定的自定义方式将信息从一个记录复制到另一个记录

❖ 假定即使在发生异常的情况下也执行托管记录的终结处理（编译器会自动生成 try-finally 块），所以它们通常用作保护资源分配或实现清除操作的替代方法。我们将在第 9 章的“使用托管记录还原游标”部分中看到此用法的示例。

带有初始化和终结运算符的记录

让我们从初始化和终结开始。 以下是一个简单的代码段：

```
type
  TMyRecord = record
    Value: Integer;
    class operator Initialize (out Dest: TMyRecord);
    class operator Finalize(var Dest: TMyRecord);
  end;
```

当然，您需要为这两个类方法编写代码，例如记录它们的执行或初始化记录值。在此示例（ManagedRecords_101 演示项目的一部分）中，我还记录了对内存位置的引用，以查看哪个记录正在执行每个单独的操作：

```
class operator TMyRecord.Initialize (out Dest: TMyRecord);
begin
```

```
    Dest.Value := 10;
    Log('created' + IntToHex (Integer(Pointer(@Dest))));
end;

class operator TMyRecord.Finalize(var Dest: TMyRecord);
begin
    Log('destroyed' + IntToHex (Integer(Pointer(@Dest))));
end;
```

这种构造机制与以前可用于记录的机制之间的区别是自动调用。如果您编写类似下面的代码，则可以调用初始化代码和终结代码，并以编译器为您的托管记录实例生成的 try-finally 块结尾：

```
procedure LocalVarTest;
var
    My1: TMyRecord;
begin
    Log (My1.Value.ToString);
end;
```

使用此代码，您将获得类似以下的日志：

```
created 0019F2A8 10
destroyed 0019F2A8
```

另一种情况是使用内联变量，例如：

```
begin
var T: TMyRecord;
    Log(T.Value.ToString);
```

通常，:=分配会平坦地复制记录字段的所有数据。对于托管记录（具有字符串和其他托管类型），编译器将对其进行适当处理。

当您具有自定义数据字段和自定义初始化时，您可能需要更改默认行为。这就是为什么对于自定义托管记录，您还可以定义一个赋值运算符的原因。使用：=语法调用 new 运算符，但定义为 Assign：MarcoCantù，Object Pascal 手册 10.4

```
class operator Assign (var Dest: TMyRecord;
    const [ref] Src: TMyRecord);
```

运算符定义必须遵循非常精确的规则，包括将第一个参数作为由引用传递的参数（var），将第二个参数作为由引用传递的 const 参数。如果不这样做，则编译器将发出如下错误消息：

[dcc32 Error] E2617 First parameter of Assign operator must be a var parameter of the container type

（[dcc32 错误] E2617 Assign 运算符的第一个参数必须是容器类型的 var 参数。）

[dcc32 Hint] H2618 Second parameter of Assign operator must be a const [Ref] or var parameter of the container type

（[dcc32 提示] H2618 Assign 运算符的第二个参数必须是容器类型的 const [Ref]或 var 参数。）

有一个示例案例，调用了 Assign 运算符

```
var
    My1, My2: TMyRecord;
begin
    My1.Value := 22;
    My2 := My1;
```

这将生成此日志（在该日志中，我还向记录添加了序列号）：

```
created 5 0019F2A0
created 6 0019F298
5 copied to 6
destroyed 6 0019F298
destroyed 50019F2A0
```

请注意，销毁顺序与构造顺序相反，创建的最后一条记录是第一个销毁的记

录。

将托管记录作为参数传递

当作为参数传递或由函数返回时，托管记录也可以不同于常规记录。以下是显示各种情况的例程：

```
procedure ParByValue (Rec: TMyRecord);
procedure ParByConstValue (const Rec: TMyRecord);
procedure ParByRef (var Rec: TMyRecord);
procedure ParByConstRef (const [ref] Rec: TMyRecord);
function ParReturned: TMyRecord;
```

现在，无需一一遍历每个日志（您可以通过运行 ManagedRecords_101 演示查看它们），这是该信息的摘要：

- ParByValue 创建一个新记录，并调用赋值运算符（如果可用）以复制数据，并在退出过程时销毁临时副本
- ParByConstValue 不进行任何复制，也不进行任何调用
- ParByRef 不复制，不调用
- ParByConstRef 不复制，不调用
- ParReturned 创建一个新记录（通过 Initialize），并在返回时调用 Assign 运算符（如果调用类似于 my1：= ParReturned），并在分配后删除临时记录。

Exceptions（例外）和托管记录 Managed Records

当引发异常时，与对象不同的是，即使没有显式的 try，finally 块，通常也会清除记录。这是根本的区别，也是托管记录真正有用的关键。

```
procedure ExceptionTest;
begin
  var A: TMRE;
  var B: TMRE;
  raise Exception.Create('Error Message');
end;
```

在此过程中，有两个构造函数调用和两个析构函数调用。同样，这是托管记录的根本区别和关键特征。

托管记录数组

如果定义托管记录的静态数组，则会在点声明处调用 Initialize 运算符来初始化它们：

```
var
  A1: array [1..5] of TMyRecord; // call here
begin
  Log ('ArrOfRec');
```

他们超出范围时都被摧毁。如果定义托管记录的动态数组，则调用初始化代码，并调整数组的大小（使用 SetLength）：

```
var
  A2: array of TMyRecord;
begin
  Log ('ArrOfDyn');
  SetLength(A2, 5); // call here
```

5.4 Variants（变体）

Object Pascal 最初以提供全面的 Windows OLE 和 COM 支持的语言引入，具有一种称为 Variant 的松散类型本机数据类型的概念。尽管该名称使人想起了变种记录（如前所述），并且实现与开放数组参数有些相似，但这是具有非常特定的实现的单独功能（在 Windows 开发领域以外的语言中很少见）。

在本节中，我不会真正涉及 OLE 以及使用此数据类型的其他情况（例如数据集的字段访问），我只想从一般角度讨论此数据类型。

在第 16 章中，我将返回动态类型，RTTI 和反射，在此还将介绍一种相关的（但类型安全且速度更快）RTL 数据类型，称为 TValue。

5.4.1 变体没有类型

通常，您可以使用变量类型的变量来存储任何基本数据类型，并执行许多操作和类型转换。自动类型转换违反了 Object Pascal 语言的一般类型安全方法，并且是一种动态类型的实现，该类型最初由 Smalltalk 和 Objective-C 等语言引入，最近在包括 JavaScript，PHP，Python 的脚本语言中流行 和 Ruby。

在运行时对变量进行类型检查和计算。编译器不会警告您代码中可能存在的错误，只有通过大量测试才能发现这些错误。总体而言，您可以将使用变体的代码部分视为解释代码，因为与解释代码一样，许多操作要等到运行时才能解决。特别是这会影响代码的速度。

现在，我已警告您不要使用 Variant 类型，现在该看看可以使用它做什么。基本上，一旦声明了以下变量，就可以：

```
var
  V: Variant;
```

您可以为其分配几种不同类型的值：

```
V := 10;
V := 'Hello, World';
V := 45.55;
```

获得变量值后，可以将其复制到任何兼容或不兼容的数据类型。 如果为不兼容的数据类型分配值，则编译器通常不会将其标记为错误，但会在有意义的情况下执行运行时转换。否则，它将发出运行时错误。从技术上讲，变体将类型信息与实际数据一起存储，从而允许进行许多方便但缓慢且不安全的运行时操作。

考虑以下代码（属于 VariantTest 应用程序项目的一部分），它是上述代码的扩展：

```
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);
```

好笑，不是吗？ 这是输出（不足为奇）：

```
20
```

Hello, World10
55.55

除了为 S 变量分配一个包含字符串的变量之外，您还可以为其分配一个包含整数或浮点数的变量。更糟糕的是，您可以使用变量 V：=V+S;来计算变量。会根据变体中存储的数据以不同的方式进行解释。在上面的代码中，同一行可以添加整数，浮点值或连接字符串。

至少可以说，编写涉及变体的表达式是有风险的。如果字符串包含数字，则一切正常。如果不是，则引发异常。没有令人信服的理由，您不应该使用 Variant 类型。坚持使用标准的 Object Pascal 数据类型和类型检查方法。

5.4.2 深度的变体

对于那些想更详细地了解变体的人，让我添加一些有关变体如何工作以及如何对它们进行更多控制的技术信息。RTL 包含一个变体记录类型 TVarData，它与变体类型具有相同的内存布局。您可以使用它来访问变体的实际类型。TVarData 结构包括 Variant 的类型（表示为 VType），一些保留字段和实际值。

❖ 有关更多详细信息，请在“Syetem（系统）”单元中的 RTL 源代码中查看 TVarData 定义。这远不是一个简单的结构，我建议只有具有一定经验的开发人员才能研究变体类型的实现细节。

VType 字段的可能值与您可以在 OLE 自动化中使用的数据类型相对应，这些数据类型通常称为 OLE 类型或变量类型。以下是可用变体类型的完整字母列表：

varAny	varArray	varBoolean
varByte	varByRef	varCurrency
varDate	varDispatch	varDouble
varEmpty	varError	varInt64
varInteger	varLongWord	varNull
varOleStr	varRecord	varShortInt
varSingle	varSmallint	varString
varTypeMask	varUInt64	varUnknown
varUString	varVariant	varWord

这些变体类型的大多数常量名称很容易理解。请注意，存在空值的概念，您可以通过分配 NULL（而不是 nil）来获得。

还有许多用于操作变体的函数，您可以使用这些函数进行特定的类型转换或询问有关变体类型的信息（例如，参见 VarType 函数）。当您使用变体编写表达式时，实际上实际上会自动调用大多数这些类型转换和赋值函数。其他变体支持例程实际上在变体数组上运行，这又是一种几乎专门用于 Windows 上的 OLE 集成的结构。

5.4.3 Variants Are Slow（变体变慢）

使用 Variant 类型的代码很慢，不仅在转换数据类型时，甚至在您简单地添加两个包含整数的变量值时也是如此。它们几乎和解释的代码一样慢。要将基于变体的算法的速度与基于整数的相同代码的速度进行比较，可以查看 VariantTest 项目的第二个按钮。

该程序运行一个循环，定时其速度并在进度栏中显示状态。

这是基于 Int64 和变体的两个非常相似的循环中的第一个：

```
const
  maxno = 10000000; // 1 千万
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;
  //  我们必须使用结果
  time2 := Now;
  Show (n2);
  Show ('Variants: ' + FormatDateTime ('ss.zzz', Time2-Time1) + ' seconds');
```

时序代码值得一看，因为它很容易适应任何类型的性能测试。如您所见，程序使用 Now 函数获取当前时间，并使用 FormatDateTime 函数输出时差，仅显示秒（“ss”）和毫秒（“zzz”）。

在此示例中，速度差实际上是如此之大，即使没有精确的时间，您也会注意到它：

```
49999995000000
Variants: 01.169 seconds
49999995000000
Integers: 00.026 second
```

这些是我在 Windows 虚拟机上获得的数字，对于基于变体的代码，这大约慢了 50 倍。实际值取决于用于运行该程序的计算机，但是相对差异不会有太大变化。即使在我的 Android 手机上，我得到的比例也差不多（但总体上要长得多）：

```
49999995000000
Variants: 07.717 seconds
49999995000000
Integers: 00.157 second
```

在我的手机上，此代码花费的时间是 Windows 的 6 倍，但事实是网络的差异超过 7 秒，这使基于变体的实现对用户而言明显变慢，而基于 Int64 的实现仍然非常快（用户 几乎不会注意到十分之一秒）。

5.5 指针呢？

Object Pascal 语言的另一种基本数据类型由指针表示。一些面向对象的语言在隐藏这种强大而危险的语言构造方面已经走了很长一段路，而 Object Pascal 则允许程序员在需要时使用它（通常并不经常使用）。

但是什么是指针，指针的名称从何而来？与大多数其他数据类型不同，指针不保存实际值，但是它保存对变量的间接引用，该变量又具有值。表示这种情况的一种更技术性的方法是，指针类型定义一个变量，该变量保存给定数据类型（或未定义类型）的另一个变量的内存地址。

❖ 这是本书的高级部分，在这里添加是因为指针是 Object Pascal 语言的一部分，

并且应该是任何开发人员的核心知识的一部分，尽管它不是基本主题，并且如果您不熟悉该语言，则可能需要第一次阅读本书时跳过此部分。同样，您可能会使用没有（显式）指针的编程语言，因此这一小节可能是一个有趣的阅读！

指针类型的定义不是基于特定的关键字，而是使用特殊的符号，即插入符号（^）。 例如，您可以使用以下声明定义一个表示 Integer 类型变量的指针的类型：

```
type
  TPointerToInt = ^Integer;
```

定义了指针变量后，可以使用@运算符为其分配另一个相同类型的变量的地址：

```
var
  P: ^Integer;
  X: Integer;
begin
  X := 10;
  P := @X;
  // 使用指针更改 X 的值
  P^ := 20;
  Show ('X: ' + X.ToString);
  Show ('P^: ' + P^.ToString);
  Show ('P: ' + UIntPtr(P).ToHexString (8));
```

此代码是 PointersTest 应用程序项目的一部分。给定指针 P 引用变量 X，您可以使用 P^引用变量的值，然后读取或更改它。您也可以通过使用特殊类型的 UIntPtr 将指针强制转换为数字来显示指针本身的值，即 X 的内存地址（有关更多信息，请参见下面的注释）。与其显示纯数字值，该代码显示了十六进制表示形式，这在内存地址中更为常见。这是输出（其中指针地址可能取决于特定的编译）：

```
X: 20
P^: 20
P: 0018FC18
```

➢ 限于 2GB 时，仅在 32 位平台上，将指针强制转换为 Integer 是正确的代码。如果启用更大的内存空间，则必须使用 Cardinal 类型。对于 64 位平台，更好的选择是使用 NativeUInt。但是，有一个这种类型的别名，专门用于指针，称为 UIntPtr，它是此方案的最佳选择，因为使用它时，您可以清楚地向 Delphi 编译器指示意图。

为了清楚起见，让我总结一下。当您有一个指针 P 时：

- 通过直接使用指针（使用表达式 P），可以引用指针所指向的存储位置的地址。
- 通过取消引用指针（使用表达式 P ^），可以引用该内存位置的实际内容。

指针也可以引用通过 New 过程动态分配在堆上的新的特定内存块，而不是引用现有的内存位置。在这种情况下，当您不再需要指针访问值时，还必须通过调用 Dispose 过程来摆脱动态分配的内存。

❖ 第 13 章介绍了一般的内存管理，尤其是堆的工作方式。简而言之，堆是一个（大）内存区域，您可以在其中按任意给定顺序分配和释放内存块。作为 New 和 Dispose 的替代方法，可以使用 GetMem 和 FreeMem，它们要求开发人员提供分配的大小（而对于 New 和 Dispose，编译器会自动确定分配大小）。

如果在编译时不知道分配大小，则 GetMem 和 FreeMem 变得很方便。

这是一个动态分配内存的代码片段：

```
var
  P: ^Integer;
begin
  // 初始化
  New (P);
  // 运作
  P^ := 20;
  Show (P^.ToString);
  // 终止
  Dispose (P);
```

如果使用完内存后仍不处理内存，则程序最终可能会耗尽所有可用内存并崩溃。释放不再需要的内存失败称为内存泄漏。

➢ 为了更安全，上面的代码确实应该使用异常处理 try-finally 块，这是我决定在本书中不介绍的主题，但我将在第 9 章中介绍。

如果指针没有值，则可以为其分配 nil 值。您可以通过直接相等测试或使用特定的 Assigned 函数来测试指针是否为 nil，以查看其当前是否指向值，如下所示。

经常使用这种测试，因为取消引用（即访问指针中存储的内存地址上的值）无效的指针会导致内存访问冲突（根据操作系统的不同，其效果会略有不同）：

```
var
  P: ^Integer;
begin
  P := nil;
  Show (P^.ToString);
```

您可以通过运行 PointersTest 应用程序项目来查看此代码的效果示例。您将看到的错误（在 Windows 上）应类似于：

Access violation at address 0080B14E in module 'PointersTest.exe'. Read of address 00000000.（模块'PointersTest.exe'中地址 0080B14E 的访问冲突。读取地址 00000000）

使指针数据访问更安全的方法之一是添加“ pointer is not null”安全检查，如下所示：

```
if P <> nil then
  Show (P^.ToString);
```

如前所述，出于可读性考虑，通常首选的另一种方法是使用 Assigned 伪函数：

```
if Assigned (P) then
  writeln (P^.ToString);
```

❖ Assigned 不是真正的函数，因为编译器会生成适当的代码来“解析”该函数。同样，它可以在过程类型变量（或方法引用）上使用，而无需实际调用它，而仅检查它是否已分配。

Object Pascal 还定义了一个 Pointer 数据类型，该类型指示未类型化的指针（例如 C 语言中的 void*）。如果使用无类型的指针，则应使用 GetMem 而不是 New（指示要分配的字节数，因为不能从类型本身推断出此值）。每当未定义要分配的内存变量的大小时，都需要 GetMem 过程。

在 Object Pascal 中很少需要指针的事实是该语言的一个有趣的优势。尽管如此，拥有此功能仍然可以帮助实现一些非常有效的低级功能，以及在调用操作系统的 API 时。

无论如何，理解指针对于高级编程和全面理解语言对象模型都很重要，后者

在幕后使用了指针（通常称为引用）。

- 当变量持有指向第二个变量的指针并且该第二个变量超出范围或被释放（如果是动态分配的）时，该指针将仅指向未定义或保存一些其他数据的内存位置。这会导致很难发现错误。

5.6 File Types, Anyone（文件类型，有人吗）？

本章最后（简要介绍）的最后一个 Object Pascal 数据类型构造函数是文件类型。文件类型代表物理磁盘文件，当然是原始 Pascal 语言的特性，因为很少有旧的或现代的编程语言将文件的概念视为原始数据类型。Object Pascal 语言有一个 file 关键字，它是类型说明符，例如数组或记录。您可以使用 file 来定义新类型，然后可以使用新数据类型来声明新变量：

```
type
  IntFile = file of Integers;
var
  IntFile1: IntFile;
```

也可以在不指示数据类型的情况下使用 file 关键字来指定未类型化的文件。或者，您可以使用在运行时库的“系统”单元中定义的 TextFile 类型来声明 ASCII 字符文件（或更准确地说，在这些时间中，是字节文件）。

尽管仍受支持，但直接使用文件的日子越来越少，因为运行时库包含许多用于在更高级别上管理二进制和文本文件的类（例如，包括对文本文件的 Unicode 编码的支持）。

Delphi 应用程序通常使用 RTL 流（TStream 和派生类）来处理任何复杂的文件读写操作。流表示虚拟文件，这些虚拟文件可以映射到物理文件，内存块，套接字或任何其他连续的字节序列。

当您仍然看到一些正在使用的旧文件管理例程时，一个地方就是编写控制台应用程序，您可以在其中使用 write，writeln，read 和相关功能来处理特殊文件，这是标准输入和标准。输出（C 和 C++对控制台的输入和输出具有类似的支持，许多其他语言也提供类似的服务）。

第六章 关于 Strings 的全部

字符串是任何编程语言中最常用的数据类型之一。Object Pascal 使字符串处理相当简单，但非常快速且功能强大。即使字符串的基础知识很容易掌握，并且在前几章中我都使用字符串进行输出，但在幕后，情况比乍一看看起来要复杂得多。文本操作涉及几个值得探讨的密切相关的主题：要完全了解字符串处理，您需要了解 Unicode 表示形式，了解字符串如何映射到字符数组，并了解运行时库中一些最相关的字符串操作，包括保存操作。字符串到文本文件并加载它们。

Object Pascal 具有多个用于字符串操作的选项，并提供了不同的数据类型和方法。本章的重点将放在标准的字符串数据类型上，但是我还将花一些时间来研究仍然可以在 Delphi 编译器中使用的较旧的字符串类型，例如 AnsiString。 在开始之前，让我从头开始：Unicode 表示。示。

6.1 Unicode：整个世界的字母

Object Pascal 字符串管理以 Unicode 字符集为中心，尤其是使用其表示形式之一称为 UTF-16。在我们了解实现的技术细节之前，值得花一些时间来完全理解 Unicode 标准。

Unicode（同时使它变得简单和复杂）的背后思想是，世界上所有已知字母中的每个字符都有其自己的描述，图形表示和唯一的数值（称为 Unicode 代码点）。

Unicode 联盟的参考网站是 http://www.unicode.org，其中包含大量文档。 最终参考是“ Unicode 标准 ”这本书，可以在 http://www.unicode.org/book/aboutbook.html 上找到。

并非所有开发人员都熟悉 Unicode，并且许多开发人员仍会以较旧的有限表示形式（例如 ASCII）和 ISO 编码来考虑字符。通过简短介绍这些较旧的标准，您最好了解 Unicode 的特殊性（和复杂性）。

6.1.1 过去的字符：从 ASCII 到 ISO 编码

字符表示法始于 60 年代初的美国信息交换标准代码（ASCII），它是计算机字符的标准编码，包括 26 个英文字母（小写和大写），10 个数字， 常见的标点符号和许多控制字符（今天仍在使用）。

ASCII 使用 7 位编码系统表示 128 个不同的字符。如图 6.1 所示（从 Windows 上运行的 Object Pascal 应用程序中提取），只有＃32（空格）和＃126（波浪号）之间的字符才具有视觉表示。

	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	{	|	}	~	▯

图 6.1：带有可打印 ASCII 字符集的表

虽然 ASCII 无疑是一个基础（其 128 个字符的基本集合仍然是 Unicode 核心的一部分），但它很快被扩展版本所取代，后者使用第 8 位在该集合中再添加 128 个字符。

现在的问题是，世界上有这么多语言，没有一种简单的方法来找出集合中还包括哪些其他字符（有时以 ASCII-8 表示）。为了使故事简短，Windows 根据您的语言环境配置和 Windows 版本采用了不同的字符集（称为代码页），其中包含一组字符。除了 Windows 代码页外，还有许多其他基于类似分页方法的标准，这些页已成为国际 ISO 标准的一部分。

当然，最相关的是 ISO8859 标准，该标准定义了多个区域集。最常用的设置（当然，在大多数西方国家使用的设置更为精确）是拉丁文设置，称为 ISO8859-1。

即使部分相似，Windows1252 代码页也不完全符合 ISO8859-1 集。

❖ Windows 在 128 到 150 的区域中添加了额外的字符，例如€符号，额外的引号等。与拉丁语集的所有其他值不同，这些 Windows 扩展名不符合 Unicode 代码点。

6.1.2 Unicode 代码点和字素

如果我真的想精确一点，我应该在代码点之外再增加一个概念。实际上，有时有时可以使用多个代码点来表示单个字形（视觉字符）。这通常不是字母，而是字母或字母和符号的组合。例如，如果您有一个表示拉丁字母 a（＃$0061）的代码点序列，然后是表示严重重音符号（＃$0300）的代码点序列，则应将其显示为单个重音字符。

用 Object Pascal 编码术语，如果您编写以下内容（属于 CodePoints 应用程序项目的一部分），则消息将带有一个带重音的字符，如图 6.2 所示。

```
var
    Str: String;
  begin
    Str := #$0061 + #$0300;
    ShowMessage (Str);
```

图 6.2：单个字素可以是多个代码点的结果

在这种情况下，我们有两个字符，代表两个代码点，但只有一个字素（或视觉元素）。事实是，在拉丁字母中，您可以使用特定的 Unicode 代码点来表示给定的字形（带有重音的字母 a 是代码点$00E0），而在其他字母中，将 Unicode 代码点组合起来是获得给定字形的唯一方法字素（以及正确的输出）。

即使显示的是带重音符号的字符，也不会自动对值进行规范化或转换（仅是正确的显示），因此字符串内部与单个字符à的字符串保持不同。

❖ 从多个代码点渲染字素可能取决于操作系统的特定支持以及所使用的文本渲染技术，因此您可能会发现，对于某些字素，并非所有操作系统都提供正确的输出。

6.1.3 从代码点到字节（UTF）

ASCII 使用字符直接直接映射到其数字表示形式，而 Unicode 使用更复杂的方法。如前所述，Unicode 字母的每个元素都有一个关联的代码点，但是到实际

表示的映射通常更复杂。

Unicode 背后的混淆因素之一是，存在多种方式来表示实际的存储，物理字节，内存或文件中的相同代码点（或 Unicode 字符数字值）。

问题源于这样一个事实，即以一种简单统一的方式表示所有 Unicode 代码点的唯一方法是为每个代码点使用四个字节。这说明了固定长度的表示形式（每个字符总是需要相同数量的字节），但是大多数开发人员会认为这在内存和处理方面过于昂贵。

❖ 在 Object Pascal 中，可以使用 UCS4Char 数据类型直接以 4 字节表示形式表示 Unicode 代码点。

因此，Unicode 标准定义了其他表示形式，通常需要较少的内存，但是每个符号的字节数根据其代码点而有所不同。这个想法是对于最常见的元素使用较小的表示，对于不经常遇到的元素使用较长的表示。

Unicode 代码点的不同物理表示形式称为 Unicode 转换格式（或 UTF）。这些是 Unicode 标准的一部分，是算法映射，将每个代码点（字符的绝对数字表示形式）映射到表示给定字符的唯一字节序列。

注意，映射可以在两个方向上使用，可以在不同的表示形式之间来回转换。

该标准定义了三种格式，具体取决于用来表示集合的初始部分（初始 128 个字符）的位数是 8、16 或 32。有趣的是，注意到这三种编码形式都需要每个代码点最多 4 个字节的数据。

- UTF-8 将字符转换为 1 到 4 个字节的可变长度编码。UTF-8 在 HTML 和类似协议中很流行，因为当大多数字符（如 HTML 中的标签）落在 ASCII 子集中时，它非常紧凑。
- UTF-16 在许多操作系统（包括 Windows 和 macOS）和开发环境中都很流行。这非常方便，因为大多数字符都可以容纳在两个字节中，并且相当紧凑并且可以快速处理。
- UTF-32 在处理上很有意义（所有代码点都具有相同的长度），但它消耗内存，并且在实践中使用受限。

常见的误解是 UTF-16 可以直接映射两个字节的所有代码点，但是由于 Unicode 定义了超过 100,000 个代码点，因此您可以很容易地发现它们不适合 64K 元素。但是，确实，开发人员有时仅使用 Unicode 的一个子集，以使其适合于每个字符 2 个字节的定长表示形式。早期，Unicode 的这个子集称为 UCS-2，现在您经常看到它被称为基本多语言平面（BMP）。但是，这只是 Unicode 的子集（许多平面之一）。

❖ 与多字节表示形式（UTF-16 和 UTF-32）有关的问题是哪个字节在前？根据标准，所有形式都是允许的，因此您可以使用 UTF-16BE（big-endian 大端）或 LE（little-endian 小端），对于 UTF-32 也是如此。大端字节序列化具有最高有效字节优先，小端字节序列化具有最低有效字节优先。字节序列化通常与 UTF 表示一起在文件中用称为字节顺序标记（BOM）的标头标记。

6.1.4 The Byte Order Mark（字节顺序标记）

当您有一个存储 Unicode 字符的文本文件时，有一种方法可以指示哪种 UTF 格式用于代码点。信息存储在文件开头的标头或标记中，称为字节顺序标记（BOM）。这是一个签名，指示正在使用的 Unicode 格式和字节顺序形式（little

或 big–LE 或 BE）。下表提供了各种 BOM 的摘要，它们的长度可以为 2、3 或 4 个字节：

00 00 FE FF	UTF-32, big-endian（大端）
FF FE 00 00	UTF-32, little-endian（小端）
FE FF	UTF-16, big-endian （大端）
FF FE	UTF-16, little-endian（小端）
EF BB BF	UTF-8

我们将在本章后面看到 Object Pascal 如何在其流类中管理 BOM。BOM 出现在文件的开头，紧随其后的是实际 Unicode 数据。因此，内容为 AB 的 UTF-8 文件包含五个十六进制值（BOM 为 3，字母为 2）：

EF BB BF 41 42

如果文本文件没有这些签名，则通常将其视为 ASCII 文本文件，但它也可能包含具有任何编码的文本。

❖ 另一方面，当您从 Web 请求或通过其他 Internet 协议接收数据时，可能有一个特定的标头（协议的一部分）指示编码，而不是依赖 BOM。

6.1.5 看一下 Unicode

我们如何创建一个 Unicode 字符表，就像我之前显示的 ASCII 字符表一样？我们可以从基本多语言平面（BMP）中显示代码点开始，不包括所谓的代理对。

❖ 并非所有的数字值都是真正的 UTF-16 代码点，因为存在一些用于形成配对代码并表示高于 65535 的代码点的字符（称为替代）无效的数字值。代理对的一个很好的示例是 F（或低音）谱号的乐谱中使用的符号。它是代码点 1D122，在 UTF-16 中由两个值 D834 和 DD22 表示。

要显示 BMP 的所有元素，将需要 256*256 的网格，难以容纳在屏幕上。这就是 ShowUnicode 应用程序项目有一个包含两页的选项卡的原因：第一个选项卡具有 256 个块的主选择器，第二个页面则显示一个带有实际 Unicode 元素的网格，一次显示一个部分。该程序的用户界面比本书中的大多数其他用户略多。如果您只对它的输出（而不是内部）感兴趣，则可以简单地浏览其代码。

程序启动时，它将用 256 个条目填充 TabControl 第一页中的 ListView 控件，每个条目指示一组 256 个字符的第一个和最后一个字符。这是该表格的 OnCreate 事件处理程序的实际代码， 用于显示每个元素的函数，产生图 6.3 的输出：

```
// helper function（辅助函数）
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 页*每个 256 个字符
  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 应用程序项目的第一页上有一串长长的 Unicode 字符列表

请注意，代码是如何在 ListView 的项的 Tag 属性中保存“页面”的编号的，该信息稍后用于填充页面。当用户选择其中一项时，应用程序将移至 TabControl 的第二页，用该部分的 256 个字符填充其字符串网格：

```
    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;
```

上面的代码中使用的 IfThen 函数是一种双向测试：如果第一个参数中传递的条件为 true，则该函数返回第二个参数的值；如果不是，则返回第三个值。第一个参数中的测试使用 Char 类型帮助器的 IsControl 方法来过滤掉不可打印的控制字符。

❖ IfThen 函数或多或少地像大多数基于 C 语法的编程语言的？：运算符一样运行。有一个用于字符串的版本，一个用于整数的单独版本。对于字符串版本，必须包括 System.StrUtils 单元，对于 IfThen 的整数版本，必须包括 System.SysUtils 单元。

由应用程序产生的 Unicode 字符网格在图 6.4 中可见。

请注意，输出会根据所选字体和特定操作系统显示给定 Unicode 字符的能力而有所不同。

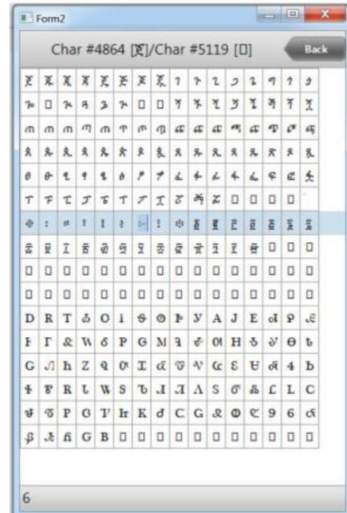

图 6.4：ShowUnicode 应用程序项目的第二页包含一些实际的 Unicode 字符

6.2 字符类型再探

在介绍完 Unicode 之后，让我们回到本章的主题，即 Object Pascal 语言如何管理字符和字符串。我在第 2 章介绍了 Char 数据类型，并提到了 Character 单元中可用的一些类型帮助函数。既然您对 Unicode 有了更好的了解，那么值得回顾一下本节并详细介绍一些细节。

首先，Char 类型不会始终代表 Unicode 代码点。实际上，数据类型为每个元素使用 2 个字节。尽管确实代表 Unicode'e 基本多语言平面（BMP）中元素的代码点，但是 Char 也可以是代表代码点的一对替代值的一部分。

从技术上讲，可以使用另一种类型来直接表示任何 Unicode 代码点，这是 UCS4Char 类型，它使用 4 个字节表示一个值。这种类型很少使用，因为通常很难证明需要额外的内存，但是您可以看到，字符单元（接下来介绍）也包含针对此数据类型的若干操作。

回到 Char 类型，请记住它是一种序数类型（即使是相当大的类型），因此它具有序列的概念，并提供了 Ord，Inc，Dec，High 和 Low 之类的代码操作。大多数扩展操作，包括特定的类型帮助程序，都不是基本系统 RTL 单元的一部分，但需要包含 Character 单元。

6.2.1 字符单元的 Unicode 操作

Unicode 字符（当然还有 Unicode 字符串）的大多数特定操作是在称为 System.Character 的特殊单位中定义的。该单元为 Char 类型定义了 TCharHelper 帮助器，使您可以将操作直接应用于该类型的变量。

❖ 字符单元还定义了一个 TCharacter 记录，该记录基本上是静态类函数的集合，以及许多映射到这些方法的全局例程。鉴于现在在 Unicode 级别上处理 Char 类型的首选方法是使用类帮助器，因此这些功能已被弃用。

该单元还定义了两种有趣的枚举类型。第一个称为 TUnicodeCategory，它在广泛的类别中映射各种字符，例如控件，空格，大写或小写字母，十进制数字，标点符号，数学符号等等。第二个枚举称为 TUnicodeBreak，它定义各种空格，连字符和中断的族。如果您习惯于 ASCII 操作，这是一个很大的变化。

Unicode 中的数字不仅是 0 到 9 之间的字符，而且还包括数字。空格不限于字符＃32；对于（更简单的）256 元素字母的许多其他假设，依此类推。

Char 类型帮助器具有 40 多种方法，其中包括许多不同的测试和操作。它们可用于：

- 获取字符的数字表示形式（GetNumericValue）。
- 询问类别（GetUnicodeCategory）或对照各种类别（IsLetterOrDigit，IsLetter，IsDigit，IsNumber，IsControl，IsWhiteSpace，IsPunctuation，IsSymbol 和 IsSeparator）之一进行检查。我在上一个演示中使用了 IsControl 操作。
- 检查它是小写还是大写（IsLower 和 IsUpper）或进行转换（ToLower 和 ToUpper）。
- 验证它是否属于 UTF-16 代理对（IsSurrogate，IsLowSurrogate 和 IsHighSurrogate），并以各种方式转换代理对。
- 在 UTF32（ConvertFromUtf32 和 ConvertToUtf32）和 UCS4Char 类型（ToUCS4Char）之间进行转换。
- 检查它是否属于给定字符列表（IsInArray）。

请注意，其中某些操作可以应用于整个类型，而不是特定的变量。这样您就可以使用 Char 类型作为前缀来调用它们，如下面的第二个代码片段所示。

为了对 Unicode 字符进行这些操作的实验，我创建了一个名为 CharTest 的应用程序项目。此演示的示例之一是对 Unicode 元素调用大写和小写操作的效果。实际上，RTL 的经典 UpCase 函数仅适用于 ANSI 表示形式的基础 26 个英语语言字符，而它却使某些确实具有特定大写表示形式的 Unicode 字符失效（并非所有字母都具有大写概念，所以这是不是普遍的概念）。

为了测试这种情况，在 CharTest 应用程序项目中，我添加了以下代码段，该代码段尝试将重音字母转换为大写字母：

```
var
  Ch1: Char;
begin
  Ch1 := 'ù';
  Show ('UpCase ù: ' + UpCase(Ch1));
  Show ('ToUpper ù: ' + Ch1.ToUpper);
```

传统的 Upcase 调用不会转换拉丁字母的重音字符，而 ToUpper 函数则可以正常工作：

```
UpCase ù: ù
ToUpper ù: Ù
```

Char 类型帮助器中有许多与 Unicode 相关的功能，如下面的代码中突出显示的功能，该功能将字符串定义为还包括 BMP（Unicode 代码点的前 64K）之外的字符。该代码段也是 CharTest 应用程序项目的一部分，对字符串的各个元素进行了一些测试，所有测试均返回 True：

```
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;
```

在这种情况下使用的显示功能是改编版：

```
procedure TForm1.ShowBool(value: Boolean);
```

```
begin
  Show(BoolToStr (Value, True));
end;
```

❖ Unicode 代码点$ 1D11E 是音乐符号 G 谱号

6.2.2 Unicode 字符文字

我们在几个示例中看到，您可以将单个字符文字或字符串文字分配给字符串类型的变量。通常，使用带有＃前缀的字符的数字表示非常简单。但是，也有一些例外。为了向后兼容，纯字符文字将根据其上下文进行转换。对数值 128 进行以下简单分配，它可能表示使用了欧元符号（€）：

```
var
  Str1: string;
begin
  Str1 := #$80;
```

该代码不符合 Unicode，因为该符号的代码点为 8364。实际上，该值不是来自官方的 ISO 代码页，而是针对 Windows 的特定 Microsoft 实现。 为了使将现有代码轻松转换为 Unicode，Object Pascal 编译器可以将 2 位字符串文字视为 ANSI 字符（这可能取决于您的实际代码页）。令人惊讶的是，如果您将该值转换为 Char，然后再次显示，则数值表示将更改为正确的值。因此，通过执行以下语句：

```
Show (Str1 + ' - ' + IntToStr (Ord (Str1[1])));
```

我将得到输出：

```
€ - 8364
```

鉴于您可能希望完全迁移代码并摆脱旧的基于 ANSI 的文字值，可以使用特殊指令$ HIGHCHARUNICODE 更改编译器行为。该指令确定编译器如何处理＃$ 80 和＃$ FF 之间的文字值。我之前讨论的是默认选项（OFF）的效果。如果将其打开，则同一程序将产生以下输出：

```
 - 128
```

该数字被解释为实际的 Unicode 代码点，并且输出将包含不可打印的控制字符。表示特定代码点（或＃$ FFFF 以下的任何 Unicode 代码点）的另一种方法是使用四位数表示法：

```
str1 := #$0080;
```

无论$ HIGHCHARUNICODE 指令的设置如何，都不会将其解释为欧元符号。

❖ 上面的代码和匹配的演示仅适用于美国或西欧地区。在其他语言环境中，128 和 255 之间的字符将被不同地解释。

很好的是，您可以使用四位数字表示法来表示远东字符，例如以下两个日语字符：

```
Str1 := #$3042#$3044;
Show (Str1 + ' - ' + IntToStr (Ord (Str1.Chars[0])) +
  ' - ' + IntToStr (Ord (Str1.Chars[1])));
```

显示为（及其整数表示形式）：

```
あい - 12354 – 12356
```

您还可以使用＃$ FFFF 以上的文字元素，这些文字元素将转换为适当的代理对。

6.2.3 1 字节字符呢？

如前所述，Object Pascal 语言将 Char 类型映射为 WideChar，但仍定义 AnsiChar 类型，主要是为了与现有代码兼容。

通常的建议是将 Byte 类型用于一个字节的数据结构，但确实，在进行 1 字节字符处理时，AnsiChar 可以派上用场。

虽然在移动平台上没有使用多个版本的 AnsiChar，但从 10.4 开始，此数据类型在所有 Delphi 编译器中均相同。将数据映射到平台 API 或保存到文件时，即使支持，通常也应远离旧的一字节字符。到目前为止，使用 Unicode 编码是首选方法。但是，与 2 字节字符处理相比，1 字节字符处理可以更快并且使用更少的内存，这是事实。

6.3 字符串数据类型

Object Pascal 中的字符串数据类型比简单的字符数组复杂得多，并且具有远远超出大多数编程语言对类似数据类型所做的功能。在本节中，我将介绍此数据类型背后的关键概念，在接下来的部分中，我们将更详细地探讨其中的一些功能。

在以下项目符号列表中，我捕获了理解字符串如何在该语言中工作的关键概念（请记住，由于内部行为非常透明，因此您可以在不了解太多内容的情况下使用字符串）：

- 字符串类型的数据在 heap（堆）上**动态分配**。字符串变量只是对实际数据的引用。您不必为此担心很多，因为编译器可以透明地进行处理。就像动态数组一样，在声明新字符串时，该字段为空。
- 尽管您可以通过多种方式将数据分配给字符串，但也可以通过调用 SetLength 函数来**分配特定的内存区域**。该参数是字符数（每个字符 2 个字节），字符串应具有的字符数。扩展字符串时，将保留现有数据（但可能会将其移动到新的物理内存位置）。减小大小时，某些内容可能会丢失。很少需要设置字符串的长度。唯一常见的情况是，您需要将字符串缓冲区传递给给定平台的操作系统功能。
- 如果要**增加内存中字符串的大小**（通过将其与另一个字符串连接），但是相邻内存中还有其他内容，则该字符串不能在相同的内存位置中增长，因此该字符串的完整副本必须因此应在其他位置制作。
- 要清除字符串，您无需对引用本身进行操作，而只需将其设置为空字符串，即”。或者，您可以使用与该值相对应的 Empty 常数。
- 根据 Object Pascal 的规则，**字符串的长度**（可以通过调用 Length 获得）是有效元素的数量，而不是分配元素的数量。与 C 具有字符串终止符（＃0）的概念不同，自早期以来，所有 Pascal 版本都倾向于使用存储实际长度信息的特定存储区（字符串的一部分）。但是，有时您会发现带有终止符的字符串。
- Object Pascal 字符串使用**参考计数**机制，该机制跟踪内存中有多少字符串变量引用给定字符串。当不再使用字符串时，即不再有引用数据的字符串变量时，引用计数将释放内存...并且引用计数达到零。
- 字符串使用高效的**写时复制**技术。当您将一个字符串分配给另一个字符串或将一个字符串传递给字符串参数时，不会复制任何数据，并且增加引用计数。但是，如果您确实更改了其中一个参考的内容，系统将首先进行复制，然后仅更改该副本，而其他参考则保持不变。
- 使用**字符串串联**将内容添加到现有字符串中通常非常快，并且没有明显的缺

点。尽管有其他方法，但是串联字符串既快捷又强大。如今，对于许多编程语言而言，情况并非如此。

现在，我猜想这个描述可能会有些混乱，所以让我们看一下实际中字符串的使用。一会儿，我将演示一个演示上述操作的演示，包括引用计数和写时复制。但是，在开始之前，让我回到字符串帮助程序操作和其他一些用于字符串管理的基本 RTL 函数。

在继续进行之前，让我根据实际代码检查上一列表中的某些元素。鉴于字符串操作是非常无缝的，除非您开始查看字符串存储结构的内部结构，否则就很难完全掌握发生的情况，这在本章的后面将进行介绍，因为它现在太高级了。因此，让我们从从 Strings101 应用程序项目中提取的一些简单的字符串操作开始：

```
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;
```

该第一个代码片段在执行时显示，如果您将两个字符串分配给相同的内容，则修改一个字符串不会影响另一个字符串。也就是说，String1 不受 String2 更改的影响：

```
1: hello world
2: hello world
1: hello world
2: hello world, again
```

不过，由于我们在以后的演示中会更好地发现，初始分配不会导致字符串的完整副本，因此副本会延迟（再次，称为**写时复制**）。

要理解的另一个重要功能是长度的管理方式。如果您要求一个字符串的长度，您将获得实际值（该值存储在字符串 meta（元）数据中，从而使操作非常快）。但是，如果调用 SetLength，则说明您正在分配内存，而该内存通常不会初始化。通常在将字符串作为缓冲区传递给外部系统函数时使用。如果需要空白字符串，则可以使用伪构造函数（create 创建）。最后，您可以使用 SetLength 修剪字符串。以下代码演示了所有这些：

```
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);
```

输出大致如下：

```
hello world
Length: 11
hello world~~~~ ~ H~ʋ~ ~ J~Щ~ы~ ~ ~ ~ ~ ~ ~ ~ ~ ʻ ჲ ѫ ƥ ɏ ə Ӽ ҝ Ꞇ h
~ ~گ~٦~٦~ڤ~؛~~٦~ ~
ے   〉 ﹛ ~ ~ ~۵ ~ټ~ټ~ټ~ټ~ټ~~ ~~~ټ~~ ~
Length: 100
hello world
Length: 11
hello world
Length: 100
```

我要在本节中强调的第三个概念是空字符串。字符串的内容为空字符串时为空。对于分配和测试，您可以使用两个连续的引号或特定的函数：

```
var
  String1: string;
begin
  String1 := 'hello world';
  if String1 = '' then
    Show('Empty')
  else
    Show('Not empty');
  String1 := ''; // 或者 String1.Empty;
  if String1.IsEmpty then
    Show('Empty')
  else
    Show('Not empty');
```

以下简单输出：

```
Not empty
Empty
```

6.3.1 将字符串作为参数传递

正如我所解释的，如果将一个字符串分配给另一个字符串，则只是在复制引用，而不会复制内存中的实际字符串。但是，如果您编写更改该字符串的代码，则会首先复制该字符串（仅在该点），然后再进行修改。

当将字符串作为参数传递给函数或过程时，会发生非常相似的事情。默认情况下，您会获得一个新的引用，并且如果您在函数中修改字符串，则更改不会影响原始字符串。如果您想要不同的行为，即能够修改函数中的原始字符串，则需要使用 var 关键字（大多数其他简单数据和托管数据类型都会发生这种情况）按引用传递字符串。

但是，如果您不修改作为参数传递的字符串怎么办？在这种情况下，可以通过对参数使用 const 修饰符来应用实际的优化。在这种情况下，编译器将不允许您在函数或过程中更改字符串，但是它还将优化参数传递操作。实际上，const 字符串不需要函数在开始时增加字符串引用计数，而在结束时减少它。尽管这些操作非常快，但是执行这些数千或数百万次将给您的程序增加一点开销。这就是为什么在不必修改字符串参数值的情况下建议将字符串作为 const 传递的原因（尽管以下说明中涵盖了潜在问题）。

用编码的术语来说，这是三个过程的声明，它们以不同的方式传递字符串参数：

```
procedure ShowMsg1 (Str: string);
procedure ShowMsg2 (var Str: string);
procedure ShowMsg3 (const Str: string);
```

❖ 近年来，大力推动将所有字符串参数作为 const 传递，除非函数和方法修改了这些字符串。但是，有一个很大的警告。对于恒定的字符串参数，编译器将获取字符串引用，而不对其进行“manage 托管”（不进行引用计数等），将其视为指向内存位置的指针。编译器会正确检查例程代码是否不会更改字符串参数。但是，它无法控制参数所引用的原始字符串会发生什么情况。对该字符串的更改可能会影响其内存布局和位置，这是常规字符串参数可以处理的事情（具有多个引用的字符串会自动执行写时复制操作），而常量字符串参数将忽略。换句话说，对原始字符串的更改会使引用它的 const 参数无效，并且使用它很可能会导致内存访问错误。

6.3.2 []和字符串字符计数模式的使用

您可能知道您是否使用过 Object Pascal 或任何其他编程语言，所以按键字符串操作正在访问字符串的元素之一，这通常使用方括号表示法（[]）来实现，操作方式与 访问数组的元素。

在 Object Pascal 中，有两种略有不同的方式来执行这些操作：

- Chars[]字符串类型帮助程序操作（整个列表在下一节中）是使用基于 0 的索引的只读字符访问。
- 标准[]字符串运算符支持读取和写入，并且默认情况下使用经典的 Pascal 基于 1 的索引。可以使用编译器指令修改此设置。

在下面简短的历史说明之后，我将对此事进行澄清。产生此注释的原因是，如果不感兴趣，您可以跳过该注释，因为如果不查看一段时间后发生的情况，就很难理解该语言为何以当前方式运行。

❖ 让我回顾过去，向您解释我们今天如何到达这里。在 Pascal 语言的早期，字符串被视为字符数组，其中第一个元素（即数组的元素零）用于存储字符串中的有效字符数或字符串长度。在那些日子里，尽管 C 语言每次都必须重新计算字符串的长度，以寻找 NULL 终止符，但是 Pascal 代码可以直接检查该字节。给定长度使用字节数字零，碰巧存储在字符串中的第一个实际字符位于位置 1。

❖ 随着时间的流逝，几乎所有其他语言都有从零开始的字符串和数组。后来，Object Pascal 使用了从零开始的动态数组，并且大多数 RTL 和组件库都使用了从零开始的数据结构，字符串是一个明显的例外。

❖ 进入移动世界时，Object Pascal 语言设计人员决定为基于零的字符串提供“优先级”，以允许开发人员在已有 Object Pascal 源代码移至原处的情况下仍使用旧模型，从而通过编译器指令。但是，在 Delphi10.4 中，原始决策被恢复为考虑到源代码的更高一致性，而与目标平台无关。换句话说，决定支持“单源多平台”目标，而不是“像其他现代语言一样”目标。

如果我们想进行比较以更好地解释指数基数的差异，请考虑如何计算欧洲和北美的底楼（老实说，我不知道世界其他地方）。在欧洲，一楼是 0 楼，一楼是它上面的一层（有时正式表示为“地上一层”）。在北美，第一层是地面层，第二层是地面上的第一层。换句话说，美国使用基于 1 的最低指数，而欧洲使用基于

0 的最低指数。

对于字符串，无论哪种语言被发明，绝大多数编程语言都使用基于 0 的表示法。Delphi 和大多数 Pascal 方言都使用基于 1 的表示法。。

让我用字符串索引更好地解释这种情况。如上所述，Chars[]表示法始终使用基于零的索引。所以如果你写：

```
var
  String1: string;
begin
  String1 := 'hello world';
  Show (String1.Chars[1]);
```

输出将是：

```
e
```

如果使用直接[]表示法，那将是以下内容的输出：

```
  Show (String1[1]);
```

默认情况下，输出为 h。但是，如果编译器定义$ZEROBASEDSTRING 已打开，则可能是 e。当前（即在 Delphi10.4。发布之后）的建议是在所有代码中使用基于 1 的字符串，并避免使用不同的模型。

但是，如果您想编写不管$ZEROBASEDSTRING 设置如何都有效的代码，该怎么办？您可以抽象化索引，例如使用 Low(string)作为第一个值的索引，并使用 High(string)作为最后一个值的索引。这些工作根据字符串基数的本地编译器设置返回正确的值：

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

换句话说，字符串始终具有从 Low 函数的结果到应用于同一字符串的 High 函数的结果的元素。

❖ 字符串只是一个字符串，从零开始的字符串的概念是完全错误的。内存中的数据结构没有任何不同，因此您可以将任何字符串传递给使用带有任何基值的表示法的任何函数，并且完全没有问题。换句话说，如果您有一个代码片段以基于零的符号访问字符串，则可以将该字符串传递给使用基于一的符号的设置编译的函数。

6.3.3 连接字符串

我已经提到过，与其他语言不同，Object Pascal 完全支持直接字符串连接，这实际上是一个相当快的操作。在本章中，我将向您展示一些字符串连接代码，同时进行一些速度测试。稍后，在第 18 章中，我将简要介绍 TStringBuilder 类，该类遵循.NET 表示法，用于从不同片段中组装字符串。尽管有使用 TStringBuilder 的理由，但性能并不是最相关的（如以下示例所示）。

那么，如何在 Object Pascal 中连接字符串？只需使用+运算符：

```
var
  Str1, Str2: string;
begin
  Str1 := 'Hello,';
```

```
Str2 := ' world';
Str1 := Str1 + Str2;
```

请注意，我是如何在赋值的左侧和右侧同时使用 str1 变量的，将更多内容添加到现有字符串中，而不是分配给全新的字符串。两种操作都是可行的，但是将内容添加到现有字符串中可以使您获得不错的性能。

也可以在循环中完成这种连接，例如从 LargeString 应用程序项目中提取以下内容：

```
uses
  System.Diagnostics;
const
  MaxLoop = 2000000; //  两百万
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;
```

通过运行此代码，我在 Windows 虚拟机和 Android 设备（计算机快很多）上获得了以下计时：

```
Length: 12000006 // Windows (在虚拟机中)
Concatenation: 59
Length: 12000006 // Android (在设备上)
Concatenation: 991
```

该应用程序项目还具有基于 TStringBuilder 类的类似代码。

尽管我不想了解该代码的细节（再次，我将在第 18 章中描述该类），但我想共享实际的时序，以便与刚刚显示的普通串联时序进行比较。

```
Length: 12000006 // Windows (在虚拟机中)
StringBuilder: 79
Length: 12000006 // Android (在设备上)
StringBuilder: 1057
```

如您所见，可以将串联安全地视为最快的选择。

6.3.4 字符串 Helper（助手）操作

考虑到字符串类型的重要性，毫不奇怪的是，这种类型的帮助器具有相当长的可执行操作列表。考虑到它的重要性以及这些操作在大多数应用程序中的通用性，我认为值得细心阅读这份清单。

使用 Delphi 经典的全局字符串操作函数和字符串帮助器方法之间有一个关键的区别：经典操作假定一个基于字符串的字符串，而字符串帮助程序使用基于零的逻辑！

我已经在逻辑上对字符串助手操作进行了分组（其中大多数都有许多重载的

版本），简短地描述了它们的功能，并经常考虑到它们的名称很直观：

- 复制或部分复制操作，例如 Copy，CopyTo，Join 和 SubString。
- 字符串修改操作，如 Insert，Remove 和 Replace。
- 为了从各种数据类型转换为字符串，可以使用“ Parse（解析）”和“Format（格式）”。
- 可以使用 ToBoolean，ToInteger，ToSingle，ToDouble 和 ToExtended 转换为各种数据类型，同时可以使用 ToCharArray 将字符串转换为字符数组。
- 用 PadLeft，PadRight 和 Create 的重载版本之一填充字符串空格或特定字符。相反，您可以使用 TrimRight，TrimLeft 和 Trim 删除字符串一端的空白，或同时删除两者。
- 字符串比较和相等性测试（Compare，CompareOrdinal，CompareText，CompareTo 和 Equals）-但请记住，在某种程度上，您还可以使用相等运算符和比较运算符。
- 使用 LowerCase 和 UpperCase，ToLower 和 ToUpper 和 ToUpperInvariant 更改大小写。
- 使用 Contains，StartsWith，EndsWith 等操作测试字符串内容。可以使用 IndexOf 查找字符串（从开头或从指定位置开始），类似的 IndexOfAny（查找字符数组的元素之一），LastIndexOf 和 LastIndexOfAny 来查找字符串。从字符串末尾开始反向执行的操作，以及特殊用途的 IsDelimiter 和 LastDelimiter（界定符）。
- 使用如下函数访问有关字符串的常规信息，这些函数包括：Length（返回字符数），CountChars（也将代理 pairs 对考虑在内），GetHashCode（返回字符串的哈希值）以及各种针对“空性”的测试，其中包括 IsEmpty，IsNullOrEmpty 和 IsNullOrWhiteSpace。
- 字符串特殊操作，例如 Split，它根据特定字符将一个字符串分成多个字符串，并使用 QuotedString 和 DeQuoted 在字符串周围删除或添加引号。
- 最后，使用 Chars[]访问各个字符，该字符在方括号中具有字符串元素的数字索引。它只能用于读取值（不能更改值），并且像所有其他字符串帮助程序操作一样使用从零开始的索引。

重要的是要注意，实际上，所有字符串帮助程序方法都是按照许多其他语言使用的字符串约定构建的，该约定包括以下概念：字符串元素从零开始，一直到字符串的长度减去一。换句话说，正如我之前已经提到过的，但值得再次强调，所有字符串 helper（助手）操作都使用基于零的索引作为参数和返回值。

❖ Split（拆分）操作对 Object Pascal RTL 来说是相对较新的。先前的常见方法是在设置特定的行分隔符之后，将字符串加载到字符串列表中，然后再访问单个字符串或行。Split（拆分）操作显着提高了效率和灵活性。

鉴于您可以直接对字符串进行大量操作，因此我可以创建几个项目来演示这些功能。相反，我将坚持一些相对简单的操作，尽管它们很常见。

StringHelperTest 应用程序项目有两个按钮。在它们的每一个中，代码的第一部分都会构建并显示一个字符串：

```
var
  Str1, Str2: string;
  I, NIndex: Integer;
begin
  Str1 := '';
  // 创建字符串
```

```
for I := 1 to 10 do
  Str1 := Str1 + 'Object ';
Str2:= string.Copy (Str1);
Str1 := Str2 + 'Pascal ' + Str2.Substring (10, 30);
Show(Str1);
```

注意我如何使用复制功能创建字符串数据的唯一副本，而不是别名...即使在此特定演示中，它也没有任何区别。 最后的 Substring 调用用于提取字符串的一部分。

结果文本为：

Object Object Object Object Object Object Object Object Object Object
Pascal ect Object Object Object Objec

初始化之后，第一个按钮具有用于搜索子字符串并使用不同的初始索引重复进行此搜索的代码，以计算给定字符串（在示例中为单个字符）的出现次数：

```
// 查找子串
Show('Pascal at: ' + Str1.IndexOf ('Pascal').ToString);
// 计算发生次数
I := -1;
NCount := 0;
repeat
  I := Str1.IndexOf('O', I + 1); // 从下一个元素搜索
  if I >= 0 then
    Inc (NCount); // 找到一个
until I < 0;
Show('O found: ' + NCount.ToString + ' times');
```

我知道重复循环不是最简单的循环：它以负索引开头，因为任何后续搜索都以当前索引之后的索引开头；它计算发生次数；并且其终止基于以下事实：如果未找到该元素，则返回-1。

代码的输出为：

Pascal at: 70
O found: 14 times

第二个按钮具有执行搜索并将字符串的一个或多个元素替换为其他内容的代码。在第一部分中，它将创建一个新的字符串，该字符串将复制初始和最终部分，并在中间添加一些新文本。在第二种方法中，它使用 Replace 函数，只需将适当的标志（rfReplaceAll）传递给它，即可对多次出现进行操作。

这是代码：

```
// 单次更换
NIndex := Str1.IndexOf ('Pascal');
Str1 := Str1.Substring(0, NIndex) + 'Object' +
  Str1.Substring(nIndex + ('Pascal').Length);
Show (Str1);
// 多次替换
Str1 := Str1.Replace('O', 'o', [rfReplaceAll]);
Show (Str1);
```

由于输出相当长且不容易阅读，因此在这里我仅列出了每个字符串的中央部分：

...Object Pascal ect Object Object...
...Object Object ect Object Object...
...object object ect object object...

同样，这只是丰富字符串操作的最小样本量，您可以使用字符串类型助手使用可用于字符串类型的操作来执行丰富字符串操作。

6.3.5 更多字符串 RTL 函数

在其他编程语言中常见的操作名称之后决定执行字符串帮助程序的决定的结果是，类型操作的名称通常与传统的 Object Pascal（现在仍可作为全局函数使用）不同。

下表列出了一些不匹配的函数名称：

Global（全局）	string type helper（字符串助手）
Pos	IndexOf
IntToStr	Parse
StrToInt	ToInteger
CharsOf	Create
StringReplace	Replace

请记住，global 操作和 Char 助手操作之间有很大的区别：第一组使用一个基于符号的表示法为字符串中的元素建立索引，而第二组使用基于零的表示法（如前所述）。

这些只是已更改名称的字符串 RTL 的最常用功能，而许多其他功能仍使用相同的功能，例如 UpperCase 或 QuotedString。System.SysUtils 单元有很多函数，而特定的 System.StrUtils 单元还具有许多专注于字符串操作的函数，这些函数不属于字符串 helper（助手）。

System.StrUtils 单元的一些值得注意的函数部分是：

- ResemblesText，它实现了 Soundex 算法（即使拼写不同，也可以找到具有相似声音的单词）；
- DupeString，返回给定字符串请求的副本数；
- IfThen，如果条件为真，它将返回传递的第一个字符串，否则它将返回第二个字符串（我在本章前面的代码段中使用了此函数）；
- ReverseString，返回具有相反字符序列的字符串。

6.3.6 格式化字符串

在使用加号（+）运算符连接字符串并使用某些转换函数时，您确实可以从各种数据类型的现有值中构建复杂的字符串，但还有另一种更强大的方法来格式化数字，货币值和其他字符串 转换为最终字符串。可以通过调用 Format 函数来实现复杂的字符串格式设置，Format 函数是一种非常传统但仍然极为普遍的机制，不仅在 Object Pascal 中而且在大多数编程语言中也是如此。

✓ “打印格式字符串”或 printf 函数家族的历史可以追溯到程序创建和诸如 FORTRAN 66，PL / 1 和 ALGOL 68 之类的语言的早期。今天仍在使用的特定格式字符串结构（由 Object Pascal 使用 ）接近 C 语言的 printf 函数。有关他的理论概述，请参阅 en.wikipedia.org/wiki/Printf_format_string

Format 函数需要一个带有基本文本的字符串和一些占位符（用%符号标记）和一个值数组（通常每个占位符一个）作为参数。例如，要将两个数字格式化为字符串，可以编写：

```
Format ('First %d, Second %d', [n1, n2]);
```

其中 n1 和 n2 是两个 Integer 值。第一个占位符将替换为第一个值，第二个将与第二个匹配，依此类推。如果占位符的输出类型（由%符号后面的字母表示）与相应参数的类型不匹配，则会发生运行时错误。没有编译时类型检查实际上是使用 Format 函数的最大缺点。同样，未传递足够的参数会导致运行时错误。

Format 函数使用开放数组参数（该参数可以具有任意数量的值或任意数据类型，如第 5 章所述）。除了使用%d 之外，您还可以使用此函数定义的许多其他占位符之一，并简要列出下表。这些占位符为给定的数据类型提供默认输出。但是，您可以使用其他格式说明符来更改默认输出。例如，宽度说明符确定输出中的固定数目的字符，而精度说明符指示十进制数字的数目。例如，

```
Format ('%8d', [n1]);
```

将数字 n1 转换为八个字符的字符串，将文本右对齐（使用减号（-）指定左对齐），并在其中填充空白。这是各种数据类型的格式占位符的列表：

d (decimal)	相应的整数值将转换为十进制数字字符串。
x (hexadecimal)	相应的整数值将转换为十六进制数字的字符串。
p (pointer)	相应的指针值将转换为以十六进制数字表示的字符串。
s (string)	相应的字符串，字符或 PChar（指向字符数组的指针）值将复制到输出字符串。
e (exponential)	相应的浮点值将根据科学计数法转换为字符串。
f (floating point)	相应的浮点值将基于浮点表示法转换为字符串。
g (general)	使用浮点或指数表示法将相应的浮点值转换为最短的十进制字符串。
n (number)	相应的浮点值将转换为浮点字符串，但也使用数千个分隔符。
m (money)	相应的浮点值将转换为表示货币金额的字符串。转换通常基于区域设置。

查看这些转换示例的最佳方法是自己尝试格式字符串。 为了使此操作更容易，我编写了 FormatString 应用程序项目，该项目允许用户为一些预定义的整数值提供格式化字符串。

程序的形式在按钮上方有一个编辑框，最初包含一个简单的预定义格式字符串，用作占位符（'%d-%d-%d'）。应用程序的第一个按钮使您可以在编辑框中显示更复杂的示例格式字符串（该代码对格式字符串'Value%d，Align%4d，Fill%4.4d'的编辑文本进行了简单分配）。第二个按钮使您可以使用以下代码将格式字符串应用于预定义的值：

```
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 (''); //  空行
end;
```

如果首先显示带有初始格式字符串的输出，然后显示带有示例格式字符串的

输出（也就是说，如果您按第二个按钮，则按第一个按钮，然后再按第二个按钮），则应该得到如下输出：

```
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
```

但是，该程序背后的想法是编辑格式字符串并进行试验，以查看所有各种可用的格式设置选项。

6.3.7 字符串的内部结构

尽管您通常可以在不了解字符串内部知识的情况下使用字符串，但查看此数据类型背后的实际数据结构很有趣。在 Pascal 语言的早期，字符串最多包含 255 个元素，每个元素一个字节，并且将使用第一个字节（或零字节）存储字符串长度。自从早期以来，已经有很多时间了，但是关于字符串存储一些额外信息作为其数据的一部分的概念仍然是 Object Pascal 语言的一种特定方法（不同于许多从 C 派生并使用字符串终止符）。

❖ ShortString 是传统 Pascal 字符串类型的名称，一个 1 字节字符的字符串或 AnsiChar 限制为 255 个字符。ShortString 类型在桌面编译器中仍然可用，但在移动编译器中不可用。您可以用字节或 TBytes 的动态数组或 Byte 元素的普通静态数组来表示类似的数据结构。

正如我已经提到的，字符串变量不过是指向在堆上分配的数据结构的指针。实际上，存储在字符串中的值不是对数据结构开头的引用，而是对字符串的第一个字符的引用，并且字符串元数据数据在相对于该位置的负偏移量处可用。

字符串类型的数据在内存中的表示形式如下：

-12	-10	-8	-4	字符串参考地址
Code page	Elem size	Ref count	Length	字符串的第一个字符

第一个元素（从字符串本身的开头开始倒数）是一个带字符串长度的 Integer，第二个元素保存引用计数。其他字段（用于桌面编译器）是元素大小（以字节为单位（1 或 2 字节））和代码页（用于基于 Ansi 的旧字符串类型）。

出乎意料的是，除了很明显的 Length 函数之外，还可以使用特定的低级字符串元数据函数访问这些字段中的大多数：

```
function StringElementSize(const S: string): Word;
function StringCodePage(const S: string): Word;
function StringRefCount(const S: string): Longint;
```

例如，您可以像在 StringMetaTest 应用程序项目中那样创建一个字符串并询问有关它的一些信息：

```
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);
```

❖ 程序有一个特定的原因可以动态地构建“ Foo”字符串，而不是分配一个常量，这是因为常量字符串禁用了引用计数（或设置为-1）。在演示中，我更喜欢为参考计数显示一个适当的值，因此是动态字符串构造。

该程序产生类似于以下内容的输出：

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: 1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

UnicodeString 返回的代码页为 1200，这是存储在全局变量 DefaultUnicodeCodePage 中的数字。在上面的代码（及其输出）中，您可以清楚地注意到字符串变量的大小（始终为 4），逻辑长度和物理长度（以字节为单位）之间的差异。

这可以通过将每个字符的字节大小乘以字符数来获得，或者通过调用 ByteLength 获得。但是，后一个函数不支持较旧的桌面编译器的某些字符串类型。

6.3.8 查看内存中的字符串

查看字符串的元数据的功能可用于更好地了解字符串内存管理的工作原理，特别是与引用计数的关系。为此，我在 StringMetaTest 应用程序项目中添加了更多代码。

该程序有两个全局字符串：MyStr1 和 MyStr2。程序将动态字符串分配给两个变量中的第一个（出于本注释前面所述的原因），然后将第二个变量分配给第一个：

```
MyStr1 := string.Create(['H', 'e', 'l', 'l', 'o']);
MyStr2 := MyStr1;
```

除了处理字符串，该程序还使用以下 StringStatus 函数显示其内部状态：

```
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;
```

在 StringStatus 函数中，重要的是将字符串参数作为 const 参数传递。通过复制传递此参数将导致副作用，即在执行函数时对字符串有一个额外的引用。相比之下，通过引用（var）或常量（const）传递参数并不意味着对该字符串的进一步引用。在这种情况下，我使用了 const 参数，因为该函数不应该修改字符串。

为了获取字符串的内存地址（用于确定其实际身份并查看两个不同的字符串何时引用相同的内存区域），我简单地进行了从字符串类型到 Integer 类型的硬编码类型转换。字符串是引用-实际上，它们是指针：它们的值保存字符串的实际内存位置，而不是字符串本身。

用于测试字符串发生什么的代码如下：

```
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
MyStr1 [1] := 'a';
Show ('Change 2nd char');
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
```

最初，您应该获得两个具有相同内容，相同存储位置和引用计数为 2 的字符串。

```
MyStr1 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
MyStr2 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
```

当应用程序更改两个字符串之一的值（与哪一个无关）时，更新后的字符串的存储位置将发生变化。这就是写时复制技术的效果。这是输出的第二部分：

```
Change 2nd char
MyStr1 - Addr: 51848300, Len: 5, Ref: 1, Val: Hallo
MyStr2 - Addr: 51837036, Len: 5, Ref: 1, Val: Hello
```

您可以自由地扩展此示例，并使用 StringStatus 函数来探索长字符串在许多其他情况下的行为，当将其作为参数传递，分配给局部变量等时，它们具有多个引用。

6.4 字符串和 Encoding（编码）

如我们所见，Object Pascal 中的字符串类型被映射为 UnicodeUTF-16 格式，每个元素 2 字节，并管理 BMP（基本多语言平面）之外的代码点的代理对。

但是，在许多情况下，您需要保存到文件，从文件加载，通过套接字连接传输或从使用不同表示形式（例如 ANSI 或 UTF-8）的连接接收文本数据。

为了在不同格式（或编码）之间转换文件和内存数据，Object Pascal RTL 具有一个方便的 TEncoding 类，该类在 System.SysUtils 单元中定义，并带有几个继承的类。

❖ Object Pascal RTL 中还有其他一些方便的类，可用于以文本格式读取和写入数据。例如，TStreamReader 和 TStreamWriter 类提供对具有任何编码的文本文件的支持。这些类将在第 18 章中介绍。

尽管我还没有介绍类和继承，但是这组编码类非常易于使用，因为每种编码已经有一个全局对象，可以为您自动创建。

换句话说，这些编码类中的每一个的对象都可以在 TEncoding 类中作为类属性使用：

```
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;
```

- Unicode 编码基于 TUnicodeEncoding 类，该类使用与字符串类型使用的相同的 UTF-16 LE（小尾数）格式。相反，BigEndianUnicode 使用的是不太常见的 Big Endian 表示形式。如果您不熟悉"字节序"，则该术语用于指示构成代码点（或任何其他数据结构）的两个字节的顺序。Little Endian 的最高有效字节在前，Big Endian 的最高有效字节在后。有关更多信息，请参见 en.wikipedia.org/wiki/Endianness。

再次，而不是一般地探讨这些类，在本书的这一点上有些困难，让我们集中在几个实际的例子上。TEncoding 类具有用于将 Unicode 字符串读写到字节数组，执行适当的转换的方法。

为了演示通过 TEncoding 类进行的 UTF 格式转换，而且还使我的示例简单而专注，并避免使用文件系统，在 EncodingsTest 应用程序项目中，我使用一些特定数据在内存中创建了 UTF-8 字符串，并将其转换 通过单个函数调用将其转换为 UTF-16：

```
var
  Utf8string: TBytes;
  Utf16string: string;
begin
  // 处理 Utf8data
  SetLength (Utf8string, 3);
  Utf8string[0] := Ord ('a'); // 单字节 ANSI char < 128
  Utf8string[1] := $c9; // 双字节，倒转拉丁 a
  Utf8string[2] := $90;
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Unicode: ' + Utf16string);
```

输出应为：

```
Unicode: aɐ
```

现在，为了更好地理解转换和表示形式的差异，我添加了以下代码：

```
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);
```

此代码为字符串的两种表示形式 UTF-8（一个字节和两个字节的代码点）和 UTF-16（两个代码点均为 2 个字节）产生带有十进制值的内存转储：

```
Utf8 bytes:
97
201
144
Utf16 bytes:
97
0
80
2
```

请注意，对于 UTF-8，直接字符到字节的转换仅适用于 ANSI-7 字符，即最大值为 127。对于更高级别的 ANSI 字符，没有直接映射，您必须使用特定的编码

（但是在多字节 UTF-8 元素上将失败）。因此，以下两项都会产生错误的输出：

```
// 错误：无法使用 char > 128
Utf8string[0] := Ord ('à');
Utf16string := TEncoding.UTF8.GetString(Utf8string);
Show ('Wrong high ANSI: ' + Utf16string);
//尝试不同的转换
Utf16string := TEncoding.ANSI.GetString(Utf8string);
Show ('Wrong double byte: ' + Utf16string);
// 输出
Wrong high ANSI:
Wrong double byte: àÉ
```

编码类使您可以在两个方向上进行转换，因此在这种情况下，我将从 UTF-16 转换为 UTF-8，对 UTF-8 字符串进行了一些处理（考虑到这种格式的可变长度性质，需要谨慎处理），然后转换回 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);
```

输出为：

```
Initial: This is my nice string with à and Æ
Final: ThIs Is my nIce strIng wIth à and Æ
```

6.5 其他类型的字符串

虽然字符串数据类型是迄今为止最常见且使用最广泛的表示字符串的类型，但是 Object Pascal 桌面编译器具有并且仍然具有多种字符串类型。这些类型中的某些类型也可以在移动应用程序上使用，在移动应用程序上，您也可以直接使用 TBytes 以 1 字节表示形式操作字符串，如上一节中所述。

虽然过去使用 Object Pascal 的开发人员可能有很多基于这些预 Unicode 类型的代码（或直接管理 UTF-8），但是现代应用程序确实需要完全的 Unicode 支持。同样，虽然某些类型（如 UTF8String）在该语言中可用，但它们在 RTL 方面的支持受到限制。建议使用普通和标准 Unicode 字符串。

❖ 关于 Object Pascal 移动编译器中最初缺少诸如 AnsiString 和 UTF8String 之类的本机类型的讨论和批评很多。在柏林的 Delphi 10.1 中，已经正式重新引入了 UTF8String 类型和低级 RawByteString 类型，后来，Delphi 10.4 在移动设备上也启用了所有桌面字符串类型。仍然值得考虑的是，几乎没有其他编程语言具有不止一种本机或内部字符串类型。多个字符串类型更难以掌握，可能会导致不良的副作用（例如使程序变慢的大量自动转换调用），并且维护所

有字符串管理和处理功能的多个版本会花费很多。因此，除了特殊情况外，建议重点是标准字符串类型或 UnicodeString。

6.5.1 UCS4String 类型

一个有趣但很少使用的字符串类型是 UCS4String 类型，可在所有编译器上使用。这只是字符串的 UTF32 表示形式，最多不过是 UTF32Char 元素或 4 个字节字符的数组。如前所述，这种类型的原因是它提供了所有 Unicode 代码点的直接表示。明显的缺点是，这样的字符串占用的内存是 UTF-16 字符串的两倍（后者已经是 ANSI 字符串的两倍）。

尽管此数据类型可以在特定情况下使用，但并不特别适合于一般情况。同样，这种类型不支持写时复制，也没有任何实际的系统功能和处理过程。

❖ 尽管 UCS4String 保证每个 Unicode 代码点一个 UTF32Char，但不能保证每个字素或“可视字符”一个 UTF32Char。

6.5.2 较旧的字符串类型

如前所述，Object Pascal 编译器为某些较旧的传统字符串类型提供支持（并且这些功能在从 Delphi 10.4 开始的所有目标平台上都可用）。这些较旧的字符串类型包括：

- ShortString 类型，它对应于原始的 Pascal 语言字符串类型。这些字符串的上限为 255 个字符。短字符串的每个元素的类型均为 ANSIChar。
- ANSIString 类型，它对应于可变长度的字符串。这些字符串是动态分配的，引用计数，并使用写时复制技术。这些字符串的大小几乎是无限的（它们最多可以存储 20 亿个字符！）。此外，此字符串类型也基于 ANSIChar 类型。即使 ANSI 表示形式特定于 Windows，并且某些特殊字符的处理方式可能因平台而异，也可以在移动编译器上使用。
- 就表示形式而言，WideString 类型类似于 2 字节 Unicode 字符串，基于 Char 类型，但与标准字符串类型不同，它不使用写时复制，并且在内存分配方面效率较低。如果您想知道为什么将其添加到该语言中，则原因是与 Microsoft COM 体系结构中的字符串管理兼容。
- UTF8String 是基于可变字符长度 UTF-8 格式的字符串。正如我提到的，这种类型的运行时库支持很少。
- RawByteString 是没有代码页集的字符数组，在该代码页上，系统从未完成任何字符转换（因此，逻辑上类似于 TBytes 结构，但允许某些当前的直接字符串操作，而字节数组目前缺少这些操作）。此数据类型应在库之外很少使用。
- 一种字符串构造机制，可让您定义与特定 ISO 代码页相关联的 1 字节字符串，该字符串是 Unicode 以前版本的残迹。

同样，所有这些字符串类型都可以在桌面编译器上使用，但仅出于向后兼容的原因而可用。目标是尽可能使用 Unicode，TEncoding 和其他现代字符串管理技术。

第二部分 Object pascal 的 oop

许多现代编程语言都支持某种形式的面向对象编程（OOP）范例。他们中的许多人使用基于类的基于三个基本概念的类：

- 具有公共接口和私有数据结构的类，数据类型，实现封装；这些数据类型的实例通常称为对象；
- 类的可扩展性或继承，即具有在不修改原始特征的情况下扩展具有新功能的数据类型的能力；
- 多态性或后期绑定，即引用了不同对象的能力。类具有统一的接口，并且仍然以其特定类型定义的方式对对象进行操作。
- ❖ 诸如 IO，JavaScript，Lua 和 Rebol 之类的其他语言使用基于原型的面向对象范例，其中可以根据其他对象的创建方式，从其他对象而不是从类中创建对象。它们确实提供了一种继承形式，但是是从另一个对象而不是类继承的，并且可以使用动态类型来实现多态，即使是以完全不同的方式也可以。

您甚至可以在不了解面向对象编程的情况下编写 Object Pascal 应用程序。在创建新表单，添加新组件和处理事件时，IDE 会自动为您准备大多数相关代码。但是了解该语言及其实现的详细信息将帮助您准确地了解系统在做什么，并使您完全掌握该语言。

您还可以在应用程序甚至整个库中创建复杂的体系结构，并包含和扩展开发环境随附的组件。

本书的第二部分重点介绍了核心的面向对象编程（OOP）技术。本书这一部分的目的是教授 OOP 的基本概念，并详细介绍 Object Pascal 如何实现它们，并将其与其他类似的 OOP 语言进行比较。

第二部分摘要

第 7 章：对象
第 8 章：继承
第 9 章：处理异常
第 10 章：属性和事件
第 11 章：接口
第 12 章：处理类
第 13 章：对象和内存

第七章 Objects（对象）

即使您不具备面向对象程序设计（OOP）的详细知识，本章也将介绍每个关键概念。 如果您已经精通 OOP，那么与您可能已经知道的其他语言相比，您可能可以相对较快地阅读材料，并专注于 Object Pascal 语言细节。

Object Pascal 中对 OOP 的支持与 C＃和 Java 等语言有很多相似之处，并且与 C ++和其他静态和强类型语言也有一些相似之处。相反，动态语言倾向于以不同的方式对 OOP 进行解释，因为它们以更加宽松和灵活的方式对待类型系统。

❖ C＃和 Object Pascal 之间在概念上有很多相似之处，这是由于两种语言共享同一位设计师 Anders Hejlsberg。他是 Turbo Pascal 编译器的最初作者，是 Delphi 的 Object Pascal 的第一个版本，后来移居到 Microsoft 并设计了 C＃（以及最近的 JavaScript 衍生 TypeScript）。 您可以在附录 A 中阅读有关 Object Pascal 语言历史的更多信息。

7.1 介绍类和对象

类和对象是 Object Pascal 和其他 OOP 语言中常用的两个术语。但是，由于它们经常被滥用，因此请确保我们从一开始就同意它们的定义：

- 类是用户定义的数据类型，它定义状态（或表示形式）和某些操作（或行为）。换句话说，一个类具有过程或函数形式的一些内部数据和一些方法。一个类通常描述许多相似对象的特征和行为，尽管有专门用于单个对象的类。
- 对象是类的实例，是该类定义的数据类型的变量。对象是实际的实体。程序运行时，对象会占用一些内存以用于其内部表示。

对象和类之间的关系与其他任何变量及其数据类型之间的关系相同。仅在这种情况下，变量具有特殊名称。

✓ OOP 术语可以追溯到采用该模型的前几种语言，例如 Smalltalk。但是，大多数原始术语后来都被弃用，而倾向于使用过程语言中的术语。 因此，尽管仍然经常使用诸如类和对象之类的术语，但是与最初向接收者（对象）发送消息的术语相比，您通常会更频繁地听到术语“调用方法”。关于 OOP 术语及其随时间的演变的完整而详细的指南可能很有趣，但在本书中会占用太多空间。

7.1.1 类的定义

在 Object Pascal 中，可以使用以下语法来定义新的类数据类型（TDate），以及一些本地数据字段（月，日，年）和一些方法（SetValue，LeapYear）：

```
type
  TDate = class
    FMonth, FDay, FYear: Integer;
    procedure SetValue (M, D, Y: Integer);
    function LeapYear: Boolean;
  end;
```

❖ 我们已经看到了类似的记录结构，就定义而言，它们与类非常相似。内存管理和其他领域存在差异，如本章稍后所述。但是，从历史上看，在 Object Pascal 中，此语法首先用于类，然后又移植回记录。

Object Pascal 中的约定是使用字母 T 作为您编写的每个类的名称的前缀，就

像其他任何类型一样（实际上 T 代表 Type）。这只是一个约定，对编译器而言，T 只是一个字母，与其他任何字母一样，但是它是如此普遍，以至于遵循它会使您的代码更容易被其他程序员理解。

与其他语言不同，Object Pascal 中的类定义不包括方法的实际实现（或定义），而仅包括其签名（或声明）。这使类代码更加紧凑，并且可读性大大提高。

✧ 尽管看起来方法的实际实现似乎比较耗时，但编辑器允许您使用 Shift + Up 和 Shift + Down 键的组合从方法声明导航至其实现，反之亦然。此外，在编写类定义后，可以使用类完成（在光标位于类定义内时按 Ctrl + C 键），让编辑器生成方法定义的框架。

还请记住，除了编写类的定义（及其字段和方法）之外，您还可以编写声明。它只有类名，如：

```
type
  TMyDate = class;
```

声明的原因在于您可能需要两个相互引用的类。在 Object Pascal 中给定，在定义符号之前不能使用符号，要引用尚未定义的类，则需要声明。我写了以下代码片段只是为了向您展示语法，而没有任何意义：

```
type
  THusband = class;

  TWife = class
    FHusband: THusband;
  end;

  THusband = class
    FWife: TWife;
  end;
```

在实际代码中，您会遇到类似的交叉引用，这就是为什么记住此语法很重要的原因。注意，就像方法一样，稍后必须在同一单元中完全定义在单元中声明的类。

7.1.2 其他 OOP 语言的类

相比之下，这是使用 C＃和 Java（在这种简化情况下是相同的）编写的 TDate 类，它使用一组更合适的命名规则，但省略了方法的代码：

```
// C#和 Java 语言
class Date
{
  int month;
  int day;
  int year;
  void setValue (int m, int d, int y)
  {
    // code
  }
  bool leapYear()
  {
    // code
  }
}
```

在 Java 和 C＃中，方法的代码位于类定义中，而在 Object Pascal 中，在类中

声明的方法应在包含该类定义的同一单元的实现部分中完全定义。 换句话说，在 Object Pascal 中，一个类总是完全在一个单元中定义的（当然，一个单元可以包含多个类）。相比之下，虽然在 C ++中像在 Object Pascal 中那样分别实现方法，但是包含类定义的头文件与使用该方法的代码的实现文件没有严格的对应关系。相应的 C ++类如下所示：

```
// C++  语言
class Date
{
   int month;
   int day;
   int year;
   void setValue (int m, int d, int y);
   BOOL leapYear();
}
```

7.1.3 类方法

与记录类似，在定义方法的代码时，需要使用类名作为前缀和点表示法来指示方法所属的类（在本示例中为 TDate 类），如以下代码所示：

```
procedure TDate.SetValue(M, D, Y: Integer);
begin
   FMonth := M;
   FDay := D;
   FYear := Y;
end;

function TDate.LeapYear: Boolean;
begin
   //  在 SysUtils.pas 中调用 IsLeapYear
   Result := IsLeapYear (FYear);
end;
```

与大多数其他将方法定义为函数的 OOP 语言不同，Object Pascal 取决于过程的返回值的存在，带来了过程和函数之间的核心区别。在 C ++中不是这种情况，在 C ++中，单独定义的方法实现如下所示：

```
// C++ method
void Date::setValue(int m, int d, int y)
{
   month = m;
   day = d;
   year = y;
};
```

7.1.4 创建一个对象

与其他流行语言进行比较之后，让我们回到 Object Pascal 来了解如何使用类。定义完类后，我们可以创建此类型的对象，并按以下代码片段所示使用它（如本节中的所有代码一样，从 Dates1 应用程序项目中提取）：

```
var
   ADay: TDate;
begin
   //  创建
   ADay := TDate.Create;
```

```
// 使用
ADay.SetValue (1, 1, 2020);
if ADay.LeapYear then
  Show ('Leap year: ' + ADay.Year.ToString);
```

所使用的表示法并不稀奇，但功能强大。我们可以编写一个复杂的函数（例如 LeapYear），然后为每个 TDate 对象访问其值，就好像它是原始数据类型一样。请注意，ADay.LeapYear 是类似于 ADay.Year 的表达式，尽管第一个是函数调用，第二个是直接数据访问。

正如我们将在第 10 章中看到的那样，Object Pascal 用于访问属性的符号再次相同。

❖ 在大多数基于 C 语言语法的编程语言中，不带参数的方法的调用都需要括号，例如 ADay.LeapYear()。 此语法在 Object Pascal 中也是合法的，但很少使用。没有参数的方法通常在不带括号的情况下调用。这与许多语言（其中对不带括号的函数或方法的引用返回函数地址）有很大不同。正如我们在第 4 章的“过程类型”一节中所看到的，Object Pascal 使用相同的表示法来调用函数或读取其地址，具体取决于表达式的上下文。

上面的代码片段的输出相当简单：

```
Leap year: 2020
```

再一次，让我将对象创建与用其他编程语言编写的类似代码进行比较：

```
// C# 和 Java 语言 (对象参考模型)
Date aDay = new Date();
// C++ language (两种替代风格)
Date aDay; // 本地分配
Date* aDay = new Date(); // "manual（手册）" reference（参考）
```

7.2 对象参考模型

在某些 C++之类的 OOP 语言中，声明一个类类型的变量会创建该类的实例（或多或少与 Object Pascal 中的记录所发生的情况类似）。

本地对象的内存从堆栈中获取，并在函数终止时释放。但是，在大多数情况下，必须显式使用指针和引用才能在管理对象的生存期方面具有更大的灵活性，从而增加了很多额外的复杂性。

相反，ObjectPascal 语言是基于对象引用模型的，与 Java 或 C＃完全一样。这个想法是，一个类类型的每个变量不包含其数据的对象的实际值（例如，用于存储日，月和年）。

而是，它仅包含一个引用或指针，以指示实际对象数据的存储位置。

❖ 在我看来，采用对象引用模型是编译器团队在该语言的早期做出的最佳设计决策之一，当时该模型在编程语言中并不那么普遍（事实上，当时 Java 还不存在）。可用且 C＃不存在）。

这就是为什么在这些语言中，您需要显式创建一个对象并将其分配给变量，因为对象不会自动初始化。换句话说，当您宣告变数时，您不会在记忆体中建立物件，而只保留记忆体位置以供参考。必须至少为您定义的类的对象手动且显式创建对象实例。（不过，在 Object Pascal 中，运行时库会自动构建您放置在表单上的组件实例）。

在 Object Pascal 中，要创建对象的实例，我们可以调用其特殊的 Create 方法，该方法是构造函数或由类本身定义的另一个自定义构造函数。再次是下面的代

码：

```
ADay := TDate.Create;
```

如您所见，构造函数应用于类（类型），而不应用于对象（变量）。那是因为您要让类创建其类型的新实例，并且结果是通常分配给变量的新对象。

Create 方法从何而来？它是类 TObject 的构造函数，所有其他类都从该类继承而来（下一章中讨论的主题）。不过，将自定义构造函数添加到类中是很常见的，正如我们将在本章后面看到的那样。

7.2.1 处理 Objects

在使用对象引用模型的语言中，您需要一种在使用对象之前创建对象的方法，并且还需要一种在不再需要时释放其占用的内存的方法。如果不进行处理，最终将用不再需要的对象填充内存，从而导致称为内存泄漏的问题。为了解决此问题，基于虚拟执行环境（或虚拟机）的 C＃和 Java 等语言将采用垃圾回收。尽管这使开发人员的工作变得更轻松，但是这种方法会遇到一些与性能相关的复杂问题，这在解释 Object Pascal 时并没有真正的意义。因此，尽管问题很有趣，但我不想在这里深入研究它们。

在 Object Pascal 中，通常可以通过调用对象的特殊 Free 方法（同样是 TObject 的方法，每个类都可用）来释放对象的内存。Free 在调用其析构函数（可以具有特殊的清除代码）后，将其从内存中删除。因此，您可以按照以下步骤完成上面的代码片段：

```
var
  ADay: TDate;
begin
  // 创建
  ADay := Tdate.Create;
  // 使用 (code missing)
  // 释放内存
  ADay.Free;
end;
```

虽然这是标准方法，但是组件库添加了诸如对象所有权之类的概念，以显著减轻手动内存管理的影响，这使其成为一个相对简单的问题。

❖ 稍后我们将看到，当使用引用对象的接口时，编译器采用自动引用计数（ARC）内存管理的形式。几年来，它也被用于 Delphi 移动编译器中的常规类类型变量。从悉尼 10.4 版开始，内存管理模型统一采用所有目标平台的经典桌面 Delphi 内存管理。

您需要了解更多有关内存管理的知识，但是鉴于这是一个相当重要的主题，而不是一个简单的主题，因此我决定在此处仅作简短介绍，并针对该主题专门整章介绍，即第 13 章。

在该章中，我将详细介绍可以使用的各种技术。

7.2.2 什么是“Nil”?

如前所述，变量可以引用给定类的对象。但是它可能尚未初始化，或者它用来引用的对象可能不再可用。

在这里可以使用 nil。这是一个常量值，指示未将该变量分配给任何对象（或

将其分配给 0 存储位置）。其他编程语言使用符号 null 表示相同的概念。

当类类型的变量没有值时，可以通过以下方式对其进行初始化：

```
ADay := nil;
```

要检查是否已为对象分配了变量，可以编写以下两个表达式之一：

```
if ADay <> nil then ...
if Assigned (ADay) then ...
```

不要错误地将 nil 分配给对象以将其从内存中删除。

将对象引用设置为 nil 并释放它是两个不同的操作。因此，您经常需要释放一个对象并将其引用设置为 nil，或者调用一次执行两项操作的专用过程，称为 FreeAndNil。同样，更多的信息和一些实际的演示将在第 13 章中介绍，重点是内存管理。

7.2.3 内存中的记录与类

如前所述，记录和对象之间的主要区别之一与它们的内存模型有关。记录类型变量使用本地内存，默认情况下按值将它们作为参数传递给函数，并且它们在分配中具有“按值复制”行为。这与在动态内存堆上分配的类类型变量形成对比，这些类类型变量通过引用传递，并且在分配时具有“按引用复制”行为（因此，将引用复制到内存中的同一对象，而不是实际数据）。

❖ 这种不同的内存管理的结果是记录缺少继承和多态性，这是我们将在下一章中重点介绍的两个功能。

● 私有访问说明符表示在声明该类的单元（源代码文件）之外不可访问的类的字段和方法。

例如，当您在堆栈上声明一个记录变量时，您可以立即开始使用它，而不必调用其构造函数（除非它们是自定义托管记录）。这意味着记录变量在内存管理器上比常规对象更精简和高效，因为它们不参与动态内存的管理。这些是为小型和简单数据结构使用记录而不是对象的主要原因。

关于记录和对象作为参数传递方式的差异，请考虑默认设置是对表示记录（包括其所有数据）的存储块或对对象的引用（当数据为未复制）。当然，您可以使用 var 或 const 记录参数来修改传递记录类型参数的默认行为，以避免任何复制。

7.3 Private（私有）, Protected（受保护）,和 Public 公共

● strict private 严格的私有访问说明符表示在类的任何方法(包括同一单元中其他类的方法）之外不可访问的字段和方法。这与大多数其他 OOP 语言中的 private 关键字的行为匹配。

一个类可以具有任意数量的数据字段和任意数量的方法。但是，对于一种好的面向对象的方法，应该在使用它的类内部将数据隐藏或封装。例如，当您访问日期时，单独更改日期的值是没有意义的。实际上，更改日期的值可能会导致无效的日期，例如 2 月 30 日。使用方法访问对象的内部表示会限制产生错误情况的风险，因为这些方法可以检查日期是否有效，如果日期无效则拒绝修改。

正确的封装特别重要，因为它使类编写者可以自由地在将来的版本中修改内部表示。

封装的概念非常简单：只需将一个类看作是“黑匣子”，其中包含一小部分可见的部分。可见的部分称为类接口，它允许程序的其他部分访问和使用该类的对象。但是，当您使用这些对象时，它们的大多数代码都是隐藏的。您很少知道对象具有哪些内部数据，并且通常无法直接访问数据。而是使用这些方法来访问对象的数据或对其执行操作。

使用私有成员和受保护成员进行封装是针对经典编程目标（称为信息隐藏）的面向对象解决方案。

Object Pascal 具有五个基本访问（或可见性）说明符：private（私有的），protected（受保护的）和 public（公共的）的。第 6 个，published（发布的）将在第 10 章中讨论。这是五个基本的：

- public（公共）访问说明符表示可以从程序的任何其他部分以及在定义它们的单元中自由访问的字段和方法。
- Protected（受保护）和 strict private（严格受保护）的访问说明符用于指示可见性有限的方法和字段。只有当前类及其派生类（或子类）可以访问受保护的元素，除非它们属于同一类，或者在任何情况下都取决于 strict 修饰符。我们将在下一章的“受保护的字段和封装”部分中再次讨论此关键字。

通常，类的字段应该是 private（私有的）或 strict private（严格私有的）。该方法通常是 piblic（公开的）。然而，这并非总是如此。如果仅在内部需要执行某些部分操作，则方法可以是私有的或受保护的。

如果可以肯定地确定字段的类型定义不会改变，并且您可能希望直接在派生类中操作它们（如下一章中所述），则可以 protected（保护）字段，尽管很少建议这样做。

通常，您应该始终避开公共字段，并通常公开一些使用属性访问数据的直接方法，我们将在第 10 章中详细介绍。属性是对其他 OOP 语言封装机制的扩展，在 Object Pascal 中非常重要。

如前所述，私有访问说明符仅限制单元外部的代码访问该单元中声明的类的某些成员。这意味着，如果两个类位于同一单元中，则除非将它们标记为严格私有，否则对其私有字段没有任何保护，这通常是个好主意。

❖ C ++语言具有“朋友类”的概念，即允许访问其他类私有数据的类。遵循此术语，我们可以说在 Object Pascal 中，同一单元中的所有类都将自动视为朋友类。

7.3.1 private（私有的）数据的例子

作为使用这些访问说明符实现封装的示例，请考虑以下新版本的 TDate 类：

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (M, D, Y: Integer);
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

在此版本中，现在将字段声明为私有字段，并且有一些新方法。第一个是 GetText，它是一个返回带有日期的字符串的函数。您可能会考虑添加其他功能，

例如 GetDay，GetMonth 和 GetYear，它们仅返回相应的私有数据，但是并不总是需要类似的直接数据访问功能。为每个字段提供访问功能可能会减少封装，削弱抽象，并使以后更难修改类的内部实现。仅当访问函数是要实现的类的逻辑接口的一部分时，才应提供访问功能，而不是因为存在匹配的字段。

第二种新方法是增加过程，该过程将日期增加一天。这绝非易事，因为您需要考虑各个月份以及 leap（闰年）和 non-leap（非闰年）的不同长度。为了使编写代码更容易，我要做的就是更改类的内部实现，以将 Object Pascal TDateTime 类型用于内部实现。因此，实际的类将更改为您可以在 Dates2 应用程序项目中找到的以下代码：

```
type
   TDate = class
   private
      FDate: TDateTime;
   public
      procedure SetValue (M, D, Y: Integer);
      function LeapYear: Boolean;
      function GetText: string;
      procedure Increase;
   end;
```

请注意，由于唯一的更改是在类的私有部分中，因此您无需修改任何使用它的现有程序。这就是封装的优势！

❖ 在该类的新版本中，(仅)字段具有以字母“F”开头的标识符。这是 Object Pascal 中相当普遍的约定，我将在本书中普遍使用。

在本节结束时，让我通过列出类方法的源代码来结束对项目的描述，这些方法依赖于一些系统函数将日期映射到内部结构，反之亦然：

```
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
   // IsLeapYear 在 SysUtils 中，YearOf 在 DateUtils 中
   Result := IsLeapYear (YearOf (FDate));
end;
```

还请注意，使用该类的代码如何不再引用 Year 值，而只能返回其方法所允许的有关 date 对象的信息：

```
var
   ADay: TDate;
begin
   // 创建
   ADay := TDate.Create;
   // 使用
```

```
ADay.SetValue (1, 1, 2020);
ADay.Increase;
if ADay.LeapYear then
   Show ('Leap year: ' + ADay.GetText);
// 释放内存 (用于非 ARC 平台)
ADay.Free;
```

输出与以前没有太大不同：

```
Leap year: 1/2/2020
```

7.3.2 Encapsulation（封装）和 Forms（表单）

封装的关键思想之一是减少程序使用的全局变量的数量。可以从程序的每个部分访问全局变量。因此，全局变量的更改会影响整个程序。另一方面，当您更改类的字段的表示形式时，只需要更改引用给定字段的该类的某些方法的代码即可，而无需进行其他操作。因此，我们可以说信息隐藏是指封装变化。

让我通过一个实际的例子来澄清这个想法。当您的程序具有多个 Forms，可以通过在 Form 单元的接口部分中将其声明为全局变量来使某些数据可用于每个 Form：

```
var
   Form1: TForm1;
   NClicks: Integer;
```

这可行，但是有两个问题。首先，数据（NClicks）未连接到表单的特定实例，而是连接到整个程序。如果您创建两个相同类型的表单，它们将共享数据。如果您希望每个相同类型的表单都拥有自己的数据副本，则唯一的解决方案是将其添加到表单类中：

```
type
   TForm1 = class(TForm)
   public
      FNClicks: Integer;
   end;
```

第二个问题是，如果您将数据定义为全局变量或表单的公共字段，则将来将无法修改其实现而不影响使用该数据的代码。例如，如果您只需要从其他 Form 读取当前值，则可以将数据声明为私有数据，并提供一种读取值的方法：

```
type
   TForm1 = class(TForm)
   // 这里的组件和事件处理程序
   public
      function GetClicks: Integer;
   private
      FNClicks: Integer;
   end;

function TForm1.GetClicks: Integer;
begin
   Result := FNClicks;
end;
```

更好的解决方案是在表单中添加一个属性，我们将在第 10 章中看到。

您可以通过打开 ClicksCount 应用程序项目来试验此代码。

简而言之，该项目的 Form 在顶部具有两个按钮和一个标签，其中大部分表面为空，以便用户单击（或点击）在其上。在这种情况下，计数会增加，并且标

签会使用新值进行更新：

```
procedure TForm1.FormMouseDown(Sender: TObject; Button:
   TMouseButton; Shift: TShiftState; X, Y: Single);
begin
   Inc (FNClicks);
   Label1.Text := FNClicks.ToString;
end;
```

您可以在图 7.1 中看到正在运行的应用程序。该项目的窗体还具有两个按钮，一个用于创建相同类型的新窗体，另一个用于关闭该窗体（因此您可以将焦点放回到上一个窗体上）。

这样做是为了强调相同表单类型的不同实例如何各自具有自己的点击次数。这是两种方法的代码：

```
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 应用程序项目的表单，显示表单上的点击或点击数（使用私有表单数据进行跟踪）

7.4 Self 关键字

我们已经看到方法与过程和函数非常相似。真正的区别是方法具有额外的隐式参数。这是对当前对象的引用，该对象是应用该方法的对象。在方法内，您可以使用 Self 关键字引用此参数（当前对象）。

当您创建同一类的多个对象时，需要使用此额外的隐藏参数，以便每次将方法应用于一个对象时，该方法将仅对特定对象的数据进行操作，而不会影响该对象的其他同类对象。

❖ 在讨论记录时，我们已经在第 5 章中看到了 Self 关键字的作用。这个概念及其实现非常相似。同样，从历史上来说，Self 最初是为类引入的，后来又扩展到记录，那时方法也被添加到该数据结构中。

例如，在前面列出的 TDate 类的 SetValue 方法中，我们仅使用 Month，Year 和 Day 来引用当前对象的字段，您可以将其表示为：

```
Self.FMonth := M;
Self.FDay := D;
```

实际上，这就是 Object Pascal 编译器翻译代码的方式，而不是您应该编写的方式。Self 关键字是编译器使用的基本语言结构，但有时程序员会使用它来解决

名称冲突并提高代码的可读性。

❖ C ++，Java，C＃和 JavaScript 语言基于关键字 this 具有类似的功能。但是在 JavaScript 中，与 C ++，C＃和 Java 不同，在此方法中使用此方法来引用对象字段是强制性的。

您真正需要了解的有关 Self 的所有信息是，对方法的调用的技术实现与对通用子例程的调用的技术实现不同。方法具有额外的隐藏参数，Self。因为所有这些都是在幕后发生的，所以您此时无需了解 Self 的工作方式。

要知道的第二件事是，您可以显式使用 Self 来整体引用当前对象，例如，将当前对象作为参数传递给另一个函数。

7.4.1 动态创建组件

作为我刚刚提到的示例，当您需要在其中一种方法中显式引用当前表单时，经常使用 Self 关键字。

一个典型的示例是在运行时创建组件，您必须在其中将组件的所有者传递给其 Create 构造函数，并将相同的值分配给其 Parent 属性。在这两种情况下，都必须提供当前表单对象作为参数或值，而最好的方法是使用 Self 关键字。

组件的所有权指示两个对象之间的生存期和内存管理关系。当组件的所有者被释放时，该组件也将被释放。父级关系是指在其表面中承载子控件的可视化控件。

为了演示这种代码，我编写了 CreateComps 应用程序项目。

此应用程序具有一个简单的 Form，没有任何组件，并且具有 OnMouseDown 事件的处理程序，该事件还接收鼠标单击的位置作为其参数。我需要此信息来在该位置创建按钮组件。

❖ 事件处理程序是第 10 章介绍的一种特殊方法，它是本书中已使用的按钮的 OnClick 事件处理程序家族的一部分。

这是该方法的代码：

```
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;
```

注意，您可能需要将 StdCtrls 单元添加到 uses 语句中，以编译此事件处理程序。

此代码的作用是在鼠标单击位置创建按钮，并在标题上显示确切的位置，如图 7.2 所示。（对于此项目，我更清楚地禁用了 FMXMobile Preview 来显示本机样式的 Windows 按钮。）在上面的代码中，尤其要注意使用 Self 关键字作为 Create 方法的参数。并作为 Parent 属性的值。

在编写类似您所看到的代码的过程时，您可能会倾向于使用 Form1 变量而

不是 Self。在此特定示例中，所做的更改不会产生任何实际变化（尽管这不是良好的编码习惯），但是，如果表单有多个实例，则使用 Form1 确实会出错。

实际上，如果 Form1 变量引用正在创建的该类型的表单（通常是第一个表单），并且如果您创建了两个相同表单类型的实例，则通过单击以下任何表单，新按钮将始终显示在第一个表单中。它的所有者和父级将是 Form1，而不是用户单击的表单。

通常，编写一种需要使用当前对象时引用同一类的特定实例的方法确实是一种糟糕的 OOP 编码风格。

图 7.2：在移动设备上 CreateComps 应用程序项目示例的输出

7.5 Constructors（构造）

在上面的代码中，要创建类的对象（或为对象分配内存），我调用了 Create 方法。这是一个构造函数，是一种特殊的方法，您可以将其应用于一个类，以为该类的新实例分配内存：

```
ADay := TDate.Create;
```

实例由构造函数返回，并且可以分配给变量以存储对象并在以后使用。创建对象时，将初始化其内存。新实例的所有数据都将设置为零（或者为 nil，或者为空字符串，或者给定数据类型的正确“默认”值）。

如果您希望实例数据以非零值开始（特别是当零值作为默认值意义不大时），则需要编写一个自定义构造函数来实现。新的构造函数可以称为 Create，也可以具有任何其他名称。

决定其角色的不是名称，而是 constructor（构造器）关键字的使用。

❖ 换句话说，Object Pascal 支持命名 constructor（构造）函数，而在许多 OOP 语言中，构造函数必须以类本身命名。使用命名 constructor（构造）函数，可以有多个具有相同参数的构造函数（除了重载 Create 符号-overloading 下一节将介绍重载）。该语言的另一个非常特殊的功能（在 OOP 语言中非常独特）是，构造函数也可以是虚拟的。在下一章介绍虚拟方法的概念之后，我将在本书后面的示例中介绍该非常好功能的后果。

向类中添加自定义构造函数的主要原因是初始化其数据。如果创建对象时未对其进行初始化，则稍后调用方法可能会导致异常行为，甚至会导致运行时错误。而不是等待这些错误出现，您应该首先使用预防技术来避免它们。一种这样的技

术是一致使用构造函数来初始化对象的数据。例如，创建对象后，我们必须调用 TDate 类的 SetValue 过程。或者，我们可以提供一个定制的构造函数，该构造函数创建对象并为其提供初始值：

```
constructor TDate.Create;
begin
  FDate := Today;
end;

constructor TDate.CreateFromValues (M, D, Y: Integer);
begin
  FDate := SetValue (M, D, Y);
end;
```

您可以按以下方式使用这些构造函数，就像在 Date3 应用程序项目中所做的那样，在附加到两个单独按钮的代码中：

```
ADay1 := TDate.Create;
ADay2 := TDate.CreateFromValues (12, 25, 2015);
```

尽管通常可以为构造函数使用任何名称，但是请记住，如果您使用 Create 以外的名称，则基本 TObject 类的 Create 构造函数仍然可用。如果您正在开发和分发代码供他人使用，则调用此默认 Create 构造函数的程序员可能会绕过您提供的初始化代码。通过使用一些参数定义 Create 构造函数（或不使用任何参数，如上例所示），可以将默认定义替换为新的定义并强制使用。

与类可以具有自定义构造函数的方式相同，它可以具有自定义析构函数，该方法用 destructor 关键字声明，并且始终称为 Destroy。此析构函数方法可以在销毁对象之前执行一些资源清除，但是在许多情况下，不需要自定义析构函数。

正如构造函数调用为对象分配内存一样，析构函数调用则释放内存。实际上，自定义析构函数仅对于在其构造函数中或生命周期中获取资源的对象（例如另一个对象）才需要。

与默认的 Create 构造函数不同，默认的 Destroy 析构函数是虚拟的，强烈建议开发人员重写此虚拟析构函数（下一章将介绍虚拟方法）。

这是因为不是直接调用析构函数来释放对象，而是一种好的通用 Object Pascal 编程实践，即调用 TObject 类的特殊 Free 方法，后者仅在对象存在时才调用 Destroy，即不为零。因此，您用不同的名称定义了一个析构函数，它将不会被 Free 调用。

同样，当我们在第 13 章中专注于内存管理时，将更多地讨论该主题。

如下一章所述，销毁是一种虚拟方法。您可以在继承的类中将其基本定义替换为新的基本定义，并用 override 关键字标记它。顺便说一句，拥有一个调用虚拟方法的静态方法是一种非常常见的编程样式，称为模板模式。在析构函数中，通常只应编写资源清除代码。尽量避免执行更复杂的操作（可能会引发异常或花费大量时间），以避免对象清理方面的麻烦，并且由于在程序终止时调用了许多析构函数，因此您希望保持快速。

7.5.1 使用构造函数和析构函数管理本地类数据

即使我在本书后面将介绍更复杂的场景，在这里我也想向您展示一个使用构造函数和析构函数的简单资源保护案例。这是使用析构函数的最常见方案。假设您有一个具有以下结构的类（也是 Date3 应用程序项目的一部分）：

```
type
```

```
TPerson = class
private
   FName: string;
   FBirthDate: TDate;
public
   constructor Create (Name: string);
   destructor Destroy; override;
   // 一些实际方法
   function Info: string;
end;
```

此类引用了另一个内部对象 FBirthDate。当创建 TPerson 类的实例时，还应该创建此内部（或子对象）对象，并且在销毁该实例时，也应该丢弃该子对象。这是如何编写构造函数和重写的析构函数以及内部方法的代码，这些代码始终可以认为内部对象存在：

```
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;
```

❖ 要了解用于定义析构函数的 override 关键字及其定义内的 Inherited 关键字，您必须等到下一章。现在足以说第一个用来表示该类具有替换基 Destroy 析构函数的新定义，而后者则用于调用该基类析构函数。还请注意，方法声明中使用了覆盖，但方法实现代码中未使用。

现在，可以在以下情况下使用外部类的对象，并且在创建 TPerson 对象时可以正确创建内部对象，并在销毁 TPerson 时及时销毁内部对象：

```
var
   Person: TPerson;
begin
   Person := TPerson.Create ('John');
   // 使用类及其内部对象
   Show (Person.Info);
   Person.Free;
end;
```

同样，您可以在 Dates3 应用程序项目中找到此代码。

7.5.2 Overload（重载）方法和构造函数

Object Pascal 支持重载的函数和方法：如果参数不同，则可以有多个具有相同名称的方法。我们已经看到相同的规则适用于方法的全局函数和过程的重载是如何工作的。通过检查参数，编译器可以确定您要调用的方法的版本。

同样，重载有两个基本规则：

- 该方法的每个版本后均必须带有 Overload（重载）关键字。
- 区别必须在参数的数量或类型或两者上。相反，返回类型不能用于区分两种方法。

如果可以将重载应用于类的所有方法，则此功能与构造函数特别相关，因为我们可以有多个构造函数并将它们全部称为 Create，这使它们易于记忆。

✓ 从历史上看，重载是专门为 C ++添加的，以允许使用多个构造函数，因为它们必须具有相同的名称（类的名称）。在 Object Pascal 中，仅因为多个构造函数可以具有不同的特定名称，否则就可以认为此功能是不必要的，但是无论如何它也已添加到该语言中，因为它在许多其他场景中也很有用。

作为重载的一个示例，我在 TDate 类中添加了两个不同版本的 SetValue 方法：

```
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;
```

完成这一简单步骤之后，我在类中添加了两个单独的 Create 构造函数，一个没有参数，它隐藏了默认构造函数，一个带有初始化值。没有参数的构造函数使用今天的日期作为默认值：

```
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;
end;
```

有了这两个构造函数，就可以通过两种不同的方式定义新的 TDate 对象：

```
var
  Day1, Day2: TDate;
begin
  Day1 := TDate.Create (2020, 12, 25);
  Day2 := TDate.Create; // today
```

此代码是 Dates4 应用程序项目的一部分。

7.5.3 完整的 TDate 类

在本章中，我向您展示了 TDate 类不同版本的源代码。第一个版本基于三个整数来存储年，月和日。第二种版本使用 RTL 提供的 TDateTime 类型的字段。这是定义 TDate 类的单元的完整接口部分：

```
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;
```

新方法的目的是增加和减少（它们的参数具有默认值），这很容易理解。 如果不带参数调用，它们会将日期值更改为第二天或前一天。如果 NumberOfDays 参数是该调用的一部分，则它们会添加或减去该数字：

```
procedure TDate.Increase (NumberOfDays: Integer = 1);
begin
  FDate := FDate + NumberOfDays;
end;
```

GetText 方法使用 DateToStr 函数进行转换，以格式化日期返回一个字符串：

```
function TDate.GetText: string;
begin
  GetText := DateToStr (FDate);
end;
```

我们已经在上一节中看到了大多数方法，因此我将不提供完整的清单。您可以在为测试该类而编写的 ViewDate 应用程序项目的代码中找到该代码。表单比书中的其他表单稍微复杂一点，并且它的标题显示日期和六个按钮，可用于修改对象的值。您可以在运行时看到 ViewDate 应用程序项目的主要形式，如图 7.3 所示。为了使 Label 组件看起来更好，我给了它一个大字体，使其与表单一样宽，将其 Alignment 属性设置为 taCenter，并将其 AutoSize 属性设置为 False。

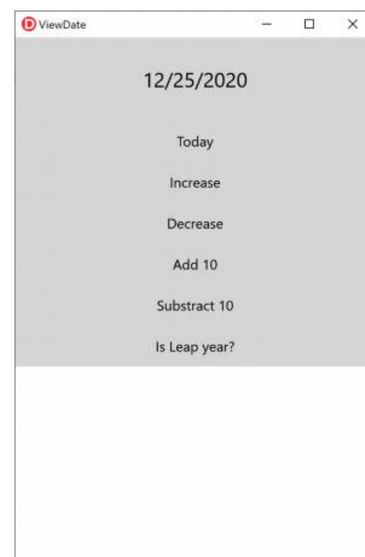

图 7.3：启动时 ViewDate 应用程序的输出

该程序的启动代码在表单的 OnCreate 事件处理程序中。在相应的方法中，我们创建 TDate 类的实例，初始化该对象，然后在标签的 Text 中显示其文本描述，如图 7.3 所示。

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
```

```
    ADay := TDate.Create;
    LabelDate.Text := ADay.GetText;
  end;
```

ADay 是 TDateForm 形式的类的私有字段。顺便说一句，当您将表单的 Name 属性更改为 DateForm 时，开发环境会自动选择类的名称。

特定的日期对象是在创建表单时创建的（建立了我们之前在 person 类及其日期子对象之间看到的相同关系），然后将其与表单一起销毁：

```
  procedure TDateForm.FormDestroy(Sender: TObject);
  begin
    ADay.Free;
  end;
```

当用户单击六个按钮之一时，我们需要将相应的方法应用于 ADay 对象，然后在标签中显示日期的新值：

```
  procedure TDateForm.BtnTodayClick(Sender: TObject);
  begin
    ADay.SetValue (Today);
    LabelDate.Text := ADay.GetText;
  end;
```

编写最后一个方法的另一种方法是销毁当前对象并创建一个新对象：

```
  procedure TDateForm.BtnTodayClick(Sender: TObject);
  begin
    ADay.Free;
    ADay := TDate.Create;
    LabelDate.Text := ADay.GetText;
  end;
```

在这种特殊情况下，这不是一个很好的方法（因为创建新对象并销毁现有对象会占用大量时间，而我们所需要做的只是更改对象的值），但是它允许我向您展示两种 Object Pascal 技术。首先要注意的是，在分配新对象之前，我们先销毁了前一个对象。实际上，赋值操作将替换引用，从而将对象保留在内存中（即使没有指针引用该对象）。当您将一个对象分配给另一个对象时，编译器仅将内存中该对象的引用复制到新的对象引用中。

另一个问题是如何将数据从一个对象复制到另一个对象。这种情况非常简单，因为只有一个字段和一种初始化它的方法。通常，如果要更改现有对象中的数据，则必须复制每个字段，或提供一种特定的方法来复制所有内部数据。一些类具有一个 Assign 方法，该方法执行此深度复制操作。

- ❖ 更准确地说，在运行时库中，所有从 TPersistent 继承的类都具有 Assign 方法，但是大多数从 TComponent 继承的类均未实现该方法，因此在调用该方法时会引发异常。原因在于运行时库支持的流传输机制以及 TPersistent 类型属性的支持，但这太复杂了，无法在本书中进行探讨。

7.6 嵌套类和嵌套常量

Object Pascal 允许您在单元的接口部分中声明新类，从而允许程序的其他单元引用它们，或者在实现部分中声明它们，这些部分只能从同一单元的其他类的方法或全局例程中进行访问。在类定义之后在该单元中实现。

最近的增加是可以在另一个类中声明一个类（或任何其他数据类型）。与该类的任何其他成员一样，嵌套类型的可见性也可能受到限制（例如，私有的或受保护的）。嵌套类型的相关示例包括同一类和实现支持类使用的枚举。

相关的语法允许您定义一个嵌套常量，一个与类关联的常量值（如果是私有的，则只能在内部使用；如果是公共的，则可以在程序的其余部分使用）。例如，请考虑以下嵌套类的声明（从 NestedTypes 应用程序项目的 NestedClass 单元中提取）：

```
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.FMsg := '嗨';
  Ins.InsideHello;
  Show ('常数为： ' + IntToStr (Foo));
  Ins.Free;
end;

procedure TOne.TInside.InsideHello;
begin
  FMsg := '新消息';
  Show ('内部调用');
  if not Assigned (InsIns) then
    InsIns := TInsideInside.Create;
  InsIns.Two;
end;

procedure TOne.TInside.TInsideInside.Two;
begin
  Show ('这是一个嵌套/嵌套类的方法');
end;
```

嵌套类可以在类内部直接使用（如清单所示），也可以在类外部使用（如果在公共部分中声明了），但具有完全限定的名称 TOne.TInside。 在嵌套类的方法定义中也使用类的全名，在本例中为 TOne.TInside。声明嵌套类之后，托管类可以立即具有嵌套类类型的字段（如您在 NestedClass 应用程序项目的代码中所见）。

具有嵌套类的类的用法如下：

```
var
  One: TOne;
```

```
begin
  One := TOne.Create;
  One.Hello;
  One.Free;
```

这将产生以下输出：

```
内部调用
这是一个嵌套/嵌套类的方法
常量为：12
```

通过使用 Object Pascal 语言中的嵌套类,您将如何受益？该概念在 Java 中通常用于实现事件处理程序委托，并且在 C# 中有意义，在 C# 中您无法将类隐藏在单元内部。在 Object Pascal 中，嵌套类是您可以拥有另一个私有类（或内部类）类型的字段而不将其添加到全局名称空间，从而使其在全局可见的唯一方法。

如果内部类仅由一种方法使用,则可以通过在单元的实现部分中声明该类来达到相同的效果。但是，如果内部类是在单元的接口部分中引用的（例如，因为它用于字段或参数），则必须在同一接口部分中声明内部类，并且最终将使其可见。声明此类泛型或基本类型的字段然后将其强制转换为特定（私有）类型的技巧要比使用嵌套类干净得多。

❖ 在第 10 章中，有一个实际的例子，其中嵌套类很方便，即为 for in 循环实现自定义迭代器。

第八章 Inheritance（继承）

如果编写类的关键原因是封装，则在类之间使用继承的关键原因是灵活性。结合这两个概念，您便可以使用可以使用的数据类型，并且不会因创建这些类型的修改版本而发生更改，这些数据类型最初被称为“开放式封闭原则”：

“软件实体（类，模块，函数等）应打开以进行扩展，但应关闭以进行修改。”（BertrandMeyer，面向对象的软件构造，1988 年）

现在确实可以肯定，继承是导致紧密耦合代码的非常强大的绑定，但是也确实为继承者提供了强大的开发能力（是的，它也带来了更多的责任）。

但是，与其在此功能上进行辩论，不如说，我的目的是向您描述类型继承的工作方式，尤其是它在 Object Pascal 语言中的工作方式。

8.1 从现有类继承

我们经常需要使用我们已经编写或有人提供给我们的现有类的稍有不同的版本。

例如，您可能需要添加一种新方法或稍微更改现有方法。您可以通过修改原始代码轻松完成此操作，除非您希望能够在不同情况下使用该类的两个不同版本。另外，如果该类最初是由其他人编写的（并且您已经在库中找到它），则可能需要将更改分开保存。

具有两个相似版本的类的典型的老式替代方法是复制原始类型定义，更改其代码以支持新功能，并为所得的类赋予新名称。这可能有效，但也可能会产生问题：在复制代码时，您还会复制错误；当在其中一个代码副本中修复了错误时，您将必须记住将修复程序应用于另一个副本；并且如果您想添加新功能，则需要添加两次或更多次，具体取决于您随时间推移制作的原始代码的副本数量。

即使这在第一次编写代码时可能不会减慢您的速度，但这对于软件维护来说也是一场灾难。而且，这种方法会导致两种完全不同的数据类型，因此编译器无法帮助您利用两种类型之间的相似性。

为了解决表达类之间相似性的这类问题，Object Pascal 允许您直接从现有类中定义一个新类。这种技术被称为继承（或子类化或类型派生），并且是面向对象编程语言的基本要素之一。

要从现有的类继承，您只需要在子类的声明开始时指示该类。例如，每次创建新表单时，此操作都会自动完成：

```
type
  TForm1 = class(TForm)
  end;
```

这个简单的定义表明 TForm1 类继承了 TForm 类的所有方法，字段，属性和事件。您可以将 TForm 类的任何公共方法应用于 TForm1 类型的对象。反过来，TForm 从另一个类继承其某些方法，依此类推，直到 TObject 类（它是所有类的基类）。

相比之下，C ++，C＃和 Java 将使用类似以下内容：

```
class Form1 : TForm
{
  ...
}
```

作为继承的一个简单示例，我们可以略微更改上一章的 ViewDate 应用程序

项目，从 TDate 派生一个新类并修改其功能之一，即 GetText。您可以在 DerivedDates 应用程序项目的 DATES.PAS 文件中找到此代码。

```
type
  TNewDate = class (TDate)
  public
    function GetText: string;
  end;
```

在此示例中，TNewDate 是从 TDate 派生的。经常说 TDate 是 TNewDate 的祖先类或基类或父类，而 TNewDate 是 TDate 的子类，后代类或子类。

为了实现 GetText 函数的新版本，我使用了 FormatDateTime 函数，该函数使用（除其他功能外）预定义的月份名称。这是 GetText 方法，其中“dddddd”代表长数据格式：

```
function TNewDate.GetText: string;
begin
  Result := FormatDateTime (‘dddddd’, FDate);
end;
```

定义新类后，需要在 DerivedDates 项目形式的代码中使用此新数据类型。只需定义类型为 TNewDate 的 ADay 对象，然后在 FormCreate 方法中调用其构造函数即可：

```
type
  TDateForm = class(TForm)
    ...
  private
    ADay: TNewDate; //  更新的声明
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  ADay := TNewDate.Create; //  更新的行
  DateLabel.text := TheDay.GetText;
end;
```

无需任何其他更改，新应用程序即可正常运行。

TNewDate 类继承了增加日期，增加天数等的方法。此外，调用这些方法的旧代码仍然有效。实际上，要调用新版本的 GetText 方法，我们不需要更改源代码！Object Pascal 编译器将自动将该调用绑定到新方法。

所有其他事件处理程序的源代码都完全相同，尽管其含义发生了很大变化，如新输出所示（请参见图 8.1）。

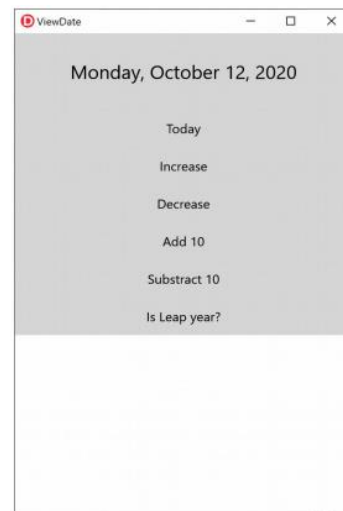

图 8.1：DerivedDates 程序的输出，其名称取决于 Windows 区域设置的月份和日期

8.2 普通基类

我们已经看到，如果可以通过编写从给定的基类继承：

```
type
  TNewDate = class (TDate)
    ...
  end;
```

但是，如果省略基类并编写：

```
type
  TNewDate = class
    ...
  end;
```

在这种情况下，您的类继承自称为 TObject 的基类。换句话说，Object Pascal 具有单根类的层次结构，其中所有类都直接或间接地从一个共同的祖先类继承。TObject 最常用的方法是 Create，Free 和 Destroy。但本书中还有很多其他内容。第 17 章提供了对该基本类的完整描述（可以视为语言的一部分，也可以视为运行时库的一部分），并对其所有方法进行了引用。

❖ 在 C＃和 Java 语言中也存在公共祖先类的概念，在这里它们简称为 Object。另一方面，C ++语言还没有这种想法，C ++程序通常具有多个独立的类层次结构。

8.3 Protected（受保护的）字段和封装

TNewDate 类的 GetText 方法的代码仅在与 TDate 类以同一单元编写时才编译。实际上，它访问祖先类的 FDate 私有字段。如果要将后代类放置在新单元中，则必须将 FDate 字段声明为受保护（或严格保护），或者在祖先类中添加一个简单的，可能受保护的方法以读取私有字段的值。

许多开发人员认为第一个解决方案始终是最好的，因为将大多数字段声明为受保护的字段将使类具有更高的可扩展性，并使编写子类更加容易。但是，这违反了封装的思想。在较大的类层次结构中，更改基类的某些受保护字段的定义与更改某些全局数据结构一样困难。如果有十个派生类正在访问此数据，则更改其定义意味着可能会修改十个类中每一个中的代码。

换句话说，灵活性，扩展性和封装性常常成为冲突的目标。发生这种情况时，您应该尝试使用封装。如果您可以在不牺牲灵活性的情况下做到这一点，那就更好了。通常，这种中间解决方案可以通过使用虚拟方法来获得，我将在下面的"后期绑定和多态性"部分中详细讨论这个主题。如果您选择不使用封装以获得更快的子类编码，则您的设计可能不遵循面向对象的原则。

还请记住，受保护的字段共享私有字段的相同访问规则，因此同一单元中的任何其他类都可以始终访问其他类的受保护成员。如上一章所述，可以通过使用严格的受保护的访问说明符来使用更强大的封装。

8.3.1 使用"Protected Hack（受保护的黑客）"

如果您是 Object Pascal 和 OOP 的新手，那么这是一个相当高级的部分，您可能想在第一次阅读本书时就跳过这一部分，因为这可能会造成很大的混乱。

给定单元保护的工作原理，甚至可以直接访问在当前单元中声明的类的基类

的受保护成员，除非您使用严格的 protected 关键字。这就是通常所说的"受保护的 hack（黑客）"的基本原理，即定义具有与基类相同的派生类的能力，其唯一目的是在基类的受保护成员处获得访问权限。下面是它的工作原理。

我们已经看到，与该类位于同一单元中的任何函数或方法都可以访问该类的私有数据和受保护的数据。例如，考虑以下简单类（属于 Protection 应用程序项目的一部分）：

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  public
    PublicData: Integer;
    function GetValue: string;
  end;
```

GetValue 方法仅返回具有两个整数值的字符串：

```
function TTest.GetValue: string;
begin
  Result := Format ('Public: %d, Protected: %d',
    [PublicData, ProtectedData]);
end;
```

将此类放在自己的单元中后，您将无法直接从其他单元访问其受保护的部分。因此，如果您编写以下代码，

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.ProtectedData := 20; //  不会编译
  Show (Obj.GetValue);
  Obj.Free;
end;
```

编译器将发出错误消息，Undeclared identifier: "ProtectedData."（未声明的标识符："ProtectedData"。）此时，您可能认为无法访问在其他单元中定义的类的受保护数据。但是，有一种解决方法。

请考虑一下，如果您创建一个显然无用的派生类，会发生什么情况：

```
type
  TFake = class (TTest);
```

现在，在声明它的同一单元中，可以调用 TFake 类的任何受保护方法。实际上，您可以调用在同一单元中声明的类的受保护方法。

但是，这如何帮助使用 TTest 类的对象？考虑到两个类共享相同的确切内存布局（没有差异），您可以强制编译器将类的对象与另一个对象视为对象，通常使用类型不安全的类型转换：

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake (Obj).ProtectedData := 20; //编译！
  Show (Obj.GetValue);
  Obj.Free;
```

```
end;
```

如运行保护应用程序项目所见，此代码可以编译并正常工作。同样，原因是 TFake 类自动继承了 TTest 基类的受保护字段，并且由于 TFake 类与试图访问继承的字段中的数据的代码位于同一单元中，因此可以访问受保护的数据。

现在，我已经向您展示了如何执行此操作，我必须警告您，这种违反类保护机制的行为很可能会导致程序错误（由于访问您实际上不应该访问的数据），并且它与良好的 OOP 技术背道而驰。但是，使用这种技术是最好的解决方案很少见，正如您通过查看库源代码和许多组件的代码所看到的。

总体而言，此技术是一种技巧，尽管可以考虑将所有效果视为语言规范的一部分，并且可以在所有平台上以及 Object Pascal 的所有当前和过去版本中使用，但应尽可能避免使用此技术。

8.4 从 Inheritance（继承）到 Polymorphism（多态）

就让您避免代码重复和在不同类之间共享代码方法而言，继承是一种不错的技术。但是，它的真正力量来自于以统一的方式处理不同类的对象的能力，这种能力在面向对象的编程语言中经常用术语“Polymorphism（多态）”表示或称为“late binding（后期绑定）”。

为了完全理解此功能，我们需要探索几个元素：派生类，虚拟方法等之间的类型兼容性，如以下几节所述。

8.4.1 Inheritance（继承）和类 Compatibility（兼容性）

如我们所知，Object Pascal 是一种严格类型化的语言。这意味着，例如，至少在没有显式类型转换的情况下，不能将整数值分配给布尔变量。 基本规则是，只有两个值具有相同的数据类型，或者（更准确地说）两个数据类型具有相同的名称并且它们的定义来自同一单元，它们才是类型兼容的。

对于类类型，此规则有一个重要例外。如果声明一个类，例如 TAnimal，并从中派生一个新类，例如 TDog，则可以将类型 TDog 的对象分配给类型 TAnimal 的变量。那是因为狗是动物！因此，尽管这可能会让您感到惊讶，但以下构造函数调用均合法：

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

更准确地说，您可以在需要任何祖先类的对象时使用后代类的对象。但是，相反的做法是不合法的。如果需要后代类的对象，则不能使用祖先类的对象。

为了简化说明，此处再次使用代码形式：

```
MyAnimal := MyDog; // This is OK
MyDog := MyAnimal; // This is an error!!!
```

实际上，尽管我们总是可以说狗是动物，但我们不能假设任何给定的动物都是狗。有时可能会如此，但并非总是如此。这是很合逻辑的，语言类型兼容性规则也遵循同样的逻辑。

在研究该语言的这一重要功能的含义之前，您可以尝试使用 Animals1 应用程序项目，该项目定义了两个简单的 TAnimal 和 TDog 类，它们继承了另一个类：

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    FKind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

这两个 Create 方法仅设置 FKind 的值，该值由 GetKind 函数返回。

此示例的 Form 如图 8.2 所示，具有两个单选按钮（由面板托管）以拾取一个或另一个类别的对象。该对象存储在 TAnimal 类型的私有字段 MyAnimal 中。每次选择一个单选按钮时都创建并重新创建表单时，将创建并初始化该类的实例（此处仅显示第二个单选按钮的代码）：

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RadioButton2Change(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

最后，Kind 按钮为当前动物调用 GetKind 方法，并在覆盖表单底部的备注中显示结果：

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
  Show(MyAnimal.GetKind);
end;
```

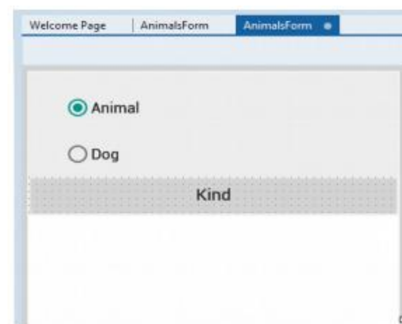

图 8.2：开发环境中 Animals1 应用程序项目的形式

8.4.2 Late Binding（后期绑定）和 Polymorphism（多态）

Object Pascal 函数和过程通常基于静态绑定，也称为早期绑定。这意味着方法调用由编译器或 Linker（链接器）解决，该调用将调用替换为对已编译函数或过程所在的特定内存位置的调用。（这也称为函数的地址。）面向对象的编程语言允许使用另一种形式的绑定，称为动态绑定或后期绑定。在这种情况下，将在运行时根据用于进行调用的实例的类型来确定要调用的方法的实际地址。

这种技术的优势被称为多态。多态性意味着您可以编写对方法的调用，并将其应用于变量，但是 Delphi 实际调用的方法取决于变量所涉及的对象的类型。

Delphi 直到运行时才能确定变量所引用的对象的实际类，这仅仅是因为上一节中讨论的类型兼容规则。

- ❖ Object Pascal 方法默认为早期绑定，如同 C ++和 C＃。原因之一是这样更有效。相反，Java 默认使用后期绑定（并提供了向编译器指示它可以使用早期绑定来优化方法的方法）。

假设一个类及其子类（再次称为 TAnimal 和 TDog）都定义了一个方法，并且该方法具有后期绑定。现在，您可以将此方法应用于通用变量，例如 MyAnimal，在运行时它可以引用类 TAnimal 的对象或类 TDog 的对象。实际的调用方法是在运行时确定的，具体取决于当前对象的类。

Animals2 应用程序项目扩展了 Animals1 项目，以演示此技术。在新版本中，TAnimal 和 TDog 类具有一个新方法：Voice（声音），这意味着将所选动物发出的声音以文本和声音的形式输出。此方法在 TAnimal 类中定义为 virtual（虚拟）方法，以后在定义 TDog 类时使用 virtual 和 override 关键字将其覆盖：

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

当然，这两种方法也需要实现。这是一个简单的方法：

```
function TAnimal.Voice: string;
begin
  Result := 'AnimalVoice';
end;

function TDog.Voice: string;
begin
  Result := 'ArfArf';
end;
```

现在，调用 MyAnimal.Voice 有什么作用？这取决于。如果 MyAnimal 变量当前引用 TAnimal 类的对象，则它将调用方法 TAnimal.Voice。如果它引用 TDog 类的对象，它将改为调用方法 TDog.Voice。这仅是因为函数是虚拟的。

对 MyAnimal.Voice 的调用将适用于 TAnimal 类的任何后代实例的对象，甚至是在此方法调用之后或其作用域之外定义的类。编译器不需要了解所有后代就可以使调用与它们兼容。只需要祖先类。换句话说，对 MyAnimal.Voice 的此调用与所有将来的 TAnimal 子类兼容。

这是面向对象的编程语言偏爱可重用性的关键技术原因。您可以编写使用层次结构中的类的代码，而无需了解该层次结构中的特定类。换句话说，即使您已经使用它编写了数千行代码，该层次结构和程序仍然是可扩展的。当然，有一个条件-层次结构的祖先类需要非常仔细地设计。

Animals2 应用程序项目演示了这些新类的用法，并具有与先前示例相似的形式。通过单击按钮执行此代码，显示输出并产生一些声音：

```
begin
  Show (MyAnimal.Voice);
  MediaPlayer1.FileName := SoundsFolder + MyAnimal.Voice + '.wav';
  MediaPlayer1.Play;
end;
```

- 该应用程序使用 MediaPlayer 组件播放该应用程序附带的两个声音文件之一（声音文件以实际声音命名，即 Voice 方法返回的值）。普通动物的声音相当随机，狗则吠叫。
- 现在，只要文件在正确的文件夹中，该代码就可以在 Windows 上轻松运行，但是在移动平台上进行部署需要一定的精力。查看实际的演示，以了解部署和文件夹的结构。

8.4.3 Override(覆盖),Redefine(重新定义)和 Reintroduce(重新引入)方法

正如我们已经看到的那样，要覆盖后代类中的后绑定方法，您需要使用 override 关键字。请注意，只有在该方法在祖先类中定义为 virtual（虚拟）方法时，才可以执行此操作。否则，如果它是静态方法，则无法通过更改祖先类的代码来激活后期绑定。

- 您可能还记得我在上一章中也使用了相同的关键字来覆盖从基本 TObject 类继承的 Destroy 默认析构函数。

规则很简单：定义为 static 的方法在每个子类中均保持静态，除非您使用具有相同名称的新虚拟方法将其隐藏。在每个子类中，定义为虚拟的方法仍会滞后。由于编译器会为后期绑定方法生成不同的代码，因此无法更改此方法。

要重新定义静态方法，只需将方法添加到具有与原始参数相同或不同的参数的子类中，而无需任何其他说明。要覆盖虚拟方法，必须指定相同的参数并使用 override 关键字：

```
type
  TMyClass = class
    procedure One; virtual;
    procedure Two; // 静态方法
  end;

  TMySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

重新定义的方法 Two 没有后期绑定。因此，将其应用于基类的变量时，无论如何（无论变量引用的是派生类的对象还是该方法的版本不同），它都将调用基类方法。

通常有两种方法可以覆盖方法。一种是用新版本替换祖先类的方法。另一种是在现有方法中添加更多代码。第二种方法可以通过使用 Inherited 关键字调用祖先类的相同方法来实现。例如，你可以写

```
procedure TMySubClass.One;
begin
  // 新代码
  ...
  // 调用继承过程 TMyClass.One
  inherited One;
end;
```

您可能想知道为什么需要使用 override 关键字。用其他语言，当您在子类中重新定义虚拟方法时，您将自动覆盖原始方法。但是，使用特定的关键字可以使编译器检查祖先类中的方法名称与子类中的方法名称之间的对应关系（拼写重新定义的函数是某些其他 OOP 语言中的常见错误），请检查方法在祖先类中是虚

拟的，依此类推。

- ❖ 另一种流行的 OOP 语言具有相同的覆盖关键字 C＃。鉴于语言共享一个共同的设计师这一事实，这不足为奇。安德斯·海斯伯格（Anders Hejlsberg）的文章很长，解释了为什么 Override 关键字是设计库的基本版本控制工具，您可以在 http://www.artima.com/intv/nonvirtual.html 上阅读。最近，Apple 的 Swift 语言还采用了 override 关键字来修改派生类中的方法。

此关键字的另一个优点是，如果您在由库的类继承的任何类中定义静态方法，即使使用与您的方法同名的新虚拟方法更新了库，也不会出现问题。已经定义。因为您的方法没有用 override 关键字标记，所以它将被视为一个单独的方法，而不是添加到库中的方法的新版本（这可能会破坏现有代码）。

对重载的支持使此图更加复杂。子类可以使用 overload（重载）关键字提供方法的新版本。如果该方法的参数与基类中的版本不同，则它实际上成为重载方法；否则，它将替换基类方法。这是一个例子：

```
type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); overload;
  end;
```

请注意，该方法不需要在基类中标记为重载。

但是，如果基类中的方法是虚拟的，则编译器将发出警告方法“One”隐藏基类型为“ TMyClass”的虚拟方法。为避免从编译器收到此消息，并根据您的意图更精确地指示编译器，请执行以下操作：可以使用特定的重新引入指令：

```
type
  TMyClass = class
    procedure One; virtual;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); reintroduce; overload;
  end;
```

您可以在 ReintroduceTest 应用程序项目中找到此代码，然后进一步进行试验。

- ❖ 当您要将自定义 Create 构造函数添加到已经从 TComponent 基类继承了虚拟 Create 构造函数的组件类中时，使用 reintroduce 关键字。

8.4.4 Inheritance（继承）和 Constructor（构造函数）

如我们所见，您可以在继承类的方法中使用 Inherited 关键字来调用相同名称的方法（或不同方法）。构造函数也是如此。在其他语言（例如 C++，C＃或 Java）中，对基类构造函数的调用是隐式的和强制性的（当您必须将参数传递给基类构造函数时），而在 Object Pascal 中，并不严格要求调用基类构造函数。

但是，在大多数情况下，手动调用基类构造函数非常重要。例如，对于任何组件类都是这种情况，因为组件初始化实际上是在 TComponent 类级别完成的：

```
constructor TMyComponent.Create (Owner: TComponent);
begin
  inherited Create (Owner);
```

```
        // 具体代码...
    end;
```

这一点特别重要，因为对于组件而言，Create 是一个虚拟方法。

同样，对于所有类，Destroy 析构函数是一个虚拟方法，您应该记住调用其中的继承方法。

仍然存在一个问题：如果要创建一个仅继承自 TObject 的类，则需要在其构造函数中调用基础 TObject.Create 构造函数吗？从技术角度来看，鉴于构造函数为空，答案为“否”。但是，我认为始终调用基类构造函数是个好习惯，无论如何。但是，如果您是一个性能狂热者，我将承认这会不必要地降低您的代码速度……减少了微秒的时间。

除了玩笑，这两种方法都有很好的理由，但是对于初学者来说，我建议始终将基类构造函数称为良好的编程习惯，以提高编码的安全性。

8.4.5 Virtual（虚拟）与 Dynamic（动态）方法

在 Object Pascal 中，有两种不同的方法来激活后期绑定。您可以像以前看到的那样将方法声明为 Virtual（虚拟）方法，也可以将其声明为 Dynamic（动态）方法。这两个关键字的语法完全相同，并且使用它们的结果也相同。区别在于编译器用于实现后期绑定的内部机制。

虚拟方法基于虚拟方法表（或 VMT，但通称 Vtable）。虚拟方法表是方法地址的数组。对于虚拟方法的调用，编译器会生成代码以跳转到对象的虚拟方法表中第 n 个插槽中存储的地址。

虚拟方法表允许快速执行方法调用。它们的主要缺点是，即使在子类中未重写该方法，也需要为每个后代类的每个虚拟方法输入一个条目。有时，这会在整个类层次结构中传播虚拟方法表条目（甚至对于未重新定义的方法）。为了多次存储相同的方法地址，这可能需要大量的内存。

另一方面，使用指示方法的唯一编号调度动态方法调用。与虚拟方法的简单单步表查找相比，对相应功能的搜索通常要慢。优点是动态方法条目仅在后代覆盖方法时才在后代中传播。对于大型或深层对象层次结构，使用动态方法而不是虚拟方法可以节省大量内存，而速度损失最小。

从程序员的角度来看，这两种方法之间的差异仅在于内部表示形式不同，并且速度或内存使用量略有不同。除此之外，虚拟和动态方法是相同的。

现在已经解释了这两种模型之间的区别，必须强调的是，在大多数情况下，应用程序开发人员使用虚拟而非动态。

Windows 上的消息处理程序

在为 Windows 构建应用程序时，可以使用特殊的后期绑定方法来处理 Windows 系统消息。为此，Object Pascal 提供了另一条指令 message，以定义消息处理方法，该方法必须是具有正确类型的单个 var 参数的过程。message 指令后跟该方法要处理的 Windows 消息的编号。例如，以下代码使您可以处理用户定义的消息，并使用 WM_USER Windows 常数指示的数值：

```
type
  TForm1 = class(TForm)
    ...
    procedure WmUser (var Msg: TMessage); message WM_USER;
  end;
```

过程的名称和参数的实际类型取决于您，只要物理数据结构与 Windows 消息结构匹配即可。用于与 Windows API 交互的单元包括各种 Windows 消息的许多预定义记录类型。该技术对于熟悉 Windows 消息和 API 函数的资深 Windows 程序员可能非常有用，但它绝对与其他操作系统（例如 macOS，iOS 和 Android）不兼容。

8.5 Abstract（抽象）方法和类

在创建类的层次结构时，有时很难确定哪个是基类，因为它可能并不代表实际的实体，而仅用于保持某些共享行为。一个例子是诸如猫或狗类之类的动物基类。此类不希望为其创建任何对象的类通常被称为抽象类，因为它没有具体且完整的实现。抽象类可以具有抽象方法，这些方法没有实际的实现。

8.5.1 Abstract（抽象）方法

abstract 关键字用于声明仅在当前类的子类中定义的虚拟方法。abstract 指令完全定义了该方法。它不是前向声明。如果尝试提供该方法的定义，则编译器会抱怨。

在 Object Pascal 中，您可以创建具有抽象方法的类的实例。

但是，当您尝试这样做时，编译器会发出警告消息：正在构造包含抽象方法的<class name>实例。如果碰巧在运行时调用抽象方法，Delphi 将引发特定的运行时异常。

C ++，Java 和其他语言使用更严格的方法：在这些语言中，您无法创建抽象类的实例。

您可能想知道为什么要使用抽象方法。原因在于使用了多态性。如果类 TAnimal 具有虚拟抽象方法 Voice，则每个子类都可以重新定义它。好处是，您现在可以使用通用的 MyAnimal 对象来引用子类定义的每个动物，然后调用此方法。如果 TAnimal 类的接口中不存在此方法，则执行静态类型检查的编译器将不允许调用。

使用通用 MyAnimal 对象，您只能调用由其自己的类 TAnimal 定义的方法。

您不能调用子类提供的方法，除非父类至少具有此方法的声明（以抽象方法的形式）。下一个应用程序项目 Animals3 演示了抽象方法的使用和抽象调用错误。这是此新示例的类的接口：

```
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;
```

最有趣的部分是类 TAnimal 的定义，其中包括一个虚拟抽象方法：Voice。 还需要注意的是，每个派生类都将覆盖此定义并添加一个新的虚拟方法 Eat。这两种不同方法的含义是什么？要调用语音功能，我们只需编写与程序先前版本中相同的代码即可：

```
Show (MyAnimal.Voice);
```

我们怎么调用 Eat 方法？我们不能将其应用于 TAnimal 类的对象。该声明：

```
Show (MyAnimal.Eat);
```

产生编译器错误 *Field identifier expected*（预期字段标识符）。

要解决此问题，可以使用动态且安全的类型转换将 TAnimal 对象视为 TCat 或 TDog 对象，但这将是非常麻烦且容易出错的方法：

```
begin
  if MyAnimal is TDog then
    Show (TDog(MyAnimal).Eat)
  else if MyAnimal is TCat then
    Show (TCat(MyAnimal).Eat);
```

该代码将在后面的"安全类型转换运算符"部分中说明。将虚拟方法定义添加到 TAnimal 类是解决问题的一种典型解决方案，而 abstract 关键字的存在有利于此选择。上面的代码看起来很丑陋，而避免使用此类代码正是使用多态的原因。

最后请注意，当类具有抽象方法时，通常将其视为抽象类。但是，您也可以使用 abstract 指令专门标记一个类（即使它没有抽象方法，也将被视为抽象类）。同样，在 Object Pascal 中，这不会阻止您创建类的实例，因此，在这种语言中，抽象类声明的用处非常有限。

8.5.2 Sealed（密封）类和 Final（最终）方法

正如我提到的，Java 具有非常动态的方法，默认为后期绑定（或 virtual 虚拟函数）。因此，该语言引入了一些概念，例如您不能继承（sealed 密封）的类和不能在派生类中覆盖的方法（(final 最终方法或 non-virtual 非虚拟方法）。

密封类是您无法进一步继承的类。如果您要分发组件（不包含源代码）或运行时包，并且想要限制其他开发人员修改代码的能力，则这可能是有道理的。最初的目标之一还在于提高运行时安全性，而在诸如 Object Pascal 之类的完全编译的语言中，您通常不需要这样做。

最终方法是虚拟方法，您无法在继承的类中进一步重写。

同样，尽管它们在 Java 中确实有意义（默认情况下所有方法都是虚拟的，并且最终方法得到了显着优化），但它们已在 C# 中被采用，其中显式标记了虚拟函数，而重要性则要低得多。同样，如果很少使用它们，则将它们添加到 Object Pascal。

就语法而言，这是一个密封类的代码：

```
type
  TDeriv1 = class sealed (TBase)
```

```
        procedure A; override;
    end;
```

尝试从中继承会导致错误"Cannot extend sealed class TDeriv1（无法扩展密封类 TDeriv1）"。

这是 final 方法的语法：

```
type
  TDeriv2 = class (TBase)
    procedure A; override; final;
  end;
```

从此类继承并覆盖 A 方法会导致编译器错误"Cannot override a final method（无法覆盖最终方法）"。

8.6 Safe（安全）类型转换运算符

正如我们之前所看到的，后代类的语言类型兼容性规则允许您在期望祖先类的地方使用后代类。正如我所提到的，相反是不可能的。

现在假设 TDog 类具有 Eat 方法，该方法在 TAnimal 类中不存在。如果变量 MyAnimal 引用了一条狗，则您可能希望能够调用该函数。但是，如果您尝试执行该操作，并且该变量引用了另一个类，则结果是错误。通过进行显式类型转换，我们可能会导致讨厌的运行时错误（或更糟糕的是，细微的内存覆盖问题），因为编译器无法确定对象的类型是否正确以及我们正在调用的方法是否实际存在。

为了解决该问题，我们可以使用基于运行时类型信息的技术。

本质上，因为每个对象在运行时"知道"其类型和其父类。

我们可以使用 is 运算符或使用 TObject 类的某些方法来请求此信息。is 运算符的参数是一个对象和一个类类型，返回值是一个布尔值：

```
if MyAnimal is TDog then
  ...
```

仅当 MyAnimal 对象当前引用类 TDog 的对象或 TDog 的后代类型并与之兼容时，才将 is 表达式评估为 True。这意味着，如果测试存储在 TAnimal 变量中的 TDog 对象是否真的是 TDog 对象，则测试将成功。换句话说，如果可以安全地将对象（MyAnimal）分配给数据类型（TDog）的变量，则此表达式的值为 True。

❖ is 运算符的实际实现由 TObject 类的 InheritsFrom 方法提供。因此，您可以编写与 MyAnimal.InheritsFrom（TDog）相同的表达式。之所以直接使用此方法，是因为它也可以应用于不支持 is 运算符的类引用和其他特殊目的类型。

既然您已经确定动物是狗，那么可以通过编写以下代码来使用直接类型转换（通常是不安全的）：

```
if MyAnimal is TDog then
begin
  MyDog := TDog (MyAnimal);
  Text := MyDog.Eat;
end;
```

相同的操作可以直接由另一个相关的类型强制转换操作符完成，例如，仅当所请求的类与实际的类兼容时才转换对象。as 运算符的参数是一个对象和一个类类型，结果是一个对象"转换"为新的类类型。我们可以编写以下代码段：

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

如果我们只想调用 Eat 函数，那么我们也可以使用更短的符号：

```
(MyAnimal as TDog).Eat;
```

该表达式的结果是 TDog 类数据类型的对象，因此您可以将其应用于该类的任何方法。传统转换与使用 as 转换之间的区别在于，第二个转换检查对象的实际类型，并且如果该类型与您尝试将其转换为的类型不兼容，则会引发异常。引发的异常是 EInvalidCast（异常将在下一章中介绍）。

➢ 相反，在 C＃语言中，如果对象不是类型兼容的，则 as 表达式将返回 nil，而直接类型转换将引发异常。因此，与 Object Pascal 相比，基本上这两个操作是相反的。

为避免此异常，请使用 is 运算符，如果成功，则进行普通类型转换（实际上，没有理由依次使用 is 和 as，进行两次类型检查-尽管您经常会看到合并的使用 is 和 as）：

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

这两种类型转换运算符在 Object Pascal 中都非常有用，因为您通常希望编写可与许多相同类型甚至不同类型的组件一起使用的通用代码。例如，将组件作为参数传递给事件响应方法时，将使用通用数据类型（TObject），因此您通常需要将其强制转换回原始组件类型：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
  ...
end;
```

这是一种常见的技术，我将在以后的示例中使用它（第 10 章介绍了事件）。

这两个类型转换运算符 is 和 as 非常强大，您可能会想将它们视为标准编程结构。尽管它们确实功能强大，但您可能应该将它们的使用限制为特殊情况。当您需要解决涉及多个类的复杂问题时，请首先尝试使用多态性。仅在不能单独使用多态的特殊情况下，才应尝试使用类型转换运算符对其进行补充。

❖ 使用类型强制转换运算符对性能有轻微的负面影响，因为它必须遍历类的层次结构以查看类型转换是否正确。如我们所见，虚拟方法调用仅需要内存查找，这要快得多。

8.7 Visual Form Inheritance（可视表单继承）

继承不仅在库类或您编写的类中使用，而且在基于 Object Pascal 的整个开发环境中非常普遍。

如我们所见，当您在 IDE 中创建表单时，这是从 TForm 继承的类的实例。因此，即使最终使用简单的事件处理程序编写大部分代码的情况下，任何可视化应用程序都具有基于继承的结构。

但是，即使是经验丰富的开发人员也鲜为人知的是，您可以从已经创建的表单中继承一种新表单，这种功能通常称为可视表单继承（这是 Object Pascal 开发环境特有的功能）。

有趣的是，您可以直观地看到继承的作用，并直接找出其规则！在实践中还有用吗？好吧，这主要取决于您正在构建的应用程序的类型。如果它具有多种 Form，其中某些 Form 彼此非常相似或仅包含公共元素，则可以将公共组件和公共事件处理程序放置在基本表单中，并将特定的行为和组件添加到子类中。另一个常见的情况是使用可视 Form 继承为特定公司自定义某些应用程序形式，而无需复制任何源代码（这是首先使用继承的核心原因）。

您也可以使用可视表单继承来为不同的操作系统和表单因素（例如，手机到平板电脑）自定义应用程序，而无需复制任何源代码或表单定义代码。只需从标准格式继承客户端的特定版本。请记住，可视继承的主要优点是您以后可以更改原始表单并自动更新所有派生的表单。这是在面向对象的编程语言中继承的众所周知的优点。但是有一个有益的副作用：polymorphism（多态）。

您可以将虚拟方法添加到基本表单中，并以子类形式覆盖它。

然后，您可以引用这两种形式，并为每种形式调用此方法。

- 构建具有相同元素的表单的另一种方法是依靠框架，即表单面板的视觉组成。在这两种情况下，您都可以在设计时使用两种版本的表单。
- 但是，在可视表单继承中，您要定义两个不同的类（父类和派生类），而对于框架，则要处理框架类和由表单托管的该框架的实例。

8.7.1 从基本表单继承

一旦您对继承是什么有了一个清晰的概念，控制视觉表单继承的规则就非常简单。基本上，子类表单具有与父表单相同的组件以及一些新组件。您不能删除基类的组件，尽管（如果它是可视控件）可以使它不可见。重要的是您可以轻松更改继承的组件的属性。

请注意，如果您以继承的形式更改组件的属性，则父表单中相同属性的任何修改都将无效。更改组件的其他属性也会影响继承的版本。您可以使用“对象”检查器的“还原为继承的本地菜单”命令来重新同步这两个属性值。通过将两个属性设置为相同的值并重新编译代码，可以完成同一件事。修改多个属性后，您可以通过应用组件的本地菜单中的“还原为继承的”命令，将所有属性重新同步到基本版本。

除了继承组件之外，新表单还继承了基本表单的所有方法，包括事件处理程序。您可以以继承的表单添加新的处理程序，也可以覆盖现有的处理程序。

为了演示可视表单继承的工作原理，我建立了一个非常简单的示例，称为VisualInheritTest。我将逐步介绍如何构建它。首先，启动一个新的 multi-device（多设备）项目，选择一个空白项目，然后在其主窗体中添加两个按钮。然后选择“文件”“新建”“其他”，然后在“新建项目”对话框中选择“可继承项目”页面（见图 8.3）。在这里，您可以选择要继承的表单。

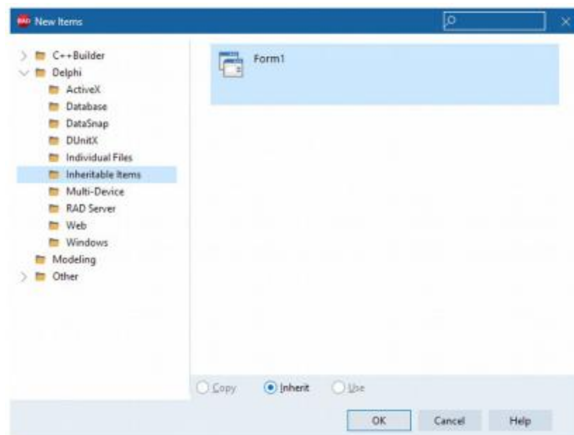

图 8.3：“新建项目”对话框使您可以创建继承的表单。

新表单具有相同的两个按钮。这是新表格的初始文字说明：

```
inherited Form2: TForm2
  Caption = ‘Form2’
  ...
end
```

这是它的初始类声明，在这里您可以看到基类不是通常的 TForm 而是实际的基类形式：

```
type
  TForm2 = class(TForm1)
  private
  { Private declarations }
  public
  { Public declarations }
  end;
```

请注意，文本描述中存在继承的关键字；还应注意，该表单确实具有一些组件，尽管它们是在基类表单中定义的。如果更改按钮之一的标题并添加新按钮，则文本描述将相应更改：

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
 inherited Button1: TButton
    Text = 'Hide Form'
  end
  object Button3: TButton
    ...
    Text = 'New Button'
      OnClick = Button3Click
    end
end
```

仅列出具有不同值的属性，因为其他属性仅照原样继承。

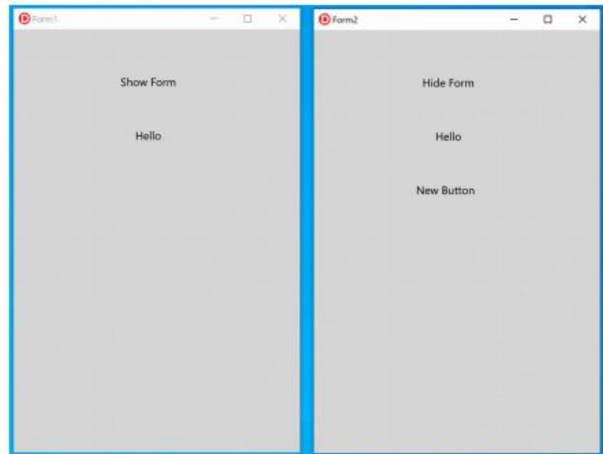

图 8.4：运行时 VirtualInheritTest 示例的两个表单

第一个表单的每个按钮都有一个带有简单代码的 OnClick 处理程序。第一个按钮显示第二个窗体，调用其 Show 方法；第二个按钮是一条简单消息。

继承表单会发生什么？首先，我们应该更改“显示”按钮的行为，以将其实现为“隐藏”按钮。这意味着不执行基类事件处理程序（因此，我已注释掉默认的继承调用）。对于 Hello 按钮，我们可以通过保留继承的调用，在基类显示的消息中添加第二条消息：

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  // inherited;
  Hide;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hello from Form2');
end;
```

请记住，与继承方法不同，后者可以使用 Inherited 关键字以相同的名称调用基类方法，在事件处理程序中，Inherited 关键字表示对基本形式的相应事件处理程序的调用（无论事件如何）处理程序方法名称）。

当然，您也可以将基本表单的每个方法视为表单的方法，然后自由调用它们。本示例使您可以探索可见表单继承的某些功能，但是要了解其真正的功能，您需要查看比本书有更多探索空间的更复杂的现实示例。

第九章 异常处理

在继续介绍 Object Pascal 语言中的类的其他功能之前，我们需要集中精力于用于处理错误情况（称为异常）的一组特定对象。

异常处理的思想是通过以简单统一的方式添加处理软件或硬件错误（以及任何其他类型的错误）的功能来使程序更健壮。程序可以幸免于此类错误或正常终止，从而允许用户在退出之前保存数据。异常使您可以将错误处理代码与常规代码分开，而不是将两者纠缠在一起。您最终将编写的代码更加紧凑，并且减少了与实际编程目标无关的维护工作。

另一个好处是，异常定义了统一且通用的错误报告机制，组件库也使用该机制。在运行时，发生错误时，系统会引发异常。如果您的代码是正确编写的，它可以确认问题并尝试解决;否则，异常将传递给其调用代码，依此类推。最终，如果您的代码中没有任何部分处理该异常，则系统通常会通过显示标准错误消息并尝试继续执行该程序来处理该异常。在特殊情况下，您的代码是在任何异常处理块之外执行的，引发异常将导致程序终止。Object Pascal 中异常处理的整个机制基于五个单独的关键字：

- try 划定受保护代码块的开头。
- except 定界受保护代码块的末尾，并引入例外处理代码。
- on 标记与特定异常绑定的单个异常处理语句，每个语句都具有异常类型 do 语句的语法。
- finally 用于指定即使发生异常也必须始终执行的代码块。
- raise 是用于触发异常的语句，并具有一个异常对象作为参数（此操作在其他编程语言中称为 throw）。

这是 Object Pascal 中异常处理关键字与基于 C ++异常语法（例如 C＃和 Java）的语言的简单比较表：

Object Pascal	C ++（例如 C＃和 Java）
try	try
except on	catch
finally	finally
raise	throw

使用 C ++语言术语，可以引发异常对象并按类型捕获它。这与 Object Pascal 中的情况相同，在异常中，您将一个异常对象传递给 raise 语句，并将其作为 except on 语句的参数接收。

9.1 Try-Except 块

让我从一个非常简单的 try-except 示例（ExceptionsTest 应用程序项目的一部分）开始，该示例具有常规的异常处理块：

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; //  如果 B 等于 0 则引发异常
    Inc (Result);
  except
    Result := 0;
  end;
```

```
    // more
  end;
```

❖ 在 Delphi 调试器中运行程序时，默认情况下，即使有异常处理程序，调试器也会在遇到异常时停止程序。当然，这通常是您想要的，因为您想知道异常发生的位置，并且可以逐步看到处理程序的调用。如果您只想让程序在正确处理了异常后运行，并查看用户会看到什么，请使用"Run without debugging（无需调试即可运行）"命令运行程序，或在调试器中禁用所有（或某种类型的）异常选项。

在上面的代码中，使异常"沉默"，并将结果设置为零在现实世界的应用程序中并没有多大意义，但是该代码旨在帮助您在简单的场景中理解核心机制。这是用于调用函数的事件处理程序的代码：

```
var
  N: Integer;
begin
  N := DividePlusOne (10, Random(3));
  Show (N.ToString);
```

如您所见，程序使用随机生成的值，因此当您单击按钮时，您可能处于有效状态（3 分之 2）或无效状态。这样，可以有两个不同的程序流：

- 如果 B 不为零，则程序将进行除法，执行增量，然后在其后的 end 语句中跳过 except 语句（//更多）
- 如果 B 为零，则除法将引发异常，以下所有语句将被跳过（在这种情况下，仅一个），直到第一个封闭的 try-expect 块被执行。在 exception 块之后，程序将不会返回到原始语句，而是跳过直到 except 块之后执行它之后的第一个语句（//more）。

描述此异常模型的一种方法是说它遵循非恢复方法。如果发生错误，尝试处理错误情况并返回引起错误的语句非常危险，因为此时的程序状态可能未定义。异常会极大地改变执行流程，跳过以下语句的执行，并回滚堆栈，直到找到正确的错误处理代码为止。

上面的代码有一个非常简单的 except 块，没有 on 语句。当您需要处理多种类型的异常（或多种异常类类型）或要访问传递给该块的异常对象时，您需要具有一个或多个 on 语句：

```
function DividePlusOneBis (A, B: Integer): Integer;
begin
  try
    Result := A div B; //  如果 B 等于 0 则错误
    Result := Result + 1;
  except
    on E: EDivByZero do
    begin
      Result := 0;
      ShowMessage (E.Message);
    end;
  end;
end;
```

在异常处理语句中，我们捕获了由运行时库定义的 EDivByZero 异常。这些异常类型中有许多涉及运行时问题（例如被零除或错误的动态转换），系统问题（例如内存不足错误）或组件错误（例如作为错误的索引）。所有这些异常类都继承自基类 Exception，该基类提供了一些最小的功能，例如我在上面的代码中使用的 Message 属性。这些类形成具有某种逻辑结构的实际层次结构。

❖ 请注意，虽然 Object Pascal 中的类型通常以开头的字母 T 标记，但是异常类对该规则采用例外，并且通常以字母 E 开头。

9.1.1 异常层次结构

这是在运行时库的 System.SysUtils 单元中定义的核心异常类的部分列表（大多数其他系统库添加了它们自己的异常类型）：

```
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
```

❖ 我不了解您，但我仍然必须弄清楚我认为最奇怪的异常类（有趣的 EProgrammerNotFound 异常）的确切使用情况！

既然您已经看到了核心的异常层次结构，那么我可以在 except-on 语句的先前描述中添加一条信息。依次评估这些语句，直到系统找到与引发的异常对象类型匹配的异常类为止。现在，使用的匹配规则是我们在上一章中讨论的类型兼容性规则：异常对象与其自身特定类型的任何基本类型都兼容（例如 TDog 对象与 TAnimal 类兼容）。

这意味着您可以具有多个与异常匹配的异常处理程序类型。 如果您希望能够处理更细粒度的异常（层次结构中的较低类）以及更通用的异常（如果以前的匹配项均不匹配），则必须列出处理程序块，从更具体到更通用（或从子异常类到其父类）。另外，如果为 Exception 类型编写处理程序，它将是一个包罗万象的子句，因此它必须是序列的最后一个。

这是一个在一个块中包含两个处理程序的代码段：

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; //  如果 B 等于 0 则错误
    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; // except 块结束
end;
```

在此代码中，在同一个 try 块之后有两个不同的异常处理程序。您可以具有任意数量的这些处理程序，这些处理程序按上述顺序进行评估。

请记住，对于每个可能的异常使用处理程序通常不是一个好选择。最好将未知异常留给系统。默认的异常处理程序通常在消息框中显示异常类的错误消息，然后恢复程序的正常运行。

❖ 您实际上可以通过为 Application.OnException 事件提供一种方法来修改常规异常处理程序，例如，将异常消息记录在文件中而不是向用户显示。

9.1.2 引发异常

您在 Object Pascal 编程中遇到的大多数异常将由系统生成，但是当您在运行时发现无效或不一致的数据时，也可以在自己的代码中引发异常。

在大多数情况下，对于自定义例外，您将定义自己的例外类型。只需创建默认异常类的新子类或我们在上面看到的现有子类之一即可：

```
type
  EArrayFull = class (Exception);
```

在大多数情况下，您无需向新的异常类添加任何方法或字段，并且只需声明一个空的派生类即可。

这种异常类型的方案是一种将元素添加到数组中的方法，当数组已满时会引发错误。这是通过创建异常对象并将其传递给 raise 关键字来实现的：

```
if MyArray.IsFull then
  raise EArrayFull.Create ('Array full');
```

此 Create 方法（从 Exception 类的基础继承）具有一个字符串参数，用于向用户描述异常。

❖ 您无需担心会破坏为异常创建的对象，因为异常处理程序机制会自动删除该对象。

还有一种使用 raise 关键字的方案。在 except 块内，您可能要执行一些操作，但不要捕获异常，让它流到封闭的异常处理程序块。在这种情况下，您可以不带任何参数调用 raise。该操作称为重新引发异常。

9.1.3 异常与 Stack（堆栈）

当程序引发异常而当前例程无法处理该异常时，您的方法和函数调用堆栈会如何处理？程序开始在堆栈中已存在的函数中搜索处理程序。这意味着程序将从现有函数退出，并且不执行其余语句。

要了解其工作原理，您可以使用调试器或添加一些简单的输出行，以便在执行某些源代码语句时得到通知。在下一个应用程序项目 ExceptionFlow 中，我遵循了第二种方法。

例如，当您以 ExceptionFlow 应用程序项目的形式按下 Raise1 按钮时，将引发并处理异常，因此将永远不会执行代码的最后一部分：

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  // 粗心的调用
  AddToArray (24);
  Show ('程序永远不会到达这里');
end;
```

请注意，此方法将调用 AddToArray 过程，该过程始终会引发异常。处理异常后，流程将在处理程序之后而不是引发异常的代码之后再次开始。考虑以下修改方法：

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
  try
    // 此过程引发异常
    AddToArray (24);
    Show ('程序永远不会到达这里');
  except
    on EArrayFull do
      Show ('处理异常');
  end;
  Show ('ButtonRaise1Click 调用已完成');
end;
```

最后一个 Show 调用将在第二个之后立即执行，而第一个始终被忽略。我建议您运行该程序，更改其代码，并进行实验以充分理解引发异常时的程序流程。

- 鉴于您处理异常的代码位置与引发异常的代码位置不同，因此能够知道实际以哪种方法引发异常会很好。
- 虽然有多种方法可以在引发异常时获取堆栈跟踪并使该信息在处理程序中可用，但这确实是一个高级主题，我不打算在此处讨论。在大多数情况下，Object Pascal 开发人员都依赖于第三方库和工具（如 Jedi Component Library 中的 JclDebug，madExcept 或 EurekaLog）。另外，您必须生成一个 MAP 文件，并将其包含在代码中，该 MAP 文件由编译器创建，并列出应用程序中每种方法和函数的内存地址。

9.2 Finally 块

最后，我提到了第四个关键字，用于异常处理。finally 块用于执行应始终执行的某些操作（通常是清理操作）。实际上，无论是否发生异常，都将处理 finally 块中的语句。相反，仅在未引发异常或引发并处理了异常的情况下，才会执行 try 块之后的普通代码。换句话说，即使在引发异常的情况下，finally 块中的代码始终在 try 块的代码之后执行。

考虑一下此方法（ExceptFinally 应用程序项目的一部分），该方法每执行一次耗时的操作，并在标题中显示其状态：

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := '计算中...';
  J := 0;
  // 长时间（错误的）计算...
  for I := 1000 downto 0 do
    J := J + J div I;
  Caption := 'Finished';
  Show ('总计: ' + J.ToString);
end;
```

由于算法存在错误（因为变量 I 可以达到 0 且也用于除法），因此程序会中断，但不会重置表格标题。这是 try-finally 块的用途：

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := '计算中...';
  J := 0;
  try
    // 长时间（错误的）计算...
    for I := 1000 downto 0 do
    J := J + J div I;
    Show ('总计: ' + J.ToString);
  finally
    Caption := '结束';
  end;
end;
```

程序执行此函数时，无论是否发生（任何形式的）异常，它始终会重置游标。此函数版本的缺点是无法处理异常。

9.2.1 Finally 和 Except

奇怪的是，在 Object Pascal 语言中，try 块后面可以是 except 或 finally 语句，但不能同时出现。鉴于您通常希望同时拥有两个块，典型的解决方案是使用两个嵌套的 try 块，根据情况需要，将内部一个与 finally 语句关联，将外部一个与 else 语句关联，反之亦然。这是 ExceptFinally 应用程序项目的第三个按钮的代码：

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := '计算中...';
  J := 0;
  try
    try
      //长时间（错误的）计算...
      for I := 1000 downto 0 do
        J := J + J div I;
      Show ('总计: ' + J.ToString);
    except
      on E: EDivByZero do
      begin
        // 用新消息重新引发异常
        raise Exception.Create ('算法错误');
      end;
    end;
  finally
    Caption := '结束';
  end;
end;
```

9.2.2 在 Finally 块中恢复游标

try-finally 块的常见用例是资源的分配和释放。另一个相关情况是，即使操作引发异常，您也需要在操作完成后重置临时配置。

您必须还原的临时配置设置的一个示例是沙漏形光标的设置，它在长时间的操作过程中显示，并在结束时将其删除，以恢复原始的活动光标。即使代码很简单，也总是会引发异常，因此您应该始终使用 try-finally 块。

在 RestoreCursor 应用程序示例（VCL 应用程序，因为 FireMonkey 中的游标管理稍微复杂一些），我编写了以下代码来保存当前光标，临时设置沙漏和最后恢复原始光标。：

```
var CurrCur := Screen.Cursor;
Screen.Cursor := crHourGlass;
try
  // 一些缓慢的操作
  Sleep (5000);
finally
  Screen.Cursor := CurrCur;
end;
```

9.2.3 使用托管记录还原游标

为了保护资源分配或定义要恢复的临时配置，可以使用托管记录来代替显式的 try-finally 块，该记录需要编译器添加一个固有的 finally 块。即使在定义记录时付出了一些最初的努力，这也导致编写更少的代码来保护资源或恢复配置。

这是一条托管记录，表示上一部分中的代码具有相同的行为，将当前光标保存在 Initialize 方法的字段中，并在 Finalize 方法中将其重置：

```
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 := Screen.Cursor;
  Screen.Cursor := crHourGlass;
end;

class operator ThourCursor.Finalize (var ADest: THourCursor);
begin
  Screen.Cursor := ADest.FCurrCur;
end;
```

定义此托管记录后

```
var HC: THourglassCursor;
// 一些缓慢的操作
Sleep (5000);
```

❖ 您可以在以下由 Erik van Bilsen 撰写的博客文章中找到更多通过托管记录进行资源保护的示例：https://blog.grijjy.com/2020/08/03/automaterestorable-operations-with-custom-managed-records/。
这是有关托管记录的一系列非常详细的博客的一部分。

9.3 现实世界中的异常

异常是一种很好的错误报告和错误处理的绝妙机制（不是在单个代码片段中，而是作为更大体系结构的一部分）。

通常，异常不应替代检查局部错误条件（尽管某些开发人员以这种方式使用它们）。

例如，如果您不确定文件名，通常认为比在任何情况下都使用异常来打开文件的情况下打开文件要好，该方法通常比使用异常来打开文件更好。但是，在写入文件之前检查是否还有足够的磁盘空间是一种检查，这种检查在所有地方都没有意义，因为这是极为罕见的情况。

一种表达方式是，程序应检查常见的错误情况，并将异常情况和意外情况留给异常处理机制。当然，这两种情况之间的界线通常是模糊的，并且不同的开发人员将有不同的判断方法。

总是使用异常的地方是让不同的类和模块相互传递错误条件。与使用异常相比，返回错误代码非常繁琐且容易出错。在组件或库类中引发异常比在事件处理

程序中引发异常更为常见。您最终可以编写很多代码，而不会引发或处理异常。

相反，在每天的代码中，非常重要且非常常见的是在发生异常的情况下使用 finally 块来保护资源。您应该始终使用 finally 语句来保护引用外部资源的块，以免在引发异常时资源泄漏。每次您在单个函数或方法中打开和关闭，连接和断开连接，创建和销毁某些对象时，都需要 finally 语句。最终，即使在引发异常的情况下，finally 语句也可以使程序保持稳定，从而使用户可以继续使用或（如果存在更重要的问题）有序关闭应用程序。

9.4 Global（全局）异常处理

如果事件处理程序引发的异常停止了标准执行流程，并且在找不到异常处理程序的情况下也会终止程序吗？对于控制台应用程序或其他特殊用途的代码结构，确实是这种情况，而大多数可视化应用程序（包括基于 VCL 或 FireMonkey 库的那些可视化应用程序）都具有全局消息处理循环，该循环将每次执行包装在 try-except 块中，以便如果事件处理程序中引发了异常，则会被捕获。

请注意，如果在激活消息循环之前在启动代码中引发了异常，则这些异常通常不会被库捕获，并且程序将简单地终止并出现错误。通过在主程序中添加一个自定义的 try-except 块，可以部分缓解此行为。静态库初始化代码将在执行主程序和启动自定义 try-except 块之前进行。

在执行过程中引发异常的一般情况下，发生的情况取决于库，但是通常存在一种编程的方法，可以使用全局处理程序来拦截这些异常，或者显示错误消息的方法。尽管某些细节有所不同，但 VCL 和 FireMonkey 都是如此。在先前的演示中，您看到了引发异常时显示的简单错误消息。

如果要更改该行为，则可以处理全局 Application 对象的 OnException 事件。尽管此操作更多地与应用程序的可视库和事件处理有关，但它也与异常处理相关，因此值得在此进行介绍。

我已经采用了之前的应用程序项目，称为 ErrorLog，并且在主窗体中添加了新方法：

```
public
  procedure LogException (Sender: TObject; E: Exception);
```

在 OnCreate 事件处理程序中，我添加了将方法挂钩到全局 OnException 事件的代码，然后，我编写了全局处理程序的实际代码：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := LogException;
end;

procedure TForm1.LogException(Sender: TObject; E: Exception);
begin
  Show('Exception ' + E.Message);
end;
```

❖ 在下一章中，您将学习如何为事件分配方法指针的详细信息（就像我上面所做的一样）。

使用全局异常处理程序中的新方法，程序将错误消息写入输出，而不用错误消息停止应用程序。

9.5 异常和构造函数

关于异常，还有一个更高级的问题，即在对象的构造函数中引发异常时会发生什么情况。并非所有的 Pascal Pascal 程序员都知道在这种情况下将调用该对象的析构函数（如果有）。

要知道这一点很重要，因为这意味着可能会对部分初始化的对象调用析构函数。认为内部对象存在于构造函数中是理所当然的，因为内部对象是在构造函数中创建的，如果您遇到实际错误（即在处理第一个异常之前引发另一个异常），可能会使您陷入某些危险情况。

这也意味着，尝试最终的正确顺序应包括在块外部创建对象，因为该对象由编译器自动保护。

因此，如果构造函数失败，则无需释放对象。这就是为什么 Object Pascal 中的标准编码风格是通过编写以下内容来保护对象的原因：

```
AnObject := AClass.Create;
try
  // 使用对象...
finally
  AnObject.Free;
end;
```

❖ 对于 TObject 类的两个特殊方法，AfterDestruction 和 BeforeConstruction，也发生了类似的事情，它们是为实现 C ++兼容性而引入的伪构造函数和伪析构函数（但很少在 Object Pascal 中使用）。注意，如果 AfterConstruction 方法引发异常，则调用 BeforeDestruction 方法（以及常规析构函数）。

鉴于我经常看到在销毁器中正确放置对象的错误，因此让我通过展示问题的实际演示程序以及实际修复程序进一步阐明问题。假设您有一个包含字符串列表的类，并且编写了以下代码来创建和销毁该类（ConstructorExcept 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;
```

乍一看，该代码似乎是正确的。构造函数正在分配子对象，而析构函数正在正确地处理它。而且，调用代码的编写方式是，如果在构造函数之后引发异常，

则调用 Free 方法，但是如果异常在构造函数中，则什么都不会发生：

```
var
  Obj: TObjectWithList;
begin
  Obj := TObjectWithList.Create (-10);
  try
    // do something
  finally
    Show ('Freeing object');
    Obj.Free;
  end;
```

那行吗？绝对不！当涉及此代码时，在创建字符串列表之前，构造函数中会引发异常，并且系统会立即调用析构函数，该析构函数将尝试清除不存在的列表，从而引发访问冲突或类似错误。

为什么会这样？同样，如果您在构造函数中颠倒顺序（首先创建字符串列表，然后引发异常），则所有操作都将正常运行，因为析构函数确实需要释放字符串列表。但这不是真正的解决办法，只能解决。您应该始终考虑的是，以一种从不假定构造函数已完全执行的方式来保护析构函数的代码。这是一个例子：

```
destructor TObjectWithList.Destroy;
begin
  if Assigned (FStringList) then
  begin
    FStringList.Clear;
    FreeAndNil(FStringList);
  end;
  inherited;
end;
```

9.6 异常的高级功能

这是本书的其中一部分，您可能在第一次阅读时可能会略过，除非您已经对该语言有很好的了解。

您可以转到下一章，以后再回到本节。

在本章的最后部分，我将介绍一些与异常处理相关的更高级的主题。我将介绍嵌套异常（RaiseOuterException）和拦截类的异常（RaisingException）。这些功能不是 ObjectPascal 语言的早期版本的一部分，而是为系统增加了强大的功能。

9.6.1 嵌套异常和 InnerException 机制

如果在异常处理程序中引发异常会怎样？传统的答案是，新异常将替换现有异常，这就是为什么通常的做法是至少合并错误消息，编写这样的代码（缺少任何实际操作并仅显示与异常相关的语句）：

```
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;
```

此代码是 AdvancedExcept 应用程序项目的一部分。调用方法并处理异常时，您将看到一个包含合并消息的异常：

```
procedure TFormExceptions.BtnTraditionalClick(Sender: TObject);
begin
  try
    ClassicReraise;
  except
    on E: Exception do
      Show ('Message: ' + E.Message);
 end;
end;
```

（非常明显的）输出是：

```
Message: Another: Hello
```

现在在 Object Pascal 中，系统范围内对嵌套异常的支持。在异常处理程序中，您可以创建并引发新的异常，并且仍然使当前异常对象保持活动状态，并将其连接到新的异常。为实现此目的，Exception 类具有一个 InnerException 属性（引用先前的异常）和 BaseException 属性，该属性使您可以访问一系列的第一个异常，因为异常嵌套可以递归进行。这些是与嵌套异常管理相关的 Exception 类的元素：

```
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;
```

❖ 静态类方法是类方法的一种特殊形式。该语言功能将在第 12 章中进行说明。

从用户的角度来看，要在保留现有异常的同时引发异常，应调用 RaiseOuterException 类方法（或相同的 ThrowOuterException，它使用面向 C++的命名）。处理类似的异常时，可以使用新属性访问更多信息。请注意，您只能在异常处理程序中调用 RaiseOuterException，这是基于源代码的文档告诉您的：

使用此函数可以从异常处理程序中引发异常实例，并且您要“acquire（获取）”活动的异常并将其链接到新的异常并保留上下文。这将导致 FInnerException 字段设置为当前正在播放的异常。

您只应从 except 块内调用此过程，否则该新异常将在其他地方处理。

有关实际示例，您可以参考 AdvancedExcept 应用程序项目。在此示例中，我添加了一种以新方式引发嵌套异常的方法（与之前列出的 ClassicReraise 方法相比）：

```
procedure TFormExceptions.MethodWithNestedException;
begin
  try
    raise Exception.Create ('Hello');
  except
    Exception.RaiseOuterException (Exception.Create ('Another'));
```

```
      end;
   end;
```

现在，在此外部异常的处理程序中，我们可以访问两个异常对象（并且还可以看到调用新 ToString 方法的效果）：

```
      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;
```

此调用的输出如下：

```
   Message: Another
   ToString: Another
   Hello
   BaseException Message: Hello
   InnerException Message: Hello
```

有两个相关的元素需要注意。第一个是在单个嵌套异常的情况下，BaseException 属性和 InnerException 属性都引用相同的异常对象，即原始对象。第二个是，虽然新异常的消息仅包含实际的消息，但通过调用 ToString，您可以访问所有嵌套异常的组合消息，并以 sLineBreak 分隔（如方法代码所示 Exception.ToString）。

在这种情况下，选择使用换行符会产生看起来很奇怪的输出，但是一旦了解了它，就可以按照自己喜欢的方式对其进行格式化，将换行符替换为您选择的符号，或者将它们指定给字符串列表的 Text 属性。

再举一个例子，让我向您展示引发两个嵌套异常的情况。这是新方法：

```
   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;
```

这称为一种方法，该方法与我们之前看到的方法相同，并产生以下输出：

```
   Message: A third
   ToString: A third
   Another
   Hello
   BaseException Message: Hello
   InnerException Message: Another
```

这次，BaseException 属性和 InnerException 属性引用了不同的对象，并且 ToString 的输出跨越了三行。

9.6.2 Intercepting（拦截）异常

随着时间的推移，Object Pascal 语言的原始异常处理系统中增加了另一个高级功能，该方法是：

procedure RaisingException(P: PExceptionRecord); virtual;

根据源代码文档：

在即将引发此异常之前，将立即调用此虚函数。在外部异常的情况下，由于“raise（引起）”条件已经在进行，因此在对象创建后立即调用此方法。

Exception 类中函数的实现管理内部异常（通过调用内部 SetInnerException），这很可能解释了为什么首先将其与内部异常机制同时引入的原因。

无论如何，现在有了此功能，我们就可以利用它。实际上，通过重写此方法，无论创建异常的构造方法如何，我们都将始终调用单个创建函数。换句话说，可以避免为异常类定义自定义构造函数，而让用户调用基类 Exception 的许多构造函数之一，并且仍然具有自定义行为。例如，您可以记录给定类（或子类）的任何异常。

这是一个自定义异常类（在 AdvancedExcept 应用程序项目中再次定义），该类重写 RaisingException 方法：

```
type
  ECustomException = class (Exception)
  protected
    procedure RaisingException(P: PExceptionRecord); override;
  end;

procedure ECustomException.RaisingException(P: PExceptionRecord);
begin
  // 记录异常信息
  FormExceptions.Show('Exception Addr: ' + IntToHex (
    Integer(P.ExceptionAddress), 8));
  FormExceptions.show('Exception Mess: ' + Message);
  // 修改消息
  Message := Message + ' (filtered)';
  // 标准加工
  inherited;
end;
```

此方法实现的作用是记录有关例外的一些信息，修改异常消息，然后调用基类的标准处理（需要嵌套异常机制起作用）。在创建异常对象之后但在引发异常之前调用该方法。可以注意到这是因为 Show 调用产生的输出是在调试器捕获异常之前生成的！同样，如果在 RaisingException 方法中放置一个断点，调试器将在捕获异常之前在此停止。

同样，嵌套异常和这种拦截机制在应用程序代码中并不常用，因为它们是更适合库和组件开发人员的语言功能。

第十章 属性和事件

在最后三章中，我介绍了 Object Pascal 中 OOP 的基础，解释了这些概念，并说明了如何具体实现大多数面向对象语言中可用的功能。自 Delphi 成立以来，Object Pascal 语言是一种完全面向对象的语言，但具有特定的风味。实际上，它还成为基于组件的可视化开发工具的语言的两倍。

这些并不是互不相干的功能：此开发模型的支持基于某些核心语言功能（例如属性和事件），这些功能最初是在 Object Pascal 之前于任何其他语言引入的，后来又被一些 OOP 语言部分复制。例如，属性可以在 Java 和 C＃等语言中找到，但是它们确实与 Object Pascal 有直接的渊源……尽管我个人更喜欢原始的实现，我将在稍后解释。

Object Pascal 支持快速应用程序开发（RAD）和可视化编程的能力是诸如属性，已发布的访问规范，事件，组件的概念以及本章涵盖的其他一些概念之类的概念的原因。

10.1 定义 property（属性）

什么是 property（属性）？属性可以描述为标识符，这些标识符使您可以访问和修改对象的状态，这可能会触发后台代码执行。在 Object Pascal 中，属性通过字段或方法抽象和隐藏数据访问，使其成为封装的主要实现。描述属性的一种方法是“最大 encapsulation（封装）”。

从技术上讲，属性是具有某种数据类型的标识符，该标识符使用某些读写说明符映射到某些实际数据。与 Java 或 C＃不同，在 Object Pascal 中，读写说明符可以是 getter 或 setter 方法，也可以是字段。

例如，以下是使用通用方法（从字段读取，使用方法写入）的日期对象的属性定义：

```
private
  FMonth: Integer;
  procedure SetMonth(Value: Integer);
public
  property Month: Integer read FMonth write SetMonth;
```

要访问 Month 属性的值，此代码必须读取专用字段 FMonth 的值，而要更改该值，它调用方法 SetMonth。更改值（防止出现负值）的代码可能类似于：

```
procedure TDate.SetMonth (Value: Integer);
begin
  if Value <= 0 then
    FMonth := 1
  else
    FMonth := Value;
end;
```

❖ 在输入不正确的情况下，例如负的月份数，通常最好显示一个错误（通过引发异常），而不是在后台调整值，但是为了简单起见，我将代码保持原样演示。

请注意，字段和属性的数据类型必须完全匹配（如果存在差异，则可以使用简单的方法进行转换）；类似地，setter 过程的单个参数的类型或 getter 函数的返回值必须与属性类型完全匹配。

可能有不同的组合（例如，我们也可以使用一种方法来读取值或直接更改

write 指令中的字段），但是使用一种方法来更改属性的值是最常见的。这是相同属性的一些替代实现：

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;
```

❖ 当您编写访问属性的代码时，重要的是要意识到可以调用一个方法。问题是其中一些方法需要一些时间才能执行。它们还会产生许多副作用，通常包括（缓慢）重新绘制屏幕上的控件。尽管很少记录属性的副作用，但是您应该意识到它们的存在，尤其是在尝试优化代码时。

属性的 write 指令也可以省略，从而使该属性成为只读属性：

```
property Month: Integer read GetMonth;
```

从技术上讲，您也可以省略 read 指令并定义一个只写属性，但这通常没有多大意义，而且很少实现。

10.1.1 与其他编程语言相比的属性

如果将其与 Java 或 C＃进行比较，则两种语言中的属性都映射到方法，但是前者具有隐式映射（属性基本上是一种约定），而后者则具有诸如 Object Pascal 的显式映射，即使仅映射到方法：

```
// Java 语言中的属性
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);

// C＃语言中的属性
private int mMonth;
public int Month {
    get { return mMonth; }
    set {
        if (value <= 0)
            mMonth = 1;
        else
            mMonth = value;
    }
}
date.Month++;
```

并不是说我不会深入讨论各种编程语言中属性的相对优点，而是如本章引言中所述，我认为拥有明确定义的属性是一个有用的想法，而且通过将属性映射到字段而获得的抽象级别，而没有方法的额外负担，这是一个很好的补充。因此，与其他语言相比，我更喜欢 Object Pascal 属性的实现。

属性是一种非常完善的 OOP 机制，是封装思想的一种经过深思熟虑的应用。本质上，您的名称隐藏了如何访问类信息的实现（直接访问数据或调用方法）。

实际上，通过使用属性，您最终会获得一个不太可能更改的界面。

同时，如果只希望允许用户访问类的某些字段，则可以轻松地将这些字段包

装为属性，而不是将其公开。您无需再编写其他代码（对简单的 Get 和 Set 方法进行编码非常无聊），并且您仍然可以更改类的实现。即使您将直接数据访问替换为基于方法的访问，也不必完全更改使用这些属性的源代码。您只需要重新编译即可。

认为这是将封装概念提升到最大功率的方法！

❖ 您可能想知道，如果定义了一个可以直接访问私有变量的属性，这是否消除了封装的优点之一？无法保护用户防止私有变量的数据类型发生任何更改，而可以使用 getter 和 setter 来保护用户。但是，由于用户将通过该属性访问数据，因此类开发人员可以随时更改基础数据类型并引入 getter 和 setter，而不会影响使用它的代码。这就是为什么我将这种封装称为“最大封装”。另一方面，这表明了 Object Pascal 的实用性，因为它允许程序员选择适合情况的任何更简便的方法（和快速的代码执行），并顺利过渡到“适当的 OOP”何时需要的方式。

不过，在 Object Pascal 中使用属性有一个警告。通常，您可以为属性分配值或读取它，并且可以随意使用属性。但是，您不能将属性作为参考参数传递给过程或方法。这是因为属性不是存储器位置，而是抽象，因此不能将其用作参考（var）参数。例如，与 C＃不同，您不能在属性上调用 Inc。

❖ 本章后面将介绍一个相关的功能，该功能通过引用传递属性。但是，它是一个很少使用的功能，需要启用特定的编译器设置，并且肯定不是主流功能。

10.1.2 给属性编码

如果向类中添加属性似乎繁琐的工作，则在编写属性声明的初始部分（在类内）时，IDE 的编辑器可让您轻松自动完成属性，如下所示：

```
type
  TMyClass = class
  public
    property Month: Integer;
  end;
```

按下 Ctrl + Shift + C 组合键，同时将光标移到属性声明上，您将获得一个新字段以及一个新的 setter 方法，以及在属性定义和完整实现中的正确映射，从而将新字段添加到类中。setter 方法的功能，并使用基本代码更改字段值。换句话说，上面使用键盘快捷键（或编辑器的本地菜单的匹配项）的代码变为：

```
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;
```

您是否还需要 getter 方法，请用 GetMonth 替换定义的读取部分，例如：

```
    property Month: Integer read GetMonth write SetMonth;
```

现在再次按 Ctrl + Shift + C，也会添加该函数，但是这次没有预定义的代码来访问该值：

```
function TMyClass.GetMonth: Integer;
begin

end;
```

10.1.3 向表单添加属性

让我们来看一个使用属性封装的特定示例。这次不是构建自定义类，而是要修改 IDE 为您创建的每个可视表单生成的表单类...并且我还将利用"类完成"功能。

当应用程序具有多个表单时，通常能够方便地从一个表单访问另一个表单的信息。您可能会想添加一个公共字段，但这总是一个坏主意。每次您希望将某个表单的某些信息提供给其他表单时，您应该使用一个属性。

只需在表单类声明中编写属性名称和类型：

```
property Clicks: Integer;
```

然后按 Ctrl + Shift + C 激活代码完成。您会看到以下效果：

```
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，这可以节省您很多键入时间。现在，当用户单击表单时，您可以通过编写以下行来增加点击计数，就像我在 FormProperties 应用程序项目表单的 OnMouseDown 事件中所做的那样：

```
Clicks := Clicks + 1;
```

您可能会想，直接增加 FClicks 怎么办？好了，在这种特定情况下可能会起作用，但是您也可以使用 SetClicks 方法来更新用户界面并实际显示当前值。如果绕过该属性并直接访问该字段，则将不会执行用于更新用户界面的 setter 方法中的其他代码，并且显示可能会不同步。

这种封装的另一个优点是，您可以通过另一个表单以适当抽象的方式引用点击次数。实际上，表单类中的属性可用于访问自定义值，但也可用于封装对表单组件的访问。例如，如果您有一个带有用于显示某些信息的标签的表单，并且想要修改辅助表单中的文本，则可能会尝试编写：

```
Form1.StatusLabel.Text := 'new text';
```

这是一种常见的做法，但是它不是一个好方法，因为它没有提供对表单结构或组件的任何封装。如果您在整个应用程序中的许多地方都有类似的代码，而后来决定修改表单的用户界面（用另一个控件替换 StatusLabel 对象），则必须在许多地方修复该代码。

另一种选择是使用一种方法甚至更好的一种属性来隐藏特定控件。您可以按照上述步骤添加具有读写方法的属性，或像下面这样完整键入它们：

```
property StatusText: string read GetStatusText write SetStatusText;
```

然后再次按 Ctrl + Shift + C 组合键，以使编辑器添加用于读取和写入属性的两种方法的定义：

```
function TFormProp.GetStatusText: string;
begin
  Result := LabelStatus.Text
end;

procedure TFormProp.SetStatusText(const Value: string);
begin
  LabelStatus.Text := Value;
end;
```

请注意，在这种情况下，该属性未映射到类的本地字段，而是在子对象的字段中映射了标签（如果使用了自动代码生成，请记住实际上要删除编辑器可能具有的 FStatusText 属性。代表您添加）。

在程序的其他窗体中，您可以简单地引用窗体的 StatusText 属性，并且如果用户界面发生更改，则仅影响该属性的 Set 和 Get 方法。您甚至还可以在原始表单中执行相同的操作，从而使两个属性的代码更加独立：

```
procedure TFormProp.SetClicks(const Value: Integer);
begin
  FClicks := Value;
  StatusText := FClicks.ToString + ' clicks';
end;
```

10.1.4 将属性添加到 TDate 类

在第 7 章中，我构建了 TDate 类。现在我们可以使用属性来扩展它。这个新的应用程序项目 DateProperties 基本上是第 7 章的 ViewDate 应用程序项目的扩展。这是该类的新声明。它具有一些新方法（用于设置和获取属性值）和四个属性：

```
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 和 Month 属性使用相应的方法读取和写入其值。这是与“Month”属性相关的两个：

```
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 ('无效月份');
  SetValue (Year, Value, Day);
end;
```

对 SetValue 的调用执行日期的实际编码，如果发生错误，则会引发异常。 我定义了一个自定义异常类，每次值超出范围时都会引发该异常类：

```
type
  EDateOutOfRange = class (Exception);
```

第四个属性，Text，仅映射到 read 方法。该函数被声明为虚拟函数，因为它已被 TNewDate 子类替代。没有理由不应该使用属性的 Get 或 Set 方法使用后期绑定（第 8 章中将详细介绍的功能）。

在此示例中要认识到的重要一点是，这些属性不会直接映射到数据。它们是简单地根据与属性似乎暗示的类型和结构不同的信息存储的。

用新的属性更新了类之后，我们现在可以更新示例以在适当的时候使用属性。例如，我们可以直接使用 Text 属性，还可以使用一些编辑框让用户读取或写入三个主要属性的值。当按下 BtnRead 按钮时，会发生这种情况：

```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
  EditYear.Text := IntToStr (TheDay.Year);
  EditMonth.Text := IntToStr (TheDay.Month);
  EditDay.Text := IntToStr (TheDay.Day);
end;
```

BtnWrite 按钮执行相反的操作。您可以通过以下两种方式之一编写代码：

```
  // 直接使用属性
  TheDay.Year := StrToInt (EditYear.Text);
  TheDay.Month := StrToInt (EditMonth.Text);
  TheDay.Day := StrToInt (EditDay.Text);
  // 一次更新所有值
  TheDay.SetValue (StrToInt (EditMonth.Text),
    StrToInt (EditDay.Text),
    StrToInt (EditYear.Text));
```

两种方法之间的差异与输入不符合有效日期时发生的情况有关。当我们分别设置每个值时，程序可能会更改年份，然后引发异常并跳过执行其余代码的过程，因此日期仅被部分修改。当我们一次设置所有值时，要么正确且已全部设置，要么一个无效，并且 date 对象保留原始值。

10.1.5 使用数组属性

属性通常使您可以访问单个值，即使是复杂数据类型之一也是如此。

Pascal 对象还定义了数组属性或在 C# 中调用的索引器。数组属性是具有任何数据类型的进一步参数的属性，该参数用作实际值的索引或（通常）选择器。

这是使用 Integer 索引并引用整数值的数组属性的定义示例：

```
private
  function GetValue(I: Integer): Integer;
  procedure SetValue(I: Integer; const Value: Integer);
public
  property Value [I: Integer]: Integer read GetValue write SetValue;
```

必须将数组属性映射为具有表示索引的额外参数的读写方法...，并且可以像常规属性一样使用“代码完成”来定义方法。值和索引有许多组合，并且 RTL 中的一些类大量使用了数组属性。例如，TStrings 类定义了其中的 5 个：

```
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;
```

虽然大多数这些数组属性使用字符串的索引作为列表中的参数，但其他属性使用字符串作为查找或搜索值（例如上面的 Values 属性）。

这些定义的最后一个使用另一个重要功能：用默认关键字标记。这是一个强大的语法 helper（助手）：可以省略数组属性的名称，以便您可以将方括号运算符直接应用于所涉及的对象。因此，如果您具有此 TStrings 类型的对象 SList，则以下两个语句将读取相同的值：

```
SList.Strings[1]
SList[1]
```

换句话说，默认数组属性提供了一种为任何对象定义自定义[]运算符的方法。

10.1.6 通过引用设置属性

如果您没有 Object Pascal 的经验，那么这是一个相当高级的主题（并且使用了很少的功能），您应该跳过。但是，如果您是，则有可能您从未听说过此功能。

在扩展 Object Pascal 编译器以直接支持 Windows COM 编程时，它具有处理“put by ref（按引用放置）”属性（以 COM 行话表示）或可以接收引用而不是值的属性的能力。

- ❖ 克里斯·本森（Chris Bensen）在博客中介绍此功能时使用了“ Put by ref”这个名称：http://chrisbensen.blogspot.com/2008/04/delphi-put-by-ref-properties.html（克里斯在 时间是该产品的研发工程师）。

这是通过在 setter 方法中使用 var 参数来实现的。鉴于这可能导致相当尴尬的情况，该功能（尽管仍是语言的一部分）被认为是一个例外而不是一条规则，

这就是为什么默认情况下该功能不处于活动状态的原因。

换句话说，要启用此功能，您必须使用 compiler 指令专门要求它：

```
{$VARPROPSETTER ON}
```

如果没有此指令，以下代码将无法编译，并会发出错误“E2282 Property setters cannot take var parameters（E2282 属性设置器无法采用 var 参数）”：

```
type
  TMyIntegerClass = class
  private
    FNumber: Integer;
    function GetNumber: Integer;
    procedure SetNumber(var Value: Integer);
  public
    property Number: Integer
    read GetNumber write SetNumber;
  end;
```

此类是 VarProp 应用程序项目的一部分。现在很奇怪的是，您在属性设置器中可能会产生副作用：

```
procedure TMyIntegerClass.SetNumber(var Value: Integer);
begin
  Inc (Value); //  副作用
  FNumber := Value;
end;
```

另一个非常不寻常的效果是，您不能给属性分配一个常量值，而只能给它分配一个变量（应该是预期的，就像任何涉及通过引用传递参数的调用一样）：

```
var
  Mic: TMyIntegerClass;
  N: Integer;
begin
  ...
  Mic.Number := 10; //  错误：E2036 需要变量
  Mic.Number := N;
```

虽然这不是您经常使用的功能，但这是一种考虑属性的高级方法，可让您初始化或更改分配给它的值。这可能会导致极其奇怪的代码，例如：

```
N := 10;
Mic.Number := N;
Mic.Number := N;
Show(Mic.Number.ToString);
```

两个连续的相同分配看起来很奇怪，但是它们确实会产生副作用，将实际数字变为 12。这可能是获得该结果最费解和无意义的方法！

10.2 Published（发布的）访问说明符

与 public（公共），prodected（受保护）和 private（私有）访问指令（以及较少使用的 strict private（严格私有）和 strict protected（严格受保护）指令一起，Object Pascal 语言还有另一种非常特殊的语言，称为 published（发布）指令。已发布的属性（或字段或方法）不仅像公共属性一样在运行时可用，而且会生成可查询的扩展运行时信息（RTTI）。

实际上，在编译语言中，编译符号由编译器处理，并且可以由调试器在测试应用程序时使用，但通常在运行时不会留下任何痕迹。换句话说（至少在 Object Pascal 的早期），如果类具有名为 Name 的属性，则可以在代码中使用它与该类

进行交互，但是您无法确定类是否具有与之匹配的属性给定的字符串“Name（名称）”。

❖ Java 和 C＃语言都是经过编译的语言，它们利用了复杂的虚拟执行环境，因此它们具有广泛的运行时信息，通常称为反射。几年后，Object Pascal 语言引入了反射（也称为扩展 RTTI），因此它既具有与本章中探讨的已发布关键字相关的一些基本 RTTI，也具有第 16 章中涉及的更全面的反射形式。

为什么需要有关类的这些额外信息？它是 Object Pascal 库所依赖的组件模型和可视化编程模型的基础之一。在开发环境中的设计时会使用某些信息，以用组件提供的属性列表填充“对象检查器”。这不是一个硬编码的列表，而是通过编译后的代码的运行时检查生成的。

另一个示例，可能有点太复杂，以至于现在不能完全研究，它是创建 FMX 和 DFM 文件以及任何附带的可视形式背后的流传输机制。流将仅在第 18 章中介绍，因为它是运行时库的一部分，而不是核心语言的一部分。

总而言之，在编写要由您自己或其他人在开发环境中使用的组件时，定期使用发布的关键字很重要。通常，组件的已发布部分仅包含适当的联系，而表单类也使用已发布的字段和方法，这将在稍后介绍。

10.2.1 Design-Time（设计时）属性

我们在本章前面已经看到，属性对于封装类数据起着重要作用。它们在启用可视化开发模型中也起着基本作用。实际上，您可以编写一个组件类，使其在开发环境中可用，通过将其添加到表单或类似的设计图面上来创建对象，并通过对象检查器与其属性进行交互。

在这种情况下，并非所有属性都可以使用，只有那些在组件类中标记为 published（发布的）属性才可以使用。这就是为什么 Object Pascal 程序员在设计时属性和仅运行时属性之间进行区分的原因。

设计时属性是在类声明的已发布部分中声明的那些属性，可以在设计时在 IDE 和代码中使用。在类的公共部分声明的任何其他属性在设计时不可用，而仅在代码中可用，并且通常仅称为运行时。

换句话说，您可以在设计时使用对象检查器查看值并更改已发布属性的值。这是可视化编程环境提供的工具，可让您访问属性。在运行时，您可以通过在代码中读取或写入其值来以完全相同的方式访问另一个类的任何公共或发布的属性。

并非所有的类都具有属性。属性存在于 TPersistent 类的组件和其他子类中，因为通常可以流式传输属性并将其保存到文件中。实际上，表单文件不过是表单上组件的已发布属性的集合。

更准确地说，您不需要继承 TPersistent 来支持已发布部分的概念，而是需要使用$M 编译器指令编译一个类。使用该指令编译的每个类或从使用该指令编译的类派生的每个类都支持发布的部分。给定 TPersistent 使用此设置编译，则任何派生类均具有此支持。

❖ 以下有关默认可见性和自动 RTTI 的两部分为$ M 指令和 Published 关键字的效果添加了更多信息。

10.2.2 Published（发布的） and Forms（表单）

IDE 生成表单时，会将其组件和方法的定义放在其定义的初始部分中，将其置于公用关键字和专用关键字之前。该类的初始部分的这些字段和方法已发布。

当在组件类的元素之前未添加任何特殊关键字时，将发布默认值。

❖ 更确切地说，仅当使用$ M +编译器指令编译该类或该类是使用$ M +编译的类的后代时，publish 是默认关键字。由于在 TPersistent 类中使用了此伪指令，因此库的大多数类和所有组件类都默认为发布。但是，非组件类（例如 TStream 和 TList）使用$ M-编译，并且默认为公共可见性。

这是一个例子：

```
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    BtnTest: TButton;
```

分配给任何事件的方法应该是发布的方法，并且应该发布与表单中的组件相对应的字段，以自动与表单文件中描述的对象连接并与表单一起创建。在表单声明的最初发布部分中，只有组件和方法可以显示在对象检查器中（在表单的组件列表中，或者在选择事件的下拉列表时显示的可用方法的列表中）。

为什么类的组件应该用发布字段声明，而它们却可以是私有的并更好地遵循 OOP 封装规则？原因在于这些组件是通过读取其流表示形式创建的，但是一旦创建它们，就需要将它们分配给相应的表单字段。

这是使用为已发布字段生成的 RTTI 完成的（在引入第 16 章介绍的扩展 RTTI 之前，它最初是 Object Pascal 中可用的唯一 RTTI 类型）。

❖ 从技术上讲，使用已发布的字段作为组件并不是强制性的。您可以通过将代码设为私有来使代码更加熟悉 OOP。但是，这确实需要额外的运行时代码。我将在本章的最后一部分“ RAD 和 OOP”中对此进行更多解释。

10.2.3 自动 RTTI

Object Pascal 编译器的另一个特殊行为是，如果将 published 关键字添加到不继承自 TPersistent 的类，则编译器将自动启用 RTTI 生成，并自动添加{$ M +}行为。

假设您有此类：

```
type
  TMyTestClass = class
  private
    FValue: Integer;
    procedure SetValue(const Value: Integer);
  published
    property Value: Integer read FValue write SetValue;
  end;
```

编译器显示如下警告：

[dcc32 Warning] AutoRTTIForm.pas(27): W1055 PUBLISHED caused RTTI ($M+) to be added to type 'TMyTestClass'

（[dcc32 警告] AutoRTTIForm.pas（27）：W1055 发布导致将 RTTI（$ M +）添加到类型'TMyTestClass'）

发生的是，编译器会自动在代码中注入{$ M +}指令，如您在 AutoRTTI 应用程序项目中所看到的那样，其中包括上面的代码。在此程序中，您可以编写以下代码，该代码动态地访问属性（使用老式的 System.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;
```

尽管我偶尔会使用 TypInfo 单元以及其中定义的诸如 GetPropValue 之类的功能，但更现代的RTTI单元及其对反射的广泛支持赋予了RTTI访问的真正力量。考虑到这是一个相当复杂的主题，我觉得有必要专门为其单独介绍一章，并区分 Object pascal 支持的两种 RTTI 风格。

10.3 事件驱动编程

在基于组件的库中（而且在许多其他场景中），您编写的代码不仅是固定的动作序列，而且大部分是反应的集合。我的意思是，您定义应用程序在发生某些事情时应如何"反应"。这种"东西"可以是用户操作，例如单击按钮，系统操作，传感器状态更改，通过远程连接可用的某些数据或几乎其他任何内容。

这些外部或内部动作触发通常称为事件。事件最初是诸如 Windows 之类的面向消息的操作系统的映射，但是自最初的概念以来已经走了很长一段路。实际上，在现代库中，大多数事件是在设置属性，调用方法或与给定组件（或间接与另一个组件）交互时在内部触发的。

事件和事件驱动的编程与 OOP 有何关系？与使用更通用的类相比，这两种方法在何时以及如何创建新的继承类方面有所不同。

在一种纯粹的面向对象编程表单中，只要一个对象的行为（或方法）与另一个不同，就应属于不同的类。我们已经看到了一些演示。

现在让我们考虑这种情况。一个表单有四个按钮。单击每个按钮时，其行为均不同。因此，以纯 OOP 术语来说，您应该有四个不同的按钮子类，每个子类具有不同版本的"click"方法。这种方法在表单上是正确的，但是将需要大量额外的代码来编写和维护，从而增加了复杂性。

事件驱动的编程考虑了类似的情况，并建议开发人员向属于同一类的按钮对象添加某些行为。该行为成为对象状态的修饰或扩展，而无需新的类。该模型也称为委托，因为对象的行为委托给该对象自身类以外的其他类的方法。

事件是通过不同的编程语言以不同的方式实现的，例如：

- 使用对方法的引用（在 Object Pascal 中称为方法指针）或对具有内部方法的事件对象的引用（在 C＃中发生）。
- 将事件代码委托给实现接口的专业类（就像 Java 中通常发生的那样）。
- 使用类似的闭包通常在 JavaScript 中发生（ObjectPascal 也支持匿名方法，第 15 章将介绍这种方法），尽管在 JavaScript 中所有方法都是闭包，因此两种语言之间的区别在该语言中有些模糊。

事件和事件驱动的编程的概念已变得非常普遍，并且得到许多不同的编程语言和用户界面库的支持。但是，Delphi 实现事件支持的方式非常独特。

下一节将详细说明其背后的技术。

10.3.1 方法指针

在第 4 章的最后部分中，我们看到了该语言具有函数指针的概念。这是一个保存函数的内存位置的变量，您可以使用它间接调用该函数。函数指针用特定的签名声明（作为参数类型和返回类型的集合，如果有的话）。

同样，该语言具有方法指针的概念。方法指针是对属于类的方法的内存位置的引用。与函数指针类型一样，方法指针类型具有特定的签名。但是，方法指针会携带更多信息，即该方法将应用到的对象（或在调用该方法时将用作 Self 参数的对象）。

换句话说，方法指针是对内存中一个特定对象的方法（在特定的内存地址处）的引用。在为方法指针分配值时，必须引用给定对象的方法，即特定实例的方法！

❖ 如果查看通常在底层使用的表示此构造的数据结构的定义（称为 TMethod），则可以更好地理解方法指针的实现。

❖ 该记录有两个字段“Code（代码）”和“Data（数据）”，分别表示方法地址和将应用于的对象。在其他类似语言中，代码引用由委托类（C#）或接口方法（Java）捕获。

方法指针类型的声明与过程类型的声明相似，不同之处在于它在声明的末尾具有 of object（对象）关键字：

```
type
   IntProceduralType = procedure (Num: Integer);
   TStringEventType = procedure (const S: string) of object;
```

声明了方法指针（例如上面的方法指针）后，可以声明此类型的变量，并为它分配任何对象的兼容方法。 什么是兼容方法？具有与方法指针类型请求的参数相同的参数的参数，例如上例中的单个字符串参数。

❖ 可以将对任何对象的方法的引用分配给方法指针，只要它与方法指针类型兼容即可。

现在您有了方法指针类型，您可以声明此类型的变量并为其分配兼容的方法：

```
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;
```

现在，这个简单的代码并没有真正解释事件的用处，因为它专注于低级方法指针类型的概念。事件基于此技术实现，但超出了此范围，方法是将方法指针存储在一个对象（例如，一个按钮）中，以引用另一个对象的方法（例如，具有按钮的 OnClick 处理程序的表单）。在大多数情况下，也可以使用适当的联系来实现事件。

尽管它不那么常见，但是在 Object Pascal 中，您也可以使用匿名方法来定义事件处理程序。之所以不那么常见，可能是因为该功能是最近在该语言中引入的，并且那时已经存在许多库。而且，它增加了一点额外的复杂性。您可以在第 15 章中找到该方法的示例。另一个可能的扩展是为单个事件定义多个处理程序，例如 C＃支持，这不是标准功能，而是可以自己实现的功能。

10.3.2 Delegation（委托）的概念

乍一看，该技术的目标可能并不明确，但这是 Object Pascal 组件技术的基础之一。秘密在于授权一词。如果有人构建了一个带有一些方法指针的对象，则只需将新方法分配给指针即可随意更改该对象的行为。这听起来很熟悉吗？这应该。

当您为按钮添加 OnClick 事件处理程序时，开发环境正是这样做的。该按钮具有一个名为 OnClick 的方法指针，您可以直接或间接为其分配表单方法。当用户单击按钮时，即使您已在另一个类（通常是在表单中）中定义了此方法，该方法也会执行。

下面的清单概述了 Delphi 库中实际使用的代码，这些代码用于定义按钮组件的事件处理程序以及表单的相关方法：

```
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;
```

现在在过程中，您可以编写

```
MyButton.OnClick := Form1.Button1Click;
```

该代码段与库代码之间的唯一真正区别是 OnClick 是属性名称，它所引用的实际数据称为 FOnClick。

实际上，显示在“对象检查器”的“事件”页面中的事件只不过是作为方法指针的属性。例如，这意味着您可以在设计时动态修改附加到组件的事件处理程序，甚至可以在运行时构建新的组件并为其分配事件处理程序。

DynamicEvents 应用程序项目展示了这两种方案。表单具有一个按钮，该按钮具有与之关联的标准 OnClick 事件处理程序。但是，我向表单添加了第二个公共方法，该方法具有相同的签名（相同的参数）：

```
public
```

```
procedure BtnTest2Click(Sender: TObject);
```

当按下按钮时，除了显示一条消息外，还将事件处理程序切换到第二个，从而更改了单击操作的未来行为：

```
procedure TForm1.BtnTestClick(Sender: TObject);
begin
  ShowMessage ('Test message');
  BtnTest.OnClick := BtnTest2Click;
end;

procedure TForm1.BtnTest2Click(Sender: TObject);
begin
  ShowMessage ('Test message, again');
end;
```

现在，第一次按下按钮时，将执行第一个（默认）事件处理程序，而其他任何时候，您将使第二个事件处理程序运行。

❖ 在键入代码以将方法分配给事件时，代码完成将为您建议可用的事件名称，并将其转换为带有括号的实际函数调用。这是不正确的。您必须将方法本身分配给事件，而无需调用它。否则，编译器将尝试分配方法调用的结果（这是一个过程，不存在），从而导致错误。

该项目的第二部分演示了一个完全动态的事件关联。当您单击窗体的表面时，将使用事件处理程序动态创建一个新按钮，该事件处理程序显示关联按钮（Sender 对象）的标题：

```
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;
```

借助此代码，由于使用了事件的 Sender 参数，即使使用单个事件处理程序，每个动态创建的按钮也将通过显示依赖于该按钮的消息来响应鼠标单击。 输出示例在图 10.1 中可见。

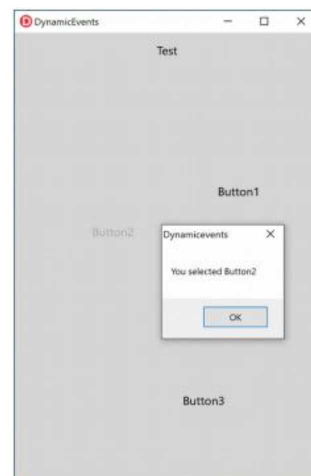

图 10.1：DynamicEvents 应用程序项目中动态创建的按钮显示的消息

10.3.3 事件就是属性

一个非常重要的概念是 Object Pascal 中的事件几乎总是作为方法指针类型的属性实现。这意味着要处理组件的事件，可以将方法分配给相应的 event 属性。就代码而言，这意味着您可以使用上一节中已经看到的以下代码为事件处理程序分配对象的方法：

```
Button1.OnClick := ButtonClickHandler;
```

同样，规则是事件的方法指针类型必须与您分配的方法的签名匹配，否则编译器将发出错误。系统为事件定义了几种常用的标准方法指针类型，从简单开始：

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;
```

通常，这是 OnClick 事件处理程序的类型，因此这意味着必须在类中将上述方法声明为：

```
procedure ButtonClickHandler (Sender: TObject);
```

如果这听起来有点令人困惑，请考虑在开发环境中会发生什么。选择一个按钮，假设是 Button1，双击它并在开发环境的 Object Inspector 中列出的 OnClick 事件上双击，然后将一个新的空方法添加到容器模块（可能是表单）中：

```
procedure TForm1.Button1Click (Sender: TObject)
Begin

end;
```

您填写了方法的代码，瞧，一切正常。这是因为将事件处理程序方法分配给事件的方式与在设计时设置的所有其他属性完全应用于组件的方式完全相同，都是在幕后进行的。

通过上面的描述，您可以了解事件与分配给它的方法之间没有一对一的对应关系。恰恰相反。您可以具有共享同一事件处理程序的多个事件，这些事件说明了频繁使用的 Sender 参数的原因，该参数指示哪个对象触发了该事件。例如，如果两个按钮具有相同的 OnClick 事件处理程序，则 Sender 值将包含一个被单击的按钮对象的引用。

您可以将相同的方法分配给代码中的不同事件，如上所述，但是也可以在设计时。在对象检查器中选择事件时，可以按事件名称右侧的箭头按钮，以查看“兼容”方法的下拉列表，该列表是具有相同方法指针类型的方法的列表。这样就很容易为不同组件的同一事件选择相同的方法。在某些情况下，您还可以将同一处理程序分配给同一组件的不同兼容事件。

10.3.4 将事件添加到 TDate 类

在 TDate 类中添加了一些属性后，我们现在可以添加一个事件。该事件将非常简单。它将被称为 OnChange，它可用于向组件用户警告日期值已更改。要定义一个事件，我们只需定义一个与之对应的属性，然后添加一些数据以存储该事件所引用的实际方法指针。这些是添加到 DateEvent 应用程序项目中的类的新定义：

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
```

```
    ...
  public
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
    ...
  end;
```

属性定义实际上非常简单。使用此类的开发人员可以为属性分配一个新值，从而为 FOnChange 私有字段分配一个新值。通常在程序启动时未分配该字段：事件处理程序是针对组件用户的，而不是针对组件编写器的。需要某种行为的组件编写者会将其添加到组件方法中。

换句话说，TDate 类仅接受事件处理程序，并在日期值更改时调用存储在 FOnChange 字段中的方法。 当然，仅在已分配事件属性的情况下才进行调用。

DoChange 方法（与事件触发方法一样，被声明为动态方法）进行测试和方法调用：

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

❖ 您可能在第 8 章中还记得，动态方法类似于虚拟方法，但是使用了不同的实现，该实现将内存占用减少到调用稍慢的代价。

每当其中一个值更改时，就会依次调用 DoChange 方法，如以下代码所示：

```
procedure TDate.SetValue (Y, M, D: Integer);
begin
  FDate := EncodeDate (Y, M, D);
  // 触发事件
  DoChange;
end;
```

现在，如果我们看一下使用此类的程序，就可以合理地简化其代码。首先，我们向表单类添加一个新的自定义方法：

```
type
  TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

此方法的代码仅使用 TDate 对象的 Text 属性的当前值更新标签：

```
procedure TDateForm.DateChange;
begin
  LabelDate.Text := TheDay.Text;
end;
```

然后，此事件处理程序将安装在 FormCreate 方法中：

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (7, 4, 1995);
  LabelDate.Text := TheDay.Text;
  // 分配事件处理程序以用于将来的更改
  TheDay.OnChange := DateChange;
end;
```

好吧，这似乎是很多工作。当我告诉您事件处理程序将为我们节省一些编码时，我是否在说谎？否。现在，添加一些代码后，当我们更改对象的某些数据时，我们完全可以忘记更新标签。作为示例，这里是按钮之一的 OnClick 事件的处理程序：

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
```

```
    TheDay.Increase;
end;
```

许多其他事件处理程序中也使用相同的简化代码。安装事件处理程序后，我们就不必记住要不断更新标签。这消除了程序中潜在的重大错误源。还要注意，我们必须在开始时编写一些代码，因为这不是开发环境中安装的组件，而只是一个类。使用组件，您只需在对象检查器中选择事件处理程序，然后编写一行代码即可更新标签。就这样。

这就引出了一个问题，用 Delphi 编写新组件有多困难？实际上很简单，下一节我将向您展示如何做。

❖ 这只是对属性和事件的作用以及编写组件的简短介绍。对这些功能的基本了解对于每个 Delphi 开发人员都很重要。本书没有深入研究编写自定义组件的细节。

10.4 创建一个 TDate 组件

现在我们了解了属性和事件，下一步就是查看什么是组件。通过将 TDate 类变成一个组件，我们将简要地探讨这个主题。

首先，我们必须从 TComponent 类继承我们的类，而不是默认的 TObject 类。这是代码：

```
type
  TDate = class (TComponent)
    ...
  public
    constructor Create (AOwner: TComponent); overload; override;
    constructor Create (Y, M, D: Integer); reintroduce; overload;
```

如您所见，第二步是向类添加新的构造函数，覆盖组件的默认构造函数以提供合适的初始值。因为有一个重载的版本，所以我们还需要对其使用 reintroduce 指令，以避免编译器发出警告消息。在调用基类构造函数之后，新构造函数的代码只需将日期设置为今天的日期即可：

```
constructor TDate.Create (AOwner: TComponent);
var
  Y, D, M: Word;
begin
  inherited Create (AOwner);
  // today...
  FDate := Date;
```

完成此操作后，我们需要向定义类的单元（DateComp 应用程序项目的 Dates 单元）中添加一个 Register 过程。（确保此标识符以大写字母 R 开头，否则将无法识别。）这是将组件添加到 IDE 所必需的。只需在单元的接口部分声明不需要参数的过程，然后在实现部分中编写以下代码：

```
procedure Register;
begin
  RegisterComponents ('Sample', [TDate]);
end;
```

此代码将新组件添加到“工具选项板”的“Sample”页面，并在必要时创建该页面。

最后一步是安装组件。 为此，我们需要创建一个包，这是一种特殊类型的应用程序项目托管组件。您要做的就是：

- 选择 File | New | Other 的 IDE 菜单，打开“新建项目”对话框。
- 选择“Package”。
- 用名称保存包（可能与带有实际组件代码的设备一起在同一文件夹中）。
- 在新创建的包项目中，在“项目管理器”窗格中，右键单击“ Contains node 包含”节点，以将新单元添加到项目中，然后选择具有 TDate 组件类的单元。
- 现在，在项目管理器中，您可以右键单击包节点，不仅可以发出 Build 命令，还可以选择 Install 菜单项，以在开发环境中安装组件。
- 如果从本书随附的代码开始，则只需完成上述序列的最后一步：从 DateComponent 文件夹中打开 DatePackage 项目并进行编译和安装。

如果现在创建一个新项目并移至“工具选项板”，则应在“Sample”下看到新组件。只需开始输入其名称以进行搜索。这将使用组件的默认图标显示。此时，您可以将组件放置在表单上，并开始在 Object Inspector 中操作其属性，如图 10.2 所示。您也可以比上一个示例更轻松地处理 OnChange 事件。

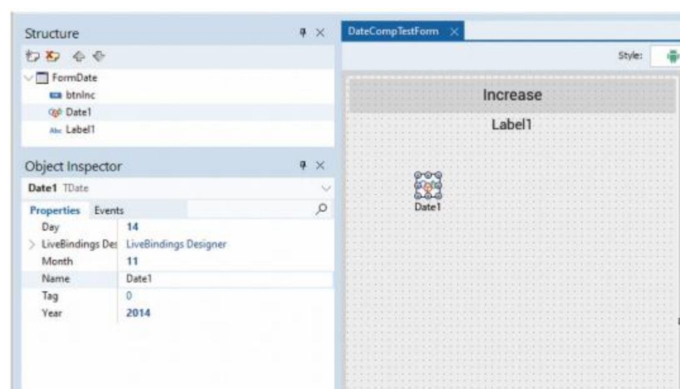

图 10.2：对象检查器中新的 TDate 组件的属性

除了尝试使用此组件构建您自己的示例应用程序（我确实建议您这样做）之外，您现在还可以打开 DateComponent 应用程序项目，该项目是我们在上逐步构建的组件的更新版本。本章的最后几节。这基本上是 DateEvent 应用程序项目的简化版本，因为现在事件处理程序可以直接在 Object Inspector 中使用。

❖ <u>如果您在编译和安装特定组件包之前打开 DateCompTest 应用程序项目（DatePackage 应用程序项目），则 IDE 在打开表单时将无法识别该组件，并会显示错误消息。在安装新组件之前，您将无法编译该程序或使其正常运行。</u>

10.5 在类中实现枚举支持

在第 3 章中，我们了解了如何使用 for-in 循环代替经典的 for 循环。在那部分中，我描述了如何对数组，字符串，集合和其他一些系统数据类型使用 for-in 循环。只要定义了枚举支持，就可以将这种循环应用于任何类。尽管最明显的示例是包含元素列表的类，但从技术角度来看，此功能是开放式的。

要支持枚举 Object Pascal 中类的元素，必须遵循两个规则：添加一个方法 GetEnumerator 来返回一个类（实际的枚举类）；使用 Next 方法和 Current 属性定义此枚举类，第一个用于在元素之间导航，第二个用于返回实际元素。完成此操作后（我将在稍后的实际示例中向您展示如何），编译器可以解决 for-in 循环，其中目标是我们的类，并且各个元素必须是 Current 的相同类型。枚举器的属性。

尽管不是绝对必要，但将枚举器支持类实现为嵌套类型（第 7 章介绍的语言功能）似乎是个好主意，因为单独使用用于枚举的特定类型确实没有任何意义。

下列类是 NumbersEnumerator 应用程序项目的一部分，它存储一定范围的数字（一种抽象集合），并允许对其进行迭代。通过定义一个枚举器使之成为可能，该枚举器声明为嵌套类型并由 GetEnumerator 函数返回：

```
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 方法创建一个嵌套类型的对象，该对象存储用于迭代数据的状态信息。

请注意，枚举器构造函数如何保留对要枚举的实际对象的引用（使用 self 作为参数传递的对象），并将初始位置设置为最开始：

```
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;
```

❖ 为什么构造函数将初始值设置为第一个值减去 1，而不是按预期的那样将第一个值设置为？事实证明，编译器为 for in 循环生成的代码与创建枚举并执行代码相对应，而 Next 确实使用 Current。由于列表可能没有值，因此在获取第一个值之前执行测试。这意味着在使用第一个元素之前调用 Next。我没有使用更复杂的逻辑来实现它，而是仅将初始值设置为第一个值之前的一个，以便在第一次将 Next 称为枚举器时将其放在第一个值上。

最后，枚举器方法使您可以访问数据并提供一种移动到列表中下一个值（或范围内的下一个元素）的方法：

```
function TNumbersRange.TNumbersRangeEnum.GetCurrent: Integer;
begin
```

```
    Result := NPos;
  end;

  function TNumbersRange.TNumbersRangeEnum.MoveNext: Boolean;
  begin
    Inc (NPos);
    Result := NPos <= FRange.NEnd;
  end;
```

如您在上面的代码中所看到的，Next 方法有两个不同的用途，移至列表的以下元素，并检查枚举数是否已到达末尾，在这种情况下，该方法返回 false。

完成所有这些工作之后，您现在可以使用 for..in 循环遍历 range 对象的值：

```
  var
    ARange: TNumbersRange;
    I: Integer;
  begin
    ARange := TNumbersRange.Create;
    ARange.nStart := 10;
    ARange.nEnd := 23;
    for I in ARange do
      Show (IntToStr (I));
```

输出仅是枚举的 10 到 23 之间的值列表：

```
  10
  11
  12
  13
  14
  15
  16
  17
  18
  19
  20
  21
  22
  23
```

❖ 在许多情况下，RTL 和 VCL 库都定义枚举器，例如，每个 TComponent 都可以枚举该组件是拥有的。缺少儿童控制枚举。在第 12 章中的“使用类帮助器添加枚举”部分中，我们将看到如何创建一个。该示例不在此处的原因是，我们首先需要讨论类帮助器。

10.6 关于混合 RAD 和 OOP 的 15 个技巧

在本章中，我介绍了属性，事件和发布的关键字，它们构成了与快速应用程序开发（RAD）或可视化开发或事件驱动的编程（三个术语引用相同的概念模型）相关的核心语言功能。尽管这是一个非常强大的模型，但它具有可靠的 OOP 架构。有时，RAD 方法可能会促使开发人员忘记良好的 OOP 做法。同时，回到编写纯代码，忽略 RAD 方法通常可能适得其反。在本章的最后部分中，我列出了一些关于桥接这两种方法的技巧和建议。

描述它的另一种方法是将其视为“RAD 之外的 OOP”一节。

❖ 本章最后部分的资料最初发表在“ The Delphi Magazine”（1999 年 7 月）第 17 期中，标题为“ Delphi 中 OOP 的 20 条规则”。现在，我修剪了一些规则，并

改写了其他规则，但本质仍然存在。

技巧 1：Form（表单）就是类

程序员通常将 Form（表单）视为对象，而实际上它们是类。区别在于您可以基于同一表单类拥有多个表单对象。

令人困惑的是，IDE 创建一个默认的全局变量，并且（取决于您的设置）还可以在启动时为您在项目中定义的每个表单类创建一个表单对象。对于新手来说，这当然是方便的，但是对于任何不平凡的应用程序来说，这通常是个坏习惯。

当然，为每个表单（及其类）和每个单元取一个有意义的名称非常重要。不幸的是，这两个名称必须不同，但是您可以使用约定以一致的方式映射这两个名称（例如 AboutForm 和 About.pas）。

在执行以下步骤时，您将看到此"Form 就是类"概念的实际效果。

技巧 2：Component（组件）名称

同样，对组件使用描述性名称也很重要。最常见的表示法是对类类型使用一些小写的首字母，然后使用组件的角色，如在 BtnAdd 或 EditName 中一样。实际上，遵循这种风格有许多类似的符号，而且实际上没有理由说任何一个都是最好的，这取决于您的个人喜好。

技巧 3：Envent（事件）名称

给事件处理方法起适当的名字同样重要。如果正确命名组件，则例如 Button1Click 的默认名称将成为 BtnAddClick。尽管我们可以从按钮名称中猜出该方法的作用，但我认为使用一个描述该方法效果的名称更好，而不是在触发该方法时。例如，如果此方法执行此操作，则 BtnAdd 按钮的 OnClick 事件可以命名为 AddToList。

这使得代码更具可读性，特别是当您从类的另一个方法调用事件处理程序时，这有助于开发人员将同一方法附加到多个事件或不同的组件上，尽管我不得不说，在任何非平凡的程序中将单个事件附加到多个用户界面元素时，使用操作是首选。

❖ Actions 和 ActionList 组件是 VCL 和 Fire Monkey UI 库的一个非常好的体系结构功能，提供了用户操作（及其状态）和与其关联的用户界面控件的概念性分离。 激活控件，执行操作。但是，如果您在逻辑上禁用该操作，则 UI 元素也会被禁用。该主题超出了本书的范围，但是如果您使用这些框架中的任何一个，则值得调查。

技巧 4：使用表单方法

如果表单是类，则其代码将收集在方法中。除了事件处理程序（其扮演着特殊角色，但仍可以称为其他方法）外，将自定义方法添加到表单类通常很有用。您可以添加执行操作并访问表单状态的方法。向表单添加公共方法要比让其他表单直接对其组件进行操作要好得多。

技巧 5：添加表单构造函数

在运行时创建的辅助窗体可以提供默认构造函数（继承自 TComponent 类）之外的其他特定构造函数。我的建议是重载 Create 方法，添加所需的初始化参数，如以下代码片段所示：

```
public
  constructor Create (Text: string); reintroduce; overload;

constructor TFormDialog.Create(Text: string);
begin
  inherited Create (Application);
  Edit1.Text := Text;
end;
```

技巧 6：避免全局变量

应避免使用全局变量（即，在单元的接口部分中声明的变量）。这里有一些建议可以帮助您做到这一点。如果您需要表单的额外数据存储，请向其中添加一些私有字段。在这种情况下，每个表单实例将拥有自己的数据副本。

您可以使用单元变量（在单元的实现部分中声明）来处理在表单类的多个实例之间共享的数据，但最好使用类数据（在第 12 章中进行了说明）。

技巧 7：永远不要在 TForm1 方法中使用 Form1

您永远不要在该对象的类的方法中引用该对象。换句话说，永远不要在 TForm1 类的方法中引用 Form1。如果需要引用当前对象，请使用 self 关键字。请记住，在大多数情况下都不需要这样做，因为您可以直接引用当前对象的方法和数据。如果您不遵守此规则，则在创建表单的多个实例时会遇到麻烦。

技巧 8：很少在其他表单中使用 Form1

即使在其他表单的代码中，也应尝试避免直接引用诸如 Form1 之类的全局对象。最好声明局部变量或私有字段以引用其他表单。例如，程序的主要 Form 可以具有引用对话框的私有字段。显然，如果您计划创建辅助表单的多个实例，则此规则变得至关重要。您可以将列表保留在主窗体的动态数组中，或者仅使用全局 Screen 对象的 Forms 数组来引用应用程序中当前存在的任何窗体。

技巧 9：删除全局 Form1 变量

实际上，我的建议是删除在您向其中添加新表单（如 Form1）时 IDE 自动将其自动添加到项目中的全局表单对象。仅当您禁用该表格的自动创建时，这才是可能的，我建议您还是应该删除该表单。或者，您可以从项目源代码中删除用于创建表单的对应行。

不用说，如果表单不是自动创建的，您将需要在应用程序中创建一些代码来创建它，并可能需要其他一些变量来引用它。

我认为删除全局表单对象对 Object Pascal 的新手非常有用，因为他们以后再

也不会在类和全局对象之间感到困惑了。实际上，删除全局对象后，对其的任何引用都将导致错误。

技巧 10：添加表单属性

正如我在本章"向表单添加属性"一节中已经讨论的那样，当您需要表单数据时，请添加一个私有字段。如果需要从其他类访问此数据，则将属性添加到表单。使用这种方法，您将能够更改表单及其数据（包括其用户界面）的代码，而不必更改其他表单或类的代码。您还应该使用属性或方法来初始化辅助表单或对话框，并读取其最终状态。正如我已经描述的，初始化也可以使用构造函数来执行。

技巧 11：公开组件属性

当您需要访问另一个表单的状态时，您不应直接引用其组件。这会将其他表单或类的代码绑定到用户界面，这是应用程序中发生最多更改的部分之一。而是声明一个映射到组件属性的表单属性：这是通过读取组件状态的 Get 方法和写入组件状态的 Set 方法来完成的。

假设您现在更改用户界面，将组件替换为另一个组件。您所需要做的就是修复与该属性相关的 Get 和 Set 方法，而不必检查和修改可能引用该组件的所有表单和类的源代码。

技巧 12：在需要时使用数组属性

如果需要处理表单中的一系列值，则可以声明一个数组属性。

如果这是表单的重要信息，则可以将其也设置为表单的默认数组属性，以便可以通过编写 SpecialForm [3]直接访问其值。在"使用索引属性"一节中，已经针对比表单更通用的情况介绍了这一点。

技巧 13：在属性中开始操作

请记住，使用属性而不是访问全局数据的优点之一是，可以在写入（或读取）属性值时调用方法并执行任何操作。例如，您可以直接在窗体表面上绘制，设置多个属性的值，调用特殊方法，立即更改多个组件的状态或触发事件（如果有）。

另一个相关示例是使用属性获取器来实现延迟创建。

无需在类构造函数中创建子对象，而是可以在第一次需要它时创建它，编写如下代码：

```
private
   FBitmap: TBitmap;
public
   property Bitmap: TBitmap read GetBitmap;

function TBitmap.GetBitmap: TBitmap;
begin
   if not Assigned (FBitmap) then
      FBitmap := ... // 创建并初始化
   Result := FBitmap;
end;
```

技巧 14：隐藏组件

我经常听到 OOP 纯粹主义者抱怨，因为表单在已发布的部分中包含了组件列表，这种方法不符合封装原理。他们实际上指出了一个重要的问题，但是他们中的大多数人似乎都不知道解决方案已经准备就绪，无需重写库或更改语言。可以将添加到表单的组件引用移到表单声明的私有部分，以便其他表单无法访问它们。这样，您可以强制使用映射到组件的属性（请参见上文）访问它们的状态。

如果 IDE 将所有组件都放置在已发布部分中，则是由于这些字段绑定到从流文件（DFM 或 FMX）创建的组件的方式。设置组件名称后，VCL 会自动将组件对象附加到表单中的引用。这仅在引用发布时才有可能，因为流传输系统使用传统的 RTTI 和某些 TObject 方法来执行操作。

因此，发生的事情是，通过将组件引用从已发布的部分移到私有部分，您失去了这种自动行为。要解决该问题，只需将其手动添加即可，只需在表单的 OnCreate 事件处理程序中为每个组件添加以下代码即可：

```
Edit1 := FindComponent('Edit1') as TEdit;
```

您要做的第二个操作是在系统中注册组件类，以使它们的 RTTI 信息包含在已编译的程序中并可供系统使用。每个组件类仅需要一次，并且仅当将此类型的所有组件引用都移到专用部分时才需要。即使不确定您的应用程序是否需要它，也可以添加此调用，因为对同一类的 RegisterClasses 方法的额外调用无效。通常将 RegisterClasses 调用添加到托管该表单的单元的初始化部分：

```
RegisterClasses([TEdit]);
```

技巧 15：使用 OOP 表单向导

对每个表单的每个组件重复上述两个操作无疑是无聊且耗时的。为了避免过多的负担，我编写了一个简单的向导，该向导会在一个小窗口中生成要添加到程序中的代码行。

您将需要对每个表单进行简单的复制和粘贴操作，因为向导不会自动将源代码放入单元的初始化部分。

如何获得向导？您可以在以下网址找到它作为“ Cantools Wizards”的一部分：https://github.com/marcocantu/cantools

技巧结论

这只是有关 RAD 和 OOP 开发模型更加平衡的一些提示和建议。当然，该主题还有更多内容，远远超出了本书的重点，而本书的重点主要是语言本身，而不是应用程序体系结构的最佳实践。

有几本书涵盖了使用 Delphi 的应用程序体系结构，但是没有什么比尼克·霍奇斯（Nick Hodges）撰写的系列书籍更好，包括“ Coding in Delphi”，“More Coding in Delphi”和“Dependency Injection in Delphi”。

您可以在 http://www.codingindelphi.com/ 上找到有关这些书的更多信息。

第十一章 Interfaces（接口）

与 C++和其他几种语言相反，Object Pascal 继承模型不支持多重继承。这意味着每个类只能有一个基类。

多重继承的有用性是 OOP 专家争论的话题。在 Object Pascal 中缺少此构造可以被认为是一个缺点，因为您没有 C++的功能，但也可以是一个优势，因为您拥有一种更简单的语言并避免了多重继承带来的一些问题。弥补 Object Pascal 中缺少多个继承的一种方法是使用接口，该接口使您可以定义同时实现多个抽象的类。

❖ 当今大多数面向对象的编程语言都不支持多重继承，而是使用接口，包括 Java 和 C＃。接口提供了声明支持在类上实现的多个接口的灵活性和功能，同时避免了继承多个实现的问题。多继承支持仍然主要限于 C ++语言。一些动态的面向对象语言支持混入，这是一种与多继承类似的不同且更简单的方法。

在这里，而不是进行辩论，我只是假设从多个“角度”对待单个对象很有用。但是在构建示例解释该原理之前，我们必须介绍接口在 Object Pascal 中的作用，并弄清它们如何工作。

从更一般的角度来看，接口与类支持的面向对象的编程模型略有不同。实现接口的对象对其所支持的每个接口都具有多态性。实际上，基于接口的模型功能强大。接口有利于封装，并且在类之间提供比继承更松散的连接。

❖ 本章介绍的技术以及对接口的整体支持最初是作为支持和实现 Windows COM（组件对象模型）体系结构的方式添加到 Object Pascal 语言的。后来，该功能扩展为在该场景之外可以完全使用，但是一些 COM 元素（例如通过 ID 的接口标识和引用计数支持）仍然保留在当前的 Object Pascal 接口实现中，这使其与大多数其他语言略有不同。

11.1 使用 Interfaces（接口）

除了声明抽象类（带有抽象方法的类）之外，在 Object Pascal 中，您还可以编写一个纯粹的抽象类。也就是说，只有虚拟抽象方法的类。这是通过使用特定的关键字 interface 来完成的。因此，我们将这些数据类型称为接口。

从技术上讲，接口不是类，尽管它可能类似于类。虽然一个类可以有一个实例，但接口不能。一个接口可以由一个或多个类实现，以便这些类的实例最终支持或实现该接口。

在 Object Pascal 中，接口具有一些独特的功能：

- 与类类型变量不同，接口类型变量是引用计数的，提供了一种自动内存管理的形式。
- 一个类只能从一个基类继承，它可以实现多个接口。
- 就像所有类都继承自 TObject 一样，所有接口也都继承自 IInterface，形成一个单独的正交层次结构。
- 按照惯例，接口名称以字母 I 开头，而不是大多数其他数据类型使用的字母 T。

❖ 最初，Object Pascal 中的基本接口类型称为 IUnknown，因为这是 COM 所需要的。IUnknown 接口后来被重命名为 IInterface，以强调您甚至可以在 Object Pascal 中甚至在 COM 领域之外以及在不存在 COM 的操作系统上使用该接口

的事实。无论如何，IInterface 的实际行为仍然与 IUnknown 的先前行为相同。

11.1.1 声明接口

看完核心概念之后，让我们转到一些实际的代码，这些代码应该有助于您了解接口在 Object Pascal 中的工作方式。实际上，接口的定义类似于类定义。该定义包含方法列表，但是这些方法没有以任何方式实现，这与常规类中的抽象方法完全一样。

以下是接口的定义：

```
type
  ICanFly = interface
    function Fly: string;
  end;
```

给定每个接口直接或间接继承自基本接口类型，这相当于编写：

```
type
  ICanFly = interface (IInterface)
    function Fly: string;
  end;
```

再过一会儿，我将向您展示从 IInterface 继承的含义以及它带来的好处。目前，足以说明 IInterface 有一些基本方法（再次类似于 TObject）。

最后一个与接口声明有关的花絮。对于接口，部分类型检查是动态完成的，并且系统要求每个接口都具有唯一的标识符或 GUID，您可以通过按 Ctrl + Shift + G 在 Delphi 编辑器中获得该标识符。这是接口的完整代码：

```
type
  ICanFly = interface
    ['{D7233EF2-B2DA-444A-9B49-09657417ADB7}']
    function Fly: string;
  end;
```

Intf101 应用程序项目中提供了此接口及其实现（如下所述）。

❖ 尽管您可以在不指定 GUID 的情况下编译和使用接口，但通常仍需要生成 GUID，因为使用该接口类型进行接口查询或动态类型转换时需要使用该 GUID。 接口的全部要点是在运行时灵活地利用大大扩展的类型，这取决于接口是否具有 GUID。

11.1.2 Implementing（实施）接口

任何类都可以通过在其继承的基类之后列出它们，并为每个接口的每个方法提供一个实现来实现一个或多个接口：

```
type
  TAirplane = class (..., ICanFly)
    function Fly: string;
  end;

function TAirplane.Fly: string;
begin
  // 实际代码
end;
```

当类实现接口时，它必须提供具有相同签名的所有接口方法的实现，因此，在这种情况下，TAirplane 类必须将 Fly 方法实现为返回字符串的函数。如果接口

也继承自基本接口（IInterface），则实现接口的类必须始终提供接口及其所有基本接口的所有方法。

这就是为什么在继承自已经实现 IInterface 基本接口方法的基类的类中实现接口的常见原因。

Object Pascal 运行时库已经提供了一些基本类来实现基本行为。最简单的一个是 TInterfacedObject 类，因此上面的代码可能变为：

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

❖ 实现接口时，可以使用虚拟方法的静态方法。如果您计划使用继承类中的方法，则使用虚拟方法很有意义。但是，还有另一种方法，即指定基类也从同一接口继承，并重写该方法。我倾向于倾向于在需要时声明将接口方法实现为虚拟方法的方法。

现在我们已经定义了一个接口和一个实现它的类，我们可以创建该类的对象。我们可以这样写：

```
var
  Airplane1: TAirplane;
begin
  Airplane1 := TAirplane.Create;
  try
    Airplane1.Fly;
  finally
    Airplane1.Free;
  end;
end;
```

在这种情况下，我们将忽略该类实现接口的事实。不同之处在于，现在我们还可以声明接口类型的变量。使用接口类型变量会自动启用参考内存模型，因此我们可以跳过 try-finally 块：

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

关于这个看似简单的代码片段的第一行，有一些相关的注意事项，这也是 Intf101 应用程序项目的一部分。首先，将对象分配给接口变量后，运行时会使用特殊版本的 as 运算符自动检查该对象是否实现了该接口。您可以通过编写以下相同的代码行来明确表示此操作：

```
  Flyer1 := TAirplane.Create as ICanFly;
```

其次，无论我们使用直接赋值还是 as 语句，运行时都会做一件事：它调用对象的_AddRef 方法，从而增加了其引用计数。这是通过调用对象从基类 TInterfacedObject 继承的方法来完成的。

同时，一旦 Flyer1 变量超出范围（即在执行 end 语句时），Delphi 运行时就会调用_Release 方法，该方法将减少引用计数，检查引用计数是否为零，并且在这种情况下会销毁对象。这就是为什么在上面的代码中，无需手动释放我们在代码中创建的对象，也无需编写 try-finally 块。

❖ 尽管上面的源代码没有 try-finally 块，但是编译器将自动向方法添加一个隐式 try-finally 块，并带有对_Release 的隐式调用。这在 Object Pascal 中的许多

情况下都会发生：基本上，每一种方法具有一个或多个托管类型（如字符串，接口或动态数组）时，编译器都会添加一个自动的和隐式的 try-finally 块。

11.1.3 接口和引用计数

正如我们在上面的代码中看到的那样，接口变量引用的 Object Pascal 对象是引用计数的（除非接口类型变量被标记为弱或不安全，如后面所述）。我们还看到，当不再有接口变量引用它们时，它们可以自动释放。

需要承认的重要一点是，虽然涉及到一些编译器魔术（隐藏的_AddRef 和 _Release 调用），但是实际的引用计数机制还是要由开发人员或运行时库提供的特定实现来实现。在最后一个示例中，由于 TInterfacedObject 类的方法中的代码（此处以略微简化的版本列出），引用计数的确在起作用：

```
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;
```

现在考虑实现 IInterface 的另一个基类，该基类也可以在 RTL（位于 Generics.Defaults 单元中）TSingletonImplementation（单例模式）中找到。

这个奇怪命名的类基本上禁用了实际的引用计数机制：

```
function TSingletonImplementation._AddRef: Integer;
begin
  Result := -1;
end;

function TSingletonImplementation._Release: Integer;
begin
  Result := -1;
end;
```

❖ TSingletonImplementation 类确实是错误的称呼，因为它与单例模式无关。我们将在下一章中看到这种模式的示例。

尽管不常用 TSingletonImplementation，但还有另一个实现接口并禁用引用计数机制的类，仅因为它具有自己的内存管理模型，而这就是 TComponent 类。

如果您想拥有一个实现接口的自定义组件，则不必担心引用计数和内存管理。在本章末尾的“用接口实现模式”部分中，我们将看到一个自定义组件实现接口的示例。

11.1.4 混合参考中的错误

使用对象时，通常应该仅使用对象变量或仅使用接口变量来访问它们。混合使用这两种方法会破坏 Object Pascal 提供的参考计数方案，并且可能导致很难跟

踪的内存错误。实际上，如果您决定使用接口，则可能应该仅使用基于接口的变量。

这是许多可能的类似情况中的一个示例。假设您有一个接口，一个实现它的类以及一个以接口为参数的全局过程：

```
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;
```

该代码看起来相当琐碎，并且 100％正确。调用过程的方式可能是错误的（此代码是 IntfError 应用程序项目的一部分）：

```
procedure TForm1.BtnMixClick(Sender: TObject);
var
  AnObj: TMyIntfObject;
begin
  AnObj := TMyIntfObject.Create;
  try
    ShowThat (AnObj);
  finally
    AnObj.Free;
  end;
end;
```

这段代码中发生的事情是我正在将一个普通对象传递给需要接口的函数。给定对象确实支持该接口，则编译器进行调用就没有问题。问题在于内存的管理方式。

最初，该对象的引用计数为零，因为没有接口引用该对象。进入 ShowThat 过程后，引用计数将增加到 1。确定，然后进行调用。现在，退出程序后，参考计数将减少并变为零，因此对象被销毁。换句话说，将 anObj 传递给过程时，它会被销毁，这确实很尴尬。如果运行此代码，它将失败并显示内存错误。

可能有多种解决方案。您可以人为地增加引用计数并使用其他低级技巧。但是真正的解决方案是不混合使用接口和对象引用，而仅使用接口来引用对象（此代码再次从 IntfError 应用程序项目中获取）：

```
procedure TForm1.BtnIntfOnlyClick(Sender: TObject);
var
  AnIntf: IMyInterface;
begin
  AnIntf := TMyIntfObject.Create;
  ShowThat (AnIntf);
end;
```

在这种特定情况下，解决方案就在眼前，但在许多其他情况下，很难提供正确的代码。再次，经验法则是避免混合使用不同类型的引用……但也请继续阅读下一部分，以了解一些最新的替代方法。

11.1.5 Weak（弱）和 Unsafe（不安全）的接口参考

从 Delphi 10.1 Berlin 开始，Object Pascal 语言在接口引用的管理方面提供了一些增强。实际上，该语言提供了不同类型的引用：

- 常规引用在分配和释放时增加和减少对象引用计数，最终在引用计数达到零时释放对象。
- 弱引用（用[weak]修饰符标记）不会增加所引用对象的引用计数。这些引用是完全托管的，因此如果引用的对象被销毁，它们将自动设置为 nil。
- 不安全的引用（用[unsafe]修饰符标记）不会增加它们所引用且不受管理的对象的引用计数-与基本指针没有太大区别。

❖ 最初，弱和不安全引用的使用是作为对移动平台的 ARC 内存管理支持的一部分而引入的。随着 ARC 的逐步淘汰，该功能仅可用于接口引用。

在引用计数处于活动状态的常见方案中，您可以具有类似以下的代码，这些代码依赖于引用计数来处置临时对象：

```
procedure TForm3.Button2Click(Sender: TObject);
var
  OneIntf: ISimpleInterface;
begin
  OneIntf := TObjectOne.Create;
  OneIntf.DoSomething;
end;
```

如果对象具有标准的引用计数实现，并且您要创建一个不包含引用总数的接口引用怎么办？现在，您可以通过在接口变量声明中添加[unsafe]属性，并将上面的代码更改为：

```
procedure TForm3.Button2Click(Sender: TObject);
var
  [unsafe] OneIntf: ISimpleInterface;
begin
  OneIntf := TObjectOne.Create;
  OneIntf.DoSomething;
end;
```

这不是一个好主意，因为上面的代码将导致内存泄漏。 通过禁用引用计数，当变量超出范围时不会发生任何事情。

在某些情况下这是有益的，因为您仍然可以使用接口而不触发额外的引用。换句话说，不安全的引用就像...一个指针一样对待，没有额外的编译器支持。

现在，在考虑使用 unsafe 属性在不增加计数的情况下使用引用时，请考虑在大多数情况下还有另一个更好的选择，即使用弱引用。弱引用也可以避免增加引用计数，但是可以对它们进行管理。这意味着系统会跟踪弱引用，并且在实际对象被删除的情况下，会将弱引用设置为 nil。

相反，使用不安全的引用，您将无法知道目标对象的状态（这种情况称为"悬挂引用"）。

弱引用在哪些情况下有用？一个经典的案例是两个具有交叉引用的对象。实际上，在这种情况下，对象会人为地增加其他对象的引用计数，并且即使它们变得不可访问，它们也基本上会永久保留在内存中（将引用计数设置为 1）。

作为示例，考虑以下接口，该接口接受对同一时间的另一个接口的引用，并使用内部引用实现该类的类：

```
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;
```

如果创建两个对象并交叉引用它们，则会导致内存泄漏：

```
  var
    One, Two: ISimpleInterface;
  begin
    One := TObjectOne.Create;
    Two := TObjectOne.Create;
    One.AddObjectRef (Two);
    Two.AddObjectRef (One);
```

现在，Delphi 中可用的解决方案是将专用字段 AnotherObj 标记为弱接口引用：

```
  private
    [weak] AnotherObj: ISimpleInterface;
```

进行此更改后，当您将对象作为参数传递给 AddObjectRef 调用时，引用计数不会被修改，它保持为 1，并且当变量超出范围时它将返回零，从而将对象从内存中释放出来。

现在，在许多其他情况下，此功能变得很方便，并且底层实现中确实存在一些复杂性。 这是一个很棒的功能，但是需要一些努力才能完全掌握。 此外，由于管理弱引用（不管理不安全引用），因此它确实有一些运行时成本。

有关接口的弱引用及其工作方式的其他信息，请参考第 13 章“对象和内存”中的“内存管理和接口：”部分。

11.2 先进的接口技术

为了进一步研究接口的功能，在研究现实世界的使用场景之前，重要的是涵盖它们的一些更高级的技术功能，例如如何实现多个接口或如何使用具有接口功能的方法来实现接口方法。不同的名称（如果名称冲突）。

接口具有的另一个重要功能是属性。为了演示与接口相关的所有这些更高级的功能，我编写了 IntfDemo 应用程序项目。

11.2.1 接口属性

本节中的代码基于两个不同的接口 IWalker 和 IJumper，它们均定义了一些方法和一个属性。接口属性只是映射到读取和写入方法的名称。与类不同，您不能仅仅因为接口不能包含字段而将接口属性映射到字段。

这些是实际的接口定义：

```
    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;
```

当使用属性实现接口时，您只需实现实际的访问方法，因为该属性是透明的，并且在类本身中不可用：

```
TRunner = class (TInterfacedObject, IWalker)
private
  FPos: Integer;
public
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

实现代码并不复杂（您可以在 IntfDemo 应用程序项目中找到它），其中的方法可以计算新位置并显示正在执行的操作：

```
function TRunner.Run: string;
begin
  Inc (FPos, 2);
  Result := FPos.ToString + ': Run';
end;
```

使用 IWalker 接口及其 TRunner 实现的演示代码如下：

```
var
  Intf: IWalker;
begin
  Intf := TRunner.Create;
  Intf.Position := 0;
  Show (Intf.Walk);
  Show (Intf.Run);
  Show (Intf.Run);
end;
```

输出应该不足为奇：

```
1: Walk
3: Run
5: Run
```

11.2.2 接口 Delegation（委托）

以类似的方式，我可以定义一个简单的类来实现 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;
```

此实现与先前的实现不同，因为它使用特定的基类 TAggregatedObject。这是一个特殊用途的类，用于定义内部用于支持接口的对象，稍后我将展示其语法。

- ❖ TAggregatedObject 类是在 System 单元中定义的 IInterface 的另一种实现。与 TInterfacedObject 相比，它在引用计数的实现（基本上将所有引用计数委派给容器或控制器）和接口查询的实现（如果容器支持多个接口）方面有所不同。

我将以其他方式使用它。在下面的类 TMyJumper 中，我不想使用类似的方法重复执行 IJumper 接口。

相反，我想将该接口的实现委派给已经实现该接口的类。这不能通过继承来完成（我们不能有两个基类）。相反，您可以使用语言的特定功能-接口委托。

以下类通过引用具有属性的实现对象来实现接口，而不是实现接口的实际方法：

```
    TMyJumper = class (TInterfacedObject, IJumper)
    private
      FJumpImpl: TJumperImpl;
    public
      constructor Create;
      destructor Destroy; override;
      property Jumper: TJumperImpl read FJumpImpl implements IJumper;
    end;
```

该声明表明 FJumpImpl 字段为 TMyJumper 类实现了 IJumper 接口。当然，该字段必须实际实现接口的所有方法。为了使这项工作有效，您需要在创建 TMyJumper 对象时为该字段创建一个适当的对象（基本 TAggregatedObject 类需要构造函数参数）：

```
  constructor TMyJumper.Create;
  begin
    FJumpImpl := TJumperImpl.Create (self);
  end;
```

该类还具有用于释放内部对象的析构函数，该析构函数由常规字段而不是接口引用（因为引用计数在这种情况下不起作用）。

这个示例很简单，但是总的来说，随着您开始修改某些方法或添加仍对内部 FJumpImpl 对象的数据进行操作的其他方法，事情变得更加复杂。这里的总体概念是，您可以在多个类中重用接口的实现。

使用此接口间接实现的代码与将编写的标准代码相同：

```
  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;
```

11.2.3 多重接口和方法别名

接口的另一个非常重要的功能是类可以实现多个功能。以下 TAthlete 类对此进行了演示，该类同时实现了 IWalker 和 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;
```

一个接口直接实现，而另一个接口则委托给内部 FJumpImpl 对象，这与我在前面的示例中所做的完全一样。

现在我们有一个问题。我们要实现的两个接口都具有带有相同参数签名的 Walk 方法，那么如何在类中实现这两个接口呢？在有多个接口的情况下，语言如何支持方法名称冲突？解决方案是给该方法一个不同的名称，并将其映射为特定的接口方法，并使用以下语句作为前缀：

```
function IWalker.Walk = Walk1;
```

该声明指出该类使用称为 Walk1 的方法（而不是使用具有相同名称的方法）实现 IWalker 接口的 Walk 方法。最后，在实现此类的所有方法时，我们需要引用 FJumpImpl 内部对象的 Position 属性。

通过声明 Position 属性的新实现，我们最终将为单个运动员获得两个位置，这是一个很奇怪的情况。以下是几个示例：

```
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;
```

我们如何创建此 TAthlete 对象的接口并同时引用 IWalker 和 IJumper 接口中的两个操作？ 好吧，我们不能完全做到这一点，因为没有可以使用的基本接口。但是，接口允许进行更动态的类型检查和类型转换，因此我们可以将接口转换为不同的接口，只要我们要引用的对象支持两个接口，编译器就可以做到这一点。仅在运行时查找。 这是该方案的代码：

```
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，我们可以选择两个接口中的一个，然后将其转换为另一个。使用 as 强制转换是一种运行时转换的方法，但是在处理接口时，还有更多选项，我们将在下一节中看到。

11.2.4 接口（Polymorphism）多态

在上一节中，我们看到了如何定义多个接口并让一个类实现其中的两个接口。当然，这可以扩展到任何数量。您还可以创建接口层次结构，因为接口可以从现有接口继承：

```
ITripleJumper = interface (IJumper)
  ['{0876F202-AAD3-11D2-8551-CCA30C584521}']
  function TripleJump: string;
end;
```

此新接口类型具有其基本接口类型的所有方法（和属性），并添加了一个新方法。当然，接口兼容性规则与类非常相似。

不过，我在本节中要重点介绍的主题略有不同，它是基于接口的多态性。 给定一个通用的基类对象，我们可以调用一个虚拟方法，并确保将调用正确的版本。接口也可能发生同样的情况。但是，有了接口，我们可以超越一步，并且经常拥有查询接口的动态代码。鉴于对象与接口的关系可能非常复杂（一个对象可以实现多个接口，也可以间接实现其基本接口），因此，更好地了解这种情况下的可能性非常重要。

首先，假设我们有一个通用的 IInterface 参考。我们如何知道它是否支持特定的接口？实际上有多种技术，与同类技术略有不同：

- 使用 is 语句进行测试（并可能在以下转换中使用 as 强制转换）。这可用于检查对象是否支持接口，但不能检查用接口引用的对象是否也支持另一个接口（即，您不能将其应用于接口）。请注意，无论如何都需要 as 转换：直接转换为接口类型几乎总是会导致错误。
- 使用许多重载版本之一调用全局支持功能。您可以将要测试的对象或接口，目标接口（使用 GUID 或类型名称）传递给此函数，并且如果函数成功，还可以将接口变量传递给将存储实际接口的接口变量。
- 直接调用 IInterface 基本接口的 QueryInterface 方法，该方法稍低一些，它总是需要接口类型变量作为额外结果，并使用数字 HRESULT 值而不是布尔结果。

这是来自同一 IntfDemo 应用程序项目的摘录，显示了如何将后两种技术应用于通用 IInterface 变量：

```
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;
```

与 QueryInterface 调用相比，我完全建议使用 Supports 函数的重载版本之一。最终，支持人员会称呼它，但提供了更简单，更高级别的选项。

您想在接口上使用多态性的另一种情况是，当您拥有一个具有更高级别接

口类型的接口数组（但也可能是对象数组，其中一些对象可能支持给定的接口）。

11.2.5 从接口引用中提取对象

很多版本的 Object Pascal 都是这种情况，当您将一个对象分配给一个接口变量时，就无法访问原始对象。有时，开发人员会在其接口中添加 GetObject 方法来执行该操作，但这是一个很奇怪的设计。

使用当今的语言，您可以将接口引用投射回原来已分配给它们的原始对象。您可以使用三种单独的操作：

- 您可以编写一个 **is** 测试以验证确实可以从接口引用中提取给定类型的对象：

  IntfVar is TMyObject

- 您可以编写 as 强制转换以执行类型强制转换，并在出现错误的情况下引发异常：

  IntfVar as TMyObject

- 您可以编写硬类型强制转换以执行相同的转换，并在出现错误的情况下返回 nil 指针：

  TMyObject(IntfVar)

❖ 在每种情况下，仅当接口最初是从 Object Pascal 对象而不是从 COM 服务器获得的时，类型转换操作才起作用。还要注意，您不仅可以转换为原始对象的确切类，还可以转换为其基类之一（遵循类型兼容性规则）。

例如，考虑具有以下简单的接口和实现类（ObjFromIntf 应用程序项目的一部分）：

```
type
  ITestIntf = interface (IInterface)
    ['{2A77A244-DC85-46BE-B98E-A9392EF2A7A7}']
    procedure DoSomething;
  end;

  TTestImpl = class (TInterfacedObject, ITestIntf)
  public
    procedure DoSomething;
    procedure DoSomethingElse; // 不在接口中
    destructor Destroy; override;
  end;
```

有了这些定义，您现在可以定义一个接口变量，为它分配一个对象，并使用它通过新的类型转换来调用不在接口中的方法：

```
var
  Intf: ITestIntf;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
  (Intf as TTestImpl).DoSomethingElse;
```

您还可以使用 is 测试和直接强制转换，通过以下方式编写代码，并且始终可以强制转换为对象实际类的基类：

```
var
  Intf: ITestIntf;
  Original: TObject;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
```

```
if Intf is TObject then
  Original := TObject (Intf);
(Original as TtestImpl).DoSomethingElse;
```

考虑到直接转换如果不成功则返回 nil，因此您还可以按以下方式编写代码（无需进行 is 测试）：

```
Original := TObject (intf);
if Assigned (Original) then
  (Original as TTestImpl).DoSomethingElse;
```

请注意，将从接口提取的对象分配给变量会使您面临引用计数问题：当接口设置为 nil 或超出范围时，该对象实际上将被删除，引用该对象的变量将变为无效。您将在示例的 BtnRefCountIssueClick 事件处理程序中找到突出显示该问题的代码。

11.3 用接口实现 Adapter Pattern（适配器模式）

作为使用接口的真实示例，我在本章中添加了涵盖适配器模式的部分。简而言之，适配器模式用于将一个类的接口转换为该类用户期望的另一种接口。这使您可以在需要定义接口的框架中使用现有类。

可以通过创建执行映射的新类层次结构或扩展现有类以使其暴露新接口的方式来实现该模式。这可以通过多重继承（以支持它的语言）或使用接口来完成。在这最后一种情况下，这就是我将在这里使用的，一个新的继承类将实现给定的接口并将其现有行为映射到其方法。

在特定情况下，适配器提供了一个通用接口来查询多个组件的值，这些接口碰巧具有不一致的接口（这在 UI 库中经常发生）。这是一个称为 ITextAndValue 的接口，因为它允许通过获取文本描述或数字描述来访问组件的状态：

```
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;
```

下一步是为我们希望能够与接口一起使用的每个组件创建一个新的子类。例如，我们可以这样写：

```
type
  TApadterLabel = class(TLabel, ITextAndValue)
  protected
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;
  end;
```

这四个方法的实现非常简单，因为如果值（或文本）是数字，则可以将它们映射到 Text 属性，以执行类型转换。但是，现在有了一个新组件，您将必须安装它（如上一章中所述），并用此新组件替换表单中的现有组件。对要调整的每个组件重复相同的过程将非常耗时。

一个更简单的替代方法是使用插入器类习惯用法（即，使用与基类相同的名

称定义一个类，但使用不同的单元）。编译器和运行时流式传输系统将正确识别这一点，因此在运行时您将获得新的特定类的对象。唯一的区别是，在设计时，您将看到基本组件类的实例并与之交互。

❖ 多年前，《德尔菲杂志》（The Delphi Magazine）首次提到了中介层类，并以此名称命名。他们当然是一个小技巧，但有时却很方便。我考虑插入器类，即与基类名称相同但在不同单元中定义的类，更多的是对象用法。请注意，要使该机制正常工作，至关重要的是，在应将其替换为具有常规类的单元之后，在 uses 语句中列出具有插入器类的单元。换句话说，在 uses 语句的最后一个单元中定义的符号替换了以前包含的单元中定义的相同符号。当然，您始终可以通过在符号前面加上单元名称来区分符号，但这确实会破坏这种黑客的全部想法，因为后者利用了全局名称解析规则。

要定义插入器类，通常需要编写一个新单元，其类与现有基类的名称相同。要引用基类，必须在其前面加上单元名称（否则编译器将其视为递归定义）：

```
type
  TLabel = class(StdCtrls.TLabel, ITextAndValue)
  protected
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;
  end;
```

在这种情况下，您不必安装组件或触摸现有程序，而只需在列表末尾添加额外的使用声明即可。在这两种情况下（但是我编写的演示应用程序都使用插入器类），您可以查询此适配器接口的表单组件，例如，编写代码将所有值设置为 50，这将影响不同的属性。不同的组件：

```
var
  Intf: ITextAndValue;
  I: integer;
begin
  for I := 0 to ComponentCount - 1 do
    if Supports (Components [I], ITextAndValue, Intf) then
      Intf.Value := 50;
end;
```

在特定的示例中，此代码将影响进度栏或数字框的值以及标签或编辑的文本。它还将完全忽略我没有为其定义适配器接口的其他几个组件。尽管这只是一个非常特殊的情况，但是如果您检查其他设计模式，您会很容易发现，利用 Object Pascal 类（例如 Java 和 C# 中的类）所具有的额外灵活性接口，可以更好地实现其中的许多设计模式。仅举几例广泛使用接口的流行语言）。

第十二章 Manipulating（操纵）类

在前几章中，您已经了解了 Object Pascal 语言的对象方面的基础：类，对象，方法，构造函数，继承，后期绑定，接口等等。现在，我们需要通过查看与类管理相关的语言的一些更高级且更具体的功能，来进一步向前迈进。从类参考到类帮助者，本章涵盖许多其他 OOP 语言中没有的功能，或者至少在实现方式上有显着差异。

重点是类，以及在运行时操作类，在第 16 章中介绍反射和属性时，将进一步讨论该主题。

12.1 类方法和类数据

当使用 Object Pascal 和大多数其他 OOP 语言定义一个类时，您将定义该类的对象（或实例）的数据结构以及可以在此类对象上执行的操作。但是，也有可能定义类的所有对象之间共享的数据以及可以为该类调用的方法，而与该类创建的任何实际对象无关。

要在 Object Pascal 中声明一个类方法，只需在它前面添加 class 关键字，就可以在过程和函数中看到它：

```
type
  TMyClass = class
    class function ClassMeanValue: Integer;
```

给定类 TMyClass 的对象 MyObject，可以通过将方法应用于对象或整个类来调用该方法：

```
var
  MyObject: TMyClass;
begin
  ...
  I := TMyClass.ClassMeanValue;
  J := MyObject.ClassMeanValue;
```

这种语法意味着您甚至可以在创建类的对象之前调用类方法。在某些情况下，类的场景仅由类方法组成，其隐含的想法是您将永远不会创建这些类的对象（可以通过将 Create 构造函数声明为私有来强制执行某些操作）。

❖ 通常，在不允许使用全局函数的 OOP 语言中，通常使用类方法，尤其是仅使用类方法构成的类。Object Pascal 仍然可以让您声明老式的全局函数，但是近年来，系统库和开发人员编写的代码越来越趋于一致地使用类方法。使用类方法的优点是它们在逻辑上绑定到一个类，该类充当一组相关函数的名称空间。

12.1.1 类数据

类数据是在类的所有对象之间共享的数据，提供全局存储，但提供类特定的访问权限（包括访问限制）。您如何声明类数据？

只需定义该类的新部分，并用 class var 关键字组合标记即可：

```
type
  TMyData = class
  private
    class var CommonCount: Integer;
```

```
    public
      class function GetCommon: Integer;
```

class var 节引入了一个包含一个或多个声明的块。您可以使用 var 部分（这是使用此关键字的一种新方法）来声明同一部分中的其他实例字段（下面是私有的）：

```
type
  TMyData = class
  private
    class var
      CommonCount: Integer;
    var
      MoreObjectData: string;
  public
    class function GetCommon: Integer;
```

除了声明类数据之外，您还可以定义类属性，这将在后面的部分中介绍。

12.1.2 虚拟类方法和隐藏的 self 参数

虽然在编程语言之间共享类方法的概念，但是 Object Pascal 实现具有一些特殊性。首先，类方法具有隐式（或隐藏）的 self 参数，与实例方法非常相似。但是，此隐藏的 self 参数是对类本身的引用，而不是对类实例的引用。

乍一看，类方法具有指向该类本身的隐藏参数的事实似乎非常无用。毕竟，编译器知道方法的类。但是，有一个独特的语言功能可以解释这一点：与大多数其他语言不同，Object Pascal 中的类方法可以是虚拟的。在派生类中，您可以重写基类方法，就像对常规方法一样。

- 对虚拟类方法的支持与对虚拟构造函数的支持（它们是某种特殊目的的类方法）联系在一起。在许多编译和强类型化的 OOP 语言中都找不到这两个功能。

12.1.3 类 Static（静态）方法

为了平台兼容性，已在该语言中引入了类 Static（静态）方法。普通类方法和类静态方法之间的区别在于，类静态方法没有对其自身类的引用（没有 self 参数指示类本身），并且不能是虚拟的。

这是一个简单的示例，其中注释了一些错误的语句，该语句摘自 ClassStatic 应用程序项目：

```
type
  TBase = class
  private
    Tmp: Integer;
  public
    class procedure One;
    class procedure Two; static;
    ...
  end;

class procedure TBase.One;
begin
  // 错误：无法在此处访问实例成员“Tmp”
  // Show (Tmp);
  Show ('one');
```

```
    Show (self.ClassName);
end;

class procedure TBase.Two;
begin
    Show ('two');
    // 错误：未声明的标识符：'self'
    // Show (self.ClassName);
    Show (ClassName);
    Two;
end;
```

在这两种情况下，您都可以直接调用这些类方法或通过一个对象调用它们：

```
    TBase.One;
    TBase.Two;
    Base := TBase.Create;
    Base.One;
    Base.Two;
```

有两个有趣的功能使类静态方法在 Object Pascal 中有用。首先是它们可以用来定义类属性，如下一节所述。第二个原因是类静态方法是完全 C 语言兼容的，如下所述。

静态类方法和 Windows API Callbacks（回调）

它们没有隐藏的 self 参数，这一事实意味着可以将静态类方法作为回调函数传递给操作系统（例如，在 Windows 上）。在实践中，您可以使用 stdcall 调用约定声明一个静态类方法，并将其用作直接的 Windows API 回调，就像我对 StaticCallBack 应用程序项目的 TimerCallBack 方法所做的那样：

```
type
    TFormCallBack = class(TForm)
        ListBox1: TListBox;
        procedure FormCreate(Sender: TObject);
    private
        class var
            NTimerCount: Integer;
    public
        class procedure TimerCallBack (hwnd: THandle;
            uMsg, idEvent, dwTime: Cardinal); static; stdcall;
    end;
```

回调将类数据用作计数器。OnCreate 处理程序调用 SetTimer API 并传递类静态过程的地址：

```
procedure TFormCallBack.FormCreate(Sender: TObject);
var
    Callback: TFNTimerProc;
begin
    NTimerCount := 0;
    Callback := TFNTimerProc(@TFormCallBack.TimerCallBack);
    SetTimer(Handle, TIMERID, 1000, Callback);
end;
```

- TFNTimeProc 的参数是方法指针，这就是为什么类静态方法的名称以@或使用 Addr 函数开头的原因。这是因为我们需要获取方法地址，而不是执行方法。

现在实际的回调函数会增加计时器并更新表单，并使用相应的全局变量引用它，因为类方法无法将表单引用为 self：

```
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;
```

try-except 块在那里，以避免将任何异常发送回 Windows。

回调或 DLL 函数必须始终遵循的规则。

12.1.4 类属性

使用类静态方法的原因之一是实现类属性。

什么是类属性？像标准属性一样，它是附加在读写机制上的符号。与标准属性不同，它与类有关，并且必须使用类数据或类 static（静态）方法来实现。TBase 类（同样来自 ClassStatic 应用程序项目）具有以两种不同方式定义的两个类属性：

```
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;
```

12.1.5 具有实例计数器的类

类数据和类方法可用于保存有关整个类的信息。此类信息的一个示例是到目前为止已创建的类实例的数量...减去已销毁的实例的数量。

CountObj 应用程序项目显示了这种情况。该程序并不是非常有用，因为我倾向于只关注特定问题及其解决方案。换句话说，目标对象有一个非常简单的类，只存储一个数值：

```
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;
```

每次创建对象时，程序都会在调用基类的构造函数（如果有）之后增加两个计数器。每次销毁对象时，计数器当前对象的计数器都会减少：

```
constructor TCountedObj.Create (AOwner: TComponent);
begin
  inherited Create;
  Inc (FTotal);
  Inc (FCurrent);
end;

destructor TCountedObj.Destroy;
begin
  Dec (FCurrent);
  inherited Destroy;
end;
```

无需引用特定对象即可访问类信息。实际上，很可能在给定的时间内存中没有对象：

```
class function TCountedObj.GetTotal: Integer;
begin
  Result := FTotal;
end;
```

您可以使用以下代码显示当前状态：

```
Label1.Text := TCountedObj.GetCurrent.ToString + '/' +
  TCountedObj.GetTotal.ToString;
```

在演示中，这是在计时器中执行的，计时器会更新标签，因此它不需要引用任何特定的对象实例，也不需要任何手动操作直接触发它。相反，应用程序项目中的按钮仅创建并释放了一些对象……或在内存中留下了一些东西（实际上程序有潜在的内存泄漏）。

12.2 类构造函数（和析构函数）

类构造器提供了一种初始化与类相关的数据的方法，并且它们具有类初始化器的作用，因为它们实际上并没有构造任何东西。类构造函数与标准实例构造函数无关：它仅是用于在使用类之前初始化类本身的代码。例如，类构造函数可以为类数据设置初始值，为类加载配置或支持文件，等等。

在 Object Pascal 中，类构造函数是单元初始化代码的替代方法。

如果两者都存在（在一个单元中），则将首先执行类构造函数，然后再执行单元初始化代码。相反，您可以定义将在完成代码之后执行的类析构函数。

但是，一个重要的区别是，如果在程序中编译单元时，单元初始化代码将始终执行，则只有在实际使用该类的情况下，类构造函数和析构函数才会链接。 这意味着使用类构造函数比使用初始化代码更加友好 Linker（链接器）。

❖ 换句话说，对于类构造函数和析构函数，如果未链接类型，则初始化代码不是程序的一部分，也不执行。与传统情况相反，初始化代码甚至可能导致 Linker（链接器）引入一些类代码，即使它从未在其他地方实际使用过。实际上，这是与手势框架一起引入的，如果不使用，则大量代码不会编译到可执行文件中。

用编码术语，您可以编写以下内容（请参阅 ClassCtor 应用程序项目）：

```
type
  TTestClass = class
  public
```

```
        class var
          StartTime: TDateTime;
          EndTime: TDateTime;
      public
        class constructor Create;
        class destructor Destroy;
      end;
```

该类具有两个类数据字段，分别由类构造函数初始化和由类析构函数修改，而单元的初始化和完成部分则使用以下数据字段：

```
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));
```

发生的结果是启动序列按预期方式工作，并且在显示信息时类数据已经可用。相反，当关闭时，在类析构函数分配值之前执行 ShowMessage 调用。

请注意，尽管 Create 和 Destroy 是很好的默认值，但您可以给类构造函数和析构函数指定任何名称。但是，您不能定义多个 Tiple 类构造函数或析构函数。如果尝试，编译器将发出以下错误消息：

[DCC Error] ClassCtorMainForm.pas(34): E2359 Multiple class constructors in class TTestClass: Create and Foo

（[DCC 错误] ClassCtorMainForm.pas（34）：E2359 类 TTestClass 中的多个类构造函数：Create 和 Foo）

12.2.1 RTL 中的类构造函数

已经有一些 RTL 类已经利用了这种语言功能，例如 Exception 类，它定义了一个类构造函数（带有下面的代码）和一个类析构函数：

```
class constructor Exception.Create;
begin
  InitExceptions;
end;
```

InitExceptions 过程先前在 System.SysUtils 单元的初始化部分中被调用。

通常，我认为使用类构造函数和析构函数比使用单元初始化和终止更好。在大多数情况下，这只是语法糖，因此您可能不希望返回并更改现有代码。但是，如果您面临初始化数据结构的风险，那么您永远不会使用（因为从未创建过该类型的类），因此转移到类构造函数将提供绝对的优势。显然，在通用库中通常不是应用程序代码，而是您没有使用所有功能的情况。

❖ 使用类构造函数的一个非常特殊的情况是泛型类。我将在专注于泛型的一章中介绍它。

12.2.2 实现 Singleton Pattern（单例模式）

对于某些类，创建一个且只有一个单个实例是有意义的。Singleton 模式（另一种非常常见的设计模式）需要这样做，并且还建议为该对象提供一个全局访问点。

单例模式可以通过多种方式实现，但是一种经典的方法是将用于访问唯一实例的函数完全称为 Instance。在许多情况下，实现也遵循延迟初始化规则，因此在程序启动时不会创建此全局实例，而仅在第一次时才创建。在下面的实现中，我利用了类数据，类方法以及类析构函数来进行最终清理。以下是相关代码：

```
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;
```

您可以通过编写以下内容来获取该类的单个实例（无论是否已创建该实例）：

```
var
  ASingle: TSingleton;
begin
  ASingle := TSingleton.Instance;
```

此外，您可以隐藏常规的类构造函数，将其声明为私有，这样就很难在不遵循模式的情况下创建类的对象。

12.3 类参考

看了几个与方法有关的主题之后，我们现在可以继续讨论类引用的主题，并进一步扩展动态创建组件的示例。要记住的第一点是，类引用不是类，不是对象，也不是对对象的引用；它只是对类类型的引用。

类引用类型确定类引用变量的类型。听起来令人困惑？几行代码可能使这一点更加清楚。假设您已定义类 TMyClass。现在，您可以定义与该类相关的新类引用类型：

```
type
  TMyClassRef = class of TMyClass;
```

现在，您可以声明两种类型的变量。第一个变量引用一个对象，第二个变量引用一个类：

```
var
  AClassRef: TMyClassRef;
```

```
        AnObject: TMyClass;
    begin
        AClassRef := TMyClass;
        AnObject := TMyClass.Create;
```

您可能想知道使用什么类引用。通常，类引用允许您在运行时操作类数据类型。您可以在合法使用数据类型的任何表达式中使用类引用。实际上，这种表达并不多，但是很少有这种情况。 最简单的情况是创建对象。我们可以按如下方式重写上面的两行：

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

这次，我将 Create 构造函数应用于类引用，而不是实际的类；我使用了一个类引用来创建该类的对象。

❖ 类引用与其他 OOP 语言中可用的元类的概念有关。但是，在 Object Pascal 中，类引用本身不是类，而只是定义对类数据的引用的特定类型。因此，与元类(描述其他类的类)的类比有点误导。实际上，TMetaclass 也是 C ++ Builder 中使用的术语。

当您具有类引用时，可以将其应用于相关类的类方法。 因此，如果 TMyClass 具有称为 Foo 的类方法，则可以编写以下任一方法：

```
TMyClass.Foo
AClassRef.Foo
```

如果类引用不支持适用于类类型的相同类型兼容性规则，那将不是非常有用。当声明一个类引用变量（例如上面的 MyClassRef）时，您可以为其分配该特定类和任何子类。因此，如果 MyNewClass 是我的类的子类，您还可以编写

```
AClassRef := MyNewClass;
```

现在要了解为什么这确实很有趣，您必须记住可以为类引用调用的类方法可以是虚拟的，因此特定的子类可以覆盖它们。使用类引用和虚拟类方法，您可以在类方法级别上实现一种多态形式，而其他静态 OOP 语言中很少（如果有的话）。还请考虑每个类都继承自 TObject，因此您可以将每个 TObject 的方法应用于每个类引用，包括 InstanceSize， ClassName， ParentClass 和 InheritsFrom。我将在第 17 章中讨论这些类方法和 TObject 类的其他方法。

12.3.1 RTL 中的类引用

System 单元和其他核心 RTL 单元声明了很多类引用，包括以下内容：

```
TClass = class of TObject;
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

特别是，TClass 类引用类型可用于存储对您在 Object Pascal 中编写的任何类的引用，因为每个类最终都是从 TOb 对象派生的。而是在基于 FireMonkey 或 VCL 的默认 Object Pascal 项目的源代码中使用 TFormClass 引用。实际上，两个库的 Application 对象的 CreateForm 方法都需要将要创建的表单类作为参数：

```
Application.CreateForm(TForm1, Form1);
```

第一个参数是类引用，第二个参数是将接收对创建的对象实例的引用的变量。

12.3.2 使用类引用创建组件

Object Pascal 中类引用的实际用途是什么？能够在运行时操作数据类型是环境的基本要素。通过从“组件面板”中选择一个新组件将其添加到表单中时，您将选择一种数据类型并创建该数据类型的对象。（实际上，这是开发环境在幕后为您提供的功能。）为了让您更好地了解类引用的工作方式，我建立了一个名为 ClassRef 的应用程序项目。此示例显示的表单非常简单。它具有三个单选按钮，位于窗体上部的面板内部。选择这些单选按钮之一并单击表单后，就可以创建按钮标签指示的三种类型的新组件：单选按钮，常规按钮和编辑框。为了使该程序正常运行，需要更改这三个组件的名称。该表格还必须具有一个类引用字段：

```
private
   FControlType: TControlClass;
   FControlNo: Integer;
```

每次用户单击三个单选按钮之一时，第一个字段都会存储新的数据类型，从而更改其状态。这是三种方法之一：

```
procedure TForm1.RadioButtonRadioChange(Sender: TObject);
begin
   FControlType := TRadioButton;
end;
```

其他两个单选按钮具有与此类似的 OnChange 事件处理程序，将值 TEdit 或 TButton 分配给 FControlType 字段。表单的 OnCreate 事件的处理程序中也存在类似的分配，用作初始化方法。当用户单击覆盖表单大部分表面的 Layout 控件时，将执行代码中有趣的部分。我选择了表单的 OnMouseDown 事件来保存鼠标单击的位置：

```
procedure TForm1.Layout1MouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Single);
var
   NewCtrl: TControl;
   NewName: String;
begin
   // 创建控件
   NewCtrl := FControlType.Create (Self);
   // 暂时隐藏它，以避免闪烁
   NewCtrl.Visible := False;
   // 设置父级和位置
   NewCtrl.Parent := Layout1;
   NewCtrl.Position.X := X;
   NewCtrl.Position.Y := Y;
   // 计算唯一的名称（和文本）
   Inc (FControlNo);
   NewName := FControlType.ClassName + FControlNo.ToString;
   Delete (NewName, 1, 1);
   NewCtrl.Name := NewName;
   // 现在显示
   NewCtrl.Visible := True;
end;
```

此方法的代码的第一行是关键。它创建一个存储在 FControlType 字段中的类数据类型的新对象。我们只需将 Create 构造函数应用于类引用即可完成此操作。现在，您可以设置 Parent 属性的值，设置新组件的位置，为其指定一个名称（该名称也将自动用作 Text）并使其可见。

特别注意用于构建名称的代码；为了模仿 Object Pascal 的默认命名约定，我

使用 TObject 类的类方法使用表达式 FControlType.ClassName 命名了该类的名称。然后，我在名称末尾添加了一个数字，并删除了字符串的首字母。

对于第一个单选按钮，基本字符串为 TRadioButton，在末尾加上 1，并在类名 RadiooButton1 的开头减去 T。看起来很熟悉？

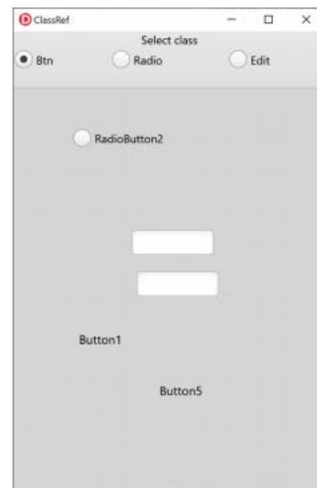

图 12.1：Window 下 ClassRef 应用程序的输出示例

您可以在图 12.1 中看到该程序的输出示例。请注意，命名与 IDE 使用的命名并不完全相同，IDE 对每种类型的控件使用一个单独的计数器。该程序改为对所有组件使用单个计数器。如果以 ClassRef 应用程序的形式放置一个单选按钮，一个按钮和一个编辑框，它们的名称将分别为 RadioButton1，Button2 和 Edit3，如图中所示（尽管该编辑没有名称的可见描述）　）。

顺便说一句，请考虑一旦创建了通用组件，就可以使用反射以一种非常动态的方式访问其属性，在第 16 章中将详细介绍该主题。在同一章中，我们将看到其他方法 在类引用旁边引用类型和类信息。

12.4 类和记录助手

正如我们在第 8 章中看到的那样，类之间的继承概念是一种扩展类以提供新功能的方式，而不会以任何方式影响原始实现。这是所谓的开放式封闭原则的实现：数据类型已完全定义（封闭），但仍可修改（开放）。

尽管类型继承是一种非常强大的机制，但在某些情况下它并不理想。事实是，当您使用现有库和复杂库时，您可能希望能够在不继承新库的情况下扩充数据类型。当以某种自动方式创建对象时，尤其如此，而替换它们的创建可能会非常复杂。

对于 Object Pascal 开发人员而言，一个相当明显的场景是组件的使用。如果要向组件类添加方法以为其提供一些新行为，则确实可以使用继承，但这意味着：创建新的派生类型，创建要安装的包，替换表单中的所有现有组件，以及其他。使用新的组件类型（同时影响表单定义和源代码文件的操作）设计图面。

另一种方法是使用类或记录助手。这些特殊用途的数据类型可以使用新方法扩展现有类型。即使它们确实有一些限制，类助手也可以让您处理一个我刚刚概述的情况，只需将新方法添加到现有组件即可，而无需修改实际的组件类型。

❖ 实际上，我们已经看到了一种替代方法，该方法通过使用继承和同名类（插入程序类惯用法）来扩展库类，而无需完全替换其引用。我在上一章的最后一节中介绍了该惯用法。类助手提供了一个更简洁的模型，但是不能像上一章的应用程序那样使用它们来替换虚拟方法或实现额外的接口。

12.4.1 类 Helper（助手）

类助手是一种向您无权修改的类（如库类）添加方法和属性的方法。使用类助手在您自己的代码中扩展类确实很不寻常，因为在这种情况下，您通常应该继续进行并更改实际的类。

在类助手中，您不能执行的操作是添加实例数据，因为数据应该存在于实际对象中，并且这些对象是由其原始类定义的，或者可以通过虚拟方法（又由原始类的物理结构定义）来实现。换句话说，助手类只能添加或替换现有类的非虚拟方法。这样，即使该类不知道该方法的存在，您也可以将新方法应用于原始类的对象。

如果不清楚（可能不是这样），我们来看一个示例（摘自 ClassHelperDemo 应用程序项目，这只是您不应该做的演示，请使用类助手来增加您自己的类）：

```
type
  TMyObject = class
  protected
    Value: Integer;
    Text: string;
  public
    procedure Increase;
  end;

  TMyObjectHelper = class helper for TMyObject
  public
    procedure Show;
  end;
```

前面的代码声明一个类和该类的助手。这意味着对于 TMyObject 类型的对象，您可以调用原始类的方法以及类助手的方法：

```
Obj := TMyObject.Create;
Obj.Text := 'foo';
Obj.Show;
```

helper 类方法成为该类的一部分，并且可以像其他任何方法一样使用 Self 来引用当前对象（该类之所以有用，因为没有实例化类助手），如以下代码所示：

```
procedure TMyObjectHelper.Show;
begin
  Show (Text + ' ' + IntToStr (Value) + ' -- ' + ClassName + ' -- ' + ToString);
end;
```

最后，请注意，助手类方法可以覆盖原始方法。在代码中，我向类和助手类都添加了 Show 方法，但是仅调用了助手类方法！

当然，使用类助手器语法在同一单元甚至同一程序中声明一个类和对该类的扩展几乎没有意义。我在演示中这样做是为了使您更容易理解此合成税的技术性。类助手程序不应用作开发应用程序体系结构的通用语言构造，而应主要用于扩展您没有源代码或不想更改以避免未来冲突的库类。

还有更多适用于类助手的规则。类助手方法：

- 可以具有与类中原始方法不同的访问说明符。
- 可以是类方法或实例方法，类变量和属性
- 可以是虚拟方法，可以在派生类中重写它(尽管我发现这在实用上有点尴尬)。
- 可以引入额外的构造函数。
- 可以将嵌套常量添加到类型定义中

他们设计上缺少的唯一功能是实例数据。还要注意，当类助手在作用域中变得可见时，它们将被启用。您需要在声明类助手器的单元上添加一个 uses 语句 Referring，以查看其方法，而不仅仅是在编译过程中包含一次。

❖ 在相当长的一段时间内，Delphi 编译器中出现错误，最终导致类助手可以访问其助手的类的私有字段，而不管声明该类所在的单元如何。这种“ hack”基本上破坏了面向对象的编程封装规则。为了增强可见性语义，最新版本的 Object Pascal 编译器（从 Delphi 10.1 Berlin 起）中的类和记录助手不能访问类的私有成员或它们扩展的记录。确实，这已经导致现有代码无法正常工作，而这些代码正是利用这种黑客手段进行老化的（从来没有打算成为语言功能）。

列表框的类助手器

类助手的实际用途是为库类提供额外的方法。原因是您不想直接更改这些类（即使您有源代码，也不想真正编辑核心库源代码）或从它们继承（因为这将迫使您替换组件库中的组件）。在设计时填写表格）。

例如，考虑一个简单的情况：您想要一种简单的方法来获取列表框当前所选内容的文本。无需编写经典代码：

```
ListBox1.Items [ListBox1.ItemIndex]
```

您可以按如下方式定义类助手程序（来自 ControlHelper 项目）：

```
type
  TListboxHelper = class helper for TListBox
    function ItemIndexValue: string;
  end;

function TListboxHelper.ItemIndexValue: string;
begin
  Result := '';
  if ItemIndex >= 0 then
    Result := Items [ItemIndex];
end;
```

现在，您可以将列表框的选定项引用为：

```
Show (ListBox1.ItemIndexValue);
```

这只是一个非常简单的案例，但是它以非常实际的方式展示了这个想法。

12.4.2 类助手和继承

助手的最大限制是在给定的时间，每个类只能有一个助手。如果编译器遇到两个助手程序类，则第二个将替换第一个。没有链式类助手器的方法，也就是说，没有一个类助手器可以进一步扩展已经由另一个类助手器扩展的类。

此问题的部分解决方案来自以下事实：您可以为一个类引入类助手器，并为继承的类添加其他类助手器...但是您不能直接从另一个类助手器继承类助手器。我真的不鼓励使用复杂的类助手器结构，因为它们确实可以将您的代码变成一些非常复杂的代码。

TGUID 记录就是一个例子，TGUID 记录是一种 Windows 数据结构，您可以在 Object Pascal 中的各个平台之间实际使用它，它具有一个助手程序，可以添加一些常见的功能：

```
type
  TGuidHelper = record helper for TGUID
    class function Create(const B: TBytes): TGUID; overload; static;
    class function Create(const S: string): TGUID; overload; static;
    // ... 更多省略创建重载
```

```
        class function NewGuid: TGUID; static;
        function ToByteArray: TBytes;
        function ToString: string;
    end;
```

您可能已经注意到 TGuidHelper 是记录助手而不是类助手。

是的，记录可以像类一样拥有帮助者。

12.4.3 使用类助手添加控件枚举

库中的任何 Delphi 组件都会自动定义一个枚举器，可用于处理每个拥有的组件或子组件。例如，在表单方法中，您可以通过以下方式枚举表单所拥有的组件：

```
for var AComp in self do
  ... // use AComp
```

另一个常见操作是 navigate（导航）子控件，该子控件仅包含具有直接父窗体形式的可视组件（不包括诸如 TMainMenu 的非可视组件）（不包括子控件托管的控件，如面板上的按钮）。我们可以用来编写用于循环遍历子控件的简单代码的一种技术是通过编写以下类助手程序，向 TWinControl 类添加新的枚举：

```
type
  TControlsEnumHelper = class helper for TWinControl
  type
    TControlsEnum = class
    private
      NPos: Integer;
      FControl: TWinControl;
    public
      constructor Create (aControl: TWinControl);
      function MoveNext: Boolean;
      function GetCurrent: TControl;
      property Current: TControl read GetCurrent;
    end;
  public
    function GetEnumerator: TControlsEnum;
  end;
```

❖ 该助手程序用于 TWinControl 而不是 TControl 的原因是，只有具有 Windows 句柄的控件才能成为其他控件的父级。这基本上不包括图形控件。

这是助手程序的完整代码，包括其唯一的方法以及嵌套类 TControlsEnum 的方法：

```
{ 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;
```

现在，如果我们创建如图 12.2 所示的形式，就可以在各种情况下测试枚举。 第一种情况是我们专门为该代码编写的，它枚举了以下形式的子控件：

图 12.2：用于在设计时在 Delphi IDE 中测试控件枚举助手的表单

```
procedure TControlEnumForm.BtnFormChildClick(Sender: TObject);
begin
  Memo1.Lines.Add ('Form Child');
  for var ACtrl in self do
    Memo1.Lines.Add (ACtrl.Name);
  Memo1.Lines.Add ('');
end;
```

这是 Memo 控件中操作的输出，列出了表单子控件，但未列出该面板为父级的其他组件或控件：

```
Form Child
Memo1
BtnFormChild
Edit1
CheckBox1
RadioButton1
Panel1
BtnFormComps
BtnButtonChild
```

如果我们列举所有组件，将显示完整列表。但是，使用我在本节开头显示的代码会遇到问题，因为我们已经用新版本（在类帮助器中）覆盖了 GetNumerator 方法，并且由于这个原因，我们无法直接访问基本的 TComponent 枚举器。该助手程序是为 TWinControl 定义的，因此我们可以使用一个技巧。如果我们将对象转换为 TComponent，则代码将调用标准的预定义枚举数：

```
procedure TControlEnumForm.BtnFormCompsClick(Sender: TObject);
begin
  Memo1.Lines.Add ('Form Components');
  for var AComp in TComponent(self) do
    Memo1.Lines.Add (AComp.Name);
  Memo1.Lines.Add ('');
end;
```

这是输出，列出了比上一个列表更多的组件：

```
Form Components
Memo1
BtnFormChild
Edit1
CheckBox1
```

```
RadioButton1
Panel1
BtnPanelChild
ComboBox1
BtnFormComps
BtnButtonChild
MainMenu1
```

在 ControlsEnum 应用程序项目中，我还添加了用于枚举面板的子控件和按钮之一的子控件的代码（主要用于测试列表为空时枚举器是否正常工作）。

12.4.4 记录内在类型的助手

记录助手概念的进一步扩展是能够向本地（或编译器固有）数据类型添加方法的功能。尽管使用了相同的"记录助手"语法，但这并不应用于记录，而是应用于常规数据类型。

❖ 记录助手目前用于扩充和向本地数据类型添加类似方法的操作，但是将来可能会改变。 当今的运行时库定义了一些将来可能会消失的本机帮助程序，保留了使用这些帮助程序的代码编写方式……但是破坏了定义它们的代码的兼容性。 这就是为什么您不应该过度使用此功能的原因，即使它确实非常好用。

内在类型助手在实践中如何工作？让我们考虑以下对 Integer 类型的帮助器的定义：

```
type
  TIntHelper = record helper for Integer
    function AsString: string;
  end;
```

现在给定一个整数变量 N，您可以编写：

```
N.AsString;
```

您如何定义该伪方法，以及如何引用该变量的值？

通过扩展 self 关键字的含义以引用该值，该函数将应用于：

```
function TIntHelper.AsString: string;
begin
  Result := IntToStr (self);
end;
```

请注意，您还可以将方法应用于常量，例如：

```
Caption := 400000.AsString;
```

但是，对于较小的值，您不能这样做，因为编译器会解释较小类型的常量。因此，如果要获取值 4 作为字符串，则必须使用第二种形式：

```
Caption := 4.AsString; //  不!
Caption := Integer(4).AsString; // ok
```

或者，您可以通过定义其他助手器来使第一个语句得以编译：

```
type
  TByteHelper = record helper for Byte...
```

正如我们在第 2 章已经看到的那样，您实际上不需要为诸如 Integer 和 Byte 之类的代码编写上述代码，因为运行时库为大多数核心数据类型定义了一个非常全面的类助手列表，包括以下内容：在 System.SysUtils 单元中定义：

```
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
TSmallIntHelper = record helper for SmallInt
TWordHelper = record helper for Word
TCardinalHelper = record helper for Cardinal
TIntegerHelper = record helper for Integer
TInt64Helper = record helper for Int64
TUInt64Helper = record helper for UInt64
TNativeIntHelper = record helper for NativeInt
TNativeUIntHelper = record helper for NativeUInt
TBooleanHelper = record helper for Boolean
TByteBoolHelper = record helper for ByteBool
TWordBoolHelper = record helper for WordBool
TLongBoolHelper = record helper for LongBool
TWordBoolHelper = record helper for WordBool
```

当前在其他单元中定义了一些其他内部类型帮助器，例如：

```
// 系统字符：
TCharHelper = record helper for Char
// 系统类：
TUInt32Helper = record helper for UInt32
```

鉴于我在本书的第一部分中已经在许多示例中介绍了这些帮助器的用法，因此无需在此重复它们。本节添加的内容描述了如何定义内部类型帮助器。

```
// System.Character:
TCharHelper = record helper for Char
// System.Classes:
TUInt32Helper = record helper for UInt32
```

鉴于我在本书的第一部分中已经在许多示例中介绍了这些帮助器的用法，因此无需在此重复它们。本节添加的内容描述了如何定义内部类型助手器。

12.4.5 类型别名的助手

如我们所见，不可能为同一个类型定义两个助手，更不用说固有类型了。那么，如何向本机类型（如 Integer）添加额外的直接操作？尽管没有明确的解决方案，但是有一些可能的解决方法（缺少复制内部类帮助程序源代码并使用多余的方法复制它的方法）。

我喜欢的解决方案是类型别名的定义。类型别名被编译器视为全新类型，因此它可以具有自己的帮助程序，而无需替换原始类型的帮助程序。现在，由于类型是分开的，您仍然不能将两个类助手的方法应用于相同的变量，但是其中之一将被抛弃。让我用代码术语解释一下。假设您创建一个类型别名，例如：

```
MyInt = type Integer;
```

现在，您可以为此类型定义一个助手器：

```
type
  TMyIntHelper = record helper for MyInt
    function AsString: string;
  end;

function MyIntHelper.AsString: string;
begin
  Result := IntToStr (self);
end;
```

如果声明此新类型的变量，则可以调用方法特定的帮助器，但仍然使用强制转换来调用 Integer 类型的帮助器方法：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  mi: MyInt;
begin
  mi := 10;
  Show (mi.asString);
  // Show (mi.toString); //这不起作用
  Show (Integer(mi).ToString)
end;
```

该代码位于 TypeAliasHelper 应用程序项目中，供您尝试进一步的变体。

第十三章 对象和内存

本章重点讨论一个非常具体且非常重要的主题，即 Object Pascal 语言中的内存管理。该语言及其运行时环境提供了一种非常独特的解决方案，介于 C ++样式的手动内存管理和 Java 或 C＃垃圾回收之间。

这种介于两者之间的方法的原因是，它有助于避免大多数手动内存管理的麻烦（但显然不是全部），而没有从多余的内存分配到不确定的处置等方面的约束和由垃圾收集引起的问题。

❖ 我没有特别的意图去研究 GC（垃圾收集）策略的问题以及它们如何在各种平台中实现。这更多是一个研究主题。与此相关的是，在受限设备（例如移动设备）上，GC 似乎还不理想，但是某些相同的问题也适用于每个平台。 忽略 Windows 应用程序内存消耗的趋势已为我们带来了每个占用 1 GB 内存的小型实用程序。

但是，使 Object Pascal 变得有些复杂的事实是，变量根据其数据类型使用内存，某些类型使用引用计数，而另一些类型则使用更传统的方法。基于组件的所有权模型和其他一些选项使内存管理成为一个复杂的话题。

本章从现代编程语言中的内存管理的一些基础以及对象引用模型背后的概念开始，以解决此问题。

多年来，Delphi 移动编译器提供了另一种称为 ARC 的存储模型，即自动参考计数。由 Apple 以其自己的语言推广，ARC 添加了对跟踪和计数对象引用的编译器支持，当不再需要 9taht 时，销毁该对象（引用计数降至零）。这与所有平台上 Delphi 中的接口引用所发生的情况非常相似。从 Delphi10.4 开始，所有平台的语言都不再支持 ARC。

13.1 全局数据，Stack（栈）和 Heap（堆）

任何平台上的任何 Object Pascal 应用程序使用的内存都可以分为两个区域：代码和数据。 程序的可执行文件，其资源（位图和表单描述文件）的部分以及程序使用的库的部分将加载到其存储空间中。 这些内存块是只读的，并且（在 Windows 等平台上）它们可以在多个进程之间共享。

查看数据部分会更有趣。 Object Pascal 程序的数据（就像用大多数其他语言编写的程序的数据一样）存储在三个明显不同的区域中：全局内存，堆栈和堆。

13.1.1 Global（全局）Memory

当 Object Pascal 编译器生成可执行文件时，它确定存储程序整个生命周期中存在的变量所需的空间。

在接口或单元的实现部分中声明的全局变量属于此类别。请注意，如果全局变量是类类型（但也可以是字符串或动态数组），则全局存储器中仅存储 4 字节或 8 字节的对象引用。

您可以使用“Project （项目）| Information（信息）”来确定全局内存的大小。编译程序后的信息菜单项。您要查看的特定字段是“数据大小”。图 13.1 显示了近 50K 的全局数据（48,956 字节）的使用情况，其中包括您的程序和所使用的库的

全局数据。

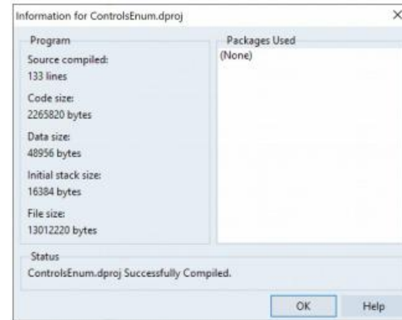

图 13.1：有关已编译程序的信息

全局存储器有时称为静态存储器，因为一旦加载了程序，变量将保留在其原始位置，或者永远不会释放该存储器。

13.1.2 Stack（栈）

Stack（栈）是一个动态内存区域，按照 LIFO 顺序分配和释放：后进先出。 这意味着您分配的最后一个内存对象将是第一个被删除的对象。您可以在图 13.2 中看到堆栈存储器的表示。

例程（过程，函数和方法调用）通常将 stack（栈）存储器用于传递参数及其返回值以及在函数或方法中声明的局部变量。例行调用终止后，将释放其在 stack（栈）上的存储区。无论如何，请记住，使用 Object Pascal 的默认寄存器调用约定，只要可能，就将参数传递到 CPU 寄存器中，而不是 stack（栈）中。

还要注意，stack（栈）存储器通常不进行初始化或清理，以节省时间。这就是为什么如果将 Integer 声明为局部变量并仅读取其值的原因，那么您几乎可以找到所有内容。所有局部变量在使用之前都需要初始化。

Stack（栈）的大小通常是固定的，并由编译过程确定。

您可以在“项目”选项的 Linker（链接器）页面中设置此参数。但是，默认通常是确定的。如果收到“stack overflow（栈溢出）”错误消息，则可能是因为您有一个函数会永远递归调用自身，而不是因为栈空间太有限。初始栈大小是“项目”提供的另一条信息。信息对话框。

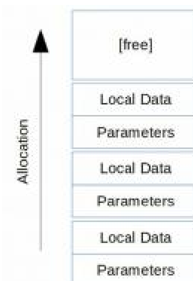

图 13.2：堆栈存储区的表示

13.1.3 Heap（堆）

Heap（堆）是内存分配和释放以随机顺序发生的区域。这意味着，如果您按顺序分配三个内存块，则以后可以按任何顺序销毁它们。Heap（堆）管理器负责所有细节，因此您只需使用低级 GetMem 函数或通过调用构造函数创建对象来请求新的内存，系统将为您返回一个新的内存块（可能重复使用已经丢弃的内存块）。Object Pascal 使用堆来分配每个对象的内存，字符串文本，动态数组以及大多数其他数据结构。

因为它是动态的，所以 Heap（堆）是程序通常存在最多问题的内存区域：

- 每次创建对象时，都需要销毁它。如果不这样做，则是一个称为“内存泄漏”的场景，它不会造成太大的伤害，除非一遍又一遍地重复直到整个 Heap（堆）

内存已满。

- 每次销毁对象时，都必须确保不再使用该对象，并且该程序不会尝试再次销毁它。

任何其他动态创建的数据结构也是如此，但是语言运行时几乎以自动方式来照顾字符串和动态数组，因此您几乎不必担心这些。

13.2 对象参考模型

正如我们在第 7 章中看到的那样，该语言中的对象被实现为引用。class 类型的变量只是指向对象数据所在的堆中内存位置的指针。实际上，还有一些额外的信息，例如类引用，这是访问对象虚拟方法表的一种方式，但这不在本章的重点之内（我将在第 13 章“此指针是否为对象引用”一节中简要介绍该信息）。

我们还看到了如何将一个对象分配给另一个对象仅是引用的副本，因此您将在内存中拥有对单个对象的两个引用。要拥有两个完全独立的对象，您需要创建第二个对象，并将第一个对象的数据复制到该对象（该操作不会自动提供，因为其实现细节可能会因实际数据结构而异）。

用编码术语来说，如果编写以下代码，则不会创建新对象，而是会创建对内存中同一对象的新引用：

```
var
  Button2: TButton;
begin
  Button2 := Button1;
```

换句话说，内存中只有一个对象，而 Button1 和 But ton2 变量均引用该对象，如图 13.3 所示。

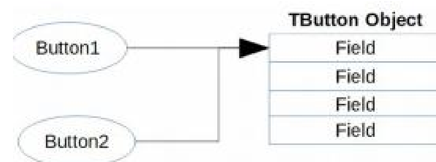

图 13.3：复制对象引用

13.2.1 将对象作为参数传递

将对象作为参数传递给函数或方法时，也会发生类似的情况。一般而言，您只是将引用复制到同一对象，并且在方法或函数中，您可以对该对象执行操作并对其进行修改，而不管参数作为 const 参数传递的事实。

例如，通过编写此过程并按如下所示调用它，可以修改 Button1 的标题，如果愿意，可以修改 AButton 的标题：

```
procedure ChangeCaption (AButton: TButton; Text: string);
begin
  AButton.Text := Text;
end;
// 调用...
ChangeCaption (Button1, 'Hello')
```

如果您需要创建一个新对象怎么办？您基本上必须先创建它，然后复制每个相关属性。一些类（尤其是大多数从 TPersistent 派生而不是从 TComponent 派生的类）定义了一个 Assign 方法来复制对象的数据。例如，你可以写：

```
ListBox1.Items.Assign (Memo1.Lines);
```

即使直接分配这些属性，Object Pascal 也会为您执行类似的代码。实际上，与列表框的 items 属性连接的 SetItems 方法将调用 TStringList 类的 Assign 方法，

该类表示列表框的实际项目。

因此，让我们尝试回顾一下通过修饰符传递给对象的各种参数的作用：

- 如果没有修饰符，则可以对对象及其引用的变量进行任何操作。您可以修改原始对象，但是如果将新对象分配给参数，则该新对象将与原始对象及其引用的变量无关。
- 如果存在 const 修饰符，则可以更改对象的值和调用方法，但不能为该参数分配新的对象。请注意，将对象作为 const 传递没有性能优势。
- 如果有一个 var 修饰符，则可以更改对象中的任何内容，也可以像调用其他 var 参数一样在调用位置中用新对象替换原始对象。约束是您需要将引用传递给变量（而不是通用表达式），并且引用类型必须与参数类型完全匹配。
- 最后，有一个相当未知的选项将对象作为参数传递，称为常量引用，并写为 [ref] const。当参数作为常量引用传递时，其行为与按引用传递（var）相似，但是它允许您传递的参数类型具有更大的灵活性，而无需精确的类型匹配（因为传递子类的对象是允许）。

13.3 内存管理技巧

Object Pascal 中的内存管理受两个简单规则约束：必须销毁创建和分配的每个对象和内存块，并且必须销毁每个对象并释放每个块一次。Object Pascal 支持对动态元素（即不在 stack（栈）和全局内存区域中的元素）进行三种类型的内存管理，本节剩余部分对此进行了详细介绍：

- 每次创建对象时，还应该释放它。如果不这样做，该对象使用的内存将不会释放给其他对象，直到程序终止。
- 创建组件时，可以指定所有者组件，将所有者传递给组件构造函数。所有者组件（通常是表单或数据模块）负责销毁其拥有的所有对象。换句话说，当您释放表单或数据模块时，它将释放它拥有的所有组件。因此，如果您创建了一个组件并将其授予所有者，则不必担心销毁它。
- 当您为字符串，动态数组和接口变量所引用的对象分配内存时（如第 11 章所述），当引用超出范围时，Object Pascal 会自动释放内存。您无需释放字符串：当字符串变得不可访问时，它将释放其内存。

13.3.1 销毁您创建的对象

在最简单的情况下，必须在桌面编译器上创建销毁的临时对象。任何非临时对象都应具有所有者，可以是集合的一部分，或者可以通过某些数据结构来访问，这些数据结构将负责在适当的时候销毁它。

通常，将用于创建和销毁临时对象的代码封装在 try-finally 块中，以便即使在使用时出现问题也可以销毁该对象：

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

另一个常见方案是一个对象被另一个对象使用，该对象成为其所有者：

```
constructor TMyOwner.Create;
```

```
begin
   FSubObj := TSubObject.Create;
end;

destructor TMyOnwer.Destroy;
begin
   FSubObj.Free;
end;
```

有一些常见的更复杂的方案，以防万一直到需要时才创建主题（延迟初始化），或者在不再需要主题时可以将其销毁。要实现延迟初始化，您无需在所有者对象构造函数中创建主题，而是在需要时创建主题：

```
function TMyOwner.GetSubObject: TSubObject
begin
   if not Assigned (FSubObj) then
      FSubObj := TSubObject.Create;
   Result := FSubObj;
end;

destructor TMyOnwer.Destroy;
begin
   FSubObj.Free;
end;
```

注意，释放对象之前不需要测试对象是否已分配，因为这正是 Free 所做的，正如我们将在下一部分中看到的那样。

13.3.2 销毁对象一次

另一个问题是，如果两次调用对象的析构函数，则会收到错误消息。析构函数是一种用于释放对象内存的方法。我们可以为析构函数编写代码，通常覆盖默认的 Destroy 析构函数，以使对象在销毁对象之前执行一些代码。

Destroy 是 TObject 类的虚拟析构函数。销毁对象时，大多数需要自定义清理代码的类都将覆盖此虚拟方法。

永远不要定义新析构函数的原因是，通常通过调用 Free 方法来销毁对象，并且此方法为您调用 Destroy 虚拟析构函数（可能是覆盖的版本）。

正如我刚刚提到的，Free 只是 TObject 类的一种方法，它被所有其他类继承。Free 方法基本上是在调用 Destroy 虚拟析构函数之前检查当前对象（Self）是否不为 nil。

- 您可能想知道，如果对象引用为 nil，为什么可以安全地调用 Free，但是却不能调用 Destroy。原因是 Free 是给定内存位置上的一种已知方法，而虚函数 Destroy 是在运行时通过查看对象的类型确定的，如果该对象不再存在，这将是非常危险的操作。

  这是其 free 的伪代码：

```
procedure TObject.Free;
begin
   if Self <> nil then
      Destroy;
end;
```

接下来，我们将注意力转移到 Assigned 函数。当我们将指针传递给该函数时，它仅测试该指针是否为 nil。因此，至少在大多数情况下，以下两个陈述是等效的：

```
if Assigned (MyObj) then
  ...
if MyObj <> nil then
  ...
```

注意，这些语句仅测试指针是否为 nil；他们不检查它是否是有效的指针。如果您编写以下代码：

```
MyObj.Free;
if MyObj <> nil then
  MyObj.DoSomething;
```

该测试的评估结果为 True，调用该对象的方法时，您会得到一条错误消息。必须意识到调用 Free 不会将对象的引用设置为 nil。

无法将对象自动设置为 nil。您可能对同一个对象有多个参照，但是 Object Pascal 不会对其进行跟踪。同时，在一个方法（例如 Free）中，我们可以对对象进行操作，但对对象引用一无所知-我们用来调用该方法的变量的内存地址。

换句话说，在 Free 方法或类的任何其他方法内部，我们知道对象的内存地址（Self），但是我们不知道引用该对象的变量（例如 MyObj）的内存位置。因此，Free 方法不能影响 MyObj 变量。

但是，当我们调用外部函数通过引用将对象作为参数传递时，这使该函数可以修改参数的原始值。如果是 FreeAndNil 过程，则该范围的现成函数。这是 FreeAndNil 的当前代码：

```
procedure FreeAndNil(const [ref] Obj: TObject); inline;
var
  Temp: TObject;
begin
  Temp := Obj;
  TObject(Pointer(@Obj)^) := nil;
  Temp.Free;
end;
```

过去，参数只是一个指针，但是缺点是您可以将 FreeAndNil 过程传递给原始指针，接口引用和其他不兼容的数据结构。这通常会导致内存损坏和难以发现的错误。从 Delphi 10.4 开始，使用 TObject 类型的 const 引用参数对代码进行了如上所示的修改，将参数限制为对象。

❖ 不少 Delphi 专家认为，永远不要使用 FreeAndNil，因为引用对象的变量的可见性应与其生存期相匹配。如果一个对象拥有另一个对象并且释放对象在析构函数中，则无需将引用设置为 nil，因为它是您将不再使用的对象的一部分。同样，带有 try　finally 块的局部变量将释放它，不需要将其设置为 nil，因为它将超出范围。

附带说明，除 Free 方法外，TObject 还具有 DisposeOf 方法，该方法是该语言几年来对 ARC 的支持。当前，DisposeOf 方法仅调用 Free。

总结一下这些内存清理操作的使用，这里有一些指导原则：

- 始终调用 Free 销毁对象，而不是调用 Destroy 析构函数。
- 使用 FreeAndNil，或在调用 Free 之后将对象引用设置为 nil，除非此后引用立即超出范围。

13.4 内存管理和接口

在第 11 章中，我介绍了接口的内存管理的关键元素，这些元素与对象的管理和引用计数不同。如前所述，接口引用会增加被引用对象的引用计数，但是您

可以将接口引用声明为弱接口以禁用引用计数（但仍要求编译器为您管理引用），也可以使用 unsafe 修饰符，完全禁用对特定参考的任何编译器支持。 在本节中，我们将对该区域进行更深入的介绍，以显示第 11 章中提供的其他示例。

13.4.1 有关 Weak（弱）References（引用）的更多信息

Delphi 用于接口的引用计数模型的一个问题是，如果两个对象相互引用，它们将形成一个循环引用，并且它们的引用计数基本上永远不会为零。弱引用提供了一种打破这些循环的机制，使您可以定义不会增加引用计数的引用。

假设两个接口使用它们的一个字段互相引用，而外部变量引用第一个。第一个对象的引用计数为 2（外部变量和第二个对象的字段）：而第二个对象的引用计数为 1（第一个对象的字段）。图 13.4 描述了此场景。

图 13.4：对象之间的引用可以形成循环，这是弱引用引起的。

现在，随着外部变量超出范围，这两个对象的引用计数将保持为 1，并且将无限期保留在内存中。要解决这种情况，您应该打破循环引用，这很简单，因为您不知道何时执行此操作（应在最后一个外部引用超出范围时执行，实际上对象不知道）。

解决这种情况以及许多类似情况的方法是使用弱引用。如前所述，弱引用是对不增加引用计数的对象的引用。从技术上讲，您可以通过向其应用[weak]属性来定义弱引用。

❖ 属性是第 16 章中介绍的高级 Object Pascal 语言功能。可以说，它们是一种添加有关符号的运行时信息的方法，以便外部代码可以确定如何处理它。

在前面的情况下，如果从第二个对象回到第一个对象的引用是一个弱引用（请参见图 13.5），则由于外部变量超出范围，两个对象都将被销毁。

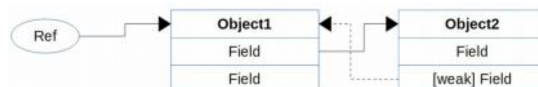

图 13.5：图 13.4 中的引用循环，使用弱引用（虚线）断开。

让我们看看代码中的这种简单情况。首先，ArcExperiments 应用示例声明了两个接口，其中一个引用了另一个：

```
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;
```

该程序的代码定义了两个不同的类，它们实现了每个接口。注意交叉引用的方式（FOwnedBy 和 FSimple 是基于接口的，并且其中两个定义为弱）：

```
type
  TMySimpleClass = class (TInterfacedObject, IMySimpleInterface)
  private
```

```
    [Weak] FOwnedBy: IMyComplexInterface;
  public
    constructor Create(Owner: IMyComplexInterface);
    destructor Destroy (); override;
    procedure DoSomething(bRaise: Boolean = False);
    function RefCount: Integer;
  end;

  TMyComplexClass = class (TInterfacedObject, IMyComplexInterface)
  private
    FSimple: IMySimpleInterface;
  public
    constructor Create();
    destructor Destroy (); override;
    function GetSimple: IMySimpleInterface;
    function RefCount: Integer;
  end;
```

在这里，“complex（费解的）”类的构造函数创建了另一个类的对象：

```
constructor TMyComplexClass.Create;
begin
  inherited Create;
  FSimple := TMySimpleClass.Create (self);
end;
```

请记住，FOwnedBy 字段是弱引用，因此它不会增加其引用的对象（在这种情况下为当前对象（self）的引用计数。给定这种代码结构，我们可以编写：

```
class procedure TMyComplexClass.CreateOnly;
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.FSimple.DoSomething;
end;
```

只要使用弱引用，就不会造成内存泄漏。例如，使用如下代码：

```
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  Log ('Complex = ' + MyComplex.RefCount.ToString);
  MyComplex.GetSimple.DoSomething (False);
```

给定每个构造函数和析构函数的执行日志，您将获得类似以下的日志：

```
Complex class created
Simple class created
Complex = 1
Simple class doing something
Complex class destroyed
Simple class destroyed
```

如果删除代码中的 weak 属性，则会看到内存泄漏，并且（在执行上述代码时）还会看到引用计数的值为 2 而不是 1。

Weak References Are Managed（弱引用被管理）

一个非常重要的元素是弱引用被管理。换句话说，系统在内存中保留了弱引用列表，当对象被销毁时，它会检查是否有弱引用引用该对象，并将其设置为 nil。

这意味着弱引用具有运行时成本。

与传统的弱引用相比，管理弱引用的好处是，您可以检查接口引用是否仍然有效（这意味着它所引用的对象已被破坏）。使用弱引用时，应始终在使用前测试是否已分配该弱引用。

在 ArcExperiments 应用程序示例中，表单具有 IMySimpleInterface 类型的私有字段，声明为弱引用：

```
private
  [weak] MySimple: IMySimpleInterface;
```

在确保该字段仍然有效之前，还有一个按钮为该字段分配了一个引用，另一个使用该字段：

```
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;
```

除非您修改代码，否则如果 Assigned（分配）的测试将失败，因为第一个按钮事件处理程序将创建并立即释放对象，因此弱引用将无效。但是，只要对其进行管理，编译器就可以帮助您跟踪其实际状态（与对对象的引用不同）。

13.4.2 The Unsafe Attribute（不安全属性）

在某些非常特殊的情况下（例如，在创建实例期间），函数可能会返回引用计数设置为零的对象。在这种情况下，为了避免编译器立即删除对象（在将其分配给变量之前，会将其引用计数增加到 1），我们必须将对象标记为““unsafe（不安全）”。

这意味着必须暂时忽略其引用计数，以使代码“safe（安全）”。通过使用新的特定属性[Unsafe]，仅在非常特定的情况下才需要使用此功能，可以完成此行为。

这是语法：

```
var
  [Unsafe] Intf1: IInterface;
  [Result: Unsafe] function GetIntf: IInterface;
```

在通用库中实现构造模式（例如工厂模式）时，可以使用此属性。

❖ 为了支持现在不建议使用的 ARC 内存模型，System 单元使用了不安全的指令，因为它无法在定义之前使用该属性（此单元后来使用）。不应在该单元之外的任何代码中使用此方法，也不再使用它（您可以在$ IFDEF 指令中看到它）。

13.5 Tracking（跟踪）和检查内存

在本章中，我们已经了解了 Object Pascal 中内存管理的基础。在大多数情况下，仅应用此处突出显示的规则就足以确保程序稳定，避免过多的内存使用，并且基本上让您无需进行内存管理。本章后面将介绍一些用于编写健壮应用程序的最佳实践。

在本节中，我重点介绍可用于跟踪内存使用情况，监视异常情况以及查找内存泄漏的技术。这对开发人员来说是重要的知识，即使它并非严格地属于语言的一部分，而是更多的运行时支持。此外，内存管理器的实现取决于目标平台和操作系统，甚至可以在 Object Pascal 应用程序中插入自定义内存管理器（尽管这不是普遍的做法）。

请注意，所有与跟踪内存状态，内存管理器，泄漏检测有关的讨论仅与 heap（堆）内存有关。 stack（栈）和 global（全局）内存的管理方式不同，您基本上无权干预，但是这些内存区域也很少引起任何麻烦。

13.5.1 Memory Status（内存状态）

跟踪堆内存状态如何？RTL 提供了一些方便的功能，GetMemoryManagerState 和 GetMemoryMap。虽然内存管理器状态指示了各种大小的已分配块的数量，但是 heap（堆）映射非常好，因为它在系统级别描述了应用程序的内存状态。您可以通过编写如下代码来检查以下每个存储块的实际状态：

```
for I := Low(aMemoryMap) to High(aMemoryMap) do
begin
  case AMemoryMap[I] of
    csUnallocated: ...        //未分配
    csAllocated: ...          //已分配
    csReserved: ...           //已预留
    csSysAllocated: ...       //系统分配
    csSysReserved: ...        //系统预留
  end;
end;
```

在 ShowMemory 应用程序项目中使用此代码创建应用程序内存状态的图形表示。

13.5.2 FastMM4

在 Windows 平台上，当前的 Object Pascal 内存管理器称为 FastMM4，主要是由 Pierre La Riche 作为开源项目开发的。

FastMM4 优化了内存分配，加快了内存分配速度，并释放了更多 RAM 供以后使用。FastMM4 能够对有效的内存清理，对已删除对象的不正确使用（包括对数据的基于接口的访问），对内存的覆盖和对缓冲区的溢出进行大量的内存检查。它还可以提供有关剩余对象的一些反馈，帮助您跟踪内存泄漏。

实际上，FastMM4 的一些更高级的功能仅在该库的完整版本中可用（在“完整 FastMM4 中的缓冲区溢出”部分中找到），而在标准 RTL 所包含的版本中不可用。这就是为什么如果要具有完整版本的功能，则必须从以下位置下载其完整源代码：

https://github.com/pleriche/FastMM4

❖ <u>该库有一个名为 FastMM5 的新版本，该版本已针对多线程应用程序进行了专门优化，并且可以在大型多核系统上更好地运行。该库的新版本具有 GPL 许可证（适用于开源项目）或其他任何人的付费商业许可证（完全值得）。有关更多信息，请参见 https://github.com/pleriche/FastMM5。</u>

13.5.3 跟踪泄漏和其他全局设置

可以使用“system”单元中的全局设置来调整 FastMM4 的 RTL 版本。请注意，尽管相关的全局声明位于 System 单元中，但实际的内存管理器在 getmem.inc RTL 源代码文件中实现。

同样，默认情况下仅对使用其平台本机内存管理器的 Windows 应用程序（而不是其他操作系统）有效。

最易于使用的设置是 ReportMemoryLeaksOnShutdown 全局变量，它使您可以轻松跟踪内存泄漏。您需要在程序执行开始时将其打开，并且在程序终止时它会告诉您代码（或正在使用的任何库）中是否有内存泄漏。

❖ <u>内存管理器的更高级设置包括用于多线程分配的 NeverSleepOnMMThreadContention 全局变量；函数 GetMinimumBlockAlignment 和 SetMinimumBlockAlignment，它们可以加快某些 SSE 操作的速度，但以占用更多内存为代价；通过调用全局过程 RegisterExpectedMemoryLeak 来注册预期的内存泄漏的能力。</u>

为了演示标准的内存泄漏报告和注册，我编写了一个简单的 LeakTest 应用程序项目。它具有一个带有此 OnClick 处理程序的按钮：

```
var
  P: Pointer;
begin
  GetMem (P, 100); // leak!
end;
```

此代码分配丢失或泄漏的 100 个字节。如果在 IDE 运行时运行 LeakTest 程序并按一次第一个按钮，则在关闭程序时，您将得到一条类似于图 13.6 上部的消息。

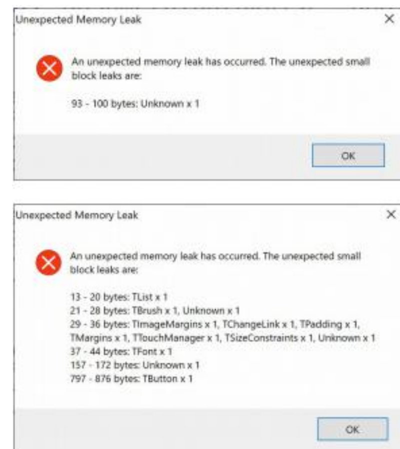

图 13.6：Windows 上的内存管理器在 LeakTest 应用程序终止后报告内存泄漏

程序的另一个“泄漏”是由创建一个 TButton 并将其保留在内存中引起的，但是由于该对象包含许多子元素，因此泄漏报告变得更加复杂，如图 13.6 底部所示。尽管如此，我们对于泄漏本身确实有一些有限的信息。

该程序还为永远不会释放的全局指针分配了一些内存，但是通过按预期方式注册此潜在的泄漏，不会得到报告：

```
procedure TFormLeakTest.FormCreate(Sender: TObject);
begin
  GetMem (GlobalPointer, 200);
```

```
    RegisterExpectedMemoryLeak(GlobalPointer);
  end;
```

同样，此默认泄漏报告仅在默认情况下仅在 Windows 平台上可用，在 Windows 平台上默认情况下实际使用 FastMM4。

13.5.4 Full FastMM4 中的缓冲区溢出

这是一个相当高级的主题，并且是 Windows 平台特有的主题，因此，我建议仅最有经验的开发人员阅读本节。

如果您想更好地控制泄漏报告（如激活基于文件的日志记录），微调分配策略并使用 FastMM4 提供的内存检查，则需要下载完整版本。它由 FastMM4.pas 文件以及配置文件 FastMM4Options.inc 组成。

您只需编辑该注释并取消注释大量指令，就可以编辑该文件来微调设置。 按照惯例，这是通过在$ DEFINE 语句之前放置一个句点，然后将其变成纯注释来完成的，如从 include 文件中摘录的这两行的第一行：

```
{.$DEFINE Align16Bytes} // 注释
{$DEFINE UseCustomFixedSizeMoveRoutines} // 激活设置
```

对于此演示，我打开了以下相关设置，在此报告，以使您了解可用的定义类型：

```
{$DEFINE FullDebugMode}
{$DEFINE LogErrorsToFile}
{$DEFINE EnableMemoryLeakReporting}
{$DEFINE HideExpectedLeaksRegisteredByPointer}
{$DEFINE RequireDebuggerPresenceForLeakReporting}
```

测试程序（在文件夹 FastMMCode 中，为方便起见还包括我使用的 FastMM4 版本的完整源），通过将其设置为第一单元，激活了项目源代码文件中内存管理器的自定义版本：

```
program FastMMCode;
uses
  FastMM4 in 'FastMM4.pas',
  Forms,
  FastMMForm in 'FastMMForm.pas'; {Form1}
```

您还需要 FastMM_FullDebugMode.dll 文件的本地副本才能使其正常工作。

该演示程序通过获取更多可以容纳在本地缓冲区中的文本来导致缓冲区溢出，因为 Length(Caption)大于提供的 5 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;
```

内存管理器在每个内存块的开头和结尾分配带有特殊值的额外字节，并在释放每个内存块时检查这些值。这就是为什么在 FreeMem 调用中出现错误的原因。当您按下按钮（在调试器中）时，您将看到一条很长的错误消息，该错误消息也记录到该文件中：

FastMMCode_MemoryManager_EventLog.txt

这是溢出错误的输出，在分配和释放操作时带有堆栈跟踪，加上当前的堆栈跟踪和内存转储（此处为部分）：

FastMM has detected an error during a FreeMem operation. The block footer has been corrupted.（FastMM 在 FreeMem 操作期间检测到错误。块页脚已损坏。）

The block size is: 5

Stack trace of when this block was allocated (return addresses)（分配该块的时间（返回地址）的堆栈跟踪）:

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（该块当前用于以下类别的对象：未知）

The allocation number is: 381（分配编号是：381）

Stack trace of when the block was previously freed (return addresses)（先前释放该块的时间的堆栈跟踪（返回地址））:

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)（当前堆栈跟踪导致此错误（返回地址））:

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（从指针地址 133DEF8 开始的 256 个字节的当前内存转储）:

46 61 73 74 4D 4D 43 6F 64 [... 省略...]

这并不是很明显，但是它应该提供足够的信息，以使您开始研究错误。请注意，在内存管理器中没有这些设置，您基本上将不会看到任何错误，并且程序将

继续运行...

尽管您可能会遇到随机错误，以防缓冲区溢出影响存储其他内容的内存区域。到那时，您会发现一些奇怪且很难跟踪的错误。

作为一个例子，我曾经看到过对象数据的初始部分的部分覆盖，在该对象中存储了类引用。由于这种内存破坏，该类变得不确定，并且对其虚拟函数之一的每次调用都将严重崩溃……这与程序完全不同区域中的内存写入操作很难联系。

13.5.5 Windows 以外的平台上的内存管理

鉴于内存管理在 Object Pascal 编译器中的工作方式，值得考虑一下确保所有内容都在控制之下的一些选项。

在继续之前，请务必注意，在非 Windows 平台上，Delphi 不使用 FastMM4 内存管理器，因此设置 ReportMemoryLeaksOnShutdown 全局标志以在程序关闭时检查内存泄漏是没有用的。

还有另一个原因，就是通常情况下，没有一种方法可以关闭移动设备上的应用程序，因为应用程序一直保留在内存中，直到被用户或操作系统强行删除为止。

在 macOS，iOS 和 Android 平台上，Object Pascal RTL 直接调用本机 libc 库的 malloc 和 free 函数。监视此平台上的内存使用情况的一种方法是依靠外部平台工具。例如，在 iOS（和 macOS）上，您可以使用 Apple 的 Instruments 工具，这是一个完整的跟踪系统，可以监视在物理设备上运行的应用程序的各个方面。

13.5.6 跟踪每个类 Allocations（分配）

最后，有一个 Object Pascal 特定方法可用于跟踪特定的类，而不是整个内存管理。实际上，对象的内存分配是通过调用 NewInstance 虚拟类方法进行的，而清除是通过 FreeInstance 虚拟方法进行的。您可以在特定的类中重写这些虚拟方法，以自定义特定的内存分配策略。

优点是您可以在不考虑构造函数（可以有多个）和析构函数的情况下执行此操作，从而清楚地将内存跟踪代码与标准对象初始化和完成代码分开。

虽然这是一个极端的极端情况（可能仅对某些大型内存结构值得这样做），但是您可以覆盖这些方法以计算创建和销毁的给定类的对象数，计算活动实例的数量，并在最后检查计数是否达到预期的零。

13.6 编写健壮的应用程序

在本章以及本节的前几章中，我已经介绍了很多技术，这些技术专注于编写健壮的应用程序以及正确管理内存分配和释放。

在本章的最后一部分集中于内存管理，我决定列出一些更高级的主题，以扩大以前的讨论范围。即使已经涵盖了使用 try-finally 块和调用析构函数，但此处突出显示的方案稍微复杂一些，并且涉及一起使用多种语言功能。

这并不是真正的高级部分，而是所有 Object Pascal 开发人员都应该真正掌握的东西，以便能够编写健壮的应用程序。指针和对象引用的最后一个子部分在范围上肯定是更高级的，因为它深入研究了对象和类引用的内部内存结构。

13.6.1 构造函数，析构函数和异常

构造函数和析构函数通常可能是应用程序中问题的根源。

虚拟构造函数必须始终首先调用其基类构造函数。销毁者通常应该将其称为最后继承。

❖ 为了遵循良好的编码习惯，通常应该在 Object Pascal 代码的每个构造函数中添加基类构造函数调用，即使这不是强制性的，并且额外的调用也可能是无用的（例如调用 TObject.Create 时）。

在本节中，我要特别关注在经典情况下，当构造函数失败时会发生什么：

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

如果创建了对象并将其分配给 MyObj 变量，则 finally 块将销毁它。但是，如果 Create 调用引发异常，则不会进入 try-finally 块（这是正确的！）。当构造函数引发异常时，将在可能是部分初始化的对象上自动执行相应的析构函数代码。例如，如果构造函数创建了两个子对象，则需要调用匹配的析构函数来清除那些子对象。但是，如果您在析构函数中假定对象已完全初始化，则可能导致潜在的麻烦。

从理论上很难理解这一点，因此让我们看一下代码中的实际演示。

SafeCode 应用程序项目包含一个带有构造函数的类和一个通常将是正确的析构函数的类……除非构造函数本身失败：

```
type
  TUnsafeDesctructor = class
  private
    FList: 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');
  FList := TList.Create;
end;

destructor TUnsafeDesctructor.Destroy;
begin
  FList.Clear;
  FList.Free;
  inherited;
end;
```

问题在于，在完全创建对象的情况下，析构函数可以正常工作，但是如果在 FList 字段仍设置为 nil 的情况下执行析构函数，则 Clear 调用将引发“访问冲突”异常。

编写相同代码的安全方法如下：

```
destructor TUnsafeDesctructor.Destroy;
begin
  if assigned (FList) then
    FList.Clear;
  FList.Free;
  inherited;
end;
```

同样，故事的寓意再也不应该在析构函数中理所当然地认为相应的构造函数已经完全初始化了对象。您可以针对任何其他方法进行此假设，但不能针对析构函数进行此假设。

13.6.2 嵌套的 Finally 块

最后，块可能是使程序安全的最重要和最常见的技术。我认为这不是一个高级话题，但是您是否最终在所有地方都使用？您是否在边界情况（例如嵌套操作）中正确使用了它，或者在单个 finally 块中组合了多个终结语句？这不是完美的代码示例：

```
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;
```

这是同一代码的安全版本（再次从 SafeCode 应用程序项目中提取）：

```
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;
```

13.6.3 Dynamic（动态）类检查

通常，类型之间，尤其是类类型之间的动态转换操作是造成陷阱的另一个可

能原因。特别是如果您不使用 is 和 as 运算符，而只是进行强制转换。实际上，每个直接类型转换都是潜在的错误源（除非它遵循 is check）。

从对象到指针，从类引用到类引用，从对象到接口，从字符串到字符串的类型转换都是非常危险的，但是在某些特殊情况下很难避免。例如，您可能想将对象引用保存在组件的 Tag 属性中。另一种情况是，将对象保存在指针列表中时，这是老式的 TList（而不是下一章介绍的类型安全的 generic（泛）列表）。这是一个非常愚蠢的示例：

```
procedure TForm1.BtnCastClick(Sender: TObject);
var
  List: TList;
begin
  List := TList.Create;
  try
    List.Add(Pointer(Sender));
    List.Add(Pointer (23422));
    // 直接转换
    TButton(List[0]).Caption := 'ouch';
    TButton(List[1]).Caption := 'ouch';
  finally
    List.Free;
  end;
end;
```

运行此代码通常会导致访问冲突。

- 我之所以这么写是因为，当您随机访问内存时，您永远不会知道实际的效果。有时程序只是覆盖内存而不会引起立即的错误...但是您稍后将很难弄清为什么其他一些数据被破坏了。

您应尽可能避免类似的情况，但是如果您碰巧没有其他选择，可以如何解决此代码？自然的方法是使用 as 安全类型转换或 is 类型检查，如以下代码片段所示：

```
// as cast（转换）
(TObject(List[0]) as TButton).Caption := 'ouch';
(TObject(List[1]) as TButton).Caption := 'ouch';
// is cast（转换）
if TObject(List[0]) is TButton then
  TButton(List[0]).Caption := 'ouch';
if TObject(List[1]) is TButton then
  TButton(List[1]).Caption := 'ouch';
```

但是，这不是解决方案，您将继续遇到访问冲突。问题在于，无论是还是最终调用 TObject.InheritsFrom，这都是对数字执行的困难操作！

解决方案？真正的解决方案是首先避免使用 TObjectList 或其他安全技术（同样，请参阅下一章了解 generic container（通用容器）类），避免出现类似情况（说实话，这种类型的代码几乎没有意义）。如果您确实喜欢低级黑客，并且喜欢玩指针，则可以尝试弄清楚给定的“数字值”是否确实是对对象的引用。但是，这并不是一件小事。它有一个有趣的方面，我以此为借口来向您解释对象的内部结构以及类引用的内部结构。

13.6.4 该指针是对象引用吗？

本节说明了对象和类引用的内部结构，并且超出了本书大部分内容的讨论范围。尽管如此，它仍可以为更多的专家读者提供一些有趣的见解，因此我决定保

留我过去编写的该材料，以作为有关内存管理的高级论文。还请注意，就内存检查而言，以下特定实现实际上是 Windows 特定的。

有时您会遇到指针（指针只是一个数字值，它表示某些数据的物理内存位置）。这些指针实际上可能是对对象的引用，并且您通常会知道它们的使用时间，并直接使用它们。但是，每次执行低级转换时，您实际上就要搞砸整个程序。有一些技术可以使这种指针管理更加安全，即使不是 100%保证它们也可以。

在使用指针之前，您可能要考虑的出发点是它实际上是否是合法指针。Assigned 函数仅检查指针是否为 nil，在这种情况下无济于事。但是，Object Pascal RTL 的鲜为人知的 FindHInstance 函数（在 System 单元中，在 Windows 平台上可用）返回包括作为参数传递的对象的堆块的基地址；如果指针指向无效页，则返回零（防止很少，但是很难跟踪内存页面错误）。如果您几乎随机抽取一个数字，则可能不会引用有效的内存页面。

这是一个很好的起点，但是我们可以做得更好，因为如果值是字符串引用或任何其他有效指针而不是对象引用，这将无济于事。现在，您如何知道指针是否实际上是对对象的引用？我提出了以下实证检验。对象的前 4 个字节是指向其类的指针。如果考虑类引用的内部数据结构，则该类引用在其 vmtSelfPtr 位置中具有指向自身的指针。这在图 13.7 的图像中进行了粗略描绘。

图 13.7：对象和类引用的内部结构的近似表示。

换句话说，通过从类引用指针中解引用内存位置 vmtSelfPtr 字节处的值（这是一个负偏移量，在内存中较低），您应该再次获得相同的类引用指针。此外，在类引用的内部数据结构中，您可以读取实例大小信息（在 vmtInstanceSize 位置），并查看其中是否存在合理的数字。这是实际的代码：

```
function IsPointerToObject (Address: Pointer): Boolean;
var
  ClassPointer, VmtPointer: PChar;
  Instsize: Integer;
begin
  Result := False;
  if (FindHInstance (Address) > 0) then
  begin
    VmtPointer := PChar(Address^);
    ClassPointer := VmtPointer + vmtSelfPtr;
    if Assigned (VmtPointer) and (FindHInstance (VmtPointer) > 0) then
    begin
      Instsize := (PInteger(Vmtpointer + vmtInstanceSize))^;
      // check self pointer and "reasonable" instance size
      //检查自身指针和“合理的”实例大小
      if (Pointer(Pointer(ClassPointer)^) = Pointer(VmtPointer)) and
        (Instsize > 0) and (Instsize < 10000) then
        Result := True;
    end;
  end;
end;
```

有了此函数之后，在之前的 SafeCode 应用程序项目中，我们可以在进行安全的强制类型转换之前添加一个指向对象的检查：

```
if IsPointerToObject (List[0]) then
   (TObject(list[0]) as TButton).Caption := 'ouch';
if IsPointerToObject (List[1]) then
   (TObject(list[1]) as TButton).Caption := 'ouch';
```

同样的想法也可以直接应用于类引用，也可以在类引用之间实现安全铸造。同样，最好首先通过编写更安全，更干净的代码来避免类似的问题，但是如果您无法避免此函数，该函数可能会派上用场。无论如何，本节应该已经解释了这些系统数据结构的一些内部结构。

第三部分 Advanced features（高级功能）

现在，我们已经深入研究了语言基础和面向对象的编程范例，是时候发现 Object Pascal 语言的一些最新和更高级的功能了。Generics（泛型），anonymous methods（匿名方法）和 reflection open（反射打开）了使用新范式开发代码的新范式，从而以重要方式扩展了面向对象的编程。

实际上，其中一些更高级的语言功能使开发人员可以采用新的代码编写方式，提供更多的类型和代码抽象，并允许使用反射功能来发挥最大潜力而采用更动态的编码方法。

本节的最后一部分将通过概述核心运行时库元素来扩展这些语言功能，这些元素是 Object Pascal 开发模型的核心，因此使语言和库之间的区别变得非常模糊。例如，我们将检查 TObject 类，正如我们前面所看到的那样，它是您编写的所有类的基类：远远超出了仅限于库实现细节的角色。

第三部分的章节：

第 14 章：Generics（泛型）。
第 15 章：Anonymous Methods（匿名方法）。
第 16 章：Reflection（反射）和 Attributes（属性）。
第 17 章：TObject 类。
第 18 章：Run Time Library（运行时库）。

第十四章 generics（泛型）

Object Pascal 提供的强类型检查对于提高代码的正确性很有用，这是我在本书中经常强调的主题。但是，强类型检查也可能很麻烦，因为您可能想编写一个对不同数据类型具有类似作用的过程或类。Object Pascal 语言的功能解决了此问题，该功能也可以在类似 C＃和 Java 的通用语言中使用。

泛型或模板类的概念实际上来自 C ++语言。这是我在 1994 年写的关于 C ++的书中写的：

您可以在不指定一个或多个数据成员的类型的情况下声明一个类：此操作可以延迟到实际声明该类的对象之前。同样，在调用函数之前，无需定义一个或多个参数的类型即可定义函数。

文本摘自我在 90 年代初与 Steve Tendon 撰写的《Borland C ++ 4.0 面向对象编程》一书。

本章从基础开始深入探讨该主题，但也涵盖了一些高级用法场景，甚至说明了泛型甚至可以应用于标准的可视化编程。

14.1 Generic Key-Value Pairs（泛型键值对）

作为泛型类的第一个示例，我实现了键值对数据结构。下面的第一个代码段显示了以传统方式编写的数据结构，并带有一个用于保存值的对象：

```
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;
```

要使用此类，您可以创建一个对象，设置其键和值，然后使用它，如下面的 KeyValueClassic 应用程序项目主要形式的各种方法的代码片段所示：

```
// 创建表单
Kv := TKeyValue.Create;
// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender;
// Button2Click
Kv.Value := self; // the form
// Button3Click
ShowMessage('[' + Kv.Key +',' + Kv.Value.ClassName + ']');
```

如果您需要一个类似的类，并持有一个 Integer 而不是一个对象，该怎么办？好吧，您可以进行非常不自然（且危险）的类型转换，也可以创建一个新的单独的类来保存具有数字值的字符串键。尽管复制和粘贴原始类听起来可能是一个解决方案，但是您最终会获得两份非常相似的代码副本，这违反了良好的编程原则……您将必须更新新功能或更正 两次或三或二十次相同的错误。

泛型使对值使用更广泛的定义成为可能，从而编写了单个泛型类。实例化键值泛型类后，它便成为与给定数据类型绑定的特定类。因此，您仍然最终将两个，

三个或二十个类编译到您的应用程序中，但是您为所有这些类定义了一个源代码，仍然可以对正确的字符串类型进行检查，并且没有运行时开销。但是我要超越自我：让我们从定义泛型类的语法开始：

```
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;
```

在此类定义中，有一个未指定类型，由占位符 T 表示，放在尖括号中。常规上经常使用符号 T，但是就编译器而言，您可以随意使用任何符号。当泛型类仅使用一种参数类型时，使用 T 通常会使代码更具可读性。如果类需要多种参数类型，通常根据它们的实际作用来命名它们，而不是像 C ++在早期那样使用字母（T，U，V）序列。

❖ 自 1990 年代 C ++语言引入模板以来，“T”一直是泛类型的标准名称或占位符。根据作者的不同，“T”代表“类型”或“模板类型”。

通用 TKeyValue <T>类使用未指定的类型作为其两个字段之一的类型，即属性值和 setter 方法参数。方法是照常定义的，但是请注意，无论它们与泛型类型有什么关系，它们的定义都包含类的完整名称，包括泛型：

```
procedure TKeyValue<T>.SetKey(const Value: string);
begin
  FKey := Value;
end;

procedure TKeyValue<T>.SetValue(const Value: T);
begin
  FValue := Value;
end;
```

要使用该类，您必须完全限定它，并提供泛型类型的实际类型。例如，您现在可以通过编写以下内容将键值对象托管按钮声明为值：

```
var
  Kv: TKeyValue<TButton>;
```

创建实例时也需要全名，因为这是实际的类型名称（而泛型的，未实例化的类型名称就像类型构造机制一样）。

使用键值对值的特定类型使代码更加健壮，因为您现在只能将 TButton（或派生）对象添加到键值对中，并且可以使用提取的对象的各种方法。这些是 KeyValueGeneric 应用程序项目的主要形式的一些片段：

```
// FormCreate
Kv := TKeyValue<TButton>.Create;
// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender as TButton;
// Button2Click
Kv.Value := Sender as TButton; // was "self"
// Button3Click
ShowMessage ('[' + Kv.Key + ',' + Kv.Value.Name + ']');
```

在先前版本的代码中分配泛型对象时，我们可以添加按钮或表单，现在我们只能使用按钮（由编译器强制执行的规则）。同样，除了在输出中使用泛型的 kv.Value.ClassName 之外，我们还可以使用组件 Name 或 TButton 类的任何其他属性。

当然，我们也可以通过声明具有对象类型的键值对来模仿原始程序，例如：

```
var
  Kvo: TKeyValue<TObject>;
```

在此版本的泛型键值对类中，我们可以将任何对象添加为值。

但是，除非将它们转换为更具体的类型，否则我们将无法对提取的对象做很多事情。为了找到一个良好的平衡，您可能需要在特定按钮和任何对象之间进行操作，要求该值成为一个组件：

```
var
  Kvc: TKeyValue<TComponent>;
```

您可以在同一 KeyValueGeneric 应用程序项目中看到相应的代码段。 最后，我们还可以创建一个泛型键值对类的实例，该实例不存储对象值，而是存储普通整数：

```
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;
```

14.1.1 内联变量和泛型类型推断

在声明泛型类型的变量时，声明可能会很长。创建该类型的对象时，必须重复相同的声明。

也就是说，除非您利用内联变量声明及其推断变量类型的能力。上面的最后一个代码片段可以写成：

```
begin
  var Kvi := TKeyValue<Integer>.Create;
  try
    ...
```

在此代码中，您不必重复完整的泛型类型声明两次。这在使用容器时特别方便，我们将在后面看到。

14.1.2 泛型的类型规则

声明泛型类型的实例时，此类型将获得特定版本，该版本由编译器在所有后续操作中强制执行。因此，如果您有一个泛型类，例如：

```
type
  TSimpleGeneric<T> = class
    Value: T;
  end;
```

当您声明具有给定类型的特定对象时，不能将其他类型分配给“Value”字段。给定以下两个对象，以下某些分配（TypeCompRules 应用程序项目的一部分）不正确：

```
var
  Sg1: TSimpleGeneric<string>;
  Sg2: TSimpleGeneric<Integer>;
begin
  Sg1 := TSimpleGeneric<string>.Create;
  Sg2 := TSimpleGeneric<Integer>.Create;
  Sg1.Value := 'foo';
  Sg1.Value := 10; // Error 错误
  // E2010 Incompatible types: 'string' and 'Integer'
  // E2010 不兼容的类型：“字符串”和“整数”
  Sg2.Value := 'foo'; // Error 错误
  // E2010 Incompatible types: 'Integer' and 'string'
  // E2010 不兼容的类型：“整数”和“字符串”
  Sg2.Value := 10;
```

在泛型声明中定义了特定类型后，编译器将强制执行该类型，这是您期望使用的强类型语言（如 Object Pascal）所期望的。

还可以对整个泛型对象进行类型检查。在为对象指定泛型参数时，不能基于不同且不兼容的类型实例为其分配相似的泛型类型。如果这看起来令人困惑，则应使用一个示例来阐明：

```
  Sg1 := TSimpleGeneric<Integer>.Create; // Error 错误
  // E2010 Incompatible types: （E2010 不兼容的类型：）
  // 'TSimpleGeneric<System.string>' and 'TSimpleGeneric<System.Integer>'
```

正如我们将在“泛类型兼容性规则”部分中看到的，在这种特殊情况下，类型兼容性规则是按结构而不是按类型名称。一旦声明了泛型类型，就不能为泛型类型分配其他不兼容的类型。

14.2 Object Pascal 中的泛型

在前面的示例中，我们看到了如何在 Object Pascal 中定义和使用泛型类。在研究技术细节之前，我决定先通过示例介绍此功能，这些技术既复杂又非常重要。从语言角度介绍泛型之后，我们将返回更多示例，包括泛型容器类的使用和定义，这是该技术在语言中的主要用途之一。

我们已经看到，当您定义一个类时，您可以在尖括号中添加一个额外的“参数”，以保留稍后提供的类型的位置：

```
type
  TMyClass<T> = class
    ...
  end;
```

泛型类型可以用作字段的类型（就像我在前面的示例中所做的一样），属性的类型，参数的类型或函数的返回值等等。请注意，在本地字段（或数组）中使用该类型不是强制性的，因为在某些情况下，泛型仅被用作结果，参数或未在类的声明中使用，而是仅在其某些方法的定义中。

这种 Form 的扩展或泛型类型声明不仅可用于类，而且可用于记录（正如我在第 5 章中介绍的那样，还可以具有方法，属性和重载运算符）。泛型类也可以具有多个参数化类型，在以下情况下，您可以为方法指定输入参数和不同类型的返回值：

```
type
  TPWGeneric<TInput,TReturn> = class
  public
    function AnyFunction (Value: TInput): TReturn;
  end;
```

与其他静态语言一样，Object Pascal 中泛型的实现也不基于运行时支持。它由编译器和 Linker（链接器）处理，几乎没有任何运行时机制。与在运行时绑定的虚拟函数调用不同，泛型类方法为您实例化的每种泛型类型生成一次，并在编译时生成！ 我们将看到这种方法可能存在的弊端，但从积极的方面来看，它意味着泛型类与普通类一样有效，或者由于减少了对运行时检查的需求而变得更加高效。不过，在介绍一些内部结构之前，让我先介绍一些非常重要的规则，这些规则会打破传统的 Pascal 语言类型兼容性规则。

14.2.1 泛型类型兼容性规则

在传统的 Pascal 和 Object Pascal 中，核心类型兼容性规则基于类型名称等效性。 换句话说，两个变量仅在它们的类型名称相同时才是类型兼容的，而与它们所引用的实际数据结构无关。

这是与静态数组类型不兼容的经典示例（TypeCompRules 应用程序项目的一部分）：

```
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 错误
  // E2010 Incompatible types: 'TArrayOf10' and 'Array'
  // E2010 不兼容的类型：“ TArrayOf10”和“ Array”
  Array3 := Array4;
  Array4 := Array1; // Error
  // E2010 Incompatible types: 'Array' and 'TArrayOf10'
  // E2010 不兼容的类型：“数组”和“ TArrayOf10”
end;
```

如您在上面的代码中看到的，所有四个数组在结构上都是相同的。但是，编译器将只允许您分配类型兼容的那些，这是因为它们的类型具有相同的显式名称（例如 TArrayOf10），或者因为它们具有与隐式声明的两个数组（或编译器生成的类型名称）相同的隐式名称。单一声明。

此类型兼容性规则具有非常有限的例外，例如与派生类相关的例外。该规则的另一个例外（也是一个重要的例外）是泛型类型的类型兼容性，编译器可能在内部使用它来确定何时从泛型类型及其所有方法中生成新类型。

新规则规定，泛型类型共享相同的泛型类定义和实例类型时，它们是兼容的，而与该定义关联的类型名称无关。换句话说，泛型类型实例的全名是泛型类型和实例类型的组合。

在下面的示例中，这四个变量都是类型兼容的：

```
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;
```

14.2.2 标准类的泛型方法

尽管使用泛型类型定义类可能是最常见的情况，但泛型类型也可以用于非泛型类中。换句话说，常规类可以具有泛型方法。在这种情况下，在创建类的实例时以及在调用方法时，您不会为泛型占位符指定特定类型。这是 GenericMethod 应用程序项目中带有泛型方法的示例类：

```
type
    TGenericFunction = class
    public
      function WithParam <T> (T1: T): string;
    end;
```

❖ 当我第一次编写此代码时，可能会想起 C ++的日子，我将参数写为（t：T）。不用说像 Object Pascal 这样的不区分大小写的语言，这不是一个好主意。编译器实际上会放任不管，但是每次您引用泛型 T 时都会发出错误。

在类似的类方法中您无能为力（至少除非您使用约束，否则将在本章后面介绍），因此我使用特殊的泛型类型函数（稍后再次介绍）和用于转换键入一个字符串，与此处无关。

```
function TGenericFunction.WithParam<T>(T1: T): string;
begin
    Result := GetTypeName (TypeInfo (T));
end;
```

如您所见，此方法甚至不使用作为参数传递的实际值，而仅获取一些类型信息。同样，完全不了解 t1 的类型使得在代码中使用它相当复杂。

您可以按以下方式调用此“全局泛型函数”的各种版本：

```
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;
```

上面的所有调用都是正确的，因为参数类型可以在这些调用中隐含。请注意，显示的是泛型类型（已指定或推断），而不是参数的实际类型，这说明了此输出：

```
string
Integer
string
ShortInt
TButton
TObject
```

如果在未指出尖括号之间的类型的情况下调用该方法，则从参数的类型推断出实际的类型。如果使用类型和参数调用方法，则参数的类型必须与泛型类型声明匹配。因此，以下三行将无法编译：

```
Show (Gf.WithParam<Integer>('foo'));
Show (Gf.WithParam<string> (122));
Show (Gf.WithParam<TButton>(self));
```

14.2.3 泛型类型实例化

注意，这是一个相当高级的部分，重点介绍了泛型的一些内部及其潜在的优化。不错，适合二读，如果您是第一次研究泛型，则不然。

除某些优化外，每次实例化泛型类型时，无论是在方法中还是在类中，编译器都会生成一个新类型。此新类型不会与相同泛型的不同实例（或相同方法的不同版本）共享代码。

让我们看一个示例（它是 GenericCodeGen 应用程序项目的一部分）。

该程序具有一个泛型类，定义为：

```
type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (value: T);
  end;
```

这三种方法的实现方式如下（请注意，One 方法绝对独立于泛型类型）：

```
procedure TSampleClass<T>.One;
begin
  Form30.Show ('OneT');
end;

function TSampleClass<T>.ReadT: T;
begin
  Result := data;
end;

procedure TSampleClass<T>.SetT(value: T);
begin
  data := value;
end;
```

现在，一旦实例（由编译器生成），主程序便主要使用泛型类型来找出其方

法的内存地址。这是代码：

```
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;
```

结果是这样的（实际值会有所不同）：

```
T1.SetT: C3045089
T2.SetT: 51EC8B55
T1.One: 4657F0BA
T2.One: 46581CBA
```

正如我预期的那样，尽管 SetT 方法在编译器为所使用的每种数据类型生成的内存中获得的版本不同，但即使 One 方法也可以，尽管它们都是相同的。

此外，如果重新声明相同的泛型类型，则将获得一组新的实现功能。类似地，在不同单元中使用的泛型类型的相同实例会迫使编译器一遍又一遍地生成相同的代码，这可能会导致严重的代码膨胀。因此，如果您有一个具有许多不依赖于该泛型类型的方法的泛型类，建议您使用这些泛型方法定义一个基类非泛型类，并使用该泛型方法定义一个继承的泛型类：这样，基类方法仅编译一次并包含在可执行文件中。

❖ 当前正在进行编译器，Linker（链接器）和低级 RTL 工作，以减少本节概述的方案中由泛型引起的大小增加。请参阅 http://delphisorcery.blogspot.it/2014/10/new-language-feature-in-xe7.html 中的注意事项。

14.2.4 泛型类型函数

到目前为止，我们所看到的泛型类型定义的最大问题是，对泛型类类型的元素几乎无能为力。您可以使用两种技术来克服此限制。首先是利用运行时库的一些特殊功能，这些功能专门支持泛型类型。第二个（功能更强大）是定义对您可以使用的类型具有约束的泛型类。

我将在本节中重点介绍第一种技术，并在下一节中重点讨论约束。

如前所述，有一些 RTL 函数可用于泛型类型定义的参数类型（T）：

- Default(T)实际上是与泛型一起引入的新函数，该泛型返回当前类型的空或“零值”或空值；可以是零，空字符串，nil 等。零初始化的存储器具有相同类型的全局变量的相同值（与局部变量不同，实际上，全局变量由编译器初始化为“零”）。
- TypeInfo(T)返回指向泛型类型当前版本的运行时信息的指针；您将在第 16 章中找到有关类型信息的更多信息。

- SizeOf(T)返回该类型的内存大小（以字节为单位）（如果是字符串或对象等引用类型，则为引用的大小，对于 32 位编译器为 4 字节，对于 64 位编译器为 8 字节）。
- IsManagedType(T)指示是否在内存中管理类型，如字符串和动态数组一样。
- HasWeakRef(T)与启用 ARC 的编译器绑定，并指示 target 类型是否具有弱引用，需要特定的内存管理支持。
- GetTypeKind(T)是从类型信息访问类型种类的快捷方式；这是一个比 TypeInfo 返回的级别定义略高级别的类型定义。

❖ 所有这些方法都返回编译器求值的常量，而不是在运行时调用实际函数。实际上，这些操作的重要性并不在于它们的运算速度非常快，而是使编译器和 Linker（链接器）可以优化生成的代码，从而删除未使用的分支。如果您有基于这些函数之一的返回值的 case 或 if 语句，则编译器可以确定对于给定类型，仅将执行其中一个分支，从而删除无用的代码。当针对不同类型编译相同的泛型方法时，它可能最终会使用不同的分支，但是编译器可以再次弄清楚并优化方法的大小。

GenericTypeFunc 应用程序项目具有一个泛型类，该泛型类显示了正在使用的三个泛型类型函数：

```
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 System.TypInfo 单元的 GetTypeName 函数，而不是直接访问数据结构，因为它从保存类型名称的编码字符串值执行正确的转换。

根据上面的声明，您可以编译以下测试代码，该代码在三个不同的泛型类型实例上重复执行三遍。我省略了重复的代码，但是显示了用于访问数据字段的语句，因为它们根据实际类型而改变：

```
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));
```

运行以下代码（来自 GenericTypeFunc 应用程序项目）将产生以下输出：

```
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，除了泛型类的上下文之外，您还可以在特定类型上使用泛型类型函数。例如，您可以编写：

```
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));
```

这是简单的输出：

```
Default Integer: 0
Default String:
TypeInfo String: string
```

14.2.5 泛型类的类构造函数

当您为泛型类定义类构造函数时，会出现一个非常有趣的情况。

实际上，编译器会生成一个这样的构造函数，并为每个泛型类实例（即，使用发现模板定义的每种实际类型）调用。这非常有趣，因为对于要在没有类构造函数的情况下在程序中创建的泛型类的每个实际实例执行初始化代码将非常复杂。

例如，考虑具有一些类数据的泛型类。 您将获得每个泛型类实例的此类数据实例。如果需要为此类数据分配初始值，则不能使用单元初始化代码，因为在定义泛型类的单元中，您不知道需要哪些实际类。

以下是泛型类的基本示例，该类具有用于初始化 DataSize 类字段的类构造函

数，该类构造函数取自 GenericClassCtor 应用程序项目：

```
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;
```

这是泛型类构造函数的代码，它使用内部字符串列表（有关实现的详细信息，请参见完整的源代码）来跟踪实际调用了哪些类构造函数：

```
class constructor TGenericWithClassCtor<T>.Create;
begin
  DataSize := SizeOf (T);
  ListSequence.Add(ClassName);
end;
```

该演示程序创建并使用了该泛型类的几个实例，还声明了第三个数据类型，该数据类型由 linker（链接器）删除：

```
var
  GenInt: TGenericWithClassCtor <SmallInt>;
  GenStr: TGenericWithClassCtor <string>;
type
  TGenDouble = TGenericWithClassCtor <Double>;
```

如果您要求程序显示 ListSequence 字符串列表的内容，则只会看到实际上已初始化的类型：

```
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

但是，如果您基于不同单元中的相同数据类型创建泛型实例，则 linker（链接器）可能无法按预期工作，并且您将有多个调用相同的泛型类构造函数（或更准确地说，两个泛型类构造函数用于相同的类型）。

- ❖ 解决类似问题并不容易。为了避免重复初始化，您可能要检查类构造函数是否已经执行。但是，一般而言，此问题是泛型类的更全面限制和链接器无法优化它们的一部分。

我在此示例的辅助单元中添加了一个名为 Useless 的过程，该过程在未注释时将以如下初始化序列突出显示该问题：

```
TGenericWithClassCtor<System.string>
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

14.3 Generic Constraints（泛型约束）

如我们所见，在泛型类型的方法上，泛型类型值几乎无能为力。您可以将其传递（即分配它）并执行我刚刚介绍的泛型类型函数所允许的有限操作。

为了能够执行类的泛型类型的某些实际操作，通常必须在其上施加约束。 例如，如果将泛型类型限制为类，则编译器将允许您在其上调用所有 TObject 方法。您还可以进一步限制类成为给定层次结构的一部分或实现特定的接口，从而可以在泛型类型的实例上调用类或接口方法。

14.3.1 Class Constraints（类约束）

您可以采用的最简单的约束是类约束。要使用它，可以将泛型声明为：

```
type
  TSampleClass <T: class> = class
```

通过指定类约束，表明您只能将对象类型用作泛型类型。带有以下声明（摘自 ClassConstraint 应用程序项目）：

```
type
  TSampleClass <T: class> = class
  private
    FData: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (T1: T);
  end;
```

您可以创建前两个实例，但不能创建第三个实例：

```
sample1: TSampleClass<TButton>;
sample2: TSampleClass<TStrings>;
sample3: TSampleClass<Integer>; // Error 错误
```

最后一条声明引起的编译器错误为：

E2511 Type parameter 'T' must be a class type

（E2511 类型参数“T”必须是类类型）

指示此约束的好处是什么？在泛型类方法中，您现在可以调用任何 TObject 方法，包括虚拟方法！这是 TSampleClass 泛型类的 One 方法：

```
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;
```

- 这里有两个评论。首先是 InstanceSize 返回对象的实际大小，这与我们之前使用的泛型 SizeOf 函数不同，后者返回引用类型的大小。其次，注意使用 TObject 类的 ToString 方法。

您可以使用该程序来查看其实际效果，因为它定义并使用了一些泛型类型的实例，如以下代码片段所示：

```
var
  Sample1: TSampleClass<TButton>;
begin
  Sample1 := TSampleClass<TButton>.Create;
  try
    Sample1.SetT (Sender as TButton);
    Sample1.One;
  finally
    Sample1.Free;
  end;
```

注意，通过使用自定义的 ToString 方法声明一个类，当数据对象为特定类型时，无论提供给泛型类型的实际类型如何，都将调用此版本。换句话说，如果您

有一个 TButton 后代，例如：

```
type
  TMyButton = class (TButton)
  public
    function ToString: string; override;
  end;
```

您可以将此对象作为 TSampleClass <TButton>的值传递，或者定义泛型的特定实例，在两种情况下，调用 One 最终都将执行特定版本的 ToString：

```
var
  Sample1: TSampleClass<TButton>;
  Sample2: TSampleClass<TMyButton>;
  Mb: TMyButton;
begin
  ...
  Sample1.SetT (Mb);
  Sample1.One;
  Sample2.SetT (Mb);
  Sample2.One;
```

与类约束类似，您可以具有记录约束，声明为：

```
type
  TSampleRec <T: record> = class
```

但是，几乎没有不同记录的共同点（没有共同祖先），因此此声明在一定程度上受到限制。

14.3.2 特定类约束

如果您的泛型类需要使用特定的类子集（特定的层次结构），则可能希望诉诸于基于给定基类的约束。

例如，如果您声明：

```
type
  TCompClass <T: TComponent> = class
```

该泛型类的实例只能应用于组件类，即任何 TComponent 后代类。这使您拥有一个非常特定的泛型类型（是的，听起来很奇怪，但这确实是事实），并且编译器将允许您在处理泛型类型时使用 TComponent 类的所有方法。

如果这看起来非常强大，请三思。如果考虑使用继承和兼容类型的规则可以实现的目标，则可以使用传统的面向对象技术解决相同的问题，而不必使用泛型类。我并不是说特定的类约束永远不会有用，但是它肯定不像更高级别的类约束或（基于我的观点很有趣）基于接口的约束那样强大。

14.3.3 接口约束

与其将泛型类限制为给定类，不如将仅实现给定接口的类作为类型参数接受，通常更为灵活。这样就可以在泛型类型的实例上调用接口。在 C# 语言中，对泛型使用接口约束的情况也很常见。让我首先向您展示一个示例（来自 IntfConstraint 应用程序项目）。首先，我们需要声明一个接口：

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
```

```
      procedure SetValue (Value: Integer);
      property Value: Integer read GetValue write SetValue;
    end;
```
接下来，我们可以定义一个实现它的类：
```
  type
    TGetValue = class (TSingletonImplementation, IGetValue)
    private
      FValue: Integer;
    public
      constructor Create (Value: Integer = 0);
      function GetValue: Integer;
      procedure SetValue (Value: Integer);
    end;
```
对于泛型类的定义开始变得有趣起来，该类仅限于实现给定接口的类型：
```
  type
    TInftClass <T: IGetValue> = class
    private
      FVal1, FVal2: T; //  或者 IGetValue
    public
      procedure Set1 (Val: T);
      procedure Set2 (Val: T);
      function GetMin: Integer;
      function GetAverage: Integer;
      procedure IncreaseByTen;
    end;
```
请注意，在此类的泛型方法的代码中，我们可以编写例如：
```
  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;
```
有了所有这些定义，我们现在可以如下使用泛型类：
```
  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;
```
为了显示该泛型类的灵活性，我为接口创建了另一个完全不同的实现：
```
  type
```

```
  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;
end;

procedure TButtonValue.SetValue(Value: Integer);
begin
  Left := Value;
end;
```

类函数（书中未列出）在父控件中的任意位置创建一个按钮，并在以下示例代码中使用：

```
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;
```

14.3.4 接口 References（参考）与泛型接口约束

在上一个示例中，我定义了一个泛型类，该类可与实现给定接口的任何对象一起使用。通过基于接口引用创建标准（非泛型）类，我可能获得类似的效果。实际上，我本可以定义一个类似的类（也是 IntfConstraint 应用程序项目的一部分）：

```
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;
```

这两种方法有什么不同？第一个区别是，在上述类中，您可以将两个不同类

型的对象传递给 setter 方法，前提是它们的类都实现了给定的接口，而在泛型版本中，您只能将给定类型的对象（传递给任何给定的实例）的泛型类）。因此，泛型版本在类型检查方面更加保守和严格。

在我看来，主要区别在于使用基于接口的版本意味着要使用 Object Pascal 的引用计数机制，而使用泛型版本时，该类将处理给定类型的普通对象，并且不涉及引用计数。

此外，泛型版本可能具有多个约束，例如构造函数约束，并允许您使用各种泛型函数（例如，询问泛型类型的实际类型），这是使用接口时无法执行的操作。（实际上，在使用接口时，您无法引用基本的 TObject 方法）。

换句话说，使用具有接口约束的泛型类可以使接口受益匪浅。仍然需要注意的是，在大多数情况下，这两种方法是等效的，而在其他情况下，基于接口的解决方案将更加灵活。

14.3.5 默认构造函数约束

还有另一个可能的泛型类型约束，称为默认构造函数或无参数构造函数。 如果需要调用默认构造函数来创建泛型的新对象（例如，用于填充列表），则可以使用此约束。理论上（根据文档），编译器应只允许您将其用于具有默认构造函数的那些类型。实际上，如果不存在默认构造函数，则编译器会放任其长，并调用 TObject 的默认构造函数。

具有构造函数约束的泛型类可以编写如下（该类由 IntfConstraint 应用程序项目提取）：

```
type
  TConstrClass <T: class, constructor> = class
  private
    FVal: T;
  public
    constructor Create;
    function Get: T;
  end;
```

❖ <u>您还可以指定构造函数约束，而无需类约束，因为前者可能暗示后者。列出它们两者可使代码更具可读性</u>

给定此声明，您可以使用构造函数创建泛型内部对象，而无需事先知道其实际类型，并编写：

```
constructor TConstrClass<T>.Create;
begin
  FVal := T.Create;
end;
```

我们如何使用该泛型类？实际的规则是什么？在下一个示例中，我定义了两个类，一个类具有默认（无参数）构造函数，第二个类具有一个具有一个参数的单个构造函数：

```
type
  TSimpleConst = class
  public
    FValue: Integer;
    constructor Create; // set Value to 10（将值设置为 10）
  end;

  TParamConst = class
```

```
    public
      FValue: Integer;
      constructor Create (I: Integer); // set Value to I（将值设置为 I）
    end;
```

正如我前面提到的，理论上您应该只能使用头等舱，而实际上您可以同时使用这两种舱：

```
  var
    ConstructObj: TConstrClass<TSimpleCost>;
    ParamCostObj: TConstrClass<TParamCost>;
  begin
    ConstructObj := TConstrClass<TSimpleCost>.Create;
    Show ('Value 1: ' + IntToStr (ConstructObj.Get.FValue));

    ParamCostObj := TConstrClass<TParamCost>.Create;
    Show ('Value 2: ' + IntToStr (ParamCostObj.Get.FValue));
```

此代码的输出是：

```
  Value 1: 10
  Value 2: 0
```

实际上，第二个对象从未初始化。如果将应用程序跟踪调试到代码中，则会看到对 TObject.Create 的调用（我认为这是错误的）。请注意，如果您尝试直接调用：

```
    with TParamConst.Create do
```

编译器将（正确）引发错误：

```
  [DCC Error] E2035 Not enough actual parameters
```

（[DCC 错误] E2035 实际参数不足）

❖ 即使对 TParamConst.Create 的直接调用在编译时失败（如此处所述），使用类引用或任何其他间接形式的类似调用也将成功，这很可能解释了构造函数约束的作用。

14.3.6 约束摘要和组合

由于可以对泛型类型施加许多不同的约束，因此让我在此处以代码形式提供简短摘要：

```
  type
    TSampleClass <T: class> = class
    TSampleRec <T: record> = class
    TCompClass <T: TButton> = class
    TInftClass <T: IGetValue> = class
    TConstrClass <T: constructor> = class
```

在查看约束后，您可能不会立即意识到（当然这花了我一些时间来习惯）是可以将它们组合在一起。例如，您可以定义一个仅限于子层次结构并且还需要给定接口的泛型类，例如：

```
  type
    TInftComp <T: TComponent, IGetValue> = class
```

并非所有组合都有意义：例如，您不能同时指定类和记录，而将类约束与特定类约束结合使用将是多余的。最后，请注意，没有什么比方法约束更重要的了，这可以通过单方法接口约束来实现（但是表达起来要复杂得多）。

14.4 预定义的泛型容器

自从 C ++语言中的模板问世以来，模板类最明显的用途之一就是模板容器或列表的定义，直到 C ++语言定义了标准模板库（STL）。

当定义对象列表时，例如 Object Pascal 自己的 TObjectList，您将拥有一个列表，该列表可能包含任何类型的对象。使用继承或组合，您确实可以为特定类型定义自定义容器，但这是一种乏味（且可能容易出错）的方法。

Object Pascal 编译器带有一小类通用容器类，您可以在 Generics.Collections 单元中找到它们。四个核心容器类都以独立的方式实现（这些类之间没有继承），都以类似的方式（使用动态数组）实现，并且都映射到旧 Contnrs（容器）单元的相应的非泛型容器类：

```
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>)
```

考虑到它们的名称，这些类之间的逻辑差异应该非常明显。测试它们的一个好方法是计算出您必须对使用非泛型容器类的现有代码执行多少更改。

❖ 该程序仅使用一些方法，因此对于泛型列表和非泛型列表之间的接口兼容性不是一个很好的测试，但是我决定采用现有程序，而不是构造一个。展示此演示的另一个原因是，您可能还拥有不使用泛型集合类的现有程序，因此会鼓励您利用此语言功能来增强它们。

14.4.1 使用 TList<T>

名为 ListDemoMd2005 的程序具有定义 TDate 类的单元，其主要表单用于引用日期的 TList。首先，我添加了一个指向 Generics.Collections 的 uses 子句，然后将主表单字段的声明更改为：

```
private
  FListDate: TList <TDate>;
```

当然，创建列表的主表单 OnCreate 事件处理程序也需要更新，变成：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FListDate := TList<TDate>.Create;
end;
```

现在，我们可以尝试按原样编译其余代码。该程序有一个"有害"错误，试图将 TButton 对象添加到列表中。用于编译的相应代码现在失败了：

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  //  在列表中添加一个按钮
  FListDate.Add (Sender); // Error:出错
  // E2010 Incompatible types: 'TDate' and 'TObject'
  // E2010 不兼容的类型：“ TDate”和“ TObject”
```

```
end;
```

在类型检查方面，新的日期列表比原始的泛型列表指针更健壮。删除该行后，程序便可以编译并运行。

仍然可以改进。

这是原始代码，用于在 ListBox 控件中显示列表的所有日期：

```
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add ((TObject(FListDate [I]) as TDate).Text);
```

请注意类型转换，因为程序使用的是指针列表（TList），而不是对象列表（TObjectList）。我们可以通过编写以下代码轻松地改进程序：

```
  for I := 0 to FListDate.Count - 1 do
    Listbox1.Items.Add (FListDate [I].Text);
```

此代码段的另一个改进可以来自使用枚举（某些预定义的泛型列表完全支持），而不是使用普通的 for 循环：

```
var
  ADate: TDate;
begin
  for ADate in FListDate do
  begin
    Listbox1.Items.Add (ADate.Text);
  end;
```

最后，可以通过使用拥有 TDate 对象的泛型 TObjectList 来改进程序，但这是下一节的主题。

如前所述，TList <T>泛型类具有高度的兼容性。它具有所有经典方法，例如 Add，Insert，Remove 和 IndexOf。容量和计数属性也在那里。奇怪的是，Items 变成 Item，但是作为默认属性（使用不带属性名称的方括号访问），无论如何很少会显式引用它。

14.4.2 对 TList <T>排序

有趣的是了解 TList <T>排序是如何工作的（我的目标是为 ListDemoMd2005 应用程序项目添加排序支持）。Sort 方法定义为：

```
procedure Sort; overload;
procedure Sort(const AComparer: IComparer<T>); overload;
```

在 Generics.Defaults 单元中声明 IComparer <T>接口的位置。如果调用第一个版本的程序，它将使用默认比较器，该默认比较器由 TList <T>的默认构造函数初始化。在我们的情况下，这将是无用的。

相反，我们需要做的是定义 IComparer <T>接口的正确实现。为了实现类型兼容性，我们需要定义一个适用于特定 TDate 类的实现。

有多种方法可以完成此操作，包括使用匿名方法（即使在下一章中介绍了该主题，该知识也会在下一部分中介绍）。一种有趣的技术（也是因为它使我有机会展示泛型的几种使用模式）是利用属于 Generics.Defaults 单元（称为 TComparer）的结构类的优势。

❖ 我称此类为结构化类是因为它有助于定义代码的结构及其体系结构，但在实际实现方面并没有增加太多。不过，可能有一个更好的名称来引用此类。

该类定义为接口的抽象和泛型实现，如下所示：

```
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;
```

我们要做的是为特定的数据类型实例化此泛型类（在示例中为 TDate），并继承一个具体的类，该类为特定的类型实现 Compare 方法。这两个操作可以使用编码 idiom（惯用法）一次完成，而编码惯用法可能需要一段时间才能消化：

```
type
  TDateComparer = class (TComparer<TDate>)
    function Compare(const Left, Right: TDate): Integer; override;
  end;
```

如果您认为这段代码看起来很不寻常，那么您并不孤单。新类继承自泛型类的特定实例，您可以在两个单独的步骤中表示为：

```
type
  TAnyDateComparer = TComparer<TDate>;
  TMyDateComparer = class (TAnyDateComparer)
    function Compare(const Left, Right: TDate): Integer; override;
  end;
```

- ❖ 具有两个单独的声明可能有助于减少在同一单元中重用基础 TAnyDateComparer 类型的情况下生成的代码。

您可以在源代码中找到 Compare 函数的实际实现，因为这不是我在这里要强调的重点。但是请记住，即使对列表进行排序，它的 IndexOf 方法也不会利用它（与 TStringList 类不同）。

14.4.3 用匿名方法排序

上一节中介绍的排序代码看起来非常复杂，实际上确实如此。直接将排序功能传递给 Sort 方法将更加容易和清洁。在过去，这通常是通过传递函数指针来实现的。在 Object Pascal 中，这可以通过传递匿名方法（一种方法指针，具有一些额外的功能，在下一章中进行详细介绍）来完成。

- ❖ 我建议即使您对匿名方法不太了解，也请阅读本节，然后在下一章中再次阅读。

实际上，可以通过调用 TComparer <T>的 Construct 方法，并将匿名方法作为定义为参数的参数来使用 TList <T>类的 Sort 方法的 IComparer <T>参数。

```
type
  TComparison<T> = reference to function(const Left, Right: T): Integer;
```

实际上，您可以编写类型兼容的函数并将其作为参数传递：

```
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
    FListDate.Sort (TComparer<TDate>.Construct (DoCompare));
  end;
```

上面的 DoCompare 方法即使有名称，也像匿名方法一样工作。我们将在后面的代码片段中看到这不是必需的。请耐心等待下一章，以获取有关此 Object Pascal 语言构造的更多信息。还要注意，使用 TDate 记录，我可以定义小于和大于运算符，从而使此代码更简单，但是即使使用类，我也可以将比较代码放在该类的方法中。

如果这看起来很传统，请考虑避免使用单独的函数声明，并将其（其源代码）作为参数传递给 Construct 方法，如下所示：

```
  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;
```

这个例子应该激发您学习更多有关匿名方法的兴趣！可以肯定的是，尽管对于许多拥有派生类的 Object Pascal 开发人员来说，看起来更清晰，更容易理解，但最后一个版本的编写要比上一节中介绍的原始比较容易得多（继承的版本更好地分离了逻辑，从而有可能代码重用更加容易，但是很多时候您还是不会使用它）。

14.4.4 对象容器

除了本节开头介绍的泛型类之外，还有四个继承的泛型类，它们是从 Generics.Collections 单元中定义的基类派生而来的，模仿了 Contnrs 单元的现有类：

```
  type
    TObjectList<T: class> = class(TList<T>)
    TObjectQueue<T: class> = class(TQueue<T>)
    TObjectStack<T: class> = class(TStack<T>)
```

与基类相比，有两个主要区别。一种是这些泛型类型只能用于对象。第二个是他们定义了一个自定义的 Notification 方法，在从列表中删除对象的情况下（除了可以选择调用 OnNotify 事件处理程序之外），还将释放对象。

换句话说，设置 OwnsObjects 属性时，TObjectList <T>类的行为类似于其非泛型对应类。如果您想知道为什么不再选择此选项，请考虑现在可以将 TList <T>直接用于对象类型，这与它的非泛型对应对象不同。

同样，还有第四类，称为 TObjectDictionary <TKey, TValue>，它以不同的方式定义，因为它可以拥有键对象，值对象或它们两者。有关更多详细信息，请参见 TDictionaryOwnerships 集和类构造函数。

14.4.5 使用泛型 Dictionary（字典）

在所有预定义的泛型容器类中，可能值得更详细研究的一个是泛型字典 TObjectDictionary <TKey, Tvalue>。

❖ 在这种情况下，字典意味着元素的集合，每个元素都有一个引用它的（唯一）键值。（它也被称为关联数组。）在经典词典中，单词是其定义的键，但是在编程术语中，键不必是字符串（即使这种情况很常见）。

其他类同样重要，但是它们似乎更易于使用和理解。作为使用字典的示例，我编写了一个应用程序，该应用程序从数据库表中获取数据，为每个记录创建一个对象，并使用带有客户 ID 和描述作为键的复合索引。进行这种分离的原因是，可以轻松地使用类似的体系结构来创建代理，其中的密钥代替了从数据库加载的实际对象的轻量级版本。

这是 CustomerDictionary 应用程序项目使用的两个类，用于键和实际值。第一个具有相应数据库表的两个相关字段，而第二个具有完整的数据结构（我省略了 private（私有）字段，getter 方法和 setter 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;
```

尽管第一个类非常简单（每个对象在创建时都会初始化），但 TCustomer 类使用了惰性初始化（或代理）模型，并保留了对所有对象共享的源数据库的引用（class var）。创建对象时，将为该对象分配对相应 TCustomerKey 的引用，而类数据字段则引用源数据集。在每种 getter 方法中，类都会在返回数据之前检查对象是否确实已初始化，如以下情况：

```
function TCustomer.GetCompany: string;
begin
  EnforceInit;
  Result := FCompany;
end;
```

EnforceInit 方法检查一个本地标志，最终调用 Init 将数据从数据库加载到内存中对象：

```
procedure TCustomer.EnforceInit;
begin
  if not FInitDone then
  Init;
end;

procedure TCustomer.Init;
begin
  RefDataSet.Locate('CustNo', CustKey.CustNo, []);
  //  也可以通过 RTTI 加载每个已发布的字段
  FCustNo := RefDataSet.FieldByName ('CustNo').AsFloat;
  FCompany := RefDataSet.FieldByName ('Company').AsString;
  FCountry := RefDataSet.FieldByName ('Country').AsString;
  ...
  FInitDone := True;
end;
```

给定这两个类，我为应用程序添加了专用字典。

此自定义词典类继承自具有适当类型实例化的泛型类，并向其添加了特定方法：

```
type
  TCustomerDictionary = class (TObjectDictionary <TCustomerKey, TCustomer>)
  public
    procedure LoadFromDataSet (Dataset: TDataSet);
  end;
```

加载方法将填充字典，仅将关键对象的数据复制到内存中：

```
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;
```

该演示程序具有一个主窗体和一个包含 ClientDataSet 组件的数据模块。主窗体具有一个 ListView 控件，当用户按下 only button 时，该控件将被填充。

❖ 您可能希望用一个真实的数据集替换 ClientDataSet 组件，从而在实用性方面大大扩展了该示例，因为您可以对键运行查询，而对每个 TCustomer 对象的实际数据运行一个单独的查询。我有类似的代码，但是在此处添加示例可能会使我们分心，而该示例的目标是尝试使用泛型词典类。

在将数据加载到字典中之后，BtnPopulateClick 方法在字典的键上使用枚举数：

```
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;
```

这将用键对象中的可用数据填充 ListView 控件的前两列。但是，无论何时用户选择 ListView 控件的一项，程序都会填充第三列：

```
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;
```

上面的方法将对象映射到给定的键，并使用其数据。

在后台，第一次使用特定对象时，属性访问方法将触发 TCustomer 对象的整个数据的加载。

14.4.6 字典与字符串 Lists（列表）

多年来，包括我在内的许多 Object Pascal 开发人员都过度使用了 TStringList 类。您不仅可以将其用于简单的字符串列表和名称/值对列表，而且还可以使用它来使列表对象与字符串相关联并搜索这些对象。自从引入泛型以来，最好使用它们而不是使用一种喜欢的工具作为瑞士军刀类方法。

特定且集中的容器类是更好的选择。例如，具有两个字符串键和一个对象值的泛型 TDictionary 通常在两个方面优于字符串列表：更干净和更安全的代码，因为涉及的类型转换较少，而执行更快，因为字典使用 hash（哈希）表。

为了演示这些差异，我编写了一个相当简单的应用程序，名为

StringListVsDictionary。它的 main form 存储两个相同的列表，声明为：

```
private
  FList: TStringList;
  FDict: TDictionary<string,TMyObject>;
```

这两个列表填充有随机但相同的条目，并带有重复此代码的循环：

```
FList.AddObject (AName, AnObject);
FDict.Add (AName, AnObject);
```

两个按钮检索列表中的每个元素，并按名称对每个元素进行搜索。两种方法都扫描字符串列表中的值，但是第一种方法在字符串列表中找到对象，而第二种方法使用字典。请注意，在第一种情况下，您需要使用 as 强制转换来获取给定类型，而字典已经与该类绑定了。这是这两种方法的主循环：

```
TheTotal := 0;
for I := 0 to sList.Count -1 do
begin
  aName := FList[I];
  // 现在搜索
  anIndex := FList.IndexOf (AName);
  // 获取对象
  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];
  // 获取对象
  AnObject := FDict.Items [AName];
  Inc (TheTotal, AnObject.Value);
end;
```

我不想按顺序访问字符串，但要弄清楚与字典的哈希键相比，在排序后的字符串列表（进行二进制搜索）中进行搜索需要多少时间。毫不奇怪，字典更快，这是测试的毫秒数：

```
Total: 99493811
StringList: 2839
Total: 99493811
Dictionary: 686
```

在初始值相同的情况下，结果是相同的，但是时间却大不相同，一百万个条目的字典大约花费时间的四分之一。

14.5 泛型接口

在“对 TList <T>排序”部分中，您可能已经注意到预定义接口的使用很奇怪，该接口具有泛型声明。值得详细研究此技术，因为它带来了很多机会。

需要注意的第一个技术元素是，定义泛型接口是完全合法的，就像我在 GenericInterface 应用程序项目中所做的那样：

```
type
  IGetValue<T> = interface
    function GetValue: T;
    procedure SetValue (Value: T);
  end;
```

❖ <u>这是 IntfContraints 应用程序项目的 IGetValue 接口的泛型版本，在本章前面</u>

的“接口约束”部分中进行了介绍。在那种情况下，该接口具有 Integer 值，现在它具有泛型值。

请注意，与标准接口不同，如果是泛型接口，则无需指定 GUID 用作接口 ID（或 IID）。即使隐式声明，编译器也会为您为泛型接口的每个实例生成一个 IID。实际上，您不必创建泛型接口的特定实例即可实现它，而可以定义一个实现泛型接口的泛型类：

```
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;
```

虽然构造函数分配了对象的初始值，但析构函数的唯一目的是记录对象已被销毁。我们可以通过编写以下内容来创建此泛型类的实例（从而在幕后生成接口类型的特定实例）：

```
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;
```

就像我们过去在 IntfConstraint 应用程序项目中看到的那样，另一种方法是使用对应类型的接口变量，使特定的接口类型定义是显式的（而不像前面的代码片段中那样隐式）：

```
procedure TFormGenericInterface.BtnIValueClick(Sender: TObject);
var
  AVal: IGetValue<string>;
begin
  AVal := TGetValue<string>.Create (Caption);
  Show ('IGetValue value: ' + AVal.GetValue);
  // 自动释放，因为它是参考计数
end;
```

当然，我们还可以定义一个实现泛型接口的特定类，如以下场景所示（来自 GenericInterface 应用程序项目）：

```
type
  TButtonValue = class (TButton, IGetValue<Integer>)
  public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
      Parent: TWinControl): TButtonValue;
  end;
```

注意，虽然 TGetValue <T>泛型类实现了泛型 IGetValue <T>接口，但是 TButtonValue 特定类实现了 IGetValue <Integer>特定接口。具体来说，如前面的示

例所示，接口被重新映射到控件的 Left 属性：

```
function TButtonValue.GetValue: Integer;
begin
  Result := Left;
end;
```

在上面的类中，MakeTButtonValue 类函数是一种易于使用的方法，用于创建该类的对象。主窗体的第三个按钮使用此方法，如下所示：

```
procedure TFormGenericInterface.BtnValueButtonClick(Sender: TObject);
var
  IVal: IGetValue<Integer>;
begin
  IVal := TButtonValue.MakeTButtonValue (self, ScrollBox1);
  Show ('Button value: ' + IntToStr (IVal.GetValue));
end;
```

尽管它与泛型类完全无关，但这是 MakeTButtonValue 类函数的实现：

```
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.Text := 'Btnv';
end;
```

14.5.1 预定义的泛型接口

现在，我们已经探索了如何定义泛型接口，以及如何将它们与泛型和特定类结合使用，我们可以再来看看 Generics.Defaults 单元。本单元定义了两个泛型 comparison（比较）接口：

- IComparer<T> 有一个比较方法
- IEqualityComparer<T> 具有 Equals 和 GetHashCode 方法

这些类由下面列出的一些泛型和特定类实现（没有实现详细信息）：

```
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>)
```

在上面的清单中，您可以看到接口的泛型实现所使用的基类是经典的引用计数的 TInterfacedObject 类或新的 TSingletonImplementation 类。这是一个名称奇怪的类，它提供 IInterface 的基本实现，而没有引用计数。

❖ 单例一词通常用于定义一类，您只能创建一个实例，而不能创建一个没有引用计数的实例。我认为这是一个错误的说法。

正如我们在本章前面的“对 TList <T>排序”部分中已经看到的，这些比较类由泛型容器使用。不过，为了使事情变得更复杂，Generics.Default 单元在很大程度上依赖于匿名方法，因此您可能应该仅在阅读下一章之后再进行研究。

14.6 Object Pascal 中的智能指针

当使用泛型时，您可能会得到一个错误的印象，即该语言结构主要用于集合。尽管这是使用泛型类的最简单情况，并且通常是书籍和文档中的第一个示例，但泛型在收集（或容器）类领域之外非常有用。在本章的最后一个示例中，我将向您展示一个非集合的泛型类型，即智能指针的定义。

如果您来自 Object Pascal 的背景，那么您可能没有听说过智能指针，这是 C ++语言的想法。在 C ++中，您可以具有指向对象的指针，必须直接和手动管理这些对象的内存，以及可以自动管理但具有许多其他限制（包括缺乏多态性）的局部对象变量。智能指针的想法是使用本地管理的对象来照顾指向您要使用的真实对象的指针的生命周期。如果这听起来太复杂，我希望 Object Pascal 版本（及其代码）将有助于澄清它。

❖ OOP 语言中的术语“多态性”用于表示您向基类的变量分配派生类的对象并调用基类虚拟方法之一的情况，最终可能会调用以下方法的虚拟方法版本：特定的子类。

14.6.1 将记录用于智能指针

在 Object Pascal 中，对象是通过引用管理的，而记录的生存期则绑定到声明它们的方法。该方法结束后，将清除记录的存储区。因此，我们可以做的是使用记录来管理 Object Pascal 对象的生存期。

但是，在 Delphi 10.4 之前，Object Pascal 记录无法提供在销毁时执行自定义代码的方式，托管记录引入了此功能。相反，旧的机制是在记录中使用接口字段，因为该接口字段是受管理的，并且用于实现该接口的对象的引用计数减少了。

另一个考虑因素是我们要使用标准记录还是泛型记录。

对于具有 TObject 类型字段的标准记录，您可以在需要时删除该对象，这样就足够了。但是，使用泛型版本，您可以获得两个优点：

- 通用智能指针可以返回对其包含的对象的引用，因此您无需保留两个引用。
- 通用智能指针可以使用无参数构造函数自动创建容器对象。

在这里，我将仅介绍两个使用泛型记录实现的智能指针示例，即使这会增加一些额外的复杂性。起点将是对对象有约束的泛型记录，例如：

```
type
  TSmartPointer<T: class> = record
  strict private
    FValue: T;
    function GetValue: T;
  public
    constructor Create(AValue: T);
    property Value: T read GetValue;
  end;
```

记录的 Create 和 GetValue 方法只需分配并读回该值。用例场景是以下代码片段，该代码片段创建一个对象，创建一个包装它的智能指针，还允许使用智能指针引用嵌入式对象并调用其方法（请参见下面的最后一行代码）：

```
var
  SL: TStringList;
begin
  SL := TStringList.Create;
  var SmartP: TsmartPointer<TStringList>.Create (SL);
  SL.Add('foo');
  SmartP.Value.Add ('bar');
```

正如您可能已经了解到的那样，此代码导致内存泄漏的方式与没有智能指针时完全相同！实际上，记录超出范围会被破坏，但不会释放内部对象。

14.6.2 使用泛型托管记录实现智能指针

尽管智能指针记录最相关的操作是它的完成，但是在下面的代码中（如您所见），我还将添加一个初始化运算符，以将对象引用设置为 nil。理想情况下，我们会阻止任何赋值操作（因为要与内部对象建立多个链接将需要配置相当复杂的引用计数机制），但是鉴于此不可能，我添加了运算符并实现了它以引发异常如果被触发。

这是泛型托管记录的完整代码：

```
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;
```

注意，在类约束之外，泛型记录还具有构造函数约束，因为我希望能够创建泛型数据类型的对象。如果调用 GetValue 方法并且尚未初始化该字段，则会发生这种情况。这是所有方法的完整代码：

```
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;
```

此代码是 SmartPointersMR 项目的一部分，其中还包括如何使用智能指针的示例。第一个严格类似于我们在后面几页中考虑的示例代码：

```
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;
```

但是，由于泛型智能指针支持自动构造指定类型的对象，因此您也可以摆脱使用显式变量，该显式变量引用字符串列表和创建它的代码：

```
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;
```

在程序中，可以通过在初始化代码中将全局 ReportMemoryLeaksOnShutdown 设置为 True 来验证所有对象实际上是否已销毁并且没有内存泄漏。作为计数器测试，程序中有一个导致泄漏的按钮，该泄漏在程序终止时被捕获。

14.6.3 使用泛型记录和接口实现智能指针

如前所述，在 Delphi 10.4 提供托管记录之前，实现智能指针的一种可能方法是使用接口，因为该记录将自动释放接口字段引用的对象。尽管现在这种方法不再那么有趣了，但它确实提供了一些附加功能，例如隐式转换运算符。鉴于这仍然是一个有趣，复杂的示例，我决定保留它，并以某种方式减少描述（这是源中的 SmartPointers 项目）。

要使用接口实现智能指针，可以编写绑定到接口的内部支持类，并使用接口引用计数机制来确定何时释放对象。内部类如下所示：

```
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;
```

我已将其声明为泛型智能指针类型的嵌套类型。为了启用此功能，我们在智能指针泛型类型中要做的就是添加接口引用，并使用 TFreeTheValue 对象对其进行初始化，该对象引用所包含的对象：

```
  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;
```

伪构造函数变为：

```
  constructor TSmartPointer<T>.Create(AValue: T);
  begin
    FValue := AValue;
    FFreeTheValue := TFreeTheValue.Create(FValue);
  end;
```

有了此代码之后，我们现在可以在程序中编写以下代码而不会引起内存泄漏（同样，该代码与我最初列出并在管理器记录版本中使用的代码相似）：

```
  procedure TFormSmartPointers.BtnSmartClick(Sender: TObject);
  var
    SL: TStringList;
    SmartP: TSmartPointer<TStringList>;
  begin
    SL := TStringList.Create;
    SmartP.Create (SL);
    SL.Add('foo');
    Show ('Count: ' + IntToStr (SL .Count));
  end;
```

在该方法的末尾，将放置 SmartP 记录，这将导致其内部接口对象被破坏，从而释放 TStringList 对象。

❖ 即使引发异常，该代码也可以工作。实际上，当您使用托管类型时，编译器会在所有位置添加隐式 try-finally 块，例如在这种情况下，带有接口字段的记录。

14.6.4 添加隐式转换

使用托管记录解决方案时，我们需要格外小心以避免记录复制操作，因为这将需要添加手动引用计数机制并使结构更复杂。但是，鉴于此功能已内置在基于接口的解决方案中，我们可以利用此模型添加转换运算符，从而可以简化数据结构的初始化和创建。

具体来说，我将添加一个隐式转换运算符以将目标对象分配给智能指针：

```
  class operator TSmartPointer<T>.Implicit(AValue: T): TSmartPointer<T>;
  begin
    Result := TSmartPointer<T>.Create(AValue);
  end;
```

使用此代码（并利用 Value 字段），我们现在可以编写更紧凑的代码版本，

例如：

```
var
  SmartP: TSmartPointer<TStringList>;
begin
  SmartP := TStringList.Create;
  SmartP.Value.Add('foo');
  Show ('Count: ' + IntToStr (SmartP.Value.Count));
```

或者，我们可以使用 TStringList 变量并使用更复杂的构造函数来初始化智能指针记录，即使没有显式引用也可以：

```
var
  SL: TStringList;
begin
  SL := TSmartPointer<TStringList>.Create(TStringList.Create).Value;
  SL.Add('foo');
  Show ('Count: ' + IntToStr (SL .Count));
```

当我们开始这条路时，我们还可以定义相反的转换，并使用强制转换符号而不是 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('foo2');
```

现在，您可能还注意到，我在上面的代码中一直使用伪构造函数，但这在记录中并不需要。我们需要的是一种初始化内部对象的方法，可能是在我们第一次使用它时调用它的构造函数。

我们无法测试内部对象是否已分配，因为记录（与类不同）未初始化为零。但是，我们可以对已初始化的接口变量执行该测试。

14.6.5 Comparing（比较）智能指针解决方案

智能指针的托管记录版本更简单且相当有效，但是基于接口的版本提供了转换运算符的优势。

它们都有优点，尽管我个人倾向于托管记录版本。

对于更清晰的分析和更复杂的解决方案（超出本书的范围），我可以推荐 Erik van Bilsen 撰写的以下博客文章：

https://blog.grijjy.com/2020/08/12/custom-managed-records-for-smart-pointers/

14.7 具有泛型的 Covariant（协变）返回类型

通常，在 Object Pascal（以及大多数其他静态的面向对象的语言）中，方法可以返回类的对象，但是您不能在派生类中重写它以返回派生类对象。这是一种相当普遍的做法，称为“协变量返回类型”，并且某些语言（如 C ++）已明确支持。

14.7.1 Animals（动物），Dog（狗）和 Cat（猫）

用编码的术语来说，如果 TDog 继承自 TAnimal，我想拥有以下方法：

```
function TAnimal.Get (AName: string): TAnimal;
function TDog.Get (AName: string): TDog;
```

但是，在 Object Pascal 中，您不能具有带有不同返回值的虚函数，也不能在返回类型上重载，而只能在使用不同参数时使用。让我向您展示一个简单演示的完整代码。这是涉及的三个类：

```
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;
```

一旦您注意到类函数实际上用于创建新对象（内部调用构造函数），则这两个方法的实现非常简单。

原因是我不想直接创建构造函数，因为这是一种更通用的技术，其中类的方法可以创建其他已分类（或类层次结构）的对象。这是代码：

```
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;
```

现在我们可以通过编写以下代码来使用该类，这是我不喜欢的，因为我们必须将结果回退为正确的类型：

```
var
  ACat: TCat;
begin
  ACat := TCat.Get('Matisse') as TCat;
  Memo1.Lines.Add (ACat.ToString);
  ACat.Free;
```

同样，我想做的就是能够分配 TCat 返回的值。无需显式转换即可获取 TCat 类的引用。我们该怎么做？

14.7.2 具有泛型返回值的方法

事实证明，泛型可以帮助我们解决问题。不是泛型类型，这是泛型的最常用形式。 但是本章前面讨论了非泛型类型的泛型方法。我可以添加到 TAnimal 类中的是带有泛型类型参数的方法，例如：

```
class function GetAs<T: class> (const AName: string): T;
```

此方法需要泛型类型参数，该参数必须是类（或实例类型），并返回该类型的对象。这里是一个示例实现：

```
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;
```

现在我们可以创建一个实例，并使用它省略 as 强制转换，尽管我们仍然必须将类型作为参数传递：

```
var
   ADog: TDog;
begin
   ADog := TDog.GetAs<TDog>('Pluto');
   Memo1.Lines.Add (ADog.ToString);
   ADog.Free;
```

14.7.3 返回不同类的派生对象

当您返回相同类的对象时，可以使用适当的构造函数替换此代码。但是使用泛型获取协变返回类型实际上更灵活。实际上，我们可以使用它来返回不同类或类层次结构的对象：

```
type
   TAnimalShop = class
      class function GetAs<T: TAnimal, constructor> (
         const AName: string): T;
   end;
```

❖ 此类用于创建不同类（或多个，取决于参数）的对象的类通常称为“类工厂”。

现在，我们可以使用特定的类约束（类本身是不可能的），并且必须指定构造函数约束，以便能够从泛型方法中创建给定类的对象：

```
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;
```

注意，现在在调用中，我们不必重复两次类类型：

```
ADog := TAnimalShop.GetAs<TDog>('Pluto');
```

第十五章 Anonymous（匿名）方法

Object Pascal 语言既包括过程类型（即，声明指向过程和函数的指针的类型），又包括方法指针（即，声明指向方法的指针的类型）。

❖ <u>如果需要更多信息，第 4 章介绍了过程类型，而第 10 章介绍了事件和方法指针类型。</u>

尽管您可能很少直接使用它们，但是这些是每个开发人员都可以使用的 Object Pascal 的关键功能。实际上，方法指针类型是组件和可视控件中事件处理程序的基础：每次声明事件处理程序，甚至是简单的 Button1Click，实际上都在声明将与事件连接的方法（在 OnClick 事件中，这种情况）使用方法指针。

匿名方法通过允许您将方法的实际代码作为参数而不是在其他地方定义的方法的名称来传递，从而扩展了此功能。但是，这并不是唯一的区别。匿名方法与其他技术的不同之处在于它们管理局部变量的生存期的方式。

上面的定义与许多其他语言（例如 JavaScript）中称为闭包的功能相匹配。如果 Object Pascal 匿名方法实际上是闭包，那么该语言怎么用不同的术语来引用它们呢？原因在于这两个术语都被不同的编程语言使用，并且 Embarcadero 生产的 C ++编译器使用术语闭包来表示 Object Pascal 所谓的事件处理程序。

多年以来，匿名方法在相当多的编程语言（最著名的是动态语言）中以不同的形式和不同的名称出现。我在 JavaScript 闭包方面拥有丰富的经验，尤其是 jQuery 库和 AJAX 调用。C# 中的相应功能称为匿名委托。

但是在这里，我不想花太多时间比较各种编程语言中的闭包和相关技术，而是详细描述它们在 Object Pascal 中的工作方式。

❖ <u>从很高的角度来看，泛型允许对类型的代码进行参数化，匿名方法允许对方法的参数进行参数化。</u>

15.1 匿名方法的语法和语义

Object Pascal 中的匿名方法是一种在表达式上下文中创建方法值的机制。 一个相当神秘的定义，但给定一个相当精确的定义，强调了与方法指针（表达式上下文）的关键区别。不过，在开始之前，让我从一个非常简单的代码示例开始（该示例与本节中的大多数其他代码段一起包含在 AnonymFirst 应用程序项目中）。

这是一个匿名方法类型的声明，鉴于 Object Pascal 是强类型语言，您需要提供一些信息：

```
type
  TIntProc = reference to procedure (N: Integer);
```

这与方法引用类型的不同之处仅在于用于声明的关键字：

```
type
  TIntMethod = procedure (N: Integer) of object;
```

15.1.1 匿名方法变量

在最简单的情况下，一旦有了匿名方法类型，就可以声明该类型的变量，分配类型兼容的匿名方法，然后通过该变量调用该方法：

```
procedure TFormAnonymFirst.BtnSimpleVarClick(Sender: TObject);
var
  AnIntProc: TIntProc;
```

```
begin
  AnIntProc := procedure (N: Integer)
               begin
                 Memo1.Lines.Add (IntToStr (N));
               end;
  AnIntProc (22);
end;
```

请注意用于使用本地代码将实际过程分配给本地变量 AnIntProc 的语法。

15.1.2 匿名方法参数

作为一个更有趣的示例（具有更令人惊讶的语法），我们可以将匿名方法作为参数传递给函数。假设您有一个使用匿名方法参数的函数：

```
procedure CallTwice (Value: Integer; AnIntProc: TIntProc);
begin
  AnIntProc (Value);
  Inc (Value);
  AnIntProc (Value);
end;
```

该函数使用两个连续的整数值调用两次作为参数传递的方法，其中一个作为参数传递，随后一个整数。您可以通过向函数传递实际的匿名方法来调用该函数，并使用直接就地代码看起来很令人惊讶：

```
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;
```

从语法角度来看，请注意该过程将作为参数传递给括号内，并且不以分号终止。该代码的实际效果是调用 48 和 49 的 IntToHex 以及 100 和 101 的平方根的 FloatToStr，产生以下输出：

```
0030
0031
10
10.0498756211209
```

15.2 使用局部变量

使用方法指针，即使语法不同且可读性较低，我们也可以达到相同的效果。使匿名方法明显不同的是它们可以引用调用方法的局部变量的方式。考虑以下代码：

```
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;
```

此处，仍传递给 CallTwice 过程的方法使用局部参数 N，但也使用来自其上下文的局部变量 ANumber。

有什么作用？匿名方法的两次调用将修改局部变量，并向其添加参数，第一次为 10，第二次为 11。ANumber 的最终值为 21。

15.2.1 延长局部变量的寿命

前面的示例显示了一个有趣的效果，但是通过一系列嵌套函数调用，您可以使用局部变量的事实并不令人惊讶。但是，匿名方法的强大之处在于它们可以使用局部变量，也可以延长其生存期，直到需要使用为止。一个例子将证明这一点，而不是冗长的解释。

❖ 在稍微详细的技术细节上，匿名方法在创建它们时将它们使用的变量和参数复制到堆中，并使其保持活动状态，直到匿名方法的特定实例为止。

我在 AnonymFirst 应用程序项目的 TFormAnonymFirst 表单类中添加了（使用类完成功能）一个匿名方法指针类型的属性（嗯，实际上我在项目的所有代码中都使用了相同的匿名方法指针类型）：

```
private
  FAnonMeth: TIntProc;
  procedure SetAnonMeth(const Value: TIntProc);
public
  property AnonMeth: TIntProc read FAnonMeth write SetAnonMeth;
```

然后，我在程序的窗体中添加了两个按钮。第一个将属性保存为使用局部变量的匿名方法（与上一个 BtnLocalValClick 方法类似）：

```
procedure TFormAnonymFirst.BtnStoreClick(Sender: TObject);
var
  ANumber: Integer;
begin
  ANumber := 3;
  AnonMeth := procedure (N: Integer)
              begin
                Inc (ANumber, N);
                Show (IntToStr (ANumber));
              end;
end;
```

当此方法执行时，匿名方法不执行，仅存储。

局部变量 ANumber 初始化为 3，未修改，超出了局部范围（随着方法的终止），并且被替换了。至少，这就是您对标准 Object Pascal 代码的期望。

我为该特定步骤添加到表单的第二个按钮调用了存储在 AnonMeth 属性中的匿名方法：

```
procedure TFormAnonymFirst.BtnCallClick(Sender: TObject);
begin
  if Assigned (AnonMeth) then
  begin
    CallTwice (2, AnonMeth);
  end;
end;
```

执行此代码后，它将调用匿名方法，该方法使用不再位于堆栈中的方法的局部变量 ANumber。但是，由于匿名方法捕获了它们的执行上下文，因此只要存在匿名方法的给定实例（即对该方法的引用）就可以使用该变量。

作为进一步的证明，请执行以下操作。按下一次"Store"按钮，两次按下"Call"按钮，您将看到正在使用相同的捕获变量：

```
5
8
10
13
```

❖ 此序列的原因是该值从 3 开始，每次对 CallTwice 的调用都将其参数第一次（即 2）传递给匿名方法，然后在将其递增（即第二次传递 3 时）传递给匿名方法）。

现在，再次按存储，然后再次按 Call。会发生什么，为什么要重置局部变量的值？通过分配新的匿名方法实例，可以删除旧的匿名方法（以及其自己的执行上下文），并捕获新的执行上下文，包括局部变量的新实例。全序列存储-呼叫-呼叫-存储-呼叫产生：

```
5
8
10
13
5
8
```

正是这种行为的含义，类似于某些其他语言所做的事情，才使匿名方法成为一种极其强大的语言功能，您可以使用它来实现过去根本不可能实现的功能。

15.3 幕后的匿名方法

如果变量捕获功能是与匿名方法最相关的功能之一，那么在我们关注一些实际示例之前，还有更多技术值得研究。不过，如果您不熟悉匿名方法，则可能要跳过此高级部分，然后在第二遍阅读中返回。

15.3.1 （可能）缺少括号

注意，在上面的代码中，我使用 AnonMeth 符号来引用匿名方法，而不是调用它。为了调用它，我应该输入：

```
AnonMeth (2)
```

区别很明显；我需要传递一个适当的参数来调用该方法。

无参数匿名方法会使事情更加混乱。如果您声明：

```
type
  TAnyProc = reference to procedure;
var
  AnyProc: TAnyProc;
```

在对 AnyProc 的调用之后必须加上一个空括号，否则编译器会认为您正在尝试获取该方法（其地址），而不是调用它：

```
AnyProc ();
```

当您调用返回匿名方法的函数时，会发生类似的情况，例如以下情况来自通常的 AnonymFirst 应用程序项目：

```
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;
```

现在的问题是，你怎么称呼它？如果你只是调用

```
GetShowMethod;
```

它编译并执行，但是它所要做的就是调用匿名方法分配代码，丢弃该函数返回的匿名方法。

您如何调用传递参数的实际匿名方法？一种选择是使用临时匿名方法变量：

```
var
   Ip: TIntProc;
begin
   Ip := GetShowMethod();
   Ip (3);
```

注意，在这种情况下，GetShowMethod 调用之后的括号。如果省略它们（Pascal 的一种标准做法），则会出现以下错误：

E2010 Incompatible types: 'TIntProc' and 'Procedure'

（E2010 不兼容的类型：“ TIntProc”和“Procedure”）

如果没有括号，编译器会认为您要分配 GetShowMethod 函数本身，而不是将其结果分配给 Ip 方法指针。不过，在这种情况下，使用临时变量可能不是最佳选择，因为这会使代码异常复杂。一个简单的调用

```
GetShowMethod(3);
```

无法编译，因为您无法将参数传递给方法。您需要将空括号添加到第一个调用中，并将 Integer 参数添加到生成的匿名方法中。奇怪的是，您可以编写：

```
GetShowMethod()(3);
```

15.3.2 匿名方法实现

在实现匿名方法的幕后会发生什么？

编译器为匿名方法生成的实际代码基于接口，具有一个称为 Invoke 的单个（隐藏）调用方法，以及通常的引用计数支持（这对于确定匿名方法的生存期和捕获的上下文很有用）。

获取内部细节的过程可能非常复杂且价值有限。

可以说，就速度而言，该实现非常高效，并且每个匿名方法都需要大约 500 个额外的字节。

换句话说，Object Pascal 中的方法引用是通过特殊的单个方法接口实现的，编译器生成的方法与正在实现的方法引用具有相同的签名。该接口利用引用计数进行自动处理。

❖ 尽管实际上用于匿名方法的接口看起来与任何其他接口一样，但是编译器会区分这些特殊接口，因此您不能将它们混入代码中。

在此隐藏接口旁边，对于匿名方法的每次调用，编译器都会创建一个隐藏对

象，该对象具有方法实现和捕获调用上下文所需的数据。这样，您可以为该方法的每次调用获取一组新的捕获变量。

15.3.3 准备使用参考类型

每次将匿名方法用作参数时，都需要定义相应的引用指针数据类型。为了避免局部类型的泛滥，Object Pascal 在 System.SysUtils 单元中提供了许多现成的引用指针类型。如下面的代码片段所示，大多数这些类型定义都使用参数化类型，因此对于单个通用声明，每种可能的数据类型都有不同的引用指针类型：

```
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);
```

使用这些声明，您可以定义采用匿名方法参数的过程，如下所示：

```
procedure UseCode (Proc: TProc);
function DoThis (Proc: TProc): string;
function DoThat (ProcInt: TProc<Integer>): string;
```

在第一种和第二种情况中，您传递了无参数匿名方法，在第三种情况中，您传递了具有单个 Integer 参数的方法：

```
UseCode ( procedure
          begin
            ...
          end);
StrRes := DoThat ( procedure (I: Integer)
                   begin
                     ...
                   end);
```

同样，System.SysUtils 单元定义了一组具有泛型返回值的匿名方法类型：

```
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;
```

这些定义非常广泛，因为您可以对多达四个参数和一个返回类型使用无数的数据类型组合。最后一个定义与第二个非常相似，但是对应于一个非常常见的特定情况，该函数采用泛型参数并返回布尔值。

15.4 现实世界中的匿名方法

乍一看，要完全了解匿名方法的功能以及可以从中受益的场景并不容易。这就是为什么我没有提出涉及该语言的更复杂的示例，而是决定将重点放在一些具有实际影响并为进一步探索提供起点的示例上。

15.4.1 匿名事件处理程序

Object Pascal 的显着特征之一是它使用方法指针实现了事件处理程序。可以使用匿名方法将新行为附加到事件，而不必声明单独的方法并捕获该方法的执行上下文。这样避免了将额外的字段添加到表单以将参数从一种方法传递到另一种方法的麻烦。

作为一个示例（AnonButton 应用程序项目），我向按钮添加了一个匿名 click 事件，声明了正确的方法指针类型，并向自定义按钮类（使用插入器类定义）添加了新的事件处理程序：

```
type
  TAnonNotif = reference to procedure (Sender: TObject);

  // interposer class（插入器类）
  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;
```

❖ 插入器类是派生类，其名称与其基类相同。可以使用两个具有相同名称的类，因为两个类的单元不同，因此它们的全名（unitname.classname）不同。声明插入器类很方便，因为您只需在窗体上放置一个 Button 控件并对其附加行为即可，而不必在 IDE 中安装新组件并将窗体上的控件替换为新类型。您必须记住的唯一技巧是，如果插入器类的定义位于单独的单元中（而不是此简单示例中的表单单元），则该单元必须在定义基类的单元之后的 uses 语句中列出 。

此类的代码非常简单，因为 setter 方法将保存新指针，而 Click 方法会在进行标准处理之前调用它（即，调用 OnClick 事件处理程序（如果可用））：

```
procedure TButton.SetAnonClick(const Value: TAnonNotif);
begin
  FAnonClick := Value;
end;

procedure TButton.Click;
begin
  if Assigned (FAnonClick) then
    FAnonClick (self)
  inherited;
end;
```

您如何使用此新事件处理程序？基本上，您可以为其分配一个匿名方法：

```
procedure TFormAnonButton.BtnAssignClick(Sender: TObject);
begin
  BtnInvoke.AnonClick := procedure (Sender: TObject)
                         begin
                           Show ((Sender as TButton).Text);
                         end;
end;
```

现在，这看起来毫无意义，因为使用标准事件处理程序方法可以轻松实现相

同的效果。相反，以下内容开始有所作为，因为匿名方法通过引用 Sender 参数捕获对分配了事件处理程序的组件的引用。

这可以在将其临时分配给局部变量之后完成，因为匿名方法的 Sender 参数隐藏了 BtnKeepRefClick 方法的 Sender 参数：

```
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;
```

按下 BtnInvoke 按钮时，您将看到其标题以及分配了匿名方法处理程序的组件的名称。

15.4.2 计时匿名方法

开发人员经常将计时代码添加到现有例程中，以比较它们的相对速度。假设您有两个代码片段，并且想通过执行几百万次来比较它们的速度，则可以编写以下内容，该内容摘自第 6 章的 LargeString 应用程序项目：

```
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;
```

第二种方法具有类似的代码，但是使用了 TStringBuilder 类而不是简单的串联。现在，我们可以在 AnonLargeStrings 应用程序项目中利用匿名方法创建计时框架，并将特定的代码作为参数传递，就像我在代码的更新版本中所做的那样。

无需一遍又一遍地重复计时代码，您可以编写一个具有计时代码的函数，该函数将通过无参数的匿名方法调用代码片段：

```
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;
```

但是请注意，如果执行标准版本和基于匿名方法的版本，您将获得略有不同的输出，匿名方法版本将受到大约 10％的罚款。原因是程序必须直接对匿名方法实现进行虚拟调用，而不是直接执行本地代码。由于这种差异是一致的，因此无论如何，测试代码都是非常合理的。

但是，如果您需要从代码中压缩性能，那么使用匿名方法将不会像使用直接函数那样直接编写代码那样快。

就性能而言，使用非虚拟方法指针可能会介于两者之间。

15.4.3 Threads Synchronization（线程同步）

在需要更新用户界面的多线程应用程序中，如果没有同步机制，您将无法访问属于主线程的可视组件（或内存对象）的属性。Delphi 视觉组件库并不是天生就具有线程安全性（大多数用户界面库都是如此）。

两个线程同时访问一个对象可能会损害其状态。

Object Pascal 中 TThread 类提供的经典解决方案是调用特殊方法 Synchronize，将对另一个方法的引用作为参数传递给该方法，该方法可以安全地执行。第二种方法不能具有参数，因此通常的做法是在线程类中添加额外的字段，以将信息从一种方法传递到另一种方法。

作为一个实际示例，在 Mastering Delphi 2005 一书中，我编写了一个 WebFind 应用程序（该程序可通过 HTTP 在 Google 上运行搜索，并从页面的 HTML 中提取结果链接），并具有以下线程类：

```
type
  TFindWebThread = class(TThread)
  protected
    FAddr, FText, FStatus: 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;
```

引入了三个受保护的字符串字段以及一些其他方法来支持与用户界面的同步。例如，挂钩到内部 THTTPClient 组件（Delphi HTTP 客户端库的组件的一部分）的 OnReceiveData 事件的 HttpWork 事件处理程序曾经具有以下代码，称为 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 RTL 的 Synchronize 方法具有两个不同的重载定义：

```
type
  TThreadMethod = procedure of object;
  TThreadProcedure = reference to procedure;
  TThread = class
  ...
  procedure Synchronize(AMethod: TThreadMethod); overload;
  procedure Synchronize(AThreadProc: TThreadProcedure); overload;
```

因此，我们可以删除 FStatus 文本字段和 ShowStatus 函数，并使用新版本的 Synchronize 和匿名方法重写 HttpWork 事件处理程序：

```
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;
```

在类的所有代码中使用相同的方法，线程类将变为以下类（您可以在本书的源代码随附的 WebFind 应用程序项目的版本中找到这两个线程类）：

```
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;
```

使用匿名方法可以简化线程同步所需的代码，因为可以避免使用临时字段。

❖ 匿名方法与线程有很多关系，因为一个线程用于运行某些代码，而匿名方法表示代码。这就是为什么在 TThread 类中支持使用它们，而且在并行编程库（在 TParallel.For 中定义 TTask）中也提供了支持。鉴于对多线程的研究已经

超出了本章的范围，因此我不会在这个方向上添加更多示例。不过，在下一个示例中，我将使用另一个线程，因为在进行 HTTP 调用时，这通常是必需的。

15.4.4 Object Pascal 中的 AJAX

本节中的最后一个示例 AnonAjax 应用程序演示是我最喜欢的匿名方法示例之一（即使有些极端）。原因是几年前我学会了在 JavaScript 中使用闭包（或匿名方法），同时使用 jQuery 库编写 AJAX 应用程序。

❖ AJAX 的缩写代表 Asynchronous（异步）JavaScript XML，因为它最初是从浏览器进行的 Web 服务调用中使用的格式。随着这项技术变得越来越流行和广泛，并且 Web 服务转移到 REST 体系结构和 JSON 格式，AJAX 这个术语已经淡出了一部分，以支持 REST。考虑到它说明了该应用程序的用途及其背后的历史，我还是决定保留该演示的旧名称。您可以在以下网址阅读更多信息：https://en.wikipedia.org/wiki/Ajax_(programming)。

AjaxCall 全局函数产生一个线程，并向该线程传递一个匿名方法以在完成时执行。该函数只是线程构造函数的包装器：

```
type
  TAjaxCallback = reference to procedure (ResponseContent: TStringStream);
procedure AjaxCall (const StrUrl: string; AjaxCallback: TAjaxCallback);
begin
  TAjaxThread.Create (StrUrl, AjaxCallback);
end;
```

所有代码都在 TAjaxThread 类中，TAjaxThread 类是带有内部 HTTP 客户端组件（来自 Delphi HTTP Client 库）的线程类，用于 Asynchronous（异步）访问给定的 URL：

```
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;
```

构造函数进行一些初始化，将其参数复制到线程类的相应本地字段中，并创建 FHttp 对象。该类的真实内容在其 Execute 方法中，该方法执行 HTTP 请求，将结果保存在流中，该流随后被重置并传递给回调函数–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;
```

作为其用法的一个示例，AnonAjax 应用程序项目具有一个按钮，用于将网页的内容复制到 Memo 控件（添加请求的 URL 之后）：

```
    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;
```

HTTP 请求完成后，您可以对其进行任何处理。一个示例是从 HTML 文件中提取链接（其方式类似于前面介绍的 WebFind 应用程序）。再次，为了使此函数具有灵活性，它以匿名方法为每个链接执行的参数作为参数：

```
    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
          // 查找 HTTP 参考的结尾
          NBegin := NBegin + 6;
          NEnd := PosEx ('"', StrData, NBegin);
          StrAddr := Copy (StrData, NBegin, NEnd - NBegin);
          // 继续
          NBegin := NEnd + 1;
          // 执行匿名方法
          ProcLink (StrAddr)
        end;
      until NBegin = 0;
    end;
```

如果将此函数应用于 AJAX 调用的结果并提供进一步的处理方法，则最终将导致两个嵌套的匿名方法调用，例如在 AnonAjax 应用程序项目的第二个按钮中：

```
    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;
```

在这种情况下，Memo 控件将接收链接的集合，而不是返回页面的 HTML。上面的链接提取例程的变体将是图像提取例程。ExtractImages 函数获取返回的 HTML 文件的 img 标记的源（src），并调用另一个 TLinkCallback 兼容的匿名方法（有关功能的详细信息，请参见源代码）。

现在，您可以设想打开一个 HTML 页面（使用 AjaxCall 函数），提取图像链接，然后再次使用 AjaxCall 来获取实际图像。这意味着在某些 Object Pascal 程序员可能会发现非常不寻常的编码结构中使用三重嵌套闭包，但这无疑是非常强大和富有表现力的：

```
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
            var RealURL := IfThen (AURL[1]='/',
              EdUrl.Text + AURL, AURL); // expand URL
            AjaxCall (RealUrl,
              procedure (AResponseContent: TStringStream)
              begin
                Image1.Picture.Graphic.
                  LoadFromStream (AResponseContent);
              end);
          end;
        end);
    end);
end;
```

❖ 该代码段是我的 2008 年 9 月的博客文章“Anonymous, Anonymous, Anonymous”的主题，该文章引起了一些评论，如您所见：http://blog.marcocantu.com/blog/anonymous_3.html。

除了仅在加载与 Image 组件中已经存在的格式相同的文件的情况下图形才起作用的事实之外，代码及其结果都令人印象深刻。请特别注意基于 NHit 局部变量的捕获的编号顺序。如果快速按两次按钮会怎样？每个匿名方法将获得不同的 NHit 计数器副本，并且它们可能在列表中不按顺序显示，第二个线程在第一个线程之前开始产生其输出。使用最后一个按钮的结果如图 15.1 所示。

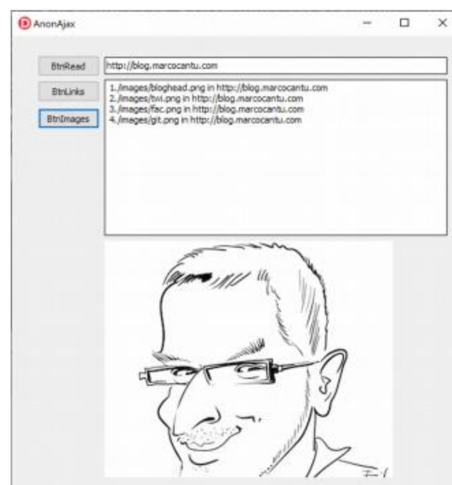

图 15.1：三重嵌套匿名方法调用的输出，用于从网页检索图像。

第十六章 reflection（反射）与 attributes（属性）

传统上，使用强静态类型语言（例如 Pascal）的编译器在运行时几乎没有提供关于可用类型的信息。有关数据类型的所有信息仅在编译阶段可见。

Object Pascal 的第一个版本打破了这一传统，它为发布的带有特定编译器指令的属性和其他类成员提供了运行时信息。已为使用特定设置{$ M +}编译的类启用了此功能，它是 VCL 的 DFM 文件（和 FireMonkey 库的 FMX 文件）背后的流传输机制的基础，以及表单和其他可视化文件的使用方式设计师。当它最初在 Delphi 1 中可用时，此功能是一个全新的想法，后来其他开发工具采用了该功能并将其扩展为几种方式。

首先，对类型系统进行了扩展（仅在 Object Pascal 中可用），以解决 COM 中的方法发现和动态调用问题。Object Pascal 仍通过分派 ID，将方法应用于变量以及其他与 COM 相关的功能来支持此功能。最终，Object Pascal 中的 COM 支持以其自身的运行时类型信息风格得到了扩展，但这是一个超出语言书籍范围的主题。

诸如 Java 和.NET 之类的托管环境的出现提出了一种非常广泛的运行时类型信息，其中详细的 RTTI 由编译器绑定到可执行模块，并且可供使用这些模块的程序发现。这具有揭露某些程序内部结构和增加模块大小的缺点，但是它带来了新的编程模型，该模型将动态语言的某些灵活性与坚固的结构和强类型的速度结合在一起。

无论您是否喜欢（在引入此功能时，这的确是激烈辩论的主题）Object Pascal 在朝着同一方向缓慢发展，并且广泛采用 RTTI 标志着迈出了非常重要的一步。那个方向。正如我们将看到的，您可以选择退出 RTTI，但如果不这样做，则可以在应用程序中利用一些额外的功能。

主题远非简单，因此我将逐步进行。我们将首先关注编译器中内置的新扩展 RTTI 和可用于探索它的 Rtti 单元的新类。其次，我将研究新的 TValue 结构和动态调用。第三，我将介绍自定义属性，该属性与.NET 相似，并且让您扩展由编译器生成的 RTTI 信息。仅在本章的最后部分，我将尝试回到扩展 RTTI 背后的原因，并查看其用法的实际示例。

16.1 扩展 RTTI

默认情况下，Object Pascal 编译器会生成许多扩展的 RTTI 信息。

此运行时信息包括所有类型，包括类和所有其他用户定义的类型，以及编译器预定义的核心数据类型，并涵盖已发布的字段以及公共字段，甚至是受保护的元素和私有元素。这需要能够深入研究任何对象的内部结构。

16.1.1 第一个例子

在我们研究由编译器生成的信息以及用于访问它们的各种技术之前，让我跳到结论并向您展示使用 RTTI 可以完成的工作。具体示例非常少，可以用较旧的 RTTI 编写，但是它应该使您了解我在说什么（还应考虑到并非所有 Object Pascal 开发人员都明确使用了传统的 RTTI）。

假设您有一个带有按钮的表单，例如在 RttiIntro 应用程序项目中。

您可以编写以下代码来读取控件的 Text 属性的值：

```
uses
  Rtti;

procedure TFormRttiIntro.BtnInfoClick(Sender: TObject);
var
  Context: TRttiContext;
begin
  Show (Context.
    GetType(TButton).
    GetProperty('Text').
    GetValue(Sender).ToString);
end;
```

该代码使用 TRttiContext 记录来引用有关 TButton 类型的信息，从该类型信息到有关属性的 RTTI 数据，该属性数据用于引用该属性的实际值，该值将转换为字符串。如果您想知道这是如何工作的，请继续阅读。我的意思是，这种方法现在不仅可以用于动态访问属性，而且还可以读取字段（包括私有字段）的值。

我们还可以更改属性的值，如 RttiIntro 应用程序项目的第二个按钮所示：

```
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;
```

此代码用随机数* s 替换文本。与上面的代码的不同之处在于，它具有一个临时的局部变量，该局部变量引用了该属性的 RTTI 信息。现在您已经了解了我们所要研究的内容，让我们从头开始，检查编译器生成的扩展 RTTI 信息。

16.1.2 编译器生成的信息

您无需执行任何操作即可让编译器将此额外信息添加到您的可执行程序（无论其类型是：应用程序，库，程序包…）。只需打开一个项目并进行编译。默认情况下，编译器会为所有字段（包括私有字段）以及公共和已发布的方法和属性生成扩展 RTTI。您可能会为私有字段获得 RTTI 信息而感到惊讶，但这对于诸如二进制对象序列化和跟踪堆上的对象之类的动态操作是必需的。

您可以根据设置矩阵控制扩展 RTTI 的生成：在一个轴上您可以看到，而在另一个轴上您可以看到这种成员。下表描述了系统默认值：

	Field	Method	Property
Private	x		
Protected	x		
Public	x	x	x
Published	x	x	x

从技术上讲，这四个可见性设置是通过在系统单元中声明的以下设置类型指示的：

```
type
  TVisibilityClasses = set of (vcPrivate, vcProtected, vcPublic, vcPublished);
```

有一些现成的常量可以用于此集合，这些常量指示默认的 RTTI 可见性设置应用于 TObject 并默认由所有其他类继承：

```
const
  DefaultMethodRttiVisibility = [vcPublic, vcPublished];
  DefaultFieldRttiVisibility = [vcPrivate..vcPublished];
  DefaultPropertyRttiVisibility = [vcPublic, vcPublished];
```

编译器产生的信息由新指令$ RTTI 控制，该指令的状态指示设置是针对给定类型还是其后代（EXPLICIT 或 INHERITED），然后由三个说明符设置方法的可见性，字段和属性。在系统单元中应用的默认值为：

```
{$RTTI INHERIT
  METHODS(DefaultMethodRttiVisibility)
  FIELDS(DefaultFieldRttiVisibility)
  PROPERTIES(DefaultPropertyRttiVisibility)}
```

要完全禁用类的所有成员的扩展 RTTI 的生成，可以使用以下指令：

```
{$RTTI EXPLICIT METHODS([]) FIELDS([]) PROPERTIES([])}
```

❖ 您不能像其他编译器指令那样将 RTTI 指令放在单元声明之前，因为它取决于系统单元中定义的设置。如果这样做，您将收到一条内部错误消息，该消息不是特别直观。无论如何，只要将其移动到 unit 语句之后即可。

使用此设置时，请考虑将其仅应用于您的代码，并且无法完全删除，因为 RTL 和其他库类的 RTTI 信息已被编译到相应的 DCU 和程序包中。还请记住，$ RTTI 指令不会对为已发布类型生成的传统 RTTI 造成任何更改：无论$ RTTI 指令如何，此指令仍会生成。

❖ RTTI 处理类可在 System.Rtti 单元中获得，并在后面的部分中进行介绍，它们与传统的 RTTI 及其 PTypeInfo 结构挂钩。

使用此伪指令可以执行的操作是停止为您自己的类生成扩展 RTTI。在规模的另一端，如果愿意，您还可以增加生成的 RTTI 数量，包括私有方法和受保护的方法以及属性（尽管这没有多大意义）。

向可执行文件中添加扩展 RTTI 信息的明显效果是文件将变大（其主要缺点是要分发更大的文件，因为额外的加载时间和内存占用空间不那么重要）。现在，您可以从程序单元中删除 RTTI，这可能会产生积极的效果……如果您决定不想在代码中使用 RTTI。正如您将在本章中看到的那样，RTTI 是一种强大的技术，在大多数情况下，值得拥有额外的可执行文件大小。

16.1.3 弱类型和强类型链接

您还可以做些什么来减小程序的大小？实际上，您可以做一些事情，即使它的影响不会很大，也很明显。

在评估可执行文件中可用的 RTTI 信息时，请考虑编译器添加的内容，链接器可能会删除。默认情况下，程序中未编译的类和方法将不会获得扩展 RTTI（这将是毫无用处的），因为它们也不会获得基本 RTTI。在规模的另一端，如果要包括所有扩展 RTTI 并使其正常工作，则需要链接甚至未在代码中明确引用的类和方法。

您可以使用两个编译器指令来控制链接到可执行文件的信息。第一个完全记录在案的是$ WeakLinkRTTI 指令。通过打开它，对于程序中未使用的类型，类型本身及其 RTTI 信息都将从最终可执行文件中删除。

另外，您可以使用$ StrongLinkTypes 指令强制包含所有类型及其扩展 RTTI。对

许多程序的影响是巨大的，程序大小几乎增加了两倍。

16.2 RTTI 单元

如果所有类型的扩展 RTTI 的生成都是 Object Pascal 反射的第一大支柱，那么第二个支柱就是能够借助 System.Rtti 单元以轻松，高级的方式导航此信息。我们将在后面看到的第三个支柱是对自定义属性的支持。但是，让我一次前进一个步骤。

传统上，Object Pascal 应用程序可以（并且仍然可以）使用 System.TypInfo 单元的功能来访问“已发布”的运行时类型信息。该单元定义了几个低级数据结构和函数（全部基于指针和记录），以及几个高级例程，使事情变得简单一些。

相反，Rtti 单元使使用扩展的 RTTI 变得非常容易，它提供了一组具有适当方法和属性的类。要访问各种对象，入口点是 TRttiContext 记录结构，该结构具有四种查找可用类型的方法：

```
function GetType (ATypeInfo: Pointer): TRttiType; overload;
function GetType (AClass: TClass): TRttiType; overload;
function GetTypes: TArray<TRttiType>;
function FindType (const AQualifiedName: string): TRttiType;
```

如您所见，您可以传递一个类，一个从类型获得的 PTypeInfo 指针，一个限定名称（用单元名称修饰的类型名称，如“System.TObject”），或检索类型的完整列表，这些类型定义为 RTTI 类型的数组，或更精确地定义为 TArray <TRttiType>。

这是我在以下清单中使用的最后一个调用，它是 TypesList 应用程序项目中代码的简化版本：

```
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 方法返回数据类型的完整列表，但是程序仅过滤代表类的类型。单元中还有大约十二种表示类型的其他类。

❖ Rtti 单元将类类型称为“实例”或“实例类型”（如 TRttiInstance Type）。这有点令人困惑，因为我们通常使用术语“instance（实例）”来指代实际对象。

类型列表中的各个对象是从 TRttiType 基类继承的类。具体来说，我们可以寻找 TRttiInstanceType 类类型，如以下修改的代码段一样重写上面的代码：

```
for AType in TheTypes do
  if AType is TRttiInstanceType then
    Show(AType.QualifiedName);
```

该演示的实际代码稍微复杂一点，因为它首先填充字符串列表，对元素进行排序，然后再填充 ListView 控件，并使用 BeginUdpate 和 EndUdpate 进行优化（并尝试最终围绕它们以确保结尾操作始终执行）：

```
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;
```

这段代码产生了一个很长的列表，其中包含数百种数据类型，实际数量取决于平台和编译器的版本，如图 16.1 所示。 请注意，该图像列出了 RTTI 单元的类型，将在下一部分中介绍。

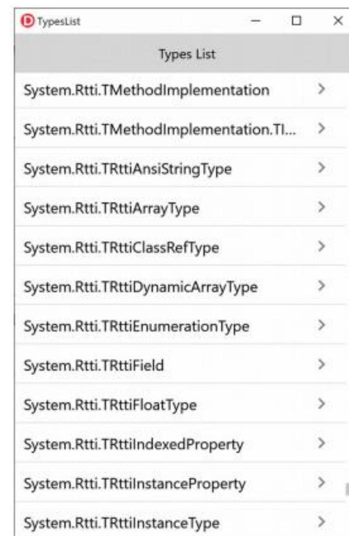

图 16.1：TypesList 应用程序项目的输出

16.2.1 Rtti 单元中的 RTTI 类

在下面的列表中，您可以看到从抽象 TRttiObject 类派生并在 System.Rtti 单元中定义的类的整个继承图：

```
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
```

这些类中的每一个都提供有关给定类型的特定信息。例如，仅 TRttiInterfaceType 提供了一种访问接口 GUID 的方法。

❖ 在 Rtti 单元的第一个实现中，没有 RTTI 对象可以访问索引属性（例如 TStringList 的 Strings []）。后来添加了它，现在可以使用，从而使运行时类型信息真正完整。

16.2.2 RTTI 对象生命周期管理和 TRttiContext 记录

如果您查看前面列出的 BtnTypesListClick 方法的源代码，则看起来有些错误。GetTypes 调用返回一个类型数组，但是代码不会释放这些内部对象。

原因是 TRttiContext 记录结构对所有正在创建的 RTTI 对象都具有所有者的效力。处理记录时（即超出范围时），将清除内部接口并调用其自己的析构函数，该析构函数将释放通过该记录创建的所有 RTTI 对象。

实际上，TRttiContext 记录具有双重作用。一方面，它控制 RTTI 对象的生存期（正如我刚才解释的那样），另一方面，它缓存 RTTI 信息，而通过搜索来重新创建它则非常昂贵。这就是为什么您可能想长时间保持对 TRttiContext 记录的引用，从而使您能够继续访问其拥有的 RTTI 对象而不必重新创建它们（同样是耗时的操作）。

在内部，TRttiContext 记录使用类型为 TRttiPool 的全局池，该池使用关键节来使其访问线程安全。

RTTI 池机制的线程安全性例外，在 Rtti 单元本身的可用注释中有详细说明。

因此，更准确地说，RTTI 池在 TRttiContext 记录之间共享，因此在至少一个 TRttiContext 记录在内存中的同时，池中保留了 RTTI 对象。引用单元中的评论：

{...使用 RTTI 对象而没有至少一个上下文处于活动状态是一个错误。使至少一个上下文保持活动状态应使 Pool 变量保持有效}

换句话说，发布 RTTI 上下文后，您必须避免缓存和保留 RTTI 对象。这是导致内存访问冲突的示例（也是 TypesList 应用程序项目的一部分）：

```
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;
```

总而言之，RTTI 对象是由上下文管理的，您不应该保留它们。上下文又是一条记录，因此会自动处理。

您可能会看到以下列方式使用 TRttiContext 的代码：

```
AContext := TRttiContext.Create;
try
  // use the context（使用上下文）
finally
  AContext.Free;
end;
```

伪构造函数和伪析构函数设置内部接口来管理池机制，该内部接口管理在后台使用的实际数据结构。但是，由于此操作对于诸如记录之类的本地类型是自动的，因此不需要此操作，除非您在某处使用指针引用了上下文记录。

16.2.3 显示类信息

您可能希望在运行时检查的最相关类型是所谓的结构化类型，即实例，接口和记录。关注实例，通过遵循可用于实例类型的 BaseType 信息，我们可以引用类之间的关系。

访问类型当然是一个有趣的起点，但是与之相关且特别新颖的是能够了解这些类型（包括其成员）的更多详细信息。当您单击一种类型（这里是 TPopup 组件类）时，程序将在选项卡控件的三页中显示该类型的属性，方法和字段的列表，如图 16.2 所示。

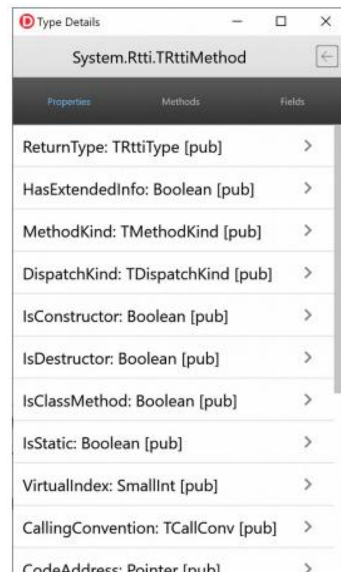

图 16.2：TypesList 应用程序项目为类 TRttiMethod 显示的详细类型信息

此第二个表单的单元可能会进行调整和扩展，以用作其他应用程序中的通用类型浏览器，它具有一个称为 ShowTypeInformation 的方法，该方法遍历给定类型的每个属性，方法和字段，并将它们添加到三个带有可见性指示的单独列表框（由 pri 表示私有，pro 表示受保护，pub 表示公共，pbl 表示公开，由 VisibilityToken 函数中的简单 case 语句返回）：

```
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;
```

您可以继续从这些属性的类型中提取更多信息，获取方法的参数列表并检查返回类型，等等。在这里，我不想构建完整的 RTTI 浏览器，而只是让您对可以实现的感觉。

16.2.4 RTTI for Packages（程序包）

记录 TRttiContext 除了可用于访问类型或类型列表的方法外，还有另一个非常有趣的方法 GetPackages，该方法返回当前应用程序使用的运行时程序包的列表。如果在没有运行时程序包编译的应用程序中执行此方法，则得到的只是可执行文件本身。但是，如果您在使用运行时程序包编译的应用程序中执行该程序，则会获得这些程序包的列表。从那时起，您可以深入研究每个软件包提供的类型。请注意，在这种情况下，类型列表会更长，因为智能链接程序不会删除应用程序未使用的 RTL 和可视库类型。

如果您使用运行时程序包，则还可以使用以下代码来检索每个程序包的类型列表（以及可执行文件本身）：

```
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;
```

❖ 如我们在第 11 章中所见，Object Pascal 中的包可用于将组件添加到开发环境中。但是，包也可以在运行时使用，通过几个运行时包来部署主可执行文件，而不是单个较大的可执行文件。如果您熟悉 Windows 开发，则程序包的作用类似于 DLL（它们在技术上是 DLL），或者更精确地说是.NET 程序集。尽管程序包在 Windows 上起着非常重要的作用，但当前在移动平台上不受支持

（这也是由于操作系统应用程序部署的限制，例如在 iOS 中）。

16.3 TValue 结构

新的扩展 RTTI 不仅使您可以浏览程序的内部结构，而且还提供特定的信息，包括属性和字段值。尽管 TypInfo 单元提供了 GetPropValue 函数来访问泛型属性并使用其值检索变量类型，但新的 Rtti 单元使用另一种结构来保存未键入的元素 TValue 记录。

该记录可以存储几乎所有可能的 Object Pascal 数据类型，并通过跟踪原始数据表示，保存数据及其数据类型来实现。它所能做的就是以给定的格式读写数据。如果您将一个 Integer 写入 TValue，则只能从中读取一个 Integer。如果您写一个字符串，则可以读回该字符串。

它不能做的是将一种格式转换为另一种格式。因此，即使 TValue 具有 AsString 和 AsInteger 方法，也只能在表示数据确实是字符串的情况下使用前者，而仅在最初为其分配整数的情况下才可以使用前者。例如，在这种情况下，可以使用 AsInteger 方法，如果调用 IsOrdinal 方法，它将返回 True：

```
var
  V1: TValue;
begin
  V1 := 100;
  if V1.IsOrdinal then
    Log (IntToStr (V1.AsInteger));
```

但是，您不能使用 AsString 方法，这将引发无效的类型转换异常：

```
var
  V1: TValue;
begin
  V1 := 100;
  Log (V1.AsString);
```

但是，如果需要字符串表示形式，则可以使用 ToString 方法，该方法具有一个大写的语句，试图容纳大多数数据类型：

```
var
  V1: TValue;
begin
  V1 := 100;
  Log (V1.ToString);
```

通过阅读曾在 RTTI 工作的 Embarcadero 研发团队前成员 Barry Kelly 的话，您可能会更好地理解：

TValue 是一种类型，用于在与基于 RTTI 的方法调用之间进行值编组以及对字段和属性的读写。

它有点像 Variant，但更适合于 Object Pascal 类型系统。例如，实例可以直接存储，也可以直接存储集，类引用等。它的类型也更严格，并且不执行（例如）静默字符串到数字的转换。

现在，您更好地了解了它的作用，让我们看一下 TValue 记录的实际功能。它具有一组用于分配和提取实际值的高级方法，以及一组基于低级指针的方法。我将专注于第一组。为了分配值，TValue 定义了几个隐式运算符，使您可以像上面的代码片段中那样执行直接分配：

```
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;
```

所有这些运算符所做的就是调用 From 泛型类方法：

```
class function From<T>(const Value: T): TValue; static;
```

当调用这些类函数时，您需要指定数据类型并传递该类型的值，例如以下代码替换了先前代码段的值 100 的分配：

```
V1 := TValue.From<Integer>(100);
```

这是一种用于将任何数据类型移入 TValue 的通用技术。分配数据后，可以使用多种方法测试其类型：

```
property Kind: TTypeKind read GetTypeKind;
function IsObject: Boolean;
function IsClass: Boolean;
function IsOrdinal: Boolean;
function IsType<T>: Boolean; overload;
function IsArray: Boolean;
```

注意，通用的 IsType 几乎可以用于任何数据类型。

有相应的数据提取方法，但是您再次只能使用与 TValue 中存储的实际数据兼容的方法，因为不会进行任何转换：

```
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;
```

在不兼容的数据类型的情况下，这些方法中的某些方法会与返回 False 的 Try 版本一起使用，而不是引发异常。还有一些有限的转换方法，其中最相关的是我已经在代码中使用过的通用 Cast 和 ToString 函数：

```
function Cast<T>: TValue; overload;
function ToString: string;
```

16.3.1 使用 TValue 读取属性

TValue 的重要性在于，这是使用扩展的 RTTI 和 Rtti 单元访问属性和字段值时使用的结构。作为使用 TValue 的实际示例，我们可以使用此记录类型来访问 TButton 对象的已发布属性和私有字段，如以下代码（RttiAccess 应用程序项目的一部分）所示：

```
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;
```

16.3.2 Invoking（调用）方法

新的扩展 RTTI 不仅使您可以访问值和字段，而且还提供了一种简化的方法调用方法。在这种情况下，您必须为方法的每个参数定义一个 TValue 元素。您可以调用一个全局 Invoke 函数来执行一个方法：

```
function Invoke(CodeAddress: Pointer; const Args: TArray<TValue>;
  CallingConvention: TCallConv; AResultType: PTypeInfo): TValue;
```

作为更好的选择，TRttiMethod 类中有一个简化的 Invoke 重载方法：

```
function Invoke(Instance: TObject;
  const Args: array of TValue): TValue; overload;
```

使用第二种简化形式的调用方法的两个示例（一个返回值，第二个要求一个参数）是 RttiAccess 应用程序项目的一部分，并在下面列出：

```
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;
```

16.4 使用属性

本章的第一部分使您对 Object Pascal 编译器生成的扩展 RTTI 以及新的 Rtti 单元引入的 RTTI 访问功能有了很好的了解。在本章的第二部分中，我们最终可以集中讨论引入整个体系结构的关键原因之一：定义自定义属性和以特定方式扩展编译器生成的 RTTI 的可能性。我们将从一个相当抽象的角度看待这项技术，然后通过查看实际示例，重点说明这是 Object Pascal 迈出的重要一步的原因。

16.4.1 什么是属性？

属性（用 Object Pascal 或 C＃术语表示）或注释（用 Java 行话表示）是注释

或指示，您可以将其添加到源代码中，并将其应用于类型，字段，方法或属性），并且编译器将嵌入程序中。通常用方括号表示，如下所示：

```
type
  [MyAttribute]
  TMyClass = class
```

通过在开发工具的设计时或最终应用程序的运行时读取此信息，程序可以根据找到的值来更改其行为。

通常，属性不用于更改对象类的实际核心功能，而是让这些类指定它们可以参与的其他机制。将类声明为可序列化不会以任何方式影响其代码，但可以使序列化代码知道它可以在该类上以及如何对其进行操作（以防您提供进一步的信息以及该属性或标记类字段或属性的其他属性）。

这正是在 Object Pascal 中使用原始且有限的 RTTI 的方式。

标记为已发布的属性可以显示在对象检查器中，流式传输到 DFM 文件，然后在运行时进行访问。属性使这种机制变得更加灵活和强大。它们和任何强大的语言功能一样，使用起来更加复杂且易于滥用。我的意思是，不要抛弃您对面向对象编程所了解的所有美好知识来拥抱这种新模型，而要相互补充。

例如，员工类别仍将在层次结构中表示为从人员类别派生的类别。员工对象仍将具有其徽章的 ID；但是您可以将职员类“标记”或“注释”为可以映射到数据库表或由特定的运行时表单显示的类。因此，我们在设计应用程序时可以使用继承（is-a），所有权（has-a）和注释（标记为-as）这三种独立的机制。

在您了解了 Object Pascal 中支持自定义属性的编译器功能并查看了一些实际示例之后，我刚才提到的抽象概念应该变得更加容易理解，或者至少这是我的希望！

16.4.2 属性类和属性声明

如何定义新的属性类（或属性类别）？您必须继承 System 单元中可用的新 TCustomAttribute 类：

```
type
  SimpleAttribute = class(TCustomAttribute)

  end;
```

您为属性类提供的类名称将成为在源代码中使用的符号，并且可以选择不包括属性后缀。因此，如果您为类 SimpleAttribute 命名，则可以在代码中使用一个名为 Simple 或 SimpleAttribute 的属性。因此，在属性情况下通常不使用 Object Pascal 类的经典初始 T。

现在，我们已经定义了一个自定义属性，可以将其应用于程序的大多数符号：类型，方法，属性，字段和参数。应用属性的语法是方括号内的属性名称：

```
type
  [Simple]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
```

在这种情况下，我已将 Simple 属性应用于整个类和方法。

除了名称，属性还可以支持一个或多个参数。传递给属性的参数必须与属性类的构造函数中指定的参数匹配（如果有）。

```
type
  ValueAttribute = class(TCustomAttribute)
  private
    FValue: Integer;
  public
    constructor Create(N: Integer);
    property Value: Integer read FValue;
  end;
```

这是通过一个参数应用此属性的方法：

```
type
  [Value(22)]
  TMyClass = class(TObject)
  public
    [Value(0)]
    procedure Two;
```

传递给其构造函数的属性值必须是常量表达式，因为它们是在编译时解析的。这就是为什么您仅限于几种数据类型的原因：序数值，字符串，集合和类引用。从积极的方面来说，您可以有多个具有不同参数的重载构造函数。

注意，您可以将多个属性应用于同一个符号，就像我在 RttiAttrib 应用程序项目中所做的那样，该项目概述了本节的代码片段：

```
type
  [Simple][Value(22)]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
    [Value(0)]
    procedure Two;
  end;
```

如果您尝试使用未定义的属性怎么办（可能是由于使用了 uses 语句缺失）？不幸的是，您收到一个非常误导性的警告消息：

[DCC Warning] RttiAttribMainForm.pas(44): W1025 Unsupported language feature: 'custom attribute'

（[DCC 警告] RttiAttribMainForm.pas（44）：W1025 不支持的语言功能：“自定义属性”）

这实际上是一个警告，表示该属性将被忽略，因此您必须提防那些警告，甚至更好地将“不受支持的语言功能”警告视为错误（可以在项目的“提示和警告”页面中执行此操作）选项对话框）：

[DCC Error] RttiAttribMainForm.pas(38): E1025 Unsupported language feature: 'custom attribute'

（[DCC 错误] RttiAttribMainForm.pas（38）：E1025 不支持的语言功能：“自定义属性”）

最后，与具有相同概念的其他实现相比，当前没有方法来限制属性的范围，例如声明属性可以应用于类型而不是方法。相反，编辑器中提供的功能完全支持重命名重构中的属性。不仅可以更改属性类的名称，而且当以全名使用属性并且不使用最后的“属性”部分时，系统都会启动。

马尔科姆·格罗夫斯（Malcolm Groves）在他的博客中首先提到了属性重构：http://www.malcolmgroves.com/blog/?p=554

16.4.3 Browsing Attributes（浏览属性）

现在，如果没有一种方法可以发现定义了哪些属性，并且可能由于这些属性而向对象注入不同的行为，那么这段代码似乎就完全没有用了。让我开始关注第一部分。Rtti 单元的类可让您确定哪些符号具有关联的属性。这是从 RttiAttrib 应用程序项目中提取的代码，显示了当前类的属性列表：

```
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);
```

运行此代码将显示出：

```
SimpleAttribute
ValueAttribute
```

您可以通过将以下代码添加到 for-in 循环代码中来扩展它，以提取给定属性类型的特定值：

```
if Attrib is ValueAttribute then
  Form39.Show (' -' + IntToStr(ValueAttribute(Attrib).Value));
```

如何获取具有给定属性或任何属性的方法？您不能预先筛选方法，而必须遍历每个方法，检查其属性，然后查看它们是否与您相关。为了帮助完成此过程，我编写了一个函数来检查某个方法是否支持给定的属性：

```
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;
```

在检查给定属性或其中任何一个属性时，RttiAttrib 程序将调用 HasAttribute 函数：

```
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);
```

效果是列出标记有给定属性的方法，如进一步的 Log 调用所述，我从上面的清单中省略了这些调用：

```
Methods marked with [Simple] attribute
One
Methods marked with any attribute
One
Two
```

除了简单地描述属性外，通常要做的是添加一些由类的属性而不是其实际代码确定的独立行为。

例如，我可以在前面的代码中注入特定的行为：目标可以是调用标记有给定属性的类的所有方法，并将它们视为无参数方法：

```
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;
```

此代码段的作用是创建一个对象，获取其类型，检查给定的属性，然后调用每个具有 Simple 属性的方法。除了从基类继承，实现接口或编写特定的代码来执行请求之外，我们要做的就是获得具有给定属性的其他方法之一，以实现新的行为。并不是说这个示例使属性的使用非常明显：对于使用属性的一些常见模式和一些实际案例研究，您可以参考本章的最后部分。

16.5 虚拟方法 Interceptors（拦截器）

本节介绍了 Object Pascal 的一项非常高级的功能，如果您刚刚开始学习 Delphi 的语言，则可能要跳过阅读它。它适用于更多的专业读者。

在引入扩展的 RTTI 之后，还添加了另一个相关功能，它具有通过为现有对象创建代理类来拦截现有类的虚拟方法的执行的功能。换句话说，您可以采用一个现有对象并更改其虚拟方法（一次特定的一个或全部）。

你为什么想做这个？在标准的 Object Pascal 应用程序中，您可能不会使用此功能。如果您需要一个行为不同的对象，只需对其进行更改或创建一个子类。库的情况有所不同，因为库应该以非常通用的方式编写，对它们将能够操纵的对象了解得很少，并且对对象本身的负担也要尽可能少。这是将虚拟方法拦截器添加到对象 Pascal 的一种情况。

❖ 巴里·凯利（Barry Kelly）关于虚拟方法拦截器(我欠很多钱)的非常详细的博客

文章可在 http://blog.barrkel.com/2010/09/virtual-method-interception.html 上找到。

在我们关注可能的场景之前，让我先讨论一下这项技术。假设您有一个至少具有虚拟方法的现有类，如下所示：

```
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;
```

现在您可以做的是创建一个 TVirtualMethodInterceptor 对象（在 RTTI 单元中定义的新类），该对象绑定到该对象的类子类，将对象的静态类更改为动态类：

```
var
  Vmi: TVirtualMethodInterceptor;
begin
  Vmi := TVirtualMethodInterceptor.Create(TPerson);
  Vmi.Proxify(Person1);
```

一旦有了 vmi 对象，就可以使用匿名方法为其事件安装特殊的处理程序（OnBefore，OnAfter 和 OnException）。这些将在任何虚拟方法调用之前，任何虚拟方法调用之后以及在虚拟方法发生异常的情况下触发。这些是三种匿名方法类型的签名：

```
  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，您都会获得对象，方法引用，参数和结果（可能已设置或未设置）。在 OnBefore 事件中，您可以设置 DoInvoke 参数以禁用标准执行。在 OnExcept 事件中，您可以获得有关异常的信息。

在使用上面的 TPerson 类的 InterceptBaseClass 应用程序项目中，我使用以下日志代码拦截了类虚拟方法：

```
procedure TFormIntercept.BtnInterceptClick(Sender: TObject);
begin
  Vmi := TVirtualMethodInterceptor.Create(TPerson);
```

```
    Vmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
      const Args: TArray<TValue>; out DoInvoke: Boolean;
      out Result: TValue)
      begin
        Show('Before calling ' + Method.Name);
      end;
    Vmi.OnAfter := procedure(Instance: TObject; Method: TRttiMethod;
      const Args: TArray<TValue>; var Result: TValue)
      begin
        Show('After calling ' + Method.Name);
      end;
    Vmi.Proxify(Person1);
  end;
```

请注意，至少要在使用 Person1 对象之前保留 vmi 对象，否则您将使用不再可用的动态类，并且将调用已发布的匿名方法。在演示中，我将其保存为表单字段，就像它所引用的对象一样。

程序通过调用其方法并检查基类名称来使用该对象：

```
    Show ('Age: ' + IntToStr (Person1.Age));
    Show ('Person: ' + Person1.ToString);
    Show ('Class: ' + Person1.ClassName);
    Show ('Base Class: ' + Person1.ClassParent.ClassName);
```

在安装拦截器之前，输出为：

```
Age: 26
Person: Mark is 26 years old
Class: TPerson
Base Class: TObject
```

安装拦截器后，输出将变为：

```
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，该类具有与基类相同的名称，但实际上是一个不同的类，即由“虚拟方法拦截器”创建的动态类。尽管没有官方的方法可以将目标对象的类还原为原始对象，但是该类本身可以在“虚拟方法拦截器”对象中使用，也可以作为对象的基类使用。尽管如此，您仍可以使用蛮力将正确的类引用分配给对象的类数据（其初始四个字节）：

```
    PPointer(Person1)^ := Vmi.OriginalClass;
```

再举一个例子，我修改了 OnBefore 代码，以便在您调用 Age 的情况下，它返回给定值，并跳过执行实际方法的步骤：

```
  Vmi.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;
```

输出与上述版本相比有以下变化（请注意，Age 调用和相对的 OnAfter 事件将被跳过）：

```
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
```

既然我们已经了解了虚拟方法拦截器背后的技术细节，我们可以回头弄清楚您要在哪些方案中使用此功能。同样，在标准应用程序中，我们基本上没有理由这样做。相反，重点主要是针对那些开发高级库并需要实现自定义行为以测试或处理对象的人员。

例如，这将有助于构建单元测试库，尽管它仅限于虚拟方法。您还可以将其与自定义属性一起使用，以实现类似于面向方面的编程的编码样式。

16.6 RTTI 案例研究

现在，我已经介绍了 RTTI 的基础和属性的使用，现在有必要研究一些实际情况，在这些情况下使用这些技术将非常有用。在许多情况下，更灵活的 RTTI 和通过属性自定义它的能力是相关的，但是我没有机会容纳一长串情况。相反，我将指导您逐步开发两个简单但重要的示例。

第一个演示程序将展示如何使用属性来标识课程中的特定信息。特别是，我们希望能够检查声明为体系结构一部分的类的对象，并具有描述和引用该对象本身的唯一 ID。这在某些情况下可能会派上用场，例如描述存储在集合中的对象（通用或传统对象）。

第二个演示将是流式传输的示例，特别是将类流式传输到 XML 文件。我将从使用已发布的 RTTI 的经典方法开始，移至新的扩展 RTTI，最后说明如何使用属性来自定义代码并使其更加灵活。

16.6.1 ID 和说明的属性

如果要在许多对象之间共享几个方法，最经典的方法是使用虚拟方法定义基类，并从基类继承各种对象，从而覆盖虚拟方法。很好，但是由于您有固定的基类，因此在可以参与体系结构的类方面存在很多限制。

克服这种情况的一种标准技术是使用接口而不是通用的基类。实现接口的多个类（但没有公共的祖先类）可以提供接口方法的实现，该方法的作用与虚拟方法非常相似。

完全不同的样式（既有优点也有缺点）是使用属性来标记参与的类和给定的方法（或属性）。这提供了更大的灵活性，不涉及接口，但是基于相对缓慢且易于出错的运行时信息查找，而不是基于编译时解析。这意味着我不主张在接口上使用这种编码样式作为一种更好的方法，而只是在某些情况下可能值得评估和使用。

描述属性类

对于此演示，我已经定义了一个属性，该设置具有指示该元素将被应用于的设置。我本可以使用三个不同的属性，但更希望避免污染属性名称空间。这是属性类定义：

```
type
  TDescriptionAttrKind = (dakClass, dakDescription, dakId);

  DescriptionAttribute = class (TCustomAttribute)
  private
    FDak: TDescriptionAttrKind;
  public
    constructor Create (ADak: TDescriptionAttrKind = dakClass);
    property Kind: TDescriptionAttrKind read FDak;
  end;
```

请注意，构造函数的唯一参数使用默认值，以使您可以使用不带参数的属性。

样本类

接下来，我编写了两个使用该属性的示例类。每个类都标记有该属性，并具有两个标记有相同属性的方法，并用不同的种类对其进行了自定义。第一个（TPerson）具有映射到 GetName 函数的描述，并使用其 TObject.GetHashCode 方法提供（临时）ID，重新声明了将属性应用于其的方法（方法代码只是对继承版本的调用）：

```
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;
```

第二类（TCompany）甚至更简单，因为它具有自己的 ID 和说明值：

```
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;
```

即使两个类之间存在相似性，它们在层次结构，公共接口或类似方面也是完全无关的。他们共享的是相同属性的使用。

示例项目和属性导航

该属性的共享用法用于显示有关添加到列表的对象的信息，这些信息以程序的主要形式声明为：

```
FObjectsList: TObjectList<TObject>;
```

在程序启动时，将创建并初始化此列表：

```
procedure TFormDescrAttr.FormCreate(Sender: TObject);
var
    APerson: TPerson;
    ACompany: TCompany;
begin
    FObjectsList := TObjectList<TObject>.Create;
    // 加一个人
    APerson := TPerson.Create;
    APerson.Name := 'Wiley';
    APerson.Country := 'Desert';
    APerson.BirthDate := Date - 1000;
    fObjectsList.Add(APerson);
    // 添加一个公司
    ACompany := TCompany.Create;
    ACompany.Name := 'ACME Inc.';
    ACompany.ID := IntToStr (GetTickCount);
    ACompany.Country := 'Worldwide';
    FObjectsList.Add(ACompany);
    // 添加不相关的对象
    FObjectsList.Add(TStringList.Create);
```

为了显示有关对象的信息（即 ID 和描述，如果有的话），程序使用通过 RTTI 进行属性发现。首先，它使用一个 helper 助手函数来确定该类是否标记有特定的属性：

```
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;
```

- 在这种情况下，您需要检查完整的类名，DescriptionAttribute，而不仅仅是“Description”，Description 是在应用属性时可以使用的符号。

如果是这种情况，程序将通过嵌套循环获取每个方法的每个属性，并检查这

是否是我们正在寻找的属性，从而继续进行：

```
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，调用带有属性的方法以两个临时字符串（后来添加到用户界面）中读取结果：

```
if Attrib is DescriptionAttribute then
  case DescriptionAttribute(Attrib).Kind of
    dakClass: ; // ignore
    dakDescription:
      strDescr := AMethod.Invoke(AnObject, []).ToString;
    dakId:
      strID := AMethod.Invoke(AnObject, []).ToString;
```

程序无法执行的操作是检查属性是否重复（也就是说，如果有多个方法标记有相同的属性，则可能要引发异常）。

总结上一页的所有代码片段，这是 UpdateList 方法的完整代码：

```
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;
      // 寻找属性完成
      // 应该检查我们是否发现了任何东西
      with ListView1.Items.Add do
      begin
        Text := STypeName;
        Detail := StrDescr;
      end;
    end;
  end;
  // 否则忽略对象，可能引发异常
```

```
end;
```

如果该程序产生的输出不太有趣，那么它的完成方式就很有意义，因为我已经用属性标记了一些类和这些类的两个方法，并且能够使用外部算法来处理这些类。

换句话说，这些类本身不需要特定的基类，接口实现或任何内部代码都可以成为体系结构的一部分，而只需要声明它们想通过使用属性来参与。管理类的全部责任在某些外部代码中。

16.6.2 XML Streaming（XML 流）

使用 RTTI 的一个有趣且非常有用的情况是创建对象的“external（外部）”图像，以将其状态保存到文件中或通过电线将其发送到另一个应用程序。传统上，用于解决此问题的“Object Pascal”方法一直是流式传输对象的已发布属性，与创建 DFM 文件时使用的方法相同。现在，RTTI 允许您保存对象的实际数据，其字段而不是外部接口。尽管它会导致额外的复杂性，例如在内部对象数据的管理中，但它更强大。

同样，该演示只是该技术的简单展示，并且没有深入探讨其所有含义。

此示例有 3 个版本，为简单起见，在一个项目中进行了编译。

第一种是基于已发布属性的传统 Object Pascal 方法，第二种使用扩展的 RTTI 和字段，第三种使用属性来自定义数据映射。

普通的 XML Writer 类

为了帮助生成 XML，我将 XmlPersist 应用程序项目基于我最初在 Delphi 2009 Handbook 中编写的 TTrivialXmlWriter 类的扩展版本上，以演示 TTextWriter 类的用法。在这里，我不想再次讨论。可以说，由于有了字符串堆栈，该类可以跟踪它打开的 XML 节点，并以 LIFO（后进先出）顺序关闭 XML 节点。

❖ 可在 http://github.com/MarcoDelphiBooks/Delphi2009Handbook/tree/master/07/ReaderWriter 中找到《Delphi 2009 手册》的 TTrivialXmlWriter 类的源代码。

在本节中，我基于本节将要探讨的三种不同方法，添加了一些有限的格式化代码和三种保存对象的方法。这是完整的类声明：

```
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 (Indent: Boolean = False);
    procedure WriteString (const SValue: string);
    procedure WriteObjectPublished (AnObj: TObject);
    procedure WriteObjectRtti (AnObj: TObject);
    procedure WriteObjectAttrib (AnObj: TObject);
    function Indentation: string;
  end;
```

为了了解代码，这是 WriteStartElement 方法，该方法使用 Indentation 函数在

内部堆栈中保留的空间是当前节点数的两倍：

```
procedure TTrivialXmlWriter.WriteStartElement(const SName: string);
begin
  FWriter.Write (Indentation + '<' + SName + '>');
  FNodes.Push (SName);
end;
```

您将在项目源代码中找到该类的完整代码。

基于经典 RTTI 的流

在介绍了支持类的介绍之后，让我从头开始，那就是使用经典的 RTTI 将对象保存为基于 XML 格式的已发布属性。

WriteObjectPublished 方法的代码非常复杂，需要一些解释。它基于 TypInfo 单元，并使用旧版 RTTI 的低级版本来获取给定对象（AnObj 参数）的已发布属性列表，其代码如下：

```
NProps := GetTypeData(AnObj.ClassInfo)^.PropCount;
GetMem(PropList, NProps * SizeOf(Pointer));
GetPropInfos(AnObj.ClassInfo, PropList);
for I := 0 to NProps - 1 do
  ...
```

这是在要求属性的数量，分配适当大小的数据结构，并用有关已发布属性的信息填充数据结构。如果您想知道是否可以编写此底层代码？好吧，您刚刚找到了引入新 RTTI 的很好理由。对于每个属性，程序都会提取数字和字符串类型的属性值，而它会提取任何子对象并对其进行递归操作：

```
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;
```

还有一些额外的复杂性，但是为了举例说明并让您对传统方法有所了解，这应该足够了。

为了演示该程序的效果，我编写了上一个示例改编的两个类（TCompany 和 TPerson）。但是，这一次，公司可以安排一个人来分配额外的资产，称为老板。在现实世界中，这将更为复杂，但是对于本示例来说，这是一个合理的假设。 这些是两个类的已发布属性：

```
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;
```

该程序的主要形式有一个按钮，用于创建和连接这两个类的两个对象，并将它们保存到 XML 流中，稍后将显示该流。

流式传输部分具有以下代码：

```
SS := TStringStream.Create;
XmlWri := TTrivialXmlWriter.Create (SS);
XmlWri.WriteStartElement('company');
XmlWri.WriteObjectPublished(aCompany);
XmlWri.WriteEndElement;
```

结果是一个 XML 文件，如：

```
<company>
  <Name>ACME Inc.</Name>
  <Country>Worldwide</Country>
  <ID>29088851</ID>
  <Boss>
    <Name>Wiley</Name>
    <Country>Desert</Country>
  </Boss>
</company>
```

具有扩展 RTTI 的流字段

有了 Object pascal 中可用的高级 RTTI，我可以将这个旧程序转换为使用扩展的 RTTI 来访问已发布的属性。相反，我要做的是使用它来保存对象的内部表示形式，即对象的私有数据字段。我不仅在做一些更严格的事情，而且还在用更高级的代码来做。WriteObjectRtti 方法的完整代码如下：

```
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;
```

生成的 XML 有点类似，但是由于字段名通常比属性名可读性差，因此某种程度上不太干净：

```
<company>
  <FName>ACME Inc.</FName>
  <FCountry>Worldwide</FCountry>
  <FID>29470148</FID>
  <FPerson>
    <FName>Wiley</FName>
    <FCountry>Desert</FCountry>
  </FPerson>
</company>
```

但是，另一个很大的区别是，在这种情况下，这些类不需要从 TPersistent 类继承，也不需要使用任何特殊选项进行编译。

使用属性来自定义流

除了标签名称的问题之外，还有另一个我没有提到的问题。

使用 XML 标记名称（实际上是已编译的符号）不是一个好主意。

另外，在代码中，无法从 XMLbase 流中排除某些属性或字段。

❖ 可以使用存储的指令来控制对象 Pascal 属性流传输，可以使用 TypInfo 单元读取该指令。尽管如此，即使 DFM 流机制有效地使用了该解决方案，它也远非简单明了。

这些是我们可以使用属性来解决的问题，尽管缺点是必须在类的声明中大量使用它们，我不太喜欢这种编码样式。对于新版本的代码，我定义了带有可选参数的属性构造函数：

```
type
  XmlAttribute = class (TCustomAttribute)
  private
    FTag: string;
  public
    constructor Create (StrTag: string = '');
    property TagName: string read FTag;
  end;
```

基于属性的流代码是基于扩展 RTTI 的最新版本的变体。唯一的区别是，现在该程序调用 CheckXmlAttr helper 函数来验证该字段是否具有 xml 属性和（可选）标记名修饰：

```
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;
```

最相关的代码在 CheckXmlAttr 助手函数中：

```
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 属性的字段将被忽略，并且 XML 输出中使用的标记是可自定义的。为了说明这一点，该程序具有以下类（这次我从清单中省略了已发布的属性，因为它们不相关）：

```
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;
    ...
```

使用这些声明，XML 输出将如下所示（注意标记名称，ID 的事实被省略以及 FCountry 字段的（不好看的）默认名称）：

```
  <company>
    <CompanyName>ACME Inc.</CompanyName>
    <Country>Worldwide</Country>
    <TheBoss>
      <Name>Wiley</Name>
```

```
        <FCountry>Desert</FCountry>
      </TheBoss>
    </company>
```

此处的区别在于，我们可以非常灵活地确定要包括哪些字段以及如何在 XML 中命名它们，而以前的版本不允许这样做。

即使这只是一个非常简单的实现，我认为，让您有机会看到从经典 RTTI 开始逐步创建最终版本的过程，也可以使您很好地了解各种技术之间的差异。

实际上，要记住的重要一点是，使用属性将永远是最好的解决方案，这不是一个必然的想法！ 另一方面，应该清楚的是，在需要在运行时检查未知对象的结构的任何情况下，RTTI 和属性都会增加很多功能和灵活性。

16.6.3 其他基于 RTTI 的库

在结束本章时，我想指出一个事实，即已经有一些库（包括产品的一部分和第三方库）开始利用扩展的 RTTI。这样的例子之一就是绑定表达式机制，它位于可视实时绑定之后。您可以创建一个绑定表达式，为它分配一个表达式（这是一个带有连接或加法等操作的文本字符串），并使该表达式引用一个外部对象及其字段。

即使我不想过多地研究这个主题，它实际上是一个特定的库，而不是语言或核心系统的一部分，但我认为简短的清单可以使您有所了解：

```
var
  BindExpr: TBindingExpression;
  Pers: TPerson;
begin
  Pers := TPerson.Create;
  Pers.Name := 'John';
  Pers.City := 'San Francisco';
  BindExpr := TBindingExpressionDefault.Create;
  BindExpr.Source := 'person.name + '' lives in '' + person.city');
  BindExpr.Compile([TBindingAssociation.Create(Pers, 'person')]);
  Show (BindExpr.Evaluate.GetValue.ToString);
  Pers.Free;
  BindExpr.Free;
end;
```

请注意，这里的优势来自于您可以在运行时更改表达式的事实（尽管在上面的特定代码段中，它是一个常量字符串）。该表达式可以来自编辑，也可以从多个可能的表达式中动态选择。首先将其分配给 TBindingExpression 对象，然后在运行时通过 Compile 调用对其进行分析和编译（将其转换为符号形式，而不是汇编代码）。执行访问 TPerson 对象时，它将使用 RTTI。

缺点是这种方法使表达式求值比类似的已编译 Object Pascal 源代码的执行慢得多。换句话说，您必须在性能下降与灵活性之间取得平衡。在此基础上构建的 Visual Live Binding 模型也为开发人员带来了非常轻松愉快的体验。

第十七章 TObject 和 system 单元

任何 Object Pascal 语言应用程序的核心都是类的层次结构。

系统中的每个类最终都是 TObject 类的子类，因此整个层次结构只有一个根。这使您可以使用 TObject 数据类型替代系统中任何类类型的数据类型。

TObject 类是在称为 System 的核心 RTL 单元中定义的，它具有如此重要的作用，因此它会自动包含在每个编译中。虽然我不会介绍所有系统单元类和其他系统单元功能，但是有一些值得研究的地方，TObject 当然是最重要的一个。

❖ 如果像 TObject 这样的核心系统类是该语言的一部分，还是它是运行时库（RTL）的一部分，则可能会进行长时间的辩论。System 单元的其他功能也是如此，它非常重要，以至于它会自动包含在任何其他单元的编译中。（将其添加到 uses 声明中实际上是非法的。）但是，这样的辩论将是徒劳的，因此我将其搁置一段时间。

17.1 TObject 类

正如我刚才提到的，TObject 类是一个非常特殊的类，因为所有其他类都继承自它。实际上，当您声明一个新类时，如果您未指定基类，则该类将自动从 TObject 继承。用编程语言术语来说，这种情况称为单根类层次结构，与 C＃，Java 和许多其他现代编程语言共享的对象 Object Pascal。值得注意的例外是 C ++，它没有单个基类的概念，并且允许您定义多个完全独立的类层次结构。

此基类不是您可以通过创建其实例直接使用的类。但是，这是个类，您将很容易使用很多类。每当需要一个可以容纳任何类型对象的变量时，就将其声明为 TObject 类型。这种用法的一个很好的例子是在组件库事件处理程序中，该事件处理程序通常将 TObject 作为第一个参数的类型，通常称为 Sender。这意味着任何实际类的任何对象。许多泛型集合也是对象的集合，并且在几种情况下可以直接使用 TObject 类型。在以下各节中，我将介绍该类的一些功能，这些功能可用于系统中的所有类。

17.1.1 构造函数和析构函数

尽管直接创建 TObject 几乎没有意义，但是此类的构造函数和析构函数很重要，因为它们会被所有其他类自动继承。如果定义一个没有构造函数的类，您仍然可以在其上调用 Create，以调用 TObject 构造函数，该方法是一个空方法（因为在此基类中没有要初始化的东西）。此 Create 构造函数是非虚拟的，除非您完全不做任何事情，否则您将在类中完全替换它。

即使直接调用 TObject.Create 并不是特别有用，调用基类构造函数对于任何子类都是一个好习惯。

❖ 我强调这是一个非虚拟的构造函数，因为还有另一个核心库类 TComponent 定义了一个虚拟的构造函数。TComponent 类虚拟构造函数在流系统中起着关键作用，下一章将进行介绍。

为了销毁对象，TObject 类具有 Free 方法（该方法最终称为 Destroy 析构函数）。在第 13 章中，我已经详细介绍了这些内容，以及有关正确使用内存的许多建议，因此在这里无需重复它们。

17.1.2 了解对象

TObject 类的一组有趣的方法是那些返回有关类型的信息的方法。最常用的是 ClassType 和 ClassName 方法。ClassName 方法返回带有类名称的字符串。

因为它是一个类方法（就像大量的 TObject 类方法一样），所以您可以将其应用于对象和类。假设您已经定义了一个 TButton 类和该类的 Button1 对象。然后，以下语句具有相同的效果：

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

当然，您也可以将其应用于通用 TObject，但不会获得 TObject 信息，而会获得有关当前分配给变量的对象的特定类的信息。例如，在按钮的 OnClick 事件处理程序中，调用：

```
Text := Sender.ClassName;
```

可能会返回与上述相同的行，即字符串“ TButton”。这是因为类名称是在运行时（由特定对象本身）而不是由编译器（仅将其视为 TObject 对象）确定的。

虽然获取类名通常可以方便地调试，记录和显示类信息，但访问类的类引用通常更为重要。例如，最好将两个类引用进行比较，而不要与具有类名称的字符串进行比较。我们可以使用 ClassType 方法获取类引用，而 ClassParent 方法则返回对当前类的基类的类引用，从而可以导航至基类列表。唯一的例外是该方法为 TObject 返回 nil（因为它没有父类）。一旦有了类引用，就可以使用它来调用任何类方法，包括 ClassName 方法。

返回有关类的信息的另一个非常有趣的方法是 InstanceSize，它返回对象的运行时大小，即对象的字段（以及从基类继承的字段）所需的内存量。当系统需要分配该类的新实例时，此功能在内部使用。

尽管您可能认为 SizeOf 全局函数也提供了此信息，但是该函数实际上返回的是对象引用的大小（取决于目标平台，指针通常为四个或八个字节），而不是对象本身的大小。

17.1.3 TObject 类的更多方法

TObject 类的其他方法可以应用于任何对象（也可以应用于任何类或类引用，因为它们是类方法）。这是部分列表，并有简短描述：

- ClassName 返回带有类名称的字符串，以进行显示。
- ClassNameIs 根据值检查类名称。
- ClassParent 返回对当前类或对象类的父类的类引用。 您可以从 ClassParent 导航到 ClassParent，直到到达 TObject 类本身，该方法在其中返回 nil。
- ClassInfo 返回一个指向类的内部，底层运行时类型信息（RTTI）的指针。 这在 TypInfo 单元的早期就已使用，但现在已被 RTTI 单元的功能所替代，如第 16 章所述。在内部，这仍然是获取类 RTTI 的方式。
- ClassType 返回对对象类的引用（不能直接应用于类，而只能应用于对象）。
- InheritsFrom 测试类是否从给定的基类继承（直接或间接）（这与 is 运算符非常相似，并且实际上是 is 运算符的最终实现方式。）。
- InstanceSize 返回以字节为单位的对象数据的大小。这是字段的总和，加上一些额外的特殊保留字节（例如，包括类引用）。再次注意，这是实例大小，

而对实例的引用仅与指针一样长（4 或 8 个字节，取决于平台）。

- UnitName 返回定义该类的单元的名称，这对于描述类可能很有用。 实际上，类名在系统中不是唯一的。正如在上一章中所看到的，在应用程序中，只有合格的类名（由单元名和类名组成，用点分隔）是唯一的。
- QualifiedClassName 返回单元名和类名的此组合，该值在正在运行的系统中确实是唯一的。

TObject 的这些方法可用于每个类的对象，因为 TObject 是每个类的公共祖先类。这是我们如何使用这些方法访问类信息的方法：

```
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));
```

该代码将检查 ClassParent 是否为 nil，以防您实际上使用的是 TObject 类型的实例，该实例没有基本类型。您可以使用其他方法执行测试。例如，您可以使用以下代码检查 Sender 对象是否为特定类型：

```
if Sender.ClassType = TButton then ...
```

您还可以通过以下测试来检查 Sender 参数是否对应于给定的对象：

```
if Sender = Button1 then...
```

通常，您无需检查特定的类或对象，而是需要测试给定类的对象的类型兼容性；也就是说，您需要检查对象的类是给定类还是其子类之一。这使您知道是否可以使用为类定义的方法对对象进行操作。

可以使用 InheritsFrom 方法完成此测试，该方法在使用 is 运算符时也会调用。以下两个测试是等效的：

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

显示类信息

有了类引用后，可以将其所有基类的列表添加到其描述（或显示信息）中。在以下代码段中，将 MyClass 的基类添加到 ListBox 控件中：

```
ListParent.Items.Clear;
while MyClass.ClassParent <> nil do
begin
  MyClass := MyClass.ClassParent;
  ListParent.Items Add (MyClass.ClassName);
end;
```

您会注意到，我们在 while 循环的心脏使用了一个类引用，该引用用于测试是否存在父类（在这种情况下，当前类为 TObject）。另外，我们可以通过以下两种方式编写 while 语句：

```
while not MyClass.ClassNameIs ('TObject') do... // 缓慢，容易出错
while MyClass <> TObject do... // 快速且可读
```

17.1.4 TObject 的虚拟方法

自 Object Pascal 语言问世以来，TObject 类的结构一直保持稳定，但在某一时刻，它看到了三个非常有用的虚方法的添加。可以像在任何其他 TObject 方法上

一样在任何对象上调用这些方法，但是相关的是，这些方法应该在您自己的类中重写并重新定义。

❖ 如果您使用过.NET 框架，您将立即认识到这些方法是 C＃基类库的 System.Object 类的一部分。类似的方法用于 Java 中可用的基类，JavaScript 和其他语言中常用的基类。其中一些的起源（例如 toString 的起源）可以追溯到 Smalltalk，Smalltalk 被认为是第一种 OOP 语言。

ToString 方法

ToString 虚拟函数是一个占位符，用于返回给定对象的文本表示形式（描述或序列化）。TObject 类中方法的默认实现返回类名称：

```
function TObject.ToString: string;
begin
  Result := ClassName;
end;
```

当然，这远远没有用。从理论上讲，每个类都应提供一种向用户描述自身的方法，例如，将对象添加到可视列表时。运行时库中的某些类会覆盖 ToString 虚拟函数，例如 TStringBuilder，TStringWriter 和 Exception 类，以返回异常列表中的消息（如第 9 章的“ InnerException 机制”一节所述）。

有一种标准的方法来返回任何对象的字符串表示形式是一个很有趣的想法，我建议您利用 TObject 类的这一核心功能，将其视为语言功能。

❖ 请注意，ToString 方法“semantically overloads（严重重载）”了 Classes 单元中定义的“ parse token String”或 toString 符号。因此，该符号被称为 Classes.toString。

Equals（等于）方法

Equals 虚拟函数是一个占位符，用于检查两个对象是否具有相同的逻辑值，这与检查两个变量是否引用内存中的同一对象的操作不同，您可以使用=运算符来实现。但是，这确实令人困惑，由于缺少更好的方法，默认实现确实做到了这一点：

```
function TObject.Equals(Obj: TObject): Boolean;
begin
  Result := Obj = Self;
end;
```

在 TStrings 类中使用此方法（带有适当的覆盖）的一个示例，其中 Equals 方法一一比较列表中的字符串数和实际字符串的内容。

泛型支持，特别是在 Generics.Default 和 Generics.Collections 单元中，是该库中显着使用此技术的部分。通常，对于库或框架而言，与对象标识分开定义对象“值等效”的概念很重要。具有“按值”比较对象的标准机制是一个很大的优势。

GetHashCode 方法

GetHashCode 虚拟函数是从.NET 框架借用的另一个占位符，用于让每个类为其对象计算哈希码。默认代码返回一个看似随机的值，即对象本身的地址：

```
function TObject.GetHashCode: Integer;
begin
  Result := Integer(self);
end;
```

❖ 由于通常从有限的堆区域集中获取要创建的对象的地址，因此这些数目的分

布不均匀，这可能会对哈希算法产生不利影响。强烈建议自定义此方法，该方法基于逻辑值创建哈希，并基于对象内部的数据（而不是其地址）具有良好的哈希分布。字典和其他数据结构依赖于哈希值，改善哈希分布可以显着提高性能。

GetHashCode 虚拟函数由支持哈希表的某些集合类使用，并用作优化某些代码的方式，例如 TDictionary <T>。

使用 TObject 虚拟方法

这是一个基于某些 TObject 虚拟方法的示例。该示例具有一个重写以下两个方法的类：

```
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;
```

在实现这三种方法时，我只需要将对 GetType 的调用更改为对 ClassType 的调用即可：

```
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;
```

请注意，如果对象具有完全相同的类并且它们的值匹配，则认为它们是相等的，而它们的字符串表示形式仅包括名称字段。该程序在开始时会创建此类的一些对象：

```
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);
  ...
```

请注意，两个引用（Ao2 和 Ao3）指向内存中的同一对象，而最后一个对象（Ao4）具有相同的数值。该程序具有一个用户界面，使用户可以选择任意两个项目并使用“等于”和直接引用比较来比较选定的对象。

以下是一些结果：

```
Comparing Ao1 and Ao4
Equals: False
Reference = False

Comparing Ao2 and Ao3
Equals: True
Reference = True

Comparing Ao3 and Ao4
Equals: True
Reference = False
```

该程序还有另一个按钮，用于测试按钮本身的某些方法：

```
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;
```

输出为以下内容（哈希值在执行时会发生变化）：

```
Equals: True
Reference = True
GetHashCode: 28253904
ToString: TButton
```

17.1.5 TObject 类摘要

总而言之，这是最新版本的编译器中 TObject 类的完整接口（省略了大部分 IFDEF 和低级重载，以及私有和受保护的部分）：

```
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;
    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 Disposed: Boolean read GetDisposed;
  end;
```

17.1.6 Unicode 和类名

重载的方法（如 MethodAddress 和 FieldAddress）可以采用 UnicodeString（通常为 UTF-16）或被视为 UTF8 字符串的 ShortString 参数。实际上，采用普通 Unicode 字符串的版本可以通过调用函数 UTF8EncodeToShortString 对其进行转换：

```
function TObject.FieldAddress(const Name: string): Pointer;
begin
  Result := FieldAddress(UTF8EncodeToShortString(Name));
end;
```

由于该语言引入了 Unicode 支持，因此 Object Pascal 中的类名称在内部使用 ShortString 表示形式（一个一字节字符的数组），但是使用 UTF-8 编码，而不是 ShortString 类型的传统 ANSI 编码。这既发生在 TObject 级别，也发生在 RTTI 级别。

例如，ClassName 方法是通过一些非常底层的代码实现的，如下所示：

```
class function TObject.ClassName: string;
begin
  Result := UTF8ToString (
    PShortString (PPointer (
      Integer(Self) + vmtClassName)^)^);
end;
```

类似地，在 TypInfo 单元中，所有访问类名称的函数都将内部 UTF-8 ShortString 表示形式转换为 UnicodeString。属性名称也会发生类似的情况。

17.2 系统单元

尽管 TObject 类显然是该语言的基本角色，但很难说出它是否是该语言或运行时库的一部分，但 System 单元中还有其他一些低级类，它们构成了基本的，集成的编译器支持的一部分。但是，该单元的大部分内容由低级数据结构，简单记录结构，函数和过程以及一些类组成。

在这里，我将主要集中在类上，但是不可否认，System 单元中的许多其他功能是该语言的关键。例如，系统单元定义了所谓的“intrinsic（内在的）”函数，这些函数没有实际代码，但直接由编译器解析。一个例子是 SizeOf，它被编译器直接替换为作为参数传递的数据结构的实际大小。

通过阅读添加到其开头的注释，您可以了解 System 单元的特殊作用（主要是解释为什么浏览系统符号会导致该单元...但不是您要查找的符号）：

{ Predefined constants, types, procedures, } {预定义的常量，类型，过程}
{ and functions (such as True, Integer, or } {和函数（例如 True，Integer 或}
{ Writeln) do not have actual declarations.} {Writeln）没有实际的声明。}
{ Instead they are built into the compiler } {相反，它们内置在编译器中}
{ and are treated as if they were declared } {并且被视为已声明}
{ at the beginning of the System unit. }{在系统单元的开头。}

阅读本单元的源代码可能会很繁琐，这也是因为在这里您可以找到整个运行时库的一些较低级别的代码。因此，我决定仅描述其内容的非常有限的选择。

17.2.1 选定的系统类型

如上所述，系统单元为不同的数字类型，其他序数类型和字符串定义核心数据类型和许多类型别名。系统在底层使用了其他核心数据类型（包括枚举，记录和强类型别名），值得一看：

- TVisibilityClasses 是用于 RTTI 可见性设置的枚举（有关更多详细信息，请参见第 16 章）。
- TGUID 是用于表示 Windows 以及其他所有受支持的操作系统上的 GUID 的记录
- TMethod 是一个核心记录，代表用于事件处理程序的结构，带有一个指向方法地址的指针和一个指向当前对象的指针（在第 10 章中简要提到）
- TMonitor 是实现 C.A.R Hoare 和 Per Brinch Hansen 发明的线程同步机制（称为“监视器”)的记录，在 Wikipedia 上以“监视器同步”的声音进行了详细说明。这是语言本身的核心线程支持功能，因为 TMonitor 信息被附加到系统中的任何对象上。
- TDateTime 是 Double 类型的强类型别名，用于存储日期信息（在值的整数部分）和时间信息（在小数部分）。其他别名包括类型 TDate 和 TTime。这些类型在第 2 章中进行了介绍。
- THandle 是数字类型的别名，用于表示对操作系统对象的引用，通常称为“句柄”（至少在 Windows API 术语中）。
- TMemoryManagerEx 是保存核心内存操作的一条记录，该记录允许使用仍可向后兼容的自定义内存（这是 TMemoryManager 的较新版本）替换系统内存管理器。
- THeapStatus 是一条记录，其中包含有关堆内存状态的信息，这在第 13 章中已提到。
- TTextLineBreakStyle 是一个枚举，指示给定操作系统上文本文件的换行样式。此类型的 DefaultTextLineBreakStyle 全局变量保存许多系统库使用的当前信息。类似地，sLineBreak 常量表示与字符串值相同的信息。

17.2.2 系统单元中的接口

有一些接口类型（和一些在核心级别实现接口的类）是 System 单元的一部分，值得研究。第 11 章介绍了接口。这是“系统”单元中与接口最相关的类型：

- IInterface 是所有其他接口继承的基本接口类型，并且具有与 TObject 相同的基本作用。

- IInvokable 和 IDispatch 是支持动态调用形式的接口（部分与 Windows COM 实现相关）
- 枚举器支持和比较操作由以下接口定义：IEnumerator，IEnumerable，IEnumerator <T>，IEnumerable <T>，IComparable，IComparable <T>和 IEquatable <T>。

还有一些核心类提供接口的基本实现。在实现接口时，通常会从这些类继承，如第 11 章所述：

- TInterfacedObject，一个具有引用计数和接口 ID 检查的基本实现的类
- TAggregatedObject 和 TContainedObject 这两个类为聚合的对象提供了特殊的实现，并实现了语法。

17.2.3 选定的系统 Routines（例程）

System 单元中固有的和标准的过程和功能的数量非常大，但是大多数都不常用。每个对象 Pascal 开发人员都应该了解的核心功能和过程的选择非常有限：

- Move 是系统中的核心内存复制操作，仅将给定数量的字节从一个内存位置复制到另一个位置（功能非常强大，速度非常快，但是有点危险）
- ParamCount 和 ParamStr 函数可用于处理应用程序的命令行参数（并且在 Windows 和 Mac 等 GUI 系统上也能实际使用）。
- Random 和 Randomize 是两个经典函数（可能来自 BASIC），为您提供随机值（但只有在您记得调用 Randomize 时才是伪随机的，否则每次执行都会得到相同的序列）
- 大量的核心数学函数，此处完全省略
- 许多字符串处理和字符串转换功能（介于 UTF-16 Unicode，UTF-8，ANSI 和其他字符串格式之间），其中一些是平台特定的

❖ 其中一些功能具有间接定义。换句话说，该函数实际上是指向实函数的指针，因此可以在运行时用代码动态替换原始系统行为。（当然，如果您知道自己在做什么，因为这可能是浪费应用程序内存的好方法）。

17.2.4 预定义的 RTTI 属性

我将在本章中提到的最后一组数据类型与属性有关，属性是可以附加到该语言的任何符号上的额外 RTTI 信息。

第 16 章讨论了该主题，但是我没有提到系统中的预定义属性。

以下是在“System”单元中定义的属性类：

- TCustomAttribute 是所有自定义属性的基类。这是您必须从中继承属性的基类（这是编译器将类标识为属性的唯一方法，因为没有特殊的声明语法）。
- WeakAttribute 用于指示接口引用的弱引用（请参阅第 13 章）
- UnsafeAttribute 用于禁用接口引用的引用计数（也在第 13 章中介绍）
- RefAttribute 显然用于参考值。
- VolatileAttribute 指示易失性变量，可以在外部进行修改，并且不应由编译器进行优化
- StoredAttribute 是表示属性的存储标志的另一种方法
- HPPGENAttribute 控制 C ++接口文件（HPP）的生成

- HFAAttribute 可用于微调 ARM 64 位 CPU 参数的传递，从而控制均质浮点聚合（HFA）

System 单元还有更多功能，但这是针对专业开发人员的。我宁愿转到上一章，在这里我将涉及 Classes 单元以及某些 RTL 功能。

第十八章 其他核心 RTL 类

如果可以将 TObject 类和 System 单元全部视为语言的结构部分，这是编译器本身用于构建任何应用程序所需的东西，那么运行时库中的所有其他内容都可以视为对核心系统的可选扩展。 。

RTL 具有大量系统功能，其中包括最常见的标准操作，并且部分可追溯到 Turbo Pascal 时代（早于 Object Pascal 语言）。RTL 的许多单元都是函数和例程的集合，包括核心实用程序（SysUtils），数学函数（Math），字符串运算（StringUtils），日期和时间处理（DateUtils）等。

在本书中，我真的不想深入研究 RTL 的这一传统部分，而是专注于核心类，这些核心类是 Object Pascal（VCL 和 FireMonkey）中使用的可视组件库以及其他子系统的基础。

例如，TComponent 类定义了“基于组件”架构的概念。这对于内存管理和其他基本功能也是至关重要的。

TPersistent 类是流式组件表示形式的关键。

我们还有很多其他类，因为 RTL 非常大，并且包含文件系统，核心线程支持，并行编程库，字符串构建，许多不同类型的集合和容器类，核心几何结构（例如点）和矩形），核心数学结构（例如向量和矩阵）等等。

鉴于本书的重点实际上是 Object Pascal 语言，而不是库指南，因此在这里，我将仅重点介绍一些选定的类，这些类是由于其关键作用而选择的，或者因为它们是近年来引入的，并在很大程度上被开发人员所忽略。

18.1 Class Unit（类单元）

Object Pascal RTL 类库（以及可视库）的基础单元适当地称为 System.Classes。本单元包含大量的类别最多的类，没有特别的重点。值得一看的是重要的，然后对最重要的进行深入分析。

18.1.1 类单元中的类

因此，这是简短的列表（大约在该单元中实际定义的类的一半）：

- TList 是指针的核心列表，通常适合作为无类型的列表。通常，建议使用 TList <T>代替，如第 14 章所述。
- TInterfaceList 是实现 IInterfaceList 的接口的线程安全列表，值得一看（在此不做介绍）。
- TBits 是一个非常简单的类，用于处理数字或其他值中的各个位。用移位和二进制或与和运算符进行位操作要高得多。
- TPersistent 是一个基础类（TComponent 的基类），在下一节中将详细介绍。
- TCollectionItem 和 TCollection 是用于定义集合属性（即具有值数组的属性）的类。对于组件开发人员（以及在使用组件时间接），这些是重要的类，对于一般的最终用户代码而言则不是那么重要。
- TStrings 是字符串的抽象列表，而 TStringList 是 TStrings 基类的实际实现，为实际字符串提供存储。每个项目还附加有一个对象，这是将字符串列表用于名称/值字符串对的标准方法。本章末尾的“使用字符串列表”部分中提供了有

关此类的更多信息。

- TStream 是一个抽象类，表示具有顺序访问权限的任何字节序列，其中可以包含许多不同的存储选项（内存，文件，字符串，套接字，BLOB 字段以及许多其他选项）。“类”单元定义许多特定的流类，包括 THandleStream，TFileStream，TCustomMemoryStream，TMemoryStream，TBytesStream，TStringStream 和 TResourceStream。其他特定的流在不同的 RTL 单元中声明。您可以在本章的“介绍流”部分中阅读有关流的介绍。
- 低级组件流的类，例如 TFiler，TReader，TWriter 和 TParser，主要由组件作者使用，而他们甚至不经常使用。
- TThread 类，它定义对与平台无关的多线程应用程序的支持。也是用于异步操作的类，称为 TBaseAsyncResult。
- 用于实现观察者模式的类（例如，在可视实时绑定中使用），包括 TObserver，TLinkObserver 和 TObserverMapping。
- 特定自定义属性的类，例如 DefaultAttribute，NoDefaultAttribute，StoredAttribute 和 ObservableMemberAttribute。
- 基本的 TComponent 类是 VCL 和 FireMonkey 中所有可视和非可视组件的基类，本章稍后将详细介绍。
- 动作和动作列表支持的类（动作是 UI 元素或内部对“命令”问题的抽象），包括 TBasicAction 和 TBasicActionLink。
- 表示非可视组件容器的类 TDataModule。
- 文件和流操作的高级接口，包括 TTextReader 和 TTextWriter，TBinaryReader 和 TBinaryWriter，TStringReader 和 TStringWriter，TStreamReader 和 TStreamWriter。 本章还将介绍这些类。

18.1.2 TPersistent（持久）类

TObject 类具有一个非常重要的子类，它是整个库的基础之一，称为 TPersistent。如果您查看类的方法，其重要性可能会令人惊讶……因为该类几乎没有。TPersistent 类的关键元素之一是使用特殊的编译器选项{M +}定义的，该选项的作用是启用已发布的关键字，如第 10 章所述。

Published 关键字在流属性中起着基本作用，这说明了类的名称。最初，只有从 TPersistent 继承的类才可以用作已发布属性的数据类型。RTTI 在 Object Pascal 编译器的更高版本中的扩展稍微改变了情况，但已发布关键字和{$ M +}编译器选项的作用仍然存在。

❖ <u>使用今天的编译器，如果您将发布的关键字添加到一个不继承自 TPersistent 且未获得{$ M +}编译器标志的类中，则系统仍将添加适当的支持，并带有警告。</u>

TPersistent 类在层次结构中的具体作用是什么？首先，它充当 TComponent 的基类，我将在下一节中介绍。其次，它用作于属性值的数据类型的基类，以便可以正确地流式传输这些属性及其内部结构。示例是表示字符串，位图，字体和其他对象的列表的类。

如果 TPersistent 类最相关的功能是对已发布的关键字的“激活”，则它仍然有几个值得研究的有趣方法。第一个是 Assign 方法，用于将对象数据从一个实例复制到另一个实例（深层副本，而不是引用的副本）。这是每个用于属性值的持久

化类都应手动实现的功能（因为该语言没有自动深度复制操作）。第二个是反向操作，AssignTo，它受保护。该类中的这两种方法以及其他几种方法主要由组件编写者使用，而不是由应用程序开发人员使用。

18.1.3 TComponent 类

TComponent 类是最常与 Object Pascal 编译器结合使用的组件库的基石。组件的概念基本上是具有一些额外的设计时行为，特定的流功能（以便可以在运行的应用程序中保存和还原设计时配置）和 PME（属性方法事件）的类的概念）模型，我们将在第 10 章中讨论。

此类定义了大量的标准行为和功能，并基于对象所有权，跨组件通知等概念引入了自己的内存模型。尽管没有对所有属性和方法进行完整的分析，但是值得一提的是，由于 TComponent 类在 RTL 中的核心作用，因此需要重点关注 TComponent 类的一些关键功能。

TComponent 类的另一个关键特性是引入了一个虚拟的 Create 构造函数，这对于在仍然调用该类的特定构造函数代码的同时，从类引用创建对象的能力至关重要。我们在第 12 章中进行了介绍，但这是 Object Pascal 语言的一个独特功能，值得理解。

Components Ownership（所有权）

所有权机制是 TComponent 类的关键元素。如果使用所有者组件创建组件（作为参数传递给其虚拟构造函数），则该所有者组件将负责销毁所拥有的组件。简而言之，每个组件都具有对其所有者的引用（Owner 属性），还具有其拥有的组件列表（Components 数组属性）及其编号（ComponentCount 属性）。

默认情况下，将组件放置在设计器（表单，框架或数据模块）中时，这被视为组件的所有者。在代码中创建组件时，由您指定所有者或传递 nil（在这种情况下，您将负责自己从内存中释放组件）。

您可以使用 Components 和 ComponentCount 属性列出组件所拥有的组件（在本例中为 aComp），其代码如下：

```
var
  I: Integer;
begin
  for I := 0 to aComp.ComponentCount – 1 do
    aComp.Components[I].DoSomething;
```

或通过以下方式使用本机枚举支持：

```
var
  childComp: TComponent;
begin
  for childComp in aComp do
    childComp.DoSomething;
```

销毁组件时，会将其从所有者列表（如果有）中删除，并销毁它拥有的所有组件。这种机制对于 Object Pascal 中的内存管理至关重要：由于没有垃圾回收，所有权可以解决您的大多数内存管理问题，正如我们在第 13 章中看到的那样。

如前所述，通常表单或数据模块中的所有组件都将表单或数据模块作为所有者。只要释放表单或数据模块，它们托管的组件也会被破坏。这是从流中创建组件时发生的情况。

组件属性

除了核心所有权机制（还包括通知和此处未介绍的其他功能）之外，任何组件都具有两个已发布的属性：

- Name 是带有组件名称的字符串。这用于动态查找组件（调用所有者的 FindComponent 方法）并将组件与引用该组件的表单字段连接。同一所有者拥有的所有组件必须具有不同的名称，但它们的名称也可以为空。这里有两个简短的规则：设置正确的组件名称以提高代码的可读性，并且永远不要在运行时更改组件的名称（除非您真的意识到可能的讨厌的副作用）。
- Tag 是库未使用的 NativeInt 值（过去曾经是 Integer），但可用于将其他信息连接到组件。该类型与指针和对象引用在大小上兼容，这些指针和对象引用通常存储在组件的 Tag 中。

组件流

FireMonkey 和 VCL 用来创建 FMX 或 DFM 文件的流传输机制都基于 TComponent 类。Delphi 流机制可保存组件及其子组件的已发布属性和事件。

这就是在 DFM 或 FMX 文件中获得的表示形式，也是将设计器中的组件复制并粘贴到文本编辑器中后得到的结果。

有一些在运行时获取相同信息的方法，包括 TStream 类的 WriteComponent 和 ReadComponent 方法，以及同一类的 ReadComponentRes 和 WriteComponentRes 方法，以及 TReader 和 TWriter 特殊类的 ReadRootComponent 和 WriteRootComponent 有助于处理问题。与组件流。这些操作通常使用表单流的二进制表示形式：可以使用全局过程 ObjectResourceToText 将表单二进制表示形式转换为文本形式，而 ObjectTextToResource 进行相反的转换。

一个关键因素是，流技术不是组件已发布属性的完整集合。 流包括：

- 值与默认值不同的组件的已发布属性（换句话说，不会保存默认值以减少大小）
- 仅标记为已存储的已发布属性（默认设置）。存储设置为 false 的属性（或返回 false 的函数）将不会保存。
- 通过覆盖 DefineProperties 方法，在运行时添加的其他条目（不是对应的组件属性）。

从流文件创建组件时，将发生以下顺序：

- 调用组件的虚拟 Create 构造函数（执行适当的初始化代码）
- 从流中加载属性和事件（如果发生事件，将方法名称重新映射到内存中的实际方法地址）
- 调用 Loaded 虚拟方法以完成加载（组件可以执行额外的自定义处理，这次使用已从流中加载的属性值）

18.2 现代文件访问

从其祖先的 Pascal 语言中借用的 Object Pascal 仍然具有用于处理文件的关键字和核心语言机制。当引入 Object Pascal 时，这些方法基本上已被弃用，我在本书中不再赘述。

我将在本节中介绍的是一些用于处理文件的现代技术，介绍了 IOUtils 单元，流类以及 readers（读者）和 writers（作家）类。

18.2.1 IOUtils（输入/输出实用程序）单元

System.IOUtils 单元是运行时库中相对较新的功能。它定义了三个主要是类方法的记录：TDirectory，TPath 和 TFile。尽管 TDirectory 很明显是用于浏览文件夹并查找其文件和子文件夹,但可能还不清楚 TPath 和 TFile 之间的区别是什么。第一个是 TPath，用于处理文件名和目录名，以及用于提取驱动器，无路径，扩展名等文件名的方法，以及用于处理 UNC 路径的方法。而 TFile 记录使您可以检查文件的时间戳和属性，还可以操作文件，对其进行写入或复制。像往常一样，值得看一个例子。IoFilesInFolder 应用程序项目可以提取给定文件夹的所有子文件夹，并且可以获取该文件夹下具有给定扩展名的所有文件。

提取子文件夹

程序可以使用 TDirectory 记录的 GetDirectories 方法，将值 TSearchOption.soAllDirectories 作为参数传递，从而用目录下的文件夹列表填充列表框。 您可以列举一个字符串数组中的结果：

```
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;
```

搜索文件

该程序的第二个按钮使您可以通过基于给定的掩码使用 GetFiles 调用扫描每个目录来获取那些文件夹的所有文件。 通过将类型为 TFilterPredicate 的匿名方法传递给 GetFiles 的重载版本，可以进行更复杂的过滤。

本示例使用基于掩码的更简单过滤，并填充内部字符串列表。删除完整路径后，仅保留文件名，然后将此字符串列表的元素复制到用户界面。调用 GetDirectories 方法时，您只会获得子文件夹，而不会获得当前文件夹。这就是程序首先在当前文件夹中搜索，然后在每个子文件夹中进行查找的原因：

```
procedure TFormIoFiles.BtnPasFilesClick(Sender: TObject);
var
  PathList, FilesList: TStringDynArray;
  StrPath, StrFile: string;
begin
  if TDirectory.Exists (EdBaseFolder.Text) then
  begin
    // 清理
    ListBox1.Items.Clear;
    // 在给定的文件夹中搜索
    FilesList := TDirectory.GetFiles (EdBaseFolder.Text, '*.pas');
    for StrFile in FilesList do
```

```
        SFilesList.Add(StrFile);
      // 在所有子文件夹中搜索
      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;
      // 现在仅将文件名（无路径）复制到列表框中
      for StrFile in SFilesList do
        ListBox1.Items.Add (TPath.GetFileName(StrFile));
    end;
  end;
```

在最后几行中，使用 TPath 的 GetFileName 函数从文件的完整路径中提取文件名。TPath 记录还有其他一些有趣的方法，包括 GetTempFileName，GetRandomFileName，用于合并路径的方法，一些用于检查路径是否有效或包含非法字符的方法，等等。

18.2.2 Streams（流）介绍

如果 IOUtils 单元用于查找和处理文件，则当您要读取或写入文件（或任何其他类似的顺序访问数据结构）时，可以使用 TStream 类及其许多后代类。TStream 抽象类只有几个属性（大小和位置），以及所有流类共享的基本接口以及主要的 Read 和 Write 方法。此类表示的概念是顺序访问。每次您读写一定数量的字节时，当前位置都会以该数量前移。对于大多数流，您可以向后移动位置，但是也可以有单向流。

普通 Stream（流）类

正如我之前提到的，Classes 单元定义了几个具体的流类...

包括以下内容：

- THandleStream 定义使用文件句柄引用的磁盘文件流。
- TFileStream 定义由文件名引用的磁盘文件流。
- TBufferedFileStream 是优化的磁盘文件流，它使用内存缓冲区来提高性能。该流类已在 Delphi 10.1 Berlin 中引入。
- TMemoryStream 定义了内存中的数据流，您也可以使用指针进行访问。
- TBytesStream 表示内存中的字节流，您也可以像字节数组一样进行访问
- TStringStream 将流与内存中的字符串关联。
- TResourceStream 定义了一个流，该流可以读取链接到应用程序的可执行文件中的资源数据。

使用 Streams（流）

创建和使用流就像创建特定类型的变量并调用组件的方法以从文件中加载内容一样简单。例如，给定一个流和一个 Memo 组件，您可以编写：

```
AStream := TFileStream.Create (FileName, fmOpenRead);
Memo1.Lines.LoadFromStream (AStream);
```

如您在此代码中看到的，文件流的 Create 方法具有两个参数：文件名和一些

标志，指示请求的访问方式。正如我提到的，流支持读和写操作，但是这些操作是很底层的，因此我建议使用下一节中讨论的 readers 和 writers 类。直接使用 stream 提供的是全面的操作，例如将整个流加载到上面的代码片段中，或将其中一个复制到另一个中：

```
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;
```

18.2.3 使用 Readers 和 Writers

写入和读取流的一种很好的方法是使用 RTL 中的 reader 和 writer 类。在“class”单元中定义了六个 readers 和 writers 类：

- TStringReader 和 TStringWriter 处理内存中的字符串（直接或使用 TStringBuilder）
- TStreamReader 和 TStreamWriter 在通用流（文件流，内存流等）上工作。
- TBinaryReader 和 TBinaryWriter 处理二进制数据而不是文本。

每个文本阅读器都实现了一些基本的阅读技术：

```
function Read: Integer; overload;
function ReadLine: string;
function ReadToEnd: string;
```

每个文本编写器都有两组重载操作，没有（Write）和（WriteLine）行尾分隔符。这是第一组：

```
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;
```

Readers（文本阅读器）和 writers（文本编写器）

为了写入流，TStreamWriter 类使用流或使用文件名，append / create 属性和

作为参数传递的 Unicode 编码来创建一个流。

因此，我们可以像在 ReaderWriter 应用程序项目中一样进行编写：

```
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;
```

为了读取 TStreamReader，您可以在流或文件上再次进行操作（在这种情况下，它可以从 UTF BOM 标记检测编码）：

```
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;
```

请注意如何检查 EndOfStream 状态。与直接使用文本流（甚至是字符串）相比，这些类特别易于使用，并提供良好的性能。

二进制读写器

TBinaryReader 和 TBinaryWriter 类用于管理二进制数据而不是文本文件。这些类通常封装一个流（文件流或任何类型的内存中流，包括套接字和数据库表 BLOB 字段），并且具有重载的 Read 和 Write 方法。

作为一个（相当简单的）示例，我编写了 BinaryFiles 应用程序项目。该程序在第一部分中将几个二进制元素写入文件（属性的值和当前时间）并读回它们，并分配属性值：

```
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;
  time: TDateTime;
```

```
begin
   br := TBinaryReader.Create('test.data');
   try
      Left := br.ReadInt32;
      Log ('Left read: ' + IntToStr (Left));
      time := br.ReadDouble;
      Log ('Time read: ' + TimeToStr (time));
   finally
      br.Free;
   end;
end;
```

使用这些读取器和写入器类的关键规则是，您必须按照写入顺序读取数据，否则将完全弄乱数据。实际上，仅保存单个字段的二进制数据，而没有有关字段本身的信息。没有什么可以阻止您在文件中插入数据和元数据的，就像在实际值或引用该字段的标记之前保存下一个数据结构的大小一样。

18.3 构建字符串和字符串列表

看完文件和流之后，我想花一些时间专注于处理字符串和字符串列表的方法。这些是非常常见的操作，并且有大量针对这些操作的 RTL 功能。在这里，我只介绍一些。

18.3.1 TStringBuilder 类

我已经在第 6 章中提到过，与其他语言不同，Object Pascal 完全支持直接字符串连接，这实际上是一个相当快的操作。但是，语言 RTL 还包括一个用于从不同数据类型的片段中组装字符串的特定类，称为 TStringBuilder。

作为使用 TStringBuilder 类的简单示例，请考虑以下代码片段：

```
var
   SBuilder: TStringBuilder;
   Str1: string;
begin
   SBuilder := TStringBuilder.Create;
   SBuilder.Append(12);
   SBuilder.Append('hello');
   Str1 := SBuilder.ToString;
   SBuilder.Free;
end;
```

注意，我们必须创建和销毁该 TStringBuilder 对象。上面您可以注意到的另一个元素是，您可以将许多不同的数据类型作为参数传递给 Append 函数。

TStringBuilder 类的其他有趣方法包括 AppendFormat（带有内部对 Format 的调用）和添加 sLineBreak 值的 AppendLine。

除了 Append，还有一系列相应的 Insert 重载方法，以及 Remove 和一些 Replace 方法。

❖ TStringBuilder 类具有一个不错的接口，并提供了良好的可用性。但是，就性能而言，使用标准的字符串连接和格式化功能可以提供更好的结果，这与其他定义不可变字符串并在纯字符串连接的情况下性能很差的编程语言不同。

StringBuilder 中的方法链接

TStringBuilder 类的一个非常特殊的函数是，大多数方法都是函数，它们返回已应用到的当前对象。

这种编码习惯使方法链接成为可能，即在上一个方法返回的对象上调用方法。而不是写：

```
SBuilder.Append(12);
SBuilder.AppendLine;
SBuilder.Append('hello');
```

你可以写：

```
SBuilder.Append(12).AppendLine.Append('hello');
```

也可以将其格式化为：

```
SBuilder.
   Append(12).
   AppendLine.
   Append('hello');
```

我倾向于比原始语法更喜欢这种语法，但是我知道它只是语法糖，有些人确实更喜欢原始版本，每行都写有该对象。无论如何，请记住，对 Append 的各种调用不会返回新对象（因此不会潜在的内存泄漏），而是将您要应用方法的对象完全相同。

18.3.2 使用字符串列表

字符串列表是许多视觉组件使用的非常常见的抽象，但也用作处理由单独的行组成的文本的方式。有两个主要类用于处理字符串列表：

- TStrings 是一个抽象类，表示所有形式的字符串列表，无论它们的存储实现如何。此类定义字符串的抽象列表。因此，TStrings 对象仅用作能够存储字符串本身的组件的属性。
- TStringList 是 TStrings 的子类，它定义了一个具有其自身存储空间的字符串列表。您可以使用此类定义程序中的字符串列表。

这两类字符串列表还具有即用型方法，可以将它们的内容存储到文本文件 SaveToFile 和 LoadFromFile（或完全从 Unicode 启用）或从文本文件中加载它们。要遍历列表，您可以基于其索引使用简单的 for 语句，就好像列表是数组或 for-in 枚举器一样。

18.4 运行时库很大

RTL 可以与 Object Pascal 编译器一起使用，还有很多其他功能，其中包括许多核心特性，可在多个操作系统上进行开发。

详细介绍整个运行时库将轻松地填充另一本与本本相同大小的书。

如果我们仅考虑库的主要部分，即“System”命名空间，则它包含以下单元（从中删除了一些很少使用的单元）：

- System.Actions 包括对动作体系结构的核心支持，该体系结构提供了一种方法来表示从用户界面层连接但抽象的用户命令。
- System.AnsiStrings 具有处理 Ansi 字符串的旧功能（仅在 Windows 上），在第 6 章中进行了介绍。
- System.Character 具有 Unicode 字符（Char 类型）的固有类型帮助器，已在第

3 章中介绍过。

- System.Classes 提供了核心系统类，并且是本章第一部分中详细介绍的单元。
- System.Contnrs 包括旧的，非泛型的容器类，例如对象列表，字典，队列和堆栈。我建议尽可能使用相同类的泛型版本。
- System.ConvUtils 具有用于不同度量单位的转换实用程序库
- System.DateUtils 具有处理日期和时间值的函数
- System.Devices 与系统设备（例如 GPS，加速度计等）接口。
- System.Diagnostics 定义了一个记录结构，用于精确测量测试代码中的经过时间，我在本书中偶尔使用过。
- System.Generics 实际上有两个单独的单元，一个用于泛型集合，一个用于泛型类型。这些单元在第 14 章中介绍。
- System.Hash 具有定义哈希值的核心支持。
- System.ImageList 包括一个抽象的，与库无关的实现，用于管理图像列表以及作为元素集合的单个图像的一部分。
- System.IniFiles 定义用于处理 INI 配置文件的接口，该接口通常在 Windows 中找到。
- System.IOUtils 定义了用于文件系统访问的记录（文件，文件夹，路径），本章前面已经介绍了这些记录。
- System.JSON 包含相同的核心类，用于处理常用的 JavaScript 对象符号（JSON）中的数据。
- System.Math 定义了用于数学运算的函数，包括三角函数和财务函数。在其名称空间中，它还具有用于向量和矩阵的其他单元。
- System.Messaging 具有用于在不同操作系统上处理消息的共享代码。
- System.NetEncoding 包括对某些常见 Internet 编码的处理，例如 base64，HTML 和 URL。
- System.RegularExpressions 定义正则表达式（regex）支持。
- System.Rtti 具有整套 RTTI 类，如第 16 章所述。
- System.StrUtils 具有核心和传统的字符串处理功能。
- System.SyncObjs 定义了一些用于同步多线程应用程序的类。
- System.SysUtils 具有系统实用程序的基本集合，其中一些最传统的实用程序可以追溯到编译器的早期。
- System.Threading 包括相当近期的并行编程库的接口，记录和类。
- System.Types 具有一些核心的附加数据类型，例如 TPoint，TRectangle 和 TSize 记录，TBitConverter 类以及 RTL 使用的更多基本数据类型。
- System.TypInfo 定义了较旧的 RTTI 接口，该接口也已在第 16 章中引入，基本上由 System.RTTI 单元中的接口取代。
- System.Variants 和 System.VarUtils 具有使用变体的功能（第 5 章中介绍的语言功能）。
- System.Zip 接口文件压缩和解压缩库。

RTL 中还有其他几个部分，它们是系统名称空间的子部分，每个部分都包含多个单元（有时为多个单元，例如 System.Win 名称空间），包括 HTTP 客户端（System.Net）和 物联网支持（System.Beacon，System.Bluetooth，System.Sensors 和 System.Tether）。当然，还有翻译后的 API 和标头文件，用于与所有受支持的操作系统进行接口连接。

同样，有大量现成的 RTL 函数，类型，记录，接口和类可供您使用，以利用 Object Pascal 的功能。 花时间浏览系统文档以了解更多信息。

in closing

第 18 章标志着本书的结尾，除了以下三个附录。这本来是我的第一本书，专门针对 Object Pascal 语言，并且我正在尽最大努力不断更新它，并随着时间的推移维护本书的文字和示例。我仅对 Delphi 10.1 Berlin 进行了 PDF 更新，现在您正在为 Delphi 10.4 Sydney 阅读此新版本。

再次参考引言以获取来自 GitHub 的最新书籍源代码，并访问书籍网站或我的博客以获取将来的信息和更新。

我希望您在过去的 25 年中喜欢阅读并喜欢撰写和撰写有关 Delphi 的书，并且喜欢它。使用 Delphi 进行编码愉快！

end.

本书的最后部分有一些附录，重点放在值得考虑的特定附带问题上，但这些内容不在本文的讨论范围之内。Pascal 和 Object Pascal 语言的历史很短，并且有词汇表。

附录摘要

附录 A：对象 Pascal 的演变
附录 B：术语表
附录 C：索引

（内容略，欲阅请查阅原著）