

# Delphi 高手突破

申 旻 著

清华大学出版社

(京)新登字 158 号

## 内 容 简 介

本书以理论结合实践的方式，论述“如何在 Delphi 中使用面向对象编程方法，构建良好设计的程序”的主题。本书第 1、2、3 章以不同于一般书籍的方式，介绍面向对象编程的基础知识及其在 Object Pascal 中的语言实现。第 4、5 章介绍 VCL 库的相关知识，其中第 4 章为您剖析部分 VCL 的核心组件源码，第 5 章介绍定制、设计组件的方法。第 6、7 章介绍程序构架设计，其中第 6 章介绍一般性的代码设计准则，其内容可以完全脱离 Delphi，因为这些准则是跨语言的；第 7 章是一个完整的代码设计实例，以编写一个多工作区的文本编辑器为例，从实践角度说明程序构架设计的方法。

本书面向 Delphi 程序员，特别是写给那些已经具有一定的实现能力而欲求寻找一种“突破”的 Delphi 程序员，作为他们提高的参考用书，同时也可以作为高校学生以及程序爱好者的参考用书。

版权所有，翻印必究。

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

书 名：Delphi 高手突破

作 者：申 旻 编著

出 版 者：清华大学出版社（北京清华大学学研大厦，邮编 100084）

<http://www.tup.tsinghua.edu.cn>

责任编辑：朱英彪

印 刷 者：

发 行 者：新华书店总店北京发行所

开 本：787×1092 1/16 印张： 字数： 千字

版 次：2002 年 月第 1 版 2002 年 月第 1 次印刷

书 号：ISBN 7-900643-57-5

印 数：0001~

定 价： 元

# 引言

感谢您阅读本书！

本书是写给程序员的，确切地说，是写给 Delphi 程序员的，再确切些，是写给已经有了一定的实现能力而欲求寻找一种“突破”的 Delphi 程序员的。

在接触了两年的 Delphi 之后，我曾经迷茫过。我可以写各种各样的程序，我懂得 VCL 大多数组件的用法，我知道应该调用哪个 Windows API 来完成我要的功能……但时常会疑惑：这就是写程序了？那时候在大学里读书，课余时间很多，每天就不停地写，写各种程序，包括课堂的作业、在网上发布的免费软件以及接到的开发项目。每天都写代码，有了 Delphi 的 help 和 MSDN，似乎不会有什么困难，只是偶尔会觉得单调。不禁又问自己：这就是写程序了？

我迷茫，是因为我感到，写程序不应该是件单调的事情；我迷茫，是因为自己总在寻求却始终没有找到一种“突破”的感觉；我迷茫，是因为我想成为“高手”却不知道如何去做……

我有很多理由喜欢 Delphi，但是应该说，正是 Delphi 的 RAD 开发方式让我陷入迷茫。我迷惑于 RAD 使人能力退化还是一种革命性的进步。我相信很多 Delphi 程序员都会和我有一样的经历。

我很幸运，就在我迷茫的时候，认识了我的同学 Lythm，受其影响，我开始涉猎面向对象编程类的书籍。从《Thinking in C++》到《C++面向对象高效编程》、从《Inside C++ Object Model》到《设计模式》……然而，我一直在寻找却始终没有找到一本完整的以 Delphi/Object Pascal 来讲述面向对象编程方法学的书，其间只有一本 Charlie Calvert 的《Delphi 4 Unleashed》赢得了我的欢心，其中关于多态的描述非常精彩，只可惜相关篇幅太少。

即便如此，我仍然感到找到了方向，于是就暂时放下手中的键盘，钻研起理论。这段时期已经不会再感到迷茫，取而代之的是一种自身能感受到的“突破”的感觉。工作后，更加有机会将自己所学的东西应用于实际开发之中，并不断修正自己头脑中的理论体系。

您是否曾经或正在经历我曾经的迷茫呢？

如果是，那么我想这本书应该是您所要寻找的，因为我所要写的，就是一本我自己梦寐以求在寻找的书，一本以 Delphi/Object Pascal 来讲述面向对象编程方法学、代码设计方法的书。

我希望您和我一样幸运，不！应该说，您比我更幸运，因为您比我多了这本书。

本书的书名是《Delphi 高手突破》，我并没有任何文字暗示自己已经成为“高手”，

所谓“高手突破”的解释并不是高手来帮助您突破。写这样一本书，我只是希望能把自己的“突破”的感觉与经验和大家共享，同时，它也是我对前一段时间学习的一个总结。

我很喜欢这本书的英文名称，是我自己起的，我愿意将它叫作《Design in Delphi》，不过请原谅我无法用中文准确地、优美地将它表述出来。

## 这本书没有什么

这是本特殊的 Delphi 编程方面的书，它不会教您如何使用 Delphi，也不会教您如何使用类似 TListBox 那样的组件，更不会涉及诸如多线程、DLL、API 等 Windows 编程的内容，也没有热门的 COM/DCOM、Web Service 等。

## 这本书有什么

这本书会告诉您面向对象编程的基础理论，会为您剖析 VCL 的部分源码，会告诉您在开始敲键盘写代码之前应该做什么，怎样使您的代码的构架被更良好地设计以致便于更容易被维护和修改。

## 这本书还没有什么

看起来，这是一本更侧重于讲设计的书。请不要误会，这本书还不会教您关于面向对象分析/设计（OOA/OOD）的内容，更没有 UML。

## 这本书究竟有什么

就本质来说，这本书只讲 OOP，当然，是用 Delphi 作为载体，因为它是写给 Delphi 程序员的。其中有设计的内容，但仅限于代码设计。本书第 7 章就一个实例向您展示代码构架设计的一种可能的方式，但此方式并不是惟一的。实例不需要照搬，而需要领会。

## 本书章节介绍

本书第 1、2、3 章介绍 OOP 的基础知识，基本上所有编程语言的书都会有这样的内容，但本书一定会带给您不同的感觉。

第 4、5 章介绍 VCL 相关的知识。其中第 4 章为您剖析部分 VCL 的核心组件源码，第 5 章介绍定制、设计组件的方法。

第 6、7 章介绍程序构架设计。第 6 章介绍一般性的代码设计准则，其内容可以完全脱离 Delphi，因为这些准则是跨语言的。第 7 章是一个完整的代码设计实例，以编写一个多工作区的文本编辑器为例，从实践角度说明程序构架设计的方法。

## 本书的支持网页

本书的支持网页为：<http://www.sunistudio.com/microsoft/book/did/>

作者 Email: [microsoft@sunistudio.com](mailto:microsoft@sunistudio.com)

希望您喜欢本书以及它的作者——我！谢谢。

## 感谢

在此，我要感谢为这本书的诞生给予过我帮助的人们。

首先，我要感谢我的女友 Esan，在我的写作过程中，她一直陪伴在我身边，不断地给我鼓励和支持，使我得以最终完成写作。

其次，我要感谢我的同事兼拍档唐沐，是他为本书的每一章创作提供了精美、有趣而又富有创意的插图，使得本书更富有一些活泼的气氛。

再次，我要感谢 CSDN 以及大富翁论坛的热心网友们，当我在网络上发表了本书的一些预览节选后，他们给我提出了许多宝贵的意见和建议。

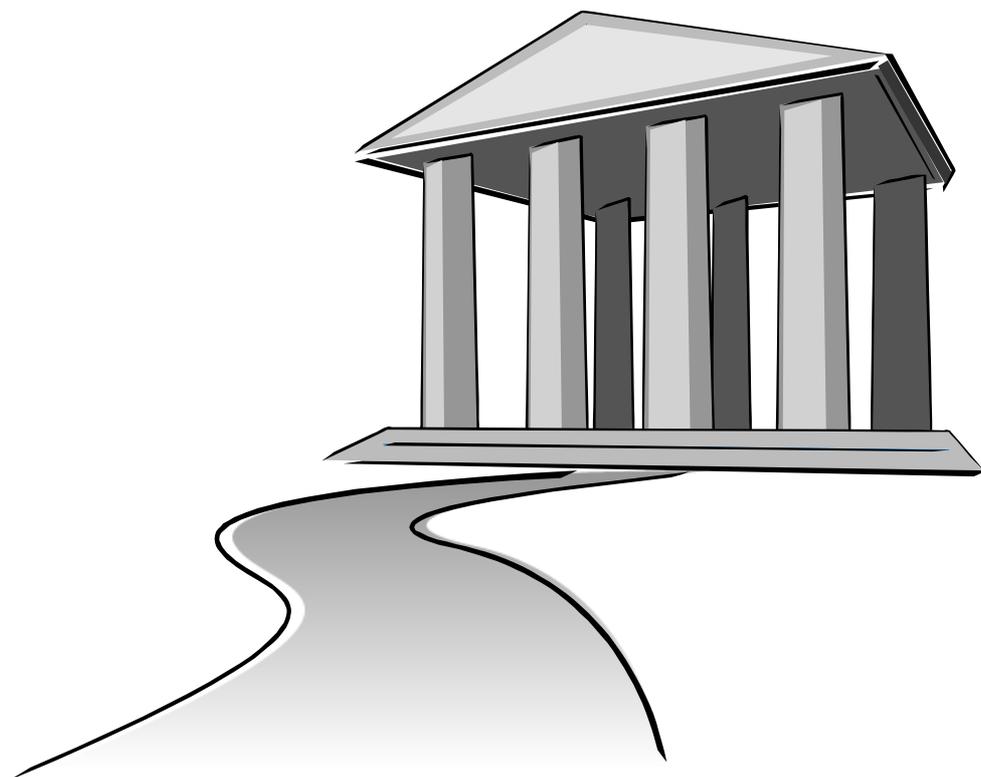
最后，我要感谢清华大学出版社的朱英彪编辑，他的宽容和给予我的帮助使得我们的合作非常愉快，也使得本书能够顺利完成及出版。

申 旻

2002 年 7 月于珠海

# 第 1 章 重新认识Delphi

简单性是这个世界上最难获得的东西：它是经验的最终界限，也是天才的最终努力目标。——George Sand



*Go to Delphi!*



你已经是一位熟练的 Delphi 程序员，可以运用 Delphi 快速地写出一个漂亮、实用的程序；你热爱 Delphi，她已经成了你工作、学习中不可或缺的一部分。我假设这些都为真，那么你当初选择 Delphi 作为自己的首选开发工具一定有自己的理由或原因。

至少，我自己是符合以上所有假设的。现在，想和你分享的，正是我选择 Delphi 的理由及原因，以及我对 Delphi 的认识。你可以把我看作一个拥护 Delphi 的狂热分子，虽然那样会让我感到你把我看得太过肤浅，我并不承认，但是我也不会介意。因为，我真的热爱她。

第一次接触 Delphi 的版本是 3.0，那时候只不过是把她当作 Visual Basic 一样的 RAD 工具来用而已。但是，随着时间的流逝，从 Delphi 3 到 Delphi 4、Delphi 5、Delphi 6 以及 Kylix，对 Delphi 的认识也越来越深。她是一个有着丰富内涵的工具，让人对她越了解，就越对她迷恋，越感觉离不开她，虽然她只是一个工具而已。

Pascal 是一种讲究程序美学的语言——毫无疑问，Pascal 代码是最优美的代码——基于 Object Pascal（一种支持面向对象的 Pascal 语言）的 Delphi 让这种美达到了极致。

现在，你可以打开 Delphi，选择“Help”→“About”菜单，出现 About 窗口后，按住 Alt 键，然后顺序输入“team”，你看到了什么？是的，Delphi 开发人员名单。感谢他们创造出如此具有艺术气质的开发工具！

## 1.1 开发工具“以人为本”论

经常可以在各个编程论坛上看到类似这样的问题：“VB 还有没有前途？”、“Delphi 是不是要淘汰了？”、“MFC 是不是要被 .NET 取代了？”……其实，这些问题在被提出的当时，是没有人能给出答案的。这是因为，一种技术、一个产品的前途，并不完全是其本身所能左右的，还与市场需求、出品公司的发展方向等因素有关。而我们所应该关注的，是否就是这些问题的答案呢？我认为不是。

我们知道，世间万物由原子组成。千变万化的程序归根结底由顺序、循环、分支三种结构构成，无论 VC 的 MFC，还是 Delphi 的 VCL，都是由面向对象技术构建的（暂且不论其面向对象的程度）。当你拨开事物表面的表象后，看到的是相同的或近似的本质！而掌握了本质之后，就会发现表象的表现形式是那么的理所当然。试想，当你能像侯捷（《深入浅出 MFC》的作者）那样把 MFC 剥得体无完肤时，还会担心 MFC 被某某框架所取代吗？从这个角度来说，对于一名专业程序员，编程的理念是万变不离其宗的。发现问题、分析问题、解决问题的过程是存在着某种模式的，当你掌握了这种模式后，不同的编程语言、不同的开发环境对你来说，是有共通之处的。

我认为 C++ 是每个专业程序员所必须掌握的。当然，并不是说单纯学习其语法（甚至可以忽略一些语法的学习），而是通过 C++ 学习面向对象的设计、编程方法。因为 C++ 博大精深无所不及。在 C++ 中，你可以学习到面向对象理论的全部。学习之后，你会被 C++ 所改造，因为在面向对象理论中存在的、但有所争议的特性（如多重继承）在 C++ 中都得到了支持。只有在掌握之后，才可能作出自己的选择（支持或反对）。在掌握了面向对

象的理论之后，无论 C++、Object Pascal 或是 Java 乃至 C#，却会感觉到它们的异曲同工之处。

那是否就是说，开发工具（或许应该称为集成开发环境，不过下文还是按我的习惯，用开发工具来称呼）之间除了支持的语言不同外，就不存在其他差异了？当然也不是。开发工具是帮助用户实现理念的工具，也就是构建在基础理念上的上层建筑。开发工具对于用户所要实现的理念的支持程度以及对实现过程的简化程度，就是开发工具的体贴度。开发工具于程序员，犹如兵器于士兵，兵器不顺手，未战先败一半。

我一直很喜欢诺基亚手机的广告词：科技以人为本！是的，“人”才是本，工具的使命是辅助人更快、更容易地达到目的。因此，开发工具也应该以人为本！

作为一名程序员，作为开发工具的最直接的使用者，我希望我所使用的开发工具真正是我的伙伴、助手，它能给我带来自由的感觉，让我自由地在代码的世界中驰骋，它能迁就我、适应我，而不是相反，给我套上枷锁！

如今在 Windows 平台上，有许许多多的开发工具可以选择——Visual C++、Visual Basic、Delphi、C++ Builder、JBuilder……它们基于不同的编程语言、忠于不同公司的产品理念。从这个角度来说，它们之间的差异是非常大的。

那什么样的开发工具才是优秀的、体贴的、以人为本的？我的标准是符合以下四点：

- (1) 能够将要解决的问题简化，并以某种理念快速实现之。
- (2) 不隐藏任何用户想知道的细节。
- (3) 可以忽略用户所不想知道的细节。
- (4) 主动去适应不同层次的程序员。

符合以上四点的开发工具有吗？我的答案是：有！那就是 Delphi！她将一切化繁为简，却从不阻止我寻求真实。用户可以在她构造的简化了的 VCL 的虚拟世界中完成任务。也可以钻进 VCL 的世界以探询她和现实世界（即 Windows 平台的真实接口）的映射关系，学习她的 Framework 的设计。此外，还可以扩展那个虚拟的 VCL 世界以适应自己的需要。

我为存在着这样的开发工具而感到幸运，更为幸运的是，我可以选择她，和她一起完成我的工作（现实中，项目中使用什么编程语言、开发工具，时常并不是个人所能左右的，会受很多因素制约。例如，客户的硬件环境、操作系统环境、开发环境、开发工具的成本、许可证等。因此，能选择自己喜欢的开发工具进行开发工作实在是幸运的）！

通过 C++ 学习面向对象的观念，用 Delphi 去解决现实世界的问题，这是我的做法。同时也验证了那句话：学从难处学，用从易处用。

真正的程序员用 C++，聪明的程序员用 Delphi。那么，真正聪明的程序员用 C++ 来理解 Delphi！

## 1.2 Delphi 更多的优势

用过很多的主流开发工具，为什么还是选择了 Delphi？也许是因为没有深入地去熟悉



其他开发工具吧，但 Delphi 本身的优秀至少是原因之一！Delphi 优秀在何处？

### ◆ 开发的高效

Delphi 是一个 RAD (Rapid Application Development, 快速开发工具)，它有可视化的开发环境。当然具有类似功能的开发工具也不少（如 Visual Basic），但 Delphi 有如下的独到之处：

(1) Delphi 是真正面向对象的。其基于 OO 技术构建的 VCL 库中的所有组件都可以被继承以创建新的组件，包括窗体类 TForm。相比之下，ActiveX 组件缺乏这种灵活性。

(2) Delphi 的 CodeInsight 技术（即代码自动完成功能）是建立在编译器信息上的，而 VB 使用的是类型库信息。使用编译器信息的好处是更具灵活性。不过，时常有程序员抱怨 Delphi 的代码提示时间太长。其实，我个人感觉是习惯了其速度之后，能体会到一种节奏的快感。

### ◆ 语言的高效

Delphi 基于 Object Pascal 语言。这是一种真正支持面向对象而又优雅美观的语言。它在功能的健全上毫不逊色于各种其他的面向对象语言，但同时又不贪多，不盲目地增加复杂性。使得开发者运用各种模式进行设计时都能得到完善的支持，实现时却不用考虑太多语言/编译器细节。

### ◆ 编译的高效

可以说，Delphi 是 Windows 平台上最快的高级语言本地代码编译器。编译速度快有什么好处呢？快速的编译器可以使用户频繁地在修改代码和编译运行的状态间切换。至少，我自己已经非常习惯了这样的工作方式：运行程序看一下效果，退出程序修改少量代码再运行程序。而 Delphi 的编译器从来不会让我有等待的感觉。

### ◆ 执行的高效

Delphi 不但编译速度快，生成的目标代码的执行效率也非常高。Delphi 与 C++ Builder 使用的是同一个后端优化器，因此其生成的代码的效率跟优秀的 C++ 编译器生成的代码相同。

Delphi 生成完全本地代码，因此 Delphi 编译结果的可执行文件可以被独立执行、分发（对于“绿色软件”的开发，这一点十分重要），不需要其他运行库支持。当然，也可以选择动态链接编译，这样可以大大减小可执行文件的长度。不过，在这种情况下分发程序时，必须同时分发必要的运行库文件。

### ◆ 维护的高效

C++ 把许多决策权给了程序员，因此功能十分强大，但同时，要用 C++ 写出出色的面向对象的代码，就要求程序员具有一定的素质。而 Visual Basic 则根本没有提供面向对象的编程机制（VB6.0 及先前版本都是基于对象，而非面向对象）。而 Delphi 程序员虽然会在一定程度上被限制在 VCL 提供的框架中（当然，完全可以在 Delphi 中摆脱 VCL 编程），但相对来说，更容易建立良好设计的代码。代码框架的优良使得软件维护成本大大降低。

基于以上所有理由，我选择 Delphi！

## 1.3 本书主题

我们平时都会写很多代码，为公司、为自己或者为朋友。有时，为了验证自己的一个想法，或学习某一种技术，会写一些试验性的代码。这样的代码的生命周期很短，基本不需要维护，随意写一下就可以。但是，当真正要完成一个项目的时候，代码设计就显得非常重要了。因为这样的代码是需要长期维护，不断修改或增强的。设计凌乱的代码会使得维护非常困难或者根本不可能，修改这样的代码意味着产生更多的 bug 甚至是灾难。

因此，代码在被编写之前，需要先被设计。这里所说的设计并不是指功能设计，而是指程序员在编码之前先对代码框架的设计，以使得今后代码更容易被理解、被维护。

经常听到这样一种说法：程序员的编程寿命只到 35 岁。我却从不相信此类说法，也许 35 岁以后，实现能力（其实就是工匠能力）是有下降的可能，而设计能力是随着经验的增加不降反升的。这才是最宝贵的。

国外的软件开发小组，一般的骨干都是 40 岁上下的人，那些才是大师级的程序员，而所谓的过了 35 岁就不能当程序员的程序员似乎根本没有资格被称为程序员。

软件工程要将程序员变成编码员，变成流水线上的一环，设计工作由专门的设计师完成（如框架设计师）。也许，分工细化是趋势，但是，是满足于做编码员还是希望成长为设计师取决于我们的眼光及努力。

放开眼光，而不是将自己局限、沉迷于“实现高手”。实现能力是基础，有一定的实现能力才可能成长，但是，它只是必要条件，而不是充分条件。否则，就像爬到山腰就以为自己到了山顶，停滞不前了。那么，你只可能是编码员，你的编程寿命也只到 35 岁。在有了这样的眼光之后，希望本书可以助你起步。

国内出版的 Delphi 相关书籍，基本都是讲解实现的。本书的书名是《Delphi 高手突破》，那么，假设你现在已经是 Delphi 高手了。所以本书不会涉及太多的实现技巧，虽然也有实例代码，但重点在于其构架的设计，而并非实现的讲解。

至此，读者也许已经猜出本书的主题了：如何在 Delphi 中使用面向对象技术，构建良好设计的代码。

在我看来，写代码是艺术创作，优雅的代码可以自解释，而不需要过多的注释。当注释过多的时候，就该考虑设计是否合理了。写书应该也是艺术创作，如果能把自己的认识、经验艺术地告诉读者，而不需要过多的“注释”（浪费篇幅的废话），就非常成功了。我希望自己能做到，至少尽量吧。

## 1.4 小 结

我相信，走上编程这条路，对于我来说是必然的。能成为专业程序员，也是我所梦想

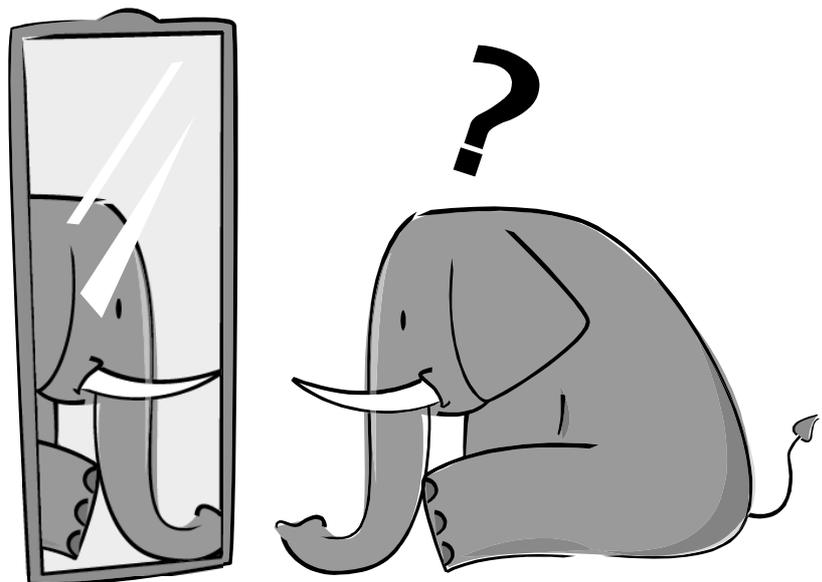


并实现了的。但是，Delphi 的出现以及被我所认识、熟悉、迷恋并成为工作的一部分，应该说是一个意外的惊喜。

在此，我所想说的就是，对于自己的志向，就更坚持一些吧。当你清醒地定位了自己之后，清楚地知道自己所选择的道路之后，就不用有所疑问、有所顾忌了，坚持走下去。最终虽然未必会成功（当然，每个人对成功的定义是不一样的），但爱我所爱，无怨无悔！

## 第 2 章 面向对象编程理论基础

面向对象是一种思维方式（理念），是一种方法论。



**面向对象并不困难！**



每个软件开发人员都会经常听到、看到“面向对象”这个词，程序员们也时常会把它挂在嘴上。

那么，什么是面向对象？什么是面向对象编程？是不是写几个类就算面向对象了？为什么要面向对象？因为别人都用，所以我也要用？

显然，并不是在程序中写了几个类就算面向对象编程了，用面向对象编程也并不是为了赶时髦。

“结构化编程”（SP）是一种编程方法，是用计算机的视角来抽象问题的方法。而“面向对象编程”（OOP）也是一种编程方法，它从更接近真实世界的视角来分析问题，使用更接近人们理解真实世界的方法来抽象问题，这种方法称为“面向对象”（OO）。

“面向对象”这个词代表的是一种认识世界、分析问题、解决问题的方法，因此它是一种方法论。而面向对象编程（OOP）则是将之应用于编程的方法。当用户会使用面向对象的方法去思考，用面向对象的模式去分析和解决问题的时候，才是真正的“面向对象”了。

本章会试图使用 Object Pascal 语言来告诉用户面向对象编程的理论知识，包括面向对象编程的特性以及这些特性在语言中的实现、在语义上对程序设计的影响。

## 2.1 类和对象的本质

“类”和“对象”是面向对象编程中最基本的概念，很多人都可以轻易地回答出什么是类，什么是对象：“类”是对一类事物的抽象（abstract），是创建对象的模板；“对象”是类的实例（instance）。

但这只是简单的概念解释而已，除此以外，还必须清楚类和对象的本质，即从语言和语义本身的角度来看，它们分别代表什么，它们是如何支撑起庞大的面向对象的世界的。

### 2.1.1 语言的“类”和“对象”

从语言的角度来说，“类”是用户自定义的数据类型，“对象”则是“类”类型的变量。类定义了所生成的对象的模板，于是也决定了对象所占用的内存空间。类成员分为两种：方法（即 C++ 中的“成员函数”，笔者个人比较喜欢“成员函数”的叫法，但“方法”已经是 Object Pascal 的一个术语。在 Delphi 相关的书中，称“方法”更合理一些，因此下文全部以“方法”称呼）和数据成员。数据成员表示类对象的状态，而方法则是改变类状态的操作。

当用 class 关键字声明一种类型时，就创造了一个类：

```
type
    TMyClass = class
end;
```

虽然在 `TMyClass` 中，还没有为它定义任何成员，但是，它的确是一个类，完全可以创建这个类的对象实例：

```
var
    MyObj : TMyClass;
begin
    MyObj := TMyClass.Create();
    .....
```

`Create()`是这个类的构造函数，它负责初始化对象数据成员。  
生老病死是人之常情，对象也一样会有被销毁的时候：

```
.....
MyObj.Free();
```

不过，`Free()`方法不是类的析构函数，`Free()`负责的是调用类的析构函数来销毁对象。至于采用这种机制的原因，稍后会讲述。

现在就来探讨一下 `Object Pascal` 中对象生存与销毁的秘密吧！

每个应用程序可以获得的内存空间分为两种：堆（`heap`）和栈（`stack`）。

堆又称为“自由存储区”，其中的内存空间的分配与释放是必须由程序员来控制的。例如，用 `GetMem` 函数获取了一定大小的内存空间，则在使用完后，必须调用 `FreeMem` 函数将空间释放，否则就会发生所谓的“内存泄漏”。“借债还钱，天经地义”。

栈又称为“自动存储区”，其中的内存空间的分配与释放是由编译器和系统自动完成的，不需要程序员过问。函数调用时按值传递的参数所占空间、函数中的局部变量等，都是在栈中被分配空间的。比如函数：

```
var
    i : Integer;
    j : Integer;
begin
    for i := 1 to 10 do
        .....
    .....
end;
```

其中，`i` 和 `j` 的空间是由编译器在栈中分配的。在函数末尾，也不需要程序员手动去释放这两个变量所占的内存空间。

`Objecgt Pascal` 遵循所谓的“引用/值”模型。无论在参数传递还是变量定义中，简单类型（如 `Integer`、`Cardinal`、`char` 以及 `record` 等）被按值传递或使用，其内存空间从栈中分配。



而复杂类型（class）则被按引用传递或使用，其内存空间从堆中分配。在 Object Pascal 中，所有对象（类类型的）都被建立在内存的堆空间上，而非栈上。因此在创建对象时，其构造函数不会被编译器自动调用，也没有 C++中所谓的“默认构造函数”。调用构造函数来创建对象以及调用析构函数来消灭对象都是程序员的职责。

如何为自己的类编写构造函数呢？在调用了诸如

```
MyObj := TMyClass.Create();
```

与

```
MyObj.Free();
```

之后究竟发生了哪些事情呢？

### ◆ 构造函数与对象内存的分配

定义构造函数使用 **Constructor** 关键字。按惯例，构造函数名称为 **Create**（当然也可以用其他名称，但那绝非优良的设计）。如：

```
type
  TMyFamily = class // 为你的家庭定义的类
  Private
    FMyFatherName : String; // 你父亲的名字
    FMyMotherName : String; // 你母亲的名字
    ..... // 你家庭中的其他成员
  Public
    Constructor Create(strFatherName, strMotherName : String);
    ..... // 其他方法
  End;
```

创建对象时则直接调用构造函数，形式如下：

```
MyFamilyObject := TMyFamily.Create('Zhang', 'Li');
```

也许有人会问，如果没有为自己的类提供构造函数，它的对象能否被建立呢？答案是：可以。在了解了构造对象的过程之后，就会明白为什么答案是“可以”。

要创建一个对象，首先需要分配对象本身所占用的内存空间，然后执行类的构造函数，以初始化各数据成员、申请对象需要的资源或创建其内部包含的子对象。

编译器在执行类似 `MyFamilyObject := TMyFamily.Create('Zhang', 'Li');` 这样的构造函数之前，会插入以下几行汇编代码：

```
test dl, dl
jz +$08
add esp, -$10
call @ClassCreate // 注意这行代码
```

以上代码的最后一行代码调用的是 `system.pas` 文件的第 8949 行的 `_ClassCreate` 函数(以 Delphi 6 为准)，该函数具体为每个对象分配合适的内存。这个动作也就是所谓的“编译器魔法”(Compiler Magic)，由这个动作完成真正的对象的内存分配，一个对象在这个时候已经有了外壳。

内存分配完成后是调用类的构造函数，即 `TMyClass.Create()`，以初始化数据成员。构造函数由定义类的程序员编写，也就是说，将对象初始化成何种模样是由程序员决定的。至此，一个对象已经诞生了。

之后，编译器会再插入以下几行汇编代码：

```
test dl, dl
jz  +$0f
call @AfterConstruction
pop dword ptr fs:[$00000000]
add esp, $0c
```

其中主要的工作是调用每个对象实例的 `AfterConstruction`。Borland 宣称这个调用在 Delphi 中没有用，它的存在是为 C++ Builder 所保留的。不过，由于 `AfterConstruction` 被声明为虚方法(virtual method)，因此完全可以利用它做一些善后的工作，尤其是在编写组件时。这些内容将在后续章节讲述，暂且不必理会。

可见，创建一个对象的步骤并不十分复杂，比之“十月怀胎”轻松了不知道多少倍。

由于对象本身所占内存的分配是由编译器完成的，因此即使没有构造函数，对象也一样可以被构造。构造函数的职责只是初始化对象的数据成员，没有构造函数只意味着不会对数据成员进行初始化而已，编译器会对所有数据进行清零初始化。此外，由于 Object Pascal 中所有类(除了 `TObject` 类本身)都是从 `TObject` 类派生，因此编译器会调用 `TObject.Create()` 构造函数。不过，这个函数只是一个空函数。

假如上面定义的 `TMyFamily` 类没有定义构造函数，则 `TObject.Create` 也不会对 `TMyFamily` 的数据成员 (`FMyFatherName`、`FMyMotherName`) 进行初始化，因为 `TObject.Create()` 根本就不认识你的父、母亲！

#### ◆ 析构函数与对象内存的回收

定义析构函数使用 `Destructor` 关键字。按惯例，析构函数名称为 `Destroy`。如：

```
type
  TMyClass = class
  Public
    Destructor Destroy(); override;
  End;
```

之所以在析构函数声明最后加上 `override` 声明，是为了保证在多态的情况下对象能正



确被析构（关于多态，将在 2.4 节中详述）。如果不加 `override` 关键字，则编译器会给出类似“Method ‘Destroy’ hides virtual method of base type ‘TObject’”的警告提示。警告的意思是用户定义的 `Destroy` 隐藏了基类的虚方法 `TObject.Destroy()`，那样的话，在多态的情况下就无法正确析构对象了（具体原因请查 2.4 节）。

 **注意：**析构函数都需要加 `override` 声明。

与构造函数类似，如果在类中没有特殊的资源需要被释放，也可以不定义析构函数，`TObject` 同样定义了一个空的析构函数。

在析构对象的时候，应该调用对象的 `Free()` 方法而不是直接调用 `Destroy()`。

```
MyFamilyObject.Free();
```

这是因为，在 `TObject` 的 `Free()` 方法中会判断对象本身是否为 `nil`，如果不为 `nil` 则调用对象的 `Destroy()`，以增加安全性。既然有这样更安全的做法，当然没有理由不这么做了。

 **注意：**永远不要直接调用对象的 `Destroy()`，而应该是 `Free()`。

要销毁一个对象，其顺序与创建对象正好相反。首先是释放对象申请的资源以及销毁内部的子对象，之后是回收对象本身所占的内存空间。

当程序执行到诸如 `MyFamilyObject.Free()` 这样的代码时，首先执行 `TObject.Free()` 方法：

```
procedure TObject.Free;  
begin  
  if Self <> nil then  
    Destroy;  
end;
```

在 `TObject` 的 `Free()` 方法中，调用了对象的析构函数。然后，编译器会在执行完 `Free()` 方法之后，插入以下几行汇编代码以完成第二个步骤（回收对象本身所占的内存空间）：

```
call @BeforeDestruction  
test dl, dl  
jle +$05  
call @ClassDestroy
```

这些代码所做的工作与构造对象分配内存时所做的是对应的，其中所调用的 `_ClassDestroy` 函数会精确地回收对象内存空间。

以下一个例程说明了如何使用构造函数和析构函数：

```
unit DllLoader;
```

```
interface
uses windows;

Type
  TDllLoader = class
  Protected
    // 之所以是 protected 成员，是为了在其派生类中具体实现加载某 DLL 时，派生类
    // 能够访问该句柄
    FhDLL : HMODULE;
  Public
    Constructor Create(strDLLName : String);
    Destructor Destroy(); override;
  End;

Implementation

Constructor TDllLoader.Create(strDLLName : String);
Begin
  FhDLL := LoadLibrary(strDLLName); // 构造函数中加载 DLL
  ASSERT(FhDLL <> 0);
End;

Destructor TDllLoader.Destroy();
Begin
  If FhDLL <> 0 then
  begin
    FreeLibrary(FhDLL); // 析构函数中释放 DLL
    FhDLL := 0;
  End;
End;
End.
```

#### ◆ 对象所占空间大小

前面为对象分配内存空间时谈到，每个对象会占用一定的内存空间，那么这个大小是如何确定的呢？

对象的大小，就是其数据成员所占用的内存空间的总和，其方法（函数）是不占用对象空间的。

不过，它不是一个简单的加法，还与编译器的“数据域对齐方式优化”有关，稍后会详述。



 **注意：**对象的大小只取决于其拥有的数据成员。

TObject 实现了一个 InstanceSize()方法，它可以取得对象实例的大小。下面以一个示例说明对象在内存中的布局。首先定义一个 TMyClass，其中包含 4 个数据成员和 1 个方法。先看一下类的定义：

```
Type
TMyClass = class
Public
    FMember1 : Integer;
    FMember2 : Integer;
    FMember3 : WORD;
    FMember4 : Integer;
    Procedure Method();
End;
```

然后，在 Application 的主 Form 中放入一个 Memo 和一个 Button，并在 Button 的 OnClick 事件中写下在 Memo 中显示出对象位置的代码。该程序源代码清单如下：

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms,
    Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Memo1: TMemo;
        Label1: TLabel;
        procedure Button1Click(Sender: TObject);
    end;

    // 自定义的 TMyClass 类
    TMyClass = class
    Public
        FMember1 : Integer;
        FMember2 : Integer;
        FMember3 : WORD;
```

```
    FMember4 : Integer;
    Procedure Method();
End;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
    Obj : TMyClass;
begin
    Obj := TMyClass.Create();

    with memo1.Lines do
    begin
        Add('对象大小: ' + IntToStr(Obj.InstanceSize));
        Add('对象所在地址    : ' + IntToStr(Integer(Obj)));
        Add('FMember1 所在地址: ' + IntToStr(Integer(@Obj.FMember1)));
        Add('FMember2 所在地址: ' + IntToStr(Integer(@Obj.FMember2)));
        Add('FMember3 所在地址: ' + IntToStr(Integer(@Obj.FMember3)));
        Add('FMember4 所在地址: ' + IntToStr(Integer(@Obj.FMember4)));
    end;

    Obj.Free();
end;

{ TMyClass }

procedure TMyClass.Method;
begin
    //no code
end;

end.
```

Button 的 Click 事件中所做的事情是，首先创建 TMyClass 类的实例，然后将对象大小以及每个数据成员的地址输出到 Memo 中。

该程序的代码和可执行文件可在配书光盘的 ObjectSize 目录下找到，运行程序并单击“开始”按钮后，其界面如图 2.1 所示。



图 2.1 ObjectSize 程序界面

图 2.1 的 Memo 中显示出了想要的结果。也许读者会看不清图片中的结果，不妨介绍一下：其中显示了对象大小为 20 个字节，对象所在首地址是 13443352（也许每次运行对象被创建的地址会不同，这没有关系，在此主要关心的是各地址之间的差值），FMember1 所在地址是 13443356，FMember2 为 13443360，FMember3 为 13443364，FMember4 为 13443368。现在来分析一下：

根据对象首地址以及大小可以算出，对象占用的内存空间范围为 13443352 ~ 13443371。而第一个成员却在 13443356，它与对象的首地址之间有一个 4 字节的空缺，这 4 个字节存放的是一个指向对象的 VMT（虚方法表）的指针。关于 VMT 将在 2.4 节多态的本质中详细讨论，此处暂且不表。

13443356~13443359 这 4 个字节即 FMember1 所占空间(32 位整数)。同样，13443360~13443363 为 FMember2 所占空间。比较容易令人疑惑的是 FMember3，计算可知，它所占地址范围为 13443364~13443367，同样也是 4 个字节，但在此定义的 FMember3 其实是 Word 类型（16 位），为什么它会占用 32 位空间呢？这与编译器的字节对齐优化有关，编译器会将无法合并的小于 32 位空间的数据域填充到 32 位大小，以加快存取速度。也就是说，FMember3 同样需要占用 4 个字节空间。可以自己试一下，如果将以上 TMyClass 类定义中的 FMember2 也改成 Word 类型，编译器会把 FMember2 和 FMember3 合并成一个 32 位空间，于是对象大小就变成了 16。

FMember4 所占的空间没什么意外，为 13443368~13443371。

整个对象的内存布局如图 2.2 所示。

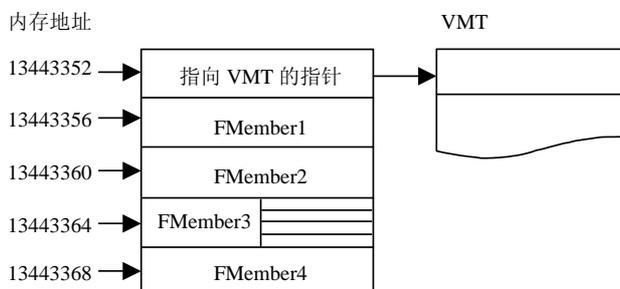


图 2.2 对象内存布局图（阴影部分为字节对齐优化时被填充的区域）

另外，也可以看到，TMyClass 类中惟一的方法 Method() 没有在对象的空间中出现。

#### ◆ “类方法”与“类引用”类型

一般所称的“方法”，都是指“对象方法”。也就是说，执行该方法，将可能导致对象的状态发生改变，即该方法可以更改对象的数据成员的值。如：

```
TMyClass2 = class
private
    FMember : Integer;
public
    procedure SetValue(Value : Integer);
end;

procedure TMyClass2.SetValue(Value : Integer);
begin
    FMember := Value;
end;
```

其中 SetValue 即为典型的对象方法。

除了“对象方法”外，还有所谓的“类方法”，也就是属于类级别的函数（而非对象级别的）。它可以改变类的状态（而非对象的状态）。

定义类方法，只需要在一般的方法声明前加上 class 关键字。如：

```
class function TObject.ClassName: ShortString;
```

既然类方法是进行类级别的操作，因此在类方法中是无法对对象的数据成员进行访问的。

在 Object Pascal 中，还有一种“类之类”的类型，也就是所谓的“类引用”。一般所称的类，是对其实例对象的抽象。定义一个类：

```
TMyClass = class;
```

而“类引用”类型却是对“类”的抽象（元类），所以被称为“类之类”。定义一个“类之类”：

```
TMyClassClass = class of TMyClass;
```

“类之类”可以直接调用“类”的“类方法”。如：

```
TMyClass = class
public
    class procedure Show();
end;
TMyClassClass = class of TMyClass;
```



```
var
    MyClass : TMyClassClass;
    MyObj : TMyClass;
begin
    MyObj := MyClass.Create();
    MyClass.Show();
    MyObj.Free();
end;
```

在此例中，TMyClassClass 作为 TMyClass 的元类，可以直接调用 TMyClass 的类方法。此前提到过的类构造函数，其实就是一个类方法，因此可以如同

```
MyObj := MyClass.Create();
```

来创建对象，其结果与

```
MyObj := TMyClass.Create();
```

完全相同。

但是，析构函数则不是类方法，而是普通的对象方法。因为析构函数只能销毁一个对象实例，其操作结果并非作用于该类的所有对象。因此，销毁对象只能通过对象来调用析构函数而不能通过类方法：

```
MyObj.Free();
```

“类方法”和“类引用”有什么作用呢？它主要用在类型参数化上，因为有时在编译时无法得知某个对象的具体类型，而需要调用其类方法（如构造函数），此时可以将类型作为一个参数来传递。在 Delphi 6 的帮助文档中，有这样一个例子：

```
type TControlClass = class of TControl;

function CreateControl(
    ControlClass: TControlClass;
    const ControlName: string;
    X, Y, W, H: Integer
): TControl;
begin
    Result := ControlClass.Create(MainForm);
    with Result do
    begin
        Parent := MainForm;
        Name := ControlName;
        SetBounds(X, Y, W, H);
    end;
end;
```

```
Visible := True;
end;
end;
```

CreateControl 函数具体创建一个控件对象，但是，由于它在编译时期并不知道需要其创建的对象的具体类型，因此其第一个参数 ControlClass 的类型是 TControl 的类引用类型——TControlClass，这样就可以将所需要创建控件的类型延迟到运行期去决定。例如，在运行期要创建一个 TButton 类型的控件对象：

```
var
    Btn : TButton;
begin
    Btn := CreateControl(TButton, 'Button1', 0, 0, 100, 20);
    .....
end;
```

还有，经常可以在 Delphi 生成的 Application 的 project 文件中找到这样的代码：

```
Application.CreateForm(TForm1, Form1);
```

TApplication 的 CreateForm()方法的第一个参数，也是类引用类型的：

```
procedure TApplication.CreateForm(
    // TComponentClass = class of TComponent; 类引用类型
    InstanceClass: TComponentClass;
    var Reference
);
```

允许在运行期确定类型，可以给程序带来莫大的灵活性。

在 Object Pascal 中，类方法中还可以使用 self。不过，此时 self 表示的是类，而非对象，因此使用上也有一些限制。

如果是通过对象引用调用类方法，则 self 的值是该对象的类型；如果是通过类名调用类方法，则 self 的值是该类本身。

由于在类方法中，self 的值是类，而非对象，因此只能通过 self 调用类的构造函数和其他类方法。

以下的代码展示了类方法中 self 的使用方法：

```
interface
Type
    TClassMethodExample = class
```



```
private
    FnInteger : Integer;
public
    class function ClassMethod1() : Integer;
    class function ClassMethod2() : Integer;
    function Method() : Integer;
end;

implementation

{ TClassMethodExample }

class function TClassMethodExample.ClassMethod1: Integer;
begin
    self.Method() ;    // 非法, 因为 Method 不是类方法
    self.ClassMethod2(); // 合法
end;

class function TClassMethodExample.ClassMethod2: Integer;
begin
    Result := self.nInteger; // 非法, 类方法中不能访问对象数据成员
end;

function TClassMethodExample.Method: Integer;
begin
    Result := 0;
end;
end.
```

## 2.1.2 语义的“类”和“对象”

从语义的角度来说，“类”是对一种逻辑的抽象，而“对象”则是这种逻辑的具体实例。带一些感情色彩地说，“类”与其他数据类型相比，更像是活的，具有生命特征。一旦类的对象被构造了，这个对象就具有了生命，如同现实中的一个社会成员。它有自己的生存空间，有自己的私有特性，能向用户提供服务，还可以和用户交流……

那么，是否每时每刻都要定义类？如果不是，那么何种情况下需要定义一个类，何种情况下不需要呢？

要明确设计类的目的——简化类的客户（使用者）的工作，提供一个良好的、明晰的接口供客户访问。类封装一个逻辑，或一个事务。将事务的复杂的逻辑隐藏于类的实现中，可以降低客户端编程的难度以及方便类的实现算法的更改。

首先，类是一道带门的墙，它保护一些东西、隐藏一些东西，同时对外开放一些东西。不能将墙内的所有东西都暴露出来，也不能关闭大门，封闭所有。隐藏实现细节，提供明晰接口是类的第一要务。

其次，类是具有生命性质的。因此，类的对象必须具有状态，这些状态通常被设计为数据成员。如果仅仅是一堆函数集合所构成的类，这样的设计是非常糟糕的（当然，作为接口的抽象类除外）。

最后，类是单一逻辑的体现。一个功能强大、接口庞大的类是不利于被理解、维护和重用的。

可以来看一些例子。

以下是一个获取系统信息的类：

```
Type
TSystemInfo = class      //系统信息类
Public
    Function GetSystemTime() : TDateTime; // 获取系统时间
    Function GetSystemDir() : String;     // 获取系统所在目录
    Function GetSystemVer() : Integer;    // 获取系统版本
    .....
End;
```

这样的设计怎么样呢？只能说，非常糟糕！为什么？看上去，它“封装”了系统相关的一系列操作，但其实什么都没有封装！它只是一堆函数的集合而已，这是一个没有状态（数据成员）的类，或者说是“行尸走肉”。不！它不是活的，它所生成的每个对象之间没有任何差别！

 **注意：**类必须有表示其实例对象的状态的数据成员。

现在还觉得有必要定义这个类并创建这个类的实例对象吗？定义一套获取系统信息相关的函数库（API）才是一个不错的主意。

好，那下面这个类呢？

```
Type
TBadClass = class
Private
    AnInteger : Integer;
    AString : String;
Public
    Function GetAnInteger() : Integer; //获取 AnInteger 的值
    Procedure SetAnInteger(nValue : Integer); //设置 AnInteger 的值
    Function GetAString() : String; //获取 AString 的值
    Procedure SetAString(strValue : String); //设置 AString 的值
End;
```



同样，只能说这还是一个非常糟糕的设计！虽然这个类有了数据成员，从此它的对象具有了状态，这是一个不小的进步。但这还远远不够，类不仅仅是一组允许客户获取和设置其中数据的值的函数的集合，类的接口还必须能够简化客户访问它的逻辑。只有“GetXXX”和“SetXXX”的函数的类，称不上类，因为它没有隐藏任何东西，也没有简化任何东西，虽然它把数据成员写在了 `private`，但它还是将一切都暴露给了外部。

可以看一下这个类的客户端代码可能是这样的：

```
var
  I : Integer;
  O : TBadClass;
begin
  O := TBadClass.Create();
  O.SetAnInteger(2);
  I := O.GetAnInteger();
  O.Free();
end;
```

试想，这样的“类”与“记录”类型有何区别？

```
Type
  TBadClassRec = record
    AnInteger : Integer;
    AString : String;
  End;
```

该记录类型的客户端代码可能是这样的：

```
var
  I : Integer;
  O : TBadClassRec;
begin
  O.AnInteger := 2;
  I := O.AnInteger;
end;
```

发现了吗？类没有对任何数据进行保护，对客户的编程也没有做任何简化，相反还多出了创建和析构对象的步骤。因此这个类的设计是失败的。



**注意：**在面向对象的世界中，类的使用者比类的构造者要多得多，如果想让别人用自己的类，就要让类容易被使用。

那么什么样的类才是设计优良的类？

```
Type
  TFile = class
  Private
    FFileName : String;
    FFileOpened : Boolean;
    FFileText : TStringList;
    ..... // 可能的其他 private 数据成员/方法
  Public
    Constructor Create(strFileName : String);
    Destructor Destroy(); override;

    Procedure Open();
    Procedure Close();
    Procedure Save(const FileText : TStringList);
    Procedure GetText(const FileText : TStringList);
  End;
```

这样一个文件操作的类和其他一切设计良好的类一样具有某些共同的特征：具有状态信息，接口简单、明了，易用。它封装了所有对文件操作的算法，客户端程序不必陷于复杂的文件操作的代码中。

定义良好设计的类是构架良好设计的程序的基础，类如同造房的砖块，砖块制作粗糙，如何能指望房子坚固呢？

## 2.2 封装

封装，是抽象数据类型（或基于对象）的特性。似乎一谈到对象，能立刻想到的就是封装。因为很容易就能把对象理解成所谓的黑匣子。

为什么要封装？

可以把程序按某种方法分成很多“块”，块与块之间可能会有联系。每个块都有一个可变的和一个稳定的部分。我们需要把可变的和稳定的部分分离开来，将稳定的部分暴露给其他块，而将可变的隐藏起来，以便于随时可以让它改变。这项工作就是封装！

例如，在用类来实现某个逻辑时，类就是以上所说的“块”，实现功能的具体代码就是“可变的”，而 **public** 的方法（称为“接口”）则是“稳定的部分”。

在 **Object Pascal** 中，实现了两个级别的封装：类级和单元级。



## 2.2.1 类级别的封装

类级别的封装是最常见的封装形式。

每个 Object Pascal 的类，有四种访问级别：`private`、`protected`、`public`、`published`。其中，`public` 的成员可以被外界的所有客户代码直接访问；`published` 和 `public` 差不多，区别仅在于 `published` 的成员可以被 Delphi 开发环境的 Object Inspector 所显示，因此一般将属性或事件声明于 `published` 段；`private` 成员为类的私有性质，仅有类本身和友元可访问；`protected` 成员基本与 `private` 类似，区别在于 `protected` 可以被该类的所有派生类访问。

在类级别的封装中，对外界的接口是 `public` 方法和 `published` 成员的集合，`private` 和 `protected` 的集合则属于类的实现细节。而对于该类的派生类来说，接口是 `public`、`published` 与 `protected` 的集合，而只有 `private` 部分为内部实现细节。

## 2.2.2 单元级别的封装

单元级别的封装包含的含义有：

1. 在一个 Unit 中声明的多个类，互为友元类。
2. 在一个 Unit 的 `interface` 部分声明的变量为全局变量，其他 Unit 可见。
3. 在一个 Unit 的 `implementation` 部分声明的变量为该 unit 的局部变量，只在该 Unit 可见。
4. 每个 Unit 可有单独的初始化段（`initialization`）和反初始化段（`finalization`），可在编译器支持下自动进行 Unit 级别的初始化和反初始化。

Object Pascal 规定，声明在同一个 Unit 之中的多个类互为友元类，友元类之间可以互相访问所有数据，无论是 `public` 的，还是 `private` 的，或者是 `protected` 的。也就是说，友元类之间没有秘密。如下面的两个类：

```
type
  TFriend1 = class
  private
    FMember1 : Integer;
  end;

  TFriend2 = class
  private
    Friend : TFriend1;
  public
    function GetFriendMember() : Integer;
  end;
```

TFriend1 和 TFriend2 之间可以互相访问私有数据成员：

```
function TFriend2.GetFriendMember: Integer;
begin
    Result := Friend.FMember1; // 访问了 TFriend1 的 private 数据
end;
```

虽然 FMember1 是 TFriend1 类的 private 数据，但在 TFriend2 中可以访问，这是合法的。粗看起来，友元类似乎破坏了封装。但其实适当地使用友元的特性，可以增强封装性。

有时一个类的两部分可能会具有不同的生命周期，也许用户会将这两部分拆分成两个相关的类，此时两个类之间可能会互相访问彼此的数据，而数据成员一般都被置于 private 节中。如果避免使用友元，则只能要么将数据置于 public 节中，要么提供 GetXXX、SetXXX 之类的方法。将数据置于 public 的做法是非常罕见的，而提供 GetXXX、SetXXX 之类的方法也绝非优良设计，这些做法其实都破坏了封装性。如何保持封装性呢？答案就是使用友元！

Object Pascal 的单元文件被分成了两个部分：interface 和 implementation。如同类的封装一样，Unit 的这两部分分别为接口和实现细节。因此，interface 部分对外是可见的，声明在 interface 段中的所有函数、过程、变量的集合，即单元文件作为一个模块的对外接口，而 implementation 部分对外是隐藏的。

而为单元文件提供初始化和反初始化机制，则保证了单元的独立性，其作用如同类的构造函数与析构函数，单元的运作由此便可脱离对其他模块的依赖。

以下是一个完整的 Unit 示例：

```
unit UnitDemo;

interface

uses Windows;

    procedure Proc1(); // 某功能函数
    procedure InitUnit(); // 单元初始化函数
    procedure UnInitUnit(); // 单元反初始化函数

var
    g_nGlobalVar : Integer; // 全局变量

implementation

var
    l_nLocalVar : Integer; // 单元级别的局部变量

procedure InitUnit();
```



```
begin
  l_nLocalVar := 0;
  ..... // 其他初始化工作
end;
procedure UnInitUnit();
begin
  ..... // 反初始化
end;

procedure Procl();
begin
  ..... // 一些代码
end;

initialization      // 初始化段
  InitUnit();        // 调用 InitUnit() 以初始化单元

finalization        // 反初始化段
  UnInitUnit();

end.
```

无论是单元的封装，还是类的封装，封装的目的都是一样的，即简化用户接口，隐藏实现细节。正如 2.1.2 小节语义的“类”和“对象”中所述，封装的难点在于如何设计接口。

首先，必须保证接口是功能的全集，即接口能够覆盖所有需求。不能完成必要功能的封装是毫无意义的。

其次，尽量让接口是最小冗余的。这是为了简化客户的学习，难用的封装是容易被人遗忘的。冗余接口的存在是被允许的，但必须保证冗余接口是有效的。也就是说，增加这个冗余接口会带来非常大的好处，比如性能的飞速提升。

最后，要保证接口是稳定的。将接口和实现分离，并将实现隐藏，就是为了能保护客户的代码在功能实现细节改变的情况下，不必随之改变。三天两头改变接口的封装是惹人讨厌的。记住一个原则：一旦接口被公布，永远也不要改变它！

绝大多数失败的设计，都来自于失败的封装！

## 2.3 继承的本质

继承是为了表现类和类之间的“是一种”关系。有了继承之后，构建多层次的类框架成为可能。同时，它也是面向对象中的另一个核心概念——多态的存在基础。

因此，继承是面向对象语言必不可少的特性，只支持封装而不支持继承的语言只能称

为“基于对象”（Object-Based）而非“面向对象”（Object-Oriented）。

在利用语言提供的继承特性之前，有必要先了解一下语言本身关于继承的一些特性及实现。

### 2.3.1 语言的“继承”

首先要了解从语言层次的视角对“继承”概念的理解与语言对其的实现支持。

继承关系也被称为派生。继承的关系中，被继承的称为基类；从基类继承而得的，称为派生类。比如说，类 B 从类 A 继承而得，则 B 为派生类，A 为基类。在 Object Pascal 语言中，定义继承关系的语法：

```
TB = class(TA)
```

表示 TB 从 TA 继承（派生），TB 是派生类，而 TA 为基类。

Object Pascal 只支持 C++ 中所谓的 public 继承，即派生类中基类的 public 成员在其中仍然是 public 的，基类的 protected 成员在派生类中仍然是 protected 的，派生类无法访问基类的 private 成员。Public 继承在语义上严格地奉行“是一种”关系。也就是说，类 B 若派生自类 A 的话，那么在任何时候，都可以称“B 是一种 A”。因此，在设计继承层次时，也应该注意，如果 B 不是在任何时候都可以被当作 A，那么就不可以将 B 从 A 派生。

Object Pascal 只支持单继承，即每个派生类只能有一个基类，由此可以保证每个派生类中，只有惟一一份基类子对象。

也许有些读者还不清楚什么是基类子对象，或者不清楚上面这句话的具体含义是什么。下面就来介绍一下。

在 2.1 节类和对象的本质中曾经提到过，对象所占的内存空间大小取决于这个对象中的数据成员。也就是说，每一个对象实例中，都包含了它所有的数据成员。更进一步，在允许继承的情况下，每个派生类的对象实例所占内存空间的大小，不但取决于自身的数据成员，还要加上其基类的数据成员。

每一个类的实例对象所占的内存空间，是其自身的数据成员与其所有基类（因为基类可能还有基类）的数据成员（不论是 private 的，还是 public 的）所占内存空间的总和（不考虑“按位对齐优化”的情况）。

每一个派生类的实例对象，内部都包含了一个完整的基类实例对象，这个完整的基类实例对象，就称为“基类子对象”，因为基类的对象永远小于或者等于派生类的对象。

虽然派生类对象无法访问基类子对象中的 private 的数据，但是，这些数据是的确存在并且占用内存空间的。

下面以一个示例程序来说明派生类对象和基类子对象的关系以及它们的内存布局情况。该程序源代码和可执行文件可在配书光盘的 inherit 目录下找到。

先定义一个三层的继承层次：

```
type  
  TBase = class
```



```
public
    FBaseMember1 : Integer;
    FBaseMember2 : Integer;
end;

TDerived = class(TBase)
public
    FDerivedMember : Integer;
end;

TDerived2 = class(TDerived)
public
    FDerived2Member1 : Integer;
    FDerived2Member2 : Integer;
end;
```

TBase 是基类，TDerived 派生自 TBase，因此它是 TBase 的派生类，但由于 TDerived2 派生自 TDerived，因此，TDerived 同时也是 TDerived2 的基类，而 TDerived2 是 TDerived 的派生类。

在此定义了一个三层的继承层次，但由于成员方法是不占用对象实例的内存空间的，因此，为方便说明起见，不定义方法成员。

定义完相关类后，在 Application 的主 Form (Form1) 上放上一个 ListBox (name 为: lst\_rs) 和一个 Button。然后在 Button 的 OnClick 事件中加入代码，使得对象位置信息显示在 ListBox 中。

该程序含有两个单元，名称为 clsinherit.pas 的单元中仅定义了 TBase、TDerived、TDerived2 3 个类（如前定义），另一个为主 Form 的代码单元，其代码清单如下：

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms,
    Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        lst_rs: TListBox;
        Label1: TLabel;
    end;
```

```
    procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

uses clsinherit;

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
    Obj : TDerived2;
begin
    Obj := TDerived2.Create();
    with lst_rs.Items do
    begin
        Add('对象大小: ' + IntToStr(Obj.InstanceSize));
        Add('对象首地址: ' + IntToStr(Integer(Obj)));
        Add('TBase 成员首地址: ' + IntToStr(Integer(@Obj.FBaseMember1)));
        Add('TDerived 扩展成员 (FDerivedMember) 首地址: ' +
            IntToStr(Integer(@Obj.FDerivedMember)));
        Add('TDerived2 扩展成员 (FDerived2Member1) 首地址: ' +
            IntToStr(Integer(@Obj.FDerived2Member1)));
    end;
    Obj.Free();
end;
end.
```

**Button1Click()**方法创建一个 **TDerived2** 类的一个实例对象，然后将对象首地址、对象大小及其所有数据成员（包括从基类中派生而得来的数据成员）的地址在 **ListBox** 中显示出来。

运行程序并单击“开始”按钮，程序结果如图 2.3 所示。



图 2.3 派生类内存布局演示程序界面

结果在图 2.3 中也显示了：对象大小为 24 字节；首地址为 13443344（也许在各位的计算机上运行该地址值有所不同，没有关系，在此只关心这些地址值之间的差值）；TBase 成员首地址为 13443348；TDerived 扩展成员首地址为 13443356；TDerived2 扩展成员首地址为 13443360。

推算可知，对象所占内存地址范围为 13443344~13443367。TBase 成员首地址和整个对象首地址之间存在 4 个字节的差值，这个空缺还是那个指向 VMT 的指针（将在 2.4 节多态的本质中详述）。TBase 的两个整型数据成员占用 8 个字节，因此 TDerived 的 FDerivedMember 的首地址就是 13443356 了。同理，再步进 4 个字节，就是 TDerived2 的 FDerived2Member1 的首地址了。

根据上面的演算，可以画出 Obj 对象的内存布局图，如图 2.4 所示。

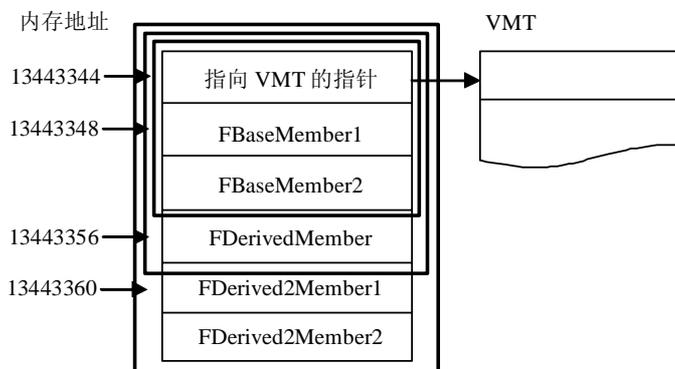


图 2.4 派生对象内存布局

（图中 3 个深色矩形框，由里向外分别表示 TBase、TDerived、TDerived2 的完整实例）

从图 2.4 中可以清楚地看到，Obj 对象（最外层的深色矩形框）中完整地包含了 TDerived 类的实例对象（中间层的深色矩形框）和 TBase 类的实例对象（最内层的深色矩形框）。最内层的矩形框表示了 TDerived2 类实例对象中的 TBase 基类子对象，中间层的矩形框表示了 TDerived2 类实例对象中的 TDerived 基类子对象。

**注意：** 每个基类子对象都是完整的。

需要特别说明一下的是，此处为了便于 Button 的 OnClick 事件中的代码可以直接访问类的所有数据成员并将它们的地址打印出来，因此在定义类时，将所有数据成员都声明为 public。但实际程序运行结果（指每个数据成员所在地址）并不依赖于它是处于 public 的还是 private 的。也就如同上面所说的，即使派生类对象无法访问基类子对象中的 private 的数据，它们依然是存在并占用内存空间的，无法访问它只是因为编译器为它做了额外的保护。

### 2.3.2 语义的“继承”

了解语言对于继承的理解与实现支持，对于设计是有所助益的。但是，设计更多的时候是根据语义的。

语义上的“继承”，更多的是作为一种“特化”的机制。也就是说，能够被继承的类（基类）总是含有并且只含有所抽象的那一类事物的共性，当需要抽象该类事物中的某一种特例时，将表示特例的类从基类继承（派生），派生类含有这类事物的所有共性，并且自己拥有特性。

例如，“水果”这个概念抽象了很多事物，这些事物有一些共性，如可以食用、属于农作物等。“苹果”这个概念表示了一种特殊的水果，它作为“水果”的特例，包含水果的一切属性——可以被食用、属于农作物等，同时它也包含自己的特性——果实为圆形、味道甜美等。

语义上的“继承”表示“是一种”的关系，派生类可以被看作“是一种”基类，这是一个最基本的、必须满足的前提。正如苹果是一种水果这么理所当然。在设计类关系时，可以将若干类的共性抽象出来，集中在它们的基类中实现。但如果类 A 不是一种类 B，也就是说，A 不能无条件地出现在 B 的位置上取代 B，那么无论如何，不要把 A 设计成 B 的派生类。这被称为“多态置换原则”。

在此需要特别指出一个最容易使设计者混淆的问题，那就是“容器”问题。必须牢记一个法则：当 A 是一种 B 时，那么 A 的容器（绝对）不是一种 B 的容器！

将苹果与水果代入以上法则：苹果是一种水果，苹果袋（苹果的容器）不是一种水果袋（水果的容器）！

看上去，这与常识有些不一样，然而，这是事实，至少在面向对象设计领域！也许一时还无法接受它，但要先强迫自己接受它，然后再看以下的分析。

先说明概念，在上面所定义的苹果袋中，只能存放苹果，如果可以放入其他水果，那就叫水果袋了。

根据“多态置换原则”，任何出现“水果”的地方，可以用“苹果”进行替换。例如，水果是可以食用的，水果是农作物……

这些都可以变换成：苹果是可以食用的，苹果是农作物……

如果有人对你说：“请给我一个水果”，那么当你递给他一个苹果时，他应该不会有任何意见，还会对你说一声“谢谢”。

现在假设，如果苹果袋是一种水果袋成立的话，同样根据“多态置换原则”，任何出



现水果袋的地方，也都可以用苹果袋取代。

“水果袋可以放任何水果”就变成了“苹果袋可以放任何水果”，这个结论显然让人无法接受，它违背了我们最初的定义。

如果有人对你说：“请给我一个水果袋，我要放一些香蕉在里面”，这样的要求非常合理，因为香蕉是被允许放入水果袋的。然而，如果你递给他一个苹果袋的话，他就会有意，因为他无法把香蕉放进去！这时也就无法指望他会对你说“谢谢”了。

也许这个例子还比较温和一些，那么再举一个后果严重一些的例子。

我们都相信，“可乐”是一种“液体”，那么你是否会相信“可乐罐”是一种“液体储存罐”呢？

“可乐罐”被定义为只能存放可乐的容器，而“液体储存罐”可以存放任何液体。

如果你坚信“可乐罐”是一种“液体储存罐”，那么可能有如下的代码定义“液体储存罐”类和“可乐罐”类：

```
TLiquidBottle = class // 液体储存罐
    // ..... 省略
public
    // 关于“virtual;”的话题，将在下一节详述
    // 倒出其中的液体
    procedure Spill(); virtual;
    // 向其中装入某种液体
    procedure Contain(const Liquid : TLiquid); virtual;
end;

TCokeBottle = class(TLiquidBottle) // 可乐罐
    // ..... 省略
public
    procedure Spill(); override;
    procedure Contain(const Liquid : TLiquid); override;
end;
```

另外有一个将液体灌入储存罐的函数和一个将某种“液体储存罐”中的液体倒出的函数：

```
procedure Contain(const Liquid : TLiquid; var Bottle : TLiquidBottle);
begin
    Bottle.Contain(Liquid);
end;

procedure Spill(var Bottle : TLiquidBottle);
begin
```

```
Bottle.Spill();
end;
```

好了，现在可乐罐是一种液体储存罐了，从此可以安全、合法地将汽油倒入可乐罐中：

```
Contain(AGas, ACokeBottle); // 编译器完全接受
```

能够安心、合法地这么做，只因为可乐罐是一种液体储存罐！

今后，也许某天打开这个可乐罐准备享受甜美的可乐时，却发现倒出的是汽油，恐怕也只能悻悻作罢了。

 **注意：**一种事物的容器，“不是一种”任何其他的容器！

这就是著名的“容器”问题。关于“多态置换原则”，还有一个和日常生活常识相悖的例子，那就是“圆/椭圆”问题。

中学的数学老师和课本告诉我们，“圆”是一种“椭圆”，然而 OOP 却说，“圆不是一种椭圆”。同样，先强迫自己接受这个观点，然后再看原因。

“圆”如果是一种“椭圆”，那么“椭圆”一定具有比“圆”更普遍的性质，而“圆”则是特化的“椭圆”，因此圆必须能做到所有的椭圆都能做到的事情。

“椭圆”允许被设置“高”和“宽”（或“长半径”和“短半径”），也就是说，椭圆的高和宽允许被分别设置，而不需要一定相等。“圆”则不允许这样做，“圆”的“高”和“宽”必须相等（否则就成了椭圆了）。

如果有一个描述椭圆的类：

```
TEllipse = class
public
    // .....
    // 设置椭圆大小，x、y 分别为高和宽
    procedure SetSize(x, y : Integer);
end;
```

如果还是相信自己数学老师，认为“圆是一种椭圆”，那么可以让圆从椭圆派生：

```
TCircle = class(TEllipse)
```

这时发生了什么？TCircle（圆）从TEllipse（椭圆）继承而得到SetSize()方法，但这个方法是被用来设置椭圆的高和宽的，x和y两个参数不必相等。但是它出现在TCircle中，这未免会引起使用者的迷惑。

发生这样的问题，原因在于椭圆拥有的能力已经无法完全包含在圆所拥有的能力之中，这时就无法满足“多态置换原则”。此时，OOP会告知：“圆不是一种椭圆”！

其实，数学老师和OOP都没有错。从数学的角度来说，圆的确是一种椭圆，因为数学上的圆具有数学上的椭圆的全部特性。但从OOP的角度来说，圆不是一种椭圆！因为OOP



的椭圆具有 OOP 的圆不具有的能力——SetSize()。

解决圆和椭圆的关系问题，只有两种方法：

(1) 让圆 (TCircle) 和椭圆 (TEllipse) 毫无关系。

(2) 让圆 (TCircle) 和椭圆 (TEllipse) 共同由另一个不含 SetSize()方法的基类 (如 TOval) 派生。

“圆/椭圆”问题之所以引人注目，是因为它是许多不良设计产生的原因，很多失败的设计都可以归结为“圆/椭圆”问题。也就是说，失败的继承关系设计，总是让基类拥有比派生类更多的额外的能力，哪怕是一个函数/方法。

这里所说的“是一种”关系，是“普遍”与“特殊”的关系，是“共性”与“个性”的关系。因此，记住总是弱化你的基类，强化你的派生类，总是让派生类比基类更强大。要真正理解“圆/椭圆”问题！

 **注意：**使得基类弱一些，派生类强一些。

无论是“容器”问题，还是“圆/椭圆”问题，都是因为其中的“是一种”的概念与日常生活中的“是一种”的概念不同。在继承关系中，“是一种”的含义以及判断标准为是否完全符合“多态置换原则”——即在任何情况下，派生类事物都能无条件地取代基类事物。

以上谈了“多态置换原则”，在继承关系的设计中，除了“多态置换原则”，有时设计的优良还与项目的具体需求有关。一个看上去很好的设计，也许在某一个具体项目中，其结果并非最佳；同样，一个看上去有缺陷的设计，也许正符合某项目的要求。

例如，设计一个描述人的类：

```
Type
TPersion = class
Private
    FName : String; // 姓名
    FAddr : String; // 家庭地址
    .....
public
    constructor Create(strName : String);
    destructor Destroy(); override;

    procedure Walk(nStep : Integer); // 走路
    procedure GoHome(); // 回家
    procedure HaveLesson(); // 上课? 或许不应该有这个特性吧
    .....
end;
```

这样描述的“人”的类中每个“人”的对象可以走路、回家（假设每个人都有家可回，

流浪汉不在讨论之列)，这些都很合理。但是加上“上课”这个特性，就值得商榷了。应该说 HaveLesson 这个特性是不应该出现在描述“人”的类中的。

如果从 TPerson 派生出描述学生的类 TStudent:

```
type
  TStudent(TPerson)
  Private
    FMajorIn : String; // 专业
  Public
    Procedure HaveLesson(); // 学生上课, 天经地义
    .....
End;
```

这样的描述体系就似乎比较合理了。

但是，如果正在开发的系统是一个学生世界，在这个世界中每个人都必须学习的话，前述的那个 TPerson 类的设计就完全符合要求，并且是优良的；如果还是引进那个 Tstudent，则反而不好，因为为系统引进不必要的类也是糟糕设计的典范。

因此，适用于所有情况下的“完美设计”是不存在的，只存在面向特定情况下的“最佳设计”。

## 2.4 多态的本质

读小学时，每周都进行大扫除。老师为每个同学都分配了不同的任务，有的扫地，有的擦桌，我的任务是擦窗，每个人拥有自己的职责。然后老师的一声“开始劳动！”就启动了每个人的不同的工作。甚至之后每个星期的大扫除，老师都只需要下一个“开始劳动”的指令，就可以驱动所有人完成不同的任务。

一个抽象的指令，可以让每个个体分别完成具有同一性质但不同内容的动作，多神奇啊！

这就是多态——面向对象编程的核心概念。为了能让读者先对多态抱有足够的重视和尊重，请相信：无论怎样强调多态在 OOP 中的重要性，都不为过。不理解它，也就不会真正明白什么是 OOP！

### 2.4.1 多态的概念与接口重用

首先，什么是多态 (Polymorphisn)？按字面的意思来讲，就是“多种形状”。笔者也没有找到对多态的非常学术性的描述，暂且引用一下 Charlie Calvert 对多态的描述——多态性是允许用户将父对象设置成为与一个或更多的它的子对象相等的技术，赋值之后，基类对象就可以根据当前赋值给它的派生类对象的特性以不同的方式运作。



更简单地说就是：多态性允许用户将派生类类型的指针赋值给基类类型的指针。多态性在 Object Pascal 中是通过虚方法（Virtual Method）实现的。

什么是“虚方法”？虚方法就是允许被其派生类重新定义的方法。派生类重新定义基类虚方法的做法，称为“覆盖”（override）。

这里有一个初学者经常混淆的概念：覆盖（override）和重载（overload）。如前所述，覆盖是指派生类重新定义基类的虚方法的方法。而重载，是指允许存在多个同名函数，这些函数的参数表不同（或许是参数个数不同，或许是参数类型不同，或许两者都不同）。重载的概念并不属于“面向对象编程”。重载的可能的实现是：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说）。例如，有两个重载的同名函数

```
function func(p : integer) : integer; overload;  
function func(p : string) : integer; overload;
```

那么编译器做过修饰后的函数名可能是：int\_func、str\_func。如果调用

```
func(2);  
func('hello');
```

那么编译器会把这两行代码分别转换成：

```
int_func(2);  
str_func('hello');
```

这两个函数的调用入口地址在编译期间就已经静态（记住：是静态！）确定了。这样的确定函数调用入口地址的方法称为早绑定。

而覆盖则是：当派生类重定义了基类的虚方法后，由于重定义的派生类的方法地址无法给出，其调用地址在编译期间便无法确定，故基类指针必须根据赋给它的不同的派生类指针，在运行期动态地（记住：是动态！）调用属于派生类的虚方法。这样的确定函数调用地址的方法称为晚绑定。引用一句 Bruce Eckel 的话：“不要犯傻，如果它不是晚绑定，它就不是多态”。

 **注意：**重载只是一种语言特性，与多态无关，与面向对象也无关！

多态是通过虚方法实现的，而虚方法是通过晚绑定（或动态绑定）实现的。

其次，多态的作用是什么呢？前两节已经讲到，封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块，它们的的目的都是为了代码重用。而多态则是为了实现另一个目的——接口重用。

什么是接口重用？举一个简单的例子，假设有一个描述飞机的基类：

```
type
  TPlane = class
  protected
    FModal : String; // 型号
  public
    procedure fly(); virtual; abstract; // 起飞抽象方法
    procedure land(); virtual; abstract; // 着陆抽象方法
    function modal() : string; virtual; // 查寻型号虚方法
    ..... // 其他可能的操作
  end;
```

然后，从 TPlane 派生出两个派生类，直升机 (TCopter) 和喷气式飞机 (TJet)：

```
TCopter = class(TPlane)
  public
    constructor Create();
    destructor Destroy(); override;
    procedure fly(); override;
    procedure land(); override;
    function modal() : string; override;
    ..... //其他可能的操作
end;

TJet = class(TPlane)
  public
    constructor Create();
    destructor Destroy(); override;
    procedure fly(); override;
    procedure land(); override;
    ..... //其他可能的操作，没有覆盖 modal()
end;
```

TPlane 类的声明中，fly 和 land 方法都是被声明为 virtual 和 abstract 的，这是向编译器指出这些方法是抽象（纯虚）的，也就是在 TPlane 类中不提供这些方法的实现，而派生类则必须实现它，即规定了一套接口。凡是含有 abstract 方法的类被称为“抽象类”，永远无法创建抽象类的实例对象。抽象类是被用来作为接口的。

现在，假设要完成一个机场管理系统，在有了以上的 TPlane 之后，再编写一个全局的函数 g\_FlyPlane()，就可以让所有传递给它的飞机起飞：



```
procedure g_FlyPlane(const Plane : TPlane);
begin
    Plane.fly();
end;
```

是的，仅仅如此就可以让所有传给它的飞机（TPlane 的派生类对象）正常起飞！不管是直升机还是喷气式飞机，甚至是现在还不存在的、以后会增加的飞碟。这是因为，每个派生类（真正的飞机）都可以通过“override”来定义适合自己的起飞方式。

可以看到，g\_FlyPlane()函数接受的参数是 TPlane 类对象的引用，而实际传递给它的都是 TPlane 的派生类对象。现在回想一下本节开头所描述的“多态”：多态性是允许将父对象设置成为与一个或更多的它的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。很显然，

```
parent := child;
```

就是多态的实质！这里的飞机类（TPlane）作为一种接口，而该接口就是被重用的目标。

多态的本质就是“将派生类类型的指针赋值给基类类型的指针”（在 Object Pascal 中是引用），只要这样的赋值发生了，就是在应用多态了，因为在此实行了“向上映射”（“上下”是指类继承层次关系）。

应用多态的例子非常普遍。在 Delphi 的 VCL 类库中，最典型的就是：TObject 类有一个虚拟的 Destroy 析构函数和一个非虚拟的 Free 方法。Free 方法中首先判断对象本身是否为 nil，保证不为 nil 时便调用 Destroy。对任何对象（都是 TObject 的派生类对象）调用其 Free()方法，但执行的都是 TObject.Free();(因为 TObject.Free()为非虚拟方法，无法被覆盖)，然后由它调用被每个类重定义了析构函数 Destroy();(因为 Destroy()为虚方法，派生类可以覆盖)，这就保证了任何类型的对象都可以正确、安全地被析构。

因此，在定义自己的类时，如果有析构函数存在，就必须在它的声明之后加上 override 关键字。否则会发生什么呢？

还拿刚才的飞机类作为例子，有一个飞机销毁站，这个销毁站有一个函数：

```
procedure DestroyPlane(var Plane : TPlane)
begin
    Plane.Free();
    Plane := nil;
end;
```

将正常的飞机（析构函数带有 override 的飞机）传给它，编译器都会正常地调用飞机的析构函数用以将飞机拆解开，将资源正常回收。

如果建造的飞机的析构函数没有被指明 override 关键字，那么，将飞机传递给这个 DestroyPlane()的函数时，编译器调用的绝对不是用户所给飞机定义的析构函数，而会是 TPlane.Destroy()。能指望 TPlane 的析构函数会好好拆解飞机吗？祈祷吧，别把油箱弄爆

炸了。

也就是说，在被执行了

```
parent := child;
```

之后，无论调用

```
parent.Free();
```

还是调用

```
child.Free();
```

都应该产生同样的结果，从语义上来说，这两行代码必须做相同的事情。当然，这样的情况不仅仅只对于析构函数而言，任何想要使通过基类对象指针做到的事情与通过派生类对象指针所做的相同，就要在基类中将这个方法声明为 **virtual**，在派生类中将该方法声明为 **override**。

 **注意：** 给自己的析构函数加上 **override** 声明！

## 2.4.2 多态的实现与 VMT/DMT

多态的本质是“将派生类类型的指针赋值给基类类型的指针”。那么，为什么这种赋值是允许的，或者说是安全的呢？

从语义上来讲，继承所表现的是“是一种”的关系，也就是说，每个派生类对象必定“是一种”基类对象。所以，任何向基类类型的请求，派生类对象都可以无条件地正常处理。因为直升机“是一种”飞机，喷气式飞机也“是一种”飞机，所以所有对飞机的操作请求，它们都应该可以正常处理。

从语言上来讲，由于派生类通常比基类拥有更多的数据成员而绝对不会更少，派生类对象所占的内存空间必定大于或等于基类对象所占的内存空间。因此，将基类类型的指针指向派生类类型的对象时，在指针的可视范围中的内存必定是可用的，这一部分内存空间必定是属于对象的，所以这种赋值行为是合法的、安全的，并且得到编译器认可的。

例如，如下的两个类：

```
TBase = class
private
    FMember1 : Integer;
    FMember2 : Integer;
end;

TDerived = class(TBase)
private
    FMember3 : Integer;
```



```
end;
```

当 TBase 类型的指针指向其派生类类型 TDerived 的对象后:

```
var
  Parent : TBase;
  Child  : TDerived;
Begin
  Child := TDerived.Create();
  Parent := Child; //当执行这行代码之后……

  // 以上两行代码也可以简化为 Parent := TDerived.Create();

  …… //之后的代码省略
End;
```

TBase 类型的指针 (Parent 指针) 指向了 Child 对象实体所在的内存首地址。在 2.3 节中说过, 每个派生类对象实体中都包含了一个完整的基类对象实体。此处的 Parent 指针可以访问的范围, 正是这个完整基类 (TBase) 对象实体的大小。因此, Parent 指针始终可以合法地访问其所指向的内存空间。

Parent 指针的可视范围如图 2.5 所示。

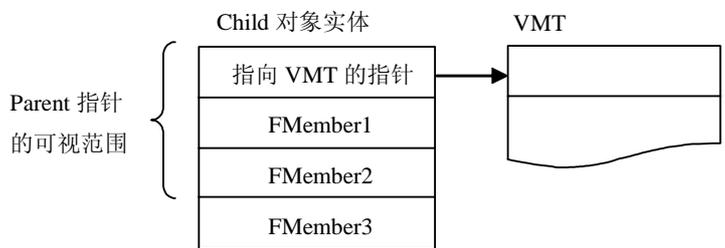


图 2.5 基类指针可视范围演示

在图 2.5 中又一次看到了 VMT, VMT 究竟是何方神圣呢? 为什么每个对象都会有一个指向 VMT 的指针呢? 这些问题可以在了解虚方法的动态绑定实现机制中找到答案。搞清这些, 便会清楚多态是如何实现的了。

当创建一个类的实例之后, 编译器会在该对象的内存空间的首 4 个字节安插一个指针, 该指针所指向的地址称为 VMT (Virtual Method Table, 虚方法表), 这个表中存放了该类的所有虚方法的入口地址。在 Object Pascal 中, 所有类实例都会有这么一个指向 VMT 的指针。如果没有在类中声明虚方法, 则该指针为 nil。

还是以前面所说的飞机抽象类和直升机类为例:

```
TPlane = class
```

```

protected
    FModal : String;
public
    procedure fly(); virtual; abstract;    // 起飞抽象方法
    procedure land(); virtual; abstract;  // 着陆抽象方法
    function modal() : string; virtual;   // 查寻型号虚方法
    ..... // 其他可能的操作
end;

TCopter = class(TPlane)
public
    constructor Create();
    destructor Destroy(); override;
    procedure fly(); override;
    procedure land(); override;
    ..... // 其他可能的操作, 没有覆盖 TPlane.modal()
end;

```

在一个全局函数中用飞机类型来创建直升机实例:

```

procedure g_CreateACopter(var Plane : TPlane);
begin
    Plane := TCopter.Create;
end;

```

当执行 `Plane := TCopter.Create` 之后, 一个直升机实例就被创建了, 并且 `Plane` 指针指向了它, 如图 2.6 所示。

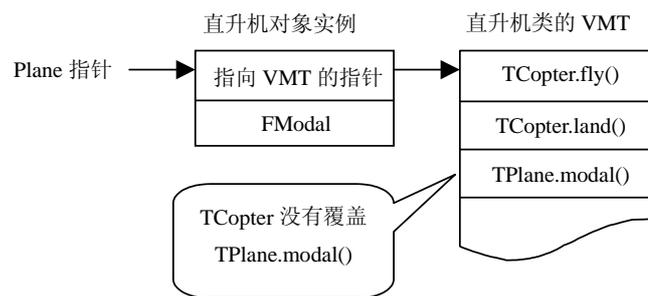


图 2.6 plane 指针指向直升机对象实例

没有被派生类覆盖的方法, 编译器会将基类的该方法的实现的入口地址填入派生类的 VMT 中。如图 2.6 所示, 直升机类(TCopter)覆盖了其基类(TPlane)的虚方法 fly()和 land(), 因此在 TCopter 的 VMT 中, fly 和 land 被确定为 TCopter 的实现方法的入口地址。但由于



TCopter没有覆盖 TPlane 的虚方法 modal(),则在 VMT 的 modal()项中被填入了 TPlane.modal() 的入口地址,即基类中该方法的入口地址。

被派生类覆盖的方法,则会将派生类实现的方法的入口地址填入 VMT 中以取代基类被覆盖的方法。

这就是“晚绑定”或“动态绑定”!

对照图 2.6 来看,当基类类型的指针指向了直升机实例对象后,可以通过基类类型的指针来让这架直升机起飞:

```
plane.fly();
```

编译器通过 plane 所指对象的“指向 VMT 的指针”可以定位到 TCopter.fly()的地址,由此便可以找到属于直升机的 fly()方法,而得以以直升机的起飞方式来让一架直升机起飞。

虽然动态绑定看起来并不太复杂,但意义重大!如果没有动态绑定,那么,试想一下,以喷气式飞机的起飞方式来让一架直升机起飞会发生什么?这可关系到飞行员的安全啊。

到这里,读者对动态绑定的实现应该清楚了。细心的读者可能又会发现一个问题, TPlane 中的析构函数也是 virtual 的,为什么没有出现在 VMT 中呢?

当然,并非 VMT 中没有析构函数,只能说它没有出现在我们所能见到的 VMT 中。之前所说的“指向 VMT 的指针”所指向的 VMT,其实只是真正的 VMT 的一部分,也就是用户定义的第一个虚方法的位置。如果以这个位置作为原点,向正方向即刚才所说的 VMT。而向负方向,则是语言定义的另一类信息所在的地址,析构函数地址就被放在了负方向上了。

Delphi 之所以这么做,是为了使得 Object Pascal 的 VMT 与 C++的以及 COM 的 vtable (虚函数表)兼容。

这里给出完整的 VMT 的地址表(来自 Delphi6 Help),不过这些内容仅能作为参考,因为 Borland 并不承诺将来不会改变这个格式,如表 2.1 所列。

表 2.1 Delphi6 VMT

偏移地址	类 型	描 述
-76	Pointer	pointer to virtual method table (or nil)
-72	Pointer	pointer to interface table (or nil)
-68	Pointer	pointer to Automation information table (or nil)
-64	Pointer	pointer to instance initialization table (or nil)
-60	Pointer	pointer to type information table (or nil)
-56	Pointer	pointer to field definition table (or nil)
-52	Pointer	pointer to method definition table (or nil)
-48	Pointer	pointer to dynamic method table (or nil) (指向 DMT 的指针)
-44	Pointer	pointer to short string containing class name
-40	Cardinal	instance size in bytes
-36	Pointer	pointer to a pointer to ancestor class (or nil)

续表

偏移地址	类 型	描 述
-32	Pointer	pointer to entry point of SafecallException method (or nil)
-28	Pointer	entry point of AfterConstruction method
-24	Pointer	entry point of BeforeDestruction method
-20	Pointer	entry point of Dispatch method
-16	Pointer	entry point of DefaultHandler method
-12	Pointer	entry point of NewInstance method
-8	Pointer	entry point of FreeInstance method
-4	Pointer	entry point of Destroy destructor
0	Pointer	entry point of first user-defined virtual method
4	Pointer	entry point of second user-defined virtual method

可以看到，偏移地址-28 ~ -4 所存放的都是 TObject 的虚方法地址，当然析构函数也在其中。

最后，有必要再谈一下 Object Pascal 所独有的 DMT（动态方法表）。

在表 2.1 的偏移地址为-48 处是一个指向 DMT 的指针，它是干什么用的？它和 VMT 有什么关系？

在 VMT 中可以看到，派生类的虚方法表完全继承了基类的虚方法表，只是将被覆盖了的虚方法的地址改变了。基类和每个派生类都有一份自己的虚方法表。可以想象，随着类层次的扩展，虚方法表将耗费非常大的内存空间。为了防止这种情况，Object Pascal 引入了“dynamic”的概念。对于程序员来说，dynamic 方法和 virtual 方法实现相同的功能，只是声明的关键字不同：

```
Procedure fly(); dynamic; // 是 dynamic 而不是 virtual
```

被声明为 dynamic 的方法，其入口地址将被放在 DMT 中。DMT 和 VMT 的区别在于：对于派生类没有覆盖的方法，这些方法的入口地址不会出现在 DMT 中，编译器要通过基类的信息来寻找它们的入口地址。

如果将 TPlane 的抽象方法 land 和虚方法 modal 改成 dynamic，即：

```
TPlane = class
protected
    FModal : String;
public
    procedure fly(); virtual; abstract; // 仍然保持 virtual
    procedure land(); dynamic; abstract; // 将 virtual 改成 dynamic
    function modal() : string; dynamic; // 将 virtual 改成 dynamic
    ..... // 其他可能的操作
end;
```



```

TCopter = class(TPlane)
public
    constructor Create();
    destructor Destroy(); override;
    procedure fly(); override;
    procedure land(); override;
    function modal() : string; // 不覆盖 Plane.modal();
    ..... // 其他可能的操作
end;

```

则 TCopter 的 VMT/DMT 会变成如图 2.7 所示的样子。

对比图 2.6 和图 2.7 可知，由于 DMT 中不会出现没有被派生类覆盖的基类 dynamic 方法，因此 DMT 会比 VMT 节省空间（大多数情况下）。当基类有许多虚方法，而派生类只覆盖很少几个时，区别尤其明显。当派生层次越来越深，派生类数量越来越多，DMT 就能节省更多的内存空间。但是 DMT 中对基类的动态方法的寻址不是直接进行的，因此 dynamic 方法的寻址比 virtual 方法要慢许多。

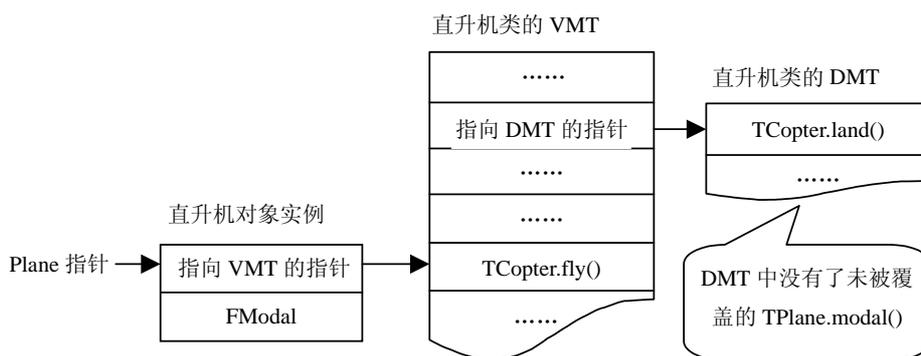


图 2.7 直升机类的 VMT/DMT

virtual 和 dynamic 的区别仅在于编译器采用不同的晚绑定策略而已，对于程序员来说，它们的功能相同。

如何取舍就看实际的需求了，一般情况下，几乎每个派生类都要覆盖的方法，将它声明为 virtual；如果类层次很深，或派生类很多，但某个方法只被很少的派生类覆盖，则将它声明为 dynamic。

另外需要注意的是，只有 VMT 才与 C++、COM 的 vtable 兼容，因此当需要这样的兼容性时，只能使用 virtual。

## 2.5 小 结

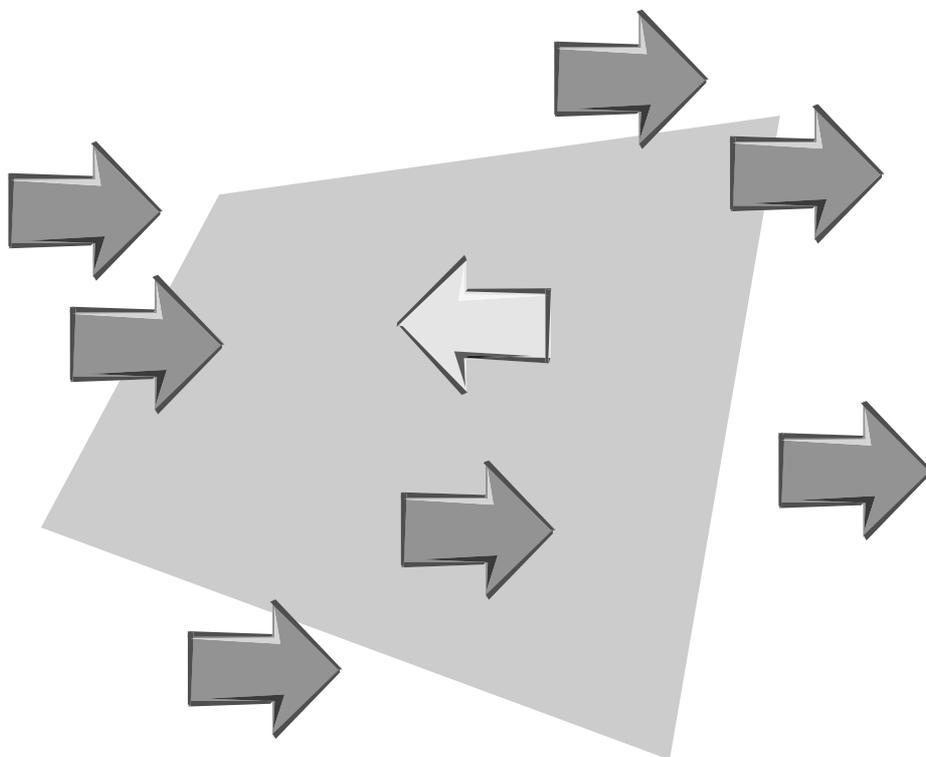
传统的说法是，封装、继承、多态是面向对象编程的三个基本特性。实际上，封装只

是抽象数据类型（ADT），有了继承才能被称为面向对象。而继承的存在，除了扩展现存类的功能外，另一个更重要的作用就是作为多态存在的基石。

多态是一种能够带来灵活性的东西，它使得通过接口重用来实现代码重用。可以毫不夸张地说，不领会多态，不明白晚绑定，就不可能明白什么是面向对象！因为只有在使用 `virtual` 后，才是真正在用面向对象的典范（paradigm）思考……

## 第 3 章 异常及错误处理

健壮的程序来自于正确的错误处理。



相信我，总会有意外的……



正如同现实生活中我们不可能事事如意，你所写的代码也不可能每一行都能得到正确的执行。生活中遇到不如意的事情，处理好了，雨过天晴；处理不好，情况会越变越糟，甚至一发而不可收拾，后果难料。程序设计中同样如此，所谓健壮的程序，并非不出错的程序，而是在出错的情况下能很好地处理的程序。

因此，错误处理一直是程序设计领域的一个重要课题。而异常就是面向对象编程提供的错误处理解决方案。它是一个非常好的工具，如果你选择了 OOP，选择了 Delphi，那么异常也就成为你的惟一选择了。

要让你信服地选择异常，需要给出一些理由。在本章中会让你清楚明白地了解异常所带来的好处。

### 3.1 异常的本质

什么是异常？为什么要用它？

在基于函数的结构中，一般使用函数返回值来标明函数是否成功执行，并给出错误类型等信息。于是就会产生如下形式的代码：

```
nRetVal := SomeFunctionToOpenFile();

if nRetVal = E_SUCCEEDED then // 成功打开
begin
    .....
end
else if nRetVal = E_FILE_NOT_FOUND then // 没有找到文件
begin
    .....
end
else if nRetVal = E_FILE_FORMAT_ERR then // 文件格式错
begin
    .....
end
else then
begin
    .....
end
```

使用返回错误代码的方法是非常普遍的，但是使用这样的方法存在两个问题：

(1) 造成冗长、繁杂的分支结构（大量的 if 或 case 语句），使得程序流程控制变得复杂，同时造成测试工作的复杂，因为测试需要走遍每个分支。

(2) 可能会存在没有被处理的错误（函数调用者如果不判断返回值的话）。

异常可以很好地解决以上两个问题。

所谓“异常”是指一个异常类的对象。Delphi 的 VCL 中，所有异常类都派生于 `Exception` 类。该类声明了异常的一般行为、性质。最重要的是，它有一个 `Message` 属性可以报告异常发生的原因。

抛出一个异常即标志一个错误的发生。使用 `raise` 保留字来抛出一个异常对象，如：

```
raise Exception.Create('An error occurred!');
```

但需要强调的是，异常用来标志错误发生，却并不因为错误发生而产生异常。产生异常仅仅是因为遇到了 `raise`，在任何时候，即使没有错误发生，`raise` 都将会导致异常的发生。

 **注意：**异常的发生，仅仅是因为 `raise`，而非其他！

一旦抛出异常，函数的代码就从异常抛出处立刻返回，从而保护其下面的敏感代码不会得到执行。对于抛出异常的函数本身来说，通过异常从函数返回和正常从函数返回（执行到函数末尾或遇到了 `Exit`）是没有什么区别的，函数代码同样会从堆栈弹出，局部简单对象（数组、记录等）会自动被清理、回收。

采用抛出异常以处理意外情况，则可以保证程序主流程中的所有代码可用，而不必加入繁杂的判断语句。

例如，函数 A 抛出异常：

```
function A() : Integer;
var
  pFile : textfile;
begin
  ..... // 一些代码
  pFile := SomeFunctionToOpenAnFile();
  if pFile = nil then
    raise Exception.Create('Open file failed!'); // 文件打开失败抛出异常
  Read(pFile, .....); // 读文件
  ..... // 其他一些对文件的操作，此时可以保证文件指针有效
end;
```

函数 A 的代码使得对文件打开的出错处理非常简单。如果打开文件失败，则抛出一个 `Exception` 类的异常对象，函数立刻返回，从而保护了以下对文件指针的操作不被执行。而之后的代码可以假设文件指针肯定有效，从而令代码更加美观。

生活中，我们每天扔掉的垃圾都会有清洁工人收拾、处理，否则生活环境中岂不到处充斥着垃圾？同样，抛出的异常也需要被捕获和处理。假设函数 B 调用了函数 A，要捕获这个文件打开失败的异常，就需要在调用 A 之前先预设一个陷阱，这个陷阱就是所谓的“try...except 块”。



先看一下函数 B 的代码：

```
procedure B();  
begin  
    ..... // 一些代码  
    try  
        A(); // 调用 A  
        SomeFunctionDependOnA(); // 依赖于 A 的结果的函数  
    except  
        ShowMessage('some error occured'); // 嘿嘿，掉进来了，发生异常  
    end;  
    ..... // 继续的代码  
end;
```

A 抛出的异常，会被 B 所设的 try...except 所捕获。一旦捕获到异常，就不再执行之后的敏感代码，而是立刻跳至 except 块执行错误处理，处理完成后再继续执行整个 try 块之后的代码。程序流程的控制权被留在了函数 B。

如果不喜欢自己收拾垃圾，因而在 B 中并没有预设 try...except 块的话，则异常会被继续抛给 B 的调用者，而如果 B 的调用者同样不负责任，则异常会被继续像踢足球一样被踢给更上层的调用者，依此类推。不过，不用担心，我们有一个大管家，大家都不要的烫手山芋，它会帮我们收拾，那就是——VCL（Delphi 的应用程序框架）。

因为 VCL 的框架使得所编写的整个应用程序被包在一个大的 try...except 中，无论什么没有被处理的异常，最终都会被它所捕获，并将程序流程返回到最外层的消息循环中，决无遗漏！这也就是为什么会看到很多用 Delphi 所编写的但并不专业的小软件有时会跳出一个报告错误的对话框（如图 3.1 所示）。发生这样的情况应该责怪软件的编写者没有很好地处理错误，但有些不明白异常机制的程序员常常会责怪 Delphi 编写的程序怎能会有这样的情况发生。其实出现这个提示，应该感谢 VCL 的异常机制让程序可以继续运行而不是“非法终止”。



图 3.1 异常被 VCL 所捕获

 **注意：** VCL 用一个大的 try...except 将代码包裹起来！

因此，在 VCL 框架中不会有不被处理的异常，换句话说，也就是不会有不被处理的错误（虽然笔者说过异常并不等于错误）。对异常的捕获也非常简单，不见了一大堆的 if 或

case，程序控制流程的走向也就十分清晰明了了，这是给测试人员带来的好消息。

## 3.2 创建自己的异常类

异常机制是完全融入面向对象的体系的，所以异常类和一般类一样具有继承和多态的性质。其实，异常类和普通类并没有什么区别。

Object Pascal 的运行时异常基类是 Exception，VCL 中所有异常类都应该从它派生。当然，Object Pascal 语言并不规定如此，可以用 raise 抛出任何除简单类型之外的类类型的对象，try...except 同样可以捕获它，在异常处理后同样会自动析构、回收它，只是 Exception 定义了异常的大多数特征。既然别人已经为我们准备了一个好用的、完备的 Exception，当然没有理由不用它。

也许读者也已经注意到，所有 VCL 的异常发生时，弹出的警告对话框都带有一段有价值的对于异常的发生原因的描述（正如图 3.1 中的“is not a valid integer value”）。这段描述对于 debug 工作是非常有用的。它正是来自于 Exception 类的 Message 属性，所有异常类被创建时都必须给出一个出错描述。因此，在定义、使用自己的异常类时，也要给出一个不会令人迷惑的、明白说出错误原因的 Message 属性。

 **注意：**从 Exception 派生自己的异常类！

下面以一个示例程序来演示如何定义、使用自己的异常类，其代码及可执行文件可在配书光盘的 exception 目录下找到。

程序运行后的界面如图 3.2 所示。

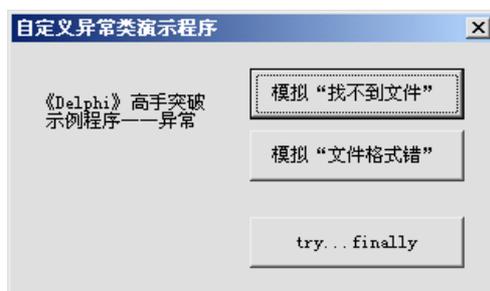


图 3.2 自定义异常类演示程序界面

该程序的运行界面充分地体现了第 1 章所说的“简单性”原则。界面上只有 3 个按钮，先看上面两个（另一个“try...finally”按钮先不说明，留待 3.3 节讲解）。一个模拟打开文件时发生“找不到文件”的错误，一个模拟发生“文件格式错”的错误。所谓模拟发生错误，就是在并没有真正发生错误的情况下抛出异常，使得编译器认为发生了错误，即单击这两个按钮后，程序会分别抛出相应的异常。

首先要定义两种错误所对应的异常类。它们的定义和实现在 ExceptionClass.pas 单元中。该单元代码清单如下：



```
unit ExceptionClass;

interface

uses SysUtils, Dialogs;

Type

  EFileOpenFailed = class(Exception) // 定义一个文件打开失败的通用异常类
  public
    procedure Warning(); virtual; abstract;
  end;

  EFileNotFound = class(EFileOpenFailed) // 细化文件打开失败的异常
  public
    procedure Warning(); override;
  end;

  EFileFormatErr = class(EFileOpenFailed) // 细化文件打开失败的异常
  public
    procedure Warning(); override;
  end;

implementation

{ EFileNotFound }

procedure EFileNotFound.Warning;
begin
  ShowMessage('真是不可思议，竟然找不到文件！');
end;

{ EFileFormatErr }

procedure EFileFormatErr.Warning;
begin
  ShowMessage('更不可思议的是，文件格式不对！');
end;

end.
```

我们先定义了一个标志打开文件失败的异常基类 EFileOpenFailed，并给它声明了一个

抽象方法 `Warning`。然后又细化了错误的原因，从而派生出两个异常类——`EFileNotFound`、`EFileFormatErr`，它们都具体实现了 `Warning` 方法。

在应用程序的主 `Form(Form1)` 中，定义一个模拟发生错误并抛出异常的 `SimulateError()` 方法来模拟发生错误、抛出异常。

然后定义一个 `ToDo()` 方法来调用会引发异常的 `SimulateError()`，并且用 `Try` 将其捕获进行异常处理。

最后在两个按钮的 `OnClick()` 事件中，调用 `ToDo()` 方法。

其代码清单如下：

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Label1: TLabel;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure SimulateError(Button : TObject);
    procedure ToDo(Button : TObject);
  end;

var
  Form1: TForm1;

implementation

uses ExceptionClass;
```



```
{ $R *.dfm }

procedure TForm1.SimulateError(Button : TObject);
begin
    if Button = Button1 then
        raise EFileNotFound.Create('File Not Found')
    else if Button = Button2 then
        raise EFileFormatErr.Create('File Format Error')
    else // Button = Button3
        raise Exception.Create('Unknoww Error');
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    ToDo(Sender);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    ToDo(Sender);
end;

procedure TForm1.ToDo(Button : TObject);
begin
    try
        SimulateError(Button)
    except
        on E : EFileOpenFailed do
            E.Warning();
        on E : Exception do
            ShowMessage(E.Message);
        end;
    end;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
    AStream : TMemoryStream;
begin
    AStream := TMemoryStream.Create();

    try
        SimulateError(Sender);
    end;
end;
```

```
finally
    AStream.Free();
end;
end;
end.
```

程序运行后，当单击界面上方的两个按钮之一时，都会调用 `ToDo` 方法。而在 `ToDo` 方法中，由于 `SimulateError` 被调用而引发一个异常，虽然并没有真的发生打开文件错误，但确实抛出了异常。这再次说明了，异常只是用来标志错误，而并不等同于错误。

程序中，我们定义了一个标志打开文件失败的异常基类 `EFileOpenFailed`，以及两个派生的异常类——`EFileNotFound`、`EfileFormatErr`。这样定义异常类框架，给错误处理部分带来了更多的灵活性。这是多态性给我们的又一个恩惠。可以自由选择需要捕获的异常的“精度”。也就是说，如果用户非常关心发生错误的具体原因，则可以捕获每个最底层的异常类；而如果只关心是否发生了打开文件的错误，那么可以只捕获 `EFileOpenFailed` 类；若关心的只是是否有错误发生，则只需捕获 `Exception` 就行了。

在 `SimulateError` 的调用之外，设置了 `try...except`，那么它所引发的异常都会被捕获。将“精度”更“细”的异常类的处理代码放在前面，而把“精度”较“粗”的异常类的处理代码放在后面。如果相反，则所有异常都会被 `Exception` 的处理代码捕获，而其他的异常类的处理代码则永远都没有机会执行了。

`Exception` 程序演示了一个很小的、自定义的异常类框架的定义、实现及使用。“麻雀虽小，五脏俱全”，它给出了一种在自己程序中错误的捕获、处理的思路。

### 3.3 try...finally

现在已经知道，在函数中引发异常将导致函数的正常返回，因此函数栈中的局部简单对象（数组、记录等）会得到释放。同时也知道了，在 `Object Pascal` 中所有的类对象都在堆中被构造，编译器不会在退出函数时自动调用它们的析构函数，那么如何保证所有的局部类对象也能被释放呢？

`Object Pascal` 引入了独特的 `try...finally` 来解决这个问题。

`try...finally` 块帮你保证一些重要的代码在无论是否发生异常的情况下都能被执行，这些代码位于 `finally` 和 `end` 之间。

再次打开 `Exception` 程序，现在来看一下没用过的第 3 个按钮。为它的 `Click` 事件添加如下的代码：

```
procedure TForm1.Button3Click(Sender: TObject);
var
    AStream : TMemoryStream;
```



```
begin
  AStream := TMemoryStream.Create();

  try
    SimulateError(Self);
  finally
    AStream.Free();
  end;
end;
```

它首先创建了一个内存流对象，以模拟该函数申请了一些系统资源。然后还是调用了 `SimulateError` 方法，不过这次 `SimulateError` 抛出的是一个 `Exception` 异常。但在此把内存流对象的销毁工作放在了 `finally` 保护之中，由此保证该对象的释放。可以自己单步跟踪试一下，无论在发生异常（即调用了 `SimulateError`）的情况下，还是正常退出（不调用 `SimulateError` 或将 `SimulateError` 的调用改为 `Exit`）的情况下，`AStream.Free()` 都会得到执行。

同时拥有 `try...except` 和 `try...finally`，应该说是 Delphi 程序员的一种幸运，值得庆幸。只是，我们想得到的会更多，会希望拥有

```
try
  .....
except
  .....
finally
```

这样的结构，只是目前还得不到满足。虽然可以用

```
try
  try
    .....
  except
    .....
  end
finally
  .....
end;
```

来取代，但显然不如所希望的那样结构美观和优雅。这不能不说是一种遗憾，让我们寄希望于下一个 Delphi 版本吧！

## 3.4 构造函数与异常

这个话题在 C++ 社区中经常会被提起，而在 Delphi 社区中似乎从来没有人注意过，也许由于语言的特性而使得 Delphi 程序员不必关心这个问题。但我想，Delphi 程序员也应该对该问题有所了解，知道语言为我们提供了什么而使得我们如此轻松，不必理会它。正所谓“身在福中须知福”。

我们知道，类的构造函数是没有返回值的，因此如果构造函数构造对象失败，则不可能依靠返回错误代码来解决。那么，在程序中如何标识构造函数的失败呢？最“标准”的方法就是：抛出一个异常。

构造函数失败，意味着对象的构造失败。那么抛出异常之后，这个“半死不活”的对象会被如何处理呢？

在此，读者有必要先对 C++ 对这种情况的处理方式有一个了解。

在 C++ 中，构造函数抛出异常后，析构函数不会被调用。这种做法是合理的，因为此时对象并没有被完整构造。

如果构造函数已经做了一些诸如分配内存、打开文件等操作，那么 C++ 类需要有自己的成员来记住做过哪些动作。当然，这样做对于类的实现者来说非常麻烦。因此，一般 C++ 类的实现者都避免在构造函数中抛出异常（可以提供诸如 `Init` 和 `UnInit` 的成员函数，由构造函数或类的客户去调用它们，以处理初始化失败的情况）。而每一本 C++ 的经典著作所提供的方案都是使用智能指针（STL 的标准类 `auto_ptr`）。

在 Object Pascal 中，这个问题变得非常简单，程序员不必为此大费周折。如果 Object Pascal 的类在构造函数中抛出异常，则编译器会自动调用类的析构函数（由于析构函数不允许被重载，可以保证只有唯一一个析构函数，因此编译器不会迷惑于多个析构函数之中）。析构函数中一般会析构成员对象，而 `Free()` 方法保证了不会对 `nil` 对象（即尚未被创建的成员对象）调用析构函数，因此在使得代码简洁优美的前提下，又保证了安全。

以下的程序演示了构造函数中抛出异常后，Object Pascal 编译器所作的处理方法。

首先定义 `TMyClass`：

```
type
  TMyClass = class
  private
    FStr : PChar; // 字符串指针
  public
    constructor Create();
    destructor Destroy(); override;
  end;
```

然后实现 `TMyClass`，并让它的构造函数中抛出异常：



```
constructor TMyClass.Create();
begin
    FStr := StrAlloc(10); // 构造函数中为字符串指针分配内存
    StrCopy(FStr, 'ABCDEFGHI');
    raise Exception.Create('error'); // 抛出异常，没有理由
end;

destructor TMyClass.Destroy();
begin
    StrDispose(FStr); // 析构函数中释放内存
    WriteLn('Free Resource');
end;
```

最后，编写程序主流程的代码。主流程中首先创建 **TMyClass** 类的实例：

```
var
    Obj : TMyClass;
    i : integer;
begin
    try
        Obj := TMyClass.Create();
        // Obj.Free(); // 不调用析构函数，但发生异常时，编译器自动调用了析构函数
        WriteLn('Succeeded');
    except
        Obj := nil;
        WriteLn('Failed');
    end;

    Read(i); // 暂停屏幕，以便观察运行结果
end.
```

这段代码中，创建 **TMyClass** 类的实例时遇到了麻烦，因为 **TMyClass** 的构造函数抛出了异常，但这段代码执行结果却是：

```
Free Resource
Failed
```

出现了“Free Resource”，说明发生异常后，析构函数被调用了。而这正是在构造函数抛出异常之后，编译器自动调用析构函数的结果。

因此，如果类的说明文档或类的作者告知你，类的构造函数可能会抛出异常，那就要记得用 **try...except** 包住它！

C++与 Object Pascal 对于构造函数抛出异常后的不同处理方式，其实正是两种语言的设计思想的体现。C++秉承 C 语言的风格，注重效率，一切交给程序员来掌握，编译器不做多余动作；Object Pascal 继承 Pascal 的风格，注重程序的美学意义，编译器帮助程序员完成复杂的工作。

### 3.5 小 结

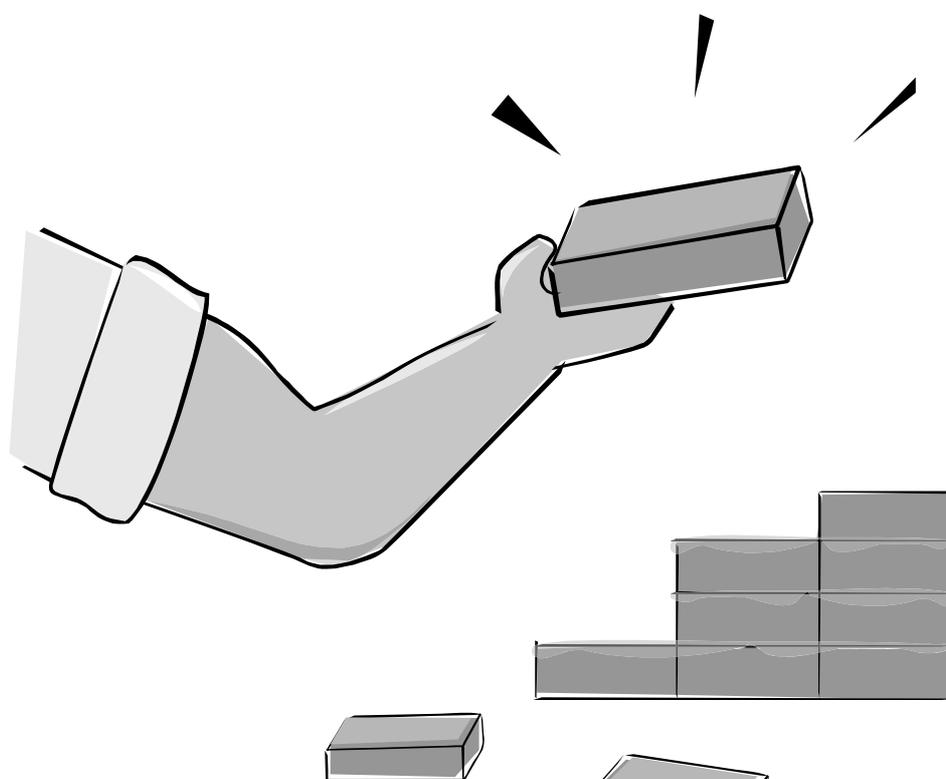
3

异常是面向对象编程带来的非常好的工具，不加以利用是很可惜的。但是，正如万事都有个“度”，滥用异常也是不可取的。使用异常不是没有代价，它会增加程序的负担，编写若干 try...except 和编写数以千计的 try...except 之间是有很大区别的。

同时，也不必过分害怕由它所带来的负担。其实，既然已经使用了 Delphi，其实就已经在使用异常了，也许只是自己还不知道。听听 Charlie Calverts 的忠告：“在似乎有用的时候，就应该使用 try...except 块。但是要试着让自己对这种技术的热情不要太过分”。

## 第 4 章 VCL 库

Delphi 高手很大程度上就是 VCL 高手。



别人造砖我砌房！



VCL——Visual Component Library，是 Delphi 的基石。Delphi 的优秀，很大程度上得益于 VCL 的优秀。

VCL 是 Delphi 所提供的基本组件库，也就是所谓的 Application Framework，它对 Windows API（应用程序接口）进行了全面封装，为桌面开发（不限于桌面开发）提供了整套的解决方案，使得程序员可以在不知晓 API 的情况下进行 Windows 编程。

不过，作为专业的程序员，不知晓 API 是不可能的。VCL 还是一个 Framework（应用程序框架），可以将 VCL 作为一个平台，程序员在其基础上构建应用程序，便可以忽略很多系统 API 的细节，而使得开发速度更快。

VCL 的组件也不同于 ActiveX 控件，VCL 组件通过源代码级连接到可执行文件中，因此其速度更快。而且，企业版的 Delphi 带有全部 VCL 库的源代码，这样程序员不单单可以知道如何使用 VCL 组件，更可以了解其运行机制与构架。

了解 VCL 的构架，无论对于编写自己的 Application，还是设计程序框架，或者创建自己的组件/类融入 VCL 构架中，都是必需和大有裨益的。

这也符合某种规律：在学习的时候，求甚解；而在应用的时候，则寻找捷径。Delphi 和 VCL 都能满足这两种需求，因为使用它

- ◆ 可以不隐藏任何想知道的细节；
- ◆ 可以忽略不想知道的细节。

在本章中，将带游历 VCL 库的核心，剖析 VCL 的代码。从此，VCL 对您来说不会再是神秘而艰涩的，因为带领读者它们同样是用代码铸造成的。

## 4.1 VCL 概貌

先看一下 VCL 类图的主要分支，如图 4.1 所示。

在图中可以看到，TObject 是 VCL 的祖先类，这也是 Object Pascal 语言所规定的。但实际上，TObject 以及 TObject 声明所在的 system.pas 整个单元，包括在“编译器魔法”话题中提到的 \_ClassCreate 等函数，都是编译器内置支持的。因此，无法修改、删除 system.pas 中的任何东西，也无法将 system.pas 加入你的 project，否则会得到“Identifier redeclared ‘system’”的错误提示，因 project 中已经被编译器自动包含了 system 单元。

意思是，TObject 是 Object Pascal 语言/编译器本身的一个性质！



**注意：**TObject 是属于编译器的特性！

TObject 封装了 Object Pascal 类/对象的最基本行为。

TPersistent 派生自 TObject，TPersistent 使得自身及其派生类对象具有自我保存、持久存在的能力。

TComponent 派生自 TPersistent，这条分支之下所有的类都可以被称为“组件”。组件的一般特性是：

- (1) 可出现在开发环境的“组件板”上。

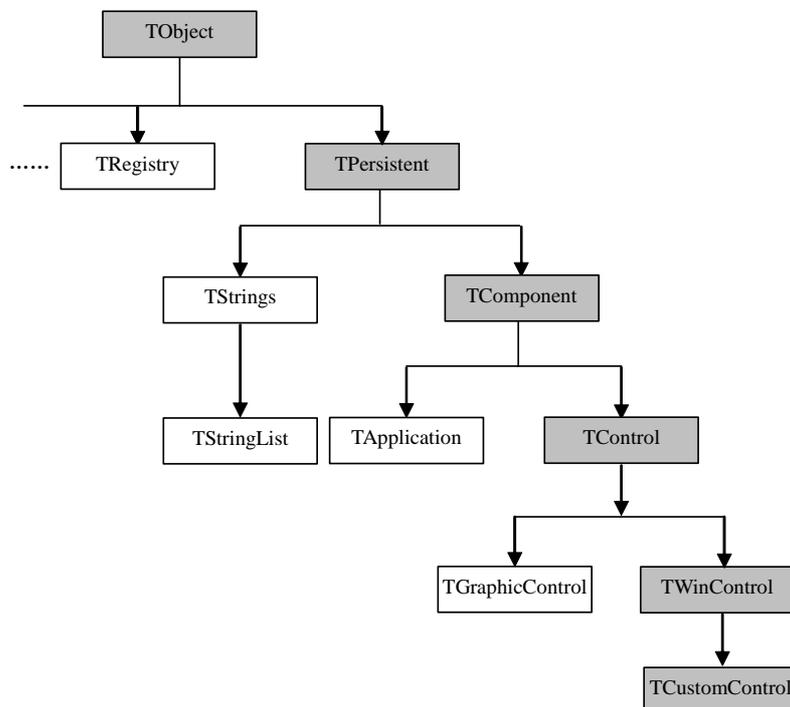


图 4.1 VCL 类图主要分支（深色表示核心分支）

(2) 能够拥有和管理其他组件。

(3) 能够存取自身（这是因为 **TComponent** 派生自 **TPersistent**）。

**TControl** 派生自 **TComponent**，其分支之下所有的类，都是在运行时可见的组件。

**TWinControl** 派生自 **TControl**，这个分支封装了 Windows 系统的屏幕对象，也就是一个真正的 Windows 窗口（拥有窗口句柄）。

**TCustomControl** 派生自 **TwinControl**。从 **TCustomControl** 开始，组件拥有了 Canvas（画布）属性。

从 4.2 节开始，将会先后结合 VCL 中一些核心类的实现代码来了解它们。

## 4.2 TObject 与消息分发

首先来看一下 **TObject** 这个“万物之源”究竟长得何等模样。它的声明如下：

```

TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
  
```



```
class function ClassName: ShortString;
class function ClassNameIs(const Name: string): Boolean;
class function ClassParent: TClass;
class function ClassInfo: Pointer;
class function InstanceSize: Longint;
class function InheritsFrom(AClass: TClass): Boolean;
class function MethodAddress(const Name: ShortString): Pointer;
class function MethodName(Address: Pointer): ShortString;
function FieldAddress(const Name: ShortString): Pointer;
function GetInterface(const IID: TGUID; out Obj): Boolean;
class function GetInterfaceEntry(const IID: TGUID):
PInterfaceEntry;
class function GetInterfaceTable: PInterfaceTable;
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;
end;
```

从 `TObject` 的声明中可以看到，`TObject` 包含了诸如实例初始化、实例析构、RTTI、消息分发等相关实现的方法。现在就来研究一下 `TObject` 与消息分发，这也是 VCL 对 Windows 消息封装的模型基础。

在 `TObject` 类中，有一个 `Dispatch()` 方法和一个 `DefaultHandler()` 方法，它们都是与消息分发机制相关的。

`Dispatch()` 负责将特定的消息分发给合适的消息处理函数。首先它会在对象本身类型的类中寻找该消息的处理函数，如果找到，则调用它；如果没有找到而该类覆盖了 `TObject` 的 `DefaultHandler()`，则调用该类的 `DefaultHandler()`；如果两者都不存在，则继续在其基类中寻找，直至寻找到 `TObject` 这一层，而 `TObject` 已经提供了默认的 `DefaultHandler()` 方法。

先来看一个示例程序，它演示了消息分发及处理的过程。该程序的代码及可执行文件可在配书光盘的 `MsgDisp` 目录下找到。

首先自定义一个消息结构 `TMyMsg`，它是我们自定义的消息记录类型。对于自定义的消息类型，VCL 只规定它的首 4 字节必须是消息编号，其后的数据类型任意。同时，VCL 也提供了一个 `TMessage` 类型用于传递消息。在此程序中，不使用 `TMessage`，而用 `TMyMsg` 代替：

```
type
  TMyMsg = record // 自定义消息结构
    Msg : Cardinal; // 首 4 字节必须是消息编号
    MsgText : ShortString; // 消息的文字描述
  end;
```

TMyMsg 记录类型的第 2 个域我们定义为 MsgText，由该域的字符串来给出对这个消息的具体描述信息。当然，这些信息都是由消息分发者给出的。

然后，定义一个类，由它接受外界发送给它的消息。这个类可以说明这个演示程序的核心问题。

```
TMsgAcceptor = class // 消息接收器类
private
  // 编号为 2000 的消息处理函数
  procedure AcceptMsg2000(var msg : TMyMsg); message 2000;
  // 编号为 2002 的消息处理函数
  procedure AcceptMsg2002(var msg : TMyMsg); message 2002;
public
  procedure DefaultHandler(var Message); override; //默认处理方法
end;
```

在 Object Pascal 中，指明类的某个方法为某一特定消息的处理函数，则在其后面添加 message 关键字与消息值，以此来通知编译器。正如上面类定义中的

```
procedure AcceptMsg2000(var msg : TMyMsg); message 2000;
```

指明 AcceptMsg2000()方法用来处理值为 2000 的消息，该消息以及参数将通过 msg 参数传递给处理函数。

TMsgAcceptor 类除提供了值为 2000 和 2002 的两个消息的处理函数外，还提供了一个默认的消息处理方法 DefaultHandler()。该方法是在 TObject 中定义的虚方法，而在 TMsgAcceptor 类中覆盖 (override) 了该方法，重新给出了新的实现。

TMyMsg 结构声明与 TMsgAcceptor 类的声明与实现都被定义在 MsgDispTest 单元中。完整的单元代码如下，请参看其中的 TMsgAcceptor 类的各方法的实现：

```
unit MsgDispTest;

interface

uses Dialogs, Messages;

type
```



```
TMyMsg = record
    Msg : Cardinal;
    MsgText : ShortString;
end;

TMsgAcceptor = class // 消息接收器类
private
    procedure AcceptMsg2000(var msg : TMyMsg); message 2000;
    procedure AcceptMsg2002(var msg : TMyMsg); message 2002;
public
    procedure DefaultHandler(var Message); override; //默认处理函数
end;

implementation

{ TMsgAcceptor }

procedure TMsgAcceptor.AcceptMsg2000(var msg: TMyMsg);
begin
    ShowMessage('嗨，我收到了编号为 2000 的消息，它的描述是：' + msg.MsgText);
end;

procedure TMsgAcceptor.AcceptMsg2002(var msg: TMyMsg);
begin
    ShowMessage('嗨，我收到了编号为 2002 的消息，它的描述是：' + msg.MsgText);
end;

procedure TMsgAcceptor.DefaultHandler(var message);
begin
    ShowMessage('嗨，这个消息我不认识，无法接收，它的描述是：' +
        TMyMsg(message).MsgText);
end;

end.
```

接着就是界面代码，我们在 **Application** 的主 Form (Form1) 上放入 3 个按钮，程序界面如图 4.2 所示。

界面上的 3 个按钮的名字分别是：**btnMsg2000**、**btnMsg2001**、**btnMsg2002**。该 3 个按钮用来分发 3 个消息，将 3 个消息的值分别定义为 2000、2001 和 2002。

在 Form 的 **OnCreate** 事件中，创建一个 **TMsgAcceptor** 类的实例。然后，在 3 个按钮的 **OnClick** 事件中分别加上代码，将 3 个不同的消息分发给 **TMsgAcceptor** 类的实例对象，以

观察 TMsgAcceptor 作出的反应。最后，在 Form 的 OnDestroy 事件中，析构 TMsgAcceptor 类的实例对象。

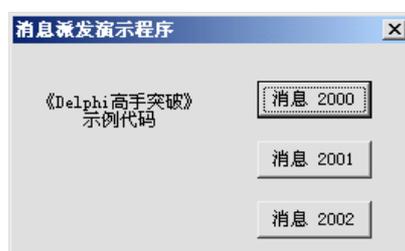


图 4.2 消息分发演示程序界面

完整的界面程序单元代码如下：

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, MsgDispTest;

type
  TForm1 = class(TForm)
    btnMsg2000: TButton;
    btnMsg2001: TButton;
    btnMsg2002: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnMsg2000Click(Sender: TObject);
    procedure btnMsg2002Click(Sender: TObject);
    procedure btnMsg2001Click(Sender: TObject);
  end;

var
  Form1: TForm1;
  MsgAccept : TMsgAcceptor; // 自定义的消息接收类

implementation

{$R *.dfm}
```



```
procedure TForm1.FormCreate(Sender: TObject);
begin
    // 创建 TMsgAcceptor 类的实例
    MsgAccept := TMsgAcceptor.Create();
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    // 析构 TMsgAcceptor 类的实例
    MsgAccept.Free();
    MsgAccept := nil;
end;

procedure TForm1.btnMsg2000Click(Sender: TObject);
var
    Msg : TMyMsg;
begin
    // 将值为 2000 的消息分发给 MsgAccept 对象, 观察其反应
    Msg.Msg := 2000;
    Msg.MsgText := 'Message 2000'; // 消息的文字描述
    MsgAccept.Dispatch(Msg); // 分发消息
end;

procedure TForm1.btnMsg2002Click(Sender: TObject);
var
    Msg : TMyMsg;
begin
    // 将值为 2002 的消息分发给 MsgAccept 对象, 观察其反应
    Msg.Msg := 2002;
    Msg.MsgText := 'Message 2002'; // 消息的文字描述
    MsgAccept.Dispatch(Msg); // 分发消息
end;

procedure TForm1.btnMsg2001Click(Sender: TObject);
var
    Msg : TMyMsg;
begin
    // 将值为 2001 的消息分发给 MsgAccept 对象, 观察其反应
    Msg.Msg := 2001;
    Msg.MsgText := 'Message 2001'; // 消息的文字描述
    MsgAccept.Dispatch(Msg); // 分发消息
end;
```

```
end;  
  
end.
```

在 `TMsgAcceptor` 类的代码中可以看到，它只能处理编号为 2000 和 2002 的消息，而没有编号为 2001 的消息的处理函数，但它覆盖了 `TObject` 的 `DefaultHandler()`，于是就提供了默认的消息处理函数。

运行程序，分别单击 3 个按钮，得到了 3 句不同的回答。对于消息 2000 和 2002，`TMsgAcceptor` 照单全收，正确识别出所接收到的消息。而只有在接收消息 2001 时，由于没有提供专门的消息处理函数，导致了对 `DefaultHandler()` 的调用。幸运的是，在 `DefaultHandler` 中，还可以使用 `message` 参数给出的附加信息（`TMyMsg` 记录类型中的 `MsgText` 域）。

### 4.3 TControl 与 Windows 消息的封装

`TObject` 提供了最基本的消息分发和处理的机制，而 VCL 真正对 Windows 系统消息的封装则是在 `TControl` 中完成的。

`TControl` 将消息转换成 VCL 的事件，以将系统消息融入 VCL 框架中。

消息分发机制在 4.2 节已经介绍过，那么系统消息是如何变成事件的呢？

现在，通过观察 `TControl` 的一个代码片段来解答这个问题。在此只以鼠标消息变成鼠标事件的过程来解释，其余的消息封装基本类似。

先摘取 `TControl` 声明中的一个片段：

```
TControl = class(TComponent)  
Private  
    .....  
    FOnMouseDown: TMouseEvent;  
    .....  
    procedure DoMouseDown(var Message: TWMMouse; Button: TMouseButton;  
        Shift: TShiftState);  
    .....  
    procedure MouseDown(Button: TMouseButton; Shift: TShiftState;  
        X, Y: Integer); dynamic;  
    .....  
    procedure WMLButtonDown(var Message: TWMLButtonDown); message  
        WM_LBUTTONDOWN;  
    procedure WMRButtonDown(var Message: TWMRButtonDown); message  
        WM_RBUTTONDOWN;  
    procedure WMMButtonDown(var Message: TWMMButtonDown); message
```



```
        WM_MBUTTONDOWN;  
        .....  
protected  
        .....  
        property OnMouseDown: TMouseEvent read FOnMouseDown write  
            FOnMouseDown;  
        .....  
end;
```

这段代码是 TControl 组件类的声明。如果你从没有接触过类似的 VCL 组件代码的代码，不明白那些 property、read、write 的意思，那么可以先跳转到 5.1 节阅读一下相关的基础知识，然后再回过头来到此处继续。

TControl 声明了一个 OnMouseDown 属性，该属性读写一个称为 FOnMouseDown 的事件指针。因此，FOnMouseDown 会指向 OnMouseDown 事件的用户代码。

TControl 声明了 WMLButtonDown、WMRButtonDown、WMButtonDown 3 个消息处理函数，它们分别处理 WM\_LBUTTONDOWN、WM\_RBUTTONDOWN、WM\_MBUTTONDOWN 3 个 Windows 消息，对应于鼠标的左键按下、右键按下、中键按下 3 个硬件事件。

另外，还有一个 DoMouseDown() 方法和一个 MouseDown() 的 dynamic 方法，它们与消息处理函数之间 2 是什么样的关系呢？

现在，就来具体看一下这些函数的实现。

这里是 3 个消息的处理函数：

```
procedure TControl.WMLButtonDown(var Message: TWMLButtonDown);  
begin  
    SendCancelMode(Self);  
    inherited;  
    if csCaptureMouse in ControlStyle then  
        MouseCapture := True;  
    if csClickEvents in ControlStyle then  
        Include(FControlState, csClicked);  
    DoMouseDown(Message, mbLeft, []);  
end;  
  
procedure TControl.WMRButtonDown(var Message: TWMRButtonDown);  
begin  
    inherited;  
    DoMouseDown(Message, mbRight, []);  
end;
```

```
procedure TControl.WMButtonDown(var Message: TWMMButtonDown);
begin
    inherited;
    DoMouseDown(Message, mbMiddle, []);
end;
```

当 TObject.Dispatch() 将 WM\_LBUTTONDOWN 消息、WM\_RBUTTONDOWN 消息或 WM\_MBUTTONDOWN 消息分发给 TControl 的派生类的实例后，WMLButtonDown()、WMRButtonDown() 或 WMMButtonDown() 被执行，然后它们都有类似这样

```
DoMouseDown(Message, mbRight, []);
```

的代码来调用 DoMouseDown():

```
procedure TControl.DoMouseDown(var Message: TWMMouse; Button:
TMouseButton; Shift: TShiftState);
begin
    if not (csNoStdEvents in ControlStyle) then
        with Message do
            if (Width > 32768) or (Height > 32768) then
                with CalcCursorPos do
                    MouseDown(Button, KeysToShiftState(Keys) + Shift, X,
Y)
            else
                MouseDown(
                    Button,
                    KeysToShiftState(Keys) + Shift,
                    Message.XPos,
                    Message.Ypos
                );
end;
```

在 DoMouseDown() 中进行一些必要的处理工作后（特殊情况下重新获取鼠标位置），就会调用 MouseDown():

```
procedure TControl.MouseDown(Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
    if Assigned(FOnMouseDown) then
        FOnMouseDown(Self, Button, Shift, X, Y);
end;
```



在 `MouseDown()` 中，才会通过 `FOnMouseDown` 事件指针真正去执行用户定义的 `OnMouseDown` 事件的代码。

由此，完成了 Windows 系统消息到 VCL 事件的转换过程。

因此，从 `TControl` 派生的类都可以拥有 `OnMouseDown` 事件，只不过该事件属性在 `TControl` 中被定义成 `protected`，只有其派生类可见，并且在派生类中可以自由选择是否公布这个属性。要公布该属性只需要简单地将其声明为 `published` 即可。如：

```
TMyControl = class(TControl)
published
  property OnMouseDown;
end;
```

这些函数过程的调用关系如图 4.3 所示。

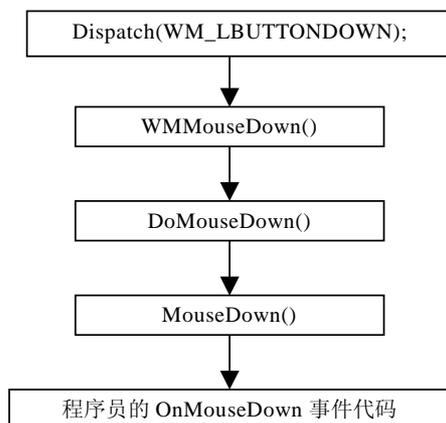


图 4.3 WM\_LBUTTONDOWN 消息到 OnMouseDown 事件的转换过程

在此，只是以 `OnMouseDown` 事件为例。其实，VCL 对 Windows 各个消息的封装大同小异，以此一例足以说明事件模型的原理。

另外，值得注意的是，在上例中的 `MouseDown()` 函数是一个 `dynamic` 方法，因此可以通过在 `TControl` 派生类中覆盖 `MouseDown()` 来处理自己所编写组件的鼠标按下事件，然后通过

```
inherited;
```

语句调用 `TControl` 的 `MouseDown()` 来执行使用组件的程序员所编写的 `OnMouseDown` 的代码。具体内容会在第 5 章中展开。

至此，读者应该已经了解了 VCL 事件与 Windows 消息的对应关系，应该知道平时为组件写的事件代码是如何被执行的。

如果读者感到自己对此还不是很清楚，那么建议您将本节与 4.2 节再多读几遍，甚至可以自己打开 Delphi 亲自查看一下 VCL 的源代码，相信很快就会明白的。

## 4.4 TApplication 与主消息循环

现在已经明白了 VCL 消息分发机制以及 VCL 的事件模型，但如果曾经使用纯 API 编写过 Windows 程序，一定知道 Windows 应用程序的每一个窗口都有一个大的消息循环以及一个窗口函数（WndProc）用以分发和处理消息。

VCL 作为一个 Framework，当然会将这些东西隐藏起来，而重新提供一种易用的、易理解的虚拟机制给程序员。

那么 VCL 是如何做到的呢？

本节就来解答这个问题。

只要代码单元中包含了 Forms.pas，就会得到一个对象——Application。利用它可以帮助我们完成许多工作。例如要退出应用程序，可以使用

```
Application.Terminate();
```

Application 对象是 VCL 提供的，在 Forms.pas 中可以看到如下这个定义：

```
var  
  Application: TApplication;
```

从表现来看，TApplication 类定义了一个应用程序的特性及行为，可以从 Application 对象得到应用程序的可执行文件名称（ExeName），设置应用程序的标题（Title）等属性，也可以执行最小化（Minimize）、打开帮助文件（HelpCommand）等操作。

当创建一个默认的应用程序时，会自动得到以下几行代码：

```
begin  
  Application.Initialize;  
  Application.CreateForm(TForm1, Form1);  
  Application.Run;  
end.
```

这几行代码很简洁地展示了 TApplication 的功能、初始化、创建必要的窗体、运行……

但是，这几行代码具体做了什么幕后操作呢？Application.Run 之后，程序流程走向了哪里？

### 4.4.1 脱离 VCL 的 Windows 程序

读者有必要先了解一个标准 Windows 程序的运行流程。如果现在还不了解，请看下面的一个示例程序。在此，给出一个用纯 Pascal 所编写的十分简单的 Windows 应用程序，以



演示标准 Windows 程序是如何被建立及运行的。该程序的代码及可执行文件可在配书光盘的 WindowDemo 目录下找到，程序可被 Delphi 编译通过。

以下是代码清单，请注意其中的注释：

```
program WindowDemo;

uses Windows, Messages;

// 窗口函数，窗口接到消息时被 Windows 所调用
function WindowProc(hwnd : HWND; uMsg : Cardinal; wParam : WPARAM;
    lParam : LPARAM) : LResult; stdcall;
begin
    Result := 0;

    case uMsg of
        // 关闭窗口消息，当用户关闭窗口后，通知主消息循环结束程序
        WM_CLOSE : PostMessage(hwnd, WM_QUIT, 0, 0);
        // 鼠标左键按下消息
        WM_LBUTTONDOWN : MessageBox(hwnd, 'Hello!', '和您打个招呼',
            MB_ICONINFORMATION);

    else
        // 其他消息做默认处理
        Result := DefWindowProc(hWnd, uMsg, wParam, lParam);
    end;
end;

var
    wndcls : WNDCLASS; // 窗口类的记录（结构）类型
    hWnd : THandle;
    Msg : tagMSG; // 消息类型
begin
    wndcls.style := CS_DBLCLKS; // 允许窗口接受鼠标双击
    wndcls.lpfnWndProc := @WindowProc; // 为窗口类指定窗口函数
    wndcls.cbClsExtra := 0;
    wndcls.cbWndExtra := 0;
    wndcls.hInstance := hInstance;
    wndcls.hIcon := 0;
    wndcls.hCursor := LoadCursor(hInstance, 'IDC_ARROW');
    wndcls.hbrBackground := COLOR_WINDOWFRAME;
    wndcls.lpszMenuName := nil;
```

```
wndcls.lpszClassName := 'WindowClassDemo'; // 窗口类名称

// 注册窗口类
if RegisterClass(wndcls) = 0 then
    Exit;

// 创建窗口
hWnd := CreateWindow(
    'WindowClassDemo', // 窗口类名称
    'WindowDemo',      // 窗口名称
    WS_BORDER or WS_CAPTION or WS_SYSMENU, // 窗口类型
    Integer(CW_USEDEFAULT),
    Integer(CW_USEDEFAULT),
    Integer(CW_USEDEFAULT),
    Integer(CW_USEDEFAULT),
    0,
    0,
    hInstance,
    nil
);
if hWnd = 0 then
    Exit;

// 显示窗口
ShowWindow(hWnd, SW_SHOWNORMAL);
UpdateWindow(hWnd);

// 创建主消息循环, 处理消息队列中的消息并分发
// 直至收到 WM_QUIT 消息, 退出主消息循环, 并结束程序
// WM_QUIT 消息由 PostMessage() 函数发送
while GetMessage(Msg, hWnd, 0, 0) do
begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
end;
end.
```

该程序没有使用 VCL, 它所做的事情就是显示一个窗口。当在窗口上单击鼠标右键时, 会弹出一个友好的对话框向您问好。如果从来不曾了解过这些, 那么建议您实际运行一下光盘上的这个程序, 对其多一些感性认识。

就是这样一个简单的程序, 演示了标准 Windows 程序的流程:



- (1) 从入口函数 `WinMain` 开始。
- (2) 注册窗口类及窗口函数 (Window Procedure)。
- (3) 创建并显示窗口。
- (4) 进入主消息循环, 从消息队列中获取并分发消息。
- (5) 消息被分发后, 由 Windows 操作系统调用窗口函数, 由窗口函数对消息进行处理。

在 Object Pascal 中看不到所谓的“WinMain”函数。不过, 其实整个 program 的 begin 处就是 Windows 程序的入口。

注册窗口类通过系统 API 函数 `RegisterClass()`来完成, 它向 Windows 系统注册一个窗口的类型。

注册窗口类型完成后, 就可以创建这个类型的窗口实例。创建出一个真正的窗口可通过 API 函数 `CreateWindow()`来实现。

创建出的窗口实例通过 API 函数 `ShowWindow()`来使得它显示在屏幕上。

当这一切都完成后, 窗口开始进入一个 while 循环以处理各种消息, 直至 API 函数 `GetMessage()`返回 0 才退出程序。循环中, 程序需要从主线程的消息队列中取出各种消息, 并将它分发给系统, 然后由 Windows 系统调用窗口的窗口函数 (`WndProc`), 以完成窗口对消息的响应处理。

也许有人会觉得, 写一个 Windows 应用程序原来是那么繁琐, 需要调用大量的 API 函数来完成平时看起来很简单的事情, 而平时使用 VCL 编写窗口应用程序时, 似乎从来没有遇到过这些东西。是的, VCL 作为一个 Framework 为我们做了很多事情, 其中的 `TApplication` 除了定义一个应用程序的特性及行为外, 另一个重要的使命就是封装以上的那些令人讨厌的、繁琐的步骤。

那它是如何做到的呢?

### 4.4.2 Application 对象的本质

在 Delphi 中, 我们为每个项目 (非 DLL 项目, 以下讨论皆是) 所定义的 Main Form 并不是主线程的主窗口。每个 Application 的主线程的主窗口 (也就是出现在系统任务栏中的) 是由 `TApplication` 创建的一个  $0 \times 0$  大小的不可见的窗口, 但它可以出现在任务栏上。其余由程序员创建的 Form, 都是该窗口的子窗口。

程序员所定义的 Main Form 由 Application 对象来调度。Delphi 所编写的应用程序有时会出现如图 4.4 所示的情况: 任务栏标题和程序主窗口标题不一致, 这也可以证明其实它们并非同一个窗口。这两个标题分别由 `Application.Title` 和 Main Form (如 `Form1`) 的 `Caption` 属性所设置。

另外, 还可以通过它们的句柄来了解它们的实质。MainForm (如 `Form1`) 的 `Handle` 所返回的, 是窗体的窗口句柄; `Application.Handle` 所返回的, 却是这个  $0 \times 0$  大小的窗口句柄。

因此, 我们可以粗略地认为, Application 其实是一个窗口!

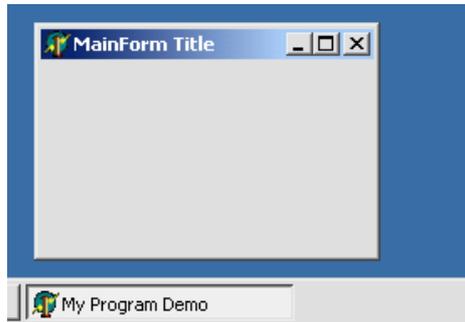


图 4.4 主窗口标题与任务栏标题不一致

4

 **注意：** Application 是一个 0\*0 大小的不可见窗口！

TApplication 类的代码可作为证明。在 TApplication 的构造函数中有这样一行代码：

```
if not IsLibrary then CreateHandle;
```

在非 DLL 项目中，构造函数会调用 CreateHandle 方法。查看该方法源代码可知，该方法的任务正是注册窗口类，并创建一个窗口实例。以下是 CreateHandle 的代码，请注意其中所加的注释：

```
procedure TApplication.CreateHandle;
var
  TempClass: TWndClass;
  SysMenu: HMenu;
begin
  if not FHandleCreated and not IsConsole then
  begin
    FObjectInstance := Classes.MakeObjectInstance(WndProc);

    // 如果窗口类不存在，则注册窗口类
    if not GetClassInfo(HInstance,
      WindowClass.lpszClassName,
      TempClass
    ) then
    begin
      WindowClass.hInstance := HInstance;
      if Windows.RegisterClass(WindowClass) = 0 then
        raise EOutOfResources.Create(SWindowClass);
    end;

    // 创建窗口，长度和宽度都是 0，位置在屏幕中央，返回的句柄 FHandle
    // 也就是 TApplication.Handle 的值
```



```
FHandle := CreateWindow(WindowClass.lpszClassName,
PChar(FTitle),
    WS_POPUP or WS_CAPTION or WS_CLIPSIBLINGS or WS_SYSMENU
    or WS_MINIMIZEBOX,
    GetSystemMetrics(SM_CXSCREEN) div 2,
    GetSystemMetrics(SM_CYSCREEN) div 2,
    0,
    0,
    0,
    0,
    HInstance,
    Nil
);

FTitle := '';
FHandleCreated := True;

// 调用 SetWindowLong 设置窗口的窗口函数 (WndProc), 下文会详述
SetWindowLong(FHandle, GWL_WNDPROC, Longint(FObjectInstance));

if NewStyleControls then
begin
    SendMessage(FHandle, WM_SETICON, 1, GetIconHandle);
    SetClassLong(FHandle, GCL_HICON, GetIconHandle);
end;
SysMenu := GetSystemMenu(FHandle, False);
DeleteMenu(SysMenu, SC_MAXIMIZE, MF_BYCOMMAND);
DeleteMenu(SysMenu, SC_SIZE, MF_BYCOMMAND);
If NewStyleControls then
    DeleteMenu(SysMenu, SC_MOVE, MF_BYCOMMAND);
end;
end;
```

对照一下此前使用纯 API 编写的窗口程序，就会发现一些它们的相似之处。在 `CreateHandle()` 中，可以看到熟悉的 `RegisterClass()`、`CreateWindow()` 等 API 函数的调用。比较特别的是，`CreateHandle()` 中通过 API 函数 `SetWindowLong()` 来设置窗口的窗口函数：

```
SetWindowLong(FHandle, GWL_WNDPROC, Longint(FObjectInstance));
```

此时，`SetWindowLong()` 的第 3 个参数为窗口函数实例的地址，其中 `FObjectInstance` 是由 `CreateHandle()` 的第 1 行代码

```
FObjectInstance := Classes.MakeObjectInstance(WndProc);
```

所创建的实例的指针，而 `WndProc()` 则成了真正的窗口函数。具体关于 `WndProc()` 的实现，将在 4.4.4 节叙述。

`TApplication` 本身有一个 `private` 成员 `FMainForm`，它指向程序员所定义的主窗体，并在 `TApplication.CreateForm` 方法中判断并赋值：

```
procedure TApplication.CreateForm(InstanceClass: TComponentClass;
  var Reference);
var
  Instance: TComponent;
begin
  Instance := TComponent(InstanceClass.NewInstance);
  ..... // 创建窗体实例的代码省略

  // 第一个创建的窗体实例就是 MainForm
  if (FMainForm = nil) and (Instance is TForm) then
  begin
    TForm(Instance).HandleNeeded;
    FMainForm := TForm(Instance);
  end;
end;
```

因此，Delphi 为每个应用程序自动生成的代码中就有对 `CreateForm` 的调用，如：

```
Application.CreateForm(TForm1, Form1);
```

值得注意的是，如果有一系列的多个 `CreateForm` 的调用，则第一个调用 `CreateForm` 被创建的窗体，就是整个 `Application` 的 `MainForm`。这一点从 `CreateForm` 的代码中不难看出。在 `Project` 的 `Options` 中设置 `MainForm`，Delphi 的 IDE 会自动调整代码。

明白了 `Application` 的本质之后，再来看一下它是如何建立主消息循环的。

### 4.4.3 TApplication 创建主消息循环

在 `TApplication` 的 `CreateHandle` 方法中可以看到，`SetWindowLong()` 的调用将 `TApplication.WndProc` 设置成了那个 `0×0` 大小窗口的窗口函数。

也就是说，在 `TApplication` 的构造函数中主要完成了两件事情：注册窗口类及窗口函数，创建 `Application` 窗口实例。

那接下来应该就是进入主消息循环了？是的，这也就是 `Application.Run` 方法所完成的事情。`TApplication` 类的 `Run` 方法中有这样一段代码：

```
repeat
  try
```



```
    HandleMessage;  
except  
    HandleException(Self);  
end;  
until Terminated;
```

是的，这就是主消息循环。看上去似乎没有取消息、分发消息的过程，其实它们都被包含在 `HandleMessage()` 方法中了。`HandleMessage()` 方法其实是对 `ProcessMessage()` 方法的调用，而在 `ProcessMessage()` 中就可以看到取消息、分发消息的动作了。以下是 `TApplication` 的 `ProcessMessage()` 方法的源代码，请注意其中的注释：

```
function TApplication.ProcessMessage(var Msg: TMsg): Boolean;  
var  
    Handled: Boolean;  
begin  
    Result := False;  
    // 取消息  
    if PeekMessage(Msg, 0, 0, 0, PM_REMOVE) then  
    begin  
        Result := True;  
        if Msg.Message <> WM_QUIT then  
        begin  
            Handled := False;  
            if Assigned(FOnMessage) then FOnMessage(Msg, Handled);  
            if (  
                not IsHintMsg(Msg) and  
                not Handled and  
                not IsMDIMsg(Msg) and  
                not IsKeyMsg(Msg) and  
                not IsDlgMsg(Msg)  
            ) then  
            begin  
                // 熟悉的分发消息过程  
                TranslateMessage(Msg);  
                DispatchMessage(Msg);  
            end;  
        end  
    else  
        // 如果取到的消息为 WM_QUIT，则将 Fterminate 设为真  
        // 以通知主消息循环退出  
        // 这和 WindowDemo 程序中判断 GetMessage() 函数返回值是否为 0 等效
```

```
    // 因为 GetMessage() 函数取出的消息如果是 WM_QUIT, 它的返回值为 0
    FTerminate := True;

end;
end;
```

ProcessMessage()方法清楚地显示了从消息队列取消息并分发消息的过程, 并且当取到的消息为 WM\_QUIT 时, 则将 FTerminate 置为 True, 标志程序退出。

4

#### 4.4.4 窗口函数 (WndProc) 处理消息

窗口函数是一个回调函数, 它被 Windows 系统所调用, 其参数会被给出消息编号、消息参数等信息, 以便进行处理。

典型的窗口函数中会包含一个大的 case 分支, 以处理不同的消息。

在 4.4.2 节中分析 TApplication.CreateHandle()的代码时提到过, CreateHandle()将 Application 窗口的窗口函数设置为 WndProc()。那么, 现在就来看一下这个 WndProc, 请注意其中的注释:

```
procedure TApplication.WndProc(var Message: TMessage);
type // 函数内嵌定义的类型, 只限函数内部使用
  TInitTestLibrary = function(Size: DWord; PAutoClassInfo: Pointer):
    Boolean; stdcall;

var
  I: Integer;
  SaveFocus, TopWindow: HWND;
  InitTestLibrary: TInitTestLibrary;

  // 内嵌函数, 默认的消息处理
  // 调用 Windows 的 API 函数 DefWindowProc
  procedure Default;
  begin
    with Message do
      Result := DefWindowProc(FHandle, Msg, WParam, LParam);
  end;

  procedure DrawAppIcon;
  var
    DC: HDC;
    PS: TPaintStruct;
  begin
    with Message do
```



```
begin
    DC := BeginPaint(FHandle, PS);
    DrawIcon(DC, 0, 0, GetIconHandle);
    EndPaint(FHandle, PS);
end;
end;

begin
    try
        Message.Result := 0;
        for I := 0 to FWindowHooks.Count - 1 do
            if TWindowHook(FWindowHooks[I]^)(Message) then Exit;
        CheckIniChange(Message);
        with Message do

            // 开始庞大的 case 分支, 对不同的消息做出不同的处理
            case Msg of
                WM_SYSCOMMAND:
                    case WParam and $FFF0 of
                        SC_MINIMIZE: Minimize;
                        SC_RESTORE: Restore;
                    else
                        Default;
                    end;
                end;
                WM_CLOSE:
                    if MainForm <> nil then MainForm.Close;
                WM_PAINT:
                    if IsIconic(FHandle) then DrawAppIcon else Default;
                WM_ERASEBKGD:
                    begin
                        Message.Msg := WM_ICONERASEBKGD;
                        Default;
                    end;
                WM_QUERYDRAGICON:
                    Result := GetIconHandle;
                WM_SETFOCUS:
                    begin
                        PostMessage(FHandle, CM_ENTER, 0, 0);
                        Default;
                    end;
                WM_ACTIVATEAPP:
                    begin
```

```
Default;
FActive := TWMActivateApp(Message).Active;
if TWMActivateApp(Message).Active then
begin
  RestoreTopMosts;
  PostMessage(FHandle, CM_ACTIVATE, 0, 0)
end
else
begin
  NormalizeTopMosts;
  PostMessage(FHandle, CM_DEACTIVATE, 0, 0);
end;
end;
WM_ENABLE:
if TWMEnable(Message).Enabled then
begin
  RestoreTopMosts;
  if FWindowList <> nil then
begin
  EnableTaskWindows(FWindowList);
  FWindowList := nil;
end;
Default;
end else
begin
Default;
if FWindowList = nil then
  FWindowList := DisableTaskWindows(Handle);
  NormalizeAllTopMosts;
end;
WM_CTLCOLORMSGBOX..WM_CTLCOLORSTATIC:
Result := SendMessage(LParam, CN_BASE + Msg, WParam, LParam);
WM_ENDSESSION:
if TWMEndSession(Message).EndSession then FTerminate := True;
WM_COPYDATA:
if (PCopyDataStruct(Message.lParam)^.dwData =
DWORD($DE534454))
and (FAllowTesting) then
if FTestLib = 0 then
begin
  FTestLib := SafeLoadLibrary('vcltest3.dll');
if FTestLib <> 0 then
```



```
begin
    Result := 0;
    @InitTestLibrary := GetProcAddress(
        FTestLib,
        'RegisterAutomation'
    );
    if @InitTestLibrary <> nil then
        InitTestLibrary(
            PCopyDataStruct(Message.lParam)^.cbData,
            PCopyDataStruct(Message.lParam)^.lpData
        );
    end
    else
    begin
        Result := GetLastError;
        FTestLib := 0;
    end;
end
else
    Result := 0;
CM_ACTIONEXECUTE, CM_ACTIONUPDATE:
    Message.Result := Ord(DispatchAction(
        Message.Msg,
        TBasicAction(Message.LParam)
    ));
CM_APPKEYDOWN:
    if IsShortCut(TWMKey(Message)) then Result := 1;
CM_APPSYSCOMMAND:
    if MainForm <> nil then
        with MainForm do
            if (Handle <> 0) and IsWindowEnabled(Handle) and
                IsWindowVisible(Handle) then
                begin
                    FocusMessages := False;
                    SaveFocus := GetFocus;
                    Windows.SetFocus(Handle);
                    Perform(WM_SYSCOMMAND, WParam, LParam);
                    Windows.SetFocus(SaveFocus);
                    FocusMessages := True;
                    Result := 1;
                end;
    end;
CM_ACTIVATE:
```

```
    if Assigned(FOnActivate) then FOnActivate(Self);
CM_DEACTIVATE:
    if Assigned(FOnDeactivate) then FOnDeactivate(Self);
CM_ENTER:
    if not IsIconic(FHandle) and (GetFocus = FHandle) then
    begin
        TopWindow := FindTopMostWindow(0);
        if TopWindow <> 0 then Windows.SetFocus(TopWindow);
    end;
WM_HELP, // MessageBox(... MB_HELP)
CM_INVOKEHELP: InvokeHelp(WParam, LParam);
CM_WINDOWHOOK:
    if wParam = 0 then
        HookMainWindow(TWindowHook(Pointer(LParam)^)) else
        UnhookMainWindow(TWindowHook(Pointer(LParam)^));
CM_DIALOGHANDLE:
    if wParam = 1 then
        Result := FDialogHandle
    else
        FDialogHandle := lParam;
WM_SETTINGCHANGE:
    begin
        Mouse.SettingChanged(wParam);
        SettingChange(TWMSettingChange(Message));
        Default;
    end;
WM_FONTCHANGE:
    begin
        Screen.ResetFonts;
        Default;
    end;
WM_NULL:
    CheckSynchronize;
else
    Default;
end;
except
    HandleException(Self);
end;
end;
```

整个 WndProc()方法，基本上只包含了一个庞大的 case 分支，其中给出了每个消息的



处理代码，“WM\_”打头的为 Windows 定义的窗口消息，“CM\_”打头的为 VCL 库自定义的消息。

需要注意的是，这里给出 WndProc 是属于 TApplication 的，也就是那个 0×0 大小的 Application 窗口的窗口函数，而每个 Form 另外都有自己的窗口函数。

至此，读者应该清楚了 VCL 框架是如何封装 Windows 程序框架的了。知道 VCL 为我们做了什么，它想要提供给我们的是怎样的一个世界，这对于我们更好地融入 VCL 是大有好处的。这比从 RAD 角度看待 VCL，有了更深一层的理解。好了，关于 VCL 和消息的话题到此为止。

## 4.5 TPersistent 与对象赋值

在 Object Pascal 中，所有的简单类型（或称编译器内置类型，即非“类”类型，如 Integer、Cardinal、Char、Record 等类型）的赋值操作所进行的都是位复制，即将一个变量所在的内存空间的二进制位值复制到被赋值的变量所载的内存空间中。

如定义这样一个记录类型：

```
type
  TExampleRec = record
    Member1 : Integer;
    Member2 : Char;
  end;
```

在代码中，声明例如两个 TExampleRec 类型的变量实体，并在它们之间进行赋值：

```
var
  A, B : TExampleRec;
begin
  A.Member1 := 1;
  A.Member2 := 'A';
  B := A;
end;
```

其中，B := A;的结果将导致 A 的所有值都被复制到 B 中，A 和 B 各自拥有一份它们的值。查看这段代码的编译结果：

```
mov [esp], $00000001      // A.Member1 := 1;
mov byte ptr [esp + $04], $41 // A.Member2 := 'A';
mov eax, [esp]           // B.Member1 := A.Member1
mov [esp + $08], eax
```

```
mov eax, [esp + $04]           // B.Member2 := A.Member2
mov [esp + $0c], eax
```

就可以非常清楚地看到:

```
B := A;
```

与

```
B.Member1 := A.Member1;
B.Member2 := A.Member2;
```

是等价的。

对于简单类型,可以简单地以变量名称来进行赋值,那么对于所谓的复杂类型——“类”类型呢?

此前曾经提到过, Delphi 向 Object Pascal 引入了所谓的“引用/值”模型,即对于简单类型的变量,采用“值”模型,它们在程序中的传递方式全部是基于“值”进行的。而复杂类型的变量,即类的实例对象,采用“引用”模型,因此在程序中所有类的对象的传递,全部基于其“引用”,也就是对象的指针。

如果将两个对象通过名称直接进行简单的赋值,将导致对象指针的转移,而并非复制它们之间的内存空间的二进制值。例如,将上述的 TExampleRec 改成 Class 类型:

```
type
  TExample = class
  public
    Member1 : Integer;
    Member2 : Char;
  end;
```

并将赋值的代码改为:

```
var
  A, B : TExample;
begin
  A := TExample.Create();
  B := TExample.Create();
  ShowMessage(IntToStr(Integer(A))); // 输出 13513320
  ShowMessage(IntToStr(Integer(B))); // 输出 13513336
  A.Member1 := 1;
  A.Member2 := 'A';
  B := A;
```



```
ShowMessage(IntToStr(Integer(B))); // 输出 13513320  
.....
```

这段代码中的 3 个 ShowMessage 调用，将输出对象所在内存空间的地址值。可以很明显看到，第 3 个 ShowMessage 输出的 B 对象所在的内存地址已经指向了 A 对象所在内存地址。此时，B 和 A 所使用的数据将是同一份数据，若修改 A 的 Member1 的值，那么 B 的 Member1 也将同时被修改。而原先 B 所在的空间 (13513336) 已经失去了引用它的指针，于是就造成了所谓的“内存泄漏”。如图 4.5 所示。

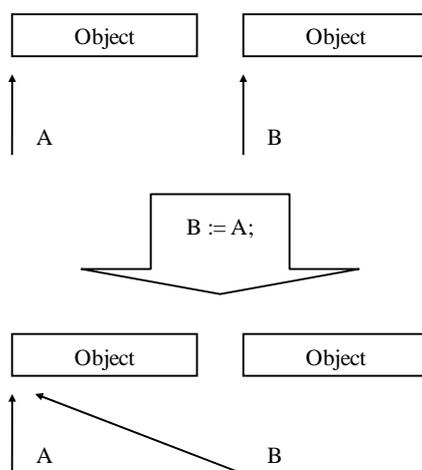


图 4.5 B:=A;的结果

可见，简单、直接地通过对象名称进行赋值是达不到复制对象的目的的。如果的确需要复制一个对象，那么难道真的要如同

```
B.Member1 := A.Member1;  
B.Member2 := A.Member2;
```

这样来进行吗？即使可以这样做，那 private 数据如何复制呢？

可以为类增加一个 Assign 方法，以进行对象间的复制。例如修改以上的 TExample 类：

```
type  
  TExample = class  
    Member1 : Integer;  
    Member2 : Char;  
  public  
    procedure Assign(Src : TExample);  
  end;
```

实现该类的 `Assign` 方法如下:

```
procedure TExample.Assign(Src: TExample);
begin
    Member1 := Src.Member1;
    Member2 := Src.Member2;
end;
```

如此便可以进行 `TExample` 类实例对象间的复制:

```
var
    A, B : TExample;
begin
    A := TExample.Create();
    B := TExample.Create();
    A.Member1 := 1;
    A.Member2 := 'A';
    B.Assign(A);
    .....
```

如此庞大的 VCL 库中,肯定需要提供这样一种机制来保证对象间的有效赋值,于是 VCL 提供了一个抽象类——`TPersistent`。

`TPersistent` 为对象间的复制式赋值定义了一套接口规范:

```
TPersistent = class(TObject)
private
    procedure AssignError(Source: TPersistent);
protected
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFile); virtual;
    function GetOwner: TPersistent; dynamic;
public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function GetNamePath: string; dynamic;
end;
```

在 `TPersistent` 的声明中,有两个 `Public` 的方法(`Destroy` 在此不讨论),其中 `GetNamePath` 是 Delphi 的集成开发环境内部使用的, VCL 不推荐直接对它的调用。而 `Assign` 方法则是为完成对象复制而存在的,并且被声明为虚方法,以允许每个派生类定义自己的复制对象的方法。



如果正在设计的类需要有这种允许对象复制的能力，则让类从 `TPersistent` 派生并重写 `Assign` 方法。

如果没有重写 `Assign` 方法，则 `TPersistent` 的 `Assign` 方法会将复制动作交给源对象来进行：

```
procedure TPersistent.Assign(Source: TPersistent);
begin
    if Source <> nil then
        Source.AssignTo(Self) // 调用源对象的 AssignTo 方法
    else
        AssignError(nil);
end;
```

可以在 `TPersistent` 类的声明的 `protected` 节中找到 `AssignTo` 方法的声明，它也是一个虚方法。

如果将复制动作交给源对象来完成，那么必须保证源对象的类已经重写了 `AssignTo` 方法，否则将抛出一个“Assign Error”异常：

```
procedure TPersistent.AssignTo(Dest: TPersistent);
begin
    Dest.AssignError(Self);
end;

procedure TPersistent.AssignError(Source: TPersistent);
var
    SourceName: string;
begin
    if Source <> nil then
        SourceName := Source.ClassName
    else
        SourceName := 'nil';
    raise EConvertError.CreateResFmt(
        @SAssignError,
        [SourceName, ClassName]
    );
end;
```

`AssignError` 是一个 `private` 方法，仅仅用于抛出赋值错误的异常。

在 `TPersistent` 的声明中，`GetOwner` 方法是被前面所述由 Delphi 内部使用的 `GetNamePath` 所调用。

最后还剩下一个虚方法 `DefineProperties()`，它则是为 `TPersistent` 的另一个使命而存在：

对象持久。一个对象要持久存在，就必须将它流化 (Streaming)，保存到一个磁盘文件 (.dfm 文件) 中。TPersistent 也使得其派生类具有这种能力，但它作为抽象类只是定义接口而并没有给出实现。可以看到，DefineProperties 是一个空的虚方法：

```
procedure TPersistent.DefineProperties(Filer: TFile);  
begin  
end;
```

这留待其派生类来实现。

对于对象持久的实现类，最典型的的就是 TComponent，每个组件都具有保存自己的能力。因此下面将以 TComponent 来说明对象持久的实现，虽然它是在 TPersistent 中定义接口的。

## 4.6 TComponent 与对象持久

Delphi IDE 的流系统用来保证所有 TPersistent 及其派生类的 published 的数据都会被自动保存和读取。而 TComponent 类派生自 TPersistent，所有组件都从 TComponent 派生，因此所有组件都具有自我保存、持久的能力，这是 Delphi IDE 的流系统所保证的。不过，这样的对象持久系统并不完善，至少，它无法保存对象的非 published 数据。

Delphi 当然会为这种情况提供解决方案，它就是 TPersistent 声明的 DefineProperties() 方法，是一个虚方法。在 TPersistent 的实现中，它是一个空方法。每个 TPersistent 的派生类需要保存非 published 数据的时候，就可以覆盖该方法。

VCL 的所有组件被放置在一个 Form 上之后，它的位置就会被记录下来。保存该 Form，后重新打开，所有放置的组件都还在原来的位置上，包括那些运行时不可见的组件，如 TTimer。这些组件并没有标识位值的“Left”或“Top”属性，那它们的位置信息是如何保存的呢？

可以在一个空白的 Form 上放置一个 TTimer 组件，并保存该 Form，然后打开该 Form 的定义文件 (如：Form1.dfm)，可以看到类似如下的内容：

```
object Form1: TForm1  
  Left = 192  
  Top = 107  
  Width = 696  
  Height = 480  
  Caption = 'Form1'  
  Color = clBtnFace  
  Font.Charset = DEFAULT_CHARSET  
  Font.Color = clWindowText  
  Font.Height = -11  
  Font.Name = 'MS Sans Serif'
```



```
Font.Style = []
OldCreateOrder = False
OnCreate = FormCreate
PixelsPerInch = 96
TextHeight = 13

object Timer1: TTimer
    Left = 160
    Top = 64
end
end
```

寻找到其中的 `object Timer1: TTimer` 这一行以及其后的数行：

```
object Timer1: TTimer
    Left = 160
    Top = 64
End
```

这几行记录了 `TTimer` 组件，可是很奇怪，`TTimer` 组件本身并没有所谓的“Left”和“Top”属性，为什么在 `dfm` 文件的定义中会出现呢？

“Left”和“Top”并非 `TTimer` 的 `published` 数据，因此它们肯定不是由 Delphi IDE 的流系统来保存的。

`TTimer` 组件派生自 `TComponent`，而 `TComponent` 正是通过重写了 `TPersistent` 的 `DefineProperties()` 方法来记录下 Form 上面组件的位置。

来查看一下被 `TComponent` 覆盖（overriding）了的 `DefineProperties()` 方法的代码：

```
procedure TComponent.DefineProperties(Filer: TFiler);
var
    Ancestor: TComponent;
    Info: Longint;
begin
    Info := 0;
    Ancestor := TComponent(Filer.Ancestor);
    if Ancestor <> nil then Info := Ancestor.FDesignInfo;
    Filer.DefineProperty('Left', ReadLeft, WriteLeft,
        LongRec(FDesignInfo).Lo <> LongRec(Info).Lo);
    Filer.DefineProperty('Top', ReadTop, WriteTop,
        LongRec(FDesignInfo).Hi <> LongRec(Info).Hi);
end;
```

这几行代码首先检查组件本身是否是从其他类派生的，因为如果存在祖先类而派生类本身没有改变要保存的属性值，该属性值就不必保存了。

然后通过传进的 TFiler 类的参数 Filer 来定义要保存的属性的读写方法：

```
Filer.DefineProperty('Left', ReadLeft, WriteLeft,  
    LongRec(FDesignInfo).Lo <> LongRec(Info).Lo);  
Filer.DefineProperty('Top', ReadTop, WriteTop,  
    LongRec(FDesignInfo).Hi <> LongRec(Info).Hi);
```

Filer.DefineProperty()方法的第 2、第 3 个参数分别是读写属性的方法。这两个方法的原型分别如下：

```
TReaderProc = procedure(Reader: TReader) of object;  
TWriterProc = procedure(Writer: TWriter) of object;
```

TComponent 类为保存“Left”和“Top”属性，分别提供了 ReadLeft/WriteLeft 和 ReadTop/WriteTop 方法：

```
procedure TComponent.ReadLeft(Reader: TReader);  
begin  
    LongRec(FDesignInfo).Lo := Reader.ReadInteger;  
end;  
  
procedure TComponent.ReadTop(Reader: TReader);  
begin  
    LongRec(FDesignInfo).Hi := Reader.ReadInteger;  
end;  
  
procedure TComponent.WriteLeft(Writer: TWriter);  
begin  
    Writer.WriteInteger(LongRec(FDesignInfo).Lo);  
end;  
  
procedure TComponent.WriteTop(Writer: TWriter);  
begin  
    Writer.WriteInteger(LongRec(FDesignInfo).Hi);  
end;
```

因此，每个 TComponent 的实例在被流化到 dfm 文件时，都会有 Left 和 Top 属性，即使组件并没有这两个属性。



## 4.7 TCanvas 与 Windows GDI

Windows 是一个图形操作系统，提供所谓的 GUI（图形用户界面）。为了使程序员能够实现 GUI 的程序，Windows 提供了一套 GDI（图形设备接口）的 API 函数。

VCL 作为对 Windows API 封装的框架类库，当然也对 GDI 进行了封装。GDI 作为 Windows API 的一个子集，本身却也非常庞大，涉及了与各种图形相关的内容，如画笔（Pens）、刷子（Brushes）、设备上下文（Device Contexts）、位图（Bitmap）以及字体、颜色等。在 VCL 中，与 GDI 相关的类、函数基本都被实现在 Graphics.pas 的单元中。

常用的 GDI 对象无非就是画笔、刷子、位图等，VCL 首先对这些 GDI 的基本对象进行了抽象，然后以这些基本对象辅助 TCanvas 实现对 GDI 的全面封装。

下面，先来研究一下那些基本对象——TPen、TBrush。

### 4.7.1 TPen

Windows 中，创建一个笔（Pen）对象，使用 API 函数 CreatePenIndirect()或 CreatePen()。CreatePen()的原型如下：

```
HPEN CreatePen(  
    int fnPenStyle,    // Pen 风格  
    int nWidth,       // 宽度  
    COLORREF crColor  // 颜色  
);
```

该函数返回一个笔对象的句柄。

要在窗口上画出一条两个像素宽度的红色直线，使用 Windows API 来完成的代码可能是这样的：

```
var  
    hOldPen : HPEN;  
    hNewPen : HPEN;  
    DC : HDC;  
begin  
    DC := GetDC(Handle);  
    hNewPen := CreatePen(PS_SOLID, 2, RGB(255, 0, 0));  
    hOldPen := SelectObject(DC, hNewPen);  
    LineTo(DC, 100, 100);  
    SelectObject(DC, hOldPen);  
    DeleteObject(hNewPen);  
    ReleaseDC(Handle, DC);
```

```
end;
```

这段代码首先获取窗口的“设备上下文句柄”（HDC）。

然后调用 API 函数 `CreatePen()` 创建一个宽度为 2 像素、颜色为红色（RGB (255, 0, 0)）的笔对象。

接着，调用 API 函数 `SelectObject()` 将所创建的笔对象选择为当前对象。需要注意的是，此时必须将 `SelectObject()` 函数所返回的原先的 GDI 对象保存起来，在使用完创建的新的 GDI 对象后，要将其还原回去，否则就会发生 GDI 资源泄漏。

再接着，调用 API 函数 `LineTo()` 画出一条直线。

完成任务，然后就是收尾工作。首先选择还原 GDI 对象，并调用 API 函数 `DeleteObject()` 删除所创建的笔对象。最后不要忘记调用 API 函数 `ReleaseDC` 以释放窗口的 HDC。

经过这一系列步骤，终于在窗口上画出了一条宽度为 2 像素的红色直线。并且，此过程中不允许有任何的疏漏，因为稍有不慎，便会导致 GDI 资源泄漏。而我们知道，Windows 的窗口经常需要被重新绘制（如被其他窗口挡住又重新出现时），GDI 资源泄漏的速度将是非常快的。

如果将以上这段代码写在某个 Form 的 `OnPaint` 事件中，并且删除 `DeleteObject()` 那行代码（假设漏写了这行），然后运行程序，拖着 Form 在桌面上晃几下，不用多久，Windows 的 GDI 资源就会消耗殆尽，这在 Windows 95/98 系统中表现得尤为明显。在 Windows 2000 中可以如此。

不妨试一下，在 Windows 2000 中打开“任务管理器”窗口，并选择显示“GDI 对象”这一列。随着鼠标的晃动，该程序所使用的 GDI 对象数飞快上升（初始为 31），很快就升到如图 4.6 所示的情况。

映像名称	PID	CPU	CPU 时间	内存使用	句柄数	线程数	GDI 对象
System Idl...	0	99	2:07:08	16 K	0	1	0
System	8	00	0:00:08	280 K	236	34	0
WinMgmt.exe	116	00	0:00:06	156 K	99	3	4
SMSS.EXE	140	00	0:00:00	624 K	38	6	0
CSRSS.EXE	168	00	0:00:28	2,748 K	359	9	0
WINLOGON.EXE	188	00	0:00:01	3,120 K	364	16	24
WINWORD.EXE	200	00	0:15:16	20,732 K	483	7	262
explorer.exe	212	01	0:00:21	3,684 K	479	16	317
SERVICES.EXE	216	00	0:00:01	6,516 K	606	39	4
LSASS.EXE	228	00	0:00:00	1,020 K	297	15	4
Project1.exe	332	00	0:00:03	2,864 K	68	1	394
svchost.exe	408	00	0:00:00	5,392 K	384	9	4
editplus.exe	436	00	0:00:06	5,904 K	105	3	25
agentsvr.exe	444	00	0:00:01	244 K	126	5	27
spoolsv.exe	452	00	0:00:00	4,236 K	153	14	4
svchost.exe	480	00	0:00:00	7,292 K	181	6	4
svchost.exe	498	00	0:00:01	9,972 K	469	29	8
regsvc.exe	528	00	0:00:00	964 K	30	2	0
mstask.exe	572	00	0:00:00	4,836 K	132	7	20
mspmbspv.exe	592	00	0:00:00	1,904 K	49	2	4
Navapw32.exe	668	00	0:00:00	9,556 K	302	11	67
RM RYR RYR	678	00	0:00:00	4,580 K	61	3	18

图 4.6 GDI 资源迅速泄漏



可见，使用最原始的 API 来写图形界面，既低效，又不安全。而 VCL 将 Windows GDI 的 Pen 对象抽象为 TPen 类，使得在窗口上作图非常方便并且安全。

来看一下 TPen 类的声明：

```
TPen = class(TGraphicsObject)
private
    FMode: TPenMode;
    procedure GetData(var PenData: TPenData);
    procedure SetData(const PenData: TPenData);
protected
    function GetColor: TColor;
    procedure SetColor(Value: TColor);
    function GetHandle: HPen;
    procedure SetHandle(Value: HPen);
    procedure SetMode(Value: TPenMode);
    function GetStyle: TPenStyle;
    procedure SetStyle(Value: TPenStyle);
    function GetWidth: Integer;
    procedure SetWidth(Value: Integer);
public
    constructor Create;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    property Handle: HPen read GetHandle write SetHandle;
published
    property Color: TColor read GetColor write SetColor default clBlack;
    property Mode: TPenMode read FMode write SetMode default pmCopy;
    property Style: TPenStyle read GetStyle write SetStyle default
psSolid;
    property Width: Integer read GetWidth write SetWidth default 1;
end;
```

TPen 基本上将 API 函数 CreatePen() 的 3 个参数都作为 TPen 的属性，使用 TPen 只需创建 TPen 的实例并且设置这些属性即可。同样画一条宽度为 2 像素的红色直线，使用 TPen 的代码就会是这样的：

```
Canvas.Pen.Color := clRed;
Canvas.Pen.Width := 2;
Canvas.PenPos := Point(0, 0);
Canvas.LineTo(100, 100);
```

这里的代码使用了 `TCustomForm` 的 `Canvas` 属性的 `Pen` 子对象。关于 `Canvas` 将在 4.7.3 节中详述，此处可以将它当作一个创建好了 `TPen` 实例对象的一个对象。

这些代码显然易懂得多，而且很安全，不需要担心资源泄漏的情况。

现在已经可以明显体会到 `TPen` 的优越之处。不过，此处的重点并非要知道 `TPen` 有多好用，而是要知道 `TPen` 是如何封装 Windows GDI 中的 `Pen` 对象的。

当调用

```
Pen := TPen.Create()
```

后，就创建了一个 `TPen` 的实例。那么 `TPen` 的构造函数做了什么呢？

```
constructor TPen.Create;  
begin  
  FResource := PenManager.AllocResource(DefPenData);  
  FMode := pmCopy;  
end;
```

在这里，可以发现 `PenManager` 的存在。为了不干扰视线，可以把它当作一个 GDI 资源的管理器。其实，它的类型正是 `TResourceManager` 类。

在 VCL 的 `Graphics.pas` 单元中，定义了同样的 3 个资源管理器：

```
var  
  FontManager: TResourceManager;  
  PenManager: TResourceManager;  
  BrushManager: TResourceManager;
```

`PenManager` 正是其中一个管理 `Pen` 资源的管理器。它内部维护了一个已经分配了所有类型的 `Pen` 的列表，当如同这样：

```
FResource := PenManager.AllocResource(DefPenData);
```

当调用它的 `AllocResource()` 方法时，它会在其内部列表中寻找是否已经分配了同类型的 `Pen`，如果有，则增加该类型的 `Pen` 的引用计数；如果没有，则分配一个新的类型的 `Pen`：

```
function TResourceManager.AllocResource(const ResData): PResource;  
var  
  ResHash: Word;  
begin  
  ResHash := GetHashCode(ResData, ResDataSize);  
  Lock;  
  try  
    Result := ResList;
```



```
while (Result <> nil) and ((Result^.HashCode <> ResHash) or
  not CompareMem(@Result^.Data, @ResData, ResDataSize)) do
  Result := Result^.Next;
if Result = nil then
begin // 没找到, 则分配
  GetMem(Result, ResDataSize + ResInfoSize);
  with Result^ do
  begin
    Next := ResList;
    RefCount := 0;
    Handle := TResData(ResData).Handle;
    HashCode := ResHash;
    Move(ResData, Data, ResDataSize);
  end;
  ResList := Result;
end;
Inc(Result^.RefCount); // 增加引用计数
finally
  Unlock;
end;
end;
```

TPen 的构造函数其实就是为其实例申请一块内存以存放该 Pen 的一些属性。该块内存为 TPenData 记录类型:

```
TPenData = record
  Handle: HPen;
  Color: TColor;
  Width: Integer;
  Style: TPenStyle;
end;
```

该记录对应于 API 函数 CreatePen() 要求定义的 Pen 的属性, 其中 Handle 为 Windows 中该 Pen 的句柄。

```
FResource := PenManager.AllocResource(DefPenData);
```

中的 DefPenData 参数, 其类型就是该记录类型的, 该变量定义了 Pen 的默认属性:

```
const
  DefPenData: TPenData = (
    Handle: 0;
```

```
Color: clBlack;  
Width: 1;  
Style: psSolid);
```

因此，TPen 的构造函数完成了 Pen 的资源分配，不过该 Pen 的句柄为 0，这是因为并没有真正向 Windows 申请创建一个 GDI 的 Pen 对象（毕竟一旦申请，就要耗费一个 GDI 对象，而 Windows 中，GDI 资源是很宝贵的）。

当真正需要使用 Pen 时，就需要将向 Windows 申请而获得的 Pen 对象的句柄赋给 VCL 的 Pen 对象。这就是通过其 Handle 属性进行的。从 TPen 的声明

```
property Handle: HPen read GetHandle write SetHandle;
```

中可以看到，当设置该属性时会调用 SetHandle()方法；当读取该属性时，会通过调用 GetHandle()方法来获得。

SetHandle()方法将句柄传递给 TPen 实例的那个 TPenData 记录：

```
procedure TPen.SetHandle(Value: HPen);  
var  
    PenData: TPenData;  
begin  
    PenData := DefPenData;  
    PenData.Handle := Value;  
    SetData(PenData);  
end;
```

而在 GetHandle()方法中，将判断其句柄是否为 0。如果为 0，则说明还没有真正向 Windows 申请创建过 Pen 对象，此时会真正地调用 API 函数 CreatePenIndirect()来创建（该函数与 CreatePen()差不多，区别只在于通过一个结构参数来指定该 Pen 的属性）一个 GDI 的 Pen 对象，并返回其句柄；如果不为 0，则直接返回该句柄：

```
function TPen.GetHandle: HPen;  
const  
    PenStyles: array[TPenStyle] of Word =  
        (PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT, PS_NULL,  
         PS_INSIDEFRAME);  
var  
    LogPen: TLogPen;  
begin  
    with FResource^ do  
    begin  
        if Handle = 0 then
```



```
begin
  PenManager.Lock;
  with LogPen do
  try
    if Handle = 0 then
    begin
      lopnStyle := PenStyles[Pen.Style];
      lopnWidth.X := Pen.Width;
      lopnColor := ColorToRGB(Pen.Color);
      Handle := CreatePenIndirect(LogPen); // 创建一个 GDI 的 Pen 对象
    end;
  finally
    PenManager.Unlock;
  end;
end;
Result := Handle;
end;
end;
```

TPen 的其他属性（如 Color、Width 等）都是通过更改 TPen 内部的 TPenData 记录类型的数据来实现的。TPen 的对象实例真正起作用作为 TCanvas 类的对象的子对象来发挥的，这些在 4.7.3 节讲述 TCanvas 类时会详细展开。

## 4.7.2 TBrush

VCL 用 TPen 封装了 Windows GDI 的 Pen 对象，而另一个主角 Brush 则也是一样，VCL 用 TBrush 封装了 Windows GDI 的 Brush 对象。

Pen 对象用于在窗口上绘制线条，而 Brush 对象则用于填充区域。

同样，先来看一下使用 GDI 的 Brush 对象是如何在窗口上绘图的。

Windows 的 GDI API 提供了一个 CreateBrushIndirect() 函数用来创建 Brush 对象。CreateBrushIndirect() 函数的原型如下：

```
HBRUSH CreateBrushIndirect (
  CONST LOGBRUSH *lp1b
);
```

其中的 LOGBRUSH 结构类型的参数指定了刷子的一些信息：

```
typedef struct tagLOGBRUSH {
  UINT    lbStyle;
  COLORREF lbColor;
```

```
LONG    lbHatch;  
} LOGBRUSH, *PLOGBRUSH;
```

在 Delphi 的 Graphics.pas 中，有该类型定义的 Pascal 语言版本：

```
tagLOGBRUSH = packed record  
  lbStyle: UINT;  
  lbColor: COLORREF;  
  lbHatch: Longint;  
end;
```

4

例如，需要将窗口的 (0, 0, 100, 100) 的正方形区域填充成红色，则使用 GDI 的代码可能是这样的：

```
var  
  lb : LOGBRUSH;  
  hNewBrush : HBRUSH;  
  hWndDC : HDC;  
  R : TRect;  
begin  
  // 设置刷子参数  
  lb.lbStyle := BS_SOLID;  
  lb.lbColor := clRed;  
  lb.lbHatch := HS_VERTICAL;  
  // 创建刷子对象  
  hNewBrush := CreateBrushIndirect(lb);  
  // 取得窗口的设备上下文句柄 (HDC)  
  HWndDC := GetDC(Handle);  
  R := Rect(0, 0, 100, 100);  
  // 用刷子填充对象  
  FillRect(hWndDC, R, hNewBrush);  
  // 删除所创建的刷子对象并释放 HDC  
  DeleteObject(hNewBrush);  
  ReleaseDC(Handle, hWndDC);  
end;
```

VCL 的 TBrush 类则对 GDI 的 Brush 进行了封装。TBrush 的声明如下：

```
TBrush = class(TGraphicsObject)  
private  
  procedure GetData(var BrushData: TBrushData);
```



```
    procedure SetData(const BrushData: TBrushData);
protected
    function GetBitmap: TBitmap;
    procedure SetBitmap(Value: TBitmap);
    function GetColor: TColor;
    procedure SetColor(Value: TColor);
    function GetHandle: HBrush;
    procedure SetHandle(Value: HBrush);
    function GetStyle: TBrushStyle;
    procedure SetStyle(Value: TBrushStyle);
public
    constructor Create;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    property Bitmap: TBitmap read GetBitmap write SetBitmap;
    property Handle: HBrush read GetHandle write SetHandle;
published
    property Color: TColor read GetColor write SetColor default clWhite;
    property Style: TBrushStyle read GetStyle write SetStyle
        default bsSolid;
end;
```

不难发现 **TBrush** 和 **TPen** 非常相似，同样将 GDI 的 Brush 对象的风格抽象成属性，并且构造函数与析构函数所做的工作也与 **TPen** 的差不多。只不过，这次 GDI 资源的管理器不是 **PenManager**，而改成了 **BrushManager**，但 **BrushManager** 与 **PenManager** 其实都是 **TResourceManager** 类的一个实例。

其实，不仅仅是 **TBrush** 与 **TPen** 之间，基本 GDI 对象在 VCL 中，其资源管理策略都是类似的，因此它们的构造函数也就会如此雷同。如 **TBrush**：

```
constructor TBrush.Create;
begin
    FResource := BrushManager.AllocResource(DefBrushData);
end;
```

它同样是调用了 **TResourceManager** 类的 **AllocResource()** 方法来分配一个内存空间以存放一个表示“刷子”默认属性的数据结构。关于 **AllocResource()**，在讲述 **TPen** 时已经介绍了，此处不再重复。

除了资源管理的实现上，在其他方面，包括抽象的方法，**TBrush** 与 **TPen** 也同样类似。例如只有在 **GetHandle()** 方法中才调用 **CreateBrushIndirect()** 去真正创建一个 GDI 的 Brush 对象：

```
function TBrush.GetHandle: HBrush;
var
  LogBrush: TLogBrush;
begin
  with FResource^ do
  begin
    if Handle = 0 then
    begin
      BrushManager.Lock;
      try
        if Handle = 0 then
        begin
          with LogBrush do
          begin
            if Brush.Bitmap <> nil then
            begin
              lbStyle := BS_PATTERN;
              Brush.Bitmap.HandleType := bmDDB;
              lbHatch := Brush.Bitmap.Handle;
            end else
            begin
              lbHatch := 0;
              case Brush.Style of
                bsSolid: lbStyle := BS_SOLID;
                bsClear: lbStyle := BS_HOLLOW;
              else
                lbStyle := BS_HATCHED;
                lbHatch := Ord(Brush.Style) - Ord(bsHorizontal);
              end;
            end;
            lbColor := ColorToRGB(Brush.Color);
            Handle := CreateBrushIndirect(LogBrush);
          end;
        finally
          BrushManager.Unlock;
        end;
      end;
      Result := Handle;
    end;
  end;
end;
```



此处对 `CreateBrushIndirect()` 的调用与此前直接使用 GDI API 的例子相比，惟一的区别在于参数的第 3 个域的赋值。此前的例子中，我们给 Brush 的信息的赋值是这样的：

```
lb.lbStyle := BS_SOLID;
lb.lbColor := clRed;
lb.lbHatch := HS_VERTICAL;
```

第 3 个参数给的是 Brush 的“开口方向”，而 VCL 的 `TBrush` 中，对 API 封装需要考虑各种情况，而且 `TBrush` 允许将“刷子”和一个位图联系起来，因此该参数的决定也比较复杂。

```
with LogBrush do
begin
  // 如果“刷子”以位图方式创建，则将位图句柄作为该参数的值
  if Brush.Bitmap <> nil then
  begin
    lbStyle := BS_PATTERN;
    Brush.Bitmap.HandleType := bmDDB;
    lbHatch := Brush.Bitmap.Handle;
  end else
  // 如果“刷子”并非以位图方式创建，则……
  begin
    lbHatch := 0;
    case Brush.Style of
      bsSolid: lbStyle := BS_SOLID; // “实心刷子”
      bsClear: lbStyle := BS_HOLLOW; // “透明”
    else
      lbStyle := BS_HATCHED;
      lbHatch := Ord(Brush.Style) - Ord(bsHorizontal);
    end;
  end;
  lbColor := ColorToRGB(Brush.Color);
end;
```

`TBrush` 与 `TPen` 同样是为了配合 `TCanvas` 的，其作用会在 4.7.3 节 `TCanvas` 中展开。GDI 的基本对象当然不止 `Pen` 与 `Brush`，还包括字体、位图等。不过，它们在 VCL 中的抽象方法与 `TPen` 和 `TBrush` 大同小异，在此不再一一介绍。如果对这方面内容感兴趣，可以参考 `graphics.pas` 单元中的代码。

### 4.7.3 TCanvas

VCL 除了封装 GDI 的对象（如 Pen 和 Brush）以外，也同时封装了 GDI 的绘图设备。VCL 将 GDI 的设备抽象成一个画布（Canvas），使得我们可以在其上任意作画。TCanvas 类就是这个画布的抽象。

先来看一下 TCanvas 类的声明：

```
TCanvas = class(TPersistent)
private
    FHandle: HDC;
    State: TCanvasState;
    FFont: TFont;
    FPen: TPen;
    FBrush: TBrush;
    FPenPos: TPoint;
    FCopyMode: TCopyMode;
    FOnChange: TNotifyEvent;
    FOnChanging: TNotifyEvent;
    FLock: TRTLCriticalSection;
    FLockCount: Integer;
    FTextFlags: Longint;
    procedure CreateBrush;
    procedure CreateFont;
    procedure CreatePen;
    procedure BrushChanged(ABrush: TObject);
    procedure DeselectHandles;
    function GetCanvasOrientation: TCanvasOrientation;
    function GetClipRect: TRect;
    function GetHandle: HDC;
    function GetPenPos: TPoint;
    function GetPixel(X, Y: Integer): TColor;
    procedure FontChanged(AFont: TObject);
    procedure PenChanged(APen: TObject);
    procedure SetBrush(Value: TBrush);
    procedure SetFont(Value: TFont);
    procedure SetHandle(Value: HDC);
    procedure SetPen(Value: TPen);
    procedure SetPenPos(Value: TPoint);
    procedure SetPixel(X, Y: Integer; Value: TColor);
protected
    procedure Changed; virtual;
    procedure Changing; virtual;
```



```
procedure CreateHandle; virtual;
procedure RequiredState(ReqState: TCanvasState);
public
  constructor Create;
  destructor Destroy; override;
  procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
  procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap;
    const Source: TRect; Color: TColor);
  procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
  procedure CopyRect(const Dest: TRect; Canvas: TCanvas;
    const Source: TRect);
  procedure Draw(X, Y: Integer; Graphic: TGraphic);
  procedure DrawFocusRect(const Rect: TRect);
  procedure Ellipse(X1, Y1, X2, Y2: Integer); overload;
  procedure Ellipse(const Rect: TRect); overload;
  procedure FillRect(const Rect: TRect);
  procedure FloodFill(X, Y: Integer; Color: TColor;
    FillStyle: TFillStyle);
  procedure FrameRect(const Rect: TRect);
  function HandleAllocated: Boolean;
  procedure LineTo(X, Y: Integer);
  procedure Lock;
  procedure MoveTo(X, Y: Integer);
  procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
  procedure Polygon(const Points: array of TPoint);
  procedure Polyline(const Points: array of TPoint);
  procedure PolyBezier(const Points: array of TPoint);
  procedure PolyBezierTo(const Points: array of TPoint);
  procedure Rectangle(X1, Y1, X2, Y2: Integer); overload;
  procedure Rectangle(const Rect: TRect); overload;
  procedure Refresh;
  procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);
  procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);
  function TextExtent(const Text: string): TSize;
  function TextHeight(const Text: string): Integer;
  procedure TextOut(X, Y: Integer; const Text: string);
  procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);
  function TextWidth(const Text: string): Integer;
  function TryLock: Boolean;
  procedure Unlock;
  property ClipRect: TRect read GetClipRect;
  property Handle: HDC read GetHandle write SetHandle;
  property LockCount: Integer read FLockCount;
  property CanvasOrientation: TCanvasOrientation read
```

```
    GetCanvasOrientation;  
    property PenPos: TPoint read GetPenPos write SetPenPos;  
    property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;  
    property TextFlags: Longint read FTextFlags write FTextFlags;  
    property OnChange: TNotifyEvent read FOnChange write FOnChange;  
    property OnChanging: TNotifyEvent read FOnChanging write FOnChanging;  
published  
    property Brush: TBrush read FBrush write SetBrush;  
    property CopyMode: TCopyMode read FCopyMode write FCopyMode  
        default cmSrcCopy;  
    property Font: TFont read FFont write SetFont;  
    property Pen: TPen read FPen write SetPen;  
end;
```

在上述的 **TPen** 和 **Tbrush** 介绍中提到过的使用 **GDI API** 直接绘图的代码示例中，都有类似这样的一行代码：

```
DC := GetDC(Handle);
```

这行代码从一个窗口句柄获取该窗口的“设备上下文句柄”（**HDC**），以便使用 **GDI** 函数在该窗口上进行绘图。

**TCanvas** 作为一个“画布”的抽象，必定需要一个“设备上下文句柄”。**TCanvas** 中 **private** 的 **FHandle** 数据成员就是保存这个“设备上下文句柄”的，并且通过 **public** 的 **Handle** 属性的 **GetHandle()**和 **SetHandle()**方法来对其进行访问。

**TCanvas** 内部还拥有各种 **GDI** 基础对象的抽象，如 **TPen**、**TBrush**、**TFont** 这样的子对象，并且在 **TCanvas** 的构造函数中便创建它们的实例：

```
constructor TCanvas.Create;  
begin  
    inherited Create;  
    InitializeCriticalSection(FLock);  
    FFont := TFont.Create;  
    FFont.OnChange := FontChanged;  
    FFont.OwnerCriticalSection := @FLock;  
    FPen := TPen.Create;  
    FPen.OnChange := PenChanged;  
    FPen.OwnerCriticalSection := @FLock;  
    FBrush := TBrush.Create;  
    FBrush.OnChange := BrushChanged;  
    FBrush.OwnerCriticalSection := @FLock;  
    FCopyMode := cmSrcCopy;  
    State := [];  
    CanvasList.Add(Self);  
end;
```



另外，TCanvas 提供了对应于 GDI 绘图 API 的各种方法，包括在“画布”上绘制各种形状的方法，如 LineTo()（画直线）、Rectangle()（画矩形）、Ellipse()（画圆/椭圆）以及直接贴位图的 Draw()等。

在此以画直线为例，跟踪一下 TCanvas 的执行路线，看它是在何时以何种方式调用相应的 GDI API 来完成的。

首先，TCanvas 在构造函数中创建了 TPen 子对象的实例 FPen：

```
FPen := TPen.Create;
```

然后，TCanvas 的客户需要将一个窗口的“设备上下文句柄”（HDC）设置给 Canvas 实例 Handle 属性。TCanvas 自己是无法提供这个 Handle 属性的值的，虽然 TCanvas 声明了一个虚方法 CreateHandle()，但该方法在 TCanvas 中的实现是空的。不过，一般在使用 TCanvas 时，都是通过某个组件（如 TForm）的 Canvas 属性来使用的（这类组件的 Canvas 属性其实是一个 TCanvas 的实例对象），因此其 Handle 属性并不需要我们来设置，而是由组件来完成的。至于空的虚方法 CreateHandle()的作用，以及在组件中使用 Canvas 属性，这些会在 4.8 节再提及。

在设置 Handle 属性时，会调用 TCanvas.SetHandle()方法：

```
procedure TCanvas.SetHandle(Value: HDC);
begin
  if FHandle <> Value then
  begin
    if FHandle <> 0 then
    begin
      DeselectHandles;
      FPenPos := GetPenPos;
      FHandle := 0;
      Exclude(State, csHandleValid);
    end;
    if Value <> 0 then
    begin
      Include(State, csHandleValid);
      FHandle := Value;
      SetPenPos(FPenPos);
    end;
  end;
end;
```

在 SetHandle()方法中，除了设置 FHandle 的值外，还会调用 SetPenPos()方法设置“画笔”的起始坐标点。

接着，客户程序可以使用 TCanvas 的 LineTo()方法来使用画笔进行画线：

```

procedure TCanvas.LineTo(X, Y: Integer);
begin
    Changing;
    RequiredState([csHandleValid, csPenValid, csBrushValid]);
    Windows.LineTo(FHandle, X, Y);
    Changed;
end;

```

在 `LineTo()` 方法中，首先调用 `RequiredState()` 方法，在 `RequiredState()` 方法中，会再调用 `CreatePen()` 方法来选中当前的画笔对象：

```

procedure TCanvas.CreatePen;
const
    PenModes: array[TPenMode] of Word =
        (R2_BLACK, R2_WHITE, R2_NOP, R2_NOT, R2_COPYPEN, R2_NOTCOPYPEN,
         R2_MERGEPEENNOT, R2_MASKPEENNOT, R2_MERGENOTPEN, R2_MASKNOTPEN,
         R2_MERGEPEEN, R2_NOTMERGEPEEN, R2_MASKPEN, R2_NOTMASKPEN, R2_XORPEN,
         R2_NOTXORPEN);
begin
    SelectObject(FHandle, Pen.GetHandle);
    SetROP2(FHandle, PenModes[Pen.Mode]);
end;

```

在 `CreatePen()` 方法中，执行了 API 函数 `SelectObject()`，将 `Pen` 对象选为当前画笔对象。最后，`LineTo()` 方法中调用 API 函数 `LineTo()` 来画出直线：

```
Windows.LineTo(FHandle, X, Y);
```

由于在 `Graphics.pas` 单元中发生了“`LineTo`”这样的名称冲突，因此，在真正调用 Windows API 的 `LineTo()` 函数时，在其前指明了命名空间（单元名）“`Windows.`”。

好了，直线画出来了。除了画直线，其他图形的操作原理类似，不再赘述。

## 4.8 TGraphicControl/TcustomControl 与画布（Canvas）

VCL 中，`TControl` 之下的组件分两条路各行其道。一条为图形组件，这类组件并非窗口，职责只在于显示图形、图像，其基类是 `TGraphicControl`；另一条为窗口组件，这类组件本身是一个 Windows 窗口（有窗口句柄），其基类是 `TWinControl`。

`TGraphicControl` 作为显示图形、图像的组件分支，从其开始就提供了一个 `TCanvas` 类型的 `Canvas` 属性，以便在组件上绘制图形、显示图像。



对于窗口组件的分支，TWinControl 并没有提供 Canvas 属性，而在其派生类 TCustomControl 才开始提供 Canvas 属性。如图 4.7 所示。

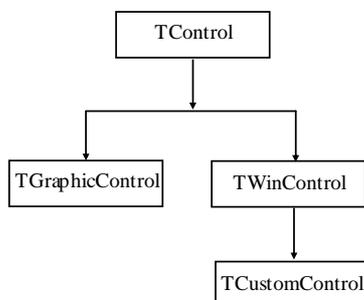


图 4.7 控件类分支

TGraphicControl 与 TCustomControl 的实现都在 Controls.pas 单元中，它们的声明看上去也是如此相似：

```
TGraphicControl = class(TControl)
private
    FCanvas: TCanvas;
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
protected
    procedure Paint; virtual;
    property Canvas: TCanvas read FCanvas;
public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
end;

TCustomControl = class(TWinControl)
private
    FCanvas: TCanvas;
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
protected
    procedure Paint; virtual;
    procedure PaintWindow(DC: HDC); override;
    property Canvas: TCanvas read FCanvas;
public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
end;
```

它们提供了 Canvas 属性，只不过此时 Canvas 属性被隐藏在 protected 节中，它们的派生类可以选择性地将其 publish。

由于 TGraphicControl 与 TCustomControl 在有关 Canvas 熟悉的实现上也非常相似，在此只以 TGraphicControl 的实现来讲解“画布”属性。

由 TGraphicControl 的声明中的

```
property Canvas: TCanvas read FCanvas;
```

可知 Canvas 是一个只读属性，其载体是 private 的成员对象 FCanvas。FCanvas 在 TGraphicControl 的构造函数中被创建：

```
constructor TGraphicControl.Create(AOwner: TComponent);  
begin  
    inherited Create(AOwner);  
    FCanvas := TControlCanvas.Create;  
    TControlCanvas(FCanvas).Control := Self;  
end;
```

在此需要注意的是，FCanvas 在声明时，是被声明为 TCanvas 类型的，而在创建时，却创建了 TControlCanvas 的实例。其实，TControlCanvas 是 TCanvas 的派生类，它提供了一些额外的属性和事件来辅助在 Control（控件）上提供“画布”属性。

这里暂停一下，先来看一下 TControlCanvas：

```
TControlCanvas = class(TCanvas)  
private  
    FControl: TControl;  
    FDeviceContext: HDC;  
    FWindowHandle: HWND;  
    procedure SetControl(AControl: TControl);  
protected  
    procedure CreateHandle; override;  
public  
    destructor Destroy; override;  
    procedure FreeHandle;  
    procedure UpdateTextFlags;  
    property Control: TControl read FControl write SetControl;  
end;
```

TControlCanvas 将 Canvas 绑定到一个 TControl 实例上，其内部的 FControl 指针即指向 Canvas 所属的 TControl 实例。

记得 4.7 节中讲过，TCanvas 提供了一个空的虚方法 CreateHandle()。这个虚方法在



TControlCanvas 中被覆盖重新实现:

```
procedure TControlCanvas.CreateHandle;  
begin  
  if FControl = nil then inherited CreateHandle else  
  begin  
    if FDeviceContext = 0 then  
    begin  
      with CanvasList.LockList do  
      try  
        if Count >= CanvasListCacheSize then FreeDeviceContext;  
        FDeviceContext := FControl.GetDeviceContext(FWindowHandle);  
        Add(Self);  
      finally  
        CanvasList.UnlockList;  
      end;  
    end;  
    Handle := FDeviceContext;  
    UpdateTextFlags;  
  end;  
end;
```

在 CreateHandle()方法中, 如果 FControl 是 TWinControl 或其派生类的实例, 即控件本身是窗口, 则取得该窗口的设备上下文句柄赋给 Handle 属性; 如果 FControl 非 TWinControl 或其派生类的实例, 即控件本身并非窗口, 则将其父窗口的设备上下文句柄赋给 Handle。这些都是通过 TControl 声明的虚函数 GetDeviceContext()实现的, 因为 TWinControl 覆盖重新实现了 GetDeviceContext()。

说完 TControlCanvas, 下面继续刚才的话题。TGraphicControl 的构造函数中创建了 TControlCanvas 实例并赋给 FCanvas。构造函数的最后一行代码

```
TControlCanvas(FCanvas).Control := Self;
```

将 Canvas 属性绑定到了控件本身。

然后, TGraphicControl 定义了一个处理 WM\_PAINT Windows 消息的消息处理函数:

```
procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
```

在 WMPaint()方法中, 根据接受到的消息的参数所给出的窗口的设备上下文句柄, 给 Canvas 属性的 Handle 重新赋值, 并且调用虚方法 Paint():

```
procedure TGraphicControl.WMPaint(var Message: TWMPaint);  
begin  
  if Message.DC <> 0 then
```

```
begin
  Canvas.Lock;
  try
    Canvas.Handle := Message.DC;
    try
      Paint;
    finally
      Canvas.Handle := 0;
    end;
  finally
    Canvas.Unlock;
  end;
end;
end;
```

虚方法 `Paint()` 可以被 `TGraphicControl` 的派生类所覆盖，重新定义并实现绘制图形、图像的方法，并且 `TGraphicControl` 的派生的实例总是可以放心使用其 `Canvas` 属性，而不必自行获得窗口的设备上下文句柄。而虚方法 `Paint()` 在 `TGraphicControl` 中的实现也只是一个空方法而已。

4.9 节中将讲述 `TGraphicControl`/`TCustomControl` 的虚方法 `Paint()` 是如何被它们的派生类所使用来进行窗口重绘的。

## 4.9 TCustomPanel 与窗口重绘

`TCustomPanel` 派生自 `TCustomControl`，是所有 `Panel` 类组件的基类。`TCustomPanel` 与 4.8 节中所述的 `TGraphicControl` 非常类似，只是 `TCustomControl` 派生自 `TWinControl`，所以它的实例是一个窗口。

`TCustomControl` 与 `TGraphicControl` 一样，拥有一个空的虚方法 `Paint()`，以便让派生类决定如何重绘窗口。

现在就来看一下 `TCustomPanel`。它从 `TCustomControl` 派生，并且覆盖重新实现了 `Paint()` 方法。在此，我们不关心 `TCustomPanel` 所实现的其他特性，而只关注其实现的 `Paint()` 方法。`TCustomPanel` 实现的 `Paint()` 方法负责将组件窗口绘制出一个 `Panel` 效果（边框、背景和标题）。先来看一下 `Paint()` 方法：

```
procedure TCustomPanel.Paint;
const
  Alignments: array[TAlignment] of Longint = (
    DT_LEFT,
    DT_RIGHT,
```



```
DT_CENTER
);
var
  Rect: TRect;
  TopColor, BottomColor: TColor;
  FontHeight: Integer;
  Flags: Longint;

procedure AdjustColors(Bevel: TPanelBevel);
begin
  TopColor := clBtnHighlight;
  if Bevel = bvLowered then TopColor := clBtnShadow;
  BottomColor := clBtnShadow;
  if Bevel = bvLowered then BottomColor := clBtnHighlight;
end;

begin
  Rect := GetClientRect;
  // 画边框
  if BevelOuter <> bvNone then
  begin
    AdjustColors(BevelOuter);
    Frame3D(Canvas, Rect, TopColor, BottomColor, BevelWidth);
  end;
  Frame3D(Canvas, Rect, Color, Color, BorderWidth);
  if BevelInner <> bvNone then
  begin
    AdjustColors(BevelInner);
    Frame3D(Canvas, Rect, TopColor, BottomColor, BevelWidth);
  end;
  with Canvas do
  begin
    // 画背景
    Brush.Color := Color;
    FillRect(Rect);
    Brush.Style := bsClear;
    // 写标题
    Font := Self.Font;
    FontHeight := TextHeight('W');
    with Rect do
    begin
      Top := ((Bottom + Top) - FontHeight) div 2;
      Bottom := Top + FontHeight;
```

```
end;  
Flags := DT_EXPANDTABS or DT_VCENTER or Alignments[FAlignment];  
Flags := DrawTextBiDiModeFlags(Flags);  
DrawText(Handle, PChar(Caption), -1, Rect, Flags);  
end;  
end;
```

Paint()方法含有一个内嵌函数 AdjustColors(), 其作用是确定边框的上下线条颜色 (一条边框由两个像素宽度的直线构成, 形成立体效果)。

TCustomPanel 使用其基类 (TCustomControl) 提供的 Canvas 属性, 覆盖其基类定义的虚方法 Paint(), 完成了窗口重绘过程。

在自己编写组件时, 如果需要在组件表面绘制图形、图像的话, 就可以如同 TCustomPanel 一样, 覆盖重新实现 Paint()方法。同时, 使用基类提供的 Canvas 属性, 对于绘图过程来说, 也是非常简单的。

由此 VCL 完全封装了 Windows 的 GDI 功能, 并提供了一个简单、易用的接口。

## 4.10 TCustomForm 与模态窗口

TCustomForm 是 Windows 窗口 (一般窗口与对话框) 的基类。它有两个显示窗口的方法: Show()和 ShowModal()分别用来显示非模态与模态的窗口。不过, 它对于模态窗口的实现并没有利用 Windows 系统提供的 DialogBox()之类的 API, 而是 VCL 自己实现的。原因可能是无法将 DialogBox()与 VCL 的 Form 机制很好地结合。

这一节来研究一下 Show()和 ShowModal()的具体实现。

先是 Show():

```
procedure TCustomForm.Show;  
begin  
  Visible := True;  
  BringToFront;  
end;
```

Show()的代码非常简单, 而且易懂, 它的行为与其名称一样的单纯。

而 ShowModal()要做的事情则多得多:

```
function TCustomForm.ShowModal: Integer;  
var  
  ..... // 省略变量声明  
begin  
  ..... // 省略部分代码
```



```
try
  Show; // 调用 Show() 方法显示窗口
  try
    SendMessage(Handle, CM_ACTIVATE, 0, 0);
    ModalResult := 0;

    // 接管线程主消息循环, 使窗口“模态”化
    repeat
      Application.HandleMessage;
      if Application.FTerminate then
        ModalResult := mrCancel
      else
        if ModalResult <> 0 then CloseModal;
    until ModalResult <> 0;
    Result := ModalResult;
    SendMessage(Handle, CM_DEACTIVATE, 0, 0);
    if GetActiveWindow <> Handle then ActiveWindow := 0;
  finally
    Hide; // 窗口消失
  end;
finally
  // 省略部分代码
end;
end;
```

可见, VCL 中的模态窗口是通过接管线程主消息循环来实现的, 只是它的退出循环条件是 `ModalResult <> 0` (`ModalResult` 初始值为 0), 那么, `ModalResult` 的值是何时被改变的呢? 有两种方式可以改变这个 `ModalResult` 的值:

一种是程序员在模态窗口中的某个事件代码中显式地改变 `ModalResult` 的值。如:

```
ModalResult := mrOK;
```

另一种是设置该窗口上的某个按钮的 `ModalResult` 的属性值, 当单击该按钮后就改变了窗口的 `ModalResult`。也许有人会奇怪, 按钮属性是如何和窗口的属性联系起来呢? 看一下 `TButton` 的 `Click` 方法就知道了, 该方法会在每个按钮被按下后被执行:

```
procedure TButton.Click;
var
  Form: TCustomForm;
begin
  // 获取按钮父窗口的 TCustomForm 对象
  Form := GetParentForm(Self);
```

```
// 改变 Form 对象的 ModalResult 值
if Form <> nil then Form.ModalResult := ModalResult;
// 调用 TControl.Click(), 即调用 OnClick 事件的用户代码
inherited Click;
end;
```

按钮被按下后，这段程序会首先得到执行，最后的那行在对 `TControl.Click()` 的调用中，才会执行 Delphi 程序员为该按钮定义的 `OnClick` 事件的代码。

## 4.11 小 结

查看经典的源代码对于每个程序员的提高，都或多或少会有所助益，尤其是像 VCL 这样经典的但文档尚未完善的库。

也许读者感觉到了，本章中 VCL 的源码的数量比较多。但是请不要忽略那些在代码中插入的注释，我个人感觉这些注释对于学会如何去看懂 VCL 源码至关重要。读完这一章后，读者对 VCL 库的几个核心类应该有了一个大概的了解，然后以此起步，学会自己研究 VCL 源码的方法，这才是本章最重要的目的。

我认为，VCL 的源代码无论对于我们掌握其实现以便更好地处理问题，还是对于学习面向对象程序的构架，都有莫大的好处。虽然在第 1 章中说过，在 Delphi 中可以忽略你所不想知道的细节，但请不要理会错了。

我的意思是，在实际的开发工作中，应该力求简单性原则，忽略不必要的、繁琐的细节而主攻程序的灵魂——业务逻辑。而在学习的时候，应该力求深度，“知其然而又知其所以然”。而且这时，Delphi 绝对不会阻碍你去探求其真实所在。这正是其他 RAD 工具所不具备的！

## 第 5 章 扩展VCL库

无法扩展的代码是死代码，无法扩展的库是死库。





VCL 的另一优秀之处就在于其具有可扩展性，程序员可以按自己的要求定制自己的组件或仅仅简单扩充现有的组件，就完全地、无缝地融入 VCL 库中。

本章将介绍如何利用自己已经拥有的对 VCL 库的知识，制作符合自己特殊需求的组件来扩展 VCL 库。

## 5.1 组件基础

在展开编写组件的话题之前，首先介绍一些必要的概念。如果已经熟知这些，可跳过本节。

所谓组件，就是 `TComponent` 的可实例化的派生类（即非抽象类）。每个组件包括属性、方法和事件。

### 5.1.1 属性

属性使用关键字 `property` 来声明。不过，属性只是一个抽象的名称，它必须有实际的载体，因此必须为每个属性提供一个相应类型的数据成员以保存属性值。另外，为了使得 Delphi 开发环境中的 `Object Inspector` 能够识别组件的属性并显示出来，需要将属性的声明置于 `published` 段中。因此，一般属性的声明看上去如下：

```
published:
    property AutoSize : Boolean Read FAutoSize Write FAutoSize;
```

这行代码为组件声明了一个 `AutoSize` 的属性，类型为 `Boolean`，其读和写都基于一个数据成员（通常是 `private` 的）`FAutoSize`。也就是说，`AutoSize` 属性的值实际上是保存在 `FAutoSize` 成员中的，`FAutoSize` 也就是 `AutoSize` 的“载体”。

`Read` 关键字和 `Write` 关键字指示了属性的读写途径。可以如上所述直接读写某个“载体”成员，也可以通过某个成员方法进行读写。如：

```
published :
    property AutoSize : Boolean Read GetAutoSize Write SetAutoSize;
```

此时，`GetAutoSize` 和 `SetAutoSize` 分别是两个特定原型的方法/函数。其原型如下：

```
function GetAutoSize() : Boolean;
procedure SetAutoSize(bValue : Boolean);
```

它们可能的实现：

```
function GetAutoSize() : Boolean;
begin
    Result := FAutoSize;
end;

procedure SetAutoSize(bValue : Boolean);
begin
    FAutoSize := bValue;
    ..... // 更改属性值后刷新组件状态的代码
end;
```

Read 和 Write 并非必须同时存在，如果只有 Read 而没有 Write，则该属性为只读。

另外，Object Pascal 的属性支持“覆盖”（Override）特性。所谓“覆盖的属性”也就是不指定属性类型的属性，它们的类型在基类中被定义。覆盖时，可以重新定义组件的可见性（组件属于 protected 或是 public 或是 published 的）或重新定义它们的读写途径或其他参数。

在 VCL 很多组件的基类中有许多 protected 的属性，它们对外是不可见的，而在其派生类中才使之可见。此时派生类中无需为该属性指定类型、读写途径等，而只需要在 published 段中加一个“property 属性名”的简单声明即可。

当派生类声明了与基类中某属性相同名称的属性，并且指定了其类型时，则会发生派生类属性“隐藏”（Hide）基类的属性，而不是“覆盖”了。隐藏的含义则是，基类与派生类各有一份该属性实例，只是在派生类中基类的属性不可见。并且，在多态的情况下，采用早绑定（相对于晚绑定）进行编译，即按照对象的编译期类型而非按照对象的运行期类型进行地址绑定。发生“隐藏”情况时，编译器会给出提示。

### 5.1.2 方法

所有 public 的成员函数都是方法，方法供程序员在运行时（而非设计时）调用。

### 5.1.3 事件

事件则是特殊的属性，因此事件也是由 property 关键字声明。只是其类型必须是一个函数指针，其所依附的数据成员也必须是函数指针。如：

```
private
    FOnClick : TNotifyEvent;
    .....
published:
    property OnClick : TNotifyEvent Read FOnClick Write FOnClick;
    .....
```



此处，OnClick 事件是 TNotifyEvent 类型的。在 Classes 单元中可以看到其定义：

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

可见它是一个函数指针类型。当然，也可以定义自己的事件类型，如：

```
type  
  TMyEvent = procedure (Sender : TObject; ClickTime : TdateTime) of object;
```

其实，事件所依附的函数指针类型的数据成员存放的就是用户在组件事件中编写的代码的入口地址。在组件中，要执行用户为事件所编写的代码，只需要调用这个指针即可。通常形如：

```
if Assigned(FOnClick) then  
  FOnClick(self);
```

### 5.1.4 包

VCL 组件以“包”的形式被 Delphi 的 IDE 所管理。“包”有如下两种含义：

在源代码级别，“包”可以被看作是一个 project (.DPK)。和一般 project 不同的是，包中所含的，都是组件的实现单元。一般情况下，一个单元文件 (.pas) 实现一个或多个组件，而包 (.dpc) 中则可以含有一个或多个这样的单元。组件的安装、卸载，均以包为单位。因此，包可以看作是含有多个组件的 project。

在二进制代码级别，“包”就是一个动态链接库 (.DLL) 文件，只是其扩展名为 .bpl。每个包编译后即产生一个 bpl。当应用程序使用了包中的组件时，如果选择动态链接方式，则可以缩小可执行文件的大小。

对于开发者而言，更多情况下会在源代码级别来理解“包”。

### 5.1.5 组件的安装

最后，讲一下注册组件和安装组件的方法。只需要在组件代码的单元中，添加一个这样的函数即可：

```
procedure Register;  
begin  
  RegisterComponents('MyComponent', [TDemo]);  
end;
```

RegisterComponents()函数的第一个参数是指定将组件注册到组件板的哪一页中；第二

个参数给出类名称，而该类必须是 TComponent 的派生类。

安装的时候，选择 Delphi 的“Component”→“Install Component...”菜单，选择组件的实现的单元文件以及要安装到的“包”，然后即可编译安装。

## 5.2 扩展现有组件

虽然 VCL 已经为用户提供了数百个功能强大的组件，但需求是千变万化的，因此有时需要扩展 VCL 以满足特殊需求。用户可以完全从头开始定制一个自己的组件，这样的方法将在 5.3 节中介绍。更有效率的做法是在 VCL 提供的现有组件的基础上稍作修改，增加一些用户需要的特性，如果这样行得通的话。扩展现有组件可以将满足用户需求的组件建立在已经可以正常工作的组件基础之上，这样便省却了许多工作，既安全又快捷。

扩展现有组件的步骤如下：

- (1) 确定要被扩展的基础组件（类）。
- (2) 明确在原组件基础上所需要增加的功能。
- (3) 将需要增加的功能转换成属性、方法或事件的体现。
- (4) 实现。

下面，用实例来说明。

### 5.2.1 实例一：支持文件拖放操作的 ListBox

首先确定需求。假设现在所编写的软件中有这样一个需求：一个文件名列表，并且允许用户直接从 Windows 资源管理器中将文件拖放到列表中，在列表中显示该文件名。

这个实例的需求很明确，就是要实现一个列表组件，以允许接受文件的拖放操作。也就是一个具有文件拖放功能的 ListBox。因此，很容易明确组件应该从 TListBox 派生，只需为其加上允许文件拖放功能的代码即可，而无需重新编写一个列表组件。

然后，设法将这个需求转化成新的组件的属性、方法或事件。要接受文件拖放，肯定需要增加一个事件——OnDropFiles。看起来这样似乎就足够了，但是否应该给组件的用户以更大的灵活性呢？例如，组件可以处于允许拖放和不允许拖放两种状态，并且可以在运行时切换两种状态。不允许拖放时，新的组件和 TListBox 没什么区别。是否提供这种灵活性全在于组件设计者的一念之间，应该说没什么非常明确的答案，主要取决于设计者的经验及对需求的理解。一般来说，应该尽可能为用户提供灵活性，但必须在不能大大增加其使用复杂性的前提之下。

现在决定为新组件再增加一个属性——DropEnabled，其值为 True 时允许文件拖放操作，其值为 False 时则不允许文件拖放操作。

接着就是具体实现这个组件了。虽然实现这个组件不是本节的主题，但为了给读者一个完整的实例体验，还是将其实现过程写出来。

简单介绍一下完成支持文件拖放所需要的 API：



- ◆ DragAcceptFiles() 初始化某窗口，使其允许/禁止接受文件拖放。
- ◆ DragQueryFile() 查询拖放的文件名。
- ◆ DragFinish() 释放拖放文件时使用的资源。

基本原理：首先调用 DragAcceptFiles()函数初始化组件窗口，使其允许接受文件拖放，然后等待 WM\_DROPFILES 消息（一旦用户进行了拖放文件操作，组件窗口即可获得此消息），获得消息后即可使用 DragQueryFile()函数查询被拖放的文件名，最后调用 DragFinish()释放资源。

新组件的名称是 TDropFileListBox，其实现代码清单如下（如果在意其具体实现，请注意看代码中的注释）：

```
unit DropFileListBox;

interface

uses
    Windows, Messages, Classes, StdCtrls, ShellApi;

type
    // 自定义 OnDropFiles 事件的原型，该原型为事件的用户提供一个
    // 拖放文件的文件名列表
    TDropFileNotifyEvent = procedure (Sender: TObject;
        FileNames : TStringList) of object;

    // 自定义组件类
    TDropFileListBox = class(TListBox)
    private
        FEnabled : Boolean;
        FDropFile : TMyNotifyEvent; // OnDropFiles 事件的指针

        // DropFiles 函数接受系统的 WM_DROPFILES 消息，并转换成 OnDropFiles 事件
        procedure DropFiles(var Msg : TMessage); message WM_DROPFILES;
        procedure SetDropEnabled(bEnabled : Boolean);

    public
        constructor Create(AOwner : TComponent); override;

    published
        property OnDropFiles : TMyNotifyEvent read FDropFile
            write FDropFile;
        property DropEnabled : Boolean read FEnabled write SetDropEnabled;
    end;
```

```
procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Sunisoft', [TDropFileListBox]);
end;

constructor TDropFileListBox.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  FEnabled := true;
end;

procedure TDropFileListBox.SetDropEnabled(bEnabled : Boolean);
begin
  // 设置 DropEnabled 属性时
  FEnabled := Enabled;
  DragAcceptFiles(Handle, bEnabled);
end;

procedure TDropFileListBox.DropFiles(var Msg : TMessage);
var
  FileNames : TStringList;
  FileName : array [0 .. MAX_PATH - 1] of char;
  i, nCount : integer;
begin
  FileNames := TStringList.Create();
  // 取得拖放的文件总数
  nCount := DragQueryFile(Msg.WParam, $FFFFFFFF, @FileName, MAX_PATH);

  // 将所有文件名添加到 FileNames 列表中
  For i := 0 to nCount - 1 do
  begin
    DragQueryFile(Msg.WParam, i, FileName, MAX_PATH);
    FileNames.Add(FileName);
  end;

  // 释放资源
  DragFinish(Msg.WParam);
```



```
// 调用用户事件代码
if Assigned(FDropFile) then
    FDropFile(self, FileNames);
FileNames.Free();
end;

end.
```

首先，定义了 `OnDropFiles` 事件的函数指针原型：`TDropFileNotifyEvent`。

然后，在组件中，通过 `SetDropEnabled()` 方法设置组件窗口是否允许接受拖放文件的操作。而 `DropFiles()` 方法，也就是该组件最核心的部分，接受并处理 `WM_DROPFILES` 消息。

该组件的使用也很简单，最典型的应用如下：

```
procedure TForm1.DropFileListBox1DropFiles(Sender: TObject;
    FileNames: TStringList);
begin
    DropFileListBox1.Items.AddStrings(FileNames);
end;
```

设计组件，哪怕只是对现存组件做细微的扩充，也和设计类一样（其实，组件也是类），需要考虑接口的完备性和简单性。因为，有缺陷的组件和难用的组件同样是找不到用户的。

## 5.2.2 实例二：能显示图片的 Panel

`TDropFileListBox` 组件对原先的组件（`TListBox`）的扩充，并没有改变其本质及存在的作用。下面再看一个例子，对现有组件扩充，却改变了原先组件所扮演的角色。

在此需要显示一个图片，也许有人会说这很简单。是的，只要放一个 `TImage` 就行了。但是，`TImage` 并不能时时都满足我们的需求。例如，它不能作为容器（容器可以带着它上面的组件一起“跑”）；它也不能被放置在一些特殊组件上（如 `THeaderControl`，即使这样的需求比较少）。

好了，其实我们要的更像是一个能显示图片的 `Panel`，因为 `Panel` 可以作为容器，也可以放在那些特殊组件上。于是，顺理成章，似乎应该从 `TPanel` 派生。还差一点，应该找到它的基类 `TCustomPanel`，因为 `TPanel` 中已经加入了太多的 `Panel` 所特有的特性，有些是我们不需要的，而太多没有用的属性只会让用户迷惑。



**注意：**一般情况下，先考虑 `TCustomXXX` 作为自己组件的基类，除非是在如同 `TDropFileListBox` 那样并不会改变基类组件角色的情况下。

既然新组件就是能显示图片的 `Panel`，那就称之为 `TImagePanel`。

首先为新组件决定需要添加哪些属性，因为这些是直接和组件用户相关的，也就是确定其需求。

显而易见，作为能够显示图片的 Panel，要给新组件增加一个 Picture 属性。与 Picture 属性相关的，自然会想到需要 AutoSize 和 Transparent 属性，分别允许设置组件是否按照 Picture 的大小自动调整尺寸以及是否使得图片透明。另外，TCustomPanel 有 Caption 属性，只是其处于 protected 状态；或许还会需要在 ImagePanel 上写标题，这个属性看来有用，应该使它 published。在新的组件中，将该属性置于 published 节中而不需要重新声明该属性的类型和读写途径（即所谓的 overriding property），如此便可让 Delphi 开发环境的 Object Inspector 能够认识它并将它显示出来。有了 Caption，自然需要 Font，也 publish 它。另外，看一下 TPanel，就知道与 Caption 相关的，还有一个 Alignment 属性（标题对齐方式），同样 publish 它。

至此，新组件的接口应该算是比较圆满了。

接着，确定组件所需要添加的方法。不过对于 TImagePanel 来说，并不需要提供什么功能性的方法，因此可以跳过这一步。

面向用户的设计完成后，最后就是具体实现组件。

以下是源代码清单（如果对其具体实现细节感兴趣，可参考代码中的注释）：

```
unit ImagePanel;

interface

uses windows, extctrls, graphics, classes, controls;

type
  TImagePanel = class(TCustomPanel)
  private
    FPicture : TPicture;      // Picture 属性的“载体”成员
    FTransparent : Boolean;   // Transparent 属性的“载体”成员
    FAutoSize : Boolean;     // AutoSize 属性的“载体”成员

    procedure PictureChanged(Sender: TObject);
    procedure SetPicture(const Value: TPicture);
    procedure SetAutoSize(const Value: Boolean); reintroduce;
    procedure SetTransparent(const Value: Boolean);
    procedure SetFont(const Value : TFont);
    procedure SetCaption(const Value : TCaption);
    procedure SetAlignment(const Value : TAlignment);

  protected
    procedure Paint(); override; // 覆盖 TCustomPanel 的自绘方法
```



```
public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy(); override;

published
    // 增加以下 3 个属性
    property Picture : TPicture read FPicture write SetPicture;
    property Transparent : Boolean Read FTransparent
        Write SetTransparent default false;
    property AutoSize : Boolean Read FAutoSize Write SetAutoSize;

    // 从基类中发布（覆盖）以下 3 个属性
    property Font write SetFont;
    property Caption write SetCaption;
    property Alignment write SetAlignment;
end;

procedure Register; // 注册安装组件的过程

implementation

{ TImagePanel }

constructor TImagePanel.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);

    FPicture := TPicture.Create();
    // FPicture 被更改时，调用 PictureChanged 以重绘组件窗口
    FPicture.OnChange := PictureChanged;

    Repaint();
end;

destructor TImagePanel.Destroy;
begin
    FPicture.Free();
    FPicture := nil;

    inherited;
end;

// 自绘组件窗口，以显示图片
```

```
procedure TImagePanel.Paint;
const
  Alignments: array[TAlignment] of Longint = (DT_LEFT, DT_RIGHT,
    DT_CENTER);
var
  Flags: Longint;
  Rect: TRect;
  FontHeight: Integer;
begin
  // 设置画布属性
  Canvas.Brush.Style := bsClear;
  Canvas.Font := Font;

  // 绘制图片
  if Assigned(FPicture.Graphic) then
  begin
    // 若 AutoSize 属性为真, 则按照图片大小调整组件窗口尺寸
    if FAutoSize then
    begin
      Width := FPicture.Width;
      Height := FPicture.Height;
    end;

    if FPicture.Graphic.Transparent <> FTransparent then
      FPicture.Graphic.Transparent := FTransparent;

    // 利用画布 (Canvas) 对象进行绘图
    Canvas.StretchDraw(ClientRect, FPicture.Graphic);
  end
  else // 如果图片属性为空, 则以 Color 属性的颜色填充组件窗口
  begin
    Canvas.Brush.Color := Color;
    Canvas.FillRect(ClientRect);
  end;

  // 如果组件被指定了 Caption 属性, 则在组件上写出标题
  if Caption <> '' then
  begin
    Rect := GetClientRect;
    FontHeight := Canvas.TextHeight('W');
    // Caption 纵向居中
    Rect.Top := ((Rect.Bottom + Rect.Top) - FontHeight) div 2;
    Rect.Bottom := Rect.Top + FontHeight;
```



```
        Flags := DT_EXPANDTABS or DT_VCENTER or Alignments[Alignment];
        Flags := DrawTextBiDiModeFlags(Flags);
        DrawText(Canvas.Handle, PChar(Caption), -1, Rect, Flags);
    end;
end;

procedure TImagePanel.PictureChanged(Sender: TObject);
begin
    Repaint();
end;

procedure TImagePanel.SetAlignment(const Value: TAlignment);
begin
    inherited Alignment := Value;
    Repaint();
end;

procedure TImagePanel.SetAutoSize(const Value: Boolean);
begin
    FAutoSize := Value;
    Repaint();
end;

procedure TImagePanel.SetCaption(const Value: TCaption);
begin
    inherited Caption := Value;
    Repaint();
end;

procedure TImagePanel.SetFont(const Value: TFont);
begin
    inherited Font := Value;
    Repaint();
end;

procedure TImagePanel.SetPicture(const Value: TPicture);
begin
    FPicture.Assign(Value);
    Repaint();
end;

procedure TImagePanel.SetTransparent(const Value: Boolean);
```

```
begin
    FTransparent := Value;
    Repaint();
end;

procedure Register;
begin
    RegisterComponents('Sunisoft', [TImagePanel]);
end;

end.
```

代码中大多是一些属性设置的方法，其中最关键、最核心的部分，是覆盖了 `TCustomPanel` 的 `Paint()` 方法，即由新组件自己决定如何重新绘制自己的窗口，以将图片显示出来。不过，由于使用了 `TCanvas` 所带来的便利，程序显得非常短小精悍。

`Paint()` 方法被覆盖，则窗口的重绘完全由组件自己负责。`TImagePanel` 的 `Paint()` 方法算法非常简单。

首先，设置画布的属性：

```
Canvas.Brush.Style := bsClear;
Canvas.Font := Font;
```

然后进行图片的绘制，如果 `Picture` 属性没有被设置过，即为空的话，则使用 `Color` 属性指定的颜色填充整个组件窗口：

```
Canvas.Brush.Color := Color;
Canvas.FillRect(ClientRect);
```

如果 `Picture` 属性被设置了某个图片，则在组件窗口中画出该图片：

```
Canvas.StretchDraw(ClientRect, FPicture.Graphic);
```

可见，`Paint()` 方法主要就是在组件 `Canvas` 属性提供的画布上进行绘制，而 `Canvas` 又很好地对 Windows 的 GDI 函数进行了封装，使得 GDI 非常好用，同时也不必担心发生 GDI 资源泄漏的情况。

有必要再看一下与 `Picture` 属性相关的一些代码。`Picture` 属性是这样定义的：

```
property Picture : TPicture read FPicture write SetPicture;
```

它指明，获取 `Picture` 属性时，从 `FPicture` 成员获得；设置 `Picture` 属性时，则调用 `SetPicture()` 方法来写入。`SetPicture()` 方法：



```
procedure TImagePanel.SetPicture(const Value: TPicture);  
begin  
    FPicture.Assign(Value);  
    Repaint();  
end;
```

在 `SetPicture()` 方法中，使用对象赋值来将设置的图片“拷贝”给 `FPicture` 对象，而不能这样：

```
FPicture := Value;
```

原因已经在 4.5 节中说过。除非是简单类型（如 `Integer`）的赋值，才可以使用“`:=`”。

 **注意：** 设置属性时，对象赋值须用 `Assign`，简单类型可以用“`:=`”

短短百来行代码，就可以得到所需要的东西。可见，站在前人成果的基础上，可以做到事半功倍。

## 5.3 定制组件

当然，更具挑战性的一项工作就是完全从头开始定制符合自己需要的组件。如果所要创建的是若干彼此关联的组件而非单个组件的话，那么设计它们的构架层次关系则会令人兴奋。

本节将以一个极其微小的例子来介绍自制组件的设计及实现方法。

界面特效是非常吸引用户的一项技术，虽然软件真正能带给用户价值的是其功能，但是良好、美观的界面能让用户在看第一眼时就被吸引住。

在此要举的组件例子，就是为了实现某些特效——要求所有按钮在鼠标悬停时与鼠标移出时以及鼠标按下时能显示不同的外观；同时还要求所有的 `RadioButton`、`CheckBox` 也能做到这样。替换了这三种基础组件之后，整体界面将为之焕然一新。

为了实现上的简单，在此决定采用贴图的方式改变这些组件上的外观，即在不同的状态下，在组件的窗口上贴不同的图片以达到改变外观的效果。虽然，这样的实现比较消耗资源，不过在组件不太大的情况下（`Button`，`RadioButton`、`CheckBox` 等都不会很大），效果还是可以接受的。

 **注意：** 这里的组件只是作为讲解的例子，如果要想实现真正商业化的组件，则需要花费更多的精力在实现算法上。

### 5.3.1 TActiveButton

首先，需要确定创建组件的基类。

先考虑特效按钮。是否应该从 `TButton` 派生实现特效按钮呢？应该考虑，如果从某个

类派生，这个类能为新的实现带来什么便利。特效的按钮，由于其外观由贴图完成，因此 TButton 原来的自绘代码对于我们来说毫无用处。TImage 呢？算了吧，它不能作为容器，不符合按钮的需求。前面曾在 5.2 节中实现了一个可以贴图的 Panel——TImagePanel。TImagePanel 是从 TCustomPanel 派生的，这次只是需求比 TImagePanel 更多（要交替地贴多幅图片）而已，因此决定还是从 TCustomPanel 派生。

这里有必要插入一段讨论。也许有人会问，是否可以从已经实现了的 TImagePanel 派生呢？我想答案是“没什么不可以”。那此处为什么还是从 TCustomPanel 派生呢？首先是基于本节目的原因，本节所讨论的是从头定制一套组件的设计方法，与 5.2 节的需求有所区分。其次是基于效率的原因，这在本节最后的实现讨论中会提到。

接着，来讨论一下算法。

组件本身存储着表示其 4 种状态（正常、鼠标悬停、鼠标按下、不可用状态）的 4 个位图，这些位图分别对应一个 TPicture 实例：FPicOnMouseDown（鼠标按下时的位图）、FPicOnMouseIn（鼠标悬停时的位图）、FPicDisabled（不可用状态时的位图）、FPicNormal（正常状态的位图）。另外有一个 TPicture 类型的指针——FPicture，其在不同的状态下指向与状态对应的 TPicture 实例。

组件的覆盖基类的 Paint() 方法，负责在组件窗口上绘制 FPicture 指针所指向的 TPicture 实例所存储的位图。

获取组件状态的算法是：如果 Enabled 属性为 false，则组件处于不可用状态，并保持不变；否则，当鼠标移进（OnMouseMove）时标识组件处于悬停状态，并建立一个计时器（每隔一定的时间间隔就检测鼠标位置是否已经移出组件窗口），一旦鼠标位置移出组件，则立即恢复成正常状态；当鼠标在按钮窗口上按下时（OnMouseDown）以及鼠标按键弹回时（OnMouseUp），分别标识组件进入鼠标按下状态和鼠标悬停状态（或者正常状态，如用户在按钮上按下鼠标不放，拖出组件再释放）。

最后，为组件设计用户接口，即提供哪些属性与方法。

很显然，我们要允许用户自己提供并设置按钮的 4 种状态的位图图片，于是需要提供 4 个 TPicture 类型的属性——PicOnMouseDown、PicOnMouseIn、PicDisabled、PicNormal，并且这 4 个属性正好与算法讨论中的 4 个 TPicture 实例一一对应，其实这 4 个 TPicture 实例正是这 4 个属性的载体。

有了 5.2 节的经验，我们知道还需要 AutoSize 和 Transparent 属性。

其余的可以在 TCustomPanel 中寻找。TCustomPanel 为了使得从它派生的组件不会有太多不必要的属性，它将很多属性都声明在 protected 节中，因此外界看不到这些属性。这样，其派生类就可以自行决定需要 publish 哪些属性，而继续隐藏哪些属性。

按钮既然允许处于不可用状态，那么 Enabled 属性是少不了的，它正是处于 TCustomPanel 的 protected 节保护之下的，因此，在此需要 publish 它。

最后，作为按钮，当然要处理用户的 OnClick 之类的鼠标事件。这些事件与 Enabled 属性一样是被 protected 的，于是同样需要被 published。

新组件是一个特效的按钮，正如本节标题一样，它被命名为 TActiveButton。

一起来看一下组件的实现代码（请注意其中的注释）：



```
unit ActiveButton;  
  
interface  
  
uses  
    Windows, Graphics, Classes, Controls, ExtCtrls, Messages;  
  
type  
  
    TActiveButton = class(TCustomPanel)  
    private  
        FPicOnMouseDown : TPicture; // 鼠标按下时的位图实例  
        FPicOnMouseIn : TPicture; // 鼠标悬停时的位图实例  
        FPicDisabled : TPicture; // 不可用状态时的位图实例  
        FPicNormal : TPicture; // 正常状态下的位图实例  
        FPicture : TPicture; // TPicture 指针, 实时指向不同的位图实例  
  
        FAutoSize : Boolean; // AutoSize 属性的载体  
        FTransparent : Boolean; // Transparent 属性的载体  
  
        FMouseInFlag : Boolean; // 鼠标是否位于按钮窗口内的标志  
        FTimer : TTimer;  
  
        procedure PictureChanged(Sender: TObject);  
        procedure MouseOut();  
        procedure TimerProc(Sender: TObject); // 计时器的回调函数  
  
        procedure SetPicOnMouseIn(Value : TPicture);  
        procedure SetPicOnMouseDown(Value : TPicture);  
        procedure SetPicDisabled(Value : TPicture);  
        procedure SetPicNormal(Value : TPicture);  
  
    protected  
        // 覆盖基类的 MouseDown, 以截获鼠标按下消息  
        procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X,  
            Y: Integer); override;  
        // 覆盖基类的 MouseUp, 以截获鼠标弹回消息  
        procedure MouseUp(Button: TMouseButton; Shift: TShiftState; X,  
            Y: Integer); override;  
        // 覆盖基类的 MouseMove, 以截获鼠标移动消息  
        procedure MouseMove(Shift: TShiftState; X, Y: Integer); override;  
  
        procedure SetEnabled(Value: Boolean); override;
```

```
procedure SetTransparent(const Value: Boolean); virtual;
procedure SetAutoSize(Value: Boolean); override;
procedure Paint(); override;

public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy(); override;

published
  // 鼠标悬停时的图片, 其载体为 FPicOnMouseIn
  property PicOnMouseIn : TPicture Read FPicOnMouseIn
    Write SetPicOnMouseIn;
  // 鼠标按下时的图片, 其载体为 FPicOnMouseDown
  property PicOnMouseDown : TPicture Read FPicOnMouseDown
    Write SetPicOnMouseDown;
  // 不可用状态时的图片, 其载体为 FPicDisabled
  property PicDisabled : TPicture Read FPicDisabled
    Write SetPicDisabled;
  // 正常状态的图片, 其载体为 FPicNormal
  property PicNormal : TPicture Read FPicNormal Write SetPicNormal;

  property AutoSize : Boolean Read FAutoSize Write SetAutoSize;
  property Transparent : Boolean Read FTransparent
    Write SetTransparent default false;

  // 基类 protected 的属性, 在此 published
  property Enabled;

  property OnClick;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
end;

procedure Register;

implementation

constructor TActiveButton.Create(AOwner: TComponent);
begin
  // 构造函数创建 4 个 TPicture 实例
  inherited Create(AOwner);
```



```
FPicOnMouseIn := TPicture.Create();
FPicOnMouseDown := TPicture.Create();
FPicDisabled := TPicture.Create();
FPicNormal := TPicture.Create();

FPicOnMouseIn.OnChange := PictureChanged;
FPicOnMouseDown.OnChange := PictureChanged;
FPicDisabled.OnChange := PictureChanged;
FPicNormal.OnChange := PictureChanged;

FPicture:= FPicNormal;

BevelOuter := bvNone;
Caption := ' ';
end;

destructor TActiveButton.Destroy();
begin
    // 析构函数销毁所有成员对象

    FTimer.Free();
    FTimer := nil;

    FPicNormal.Free();
    FPicNormal := nil;

    FPicDisabled.Free();
    FPicDisabled := nil;

    FPicOnMouseDown.Free();
    FPicOnMouseDown := nil;

    FPicOnMouseIn.Free();
    FPicOnMouseIn := nil;

    FPicture := nil;

    inherited Destroy();
end;

procedure TActiveButton.Paint(); //override
var
```

```
    Rect : TRect;
begin
    // 重新定义的 Paint 方法
    if Assigned(FPicture.Graphic) then
    begin
        Canvas.Brush.Style := bsClear;
        Canvas.Font := Font;
        if FPicture.Graphic.Transparent <> FTransparent then
            FPicture.Graphic.Transparent := FTransparent;
        if FAutoSize then
        begin
            Height := FPicture.Height;
            Width := FPicture.Width;
        end;
        Canvas.StretchDraw(ClientRect, FPicture.Graphic);
        Rect := GetClientRect();
        DrawText(
            Canvas.Handle,
            PChar(Caption),
            -1,
            Rect,
            DT_CENTER or DT_SINGLELINE or DT_VCENTER
        );
    end
    else
        inherited Paint();
end;

procedure TActiveButton.PictureChanged(Sender: TObject);
begin
    // 4 个 TPicture 属性被更改时, 重绘按钮
    RePaint();
end;

procedure TActiveButton.SetEnabled(Value: Boolean); //override
begin
    // 设置 Enabled 属性, 根据新的值改变 FPicture 所指向的图片, 并重绘
    if (
        (not Value) and
        (Assigned(FPicDisabled.Graphic))
    ) then
        FPicture := FPicDisabled
    else
```



```
        FPicture := FPicNormal;

        RePaint();
        Inherited SetEnabled(Value);
    end;

procedure TActiveButton.SetPicDisabled(Value: TPicture);
begin
    FPicDisabled.Assign(Value);

    RePaint();
end;

procedure TActiveButton.SetPicOnMouseDown(Value: TPicture);
begin
    FPicOnMouseDown.Assign(Value);
end;

procedure TActiveButton.SetPicOnMouseIn(Value: TPicture);
begin
    FPicOnMouseIn.Assign(Value);
end;

procedure TActiveButton.SetPicNormal(Value: TPicture);
begin
    FPicNormal.Assign(Value);

    RePaint();
end;

procedure TActiveButton.SetAutoSize(Value: Boolean);
begin
    FAutoSize := Value;

    RePaint();
end;

procedure TActiveButton.SetTransparent(const Value: Boolean);
begin
    FTransparent := Value;

    RePaint();
end;
```

```
procedure TActiveButton.MouseOut();
begin
    // 截获到鼠标移出后
    FMouseInFlag := false;
    FTimer.Free;
    FTimer := nil;

    if Enabled then
        FPicture := FPicNormal
    else
        FPicture := FPicDisabled;
    Repaint();
end;

procedure TActiveButton.TimerProc(Sender: TObject);
var
    CurPos : TPoint;
    LeftTop : TPoint;
    RightBottum : TPoint;
begin
    // FTimer 的 OnTimer 事件处理函数
    // 判断鼠标指针是否仍然处于按钮窗口内
    // 如果已经移出, 则调用 MouseOut

    LeftTop.x := 0;
    LeftTop.y := 0;
    RightBottum.x := Width;
    RightBottum.y := Height;
    GetCursorPos(CurPos);

    if Parent <> nil then
        begin
            LeftTop := ClientToScreen(LeftTop);
            RightBottum := ClientToScreen(RightBottum);
        end
    else
        MouseOut();

    if (
        (CurPos.x > LeftTop.x) and
        (CurPos.x < RightBottum.x) and
        (CurPos.y > LeftTop.y) and
```



```
        (CurPos.y < RightBottom.y)
    ) then
        Exit;

    MouseOut();
end;

procedure TActiveButton.MouseDown(Button: TMouseButton; Shift:
    TShiftState; X, Y: Integer);
begin
    // 鼠标按下, 更改 FPicture 指针指向 FPicOnMouseDown
    // 然后调用 inherited 以调用基类的 MouseDown 方法
    if Button = mbLeft then
    begin
        if (
            Assigned(FPicOnMouseDown.Graphic) and
            Enabled
        ) then
        begin
            FPicture := FPicOnMouseDown;
            Repaint();
        end;
    end;

    inherited;
end;

procedure TActiveButton.MouseMove(Shift: TShiftState; X, Y: Integer);
begin
    // 鼠标在按钮上移动, 如果上一个状态鼠标不在按钮窗口之上
    // 则表明鼠标移入按钮, 将 FPicture 指针指向 FPicOnMouseIn
    // 并建立计时器检查鼠标是否移出
    if (
        (X < 0) or
        (Y < 0) or
        (X > Width) or
        (Y > Height)
    ) then
    begin
        MouseOut();
        inherited;
        Exit;
    end;
end;
```

```
if (
    (FMouseInFlag) or
    (not Enabled) or
    (FPicOnMouseIn.Graphic = nil)
) then
begin
    inherited;
    Exit;
end;

FMouseInFlag := true;

if (
    Assigned(FPicOnMouseIn.Graphic) and
    Enabled
) then
begin
    FPicture := FPicOnMouseIn;
    RePaint();
end;

if (
    (FTimer = nil) and
    Assigned(FPicOnMouseIn.Graphic)
) then
begin
    FTimer := TTimer.Create(nil);
    FTimer.Interval := 168;
    FTimer.OnTimer := TimerProc;
end;

inherited;
end;

procedure TActiveButton.MouseUp(Button: TMouseButton; Shift:
TShiftState;
    X, Y: Integer);
begin
    // 鼠标按键弹回
    if (
        (X < 0) or
        (Y < 0) or
```



```
(X > Width) or
(Y > Height)
) then
    Exit;

if Button = mbLeft then
begin
    if (
        (FMouseInFlag) and
        (Assigned(FPicOnMouseIn.Graphic)) and
        Enabled
    ) then
        FPicture := FPicOnMouseIn
    else if Enabled then
        FPicture := FPicNormal
    else
        FPicture := FPicDisabled;

    Repaint();
end;

inherited;
end;

procedure Register;
begin
    RegisterComponents('Sunisoft', [TActiveButton]);
end;

end.
```

该组件相当部分的代码都用在了处理组件的 4 个状态的切换上。覆盖的 `MouseMove()` 方法的职责在于捕获鼠标指针移入按钮窗口的时机，以及对用户按下按钮不放拖出按钮窗口才放开的情况的判断。`MouseMove()` 方法首先判断鼠标是否还在按钮窗口之内，如果不在，则调用 `MouseOut()` 处理鼠标移出：

```
if (
    (X < 0) or
    (Y < 0) or
    (X > Width) or
    (Y > Height)
) then
```

```
begin
    MouseOut();
    inherited;
    Exit;
end;
```

然后，判断上一个状态时鼠标指针是否处于按钮窗口之内，即 `FMouseInFlag` 是否为 `true`，是则不必做任何处理，退出函数：

```
if (
    (FMouseInFlag) or
    (not Enabled) or
    (FPicOnMouseIn.Graphic = nil)
) then
begin
    inherited;
    Exit;
end;
```

如果 `FMouseInFlag` 为 `false`，则表示前一个状态时，鼠标指针在按钮窗口外，现在移入了按钮窗口，则设置 `FMouseInFlag` 为 `true`，并将 `FPicture` 指向 `FPicOnMouseIn`，然后创建计时器，开始周期性地检查鼠标是否仍然处于窗口内：

```
FMouseInFlag := true;

if (
    Assigned(FPicOnMouseIn.Graphic) and
    Enabled
) then
begin
    FPicture := FPicOnMouseIn;
    Repaint();
end;

if (
    (FTimer = nil) and
    Assigned(FPicOnMouseIn.Graphic)
) then
begin
    FTimer := TTimer.Create(nil);
    FTimer.Interval := 168;
```



```
FTimer.OnTimer := TimerProc;  
end;
```

最后别忘记调用 `inherited;`，否则程序员编写的 `OnMouseMove` 事件的代码将得不到执行。

计时器被创建后，每隔一定时间（168ms）就会调用 `TimerProc()`方法，以检查鼠标指针是否还在按钮窗口内，如果已经移出了，则调用 `MouseOut()`。

在 `MouseOut()`方法中将 `FMouseInFlag` 设置为 `false`，并且销毁计时器，然后将 `FPicture` 指针指向 `FPicNormal`，按钮回归到正常状态：

```
FMouseInFlag := false;  
FTimer.Free;  
FTimer := nil;  
  
if Enabled then  
    FPicture := FPicNormal  
else  
    FPicture := FPicDisabled;  
RePaint();
```

覆盖的 `MouseDown()`方法和 `MouseUp()`方法改变按钮状态与 `MouseMove()`十分相似。

覆盖的 `Paint()`方法与 `TImagePanel` 的 `Paint()`相比，并没有复杂多少，惟一不同的是如果 `FPicture` 指针指向的图片为空，则调用基类的 `Paint()`方法：

```
inherited Paint();
```

前面提到过，我们选择将 `TActiveButton` 从 `TCustomPanel` 派生而不是从 5.2 节中实现的 `TImagePanel` 派生，除了章节安排上的原因，还有一个原因就是如果从 `TImagePanel` 派生，组件就会从 `TImagePanel` 继承得到一个 `Picture` 属性，而该属性在 `TActiveButton` 中是无用的。

### 5.3.2 TActiveCheckBox

按照最初的需求，还要实现有特效的 `CheckBox` 和 `RadioButton`，该怎么办呢？很自然，又想到了贴图的方法，因为刚刚用贴图实现了一个特效按钮。那如何寻找基类呢？从 `TCheckBox/TCustomCheckBox` 和 `TRadioButton` 派生？不！那并不能减少任何的工作量，相反可能会更复杂。

其实，静下心来仔细考虑一下就会发现，`CheckBox` 和 `RadioButton` 具有两种状态（选中和非选中），而每处于一种状态时，其行为与刚实现的 `ActiveButton` 何其类似？也就是说，`CheckBox` 和 `RadioButton` 不过是具有两种状态的 `ActiveButton`！

如果发现了这个本质，那么就已经成功了一半了。

在此可以将特效的 `CheckBox` 定义为：具有两种状态（`Checked` 和 `UnChecked`）的 `ActiveButton`。那 `RadioButton` 呢？它只是在 `CheckBox` 的基础上，要求一组 `RadioButton` 中只有一个可以被选中而已。

将特效的 `CheckBox` 命名为 `TActiveCheckBox`，将特效的 `RadioButton` 命名为 `TActiveRadioButton`。

现在可以推论出：`TActiveCheckBox` 是一种特殊的 `TActiveButton`，`TActiveRadioButton` 是一种特殊的 `TActiveCheckBox`。

根据这个结论可以明白，应该从 `TActiveButton` 派生实现 `TActiveCheckBox`，从 `TActiveCheckBox` 派生实现 `TActiveRadioButton`。

这将大大降低工作的复杂度，并有效地复用之前实现的代码。复用是美好的！

以下是 `TActiveCheckBox` 组件的源代码清单：

```
unit ActiveCheckBox;

interface

uses
  Windows, Graphics, Classes,
  ActiveButton;

type

  TActiveCheckBox = class(TActiveButton)
  private
    FChecked : Boolean;

    // UnChecked 和 Checked 状态各对应 4 个图片
    // 因此一共需要 8 个 TPictuer 实例，很费资源哦：)
    FPicNormalChecked : TPicture;
    FPicMouseInChecked : TPicture;
    FPicMouseDownChecked : TPicture;
    FPicDisabledChecked : TPicture;
    FPicNormal : TPicture;
    FPicMouseIn : TPicture;
    FPicMouseDown : TPicture;
    FPicDisabled : TPicture;

    procedure SetPicDisabled(const Value: TPicture);
    procedure SetPicDisabledChecked(const Value: TPicture);
    procedure SetPicMouseDown(const Value: TPicture);
    procedure SetPicMouseDownChecked(const Value: TPicture);
```



```
procedure SetPicMouseIn(const Value: TPicture);
procedure SetPicMouseInChecked(const Value: TPicture);
procedure SetPicNormal(const Value: TPicture);
procedure SetPicNormalChecked(const Value: TPicture);

procedure PictureChanged(Sender: TObject);

protected
  procedure SetChecked(const Value: Boolean);
  procedure UpdateCheckState();
  procedure Click(); override;
  procedure DblClick(); override;

public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy(); override;

published
  property Checked : Boolean read FChecked write SetChecked
    default false;
  property PicNormalChecked : TPicture read FPicNormalChecked
    write SetPicNormalChecked;
  property PicOnMouseInChecked : TPicture read FPicMouseInChecked
    write SetPicMouseInChecked;
  property PicOnMouseDownChecked : TPicture read FPicMouseDownChecked
    write SetPicMouseDownChecked;
  property PicDisabledChecked : TPicture read FPicDisabledChecked
    write SetPicDisabledChecked;
  property PicNormal : TPicture read FPicNormal write SetPicNormal;
  property PicOnMouseIn : TPicture read FPicMouseIn write SetPicMouseIn;
  property PicOnMouseDown : TPicture read FPicMouseDown
    write SetPicMouseDown;
  property PicDisabled : TPicture read FPicDisabled write
SetPicDisabled;

end;

procedure Register;

implementation

constructor TActiveCheckBox.Create(AOwner: TComponent);
```

```
begin
    // 构造函数创建 TPicture 实例
    inherited Create(AOwner);

    FPicNormalChecked := TPicture.Create();
    FPicMouseInChecked := TPicture.Create();
    FPicMouseDownChecked := TPicture.Create();
    FPicDisabledChecked := TPicture.Create();

    FPicNormal := TPicture.Create();
    FPicMouseIn := TPicture.Create();
    FPicMouseDown := TPicture.Create();
    FPicDisabled := TPicture.Create();

    FChecked := false;

    FPicNormal.OnChange := PictureChanged;
    FPicNormalChecked.OnChange := PictureChanged;
    FPicMouseIn.OnChange := PictureChanged;
    FPicMouseInChecked.OnChange := PictureChanged;
    FPicMouseDown.OnChange := PictureChanged;
    FPicMouseDownChecked.OnChange := PictureChanged;
    FPicDisabled.OnChange := PictureChanged;
    FPicDisabledChecked.OnChange := PictureChanged;
end;

destructor TActiveCheckBox.Destroy();
begin
    // 析构函数销毁所有 TPicture 实例

    FPicDisabled.Free();
    FPicDisabled := nil;

    FPicMouseDown.Free();
    FPicMouseDown := nil;

    FPicMouseIn.Free();
    FPicMouseIn := nil;

    FPicNormal.Free();
    FPicNormal := nil;

    FPicDisabledChecked.Free();
```



```
    FPicDisabledChecked := nil;

    FPicMouseDownChecked.Free();
    FPicMouseDownChecked := nil;

    FPicMouseInChecked.Free();
    FPicMouseInChecked := nil;

    FPicNormalChecked.Free();
    FPicNormalChecked := nil;

    inherited Destroy();
end;

procedure TActiveCheckBox.PictureChanged(Sender: TObject);
begin
    UpdateCheckState();
end;

procedure TActiveCheckBox.SetChecked(const Value: Boolean);
begin
    FChecked := Value;

    UpdateCheckState();
end;

procedure TActiveCheckBox.SetPicDisabled(const Value: TPicture);
begin
    FPicDisabled.Assign(Value);
end;

procedure TActiveCheckBox.SetPicDisabledChecked(const Value:
TPicture);
begin
    FPicDisabledChecked.Assign(Value);
end;

procedure TActiveCheckBox.SetPicMouseDown(const Value: TPicture);
begin
    FPicMouseDown.Assign(Value);
end;

procedure TActiveCheckBox.SetPicMouseDownChecked(const Value:
```

```
TPicture);
begin
    FPicMouseDownChecked.Assign(Value);
end;

procedure TActiveCheckBox.SetPicMouseIn(const Value: TPicture);
begin
    FPicMouseIn.Assign(Value);
end;

procedure TActiveCheckBox.SetPicMouseInChecked(const Value: TPicture);
begin
    FPicMouseInChecked.Assign(Value);
end;

procedure TActiveCheckBox.SetPicNormal(const Value: TPicture);
begin
    FPicNormal.Assign(Value);

    if csDesigning in ComponentState then
        UpdateCheckState();
end;

procedure TActiveCheckBox.SetPicNormalChecked(const Value: TPicture);
begin
    FPicNormalChecked.Assign(Value);

    if csDesigning in ComponentState then
        UpdateCheckState();
end;

procedure TActiveCheckBox.UpdateCheckState();
begin
    // 根据 FChecked 设置不同的两组图片

    if FChecked then
    begin
        inherited PicNormal := FPicNormalChecked;
        inherited PicOnMouseIn := FPicMouseInChecked;
        inherited PicOnMouseDown := FPicMouseDownChecked;
        inherited PicDisabled := FPicDisabledChecked;
    end
    else
```



```
begin
    inherited PicNormal := FPicNormal;
    inherited PicOnMouseIn := FPicMouseIn;
    inherited PicOnMouseDown := FPicMouseDown;
    inherited PicDisabled := FPicDisabled;
end;

RePaint();
end;

procedure TActiveCheckBox.Click;
begin
    // 覆盖基类的 Click, 截获鼠标单击消息
    Checked := not Checked;
    UpdateCheckState();

    inherited;
end;

procedure TActiveCheckBox.DblClick;
begin
    // 覆盖基类的 DblClick, 截获鼠标双击消息
    Checked := not Checked;
    UpdateCheckState();

    inherited;
end;

procedure Register;
begin
    RegisterComponents('Sunisoft', [TActiveCheckBox]);
end;

end.
```

看完代码, 就会发现, TActiveCheckBox 由于充分重用了 TActiveButton, 其代码异常简单。大多数都是属性赋值的方法, 惟一特别的就是 UpdateCheckState()方法, 它负责根据是否选中的状态来切换两组图片。在 UpdateCheckState()方法中, 判断 FChecked 的值, 然后分别向基类的 PicNormal、PicOnMouseDown、PicOnMouseIn、PicDisabled 4 个属性赋值, 接着重绘组件窗口:

```
if FChecked then
```

```
begin
    inherited PicNormal := FPicNormalChecked;
    inherited PicOnMouseIn := FPicMouseInChecked;
    inherited PicOnMouseDown := FPicMouseDownChecked;
    inherited PicDisabled := FPicDisabledChecked;
end
else
begin
    inherited PicNormal := FPicNormal;
    inherited PicOnMouseIn := FPicMouseIn;
    inherited PicOnMouseDown := FPicMouseDown;
    inherited PicDisabled := FPicDisabled;
end;

RePaint();
```

100 行左右的代码就完成了 TActiveCheckBox，即使是最核心的 UpdateCheckState() 方法也是简单之极。复用的确是美好的！

### 5.3.3 TActiveRadioButton

完成了特效按钮 TActiveButton、特效复选按钮 TactiveCheckBox 之后，最后来实现特效单选按钮 TActiveRadioButton。

5.3.2 节中已经提到过，可以把 TActiveRadioButton 当作一种较为特殊的 TactiveCheckBox，其特殊之处只是同一组的 TActiveRadioButton 中只能有一个被选中，并且鼠标单击都只是选中，而不像复选按钮那样在选中/非选中两种状态间进行切换。

以下是 TActiveRadioButton 的代码清单：

```
unit ActiveRadioButton;

interface

uses
    Windows, Messages, SysUtils, Classes, Controls, ExtCtrls, ActiveButton,
    ActiveCheckBox;

type
    TActiveRadioButton = class(TActiveCheckBox)
    private
        FGroupIndex : Integer;

        procedure UnCheckGroup();
```



```
protected
    procedure SetChecked(const Value: Boolean); override;
    procedure Click(); override;
    procedure DblClick(); override;

published
    property GroupIndex : Integer read FGroupIndex write FGroupIndex
        default 0;

end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Sunisoft', [TActiveRadioButton]);
end;

procedure TActiveRadioButton.Click;
begin
    inherited;

    Checked := true;
end;

procedure TActiveRadioButton.DblClick;
begin
    inherited;

    Checked := true;
end;

procedure TActiveRadioButton.SetChecked(const Value: Boolean);
begin
    if Value and (not Checked) then
        UnCheckGroup();

    inherited;
end;

procedure TActiveRadioButton.UnCheckGroup;
```

```
var
  i : Integer;
begin
  if Parent = nil then
    Exit;

  for i := 0 to Parent.ControlCount - 1 do
  begin
    if not (Parent.Controls[i] is TActiveRadioButton) then
      continue;
    if (Parent.Controls[i] as TActiveRadioButton).GroupIndex
      <> GroupIndex then
      continue;
    (Parent.Controls[i] as TActiveRadioButton).Checked := false;
  end;
end;

end.
```

`TActiveRadioButton` 的实现代码仅仅与其和 `TActiveCheckBox` 之间的差异相关，也就是说，它与基类之间有多少差异，那么它的代码复杂度就有多少。

由于分析出了 `TActiveRadioButton` 是一种特殊的 `TActiveCheckBox`，并且清楚地知道它们之间的两个差别：

1. 一组组件中，只有一个可以处于 `Checked` 状态。
2. 鼠标单击意味着选中。

因此，我们的代码也就围绕着这两点来实现。

首先，`TActiveRadioButton` 增加了一个 `GroupIndex` 属性，用于将组件分组。`GroupIndex` 值相同的多个 `TActiveRadioButton` 组件为一个组。同组的组件中，只允许有一个被选中。

其次，它覆盖了其基类 `TActiveCheckBox` 的 `protected` 的虚方法——`SetChecked()`。该方法用于对 `Checked` 属性进行设置。`TActiveRadioButton` 覆盖了该方法，得以接管对 `Checked` 属性的设置：

```
if Value and (not Checked) then
  UnCheckGroup();
inherited;
```

在设置 `Checked` 属性时，如果将 `Checked` 属性设置为 `true`，则调用 `UnCheckGroup()` 来使得与其同组的其他单选按钮的 `Checked` 属性变为 `false`。`UnCheckGroup()` 方法则在父窗口中遍历所有组件。根据 RTTI，如果组件类别同样为 `TActiveRadioButton` 并且与本组件属于同一个组（`GroupIndex` 属性值相同），则将其 `Checked` 属性设置为 `false`。



```
var
    i : Integer;
begin
    if Parent = nil then
        Exit;

    for i := 0 to Parent.ControlCount - 1 do
    begin
        if not (Parent.Controls[i] is TActiveRadioButton) then
            continue;
        if (Parent.Controls[i] as TActiveRadioButton).GroupIndex
            <> GroupIndex then
            continue;
        (Parent.Controls[i] as TActiveRadioButton).Checked := false;
    end;
```

最后，TActiveRadioButton 覆盖了基类的 Click()和 DblClick()方法，用于处理鼠标单击和双击消息。

```
procedure TActiveRadioButton.Click;
begin
    inherited;
    Checked := true;
end;

procedure TActiveRadioButton.DblClick;
begin
    inherited;
    Checked := true;
end;
```

可以看到，它们都只是简单地将本组件的 Checked 属性设置为 true 即可。

至此，已经完成了一个单线的三层的组件构架。

当然，这些组件虽然可以使用，不过实现上还是有所缺陷，最突出的就是资源占用过大。如果要真正作为商用，一定要加以更改。不过，在此作为组件编写的演示实例，却是绰绰有余了。

## 5.4 光盘上的组件以及演示程序代码

配书光盘中带有本章所介绍的所有组件的源代码以及一个演示这些组件效果的示例程序的可执行文件与代码。

组件代码存放在配书光盘中的 Component 目录下。

要使用这些组件，需要先在 Delphi 的 IDE 中安装它们。

打开其中的“UIComponent.dpk”文件，可以把它当作组件的 project（包含多个组件的工程）。打开后，单击“Install”即可完成安装。还要记得，在 Delphi 环境的 Library Path 列表中加入组件代码所在的目录，否则 Delphi 会提示找不到组件的代码文件。

演示程序放在配书光盘中的 CompDemo 目录下。

直接运行 UICmpDemo.exe 可以直接看到效果，如图 5.1 所示。同时该目录下还有该演示程序的代码。



图 5.1 自制组件演示程序的界面

该程序中，使用在本章中所实现的 TImagePanel、TActiveButton、TActiveCheckBox、TActiveRadioButton 4 个组件。

该程序模拟一个登录过程。运行该程序，根据不同的选择，单击“登录”按钮可以得到不同的结果，如图 5.2 所示。



图 5.2 运行程序后单击“登录”按钮



## 5.5 小 结

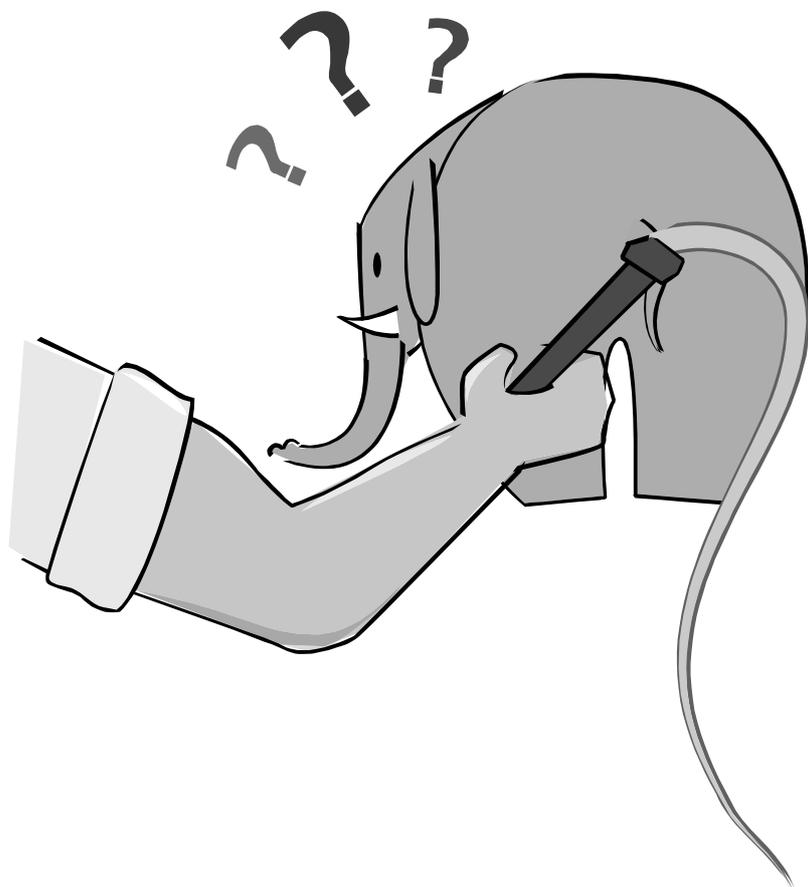
Delphi 的生命力更多是来自于其 VCL 库的可扩展性，这样用户才能用到更多、更好的第三方厂商提供的组件（库）。因为复用是美好的，而可扩展的组件库为用户提供了一种代码复用的方式。

另外，学习扩展组件库的方法更大的意义在于彻底了解组件的设计期、运行期机制，这对于哪怕只是使用组件开发应用程序也是很有好处的，正如我们小学、中学、大学都教授数学，而目的并不是要把每个人都培养成数学家。

在有了实现组件的能力基础之后，在设计多个组件时，需要花更多的精力设计组件之间的构架关系。因为良好的设计可以使你事半功倍……

## 第 6 章 代码设计基础

学会一种程序设计语言，是一回事；学会如何以此语言设计并实现有效的程序，又是一回事。——Scott Meyers



设计的关键是“抽象”！



见过很多 Delphi 程序，有面团式的代码——整个程序揉成一团，功能模块拆开谁都无法工作；有面条式的代码——模块间彼此交叉链接，改动一个模块就会牵动全身。

本章将试图告诉读者创建良好设计代码的一些基本原则。

## 6.1 创建良好设计的代码

设计并非是使得代码完美，完美并不存在于这个世界上。

设计是在实现的基础上的更高的要求。写软件，首先当然是给人用的，也就是实现功能。然后，就是便于自己或其他人理解和维护，这就是设计的目的。良好设计的代码，更易被理解，被维护，被复用。而这些好处的最终结果是，可以大大降低开发成本，并积累一些可重用的代码库。

如何创建良好设计的代码呢？

首先，要建立编写程序是一种艺术创作过程的信仰。程序员是需要有一点信仰的。比如，同样完成一个算法，代码的写法千变万化，可以写得很烂，也可以尽自己的能力把它写好。一般来说，写烂代码速度更快，而老板可能只关心你做出了什么功能，并不关心是如何做的。写烂代码可能更容易取悦老板，况且之后的维护工作不见得还是由你来完成。然而，在这种情况下，我想包括你我在内的程序员们，是不会选择写烂代码交差的。这就是一种信仰，写出来的代码要对得起自己，要可以毫无愧色地对别人说，“这是我写的程序！”。

虽然经常可以听到“程序员只不过是代码工人”、“人家印度都是高中生写代码”之类的说法。但是，代码工人只是编程的机器，没有思想，只懂得实现，那是 coder、coding machine。而信仰告诉我们，我们要做的是 programmer 甚至 designer！程序员应该是有思想的，会用代码来表达自己的思想。程序员会把程序当作自己的作品，在时间允许的情况下愿意对它精雕细琢。而程序员更相信——优美的代码足以使人延年益寿！

其次，要学会不断反思、总结，以积累经验。良好的设计并非一蹴而就的，需要不断地对最初的设计进行调整和重构。最初的设计往往是想当然的，而随着开发的进展，对需求的理解更深刻后，可能会发现设计的不合理之处。这时不要犹豫，立刻动手把代码往更优的方向拉，正所谓“不破不立”。因为最初的不良设计，随着整个项目代码的扩展，不合理处会被以几何级的速度放大。如果由于舍不得原来的代码，致使到项目末期才发现工作无法继续，这才是最致命的。

最后，需要不断地学习他人的经验。看过《设计模式》一书的人，一定会为其中先人的成果所叹服。或许读者会发现，其中某些模式的手法其实自己已经使用过了或正在使用。“模式”正是这样一种被很多系统、很多人所实践的并总结出来的设计经验。学习这些，可以使自己站在更多前人的肩膀上；然后，甚至在自己编写大量的代码的过程中，能够将自己设计中惯用的某些手法也提炼、整理成模式——这也是一种重用。

学习他人经验的最容易的途径，就是经常去看一些经典的代码以及相关的书籍。例如，VCL 库本身的代码就是很好的例子。而书籍方面，大多数涉及 OOP 设计的书籍都是以 C++

为讲述语言的，而以 Delphi 来讲述的则非常少，这也正是为什么在本书第 1 章中提出“真正聪明的程序员用 C++ 来理解 Delphi”的原因。

## 6.2 使用 OOP 进行代码设计

“设计”一词也许会给人一种虚无缥缈、无法掌控的感觉，但其实还是有一定的原则性的东西存在的，尤其在使用面向对象编程技术进行设计时。

### ◆ 原则一：使你的代码简单

自己编写的代码并不仅仅是给计算机执行的，其更大的价值在于能被其他人以及自己所理解。晦涩的代码会使得其难以被维护而可能被弃置。

因此，应尽量保持代码的简单性，如果是为了炫耀一些技巧而使得代码晦涩难懂则更是要不得。

Pascal 语言本身非常讲究程序的优雅，这可以在一定程度上防止程序员写出太过技巧性的代码。比如，用 C 语言就可以写出如下代码：

```
n = ++(++>(*p1++))+(**p2);
```

这样的代码除了在考试的情况下出现外，恐怕只有计算机能理解了。

但即使语言要求代码美观，有时还是无法阻止程序员干一些傻事，因此简单性原则还需有程序员的配合才能达到。

首先，使得每个函数、过程的代码简短。一般一个函数或过程的代码不宜超过 25~30 行，庞大的函数是难以被读懂并修改的。当发现无法阻止一个函数成为“巨人”的时候，就是重新考虑是否该从该函数中拆出一些东西的时候了。

其次，减少代码缩进。代码缩进的风格是为了使得代码逻辑关系清晰，但是，过深的缩进（一般指超过三层的缩进）却会让人迷惑。例如，要写一个打开文件的函数 LoadFile(FileName: String)。它的功能是：如果传给它的文件名为空字符串，则创建一个新文件；否则检查该文件是否存在、是否有效（是否有效的标准就看项目要求了），是则打开文件进行一些操作。该函数出现错误情况时返回 0，正常时返回 1。

看一下如下的函数实现：

```
function LoadFile(FileName : String) : Integer;  
begin  
    if FileName = '' then  
        begin  
            // 创建一个新的文件...  
            Result := 1;  
        end  
    else if FileExists(FileName) then  
        begin // 文件存在
```



```
if IsFileValid(FileName) then
begin //文件有效
    // 打开文件...
    Result := 1;
end
else
    Result := 0;
end
else
    Result := 0;
end;
```

无法说这些代码有什么错误，但是阅读起来就不是那么一目了然了。再看一下与之完成相同功能的一段代码：

```
function LoadFile(FileName : String) : Integer;
begin
    Result := 0;

    if FileName = '' then
    begin
        // 创建一个新的文件...
        Result := 1;
        Exit;
    end;

    if not FileExists(FileName) then
        Exit; // 文件不存在则返回 0

    if not IsFileValid(FileName) then
        Exit; // 文件无效则返回 0

    // 打开文件...
    Result := 1;
end;
```

相比之下，第 2 个函数实现更能让人一目了然，容易知道该函数所完成的功能。

最后，让自己的代码“傻瓜化”。尽量用人的语言来编写程序而不是用计算机的语言。例如：

```
var
```

```
n : Integer;  
b : Boolean;  
begin  
  if n >= 60 then  
    b := true  
  else  
    b := false;  
  // .....  
end;
```

虽然比

```
var  
  n : Integer;  
  b : Boolean;  
begin  
  b := (n >= 60);  
  // .....  
end;
```

看上去更罗嗦一些，但是，显然第一种写法更让人一目了然。

也许有些读者会觉得这样的程序不足以带给你强烈的成就感。但是，要知道，写出别人都看不懂的代码的程序员并非好的程序员，也没什么值得骄傲的，因为别人无法看懂的代码，过一段时间之后，自己也将会无法看懂。

代码简单化之后，还有一个好处就是可以省略许多的注释。程序员间经常有这样的误解，认为写注释多的程序员才是敬业的程序员。其实，好的代码是可以“自解释”的，代码本身已经说明了一切，根本不需要“画蛇添足”地加上许多的注释。好的代码就如同小说一样，通顺流畅。有谁见过每一段旁边都再加一段文字，说明这一段写了什么的小说吗（评点本除外）？当觉得有必要写下一大段注释时，应该考虑一下是否是自己的原因让代码难懂了。

关于注释，顺便提一下，当你写下过多的不必要的注释时，你可能是在侮辱以后会阅读你的代码的程序员智商。

#### ◆ 原则二：所有模块被以面向接口来实现

一个系统，可能由几十、几百个甚至几千个模块组成。如此众多的对象之间的关系是非常复杂的。每个对象犹如社会上的每个人，在为别人提供服务的同时，也在享用其他人提供的服务，因此对象之间的通信大多数情况下都是双向的——能接受消息，同时也能向其他对象发送消息。

需求的多变总会带来每个模块/类被不断地修改和维护，而如果每个模块/类被以面向接口来实现（即总是通过接口来访问其他模块/类提供的服务、功能），则可以将这种修改所



带来的影响局限到最小范围——真正与改变相关的部分。当然，这还需要具有一定的预期变化的能力。

### ◆ 原则三：将界面与功能模块分离

其实，该原则是原则二的特例。可以将 UI（用户界面）看作是一个与用户交互的大的模块，而非 UI 部分作为一个实现具体功能的大的模块。该原则可以解释为将这两个大的模块都以面向接口来实现，将其代码分离。

UI 一般包括菜单命令、状态显示等面向用户的交互处理。处理菜单命令时，可以将 UI 层作为功能层的客户，UI 层调用功能层提供的接口来完成菜单命令的响应。处理状态显示时，则可以将功能层作为 UI 层的客户，由功能层通知 UI 层更新状态。当然，也可以由 UI 层主动去获取状态信息。

如果读者正在用 Delphi 编写一个纯文本编辑器，而且不再满足于使用简单的 TMemO 来实现，但是目前自己手上又没有更好的编辑器组件。自己写一个？这很耗费精力和时间，可以估算一下，结论是不值得。因为你坚信已经存在有很棒的第三方的编辑器组件，只是现在没有发现而已。那就停下所有的工作去找组件？这不太现实。这时可能会想到，暂且使用 TMemO 也无妨，等以后找到新的符合要求的组件再更换。于是，便会考虑如何将未来更换组件的代价降到最低，就会想到将编辑器功能与 UI 分离开来。

编辑器最基本的功能无非是打开文件、保存文本以及对其他一些编辑操作的支持。所有的编辑器组件的核心功能都是相似的，从而也就带来了机会。

可以为所有的编辑器设计一个公共接口：

```
TEditor = class
public
    function Load(FileName : String) : Boolean;
    function Save() : Boolean;
    function SaveAs(FileName : String) : Boolean;
    // ... 其他需要的操作，由需求而定
end;
```

每个具体的编辑器的实现（如使用 TMemO），都从 TEditor 派生，然后声明一个 TEditor 类的全局对象：

```
var
    g_Editor : TEditor;
```

而在打开文件菜单命令响应中，就会有类似这样的代码：

```
procedure TForm1.menu_openClick(Sender: TObject);
begin
    if dlg_open.Execute() then // 弹出打开文件对话框
```

```
g_Editor.Load(dlg_open.FileName);  
end;
```

保存文件的菜单命令响应中，代码可能是这样的：

```
procedure TForm1.menu_saveClick(Sender: TObject);  
begin  
    g_Editor.Save();  
end;
```

而当更换编辑器组件时，只是从 `TEditor` 类再派生并实现一个子类而已，界面层的代码并不需要做任何改变。

#### ◆ 原则四：善于使用多态给程序带来灵活性

多态性，是一种能给程序带来灵活性的东西。但要真正获取这种灵活性，需要良好的设计。

继续“原则三”中提到的那个纯文本编辑器的例子。在定义了 `TEditor` 类的接口后，又如何实现一个使用 `TMemo` 的编辑器组件呢？以后更换编辑器组件时，如何将添加代码的量减到最少呢？

前文粗略地给出了一个 `TEditor` 的声明，现在就来仔细地设计它。为了简化问题，在此只讨论读取文件、保存文本操作的实现，其余的操作都类似。先看一下类的声明：

```
TEditor = class // 抽象基类  
private  
    FFileName : String;  
protected  
    function DoLoad(FileName : String) : Boolean; virtual; abstract;  
public  
    function Load(FileName : String) : Boolean;  
    function Save() : Boolean;  
    function SaveAs(FileName : String) : Boolean; virtual; abstract;  
    // ... 其他需要的操作，由需求而定  
end;
```

也许有人会惊讶于这样的声明，先来解释一下。

首先，其中有一个字符串类型的 `private` 数据成员——`FFileName`，其作用是保存打开的文件名称。

其次，有 3 个 `public` 的方法：`Load`——读取文件；`Save`——保存当前文本；`SaveAs`——以新的文件名称保存当前文本。但是它们的声明又有些不同，`Load` 和 `Save` 为非虚方法，而 `SaveAs` 却是抽象虚方法。为什么呢？



SaveAs 的实现，完全依赖于不同的编辑器组件，基类本身是不知道如何保存文本的，因此将其声明为抽象虚方法（virtual; abstract;），由具体的派生类去实现。

Save 的实现，完全依赖于 SaveAs，Save 的算法非常简单。只需要调用 SaveAs() 并将 FFileName 的值传递给它就行了。基类完全知道该如何做，因此声明为非虚方法。

Load 的实现有些特别。Load 的算法是首先读取文件到编辑器控件，然后将 FFileName 的值设为所读取的文件名。第 1 个步骤的实现与 SaveAs 性质相似，依赖于不同的编辑器组件，应该由派生类去实现它。而第 2 个步骤派生类却无法实现，因为 FFileName 为 private 属性，派生类是看不到的。但也不能因此而将 FFileName 移到 protected 节中，以使派生类可以访问它。因为如果这样的话，每个派生类都要记住在打开文件后去设置一个基类的属性。这不仅造成代码重复（每个派生类都必须这样做），而且会造成抽象的混乱（FFileName 在逻辑上是属于 TEditor 的），所以设置 FFileName 不应该是派生类的义务。那么第 2 个步骤很明显应该在基类中被实现。如何做到呢？

最后可以看到，有一个 protected 的抽象虚方法——DoLoad，它正是配合 Load 完成读取文件的算法。由派生类的 DoLoad 实现具体的每个编辑器组件读取文件，Load() 负责调用 DoLoad 完成第 1 个步骤，然后设置 FFileName 完成第 2 个步骤。

具体来看一下基类 TEditor 的实现：

```
function TEditor.Load(FileName : String) : Boolean;
begin
    Result := DoLoad(FileName); // 步骤一
    if Result then
        FFileName := FileName; // 步骤二
end;

function TEditor.Save() : Boolean;
begin
    SaveAs(FFileName); // 调用抽象的 SaveAs
end;
```

基类实现完成后，就可以从它派生，具体实现编辑器组件相关的子类了。假设现在用 TMemo 来实现，则声明 TMemEditor 如下：

```
TMemEditor = class(TEditor)
private
    FEditor : TMemo;
protected
    // 覆盖 DoLoad, 给出具体实现
    function DoLoad(FileName : String) : Boolean; override;
public
```

```
constructor Create();
destructor Destroy(); override;

// 覆盖 SaveAs, 给出具体实现
function SaveAs(FileName : String) : Boolean; override;
// ...其他需要的操作
end;
```

在该派生类中, private 成员 FEditor 保存了一份 TMemo 组件的实例, 也就是具体的编辑器。另外, 还覆盖了基类中的抽象方法 DoLoad 和 SaveAs。TMemoEditor 的具体实现如下:

```
function TMemoEditor.Create();
begin
    // 创建 TMemo 实例
    m_Editor := TMemo.Create(nil);

    // 接着完成将 TMemo 实例置于界面上显示出来等操作, 省略
end;

function TMemoEditor.Destroy();
begin
    // 诸如将 TMemo 实例从界面上删除等其他清理工作, 省略

    m_Editor.Free();
    m_Editor := nil;
end;

function TMemoEditor.DoLoad(FileName : String) : Boolean;
begin
    Result := false;
    try
        m_Editor.LoadFromFile(FileName);
    except
    end;

    Result := true;
end;

function TMemoEditor.SaveAs(FileName : String) : Boolean;
begin
    Result := false;
```



```
try
    m_Editor.SaveToFile(FileName);
except
end;
Result := true;
end;
```

有了这些实现，编辑器便可以正常工作了。当找到更好的编辑器组件时，你所要做的，就是从 TEditor 再派生一个具体类，实现一个 TXXXEditor，其他部分的代码不用做任何改动。而且，TXXXEditor 的具体实现，也只和组件本身的特性相关（如读取、保存文件的方法）。公共逻辑已经在 TEditor 类中实现了。

因此，善于使用多态可以给程序带来莫大的灵活性和可重用性。基于笔者个人的认识，给出几条经验：

（1）如果基类不知道如何实现某方法（只有派生类知道），而基类的其他方法又必须使用该方法，则把该方法声明为抽象虚方法——`virtual; abstract;`。

（2）如果基类能够为某方法提供一种默认实现，但派生类可能完全重写这个实现，则将该方法声明为虚方法——`virtual;`，并实现默认算法。

（3）如果基类能够且必须提供某方法的部分的实现，而派生类必须提供另一部分的实现，则将该方法声明为非虚方法，并在基类中为其配套提供一个虚方法或抽象虚方法，以允许由基类本身调用和被派生类所覆盖，如上例中的 Load 与 DoLoad。

## 6.3 小 结

请相信，优美的代码足以使人延年益寿！

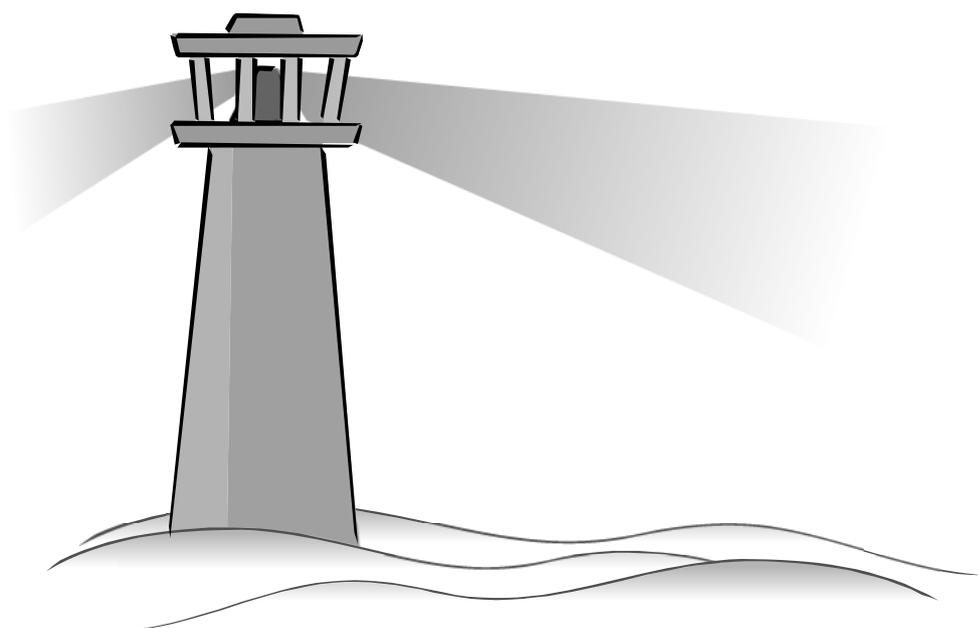
编写良好设计的代码，可以给自己和他人带来维护上的便利，给公司带来降低成本的便利。

请接纳 OOP 所带来的礼物吧，不用去感谢它，而只需要去学习、掌握、运用它……

# 第 7 章 代码设计实例

——Sunny SmartNote

实例是最好的教材，我愿与你分享它——作者



**寻找自己的方向!**



行文至此，笔者感到需要发生一些改变。你在阅读了本书的前 6 章、学习了相关的基础知识之后，最需要的是什么呢？

不知道各位的答案是否和我所想的一样：一个实例。如果是，我很高兴，在此将满足你的需求，说不定比你所想要的更多。

本章将给出笔者实际正在编写的一个作品——Sunny SmartNote 5.0 的开放源代码的版本的代码。Sunny SmartNote 是一个多工作区的文本编辑器，在开放源代码版本中舍弃了一些辅助功能。

要说明的是，在本章讲解实例时，会尝试采用一种比较特别的方式。由于编程的学习主要是对其一种方法的学习，而面向对象更是一种方法论，因此在解说实例时，会着重讲解思考的方式、步骤以及决策的原因，而不是一味地解释代码。笔者会尽力将进行代码设计时所想的用文字表达出来，以便于读者参考其中的方法。不过，由于思维性的东西始终是比较抽象的，学习起来终究有些困难，所以感到困惑时，请不要气馁。

在这一章里会具体介绍 Sunny SmartNote 程序的构架设计以及实现。作为实例，应该是具有示范作用的。但是，只能说，对于每个 project，优良的设计方案可能并非只有一种，而且不见得这里给出的就是最好的。注意，我所给出的，只是实现方案的所有的可能中的一种——被我所实践的那一种而已。

另外，本章中代码会比较多。文字叙述中基本不会提及功能实现上的问题，而只讨论设计方法及原因。如果对某部分的功能实现更感兴趣，可以查看代码。代码中一般都会加上注释，这些注释对于理解实现可能是非常有用的。

## 7.1 需求概述与代码风格说明

### 7.1.1 需求概述

Sunny SmartNote 是一个多工作区的文本编辑器。所谓工作区，即一个编辑环境。一个工作区，对应于一个文本编辑区域。多工作区即意味着可以同时打开多个文件进行编辑。不过，它和 MDI（多文档界面）的概念不同，MDI 是多工作区的一种界面风格或一种实现，而多工作区则不见得一定是 MDI，如可以使用页组件（PageControl）来实现多工作区。

由于文本编辑器这个需求非常普遍，要理解这个需求并非难事。因此只在此说明一下要实现的功能，这种需求从客户的角度出发，与实际的实现无关，它表述的概念也只出现在客户逻辑中。

首先，是最基本的文件操作。由于 Sunny SmartNote 工作于“工作区”的概念之下，而每个文件的操作基本上都对应于对工作区的操作，因此将功能描述为“工作区操作相关功能”：新建一个空工作区、保存工作区中的文本到文件、读取文件中的文本到工作区、关闭工作区、工作区的切换。其中还包括了对所有工作区的类似操作，如保存所有工作区、关闭所有工作区。

其次，是普遍的编辑操作。在此，将功能描述为“单个工作区的编辑操作相关功能”，

包括：撤销/重做、剪切、复制、粘贴、删除（包括删除选中文本和删除一行）、全选、查找、替换、自动换行设置、统计字数。

最后，是辅助功能。功能描述为“辅助功能”，允许用户设置一些选项，如工作区中的字体等。

以上是从客户的角度描述的需求。接下来说明功能实现上的需求，即从开发者角度描述的需求。

作为纯文本编辑器最核心的部分——编辑器组件，是整个软件的灵魂。因此，选用的组件的好坏会直接影响到软件本身。最基本的可以使用 TMemo，但它的功能实在太弱。为了先实现其他部分的功能，可以允许先用 TMemo，等找到更好的编辑器组件或以后自己再实现一个也可以。这就要求编辑器组件是允许被替换的。

工作区的风格也是多种多样的，必须允许被替换。

GUI 的风格，如各种对话框，必须允许被替换。

这些是对程序灵活性的要求。

### 7.1.2 代码风格说明

类名。接口以 Issn 开头，一般类名以 Tssn 开头。此处所说的接口，并非 Object Pascal 中的 Interface，而是一个纯抽象基类，即该类不提供任何实现，而只声明一套操作规范。在语言中仍以 class 声明。

数据成员命名。数据成员均以 m\_ 作为前缀，虽然 VCL 规范以 F 作为数据域成员的前缀，但笔者认为 m\_ 能更清晰地表示其身份。

变量命名。全局变量均以 g\_ 作为前缀，单元局部变量以 l\_ 作为前缀。

缩进。代码中，所有手工代码缩进以 4 个空格符作为缩进，而自动生成的代码则不作改变。

begin 与 end。笔者非常喜欢将 begin 与 end 单独成行，即使是 if...else...的情况下，也会写为：

```
if X then
begin
    // some code
end
else
begin
    // some code
end;
```

### 7.1.3 图的说明

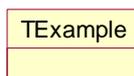
在以下的讲述中，会出现多个与设计相关的图例以说明系统中每个模块/类之间的关



系。现在，有必要先讲解以下图中出现的各个符号的意义。

1. 类以一个矩形表示，其中被分为上、下两个部分，上半部分为类名，下半部分为类的方法和属性（如果需要列出来的话）。如果类是抽象类，则类名以斜体字显示。

示例：



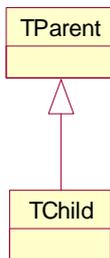
2. 子系统是整体系统中的一个模块，一般由协同完成一种类型任务的多个类组成。以一个带标签的矩形表示。

示例：



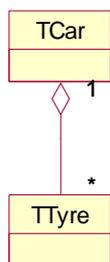
2. 继承（派生）关系指明一个类从另一个类派生。以一条实线加一个单向的空心三角形箭头表示，箭头从派生类指向基类。

示例：



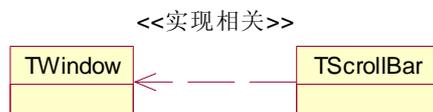
3. 组合（包含）关系指明在一个类中包含另一个类的实例。以一条实线加一个菱形表示，菱形端为包含端（即表示整体的类），另一端表示被包含端（即表示部分的类）。如果线的两端有数字，则表示两个类的对应数量（符号“\*”表示多个）。

示例：



4. 关联（相关）关系指明两个类的实现是相互关联的。以一条虚线加一个线形箭头表示，箭头指向被关联类。

示例：



## 7.2 IssnEditor/TssnEditor/TssnWorkSpace

7

根据需求，必须允许所使用的编辑器组件随时可替换。

由于所使用的各种编辑器组件本身都具有自己的接口，而且可能其差异非常大（不可能要求所有编辑器组件的作者都遵循一种接口）。而如果想达到更改组件而不更改其他不相关部分的代码的目的，就需要为编辑器组件和应用程序的其他部分之间提供一个隔离层。隔离层之外，按照本身应用程序的需求，设计一个统一的接口。应用程序的其他部分代码，都使用这个统一的接口对编辑器进行访问。而隔离层之内的具体的编辑器组件接口虽然千变万化，但隔离层可以将它们的接口转换到统一的接口。也就是说，该隔离层其实就是所谓的“接口转换器”。这种转换器的设计，是设计模式的一种，被称为“adapter”模式。

为应用程序设计这个统一的接口，就是平时所说的抽象！

因此首先需要为编辑器组件抽象。

于是，需要一个抽象类 `TssnEditor` 以抽象出该应用程序需求中所需要的所有编辑器组件的特征——行为和属性，然后从它派生出依赖于不同编辑器组件的具体实现的子类。

接着，就要抽象出“工作区”的概念。`TssnEditor` 所表示的编辑器组件并没有出现在客户逻辑中，也就是说客户端只知道“工作区”的概念，而并不知道有所谓的“编辑器组件”存在。比如，保存一个文件的客户逻辑是，向“工作区”发出“保存”的命令，而不是向“编辑器组件”。但实际程序的实现上，是由“编辑器组件”去执行“保存”命令。因此，`TssnEditor` 的实例需要一个符合客户逻辑的容器以作为代理，而“工作区”正是扮演这个“代理”的角色。

于是又需要一个抽象类 `TssnWorkSpace`。其内部包含并管理，维护一个 `TssnEditor` 实例，其外部客户端（如界面部分）不需要了解、感知 `TssnEditor` 的存在。外部所有对编辑器的操作请求，都通过 `TssnWorkSpace` 转达给其内部所包含的 `TssnEditor`。另外，根据需求，工作区的实现风格也是可变的，如可以用 MDI 风格的界面实现，也可以使用 `PageControl` 组件分页来实现。因此，`TssnWorkSpace` 同时将作为另一个“接口转换器”，它为应用程序的其他部分提供一个对“工作区”操作的统一的接口，而各种风格的工作区的具体实现都从 `TssnWorkSpace` 派生。

接下来，既然 `TssnWorkSpace` 代理了 `TssnEditor` 的功能，为了保证发送给 `TssnWorkSpace` 的与编辑器组件相关的请求，都能够正确地转送给 `TssnEditor` 处理，它们必须拥有一个相同的功能子集。在这个集合中的所有操作都可以通过 `TssnWorkSpace` 转发给 `TssnEditor`。



于是需要为 `TssnWorkSpace` 和 `TssnEditor` 指定一个共同的基类——`IssnEditor`，以便它们都实现一个相同的操作集合。

`IssnEditor` 是一个纯抽象基类，它声明且只声明一套编辑器的基本操作与状态获取方法，不含任何实现。`TssnEditor` 与 `TssnWorkSpace` 都从它派生，继承其声明的接口。

`IssnEditor/TssnEditor/TssnWorkSpace` 三者的关系如图 7.1 所示。

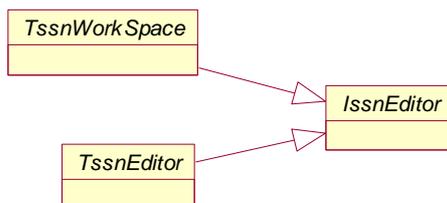


图 7.1 `IssnEditor/TssnWorkSpace/TssnEditor` 关系图

## 7.2.1 `IssnEditor`

`IssnEditor` 是一个接口，它规定了它的派生必须实现哪些功能。当 `TssnWorkSpace` 和 `TssnEditor` 都从它派生时，`TssnWorkSpace` 就会有绝对的“自信”将它接到的请求转发给其内部的 `TssnEditor` 实例。

`IssnEditor` 声明的功能集合是对于“工作区”而言的。例如，编辑器组件通常有 `LoadFromFile` 之类的方法，以允许将一个存在于磁盘文件中的文本读取到编辑器中。那么在设计 `IssnEditor` 的接口时，是否应该将 `LoadFromFile` 加入其中呢？正如前面所说，考虑的出发点应该是基于“工作区”——`TssnWorkSpace`，而不是编辑器组件——`TssnEditor`。我们应该问：“工作区功能集合中有没有必要加入 `LoadFromFile`？”。然后可以考虑，工作区是因为打开一个文件而被创建，已经存在的工作区是不可能重新 `Load` 一个文件的，因为当用户再次选择“打开文件”菜单命令时，`Sunny SmartNote` 将会建立一个新的工作区而并不会在旧有工作区中去读取文本。由此可以下结论：“工作区”是没有 `LoadFromFile` 操作的，它读取文本是在本身被构造时就完成了的，因此 `IssnEditor` 接口也不需要 `LoadFromFile`。

现在就可以根据需求来给出 `IssnEditor` 的声明：

```
IssnEditor = class
public
    function GetFileName() : String; virtual; abstract;
    function GetSaved() : Boolean; virtual; abstract;
    function Save() : Boolean; virtual; abstract;
    function SaveAs() : Boolean; virtual; abstract;
    function GetSelectText() : String; virtual; abstract;
    procedure SetFont(Font : TFont); virtual; abstract;
```

```

procedure Undo(); virtual; abstract;
function CanUndo() : Boolean; virtual; abstract;
procedure Redo(); virtual; abstract;
function CanRedo() : Boolean; virtual; abstract;
procedure Cut(); virtual; abstract;
function CanCut() : Boolean; virtual; abstract;
procedure Copy(); virtual; abstract;
function CanCopy() : Boolean; virtual; abstract;
procedure Paste(); virtual; abstract;
function CanPaste() : Boolean; virtual; abstract;
procedure DeleteSelection(); virtual; abstract;
function CanDeleteSelection() : Boolean; virtual; abstract;
procedure DeleteLine(); virtual; abstract;
procedure SelectAll(); virtual; abstract;
function FindNext(Text : String; Option : TFindOptions) : Boolean;
    virtual; abstract;
function Replace(FindText, ReplaceText : String; Option :
TFindOptions):
    Integer; virtual; abstract;
function GetWordCount() : TssnWordCountRec; virtual; abstract;
function GetWordWrap() : Boolean; virtual; abstract;
procedure SetWordWrap(WordWrap : Boolean); virtual; abstract;
end;

```

这些接口方法所规定的含义，如表 7.1 所示。

表 7.1 接口方法所规定的含义

方法名称	含 义
CanCopy	工作区是否处于允许执行“复制”操作状态
CanCut	工作区是否处于允许执行“剪切”操作状态
CanDeleteSelection	工作区是否处于允许执行“删除所选字符”操作状态
CanPaste	工作区是否处于允许执行“粘贴”操作状态
CanRedo	工作区是否处于允许执行“重做”操作状态
CanUndo	工作区是否处于允许执行“撤销”操作状态
GetWordCount	统计当前工作区字数
GetWordWrap	获取工作区是否处于“自动换行”状态
GetSaved	获取工作区是否处于“已保存”状态
GetSelectText	获取工作区中正被选中的字符串
GetFileName	获取当前工作区所对应的文件名
Copy	执行“复制”操作
Cut	执行“剪切”操作



续表

方法名称	含 义
DeleteLine	执行“删除一行”操作
DeleteSelection	执行“删除所选字符”操作
Paste	执行“粘贴”操作
Redo	执行“重做”操作
Undo	执行“撤销”操作
SelectAll	执行“全选”操作
SetFont	设置工作区显示字体
SetWordWrap	设置是否“自动换行”
FindNext	查找下一个
Replace	替换
Save	以当前文件名保存工作区内的文本到文件
SaveAs	另存为

## 7.2.2 TssnEditor

TssnEditor 的职责是为所有的编辑器组件进行抽象，并完成它们的公共代码。

首先，考虑一下 TssnEditor 本身的逻辑。编辑器的操作无非是两大类：文件访问（读取和保存）和文字编辑。其中，文字编辑一般都与具体的编辑器组件密切相关。

因此，先来说一下文件访问。文件访问要求在 TssnEditor 内部保存有该组件所对应的文件名称，可以定义一个 private 的 m\_FileName（字符串类型），在读取文件时设置该成员的值，在“另存为”的时候改变该成员的值，在保存的时候直接将编辑器组件中的文本保存到 m\_FileName 所指的文件。好，现在的 TssnEditor 可能是这样的：

```
TssnEditor = class(IssnEditor)
private
    m_FileName : String;
end;
```

当然，这只是最初的状态，稍后会继续完善它。

有了 m\_FileName，然后就必须为 TssnEditor 加上必要的 LoadFromFile 方法，虽然对于“工作区”而言，不需要它，但是对于编辑器组件来说，它却是至关重要的。

LoadFromFile 的算法是将 m\_FileName 设置为所要打开的文件名，然后打开文件。其中为 m\_FileName 设置值是派生类无法做到的，需要在 TssnEditor 中实现。而实际打开文件的动作与具体的编辑器组件有关，必须延迟到派生类中实现。这时，virtual 就可以帮上忙了。可以定义一个非虚方法 LoadFromFile 并声明一个抽象虚方法 DoLoadFromFile，由派生类所实现的 DoLoadFromFile 实际地将文件读取到编辑器组件中，而 LoadFromFile 负责调

用 DoLoadFromFile 并设置 m\_FileName 的值。

为 TssnEditor 加上 LoadFromFile()和 DoLoadFromFile():

```
TssnEditor = class(IssnEditor)
private
    m_FileName : String;
protected
    procedure DoLoadFromFile(FileName : String); virtual; abstract;
public
    procedure LoadFromFile(FileName : String);
end;
```

之所以将 DoLoadFromFile()置于 protected 节而将 LoadFromFile()置于 public 节，是因为对于 TssnEditor 的外部来说，LoadFromFile()是一个接口，是 TssnEditor 提供的一个功能，而 DoLoadFromFile()只是一个实现细节。对于 TssnEditor 的派生类来说，需要它们来实现 DoLoadFromFile()这个抽象虚方法。DoLoadFromFile()是 TssnEditor 为其派生类规定的一个接口。

LoadFromFile()方法的代码如下：

```
procedure TssnEditor.LoadFromFile(FileName: String);
begin
    m_FileName := FileName;

    // 实际读取文件到编辑器的动作，延迟到派生类中实现
    // 因此 DoLoadFromFile 为 virtual; abstract;的
    // 如果 FileName = '', 表示新建一个编辑器组件，不打开任何文件
    if FileName <> '' then
        DoLoadFromFile(FileName);
end;
```

这种做法也是设计模式的一种，被称为“Template Method”，它的应用非常普遍。在 VCL 中，TObject 非虚的 Free()方法和虚方法 Destroy()之间的关系，就如同 LoadFromFile()和 DoLoadFromFile()的关系一样。

文件访问，除了读取文件，显然还有保存文件。IssnEditor 接口已经给出了两个方法 Save()和 SaveAs()用于保存功能，在 TssnEditor 中可以实现这两个方法。

SaveAs()的算法是显示一个“另存为”的对话框，如果指定了正确的文件名，则保存文本到文件。与 LoadFromFile 类似，只有真正保存文件的动作才与实际编辑器组件相关，可以另外声明一个 SaveToFile 的抽象虚方法并由派生类实现具体的保存文本的动作，而弹出“另存为”对话框则属于公共逻辑，可以在 TssnEditor.SaveAs()中实现。

现在再为 TssnEditor 加上 SaveAs()、Save()以及抽象虚方法 SaveToFile()的声明：



```
TssnEditor = class(IssnEditor)
private
    m_FileName : String;

protected
    procedure DoLoadFromFile(FileName : String); virtual; abstract;
public
    procedure LoadFromFile(FileName : String);
    procedure SaveToFile(FileName : String); virtual; abstract;
    function Save() : Boolean; override;
    function SaveAs() : Boolean; override;
end;
```

SaveAs()方法的具体实现:

```
function TssnEditor.SaveAs: Boolean;
var
    FileName : String;
begin
    Result := false;
    // 弹出“另存为”对话框, 得到要保存的文件名
    // 关于 g_InterActive, 将在 7.6 节详述
    // 在此可以认为它的功能是弹出“另存为”对话框并返回文件名
    FileName := g_InterActive.ShowSaveDlg();
    if FileName = '' then
        Exit;
    // SaveToFile 的实现与具体编辑器组件相关, 由派生类实现
    SaveToFile(FileName);
    m_FileName := FileName;
    Result := true;
end;
```

可以发现, 这同样是一个 Template Method 模式的应用。

Save()的算法是检查 m\_FileName 是否为空字符串, 是(未命名文件)则调用 SaveAs(), 否则调用 SaveToFile()保存文本:

```
function TssnEditor.Save: Boolean;
begin
    if m_FileName = '' then
        begin
            Result := SaveAs(); // 文件未命名时, 调用 SaveAs
```

```

        Exit;
    end;

    // SaveToFile 的实现与具体编辑器组件相关，由派生类实现
    SaveToFile(m_FileName);
    Result := true;
end;

```

好了，TssnEditor 的文件访问功能已经实现完成（除了应该由派生类做的，如 DoLoadFromFile 和 SaveToFile），剩下的文字编辑功能基本上都与具体编辑器组件相关，因此应由 TssnEditor 的派生类实现。不过，其中的统计字数功能，由于在此准备对所有编辑器采用一致的算法进行字数统计，因此决定将该功能在 TssnEditor 中实现。统计字数的算法是首先取得编辑器中的所有文本（字符串形式），然后逐字符进行统计。也许读者也已经想到，取得编辑器中的所有文本字符串需要由具体编辑器组件（TssnEditor 的派生类）提供，而统计步骤则在 TssnEditor 中实现，这又是一个 Template Method 模式！

于是，会为 TssnEditor 加上抽象虚方法 GetText() 和非虚方法 GetWordCount()：

```

TssnEditor = class(IssnEditor)
private
    m_FileName : String;
protected
    procedure DoLoadFromFile(FileName : String); virtual; abstract;
    function GetText() : String; virtual; abstract;
public
    procedure LoadFromFile(FileName : String);
    procedure SaveToFile(FileName : String); virtual; abstract;
    function Save() : Boolean; override;
    function SaveAs() : Boolean; override;
    function GetWordCount() : TssnWordCountRec; override;
end;

```

GetWordCount 先调用 GetText() 以取得文本字符串，然后进行逐字符统计：

```

function TssnEditor.GetWordCount: TssnWordCountRec;
var
    AllText : String;
    bHalf : Boolean;
    i : Integer;
    nAsc : Integer;
begin
    AllText := GetText(); // 取得文本

```



```
bHalf := false;

for i := 1 to Length(AllText) do
begin
    nAsc := ord(AllText[i]); // 取每个字符的 ASCII 代码

    if bHalf then // 如果处于双字节字符的第二个字符，则忽略该字符
    begin
        bHalf := false;
        nAsc := 0;
    end;

    // 如果当前字节 ASCII 代码>127，则认为其为双字节字符
    if nAsc > 127 then
    begin
        if not bHalf then
            Inc(Result.MultiChar);
        bHalf := not bHalf;
    end

    else if (nAsc >= 48) and (nAsc <= 57) then
        Inc(Result.NumChar) // 数字字符
    else if (nAsc >= 65) and (nAsc <= 90) then
        Inc(Result.AnsiChar) // 英文字母字符
    else if (nAsc >= 97) and (nAsc <= 122) then
        Inc(Result.AnsiChar) // 英文字母字符
    else if (nAsc <> 32) and (nAsc <> 0) and
        (nAsc <> 13) and (nAsc <> 10) then
        Inc(Result.Other); // 其他字符
    end;
end;
```

最后，重新审视一遍 `TssnEditor` 接口所声明的所有方法，显然能够发现，由于 `m_FileName` 是 `private` 的，因此 `GetFileName()` 方法（取得工作区/编辑器组件对应的文件名）应该也只能在 `TssnEditor` 中实现：

```
function TssnEditor.GetFileName: String;
begin
    Result := m_FileName; // 直接将 m_FileName 返回
end;
```

至此，TssnEditor 应该是这样的：

```
TssnEditor = class(IssnEditor)
private
    m_FileName : String;
protected
    procedure DoLoadFromFile(FileName : String); virtual; abstract;
    function GetText() : String; virtual; abstract;
public
    procedure LoadFromFile(FileName : String);
    procedure SaveToFile(FileName : String); virtual; abstract;
    function GetFileName() : String; override;
    function Save() : Boolean; override;
    function SaveAs() : Boolean; override;
    function GetWordCount() : TssnWordCountRec; override;
end;
```



TssnEditor 实现于 Editor.pas 中，该单元的代码清单如下：

```
unit Editor;

interface

uses Controls, Graphics,
    IntfEditor;

type
    TssnEditor = class(IssnEditor)
    private
        m_FileName : String;

    protected
        procedure DoLoadFromFile(FileName : String); virtual; abstract;
        procedure OnEditorSelectionChange(Sender : TObject);
        function GetText() : String; virtual; abstract;

    public
        procedure LoadFromFile(FileName : String);
        procedure SaveToFile(FileName : String); virtual; abstract;
        function Save() : Boolean; override;
        function SaveAs() : Boolean; override;
        function GetWordCount() : TssnWordCountRec; override;
```



```
function GetFileName() : String; override;
end;

implementation

uses GlobalObject;

{ TssnEditor }

function TssnEditor.GetFileName: String;
begin
    Result := m_FileName;
end;

function TssnEditor.GetWordCount: TssnWordCountRec;
var
    AllText : String;
    bHalf : Boolean;
    i : Integer;
    nAsc : Integer;
begin
    AllText := GetText();
    bHalf := false;

    for i := 1 to Length(AllText) do
    begin
        nAsc := ord(AllText[i]);

        if bHalf then
        begin
            bHalf := false;
            nAsc := 0;
        end;

        if nAsc > 127 then
        begin //chinese
            if not bHalf then
                Inc(Result.MultiChar);
            bHalf := not bHalf;
        end

        else if (nAsc >= 48) and (nAsc <= 57) then
```

```
        Inc(Result.NumChar)

    else if (nAsc >= 65) and (nAsc <= 90) then
        Inc(Result.AnsiChar)

    else if (nAsc >= 97) and (nAsc <= 122) then
        Inc(Result.AnsiChar)

    else if (nAsc <> 32) and (nAsc <> 0) and
        (nAsc <> 13) and (nAsc <> 10) then
        Inc(Result.Other);
    end;
end;

procedure TssnEditor.LoadFromFile(FileName: String);
begin
    m_FileName := FileName;
    if FileName <> '' then
        DoLoadFromFile(FileName);
end;

procedure TssnEditor.OnEditorSelectionChange(Sender: TObject);
begin
    g_EditorEvent.OnEditorSelectionChange(Sender);
end;

function TssnEditor.Save: Boolean;
begin
    if m_FileName = '' then
    begin
        Result := SaveAs();
        Exit;
    end;

    SaveToFile(m_FileName);
    Result := true;
end;

function TssnEditor.SaveAs: Boolean;
var
    FileName : String;
begin
```



```
Result := false;
FileName := g_InterActive.ShowSaveDlg();
if FileName = '' then
    Exit;
SaveToFile(FileName);
m_FileName := FileName;
Result := true;
end;
end.
```

代码清单中，TssnEditor 多出一个 OnEditorSelectionChange()方法的声明与实现：

```
procedure TssnEditor.OnEditorSelectionChange(Sender: TObject);
begin
    g_EditorEvent.OnEditorSelectionChange(Sender);
end;
```

该方法与整个软件框架中的时间委托模型相关，g\_EditorEvent 对象以及更具体的内容将在 7.5 节中详述。

### 7.2.3 TssnMemoEditor

随着 TssnEditor 的实现完成，现在可能急于想实现一个具体的编辑器组件以让它工作起来，也可以暂时满足一下我们的成就感。好，先用最简单的 TMemo 组件来实现一个 TssnEditor，它就是 TssnMemoEditor，将实现 TssnEditor 接口中 TssnEditor 未实现的部分以及 TssnEditor 声明的抽象虚方法。由于 TssnMemoEditor 已经属于 Final 类型的类，即处于最底层的类，不会再有其他类从它派生。因此，它基本上都是实现功能的基本代码。来看一下它的声明：

```
TssnMemoEditor = class(TssnEditor)
private
    // 内部包含一个 TMemo 的实例，TssnMemoEditor 只是对它进行接口改造
    m_Edit : TMemo;

protected
    // TssnEditor 声明的抽象虚方法
    procedure DoLoadFromFile(FileName : String); override;
    function GetText() : String; override;

public
```

```
constructor Create(ParentCtrl : TWinControl);
destructor Destroy(); override;

// TssnEditor 声明的抽象虚方法
procedure SaveToFile(FileName: String); override;

// TssnEditor 声明的, 但 TssnEditor 未实现的
function GetSaved() : Boolean; override;
function GetSelectText() : String; override;
procedure SetFont(Font : TFont); override;
procedure Undo(); override;
function CanUndo() : Boolean; override;
procedure Redo(); override;
function CanRedo() : Boolean; override;
procedure Cut(); override;
function CanCut() : Boolean; override;
procedure Copy(); override;
function CanCopy() : Boolean; override;
procedure Paste(); override;
function CanPaste() : Boolean; override;
procedure DeleteSelection(); override;
function CanDeleteSelection() : Boolean; override;
procedure DeleteLine(); override;
procedure SelectAll(); override;
function FindNext(Text : String; Option : TFindOptions) : Boolean;
    override;
function Replace(FindText, ReplaceText : String; Option :
TFindOptions):
    Integer; override;
function GetWordWrap() : Boolean; override;
procedure SetWordWrap(WordWrap : Boolean); override;
end;
```

TssnMemoEditor 从 TssnEditor 派生, 内部拥有一个 TMemo 的实例——m\_Edit, TssnMemoEditor 的实现全部依赖于其内部的 TMemo。之前已经说过, TssnMemoEditor 其实就是一个将 TMemo 的接口转换成 TssnEditor 的接口的“接口转换器”。

在构造函数中创建 m\_Edit 实例:

```
constructor TssnMemoEditor.Create(ParentCtrl: TWinControl);
begin
    // ParentCtrl 参数为该 TMemo 组件实例所在容器
```



```
m_Edit := TMemo.Create(nil);
m_Edit.Parent := ParentCtrl; // 指定容器
m_Edit.Align := alClient;
m_Edit.Visible := true;
m_Edit.WordWrap := false;
m_Edit.ScrollBars := ssBoth;

// 设置 m_Edit 的事件处理函数指针, 相关内容将在 7.5 节详述, 此处可以忽略
// m_Edit.OnMouseUp := OnMouseUp; // 将在 7.5 节详述, 暂且注释这些代码
// m_Edit.OnKeyUp := OnKeyUp; // 将在 7.5 节详述, 暂且注释这些代码

if m_Edit.CanFocus() then
    m_Edit.SetFocus();
end;
```

在析构函数中销毁 m\_Edit:

```
destructor TssnMemoEditor.Destroy;
begin
    m_Edit.Free();
    m_Edit := nil;
end;
```

在介绍 TssnEditor 一节时曾经提到过, 编辑器读取文本的接口 LoadFromFile()和 DoLoadFromFile()这两个方法。DoLoadFromFile()是在 TssnEditor 中声明的抽象虚方法, 延迟到 TssnEditor 的派生类来实现。同样, 保存文件也是 Save()、SaveAs()和 SaveToFile()方法。其中 SaveToFile()为抽象虚方法, 延迟到 TssnEditor 的派生类来实现。还有配合统计字数功能实现的 GetText()方法, 延迟到 TssnEditor 的派生类来实现。现在所要完成的 TssnMemoEditor 就具有这样的职责, 必须给出 DoLoadFromFile()和 SaveToFile()的实现:

```
procedure TssnMemoEditor.DoLoadFromFile(FileName: String);
begin
    // 读取文本到编辑器
    m_Edit.Lines.LoadFromFile(FileName);
end;

procedure TssnMemoEditor.SaveToFile(FileName: String);
begin
    // 真正完成保存文件的动作
    m_Edit.Lines.SaveToFile(FileName);
end;
```

```
function TssnMemoEditor.GetText: String;
begin
    // 获取编辑器中所有文本
    Result := m_Edit.Lines.Text;
end;
```

最后，就是实现 `IssnEditor` 所声明的尚未被实现的方法。这些方法的实现，有的非常简单（如取得、设置编辑器的各种状态），有的稍微复杂一些（如查找、替换）。但由于使用了良好的构架，就可以成功地将复杂部分局限在某个有限的范围里，而不致于在修改这些部分时“牵一发而动全身”。

在此，给出所有方法的意义，列出实现稍微复杂一些的方法并简单介绍一下它们的算法。其余的简单方法，可以在本小节最后的该代码单元的清单中看到。

所有方法的含义如表 7.2 所示。

表 7.2 TssnMemoEditor 实现的方法与含义

方法名称	含 义
CanCopy	编辑器是否处于允许执行“复制”操作状态
CanCut	编辑器是否处于允许执行“剪切”操作状态
CanDeleteSelection	编辑器是否处于允许执行“删除所选字符”操作状态
CanPaste	编辑器是否处于允许执行“粘贴”操作状态
CanRedo	编辑器是否处于允许执行“重做”操作状态
CanUndo	编辑器是否处于允许执行“撤销”操作状态
GetWordWrap	获取编辑器是否处于“自动换行”状态
GetSaved	获取编辑器是否处于“已保存”状态
GetSelectText	获取编辑器中正被选中的字符串
Copy	执行“复制”操作
Cut	执行“剪切”操作
DeleteLine	执行“删除一行”操作
DeleteSelection	执行“删除所选字符”操作
Paste	执行“粘贴”操作
Redo	执行“重做”操作
Undo	执行“撤销”操作
SelectAll	执行“全选”操作
SetFont	设置编辑器显示字体
SetWordWrap	设置是否“自动换行”
FindNext	查找下一个
Replace	替换

由于 `TMemo` 对于查找、替换基本没有提供任何支持，因此查找、替换在实现上较为



复杂。

`FindNext()`的算法是：首先按照查找方向（向下或向上）取出查找范围内的所有字符串；然后，如果是非大小写敏感查找，则将字符串中所有字符转换成大写字符；最后根据查找方向搜索要查找的字符串，如果找到，则将光标定位到找到的字符串并返回 `true`，否则返回 `false`。`FindNext()`的实现代码如下：

```
function TssnMemoEditor.FindNext(Text: String; Option: TFindOptions) :
    Boolean;
var
    FoundAt : Integer;
    LastFoundAt : Integer;
    AllText : String;
begin
    // “查找下一个”算法
    Result := false;

    if frDown in Option then // 向下查找，找出搜索范围
        AllText := System.Copy(
            m_Edit.Text,
            m_Edit.SelStart + m_Edit.SelLength + 1,
            Length(m_Edit.Text)
        )
    else // 向上查找，找出搜索范围
        AllText := System.Copy(m_Edit.Text, 1, m_Edit.SelStart);

    if frMatchCase in Option then
    begin // 大小写敏感
        AllText := UpperCase(AllText);
        Text := UpperCase(Text);
    end;

    if frDown in Option then
    begin // 执行“向下查找”
        FoundAt := Pos(Text, AllText);
        if FoundAt = 0 then
            Exit;
        // 定位光标到指定的字符串
        m_Edit.SelStart := m_Edit.SelStart + m_Edit.SelLength + FoundAt;
    end
    else
    begin // 执行“向上查找”
```

```

LastFoundAt := 0;
repeat
  FoundAt := Pos(Text, AllText);
  if FoundAt <> 0 then
    begin
      AllText := System.Copy(
        AllText,
        FoundAt + 1,
        Length(AllText)
      );
      LastFoundAt := LastFoundAt + FoundAt;
    end
  until FoundAt = 0;
  if LastFoundAt = 0 then
    Exit;
  // 定位光标到找到的字符串
  m_Edit.SelStart := LastFoundAt;
end;

m_Edit.SelLength := Length(Text); // 选中找到的字符串
Result := true;
if m_Edit.CanFocus() then
  m_Edit.SetFocus();
end;

```

替换算法是：首先检查当前选中的是否为要被替换的字符串，如果是，则替换。然后判断是否是“替换全部”，是则循环调用 FindNext()查找到被替换字符串然后替换，直至循环退出（找不到下一个时），否则退出。其在 TssnMemoEditor 中的具体实现代码如下：

```

function TssnMemoEditor.Replace(FindText, ReplaceText: String;
  Option: TFindOptions): Integer;
var
  SelText : String;
begin
  // “替换”算法
  Result := 0;

  if frMatchCase in Option then
    begin // 大小写敏感
      SelText := UpperCase(m_Edit.SelText);
      FindText := UpperCase(FindText);
    end
end

```



```
else
    SelText := m_Edit.SelText;

// 若当前选择字符串与要替换的字符串相同，则替换
if FindText = SelText then
begin
    m_Edit.SelText := ReplaceText;
    Result := 1;
end;

if not (frReplaceAll in Option) then // 如果并非“替换所有”，则结束
    Exit;

// 查找下一个，并替换
while FindNext(FindText, Option) do
begin
    m_Edit.SelText := ReplaceText;
    Inc(Result);
end;
end;
```

TssnMemoEditor 实现在 MemoEditor.pas 单元中。整个单元的代码清单如下：

```
unit MemoEditor;

interface

uses StdCtrls, Controls, Graphics, Classes, Dialogs, SysUtils,
    Editor, IntfEditor;

type
    TssnMemoEditor = class(TssnEditor)
    private
        m_Edit : TMemo;

    protected
        procedure DoLoadFromFile(FileName : String); override;

        procedure OnMouseUp(Sender: TObject; Button: TMouseButton;
            Shift: TShiftState; X, Y: Integer);
        procedure OnKeyUp(Sender: TObject; var Key: Word; Shift:
            TShiftState);
```

```
function GetText() : String; override;
public
  constructor Create(ParentCtrl : TWinControl);
  destructor Destroy(); override;

  procedure SaveToFile(FileName: String); override;
  function GetSaved() : Boolean; override;
  function GetSelectText() : String; override;
  procedure SetFont(Font : TFont); override;
  procedure Undo(); override;
  function CanUndo() : Boolean; override;
  procedure Redo(); override;
  function CanRedo() : Boolean; override;
  procedure Cut(); override;
  function CanCut() : Boolean; override;
  procedure Copy(); override;
  function CanCopy() : Boolean; override;
  procedure Paste(); override;
  function CanPaste() : Boolean; override;
  procedure DeleteSelection(); override;
  function CanDeleteSelection() : Boolean; override;
  procedure DeleteLine(); override;
  procedure SelectAll(); override;
  function FindNext(Text : String; Option : TFindOptions) : Boolean;
    override;
  function Replace(FindText, ReplaceText : String; Option :
    TFindOptions) : Integer; override;
  function GetWordWrap() : Boolean; override;
  procedure SetWordWrap(WordWrap : Boolean); override;
end;

implementation

{ TssnMemoEditor }

function TssnMemoEditor.CanCopy: Boolean;
begin
  Result := m_Edit.SelLength <> 0;
end;

function TssnMemoEditor.CanCut: Boolean;
begin
```



```
    Result := m_Edit.SelLength <> 0;
end;

function TssnMemoEditor.CanDeleteSelection: Boolean;
begin
    Result := m_Edit.SelLength <> 0;
end;

function TssnMemoEditor.CanPaste: Boolean;
begin
    Result := true;
end;

function TssnMemoEditor.CanRedo: Boolean;
begin
    Result := true;
end;

function TssnMemoEditor.CanUndo: Boolean;
begin
    Result := true;
end;

procedure TssnMemoEditor.Copy;
begin
    m_Edit.CopyToClipboard();
end;

constructor TssnMemoEditor.Create(ParentCtrl: TWinControl);
begin
    m_Edit := TMemo.Create(nil);
    m_Edit.Parent := ParentCtrl;
    m_Edit.Align := alClient;
    m_Edit.Visible := true;
    m_Edit.WordWrap := false;
    m_Edit.ScrollBars := ssBoth;
    m_Edit.OnMouseUp := OnMouseUp;
    m_Edit.OnKeyUp := OnKeyUp;
    if m_Edit.CanFocus() then
        m_Edit.SetFocus();
end;
```

```
procedure TssnMemoEditor.Cut;
begin
    m_Edit.CutToClipboard();
end;

procedure TssnMemoEditor.DeleteLine;
begin
    m_Edit.SelStart := m_Edit.SelStart - m_Edit.CaretPos.X;
    m_Edit.SelLength := Length(m_Edit.Lines[m_Edit.CaretPos.Y]) + 2;
    m_Edit.ClearSelection();
end;

procedure TssnMemoEditor.DeleteSelection;
begin
    m_Edit.ClearSelection();
end;

function TssnMemoEditor.FindNext(Text: String; Option: TFindOptions) :
Boolean;
var
    FoundAt : Integer;
    LastFoundAt : Integer;
    AllText : String;
begin
    Result := false;

    if frDown in Option then
        AllText := System.Copy(m_Edit.Text, m_Edit.SelStart +
            m_Edit.SelLength + 1, Length(m_Edit.Text))
    else
        AllText := System.Copy(m_Edit.Text, 1, m_Edit.SelStart);

    if frMatchCase in Option then
        begin
            AllText := UpperCase(AllText);
            Text := UpperCase(Text);
        end;

    if frDown in Option then
        begin
            FoundAt := Pos(Text, AllText);
            if FoundAt = 0 then
```



```
        Exit;
        m_Edit.SelStart := m_Edit.SelStart + m_Edit.SelLength + FoundAt;
    end
    else
    begin
        LastFoundAt := 0;
        repeat
            FoundAt := Pos(Text, AllText);
            if FoundAt <> 0 then
            begin
                AllText := System.Copy(
                    AllText,
                    FoundAt + 1,
                    Length(AllText)
                );
                LastFoundAt := LastFoundAt + FoundAt;
            end
        until FoundAt = 0;
        if LastFoundAt = 0 then
            Exit;
        m_Edit.SelStart := LastFoundAt;
    end;

    m_Edit.SelLength := Length(Text);
    Result := true;
    if m_Edit.CanFocus() then
        m_Edit.SetFocus();
end;

function TssnMemoEditor.GetSaved: Boolean;
begin
    Result := not m_Edit.Modified;
end;

function TssnMemoEditor.GetSelectText: String;
begin
    Result := m_Edit.SelText;
end;

procedure TssnMemoEditor.OnKeyUp(Sender: TObject; var Key: Word;
    Shift: TShiftState);
begin
```

```
    OnEditorSelectionChange(Sender);
end;

procedure TssnMemoEditor.DoLoadFromFile(FileName: String);
begin
    m_Edit.Lines.LoadFromFile(FileName);
end;

procedure TssnMemoEditor.OnMouseUp(Sender: TObject; Button:
TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    OnEditorSelectionChange(Sender);
end;

procedure TssnMemoEditor.Paste;
begin
    m_Edit.PasteFromClipboard();
end;

procedure TssnMemoEditor.Redo;
begin
    m_edit.Undo();
end;

function TssnMemoEditor.Replace(FindText, ReplaceText: String;
    Option: TFindOptions): Integer;
var
    SelText : String;
begin
    Result := 0;

    if frMatchCase in Option then
    begin
        SelText := UpperCase(m_Edit.SelText);
        FindText := UpperCase(FindText);
    end
    else
        SelText := m_Edit.SelText;

    if FindText = SelText then
    begin
```



```
        m_Edit.SelText := ReplaceText;
        Result := 1;
    end;

    if not (frReplaceAll in Option) then
        Exit;

    while FindNext(FindText, Option) do
    begin
        m_Edit.SelText := ReplaceText;
        Inc(Result);
    end;
end;

procedure TssnMemoEditor.SaveToFile(FileName: String);
begin
    m_Edit.Lines.SaveToFile(FileName);
end;

procedure TssnMemoEditor.SelectAll;
begin
    m_Edit.SelectAll();
end;

procedure TssnMemoEditor.SetFont(Font: TFont);
begin
    m_Edit.Font := Font;
end;

procedure TssnMemoEditor.Undo;
begin
    m_edit.Undo();
end;

function TssnMemoEditor.GetText: String;
begin
    Result := m_Edit.Lines.Text;
end;

function TssnMemoEditor.GetWordWrap: Boolean;
begin
    Result := m_Edit.WordWrap;
end;
```

```

end;

procedure TssnMemoEditor.SetWordWrap(WordWrap: Boolean);
begin
  m_Edit.WordWrap := WordWrap;
  if WordWrap then
    m_Edit.ScrollBars := ssVertical
  else
    m_Edit.ScrollBars := ssBoth;
end;

destructor TssnMemoEditor.Destroy;
begin
  m_Edit.Free();
  m_Edit := nil;
end;

end.

```

好了，TssnMemoEditor 已经基本实现好了（除了事件处理相关的内容与代码，将在 7.5 节详述），有没有一些成就感呢？

TssnMemoEditor 依赖于 TMemo，因此它本身的实现并不复杂，这也算是重用带来的好处吧。假设以后的某一天找到了更好的编辑器组件，那么只需要重新实现一个类似 TssnMemoEditor 的类，即可扩展。

它们之间的关系如图 7.2 所示。

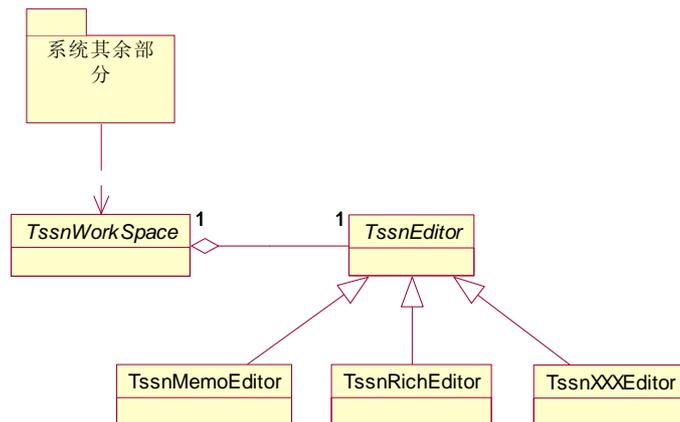


图 7.2 TssnEditor 与它的派生类

在图 7.2 中可以看到，TssnWorkspace 是一个“工作区”的抽象接口，它内含一个 TssnEditor，通过 TssnEditor 的接口对它进行各种功能调用。也就是说，TssnWorkspace 的



可见视野里，只有 `TssnEditor`，而没有 `TssnMemoEditor`，因此，当实现了 `TssnRichEditor` 或者其他的 `TssnXXXEditor` 之后，程序其余部分不用做任何改动。

下面用 `TRichEdit` 再实现一个编辑器组件，以说明当需要更换编辑器组件时，需要做什么工作，以及这样设计所带来的好处。

### 7.2.4 TssnRichEditor

前面用 `TMemo` 实现了 `IssnEditor` 接口，可是某天你的经理可能会对你说，`TMemo` 太土啦，在 Win9x 平台下，`TMemo` 无法承载超过 64KB 的文本。怎么办？只好重新实现一个。不过，幸好我们已经为这样的动作做了充分的准备，这也是 `TssnEditor` 存在的原因。于是便可以心平气和地遵从经理的意见。很显然，`TRichEdit` 可以打开超过 64KB 大小的文本，虽然 `TRichEdit` 是为富文本文件编辑所准备的，不过只需要将它的 `PlainText` 属性设置为 `true`，即可仅用于纯文本的处理。

可以参照已经实现了的 `TssnMemoEditor`，再实现一个 `TssnRichEditor`，也许花不了一个小时的时间，就可以轻松搞定。

`TssnRichEditor` 的声明和 `TssnMemoEditor` 非常类似，因为它们都是实现 `TssnEditor` 所规定的接口，不同之处在于 `TssnRichEditor` 内部包含的是一个 `TRichEdit` 的实例。另外，`TRichEdit` 与 `TMemo` 由于事件接口有所不同，因此事件处理也不一样。不过，在此不讨论与它们的事件相关的内容，这些内容将在 7.5 节详述。

`TssnRichEditor` 的声明如下：

```
TssnRichEditor = class(TssnEditor)
private
    // 内部包含一个 TRichEdit 实例，TssnRichEditor 只是对它进行接口改造
    m_Edit : TRichEdit;

protected
    procedure DoLoadFromFile(FileName : String); override;
    procedure OnRichEditSelectionChanged(Sender : TObject);
    function GetText() : String; override;

public
    constructor Create(ParentCtrl : TWinControl);
    destructor Destroy(); override;

    procedure SaveToFile(FileName: String); override;
    function GetSaved() : Boolean; override;
    function GetSelectText() : String; override;
    procedure SetFont(Font : TFont); override;
    procedure Undo(); override;
```

```

function CanUndo() : Boolean; override;
procedure Redo(); override;
function CanRedo() : Boolean; override;
procedure Cut(); override;
function CanCut() : Boolean; override;
procedure Copy(); override;
function CanCopy() : Boolean; override;
procedure Paste(); override;
function CanPaste() : Boolean; override;
procedure DeleteSelection(); override;
function CanDeleteSelection() : Boolean; override;
procedure DeleteLine(); override;
procedure SelectAll(); override;
function FindNext(Text : String; Option : TFindOptions) :
    Boolean; override;
function Replace(FindText, ReplaceText : String; Option :
    TFindOptions) : Integer; override;
function GetWordWrap() : Boolean; override;
procedure SetWordWrap(WordWrap : Boolean); override;
end;

```

至于具体的实现，与 `TssnMemoEditor` 的区别仅仅在于所基于的组件不同。也就是说，`TssnRichEditor` 与 `TssnMemoEditor` 都是同种性质的所谓“接口适配器”。只不过，一个是将 `TMemo` 的接口转换成 `TssnEditor`，一个是将 `TRichEdit` 的接口转换成 `TssnEditor`。

由于 `TRichEdit` 与 `TMemo` 的接口差异并不是很大，因此 `TssnRichEdit` 与 `TssnMemoEditor` 很多实现代码都是相同的。只有查找算法，由于 `TRichEdit` 提供了 `FindText()` 方法，使得查找算法的实现更为简单：

```

function TssnRichEditor.FindNext(Text: String;
    Option: TFindOptions): Boolean;
var
    FoundAt : LongInt;
    StartPos, ToEnd : Integer;
begin
    Result := false;

    // 查找起始位置
    StartPos := m_Edit.SelStart + m_Edit.SelLength;
    // 查找终止位置
    ToEnd := Length(m_Edit.Text) - StartPos;
    // 具体实施查找由 TRichEdit.FindText 实现

```



```
if (frMatchCase in Option) then
    FoundAt := m_Edit.FindText(Text, StartPos, ToEnd, [stMatchCase])
else
    FoundAt := m_Edit.FindText(Text, StartPos, ToEnd, []);

if FoundAt = -1 then // 没找到
    Exit;

m_Edit.SetFocus();
m_Edit.SelStart := FoundAt;
m_Edit.SelLength := Length(WideString(Text));
Result := true;
end;
```

实现的其余部分与 `TssnMemoEditor` 基本一样，在此不作更多解释。`TssnRichEditor` 实现在 `RichEditor.pas` 单元中。该单元代码清单如下：

```
unit RichEditor;

interface

uses ComCtrls, Controls, Classes, Graphics, Dialogs, StdCtrls, SysUtils,
    Editor, IntfEditor;

type
    TssnRichEditor = class(TssnEditor)
    private
        m_Edit : TRichEdit;
    protected
        procedure DoLoadFromFile(FileName : String); override;
        procedure OnRichEditSelectionChanged(Sender : TObject);
        function GetText() : String; override;
    public
        constructor Create(ParentCtrl : TWinControl);
        destructor Destroy(); override;
        procedure SaveToFile(FileName: String); override;
        function GetSaved() : Boolean; override;
        function GetSelectText() : String; override;
        procedure SetFont(Font : TFont); override;
        procedure Undo(); override;
        function CanUndo() : Boolean; override;
        procedure Redo(); override;
```

```
function CanRedo() : Boolean; override;
procedure Cut(); override;
function CanCut() : Boolean; override;
procedure Copy(); override;
function CanCopy() : Boolean; override;
procedure Paste(); override;
function CanPaste() : Boolean; override;
procedure DeleteSelection(); override;
function CanDeleteSelection() : Boolean; override;
procedure DeleteLine(); override;
procedure SelectAll(); override;
function FindNext(Text : String; Option : TFindOptions) : Boolean;
    override;
function Replace(FindText, ReplaceText : String; Option :
    TFindOptions) : Integer; override;
function GetWordWrap() : Boolean; override;
procedure SetWordWrap(WordWrap : Boolean); override;
end;

implementation

{ TssnRichEditor }

function TssnRichEditor.CanCopy: Boolean;
begin
    Result := m_Edit.SelLength <> 0;
end;

function TssnRichEditor.CanCut: Boolean;
begin
    Result := m_Edit.SelLength <> 0;
end;

function TssnRichEditor.CanDeleteSelection: Boolean;
begin
    Result := m_Edit.SelLength <> 0;
end;

function TssnRichEditor.CanPaste: Boolean;
begin
    Result := true;
end;
```



```
function TssnRichEditor.CanRedo: Boolean;
begin
    Result := m_Edit.CanUndo;
end;

function TssnRichEditor.CanUndo: Boolean;
begin
    Result := m_Edit.CanUndo;
end;

procedure TssnRichEditor.Copy;
begin
    m_Edit.CopyToClipboard();
end;

constructor TssnRichEditor.Create(ParentCtrl: TWinControl);
begin
    m_Edit := TRichEdit.Create(nil);
    m_Edit.PlainText := true;
    m_Edit.Parent := ParentCtrl;
    m_Edit.Align := alClient;
    m_Edit.Visible := true;
    m_Edit.WordWrap := false;
    m_Edit.ScrollBars := ssBoth;
    m_Edit.OnSelectionChange := OnRichEditSelectionChanged;
    if m_Edit.CanFocus() then
        m_Edit.SetFocus();
end;

procedure TssnRichEditor.Cut;
begin
    m_Edit.CutToClipboard();
end;

procedure TssnRichEditor.DeleteLine;
begin
    m_Edit.SelStart := m_Edit.SelStart - m_Edit.CaretPos.X;
    m_Edit.SelLength := Length(m_Edit.Lines[m_Edit.CaretPos.Y]) + 2;
    m_Edit.ClearSelection();
end;
```

```
procedure TssnRichEditor.DeleteSelection;
begin
    m_Edit.ClearSelection();
end;

destructor TssnRichEditor.Destroy;
begin
    m_Edit.Free();
    m_Edit := nil;
end;

procedure TssnRichEditor.DoLoadFromFile(FileName: String);
begin
    m_Edit.Lines.LoadFromFile(FileName);
end;

function TssnRichEditor.FindNext(Text: String;
    Option: TFindOptions): Boolean;
var
    FoundAt : LongInt;
    StartPos, ToEnd : Integer;
begin
    Result := false;

    StartPos := m_Edit.SelStart + m_Edit.SelLength;
    ToEnd := Length(m_Edit.Text) - StartPos;
    if (frMatchCase in Option) then
        FoundAt := m_Edit.FindText(Text, StartPos, ToEnd, [stMatchCase])
    else
        FoundAt := m_Edit.FindText(Text, StartPos, ToEnd, []);

    if FoundAt = -1 then
        Exit;

    m_Edit.SetFocus();
    m_Edit.SelStart := FoundAt;
    m_Edit.SelLength := Length(WideString(Text));
    Result := true;
end;

function TssnRichEditor.GetSaved: Boolean;
begin
```



```
    Result := not m_Edit.Modified;
end;

function TssnRichEditor.GetSelectText: String;
begin
    Result := m_Edit.SelText;
end;

function TssnRichEditor.GetText: String;
begin
    Result := m_Edit.Text;
end;

function TssnRichEditor.GetWordWrap: Boolean;
begin
    Result := m_Edit.WordWrap;
end;

procedure TssnRichEditor.OnRichEditSelectionChanged(Sender: TObject);
begin
    OnEditorSelectionChange(Sender);
end;

procedure TssnRichEditor.Paste;
begin
    m_Edit.PasteFromClipboard();
end;

procedure TssnRichEditor.Redo;
begin
    m_Edit.Undo();
end;

function TssnRichEditor.Replace(FindText, ReplaceText: String;
    Option: TFindOptions): Integer;
var
    SelText : String;
begin
    Result := 0;

    if frMatchCase in Option then
        begin
```

```
    SelText := UpperCase(m_Edit.SelText);
    FindText := UpperCase(FindText);
end
else
    SelText := m_Edit.SelText;

if FindText = SelText then
begin
    m_Edit.SelText := ReplaceText;
    Result := 1;
end;

if not (frReplaceAll in Option) then
    Exit;

while FindNext(FindText, Option) do
begin
    m_Edit.SelText := ReplaceText;
    Inc(Result);
end;
end;

procedure TssnRichEditor.SaveToFile(FileName: String);
begin
    m_Edit.Lines.SaveToFile(FileName);
end;

procedure TssnRichEditor.SelectAll;
begin
    m_Edit.SelectAll();
end;

procedure TssnRichEditor.SetFont(Font: TFont);
begin
    m_Edit.Font := Font;
end;

procedure TssnRichEditor.SetWordWrap(WordWrap: Boolean);
begin
    m_Edit.WordWrap := WordWrap;
end;
```





```
procedure TssnRichEditor.Undo;  
begin  
    m_Edit.Undo();  
end;  
  
end.
```

## 7.2.5 TssnWorkSpace

TssnEditor 完成的是对编辑器组件的抽象，TssnWorkSpace 则是对“工作区”概念的抽象。在逻辑上，它是 TssnEditor 的容器；在实现上，它是 TssnEditor 的代理。

首先，由于 TssnWorkSpace 是 TssnEditor 的容器，如图 7.1 所示，两者是包含与被包含的关系。因此 TssnWorkSpace 一定会有一个 private 的 TssnEditor 实例的成员。除此之外，由于有多工作区的存在，必须为每个工作区提供一个标识符（编号），这也便于工作区的更上层的管理器（见 7.3 节）进行管理工作。

于是，TssnWorkSpace 会至少拥有两个 private 成员：

```
m_Editor : TssnEditor;  
m_Index : Integer;
```

其中，m\_Editor 是不允许被外界所了解和访问的，而标识符可能是外界需要的，但它的值不能被随便设置（只有在构造时，由其管理器/创建者指定），因此需要提供一个获取 m\_Index 的方法——GetIndex()。

另外，由于这两个成员都可能被 TssnWorkSpace 的派生类所访问，因此决定将它们从 private 节移到 protected 节中。

其次，由于 TssnWorkSpace 也是从 IssnEditor 接口派生，因此它（或者它的派生类）必须实现 IssnEditor 所声明的所有方法。不过，对于这些方法，TssnWorkSpace 只需要履行作为代理者的职责即可——即简单地将请求直接转发给内部的 m\_Editor(TssnEditor 实例)，因为它们都是派生自同一个接口 IssnEditor。

最后，还得考虑一个问题，就是工作区被关闭的问题。关闭意味着销毁其中的 m\_Editor 实例，然后销毁自己，这似乎在析构函数中就可以完成了。但可以设想，用户选择关闭工作区时，如果该工作区的文本尚未被保存，系统一般会提示询问用户是否保存该文本，同时也允许选择“取消”，即取消关闭操作。此时，m\_Editor 将不能被销毁，工作区自己也不能被销毁。而如果把关闭的逻辑置于析构函数中，则将无法“反悔”。解决这个问题很简单，可以提供一个 Close()方法，以完成判断是否已经保存、询问用户是否保存等逻辑；同时，如果用户选择了“取消”，则不作任何处理。如果确定要关闭工作区，则销毁 m\_Editor 实例。这里似乎还有销毁自己的职责，但是我只能说，没有！为什么？因为在一个对象的方法中，显式调用 self.Free()会造成少量的资源泄漏（系统 User 对象以及内存），这应该

是 Delphi 的编译器的 bug。所以一般不要在对象方法中显式调用自己的析构函数，而应该将这个任务交给外界管理器，也许在 Delphi 的世界里没有人愿意自杀吧。因此，把工作区的销毁工作应交给它的管理器（见 7.3 节）。

好，看一下 TssnWorkspace 的声明：

```
TssnWorkspace = class(IssnEditor)
protected
    m_Editor : TssnEditor;
    m_Index : Integer;

public
    constructor Create(ParentCtrl : TWinControl; FileName : String;
        nIndex : Integer);
    // 用 Close 取代析构函数
    function Close() : Integer;
    // 获取工作区的标识符
    function GetIndex() : Integer;

    // 实现 IssnEditor 所声明的方法
    function GetFileName() : String; override;
    function GetSaved() : Boolean; override;
    function Save() : Boolean; override;
    function SaveAs() : Boolean; override;
    function GetSelectText() : String; override;
    procedure SetFont(Font : TFont); override;
    procedure Undo(); override;
    function CanUndo() : Boolean; override;
    procedure Redo(); override;
    function CanRedo() : Boolean; override;
    procedure Cut(); override;
    function CanCut() : Boolean; override;
    procedure Copy(); override;
    function CanCopy() : Boolean; override;
    procedure Paste(); override;
    function CanPaste() : Boolean; override;
    procedure DeleteSelection(); override;
    function CanDeleteSelection() : Boolean; override;
    procedure DeleteLine(); override;
    procedure SelectAll(); override;
    function FindNext(Text : String; Option : TFindOptions) : Boolean;
        override;
```



```
function Replace(FindText, ReplaceText : String; Option :
TFindOptions):
    Integer; override;
function GetWordCount() : TssnWordCountRec; override;
function GetWordWrap() : Boolean; override;
procedure SetWordWrap(WordWrap : Boolean); override;
end;
```

工作区需要有实际的载体组件，而其载体组件随着工作区的不同风格的实现也不相同，因此在构造函数中传入一个 `TWinControl` 类型的组件引用，以指定工作区具体的载体组件，将其作为工作区的“父窗口”：

```
constructor TssnWorkSpace.Create(ParentCtrl : TWinControl;
    FileName : String; nIndex : Integer);
begin
    // 通过构造器创建 m_Editor 实例
    // 关于“构造器”的内容，详见 7.4 节
    g_EditorCtor.CreateAnEditor(m_Editor, ParentCtrl);

    if FileName <> '' then
        m_Editor.LoadFromFile(FileName);

    m_Index := nIndex;
end;
```

`Close()`方法取代了析构函数（确切地说，不是“取代”，因为其功能与析构函数并不完全相同）：

```
function TssnWorkSpace.Close: Integer;
var
    AskRusult : Integer;
begin
    Result := 0;

    // 如果文件尚未被保存，则询问用户是否保存并允许“取消”
    if not m_Editor.GetSaved() then
    begin
        // 关于 g_InterActive，将在 7.6 节详述
        // 在此可以认为它的功能与 API MessageBox 类似
        AskRusult := g_InterActive.MessageBox(
```

```
        str_PromptSave,  
        Application.Title,  
        MB_YESNOCANCEL or MB_ICONQUESTION  
    );  
  
    if AskRusult = IDYES then  
    begin // 保存文本并关闭  
        try  
            if not m_Editor.Save() then  
                Exit;  
        except // 保存失败（如文件为只读），提示用户并取消  
            g_InterActive.MessageBox(  
                str_SaveError,  
                Application.Title,  
                MB_ICONSTOP  
            );  
            Exit;  
        end; // try  
    end  
    else if AskRusult = IDCANCEL then // 取消  
        Exit;  
    end;  
  
    // 销毁工作区组件  
    m_Editor.Free();  
    m_Editor := nil;  
  
    Result := 1;  
end;
```

取得工作区标识符的 `GetIndex()` 方法的实现只需直接返回 `m_Index` 即可：

```
function TssnWorkspace.GetIndex: Integer;  
begin  
    Result := m_Index;  
end;
```

剩余的方法皆是从 `IssnEditor` 派生并需要 `override` 的。对于这些方法，只需履行代理者的任务，直接将功能调用请求全部转发给 `m_Editor` 即可。如：

```
function TssnWorkspace.CanCopy: Boolean;
```



```
begin
    Result := m_Editor.CanCopy();
end;

function TssnWorkSpace.CanCut: Boolean;
begin
    Result := m_Editor.CanCut();
end;

.....
```

TssnWorkSpace 实现在 WorkSpace.pas 单元中。该单元的代码清单如下：

```
unit WorkSpace;

interface

uses Controls, Forms, Windows, Graphics, Dialogs,
    Editor, IntfEditor;

type
    TssnWorkSpace = class(IssnEditor)
    protected
        m_Editor : TssnEditor;
        m_Index : Integer;

        procedure OnSave(); virtual; abstract;

    public
        constructor Create(ParentCtrl : TWinControl; FileName : String;
nIndex : Integer);

        function Close() : Integer;
        function GetIndex() : Integer;

        function GetFileName() : String; override;
        function GetSaved() : Boolean; override;
        function Save() : Boolean; override;
        function SaveAs() : Boolean; override;
        function GetSelectText() : String; override;
        procedure SetFont(Font : TFont); override;
        procedure Undo(); override;
```

```
function CanUndo() : Boolean; override;
procedure Redo(); override;
function CanRedo() : Boolean; override;
procedure Cut(); override;
function CanCut() : Boolean; override;
procedure Copy(); override;
function CanCopy() : Boolean; override;
procedure Paste(); override;
function CanPaste() : Boolean; override;
procedure DeleteSelection(); override;
function CanDeleteSelection() : Boolean; override;
procedure DeleteLine(); override;
procedure SelectAll(); override;
function FindNext(Text : String; Option : TFindOptions) : Boolean;
override;
function Replace(FindText, ReplaceText : String; Option :
    TFindOptions) : Integer; override;
function GetWordCount() : TssnWordCountRec; override;
function GetWordWrap() : Boolean; override;
procedure SetWordWrap(WordWrap : Boolean); override;
end;

implementation

uses GlobalObject, MultiLan;

{ TssnWorkspace }

function TssnWorkspace.CanCopy: Boolean;
begin
    Result := m_Editor.CanCopy();
end;

function TssnWorkspace.CanCut: Boolean;
begin
    Result := m_Editor.CanCut();
end;

function TssnWorkspace.CanDeleteSelection: Boolean;
begin
    Result := m_Editor.CanDeleteSelection();
end;
```



```
function TssnWorkSpace.CanPaste: Boolean;
begin
    Result := m_Editor.CanPaste();
end;

function TssnWorkSpace.CanRedo: Boolean;
begin
    Result := m_Editor.CanRedo();
end;

function TssnWorkSpace.CanUndo: Boolean;
begin
    Result := m_Editor.CanUndo();
end;

function TssnWorkSpace.Close: Integer;
var
    AskRusult : Integer;
begin
    Result := 0;

    if not m_Editor.GetSaved() then
    begin
        AskRusult := g_InterActive.MessageBox(str_PromptSave,
            Application.Title, MB_YESNOCANCEL or MB_ICONQUESTION);
        if AskRusult = IDYES then
        begin // save
            try
                if not m_Editor.Save() then
                    Exit;
            except
                g_InterActive.MessageBox(
                    str_SaveError,
                    Application.Title,
                    MB_ICONSTOP
                );
                Exit;
            end;
        end
        else if AskRusult = IDCANCEL then
            Exit;
```

```
end;

m_Editor.Free();
m_Editor := nil;

Result := 1;
end;

procedure TssnWorkSpace.Copy;
begin
    m_Editor.Copy();
end;

constructor TssnWorkSpace.Create(ParentCtrl : TWinControl; FileName :
String; nIndex : Integer);
begin
    g_EditorCtor.CreateAnEditor(m_Editor, ParentCtrl);

    if FileName <> '' then
        m_Editor.LoadFromFile(FileName);

    m_Index := nIndex;
end;

procedure TssnWorkSpace.Cut;
begin
    m_Editor.Cut();
end;

procedure TssnWorkSpace.DeleteLine;
begin
    m_Editor.DeleteLine();
end;

procedure TssnWorkSpace.DeleteSelection;
begin
    m_Editor.DeleteSelection();
end;

function TssnWorkSpace.FindNext(Text: String; Option: TFindOptions) :
Boolean;
begin
```



```
    Result := m_Editor.FindNext(Text, Option);
end;

function TssnWorkSpace.GetFileName: String;
begin
    Result := m_Editor.GetFileName();
end;

function TssnWorkSpace.GetIndex: Integer;
begin
    Result := m_Index;
end;

function TssnWorkSpace.GetSaved: Boolean;
begin
    Result := m_Editor.GetSaved();
end;

function TssnWorkSpace.GetSelectText: String;
begin
    Result := m_Editor.GetSelectText();
end;

function TssnWorkSpace.GetWordCount: TssnWordCountRec;
begin
    Result := m_Editor.GetWordCount();
end;

function TssnWorkSpace.GetWordWrap: Boolean;
begin
    Result := m_Editor.GetWordWrap();
end;

procedure TssnWorkSpace.Paste;
begin
    m_Editor.Paste();
end;

procedure TssnWorkSpace.Redo;
begin
    m_Editor.Redo();
end;
```

```
function TssnWorkSpace.Replace(FindText, ReplaceText: String;
    Option: TFindOptions): Integer;
begin
    Result := m_Editor.Replace(FindText, ReplaceText, Option);
end;

function TssnWorkSpace.Save : Boolean;
begin
    Result := m_Editor.Save();
    OnSave();
end;

function TssnWorkSpace.SaveAs: Boolean;
begin
    Result := m_Editor.SaveAs();
    OnSave();
end;

procedure TssnWorkSpace.SelectAll;
begin
    m_Editor.SelectAll();
end;

procedure TssnWorkSpace.SetFont(Font: TFont);
begin
    m_Editor.SetFont(Font);
end;

procedure TssnWorkSpace.SetWordWrap(WordWrap: Boolean);
begin
    m_Editor.SetWordWrap(WordWrap);
end;

procedure TssnWorkSpace.Undo;
begin
    m_Editor.Undo();
end;

end.
```

代码清单中，已经完整实现了 TssnWorkSpace，其中包括了尚未论及的事件处理代码，



如 `OnSave()`方法就是一个用于事件处理的抽象虚方法，这些内容将在 7.5 节详述，暂且忽略。

至此，我们已经实现了 `TssnWorkSpace`，下一步当然就要考虑工作区的具体实现及其管理器了。

## 7.3 TssnWorkSpaceMgr/TssnWorkSpace

根据需求，Sunny SmartNote 是一个多工作区的纯文本编辑软件。因此，需要对工作区的管理器进行抽象，由它负责维护、管理所有的工作区，如新建、关闭一个工作区，获取当前激活的工作区等。这样，界面模块的代码就可以从繁琐的工作区管理工作中解脱出来。

自然地，就有了 `TssnWorkSpaceMgr`。

### 7.3.1 TssnWorkSpaceMgr

`TssnWorkSpaceMgr` 如同 `TssnWorkSpace`、`TssnEditor` 一样，也是一个抽象类。它给出的是作为一个“工作区管理器”所必需的接口。具体实现每一种界面风格的“工作区管理器”的类都是从 `TssnWorkSpaceMgr` 派生，以此将“工作区管理器”的实现与接口分离。接口是不变的，实现可以随意更改。

另外，“工作区管理器”的每一种界面风格的具体实现，其实是与“工作区”的界面风格的实现相关的。比如，用 MDI 风格实现多工作区，那么多工作区管理器就是一个“子窗口管理器”，维护每一个打开的“工作区”子窗口；如果用分页风格（`TPageControl`）来实现多工作区，那么工作区管理器就对应于一个 `TPageControl` 组件，而具体的每个工作区就对应于一个 `TTabSheet` 组件。

`TssnWorkSpaceMgr` 与 `TssnWorkSpace` 的关系是一对多的关系。也就是说，一个“工作区管理器”包含多个“工作区”，如图 7.3 所示。

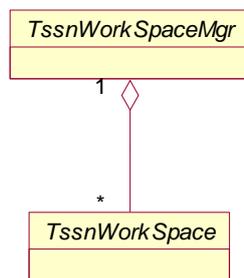


图 7.3 TssnWorkSpaceMgr 与 TssnWorkSpace 之间的关系

首先，应确定一个同时允许打开的工作区的最大数目，也就是说，`TssnWorkSpaceMgr` 最多只需要管理这些数量的工作区。在此假定为 128，即允许用户同时打开 128 个文件进

行编辑，超过此限制时将提供给用户错误提示。

其次，作为多工作区的管理器，它必须认识、知道所有的工作区，并维护它们的状态。因此 `TssnWorkSpace` 拥有两个列表——工作区列表和状态列表。另外，为了效率问题，决定给 `TssnWorkSpaceMgr` 配备一个表示已经打开工作区总数的数据成员，以免当需要该数值时才“临时抱佛脚”匆忙去计算出来。

最后，由于工作区列表和状态列表可能会被 `TssnWorkSpaceMgr` 的派生类所用到（如派生类需要知道某一工作区状态时），因此将两个列表放置在 `protected` 节中。

至此，`TssnWorkSpaceMgr` 看起来应该是这样的：

```
TssnWorkSpaceMgr = class
private
    m_OpenedCount : Integer;
protected
    // 工作区列表
    m_WorkList : array [1 .. SSN_MAX_WORKSPACE] of TssnWorkSpace;
    // 工作区状态列表，两表一一对应
    m_WorkStatus : array [1 .. SSN_MAX_WORKSPACE] of
TssnWorkSpaceStatusRec;
end;
```

其中的 `TssnWorkSpaceStatusRec` 记录类型标明工作区状态（已打开还是已关闭）：

```
TssnWorkSpaceStatusRec = record
    Opened : Boolean;
end;
```

目前，对于工作区只需要知道其处于打开还是关闭状态，因此，只有一个 `Opened` 数据域。当今后的需求发生变动，需要表示出工作区繁荣更多状态时，可以扩展 `TssnWorkSpaceStatusRec` 记录类型。

除了这些数据成员之外，`TssnWorkSpaceMgr` 当然也要同 `IssnEditor` 一样，定义出一套工作区功能集合的规范，由自己或其派生类实现，以使其客户程序可以放心地将多工作区管理工作交给 `TssnWorkSpaceMgr`。

如第 2 章中所说，定义接口功能集合最重要的是使得接口圆满并且是最小冗余的。也就是说，所提供的功能集合可以完成该类的所有职责并且尽可能不引进多余的功能，如果有必要引进，也一定是因为可以大大减轻客户程序的工作量或大大提高效率（即该接口是有效的）。

基于以上原则，来考虑一下 `TssnWorkSpaceMgr` 的功能集合。

首先最基本的，也是最明显的功能，如新建一个工作区、关闭一个工作区、激活（选中）一个工作区、获取某一个工作区、查询已打开工作区的总数，这些都是最基本的，因



此很容易做出取舍的决策。

其次，有了这些功能后，集合是否圆满了呢？还差一项功能，即查询获取当前激活的工作区。实现此功能有两种途径：

- (1) 在集合中加入一个直接获取当前被激活的工作区的方法；
- (2) 在集合中加入一个获取当前被激活的工作区的标识符（索引值）的方法。

两种途径都可以达到目的，选谁呢？

假设选择途径 1，那么要获取当前激活的工作区的标识符（索引值）就必须先取得活动工作区，然后通过 `TssnWorkSpace.GetIndex()` 返回其标识符（索引值）；如果选择途径 2，那么要获取当前激活的工作区，就必须先取得其标识符（索引值），然后通过基本功能集合中的“获取某一工作区”方法来完成。

这里的视角，只是客户程序的视角。其实，在实现时你会发现，其实两种方案是互补的，即两者都需要。只是在 `TssnWorkSpaceMgr.GetActiveWorkSpace()`（取得实际的当前激活的工作区对象）和 `TssnWorkSpaceMgr.GetActiveWorkSpaceIndex()`（取得当前激活的工作区的索引值）中，选择将谁作为 `public`，将谁作为 `private/protected` 而已。

由于在大多数情况下，客户程序都是希望直接获取激活的工作区对象（而非其索引值），因此选择采用方案 1。

最后，在已经“圆满”的功能集合之外，还需考虑要为客户程序提供什么样的有效的功能。

一类是所有工作区的统一操作，如：保存所有工作区文本、关闭所有工作区。对于这类操作，客户程序完全可以通过以上的功能集合来完成——遍历所有工作区，分别对每个工作区进行功能调用以完成任务。但这类代码会使得客户程序非常累赘而不美观，而如果由 `TssnWorkSpaceMgr` 来实现，则有了优化的可能（如：跳过未打开的工作区），即使性能上没有优化，却也可以让客户程序只需简单的一个调用即可完成，大大简化了客户程序。因此，决定将它们加入 `TssnWorkSpaceMgr` 的功能集合中。

另一类可以加入的冗余功能是客户程序调用频度非常高的，将它们加入功能集合中，也是合理的。这里，决定加入的是“激活当前工作区的下一个工作区”。

看一下 `TssnWorkSpaceMgr` 的样子：

```
TssnWorkSpaceMgr = class
private
    m_OpenedCount : Integer;
protected
    m_WorkList : array [1 .. SSN_MAX_WORKSPACE] of TssnWorkSpace;
    m_WorkStatus : array [1 .. SSN_MAX_WORKSPACE] of
TssnWorkSpaceStatusRec;

    function DoNewWorkSpace (var WorkSpace : TssnWorkSpace; FileName :
        String; nIndex : Integer) : Integer; virtual; abstract;
    procedure DoActiveWorkSpace (nIndex : Integer); virtual; abstract;
```

```

function GetActiveWorkSpaceIndex() : Integer; virtual; abstract;

public
  constructor Create();
  destructor Destroy(); override;

  function NewWorkSpace(FileName : String) : Integer;
  function CloseWorkSpace(nIndex : Integer) : Boolean;
  procedure ActiveWorkSpace(nIndex : Integer);
  function ActiveNextWorkSpace() : Integer;
  function GetWorkSpace(nIndex : Integer) : TssnWorkSpace;
  function GetWorkSpaceCount() : Integer;
  function GetActiveWorkSpace() : TssnWorkSpace;
  function CloseAll() : Boolean;
  function SaveAll() : Boolean;
end;

```

可以发现，除了以上提到的功能集合与数据成员之外，`protected` 节中还多了 3 个方法：`DoNewWorkSpace()`、`DoActiveWorkSpace()`、`GetActiveWorkSpaceIndex()`。

前面提到过，`GetActiveWorkSpaceIndex()`与 `GetActiveWorkSpace()`其实是互补的。在此处，`GetActiveWorkSpaceIndex()`的实现与具体的工作区管理类的实现（即 `TssnWorkSpaceMgr` 的派生类）有关，因此将其声明为抽象虚方法。

至于 `DoNewWorkSpace()`与 `DoActiveWorkSpace()`，也许读者已经猜出来它们所扮演的角色了。是的，还是那个熟悉的 `Template Method` 模式的应用。`DoNewWorkSpace()`和 `NewWorkSpace()`、`DoActiveWorkSpace()`和 `ActiveWorkSpace()`一起再现了 7.2 节中提到过的 `Template Method` 模式——即将某操作的部分延迟到派生类实现，而同时又保证其余部分的实现在基类中必定得到执行的一种方法。由 `NewWorkSpace()`完成某些功能，并调用派生类覆盖的抽象虚方法 `DoNewWorkSpace()`来真正创建一个新的工作区；由 `ActiveWorkSpace()`完成某些功能，并调用派生类覆盖的抽象虚方法 `DoActiveWorkSpace()`来真正激活某一个工作区。

来看一下具体实现代码。`TssnWorkSpaceMgr` 实现在 `WorkSpaceMgr.pas` 单元中，在本小节最后，会给出该单元的代码清单。在此，只介绍部分方法的实现。

先来看一下关闭某个工作区——`CloseWorkSpace()`方法的实现。其中涉及到了 7.2.5 节中为 `TssnWorkSpace` 设计的 `Close()`方法的调用，还记得这个 `Close()`方法吗？为了实现关闭工作区时，允许用户选择提示中的“取消”，即允许反悔关闭操作。在此没有把“工作区”的销毁工作放在 `TssnWorkSpace` 中实现，而决定交给“工作区管理器”来做。现在就要来实现它。`TssnWorkSpace` 的 `Close()`方法如果被用户取消，会返回 0。因此在此处的实现中，需要先调用即将被关闭的工作区对象的 `Close()`方法，然后判断其返回值是否为 0。是则表示取消关闭操作，直接退出；否则就要销毁工作区对象，并激活下一个打开的工作区。其代码如下：



```
function TssnWorkspaceMgr.CloseWorkspace(nIndex : Integer): Boolean;
var
    i : Integer;
begin
    // 关闭某一个工作区
    Result := false;

    if nIndex = 0 then
        Exit;
    if not m_WorkStatus[nIndex].Opened then
        Exit;

    if m_WorkList[nIndex].Close() <> 0 then
    begin // 如果关闭工作区成功（即没有被用户取消）
        Result := true;
        m_WorkList[nIndex].Free();
        m_WorkList[nIndex] := nil;
        m_WorkStatus[nIndex].Opened := false;
        Dec(m_OpenedCount);
        // 关闭工作区后，寻找下一个被激活的工作区
        for i := nIndex - 1 downto 1 do
        begin
            if m_WorkStatus[i].Opened then
            begin
                ActiveWorkspace(i);
                Exit;
            end;
        end;

        for i := nIndex + 1 to SSN_MAX_WORKSPACE do
        begin
            if m_WorkStatus[i].Opened then
            begin
                ActiveWorkspace(i);
                Exit;
            end;
        end;
    end;

    // 事件委托相关的代码，将在 7.5 节详述，此处不表
    // g_WorkSpaceEvent.OnWorkSpaceOpenClose(nil);
end;
```

然后，来看一下新建一个工作区的实现。当客户请求新建一个工作区时，首先查找空闲工作区，如果找到则创建出工作区对象，并激活它，返回值为新建工作区的索引号（标识符）；如果没找到则返回 0。其中，真正创建一个工作区对象，是调用前述的 DoNewWorkspace()完成的。代码如下：

```
function TssnWorkspaceMgr.NewWorkspace(FileName : String): Integer;
var
    i : Integer;
begin
    // 新建一个工作区
    Result := 0;

    // 寻找一个“未打开”（空闲的）的工作区
    for i := 1 to SSN_MAX_WORKSPACE do
    begin
        if not m_WorkStatus[i].Opened then
        begin
            Result := i;
            break;
        end;
    end;

    // 如果找到
    if Result <> 0 then
    begin
        m_WorkStatus[i].Opened := true;
        DoNewWorkspace(m_WorkList[i], FileName, i);
        ActiveWorkspace(i);
        Inc(m_OpenedCount);
    end;

    // 事件委托相关的代码，将在 7.5 节详述，此处不表
    // g_WorkSpaceEvent.OnWorkspaceOpenClose(nil);
end;
```

最后，来看一下获取当前工作区对象的 GetActiveWorkspace()方法的实现。之前提到过，GetActiveWorkspace()和 GetActiveWorkSpaceIndex()（获取工作区索引号）两个方法是互补的，都是需要的。在此选择了将 GetActiveWorkspace()方法 public 出来。因此，GetActiveWorkspace()方法依赖于 GetActiveWorkSpaceIndex()来实现，其算法就是先获得当前激活的工作区的索引号，然后通过索引号在工作区列表中找到真正的工作区对象。其代码如下：



```
function TssnWorkspaceMgr.GetActiveWorkspace: TssnWorkspace;
var
    nActive : Integer;
begin
    // 获取当前激活的工作区对象
    Result := nil;
    // 调用派生类的 GetActiveWrokSpaceIndex() 获取激活工作区的索引值
    nActive := GetActiveWorkSpaceIndex();
    if nActive = 0 then
        Exit;
    Result := m_WorkList[nActive];
end;
```

实现 TssnWorkspaceMgr 的 WorkspaceMgr.pas 单元代码清单如下（实现中涉及到一些事件处理的内容，将在 7.5 节详述）：

```
unit WorkspaceMgr;

interface

uses controls, dialogs, SysUtils,
    ssnPublic, Workspace;

type
    TssnWorkspaceStatusRec = record
        Opened : Boolean;
    end;

    TssnWorkspaceMgr = class
    private
        m_OpenedCount : Integer;
    protected
        m_WorkList : array [1 .. SSN_MAX_WORKSPACE] of TssnWorkspace;
        m_WorkStatus : array [1 .. SSN_MAX_WORKSPACE] of
            TssnWorkspaceStatusRec;

        function DoNewWorkspace (var Workspace : TssnWorkspace; FileName :
            String; nIndex : Integer) : Integer; virtual; abstract;
        procedure DoActiveWorkspace (nIndex : Integer); virtual; abstract;
        function GetActiveWorkSpaceIndex() : Integer; virtual; abstract;

    public
```

```
constructor Create();
destructor Destroy(); override;

function NewWorkspace(FileName : String) : Integer;
function CloseWorkspace(nIndex : Integer) : Boolean;
procedure ActiveWorkspace(nIndex : Integer);
function GetWorkspace(nIndex : Integer) : TssnWorkspace;
function GetWorkspaceCount() : Integer;

function CloseAll() : Boolean;
function SaveAll() : Boolean;

function ActiveNextWorkspace() : Integer;
function GetActiveWorkspace() : TssnWorkspace;
end;

implementation

uses GlobalObject;

{ TssnWorkspaceMgr }

constructor TssnWorkspaceMgr.Create();
var
    i : Integer;
begin
    for i := 1 to SSN_MAX_WORKSPACE do
        begin
            m_WorkList[i] := nil;
            m_WorkStatus[i].Opened := false;
        end;

        m_OpenedCount := 0;
    end;

destructor TssnWorkspaceMgr.Destroy();
var
    i : Integer;
begin
    for i := 1 to SSN_MAX_WORKSPACE do
        begin
            if m_WorkStatus[i].Opened then
```



```
begin
    m_WorkList[i].Free();
    m_WorkList[i] := nil;
    m_WorkStatus[i].Opened := false;
end;
end;
end;

function TssnWorkspaceMgr.CloseWorkspace(nIndex : Integer) : Boolean;
var
    i : Integer;
begin
    Result := false;

    if nIndex = 0 then
        Exit;
    if not m_WorkStatus[nIndex].Opened then
        Exit;

    if m_WorkList[nIndex].Close() <> 0 then
    begin
        Result := true;
        m_WorkList[nIndex].Free();
        m_WorkList[nIndex] := nil;
        m_WorkStatus[nIndex].Opened := false;
        Dec(m_OpenedCount);

        for i := nIndex - 1 downto 1 do
        begin
            if m_WorkStatus[i].Opened then
            begin
                ActiveWorkspace(i);
                Exit;
            end;
        end;

        for i := nIndex + 1 to SSN_MAX_WORKSPACE do
        begin
            if m_WorkStatus[i].Opened then
            begin
                ActiveWorkspace(i);
                Exit;
            end;
        end;
    end;
end;
```

```
        end;
    end;
end;

g_WorkSpaceEvent.OnWorkSpaceOpenClose(nil);
end;

function TssnWorkSpaceMgr.NewWorkSpace(FileName : String): Integer;
var
    i : Integer;
begin
    Result := 0;

    for i := 1 to SSN_MAX_WORKSPACE do
    begin
        if not m_WorkStatus[i].Opened then
        begin
            Result := i;
            break;
        end;
    end;
end;

if Result <> 0 then
begin
    m_WorkStatus[i].Opened := true;
    DoNewWorkSpace(m_WorkList[i], FileName, i);
    ActiveWorkSpace(i);
    Inc(m_OpenedCount);
end;

g_WorkSpaceEvent.OnWorkSpaceOpenClose(nil);
end;

procedure TssnWorkSpaceMgr.ActiveWorkSpace(nIndex: Integer);
begin
    if nIndex = 0 then
        Exit;
    if not m_WorkStatus[nIndex].Opened then
        Exit;
    DoActiveWorkSpace(nIndex);
end;
```





```
function TssnWorkspaceMgr.GetActiveWorkspace: TssnWorkspace;
var
    nActive : Integer;
begin
    Result := nil;
    nActive := GetActiveWorkspaceIndex();
    if nActive = 0 then
        Exit;
    Result := m_WorkList[nActive];
end;

function TssnWorkspaceMgr.GetWorkspace(nIndex : Integer) :
TssnWorkspace;
begin
    Result := nil;
    if m_WorkStatus[nIndex].Opened then
        Result := m_WorkList[nIndex];
end;

function TssnWorkspaceMgr.GetWorkspaceCount: Integer;
begin
    Result := m_OpenedCount;
end;

function TssnWorkspaceMgr.CloseAll: Boolean;
var
    i : Integer;
begin
    Result := false;
    for i := 1 to SSN_MAX_WORKSPACE do
        begin
            if not m_WorkStatus[i].Opened then
                continue;
            if not CloseWorkspace(i) then
                Exit;
        end;
    Result := true;
end;

function TssnWorkspaceMgr.SaveAll: Boolean;
var
    i : Integer;
```

```
begin
    Result := false;
    for i := 1 to SSN_MAX_WORKSPACE do
    begin
        if not m_WorkStatus[i].Opened then
            continue;
        if not m_WorkList[i].Save() then
            Exit;
        end;
        Result := true;
    end;
end;

function TssnWorkSpaceMgr.ActiveNextWorkSpace: Integer;
var
    i : Integer;
begin
    Result := GetActiveWorkSpaceIndex();

    if m_OpenedCount <= 1 then
        Exit;

    for i := Result + 1 to SSN_MAX_WORKSPACE do
    begin
        if not m_WorkStatus[i].Opened then
            continue;
        ActiveWorkSpace(i);
        Result := i;
        Exit;
    end;

    for i := 1 to Result - 1 do
    begin
        if not m_WorkStatus[i].Opened then
            continue;
        ActiveWorkSpace(i);
        Result := i;
        Exit;
    end;
end;
end.
```





上面接连实现了 TssnWorkSpace 和 TssnWorkSpaceMgr，它们都是抽象类，并不能直接创建出它们的实例，而是需要等待它们的派生类来合作，具体实现某一种多工作区的界面风格。

在 Sunny SmartNote 中，我们决定采用分页的用户界面来实现多工作区，如图 7.4 所示。

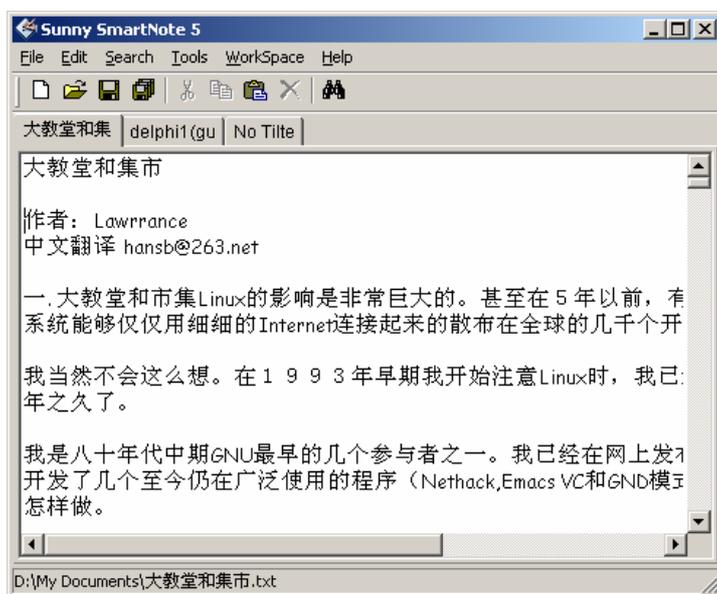


图 7.4 Sunny SmartNote 的多工作区风格

因此，根据多页的组件（TPageControl）要求，我们从 TssnWorkSpace 派生实现了 TssnTabWorkSpace，从 TssnWorkSpaceMgr 派生实现了 TssnTabWorkSpaceMgr。

当然，需求也要求可以随时更换多工作区的实现，那时只需要再从 TssnWorkSpace 与 TssnWorkSpaceMgr 派生出一套新的界面风格实现即可。

它们的关系如图 7.5 所示。

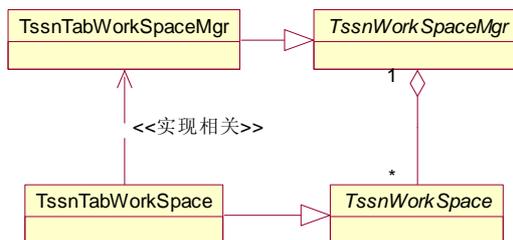


图 7.5 工作区与工作区管理器及其实现者的关系

### 7.3.2 TssnTabWorkSpace

TssnTabWorkSpace 从 TssnWorkSpace 派生，在逻辑上的职责是具体实现一个工作区的

风格。

多工作区界面风格已经确定由分页组件 `TPageControl` 来实现。`TPageControl` 组件其实是分页中的每一页——`TTabSheet` 的容器。每创建一个页，也就是创建了一个 `TTabSheet` 的实例。`TPageControl` 和 `TTabSheet` 的关系，与 `TssnWorkSpaceMgr` 和 `TssnWorkSpace` 的关系是多么地类似啊。很显然，在实现上，`TssnTabWorkSpace`（工作区）对应于 `TTabSheet`，`TssnTabWorkSpaceMgr`（工作区管理器）对应于 `TPageControl`。

`TssnTabWorkSpace` 的设计已经自然而然地浮出水面。

首先，其内部必然包含一个 `TTabSheet` 的实例。

其次，每个 `TTabSheet` 的实例都会有一个标题。在此提供一个方法来设置这个标题，也就是工作区的标题。

最后，还需要向其管理器与实现相关者——`TssnTabWorkSpaceMgr` 提供每个 `TTabSheet` 实例在 `TPageControl` 中的索引值。该索引值与为每个工作区赋予的标识符不同，工作区的标识符其实是一个逻辑上的符号（编号），它与 `TTabSheet` 在 `TPageControl` 中的索引值是不同的，该索引值是 `TPageControl` 在管理多个 `TTabSheet` 时所使用的。在具体实现 激活某个工作区时，必须告诉 `TPageControl` 要激活的那一页的索引值，而不是工作区的标识符。

就那么多，看一下 `TssnTabWorkSpace`：

```
TssnTabWorkSpace = class(TssnWorkSpace)
private
    m_TabSheet : TTabSheet; // TTabSheet 实例
    procedure SetCaption(FileName : String); // 设置标题
public
    constructor Create(Mgr : TPageControl; FileName : String;
        nIndex : Integer);
    destructor Destroy(); override;

    // 获取 PageControl 索引值
    // 该索引值与“工作区”编号索引值不同
    function TabWS_GetPageIndex() : Integer;
end;
```

可以看到，它的构造函数参数比较多。`Mgr` 指定管理器的组件，它是一个 `TPageControl`。`FileName` 给出工作区需要打开的文件名，无标题工作区则指定文件名为 ''。`nIndex` 是工作区管理器分配给新工作区的标识符。刚才说过，`TPageControl` 分配给每一个 `TTabSheet` 的索引值与在工作区列表中给每个工作区指定的索引值是完全不同的。`nIndex` 参数是工作区管理器分配给每个工作区的索引值，因此 `TTabSheet` 也必须将这个值保存下来。如何保存呢？在此处，利用了 `VCL` 中每个组件的 `tag` 属性，这个属性允许每个组件放置一个整型的附加值，正好可以利用。看一下构造函数的代码：



```
constructor TssnTabWorkspace.Create(Mgr : TPageControl; FileName :
String;
    nIndex : Integer);
begin
    // 首先创建 TTabSheet 的实例，并指定其 PageControl 属性
    m_TabSheet := TTabSheet.Create(nil);
    m_TabSheet.PageControl := Mgr;
    m_TabSheet.Align := alClient;
    m_TabSheet.Visible := true;
    // 利用 TTabSheet 组件的 tag 属性，记住该工作区的逻辑标识符
    m_TabSheet.Tag := nIndex;

    // 设置标题，并调用基类的构造函数
    SetCaption(FileName);
    inherited Create(m_TabSheet, FileName, nIndex);
end;
```

在此处，只介绍一下 TssnTabWorkspace 的构造函数的实现。如果对其他方法的实现感兴趣，可以在下一小节找到 TssnTabWorkspace 的完整代码清单。

### 7.3.3 TssnTabWorkspaceMgr

配合 TssnTabWorkspace，就有了 TssnTabWorkspaceMgr。

TssnTabWorkspaceMgr 派生自 TssnWorkspaceMgr，其职责是用 TPageControl 组件实现一个可实例化的工作区管理器。

因此，TssnTabWorkspaceMgr 只需内部包含一个 TPageControl 的实例，并实现 TssnWorkspaceMgr 声明的抽象虚方法即可。

其类声明如下：

```
TssnTabWorkspaceMgr = class(TssnWorkspaceMgr)
private
    m_Tab : TPageControl;
protected
    function DoNewWorkspace(var Workspace : TssnWorkspace;
        FileName : String; nIndex : Integer) : Integer; override;
    procedure DoActiveWorkspace(nIndex : Integer); override;
    function GetActiveWorkspaceIndex() : Integer; override;
public
    constructor Create(ParentCtrl : TWinControl);
    destructor Destroy(); override;
```

```
end;
```

其构造函数有一个 `ParentCtrl` 参数，以指定 `TPageControl` 组件放在哪一个组件中，在构造函数中完成对 `TPageControl` 实例——`m_Tab` 的创建和初始化工作：

```
constructor TssnTabWorkSpaceMgr.Create(ParentCtrl: TWinControl);
begin
    inherited Create();

    m_Tab := TPageControl.Create(nil);
    m_Tab.Parent := ParentCtrl;
    m_Tab.Align := alClient;
    m_Tab.Visible := true;

    // 以下代码为事件处理相关，关于事件委托模型将在 7.5 节详述
    // 暂且不必理会
    // m_Tab.OnChange := g_WorkSpaceEvent.OnWorkSpaceChange;
end;
```

然后是覆盖基类的 3 个抽象虚方法的实现。如前面讨论过的获取当前激活的工作区的标识符（索引值）的 `GetActiveWorkSpaceIndex()`，它只是把保存在每个 `TTabSheet` 组件的 `tag` 属性中的值取出来而已：

```
function TssnTabWorkSpaceMgr.GetActiveWorkSpaceIndex: Integer;
begin
    // 获取当前激活的工作区的标识符（索引值）
    Result := 0;
    if m_Tab.PageCount = 0 then
        Exit;

    // 还记得在创建每一个 TTabSheet 时，
    // 将其索引值放置在组件的 tag 属性中吗
    Result := m_Tab.ActivePage.Tag;
end;
```

还有配合基类的 `NewWorkSpace()` 完成 `Template Method` 模式的 `DoNewWorkSpace()` 以及配合 `ActiveWorkSpace()` 的 `DoActiveWorkSpace`：

```
function TssnTabWorkSpaceMgr.DoNewWorkSpace (
    var WorkSpace : TssnWorkSpace;
    FileName : String;
```



```
nIndex : Integer
): Integer;
begin
    // 创建一个新的工作区,
    // 即创建一个新的 TssnTabWorkSpace 实例 (TTabSheet 组件)
    WorkSpace := TssnTabWorkSpace.Create(self.m_Tab, FileName, nIndex);
    Result := Integer(WorkSpace <> nil);
end;

procedure TssnTabWorkSpaceMgr.DoActiveWorkSpace(nIndex: Integer);
begin
    // 激活一个工作区
    m_Tab.ActivePageIndex :=
        TssnTabWorkSpace(m_WorkList[nIndex]).TabWS_GetPageIndex;
end;
```

由于 `TssnTabWorkSpaceMgr` 与 `TssnTabWorkSpace` 在实现上的相关性，在此将它们实现放在同一个单元文件中，单元文件名称为 `TabWorkSpaceMgr.pas`。其代码清单如下（其中有关事件处理的内容，将在 7.5 节详述）：

```
unit TabWorkSpaceMgr;

interface

uses comctrls, Classes, ExtCtrls, Controls, Sysutils,
    WorkSpaceMgr, WorkSpace, MultiLan;

type
    TssnTabWorkSpaceMgr = class(TssnWorkSpaceMgr)
    private
        m_Tab : TPageControl;

    protected
        function DoNewWorkSpace(var WorkSpace : TssnWorkSpace; FileName :
String; nIndex : Integer) : Integer; override;
        procedure DoActiveWorkSpace(nIndex : Integer); override;
        function GetActiveWorkSpaceIndex() : Integer; override;

    public
        constructor Create(ParentCtrl : TWinControl);
        destructor Destroy(); override;
```

```
end;

TssnTabWorkSpace = class(TssnWorkSpace)
private
    m_TabSheet : TTabSheet;
    procedure SetCaption(FileName : String);

protected
    procedure OnSave(); override;

public
    constructor Create(Mgr : TPageControl; FileName : String; nIndex :
Integer);
    destructor Destroy(); override;

    function TabWS_GetPageIndex() : Integer;
end;

implementation

uses GlobalObject;

{ TssnTabWorkSpaceMgr }

constructor TssnTabWorkSpaceMgr.Create(ParentCtrl: TWinControl);
begin
    inherited Create();

    m_Tab := TPageControl.Create(ParentCtrl);
    m_Tab.Parent := ParentCtrl;
    m_Tab.Align := alClient;
    m_Tab.Visible := true;
    m_Tab.OnChange := g_WorkSpaceEvent.OnWorkSpaceChange;
end;

destructor TssnTabWorkSpaceMgr.Destroy;
begin
    inherited;

    m_Tab.Free();
    m_Tab := nil;
end;
end;
```



```
end;

function TssnTabWorkSpaceMgr.DoNewWorkSpace(var WorkSpace:
TssnWorkSpace; FileName : String; nIndex : Integer): Integer;
begin
    WorkSpace := TssnTabWorkSpace.Create(self.m_Tab, FileName, nIndex);
    Result := Integer(WorkSpace <> nil);
end;

procedure TssnTabWorkSpaceMgr.DoActiveWorkSpace(nIndex: Integer);
begin
    m_Tab.ActivePageIndex :=
TssnTabWorkSpace(m_WorkList[nIndex]).TabWS_GetPageIndex;
end;

function TssnTabWorkSpaceMgr.GetActiveWorkSpaceIndex: Integer;
begin
    Result := 0;
    if m_Tab.PageCount = 0 then
        Exit;

    Result := m_Tab.ActivePage.Tag;
end;

{ TssnTabWorkSpace }

constructor TssnTabWorkSpace.Create(Mgr : TPageControl; FileName :
String; nIndex : Integer);
begin
    m_TabSheet := TTabSheet.Create(nil);
    m_TabSheet.PageControl := Mgr;
    m_TabSheet.Align := alClient;
    m_TabSheet.Visible := true;
    m_TabSheet.Tag := nIndex;

    SetCaption(FileName);
    inherited Create(m_TabSheet, FileName, nIndex);
end;

destructor TssnTabWorkSpace.Destroy;
begin
    inherited;
end;
```

```
m_TabSheet.Free();
m_TabSheet := nil;
end;

procedure TssnTabWorkSpace.OnSave;
begin
    SetCaption(m_Editor.GetFileName());
end;

procedure TssnTabWorkSpace.SetCaption(FileName : String);
begin
    if FileName = '' then
        m_TabSheet.Caption := str_NoTitle
    else
        m_TabSheet.Caption := System.Copy(ExtractFileName(FileName), 1,
10);
end;

function TssnTabWorkSpace.TabWS_GetPageIndex: Integer;
begin
    Result := m_TabSheet.PageIndex;
end;

end.
```

至此，已经有了一整套的、可以实际工作的工作区（TssnTabWorkSpace）以及工作区管理器（TssnTabWorkSpaceMgr）。

将以上已经实现的 TssnEditor、TssnEditor、TssnMemoEditor、TssnWorkSpace、TssnWorkSpaceMgr、TssnTabWorkSpace、TssnTabWorkSpaceMgr 组合在一起，已经是可以运行的一个子系统了。作为 Sunny SmartNote 开源版本的核心功能部分的这个子系统，已经基本完成，只是这些类还没有被实际地创建出实例对象而已。

要创建出类的对象，当然是很容易的。只是，由于 Sunny SmartNote 支持多种编辑器组件的切换（即 TssnEditor 不同派生类的切换），支持多种工作区界面风格的切换（即 TssnWorkSpace/TssnWorkSpaceMgr 不同派生类的切换），简单地创建这些类的对象是达不到这种灵活性的目的的。

因此，必须另辟蹊径，来寻求这种灵活性。Sunny SmartNote 找到了解决方案，这就是下一节将要讲述的“构造器 TssnEditorCtor/TssnWorkSpaceMgrCtor”。

在结束这一节之前，给出系统设计实现至此所有的类之间的关系设计图，如图 7.6 所示。

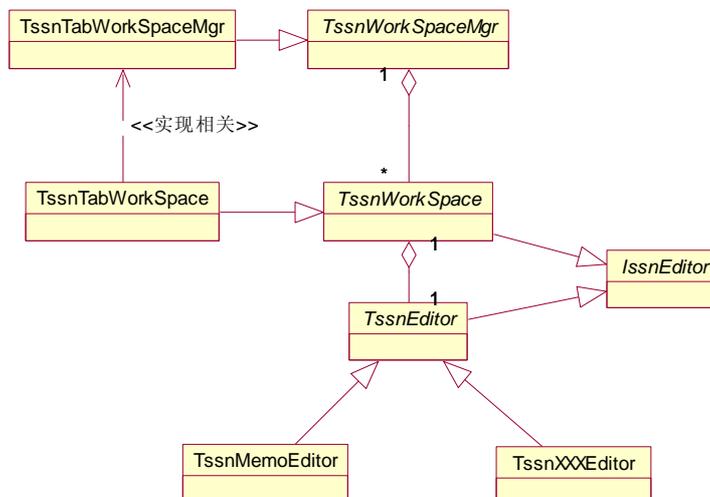


图 7.6 编辑器组件/工作区/工作区管理器子系统的类图

## 7.4 构造器 TssnEditorCtor/TssnWorkSpaceMgrCtor

需求中，要求编辑器组件是可替换的，多工作区的界面风格也是可替换的。这就要求对于如同 TssnEditor 与 TssnWorkSpace/TssnWorkSpaceMgr 的派生类实例的创建具有灵活性，即可以允许程序指定创建何种类型、何种风格的实例对象。

凡是需要灵活性的地方，就需要抽象！

是的，现在是对创建类的实例的逻辑进行抽象的时候了。这时，就需要“构造器”了。“构造器”类似一个工厂，只需要告诉它需要什么类型的产品（对象实例），即可由它为我们生产。

首先，需要为某一种“构造器”定义一个接口规范，即抽象类。然后从该抽象类派生出具体生产某一类型产品的派生类。这样，在需要创建不同的产品时，只要创建出不同的“构造器”实例对象，就可以生产出不同的“产品”。

### 7.4.1 TssnEditorCtor/TssnMemoEditorCtor/TssnRichEditorCtor

现在，要实现一个创建编辑器组件的“构造器”——TssnEditorCtor。这是一个抽象类，它为其所有派生类（具体的构造器）定义一套统一规范。其派生类对应于每一个 TssnEditor 的派生类，作为它们的生产厂。“构造器”与“产品”之间的关系如图 7.7 所示。

构造器的对外接口非常简单，一般只需要一个建造产品实例的方法即可。在此，为 TssnEditoCtor 加入惟一的一个方法：CreateAnEditor()。该方法通过参数，传出被创建出来的 TssnEditor 类型的对象引用。

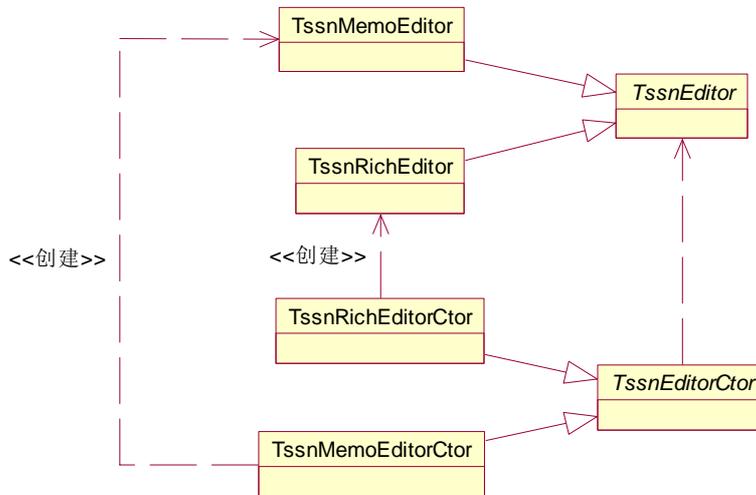


图 7.7 TssnEditor/TssnEditorCtor 及它们的派生类关系图

可以把 `CreateAnEditor()` 声明为抽象虚方法，由具体创建每种风格的编辑器实例的 `TssnEditorCtor` 的派生类来实现。但是，有一些事情需要在基类中完成。因此还是使用已经使用过多次的 `Template Method` 的做法，将 `CreateAnEditor()` 声明为非虚方法，而为其配套加入一个 `DoCreateAnEditor()` 的抽象虚方法，由派生类实现它。

```

TssnEditorCtor = class
protected
    function DoCreateAnEditor(
        var Editor : TssnEditor;
        ParentCtrl : TWinControl
    ) : Integer; virtual; abstract;
public
    function CreateAnEditor(
        var Editor : TssnEditor;
        ParentCtrl : TWinControl
    ) : Integer;
end;
  
```

在 `TssnEditorCtor` 中要实现的，仅仅只有 `CreateAnEditor()`。该方法的算法是先调用被派生类覆盖的抽象虚方法 `DoCreateAnEditor()`，然后为创建出来的对象设置一些初始值。其代码如下：

```

function TssnEditorCtor.CreateAnEditor(
    var Editor: TssnEditor;
    ParentCtrl: TWinControl
): Integer;
  
```



```
var
    DefFont : TFont;
begin
    // 传入 ParentCtrl 指明编辑器组件所在父窗口组件
    // 调用派生类的 DoCreateAnEditor 以创建真正的编辑器组件实例
    // 具体的编辑器组件风格决定于使用何种 TssnEditorCtor 的派生类实例
    Result := DoCreateAnEditor(Editor, ParentCtrl);

    // 创建成功, 则将默认字体设置作用于编辑器组件
    // 关于 g_SettingMgr, 将在 7.6 节详述, 其作用是管理整个系统的默认设置
    // 此处可以忽略
    if Boolean(Result) then
    begin
        DefFont := TFont.Create();
        g_SettingMgr.GetDefaultFont(DefFont);
        Editor.SetFont(DefFont);
        DefFont.Free();
    end;
end;
```

实现完 `TssnEditorCtor`, 继续完成其一个具体的派生类。Sunny SmartNote 在前面的实现中, 用 `TMemo` 和 `TRichEdit` 分别实现了两种 `TssnEditor` —— `TssnMemoEditor`、`TssnRichEditor`, 因此现在需要实现两个具体的构造器来创建出 `TssnMemoEditor` 和 `TssnRichEditor` 的实例对象。

从 `TssnEditorCtor` 派生出所需要的 `TssnMemoEditorCtor` —— `TssnMemoEditor` 的构造器和 `TssnRichEditorCtor` —— `TssnRichEditor` 的构造器, 它们都只需要 `override` 基类的 `DoCreate- AnEditor()`方法即可:

```
TssnMemoEditorCtor = class(TssnEditorCtor)
protected
    function DoCreateAnEditor(
        var Editor : TssnEditor;
        ParentCtrl : TWinControl
    ) : Integer; override;
end;

TssnRichEditorCtor = class(TssnEditorCtor)
protected
    function DoCreateAnEditor(
        var Editor : TssnEditor;
        ParentCtrl : TWinControl
```

```
) : Integer; override;
end;
```

它们的实现就是分别创建 TssnMemoEditor 和 TssnRichEditor 的实例对象：

```
function TssnMemoEditorCtor.DoCreateAnEditor(
    var Editor: TssnEditor;
    ParentCtrl : TWinControl
): Integer;
begin
    Editor := TssnMemoEditor.Create(ParentCtrl);
    Result := Integer(Editor <> nil);
end;

function TssnRichEditorCtor.DoCreateAnEditor(
    var Editor: TssnEditor;
    ParentCtrl: TWinControl
): Integer;
begin
    Editor := TssnRichEditor.Create(ParentCtrl);
    Result := Integer(Editor <> nil);
end;
```



到此，编辑器组件的构造器已经实现完成。它能带来什么好处也许有些读者还没有直观地了解到。下面就来看一下。

先声明一个全局的构造器对象：

```
var
    g_EditorCtor : TssnEditorCtor = nil;
```

最后在程序初始化时，创建其对象实例：

```
g_EditorCtor := TssnMemoEditorCtor.Create();
```

然后，调用 g\_EditorCtor.CreateAnEditor 就可以创建出一个 TssnMemoEditor 的实例，正如前面在 TssnWorkspace 的构造函数中的实现一样：

```
constructor TssnWorkspace.Create(
    ParentCtrl : TWinControl;
    FileName : String;
    nIndex : Integer
);
```



```
begin
    // 调用 g_EditorCtor.CreateAnEditor 创建一个 TssnMemoEditor 的实例
    g_EditorCtor.CreateAnEditor(m_Editor, ParentCtrl);

    if FileName <> '' then
        m_Editor.LoadFromFile(FileName);

    m_Index := nIndex;
end;
```

现在假想用 **TssnRichEditor** 了，这时只需要将构造器的创建代码：

```
g_EditorCtor := TssnMemoEditorCtor.Create();
```

改为：

```
g_EditorCtor := TssnRichEditorCtor.Create();
```

其他代码都不需要做任何修改，所有的 `g_EditorCtor.CreateAnEditor()` 调用将创建出 **TssnRichEditor** 的实例，这对于组件风格的更改来说，工作量会非常少。

如果在程序中创建多个 **TssnEditorCtor** 对象，而它们是不同的风格的编辑器组件构造器，就可以在程序中选择性地创建不同编辑器。如声明如下 3 个全局对象：

```
var
    // 统一的“构造器”指针
    g_EditorCtor : TssnEditorCtor = nil;

    // 两种风格的“构造器”
    g_MemoEditorCtor : TssnMemoEditorCtor = nil;
    g_RichEditorCtor : TssnRichEditorCtor = nil;
```

并实际创建两种风格的构造器对象：

```
g_MemoEditorCtor := TssnMemoEditorCtor.Create();
g_RichEditorCtor := TssnRichEditorCtor.Create();
```

然后将 `g_EditorCtor` 动态地指向两个构造器对象中的任何一个，就可以动态改变其所创建的编辑器组件实例的风格：

```
g_EditorCtor := g_MemoEditorCtor;
```

或

```
g_EditorCtor := g_RichEditorCtor;
```

程序的其他部分仍然使用 `g_EditorCtor`（类型为“构造器”的抽象类 `TssnEditorCtor`）作为构造器的操作接口，这样程序的灵活性就大大增强了。

好了，编辑器的构造器已经实现完成。`TssnEditorCtor/TssnMemoEditorCtor/TssnRichEditorCtor` 3 个类实现在同一个单元中，单元文件名称为 `EditorCtor.pas`。下面给出该单元的代码清单：

```
unit EditorCtor;

interface

uses stdctrls, controls,
    Editor;

type
    TssnEditorCtor = class
    protected
        function DoCreateAnEditor(
            var Editor : TssnEditor;
            ParentCtrl : TWinControl
        ) : Integer; virtual; abstract;
    public
        function CreateAnEditor(
            var Editor : TssnEditor;
            ParentCtrl : TWinControl
        ) : Integer;
    end;

    TssnMemoEditorCtor = class(TssnEditorCtor)
    protected
        function DoCreateAnEditor(
            var Editor : TssnEditor;
            ParentCtrl : TWinControl
        ) : Integer; override;
    end;

    TssnRichEditorCtor = class(TssnEditorCtor)
    protected
        function DoCreateAnEditor(
            var Editor : TssnEditor;
            ParentCtrl : TWinControl
        ) : Integer; override;
```



```
end;

implementation

uses MemoEditor, RichEditor, SettingMgr, GlobalObject;

{ TssnMemoEditorCtor }

function TssnMemoEditorCtor.DoCreateAnEditor(
    var Editor: TssnEditor;
    ParentCtrl : TWinControl
): Integer;
begin
    Editor := TssnMemoEditor.Create(ParentCtrl);
    Result := Integer(Editor <> nil);
end;

{ TssnEditorCtor }

function TssnEditorCtor.CreateAnEditor(
    var Editor: TssnEditor;
    ParentCtrl: TWinControl
): Integer;
begin
    Result := DoCreateAnEditor(Editor, ParentCtrl);
    if Boolean(Result) then
        Editor.SetFont(g_SettingMgr.GetDefaultFont());
end;

{ TssnRichEditorCtor }

function TssnRichEditorCtor.DoCreateAnEditor(
    var Editor: TssnEditor;
    ParentCtrl: TWinControl
): Integer;
begin
    Editor := TssnRichEditor.Create(ParentCtrl);
    Result := Integer(Editor <> nil);
end;

end.
```

## 7.4.2 TssnWorkSpaceMgrCtor/TssnTabWorkSpaceMgrCtor

另一个需要构造器的地方是工作区管理器，它负责创建不同界面风格的工作区。TssnWorkSpaceMgrCtor 与 7.4.1 节已经实现了的 TssnEditorCtor 同属于构造器模式，因此它们非常相似。只不过 TssnWorkSpaceMgrCtor 只需要声明一个抽象虚方法 CreateAWorkSpaceMgr()即可：

```
TssnWorkSpaceMgrCtor = class
public
    function CreateAWorkSpaceMgr (
        var WorkSpaceMgr : TssnWorkSpaceMgr;
        ParentCtrl : TWinControl
    ) : Integer; virtual; abstract;
end;
```

因此，TssnWorkSpaceMgrCtor 作为一个只有一个抽象虚方法的抽象类，没有任何实现代码。真正创建一个工作区管理器的任务由其派生类来实现。

在这之前只用 TPageControl 实现了分页式的工作区管理器 TssnTabWorkSpaceMgr，在此处也只实现这种工作区的构造器——TssnTabWorkSpaceMgrCtor。

TssnTabWorkSpaceMgrCtor 只需 override 其基类的 CreateAWorkSpaceMgr()抽象虚方法即可：

```
TssnTabWorkSpaceMgrCtor = class (TssnWorkSpaceMgrCtor)
public
    function CreateAWorkSpaceMgr (
        var WorkSpaceMgr : TssnWorkSpaceMgr;
        ParentCtrl : TWinControl
    ) : Integer; override;
end;
```

其实现也很简单：

```
function TssnTabWorkSpaceMgrCtor.CreateAWorkSpaceMgr (
    var WorkSpaceMgr: TssnWorkSpaceMgr;
    ParentCtrl: TWinControl
): Integer;
begin
    // 创建 TssnTabWorkSpaceMgr 实例
    WorkSpaceMgr := TssnTabWorkSpaceMgr.Create (ParentCtrl);
```



```
Result := Integer(WorkSpaceMgr <> nil);  
end;
```

这么几行代码就完成了工作区管理器的构造器。与编辑器组件的构造器不同的是，由于工作区管理器在整个系统中只会被创建一次，所以不用将该构造器声明为全局对象，而是将“工作区管理器”对象声明为全局的：

```
var  
    g_WorkSpaceMgr : TssnWorkSpaceMgr = nil;
```

然后，在全局初始化函数中用临时创建出的构造器创建出工作区管理器的实例对象：

```
function InitObjects() : Integer;  
var  
    WorkspaceMgrCtor : TssnWorkSpaceMgrCtor;  
begin  
    // ..... 其他初始化代码  
  
    // 首先创建主 Form  
    Application.CreateForm(TMainForm, g_MainForm);  
    // 创建“构造器”对象  
    WorkspaceMgrCtor := TssnTabWorkSpaceMgrCtor.Create();  
    // 创建全局的“工作区管理器”对象实例  
    WorkspaceMgrCtor.CreateAWorkSpaceMgr(  
        g_WorkSpaceMgr,  
        g_MainForm.pnl_WorkSpace  
    );  
    // 销毁“构造器对象”  
    WorkspaceMgrCtor.Free();  
  
    // ..... 其他初始化代码  
end;
```

TssnWorkSpaceMgrCtor 及其派生类 TssnTabWorkSpaceMgrCtor 实现在同一个单元文件中，单元文件名称为 WorkSpaceMgrCtor.pas。在此给出该单元的代码清单：

```
unit WorkSpaceMgrCtor;  
  
interface  
  
uses Controls,
```

```
WorkspaceMgr;  
  
type  
  TssnWorkspaceMgrCtor = class  
  public  
    function CreateAWorkspaceMgr(  
      var WorkspaceMgr : TssnWorkspaceMgr;  
      ParentCtrl : TWinControl  
    ) : Integer; virtual; abstract;  
  end;  
  
  TssnTabWorkspaceMgrCtor = class(TssnWorkspaceMgrCtor)  
  public  
    function CreateAWorkspaceMgr(  
      var WorkspaceMgr : TssnWorkspaceMgr;  
      ParentCtrl : TWinControl  
    ) : Integer; override;  
  end;  
  
implementation  
  
uses TabWorkspaceMgr;  
  
{ TssnTabWorkspaceMgrCtor }  
  
function TssnTabWorkspaceMgrCtor.CreateAWorkspaceMgr(  
  var WorkspaceMgr: TssnWorkspaceMgr;  
  ParentCtrl: TWinControl  
): Integer;  
begin  
  WorkspaceMgr := TssnTabWorkspaceMgr.Create(ParentCtrl);  
  Result := Integer(WorkspaceMgr <> nil);  
end;  
  
end.
```

至此，“构造器”类也已经全部实现完毕。在此给出“构造器”类图，如图 7.8 所示。

结合图 7.6（编辑器组件/工作区/工作区管理器子系统的类图）与图 7.8（“构造器”类图），可以得到一个更完整一些的系统类图，以便更清晰地理解整个系统各模块关系，如图 7.9 所示。

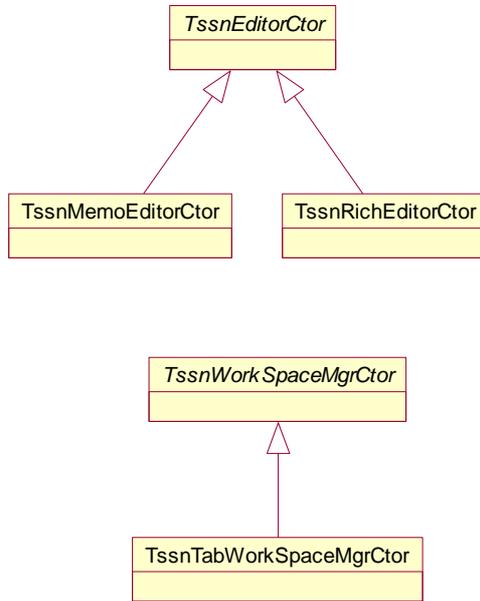


图 7.8 “构造器”类图

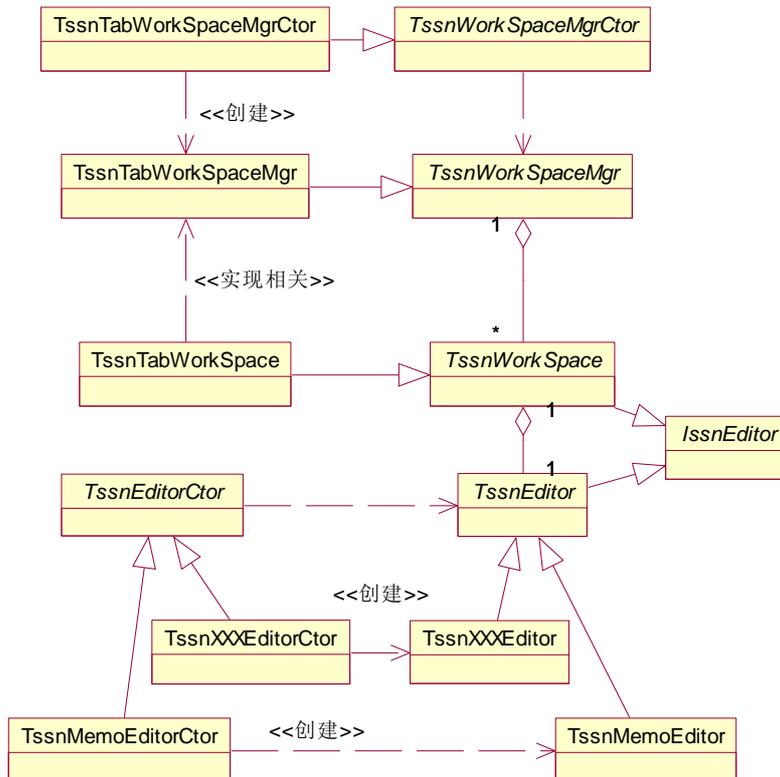


图 7.9 编辑器组件/工作区/工作区管理器子系统与构造器

## 7.5 事件委托 TssnEditorEvent/TssnWorkspaceEvent

已经将所设计的“工作区”的概念模型转变成了可执行的代码，并且似乎已经可以工作（是的，它的确已经可以工作）。

只是，现在的通信状况是多层单向的。也就是说，假设用户在菜单中选择一个操作，界面层会将请求发送给“工作区管理器”（TssnWorkspaceMgr），然后“工作区管理器”将请求转发给“工作区”（TssnWorkspace），而我们知道，“工作区”只是编辑器组件的代理，因此，最终这些请求会被转发给最内层的编辑器组件（TssnEditor）。

问题由此产生了，如此的通信机制，最内层的编辑器组件如何将自己的状态报告给外层呢？

 **注意：**这里的“内层”、“外层”指模块间的逻辑关系，外层为管理者或代理者，内层为响应者或实现者。

当然，可以采用“询问”策略，即由外层定时或不定时地向内层对象查询其状态。但是，决定询问的时机是个难题，因为外层对象是不知道内层的对象何时被改变了状态的。例如，用户没有在编辑器中选定任何文本的时候，界面层菜单中的“复制”、“剪切”等命令选项应该是处于“不可用”状态（没有什么东西可以被复制或剪切）。当用户突然选中了某一段文本时，此时菜单应该立即作出响应，由“不可用”状态变成“可用”状态，否则用户就无法进行所需要的复制或剪切等操作了。可是现在界面层显得比较木讷，因为它根本就不知道编辑器状态变化了，直至它下次去查询编辑器状态的时候。不过，也许那时候编辑器状态再次发生了变化。

因此，需要开辟第二条通信线路，即从内到外的通信，以便内层对象通知外层对象其状态已经发生了改变。

这条通信线路的载体，就是所谓的“事件委托”。通过“事件”这个概念来实现这条由内到外的通信线路。

外层对象内含内层对象，内层对象一般作为外层对象的一个成员而存在。因此外层对象可以看到、访问到内层对象。例如，“工作区管理器”可以很方便地看到并访问自己所管辖的多个工作区对象，而工作区对象可以很方便地看到并访问自己所代理的那个编辑器组件。但是，反之就不同了，内层对象看不到自己被谁所管辖。而且，一般来说需要对状态改变作出响应的，只是界面层，因此也不需要经过多层才将消息传递出来。

由此，可以设想有一个全局的、所有模块都能看到的对象来专门处理这类消息的传送。它保存有外层对象给它的“令牌”，当内部对象状态发生改变并需要通知外层对象时，告诉该对象，由它用“令牌”去敲外层（界面层）对象的门，并通知其作出响应。

好了，这就是所谓的“事件委托”。所谓的“令牌”也就是“事件”。在具体实现中，由外层对象（如界面菜单）首先设置一个回调函数，将函数指针（令牌/事件）交给事件委托对象。内层对象（如编辑器）状态变化后，发出消息告诉事件委托对象，由事件委托对



象调用外层对象的回调函数，以对状态改变作出响应。

### 7.5.1 TssnEditorEvent

编辑器组件需要事件委托，其需要委托的事件只有一种，即编辑器状态的更新。当其任何一个状态发生改变，都通过一个事件通知界面层作出响应。当然，也可以将每个状态的改变拆分成多个事件。不过，那样只会增加程序逻辑的复杂度，却不会带来任何好处。因为事件拆分过细，会导致界面层需提供多个回调函数分别对不同的事件作出响应。一般将事件拆分一般有两个原因：1. 为了效率，当同一个回调函数处理各种不同的事件会影响到执行效率时，就应该拆分；2. 若干事件之间逻辑上的区别非常明显，完全是不同种类的事件，需要作出响应的那部分程序无法对不同事件作出相同的反应，则也应该拆分。

在此处，编辑器组件的状态只包括了是否允许复制、剪切等，而且逻辑上同属于编辑器的选择状态，界面模块的响应程序可以对这类事件做同一件事情——取编辑器的选中状态，刷新菜单。因此可将它们归到同一个事件中：**OnEditorSelectionChange**。

下面定义一个事件委托类 **TssnEditorEvent**。

由于它只管理一个事件，因此只有两个 **public** 的方法，一个由界面层调用设置其回调函数指针，另一个由编辑器组件调用以通知其状态更新了。当然，其内部还有一个 **private** 的函数指针类型的数据成员，以保存界面层提供给它的回调函数指针：

```
TssnEditorEvent = class
private
    // 回调函数指针
    m_OnEditorChange : TNotifyEvent;
public
    procedure OnEditorSelectionChange(Sender : TObject);
    procedure SetOnEditorSelectionChange(Value : TNotifyEvent);
end;
```

界面层模块通过调用 **SetOnEditorSelectionChange()**方法来设置回调函数指针（即前面所谓的“令牌”）：

```
procedure TssnEditorEvent.SetOnEditorSelectionChange(
    Value : TNotifyEvent);
begin
    m_OnEditorChange := Value;
end;
```

当编辑器的可编辑状态发生变化时，调用 **OnEditorChange()**来通知 **TssnEditorEvent** 类的对象，然后由 **TssnEditorEvent** 的实例对象调用回调函数通知界面模块作出响应：

```
procedure TssnEditorEvent.OnEditorSelectionChange(Sender : TObject);
begin
    if Assigned(m_OnEditorChange) then
        m_OnEditorChange(Sender); // 调用回调函数
end;
```

该类的实现就是如此简单，毕竟其只是作为一个中介，完成委托职责而已。下面将 `TssnEditorEvent` 的实例对象声明为全局对象：

```
var
    g_EditorEvent : TssnEditorEvent = nil;
```

并在全局初始化函数中创建它：

```
function InitObjects() : Integer;
begin
    // ..... 其他的代码
    g_EditorEvent := TssnEditorEvent.Create();
    // ..... 其他的代码
end;
```

并在界面层模块初始化时，为 `g_EditorEvent` 设置回调函数指针：

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
    // 设置回调函数，参数中的 OnEditorChange 为界面模块的一个函数指针
    g_EditorEvent.SetOnEditorSelectionChange(OnEditorChange);
    // ..... 其他代码
end;
```

参数中的 `OnEditorChange` 为界面模块的一个函数指针，在 `OnEditorChange` 函数中，将完成获取编辑器选择状态的操作，以此更新相应菜单（如复制、剪切等）状态，决定其是否可用。

然后，在 `TssnEditor`（及其派生类）的会导致编辑器中选择状态变化的时机中（如 `OnMouseDown`、`OnKeyUp`、`OnChange` 等）都加入这样的代码：

```
g_EditorEvent.OnEditorSelectionChange(Sender);
```

这样，编辑器与界面层的第 2 条通信线路就借助它们的事件委托 `TssnEditorEvent` 建立起来了。

图 7.10 演示了界面模块与编辑器组件子系统（`TssnEditor` 及其派生类）之间的通信模式。

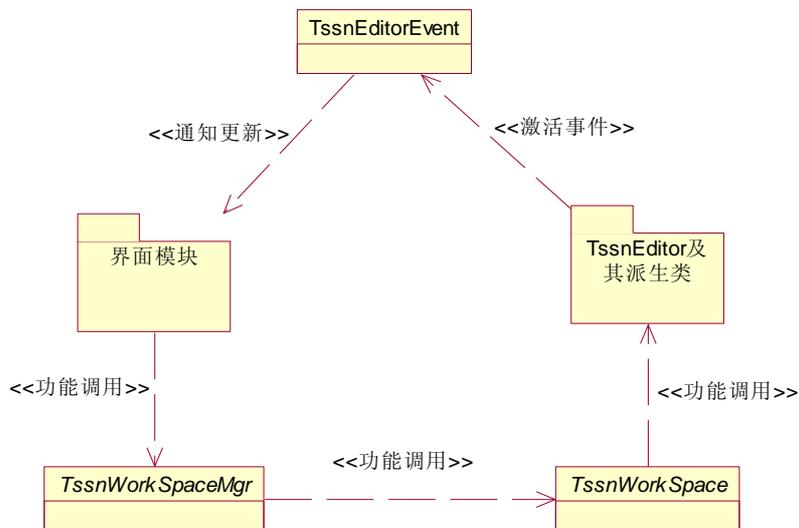


图 7.10 界面模块与编辑器组件子系统（TssnEditor 及其派生类）之间的通信模式

TssnEditorEvent 实现在 EditorEvent.pas 单元中。该单元的代码清单如下：

```
unit EditorEvent;  
  
interface  
  
uses Classes;  
  
type  
  TssnEditorEvent = class  
  private  
    m_OnEditorChange : TNotifyEvent;  
  public  
    procedure OnEditorSelectionChange(Sender : TObject);  
    procedure SetOnEditorSelectionChange(Value : TNotifyEvent);  
  end;  
  
implementation  
  
{ TssnEditorEvent }  
  
procedure TssnEditorEvent.OnEditorSelectionChange(Sender : TObject);  
begin  
  if Assigned(m_OnEditorChange) then  
    m_OnEditorChange(Sender);  
end;
```

```
procedure TssnEditorEvent.SetOnEditorSelectionChange(Value :  
TNotifyEvent);  
begin  
    m_OnEditorChange := Value;  
end;  
  
end.
```

## 7.5.2 TssnWorkspaceEvent

同样，工作区状态也会变化（比如新打开一个工作区、关闭了一个工作区等），工作区对象也需要和界面层模块交互。

由于前面已经有了 `TssnEditorEvent` 的设计经验，而设计 `TssnWorkspaceEvent` 也是类似的工作，因此很容易可以知道该如何去做。

首先，确定工作区状态发生变化的种类：一类是工作区数量发生了变化，如打开、关闭一个或多个工作区，例如打开的工作区数目变为 0 时，必须使菜单中如“关闭工作区”等菜单项变为不可用状态；另一类是当前工作区被改变了，如用户激活了另一个工作区，那么状态栏上的文件名必须做出相应的改变。这两类事件对于界面层来说，是无法以一种方式来处理的。

因此，决定为 `TssnWorkspaceEvent` 定义两个事件，于是 `TssnWorkspaceEvent` 应是这样的：

```
TssnWorkspaceEvent = class  
private  
    m_OnWorkspaceOpenClose : TNotifyEvent;  
    m_OnWorkspaceChange : TNotifyEvent;  
  
public  
    procedure OnWorkspaceOpenClose(Sender : TObject);  
    procedure OnWorkspaceChange(Sender : TObject);  
  
    procedure SetOnWorkspaceOpenClose(Value : TNotifyEvent);  
    procedure SetOnWorkspaceChange(Value : TNotifyEvent);  
end;
```

该类的实现更没有什么特殊之处，与 `TssnEditorEvent` 的实现完全类似：

```
procedure TssnWorkspaceEvent.OnWorkspaceChange(Sender: TObject);
```



```
begin
    if Assigned(m_OnWorkspaceChange) then
        m_OnWorkspaceChange(Sender);
end;

procedure TssnWorkspaceEvent.OnWorkspaceOpenClose(Sender: TObject);
begin
    if Assigned(m_OnWorkspaceOpenClose) then
        m_OnWorkspaceOpenClose(Sender);
end;

procedure TssnWorkspaceEvent.SetOnWorkspaceChange(Value:
TNotifyEvent);
begin
    m_OnWorkspaceChange := Value;
end;

procedure TssnWorkspaceEvent.SetOnWorkspaceOpenClose(Value:
TNotifyEvent);
begin
    m_OnWorkspaceOpenClose := Value;
end;
```

界面层通过调用 `SetOnWorkspaceXXX()` 方法设置响应事件的回调函数指针，而工作区管理器对象与工作区对象调用 `OnWorkspaceXXX()` 方法来激活事件。

该事件委托类的使用方法和 `TssnEditorEvent` 类似，在此不再赘述。

图 7.11 展示了界面模块与工作区子系统（工作区管理器/工作区）之间通过事件委托 `TssnWorkspaceEvent` 的通信模式。

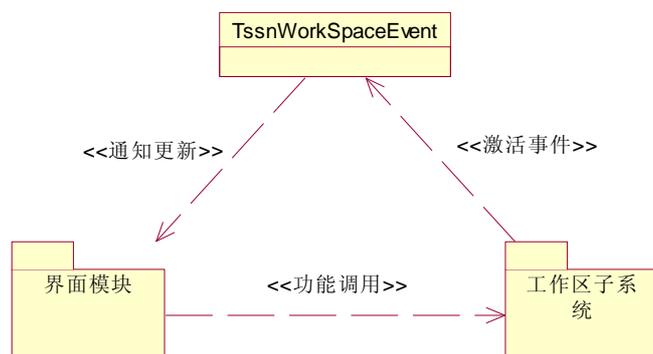


图 7.11 界面模块与工作区子系统（工作区管理器/工作区）之间的通信模式

`TssnWorkspaceEvent` 实现在 `WorkspaceEvent.pas` 单元中。该单元的代码清单如下：

```
unit WorkspaceEvent;

interface

uses Classes;

type
  TssnWorkspaceEvent = class
  private
    m_OnWorkspaceOpenClose : TNotifyEvent;
    m_OnWorkspaceChange : TNotifyEvent;
  public
    procedure OnWorkspaceOpenClose(Sender : TObject);
    procedure OnWorkspaceChange(Sender : TObject);

    procedure SetOnWorkspaceOpenClose(Value : TNotifyEvent);
    procedure SetOnWorkspaceChange(Value : TNotifyEvent);
  end;

implementation

{ TssnWorkspaceEvent }

procedure TssnWorkspaceEvent.OnWorkspaceChange(Sender: TObject);
begin
  if Assigned(m_OnWorkspaceChange) then
    m_OnWorkspaceChange(Sender);
end;

procedure TssnWorkspaceEvent.OnWorkspaceOpenClose(Sender: TObject);
begin
  if Assigned(m_OnWorkspaceOpenClose) then
    m_OnWorkspaceOpenClose(Sender);
end;

procedure TssnWorkspaceEvent.SetOnWorkspaceChange(Value:
TNotifyEvent);
begin
  m_OnWorkspaceChange := Value;
end;

procedure TssnWorkspaceEvent.SetOnWorkspaceOpenClose(Value:
```



```
TNotifyEvent);  
begin  
    m_OnWorkSpaceOpenClose := Value;  
end;  
  
end.
```

## 7.6 其他模块

至此，Sunny SmartNote 的核心构架已经设计、实现完成了，余下的还有一些辅助功能的模块。

### 7.6.1 默认设置管理

每个应用软件都会有功能上的选项允许用户设置，而这些设置可能会涉及各个方面，如何在程序中有序地对这些选项设置进行管理是非常重要的。

可以将默认的选项设置的管理逻辑抽象出来，形成一个独立的类，由它负责集中管理。其职责包括：

- (1) 向其他模块提供某一个选项的默认设置值。
- (2) 接受新的某选项默认设置值。
- (3) 在应用程序启动时，从外部（磁盘或注册表）取得所有选项的默认设置值。
- (4) 在应用程序退出时保存所有设置的值到外部（磁盘或注册表）。

这个抽象称为 **TssnSettingMgr**。

这个类的设计比较直观。

首先考虑算法，在程序启动时，将所有选项的默认值从外部（磁盘）读取到内存中。其他模块请求获得某个值时，将内存中的该值返回给它；其他模块请求更新某个值时，更新内存中的数据。在需要的时候（程序退出时或更新过某个值时）将所有设置的值保存到外部（磁盘或注册表）。

然后考虑实现，只需为每一个选项提供一个内部的数据成员保存其值，并提供这些值的访问接口。在该类被初始化时（程序启动时），从外部（磁盘或注册表）取得默认设置值来初始化这些数据成员；在该类被析构时（程序退出时），将这些数据成员的值保存到外部（磁盘或注册表）。另外，客户程序在设置某一个选项的默认值时，可能要求立刻保存到外部（磁盘或注册表）以增加安全性。**TssnSettingMgr** 也应该能满足这个需求。

Sunny SmartNote 的开放源码版本中，为简化起见，选项设置模块只允许设置工作区的默认字体。

根据以上的设计思想，**TssnSettingMgr** 看起来会是这样的：

```

TssnSettingMgr = class
protected
    // 是否在设置选项默认值时，立刻保存到外部（磁盘或注册表）
    m_SaveWhenSet : Boolean;

    // 设置的名称，其意义依赖不同实现而不同
    // 例如，如果保存设置的是磁盘文件，则该名称可被用作文件名
    // 如果保存设置到注册表，则该名称可被用作键名
    m_SettingName : String;

    // 保存默认字体的数据成员
    m_Font : TFont;
    // 如果有其他选项，可加入更多数据成员

    // 从外部（磁盘或注册表）读取所有默认设置值
    procedure LoadSettings(); virtual;
    // 保存所有默认设置值到外部（磁盘或注册表）
    procedure SaveSettings(); virtual;

public
    procedure SetDefaultFont(Font : TFont); // 设置默认字体的方法
    procedure GetDefaultFont(var Font : TFont); // 取得默认字体的方法
    // 如果有其他选项，可加入更多的方法

    constructor Create(SettingName : String; SaveWhenSet : Boolean);
    destructor Destroy(); override;
end;

```

构造函数传入 2 个参数，一个为“设置的名称”，一个指定是否需要在设置默认值时即真正地保存到外部（磁盘或注册表）。所谓“设置的名称”，依赖于不同的实现，其意义也不同。在 Sunny SmartNote 中，TssnSettingMgr 被实现为将所有设置值保存到磁盘文件，此时“设置的名称”即为磁盘文件名。TssnSettingMgr 也允许被派生以提供其他方式的实现，比如可以实现一个将设置值保存到注册表的派生类，那么，此时“设置的名称”可被用作注册表的键名。构造函数的实现代码如下：

```

constructor TssnSettingMgr.Create(SettingName: String;
    SaveWhenSet : Boolean);
begin
    m_SettingName := SettingName;
    m_SaveWhenSet := SaveWhenSet;
    m_Font := TFont.Create();;

```



```
// 读取所有设置
LoadSettings();
end;
```

析构函数中，只需要将所有设置保存即可：

```
destructor TssnSettingMgr.Destroy;
begin
    // 保存所有设置
    SaveSettings();
    m_Font.Free();
end;
```

在 `protected` 节中，可以看到有两个方法——`LoadSettings()`和 `SaveSettings`，它们执行具体的读取与保存设置的动作。在此，将它声明为 `virtual`；，意味着在 `TssnSettingMgr` 中将为它们提供一种默认的实现——从磁盘文件存取。当然，`TssnSettingMgr` 的派生类有机会 `override` 这两个函数，以提供不同的存取实现，如保存到系统注册表。

它们的默认实现如下：

```
procedure TssnSettingMgr.LoadSettings;
var
    IniFile : TIniFile;
begin
    IniFile := TIniFile.Create(GetExePath() + m_SettingName + '.ini');

    // 字体的选项默认值读取
    m_Font.Name := IniFile.ReadString(
        'Editor',
        'FontName',
        'Comic Sans MS'
    );
    m_Font.Size := IniFile.ReadInteger('Editor', 'FontSize', 10);
    m_Font.Color := IniFile.ReadInteger('Editor', 'FontColor',
    clBlack);
    // 如果有其他选项的话，可在此加入 .....

    IniFile.Free();
end;

procedure TssnSettingMgr.SaveSettings;
var
```

```

    IniFile : TIniFile;
begin
    IniFile := TIniFile.Create(GetExePath() + m_SettingName + '.ini');

    // 字体的选项默认值保存
    IniFile.WriteString('Editor', 'FontName', m_Font.Name);
    IniFile.WriteInteger('Editor', 'FontSize', m_Font.Size);
    IniFile.WriteInteger('Editor', 'FontColor', m_Font.Color);
    // 如果有其他的选项的话, 可在此加入 .....
    IniFile.Free();
end;
```

接着, 就是具体每个选项的访问方法了。默认字体的选项的默认值访问实现如下 (由于 TFont 类型属于复杂类型/类类型, 因此其赋值必须通过 TPersistent 的 Assign() 方法进行):

```

procedure TssnSettingMgr.GetDefaultFont(var Font : TFont);
begin
    Result.Assign(m_Font);
end;

procedure TssnSettingMgr.SetDefaultFont(Font: TFont);
begin
    m_Font.Assign(Font);

    // 如果需要立即保存
    if m_SaveWhenSet then
        SaveSettings();
end;
```

获取默认字体的方法之所以通过参数而不通过返回值返回, 是由于

```
Result := m_Font;
```

将导致外部程序可以直接更改 TssnSettingMgr 内部的 m\_Font, 从而失去类的壁垒的保护。不过, 对于诸如 Integer、Boolean 等简单类型, 则可以直接通过返回值返回, 那样也是安全的。

在此, 只提供字体选项的默认值管理。当然, 如果需求需要更多的选项, 可以用和字体选项完全类似的方式加入更多选项的管理。

TssnSettingMgr 实现在 SettingMgr.pas 单元中。在此给出该单元的代码清单:

```
unit SettingMgr;
```



```
interface

uses Graphics, IniFiles;

type
  TssnSettingMgr = class
  private
    m_SaveWhenSet : Boolean;

  protected
    m_SettingName : String;
    m_Font : TFont;

    procedure LoadSettings(); virtual;
    procedure SaveSettings(); virtual;

  public
    procedure SetDefaultFont(Font : TFont);
    function GetDefaultFont() : TFont;

    constructor Create(SettingName : String; SaveWhenSet : Boolean);
    destructor Destroy(); override;
  end;

implementation

uses ssnPublic;

{ TssnSettingMgr }

constructor TssnSettingMgr.Create(SettingName: String; SaveWhenSet :
Boolean);
begin
  m_SettingName := SettingName;
  m_SaveWhenSet := SaveWhenSet;
  m_Font := TFont.Create();;
  LoadSettings();
end;

destructor TssnSettingMgr.Destroy;
begin
  SaveSettings();
```

```
    m_Font.Free();
end;

function TssnSettingMgr.GetDefaultFont: TFont;
begin
    Result := m_Font;
end;

procedure TssnSettingMgr.LoadSettings;
var
    IniFile : TIniFile;
begin
    IniFile := TIniFile.Create(GetExePath() + m_SettingName + '.ini');

    // Font
    m_Font.Name := IniFile.ReadString(
        'Editor',
        'FontName',
        'Comic Sans MS'
    );
    m_Font.Size := IniFile.ReadInteger('Editor', 'FontSize', 10);
    m_Font.Color := IniFile.ReadInteger('Editor', 'FontColor',
    clBlack);

    IniFile.Free();
end;

procedure TssnSettingMgr.SaveSettings;
var
    IniFile : TIniFile;
begin
    IniFile := TIniFile.Create(GetExePath() + m_SettingName + '.ini');

    IniFile.WriteString('Editor', 'FontName', m_Font.Name);
    IniFile.WriteInteger('Editor', 'FontSize', m_Font.Size);
    IniFile.WriteInteger('Editor', 'FontColor', m_Font.Color);

    IniFile.Free();
end;

procedure TssnSettingMgr.SetDefaultFont(Font: TFont);
begin
```



```
m_Font.Assign(Font);  
if m_SaveWhenSet then  
    SaveSettings();  
end;  
  
end.
```

## 7.6.2 用户交互

应用程序少不了要和用户交互，于是程序中会使用大量的各种对话框。Windows 系统 API 也为用户提供了标准的必备的对话框。一般情况下，系统提供的标准对话框已经可以满足用户的需要。但有时随着应用程序版本的升级，可能会不满足于使用标准对话框，或者软件界面风格发生变化，如果希望所有对话框风格能够与软件主界面一起变化以使得整体界面更为协调、统一，那么就可能需要重写对话框部分相关的代码。

如何降低这种需求的变化给代码带来的影响呢？是的，答案还是一——抽象！

于是，又抽象出一个对话框的逻辑的类——TssnInterActive。它负责定义一套对话框使用规范，对外提供各种对话框（打开文件、文件另存为、查找、替换等），而具体每种风格的对话框由其派生类实现。当然，可以在 TssnInterActive 中给出使用 Windows 标准对话框的默认实现，这样当应用程序使用 Windows 标准对话框时（大多数情况下），就不必为其写派生类而可以直接使用 TssnInterActive 了。

为每种对话框提供一个方法以供客户程序调用——MessageBox（消息框）、ShowFontDlg（字体选择对话框）、ShowSaveDlg（保存文件对话框）、ShowOpenDlg（打开文件对话框）、ShowFindDlg（查找对话框）、ShowReplaceDlg（替换对话框）。

由于在显示某些对话框时，可能需要指定其父窗口的句柄（如 MessageBox），因此在构造该类的实例时，由外部传入一个主窗口的句柄并保存在一个数据成员中。另外，TssnInterActive 还有几个数据成员，它们分别是对应于每种对话框的 VCL 库中的组件类实例，如 TSaveDialog、TOpenDialog、TFontDialog、TFindDialog、TreplaceDialog。使用这些组件来实现 TssnInterActive。

在这些对话框中，TFindDialog 与 TReplaceDialog 比较特殊，它们是非模态的，其流程和事件有关。例如，TFindDialog 的对话框显示后，主线程并不阻塞仍可以继续工作，直到用户单击了“查找下一个”按钮触发 OnFind 事件或者单击“取消”按钮而关闭对话框。因此，需要为 TFindDialog 与 TReplaceDialog 提供一套回调机制。可以在 TssnInterActive 内部设置两个函数指针类型的数据成员，在客户程序调用 ShowFindDlg 或 ShowReplaceDlg 时，传入回调函数指针并保存在这两个数据成员中，在对话框组件的 OnFind 与 OnReplace 等事件被触发时，调用客户程序的回调函数即可。

首先需要定义这两个回调函数的原型：

```
type
```

```

TssnOnFindEvent = procedure (FindText : String; Options :
TFindOptions)
  of Object;
TssnOnReplaceEvent = procedure (FindText, ReplaceText : String;
Options : TFindOptions) of Object;

```

然后就可以把类的设计转成声明代码:

```

TssnInterActive = class
private
  m_hWnd : HWND; // 构造函数传入的主窗口句柄
  m_SaveDlg : TSaveDialog; // “保存文件”对话框
  m_FontDlg : TFontDialog; // “字体选择”对话框
  m_OpenDlg : TOpenDialog; // “打开文件”对话框
  m_FindDlg : TFindDialog; // “查找”对话框
  m_ReplaceDlg : TReplaceDialog; // “替换”对话框

  m_OnFindCallBack : TssnOnFindEvent; // “查找”的回调函数指针
  m_OnReplaceCallBack : TssnOnReplaceEvent; // “替换”的回调函数指针

  // TFindDialog 的 OnFind 事件函数
  procedure OnFind(Sender : TObject);
  // TReplaceDialog 的 OnReplace 事件函数
  procedure OnReplace(Sender : TObject);
  // TReplaceDialog 的 OnReplaceFind 事件函数
  procedure OnReplaceFind(Sender : TObject);

public
  constructor Create(MainWnd : HWND);
  destructor Destroy(); override;

  function MessageBox(Text, Caption : String; uType : Cardinal) :
Integer;
  virtual;
  function ShowSaveDlg() : string; virtual;
  function ShowFontDlg() : TFont; virtual;
  function ShowOpenDlg() : TStrings; virtual;
  procedure ShowFindDlg(defFindText : String; pfOnFind :
ssnOnFindEvent);
  virtual;
  procedure ShowReplaceDlg(
  defFindText : String;

```



```
    pfOnFind : TssnOnFindEvent;  
    pfOnReplace : TssnOnReplaceEvent  
); virtual;  
end;
```

**TssnInterActive** 会在构造函数中创建出所有对话框的实例并对它们进行初始化。以后每当客户程序需要显示对话框时,由 **TssnInterActive** 的实例调用这些对话框对象的 **Execute()** 方法。当然,这种算法也许显得有些浪费资源(对话框对象在还没有使用时就被创建出来了)。当感到这种算法不佳时,可以随时更改 **TssnInterActive** 的实现,而不会影响整个程序的其他模块。不过,在此还是用这种比较浪费资源的算法来实现。

由此,构造函数/析构函数是这样的:

```
constructor TssnInterActive.Create(MainWnd: HWND);  
begin  
    // 构造函数,初始化对话框实例  
    m_OnFindCallBack := nil;  
    m_hWnd := MainWnd;  
  
    m_SaveDlg := TSaveDialog.Create(nil);  
    m_SaveDlg.Options := [ofOverwritePrompt, ofHideReadOnly];  
  
    m_FontDlg := TFontDialog.Create(nil);  
  
    m_OpenDlg := TOpenDialog.Create(nil);  
    m_OpenDlg.Options := [ofAllowMultiSelect, ofFileMustExist];  
  
    m_FindDlg := TFindDialog.Create(nil);  
    m_FindDlg.OnFind := OnFind; // 设置事件函数  
  
    m_ReplaceDlg := TReplaceDialog.Create(nil);  
    m_ReplaceDlg.OnFind := OnReplaceFind; // 设置事件函数  
    m_ReplaceDlg.OnReplace := OnReplace; // 设置事件函数  
end;  
  
destructor TssnInterActive.Destroy;  
begin  
    m_ReplaceDlg.Free();  
    m_ReplaceDlg := nil;  
  
    m_FindDlg.Free();  
    m_FindDlg := nil;
```

```

m_OpenDlg.Free();
m_OpenDlg := nil;

m_FontDlg.Free();
m_FontDlg := nil;

m_SaveDlg.Free();
m_SaveDlg := nil;

m_hWnd := 0;
end;

```

其余的显示对话框方法，只是简单地调用对话框对象（如 `TFontDialog` 的实例）的 `Execute()` 方法并将用户选择结果通过返回值返回而已。

在此，需要再提一下查找/替换对话框的 `OnFind` 和 `OnReplace` 事件处理。`TFindDialog` 和 `TReplaceDialog` 是非模态的对话框，打开它们后，不会阻塞主线程，直至用户单击了“查找”、“替换”之类的按钮之后，才会调用它们的 `OnFind`、`OnReplace` 之类的事件。

可以看到，在构造函数中有类似这样的代码：

```
m_FindDlg.OnFind := OnFind;
```

此时，就已经将 `TFindDialog` 对象的 `OnFind` 事件指针指向了 `TssnInterActive` 内部的一个 `OnFind` 方法。也就是说，`TFindDialog` 对象的 `OnFind` 事件被接管了下来，可以在 `OnFind` 方法中调用外部模块传进的“查找”事件的回调函数指针：

```

procedure TssnInterActive.OnFind(Sender: TObject);
begin
    // 用户在“查找”对话框中，单击了“查找下一个”按钮之后
    m_FindDlg.CloseDialog();
    if Assigned(m_OnFindCallBack) then // 调用客户程序的回调函数
        m_OnFindCallBack(m_FindDlg.FindText, m_FindDlg.Options);
end;

```

“替换”对话框的 `OnReplace` 和 `OnReplaceFind` 事件的处理方法与此类似，在此不再赘述。

`TssnInterActive` 实现在 `InterActive` 单元中。在此给出该单元的代码清单：

```

unit InterActive;

interface

```



```
uses Windows, Dialogs, Graphics, Classes;

type
  TssnOnFindEvent = procedure (
    FindText : String;
    Options : TFindOptions
  ) of Object;

  TssnOnReplaceEvent = procedure (
    FindText,
    ReplaceText : String;
    Options : TFindOptions
  ) of Object;

  TssnInterActive = class
  private
    m_hWnd : HWND;
    m_SaveDlg : TSaveDialog;
    m_FontDlg : TFontDialog;
    m_OpenDlg : TOpenDialog;
    m_FindDlg : TFindDialog;
    m_ReplaceDlg : TReplaceDialog;

    m_OnFindCallBack : TssnOnFindEvent;
    m_OnReplaceCallBack : TssnOnReplaceEvent;
    procedure OnFind(Sender : TObject);
    procedure OnReplace(Sender : TObject);
    procedure OnReplaceFind(Sender : TObject);

  public
    constructor Create(MainWnd : HWND);
    destructor Destroy(); override;

    function MessageBox(
      Text,
      Caption : String;
      uType : Cardinal
    ) : Integer; virtual;
    function ShowSaveDlg() : string; virtual;
    function ShowFontDlg() : TFont; virtual;
    function ShowOpenDlg() : TStrings; virtual;
```

```
procedure ShowFindDlg(  
    defFindText : String;  
    pfOnFind : TssnOnFindEvent  
); virtual;  
procedure ShowReplaceDlg(  
    defFindText : String;  
    pfOnFind : TssnOnFindEvent;  
    pfOnReplace : TssnOnReplaceEvent  
); virtual;  
end;  
  
implementation  
  
{ TssnInterActive }  
  
constructor TssnInterActive.Create(MainWnd: HWND);  
begin  
    m_OnFindCallBack := nil;  
    m_hWnd := MainWnd;  
  
    m_SaveDlg := TSaveDialog.Create(nil);  
    m_SaveDlg.Options := [ofOverwritePrompt, ofHideReadOnly];  
  
    m_FontDlg := TFontDialog.Create(nil);  
  
    m_OpenDlg := TOpenDialog.Create(nil);  
    m_OpenDlg.Options := [ofAllowMultiSelect, ofFileMustExist];  
  
    m_FindDlg := TFindDialog.Create(nil);  
    m_FindDlg.OnFind := OnFind;  
  
    m_ReplaceDlg := TReplaceDialog.Create(nil);  
    m_ReplaceDlg.OnFind := OnReplaceFind;  
    m_ReplaceDlg.OnReplace := OnReplace;  
end;  
  
destructor TssnInterActive.Destroy;  
begin  
    m_ReplaceDlg.Free();  
    m_ReplaceDlg := nil;  
end;
```



```
m_FindDlg.Free();
m_FindDlg := nil;

m_OpenDlg.Free();
m_OpenDlg := nil;

m_FontDlg.Free();
m_FontDlg := nil;

m_SaveDlg.Free();
m_SaveDlg := nil;

m_hWnd := 0;
end;

function TssnInterActive.MessageBox(
    Text,
    Caption: String;
    uType: Cardinal
): Integer;
begin
    Result := Windows.MessageBox(
        m_hWnd,
        PChar(Text),
        PChar(Caption),
        uType
    );
end;

procedure TssnInterActive.OnFind(Sender: TObject);
begin
    m_FindDlg.CloseDialog();
    if Assigned(m_OnFindCallBack) then
        m_OnFindCallBack(
            m_FindDlg.FindText,
            m_FindDlg.Options
        );
end;

procedure TssnInterActive.OnReplace(Sender: TObject);
begin
    if Assigned(m_OnReplaceCallBack) then
```

```
        m_OnReplaceCallBack(  
            m_ReplaceDlg.FindText,  
            m_ReplaceDlg.ReplaceText,  
            m_ReplaceDlg.Options  
        );  
end;  
  
procedure TssnInterActive.OnReplaceFind(Sender: TObject);  
begin  
    if Assigned(m_OnFindCallBack) then  
        m_OnFindCallBack(  
            m_ReplaceDlg.FindText,  
            m_ReplaceDlg.Options  
        );  
end;  
  
procedure TssnInterActive.ShowFindDlg(  
    defFindText : String;  
    pfOnFind : TssnOnFindEvent  
);  
begin  
    m_OnFindCallBack := pfOnFind;  
    m_FindDlg.FindText := defFindText;  
    m_FindDlg.Execute();  
end;  
  
function TssnInterActive.ShowFontDlg: TFont;  
begin  
    Result := nil;  
    if m_FontDlg.Execute() then  
        Result := m_FontDlg.Font;  
end;  
  
function TssnInterActive.ShowOpenDlg: TStrings;  
begin  
    Result := nil;  
    if m_OpenDlg.Execute() then  
        Result := m_OpenDlg.Files;  
end;  
  
procedure TssnInterActive.ShowReplaceDlg(  
    defFindText: String;  
    pfOnFind: TssnOnFindEvent;
```



```
    pfOnReplace: TssnOnReplaceEvent
);
begin
    m_OnReplaceCallBack := pfOnReplace;
    m_OnFindCallBack := pfOnFind;
    m_ReplaceDlg.Execute();
end;

function TssnInterActive.ShowSaveDlg: string;
begin
    Result := '';
    if m_SaveDlg.Execute() then
        Result := m_SaveDlg.FileName;
end;
end.
```

## 7.6.3 界面模块

前面所有的代码实现了 **Sunny SmartNote** 的功能。只是，至今还没有提到过主界面代码。

现在就来看一下整个程序中最直观、最可视的模块——界面模块。完成了该模块后，程序就处于随时可运行的状态了。

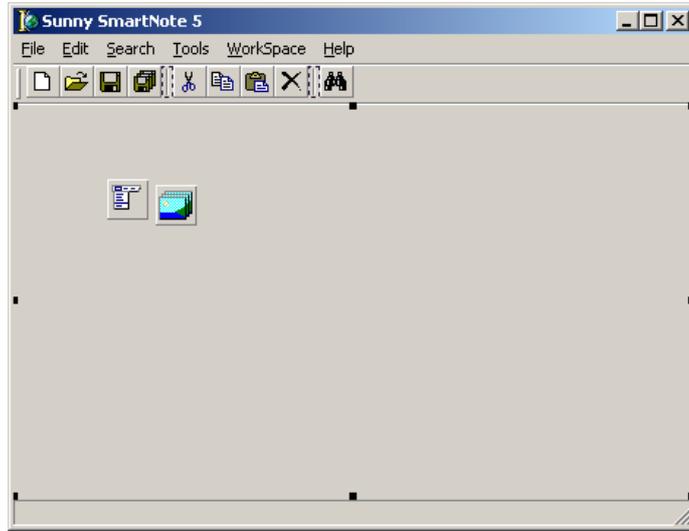
先看一下设计期的界面图，如图 7.12 所示。

界面模块部分，主要是配合我们设计的其他功能模块，将功能以 UI（用户界面）方式展示给用户。这对于习惯于 **Delphi** 这样的可视化开发工具的用户来说，应该是非常熟悉的。因此，不准备浪费口舌详细介绍界面模块的代码。可以如同使用组件一样来使用我们构造的各个子模块，如 **TssnWorkspace**、**TssnWorkspaceMgr**、**TssnInterActive** 等。

在此，只给出典型的几个菜单命令的代码，从中可以体会到界面层编码的简单性，有兴趣的读者，可以在本小节最后或光盘上找到 **Sunny SmartNote** 的全部源代码。

“File” → “New” 菜单项负责新建一个工作区，其只需要调用 **TssnWorkspaceMgr** 对象的 **NewWorkspace()** 方法即可：

```
procedure TMainForm.menu_newClick(Sender: TObject);
begin
    g_WorkSpaceMgr.NewWorkspace(''); // 创建一个新工作区，文件名为空
    UpdateMenuToolBar_WorkSpace(); // 更新工作区相关菜单
    UpdateMenuToolBar_Editor(); // 更新编辑器相关菜单
end;
```



7

图 7.12 设计期界面图

“File” → “Save” 菜单项负责保存当前工作区文本。它首先获得当前激活的工作区对象，然后调用其 Save() 方法进行保存工作。不过，需要将 Save() 方法的调用用 try 块保护起来，因为 Save() 方法可能会引发异常，如文件为只读时，保存会失败。如果保存失败，则提示用户“文件无法保存”，然后自动转为“另存为”功能：

```
procedure TMainForm.menu_saveClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    // 取得当前工作区
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    try
        CurWorkSpace.Save() // 保存
    except
        g_InterActive.MessageBox(
            str_SaveError,
            Application.Title,
            MB_ICONSTOP
        );
        menu_saveas.Click();
    end;
end;
```



“Edit” → “Cut” 菜单项是对当前工作区中的选中文本进行剪切操作。只需先获得当前激活的工作区对象，然后调用其 `Cut()` 方法即可。如前几节所说，工作区对象会将 `Cut()` 方法的调用转发给其内部的编辑器组件。当然，这属于内部实现细节了，界面层的代码编写不需要知道这些：

```
procedure TMainForm.menu_cutClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
Begin
    // 取得当前工作区
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    // 剪切
    CurWorkSpace.Cut();
    // 更新编辑器相关的菜单
    UpdateMenuToolBar_Editor();
end;
```

整个程序中，只有这样一个和 Form 相关的单元，它的 Form 文件名为 `UMainForm.dfm`，代码实现单元的文件名为 `UMainForm.pas`。在此给出 `UMainForm.pas` 的源代码清单：

```
unit UMainForm;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs,
    ComCtrls, ToolWin, ExtCtrls, StdCtrls, Menus, ImgList, Buttons;

type
    TMainForm = class(TForm)
        CoolBar: TCoolBar;
        ToolBar: TToolBar;
        StatusBar: TStatusBar;
        pnl_WorkSpace: TPanel;
        tb_new: TToolButton;
        tb_open: TToolButton;
        MainMenu: TMainMenu;
        menu_file: TMenuItem;
    end;
```

```
menu_new: TMenuItem;
menu_open: TMenuItem;
ImageList: TImageList;
menu_line0: TMenuItem;
menu_save: TMenuItem;
tb_save: TToolButton;
tb_line1: TToolButton;
menu_saveall: TMenuItem;
tb_saveall: TToolButton;
menu_saveas: TMenuItem;
menu_line1: TMenuItem;
menu_close: TMenuItem;
menu_closeall: TMenuItem;
menu_line2: TMenuItem;
menu_exit: TMenuItem;
menu_edit: TMenuItem;
menu_undo: TMenuItem;
menu_redo: TMenuItem;
menu_line3: TMenuItem;
menu_cut: TMenuItem;
menu_copy: TMenuItem;
menu_paste: TMenuItem;
menu_del: TMenuItem;
menu_DeleteSelection: TMenuItem;
menu_DeleteLine: TMenuItem;
tb_cut: TToolButton;
tb_copy: TToolButton;
tb_paste: TToolButton;
tb_del: TToolButton;
menu_line4: TMenuItem;
menu_selectall: TMenuItem;
tb_line2: TToolButton;
tb_find: TToolButton;
menu_search: TMenuItem;
menu_find: TMenuItem;
menu_findnext: TMenuItem;
menu_line5: TMenuItem;
menu_replace: TMenuItem;
menu_tools: TMenuItem;
menu_WorkSpace: TMenuItem;
menu_Help: TMenuItem;
menu_wordcount: TMenuItem;
```





```
menu_line6: TMenuItem;
menu_setting: TMenuItem;
menu_nextworkspace: TMenuItem;
menu_about: TMenuItem;
menu_line7: TMenuItem;
menu_wrap: TMenuItem;
procedure menu_newClick(Sender: TObject);
procedure menu_openClick(Sender: TObject);
procedure menu_saveClick(Sender: TObject);
procedure FormClose(Sender: TObject; var Action: TCloseAction);
procedure menu_saveallClick(Sender: TObject);
procedure menu_saveasClick(Sender: TObject);
procedure menu_closeClick(Sender: TObject);
procedure menu_closeallClick(Sender: TObject);
procedure menu_exitClick(Sender: TObject);
procedure menu_undoClick(Sender: TObject);
procedure menu_redoClick(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure menu_cutClick(Sender: TObject);
procedure menu_copyClick(Sender: TObject);
procedure menu_pasteClick(Sender: TObject);
procedure menu_DeleteSelectionClick(Sender: TObject);
procedure menu_DeleteLineClick(Sender: TObject);
procedure menu_selectallClick(Sender: TObject);
procedure menu_findClick(Sender: TObject);
procedure menu_findnextClick(Sender: TObject);
procedure menu_replaceClick(Sender: TObject);
procedure menu_wordcountClick(Sender: TObject);
procedure menu_nextworkspaceClick(Sender: TObject);
procedure menu_aboutClick(Sender: TObject);
procedure menu_settingClick(Sender: TObject);
procedure menu_wrapClick(Sender: TObject);
private
    m_LastFindText : String;
    m_LastFindOption : TFindOptions;

public
    procedure Init();
    procedure UpdateMenuToolBar_WorkSpace();
    procedure UpdateMenuToolBar_Editor();

    procedure OnEditorChange(Sender : TObject);
```

```
procedure OnWorkSpaceOpenClose(Sender : TObject);
procedure OnWorkSpaceChange(Sender : TObject);
procedure OnFind(FindText : String; Options : TFindOptions);
procedure OnReplace(
    FindText,
    ReplaceText : String;
    Options : TFindOptions
);
end;

implementation

uses GlobalObject, WorkSpace, MultiLan, IntfEditor, ssnPublic;

{$R *.DFM}

procedure TMainForm.menu_newClick(Sender: TObject);
begin
    g_WorkSpaceMgr.NewWorkSpace('');
    UpdateMenuToolBar_WorkSpace();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_openClick(Sender: TObject);
var
    FileList : TStrings;
    i : Integer;
begin
    FileList := g_InterActive.ShowOpenDlg();
    if FileList = nil then
        Exit;

    for i := 0 to FileList.Count - 1 do
        begin
            try
                g_WorkSpaceMgr.NewWorkSpace(FileList[i]);
            except
                g_InterActive.MessageBox(
                    Format(str_LoadError, [FileList[i]]),
                    Application.Title,
                    MB_ICONSTOP
                );
            end;
        end;
    end;
end;
```



```
        end;
    end;

    UpdateMenuToolBar_WorkSpace();
end;

procedure TMainForm.UpdateMenuToolBar_WorkSpace;
var
    bEnable : Boolean;
    nWorkSpaceCount : Integer;
begin
    nWorkSpaceCount := g_WorkSpaceMgr.GetWorkSpaceCount();

    if nWorkSpaceCount > 0 then
        bEnable := true
    else
        bEnable := false;

    menu_save.Enabled := bEnable;
    tb_save.Enabled := bEnable;
    menu_saveas.Enabled := bEnable;
    menu_saveall.Enabled := bEnable;
    tb_saveall.Enabled := bEnable;
    menu_close.Enabled := bEnable;
    menu_closeall.Enabled := bEnable;
    menu_undo.Enabled := bEnable;
    menu_redo.Enabled := bEnable;
    menu_cut.Enabled := bEnable;
    tb_cut.Enabled := bEnable;
    menu_copy.Enabled := bEnable;
    tb_copy.Enabled := bEnable;
    menu_paste.Enabled := bEnable;
    tb_paste.Enabled := bEnable;
    menu_del.Enabled := bEnable;
    tb_del.Enabled := bEnable;
    menu_selectall.Enabled := bEnable;
    menu_find.Enabled := bEnable;
    tb_find.Enabled := bEnable;
    menu_findnext.Enabled := bEnable;
    menu_replace.Enabled := bEnable;
    menu_wordcount.Enabled := bEnable;
    menu_wrap.Enabled := bEnable;
```

```
    if nWorkspaceCount > 1 then
        menu_nextworkspace.Enabled := true
    else
        menu_nextworkspace.Enabled := false;
end;

procedure TMainForm.menu_saveClick(Sender: TObject);
var
    CurWorkspace : TssnWorkspace;
begin
    CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
    if CurWorkspace = nil then
        Exit;
    try
        CurWorkspace.Save()
    except
        g_InterActive.MessageBox(
            str_SaveError,
            Application.Title,
            MB_ICONSTOP
        );
        menu_saveas.Click();
    end;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
    if not g_WorkSpaceMgr.CloseAll() then
        Action := caNone;
end;

procedure TMainForm.menu_saveallClick(Sender: TObject);
begin
    g_WorkSpaceMgr.SaveAll();
end;

procedure TMainForm.menu_saveasClick(Sender: TObject);
var
    CurWorkspace : TssnWorkspace;
begin
```



```
CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
if CurWorkspace = nil then
    Exit;
try
    CurWorkspace.SaveAs()
except
    g_InterActive.MessageBox(
        str_SaveError,
        Application.Title,
        MB_ICONSTOP
    );
end;
end;

procedure TMainForm.menu_closeClick(Sender: TObject);
var
    CurWorkspace : TssnWorkspace;
begin
    CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
    if CurWorkspace = nil then
        Exit;
    g_WorkSpaceMgr.CloseWorkspace(CurWorkspace.GetIndex());
end;

procedure TMainForm.menu_closeallClick(Sender: TObject);
begin
    g_WorkSpaceMgr.CloseAll();
end;

procedure TMainForm.menu_exitClick(Sender: TObject);
begin
    Close();
end;

procedure TMainForm.UpdateMenuToolBar_Editor;
var
    CurWorkspace : TssnWorkspace;
begin
    CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
    if CurWorkspace = nil then
        Exit;
    menu_undo.Enabled := CurWorkspace.CanUndo();
```

```
menu_redo.Enabled := CurWorkspace.CanRedo();
menu_cut.Enabled := CurWorkspace.CanCut();
tb_cut.Enabled := menu_cut.Enabled;
menu_copy.Enabled := CurWorkspace.CanCopy();
tb_copy.Enabled := menu_copy.Enabled;
menu_paste.Enabled := CurWorkspace.CanPaste();
tb_paste.Enabled := menu_paste.Enabled;
menu_deleteselection.Enabled := CurWorkspace.CanDeleteSelection();
tb_del.Enabled := menu_deleteselection.Enabled;
end;

procedure TMainForm.menu_undoClick(Sender: TObject);
var
    CurWorkspace : TssnWorkspace;
begin
    CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
    if CurWorkspace = nil then
        Exit;
    CurWorkspace.Undo();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_redoClick(Sender: TObject);
var
    CurWorkspace : TssnWorkspace;
begin
    CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
    if CurWorkspace = nil then
        Exit;
    CurWorkspace.Redo();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.Init;
begin
    UpdateMenuToolBar_WorkSpace();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
    g_EditorEvent.SetOnEditorSelectionChange(OnEditorChange);
```



```
g_WorkSpaceEvent.SetOnWorkSpaceOpenClose (OnWorkSpaceOpenClose);
g_WorkSpaceEvent.SetOnWorkSpaceChange (OnWorkSpaceChange);
end;

procedure TMainForm.OnEditorChange(Sender: TObject);
begin
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_cutClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    CurWorkSpace.Cut();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_copyClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    CurWorkSpace.Copy();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_pasteClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    CurWorkSpace.Paste();
    UpdateMenuToolBar_Editor();
end;
```

```
procedure TMainForm.menu_DeleteSelectionClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    CurWorkSpace.DeleteSelection();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_DeleteLineClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    CurWorkSpace.DeleteLine();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.OnWorkSpaceOpenClose(Sender: TObject);
begin
    UpdateMenuToolBar_WorkSpace();
end;

procedure TMainForm.menu_selectallClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    CurWorkSpace.SelectAll();
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_findClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
```



```
CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
if CurWorkSpace = nil then
    Exit;
g_InterActive.ShowFindDlg(CurWorkSpace.GetSelectText(), OnFind);
UpdateMenuToolBar_Editor();
end;

procedure TMainForm.OnFind(FindText: String; Options: TFindOptions);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    m_LastFindText := FindText;
    m_LastFindOption := Options;
    if not CurWorkSpace.FindNext(FindText, Options) then
        g_InterActive.MessageBox(
            Format(str_NotFindText, [FindText]),
            Application.Title,
            MB_ICONINFORMATION
        );
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_findnextClick(Sender: TObject);
var
    CurWorkSpace : TssnWorkSpace;
begin
    CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
    if CurWorkSpace = nil then
        Exit;
    if not CurWorkSpace.FindNext(m_LastFindText, m_LastFindOption) then
        g_InterActive.MessageBox(
            Format(str_NotFindText, [m_LastFindText]),
            Application.Title,
            MB_ICONINFORMATION
        );
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.OnReplace(FindText, ReplaceText: String;
```

```
Options: TFindOptions);
var
  CurWorkSpace : TssnWorkSpace;
  ReplaceCount : Integer;
begin
  CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
  if CurWorkSpace = nil then
    Exit;
  ReplaceCount := CurWorkSpace.Replace(FindText, ReplaceText,
Options);
  if frReplaceAll in Options then
    g_InterActive.MessageBox(
      Format(str_ReplacedAll, [ReplaceCount]),
      Application.Title,
      MB_ICONINFORMATION
    )
  else if not CurWorkSpace.FindNext(FindText, Options) then
    g_InterActive.MessageBox(
      Format(str_NotFindText, [m_LastFindText]),
      Application.Title,
      MB_ICONINFORMATION
    );
  UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_replaceClick(Sender: TObject);
var
  CurWorkSpace : TssnWorkSpace;
begin
  CurWorkSpace := g_WorkSpaceMgr.GetActiveWorkSpace();
  if CurWorkSpace = nil then
    Exit;
  g_InterActive.ShowReplaceDlg(
    CurWorkSpace.GetSelectText(),
    OnFind,
    OnReplace
  );
  UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_wordcountClick(Sender: TObject);
var
```



```
CurWorkspace : TssnWorkspace;
CountResult : TssnWordCountRec;
begin
  CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
  if CurWorkspace = nil then
    Exit;
  CountResult := CurWorkspace.GetWordCount();
  g_InterActive.MessageBox(
    String(str_CountResult) + SSN_ENTER_CHAR + SSN_ENTER_CHAR +
    String(str_AnsiChar) + IntToStr(CountResult.AnsiChar) +
    SSN_ENTER_CHAR + String(str_MultiChar) +
    IntToStr(CountResult.MultiChar) + SSN_ENTER_CHAR +
    String(str_NumChar) + IntToStr(CountResult.NumChar) +
    SSN_ENTER_CHAR + String(str_OtherChar) +
    IntToStr(CountResult.Other),
    Application.Title,
    MB_ICONINFORMATION
  );
  UpdateMenuToolBar_Editor();
end;

procedure TMainForm.menu_nextworkspaceClick(Sender: TObject);
begin
  g_WorkSpaceMgr.ActiveNextWorkSpace();
end;

procedure TMainForm.menu_aboutClick(Sender: TObject);
begin
  g_InterActive.MessageBox(
    'Sunny SmartNote 5.0 (OpenSource Edition)' + SSN_ENTER_CHAR +
    SSN_ENTER_CHAR + 'build 2002.5.17' + SSN_ENTER_CHAR +
    SSN_ENTER_CHAR + 'Author : Shen Min' + SSN_ENTER_CHAR +
    'Copyright(c) 1999-2002 by Sunisoft' + SSN_ENTER_CHAR +
    'http://www.sunisoft.com',
    Application.Title,
    MB_ICONINFORMATION
  );
end;

procedure TMainForm.menu_settingClick(Sender: TObject);
var
```

```
    Font : TFont;
begin
    Font := g_InterActive.ShowFontDlg();
    if Font <> nil then
        g_SettingMgr.SetDefaultFont(Font);
end;

procedure TMainForm.menu_wrapClick(Sender: TObject);
var
    CurWorkspace : TssnWorkspace;
begin
    CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
    if CurWorkspace = nil then
        Exit;
    CurWorkspace.SetWordWrap(menu_wrap.Checked);
    UpdateMenuToolBar_Editor();
end;

procedure TMainForm.OnWorkspaceChange(Sender: TObject);
var
    CurWorkspace : TssnWorkspace;
begin
    CurWorkspace := g_WorkSpaceMgr.GetActiveWorkspace();
    if CurWorkspace = nil then
        Exit;
    menu_wrap.Checked := CurWorkspace.GetWordWrap();
    StatusBar.SimpleText := CurWorkspace.GetFileName();
end;

end.
```

#### 7.6.4 其他单元

除了以上各小节给出了每个单元所实现的模块外，该 project 中还有另外几个单元未被提及，它们是 MultiLan.pas、ssnPublic.pas、GlobalObject.pas 以及 project 文件本身 snote.dpr。

MultiLan.pas 单元是为实现多语言版本而准备的。在该单元中，定义了所有程序中使用的多种语言的字符串资源以及一套可以动态指向不同字符串资源的字符指针。在编译时，使用条件编译符决定编译的语言版本，那套字符指针便被初始化为指向所对应的语言的字符串资源。



在上一小节“界面模块”的代码中，也许读者会注意到有类似这样的代码：

```
g_InterActive.MessageBox(  
    Format(str_LoadError, [FileList[i]]),  
    Application.Title,  
    MB_ICONSTOP  
);
```

其中，str\_LoadError 便是在 MultiLan.pas 中定义的字符指针：

```
var  
    str_LoadError : PChar;
```

与该字符指针相关的两个字符串资源的声明：

```
const  
    English_LoadError      = 'The file can''t be opened : %s';  
    Chinese_LoadError     = '打开 %s 文件失败';
```

在编译时根据条件编译符来决定 str\_LoadError 指针指向哪一个字符串。这样，就实现了多语言版本的切换。定义编译条件，可以选择“Project”→“Options”菜单项弹出 Project Options 对话框，其中的“Directories/Conditionals”选项卡中的条件编译符定义中指定，如图 7.13 所示。

另外，值得注意的是，MultiLan.pas 中只定义了通过对话框显示的字符串资源，而没有包括 Form 界面上的字符串资源，如菜单上的字符串。这是因为，对于 Form 上的资源，Delphi 本身已经提供了一种多语言的机制来提供。在 Delphi IDE 的“Project”→“Languages”菜单项中可以自行添加各种语言支持。关于 Delphi 本身对多语言支持的机制不属于本书范围，在此不做更进一步的讨论了。

ssnPublic.pas 单元定义了程序中的公用函数及常量，如获取当前可执行文件所在的路径的函数等。

GlobalObject.pas 单元定义程序中所用到的全局对象以及全局对象的初始化、反初始化函数。前面提到过的“编辑器的构造器”以及“工作区管理器”等全局对象，都是在全局对象初始化函数中被创建的。

在 project 文件的 snote.dpr 中，除了 Delphi 自动生成的 Application 对象初始化代码外，我们加入了对 GlobalObject.pas 单元中的全局对象初始化、反初始化函数的调用，并将反初始化函数放入 finally 块中，使得无论程序发生什么异常，都能保证反初始化函数得到执行。

下面给出这 4 个单元的代码清单。

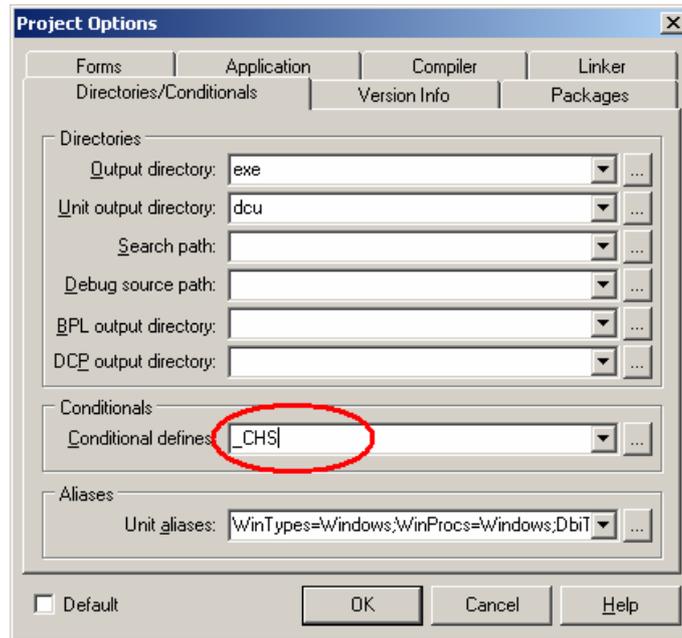


图 7.13 条件编译定义

MultiLan.pas 单元的代码清单:

```

unit MultiLan;

interface

Type
  LanguageType = (LT_English, LT_Chinese);

const
  English_NoTitle      = 'No Title';
  Chinese_NoTitle     = '无标题';

  English_PromptSave  = 'Save changes?';
  Chinese_PromptSave  = '该文件被修改过, 是否保存? ';

  English_SaveError   = 'Save file failed, maybe it's read only';
  Chinese_SaveError   = '保存文件失败, 可能由于该文件为只读文件';

  English_LoadError   = 'The file can't be opened : %s';
  Chinese_LoadError   = '打开 %s 文件失败';

  English_NotFindText = 'Sorry, can't find string "%s"';

```



```
Chinese_NotFindText      = '抱歉, 没有找到 "%s" ';

English_ReplacedAll      = '%d occurrences have been replaced';
Chinese_ReplacedAll      = '%d 处已被替换';

English_CountResult     = 'WordCount Result: ';
Chinese_CountResult     = '字数统计结果: ';

English_AnsiChar        = 'Ansi character: ';
Chinese_AnsiChar        = '英文字符: ';

English_MultiChar       = 'Multibyte character: ';
Chinese_MultiChar       = '中文字符: ';

English_NumChar         = 'Numeric character: ';
Chinese_NumChar         = '数字字符: ';

English_OtherChar       = 'Other character: ';
Chinese_OtherChar       = '其他字符: ';

var
  str_NoTitle,
  str_PromptSave,
  str_SaveError,
  str_LoadError,
  str_NotFindText,
  str_ReplacedAll,
  str_CountResult,
  str_AnsiChar,
  str_MultiChar,
  str_NumChar,
  str_OtherChar : PChar;

implementation

procedure InitLanguage(Language : LanguageType);
begin
  if Language = LT_Chinese then
  begin
    str_NoTitle      := Chinese_NoTitle;
    str_PromptSave   := Chinese_PromptSave;
    str_SaveError    := Chinese_SaveError;
```

```

    str_LoadError      := Chinese_LoadError;
    str_NotFindText    := Chinese_NotFindText;
    str_ReplacedAll    := Chinese_ReplacedAll;
    str_CountResult    := Chinese_CountResult;
    str_AnsiChar       := Chinese_AnsiChar;
    str_MultiChar      := Chinese_MultiChar;
    str_NumChar        := Chinese_NumChar;
    str_OtherChar      := Chinese_OtherChar;
end
else // default is English
begin
    str_NoTitle        := English_NoTitle;
    str_PromptSave     := English_PromptSave;
    str_SaveError      := English_SaveError;
    str_LoadError      := English_LoadError;
    str_NotFindText    := English_NotFindText;
    str_ReplacedAll    := English_ReplacedAll;
    str_CountResult    := English_CountResult;
    str_AnsiChar       := English_AnsiChar;
    str_MultiChar      := English_MultiChar;
    str_NumChar        := English_NumChar;
    str_OtherChar      := English_OtherChar;
end;
end;

initialization
    {$IFDEF _CHS}
    InitLanguage(LT_Chinese);
    {$ELSE}
    InitLanguage(LT_English);
    {$ENDIF}
end.

```

ssnPublic 单元代码清单:

```

unit ssnPublic;

interface

uses SysUtils, Forms;

```



```
const
    SSN_MAX_WORKSPACE    = 128;
    SSN_ENTER_CHAR      = #13 + #10;

    function GetExePath() : String;

implementation

var
    Path_ExePath : String = '';

function GetExePath() : String;
begin
    if Path_ExePath = '' then
    begin
        Path_ExePath := ExtractFilePath(Application.ExeName);
        if Path_ExePath[Length(Path_ExePath)] <> '\' then
            Path_ExePath := Path_ExePath + '\';
        Result := Path_ExePath
    end
    else
        Result := Path_ExePath;
end;

end.
```

GlobalObject.pas 单元的代码清单:

```
unit GlobalObject;

interface

uses Forms, SysUtils,
    WorkspaceMgr, EditorCtor, InterActive, UMainForm, EditorEvent,
    WorkspaceEvent, SettingMgr;

var
    g_EditorCtor : TssnEditorCtor = nil;
    g_WorkSpaceMgr : TssnWorkSpaceMgr = nil;
    g_InterActive : TssnInterActive = nil;
    g_EditorEvent : TssnEditorEvent = nil;
```

```
g_WorkSpaceEvent : TssnWorkSpaceEvent = nil;
g_SettingMgr : TssnSettingMgr = nil;

g_MainForm: TMainForm = nil;

function InitObjects() : Integer;
procedure UnInitObjects();

implementation

uses WorkspaceMgrCtor;

function InitObjects() : Integer;
var
    WorkspaceMgrCtor : TssnWorkSpaceMgrCtor;
begin
    g_SettingMgr := TssnSettingMgr.Create(
        ChangeFileExt(ExtractFileName(Application.ExeName), ''),
        true
    );

    g_EditorEvent := TssnEditorEvent.Create();
    g_WorkSpaceEvent := TssnWorkSpaceEvent.Create();
    Application.CreateForm(TMainForm, g_MainForm);

    WorkspaceMgrCtor := TssnTabWorkSpaceMgrCtor.Create();
    g_EditorCtor := TssnRichEditorCtor.Create();

    WorkspaceMgrCtor.CreateAWorkSpaceMgr(
        g_WorkSpaceMgr,
        g_MainForm.pnl_WorkSpace
    );
    WorkspaceMgrCtor.Free();
    g_InterActive := TssnInterActive.Create(g_MainForm.Handle);
    g_MainForm.Init();
    Result := 1;
end;

procedure UnInitObjects();
begin
    g_InterActive.Free();
```



```
g_InterActive := nil;

g_EditorCtor.Free();
g_EditorCtor := nil;

g_WorkSpaceMgr.Free();
g_WorkSpaceMgr := nil;

g_WorkSpaceEvent.Free();
g_WorkSpaceEvent := nil;

g_EditorEvent.Free();
g_EditorEvent := nil;

g_SettingMgr.Free();
g_SettingMgr := nil;
end;

end.
```

project 文件 snote.dpr 代码清单:

```
program snote;

uses
  Forms,
  UMainForm in 'UMainForm.pas' {MainForm},
  Editor in 'Editor.pas',
  ssnPublic in 'ssnPublic.pas',
  WorkspaceMgr in 'WorkspaceMgr.pas',
  Workspace in 'Workspace.pas',
  TabWorkspaceMgr in 'TabWorkspaceMgr.pas',
  GlobalObject in 'GlobalObject.pas',
  EditorCtor in 'EditorCtor.pas',
  MemoEditor in 'MemoEditor.pas',
  WorkspaceMgrCtor in 'WorkspaceMgrCtor.pas',
  MultiLan in 'MultiLan.pas',
  InterActive in 'InterActive.pas',
  IntfEditor in 'IntfEditor.pas',
  SettingMgr in 'SettingMgr.pas',
  EditorEvent in 'EditorEvent.pas',
  WorkspaceEvent in 'WorkspaceEvent.pas',
```

```

RichEditor in 'RichEditor.pas';

{$R *.RES}

begin
  Application.Initialize;
  try
    InitObjects();
    Application.Title := 'Sunny SmartNote 5';
    Application.Run;
  finally
    UnInitObjects();
  end;
end.

```

## 7.7 光盘上的代码说明

Sunny SmartNote 5.0 开放源代码版本的所有代码都可以在配书光盘的 SmartNote 子目录下找到。注意，要编译运行代码，请确保其目录下存在有 `dcu` 和 `exe` 两个子目录。其中，所有编译中间文件 (\*.dcu) 会被生成在 `dcu` 子目录下，而最终 `build` 而成的可执行文件会被放置在 `exe` 子目录下。所有程序在 Delphi 5/Delphi 6 下编译通过。

在此，给出文件列表以及每个文件的模块说明，如表 7.3 所示。

表 7.3 代码文件与模块说明

文件名	模块说明
snote.dpr 以及 snote.*	项目代码及配置文件
IntfEditor.pas	IssnEditor 接口定义 (详见 7.2.1 节)
Editor.pas	编辑器组件抽象 TssnEditor 的定义与实现 (详见 7.2.2 节)
MemoEditor.pas	以 TMemo 实现的 TssnEditor 派生类的实现 (详见 7.2.3 节)
RichEditor.pas	以 TRichEdit 实现的 TssnEditor 派生类的实现 (详见 7.2.4 节)
WorkSpace.pas	工作区抽象 TssnWorkSpace 的定义与实现 (详见 7.2.5 节)
WorkSpaceMgr.pas	工作区管理器抽象 TssnWorkSpaceMgr 的定义与实现 (详见 7.3.1 节)
TabWorkSpaceMgr.pas	以 TPageControl 实现的 TssnWorkSpaceMgr 与 TssnWorkSpace 的派生类的实现 (详见 7.3.2、7.3.3 节)
EditorCtor.pas	编辑器组件构造器 TssnEditorCtor 及 TssnMemoEditorCtor 的定义与实现 (详见 7.4.1 节)
WorkSpaceMgrCtor.pas	工作区管理器构造器 TssnWorkSpaceMgrCtor 及 TssnTabWorkSpaceMgrCtor 的定义与实现 (详见 7.4.2 节)
EditorEvent.pas	编辑器组件事件委托 TssnEditorEvent 的定义与实现 (详见 7.5.1 节)



续表

文件名	模块说明
WorkspaceEvent.pas	工作区事件委托 TssnWorkspaceEvent 的定义与实现（详见 7.5.2 节）
SettingMgr.pas	选项默认值管理 TssnSettingMgr 的定义与实现（详见 7.6.1 节）
InterActive.pas	用户交互对话框风格管理 TssnInterActive 的定义与实现（详见 7.6.2 节）
MultiLan.pas	多语言支持，字符串资源定义（详见 7.6.4 节）
ssnPublic.pas	公用函数声明与实现（详见 7.6.4 节）
GlobalObject.pas	全局对象、全局变量定义、初始化与反初始化（详见 7.6.4 节）
U MainForm.pas 与 U MainForm.dfm	界面模块（详见 7.6.3 节）

## 7.8 小 结

Sunny SmartNote 5.0 开放源代码版本的代码量并不多，即使作为编辑器软件来说，也还是非常不完善的。

在本章中，没有试图解释代码。解释现存的代码对于作为作者来说可能比较轻松，然而对于读者，却可能没有多少助益，或者有所助益，但这不是本书的目的。本书希望读者能从实现功能的世界中跳出来，开始尝试更多地关注代码设计以至更高层次的系统分析、设计。因此，笔者尝试将程序中的每个模块具体剖析，并详细介绍其设计思路，而将具体的代码解释放到了代码的注释中。

由于是先有代码，后有本章，所以笔者是通过回忆来介绍设计过程。笔者无法保证回忆与真实情况完全相符（事实上是不可能完全相符的），就如同在本章最后才引入了界面模块，而其实界面模块的编写是贯穿于整个代码编写的过程之中的。

不过，这样的设计过程讲述是笔者所愿尝试的行文方式。

阅读本书时，希望读者能够更多地关注文字部分以及代码的框架，而不是代码中的具体实现。其实，这些代码的实现都是非常常规的，并没有什么惊人之处。

实例是最好的教材，但愿本章能给你一个结合前面各章节理论的机会。

最后，再重复一下本章开头处所说的：

“作为实例，应该是具有示范作用的。但是，我只能说，对于每个 project，优良的设计方案可能并非只有一种，而且不见得我给出的就是最好的。注意，我所给出的，只是实现方案的所有的可能中的一种——被我所实践的那一种而已。”

# 附录

## A 浅谈 Object Pascal 的指针

大家都认为，C 语言之所以强大，且具有自由性，很大部分体现在其灵活的指针运用上。因此，说指针是 C 语言的灵魂，一点都不为过。同时，这种说法也让很多人产生误解，似乎只有 C 语言的指针才能算作指针。

其实，Pascal 语言本身也是支持指针的。从最初的 Pascal 发展至今的 Object Pascal，可以说在指针运用上，丝毫不会逊色于 C 语言的指针。

以下内容分为 8 个部分，分别是：

- ◆ 类型指针的定义
- ◆ 无类型指针的定义
- ◆ 指针的解除引用
- ◆ 取地址（指针赋值）
- ◆ 指针运算
- ◆ 动态内存分配
- ◆ 字符数组的运算
- ◆ 函数指针

### 1. 类型指针的定义

对于指向特定类型的指针，在 C 语言中是这样定义的：

```
int *ptr;
char *ptr;
```

与之等价的 Object Pascal 又是如何定义的呢？

```
var ptr : ^Integer;
ptr : ^char;
```

其实也就是符号的差别而已。

### 2. 无类型指针的定义

C 中有 void \*类型，也就是可以指向任何类型数据的指针。Object Pascal 为其定义了一个专门的类型——Pointer。于是，



```
ptr : Pointer;
```

就与 C 语言中的

```
void *ptr;
```

等价了。

### 3. 指针的解除引用

要解除指针引用（即取出指针所指区域的值），C 语言的语法是(\*ptr)，Object Pascal 则是 ptr^。

### 4. 取地址（指针赋值）

取某对象的地址并将其赋值给指针变量，C 语言的语法是：

```
ptr = &Object;
```

Object Pascal 则是：

```
ptr := @Object;
```

也只是符号的差别而已。

### 5. 指针运算

在 C 语言中，可以对指针进行移动的运算，如：

```
char a[20];  
char *ptr=a;  
ptr++;  
ptr+=2;
```

当执行 ptr++时，编译器会产生让 ptr 前进 sizeof(char)步长的代码，之后，ptr 将指向 a[1]。ptr+=2;这句使得 ptr 前进两个 sizeof(char)大小的步长。同样，来看一下 Object Pascal 中如何实现：

```
var  
  a : array [1..20] of Char;  
  ptr : PChar;    // PChar 可以看作^Char  
begin  
  ptr := @a;  
  Inc(ptr);      // 这句等价于语言 C 的 ptr++;  
  Inc(ptr, 2);   // 这句等价于语言 C 的 ptr+=2;  
end;
```

### 6. 动态内存分配

在 C 语言中，使用 malloc()库函数分配内存，free()函数释放内存。如这样的代码：

```
int *ptr, *ptr2;
int i;
ptr = (int*) malloc(sizeof(int) * 20);
ptr2 = ptr;
for (i=0; i<20; i++){
    *ptr = i; ptr++;
}
free(ptr2);
```

Object Pascal 中, 动态分配内存的函数是 `GetMem()`, 与之对应的释放函数为 `FreeMem()` (传统的 Pascal 中获取内存的函数是 `New()`和 `Dispose()`, 但 `New()`只能获得对象的单个实体的内存大小, 无法取得连续的存放多个对象的内存块)。因此, 与上面那段 C 语言的代码等价的 Object Pascal 的代码为:

```
var
    ptr, ptr2 : ^integer;
    i : integer;
begin
    GetMem(ptr, sizeof(integer) * 20);
    //这句等价于C的 ptr = (int*) malloc(sizeof(int) * 20);
    ptr2 := ptr; //保留原始指针位置
    for i := 0 to 19 do
        begin
            ptr^ := i;
            Inc(ptr);
        end;
    FreeMem(ptr2);
end;
```

对于以上这个例子 (无论是 C 版本的, 还是 Object Pascal 版本的), 都要注意一个问题, 就是分配内存的单位是字节 (Byte), 因此在使用 `GetMem` 时, 其第 2 个参数如果想当然地写成 20, 那么就会出问题了 (内存访问越界)。因为 `GetMem(ptr, 20)`实际只分配了 20 个字节的内存空间, 而一个整形的大小是 4 个字节, 那么访问第 5 个之后的所有元素都是非法的了 (对于 `malloc()`的参数同样)。

#### 7. 字符数组的运算

C 语言中是没有字符串类型的, 因此, 字符串都是用字符数组来实现, 于是也有一套 `str` 打头的库函数以进行字符数组的运算。如以下代码:

```
char str[15];
char *pstr;
```



```
strcpy(str, "teststr");
strcat(str, "_testok");
pstr = (char*) malloc(sizeof(char) * 15);
strcpy(pstr, str);
printf(pstr);
free(pstr);
```

而在 Object Pascal 中因为有了 String 类型，因此可以很方便地对字符串进行各种运算。但是，有时我们的 Pascal 代码需要与 C 语言的代码交互（例如，用 Object Pascal 的代码调用 C 语言编写的 DLL 或者用 Object Pascal 编写的 DLL 准备允许用 C 语言编写客户端的代码），就不能使用 String 类型了，而必须使用两种语言通用的字符数组。其实，Object Pascal 提供了完全类似 C 语言的一整套字符数组的运算函数。以上那段代码的 Object Pascal 版本是这样的：

```
var
  str : array [1..15] of char;
  pstr : PChar; //Pchar 也就是 ^Char
begin
  StrCopy(@str, 'teststr');
  // 在 C 语言中，数组的名称可以直接作为数组首地址指针来用
  // 但 Pascal 不是这样的，因此 str 前要加上取址运算符
  // 但如果数组是以 0 为起始下标，则与 C 语言一样，可以数组名称作为数组首地址
  StrCat(@str, '_testok');
  GetMem(pstr, sizeof(char) * 15);
  StrCopy(pstr, @str);
  Write(pstr);
  FreeMem(pstr);
end;
```

### 8. 函数指针

在动态调用 DLL 中的函数时，就会用到函数指针。假设用 C 语言编写的一段代码如下：

```
typedef int (*PVFN)(int); //定义函数指针类型
int main()
{
  HMODULE hModule = LoadLibrary("test.dll");
  PVFN pvfn = NULL;
  pvfn = (PVFN) GetProcAddress(hModule, "Function1");
  pvfn(2);
  FreeLibrary(hModule);
}
```

就笔者个人感觉来说，C 语言中定义函数指针类型的 `typedef` 代码的语法有些晦涩，而同样的代码在 Object Pascal 中却非常易懂：

```
type PVFN = Function (para : Integer) : Integer;
var
  fn : PVFN;
  // 也可以直接在此处定义，如：fn : function (para:Integer):Integer;
  hDLL : HMODULE;
begin
  hDLL := LoadLibrary('test.dll');
  fn := GetProcAddress(hDLL, 'Function1');
  fn(2);
  FreeLibrary(hDLL);
end;
```

## B RAD 与 non-RAD

似乎说到 Delphi，就会谈到这个话题。不错，Delphi 是 RAD（Rapid Application Development，快速应用开发工具）。

VB 的出现掀起了一场编程方式的革命，它带来了可视化编程，一种无数程序员所梦想的编程方式。但从此，也给人留下了这样的印象——RAD 就是“搭积木”。这也让一些“高手”对 RAD 不屑一顾。很多初学者感觉 VB 好用，立刻就可以写出一个能看得上眼的程序，但在学习、使用一段时间后，便感觉无从更进一步了。于是，又责怪 RAD 太过于简单。甚至许多熟练的 Delphi 程序员都会有着类似的担心，RAD 是不是难登大雅之堂？

或许，用 VB 进行开发，的确可以算是“搭积木”。但是，请不要认为这就是 RAD 的全部。而且，Delphi 绝对不等同于 VB！

首先，RAD 是提高生产效率的工具。工具是被人们用来解决实际问题的，而不是用来炫耀或作为理论来学习的。好的工具应该可以帮助使用者快速、有效地达成目的，那么，RAD 正是扮演了这样一个角色。它让你能够专注于问题的重点，用别人写好的、经过大量使用测试的现成控件按照项目的业务逻辑，组装成符合客户需要的软件产品，既保证了开发速度，又提高了质量（可以近似认为现成的控件是零 bug 的）。从这个意义上来说，“搭积木”的开发方式是有存在的价值的，这也是全世界程序员所梦想的“模块化”的一种形式。“搭积木”本身并没有错，RAD 并没有错。

其次，RAD 是工业的开发工具，不是学习工具。众所周知，RAD 简单易用，初学者一接触到 RAD，就仿佛找到了学习的捷径。但是，最终可能会让这些人丧气，因为简单易用的背后，是需要扎实的基础作为依托的。RAD 出现的初衷就是让开发者不必考虑太多的恼人的细节，但并非不需要相关的基础理论的支持。例如，当用 Delphi 开发基于 TCP 协议



的网络应用时，你不必知道 TCP 包的格式，但是，如果连 IP 地址都不知道是何物，纵使有 RAD 也无补于事，而且，知道得越多，在出问题时就更容易解决。RAD 的简单易用只是降低了开发人员在尝试使用 RAD 时的学习曲线。又如，假设精通 Windows 环境下的 SDK 编程，能脱口而出所需要的 Win32 API，于是，当你使用 Delphi 时，就不会有什么困难，至多熟悉一下 VCL 是如何封装这些 API 就可以了。如果 Delphi 是非常难学难懂的开发工具，你会耗费精力去用它吗？不如直接用 SDK 开发了。如果将掌握 RAD 本身作为学习目标的话，那么你注定至多只是个业余的编程爱好者而已（没有贬低爱好者的意思，只是相对于专业程序员而言）。在此，笔者有一些题外话，这就是经常可以听到有人说现在大学计算机专业学习的内容与实际需要脱节之类的话。其实如何才算不脱节呢？真的要学校教 VB、VC、Delphi 才算不脱节？如果真是这样的话，VB、VC、Delphi 都淘汰了怎么办？那才是真正的悲哀呢。言归正传，所以，RAD 不是学习的捷径，RAD 是实现的利器。

最后，RAD 和 non-RAD 是互补的。RAD 和 non-RAD 有着不同的适合领域，拥有各自的存在原因和拥护群体，它们的存在并没有冲突，也不必人为地将它们对立起来。常可见 non-RAD 的使用者嘲笑 RAD 的使用者“低阶”，其实这些所谓“高手”只是眼高而已。使用 VC 的一定比使用 VB 的人“高阶”？项目中使用什么编程语言、开发工具，时常并不是你个人所能左右的，会受很多因素制约。例如，客户的硬件环境、操作系统环境，开发环境，开发工具的成本、许可证等，所以使用什么工具说明不了问题。况且，专业程序员不应该是忠于开发工具，而是应该忠于自己的编程理念。一个人只会说“最喜爱 XX 开发工具”，“最熟悉 XX 开发工具”，绝不会说“我是使用 XX 开发工具的”。所以，RAD 并不等于低阶，non-RAD 并不意味着高人一等。RAD 和 non-RAD 的存在，并不冲突。个人水准的高低，也不是通过所使用的工具来表现的。

很多刚入门的编程爱好者，摆弄了几天 VB 或 Delphi 或 C++Builder，拖拉几个控件弄出个 EXE 之后，便宣称自己已经学会了编程，这就有些盲目了。这也是为什么 RAD 会被人误会的原因之一。不可否认，RAD 的出现，使得编程的门槛变低了。但是，入了门槛并不等于都掌握了，程序设计本身是一门博大精深的学问，做学问最重要的就是严谨、踏实。

另外，虽然 Borland 一直宣称 Delphi 是 RAD，但其实完全可以把 Delphi 当作 non-RAD 的开发工具来用（即纯 Object Pascal 编译器），这也是其提供的自由度的一个表现吧！

# 后 记

写完全书，突然有一种解脱的感觉。

从小时候起，一直有很强烈的发布欲，写了什么，做了什么，都希望让很多人看，让很多人用。

写书的机会对于有如此强烈发布欲的我来说，当然是不容错过的。况且早就梦想自己有一天能够著书立说，成为一代大师级的人物。

去年年底，不知是偶然还是必然，获得了这样一个写书的机会，一开始踌躇满志、信心百倍。虽然说不上是著书立说，毕竟自己的经验、资历还没有到这样的程度，不过对于一个大学毕业不久的年轻人来说，还是非常有诱惑力的，至少形式上是在著书立说了。

我想趁此写书的机会，向很多人推荐我所喜爱的 Delphi，与大家共享我学习 Delphi 的一些经验，同时，也是作为此前一段时间学习的小结。

然而写作半年多来，越来越感疲惫。也许是我太过稚嫩，缺乏太多的经验，致使有时还是感到力不从心；也许是我太过年轻，有着太多的想法，但是写书占据了我大部分的时间和心力，致使我想做的很多事情无法开始，从而感到身心疲惫。

写作和写程序毕竟不同。写程序可以在时间非常紧迫的压力下顺利进行，而写作却不行，毕竟写作对于我来说，其乐趣还是无法与写程序相比的，虽然我也自感文字方面水准在程序员中算是不错的了。

在写完的时候，心底几乎要呐喊：“结束了！”。如同高考最后一场考试结束走出考场的感觉，准备美美地计划一下明天可以做什么，后天可以做什么……

但愿这 7 个多月的辛苦，能换来一些什么，而不要仅仅换来对于写书的恐惧之情，那就……不太好了……