（京）新登字 035 号

工程建设规范汇编
49
城镇道路规范
本 社 编
•
中国建筑工业出版社出版、发行（北京西郊百万庄）
新 华 书 店 经 销
北京市顺义县燕华印刷厂印刷
•
开本：787×1092 毫米 1/32 印张：16³/₈ 插页：3 字数：368 千字
1993 年 3 月第一版 1993 年 3 月第一次印刷
印数：1—13,100册 定价：9.50 元
ISBN7—112—01879—X/TU·1424
(6904)

Visual C++ 开发 GIS 系统

—— 开发实例剖析

陈建春　　编著

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内容简介

本书是GIS系统开发技术和Visual C++编程技术的集合体，全面介绍和分析了MFC程序的框架结构和各种MFC辅助工具的使用方法，实例剖析了MFC中与图形和数据库应用程序设计有关的类。介绍了GIS系统的开发方法和思路，结合具体代码开发实现了矢量图形系统、数据库管理系统、矢量图形系统和数据库管理系统之间的连接、信息查询功能、空间信息统计和分析功能等，实现了一个基本GIS系统，并介绍了开发多用户GIS系统的解决方案和实现方法。

本书是根据作者多年实际开发经验和技术精炼而成，具有融合开发思路、系统构筑、编程技术、实现代码于一体的特点，相信对全面提高读者系统设计的综合能力一定会有所帮助。

本书适用于GIS、CAD、MIS等领域从事软件开发的广大技术人员以及大专院校的师生。本书的代码适用于VC++6.0和VC++5.0版本，随书附包含完整源程序代码的CD-ROM光盘。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，翻版必究。

图书在版编目(CIP)数据

Visual C++开发GIS系统：开发实例剖析/陈建春编著．－北京：电子工业出版社，2000.9
ISBN 7-5053-5955-X

Ⅰ.V… Ⅱ.陈… Ⅲ.C语言-程序设计-应用-地理信息系统 Ⅳ.P91

中国版本图书馆CIP数据核字(2000)第60373号

书　　名：Visual C++开发GIS系统——开发实例剖析
编 著 者：陈建春
责任编辑：祖振升
排版制作：电子工业出版社计算机排版室
印 刷 者：北京兴华印刷厂
出版发行：电子工业出版社　URL：http://www.phei.com.cn
　　北京市海淀区万寿路173信箱　邮编100036　68279077(门市)　68273574(批发)
经　　销：各地新华书店
开　　本：787×1092　1/16　印张：37.5　字数：980千字
版　　次：2000年9月第1版　2000年9月第1次印刷
书　　号：ISBN 7-5053-5955-X / TP·3120
印　　数：5000册　定价：72.00元　(含光盘)

凡购买电子工业出版社的图书，如有缺页、倒页、脱页、所附磁盘或光盘有问题者，请向购买书店调换；若书店售缺，请与本社发行部联系调换。电话　68279077

前　言

近几年来，地理信息系统(GIS)技术在全球得到了迅猛的发展，已经成为IT的重要组成部分，用于开发GIS系统的专业开发工具得到了飞速发展，出现了大量GIS系统的专业开发工具。应用专业开发工具开发GIS系统，具有开发起点高、可靠性好、相对简单易学等优点。但也存在着可扩展性差、适用性差、系统庞大、没有系统版权等缺点。尤其是，这些专业开发工具总体上来说还是一个难于真正修改的"黑匣子"，很难通过无缝集成实现实用系统的特殊功能。

在很多情况下，开发GIS系统的目的，不是为了精确地管理图纸、海量的坐标数据及性质数据，而只是利用GIS技术来完善MIS、CAD、DSS等系统的功能。在这种对空间坐标数据的管理要求并非很高的情况下，应用Visual C++可以迅速开发出满足特殊要求的实用GIS系统，并在灵活性、可扩展性、系统版权、技术积累等各个方面具有无可比拟的优点。

编写本书的思路

编写本书的思路，在于以开发一个基本GIS系统为主线，一步步引导读者，去掌握面向对象程序设计的精髓，全面分析MFC程序框架的组成和结构，掌握GIS系统的开发思路和开发方法。结合Visual C++开发技术、实例开发组成GIS系统的矢量图形系统、数据库管理系统、矢量图形系统和数据库管理系统的连接、空间信息统计和分析等功能模块，使读者能够从程序设计思想、系统构筑、数据组织、MFC程序设计技术、Visual C++开发工具应用等各个方面，综合掌握MFC程序设计的技术精华。通过本书的讲解，期望对读者能有所启发，从程序设计思想和编程技术等方面全面提高系统分析能力和设计能力，并在掌握本书实例开发的GIS系统基础上，能够去完善各个功能，开发出适应于各行业的实用GIS系统。

本书的特点

作者多年从事利用Visual C++开发实用GIS系统、MIS系统、DSS系统、CAD系统的实际工作，本书的大部分内容是实践经验和技术的提炼和总结，所以本书着意突出以下特点。

• 较强的连贯性和完整的源程序代码

本书从一个利用AppWizard生成的程序框架出发，并依此为基础，分析和组织GIS系统的数据结构，开发实现了包含基本功能的矢量图形系统、数据库管理系统、矢量图形系统和数据库管理系统的连接、空间信息统计和分析等功能模块，在开发过程中，结合实际代码进行了实例剖析，让读者在一个实际的开发过程中，掌握系统的设计思路和各种开发技术、开发方法。作者以实际代码结合Visual C++设计技术向读者介绍系统的具体设计过程，避免了枯燥无味的讲解。

• 全面翔实的代码注释

本书中的内容都是从实现思路、实现方法和具体实现等几个方面进行讲解。作者给出了以标准格式书写并经过全面调试的代码，并在代码中给出了完整的中文注释，使读者尤其是初

学者能够轻松地掌握书中的内容。

• 适合不同读者的需要

不论是初学者，还是高级程序设计人员，都能够从本书中学到开发思路、系统构筑、编程技术等很多方面的东西。

本书概要

本书的第 1 章 介绍了 GIS 系统的特点及其开发方法，介绍了使用 Visual C++ 开发 GIS 系统的思路。本书共分为五篇。

第一篇 介绍了 Visual C++ 程序设计要点。第 2 章 介绍和剖析了 AppWizard、ClassWizard、ClassView 、ResourceView、FileView 等各个 MFC 应用程序设计中必须或辅助使用的开发工具；第 3 章 剖析了 MFC 应用程序框架，介绍了 MFC 程序框架的组成、基本类和构成思路。

第二篇 具体开发实现了一个基本矢量图形系统。第 4 章 介绍了各种与图形系统设计有关的 MFC 类和与绘图有关的一些基本操作；第 5 章 实现了基本矢量图形系统的文档和视图；第 6 章 介绍了鼠标交互绘制图形元素的方法，实现了直线、连续直线或多边形区域、圆或圆形区域、圆弧、标注文本等的交互绘制功能；第 7 章 实现了矢量图形系统的重画、放缩、移动、回溯、显示全图、滚动等操作功能；第 8 章 实现了鼠标点选图形元素的功能，实现了删除图形元素及恢复删除操作的逆向操作功能；第 9 章 介绍了利用串行化存取文档的方法，实现了图形外部文件存取功能；第 10 章 介绍了矢量图形系统的其他一些功能，介绍了图形块及图例管理、图形参数管理、图形打印输出、与 DXF 交换文件接口等功能的实现方法或思路。

第三篇 具体开发实现了一个数据库管理系统。第 11 章 介绍了开发数据库应用程序的基础知识，介绍了关系数据库、SQL 语言中的基本子式，以及创建 ODBC 数据源的方法；第 12 章 剖析了数据库应用程序开发技术，介绍了用 AppWizard 创建基于 ODBC 数据库应用程序的步骤，详细介绍记录集合类 CRecordSet 和视图类 CDataView 所能完成的功能，剖析了各种数据库管理功能的实现步骤；第 13 章 介绍了用 DataView 管理数据源的方法和操作步骤；第 14 章 介绍了数据库浏览功能，具体实现了一个数据库浏览视图；第 15 章 具体实现了增加、修改、删除记录等数据库编辑功能；第 16 章 介绍了通用数据库过滤功能，具体实现了一个通用过滤条件编辑器；第 17 章 介绍了通用数据库排序功能，具体实现了一个过滤子式编辑器；第 18 章 介绍了通用化数据库管理功能的方法，具体实现了对数据库管理功能的通用化。

第四篇 具体开发实现了基本 GIS 系统。第 19 章 介绍了在矢量图形系统与数据库管理系统之间建立连接的方式和方法；第 20 章 具体开发实现了矢量图形系统与数据库管理系统之间的连接操作，介绍了实体的概念和实现实体的方法；第 21 章 具体实现了矢量图形系统与数据库管理系统之间的双向信息查询功能；第 22 章具体实现了各类图形元素的空间拓扑关系计算；第 23 章 介绍了开发 GIS 系统空间信息统计和分析功能的实现步骤，以及空间信息统计和分析功能在实用 GIS 系统中的具体应用。

第五篇 介绍了实现多用户 GIS 系统的方法。第 24 章 介绍了对数据库管理系统进行多用户化的具体技术；第 25 章 介绍了对 GIS 系统多用户化的方案和技术要点。

对读者的要求

阅读本书要求读者具有 C 和 C++ 编程的一些基本知识。受篇幅所限，本书对具体的 MFC

类函数不做过多的说明，读者可以结合一些 Visual C++ 的参考手册阅读。本书介绍的编程技术和开发实例是在 Visual C++ 6.0 企业版上实现的，对 Visual C++ 5.0 版本同样适用。

曾学贵教授对于本书的编排进行了悉心指导并提供了宝贵意见，张海燕、郭文军、郑顺义、万传风、郑屹、刘建峰，徐学卫，石红兵等同志参加了本书的编写工作，杜薇等同志参加了本书的录入工作，北方交通大学土建学院对本书提供了大力支持，在此一并感谢。由于作者水平有限，书中难免有错误和疏漏之处，敬请各位专家、老师和读者给予批评指正。

陈建春

2000 年 5 月于北方交大

关于作者

陈建春，31岁，北方交通大学教师、博士。长期从事GIS系统、MIS系统、CAD系统、DSS系统等的研究和开发工作。组织完成“通用多媒体GIS系统”、“新建铁路施工指挥系统”、“铁路GIS系统”、“秦沈客运专线施工管理信息系统”等系统的研究和开发工作。出版过《Visual C++高级编程技术——开发技术剖析》、《Microsoft Visual C++图形系统开发技术基础》等多部专著。翻译出版了《深入学习：JavaScript开发与实例》、《实用技术：Windows 2000 Server规划与移植》等著作。

E-mail：chenjianchun@263.net

目　录

第1章　VC++开发GIS系统的思路 …… (1)
1.1　GIS系统简介 …… (1)
1.1.1　GIS系统的应用领域和范围 …… (1)
1.1.2　GIS系统与其他系统的结合 …… (1)
1.1.3　GIS系统的专业开发工具 …… (2)
1.1.4　GIS系统专业开发工具的优缺点 …… (3)
1.1.5　VC++开发GIS系统的优缺点及应用前景 …… (4)
1.1.6　选用GIS系统的开发方法 …… (5)
1.2　VC++开发GIS系统的思路 …… (5)
1.2.1　开发相对完整的矢量图形系统 …… (5)
1.2.2　开发数据库管理系统 …… (6)
1.2.3　建立矢量图形系统与数据库管理系统的连接 …… (7)
1.2.4　开发各种查询、空间信息统计和分析功能 …… (7)
1.2.5　开发多用户GIS系统 …… (7)

第一篇　VC++程序设计技术要点

第2章　MFC应用程序设计必须或辅助使用的几个开发工具 …… (11)
2.1　使用MFC AppWizard …… (12)
2.1.1　MFC AppWizard的启动 …… (12)
2.1.2　利用MFC AppWizard产生MFC应用程序的步骤 …… (14)
2.2　使用classview …… (18)
2.2.1　启动ClassView …… (18)
2.2.2　编辑定位 …… (18)
2.2.3　增加类 …… (19)
2.2.4　增加类的成员 …… (22)
2.3　使用ResourceView …… (23)
2.3.1　ResourceView的调用 …… (24)
2.3.2　ResourceView工作分析 …… (24)
2.4　使用FileView …… (27)
2.4.1　打开一个程序项目 …… (28)
2.4.2　FileView相关的操作功能 …… (28)
2.5　使用ClassWizard …… (30)
2.5.1　Windows消息 …… (30)

2.5.2 消息处理函数 …… (31)
2.5.3 ClassWizard 的作用和启动 …… (33)
2.5.4 ClassWizard 管理消息处理函数 …… (34)
2.5.5 ClassWizard 管理重载虚函数 …… (38)
2.5.6 ClassWizard 管理映射变量 …… (40)
2.5.7 管理 Automation 和 ActiveX 事件 …… (44)
2.5.8 重新构造开发项目的 ClassWizard 工作信息 …… (44)
2.6 使用 DataView …… (45)
2.7 管理多个程序项目 …… (45)
2.8 使用各设计工具进行编程的几点建议 …… (47)
第 3 章 MFC 应用程序框架剖析 …… (49)
3.1 MFC 程序基本框架类和文件组成 …… (49)
3.1.1 基本程序框架类 …… (49)
3.1.2 应用程序主要源文件组成 …… (50)
3.2 MFC 应用程序中各个框架类的作用 …… (51)
3.3 MFC 应用程序的执行步骤分析 …… (52)
3.4 MFC 应用程序设计结构体系剖析 …… (55)
3.4.1 标准菜单的执行分析 …… (55)
3.4.2 修改标准菜单执行功能 …… (57)
3.4.3 MFC 应用程序多态性的实现 …… (58)
3.4.4 MFC 应用程序中各个框架类对象的相互调用关系 …… (59)

第 二 篇 基本矢量图形系统

第 4 章 与图形操作有关的 MFC 类及操作 …… (65)
4.1 绘图类 …… (65)
4.1.1 绘图类简介 …… (65)
4.1.2 几个绘图类的主要差别 …… (66)
4.1.3 绘图类的使用方法 …… (67)
4.2 绘图设备类 …… (68)
4.2.1 CGdiObject 类 …… (68)
4.2.2 CPen 类 …… (68)
4.2.3 CBrush 类 …… (70)
4.2.4 CFont 类 …… (73)
4.2.5 CRgn 类 …… (76)
4.2.6 CPalette 和 CBitmap 类 …… (81)
4.3 坐标映像方式 …… (82)
4.3.1 Windows 映像方式及其设置方法 …… (82)
4.3.2 逻辑坐标和设备坐标的转换 …… (83)
4.4 窗口和视口 …… (84)

4.4.1 窗口和视口的概念 …… (84)
4.4.2 窗口和视口有关的操作函数 …… (84)
4.4.3 窗口逻辑坐标与视口设备坐标的转换 …… (85)
4.5 其他绘图类操作函数 …… (88)
4.5.1 选用绘图设备对象 …… (88)
4.5.2 主要绘图属性设置函数 …… (89)
第5章 实现基本矢量图形系统的文档和视图 …… (92)
5.1 组织矢量图形系统的图形元素类 …… (92)
5.1.1 图形元素基类的组织 …… (92)
5.1.2 直线类组织 …… (93)
5.1.3 连续直线或封闭多边形类组织 …… (94)
5.1.4 圆类组织 …… (95)
5.1.5 圆弧类组织 …… (96)
5.1.6 标注文本类 …… (97)
5.1.7 图形参数类 …… (97)
5.2 组织矢量图形系统的文档 …… (99)
5.2.1 组织面向对象的文档存储管理机制 …… (99)
5.2.2 利用 MFC 模板定义管理图形元素对象指针的对象 …… (100)
5.2.3 实现矢量图形系统的文档 …… (100)
5.2.4 实现文档的管理功能 …… (101)
5.3 实现矢量图形系统的视图 …… (107)
5.3.1 建立坐标系 …… (107)
5.3.2 实现各图形元素类的绘制函数 …… (112)
5.3.3 视图类的 OnDraw 函数 …… (121)
5.3.4 实现视图 …… (122)
第6章 鼠标交互绘图 …… (124)
6.1 用鼠标绘图要解决的主要问题 …… (124)
6.1.1 捕捉鼠标操作消息 …… (124)
6.1.2 捕捉所有的鼠标输入 …… (126)
6.1.3 在屏幕上拖动图形 …… (126)
6.1.4 保存图形数据到文档 …… (126)
6.1.5 将图形以实际的形态重画 …… (126)
6.2 加入一个绘图菜单 …… (127)
6.2.1 增加菜单资源 …… (127)
6.2.2 增加菜单的消息处理函数 …… (128)
6.3 交互绘制各种图形元素 …… (130)
6.3.1 获得图形元素的唯一识别号 …… (130)
6.3.2 绘制直线 …… (132)
6.3.3 绘制连续直线和多边形区域 …… (135)

6.3.4 绘制圆和圆形区域 …… (138)
6.3.5 绘制圆弧 …… (141)
6.4 交互绘制标注文本 …… (148)
6.4.1 对话框简介 …… (148)
6.4.2 交互绘制标注文本的过程 …… (149)
6.4.3 创建对话框资源 …… (149)
6.4.4 实现文本标注对话框类 …… (150)
6.4.5 实现标注文本交互绘制功能 …… (154)
第7章 矢量图形系统的操作功能 …… (158)
7.1 增加图形操作菜单 …… (158)
7.1.1 增加菜单资源 …… (158)
7.1.2 增加消息处理函数 …… (159)
7.2 图形重画 …… (160)
7.2.1 全屏重画 …… (160)
7.2.2 局部重画 …… (161)
7.3 图形放大和移动 …… (162)
7.3.1 图形放大和移动 …… (162)
7.3.2 图形放大和移动功能的实现 …… (163)
7.4 重画上屏和重画首屏 …… (167)
7.4.1 记录屏幕状态 …… (167)
7.4.2 实现重画上屏和重画首屏功能 …… (170)
7.5 显示全图 …… (171)
7.5.1 各类图形元素的边界矩形计算 …… (171)
7.5.2 实现显示全图功能 …… (176)
7.6 提高矢量图形系统重画速度的基本方法 …… (178)
7.6.1 提高图形重画速度的方法 …… (178)
7.6.2 提高图形重画速度的实现方法 …… (179)
7.6.3 加快图形元素的绘制速度 …… (180)
7.7 实现滚动视图 …… (181)
7.7.1 滚动视图的实现方法 …… (181)
7.7.2 滚动区域参数设置 …… (181)
7.8 定制图形滚动功能 …… (183)
7.8.1 确定滚动步长 …… (183)
7.8.2 创建滚动条 …… (184)
7.8.3 实现图形滚动功能 …… (184)
第8章 图形的删除和恢复 …… (190)
8.1 鼠标点选图形元素 …… (190)
8.1.1 各种图形元素的选中区域 …… (190)
8.1.2 各种图形元素的点选判断函数 …… (195)

8.1.3 选中图形元素的特殊显示 …… (199)
8.1.4 实现点选图形元素功能 …… (199)
8.1.5 放弃选中的图形元素 …… (205)
8.1.6 图形元素的显示和选中顺序 …… (205)
8.2 图形元素的删除 …… (206)
8.2.1 删除操作的类型 …… (206)
8.2.2 从屏幕上删除图形元素的方法 …… (207)
8.2.3 各类图形元素的删除操作 …… (208)
8.2.4 实现图形元素删除操作 …… (212)
8.3 文档操作逆向化 …… (213)
8.3.1 逆向操作的含义 …… (213)
8.3.2 记录文档编辑操作过程 …… (214)
8.3.3 实现文档逆向操作 …… (220)
8.3.4 实现操作过程的完全逆向化 …… (222)
第9章 图形的存取功能 …… (223)
9.1 利用串行化存取文档 …… (223)
9.1.1 CArchive 类及 CArchive 对象的创建 …… (223)
9.1.2 Serialize 函数 …… (225)
9.1.3 CObArray 类的 Serialize 函数 …… (227)
9.2 实现矢量图形系统的存取功能 …… (228)
9.2.1 实现各个图形元素类的串行化 …… (228)
9.2.2 MFC 框架程序的文档串行化机制 …… (232)
9.2.3 实现矢量图形系统的文档存取功能 …… (233)
9.2.4 SetModifiedFlag 函数 …… (235)
第10章 矢量图形系统的其他功能 …… (238)
10.1 图形块管理 …… (238)
10.1.1 组织管理图形块 …… (238)
10.1.2 组织图例 …… (240)
10.1.3 创建图形块的方法 …… (241)
10.2 管理各种图形参数 …… (242)
10.2.1 管理颜色 …… (242)
10.2.2 管理图层 …… (243)
10.2.3 管理线型 …… (246)
10.2.4 增加辅助绘图模式 …… (246)
10.3 图形的打印输出 …… (247)
10.3.1 视图的打印机理 …… (247)
10.3.2 打印功能的实现方法 …… (248)
10.4 与 AutoCAD 的 DXF 文件的接口 …… (251)
10.4.1 与 AutoCAD 接口的条件 …… (251)

10.4.2 实现与 DXF 文件的接口功能 …… (251)

第三篇 数据库管理系统

第 11 章 开发数据库应用程序的基础知识 …… (259)
11.1 关系数据库简介 …… (259)
11.1.1 关系数据库的概念 …… (259)
11.1.2 关系数据库的基本术语 …… (260)
11.2 SQL 语言简介 …… (261)
11.2.1 选择子式 …… (262)
11.2.2 过滤子式 …… (264)
11.2.3 排序子式 …… (265)
11.3 ODBC 及创建 ODBC 数据源的方法 …… (267)
11.3.1 ODBC 简介 …… (267)
11.3.2 VC++ 中使用 ODBC 的方法 …… (268)
11.3.3 创建 ODBC 数据源 …… (268)
第 12 章 数据库应用程序开发技术 …… (275)
12.1 创建基于 ODBC 的数据库应用程序 …… (275)
12.1.1 创建一个 ODBC 数据源 …… (275)
12.1.2 MFC AppWizard 创建一个数据库应用程序的步骤 …… (276)
12.2 CRecordSet 类功能分析 …… (277)
12.2.1 连接数据源和创建记录集合 …… (278)
12.2.2 与记录集合的数据交换 …… (281)
12.2.3 打开和关闭记录集合对象 …… (282)
12.2.4 过滤记录集合 …… (283)
12.2.5 排序记录集合 …… (285)
12.2.6 移动指向记录集合的记录指针 …… (287)
12.2.7 修改记录集合 …… (290)
12.2.8 CRecordSet 类的其他操作功能 …… (292)
12.2.9 MFC ClassWizard 管理 CRecordSet 派生类 …… (295)
12.3 CRecordView 视图类分析 …… (299)
12.3.1 CRecordView 视图类 …… (299)
12.3.2 CRecordView 视图对象与记录集合对象的联系 …… (301)
12.3.3 MFC ClassWizard 管理 CRecordView 派生类 …… (302)
12.3.4 视图控制与记录集合数据交换过程剖析 …… (307)
12.4 基本数据库管理功能实现剖析 …… (310)
12.4.1 浏览功能 …… (310)
12.4.2 增加一个操作菜单 …… (316)
12.4.3 记录编辑功能 …… (318)
12.4.4 数据库过滤功能 …… (321)

12.4.5 数据库排序功能 …… (324)
12.5 DAO 数据库应用程序设计 …… (325)
第 13 章 使用 DataView 管理数据源 …… (328)
13.1 创建 Database Project …… (328)
13.1.1 创建 Database Project 的目的 …… (328)
13.1.2 用 AppWizard 创建 Database Project …… (329)
13.2 使用 DataView 管理数据源的普通功能 …… (330)
13.2.1 浏览数据库的组成和结构 …… (330)
13.2.2 Query 工具条功能介绍 …… (331)
13.2.3 组织 Select 操作 …… (334)
13.2.4 组织 SQL 的 Insert 操作 …… (337)
13.2.5 组织 SQL 的 Insert Value 操作 …… (338)
13.2.6 组织 SQL 的 Update 操作 …… (339)
13.2.7 组织 SQL 的 Delete 操作 …… (340)
13.2.8 组织 SQL 的 Make Table 操作 …… (341)
13.3 DataView 管理 SQL 数据库 …… (342)
13.3.1 针对 SQL Server 数据源的 Database Project …… (342)
13.3.2 创建数据表和修改数据表结构 …… (343)
13.3.3 新建数据视图或修改已有的数据视图结构 …… (344)
13.3.4 创建和使用存储过程 …… (345)
13.3.5 创建和使用数据库图表 …… (348)
13.3.6 Database Diagram 工具条功能 …… (349)
第 14 章 实现数据库浏览功能 …… (353)
14.1 创建一个数据库浏览视图 …… (353)
14.1.1 视图的创建过程 …… (353)
14.1.2 创建另外一类视图的步骤 …… (355)
14.1.3 在矢量图形系统中创建数据库浏览视图 …… (358)
14.2 创建一个 CRecordSet 派生类对象 …… (365)
14.2.1 创建一个 ODBC 数据源 …… (365)
14.2.2 创建管理记录集合的 CRecordSet 派生类 …… (366)
14.2.3 创建记录集合对象 …… (371)
14.3 实现数据库浏览视图 …… (371)
14.3.1 建立视图与记录集合对象的联系 …… (371)
14.3.2 获得数据库浏览视图的滚动区域 …… (372)
14.3.3 实现数据库浏览视图 …… (374)
第 15 章 实现数据库编辑功能 …… (379)
15.1 在数据库浏览视图中建立记录指针 …… (379)
15.2 建立数据库编辑功能操作菜单 …… (382)
15.2.1 修改菜单资源 …… (383)

15.2.2 建立操作菜单消息处理函数 …… (383)
15.3 建立编辑数据记录的对话框类 …… (385)
15.3.1 建立对话框资源 …… (385)
15.3.2 创建编辑数据记录的对话框类 …… (386)
15.3.3 设置对话框类的外部对象引用 …… (387)
15.3.4 建立对话框控制与记录集合类的映射变量的映射联系 …… (389)
15.4 增加和修改数据记录 …… (391)
15.4.1 获得记录的唯一识别号 …… (391)
15.4.2 增加记录 …… (393)
15.4.3 修改记录 …… (396)
15.5 删除记录 …… (398)
15.5.1 删除当前记录 …… (398)
15.5.2 删除所有记录 …… (398)
第 16 章 通用数据库过滤功能 …… (400)
16.1 过滤操作的实现方式 …… (400)
16.1.1 固定条件过滤操作 …… (400)
16.1.2 固定方式过滤操作 …… (400)
16.1.3 通用条件过滤操作 …… (401)
16.2 创建组织过滤条件的对话框类 …… (402)
16.2.1 创建对话框资源 …… (402)
16.2.2 创建组织过滤条件的对话框类 …… (404)
16.3 组织过滤条件编辑器的各种功能 …… (405)
16.3.1 过滤条件编辑器编辑的过滤条件式的语法格式 …… (405)
16.3.2 定义和初始化对话框类的变量 …… (407)
16.3.3 产生和分解各级条件子式 …… (409)
16.3.4 初始化对话框的状态 …… (414)
16.3.5 增加条件子式 …… (416)
16.3.6 删除和修改条件子式 …… (421)
16.3.7 条件子式的分解功能 …… (425)
16.3.8 实现过滤操作 …… (428)
16.4 在数据库浏览视图中实现过滤功能 …… (430)
16.4.1 增加一个操作菜单 …… (430)
16.4.2 实现数据库过滤操作功能 …… (431)
16.5 增强过滤条件编辑器的功能 …… (432)
16.5.1 增强过滤条件分解能力 …… (432)
16.5.2 增强交互式编辑界面 …… (433)
16.5.3 存储过滤条件 …… (434)
第 17 章 通用数据库排序功能 …… (435)
17.1 排序操作的方式 …… (435)

17.2 通用排序子式编辑器的实现 …… (436)
17.2.1 创建一个对话框类 …… (436)
17.2.2 定义和初始化对话框类的成员变量 …… (439)
17.2.3 产生和分解排序子式 …… (442)
17.2.4 增加、插入、修改、删除排序字段 …… (445)
17.2.5 进行排序 …… (452)
17.3 实现数据库排序功能 …… (453)
第 18 章 数据库管理功能通用化 …… (454)
18.1 各种数据库管理功能的可通用化分析 …… (454)
18.1.1 可通用化的数据库管理功能 …… (454)
18.1.2 不易通用化的功能 …… (455)
18.2 实现数据库管理功能通用化 …… (456)
18.2.1 通用化功能模块的方法 …… (456)
18.2.2 功能模块通用化的步骤 …… (458)
18.2.3 选择操作不同的记录集合对象 …… (463)
18.2.4 增加应用程序管理的记录集合对象 …… (465)
18.2.5 增加通用数据库操作功能 …… (469)
18.3 数据库管理功能完全通用化 …… (471)

第四篇 基本 GIS 系统

第 19 章 矢量图形系统与数据库管理系统的连接 …… (475)
19.1 连接方式及连接单元 …… (475)
19.1.1 连接方式 …… (475)
19.1.2 连接单元 …… (476)
19.1.3 连接的特性 …… (476)
19.2 唯一标识连接单元 …… (477)
19.2.1 标识图形连接单元 …… (477)
19.2.2 标识数据连接单元 …… (478)
19.3 管理连接信息的方法 …… (479)
19.3.1 连接的参数表示 …… (479)
19.3.2 存储组织连接信息的方法 …… (480)
第 20 章 在应用程序 Draw 中实现连接功能 …… (484)
20.1 建立管理连接信息的记录集合对象 …… (484)
20.1.1 新建记录集合类 …… (484)
20.1.2 创建管理连接信息的记录集合对象 …… (487)
20.2 组织进行连接操作的对话框类 …… (488)
20.2.1 连接操作的具体步骤 …… (488)
20.2.2 创建对话框类 …… (488)
20.2.3 初始化对话框类 …… (492)

20.2.4 实现对话框类的数据交换机制 …… (496)
20.2.5 实现对话框类的各操作功能 …… (496)
20.3 实现矢量图形系统和数据库管理系统的连接 …… (503)
20.3.1 实现连接功能 …… (503)
20.3.2 完善连接功能 …… (506)
20.4 实现实体 …… (507)
20.4.1 实体的概念 …… (507)
20.4.2 实体的数据结构 …… (507)
20.4.3 有关实体的操作 …… (509)
第 21 章 实现双向信息查询 …… (512)
21.1 建立显示和编辑数据信息的对话框类 …… (512)
21.2 实现对话框类的操作功能 …… (515)
21.2.1 对话框类的初始化 …… (515)
21.2.2 编辑性质数据 …… (518)
21.2.3 删除与数据连接单元的连接 …… (520)
21.3 实现信息查询功能 …… (521)
第 22 章 图形元素的空间拓扑关系 …… (525)
22.1 直线与多边形区域的关系 …… (525)
22.1.1 直线与直线的关系 …… (525)
22.1.2 直线与多边形区域的相交关系 …… (530)
22.1.3 连续直线与多边形区域的相交关系 …… (535)
22.2 圆弧与多边形区域的关系 …… (537)
22.2.1 圆与直线的关系 …… (538)
22.2.2 圆弧与直线的关系 …… (542)
22.2.3 圆弧与多边形区域的关系 …… (544)
22.3 图形元素与圆形区域的关系 …… (548)
22.3.1 圆与圆的关系 …… (549)
22.3.2 圆弧与圆形区域的关系 …… (551)
22.3.3 直线与圆形区域的关系 …… (554)
第 23 章 GIS 系统空间信息统计和分析功能 …… (556)
23.1 空间信息统计和分析的实现步骤 …… (556)
23.2 区域信息统计和分析 …… (557)
23.2.1 区域选择图形元素的方法 …… (557)
23.2.2 区域信息统计和分析 …… (559)
23.3 线类相交信息统计和分析 …… (560)
23.3.1 线类相交选择图形元素的方法 …… (560)
23.3.2 线类相交区域计算 …… (561)
23.3.3 线类相交信息统计和分析 …… (561)
23.4 实用 GIS 系统开发 …… (562)

第五篇　多用户 GIS 系统

第 24 章　开发分布式多用户数据库管理系统 …… (567)
24.1　Visual C++开发分布式多用户数据库管理系统 …… (567)
24.1.1　服务器/客户机应用程序设计 …… (567)
24.1.2　Visual C++开发基于服务器/客户机体系的数据库管理系统 …… (567)
24.2　分布式多用户数据库管理系统开发技术 …… (568)
24.2.1　定期更新记录集合对象 …… (568)
24.2.2　多用户应用程序中对记录的修改策略 …… (570)
第 25 章　实现多用户 GIS 系统 …… (574)
25.1　部分多用户化 GIS 系统 …… (574)
25.1.1　部分多用户化 GIS 系统 …… (574)
25.1.2　实现数据库管理系统的多用户化 …… (574)
25.1.3　实现连接信息管理多用户化 …… (575)
25.2　完全多用户化 GIS 系统 …… (575)
25.2.1　矢量图形元素的数据库记录化 …… (575)
25.3.2　转换数据表的内容为图形元素 …… (577)

附录　本书所附光盘内容及技术支持 …… (578)
参考书目 …… (579)

第 1 章　VC++ 开发 GIS 系统的思路

GIS 系统是地理信息系统(Geographical Information System,简称为 GIS 系统)的简称,是一种将空间位置信息与属性数据结合在一起的系统。

近几年来,GIS 技术在全球得到了迅速的发展,GIS 技术已经成为 IT 的重要组成部分。在国外,用于开发 GIS 系统的开发工具得到了飞速发展,发展速度之快,出乎意料之外。

1.1　GIS 系统简介

1.1.1　GIS 系统的应用领域和范围

从最初概念来看,GIS 系统指的是将地球科学与属性数据相结合完成的一种大型的集成系统。但是,在实际应用中,以 GIS 系统技术为基础建立起来的系统已经应用到各种领域。各种 GIS 系统的复杂程度相差很大,大到针对某个地区(以至全球)的数字地面模型资料所建立起来的庞大系统,如美国在海湾战争、科索沃战争中采用的辅助指挥作战的 GIS 系统;小到只是结合了简单平面位置坐标的小型管理系统,如一个住宅小区的平面管理系统,一个商场的触摸式导购系统等。从 GIS 系统中空间坐标的表示方法来看,有利用数字地面模型资料建立起来的三维图形为基础的 GIS 系统,也有使用二维平面示意图形的 GIS 系统。

GIS 系统将普通管理信息系统(MIS)与矢量图形系统结合在一起,克服了 MIS 系统的局限性,大大提高了系统的直观性和可操作性,同时提供了 MIS 系统所不能解决的功能,如空间信息统计、计算等。

1.1.2　GIS 系统与其他系统的结合

单纯的 GIS 系统实用价值受到限制。在实际应用中,GIS 系统往往与其他系统结合在一起,GIS 系统通常与遥感系统(RS)、全球定位系统(GPS)结合在一起,将三者一体化,也就是通常所说的"3S"系统。现在的 GIS 系统的应用范畴非常广泛,它已经与 CAD、多媒体、通信、办公自动化(OA)、虚拟现实等技术相结合,形成了综合的信息技术。

GIS 系统作为一个基础,与其他系统相结合,来设计实现各种实用系统。作者开发的"新建铁路施工指挥系统",利用 GIS 系统技术将施工地区的施工指挥平面图矢量化,用来直观地管理与工程建设有关的各种信息,如各种材料点、交通网、城市、地质情况、线路上的各种重点工程等;而系统的主要功能则是实现铁路建设中的管理功能,如施工组织计划的制定、施工组织进度图的辅助制定和绘制、重点工程工期的管理、施工质量、施工进度、工程投资的管理等等。整个系统应该属于 MIS 系统的范畴,但结合了 GIS 系统技术后,使 MIS 系统的操作功能有了很大的改进,在实用性、直观性等方面达到了新的高度。再比如,作者开发的"铁路建设前期决策系统",通过实现铁路 GIS 系统,在以铁路网为主的全国平面图中记录了路网信息、路网中的各种性质信息(如通过能力、地形条件等)、各经济据点的信息、粗略的数模信息等;而系统最

终的设计功能，并不是为了单纯完成对信息的管理、查询，以及对路网平面图的管理等功能，而是以此为基础，通过对路网通路的评价（如最短路径的选择，路网性能的分析和评价，路网中流量的分析等），得到路网的薄弱环节，为新建铁路、改造铁路提供决策依据；并通过在路网平面图中选择新建铁路方案和既有铁路方案，通过地形、经济点的运量、工程造价等信息，计算得到各种方案的技术标准、工程造价，并提供经济评价，为铁路建设决策部门提供决策依据。所以这个系统，可以认为是 GIS 系统与 MIS、DSS 以及 CAD 等系统的结合，而 GIS 系统是实现这些系统的基础。

所以，在很多情况下，设计 GIS 系统的目的并不单纯是为了信息管理，而是以此为基础和平台，开发完成信息管理系统（MIS）、决策系统（DSS）、辅助设计系统（CAD）、办公自动化系统（OA）等综合实用系统。

1.1.3 GIS 系统的专业开发工具

在 GIS 系统的发展过程中，目前已出现了大量的 GIS 系统的专业开发工具。从这些专业开发工具的组成结构上，可以归纳为以下的几个主要类别。

(1) 集成式 GIS

集成式 GIS 系统，指的是集合各种功能模块的大型 GIS 系统软件包。ESRI 公司推出的 Arc/Info，Genasys 公司的 GenaMap 等都是集成式 GIS 开发工具。集成式 GIS 系统的优点是各项功能已形成独立的完整系统，缺点是系统复杂、庞大，成本较高，并且难于与其他应用系统集成。

(2) 模块化 GIS

模块化 GIS 系统是把 GIS 系统按功能划分成一系列模块，运行于统一的基础环境中。Intergraph 公司的 MGE 就是一个有代表性的模块化 GIS 系统。模块化 GIS 系统具有较强的工程针对性，便于开发和应用。

(3) 组件式 GIS

组件式 GIS 是随着近年计算机软件技术的发展而产生的，代表了 GIS 系统的发展潮流。组件式 GIS 具有标准的组件式平台，各个组件不但可以进行自由、灵活的重组，而且具有可视化的界面和使用方便的标准接口。最主要的组件式 GIS 平台就是 Microsoft 的 COM（Component Object Model 的简称，即组件对象模型）。基于 COM，Microsoft 推出了 ActiveX 控件技术。ActiveX 控件已经成为当今可视化程序设计的标准控件，新一代的组件式 GIS 大都是采用 ActiveX 控件来实现的。Intergraph 公司推出的 GeoMedia，ESRI 公司推出的 MapObjects，MapInfo 公司推出的 MapX，中科院地理信息产业发展中心开发的 ActiveMap 等，都是采用 ActiveX 来实现的。这类 GIS 系统提供的是为完成 GIS 系统而推出的各种标准 ActiveX 控件，使 GIS 系统开发者不必掌握专门的 GIS 系统开发语言，只需熟悉基于 Windows 平台的通用集成开发环境，知道组件式 GIS 各个控件的属性、方法和事件，就可以利用各种可视化开发语言（如 Visual C++、Visual Basic、Visual Foxpro、Borland C++、Delphi、C++ Builder、Power Builder 等等）和利用这些控件组织实

现 GIS 系统。所以，组件式 GIS 在系统的无缝集成和灵活性方面具有优势，从一定意义上讲，它代表了 GIS 系统的发展方向。

(4) 万维网 GIS

万维网 GIS 即 WebGIS。是 Internet 技术与 GIS 系统相结合的产物。已经有一些公司推出了万维网 GIS 产品，如 MapInfo 公司的 MapInfo ProServer、MapXtreme，Intergraph 公司的 GeoMedia Web Map，ESRI 公司的 Internet Map Server for ArcVire & MapObject 和 Autodesk 公司的 MapGuide 等等。已经推出的万维网 GIS 是利用现有的 GIS 软件通过 CGI 或者 Server API 构造的过渡产品。随着组件式 GIS 的发展和分布式对象 Web(Distributed Object Web)技术的逐渐成熟，未来的万维网 GIS 将是基于 COM/ActiveX 或 COBRA/Java 开发的分布式对象 GIS 系统。

1.1.4　GIS 系统专业开发工具的优缺点

上一小节介绍了一些主要的 GIS 系统专业开发工具，应用这些工具进行实用 GIS 系统的开发，具有较大的优点，也有明显的缺点。

(1) 优点

应用 GIS 系统专业开发工具进行 GIS 系统的开发时，其主要的优点表现在以下几个方面：

① 较高的开发平台。各种 GIS 系统专业开发工具一般提供了强大的数据输入输出功能、空间分析功能、良好的图形平台、巨大的存储容量、良好的可靠性能，可以在较高的起点上直接进行 GIS 系统的组织开发工作，开发的系统性能和可靠性好。

② 可靠性好。专业的 GIS 系统开发工具作为成熟的产品，它在容量、可靠性等各个方面经过了严格的测试和许多用户的使用。

③ 简单易学。GIS 系统专业开发工具是一个应用软件，即使提供的二次开发功能，也是建立在已经实现的各种功能基础上的，所以对于用户来说，不需要特别高的程序设计思想和数据控制能力，掌握利用这些工具开发 GIS 系统的技术相对比较容易。

(2) 缺点

利用 GIS 系统专业开发工具进行 GIS 系统开发时，也存在着如下一些不可克服的缺点：

① 可扩展性差。一些传统的 GIS 系统专业开发工具虽然也提供了一些接口语言，但总体上来说还是一个难于真正修改的“黑匣子”。应用这些工具开发实用系统时，往往遇到这种情况：我们要开发的系统，往往只需使用专业开发工具提供的很少一部分功能(如 20%)，而采用专业开发工具可能只能实现系统的 90%功能，而剩余的 10%功能很难用这个开发工具开发完成，甚至根本无法实现。造成这种情况的原因是：开发工具从总体上讲，是一个“黑匣子”，用户很难甚至不可能知道和控制(或完全控制)“黑匣子”内的功能；而在实际应用中，要开发的往往并不全是传统意义上的 GIS 系统，而是将 GIS 技术与 MIS、DSS 和 CAD 等系统集成的系统，在这种情况下，应用这些开发工具，很难实现与这些实用系统的无缝集成。总之，对于 GIS 系统专业开发工具来说，它已经实现并提供的功能，GIS 系统开发者可以非常容易地使用，而专业开发工具没有提供的功能，则很难甚至根本无法用专业开发工具去实现。近年来，提出的组件式

GIS 系统开发工具就是为克服这些困难而做的改进，但这种 GIS 系统也很难达到真正的无缝集成。

② 较为庞大，软硬件要求高。为了能够支撑运行这些 GIS 系统专业开发工具，往往对系统的软、硬件有较高的要求。这在开发一些小型的 GIS 系统，特别是开发传统意义以外的应用 GIS 技术的专用系统中，需付出额外的代价。

③ 没有系统的版权。这一点，在很大程度上，是制约使用这些 GIS 系统专业开发工具的重要因素。利用这些 GIS 系统专业工具开发的 GIS 系统，实际上只是在人家系统的基础上做一些简单的应用开发，开发完成的产品同样需要这些专业开发工具的支撑平台，也就是用户除了支付开发费用外，还需要再购买这些支撑平台。对于 GIS 系统开发者来说，这就等于没有自己的产品，没有自主的系统版权，还要受到软件升级等各种因素的制约。

1.1.5 VC++ 开发 GIS 系统的优缺点及应用前景

本小节将介绍应用 VC++ 从底层开发 GIS 系统的应用前景和优缺点。实际上，目前比较流行的一些 GIS 系统专业开发工具很多都是用 C++ 开发完成的。在短时间内，我们用有限的开发力量和资金投入，很难开发出像 GIS 系统专业开发工具那样的软件系统，但是应用 VC++ 开发以信息管理、决策服务及设计为主的实际 GIS 系统时，具有良好的应用前景。在 1.1.2 节，我们讨论过，在很多情况下，建立 GIS 系统的目的，不是为了能够精确的管理图纸、管理海量的坐标数据及性质数据，而只是利用 GIS 技术来完善 MIS、CAD、DSS 等系统的功能，在这种对空间坐标数据的管理要求并非很高的情况下，应用 VC++ 可以迅速开发出满足要求的实用系统，并在灵活性、可操作性各方面具有无可比拟的优点。

(1) 优点

① 较强的灵活性。灵活性是利用 VC++ 开发 GIS 系统的最大优点。应用 VC++ 开发 GIS 系统时，因为系统的所有流程和数据都可以在设计者的控制之下，可以根据系统的具体要求实现具体的操作功能，在一些 GIS(特别是在一些小型或并非以传统的 GIS 功能为主的)系统开发时，具有无可比拟的优势。它可以根据系统的需要来实现功能，设计的系统短小精悍，软硬件要求低，运行速度快。

② 易于扩展成各种系统。用 VC++ 开发的 GIS 系统，与使用 GIS 系统专业开发工具不同：用 GIS 系统专业开发工具开发 GIS 系统时，开发者所做的是在别人系统基础上的简单开发和应用，完全受该专业开发工具的制约，开发者形成不了自主的技术积累和创新；而用 VC++ 开发 GIS 系统时，开发者可以在开发过程中，不断完善和综合开发技术，从各个方面进行完善，把系统的开发从应用项目级提高到开发工具级，最终能够完成自身的 GIS 系统开发工具和底层开发技术，并以此为基础，使其在信息管理系统(MIS)、决策系统(DSS)、控制系统(C^3I)、辅助设计系统(CAD)等各种实用系统开发中得到迅速地推广应用。

③ 有系统版权。开发者自身具有系统版权，在一些行业的大规模推广中具有无可比拟的优势。

(2) 缺点

① 开发有一定困难且开发量大。用 VC++ 开发 GIS 系统时，因为各种技术需要从底层进行开发，进行开发的出发点低，需要较高的开发技术和很大的开发量，要开发完成一个满足实用要求的二维矢量图形系统，所需要的开发量已经非常可观，如果再考虑实现三维的、处理数模等操作功能，其开发量更大。开发出的矢量图形平台要经过反复修改调试，在短时间内可能无法与成熟的 GIS 系统开发工具提供的平台媲美。

② 开发连续性难以得到保障。目前在国内进行的软件开发项目，往往受商业利益的驱动，只是短期行为。通常的作法是采用一些最新的开发工具，拼装成用户需要的系统。这样开发出来的系统，常常是能用，但不好用；能解决一些问题，但解决不了全部问题；当前凑合着用，但难以改进和升级。这样做的结果是在软件设计中很难有大的突破。一些研究机构，则由于受管理体制等诸多因素的限制，也很难开发出具有竞争力的 GIS 产品，往往停留在研究阶段。

③ 对开发人员的素质要求高。用 VC++ 开发 GIS 系统，不同于使用成熟的 GIS 系统开发工具，需要有较强的设计思想、强大的数据和流程控制能力和良好的协作精神。由于受国外软件产品的冲击和软件设计学习导向的影响，目前软件设计人员和高校毕业生急需提高用 VC++ 开发 GIS 系统的综合技术能力。

1.1.6　选用 GIS 系统的开发方法

以上各小节比较了用 GIS 系统专业开发工具和用 VC++ 设计 GIS 系统的优缺点。在进行 GIS 系统设计时，具体选用哪种方法，则需要考虑系统本身的性质和特点，以及设计者自身的具体情况来确定。一般来说，对于一些大型的 GIS 系统开发项目，如一个地区的综合信息管理系统等，因为其图形平台、容量、可靠性等各方面的要求，一般易于选择已经成熟的 GIS 系统开发工具来组织开发。而对于一些小型的 GIS 系统，特别是一些以数据管理、决策研究、辅助设计等具体应用为主的系统，或者在某个行业中需要推广使用的实用系统，因为这些系统对矢量图形平台要求不是太高，开发的重点在于 GIS 技术的实际功能，可考虑采用 VC++ 来组织系统的开发，并把它作为开发具有自主版权的大型 GIS 系统的“练兵”。在开发的过程中，不断地探索和完善开发技术，最终达到和超越 GIS 系统专业开发工具具有的功能。进而，开发出具有独立自主版权的软件产品。作者认为这是我国发展软件产业的必由之路。

1.2　VC++ 开发 GIS 系统的思路

用 VC++ 开发 GIS 系统，其实现的难度是较大的。一个最基本的 GIS 系统，需要包含如下的组成部分：管理空间坐标数据的矢量图形系统，管理性质数据的数据库管理系统，以及实现矢量图形系统与数据库管理系统双向连接的连接系统。一些 GIS 系统专业开发工具，如 MapInfo 等，也是用 C++ 通过如上的思路，在矢量图形系统基础上开发完成的。

1.2.1　开发相对完整的矢量图形系统

矢量图形系统是 GIS 系统最重要的组成部分，也是用 VC++ 开发 GIS 系统的重点所在。

应用在不同领域的 GIS 系统，其需要的矢量图形系统的功能是有差别的。建立一个城市信息管理 GIS 系统时，要求矢量图形系统有丰富的图形元素、丰富的线型、巨大的存储容量、完善的输入和输出功能、强大的地图编辑功能，甚至需要三维显示和处理能力；而某些管理型 GIS 系统所需要的矢量图形系统，可能只需要具有简单地处理二维普通图形元素的功能。

一个基本的矢量图形系统一般需要如下的功能：

① 相对完善的图形元素。应具备处理点、直线、圆、连续直线、多边形区域、标注文本等图形元素的能力，并具有处理图例(即图形块)的能力。具有了基本图形元素和图例后的矢量图形系统，在很大程度上能够满足普通的管理型 GIS 系统的矢量图形要求。提供矢量显示图像的功能在很大程度上能够改善矢量图形界面的质量。

② 相对完善的图形操作功能。应具备图形的放缩、移动、回溯等各种操作功能。

③ 相对完善的图形输入和输出功能。应具备鼠标交互绘制功能，图形数据交互输入等功能。根据具体情况需要具有数字化仪输入、与其他图形系统的数据接口等功能。系统应具备从打印或绘图设备输出图形的能力。

④ 具有图层、颜色、线型等的设置功能。

⑤ 具有较大的存储容量。处理图形元素的容量应该能够达到一般实用要求。

⑥ 具有较强的容错能力和可恢复性。

⑦ 具有较高的处理速度。

本书的第二篇，将结合具体的实现代码，介绍一个包含了以上基本功能的矢量图形系统的开发过程。读者可以在这个矢量图形系统的基础上，根据需要不断地对这个矢量图形系统的功能进行扩充和完善。

1.2.2 开发数据库管理系统

数据库管理系统是 GIS 系统重要的组成部分，用来管理 GIS 系统中的各种性质数据。我们可以采用不同的方法来开发这个数据库管理系统。作者在开发“多媒体 GIS 系统”时，采用 VC++ 开发了一套自身的数据库管理系统，具有极大的使用灵活性，并可以与矢量图形系统完美的结合。虽然这种方法开发的数据库管理系统灵活方便、集成性好，但需要有很强的程序设计能力和大量的开发工作，而且在开发多用户 GIS 系统时比较困难。所以，除非开发系统级的 GIS 系统，一般不推荐使用这种方法。最常用的开发数据库管理系统的方法是使用 ODBC。ODBC 做为一种开放的数据库标准，可以使用各种数据库系统(如 FoxPro、SQL Server、Oracle、Acecess 等)的数据库文件作为数据源，使用 SQL 语言作为操作和查询语言，按照统一的方法来实现数据库管理系统。VC++，特别是最新的 VC++6.0 企业版，为开发基于 ODBC 的数据库管理系统提供了完善的开发技术。

一个基本的数据库管理系统需要提供以下的操作功能：

① 数据浏览功能。实现操作界面对数据表的浏览。

② 编辑功能。能够对数据表进行增加记录、修改记录、删除记录等操作，即可以任意修改数据表的内容。

③查询功能。能够提供操作界面，任意组织数据表的过滤条件，对数据表进行查询操作。

④排序功能。能够组织排序字段，对数据表进行多字段和可选顺序的排序操作。

本书的第三篇，将以具体的实现代码，介绍开发基于 ODBC 的数据库管理系统的过程。

1.2.3 建立矢量图形系统与数据库管理系统的连接

在GIS系统中,矢量图形系统与数据库管理系统并非相互独立的,而是需要在两者之间建立起连接关系。所谓建立连接关系,就是在矢量图形系统的图形元素与数据库管理系统的数据库记录(或数据库视图)之间建立连接,把性质数据赋给矢量图形元素。对于一个比较完善的GIS系统来说,这种连接必须具有以下的特点:

① 连接的双向性。即通过矢量图形元素可以得到并维护性质数据库中的相应性质数据,同时,通过性质数据库也能够得到并维护矢量图形系统中的相应图形元素。

② 连接的多项性。可以这样来理解联系的多项性:对于矢量图形元素来说,连接的多项性指的是一个矢量图形元素可以与多个性质数据建立连接,如针对矢量图形上的一个城市标志,建立了两个数据库来存储城市的信息,一个数据库是用来存储各种资源方面的(如各种资源的数量等),另外一个数据库是用来存储各种工农业生产指标方面的,同样,对于性质数据来说(如对于数据库中的一条记录),连接的多项性指的是它可以与多个矢量图形元素建立起连接。

③ 连接的稳定性。矢量图形元素与性质数据间的连接一旦建立,就会永远的存在下去(除非解除了连接),不会因为对矢量图形元素或性质数据库的增加、插入等操作而受影响。连接的稳定性是通过对矢量图形元素和性质数据建立唯一识别号来实现的。在GIS系统中,连接单元(矢量图形系统中的一个图形元素和性质数据库中的一条记录都是最基本的连接单元)必须具有唯一的识别号,即每一个连接单元必须具有与其他连接单元区分开来的唯一标志。在第二篇实现的矢量图形系统和第三篇实现的数据库管理系统,都为各种连接单元建立了唯一的识别号。

本书的第四篇,将以具体的实现代码,介绍在矢量图形系统与数据库管理系统之间建立连接关系的实现方法和实现过程。

1.2.4 开发各种查询、空间信息统计和分析功能

在建立了矢量图形系统和数据库管理系统的连接关系后,就可以通过对矢量图形系统的图形元素的操作(选中、图形元素间的拓扑关系等)来得到或操作与之连接的性质数据,实现空间信息统计和分析功能。即以建立起的GIS系统框架为基础,开发实现实用系统的具体功能。

本书的第四篇,将以具体的实现代码,介绍通过图形元素查询数据信息的方法,介绍图形元素间的相互关系计算函数,以及各种空间信息统计、分析、处理等功能。

1.2.5 开发多用户GIS系统

随着计算机网络的普及和发展,不具备网络功能的计算机软件系统的应用将受到很大制约,需要用VC++开发多用户GIS系统。开发实现多用户GIS系统,有不同的实现方案,即网络化的程度是不一样的。以下是两种不同程度的网络化方案:

① 实现性质数据库的多用户化。这种方案是只对GIS系统中的数据库管理系统多用户化。在某些GIS系统中,矢量图形一旦建立,可能不再改动或较少改动,即使做一些改动,也可以在单用户的运行状态下进行。在这种情况下,矢量图形系统可以以图形文件来组织存放,而

性质数据库则使用运行在网络环境下的数据库系统来实现，如可以使用SQL Server在服务器中存储性质数据，将GIS系统中的数据库管理系统多用户化。利用这种方式建立起来的GIS系统，其性质数据采用的是服务器中的共享数据库，各个客户机可以通过矢量图形界面来查询、统计、管理各种性质数据，并完成各种管理、决策功能。例如，在以GIS技术建立施工指挥管理系统时，施工组织平面图一旦建立，可能较少改动，就可以采用这种把性质数据放在网络共享数据库中的解决方法，在多用户状态下，各个客户机可以完成各种性质数据的输入、修改等操作，而不能在多用户状态下，对矢量图形进行修改。

② 实现GIS系统完全多用户化。①中实现的多用户网络化方案只是对性质数据管理系统进行了多用户化，而没有对矢量图形系统和连接系统多用户化。要实现一个真正意义上的多用户GIS系统，必须把矢量图形系统和连接系统也放置到共享数据库中，不能再用普通的图形文件来组织存储矢量图形，同①中管理性质数据类似，需要将矢量图形元素和连接信息存储在共享数据库中，通过建立不同结构的数据表来存储不同类别的矢量图形元素或连接信息，实现用数据库管理系统的方法来管理各种矢量图形数据和连接信息，以此来实现多用户GIS系统。

本书的第五篇中，将介绍实现多用户GIS系统的解决方案和实现方法。对用VC++开发实现服务器/客户机模式的GIS系统提供具体指导。

VC++ 程序设计技术要点

本篇将介绍进行 MFC 应用程序设计的方法，剖析 MFC 应用程序框架的组成结构，并介绍辅助程序设计的各个工具的工作基础和使用方法。

第 2 章 MFC 应用程序设计必须或辅助使用的几个开发工具

第 3 章 MFC 应用程序框架剖析

第一篇

第 2 章　MFC 应用程序设计必须或辅助使用的几个开发工具

VC++(本书以下各章凡提到 VC++,均是指 5.0 版本和 6.0 版本)是一个功能强大的可视化编程工具,是目前功能最为强大的程序开发平台之一,而利用 MFC 程序框架进行应用程序设计,是 VC++最有效和强大的部分。当然,一个开发工具的功能越强大,那么其本身组成就越庞大,而要完全掌握这个工具所需要的难度就越大,企望功能强大且省事又短小的程序设计在目前阶段还是不现实的。VC++是一个非常复杂的程序开发包,包含了文本编辑器、资源编辑器、工程创建和管理工具、调试器、在线帮助等各个工具,在这个环境中,程序员可以完成应用程序的创建、编码、测试、完善等各个阶段的工作。图 2-1 是 VC++ 6.0 的工作界面。

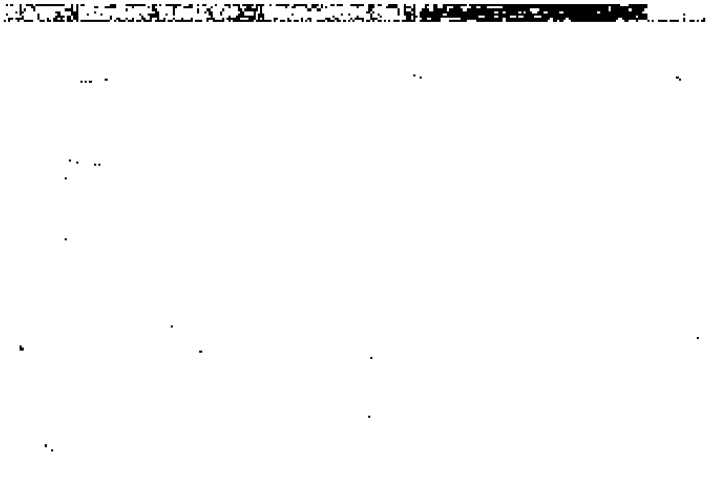


图 2-1　VC++ 6.0 的工作界面

本章对 VC++开发环境中的一些辅助开发工具进行简要的介绍。对于这些辅助开发工具,在关于 VC++的书籍中大都有介绍,但多数是使用这些工具的操作步骤,而没有具体分析这些辅助工具的工作基础。这样一来,造成许多程序设计者对工具的神秘感和依赖性,离开这些辅助设计工具就不能进行程序设计,更不能按照自己的意愿灵活的组织程序,而陷于固定的程序框架中而不能自拔。本章,将从这些工具的工作基础及实现过程等方面分析这些工具的作用,达到真正理解 MFC 程序框架的目的。作者结合自己进行 VC++程序设计的一些经验和体会,对进行程序设计时用的比较多的几个工具进行完整的剖析。在介绍这些工具之前先要告诉读者的是:大部分工具都是在进行 VC++程序设计时起辅助作用的,而非必须的,从一定意义上讲,这些工具更适合于具有一定开发经验的程序员。对初学者来说,如果不去真正剖析程序的结构而只停留在使用这些工具的层面上,从长远发展来看是不利的,具体搞开发工作也

会事倍功半。

本章主要介绍 AppWizard、ClassView、ClassWizard、ResourceView、FileView、DataView 几个工具及其相关的操作。

AppWizard 的作用是用来建立程序框架。因为 VC++ 的功能非常强大,可以用来开发多种不同的应用程序,如:可以开发基于 MFC 的 Window 应用程序(.exe),基于 Windows 的动态链接库程序(.dll),ActiveX 控件程序(.ocx)等等。对于每一类应用程序来说,并不是从零开始进行开发,微软的系统设计者们为开发多种应用程序设计好了程序的框架部分,通过 AppWizard 就可以自动生成包含了基本程序框架的应用程序包,开发者可以在此基础上进行程序的设计。

ClassView 是 VC++ 工作平台上的一个工具,它通过对应用程序的头文件(.h)和实现文件(.cpp)进行处理,从中得到并管理应用程序中所包含的类。可以实现增加类、增加类的函数和变量,以及在头文件(.h)和实现文件(.cpp)中定位各个类、类的成员函数及变量等各种操作。

ClassWizard 是 VC++ 中功能十分强大的一个辅助编程工具,它通过对头文件(.h)和实现文件(.cpp)进行处理,能够对类中的消息处理函数、可重载的虚函数、映射数据、Automation 属性、ActiveX 事件等进行管理和辅助编程。

ResourceView 是 VC++ 中对程序的资源进行可视化管理的工具。利用它调用 AppStudio 可以对程序所拥有的各种资源(菜单、工具条、对话框、加速键、热键等)进行可视化管理。

FileView 是管理一个开发项目所包含的所有文件的工具。通过它可以用来管理一个开发项目所包含的源文件。

DataView 是一个在 VC++ 平台中管理和使用 ODBC 数据源的一个工具。它用来管理进行数据库程序设计时连接的数据源。在 VC++ 中利用 ODBC 或 DAO 等进行数据库程序设计时,需要使用别的数据库系统提供的数据源,如可以使用 Foxpro、Access、Oracle、SQL Server 等的数据库文件,通过 DataView 可以在 VC++ 开发环境下对各种数据库提供的数据源进行浏览和管理,这对于数据库应用程序设计具有十分重要的作用。

在以下的各节中,将对这些主要工具进行分析,并介绍与其有关的主要操作。

2.1 使用 MFC AppWizard

VC++ 能够用来创建多种类型的 MFC 应用程序,不同的应用程序具有不同的应用程序框架,VC++ 系统的设计者们创建了程序框架生成工具 AppWizard,用它来建立开发各类 MFC 应用程序所需要的程序框架。这样,不论是资深程序员还是初学者,都可以用这个工具生成包含了一些基本功能的框架程序,在此基础上再进行应用程序的开发工作。

2.1.1 MFC AppWizard 的启动

在 VC++ 工作平台下,运行"File"菜单下的菜单项"New...",就会进入一个 New 对话框,在对话框顶部有 4 个供选择的标签"Files"、"Projects"、"Workspaces"、"Other Documents"(见图 2-2)。"Files"标签下的选项用来选择创建各种类型的文件(如.h,.cpp 等);"Workspaces"标签下的选项用来创建一个工作平台;"Other Documents"标签下的选项用来创建 Excel、Word 等文档;"Projects"标签下的选项用来创建各种应用程序框架。用鼠标点中"Projects"标签,就会进入如图 2-2 所示的对话框界面。在 New 对话框中可以选择创建的应用程序框架见表 2-1。

图 2-2　New 对话框

表 2-1　AppWizard 创建的应用程序类型

选 项 名 称	创建的应用程序框架
ATL COM AppWizard	创建 ATL(Active Template Library)应用模块程序
Cluster Resource Type Wizard	
Custom AppWizard	创建非标准的应用程序
Database Project	创建数据库管理项目
DevStudio Add-in Wizard	创建 ActiveX 组件或 VBScript macros
Externed Stored Proc Wizard	
ISAPI Extension Wizard	创建 Internet Server 扩展
Makefile	创建独立于 VC++ 开发环境的应用程序
MFC ActiveX ControlWizard	创建 ActiveX 控件应用程序
MFC AppWizard(dll)	创建 MFC 的动态链接库程序
MFC AppWizard(exe)	创建一般的 MFC 应用程序
New Database Wizard	支持 SQL 等的数据库应用程序
Win32 Application	Win32 的应用程序(非 MFC)
Win32 Console Application	Win32 的控制台应用程序(非 MFC)
Win32 Dynamic-Link Library	Win32 动态链接库
Win32 Static-Link Library	Win32 静态链接库

这些应用程序框架中，最常用的是开发基于 MFC 的一般应用程序(MFC AppWizard(exe))。本书讲述的主要内容归结起来讲就是：以利用 AppWizard 生成的 MFC 应用程序框架为基础，引导读者具体开发一个实际的图形系统和数据库管理系统，以及矢量图形系统与数据库管理系统的连接。同时，在介绍系统开发的过程中，使读者一步步地掌握 GIS 编程技术和 MFC 程序

设计方法。这是作者写作此书的出发点和归宿。

2.1.2 利用 MFC AppWizard 产生 MFC 应用程序的步骤

在这一节中,应用 MFC AppWizard 生成一个叫做 Draw 的应用程序。以下是利用 AppWizard 产生 MFC 应用程序框架的步骤:

首先,进入 VC++的工作平台,运行工作平台上的"File"菜单下的"New"命令,打开 New 对话框,然后进行如下步骤的操作:

① 选择 New 对话框中的"Project"标签,进入到图 2-2 所示的对话框界面。因为目前要生成的是一般 MFC 程序,所以从列表框中选择 MFC AppWizard(exe)选项,在"Project Name"(项目名称)编辑框中,输入要生成的应用项目的名称 Draw,在"Location"编辑框中列出的是这个项目所在的目录名称。对所在目录可以进行修改而设置其他目录路径,如果用户输入的路径不存在,AppWizard 会自动进行创建。如图 2-2 所示,在"Project name"和"Location"编辑框中输入 Draw 和 C:\Draw。

② 在 PlatForms(平台)区中只选择"Win32"复选框,清除其他的选项。

③ 在完成了以上的输入或选择操作之后,单击"OK"按钮,MFC AppWizard 会生成该项目目录,并进入到如图 2-3 所示的 MFC AppWizard - Step 1 of 6 对话框。

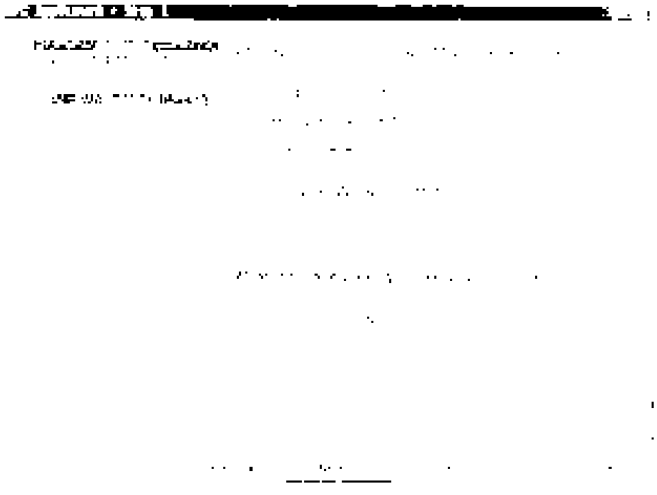


图 2-3 MFC AppWizard-Step 1 of 6 对话框

在此对话框中,要求用户对生成程序的文档类型进行选择,共有 3 个选项:Single documents(单文档界面)、Multiple documents(多文档界面)和 Dialog-based(基于对话框的)。默认的选项为 Multiple documents。本例子中保持默认选项。在此对话框中,还要对资源文件中使用的语言进行选择,选择"中文[中国]APPWZCHS.DLL"选项。单击 Next 按钮,就会进入如图 2-4 所示的 MFC AppWizard - Step 2 of 6 对话框。

④ 在如图 2-4 所示的对话框中,要求用户对应用程序支持的数据库类型进行选择。共有

图 2-4　MFC AppWizard-Step 2 of 6 对话框

4 种选择：None(不支持数据库)、Header file only(仅包含头文件)、Database view without file support(无文件支持的数据库视结构)和 Database view with file support(有文件支持的数据库视结构)。本应用程序在开发中要涉及到数据库操作，但又不是直接用 AppWizard 产生的应用程序框架进行程序设计，所以选择第 2 个选项：Heading file only，表示在产生的应用程序中只包含了进行数据库应用程序设计所需要的头文件(在 stdafx.h 文件中插入了头文件)。单击 Next 按钮，就会进入如图 2-5 所示的 MFC AppWizard – Step 3 of 6 对话框。

图 2-5　MFC AppWizard-Step 3 of 6 对话框

⑤ 在图 2-5 所示的 AppWizard – Step 3 of 6 对话框中,要求用户选择支持的 OLE 类型。共有 5 种选择:None(不支持 OLE)、Container(生成 OLE 容器应用程序)、Mini-server(生成小型 OLE 服务器)、Full-server(生成全 OLE 服务器应用程序)和 Both container and server(生成 OLE 容器和服务器应用程序)。在本应用程序中,不进行与 OLE 有关的程序功能设计,保持默认选形:None。单击 Next 按钮,进入图 2-6 所示的 MFC AppWizard-Step 4 of 6 对话框。

图 2-6 MFC AppWizard-Step 4 of 6 对话框

⑥ 在图 2-6 所示的 MFC AppWizard – Step 4 of 6 对话框中,要求对生成的应用程序包含的特征进行选择。具有以下几个可选项:Doching toolbar(具有一个固定的工具条)、Initial status bar(具有可初始化的状态条)、Printing and print preview(具有打印和打印预览功能)、Context – sensitive Help(具有上下文相关帮助功能)、3D control(3D 控件)。在本例中,选中所有的这些选项。单击 Next 按钮,就会进入图 2-7 所示的 MFC AppWizard – Step 5 of 6 对话框。

⑦ 在图 2-7 所示的 MFC AppWizard-Step 5 of 6 对话框中,用户需要对是否在生成的程序源代码中加入注释进行选择。在本例中,选择"Yes, please"(加入注释)选项。另外有一个选项是选择在产生的应用程序中如何应用 MFC,有两种选择:"As a shared DLL"(作为共享动态链接库)和"As a statically linked library"(作为静态链接库)。选择"As a shared DLL"选项。单击 Next 按钮,就会进入图 2-8 所示的 MFC AppWizard-Step 6 of 6 对话框。

在图 2-8 所示的 MFC AppWizard-step 6 of 6 对话框中,用户可以选择或修改要创建应用程序的类、文件名称,以及所要继承的基类,一般保留默认参数不变。

在这个对话框中单击"Finish"按钮,就完成了生成新程序项目 Draw 的操作过程。MFC AppWizard 将在选定的目录下生成一个应用程序项目 Draw 的所有源文件,并在当前开发平台上打开这个程序项目 Draw。对程序项目进行编译后运行,就会得到如图 2-9 的执行界面。

应用程序的执行结果是令人鼓舞的,轻而易举地就产生了一个与常见的 Windows 系统类似的窗口系统。它包含着一个主菜单,一个工具条和一个状态条等部件。在第 3 章"MFC 应用程序框架剖析"中,将对程序的组成和结构进行分析,在以后的章节中,我们以这个程序框架为

基础，逐步扩充系统的功能，最终实现一个基本的 GIS 系统。

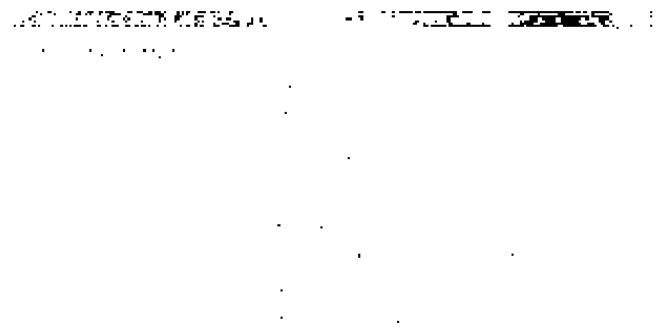


图 2-7　MFC AppWizard-Step 5 of 6 对话框



图 2-8　MFC AppWizard-Step 6 of 6 对话框

图 2-9 应用程序 Draw 的执行界面

2.2 使用 classview

ClassView 是 VC++工作平台上的一个工具,它通过对应用程序的头文件(.h)和实现文件(.cpp)进行处理,从中得到应用程序中所包含的类,并能够实现增加类、增加类的成员函数和变量,以及在头文件(.h)和实现文件(.cpp)中定位各个类、类的成员函数和变量等各种操作。

用 VC++开发应用程序时,随着系统功能的增加、代码的增多,系统具有的头文件和实现文件会越来越多,文件也会越来越大,在源代码文件中寻找类和类的成员函数、成员变量的位置将会变的困难而繁琐。通过 ClassView 可以使成员函数、变量的定位和增加类、类的成员的工作变得较为容易。

2.2.1 启动 ClassView

ClassView 的启动非常简单,在图 2-1 所示的 VC++开发工作界面上的工作区中单击"ClassView"选择标签,就会在工作区中列出当前应用程序项目所包含的类(图 2-10),用鼠标单击"+"会打开各子项目(如在一个类边上单击,会列出此类包含的成员函数和变量),单击"-"会关闭打开的子项目。

ClassView 能够完成的功能较多,在下面各节对一些常见的操作功能进行介绍。

2.2.2 编辑定位

可以利用 ClassView 的定位功能在源文件中迅速找到需要进行编辑的位置。在图 2-10 所示的操作界面上,在需要编辑的类或类的成员上双击鼠标左键,就会在文件编辑框内打开包含类的定义或类的成员的源文件,并将编辑位置移动到类或类的成员的定义处或开始处。也可以在类或类的成员上单击鼠标右键,调出一个快捷菜单,选择运行快捷菜单中的"Go to Difination"(到定义处)或"Go to Declaration"(到实际实现处)菜单项,同样可以在文件编辑框中指向类或类的成员的定义处或实现代码开始处(对于类来说,只有定义处)。

图 2-10　ClassView 的工作界面

2.2.3　增加类

在如图 2-10 所示的 ClassView 工作区中，在最高一级的列表项目上（显示“Draw Classes”处）单击鼠标右键，调出快捷菜单，在菜单中选择运行“New Class...”菜单项，就会进入图 2-11 所示的对话框。

在图 2-11 所示的对话框中，在“Class Type”列表框中有 3 种类型的类供选择：MFC Class、Generic Class、Form Class（在 VC++5.0 中，只有 MFC Class 和 Generic Class 两个选项）。其中，MFC Class 选项用来选择增加 MFC 类的派生类；Generic Class 选项用来选择增加一般的 C++ 类；Form Class 选项用来选择增加几个特殊 MFC 类的派生类，通过选择 Form Class 选项可选择增加 CDialog、CRecordView、CDaoRecordView、CFormView 等类的派生类。图 2-11 所示的界面是选择 MFC Class 选项后得到的。

（1）增加 MFC 派生类

进行增加一个 MFC 派生类的操作时，需要作出以下选择：

选择基类

如果增加的类是从已有的 MFC 类派生而来的，则需要对继承的基类进行选择。例如，在 2.1.2 小节生成的应用程序 Draw 框架中，CDrawDoc 类是由 MFC 文档类 CDocument 类派生而来的；CView 类是由窗口类 CWnd 类派生而来的。从“Base Class”选择列表框中选择基类。

确定类的存放位置

此项是确定增加类的定义和实现部分分别所在的文件，即这个类的头文件和实现文件。默认情况下，当你输入增加类的名称时，系统自动将头文件和实现文件设置成与类的名称相同（如在对话框的“Name”编辑框中输入类名称 CLine 时，则头文件名称为 line.h，实现文件名称为

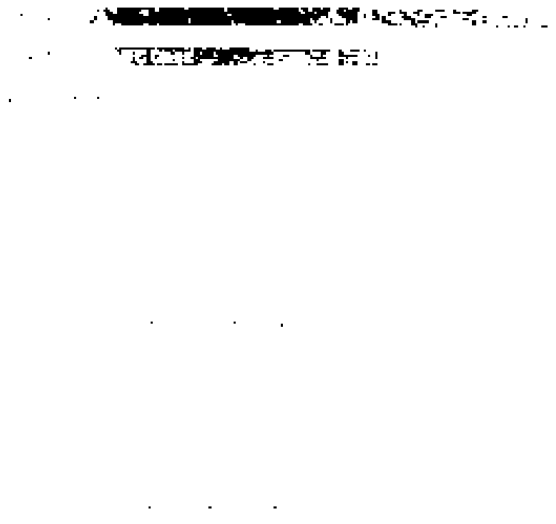


图 2-11　增加 MFC 类对话框

line.cpp)。如果需要人为确定头文件和实现文件,则需单击"Change..."按钮,进入图 2-12 所示的对话框,在这个对话框中输入头文件和实现文件的名称,可以采用已经有的文件,也可以重建新的文件。头文件和实现文件也可以在同一个文件中(一般在.cpp 文件)。按 OK 退出文件选择对话框就确定了类的头文件和实现文件。

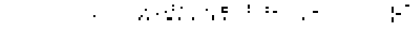


图 2-12　类的头文件和实现文件名称管理对话框

确定了以上参数后,在"Name"编辑框中输入增加类的名称,按中图 2-11 所示对话框中的"OK"按钮,系统就会在当前应用程序 Draw 中增加类的定义和实现代码部分。应用程序 Draw 中的 CDrawDoc、CDrawView 等几个类,就是在程序生成阶段由上面讨论的增加类的工具产生的,在此不再举例说明。

(2) 增加 Form 派生类

在图 2-11 对话框的"Class Type"选择列表框中选择 Form Class 选项,就变成增加 Form 派生类的对话框界面,与图 2-11 增加 MFC 派生类界面不同的是,"Dialog ID"选择列表框处于可选

择状态。增加 Form 派生类的过程与增加 MFC 派生类相同，唯一不同的是需要从“Dialog ID”列表选择框中选择要增加的 Form 派生类所采用的对话框资源。

(3) 增加一般类

在图 2-11 所示的对话框中，从“Class Type”列表选择框中选择 Generic Class 选项，就会进入图 2-13 所示的对话框。在这个对话框中可以增加一般 C++ 类，操作过程如下：

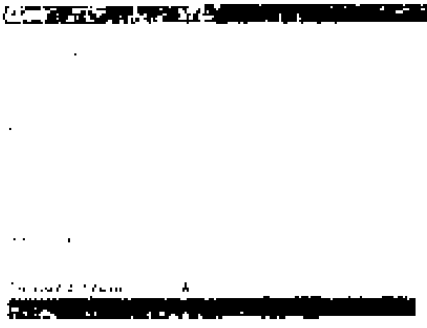


图 2-13　增加一般类的对话框

① 按“Change...”按钮，进入图 2-12 所示的对话框输入类的头文件和实现文件。

② 在“Base Class[es]”列表框中输入要继承的基类(可以一个，也可以多个)，用鼠标在列表框“Base Class[es]”中点中，就可以出现编辑框和列表框输入基类的名称和选择继承基类时的类型。

③ 在“Name”编辑框中输入类的名称。按中“OK”按钮，就在当前应用程序 Draw 中增加一个一般 C++ 类。

按照以上的步骤，在应用程序 Draw 中增加一个叫做 CLine 的类，类的头文件在 drawdoc.h，实现文件在 drawdoc.cpp。

操作完成后，在头文件 drawdoc.h 中可以找到 CLine 类的定义：

```
class CLine
{
public:
    CLine();                //构造函数
    virtual ~CLine();       //析构函数
};
```

在实现文件 drawdoc.cpp 中,可找到空的构造函数和析构函数实现代码:

```
CLine::CLine()
{
}
CLine::~CLine()
{
}
```

2.2.4 增加类的成员

用 ClassView 可以用来增加类的成员,这些成员包括成员变量、成员函数,以及重载虚函数等。以下简介一些主要操作过程。

(1) 增加成员变量

在图 2-10 所示的工作区中,在需要增加成员变量的类上单击鼠标右键,在弹出的快捷菜单中选择运行"Add Member Variable..."菜单项,就会进入图 2-14 所示的对话框。

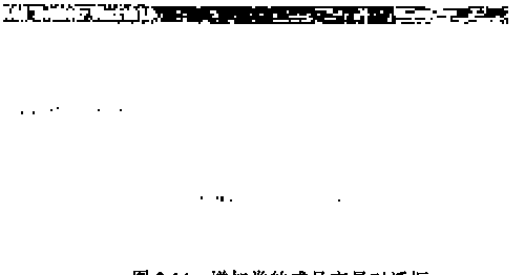


图 2-14 增加类的成员变量对话框

成员变量的增加过程非常简单,在对话框中的"Variable Type"编辑框中输入变量的类型(如 int,float,char,CPoint 等),在"Variable Name"编辑框中输入变量的名称,在"Access"框中通过按钮选择变量的类型,按"OK"按钮就完成了增加成员变量的操作。

增加成员变量就是在类中定义了一个成员变量,与手工增加过程完全一样,并不省事。

(2) 增加成员函数

在图 2-10 所示的工作区中,在需要增加成员函数的类上单击鼠标右键,在弹出的选择菜单上选择运行"Add Member Function..."菜单项,就会进入图 2-15 所示的对话框。

成员函数的增加方法与成员变量类似。不同的是,除了需要给定函数的返回类型、函数名称、函数的类型外,还可以通过选择"Static"或"Visutal"选择框将函数定义成静态函数或虚函数。增加成员函数操作完毕后,会在类中增加成员函数的定义,在类的实现文件中增加一个空函数。

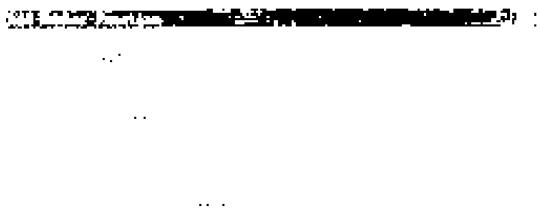


图 2-15　增加类的成员函数对话框

(3) 重载虚函数

如果一个类是由 MFC 类派生的，那么可以在这个类中重载 MFC 类中的虚函数。在图 2-11 所示的 ClassView 工作区中，在 MFC 派生类上单击鼠标右键，在弹出的快捷菜单中选择运行“Add Virtual Function...”菜单项，就会进入重载虚函数的对话框。图 2-16 所示的对话框中列出的是 CDrawDoc 类(其基类是 CDocument)中所能重载的虚函数和已经重载的虚函数的列表。在左边的列表框中选择要重载的虚函数，点击“Add Handler”(增加)、“Add and Edit”(增加并马上进行编辑)按钮，就可以在当前文档类 CDrawDoc 中重载这个虚函数。此功能与 ClassWizard 中的重载虚函数的功能类似(见 2.5.5 小节)。

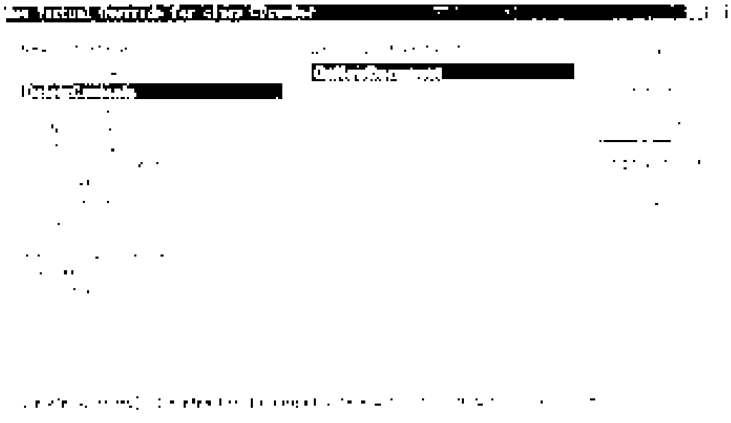


图 2-16　重载虚函数对话框

2.3　使用 ResourceView

ResourceView 及其相应工具 AppStudio 是 VC++ 对当前应用程序的资源进行可视化管理的

工具,利用它可以对程序所用的各种资源(菜单、工具条、对话框、位图、图标、加速键、热键等)进行可视化的管理。ResourceView 的工作基础是资源文件(.rc)、资源 ID 号定义文件(Resource.h)和一些实际的资源(一些图像文件、图标文件等)。在 2.1.2 小节生成的应用程序 Draw 中,资源文件是 draw.rc,资源 ID 号定义文件是 resource.h,res 目录下的图像、图标文件等是 draw.rc 需要的一些具体资源。另外,在 res 目录下,有一个 draw.rc2 文件,这个文件是用来手工定义资源用的,ResourceView 工作时并不处理这个文件,所以在这个文件中定义的资源在可视化的编辑界面上并不存在。而程序进行编译时,这个资源文件中的内容被插入到资源文件 draw.rc 中(在文件 draw.rc 中可以找到一句代码 #include "res\\draw.rc2")。

2.3.1 ResourceView 的调用

在 VC++ 工作平台下单击 "ResourceView"标签,就会进入到图 2-17 所示的资源管理状态。

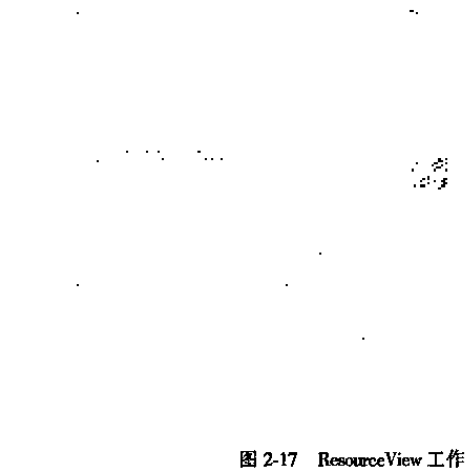


图 2-17 ResourceView 工作区界面

2.3.2 ResourceView 工作分析

本节并不打算讨论对各种资源进行可视编辑的具体操作步骤,而只分析利用 ResourceView 对资源进行可视化管理时的工作过程。

(1) 资源的几个要素

VC++ 的应用程序中下包含了开发 Windows 应用程序的各种资源,如菜单、对话框、对话框内的各种控制等等,不论哪一种资源,一般都包含以下几个基本要素:

类别: 即这个资源项属于哪一类资源。

位置和内容: 这一项对于不同的资源是不一样的,主要用来表示资源项的位置、显示内容等。

风格： 此参数表示这个资源具有这类资源的哪些风格，以及具有哪些可选参数。

ID号： 标识资源项的特征号，也就是这个资源的识别号。在程序运行时，就是通过这个ID号标识这个资源，对资源进行操作，也是通过这个ID号标识资源发出的消息。对于资源的ID号，需要给它定义一个ID值，ID值是在文件resource.h中定义的。

各种资源在资源文件中(如应用程序Draw的draw.rc文件)是以文本形式来描述的。可以在资源文件draw.rc中找到应用程序Draw的菜单资源定义：

```
IDR_DRAWTYPE MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "文件(&F)"
  BEGIN
    MENUITEM "新建(&N)\tCtrl+N",          ID_FILE_NEW
    MENUITEM "打开(&O)...\tCtrl+O",       ID_FILE_OPEN
    MENUITEM "关闭(&C)",                  ID_FILE_CLOSE
    MENUITEM "保存(&S)\tCtrl+S",          ID_FILE_SAVE
    MENUITEM "另存为(&A)...",             ID_FILE_SAVE_AS
    MENUITEM SEPARATOR
    MENUITEM "打印(&P)...\tCtrl+P",       ID_FILE_PRINT
    MENUITEM "打印预览(&V)",              ID_FILE_PRINT_PREVIEW
    MENUITEM "打印设置(&R)...",           ID_FILE_PRINT_SETUP
    MENUITEM SEPARATOR
    MENUITEM "最近文件",                  ID_FILE_MRU_FILE1, GRAYED
    MENUITEM SEPARATOR
    MENUITEM "退出(&X)",                  ID_APP_EXIT
  END
  ......
END
```

对于一个菜单项，MENUITEM注明了资源的类别，引号内是菜单的显示内容，ID_FILE_×××是菜单项的ID号，GRAYED是一个可选的菜单的风格。从draw.rc中可以找到其他资源的定义代码。

对于资源的ID号，需要给它定义一个值。ID号的值是在文件resource.h中定义的，对于资源文件中每个资源项的ID号，AppStudio自动在文件resource.h中为它分配一个值，从resource.h文件中可以找到对菜单资源ID号IDR_DRAWTYPE的定义：

```
#define IDR_DRAWTYPE            129
```

读者可能会问，对于资源文件draw.rc中很多资源的ID号，为什么不能在resource.h找到其ID值的定义？原因是这样的，对于很多ID号，如ID_FILE_OPEN、ID_FILE_NEW等，这些ID号的值已经在MFC中进行了定义。读者如果有兴趣，可以在头文件afxres.h中找到这些ID值的定义，afxres.h文件被插入到资源文件draw.rc中(#include <afxres.h>)。以下是在afxres.h文件中部分ID号的定义：

```
.....
# define ID_FILE_NEW                0xE100
# define ID_FILE_OPEN               0xE101
# define ID_FILE_CLOSE              0xE102
# define ID_FILE_SAVE               0xE103
# define ID_FILE_SAVE_AS            0xE104
# define ID_FILE_PAGE_SETUP         0xE105
# define ID_FILE_PRINT_SETUP        0xE106
# define ID_FILE_PRINT              0xE107
# define ID_FILE_PRINT_DIRECT       0xE108
# define ID_FILE_PRINT_PREVIEW      0xE109
# define ID_FILE_UPDATE             0xE10A
# define ID_FILE_SAVE_COPY_AS       0xE10B
# define ID_FILE_SEND_MAIL          0xE10C
.....
```

可以直接编辑资源文件 draw.rc 和 ID 号定义文件 resource.h 来管理资源。但是这种方法不方便而且容易出错。例如,很难光凭直觉想象快速设计出一个对话框界面,可视化资源编辑工具非常好地解决了这个问题。

(2) ResourceView 工作过程分析

ResourceView 及其工具,类似于一个翻译器,它将资源文件中文本描述的资源翻译成可视化的的资源形态,并在可视化状态下修改;修改完毕后,又把资源以文本方式写入到资源文件中。其工作过程如下:

读入资源

当打开一个资源文件时(如 draw.rc),ResourceView 相关的工具 AppStudio 将资源文件中的内容顺序读入并翻译成各种资源,通过 ResourceView 列出。在这个翻译过程中,将要进行很多检查,如各种资源的 ID 号是否定义(从 Resource.h 中检查),ID 号取值是否重复、是否合法,插入到资源文件中的其他实际资源是否存在(如资源中需要一个位图文件 res \ toolbar.bmp 作为工具条,检查这个位图文件是否存在)等等。如果没有错误,打开资源文件成功。

编辑资源

对于装入的资源,可以调用 AppStudio 在可视化状态下进行编辑。所谓可视化状态,就是这些资源在系统运行时的实际状态,是一种所见即所得的的编辑形式。

保存资源

当对资源进行编辑完成后,AppStudio 把资源用文本的形式写入资源文件。对于新增加或改动的资源,将会在资源文件中增加或修改描述这些资源的文本内容。对新增的 ID 号,在 Resource.h 中自动为其定义 ID 值。

(3) 具体分析增加一个菜单的过程

对应用程序 Draw 中的菜单资源 IDR_DRAWTYPE 进行修改,增加一个菜单“图形绘制”,

在这个菜单中增加一个菜单项“绘制直线”(如图 2-18)。这个菜单项的 ID 号为 ID _ DRAW _ LINE,操作完成后,在资源文件 draw.rc 中会找到对这个菜单项的定义(黑体部分):

图 2-18　增加菜单项后的 IDR _ DRAWTYPE 菜单资源

```
.....
    POPUP "编辑(&E)"
    BEGIN
        MENUITEM "撤消(&U) \ tCtrl + Z",          ID _ EDIT _ UNDO
        MENUITEM SEPARATOR
        MENUITEM "剪切(&T) \ tCtrl + X",          ID _ EDIT _ CUT
        MENUITEM "复制(&C) \ tCtrl + C",          ID _ EDIT _ COPY
        MENUITEM "粘贴(&P) \ tCtrl + V",          ID _ EDIT _ PASTE
    END
    POPUP "&D 图形绘制"
    BEGIN
        MENUITEM "&L 绘制直线",                   ID _ DRAW _ LINE
    END
    POPUP "查看(&V)"
    BEGIN
        MENUITEM "工具栏(&T)",                    ID _ VIEW _ TOOLBAR
        MENUITEM "状态栏(&S)",                    ID _ VIEW _ STATUS _ BAR
    END
.....
```

在 resource.h 中发现菜单项 ID 号的定义:

```
# define ID _ DRAW _ LINE          32771
```

对于其他资源的增加和管理,工作程序与增加一个菜单时类似,在此不再多述。

2.4　使用 FileView

FileView 是 VC++ 开发平台上用来管理一个开发项目所包含的各种源文件的工具。2.1 节用 AppWizard 开发完成的应用程序 Draw,是由多个文件组成的程序包,其中包含大量的头文件(.h 文件)、实现文件(.cpp)文件、资源文件(.rc)、定义文件(.def)等。通过 FileView 可以对一个开发项目包含的文件列表进行管理。这个文件列表指导编译程序对哪些文件进行编译,同时也用来指导 ClassView 和 ClassWizard 等对哪些文件进行处理。在应用程序设计中,有时会发生程序中的类没有在 ClassView 中列出的情况,原因就是类的头文件没有放到文件列表中(因为头文件在实现文件中被插入,所以只要实现文件在项目中,在程序编译时头文件也就被编译程序处理了,所以即使头文件没有在项目中,但不影响程序的编译)。

2.4.1 打开一个程序项目

一个程序项目所包含的文件信息存放在.dsp文件中(应用程序Draw中的draw.dsp文件)。在VC++中,.dsw文件中存储的是一个工作平台的信息。因为在一个工作平台上可以有多个程序项目,具体包含哪些程序项目、开发项目间的依赖关系等都存储在.dsw文件中(应用程序Draw中的draw.dsw文件)。

打开一个程序项目的操作非常简单,在VC++工作界面下运行File菜单下的"Open Worksapce..."就会出现一个对话框,选择.dsw文件或者.dsp文件就可以打开一个程序项目。在VC++的早期版本,是通过.mak文件来组成程序项目的。在VC++工作平台上打开早期版本的开发项目时,需要选择.mak的文件类型。在VC++工作平台上,如果要打开的程序项目原来是在较低版本上形成的,在打开操作时,需要进行转换(确认打开操作时的提示对话框),形成现版本所用的.dsp和.dsw文件。

2.4.2 FileView相关的操作功能

(1) 启动FileView

在打开一个程序项目的情况下,点中VC++工作区中的FileView标签,就会进入图2-19所示的FileView操作界面。

(2) 在项目中增删文件

可以随时在项目中增加一个文件或将一个文件从项目文件列表中去掉。

增加文件到程序项目中

在程序项目中增加一个文件的方法有多种,现在介绍其中常用的两种:

一种方法是通过运行VC++工作平台上的菜单"Project"下的"Add Project..."菜单项。如果需要加入的是一个已经存在的文件,则运行"Add Project..."菜单项中的"Files"菜单项调出文件选择窗口,选择需增加的文件;如果需要增加的是一个新建文件,则需要运行"Add Project..."菜单的"New"菜单项调出New对话框,创建一个新建文件。FileView会根据文件的后缀自动将文件增加到对应的列表项目下(如把.cpp类文件放在"Source Files"项下,而把.h文件放在"Header Files"项下)。

另一种方法是在图2-19中的工作区中单击鼠标右键调出快捷菜单来增加一个文件。如在"draw files"项上单击鼠标右键,就会弹出一个快捷菜单,运行快捷菜单中的"Add File To Project..."菜单项,就可以进行与第一种方法相同的操作。在一个项目(如Source Files)上单击鼠标右键,运行弹出的快捷菜单中的菜单项"Add Files to Folder...",通过弹出的文件选择对话框选择需要加入到当前项下的文件,就会把这个文件增加在当前项目下(单击Source Files操作时,会把文件放在Source Files项下,而不管其后缀如何)。

从程序项目中删除文件

从程序项目中删除一个文件很简单,用鼠标在FileView工作区中的文件列表中选择要删

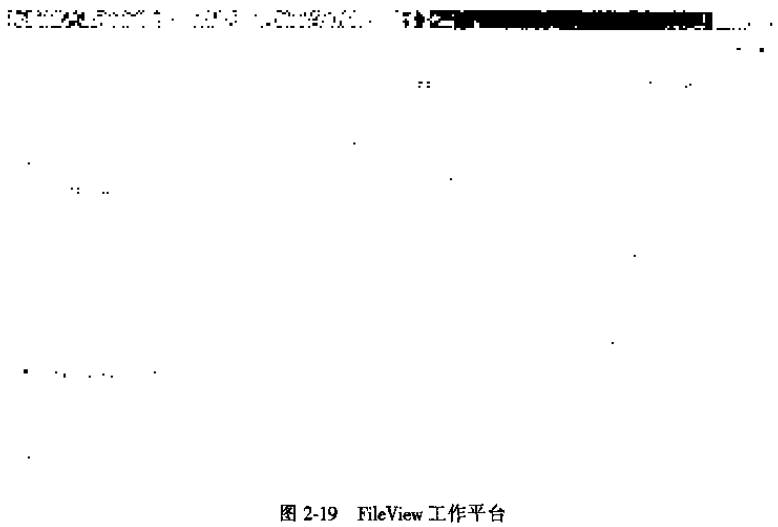


图 2-19　FileView 工作平台

除掉的文件，按删除键(Del)就会把这个文件从项目列表中删除。注意，这个删除只是从项目所包含的文件列表中删除了这个文件，而不是把这个文件从计算机中删除。

(3) 在 FileView 中增加目录项

在 FileView 工作区的文件列表中有一些目录项(如 Source Files、Header Files 等)，这些相当于管理文件时的目录。每个目录存储一定后缀的文件，当用“Add Files to Project...”增加一个文件时，根据文件的后缀自动把文件列在合适的目录项下。这个目录项也是可以任意创建和删除的，在 FileView 工作区的目录项上单击鼠标右键，运行弹出的快捷菜单中的菜单项“New Folder...”，就会弹出一个如图 2-20 的对话框，在这个对话框的“Name of The New Folder”编辑框中输入目录项的名称，在“File extension”编辑框中输入放在这个目录项中的文件的后缀，按“OK”按钮就会在选中的目录下建立一个子目录项。当再用“Add Files to Project”菜单项在程序项目中增加文件时，FileView 就会把后缀为 .gif、.tif、.bmp、.pcx 的文件放在这个新增的目录项下。

图 2-20　增加 FileView 目录项的对话框

删除 FileView 工作区中的目录项的操作也很简单，只需用鼠标键选中要删除的目录项，按删除键(Del)就会把这个目录项及目录项下的文件从文件列表中删除(并非实际删除文件)。

2.5 使用 ClassWizard

MFC ClassWizard 是 VC++程序设计中一个非常重要的工具。Windows 应用程序是通过消息来控制流程的,MFC AppWizard 生成的应用程序为通过消息设计程序提供了很好的框架。程序设计者甚至不必知道也不必参与控制消息传递的实际流程,而只需掌握简单的使用方法就可以达到程序设计的目的。MFC ClassWizard 为管理消息处理函数等提供了一个强大的工具。

2.5.1 Windows 消息

Windows 应用程序一般是由消息驱动的(message - driven)。为处理诸如击鼠标键、窗口移动或改变大小、键盘输入等情况,Windows 会向特定的窗口发送消息。在 VC++中,一般的消息处理被包含在 MFC 中,用户特定的消息由程序设计者自己处理。应用类 CWinApp 中的 Run()函数从 Windows 消息队列中获取消息,并在消息循环中把消息发送到适当的窗口。Windows 程序中的消息主要有三种类型:窗口消息、控制的通知消息和命令消息。

(1) 窗口消息

窗口消息包括由 WM_开头的消息(除 WM_COMMAND 消息外),一般由窗口对象或视图对象处理。窗口消息往往带有参数,它是在窗口创建、改变大小、删除过程中,以及在受到操作(如受鼠标操作)时产生的,如 WM_LBUTTONDOWN 消息是在窗口中单击鼠标左键时产生的,WM_SIZE 消息在窗口大小改变时产生的。窗口消息是由窗口对象接收并处理的。窗口对象指的是从 CWnd 中派生出的类的对象,如从 CWnd,CFrameWnd,CMDIFrameWnd,CMDIChildWnd,CView,CDialog 等 MFC 类或这些类的派生类定义的对象(这些类都是由窗口类 CWnd 直接或间接派生的)。

(2) 控制的通知消息

控制是一个小的子窗口,属于其他窗口(如对话框),能够接受操作并向父窗口发送消息。常见的控制有:按钮(Button)、列表框(ListBox)、编辑框(Edit)、复合框(ComboList)、滚动条(ScrollBar)等。在利用资源编辑器编辑对话框时,工具条中列出的都是一些控制。对于 ActiveX 控件,像别的控制一样能发出通知消息(ActiveX 控件中的"事件")。控制消息也是由窗口对象接收并处理的。

(3) 命令消息

命令消息主要包括由用户交互对象发送的 WM_COMMAND 消息,用户交互对象指的是程序操作界面上的菜单、工具条、快捷键等。命令消息的传递和处理方式与其他两种消息略有不同,命令消息可以被多种对象接收、处理。文档类、框架类、应用程序类、视图类等的对象均可以接收并对命令消息作出反应。一个对象接收到命令消息后,如果不处理,就把命令消息传递到其他的对象,直到被某个对象处理。命令消息在各个对象中的传递顺序如表 2-2 所示。当

某个命令消息在多于一个对象中有处理函数时，只有传递顺序在前面的窗口能够接收并处理消息。

表 2-2　命令消息传递顺序

接收命令消息的对象	传递顺序
MDI 边框主窗口对象	1. 当前活动的 CMDIChildWnd 对象
(CMDIFrameWnd)	2. 边框主窗口对象本身
	3. 应用对象(CwinApp)
文档窗口对象	4. 当前活动的视对象(Cview)
(CframeWnd, CMDIChildWnd)	5. 文档窗口本身
	6. 应用对象(CwinApp)
视图对象	7. 视图对象本身
	8. 与视对象相连的文档
文档对象	9. 文档本身
	10. 文档所属的文档模板对象
对话框对象	11. 对话框对象本身
	12. 对话框对象的父窗口对象
	13. 应用对象(CwinApp 对象)

在 MFC 中定义了一些命令消息，MFC 应用程序中的标准菜单的 ID 号(如“文件”菜单下“新建”菜单项，ID 号 ID_FILE_NEW)都已经在 MFC 中定义(定义在头文件 afxres.h 中)，这些消息的处理函数也被定义在 MFC 中，程序设计者可以在当前应用程序中为这些消息定义消息处理函数，来代替定义在 MFC 类中的消息处理函数。

2.5.2　消息处理函数

消息处理函数指的是与某个消息对应的函数。消息处理函数的执行是由其对应的消息引发的，这个机制构成了 Windows 程序控制流程的基础。某个类对象中的消息处理函数是与这个类对象能够得到的一个消息对应的，当这个消息发生时，这个函数就会被执行。

消息处理函数的定义和使用方法与一般的成员函数不同。一般成员函数包括两部分：函数的定义和函数的实现部分。定义一个一般成员函数时，在类中定义函数的原型(一般在头文件中)，在实现文件中实现函数的实现代码。而消息处理函数除了这两部分以外，还有第三部分：消息映射部分，通过消息映射在消息处理函数和引发它的消息之间建立联系。消息处理函数的定义方法与一般成员函数也不完全相同。以下对各类消息处理函数的定义、映射的方法进行讨论。

(1) 窗口消息处理函数

窗口消息处理函数是一些已经被保护的成员函数。窗口消息处理函数有固定的名称，通过对应的消息激活。如，在应用程序 Draw 的 CDrawView 类中，加入一个单击鼠标左键激活的处理函数，其手工完成步骤如下：

首先,在视图类 CDrawView 中加入函数的定义:

```
......
protected:
//{{AFX_MSG(CDrawView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);  //单击鼠标左键调用
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
......
```

其次,在 CDrawView 类的实现文件 drawview.cpp 中加入函数的实现代码:

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CView::OnLButtonDown(nFlags, point);
}
```

最后,在实现文件 drawview.cpp 中,加入函数的消息映射机制:

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
//{{AFX_MSG_MAP(CDrawView)
    ......//省略以上代码
    ON_WM_LBUTTONDOWN()  //单击鼠标左键,OnLButtonDown 函数的消息映射
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

(2) 命令消息处理函数

对于一个命令交互对象,一般有两种类型的命令消息:一种是 WM_COMMAND 消息,这种消息是在执行命令交互对象时发出的,如果一个命令交互对象没有对应的消息处理函数,那么这个交互对象就处于失效不能执行状态。还有另外一种 ON_UPDATE_COMMAND_UI 消息,这种消息是在更新一个命令交互对象的外观前发出的,一般通过对这个消息的处理,来达到对这个命令交互对象显示状态的控制(如失效、激活、加检查标志等)。在应用程序 Draw 的资源中增加了一个包含菜单项"绘制直线"的菜单"图形绘制"(如图 2-18 所示,菜单"绘制直线"的 ID 号为 ID_DRAW_LINE),用手工在 CDrawView 类中增加两种类型的消息处理函数的过程如下:

首先,在 CDrawView 类中加入函数的定义代码:

```
protected:
//{{AFX_MSG(CDrawView)
    ......//省略以上代码
    afx_msg void OnDrawLine();                      //绘制直线
    afx_msg void OnUpdateLine(CCmdUI* pCmdUI);      //是否能够执行"绘制直线"菜单项
//}}AFX_MSG
```

与窗口消息不同,命令消息处理函数是新定义的,可以定义成任何符合命名规则的名称。

然后，在实现文件 drawview.cpp 中加入函数的实现代码：

```
void CDrawView::OnDrawLine()
{
    //加入实际的处理代码
}

void CDrawView::OnUpdateLine(CCmdUI * pCmdUI)
{
    //加入实际处理代码
}
```

最后，在实现文件 drawview.cpp 中加入函数的消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
//{{AFX_MSG_MAP(CDrawView)
    ......//省略以上代码
    ON_COMMAND(ID_DRAW_LINE, OnDrawLine)  //开始直线绘制
    ON_UPDATE_COMMAND_UI(ID_DRAW_LINE, OnUpdateLine)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

(3) 控制消息处理函数

控制消息与命令消息类似，与命令消息不同的是，根据控制的不同，所激发的消息种类和数量也不一样，如对于 Button 控制来说，有 BN_CLICKED(鼠标单击按钮时发生)、BN_DOUBLECLICKED(鼠标双击按钮时发生)等消息；对于 Edit 控制来说，有各种诸如 EN_×××××× 的消息。对于不同的消息，可以增加不同的消息处理函数。手工增加控制消息处理函数的方法与命令消息类似。

2.5.3　ClassWizard 的作用和启动

(1) ClassWizard 的作用

在 2.5.2 小节中，讨论了消息处理函数及手工增加的方法，读者会发现，增加一个消息处理函数，方法虽然简单，但实行起来有困难，此困难主要表现在以下几个方面：

① 很难记清某个类能够得到的消息。因为消息的数量非常多，即使对于熟练编程者也不能完全记清，要花费大量的时间查阅资料。

② 增加过程麻烦。对于各种窗口消息，如果不查资料就不知道其保护的消息处理函数是什么，即使知道，增加起来也比较麻烦。增加一个消息处理函数需要三个步骤(定义、实现函数、消息映射)。另外，各种保护过的消息处理函数名称的繁琐，以及各种映射机制的多样性等，都使开发者不能轻松地增加一个消息处理函数。

MFC ClassWizard 是微软公司的设计者们为解决这个问题而设计的，是一个非常好的辅助

设计工具,使你轻松地完成对消息处理函数、虚函数、映射变量、自动化变量,以及 ActiveX 事件等的管理工作。

(2) ClassWizard 的启动

ClassWizard 的启动方式有多种,以下是两种常用的方法:一是运行 VC++开发平台中"View"菜单下的"Class Wizard..."菜单项,或者单击快捷键 CTRL_W;二是在 VC++工作平台的文件编辑窗口(当窗口中打开属于当前程序项目的源文件时)中单击鼠标右键,选择运行弹出的快捷菜单中的菜单项"ClassWizard..."。

不论用那种方法启动 ClassWizard,系统都会进入图 2-21 所示的对话框。对话框的顶部有五个选择标签"Message Maps"、"Member Variable"、"Automation"、"ActiveX Events"和"Class Info"。选择不同的标签,会出现不同的对话框操作界面,图 2-21 所示的对话框是在选择"Message Maps"标签后得到。在五个不同标签对应的操作界面中,都有"Project"和"ClassName"两个列表选择框。从"Project"列表选择框中选择 ClassWizard 要管理的程序项目(一般情况下,一个工作平台上只有一个程序项目,2.8 节将讨论在一个开发平台上管理多个程序项目的方法)。在"ClassName"列表选择框内列出的是在选定程序(如 "Draw")中符合 ClassWizard 工作条件的类。并不是程序项目中的所有类都在这个列表选择框中列出,列出的只是那些包含消息处理函数、映射变量、ActiveX 事件等标识代码的类。

图 2-21 ClassWizard 的工作界面

2.5.4 ClassWizard 管理消息处理函数

以下对 ClassWizard 管理消息处理函数的功能操作进行介绍。在 ClassWizard 对话框中选择"Message Maps"标签,就进入图 2-21 所示的对话框。在这个对话框界面中,可以对类的消息

处理函数函数进行管理。

(1) 管理窗口消息处理函数

窗口消息包括由 WM_开头的消息(除 WM_COMMAND),一般由窗口对象处理。窗口对象即指从 CWnd 中派生的类的对象,CWnd、CFrameWnd、CMDIFrameWnd、CMDIChildWnd、CView、CDialog 等 MFC 类,都是直接或间接由 MFC 派生而来的,能够接受窗口消息并加入对应的消息处理函数。

增加一个窗口消息处理函数

以应用程序 Draw 为例,说明用 ClassWizard 增加一个窗口消息处理函数的过程。在 VC++ 工作界面下打开 Draw 项目,启动 ClassWizard,进入到图 2-21 所示的对话框界面。从"Class Name"复合框中选择视图类 CDrawView,这样在"Object IDs"列表框中就会列出各种消息来源的组件。从"Object IDs"列表框中选择 CDrawView,在列表框"Messages"中列出的就是 CDrawView 类对象能够从窗口对象 CWnd 接受到的窗口消息和各种可重载的虚函数。从中可以找到一个名为 WM_LBUTTONDOWN 的消息,用鼠标选中这个消息。此时,"Add Function"按钮处于可执行状态,按中此按扭,就会在 CDrawView 类中增加一个被 CWnd 保护的消息处理函数 OnLButtonDown。在"Member functions"列表框中能够发现这个函数及其对应的消息映射机制:"OnLButtonDown ON_WM_LBUTTONDOWN"。在函数前有一个 W 标志,表示这个函数是一个窗口消息处理函数。

增加窗口消息处理函数过程的分析

以上讨论的用 ClassWizard 增加窗口消息处理函数的过程,实际上是完成了本章 2.5.2 小节中用手工增加窗口消息处理函数的过程,对于应用程序 Draw 的代码,做了如下修改:

① 在视图类 CDrawView 中,增加了消息处理函数的定义代码:

```
protected:
//{{AFX_MSG(CDrawView)
    ......//省略以上代码
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);  //单击鼠标左键调用
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

② 在 CDrawView 类的实现文件 drawview.cpp 中,增加了函数的实际代码:

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CView::OnLButtonDown(nFlags, point);
}
```

③ 在实现文件 drawview.cpp 中,增加了函数的消息映射机制。在实现文件 drawview.cpp 中可以找到增加的消息映射机制:

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
//{{AFX_MSG_MAP(CDrawView)
```

```
    ……//省略以上代码
    ON_WM_LBUTTONDOWN()    //单击鼠标左键,OnLButtonDown 函数的消息映射
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

ClassWizard 识别和处理窗口消息处理函数所需要的标识

CLassWizard 通过固定的代码来识别窗口消息处理函数,通过以下两行代码来标识消息处理函数的定义:

```
//{{AFX_MSG(CDrawView)        //ClassWizard 标识消息处理函数定义的起始代码
//}}AFX_MSG                   //ClassWizard 标识消息处理函数定义的结束代码
```

扩号中的内容(CDrawView)是消息处理函数所在类的名称,只有将消息处理函数定义在以上两行代码间时,才能被 ClassWizard 识别和处理。消息处理函数与普通函数的定义格式略有不同,需要在函数定义前加标识符 afx_msg。

在实现文件中,消息映射机制写在如下的两行代码间而被 ClassWizard 识别:

```
//{{AFX_MSG_MAP(CDrawView)    //ClassWizard 标识消息处理函数映射机制的起始代码
//}}AFX_MSG_MAP               //ClassWizard 标识消息处理函数映射机制的结束代码
```

同样,扩号中的内容是拥有消息映射机制的类的名称,只有当一个消息处理函数的消息映射机制写在以上两行代码间时,才能被 ClassWizard 识别和处理。

删除窗口消息处理函数

用 ClassWizard 删除窗口消息处理函数的方法很简单。在图 2-21 所示的操作界面中,从"Member Functions"列表框中选择刚刚增加的 OnLButtonDown 函数,"Delete Function"按钮处于激活可执行状态,按中此按钮,系统会提示是否对 OnLButtonDown 函数进行删除,并提示必须用手工从实现文件中删除函数的实际代码,确认后就删除了这个消息处理函数。

以上操作只是从视图类 CDrawView 中删除了函数的定义代码,从实现文件中删除了映射消息机制,而对于实现文件中的函数实现代码,需要手工进行删除。这样做的目的是避免误操作而删除大量有用的代码。

(2) 命令消息处理函数

以上讨论了利用 ClassWizard 管理窗口消息处理函数的方法,现在讨论利用 ClassWizard 管理命令消息处理函数的方法。

在 2.5.1 小节,已对命令消息进行了简述。与窗口消息不同,命令消息可以在程序框架中的多个类对象中按一定的顺序传递,直到找到对应的消息处理函数为止。反映到消息处理函数上,就是可以在属于框架的各个类中定义消息处理函数。例如,在应用程序 Draw 中加入了一个"图形绘制"菜单,其中有一个"绘制直线"菜单项,对于这个菜单项的消息处理函数,可以

在文档、视图、框架、应用等多个类中定义。如果在多于一个类中定义了同一个命令消息的处理函数，则只有先得到命令消息的对象中的消息处理函数才能够被运行。

增加命令消息处理函数

用 ClassWizard 增加命令消息处理函数的操作方法，与增加窗口消息处理函数类似，在此举例介绍其操作过程。在应用程序 Draw 的菜单资源 IDR _ DRAWTYPE 中增加了一个菜单项“绘制直线”(ID 定为 ID _ DRAW _ LINE)，用 ClassWizard 在类 CDrawView 中增加消息映射函数的步骤如下：

进入 ClassWizard，在“Class Name”列表框中选择 CDrawView 类，在“Object Ids”列表框中选中“绘制直线”菜单项的 ID 号 ID _ DRAW _ LINE。此时，在 Messages 列表框中有两个消息可供选择：COMMAND 和 UPDATE _ COMMAND _ UI。选择 COMMAND 消息，单击“Add Function”按钮，与增加窗口消息处理函数时不同，会出现一个对话框要求输入消息处理函数的名称(因为命令消息处理函不像窗口消息具有已被保护的处理函数)，如果接受默认的函数名称 OnDrawLine，就会在 CDrawView 类中增加一个命令消息处理函数 OnDrawLine。用同样的方法可以在 CDrawView 类中增加一个 UPDATE _ COMMAND _ UI 消息的处理函数 OnUpdateCommandUI。按“OK”按钮退出对话框，就完成了增加命令消息处理函数的操作。以上操作，在相应的源文件中对程序代码进行了如下修改：

① 在视图类 CDrawView 中，增加了消息处理函数的定义代码：

```
//{{AFX_MSG(CDrawView)
.....//省略以上代码
afx_msg void OnDrawLine();
afx_msg void OnUpdateDrawLine(CCmdUI* pCmdUI);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

② 在类 CDrawView 的实现文件 drawview.cpp 中，增加了函数的实现代码：

```
void CDrawView::OnDrawLine()
{
    // TODO: Add your command handler code here

}

void CDrawView::OnUpdateDrawLine(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
}
```

③ 在实现文件 drawview.cpp 中，增加了消息映射映射：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
//{{AFX_MSG_MAP(CDrawView)
.....//省略以上代码
```

```
ON_COMMAND(ID_DRAW_LINE, OnDrawLine)
ON_UPDATE_COMMAND_UI(ID_DRAW_LINE, OnUpdateDrawLine)
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

ClassWizard 对于命令消息处理函数的识别标识,与窗口消息处理函数一样,在此不再多述。

(3) 控制消息处理函数

控制消息处理函数是对应于 VC++ 中各种控制消息的处理函数,多数存在于基于对话框的类中。对话框中存在着用户定义的很多控制(如编辑框、列表框、复合框、滚动条等),这些控制在建立或被操作时会产生各种不同的控制消息。ClassWizard 能够对这些控制消息的处理函数实行管理。ClassWizard 管理控制消息处理函数的操作方法与管理命令消息时是相同的。

2.5.5 ClassWizard 管理重载虚函数

MFC 框架程序中具有很多标准操作功能。例如,运行应用程序 Draw"文件"菜单下的"打开..."菜单项时,就会弹出文件选择对话框选择需要打开的文档名称,然后进行文档的打开工作。以上这个操作过程在 MFC 中实现,它是由一系列函数组成的。为了让程序开发者能够对这个过程进行修改和控制,在以上这个操作过程中,应用了很多虚函数,在应用程序的当前类中,可以对基类中的这些虚函数进行重载,通过多态性而使 MFC 实现的标准操作过程执行当前类中的虚函数,从而达到控制和修改 MFC 中已经实现的标准操作过程的目的。MFC ClassWizard 可以用来管理这些可以被重载的虚函数。

(1) 重载虚函数

例如,要在类 CDrawDoc 中重载一个虚函数 OnOpenDocument。这个虚函数在打开文档操作时被 MFC 程序框架调用,通常是作为 File→Open 命令操作的一部分。默认情况下,这个虚函数用来打开指定的文件,调用 DeleteContents 虚函数清空文档,调用虚函数 Serialize 读取文件的内容。现在我们用 ClassWizard 在当前文档类 CDrawDoc 中重载这个虚函数。

进入图 2-21 所示的 ClassWizard 操作界面,在"Class Name"列表框中选择类 CDrawDoc,在"Object IDs"列表框中选择 CDrawDoc 项,在"Message"列表框中列出的是类 CDrawDoc 能够得到的各种窗口消息和可重载的虚函数。从"Message"列表框中选择虚函数 OnOpenDocument,单击"Add Function"按钮,按"OK"按钮退出对话框,ClassWizard 就会在 CDrawDoc 类中重载了虚函数 OnOpenDocument。这个操作过程对程序代码做了如下修改:

① 在文档类 CDrawDoc 中,增加了函数的重载定义:

```
//{{AFX_VIRTUAL(CDrawDoc)
```

```
public:
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
//{{AFX_VIRTUAL
```

② 在 CDrawDoc 类的实现文件 drawdoc.cpp 中，增加了函数的实现代码：

```
BOOL CDrawDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (! CDocument::OnOpenDocument(lpszPathName))
        return FALSE;
        // TODO: Add your specialized creation code here
    return TRUE;
}
```

这个函数的第一行代码 CDocument::OnOpenDocument(lpszPathName)，调用了基类 CDocument 的 OnOpenDocument 函数，完成与不重载时相同的操作。可以在当前文档类的 OnOpenDocument 函数中加入代码，完成需要的操作功能。如下改动后的 OnOpenDocument 函数，能够在进行打开文档文件操作的同时，打开一个与当前文件名称相同、而后缀不同的文件，作为应用程序 Draw 运行时的临时文件。

```
BOOL CDrawDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (! COleDocument::OnOpenDocument(lpszPathName))
        return FALSE;
    FilePath = lpszPathName;          //得到当前文档的名称
    int n = FilePath.Find('.');       //得到文件名中.的位置
    if(n>0)   //如果当前的文档名称有后缀，去掉后缀
        FileName = FilePath.Left(n);
    else        //如果文档名称中没有后缀
    FileName = FilePath;
    FileName += ".tmp";          //增加一个".tmp"后缀
    //以二进制方式打开一个可读写的文件，文件不存在就新建一个
    Fundo.Open(FileName,CFile::modeCreate|CFile::modeReadWrite|CFile::typeBinary);
    return TRUE;
}
```

函数中用到的变量是在文档类 CDrawDoc 中定义：

```
CFile Fundo;
CString FileName,FilePath;
```

(2) ClassWizard 识别和处理重载虚函数所需要的标识

ClassWizard 能够识别定义在以下两行代码间的虚函数：

```
//{{AFX_VIRTUAL(CDrawDoc)    //ClassWizard 用来识别虚函数定义的开始代码
//}}AFX_VIRTUAL              //ClassWizard 用来识别虚函数定义的结束代码
```

第一行代码扩号内为重载虚函数的类的名称。

2.5.6　ClassWizard 管理映射变量

(1) 映射变量

在对话框中使用各种控制的目的就是为了方便地显示、修改各种数据资料。对话框中包含着各种各样的控制，大多数控制的内容和状态都可以用一定类型的变量来表示。例如，对于编辑框，当用来处理字符型的数据资料时，它的内容可以由 CString 或 Char 类型的变量来表示；当用来编辑一个数值时，其内容可以由 float 型变量来表示。对于滚动条，滚动条的位置可以用 int 变量来代替，检查按钮的状态可以由 BOOL 类型的变量来表示等等。映射变量是这样一个变量，一方面它是一个普通的变量；另一方面，它通过一定的机制与某个控制建立联系。这样，一方面可以将控制内的内容转换成这个变量，另一方面可以用这个变量来初始化某个控制的内容和状态。

用 ClassWizard 可以方便地管理映射变量。

(2) 用 ClassWizard 管理映射变量

下面用一个例子来说明用 ClassWizard 管理映射变量的方法。

建立一个包含控制的对话框

还是以应用程序 Draw 为例进行讨论。首先需要建立一个包含控制的对话框资源。为方便起见，不再建立新的对话框，而只是对系统既有的对话框资源 IDD_ABOUT 进行修改，在原对话框的基础上增加了两个编辑框(IDC_EDIT1 和 IDC_EDIT2)和一个检查控制(IDC_CHECK1)，得到如图 2-22 所示的对话框资源。



图 2-22　修改后的 About Draw 对话框

增加映射变量

进入 ClassWizard 对话框，在对话框顶部选择“Member Variable”标签，在“Class Name”列表框中选择类 CAboutDlg，就进入图 2-23 所示的对话框界面。

对话框中的控制都列在了“Control Ids”列表框中。增加一个映射变量的方法如下：

在“Control Ids”列表框中用鼠标选择要增加映射变量的控制(如 IDC_EDIT1)，按下“Add

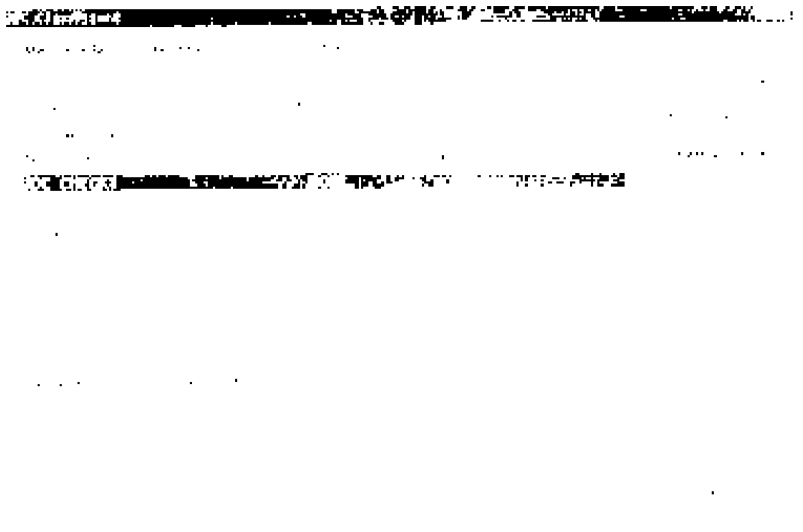


图 2-23　ClassWizard 管理映射变量的界面

Variable..."按钮，就进入到如图 2-24 所示的变量输入对话框。

图 2-24　映射变量输入对话框

在变量输入对话框中，在"Member variable name"编辑框输入变量的名称，在"Category"列表选择框中选择增加的是一个值变量 Value，还是一个控制变量 Control，如果选择的是值变量 Value，还需要选择变量的类型。按"OK"按钮退出，就给所选择的控制增加了一个映射变量。对于一些控制，还可以设定长度（如设定对应 IDC _ EDIT1 的 CString 型的变量最大长度为 10）。

按照以上增加映射变量的方法，为对话框 IDD _ ABOUT 中的几个控制，增加如下映射变量：

IDC _ EDIT1	类型 CString	名称 m _ Name	最大长度 10	
IDC _ EDIT2	类型 float	名称 m _ Value	最小值 0	最大值 100
IDC _ CHECK1	类型 BOOL	名称 m _ IsPioneer		

删除映射变量

在图 2-23 所示的列表框中选择需要删除映射变量的控制，按下“Delete Variable”按钮就会删除控制对应的映射变量。

(3) 分析 ClassWizard 增加映射变量的过程

以上增加映射变量的操作，在应用程序 Draw 的源文件中增加了如下的代码：

① 在实现文件 draw.cpp 中的 CAboutDlg 类中，增加映射变量的定义代码：

```
// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    BOOL   m_IsPioneer;
    CString   m_Name;
    float   m_Value;
    //}}AFX_DATA
```

② 在实现文件 draw.cpp 中 CAboutDlg 类的构造函数中，增加了变量的初始化代码：

```
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    m_IsPioneer = FALSE;
    m_Name = _T("");
    m_Value = 0.0f;
    //}}AFX_DATA_INIT
}
```

③ 在实现文件 draw.cpp 中的函数 DoDataExchange 中，增加了控制和变量间的映射宏：

```
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    DDX_Check(pDX, IDC_CHECK1, m_IsPioneer);
    DDX_Text(pDX, IDC_EDIT1, m_Name);
    DDV_MaxChars(pDX, m_Name, 10);
    DDX_Text(pDX, IDC_EDIT2, m_Value);
    DDV_MinMaxFloat(pDX, m_Value, 0.f, 100.f);
    //}}AFX_DATA_MAP
}
```

重载函数 DoDataExchange 的功能，是管理对话框控制与对话框类的映射变量间的数据交换，并在交换的同时检查数据的有效性。在对话框显示前，对话框类的初始化函数 OnInitDialog 被执行，在 OnInitDialog 函数中调用 UpdateData 函数，通过 UpdateData 函数调用函数 DoDataExchange，把影射变量内容转换成对话框各控制的状态。当用户单击“OK”按钮退出对话框时，

CDialog::OnOK 函数调用了 UpdateData 函数，将对话框控制中的内容转换成与之对应的映射变量，并检查数据的有效性（数据无效则给出提示信息）。函数 UpdateData 只有一个 BOOL 型参数，当参数为 TRUE 时将控制的内容转换成映射变量，而当参数为 FALSE 时将映射变量转换成控制信息。在对话框类中，可通过以不同参数调用 UpdateData 函数，随时完成对话框控制与映射变量间的数据交换。

DDX 宏完成数据交换，DDV 宏完成数据的有效性检查。程序设计者可以定义自己的 DDX 和 DDV 宏。DDX_Text(pDX,IDC_EDIT2,m_Value)代码完成编辑框控制 IDC_EDIT2 与映射变量 m_Value 之间的数据交换。这种数据交换如果不用 DDX 宏来完成，操作是比较繁琐的。下列代码可以完成把编辑框控制 IDC_EDIT2 内的内容赋给变量 m_Value 并进行合法性检查：

```
Char p1[12];
GetDlgItemText(IDC_EDIT2,p1,11);          //将控制 IDC_EDIT2 内的内容读入字符串 p1
m_Value = atof(p1);                       //将字符串转成浮点数值
if(!(m_Value >= 0&&m_Value <= 100)) //如果数值大于 100 或者小于 0
    ;  //给出提示信息
```

同样，下列代码将变量 m_Value 的值转换成字符串，并填入到编辑框控制 IDC_EDIT2 内：

```
char p1[12];
sprintf(p1,"%f",m_Value);                 //将数值转变成字符串
SetDlgItemText(IDC_EDIT2,p1);             //将字符串写入 IDC_EDIT2 控制
```

(4) ClassWizard 识别和处理映射变量所需要的标识

ClassWizard 采用固定的标识来实现对映射变量的管理。

映射变量的定义

映射变量必须定义在以下两行代码间（括号内为映射变量属于的类名）时，才能被 ClassWizard 识别和处理：

```
//{{AFX_DATA(CAboutDlg)          //ClassWizard 用来识别映射变量定义的开始代码
//}}AFX_DATA                     //ClassWizard 用来识别映射变量定义的结束代码
```

映射变量的初始化

映射变量的初始化代码放在如下的两行代码间（在对话框类的构造函数内）时，才能被 ClassWizard 识别和处理：

```
//{{AFX_DATA_INIT(CAboutDlg)     //ClassWizard 用来识别映射变量初始化的开始代码
//}}AFX_DATA_INIT                //ClassWizard 用来识别映射变量初始化的结束代码
```

映射变量与对话框控制间的映射宏

映射变量和控制间的映射宏代码必须放在如下两行代码间（在函数 DoDataChange 函数中）时，才能被 ClassWizard 识别和处理：

```
//{{AFX_DATA_MAP(CAboutDlg)      //ClassWizard 用来识别映射变量的映射宏的开始代码
//}}AFX_DATA_MAP                 //ClassWizard 用来识别映射变量的映射宏的结束代码
```

(5) 连接其他类的映射变量

以上讨论的创建映射变量的操作,都是在当前对话类中创建了一个变量,并建立了这个变量与控制间的映射关系。在很多情况下,可以把对话框中的控制与别的类中已经存在的映射变量建立联系,而非在当前类中定义新的映射变量。当在当前对话框类(如 CAboutDlg 类)中增加了对其他类的外部引用时,在如图 2-24 所示的对话框中增加映射变量时,在"Member variable name"列表编辑框中可以列出外部引用类中的映射变量,并可以选择一个影射变量与当前对话框控制建立映射关系。

在本书的第 16.1 节,将详细讨论用 ClassWizard 在对话框类中建立对话框控制与外部引用类中的映射变量建立映射联系的方法,实现编辑记录集合当前记录各个字段内容的对话框。

2.5.7 管理 Automation 和 ActiveX 事件

启动 ClassWizard,进入到图 2-21 所示的 ClassWizard 对话框界面,选择对话框顶部的"Automaiton"或"ActiveX Event"标签,就进入管理 Automation 或 ActiveX 事件的操作界面。因为在本书利用 VC++开发 GIS 系统的过程中,并不涉及 Automation 和 ActiveX 事件的讨论,所以对于用 ClassWizard 管理 Automation 和 ActiveX 的过程不再进行介绍。在作者所著的《Visual C++高级编程技术——实例开发剖析》一书中(见本书参考书目)18.5.2 小节,对用 ClassWizard 管理自动化(Automation)的方法作了详细的介绍,对使用 ClassWizard 管理 Automation 和 ActiveX 事件,进行 OLE 自动化服务器和 ActiveX 控件设计有详细的讨论。

2.5.8 重新构造开发项目的 ClassWizard 工作信息

在一个程序项目中,支持 ClassWizard 工作的一些信息(如编辑存在的类和增加新类的信息,建立和编辑各种消息处理函数和映射变量等的信息),存储在一个后缀.clw 文件中。在很多情况下(如用手工增加一些类,用手工对类的名称进行了改变等),类的实际内容发生了变化,而.clw 文件中的信息并没有更正,引起 ClassWizard 的显示和工作可能不正确,此时需要重新生成 ClassWizard 工作使用的信息。

以下介绍重新生成 ClassWizard 工作信息的方法:

① 在应用程序的主目录下,你可以找到一个与项目同名的后缀是.clw 的文件(如应用程序 Draw 的 draw.clw 文件),这个文件中存储了 ClassWizard 管理这个程序所使用的一些信息,将这个文件删除。

② 启动 ClassWizard,因为此时已没有了存储 ClassWizard 工作信息的.clw 文件,ClassWizard 会提示是否重新生成所需信息,确认后系统会进入一个如图 2-25 所示的对话框,在这个对话框中选择生成 ClassWizard 信息所需要的文件。ClassWizard 会利用选择的头文件和实现文件来

重新生成 ClassWizard 的工作信息，将应用程序的所有头文件和实现文件增加到图 2-25 所示对话框中的“Files in Project”列表框中，按“OK”按钮，ClassWizard 就会根据头文件和实现文件重新生成 ClassWizard 工作需要的信息。

图 2-25　重建 ClassWizard 工作信息时的文件选择框

2.6　使用 DataView

利用 VC++ 进行数据库应用程序设计时，一般采用 ODBC（或 DAO）利用各种数据源（例如利用 Foxpro、Access、Oracle、SQL Server 等数据库系统的数据库文件）进行程序设计，为了方便数据库程序的设计，就需要在 VC++ 开发环境中能够对数据源中数据表的结构、数据表的内容等进行管理和修改，甚至进行建立数据表、修改数据表的结构、管理 SQL 过程等操作，通过创建一个 Data Project 项目，利用 DataView 在 VC++ 的集成开发环境中，完成以前必须在数据库系统中完成的功能，DataView 对组织数据库应用程序的设计具有非常大的作用。

在 16.2 节，将详细介绍创建 Data Project 项目的方法和步骤，以及利用 DataView 进行数据库的设计和管理的具体操作。

2.7　管理多个程序项目

VC++ 开发平台可以同时管理多个程序项目。例如，在打开矢量图形系统项目 Draw 的情况下，可以在应用程序 Draw 的基础上再创建另外一个应用程序 Draw1，同时对两个程序项目进行管理。

（1）新建增加另外一个程序项目

在 VC++ 开发平台上，打开 Draw 项目，运行“File”菜单下的“New”菜单进入 New 对话框，在

New 对话框中选择“Project”标签，进入到图 2-2 所示的对话框界面，在这个对话框界面中选择“Add To Current Workspace”按钮，在列表中选择“MFC AppWizard(exe)”选项，在“Location”编辑框中输入新建项目的工作目录，在“Project name”编辑框中输入程序项目名称 Draw1，按照 2.1.2 小节生成应用程序 Draw 的步骤，组织生成一个新的程序项目 Draw1。这时，在 VC++ 开发平台上就会有两个程序项目：Draw 和 Draw1。

(2) 增加一个存在的程序项目

如果一个程序项目 Draw1 已经存在，可将它加入到当前的工作平台上。具体操作方法如下：在 VC++ 开发平台中打开应用程序 Draw 项目，运行“Project”菜单下的“Add Project to Workspace...”菜单项弹出文件选择对话框，在文件选择对话框中选择已存在的程序项目 Draw1 的 draw1.dsp 文件，就可以打开这个程序项目，并把它加入到当前的工作平台上。

通过现在的操作，读者应能够理解一个程序项目的.dsw 文件和.dsp 文件的区别：.dsw 文件中存储的是一个工作平台的信息，包括需要打开哪些程序项目的.dsp 文件、各个程序项目间的依赖关系等；而.dsp 文件存储的是一个程序项目中的文件组成、参数配置、文件打开情况等具体信息。

(3) 删除一个程序项目

删除一个程序项目，指的是把一个程序项目从工作平台中去掉，而不是删除一个程序项目的实际内容。在 VC++ 工作区中，选择“FileView”标签，在工作区中选择要删除的程序项目，按“Del”键就能够把这个程序项目从当前工作平台上删除。

(4) 激活一个程序项目

VC++ 的很多操作功能(如 ClassWizard 等)只能应用于工作平台上处于活动状态的程序项目，通过运行“Project”菜单下的“Set Active Project”菜单项，可以选择激活一个工程程序项目。在一些别的操作界面中(如运行“Project”菜单下的“Settings”菜单项得到的 Project Settings 对话框)，列出了包含各个程序项目的列表框，通过在列表中选择程序项目而保证对这个程序项目进行操作。

(5) 设置相互依赖关系

运行“Project”菜单下的“Dependences”菜单项，进入到图 2-26 所示的 Project DepenDences 对话框，在这个对话框中，设置各个程序项目的依赖关系。

依赖关系指的是各个程序项目之间的依存关系。如图 2-26 中所示，程序项目 Draw 依赖于程序项目 DLL1，指的是 Draw 程序项目使用 DLL1 程序项目的结果，所以在编译应用程序 Draw 项目时，如果 Draw1 程序项目没有进行编译，或根据程序项目的依赖性检查，Draw1 程序项目在上次编译后进行了修改，则需要首先对 Draw1 程序项目进行编译。

图 2-26　Project Dependencies 对话框

2.8　使用各设计工具进行编程的几点建议

本章讨论了 VC++ 中进行程序设计时采用的几个工具。这几个工具中，功能最大、最常使用的三个工具为 AppWizard、ClassWizard、AppStudio，在组织数据库系统设计时，DataView 的作用也是十分巨大。AppWizard 是进行 MFC 程序设计必须使用的，ClassWizard 工具对程序设计起到省时省力的作用，应熟练掌握，AppStudio 及相关工具对资源实现可视化管理，极大地方便了组织和构造资源。以下对于使用这些工具提出几点建议。

必须在真正掌握程序的框架构造基础上使用工具

使用这些工具必须是在深刻理解程序的框架构造和这些工具的真正作用基础上。从一定意义上讲，这些工具中的大部分是为熟练的程序设计人员准备的，初学者应先花费大量的精力弄懂 MFC 程序框架的组成结构，而后才能够游刃有余的控制程序。最不可取的是不懂程序的框架结构和工具的本质作用而盲目使用它。具体地说，程序设计者只有在用手工能够熟练地完成任务的前提下，再使用这些能自动完成任务的辅助编程工具（如 ClassWizard、ResourceView、ClassView 等）。使用这些工具的目的，只是为了提高书写代码的效率，提高编制程序的速度，而不是用来代替程序设计。

正确把握程序设计的要点

程序设计者特别是初学者必须认识到，程序设计的最终目的是为了完成具体的任务。所以在应用程序设计中，程序的框架和界面等部分在一个程序中占的比重是较小的（在管理系统中要大一些），而不是程序设计的本质。VC++ 为程序设计提供了很好的框架，通过继承 MFC 类很容易实现一些进行程序设计所用到的普通功能，使程序设计者可以把更多的精力放到程序的本质内容上去，这是十分有益的。

程序设计的能力主要体现在对数据的组织和控制、流程的控制等上面。不同的计算机语言虽各有特点，但几乎所有的计算机语言其设计思想是内在相通的，程序设计能力的高低与采用哪种计算机语言没有必然地联系。现在，计算机语言的书籍多数都集中介绍各种语言的操作功能，缺少对读者进行程序设计实际能力的培养，给初学者造成一些错误的导向，盲目地追求语言版本。似乎只要语言版本越新，自己进行程序设计的能力越强。结果是，一些学习程序

设计的人,计算机语言的基础知识不牢固,对程序设计的基本思想和方法不明晰,对新出现的一些编程工具的本质没领会透彻,所以,在进行具体任务的程序设计时盲目性很大,往往事倍功半,甚至越搞越糊涂。这些都不奇怪,就像会使用电视机的人与会设计、制造、维修电视机的人不同一样,前者只需要知道电视机(包括遥控器)的各种按钮使用功能即可,后者则必须非常熟练地懂得电视技术原理,各种电路,各种集成电路、元件功能,以及整机印制电路板、走线等。用计算机语言进行程序设计,企望具有很强的程序设计能力(简捷、高效),并不在于学了多少语句、函数、对象和方法,而关键是要有明确的设计思路,把具体设计任务与程序的创建、编码、使用辅助编程工具等联系起来,对数据和流程进行组织、控制。要做到这一点,首先要下功夫掌握计算机语言的实质内容。以下各章用实例剖析的方法,在具体开发实现一个基本 GIS 系统的过程中,引导读者进一步掌握 VC++ 的常用开发工具和程序设计方法。

第 3 章　MFC 应用程序框架剖析

在第 2 章 2.1.2 小节用 MFC AppWizard 生成了一个完整的 MFC 应用程序 Draw。相信读者在生成并对它编译执行时，对执行的结果是满意的，对用 MFC 编程一定会产生浓厚的兴趣。你可能会迫不及待的打开产生的源代码文件，寻找实际的执行代码。结果肯定会让你感到大失所望。第一次打开这些文件时，你可能觉得自己在看一些甲骨文的书籍，你连程序是从哪里执行，怎么执行的都无法找到，无法弄明白。不要失望，在本章将对 MFC 应用程序的结构和工作流程进行分析。在以后的各章，将以一个基本 GIS 系统的开发为例，引导读者掌握 MFC 应用程序设计的核心技术。

3.1　MFC 程序基本框架类和文件组成

3.1.1　基本程序框架类

通过对应用程序 Draw 进行分析，可知普通的 MFC 应用程序包含着以下四个主要的类：文档类、视图类、主框架窗口类、应用类。

程序的主要任务在这四个类中分配，AppWizard 为每个类产生了各自的源文件。C++ 类在文件组织上一般分为两部分：头(包含)文件(.h)和实现文件(.cpp)。头文件是.h 文件，一般用来存放类、结构等的定义代码，通过 include 宏插入到实现文件中。实现文件是通常的.cpp文件，用来存放类的成员函数的实际代码。

(1) 文档类

应用程序 Draw 的文档类名称是 CDrawDoc，这是 AppWizard 根据项目名称默认所取的名称，它是由 MFC 的基类 CDocument 派生的。CDrawDoc 类的头文件是 drawdoc.h，实现文件是 drawdoc.cpp。头文件中存放的是类的定义代码，实现文件中存放的是类的各个成员函数的实现代码。文档类负责存放程序数据并读取磁盘文件数据，或将磁盘文件数据写入磁盘文件中。在建立一个应用程序项目时，在文档类中对应用程序所需要的数据资料进行管理。

(2) 视图类

应用程序 Draw 中视图类的名称是 CDrawView，它是由 MFC 的 CView 类派生的。CDrawView 类的头文件是 drawview.h，实现文件是 drawview.cpp。视图类负责显示文档类中的数据，可以显示在屏幕上，也可以输出到打印机或其他设备上。视图类也负责处理用户的输入。在一个矢量图形系统中，在屏幕上显示图形、在打印机上绘制图形的操作功能都是由视图类完成的。

(3) 框架类

在使用 AppWizard 生成应用程序 Draw 时,选择的是一个多文档选项(在图 2-3 中,选择的是 Multiple documents 选项),所以应用程序 Draw 中包含着两个框架类:主框架类 CMainFrame 和子框架类 CChildFrame。主框架类 CMainFrame 是由 MFC 的 CMDIFrameWnd 类派生的,其头文件是 mainfrm.h,实现文件是 mainfrm.cpp。CMainFrame 类提供了管理一个多文档界面(MDI)的主窗口的所有功能及管理窗口中的子窗口,用来显示一个标题、一个菜单条、窗口最大化和最小化键、边框、一个系统菜单、工具条以及状态条等。子框架类 CChildFrame 是由 MFC 的 CMDIChildWnd 类派生的,其头文件是 childfrm.h,实现文件是 childfrm.cpp,子框架类的功能是用来管理子窗口。一个子窗口非常像主窗口,但子窗口仅能在主窗口内显示,而不在 Windows 桌面上,子窗口没有自己的菜单条,但它分享主窗口的菜单,程序自动将框架窗口菜单作为当前子窗口的菜单。

(4) 应用类

应用程序 Draw 的应用类是 CDrawApp,它是由 MFC 的 CWinApp 类派生的。其头文件是 draw.h,实现文件是 draw.cpp。应用类管理程序的总体,它完成不属于任何其他三类的一般工作,例如初始化程序以及进行最后的程序清除工作。每个 MFC 应用程序程序必须正确的生成由 CWinAPP 派生类的一个实例(对象)。

3.1.2 应用程序主要源文件组成

在组成应用程序 Draw 的源文件中,除了五个主要类的源文件外(10 个源文件),还有其他一些文件,对他们简述如下:

draw.rc: 资源定义文件,包含程序资源的定义(菜单、对话框、字串、键盘加速键和图标等)。一般情况下,这个文件是由 AppStudio 进行可视化编辑维护,一般不用手工修改这个文件。

draw.dsw: 工作平台信息文件,存储着一个工作平台的组成情况(如包含哪些程序项目、程序项目的依赖关系等),用来存储和打开一个工作平台。

draw.clw: ClassWizard 信息存储文件,存储 ClassWizard 编辑存在的类和增加新类的信息,文件也包含了 ClassWizard 建立和编辑各种消息处理函数和映射变量等的信息。

draw.dsp: 程序项目工程文件,存储着一个应用程序项目所包含的具体信息,在打开项目时用。

makehelp.ybat: 批处理文件,用来建立应用程序的帮助文件 hlp \ draw.hlp。

hlp \ draw.hpj: 被帮助文件编译器用来建立系统的帮助文件时使用的工程文件。

hlp \ *.bmp: 被用来作为 MFC 标准命令帮助主题的位图文件。

hlp \ *.rtf: 帮助主题文件,包含着 MFC 标准命令的帮助主题。

res \ draw.rc2: 资源定义文件,包含着不用 AppStudio 编辑的程序资源定义。当手工(而不是 AppStudio)定义程序资源,可以将资源定义在这个文件中。这个文件被插入(#include)在文件 draw.rc 中。

res \ draw.ico： 应用程序的图标文件。最初，这个文件包含标准的“AFX”图标。可以用AppStudio对图标进行修改。

res \ toolbar.bmp： 工具条的位图文件。

stdafx.cpp和stdafx.h： 预编译文件，用于生成预编译信息。

resource.h： 资源ID号定义文件，这个文件与draw.rc相对应，包含程序资源中ID号的常量定义。一般也是由AppStudio自动进行维护的。

draw.def： 它提供Microsoft LINK程序进行连接时，用于准备可执行的应用程序文件的有关的信息。一般不必编辑这个文件，因为它包含适合于大多数MFC Windows程序的值。

readme.txt： 包含对所有该程序的源文件的解释信息。

3.2 MFC应用程序中各个框架类的作用

以上讨论了应用程序Draw中包含的四个主要类及其文件组成。本节进一步分析一下这个框架的组成及各个类的主要功能。学习VC++编程的人，特别是初学者，一定感觉到对程序的框架很难理解，即使有一定的理解，使用起来也不能得心应手。作者根据实际的经验，举一个可能并非完全恰当的比喻来分析一下程序的框架，使读者对框架产生一个感性的认识，再随着学习书中其他各章节，通过介绍一些开发实例，期望读者在VC++编程技术上有比较大的进步。

假定有一些文稿，一块黑板，一些工具盒，由这些组成了一个工作环境或平台。程序框架中的四个主要的类所完成的工作就可以如下理解：

① 文档类创建的对象负责文稿的管理工作，每个对象负责一份文稿的管理。在单文档程序框架(SDI)中只有一份文稿，也就只需创建一个文档对象负责文稿的管理工作；在多文档程序框架(MDI)中可以有多份不同的文稿，那么就需要文档类创建多个文档对象来管理多份文档，一个对象管理一份文档资料。文稿中内容的增加、删除、修改、归档保存等管理和维护操作，都是由文档对象来完成的。

② 黑板用来显示文档的内容，框架类的对象负责黑板的管理工作。一个主框架类的对象负责在黑板上划出一个区域来(主框架)，在这个区域内，主框架类对象负责安排摆放如粉笔盒的位置等(菜单、工具条、状态条等的布置)，同时，主框架类对象又能够在自己管辖范围内划出一块区域(窗口)交给一个子框架类对象来管理，这个子框架类对象把管理的窗口中划出一部分交给一个视图对象来使用(客户区)，视图对象可以在这个区域内书写内容。在单文档框架中，主框架类管理的区域内只能分配一个区域(窗口)，此时，只能有一个客户区供一个视图对象来使用。而在多文档程序框架中，主框架类对象可以将黑板分成多个区域(窗口)，每个区域交给一个子框架类对象来管理。子框架类对象将这个区域的一部分(客户区)交给一个视图对象来使用。主框架类根据需求来维护和管理这些区域，像区域的创建、删除、改变大小等工作都是由主框架类来完成的。Windows下的窗口与黑板中的区域不同的是，窗口是可以任意重叠的。

③ 视图类的作用是将文稿中的内容进行显示，在黑板中创建的一个子窗口内的客户区被一个视图对象所使用。同时，这个视图类对象在创建时已经被规定了是用来显示哪份文稿的。它的任务是将这份文稿中的内容显示在黑板中其拥有的客户区内。在单文档程序框架下，只有一份文档，即只有一个文档类对象，也只有一个视图类对象，这个视图类对象负责把文档的

内容显示到黑板上，如何显示(用文字还是用图表现，用大字还是小字，用红粉笔还是白粉笔)是由视图类对象决定的。而在多文档资料框架(MDI)下，情况复杂多了，这时可以有多份文稿，黑板中可以创建有多个子窗口区域；一份文稿，可以创建多个视图类对象来显示。如，有两份文档，对其中的一份创建了五个视图类对象，对于另外一个文档创建了三个视图类对象，这时，就有八个视图类对象，创建了八个子框架类对象，管理黑板中创建的八个子窗口，五个视图类对象在显示第一份文档的内容，三个视图类对象在显示另一份文档的内容。一份文稿由多于一个视图对象在不同窗口的区域(客户区)上进行显示，就是所谓的多视图有多于一个的文稿在不同的区域上被显示就是多文档。

④ 应用程序类创建一个对象(且只创建一个对象)，负责建立并启动这个工作环境，建立起文档、视图、框架对象之间的相互联系。如，当建立或打开一个文档时，文档类会创建一个文档类对象，框架类会创建一个子框架类对象，同时视图类会创建一个视图类，并建立了文档、子框架及视图类对象间的相互联系。这些工作是由应用类对象来完成的。

对于一个多文档 MFC 应用程序，在程序运行过程中，只有一个应用程序类对象，一个主框架类对象，可以有多个子框架类对象，多个文档类对象和多个视图类对象。子框架类对象的数目与视图类对象相同，文档类对象可以比视图、子框架类对象少，因为一个文档类对象可以有多个视图类对象。

在应用程序 Draw 中，已经建立起了窗口、视图、文档之间的关系，不用程序设计者再去建立这个框架。运行“文件”菜单下的“新建”菜单项，就会在主框架窗口内产生一个子窗口，子窗口内的区域(客户区)属于新建的视图类对象，这个子窗口对应着新建的空文档。运行“文件”菜单下的“打开...”菜单项，选择打开一个文档时，也会在主框架窗口内产生一个子窗口，这个子窗口对应的文档是刚刚打开的文档。在应用程序 Draw 中有打开文档的情况下，运行“窗口”菜单下的“新建”菜单项，也会在主框架窗口中产生子窗口，这个窗口中显示的是当前文档的另外一个视图。

以上讨论了程序框架中文档、视图、窗口间的关系和各主要类的作用。在实际的程序设计中，用的最多的是文档、视图类。在有些情况中，文档类和视图类的一些作用区分是不明显的，特别是在单文档编制程序的情况下，因为在单文档情况下，只有一个文档，也只有一个视图，换句话说，文档类和视图类都是被实例化了一次，即都只创建了一个对象，所以有些数据和函数放在文档类和视图类中是一样的。而在多文档程序设计中就不一样了，举一个简单的例子，来说明将变量定义在文档类和视图类的区别：如果有一个变量来控制文档的显示比例，这个变量定义在文档类中时，对于这个文档类对象的多个视图在显示比例上是一样的，一个视图比例变化时，其他属于这个文档类对象的视图比例也发生变化。如果这个变量在视图类中定义，多个视图在显示比例上就可以不一样。所以，只有真正理解了视图/文档体系结构，才能随心所欲的来组织程序。

3.3 MFC 应用程序的执行步骤分析

如果你熟悉传统的程序设计，面对由 AppWizard 生成的 MFC 应用程序 Draw，读者肯定会问，Draw 从哪里开始执行的？从哪里吸收控制？在本节中，对这个问题进行简单的介绍。以下是一般 MFC 应用程序运行的主要步骤：

① 调用 CWinApp 类构造函数。
② 程序入口函数 WinMain 接收控制。
③ WinMain 调用程序的 InitInstance 函数。
④ WinMian 进入消息循环。
⑤ WinMain 退出，程序终止。

MFC 应用程序必须定义有且只有一个应用类对象。在实现文件 draw.cpp 中可以找到一个全局定义的应用程序类的对象：

```
CDrawApp theApp;
```

以下介绍对应用程序执行的这几个主要步骤。

(1) 调用 CWinApp 类构造函数

由于全局定义应用类对象 theApp，在程序入口函数 WinMain 接收控制前，先调用应用类的构造函数。在实现文件 draw.cpp 中可以找到应用类 CDrawApp 的构造函数：

```
CDrawApp::CDrawApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}
```

在调用这个构造函数时，同时调用基类 CWinApp 的构造函数完成构造任务。将应用类对象 theApp 定义成全局的，是为了在 MFC 中能够对 CWinApp 的成员函数进行调用。

(2) WinMain 接受控制

定义全局应用类对象 theApp 后，程序入口函数 WinMain 接受控制。这个函数已经在 MFC 中做了定义，在建立执行文件时能够链接到应用程序 Draw。WinMain 完成的任务很多，以下介绍其中重要的一些任务。

(3) 调用函数 InitInstance 进行初始化

在 WinMain 得到控制后不久，就调用执行应用类 CDrawApp 类的成员函数 InitInstance 来进行程序的初始化。在实现文件 draw.cpp 中可以找到 InitInstance 函数的实现代码：

```
BOOL CDrawApp::InitInstance()
{
    AfxEnableControlContainer();

#ifdef _AFXDLL                          //当把 MFC 作为一个动态链接库使用时
    Enable3dControls();                 //使 Windows 显示具有三维外观的控件
#else                                   //当把 MFC 作为一个静态库使用时
    Enable3dControlsStatic();
#endif
```

```
    //以下是程序注册的关键字,可以改为用户需要的信息
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));
    LoadStdProfileSettings();                    // 装入标准初始化文件选择
    CMultiDocTemplate* pDocTemplate;             //定义一个多文档模板对象的指针
    //构造一个文档资料模板对象,并把指针保存在以上定义的指针中
    pDocTemplate = new CMultiDocTemplate(
        IDR_DRAWTYPE,
        RUNTIME_CLASS(CDrawDoc),                 //注册文档类
        RUNTIME_CLASS(CChildFrame),              //注册多文档子框架窗口类
        RUNTIME_CLASS(CDrawView));               //注册视图类
    AddDocTemplate(pDocTemplate);                //增加一个文档模板

    // 创建一个主框架窗口
    CMainFrame* pMainFrame = new CMainFrame;
    if (! pMainFrame->LoadFrame(IDR_MAINFRAME))  //装入主框架
        return FALSE;
    m_pMainWnd = pMainFrame;

    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);                   //取得运行程序时的命令行
    // Dispatch commands specified on the command line
    if (! ProcessShellCommand(cmdInfo))          //处理命令行
        return FALSE;
    // The main window has been initialized, so show and update it.
    pMainFrame->ShowWindow(m_nCmdShow);  //显示主框架窗口
    pMainFrame->UpdateWindow();
    return TRUE;
}
```

这个函数所完成的最主要任务如下:

① 定义并初始化了一个多文档模板对象 pDocTemplate,构造这个对象时用了四个参数。第一个参数是显示和管理文档的程序资源 IDR_DRAWTYPE,就是应用程序 Draw 有打开文档时,显示在主框架窗口的菜单、图标等资源(可以打开 Draw.rc,在里面找到 ID 号是 IDR_DRAWTYPE 的资源)。另外三个参数是文档类、子框架类和视图类的信息,通过 MFC 宏 RUNTIME_CLASS 取得。MFC 应用程序通过这些信息,在打开或新建一个新文档时,分别生成一个文档类、子框架类和视图类的对象,并用第一个参数指定的资源进行管理。

② 完成多文档模板的定义后,程序产生了一个主框架窗口,创建主框架窗口时采用的是 IDR_MAINFRAME 资源。应用程序 Draw 运行时,当所有的文档被关闭时显示在屏幕上的资源就是 IDR_MAINFRAME 资源。在有打开的文档的情况下,这个资源被注册到文档模板类对象中的 IDR_DRAWFRAME 资源所代替。

③ 创建主框架窗口后,程序将处理命令行信息。命令行信息是跟随在命令后的参数。如

可以用如下的语句启动应用程序 Draw:

```
Draw draw1.dat      //启动应用程序 Draw,并在启动后自动打开 draw1.dat 文件
```

ParseCommandLine 函数用来取得命令行,函数 ProcessShellCommand 处理命令行的信息。命令行参数存在,此函数会取出参数调用 CWinApp 的 OnOpenDocument 打开参数指定的文档;命令行参数不存在,则调用 CWinApp 的 OnOpenDocument 函数创建新的文档。不论是打开文档还是新建文档,CWinApp 都是通过调用多文档模板来构造一个文档类对象、一个子框架类对象和一个视图类对象,并调用其相关的资源产生运行界面。

④ 最后,程序显示并更新主框架窗口。

(4) WinMain 处理消息

完成初始化任务后,WinMain 进入消息循环。通过 Run()函数把消息从消息队列中取出,并发送到别的窗口。大多数命令消息被发送到应用程序主窗口,其中的函数 WindowProc()得到消息,再根据消息的类别送到适当的对象,激活并运行对象中的消息处理函数。

(5) WinMain 退出,程序终止

在应用程序 Draw 中,运行"文件"菜单下的"退出"菜单项,或运行系统菜单的"关闭"菜单项,或单击主框间窗口中的 Close 框时,应用程序删除程序窗口,并调用 Win32 API 函数 ::PostQuit Message,使消息循环退出。然后 WinMain 函数返回,程序终止。

3.4 MFC 应用程序设计结构体系剖析

运行应用程序 Draw 时。从运行情况来看,应用程序 Draw 包含很多标准菜单(如"文件"、"编辑"、"窗口"、"显示"等菜单下的各菜单项),运行标准菜单可以执行一系列的操作。初学者可能会急着打开程序的源代码文件,寻找对应于每个菜单的程序执行代码,跟踪一下程序是怎么运行的。但把代码看了个遍也找不到菜单项是如何被执行的,更谈不上控制这些菜单项的执行情况了。这到底是怎么回事呢?

3.4.1 标准菜单的执行分析

在 2.5.1 小节曾经讨论过,MFC 定义了一些标准的命令消息和窗口消息,如 ID_FILE_NEW、ID_FILE_OPEN 等,应用程序 Draw 各个标准菜单项对应的都是这些已经在 MFC 中定义的消息,同时,MFC 为这些消息都定义了消息处理函数并建立了消息映射机制。当选中并执行这些菜单项的时候,MFC 中的消息处理函数被执行。读者有兴趣;可以从 VC++ 主目录下的 mfc \ src 子目录下找到各种已经定义好的函数。对于每一个菜单命令消息,在 MFC 中实现的消息处理函数中调用了一系列的函数,经过一系列的步骤来完成这一个操作。如对于 ID_FILE_OPEN 消息(即按中"文件"菜单下的"打开..."菜单项时),在 CWinApp 类中定义了消息处理函数 OnFileOpen,OnFileOpen 函数执行时,根据在应用类 CDrawApp 中的 InitInstance 函数中注册的多文档程序模板,分别去创建一个文档类、子框架类和视图类对象。没有必要去分析操

作的全过程，以下这个函数是在 OnFileOpen 函数操作过程中调用的一个属于多文档模板类的成员函数，它完成创建一个文档类对象和属于这个文档类对象的框架类对象的功能，以下是这个函数的实现代码：

```
CDocument * CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName, BOOL bMakeVisible)
{
    CDocument * pDocument = CreateNewDocument();//创建新文档对象，pDocument 指向这个对象
    if (pDocument == NULL)                          //如果创建文档类对象失败，函数非正常退出
    {
        TRACE0("CDocTemplate::CreateNewDocument returned NULL.\n");
        AfxMessageBox(AFX_IDP_FAILED_TO_CREATE_DOC);
        return NULL;
    }
    ASSERT_VALID(pDocument);
    BOOL bAutoDelete = pDocument->m_bAutoDelete;
    pDocument->m_bAutoDelete = FALSE; // don't destroy if something goes wrong
    CFrameWnd * pFrame = CreateNewFrame(pDocument, NULL); //创建一个子框架类对象
    pDocument->m_bAutoDelete = bAutoDelete;
    if (pFrame == NULL)          //如果创建子框架类对象失败，函数非正常退出
    {
        AfxMessageBox(AFX_IDP_FAILED_TO_CREATE_DOC);
        delete pDocument; // explicit delete on error
        return NULL;
    }
    ASSERT_VALID(pFrame);
    if (lpszPathName == NULL)     //如果文档没有名称，即是在执行"新建"菜单时调用此函数
    {
        // create a new document - with default document name
        SetDefaultTitle(pDocument);  //给定一个默认名称
        // avoid creating temporary compound file when starting up invisible
        if (! bMakeVisible)
            pDocument->m_bEmbedded = TRUE;
        if (! pDocument->OnNewDocument())
        {
            // user has be alerted to what failed in OnNewDocument
            TRACE0("CDocument::OnNewDocument returned FALSE.\n");
            pFrame->DestroyWindow();
            return NULL;
        }
        m_nUntitledCount++;       //给计数器加 1，通过此计数器产生无名的文档名称
    }
    else          //如果文档有名称，即运行"打开..."菜单项时运行的本函数
    {
```

```
        // open an existing document
        CWaitCursor wait;
        if (! pDocument - > OnOpenDocument(lpszPathName))//调用产生文档类的函数
        {
            // user has be alerted to what failed in OnOpenDocument
            TRACE0("CDocument::OnOpenDocument returned FALSE.\n");
            pFrame - > DestroyWindow();
            return NULL;
        }
    # ifdef _ MAC
        // if the document is dirty, we must have opened a stationery pad
        // - don't change the pathname because we want to treat the document
        // as untitled
        if (! pDocument - > IsModified())
    # endif
        pDocument - > SetPathName(lpszPathName);
    }
    InitialUpdateFrame(pFrame, pDocument, bMakeVisible);
    return pDocument;
}
```

3.4.2　修改标准菜单执行功能

在 3.4.1 小节中，讨论了 MFC 应用程序标准菜单的执行情况。在 MFC 中已经实现的针对标准菜单的操作功能，不可能满足实际应用程序设计的要求。但是，程序设计者不可能去修改 MFC 中实现标准操作功能的代码，来实现实际的操作功能。所以，如何在应用程序设计中改变和控制这些标准菜单的执行功能，是一个关键的问题所在。

从上小节列出的函数 OpenDocumentFile 中可以看到，在完成打开一个文档的操作功能时，MFC 实现的标准操作通过当前文档类对象指针调用了成员函数 OpenDocument(见黑体部分代码 pDocument - > OnOpenDocument(lpszPathName))。你留心一下的话，在应用程序 Draw 的文档类 CDrawDoc 中找不到这个函数。这时此句代码执行的是基类 CDocument 中的 OpenDocument 函数。这个函数是虚函数，如果当前类没有这个函数，就会执行基类中的虚函数。而如果在当前类中重载了这个函数，代码就转而执行当前类的这个虚函数。这样，只要在当前类中重载和改变各个虚函数，就达到修改标准菜单执行功能的目的。

MFC 程序框架正是基于此实现的。由 AppWizard 产生的框架程序中，其标准菜单执行的操作过程是在 MFC 中定义实现时，在多个环节中调用了一些虚函数来完成操作。在代码不做改动的情况下，这些调用都是通过定义在 MFC 的基类中的虚函数完成的。如果在当前程序类中对这些虚函数进行重载，就使这些环节中程序执行的是当前类的函数，从而达到了改变和控制标准菜单操作任务的目的，实现自己特定的功能。此项功能的实现是通过虚函数和多态性来实现的。

3.4.3 MFC应用程序多态性的实现

(1) 实现多态性的前提条件

通过虚函数实现多态性的关键,是将调用函数时的对象指针指向实际对象。例如,在3.4.1小节列出的OnOpenDocumnet虚函数,实现多态性的关键问题是指针pDocment指向当前应用程序的文档类对象。如果pDocument指针指向的是一个CDocument类对象(可以用以下的代码来描述这个过程):

```
CDocument m_Document1;
CDocument * pDocument = &m_Document1;
```

即使在当前应用程序Draw的文档类CDrawDoc中重载虚函数OnOpenDocument,通过pDocument指针调用虚函数OnOpenDocument时,程序流程也不会去执行CDrawDoc类中的重载虚函数OnOpenDocument。只有把对象指针pDocument指向当前应用程序文档类对象时(用以下代码来描述这个过程):

```
CDrawDoc m_Document1;
CDocument * pDocument = &m_Document1;
```

通过pDocument指针去调用虚函数OnOpenDocument时:

```
pDocument -> OnOpenDocument(lpszPathName);
```

如果在CDrawDoc类中重载了OnOpenDocument函数,以上代码调用就会执行CDrawDoc类中的虚函数OnOpenDocument,从而达到了程序执行的多态性。

(2) MFC应用程序实现多态性的过程

现在来讨论MFC应用程序是如何实现多态性的。在MFC类中实现了大量完成标准功能的标准操作,在这些操作过程中调用了大量当前应用程序基类(如CDocument、CView、CFrameWnd等)中的虚函数。MFC应用程序的启动和设置部分是在MFC类中实现的(这就是不能从当前应用程序中找到程序启动部分的原因),在执行过程中。通过定义的全局CDrawApp类对象theApp,调用应用类CDrawApp的InitInstance函数将当前应用程序的文档类、视图类、框架类都注册到了多文档模板对象中。通过这个多文档模板对象在新建和打开文档

操作时,创建的是当前应用程序类的对象。而在MFC类中实现的标准操作过程中,利用指向当前应用程序类对象的基类指针,调用了基类中定义的虚函数去实现操作功能(如在Draw应用程序打开文档操作时,实际上是在MFC中创建了一个CDrawDoc类对象,然后将CDocument类指针指向这个对象)。这样,就满足了实现多态性操作的条件,当在当前应用程序的框架类(如CDrawDoc、CDrawView、CMainFrame等)重载其基类的虚函数时,MFC应用程序中实现的标准功能就会去执行这些重载的虚函数,而达到改变MFC程序设计标准功能的目的。

在MFC应用程序新建或打开文档操作时,可通俗的理解为在MFC中执行了如下操作:

```
CDrawDoc m_Drawdoc1;
CDrawView m_Drawview1;
CChildFrame m_Childframe1;
```

以上代码只作为理解用,对象的名称也是为了加深读者的理解而给定的。MFC应用程序建立了各个对象之间的关系,并将各个基类指针指向了这些类对象:

```
pDocument = &m_DrawDoc1;
pView = &m_Drawview1;
pFrame = &m_Childframe1;
```

pDocument、pView、pFrame是在MFC中定义的基类指针:

```
CDocument * pDocument;
CView * pView;
CFrameWnd * pFrame;
```

这些指针被用来在各个MFC中的消息处理函数中,调用基类的虚函数完成各个标准操作功能。

在应用程序进行打开视图的操作时(运行应用程序Draw"窗口"菜单下的"新建"菜单项时),MFC程序框架执行了如下操作:

```
CDrawView m_Drawview1;
CChildFrame m_Childframe1;
```

在MFC中创建了一个视图类和子框架类对象,而没有新建文档类对象,在建立各个对象间的关系时,采用的是已经存在的处于活动状态的文档类对象。将各个基类指针指向了这些新建的类对象:

```
pView = &m_Drawview1;
pFrame = &m_Childframe1;
```

3.4.4 MFC应用程序中各个框架类对象的相互调用关系

从3.4.3小节的讨论中可以知道,在多文档MFC应用程序执行过程中,创建了多于一个的文档类、视图类、子框架类对象和一个主框架类、应用类对象。这些对象之间是通过一定的方式联系在一起的,在应用程序设计中,时常需要通过这些对象之间的关系来实现在文档、视图、框架类对象及应用类对象间相互引用和调用。本小节,将对这些对象间的调用方法进行简述。

(1) 视图类对文档类的调用

视图类对象的作用是在客户区显示文档类对象的内容,两者之间的调用最为常用。在视图类对象中,通过以下函数得到文档类对象的指针。在视图类 CDrawView 中可以找到一个函数:

```
public:
    CDrawDoc * GetDocument();
```

在实现文件 drawview.cpp 中有这个函数的实现代码:

```
CDrawDoc * CDrawView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CDrawDoc)));
    return (CDrawDoc *)m_pDocument;
}
```

GetDocument 函数的作用就是得到当前视图类对象属于的文档类对象的指针。在视图类 CDrawView 的成员函数中,可以通过以下代码调用 GetDocument 函数得到当前文档类对象指针:

```
CDrawDoc * pDoc = GetDocument();
ASSERT_VALID(pDoc);
```

得到指向当前文档类对象的指针 pDoc 后,同使用普通 C++ 类的指针一样可以调用文档类的成员变量和成员函数(只能调用公有型的)。

(2) 在框架类中获得当前的文档和视图类对象指针

在主框架类 CMainFrame 和子框架类 CChildFrame 中,都可以利用基类的函数,得到指向当前活动的文档类对象和视图类对象的指针。下面代码可以获得指向当前活动的文档类对象和视图类对象的指针:

```
CDrawDoc * pDoc = (CDrawDoc *)GetActiveDocument();
CDrawView * pView = (CDrawView *)GetActiveView();
```

(3) 获得应用程序类对象的指针

在 MFC 应用程序运行过程中,自始至终存在一个应用类对象,可以在 MFC 派生类中或非 MFC 派生类中,通过全局 API 函数 AfxGetApp 得到应用类对象的指针。如可以在应用程序 Draw 中的任何地方,通过以下代码获得指向应用类对象的指针:

```
CDrawApp * pWinApp = (CDrawApp *)AfxGetApp();
```

(4) 从应用类对象中获得主框架类对象的指针

通过 MFC 应用程序的应用类对象,可以获得指向当前系统的主框架类对象的指针,在应用类 CWinApp 中有一个数据成员 m_pMainWnd,这个数据成员保存了指向应用程序主框架类

对象的指针(在2.3节讨论的CDrawApp类的InitInstance函数中有此数据成员的初始化代码)。通过此数据成员可以获得指向主框架类对象的指针。以下代码可获得指向主框架对象的指针:

```
CMainFrame * pFrame = (CMainFrame * )(AfxGetApp() - > m_pMainWnd);
```

在视图类、文档类,以及非MFC派生类的成员函数(如全局函数)中,通过API函数AfxGetApp()获得指向应用类对象的指针,再通过应用类中的成员变量m_pMainWnd,得到指向主框架类对象的指针。

通过API函数AfxGetApp()得到指向应用类对象的指针,再通过应用类对象得到指向主框架类对象的指针,通过主框架类对象又可以获得指向当前活动的文档和视图类对象的指针。这种调用可以保证在各种情况下,获得指向MFC应用程序中各主要框架类对象的指针,通过指针使用各框架类对象的成员(数据或函数)。

基本矢量图形系统

本篇将在用 MFC AppWizard 产生的应用程序 Draw 的基础上，去具体组织一个基本矢量图形系统的开发。将讨论与矢量图形系统设计有关的类及其主要操作。从矢量图形系统的类组织、文档管理、视图组织、图形绘制、图形操作、图形选中、图形删除、文档逆操作、图形保存等各个方面，实例开发实现一个用于 GIS 系统的基本矢量图形系统。

通过本篇的学习，能够从理论和技术上掌握基本矢量图形系统的设计方法，掌握 MFC 应用程序的设计技术。

第 4 章 与图形操作有关的 MFC 类及操作
第 5 章 实现基本矢量图形系统的文档和视图
第 6 章 鼠标交互绘图
第 7 章 矢量图形系统的操作功能
第 8 章 图形的删除和恢复
第 9 章 图形的存取功能
第 10 章 矢量图形系统的其他功能

第二篇

第 4 章　与图形操作有关的 MFC 类及操作

在 MFC 中，有很多与图形操作有关的类，在本章中，将对主要与绘图有关的 MFC 类的功能及相关操作进行介绍。

4.1　绘图类

绘图就要用到 MFC 的类 CDC、CClientDC、CPaintDC、CWindowDC 等，其中 CDC 类是基础，另外几个类是为了特殊应用由其派生出来的。用 VC++ 编程的读者对怎么使用这几个类往往容易模糊和混淆。这几个类所包含的具体内容，读者可阅读有关类库说明的用户手册。作者感到，各种情况下该使用哪个类，怎样引用这个类，这些都是最令读者头疼的问题。因此，本节就这些问题进行叙述，这对程序设计是非常有用的。

4.1.1　绘图类简介

(1) CDC 类

CDC 类是 CObject 类的一个派生类，CDC 类是所有绘图类的基类。CDC 类定义了一个设备描述对象，CDC 类提供了对设备描述对象进行操作的成员函数，以及对与窗口客户区有关的显示区进行操作的成员函数。通过 CDC 类及其派生类定义的对象，可以利用 CDC 类的所有成员函数完成图形的绘制工作。CDC 类提供的成员函数，可以用于对设备描述对象进行的操作、绘图工具的使用、图形设备界面(GDI)的选择，以及颜色和调色板的操作。它提供的成员函数还用于取得和设置绘图属性、映像方式、视口和窗口范围的操作、坐标的转换、区域的使用、剪取、画线，以及绘制图形、文字等操作。

总之，CDC 类中包含了与图形绘制有关的大部分操作，为了能适应 Windows 编程时的具体需要，又派生出了以下几个派生类。

(2) CPaintDC 类

CPaintDC 类是 CDC 类的一个派生类，它用于模拟响应 Windows 消息 WM _ PAINT 的设备环境，在这个类中可以使用 CDC 中的所有函数。CPaintDC 类一般用在函数 OnPaint 内，以响应 WM _ PAINT 消息。WM _ PAINT 消息是当窗口的某个区域需要重画时激发的窗口消息。一个用 CPaintDC 类定义的对象是按以下机制调用的：Windows 的消息循环机制一直在采集各种消息，当接到 WM _ PAINT 消息时，就自动调用消息处理函数 OnPaint，在 OnPaint 函数，定义了 CPaintDC 类对象，通过这个类对象使用 CDC 类的成员函数，完成视图客户区中的图形绘制操作。

(3) CClientDC 类

CClientDC 类也是 CDC 类的一个派生类,它为非响应 Windows 消息 WM _ PAINT 的成员函数提供了一个设备环境。

(4) CWindowDC 类

CWindowDC 类也是 CDC 类的一个派生类,它提供了一个设备环境,以便成员函数在包括非客户区在内的整个屏幕区域内绘图。

4.1.2 几个绘图类的主要差别

本小节对几个绘图类的主要区别进行讨论。

(1) CWindowDC 类与 CPaintDC、CClientDC 类的区别

CWindowDC 类与 CPaintDC、CClientDC 类区别的一个方面是:用 CPaintDC 类和 CClientDC 类的对象绘制图形时,绘制区只能是客户区,而不能在非客户区,而 CWindowDC 可以在非客户区进行图形绘制。客户区和非客户区的概念可以通过图 4-1 来理解。当画一条起始坐标都在屏幕外且通过屏幕的直线时,用 CPaintDC 和 CClientDC 类对象绘制时,只有在客户区的直线能够绘制出来;而用 CWindowDC 类对象绘制时,直线可以画在非客户区。CWindowDC 类一般在框架类(CMainFrame 类)中使用,在视图类(如 CDrawView 类)中使用 CWindowDC 类时,由于视图类只能管理客户区,所以并不能在非客户区进行绘制。

CWindowDC 类与 CPaintDC 和 CClientDC 类区别的另一个方面是:在 CWindowDC 绘图类下,坐标系是建立在整个屏幕上的,在像素坐标方式下,坐标原点在屏幕的左上角;而在 CPaintDC 和 CClientDC 绘图类下,坐标系是建立在客户区上的,在像素坐标方式下,坐标原点在客户区的左上角。图 4-1 中的两条线是分别用 CWindowDC 类和 CClientDC 类绘制的直线,直线的起点为(100,0),终点为(400,600),绘制时,框架窗口在屏幕上处于最大状态。很明显,CWindowDC 类可以在非客户区中进行图形绘制。

(2) CPaintDC 类与 CClientDC 类的区别

CPaintDC 类与 CClientDC 类对象都是在窗口的客户区内绘制图形,但两者在绘制机制上有着本质的区别。CPaintDC 类对象应用在 OnPaint 函数中,以响应 Windows 的 WM _ PAINT 消息,而 CClientDC 类对象应用在非响应消息 WM _ PAINT 的情况下。CPaintDC 类响应 WM _ PAINT 消息,自动完成绘制,这对维护图形的完整性有着重要的作用。例如在一个窗口中,已经画有 n 条直线,这个窗口完整性可能被破坏(如被对话框覆盖),当破坏完整性的程序结束时(如覆盖取消),这个窗口就会收到一个 WM _ PAINT 消息,激发消息处理函数 OnPaint 进行图形绘制。CPaintDC 类对象就担负着此时的绘制任务。如果现在想在屏幕上再绘制一条直线(如用鼠标实时绘制,当第二次点中鼠标左键时完成绘制),就要用到 CClientDC 类对象,这个类可以实时地将图形绘制到屏幕上。如果用 CPaintDC 类对象完成同样的工作,只能发出消息让屏幕上包

含这条直线的区域重画，以把这条直线绘制到屏幕上。当然，重画区域内的其他图形元素同时也被重画。

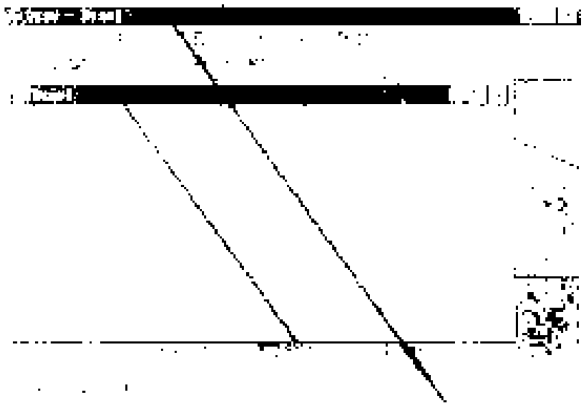


图 4-1　CWindowDC 类与 CPaintDC 类和 CClientDC 类的区别

4.1.3　绘图类的使用方法

(1) CDC 类

因为 CDC 类不能用窗口对象指针初始化对象，所以一般不直接定义对象应用，但经常用来建立一个内存设备描述对象，创建方法如下：

```
CDC dc;
```

(2) CPaintDC 类

CPaintDC 类一般用在窗口类的 OnPaint 函数中，可采用如下代码定义一个 CPaintDC 类对象：

```
CPaintDC dc(this);
```

以上代码定义了一个 CPaintDC 类对象 dc，并用当前的窗口对象指针 this 对对象进行了初始化。CClientDC 类和 CWindowDC 也采用这样的使用方法。

(3) CClientDC 类

对于 CClientDC 类，利用以下代码定义一个对象：

```
CClientDC dc(this);
```

以上代码创建定义了一个 CClientDC 类对象 dc，并采用当前窗口对象指针 this 对对象进行了初始化。

(4) CWindowDC 类

CWindowDC 类一般在框架类 CMainFrame 中使用,以下代码创建一个 CWindowDC 类对象:

```
CWindowDC dc(this);
```

以上代码创建了一个 CWindowDC 类的对象 dc,并采用当前窗口对象指针 this 对对象进行了初始化。

4.2 绘图设备类

绘图设备类是一些已经定义的 MFC 类。通过这些绘图设备类定义的对象能够完成图形操作过程中的某类操作功能,可以将这些绘图设备类创建的对象选入到绘图类对象中,完成有关的操作。在 MFC 中,主要有以下几个绘图设备类:CGdiObject、CPen、CBrush、CFont、CRgn、CPalette、CBitmap,本节将讨论这些绘图设备类的使用。

4.2.1 CGdiObject 类

CGdiObject 类是 CObject 类的派生类,它为不同的 Windows 图形设备界面(GDI)提供了基类。CGdiObject 为它的派生类提供了大部分的操作,可利用这些派生类创建一个特定的 GDI 对象(如画笔、画刷、字体、位图等)。CGdiObject 类不能直接用来创建一个对象。

4.2.2 CPen 类

CPen 类是 CGdiObject 类的一个派生类,它模拟画笔图形设备界面。其定义如下:

```
class CPen : public CGdiObject
{
DECLARE_DYNAMIC(CPen)

public:
static CPen* PASCAL FromHandle(HPEN hPen);
// Constructors
CPen();
CPen(int nPenStyle, int nWidth, COLORREF crColor);
#ifndef _MAC
CPen(int nPenStyle, int nWidth, const LOGBRUSH* pLogBrush,
```

```
        int nStyleCount = 0, const DWORD * lpStyle = NULL);
#endif
BOOL CreatePen(int nPenStyle, int nWidth, COLORREF crColor);
#ifndef _MAC
BOOL CreatePen(int nPenStyle, int nWidth, const LOGBRUSH * pLogBrush,
        int nStyleCount = 0, const DWORD * lpStyle = NULL);
#endif
BOOL CreatePenIndirect(LPLOGPEN lpLogPen);

// Attributes
operator HPEN() const;
int GetLogPen(LOGPEN * pLogPen);
#ifndef _MAC
int GetExtLogPen(EXTLOGPEN * pLogPen);
#endif

// Implementation
public:
virtual ~CPen();
#ifdef _DEBUG
virtual void Dump(CDumpContext& dc) const;
#endif
};
```

以下对 CPen 对象的创建和在绘图类对象中的引用方法进行介绍。

(1) 创建 CPen 对象

有以下几种方法来创建一个 CPen 对象：

① 定义一个 CPen 对象，用其成员函数 CreatePen 或 CreatePenIndirect 对其进行初始化。以下是用 CreatePen 初始化一个画笔的代码：

```
CPen pen;
pen.CreatePen(PS_SOLID,1,RGB(255,0,0));
```

以上代码定义一个 CPen 类对象 pen，并把这个 CPen 类对象初始化为实心笔，1 个像素宽，红色。

② 定义一个 CPen 对象，并一次性地初始化它的所有参数。

```
CPen pen(PS_SOLID,1,RGB(255,0,RGB(255,0,0));
```

用这种方法一次性的创建了一个同样的 CPen 对象。

③ 动态创建一个 CPen 对象。

```
CPen * pen;
pen = new CPen(PS_SOLID,1,RGB(255,0,0));
```

除了 CPen 对象是动态分配的外，这种构造方法实际上与②相同。在一个函数中多次创建一个 CPen 对象时，可以采用这种方法。应用这种方法时，特别注意操作完毕后要删除分配的 CPen 对象：

```
delete pen;
```

不论采用哪一种方法，都是用三个参数来初始化画笔。三个参数是线型(nPenStyle)、线宽(nWidth)和颜色(COLORREF crColor)。

参数 nPenStyle　设置画笔画线的线型，可选的线型如图 4-2 所示。注意，线型 PS_DASH、PS_DOT、PS_DASHDOT、PS_DASHDOTDOT 只有在画笔宽度为 1 个逻辑单位时才有效。

序号	名称	线型
0	PS—SOLID	————
1	PS—SASH	- - - - - - - - - -
2	PS—DOT	----------
3	PS—DASHDOT	—— · —— · ——
4	PS—DASHDOTDOT	———— · · ————
5	PS—NULL	
6	PS—INSIDEFRAME	————

图 4-2　nPenStyle 可选的线型

参数 nWidth　设置以逻辑单位衡量的线宽。

参数 crColor　设置画笔的颜色，颜色通过 RGB 宏来确定：

```
COLORREF RGB(cRed,cGreen,cBlue);
```

参数 cRed、cGreen、cBlue 分别指定红、绿、蓝三色的相对亮度，每个参数均赋予 0 到 255 之间的值。

(2) 在绘图对象中选入画笔

以在 CDrawView::OnDraw(CDC * pDC)函数中操作为例，讨论在绘图对象中选入画笔即 CPen 对象的方法：

```
void CDrawView::OnDraw(CDC * pDC)
{
    CPen pen(PS_SOLID,1,RGB(255,0,0));          //定义并初始化一个画笔实例
    CPen * pOldPen = pDC->SelectObject(&pen);   //将画笔选入绘图对象
    ……//省略各种绘制操作代码
    pDC->SelectObject(pOldPen);                 //恢复原来的画笔
}
```

在 CClientDC、CWindowDC 等其他绘图对象中，可以用类似的方法来选入 CPen 对象。

4.2.3　CBrush 类

CBrush 类是 CGdiObject 的一个派生类，它模拟一个 Windows 画刷图形设备界面。CBrush

类的用法与 CPen 类似。以下是 MFC 中 CBrush 类的定义：

```
class CBrush : public CGdiObject
{
DECLARE_DYNAMIC(CBrush)
public:
static CBrush* PASCAL FromHandle(HBRUSH hBrush);
// 构造函数
CBrush();
CBrush(COLORREF crColor);                    // CreateSolidBrush
CBrush(int nIndex, COLORREF crColor);        // CreateHatchBrush
CBrush(CBitmap* pBitmap);                    // CreatePatternBrush
BOOL CreateSolidBrush(COLORREF crColor);
BOOL CreateHatchBrush(int nIndex, COLORREF crColor);
BOOL CreateBrushIndirect(const LOGBRUSH* lpLogBrush);
BOOL CreatePatternBrush(CBitmap* pBitmap);
BOOL CreateDIBPatternBrush(HGLOBAL hPackedDIB, UINT nUsage);
BOOL CreateDIBPatternBrush(const void* lpPackedDIB, UINT nUsage);
BOOL CreateSysColorBrush(int nIndex);
// Attributes
operator HBRUSH() const;
int GetLogBrush(LOGBRUSH* pLogBrush);
// Implementation
public:
virtual ~CBrush();
#ifdef _DEBUG
virtual void Dump(CDumpContext& dc) const;
#endif
};
```

(1) 创建 CBrush 对象

CBrush 类有 4 个构造函数，本节只对主要的几个构造函数进行介绍。以下按照实心、影线及位图填充几类画刷来讨论 CBrush 对象的创建过程。

实心画刷

用实心画刷进行图形填充时，以这种画刷的颜色实心填充。与 CPen 对象类似，可以有几种构造 CBrush 对象的方法：

① 一次性创建。定义 CBrush 类对象，并利用构造函数对对象的参数进行初始化。

```
CBrush brush(RGB(255,0,0));
```

② 动态创建。用这种方法定义时，在操作完成后要删除创建的 CBrush 对象。

```
CBrush * brush;
brush = new CBrush(RGB(255,0,0));
......
delete brush;
```

③ 定义 CBrush 对象，用函数 CreateSolidBrush 对对象进行初始化。

```
CBrush brush;
brush.CreateSolidBrush(RGB(255,0,0));
```

影线画刷

影线画刷不是用画刷的颜色实心填充图形，而是填充不同的阴影图案，填充方式如图 4-3 所示。影线画刷的创建方法同实心画刷。

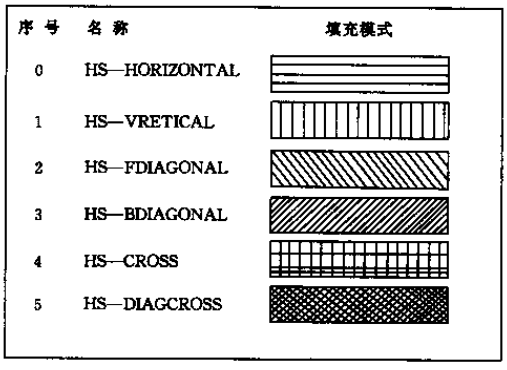


图 4-3　画刷的影线方式

① 一次性创建。

```
CBrush brush(HS_HORIZONTAL,RGB(255,0,0));
```

② 动态创建。

用这种方法创建的 CBrush 对象，在操作完成后要删除 CBrush 对象。

```
CBrush * brush;
brush = new CBrush(HS_HORIZONTAL,RGB(255,0,0));
......
delete brush;
```

③ 定义 CBrush 对象，用函数 CreateSolidBrush 初始化。

```
CBrush brush;
brush.CreateSolidBrush(PS_SOLID,RGB(255,0,0));
```

位图画刷

位图画刷用位图图像来填充图形。函数 CreatePatternBrush 用来初始化一个位图画刷。以下代码创建一个位图画刷，pBitmap 指向一个位图对象：

```
CBrush brush;
brush.CreatePatternBrush(pBitmap);
```

(2) 在绘图对象中选入画刷

在绘图对象中选入画刷，即 CBrush 对象的操作，是与选入画笔的操作相同的。下面以在 CDrawView::OnDraw(CDC * pDC)函数中为例予以介绍：

```
void CDrawView::OnDraw(CDC * pDC)
{
    CBrush brush(HS_CROSS,RGB(255,0,0));              //定义并初始化一个 CBrush 对象
    CBrush * pOldBrush = pDC->SelectObject(&brush);   //将画刷选入绘图对象
    ……//省略以下的绘制操作
    pDC->SelectObject(pOldBrush);                     //恢复原来的画刷
}
```

4.2.4　CFont 类

CFont 类是 CGdiObject 类的一个派生类，它模拟字体。以下是 MFC 中这个类的定义：

```
class CFont : public CGdiObject
{
    DECLARE_DYNAMIC(CFont)
    public:
        static CFont * PASCAL FromHandle(HFONT hFont);

// Constructors
    CFont();
    BOOL CreateFontIndirect(const LOGFONT * lpLogFont);
    BOOL CreateFont(int nHeight, int nWidth, int nEscapement,
        int nOrientation, int nWeight, BYTE bItalic, BYTE bUnderline,
        BYTE cStrikeOut, BYTE nCharSet, BYTE nOutPrecision,
        BYTE nClipPrecision, BYTE nQuality, BYTE nPitchAndFamily,
        LPCTSTR lpszFacename);
    BOOL CreatePointFont(int nPointSize, LPCTSTR lpszFaceName,CDC * pDC = NULL);
    BOOL CreatePointFontIndirect(const LOGFONT * lpLogFont, CDC * pDC = NULL);
// Attributes
    operator HFONT() const;
    int GetLogFont(LOGFONT * pLogFont);
// Implementation
public:
    virtual ~CFont();
 # ifdef _DEBUG
    virtual void Dump(CDumpContext& dc) const;
```

```
#endif
};
```

(1) 创建 CFont 对象

CFont 类只有一个构造函数,定义一个 CFont 对象后,必须用成员函数 CreateFont 或 CreateFontIndirect 来初始化 CFont 对象的参数。

CreateFont 函数的参数非常多,各个参数的含义如下:

参数 nHeight: 定义以逻辑单位度量的字体高度。

参数 nWidth: 定义以逻辑单位度量的字体的平均宽度。

参数 nEscapement: 定义字体标注行的逆时针方向的旋转角度(以 1/10 度为单位)。正常情况(即水平进行标注时)的角度为 0。当标注行逆时针旋转时,字体也旋转同样的角度。图 4-4 是当旋转角度为 100(10 度)时显示的字符串。

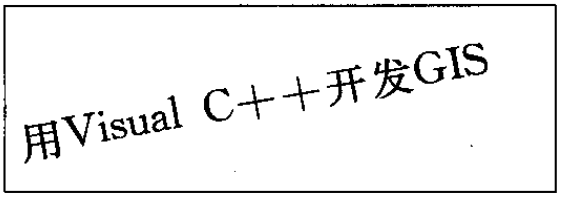


图 4-4 旋转角度为 10 度时的文本绘制

参数 nOrientation: 定义字体的角度(以 1/10 度为单位)。该角度是一个字符的基线与 X 轴之间的夹角。当 Y 坐标的值是向下为增加时,计算从 X 轴开始逆时针方向的角度;相反,当 Y 坐标的值是向上增大时,计算从 X 轴开始顺时针方向的角度。

参数 nWeight: 定义了字体的浓度既字体的粗细程度(像素数/1000)。表 4-1 列出了参数 nWeight 的通用常量的值。图 4-5 是分别用 900 和 100 书写的同样文字的对比。

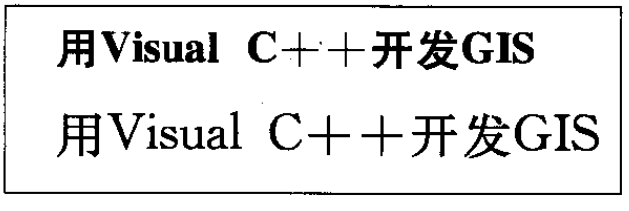


图 4-5 不同粗细程度标注文字的比较

表 4-1 字体的粗细程度(浓度)通用常量

变 量	值
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200

续表

变　量	值
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGUNAL	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

布尔参数 bItalic： 确定字体是否为斜体。

布尔参数 bUnderline： 确定字体是否加下划线。

参数 cStrikeOut： 确定字体是否被穿透(字符中央有一条横线)。图 4-6 是将参数 bItalic、bUnderline、cStrikeOut 设为 1 后输出的字符串。

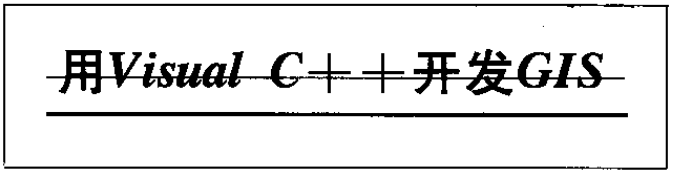


图 4-6　参数 bItalic、bUnderline、cStrikeOut 为 1 时的文字显示

参数 cCharSet： 指出了字体的字符集。表 4-2 定义了预定义的字符集常量的值。

表 4-2　字符集常量

常　量	值
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255(依赖于系统)

参数 nOutputPrecision： 指出了所要求的输出精度。该精度确定了输出与所要求的字体高度、宽度、控制、字符方位及间距的匹配和接近程度。我们所建立的字体的高度，宽度等并不是与实际的参数确定的值完全相同的，匹配越准确，显示时花费的时间就越长。这个参数可以控制输出匹配和接近程度。一般取下列值之一：OUT_CHARACTER_PRECIS, OUT_DEFAULT_PRECIS, OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS, OUT_STRING_PRECIS, OUT_

STROKE _ PRECIS 和 OUT _ TT _ PRECIS。

参数 nClipPrecision：指定了所要求的裁剪精度。当对图像的一个区域进行裁剪时，字体可能正好在裁剪线上。该参数说明怎样裁剪位于裁剪区之外的部分字符。

参数 nQuality：指出了输出字体的重量。

参数 nPitchAndFamily：确定了字体的间距和所属的族。这个 8 位参数的低 2 位确定了字体的间距，它们可以是 DEFAULT _ PITCH、FIXED _ PITCH 或 VARIABLE _ PITCH。高 4 位指定了字体族。

参数 lpszFacename：是一个指向 ASZIIZ 字符串的指针。该字符串确定了目标字体的字型名。该参数的值一定不能超过 30 个字符，这个名称可由用户任意定义。

(2) CFont 对象的选用

以下介绍创建 CFont 对象和在绘图对象中选用的过程。以函数 CDrawView::OnDraw(CDC * pDC)为例进行介绍，在函数中加入以下代码：

```
void CDrawView::OnDraw(CDC * pDC)
{
    CFont font;                                    //定义一个 CFont 对象
    //以下用各种参数初始化 CFont 对象
    font.CreateFont(30,15,0,0,200,0,0,0,255,0,0,2,DEFAULT _ PITCH, "VC");
    CFont * pOldFont = pDC -> SelectObject(&font);//将 CFont 对象选入绘图对象
    ……//省略进行文字绘制的部分
    pDC -> SelectObject(pOldFont);                 //恢复绘图对象中原来的字体对象
}
```

4.2.5 CRgn 类

CRgn 类是 CGdiObject 类的一个派生类，它封装了区域的种种特性和操作。它是窗口中的椭圆或多边形区域的组合，可被用来进行填充、反显、移动、判定某点是否在区域等操作，CRgn 在 MFC 中被定义如下：

```
class CRgn : public CGdiObject
{
    DECLARE _ DYNAMIC(CRgn)
public:
    static CRgn * PASCAL FromHandle(HRGN hRgn);
    operator HRGN() const;
// Constructors
    CRgn();
    BOOL CreateRectRgn(int x1, int y1, int x2, int y2);
    BOOL CreateRectRgnIndirect(LPCRECT lpRect);
    BOOL CreateEllipticRgn(int x1, int y1, int x2, int y2);
    BOOL CreateEllipticRgnIndirect(LPCRECT lpRect);
```

```
    BOOL CreatePolygonRgn(LPPOINT lpPoints, int nCount, int nMode);
# ifndef _ MAC
    BOOL CreatePolyPolygonRgn(LPPOINT lpPoints, LPINT lpPolyCounts,
        int nCount, int nPolyFillMode);
# endif
    BOOL CreateRoundRectRgn(int x1, int y1, int x2, int y2, int x3, int y3);
# ifndef _ MAC
    BOOL CreateFromPath(CDC * pDC);
    BOOL CreateFromData(const XFORM * lpXForm, int nCount,
        const RGNDATA * pRgnData);
# endif
// Operations
    void SetRectRgn(int x1, int y1, int x2, int y2);
    void SetRectRgn(LPCRECT lpRect);
    int CombineRgn(CRgn * pRgn1, CRgn * pRgn2, int nCombineMode);
    int CopyRgn(CRgn * pRgnSrc);
    BOOL EqualRgn(CRgn * pRgn) const;
    int OffsetRgn(int x, int y);
    int OffsetRgn(POINT point);
    int GetRgnBox(LPRECT lpRect) const;
    BOOL PtInRegion(int x, int y) const;
    BOOL PtInRegion(POINT point) const;
    BOOL RectInRegion(LPCRECT lpRect) const;
# ifndef _ MAC
    int GetRegionData(LPRGNDATA lpRgnData, int nCount) const;
# endif
// Implementation
    virtual CRgn();
};
```

(1) 创建 CRgn 对象

CRgn 类的构造函数只有一个，定义一个 CRgn 对象后，必须调用一个初始化函数对对象进行初始化后才能被使用。下面对它的一些常用初始化函数进行介绍：

① 函数 CreateRectRgn 和 CreateRectRgnIndirect

这两个函数在 CRgn 类中的完整定义如下：

```
BOOL CreateRectRgn(int x1, int y1, int x2, int y2);
BOOL CreateRectRgnIndirect(LPCRECT lpRect);
```

这两个函数都是用来初始化一个矩形区域，矩形区域的顶点坐标必须在小于 32767 并大于 -32768 范围内(逻辑坐标)。在第 1 个函数中，矩形区域的对角坐标分别为(x1,y1)和(x2,y2)，在第 2 个函数中，矩形区域的对角坐标存储在一个 RECT 结构或 CRect 对象中。下面代码

分别调用了这两个函数，初始化形成一个矩形区域(0,0,200,200)：

```
CRect rr;          //定义一个 CRect 对象
CRgn m_Rgn;   //定义一个 CRgn 对象
m_Rgn.CreateRectRgn(0,0,200,100);   //用 CreateRectRgn 函数初始化 CRgn 对象
rr.left = 0; rr.top = 0;
rr.right = 200; rr.bottom = 100;
m_Rgn.CreateRectRgnIndirect(&rr);      //用 CreateRectRgnIndirect 函数初始化 CRgn 对象
```

② 函数 CreateEllipticRgn 和 CreateEllipticRgnIndirect

这两个函数在 CRgn 类中的完整定义如下：

```
BOOL CreateEllipticRgn(int x1, int y1, int x2, int y2);
CreateEllipticRgnIndirect(LPCRECT lpRect);
```

这两个函数都用来初始化一个椭圆区域，椭圆的矩形边界顶点坐标必须小于 32767 并大于 -32768(逻辑坐标)。在用 CreateEllipticRgn 函数初始化椭圆区域时，椭圆的矩形边界顶点坐标用参数(x1,y1)和(x2,y2)来表示。用 CreateEllipticRgnIndirect 函数来初始化椭圆区域时，椭圆的矩形边界顶点坐标用一个 RECT 结构或一个 CRect 对象来表示。这两个函数的使用方法与①中的两个函数相同。

③ 函数 CreatePolygonRgn

函数在 CRgn 类中的完整定义为：

```
BOOL CreatePolygonRgn(LPPOINT lpPoints, int nCount, int nMode);
```

CreatePolygonRgn 函数初始化一个多边形区域，多边形区域的顶点坐标必须小于 32767 并大于 -32768(逻辑坐标)。参数 lpPoints 指向一个 POINT 结构或 CPoint 对象数组，Count 是多边形区域顶点的数目，nMode 是多边形的填充模式，可选用参数 ALTERNATE 或 WINDING。如下代码创建了一个三角形区域，三角形的顶点为(0,0)，(10,10)，(20,0)：

```
CPoint point[3];                    //定义一个 CPoint 对象数组
CRgn m_Rgn;                         //定义一个 CRgn 对象
//以下设置 CPoint 对象数组中存储的三角形区域顶点的坐标
point[0].x = 0;point[0].y = 0;
point[1].x = 10;point[1].y = 10;
point[2].x = 20;point[2].y = 0;
m_Rgn.CreatePolygonRgn(point,3,0);   //初始化一个三角形区域
```

④ 函数 CreatePolyPolygonRgn

函数在 CRgn 类中的完整定义为：

```
BOOL CreatePolyPolygonRgn(LPPOINT lpPoints, LPINT lpPolyCounts, int nCount, int nPolyFillMode);
```

CreatePolyPolygonRgn 函数用来初始化一个包含一系列多边形的区域，其顶点坐标必须小于 32767 并大于 -32768(逻辑坐标)。创建的多边形可是独立的，也可相交。lpPoints 指向一个 POINT 结构数组或 CPoint 对象数组，数组中的每个结构或对象用来表示一个顶点的坐标。参数 lpPolyCounts 是一个整数数组，存储的是各个多边形的顶点数。参数 nCount 表示区域中多边

形的数目，也就是表示 lpPolyCounts 数组的项数。参数 nPolyFillMode 表示填充模式。以下代码用函数 CreatePolyPolygonRgn 创建了包括三个多边形的区域：

```
CRgn m_Rgn;
POINT point[12];                                    //定义一个 POINT 结构数组
...//省略对坐标的数值输入
int nCount[3];                                      //定义一个整数数组
nCount[0]=3; nCount[1]=4; nCount[2]=5;              //区域中各个多边形的顶点数目
m_Rgn. CreatePolyPolygonRgn(point,nCount,3,0);      //初始化区域对象
```

以下代码初始化一个包含三个多边形的区域，三个多边形的顶点数分别为 3,4,5，结构数组 Point 中存放的是顶点坐标，一共有 12 个顶点，顺序放着三个多边形的顶点坐标。

(2) 删除 CRgn 对象

一个 CRgn 类的构造函数没有进行对象的初始化工作，定义对象后，由(1)中介绍的一些函数对对象进行初始化。同样，CRgn 类的析构函数也不能删除初始化后的 CRgn 对象，需要调用基类的 DeleteObject 函数来删除。例如，对于一个初始化的 CRgn 对象 m_Rgn，采用下面的代码删除它：

```
m_Rgn.DeleteObject();   //删除 m_Rgn 对象
```

(3) 常用操作

以下对 CRgn 类中的几个常用的成员函数进行介绍，讨论这些函数所完成的操作功能。

① CombineRgn 函数

函数在 CRgn 类中的完整定义为：

```
int CombineRgn(CRgn * pRgn1, CRgn * pRgn2, int nCombineMode);
```

CombineRgn 函数也可以初始化一个区域，不过这个区域是通过两个已经存在区域的相互关系确定的。参数 PRgn1 和 pRgn2 分别指向两个已经存在的 CRgn 对象，参数 nCombineMode 是两个参数指向的区域相互操作时采用的模式，下面是参数 nCombineMode 可以采用的一些参数值及对应的操作模式：

RGN_AND:　　以两个区域的相交部分初始化新区域。
RGN_COPY:　　初始化一个同第 1 区域相同的区域。
RGN_DIFF:　　用第 1 区域中没有与第 2 个区域相交的部分初始化区域。
RGN_OR:　　初始化一个包含两个区域的区域。
RGN_XOR:　　初始化一个包含两个区域，但不包含两个区域相交部分的区域。

根据操作结果，函数返回如下值，其含义如下：

COMPLEXREGION:　区域覆盖边界。
ERROR:　　操作无效。
NULLREGION:　　区域为空。
SIMPLEREGION:　　区域没有覆盖边界。

② CopyRgn 函数

函数在 CRgn 类中的完整定义为：

```
int CopyRgn(CRgn * pRgnSrc);
```

CopyRgn 函数用一个存在的区域复制初始化当前的区域，该函数是 CombineRgn 函数采用 RGN_COPY 参数时的一种特殊情况。如下操作可把区域 m_Rgn1 初始化成与区域 m_Rgn2 相同：

```
CRgn m_Rgn1,m_Rgn2;                    //定义两个 CRgn 对象
m_Rgn2.CreateRectRgn(0,0,200,100);     //初始化 m_Rgn2 对象
m_Rgn1.CopyRgn(&m_Rgn2);               //将 m_Rgn1 对象初始化成与 m_Rgn2 对象相同
```

③ EqualRgn 函数

函数在 CRgn 类中的完整定义为：

```
BOOL EqualRgn(CRgn * pRgn) const;
```

EqualRgn 函数检查当前区域与参数 pRgn 指向的区域是否相同。相同时函数返回 TRUE，否则函数返回 FALSE。以下操作代码将返回 TRUE(yN = 1)：

```
CRgn m_Rgn1,m_Rgn2;                    //定义两个 CRgn 对象
m_Rgn2.CreateRectRgn(0,0,200,100);     //初始化 CRgn 对象 m_Rgn2
m_Rgn1.CopyRgn(&m_Rgn2);               //将 m_Rgn1 对象初始化成与 m_Rgn2 对象相同
BOOL yN = m_Rgn1.EqualRgn(&m_Rgn2);    //检查两个区域是否相同
```

④ OffsetRgn 函数

OffSetRgn 函数在 CRgn 类中有两个重载定义：

```
int OffsetRgn(int x, int y);
int OffsetRgn(POINT point);
```

OffSetRgn 函数的作用是将当前的区域相对移动一个距离，函数的参数代表横纵移动的距离量(在第一个函数中，横向移动量为 x，纵向移动量为 y，在第二个函数中，横向移动量为 point.x，纵向移动量为 point.y)。函数的返回值及含义同①中的 CombineRgn 函数。

⑤ GetRgnBox 函数

函数在 CRgn 类中的完整定义为：

```
int GetRgnBox(LPRECT lpRect) const;
```

GetRgnBox 函数用来得到区域的边界矩形，边界矩形是能够包含一个区域的最小矩形，边界矩形的对角顶点坐标通过一个 RECT 结构或一个 CRect 对象返回。如下代码将定义一个区域，并返回区域的边界矩形：

```
CPoint point[3];      //定义一个 CPoint 对象数组
CRgn m_Rgn;           //定义一个 CRgn 对象
RECT rr;              //定义一个 CRect 对象
point[0].x = 0;point[0].y = 0;
point[1].x = 10;point[1].y = 10;
```

```
point[2].x = 20;point[2].y = 0;
m_Rgn.CreatePolygonRgn(point,3,0);      //初始化一个有 3 个顶点的多边形区域
m_Rgn. GetRgnBox(&rr);
```

实际运行以上代码,会得到边界矩形|0,0,20,10|

⑥ PtInRegion 函数

PtInRegion 函数在 CRgn 类中有两个重载定义:

此函数用两个重载函数,其完整定义如下:

```
BOOL PtInRegion(int x, int y) const;
BOOL PtInRegion(POINT point) const;
```

PtInRegion 函数用来检查一个点是否在当前区域内,如果在当前区域内,函数返回 TRUE,否则返回 FALSE。点的坐标在两个重载函数中分别通过两个整型变量 x 和 y 或一个 POINT 结构 point 确定。以下代码创建一个区域,并能够检查一个点是否在区域内:

```
CPoint point[3];                        //定义一个对象数组
CRgn m_Rgn;                             //定义一个 CRgn 对象
POINT point;
point[0].x = 0;point[0].y = 0;
point[1].x = 10;point[1].y = 10;
point[2].x = 20;point[2].y = 0;
m_Rgn.CreatePolygonRgn(point,3,0);      //初始化一个有 3 个顶点的多边形区域
point.x = 5; point.y = 5;               //需要检查的点的坐标
BOOL YN = m_Rgn. PtInRegion(point);
```

如果将以上代码放在函数中运行,得到 YN 的值为 1(TRUE),表明点在区域内。

⑦ RectInRegion 函数

函数在 CRgn 类中的完整定义为:

```
BOOL RectInRegion(LPCRECT lpRect) const;
```

RectInRegion 函数用来检查一个矩形是否在一个区域内。矩形的坐标存储在参数 lpRect 中。如果矩形在区域内,函数返回 TRUE,否则函数返回 FALSE。RectInRegion 的具体使用方法与函数 PtInRegion 类似,不再举例。

(4) CRgn 对象的引用

创建并初始化的 CRgn 对象,可以作为其他函数的参数被引用。本节讨论的 CRgn 类的成员函数 CombineRgn 中,就使用 CRgn 对象作为参数。还有其他一些函数也使用 CRgn 对象作为参数,如在视图类中。以下代码使视图中 m_Rgn 对象确定的区域失效并重画:

```
InvalidateRgn(&m_Rgn);
```

4.2.6　CPalette 和 CBitmap 类

调色板是应用程序和彩色输出设备(如彩显)之间的接口。这个接口允许应用程序充分利

用彩色设备的颜色特性,而不太影响其他应用程序的显示效果。Windows 使用应用程序的逻辑调色板和系统调色板来确定使用的颜色。类 CPalette 封装了 Windows 调色板操作的各方面的功能。CBitmap 类封装了进行位图操作的各方面的功能。

本书中没有具体讨论有关 CPalatte 和 CBitmap 类的具体操作。作者所著的《Visual C++ 高级编程技术——开发实例剖析》一书中,详细讨论了 CPatette 类和 CBitmap 类的使用方法,实例介绍了显示位图图像的多种方法,请读者参阅。

4.3 坐标映像方式

在 VC++ 中进行 Windows 应用程序设计时,可以采用多种不同的映像方式即不同的坐标系。本节对映像方式的分类、特点和设置方法进行介绍。

4.3.1 Windows 映像方式及其设置方法

(1) Windows 映像方式

Windows 映像方式就是 Windows 下的逻辑坐标方式。一个实际物理屏幕是由像素组成的,如平常所说的 640×480,1024×768 指的就是物理屏幕的实际宽度和高度的像素数目。为了方便各种情况下的程序开发,减轻程序开发的负担,Windows 提供了几种映像方式,每一种映像方式提供不同的测量单位和坐标原点。Windows 的映像方式如表 4-3 所示。

表 4-3 Windows 映像方式

映像方式	映像识别码	逻辑单位	X 和 Y 轴正向
MM_TEXT	1	Pixels	右下
MM_LOMETRIC	2	0.1mm	右上
MM_HIMETRIC	3	0.01mm	右上
MM_LONGLISH	4	0.1inch	右上
MM_HIENGLISH	5	0.01inch	右上
MM_TWIPS	6	1/1440inch	右上
MM_ISOTROPIC	7	Variable(x = = y)	变化的
MM_ANISOTROPIC	8	Variable(x < > y)	变化的

MM_TEXT 方式(默认方式)基本上等价于 DOS 文本方式,不同的是 DOS 在字符状态下操作,而 MM_TEXT 方式允许应用程序利用设备像素工作,这是一种与设备坐标方式相同的映像方式。屏幕(窗口)的原点约定在左上角点,而 X 和 Y 的正方向分别为向右和向下。

MM_LOMETRIC,MM_HIMETRIC,MM_LONGLISH,MM_HIENGLISH 四种映像方式默认的坐标原点在窗口的左上角,其坐标的正方向为向右和向上(笛卡尔坐标)。坐标位置用英寸(in)或毫米(mm)表示,这对一些应用程序(例如一个绘图程序)来说是很方便的,可以直接利用原来的单位绘图,而不必进行坐标转换。

MM _ ISOTROPIC 和 MM _ ANISOTROPIC 映像方式用的也是笛卡尔坐标，这两种映像方式提供可变的逻辑单位和可变的原点。在 MM _ ISOTROPIC 映像方式中，X 轴和 Y 轴方向坐标刻度是相同的。而 MM _ ANISOTROPIC 方式，允许对 X 轴和 Y 轴设置不同的刻度。

MM _ TWRIPS 方式是一种专门为排版设计的映像方式，TWIPS 是对“一个点的 1/20”这个词稍微删改后的缩写。“点”是排版的一个术语，也称之为“磅”(简写 pt)，一点等于 1/72 英寸。设立这个映像方式完全是为了设计排版用应用程序的需要。

这八种 Windows 映像方式，是为了满足各种具体方式下的需要，以 MM _ TEXT 方式(像素方式)为基础，通过在 Windows 内置了一些完成各种映像方式下的逻辑坐标到屏幕设备坐标的转换函数来实现的。任何一种映像方式都可以用 MM _ TEXT 方式通过程序设计转换而成。不同的是，直接应用各种映像方式时的转换是由 Windows 内置程序来完成的，不用程序设计者自己编制程序。

(2) 设置映像方式的方法

在 MFC 应用程序中，通过调用 CDC 类的 SetMapMode 函数设置映像方式。如在 CDrawView:: OnDraw(CDC * pDC)函数中，可以通过以下代码设置第 n 种映像方式：

```
pDC - > SetMapMode(n);
```

n 是在表 4-3 中列出的映像方式的识别码。

4.3.2　逻辑坐标和设备坐标的转换

(1) 设备无关性

Windows 系统提供了八种映像方式，并实现了设备无关性。所谓设备无关性，指的是只要设定了映像方式，那么就会在不同的设备上显示出同样大小的图形。例如，在 MM _ LOMETRIC (单位是 0.1mm)下画一条 100 逻辑单位的线，不管在什么样的显示器和分辨率下这条线的显示长度为 1 厘米(100 个 0.1 毫米)，在打印机上打印输出时长度也是 1 厘米，Windows 系统内部实现了这个与设备无关的映像过程。所以使用 MM _ LOMETRIC、MM _ HIMETRIC、MM _ LONGLISH、MM _ HIENGLISH、MM _ TWRIPS 这些映像方式时，可以保证在不同的显示和输出设备上显示相同大小的图形。对于 MM _ TEXT 映像方式来说，因为其单位是像素，长度不固定(即一个像素所占的实际长度随着设备的不同而不同)，所以在不同的显示和输出设备上不能保证图形的大小一样。

(2) 逻辑坐标和像素坐标的转换方法

为了完成各种映像方式下逻辑坐标和设备坐标的转换，VC++ 提供了两个成员函数：

```
DPtoLP(LPPOINT lpPoints);    //实现从设备坐标转换到逻辑坐标
LPtoDP(LPPOINT lpPoints);    //实现从逻辑坐标转换到设备坐标
```

以下代码将一个点由设备坐标转换成逻辑坐标：

```
CPoint point;                    //定义一个 CPoint 对象
point.x = 100;
point.y = 100;                   //给点赋一坐标值
pDC - > DPtoLP(&point);          //将点从设备坐标转换成逻辑坐标
```

4.4　窗口和视口

本节讨论窗口和视口的概念,以及窗口和视口的坐标转换方法。

4.4.1　窗口和视口的概念

窗口指的是虚拟存在的一个屏幕,而视口是在屏幕上看到的视图客户区域。之所以分为窗口和视口,是与 Windows 进行图形绘制的机制有关。Windows 绘制图形时,并不是把图形直接绘制到屏幕上,而是以各种映像方式下的逻辑坐标系将图形绘制到虚拟的窗口中,然后将这个窗口内的内容映像到视口中。如果是映像到屏幕上,就实现了图形的显示;如果是映像到打印机等输出设备上,就实现了图形的打印输出。

千万不要把窗口原点和视口原点与坐标系中的原点混淆起来。在这里窗口原点和视口原点指的是同一个点在窗口逻辑坐标系和视口设备坐标系中的坐标值,它决定了图形由窗口映像到视口时的相对位置。

4.4.2　窗口和视口有关的操作函数

本小节中,将介绍在 CWnd 类中与窗口和视口有关的操作函数的功能和使用方法。

(1) SetWindowOrg 函数

函数 SetWindowOrg 用来设置一个与设备环境有关的窗口原点。函数 SetWindowOrg 有如下两个重载如下:

```
CPoint SetWindowOrg(int x,int y);
CPoint SetWindowOrg(POINT point);
```

参数 x 和 y 或 point 用来记录要设置的窗口原点。函数返回设置前的窗口原点。

(2) SetViewportOrg 函数

函数 SetViewportOrg 用来设置一个与设备环境有关的视口原点。函数 SetWindowOrg 有如下两个重载定义:

```
CPoint SetViewportOrg(int x,int y);
CPoint SetViewportOrg(POINT point);
```

参数 x 和 y 或 point 用来记录要设置的视口原点。函数返回设置前的视口原点。

(3) SetWindowExt 函数

函数 SetWindowExt 用来设置与设备环境有关的窗口在 x 和 y 方向的幅度。只有在 MM_

ANISOTROPIC 和 MM _ ISOTROPIC 映像方式下该函数才有效。函数 SetWindowExt 有如下两个重载定义：

```
virtual CSize SetWindowExt(int cx,int cy);
virtual CSize SetWindowExt(SIZE size);
```

参数 cx 和 cy 或 size 用来记录要设置的 x 和 y 方向的窗口幅度。函数返回设置前的幅度值。

(4) SetViewportExt 函数

函数 SetViewportExt 用来设置与设备环境有关的视口在 x 和 y 方向的幅度。只有在 MM _ ANISOTROPIC 和 MM _ ISOTROPIC 映像方式下该函数才有效。函数 SetViewportExt 有如下两个重载定义：

```
virtual CSize SetWindowExt(int cx,int cy);
virtual CSize SetWindowExt(SIZE size);
```

参数 cx 和 cy 或 size 用来记录要设置的 x 和 y 方向的视口幅度。函数返回设置前的幅度值。必须注意，在 MM _ ISOTROPIC 映像方式下，必须先设置窗口幅度，再设置视口幅度。

4.4.3　窗口逻辑坐标与视口设备坐标的转换

接触过 VC++ 的窗口和视口的概念以及坐标转换的读者，可能对它们的操作感到模糊。VC++ 窗口和视口的作用是为了实现设备无关性，所谓设备无关性就是可以把图形以实际逻辑单位的大小显示到任何设备上(如屏幕，打印机)，而不必考虑这些设备的硬件情况。下面对表 4-3 所列的八种映像方式下的坐标转换进行讨论，以理解关于窗口和视口的概念。

(1) 逻辑坐标和设备坐标的转换方法

假定在窗口中有一幅图形，图形用的是逻辑坐标系，它的原点在(0,0)，宽度为 1000 个逻辑单位，高度也为 1000 个逻辑单位(如在 MM _ LOMETRIC 映像方式下，逻辑单位就是0.1mm)。在这幅图形中设置一个点(a1,b1)为窗口原点：

```
pDC->SetWindowOrg(a1,b1);
```

a1 和 b1 是窗口原点在窗口逻辑坐标系中的坐标值，单位是逻辑单位(如在 MM _ LOMETRIC 映像方式下，逻辑单位就是 0.1mm)。

当 Windows 把这幅图形映像到视口(如屏幕)上时，在视口中坐标单位不再是逻辑单位，而是设备坐标(像素坐标)。根据设备的无关性，图形映像在视口上时图形的大小应是不变的，但是因为视口中坐标系的单位是像素而不是逻辑单位，所以映像出来的图形用像素表示的幅度是不一样。这就要求应该有像素与逻辑单位(0.1mm)的转换比例(这个比例的大小随着显示器屏幕的大小和分边辨率的高低而不同)。假定一个像素的实际大小等于 n 个逻辑单位，那么这幅图形的实际宽度为 1000/n 个像素，实际高度为 1000/n 个像素。另外，当把这个图形映像到视口上时，还需要确定把图形映像到视口的什么位置。在窗口中，在图形上取定了一个点

(a1,b1)为窗口原点,现在只要能够确定这个点映像到视口时,在视口中的坐标位置,就确定了整个图形的映像位置。这个点在视口中的坐标可以通过设置视口原点来实现,如将视口原点设置为(a2,b2):

```
pDC->SetViewportOrg(a2,b2);
```

a2 和 b2 是视口原点在视口的像素坐标系中的坐标值。通过窗口原点和视口原点就确定了图形从窗口映像到视口时的位置,即一个在窗口的逻辑坐标系中坐标为(a1,b1)的点,映像到视口后在设备像素坐标系中的坐标值为(a2,b2)。通过窗口原点和视口原点,可以得到图形中的同一个点的设备坐标和逻辑坐标的对应关系。假定一个点在窗口逻辑坐标为(x1,y1),在视口设备像素坐标为(x2,y2),在两个坐标系 y 轴同向的情况下,根据在窗口和视口中这个点到原点的距离相等,则有如下转换关系:

```
(x1-a1)=(x2-a2)*n
(y1-b1)=(y2-b2)*n
```

在 y 轴反向的情况下,则有如下的转换关系。

```
(x1-a1)=(x2-a2)*n
(y1-b1)=-(y2-b2)*n
```

以下结合具体的映像方式来说明这个坐标转换过程。

(2) MM_TEXT 映像方式的坐标转换

MM_TEXT 映像方式是最简单的一种方式,此时窗口中的逻辑坐标也是像素坐标,方向与视口中的相同。在这种方式下,如果窗口原点和视口原点相同,设备像素坐标(视口中)和逻辑坐标(窗口中)是相同的。图形映像时,实际上是原封不动的由窗口映像到视口。当窗口原点与视口原点不同时,两者的坐标转换如下:

在 MM_TEXT 映像方式下,因为在窗口和视口中都采用像素坐标,所以 n 等于 1。依据以上讨论的坐标转换公式,可得到窗口逻辑坐标到视口设备坐标的转换关系为:

```
x2=x1-a1+a2;
y2=y1-b1+b2;
```

将窗口原点设置为(100,100),视口原点设置为(50,50)时:

```
pDC->SetWindowOrg(100,100);
pDC->SetViewportOrg(50,50);
```

则有如下的转化关系:

```
x2=x1-50+100;
y2=y1-50+100;
```

同样,可以得到视口设备坐标到窗口逻辑坐标的转换公式:

```
x1=x2+a1-a2;
y1=y2+b1-b2;
```

所以在 MM _ TEXT 映像方式下，当视口原点和窗口原点重合在一起时，窗口和视口中坐标系是完全一样的，这是系统默认的映像方式。通过移动视口原点和窗口原点都可以达到移动图形的目的。

(3) MM _ LOMETRIC 等 4 种映像方式下的坐标转换

*以上讨论了在 MM _ TEXT 映像方式下的坐标转换，现在讨论另外几种方式下的坐标转换问题。*MM _ LOMETRIC、MM _ HIMETRIC、MM _ LONGLISH、MM _ HIENGLISH、MM _ TWRIPS 等四种映像方式，除了逻辑坐标单位不一样外，其他方面都一样，现在就以 MM _ LOMETRIC 映像方式为例，讨论这四种映像方式下的坐标转换问题。

在 MM _ LOMETRIC 映像方式下，窗口中的逻辑坐标单位是 0.1mm，而视口中的坐标单位仍然是像素，假定一个像素等于 n 个逻辑单位(0.1mm)。

因为 MM _ LOMETRIC 映像方式下的逻辑坐标与设备坐标 Y 轴反向，所以得到逻辑坐标到设备坐标的转换公式为：

```
x2 = (x1 - a1)/n + a2;
y2 = -(y1 - b1)/n + b2;
```

窗口原点和视口原点的坐标都为(0,0)时，逻辑坐标到设备坐标的转换公式为：

```
x2 = x1/n;
y2 = -y1/n;      //因为逻辑坐标系和设备坐标系方向相反，所以为负值。
```

窗口原点坐标为(100,100)而视口原点坐标为(0,0)时，坐标转换公式为：

```
x2 = 100 + x1/n;
y2 = 100 - y1/n;
```

窗口原点坐标为(0,0)而视口原点坐标为(100,100)时，坐标转换公式为：

```
x2 = (x1 - 100)/n;
y2 = -(y1 - 100)/n;
```

窗口原点坐标为(100,100)而视口原点坐标为(100,100)时，坐标转换公式为：

```
x2 = (x1 - 100)/n + 100;
y2 = -(y1 - 100)/n + 100;
```

在 MM _ LOMETRIC 等四种映像方式下，虽然移动窗口原点和视口原点都可以达到移动图形的目的，但移动量是不一样的。移动窗口原点时，移动量是以逻辑坐标为单位，而移动视口原点时，移动量是以设备坐标(像素)为单位。

(4) MM _ ISOTROPIC 和 MM _ ANISOTROPIC 映像方式的坐标转换

对于 Windows 系统来说，MM _ ISOTROPIC 和 MM _ ANISOTROPIC 映像方式是非常重要的。在这两个映像方式下，有着与其他六种映像方式不同的映像概念。其他六种映像方式，一个共同的特点是由窗口映像到视口时图形的大小是不变的，而且视口都采用设备(像素)坐标。而在这两种映像方式下，映像后图形的大小要发生变化，视口中的坐标也不再以像素为单位，而

窗口中的逻辑坐标则是以像素为单位。

通过 SetWindowExt 建立一个窗口的范围，通过 SetViewportExt 建立一个视口的范围，这两个范围决定了由窗口映像到视口的操作。进行映像操作时，就像放缩复印一样把窗口内的图形复制到视口内。与前面讨论六种映像方式一样，靠窗口原点和视口原点来确定映像操作时图形的相对位置。在 MM _ ISOTROPIC 映像方式下，X 和 Y 两方向同比例放缩，而在 MM _ ANISOTROPIC 映像方式下，允许两方向不同比例放缩。

以下代码设置 MM _ ANISOTROPIC 映像方式，并设置窗口和视口的幅度：

```
pDC - > SetMapMode(MM _ ANISOTROPIC);
pDC - > SetWindowExt(100,100);          //设置窗口的 x 和 y 方向的幅度为 100 和 100 单位
pDC - > SetViewportExt(500,300);        //设置视口的 x 和 y 方向的幅度为 500 和 300 单位
```

当窗口和视口在原点都在(0,0)的情况下，进行映像操作时，就会把窗口中矩形区域(0,0,100,100)中的图形，映像到了视口的矩形区域(0,0,500,300)中。这时，在窗口和视口中图形具有相同大小的坐标值，即可以认为视口中的矩形区域也为(0,0,100,100)，因为在窗口中采用的是像素坐标，所以在视口中，一个坐标单位的宽度是 5 个像素，高度是 3 个像素，窗口中的每个像素映像到视口中对应的坐标单位去。

所以，在进行窗口幅度和视口幅度设置时，幅度值并不重要，重要的是两个方向的幅度值的比例，因为它决定了映像的操作比例。

当在以上设置的映像方式下画一个圆心在(50,50)，半径为 20 的圆时，得到如图 4-7 所示的结果。圆实际上是一个长轴长 200 像素，短轴长 120 像素的椭圆，椭圆心在视图中的像素坐标为(250,150)，椭圆的笔宽增加几倍。

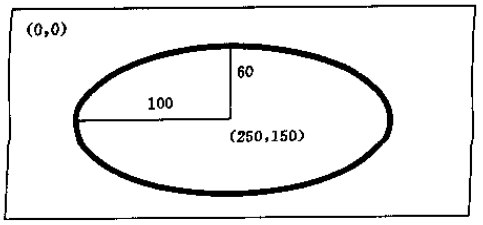


图 4-7 MM _ ANISOTROPIC 方式下窗口到视口的映像

4.5 其他绘图类操作函数

在本章 4.1 节中，介绍了绘图类 CDC 及其派生类。本节将对绘图类 CDC 中的一些功能操作做一些介绍。

4.5.1 选用绘图设备对象

一个设备环境拥有默认的绘图设备对象 GDI，如画笔和画刷等，但通常也需要选择其他定制的绘图设备 GDI 对象。针对这两种情况，有两种选择绘图设备对象的方式：选择库存对象和

选择定制对象。

(1) 选择库存绘图设备对象

成员函数 SelectStockObject 用来选择一个库存的绘图设备对象,其定义如下:

```
virtual CGdiObject * SelectStockObject(int nIndex);
```

参数 nIndex 是标识库存绘图设备对象的序号,表 4-4 列出了参数 nIndex 的主要取值。

表 4-4　可供选择的主要绘图设备对象列表

常　量	库存对象
BLACK_BRUSH	黑色刷子
DKGRAY_BRUSH	暗灰色刷子
GRAY_BRUSH	灰色刷子
HOLLOW_BRUSH	空心刷子
LTGRAY_BRUSH	亮灰刷子
NULL_BRUSH	空刷子
WHITE_BRUSH	白刷子
BLACK_PEN	黑画笔
NULL_PEN	空画笔
WHITE_PEN	白画笔

(2) 选择定制绘图设备对象

成员函数 SelectObject 用来选择定制的画笔、画刷、字体、位图等绘图设备对象等,具体操作在 4.2 节中已经作了介绍。

4.5.2　主要绘图属性设置函数

本小节介绍 CDC 类中的绘图属性设置函数。

(1) 设置和获得背景颜色

SetBkColor 函数用来设置当前背景颜色。当背景模式为 OPAQUE 时,系统使用背景颜色来填充格式线中的间隔(如点画线)、画刷中阴影线间的间隔,以及字符单元的背景。系统在彩色和单色设备之间转换位图时,也要使用背景颜色。SetBkColor 函数定义如下:

```
virtual COLORREF SetBkColor(COLORREF crColor);
```

SetBkColor 函数设置背景颜色为参数 crColor 指定的颜色,函数返回原来的背景颜色。pDC 指向当前的绘图对象时,以下代码将背景颜色设置为红色:

```
pDC->SetBkColor(RGB(255,0,0));
```

GetBkColor 函数用来取得当前的背景颜色。函数定义为:

```
COLORREF GetBkColor();
```

GetBkColor 函数返回当前的背景颜色，pDC 指向绘图对象时，以下代码取得当前的背景颜色：

```
COLORREF color = pDC - > GetBkColor();
```

(2) 设置和获得背景模式

背景模式决定了在绘制文本、影线画刷和格式线前，是否清除绘制画面上的背景彩色。主要有两种模式 OPAQUE 和 TRANSPARENT。OPAQUE 模式下，要在绘制前清除绘制画面上的背景颜色，即格式线中的间隔、画刷影线间的间隔，以及字符单元的背景，都要用当前的填充色填充。当用 TRANSARENT 模式时，绘制前不清除绘制画面上的背景颜色，即格式线中的间隔、画刷影线间的间隔，以及字符单元的背景都保持不变。CDC 类中的 SetBkMode 用来设置背景模式，函数定义如下：

```
int SetBkMode(int nBkMode);
```

SetBkMode 函数将背景模式设置参数 nBkMode 指定的模式，并返回原来的背景模式。pDC 指向当前的绘图对象时，以下代码将背景模式设置为 TRANSPARENT：

```
pDC - > SetBkMode(TRANSPARENT);
```

CDC 类中的成员函数 GetBkMode 用来取得当前的背景模式，函数定义如下：

```
int GetBkMode();
```

函数的调用方式与 SetBkMode 类似，pDC 指向当前的绘图对象时，以下代码获得当前的背景模式：

```
int nBkMode = pDC - > GetBkMode();
```

(3) 设置和获得绘制模式

绘制模式指定了画笔和填充区域内部的彩色在绘制时与屏幕上已经存在的彩色的组合模式。绘制模式只适用于光栅扫描设备(如显示器)，而不适用于矢量设备(如绘图机)。绘图模式是二进制的光栅扫描操作代码，代表了两个变量所有可能的布尔组合，这个组合通过 AND、OR、XOR、NOT 运算构成。绘制模式很多，这里只介绍常用的两种：

① R2 _ COPYPEN 模式。有画笔颜色绘制图素，即直接将画笔的颜色绘制到屏幕上，屏幕底色不起作用。在屏幕上绘制图形时一般采用这种模式。

② R2 _ NOT 模式。用屏幕底色的反色绘制图素。这种绘制模式最大的特点就是两次绘制屏幕颜色保持不变。所以，可以利用这种模式通过对图形的两次绘制(第一次绘制将图形绘制在屏幕上，第二次绘制从屏幕上擦除绘制的图形)，实现在屏幕上拖动图形。

CDC 类中的成员函数 SetROP2 用来设置当前绘制模式。函数定义如下：

```
int SetROP2(int nDrawMode);
```

SetROP2 函数将绘制模式设置为参数 nDrawMode 指定的模式，函数返回原来的绘制模式。以下代码可以把 pDC 指向的绘图对象的绘制模式设置为 R2_NOT：

```
pDC->SetROP2(R2_NOT);
```

CDC 类的成员函数 GetROP2，用来获得当前绘图对象的绘制模式，函数的定义如下：

```
int GetROP2(int nDrawMode);
```

以下代码可以得到 pDC 指向的绘图对象的绘制模式：

```
int nDrawMode = pDC->GetROP2();
```

第5章　实现基本矢量图形系统的文档和视图

从本章开始，将具体组织开发一个用于GIS系统的矢量图形系统。在2.1.3小节，用MFC AppWizard产生了一个应用程序Draw，我们将在这个应用程序基础上来组织矢量图形系统的开发工作。

本章将实例讨论基本矢量图形系统中图形元素类的组织，实现矢量图形系统的文档和视图。

5.1　组织矢量图形系统的图形元素类

面向对象的程序设计，是目前程序设计的主流方法。在本节中，将利用面向对象的程序设计和C++类的组织方法，组织建立起一个基本矢量图形系统的图形元素类。我们组织实现的基本矢量图形系统，能够处理直线、连续直线及多边形区域、圆及圆形区域、圆弧、标注文本等图形元素，针对每类图形元素组织建立起对其组织管理的C++类，把每个图形元素作为一个独立的对象来管理。

5.1.1　图形元素基类的组织

对各种图形元素进行分析，可以发现各类图形元素具有一些相同的属性和操作功能，如图形元素的颜色、线型、线宽、所在层等属性和得到一个图形元素是否做了删除标志等操作。把这些图形元素中共性的东西(属性和操作)组织存放在一个图形元素基类中，具体的图形元素类由这个类派生。

在头文件drawdoc.h中，在CDrawDoc类的定义前，定义一个图形元素基类CDraw(当然可以把类定义在新增的其他头文件中，为介绍方便起见，各类图形元素类都以drawdoc.h文件作为头文件，以drawdoc.cpp文件作为实现文件)：

```
class CDraw:public CObject          //图形元素基类,用来存储图形的颜色、线型、层等信息
{
protected:
    short m_ColorPen;               //笔色
    short m_ColorBrush;             //填充颜色
    short m_LineWide;               //线宽(像素)
    short m_LineType;               //线型
    short m_Layer;                  //所处图层
    BOOL b_Delete;                  //是否处于删除状态
    Int m_id_only;                  //图形元素唯一的识别号
public:
    CDraw(){}                       //构造函数
```

```
    CDraw(short ColorPen,short ColorBrush,short LineWide,
        short LineType,short Layer,int id_only,BOOL Delete) //构造函数
    {
        m_ColorPen = ColorPen;
        m_ColorBrush = ColorBrush;
        m_LineWide = LineWide;
        m_LineType = LineType;
        m_Layer = Layer;
        b_Delete = Delete;
        m_id_only = id_only;
    }
};
```

在这个图形元素基类 CDraw 中，有二个构造函数，第一个构造函数不带参数，第二个构造函数带七个参数，用来初始化类中的成员变量。

在 CDraw 类中，有一个成员变量 m_id_only，这个成员变量对于实现 GIS 系统是至关重要的。通过这个成员变量来给每个图形元素以唯一的识别号，对于不同的图形元素对象，这个成员变量互不相同。这有点类似于学生的学号，在一个学校中，通过一个学号可以唯一的确定一个学生，而对于图形元素来说，通过一个识别号可以唯一地确定一个图形元素。识别号的唯一性是在一定范围内的，可以采用不同的范围标准：一种方法是对于一个矢量图形系统中的所有图形元素（包括直线、圆、连续直线、圆弧、标注文本等），其识别号各不相同，所以只通过识别号就能够唯一地确定一个图形元素；另外一种方法是，同一类别的所有图形元素，其识别号各不相同，例如，一条直线具有与其他所有直线不同的识别号，但可以与其他类别的图形元素（如圆）具有相同的识别号。在后一种情况下，要唯一的确定一个图形元素，则需要两个参数：图形类别和图形元素的识别号。在以下应用程序 Draw 的设计中，采用的是后一种方法，即识别号在本图形类别内具有唯一性，一个图形元素通过图形类别和识别号二个参数来唯一的确定。

5.1.2　直线类组织

直线类从图形元素基类 CDraw 派生而来，直线的一些基本参数（如线型、线宽、颜色、删除标志等）从 CDraw 类中继承而来，而对于直线的特殊性质（如直线的起点和终点坐标）则在直线类中定义。在头文件 drawdoc.h 中（在 CDraw 类定义后，CDrawDoc 类定义前）定义直线类 CLine：

```
class CLine:public CDraw                    //直线类
{
protected:
    float m_X1,m_X2,m_Y1,m_Y2;   //直线的起点和终点坐标
public:
    CLine(){}                               //不带任何参数的构造函数
    //以下是有初始化参数的构造函数
    CLine(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer
        ,int id_only,BOOL Delete,float X1,float Y1,float X2,float Y2)
        :CDraw(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,Delete)
```

```
    {
        m_X1 = X1;
        m_Y1 = Y1;
        m_X2 = X2;
        m_Y2 = Y2;
    }
};
```

在这个直线类 CLine 中，也重载了二个构造函数，第一个构造函数没有参数，所以可以用不带参数的方式定义一个 CLine 对象；另一个构造函数有十一个参数，并利用前七个参数调用基类 CDraw 的构造函数，对基类 CDraw 中的成员变量进行了初始化，在本身的构造函数中对直线类 CLine 自身的四个参数(直线的起点和终点)进行初始化。

5.1.3　连续直线或封闭多边形类组织

连续直线除了具有图形元素基类 CDraw 所具有的属性外，从图形的几何特征上，连续直线是由很多顶点组成的，顶点的数目是不确定的，对一条连续直线来说可能只有二个顶点，也可能有几千个顶点。在头文件 drawdoc.h 中，定义一个如下的结构来存储连续直线的一个顶点坐标：

```
typedef struct
{
    float x;                    //顶点的横坐标
    float y;                    //顶点的纵坐标
    float z;
}PointStruct;                   //存储每个顶点坐标的结构
```

为了节省存储空间，对于连续直线的顶点坐标，采用动态分配存储空间的方法，即根据连续直线的顶点数目，在 CPline 对象中分配连续直线的存储空间。在头文件 drawdoc.h 中，定义连续直线或多边形区域类 CPline 如下：

```
class CPline:public CDraw          //连续直线或多边形区域类
{
protected:
    int m_Numble;                  //连续直线或多边形区域的顶点数目
    BOOL b_Fill;                   //表示是否填充，1-多边形区域，0-连续直线
    PointStruct * m_PointList;     //存储顶点坐标的数组指针
public:
    CPline()                       //不带任何参数的构造函数
    { m_Numble = 0;}
    CPline(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
        int id_only,BOOL Delete,int Numble,PointStruct * PointList,BOOL Fill)
        :CDraw(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,Delete)
    {
        m_Numble = Numble;
```

```
        b_Fill = Fill;
        m_PointList = new PointStruct[Numble + 1];          //动态分配存储顶点坐标的结构数组
        if(Numble > 0)          //如果顶点数目大于0
        {
            for(int i = 0;i < Numble;i++)
                m_PointList[i] = PointList[i];
        }
    }
    ~CPline()                   //析构函数
    {
        if(m_Numble > 0)        //如果顶点数目大于0,删除动态分配的结构数组
            delete m_PointList;
    }
};
```

从CPline类的定义中可以看出,连续直线(或多边形区域)的顶点坐标存储在一个结构数组m_PointList中,一个顶点坐标由一个PointStruct结构来存储,连续直线的顶点数目为m_Numble。CPline类中有二个构造函数,第一个构造函数没有参数,第二个构造函数包含多个参数,并利用前七个参数调用基类CDraw的构造函数,对基类CDraw的成员变量进行了初始化。在第二个构造函数内,对顶点数大于0的连续直线(或多边形区域)动态分配了存储顶点坐标的结构数组m_PointList。在CPline类中还定义了一个成员变量b_Fill,用来表示图形元素是连续直线还是多边形区域,b_Fill为TRUE时,这个图形元素是多边形区域,b_Fill为FALSE时,这个图形元素是连续直线。

5.1.4　圆类组织

对圆类的定义方法与直线类CPline类似。圆除了具有图形元素的基本特征外,还有自身的集合特征,可以用圆心和半径作为特征参数表示圆的几何特征(当然也可以采用其他的方法,如边界矩形、圆心直径法等)。在头文件drawdoc.h中定义圆类CCircle如下:

```
class CCircle:public CDraw          //圆及圆形区域类
{
protected:
    float m_CircleX,m_CircleY,m_CircleR;    //记录圆心坐标及半径的变量
    BOOL b_Fill;                    //表示是否填充,1-圆形区域,0-普通圆
public:
    CCircle()                       //不带任何参数的构造函数
    {}
    CCircle(short ColorPen,short ColorBrush, short LineWide,short LineType,short Layer,
        int id_only,BOOL Delete,float CircleX,float CircleY,float CircleR, BOOL Fill)
        :CDraw(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,Delete)
    {
        m_CircleX = CircleX;
```

```
        m_CircleY = CircleY;
        m_CircleR = CircleR;
        b_Fill = Fill;
    }
};
```

在圆类 CCircle 中定义了三个成员变量 m_CircleX,m_CircleY,m_CircleR 来记录圆心和半径,还定义了一个成员变量参数 b_Fill,用来标识图形元素是圆还是圆形区域,当 b_Fill 为 TRUE 时是圆形区域,否则是普通圆。

在 CCircle 类中有二个构造函数,第一个构造函数没有参数,第二个构造函数具有多个参数,并调用了基类 Draw 的构造函数对基类 CDraw 的成员变量进行初始化。

在以上定义的圆类 CCircle 中,用圆心半径法来表示一个圆的几何特征。在实际的图形系统设计中,可以采用具有更广泛意义的边界矩形方法来表示一个椭圆(圆是椭圆的一个特例)的集合特征。

5.1.5　圆弧类组织

圆是圆弧的特例,当一个圆弧的弧度等于 2π 时就是一个圆。所以可以从圆类派生而得到一个圆弧类。在头文件 drawdoc.h 中,定义圆弧类 CArc 如下:

```
class CArc:public CCircle                    //圆弧类
{
protected:
    float m_Angle1,m_Angle2;                 //圆弧的起点和终点角度(用弧度表示)
public:
    CArc()//不带任何参数的构造函数
    {}
    CArc(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
        int id_only,BOOL Delete,float CircleX,float CircleY,float CircleR,BOOL Fill,
        float Angle1,float Angle2)
        :CCircle(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,Delete,CircleX,
            CircleY,CircleR,Fill)
    {
        m_Angle1 = Angle1;
        m_Angle2 = Angle2;
    }
};
```

圆弧类 CArc 是从圆类 CCircle 派生而来。在 CArc 类中定义了两个变量 m_Angle1 和 m_Angle2,分别来表示一个圆弧的起始和结束弧度。CArc 类有二个构造函数,第一个构造函数没有参数,第二个构造函数有多个参数,并利用多个参数调用了基类 CCircle 的构造函数对圆类 CCircle 和图形元素基类 CDraw 的成员变量进行初始化。

5.1.6　标注文本类

图形中可以有大量的标注文本，标注文本除了具有图形元素的基本特征外，还具有位置、字体及标注内容等自身的信息。在头文件 drawdoc.h 中，定义标注文本类 CText 如下：

```
class CText:public CDraw                    //标注文本类
{
protected:
    float m_StartX;                         //标注文本起点横坐标
    float m_StartY;                         //标注文本起点纵坐标
    float m_Angle1;                         //标注角度
    float m_Angle2;                         //字体旋转角度
    float m_TextHeight;                     //字体高度
    float m_TextWide;                       //字体宽度
    float m_OffWide;                        //字体标注间隔
    unsigned char m_TextFont;               //字体
    CString c_Text;                         //标注文本内容
    int m_TextLong;                         //标注文本的字符长度
public:
    CText()                                 //不带任何参数的构造函数
    {}
    CText(short ColorPen,short ColorBrush,short LineWide,short LineType, short Layer,
        int id_only,BOOL Delete,float StartX, float StartY,float Angle1, float Angle2,float TextHeight,
        float TextWide,float OffWide,unsigned char TextFont,CString Text)
        :CDraw(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,Delete)
    {
        m_StartX = StartX;
        m_StartY = StartY;
        m_Angle1 = Angle1;
        m_Angle2 = Angle2;
        m_TextHeight = TextHeight;
        m_TextWide = TextWide;
        m_OffWide = OffWide;
        m_TextLong = Text.GetLength();    //计算字符的长度
        c_Text = Text;
    }
};
```

CText 类也有二个构造函数，第二个构造函数有多个参数，对类的成员变量进行初始化，在构造函数中，调用基类 CDraw 的构造函数对基类 CDarw 的成员变量进行初始化。

5.1.7　图形参数类

在以上定义的图形元素基类 CDraw 中，成员变量 m_ColorPen 等存储的是在颜色列表的序

列号，m_ Layer 中存储的也是在图层列表中的序列号。现在建立一个类来管理诸如颜色、图层等方面的信息。在头文件 drawdoc.h 中，定义图形参数类 CGraphPara 如下：

```
class CGraphPara                    //用来存储图形参数的类
{
protected:
    int n_ColorNumbAll;             //总的颜色数
    int n_LayerNumbAll;             //总的图层数
    int n_ColorNumb;                //系统当前具有的颜色数
    int n_LayerNumb;                //系统当前具有的图层数
    long * m_ColorList;             //用来存储颜色列表的数组
    LayerStruct * m_LayerList;      //用来存储图层列表的结构数组
public:
    CGraphPara()
    {
        n_ColorNumb = 100;          //最多具有 100 种颜色
        n_LayerNumb = 100;          //最多具有 100 图层
        m_ColorList = new long[n_ColorNumb];
        m_LayerList = new LayerStruct[n_LayerNumb];
        n_ColorNumb = 4;            //当前有四种颜色
        n_LayerNumb = 1;            //当前有一个图层
        //以下初始化颜色和图层
        m_ColorList[0] = RGB(0,0,0);
        m_ColorList[1] = RGB(255,0,0);
        m_ColorList[2] = RGB(0,255,0);
        m_ColorList[3] = RGB(0,0,255);
        m_LayerList[0].b_Display = 1;
        strcpy(m_LayerList[0].m_Name,"Layer 0");
    }
    ~CGraphPara()                   //析构函数
    {
        delete m_LayerList;
        delete m_ColorList;
    }
public:
    COLORREF GetColor(int n);       //得到第 n 种颜色的实际颜色
    BOOL GetDisplayStatue(int n);   //得到第 n 层的显示状态
};
```

在 Windows 中，颜色是由 RGB(r_Color,g_Color,b_Color)宏来表示的，参数 r_Color、g_Color、b_Color 分别是一个颜色中红、绿、蓝三种颜色占的值，取值范围为 0～255。颜色也可以由一个 long 型数值来表示：

```
long color = r_Color + g_Color * 256 + b_Color * 256 * 256;
```

在图形参数类 CGraphPara 中，定义一个指向 long 型数组 m _ ColorList 来管理颜色列表。

在应用程序 Draw 中，每个图层的信息用数据结构 LayerStruct 来表示，在头文件 drawdoc.h 中，在 CGraphPara 类前定义 LayerStruct 结构：

```
typedef struct                  //用来存储图层的信息的结构
{
    char m _ Name[21];          //图层的名称
    BOOL b _ Display;           //是否显示，1－显示，0－隐藏
    int reserved;               //备用
}LayerStruct;
```

在 CGraphPara 类中，定义 LayerStruct 结构数组 m _ LayerList，用来管理图层列表。另外，在 CGraphPara 类中，定义了两个成员函数 GetColor 和 GetDisplayStatur，分别用来得到颜色列表的第 n 个颜色的颜色值和图层列表的第 n 个图层的显示状态。在实现文件 drawdoc.cpp 中加入这两个函数的实现代码如下：

```
COLORREF CGraphPara::GetColor(int n)
{
    return m _ ColorList[n];                //返回第 n 种颜色值
}

BOOL CGraphPara::GetDisplayStatue(int n)
{
    return m _ LayerList[n].b _ Display;    //返回第 n 个图层的显示状态
}
```

本节介绍了直线、连续直线或多边形区域、圆、圆弧、标注文本等几种最基本和有代表性的图形元素类的组织，对于其他类型的图形元素，如 AutoCAD 中的 pline 式的连续线等，读者可根据以上的思路独立组织。

以上组织实现了开发基本矢量图形系统所需要的各个图形元素类，各个类中除了构造函数外，几乎没有其他的成员函数。在以后的章节中，将在开发过程中逐渐增加类的成员函数，开发实现一个基本矢量图形系统。

5.2　组织矢量图形系统的文档

在 5.1 节，组织实现了一个基本矢量图形系统的图形元素类。本节中将使用这些类，利用 MFC 应用程序的文档管理体系组织文档，实现基本矢量图形系统的文档管理功能。

5.2.1　组织面向对象的文档存储管理机制

本小节，讨论管理矢量图形系统图形数据的管理方法，即文档管理机制。本小节将讨论一套完全的面向对象的文档组织机制，通过图形元素类创建很多图形元素对象（如可以利用 CLine 类创建 20000CLine 对象），每个图形元素对象作为一个整体来组织存储空间的分配、保存及读取等各种管理功能。通过建立一种存储机制，来管理指向所有图形元素对象的指针，达到

管理所有图形元素对象的目的。这种文档管理机制具有组织简单、结构化和移植性好,以及比较容易利用 VC++ 程序设计语言的新技术等优点。缺点是需要较大的内存空间。

5.2.2 利用 MFC 模板定义管理图形元素对象指针的对象

本小节来讨论文档管理机制的具体实现。管理一个矢量图形系统文档的思路是:每个图形元素是图形元素类创建的一个对象,在创建这个对象时得到指向这个对象的指针,通过建立一个对象指针数组来管理这些指针,来达到管理所有图形元素对象的目的。

在 VC++ 下可以较容易地实现对指向图形元素对象的指针的组织和管理。在 MFC 类中有一个类模板 CTypedPtrArray,可以用它来定义一个管理类指针的对象。例如,可以定义一个管理 CLine 类指针的对象如下:

```
CTypedPtrArray < CObArray, CLine * > m_LineArray;
```

CTypedPtrArray 定义的对象 m_LineArray,由第一个参数指定的类(CObArray)创建且管理第二个参数指定的类指针。*所以,上述代码的含义是:对象 m_LineArray 是由 CObArray 类创建,*用来存放 CLine 类指针。

CObArray 类是 MFC 的通用类之一,可以用来存放变量或对象指针。CObArray 对象用阵列状的数据结构存放一组指向 CObject 对象(或 CObject 派生类的对象)的指针(CObject 直接或间接派生几乎所有的其他 MFC 类,在应用程序 Draw 中,把图形元素基类 CDraw 定义为 CObject 派生类,是为了用 CObArray 对象管理其派生类对象的指针)。它管理对象的方法可以理解为一个指针数组,不同的是在 CObArray 中封装了完成各种数组操作功能的函数(如增加、删除、插入、得到数组项),可以非常容易的完成各种操作功能。当然,也可以直接利用 CObArray 类创建一个对象来管理 CLine 对象指针,但用 CTypePtrArray 模板创建 CObArray 对象的好处是:这种对象使编译器进行更广泛的类型检查,减少错误,并减少使用对象指针时需要的类型转换。采用 CObArray 类直接创建的对象管理 CLine 对象指针时,需要进行类型转换,如在对象指针前需加上强制转换符(CLine *),而用 CtypePtrArray 模板创建的 CObArray 对象管理 CLine 对象指针时,因为在创建过程中已经规定为管理 CLine 对象指针,所以不需要进行类型转换。

5.2.3 实现矢量图形系统的文档

上小节讨论了用类模板创建的 CObArray 对象管理 CLine 对象指针的方法。在应用程序 Draw 中,可以派生几个 CObArray 对象来分别管理几类图形元素。在文档类 CDrawDoc 中创建几个管理各类图形元素对象指针的 CObArray 对象如下:

```
private:
  CTypedPtrArray < CObArray, CLine * > m_LineArray;        //管理直线对象指针
  CTypedPtrArray < CObArray, CPline * > m_PLineArray;      //管理连续直线或多边形区域对象指针
  CTypedPtrArray < CObArray, CCircle * > m_CircleArray;    //管理圆对象指针
  CTypedPtrArray < CObArray, CArc * > m_ArcArray;          //管理圆弧对象指针
  CTypedPtrArray < CObArray, CText * > m_TextArray;        //管理标注文本对象指针
```

为了管理一个矢量图形的图形参数,在文档类 CDrawDoc 中定义一个 CGraphPara 对象:

```
CGraphPara m_GraphPara;          //定义管理一个图形参数的对象
```

为了在各个图形元素类以及一些对话框类使用这个图形参数对象 m_GraphPara，可以定义一个全局的 CGraphPara 类指针，并把这个指针指向当前文档对象中的 CGraphPara 对象 m_GraphPara。在实现文件 drawdoc.cpp 中，定义一个全局 CGraphPara 类指针（所谓全局定义，就是把变量定义在所有函数的外面）：

```
CGraphPara * p_GraphPara;
```

为了把这个指针指向当前文档对象中的 CGraphPara 对象 m_GraphPara，在文档类 CDrawDoc 的构造函数中加入如下代码：

```
CDrawDoc::CDrawDoc()
{
    p_GraphPara = &m_GraphPara; //将全局 GraphPara 类指针指向当前文档对象的 GraphPara 对象
}
```

如果要在其他别的实现文件（如 drawview.cpp 文件）中使用这个对象指针，只需在其他实现文件中加入这个全局指针的外部引用：

```
extern CGraphPara * p_GraphPara;
```

5.2.4　实现文档的管理功能

本小节中，讨论实现对矢量图形系统中的几类图形元素的管理功能。

（1）增加图形元素

在应用程序 Draw 中增加一个图形元素对象（如一条直线）时，需要进行以下两个步骤的操作：

① 需要创建一个图形元素对象，并用图形元素的实际数据初始化这个图形元素对象。如增加一条直线时，需要创建一个 CLine 对象，并用这条直线的实际数据（起终点、所在图层、颜色等）初始化这个 CLine 对象的成员变量。

② 需要把指向新创建的图形元素对象的指针，增加到文档类中管理图形元素对象指针的对象中。如增加一条 CLine 对象后，需要把指向这个 CLine 对象的指针增加到文档类的 m_LineArray 对象中。

为了实现增加各类图形元素的功能，在文档类 CDrawDoc 中定义几个函数，分别来完成增加各类图形元素的操作功能：

```
public:
CLine * AddLine(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
    int id_only,float X1,float Y1,float X2,float Y2);
CPline * AddPLine(short ColorPen,short ColorBrush, short LineWide,short LineType,
    short Layer,int id_only,int Numble,PointStruct * PointList,BOOL b_Fill);
CCircle * AddCircle(short ColorPen,short ColorBrush,short LineWide,short LineType,
    short Layer,int id_only,float CircleX,float CircleY,float CircleR,BOOL Fill);
```

```
CArc * AddArc(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
    int id_only,float CircleX,float CircleY,float CircleR,BOOL Fill,float Angle1,
    float Angle2);
CText * AddText(short ColorPen,short ColorBrush,short LineWide,short LineType, short Layer,
    int id_only,float StartX,float StartY,float Angle1,float Angle2,float TextHeight,
    float TextWide,float OffWide,unsigned char TextFont,int TextLong,CString);
```

这五个函数分别用来实现增加一条直线、一条连续直线或封闭多边形区域、一个圆或圆形区域、一个圆弧、一个标注文本的功能。函数返回指向新增图形元素对象的指针。在实现文件drawdoc.cpp中加入这五个函数的实现代码：

```
//增加一个直线对象
CLine * CDrawDoc::AddLine(short ColorPen,short ColorBrush,short LineWide, short LineType,
    short Layer,int id_only,float X1,float Y1,float X2,float Y2)
{
    CLine * p_Line = new CLine(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,
        0,X1,Y1,X2,Y2);          //动态创建并初始化一个 CLine 对象
    m_LineArray.Add(p_Line);     //将指向新建 CLine 对象的指针增加到 m_LineArray 对象中
    return p_Line;               //返回指向新增 CLine 对象的指针
}

//增加一个连续直线或多边形区域对象
CPline * CDrawDoc::AddPLine(short ColorPen,short ColorBrush,short LineWide,short
    LineType,short Layer,int id_only,int Numble,PointStruct * PointList,BOOL Fill)
{
    CPline * p_Pline = new CPline(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,
        0,Numble,PointList,Fill);  //动态创建并初始化一个 CPline 对象
    m_PLineArray.Add(p_Pline);   //将指向新建 CPline 对象的指针增加到 m_PLineArray 对象中
    return p_Pline;              //返回指向新增 CPline 对象的指针
}

//增加一个圆或圆形区域对象
CCircle * CDrawDoc::AddCircle(short ColorPen,short ColorBrush,short LineWide,short
    LineType,short Layer,int id_only,float CircleX,float CircleY,float CircleR,BOOL Fill)
{
    CCircle * p_Circle = new CCircle(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,
        0,CircleX,CircleY,CircleR,Fill);  //动态创建并初始化一个 CCircle 对象
    m_CircleArray.Add(p_Circle);         //将指向新建圆对象的指针增加到 m_CircleArray 对象中
    return p_Circle;                     //返回指向新增圆对象的指针
}

//增加一个圆弧类 CArc 对象
CArc * CDrawDoc::AddArc(short ColorPen,short ColorBrush,short LineWide,short LineType,
    short Layer,int id_only,float CircleX,float CircleY,float CircleR,BOOL Fill,
```

```
    float Angle1,float Angle2)
{
    CArc * p_Arc = new CArc(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,0,
        CircleX,CircleY,CircleR,Fill,Angle1,Angle2);   //动态创建并初始化一个 CArc 对象
    m_ArcArray.Add(p_Arc);   //将指向新增圆弧对象的指针增加到 m_ArcArray 对象中
    return p_Arc;            //返回指向新增圆弧对象的指针
}

//增加一个标注文本类 CText 对象
CText * CDrawDoc::AddText(short ColorPen,short ColorBrush,short LineWide, short LineType,
    short Layer,int id_only,float StartX,float StartY,float Angle1,float Angle2,
    float TextHeight,float TextWide,float OffWide,unsigned char TextFont,int TextLong,
    CString Text)
{
    //以下动态创建并初始化一个 CText 对象
    CText * p_Text = new CText(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,0,
        StartX,StartY,Angle1,Angle2,TextHeight,TextWide,OffWide, 0,Text);
    m_TextArray.Add(p_Text);  //将指向新赠标注文本对象的指针增加到 m_TextArray 对象中
    return p_Text;            //返回指向新增标注文本对象的指针
}
```

(2) 得到指向图形元素对象的指针

现在介绍获得图形元素对象指针的方法,在一个图形系统中,有大量的图形元素。指向图形元素对象的指针被保存在文档类对应的 CObArray 对象中。在系统的操作过程中,为了对图形元素实现各种操作功能,需要得到指向指定图形元素对象的指针。为了获得指向指定图形元素对象的指针,在文档类 CDrawDoc 中定义一个函数如下:

```
public:
    CDraw * GetGraph(short Lb,int Index);
```

函数 GetGraph 用来获得指向某个图形元素对象的 CDraw 类指针。参数 Lb 用来表示要返回的图形类别:1—直线,2—连续直线或多边形区域,3—圆或圆形区域,4—圆弧,5—标注文本。参数 Index 是要得到的图形元素对象指针在相应 CObArray 对象中的 0 基序号(数组的下标)。在实现文件 drawdoc.cpp 中加入函数的实现代码:

```
CDraw * CDrawDoc::GetGraph(short Lb,int Index)
{
    switch(Lb)          //根据图形类别,进行不同的操作
    {
    case 1:             //如果是直线
        if(Index < 0||Index > m_LineArray.GetUpperBound())   //如要读取的项不在数组范围内
            return 0;  //函数返回空
        return m_LineArray.GetAt(Index);   //返回 m_LineArray 对象中存储在第 Index 项的指针
```

```
            break;
        case 2:              //如果是连续直线或多边形区域
            if(Index < 0||Index > m_PLineArray.GetUpperBound())   //如要读取的项不在数组范围内
                return 0;
            return m_PLineArray.GetAt(Index);  //返回 m_PLineArray 对象中存储在第 Index 项的指针
            break;
        case 3:              //如果是圆或圆形区域
            if(Index < 0||Index > m_CircleArray.GetUpperBound())  //如要读取的项不在数组范围内
                return 0;
            return m_CircleArray.GetAt(Index);  //返回 m_CircleArray 对象存储在第 Index 项的指针
            break;
        case 4:              //如果是圆弧
            if(Index < 0||Index > m_ArcArray.GetUpperBound())   //如要读取的项不在数组范围内
                return 0;
            return m_ArcArray.GetAt(Index);   //返回 m_ArcArray 对象中存储在第 Index 项的指针
            break;
        case 5:              //如果是标注文本
            if(Index < 0||Index > m_TextArray.GetUpperBound())   //如要读取的项不在数组范围内
                return 0;
            return m_TextArray.GetAt(Index);   //返回 m_TextArray 对象中存储在第 Index 项的指针
            break;
        default:
            return 0;
        }
    }
```

CObArray 类的成员函数 GetUpperBound 用来获得 CObArray 对象中存储的最大项目数，也就是数组的最大下标。GetAt 函数用来得到 CObArray 对象中某一项的内容，即获得数组中某一项的内容。为了得到存储在 CObArray 对象中第 Index 项的指针，在 GetGraph 函数中首先检查要返回的项 Index 是否合法(在 CObArray 对象所存储的指针数组的范围内)，然后返回存储在 CObArray 对象中第 Index 项的指针。

GetGraph 函数返回的是 CDraw 类指针，这个指针指向指定的图形元素对象，通过 CDraw 类指针调用 CDraw 类的成员函数实现各个操作功能时，利用虚函数以及由此引发的多态性，来调用实际图形元素类中的虚函数，来实现针对图形元素类的操作功能。

(3) 删除图形元素对象

现在介绍从文档类的 CObArray 对象中删除图形元素对象指针的操作功能。为了能够从 CObArray 对象中删除一个指定图形元素对象指针，在文档类 CDrawDoc 中定义一个成员函数：

```
public:
    void DeleteGraph(short Lb,int Index);
```

DeleteGraph 函数中各个参数的含义与 GetGraph 函数相同。在实现文件 drawdoc.cpp 中加

入 DeleteGraph 函数的实现代码：

```
void CDrawDoc::DeleteGraph(short Lb,int Index)
{
    switch(Lb)        //根据图形类别,进行不同的操作
    {
    case 1:           //如果是直线
        if(Index<0||Index>m_LineArray.GetUpperBound())  //如要删除的项不在数组范围内
            return;                                     //函数返回
        return m_LineArray.RemoveAt(Index);     //删除 m_LineArray 对象的第 Index 个数组项
        break;
    case 2:           //如果是连续直线或多边形区域
        if(Index<0||Index>m_PLineArray.GetUpperBound())  //如要删除的项不在数组范围内
            return;
        return m_PLineArray.RemoveAt(Index);  //删除 m_PLineArray 对象的第 Index 个数组项
        break;
    case 3:           //如果是圆或圆形区域
        if(Index<0||Index>m_CircleArray.GetUpperBound())  //如要删除的项不在数组范围内
            return;
        return m_CircleArray.RemoveAt(Index);  //删除 m_CircleArray 对象的第 Index 个数组项
        break;
    case 4:           //如果是圆弧
        if(Index<0||Index>m_ArcArray.GetUpperBound())  //如要删除的项不在数组范围内
            return;
        return m_ArcArray.RemoveAt(Index);     //删除 m_ArcArray 对象的第 Index 个数组项
        break;
    case 5:           //如果是标注文本
        if(Index<0||Index>m_TextArray.GetUpperBound())  //如要删除的项不在数组范围内
            return;
        return m_TextArray.RemoveAt(Index);     //删除 m_TextArray 对象的第 Index 个数组项
        break;
    default:
        return;
    }
}
```

CObArray 类的成员函数 RemoveAt 函数，能够删除 CObArray 对象中指定序号的数组项。DeleteGraph 函数根据 Lb 参数指定的图形类别，去删除 Index 参数所指定的数组项，达到删除指定的图形元素对象指针的目的。

(4) 得到各类图形元素的数目

为了随时能够得到文档中各种图形元素对象的数目，在文档类 CDrawDoc 中，定义一个成员函数如下：

```
public:
    int GetGraphNumb(short Lb);
```

在实现文件 drawdoc.cpp 中，加入这个函数的实现代码：

```
int CDrawDoc::GetGraphNumb(short Lb)
{
    switch(Lb)          //根据图形类别,获得各种图形元素的数目
    {
    case 1:             //如果是直线
        return m_LineArray.GetSize();      //返回存储在 m_LineArray 对象中的指针数目
        break;
    case 2:             //如果是连续直线或多边形区域
        return m_PLineArray.GetSize();     //返回存储在 m_PLineArray 对象中的指针数目
        break;
    case 3:             //如果是圆或圆形区域
        return m_CircleArray.GetSize();    //返回存储在 m_CircleArray 对象中的指针数目
        break;
    case 4:             //如果是圆弧
        return m_ArcArray.GetSize();       //返回存储在 m_ArcArray 对象中的指针数目
        break;
    case 5:             //如果是标注文本
        return m_TextArray.GetSize();      //返回存储在 m_TextArray 对象中的指针数目
        break;
    default:
        return 0;
    }
}
```

在 GetGraphNumb 函数中，根据不同的图形元素类别，分别用不同的 CObArray 对象调用函数 GetSize，得到存储在 CObArray 对象中的指针数目，即得到各类图形元素的数目。

(5) 得到存储各类图形元素 CObArray 对象数组的上标

在 CObArray 类中，函数 GetSize 与函数 GetUpperBound 返回不同的结果。GetSize 函数得到的是 CObArray 对象中存储的数组项数，GetUpperBound 得到的是 CObArray 对象中最大数组下标。例如，在 m_LineArray 对象中增加 10 个指向 CLine 对象的指针时，调用 GetUpperBound 函数得到的结果为 9，调用 GetSize 函数得到的结果为 10。如果调用 CObArray 类的函数 RemoveAt 删除 m_LineArray 对象中第 2、3 项，再调用 GetUpperBound 函数得到的结果还是 9，而调用 GetSize 函数时得到的结果为 8。

在对各个图形元素类的所有图形元素进行操作时，需要得到存储在各个 CObArray 对象中的最大数组下标，然后对各个数组项实现遍历操作。为了得到存储在各个 CObArray 对象中的最大数组下标，在文档类 CDrawDoc 中定义一个函数如下：

```
public:
```

```
int GetGraphUpperBound(short Lb);
```

GetGraphUpperBound 函数的参数 Lb 用来表示图形元素类别，函数返回与图形类别对应的 CObArray 对象中最大数组下标的值。在实现文件 drawdoc.cpp 中加入函数的实现代码：

```
int CDrawDoc::GetGraphUpperBound(short Lb)
{
    switch(Lb)   //根据不同的图形类别,进行不同的操作
    {
    case 1:
        return m_LineArray.GetUpperBound();
        break;
    case 2:
        return m_PLineArray.GetUpperBound();
        break;
    case 3:
        return m_CircleArray.GetUpperBound();
        break;
    case 4:
        return m_ArcArray.GetUpperBound();
        break;
    case 5:
        return m_TextArray.GetUpperBound();
        break;
    }
    return -1;
}
```

函数 GetGraphUpperBound 的代码含义与函数 GetGraphNumb 类似。

5.3　实现矢量图形系统的视图

本节将在 5.2 节实现的文档管理功能基础上，实现矢量图形系统的视图，完成矢量图形系统的图形元素绘制功能。

5.3.1　建立坐标系

在组织矢量图形系统的视图，即组织矢量图形系统的图形绘画功能前，首先要确定和实现矢量图形系统所采用的坐标系。本小节，将为矢量图形系统组织实现一个适用于各种坐标映像方式的坐标系。

(1) 建立坐标系

在 4.3 节中详细讨论了 Windows 的映像方式。在组织一幅图形时，采用哪一种映像方式

组织矢量图形系统的坐标系值得进行研究。在进行特定的图形系统开发时,可以选择一个固定的坐标映像方式,但是在组织矢量图形系统的开发时,因为矢量图形是无级放缩的,大多数映像方式的设备无关性就没有实际的意义。在应用程序 Draw 中,图形元素的的坐标是用浮点数(也可把图形元素类的成员变量改成双精度型,以提高设计精度)来表示的,系统应该能够满足各种数量级图形数据的需要,所以直接采用一种固定坐标单位的映像方式不能达到此操作目的。

在矢量图形系统 Draw 的开发中,可以建立一个如图 5-1 所示的坐标系,然后建立这个坐标系与 Windows 各个坐标映像方式间的转换关系。在这个坐标系中,取图形屏幕的左下角点作为图形屏幕的基点,给定这个点的实际坐标为(m_xStart,m_yStart),并确定一个变量 blc,作为逻辑坐标和实际坐标的比例关系。

图 5-1 实际坐标系

(2) 实际坐标与逻辑坐标的转换

变量 blc 作为各映像模式下的逻辑坐标与实际坐标的比例关系。blc 记录的是一个逻辑单位所代表的实际坐标数值,例如在 MM_TEXT 映像模式下,blc 等于 10 的含义是视图中的一个像素长度等于 10 个实际坐标单位。在 MM_LOMETRIC 映像模式下,blc 等于 10 的含义是一个逻辑单位(即 0.1mm)等于 10 个实际坐标单位。通过建立实际坐标与逻辑坐标的这种转换关系的函数,实现逻辑坐标与实际坐标间的转换。

4.3 节讨论了 Windows 的各种映像方式,前六种映像方式都可以用来组织图形系统使用的逻辑坐标系。这六种坐标系的坐标原点都在客户区窗口的左上角,除 MM_TEXT 映像模式外,坐标系 Y 轴方向都与实际坐标系的 Y 轴方向相同。MM_TEXT 映像方式下,坐标原点在窗口区域的左上角,向右为 x 轴正方向,向下为 y 轴方向(如图 5-2),坐标系单位为像素。为了实现实际坐标与逻辑坐标的转换,在视图类 CDrawView 中定义如下的两个转换函数:

```
public:
    void DPtoVP(float x,float y,int * X,int * Y);
    void VPtoDP(int x,int y,float * X,float * Y);
```

函数 VPtoDP 用来将逻辑坐标转换成实际坐标,参数(x,y)是一个点的逻辑坐标,参数(*X,*Y)是返回转换后的实际坐标值。函数的实现代码如下:

图 5-2　MM _ TEXT 映像方式下的坐标系

```
void CDrawView::VPtoDP(int x,int y,float * X,float * Y)
{
    * X = m_xStart + x * blc;      //得到点的实际坐标横坐标
    if(m_MapMode = = 1)            //如果是 MM_TEXT 映像方式
        * Y = m_yStart + blc * (m_hScreen - y);  //得到点的实际纵横坐标
    else                           //如果是其他的映像方式
        * Y = m_yStart + blc * (y + m_hScreen);  //得到点的实际纵坐标
}
```

因为在 MM _ TEXT 映像模式下，Y 轴方向与实际坐标系的 Y 轴方向相反，所以在这种映像方式下进行坐标转换的方式与其他五种映像方式不同。以下代码可以将逻辑坐标(screenx，screeny)转换成图形系统的实际坐标(X,Y)：

```
VPtoDP(screenx,screeny,&X,&Y);
```

函数 DPtoVP 用来将实际坐标转换成各种映像方式下的逻辑坐标，参数(x,y)是一个点的实际坐标，参数(* X, * Y)返回转换后的逻辑坐标值。函数的实现代码如下：

```
void CDrawView::DPtoVP(float x,float y,int * X,int * Y)
{
    * X = (int)((x - m_xStart)/blc);                    //得到点的逻辑横坐标
    if(m_MapMode = = 1)                                 //如果是 MM_TEXT 映像方式
        * Y = m_hScreen - (int)((y - m_yStart)/blc);    //得到点的逻辑纵坐标
    else                                                //如果是其他坐标映像方式
        * Y = (int)((y - m_yStart)/blc) - m_hScreen;    //得到点的逻辑纵坐标
}
```

以下代码可以将实际坐标(X,Y)转换成逻辑坐标(screenx，screeny)：

```
DptoVP(X,Y,&screenx,&screeny);
```

两个转换函数中用到的成员变量在视图类 CDrawView 中定义：

```
private:
```

```
    float m_xStart,m_yStart;blc;      //存储屏幕左下角的实际坐标和转换比例尺
int m_wScreen,m_hScreen;              //存储视图屏幕的实际宽度和高度(逻辑坐标)
```

在实现文件 drawview.cpp 中,在 CDrawView 类的构造函数中,加入成员变量的初始化代码:

```
CDrawView::CDrawView()
{
    ......//省略以上代码
    m_xStart=0.0;
    m_yStart=0.0;
    blc=1.0;
}
```

成员变量 m_wScreen、m_hScreen 在消息处理函数 OnSize 中初始化,当窗口创建或窗口大小变化时,OnSize 消息处理函数被调用。用 2.5.2 小节讨论的增加消息处理函数的方法,在视图类 CDrawView 中增加一个消息处理函数 OnSize,操作完毕后,程序代码做如下修改:

① 在视图类 CDrawView 中,增加了消息处理函数的定义:

```
//{{AFX_MSG(CDrawView)
afx_msg void OnSize(UINT nType, int cx, int cy);
//}}AFX_MSG
```

② 在实现文件 drawview.cpp 中增加了函数的实现代码,以下是增加了对变量 m_hScreen 和 m_wScreen 初始化后的函数实现代码:

```
void CDrawView::OnSize(UINT nType, int cx, int cy)
{
    m_wScreen=cx;          //视图屏幕的宽度(逻辑坐标)
    m_hScreen=cy;          //视图屏幕的高度(逻辑坐标)
    CView::OnSize(nType, cx, cy);
}
```

③ 在实现文件 drawview.cpp 中增加了消息映射机制:

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    //{{AFX_MSG_MAP(CDrawView)
    ......//省略以上代码
    ON_WM_SIZE()        //OnSize 消息处理函数的映射宏
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

为了在应用程序 Draw 的其他类中能够使用两个坐标转换函数 DPtoVP 和 VPtoDP,可以定义两个全局函数来实现转换操作功能。在实现文件 drawview.cpp 中定义两个全局函数如下:

```
void DPtoVP(float x,float y,int *X,int *Y);
void VPtoDP(int x,int y,float *X,float *Y);
```

在实现文件 drawview.cpp 中加入这两个函数的实现代码:

```
void DPtoVP(float x,float y,int * X,int * Y)
{
    p_View->DPtoVP(x,y,X,Y);  //调用视图类的 DPtoVP 函数
}

void VPtoDP(int x,int y,float * X,float * Y)
{
    p_View->VPtoDP(x,y,X,Y);  //调用视图类的 VPtoDP 函数
}
```

以上两个全局函数利用视图类指针 p_View 调用视图类的两个坐标转换函数，来完成坐标的转换操作功能。如果要在某个实现文件中使用这两个全局转换函数，只需在这个实现文件的开始处加入全局函数的外部引用：

```
extern void DPtoVP(float x,float y,int * X,int * Y);
extern void VPtoDP(int x,int y,float * X,float * Y);
```

在以上的两个全局函数中，p_View 指向当前激活的视图对象。p_View 是一个在实现文件 drawview.cpp 中全局定义的视图类指针：

```
CDrawView * p_View;
```

在视图类 CDrawView 的重载虚函数 OnActivateView 中，对 p_View 指针进行初始化，以保证 p_View 指针时刻指向当前激活的视图对象：

```
void CDrawView::OnActivateView(BOOL bActivate, CView * pActivateView, CView * pDeactiveView)
{
    p_View = this;
    CView::OnActivateView(bActivate, pActivateView, pDeactiveView);
}
```

OnActivateView 函数是 CView 类的一个虚函数，可以在当前视图类 CDrawView 中重载此函数(操作方法见 2.5.5 小节)，这个函数在视图对象被激活时被调用(视图第一次显示，以及在多视图状态下，激活一个视图时，都可以激发运行 OnActivateView 函数)，就保证 p_View 指向当前活动的视图对象(如果将 p_View 指针的初始化放在视图类的构造函数中，那么只在一个视图对象被创建时 p_View 被初始化，在多于一个视图的情况下，p_View 不能保证指向当前活动的视图对象)。

(3) 确定实际坐标系的参数

在系统的运行过程中，图形的绘制区域(即图幅)可能发生变化，在这种情况下，当需要显示全幅图形时，就需要根据图幅的大小，确定图形屏幕的实际坐标参数(m_xStart, m_yStart, blc)。

下面以具体的实例，来说明确定实际坐标系参数的方法。一幅图形，假定图形的实际最小坐标为(100.0,100.0)，实际最大坐标为(20000.0,10000.0)。为了能够把这幅图形全部显示到当前窗口中，需要确定当前视图窗口左下角点的实际坐标(m_xStart,m_yStart)和显示比例

blc。以下代码能够计算得到显示比例 blc：

```
float minx,miny,maxx,maxy;
float bl1;
minx = 100.0;
miny = 100.0;
maxx = 20000.0;
maxy = 10000.0;
bl1 = (20000.0 - 100.0)/m _ wScreen - 20); //横向比例
blc = (10000.0 - 100.0)/m _ hScreen - 20); //纵向比例
if(bl1 > blc)
    blc = bl1; //取较大的一个作为比例尺
```

接下来确定绘图窗口左下角的实际坐标，将图幅的最小坐标点放在离左边界和下边界都是 10 个像素单位(在图形系统中，在屏幕上绘制图形时采用的是 MM _ TEXT 映像方式)，所以绘图窗口左下角的实际坐标为：

```
m _ xStart = 100.0 - 10 * blc;
m _ yStart = 100.0 - 10 * blc;
```

这样就确定了图形屏幕实际坐标系的参数，实际坐标(x,y)到逻辑坐标(screenx, screeny)的转换通过函数 DPtoVP 来实现：

```
DPtoVP(x,y,&screenx,&screeny);
```

反过来，逻辑坐标到实际坐标的转换通过函数 VPtoDP 来实现：

```
VPtoDP(screenx,screeny,&x,&y);
```

(4) 坐标映像模式的使用

如上所述，矢量图形在视图上显示时，因为矢量图形系统的无级放缩性，大部分映像方式的设备无关性没有实际意义，所以在视图中绘制图形元素时，只采用 MM _ TEXT 映像方式，通过实际坐标与逻辑坐标的转换来绘制图形。同时，在图形的显示操作中一直保持窗口原点和视口原点坐标都为(0,0)，窗口和视口是重叠在一起的，不存在 4.4.3 小节讨论的窗口到视口的坐标转换问题。实际坐标和逻辑坐标的转换是通过定义的转换函数来实现的。

在进行图形打印输出操作时，可以采用其他的映像方式，通过矢量图形系统中的实际坐标与逻辑坐标的转换，通过这些映像模式按照设备无关性输出符合一定尺寸的图形。

5.3.2 实现各图形元素类的绘制函数

在本小节中，将实现各类图形元素的绘制功能。

利用虚函数来实现各种图形元素的绘制功能。在图形元素基类 CDraw 中，抽象定义一个进行绘制操作的虚函数。因为在应用程序 Draw 中，不用 CDraw 类直接来定义对象，所以可以将虚函数定义成纯虚函数：

```
class CDraw:public CObject  //图形元素基类，存储图形的颜色、线型、层等信息
```

```
{
protected:
    CDraw(){}                    //构造函数
    short m_ColorPen;            //笔色
    short m_ColorBrush;          //填充刷颜色
    short m_LineWide;            //线宽(逻辑坐标)
    short m_LineType;            //线型
    short m_Layer;               //所在图层
    BOOL b_Delete;               //是否处于删除状态
public:
    CDraw(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
        int id_only,BOOL Delete)     //构造函数
    {
        m_ColorPen = ColorPen;
        m_ColorBrush = ColorBrush;
        m_LineWide = LineWide;
        m_LineType = LineType;
        m_Layer = Layer;
        m_id_only = id_only;
        b_Delete = Delete;
    }
    virtual void Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor) = 0;
}
```

Draw 函数有多个参数,各参数的含义如下:

pDC:　指向当前绘图对象的指针。

m_DrawMode:　绘制模式。1—R2_COPYPEN,0—R2_NOT。根据需要可以加入其他的模式。

m_DrawMode1:　直线绘制方式。0—正常显示,1—特殊显示(如用鼠标选中时的显示方式),2—用指定颜色绘制。根据需要可以加入其他的方式。

BackColor:　当参数 m_DrawMode1 取 2 时,此参数指定颜色号。

在各种图形元素类(直线、连续直线、圆、圆弧、标注文本等)中,重载虚函数 Draw(对于用来创建对象的类,必须有基类纯虚函数的重载):

```
public:
    virtual void Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor);
```

以下将根据各类图形元素的具体绘制方法,在各个图形元素类中加入 Draw 函数的实现代码。

(1) 直线的绘制

为了完成直线的绘制功能,在类 CLine 的实现文件 drawdoc.cpp 中,加入重载函数 Draw 的实现代码:

```
void CLine::Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor)
{
    int x1,y1,x2,y2;
    if(b_Delete)                    //如果图形元素处于删除状态,退出函数
        return;
    short LineType = m_LineType;
    short ColorPen = m_ColorPen;
    if(m_DrawMode1 = = 1)       //如果图形元素需要特殊显示
    {
        if(m_LineType! = 2) //如果图形元素采用的不是第二种线型(虚线),则采用第二种线型
            LineType = 2;
        else                        //如果图形元素采用的是第二种线型(虚线),则采用第三种线型
            LineType ++ ;
    }
    else if(m_DrawMode1 = = 2)   //如果需要用指定颜色绘制直线,将笔色设置为屏幕底色 BackColor
        ColorPen = BackColor;
    //创建并初始化一个 CPen 对象
    CPen pen((int)LineType,(int)m_LineWide,p_GraphPara -> GetColor(ColorPen));
    CPen * pOldPen = pDC -> SelectObject(&pen);       //在绘图对象中选入 CPen 对象
    if(m_DrawMode = = 0)                              //如果是采用第 0 种绘制模式
        pDC -> SetROP2(R2_COPYPEN);                   //设定覆盖的绘制模式
    else if( m_DrawMode = = 1)                        //如果是采用第一种绘制模式
        pDC -> SetROP2(R2_NOT);                       //设置反写的绘制模式
    DPtoVP(m_X1,m_Y1,&x1,&y1);                        //将直线起点坐标转换成逻辑坐标
    DPtoVP(m_X2,m_Y2,&x2,&y2);                        //将直线终点坐标转换成逻辑坐标
    pDC -> MoveTo(x1,y1);
    pDC -> LineTo(x2,y2);                             //绘制直线
    pDC -> SelectObject(pOldPen);                     //恢复原来的画笔
}
```

(2) 连续直线和多边形区域的绘制

对于连续直线和封闭区域的绘制,主要用到绘图类 CDC 的成员函数:

```
BOOL Polygon(LPPINT lpPoints,int nCount); //绘制有 nCount 顶点的多边形
```

Polygon 函数使用当前画笔绘制一组由 lpPoints 指定的点。Polygon 函数绘制一个多边形,必要时系统自动把第一个和最后一个顶点连接成封闭多边形,用 SetPolyFillMode 设置的填充模式填充。在文件 drawdoc.cpp 中加入绘制函数的实现代码:

```
void CPline::Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor)
{
    int x1,y1;
    if(b_Delete)          //如果已经处于删除状态
```

```
    return;
short LineType = m_LineType;
short ColorPen = m_ColorPen;
short ColorBrush = m_ColorBrush;
if(m_DrawMode1 = = 1)    //如果图形元素需要特殊显示
{
    if(m_LineType! = 2)  //如果图形元素采用的不是第二种线型(虚线),则采用第二种线型
        LineType = 2;
    else                 //如果图形元素采用的是第二种线型(虚线),则采用第三种线型
        LineType ++ ;
}
else if(m_DrawMode1 = = 2) //如果需要用指定颜色绘制连续直线或多边形区域,将笔色和填
                           //充色设置为屏幕底色 BackColor
{
    ColorPen = BackColor;
    ColorBrush = BackColor;
}
//创建并初始化一个 CPen 对象
CPen pen((int)LineType,(int)m_LineWide,p_GraphPara - > GetColor(ColorPen));
CPen * pOldPen = pDC - > SelectObject(&pen);              //在绘图对象中选入 CPen 对象
CBrush brush(p_GraphPara - > GetColor(m_ColorBrush));    //在绘图对象中选入 CBrush 对象
CBrush * pOldBrush = pDC - > SelectObject(&brush);
POINT * ppoint;
CRgn rgn;
if(m_DrawMode = = 0)                  //如果是采用第 0 种绘制模式
    pDC - > SetROP2(R2_COPYPEN); //设定覆盖的绘制模式
else if( m_DrawMode = = 1)            //如果是采用第一种绘制模式
    pDC - > SetROP2(R2_NOT);      //设置反写的绘制模式
ppoint = new POINT[m_Numble + 1];
for(int i = 0;i < m_Numble;i++ )      //将顶点的实际坐标转换成屏幕逻辑坐标,存入 ppoint
{
    DPtoVP(m_PointList[i].x,m_PointList[i].y,&x1,&y1);
    ppoint[i].x = x1;
    ppoint[i].y = y1;
}
if(! b_Fill)                          //如果是连续直线,则依次绘制各个直线段
{
    ppoint[m_Numble] = ppoint[0];
    pDC - > MoveTo(ppoint[0].x,ppoint[0].y);
    for(i = 0;i < m_Numble;i++ )
        pDC - > LineTo(ppoint[i].x,ppoint[i].y);
}
else                                  //如果是多边形区域
```

```
    {
        if(m_DrawModel == 0 || m_DrawModel == 2)      //如果是正常显示
            pDC->Polygon(ppoint,m_Numble);            //绘制多边形
        else if(m_DrawModel == 1)                     //如果是反色显示
        {
            rgn.CreatePolygonRgn(ppoint,m_Numble,0);  //创建一个 CRgn 对象
            pDC->InvertRgn(&rgn);  //将 CRgn 对象确定的区域反色显示
        }
    }
    delete ppoint;                          //删除动态分配的结构数组
    pDC->SelectObject(pOldPen);             //恢复原来的画笔
    pDC->SelectObject(pOldBrush);           //恢复原来的画刷
}
```

在这个绘制函数中，根据 CPline 类中的 b_Fill 成员变量的不同，选择不同的操作：

当 b_Fill 为 TRUE 时，当前图形元素是一个多边形区域，如果是正常显示或特殊颜色显示时(m_DrawModel == 0 || m_DrawModel == 2)，调用函数 Polygon 绘制了一个多边形区域；如果是特殊显示(m_DrawModel == 1)，用连续直线的顶点坐标初始化一个区域 rgn，用 CDC 类的成员函数 InvertRgn 反色显示这个区域。

当 b_Fill 为 FALSE 时，当前图形元素是一个连续直线，调用 LineTo 函数分段绘制连续直线的各个直线段。

(3) 圆和填充圆的绘制

要绘制一个圆，用到 CDC 类的成员函数 Ellipse：

```
BOOL Ellipse(int x1,int y1,int x2,int y2);
BOOL Ellipse(LPCRECT lpRect);
```

Ellipse 函数的作用是画一个椭圆。椭圆的位置在一个边界矩形之内，椭圆的中心是边界矩形的中心，而边界矩形通过参数 x1、y1、x2、y2 或一个 lpRect 结构所指定。椭圆用当前画笔绘制，其内部用当前画刷填充。

圆是椭圆的一个特殊情况，即边界矩形是正方形的情况。所以，可以使用 Ellipse 函数来绘制圆。至于是否对圆进行填充，可以通过选入一个库存的绘图设备对象来实现：

```
SelectStockObject(NULL_BRUSH);  //设置不填充状态
```

圆的绘制比较简单，将圆心坐标(m_CircleX,m_CircleY)和圆的半径(m_CircleR)都转换成逻辑坐标，得到圆的边界矩形的顶点坐标并传递给 Ellipse 函数绘制即可。在实现文件 drawdoc.cpp 中加入 CCircle 类中虚函数 Draw 的实现代码：

```
void CCircle::Draw(CDC * pDC,int m_DrawMode,int m_DrawModel,short BackColor)
{
    if(b_Delete)          //如果图形元素对象处于删除状态，函数退出
        return;
```

```
int x,y,r;
short LineType = m_LineType;
short ColorPen = m_ColorPen;
short ColorBrush = m_ColorBrush;
if(m_DrawMode1 = = 1)      //如果图形元素需要特殊显示
{
    if(m_LineType! = 2)  //如果图形元素采用的不是第二种线型(虚线),则采用第二种线型
        LineType = 2;
    else                       //如果图形元素采用的是第二种线型(虚线),则采用第三种线型
        LineType++;
}
else if(m_DrawMode1 = = 2) //如果需要用指定颜色绘制圆或圆形区域,将笔色和填充色
                           //设置为屏幕底色 BackColor
{
    ColorPen = BackColor;
    ColorBrush = BackColor;
}
//创建并初始化一个 CPen 对象
CPen pen((int)LineType,(int)m_LineWide,p_GraphPara->GetColor(ColorPen));
CPen* pOldPen = pDC->SelectObject(&pen);                    //在绘图对象中选入 CPen 对象
CBrush brush(p_GraphPara->GetColor(m_ColorBrush));          //在绘图对象中选入 CBrush 对象
CBrush* pOldBrush = pDC->SelectObject(&brush);
if(m_DrawMode = = 0)                        //如果是采用第0种绘制模式
    pDC->SetROP2(R2_COPYPEN);               //设定覆盖的绘制模式
else if( m_DrawMode = = 1)                  //如果是采用第一种绘制模式
    pDC->SetROP2(R2_NOT);                   //设置反写的绘制模式
if(! b_Fill)                                //如果是普通圆,设置不填充模式
    pDC->SelectStockObject(NULL_BRUSH);
DPtoVP(m_CircleX,m_CircleY,&x,&y);          //计算得到圆心的逻辑坐标
r = DLtoVL(m_CircleR);                      //计算得到半径的逻辑坐标长度
pDC->Ellipse(x-r,y-r,x+r,y+r);              //绘制圆
pDC->SelectObject(pOldPen);                 //恢复原来的画笔
pDC->SelectObject(pOldBrush);               //恢复原来的画刷
}
```

(4) 圆弧的绘制

绘制圆弧,要用到 CDC 类的成员函数 Arc:

```
BOOL Arc(int x1,int y1,int x2,int y2,int x3,int y3,int x4,int y4);
BOOL Arc(LPCRECT lpRect,POINT ptStart,POINT ptEND);
```

使用 Arc 函数可以绘制一段椭圆弧,椭圆的大小和位置由前四个参数指定的边界矩形确定。弧的起点是边界矩形中心到参数(x3,y3)或 ptStart 指定点的连线与椭圆的交点,弧的终点

是边界矩形中心到参数(x4,y4)或 ptEnd 指定点的连线与椭圆的交点。弧按逆时针方向为正进行绘制。

与圆是椭圆的特殊情况一样,圆弧也是椭圆弧的一个特殊情况。圆弧的绘制方法与圆类似,只不过需要计算圆弧的起点和终点。圆弧的起点坐标(x,y)用以下公式计算:

```
x = m_CircleX + m_CircleR * cos(m_Angle1);
y = m_CircleY + m_CircleR * sin(m_Angle1);
```

圆弧的终点的计算方法与起点的相同。在实现文件 drawdoc.cpp 中加入类 CArc 的重载虚函数 Draw 的实现代码:

```
void CArc::Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor)
{
    if(b_Delete)                    //如果图形元素处于删除状态,函数退出
        return;
    int x,y,rr;
    int x1,x2,y1,y2;
    short LineType = m_LineType;
    short ColorPen = m_ColorPen;
    if(m_DrawMode1 == 1)            //如果图形元素需要特殊显示
    {
        if(m_LineType != 2)         //如果图形元素采用的不是第二种线型(虚线),则采用第二种线型
            LineType = 2;
        else                        //如果图形元素采用的是第二种线型(虚线),则采用第三种线型
        LineType ++ ;
    }
    else if(m_DrawMode1 == 2) //如果需要用指定颜色绘制圆弧
        ColorPen = BackColor;
    //创建并初始化一个 CPen 对象
    CPen pen((int)LineType,(int)m_LineWide,p_GraphPara->GetColor(ColorPen));
    CPen * pOldPen = pDC->SelectObject(&pen);   //在绘图对象中选入 CPen 对象
    if(m_DrawMode == 0)                         //如果是采用第 0 种绘制模式
        pDC->SetROP2(R2_COPYPEN);               //设定覆盖的绘制模式
    else if( m_DrawMode == 1)                   //如果是采用第一种绘制模式
        pDC->SetROP2(R2_NOT);                   //设置反写的绘制模式
    DPtoVP(m_CircleX,m_CircleY,&x,&y);          //将圆心实际坐标转为逻辑坐标
    rr = DLtoVL(m_CircleR);                     //得到半径的逻辑坐标长度
    //以下得到圆弧起点的逻辑坐标坐标
    x1 = x + DLtoVL(m_CircleR * cos(m_Angle1));
    y1 = y - DLtoVL(m_CircleR * sin(m_Angle1));
    //以下得到圆弧终点的逻辑坐标坐标
    x2 = x + DLtoVL(m_CircleR * cos(m_Angle2));
    y2 = y - DLtoVL(m_CircleR * sin(m_Angle2));
    pDC->Arc(x - rr,y - rr,x + rr,y + rr,x1,y1,x2,y2);     //绘制圆弧
```

```
    pDC->SelectObject(pOldPen);                    //恢复原来的画笔
}
```

(5) 标注文本的绘制

要绘制标注文本，就要用到 CDC 类的成员函数 TextOut：

```
Virtual BOOL TextOut(int x,int y,LPCSTR lpszString,int nCount);
Virtual BOOL TextOut(int x,int y,const Cstring &str);
```

TextOut 函数使用当前选择的字模在指定位置书写一个字符串，原点(x,y)在标注字符串的左上角，要绘制的字符串存储在参数 lpszString 或 str 中。

因为在矢量图形系统中，标注文本是无级放缩的，而且标注行可能不是水平的，字体的间距也是任意的，所以直接应用 TextOut 函数不能满足矢量图形系统中绘制标注文本的要求。要完成任意绘制标注文本的功能，需要经过一些处理操作，其操作步骤如下：

① 根据标注文本的实际高度得到在当前视图中文本的字符高度，根据字体的宽度比例得到字符宽度，根据字体的旋转角度得到字体的旋转角度，然后以这些参数创建相应的字模，创建字模的方法在第 4.2.4 小节中作已过介绍。

② 在绘制一行文本的时候，一个字符(或汉字)一个字符书写的，根据字体的高度、宽度等得到每一个字符(或汉字)的起始位置。在这个环节上，需要判断要绘制的下一个字符是一个西文字符还是一个汉字。

③ 在屏幕相应的位置上用 TextOut 函数绘制字符，直到绘制完标注文本的所有字符。

判断下一个字符是汉字还是西文字符的方法如下：西文字符占有一个字节，一般根据其最高位是否为 1，判断其为基本的西文字符或扩展码两部分。一个汉字占有两个字节，并且每个字节的最高位都为 1，一般就用相连的两个字符的最高位是否都为 1 来判断是否是一个汉字。可以用这个字符的值是否大于 127 来判断字符的最高位是否为 1，以下代码可以判断字符指针 ls 所指的字符是否为一个汉字：

```
unsigned char c1,c2;
c1 = *ls;
c2 = *(ls+1);
if(c1>127&&c2>127)          //如果两个字符最高位都为 1,
    ;                       //字符指针 ls 指向的位置是一个汉字
```

在实现文件 drawdoc.cpp 中加入这个绘制函数的代码：

```
void CText::Draw(CDC *pDC,int m_DrawMode,int m_DrawMode1,short BackColor)
{
    if(b_Delete)            //如果图形元素处于删除状态，函数退出
        return;
    short ColorPen=m_ColorPen;
    if(m_DrawMode1==2) //如按指定颜色绘制，将笔色设置为参数 BackColor 指定的颜色号
        ColorPen=BackColor;
    CFont cjcf;             //定义一个字模
```

```
int high,wide,cc1,cc2,cd,angg;
unsigned char c1,c2;
float x1,y1,angl;
char hz[4];
const char * ls1;
if(m_DrawMode==0)                          //如果是采用第0种绘制模式
    pDC->SetROP2(R2_COPYPEN);              //设定覆盖的绘制模式
else if(m_DrawMode==1)                     //如果是采用第一种绘制模式
    pDC->SetROP2(R2_NOT);                  //设置反写的绘制模式
pDC->SetBkMode(TRANSPARENT);
high=DLtoVL(m_TextHeight);                 //得到字体的逻辑高度
angl=(float)(m_Angle1*pi/180);             //得到标注行与X轴正向的弧度
if(high<31||high>200)
    return;                                //字体太小或太大,将不显示
x1=m_StartX-m_TextHeight*(float)sin(angl); //标注起点的实际横坐标
y1=m_StartY+m_TextHeight*(float)cos(angl); //标注起点的实际纵坐标
angg=(int)(m_Angle2*10);
wide=DLtoVL(m_TextWide);  //字体的逻辑坐标宽度
DPtoVP(x1,y1,&cc1,&cc2);  //标注始点的逻辑坐标
cjcf.CreateFont(high,wide,angg,0,50,0,0,0,255,OUT_TT_PRECIS,CLIP_CHARACTER_PRECIS,
      DEFAULT_QUALITY,FIXED_PITCH,"cjc");          //创建字模
CFont * cjcbakf=pDC->SelectObject(&cjcf);          //选入字模
pDC->SetBkMode(TRANSPARENT);                       //设定绘制方式
pDC->SetTextColor(p_GraphPara->GetColor(ColorPen)); //设置标注文本颜色
ls1=(const char *)c_Text;      //字符指针ls1指向标注文本内容
cd=strlen(ls1);                //标注文本的字符长度
while(cd>0)                    //绘制标注文本的所有字符
{
    c1=*ls1;
    c2=*(ls1+1);
    if(c1>127&&c2>127)                 //如果下一个字符是一个汉字
    {
        strncpy(hz,ls1,2);             //拷贝一个汉字到字符串hz中
        hz[2]=0;
        ls1=ls1+2;                     //跳过汉字指向下一个字符位置
        pDC->TextOut(cc1,cc2,hz);      //在屏幕上绘制这个汉字
        cd=cd-2;                       //字符串长度减2
        //下一个字符的实际显示位置
        x1=x1+(m_TextWide*2+m_OffWide)*(float)cos(angl);
        y1=y1+(m_TextWide*2+m_OffWide)*(float)sin(angl);
        DPtoVP(x1,y1,&cc1,&cc2);       //得到下一个字符显示位置的逻辑坐标
    }
    else                       //如果是一个西文字符
```

```
            {
                strncpy(bz,ls1,1);              //拷贝一个字符到字符串 bz 中
                bz[1] = 0;
                ls1 ++ ;                        //指向字符串中的下一个字符
                pDC - > TextOut(cc1,cc2,bz);    //在屏幕上绘制字符
                cd = cd - 1;                    //字符串长度减 1
                //下一个字符显示位置的实际坐标
                x1 = x1 + (m _ TextWide + m _ OffWide/2) * (float)cos(angl);
                y1 = y1 + (m _ TextWide + m _ OffWide/2) * (float)sin(angl);
                DPtoVP(x1,y1,&cc1,&cc2);        //得到下一个字符显示位置的屏幕逻辑坐标
            }
        }
        pDC - > SelectObject(cjcbakf);    //恢复字模
    }
```

5.3.3　视图类的 OnDraw 函数

在 5.3.2 小节中，实现了各个图形元素类的绘制函数 Draw，而如何在视图类中调用这些绘制函数完成图形的绘制，是我们关心的又一个问题。

在应用程序 Draw 源代码中，在视图类 CDrawView 中可以找到一个虚函数 OnDraw，这个函数负责视图的绘制工作。读者可能不清楚 OnDraw 函数从何处被调用，下面对 OnDraw 函数的调用和工作过程进行分析。

OnDraw 函数是被 CView 类的消息处理函数 OnPaint 调用的。从 MFC 中可以找到 OnPaint 函数的实现代码：

```
void CView::OnPaint()
{
    // standard paint routine
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    OnDraw(&dc);
}
```

OnPaint 函数是窗口消息 WM _ PAINT 的消息处理函数，WM _ PAINT 消息是当窗口的完整性受到破坏时发出的一个窗口消息。也就是说，当窗口中产生 WM _ PAINT 消息时，视图类 CDrawView 基类 CView 中的 OnPaint()函数被执行(因为当前视图类中没有定义 OnPaint 消息处理函数)。在 OnPaint 函数中，调用了 OnDraw 函数，因为 OnDraw 函数是在 CView 类中定义的虚函数，所以根据多态性调用了当前视图类 CDrawView 中重载的虚函数 OnDraw，来完成视图的绘制工作。

为了分析这个调用过程，可以用 ClassWizard 在当前视图类 CDrawView 中加入消息处理函数 OnPaint，操作完毕后，在实现文件 drawview.cpp 中，可以找到自动加入的 OnPaint 函数的实现代码：

```
void CDrawView::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    // TODO: Add your message handler code here
    // Do not call CView::OnPaint() for painting messages
}
```

以上的 OnPaint 函数中没有完成任何操作，可以在其中加入调用 OnDraw 函数的代码，得到新的实现代码如下：

```
void CDrawView::OnPaint()
{
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    OnDraw(&dc);
}
```

OnPaint 函数代替了 CView 类中的 OnPaint 函数，来响应 WM_PAINT 窗口消息，并调用 OnDraw 函数完成视图的绘制工作。

以上分析了 OnDraw 函数的工作过程。在实际应用程序设计中，不必在当前视图类中重载 OnPaint 消息处理函数。

5.3.4 实现视图

在 5.3.3 小节讨论了 OnDraw 函数被调用的过程。在 OnDraw 函数中加入适当的代码，就能够实现利用文档完成矢量图形的绘制工作。

在 3.4.4 小节讨论了在视图对象中得到其对应的文档对象的方法，即通过 GetDocument 函数得到指向当前文档对象的指针。在 OnDraw 函数中利用 GetDocument 函数得到指向当前文档对象的指针，然后利用这个指针调用各个图形元素类的绘制函数，完成图形的绘制工作。这一步骤，也是文档/视图体系的基础。

为了实现应用程序 Draw 的图形绘制工作，在文档类 CDrawDoc 中定义一个函数如下：

```
public:
    void Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor);
```

在实现文件 drawdoc.cpp 中，加入这个函数的实现代码：

```
void CDrawDoc::Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor)
{
    for(int i=1;i<=5;i++)          //对直线、连续直线、圆、圆弧、标注文本等五类图形元素循环
    {
        int nn= GetGraphUpperBound(i)+1; //得到存储第 i 类图形元素对象指针数组的最大下标
        while(nn--)  //对各类所有的图形元素循环，得到指向图形元素对象的指针并进行绘制
            GetGraph(i,nn)->Draw(pDC,m_DrawMode,m_DrawMode1,BackColor);
    }
}
```

以上函数实现了对直线、连续直线及多边形区域、圆及圆形区域、圆弧，以及标注文本等五类图形元素的绘制工作。对于每一类图形元素，绘制过程如下：

① 调用 GetGraphUpperBound 函数，得到存储这类图形元素对象指针的最大数组下标。

② 对所有的图形元素循环，得到指向每一个图形元素对象的指针。然后通过这个指针调用图形元素类的 Draw 函数实现图形元素的绘制操作。

必须注意，GetGraph 函数返回的是 CDraw 类指针，这个指针指向了具体的图形元素对象。因为 CDraw 类中的 Draw 函数是虚函数，在各个派生类(CLine、CPline 等)中重载了这个虚函数，所以，根据程序的多态性，CDraw 指针会分别调用对应图形元素类的 Draw 函数，完成图形元素的绘制工作。

为了实现图形元素在视图中的绘制，对视图类的 OnDraw 函数进行修改，修改后的 OnDraw 函数实现代码如下：

```
void CDrawView::OnDraw(CDC * pDC)
{
    CDrawDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    DrawBack(pDC);          //以底色填充整个屏幕
    pDoc->Draw(pDC,0,0,m_bColor);
}
```

在 OnDraw 函数中，调用了文档类的 Draw 函数来绘制所有的图形元素。DrawBack 函数被调用，用来在绘制图形前将视图屏幕填充成屏幕底色，DrawBack 函数在 CDrawView 类中定义，在实现文件 drawview.cpp 中加入了实现代码：

```
void CDrawView::DrawBack(CDC * pDC)
{
    CBrush brush(p_GraphPara->GetColor(m_bColor));          //创建 CBrush 对象
    CBrush * pOldBrush = pDC->SelectObject(&brush);         //将画刷选入绘图对象
    pDC->PatBlt(0,0,m_wScreen,m_hScreen,PATCOPY);           //用画刷填充视图区域
    pDC->SelectObject(pOldBrush);                           //恢复原来的画刷
}
```

在 DrawBack 函数中，调用了 CDC 类的位操作函数 PatBlt 将整个视图屏幕(矩形区域{0,0,m_wScreen,m_hScreen})用屏幕底色 m_bColor 填充。

本章对直线、连续直线及封闭多边形区域、圆、圆弧、标注文本等基本的图形元素的数据组织和绘制方法作了介绍，对于其他的图形元素，请读者参考这些图形元素的实现方法组织实现。

第 6 章　鼠标交互绘图

本章讨论用鼠标在视图中交互绘制图形元素，并将绘制的图形元素保存到文档中的方法。读者只要用过 AutoCAD 或 Windows 的 Pbrush 作过图，都会对用鼠标交互绘图的步骤有一些了解，但可能对如何完成用鼠标作图的实现过程不太了解。本章将在矢量图形系统中，讨论实现鼠标交互绘图功能。

6.1　用鼠标绘图要解决的主要问题

要完整的实现一个鼠标交互绘图功能，需要解决以下各小节的主要问题。

6.1.1　捕捉鼠标操作消息

VC++ 提供了足够的窗口消息和保护的消息处理函数解决这个问题。在窗口类 CWnd 中有大量的已经保护的鼠标消息处理函数，对应着各种鼠标操作所产生的窗口消息。当对鼠标进行某个操作时（如按下某个键，松开某个键，双击某个键，移动鼠标时），会产生对应的窗口消息，Windows 会采集到这些消息并交给活动的窗口，如果在这个窗口中有对应的消息处理函数，就会执行这个函数。用手工和 ClassWizard 增加一个消息处理函数的方法在 2.5.4 小节已作了介绍。表 6-1 列出了几个主要鼠标操作动作的消息处理函数及其对应的消息。

表 6-1　主要鼠标操作动作的消息处理函数和对应消息

鼠 标 操 作	已经保护的消息处理函数	对 应 消 息
鼠标移动时	OnMouseMove(UINT nFlags, Cpoint point)	WM _ MOUSEMOVE
按下鼠标左键	OnLButtonDown(UINT nFlags, CPoint point)	WM _ LBUTTONDOWN
松开鼠标左键	OnLButtonUp(UINT nFlags, CPoint point)	WM _ LBUTTONUP
按下鼠标右键	OnLButtonDown(UINT nFlags, CPoint point)	WM _ RBUTTONDOWN
松开鼠标右键	OnLButtonUp(UINT nFlags, CPoint point)	WM _ RBUTTONUP
双击鼠标左键	OnLButtonDblClk(UINT nFlags, CPoint point)	WM _ LBBUTTONDBLCLK
双击鼠标右键	OnRButtonDblClk(UINT nFlags, CPoint point)	WM _ RBBUTTONDBLCLK

在表 6-1 列出的消息处理函数中，参数 point 返回当时窗口内点的像素坐标，通过在这些已保护的消息处理函数中加入代码，就能够达到捕捉和处理各种鼠标操作的目的。

各种鼠标消息处理函数的使用方法是一样的。在应用程序设计时，具体采用哪个消息处理函数，应根据要完成操作的具体要求来选用（如用鼠标交互绘制一条直线，可以用按下鼠标器左键时作为直线的起点，松开鼠标器左键时作为直线的终点来完成，同样也可以用两次按中鼠标器左键作为直线的起终点的方法来实现）。在应用程序 Draw 中实现鼠标交互绘制图形元素时，将主要用到以下三种鼠标操作：

鼠标移动	OnMouseMove(UINT nFlags,CPoint point)	WM_MOUSEMOVE
按下左键	OnLButtonDown(UINT nFlags,CPoint point)	WM_LBUTTONDOWN
按下右键	OnRButtonDown(UINT nFlags,CPoint point)	WM_RBUTTONDOWN

为了能够在应用程序 Draw 中使用这三种鼠标操作，用 ClassWizard 在应用程序 Draw 的视图类 CDrawView 中增加三个消息处理函数。增加消息处理函数的方法在 2.5.4 小节中已作了介绍。操作完毕后，应用程序 Draw 源代码做了如下修改：

① 在视图类类 CDrawView 中，加入这三个鼠标消息处理函数的定义：

```
//{{AFX_MSG(CDrawView)
afx_msg void OnSize(UINT nType, int cx, int cy);
afx_msg void OnMouseMove(UINT nFlags,CPoint point);
afx_msg void OnLButtonDown(UINT nFlags,CPoint point);
afx_msg void OnRButtonDown(UINT nFlags,CPoint point);
//}}AFX_MSG
```

② 在实现文件 drawview.cpp 中，加入了这三个函数的消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    ON_WM_SIZE();
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONDOWN()
    ON_WM_RBUTTONDOWN()
END_MESSAGE_MAP()
```

③ 在实现文件 drawview.cpp 中，加入了这三个消息处理函数的实现代码：

```
void CDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    CView::OnMouseMove(nFlags,point);
}

void CDrawView::OnLButtonDown(UINT nFlags,CPoint point)
{
    CView::OnLButtonDown(nFlags,point);
}

void CDrawView::OnRButtonDown(UINT nFlags,CPoint point)
{
    CView::OnRButtonDown(nFlags,point);
}
```

在这三个函数中没有执行任何操作。如果此时编译并执行应用程序 Draw，当移动鼠标、按下鼠标左键、按下鼠标右键等操作时，程序会调用相应的消息处理函数，但没有完成具体的操作。在这三个消息处理函数中加入合适的代码，就能利用捕捉的鼠标信息来完成图形元素的鼠标交互绘制。具体实现方法参见以下各节的讨论。

6.1.2 捕捉所有的鼠标输入

在用鼠标交互绘制一个图形元素时,在已经开始绘图的情况下(如在绘制一条直线时,已经按中了直线的起点并正在屏幕上拖动时),不希望还能进行别的操作(如打开菜单执行另一项功能),以避免造成系统流程和变量初始化等方面的错误。VC++提供了两个成员函数来实现以上的功能需求:

```
CWnd::SetCapture();
CWnd::ReleaseCapture();
```

SetCapture 函数运行后,使所有的后续鼠标输入被送至当前 CWnd 对象,这样就把所有的鼠标输入采集到了(当然就不能运行其他菜单了)。ReleaseCapture 与 SetCapture 函数相反,用以释放这种捕捉。这样,就可以在开始绘图时运行 SetCapture 函数,在图形元素交互绘制完毕后再运行 ReleaseCapture 函数,用来避免在交互绘制图形元素过程中其他操作的"干扰"。

6.1.3 在屏幕上拖动图形

在用鼠标交互绘制图形元素时,为了直观的看到所绘制的图形,一般采用拖动图形的方法。这种情况一般是在一个图形元素只剩最后一个控制点没有确定的情况下,以鼠标移动点作为图形元素的最后一个控制点形成图形元素的具体形状,这样当鼠标移动时,图形也跟着变化,就产生了"拖动"的效果。例如,在交互绘制直线时,在确定了直线的起点后,鼠标移动时就拖动一条"橡皮"线移动。这种功能是一个鼠标交互绘图程序所必须的。

要实现拖动图形的功能,首先要用 CDC 类的函数 SetRop2 设置绘制模式(参见 3.3.3 小节)。当把绘制模式设置成 R2_NOT 时,因为两次绘制同一个图形元素视图屏幕不变,所以在用鼠标交互绘制图形元素时,在只剩下一个控制点的情况下,当鼠标移动时,首先把上一个移动点所决定的图形元素绘制一次(擦除上一个图形元素),再把当前鼠标点决定的图形元素绘制一次,并记录下鼠标点作为下一个移动点时的上一个点,这样当鼠标移动时就实现了拖动图形的效果。在编制拖动图形元素的程序时,要注意对第一次移动操作进行处理,因为第一次移动图形元素时,没有上一个图形元素要擦除。

6.1.4 保存图形数据到文档

用鼠标在屏幕上交互绘制的图形元素,要创建一个图形元素对象并将指向这个图形元素对象的指针保存起来。也就是说,在用鼠标在视图中交互绘制图形元素时,在屏幕上完成图形元素绘制的同时,还需要把这个图形元素保存到文档中,否则,重画视图时这个交互绘制的图形元素就消失了。

6.1.5 将图形以实际的形态重画

用鼠标交互绘制一个图形元素后,要在屏幕上马上显示出这个图形元素的实际形态,必须将图形元素以实际的形态重画。读者也许要问,在鼠标交互绘图时不是时刻有一个拖动的图形,为什么还要对这个图形元素进行重画?原因是,在 VC++的 R2_NOT 绘制模式下,线条不

是彩色的,拖动的图形都不是图形元素的实际颜色,必须对图形元素以实际形态进行重画。重画这个新绘制的图形元素,可以通过发送一个窗口消息使屏幕全部或部分区域重画来实现。但这个方法效果不佳,因为要达到使新绘制的图形元素重画的目的,重画的最小区域也是包含这个图形元素的边界矩形(当然也可以重画一个任意多边形区域),在有大量图形元素的情况下,重画的速度是很慢的,所以一般不采用这种方法来重画交互绘制的图形元素。在矢量图形系统设计中,一般采用 CClientDC 对象,在交互绘制图形元素操作完成后,即时在视图中重画这个新增的图形元素。

6.2 加入一个绘图菜单

为了能够在应用程序 Draw 中实现交互绘制图形元素的功能,需要首先在应用程序 Draw 中增加一个菜单来激发进行各类图形元素的交互绘制操作。本节,将在应用程序 Draw 中增加一个图形交互绘制菜单。

6.2.1 增加菜单资源

首先,利用 ResoreceView 和相应的工具(参考 2.3.2 小节)在应用程序 Draw 的菜单资源 IDR_DRAWTYPE 中增加一个菜单"鼠标绘图",在这个菜单中包含"直线"、"连续直线"、"多边形区域"、"圆"、"圆弧"、"标注文本"等七个菜单项。增加菜单资源后,应用程序 Draw 的代码做了如下修改:

① 在资源文件 draw.rc 中加入了增加菜单的描述代码:

```
IDR_DRAWTYPE MENU PRELOAD DISCARDABLE
BEGIN
    ..... //省略以上代码
    POPUP "鼠标绘图"
    BEGIN
        MENUITEM "直线",          ID_DRAW_LINE
        MENUITEM "连续直线",      ID_DRAW_PLINE
        MENUITEM "多边形区域",    ID_DRAW_RGN
        MENUITEM "圆",            ID_DRAW_CIRCLE
        MENUITEM "圆形区域",      ID_DRAW_CIRCLE1
        MENUITEM "圆弧",          ID_DRAW_ARC
        MENUITEM "标注文本",      ID_DRAW_TEXT
    END
    ..... //省略以下代码
END
```

② 在资源 ID 号定义文件 resource.h 中,增加了对菜单命令 ID 号的定义

```
#define ID_DRAW_LINE          32771
#define ID_DRAW_PLINE         32772
#define ID_DRAW_RGN           32773
```

```
# define ID_DRAW_CIRCLE          32774
# define ID_DRAW_CIRCLE1         32775
# define ID_DRAW_ARC             32776
# define ID_DRAW_TEXT            32777
```

6.2.2　增加菜单的消息处理函数

增加菜单资源后，利用 ClassWizard 为增加的菜单项在视图类 CDrawView 中建立消息处理函数，具体操作方法见 2.5.4 小节(也可以手工增加)。以下是增加消息处理函数后程序源代码所做的修改：

① 在视图类 CDrawView 中，加入了消息处理函数的定义：

```
//{{AFX_MSG(CDrawView)
afx_msg void OnSize(UINT nType, int cx, int cy);
afx_msg void OnMouseMove(UINT nFlags,CPoint point);
afx_msg void OnLButtonDown(UINT nFlags,CPoint point);
afx_msg void OnRButtonDown(UINT nFlags,CPoint point);
afx_msg void OnDrawArc();          //绘制圆弧
afx_msg void OnDrawCircle();       //绘制圆
afx_msg void OnDrawCircle1();      //绘制圆形区域
afx_msg void OnDrawLine();         //绘制直线
afx_msg void OnDrawPline();        //绘制连续直线
afx_msg void OnDrawRgn();          //绘制多边形区域
afx_msg void OnDrawText();         //绘制标注文本
//}}AFX_MSG
```

② 在实现文件 drawview.cpp 中，增加了函数的消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    ......//省略以上代码
    ON_COMMAND(ID_DRAW_ARC, OnDrawArc)
    ON_COMMAND(ID_DRAW_CIRCLE, OnDrawCircle)
    ON_COMMAND(ID_DRAW_CIRCLE1, OnDrawCircle1)
    ON_COMMAND(ID_DRAW_LINE, OnDrawLine)
    ON_COMMAND(ID_DRAW_PLINE, OnDrawPline)
    ON_COMMAND(ID_DRAW_RGN, OnDrawRgn)
    ON_COMMAND(ID_DRAW_TEXT, OnDrawText)
END_MESSAGE_MAP()
```

③ 在实现文件 drawview.cpp 中，增加了消息处理函数的实现代码，以下是经过修改后的函数实现代码列表：

```
void CDrawView::OnDrawLine()
{
    PushNumb = 0;              //鼠标左键按下次数为 0
```

```
    m_DrawCurrent = 1;          //标识进行直线绘制操作
}

void CDrawView::OnDrawPline()
{
    PushNumb = 0;               //鼠标左键按下次数为0
    m_DrawCurrent = 2;          //标识进行连续直线绘制操作
}

void CDrawView::OnDrawRgn()
{
    PushNumb = 0;               //鼠标左键按下次数为0
    m_DrawCurrent = 3;          //标识进行多边形区域绘制操作
}

void CDrawView::OnDrawCircle()
{
    PushNumb = 0;               //鼠标左键按下次数为0
    m_DrawCurrent = 4;          //标识进行圆绘制操作
}

void CDrawView::OnDrawCircle1()
{
    PushNumb = 0;               //鼠标左键按下次数为0
    m_DrawCurrent = 5;          //标识进行圆形区域操作
}

void CDrawView::OnDrawArc()
{
    PushNumb = 0;               //鼠标左键按下次数为0
    m_DrawCurrent = 6;          //标识进行圆弧绘制操作
}

void CDrawView::OnDrawText()
{
    PushNumb = 0;               //鼠标左键按下次数为0
    m_DrawCurrent = 7;          //标识进行标注文本绘制操作
}
```

这几个函数的实现代码非常简单，只是对两个变量 PushNumb 和 m_DrawCurrent 进行了初始化，这两个变量用来记录当前进行的操作类别和按下鼠标左键的次数，这两个变量在视图类 CDrawView 中定义：

```
protected:
```

```
int m_DrawCurrent; //用来记录现在的操作类别
int PushNumb;      //用来记录按下鼠标左键的次数
```

经过以上步骤的操作，就在应用程序 Draw 中增加一个新的菜单，并在视图类中增加了各个菜单项的消息处理函数。程序编译后运行，就会得到如图 6-1 所示的运行界面，执行菜单“鼠标绘图”下的菜单项后，系统没有进行任何操作。在下节中，要对应用程序 Draw 进行修改，使这几个菜单项完成具体的交互绘制图形元素功能。

图 6-1　应用程序 Draw 的鼠标绘制菜单

6.3　交互绘制各种图形元素

本节将介绍交互绘制直线、连续直线、多边形区域、圆、圆形区域、圆弧等图形元素的实现过程。

6.3.1　获得图形元素的唯一识别号

我们设计的矢量图形系统，要应用在 GIS 系统中，所以必须给每个图形元素一个唯一的识别号。本小节，讨论为每类图形元素创建唯一识别号的实现方法。

(1) 唯一识别号的范围

在 5.1 节曾经讨论了图形元素识别号的唯一性范围。在应用程序 Draw 中，我们将采用同类图形元素具有唯一识别号的方法，即每一类别的所有图形元素具有不同的成员变量 m_id_only。

(2) 得到图形元素的唯一识别号

为了得到每类图形元素的唯一识别号，在文档类 CDrawDoc 中定义一个成员函数：

```
public:
    int GetGraphID(short Lb);    //用来得到要增加的各类图形元素的唯一识别号
```

GetGraphID 函数用来得到要增加的各类图形元素的唯一识别号，参数 Lb 用来表示图形类

别,在实现文件 drawdoc.cpp 中加入函数的实现代码:

```
int CDrawDoc::GetGraphID(short Lb)
{
    int nn = GetGraphUpperBound(Lb);//用来得到存储第 Lb 类图形元素对象指针数组的最大下标
    for(int i = 0;i < 20000;i++)      //将整数数组 m_Index 的前 20000 项设置为 0
        m_Index[i] = 0;
    for(i = 0;i < = nn;i++)           //对存储图形元素对象指针数组的所有数组项循环
    {
        if(GetGraph(Lb,i)) //如果读到的是一个对象指针,得到指针指向的图形元素的唯一
                           //识别号,将整数数组 m_Index 中以这个识别号为下标的项设置为 1
            m_Index[GetGraph(Lb,i) - > GetID()] = 1;
    }
    for(i = 0;i < 20000;i++)      //对 m_Index 数组项循环
    {
        if(m_Index[i] = = 0)    //如果数组项等于 0,则这个数组项的下标就是要找的唯一识别号
            return i;
    }
    return -1;
}
```

GetGraphID 函数用来得到与某个类别的所有图形元素对象的成员变量 m_id_only 不同的最小正整数。函数中所用的成员变量 m_Index 是一个整数数组,在文档类 CDrawDoc 中定义:

```
public:
    int * m_Index;
```

在 CDrawDoc 类的构造函数中被动态分配空间:

```
CDrawDoc::CDrawDoc()
{
    ......//省略以上代码
    m_Index = new int[20000];
}
```

在 CDrawDoc 类的析构函数中删除这个动态分配的整型数组:

```
CDrawDoc::CDrawDoc()
{
    delete m_Index;
}
```

在 GetGraphID 函数中,函数 GetID 被调用来得到一个图形元素对象的识别号。GetID 函数在图形元素基类 CDraw 中定义:

```
public:
    int GetID();
```

在实现文件 drawdoc.cpp 中,增加了函数的实现代码:

```
int CDraw::GetID()
{
    return m_id_only;
}
```

GetID 函数返回了图形元素对象的唯一识别号,即成员变量 m_id_only 的值。

6.3.2 绘制直线

本小节实现鼠标交互绘制直线的功能。具体的操作步骤是:选中"鼠标绘图"菜单中的"直线"菜单项后,按下鼠标左键点中直线的起点,然后当鼠标在屏幕上移动时就拖动一条直线在屏幕上移动,如果再按下鼠标左键选中直线的终点,就完成了直线的交互绘制操作,如果在按下直线起点后按下的是鼠标右键,则擦除拖动的橡皮线,放弃本次直线绘制操作。以下是具体的实现过程。

(1) 修改消息处理函数 OnLButtonDown

为了完成鼠标交互绘制直线的操作功能,在实现文件 drawview.cpp 中的消息处理函数 OnLButtonDown 中加入如下代码:

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CDrawDoc* pDoc = GetDocument();   //得到指向当前文档的指针
    CClientDC ht(this);
    float r;
    float xx1,yy1,xx2,yy2;
    int x1,x2,y1,y2,Lb,Index,id_only;
    BOOL m_Fill;
    if(m_DrawCurrent==1)              //如果正在绘制直线
    {
        if(PushNumb==0)               //如果是第一次按下左鼠标键
        {
            PushNumb++;               //按下鼠标左键的次数加一次
            mPointOrign=point;        //直线的起点等于按中点
            mPointOld=point;          //记录本次按中点
            SetCapture();             //捕捉鼠标输入
        }
        else if(PushNumb==1)          //第二次按下左鼠标键(即按下直线的结束点)
        {
            //以下得到直线起终点的实际坐标
            VPtoDP(mPointOrign.x,mPointOrign.y,&xx1,&yy1);
            VPtoDP(point.x,point.y,&xx2,&yy2);
            id_only=pDoc->GetGraphID(1);      //得到要增加直线对象的唯一识别号
```

```
                pDoc -> AddLine(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
                    xx1,yy1,xx2,yy2) -> Draw(&ht,0,0,m_bColor); //增加并重新绘制此直线
                PushNumb = 0;          //鼠标左键按下次数为 0,重新进行直线的绘制
                ReleaseCapture();      //释放捕捉的光标
            }
        }
    }
```

以上代码,在绘制直线状态下(DrawCurrent = = 1),当第一次按下鼠标左键时(PushNumb = = 0),把鼠标按下点(mPointOrign)记录为直线的起点和鼠标移动时的上一个点(mPointEnd),记录下按键次数并捕捉以后的鼠标操作。当第二次按下鼠标左键时(PushNumb = = 1),调用 CDrawDoc 类的函数 AddLine 增加一个直线对象并记录指向直线对象的指针,并利用返回的直线对象指针调用 CLine 类的绘制函数 Draw,完成直线的重画操作。然后,释放了捕捉的鼠标,并将鼠标左键次数设置为 0(PushNumb = 0),重新开始直线的交互绘制操作。

OnlButtonDown 函数中所调用的一些变量,在视图类 CDrawView 中定义:

```
private:
    CPoint mPointOrign,mPointOrign1,mPointOld; //记录鼠标移动时的鼠标点的坐标
    short m_pColor;     //当前画笔颜色的序号
    short m_bColor;     //当前屏幕底色的序号
    short m_brColor;    //当前画刷颜色的序号
    short m_LineWide;   //当前直线宽度(像素)
    short m_LineType;   //当前线型
    short m_Layer;      //当前图层
```

在实现文件 drawview.cpp 中 CDrawView 类的构造函数中,这些成员变量被初始化:

```
CDrawView::CDrawView()
{
    ......
    m_pColor = 1;       //设置笔色
    m_bColor = 0;       //当前底色的序号
    m_brColor = 1;      //设置填充色
    m_LineWide = 1;     //设置线宽
    m_LineType = 0;     //设置线型
    m_Layer = 1;        //选择当前图层
    ......
}
```

(2) 修改消息处理函数 OnMouseMove

为了在交互绘制直线时能实现拖动橡皮线的功能,在实现文件 drawview.cpp 中的消息处理函数 OnMouseMove 中加入如下代码:

```
void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
```

```
{
    CDrawDoc * pDoc = GetDocument();          //得到指向当前文档对象的指针
    CClientDC ddd(this);
    CPen pen(0,0,RGB(0,0,0));                 //创建 CPen 对象
    CPen * pOldPen = ddd.SelectObject(&pen);  //在绘图对象中选择 CPen 对象
    ddd.SetROP2(R2_NOT);                      //选择反色的绘制模式
    if(m_DrawCurrent == 1&&PushNumb == 1) //如果是绘制直线并且已经按下了直线的起点
    {
        if(mPointOld! = point)   //如果鼠标的现在移动点与上一个点不相同,将起点到上一个
                                 //移动点的直线擦除,绘制从起点到现在移动点的直线
        {
            ddd.MoveTo(mPointOrign);
            ddd.LineTo(mPointOld);   //擦除上一条直线
            ddd.MoveTo(mPointOrign);
            ddd.LineTo(point);       //绘制直线起点到鼠标移动点的连线
            mPointOld = point;       //将当前鼠标点记录为下次移动操作的上一个移动点
        }
    }
}
```

以上这些代码实现橡皮线的绘制。如果正在绘制一条直线,在已经第一次按下鼠标左键选中了直线起点(m_DrawCurrent == 1&&PushNumb == 1)的情况下,如果现在的鼠标点与上一个点不相同(mPointOld! = point),就要设置 R2_NOT 的绘制模式(SetROP2(R2_NOT);)。首先将原来的橡皮线擦除,然后绘制直线起点到鼠标点的连线,并将当前鼠标点记录为上一个移动点 mPointEnd。当鼠标移动时,这个过程一直进行下去,就实现了拖动橡皮线的功能。

在进行第一次移动操作时,没有上一条橡皮线供擦除,但因为这时鼠标点与直线起点相同,所以擦除的直线的起终点相同,不会在屏幕上留下一条多余的直线。

(3) 修改消息处理函数 OnRButtonDown

为了在交互绘制直线的操作过程中,能够放弃正在进行的绘制操作,需要在实现文件 drawview.cpp 中的消息处理函数 OnRButtonDown 中,加入如下代码:

```
void CDrawView::OnRButtonDown(UINT nFlags, CPoint point)
{
    CDrawDoc * pDoc = GetDocument();
    CClientDC ddd(this);
    CPen pen(0,0,RGB(0,0,0));                 //创建一个 CPen 对象
    CPen * pOldPen = ddd.SelectObject(&pen);  //将 CPen 对象选入绘图对象
    ddd.SetROP2(R2_NOT);                      //设置反色的绘制模式
    if(m_DrawCurrent == 1&&PushNumb == 1) //如果正在绘制直线并且已经按下了直线的起点
    {
        ddd.MoveTo(mPointOrign);
        ddd.LineTo(mPointOld);                //擦除视图屏幕上拖动的橡皮线
```

```
            PushNumb = 0;
            ReleaseCapture();                    //释放捕捉的鼠标
        }
        ddd.SelectObject(pOldPen);               //恢复原来的画笔
    }
```

在交互绘制直线并且已经按中了直线起点(正在进行直线拖动)的情况下,按中鼠标右键会将屏幕上拖动的橡皮线擦除,结束本次直线的绘制并释放捕捉的鼠标。

至此,完成了在屏幕上交互绘制直线的操作功能,编译并执行应用程序 Draw,就会得到如图 6-2 的运行界面。

图 6-2　用鼠标在屏幕上绘制直线

6.3.3　绘制连续直线和多边形区域

连续直线和多边形区域的鼠标交互绘制,采用与直线相似的方法。不同的是,连续直线是由多个顶点组成的,在纯鼠标操作状态下,需要加入更多的鼠标控制来完成操作。现在设定连续直线和多边形区域的绘制方法如下:运行绘制连续直线或多边形区域的操作菜单后,按下鼠标左键时,会在视图屏幕上连续绘制直线段,鼠标移动时,移动点和上一个顶点之间有拖动线,按下鼠标右键时,就会完成连续直线或多边形区域的绘制操作,创建并初始化一个 CPline 对象,并将指向这个对象的指针保存起来,并调用 CPline 类的 Draw 函数重画这个图形元素。

为了实现以上的交互绘制功能,同样需要对视图类 CDrawView 中的三个鼠标消息处理函数进行修改。

(1) 修改消息处理函数 OnLButtonDown

在实现文件 drawview.cpp 中的消息处理函数 OnLButtonDown 中,加入如下代码:

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    ......//省略以上代码
    else if(m_DrawCurrent==2||m_DrawCurrent==3)//正在进行交互绘制连续直线或多边形区域
    {
```

```
            //将鼠标点转换成实际坐标,并保存在结构数组 Pointxy 中
            PointXyz[PushNumb].x = m_xStart + blc * point.x;
            PointXyz[PushNumb].y = m_yStart + blc * (m_hScreen - point.y);
            if(PushNumb = =0)                    //如果是第一次按下左鼠标键
            {
                SetCapture();                    //捕捉鼠标
                mPointOrign = point;             //原点等于鼠标按中点
                mPointOld = point;               //将鼠标按中点作为鼠标移动时的上一个点
            }
            else                                 //二次以上按下鼠标左键时
            {
                mPointOrign = mPointOld;         //将上一个点设为原点
                mPointOld = point;               //将鼠标按中点作为鼠标移动时的上一个点
            }
            PushNumb++;                          //记录按下鼠标左键的次数
        }
    }
```

以上代码中,利用结构数组 PointXyz 来存储已经按下的连续直线顶点的实际坐标,每次按下鼠标左键,得到一个按下点的像素坐标后,将像素坐标转成实际坐标存在 PointXyz 数组中,用 PushNumb 记录按下鼠标左键的数目(即已有的顶点数目)。同绘制直线时类似,利用 mPointOrign 和 mPointOld 来实现橡皮线的拖动,mPointOrign 记录上一个顶点的坐标,mPointOld 记录上一个拖动点的坐标。

结构指针 PointXyz 在 CDrawView 类中定义:

```
PointStruct * PointXyz;
```

PointStruct 结构在头文件 drawdoc.h 中定义(见 5.1.3 小节)。在 CDrawView 类的构造函数中动态分配结构数组:

```
CDrawView::CDrawView()
{
    …… //省略以上代码
    PointXyz = new PointStruct[3000]; //动态存储连续直线顶点的结构数组
}
```

在析构函数中删除动态分配的结构数组:

```
CDrawView::~CDrawView()
{
    …… //省略以上代码
    delete PointXyz;
}
```

以上定义的结构数组 PointXyz,最大可处理 3000 个顶点的连续直线。

(2) 修改消息处理函数 OnMouseMove

在实现文件 drawview.cpp 中的消息处理函数 OnMouseMove 中，加入如下代码：

```
void CDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    ......//省略以上代码
    else if((m_DrawCurrent==2||m_DrawCurrent==3)&&PushNumb>=1)
    //如果正在绘制连续直线或多边形区域,并且已经按下第一点
    {
        if(mPointOld!=point) //如果当前鼠标移动点与上一个移动点不同,将上个顶点到上个
                             //移动点的连线擦除,绘制从上个顶点到当前鼠标移动点的连线
        {
            ddd.MoveTo(mPointOrign);
            ddd.LineTo(mPointOld);          //擦除原来的直线
            ddd.MoveTo(mPointOrign);
            ddd.LineTo(point);              //绘制上个顶点到当前鼠标移动点的直线
            mPointOld=point;
        }
    }
}
```

以上实现代码与交互绘制直线操作时类似，其操作含义可参考进行直线绘制时的代码。

(3) 修改消息处理函数 OnRButtonDown

在连续直线或多边形区域的绘制过程中，按下鼠标右键时，会保存交互绘制的连续直线或多边形区域并对它进行重画，完成交互绘制连续直线或多边形区域的功能。在实现文件 drawview.cpp 中的函数 OnRButtonDown 中，加入如下代码：

```
void CDrawView::OnRButtonDown(UINT nFlags, CPoint point)
{
    ......//省略以上代码
    else if((m_DrawCurrent==2||m_DrawCurrent==3)&&PushNumb>0)
    //正在绘制连续直线或多边形区域操作,并且已经按下了一次以上鼠标左键
    {
        //擦除最后一段拖动的橡皮线
        ddd.MoveTo(mPointOrign);
        ddd.LineTo(mPointOld);
        if(m_DrawCurrent==2&&PushNumb>1) //如果正在绘制连续直线,并且顶点数大于2
        {
            id_only=pDoc->GetGraphID(2); //得到要增加连续直线或多边形区域的唯一识别号
            //以下保存交互绘制的连续直线,并重新绘制连续直线
            pDoc->AddPLine(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,
```

```
                id_only,PushNumb,PointXyz,0) - > Draw(&ddd,0,0,m_bColor);
            ReleaseCapture();                    //释放捕捉的鼠标
        }
        else if(m_DrawCurrent = = 3&&PushNumb > 2) //如果正在绘制多边形区域,并且顶点数大于 2
        {
            id_only = pDoc - > GetGraphID(2);     //得到要增加多边形区域的唯一识别号
            //以下保存交互绘制的多边形区域,并重新绘制多边形区域
            pDoc - > AddPLine(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
                PushNumb,PointXyz,1) - > Draw(&ddd,0,0,m_bColor);
            ReleaseCapture();
        }
        PushNumb = 0;          //按下鼠标左键的次数为 0
        ReleaseCapture();      //释放捕捉的鼠标
    }
}
```

以上代码中,首先将拖动的橡皮线擦掉,如果绘制的是连续直线且按下了二次以上鼠标左键(一个顶点不能形成一条直线),或者绘制的是多边形区域且按下了三次以上鼠标左键,调用 AddPline 函数创建一个 CPline 对象,并保存了指向这个对象的指针,并对连续直线(或多边形区域)进行重画,最后,完成本次交互绘制操作并释放捕捉的鼠标。

至此,完成了鼠标交互绘制连续直线和多边形区域的功能,编译并执行应用程序 Draw,可得到如图 6-3 的运行界面。

图 6-3　用鼠标绘制连续直线和多边形

以上讨论了连续直线或多边形区域的鼠标绘制方法。读者可以按照这个实现过程编制出其他绘制步骤和操作规则的程序来。需要说明的是,目前实现的交互绘制方法,在开始绘制后不能对操作进行取消,用户可加入别的鼠标控制(如双击右键),实现能够进行放弃操作的鼠标交互绘制功能。

6.3.4　绘制圆和圆形区域

本小节讨论圆和圆形圆的鼠标交互绘制问题。在屏幕上直接用鼠标绘制圆所用的方法很

多,如圆心及半径法、三点法(点中三点确定一个圆),两点法(点中两点作为直径确定一个圆)等方法。本小节以圆心半径法实现交互绘制圆和圆形区域的功能,用圆心半径法画圆的操作步骤是:用鼠标左键点中一点作为圆心,然后当鼠标移动时,就会拖动一个以按下点为圆心,以第一个按下点和鼠标目前的移动点之间的距离为半径的圆移动。在这种情况下,按下鼠标左键会完成一个圆的绘制操作,而按下鼠标右键就会擦除拖动的圆而放弃绘制操作。

为了完成圆(或圆形区域)的鼠标交互绘制功能,同样需要对视图类中的三个鼠标消息处理函数进行修改。

(1) 修改消息处理函数 OnLButtonDown

在实现文件 drawview.cpp 的消息处理函数 OnLButtonDown 中,加入如下代码:

```
void CDrawView::OnLButtonDown(UINT nFlags,CPoint point)
{
    float r;
    float xx1,yy1,xx2,yy2;
    int x1,x2,y1,y2,Lb,Index;
    BOOL m_Fill;
    CDrawDoc * pDoc = GetDocument(); //得到指向当前文档对象的指针
    CClientDC ht(this);
    ……//省略以上代码
    else if(m_DrawCurrent==4||m_DrawCurrent==5) //如果正在绘制圆或者圆形区域
    {
      if(PushNumb==0)              //如果是第一次按下左鼠标键
      {
            mPointOrign=point;     //原点等于按中点
            mPointOld=point;       //上一个点也等于按中点
            PushNumb++;            //按下鼠标左键的次数加1
            SetCapture();          //捕捉鼠标
      }
      else                         //如果二次以上按下鼠标左键
      {
          r=(float)sqrt(pow((float)(mPointOrign.x-mPointOld.x),2)+
                pow((float)(mPointOrign.y-mPointOld.y),2));   //计算得到圆的半径
          VPtoDP(mPointOrign.x,mPointOrign.y,&xx1,&yy1);      //得到圆心的实际坐标
          if(m_DrawCurrent==4)   //如果是在绘制圆
                m_Fill=0;
          else                         //如果是在绘制圆形区域
                m_Fill=1;
          id_only=pDoc->GetGraphID(3);                        //得到要增加圆的唯一识别号
          pDoc->AddCircle(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
                xx1,yy1,r,m_Fill)->Draw(&ht,0,0,m_bColor); //保存圆并重画圆
          PushNumb=0;      //鼠标按键次数为0,重新进行圆的交互绘制操作
```

```
                ReleaseCapture();  //释放捕捉的鼠标
            }
        }
    }
```

在以上代码中,计算得到了圆心和半径的实际坐标和长度,通过调用文档类的成员函数 GetCircleId 来得到交互绘制得到的的圆的唯一识别号,然后调用文档类的 AddCircle 函数创建并初始化一个 CCircle 对象,并记录下指向这个 CCircle 对象的指针,完成本次交互绘制圆的操作。

(2) 修改消息处理函数 OnMouseMove

为了在鼠标交互绘制圆的过程中能够拖动橡皮圆,在实现文件 drawview.cpp 中的消息处理函数 OnMouseMove 中,加入如下代码:

```
void CDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    …… //省略以上代码
    else if((m_DrawCurrent==4||m_DrawCurrent==5)&&PushNumb>=1)
    //如果正在绘制圆或圆形区域,并且已经按下圆心点
    {
        if(mPointOld!=point)   //如果当前鼠标移动点与上一个移动点不同,将经过原来移动
                               //点的圆擦掉,绘制经过当前点的圆
        {
            if(m_DrawCurrent==4)      //如果是普通圆,设置不填充状态
                ddd.SelectStockObject(NULL_BRUSH);
            r=(int)sqrt(pow((float)(mPointOrign.x-mPointOld.x),2)+
                pow((float)(mPointOrign.y-mPointOld.y),2));      //得到上一个圆的半径
            ddd.Ellipse(mPointOrign.x-r,mPointOrign.y-r,mPointOrign.x+
                r,mPointOrign.y+r);      //重画而擦除上一个圆
            r=(int)sqrt(pow((long)(mPointOrign.x-point.x),2)+
                pow((float)(mPointOrign.y-point.y),2));  //得到当前移动点所在圆的半径
            ddd.Ellipse(mPointOrign.x-r,mPointOrign.y-r,mPointOrign.x+r,
                mPointOrign.y+r);      //绘制一个经过当前鼠标移动的圆
            mPointOld=point;           //记录当前鼠标移动点作为下一次移动时的上一点
        }
    }
}
```

(3) 修改消息处理函数 OnRButtonDown

为了能够在交互绘制圆或圆形区域的过程中,能够进行放弃操作,在实现文件 drawview.cpp 中的消息处理函数 OnRButtonDown 中,加入如下的代码:

```
void CDrawView::OnRButtonDown(UINT nFlags,CPoint point)
{
    ……//省略以上代码
    else if((m_DrawCurrent==4||m_DrawCurrent==5)&&PushNumb>0)
    //如果正在绘制圆或圆形区域,并且已经按下第一点
    {
        if(m_DrawCurrent==4)                    //如果是圆,设置不填充状态
            ddd.SelectStockObject(NULL_BRUSH);
        r=(int)sqrt(pow((float)(mPointOrign.x-mPointOld.x),2)+
            pow((float)(mPointOrign.y-mPointOld.y),2));//得到上一个圆的半径
        //重画并擦除经过上一个点的圆
        ddd.Ellipse(mPointOrign.x-r,mPointOrign.y-r,mPointOrign.x+r,mPointOrign.y+r);
        ReleaseCapture();                       //释放捕捉的鼠标
        PushNumb=0;                             //按下鼠标左键的次数为0
    }
}
```

至此,实现了鼠标交互绘制圆或圆形区域的功能,对应用程序 Draw 编译后运行,得到如图6-4所示的运行界面。

图6-4　用鼠标绘制圆和圆形区域

6.3.5　绘制圆弧

本小节来讨论实现鼠标交互绘制圆弧的功能。用鼠标直接在屏幕上绘制圆弧有很多种方法,如圆心起点终点法(首先确定圆心,然后确定一点作为弧的起点,再确定一点与圆心的连线与圆的交点作为弧的结束点,此方法与讨论过的作圆的方法类似)、三点法(顺序的三个点固定一条圆弧)等。在6.3.4小节圆的交互绘制操作中,用的是圆心半径法,本小节介绍三点法绘制一条圆弧的方法。用三点法绘制一条圆弧,具体实现如下:用鼠标左键点中一个点作为圆弧的起点,再用鼠标左键点中一个点作为圆弧的中间经过点,此时移动鼠标时,就会在屏幕拖动一个经过当前鼠标移动点和前两个按下点的圆弧,如果再按下鼠标左键,就完成了一条圆弧的绘制操作。不论是按下圆弧的第一点后还是第二点后,按下鼠标右键就会放弃圆弧的绘制操作(如果鼠标左键已经按下二次,会擦除拖动的弧)。

(1) 由三点计算圆弧图形要素的方法

三点法绘制一条圆弧时,圆弧的起点是第一个点(x1,y1),经过第二个点(x2,y2)并到达第三个点(x3,y3)。为了圆弧计算和绘制时的方便,一般采用圆心、半径和起终弧度法来记录圆弧的几何特征,这就需要把由三个点确定的圆弧的集合特征要素计算出来。

圆心和半径

把三个点按顺序连成直线,就得到两条直线段,经过这两条直线的中点做这两条直线的垂线,两条垂线的交点就是圆心的位置(图 6-5)。在用程序实现这个操作过程时,必须考虑两条垂线是否为竖直线。

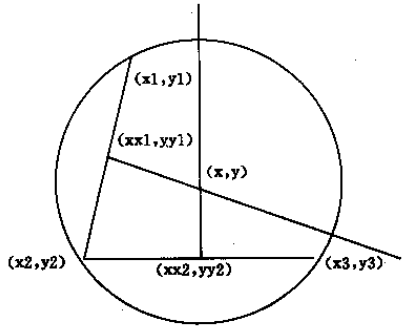


图 6-5 由三点计算圆心及半径

圆弧的起始和结束弧度

有了圆心坐标(x,y)和半径 r,就能够得到三个点相对于圆心的弧度,因为画这条圆弧时不一定是按逆时针方向,所以要经过判断才能得到起始弧度,具体方法是:

从第一个点由逆时针方向到第三个点连成圆弧,如果第二个点在这段圆弧上,这段圆弧就是逆时针方向绘制的,第一点和第三点的弧度分别是这条圆弧的起终弧度,相反,第三点和第一点的弧度分别是圆弧的起终弧度。

函数实现

为了计算得到三个点确定的圆弧的几何要素,在视图类 CDrawView 中定义了一个函数:

```
protected:
BOOL jsarc(CPoint p1,CPoint p2,CPoint p3,float * CircleX,float * CircleY,
    float * CircleR,float * Angle1,float * Angle2);
```

jsarc 函数计算得到由三点(p1,p2,p3)确定的圆弧的几何要素,通过指针变量返回圆弧的圆心(* CircleX, * CircleY)、半径(* CircleR)和起终弧度(* Angle1, * Angle2),计算成功时函数返回 TRUE,否则返回 FALSE。在实现文件 drawview.cpp 中加入函数的实现代码:

```
BOOL CDrawView::jsarc(CPoint p1,CPoint p2,CPoint p3,float * CircleX,float * CircleY,
    float * CircleR,float * Angle1,float * Angle2)
```

```
{
    float an1,an2,an3, k1,k2;;
    float x1,y1,x2,y2,x3,y3,xx1,xx2,yy1,yy2,xx,yy,rr;
    //将三个点的坐标赋给浮点变量
    x1 = (float)p1.x;y1 = (float)p1.y;x2 = (float)p2.x;
    y2 = (float)p2.y;x3 = (float)p3.x;y3 = (float)p3.y;
    if(x1 == x2&&y1 == y2||x1 == x3&&y1 == y3||x2 == x3&&y2 == y3) //如果有两个点的坐标相同
        return FALSE; //操作不成功,返回 FALSE
    //得到顺三点方向的两条直线段中点的坐标
    xx1 = (x1 + x2)/2; yy1 = (y1 + y2)/2;
    xx2 = (x2 + x3)/2; yy2 = (y2 + y3)/2;
    if(fabs(y2 - y1) > 0.5)                    //如果第一条直线段的垂线不是竖直线
        k1 = -(x2 - x1)/(y2 - y1);             //得到第一条直线段垂线的斜率
    if(fabs(y3 - y2) > 0.5)                    //如果第二条直线段的垂线不是竖直线
        k2 = -(x3 - x2)/(y3 - y2);             //得到第二条直线段垂线的斜率
    if(k1 = = k2&&k1 = = 0.0)
        return 0;
    if(fabs(y2 - y1) < 0.5)      //如果第一条直线段是水平线(即第一条直线段的垂线是竖直线)
    {
        xx = xx1;
        if(fabs(y3 - y2) < 0.5)            //如果第二条直线段也是水平线
        {
            yy = y2 + 10000;               //给定圆弧的半径为 10000
            rr = 10000;
        }
        else              // 如果第二条直线段不是平行线(即第二条直线段的垂线不是竖直线)
        {
            yy = yy2 + k2 * (xx2 - xx); //求得圆心的纵坐标
            rr = yy - y2;
            if(rr < 0)
                rr = -rr;                  //得到圆的半径
        }
    }
    else if(fabs(y2 - y3) < 0.5)           //如果第二条直线是水平线,而第一条直线不是水平线
    {
        yy = yy1 + k1 * (xx1 - xx);        //解得圆心的 y 坐标
        rr = yy - y2;
        if(rr < 0)
            rr = -rr;                      //得到圆的半径
    }
    else   //如果两条直线段都不是水平线(即两条直线段的垂线都不是竖直线)
    {
        if(k1 = = k2)                      //如果两条直线段平行(即三个点在一条直线上)
```

```
        {
            //以下假定这个圆弧半径为 10000,求得圆心坐标
            xx = x2 + (float)(10000/sqrt(1 + k1 * k1));                    //圆心横坐标
            yy = y2 + (float)(10000/sqrt((1 + k1 * k1)/(k1 * k1)));        //圆心纵坐标
            rr = 10000;                //圆的半径
        }
        else //如果两条直线段不平行
        {
            //解得两条直线段的垂线的交点,即圆心的坐标
            xx = (yy2 - yy1 + k1 * xx1 - k2 * xx2)/(k1 - k2);              //圆心横坐标
            yy = yy1 + k1 * (xx - xx1);                                    //圆心纵坐标
            rr = (float)sqrt((xx - x1) * (xx - x1) + (yy - y1) * (yy - y1));   //圆的半径
        }
    }
    //将圆心和半径都转成实际坐标
    * CircleX = m_xStart + blc * xx;
    * CircleY = m_yStart + blc * (m_hScreen - yy) * blc;
    * CircleR = blc * rr;
    an1 = (float)acos((x1 - xx)/rr);      //得到第一个点相对于圆心的弧度
    if(y1 - yy > 0)
        an1 = (float)(pi * 2 - an1);
    an2 = (float)acos((x2 - xx)/rr);      //得到第二个点相对于圆心的弧度
    if(y2 - yy > 0)
        an2 = (float)(pi * 2 - an2);
    an3 = (float)acos((x3 - xx)/rr);      //得到第三个点相对于圆心的弧度
    if(y3 - yy > 0)
        an3 = (float)(pi * 2 - an3);
    if(an2 > an1&&an2 < an1 + pi||an2 < an1&&an2 + pi < an1)      //如果圆弧是逆时针方向画的
    {
        * Angle1 = an1;   //得到圆弧的起始弧度
        * Angle2 = an3;   //得到圆弧的结束弧度
    }
    else                  //如果圆弧是顺时针方向画的
    {
        * Angle1 = an3;   //得到圆弧的起始弧度
        * Angle2 = an1;   //得到圆弧的结束弧度
    }
    return TRUE;          //操作成功
}
```

以下是 jsarc 函数的主要操作步骤：

① 对三个点进行检查，检查是否有二个点坐标相同。如果有两个点坐标相同，就以 FALSE 退出函数。

② 得到两条直线段的中点坐标：(xx1，yy1)和(xx2，yy2)。

③ 计算两条直线段垂线的斜率：k1 和 k2。计算斜率时必须检查除数是否为 0(即垂线是否竖直)，分别利用斜率和两个直线段的中点，就可以得到两条直线垂线的方程。

④ 如果第一条直线段的垂线是竖直线时，分为如下两种情况进行计算：一种情况是第二条垂线也是竖直线，此时三点一条线，假定一个较大的半径近似解得圆心坐标；另外一种情况是第二条垂线不是竖直线，此时已知圆心 x 坐标，解得圆心 y 坐标和半径 r。

⑤ 如果第二条垂线是竖直线而第一条直线不是竖直线，与④相同的方法得到圆的圆心和半径。

⑥ 如果两条垂线都不是竖直线时，分如下两种情况进行计算：一种情况是两条垂线斜率相等，此时三点一条线，可以通过假定一个较大的半径近似解得圆心坐标，另外一种情况是两条垂线斜率不等，由两条直线的方程得到圆心坐标和半径大小。

⑦ 根据三个点相对于圆心的弧度，计算确定圆弧的起始和结束弧度。注意，在计算起终弧度时，需要判断圆弧的绘制方向(逆时针还是顺时针方向)。

如果三个点在一条直线上，那么通过这三点的圆的半径无限大。为了解题的方便，当三点在一条直线上时，在以上 jsarc 函数代码中，给圆设置一个比较大的半径(在以上代码中，假定半径大小为 10000)，近似计算出圆心坐标。

(2) 修改消息处理函数 OnLButtonDown

为了实现圆弧的交互绘制功能，在实现文件 drawview.cpp 中的消息处理函数 OnLButtonDown 中，加入如下代码：

```
void CDrawView::OnLButtonDown(UINT nFlags,CPoint point)
{
    …… //省略以上代码
    else if(m_DrawCurrent==6)                    //正在绘制圆弧
    {
        if(PushNumb==0)                          //第一次按下鼠标左键
        {
            SetCapture();
            DrawArcYes=0;
            mPointOrign1=point;                  //记录圆弧第一点
            PushNumb++;
        }
        else if(PushNumb==1)                     //第二次按下鼠标左键
        {
            mPointOrign=point;                   //记录下圆弧的第二点
            PushNumb++;
        }
        else                                     //第三次按下鼠标键
        {
            jsarc(mPointOrign1,mPointOrign,point,&m_CircleX,&m_CircleY,&m_CircleR,
```

```
                &m_Angle1,&m_Angle2);      //计算圆弧的几何要素
            id_only = pDoc->GetGraphID(4);  //得到要增加圆弧的唯一识别号
            pDoc->AddArc(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
              m_CircleX,m_CircleY,m_CircleR,m_Fill,m_Angle1,m_Angle2)
              ->Draw(&dt,0,0,m_bColor);   //增加一个圆弧并重新绘制圆弧
            PushNumb = 0;                 //按下鼠标左键次数为0,重新进行圆弧的绘制
            ReleaseCapture();             //释放捕捉的鼠标
        }
    }
}
```

在以上代码中,第一次按下鼠标左键时,将按中点的坐标记录在变量 mPointOrign1 中。第二次按下鼠标左键时,将按中点的坐标记录在 mPointOrign 中。第三次按下鼠标左键时,调用 jsarc 函数得到计算圆弧的几何要素,然后调用 AddArc 函数创建并初始化一个 CArc 对象并保存指向这个对象的指针,通过返回的指针重画圆弧。在以上代码中,使用的几个成员变量视图类 CDrawView 中定义:

```
private:
    float m_CircleX,m_CircleY,m_CircleR,m_Angle1,m_Angle2;
```

(3) 修改消息处理函数 OnMouseMove

为了在交互绘制圆弧的过程中,实现拖动圆弧的功能,在实现文件 drawview.cpp 中的消息处理函数 OnMouseMove 中,加入如下代码:

```
void CDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    …… //省略以上代码
    else if(m_DrawCurrent==6)              //正在绘制圆弧操作
    {
        if(PushNumb==2&&mPointOld!=point) //如果按下二个点且当前鼠标移动点与上个移动
                                          //点不同
        {
            if(DrawArcYes>0)              //如果已经在视图中绘制了 m_Arc1 对象存储的圆弧
                m_Arc1.Draw(&ddd,1,0,m_bColor);  //重画以擦除圆弧
            pb=jsarc(mPointOrign1,mPointOrign,point,&m_CircleX,&m_CircleY,&m_CircleR,
                &m_Angle1,&m_Angle2);     //计算经过当前鼠标移动点的圆弧的几何要素
            if(pb)  //如果能计算得到圆弧的几何要素,初始化 m_Arc1 对象并绘制这个圆弧
            {
                m_Arc1.Init(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,0,
                    m_CircleX,m_CircleY,m_CircleR,0,m_Angle1,m_Angle2);
                m_Arc1.Draw(&ddd,m_hScreen,1,0,m_bColor);
            }
            DrawArcYes=pb;                //判断这次是否绘制了 m_Arc1 对象
```

```
            mPointOld = point;
        }
    }
}
```

在屏幕上拖动圆弧时，与前面讲到的直线和圆等不同。在绘制直线过程中第一次拖动直线时，函数虽然也是首先在屏幕上擦除原有的直线，但擦除的第一条直线起终点相同，所以不会在屏幕上留下一条多余的直线。但用三点绘制圆弧时不一样，第一次擦除时所画的圆弧会留在屏幕上，所以定义了一个变量 DrawArcYes 来控制擦除操作。只有上一次在屏幕上已经画下圆弧时，才进行擦除操作。

变量 DrawArcYes 和圆弧对象 m_Arc1 在视图类 CDrawView 类中定义：

```
private:
    CArc m_Arc1;        //定义一个圆弧类指针
    BOOL DrawArcYes;
```

为了对一个圆弧对象的成员变量进行初始化，在圆弧类 CArc 中，定义一个初始化函数：

```
public:
void Init(short ColorPen, short ColorBrush, short LineWide, short LineType, short Layer, BOOL Delete, float CircleX, float CircleY, float CircleR, BOOL Fill, float Angle1, float Angle2);
```

Init 函数能够对圆弧对象的成员变量进行初始化。在实现文件 drawdoc.cpp 中加入函数的实现代码：

```
void CArc::Init(short ColorPen, short ColorBrush, short LineWide, short LineType, short Layer, BOOL Delete,
    float CircleX, float CircleY, float CircleR, BOOL Fill, float Angle1, float Angle2)
{
    m_ColorPen = ColorPen;
    m_ColorBrush = ColorBrush;
    m_LineWide = LineWide;
    m_LineType = LineType;
    m_Layer = Layer;
    m_CircleX = CircleX;
    m_CircleY = CircleY;
    m_CircleR = CircleR;
    b_Fill = Fill;
    m_Angle1 = Angle1;
    m_Angle2 = Angle2;
    b_Delete = Delete;
}
```

(4) 修改消息处理函数 OnRButtonDown

为了在进行鼠标交互绘制圆弧的过程中能够实现放弃操作，在实现文件 drawview.cpp 中的消息处理函数 OnRButtonDown 中，加入以下代码：

```
void CDrawView::OnRButtonDown(UINT nFlags,CPoint point)
{
    ......//省略以上代码
    else if(m_DrawCurrent==6&&PushNumb>0)    //正在绘制圆弧并且按下了一次以上鼠标左键
    {
        if(PushNumb==2)                      //已经按下两个点,正在拖动圆弧
            m_Arc1.Draw(&ddd,1,0,m_bColor); //擦掉拖动的圆弧
        PushNumb=0;
        ReleaseCapture();                    //释放捕捉的鼠标
    }
}
```

至此,完成了用鼠标交互绘制圆弧的操作功能,编译应用程序 Draw 后运行,就可得到如图 6-6所示的运行界面。

图 6-6　鼠标绘制圆弧

6.4　交互绘制标注文本

6.3 节实现了用鼠标交互绘制直线、连续直线及多边形区域、圆及圆形区域、圆弧等图形元素的操作功能。这些图形元素的交互绘制,都可以只使用鼠标来操作实现,而对于标注文本的绘制,因为涉及到标注文本内容的输入等操作,单靠鼠标不能完成所有的绘制过程。本节将介绍利用鼠标和无模式对话框相结合,来完成标注文本的交互绘制功能。

6.4.1　对话框简介

对话框是 Windows 应用程序与用户交换信息的重要手段,一个没有对话框的 Windows 应用程序是不可想象的。在程序的运行过程中,对话框可用来捕捉用户输入的数据资料,达到管理数据并与用户交换信息的目的。在对话框中,可以有各种的对话框控制,如编辑框、按钮、列表框、组合框、滚动条、检查框等。在对话框中,可以操纵这些控制并接受和处理各种控制发出的窗口消息。对话框是一种特殊的窗口。

VC++ 为对话框编程提供了强大的工具支持,第 2 章讨论过的 ResourceView 及调用的 App-

Studio工具可以可视化的编辑对话框资源,ClassWizard可以用来方便地管理对话框中的消息处理函数、可重载的虚函数及映射变量。使用VC++的组件库,可对卡片式对话框进行编程。

本小节对对话框的一些基本情况进行介绍。对话框可以分为两种类型:模式对话框和无模式对话框,有些教材中也叫做模型对话框和非模型对话框。两者是通过对话框在执行时的情况不同划分的,以下讨论这两种对话框的基本特点。

(1) 模式对话框

模式对话框最显著的特点是,这个对话框一旦启动,就完全控制了应用程序的流程,在这个对话框退出之前,所在的应用程序不能执行其他的操作。读者在使用Windows应用程序时可能会注意到,有些对话框出现在屏幕上时,就只能进行对话框内的操作,而不能进行对话框外的应用程序的任何操作,只有在对话框退出后,才能进行应用程序其他的功能操作,这种对话框就是模式对话框。模式对话框中一般有"OK"或"Cancle"两个按钮(中文一般是"确定"和"放弃"),用来保存或放弃对话框中修改的内容,并退出对话框。

(2) 无模式对话框

无模式对话框启动后,并不完全控制应用程序的流程。它可以一直显示在屏幕上,用户可以进行应用程序的其他操作,当需要使用它时,用户可以点中或用其他的指令方式把它激活,使之又控制程序的流程。

两种对话框的资源是完全一样的,都可以用Resourse及其相应工具来可视化创建,用ClassWizard辅助编程,但是两种对话框在创建和使用上存在差别。本节将利用无模式对话框来完成标注文本的交互绘制。

6.4.2 交互绘制标注文本的过程

交互绘制标注文本的操作过程如下:当运行应用程序Draw的"鼠标绘图"下的"标注文本"菜单项后,用鼠标左键在视图中点中标注文本的起点,系统就会弹出一个对话框(见图6-3),对话框中有一个编辑框,当在编辑框中输入标注文本的内容时,会马上在视图上绘出标注文本。当按中对话框的"确定"按钮时,会完成本次文本标注工作并退出对话框,系统会保存绘制的标注文本对象。当按中"放弃"按钮时,系统会放弃本次绘制标注文本操作并退出对话框。

6.4.3 创建对话框资源

为了完成6.4.2小节讨论的功能,首先需要定义一个对话框资源。对话框资源如图6-7所示,编辑对话框资源的具体操作不再讨论,操作完成后可以在资源文件draw.rc中找到对话框的定义代码:

图6-7 IDD_WRITE_TEXT对话框资源

```
IDD_WRITE_TEXT DIALOG DISCARDABLE 10, 10, 200, 38
STYLE WS_POPUP | WS_VISIBLE | WS_CLIPSIBLINGS | WS_CAPTION | WS_SYSMENU
CAPTION "标注信息窗口"
FONT 8, "System"
BEGIN
    DEFPUSHBUTTON "确定",IDOK,9,19,37,13
    PUSHBUTTON"放弃",IDCANCEL,65,19,37,13
    LTEXT "请输入标注的信息",IDC_STATIC,8,4,83,12
    EDITTEXT IDC_TEXT_EDIT1,92,2,100,12,ES_AUTOHSCROLL
END
```

在资源 ID 号定义文件 resource.h 中，可以找到对话框资源中的 ID 号的定义：

```
#define IDD_WRITE_TEXT      130          //对话框资源的 ID 值
#define IDC_TEXT_EDIT1      602          //编辑框 IDC_TEXT_EDIT1 的 ID 值
```

6.4.4　实现文本标注对话框类

在定义了对话框资源后，可以调用 ClassWizard 产生一个对话框类 TextWriteDlg(增加类的操作步骤见 2.3.3 小节)，对话框类的头文件是 textwdlg.h，实现文件 textwdlg.cpp。在创建的对话框类的基础上，进行实际的开发工作。

(1) 增加成员变量

在对话框类 TextWriteDlg 中增加如下成员变量：

```
public:
    CDrawView * m_pView;      //用来记录指向调用对话框的视图对象的指针
```

m_pView 用来记录指向要标注文本的的视图对象的指针。

(2) 重载构造函数

在对话框类 TextWriteDlg 中，重载一个构造函数：

```
TextWriteDlg(CWnd * pParent,CString Text);
```

在实现文件 textwdlg.CPP 中，增加构造函数的实现代码：

```
TextWriteDlg::TextWriteDlg(CWnd * pParent,CString Text)
    : CDialog(TextWriteDlg::IDD, pParent)
{
    m_pView=(CDrawView * )pParent;          //记录传入的指针
    m_Text=Text;                            //标注文本的内容
}
```

变量 m_Text 是一个映射变量，用来与对话框中输入标注文本内容的编辑框交换数据，在下述(3)中定义。

(3) 增加映射变量

在对话框类 TextWriteDlg 中增加一个映射变量,用来与对话框中的输入标注文本内容的编辑框交换数据。用 ClassWizard 增加映射变量的方法在 2.5.6 小节已做介绍,增加映射变量后,在 TextWriteDlg 类中增加了变量的定义:

```
//{{AFX_DATA(TextWriteDlg)
enum { IDD = ID_WRITE_TEXT };
CString m_Text;
//}}AFX_DATA
```

在构造函数中加入了变量的初始化代码:

```
TextWriteDlg::TextWriteDlg(CWnd* pParent /* = NULL */)
    : CDialog(TextWriteDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(TextWriteDlg)
    m_Text = _T("");
    //}}AFX_DATA_INIT
}
```

在实现文件 textwdlg.cpp 中的 DoDataExchange 函数中,加入了数据交换机制:

```
void TextWriteDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(TextWriteDlg)
    DDX_Text(pDX, IDC_TEXT_EDIT1, m_Text);
    DDV_MaxChars(pDX, m_Text, 300);
    //}}AFX_DATA_MAP
}
```

DDX_Text 宏完成对话框控制与映射变量 m_Text 间的数据交换,DDV_MaxChars 宏设置最大能够输入 300 个字节长度的标注文本。

(4) 增加消息处理函数

在对话框类 TextWriteDlg 中,为"确定"、"放弃"按钮、IDC_TEXT_EDIT1 编辑框以及 WM_INITDIALOG 窗口消息定义四个消息处理函数,增加操作完成后,在 TextWriteDlg 类中增加了消息处理函数的定义:

```
//{{AFX_MSG(TextWriteDlg)
afx_msg void OnChangeText();
virtual void OnOK();
virtual void OnCancel();
virtual BOOL OnInitDialog();
```

```
//}}AFX_MSG
```

在实现文件 textwdlg.cpp 中,增加了消息映射机制:

```
BEGIN_MESSAGE_MAP(TextWriteDlg, CDialog)
    //{{AFX_MSG_MAP(TextWriteDlg)
    ON_EN_CHANGE(IDC_TEXT_EDIT1, OnChangeText)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

ON_EN_CHANGE 宏用来实现 EN_CHANGE 消息的函数映射,EN_CHANGE 消息是编辑框内的内容发生变化时发出的消息。

在实现文件 textwritedlg.cpp 中,对空的函数实现代码进行修改,得到修改后的函数实现代码如下:

```
void TextWriteDlg::OnChangeText()
{
    UpdateData();   //将控制(编辑框 IDC_TEXT_EDIT1)中的内容传递到映射变量
    m_pView->SendMessage(WM_COMMAND,ID_TEXT_MESSAGE);
}
```

OnChangeText 函数在对话框中编辑框 IDC_TEXT_EDIT1 内的内容修改时执行,通过 UpdateData 函数将编辑框内的内容存入成员变量,并向 p_View 指向的视图对象发送 ID_TEXT_MESSAGE 消息。

```
void TextWriteDlg::OnOK()
{
    UpdateData(); //将控制(编辑框 IDC_TEXT_EDIT1)中的内容传递到映射变量
    m_pView->SendMessage(WM_COMMAND,ID_TEXT_ONOK);
    CDialog::OnOK();
}
```

OnOk 函数在按中"确定"按钮时被执行,通过 UpdateData 函数将编辑框内的内容存入成员变量 m_text,并向 p_View 指向的视图对象发送 ID_TEXT_ONOK 消息,调用 CDialog::OnOK 函数退出对话框。

```
void TextWriteDlg::OnCancel()
{
    m_pView->SendMessage(WM_COMMAND,ID_TEXT_ONCANCEL);
    CDialog::OnCancel();
}
```

OnCancle 函数在按中"放弃"按钮时被执行,并向 p_View 指向的视图对象发送ID_TEXT_CANCEL 消息,调用 CDialog::OnCancle 函数退出对话框。

```
BOOL TextWriteDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
```

```
    CEdit * pedit = (CEdit * )GetDlgItem(IDC_TEXT_EDIT1);
    pedit - > SetFocus();        //进入对话框时,将输入焦点设置在编辑框内
    return TRUE;
}
```

OnInitDialog 函数是窗口消息 WM_INITDIALOG 的处理函数,在对话框启动时自动执行,也可以通过发送 WM_INITDIALOG 消息激活执行。在这个函数中,实现了将输入焦点设置到编辑框 IDC_TEXT_EDIT1 的功能,即进入对话框时,IDC_TEXT_EDIT1 编辑框处于激活输入状态。

(5) 定义消息

在(4)中实现的消息处理函数中发送了三个 COMMAND 型的消息,需要在 ID 号定义文件 resource.h 中手工定义这些消息。打开 resource.h 文件,在文件的最后加入以下代码:

```
# define ID_TEXT_MESSAGE          WM_USER + 7
# define ID_TEXT_ONOK             WM_USER + 8
# define ID_TEXT_ONCANCEL         WM_USER + 9
```

无模式对话框向视图对象发送这三种消息,可以在视图类中加入消息处理函数,完成相应的操作。

(6) 增加成员函数

在对话框类 TextWriteDlg 中,增加以下三个成员函数:

```
public:
    BOOL IsOpen();
    BOOL IsVisible();
    void Init(CString Text);
```

在实现文件 textwdlg.cpp 中,加入这三个函数的实现代码:

```
BOOL TextWriteDlg::IsOpen()   //检查对话框窗口是否建立
{
    return(m_hWnd? TRUE:FALSE);
}

BOOL TextWriteDlg::IsVisible()  //检查对话框窗口是否可见
{
    if(! IsOpen())              //如果没有建立窗口
        return FALSE;
    WINDOWPLACEMENT scrn;
    GetWindowPlacement(&scrn);
    if(scrn.showCmd = = SW_SHOW)   //窗口是否可见(如果成员变量 showCmd 等于 SW_SHOW,
                                   //则窗口可见)
```

```
        return TRUE;
    else
        return FALSE;
}

void TextWriteDlg::Init(CString Text)
{
    m_Text = Text;
}
```

IsOpen 函数用来检查对话框窗口是否建立;IsVisible 函数用来检查当前对话框窗口在屏幕上是否可见;Init 函数用来初始化对话框类的成员变量 m_Text。

以上就完成了文本标注对话框类的各项操作功能,在 6.4.5 小节中,我们将在视图类 CDrawView 的消息处理函数中调用这个对话框类,实现交互绘制标注文本的功能。

6.4.5　实现标注文本交互绘制功能

6.4.4 小节实现了用来标注文本的对话框类 TextWriteDlg。为了实现标注文本的交互绘制工作,需要修改视图类 CDrawView,增加与交互绘制标注文本有关的功能。

(1) 定义对话框类指针和其他成员变量

为了进行标注文本的交互绘制功能,在视图类 CDrawView 中定义一个对话框类对象指针和一些成员变量:

```
protected:
    TextWriteDlg * pTextDlg;        //标注文本的对话框类指针
    float m_FontHeight,m_FontWide,m_FontBetween,m_TextAngle,m_FontAngle;
    float m_TextX,m_TextY;          //标注文本的起点位置
    CString m_TextString;           //存储标注文本的字符串
    CText * m_Text1;                //定义一个标注文本类指针
```

在实现文件 drawview.cpp 中的视图类构造函数中,对这些变量进行初始化:

```
CDrawView::CDrawView()
{
    ......//省略以上代码
    pTextDlg = NULL;
    //以下设置初始字体大小
    m_FontHeight = 20;
    m_FontWide = 10;
    m_TextAngle = 0;
    m_FontAngle = 0;
    m_FontBetween = 1;
    m_TextString.Empty();
```

```
    m_Text1 = new CText();    //创建一个标注文本对象
}
```

在视图类 CDrawView 的析构函数中，删除动态创建的标注文本对象：

```
CDrawView::~CDrawView()
{
    ...//省略以上操作
    delete m_Text1;
}
```

(2) 开始文本标注

在 6.3 节实现了直线、连续直线、圆、圆弧等图形元素的交互绘制功能，现在来实现标注文本的交互绘制功能。当选择了“鼠标绘图”菜单下“标注文本”菜单项时，用鼠标左键点中视图屏幕，就以这个点为起点，调出 TextWriteDlg 对话框进行文本的输入工作。

为了能够得到标注起点并调出 TextWriteDlg 对话框，需对消息处理函数 OnLButtonDown 进行修改，在函数中加入如下代码：

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    ......//省略
    else if(m_DrawCurrent==7)          //正在进行标注文本的交互绘制操作
    {
        VPtoDP(point.x,point.y,&m_TextX,&m_TextY);    //得到按中点的实际坐标
        if(!pTextDlg)                  //如果标注文本对话框对象不存在，创建对话框对象
        {
            pTextDlg=new TextWriteDlg(this,m_TextString);// 创建标注文本对话框对象
            ASSERT(pTextDlg);
        }
        if(!(pTextDlg->IsOpen()))      //如果对话框窗口没有建立，建立对话框窗口
            pTextDlg->Create(ID_WRITE_TEXT,this);
        if(!(pTextDlg->IsVisible()))   //如果对话框窗口当前不可见
            pTextDlg->ShowWindow(SW_SHOW);    //使标注文本对话框可见
    }
}
```

(3) 增加消息处理函数

在 CDrawView 类中定义三个消息处理函数，对应于 6.3.3 小节中定义的三个命令消息 ID_TEXT_MESSAGE、IDOK、IDCANCEL。增加操作只能手工来实现，步骤如下：

① 在视图类 CDrawView 中，定义三个消息处理函数：

```
//{{AFX_MSG(CDrawView)
```

```
.....//省略以上代码
afx_msg void DrawText();              //在对话框编辑框中内容变化时被调用,在屏幕上绘制文本
afx_msg void DrawTextOnOk();          //按'确定'按钮退出标注文本对话框时被调用
afx_msg void DrawTextOnCancel();      //按'放弃'按钮退出标注文本对话框时被调用
//}}AFX_MSG
```

② 在实现文件 drawview.cpp 中,加入消息处理函数的消息映射机制:

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    .....//省略以上代码
    ON_COMMAND(ID_TEXT_MESSAGE,DrawText)
    ON_COMMAND(ID_TEXT_ONOK,DrawTextOnOk)
    ON_COMMAND(ID_TEXT_ONCANCEL,DrawTextOnCancel)
END_MESSAGE_MAP()
```

③ 在实现文件中 drawview.cpp 中,增加消息处理函数的实现代码:

```
void CDrawView::DrawText()
{
    CClientDC ht(this);                          //创建一个绘图对象
    m_TextString=pTextDlg->m_Text;               //从标注文本对话框中得到标注文本内容
    //对标注文本对象的参数进行初始化
    m_Text1->Init(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,m_TextX,m_TextY,
        m_TextAngle,m_FontAngle,m_FontHeight,m_FontWide,m_FontBetween,0,m_TextString);
    m_Text1->Draw(&ddd,0,0,m_bColor);            //绘制标注文本
}
```

消息处理函数 DrawText 由消息 ID_TEXT_MESSAGE 激发运行,用来完成在视图中绘制正在标注的文本,当在文本标注对话框的编辑框中修改标注文本内容时,这个消息处理函数被调用,将标注文本的内容绘制到屏幕上。函数的工作步骤为:得到标注文本的内容,用 CText 类的 Init 函数初始化标注文本对象 m_Text1 的成员变量,调用 CText 类的成员函数 Draw 对标注文本对象 m_Text1 进行绘制。

```
void CDrawView::DrawTextOnOk()
{
    CDrawDoc* pDoc = GetDocument();              //得到指向当前文档对象的指针
    CClientDC ht(this);                          //创建一个绘图对象
    int TextLong;
    PushNumb=0;
    m_TextString=pTextDlg->m_Text;               //得到标注文本内容
    TextLong=m_TextString.GetLength();           //标注文本的字符长度
    if(TextLong>0)                               //如果有标注文本
    {
        int id_only=pDoc->GetGraphID(5);         //得到要增加的标注文本对象的识别号
        pDoc->AddText(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
            m_TextX,m_TextY,m_TextAngle,m_FontAngle,m_FontHeight,m_FontWide,
```

```
            m_FontBetween,0,TextLong,m_TextString)->Draw(&ht,0,0,m_bColor);
        pTextDlg->m_Text.Empty();                    //清空字符串 m_Text 的内容
        pTextDlg->SendMessage(WM_INITDIALOG);//激发消息调用对话框的 OnInitDialog 函数
    }
}
```

消息处理函数 DrawTextOnOk 在文本标注对话框中按下“确定”按钮时被执行，实现在文档中增加一个标注文本对象的功能。在以上的函数代码中，首先得到填写在对话框中编辑框控制中的字符串，如果字符串长度大于 0，通过调用文档类的成员函数 GetTextId 来得到要增加标注文本对象的识别号，然后调用文档类的 AddArc 函数增加并初始化一个标注文本对象，并重画这个标注文本对象。

```
void CDrawView::DrawTextOnCancel()
{
    pTextDlg->m_Text.Empty();                    //清空标注文本对话框中编辑框中的内容
    pTextDlg->SendMessage(WM_INITDIALOG);
}
```

消息处理函数 DrawTextCancel 在文本标注对话框中按下“放弃”按钮时被执行，放弃当前的交互绘制标注文本的操作。

对应用程序 Draw 进行编译执行，可以得到如图 6-8 所示的运行界面。

图 6-8　文本的标注界面

以上实现了交互绘制标注文本的基本操作，如果要达到完美实用，还需要增加另外一些功能考虑（如在进行文本标注操作时，进行后退操作及放弃操作时，图形中的标注文本需要回退等），请读者在此程序框架基础上自己开发实现。

在作者所著的《Visual C++ 高级编程技术——开发实例剖析》一书中，详细介绍了模式和无模式对话框编程的技术，实现了标注文本的交互绘制功能和字体参数的设置功能，请读者参阅。

第 7 章　矢量图形系统的操作功能

对于一个基本矢量图形系统，需要具有一些基本的操作功能。本章将对矢量图形系统 Draw 的图形操作功能进行扩充，完成对图形进行重画、无级放缩、移动、恢复上屏、重画全屏等功能，并实现滚动视图。

7.1　增加图形操作菜单

像 6.2 节给应用程序 Draw 增加了一个“鼠标绘图”菜单一样，为实现图形的重画、放缩等功能，在应用程序 Draw 中增加一个“图形操作”菜单，它包含“图形重画”、“图形放大”、“图形移动”、“重画上屏”、“重画首屏”、“绘制全屏”等菜单项。

7.1.1　增加菜单资源

用 ResouceView 及其相关工具在应用程序 Draw 的 IDR _ DRAWTYPE 菜单资源中增加一个菜单，增加完成后，应用程序 Draw 源代码作如下修改：

① 在资源文件 draw.rc 中，加入了菜单资源的定义

```
IDR _ DRAWTYPE MENU PRELOAD DISCARDABLE
BEGIN

    ......//省略以上代码
    POPUP "C 图形操作"
    BEGIN
        MENUITEM "图形重画 \ tAlt + R",          ID _ GRAPH _ REDRAW
        MENUITEM "图形放大 \ tAlt + Z",          ID _ GRAPH _ ZOOM
        MENUITEM "图形移动 \ tAlt + P",          ID _ GRAPH _ PAN
        MENUITEM "重画上屏 \ tAlt + UP",         ID _ GRAPH _ UP
        MENUITEM "重画首屏 \ tAlt + HOME",       ID _ GRAPH _ FIRST
        MENUITEM "显示全图 \ tAlt + A",          ID _ GRAPH _ ALL
    END

    ..... //省略以下代码
END
```

② 在资源 ID 号定义文件 resource.h 中，加入了对新增菜单项 ID 号的定义：

```
# define ID _ GRAPH _ REDRAW        32779
# define ID _ GRAPH _ ZOOM          32780
# define ID _ GRAPH _ PAN           32781
```

```
# define ID_GRAPH_UP                32782
# define ID_GRAPH_FIRST             32783
# define ID_GRAPH_ALL               32784
```

7.1.2　增加消息处理函数

用 ClassWizard 为新增加的各菜单项在视图类 CDrawView 中建立消息处理函数，增加完成后，应用程序 Draw 源代码改动如下：

① 在视图类 CDrawView 中，加入消息处理函数的定义：

```
//{{AFX_MSG(CDrawView)
.....//省略以上代码
afx_msg void OnGraphRedraw();    //图形重画
afx_msg void OnGraphZoom();      //图形放大
afx_msg void OnGraphPan();       //图形移动
afx_msg void OnGraphUp();        //重画上屏图形
afx_msg void OnGraphFirst();     //重画首屏图形
afx_msg void OnGraphAll();       //重画全屏
//}}AFX_MSG
```

② 在实现文件 drawview.cpp 中，加入消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    ·····//省略以上代码
    ON_COMMAND(ID_GRAPH_REDRAW, OnGraphRedraw)
    ON_COMMAND(ID_GRAPH_ZOOM, OnGraphZoom)
    ON_COMMAND(ID_GRAPH_PAN, OnGraphPan)
    ON_COMMAND(ID_GRAPH_UP, OnGraphUp)
    ON_COMMAND(ID_GRAPH_FIRST, OnGraphFirst)
    ON_COMMAND(ID_GRAPH_ALL, OnGraphAll)
END_MESSAGE_MAP()
```

③ 在实现文件 drawdoc.cpp 中，加入空的消息处理函数函数：

```
void CDrawView::OnGraphRedraw()
{
}

void CDrawView::OnGraphZoom()
{
}

void CDrawView::OnGraphPan()
{
}
```

```
void CDrawView::OnGraphUp()
{
}

void CDrawView::OnGraphFirst()
{
}

void CDrawView::OnGraphAll()
{
}
```

对应用程序 Draw 经过编译后运行，会得到如图 7-1 所示的运行界面。以下各节，将对各个消息处理函数的实现代码进行修改，实现各种图形操作功能。

图 7-1　应用程序 Draw 的图形操作菜单

7.2　图形重画

本节讨论实现矢量图形系统的图形重画功能。所谓图形重画，就是把当前视图屏幕重新绘制一次，也就是让函数视图类的 OnDraw(CDC * pDC)重新执行一次。根据需要重画区域的不同，图形重画可以分为全屏重画和局部重画。

7.2.1　全屏重画

本小节讨论实现矢量图形系统的全屏重画功能。

(1) 激活视图重画的方法

要想使视图屏幕重新绘制，就要激活视图的绘制机制。有两种方法可以实现这个功能：

① 在视图类函数的实现代码中，用以下代码来激活重画机制：

```
CDrawDoc * pDoc = GetDocument();
pDoc - > UpdateAllViews(this);
```

UpdateAllViews 函数使属于当前文档对象的所有视图重画。如果应用程序采用的单文档结构，系统只有一个视图，执行此命令时使这个视图失效并重画，即激发视图类的 OnDraw 执行；如果应用程序采用的是多文档结构，一个文档对象可能具有多个视图，UpdateAllViews 可使属于这个文档对象的所有视图都重新绘制。

② 在视图类函数的实现代码中，用以下代码激活重画机制：

```
Invalidate();
```

Invalidate 函数使当前视图屏幕的客户区失效，从而激活图形重画。Invalidate 函数与 UpdateAllViews 函数的不同点是，前者只能使当前视图重画，而与其同文档的其他视图并不重画。

(2) 实现图形重画功能

要实现图形重画功能，需要对视图类 CDrawView 的 OnGraphRedraw 函数进行修改。修改后的代码如下：

```
void CDrawView::OnGraphRedraw()
{
    CDrawDoc * pDoc = GetDocument();        //得到指向当前文档对象的指针
    pDoc - > UpdateAllViews(this);          //使文档对象的所有视图重画
    Invalidate();
}
```

OnGraphRedraw 函数首先得到指向当前文档对象的指针，然后调用 UpdateAllViews 函数重画此文档对象所有的视图，参数 this 是指向当前视图对象的指针。

7.2.2　局部重画

在很多情况下，只希望当前视图屏幕的一部分区域重画，即局部重画，有两个相关的函数可以完成此功能。

(1) InvalidateRect 函数

函数的定义为：

```
void InvalidateRect(LPCRECT lpRect, BOOL bErase = TRUE);
```

InvalidateRect 函数可使视图中一个指定的矩形区域重画。这个函数的调用方法与函数 Invalidate 类似。参数 lpRect 中存储着视图中需要重画的区域，参数 bErase 用于指定当区域重画时，背景是否擦除，当 bErase 为 TRUE 时，背景被擦除，当 bErase 为 FALSE，背景保持不变。以下代码可以在视图类中完成重画矩形区域{100,100,200,200}的功能：

```
CRect r;
r.SetRect(100,100,200,200);
```

```
InvalidateRect(r,TRUE);
```

(2) InvalidateRgn 函数

函数的定义为:

```
void InvalidateRgn(CRgn * pRgn,BOOL bErase = TRUE);
```

InvalidateRgn 函数的功能和使用方法与 InvalidateRect 函数类似,唯一不同的是 InvalidateRgn 函数是使一个指定的区域重画。参数 pRgn 是指向一个 CRgn 对象的指针,创建 CRgn 对象的方法在 4.2.5 小节中已做了详细的介绍,InvalidateRgn 函数可以使一个已经初始化的 CRgn 对象所指定的区域重画。以下代码可使 CRgn 对象 rgn 指定的区域重画:

```
InvalidateRgn(&rgn);
```

7.3 图形放大和移动

本节讨论实现图形的放大和移动功能。图形放大指的是通过调整视图屏幕,使当前视图屏幕中的一部分区域放大显示到整个视图屏幕中;图形移动指的是通过调整视图屏幕来显示图形的其他部分。

7.3.1 图形放大和移动

本小节讨论在进行图形放大和移动操作时,屏幕参数的计算方法。

(1) 图形放大

矢量图形系统应具有无级放缩的功能。所谓"无级",指的是图形能以任何的比例进行显示。在应用程序 Draw 已经建立的坐标系基础上,可以很容易实现图形的无级放大功能。实现矢量图形放大的操作方法较多,如可以进行固定比例的图形放大,即运行放大功能,用鼠标按中图形中的一个点,系统会把图形放大一个固定的比例,鼠标按中点处于放大后的图形屏幕的中心;再如可以通过窗口放大,即用鼠标在图形屏幕上选中一个窗口区域,系统会调整放大比例和屏幕基点坐标,把选中的区域正好放大到整个图形屏幕区域中。本节将对用窗口放大图形的操作方法和具体实现步骤进行介绍,在此基础上,可以轻而易举的实现其他方法的图形放大操作功能。

如图 7-2 所示,视图屏幕范围为子窗口客户区域,现在要把屏幕上两个对角点所确定的矩形内的区域放大到整个视图屏幕内,假定矩形的左下点坐标为 mPointOrign,右上角点坐标为点 point,实现放大功能就是使 mPointOrign 点成为视图屏幕的左下点,即计算出 mPointOrign 的实际坐标作为(m_xStart,m_yStart),并且调整比例 blc,使矩形窗口内的区域正好能够显示到视图屏幕内。因为应用程序 Draw 中采用的是横向和纵向相同比例的放缩,所以要保证放大倍数变化较小的方向正好填满显示屏。

要实现放大,就需要计算出 mPointOrign 点的实际坐标,作为图形屏幕左下基点的坐标(m_

图 7-2　图形的放大操作

Xstart,m _ Ystart)：

```
VPtoDP(mPointOrign.x,mPointOrign.y,&m _ xStart,&m _ yStart);
```

同时,还要确定图形的放大倍数。横向放大倍数为：

```
bl1 = (float)m _ wScreen/(point.x - mPointOrign.x);
```

纵向放大倍数为：

```
bl2 = (float)m _ hScreen/(point.y - mPointOrign.y);
```

取绝对值较小的那一个值作为放大倍数：

```
if(bl2 < bl1)
    bl1 = bl2;          //取较小值作为放大倍数
blc = blc/bl1;          //得到显示比例
```

用调整后的 m _ Xstart,m _ Ystart,blc 重新绘制图形时,就实现了图形的放大功能。

(2) 图形移动

图形移动的实现方法与图形放大类似,不同的是对于图形屏幕只是参数 m _ Xstart 和 m _ Ystart 变化,比例尺 blc 不变。假定图形移动时,图形从基点 mPointOrign 到目标点 point 做相对移动,这时屏幕左下点的实际坐标为：

```
VPtoDP(mPointOrign.x,mPointOrign.y,&m _ xStart,&m _ yStart);
```

用变化后的 m _ xStart 和 m _ yStart 重新绘制屏幕时,就实现了图形移动。

7.3.2　图形放大和移动功能的实现

设定应用程序 Draw 中实现图形放大和图形移动功能的操作步骤如下：选中“图形放大”菜单项后,按下鼠标左键点中一点作为窗口放大矩形边框的一个顶点,此时移动鼠标时就会拖动一个矩形边框移动,如再按下鼠标左键,就会把两个对角点确定的矩形区域放大到整个视图屏

幕上完成放大操作功能，在确定矩形边框的一个顶点后，如果按下了鼠标右键，就会擦除矩形框放弃放大操作。而选中“图形移动”菜单项时，按下鼠标左键点中一点作为移动的基点，此时鼠标移动时就会拖动一条橡皮线跟着移动，如果再按下鼠标左键选中目标点，就会以这两个点为相对位置完成图形移动操作，而如果在选择基点后按下鼠标右键，就会擦除橡皮线放弃移动操作。下面介绍这两种功能的实现步骤。

(1) 修改图形操作菜单的消息处理函数

对“图形放大”和“图形移动”菜单项在 CDrawView 类中的消息处理函数 OnGraphZoom 和 OnGraphPan 进行修改，使之能够标识进行的操作，修改后的函数代码如下：

```
void CDrawView::OnGraphZoom()
{
    m_DrawCurrent = 11;          //标识进行图形放大操作
    PushNumb = 0;                //按下鼠标左键的次数为 0
}

void CDrawView::OnGraphPan()
{
    m_DrawCurrent = 12;          //标识进行图形移动操作
    PushNumb = 0;                //按下鼠标左键的次数为 0
}
```

(2) 修改鼠标消息处理函数

图形放大和移动操作涉及到第 6 章讨论过的三个鼠标消息处理函数，需要对这些消息处理函数进行修改。

① 在实现文件 drawview.cpp 中的消息处理函数 OnLButtonDown 中，加入如下代码：

```
void CDrawView::OnLButtonDown(UINT nFlags,CPoint point)
{
    …… //省略以上代码
    else if(m_DrawCurrent = = 11||m_DrawCurrent = = 12) //正在进行图形放大或图形移动操作
    {
        if(PushNumb = = 0)          //如果第一次按下鼠标左键
        {
            mPointOrign = point;    //原点等于按中点
            mPointOld = point;      //上一个点等于按中点
            PushNumb++;             //鼠标左键按下次数
            SetCapture();           //捕捉鼠标
        }
        else if(PushNumb = = 1)     //第二次按下鼠标左键
        {
```

```
            if(m_DrawCurrent = = 11) //如果是图形放大操作
            {
                //得到放大区域左下点和右上点像素坐标
                x1 = min(mPointOrign.x,point.x);
                y1 = max(mPointOrign.y,point.y);
                x2 = max(mPointOrign.x,point.x);
                y2 = min(mPointOrign.y,point.y);
                VPtoDP(x1,y1,&m_xStart,&m_yStart);        //得到视图屏幕原点的实际坐标
                float bl1 = (float)m_wScreen/(float)(x2 - x1); //得到横向放大比例
                float bl2 = (float)m_hScreen/(float)(y1 - y2); //得到纵向放大比例
                if(bl2 < bl1)
                    bl1 = bl2;          //取较小值作为放大比例
                blc = blc/bl1;          //得到放大后的图形显示比例
            }
            else                        //如果是图形移动操作
            {
                //改变屏幕原点(左下角点)的实际坐标
                m_xStart = m_xStart - blc * (point.x - mPointOrign.x);
                m_yStart = m_yStart + blc * (point.y - mPointOrign.y);
            }
            PushNumb = 0;               //左键按下数置 0,重新进行图形放大或修改操作
            ReleaseCapture();           //释放捕捉的鼠标
            Invalidate();               //使图形重画
        }
    }
    ..... //省略以下代码
}
```

在以上函数代码,在第一次按下鼠标左键时,记录下按中点 mPointOrign 并捕捉鼠标,第二次按下鼠标左键时,进行屏幕基点坐标和显示比例(在"图形放大"时)参数的计算,释放捕捉的鼠标,并激活图形重画。

② 在实现文件 drawview.cpp 中的鼠标移动消息处理函数 OnMouseMove 中,加入如下代码:

```
void CDrawView::OnMouseMove(UINT nFlags,CPoint point)
{
    ..... //省略以上代码
    else if((m_DrawCurrent = = 11||m_DrawCurrent = = 12)&&PushNumb = = 1)
    //如果是图形放大或移动操作,并且按下了第一点
    {
        if(point! = mPointOld)          //如果鼠标移动点与上一个移动点不同
        {
            if(m_DrawCurrent = = 11)  //如果是图形放大操作
            {
```

```
                ddd.SelectStockObject(NULL_BRUSH); //设置不填充状态
                //擦除原来的矩形
                ddd.Rectangle(mPointOrign.x,mPointOrign.y,mPointOld.x,mPointOld.y);
                //画当前的矩形
                ddd.Rectangle(mPointOrign.x,mPointOrign.y,point.x,point.y);
            }
            else                        //如果是图形移动操作
            {
                ddd.MoveTo(mPointOrign); ddd.LineTo(mPointOld);  //擦除原来的橡皮线
                ddd.MoveTo(mPointOrign); ddd.LineTo(point);      //画新的橡皮线
            }
            mPointOld = point;          //将本次鼠标移动点作为下次移动的上一点
        }
    }
    ..... //省略以下代码
}
```

以上这一些代码在运行"图形放大"或"图形移动"菜单项后,在按下了一次鼠标左键的情况下,在屏幕中拖动矩形边框或橡皮线。在以上代码中,用到了一个 CDC 类的成员函数 Rectangle,这个函数是以当前的画笔和画刷画一个矩形,参数是两个对角顶点的坐标。

③ 在实现文件 drawview.cpp 中的消息处理函数 OnRButtonDown 中,加入如下代码:

```
void CDrawView::OnRButtonDown(UINT nFlags,CPoint point)
{
    ..... //省略以上代码
    else if((m_DrawCurrent==11||m_DrawCurrent==12)&&PushNumb==1)
    //如果在进行图形放大或图形移动操作,并且按下了一次鼠标左键
    {
        ddd.SelectStockObject(NULL_BRUSH);   //设置不填充状态
        if(m_DrawCurrent==11)                //如果是进行图形放大操作,擦除矩形框
            ddd.Rectangle(mPointOrign.x,mPointOrign.y,point.x,point.y);
        else                        //如果是进行图形移动操作,擦除橡皮线
        {
            ddd.MoveTo(mPointOrign);
            ddd.LineTo(point);
        }
        ReleaseCapture();           //释放捕捉的鼠标
        PushNumb=0;                 //放弃操作,使左键按下数置 0
    }
}
```

通过以上步骤,就实现了图形的放大和移动功能。在本章的 7.7 节中,将讨论实现用滚动条滚动视图来进行图形移动的功能。

7.4 重画上屏和重画首屏

7.3节实现了图形的放大、移动功能，可以将图形放大到任何比例及移动到任何位置。经过放大、移动操作后，要有把图形屏幕再恢复到前一屏幕(重画上屏)或者直接使图形回到第一屏(重画首屏)的功能。要实现重画上屏和重画首屏功能，就需要记录下每一屏幕的状态，并时刻跟踪着当时的状态。在应用程序 Draw 中建立的画图机制下，实现重画上屏和重画首屏功能比较简单。本节将具体讨论这个功能的实现方法和具体代码。

7.4.1 记录屏幕状态

对图形屏幕进行分析，就会发现一个图形屏幕是由三个参数来确定的：屏幕左下角原点的实际横纵坐标 m_xStart、m_yStart 和显示比例 blc，只要记录下这三个参数，就可以记录下这一屏幕的状态。

为了实现记录屏幕状态的功能，在文档类 CDrawDoc 中定义一个结构数组来存储历史图形屏幕。在 CDrawDoc 类中定义以下变量：

```
public:
    int m_CurrentScreen;        //当前屏幕的序号
    int m_MaxScreen;            //最多能够存储的屏幕数
    ScreenStruct * m_Screen;    //记录屏幕参数的结构指针
```

结构 ScreenStruct 在头文件 drawdoc.h 中的 CDrawDoc 类前定义：

```
typedef struct                  //用来记录历史屏幕参数的结构
{
    float blc;                  //显示比例
    float sx;                   //屏幕左下角横坐标
    float sy;                   //屏幕左下角纵坐标
}ScreenStruct;
```

在实现文件 drawdoc.cpp 中的 CDrawDoc 类的构造函数中，对定义的变量进行初始化：

```
CDrawDoc::CDrawDoc()
{
    ..... //省略以上代码
    m_MaxScreen = 1000;                          //最多能够记录在1000个历史屏幕
    m_Screen = new ScreenStruct[m_MaxScreen]; //动态分配记录屏幕的结构数组
    //以下设置首屏的参数，并将当前屏幕设置为首屏
    m_CurrentScreen = 0;
    m_Screen[0].sx = 0;
    m_Screen[0].sy = 0;
    m_Screen[0].blc = 1;
}
```

m_Screen是一个结构数组，每一个数组项用来存储一个屏幕状态，结构中的变量sx存储的是每一个屏幕的左下角原点的实际横坐标m_xStart，sy存储的是原点的实际纵坐标m_yStart，blc存储的是屏幕显示比例blc。m_CurrentScreen记录的是当前屏幕的序号。在以上的构造函数里，结构数组m_Screen第一项m_Screen[0]存储的是首屏的参数。在进行放大，移动等操作时，将变化后的参数记录到结构数组中，并将m_CurrentScreen增加1，以记录下当时的操作屏的位置。这样就有一个问题，不管采用多大的数组（数组大时占用较多内存），都会在操作中有到最后一项的时候（如以上定义的数组就只能存储m_MaxScreen个屏幕状态），不进行处理，程序就会出现致命的错误。为了解决这个问题，使结构数组产生滚动，当存储的屏幕数大于m_MaxScreen时，数组弹出第二项（即弹出m_Screen[1]的内容，m_Screen[0]记录的是首屏的内容，不做改变，所以不管操作多少次，数组的第一项仍然存着首屏，可以回到第一屏去），并将当前屏幕参数记录在结构数组的最后一项，加入一个函数来实现此功能。

在文档类CDrawDoc中定义一个函数，来完成存储一个图形屏幕状态的操作功能：

```
public:
void AddScreen(float StartX,float StartY,float blc);  //增加一个图形屏幕
```

在实现文件drawdoc.cpp中，加入函数的实现代码：

```
void CDrawDoc::AddScreen(float StartX,float StartY,float blc)
{
    int i;
    if(m_CurrentScreen==m_MaxScreen)
    //如果数组中没有空间再记录屏幕信息
    {
        //将数组做堆式滚动，去掉第二项([0]中存的是首屏信息，操作时不改变)
        for(i=1;i<m_MaxScreen-1;i++)
            m_Screen[i]=m_Screen[i+1];
    }
    else     //如果不是在数组的最后一项，屏幕记录号增加1
        m_CurrentScreen++;
    //记录下当前屏幕的参数，到结构数组对应的项中
    m_Screen[m_CurrentScreen].sx=StartX;
    m_Screen[m_CurrentScreen].sy=StartY;
    m_Screen[m_CurrentScreen].blc=blc;
}
```

在以上操作中，将记录图形屏幕的结构数组m_Screen放在了文档类CDrawDOc中，这样就能够保证在多视图状态下，同一个文档对象的多个视图具有相同的图形屏幕和历史屏幕。如果将结构数组m_Screen放在视图类CDrawView中，在多视图状态下，就能够保证一个文档对象的多个视图具有不同的图形屏幕和历史屏幕。实现一个文档对象的多个视图具有不同历史屏幕的操作，以及多文档、多视图编程技术的详细剖析，可参考作者所著的《Visual C++高级编程技术——开发实例剖析》一书。

为了在完成“图形放大”或“图形移动”等操作功能后，能够记录下历史屏幕的状态，需要在

CDrawView 类的消息处理函数 OnLButtonDown 中，加入调用 AddScreen()函数的代码：

```
void CDrawView::OnLButtonDown(UINT nFlags,CPoint point)
{
    CDrawDoc * pDoc = (CDrawDoc * )GetDocument();

    ......// 省略以上代码
    else if(m_DrawCurrent = = 111|m_DrawCurrent = = 12)          //进行图形放大或移动
    {
        if(PushNumb = = 0)              //如果第一次按下鼠标左键
        {
            mPointOrign = point;        //原点等于按中点
            mPointOld = point;          //上一个点等于按中点
            PushNumb++;                 //按键数增加 1
            SetCapture();               //捕捉鼠标
        }
        else if(PushNumb = = 1)         //第二次按下左键
        {
            if(m_DrawCurrent = = 11) //图形放大
            {
                //得到左下点和右上点
                x1 = min(mPointOrign.x,point.x);
                y1 = max(mPointOrign.y,point.y);
                x2 = max(mPointOrign.x,point.x);
                y2 = min(mPointOrign.y,point.y);
                //改变原点的坐标
                m_xStart = m_xStart + blc * x1;
                m_yStart = m_yStart + blc * (m_hScreen - y1);
                //改变比例
                float bl1 = (float)m_wScreen/(float)(x2 - x1);  //横向放大比
                float bl2 = (float)m_hScreen/(float)(y1 - y2);  //纵向放大比
                if(bl2 < bl1)
                    bl1 = bl2;      //取较小的变化比例
                blc = blc/bl1;      //得到显示比例
            }
            else                    //图形移动操作
            {
                //改变原点(左下角点)的实际坐标
                m_xStart = m_xStart - blc * (point.x - mPointOrign.x);
                m_yStart = m_yStart + blc * (point.y - mPointOrign.y);
            }
            PushNumb = 0;           //完成操作，左键按下数置 0
            m_DrawCurrent = 0;      //完成一次后，不继续运行此功能
```

```
            pDoc->AddScreen(m_xStart,m_yStart,blc);   //保存当前屏幕的参数
            ReleaseCapture();          //释放捕捉的鼠标
            Invalidate();              //使图形重画
        }
    }
    ......// 省略以下代码
}
```

7.4.2　实现重画上屏和重画首屏功能

有了以上屏幕状态的存储机制,将非常容易地完成重画上屏和重画首屏操作功能。

(1) 重画上屏

要实现重画上屏操作功能,只需把 m_CurrentScreen 的值减去 1,从结构数组 m_Screen 中取出屏幕参数重画图形即可。在实现文件 drawview.cpp 中,将"重画上屏"菜单项的消息处理函数 OnGraphUp 的实现代码修改如下:

```
void CDrawView::OnGraphUp()
{
    CDrawDoc * pDoc = (CDrawDoc * )GetDocument();
    if(pDoc->m_CurrentScreen>0)      //如果当前不是首屏,当前图形屏幕序号减 1
        pDoc->m_CurrentScreen--;
    //得到上一屏幕的图形参数并重画图形
    m_xStart = p_Screen[pDoc->m_CurrentScreen].sx;
    m_yStart = p_Screen[pDoc->m_CurrentScreen].sy;
    blc = p_Screen[pDoc->m_CurrentScreen].blc;
    Invalidate();                    //重画图形
}
```

p_Screen 是 CDrawView 类中定义的指向文档对象中的结构数组 m_Screen 的指针,在视图类 CDrawView 中定义:

```
ScreenStruct * p_Screen;
```

在实现文件 drawview.cpp 中,在重载的虚函数 OnActiveView 中对 p_Screen 进行初始化,将它指向当前文档对象中的结构数组 m_Screen:

```
void CDrawView::OnActivateView(BOOL bActivate, CView * pActivateView, CView * pDeactiveView)
{
    CDrawDoc * pDoc = (CDrawDoc * )GetDocument();
    p_Screen = pDoc->m_Screen;
    CView::OnActivateView(bActivate, pActivateView, pDeactiveView);
}
```

(2) 重画首屏

对于重画首屏功能来说，只要取出存储在结构数组 m _ Screen 中的首屏参数，并对视图重画即可。修改实现文件 drawview.cpp 中的消息处理函数 OnGraphFirst，得到实现代码如下：

```
void CDrawView::OnGraphFirst()
{
    CDrawDoc * pDoc = GetDocument();          //得到指前当前文档对象的指针
    if(pDoc->m_CurrentScreen==0)              //如果已是首屏，函数退出
        return;
    //得到首屏的参数，并重画图形
    m_xStart=p_Screen[0].sx;
    m_yStart=p_Screen[0].sy;
    blc=p_Screen[0].blc;
    pDoc->AddScreen(m_xStart,m_yStart,blc);
    Invalidate();
}
```

通过以上操作，就实现了矢量图形系统的重画上屏和重画首屏功能。

7.5　显示全图

在矢量图形系统中，随着一些图形元素通过外部接口或者系统本身的绘制功能的加入，图形首屏可能包含不了所有的图形元素。还有一种情况是，所有的图形元素都在视图屏幕的一个角上，只占图幅的一小部分。这就需要一个操作功能，调整视图屏幕左下角原点的实际坐标(m _ xStart，m _ yStart)和显示比例 blc，使所有的图形元素正好能够绘制到视图屏幕中(在屏幕上留有一部分边界区域)，显示全图功能就是为完成此功能而设置的。实现显示全图功能的思路是，通过得到每一类图形元素(如直线、连续直线)的边界矩形(即能够包含所有直线的最小矩形)，比较运算得到所有图形元素的边界矩形，然后根据视图屏幕的大小(逻辑坐标)进行计算，得到屏幕的参数 m _ xStart，m _ yStart，blc。所以，解决问题的关键在于得到各类图形元素的边界矩形。

7.5.1　各类图形元素的边界矩形计算

本小节将讨论各类图形元素边界矩形的计算方法，并实现各类图形元素边界矩形的计算函数。为了通过 CDraw 类指针利用多态性调用完成各类图形元素的边界矩形计算，在图形元素基类 CDraw 中定义一个虚函数：

```
public:
    virtual void GetRect(float * minX,float * minY,float * maxX,float * maxY)=0;
```

GetRect 函数用来计算图形元素的边界矩形，指针参数(* minX，* minY)和(* maxX，* maxY)用来返回边界矩形的最小和最大坐标。

因为不用 CDraw 类直接定义对象，所以将函数 GetRect 定义成纯虚函数，这样当 CDraw 类的派生类需要能够定义对象时，必须在派生类中重载这个虚函数。在由 CDraw 类派生的各个图形元素类 CLine、CPline、CCircle、CArc、CText 中重载虚函数 GetRect，即在各个类中加入以下定义代码：

```
public:
    virtual void GetRect(float * minX,float * minY,float * maxX,float * maxY);
```

以下，在各个图形元素类 CLine、CPline、CCircle、CArc、CText 中，实现虚函数 GetRect 的实现代码，以完成各类图形元素边界矩形的实际计算。

(1) 直线

直线的边界矩形就是以直线的两个顶点为对角顶点的矩形，在实现文件 drawdoc.cpp 中，加入直线类 CLine 的重载虚函数 GetRect 的实现代码：

```
void CLine::GetRect(float * minX,float * minY,float * maxX,float * maxY)
{
    * minX = min(m_X1,m_X2);      //计算并返回最小 x 值
    * maxX = max(m_X1,m_X2);      //计算并返回最大 x 值
    * minY = min(m_Y1,m_Y2);      //计算并返回最小 y 值
    * maxY = max(m_Y1,m_Y2);      //计算并返回最大 y 值
}
```

(2) 连续直线或多边形区域

连续直线或多边形区域是多条直线的集合，计算边界矩形坐标的方法与直线类似。对于连续直线(或多边形区域)来说，就是要得到所有顶点的最小及最大坐标。在实现文件 drawdoc.cpp 中，加入连续直线类 CPline 的重载虚函数 GetRect 的实现代码：

```
void CPline::GetRect(float * minX,float * minY,float * maxX,float * maxY)
{
    float x1,y1,x2,y2;
    //用第一个顶点的坐标初始化变量
    x1 = m_PointList[0].x;
    x2 = m_PointList[0].y;
    y1 = m_PointList[0].x;
    y2 = m_PointList[0].y;
    for(int i = 1;i < m_Numble;i++ )          //从第二个顶点开始循环
    {
        x1 = min(x1,m_PointList[i].x);        //得到最小 x 坐标
        x2 = max(x2,m_PointList[i].x);        //得到最大 x 坐标
        y1 = min(y1,m_PointList[i].y);        //得到最小 y 坐标
        y2 = max(y2,m_PointList[i].y);        //得到最大 y 坐标
    }
```

```
    * minX = x1; * maxX = x2; * minY = y1; * maxY = y2; //将边界矩形顶点坐标值赋给地址返回
}
```

(3) 圆和圆形区域

圆的边界矩形就是一个中心在圆心,边长等于圆直径的矩形。在实现文件 drawdoc.cpp 中,加入圆及圆形区域类 CCircle 的重载虚函数 GetRect 的实现代码:

```
void CCircle::GetRect(float * minX,float * minY,float * maxX,float * maxY)
{
    * minX = m_CircleX - m_CircleR;          //得到并返回最小 x 值
    * minY = m_CircleY - m_CircleR;          //得到并返回最小 y 值
    * maxX = m_CircleX + m_CircleR;          //得到并返回最大 x 值
    * maxY = m_CircleY + m_CircleR;          //得到并返回最大 y 值
}
```

(4) 圆弧

圆弧边界矩形的计算方法稍微复杂一些,计算思路如下:圆弧的边界矩形是由几个特征点决定的,圆弧的起点、终点以及圆弧与坐标轴的交点。根据圆弧的起始弧度和结束弧度确定圆弧相交了几个坐标轴,并计算出与坐标轴的交点,然后求出圆弧的起终点及与坐标轴交点的最小值和最大值即计算得到了圆弧的边界矩形。

在实现文件 drawdoc.cpp 中,加入圆弧类 CArc 的重载虚函数 GetRect 的实现代码:

```
void CArc::GetRect(float * minX,float * minY,float * maxX,float * maxY)
{
    float x,y,x1,y1,x2,y2;
    int i,pb;
    // 得到圆弧起点坐标
    x = m_CircleX + (float)(cos(m_Angle1) * m_CircleR);
    y = m_CircleY + (float)(sin(m_Angle1) * m_CircleR);
    //用圆弧起点坐标来初始化边界矩形
    x1 = x; y1 = y;
    x2 = x; y2 = y;
    // 得到圆弧的终点坐标
    x = m_CircleX + (float)(cos(m_Angle2) * m_CircleR);
    y = m_CircleY + (float)(sin(m_Angle2) * m_CircleR);
    //得到最小和最大坐标
    x1 = min(x,x1);x2 = max(x,x2);
    y1 = min(y,y1);y2 = max(y,y2);
    for(i = 0;i < 4;i++)          //对四个坐标轴循环,检查坐标轴是否与圆弧相交
    {
        if(IaInArc(pi/2 * i))
```

```
        {
            if(i = =0)          //如果与 X 轴正方向相交,得到交点坐标
            {
                x = m_CircleX + m_CircleR;
                y = m_CircleY;
            }
            else if(i = =1)     //如果与 Y 轴正方向相交,得到交点坐标
            {
                x = m_CircleX;
                y = m_CircleY + m_CircleR;
            }
            else if(i = =2)     //如果与 x 轴负方向相交,得到交点坐标
            {
                x = m_CircleX - m_CircleR;
                y = m_CircleY;
            }
            else if(i = =3)     //如果与 Y 轴正方向相交,得到交点坐标
            {
                x = m_CircleX;
                y = m_CircleY - m_CircleR;
            }
            //得到最小和最大坐标值
            x1 = min(x,x1);x2 = max(x,x2);
            y1 = min(y,y1);y2 = max(y,y2);
        }
    }
    * minX = x1; * maxX = x2; * minY = y1; * maxY = y2; //赋值给指针参数返回边界矩形的顶点坐标
}
```

函数 GetRect 的主要实现步骤如下：

① 计算得到圆弧起点的实际坐标值，并把它赋给最小坐标点(x1,y1)和最大坐标点(x2,y2)。

② 计算得到圆弧终点的实际坐标值，并与最小坐标点和最大坐标点比较，得到最小和最大坐标值。

③ 依次判断圆弧是否与 X 轴正向、Y 轴正向，X 轴负向，Y 轴负向相交，如相交得到交点，并比较计算出最小和最大坐标值。

④ 返回得到的最小和最大坐标值，即矩形边界的左下和右上顶点坐标。

在 GetRect 函数中，调用了一个函数 IsArc 来判断各个坐标轴是否与圆弧相交，IsInArc 函数在圆弧类 CArc 中定义：

```
BOOL IsInArc(float angle);
```

IsInArc 函数用来检查弧度 angle 是否在圆弧范围内，如果在圆弧范围内函数返回 TRUE，否则返回 FALSE。在实现文件 drawdoc.cpp 中加入函数的实现代码：

```
BOOL CArc::IsInArc(float angle)
{
    if((m_Angle1 < m_Angle2&&angle < m_Angle2&&angle > m_Angle1)
        ||m_Angle1 > m_Angle2&&(!(angle <= m_Angle1&&angle >= m_Angle2)))
        return TRUE;
    else
        return FALSE;
}
```

(5) 标注文本

标注文本边界矩形的计算方法与连续直线类似，每一行标注文本是在一个矩形内，文本旋转时根据旋转角度得到矩形四个顶点的实际坐标，通过比较运算即可得到这个标注文本的边界矩形。在实现文件 drawdoc.cpp 中，加入标注文本类 CText 的重载虚函数 GetRect 的实现代码：

```
void CText::GetRect(float *minX,float *minY,float *maxX,float *maxY)
{
    float x[4],y[4];
    double ang=pi*m_Angle1/180;
    //以下是文本标注起点的横纵坐标
    x[0]=m_StartX; y[0]=m_StartY;
    //得到标注文本框的右下角坐标
    x[1]=m_StartX+(float)((m_TextWide+m_OffWide)*m_TextLong*cos(ang));
    y[1]=m_StartY+(float)((m_TextWide+m_OffWide)*m_TextLong*sin(ang));
    //得到标注文本框的右上角坐标
    x[2]=x[1]-(float)(sin(ang)*m_TextHeight);
    y[2]=y[1]+(float)(cos(ang)*m_TextHeight);
    //得到标注文本框的左上角坐标
    x[3]=m_StartX-(float)(sin(ang)*m_TextHeight);
    y[3]=m_StartY+(float)(cos(ang)*m_TextHeight);
    *minX=m_StartX;
    *minY=m_StartY;
    *maxX=m_StartX;
    *maxY=m_StartY;
    for(int i=1;i<4;i++)          //比较得到最小及最大坐标值
    {
        if(*minX>x[i])*minX=x[i];
        if(*minY>y[i])*minY=y[i];
        if(*maxX<x[i])*maxX=x[i];
        if(*maxY<y[i])*maxY=y[i];
    }
}
```

对于其他类别的图形元素，可根据这些图形元素的几何特征，参考以上介绍的计算直线、连续直线、圆、圆弧、标注文本等类边界矩形的方法，实现计算其边界矩形的函数。

7.5.2 实现显示全图功能

上小节讨论实现了各类图形元素边界矩形的计算函数。根据这些函数计算出了这些图形元素的边界矩形，也就是这幅图形的实际范围，然后调整屏幕左下角的实际坐标(m_xSstart,m_yStart)和图形的显示比例 blc，使所有的图形元素(也就是这个矩形)能够显示到视图屏幕的范围内(这个算法在 7.3.1 小节中已经做了讨论)。

(1) 得到所有图形元素的边界矩形

把所有图形元素的边界矩形坐标进行比较，就可以得到所有图形元素的边界矩形。为了实现这个功能，在文档类 CDrawDoc 中定义一个函数：

```
public:
    BOOL GetRect(float *m_Xmin,float *m_Ymin,float *m_Xmax,float *m_Ymax);
```

指针参数(*m_Xmin,*m_Ymin)和(*m_Xmax,*m_Ymax)用来返回边界矩形的最小和最大坐标，当能够计算得到图形的边界矩形时，函数返回 TRUE，否则，函数返回 FALSE。在实现文件 drawdoc.cpp 中，加入这个函数的实现代码：

```
BOOL CDrawDoc::GetRect(float * m_Xmin, float * m_Ymin, float * m_Xmax, float * m_Ymax)
{
    float m_minX,m_minY,m_maxX,m_maxY;
    BOOL pb;
    //把记录最小值的变量赋大值,把记录最大值的变量赋小值,为了对变量初始化
    *m_Xmin = 1E+20;
    *m_Ymin = 1E+20;
    *m_Xmax = 1E-20;
    *m_Ymax = 1E-20;
    pb = 0
    for(int i = 1;i <= 5;i++)//依次对直线、连续直线、圆、圆弧、标注文本等几类图形元素循环
    {
        int nn = GetGraphUpperbound(i) + 1;//得到存储第 i 类图形元素 CObArray 对象数组最大下标
        while(nn--)                    //对某类所有图形元素遍历
        {
            if (GetGraph(i,nn))        //如果能够得到指向图形元素的指针
                GetGraph(i,nn)->GetRect(&m_minX,&m_minY,&m_maxX,&m_maxY);
            else
                continue;
            pb = 1;                    //表示能够计算出边界矩形
            //比较计算得到最小和最大坐标值
            if(*m_Xmin > m_minX)
```

```
                * m_Xmin = m_minX;
            if( * m_Xmax < m_maxX)
                * m_Xmax = m_maxX;
            if( * m_Ymin > m_minY)
                * m_Ymin = m_minY;
            if( * m_Ymax < m_maxY)
                * m_Ymax = m_maxY;
        }
    }
    return pb;
}
```

在 GetRect 函数中，首先对记录最小值和最大值的变量初始化，然后对所有的直线、连续直线或多边形区域、圆、圆弧、矢量文本等类图形元素进行循环，得到每个图形元素的边界矩形，并比较运算得到所有图形元素的边界矩形。当至少有一个图形元素时函数返回 TURE，没有任何图形元素时函数返回 FALSE，通过函数的返回值来确定是否计算出了图形元素的边界矩形。如果计算出了边界矩形，其边界矩形坐标通过浮点指针 * m_Xmin，* m_Ymin，* m_Xmax，* m_Ymax 返回。

(2) 实现显示全屏功能

在得到所有图形元素的边界矩形后，就可以调整视图屏幕的基点和比例，使当前视图正好能够绘制下所有的图形元素。在实现文件 drawview.cpp 中，对菜单项“重画全屏”的消息处理函数 OnGraphAll()进行修改，修改后的代码如下：

```
void CDrawView::OnGraphAll()
{
    float minx,miny,maxx,maxy,bl1;
    CDrawDoc * pDoc = GetDocument();
    //捕捉光标，并使光标成沙漏等待光标
    SetCapture();                                    //捕捉光标
    SetCursor(LoadCursor(NULL,IDC_WAIT));            //设置沙漏等待光标
    BOOL pb = pDoc->GetRect(&minx,&miny,&maxx,&maxy);    //计算得到图形元素的边界矩形
    SetCursor(LoadCursor(NULL,IDC_ARROW));           //设置原来的箭头形光标
    ReleaseCapture();                                //释放捕捉的光标
    if(! pb)                                         //如果没有计算出边界矩形，函数返回
        return;
    bl1 = (maxx - minx)/(m_wScreen - 20);            //横向比例尺
    blc = (maxy - miny)/(m_hScreen - 20);            //纵向比例尺
    if(bl1 > blc)                                    //取较大的一个值作为比例尺的取值
        blc = bl1;
    //以下确定屏幕左下角的实际坐标，在左侧和下侧都留了 10 逻辑单位的区域
    m_xStart = minx - 10 * blc;
    m_yStart = miny - 10 * blc;
```

```
        //将首屏的参数设置为当前计算得到的参数,把视图屏幕设为首屏,并重画视图
        p_Screen[0].sx = m_xStart;
        p_Screen[0].sy = m_yStart;
        p_Screen[0].blc = blc;
        pDoc->m_CurrentScreen = 0;
        Invalidate();
    }
```

在 OnGraphAll 函数中,调用当前文档类的 GetRect 函数得到图形中所有图形元素的边界矩形。如果存在边界矩形,则调整屏幕左下角点的坐标和显示比例,使所有的图形元素能够正好显示在视图屏幕范围内。

在 OnGraphAll 函数中,还用到了设置光标的函数:SetCursor(LoadCursor (NULL, IDC_WAIT)),SetCursor 函数用来设置光标类型,显示全屏可能要花费很长时间,所以捕捉了光标并把光标设置为沙漏型(IDC_WAIT),使用户知道正在进行边界矩形的计算,计算结束以后,又恢复了箭头型(IDC_ARROW)光标。

7.6 提高矢量图形系统重画速度的基本方法

本节将对提高矢量图形系统重画速度的基本方法进行介绍。

7.6.1 提高图形重画速度的方法

虽然计算机硬件在加速发展,运行速度越来越快,但是对于一个复杂的图形来说,因为它含有大量的图形元素,图形显示一次需要花费大量的时间。尤其是绘图函数,不管多么简单的一个绘图函数(如画一条直线的 LineTo 函数),都要经过大量的运算,计算出它应该显示的像素位置并实现对每个像素的显示,需要较长的运行时间。在开发矢量图形软件时,要尽可能采取好的策略,提高软件的运行速度。运行速度的快慢是决定一个图形软件实用性的决定性因素之一。所以,提高图形的显示速度及选中图形(8.1 节介绍)的速度是非常必要的。

怎样提高图形的显示速度呢?首先,对图形系统的工作步骤进行分析,分析后会发现一个比较大的弱点,即矢量图形系统具有无级放缩的功能。一般情况下并不是所有的图形元素都能够显示到屏幕上(除非在显示全图状态),而上述的作图机制则是把所有图形元素,不管是屏幕内的还是屏幕外的都进行绘制,严重影响了绘图速度(放大的比例越大,影响越严重)。也就是说,系统的绘图函数不会自己判断所画的图形元素是不是在视图屏幕上,一律对图形元素进行绘制。为了解决这个问题,一个基本的方法就是在显示一个图形元素前,对这个图形元素进行判断,判断这个图形元素是否在视图屏幕中,如果这个图形元素不在当前的视图屏幕中,就不用对这个图形元素进行绘制,以此节省绘制时间。

怎样判断一个图形元素是否在视图屏幕上呢?严格的说,要通过计算判断图形元素与当前视图屏幕矩形是否相交才能确定,如果按此方法判断,必须经过大量的计算,更影响显示速度。所以,采用的判断图形元素是否需要绘制的方法应允许不太严密,但是要计算简单。所谓不严密,指的是用这种方法判断是在视图屏幕中的图形元素,实际也可能不在视图屏幕中,而经过判断不在视图屏幕中的图形元素一定不在视图屏幕中。一般采用这个图形元素的边界矩

形和视图屏幕边界矩形是否相交的方法。7.5节讨论显示全图功能时，讨论了各种图形元素边界矩形的计算方法并给出了计算函数，利用这些函数并对图形元素的绘制函数略做修改，就可以达到快速绘制的目的。

7.6.2　提高图形重画速度的实现方法

首先要得到视图屏幕，即(客户区)的大小，在应用程序 Draw 中，在视图类 CDrawView 中有两个变量 m_hScreen 和 m_wScreen 分别用来存储当前视图的高度和宽度(逻辑坐标)。在进行图形绘制前，把视图屏幕矩形要转换成实际坐标。

定义以下四个全局浮点变量，用来传递要绘制视图的左下角和右上角的实际坐标。在实现文件 drawdoc.cpp 中，增加全局变量的定义：

```
float xMinScreen,yMinScreen,xMaxScreen,yMaxScreen;
```

为了在实现文件 drawview.cpp 中使用这些变量，在实现文件 drawview.cpp 中增加对这些全局变量的引用：

```
extern float xMinScreen,yMinScreen,xMaxScreen,yMaxScreen;
```

这四个全局变量在每次调用 OnDraw 函数实现视图的绘制前被初始化，在实现文件 drawview.cpp 中的绘制函数 CDrawView::OnDraw(CDC * pDC)中加入代码，实现在每次绘制前得到视图屏幕边界的实际坐标：

```
void CDrawView::OnDraw(CDC * pDC)
{
    CRect rr;
    CDrawDoc * pDoc = GetDocument();          //得到指向当前文档对象的指针
    ASSERT_VALID(pDoc);
    xMinScreen = m_xStart;
    yMinScreen = m_yStart;
    xMaxScreen = xMinScreen + blc * m_wScreen;
    yMaxScreen = yMinScreen + blc * m_hScreen;
    DrawBack(pDC);                            //以底色填充整个屏幕
    pDoc->Draw(pDC,0,0,m_bColor);             //绘制所有图形元素
}
```

这样在以当前的屏幕参数(m_xStart,m_yStart,blc)绘制图形前，把视图屏幕的左下角和右上角的实际坐标计算出来。

为判断一个矩形区域与视图屏幕矩形是否相交。在 drawdoc.cpp 中定义一个全局函数：

```
BOOL IsRectCross(float minx,float miny,float maxx,float maxy);
```

函数的四个参数是矩形区域的左下角(minx,miny)和右上角(maxx, maxy)的实际坐标。检查与其相交的区域在点(xMinScreen,yMinScreen)和点(xMaxScreen,yMaxScreen)所确定的矩形内。在实现文件 drawdoc.pp 中加入函数的实现代码：

```
BOOL IsRectCross(float minx,float miny,float maxx,float maxy)
```

```
{
    if(minx > xMaxScreen || maxx < xMinScreen || miny > yMaxScreen || maxy < yMinScreen)
        return FALSE;    //如两个矩形不相交,函数返回 FALSE
    else
        return TRUE;     //如果两个矩形相交,函数返回 TURE
}
```

IsRectCross 函数在一个矩形区域与视图屏幕区域相交时返回 TURE,否则返回 FALSE。

7.6.3　加快图形元素的绘制速度

对各类图形元素的绘制函数 Draw 进行修改,使其能够先判断图形元素的边界矩形是否与图形屏幕区域相交,来决定是否需要对图形元素进行绘制。例如,可以在直线类 CLine 的绘制函数 Draw 中加入黑体部分的代码:

```
void CLine::Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor)
{
    int x1,y1,x2,y2;
    if(b_Delete)  //如果已经处于删除状态
        return;
    float minx,miny,maxx,maxy;
    GetRect(&minx,&miny,&maxx,&maxy);              //得到边界矩形
    if(! IsRectCross(minx,miny,maxx,maxy))         //如果边界矩形不与屏幕窗口相交
        return;
    short LineType = m_LineType;
    short ColorPen = m_ColorPen;
    if(m_DrawMode1 == 1)     //如果图形元素需要特殊显示
    {
        if(m_LineType! =2)  //如果图形元素采用的不是第二种线型(虚线),则采用第二种线型
            LineType = 2;
        else                //如果图形元素采用的是第二种线型(虚线),则采用第三种线型
            LineType++;
    }
    else if(m_DrawMode1 == 2) //如果需要用指定颜色绘制直线,将笔色设置为屏幕底色 BackColor
        ColorPen = BackColor;
    //创建并初始化一个 CPen 对象
    CPen pen((int)LineType,(int)m_LineWide,p_GraphPara->GetColor(ColorPen));
    CPen * pOldPen = pDC->SelectObject(&pen);      //在绘图对象中选入 CPen 对象
    if(m_DrawMode == 0)                            //如果是采用第 0 种绘制模式
        pDC->SetROP2(R2_COPYPEN);                  //设定覆盖的绘制模式
    else if( m_DrawMode == 1)                      //如果是采用第一种绘制模式
        pDC->SetROP2(R2_NOT);                      //设置反写的绘制模式
    DPtoVP(m_X1,m_Y1,&x1,&y1);                     //将直线起点坐标转换成逻辑坐标
    DPtoVP(m_X2,m_Y2,&x2,&y2);                     //将直线终点坐标转换成逻辑坐标
```

```
    pDC - > MoveTo(x1,y1);
    pDC - > LineTo(x2,y2);                    //绘制直线
    pDC - > SelectObject(pOldPen);            //恢复原来的画笔
}
```

对于 CPline、CCircle、CArc、CText 等图形元素类，同样也需要在其绘制函数 Draw 中加入相同的代码，在此不再一一列出。

以上讨论了通过判断图形元素的边界矩形是否与视图屏幕区域相交，来确定是否绘制图形元素的方法。这是加快图形重画速度的基本方法。

在目前计算机的硬件和软件技术都有很大提高的情况下，进行视图绘制时，采用的边界矩形一般比当前视图屏幕大的多（即将图形绘制在一个比客户区域大的多的窗口内），这样就可以通过调整视口的原点实现对图形的迅速滚动浏览。这种操作需要较大的内存（设定的图形屏幕越大，需要的内存越多），并且在第一次形成图形时所需要的时间较长。为了解决需要时间长的问题，一般采用多线程技术，将图形的绘制函数放在辅助线程中实现，而不影响主线程的操作。在作者编著的《Visual C++ 高级编程技术——开发实例剖析》一书中，详细介绍了矢量图形系统后台绘画的多线程技术，请读者参阅。

一个设计完善的图形系统，应该能够根据计算机的硬件配置情况，调整图形元素的绘制方法和参数（如图形屏幕的大小等），实现能够较大限度地使用硬件资源，改善运行性能。做到在较低的硬件环境下能够运行，而在较高的硬件环境下又能够最大限度地提高系统的性能。

7.7　实现滚动视图

在 7.3 节介绍了利用鼠标在屏幕上选中基点和目标点来移动图形的方法。在 DOS 程序设计中，图形移动一般都是采用这种方法。Windows 下提供了大量的控件来优化程序的操作界面，滚动条就是其中的一种。本节将讨论在无级放缩的矢量图形系统中，利用固有的滚动条滚动图形的方法。

7.7.1　滚动视图的实现方法

在 MFC 类中，有一个由类 CView 派生的类 CScrollView，利用 CScrollView 类作为应用程序视图类的基类实现的视图，通过简单的设定一些参数，就会自动实现滚动视图。

为了使视图自动具有滚动的功能，只要在建立视图类的时候，使视图类（如应用程序 Draw 的 CDrawView）由 MFC 的滚动视图类 CScrollView 派生而不是由 CView 派生即可。在已经存在的应用程序 Draw 中，将视图改造成滚动视图的方法比较简单，只要在视图类的头文件 drawview.h 和实现文件 drawview.cpp 中，把所有的类名 CView 用类名 CScrollView 代替即可。

7.7.2　滚动区域参数设置

用 7.7.1 小节方法形成的视图，做一些必要的参数设置，就可以实现滚动功能。

（1）设置滚动区域

滚动区域是一个窗口区域，即是滚动视图能够管理的窗口大小。只有滚动区域比当前视

图屏幕大的情况下,才可能进行滚动。

在视图类中有一个重载的虚函数 OnInitialUpdate,它在视图对象初始化或改变时被调用。在此函数中加入初始化视图滚动区域的代码:

```
void CDrawView::OnInitialUpdate()
{
    SIZE Size = {1000,1000};
    SetScrollSizes(MM_TEXT,Size);
    CView::OnInitialUpdate();
}
```

以上操作将滚动区域设置成在 MM_TEXT 映像模式下,长和宽为 1000 像素的区域。把经过改动后的应用程序 Draw 进行编译后运行,你就会惊喜的发现视图窗口上出现了滚动条,按中滚动条可以实现对视图进行滚动。

(2) 设置滚动步长

对于以上实现的滚动视图,当用滚动条进行滚动操作时,在用鼠标按中滚动按钮时,视图滚动一定距离,按中滚动条中的按钮和滚动钮之间的区域时,视图会滚动更大的距离,这两个滚动距离可以分别给定一个名称:行滚动步长和页滚动步长。这两个步长可以通过 SetScrollSizes 函数来设定。

在 SetScrollSizes 函数中,除了以上讨论的两个参数(映像方式和滚动区域)外,还有可选的参数用来设置滚动页的大小和滚动步长。以下是 CScrollView 类中 SetScrollSizes 函数的完整定义:

```
void SetScrollSizes(int nMapMode, SIZE sizeTotal,
    const SIZE& sizePage = sizeDefault,
    const SIZE& sizeLine = sizeDefault);
```

SetScrollSizes 函数的第一个参数是坐标映像模式;第二个参数 sizeTotl 为滚动区域的大小;第三个参数为页滚动步长的大小,页滚动步长包含两个方向:横向和纵向,通过一个 SIZE 结构 sizePage 来表示;第四个参数 sizeLine 用来设置行滚动步长的大小,行滚动步长同样包含两个方向,也是通过一个 SIZE 结构 sizeLine 来表示。以下是一个操作实例:

```
void CDrawView::OnInitialUpdate()
{
    SIZE Size = {3000,2000};
    SIZE pSize = {500,400};
    SIZE lSize = {50,40};
    SetScrollSizes(MM_TEXT,Size,&pSize,&lSize);
    CView::OnInitialUpdate();
}
```

以上代码将滚动区域设置成{3000,2000},横向滚动一页的步长是 500(像素),滚动一行的步长是 50 像素,纵向滚动一页的步长是 400(像素),滚动一行的步长是 40 像素。

滚动步长一般采用系统的默认值。此值能够随着视图窗口的变化而调整。

7.8 定制图形滚动功能

用7.7节的方法可以迅速的实现一个滚动视图,这对实现有限范围的图形滚动是非常方便的。

而在应用程序 Draw 中,直接应用滚动视图存在着两个问题:一方面应用程序 Draw 是一个无级放缩的矢量图形系统,图形的滚动区域不好确定(更不好用整数确定),在滚动区域大的情况下,系统为了存储滚动区域需要开销很大的内存;另一方面,在7.6节加快图形的显示速度中曾经介绍,为了加快图形的显示速度,只对在当前视图屏幕范围内的图形元素进行了绘制,视图外的图形元素并没有绘制,通过调整视图原点实现滚动时,新出现的区域中的图形元素可能没有绘制,因而无法显示。当然可以对应用程序 Draw 的绘制区域进行调整(比视图大但不大于一定的区域),在一定范围内实现滚动视图。

本节将介绍在应用程序 Draw 中,实现一个能自己控制的滚动视图(而不是靠采用 CScrollView 类来实现),使读者掌握滚动条程序设计的原理和操作方法,达到随心所欲地改动和组织视图的滚动。

7.8.1 确定滚动步长

首先讨论确定滚动步长的方法。一个无级放缩的图形,假定首屏正好能够充满视图屏幕,在放大100倍后,目前的视图屏幕在实际图形上只占1/100的宽度,视图屏幕由图形的最上端到最下端要滚动过99个整屏,在不同的比例尺下要滚动的屏数是不一样的。怎么确定滚动的最小步长呢?所谓滚动步长就是做一次滚动的最小距离,这个距离,可以用每次滚动的屏幕像素的长度来确定。例如,设定最小的滚动步长为60个像素(对于一个600像素宽的屏幕来说,不论在什么比例下,需要按10次能够滚动完一个屏幕),用这个滚动步长时,滚动范围可以达到3276个屏幕(在16位 C++ 下滚动条的滚动范围也是整数型,32767/10 = 3276),也就是说图形系统能够最大实现3276倍的放大,这对一个一般的矢量图形系统来说是足够的。加大滚动步长的像素数,放大范围还可以扩大。

同用鼠标实现图形滚动时一样,在应用程序 Draw 中,图形滚动是靠调整视图屏幕的实际起点 m_xStart,m_yStart 来实现的。在一定的显示比例 blc 下,最小的像素滚动步长可以转换成实际坐标长:60 * blc,当按下滚动条时,通过调整视图屏幕基点的实际坐标(m_xStart,m_yStart),就可以达到滚动图形的目的。以下讨论这种图形滚动功能在应用程序 Draw 中的具体实现方法。

7.8.2　创建滚动条

可以在应用程序 Draw 中增加一个滚动条，在窗口类中有一个函数 PreCreateWindow，它在产生一个窗口以前被调用，我们可以用它来修改窗口的风格。修改视图类 CDrawView 的 PreCreateWindow 函数，在此函数中加入黑体部分代码：

```
BOOL CDrawView::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style = cs.style|WS_HSCROLL|WS_VSCROLL;
    return CMDIChildWnd::PreCreateWindow(cs);
}
```

当改变了视图窗口的风格后，编译运行应用程序 Draw 时，视图窗口中可能没有出现滚动条。这没关系，只要用函数对滚动条的参数进行设置后，滚动条就会自动出现。以下介绍控制和使用滚动条实现无级放缩图形系统下的滚动视图的方法。

7.8.3　实现图形滚动功能

本小节来具体讨论图形滚动功能的实现。

(1) 设置滚动步长

为实现在应用程序 Draw 下的实现滚动视图。在视图类 CDrawView 中加入以下几个变量：

```
protected:
    int nHScrollMax,nHScrollPos,nVScrollMax,nVScrollPos;
    int nXPage,nXLine,nYPage,nYLine,nScrollMin;
```

参数 nHScrollMax 用来存储横向滚动条的最大滚动范围，参数 nHScrollPos 用来存储横向滚动条的当前位置，参数 nVScrollMax 用来存储纵向滚动条的最大滚动范围，参数 nVScrollPos 用来存储纵向滚动条的当前位置。参数 nXPage 用来存储按中横向滚动条的中间区域时滚动一个页面时的范围数(滚动几个最小滚动距离)，参数 nXLine 用来存储按中横向滚动条的按钮时滚动的范围数(滚动几个最小滚动距离，一般为 1)，参数 nYLine 和参数 nYPage 分别用来存储纵向滚动条相应的滚动范围数，参数 nScrollMin 用来存储一个最小滚动距离(像素表示)。

在实现文件 drawview.cpp 中，在视图类 CDrawView 的构造函数中加入这些变量的初始化代码：

```
CDrawView::CDrawView()
{
    ..... //省略
    nScrollMin = 50;
    nXLine = 1;
    nYLine = 1;
}
```

以上代码,设定滚动条最小滚动距离是50个像素,当按下横向和纵向滚动条的按键时,滚动一个最小滚动距离。

随着视图屏幕大小的变化,滚动一个页面所需要的滚动次数也随着变化,所以在消息处理函数 OnSize 中,对横向和纵向滚动一个页面所需要的基本滚动次数进行初始化:

```
void CDrawView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    CDrawDoc * pDoc = (CDrawDoc * )GetDocument();      //得到指向当前文档对象的指针
    p_Screen = pDoc->m_Screen;
    //记录改变后的屏幕大小
    m_wScreen = cx;
    m_hScreen = cy;
    nXPage = m_hScreen/nScrollMin;      //横向滚动一屏需要的基本滚动数
    nYPage = m_wScreen/nScrollMin;      //纵向滚动一屏需要的基本滚动数
}
```

以上代码计算出横向和纵向滚动一个视图屏幕时的最小滚动次数。

(2) 实现工具条设置函数

为了对滚动条进行初始化,所谓初始化就是设定滚动条的最大滚动范围和当前所在位置,在视图类 CDrawView 中定义两个函数:

```
private:
void InitVScroll();   //设定纵向滚动条
void InitHScroll();   //设定横向滚动条
```

在实现文件 drawview.cpp 中增加函数的实现代码:

```
void CDrawView::InitVScroll()
{
    //得到纵向滚动条的最大滚动范围
    nVScrollMax = (int)(((float)p_hScreen * m_Screen[0].blc)/(blc * nScrollMin));
    //得到纵向滚动条的应设置的滚动位置
    nVScrollPos = nVScrollMax - (int)((m_yStart - p_Screen[0].sy)/(blc * nScrollMin));
    SetScrollRange(SB_VERT,0,nVScrollMax,TRUE);      //设置滚动条的最大滚动范围
    SetScrollPos(SB_VERT,nVScrollPos);               //设置纵向滚动条的当前位置
    UpdateWindow();
}
```

InitVScroll 函数是用来设定纵向滚动条的范围和当前位置。纵向滚动条的范围一般是这样确定的:把没有进行放大的首屏高度作为图形的纵向图幅范围:m_hScreen * p_Screen[0].blc,p_Screen[0].blc 是首屏的比例尺。而最小滚动距离为:blc * ScrollMin,blc 是当前图形的比例尺,ScrollMin 是最小滚动距离的像素数。纵向图幅高度与最小滚动距离相除就得到滚动条的最大滚动范围,根据现在视图屏幕基点的实际纵坐标与图幅的纵坐标相比就得到了滚动

条的当前滚动位置。SetScrollRange 和 SetScrollPos 函数分别用来设置滚动条的最大范围和当前位置，参数 SB _ VERT 代表是纵向滚动条。这样，就把滚动条设置为滚动范围为 0 ~ nVSrollMax，当前位置是 nVScrollPos。调用 UpdateWindow 在屏幕上改动设定过的滚动条。

对于横向滚动条来说，对滚动条初始化的函数与纵向滚动条类似。在实现文件 drawview .cpp中加入函数 InitHScroll 的实现代码：

```
void CDrawView::InitHScroll()
{
    //得到横向滚动条的最大滚动范围
    nHScrollMax = (int)(((float)m_hScreen * m_Screen[0].blc/blc)/nScrollMin);
    //得到横向滚动条的当前滚动位置
    nHScrollPos = (int)((m_xStart - m_Screen[0].sx)/blc/nScrollMin);
    SetScrollRange(SB_HORZ,0,nHScrollMax,TRUE);      //设置滚动条的最大滚动范围
    SetScrollPos(SB_HORZ,nHScrollPos);               //设置横向滚动条的当前位置
    UpdateWindow();
}
```

建立这两个函数的目的是对滚动条进行初始化，所以需要在 OnSize 函数中，以及图形放大、图形移动(用鼠标)、重画上屏、重画首屏、重画全屏等操作函数运行后，都要加入以下两行代码：

```
InitHScroll();
InitVScroll();
```

(3) 实现工具条消息处理函数

以上完成了对横向和纵向滚动条的初始化。现在进行下一步，也就是最关键的一步：获得鼠标按中滚动条的消息。在窗口类中有两个窗口消息 WM _ VSCROLL 和 WM _ HSCROLL，分别是当用鼠标操作纵向和横向滚动条时产生的，可以用 ClassWizard(或手工)在视图类 CDrawView 中加入这两个消息的处理函数。操作完成后，应用程序 Draw 代码修改如下：

在视图类 CDrawView 中，增加了消息处理函数的定义：

```
//{{AFX_MSG(CDrawView)
    ......//省略以上代码
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
    afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
    //}}AFX_MSG
```

在实现文件 drawview.cpp 中，加入消息处理函数的消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    //{{AFX_MSG_MAP(CDrawView)
    ......//省略以上代码
    ON_WM_VSCROLL()
    ON_WM_HSCROLL()
```

```
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

在实现文件 drawview.cpp 中，加入这两个消息处理函数的实现代码：

```
void CDrawView::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)
{
    short nScrollInc;
    int nNewPos;
    switch(nSBCode)                          //判断按键位置
    {
        caseSB_TOP:                          //如果将滚动条滚动到顶部
            nScrollInc = -nVScrollPos;       //得到相对移动的最小滚动距离数
            break;
        caseSB_BOTTOM:                       //如果将滚动条滚动到底部
            nScrollInc = nVScrollMax - nVScrollPos;  //得到移动最小滚动距离数
            break;
        caseSB_LINEUP:                       //如果按了滚动条中的向上按键
            nScrollInc = -nYLine;            //得到相对移动的最小滚动距离数
            break;
        caseSB_LINEDOWN:                     //如果按中了滚动条向下的按键
            nScrollInc = nYLine;             //得到相对移动的最小滚动距离数
            break;
        caseSB_PAGEUP:                       //如果按中了中间活动按钮的上部区域
            nScrollInc = -nYPage;            //得到相对移动的最小滚动距离数
            break;
        caseSB_PAGEDOWN:                     //如果按中了中间活动按钮的下部区域
            nScrollInc = nYPage;             //得到相对移动的最小滚动距离数
            break;
        caseSB_THUMBPOSITION:                //如果用鼠标拖动中间活动钮到一个位置
            nScrollInc = nPos - nVScrollPos; //得到相对移动的最小滚动距离数
            break;
        default:
            nScrollInc = 0;
    }
    //得到滚动条实际滚动位置(不能超出滚动条的纵向滚动范围)
    nNewPos = max(0, min(nVScrollPos + nScrollInc, nVScrollMax));
    //得到实际的相对滚动范围(最小滚动距离数目)
    nScrollInc = nNewPos - nVScrollPos;
    if(nScrollInc)        //如果产生了滚动
```

```
    {
        nVScrollPos = nNewPos;                          //得到新的滚动位置
        SetScrollPos(SB_VERT,nVScrollPos);              //设置纵向滚动条的新位置
        UpdateWindow();                                 //更新滚动条显示
        m_yStart = m_yStart - blc * nScrollInc * nScrollMin;  //调整纵坐标使图形产生滚动
        Invalidate();        //重画图形
    }
}
```

OnVScroll 函数中,对鼠标按中滚动条的各种情况作出响应,通过调整视图屏幕基点的实际坐标达到图形移动的目的,具体操作已经在程序中做了详尽的解释。同样,可以实现横向滚动条的消息处理函数:

```
void CDrawView::OnHScroll(UINT nSBCode,UINT nPos,CScrollBar * pScrollBar)
{
    short nScrollInc;
    int nNewPos;
    switch(nSBCode)                             //判断按键位置
    {
        caseSB_TOP:                             //如果将滚动条滚动到左部
            nScrollInc = -nHScrollPos;          //得到相对移动的最小滚动距离数
            break;
        caseSB_BOTTOM:                          //如果将滚动条滚动到右部
            nScrollInc = nHScrollMax - nHScrollPos; //得到相对的最小滚动距离数
            break;
        caseSB_LINEUP:                          //如果按了滚动条中的向右的按键
            nScrollInc = -nXLine;               //得到相对移动的最小滚动距离数
            break;
        caseSB_LINEDOWN:                        //如果按了滚动条中的向左的按键
            nScrollInc = nXLine;                //得到相对移动的最小滚动距离数
            break;
        caseSB_PAGEUP:                          //如果按中了中间活动按钮的右部区域
            nScrollInc = -nXPage;               //得到相对移动的最小滚动距离数
            break;
        caseSB_PAGEDOWN:                        //如果按中了中间活动按钮的左部区域
            nScrollInc = nXPage;                //得到相对移动的最小滚动距离数
            break;
        caseSB_THUMBPOSITION:                   //如果用鼠标拖动中间活动按钮到一个位置
            nScrollInc = nPos - nHScrollPos;    //得到相对移动的最小滚动距离数
            break;
        default:
            nScrollInc = 0;
    }
    //得到滚动条实际的滚动位置(不能超出滚动条的横向滚动范围)
```

```
    nNewPos = max(0,min(nHScrollPos + nScrollInc,nHScrollMax));
    //得到实际的相对滚动范围(最小滚动距离数目)
    nScrollInc = nNewPos - nHScrollPos;
    if(nScrollInc)          //如果产生了滚动
    {
        nHScrollPos = nNewPos;                  //得到横向滚动条中新的滚动位置
        SetScrollPos(SB_HORZ,nHScrollPos);      //设置横向滚动条的位置
        UpdateWindow();                         //更新滚动条显示
        //调整横坐标使图形产生滚动
        m_xStart = m_xSetart + blc * nScrollInc * nScrollMin;
        Invalidate();       //重画图形
    }
}
```

横向滚动条的消息处理函数几乎与纵向滚动条的一模一样。至此实现了在矢量图形系统中使用滚动条进行图形移动的功能。现在实现的这个滚动功能，即使在早期版本的 VC++ 中，在图形屏幕是 600×400 像素的情况下，也能够在处理 2732 的放大倍数内正确的使用这个工具条，如果加大最小移动距离 nScrollMin 到 100 个像素，那么处理的放大倍数就达到了 5464 倍。这个放大倍数已能够满足无级放缩的矢量图形系统滚动视图的要求。

在以上实现的滚动操作中，在图形滚动后，整个视图屏幕重新绘制了一次(调用 Invalidate 函数)。这在有些滚动情况下(如滚动一屏或者滚动一屏以上)是必须的，但在大部分情况下(用鼠标点中滚动条的上下或左右按键进行图形滚动时)，只是部分屏幕进行了滚动，此时整个视图屏幕重画，一方面浪费了时间，另一方面产生不了滚动的效果。关于滚动视图操作时的局部重画方法和键盘滚动视图的操作可参考作者所著的《Visual C++ 高级编程技术——开发实例剖析》一书。

第 8 章　图形的删除和恢复

在本章中，将对文档的删除及恢复等操作进行讨论。并在矢量图形系统 Draw 中，实现图形元素的点选、图形元素的删除、删除的恢复，以及逆向操作等各种功能。

8.1　鼠标点选图形元素

为了实现图形的删除和恢复等操作功能，首先应该在矢量图形系统中实现图形的选中操作。

对于一个矢量图形系统来说，不但要能够实现交互图形绘制、放缩操作等，还应有一项基本的功能，就是能够选中图形元素。选中图形元素是对图形进行修改、删除、直接从图形上得到信息（在 GIS 系统中），以及在 GIS 系统中实现空间统计和分析功能等操作的基础。

图形元素的选中操作方法有多种，可以点选，也可以用区域选中。点选图形元素的原理很简单，当在视图屏幕上按下鼠标键时，就得到按下点的屏幕像素坐标，将这个像素坐标转换成实际坐标后，通过与所有的图形元素比较，判断这个点所在的图形元素即为选中的图形元素。区域选中图形元素的实现过程比点选复杂一些，是通过判断各种图形元素是否与一个区域相交或包含在一个区域内来实现的。区域选中操作最简单的是窗口选中（即矩形区域选中），另外还有任意多边形区域选中、圆形区域选中等，这些选中方法在实现 GIS 系统时尤其重要。本节，将对点选图形元素的实现方法及实现过程进行讨论，对于区域选中图形元素的一些算法，将在第 16 章中做介绍。

点选图形元素时，问题的关键在于如何判断一个点是否在一个图形元素上，也就是要确定每一类图形元素的选中区域。还是以直线、连续直线或多边形区域、圆、圆弧、标注文本等图形元素为例，对图形的点选操作进行讨论。

8.1.1　各种图形元素的选中区域

以下讨论应用程序 Draw 具有的各类图形元素的选中区域。图形元素的选中区域，指的是对于一类图形元素划定的一个区域，鼠标点中在哪些范围内能够选中这个图形元素。以下对各种图形元素的选中区域分别进行讨论。

(1) 直线和连续直线

在这里讨论的直线和连续直线，都是没有宽度的线。连续直线是多个直线的组合，所以连续直线的情况和直线是一样的，判断一个点是否在一条直线上是问题的关键。作为这种没有宽度的线，怎么确定它的图形选中区域呢？因为直线是没有宽度的，所以，这种选中判断实际上是要判断一个点与一条直线的距离是否在一定的距离范围之内。这个距离范围是可变的，规定距离范围较大，鼠标点在相对较远的位置就能选中直线；而规定距离范围较小，鼠标点必

须十分贴近直线时才能选中。问题的实质就是，如何确定这个判断选中的距离范围和怎么计算一个点与一条直线的距离。

有效距离

应用程序 Draw 是一个无级放缩的矢量图形系统，用鼠标在屏幕上点选图形元素的操作是在不同的比例下进行的，所以不能用实际的距离作为距离范围，即有效距离。从操作的直观性上看，这个距离应为在屏幕上的像素距离（因为在不同的分辨率下，一个像素代表的实际距离差别不大）。确定有效距离的方法是，首先确定在视图屏幕上有效距离的大小（如 n 个像素），也就是说，不管在什么比例的情况下，选中图形元素时都以在屏幕上与要选直线的距离小于 n 个像素为标准。用像素为单位进行计算时不方便，于是就求出 n 个像素在当时显示比例下的实际距离，这个实际距离就是点选图形元素时的有效距离。只要鼠标按中点到某条直线的实际距离小于这个有效距离，就算选中了这条直线。有效距离的大小为：

```
n * blc; //即 n 个像素的实际长度
```

blc 是当前视图屏幕的显示比例，n 是有效距离的像素数。

点与直线距离的计算方法

有效距离确定以后，接下来的问题是计算一个点到一条直线的距离。严格上说是一个点到一条线段的距离，现在给出计算的思路：

如图 8-1 所示，一个点 C 和一条顶点是 A 和 B 的线段，计算点 C 到线段 AB 距离，其步骤如下：

① 计算出三条边的长度 a,b,c；

② 比较 a 和 b，找出长度较小的那一条边，计算出这条边与直线的交角 ang；

③ 判断交角 ang 的大小，交角是钝角，长度较小的边的长度（a 或者 b）就是点到直线的距离（注意，这个距离不是指的垂直距离，而是点 C 到线段 AB 最近的距离）；交角是锐角，则用以下公式计算点 C 到线段 AB 的距离（此时是垂直距离）：

$$a(\text{或者 } b) \times \sin(ang)$$

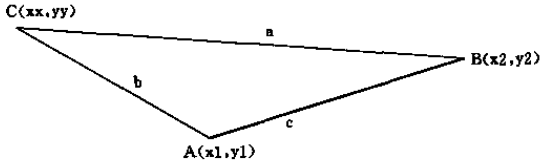


图 8-1　计算点 C 到线段 AB 的距离

为了完成计算点到直线距离的功能，需要定义一个函数，为了使这个函数在直线和连续直线类中都能够使用，把这个函数定义在图形元素基类 CDraw 中：

```
public:
    float PointLine(float xx,float yy,float x1,float y1,float x2,float y2);
```

参数分别是点 C(xx,yy)与直线两个顶点 A(x1,y1)和 B(x2,y2)的实际坐标，函数返回计算得到的距离值。在实现文件 drawdoc.cpp 中，加入函数实现代码：

```
float CDraw::PointLine(float xx,float yy,float x1,float y1,float x2,float y2)
{
    float a,b,c,ang1,ang2,ang;
    //计算三条边的长度
    a = CalDisp(x1,y1,xx,yy);
    if(a = =0.0)return 0.0;
    b = CalDisp(x2,y2,xx,yy);
    if(b = =0.0)return 0.0;
    c = CalDisp(x1,y1,x2,y2);
    //如果点(x1,y1)和(x2,y2)的坐标相同,即线段是一个点,退出函数并返回距离
    if(c = =0.0) return a;
    if(a<b)                  //如果点(xx,yy)到点(x1,y1)这条边较短
    {
      if(y1 = =y2)           //如果直线段 AB 是水平线,得到直线段 AB 的弧度
      {
          if(x1<x2)
            ang1 = 0;
          else
            ang1 = (float)pi;
      }
      else                   //如果直线段 AB 不是水平线,得到直线段 AB 与 X 轴正向夹角的弧度
      {
          ang1 = (float)acos((x2-x1)/c);
          if(y1>y2)
              ang1 = (float)pi*2-ang1;      //直线(x1,y1)-(x2,y2)与 X 轴正向夹角的弧度
      }
      ang2 = (float)acos((xx-x1)/a);
      if(y1>yy)ang2 = (float)pi*2-ang2;     //直线(x1,y1)-(xx,yy)与 X 轴正向夹角的弧度
      ang = ang2-ang1;
      if(ang<0)ang = -ang;
      if(ang>pi) ang = (float)pi*2-ang;
      if(ang>pi/2) return a;      //如果为钝角,直接返回距离
      else                        //如果为锐角,返回计算得到的距离
        return (a*(float)sin(ang));
    }
    else                          //如果(xx,yy)到点(x2,y2)这条边较短
    {
      if(y1 = =y2)                //如果两个点的纵坐标相同,则直接得到直线斜率的弧度
      {
        if(x1<x2)
          ang1 = (float)pi;
        else
          ang1 = 0;
```

```
        }
        else                    //如果两个点的纵坐标不同
        {
          angl = (float)acos((x1 - x2)/c);      //直线(x2,y2) - (x1,y1)斜率的弧度
          if(y2 > y1)angl = (float)pi * 2 - angl;
        }
        ang2 = (float)acos((xx - x2)/b);        //直线(x2,x1) - (xx,yy)斜率的弧度
        if(y2 > yy)ang2 = (float)pi * 2 - ang2;
        ang = ang2 - angl;
        if(ang < 0) ang = - ang;
        if(ang > pi) ang = (float)pi * 2 - ang;   //交角的大小
        if(ang > pi/2) return b;                  //如果为钝角,直接返回距离
        else                                      //如果是锐角
          return(b * (float)sin(ang));            //返回计算得到的距离
      }
    }
```

在上述函数中调用了一个计算两点距离的函数 CalDisp,它也是在 CDraw 类中定义的:

```
public:
    float CalDisp(float x1,float y1,float x2,float y2);
```

CalDisp 函数计算并返回两点(x1,y1)和(x2,y2)的距离,在实现文件 drawdoc.cpp 中,加入函数的实现代码:

```
float CDraw::CalDisp(float x1,float y1,float x2,float y2)
{
    return (float)sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}
```

(2) 圆和圆形区域

确定圆的选中区域非常简单,如果有效距离为 jl,计算得到的鼠标按中点到圆心的距离为 jl1;选中区域就是:

```
|jl1 - circle.r| < jl;
```

即按中点到圆的距离小于 jl。

对于圆形区域,选中区域则是:

```
jl1 < = jl;
```

(3) 圆弧

圆弧的选中区域同圆类似,首先要确定按中点是否在圆弧所在的圆上,既按中点到圆心的距离 jl1 符合:

```
|jl1 - circle.r| < jl;
```

因为圆弧是不封闭的,还要确定按中点相对于圆心的弧度是否在圆弧的弧度范围内,可以通过在 7.5.1 小节定义的 CArc 类中的成员函数 IsInArc 函数来确定。

(4) 多边形区域

图形元素多边形区域和标注文本(四边形)都属于这种情况。最简单的多边形区域就是没有旋转的矩形(如水平标注的文本就是这种情况)。判断一个点是否在这种矩形中的算法非常简单,现在讨论的是点与任意边数、任意形状的多边形区域的关系,需要一个通用的算法来解决这个问题。VC++ 的类 CRgn 中提供了一个成员函数来实现这个操作功能(见 3.2.5 小节)。CRgn 类封装了一些进行区域操作的函数,可以直接解决按中点是否在多边形区域中的问题。以下是实现步骤:

① 首先定义一个 CRgn 对象:

```
CRgn rgn; //定义一个 CRgn 对象
```

② 把 CRgn 对象初始化成一个具有 n 个顶点的多边形区域:

```
rgn.CreatePolygonRgn(bbcc,n,0);
```

参数 bbcc 是一个 POINT 型的指针,存储着参数 n 个顶点的坐标,第三个参数存储的是初始化的模式。函数的详细操作参考 VC++ 的类库手册。

③ 建立一个多边形区域以后,调用 PtInRegion 函数来判断一个点是否在区域内。

```
rgn.PtInRegion(x1,y1);
```

函数 PtInRegion 是 CRgn 类的成员函数,它可以判断点(x1,y1)是否在初始化的区域内。如果左区域内函数返回非 0 值,否则返回 0。

在 CDraw 类中建立一个函数完成这个操作:

```
public:
BOOL PointRgn(float x,float y,int Numble,PointStruct * PointList,float blc);
```

参数 x 和 y 是按中点的实际横纵坐标,Numble 是多边形顶点的数目,顶点的实际坐标放在结构数组 PointList 中,blc 为将实际坐标转换成逻辑坐标的比例尺。当按中点在多边形区域中时返回 TRUE,否则返回 FALSE。在实现文件 drawdoc.cpp 中,加入函数的实现代码:

```
BOOL CDraw::PointRgn(float x,float y,int Numble,PointStruct * PointList,float blc)
{
    CRgn rgn;                    //定义一个 CRgn 对象
    int x1,y1,i;
    POINT * bbcc;                // 定义一个点结构指针
    if(Numble < 3)
        return 0;                //如果顶点的数目 < 3,即不是一个区域,返回不选中标志
    bbcc = new POINT[Numble];    //给结构指针动态分配 Numble 顶点
    //将按中点的坐标转变成像素坐标
    x1 = (int)(x/blc);
```

```
    y1 = (int)(y/blc);
    //将多边形区域各顶点的坐标转成像素坐标
    for(i = 0;i < Numble;i++)
    {
        bbcc[i].x = (int)((PointList[i].x)/blc);
        bbcc[i].y = (int)((PointList[i].y)/blc);
    }
    rgn.CreatePolygonRgn(bbcc,Numble,0);  //初始化一个多边形区域
    i = (rgn.PtInRegion(x1,y1)! =0);      //如果点在区域内 j = 1,否则 j = 0;
    rgn.DeleteObject();                   //删除定义的 rgn 对象
    return i;
}
```

8.1.2　各种图形元素的点选判断函数

本小节在各个图形元素类中实现点选判断函数，与在图形元素类中增加绘制函数(Draw函数)和边界矩形计算函数(GetRect)类似，也需要把点选函数定义成虚函数的形式。在图形基类 CDraw 中定义一个纯虚函数：

```
public:
    virtual BOOL IsPoint(float x,float y,float jl,float blc) = 0;
```

在各个图形元素类 CLine、CPline、CCircle、CArc、CText 中重载虚函数 IsPoint，即在这些类中加入这个函数的重载定义：

```
public:
    virtual BOOL IsPoint(float x,float y,float jl,float blc) = 0;
```

IsPoint 函数用来判断是否点选了图形元素，如果选中了图形元素，函数返回 TRUE，否则返回 FALSE。参数(x,y)是点的实际坐标，参数 jl 是选择操作的有效距离，blc 是将实际坐标转换成逻辑坐标时的比例尺(在多边形区域点选操作时使用)。

以下讨论实现各个图形元素类重载的 IsPoint 函数的实现代码，实现各个图形元素类的点选判断功能。

(1) 直线

为了完成对直线的点选判断，在实现文件 drawdow.cpp 中，加入 CLine 类的重载函数 IsPoint 的实现代码：

```
BOOL CLine::IsPoint(float x,float y,float jl,float blc)
{
    float xx,x1,x2,y1,y2;
    if(b_Delete)                    //如果图形元素作了删除标志,返回没有选中标志
        return FALSE;
    GetRect(&x1,&y1,&x2,&y2); //得到直线的边界矩形
```

```
    if(!(x>=x1&&x<=x2&&y>=y1&&y<=y2))
        return FALSE;              //如果按中点不在边界矩形内,函数返回 FALSE
    xx=PointLine(x,y,m_X1,m_Y1,m_X2,m_Y2);     //得到按中点到直线的距离
    if(xx<jl)                  //如果点到直线的距离小于有效距离,则选中了直线
        return TRUE;
    return FALSE;
}
```

CLine 类的 IsPoint 函数的主要操作步骤如下：

① 首先判断按中点(x,y)是否在直线的边界矩形内,如果不在,函数返回 FALSE。

② 如果按中点在直线的边界矩形内,调用 PointLine 函数计算按中点到直线的距离 xx。如果距离 xx 小于有效距离 jl,即选中了直线,函数返回 TRUE,否则,函数返回 FALSE。

(2) 连续直线和多边形区域

为了完成对连续直线和多边形区域的点选判断,在实现文件 drawdow.cpp 中,加入 CPline 类的重载函数 IsPoint 的实现代码：

```
BOOL CPline::IsPoint(float x,float y,float jl,float blc)
{
    int i;
    float xx,x1,y1,x2,y2;
    if(b_Delete)                    //如果图形元素做了删除标志,函数返回 FALSE
        return FALSE;
    GetRect(&x1,&y1,&x2,&y2);   //得到连续直线或多边形区域的边界矩形
    if(!(x>=x1-jl&&x<=x2+jl&&y>=y1-jl&&y<=y2+jl)) //如果按中点不在边界矩形内
                                                  //则返回 FALSE
        return FALSE;
    if(!b_Fill)                     //如果是连续直线
    {
        for(i=0;i<m_Numble-1;i++) //依次对连续直线的每条直线进行判断
        {
            xx=PointLine(x,y,m_PointList[i].x,m_PointList[i].y,
                m_PointList[i+1].x,m_PointList[i+1].y);      //计算按中点到直线段的距离
            if(xx<jl)           //如果点到直线段的距离小于有效距离,函数返回 TRUE
                return TRUE;
        }
    }
    else                            //如果是多边形区域
    {
        if(PointRgn(x,y,m_Numble,m_PointList,blc))              //如果按中点在多边形区域内
            return TRUE;
    }
    return FALSE;
```

```
}
```

CPline 类的 IsPoint 函数的主要操作步骤如下：

① 判断按中点(x,y)是否在这条连续直线(或多边形区域)的边界矩形之内。

② 如果按中点(x,y)在边界矩形内，则要判断这个元素是连续直线还是多边形区域。

③ 如果这个图形元素是连续直线，就要对连续直线的各直线段进行循环，调用函数 PointLine 计算按中点到直线的距离 xx，并根据距离是否小于有效距离 jl，来判断按中点是否选中了连续直线。

④ 如果这个图形元素是多边形区域，就直接调用函数 PointRgn 来判断按中点是否在多边形区域内。

(3) 圆和圆形区域

为了完成对圆和圆形区域的点选判断，在实现文件 drawdow.cpp 中，加入 CCircle 类的重载函数 IsPoint 的实现代码：

```
BOOL CCircle::IsPoint(float x,float y,float jl,float blc)
{
    float xx;
    if(b_Delete)                    //如果图形元素做了删除标志，函数返回 FALSE
        return FALSE;
    xx = CalDisp(x,y,m_CircleX,m_CircleY);          //计算按中点到圆心的距离
    if(! b_Fill)                    //如果是普通圆
    {
        if((xx > m_CircleR - jl&&xx < m_CircleR + jl)) //如果按中点到圆的距离小于有效距离
            return TRUE;
    }
    else                            //如果是圆形区域
    {
        if(xx < = m_CircleR)        //如果按中点在圆形区域内，函数返回 TRUE
            return TRUE;
    }
    return FALSE;
}
```

CCircle 类的 IsPoint 函数首先计算按中点(x,y)到圆心的距离 xx，然后根据当前图形元素是圆还是圆形区域，分别判断按中点是否选中了图形元素。

(4) 圆弧

判断点选圆弧的操作同圆类似，首先判断按中点是否在圆上，如果在圆上，则继续判断按中点是否在圆弧的弧度范围内。为了完成对圆弧的点选判断，在实现文件 drawdow.cpp 中，加入 CArc 类的重载函数 IsPoint 的实现代码：

```
BOOL CArc::IsPoint(float x,float y,float jl,float blc)
{
    if(CCircle::IsPoint(x,y,jl))      //调用 CCircle 类的 IsPoint 函数判断是否选中了圆
    {
        float xx = CalDisp(x,y,m_CircleX,m_CircleY);     // 按中点到圆心的距离
        float ang = (float)acos((x - m_CircleX)/xx);     // 按中点相对于圆心的弧度
        if(y - m_CircleY < 0)
            ang = (float)pi * 2 - ang;
        if(IsInArc(ang))      //如果按中点相对于圆心的弧度在圆弧的弧度范围内,则返回 TRUE
            return TRUE;
    }
    return FALSE;             //如果没有选中圆弧,返回 FALSE
}
```

CArc 类的 IsPoint 函数的主要操作步骤如下：

① 调用基类(CCircle)函数 IsPoint 判断按中点是否选中圆弧所在的圆。

② 如果按中点选中了圆,计算按中点相对于圆心的弧度,并判断弧度是否在圆弧的弧度范围内。

(5) 标注文本

标注文本的点选操作与多边形区域类似。一个标注文本实际上是一个矩形区域,根据字体的高度、宽度、间距、标注行的角度、标注信息的字符数,可以确定出标注矩形的 4 个顶点的实际坐标,然后判断按中点是否在矩形区域内。在实现文件 drawdow.cpp 中,加入 CText 类的重载函数 IsPoint 的实现代码：

```
BOOL CText::IsPoint(float x,float y,float jl,float blc)
{
    PointStruct XY[4];
    float x1,y1,x2,y2;
    if(b_Delete)                  //如果图形元素做了删除标志,函数返回 FALSE
        return FALSE;
    GetRect(&x1,&y1,&x2,&y2); //得到标注文本的边界矩形
    if(!(x>=x1&&x<=x2&&y>=y1&&y<=y2))     //如果按中点不在边界矩形内,函数返回
                                          //FALSE
        return FALSE;
    float ang = (float)pi * m_Angle1/180;
    //以下得到标注文本四个顶点的实际坐标
    XY[0].x = m_StartX;
    XY[0].y = m_StartY;
    XY[1].x = m_StartX + (float)((m_TextWide + m_OffWide) * m_TextLong * cos(ang));
    XY[1].y = m_StartY + (float)((m_TextWide + m_OffWide) * m_TextLong * sin(ang));
    XY[2].x = XY[1].x - (float)(sin(ang) * m_TextHeight);
    XY[2].y = XY[1].y + (float)(cos(ang) * m_TextHeight);
```

```
    XY[3].x = m_StartX - (float)(sin(ang) * m_TextHeight);
    XY[3].y = m_StartY + (float)(cos(ang) * m_TextHeight);
    if(PointRgn(x,y,4,XY,blc))      //如果按中点在多边形区域内,函数返回 TRUE
        return TRUE;
    return FALSE;
}
```

CText 类的 IsPoint 函数的主要操作步骤如下：

① 判断按中点是否在标注文本的边界矩形内，如果不在，函数返回 FALSE。

② 得到标注文本的四个顶点的实际坐标。

③ 调用函数 PointRgn 判断按中点是否在标注文本的四边形区域内。

8.1.3　选中图形元素的特殊显示

当用鼠标在屏幕上点选图形元素时，通过计算可以判断是否选中了图形元素，但怎样让用户知道是否已经选中了图形元素呢？一是可以给出提示信息，告诉用户选中了图形元素；二是把选中的图形用特殊的方式显示。选中图形元素后，马上用特殊的形式把图形元素显示出来，会起到很好的效果。特殊显示的方式很多，如可以虚线显示、变色显示、闪烁、用箭头指示等，让用户一目了然。实现图形元素的特殊显示，实际上就是用不同的形式重新绘制这个图形元素。在矢量图形系统 Draw 中，要求选中线类图形元素时，用虚线显示；选中区域类图形元素时，就用虚边线、变色或反色显示。各个图形元素类的绘制函数 Draw(见 6.2.2 小节)中，有一个参数 DrawModel，是用来确定图形元素显示方式的，读者可以对 Draw 函数进行修改，增加更多的显示方式。

8.1.4　实现点选图形元素功能

各类图形元素有了选中判断和特殊显示函数后，就可以具体的组织图形元素的点选功能了。在系统中再加入一个新的菜单来完成点选图形元素的操作。

(1) 增加一个菜单及消息处理函数

用 ResourceView 及其相应工具在菜单资源 IDR_DRAWTYPE 中增加一个菜单“图形选中”，在这个菜单中包含“鼠标点选”和“放弃选中”二个菜单项。增加完毕后，在资源文件 draw.rc 中增加了新增菜单项的定义：

```
IDR_DRAWTYPE MENU PRELOAD DISCARDABLE
BEGIN
    ......//省略以上代码
    POPUP "图形选中"
    BEGIN
        MENUITEM "&S 鼠标点选",          ID_SELECT_MOUSE
        MENUITEM "&U 放弃选中",          ID_SELECT_CLEAR
    END
```

```
    ......//省略以下代码
END
```

在资源 ID 号定义文件 resoruce.h 给菜单资源定义 ID 值：

```
#define ID_SELECT_CLEAR 213
#define ID_SELECT_MOUSE 214
```

用 ClassWizard 给这两个新增菜单项在视图类 CDrawView 中增加消息处理函数，增加完毕后，应用程序 Draw 的代码修改如下：

① 在视图类 CDrawView 中增加了消息处理函数的定义代码：

```
//{{AFX_MSG(CDrawView)
......//省略以上代码
afx_msg void OnSelectMouse();      //鼠标点选图形元素操作功能
afx_msg void OnSelectClear();      //放弃对图形元素的选中
//}}AFX_MSG
```

② 在实现文件 drawview.cpp 中加入消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    ......//省略以上代码
    ON_COMMAND(ID_SELECT_MOUSE, OnSelectMouse)
    ON_COMMAND(ID_SELECT_CLEAR, OnSelectClear)
END_MESSAGE_MAP()
```

③ 在实现文件 drawview.cpp 中加入函数的实现代码：

```
void CDrawView::OnSelectMouse()
{
    m_DrawCurrent = 30;          //标识进行图形元素点选操作
    PushNumb = 0;
}

void CDrawView::OnSelectClear()
{
}
```

函数 OnSelectMouse 在点中菜单项“图形点选”后执行，函数代码只是为了能够使程序识别正在进行图形元素点选操作(m_DrawCurrent = 30)，函数 OnSelectClear 完成取消所有选中图形元素的功能(这个函数将在 8.1.5 小节中讨论)。

(2) 保留选中的图形元素

在图形选中操作过程中，一般并不是只选择一个图形元素，往往需要对大量的图形元素进行选择。这就需要系统有同时选中并保留大量图形元素的能力。保留选中的图形元素，有两个方面的含义：一方面是数据的保留，即保留已经选中的那些图形元素类别和序号；另一方面

是视图屏幕上的保留，即选中的图形元素一直特殊显示在屏幕上。如果第二个方面不保留，当进行图形放大、重画等涉及到图形重画的操作时，特殊显示就不存在了。

数据保留

要实现数据保留，就需要建立数据结构来存储选中的图形元素。在文档类 CDrawDoc 中定义一个结构数组和一个变量：

```
protected:
    GraphSelectStruct * GraphSelect;    //存储选中的图形元素
    int n_GraphSelect;                  //选中图形元素的数目
```

结构 GraphSelectStruct 在头文件 drawdoc.h 中(CDrawDoc 类前)定义为：

```
typedef struct
{
    short int Lb;           //图形类别,1-直线,2-连续直线,3-圆,4-圆弧,5-文本
        int Index;          //图形元素的序列号
        int id_only;        //图形元素的识别号
}GraphSelectStruct;         //用来存储选中图形元素的结构
```

将选中的图形元素存储在结构数组 GraphSelect 中，成员变量 Lb 存储选中图形元素的类别，成员变量 Index 存储图形元素的序列号，即图形元素在指针数组中的下标，成员变量 id_only 存储这个图形元素唯一的识别号，即图形元素中的成员变量 m_id_only。

在实现文件 drawdoc.cpp 中，在 CDrawDoc 文档类的构造函数内动态地分配结构数组 GraphSelect：

```
CDrawDoc::CDrawDoc()
{
    ......//省略
    GraphSelect = new GraphSelectStruct[2000];      //最多可选中 2000 个图形元素
}
```

在 CDrawDoc 类的析构函数中删除动态分配的结构数组：

```
CDrawDoc::~CDrawDoc()
{
    ......//省略以上操作
    delete GraphSelect;
}
```

为了将一个选中的图形元素存储到结构数组 GraphSelect 中。在文档类 CDrawDoc 中定义一个函数：

```
public:
    BOOL AddSelectList(int Lb, int Index,int Id_Only);
```

AddSelectList 函数在结构数组 GraphSelect 中增加一个选中的图形元素，增加成功时函数返回 TRUE，否则函数返回 FALSE。参数 Lb 表示选中图形元素的类别，Index 是选中图形元素在

记录图形元素对象指针的 CObArray 对象中的下标序号，Id _ Only 是图形元素的识别号。在实现文件 drawdoc.cpp 中加入函数的实现代码：

```
BOOL CDrawDoc::AddSelectList(int Lb, int Index,int Id _ Only)
{
    for(int i=0;i<n _ GraphSelect;i++)          //判断这个图形元素是否已经被选中
    {
        if(Lb==GraphSelect[i].Lb&&Index==GraphSelect[i].Index&&
            Id _ Only==GraphSelect[i].id _ only)//如果这个图形元素已经被选中，函数返回 FALSE
            return FALSE;
    }
    //将选中图形元素的类别、序列号和识别号存入结构数组，并使选中的图形数目加 1
    GraphSelect[n _ GraphSelect].Lb=Lb;
    GraphSelect[n _ GraphSelect].Index=Index;
    GraphSelect[n _ GraphSelect++].id _ only=Id _ Only;
    return TRUE;
}
```

AddSelectList 函数首先检查当前被选中的图形元素是否在此之前已经被选中，如果这个图形元素还没有被选中，把图形元素的类别、序列号和识别号存入数组 GraphSelect，并把选中的图形数目 n _ GraphSelect 增加 1。

屏幕保留

为了能够将选中的图形元素在屏幕上保留，需要修改图形绘制函数，使视图重画时，选中的图形元素仍然能特殊显示。

为了选择绘制某个图形元素，在文档类 CDrawDoc 中定义一个函数：

```
private:
    void DrawGraph(CDC* pDC,int Lb,int Index,int Mode,int Mode1,short BackColor);
```

参数 Lb 是要绘制图形元素的类别，参数 Index 是要绘制的图形元素在对象指针数组中的序号，参数 Mode 是绘制时采用的覆盖模式，参数 Mode1 是绘制方式。在实现文件 drawdoc.cpp 中加入函数的实现代码：

```
void CDrawDoc::DrawGraph(CDC* pDC,int Lb,int Index,int Mode,int Mode1,short BackColor)
{
    GetGraph(Lb,Index)->Draw(pDC,Mode,Mode1,BackColor);
}
```

在文档类 CDrawDoc 的视图绘制函数 Draw 中（在 5.3.4 小节实现），增加代码特殊绘制被选中的图形元素：

```
void CDrawDoc::Draw(CDC *pDC,int m _ DrawMode,int m _ DrawMode1,short BackColor)
{
    for(int i=1;i<=5;i++)
    {
```

```
        int nn = GetGraphUpperBound(i) + 1; //得到存储图形元素对象指针数组的最大下标加 1
        while(nn - - )      //遍历所有的图形元素,得到图形元素对象指针并绘制图形元素
            GetGraph(i,nn) - > Draw(pDC,m_DrawMode,m_DrawMode1,BackColor);
    }
    for(i = 0;i < n_GraphSelect;i++ )      //对所有选中的图形元素循环,特殊绘制这些图形元素
        DrawGraph(pDC,GraphSelect[i].Lb,GraphSelect[i].Index,0,1,BackColor);
}
```

(3) 实现点选图形元素功能

为了能够对所有的图形元素进行检查,计算并返回选中的图形元素的参数,在文档类 CDrawDoc 中定义一个函数:

```
public:
    BOOL PointSelect(float x,float y,float jl,float blc,int * Lb,int * Index,int * id_only);
```

PointSelect 函数用来计算得到点选的图形元素,如果选中了图形元素,函数返回 TRUE,通过指针参数 * Lb、* Index 和 * id_only 分别返回图形元素的类别、序列号以及识别号,否则函数返回 FALSE。参数(x,y)是按中点的实际坐标,参数 jl 是点选操作的有效距离,参数 blc 是实际坐标转成逻辑坐标的比例尺。在实现文件 drawdoc.cpp 中加入函数的实现代码:

```
BOOL CDrawDoc::PointSelect(float x,float y,float jl,float blc,int * Lb,int * Index,int * pbh)
{
    for(int i = 1;i < = 5;i++ )   //依次对直线、连续直线、圆、圆弧、标注文本等五种图形元素循环
    {
        int nn = GetGraphUpperBound(i) + 1;   //得到存储第 i 类图形元素对象指针的 CObArray
                                              //对象的最大数组下标加 1
        while(nn - - )            //对一个图形类所有的图形元素对象遍历
        {
            if(GetGraph(i,nn) - > IsPoint(x,y,jl,blc))      //如果点选了图形元素
            {
                * Index = nn;  //图形元素对象在对象数组中的序号
                * Lb = i;      //图形元素的类别
                * pbh = GetGraph(i,nn) - > GetID(); //图形元素的识别号
                return TRUE;
            }
        }
    }
    return FALSE;
}
```

PointSelect 函数的主要操作步骤如下:

① 依次对直线、连续直线、圆、圆弧、标注文本等五种类别的图形元素进行点选操作。

② 对于每一类图形元素,通过 GetGraphUpperBound 函数得到存储这类图形元素对象指针

数组的最大下标加 1。

③ 通过数组的最大下标对数组的所有项进行遍历,得到一个类别中的各个图形元素对象的指针,调用 IsPoint 函数判断这个图形元素是否被选中,如果选中,则通过指针变量 * Lb、* Index和 * id _ only 返回图形元素的类别、图形元素指针在数组中记录的序号,以及图形元素的识别号,函数返回 TRUE。

④ 如果所有图形元素都没有被选中,函数返回 FALSE。

值得注意的是,在调用 IsPoint 函数时,根据虚函数和由此引发的多态性,根据 CDraw 类指针指向的图形元素对象的图形类别,分别调用了各个图形元素类的 IsPoint 函数。

为了在应用程序 Draw 中实现点选图形元素的功能,需要在实现文件 drawview.cpp 中,在视图类的消息处理函数 OnLButtonDown 中加入以下代码:

```
void CDrawView::OnLButtonDown(UINT nFlags,CPoint point)
{
    …… //省略以上操作
    else if(m _ DrawCurrent = = 30)                    //进行点选图形元素操作
    {
        if(pDoc - > n _ GraphSelect = = 2000)    //如选中的图形元素数是 2000,则不再选中
        {
            AfxMessageBox("最多只能选择 2000 个图形元素");
            return;
        }
        VPtoDP(point.x,point.y,&xx1,&yy1);    //得到按中点的实际坐标
        float jl = blc * 4;                              //点选判断的有效距离(4 个像素的实际距离)
        //调用文档类的 PointSelect 函数得到选中的图形元素参数
        BOOL pb = pDoc - > PointSelect(xx1,yy1,jl,blc,&Lb,&Index,&Index1);
        if(pb)     //如果选中了某个图形元素,记录下选中的图形元素并特殊显示这个图形元素
        {
            BOOL pb1 = pDoc - > AddSelectList(Lb,Index,Index1);
            if(pb1)      //如果记录选中的图形元素成功,则特殊显示这个图形元素
                pDoc - > DrawGraph(&ht,Lb,Index,0,1);
        }
    }
}
```

完成的主要操作功能如下:

① 在进行鼠标点选图形元素操作时,当用鼠标左键点中屏幕时,计算出鼠标按中点的实际坐标(xx1,yy1)和点选判断的有效距离 jl。

② 通过调用文档类 CDrawDoc 的函数 PointSelect 计算选中的图形元素,如果选中了图形元素,调用文档类 CDrawDoc 的 AddSelectList 函数记录选中的图形元素。

③ 如果调用 AddSelectList 函数记录选中的图形元素成功(即这个图形元素在此之前还没有被选中),将这个图形元素特殊显示。

至此,完成了鼠标点选图形元素的操作功能。对应用程序 Draw 进行编译后运行,执行"图形

选中"菜单下的"鼠标点选"菜单项时,就可以用鼠标在视图中点选各个图形元素。对于选中的图形元素进行特殊显示,线类图形元素是虚线显示、区域类图形元素是反色显示(如图 8-2)。

图 8-2　选中图形元素后的显示界面

本小节实现了对直线、连续直线或封闭多边形、圆或圆区域、圆弧、文本等几个典型图形元素的点选操作。对于其他类型的图形元素,读者可以参考以上方法,根据图形元素的几何特征独立完成。

8.1.5　放弃选中的图形元素

有选中图形元素的功能,就需要有相应的放弃选中图形元素的功能。所谓"放弃",就是取消对图形元素的选中,使选中的图形元素数为 0(m_GraphSeleNumb = 0)。这从数据操作角度来说是非常简单的,但除了数据方面所做的修改外,还要在视图中把选中的图形元素恢复到正常显示状态下,从直觉上感觉到放弃了对图形元素的选中。对在 8.1.2 小节中增加的"放弃选中"菜单项的消息处理函数 OnSelectClear 进行修改,得到修改后的函数实现代码如下:

```
void CDrawView::OnSelectClear()
{
    CDrawDoc * pDoc = GetDocument();                    //得到指向当前文档对象的指针
    CClientDC ht(this);                                 //创建一个绘图对象
    for(int i=0;i<pDoc->n_GraphSelect;i++)              //对所有选中的图形元素原样绘制
        pDoc->DrawGraph(&ht,pDoc->GraphSelect[i].Lb,pDoc->GraphSelect[i].Index,0,0,0);
    pDoc->n_GraphSelect=0;                              //选中的图形元素数目为 0
}
```

当点中应用程序 Draw 的菜单项"放弃选中"时,视图中被选中的的图形元素恢复到正常显示状态,并把选中图形元素的数目 m_GraphSelect 置 0。至此,完成了放弃对图形元素的选中功能。

8.1.6　图形元素的显示和选中顺序

为叙述方便,应用程序 Draw 中对图形元素的绘制和点选操作,是按讲解的顺序进行的。

在实际系统中,安排图形元素的显示和点选顺序很重要,有以下两个主要原则。

(1) 区域类图形元素要显示在前,选中在后

这一点很重要。区域类图形元素一般都填充,它们在线类图形元素后显示会对已经绘制的线类图形元素形成覆盖。在点选图形元素操作时,如果首先对区域类图形元素进行点选判断,就无法或者说不能正常选中画在它上面的线类图形元素。如有一个多边形区域,区域上面又有一条直线,如果进行点选判断时先判断区域,就不能选中这条直线,即便增强系统的点选功能,使系统能够在选中已经选中的图形元素后,能够继续进行选中操作,也只能是在选中了区域类图形元素的前提下,才能选中直线。

(2) 对系统重要的图形元素点选时应该首先判断

这是为了优化系统的选中速度根据具体情况而定的。如在作者实现的铁路 GIS 系统中,有一类叫做"车站"的图形元素,在系统运行中,需要经常用鼠标查询车站的信息,在安排选中的次序时,把"车站"这种图形元素放在选中判断计算的前面能够提高系统的可操作性。

8.2 图形元素的删除

在 8.1 节实现的鼠标器点选图形元素的基础上,可以删除选中的图形元素。删除一个图形元素时,需要做两方面的删除操作:从数据中删除图形元素和从屏幕上即时删除图形元素。

8.2.1 删除操作的类型

在一个计算机软件系统中,根据删除后能不能进行恢复把删除类型分为两种:可以恢复型和不可恢复型。

(1) 可以恢复型

可以恢复型的删除,是在操作时只把要删除的数据部分作删除标记,系统正常运行时不再操作这种作了标记的数据项,从而从表面上达到了删除的目的,而删除的数据仍然存在。用过 DOS 操作系统删除文件的读者都知道,用 DOS 命令(del)删除的文件是可以恢复的(用 undelete 命令或 PCTools 等工具)。但在经过大量有关磁盘的操作(如建新文件等)之后,就会不能完全恢复删除的文件。这与文件的存储机制有关。DOS 管理文件的方法是把硬盘分为很多最小的存储单位(一般是 512 个字节),用一个叫做文件分配表进行管理,在磁盘中有一个文件目录表,每一个文件有一个目录项管理着这个文件的一些基本信息(如文件名、日期、第一个存储块的位置等)。当做删除操作时,这个目录项并不删除,只是作了一个删除标志,并对这个文件所占用的存储单元作了可以分配的标志,在没有经过磁盘操作时,这些存储单元没有被占用,就可以进行成功的恢复。如果经过大量的磁盘操作后,原来属于这个文件的存储单元被其他的文件占用了,这需就不能对删除的文件进行成功的恢复。所以 DOS 系统的删除机制虽然是一种可以被恢复的删除机制,但不是一种绝对可以恢复的机制。要实现这种绝对可以恢复的机制,就需在对要删除对象做删除标志的同时,属于这个删除对象所占用的空间不能改变为可分

配状态。

对矢量图形系统 Draw，实现可以恢复的删除，即对要删除的图形元素对象做删除标志。在图形元素基类 CDraw 中有一个成员变量 b _ Delete，要对一个图形元素进行删除，只需把这个变量的值设为 TRUE 即可：

```
b _ Delete = TRUE;
```

对图形元素进行操作(如绘制、选中等)时，不对处于删除状态的图形元素进行操作，就达到了删除的效果。如在进行图形点选操作时，在 IsPoint 函数中有以下代码：

```
if(b _ Delete)
    return FALSE;
```

在图形元素的绘制函数 Draw 中也有相应的代码。对删除的图形元素需恢复时，只要将这个图形元素的状态由删除改变成正常即可：

```
b _ Delete = FALSE;
```

(2) 不可恢复型

对图形元素进行删除操作时，如果删除图形元素的对象，从对象指针数组中删除指向要删除图形元素对象的指针，这种删除就是不可恢复型的。如对第 Index 条直线进行删除，操作如下：

```
delete m _ LineArray.GetLine(Index);      //删除直线对象
m _ LineArray.RemoveAt(Index);            //把对象指针从数组中删除
```

本节将对矢量图形系统实现可以恢复型的删除操作。

8.2.2　从屏幕上删除图形元素的方法

当把选中的图形元素进行删除时，要求在屏幕上实时地把要删除的图形元素去掉。为了达到这个目的，最简单的方法是把图形重新绘制，这时，做了删除标志的图形元素不再绘制，而从视图屏幕上消失。例如，在多视图的情况下调用 UpdateAllViews 函数，使与当前视图同文档的所有视图更新。这种方法有一个缺点，即每删除一个图形元素，图形都要重画一次，既耽误时间，又使软件的实用性受影响。所以，一般不采用这种全屏重画的方法来删除图形，而用以下两种方法来从视图中删除图形元素。

(1) 底色重画

从屏幕上删除线类图形元素(如直线，连续直线，圆，弧等)时，用屏幕底色重新绘制这些图形元素，图形元素就从屏幕上消失了。当一个图形元素与别的线类图形元素相交时，这种删除方法在交点处出现缺口，但并不影响删除效果。如果这个图形元素是在区域类的图形元素上面时(如在一个填充的多边形区域上或者在图像上)，会留下重画的线的痕迹，从视觉上来说，这种痕迹是可以接受的。如要完全消除这种痕迹，只能全部或局部重画图形。

(2) 局部重画

在删除一些区域类图形元素(如多边形区域或圆形区域)时。它的上面可能有很多别的图形元素,用"底色重画"的方法来删除,会擦除这些图形元素。这时就可以采用局部重画的方法,局部重画就是只重画要删除的图形元素的边界矩形内的区域,从而达到从视图屏幕上即时删除图形元素的目的。

激活局部图形重画的方法在 7.2.2 小节曾经作过介绍:

```
InvalidateRect(rr,TRUE);
```

rr 是一个 CRect 对象,它保存的是需要重画的矩形区域。InvalidateRect 函数的作用是使视图屏幕上 rr 所包含的矩形区域内的图形元素重画。

8.2.3 各类图形元素的删除操作

对图形元素进行删除操作时,采用的是可恢复的删除方法,所以,只需对图形元素做删除标志即可。

(1) 删除图形元素

为了给一个图形元素做删除标志,在图形元素基类 CDraw 中定义一个函数:

```
public:
void Delete(BOOL Is); //删除或恢复删除图形元素的函数
```

Delete 函数有一个参数 Is,当 Is 为 TRUE 时,给图形元素做删除标志,当 Is 为 FALSE 时,去掉图形元素的删除标志。在实现文件 drawdoc.cpp 中加入函数的实现代码:

```
void CDraw::Delete(BOOL Is)
{
    if(Is)          //如果是删除操作
        b_Delete = TRUE;
    else            //如果是恢复操作
        b_Delete = FALSE;
}
```

因为 CDraw 类是各种图形元素类的基类,所以各个图形元素类都可以使用这个函数来对图形元素做删除标志或取消删除标志。

(2) 在视图类中实现删除图形元素的函数

为了在视图类 CDrawView 中,任意给一个指定的图形元素做删除标志,并从视图屏幕上删除这个图形元素,在视图类 CDrawView 中定义一个函数:

```
public:
    void Delete(CDC * pDC,int Lb,int Index);
```

参数 Lb 是图形元素的类别(1 - 直线,2 - 连续直线,3 - 圆,4 - 圆弧,5 - 标注文本),参数 Index 为图形元素在对象指针数组中的序号。在实现文件 drawview.cpp 中加入函数的实现代码:

```
void CDrawView::Delete(CDC * pDC,int Lb,int Index)
{
    float x1,y1,x2,y2;
    CDrawDoc * pDoc = GetDocument();   //得到指向当前文档对象的指针
    CDraw * pDraw;
    pDraw = pDoc -> GetGraph(Lb,Index); //得到指向要删除的图形元素对象的指针
    pDraw -> Delete(1);                 //调用 Delete 函数给图形元素做删除标志
    if(Lb == 1)          //如果图形元素是直线,以屏幕底色重画直线达到删除的效果
        pDraw -> Draw(pDC,0,2,m_bColor);
    else if(Lb == 2)     //如果图形元素是连续直线或多边形区域
    {
        if(! pDraw -> IsFill()) //如果是连续直线,则以屏幕底色重画连续直线达到删除的效果
            pDraw -> Draw(pDC,0,2,m_bColor);
        else          //如果是多边形区域,则得到其边界矩形并重画这个区域
        {
            pDraw -> GetRect(&x1,&y1,&x2,&y2);        //得到多边形区域的边界矩形
            ReDrawRect(x1,y1,x2,y2);                  //重画多边形区域的边界矩形区域
        }
    }
    else if(Lb == 3)     //如果是圆或圆形区域
    {
        if(! pDraw -> IsFill())     //如果是圆,则以屏幕底色进行重画圆以达到删除效果
            pDraw -> Draw(pDC,0,2,m_bColor);
        else     //如果是圆形区域,则得到其边界矩形并重画这个区域
        {
            pDraw -> GetRect(&x1,&y1,&x2,&y2);        //得到边界矩形
            ReDrawRect(x1,y1,x2,y2);                  //重画边界矩形区域
        }
    }
    else if(Lb == 4)     //如果是圆弧,则以屏幕底色进行重画以达到删除效果
    {
        pDraw -> Draw(pDC,0,2,m_bColor);
    }
    else if(Lb == 5)     //如果是标注文本,得到其边界矩形并重画这个矩形区域
    {
        pDraw -> GetRect(&x1,&y1,&x2,&y2);        //得到标注文本的边界矩形
        ReDrawRect(x1,y1,x2,y2);                  //重画边界矩形区域
    }
}
```

视图类 CDrawView 的 Delete 函数的主要操作步骤如下：

① 得到指向要删除图形元素对象的 CDraw 类指针 pDraw，调用 CDraw 类的函数 Delete 给要删除的图形元素做删除标志。

② 根据图形的类别(Lb)，对于线类(直线、连续直线、圆、圆弧)图形元素，用屏幕底色(m_bColor)重画得到删除的效果。

③ 对于多边形区域和圆形区域等区域类图形元素，则调用 GetRect 函数得到图形元素的边界矩形，然后调用 ReDrawRect 函数重画这个边界矩形区域来从视图中删除图形元素。

在 CDrawView 类的 Delete 函数中，调用了 ReDrawRect 函数来重画视图屏幕中的一个区域，ReDrawRect 函数在视图类 CDrawView 中定义：

```
private:
    void ReDrawRect(float X1,float Y1,float X2,float Y2);
```

ReDrawRect 函数的实现代码被加入到实现文件 drawview.cpp 中：

```
void CDrawView::ReDrawRect(float X1, float Y1, float X2, float Y2)
{
    float xx1,xx2,yy1,yy2;
    CRect r1;
    //以下是视图屏幕矩形区域的实际坐标
    xx1 = xMinScreen;
    xx2 = xMaxScreen;
    yy1 = yMinScreen;
    yy2 = yMaxScreen;
    //得到需绘制区域与视图屏幕区域相交的矩形
    BOOL IsCross = RectCross(&xx1,&yy1,&xx2,&yy2,X1,Y1,X2,Y2);
    if(IsCross)                    //如果相交,则重画这一相交区域
    {
        //以下得到这个相交区域的逻辑坐标
        r1.left = (int)((xx1 - m_xStart)/blc) - 1;
        r1.right = (int)((xx2 - m_xStart)/blc) + 1;
        r1.top = m_hScreen - (int)((yy2 - m_yStart)/blc) - 1;
        r1.bottom = m_hScreen - (int)((yy1 - m_yStart)/blc) + 1;
        InvalidateRect(r1);     //重画矩形区域 r1
    }
}
```

在 ReDrawRect 函数中，调用了函数 RectCross 得到重画区域与视图屏幕区域{xMinScreen, yMinScreen, xMaxScreen,yMaxScreen}的相交矩形，如果两个区域相交，将相交矩形的实际坐标转换为逻辑坐标，用 InvalidateRect 函数激发重画这一个区域。函数 RectCorss 在视图类 CDrawView 中定义：

```
private:
BOOL RectCross(float * x1,float * y1,float * x2,float * y2,float xx1, float yy1,float xx2,float yy2);
```

函数 RectCross 用来计算两个矩形区域的相交矩形。参数{ * x1, * y1, * x2, * y2}和{xx1, yy1,xx2,yy2}代表两个矩形的对角顶点坐标。两个矩形区域相交时,函数返回 TRUE,并通过指针参数{ * x1, * y1, * x2, * y2}返回相交矩形的对角顶点坐标。函数的实现代码在实现文件 drawview.cpp 中定义:

```
BOOL CDrawView::RectCross(float * x1,float * y1,float * x2,float * y2,float xx1,float yy1,float xx2,float
    yy2)
{
    float m_X1,m_Y1,m_X2,m_Y2;
    m_X1 = * x1; m_Y1 = * y1; m_X2 = * x2 ;m_Y2 = * y2;
    if(m_X1 > xx2||m_X2 < xx1||m_Y1 > yy2||m_Y2 < yy1) //两个矩形区域不相交
                                                        //函数返回 FALSE
        return FALSE;
    else                    //两个矩形相交,得到相交矩形的对角顶点坐标,函数返回 TRUE
    {
        * x1 = max(m_X1,xx1);
        * y1 = max(m_Y1,yy1);
        * x2 = min(m_X2,xx2);
        * y2 = min(m_Y2,yy2);
        return TRUE;     //函数返回 TURE
    }
}
```

另外,在视图类 CDrawView 的 Delete 函数实现代码中,调用了一个成员函数 IsFill 来判断图形元素是否填充,通过 IsFill 函数判断一个图形元素是连续直线还是多边形区域、一个图形元素是圆还是圆形区域。IsFill 函数在图形元素基类 CDraw 中被定义成虚函数:

```
public:
    virtual BOOL IsFill();
```

IsFill 函数的实现代码在实现文件 drawdoc.cpp 中定义:

```
BOOL CDraw::IsFill()
{
    return FALSE;
}
```

在图形元素类 CPline 和 CCircle 中,重载这个虚函数 IsFill,即在这两个类中增加虚函数的重载定义:

```
public:
    virtual BOOL IsFill();
```

在实现文件 drawdoc.cpp 中,加入了这两个重载函数的实现代码:

```
BOOL CPline::IsFill()
{
```

```
    return b_Fill;
}

BOOL CCircle::IsFill()
{
    return b_Fill;
}
```

这样，在 CDrawView::Delete 函数中调用 IsFill 函数时，根据多态性分别执行了 CPline 类和 CCircle 类中的 IsFill 函数，来判断图形元素是否填充（根据是否填充来确定是线类还是区域类）。也可以通过把成员变量 b_Fill 定义在图形元素基类 CDraw 中，通过在 CDraw 类中定义一个成员函数 IsFill（非虚函数）来直接判断图形元素是否填充。

8.2.4　实现图形元素删除操作

在 8.2.3 节讨论的删除方法和实现的操作函数基础上，可以在应用程序 Draw 中实现图形元素的删除功能。在应用程序 Draw 的菜单资源 IDR_DRAWTYPE 中加入一个菜单项“图形删除”。为了方便，把这个菜单项定义在“图形选中”菜单下。在 draw.rc 资源文件中加入了菜单项的定义代码：

```
IDR_DRAWTYPE MENU PRELOAD DISCARDABLE
BEGIN
    ......//省略以上代码
    POPUP "图形选中"
    BEGIN
        MENUITEM "&S 鼠标点中",          ID_SELECT_MOUSE
        MENUITEM "&U 放弃选中",          ID_SELECT_CLEAR
        MENUITEM "图形删除"              ID_SELECT_DELETE
    END
    ......//省略以下代码
END
```

在资源 ID 号定义文件 resource.h 中加入对菜单项 ID 号的定义：

```
#define ID_SELECT_DELETE    215
```

在视图类 CDrawView 中定义这个菜单项的消息处理函数。在 CDrawView 类中增加了消息处理函数的定义：

```
//{{AFX_MSG(CDrawView)
......//省略以上代码
afx_msg void OnSelectDelete();    //删除选中的图形元素
//}}AFX_MSG
```

在实现文件 drawview.cpp 中，加入消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    ……//省略以上代码
    ON_COMMAND(ID_SELECT_DELETE, OnSelectDelete)     //删除选中的图形元素
END_MESSAGE_MAP()
```

在实现文件 drawview.cpp 中，加入函数的实现代码，以下是增加了删除操作代码后的函数实现代码：

```
void CDrawView::OnSelectDelete()
{
    CDrawDoc * pDoc = GetDocument();              //得到指向当前文档对象的指针
    CClientDC ht(this);
    for(int i=0;i<pDoc->n_GraphSelect;i++)        //对所有选中的图形元素循环，进行删除操作
        Delete(&ht,pDoc->GraphSelect[i].Lb,pDoc->GraphSelect[i].Index);
    ReleaseCapture();                             //释放捕捉的光标
    pDoc->n_GraphSelect=0;                        //选中图形元素的数目为 0
}
```

OnSelectDelete 函数对所有选中的图形元素进行循环，调用 8.2.3 小节定义的 Delete 函数删除各个图形元素，并把选中的图形元素数目 n_GraphSelect 置 0。这样就完成了从文档和视图上删除图形元素的操作。

8.3　文档操作逆向化

一个成熟的系统，需要能够提供编辑逆向恢复的功能。对于矢量图形系统来说，就是能够把按照增加和删除图形元素的顺序一步步的逆向恢复各操作过程，这对于实现矢量图形系统的可操作性是至关重要的。本节，将实现矢量图形系统 Draw 的文档逆向操作。

8.3.1　逆向操作的含义

在系统操作过程中，因为经常存在着误操作或不合适操作，需要系统具有恢复功能即实现逆向操作，以提高系统的可操作性。所谓逆向操作，就是系统一步步恢复到开始操作状态。如在系统中增加一个图形元素后，进行逆向操作时，就能够删除这个图形元素，完全恢复到没有增加这个图形元素时的状态；在系统中删除了一些图形元素，进行逆向操作时，将删除的图形元素恢复。

逆向操作的程度是不一样的。有些逆向操作只对文档而不对其他操作，如对矢量图形系统实现一个逆向操作机制，能够恢复涉及文档的操作（如增加图形元素）而不能恢复其他操作（如图形放大）。而有些逆向操作是完全可逆的，如使用 Word 编辑文档时，用 Undo 操作可以恢复文档到最初状态。有些逆向操作是有一定步数的，如使用 Windows 的画图工具 PBrush 编辑图像时，只能恢复几步操作。

要对文档实现逆向操作功能，一方面需要保留删除的文档（在操作过程中，删除的文档资料一定能够被保留），另一方面，系统必须能够记录进行文档操作的全过程及每个操作过程的参数。对一个矢量图形系统实现文档的逆向操作功能时，需要记录增加和删除图形元素的过

程,并对这个操作过程的操作参数(如增加了哪个图形元素、删除了哪些图形元素)进行记录。

本节将在矢量图形系统 Draw 中讨论实现文档逆向操作功能。

8.3.2 记录文档编辑操作过程

本小节,讨论实现在矢量系统操作过程中,记录有关文档的操作步骤。

(1) 建立存储文档操作步骤的数据结构

实现逆向操作需要对历史操作文档的步骤都有记录,所以非常占用内存或硬盘空间,这也是一些系统只能进行有限步逆向操作的原因。为了在矢量图形系统 Draw 中较好的实现逆向操作,需要建立一个如下的存储机制:在内存中建立一个记录各步文档操作信息的索引,给每个文档在硬盘中建立一个外部临时文件来存储每个操作的实际参数,在索引中建立指向硬盘文件实际参数的存储位置指针。

为存储各个操作过程的索引信息,在头文件 drawdoc.h 中建立一个如下的结构:

```
typedef struct              //用来存储逆操作信息的结构
{
    unsigned char Lb;       //操作类别
    long l_Start;           //在外部临时文件中操作参数的记录位置
    int Index;              //在记录删除和增加图形元素操作时,是图形元素的数量
}UndoStruct;
```

结构中的 Lb 用来判别进行操作的类别,如可以定义为:2 - 增加文档操作,3 - 删除文档操作。对于其他的操作,如放大、移动图形操作等,都可以建立一个识别号进行记录,本节只讨论对文档操作的记录。l_Start 用来记录本次操作的操作参数信息在外部文件中的记录位置。Index 用来记录在进行删除和增加操作时涉及的图形元素的数量。

为了记录每个文档对象的操作过程,在文档类 CDrawDoc 中定义如下变量:

```
public:
    CFile Fundo;                        //用来存储逆操作信息的临时文件对象
    UndoStruct * m_UndoList;            //存储逆操作的索引信息
    int n_CurUnIndex,n_MaxUnIndex;      //当前处于操作的序号和总的操作次数
    CString FileName;                   //用来存储逆操作信息临时文件的名称
```

在实现文件 drawdoc.cpp 中,在文档类 CDrawDoc 的构造函数中对变量进行初始化:

```
CDrawDoc::CDrawDoc()
{
    ......//省略以上代码
    n_MaxUnIndex = 10000;                       //最多可以进行 10000 步的逆操作
    m_UndoList = new UndoStruct[n_MaxUnIndex];  //动态分配存储索引信息的结构数组
    n_CurUnIndex = 0;
    char p1[4];
    sprintf(p1,"%d",RunTimes);
```

```
    RunTimes++;
    //以下打开一个临时文件
    FileName="c:\\draw";
    FileName+=p1;
    FileName+=".tmp"
    Fundo.Open(FileName,CFile::modeCreate|CFile::modeReadWrite|CFile::typeBinary);
.....//省略以下代码
}
```

为了在关闭文档对象时删除动态分配的结构数组和关闭并删除临时文件，在实现文件 drawdoc.cpp 中，在文档类的析构函数中加入以下代码：

```
CDrawDoc::~CDrawDoc()
{
    .....//省略以上代码
    delete m_UndoList;
    Fundo.Close();              //关闭临时文件
    remove(FileName);           //删除临时文件
}
```

在文档类的构造函数中，给结构数组指针 m_UndoList 分配了 10000 个数组项，即可以记录 10000 步操作的索引信息。读者也可以动态管理这个结构数组来管理更多次数的操作，具体实现不再叙述。m_UndoList 数组用来存储操作过程的索引部分，操作过程的参数部分（如增加或删除的图形元素的序号等）存储在一个外部临时文件中。

文档类定义了一个 CFile 对象 Fundo 来管理外部临时文件，如何初始化这个文件对象是值得考虑的问题。如果进行的是单文档程序设计，可以在文档类的构造函数中把其初始化成固定的文件名称，在析构函数中删除这个临时文件。而现在进行的是多文档程序设计，如果还是在构造函数中初始化固定名称的外部文件，在使用多文档时，几个文档都初始化一个同名的文件就会引起运行错误。解决的办法是，定义一个全局变量，记录打开文档即运行构造函数的次数，用这个变量来构造一个各文档中互不相同的文件名。在实现文件 drawdoc.cpp 中定义一个全局变量：

```
int RunTimes=0;
```

通过这个变量的值来构造一个在各文档对象中各不相同的临时文件名 draw???.tmp。

(2) 建立存储文档编辑操作过程的函数

为了存储增加或删除图形元素的操作过程，在文档类 CDrawDoc 中定义了一个函数：

```
public:
    BOOL AddUndo(int Lb, int Numble, GraphSelectStruct * GraphList);
```

AddUndo 函数的参数 Lb 为操作的类别（例如，2－增加图形元素，3－删除图形元素），Numble 为增加或删除的图形元素的数目，结构数组 GraphList 记录了要增加或删除的图形元素的类

别和序号。在实现文件 drawdoc.cpp 中加入函数的实现代码：

```
BOOL CDrawDoc::AddUndo(int Lb, int Numble, GraphSelectStruct * GraphList)
{
    long l;
    l = m_UndoList[n_CurUnIndex].l_Start;      //得到在临时文件中的起点
    Fundo.Seek(l,0);                           //在临时文件中定位起点
    for(int i = 0;i < Numble;i++)              //对所有的图形元素循环,存储图形元素的信息
        Fundo.Write((unsigned char * )&GraphList[i],sizeof(GraphList[i]));
    m_UndoList[n_CurUnIndex].Lb = Lb;          //记录操作的类型
    m_UndoList[n_CurUnIndex].Index = Numble;   //记录图形元素的数目
    n_CurUnIndex++;                            //操作步骤加 1
    //得到存储下一个操作步骤时的外部临时文件的起点
    m_UndoList[n_CurUnIndex].l_Start = l + sizeof(GraphList[0]) * Numble;
    return TRUE;
}
```

通过 AddUndo 函数的实现代码可知，操作类别和图形元素数目保存在 m_UndoList 结构数组中，各个图形元素的类别和识别号等信息存储在临时文件对象 Fundo 中，并通过 m_UndoList 的 l_Start 来记录在临时文件中的起始存储位置。

(3) 记录文档操作过程

为了能够记录矢量图形系统 Draw 中的文档操作过程，需要在系统代码中涉及到图形元素增加或删除操作的地方加入存储操作过程的代码。以下是需要修改的代码部分。

① 在实现文件 drawview.cpp 中的鼠标消息处理函数 OnLButtonDown 中，加入对文档操作步骤的记录：

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    ......//省略以上代码
    if(m_DrawCurrent = = 1)              //如果正在绘制直线
    {
        if(PushNumb = = 0)               //如果是第一次按下左鼠标键
        {
            PushNumb++;                  //按下鼠标左键的次数加 1
            mPointOrign = point;         //直线的起点等于按中点
            mPointOld = point;           //记录本次按中点
            SetCapture();                //捕捉鼠标输入
        }
        else if(PushNumb = = 1)          //第二次按下左鼠标键(即按下直线的结束点)
        {
            //以下得到直线起终点的实际坐标
            VPtoDP(mPointOrign.x,mPointOrign.y,&xx1,&yy1);
```

```
            VPtoDP(point.x,point.y,&xx2,&yy2);
            id_only = pDoc -> GetGraphID(1);      //得到要增加直线对象的唯一识别号
            pDoc -> AddLine(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
              xx1,yy1,xx2,yy2) -> Draw(&ht,0,0,m_bColor);      //增加并重新绘制此直线
            PushNumb = 0;                          //鼠标左键按下次数为 0,重新进行直线的绘制
            //以下是记录增加直线这一操作,供逆操作时用
            GraphUndo[0].Index = pDoc -> GetGraphUpperBound(1); //得到此直线对象的序号
            GraphUndo[0].Lb = 1;                   //记录图形元素的类别是直线
            pDoc -> AddUndo(2,1,GraphUndo);        //记录一个增加图形元素的过程
            ReleaseCapture();
        }
    }
    ......//省略中间代码
    else if(m_DrawCurrent = =4||m_DrawCurrent = =5)      //如果正在绘制圆或者圆形区域
    {
        if(PushNumb = =0)                                //如果是第一次按下左鼠标键
        {
            mPointOrign = point;                         //原点等于按中点
            mPointOld = point;                           //上一个点也等于按中点
            PushNumb++;                                  //按下鼠标左键的次数加 1
            SetCapture();                                //捕捉鼠标
        }
        else                                             //如果二次以上按下鼠标左键
        {
            r = (float)sqrt(pow((float)(mPointOrign.x - mPointOld.x),2) +
                pow((float)(mPointOrign.y - mPointOld.y),2));    //计算得到圆的半径
            VPtoDP(mPointOrign.x,mPointOrign.y,&xx1,&yy1);       //得到圆心的实际坐标
            if(m_DrawCurrent = =4)                       //如果是在绘制圆
                m_Fill = 0;
            else                                         //如果是在绘制圆形区域
                m_Fill = 1;
            id_only = pDoc -> GetGraphID(3);             //得到要增加圆的唯一识别号
            pDoc -> AddCircle(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
                xx1,yy1,r,m_Fill) -> Draw(&ht,0,0,m_bColor);     //保存圆并重画圆
            PushNumb = 0;                          //鼠标按键次数为 0,重新进行圆的交互绘制操作
            //以下是记录增加圆这一操作,供逆操作时用
            GraphUndo[0].Index = pDoc -> GetGraphUpperBound(3);  //得到圆对象的序号
            GraphUndo[0].Lb = 3;                                 //图形元素的类别是圆
            pDoc -> AddUndo(2,1,GraphUndo);                      //记录一个增加图形元素的过程
            ReleaseCapture();
        }
    }
```

```
else if(m_DrawCurrent==6)                    //正在绘制圆弧
{
    if(PushNumb==0)                          //第一次按下鼠标左键
    {
        SetCapture();
        DrawArcYes=0;
        mPointOrign1=point;                  //记录圆弧第一点
        PushNumb++;
    }
    else if(PushNumb==1)                     //第二次按下鼠标左键
    {
        mPointOrign=point;                   //记录下圆弧的第二点
        PushNumb++;
    }
    else                                     //第三次按下鼠标键
    {
        jsarc(mPointOrign1,mPointOrign,point,&m_CircleX,&m_CircleY,&m_CircleR,
          &m_Angle1,&m_Angle2);              //计算圆弧的几何要素
        id_only=pDoc->GetGraphID(4);         //得到要增加圆弧的唯一识别号
        pDoc->AddArc(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
          m_CircleX,m_CircleY,m_CircleR,m_Fill,m_Angle1,m_Angle2)
          ->Draw(&ht,0,0,m_bColor);          //增加一个圆弧并重新绘制圆弧
        PushNumb=0;                          //按下鼠标左键次数为0,重新进行圆弧的绘制
        //以下是记录增加圆弧这一操作,供逆操作时用
        GraphUndo[0].Index=pDoc->GetGraphUpperBound(4);   //得到圆弧对象的序号
          GraphUndo[0].Lb=4;                              //图形元素的类别是圆弧
          pDoc->AddUndo(2,1,GraphUndo);                   //记录一个增加图形元素的过程
        ReleaseCapture();      //释放捕捉的鼠标
    }
}
  ......//省略以下代码
  CView::OnLButtonDown(nFlags, point);
}
```

② 在实现文件 drawview.cpp 中,在消息处理函数 OnRButtonDown 中加入记录文档操作步骤的代码:

```
void CDrawView::OnRButtonDown(UINT nFlags, CPoint point)
{
    ......//省略以上代码
    else if((m_DrawCurrent==2||m_DrawCurrent==3)&&PushNumb>0)
    //正在绘制连续直线或多边形区域操作,并且已经按下了一次以上鼠标左键
    {
```

```
        //擦除最后一段拖动的橡皮线
        ddd.MoveTo(mPointOrign);
        ddd.LineTo(mPointOld);
        if(m_DrawCurrent==2&&PushNumb>1) //如果正在绘制连续直线,并且顶点数大于 2
        {
            id_only=pDoc->GetGraphID(2); //得到要增加连续直线或多边形区域的唯一识别号
            //以下保存交互绘制的连续直线,并重新绘制连续直线
            pDoc->AddPLine(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,
              id_only,PushNumb,PointXyz,0)->Draw(&ddd,0,0,m_bColor);
            //以下是记录增加连续直线这一操作,供逆操作时用
            GraphUndo[0].Index=pDoc->GetGraphUpperBound(2);  //得到连续直线的序号
            GraphUndo[0].Lb=2;                    //图形元素的类别是连续直线
            pDoc->AddUndo(2,1,GraphUndo);         //记录一个增加图形元素的过程
            ReleaseCapture();
        }
        else if(m_DrawCurrent==3&&PushNumb>2) //如果正在绘制多边形区域,并且顶点数大于 2
        {
            id_only=pDoc->GetGraphID(2);      //得到要增加多边形区域的唯一识别号
            //以下保存交互绘制的多边形区域,并重新绘制多边形区域
            pDoc->AddPLine(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
              PushNumb,PointXyz,1)->Draw(&ddd,0,0,m_bColor);
            //以下是记录增加多边形区域这一操作,供逆操作时用
            GraphUndo[0].Index=pDoc->GetGraphUpperBound(2); //得到多边形区域的序号
            GraphUndo[0].Lb=2;                    //图形元素的类别是多边形区域
            pDoc->AddUndo(2,1,GraphUndo);         //记录一个增加图形元素的过程
            ReleaseCapture();
        }
        PushNumb=0;
        ReleaseCapture();                         //释放捕捉的鼠标
    }
        ......//省略以下代码
    CView::OnRButtonDown(nFlags, point);
}
```

③ 在实现文件 drawview.cpp 中,在增加标注文本的函数 DrawTextOnOk(见 6.3.4 小节)中,加入记录增加标注文本操作过程的代码:

```
void CDrawView::DrawTextOnOk()
{
    CDrawDoc* pDoc = GetDocument();          //得到指向当前文档对象的指针
    CClientDC ht(this);                      //创建一个绘图对象
    int TextLong;
    PushNumb=0;
    m_TextString=pTextDlg->m_Text;           //得到标注文本内容
```

```
        TextLong = m_TextString.GetLength();          //标注文本的字符长度
        if(TextLong > 0)                              //如果有标注文本
        {
            int id_only = pDoc -> GetGraphID(5);    //得到要增加的标注文本对象的识别号
            pDoc -> AddText(m_pColor,m_brColor,m_LineWide,m_LineType,m_Layer,id_only,
                m_TextX, m_TextY,m_TextAngle,m_FontAngle,m_FontHeight,m_FontWide,
                m_FontBetween,0,TextLong,m_TextString) -> Draw(&ht,0,0,m_bColor);
            pTextDlg -> m_Text.Empty();               //清空字符串 m_Text 的内容
            pTextDlg -> SendMessage(WM_INITDIALOG);  //激发消息调用对话框的 OnInitDialog 函数
            //以下是记录增加标注文本这一操作,供逆操作时用
            GraphUndo[0].Index = pDoc -> GetGraphUpperBound(5);  //得此标注文字的序号
            GraphUndo[0].Lb = 5;                      //图形元素类别是标注文本
            pDoc -> AddUndo(2,1,GraphUndo);           //记录一个增加图形元素的过程
            ReleaseCapture();                         //释放捕捉的鼠标
        }
    }
```

④ 在实现文件 drawview.cpp 中,在删除图形元素的函数 OnSelectDelete(见 8.2.4 小节)中,加入记录删除图形元素操作过程的代码:

```
void CDrawView::OnSelectDelete()
{
    CDrawDoc * pDoc = GetDocument();              //得到指向当前文档对象的指针
    CClientDC ht(this);
    for(int i = 0;i < pDoc -> n_GraphSelect;i++)   //对所有选中的图形元素循环,进行删除操作
        Delete(&ht,pDoc -> GraphSelect[i].Lb,pDoc -> GraphSelect[i].Index);
    //以下是记录删除图形这一操作,供逆操作时用
    pDoc -> AddUndo(3,pDoc -> n_GraphSelect,pDoc -> GraphSelect);
    ReleaseCapture();          //释放捕捉的光标
    pDoc -> n_GraphSelect = 0;//选中的图形元素为 0
}
```

在以上增加的代码中,GetGraphUpperBound 函数被调用,用来得到刚增加的图形元素在对象指针数组中的位置。因为在对象指针数组中,新增的数组项在数组的最后,所以通过 GetGraphUpperBound 函数能够得到刚增加的图形元素对象指针在记录对象指针的数组中的位置。

在以上增加的代码中,用到的成员变量 GraphUndo,在视图类 CDrawView 中定义:

```
private:
GraphSelectStruct GraphUndo[4]; //定义一个存储文档操作参数的数组
```

8.3.3　实现文档逆向操作

在 8.3.2 小节实现的记录文档操作过程基础上,可以在应用程序 Draw 中实现文档逆向操作功能。

我们首先定义一个菜单去激发完成文档逆向操作功能。在应用程序 Draw 的菜单资源

IDR _ DRAWTYPE 中，有一个“编辑”菜单，菜单中有一个“撤消”菜单项，用 ClassWizard 为这个菜单项在视图类 CDrawView 中增加一个消息处理函数。增加完毕后，在视图类 CDrawView 中定义一个消息处理函数：

```
//{{AFX_MSG(CDrawView)
.....//省略以上代码
afx_msg void OnEditUndo();
//}}AFX_MSG
```

在实现文件 drawview.cpp 中，加入了消息映射机制：

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    .....//省略以上代码
    ON_COMMAND(ID_EDIT_UNDO, OnEditUndo)
END_MESSAGE_MAP()
```

在实现文件 drawview.cpp 中，加入函数的实现代码，在函数中加入实现逆向操作的代码，得到 OnEditUndo 函数的实现代码如下：

```
void CDrawView::OnEditUndo()
{
    CDrawDoc* pDoc = GetDocument();     //得到指向当前文档对象的指针
    pDoc->OnEditUndo();                 //进行一步逆操作
    pDoc->UpdateAllViews(this);         //重画当前文档的所有视图
}
```

当按中“撤消”菜单项时，函数 CDrawView::OnEditUndo 运行，调用 CDrawDoc 类的函数 OnEditUndo 完成文档的逆向操作，并重画当前文档的所有视图。

OnEditUndo 函数具体完成文档逆向操作，在文档类 CDrawDoc 中定义：

```
public:
    void OnEditUndo()
```

在实现文件 drawdoc.cpp 中，加入函数的实现代码：

```
void CDrawDoc::OnEditUndo()
{
    int i,Numble;
    if(n_CurUnIndex<=0)                          //如果没有记录操作过程，不能进行逆向操作
        return;
    Numble=m_UndoList[n_CurUnIndex-1].Index;                    //图形元素的数目
    Fundo.Seek(m_UndoList[n_CurUnIndex-1].l_Start,0);           //定位临时文件
    if(m_UndoList[n_CurUnIndex-1].Lb==2)     //如果原来是增加图形元素操作
    {
        for(i=0;i<Numble;i++)                    //读出所有图形元素的类别和序号
            Fundo.Read((unsigned char *)&GraphSelect[i],sizeof(GraphSelect[i]));
        for(i=Numble-1;i>=0;i--)                 //对所有的图形元素逆向循环
```

```
            {
                delete GetGraph(GraphSelect[i].Lb,GraphSelect[i].Index);  //删除图形元素对象
                //以下删除记录对象指针的数组项
                deleteGraph(GraphSelect[i].Lb,GraphSelect[i].Index);
            }
        }
        if(m_UndoList[n_CurUnIndex-1].Lb==3)     //如果原来是删除图形元素操作
        {
            for(i=0;i<Numble;i++)                //对所有的图形元素循环
            {
                //以下读出一个图形元素的参数
                Fundo.Read((unsigned char *)&GraphSelect[i],sizeof(GraphSelect[i]));
                //以下去掉这个图形元素的删除标志
                GetGraph(GraphSelect[i].Lb,GraphSelect[i].Index)->Delete(0);
            }
        }
        n_CurUnIndex--;                          //文档操作步数减1,指向上一个文档操作步骤
    }
```

OnEditUndo 函数中完成增加或删除文档操作的逆向操作,主要操作步骤如下:

① 根据当前记录的文档操作索引数 m_CurUnIndex 得到最后一次文档操作的索引信息,根据索引信息定位临时文件。

② 如果原来进行的是增加图形元素操作,从临时文件中读出增加的各个图形元素的类别和序号。对这些图形元素逆向循环,得到并删除各个图形元素对象,并从存储对象指针的数组中删除相应的数组项;

③ 如果原来进行的是删除图形元素操作,从临时文件中读出删除的各个图形元素的类别和序号,得到指向各个图形元素对象的指针并以参数 FALSE 调用 Delete 函数,去掉图形元素的删除标志,恢复删除的图形元素;

至此,完成了对文档的逐步逆向操作,编译并运行应用程序 Draw,当执行"撤消"菜单项或者按中加速键 CTRL+"Z"时,系统会逆向逐步对进行过的增加或删除图形元素的操作进行恢复。

8.3.4　实现操作过程的完全逆向化

在实际矢量图形系统中,不只是文档需要逆向操作,对于别的操作(如放大图形、点选图形等)也需要逆向操作。在很多成熟的系统中,能够将操作过程完全逆向化。

有了 8.3.2 小节实现文档逆向操作的基础,可以比较容易地实现操作过程的完全逆操作化。具体的实现过程是:分析并得到系统具有的各个操作过程(如图形放大、图形移动、图形选中等)的参数,如图形放大、移动等操作都可以用视图屏幕的起点实际坐标和显示比例(m_xStart,m_yStart,blc)三个参数来描述,增加 m_UndoList 数组能够记录的操作类别(如 Lb=4,记录图形屏幕的操作),重载 AddUndo 函数来记录别的操作过程的参数,增加函数 CDrawDoc::OnEditUndo 中处理其他操作的能力。对这几方面进行修改,就可以实现矢量图形系统中操作过程的全面逆向化。读者可在掌握本节文档操作逆向化功能的基础上自己去开发实现。

第 9 章　图形的存取功能

系统的文档往往需要存盘做永久备份。对于矢量图形系统 Draw 来说，要把图形以文件的形式保存起来，并能够打开文件重新编辑图形。应用程序 Draw 的文档是基于对象的，而保存文档资料时，不可能把对象整个存储，重点在于存储对象中的成员变量，同时需要存储一些类的信息，使系统在打开保存图形的文件时可以利用类的信息和保存的成员变量重新创建对象，而不损失任何信息。

不管是在传统的还是在面向对象的程序设计中，一般都是用二进制文件来保存文档资料。在 VC++ 中使用文件流的方法很多，但在面向对象的程序设计中，最方便最有效的方法是利用串行化功能。

9.1　利用串行化存取文档

CArchive 类用于以持久的二进制形式(通常是磁盘文件)来存储一个复杂的对象网络。用户可以把对象的内容存储到存储区中，也可以从存储区中读取内容重新创建对象。这个过程称过"串行化"。

在传统的应用程序设计中，一般直接对二进制文件进行操作来完成数据资料的存取操作，如利用 CFile 文件对象、fstream 文件流等。但在面向对象的程序设计中，利用二进制文件流操作直接存储数据资料，存在着以下两个问题：一是过程比较繁琐，例如应用程序 Draw 中有大量的直线对象，可以在直线类中增加一个成员函数完成对直线类的成员变量的存储，但是在把直线的数据资料从存储对象(文件)中读取出来的时候，需要首先构造一个直线对象，然后调用直线类的读取函数读取直线的成员变量；另外一个问题是功能受限，VC++ 为了程序集成的需要，设计成了复合文档，可以把各种外来对象(如 OLE 对象的嵌入和链接)的内容存储到外部文件中并从外部文件中读取内容重新创建对象，而程序的设计者并不知道这些对象中需要存储数据的内容和格式，在这种情况下，直接对文件流进行的存取操作就无能为力了。

串行化就是为了适应在面向对象的程序设计中解决对象的存储和构造而设计的。文档串行化是通过一个 CArchive 对象来完成管理对象的数据资料的功能，而一个 CArchive 对象是在一个 CFile 对象基础上建立起来的。也就是说，串行化管理的文档(一般是磁盘文件)是一个用 CFile 对象管理的文件，不过不再利用 CFile 类的成员函数来管理这个文件，而是通过 CArchive 对象利用 CArchive 类提供的功能来对文件进行管理。串行化允许从存储区中缓冲地读取数据，或把数据写入存储区中，以一种有效无冗余的方式处理二进制对象数据。

9.1.1　CArchive 类及 CArchive 对象的创建

本小节对 CArvhive 对象的创建方法和 CArchive 类的主要操作功能进行介绍。

(1) 创建 CArchive 对象

用户在创建一个 CArchive 对象前,必须首先创建一个 CFile 对象,另外,还必须保证文件的读入/存储状态与文件的打开模式兼容。CArchive 类的构造函数如下:

```
CArchive(CFile * pFile, UINT nMode, int bBufSize = 512, void FAR * lpBuf = NULL);
```

从以上构造函数可以看出,创建一个 CArchive 对象时,必须有一个指向 CFile 对象的指针,通过这个指针指向的 CFile 对象与外部文件联系,并操作外部文件。

CArchive 类的析构函数~CArchive()完成清除缓冲区中剩余的所有数据,关闭文档,从文件中断开文档的作用。它保证所有的数据从文档转移到文件中。

(2) CArchive 类提供的主要操作

以下讨论 CArchive 类能够提供的主要操作:

① IsStoring 和 IsLoading 函数

这两个函数在 CArchive 类中定义如下:

```
BOOL IsStoring();
BOOL IsLoading();
```

这两个函数用来判断正在进行的是读取还是存储操作。当从存储区中进行读取操作时,IsStoring 函数返回 FALSE,IsLoading 函数返回 TRUE;而在进行存储操作时,IsStoring 函数返回 TRUE,IsLoading 函数返回 FALSE。

② GetFile 函数

这个函数在 CArchive 类中定义如下:

```
CFile * GetFile() const;
```

GetFile 函数返回指向当前 CArchive 对象连接的 CFile 对象的指针。在 MFC 框架程序中进行读写文档操作时,可以在 Serialize 函数中通过此函数得到指向当前 CFile 对象的指针,而直接用 CFile 类来存取文档资料。

③ "<<"和">>"操作符

重载抽取">>"和插入"<<"操作符是方便的文档编程接口,它们支持原始类型和 CObject 派生类。这两个操作符经过了重载,是进行串行化的关键。"<<"可以存储指定对象或数据原型到存储区,">>"可以从存储区读取对象或数据原型。

用这两个操作符可以对各种数据原型进行进行操作,而不必指明数据原型的类型。如对于以下定义的变量:

```
int m_X1;
float m_X2;
m_X1 = 100;
m_X2 = 200.0;
```

可以用操作符进行存储和读取操作:

```
ar << m_X1 << m_X2;
ar >> m_X1 >> m_X2;
```

除了能够对各种数据原型进行操作外,更主要的是这两个操作符能够对各个 CObject 派生的对象,可以直接用操作符进行操作,利用串行化存取对象,具体操作见下一小节中的 Serialize 函数。

9.1.2 Serialize 函数

一个 CArchive 对象不仅可以处理原始数据类型(如 int ,float ,DWORD,WORD,char 等),最重要的是,还可以处理经过串行化的 CObject 派生类对象。一个串行化的类必须有一个 Serialize 的成员函数,必须使用 DECLARE_SERIAL 和 IMPLEMENT_SERIAL 宏。

一个类,只要直接或间接是 CObject 类的派生类,都可以对其进行串行化。实现串行化的基础是在这个类中重载 CObject 类的虚函数 Serialize,此函数是在 CObject 中定义的,函数定义如下:

```
virtual void Serialize(CArchive & ar);
```

Serialize 函数有一个参数 ar,ar 指向一个 CArchive 对象,利用 ar 指针调用 CArchive 类的成员函数完成类的串行化。现在来讨论串行化一个类的步骤。

因为 Serialize 函数是在 CObject 类中定义的虚函数,所有只有直接或间接从 CObject 派生的类才能重载这个函数实现串行化。以在应用程序 Draw 中讨论的直线类 CLine 为例,讨论具体的实现步骤:

① 在 CLine 类中重载 Serialize 函数并定义一个宏:

```
class CLine:public CDraw                    //直线类
{
protected:
    DECLARE_SERIAL(CLine);                  //定义一个宏
    float m_X1,m_X2,m_Y1,m_Y2;              //直线的起点和终点
public:
    CLine(){}                               //不带任何参数的构造函数
    //以下是有初始化参数的构造函数
    CLine(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
        int id_only,BOOL Delete,float X1,float Y1,float X2,float Y2)
        :CDraw(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,Delete)
    {
        m_X1 = X1;
        m_Y1 = Y1;
        m_X2 = X2;
        m_Y2 = Y2;
    }
    //直线的绘制函数
    virtual void Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor);
```

```
    virtual void GetRect(float * minX,float * minY,float * maxX,float * maxY);
    virtual BOOL IsPoint(float x,float y,float jl);      //判断是否被点选中的函数
    virtual void Serialize(CArchive& ar);                //串行化函数
};
```

以上代码中的黑体部分,在类 CLine 中重载定义了串行化函数 Serialize 并定义了一个宏。

② 在实现文件 drawdoc.cpp 中,加入 Serialize 函数的实现代码:

```
void CLine::Serialize(CArchive& ar)
{
    if(ar.IsStoring())        //如果正在进行存储操作
    {
        ar << b_Delete << m_ColorPen << m_ColorBrush << m_LineWide << m_LineType << m_Layer;
        ar << m_id_only << m_X1 << m_Y1 << m_X2 << m_Y2;
    }
    else                      //如果正在进行读取操作
    {
        ar >> b_Delete >> m_ColorPen >> m_ColorBrush >> m_LineWide >> m_LineType >> m_Layer;
        ar >> m_id_only >> m_X1 >> m_Y1 >> m_X2 >> m_Y2;
    }
}
```

以上所示的串行化函数 Serialize,完成了保存和读取 CLine 类的所有成员变量的功能。

③ 针对宏定义在实现文件 drawdoc.cpp 中插入一个宏调用:

```
IMPLEMENT_SERIAL(CLine,CObject,1)
```

这个宏调用的第一个参数是定义串行化的类,第二个参数是这个类的基类,第三个参数为版本号。第一个参数指定的需要进行串行化的类(CLine)中必须有一个默认构造函数(即一个不带参数的构造函数),在读取文档操作时,宏使用这个构造函数来定义一个类的对象。

在一个类中实现了串行化以后,就可以对这个类的对象用 CArchive 对象进行存取管理,如可以在类 CDrawDoc 中定义一个 CLine 对象和一个 CLine 指针:

```
CLine line1(1,1,1,0,0,0,1,0,0,100,200);
CLine * line2;
```

在 CDrawDoc 类的 Serialize 函数中,可以进行如下调用:

```
void CDrawDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())          //如果正在进行存储操作
    {
        ar << &line1;
    }
    else                         //如果正在进行读取操作
    {
        ar >> line2;
```

```
    }
}
```

在存储操作时，IMPLEMENT _ SERIAL 宏在存储区域存储了创建 CLine 对象需要的类信息、版本信息，并调用 CLine 类的串行化函数 Serialize，保存 CLine 对象的成员变量到存储区域。

在读取操作时，IMPLEMENT _ SERIAL 宏利用保存的类信息创建了一个 CLine 对象，读取并识别了版本信息，并调用 CLine 类的串行化函数 Serialize，读取保存的成员变量形成一个 CLine 对象。对 line2 类指针相当于完成了如下的操作：

```
line2 = new CLine();        //通过默认构造函数定义一个 CLine 对象
line2 - > Serialize(ar);    //通过 Serialize 函数给成员变量初始化
```

所以在串行化操作中，当用 CArchive 对象存取一个对象时，会保存这个对象的类的一些信息(类信息、版本信息)，并调用类的串行化函数保存对象的成员变量。当用 CArchive 对象读取一个对象时，会读取保存的类信息，利用这些信息创建类的对象，并调用类的串行化函数读取成员变量到对象中。

这种调用可分层的进行下去，如一个类的成员变量中包含另一个类的对象，当对上层这个对象串行化操作时，又会对这个对象中的对象进行串行化操作。

9.1.3　CObArray 类的 Serialize 函数

利用串行化存取文档操作的方便和功能的强大，还在于很多 MFC 类都重载并实现了自己的串形化函数 Serialize，可以利用这些函数来完成一些存取操作。以下对 CObArray 类的 Serialize 进行简单的介绍。

CObArray 类支持 CObject 类对象指针数组，它如同一个指针数组来管理直接或间接派生的 CObject 对象指针。在应用程序 Draw 中，各种图形元素的对象指针都是由 CObArray 类定义的对象来管理。从 MFC 源代码中，可以找到 CObArray 类的 Serialize 函数：

```
void CObArray::Serialize(CArchive& ar)
{
    ASSERT_VALID(this);
    CObject::Serialize(ar);
    if (ar.IsStoring())      //如果是存储操作
    {
        ar.WriteCount(m_nSize);                  //保存存储的对象指针的数目
        for (int i = 0; i < m_nSize; i++)        //依次存储各对象
            ar << m_pData[i];
    }
    else                     //如果是读取操作
    {
        DWORD nOldSize = ar.ReadCount();         //得到存储的对象指针的数目
        SetSize(nOldSize);                       //将数组的数目设置为成对象指针的数目
        for (int i = 0; i < m_nSize; i++)        //依次读取各对象
            ar >> m_pData[i];
```

```
        }
    }
```

从以上函数可以看出，当调用一个 CObArray 对象的 Serialize 函数进行存储操作时，首先存储了数组的项数，然后循环对存储在其中的各个项目(各个对象)进行存储，进行了如下操作：

① 将对象的类信息写入文件。

② 调用对象的 Serialize 成员函数，将对象的成员变量写入文件。

而当进行读取操作时，首先读取了数组的项目数，然后将数组的项目初始化成需要的项目数，然后进行如下操作：

① 从文件读取类信息，动态创建相应类(如 CLine)的对象，并存放指向对象的类指针。

② 调用对象的 Serialize 成员函数，从文件读取对象的成员变量初始化新生成的对象。

所以利用 CObArray 类的 Serialize 函数可以实现对存储在 CObArray 对象中的对象的存储和读取。9.2 节将讨论用串行化为应用程序 Draw 增加文档的存取功能。

9.2　实现矢量图形系统的存取功能

在 9.1 节讨论了对类实现串行化的方法，本节将利用串行化给应用程序 Draw 增加文档的存取功能。

9.2.1　实现各个图形元素类的串行化

9.1.2 小节讲述了 CObject 派生类实现串行化的方法，本节将采用这个方法对应用程序 Draw 的各个图形元素类进行串行化。

(1) CDraw 类

CDraw 类是各个图形元素类的基类，其本身并不创建对象，但为了存储其中的成员变量，也在类中增加一个串行化函数。因为 CDraw 类不定义对象，所以不必在 CDraw 类中定义宏和在实现文件中插入宏。只在 CDraw 类中加入 Serialize 函数的重载定义：

```
protected:
    virtual void Serialize(CArchive& ar);
```

在实现文件 drawdoc.cpp 中，加入 Serialize 函数的实现代码：

```
void CDraw::Serialize(CArchive& ar)
{
    if(ar.IsStoring())  //如果是存储操作
    {
        ar << b_Delete << m_ColorPen << m_ColorBrush << m_LineWide << m_LineType << m_Layer;
        ar << m_id_only;
    }
    else                //如果是读取操作
    {
```

```
        ar >> b_Delete >> m_ColorPen >> m_ColorBrush >> m_LineWide >> m_LineType >> m_Layer;
        ar >> m_id_only;
    }
}
```

(2) CLine 类

CLine 对象需要用串行化进行存储和读取。为了实现 CLine 类的串行化，必须在 CLine 类中重载 Serialize 函数和加入 DECLARE _ SERIAL(CLine)宏定义：

```
class CLine:public CDraw                    //直线类
{
    protected:
        DECLARE_SERIAL(CLine);              //声明串行化
        float m_X1,m_X2,m_Y1,m_Y2;          //直线的起点和终点
    public:
        CLine(){}                           //不带任何参数的构造函数
    //以下是有初始化参数的构造函数
    CLine(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
        int id_only,BOOL Delete,float X1,float Y1,float X2,float Y2)
        :CDraw(ColorPen,ColorBrush,LineWide,LineType,Layer,id_only,Delete)
    {
        m_X1 = X1;
        m_Y1 = Y1;
        m_X2 = X2;
        m_Y2 = Y2;
    }
    virtual void Draw(CDC * pDC,int m_DrawMode,int m_DrawMode1,short BackColor);
    virtual void GetRect(float * minX,float * minY,float * maxX,float * maxY);
    virtual BOOL IsPoint(float x,float y,float jl);      //判断是否被点选中的函数
    virtual void Serialize(CArchive& ar);               //串行化函数
};
```

在实现文件 drawdoc.cpp 中，加入 Serialize 函数的实现代码并插入 IMPLEMENT _ SERIAL 宏：

```
IMPLEMENT_SERIAL(CLine,CObject,1)
void CLine::Serialize(CArchive& ar)
{
    CDraw::Serialize(ar);   //调用基类的串行化函数
    if(ar.IsStoring())      //如果是保存操作
    {
        ar << m_X1 << m_Y1 << m_X2 << m_Y2;
    }
```

```
    else                    //如果是读取操作
    {
        ar >> m_X1 >> m_Y1 >> m_X2 >> m_Y2;
    }
}
```

Serialize 函数中首先调用基类 CDraw 的 Serialize 函数存储了基类中的成员变量，然后分别存储和读取 CLine 类中的成员变量。

(3) CPline 类

与对 CLine 类进行串行化操作相同，在 CPline 类中增加重载函数 Serialize 的定义和宏定义：

```
protected:
DECLARE_SERIAL(CPline);                 //定义串行化宏

public:
    virtual void Serialize(CArchive& ar);       //串行化函数
```

在实现文件 drawdoc.cpp 中，加入 Serialize 函数的实现代码，并插入 IMPLEMENT_SERIAL 宏：

```
IMPLEMENT_SERIAL(CPline,CObject,1)
void CPline::Serialize(CArchive& ar)
{
    int i;
    CDraw::Serialize(ar);                   //调用基类的串行化函数
    if(ar.IsStoring())                      //如果是存储操作
    {
        ar<<b_Fill<<m_Numble;
        for(i=0;i<m_Numble;i++)             //存储所有的顶点坐标
            ar<<m_PointList[i].x<<m_PointList[i].y;
    }
    else                                    //如果是读取操作
    {
        ar>>b_Fill>>m_Numble;
        m_PointList=new PointStruct[m_Numble];      //动态分配存储连续直线顶点的结构数组
        for(i=0;i<m_Numble;i++)                     //读出各个顶点坐标
            ar>>m_PointList[i].x>>m_PointList[i].y;
    }
}
```

(4) CCircle 类

与对直线类 CLine 串行化操作相同，在圆类 CCircle 中增加重载函数的定义和宏定义：

```
protected:
DECLARE_SERIAL(CCircle);                    //定义串行化宏

public:
virtual void Serialize(CArchive& ar);       //串行化函数
```

在实现文件 drawdoc.cpp 中，加入 Serialize 函数的实现代码，并插入 IMPLEMENT_SERIAL 宏：

```
IMPLEMENT_SERIAL(CCirlce,CObject,1)
void CCircle::Serialize(CArchive& ar)
{
    CDraw::Serialize(ar);                   //调用基类的函数
    if(ar.IsStoring())                      //如果是存储操作
    {
        ar<<b_Fill<<m_CircleX<<m_CircleY<<m_CircleR;
    }
    else                                    //如果是读取操作
    {
        ar>>b_Fill>>m_CircleX>>m_CircleY>>m_CircleR;
    }
}
```

(5) CArc 类

与对直线类 CLine 串行化操作相同，在圆弧类 CArc 中增加重载函数的定义和宏定义：

```
protected:
DECLARE_SERIAL(CArc);                       //定义串行化宏

public:
virtual void Serialize(CArchive& ar);       //串行化函数
```

在实现文件 drawdoc.cpp 中，加入 Serialize 函数的实现代码，并插入 IMPLEMENT_SERIAL 宏：

```
IMPLEMENT_SERIAL(CArc,CObject,1)
void CArc::Serialize(CArchive& ar)
{
    CCircle::Serialize(ar);                 //调用圆类的串行化函数
    if(ar.IsStoring())                      //如果是存储操作
    {
        ar<<m_Angle1<<m_Angle2;
    }
    else                                    //如果是读取操作
    {
```

```
            ar >> m_Angle1 >> m_Angle2;
        }
    }
```

以上函数首先调用基类 CCircle 的 Serialize 函数对圆类 CCircle 中的成员变量进行了存取，然后对圆弧类 CArc 本身的两个成员变量进行存取。

(6) CText 类

与对直线类 CLine 串行化操作相同，在 CArc 类中增加重载函数的定义和宏定义：

```
protected:
DECLARE_SERIAL(CText);                    //定义串行化宏

public:
virtual void Serialize(CArchive& ar);     //串行化函数
```

在 drawdoc.cpp 中，加入 Serialize 函数的实现代码，并插入 IMPLEMENT_SERIAL 宏：

```
IMPLEMENT_SERIAL(CText,CObject,1)
void CText::Serialize(CArchive& ar)
{
    CDraw::Serialize(ar);                 //调用基类的函数
    if(ar.IsStoring())                    //如果是存储操作
    {
        ar << m_StartX << m_StartY << m_Angle1 << m_Angle2 << m_TextHeight << m_TextWide;
        ar << m_OffWide << m_TextFont << m_TextLong << c_Text;
    }
    else                                  //如果是读取操作
    {
        ar >> m_StartX >> m_StartY >> m_Angle1 >> m_Angle2 >> m_TextHeight >> m_TextWide;
        ar >> m_OffWide >> m_TextFont >> m_TextLong >> c_Text;
    }
}
```

至此，对各个图形元素类实现了串行化。9.2.3 小节将讲述在 MFC 产生的程序框架中，调用各图形类的串行化函数完成应用程序 Draw 的图形存取功能。

9.2.2　MFC 框架程序的文档串行化机制

在 VC++ 开发应用程序时，用 AppWizard 产生的 MFC 应用程序提供了实现文档序列化操作的完整框架。实现文档串行化的大部分工作都由 MFC 应用程序框架来完成，程序开发者只需重载文档类的函数 Serialize，并在函数中加入存储和读取文档资料的代码。应用程序 Draw 中，在文档类 CDrawDoc 中已经重载了串行化函数 Serialize，以下是 Serialize 原始的实现代码：

```
void CDrawDoc::Serialize(CArchive& ar)
```

```
{
    if (ar.IsStoring())          //如果是存储操作
    {
    // TODO: add storing code here
    }
    else                         //如果是读取操作
    {
    // TODO: add loading code here
    }
}
```

Serialize 函数有一个指向 CArchive 对象的引用指针 ar,这个 CArchive 对象是在 MFC 中创建的。在 3.4.2 小节讨论 MFC 应用程序的框架结构时曾经讲述过,MFC 应用程序中的标准菜单(如"新建(N)"、"打开(N)..."、"保存(S)"、"另存为(A)..."等),其 ID 号(如 ID_FILE_NEW、ID_FILE_OPEN、ID_FILE_SAVE、ID_FILE_SAVEAS)已经在 MFC 中定义并实现了消息处理函数。这些消息处理函数的实现过程中,调用了一些虚函数,在当前的文档类中重载这些虚函数就能完成相应的操作。文档类中的 Serialize 函数,就是一个被许多标准菜单的实现过程调用的虚函数,所以只需重载 Serialize 函数,就可以实现应用程序中的标准菜单,如菜单"打开..."、"保存"、"另存为..."等的操作功能。

菜单"文件"下的菜单项"打开..."、"保存"、"另存为..."运行时,调用文档类 CDrawDoc 的 Serialize 函数进行文档存取操作。如对"打开..."菜单,其 ID 号为 ID_FILE_OPEN,其命令消息处理函数(在应用类 CWinApp 中定义实现)的主要工作步骤如下:

① 显示出"打开文件"对话框。

② 用户在对话框中作出选择后,获取用户选择的文件名,创建一个将文件装入文档的 CArchive 对象,再将该对象的指针传给文档类的 Serialize 函数。

③ 通过 CArchive 对象读入文件中的内容,创建应用程序的文档。在此操作中,在 Serialize 函数中的函数 ar.IsStoring 为 FALSE。

"保存"和"另存为..."标准菜单的消息处理函数的实现过程中,也调用了 Serialize 函数(ar.IsStoring 为 TRUE)。

9.2.3 实现矢量图形系统的文档存取功能

9.2.1 小节实现了对各种图形元素类的串行化,本节将利用各个图形元素类的串行化函数,实现矢量图形系统的存取功能。

(1) CObArray 类的 Serialize 函数

在 9.1.2 小节中讨论了 CObArray 类的 Serialize 函数,这个函数首先存取数组的数目,然后通过存储在 CObArray 对象中的各个对象指针对各个对象串行化。对每个对象指针串行化时执行了如下操作:

数据写入文件时,CobArray:: Serialize 函数对存储在 CObArray 对象中的对象完成以下两个主要步骤的操作:

① 将对象的类信息写入文件。

② 调用对象的 Serialize 成员函数,将对象的成员变量写入文件。

读取文件数据时,则 CObArray:: Serialize 函数对存储在 CObArray 对象中的每个对象完成以下两步:

① 从文件读取类信息,动态创建相应类(即 CLine)的对象,在 CObArray 对象中的存放指向对象的指针。

② 调用对象的 Serialize 成员函数,从文件读取成员变量到新创建的对象中。

可见利用 CObArray 类的 Serialize 函数,能够非常容易地实现矢量图形系统 Draw 的文档存取功能。

(2) 文档存取功能的具体实现

在实现文件 drawdoc.cpp 中,对文档类 CDrawDoc 的串行化函数 Serialize 进行修改,加入存取图形文档的实际代码。修改后的函数如下:

```
void CDrawDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring()) //如果是存储操作
    {
        for(int i=1;i<=5;i++)
        //依次对直线、连续直线、圆、圆弧、标注文本等五类图形元素循环
        {
            int nn=GetGraphUpperBound(i)+1;//存储第 i 类图形元素对象指针最大数组下标加 1
            while(nn--) //对一类图形元素的所有图形元素遍历
            {
                if(GetGraph(i,nn)->IsDelete())
                //得到一个图形元素对象指针,如果这个图形元素做了删除标志
                {
                    delete GetGraph(i,nn); //删除图形元素对象
                    DeleteGraph(i,nn);//从存储图形元素对象指针的数组中删除对应数组项
                }
            }
        }
    }
    else                //如果是读取操作
    {
    }
    m_LineArray.Serialize(ar);      //对直线对象进行串行化
    m_PLineArray.Serialize(ar);     //对连续直线和多边形区域对象进行串行化
    m_CircleArray.Serialize(ar);    //对圆和圆形区域对象进行串行化
    m_ArcArray.Serialize(ar);       //对圆弧对象进行串行化
    m_TextArray.Serialize(ar);      //对标注文本对象进行串行化
    n_CurUnIndex=0;                 //文档的操作过程为 0,即不能再进行文档逆操作
```

```
}
```

CDrawDoc::Serialize 函数的主要操作步骤如下：

① 当对文档进行保存操作时(if(ar.IsStoring())),把各类图形元素中做了删除标志的图形元素对象删除(删除图形元素对象,并删除对象指针数组中存储这个图形元素对象指针的数组项)。

② 调用存储各图形元素对象指针的 CObArray 对象的 Serialize 函数,对各类图形对象进行串行化操作。因为 m_LineArray、m_PlineArray、m_CircleArray、m_ArcArray、m_TextArray 都是 CObArray 对象,所以调用 CObArray 类的 Serialize 函数来存取各类图形元素的数目并对各类图形元素进行了串行化。

③ 把文档操作的步数置 0(n_CurUnIndex = 0)。也就是说,对文档进行保存后,就不能再对以前的文档操作进行逆操作。

在 CDrawDoc::Serialize 函数中,用到了一个检查图形元素是否处于删除状态的函数 IsDelete,这个函数在图形元素基类 CDraw 中定义：

```
public:
    BOOL IsDelete();
```

IsDelete 函数的实现代码加入在实现文件 drawdoc.cpp 中：

```
BOOL CDraw::IsDelete()
{
    return b_Delete;
}
```

至此,用串行化操作实现了矢量图形系统 Draw 的文档存取功能。进行编译后运行应用程序 Draw,就可以利用"文件"菜单的菜单项"保存"和"另存为..."来保存或换名保存矢量图形到外部文件中,用"打开..."菜单项打开保存在外部文件中的矢量图形。

(3) 存储其他文档信息

在实际的矢量图形系统设计中,同时也需要对图形参数类 CGraphPara 进行串行化,存取图形的分层、颜色等等各方面的信息,另外在文档类中还要定义很多别的参数(如图幅大小等),这些参数也需要存储到文件中。篇幅所限,在此不再介绍。

9.2.4　SetModifiedFlag 函数

(1) SetModifiedFlag 函数功能

应用程序的文档被修改后,需要作出修改标志。修改标志的作用是当关闭或退出正在编辑的文档时,应用程序会给出对话框,提示是否对修改的文档进行保存,避免了信息的丢失。设置文档修改标志是由 CDocument 类的 SetModifiedFlag 函数来完成的。函数在 CDocument 类中定义如下：

```
void SetModifiedFlag(BOOL bModified = TRUE);
```

SetModifiedFlag 函数有一个参数，其默认值为 TURE。当用 TURE 参数调用 SetModifiedFlag 函数时，设置文档被修改的标志，而当用 FALSE 参数调用这个函数时，取消文档被修改的标志。

MFC 程序框架实现了根据 SetModifiedFlag 设置的标志，来实现是否调用消息框来提示保存文档的操作功能。当用 SetModifiedFlag 函数给文档设置了修改标志后，当关闭文档时系统会自动给出消息框提示是否保存文档（主框架类调用 CDocument::SaveModified 函数实现）。为了使应用程序 Draw 能够具有此项功能，需要在应用程序 Draw 中在文档修改后用 SetModifiedFlag 函数设置文档修改标志。

(2) 设置文档修改标志

在第 8 章，为实现文档的逆操作，在文档类 CDrawDoc 中实现了一个函数 AddUndo。当文档变化时，需要运行该函数记录文档的修改过程，可以在此函数中给文档设置修改标志：

```
BOOL CDrawDoc::AddUndo(int Lb, int Numble, GraphSelectStruct * GraphList)
{
    long l;
    l = m_UndoList[n_CurUnIndex].l_Start;      //得到在临时文件中的起点
    Fundo.Seek(l,0);                           //在临时文件中定位起点
    for(int i=0;i<Numble;i++)                  //对所有的图形元素循环,存储图形元素的信息
    Fundo.Write((unsigned char *)&GraphList[i],sizeof(GraphList[i]));
    m_UndoList[n_CurUnIndex].Lb = Lb;          //记录操作的类型
    m_UndoList[n_CurUnIndex].Index = Numble;   //记录图形元素的数目
    n_CurUnIndex++;                            //操作步骤加 1
    //得到存储下一个操作步骤时的外部临时文件的起点
    m_UndoList[n_CurUnIndex].l_Start = l + sizeof(GraphList[0]) * Numble;
    SetModifiedFlag(1);                        //设置文档修改标志
    return TRUE;
}
```

(3) 取消文档修改标志

当进行逆操作达到最初操作状态或者保存了文档后，又要取消文档修改标志。在文档类 CDrawDoc 中进行逆操作的函数 OnEditUndo 中加入代码，实现当文档操作步数为 0 时，取消文档修改标志：

```
void CDrawDoc::OnEditUndo()
{
    int i,Numble;
    if(n_CurUnIndex <= 0)                  //如果没有记录操作过程,不能进行逆向操作
        return;
    Numble = m_UndoList[n_CurUnIndex - 1].Index;              //图形元素的数目
    Fundo.Seek(m_UndoList[n_CurUnIndex - 1].l_Start,0);       //定位临时文件
```

```
        if(m_UndoList[n_CurUnIndex-1].Lb==2)  //如果原来是增加图形元素操作
        {
            for(i=0;i<Numble;i++)                    //读出所有图形元素的类别和序号
                Fundo.Read((unsigned char *)&GraphSelect[i],sizeof(GraphSelect[i]));
            for(i=Numble-1;i>=0;i--)                 //对所有的图形元素逆向循环
            {
                delete GetGraph(GraphSelect[i].Lb,GraphSelect[i].Index);  //删除图形元素对象
                //以下删除记录对象指针的数组项
                deleteGraph(GraphSelect[i].Lb,GraphSelect[i].Index);
            }
        }
        if(m_UndoList[n_CurUnIndex-1].Lb==3)  //如果原来是删除图形元素操作
        {
            for(i=0;i<Numble;i++)                    //对所有的图形元素循环
            {
                //以下读出一个图形元素的参数
                Fundo.Read((unsigned char *)&GraphSelect[i],sizeof(GraphSelect[i]));
                //以下去掉这个图形元素的删除标志
                GetGraph(GraphSelect[i].Lb,GraphSelect[i].Index)->Delete(0);
            }
        }
        n_CurUnIndex--;                              //文档操作步数减1,指向上一个文档操作步骤
        if(n_CurUnIndex==0)
        SetModefiedFlag(0);
    }
```

如果像8.3.4小节讨论的那样,实现的是完全逆操作而不只是文档逆操作,就要对OnEditUndo函数进行修改,在进行完一步文档逆操作后,检查数组m_UndoList中是否还有修改文档的操作,而不是只检查m_CurUnIndex是否为0。

对文档进行存储后,也需要取消文档的修改标志,所以需要在文档类DrawDoc的串行化函数Serialize中,加入代码取消文档修改标志:

```
void CDrawDoc::Serialize(CArchive& ar)
{
    ......//省略以上代码
    SetModefiedFlag(0);
}
```

本章讨论实现了利用串行化来实现矢量图形系统的文档存取功能,除此之外,还可以用其他的方法来完成系统的存取功能。在作者所著的《Visual C++高级编程技术——开发实例剖析》一书中,讨论实现了利用CFile对象实现系统的存取功能的方法和具体实现代码,请读者参考。

第 10 章　矢量图形系统的其他功能

本篇以上各章，讨论实现了矢量图形系统的一些基本功能。本章将讨论矢量图形系统其他一些功能的实现方法或思路。

10.1　图形块管理

本篇完成的矢量图形系统中只包含了一些基本图形元素，本节将对图形块的组织和使用进行讨论。图形块也可以称为图例，指的是多个图形元素的集合。AutoCAD 中的块（Block）就是图形块。图形块对于组织一个实用的矢量图形系统是必不可少的，在组成 GIS 系统的矢量图形中，图形块用来在矢量图形中组织各种图例。具有了图形块的矢量图形系统在很大程度上能够满足管理型 GIS 系统的矢量图形要求。

在一个矢量图形系统中，对图例块的管理可分为两个方面：一方面是对图例块的图形元素组成进行管理，即通过建立一个复杂的结构来管理一个图形块是由那些图形元素组成的，以及各种组成参数（如基点坐标、边界矩形坐标等）等；另外一方面是管理把图形块插入到矢量图形中所形成的图例。图形块的管理和绘制等各个方面都比较复杂，本节只对实现思路和基本结构进行讨论，读者可以在其基础上实现图形块的制作和管理功能。

10.1.1　组织管理图形块

一个图形块可以看作是一个完整的图形文档，即可以认为是一个包含了各种图形元素的图形整体。

（1）建立管理图形块的类

为了管理图形块，需要建立管理图形块的类，管理图形块的类在组成内容上与文档类 CDrawDoc 类似。以下是作者在开发图形块时所定义类的基本部分：

```
class CBlock :public CObject    //管理一个图形块的结构
{
    public:
    CBlock(int m_iD,CString fName);
    CBlock()
    {
        m_GraphNumb = 0;
        IsOpen = FALSE;
    }
    virtual CBlock();
protected:
```

```
    DWORD reverserd;
    CString fileName;                    //图形块外部存储文件的名称
    float x,y;                           //插入点的坐标
    short int m_id_only;                 //唯一的识别号
    float m_xMin,m_xMax,m_yMin,m_yMax;   //图形块的边界矩形
    int m_GraphNumb;                     //各种图形元素的数目
    CFile fblock;                        //用来将图形快存储到文件的 CFile 对象
    BOOL IsOpen;                         //图例文件是否打开的标志
    DECLARE_SERIAL(CBlock);
private:
    CObArray m_GraphArray;               //管理图形元素对象指针的对象
public:
    CLine * AddLine(CLine * pLine);
    CPline * AddPline(CPline * pPline);
    CCircle * AddCircle(CCircle * pCircle);
    CArc * AddArc(CArc * pArc);
    CText * AddText(CText * pText);
    CDraw * GetGraph(int Index);         //得到第 Index 个图形元素对象
    int GetGraphNumb();                  //得到图形块中图形元素对象的数目
    BOOL ReadFile(BOOL isRead);          //存取图形块类对象中的图形元素对象到外部文件的函数
    BOOL Clear(BOOL all);                //清除图例块的内容
    void CalRect();                      //计算图例块的边界矩形
    void SetInsertPoint(float x_Insert,float y_Insert); //设置图形块的插入点
    void Draw(CDC * pDC,float xInsert,float yInsert,float x_scale,float y_scale,
        int DrawMode,int DrawMode1);     //图形块的绘制函数
    void Draw1(CDC * pDC,int sx1,int sy1,int width,int height,int ScreenHeight,
        int DrawMode,int DrawMode1);
    void GetRect(float * m_Xmin,float * m_Ymin,float * m_Xmax,float * m_Ymax);
    BOOL isOpen();                       //检查图例块是否打开
    virtual void Serialize(CArchive& ar);
};
```

CBlock 类，是作者开发矢量图形系统时为管理图形块而定义的。在这个定义图形块中，可以管理直线、连续直线或多边形区域、圆或圆形区域、圆弧、标注文本等几种已经讨论过的图形元素，通过一个 CObArray 对象来对这几种图形元素的对象指针进行管理，其管理图形元素对象时所采用的方法和技术与用文档类直接管理图形元素对象时类似。

(2) 图形块对象的管理

通过(1)中定义的 CBlock 类可以定义图形块对象，一个 CBlock 对象用来管理一个图形块。与管理基本图形元素时类似，在文档类 CDrawDoc 中建立一个 CObArray 对象来管理各个图形块对象的指针：

```
CObArray m_BlockArray;
```

CObArray对象 m _ BlockArray 用来对各个图形块对象的指针进行管理，可以在文档类 CDrawDoc 中实现管理图形块对象指针的操作功能。

10.1.2 组织图例

将一个图形块以一定的比例插入到矢量图形中的某个位置就形成一个图例，可以通过有限的图形块来创建大量的图例。例如，在图形系统中创建一个图形块作为城市的标志，则可以通过这个图形块在图形中创建几百甚至几千个城市图例。

(1) 图例对象的创建

为了对图例进行有效的管理，需要创建一个管理图例的类。以下是一个图例类的部分定义代码：

```
class CTag: public CDraw
{
    public:
    CTag(){};
    CTag(short ColorPen,short ColorBrush,short LineWide,short LineType,short Layer,
        int id _ only,BOOL Delete,float xInsert,float yInsert,float xScale, float yScale,
        CBlock * pBlock);
    ~CTag(){};
    int idBlock;          //图形块的 ID 号
protected:
    float x _ Insert;     //插入点横坐标
    float y _ Insert;     //插入点纵坐标
    float x _ Scale;      //横向放缩比例
    float y _ Scale;      //纵向放缩比例
    CBlock * p _ Block;   //指向图形块对象的指针
    DECLARE _ SERIAL(CTag);
public:
    virtual void Draw(CDC * pDC,int DrawMode,int DrawMode1,short BackColor);
    virtual void GetRect(float * minX,float * minY,float * maxX,float * maxY);
    virtual void Serialize(CArchive& ar);
    void SetBlock(CBlock * pBlock);
};
```

与 CLine、CPline 等基本图形类类似，CTag 类也定义成图形元素基类 CDraw 的派生类，可以从 CDraw 类中重载各个虚函数，通过多态性可以使用矢量图形系统中的各个操作功能。

在 CTag 类中，需要特别理解以下几个成员变量：

p _ Block： 是一个 CBlock 类指针，指向本图例属于的图形块对象，通过这个指针来与具体的图形块对象建立联系。

x _ Insert 和 y _ Insert： 图例的插入点坐标，即把图形块插入到矢量图形中时，图形块的插入点在图形中对应的实际坐标。

idBlock：图例对应的图形块对象的识别号，即当前图例所对应的图形块的唯一识别号（CBlock 类中的成员变量 m_id_only）。通过这个成员变量保持与图形块的永远联系，因为在把存储在文件中的矢量图形重新打开时，各图形块对象指针是变化的，所以在打开图形操作时，成员变量 p_Block 指针是通过成员变量 idBlock 索引指向所属的图形块对象。

(2) 图例对象的管理

为了对图例对象进行管理，同样需要在文档类 CDrawDoc 中建立一个 CObArray 对象：

```
CObArray m_TagArray;
```

CObArray 对象 m_TagArray 用来对各个图例（CTag 类）对象的指针进行管理，为了完成管理功能，需要在文档类 CDrawDoc 中增加或修改有关的成员函数。

10.1.3　创建图形块的方法

在一个矢量图形系统中，可以采用多种方法来创建一个图形块。以下介绍创建或编辑图例块的几种方法：

(1) 选中的图形元素形成图形块

这是一个创建图形块时常用而且简单的方法，在 AutoCAD 中也可以用此方法来产生一个图形块。在矢量图形中，用鼠标选中要形成图形块的图形元素，并选择图形块的插入基点，系统对选中的图形元素进行处理，创建并初始化一个 CBlock 对象形成一个图形块。

这种方法产生图形块操作简单，在矢量图形系统中不需要额外的操作界面。

(2) 采用图形数据接口

第二种产生或编辑图形块的方法是通过图形元素数据接口，通过建立以对话框为主的交互界面，以数据的方式来增加或编辑一个图形块，用数据方式编辑图形块所包含的图形元素。

这种增加或管理图形块的方法直观性不强，但在建立有精确数据的图形块时有较大的作用。

(3) 提供单独的编辑界面

另外一种编辑图形块的方法，是通过一个单独的图形编辑界面。用这种方法编辑图形块时，要求提供一个专门的编辑界面对图形块进行编辑，在这个界面上像组织一幅矢量图形那样来编辑一个图形块，经过一定的技术处理，可以利用现有的视图屏幕来编辑图形块。

这种编辑图形块的方法，可以很容易地被延伸用来可视化的编辑表格框架等。

本节对实现矢量图形系统的图形块和图例的方法进行了介绍，并给出了图形块和图例类的定义，以及在文档类中对其进行管理的方法。

对于实现图形块和图例所需要的各种操作功能，因篇幅所限，在书中不再进行详细讨论，有兴趣的读者可参考作者提供的类，参考其他基本图形元素的实现方法自己去组织实现，也可以参考作者提供的程序源代码（见附录）。

10.2 管理各种图形参数

在矢量图形系统中,有一些成员变量是用来存储矢量图形的参数,如 CDrawDoc 类中的 CGraphPara 对象用来管理图形的颜色、图层等信息,视图类中的 ColorPen、ColorBrush 等成员变量是用来记录当前的绘制颜色、填充颜色等。在本篇实现的矢量图形系统中,这些成员变量都是通过直接赋值来管理的,不能在系统的运行中动态修改。作为实用系统,需要提供在系统运行中动态修改这些参数的功能,通过系统开发者提供的交互式操作界面(通过对话框、对话条等)对这些成员变量进行修改。

本节,将对矢量图形系统中的一些主要图形参数的管理方式和实现方法进行讨论。

10.2.1 管理颜色

在本篇实现的矢量图形系统 CDraw 中,为了节省图形元素类的字节空间,特别是为了能够对颜色进行统一的管理,对于颜色的管理是通过一个颜色列表来实现的。在头文件 drawdoc.h 中的类 CGraphPara 类中,定义了一个 long 型的数组 m_ColorList 来管理这个颜色列表,数组的一个项用来记录一个颜色值。具体讨论参见 5.1.7 小节"图形参数类"。

在矢量图形元素中,通过记录颜色列表的序号来为每个图形元素确定颜色。如,在 CDraw 类中有一个成员变量 m_ColorPen,用来记录图形元素的颜色,m_ColorPen 记录的是颜色列表的序号,m_ColorPen 等于 1 时指的是这个图形元素的颜色值是 m_ColorList[1]。

所以在矢量图形系统中,需要对颜色进行两个方面的管理:一个方面是要管理这个颜色列表,包括对颜色表的增加和修改等操作;另外一个方面是管理系统当前各颜色参数,即对记录笔色、填充色、图形屏幕底色、间隔色等颜色序号的成员变量进行设置。

对于颜色的管理功能,在数据操作上非常简单,实际上就是修改 long 型数组 m_ColorList 以及记录各种颜色的成员变量的值。操作功能主要体现在方便性和可视化上,可视化指的是,在进行颜色选择和设置操作时,是通过选择实际效果显示的颜色,而不是通过修改数据去完成。以下是作者在开发 GIS 系统时对颜色进行管理的操作功能。

(1) 管理颜色列表

创建和实现一个如图 10-1 所示的对话框界面,从上到下、从左到右的各个方格内显示的是 long 型数组 m_ColorList 中记录的各个颜色。可以用鼠标器点中不同的方格而选择不同的颜色,对话框退出时返回点中的方格在颜色列表中的序号。

可以对颜色列表中的颜色值进行可视化修改,在选择颜色列表中的某个项(用鼠标在图 10-1 中点选对应的方格)时,按"修改"按钮能够对这个颜色项进行修改。对颜色项的修改是通过调用 MFC 中的颜色对话框类 CColorDialog 来实现的,此对话框运行时的界面如图 10-2 所示,可以在对话框中选择既有的颜色,也可以在其中配制新的颜色,对话框返回选中颜色的值。以下代码完成调用颜色对话框选中颜色值的功能:

```
long m_color;                          //定义一个变量记录颜色值
CColorDialog ColorDlgBox(0,0,this);    //定义一个对颜色对话框类
```

```
if(ColorDlgBox.DoModal() = = IDOK)      //调用颜色对话框,如果按"OK"退出对话框时
    m_color = ColorDlgBox.GetColor();  //得到选中颜色的值
```

只需用返回的颜色值 m_color 来修改记录颜色列表的 m_ColorList 数组中的对应项,就可以达到改变某个矢量图形颜色列表的功能。

图 10-1　颜色列表管理对话框

图 10-2　颜色设置对话框

(2) 设置图形元素的颜色参数

在矢量图形系统 Draw 中,在视图类 CDrawView 中定义了多个成员变量来管理各个颜色参数:

```
short m_pColor;     //当前画笔颜色的序号
short m_bColor;     //当前屏幕底色的序号
short m_brColor;    //当前画刷颜色的序号
short m_bColor;     //背景颜色,即虚线中虚线部分的颜色
```

各个成员变量记录的是各种颜色在 m_ColorList 数组中的序号,在系统的实际运行过程中,不能直接去修改这些变量的值去改变颜色,而是需要提供操作界面对颜色参数进行可视化的修改。图 10-3 是作者所完成的 GIS 系统中对颜色参数进行修改的对话框界面,在"笔色"、"填充色"、"屏幕底色"等按钮后有一个以该颜色填充的矩形边框,在按下"笔色"等按钮时,调用图 10-1 的颜色列表对话框来选择对应的颜色。这样就可视化完成了对图形颜色参数的设置。

10.2.2　管理图层

本小节对图层的数据组织、图层的功能,以及对图层的管理方法进行介绍。

图 10-3　颜色设置对话框

(1) 图层的数据组织

图层是为了对矢量图形进行分类管理的有效手段。在本篇开发的矢量图形系统 Draw 中，在 5.1.7 小节创建了管理图层的数据结构，在头文件 drawdoc.h 中有图层数据结构的定义：

```
typedef struct                    //用来存储层的信息的结构
{
    char m_Name[21];              //图层的名称
    BOOL b_Display;               //图层是否显示,1-显示,0-隐藏
    short int m_ColorIndex;       //颜色号,即在这个图层中采用笔色的序号,在与 AutoCAD 接口时用
}LayerStruct;
```

在图形参数类 CGraphPara 中，定义了存储图层信息的结构指针：

```
LayerStruct * m_LayerList;
```

以上定义的结构指针 m_LayerList，为它动态分配了一个管理图层信息的结构数组：

```
m_LayerList = new LayerStruct[n_LayerNumbAll];
```

以上分配的结构数组的项数是 n_LayerNumbAll，即最多可以管理 n_LayerNumbAll 个图层，每个图层的信息由一个 LayerStruct 结构来存储。

(2) 图层的功能

在矢量图形系统中，图层用来对矢量图形元素进行分类管理。在本篇实现的矢量图形系统中，在基本图形类 CDraw 的定义中，有一个成员变量 m_Layer，用来记录这个图形元素所在图层的序号，图层的具体信息由 m_LayerList 结构数组中这个序号所对应的数组项存储。

本篇实现的矢量图形系统 CDraw 中，在各个操作功能中并没有考虑图层的影响。不过在已经建立了管理图层数据结构的基础上，可以很容易地实现与图层有关的操作。例如，要在系统中增加隐藏图层的功能，即当一个图层处于隐藏状态时，这个图层中的所有图形元素不能进行图形绘制、选中等操作，也就是说，系统只能对处于显示状态的图层中的图形元素进行图形

绘制、选中等操作。为了实现这个功能，只需对各个图形元素类中与图形绘制、选中有关的函数进行修改即可。例如，在直线类的成员函数 Draw、IsPoint 的开始处加入如下代码：

```
if(! p_GraphPara->GetDisplayStatue(m_Layer))      //如果当前图层处于隐藏状态
    return;
```

上述代码的含义是：如果当前这个图形元素所在的图层处于隐藏状态，则退出函数。参数 p_GraphPara 是指向当前文档对象中图形参数对象 m_GraphPara 的指针，函数 GetDisplayStatue 是 CGraphPara 类的成员函数（在 5.1.7 小节定义和实现），用来返回以参数为序号的图层的显示状态（1－显示，0－隐藏）。

(3) 图层的管理

对图层需要提供新建和修改等操作，也就是需要能够对 CGraphPara 类中的结构数组 m_LayerList 的内容进行管理。在实用系统中，对结构数组 m_LayerList 的管理是通过交互式操作界面来完成的。图 10-4 是作者所完成的 GIS 系统中的一个管理图层的界面，可以在这个操作界面中增加图层和修改图层。

图 10-4　管理图层对话框

本篇讨论的对图层的管理机制，即通过在图形元素中记录所属图层的序号来确定图层信息，在应用上有很大的局限性。这种管理图层的机制，要求系统只能对图层实现增加操作，而不能进行插入和删除操作。因为对图层的插入和删除操作会影响已经有的图层的序号，造成图形元素的图层属性紊乱。为了克服这个缺点，需要对图层的数据结构进行修改，增加一个成员变量来存储图层的识别号：

```
typedef struct                    //用来存储层的信息的结构
{
    short m_id_only;              //图层的识别号
    char m_Name[21];              //图层的名称
    BOOL b_Display;               //是否显示 1－显示，0－隐藏
    short int m_ColorIndex;       //颜色号，即在这个图层中采用的笔色，在与 AutoCAD 接口时用
}LayerStruct;
```

像每种类别的各个图形元素具有唯一识别号一样，每个图层具有唯一的识别号（即 m_id_only 的值各不相同）。而在图形元素对象中，成员变量 m_Layer 记录的是图层的识别号而非图层在图层列表中的序号，这样就保证了图形元素图层属性的稳定性，在插入图层、删除图层

等操作后仍然正确。对于这种管理方式,在新建图层时需要首先计算并得到一个图层对象的唯一识别号,在图形元素中使用图层信息前需要通过计算,由记录在图形元素中的图层识别号得到这个图层列表中的序号,通过这个序号去得到图层的具体信息。

10.2.3 管理线型

VC++直接能够提供的线型较少,只能使用图3-4所示的线型,即使对这些线型,也只有在线宽为一个逻辑单位时有效。

在本篇讨论实现的矢量图形系统中,在图形元素基类CDraw中定义了一个成员变量m_LineType来记录线型的序号。可以通过可视化对话框操作界面来改变变量m_LineType的值,从而改变图形元素的线型。图10-5是一个用来选择线型的操作界面,可以通过单选圆按钮来可视化的选择线型。也可以通过创建一个对话条在系统的主操作界面上选择线型。使用对话条管理线型的操作可参考作者所著的《Visual C++高级编程技术——开发实例剖析》。

图10-5 选择线型对话框

线型缺乏的矢量图形系统只能提供管理型(矢量图形只是为管理系统提供草图)用图的需要,而不能满足图纸用图的要求。要扩充线型,需要创建自身的线型制定、管理和绘制功能。

10.2.4 增加辅助绘图模式

一个完善的矢量图形系统中,需要提供一些图形绘制模式来方便图形元素的交互绘制功能。例如,在鼠标交互绘制图形时,可以设置只能绘制水平线或竖直线、捕捉栅格点等模式。要实现这些功能,一方面需要在视图类(或文档类)中定义一些变量来识别当前的绘制模式,并提供交互界面来方便改变这些模式;另外一个方面,需要在一些具体的操作功能中,实现在各种绘制模式下应采取的操作功能。例如,如果设置鼠标绘制图形元素时鼠标点只能相对前一个点水平或竖直方向移动,则需要在视图类CDrawView的消息处理函数OnMonseMove中加入如下代码:

```
if(OrthMode = = 1&&PushNumb > 0&&m_DrawCurrent > 0&&m_DrawCurrent < = 7)
{
    int x = abs(point.x - mPointOrign.x);      //横向移动的距离
    int y = abs(point.y - mPointOrign.y);      //垂直方向移动的距离
    if(x > = y)                                //如果水平移动距离较大
```

```
            point.y = mPointOrign.y;          //将当前点的纵坐标设置成与前一个点相同
        else                                  //如果垂直移动距离较大
            point.x = mPointOrign.x;          //将当前点的横坐标设置成与前一个点相同
    }
```

变量 OrthMode 用来记录绘制模式，OrthMode = 1 时表示图形系统只能进行水平或竖直线的绘制，当正在进行图形交互绘制操作并且已经按下了鼠标点(PushNumb > 0，上一个点的坐标在 mPointOrign)的情况下，调整鼠标移动点的坐标，保证鼠标只能保持水平或垂直方向移动。

对于其他的绘制模式，可参考以上功能的实现方法，通过修改相应的鼠标消息处理函数来实现。

10.3　图形的打印输出

打印输出是矢量图形系统的基本功能。VC++ 中的 MFC 应用程序为实现图形的打印和打印预显功能提供了良好的程序框架。开发者在掌握这种程序框架的基础上，可以较容易地为应用程序增加标准 Windows 打印和打印预显功能。

由 AppWizard 产生 MFC 应用程序时，只要在图 2-6 所示的对话框中选择“Printing and Print Preview”选项，则在产生的应用程序的“文件”菜单中包含以下的三个菜单项：“打印...”、“打印预显”、“打印机设置...”。“打印...”菜单实现视图打印输出的功能；“打印预显”菜单实现打印预显的功能，使得用户可以在把图形送往打印机以前，先在屏幕上观察打印效果；“打印机设置...”菜单实现选择打印机和设置打印参数的功能。这几个菜单项所对应的都是 Windows 的标准操作。本节将对 MFC 应用程序中视图打印过程的实现机理和实现步骤进行讨论。

10.3.1　视图的打印机理

在 Windows 系统中，打印机的输出机理很简单，但功能很强大。这个机理的核心还是 Windows 的映像操作。在 4.3.1 小节讨论视口和窗口的时候曾经介绍过，Windows 进行图形输出时，实际上是先把图形用当前的逻辑坐标绘制到设备无关的窗口中，然后再映像到实际设备上。当把图形映像到屏幕上时，实现了屏幕显示，而把图形映像到打印机时，就实现了图形打印。当进行图形打印时，操作过程与把图形绘制到屏幕上相同，同样用到视图类的 OnDraw(CDC * pDC)函数，因为 Windows 的图形设备接口(GDI)是与设备无关的，所以只要为打印机创建一个绘图对象(CDC 对象)，通过指向这个对象的指针作为参数调用 OnDraw 函数，就可以实现在视图的打印输出图形。

在对矢量图形系统 Draw 的打印功能不做任何改动的情况下，连接上打印机，运行应用程序 Draw 的菜单项“打印...”时，打印机打印出来了图形。改变打印机的类型(例如由针式打印机改为激光打印机)，打印出来的图形大小发生了变化(激光打印机下较小)。这是什么原因呢？原因很简单，因为在矢量图形系统 CDraw 中采用的映像方式是像素(MM_TEXT)方式，当将图形输出到打印机时，是按照像素值进行输出的，而一个像素的长度随着设备分辨率的不同而不同，所以在不同的设备上图形的大小是不一样的。如屏幕上有一条长度为 600 像素的水平线，进行打印机输出，当打印机的分辨率为 180dpi × 180dpi(分辨率指的是每英寸的像素数)，输出的直线的长度为大约为 3.3 英寸，明显的比屏幕上的长度小。如果采用分辨率为 300dpi

×300dpi 激光打印机时，那么输出的图形中直线的长度只有 2 英寸。

如果打印输出时，采用的是 Windows 中其他以固定长度为单位的映像方式，就可以在各种不同的打印机上输出相同大小的图形。如在第 2 种映像方式 MM _ LOMETRIC(单位是 0.1 毫米)下，画一条长度 1000 的直线，在屏幕上直线的长度是 100 毫米，在打印机上也是 100 毫米(任何打印机)，这就实现了设备无关性。

10.3.2 打印功能的实现方法

本小节讨论在 MFC 应用程序中实现视图打印输出功能的方法。

(1) 打印菜单消息处理函数

如上所述，MFC 应用程序的“文件”菜单中与打印操作有关的三个菜单项“打印...”、“打印预显”、“打印机设置...”，其对应的 ID 号(ID _ FILE _ PRINT、IN _ FILE _ PRINT _ PREVIEW、IF _ FILE _ PRINT _ SETUP)在 MFC 中定义了相应的消息处理函数：

ID _ FILE _ PRINT	CView::OnFilePrint
ID _ FILE _ PRINT _ PREVIEW、	CView::OnFilePrintPreview
ID _ FILE _ PRINT _ SETUP	CWinApp::OnFilePrintSetup

所以，在 MFC 应用程序中执行这三个菜单项时，就会执行 MFC 类中相对应的消息处理函数，完成标准的操作功能。

(2) 修改打印功能

在 MFC 应用程序中的打印菜单，在 MFC 中实现的消息处理函数能够完成标准的打印过程，而在实际系统中，需要对标准打印功能进行修改，实现系统专有的打印输出功能。对系统的标准打印功能进行修改，是通过虚函数和由此引发的多态性来实现的。在 3.4.1 小节，讨论了 MFC 应用程序设计的核心技术，讨论了虚函数和多态性的概念，对于打印操作功能来说，可以按照同样的思路去组织打印功能。

“打印...”菜单的消息处理函数 CView::OnFilePrint，抽象实现了一个标准的打印操作过程。在这个打印操作过程中，调用了大量的定义在 CView 类中的虚函数，只需在当前应用程序的视图类 CDrawView(CView 类的派生类)中重载这些虚函数，并在虚函数中加入当前应用程序进行打印操作时的一些特殊代码，就可以实现应用程序所固有的打印功能。

(3) 与打印操作有关的虚函数

在 CView 类中，定义了多个与打印操作有关的虚函数，这些虚函数在标准打印操作函数 CView::OnFilePrint 中被调用。以下对主要的几个虚函数及其在打印操作实现过程中的作用进行讨论。

① OnPreparePrinting 函数。在应用程序 Draw 的视图类 CDrawView 中，可以找到重载的 OnPreparePrintfing 函数的实现代码：

```
BOOL CDrawView::OnPreparePrinting(CPrintInfo * pInfo)
```

```
{
    // default preparation
    return DoPreparePrinting(pInfo);
}
```

在 OnPreparePrinting 函数中，通过参数传递来的指向当前打印对象的指针 pInfo，调用了 DoPreparePrinting 函数，DoPreparePrinting 函数能够弹出打印机对话框（图 10-6），并用 pInfo 所指向打印对象的参数设置打印对话框的初始状态，并在按“确定”按钮退出对话框时，将对话框中的设置存入到 pInfo 所指向的对象。按“确定”退出对话框时，函数返回 TURE，否则函数返回 FALSE。OnFilePrint 函数利用返回值决定是否进行下面的打印工作。

在调用 DoPreparePrinting 函数调用对话框之前，可以对 pInfo 指向打印对象的成员变量进行修改，最典型的例子是设置打印的最小和最大页号，这样在图 10-6 的对话框启动时就会在“从”和“到”后的列表框中填入开始和结束的页号。在本书讨论的矢量图形系统中，页面的大小只有在调用了打印对话框得到了设备页面大小参数后才能得到，所以在调用 OnPreparePrinting 对话框时不能确定输出页面的大小，也就不能确定分页的情况。在有些系统，如 Word，通过“文件”菜单的“页面设置...”菜单可以预先设置好了页面的大小，所以在通过 DoPreparePrinting 函数调用对话框以前，已经能够知道分页的情况，就可以通过修改 pInfo 指向的打印对象来设置最大和最小页号：

图 10-6　打印对话框

```
BOOL CDrawView::OnPreparePrinting(CPrintInfo * pInfo)
{
    // default preparation
    pInfo->SetMinPage(1);          //设置最小页号
    pInfo->SetMaxPage(10);         //设置最大页号
    return DoPreparePrinting(pInfo);
}
```

② OnBeginPrinting 函数。在实现文件 drawview.cpp 中，我们可以找到重载函数 OnBeginPrinting 的实现代码：

```
void CDrawView::OnBeginPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
```

```
{
    // TODO: add extra initialization before printing
}
```

OnBeginPrinting 函数在按中了图 10-6 的打印对话框中的“确定”按钮后，在打印进行前被调用。可以在 OnBeginPrinting 函数内进行 GDI 资源的分配工作(如设置字库等)，将一些打印视图时需要的图形对象选入当前的参数 pDC 指向的绘图对象。函数的另外一个参数 pInfo 指向记录打印信息打印对象，在进行一个矢量图形的打印操作前，在 OnBeginPrinting 函数中可以利用绘图对象(CDC)得到设备页面的大小，根据设备页面的大小对图形进行分页。在 OnBeginPrinting 函数中，参数被注释掉，要使用哪个参数，只要把参数前后的注释去掉即可。

③ OnPrepareDC 函数。此函数在进行图形每页的打印操作前被调用，可以用来设置每页的设备环境属性(如坐标系)。

④ OnPrint 函数。在 CView 类中，OnPrint 函数调用 OnDraw 函数完成每页的实际打印操作。可以在当前视图类 CDrawView 中对这个函数进行重载，得到重载后的函数实现代码如下所示：

```
void CDrawView::OnPrint(CDC * pDC, CPrintInfo * pInfo)
{
    // TODO: Add your specialized code here and/or call the base class
    CView::OnPrint(pDC, pInfo);
}
```

在 OnPrint 函数中，可以直接调用视图类 CDrawView 的 OnDraw 函数完成一页的实际绘制工作，而不用通过调用 CView 的 OnPrint 函数，来间接调用视图类的 OnDraw 函数。在这个函数中，可以加入一些在打印操作时需要的绘制任务，如给每个打印页加一个名称，加入标题、页号等信息。总之，可以在 OnPrint 函数中实现一些在 OnDraw 函数中没有的只对打印输出操作所有的一些功能。

⑤ OnEndPrinting 函数。此函数与 OnBeginPrinting 函数相对应，在打印结束后调用，可以用来释放在 OnBeginPrinting 函数定义或选入的 GDI 资源。其空的重载函数实现代码如下所示：

```
void CDrawView::OnEndPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
{
    // TODO: add cleanup after printing
}
```

(4) 实现矢量图形系统的打印功能

通过在视图类 CDrawView 中对(3)列出的几个虚函数进行重载，并对重载函数的实现代码进行修改，就可以实现矢量图形系统的打印功能。实现打印功能时，应注意考虑实现以下的功能：

① 以多种坐标映像方式输出。在矢量图形系统 Draw 中，在屏幕上显示图形时，采用的是 MM_TEXT 映像方式；而在打印机上输出图形时，为了按照固定的尺寸输出图形，应该提供按多种映像方式打印输出图形的能力，只要建立矢量图形系统中的实际坐标与映像方式下逻辑坐标的转换关系(转换的比例等)，就可以按多种映像方式打印输出图形。提供交互操作界面

对映像方式和输出比例进行选择。

② 自动分页输出。当要输出的图形区域大于打印机输出的页面时，需要能够自动进行分页，把图形通过多个页面打印输出。

③ 提供多种选择图形输出区域的方式。在实现图形的打印功能时，提供多种方式来选择要输出的图形区域。如窗口选择区域、坐标输入区域、全图输出等。

以上简单讨论了矢量图形系统中打印功能的实现技术，在掌握这些技术基础上，读者可自己开发实现矢量图形系统的打印输出功能。在作者所著的《Visual C++ 高级编程技术——开发实例剖析》一书中，详细剖析了视图的打印实现过程和设计技术，并实例实现了适用多种映像方式、任意比例、自动分页的打印输出功能，请读者参阅。

10.4　与 AutoCAD 的 DXF 文件的接口

AutoCAD 作为目前最常用的图形软件，其应用范围非常广泛。在矢量图形系统中，实现与 AutoCAD 图形文件的接口功能具有十分重要的作用。AutoCAD 提供了一个接口文件即 DXF 文件，是一种文本格式的文件。针对这个接口文件的格式，编写接口程序，实现矢量图形系统与 AutoCAD 图形文件的双向接口，对于利用 AutoCAD 既有的图形组织 GIS 系统，以及利用 AutoCAD 的输出功能输出矢量图形系统的图形等都有巨大的作用。

10.4.1　与 AutoCAD 接口的条件

要实现矢量图形系统与 AutoCAD 图形文件的接口，首先需要矢量图形系统能够处理 AutoCAD 图形系统中的图形元素。也就是说，要实现正确的接口，我们编制的矢量图形系统必须也能够处理 AutoCAD 所包含的图形元素。例如，在 AutoCAD 中，图形元素 Pline(连续直线)是非常复杂的，它可以把变宽度的线、圆弧等组合在一起，可以说 Pline 是 AutoCAD 的精华之一。而本篇所讨论的矢量图形系统 Draw，只是实现了没有宽度的连续直线这种图形元素。所以，在本篇讨论的矢量图形系统中，是无法将 AutoCAD 的 pline 图形元素完整的转变过来的，而只能把 AutoCAD 的 pline 线近似转成矢量图形系统 Draw 中没有宽度的连续直线。要达到完全转化的目的，需要扩充矢量图形系统，使它能够处理 AutoCAD 的 pline 线式的图形元素。再比如，AutoCAD 能够处理三维图形元素，而本篇讨论的矢量图形系统只能处理二维的图形元素。所以，也不能把三维的 AutoCAD 图形文件正确的转成矢量图形系统 Draw 的图形格式。

10.4.2　实现与 DXF 文件的接口功能

本小节将讨论 DXF 文件的格式，并简要讨论 DXF 文件与应用程序 Draw 的图形之间相互转换功能的实现方法。

(1) DXF 文件的格式

DXF 文件是一种文本格式的文件，它用文本格式描述了 AutoCAD 的图形文件。在这个文件中，有以下三个基本组成元素：

关键字： 在 DXF 文件中有很多(如“Pline”、“Line”、“Circle”、“Block”等)文本行,这是一些关键字,表示了以下的各文本行要描述的内容。

标识符： 在 DXF 文件中有很多包含一个数字的文本行,这些数字在不同的关键字下有不同的含义。

数值： 表示的是对应于标识符的内容,如一个坐标值,一个标注文本的内容等。

例如,在 AutoCAD 中,对于一条起点在|100.0,50.0,0.0|,终点在|200.0,100.0,0.0|,所在图层名称为“0”的直线,用 DXF 格式描述如下(“//”符后的内容是作者加入的注释):

```
LINE          //关键字,表示以下各文本行描述的是一条直线
  8           //标识符,表示下一个文本行记录的是图层名称
0             //数值,记录图层的名称
  10          //标识符,表示下一个文本行记录的直线起点的 X 坐标
100.000       //数值,记录直线起点的 X 坐标
  20          //标识符,表示下一个文本行记录的直线起点的 Y 坐标
50.000        //数值,记录直线起点的 Y 坐标
  30          //标识符,表示下一个文本行记录的直线起点的 Z 坐标
0.0           //数值,记录直线起点的 Z 坐标
  11          //标识符,表示下一个文本行记录的直线终点的 X 坐标
200.000       //数值,记录直线终点的 X 坐标
  21          //标识符,表示下一个文本行记录的直线终点的 Y 坐标
100.000       //数值,记录直线终点的 Y 坐标
  31          //标识符,表示下一个文本行记录的直线终点的 Z 坐标
0.0           //数值,记录直线终点的 Z 坐标
  0           //标识符,表示直线描述的结束
```

对于 DXF 文件中关键字、标识符的含义,读者可参考有关的 AutoCAD 手册,在此不再详述。

一个 DXF 文件,从组织上可以分为两部分:文件头和图形数据,文件头部分存储了一个图形文件的各种参数,图形数据部分存储了图形文件中的各个图形元素。

(2) 读取 DXF 文件到矢量图形系统

要实现应用程序 Draw 的图形与 DXF 文件的接口,首先需要掌握 DXF 文件的格式,然后通过文件操作来读取 DXF 文件的内容,处理得到应用程序 Draw 中的图形元素。

以下是在直线类 CLine 中定义的一个成员函数 ReadDxf,这个函数读取 DXF 文件中描述一条直线的文本行到一个直线对象的成员变量中,初始化一个直线对象的成员变量。函数的实现代码如下:

```
void CLine::ReadDxf(CStdioFile * pFile)
{
    char lss[50];
    int pb;
    do
```

```
{
    pFile - > ReadString(lss,45);          //读出一行文本
    if(strnicmp(lss," 10",3) = = 0)        //如果文本行的内容是" 10"
    {
        pFile - > ReadString(lss,45);      //读取一行文本
        m _ X1 = atof(lss);                //得到直线的起点 X 坐标
    }
    if(strnicmp(lss," 20",3) = = 0)        //如果文本行的内容是" 20"
    {
        pFile - > ReadString(lss,45);      //读取一行文本
        m _ Y1 = atof(lss);                //得到直线的起点 Y 坐标
    }
    if(strnicmp(lss," 30",3) = = 0)        //如果文本行的内容为" 30"
    {
        pFile - > ReadString(lss,45);      //读取一行文本
        //如果是三维的图形系统,在此加入起点 Z 坐标
    }

    if(strnicmp(lss," 11",3) = = 0)        //如果文本行的内容为" 11"
    {
        pFile - > ReadString(lss,45);      //读取一行文本
        m _ X2 = atof(lss);                //得到直线的终点 X 坐标
    }
    if(strnicmp(lss," 21",3) = = 0)        //如果文本行的内容为" 21"
    {
        pFile - > ReadString(lss,45);      //读取一行文本
        m _ Y2 = atof(lss);                //得到直线的终点 Y 坐标
    }
    if(strnicmp(lss," 31",3) = = 0)        //如果文本行的内容为" 31"
    {
        pFile - > ReadString(lss,45);      //读取一行文本
        //如果是三维的图形系统,在此加入终点 Z 坐标
    }
    if(strnicmp(lss," 8",3) = = 0)         //如果文本行的内容为" 8"
    {
        pFile - > ReadString(lss,45);      //读取一行文本得到图层的名称
        pb = p _ GraphPara - > GetLayerIndex(lss);    //得到这个图层的序号
        if(pb > = 0)                       //如果有这个图层
            CDraw: : m _ Layer = pb;       //设置图形元素的图层
        else                               //如果没有这个图层
            CDraw: : m _ Layer = p _ GraphPara - > AddLayer(lss,1,1); //增加一个图层
    }
}while(strnicmp(lss," 0",3)! = 0);         //读到行" 0"
```

```
    CDraw::m_LineType = 0;                          //给定直线的类型
    CDraw::m_LineWide = 1;                          //给定直线的宽度
    CDraw::b_Delete = 0;                            //设定非删除状态
    CDraw::m_ColorPen = p_GraphPara->m_LayerList[m_Layer].m_ColorIndex + 1;//根据图层定
                                                                          //笔色
}
```

ReadDxf 函数在读取 DXF 文件的主函数在 DXF 文件中读取到关键字“Line”后被调用，函数的参数是一个 CStdioFile 类指针，指向管理 DXF 文件的文件对象。在文档类 CDrawDoc 中可以增加一个函数 ReadDxf 来完成转变 DXF 文件到应用程序 Draw 图形的功能，在这个函数中，调用了 CLine 的 ReadDxf 函数来读取一条直线的描述信息到一个直线对象。CDrawDoc::ReadDxf 函数的实现代码如下所示：

```
BOOL CDrawDoc::ReadDxf(CString filename)
{
    CStaioFile Fdxf;                            //定义一个文件对象
    CString m_str;
    int sm,colorindex;
    char *lss,*ss;
    char *ss1;
    lss=(char *)malloc(50);
    ss=(char *)malloc(50);
    sm=0;
    BOOL YN=Fdxf.Open(filename,CFile::typeText|CFile::modeRead);     //打开 DXF 文件
    if(YN==0)                                   //如果打开 DXF 文件出错,给出提示信息退出函数
    {
        m_str="打开 AutoCAD 的交换文件";
        m_str+=filename;
        m_str+="出错";
        AfxMessageBox(outchar);
        return FALSE;
    }
    ......//省略操作步骤
    do
    {
        ss1=Fdxf.ReadString(ss,45);             //读出一行文本
        ......//省略操作步骤

        if(strnicmp(ss,"LINE",4)==0)            //如果是文本行的内容是"LINE"
        {
            AddLine()->ReadDxf(&Fdxf);//增加一个直线对象,并调用 Cline 类的 ReadDxf 函数
        }
```

```
        ......//省略操作步骤
    }while(ss1! = NULL);                    //直到读到 DXF 文件的结束
    free(ss);
    free(lss);
    Fdxf.Close();
    return TRUE;
}
```

以上函数的参数 filename,记录的是要进行转换的 DXF 文件的名称(可以通过 CFileDialog 对象调用文件选择对话框,来选择得到要进行转换的 DXF 文件名称)。ReadDxf 函数中以文本格式打开 DXF 文件,然后读取文件的内容转变成应用程序 Draw 的图形格式。如果 DXF 文件中存储着直线,则在读取到描述直线的文本行时,首先创建一个 CLine 对象,然后通过这个 Cline 对象调用类 CLine 的成员函数 ReadDxf,来读取一条直线的数据到 CLine 对象。对于 DXF 文件头和其他类图形元素的读取和转化操作,读者可参考以上函数和直线类 CLine 的 ReadDxf 函数去组织实现。

(3) 矢量图形写成 DXF 文件

作为(2)中读取 DXF 文件到应用程序 Draw 图形的逆过程,将应用程序 Draw 中的图形能够写成 DXF 文件格式。这个过程与读取 DXF 文件操作相比,相对比较简单,只要按照 DXF 文件的组织格式,将应用程序 Draw 图形的参数和包含的图形元素,转变成文本的格式并写到外部文件中。在此不再详细讨论具体的实现过程。

本节讨论了本篇开发的 Draw 与 AutoCAD 的 DXF 文件实现接口的功能,并给出了主要实现步骤,读者可根据本节中的讨论,去开发完成矢量图形系统 Draw 与 AutoCAD 的 DXF 文件的接口。也可参考作者提供的源代码(见附录)。

数据库管理系统

GIS 系统从本质上还包含一个数据库管理系统，用来管理 GIS 系统的性质数据。本篇将讨论利用 VC++的 ODBC（Open Database Connectivity——开放式数据库连接）接口开发数据库管理系统的技术，并实例开发一个应用于 GIS 系统的数据库管理系统，实现通用的数据库浏览视图、数据库记录编辑功能，以及通用的数据库过滤和排序功能。并讨论开发通用数据库管理系统的方法和技术。

第 11 章　开发数据库应用程序的基础知识
第 12 章　数据库应用程序开发技术
第 13 章　使用 DataView 管理数据源
第 14 章　实现数据库浏览功能
第 15 章　实现数据库编辑功能
第 16 章　通用数据库过滤功能
第 17 章　通用数据库排序功能
第 18 章　数据库管理功能通用化

第三篇

第 11 章　开发数据库应用程序的基础知识

本篇将讲述关系数据库、SQL 语言，以及 ODBC 的概念和创建 ODBC 数据源的方法。

11.1　关系数据库简介

数据库是数据的集合体，有人也把数据库看作是有存储信息能力的组织机制，通过数据库用户可以高效而方便地检索存储的信息。本节将介绍关系数据库的概念。

11.1.1　关系数据库的概念

关系数据库可以看作是由叫做“表”的逻辑单元组成的整体，数据库中的表彼此联系。关系数据库允许将数据分成更小更容易管理的逻辑单元，根据组织层次提供更简单的维护和更优化的数据库管理功能。

(1) 平面数据库及其缺点

在日常的应用中，接触最多的是平面数据库，所谓平面数据库，指的是一个独立存储数据信息的表，经常用到的 Foxbase、Foxpro 中的 .dbf 数据库，就是平面数据库。现在建立一个平面数据库 Table 来存储一些图书的基本信息，表 11-1 列出了 Table 数据库的部分记录：

表 11-1　数据库 Table 列表

图书名称	定价	简介	出版单位	联系电话	邮编
Visual C++ 高级编程技术	80		电子工业出版社	(010)68279077	100036
SQL 自学通	32		机械工业出版社	(010)68326335	100037
21 天自学教程——Visual C++	48		清华大学出版社	(010)62786544	100084
Visual C++ 6 从入门到精通	92		电子工业出版社	(010)68279077	100036
Visual C++ 图形系统开发技术基础	32		电子工业出版社	(010)68279077	100036

当然，记录一本图书的基本信息所需要的字段比以上列出的字段要多得多，作为例子只列出了以上几个字段。

用以上列出的 Table 这个平面数据库来存储数据信息时，有以下缺点：

① 数据库中记录了非常多的冗余信息。例如，对于每一本图书，都需要给它填入出版社的信息，对于整个数据库来说，就存在着大量的重复信息，既浪费空间、操作麻烦，又容易出错。

② 扩充信息麻烦。如果想扩充更多的信息(如还需要记录图书的发行号)，则需要在数据库中增加一个字段，造成数据库组成结构的膨胀，一般来说，一个平面数据库的字段越多，记录的冗余信息也越多。

③ 数据库的信息不能被别的数据库共享，如果另外一个数据库也需要使用数据库 Table

的一个或多个字段的信息，则需要在那个数据库中建立相应的字段来重复存储这些信息。

(2) 关系数据库

为了克服(1)中平面数据库管理数据时的缺点，就创建和使用了关系数据库。对于 Table 数据库记录的数据信息(见表 11-1)，可以分成两部分，分别由两个数据库 Table1 和 Table2 来存储，表 11-2 和表 11-3 是这两个数据库的部分记录列表：

表 11-2 图书基本信息数据库

图书名称	定价	简介	house_id
Visual C++ 高级编程技术	80		1
SQL 自学通	32		3
21 天自学教程——Visual C++	48		2
Visual C++ 6 从入门到精通	92		1
Visual C++ 图形系统开发技术基础	32		1

表 11-3 出版社基本信息数据库

house_id	出版社名称	联系电话	邮编
1	电子工业出版社	(010)68279077	100036
2	清华大学出版社	(010)62786544	100084
3	机械工业出版社	(010)68326335	100037

将平面数据库 Table 分成了两个平面数据库 Table1 和 Table2，并在两个数据库中建立了有且只有一个关键字字段 house_id，通过这个共同的字段能够将以上的两个数据库组合成一个数据库来使用，这就是关系数据库。

关系数据库提供了一种更有效的存储信息的方式，在关系数据库中，一个数据库是一个存储一类数据信息的整体，把这一类数据分成更小更容易管理的逻辑单元，这些逻辑单元即是各个平面数据库，在关系数据库中称为“表”或“数据表”。通过在各个数据表中设置关键字字段，可以把多个数据表组成一个整体来管理和使用。这样就可以根据需要，从多个存储数据信息的数据表中选择内容组成一个数据整体来使用，关系数据库对这个数据整体提供了与单个数据表相同的操作功能。这样，就从根本上避免了数据库存储冗余信息，实现了通过各个逻辑单元来共享数据，提供了对数据信息的更简单的维护功能和更优化的数据库管理功能。

当然，在平面数据库设计中，通过程序设计，也可以通过表 11-2 和表 11-3 所示的数据库来完成信息管理功能，但是这种功能是通过程序设计而非数据库本身的功能来实现的。

(3) 主要的关系数据库

目前有许多成熟的关系数据库系统，比较常用的有 Oracle、SQL Server、Informix、Sybase 等。

11.1.2 关系数据库的基本术语

在以上的讨论中，我们都用“数据库”这个术语来统指数据的整体。以下介绍关系数据库

中的几个常用术语：

(1) 数据库

在关系数据库中，数据库指的是存储数据信息的整体，是在物理存储区域划出的一块物理存储空间。所有的数据对象，如各种的数据表、视图、过程等都是存储在数据库中。

(2) 表

表也称为数据表，是数据库中数据存储的基本单元。一个表就是一个平面数据库。表在数据库中分配物理存储空间，一个关系数据库中可以包含多个表。为了讨论问题时含义明确，在本书中都将关系数据库的表等称为数据表。

(3) 视图

视图是一个虚拟的数据表。就用户而言，视图看起来像数据表，其行为也像数据表。视图是由一个或多个数据表由预先定义的查询式所形成的，是由一个或多个数据表中的全部或部分字段、全部或部分记录组成的一个虚拟表。视图具有数据表的外观，可以像数据表一样对其进行存取，但视图是虚拟存在的，没有自己的存储空间，对视图的存取实际上影响的是组成视图的数据表。视图与数据表的主要区别在于：数据表中的数据要消耗物理空间来组织存储，而视图并不需要物理存储空间，是一个建立在表之上的虚拟整体。

(4) 记录集合

在关系数据库中，记录集合指的是一个数据整体。是由一个或多个数据表(一般情况下，数据表之间通过关键字相互连接)的全部或部分字段、全部或部分记录组成的一个虚拟数据表。在关系数据库中，通过记录集合来组织管理存储在表中的数据。

记录集合与视图概念类似，两者的区别是：视图是关系数据库中的一个实际存在的逻辑单元(在一个关系数据库中，如同有多个数据表一样，也可以有多个视图)，而记录集合只是存在于数据库操作过程中的一个数据整体，是进行数据库操作时的数[illegible]象。

11.2　SQL 语言简介

SQL 是结构化查询语言(Structrured Query Language)的英文简称，是用于关系型数据库通信的标准语言。各种关系数据库，都可以采用 SQL 语言来完成数据库的操作功能，SQL 是用于定义、操纵和管理数据表的被广泛接受的工业标准，甚至不使用关系数据模型的数据库也支持 SQL。

因为 SQL 在 ODBC(开放式数据库连接)中起着重要作用，所以本节对 SQL 进行简单的介绍。SQL 作为一门语言，包含了大量的操作功能，本节只对与用 ODBC 开发数据库应用程序有关的几个基本操作子式进行介绍。

11.2.1 选择子式

选择子式是SQL语言中的最基本操作式，在SQL中，通过这个子式来确定要进行操作的数据对象即记录集合。因为大部分的SQL功能都是针对一定记录集合的，所以这个子式是其他SQL功能操作的基础。在VC++中开发基于ODBC的数据库应用程序时，要管理的记录集合就是通过选择子式来决定的。

假定，在一个数据库中有两个数据表Table1和Table2，Table1数据表含有book_id、book_name、price、introduction、house_id等字段，Table2数据表含有house_id、house_name、phone_number、postcode等字段。这两个数据表通过关键字字段house_id建立连接。表11-4和表11-5中列出了这两个数据表的部分记录。

表11-4 Table1数据表的内容

book_id	book_name	price	introduction	house_id
1	Visual C++高级编程技术	80		1
2	SQL自学通	32		3

表11-5 Table2数据表的内容

house_id	house_name	phone_number	postcode
1	电子工业出版社	(010)68279077	100036
2	清华大学出版社	(010)62786544	100084
3	机械工业出版社	(010)68326335	100037

下面以这两个数据表为例，介绍SQL的选择子式。

(1) 从一个数据表中选择记录集合

现在介绍用选择子式从一个数据表中选择全部或部分字段组成记录集合的操作。如果选择数据表Table1的所有字段、所有记录组成记录集合，则有以下的SQL语句：

```
Select * from Table1
```

以上SQL语句中，*表示数据表中的所有字段，Table1是数据表的名称，Select是选择子式的关键字，其后跟随的是要选择的字段名称，from也是一个关键字，其后跟随的是数据表的名称。以上SQL语句返回数据表Table1中所有的记录。

再比如，从数据表Table1中只选择两个字段book_name、price组成记录集合，则SQL语句如下所示：

```
Select book_name, price from Table1
```

Select后跟随的是选择要选择的字段的名称，各个字段间用逗号分开。这个SQL语句返回的记录集合如表11-6所示。

表 11-6

book _ name	price
Visual C++ 高级编程技术	80
SQL 自学通	32

(2) 从多个数据表中选择记录集合

利用选择子式可以从多个数据表中选择字段组成记录集合，在这个操作过程中就体现了关系数据库中数据表的连接功能。例如，选择数据表 Table1 和 Table2 中的所有字段组成记录集合，SQL 语句如下：

```
Select * from Table1,Table2
```

在 from 关键字后有两个数据表，数据表名称之间通过逗号分开。在从多个数据表中选择字段组成记录集合时，这两个数据表之间是否建立了连接关系直接影响到产生的结果。如果数据表之间通过共有关键字字段建立了连接关系，则返回的记录数是由第一个数据表决定的，以上的 SQL 语句返回如表 11-7 所示的记录集合。

表 11-7

book _ id	book _ name	price	introduction	house _ id	house _ name	phone _ number	postcode
1	Visual C++ 高级编程技术	80		1	电子工业出版社	(010)68279077	100036
2	SQL 自学通	32		3	机械工业出版社	(010)68326335	100037

从操作结果可以看出，两个数据表 Table1 和 Table2，通过共有字段 house _ id 连接在一起产生了记录集合。可以像操作一个数据表那样来操纵这个记录集合，即能够对记录集合提供的操作功能与这个记录集合来源于一个还是多个数据表无关，在操作过程中，可以把一个记录集合看作一个数据表来使用。这样，**在关系数据库中，可以由数据表来连接形成各种记录集合即虚拟的数据表，对记录集合提供各种操作功能，这就是关系数据库的功能本质所在。**

而如果这两个数据表没有建立连接关系，则以上列出的 SQL 语句返回的是第一个数据表中的所有记录分别与第二个数据表中的所有记录一一结合后得到的记录集合。如果两个数据表 Table1 和 Table2 中都不存在 house _ id 字段，则以上的 SQL 语句返回如表 11-8 所示的记录集合。

表 11-8

book _ id	book _ name	price	introduction	publishing _ name	phone _ number	postcode
1	Visual C++ 高级编程技术	80		电子工业出版社	(010)68279077	100036
1	Visual C++ 高级编程技术	80		清华大学出版社	(010)62786544	100084
1	Visual C++ 高级编程技术	80		机械工业出版社	(010)68326335	100037
2	SQL 自学通	32		电子工业出版社	(010)68279077	100036
2	SQL 自学通	32		清华大学出版社	(010)62786544	100084
2	SQL 自学通	32		机械工业出版社	(010)68326335	100037

也可以只选择多个数据表中的部分字段来得到记录集合。例如，选择数据表 Table1 中的

book _ name、price 字段和数据表 Table2 中的 house _ name、phone _ number 字段组成记录集合，则 SQL 语句为：

```
Select Table1.book _ name,Table1.price,Table2.publishing _ name,Table2.phone _ number
from Table1,Table2
```

在 Select 关键字后，字段名称前增加了数据表的名称(如 Table1.book _ name)，表示字段是属于这个数据表的，也可以在字段名称前不加数据表的名称，但在要选择的字段在多个数据表中有同名字段时，字段名称前一定要加上数据表的名称。在两个数据表建立了连接的情况下，以上 SQL 语句返回如表 11-9 所示的记录集合。

表 11-9

book _ name	price	publishing _ name	phone _ number
Visual C++ 高级编程技术	80	电子工业出版社	(010)68279077
SQL 自学通	32	机械工业出版社	(010)68326335

而如果数据表 Table1 和 Table2 之间没有建立连接关系，则返回如表 11-10 所示的记录集合。

表 11-10

book _ name	price	publishing _ name	phone _ number
Visual C++ 高级编程技术	80	电子工业出版社	(010)68279077
Visual C++ 高级编程技术	80	清华大学出版社	(010)62786544
Visual C++ 高级编程技术	80	机械工业出版社	(010)68326335
SQL 自学通	32	电子工业出版社	(010)68279077
SQL 自学通	32	清华大学出版社	(010)62786544
SQL 自学通	32	机械工业出版社	(010)68326335

11.2.2 过滤子式

上一小节中，从数据表中选择内容记录集合时，选择的是数据表中所有的记录。在 SQL 中，过滤子式被广泛应用，为选择子式提供过滤条件。本小节对过滤子式进行讨论。

(1) 过滤子式的表示

在 SQL 中，过滤子式由关键字 where 开始。过滤子式一般与选择子式相结合，为在选择记录集合操作时提供过滤条件。如对于 11.2.1 小节中定义的数据表 Table1，可以采用以下的 SQL 语句来选择得到记录集合：

```
Select * from Table1 where price > 50
```

以上代码选择数据表 Table1 的所有字段组成记录集合。在组成记录集合时，通过过滤子式“where price > 50”进行过滤，只选择“price > 50”即图书售价大于 50 元的记录。以上的 SQL 语句得到如表 11-11 所示的记录集合。

表 11-11

book _ id	book _ name	price	introduction	publishing _ id
1	Visual C++ 高级编程技术	80		1

(2) 过滤子式中的比较操作符

在过滤子式中,可以使用多种比较操作符来组织基本过滤条件式。常用的比较操作符有“>”、“<”、“=”、“<>”、“>=”、“<=”等几种,可以用这些操作符来组织基本的过滤条件式。例如,以下 SQL 语句,可以从数据表 Table1 中选择由电子工业出版社出版的图书来得到记录集合:

```
Select Table1. * from Table1,Table2 where Table2.house _ name = "电子工业出版社"
```

(3) 过滤子式中的连接符

在 SQL 中,可以通过连接符将多个过滤基本子式组合成复合过滤子式。在 SQL 中,有两个主要的连接符“AND”和“OR”,“AND”提供与操作,即通常所说的“且”,“OR”提供或操作,即是通常说的“或”。例如,从表 Table1 中选择“售价大于 50 元且由电子出版社出版的图书”,可以用如下的 SQL 语句来表示:

```
Select Table1. * from Table1,Table2
where (Table1.price > 50) AND (Table2.publishing _ name = "电子工业出版社")
```

SQL 过滤子式可以通过连接符配合括号组成多层次的过滤子式。

11.2.3　排序子式

11.2.1 小节讨论的选择子式,返回的记录集合是按照数据表中记录的顺序进行排序的。在 SQL 中,提供了排序子式来对记录集合的记录进行排序。

(1) 排序子式的表示

在 SQL 中,排序子式通过关键字“ORDER BY”引导。排序子式一般与选择子式相结合,得到按一定顺序排列的记录集合。例如,通过以下的 SQL 语句来选择 Table1 数据表的全部字段组成记录集合,记录集合按 price 字段排序:

```
Select * from Table1 ORDER BY price
```

对于包含表 11-12 所示记录的 Table1 数据表:

表 11-12

book_id	book_name	price	introduction	publishing_id
1	Visual C++高级编程技术	80		1
2	SQL 自学通	32		3
3	21 天自学教程——Visual C++	48		2
4	Visual C++图形系统开发技术基础	32		1
5	Visual C++6 从入门到精通	92		1
6	Visual C++6 开发使用手册	62		3

以上 SQL 语句能返回如表 11-13 所示的记录集合：

表 11-13

book_id	book_name	price	introduction	house_id
2	SQL 自学通	32		3
4	Visual C++图形系统开发技术基础	32		1
3	21 天自学教程——Visual C++	48		2
6	Visual C++6 开发使用手册	62		3
1	Visual C++高级编程技术	80		1
5	Visual C++6 从入门到精通	92		1

过滤子式和排序子式可以同时使用，来为选择子式提供过滤条件和排序字段。例如，以下的 SQL 语句中包含了过滤子式和排序子式：

```
Select * from Table1 where price > 50 ORDER BY price
```

对于以上的 Table1 数据表，可以得到如表 11-14 所示的记录集合：

表 11-14

book_id	book_name	price	introduction	house_id
6	Visual C++6 开发使用手册	62		3
1	Visual C++高级编程技术	80		1
5	Visual C++6 从入门到精通	92		1

(2) 排列顺序

将记录集合中的记录按一个字段排序时，有两种排序方式：升序和降序。以上讨论的操作是按升序排列记录集合的。在 SQL 中，升序用字符串“ASC”来表示，降序用字符串“DESC”来表示，为了表示排序顺序，在排序字段的后面加上表示排序顺序的字符串(“ASC”或者“DESC”)。排序字段后不加任何字符串时，采用升序这个默认的排序方式。对于数据表 Table1，以字段 price 降序排列选择得到记录集合时，SQL 语句如下：

```
Select * from Table1 ORDER BY price DESC
```

以上 SQL 语句返回如表 11-15 所示的记录集合：

表 11-15

book_id	book_name	price	introduction	house_id
5	Visual C++6 从入门到精通	92		1
1	Visual C++ 高级编程技术	80		1
6	Visual C++6 开发使用手册	62		3
3	21 天自学教程——Visual C++	48		2
2	SQL 自学通	32		3
4	Visual C++ 图形系统开发技术基础	32		1

(3) 多字段排序

在组织排序子式时，可以采用多于一个的字段作为排序字段，如从以上列出的数据表 Table1 和在 11.2.1 小节中列出的数据表 Table2 中，选择得到记录集合时，以 house_name(升序)和 price(降序)两个字段进行排序，SQL 语句如下：

```
Select Table1.book_name,Table1.price,Table2.house_name,Table2.phone_number
from Table1,Table2 ORDER BY Table2.house_name,Table1.price DESC
```

以上 SQL 语句能够得到如表 11-16 所示的记录集合：

表 11-16

book_id	book_name	price	publishing_name	phone_number
5	Visual C++6 从入门到精通	92	电子工业出版社	(010)68279077
1	Visual C++ 高级编程技术	80	电子工业出版社	(010)68279077
4	Visual C++ 图形系统开发技术基础	32	电子工业出版社	(010)68279077
6	Visual C++6 开发使用手册	62	机械工业出版社	(010)68326335
2	SQL 自学通	32	机械工业出版社	(010)68326335
3	21 天自学教程——Visual C++	48	清华大学出版社	(010)62786544

本节讨论了 SQL 中的选择子式、过滤子式、排序子式的基本组织方法。这三种子式中，选择子式是基础，过滤子式和排序子式可作为从句应用在过滤子式中。在开发基于 ODBC 的数据库应用程序时可以通过这三种子式组织记录集合、组织过滤和排序操作功能。

关于这三种子式更复杂的组织方法以及 SQL 的其他功能，可参考有关的 SQL 手册。

11.3　ODBC 及创建 ODBC 数据源的方法

11.3.1　ODBC 简介

ODBC(Open Database Connectivity)是开放式数据库连接的英文简称，它为数据库应用程序提供了一个共同的框架，通过这个共同的框架使用标准的 API 函数来存取完全不同的数据库。传统上，要存取一指定的数据库时，往往需要具体针对数据库的 API 函数，对于不同的数据库需要采用不同的函数 API，给使用数据库造成很大的不便。而 ODBC 通过使用分层结构，通过

一标准的基于 SQL 的函数 API,将应用程序同具体针对数据库的代码分离开来。也就是说,通过 ODBC 可以采用相同的方法来存取不同的数据库,例如,可以用 ODBC 使用 FoxPro、Access、SQL Server、Oracle 等不同的数据库,而使用这些数据库的方法是完全一致的。ODBC 为进行数据库应用程序设计提供了极大的方便,只要一种数据库系统支持 ODBC,我们就可以采用 ODBC 使用统一的方法来实现对这个数据库系统的数据库的存取和查询等操作,而不必知道这个数据库系统本身的存储方式和操作语言。

11.3.2 VC++ 中使用 ODBC 的方法

ODBC 的程序设计是通过一组 API 函数来实现的。在 VC++ 的 MFC 应用程序中,可以直接利用这些 API 来组织数据库应用程序的设计。直接利用 ODBC 的 API 函数进行程序的开发工作相对比较复杂和难学。为了更方便地开发数据库应用程序,在 MFC 中有封装了几个进行数据库操作的 MFC 类,在这些类中将一些 ODBC 的 API 函数封装在一起,可以完成一些具体的操作功能,应用这些 MFC 类可以更快更容易地完成数据库应用程序的设计。

这两种开发数据库应用程序的方法各有利弊,直接利用 ODBC 的 API 函数进行应用程序设计比较复杂,但可以灵活地实现各种功能;而使用 MFC 中的类进行数据库应用程序设计比较简单易学,但能够实现的功能固定在框架内,不能灵活地组织框架外的各种功能。采用哪种方法来完成数据库应用程序的设计,主要针对具体的情况,对于实现一些固定数据库结构、固定功能的数据库应用程序设计时,应用 MFC 类具有较大的优势;而对于需要对数据库进行更多的控制,需要动态的创建或维护数据库应用程序设计时,采用 ODBC 的 API 函数去组织应用程序的设计更合适。

11.3.3 创建 ODBC 数据源

在 VC++ 中利用 ODBC 进行数据库应用程序设计时,需要首先创建 ODBC 数据引擎来连接要管理的数据库,ODBC 作为一个开放的标准数据库应用接口,可以与所有支持 ODBC 的数据库系统的数据库建立联系来存取和使用这些数据库,这个联系是通过创建一个连接这些数据库的 ODBC 数据源来实现的。以下对创建常用的几种数据库系统数据源的方法进行介绍。

(1) 创建 Access 数据库的数据源

Access 是微软公司 Office 应用程序包的重要组成部分,通过它可以方便的实现对数据库的管理。Access 以 .mdb 文件来组织数据库,在数据库中可以包含多个数据表。用 Access 创建和使用数据库的操作,读者可参考有关 Access 的说明书。以下是创建 Access 的 ODBC 数据源的步骤。

① 在 Windows95(或 Windows98、WindowsNT)下运行开始→设置→控制面板,在控制面板下找到"ODBC"图标项,双击"ODBC"图标项,在进入的操作界面中选择"User DSN"标签,则进入图 11-1 所示的操作界面,"User Data Sources"列表框中列出的是已经创建的 ODBC 数据源。

② 如果要创建新的 ODBC 数据源,在图 11-1 所示的操作界面上,选择运行"Add..."按钮,则进入如图 11-2 所示的对话框界面,在这个操作界面中需要选择操作引擎,为了创建连接 Access 数据库的 ODBC 数据源,需要在列表框中选择"Microsoft Access Driver(.mdb)"选项,按"完

成"按钮,则进入图 11-3 所示的对话框。

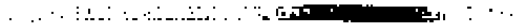


图 11-1　ODBC 数据源管理对话框

图 11-2　新建 ODBC 数据源对话框



图 11-3　Access 数据源参数设置对话框

③ 在图 11-3 所示的对话框中，对数据源的参数进行设置，具体步骤如下：在“Data Source Name”编辑框中输入要创建的数据源的名称（输入“GISDATA”），在“Dscription”编辑框中输入对数据源的描述信息，按下“Select...”按钮调出文件选择对话框选择一个 Access 数据库（.mdb 文件）作为要连接的数据源，按“OK”键退出对话框，就完成了利用已经存在的 Access 数据库创建 ODBC 数据源的操作。

在图 11-3 所示的对话框中，还有其他的按钮用来完成不同的操作：“Create...”按钮用来完成创建新的数据库作为数据源；“Repair...”按钮用来调用修复数据库的功能；“Compact...”按钮用来压缩数据库；“Advanced...”按钮用来设置数据源的高级选项（如数据库的登入号及密码等）。

在完成了利用已有的 Access 数据库创建 ODBC 数据源的操作后，再进入图 11-1 所示的对话框界面，就会在“User Data Sources”列表框中列出创建的 ODBC 数据源（“GISDATA”）。除了能够创建“User DSN”数据源外，还可以创建“File DSN”和“System DSN”数据源，操作过程与创建“User DSN”类似。

(2) 创建 Foxpro 数据库的数据源

Foxpro 是微机上最常用的小型数据库管理系统，利用 Foxpro 的数据库文件创建 ODBC 数据源的方法与创建 Access 数据源类似。不过，因为 Foxpro 组织数据库的方式与 Access、SQL Server 等是不同的，在 Foxpro 中，没有关系数据库中的数据库这个概念（在最新的 Visual Foxpro 中也引入了关系数据库的概念），每个.dbf 文件相当于关系数据库下的一个表，所以不能选择一个.dbf 文件作为数据源，而是选择一个目录模拟实现一个数据源。一个目录就相当于 Access、SQL Server 下的一个数据库，此目录下的 Foxpro 数据库文件（.dbf）就相当于关系数据库（一个目录）中的数据表。所以，建立 Foxpro 数据库 ODBC 数据源的过程与 Access 存在着一定差别。以下是操作步骤：

① 进行与创建 Access 数据源中的①相同的步骤，进入图 11-2 操作界面。

② 在图 11-2 的操作界面上选择“Microsoft Foxpro Driver(.dbf)”选项（注意 Foxpro 的版本），按“完成”键，进入图 11-4 所示的对话框。



图 11-4 Foxpro 数据源参数设置对话框

③ 在图 11-4 所示的对话框中，在“Data Source Name”编辑框中输入数据源的名称，在“Description”编辑框中输入数据源的描述信息，在默认状态下，“Use Current Directory”选项被选中，表示以使用 ODBC 数据源的应用程序当前的目录作为 Foxpro 的目录。取消对这个选项的选

择，则“Select Directory...”和“Select Indexs...”按钮处于可以执行状态，通过“Select Directory...”按钮可以调出文件选择对话框选择一个目录作为 Foxpro 数据源的目录，通过“Select Indexs...”按钮可以调出对话框给 Foxpro 数据源目录下的.dbf 文件选择索引文件。

④ 在输入了数据源的名称并对各选项进行了选择后，按“OK”按钮退出对话框，就完成了创建 Foxpro 数据库的 ODBC 数据源的操作过程。

(3) 创建 SQL Server 数据库的数据源

因为 ODBC 本身就是基于 SQL 建立起来的，所以 ODBC 管理数据源的方法与 SQL Server 是类似的。所以，利用 ODBC 管理 SQL Server 数据库的方式与在 SQL Server 系统本身的管理方式类似。在 VC++6.0 企业版的开发平台上，可以直接来创建、管理、操作 SQL Server 数据库。以下是在 VC++6.0 下，创建 SQL Server 的 ODBC 数据源的步骤：

① 保证 SQL Server 的服务应用程序处于运行状态。如果服务应用程序没有打开，打开的步骤如下：在服务器中，选择 SQL Server 菜单的“SQL Service Manager”项，进入图 11-5 所示的操作界面，选择要打开的服务程序“MSSQLServer”和“SQLExecutive”，双击“Start/Continue”按钮来打开服务程序。

图 11-5　SQL Server Manager 对话框

② 进行与创建 Access 数据源中的①相同的步骤，进入图 11-2 所示的对话框。

③ 在对话框中选择“SQL Server”选项，按“完成”键，进入图 11-6 所示的对话框。

图 11-6　SQL Server 数据库的 ODBC 数据源参数设置对话框 1

④ 在图 11-6 所示的对话框中输入各个选项,在"Name"编辑框中输入创建的 ODBC 数据源的名称,在"Description"中输入 ODBC 数据源的描述信息,在"Server"选择框中选择要连接的 SQL Server 服务器(连接当前服务器的 SQL Server 时,有"(local)"选项)。按"下一步"按钮进入到如图 11-7 所示的对话框,在这个对话框有两个选项:"With Windows NT authertication using the network login"和"With SQL Server authertication...",选中前一个选项时,在登入到 Windows NT 时就能够使用 SQL Server;对于第二个选项,与 SQL Server 连接时需要另外的登入操作,在"Login"对话框中输入登入 SQL Server 时的帐号,在 Password 编辑框中输入密码。按"Next"按钮,进入到图 11-8 所示的对话框。

图 11-7 SQL Server 数据库的 ODBC 数据源参数设置对话框 2

图 11-8 SQL Server 数据库的 ODBC 数据源参数设置对话框 3

⑤ 在图 11-8 所示的对话框中,主要对所要连接的数据库进行选择,选中"Change the Default Databases to"激活跟随其后的列表框,从列表框中选择需要连接的数据库,保持其他的选择不变,按"下一步"键进入图 11-9 所示的对话框。

⑥ 保持图 11-9 所示的对话框中的选择不变,按"下一步"键,进入图 11-10 所示的对话框。在这个对话框中可以设置两个日志文件的名称,可以保持原来的参数不变,按"完成"按钮,就完成 ODBC 数据源的创建操作,进入图 11-11 所示的对话框。

⑦ 在图 10-11 所示的对话框中列出了要建立的 ODBC 数据源连接的各个选项,按"Test Da-

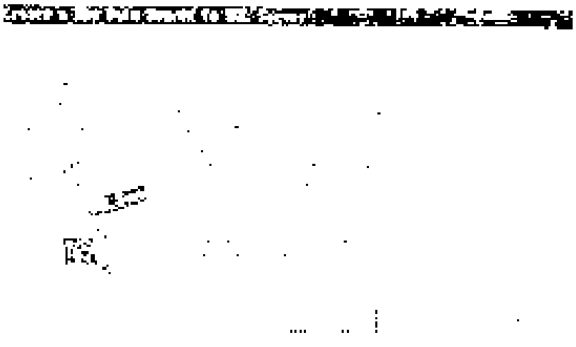


图 11-9　SQL Server 数据库的 ODBC 数据源参数设置对话框 4

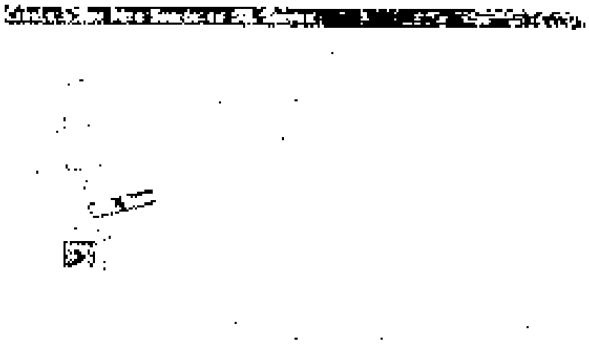


图 11-10　SQL Server 数据库的 ODBC 数据源参数设置对话框 5

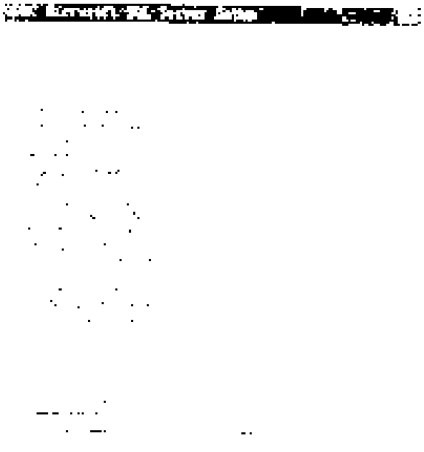


图 11-11　SQL Server 数据库的 ODBC 数据源参数对话框

ta Source"按钮，可以对连接是否成功进行测试，检查以上参数设置的数据源连接是否成功。如

果测试连接成功，按“OK”按钮就完成创建一个 SQL Server 数据库的 ODBC 数据源的过程。

对于 SQL Server 数据库的数据源，其工作方式是与其他方式的数据源（如 Foxpro、Access 等 ODBC 数据源）不一样的，Foxpro、Access 下的 ODBC 数据源只是建立了与数据库文件的联系，对数据库文件的操作是通过 ODBC 实现的，与这些数据库系统没有关系（在计算机上是否安装了相应的数据库系统不影响利用 ODBC 对数据库文件的操作）；而在 SQL Server 下，是通过 ODBC 建立了与 SQL Server 的服务程序的连接，数据库应用程序利用 ODBC 数据源对 SQL Server 数据库的操作是通过 ODBC 调用 SQL Server 的服务程序完成的。

第 12 章　数据库应用程序开发技术

在 VC++ 中，为利用 ODBC 进行数据库应用程序的开发，在 MFC 中实现了几个与数据库应用程序开发有关的类。这些类通过对大量的 ODBC API 函数的封装而成，可以非常方便地完成一些数据库管理功能。同时，MFC AppWizard 可以用来产生多个针对开发数据库应用程序的程序框架，在这些程序框架基础上，可以非常方便地完成一般数据库应用程序的设计。本节，将对 MFC AppWizard 创建数据库应用程序的过程进行介绍，并详细分析利用 AppWizard 产生的数据库应用程序的框架，达到从界面和功能上灵活组织数据库应用程序设计的目的。在第二篇讨论的矢量图形系统的基础上，扩展数据库管理功能，并最终实现一个基本的 GIS 系统。

12.1　创建基于 ODBC 的数据库应用程序

在 VC++ 中，为利用 ODBC 进行数据库应用程序开发准备了良好的程序框架，程序设计者可以在这个程序框架的基础上去组织数据库应用程序的开发工作。本节介绍产生一个基于 ODBC 的数据库应用程序的步骤。

12.1.1　创建一个 ODBC 数据源

在用 MFC AppWizard 产生一个数据库应用程序之前，需要首先创建一个 ODBC 数据源，即为创建的数据库应用程序组织要管理的数据库，并建立针对这个数据库的 ODBC 数据源。

假定在硬盘的某个目录下有一个 Access 的数据库 Info.mdb，在数据库中有一个数据表 city，这个数据表的结构如表 12-1 所示：

表 12-1

字段名称	数据类型	含义
only_id	int	识别号
city_name	char[20]	城市名称
population	value	人口数量
area	value	面积
industry	value	工农业产值
translate	char[20]	交通状况
environment	char[20]	环境状况

采用 11.3.2 小节中介绍的创建 Access 数据库 ODBC 数据源的步骤，建立一个针对 Info.mdb数据库的数据源 GISDATA，将利用这个 ODBC 数据源为数据库应用程序提供要管理的数据对象。

12.1.2 MFC AppWizard 创建一个数据库应用程序的步骤

在创建了 ODBC 数据源 GISDATA 后，就可以利用 MFC AppWiard 来创建一个基于 ODBC 的数据库应用程序，具体步骤如下：

① 在 VC++ 的开发环境中，选择运行“File”菜单下的“New...”菜单项，进入到“New”对话框，在对话框中选择“Project”标签，在进入图 2-2 所示的对话框界面。在对话框中选择“MFC AppWizard(exe)”选项，在对话框的“Project Name:”编辑框中输入项目的名称 Data1，保持默认的选项“Create new Workspace”不变，在“Platforms”列表框中，保持选中“Win32”选项。按“OK”按钮，进入图 2-3 所示的对话框。

② 在图 2-3 所示的对话框中，需要对文档类型进行选择，各种文档类型的选择可参考 2.1.2节的②，选择单文档类型“Single Document”，并保持对“Document/View architecture Support”选项的选择。按“Next”按钮，进入图 2-4 所示的对话框。

③ 本步骤是产生数据库应用程序的关键。在图 2-4 所示的对话框中有 4 个选项：“None”，“Header files only”，“Database View without file support”，“Database View with file Support”。当选择“None”时，不支持数据库应用程序设计；选择“header files only”选项时，产生的应用程序与选择“None”时类似，只是在产生的程序框架中的头文件 stdafx.h 和资源文件中包含了进行数据库程序设计所必须的头文件；选择“Database View without file support”选项时，将产生一个浏览数据库视图的应用程序，但应用程序中不包含对文件的支持(因为不包含对文件的支持，所有具有这个选项的应用程序只能支持单文档结构(Single Document))；选择“Database View with file Support”选项时，将产生一个包含数据库浏览视图的应用程序，应用程序能够支持对文件的操作。当选择后两个选项时，需要对要连接的 ODBC 数据源进行选择。

当前操作中，选择“Database View without file Support”选项，当选择这个选项时，“Data Source...”按钮处于可执行状态，按下“Data Source...”按钮，进入图 12-1 对话框。

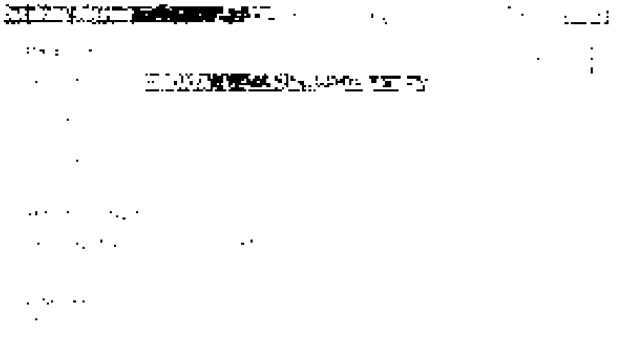


图 12-1 选择数据源对话框

④ 在图 12-1 对话框中，需要对产生的数据库应用程序连接的数据源和使用数据源的方式进行选择。在开发基于 ODBC 的数据库应用程序时，需要在“DataSource”框中选择方式“ODBC”(按中“ODBC”前的圆按钮)，从“ODBC”方式的列表框中选择在 12.1.1 小节中创建的 ODBC 数

据源 GISDATA。保持“RecordSet Type”框中的“Snapshot”(快照)选项不变,按“OK”按钮,系统会调出一个如图 12-2 所示的对话框 Select Database Tables,来选择数据源中包含的数据表或数据视图。



图 12-2　选择数据表对话框

⑤ 在图 12-2 所示的对话框中,需要选择一个或多个数据表,在产生的数据库应用程序中,将利用选择的数据表组成应用程序要管理的记录集合,列表框中列出了 ODBC 数据源 GISDATA 连接的数据库中的各个数据表,可以对一个或多个数据表进行选择(用鼠标可以选择或放弃点中的数据表)。选择多个数据表时注意,这多个数据表应该是通过主键建立了联系的,选择两个没有建立联系的数据表对于实际操作没有太大意义。为方便操作起见,在列表框中只选择数据表 city,按“OK”按钮,就会进入到如图 2-4 所示的对话框 。

⑥ 对应用程序需要支持的 OLE 操作,保持“None”选择不变,表示应用程序不支持 OLE 操作。必须注意,如果选择了对 OLE 操作的支持(“None”选项外的其他任何选择),所产生的应用程序必须支持文件操作(在图 2-4 对话框中选择的“Database View without file Support”选项,会转变成“Database View with file Support”选项)。按下“Next”按钮,就会进入图 2-5 所示的对话框,此后完成 2.1.2 节中的步骤④~⑥,就会产生一个基于 ODBC 的数据库应用程序 Data1。

本章的以下各节将以 Data1 应用程序的记录集合类 CData1Set 和视图类 CData1View 为基础,讨论利用 ODBC 进行数据库应用程序设计的主要技术。

12.2　CRecordSet 类功能分析

在 12.1.2 小节用 MFC AppWizard 产生的数据库应用程序 Data1 中,包含一个记录集合类 CData1Set,这个类由 MFC 的 CRecordSet 类派生,用来完成通过 ODBC 对数据源中的数据表的管理功能。CReocordSet 类由大量的 ODBC API 函数封装而成,实现了对记录集合的各种操作功能,以及记录集合与数据源之间的数据交换。CRecordSet 类可以看作是进行基于 ODBC 数据

库应用程序设计的核心。

在应用程序 Data1 的文档类 CData1Doc 中,定义了一个 CData1Set 类的对象:

```
public:
    CData1Set m_data1Set;
```

在应用程序 Data1 的视图类 CData1View 中,定义了一个 CData1Set 类指针:

```
public:
    CData1Set * m_pSet;
```

这个类指针被初始化指向文档对象中的 CData1Set 对象 m_data1Set,在视图类 CData1View 的函数 OnInitialUpdate 中,可以找到如下的代码:

```
void CData1View::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_data1Set;
    ......//省略以下代码
}
```

可见,视图对象中的 CData1Set 类指针 m_pSet,指向了当前文档对象中的 CData1Set 对象 m_data1Set。

本节将以 CData1Set 对象 m_data1Set 为例,讨论 CRecordSet 类所能完成的各个操作功能。

12.2.1 连接数据源和创建记录集合

当 CRecordSet 类的派生类对象利用 CRecordSet 类的成员函数完成数据库的各种管理功能之前,首先应该与要管理的数据源建立连接,并从数据源中选择数据表的内容组成一个记录集合。

(1) 连接数据源

所谓连接数据源,就是通过 ODBC 与实际的数据库文件建立联系。在 VC++ 开发数据库应用程序时,一般利用别的数据库系统的数据库文件来组织存储数据,例如,可以利用 Access 的数据库文件(.mdb),也可以采用 Foxpro 的 .dbf 数据库文件来存储数据库应用程序需要使用的数据库。为了能够使用别的数据库系统的数据库,首先应该与这些数据库建立连接,与数据库的连接,是通过 ODBC 来实现的。通过 ODBC 接口,程序开发者可以采用同样的方法使用来自不同数据源的数据库,开发针对不同数据源的数据库应用程序。利用 ODBC 与实际数据源建立联系的方法就是创建 ODBC 数据源,在第 11 章 11.3.3 小节介绍了 ODBC 数据源的创建方法。CRecordSet 派生类对象就是通过建立的 ODBC 数据源与实际的数据库建立联系。

在 CRecordSet 类的派生类对象中,调用 CRecordSet 类的 Open 函数打开记录集合对象,如下代码所示:

```
m_pSet->Open();
```

m_pSet 是指向 CRecordSet 派生类对象的指针。在 CRecordSet 类的 Open 函数中,完成了连

接数据源的工作，在 Open 函数的操作过程中，调用了定义在 CRecordSet 类中的虚函数 GetDefaultConnect，得到要连接的数据源名称和其他参数，所以，在 CRecordSet 类的派生类中重载这个函数并加入相应的代码，就可以达到设置要连接的 ODBC 数据源及其连接参数的目的。在应用程序 Data1 的记录集合类 CData1Set 中，可以找到这个虚函数重载的实现代码：

```
CString CData1Set::GetDefaultConnect()
{
    return _T("ODBC;DSN = GISDATA");
}
```

以上代码中，返回了一个字符串"ODBC;DSN = GISDATA"，这个字符串表示要连接的 ODBC 数据源，"ODBC"表示采用的是 ODBC 数据源，"DSN = GISDATA"表示采用的是 DSN 数据源（在 12.1.1 中创建的就是 DSN 数据源），数据源的名称为 GISDATA。对这个字符串进行修改，就可以将记录集合对象连接到别的数据源。

在 GetDefaultConnect 函数中，除了可以用字符串返回这两个参数外，还可以返回其他的参数。在连接一些数据库的数据源时，需要登入到数据库中去，例如，在用 ODBC 连接 SQL Server 数据库时，就需要在建立连接前，弹出登入对话框，输入帐号和密码首先登入到 SQL Server 中。可以在 GetDefaultConnect 函数中要返回的字符串中加入其他的内容，作为注册时的帐号和密码。例如，可以将 GetDefaultConnect 函数修改如下：

```
CString CData1Set::GetDefaultConnect()
{
    return _T("ODBC;DSN = access;UID = sa;PWD = GIS");
}
```

以上代码表示登入的帐号为 sa，密码为 GIS，这样在连接数据源时，就可以直接登入到数据源的数据库系统中。否则，当 GetDefaultConnect 函数中没有设置帐号和密码时，在数据库应用程序运行时，在连接数据源时，会弹出一个登入对话框，输入帐号和密码登入到数据库系统。

(2) 创建记录集合

记录集合指的是记录集合类能够管理的数据整体。创建记录集合就是完成第 11 章 11.2.1小节所完成的选择子式的操作，即从 ODBC 数据源所连接的数据库中，从一个或多个数据表内选择全部或部分字段组成一个记录集合，在创建记录集合过程中，可以通过过滤子式和排序子式得到经过过滤和排序后的记录集合。**CRecordSet 派生类对象就是对这个创建的记录集合进行管理，并能够将对记录集合的修改通过 ODBC 影响到实际的数据源，这就是利用 ODBC 进行数据库应用程序设计的核心。**CRecordSet 类的绝大部分操作函数，都是针对这个记录集合进行的。

与连接数据源操作类似，在 CRecordSet 类的派生类对象中调用 Open 函数打开记录集合对象时，完成了创建记录集合的操作。在 Open 函数的实现代码中，调用了 CRecordSet 类的虚函数 GetDefaultSQL，来得到创建记录集合所用到的选择子式，通过在 CRecordSet 派生类中重载并修改这个虚函数，就可以达到修改选择子式，修改记录集合的目的。在应用程序 Data1 的 CData1Set 类中，可以找到重载的 GetDefaultSql 函数的实现代码：

```
CString CDataISet::GetDefaultSQL()
{
    return _T("[city]");
}
```

GetDefaultSQL 函数返回一个字符串,字符串"[city]"表示选择数据源中的数据表 city 中的所有内容形成记录集合。在 Open 函数的实现过程中,"[city]"被处理成以下的 SQL 选择子式:

```
Select * from city
```

当然,也可以从多个数据表中选择内容创建记录集合,在 12.1.2 中产生应用程序的步骤⑤,如果从图 12-2 中的列表框中选择多个数据表,例如选择了两个数据表 city 和 mine,则在产生的应用程序中,可以找到如下的 GetDefaultSQL 函数实现代码:

```
CString CDataISet::GetDefaultSQL()
{
    return _T("[city],[mine]");
}
```

字符串"[city],[mine]"在 Open 函数的实现过程中,被处理成以下的 SQL 选择子式:

```
Select * from city,mine
```

必须注意,使用多个数据表来创建一个记录集合,一般应用在关系数据库中并通过主键字段在数据表之间建立了连接的情况下,否则,创建的记录集合没有实际的意义。例如,创建包含两个数据表的记录集合时,如果这两个表之间没有任何的联系,第一个数据表有 10 条记录,第二个数据表有 5 条记录时,那么创建的记录集合将有 50 条记录(10×5),即记录集合是由第一个数据表的每条记录分别与第二个数据表的所有记录组合后形成的。

选择数据表创建记录集合时,并非一定采用数据表中所有的字段,可以只选择一个或多个数据表中的一部分字段来创建记录集合,以下的 GetDefaultSQL 函数可以只利用数据表 city 中的两个字段 city _ name 和 population 创建记录集合:

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city_name,population from city");
}
```

也可以从多个数据表中有选择性的选择字段来创建记录集合,以下的 GetDefaultSQl 函数,可以实现从两个数据表中选择部分字段来构成一个记录集合:

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city_name,population,mine_name from city,mine");
}
```

其中 mine _ name 是 mine 数据表中的一个字段。

从以上操作可以看出,可以在 GetDefaultSQL 函数内返回第 11 章 11.2.1 小节讨论的选择子式来选择记录集合的内容,在选择子式中也可以增加过滤子式(见 11.2.2 小节)和排序子式

(见 11.2.3 小节)得到经过过滤和排序后的记录集合。不过,在 CRecordSet 派生类中,一般不通过在 GetDefaultSQL 函数中增加过滤和排序子式来完成记录集合的过滤和排序操作,而是通过设置 CRecordSet 类的成员变量 m _ strFilter 和 m _ strSort 来实现,具体操作见 12.2.4 小节和 12.2.5小节。

除了用以上的操作来创建记录集合外,在使用 SQL Server 数据库时,还可以通过创建存储过程来实现相应的操作,具体操作见第 13 章 13.3.4 小节。

在创建记录集合时必须注意,记录集合中所包含的字段改变后,CRecordSet 派生类中记录集合与成员变量数据交换的代码必须随着变化,否则,会引起致命的错误。具体操作参见 12.2.2小节。

12.2.2　与记录集合的数据交换

CRecordSet 派生类对象完成管理记录集合的功能。对记录集合进行管理时,采用如下的方式:在记录集合中包含大量的数据记录,CRecordSet 派生类对象如同一个记录指针,能够指向记录集合中的一条记录,通过在记录集合中滚动,这个记录指针能够指向记录集合中的任意一条记录。当记录指针指向一个记录后,通过将记录中的各个字段与 CRecordSet 派生类中定义的成员变量间进行数据交换,将记录指针指向的记录中各字段的内容用成员变量来表示,同时,又能够通过数据交换用成员变量的值去修改记录指针指向的记录中的各个字段的值。记录集合中当前记录与 CRecordSet 派生类中的成员变量的数据交换,是通过 CRecordSet 类中的一个虚函数 DoFieldExchange 来完成的,在 CRcordSet 派生类中重载并修改这个虚函数,就可以完成 CRcordSet 派生类中的成员变量与派生类对象当前指向的记录之间的数据交换。在应用程序 Data1 中,在 CData1Set 类中可以找到重载的 DoFieldExchange 函数的实现代码:

```
void CData1Set::DoFieldExchange(CFieldExchange * pFX)
{
    //{{AFX_FIELD_MAP(CData1Set)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[only_id]"), m_only_id);
    RFX_Text(pFX, _T("[city_name]"), m_city_name);
    RFX_Long(pFX, _T("[population]"), m_population);
    RFX_Long(pFX, _T("[area]"), m_area);
    RFX_Long(pFX, _T("[industry]"), m_industry);
    RFX_Text(pFX, _T("[translate]"), m_translate);
    RFX_Text(pFX, _T("[enviroment]"), m_enviroment);
    //}}AFX_FIELD_MAP
}
```

在 DoFieldExchange 函数中,通过宏(RFX _ Text、RFX _ Long 等)实现当前记录中的各个字段与类中对应的成员变量的数据交换。如以上的代码 RFX _ Long(pFX, _ T("[city _ name]"), m _ city _ name),实现将记录集合中当前记录中的字段 city _ name 与类中的变量 m _ city _ name 进行数据交换的功能。这种数据交换是双向的,也就是说,通过这个数据交换,可以把记录集合中的一条记录的各个字段内容转移到各成员变量中,同时也可以把类中的成员变量的值转移

到记录集合的当前记录中，实现对记录集合当前记录的修改。

各数据交换宏(RFX_Long、RFX_Text等)，根据CFieldExchange类指针pFX指向的CFieldExchange对象中的参数来决定应该进行读出操作(从当前记录的字段到成员变量)还是进行写入操作(从成员变量到当前记录的字段)。DoFieldExchange函数在许多其他功能中被调用完成数据交换工作，如在移动记录指针(Move函数)、打开记录集合对象(Open函数)等各种操作时，DoFieldExchange被调用来进行读出操作；在进行修改记录集合等操作时被调用进行写入操作。

一般情况下，程序设计者不能直接调用DoFieldExchange函数来进行操作。**一定要保持DoFieldExchange函数中的交换宏与记录集合所包含的字段的一致性**。所谓一致性，就是说在DoFieldExchange函数中必须按照记录集合中所具有的各个字段的顺序对所有的字段通过宏进行数据交换。这就要求在对记录集合所包含的字段进行了改动后，必须对DoFieldExchange函数进行相应修改，以适应修改后的记录集合数据交换的需要。例如，如果记录集合是通过以下GetDefaultSQL函数返回的SQL语句创建的：

```
CString CData1Set::GetDefaultSQL()
{
    return _T("select city_name,population from city");
}
```

则必须将DoFieldExchange函数修改为如下代码：

```
void CData1Set::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CData1Set)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, _T("[city_name]"), m_city_name);
    RFX_Long(pFX, _T("[population]"), m_population);
    //}}AFX_FIELD_MAP
}
```

如果记录集合包含的字段与在DoFieldExchange函数中的宏代码不对应，则在系统运行时会引起致命错误。

对于CRecordSet派生类中成员变量的定义，以及DoFieldExchange函数中的宏代码，程序设计者都不必通过手工书写代码去实现，而可以通过MFC ClassWizard来辅助完成(参见12.4.1小节)。

12.2.3 打开和关闭记录集合对象

记录集合对象的打开和关闭操作是通过调用CRecordSet类的成员函数Open和Close来完成的：

```
m_pSet->Open();
m_pSet->Close();
```

m_pSet是指向CRecordSet派生类对象的指针。记录集合对象在进行打开操作时，实际上是在进行重新创建记录集合的过程。在程序运行过程中，可以通过在记录集合对象中的一些

参数(如过滤子式、排序子式等)修改后,重新打开记录集合对象来重新创建记录集合。

12.2.4　过滤记录集合

在选择数据源中的一个或多个数据表的内容来创建记录集合时,可以只选择数据表中符合某个过滤条件的部分记录。

(1) 过滤操作的方式

在第 11 章 11.2.2 小节,讨论了 SQL 中的过滤子式,在创建记录集合时,可以采用一定的方式来组织过滤子式,达到在创建记录集合时对数据表中的记录进行过滤的目的。实现对记录集合的过滤操作,有几种不同的方式,以下介绍其中的两种。

第一种方式是在 12.2.1 小节讨论的 GetDefaultSQL 函数中加入过滤子式,通过这个函数返回过滤子式。如以下的 GetDefaultSQL 函数返回数据表 city 中符合条件"population > 100"的记录组成一个记录集合(记录集合包含两个字段 city _ name,population):

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city_name,population from city where population > 100");
}
```

第二种方法是通过修改 CRecordSet 类中的成员变量来实现。在 CRecordSet 类中,有一个 CString 型的成员变量 m _ strFilter,用来存储产生记录集合时的过滤条件。通过第一种方式实现的过滤操作,可以通过以下的步骤来实现:

```
if(m_pSet->IsOpen())                    //如果打开了记录集合对象,则关闭它
    m_pSet->Close();
m_pSet->m_strFilter = "population > 100"  //设置过滤条件
m_pSet->Open();                         //重新打开记录集合对象
```

以上代码首先关闭记录集合对象,通过指向记录集合对象的指针 m _ pSet 将其基类 CRecordSet 中的成员变量 m _ strFilter 设置成过滤子式,再重新打开记录集合对象时,就只有符合条件的记录组成记录集合,从而实现了对记录集合的过滤功能。

(2) 过滤操作过程分析

在 CRecordSet 类中有一个 CString 对象 m _ strSQL,这个 CString 对象中存储的是打开 CRecordSet 派生类对象创建记录集合时使用的 SQL 语句。m _ strSQL 字符串是在执行 Open 操作过程中,由几个字符串来合并成的,它来源于以下几个字符串:GetDefaultSQL 函数返回的字符串、CRecordSet 类中的 m _ strFilter 字符串、CRecordSet 类中的 m _ strSort 字符串(m _ strSort 字符串将在 12.2.5 小节讨论)等,具体组成方式如下:将 GetDefaultSQL 函数返回的字符串通过一个字符串"where"与字符串 m _ strFilter 连接,通过一个字符串"Order by"与 m _ strSort 连接,合并组成字符串 m _ strSQL。

在(1)中实现的第一种过滤方式下,如果用 m _ str 来记录 GetDefaultSQL 返回的字符串,则

各字符串内容如下：

```
m_str = "select city_name, population from city where population > 100"
m_strFilter = ""
```

m_strFilter 字符串为空，在 m_strFilter 为空的情况下，m_strSQL 直接由 GetDefaultSQL 返回的字符串得到，其内容为：

```
m_strSQL = " select city_name,population from city where population > 100"
```

通过执行 m_strSQL 字符串表示的 SQL 语句创建了记录集合。

在(1)中实现第二种过滤方式下，GetDefaultSQL 返回的是数据表 city 的全部字段内容，m_strFilter 中记录了过滤子式，这种方式下 GetDefaultSQL 返回的字符串和 m_strFilter 中记录的字符串如下所示：

```
m_str = "Select m_only_id,m_city_name,area,enviroment,industry,population,translate"
m_strFilter = "population > 100"
```

因为 m_strFilter 字符串不为空，所以 m_strSQL 字符串由以上两个字符串通过字符串"where"连接而成：

```
m_strSQL = "Select m_only_id,m_city_name,area,enviroment,industry,population,translate where
    m_strFilter = "population > 100"
```

通过执行 m_strSQL 字符串表示的 SQL 语句创建了记录集合。

通过以上讨论可知，用两种方式都可以实现对数据源中数据表的过滤操作而得到记录集合。**但不能同时采用两种方法，因为这样会引起组合成的 SQL 语句不符合语法规则，引起执行错误。**如，当同时应用两种过滤方法进行过滤操作时，假定 GetDefaultSQL 函数返回的字符串(用 m_str 表示)和 m_strFilter 字符串被设置成如下内容：

```
m_str = "select city_name, population from city where population > 100"
m_strFilter = "population > 100"
```

则在这种情况下，m_strSQL 字符串的内容仍为两个字符串通过"where"字符串连接而成，得到字符串如下：

```
m_strSQL = "select city_name, population from city where population > 100 where population > 100"
```

这个 SQL 语句不符合 SQL 语法规则，在创建记录集合时会引起致命错误。

(3) 一般采用的过滤方法

本小节讨论了两种对数据源中的数据表进行过滤创建记录集合的方式。在对数据源采用固定过滤条件的过滤操作的情况下，这两种过滤方式能够达到同样的目的。但是在动态的创建记录集合来完成动态的数据库过滤功能时，采用第二种方法具有更大的灵活性和方便性。一般采用第二种方法来组织对记录集合的过滤操作，实现数据库过滤(或查询)功能，实现步骤如下：

① 关闭记录集合对象。

② 在打开记录集合对象前，设置 CRecordSet 类的成员变量 m _ strFilter。这个设置过程，在具体实现上可以采用各种形式，可以通过提供交互式的操作界面，利用各种控制，来选择和组织各种过滤条件，然后将动态产生的过滤条件赋个成员变量 m _ strFilter。在 16 章，将讨论实现一个任意组织过滤条件式的过滤条件编辑器。

③ 重新打开记录集合对象。重新打开记录集合对象，就可以得到经过过滤操作创建的记录集合。

12.2.5　排序记录集合

所谓排序记录集合，是指在创建记录集合时，需要对来源于数据源中的数据表的数据记录按照一定的字段进行排序。在 12.2.4 小节讨论了创建记录集合时的过滤操作，排序操作的实现过程与过滤操作非常类似。本小节简要介绍一下对记录集合排序操作的方法和步骤。

(1) 排序操作的方式

与对记录集合的过滤操作类似，同样可以采用两种不同的方式来实现排序操作。

第一种方法是通过修改 GetDefaultSQL 函数来完成。例如采用以下的 GetDefaultSQL 函数可以以 Population 字段排序，从数据表 city 中返回一个记录集合(记录集合包含两个字段 city _ name，population)：

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city _ name,population from city Order by population");
}
```

对于字段 population，采用的升序排列顺序(默认的排序顺序)。在排序时可以显式的设置升序和降序，ASC 代表升序，DESC 代表降序，对于以上 GetDefaultSQL 函数，可将字段 population 修改为降序的排序方式：

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city _ name,population from city Order by population DESC");
}
```

也可以利用多个字段进行排序。例如，在创建记录集合时，需要利用字段 population 和 area 两个字段对记录集合进行排序，排序方式分别为升序和降序，则 GerDefaultSQL 函数实现代码如下：

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city _ name,population from city Order by population,area DESC");
}
```

第二种方式是通过对 CRecordSet 类的一个成员变量进行修改来实现。与排序操作时类似，在 CRecordSet 类中有一个成员变量 m _ strSort，它是用来存储 SQL 排序子式的。对于以下

GetDefaultSQL 函数完成的排序操作：

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city_name,population from city Order by population,area DESC");
}
```

可以通过以下的步骤来实现：在 GetDefaultSQL 函数中，在返回的字符串中不设置排序子式：

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city_name,population from city ");
}
```

在调用 Open 函数打开记录集合对象前，将 m_strSort 字符串设置成排序子式：

```
if(m_pSet->IsOpen())                    //如果打开了记录集合对象,则关闭它
    m_pSet->Close();
m_pSet->m_strSort=" population,area DESC"  //设置排序子式
m_pSet->Open();                         //重新打开记录集合对象
```

(2) 排序操作过程分析

与过滤操作时相同，CRecordSet 类中的 m_strSort 字符串在前面加上“order by ”字符串，与 GetDefaultSQL 返回的字符串合并成 m_strSQL 中的字符串。必须注意，如果 m_strFilter 字符串不为空，GetDefaultSQL 函数返回的字符串首先与 m_strFilter 字符串通过“where”连接，然后与 m_strSort通过“order by”连接得到 m_strSQL。如，GetDefaultSQL 函数如下所示：

```
CString CDataISet::GetDefaultSQL()
{
    return _T("select city_name,population from city ");
}
```

在进行如下操作时：

```
if(m_pSet->IsOpen())                    //如果打开了记录集合对象,则关闭它
    m_pSet->Close();
m_pSet->m_strFilter=" population>100"   //设置过滤子式
m_pSet->m_strSort=" population,area DESC"  //设置排序子式
m_pSet->Open();                         //重新打开记录集合对象
```

如果以 m_str 记录 GetDefaultSQL 函数返回的字符串，则各字符串的内容为：

```
m_str="select city_name,population from city "
m_strFilter=" population>100"
m_strSort=" population,area DESC"
```

合并后得到的 m_strSQL 字符串如下所示：

```
m_str="select city_name,population from city where population>100
    order by population,area DESC"
```

执行由 m_strSQL 字符串记录的 SQL 语句来创建记录集合。

同样,不能同时采用两种方式来组织排序操作,因为这样会引起组合成的 SQL 语句不符合语法规则,引起执行错误。

(3) 排序操作的实现

在以固定的排序方式产生记录集合时,两种排序方式能够达到同样的的功能。但是在数据库应用程序中,需要动态的组织排序操作来创建记录集合时,一般需要采用第二种方式,通过动态产生成员变量 m_strSort 所存储的字符串的内容,实现根据具体的要求来动态的创建记录集合。对于排序操作的实现,一般采用如下步骤:

① 关闭记录集合对象。

② 在打开记录集合对象前,设置成员变量 m_strSort 的内容。这个设置过程,在具体实现上可以采用各种形式,可以通过交互式操作界面,利用各种控制,通过选择和设置得到排序方式,然后把选择的内容转变成字符串的方式而赋给成员变量 m_strSort。在第 17 章,将组织实现一个任意组织排序子式的编辑器。

③ 重新打开记录集合对象。这时得到的是经过指定方式排序后的记录集合。

12.2.6　移动指向记录集合的记录指针

一个 CRecordSet 派生类对象用来管理一个记录集合,一个记录集合包含大量的数据记录,而 CRecordSet 派生类对象的记录指针指向的是记录集合中的一条记录,通过 CRecordSet 派生类中的成员变量与这条记录中的字段进行数据交换,达到管理这条记录的目的。CRecordSet 类中提供了一些成员函数,用来移动指向记录集合对象的记录指针,来指向记录集合中的其他记录,同时提供成员函数检测移动指针操作是否成功。

(1) 在记录集合中移动记录指针

在 CRecordSet 类中提供了多个成员函数,来移动记录集合中的记录指针,以下是几个主要的成员函数。

① MoveFirst 函数。MoveFirst 函数通过移动记录指针,指向记录集合中的第一条记录。以下是函数的具体调用:

```
m_pSet->MoveFirst();
```

m_pSet 是指向一个记录集合对象(CRecordSet 派生类对象)的指针。

② MoveLast 函数。MoveLast 函数与 MoveFirst 函数正好相反,移动记录指针指向记录集合的最后一条记录。以下是函数的具体调用:

```
m_pSet->MoveLast();
```

③ MovePrev 函数。MovePrev 函数移动记录指针,指向记录集合中当前记录的上一条记录。即将记录指针向前移动一条记录。以下是函数的具体调用:

```
m_pSet->MovePrev();
```

④ MoveNext 函数。MoveNext 函数正好与 MovePrev 函数相反，移动记录指针，指向记录集合中的下一条记录。以下是函数的具体调用：

```
m_pSet->MoveLast();
```

⑤ Move 函数。Move 函数是移动记录指针的基础函数，以上的四个函数都是这个函数的特例，都是通过调用这个函数来实现的，从 MFC 中可以找到以上四个函数的实现代码：

```
_AFXDBCORE_INLINE void CRecordset::MoveNext()
{ ASSERT(IsOpen()); Move(1, SQL_FETCH_NEXT); }
_AFXDBCORE_INLINE void CRecordset::MovePrev()
{ ASSERT(IsOpen()); Move(-1, SQL_FETCH_PRIOR); }
_AFXDBCORE_INLINE void CRecordset::MoveFirst()
{ ASSERT(IsOpen()); Move(1, SQL_FETCH_FIRST); }
_AFXDBCORE_INLINE void CRecordset::MoveLast()
{ ASSERT(IsOpen()); Move(-1, SQL_FETCH_LAST); }
```

从以上代码可以看出，成员函数 MoveFirst、MoveLast、MovePrev、MoveNext 是通过以不同的参数来调用成员函数 Move 实现的。Move 函数的定义如下：

```
virtual void Move(long nRows, WORD wFetchType = SQL_FETCH_RELATIVE);
```

Move 函数有两个参数，第一个参数是相对移动的记录数，在记录集合中向后面的记录移动为正，向前面的记录移动为负。第二个参数是移动时的参数选项，可以用 SQL_FETCH_NEXT、SQL_FETCH_PREV、SQL_FETCH_FIRST、SQL_FETCH_LAST、SQL_FETCH_RELATIVE、SQL_FETCH_ABSOLUTE 等参数，默认参数为 SQL_FETCH_RELATIVE，表示从记录集合的当前记录做相对移动。以下两行代码都可以将记录指针向后移动两条记录：

```
m_pSet->Move(2);
m_pSet->Move(2,SQL_FETCH_RELATIVE);
```

不同于 SQL_FETCH_RELATIVE 参数，SQL_FETCH_ABSOLUTE 参数表示从记录集合的开始做相对移动，以下代码表示将记录集合的指针指向记录集合的第三条记录：

```
m_pSet->Move(3,SQL_FETCH_ABSOLUTE);
```

(2) 检查移动记录指针操作的结果

当在记录集合中移动记录指针时，在某些情况下可能会使指针移动到记录集合之外。例如，如果当前记录指针指向记录集合的第一条记录，进行 MovePrev 操作时就使记录指针指向了第一条记录的前面；同样，如果当前记录指针指向记录集合的最后一条记录，进行 MoveNext 操作也会将记录指针移动到记录集合外。对指向记录集合外的记录指针调用 CRecordSet 类的成员函数进行操作时，会引起操作错误。所以，在一个设计完善的数据库管理系统中，对于记录指针的移动等操作，在操作前或操作后必须对记录指针进行检查，以确定记录指针是否指向有效的记录。以下是 CRecordSet 类中的几个常见检查函数。

① IsBOF 函数。IsBOF 成员函数的定义如下：

```
BOOL IsBOF()const;
```

IsBOF 函数用来检测记录指针是否位于当前记录集合中第一条记录的前面。记录指针在记录集合第一记录的前面时，函数返回 TRUE，否则，函数返回 FALSE。在记录集合中向前移动记录指针时，一般通过这个函数进行检测，确保记录指针指向有效的记录。以下的代码完成了这个检测功能：

```
m_pSet->MovePrev();              //向前移动一条记录
if(m_pSet->IsBOF())              //如果指针指向了记录集合范围外
{
    ......//加入提示信息，表明记录移动出了范围
    m_pSet->MoveFirst();         //将记录指针指向第一条记录
}
```

在打开一个记录集合对象时，可以通过成员函数 IsBOF 检查记录指针来确定一个记录集合是否为空(不包含任何记录)：

```
m_pSet->Open();                  //打开记录集合对象
if(m_pSet->IsBOF())              //如果记录指针指向了记录集合范围外
{
    ......//加入处理代码，处理记录集合为空的情况
}
```

② IsEOF 函数。IsEOF 函数的定义如下：

```
BOOL IsEOF()const;
```

IsEOF 函数用来检测记录指针是否位于记录集合最后记录的后面。如果记录指针在记录集合最后记录之后，函数返回 TRUE，否则，函数返回 FALSE。在记录集合中向后移动记录指针时，一般通过这个函数进行检测，确保记录指针指向有效的记录。如以下代码所示：

```
m_pSet->MoveNext();              //向后移动一条记录
if(m_pSet->IsEOF())              //如果记录指针指向了记录集合范围外
{
    ......//加入提示信息，表明记录移动出了范围
    m_pSet->MoveLast();          //将记录指针指向最后一条记录
}
```

③ CanScroll 函数。在 CRecordSet 类中，有一个成员函数 CanScroll，用来检测当前的记录集合对象是否支持滚动功能。所谓滚动功能，是指在记录集合中对记录指针进行移动操作(如 MoveFirst、MoveLast、MovePrev、MoveNext 等操作)。在利用 ODBC 进行数据库应用程序设计时，一般首先利用 CanScroll 函数来检测当前的记录集合对象是否支持滚动操作，根据检测的结果来控制程序的功能和流程。CanScroll 函数在 CRecordSet 类中定义如下：

```
BOOL CanScroll() const;
```

当记录集合对象支持滚动操作时，函数返回 TRUE，否则，函数返回 FALSE。以下是函数的

调用代码：

```
if(! m_pSet->CanScroll())          //如果记录集合对象不能支持滚动功能
    ......//加入具体的处理过程
```

12.2.7 修改记录集合

CRecordSet 类提供对记录集合的编辑功能，并通过对记录集合的编辑转而对数据源中的实际数据表进行修改。记录集合的编辑功能主要存在三个方面：增加记录、修改记录和删除记录。增加记录就是在当前的记录集合中增加一条新记录，并通过记录集合对象在数据源的实际数据表中也增加记录；修改记录指的是对记录指针指向的当前记录的内容进行修改，并通过记录集合对象修改数据源中的数据表中的对应记录；删除记录是删除记录集合中记录指针指向的当前记录，并通过记录集合对象对数据源中的实际数据表中的相应记录进行删除。所以，对于记录集合的编辑功能包含如下两个方面：对记录集合进行编辑修改和使对记录集合进行的修改影响到数据源，实现对实际数据表的修改。以下具体讨论在 CRecordSet 类中对增加、修改、删除记录操作的实现方法：

(1) 增加新记录

在记录集合中增加一条记录的操作过程比较简单。但是，必须特别注意在增加操作时的几个问题。增加新记录操作，需要以下几个过程：

① 调用 CRecordSet 类中的 AddNew 函数，在当前记录集合中增加一条新记录，并将记录指针指向这条新增加的记录。具体操作如下：

```
m_pSet->AddNew();
```

m_pSet 是指向 CRecordSet 派生类对象(记录集合对象)的指针。记录集合对象不一定都能够支持增加新记录的操作，在 CRecordSet 类中有一个成员函数 CanAppend 用来检测当前的记录集合对象是否支持增加新记录的操作。以下代码实现在增加新记录前首先检测记录集合对象是否支持增加新记录操作：

```
if(m_pSet->CanAppend())          //如果记录集合对象支持增加新记录的操作
    m_pSet->AddNew();
else
    ......//加入具体的处理代码
```

② 修改数据源的内容。以上操作只是在记录集合中增加了一条新记录，并没有真正修改通过 ODBC 所连接数据源中的数据表。必须通过一定的操作，利用记录集合对象来更新数据源中的实际数据表，达到增加新记录的目的。在 CRecordSet 类中有一个成员函数 Update。Update 函数在调用了成员函数 AddNew 或 Edit 后被调用，来完成更新数据源的功能。所以，当利用 AddNew 成员函数在记录集合中增加了一新记录后，必须调用 Update 函数来更新数据源，达到真正增加新记录的目的。具体实现代码如下：

```
m_pSet->Update();
```

必须注意！一般情况下，并不是在调用 AddNew 函数后直接调用 Update 函数来增加一个空的记录。因为在某些数据表中，有一些字段是不能为空的(NOT NULL)，在调用 AddNew 函数后直接调用 Update 函数不能正常完成操作，即不能对数据源的实际数据表进行修改。所以，当调用了 AddNew 在记录集合中增加一条新记录后，可通过调用表单视图或对话框等组织的交互操作界面，对新增加记录的字段内容进行编辑修改，然后调用 Update 函数来对数据源的数据表进行更新。在 12.4.3 小节，将讨论增加记录的具体实现过程。

并不是所有的数据源都能够支持修改操作，即能够通过记录集合对象对数据源进行修改。所以，在调用 Update 函数对数据源进行修改前，需要调用 CRecordSet 类的成员函数 CanUpdate 来检测是否能够对数据源进行修改操作。CanUpdate 函数定义如下：

```
BOOL CanUpdate() const;
```

当记录集合对象能够对数据源进行修改时，函数返回 TRUE，否则函数返回 FALSE。以下代码，在对数据源进行修改前首先检查能否对数据源进行修改：

```
if(m_pSet->CanUpdate())        //如果能够对数据源进行修改，则修改数据源
    m_pSet->Update();
```

③ 重新创建记录集合对象。因为记录集合可能只是数据源的数据表中符合过滤条件的部分记录，而新增加的记录可能并不符合当前的过滤条件，为了避免操作错误，在增加新记录后应该重新创建记录集合。即采用关闭并重新打开记录集合对象的方法：

```
if(m_pSet->IsOpen())           //如果记录集合对象处于打开状态
    m_pSet->Close();           //关闭记录集合对象
m_pSet->Open();                //打开记录集合对象
```

也可以通过调用 CRecordSet 类的 Requery 函数重新产生记录集合包含的内容。

```
m_pSet->Requery();
```

(2) 修改记录

(1) 中讨论了对记录集合增加新记录的操作步骤，除此之外，还可以对记录集合中的当前记录进行修改。修改记录操作的过程与增加操作类似。以下是实现的步骤：

① 开始修改记录操作。为了对记录集合的当前记录进行修改，首先使当前记录处于编辑状态，这个过程是通过调用 CRecordSet 类的 Edit 函数来实现的。Edit 函数的定义如下：

```
virtual void Edit();
```

在对记录集合的当前记录进行编辑操作前，必须首先调用 Edit 函数：

```
m_pSet->Edit();
```

Edit 函数并不完成对当前记录的修改操作，而只是将当前记录设置成修改状态，只有运行了 Edit 函数后，记录集合中的当前记录才能被编辑修改，才能使对记录集合的修改影响到数据源中的实际数据表。也就是说，只有运行了 Edit 函数后，才能调用 Update 函数实现对数据源中数据表中的对应记录的修改操作。Update 函数只有在 AddNew 和 Edit 函数调用后，才能被

调用,来实现对数据源的修改操作。

② 在调用了 Edit 函数后,其他的操作步骤与(1)中增加新记录操作时的②、③步骤相同,不再进行重述。

(3) 删除记录

对于记录集合中当前记录的删除,是通过调用 CRecordSet 类的成员函数 Delete 实现的。删除操作的实现比较简单,以下代码可以将记录集合中的当前记录删除,并删除数据源中的实际数据表的对应记录:

```
m_pSet->Delete();
```

必须注意! 进行删除操作后,应该将记录集合中的记录指针指向其他的记录。以下代码实现这个操作过程:

```
m_pSet->Delete();                    //删除记录集合中的当前记录
m_pSet->MoveNext();                  //将记录指针指向下一条记录
if(m_pSet->IsEOF())                  //如果记录指针超出了范围
    m_pSet->MoveLast();              //指向最后一条记录
if(m_pSet->IsBOF())                  //如果指向第一条记录的前面,即当前记录集合没有记录
    m_pSet->SetFieldNull(NULL);      //设置为空记录,即所有的字段为空
```

12.2.8 CRecordSet 类的其他操作功能

在以上各小节,结合具体的功能介绍了 CRecordSet 类中的一些成员函数。这一小节将介绍 CRecordSet 类中的其他一些常用成员函数。

(1) 书签的操作函数

某些记录集合能够支持书签操作。所谓书签,就是给记录集合中的某条记录一个特殊的标记(书签)。对于设置了书签的记录,可以根据书签的值将记录集合的指针移动到这条记录。在 CRecordSet 类中有以下几个与书签操作有关的成员函数:

```
BOOL CanBookmark()const;
void GetBookmark(CDBVariant& varBookmark);
void SetBookmark(const CDBVariant& varBookmark);
```

CanBookmark 函数用来检测记录集合对象是否允许用书签的形式为记录做标记。如果能够以书签做标记,函数返回 TRUE,否则,函数返回 FALSE;GetBookmark 函数用来返回当前记录中所做的标签;SetBookmark 将记录指针移动到书签值所代表的记录上。

(2) 获得记录集合对象的信息

在 CRecordSet 类中有大量的成员函数,用来得到记录集合对象中有关记录集合的信息,以下对主要的一些成员函数进行介绍:

① GetRecordCount 函数。GetRecordCount 函数在 CRecordSet 类中定义如下：

```
long GetRecordCount()const;
```

GetRecordCount 函数用来得到记录集合的记录数目。以下代码用来获得 m_pSet 指向的记录集合对象中包含的记录数目：

```
long nRecordCount = m_pSet->GetRecordCount();
```

必须注意，对于某些记录集合对象，在没有对记录集合用指针进行遍览的情况下，不能正确的得到记录数。在打开记录集合对象时直接调用 GetRecordCount 函数：

```
m_pSet->Open();
long nRecordCount = m_pSet->GetRecordCount();
```

即使在记录集合中有多于一条记录的情况下，函数返回的值 nRecordCount 也为 1，只有在对记录集合进行遍历后，才能返回正确的值：

```
m_pSet->Open();                                  //打开记录集合对象
m_pSet->MoveFirst();                             //移动到记录集合的第一条记录
while(! m_pSet->IsEOF())                         //遍历记录集合的各记录
    m_pSet->MoveNext();                          //记录指针指向下一条记录
long nRecordCount = m_pSet->GetRecordCount();    //得到记录集合中的记录数目
```

② GetODBCFieldCount 函数。GetODBCFieldCount 函数在 CRecordSet 类中定义如下：

```
short GetODBCFieldCount() const;
```

GetODBCFieldCount 函数用来得到记录集合对象中的字段数目，即记录集合包含着几个字段。以下代码可以用来获得 m_pSet 指向的记录集合对象包含的字段数目：

```
short nFieldCount = m_pSet->GetODBCFieldCount();
```

对于按以下的 GetDefaultSQL 函数来组织实现的记录集合，以上代码返回 2，即记录集合中有两个字段。

```
CString CData1Set::GetDefaultSQL()
{
    return _T("select city_name,population from city");
}
```

③ GetODBCFieldInfo 函数。GetODBCFieldInfo 函数在 CRecordSet 类中有两个重载，函数定义如下：

```
void GetODBCFieldInfo(LPCTSTR lpszName,CODBCFieldInfo& fieldinfo);
void GetODBCFieldInfo(short nIndex lpszName,CODBCFieldInfo& fieldinfo);
```

GetODBCFieldInfo 函数用来得到记录集合对象中某个字段的结构信息，字段的结构信息通过一个 CODBCFieldInfo 结构返回，结构中包含了字段名称、类型、长度等各方面的信息。第一个 GetODBCFieldInfo 函数返回名称为第一个参数的字段信息，第二个函数返回第一参数确定的序列号(即第几个字段)的字段信息。以下代码用来返回 m_pSet 指向的记录集合对象中“city

_ name"字段的信息：

```
CODBCFieldInfo fieldinfo1;                    //定义一个 CODBCFieldInfo 结构
CODBCFieldInfo& fieldinfo = fieldinfo1;       //定义一个 CODBCFieldInfo 结构的引用
m_pSet->GetODBCFieldInfo("city_name",fieldinfo);
```

以上代码中，字段信息通过 fieldinfo 引用返回，即返回的信息记录在了 fieldinfo1 结构中。同样可以利用 GetODBCFieldInfo 函数的另外一个重载，返回记录集合中某个字段的信息。以下代码返回 m_pSet 指向的记录集合对象中第二个字段的信息：

```
CODBCFieldInfo fieldinfo1;                    //定义一个 CODBCFieldInfo 结构
CODBCFieldInfo& fieldinfo = fieldinfo1;       //定义一个 CODBCFieldInfo 结构的引用
m_pSet->GetODBCFieldInfo(1,fieldinfo);
```

因为字段序号是 0 基的(即第一个字段的序号为 0)，所以以上代码中通过参数"1"来返回第二个字段的信息。

④ GetFieldValue 函数。GetFieldValue 函数用来返回记录集合中当前记录的某个字段的值。在 CRecordSet 类中有四个重载函数：

```
void GetFieldValue(LPCTSTR lpszName,CDBVariant& varValue,short
    nFieldType = DEFAULT_FIELD_TYPE);
void GetFieldValue(short nIndex,CDBVariant& varValue,short nFieldType = DEFAULT_FIELD_TYPE);
void GetFieldValue(LPCTSTR lpszName ,CString& strValue);
void GetFieldValue(short nIndex,CString& strValue);
```

第一个函数重载用来返回记录集合中当前记录中名称为参数 lpszName 的字段的值，字段的值通过函数的第二个参数返回，第三个参数为字段的类型，一般采用默认类型。以下代码返回 m_pSet 指向的记录集合对象中当前记录中字段"city_name"的值：

```
CDBVariant varValue1;
CDBVariant& varValue = varValue1;
GetFieldValue("city_name",varValue);
```

以上代码中，字段值通过 varValue 引用返回，即字段值记录在了 varValue1 结构中。同样可以利用 GetFieldValue 函数的第二个重载返回记录集合中某个字段的值。以下代码用来返回 m_pSet指向的记录集合对象中第二个字段的值：

```
CDBVariant varValue1;
CDBVariant& varValue = varValue1;
GetFieldValue(1,varValue);
```

因为字段序号是 0 基的(即第一个字段的序号为 0)，所以以上代码中通过参数"1"来返回第二个字段的值。

GetFieldValue 函数的第三个和第四个重载用字符串来返回指定字段的内容。即不管字段是什么类型，都是返回表示这个字段的字符串。以下代码返回 m_pSet 指向的记录集合对象中当前字段中的"city_name"字段的字符串内容：

```
CString varValue1;
```

```
CString& varValue = varValue1;
GetFieldValue("city _ name",varValue);
```

记录集合当前记录中的字段“city _ name”的内容记录在了字符串 varValue1 中。同样可以用第四个函数重载返回某个序号字段的字符串内容。

在 CRecordSet 类中还有其他的一些成员函数，这些成员函数的具体应用可查阅 CRecordSet 类的使用手册。

12.2.9　MFC ClassWizard 管理 CRecordSet 派生类

在数据库应用程序中，利用 CRecordSet 的派生类组织对记录集合的管理功能时，可以利用 MFC ClassWizard 作为辅助工具简化程序设计的步骤。在第 2 章 2.5 节，对 MFC ClassWizard 完成的功能进行了详细剖析。这一小节将具体介绍和剖析利用 MFC ClassWizard 管理 CRecordSet 派生类的一些具体操作。

(1) 重载 CRecordSet 类的成员函数

打开用 MFC AppWizard 产生的应用程序 Data1(参见 12.1.2 小节)，启动进入 ClassWizard 操作界面(参见第 2 章 2.3.1 小节)，在 ClassWizard 对话框的“Class name”列表框中选择类 CData1Set，选择“Message Maps”标签，进入图 12-3 所示的对话框。在这个对话框中，可以增加或删除 CRecordSet 类的虚函数，具体的操作可参见 2.5.5 小节。

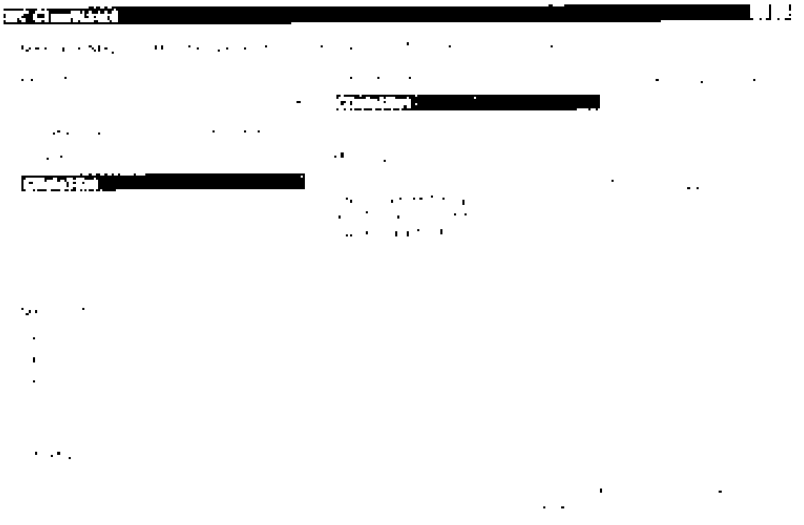


图 12-3　ClassWizard 虚函数管理对话框

(2) 得到 CRecordSet 派生类管理的记录集合的字段

在图 12-3 所示的对话框中选择“Member Variables”标签，进入到如图 12-4 所示的对话框。

在图 12-4 对话框中,列表框中列出了 CRecordSet 派生类 CDataISet 所管理的记录集合的各个字段及其对应的各映射变量,在"Column Names"下列出的是记录集合所具有的各字段。在某些情况下,需要修改 CDataISet 类连接的数据源,按中"Update Columns"按钮,进入到图 12-1 所示的操作界面,按照 12.1.2 小节中介绍的步骤进行操作,选择需要连接的 ODBC 数据源和数据表,返回到图 12-4 所示的操作界面,就会在列表框的"Column Names"中列出所选择的数据表中包含的字段。



图 12-4 ClassWizard 映射变量管理对话框

必须注意,以上对 ODBC 数据源和数据源中数据表的选择操作,其目的只是更改了 ClassWizard 所使用的信息,即在图 12-4 所示的列表框中能够列出这些数据表的字段,供绑定字段到映射变量时用,这些设定并不能改变 CDataISet 类的函数 GetDefaultDefault 和 GetDefaultSQL 中的代码,从而改变 CRecordSet 的派生类 CDataISet 所连接的数据源和管理的记录集合的字段。

(3) 绑定字段

绑定字段就是通过增加映射变量,建立 CRecordSet 派生类中的变量与记录集合对象中的字段的交换机制。在利用 MFC AppWizard 产生数据库应用程序框架时(见 12.1.2 小节),默认对所有的字段已经进行了绑定。现在介绍用 MFC ClassWizard 绑定一个字段的操作过程:从图 12-4 所示对话框的列表框中选择一个字段(选中一个"Column Names"列表中的项),按"Add Variable..."按钮,就会进入到如图 12-5 所示的对话框。在这个对话框中,在"Member Variable name"列表编辑框中输入一个映射变量的名称,在"Variable Type"列表框中选择变量的类型,按"OK"按钮退出图 12-5 对话框并返回到图 12-4 所示的对话框,就完成了增加一个映射变量即一个记录集合字段的绑定过程,在列表框中就会看到刚刚增加的映射变量。在这个增加映射变量过程中,ClassWizard 对程序的代码进行了多处修改。以给记录集合中的 area 字段增加映射变量为例,介绍对程序代码所做的修改:

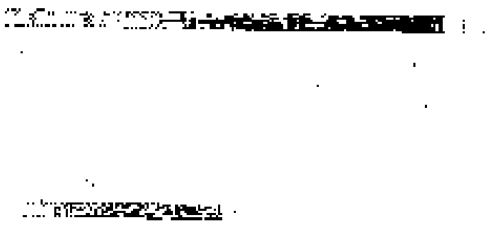


图 12-5　增加映射变量对话框

① 在 CDataISet 类中增加了映射变量的定义：

```
//{{AFX_FIELD(CDataISet, CRecordset)
long m_only_id;
CString m_city_name;
long m_area;
CString m_enviroment;
long m_industry;
long m_population;
CString m_translate;
//}}AFX_FIELD
```

必须注意，以下两行代码被 ClassWizard 用来识别记录集合字段对应的映射变量定义：

```
//{{AFX_FIELD(CDataISet, CRecordset) //ClassWizard 来标识记录集合字段绑定变量定义的开始
//}}AFX_FIELD                        //ClassWizard 来标识记录集合字段绑定变量定义的结束
```

只有在这两行代码内定义的变量才能被 ClassWizard 识别，作为记录集合字段的映射变量。第一行代码括号内有两个变量，第一个变量为类的名称，第二个变量是类的基类名称。

② 在 CDataISet 类的构造函数中，加入了映射变量的初始化代码：

```
CDataISet::CDataISet(CDatabase* pdb)
    : CRecordset(pdb)
{
    //{{AFX_FIELD_INIT(CDataISet)
    m_only_id = 0;
    m_city_name = _T("");
    m_area = 0;
    m_enviroment = _T("");
    m_industry = 0;
```

```
    m_population = 0;
    m_translate = _T("");
    m_nFields = 7;
    //}}AFX_FIELD_INIT
    m_nDefaultType = snapshot;
}
```

初始化代码必须放在如下两行代码内，才能被 ClassWizard 所识别：

```
//{{AFX_FIELD_INIT(CData1Set)      //ClassWizard 用来识别绑定变量初始化的开始
//}}AFX_FIELD_INIT                 //ClassWizard 用来识别绑定变量初始化的结束
```

第一行代码括号内有一个参数，表示的是这些代码属于类的名称。在构造函数代码中，还有一行代码用于对 m_nFields 变量的赋值，这个变量是 CRecordSet 类的成员变量，在此由 ClassWizard 管理，用来记录绑定变量的数目，当增加一个绑定变量时，其值增加 1，当删除一个绑定变量时，其值减少 1。

③ 在成员函数 DoFieldExchange 中，增加了记录集合字段与成员变量的数据交换宏：

```
void CData1Set::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CData1Set)      //ClassWizard 用来表示绑定变量交换机制的开始
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[only_id]"), m_only_id);
    RFX_Text(pFX, _T("[city_name]"), m_city_name);
    RFX_Long(pFX, _T("[area]"), m_area);
    RFX_Text(pFX, _T("[enviroment]"), m_enviroment);
    RFX_Long(pFX, _T("[industry]"), m_industry);
    RFX_Long(pFX, _T("[population]"), m_population);
    RFX_Text(pFX, _T("[translate]"), m_translate);
    //}}AFX_FIELD_MAP                 //ClassWizard 用来表示绑定变量交换机制的结束
}
```

DoFieldExchange 函数是 CRecordSet 类的虚函数，在当前类 CData1Set 中重载，在这个函数中实现映射变量与记录集合中字段的数据交换（见 12.2.2 小节）。当用 ClassWizard 增加一个绑定变量时，就会在这个函数中增加一行数据交换宏代码。宏代码必须写在以下两行代码之间才能被 ClassWizard 所识别：

```
//{{AFX_FIELD_MAP(CData1Set)      //ClassWizard 用来识别绑定变量交换机制的开始
//}}AFX_FIELD_MAP                 //ClassWizard 用来识别绑定变量交换机制的结束
```

第 1 行代码括号内的参数是当前交换机制所属于的类。

(4) 绑定全部记录集合字段

ClassWizard 还提供了一个功能，那就是一次对记录集合具有的所有字段进行绑定。选择

运行图 12-4 对话框界面中的“Bind All”按钮，ClassWizard 就会把记录集合中的所有字段用根据字段名称给定的默认映射变量进行绑定，一次性实现了对所有未绑定字段的绑定操作。其具体实现步骤相当于按照(3)的步骤分别对各记录集合字段进行绑定操作，对程序代码的修改过程与(3)中的操作相同。

(5) 删除对记录集合字段的绑定

删除绑定是(3)中对一个字段绑定操作的逆操作。操作过程非常简单，在图 12-4 所示的对话框界面中，在列表框中用鼠标点中要删除绑定的字段，按中“Delete Variable”按钮，就会删除对选中的字段的绑定。对于字段的绑定删除操作包含了如下的操作步骤：

① 在 CData1Set 类的定义中删除字段所对应的映射变量的定义代码。

② 在 CData1Set 类的构造函数中，删除对映射变量的初始化代码，并将记录绑定字段数目的变量 m_nFields 减 1。

③ 在 CData1Set 类的 DoFieldExchange 函数中，删除字段与其映射变量之间的交换宏代码。

(6) 进行绑定字段操作时必须注意的问题

以上讨论了用 ClassWizard 来实现绑定字段的操作过程，**在管理记录集合的绑定操作时必须注意，对于记录集合字段的绑定顺序和内容必须严格符合记录集合所实际包含的字段。**在 12.2.1 小节的讨论可知，记录集合是通过选择 ODBC 数据源中的一个或多个数据表中的一些字段组成的，记录集合的记录通过 DoFieldExchange 函数与 CRecordSet 类中的映射变量进行数据交换，必须按照记录集合所包含字段的顺序一一对应的对记录集合的字段进行绑定。换言之，记录集合中包含几个字段，就只能有几个对应的映射变量，并且在函数 DoFieldExchange 中，记录集合字段与映射变量的交换顺序必须严格按照记录集合中的字段顺序。记录集合中的字段与字段绑定不对应(有些记录集合中的字段没有绑定或者建立了没有包含在记录集合中的字段的绑定，或者对于一个字段建立了多于一个的绑定)，都会在打开记录集合对象时引起致命错误。记录集合字段的顺序与 DoFieldExchange 函数中各交换宏代码的顺序不一致，在打开记录集合对象时，同样会产生运行错误或数据交换不正确。

12.3　CRecordView 视图类分析

本节将对 12.1.2 小节中产生的数据库应用程序 Data1 的视图类进行分析，并以此为基础，介绍 CRecordView 类的一些基本功能。

12.3.1　CRecordView 视图类

(1) 表单视图

普通 MFC 应用程序的视图类是由 MFC 的 CView 类或者 CScrollView 类派生的。而在 12.1.2小节用 AppWizard 创建的数据库应用程序 Data1 中，视图类 CData1View 是由 CRecordView 类派生的。由 CRecordView 类派生的视图，称之为**表单视图**。CRecordView 类是 CFormView 类

的派生类。由 CFormView 类直接或间接派生的类对象管理的视图是普通视图与对话框的联合体,这种视图从创建过程或窗口的性质上看,它属于文档/视图体系中的视图。视图中的各种控制与菜单项类似,可以被各个基本类(文档类、视图类、主框架类、子框架类、应用类)来使用。而在具体的组成内容和外观上,又属于一个对话框的范畴。

创建表单视图时需要一个对话框资源,在应用程序 Data1 视图类 CData1View 的定义中,可以找到产生视图时需要的对话框资源:

```
public:
    //{{AFX_DATA(CData1View)
    enum{ IDD = IDD_DATA1_FORM };
    CData1Set* m_pSet;
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA
```

枚举定义 IDD 等于资源 ID 号 DD_DATA1_FORM。IDD_DATA1_FORM 是一个对话框资源,这个对话框资源构成了 CData1View 视图的界面。从视图类 CData1View 的构造函数中,可以找到如下的代码:

```
CData1View::CData1View()
    : CRecordView(CData1View::IDD)
{
    //{{AFX_DATA_INIT(CData1View)
        // NOTE: the ClassWizard will add member initialization here
    m_pSet = NULL;
    //}}AFX_DATA_INIT
    // TODO: add construction code here
}
```

在创建视图类 CData1View 对象时,以对话框资源号 IDD 为参数调用 CRecordView 类的构造函数,来产生以对话框资源 IDD_DATA1_FORM 为显示界面的视图。

(2) 视图对象的创建过程

表单视图类 CData1View 对象的创建过程与普通视图对象创建时相同,是通过将视图类作为参数注册进文档模板中,在应用程序创建文档或创建视图时创建的。在应用程序 Data1 的应用类 CData1App 中的 InitInstance 函数中,可以找到创建和注册文档模板对象的代码:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CData1Doc),          //注册文档类
    RUNTIME_CLASS(CMainFrame),         //注册主框架类
    RUNTIME_CLASS(CData1View));        //注册视图类
AddDocTemplate(pDocTemplate);
```

以上创建的是一个单文档模板(CSingleDocTemplate),其资源(包括菜单、字符串、图标等)

采用的是资源 IDR _ MAINFRAME，注册的文档类是 CData1Doc 类，注册的框架类 CMainFrame（如果是在多文档下，则采用子框架类），注册的视图类就是以上讨论的表单视图类 CData1View。

可以在多文档情况下创建多种视图，所谓"多种视图"，指在同一个应用程序中，可以创建多个不同性质的视图。如在第二篇讨论实现的矢量图形系统 Draw 中，采用的是普通视图，可以在这个视图基础上，再创建一个用来浏览和管理记录集合的表单视图，使这两个视图同时存在。在第 14 章 14.1 节，将讨论创建多种视图的方法，并在第二篇讨论的矢量图形系统 Draw 基础上，再创建一个管理数据库记录集合的表单视图，达到在不同的视图中分别管理矢量图形和数据库记录集合的功能，为 GIS 系统的开发奠定基础。

12.3.2　CRecordView 视图对象与记录集合对象的联系

在数据库应用程序中，由 CRecordView 派生的视图类对象，最标准的功能就是对记录集合中当前记录的内容进行交互式管理。也就是说，通过在 CRecordView 视图类所使用的对话框资源中创建各种对话框控制，并建立这些控制与记录集合对象当前记录中的各个字段间的联系，实现利用视图中的各个控制来管理记录集合的当前记录。为了实现这个功能，一方面需要在视图所采用的对话框中建立各种控制，组成交互式编辑记录集合当前记录的界面；另一方面，需要在视图对象中能够得到和操纵记录集合对象。对于第一个方面，可以通过 ResourceView 及有关的辅助工具来实现。对于第二个方面，则通过文档/视图结构中的文档和视图间的相互关系来实现。

在开发基于 ODBC 的数据库应用程序时，记录集合是通过 CRecordSet 派生类对象来管理的（参见 12.2 节），在 CRcordView 视图对象中，则需要与具体的 CRecordSet 派生类对象建立联系。以下是在应用程序 Data1 中建立联系的过程：

① 建立一个 CRecordSet 派生类对象。在 Data1 应用程序中，定义了一个 CRecordSet 派生类 CData1Set 来管理从 ODBC 数据源创建的记录集合。为了实现具体的管理功能，在文档类 CData1Doc 中定义了一个 CData1Set 对象，在文档类 CData1Doc 的头文件 data1doc.h 中，可以找到 CData1Set 对象的定义代码：

```
public:
    CData1Set m _ data1Set;
```

② 在视图类 CData1View 中定义了一个 CData1Set 类指针，并把指针指向所属文档对象中定义的 CData1Set 对象。在视图类 CData1View 中，可以找到类指针的定义：

```
public:
    //{{AFX _ DATA(CData1View)
    enum{ IDD = IDD _ DATA1 _ FORM };
    CData1Set * m _ pSet;
    // NOTE: the ClassWizard will add data members here
//}}AFX _ DATA
```

m _ pSet 就是一个 CData1Set 类指针，这个指针在视图类 CData1View 的构造函数中被初始化：

```
CData1View::CData1View()
    : CRecordView(CData1View::IDD)
{
    //{{AFX_DATA_INIT(CData1View)
        // NOTE: the ClassWizard will add member initialization here
    m_pSet = NULL;
    //}}AFX_DATA_INIT
    // TODO: add construction code here
}
```

m_pSet 指针在视图类 CData1View 的初始化函数 OnInitialUpdate 中，被指向当前文档对象中的 CData1Set 对象 m_data1Set：

```
void CData1View::OnInitialUpdate()
{
  m_pSet = &GetDocument()->m_data1Set;//将指针 m_pSet 指向文档对象中的 m_data1Set 对象
  CRecordView::OnInitialUpdate();              //调用基类的初始化函数
  //以下调用主框架对象的 Recalcayout 函数重新布置程序界面中的各部分
  GetParentFrame()->RecalcLayout();
  ResizeParentToFit();
}
```

通过以上的两个步骤，我们就建立视图对象与记录集合对象间的联系，最终达到利用 CData1View 视图对象来交互式管理记录集合当前记录的目的。视图中的各控制与记录集合中各个字段间相互关系的建立可参见 12.3.4 小节。

12.3.3 MFC ClassWizard 管理 CRecordView 派生类

CRecordView 派生类可作为 MFC 应用程序的视图类来组成表单视图。为了实现利用表单视图交互式管理记录集合当前记录的功能，需要建立 CRecordView 派生类所采用的对话框资源中的各个控制与记录集合的各个字段间的联系。可以采用 ClassWizard 来管理这些联系，所以，在 12.3.4 小节中详细剖析视图中的控制与记录集合的字段进行数据交换的功能实现之前，在本小节中，首先对用 ClassWizard 管理 CRecordView 派生类的一些操作进行介绍和分析。

(1) 管理消息处理函数和重载的虚函数

在 VC++ 开发平台上，打开 12.1.2 小节产生的应用程序 Data1，启动 ClassWizard 进入 ClassWizard 工作界面(参见第 2 章 2.3 节)，在"Class name"列表框中选择类 CData1View，在对话框顶部选择"Message Maps"标签，就进入图 12-6 所示的对话框，在这个对话框中可以对命令消息处理函数、控制消息处理函数以及重载虚函数进行管理，具体的操作步骤可参见 2.3 节。控制消息处理函数指的是针对 CData1View 类所使用的对话框资源 IDD_DATA1_FORM 中的各种控制建立的消息处理函数，用 ClassWizard 管理这些控制消息处理函数的方法和步骤与管理菜单的命令消息处理函数相同。

(2) 管理映射变量

ClassWizard 管理 CRecordView 派生类中的映射变量的方法和步骤，是与管理对话框类时的映射变量是相同的。CRecordView 类是普通视图类和对话框类的混合体，对于其中的各种对话框控制，可以建立各种映射变量来实现这些控制与变量间的数据交换。在 CRecordView 类中建立映射变量时可以分为以下两种情况：

① 直接在 CRecordView 派生类 CData1View 中建立映射变量，并建立对话框控制与变量之间的数据交换机制，这种类型映射变量的管理方法及实现过程分析读者可以参见 2.3.2 小节。

② 建立 CRecordView 派生类 CData1View 中的对话框控制与外部引用类对象中的映射变量的数据交换。CRecordView 视图的最终目的，就是通过 CRecordView 派生类 CData1View 中所使用的对话框资源中的各种控制，与 CRecordSet 派生类对象中的映射变量(与记录集合中字段绑定的映射变量)建立联系，最终达到通过 CRecordView 视图中的对话框控制来交互式管理记录集合当前记录中的字段。

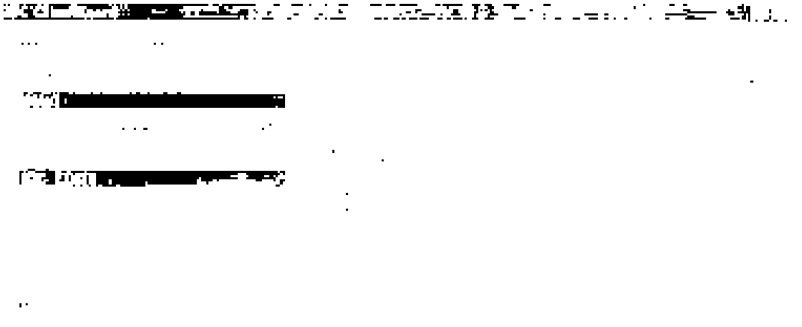


图 12-6　ClassWizard 对话框 Message Maps 界面

以下是利用 ClassWizard 建立这个联系的操作过程：

① 在应用程序 Data1 中，视图类 CData1View 所使用的对话框资源 IDD _ DATA1 _ FORM 中不包含任何控制，为了管理记录集合的当前记录，需要首先在对话框资源 IDD _ DATA1 _ FORM 中增加控制。为讨论方便起见，只在 IDD _ DATA1 _ FORM 对话框资源中增加一个编辑框 IDC _ EDIT1，得到如图 12-7 所示的对话框。

② 进入 ClassWizard 对话框，在对话框界面中选择“Member Variables”标签，在“Class Name”列表框中选择类 CData1View，就会进入图 12-8 所示的对话框界面。

③ 在图 12-8 所示对话框的列表框中选择控制 IDC _ EDIT1，按中“Add Viarable...”按钮，就会进入到如图 12-9 所示的对话框。

图 12-7　IDD _ DATA1 _ FORM 对话框资源

④ 在图 12-9 所示的对话框中，在“Member Variable Name”列表框选择一个外部的变量 m _ pSet - > m _ city _ name，保持“Catalog”和“Variable Type”中的默认选项，按“OK”按钮退出图 12-9 所示对话框，再按“OK”按钮退出图 12-8 所示对话框，就建立了对话框控制 IDC _ EDIT1 与 m _ pSet 指向的记录集合对象中的映射变量 m _ city _ name 的联系。

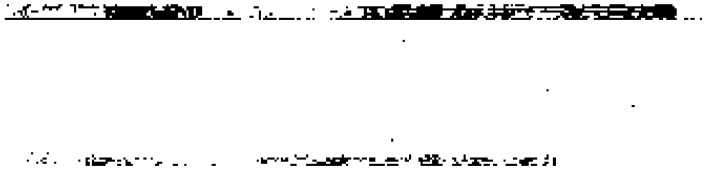


图 12-8　ClassWiard 对话框 Member Variable 界面

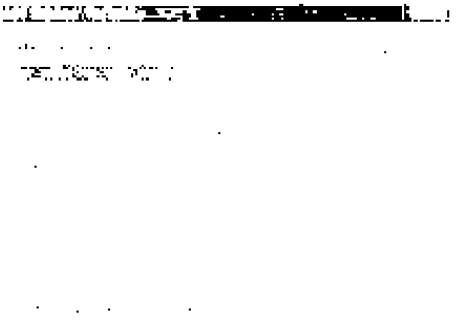


图 12-9　增加映射变量对话框

以上的操作过程，对于源程序代码来说，只是在 CData1View 类的 DoDataExchange 函数中增加了一行宏代码：

```
void CData1View::DoDataExchange(CDataExchange * pDX)
{
    CRecordView::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CData1View)
    DDX_FieldText(pDX, IDC_EDIT1, m_pSet->m_city_name, m_pSet);
    //}}AFX_DATA_MAP
}
```

DDX_FieldText 宏用来完成对话框控制 IDC_EDIT1 与指针 m_pSet 所指向的 CData1Set 对象中的映射变量 m_city_name 的数据交换。

(3) 设置外部引用对象

在(2)中建立映射变量时，在图 12-9 所示的对话框中，在“Member Variable Name”列表框中列出了 CData1Set 类中的映射变量。这是因为在 CData1View 类中设置了对 CData1Set 类的外部引用指针 m_pSet。对于这个指针的设置，既可以手工修改代码来完成，也可以通过 ClassWizard 完成。以下介绍用 ClassWizard 来管理这个外部引用指针的方法。

在图 12-8 所示的操作界面中，在对话框的顶部选择“Class Info”标签，在“Class Type”列表框中选择类 CData1View，就会进入图 12-10 所示的对话框。在对话框下部的“Advanced Options”边框内，列出的是 CData1View 类的外部类指针的设置选项，在“Foreign Class”编辑框内列出的是外部引用的类 CData1Set，在“Foreign Variable”编辑框内列出的是类指针变量的名称 m_pSet。现在做一个简单的试验，在“Foreign Class”列表框中选择“None”选项，即不选择外部类引用，按“OK”按钮退出 ClassWizard，再进入图 12-10 所示的对话框时，在“Member Variable Name”列表框中，将找不到 CData1Set 类的映射变量。

现在再进入图 12-10 所示的对话框，在“Foreign Class”列表框中选择 CData1Set 类，在“Foreign Variable”编辑框中输入变量名称 m_pSet，按“OK”按钮退出 ClassWizard，就又可以在 CData1View 类中增加一个外部引用类，此增加过程对代码的修改如下：

① 在 CData1View 类中定义了一个 CData1Set 类指针：

```
public:
    //{{AFX_DATA(CData1View)
    enum { IDD = IDD_DATA1_FORM };
```

```
    CDataSet * m_pSet;
    //{{AFX_DATA
```

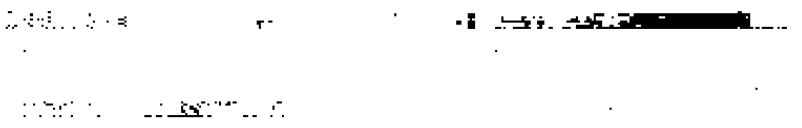


图 12-10　ClassWizard 对话框 Class Info 界面

② 在 CDataView 类的构造函数中，加入了对类指针的初始化代码：

```
CDataView::CDataView()
    : CRecordView(CDataView::IDD)
{
    //{{AFX_DATA_INIT(CDataView)
    m_pSet = NULL;
    //}}AFX_DATA_INIT
    // TODO: add construction code here
}
```

用 ClassWizard 增加一个类的外部类引用时，只能对程序代码进行以上两个步骤的修改。而只设定一个类指针而不把这个指针指向具体的 CDataSet 对象，对于完成对记录集合的管理功能是没有任何作用的。所以，真正进行增加外部引用类的操作时，还需要手工加入代码来初始化类指针。从 CDataView 类的初始化函数 OnInitialUpdate 中，可以找到对类指针进行初始以指向实际 CDataSet 对象的代码：

```
void CDataView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_dataSet;
    m_pSet->Open();
    CRecordView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
```

```
    ResizeParentToFit();
}
```

从以上黑体代码可以看出，m_pSet 指向了当前文档对象中的一个成员变量 m_data1Set。m_data1Set 是在文档类中定义的 CData1Set 对象：

```
public:
    CData1Set m_data1Set;
```

(4) 数据库应用程序管理记录集合的主要步骤

通过以上的讨论，读者可整理出应用程序 Data1 管理记录集合的实现步骤：

① 建立 CRecordSet 派生类 CData1Set 用来创建和管理记录集合对象。

② 在文档类中 CDrawDoc 中建立了一个 CData1Set 对象，来具体管理一个记录集合。

③ 建立一个 CRecordView 派生类 CData1View，这个类用做应用程序的视图类，在这个类中建立一个 CData1Set 类的外部引用指针 m_pSet，将这个指针指向当前文档对象中的 CData1Set 对象 m_data1Set。

④ 建立视图类 CData1View 使用的对话框资源中的控制与 m_pSet 指向的记录集合对象中的映射变量之间的数据交换，实现对话框控制与记录集合当前记录的各字段间的数据交换。

以上是应用程序 Data1 进行数据库应用程序设计时的主要步骤。在 12.4 节中将更详细地讨论利用这个程序框架完成一些数据库管理基本功能的实现步骤和方法，从中读者可更深刻地领会到这种数据库应用程序框架的精髓，达到最大限度利用 MFC 的既有程序框架和功能，灵活组织数据库应用程序设计的最终目的。

12.3.4　视图控制与记录集合数据交换过程剖析

在应用程序 Data1 的视图类 CData1View 中，能够实现视图中的对话框控制与记录集合当前记录各个字段的数据交换。数据交换是通过 CData1View 类的 DoDataExchange 函数来实现的，DoDataExchange 函数实现视图中的对话框控制与记录集合当前记录的各个字段间的数据交换。读者从应用程序代码中不可能知道 DoDataExchange 函数是何时调用的，又是如何调用的。这一小节，将剖析这个过程的实现步骤。

(1) DoDataExchange 函数的调用执行

在对话框应用程序中，有一个在窗口类中定义的虚函数 UpdateData。这个函数有一个参数，以 FALSE 为参数调用这个函数时，UpdateData 函数会调用 DoDataExchange 函数完成将映射变量的内容转变并填写到控制中去；而当以 TRUE 为参数调用 UpdateData 函数时，会执行相反的操作，将控制中的内容转变到映射变量中。CRecordView 类是一个综合了对话框和普通视图两者功能和特点的类。在 CRecordView 类中，也可以以不同的参数调用函数 UpdateData，实现以不同的参数调用 DoDataExchange 函数，完成对话框控制与记录集合对象的映射变量间的双向数据交换。以下是定义在 CRecordView 类中的 UpdateData 函数的代码(作者加入了注释信息)：

```
BOOL CWnd::UpdateData(BOOL bSaveAndValidate)
```

```
{
    ASSERT(::IsWindow(m_hWnd)); // calling UpdateData before DoModal?
    //以下以当前窗口指针和函数的参数来初始化一个 CDataExchange 对象 dx
    CDataExchange dx(this, bSaveAndValidate);
    // prevent control notifications from being dispatched during UpdateData
    _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
    HWND hWndOldLockout = pThreadState->m_hLockoutNotifyWindow;
    ASSERT(hWndOldLockout != m_hWnd);   // must not recurse
    pThreadState->m_hLockoutNotifyWindow = m_hWnd;
    BOOL bOK = FALSE;   // assume failure
    TRY
    {
    DoDataExchange(&dx);   //以 &dx 为参数调用 DoDataExchange 函数
        bOK = TRUE;   // it worked
    }
    CATCH(CUserException, e)
    {
        // validation failed - user already alerted, fall through
        ASSERT(!bOK);
        // Note: DELETE_EXCEPTION_(e) not required
    }
    AND_CATCH_ALL(e)
    {
        // validation failed due to OOM or other resource failure
        e->ReportError(MB_ICONEXCLAMATION, AFX_IDP_INTERNAL_FAILURE);
        ASSERT(!bOK);
        DELETE_EXCEPTION(e);
    }
    END_CATCH_ALL
    pThreadState->m_hLockoutNotifyWindow = hWndOldLockout;
    return bOK;
}
```

从以上代码可以看出，在 UpdateData 函数中，首先，以当前窗口指针和函数的参数为参数定义了一个 DataExchange 对象 dx：

```
CDataExchange dx(this, bSaveAndValidate);
```

然后，以指向对象 dx 的指针为参数调用了 DoDataExchange 函数，来实现对话框控制与映射变量间的数据交换：

```
DoDataExchange(&dx);
```

所以，以不同的参数(TURE 和 FALSE)为参数调用 UpdateData 函数，可以分别在两个方向上实现对话框控制与记录集合的映射变量间的数据交换。

(2) 对话框控制与记录集合中的映射变量间的数据交换

在 CData1View 类的 DoDataExchange 函数中,对话框控制与 CData1Set 类的映射变量间的数据交换是通过各个宏 DDX _ FieldText 来实现的。以下宏代码实现对话框控制 IDC _ EDIT1 与 CData1Set 类中的映射变量 m _ city _ name 间的数据交换:

```
DDX _ FieldText(pDX, IDC _ EDIT1, m _ pSet - > m _ city _ name, m _ pSet);
```

这个过程的实现步骤较为复杂,而且在实现过程中不止一次地调用了 CData1Set 类的 DoDataExchange 函数,具体的实现过程在此不再具体叙述。以下分别以参数 FALSE 和 TRUE 调用 UpdateData 函数时,以理想化的步骤,讨论对话框控制与记录集合当前记录的字段间的数据交换。在 CRecordView 派生类 CData1View 中,以 FALSE 为参数调用 UpdateData 函数时,实现如下的功能:

① 调用 CData1Set 类的 DoDataExchange 函数,将记录集合当前记录的各个字段的内容转变成 CData1Set 类中绑定这些字段的映射变量。

② 调用 CData1View 类的 DoDataExchange 函数,将 CData1Set 类中绑定记录集合字段的映射变量转变并填写到视图中相应的对话框控制中。

通过以上两个步骤,就完成了将记录集合当前记录中的字段内容转变成了视图中的对话框控制状态,即显示在了对话框控制中。

而在以 TRUE 为参数调用 UpdateData 函数时,实现如下的功能:

① 调用 CDtata1View 类的 DoDataExchange 函数,将视图中的对话框控制中的内容转变成了 CData1Set 类的映射变量。

② 调用 CData1Set 类的 DoDataExchange 函数,将 CData1Set 类中的映射变量转变成记录集合当前记录中对应的字段。

通过以上两个步骤,就实现了将视图中的对话框控制中的内容转变成记录集合当前记录中的对应字段,即达到了用视图中的对话框控制,来修改记录集合当前记录中对应字段值的功能。

通过以上的步骤,读者可以比较容易地理解视图中的对话框控制和记录集合当前记录的字段间的数据交换过程。这个交换过程的实际代码复杂的多,读者一般也没有必要具体去分析实际的实现过程。

本节讨论了 CRecordView 表单视图与记录集合对象间建立联系并实现数据交换的步骤。在对话框程序设计中,同样可以采用以上类似的步骤建立对话框与记录集合对象间的联系,实现利用对话框来交互管理记录集合当前记录,即通过对话框来实现对记录集合的交互编辑功

能。在第 15 章 15.1 节中,将讨论实现利用对话框对记录集合记录的交互编辑功能。

12.4 基本数据库管理功能实现剖析

在 12.1.2 小节,用 AppWizard 产生了一个数据库应用程序 Data1。在应用程序 Data1 中,包含了一些基本的数据库管理功能。本节将剖析这些管理功能的实现过程,并组织实现其他的一些基本管理功能,为灵活的组织数据库应用程序设计作好技术准备。

12.4.1 浏览功能

应用程序 Data1 中包含了大量的既有功能,本节将对这些既有功能进行分析,讨论数据记录浏览和编辑视图的实现。

(1) 实现数据记录编辑视图

Data1 应用程序采用的是表单视图。对表单视图类 CData1View 所采用的对话框资源进行修改,在其中加入大量的控制,并建立各种控制与记录集合对象各个字段间的联系,就可以实现一个简单的数据记录浏览视图。其具体实现步骤如下:

① 编辑对话框资源。表单视图类 CData1View 采用的对话框资源是 IDD _ DATA1 _ FORM,通过 ResourceView 及其编辑工具对这个对话框资源进行可视化编辑。编辑后的对话框资源如图 12-11 所示,在资源文件 draw.rc 中资源描述为:

图 12-11 IDD _ DATA1 _ FORM 对话框资源

```
IDD_DATA1_FORM DIALOG DISCARDABLE 0, 0, 242, 102
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "城市数据库记录编辑窗口"
FONT 10, "System"
BEGIN
    LTEXT           "标识号",IDC_STATIC,7,19,27,10
    EDITTEXT        IDC_EDIT1,36,18,40,12,ES_AUTOHSCROLL
    LTEXT           "城市名称",IDC_STATIC,108,18,38,12
    LTEXT           "人口数量",IDC_STATIC,6,40,38,12
    LTEXT           "城市面积",IDC_STATIC,108,38,38,12
    LTEXT           "工农业产值",IDC_STATIC,7,59,40,12
```

```
    LTEXT           "交通状况",IDC_STATIC,107,60,38,10
    LTEXT           "环境状况",IDC_STATIC,7,78,44,10
    EDITTEXT        IDC_EDIT2,149,17,69,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT3,52,38,49,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT4,149,38,54,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT5,53,58,48,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT6,148,58,57,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT7,53,77,47,12,ES_AUTOHSCROLL
END
```

② 按照 12.3.4 小节中介绍的在 CRecordView 派生类中增加映射变量的方法，在 CData1View 类中增加映射变量，建立对话框资源 IDD_DATA1_FORM 中的控制与 CData1Set 类中的变量之间的联系。操作完毕，CData1View 类的 DoDataExchange 函数实现代码修改如下：

```
void CData1View::DoDataExchange(CDataExchange * pDX)
{
    CRecordView::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CData1View)
    DDX_FieldText(pDX, IDC_EDIT1, m_pSet->m_only_id, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT2, m_pSet->m_city_name, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT3, m_pSet->m_area, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT4, m_pSet->m_population, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT5, m_pSet->m_industry, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT6, m_pSet->m_translate, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT7, m_pSet->m_enviroment, m_pSet);
    //}}AFX_DATA_MAP
}
```

对于在 12.1.2 小节创建的应用程序 Data1，进行了步骤①的操作后，编译后运行，就会进入到图 12-12 所示的运行界面。在这个运行界面上，通过运行“记录(R)”菜单下的菜单项(或者通过工具条上的按钮)，可以对记录集合进行浏览和修改等操作功能。对于程序初学者来说，因为在应用程序 Data1 的代码中找不到相应的执行代码，所以对这些功能的具体实现过程很困惑，以下对这些功能的实现过程进行剖析。

(2) 滚动浏览记录集合

在图 12-12 所示的应用程序界面上，在菜单“记录(R)”下有四个菜单项。以下是这四个菜单项及其对应的 ID 号。

- 第一个记录　　ID_RECORD_FIRST
- 前一个记录　　ID_RECORD_PREV
- 下一个记录　　ID_RECORD_NEXT
- 最后一个记录　　ID_RECORD_LAST

在工具条中有四个与这些菜单项相对应的工具条按钮。当通过菜单项或工具条按钮来执

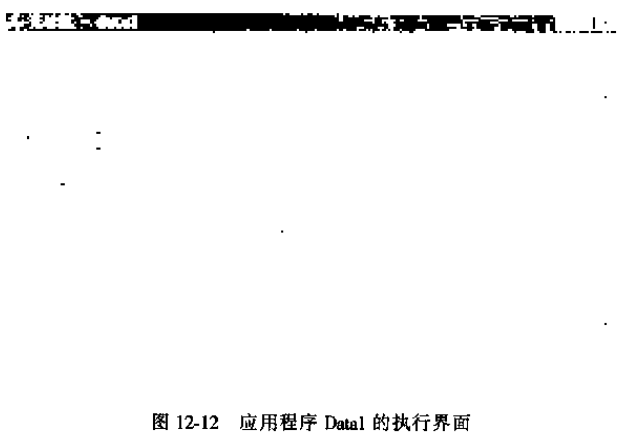


图 12-12　应用程序 Data1 的执行界面

行这四个菜单项时，从视图中显示的内容可以看出，记录集合中记录指针指向的记录（即当前记录）发生了变化，可以通过这四个菜单实现对记录集合的浏览。对于这些操作功能，在应用程序 Data1 中找不到任何与其相关的代码，以下对这些菜单功能的执行过程进行分析。

在视图类 CRecordView 中建立了针对以上四个菜单项 ID 号的消息处理函数，当执行这四个菜单项时，发送的 COMMAND 消息被主框架窗口获得，经过多次调用后，调用了 CRecordView 类中的 OnMove 函数，OnMove 函数的参数是调用这个函数的命令消息 ID 号，以上讨论的四个操作功能，最终都是以不同参数调用了 OnMove 函数，来完成在记录集合中移动记录指针的功能。以下是加入了详细注释后的 OnMove 函数的实现代码：

```
BOOL CRecordView::OnMove(UINT nIDMoveCommand)
{
    CRecordset* pSet = OnGetRecordset();        //得到当前的 CRecordSet 派生类对象指针
    //如果 pSet 指向的记录集合进行修改并且当前记录没有被删除
    if (pSet->CanUpdate() && !pSet->IsDeleted())
    {
        pSet->Edit();           //开始修改记录集合的当前记录
        //以默认参数 TRUE 调用 UpdateData 函数，以视图控制的内容修改记录集合当前记录的字段
        if (!UpdateData())
            return TRUE;
        pSet->Update();         //对记录集合的修改影响到数据源，即对数据源进行修改
    }
    switch (nIDMoveCommand)     //根据 OnMove 的参数值进行不同的操作
    {
    case ID_RECORD_PREV:        //如果是移动到上一记录
        pSet->MovePrev();       //将记录指针移动到下一个记录
        if (!pSet->IsBOF())     //如果没有超出记录集合的范围
            break;
    case ID_RECORD_FIRST:       //如果移动到第一条记录
        pSet->MoveFirst();      //将记录指针移动到第一条记录
```

```
            break;
        case ID_RECORD_NEXT:    //如果是移动到下一条记录
            pSet->MoveNext();    //将记录指针移动到下一条记录
            if (!pSet->IsEOF())  //如果没有超出记录集合的范围
                break;
            if (!pSet->CanScroll())//如果记录集合对象不能进行滚动
            {
                // clear out screen since we're sitting on EOF
                pSet->SetFieldNull(NULL);    //清空记录
                break;
            }
        case ID_RECORD_LAST:    //如果是移动到最后一条记录
            pSet->MoveLast();    //将记录指针移动到最后的记录
            break;
        default:
            // Unexpected case value
            ASSERT(FALSE);
        }
        // 以下以 FALSE 为参数调用 UpdateData 函数,将记录集合当前记录的内容显示在各种控制中
        UpdateData(FALSE);
        return TRUE;
    }
```

OnMove 函数的主要操作步骤如下：

① 以 TRUE 为参数调用 UpdateData 函数，将视图中对话框控制内的内容转换到记录集合当前记录的各字段中，即在进行对记录集合的滚动操作前，首先将在视图各控制中对当前记录的修改保存到了记录集合中。UpdateData 函数的功能见 12.3.4 小节。

② 根据不同的参数(即在按中了图 12-12 菜单“记录(R)”中的不同菜单项来调用 OnMove 函数)，移动记录集合中的记录指针，改变记录集合中记录指针当前指向的记录。

③ 以 FALSE 为参数调用 UpdateData 函数，将记录集合当前记录的内容转换到视图的各对话框控制中去，即以视图中的控制来浏览记录集合当前记录的内容。

对于应用程序 Data1 菜单“记录(R)”下的菜单项与消息处理函数 OnMove 间的消息映射，在 CRecordView 类的影射机制中来实现：

```
BEGIN_MESSAGE_MAP(CRecordView, CFormView)
    //{{AFX_MSG_MAP(CRecordView)
        // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
    ON_COMMAND_EX(ID_RECORD_FIRST, OnMove)      //显示第一条记录
    ON_UPDATE_COMMAND_UI(ID_RECORD_FIRST, OnUpdateRecordFirst)
    ON_COMMAND_EX(ID_RECORD_PREV, OnMove)       //显示上一条记录
    ON_UPDATE_COMMAND_UI(ID_RECORD_PREV, OnUpdateRecordPrev)
    ON_COMMAND_EX(ID_RECORD_NEXT, OnMove)       //显示下一条记录
```

```
    ON_UPDATE_COMMAND_UI(ID_RECORD_NEXT, OnUpdateRecordNext)
    ON_COMMAND_EX(ID_RECORD_LAST, OnMove)        //显示最后一条记录
    ON_UPDATE_COMMAND_UI(ID_RECORD_LAST, OnUpdateRecordLast)
END_MESSAGE_MAP()
```

各个菜单项的 ID 号，由 ON_COMMAND_EX 宏来实现消息 ID 号与消息处理函数 OnMove 的映射，与 ON_COMMAND 宏不同的是，ON_COMMAND_EX 宏可以将消息的 ID 号作为参数传递给消息处理函数 OnMove，实现利用同一个消息处理函数映射处理多个不同的消息，根据消息 ID 号的不同在 OnMove 函数中进行不同的操作。

(3) 应用程序用户交互对象的显示状态控制

在运行应用程序 Data1 时，读者可以注意到，当用菜单项或工具条对记录集合进行滚动操作时，根据记录集合当前记录的不同，菜单项或工具条的显示状态会相应发生变化。如当记录指针指向记录集合第一条记录时(即第一条记录是记录集合的当前记录)，“第一条记录”和“上一条记录”菜单项或工具条按钮就会变暗，处于不能执行状态。这些功能是通过 UPDATE_COMMAND_UI 消息处理函数来实现的，UPDATE_COMMAND_UI 消息在交互操作工具(如菜单、工具条等)在显示前被发送，通过给这些交互操作对象建立 UPDATE_COMMAND_UI 消息处理函数，就可以达到控制这些交互操作对象显示和执行状态的目的。

在(2)中列出的视图类 CRecordView 的映射机制中可以看到，对于应用程序 Data1“记录(R)”菜单下的 4 个菜单项，在 CRecordView 类中有对应的 UPDATE_COMMAND_UI 消息处理函数。在 CRecordView 类中可以找到这 4 个消息处理函数的实现代码(加入了中文注释)：

```
//菜单“第一条记录”(ID 号 ID_RECORD_FIRSAT)的消息处理函数
void CRecordView::OnUpdateRecordFirst(CCmdUI * pCmdUI)
{
    CRecordset * prs = OnGetRecordset();        //得到指向记录集合对象的指针
    //如果记录集合处于打开状态，记录集合允许滚动，记录集合中的记录指针没有指向
    //记录集合范围外，并且当前记录指针不是指向第一条记录时，菜单处于可执行状态
    pCmdUI->Enable(prs->IsOpen() && prs->CanScroll() &&
        !(prs->IsEOF() && prs->IsBOF()) && !IsOnFirstRecord());
}

//菜单“上一条记录”(ID 号 ID_RECORD_PREV)的消息处理函数
void CRecordView::OnUpdateRecordPrev(CCmdUI * pCmdUI)
{
    //“上一条记录”菜单项的显示状态与“第一条记录”菜单项显示状态相同
    CRecordView::OnUpdateRecordFirst(pCmdUI); //调 OnUpdateRecordFirst 处理函数
}

//菜单“下一条记录”(ID 号 ID_RECORD_NEXT)的消息处理函数
void CRecordView::OnUpdateRecordNext(CCmdUI * pCmdUI)
{
```

```
    CRecordset * prs = OnGetRecordset();      //得到指向记录集合对象的指针
    //如果记录集合对象处于打开状态,记录指针没有指向记录集合范围外,且记录指针不是
    //指向记录集合的最后一条记录时,菜单处于可执行状态
    pCmdUI->Enable(prs->IsOpen() && !(prs->IsEOF() && prs->IsBOF())
        && !IsOnLastRecord());
}

//菜单"最后一条记录"(ID 号 ID_RECORD_LAST)的消息处理函数
void CRecordView::OnUpdateRecordLast(CCmdUI * pCmdUI)
{

    CRecordset * prs = OnGetRecordset();      //得到指向记录集合对象的指针
    //如果记录集合处于打开状态,且记录集合允许滚动,记录指针没有指向记录集合范围外,
    //且记录指针不是指向记录集合的最后一条记录时,菜单处于可执行状态
    pCmdUI->Enable(prs->IsOpen() && prs->CanScroll() &&
    !(prs->IsEOF() && prs->IsBOF()) && !IsOnLastRecord());
}
```

以上函数中调用的函数 IsOnFirstRecord 和 IsOnLastRecord，分别用来检查记录指针是否指向记录集合的第一条记录或最后一条记录。这两个函数在 CRecordView 类中的实现代码如下：

```
BOOL CRecordView::IsOnFirstRecord()
{
    ASSERT_VALID(this);
    CRecordsetStatus status;
    OnGetRecordset()->GetStatus(status);          //得到记录集合对象的状态
    return status.m_lCurrentRecord == 0;
}
```

如果记录指针指向记录集合的第一条记录，IsOnFirstRecord 函数返回 TRUE，否则返回 FALSE。

```
BOOL CRecordView::IsOnLastRecord()
{
    ASSERT_VALID(this);
    CRecordset * pRecordset = OnGetRecordset();      //得到指向记录集合对象的指针
    CRecordsetStatus status;
    pRecordset->GetStatus(status);                   //得到记录集合对象的状态
    if (! status.m_bRecordCountFinal)
        return FALSE;
    return ((status.m_lCurrentRecord + 1 == pRecordset->GetRecordCount()));
}
```

如果记录指针指向记录集合的最后一条记录，IsOnLastRecord 函数返回 TRUE，否则返回

FALSE。

12.4.2　增加一个操作菜单

在数据库应用程序 Data1 中，实现了对记录集合的滚动浏览功能。在浏览的过程中，能够对记录集合的当前记录进行修改。以下各小节，将讨论在应用程序 Data1 中实现其他的一些数据库操作功能。在具体讨论这些数据库操作功能之前，首先在应用程序中增加一个操作菜单，并为各菜单项建立消息处理函数。

(1) 增加菜单资源

为具体实现对于记录集合的增加、修改和删除记录功能，用 ResourceView 及其可视化编辑工具在应用程序 Data1 的菜单资源 IDR_MAINFRAME 中增加一个菜单“操作(O)”，在这个菜单中增加 5 个菜单项“增加记录”、“修改记录”、“删除记录”、“过滤操作”、“排序操作”(如图 12-13 所示)。操作完毕，在资源文件 data1.rc 中增加了以下代码：

图 12-13　IDR_MAINFRAME 对话框资源

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    ......//省略以上代码
    POPUP "操作(&O)"
    BEGIN
        MENUITEM "增加记录(&A)",          ID_RECORD_ADD
        MENUITEM "修改记录(&X)",          ID_RECORD_MODIFY
        MENUITEM "删除记录(&D)",          ID_RECORD_DELETE
        MENUITEM "过滤操作(&F)",          ID_RECORD_FILTER
        MENUITEM "排序操作(&S)",          ID_RECORD_SORT
    END
    ......//省略以下代码
END
```

在 resource.h 文件中，加入了资源 ID 号的定义：

```
#define ID_RECORD_ADD              32771
#define ID_RECORD_MODIFY           32772
#define ID_RECORD_DELETE           32773
```

```
# define ID_RECORD_FILTER        32774
# define ID_RECORD_SORT          32775
```

(2) 增加消息处理函数

对于这五个菜单项,用 ClassWizard 在视图类 CData1View 中分别为其增加消息处理函数。增加完毕,应用程序 Data1 的代码修改如下:

① 在视图类 CData1View 中增加了消息处理函数的定义:

```
//{{AFX_MSG(CData1View)
afx_msg void OnRecordAdd();
afx_msg void OnRecordModify();
afx_msg void OnRecordDelete();
afx_msg void OnRecordFilter();
afx_msg void OnRecordSort();
//}}AFX_MSG
```

② 在实现文件 data1view.cpp 中,增加了消息处理函数的映射机制:

```
EGIN_MESSAGE_MAP(CData1View, CRecordView)
    //{{AFX_MSG_MAP(CData1View)
    ON_COMMAND(ID_RECORD_ADD, OnRecordAdd)
    ON_COMMAND(ID_RECORD_MODIFY, OnRecordModify)
    ON_COMMAND(ID_RECORD_DELETE, OnRecordDelete)
    ON_COMMAND(ID_RECORD_FILTER, OnRecordFilter)
    ON_COMMAND(ID_RECORD_SORT, OnRecordSort)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CRecordView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

③ 在实现文件 data1view.cpp 中,增加了空的函数实现代码:

```
void CData1View::OnRecordAdd()
{
    // TODO: Add your command handler code here

}
void CData1View::OnRecordModify()
{
    // TODO: Add your command handler code here

}
```

```
void CData1View::OnRecordDelete()
{
    // TODO: Add your command handler code here

}
void CData1View::OnRecordFilter()
{
    // TODO: Add your command handler code here

}
void CData1View::OnRecordSort()
{
    // TODO: Add your command handler code here

}
```

以下各小节将对这些消息处理函数进行修改，完成对记录集合的增加记录、修改记录和删除记录以及过滤、排序等操作功能。

12.4.3　记录编辑功能

本小节将在应用程序 Data1 中实现针对记录集合的增加、修改及删除记录功能。

(1) 增加记录

在 12.2.7 小节讨论 CRecordSet 类时讲过，在记录集合中增加一条记录，一般需要几个步骤：

① 调用 CRecordSet 类的 AddNew 函数，在记录集合中增加一条记录。

② 利用交互操作界面对增加记录的内容进行修改。

③ 调用 CRecordSet 类的 Update 函数，将对记录集合进行的修改影响到数据源中的实际数据表。

④ 重新创建记录集合。

为了实现增加功能，需要对菜单项“增加记录(A)”的消息处理函数 OnRecordAdd 进行修改，得到修改后的实现代码如下：

```
void CData1View::OnRecordAdd()
{
    m_pSet->AddNew();    //在记录集合中增加一条记录
    m_bAdding=TRUE;      //标识正在进行增加记录操作
    UpdateData(FALSE);   //将新增空记录的内容转移到视图中的控制中
}
```

变量 m_bAdding 用来标识正在进行增加记录的操作，当它的值为 TRUE 时，表示正在进行新增记录的操作(增加的新记录还没有保存)。这个变量在视图类 CData1View 中定义：

```
protected:
    BOOL m_bAdding;
```

OnRecordAdd 函数在记录集合的中增加一条空记录，并把空记录的内容转移到视图的对话框控制中，此时，就可以在视图的对话框控制中输入新增记录中各个字段的值。

因为只有在运行了 CRecordSet 类的 Update 函数后，才能把用 AddNew 函数增加的记录保存到数据源中（在数据源的数据表中也增加一条记录），所以，为了完成增加记录操作，还需要在应用程序中增加另外的代码。可以在“操作(O)”菜单中再增加一个菜单项“保存增加记录”，为这个菜单项增加一个消息处理函数 OnRecordSave，运行了“增加记录(A)”菜单项后，“保存增加记录”菜单可以被执行，完成增加记录过程的剩余操作。OnRecordSave 函数的代码可以如下定义：

```
void CData1View::OnRecordSave()
{
    if(m_bAdding)                    //如果正在进行增加新记录的操作
    {
        m_bAdding = FALSE;           //不再是进行增加新记录操作
        UpdateData(TRUE);            //以视图中对话框控制的内容修改记录集合当前记录
        if(m_pSet->CanUpdate())      //如果能够对数据源进行更新
            m_pSet->Update();        //更新数据源
        m_pSet->Requery();           //重新产生记录集合
        UpdateData(FALSE);           //以记录集合当前记录的内容修改视图中的对话框控制
        return TRUE;
    }
}
```

以上通过再设置一个菜单来完成增加记录操作的功能，这在流程控制上有一定问题，执行了“增加记录”菜单后，执行“记录(R)”下的各个菜单滚动记录集合时会造成流程不正确。这就需要在对记录集合进行滚动操作前，能够处理正在进行增加记录操作这种情况。

12.4.1 小节讨论了记录集合滚动操作时的消息处理函数 CRecordView::OnMove，可以通过在当前视图类 CData1View 中重载并修改这个消息处理函数，来处理系统正在进行增加记录操作这种情况。具体实现步骤如下：

① 在当前视图类 CData1View 中，重载 CRecordView 类的消息处理函数 OnMove（虚函数），重载虚函数的方法可以参见 2.5.5 小节。重载操作完成后，在视图类 CData1View 中增加了虚函数 OnMove 的重载定义：

```
//{{AFX_VIRTUAL(CData1View)
public:
virtual CRecordset* OnGetRecordset();
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
virtual BOOL OnMove(UINT nIDMoveCommand);
protected:
virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
virtual void OnInitialUpdate(); // called first time after construct
```

```
virtual BOOL OnPreparePrinting(CPrintInfo * pInfo);
virtual void OnBeginPrinting(CDC * pDC, CPrintInfo * pInfo);
virtual void OnEndPrinting(CDC * pDC, CPrintInfo * pInfo);
//{{AFX_VIRTUAL
```

在实现文件 data1view.cpp 中,增加了 OnMove 函数的实现代码:

```
BOOL CData1View::OnMove(UINT nIDMoveCommand)
{
    // TODO: Add your specialized code here and/or call the base class
    return CRecordView::OnMove(nIDMoveCommand);     //调用基类的 OnMove 函数
}
```

② 对函数 OnMove 进行修改,使它能够对增加的记录进行保存。以下是经过修改后的 OnMove 函数的实现代码:

```
BOOL CData1View::OnMove(UINT nIDMoveCommand)
{
    if(m_bAdding)                    //如果正在进行增加新记录的操作
    {
        m_bAdding = FALSE;           //不再是正在进行新记录状态
        UpdateData(TRUE);            //以视图中控制的内容修改记录集合的当前记录
        if(m_pSet->CanUpdate())      //如果能够对数据源进行更新
            m_pSet->Update();        //更新数据源
        m_pSet->Requery();           //重新创建记录集合
        UpdateData(FALSE);           //以记录集合中当前记录的内容修改视图中的对话框控制
        return TRUE;
    }
    else          //如果不是正在进行修改新记录的操作,调用基类的 OnMove 函数
        return CRecordView::OnMove(nIDMoveCommand);
}
```

(2) 修改记录

当运行"操作(O)"菜单下的"修改记录"菜单项时,需要能够以当前视图中对话框控制的内容来修改记录集合的当前记录,同时对数据源进行修改。在 12.2.7 小节讨论 CRecordSet 类时知道,对记录集合的当前记录进行修改时,一般需要几个步骤:

① 调用 CRecordSet 类的 Edit 函数使当前记录处于编辑修改状态。

② 利用交互操作界面对记录集合的当前记录进行修改。

③ 调用 CRecordSet 类的 Update 函数,将对记录集合进行的修改影响到数据源中的实际数据表。

④ 重新创建记录集合。

为了实现对记录集合中当前记录的修改,需要对 CData1View 类中的消息处理函数 OnRecordModify 进行修改,修改后的函数实现代码如下:

```
void CData1View::OnRecordModify()
{
    m_pSet->Edit();                    //使当前记录处于编辑状态
    UpdateData(TRUE);                  //以视图中对话框控制的内容修改记录集合的当前记录
    if(m_pSet->CanUpdate())            //如果能够对数据源进行更新
        m_pSet->Update();              //更新数据源
    m_pSet->Requery();                 //重新产生数据源
    UpdateData(FALSE);                 //以记录集合中当前记录的内容修改视图中的对话框控制
}
```

(3) 删除记录

当运行“操作(O)”菜单下的“删除记录”菜单时，需要能够完成对当前记录的删除功能。对消息处理函数 OnRecordDelete 修改如下：

```
void CData1View::OnRecordDelete()
{
    m_pSet->Delete();                  //删除记录集合的当前记录
    m_pSet->MoveNext();                //将记录指针移动到下一条记录
    if(m_pSet->IsEOF())                //如果超出记录集合的范围
        m_pSet->MoveLast();            //将记录指针移动到最后一条记录
    if(m_pSet->IsBOF())                //如果记录指针在第一条记录之前，即记录集合中无任何记录
        m_pSet->SetFieldNull(NULL);        //将记录设置为空
    UpdateData(FALSE);                     //以记录集合当前记录的内容修改视图中的控制
}
```

从以上代码中可以看出，在调用 CRecordSet 类的成员函数 Delete 删除记录集合的当前记录后，需要调整记录指针指向记录集合中有效的记录。

12.4.4　数据库过滤功能

在进行数据库应用程序设计时，在很多情况下，并不是选择数据源中数据表的所有记录组成记录集合，而只是选择其中符合某些条件的记录。这个产生记录集合时所选择的条件，称之为过滤条件，利用过滤条件有选择性的选择记录组成记录集合的过程，被称为过滤操作。在 12.2.4 小节，讨论了在 CRecordSet 类中组织过滤条件实现过滤操作的步骤，本小节介绍在应用程序 Data1 中实现过滤操作的几种方法。

(1) 实现固定条件的过滤操作

在 12.4.2 小节中，在应用程序 Data1 的视图类中建立了一个用于过滤操作的 OnRecordFilter 函数，以下对这个消息处理函数进行修改，实现一个固定条件的过滤操作：

```
void CData1View::OnRecordFilter()
{
```

```
    if(m_pSet->IsOpen())                    //如果记录集合对象处于打开状态,则关闭记录集合对象
        m_pSet->Close();
    m_pSet->m_strFilter = "Population > 100"  //设置记录集合对象的过滤条件
    m_pSet->Open();                          //重新打开记录集合对象
}
```

在 12.2.4 小节中的讨论可知,在用 Open 函数打开记录集合对象时,会将存储在 CRecordSet 类 m_strFilter 成员变量中的过滤条件 m_strFilter,与由函数 GetDefaultSQL 返回的 SQL 语句组织成一个 SQL 语句(存储在 CRecordSet 类中的成员变量 m_strSQL 中),利用这个 SQL 语句来产生记录集合。

在以上的 OnRecordFilter 函数中实现了一个固定条件的过滤功能,当运行"操作(O)"菜单下的"过滤操作"菜单项时,会根据过滤条件"population > 100"来重新创建一个记录集合。这是最简单最基本的一种过滤操作方式。

(2) 任意组织过滤条件的过滤功能

(1)中实现了对记录集合的固定条件过滤操作,在进行实际的数据库应用程序设计时,这种方式的作用是十分有限的。现在介绍通过动态组织过滤条件来实现过滤操作的方式。用这种方式组织过滤功能时,过滤操作的实现步骤与(1)中基本是相同的,所不同的是,过滤条件(即字符串 m_strFilter 的内容)不是固定给定的,而是通过交互操作界面来选择生成的,即通过提供一个交互式操作界面,组织一个过滤条件,然后以这个过滤条件进行过滤操作。在 16 章,将具体讨论实现一个通用的过滤条件编辑器,通过这个过滤条件编辑器来组织过滤操作用的过滤条件,然后进行过滤操作。

(3) 利用存储过程组织过滤功能

VC++ 6.0 企业版是为需要把 C++ 程序与 SQL 数据库集成的用户开发的,尤其适用于使用存储过程的用户。一个过程就是一些 SQL 代码段,这个代码段有一个名称,在 C++ 代码中可以通过这个名称来调用这个过程。同时,这个过程可以有多个参数,以不同的参数调用过程就相当于调用不同的 SQL 代码段。存储过程的创建和管理方法可参考第 15 章 15.2.1 小节。

所以,当利用 VC++ 6.0 企业版以 SQL 数据库为数据源(如 SQL Server 6.5)来组织数据库应用程序设计时,可以把一些固定方式的查询做成包含参数的存储过程的形式,以不同的参数调用这个存储过程实际上就完成了一个半固定形式的过滤操作。

在 C++ 代码中,调用存储过程实际上就是获得了存储过程中的 SQL 代码,可以在 CRecordSet 派生类的 GetDefaultSQL 函数中,通过调用存储过程来获得创建记录集合的 SQL 代码,以这个 SQL 代码来创建记录集合。通过在存储过程中包含过滤操作用的代码而达到过滤操作的目的。

例如,如果有一个针对某个 SQL 数据库的数据表 city 的存储过程 PROC1(**注意! 必须保证数据表是 SQL 数据表**),PROC1 存储过程的定义如下:

```
CREATE PROCEDURE PROC1 AS
select city_name,population
```

```
from city
where population > 100
```

可以在记录集合类 CData1Set 的 GetDefaultSQL 函数中，如下调用这个存储过程：

```
CString CData1Set::GetDefaultSQL()
{
    return _T("{CALL PROC1()}");
}
```

以上这个调用相当于实现如下的代码：

```
CString CData1Set::GetDefaultSQL()
{
    return _T("select city_name,population from city where population > 100";
}
```

以上这种使用存储过程组织记录集合的过滤条件的操作没有多大的优势，一般需要在程序运行过程中动态调用存储过程去完成过滤操作。为了动态的使用存储过程，可以在记录集合类 CData1Set 中定义一个记录存储过程名称的字符串变量：

```
public:
    CSting m_strProc;
```

将 SetDefultSQL 函数进行修改如下：

```
CString CData1Set::GetDefaultSQL()
{
    return _T(m_strProc);
}
```

在过滤函数中通过修改这个字符串 m_strProc 的内容，实现调用不同的存储过程来完成不同的过滤操作：

```
void CData1View::OnRecordFilter()
{
    if(m_pSet->IsOpen())   //如果记录集合对象处于打开状态，则关闭记录集合对象
        m_pSet->Close();
    m_pSet->m_strProc="{CALL PROC1()}"
    m_pSet->Open();        //重新打开记录集合对象
}
```

使用带有参数的存储过程，就可以比较容易地组织起具有活动参数的半固定模式的过滤操作。例如，在 SQL 数据库中创建一个存储过程 Proc2：

```
CREATE PROCEDURE PROC2@m_population int
AS select city_name,population
from city
where population > m_mpopulation
```

以上这个存储过程中包含了一个参数 m _ population,参数的类型是 int,参数名称前用字符@引导,在存储过程的定义体中利用这个参数组织了过滤条件。当然也可以定义多个参数的存储过程,如下这个存储过程 Proc3 包含了二个参数:

```
CREATE PROCEDURE PROC3 @m _ population int, @m _ enviroment, char(20)
AS select city _ name, population
from city
where (population > m _ mpopulation) and (enviroment = m _ enviroment)
```

利用带参数的存储过程,可以较容易地组织半固定模式的查询。以下的 OnRecordFilter 函数就可以利用存储过程 Proc2 来完成过滤条件为"population > 200"的过滤操作:

```
void CDataIView::OnRecordFilter()
{
    if(m _ pSet -> IsOpen())   //如果记录集合对象处于打开状态,则关闭记录集合对象
        m _ pSet -> Close();
    m _ pSet -> m _ strProc = "{CALL PROC2(200)}"
    m _ pSet -> Open();        //重新打开记录集合对象
}
```

通过以不同的参数去调用存储过程(只需利用不同的参数来产生 m _ strProc 字符串),可以方便的实现半固定模式的过滤操作。

(4) 过滤方式比较

除了固定条件的过滤操作外,本小节讨论了动态组织过滤条件和利用存储过程这两种过滤操作方式(在此称为第一种和第二种方式),以下对这两种方式进行比较:

这两种方式的最终目的,都是为了能够动态或相对动态的组织创建记录集合时的过滤子式,对于 CRecordSet 类来说,就是动态组织 CRecordSet 类的 m _ strSQL 字符串中的过滤子式。第一种方式具有更好的灵活性和适应性,可以应用在利用任何数据源的数据库应用程序设计中,而第二种方式具有更大的方便性,特别是配合 VC++ 6.0 企业版中的查询设计器,将可视化组织设计的过滤条件通过过程存储起来(参见第 13 章 13.2 节),使不懂 SQL 语言的程序开发者也可以去组织过滤和查询功能。

12.4.5 数据库排序功能

12.4.4 小节中讨论了对记录集合的过滤功能,采用类似的方法可以实现对记录集合的排序功能。在 11.2.3 小节讨论了 SQL 排序子式的组成,在 12.2.5 小节的讨论中,介绍了在利用 ODBC 进行数据库应用程序设计时,完成排序操作的实现方法。本小节将结合应用程序 DataI,

介绍数据库排序功能的实现方法。

在利用 ODBC 组织数据库应用程序设计时，是通过在打开记录集合对象前，修改 CRecordSet 类的成员变量 m _ strSort，把它作为创建记录集合时采用的过滤子式，而创建一个按照给定字段排序的记录集合。

(1) 实现固定字段的排序操作

在 12.4.1 小节，为菜单“操作(O)”的菜单项“数据排序”创建了一个消息处理函数 OnRecordSort，为了创建一个按字段 population 降序排列的记录集合，可将函数 OnRecordSort 的实现代码修改如下：

```
void CDataIView::OnRecordSort()
{
    if(m_pSet->IsOpen())          //如果记录集合对象处于打开状态，则关闭记录集合对象
        m_pSet->Close();
    m_pSet->m_strSort="population DESC"  //设置排序子式
    m_pSet->Open();                      //重新打开记录集合对象
}
```

在以上函数代码中，关闭记录集合对象，然后通过把记录集合对象的成员变量 m _ strSort (CRecordSet 类的成员变量)设置成"population DESC"，再重新打开记录集合对象时，就得到了按指定字段、指定方式排序的记录集合。

同样，可以来组织多个排序字段的排序操作：

```
void CDataIView::OnRecordSort()
{
    if(m_pSet->IsOpen())          //如果记录集合对象处于打开状态，则关闭记录集合对象
        m_pSet->Close();
    m_pSet->m_strSort="enviroment,population DESC"  //设置排序子式
    m_pSet->Open();                                 //重新打开记录集合对象
}
```

以上代码把记录集合按字段 enviroment 升序、按字段 population 降序排列。

(2) 实现任意组织排序字段的排序操作

与 12.4.4 小节中组织任意过滤条件的过滤操作类似，可以组织实现任意组织排序字段和排序方式的排序操作。在进行排序操作前，可通过交互操作界面去组织排序字段和排序方式(升序或降序)，动态生成排序操作用的排序子式(成员变量 m _ strSort)，然后对记录集合以组织的排序字段和方式进行排序。

应该说，实现任意组织排序子式的排序功能，比实现任意组织过滤条件的过滤功能简单得多。在第 17 章，将讨论实现一个任意组织排序字段和排序方式的排序子式编辑器。

12.5　DAO 数据库应用程序设计

DAO(Data Access Object)是数据库应用对象的英文简称。本章以上各节讨论了以 ODBC 为

基础的数据库应用程序的设计方法。除了利用 ODBC 外，还可以利用 DAO 来组织数据库应用程序设计。本节将介绍利用 DAO 来组织数据库应用程序设计的有关问题。

(1) DAO 数据库应用程序设计简介

在 VC++ 中，MFC 中提供了一些类，如 CRecordSet、CRecordView、CDatabase 等，来组织基于 ODBC 的数据库应用程序设计。与这些类相对应，MFC 中提供另外一些类，如 CDaoRecordSet、CDaoRecordView、CDaoDatabase 等，来组织基于 DAO 的数据库应用程序设计。在应用程序中，定义 CDaoRecordSet 对象来组织管理记录集合，定义 CDaoRecordView 对象来组织针对 CDaoRecordSet 对象记录的记录浏览编辑视图，定义 CDaoDatabase 类来管理数据库对象。

DAO 类在很大程度上是 ODBC 类的超集，它包含了 ODBC 类的绝大部分功能并增加了一些自己特殊的功能。应该说，用 DAO 来组织数据库应用程序设计时，具有更大的方便性和更强的功能。但是，用 DAO 组织数据库应用程序设计时，其数据源要求较狭窄，DAO 具有很大的针对性，最适合处理 Microsoft Access 的 .mdb 数据库，可以直接对 .mdb 数据库进行处理并具有较强的功能和较高的效率。从一定角度上，DAO 是专为 Access 的 .mdb 数据库设计的，在用 DAO 进行数据库应用程序设计时，能够提供比 ODBC 方法更多的功能。所以，在编制使用并将一直使用 Access 数据库的数据库应用程序时，采用 DAO 是明智的选择，如果是在利用其他的数据源进行数据库应用程序设计时，应采用 ODBC 方式。

虽然 DAO 能够读取带有 ODBC 驱动程序的数据源，当在处理任务时，并不特别有效，所以一般不采用 DAO 来利用 ODBC 数据源来组织程序设计。

(2) ODBC 和 DAO 的比较

ODBC 和 DAO 有很多的相同点，也有一些不同点，以下进行简单的介绍。

ODBC 和 DAO 的相同点主要体现在以下几个方面：

① ODBC 和 DAO 方法都能够处理 ODBC 数据源。在处理的效率上，ODBC 明显优于 DAO，但在处理 Access 的 .mdb 数据库上，DAO 可以对数据库直接处理并具有较高的效率。

② ODBC 和 DAO 进行数据库应用程序设计时，程序的构成思路和应用程序框架是完全一样的。都可以用 AppWizard 产生数据库应用程序的程序框架，除了应用程序中的一些类的基类不同外，从程序的组织到设计的方法上是基本一致的。

③ 在处理数据源的数据库时，ODBC 和 DAO 的方法是相同的。都是利用一个 MFC 类对象建立与要访问的数据库间的连接。在 ODBC 中，采用的是 CDatabase 类创建的数据库对象，而在 DAO 中，采用的是 CDaoDatabase 类创建的数据库对象，CDatabase 类和 CDaoDatabase 类中有很多相似的成员。在处理数据库时，都是创建一个记录集合对象来存储和管理从数据库中选择的记录集合。在 ODBC 中，这个记录集合类采用 CRecordSet 类为基类，而在 DAO 中，这个记录集合类采用 CDaoRecordSet 类作为基类。在 CDaoRecordSet 类中几乎包含了 CRecordSet 类中所有的成员，同时包含了一些附加的成员。ODBC 和 DAO 都使用类似的过程去管理数据源的内容，应用程序都是通过创建一个数据库对象与数据源的数据库建立联系，通过创建一个记录集合对象来管理数据源中数据库中的内容，通过调用相应类的成员函数去操纵记录集合。

ODBC 和 DAO 之间也存在着一些不同点，这些不同点主要表现在：

① ODBC 和 DAO 使用的 MFC 类尽管非常类似（如 ODBC 的 CDatabase 与 DAO 的 CDaoDatabase 类类似、ODBC 的 CRecordSet 类和 DAO 的 CDaoRecordSet 类类似）。但是两者之间有着明显的差别，在 DAO 采用的 MFC 类中包含更多的成员函数，能够组织和完成更多的操作功能。

② ODBC 使用宏和枚举来定义打开记录集合时可以使用的选项，DAO 可以通过定义常量来完成此目的。

③ 在 ODBC 环境下，定义数据库对象时默认使用快照模式（Snapshot），而在 DAO 环境下，采用的是动态模式（Dynaset）。

④ ODBC 方式可以适用于很多的数据库文件格式，只要这种数据库支持 SQL 并提供了 ODBC 驱动程序，所以采用 ODBC 可以用相同的方式来进行以各种数据源的数据库应用程序设计，其适用范围非常广。而 DAO 方式最适用于直接访问 Microsort Access 的 .mdb 数据库，而利用 ODBC 数据源访问数据库效率较低，一般不予采用。

⑤ 在低层上，ODBC 通过一组动态链接库（DLL）来实现，而 DAO 利用对象构造模型（COM）来实现。

⑥ 在 ODBC 环境下，数据库对象（CDatabase 对象）直接与数据源进行数据交换；而在 DAO 环境下，一个 CDaoWorkspace 对象处于 CDaoRecordSet 和 CDaoDatabase 对象之间，使得这个工作区可以与多个数据库对象进行数据交换。

(3) DAO 数据库应用程序设计方法

在 12.1.2 小节用 AppWizard 产生一个数据库应用程序 Data1 时，在第④个步骤，在图 12-1 操作界面中，在“DataSource”框中选择方式“DAO”（按中“DAO”前的圆按钮），在这种方式右边的编辑框中填入要作为数据源的 .mdb 数据库文件的名称（可通过按编辑框边的“...”按钮调出文件选择框选择得到），保持“RecordSet Type”框中的“Dynaset”（动态）选项。按“OK”按钮，系统会调出一个如图 12-2 所示的对话框 Select Database Tables，来选择以上选定的 Access 数据库（.mdb）中包含的数据表或数据视图，继续 12.1.2 小节中的以下步骤，就可以得到一个利用 DAO 的数据库应用程序框架。

第 13 章　使用 DataView 管理数据源

在第 2 章，介绍了 VC++ 中辅助程序设计的一些主要工具。在 VC++ 开发数据库应用程序时，为了对应用程序所采用的数据源进行管理，需要创建一个 Database Project 与数据源建立联系，并采用 VC++ 工作平台中的 DataView 工具通过创建的 Database Project 来管理数据源。本章将介绍用 AppWizard 创建 Database Project 的方法，以及利用 DataView 管理数据源的各项操作功能。

13.1　创建 Database Project

本节介绍创建 Database Project 的目的，以及利用 AppWizard 创建 Database Project 的步骤。

13.1.1　创建 Database Project 的目的

在 VC++ 中，当开发基于 ODBC 或 DAO 的数据库应用程序时，应用程序一般采用其他数据库系统的数据库作为数据源，而并非 VC++ 本身有自己的数据库。例如，通过 ODBC 可以开发利用 Foxpro、Access、Excel、SQL、Oracle 等数据库的数据库应用程序。而在开发过程中，为了满足开发和调试等各方面的需要，开发者需要能够随时知道所使用的数据库的内容，需要知道数据库所包含的数据表、视图，以及数据表的数据结构和数据表实际包含的记录情况等。要做到这一点，需要同时运行数据库属于的数据库系统，通过这些数据库系统来得到或管理数据库包含的内容。例如，如果开发的数据库应用程序采用的是 Access 的 .mdb 数据库，则需要运行 Microsoft Access 数据库系统，来获得或管理这个 .mdb 数据库的内容。这在应用上造成如下的不便：

① 需要保证在计算机中安装有这种数据库系统。

② 需要在开发数据库应用程序的同时运行数据库系统，并在 VC++ 开发环境和数据库系统中来回转换。

③ 更为不便的是，需要开发者必须会使用这种数据库系统。例如，如果用 VC++ 开发利用 Access 的 .mdb 数据库的应用程序，则要求开发者必须懂得 Access 系统的操作和使用方法。

为了克服这个问题，在 VC++ 中可以创建另外一种程序项目，就是我们要讨论的 Database Project，创建一个 Database Project 的目的，不是为开发应用程序创建程序框架，而是通过这个项目利用 VC++ 的辅助工具 DataView，对数据库应用程序中所采用的数据源进行浏览和管理。也就是说，在 VC++ 中通过使用一个通用的界面，去具体管理数据库应用程序所采用的数据源，这样，程序设计者就可以在 VC++ 开发环境下，利用统一方式来对来自不同数据源的数据表进行浏览及简单的管理功能，可极大地方便数据库应用程序的开发工作。

对于不同数据库系统的数据源建立的 Database Project，VC++ 的 DataView 能够提供的管理功能是有差别的。但对于所有支持 ODBC 的数据源，通过 DataView 都能够对数据源提供统一的浏览和简单的数据库操作，如浏览数据表结构、增加数据表的记录等。

VC++6.0 企业版，为集成开发 VC++ 和 SQL 数据库应用程序提供了强大的功能。在安装了 Visual Studio 辅助工具的情况下，在 VC++ 6.0 的集成开发环境中，可以利用 DataView 直接管理 SQL 数据库。即在 VC++ 和 SQL 之间建立了连接关系，可以在 VC++ 的开发环境下，通过无缝接口来操作和使用 SQL 数据库，不但能够完成对 SQL 数据库中的数据表、视图等的浏览等通用操作，还可以进行创建数据表、创建视图、创建并存储过程等操作，即实现了在 VC++ 开发环境下完成 SQL 数据库的很多管理功能。这样就可使开发者在完全不懂 SQL 数据库，完全不懂 SQL 语言的情况下，能够组织利用 SQL 数据库的数据库应用程序设计。

13.1.2　用 AppWizard 创建 Database Project

本小节介绍用 AppWizard 创建一个 Database Project 的步骤。创建一个 Database Project 的目的，是为了对 12.1.2 小节中产生的应用程序 Data1 所使用的数据源进行管理，所以需要在应用程序项目 Data1 的工作平台上再创建一个 Database Project 应用程序项目。以下是具体的创建步骤：

① 在 VC++ 开发环境中，打开 Data1 项目的情况下，运行“File”菜单下的“New”菜单项，进入 New 对话框界面，选择“Project”标签，并从列表框中选择“Database Project”选项，在“Project name”编辑框中输入要产生的项目的名称 Data1p，选择“Add Current Workspace”选项，得到如图 13-1 所示的对话框，按下“OK”按钮，就会进入到图 13-2 所示的对话框界面。

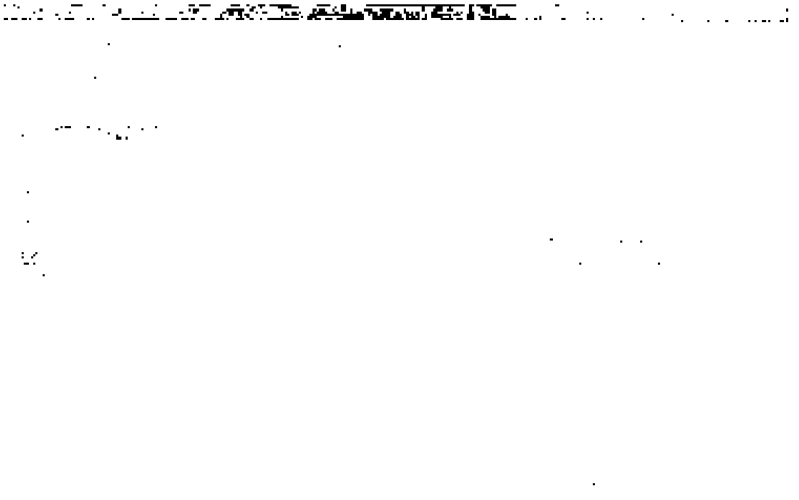


图 13-1　New 对话框

② 在图 13-2 所示的对话框中，顶部有两个标签：“File Data Source”和“Machine Data Source”，选中“Machine Data Source”标签时，列表框中列出的就是已经创建的“User DSN”数据源（也可以通过按“New...”按钮去创建一个新的 ODBC 数据源）。从列表框中选择创建应用程序 Data1 时采用的 ODBC 数据源 GISDATA，按“确定”按钮，就完成了 Database Project 的创建工作。在 VC++ 工作平台上就会出现一个新的工作项目“data1p”（注意，对于需要登录和密码确认的



图 13-2 选择 ODBC 数据源对话框

数据源,在按"确定"按钮后还会出现对话框界面要求输入登录帐户和密码),并且在 VC++ 工作区中出现了一个"DataView"标签。管理一个开发平台上多个程序项目的方法可参考第 2 章 2.7 节的讨论。

对于创建的 Database Project 项目 Data1p,可以通过 VC++ 开发环境中的 DataView 对数据源进行各种交互式的管理操作功能。DataView 能够对数据源提供的交互操作功能的多少,是与采用的数据源和 VC++ 的版本有关系的。对于非 SQL 的数据库提供的数据源,如 Access、Foxpro 等,DataView 能够提供普通的一些操作功能;而对于 SQL 数据库,如 Microsoft 的 SQL Server,在用 VC++6.0 企业版,并安装了 Microsoft Visual Stadio 中的辅助开发工具时,可以提供针对 SQL 数据库的更强大的管理功能。在 13.2 节,将对 DataView 提供的一些普通管理功能进行介绍,在 13.3 节,将介绍利用 VC++6.0 企业版的 DataView 对 SQL 数据库进行管理的其他功能。

13.2 使用 DataView 管理数据源的普通功能

本节介绍使用 DataView 管理数据源的普通功能。在 VC++ 开发环境中,打开 13.1.2 小节创建的包含 Data1 和 Data1p 的程序项目 Data1 时,用鼠标点中"DataView"标签,并按"+"打开 Data1p 项目,就会出现如图 13-3 所示的操作界面。对于 DataView 工作区中的操作方法,与其他的辅助工具如 ClassView 等类似。本节具体介绍利用 DataView 对 Data1p 项目所管理的数据源进行的各种操作功能。

13.2.1 浏览数据库的组成和结构

在图 13-3 所示的操作界面上,在工作区中会出项一个"Data1p"项目,按下"+"打开 Data1p 项目,在就会出现两个子项目:"Tables"和"Views","Tables"项目下列出的是当前连接的数据库中包含的数据表,"Views"项目下列出的是数据库中包含的各个数据视图。再按"+"打开数据表"city",可以列出数据表的各个字段。同样,再按"+"打开一个视图,在视图下列出的是这个

视图所包含的各个字段。通过用“+”打开子项目的操作，可以对数据源的组成和数据源中的每个数据表和视图的数据结构进行浏览。

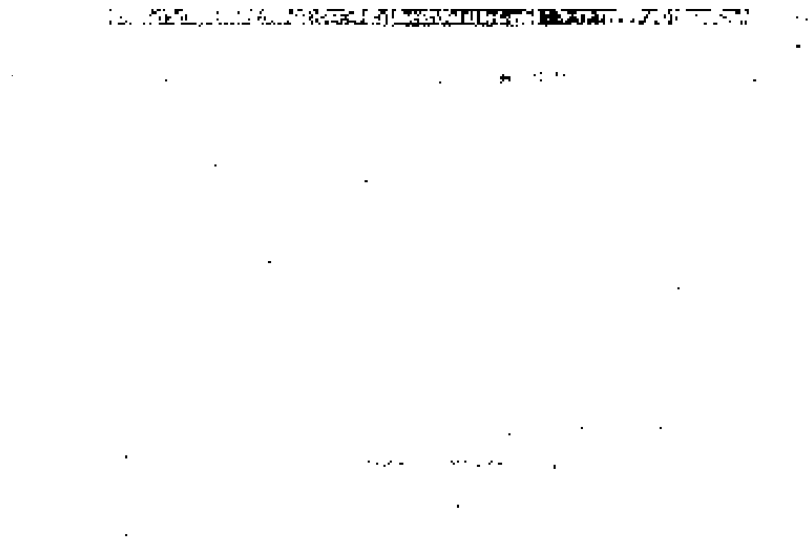


图 13-3　VC++ 的 DataView 工作区

13.2.2　Query 工具条功能介绍

在图 13-3 所示的操作界面中，双击数据表“city”，就会在 VC++ 开发环境的文件编辑区中打开一个或多个窗口并列出一个工具条 Query，系统的主菜单也会发生变化，会增加一个包含了大量菜单项的“Query”菜单，进入到图 13-4 所示的操作界面。在这个操作界面上，通过工具条 Query，可以对数据表提供各种交互操作功能（在“Query”菜单中有对应工具条各按钮的菜单项）。本小节对工具条能够完成的功能进行简单的介绍。图 13-5 列出了 Query 工具条。

（1）数据表的不同编辑窗口

在图 13-4 所示的编辑区中，可以打开四类子窗口来完成交互式编辑数据表的功能（在图 13-4 中只打开了一个窗口）。这四类窗口的显示或隐藏是通过工具条上的①～④按钮分别来控制（或通过“Query”菜单的对应菜单项）。以下对这四类窗口进行简单介绍：

① 图表窗口。通过 Query 工具条的第(1)按钮来显示或隐藏这个窗口。图 13-6 是针对数据表 city 的图表窗口，在图表窗口中，包含多个项目，第一个项目“*”表示数据表的所有字段，以下的其他项目是数据表的字段列表。对于每个项目，具有多个状态：选择状态、是否用于排序、排序方式等。通过这个窗口，可以交互式的完成选择数据表中的字段、设置排序字段等操作。一个图表窗口用来管理一个数据表，因为 SQL 操作经常涉及到多个数据表（如从多个数据表中选择字段形成一个记录集合），所以可以在 VC++ 编辑区中为多个数据表打开图表窗口。

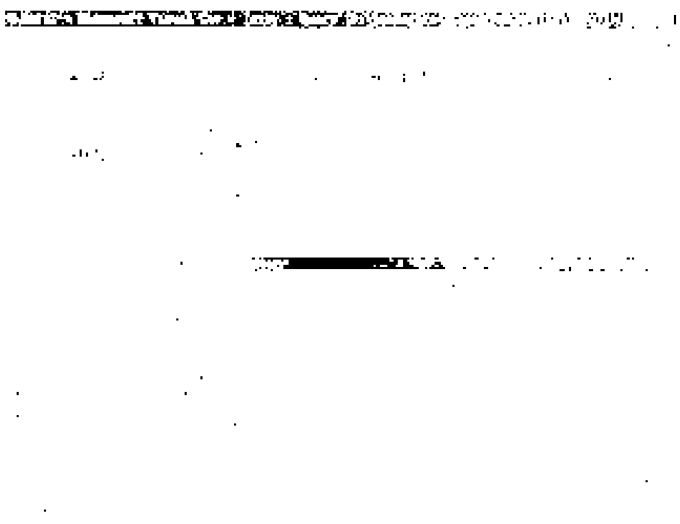


图 13-4 数据表编辑操作工作界面

图 13-5 Query 工具条



图 13-6 图表窗口

② 网格窗口。通过 Query 工具条的第(1)按钮可以显示或隐藏网格窗口。图 13-7 列出的是一个网格窗口,在 VC++ 编辑区中有且只有一个网格窗口。网格窗口中有一个网格图表,网格窗表有多个列,通过网格图表,可以组织针对多个数据表的过滤子式、排序子式等操作功能,在以下各小节中将具体讨论实现这些操作的方法。

③ SQL 窗口。通过工具条的第(3)个按钮可以来显示或隐藏 SQL 窗口。在 SQL 窗口中,列出的是 SQL 语言代码,这些 SQL 语言代码就是通过在图表窗口和网格窗口进行交互式的选择或操作形成的 SQL 语句。SQL 窗口中的 SQL 语句是用 Query 工具条完成操作的核心,图表

图 13-7　网格窗口

窗口和网格窗口中提供的操作功能都是为了得到 SQL 窗口中的 SQL 语句。

④ 结果窗口。通过工具条的第(4)个按钮可以来显示或隐藏结果窗口。结果窗口中显示的是执行 SQL 窗口中的 SQL 语句后得到的结果。图 13-4 所示操作界面中的结果窗口,列出的是数据表 city 的所有记录。

图 13-8 是打开了四个窗口后的操作界面。图表窗口和网格窗口是为交互式的表示和组织 SQL 语句而设计的,图表窗口、网格窗口、SQL 窗口从不同的角度来描述 SQL 语句,其内在是相通的,对一个窗口内容的修改会影响到其他窗口的内容。

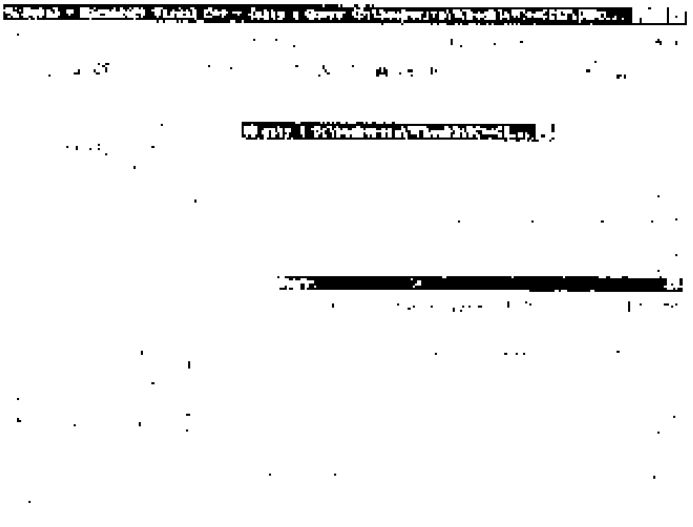


图 13-8　打开多个窗口的数据表编辑界面

(2) 选择操作类型

按下 Query 工具条中的"Change Type"按钮,可以下拉得到一个列表框,列表框中列出的是能够组织的 SQL 操作。在列表框中包含如下的操作:"Select"、"Insert"、"Insert Value"、"Update"、"Delete"、"Make Table"等。

在列表框中选择不同的操作时,SQL 窗口中的 SQL 语句会发生变化。

(3) 检验和执行 SQL 操作

Query 工具条中的第(7)个按钮,可以用来检查 SQL 窗口中的 SQL 语句是否合法。按中工具条的第(6)个按钮("!")用来执行 SQL 语句,按下这个按钮时,系统会执行 SQL 窗口中的 SQL 代码,完成 SQL 操作后,得到的结果显示在结果窗口中。

(4) 辅助操作按钮

Query 工具条上的第(8)个按钮,用来设置或取消数据表的一个字段为升序排序字段;Query 工具条上的第(9)个按钮,用来设置或取消数据表的一个字段为降序字段;Query 工具条的第(10)按钮用来取消针对数据表的某个字段的过滤子式。以上这三个按钮的操作是针对图表窗口的,只有在某个图表窗口中选中了一个字段时才能被执行(第(10)个按钮只有在选中的字段有过滤子式的情况下才能别执行)。

(5) Group by 按钮

Query 工具条上的第(11)个按钮用来组织 SQL 的 Group by 子式。Group by 子式的构成相对比较复杂,本书中不再详述,读者可参阅有关的 SQL 语言手册。

(6) Query 工具条的作用

在 Query 工具条中,能够选择多种操作,在 SQL 窗口中提供出完成每种操作的 SQL 语言框架,然后通过图表窗口和网格窗口来补充和完成完整的操作语句,其最终目的是提供一套交互式的 SQL 语句组织工具,为不懂 SQL 语言的数据库程序开发者完成 SQL 语句的书写提供帮助,为懂 SQL 语言的程序设计者提供一套辅助产生 SQL 语句的工具。

在图 13-4 所示的编辑界面上,以 Query 工具条为引导,通过图表窗口、网格窗口和 SQL 窗口,用来完成各个 SQL 操作功能:交互式组织一个 SQL 语句,检查 SQL 语句的合法性,执行 SQL 语句并返回执行的结果到结果窗口中。SQL 窗口用来显示 SQL 语句,图表窗口和网格窗口用来辅助组织 SQL 语句。实际上,图表窗口、网格窗口和 SQL 窗口反映的是同样的内容,只是从不同的角度来体现 SQL 语句。如果您熟悉 SQL 语言,就可以直接在 SQL 窗口中书写 SQL 语句,而不用图表窗口和网格窗口去组织。

以下各小节,介绍完成各个 SQL 操作的组织实现步骤。

13.2.3 组织 Select 操作

在 11.2 节,介绍了 SQL 语言的选择子式、过滤子式、排序子式的语法规则。本小节介绍利用 Query 工具条交互式组织 Select 操作的步骤。在 Query 工具条中,按中"Change Type"按钮选择"Select"选项。就可以利用编辑界面的各个窗口来组织与 Select 操作有关的 SQL 语句。

(1) 选择字段

Select 操作中,首先需要选择记录集合的组成字段,即选择需要返回哪个数据表的哪些字

段。要完成这个工作，可以通过图表窗口和网格窗口来分别完成，以下分别介绍利用图表窗口和网格窗口选择字段的操作步骤：

① 通过图表窗口选择。可以在 VC++ 编辑区中打开一个或多个图表窗口（当图表窗口处于显示状态时，可以把工作区中的数据表拖动到编辑窗口的图表区中来打开这个数据库的图表窗口），每个图表窗口用来管理一个数据表。要从某个数据表中选择字段组成记录集合时，在编辑窗口中打开这个数据表的图表窗口，在图表窗口中，通过选中或取消各个项目前的“√”标志来选择或取消选择这个字段。在选择操作过程中，可以看到 SQL 窗口中的 SQL 语句的 Select 子式会发生变化。

② 通过网格窗口选择。同样可以通过网格窗口来组织记录集合包含的字段。在网格窗格中，选择一个数据表的一个字段的操作步骤如下：在网格窗口中一个空行的 Table 栏中点中鼠标键得到一个列表框，从列表框中选择要选择字段的数据表（在 VC++ 编辑区域打开了几个数据表的图表窗口，在列表框中也就有几个对应于这几个数据表的选项），在“Column”列中单击鼠标左键调出一个列表框，从列表框中选择要选择的字段（“*”代表所有的字段），在“Output”列中做选择标志“√”，就将这个字段选入了记录集合，从 SQL 窗口中可以看到 Select 子式的相应修改。选择字段时，可以在“Alias”列中输入一个字段别名，可以对字段别名进行取消和修改，按回车键会将所做的修改反映到 SQL 窗口中。

(2) 组织过滤条件

可以通过网格窗口来组织进行 Select 操作时的过滤条件，也就是形成 SQL 语句中的过滤子句。在网格窗口中，每行中有三个“Or...”列，如果此行对应着一个字段（不是对应着“*”表示的所有字段），就可以在一个或多个“Or...”列中输入一个基本条件子式。可以在网格窗口的多个行的“Or...”列中输入基本条件子式。按照以下的步骤来得到 SQL 语句的过滤子式：将网格窗口同一个“Or...”列中各个行的基本条件子式由连接符“AND”连接成一个 SQL 子式，再将各个“Or...”列的 SQL 子式由连接符“OR”连接得到最后的过滤子式。

以下我们以具体的实例来讨论过滤子式的组织方法。如果现在要组织一个过滤条件“population > 100”，只需在网格窗口中增加一行，在“Column”列中选择字段“population”，在第一个“Or...”列中输入“> 100”，就会在 SQL 窗口中得到过滤子式：

```
WHERE (population > 100)
```

而对于过滤条件“(population > 100) AND (population < 800)”，则需要在网格窗口中增加两行，在“Column”列中，将字段名称都选择为“population”，在第一行的第一个“Or...”列中输入字符串“> 100”，在第二行的第一个“Or...”列中输入字符串“< 800”，则会在 SQL 窗口中得到如下的过滤子式：

```
WHERE (population > 100) AND (population < 800)
```

而对于过滤条件“(population > 100) AND (population < 800) OR (area > 500)”，则需要在图表窗口中增加三行，在“Column”列中，将第一和第二行的字段名称选择为“population”，在第三行的字段名称选择为“area”，在第一行的第一个“Or...”列中输入字符串“> 100”，在第二行的第一个“Or...”列中输入字符串“< 800”，在第三行的第二个“Or...”列中输入字符串“> 500”，

则会在 SQL 窗口中得到 SQL 语句中的过滤子式：

```
WHERE (population > 100) AND (population < 800) OR(area > 500)
```

在网格图表中编辑过滤条件时，可以通过对网格各行的“Or...”列中的内容进行删除，达到删除这个字段的过滤子式的目的。为了加快编辑效率，当需要删除一个字段所构成的所有基本过滤子式时，可以通过图表窗口来实现。在图表窗口中用鼠标点中要删除过滤子句的字段，按下 Query 工具条的第(10)按钮，就可以删除这个字段组织的基本过滤子句。对于这个操作，也可以通过在图表窗口中按右键调出快捷菜单，通过运行菜单项“Remove Filter”来完成(同样可以使用 VC++ 开发环境中的“Query”菜单下的“Remove Filter”菜单项来完成)。操作完毕后可以看到，在网格窗口中对应这个字段的过滤子句全部删除(即网格窗口中列“Column”是这个字段的行中的所有“Or...”列的内容全部删除)。在 SQL 窗口中可以得到新的 SQL 语句(所有使用这个字段的过滤子式都已经删除)。

如果网格窗口中的某行是专门用来组织过滤条件的(如不是用来选择字段)，可以通过将这一行全部删除来修改过滤条件。删除网格窗口的行的操作步骤是：在网格窗口要删除的行前单击鼠标右键，选中这一行(变成黑色)，按键盘的“Del”键，就可以全部删除这一行的内容。

(3) 组织排序字段

在 11.2.3 小节中曾经讨论了 SQL 排序子式的语法，现在利用图表窗口或网格窗口来交互组织排序字段，得到 SQL 排序子式。以下介绍通过图表窗口或网格窗口组织排序字段的步骤。

① 使用图表窗口组织排序字段。在一个图表窗口中，用鼠标选中要用做排序操作的字段，按下 Query 工具条中的第(8)个按钮，就可以选择这个字段为排序字段，排序方式是升序；按中 Query 工具条第(9)个按钮，就可以选择这个字段为排序字段，排序方式是降序。在 SQL 窗口中产生相应的排序子式。例如，将 city 数据表的“population”字段选择为升序排序字段时，在 SQL 窗口中有如下的排序子式：

```
ORDER BY population
```

而将这个字段选择为降序排序字段时，排序子式如下：

```
ORDER BY population DESC
```

在图表窗口中，对于已经是排序字段的字段进行以上相同步骤的操作，可以取消这个排序字段。

对于以上完成的操作步骤，同样可以通过在图表窗口中，在要设置为排序字段的字段上击鼠标右键调出快捷菜单，选择菜单项“Sort Ascending”或“Sort Descending”来将字段设置或取消为方式为升序或降序的排序字段。也可以调用主菜单中“Query”菜单下的两个菜单项“Sort Ascending”或“Sort Descending”来完成相应的操作。

可以在图表窗口中选择多个字段作为排序字段。排序字段的先后顺序是按照在图表窗口中设置排序字段的先后顺序来排列的，在图表窗口中无法看到各个排序字段的先后顺序。

② 使用网格窗口组织排序字段。同样可以利用网格窗口来组织排序字段。如果要增加一个排序字段，则在要设置为排序字段的行(如果在网格窗口中没有这个字段，可以按照(1)中

介绍的方法在网格中增加一新的行)的"Sort Type"列中单击鼠标左键,就会弹出一个包含以下三个选项的列表框:Ascending、Descending 和 Unsorted(只有在这个字段已经设置为排序字段时才有这个选项),通过这三个选项分别可以把这个字段设置成升序、降序或非排序字段。

如果有多个排序字段时,可以在网格窗口中调整排序字段的先后顺序。在网格窗口的要改变排序先后顺序的字段所在行中,在"Sort Order"列中击鼠标键,就会弹出一个列表框,列表框中有可选择的顺序号(如果有三个排序字段,则在列表框中用三个数字 1、2、3 可供选择),通过选择一个顺序号来确定当前字段在排序字段中的顺序,其他排序字段的顺序号也会随着变化。修改排序字段的先后顺序后,在 SQL 窗口中,SQL 语句的排序子式中的各个字段的顺序会发生相应变化。

(4) 执行 SQL 操作

除了能够组织以上的操作外,还可以进行组织"Group by"子式等操作。在完成 SQL 语句的组织后,按中工具条中的按钮(6)("!"),就可以执行 SQL 窗口中的 SQL 语句,操作返回的结果显示在结果窗口中。在执行 SQL 操作前,可以首先按中工具条中的第(7)个按钮来检查 SQL 语句是否合法。

13.2.4　组织 SQL 的 Insert 操作

SQL 的 Insert 操作完成的功能是:从一个或多个数据表中选择字段形成记录集合,将记录集合增加到一个已经存在的数据表中。通过 VC++ 的 Requery 工具条可以组织 SQL 的 Insert 操作。从 Query 工具条中的"Change Type"列表框选择操作项"Insert",选择"Insert"选项后,系统会弹出一个如图 13-9 所示的对话框,在对话框的"Table name"列表编辑框中选择将记录集合插入到的数据表的名称。就进入 SQL 的 Insert 操作。

图 13-9　插入数据表选择对话框

如果在图 13-9 中选择了一个已经存在的数据表名称"city1"(假定数据表 city1 在数据源中存在)的情况下,打开 SQL 窗口,可以看到原始的 SQL 语句:

```
INSERT INTO "city1"
SELECT *
FROM city
```

city1 是要将记录集合插入到的数据表名称;SELECT 子句以下的操作是从当前数据表 city 中选择记录集合(也可能从多于一个的数据表中选择记录集合),即完成与 13.2.3 小节相同的操作。

所以,组织 Insert 操作的步骤与组织 SELECT 操作的步骤几乎完全一样,通过图表窗口和网格窗口从 city 数据表中选择记录集合的字段、过滤条件、排序字段、Group by 子式等,其具体

操作可参照 13.2.3 小节中所进行的所有操作。值得注意的是,从数据表中选择的记录集合的结构一定与要插入到的数据表的结构一致,Insert 操作多用在两个同结构的数据表间拷贝数据记录,将一个数据表中符合过滤条件的记录集合增加到要插入到的数据表记录后。

通过执行 SQL 操作(执行 Query 工具条中的第(6)个按钮)可以完成 SQL 的 Insert 操作,操作不成功会返回失败消息。

13.2.5 组织 SQL 的 Insert Value 操作

SQL 的 Insert Value 操作用来完成在一指定数据表中增加一条记录。用编辑界面可以较容易地组织进行 Invert Value 操作的 SQL 语句。

从 Query 工具条的"Change Type..."列表框中选择"Insert Value"选项,就会进入到 Insert Value 操作状态。在编辑窗口中打开了 city 数据表的图表窗口的情况下,在 SQL 窗口中,可以看到原始的 SQL 语句:

```
INSERT INTO city
    ()
VALUES ()
```

在 Invert Value 操作下,用 Query 的各图表来选择要赋值的字段并输入字段值,组织一个 SQL 语句,完成在数据表中增加新记录的操作。

(1) 选择要插入值的字段

可以在图表窗口选择要插入数值的字段,在图表窗口中通过选择数据表的字段(此时不能选中"*"代表的所有字段),来选择要对数据表中的哪些字段赋值,SQL 窗口中的 SQL 代码也相应随着变化。例如,在图表窗口中选择字段"city_name"和"population"时,SQL 窗口中的 SQL 语句如下所示:

```
INSERT INTO city
(city_name, population)
VALUES (,)
```

在以上代码的第 2 行的括号中列出的是要插入值的字段,这些字段相应的字段值填写在第 3 行代码 VALUES 所跟随的括号内(按第 2 行代码中字段的顺序)。例如,如果要将字段 population 的值设置为 1000,将 city_name 字段的值设置为"北京",则可将 SQL 窗口中的 SQL 代码修改如下:

```
INSERT INTO city
    (city_name, population)
VALUES ('北京', 1000)
```

(2) 利用网格窗口交互输入字段值

在 Invert Value 操作状态下,网格窗口发生变化,图 13-10 列出的是选择对两个字段赋值后的网格窗口。

图 13-10　Invert Value 操作时的网格窗口

网格窗口中的图表中只包含两列：Column 和 New Values，Column 列中列出的是数据表的字段名称，New Values 列中列出的是对应字段的值。在这个网格窗口中可以非常容易地完成给各字段赋值的工作，在网格窗口中空行的“Column”列中单击鼠标左键，就会弹出一个列表框，从列表框中可以选择要赋值的字段名称，在“New Value”列中输入字段的值，按回车键后，就会看到 SQL 窗口中 SQL 语句会随着发生变化（图表窗口中也发生变化，选中了要赋值的字段）。输入字段值时，对于字符型字段对应的字段值，不用输入引号，回车时系统会根据字段的类型自动加上引号。

(3) 执行 SQL 操作

按下 Query 工具条的第(6)个按钮（“!”），就会执行 SQL 窗口中的 SQL 语句，完成 SQL 的 Insert Value 操作（在执行前可以通过第(7)个按钮来检查 SQL 语句的合法性），如果操作成功，就会在数据表中插入一个新记录。必须注意，并不是每个插入新记录的操作都能成功，如没有给非空字段（Not Null）赋值，以及主索引字段值在数据表中已经存在等，都会引起操作不成功，返回错误提示信息。

13.2.6　组织 SQL 的 Update 操作

Update 操作用来完成将数据表中全部或部分记录中指定的字段用新的值来替换，即用来完成对数据表中已有记录字段值的替换。从 Query 工具条的“Change Type”列表框中选择“Update”选项，就进入到了 SQL 的 Update 操作状态。在这种操作状态下，通过图表窗口和网格窗口来组织进行 Update 操作的 SQL 语句。以下是选择“Update”操作项后，显示在 SQL 窗口的 SQL 语句：

```
UPDATE city
SET
```

关键字 SET 后面跟随的是有关修改各字段值的代码。以下讨论具体的实现步骤。

(1) 用图表窗口选择要替换值的字段

可以用图表窗口选择需要对数据表中的哪些字段的值进行替换。字段的选择操作与 13.2.5小节完成 Insert Values 操作时相同。例如，从 city 数据表的图表窗口中选择字段 population 和 enviroment，在选择了这个字段后，SQL 窗口中的 SQL 语句如下所示：

```
UPDATE city
SET population = , enviroment =
```

如果将字段 population 和 enviroment 的值分别设置为 1000 和“一般”,可以将 SQL 语句改为如下:

```
UPDATE city
SET population = 1000, enviroment = '一般'
```

(2) 用网格窗口交互修改各字段值

同 13.2.5 小节讨论的组织 Invert Value 操作相同,可以用网格窗口来交互选择和设置需要修改值的字段名称,以及对应要修改成的值。图 13-11 列出的是给两个字段赋值的网格窗口。

图 13-11 Update 操作时的网格窗口

在网格中有“Column”和“New Value”列,可以参考 13.2.5 小节的操作来选择和设置需要修改的字段和字段值。与 Invert Value 操作时不同的是,在 Update 操作时,因为可以对由多个数据表组成的关系数据库的记录进行修改,所以在网格中还有一个“Table”列,在这个列中用来选择数据表的名称。在这个列中选择数据表后,在“Column”列中的列表框中列出的是这个数据表的各个字段。在网格窗口中选择和设置字段值后,在 SQL 窗口中的 SQL 语句会发生相应的变化。

(3) 组织过滤条件

执行(2)中操作得到的 SQL 语句,被执行时会对数据表(或几个数据表组成的关系数据库表)中的所有记录相应的字段值进行修改,将字段设置成 SQL 语句中赋予的值。通常情况下,只是需要将数据表的一条或部分记录进行修改操作,这就需要设置一个过滤条件,即为 Update 操作语句再设置一个过滤子式,通过过滤子式来对数据表的记录进行过滤,只对符合过滤条件的数据表记录的字段值进行修改。从图 13-11 中可以看出,在网格窗口中包含了三个“Or...”列,可以应用这些列来组织过滤条件,具体操作方法可参考 13.2.3 小节中的(2)所做的介绍。设置了过滤条件后,在 SQL 窗口的 SQL 语句中,会找到相应的过滤子式。

(4) 执行 SQL 操作

完成了以上的各操作步骤,得到进行 Update 操作的 SQL 语句后,按中 Query 工具条上的第(6)个按钮(“!”),执行 SQL 窗口中的 SQL 语句(在执行前可以先运行按钮(7),检查 SQL 语句的合法性),操作成功时,可以在结果窗口中看到经过修改后的记录集合列表。

13.2.7 组织 SQL 的 Delete 操作

SQL 的 Delete 操作完成从记录集合(一个数据表或多个关系数据表组成的记录集合)中删

除全部或部分记录。从 Query 工具条中的“Change Type”列表中选择“Delete”项，就进入了 Delete 操作状态。在这种操作状态下，可以组织和执行 Delete 操作的 SQL 语句，完成从记录集合中删除全部或部分记录的功能。以下是选择“Delete”项后在 SQL 窗口中得到的 SQL 代码：

```
DELETE FROM city
```

因为对记录集合记录的删除涉及到所有的字段，所以，在 Delete 操作状态下，图表窗口没有直接用处(但可以在图表窗口删除每个字段的过滤子式)，最主要的操作是在网格窗口中组织过滤条件，即需要对记录集合的哪些记录进行删除。

(1) 设置过滤条件

执行以上不带任何过滤条件的 SQL 语句，会将整个数据表删除。在进行删除操作时，一般需要设置过滤条件，来删除记录集合中符合过滤条件的记录。过滤子式可以通过网格窗口来组织，图 13-12 是在 Delete 状态下的网格窗口，在这个网格窗口中可以组织过滤条件，用网格窗口组织过滤条件的操作方法可参考 13.2.3 小节中的(2)。设置过滤条件后，在 SQL 窗口中的 SQL 语句中会产生一个相应的过滤子式。

图 13-12　Delete 操作时的网格窗口

(2) 执行 SQL 操作

按中 Query 工具条上的第(6)个按钮，执行 SQL 窗口中的 SQL 语句(在执行前可以先运行按钮(7)，来检查 SQL 语句的合法性)，执行成功后，在结果窗口中可以看到删除了符合过滤条件的记录后的记录集合。

13.2.8　组织 SQL 的 Make Table 操作

SQL 的 Make Table 操作完成的功能是：从已经有的数据表中选择字段形成一个记录集合，根据记录集合产生一个新的数据表，并将记录集合的内容插入到数据表中。从 Query 工具条中的“Change Type”列表中选择“Make Table”项，系统会弹出一个对话框(如图 13-13 所示)要求在对话框中输入要产生的数据表的名称，输入一个数据表名称“Table1”并退出对话框，就进入了 Make Table 操作状态。从 SQL 窗口中可以看到没有经过任何修改时的原始 SQL 代码：

```
SELECT
INTO Table1
FROM city
```

以上代码可以完成数据表的复制功能，将数据表 city 的内容完整的复制到数据表 Table1

图 13-13 数据表名称输入对话框

中。SELECT 字句用来组织记录集合的字段。组织 Make Table 操作的步骤与组织"SELECT"操作(参见 13.2.3 小节)几乎完全一样,不同的是,Select 操作时将得到的记录集合显示在了结果窗口中,而 Maek Table 操作则把得到的记录集合存储到了一个新的数据表中。可以通过图表窗口和网格窗口组织记录集合的字段、过滤条件、排序字段、Group by 子式等。其具体操作可参照 13.2.3 小节。

在完成各种设置,组织完成 SQL 语句后,按中第(6)个按钮,执行 SQL 窗口中的 SQL 语句。如果操作成功,会产生一个新的数据表 Table1,数据表 Table1 中包含的内容就是 SQL 语句中得到的记录集合,可以在"DataView"工作区中的"Tables"项目下找到新产生的数据表 Table1。

13.3 DataView 管理 SQL 数据库

VC++6.0 企业版为开发 C++ 和 SQL 数据库程序提供了极大的方便。在 VC++ 6.0 中,开发以 SQL Server 数据库为数据源的应用程序时,对于 DataView 辅助开发工具,能够提供比普通的数据源更多的管理功能。

13.3.1 针对 SQL Server 数据源的 Database Project

如果要利用 SQL Server 数据库作为数据源来进行数据库应用程序的开发,需要进行以下的步骤:

① 建立 ODBC 数据源,在 11.3.3 小节中介绍了 SQL Server 数据库的"User DSN"数据源的创建方法,创建一个使用 SQL Server 数据库的 ODBC 数据源。

② 用 AppWizard 创建一个数据库应用程序。

③ 利用①建立的数据源建立一个 Database Project 应用程序。

ODBC 处理 SQL Server 数据源与其他数据源是不一样的,利用 ODBC 使用一些数据源(如 Foxpro 等)时,只是通过 ODBC 去操作数据库文件,而与这些数据库系统本身没有任何关系。而 VC++ 则是通过 ODBC 建立了与 SQL Server 数据库系统的连接,VC++ 中实现的一些数据库管理功能是通过调用 SQL Server 数据库系统来实现的。

针对 SQL Server 数据库建立的 Database Project,可以利用 DataView 提供更多的数据库管理功能,即除了完成 13.2 节完成的功能之外,还能够提供其他的管理功能,实现了一套针对 SQL Server 数据库的完整管理工具。本节将对这些额外的功能进行介绍。在 SQL Server 数据库系统中,有一个例子数据库 pubs,针对这个数据库建立一个 ODBC 数据源 pubs,利用这个数据源用 AppWizard 产生一个 Database Project 项目 Data2p,本节将以 Data2p 项目为例,讨论 DataView 有关的各种功能。

在 VC++ 6.0 企业版中，在开发环境上打开 Database Project 项目 Data2p，按“DataView”标签进入 DataView 工作区，在工作区中按“+”依次打开各个项目，得到如图 13-14 所示的操作界面。从操作界面可以看出，在 DataView 工作区中，除了“Tables”、“Views”子项目外，还有“Database Dialgrams”和“Stored Producures”等两个子项目，“Database Dialgrams”项目下列出的是表示数据表之间关系的数据库图表，而在“Stored Producures”下列出的是各个 SQL 存储过程。



图 13-14　Data View 工作区界面

以下各小节，将对 DataView 中的一些额外功能进行简单介绍。

13.3.2　创建数据表和修改数据表结构

在 VC++ 6.0 企业版中，可以在连接的 SQL 数据库中创建新的数据表，也可以对 SQL 数据库中已经存在的数据表的结构进行修改。在 VC++6.0 企业版中，建立了一个 SQL 数据表结构交互编辑工具，可以非常方便地编辑数据表的结构。

(1) 新建数据表

在 DataView 工作区中的“Tables”项目上单击鼠标右键弹出一个快捷菜单，在快捷菜单中有一个“New Table”菜单项，在安装了 Visual Develop Stadio 中的辅助开发工具，并且数据源是 SQL 数据库时，此菜单处于可执行状态。运行这个菜单，就会弹出一个对话框，在对话框中输入新建数据表的名称并退出对话框后，就会在编辑区中出现一个交互编辑数据表结构的表格(如图 13-15)，利用它可以很方便地对数据表的结构进行修改。

在图 13-15 所示的编辑表中，数据表的一个字段占用表格的一行，包含了大量的性质信息。SQL 数据表的字段类型及其各种参数的描述可参考有关的 SQL 数据库使用手册。在了解了 SQL 数据库字段组成参数的情况下，可以利用图 13-15 所示的操作界面进行数据表结构的

图 13-15 SQL数据表结构编辑界面

编辑工作。因篇幅关系,在此不再详细讨论 SQL 数据表的字段构成参数,而只对编辑数据表结构时的一些操作进行简单的介绍。

① 增加字段。在图 13-15 所示的数据表结构编辑表中,在数据表结构后的新行中输入一个字段的各个参数(包括字段名称、类型、长度、是否非空等),就在数据表结构中增加了一个新字段。

② 插入字段。在图 13-15 所示的数据表结构编辑表中,用鼠标右键单击某个字段所在的行,在弹出的快捷菜单中选择运行菜单项"Insert Column",则会在这当前行前插入一空行,在这个空行内输入要插入字段的各个参数,就在数据表结构中的指定字段前插入了一个新字段。

③ 选择字段。为了对数据表结构编辑表中的某个字段进行删除、剪裁、拷贝等操作,需要首先能够选择字段。选择的方法是:用鼠标键单击编辑表某行左边的方格区域,这一行就会变黑,处于选中状态,按下 Shift 键后再按下鼠标键,可以选择多个字段行。

④ 删除字段。用③介绍的方法选择一个或多个字段行,按下键盘中的删除键"Del",就能够将这些选中的字段删除。也可以通过在某个字段行上右击鼠标键,在弹出的快捷菜单中选择"Delete Column"菜单项来删除这个字段。

⑤ 剪裁和复制字段。对于字段行,也支持剪裁板操作,用③中介绍的方法选择字段行,按 Ctrl + X 键,能够删除选择的字段行并将它存储到剪裁板中;按 Ctrl + C 键,可以把选择的字段行存储到剪裁板;在编辑表的某行中按 Ctrl + V 键,可以将剪裁板中的字段行插入到当前位置。

(2) 修改数据表结构

在 DataView 工作区中的"Tables"项目下要修改结构的数据表上单击鼠标右键,就会弹出一个快捷菜单,运行快捷菜单中的菜单项"Design",就会进入到图 13-15 所示的数据表结构编辑表格。在这个编辑表格中,包含了这个数据表结构已经存在的字段。可以编辑修改这个数据表的结构,操作方法与(1)相同。

13.3.3 新建数据视图或修改已有的数据视图结构

在 VC++ 6.0 企业版开发环境中,可以在数据表的基础上创建新的数据视图,也可以对既有的数据视图的数据结构进行修改。

(1) 新建数据视图

在 DataView 工作区中的"Views"项目上,单击鼠标右键弹出快捷菜单,运行快捷菜单中的菜单项"New Views",就会进入到 Query 工具条的工作状态(如图 13-3)。这个工作环境与 13.2.3 小节中讨论的 Query 工作环境几乎完全相同,不同的是在当前的编辑状态下只能组织 SQL 的 Selelct 操作(Query 工具条中只包含与组织 Select 操作有关的按钮,并且不能再选择其他类型的 SQL 操作)。

创建一个视图的过程,与 13.2.3 小节组织 SQL 的 Select 操作基本相同,打开要创建视图需要的数据表的图表窗口,并选择字段、组织过滤条件、组织排序字段、组织 Group by 子式等。

在组织完成 Select 操作的 SQL 语句后,可以执行 SQL 语句,在结果窗口中返回了记录集合。在组织完成 SQL 语句后,可以通过保存操作来创建一个数据视图,运行 VC++ "File"菜单下的菜单项"Save"(或按下快捷键 Ctrl + S),系统会弹出一个对话框(如图 13-16),在对话框中输入要创建的数据视图名称,按"OK"按钮,就会将 SQL 语句保存而创建一个新的数据视图。从 DataView 工作区的"Views"项目下可以找到新创建的数据视图。

图 13-16　数据视图名称选择对话框

数据视图没有自己的物理存储区域,是建立在数据表上的一个虚表。所以,以上保存操作形成一个数据视图时,并不是把 Select 操作得到的记录集合保存形成一个实际存在的数据表,而只是保存了形成记录集合的 SQL 语句,数据视图就是通过这个 SQL 语句由实际的数据表虚拟出一个记录集合,可以像一个实际的数据表那样来对数据视图进行处理。

(2) 修改既有数据视图

除了能够新建数据视图外,还可以对已经存在的数据视图进行修改。修改数据视图的操作方法如下:在 DataView 工作区中"Views"项目上,在要进行修改的数据视图上单击鼠标右键,在弹出的快捷菜单中选择"Design"菜单项,就会进入与(1)中增加数据视图时相同的操作界面。不同的是,操作界面中各个窗口的初始状态和内容都是按照这个数据视图的 SQL 语句来设置,即操作界面的初始状态正好能够组成定义当前数据视图的 SQL 语句。

可按照 13.2.3 小节中介绍的方法,对 SQL 语句进行修改。修改完毕后,运行 VC++ "File"菜单下的菜单项"Save"(或按下快捷键 Ctrl + S),可将修改后的数据视图保存起来。

13.3.4　创建和使用存储过程

在 SQL 语言中,存储过程类似于一个独立的函数,是 SQL 语句的集合。在 VC++ 6.0 企业版中,可以创建和维护 SQL 数据库的存储过程。在 13.2 节中介绍了利用 Query 工具条组织

简单的 SQL 语句的方法(即使对 SQL 编程技术不高的开发者),这种方法在交互式操作数据库时非常有用,我们可以将这些 SQL 语句存储成存储过程,通过对存储过程的调用就达到了调用这些 SQL 语句的目的。存储过程实际上提供了一个组织 SQL 语句的手段,可以通过存储过程组织大量的功能模块。

(1) 存储过程的语法组成

在 VC++ 开发环境的 DataView 工作区中,在数据库项目下有一个存储过程项目(Stored Procedure)。展开这个存储过程项目,就可以列出在这个数据库中已经存在的各个存储过程,双击一个已经存在的存储过程 reptq3,就会在编辑区中打开一个窗口,在窗口中列出了这个存储过程的组成代码:

```
CREATE PROCEDURE reptq3 @lolimit money, @hilimit money,
@type char(12)
AS
select pub_id, type, title_id, price
from titles
where price > @lolimit AND price < @hilimit AND type = @type OR type LIKE '%cook%'
order by pub_id, type
COMPUTE count(title_id) BY pub_id, type
```

以下对存储过程的编辑风格和语法组成进行简单的介绍:

① 编辑窗口的着色风格。在编辑存储过程组成代码的窗口中,对于存储过程中的不同部分有不同的颜色显示:关键字(如 SELECT、from 等)用蓝色表示,注释语句用绿色表示,其他类型的文本用黑色表示。

② 存储过程的头。以上列出的存储过程 reptp3,其头由以下几行代码组成:

```
CREATE PROCEDURE reptq3 @lolimit money, @hilimit money,
@type char(12)
AS
```

存储过程定义以"CREATE PROCEDURE"关键字开始,这个关键字后跟随的是存储过程的名称("reptp3"),再后面跟随的是存储过程的参数及类型("@lolimit money, @hilimit money, @type char(12)")。在存储过程 reptp3 中,一共有 3 个参数:lolimit、hilimit、type,其类型分别为 money、money、char(12),各参数间由逗号隔开。在存储过程定义中,参数名称由"@"符来引用。

"AS"关键字后面跟随的是存储过程的代码体。

③ 存储过程的代码体。"AS"关键字后面跟随的就是存储过程的代码体,这是一些 SQL 代码,可以通过 Query 工具条提供的辅助功能去组织产生(见 13.2 节),并将产生的代码拷贝到编辑存储过程的窗口中,实现将 Query 工具条辅助设计的 SQL 代码以存储过程的形式存储起来。

在存储过程的代码体中,应该包含着过程头中定义的各个参数,通过以不同的参数值调用存储过程,来实现可变参数模式的固定操作(例如实现一定模式的查询操作)。

(2) 新建存储过程

可以在 VC++6.0 企业版的开发环境中创建新的存储过程，在 DataView 工作区中的“Stored Prodecure”项目上单击鼠标右键，在弹出的快捷菜单中选择运行菜单项“New Stored Procedure”，就会在编辑区中打开一个编辑新过程的窗口，在窗口中有原始的存储过程组成代码：

```
Create Procedure / * Procedure _ Name * /
As
    return (0)
```

按照(1)中讨论的格式去组织存储过程的组成代码。在新建存储过程的操作中，可以利用 Query 工具条来组织各种 SQL 语句，将用 Query 工具通过交互操作产生的 SQL 语句拷贝到存储过程的编辑窗口中“AS”关键字的后面。对于不熟悉 SQL 语言的开发者，通过这种方法也可以比较容易地组织存储过程的编写工作。

完成对编辑窗口中存储过程组成代码的修改后，运行主菜单“File”下的“Save”菜单项(或直接按下 Ctrl + S 键)，就可以将这个新建的存储过程进行保存。从 DataView 工作区中的“Stored Procedures”项目下可以找到新增加的存储过程。

(3) 修改存储过程

在 DataView 工作区中的“Stored Procedures”项目下，找到要进行修改的存储过程，用鼠标双击这个存储过程(或者在这个存储过程上单击鼠标右键，在弹出的快捷菜单中运行菜单项“Open”)，就能够用编辑窗口打开一个存储过程。在编辑窗口中对存储过程的组成代码进行修改，修改完毕后，调用“File”菜单的“Save”菜单项(或用快捷键 Ctrl + S)保存对存储过程的修改。

(4) 调用存储过程

在 VC++ 开发环境下，可以用以下的方法运行存储过程：在 DataView 工作区中的“Stored Procedures”项目下，在要运行的存储过程上单击鼠标右键弹出快捷菜单，运行快捷菜单中的“! Run”菜单项，就可以运行存储过程(在存储过程的编辑窗口中单击鼠标右键弹出快捷菜单，在快捷菜单中也有一个“! Run”菜单项来执行这个存储过程)。如果要执行的存储过程没有参数，存储过程就会直接被执行；如果要执行的存储过程有参数，如运行 reptp3 时(这个过程有 3 个参数，见(1))，就会弹出一个如图 13-17 所示的对话框，在这个对话框中在“VALUE”列中填入参数的值，按“OK”按钮后，就会以填入的 3 个参数调用存储过程 reptp3。

运行存储过程返回的结果显示在 VC++ 开发环境的 Output 窗口中(即编译应用程序时显示编译信息的窗口)。

(5) 在 C++ 代码中调用存储过程

(4)中介绍了在 VC++ 开发环境中直接执行存储过程的方法，在用 VC++ 进行应用程序设计时，对于存储过程来说，不只是要在开发环境中被调用，最主要的是要能够在 C++ 代码中被调用。在 12.2.4 小节中，介绍了在 CRecordSet 类中调用存储过程的方法，请读者参阅。

图 13-17 过程参数输入对话框

13.3.5 创建和使用数据库图表

在关系数据库应用程序设计中,必须详细知道数据库中的各个数据表的结构,以及各个数据表之间的相互连接关系,这对组织数据库应用程序的各种功能是极为重要的。而且在很多情况下,一个数据库应用程序往往是由多个开发者合作完成的,让合作开发者迅速地弄清楚这些数据表的结构以及数据表的相互关系也是必须的。为了尽快地了解大量的数据表结构信息和数据表之间的连接关系,VC++6.0 企业版中提供了一个叫做数据库图表(Database Diagrams)的项目,用图表的方式简捷地来展现数据表的结构以及数据表之间的连接关系。同时提供了用图表的方式方便地组织数据表结构和数据表之间的相互关系的功能。

在 DataView 工作区中,在数据库下有一个"Databse Dialgrams"项目,在这个项目下列出的是已经存在的数据库图表。

(1) 创建数据库图表

在 SQL 数据库中可以新建一个数据库图表,以下是新建一个数据库图表的步骤:

① 在 DataView 工作区中的"Database Diagrams"项目上单击鼠标右键弹出快捷菜单,运行快捷菜单中的"New Diagrams"菜单项。就会在编辑区中打开一个空的编辑窗口,在 VC++ 的主菜单中增加了一个菜单"Diagram",在编辑区中出现了一个"Database Diagram"工具条。

② 在 DataView 工作区中的"Tables"目录下,选择一个数据表(如 authors),并把它拖动到编辑区中(在数据表上按下鼠标左键,移动到编辑区中并松开鼠标左键),在编辑区中就会打开此数据表的图表窗口(图表窗口的含义见 13.2.2 小节)。

③ 采用步骤②,把其他的数据表(如 titleauthor、titles)也拖动到编辑区中,打开相应的图表窗口。

④ 在编辑区中会出现各个数据表的连接标志,可以拖动图表窗口和连接标志的位置,使它们的关键字按照图 13-18 所示的方式排列。这样就一目了然地清楚了各个数据图表间的相互关系。

⑤ 运行"File"菜单下的"Save"菜单(或通过工具条上的相应按钮或直接使用快捷键 Ctrl+S)保存这个数据库图表,系统会弹出一个对话框要求输入数据库图表的名称,然后将这个数据库图表以输入的名称保存起来。在 DataView 工作区的"Database Diagrams"项目下就会列出这个数据库图表。

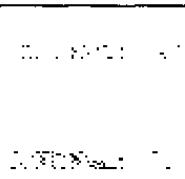


图 13-18　数据库图表编辑界面

(2) 编辑既有数据库图表

在 DataView 工作区的"Database Diagrams"项目下，选择需要修改的数据库图表，用鼠标双击这个数据库图表，就会进入到图 13-18 所示的数据库图表编辑界面，并在编辑界面上打开这个数据库图表(以保存时的形态显示在编辑界面中)。参考(1)的操作步骤对这个数据库图表进行编辑修改，修改完毕后，保存数据库图表。

13.3.6　Database Diagram 工具条功能

在如图 13-18 所示的编辑数据库图表的界面上，有一个工具条 Database Diagram。这个工具条中有很多按钮，提供了很多设计和管理 SQL 数据库的功能，可以方便而且直观地完成数据库的设计和管理。其操作的方便性甚至优于 SQL Server 数据库系统自身提供的设计和管理数据库的功能。在 VC++ 开发环境的菜单系统中，在编辑数据库图表的操作状态下，有一个菜单"Diagram"，菜单中有大量菜单项，提供了与工具条相应的操作功能。这一小节简要介绍 Database Diagram 工具条能够提供的操作功能。图 13-19 是对各按钮进行了编号后的 Database Diagram 工具条。



图 13-19　Database Diagram 工具条

(1) 表示数据表结构的不同图表

在 Database Diagram 工具条编辑界面上，对于其中的每一个数据表，可以用五种不同的图表方式来显示或编辑数据表及其结构。在 Database Diagram 工具条上，有五个按钮用来选择各

个图表格式。

按钮(7)用来选择数据表的结构图表。在这个图表中,数据表每个字段的详细信息列在图表中,可以对每个字段进行修改。图 13-20 中列出的是数据表 Titles 的结构图表,此图表提供对数据结构的修改操作,与 13.3.1 小节提供的操作功能相同。



图 13-20 Titles 数据表的结构图表

按钮(8)用来选择数据表的字段名称图表。在这个图表中,只包含了数据表的字段名称信息。这个是默认形式的图表。图 13-18 列出的数据库图表中,各个数据表都是以这种图表形式显示。

按钮(9)用来选择主关键字图表。这个图表与字段名称图表类似,不同的是在图表中只列出了与其他数据表建立联系的关键字(有一个关键字是本数据表的主关键字)。可以在这个图表中进行与关键字有关的操作。图 13-21 列出的是 Titles 数据表的主关键字图表。

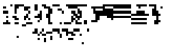


图 13-21 Titles 关键字图表

按钮(10)用来选择数据表名称图表。在这个图表中,只列出了数据表的名称。

按钮(11)用来选择一个定制字段图表。定制字段图表是图 13-20 结构图表的一部分,是从图 13-20 图表中选择部分列来组成的图表。在数据表 Titles 的图表上单击鼠标右键,就会弹出一个快捷菜单,运行快捷菜单的"Modify Custom View"菜单项,就会进入图 13-22 所示的对话框,在这个对话框中就可以选择要在定制字段图表中显示的字段。

图 13-22 所示的对话框中有两个列表框"Available columns"和"Selected Columns",左边列表框中列出的是允许进行选择的列,在右边列表框中列出的是已经选择的列,定制字段图表就是顺序显示列表框"Select Columns"中列出的列。当选择所有允许的列时,定制字段图表就是一个图 13-20 所示的结构图表;当只选择"Column Name"列时,定制字段图表就是字段名称图表。在图 13-22 所示的操作界面中,可以将各个项在两个列表框中转移:

"＞"按钮,把在"Available columns"列表框中选择的项转移到"Select Columns"列表框中。

"＞＞"按钮,把"Available columns"列表框中的所有项都转移到"Select Columns"列表框中,即定制字段图表将包含所有的字段,此时,定制字段图表就是结构图表。

"＜"按钮,把"Select Columns"列表框中选择的项移到"Available columns"列表框中,在定制字段图表中去掉这个列。

"＜＜"按钮,把"Select Columns"列表框中所有项移到"Available columns"列表框中,在"Select

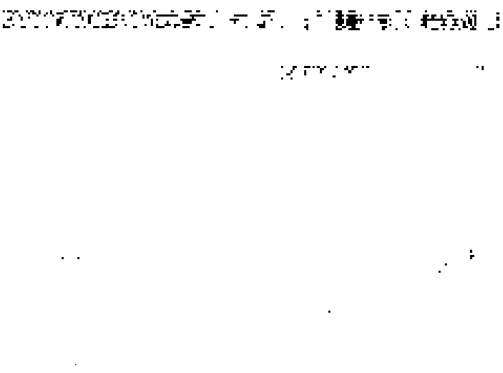


图 13-22　图表显示字段设置对话框

Columns"列表框中将不包含任何项，即定制字段图表中将不包含任何列。

在图 13-22 所示的对话框中还有两个按钮用来调整"Select columns"列表框中项目的顺序：

"↑"按钮，把"Select columns"列表框当前选择项与上一个项交换位置。

"↓"按钮，把"Select columns"列表框当前选择的项与下一个项交换位置。

在图 13-22 所示的对话框中设置完毕要在定制字段图表中显示的列后，按"OK"按钮退出对话框，在编辑界面上，就会显示数据表 Titles 的定制字段图表。

(2) 增加、修改、删除及移走数据表

运行 Database Diagram 工具条的第(1)个按钮，系统就会弹出一个对话框，在对话框中输入新建数据表的名称，然后就会在编辑界面上新建一个名称为输入名称的数据表。这个数据表以结构图表的形式显示，可以在这个图表中编辑新建数据表的结构，来创建一个新的数据表。

对于已经存在的数据表，将这个数据表从 DataView 工作区拖到 Database Diagram 工具条的编辑区域，将数据表的显示方式选择为结构图表，在结构图表中对数据表的结构进行修改。

在编辑区中用鼠标点中一个要删除的数据表，按中 Database Diagram 工具条的第(2)个按钮，确认提示信息后，会将这个数据表从数据库中删除。

在编辑区中用鼠标选中一个要从数据库图表中移走的数据表，按中工具条 Database Diagram 的第(3)个按钮，就会从当前的数据库图表中移走这个数据表。此操作与删除数据表操作是完全不同的，删除数据表操作是从数据库中实际删除了数据表，而移走数据表操作只是把这个数据表从当前编辑的数据库图表中移走，数据表本身并没有删除。

(3) 排列数据库图表

按中 Database Diagram 工具条的第(18)个按钮，系统就会自动排列编辑区域中所有数据表的位置。

(4) 增加连接表

在 Database Diagram 编辑区中选择一个数据表，按中工具条上的第(15)个按钮，系统就会把

与选中数据表有连接关系的数据表全部拖到编辑区域(在编辑区打开),并通过指针表明了与选择数据表的连接关系。

(5) 增加注释信息

按中 Database Diagram 工具条的第(5)个按钮,就会在编辑区中增加一个类似于 CStatic 控制的文本框,可以在数据库图表中任意拖动文本框,这样就可以在数据库图表中加入各种提示信息,来帮助应用程序合作开发者更好地理解数据库图表。

(6) 显示数据表的关系连接

按中 Database Diagram 工具条上的第(12)个按钮,就可以在数据库图表中显示出图表之间的连接关系。

第 14 章　实现数据库浏览功能

本章将在第二篇讨论的矢量图形系统 Draw 的基础上，实现一个数据库浏览视图，实现对数据库的浏览功能。在 2.1.2 节用 AppWizard 产生应用程序 Draw 时，在步骤(5)(如图 2-5 所示)选择的是“Header files only”选项，选择了这个选项后产生的应用程序代码中，在头文件 stdafx.h 中包含了如下代码：

```
#include <afxdb.h>        // MFC ODBC database classes
```

在 draw.rc 中包含了如下代码：

```
#include "l.chs \ afxdb.rc"        // Database resources
```

头文件 afxdb.h 中包含了用 ODBC 进行数据库应用程序设计所需要的各种定义，afxdb.rc 资源文件中包含了开发 ODBC 数据库应用程序所需要的一些资源。对于在如图 2-5 中选择“None”选项产生的应用程序框架，可以通过在头文件 stdafx.h 和资源文件 draw.rc 中插入以上代码来达到同样的目的。

14.1　创建一个数据库浏览视图

为了在矢量图形系统 Draw 中方便地对图形系统和数据库系统同时进行管理，可以考虑在应用程序 Draw 当前视图(CDrawView 类来管理)之外再新建另外一个视图，在这个视图中实现对数据库记录集合的管理功能。应用程序 Draw 运行时，对于一个文档，可以创建两个不同的视图，一个视图中显示矢量图形，主菜单提供有关矢量图形操作的各种功能；另外一个视图中显示数据库的记录集合，提供另外一个主菜单实现各种数据库管理功能。要达到这个目的，实际上是完成多视图操作，与 MFC 应用程序固有的多视图操作不同的是，需要提供另外一类不同的视图。本节介绍在 MFC 应用程序中创建另外一类视图的方法，并为应用程序 Draw 创建另外一个数据库浏览视图。

14.1.1　视图的创建过程

为了能够在 MFC 文档/视图结构的应用程序中创建一类新的视图，首先分析视图的创建过程。在 2.2 节讨论应用程序的执行步骤时，曾介绍在 MFC 应用程序的应用类中有一个 InitInstance()函数，在应用程序的执行过程中，这个函数被 MFC 应用程序框架调用，完成程序的初始化和类的注册功能。在应用程序 Draw 的应用类 CDrawApp 中，可以找到函数 InitInstance 的实现代码：

```
BOOL CDrawApp::InitInstance()
{
    ..... //省略以上代码
```

```
    CMultiDocTemplate * pDocTemplate =
        new CMultiDocTemplate(
        IDR_DRAWTYPE,                    //注册所用的资源
        RUNTIME_CLASS(CDrawDoc),         //注册文档类
        RUNTIME_CLASS(CChildFrame),      //注册子框架类
        RUNTIME_CLASS(CDrawView));       //注册视图类
    AddDocTemplate(pDocTemplate);        //增加一个文档模板对象
    ..... //省略以下代码
}
```

从以上代码可以看出,在程序运行时,首先创建了一个多文档模板对象,并将这个模板对象加入到了当前的文档模板中。在程序运行时,MFC 应用程序的一些标准操作都是靠注册到模板对象的类来组织实现的。

① 打开文档操作。当运行标准菜单的"新建"和"打开..."菜单项时,在 MFC 中实现的框架程序就会利用注册到文档模板对象中的文档类 CDrawDoc 来创建一个新的文档对象,利用注册的子框架类 CChildFrame 来创建一个新的子框架窗口对象,利用注册的视图类 CDrawView 来创建一个新的视图对象,并建立三者之间的关系。这个创建过程是在 MFC 中实现的,在当前的应用程序中找不到实现代码。应用程序执行时,会在程序主框架窗口中创建一个新的窗口。这个窗口连接的是新创建的 CDrawDoc 对象,通过新创建的 CChildFrame 对象来管理,窗口中的客户区由新创建的 CDrawView 对象来管理。当视图处于激活状态时,在主框架窗口中采用的是 IDR_DRAWTYPE 菜单资源。

② 打开视图操作。当运行 MFC 多文档应用程序的标准菜单项"新建窗口"时,在 MFC 中实现的框架程序就会利用注册的子框架类 CChildFrame 来创建一个子框架窗口对象,利用注册的视图类 CDrawView 类创建一个视图对象,利用当前框架窗口中活动的文档对象作为连接的文档对象,并建立三者间的联系。菜单采用的是 IDR_DRAWTYPE 菜单资源。以下是 MFC 中实现的"新建窗口"菜单(ID 号为 ID_WINDOW_NEW)的消息处理函数的代码:

```
void CMDIFrameWnd::OnWindowNew()
{
    CMDIChildWnd * pActiveChild = MDIGetActive(); //得到活动的子框架窗口对象指针
    CDocument * pDocument;
    if (pActiveChild == NULL ||
       (pDocument = pActiveChild->GetActiveDocument()) == NULL)
    //如果不存在子框架窗口对象或者不存在活动的文档对象,应用程序出错,给出提示信息退出
    {
        TRACE0("Warning: No active document for WindowNew command.\n");
        AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
        return; // command failed
    }
    CDocTemplate * pTemplate = pDocument->GetDocTemplate();  //得到指向文档模板对象的指针
    ASSERT_VALID(pTemplate);
    //利用当前活动的文档对象指针,调用 CreatNewFrame 函数创建一个子框架窗口对象和视图对象
```

```
    CFrameWnd * pFrame = pTemplate - > CreateNewFrame(pDocument, pActiveChild);
    if (pFrame = = NULL)      //如果创建子框架类对象不成功
    {
        TRACE0("Warning: failed to create new frame. \ n");
        return;      // command failed
    }
    pTemplate - > InitialUpdateFrame(pFrame, pDocument);   //在屏幕上显示创建的视图
}
```

在 MFC 多文档应用程序中，运行“新建窗口”菜单项时创建的视图是相同的，即一个文档所具有的视图都是相同的，都是由 CDrawView 对象来管理，都采用 IDR _ DRAWTYPE 菜单资源。以下小节讨论对于同一个文档对象，创建不同类别视图的方法。

14.1.2　创建另外一类视图的步骤

14.1.1 小节分析了视图的创建过程，这小节讨论通过控制这个创建过程，来实现对于当前的文档对象创建另外一类视图的方法和步骤。

(1) 新建一个视图类

为了能够新建一类视图，需要首先创建一个视图类。通过 MFC 的 CView 及其派生类(如 CScrollView、CFormView、CRecordView、CDaoView 等)派生一个视图类，通过这个视图类来对要新建的另外一类视图进行管理。为了在矢量图形系统 Draw 中创建另外一类视图，实现对记录集合的管理功能，需要创建一个新的视图类 CDatabaseView(在 14.1.3 小节中介绍创建视图类的步骤)。

(2) 新建一个菜单资源

当在应用程序中创建另外一类视图时，因为要在这类视图中提供不同的操作功能，所以一般需要新建一个菜单资源，供新建的视图使用(当然，也可以与应用程序原来的视图采用相同的菜单资源)。

(3) 新建一个多文档模板对象

从 14.1.1 中列出的 OnWindowNew 函数代码中可以看出，MFC 应用程序的视图对象是通过一个多文档模板对象来创建的。为了能够创建另外一类视图，需要创建另外一个多文档模板对象，通过这个模板对象来创建新的视图。在应用程序类的 InitInstance 函数中加入代码，创建一个新的文档模板对象：

```
BOOL CDrawApp::InitInstance()
{
    ..... //省略以上代码
    DocTemplate1 = new CMultiDocTemplate(
        IDR _ DRAWTYPE1,                    //注册所用的资源
```

```
        RUNTIME_CLASS(CDrawDoc),        //注册文档类
        RUNTIME_CLASS(CChildFrame),     //注册子框架类
        RUNTIME_CLASS(CDdatebaseView)); //注册视图类

    pDocTemplate = new CMultiDocTemplate(
        IDR_DRAWTYPE,                   //注册所用的资源
        RUNTIME_CLASS(CDrawDoc),        //注册文档类
        RUNTIME_CLASS(CChildFrame),     //注册子框架类
        RUNTIME_CLASS(CDrawView));      //注册视图类
    AddDocTemplate(pDocTemplate);       //增加一个文档模板对象
    ..... //省略以下代码
}
```

以上 InitInstance 函数中的黑体代码，创建了一个新的文档模板对象 pDocTemplate1。CDatabaseView 是新视图类的名称，IDR_DRAWTYPE1 是视图激活时所采用的资源名称。变量 pDocTemplate 和 pDocTemplate1 在应用类 CDrawApp 中定义：

```
private:
    CDocTemplate * pDocTemplate1;
    CDocTemplate * pDocTemplate;
```

(4) 创建视图

视图的创建代码可以参考 CMDIFrameWnd 类中的 OnNewWindow 函数来实现，在应用程序 Draw 的主框架窗口类 CMainFrame 中为菜单“新建窗口”(ID 号 ID_WINDOW_NEW)增加一个消息处理函数，以替代原来的消息处理函数(CMDIFrameWnd 类中的 OnNewWindow 函数)。为了在运行“新建窗口”菜单项时能够创建新的视图，修改消息处理函数 OnWindowNew 的实现代码如下：

```
void CMainFrame::OnWindowNew()
{
    CMDIChildWnd * pActiveChild = MDIGetActive();     //得到指向活动的子框架窗口对象的指针
    CDocument * pDocument;
    if (pActiveChild == NULL ||
      (pDocument = pActiveChild->GetActiveDocument()) == NULL)
    //如果不存在子框架窗口对象或者不存在活动的文档对象，应用程序出错，给出提示信息退出
    {
        TRACE0("Warning: No active document for WindowNew command. \n");
        AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
        return;      // command failed
    }
    CDrawApp * pApp = (CDrawApp * )AfxGetApp();          //得到指向当前应用类对象的指针
    CDocTemplate * pTemplate = pApp->GetDocTemplate1();
    ASSERT_VALID(pTemplate);
```

```
    //利用当前活动的文档对象指针,调用 CreatNewFrame 函数创建一个子框架窗口对象和视图对象
    CFrameWnd * pFrame = pTemplate - > CreateNewFrame(pDocument, pActiveChild);
    if (pFrame = = NULL)                    //如果创建子框架类对象不成功
    {
        TRACE0("Warning: failed to create new frame. \ n");
        return; // command failed
    }
    pTemplate - > InitialUpdateFrame(pFrame, pDocument);      //在屏幕上显示创建的视图
}
```

以上 OnWindowNew 函数中的 GetDocTemplate1 函数用来获得文档模板对象的指针,GetDocTemplate1 函数在应用类 CDrawApp 中定义:

```
public:
    CDocTemplate * GetDocTemplate1();
```

在 CDrawApp 类的实现文件 draw.cpp 中,加入函数实现代码:

```
CDocTemplate * CDrawApp::GetDocTemplate1()
{
    return pDocTemplate1;
}
```

OnWindowNew 函数的实现步骤与 CMDIFrameWnd 类中的 OnWindowNew 函数相同,以下是主要的操作实现过程:

① 得到指向当前活动视图的子框架窗口对象的指针:

```
CMDIChildWnd * pActiveChild = MDIGetActive();
```

通过当前活动的子框架窗口对象得到当前活动的文档对象。在当前屏幕上不存在活动的子框架窗口对象,或者存在子框架窗口对象但不存在活动文档对象的情况下,显示出错信息退出应用程序:

```
if (pActiveChild = = NULL || (pDocument = pActiveChild - > GetActiveDocument()) = = NULL)
{
    TRACE0("Warning: No active document for WindowNew command. \ n");
    AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
    return; // command failed
}
```

② 通过全局函数 AfxGetApp 获得指向当前应用程序应用类对象的指针,通过应用类对象指针得到指向定义的文档模板对象的指针:

```
CDrawApp * pApp = (CDrawApp * )AfxGetApp();
CDocTemplate * pTemplate = pApp - > GetDocTemplate();
ASSERT_VALID(pTemplate);
```

③ 通过指向文档模板对象的指针,以当前活动的文档对象指针和子框架窗口对象指针为

参数,调用文档模板类的 CreateNewFrame 函数,创建子框架窗口对象和视图对象,然后通过调用文档模板类的 InitialUpdateFrame 函数,初始化并显示创建的视图窗口。

```
CFrameWnd * pFrame = pTemplate - > CreateNewFrame(pDocument, pActiveChild);
if (pFrame == NULL)
{
    TRACE0("Warning: failed to create new frame. \n");
    return; // command failed
}
pTemplate - > InitialUpdateFrame(pFrame, pDocument);
```

这一小节讨论了在文档/视图结构的 MFC 应用程序中创建另外一类视图的过程。14.1.3 小节将在矢量图形系统 Draw 的基础上,创建一个新的数据库浏览视图。

14.1.3 在矢量图形系统中创建数据库浏览视图

本小节将在第二篇讨论的矢量图形系统 Draw 的基础上,创建另外一类用来浏览数据库记录集合的视图。

(1) 创建并修改一个视图类

为了创建另外的数据库浏览视图,需要新建一个视图类。在此,创建一个普通的滚动视图类,通过这个滚动视图类实现对记录集合的浏览,并通过操作菜单提供对记录集合的操作功能。

在 2.2.3 小节中介绍了用 ClassWizard 增加一个 MFC 派生类的方法。在 VC++ 开发环境中打开应用程序 Draw,启动 ClassWizard,在 ClassWizard 对话框界面中选择运行"Add Class..."按钮,进入到图 2-11 所示的对话框(或者在 ClassView 工作区中,通过在"Draw Classes"项目上击鼠标右键调出快捷菜单,运行菜单项"New Class..."进入对话框),在"Base Class"列表编辑框中选择 CScrollView 类作为基类,在"Name"编辑框中输入类的名称"CDatabaseView",保持"File name"编辑框中的默认文件名 dbaseview.cpp,按"OK"按钮,就会在应用程序 Draw 中增加一个视图类 CDatabaseView,类的头文件在 dbaseview.h,实现文件在 dbaseview.cpp。

对于以上创建的视图类 CDatabaseView,还没有建立与文档类 CDrawDoc 的联系。所以需要在视图类 CDatabaseView 中,增加获得文档对象的成员函数 GetDocument 的运行和调试部分。在视图类 CDatabaseView 中加入获得文档对象指针的函数 GetDocument 后。CDatabaseView 类的定义如下(黑体部分是增加的有关 GetDocument 函数的代码):

```
class CDatabaseView : public CScrollView
{
protected:
    CDatabaseView();      // protected constructor used by dynamic creation
    DECLARE_DYNCREATE(CDatabaseView)

// Attributes
public:
```

```
    CDrawDoc * GetDocument();        //得到文档对象指针的函数

// Operations
public:

    // Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDatabaseView)
    protected:
    virtual void OnDraw(CDC * pDC);      // overridden to draw this view
    virtual void OnInitialUpdate();      // first time after construct
    //}}AFX_VIRTUAL

// Implementation
protected:
    virtual CDatabaseView();
# ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
# endif

    // Generated message map functions
    //{{AFX_MSG(CDatabaseView)
    // NOTE - the ClassWizard will add and remove member functions here.
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
# ifndef _DEBUG        // 文件 Dbaseview.cpp 中的调试版本
inline CDrawDoc * CDrawView::GetDocument()
    { return (CDrawDoc * )m_pDocument; }
# endif
```

在实现文件 dbaseview.cpp 中,列出了视图类 CDatabaseView 的实现代码(黑体部分是增加的有关 GetDocument 函数的代码):

```
# include "stdafx.h"
# include "Draw.h"
# include "Dbaseview.h"

# ifdef _DEBUG
# define new DEBUG_NEW
# undef THIS_FILE
static char THIS_FILE[] = __FILE__;
# endif
```

```
/////////////////////////////////////////
// CDatabaseView
IMPLEMENT_DYNCREATE(CDatabaseView, CScrollView)

//类的构造函数
CDatabaseView::CDatabaseView()
{
}

//类的析构函数
CDatabaseView::~CDatabaseView()
{
}

BEGIN_MESSAGE_MAP(CDatabaseView, CScrollView)
    //{{AFX_MSG_MAP(CDatabaseView)
        // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////
// CDatabaseView drawing

//类的初始化函数
void CDatabaseView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal;
    // TODO: calculate the total size of this view
    sizeTotal.cx = sizeTotal.cy = 100;
    SetScrollSizes(MM_TEXT, sizeTotal);
}

void CDatabaseView::OnDraw(CDC* pDC)
{
    CDocument* pDoc = GetDocument();
    // TODO: add draw code here
}

/////////////////////////////////////
// CDatabaseView diagnostics
#ifdef _DEBUG
```

```
void CDatabaseView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CDatabaseView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

//获得指向当前文档对象指针的函数
CDrawDoc* CDatabaseView::GetDocument()    // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CDrawDoc)));
    return (CDrawDoc*)m_pDocument;
}

#endif //_DEBUG
```

(2) 创建一个菜单资源

在应用程序 Draw 运行过程中，当在主框架窗口中激活的是 CDatabaseView 视图时，因为需要在这个视图界面中完成与数据库管理有关的操作，所以需要给它定义一个另外的菜单资源。当应用程序 Draw 中激活的是 CDrawView 视图时，主框架使用固有的菜单资源 IDR_DRAWTYPE 形成的菜单，而当激活的是 CDatabaseView 视图时，主框架使用新建菜单资源形成的菜单。

可以利用 ResourceView 工具调用 AppStudio 可视化地创建菜单资源。在 ResourceView 工作区中的“Menu”项目上单击鼠标右键，通过选择快捷菜单项“Insert Menu”，进入增加菜单资源的可视化编辑界面，在这个编辑界面中编辑完成一个新的菜单资源。另外，因为新建菜单资源中应该包含主框架菜单 IDR_MAINFRAME 中的一些标准菜单项，也可以通过拷贝 IDR_MAINFRAME 菜单资源再修改的方法，可视化编辑修改得到一个菜单资源。

在应用程序 Draw 的资源中，创建一个名称为 IDR_DRAWDATA 的菜单资源。IDR_DRAWDATA 菜单资源的定义如下：

```
IDR_DRAWDATA MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Print...\tCtrl+P",        ID_FILE_PRINT
        MENUITEM "Print Pre&view",           ID_FILE_PRINT_PREVIEW
        MENUITEM "P&rint Setup...",          ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
```

```
        MENUITEM "Recent File",                 ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                       ID_APP_EXIT
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbar",                    ID_VIEW_TOOLBAR
        MENUITEM "&Status Bar",                 ID_VIEW_STATUS_BAR
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help Topics",                ID_HELP_FINDER
        MENUITEM SEPARATOR
        MENUITEM "&About draw...",              ID_APP_ABOUT
    END
END
```

菜单资源 IDR_DRAWDATA 中目前只包含了一些标准的操作菜单项,在本篇的以下章节中,会在这个菜单资源中加入新的菜单,用来激发实现数据库的各种操作功能。

(3) 创建数据库浏览视图

现在来完成创建 CDatabaseView 滚动视图的具体操作。以下是具体的操作步骤:

① 修改应用程序 Draw 的应用类 CDrawApp 的 InitInstance 函数,在函数中加入代码创建一个新的文档模板对象。以下是经过局部修改的 InitInstance 函数代码:

```
BOOL CDrawApp::InitInstance()
{
    ..... //省略以上代码
    DocTemplate1 = new CMultiDocTemplate(
        IDR_DRAWTYPE1,                  //注册所用的资源
        RUNTIME_CLASS(CDrawDoc),        //注册文档类
        RUNTIME_CLASS(CChildFrame),     //注册子框架类
        RUNTIME_CLASS(CDdatebaseView)); //注册视图类

    pDocTemplate = new CMultiDocTemplate(
        IDR_DRAWTYPE,                   //注册所用的资源
        RUNTIME_CLASS(CDrawDoc),        //注册文档类
        RUNTIME_CLASS(CChildFrame),     //注册子框架类
        RUNTIME_CLASS(CDrawView));      //注册视图类
    AddDocTemplate(pDocTemplate);       //增加一个文档模板对象
    ..... //省略以下代码
}
```

文档模板类指针 pDocTemplate 和 pDocTemplate1 在应用类 CDrawApp 中定义：

```
private:
    CDocTemplate * pDocTemplate1;
    CDocTemplate * pDocTemplate;
```

② 在应用类 CDrawApp 中定义两个成员函数，分别用来返回指向两个文档模板对象的指针。这两个函数在应用类 CDrawApp 中定义如下：

```
public:
    CDocTemplate * GetDocTemplate();
    CDocTemplate * GetDocTemplate1();
```

在应用类 CDrawApp 的实现文件 draw.cpp 中，加入这两个成员函数的实现代码：

```
CDocTemplate * CDrawApp::GetDocTemplate()
{
    return pDocTemplate;
}

CDocTemplate * CDrawApp::GetDocTemplate1()
{
    return pDocTemplate1;
}
```

③ 加入创建 CDatabaseView 视图的代码。为了创建 CDatabaseView 视图，需要在应用程序 Draw 中加入一个菜单项来控制视图的创建。在应用程序 Draw 的 IDR_MAINFRAME 和 IDR_DRAWTYPE 菜单资源的菜单“显示”下都加入一个菜单项“显示数据库视图”。增加完毕后，在资源文件 draw.rc 文件中的 IDR_MAINFRAME 和 IDR_DRAWTYPE 菜单资源的定义下都加入了一个菜单项(黑体)：

```
.....//省略以上代码
POPUP "&View"
BEGIN
    MENUITEM "&Toolbar",                ID_VIEW_TOOLBAR
    MENUITEM "&Status Bar",             ID_VIEW_STATUS_BAR
    MENUITEM "显示数据表视图",          ID_VIEW_DATABASE
END
.....//省略以下代码
```

在资源定义文件 resource.h 中加入了 ID 号值的定义：

```
#define ID_VIEW_DATABASE                32797
```

对于 ID 号 ID_VIEW_DATABASE，用 ClassWizard 在主框架类 CMainFrame 中增加一个消息处理函数 OnViewDatabase。在 CMainFrame 类中加入了消息处理函数的定义：

```
//{{AFX_MSG(CMainFrame)
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
```

```
afx_msg void OnViewDatabase();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

在 CMainFrme 类的实现文件 mainfrm.cpp 中，加入消息处理函数的映射机制和实现代码：

```
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ......//省略以上代码
    ON_COMMAND(ID_VIEW_DATABASE, OnViewDatabase)
    //}}AFX_MSG_MAP
    // Global help commands
    ON_COMMAND(ID_HELP_FINDER, CMDIFrameWnd::OnHelpFinder)
    ON_COMMAND(ID_HELP, CMDIFrameWnd::OnHelp)
    ON_COMMAND(ID_CONTEXT_HELP, CMDIFrameWnd::OnContextHelp)
    ON_COMMAND(ID_DEFAULT_HELP, CMDIFrameWnd::OnHelpFinder)
END_MESSAGE_MAP()

void CMainFrame::OnViewDatabase()
{
    // TODO: Add your command handler code here

}
```

对 OnViewDatabase 函数进行修改，以完成创建 CDatabaseView 视图的功能，修改后的函数如下：

```
void CMainFrame::OnViewDatabase()
{
    CMDIChildWnd * pActiveChild = MDIGetActive(); //得到指向活动的子框架窗口对象的指针
    CDocument * pDocument;
    if (pActiveChild == NULL ||
      (pDocument = pActiveChild->GetActiveDocument()) == NULL)
    //如果不存在子框架窗口对象或者不存在活动的文档对象，应用程序出错，给出提示信息退出
    {
        TRACE0("Warning: No active document for WindowNew command.\n");
        AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
        return;      // command failed
    }

    CDrawApp * pApp = (CDrawApp * )AfxGetApp();
    CDocTemplate * pTemplate = pApp->GetDocTemplate1();
    ASSERT_VALID(pTemplate);
    //利用当前活动的文档对象指针，调用 CreatNewFrame 函数创建一个子框架窗口对象和视图对象
    CFrameWnd * pFrame = pTemplate->CreateNewFrame(pDocument, pActiveChild);
```

```
    if (pFrame == NULL)     //如果创建子框架类对象不成功
    {
        TRACE0("Warning: failed to create new frame. \n");
        return;     // command failed
    }
    pTemplate->InitialUpdateFrame(pFrame, pDocument); //在屏幕上显示创建的视图
}
```

读者可参考 14.1.2 小节介绍的创建视图的步骤，理解 OnViewDatabase 函数代码的含义。

在对应用程序 Draw 进行了以上修改后，对其进行编译连接后运行，在“显示”菜单下运行菜单项“显示数据库视图”时，就会在主框架窗口中显示一个新的视图窗口，主框架窗口中的主菜单采用的是菜单资源 IDR_DRAWDATA（如图 14-1 所示）。在 CDatabaseView 视图操作界面下，目前尚不能进行任何功能操作。在本篇的以下各章节，将扩充应用程序 Draw 的功能，实现在 CDratabaseView 视图界面下完成各种数据库管理功能。

图 14-1　应用程序 Draw 的数据库浏览视图界面

14.2　创建一个 CRecordSet 派生类对象

通过 12.2 节的讨论可知，利用 ODBC 开发数据库应用程序时，是通过 CRecordSet 派生类对象来管理记录集合的。在关系数据库中，记录集合是通过一个或多个数据表的字段组成的，记录集合一旦建立，就可以通过 CRecordSet 类的成员函数采用同样的方法来对它进行管理（而不管记录集合来源于几个数据表）。为方便起见，在本书的讨论中，记录集合来源于单个数据表。

本节将实现利用 CDatabaseView 视图浏览记录集合的功能。

14.2.1　创建一个 ODBC 数据源

为了开发基于 ODBC 的数据库应用程序，首先需要利用一定的数据库系统创建数据库，并建立针对数据库的 ODBC 数据源。

(1) 组织数据表

在利用 ODBC 开发数据库应用程序时,可以利用来源于不同数据库的数据源,如可以是 FoxPro 的 .dbf 文件、Access 数据库(.mdb),以及 SQL Server 的数据库等。

在本书作为例子开发数据库应用程序时,假定在硬盘的某个目录下有一个 Access 的数据库 Info.mdb,在数据库中有一个数据表 city,数据表的结构如表 14-1 所示:

表 14-1 数据表结构

字段名称	数据类型	含 义
Only_id	int	识别号
Ciry_name	char[20]	城市名称
Population	value	人口数量
Area	value	面积
Industry	value	工农业产值
Tranelate	char[20]	交通状况
Envirument	char[20]	环境状况

在 GIS 系统中,用作存储性质数据的数据表,其数据结构上应该能够保证数据表的每条记录有一个唯一的识别号。也就是说,在数据表中应该有一个字段,对于不同的记录,这个字段具有唯一不同的值,即可以通过这个字段值唯一的确定这条记录。在不同的数据库系统中有不同的方法来完成这个要求:对于关系数据库,如 SQL Server,可以通过在数据表中定义一个 int 型字段,并给予这个字段以 identity 标志,对于这种类型的数据表,当增加一条记录时,SQL Server 自动为这个字段分配一个与其他字段不同的字段值。所以,在 SQL Server 数据库的数据表中,不需要应用程序进行干涉,就可以保证产生一个唯一的识别号;而对于其他数据库系统提供的数据源,如 Foxpro 的 .dbf 文件,则不能有这种类型的字段来实现这个功能,所以在增加记录时,需要计算出一个与既有的其他记录中这个字段不同的字段值,用这个值作为要增加记录相应的字段值。

在本书的讨论中,通过应用程序来为一条新增记录确定一个唯一的字段值,作为这条记录的唯一识别号。这种方法可以适用于采用任何类型数据源的情况。

(2) 创建一个 ODBC 数据源

为了能够利用 Access 数据库 Info.mdb 进行数据库应用程序设计,需要创建一个连接这个数据库的 ODBC 数据源。Access 数据库的 ODBC 数据源的创建方法在 11.2.2 小节已做了详细介绍,采用相同的方法,为 Info.mdb 创建一个 ODBC 数据源 GISDATA。

14.2.2 创建管理记录集合的 CRecordSet 派生类

对于 14.2.1 小节中创建的数据源 GISDATA,在应用程序 Draw 中创建一个 CRecordSet 类的派生类来对它进行管理。

(1) 创建 CRecordSet 派生类

在应用程序 Draw 中创建一个 CRecordSet 派生类的步骤如下：

① 在 VC++ 开发环境中打开应用程序 Draw，启动 ClassWizard 进入 MFC ClassWizard 对话框界面(如图 2-21 所示)，在 ClassWizard 操作界面中选择“Add Class...”按钮，进入到 Add Class 对话框(如图 2-11)，从“Base Class”列表框中选择 CRecordSet 类作为基类，在“Name”编辑框中输入要创建类的名称 CDataISet，保持默认的实现文件名称 dataIset.cpp 不变(如图 14-2 所示)。

图 14-2　新建类对话框

② 在图 14-2 中按下“OK”按钮，就会进入到图 14-3 所示的对话框，在对话框中选择“ODBC”选项，从“ODBC”边的列表框中选择 14.2.2 小节创建的 ODBC 数据源 GISDATA，在“Record Type”框中保持默认的选项 Snapshot 不变。

③ 在图 14-3 中按下“OK”按钮，就会进入到如图 14-4 所示的对话框，在对话框的列表框中选择数据表 city，按“OK”按钮，就会在应用程序 Draw 中产生一个新的类 CDataISet，类的头文件在 dataIset.h，实现文件在 dataIset.cpp。

(2) CDataISet 类的代码列表

对于以上建立的 CDataISet 类，在头文件 dataIset.h 中可以找到类的定义代码：

```
class CDataISet : public CRecordset
{
public:
    CDataISet(CDatabase * pDatabase = NULL);
    DECLARE_DYNAMIC(CDataISet)
```

图 14-3　选择数据源对话框



图 14-4　选择数据表对话框

```
// Field/Param Data
    //{{AFX_FIELD(CData1Set, CRecordset)
    long m_only_id;
    CString m_city_name;
    long m_population;
    doublem_area;
    doublem_industry;
    CString m_translate;
    CString m_enviroment;
    //}}AFX_FIELD

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CData1Set)
    public:
    virtual CString GetDefaultConnect();    // Default connection string
```

```
    virtual CString GetDefaultSQL();    // Default SQL for Recordset
    virtual void DoFieldExchange(CFieldExchange * pFX);    // RFX support
    //}}AFX_VIRTUAL

// Implementation
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
};
```

CData1Set 类的实现代码在实现文件 data1set.cpp 中。以下是实现文件 data1set.cpp 的代码列表：

```
#include "stdafx.h"
#include "draw.h"
#include "Data1Set.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////
// CData1Set

IMPLEMENT_DYNAMIC(CData1Set, CRecordset)

CData1Set::CData1Set(CDatabase * pdb)
    : CRecordset(pdb)
{
    //{{AFX_FIELD_INIT(CData1Set)
    m_only_id = 0;
    m_city_name = _T("");
    m_population = 0;
    m_area = 0.0;
    m_industry = 0.0;
    m_translate = _T("");
    m_enviroment = _T("");
    m_nFields = 7;
    //}}AFX_FIELD_INIT
    m_nDefaultType = snapshot;
}
```

```
CString CData1Set::GetDefaultConnect()
{
    return _T("ODBC;DSN = GISDATA");
}

CString CData1Set::GetDefaultSQL()
{
    return _T("[city]");
}

void CData1Set::DoFieldExchange(CFieldExchange * pFX)
{
    //{{AFX_FIELD_MAP(CData1Set)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[only_id]"), m_only_id);
    RFX_Text(pFX, _T("[city_name]"), m_city_name);
    RFX_Long(pFX, _T("[population]"), m_population);
    RFX_Double(pFX, _T("[area]"), m_area);
    RFX_Double(pFX, _T("[industry]"), m_industry);
    RFX_Text(pFX, _T("[translate]"), m_translate);
    RFX_Text(pFX, _T("[environment]"), m_enviroment);
    //}}AFX_FIELD_MAP
}

/////////////////////////////////////
// CData1Set diagnostics

#ifdef _DEBUG
void CData1Set::AssertValid() const
{
    CRecordset::AssertValid();
}

void CData1Set::Dump(CDumpContext& dc) const
{
    CRecordset::Dump(dc);
}
#endif //_DEBUG
```

(3) 组织记录集合

12.2.1 小节中介绍了组织记录集合的方法，可以利用这些方法在 CData1Set 类中重新组织

包含指定字段的记录集合。为简化起见，对代码不做修改，即把整个数据表 city 选做记录集合，通过 CDataISet 类创建的对象，就可以对整个数据表 city 进行管理。

14.2.3　创建记录集合对象

在 14.2.2 小节中，创建了一个管理数据表 city 的 CRecordSet 派生类 CDataISet。要实现对记录集合的具体管理功能，必须创建一个 CDataISet 对象。应用程序 Draw 是一个典型的使用视图/文档体系结构的应用程序，为了能够方便对记录集合的管理，将 CDataISet 对象定义在文档类 CDrawDoc 中。在文档类 CDrawDoc 中定义一个 CDataISet 对象：

```
public:
    CDataISet m_data1;
```

为了能够在文档类 CDrawDoc 中定义 CDataISet 对象，在头文件 drawdoc.h 中，将 CDataISet 类的头文件插入到 CDrawDoc 类的定义前：

```
#include "dataIset.h"
```

在文档类中创建的记录集合对象 m_data1，可以在 CDatabaseView 视图类中通过文档对象指针来得到，以实现通过视图对象管理文档对象中的记录集合对象。

14.3　实现数据库浏览视图

14.1 节在应用程序 Draw 中创建了一个由视图类 CDatabaseView 管理的滚动视图。14.2 节中创建了一个记录集合类 CDataISet，并且在文档类 CDrawDoc 中创建了一个记录集合对象 m_data1。本节对视图类 CDatabaseView 的功能进行扩充，使它能够对记录集合对象 m_data1 管理的记录集合进行浏览，实现一个数据库浏览视图。

14.3.1　建立视图与记录集合对象的联系

为了能够在 CDatabaseView 视图对象中操作在文档对象中定义的记录集合对象 m_data1，需要对视图类 CDatabaseView 进行如下修改：

① 在视图类 CDatabaseView 中定义一个 CDataISet 类指针：

```
privated:
    CDataISet * m_pSet1;
```

为了能够在视图类 CDatabaseSet 中使用 CDataISet 类，需要在头文件 databaseset.h 中 CDatabaseSet 类的定义前增加对 CDataISet 类的引用代码：

```
class CDataISet;
```

② 在视图类 CDatabaseSet 的 OnInitialUpdate 函数中，得到指向文档对象中的 CDataIset 对象 m_data1 的指针，修改后的 OnInitialUpdate 函数的代码如下所示：

```
void CDatabaseView::OnInitialUpdate()
{
```

```
    CScrollView::OnInitialUpdate();
    CDrawDoc * pDoc = GetDocument();    //得到指向当前文档对象的指针
    m_pSet1 = &pDoc->m_data1;           //得到指向记录集合对象的指针
    CSize sizeTotal;
    sizeTotal.cx = sizeTotal.cy = 100;
    SetScrollSizes(MM_TEXT, sizeTotal);
}
```

以上黑体部分代码,通过 GetDocument 函数得到指向当前视图对象所属的文档对象的指针 pDoc。通过指针 pDoc,将当前视图对象中的 CData1Set 类指针 m_pSet1 指向文档对象中的 CData1Set 对象 m_data1。

14.3.2　获得数据库浏览视图的滚动区域

在 7.7 节中对 MFC 应用程序固有的滚动视图进行了讨论。现在,将利用这个固有的滚动视图来实现一个数据库浏览视图。要实现数据库浏览视图,首先需要计算出滚动区域的大小,并设置滚动步长。

滚动区域的大小应该是保证正好显示记录集合全部记录的一个矩形区域。在数据库浏览视图内,记录集合的一条记录显示在一行上,所以滚动区域的高度与记录集合的记录数和每行记录的显示高度有关。滚动区域的宽度与记录集合每条记录的字符长度和每个字符的宽度有关。所以为了得到滚动区域的大小,需要确定以下参数。

(1) 字符的高度和宽度

字符的高度和宽度通过取得当前的字模参数来获得。以下代码用来获得字符的高度和宽度参数:

```
TEXTMETRIC tm;
CDC * dc = GetDC();                     //得到目前的设备描述对象
dc->GetTextMetrics(&tm);                //获得字模参数
ReleaseDC(dc);
nCharHeight = tm.tmHeight + 2;          //得到字符高度
nCharWidth = tm.tmAveCharWidth + 1;     //得到字符宽度
```

以上代码,将字符的高度和平均宽度分别加 2 和 1,得到在视图中显示记录时字符的高度和宽度。在纵向上,每行记录的显示高度即行的间隔为 nCharHeight;在横向上,每个字符的宽度为 nCharWidth。

(2) 记录集合的记录数和每条记录的字符宽度

为了确定视图滚动区域的大小,需要得到显示在滚动视图中的记录集合包含的记录数,同时也要得到每行记录的字符数。记录集合记录数的获得是通过调用 CRecordSet 类的相应函数来实现的(参见 12.2.8 小节)。可以在视图类 CDatabaseView 中使用以下代码来得到记录集合的记录数:

```
if(m_pSet1->IsOpen())          //如果打开了记录集合对象
    m_pSet1->Close();          //关闭记录集合
m_pSet1->Open();               //重新打开记录集合对象
if(m_pSet1->IsBOF())           //如果指针超出记录范围,则记录数为0
    nRecord=0;
else
{
    m_pSet1->MoveFirst();      //将记录指针移动到第一条记录
    nRecord=0;
    while(!m_pSet1->IsEOF())//直到记录指针指向记录范围外
    {
        m_pSet1->MoveNext()  //指向下一条记录
        nRecord++;           //记录数加1
    }
}
```

记录集合每条记录的字符长度是通过累加记录集合所包含的各个字段的字符宽度来实现。记录集合所包含的字段数目为：

```
nField=m_pSet1->GetODBCFieldCount();
```

记录集合中第 i 个字段的信息，可以通过成员函数 GetODBCFieldInfo 来得到。以下代码得到记录集合中第 i 个字段的字段信息：

```
CODBCFieldInfo fieldinfo1;
CODBCFieldInfo& fieldinfo=fieldinfo1;
m_pSet1->GetODBCFieldInfo(i,fieldinfo);
```

字段信息返回到 CODBCFieldInfo 结构 fieldinfo1 中，CODBCFieldInfo 结构中有一个成员变量 m_nPrecision，用来记录字段的字符长度，所以第 i 个字段的字符长度为：

```
FieldWidth=fieldinfo1.m_nPrecision;
```

而一条记录的字符宽度可以通过累加所有字段的字符宽度来得到。

(3) 滚动视图区域的高度和宽度

在已知记录集合的记录数和记录集合中各个字段的字符宽度，并已知视图中每行记录的显示高度和每个字符平均宽度的情况下，可以得到视图滚动区域的高度和宽度：

滚动区域的高度 = 每行记录的显示高度 × 记录集合的记录数
滚动区域的宽度 = 每个字符的宽度 × 每条记录的字符宽度

(4) 滚动区域的限制

受系统容量的限制，视图滚动区域的大小不是任意的。视图滚动区域的高度和宽度最多为 16 位整数值 32767(逻辑单位)，所以在组织有大量记录的记录集合的滚动视图时，需要加入

一定的代码,处理视图实际滚动区域大于最大允许区域的情况。

14.3.3 实现数据库浏览视图

现在对应用程序 Draw 中的视图类 CDatabaseView 进行修改,来实现对记录集合对象 m_data1 的浏览视图。

(1) 在 CDatabaseView 类中定义变量

首先在视图类 CDatabaseView 中定义实现浏览视图所需要的变量:

```
private:
    short nCharHeight,nCharWidth;        //字体的高度和宽度
    int nRecord,nField;                  //记录集合的记录数和字段数
    int nScreenWide,nScreenHeight;       //视图的宽度和高度
    short nFieldWide[200];               //每个字段的显示宽度
    short nFieldStart[200];              //每个字段开始显示的位置
    CSize sizeTotal;                     //用来记录滚动区域的大小
    CSize sizePage;                      //用来记录滚动页的大小
    CSize sizeLine;                      //用来记录滚动行的大小
```

在以上定义的变量中,nFiledWide 数组用来记录每个字段的显示宽度(逻辑坐标),而 nFieldStart 数组用来记录每个字段在横向上的起始显示位置。

(2) 获得视图横向和纵向参数

在视图类 CDatabaseView 中,定义两个函数 Init1 和 Init2,分别用来得到视图横向或纵向有关的参数:

```
public:
    void Init1();
    void Init2();
```

在实现文件 dbaseview.cpp 中,加入这两个函数的实现代码:

```
void CDatabaseView::Init1()
{
    TEXTMETRIC tm;
    CDC * dc = GetDC();                          //得到指向设备描述对象的指针
    dc->GetTextMetrics(&tm);                     //得到字模参数
    ReleaseDC(dc);
    nCharHeight = tm.tmHeight + 2;               //得到每行记录所占的高度
    nCharWidth = tm.tmAveCharWidth + 2;          //得到每个字符的宽度
    if(m_pSet1->IsOpen())                        //如果打开了记录集合对象
        m_pSet1->Close();                        //关闭记录集合对象
    m_pSet1->Open();                             //打开记录集合对象
```

```
    nField = m_pSet1 -> GetODBCFieldCount();    //得到记录集合包含的字段数目
    CODBCFieldInfo fieldinfo1;
    CODBCFieldInfo& fieldinfo = fieldinfo1;
    for(int i = 0;i < nField;i++)                //得到记录集合各个字段的参数
    {
        m_pSet1 -> GetODBCFieldInfo(i,fieldinfo);        //得到第 i 个字段的信息
        nFieldWide[i] = (short)fieldinfo1.m_nPrecision;    //得到第 i 个字段的宽度
    }
    nScreenWide = 0;
    nFieldStart[0] = 0;
    for(i = 0;i < nField;i++)                //得到记录集合各个字段的起始显示位置
    {
        nScreenWide = nScreenWide + nFieldWide[i] * nCharWidth;
        nFie ldStart[i + 1] = nFieldStart[i] + nFieldWide[i] * nCharWidth;
    }
    sizeTotal.cx = nScreenWide;                //得到滚动区域的宽度
    sizeLine.cx = nCharWidth * 5;              //得到横向行滚动距离
}
```

Init1 函数用来获得滚动视图横向的各个参数，具体实现步骤可参考代码中的详细注释。

```
void CDatabaseView::Init2()
{
    if(m_pSet1 -> IsOpen())                  //如果打开了记录集合对象
        m_pSet1 -> Close();                  //关闭记录集合对象
    m_pSet1 -> Open();                       //打开记录集合对象
    if(m_pSet1 -> IsBOF())                   //如果记录指针指向记录集合范围外，即记录集合不
                                             //包含记录
        nRecord = 0;                         //记录集合的记录数为 0
    else                                     //如果记录集合包含记录，得到记录数
    {
        m_pSet1 -> MoveFirst();              //指向第一条记录
        nRecord = 0;
        while(! m_pSet1 -> IsEOF())          //直到指向最后一条记录范围外
        {
            m_pSet1 -> MoveNext();           //移动记录指针到下一条记录
            nRecord ++ ;                     //记录数加 1
        }
    }
    nScreenHeight = nRecord * nCharHeight;   //得到滚动区域的高度
    if(nScreenHeight = = 0)                  //如果滚动区域高度为 0，设置为 20
        nScreenHeight = 20;
    sizeTotal.cy = nScreenHeight;            //设置滚动区域的高度
    sizeLine.cy = nCharHeight + 2;           //得到行纵向滚动区域，即最小一次滚动一条记录
```

```
}
```

Init2 函数用来获得滚动视图纵向的各个参数，具体实现步骤可参考代码中的详细注释。

(3) 修改 OnInitialUpdate 函数

对视图类 CDatabaseView 的 OnInitialUpdate 函数进行修改，使之能够根据记录集合的具体内容设置滚动区域，修改后的函数代码如下：

```
void CDatabaseView::OnInitialUpdate()
{
    CDrawDoc * pDoc = GetDocument();      //得到指向文档对象的指针
    m_pSet1 = &pDoc->m_data1;             //将指针 m_pSet1 指向文档对象中的记录集合对象
    Init1();                              //设置视图的横向滚动参数
    Init2();                              //设置视图的纵向滚动参数
    CScrollView::OnInitialUpdate();
    SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);      //设置滚动区域的大小
}
```

(4) 增加和修改 OnSize 消息处理函数

理想情况，以视图窗口的大小作为滚动视图的页滚动区域。为了适应视图窗口大小的变化，用 ClassWizard 在视图类 CDatabaseView 中增加窗口消息 WM_SIZE 的消息处理函数 OnSize，OnSize 函数在视图窗口大小发生变化时被调用。增加 OnSize 函数引起以下相关的代码变化：

① 在视图类 CDtatbaseView 中增加了消息处理函数的定义：

```
//{{AFX_MSG(CDatabaseView)
afx_msg void OnSize(UINT nType, int cx, int cy);
//}}AFX_MSG
```

② 在实现文件 dbaseview.cpp 中增加了映射机制：

```
BEGIN_MESSAGE_MAP(CDatabaseView, CScrollView)
    //{{AFX_MSG_MAP(CDatabaseView)
    ON_WM_SIZE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

③ 在实现文件 dbaseview.cpp 中增加了函数的实现代码。对函数的实现代码进行修改，实现设置滚动视图参数的功能。修改后的函数代码如下：

```
void CDatabaseView::OnSize(UINT nType, int cx, int cy)
{
    CScrollView::OnSize(nType, cx, cy);
    sizePage.cx = cx;                                    //设置页滚动区域的宽度
    sizePage.cy = cy;                                    //设置页滚动区域的高度
    SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);      //设置视图滚动范围
```

```
}
```

(5) 修改 OnDraw 函数

视图中显示的内容是通过视图类中的函数 OnDraw 来绘制的。修改 OnDraw 函数，对数据库浏览视图的内容进行绘制。修改后的 OnDraw 函数代码如下：

```
void CDatabaseView::OnDraw(CDC * pDC)
{
    CDrawDoc * pDoc = GetDocument();          //得到指向当前文档对象的指针
    pDC->SetTextColor(RGB(255,0,0));          //设置文本颜色
    CString c1;
    CString& c2 = c1;
    pDC->SetMapMode(1);                       //设置 MM_TEXT 坐标映像方式
    m_pSet1->MoveFirst();                     //移动记录指针到记录集合的第一条记录
    for(int i = 0;i < nRecord;i++)            //依次绘制记录集合的所有记录
    {
        for(int j = 0;j < nField;j++)         //依次绘制一条记录的各个字段
        {
            m_pSet1->GetFieldValue(j,c2); //得到第 j 个字段的字符串
            pDC->TextOut(nFieldStart[j],i * nCharHeight,c1);   //在指定位置绘制字段的内容
        }
        m_pSet1->MoveNext();                  //将记录指针指向记录集合的下一条记录
    }
}
```

在 OnDraw 函数中，通过 CRecordSet 类的成员函数 GetFieldValue 来得到当前记录指定字段的内容，函数的第一个参数是字段在记录集合中的 0 基序列号，第二个参数是一个 CString& 变量，通过它返回字段的字符串内容。这种读取字段内容的方法不使用记录集合类的成员变量，有利于对所有的记录集合对象通用。调用 CDC 类的 TextOut 函数将字符串写到视图中。第 i 条记录第 j 个字段的横向起点位置通过数组 nFieldStart 记录的值来确定，纵向起点位置是 i * nCharHeight。

对应用程序 Draw 的源代码做以上修改后，编译运行应用程序，当运行“显示”菜单下的菜单项“显示数据浏览视图”时，就会得到如图 14-5 所示的运行界面。数据表 city 的内容显示到了滚动视图中，通过横向和纵向滚动条可以对数据表进行浏览。

本章实现的浏览视图，只能对 CDaa1Set 类对象 m_data1 管理的记录集合对象进行浏览，即浏览的记录集合是固定的。要浏览另外一个记录集合的内容时，需要对程序代码做较大的修改：在 CDatabaseView 类中定义另外一个记录集合类指针，在 CDatabaseView 的实现代码中，在多处要对采用哪个记录集合指针进行选择等。这种方法实现起来非常麻烦，不利于用浏览视图来浏览更多的记录集合。在 17 章“通用数据库排序功能”中，将介绍通过建立一个记录集合抽象类，通过抽象类指针来实现对所有记录集合对象的通用浏览功能。

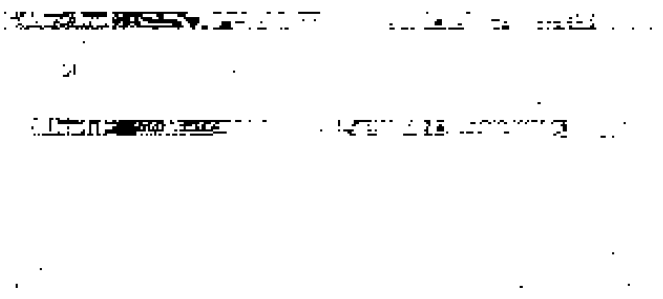


图 14-5　应用程序 Draw 的数据库浏览视图

第 15 章　实现数据库编辑功能

建立数据库浏览视图的目的，一方面是为了浏览记录集合，而另一个更重要的方面，是在这个浏览视图的基础上，去组织实现数据库的其他操作功能，如记录的增加、删除、修改功能，数据库过滤(查询)和排序功能等。本章将实现对记录集合记录的增加、记录修改和记录删除等操作功能。

15.1　在数据库浏览视图中建立记录指针

要对记录集合的记录进行删除、修改等操作时，首先需要确定对记录集合的哪条记录进行操作。这就需要在数据库浏览视图中建立一个记录指针，来确定当前选择的是哪条记录。

在应用程序 Draw 实现的数据库浏览视图中，按照以下的方法来操作记录指针：在数据库浏览视图中，记录指针指向的记录用反色显示。用鼠标在视图中点中另外一条记录时，就将记录指针指向这条记录，并反色显示这条记录。以下对视图类 CDatabaseView 进行修改，在数据库浏览视图中实现记录指针的功能。

(1) 建立并初始化存储记录指针的变量

记录集合中记录指针的位置可以通过一个整型变量来记录，记录集合的第一条记录是 1，第二条记录是 2，...。在视图类 CDtatabaseView 中定义一个变量：

```
protected:
    int nIndex;    //当前指向的记录序号
```

在视图类 CDatabaseView 的构造函数中，加入对这个变量的初始化代码：

```
CDatabaseView::CDatabaseView()
{
    nIndex = -1;
    sizeTotal.cx = sizeTotal.cy = 300;
    sizePage.cx = sizePage.cy = 300;
    sizeLine.cx = sizeLine.cy = 10;
}
```

将 nIndex 设置为 -1，指的是将记录指针指向记录集合外，即不指向记录集合的任何记录。这样做的目的是避免记录集合为空时，记录指针还指向一定序号的记录，引起操作错误。

在视图类 CDatabaseView 的计算视图区域纵向参数的函数 Init2 中，加入代码对存储记录指针序号的变量 nIndex 进行初始化(黑体代码)：

```
void CDatabaseView::Init2()
{
```

```
    if(m_pSet1->IsOpen())          //如果打开了记录集合对象
        m_pSet1->Close();          //关闭记录集合对象
    m_pSet1->Open();               //打开记录集合对象
    if(m_pSet1->IsBOF())           //如果记录指针指向记录集合范围外,即记录集合不包含记录
        nRecord=0;                 //记录集合的记录数为0
    else                           //如果记录集合包含记录,得到记录数
    {
        m_pSet1->MoveFirst();      //指向第一条记录
        nRecord=0;
        while(!m_pSet1->IsEOF())//直到指向最后一条记录范围外
        {
            m_pSet1->MoveNext(); //移动记录指针到下一条记录
            nRecord++;             //记录数加1
        }
    }
    nScreenHeight=nRecord*nCharHeight;   //得到滚动区域的高度
    if(nScreenHeight==0)                 //如果滚动区域高度为0,设置为20
        nScreenHeight=20;
    sizeTotal.cy=nScreenHeight;          //得到滚动区域的高度
    sizeLine.cy=nCharHeight+2;           //得到纵向行滚动区域,即最小一次滚动一条记录
    if(nRecord>0)                        //如果记录集合的记录数大于0
    {
        if(nIndex<1||nIndex>Record)      //如果记录指针指向记录集合的范围外
            nIndex=1;                    //将记录指针指向第一条记录
    }
    else                             //如果记录集合的记录数等于0,将记录指针指向记录集合外
        nIndex=-1;
}
```

以上黑体代码实现,当记录集合的记录数 nRecord 大于 0 时,在记录指针指向记录集合范围外时(nIndex < 1 || nIndex > Record),将记录指针指向第一条记录;当记录集合的记录数 nRecord 等于 0 时,将记录指针记录集合外(nIndex = -1)。

(2) 在浏览视图上反写记录集合指向的记录

为了在数据库浏览视图中,对记录指针指向的记录用反色显示,我们需要对视图类 CDatabaseView 的函数 OnDraw 进行修改。以下是修改后的 OnDraw 函数代码:

```
void CDatabaseView::OnDraw(CDC* pDC)
{
    CDrawDoc* pDoc = GetDocument();   //得到指向当前文档对象的指针
    CPen pen1(0,1,RGB(255,0,0));      //创建一个画笔对象
    pDC->SetTextColor(RGB(255,0,0)); //设置文本颜色
    CString c1;
```

```
    CString& c2 = c1;
    pDC - > SetMapMode(1);                    //设置 MM _ TEXT 坐标映像方式
    m _ pSet1 - > MoveFirst();                //移动记录指针到记录集合的第一条记录
    for(int i = 0;i < nRecord;i++ )           //依次绘制记录集合的所有记录
    {
        for(int j = 0;j < nField;j++ )        //依次绘制一条记录的各个字段
        {
            m _ pSet1 - > GetFieldValue(j,c2);      //得到当前记录第 j 个字段内容的字符串
            pDC - > TextOut(nFieldStart[j],i * nCharHeight,c1); //在指定位置绘制字段的内容
        }
        m _ pSet1 - > MoveNext();             //将记录指针指向记录集合的下一条记录
    }
    CRect r1;
    if(nIndex > 0)                            //如果记录指针指向记录集合中的记录
    {
        GetRect(nIndex,&r1);                  //得到第 nIndex 条记录在浏览视图中显示的矩形坐标
        pDC - > InvertRect(&r1);              //反色显示记录指针指向记录的显示区域
    }
}
```

以上黑体部分的代码用来在浏览视图中反色显示记录指针指向的记录，函数 GetRect 用来得到第 nIndex 条记录在浏览视图中的显示区域。GetRect 函数在视图类 CDatabaseView 中定义如下：

```
public:
    void GetRect(int nRow,CRect * r1);
```

GetRect 函数用来得到第 nRow 条记录在视图中显示的矩形区域，矩形区域坐标通过 CRect 指针 r1 返回。在实现文件 dbaseview.cpp 中，加入函数的实现代码：

```
void CDatabaseView::GetRect(int nRow, CRect * r1)
{
    r1 - > left = 0;                          //显示区域的起点横坐标
    r1 - > right = nFieldStart[nField - 1] + nFieldWide[nField - 1] * nCharWidth; //区域终点横坐标
    r1 - > top = (nRow - 1) * nCharHeight;    //显示区域的起点纵坐标
    r1 - > bottom = r1 - > top + nCharHeight; //显示区域的终点纵坐标
}
```

(3) 修改记录集合的记录指针

为了用鼠标在浏览视图中修改记录指针(将记录指针指向其他记录)，在视图类 CDatabaseView 中增加按下鼠标左键消息(WM _ LBUTTONDOWN)的处理函数 OnLButtonDown(增加鼠标消息处理函数的操作参考 6.1.1 小节)，对这个消息处理函数进行修改，得到修改后的实现代码如下：

```
void CDatabaseView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect r1;
    CClientDC ht(this);
    OnPrepareDC(&ht);
    ht.DPtoLP(&point);                          //将设备坐标转成逻辑坐标
    int nIndex1 = point.y/nCharHeight + 1;      //得到记录指针的序号
    if(nIndex1!  = nIndex&&nIndex1 > 0&&nIndex1 < = nRecord)
    {
        if(nIndex > 0)                          //如果原来记录指针指向记录范围内
        {
            GetRect(nIndex,&r1);                //得到记录的显示区域
            ht.InvertRect(&r1);                 //反色显示记录区域
        }
        GetRect(nIndex1,&r1);                   //得到记录的显示区域
        ht.InvertRect(&r1);                     //反色显示记录区域
        nIndex = nIndex1;                       //记录下记录指针
    }
    CScrollView::OnLButtonDown(nFlags, point);
}
```

OnLButtonDown 函数的工作步骤为：

① 将鼠标点 point 的设备坐标转换成逻辑坐标。因为浏览视图是通过调整视口原点来实现滚动的，所以即使采用的是 MM _ TEXT 坐标映像方式，设备坐标与逻辑坐标也是不一样的。

② 通过鼠标按中点的逻辑坐标，计算得到鼠标点在视图中点中记录的序号 nIndex1。

③ 如果点中记录的序号 nIndex1 在记录集合范围内(nIndex1 > 0&&nIndex1 < = nRecord)，且不是原来记录指针指向的记录(nIndex1! = nIndex)，则进行如下的两步操作：

④ 如果原来记录指针 nIndex 指向记录集合范围内，则在视图中对这个记录所在的矩形区域反色显示，以消除原来的反写标志。

⑤ 反色显示当前点中的记录所在的矩形区域，并将记录记录指针位置的变量 nIndex 指向鼠标点中记录的序号 nIndex1。

对应用程序 Draw 的代码进行以上修改，然后编译并运行时，在数据库浏览视图内就可以用鼠标来改变记录指针。**必须注意的是，在本节的讨论中，只是通过一个变量 nIndex 记录了当前记录在记录集合中的序号，而没有将记录集合对象 m _ pSet1 真正指向这条记录。**这样做的目的是为了避免对记录集合进行无谓的操作，加快操作速度。只有在对记录集合进行具体的操作前，如删除、修改当前记录时，才将记录集合中的记录指针真正指向实际的记录。

15.2　建立数据库编辑功能操作菜单

为了能够在数据库浏览视图中对记录集合实现诸如增加记录、修改记录和删除记录等操作，需要在视图类 CDatabaseView 所采用的菜单资源 IDR _ DRAWDATA 中加入一些菜单项，并

在视图类 CDatabaseView 中建立针对这些菜单项的消息处理函数。

15.2.1　修改菜单资源

在 VC++ 开发环境中，用 ResourceView 选择并激活 IDR _ DRAWDATA 菜单资源，进入到可视化编辑状态。在这个菜单资源中增加一个“记录操作”菜单，此菜单下包含“增加记录”、“修改记录”、“删除记录”、“删除所有记录”等 4 个菜单项（见图 15-1 所示）。

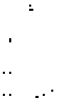


图 15-1　IDR _ DRAWDATA 菜单资源的记录操作菜单

可视化编辑完毕后，在资源文件 draw.rc 中可以找到增加的菜单项描述代码：

```
IDR _ DRAWDATA MENU PRELOAD DISCARDABLE
BEGIN
    ......//省略以上代码
    POPUP "记录操作"
    BEGIN
        MENUITEM "增加记录...",          ID _ RECORD _ ADD
        MENUITEM "修改记录...",          ID _ RECORD _ MODIFY
        MENUITEM "删除记录...",          ID _ RECORD _ DELETE
        MENUITEM "删除所有记录",         ID _ RECORD _ DELETE _ ALL
    END
    ......//省略以下代码
END
```

在资源 ID 号定义文件 resource.h 中，增加了对新增菜单 ID 号值的定义：

```
# define ID _ RECORD _ ADD              32792
# define ID _ RECORD _ MODIFY           32793
# define ID _ RECORD _ DELETE           32794
# def ine ID _ RECORD _ DELETE _ ALL    32795
```

15.2.2　建立操作菜单消息处理函数

对于 15.2.1 小节中在菜单资源 IDR _ DRAWDATA 中增加的菜单项，用 ClassWizard 在视图类 CDatabaseView 中为这些菜单项增加消息处理函数。以下是增加消息处理函数后，程序代码所作的改变：

① 在视图类 CDatabaseView 中定义了消息处理函数：

```
//{{AFX _ MSG(CDatabaseView)
afx _ msg void OnSize(UINT nType, int cx, int cy);
```

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnRecordAdd();
afx_msg void OnRecordModify();
afx_msg void OnRecordDelete();
afx_msg void OnRecordDeleteAll();
//}}AFX_MSG
```

② 在视图类 CDatabaseView 的实现文件 dbaseview.cpp 中,增加了消息映射机制和空的函数实现代码:

```
BEGIN_MESSAGE_MAP(CDatabaseView, CScrollView)
    //{{AFX_MSG_MAP(CDatabaseView)
    ON_WM_SIZE()
    ON_WM_LBUTTONDOWN()
    ON_COMMAND(ID_RECORD_ADD, OnRecordAdd)
    ON_COMMAND(ID_RECORD_MODIFY, OnRecordModify)
    ON_COMMAND(ID_RECORD_DELETE, OnRecordDelete)
    ON_COMMAND(ID_RECORD_DELETE_ALL, OnRecordDeleteAll)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CDatabaseView::OnRecordAdd()
{
    // TODO: Add your command handler code here
}

void CDatabaseView::OnRecordModify()
{
    // TODO: Add your command handler code here
}

void CDatabaseView::OnRecordDelete()
{
    // TODO: Add your command handler code here
}

void CDatabaseView::OnRecordDeleteAll()
{
    // TODO: Add your command handler code here
}
```

本章以下的各节中,将对这些消息处理函数进行修改,实现增加记录、修改记录、删除记录以及删除全部记录等操作功能。

15.3　建立编辑数据记录的对话框类

在 12.4.3 小节介绍了利用 CRecordView 视图来编辑数据记录的实现过程，分析了各种编辑功能的实现步骤。本节利用模式对话框来对记录集合的一条记录进行交互编辑。要求实现如下的操作步骤：在 CDatabaseView 视图中对记录集合进行浏览时，选择“记录操作”菜单的“增加记录”菜单项可以调出一个对话框，利用对话框中的各个控制输入数据记录各个字段的值，即通过这个对话框对数据记录进行交互修改，当按“OK”按钮退出对话框时，会在记录集合中利用对话框中输入的字段值增加一条记录；当选择运行“记录操作”菜单下的“修改记录”菜单项时，调用同一个对话框，对记录集合中记录指针指向的记录进行交互编辑修改。

用模式对话框对记录集合的一条记录进行编辑，其实现思路和过程与用 CRecordView 视图相同(参见 12.4.3 小节)。下面结合具体的实现代码，来讨论用模式对话框交互编辑数据记录的实现过程。

15.3.1　建立对话框资源

为了实现交互编辑数据记录的对话框，首先需要创建一个对话框资源。在对话框资源中建立针对记录集合各个字段的对话框控制。一般来说，对于不同的记录集合对象，需要提供不同的编辑数据记录的对话框界面，也就需要建立专门针对要编辑的记录集合对象的对话框资源。

为了编辑 CData1Set 对象管理的记录集合的数据记录，可以建立一个如图 15-2 所示的对话框资源，对话框的 ID 号为 IDD_DATA1。完成对话框资源的编辑后，可以在资源文件 draw.rc 中找到描述这个对话框资源的代码：

图 15-2　IDD_DATA1 对话框资源

```
IDD_DATA1 DIALOG DISCARDABLE 0, 0, 262, 124
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON   "Y 保存退出",IDOK,134,97,50,14
    PUSHBUTTON      "N 放弃退出",IDCANCEL,193,97,50,14
```

```
    LTEXT           "标识号",IDC_STATIC,7,19,27,10
    EDITTEXT        IDC_EDIT1,36,18,40,12,ES_AUTOHSCROLL | ES_READONLY
    LTEXT           "城市名称",IDC_STATIC,108,18,38,12
    LTEXT           "人口数量",IDC_STATIC,6,40,38,12
    LTEXT           "城市面积",IDC_STATIC,108,38,38,12
    LTEXT           "工农业产值",IDC_STATIC,7,59,40,12
    LTEXT           "交通状况",IDC_STATIC,107,60,38,10
    EDITTEXT        IDC_EDIT2,149,17,69,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT3,52,38,49,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT4,149,38,54,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT5,53,58,48,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_EDIT6,148,58,71,12,ES_AUTOHSCROLL
    LTEXT           "环境状况",IDC_STATIC,7,78,38,10
    EDITTEXT        IDC_EDIT7,53,76,57,12,ES_AUTOHSCROLL
END
```

将对话框资源中的编辑框 IDC_EDIT1 设置成只读状态(ES_READONLY)。因为这个控制将来用来输入记录集合中的字段 only_id。only_id 记录的是数据记录的唯一识别号,它的值通过计算得到而非手工输入。

同样,在 ID 号定义文件 resource.h 中,增加了 IDD_DATA1 对话框资源及其各个对话框控制 ID 号的定义代码。

15.3.2　创建编辑数据记录的对话框类

用 ClassWizard 创建对话框类的步骤在 6.4.4 小节已做了介绍。采用相同的步骤在应用程序 Draw 中创建一个对话框类 CData1Dlg,这个对话框类采用对话框资源 IDD_DATA1,头文件在 data1dlg.h,实现文件在 data1dlg.cpp。用 ClassWizard 创建对话框类 CData1Dlg 后,在头文件 data1dlg.h 中,可以找到对话框类 CData1Dlg 的定义:

```
class CData1Dlg : public CDialog
{
// Construction
public:
    CData1Dlg(CWnd* pParent = NULL); // standard constructor
// Dialog Data
    //{{AFX_DATA(CData1Dlg)
    enum { IDD = IDD_DATA1 };
    //}}AFX_DATA

protected:
    BOOL b_IsAdd;
    int m_id;
    // Overrides
    // ClassWizard generated virtual function overrides
```

```
    //{{AFX_VIRTUAL(CData1Dlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(CData1Dlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

在实现文件 data1dlg.cpp 中，可以找到对话框类 CData1Dlg 的实现代码：

```
CData1Dlg::CData1Dlg(CWnd* pParent /* = NULL */)
    : CDialog(CData1Dlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CData1Dlg)
    //}}AFX_DATA_INIT
}

void CData1Dlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CData1Dlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CData1Dlg, CDialog)
    //{{AFX_MSG_MAP(CData1Dlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////
// CData1Dlg message handlers
```

15.3.3　设置对话框类的外部对象引用

用对话框编辑记录集合数据记录的过程，与在 12.3 节讨论的用 CRecordView 视图管理数据记录相同：通过在对话框类中创建映射变量，建立对话框中的对话框控制与记录集合对象（如文档对象中的 m_data1 对象）中的映射变量间的联系，实现两者间的数据交换。而记录集合对象中的映射变量又与记录集合中的字段建立联系，实现两者间的数据交换，这样就实现了对话框中的对话框控制与记录集合中当前记录各个字段之间的联系，最终实现通过对话框控

制来修改记录集合中当前记录各个字段的目的，并通过记录集合对象对实际的数据源进行修改。

在对话框类中，必须能够获得和使用要进行编辑的记录集合对象。这项工作，是通过在对话框类中建立外部引用对象来实现的。

(1) 给对话框类 CData1Dlg 建立外部引用类

利用 ClassWizard 为 CData1Dlg 对话框类建立外部引用类的步骤：启动 ClassWizard，进入 ClassWizard 对话框，在“Class Name”列表框中选择类 CData1Dlg，在对话框的顶部选择“Class Info”标签，进入到“Class Info”操作界面。从操作界面的“Foreign Class”列表框中选择类 CData1Set(原来的选项是 None)，在“Foreign Varible”编辑框中输入变量名称 m_pSet。输入完毕后的操作界面如图 15-3 所示。

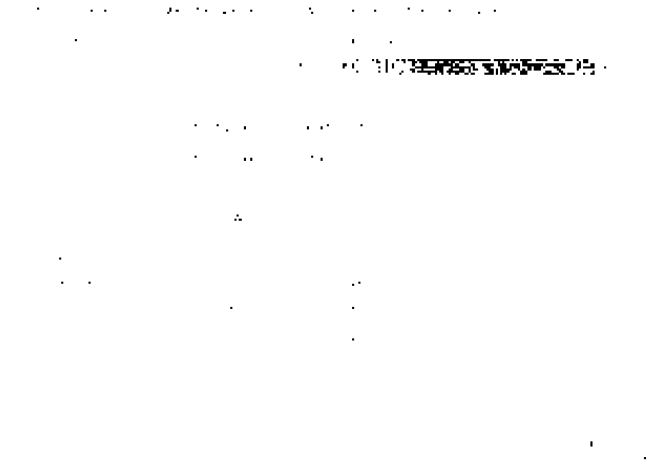


图 15-3 MFC ClassWizard 对话框 Class Info 操作界面

在图 15-3 对话框界面中，按“OK”按钮退出 ClassWizard 对话框，就会在对话框类 CData1Dlg 中增加一个外部引用类指针。以上操作对代码修改如下：

① 在对话框类 CData1Dlg 中定义了一个 CData1Set 类指针：

```
//{{AFX_DATA(CData1Dlg)
enum { IDD = IDD_DATA1 };
Data1Set* m_pSet;
//}}AFX_DATA
```

类的名称和指针的名称是在图 15-3 所示的操作界面上输入的。

② 在对话框类 CData1Dlg 的构造函数中加入了类指针的初始化代码：

```
CData1Dlg::CData1Dlg(CWnd* pParent /* = NULL */)
```

```
    : CDialog(CData1Dlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CData1Dlg)
    m_pSet = NULL;
    //}}AFX_DATA_INIT
}
```

当然,可以直接用手工(而不是用 ClassWizard)来完成以上两个步骤的代码改动,在对话框类 CData1Dlg 中设置对 CData1Set 对象的外部引用类指针。

以上操作在对话框类 CData1Dlg 中定义了一个 CData1Set 类指针,为了能够在对话框类 CData1Dlg 中识别类 CData1Set,可在对话框类 CData1Dlg 的定义前增加如下代码:

```
class CData1Set;
```

(2) 外部引用类指针的初始化

(1)中在对话框类 CData1Dlg 中增加了一个 CData1Set 类指针 m_pSet。不做任何改动时,这个指针指向 NULL,没有指向具体的 CData1Set 对象。实现用对话框编辑记录集合的数据记录时,需要对这个指针进行初始化,使其指向要管理的记录集合对象,即 CData1Set 对象。在 15.4 节实现增加及修改数据记录功能时,将介绍利用对话框类 CData1Dlg 的构造函数对这个指针进行初始化的方法。

15.3.4　建立对话框控制与记录集合类的映射变量的映射联系

为了能够利用对话框的各个对话框控制来修改记录集合当前记录的各个字段,需要建立对话框控制与记录集合类 CData1Set 中的映射变量间的映射关系。具体的实现过程可参考 12.3.3小节。以下是建立一个映射的实现过程:

启动 ClassWizard,进入到 ClassWizard 对话框,在"Class Name"列表框中选择类 CData1Dlg,在对话框顶部选择"Member Variables"标签,就会进入到图 15-4 所示的对话框。在"Contol IDs"列表框中列出了对话框中各个控制的 ID 号及其连接的映射变量的类型和名称,选择 IDC_EDIT1 控制,按"Add Variable..."按钮,就会进入到图 15-5 所示的对话框,从"Member Variable names"列表编辑框中选择变量 m_pSet->m_only_id,按"OK"按钮退出对话框,就会建立编辑框 IDC_EDIT1 与 CData1Set 类的映射变量 m_only_id 间的映射关系。

因为在 15.3.3 小节中为对话框类 CData1Dlg 建立了外部引用类 CData1Set 指针,所以 ClassWizard 能够在图 15-5 所示的"Member Variable names"的列表框中列出 CData1Set 类的映射变量。

与给编辑框 IDC_EDIT1 增加(实际上是连接)一个映射变量 m_pSet->m_only_id 的步骤相同,可以为其他的对话框控制连接相应的映射变量。增加完毕后,在对话框类 CData1Dlg 的成员函数 DoDataExchange 中,增加了各个宏代码:

```
void CData1Dlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
```

图 15-4　MFC ClassWizard 对话框 Member Variables 操作界面

图 15-5　增加对话框控制的映射变量对话框

```
//{{AFX_DATA_MAP(CData1Dlg)
DDX_FieldText(pDX, IDC_EDIT1, m_pSet->m_only_id, m_pSet);
DDX_FieldText(pDX, IDC_EDIT2, m_pSet->m_city_name, m_pSet);
DDX_FieldText(pDX, IDC_EDIT3, m_pSet->m_population, m_pSet);
DDX_FieldText(pDX, IDC_EDIT4, m_pSet->m_area, m_pSet);
DDX_FieldText(pDX, IDC_EDIT5, m_pSet->m_industry, m_pSet);
DDX_FieldText(pDX, IDC_EDIT6, m_pSet->m_translate, m_pSet);
DDX_FieldText(pDX, IDC_EDIT7, m_pSet->m_enviroment, m_pSet);
//}}AFX_DATA_MAP
```

}

当然，还可以对映射变量值的范围等进行设置（参见 2.5.6 小节）。设置完毕后，在 DoDataExchange 函数中也会加入相应的宏代码。

DoDataExchange 函数实现对话框控制与映射变量间的数据交换，当用 TRUE 参数调用 UpdateData 函数时，DoDataExchange 函数被调用，能够将控制中的值转变到映射变量中；当用 FALSE 为参数调用 UpdateData 函数时，DoDataExchange 函数被调用，能够将映射变量的值转变到对话框控制中。所以通过 UpdateData 函数的灵活应用，就可以达到在对话框控制与映射变量间的随意数据交换。又通过在 CDataISet 类中实现的映射变量与记录集合当前记录各个字段间的数据交换，最终达到利用对话框控制修改记录集合当前记录的目的。

15.4　增加和修改数据记录

15.3 节创建了一个对话框类 CDataIDlg，用来组织记录集合类 CDataISet 对象的交互编辑界面。本节介绍利用对话框类 CDataIDlg 来实现对记录集合的增加记录和修改记录功能。

15.4.1　获得记录的唯一识别号

本章介绍的用来管理数据表 city 的记录集合类 CDataISet，其管理的记录集合要用做 GIS 系统的性质数据库，所以记录集合中的每条记录必须具有唯一识别号，即在记录集合中设置一个存储唯一整数值的字段。采用不同的数据库系统时，有不同的获得记录唯一识别号的方法。对于使用 SQL Server 数据库作为数据源时，因为 VC++ 最终是靠调用 SQL Server 系统来实现各种数据库管理功能，当在数据表中设置一个 identity 属性的字段时，在数据表中增加一条记录时，自动为这个字段设置一个唯一值。而当用其他数据库文件作为数据源时，不能通过设置 identity属性字段来自动产生唯一识别号，需要在增加记录前，采用一定的方法来计算得到一个唯一识别号。下面介绍获得唯一识别号的方法。

(1) 通过记录集合最后记录的字段值获得唯一识别号

这种方法的实现思路是：通过得到记录集合中最后一条记录相应的字段值，将这个值加 1 作为要增加记录这个字段的数值。实际上，SQL Server 的 identity 属性字段就是靠这种方法来自动设定字段值的。这种方法不用与记录集合的其他记录进行比较，具有速度快的优点。但这种方法具有以下缺点：

① 必须保证能够获得最后一条记录，否则就没有实际意义。对于有些数据库系统，在完成插入记录的操作后，不能得到那条记录是最后加入的记录。

② 唯一识别号只能增加。当用这种方法增加记录时，唯一识别号的值一直在增加，当数据库经常进行删除操作的情况下，唯一识别号比较零散。

(2) 通过计算获得唯一识别号

另外一种方法是，对数据表已有的全部记录进行处理，比较选择出一个最小的整数作为唯一识别号。6.3.1 小节增加图形元素时，图形元素的唯一识别号就是通过这种方法获得的。

在视图类 CDatabaseView 中增加一个成员函数，用来获得 m_pSet1 指向的记录集合对象 m_data1中字段 only_id 的唯一识别号。函数的定义如下：

```
protetted:
    int GetOnlyID()
```

在实现文件 dbaseview.cpp 中，加入函数的实现代码：

```
int CDatabaseView::GetOnlyID()
{
    int *Index1,id;
    Index1 = GetDocument()->m_Index;          //指向文档对象中定义的数组 m_Index
    CString m_str1 = m_pSet1->m_strFilter;    //存储记录集合对象的过滤子式
    m_pSet1->m_strFilter = "";  //将过滤子式设为空，即产生的记录集合包含数据表的所有记录
    if(m_pSet1->IsOpen())                     //如果打开了集合对象
        m_pSet1->Close();                     //关闭记录集合对象
    m_pSet1->Open();                          //打开记录集合对象
    for(int i=0;i<30000;i++)                  //将整数数组 Index1 的所有项设为 0
        Index1[i]=0;
    while(! m_pSet1->IsEOF())                 //对记录集合的所有记录遍历
    {
        Index1[m_pSet1->m_only_id]=1;         //将以记录的字段值为下标的 Index1 数组项设为 1
        m_pSet1->MoveNext();                  //记录指针指向下一条记录
    }
    //以下得到数组 Index1 中第一个等于 0 的项，其下标号是与所有记录字段值不同的唯一值
    for(i=0;i<30000;i++)
    {
        if(Index1[i]==0)
        {
            id=i;
            break;
        }
    }
    m_pSet1->Close();                         //关闭记录集合对象
    m_pSet1->m_strFilter=m_str1;              //恢复记录集合对象原来的过滤子式
    m_pSet1->Open();                          //重新打开记录集合对象
    return id;                                //返回得到的唯一识别号
}
```

在 GetOnlyID 函数中，当打开记录集合对象时，将 CRecordSet 类的 m_strFilter 字符串设为空，使创建的记录集合包含数据表中的所有记录，来保证得到真正的唯一识别号。在改变 m_strFilter 字符串前，通过字符串 m_str1 将成员变量 m_strFilter 记录下来，操作完毕后，有恢复了记录集合对象的成员变量 m_strFilter，并重新打开记录集合，避免了 GetOnlyID 函数对记录集合对象的改变。

以上这种获得唯一识别号的方法，在记录集合记录数目较多的情况下，需要一定的运行时间。所以，如果利用 SQL Server 等数据库作为数据源进行数据库应用程序设计时，可考虑直接采用 identity 属性的字段，在增加记录时自动获得唯一的字段值。

15.4.2　增加记录

本小节在应用程序 Draw 中具体实现增加记录操作。

(1) 初始化对话框类 CData1Dlg 对象

因为增加和修改记录操作都是采用对话框类 CData1Dlg 来组织完成，而增加和修改记录操作过程略有不同，所以，为了识别正在进行的操作，在对话框类 CData1Dlg 中定义一个变量：

```
protected:
    BOOL b_IsAdd;
```

成员变量 b_IsAdd 为 TURE 时，表示正在进行增加记录操作；而 b_IsAdd 为 FALSE 时，表示正在进行修改记录操作。另外，在对话框类 CData1Dlg 中定义一个变量，来记录要增加记录的字段 only_id 的值。

```
protected:
    int m_id;
```

为了在创建 CData1Dlg 对象时能够对一些成员变量初始化，在对话类 CData1Dlg 中重载一个构造函数：

```
class CData1Dlg : public CDialog
{
// Construction
public:
    CData1Dlg(CWnd * pParent = NULL); // standard constructor
    CData1Dlg(CDataISet * pSet,BOOL IsAdd,int id,CWnd * pParent = NULL);

    ..... //省略以下代码
}
```

构造函数的第一个参数，用来传递指向要编辑修改的 CData1Set 对象的指针，第二个参数用来传递进行的操作类型(增加还是修改记录操作)，第三个参数用来传递 only_id 字段值(在增加记录操作时，表示要增加记录的唯一识别值，而在修改记录操作时，用来记录要修改记录的唯一识别值)。构造函数的实现代码如下：

```
CData1Dlg::CData1Dlg(DataISet * pSet,BOOL IsAdd,int id,CWnd * pParent /* = NULL */)
    : CDialog(CData1Dlg::IDD, pParent)
{
    m_pSet = pSet;          //将 m_pSet 指向要修改的记录集合对象
    b_IsAdd = IsAdd;        //记录操作类型
    m_id = id;              //记录 only_id 字段的唯一值
```

```
}
```

这样,在对话框类 CData1Dlg 的构造函数中,CData1Set 类指针 m_pSet 指向了实际的 CData1Set 对象,利用 m_pSet 指针就可以对要编辑的记录集合对象进行操作。

为了初始化对话框中各个控制的状态,用 ClassWizard 在对话框类 CData1Dlg 中重载 CDialog 类的虚函数 OnInitDialog。操作完毕后,在对话框类 CData1Dlg 中增加了虚函数的重载定义:

```
//{{AFX_MSG(CData1Dlg)
virtual BOOL OnInitDialog();
//}}AFX_MSG
```

在实现文件中增加了 OnInitDialog 函数的实现代码:

```
BOOL CData1Dlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    return TRUE; // return TRUE unless you set the focus to a control
}
```

为了分别完成在增加记录和修改记录操作时的初始化功能,将 OnInitDialog 函数修改如下:

```
BOOL CData1Dlg::OnInitDialog()
{
    char p1[7];
    if(b_IsAdd)                          //如果是增加记录操作
    {
        sprintf(p1,"%d",m_id);           //将 m_id 值转变成字符串 p1
        SetDlgItemText(IDC_EDIT1,p1);    //将 p1 写入编辑框 IDC_EDIT1(记录识别号的编辑框)
    }
    else                                 //如果是修改记录操作
        CDialog::OnInitDialog();
    return TRUE; // return TRUE unless you set the focus to a control
}
```

以上代码完成的功能是:在增加记录操作时,进入到对话框时,将编辑框 IDC_EDIT1 的内容设置为 m_id 记录的唯一识别值;而在修改记录操作时,则通过调用 CDialog 类的 OnInitDialog 函数,将记录集合当前记录的各个字段内容填入到相应的对话框控制中(因为在 CDialog::OnInitDialog 函数中,以 FALSE 为参数调用了 UpdateData 函数)。

(2) 在对话框类中实现增加记录操作

在 CData1Dlg 对话框中进行增加记录操作时,按中"保存退出"按钮时,会在记录集合中增加一条记录,记录的内容通过对话框的各个编辑控制得到。用 ClassWizard 为"保存退出"按钮(ID 号为 IDOK)增加消息处理函数,增加完毕后,对程序代码进行如下修改:

在对话框类 CData1Dlg 中增加了重载函数定义:

```
//{{AFX_MSG(CData1Dlg)
virtual BOOL OnInitDialog();
virtual void OnOK();
//}}AFX_MSG
```

在实现文件 data1dlg.cpp 中增加了函数实现代码：

```
void CData1Dlg::OnOK()
{
    // TODO: Add extra validation here
    EndDialog(IDOK);
}
```

为了完成增加记录操作功能，对函数 OnOK 的实现代码修改如下：

```
void CData1Dlg::OnOK()
{
    char p1[7];
    if(b_IsAdd)                    //如果是增加记录操作
    {
        m_pSet->AddNew();          //在记录集合中增加一条记录
        UpdateData();              //以 TRUE 为参数调用 UpdateData,保存对话框各控制的内
                                   //容转到当前记录
        if(m_pSet->CanUpdate())    //如果能够修改数据源
            m_pSet->Update();      //修改数据源,在实际数据表中增加记录
        else                       //如果不能修改数据源,给出提示信息
            ;                      //在此增加提示信息
    }
    EndDialog(IDOK);               //退出对话框
}
```

OnOK 函数的工作步骤如下：

① 调用 CRecordSet 类的 AddNew 函数在记录集合中增加一条新记录。

② 用参数 TRUE 调用 UpdateData 函数(TRUE 是默认参数，可以省略)，保存对话框控制中的内容到 m_pSet 指向的记录集合当前记录中。

③ 如果数据源能够被更新，则调用 CRecordSet 类的 Update 函数，将对记录集合的增加记录操作反映到数据源，对实际数据表增加记录；如果不能进行对数据源的更新操作，则给出提示信息。

(3) 实现增加记录操作功能

为了能够利用菜单“记录操作”的“增加记录”菜单项激发完成增加记录的具体操作，对视图类 CDatabaseView 中的消息处理函数 OnRecordAdd(参见 14.1 节)进行修改。修改后的函数实现代码如下：

```
void CDatabaseView::OnRecordAdd()
```

```
    {
        CData1Dlg dlg(m_pSet1,1,GetOnlyID()); //定义一个对话框类对象
        if(dlg.DoModal() == IDOK)             //如果按"保存退出"按钮退出对话框
        {
            Init2();                          //设置浏览视图的纵向参数
            SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine); //设置视图的滚动范围
            UpdateWindow();                   //更新滚动条
            Invalidate();                     //重画视图
        }
    }
```

在以上函数代码中,定义了一个对话框类 CData1Dlg 对象 dlg,在定义对象时,利用 GetOnlyID 函数得到要增加记录的唯一识别号,并作为参数传到了 dlg 对象中。

在运行对话框中的"保存退出"按钮退出对话框后,调用 Init2 函数重新计算视图滚动区域的纵向参数,重新设置滚动区域并重画视图。

因为在考虑了记录集合的过滤操作时(参见第 16 章),增加的记录可能并不在记录集合中,在考虑了记录集合的排序操作(参见第 17 章)时,需要确定增加记录要显示的实际位置。所以,在增加记录后,一般采用重新创建记录集合并重画视图的方法,而不采用对视图进行局部修改的方法。重新生成并重画视图可能需要较长的时间。

15.4.3　修改记录

对于上一小节中实现的增加记录功能进行略微修改,就可以利用对话框类 CData1Dlg 完成应用程序 Draw 中的修改记录操作。

(1) 修改对话框类 CData1Dlg 的 OnOk 函数

为了在 CData1Dlg 对话框中按中"保存退出"按钮时实现对记录集合当前记录进行修改,对消息处理函数 OnOK 进行修改,加入黑体部分的代码:

```
void CData1Dlg::OnOK()
{
    char p1[7];
    if(b_IsAdd)                          //如果是增加记录操作
    {
        m_pSet->AddNew();                //在记录集合对象中增加一条记录
        UpdateData(); //以 TRUE 为参数调用 UpdateData,保存对话框各控制的内容转到当前记录
        if(m_pSet->CanUpdate())          //如果能够修改数据源
            m_pSet->Update();            //修改数据源,在实际数据表中增加记录
        else                             //如果不能修改数据源,给出提示信息
            ;                            //在此增加提示信息
    }
    else                                 //如果正在进行修改记录操作
    {
```

```
        m_pSet->Edit();                    //将当前记录设置成修改状态
        UpdateData(); //以 TRUE 为参数调用 UpdateData,保存对话框各控制的内容转到当前记录
        if(m_pSet->CanUpdate())            //如果能够修改数据源
            m_pSet->Update();              //修改数据源,在实际数据表中增加记录
        else                               //如果不能修改数据源,给出提示信息
            ;                              //在此增加提示信息
    }
    EndDialog(IDOK);                       //退出对话框
}
```

当进行修改记录操作,按"保存退出"按钮退出对话框时,首先调用 CRecordSet 类的 Edit 函数使当前记录处于编辑状态,用参数 TRUE 调用 UpdateData 函数(TRUE 是默认参数,可以省略),保存对话框控制中的内容到 m_pSet 指向的记录集合当前记录中。最后调用 Update 函数将对记录集合的改变反映到数据源中,实现对数据源中数据表的修改。

(2) 修改视图类 CDatabaseView 中的消息处理函数

在应用程序 Draw 的数据库浏览视图上,用鼠标在浏览视图中选择要进行修改的记录,运行"记录操作"菜单下的"修改记录"菜单项,就会进入到 CData1Dlg 对话框中对这条记录进行修改。为了完成这个功能,对视图类 CDatabaseView 中的消息处理函数 OnRecordModify(参见 15.1.2小节)修改如下:

```
void CDatabaseView::OnRecordModify()
{
    if(nIndex<=0&&nIndex>nRecord)  //如果记录指针没有指向记录集合的一条记录,函数返回
        return;
    m_pSet1->Move(nIndex,SQL_FETCH_ABSOLUTE);      //将记录指针指向第 nIndex 条记录
    CData1Dlg dlg(m_pSet1,0,m_pSet1->m_only_id);   //定义一个对话框类对象
    if(dlg.DoModal()==IDOK)            //调用对话框,如果按中"保存退出"按钮退出对话框
    {
        Init2();                       //设置浏览视图的纵向参数
        SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);  //设置视图的滚动区域
        Invalidate();                  //重画视图
    }
}
```

OnRecordModify 函数的工作步骤如下,

① 根据变量 nIndex 所记录的记录指针位置,将记录集合的记录指针指向这条记录。

② 定义一个对话框类 CData1Dlg 对象,定义对象时第一个参数是指向记录集合对象的指针,第二个参数用来标识进行的是修改记录操作,第三个参数是这条记录的 only_id 字段值。

③ 调用对话框,如果是按"保存退出"按钮退出对话框,则调用 Init2 函数重新计算浏览视图的纵向参数,然后重画视图。

15.5　删除记录

本节来讨论实现删除记录集合中指定记录的操作。对记录的删除可分为对当前记录的删除和对记录集合所有记录删除两种。

15.5.1　删除当前记录

当运行应用程序 Draw 中菜单“记录操作”下的“删除记录”菜单项时，删除记录集合中记录指针指向的记录(当前记录)。为了完成这个操作功能，对“删除记录”菜单项的消息处理函数 OnRecordDelete(参见 15.1.2 小节)修改如下：

```
void CDatabaseView::OnRecordDelete()
{
    if(nIndex < 1||nIndex > nRecord) //如果记录指针没有指向记录集合的一条记录，函数返回
        return;
    if(! m_pSet1 -> IsOpen())    //如果没有打开记录集合对象
        m_pSet1 -> Open();       //打开记录集合对象
    CPoint p1;
    p1 = GetScrollPosition();        //记录下视图的滚动位置
    m_pSet -> Move(nIndex,SQL_FETCH_ABSOLUTE);   //将记录指针指向第 nIndex 条记录
    m_pSet1 -> Delete();             //删除当前记录
    if(nIndex = = nRecord)           //如果删除的是记录集合的最后一条记录
        nIndex - -;                  //将记录指针指向前一条记录
    Init2();                         //设置浏览视图的纵向参数
    SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);   //设置滚动区域
    Invalidate();                    //重画视图
    ScrollToPosition(p1);            //设置浏览视图的位置
}
```

OnRecordDelete 函数的工作步骤如下：

① 将记录指针指向记录集合第 nIndex 条记录。

② 通过 GetScrollPosition 函数得到视图的滚动位置 p1。

③ 调用 CRecordSet 类的 Delete 函数删除当前记录。如果删除的记录是最后一条记录，将 nIndex 变量减 1，即把上一条记录作为删除后记录指针指向的记录。

④ 调用 Init2 函数重新计算浏览视图的纵向参数，并激活视图重画。

⑤ 将视图滚动到记录下的位置 p1。保证删除记录后浏览视图的所在位置与删除记录前相同。

15.5.2　删除所有记录

在很多情况下，要求提供删除记录集合中所有记录的功能，特别是与记录集合的过滤操作相配合，来完成删除数据表中符合过滤条件的所有记录。

为了在应用程序 Draw 的数据库浏览视图界面中，在运行“记录操作”菜单的“删除所有记录”菜单项时，能够删除当前记录集合的所有记录，将这个菜单项的消息处理函数 OnRecordDeleteAll(见 15.1.2 小节)修改如下：

```
void CDatabaseView::OnRecordDeleteAll()
{
    if(! m_pSet1->IsOpen())          //如果没有打开记录集合对象
        m_pSet1->Open();             //打开记录集合对象
    for(int i=nRecord;i>0;i--)       //从后到前对记录集合的所有记录进行循环
    {
        m_pSet1->Move(i,SQL_FETCH_ABSOLUTE);
        m_pSet1->Delete();           //删除当前记录
    }
    nIndex=-1;                       //将记录指针指向记录集合范围外
    Init2();                         //重新设置浏览视图的纵向参数
    SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);   //设置视图滚动区域
    Invalidate();                    //重画视图
}
```

在以上代码中，对当前记录集合中所有的 nReord 条记录循环，由后向前依次删除各条记录，然后重新计算视图的纵向参数，并重画视图。

第 16 章　通用数据库过滤功能

数据库的过滤功能是数据库应用程序的一个基本操作功能。本章将在应用程序 Draw 的数据库浏览视图中实现通用的过滤功能，实现一个针对所有记录集合都适用的通用过滤条件编辑器。

16.1　过滤操作的实现方式

12.4.4 小节中，对利用 ODBC 进行数据库应用程序设计时，进行过滤操作的方式进行了介绍。按照组织过滤条件的方式不同，一般将过滤操作可以分为以下三种方式：固定条件、固定方式和通用条件。本节对各种操作方式的实现方法进行简单介绍。

16.1.1　固定条件过滤操作

固定条件过滤操作是一种最简单的过滤操作方法，在这种操作中，只能按照给出的固定条件来组织过滤操作。例如，对于在应用程序 Draw 中的管理 city 数据表的 CDataISet 记录集合对象，如果需要得到过滤条件是“面积大于 100 平方公里且人口大于 100 万人”的城市，可以通过一个固定条件过滤操作，用以下的代码来实现：

```
if(m_pSet->IsOpen())    //如果打开了记录集合对象
    m_pSet->Close();    //关闭记录集合对象
m_pSet->m_strFilter="(area>100) and (population>100)";    //设置过滤条件
m_pSet->Open();         //重新打开记录集合对象
```

以上代码就实现了一个固定过滤条件的过滤操作，m_pSet 是指向记录集合对象的 CDataISet 类指针。这种过滤操作方式也可以通过创建没有参数的存储过程来实现（存储过程的创建和使用方法可参见 13.3.4 小节）。

这种过滤操作具有很大的局限性，它只能针对确定的记录集合对象进行固定条件的过滤操作。

16.1.2　固定方式过滤操作

固定方式过滤操作指的是：过滤条件式包含的关键字和操作符是固定的，而过滤条件式中的操作值是改变的。如对于管理 city 数据表的 CDataISet 对象，可以提供这样一种过滤操作方式：“面积大于??? 平方公里且人口大于??? 万人的城市”，即过滤条件式是固定的，而过滤条件式中的各个操作值（面积的数目和人口的数目）是活动的，在进行过滤操作前得到这些操作值而形成过滤条件，然后去组织过滤操作。

这种过滤操作方式的应用比较普遍，可以用来对记录集合实现固定模式的过滤操作。以下介绍实现这种方式过滤操作的方法。

(1) 动态产生过滤条件式

可以通过动态产生字符串的方式,来得到过滤操作的过滤条件式,然后去组织过滤操作。以下代码可以动态产生一个过滤子式,并完成过滤操作:

```
char p1[8];
CString m_str;
m_str="(area>"
sprintf(p1,"%f",m_area);
m_str+=p1;
m_str+=") and (population>";
sprintf(p1,"%f",m_population);
m_str+=p1;
m_str+=")";
m_pSet->Close();
m_pSet->m_strFilter=m_str;
m_pSet->Open();
```

在以上操作中,m_area 和 m_population 是用来记录面积和人口数值的变量,根据变量值的不同,能够得到不同的过滤条件。例如,当 m_area=200.00、m_population =100.2 时,以上代码可得到如下的字符串"(area>200.00) and (population>100.2)",利用这个字符串作为过滤子式,就可以实现过滤操作。改变变量 m_area 和 m_population 的值,就可以得到不同的过滤子式,可以通过提供交互界面来得到这两个变量的值,从而实现固定方式的过滤操作。

(2) 创建带参数的存储过程

在 VC++ 6.0 企业版中,利用 SQL Server 数据库作为数据源进行数据库应用程序设计时,还可以通过创建一个带参数的存储过程来实现固定方式的过滤操作。把 m_area 和 m_population 作为存储过程的两个参数,通过交互方式得到面积和人口的数值,以两个数值作为参数来调用存储过程,同样可以达到固定方式过滤操作的目的。具体操作可参考 13.3.4 小节。

固定方式的过滤操作,也是只能针对确定的记录集合对象,而且能够实现的过滤操作也是有限的,需要创建大量的过滤操作模式来满足不同的过滤操作要求,不能进行提供的过滤模式以外的过滤操作。但它实现简单,用户操作简单方便,在数据库应用程序中广泛采用。

16.1.3　通用条件过滤操作

所谓通用条件过滤操作,就是通过提供一个交互式操作界面,来动态组织任意的过滤条件式,利用动态产生的过滤条件式来进行过滤操作。也就是说,过滤条件式是用户按照一定的方式自己组织的,因此可以灵活的实现各种条件的过滤操作。

要实现通用条件过滤操作,必须具备两个方面的功能:一个方面是必须提供一个易于操作的交互界面来组织过滤条件。操作的方便性极为重要,因为应用程序面对的是普通用户而非程序设计专家(对于程序设计专家来说,可以通过直接书写条件式来实现任意条件的过滤操

作);另外一个方面是必须提供对过滤条件式的分解能力,也就是说必须能够把通过交互操作界面产生的过滤条件式,又能够分解成易于操作和理解的交互编辑状态。

本章将具体讨论实现一个通用条件编辑器,只要传递给这个条件编辑器一个记录集合对象指针,就可以任意组织针对这个记录集合对象的过滤条件,从而实现任意条件的过滤操作,同时又能够将产生的过滤条件式分解为编辑状态去编辑修改。将这个过滤条件编辑器称为"通用",包含两个方面的含义:一是指过滤条件编辑器能够适用于所有的记录集合对象,即与具体的记录集合对象无关;二是指可以任意组织过滤条件式。

不管过滤条件编辑器的操作有多方便,其操作过程也比其他的过滤方式麻烦,所以在一般的数据库应用程序中,往往采用固定方式的过滤操作和通用条件过滤操作相结合的方法:通过固定方式的过滤操作来提供一些固定的和主要的过滤操作功能,而通过通用条件过滤操作来提供固定查询方式以外的过滤操作功能。

16.2 创建组织过滤条件的对话框类

为了实现通用过滤条件编辑器,需要创建一个对话框类,通过这个对话框类,完成过滤条件的组织和分解工作。

16.2.1 创建对话框资源

为了创建组织过滤条件的对话框类,首先需要创建一个对话框资源。在 VC++ 开发环境下,利用 ResoureView 工具调用 AppStudio,可视化的编辑创建一个对话框资源 IDD _ FILTER(如图 16-1 所示)。对话框资源的描述信息,可以在资源文件 draw.rc 中找到:

```
IDD_FILTER DIALOG DISCARDABLE 0, 0, 347, 221
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON   "F 查询",IDOK,270,138,50,14
    PUSHBUTTON      "C 放弃退出",IDCANCEL,271,158,50,14
    COMBOBOX        IDC_COMBO1,220,80,68,77,CBS_DROPDOWNLIST | WS_VSCROLL | WS_
                    TABSTOP
    LTEXT           "字段名称",IDC_STATIC,180,81,32,11
    LISTBOX         IDC_LIST1,9,13,324,59,LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_
                    TABSTOP
    COMBOBOX        IDC_COMBO2,220,97,35,46,CBS_DROPDOWNLIST | WS_VSCROLL | WS_
                    TABSTOP
    EDITTEXT        IDC_EDIT1,220,113,89,14,ES_AUTOHSCROLL
    LTEXT           "操作符",IDC_STATIC,181,99,32,11
    LTEXT           "操作值",IDC_STATIC,182,115,32,11
    LISTBOX         IDC_LIST2,10,79,166,53,LBS_NOINTEGRALHEIGHT|WS_VSCROLL|WS_
```

```
                    TABSTOP
    DEFPUSHBUTTON   "增加.OR.子式",ID_OR_ADD,14,140,53,14
    DEFPUSHBUTTON   "增加.AND.子式",ID_AND_ADD,77,140,56,14
    DEFPUSHBUTTON   "删除.OR.子式",ID_OR_DEL,13,158,53,14
    DEFPUSHBUTTON   "删除.AND.子式",ID_AND_DEL,78,159,56,14
    DEFPUSHBUTTON   "修改.OR.子式",ID_OR_MODIFY,143,140,53,14
    EDITTEXT        IDC_EDIT2,14,179,318,29,ES_AUTOHSCROLL
END
```

图 16-1　IDD_FILTER 对话框资源

在 ID 号定义文件 resource.h 中，给对话框资源中各个 ID 号定义了 ID 值。



图 16-2　列表框控制性质设置对话框

16.2.2 创建组织过滤条件的对话框类

现在利用 ClassWizard 来创建一个对话框类 CFilterDlg,对话框类使用对话框资源 IDD _ FILTER,类的头文件在 filterdlg.h,实现文件在 filterdlg.cpp(用 ClassWizard 新建对话框类的方法见 6.4.4 小节)。在对话框类 CFilterDlg 的头文件 filterdlg.h 中,可以找到对话框类 CFilterDlg 的定义代码:

```
class CFilterDlg : public CDialog
{
// Construction
public:
    CFilterDlg(CWnd * pParent = NULL);
protected:
// Dialog Data
    //{{AFX_DATA(CFilterDlg)
    enum { IDD = IDD_FILTER };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CFilterDlg)
    protected:
    virtual void DoDataExchange(CDataExchange * pDX); // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CFilterDlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

在实现文件 filterdlg.cpp 中,定义了对话框类 CFilterDlg 的实现代码。以下是 filterdlg.cpp 文件的代码列表:

```
// FilterDlg.cpp : implementation file
#include "stdafx.h"
#include "draw.h"
#include "FilterDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
```

```
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////
// CFilterDlg dialog

CFilterDlg::CFilterDlg(CWnd* pParent /* = NULL*/)
    : CDialog(CFilterDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CFilterDlg)
    //}}AFX_DATA_INIT
}

void CFilterDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CFilterDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CFilterDlg, CDialog)
    //{{AFX_MSG_MAP(CFilterDlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////
// CFilterDlg message handlers
```

以下各节，将在本节产生的对话框类 CFilterDlg 的基础上，开发实现组织通用过滤条件的编辑器。

16.3　组织过滤条件编辑器的各种功能

本节在上述产生的对话框类 CFilterDlg 的基础上，组织实现过滤条件编辑器的各项功能。

16.3.1　过滤条件编辑器编辑的过滤条件式的语法格式

对于不同的过滤条件编辑器，能够组织不同复杂程度的过滤条件式。在组织过滤条件编辑器的功能设计时，一般采用如下的原则：过滤条件编辑器能够组织复杂程度适中的过滤条件式，因为能够组织的过滤条件式的复杂程度越高，组织过滤条件的操作界面和操作过程就会越复杂，用户就越不容易掌握，也就失去了过滤条件编辑器的存在意义；另一方面，过滤条件编辑

器的功能又不能太简单，必须能够组织常用的过滤条件式。

本章介绍的过滤条件编辑器能够处理以下规则的过滤条件式。

(1) 运算符

在过滤条件式中，能够处理 < >、>、<、=、> =、< = 等六种运算符，而不能处理 SQL 语句中的各种运算函数(如 between 等)。即基本过滤条件子式格式为：

字段名称　　运算符　　操作值

如对于 CDataISet 对象，可以组织以下的基本子式：

```
population > 100
area < 50
area < = 30
city _ id = 1
city _ name = "北京"
......
```

(2) 连接符

在过滤条件中能够处理两个连接符 AND(且)和 OR(或)，通过这两个连接符可以将基本条件子式连接成较为复杂的过滤条件子式。如对于 CDataISet 对象，可以组织以下的过滤条件子式：

```
(area < 50) AND (area < = 30)
(population > 100) OR (area < 50)
......
```

(3) 过滤条件式的组织层次

在 SQL 语言的过滤条件式中，通过括号可以将过滤条件组成不同的层次。为了利用交互界面编辑过滤条件的需要，过滤条件编辑器一般只能按照一定的层次来组织过滤条件式，本章讨论的过滤条件编辑器只能组织包含两个层次的过滤条件式：

① 第一个层次是可以把各个基本条件子式通过 OR 连接符连接成过滤条件子式。以下的过滤条件子式都是由基本条件子式通过连接符连接而成：

```
(area < 50) OR (area < = 30)
(population > 100) OR (area < 50)
(population > 100) OR (area < 50) OR (area > 100)
(population < 30) OR(population > 100) OR (area < 50) OR (area > 100)
```

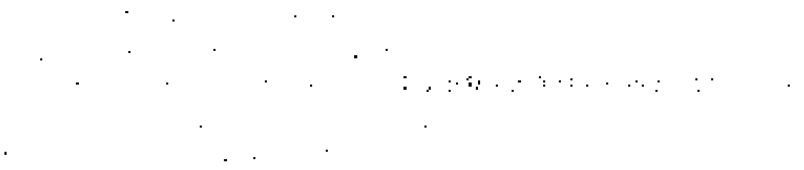
② 第二个层次是可以把①中产生的第一个层次的过滤条件子式，通过 AND 连接符连接成更复杂的过滤条件式：

```
((area < 50) OR (area < = 30)) AND ((population > 100) OR (area < 50))
```

((population > 100) OR (area < 50)) AND ((population > 100) OR (area < 50) OR (area > 100))

.....

即用本章实现的过滤条件编辑器产生的过滤条件式，可以分解成多个以 AND 连接符连接的条件子式(在本章的讨论中，把这种条件子式称为 AND 子式)，而把 AND 子式又可以被分解成一个或多个以 OR 关系符连接的基本条件子式(在本章的讨论中，把这种基本条件子式称为 OR 子式)。



16.3.2　定义和初始化对话框类的变量

现在开始在 16.2 节产生的对话框类 CFilterDlg 的基础上，进行过滤条件编辑器各项功能的开发工作。本小节首先讨论需要在对话框类 CFilterDlg 中定义各个成员变量。

(1) 在对话框类 CFilterDlg 中定义成员变量

首先在对话框类 CFilterDlg 中，定义完成过滤条件编辑器所需要的各个变量：

```
class CFilterDlg : public CDialog
{

.....//省略以上代码
public:
    Data1Set * m_pSet;                    //指向记录集合对象的指针
    CString m_strFilter;                  //存储过滤条件式
private:
    CListBox * plist1, * plist2;
    CComboBox * pcom1, * pcom2;
protected:
    CString m_strAnd[14];                 //存储 AND 过滤条件子式的字符串数组
    CString m_strOr[6];                   //存储 OR 过滤条件子式的字符串数组
    short m_NumbAnd, m_NumbOr;
    short m_NumbOperator;
    short m_NumbField;
    char * m_strOperator[10];
    short m_OperatorLong[10];
    CODBCFieldInfo pFieldInfo[200];
```

```
.....//省略以下代码
}
```

变量 m_pSet 是一个 CDaa1Set 类指针,用来指向要组织过滤条件的 CDaa1Set 记录集合对象。为了能够在对话框类 CFilterDlg 中使用 CDaa1Set 类,在对话框类 CFilterDlg 的定义前加入引用 CDaa1Set 类的代码:

```
class CDaa1Set;
```

m_strFilter 是一个 CString 对象,用来存储在过滤条件编辑器产生的过滤条件式;m_strAnd 是一个 CString 型对象数组,用来存储一个过滤条件中的 AND 子式;m_strOr 也是一个 CString 型对象数组,用来存储一个 AND 子式包含的 OR 子式(基本条件子式)。例如,对于如下的过滤条件式:

```
((population > 100) OR (area < 50)) AND ((population > 100) OR (area < 50) OR (area > 100))
```

当在过滤条件编辑器中对这个条件进行编辑时,m_strAnd 数组中的各个项内容如下:

```
m_strAnd[0] = "(population > 100) OR (area < 50)"
m_strAnd[1] = "(population > 100) OR (area < 50) OR (area > 100)"
```

而对于 m_strAnd[0]中的 AND 子式,经过分解后在 m_strOr 数组中存储内容如下:

```
m_strOr[0] = "population > 100"
m_strOr[1] = "area < 50"
```

m_NumbAnd 和 m_NumbOr 分别用来记录 m_strAnd 和 m_strOr 数组中存储的条件子式的项数。对于以上例子,m_NumbAnd 和 m_NumbOr 的值都等于 2。

变量 m_NumbOperator 用来存储在过滤条件中处理的运算符的数目,而 CString 数组 m_strOperator 用来存储各个运算符,short 数组 m_OperatorLong 用来存储 m_strOperator 数组中对应的运算符的字符长度。

short 型变量 m_NumbField 用来存储 m_pSet 指向的记录集合对象中包含的字段数目,而 ODBCFieldInfo 型数组 pFieldInfo 用来存储记录集合对象中各个字段的信息。

(2) 初始化对话框类中的变量

为了能够将指向需要组织过滤条件的记录集合对象的指针传入,以及对各成员变量进行初始化,需要对 CFilterDlg 的构造函数进行修改(也可重载另外一个构造函数)。修改后的构造函数定义如下:

```
CFilterDlg(CDaa1Set * pSet,CWnd * pParent /* = NULL*/);
```

在构造函数中加入代码,对对话框类中定义的一些成员变量进行初始化。修改后的构造函数实现代码如下:

```
CFilterDlg::CFilterDlg(CDaa1Set * pSet,CWnd * pParent /* = NULL*/)
    : CDialog(CFilterDlg::IDD, pParent)
{
    m_pSet = pSet;                    //将 m_pSet 指向需要组织过滤条件的记录集合对象
```

```
    for(int i = 0;i < 10;i++ )      //为字符数组分配空间
        m_strOperator[i] = (char *)malloc(4);
    m_NumbOperator = 6;             //有六个运算符
    //以下是 6 个运算符的名称
    strcpy(m_strOperator[0],"< >");
    strcpy(m_strOperator[1],"< =");
    strcpy(m_strOperator[2],"> =");
    strcpy(m_strOperator[3],"=");
    strcpy(m_strOperator[4],"<");
    strcpy(m_strOperator[5],">");
    //以下是 6 个运算符的字符长度
    m_OperatorLong[0] = 2;
    m_OperatorLong[1] = 2;
    m_OperatorLong[2] = 2;
    m_OperatorLong[3] = 1;
    m_OperatorLong[4] = 1;
    m_OperatorLong[5] = 1;
    if(! m_pSet->IsOpen())                    //如果没有打开记录集合对象
        m_pSet->Open();                       //打开记录集合
    m_strFilter = m_pSet->m_strFilter;        //保存记录集合对象开始的过滤条件
    m_NumbField = m_pSet->GetODBCFieldCount(); //得到记录集合所包含的字段数目
    CODBCFieldInfo fieldinfo1;
    CODBCFieldInfo& fieldinfo = fieldinfo1;
    for(i = 0;i < m_NumbField;i++ )           //得到各个字段的参数信息
    {
        m_pSet->GetODBCFieldInfo(i,fieldinfo);  //得到第 i 个字段的信息
        pFieldInfo[i] = fieldinfo1;             //将各个字段信息记录在数组 pFieldInfo
    }
    m_NumbAnd = 0;                            //AND 子式的数目为 0
    m_NumbOr = 0;                             //OR 子式的数目为 0
}
```

在以上构造函数中，对多个变量进行了初始化，并打开了记录集合对象，保存了记录集合对象中原来的过滤条件式，并得到并保存了记录集合对象中的字段数目及各个字段的参数信息。

为了能够在实现文件 filterdlg.cpp 中使用 CData1Set 类，在实现文件 filterdlg.cpp 的开始处插入 CData1Set 类的头文件：

```
#include "data1set.h"
```

16.3.3　产生和分解各级条件子式

实现过滤条件编辑器的关键是实现过滤条件式的生成和分解，既能够将各个过滤条件子

式通过连接符按层次组成复杂的过滤条件式，又能够将过滤条件式分解成可以用对话框界面交互表示的各级过滤条件子式。为了实现这些操作功能，在对话框类 CFilterDlg 中定义以下几个函数：

```
public:
    int BreakAndStr(CString m_str);
    int BreakOrStr(CString m_str);
    BOOL MakeAndStr(CString& m_str);
    BOOL MakeOrStr(CString& m_str);
    BOOL MakeStr();
```

以下分别介绍各个成员函数的功能和代码实现。

(1) BreakAndStr 函数

BreakAndStr 函数可以将一个过滤条件式分解成 AND 子式。函数的参数 m_str 是要分解的过滤条件式，函数返回分解成的 AND 子式的数目，分解后得到的各个 AND 子式存储在 CString 对象数组 m_strAnd 中。例如，对于以下过滤条件式进行分解：

"((population > 100) OR (area < 50)) AND ((population > 100) OR (area < 50) OR (area > 100))"

BreakAndStr 函数返回 2，m_strAnd 对象数组中前两项的内容为：

```
m_strAnd[0] = "(population > 100) OR (area < 50)"
m_strAnd[1] = "(population > 100) OR (area < 50) OR (area > 100)"
```

在实现文件 filterdlg.cpp 中，加入 BreakAndStr 函数的实现代码：

```
int CFilterDlg::BreakAndStr(CString m_str)
{
    int nChar,nChar1;
    m_str.TrimLeft();                    //去掉字符串左边的空格
    m_str.TrimRight();                   //去掉字符串后边的空格
    nChar = m_str.GetLength();           //得到字符串的长度
    CString m_str1 = m_str;              //复制一个字符串
    CString m_str2;
    int nStart;
    m_str1.MakeUpper();                  //将字符串 m_str1 转成大写
    nStart = 0;
    m_NumbAnd = 0;                       //AND 子式的数目为 0
    int n;
    do
    {
        n = m_str1.Find(" AND ",nStart);//从字符串 m_str1 中第 nStart 字符后寻找字符串" AND "
        if(n >= 0)                       //n 为找到的字符串第一个字符的位置，n >= 0 表明在字
                                         //符串 m_str1 中找到" AND "
        {
```

```
            if(n > nStart)                    //如果找到的字符串位置在开始寻找的起点位置之后
            {
        //将字符串 m_str1 从 nStart 处到" AND "字符串前的字符串拷贝到字符串 m_str2
        m_str2 = m_str.Mid(nStart, n - nStart);
        m_str2.TrimLeft();                    //去掉字符串前面的空格
        m_str2.TrimRight();                   //去掉字符串后面的空格
        nChar1 = m_str2.GetLength();          //得到字符串的长度
        //如果字符串 m_str2 的第一和最后一个字符分别为"("和")"
        if(m_str2.GetAt(0) == 40&&m_str2.GetAt(nChar1 - 1) == 41)
            m_strAnd[m_NumbAnd] = m_str2.Mid(1, nChar1 - 2);      //去掉字符串前后的括号
        else                                  //如果字符串 m_str 的前后不是一对括号
            m_strAnd[m_NumbAnd] = m_str2;     //将字符串 m_str2 作为一个 AND 子式保存
        m_NumbAnd++;                          //AND 子式的数目加 1
        m_str2.Empty();
    }
        nStart = n + 5;          //将" AND "后的位置作为起始位置,从这位置继续寻找" AND "字符
    }
    else          //如果没有在 m_str1 字符串中,从 nStart 个字符开始的部分中找到字符串" AND "
    {
        if(nStart < nChar)                    //如果开始寻找的位置小于字符串的长度
        {
            m_str2 = m_str.Mid(nStart, nChar);    //将 m_str 中 nStart 后的字符串拷贝到 m_str2
            m_str2.TrimLeft();                //去掉字符串前面的空格
            m_str2.TrimRight();               //去掉字符串后面的空格
            nChar1 = m_str2.GetLength();      //得到字符串的长度
            //如果字符串 m_str2 的第一和最后一个字符分别为"("和")"
            if(m_str2.GetAt(0) == 40&&m_str2.GetAt(nChar1 - 1) == 41)
                m_strAnd[m_NumbAnd] = m_str2.Mid(1, nChar1 - 2);
            else                              //如果字符串 m_str 的前后不是一对括号
                m_strAnd[m_NumbAnd] = m_str2; //将字符串 m_str2 作为一个 AND 子式保存
            m_NumbAnd++;                      //AND 子式的数目加 1
            m_str2.Empty();
        }
    }
    while(n >= 0);
    return m_NumbAnd;                         //返回能够分解成的 AND 子式的数目
}
```

BreakAndStr 函数的主要操作步骤如下：

① 将参数传入的字符串 m_str 去掉前后的空格，得到字符串的长度 nChar，并将字符串 m_str复制到字符串 m_str1 中，将字符串 m_str1 全部变成大写。

② 从字符串的 nStart 个字符开始查找" AND "字符串。如果找到" AND " 字符串，则将字符串 m_str 中从 nStart 个字符开始到找到的" AND "字符串前的部分，拷贝到字符串 m_str2，

去掉 m_str2 字符串前后的空格和前后的一层括号，然后将其拷贝到 CString 对象数组 m_strAnd 中对应的项目下，记录 AND 子式数目的变量 m_NumbAnd 加 1。

③ 在 m_str1 字符串中，从上一次找到的字符串“ AND ”后的位置开始查找字符串“AND”。如果找到了“ AND ”字符串，则进行与步骤②相同的操作，分解得到一个 AND 子式。重复这个步骤，直到不能找到字符串“ AND ”，这时，将字符串 m_str 中剩余的一部分字符串处理（去掉空格、括号等）后也作为一个 AND 子式存储到 m_strAnd 数组中。

④ 函数返回将字符串 m_str 分解成的 AND 子式的数目。

(2) BreakOrStr 函数

BreakOrStr 函数将一个 AND 子式分解成多个基本条件子式（OR 子式）。函数的参数 m_str 是要分解的 AND 子式，函数返回分解成的基本条件子式的数目，分解得到的各基本条件子式存储到 CString 对象数组 m_strOr 中。例如，对于以下的 AND 子式：

```
"(population > 100) OR (area < 50)"
```

用 BreakOrStr 函数进行分解，函数返回 2，m_strOr 数组中前两项的内容如下：

```
m_strOr[0] = "population > 100"
m_strOr[1] = "area < 50"
```

BreakOrStr 函数的实现过程与 BreakAndStr 函数类似。在实现文件 filterdlg.cpp 中，加入 BreakOrStr 函数的实现代码：

```
int CFilterDlg::BreakOrStr(CString m_str)
{
    int nChar,nChar1;
    m_str.TrimLeft();
    m_str.TrimRight();
    nChar = m_str.GetLength();
    CString m_str1 = m_str;
    CString m_str2;
    int nStart;
    m_str1.MakeUpper();
    nStart = 0;
    m_NumbOr = 0;
    int n;
    do
    {
        n = m_str1.Find(" OR ",nStart);
        if(n > = 0)
        {
            if(n > nStart)
            {
                m_str2 = m_str.Mid(nStart,n - nStart);
```

```
                m_str2.TrimLeft();
                m_str2.TrimRight();
                nChar1 = m_str2.GetLength();
                if(m_str2.GetAt(0) = = 40&&m_str2.GetAt(nChar1 - 1) = = 41)
                    m_strOr[m_NumbOr] = m_str2.Mid(1,nChar1 - 2);
                else
                    m_strOr[m_NumbOr] = m_str2;
                m_NumbOr++;
                m_str2.Empty();
            }
            nStart = n + 4;
        }
        else
        {
            if(nStart < nChar)
            {
                m_str2 = m_str.Mid(nStart,nChar);
                m_str2.TrimLeft();
                m_str2.TrimRight();
                nChar1 = m_str2.GetLength();
                if(m_str2.GetAt(0) = = 40&&m_str2.GetAt(nChar1 - 1) = = 41)
                    m_strOr[m_NumbOr] = m_str2.Mid(1,nChar1 - 2);
                else
                    m_strOr[m_NumbOr] = m_str2;
                m_NumbOr++;
                m_str2.Empty();
            }
        }
    } while(n > = 0);
    return m_NumbOr;
}
```

BreakOrStr 函数的实现步骤与 BreakAndStr 函数类似，读者可参考 BreakAndStr 函数的代码注释和实现步骤。

(3) MakeAndStr 函数

MakeAndStr 函数实现将多个基本条件子式(OR 子式)，以 OR 连接符连接成一个 AND 子式，函数操作成功时返回 TRUE，否则返回 FALSE。函数产生的 AND 子式通过 CString& 型参数返回。以下是 MakeAndStr 函数的实现代码：

```
BOOL CFilterDlg::MakeAndStr(CString& m_str)
{
    if(m_NumbOr < = 0)      //如果 OR 子式(基本条件子式)的数目小于 0,则函数返回 FALSE
```

```
        return FALSE;
    m_str = m_strOr[0];                    //字符串等于第一个 OR 子式
    for(int i = 1;i < m_NumbOr;i++)        //对其余的 OR 子式循环
    {
        m_str += " or ";                   //在字符串中增加连接符 OR
        m_str += m_strOr[i];               //连接 OR 子式
    }
    return TRUE;
}
```

(4) MakeStr 函数

MakeStr 函数实现将多个 AND 子式,通过连接符 AND 连接成一个过滤条件式。函数产生的条件式存储在对话框类 CFilterDlg 的成员变量 m_strFilter 中。以下是函数的实现代码:

```
BOOL CFilterDlg::MakeStr()
{
    m_strFilter.Empty();                   //清空 m_strFilter 字符串
    for(int i = 0;i < m_NumbAnd;i++)       //对所有的 AND 子式循环
    {
        m_strAnd[i].TrimLeft();            //去掉 AND 子式前面的空格
        m_strAnd[i].TrimRight();           //去掉 AND 子式后面的空格
        if(i > 0)                          //如果不是第一个 AND 子式
            m_strFilter += " AND ";        //在字符串中增加 AND 连接符
        m_strFilter += "(";                //在字符串中增加左括号
        m_strFilter += m_strAnd[i];        //在字符串中增加第 i 个 AND 子式
        m_strFilter += ")";                //在字符串中增加右括号
    }
    SetDlgItemText(IDC_EDIT2,m_strFilter);  //将得到的过滤条件式显在编辑框 IDC_EDIT2 中
    return TRUE;
}
```

当 AND 子式的数目 m_NumbAnd 等于 2,m_strAnd 前两项存储的 AND 子式如下所示时:

```
m_strAnd[0] = "(population > 100) OR (area < 50)"
m_strAnd[1] = "(population > 100) OR (area < 50) OR (area > 100)"
```

MakeStr 函数能够产生一个如下的字符串,并将这个字符串存储在 CFilterDlg 对象的成员变量 m_strFilter 中:

```
"((population > 100) OR (area < 50)) AND ((population > 100) OR (area < 50) OR (area > 100))"
```

16.3.4 初始化对话框的状态

为了在启动 CFilterDlg 对话框时完成必要的初始化操作,用 ClassWizard 在对话框类 CFilter-

Dlg 中重载虚函数 OnInitDialog。操作完成后，在对话框类 CFilterDlg 中增加了虚函数的定义：

```
//{{AFX_MSG(CFilterDlg)
virtual BOOL OnInitDialog();
//}}AFX_MSG
```

在实现文件 filterdlg.cpp 中，增加了 OnInitDialog 函数的实现代码。为了具体完成初始化对话框的操作，对 OnInitDialog 函数进行修改，修改后的实现代码如下：

```
BOOL CFilterDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    plist1 = (CListBox *)GetDlgItem(IDC_LIST1);          //得到指向列表框 IDC_LIST1 的指针
    plist2 = (CListBox *)GetDlgItem(IDC_LIST2);          //得到指向列表框 IDC_LIST2 的指针
    pcom1 = (CComboBox *)GetDlgItem(IDC_COMBO1); //指向列表选择框 IDC_COMBO1 的指针
    pcom2 = (CComboBox *)GetDlgItem(IDC_COMBO2); //指向列表选择框 IDC_COMBO2 的指针
    plist1->ResetContent();                              //清空列表框 IDC_LIST1 中的项目
    plist2->ResetContent();                              //清空列表框 IDC_LIST2 中的项目
    pcom1->ResetContent();                               //清空列表选择框 IDC_COMBO1 中的项目
    pcom2->ResetContent();                               //清空列表选择框 IDC_COMBO1 中的项目
    //以下在列表选择框 IDC_COMBO2 中增加各个运算符
    for(int i=0;i<m_NumbOperator;i++)
        pcom2->AddString(m_strOperator[i]);
    //以下在列表选择框 IDC_COMBO1 中，增加记录集合对象的各个字段名称
    for(i=0;i<m_NumbField;i++)
        pcom1->AddString(pFieldInfo[i].m_strName);
    BreakAndStr(m_strFilter);    //将过滤条件式分解成 AND 子式
    //以下将各 AND 子式增加到列表框 IDC_LIST1 中
    for(i=0;i<m_NumbAnd;i++)
        plist1->AddString(m_strAnd[i]);
    if(m_NumbAnd>0)              //如果 AND 子式数目大于 0
    {
        plist1->SetCurSel(0); //选择列表框 IDC_LIST1 的第一个项目
        OnSelchangeList1();      //调用 OnSelchangeList1 函数来分解 AND 子式
    }
    MakeStr();                   //重新计算得到过滤条件式 m_strFilter
    SetDlgItemText(IDC_EDIT2,m_strFilter);
    return TRUE; // return TRUE unless you set the focus to a control
}
```

OnInitDialog 函数的主要工作步骤如下：

① 得到指向列表框 IDC_LIST1 和 IDC_LIST2 的指针 plist1 和 plist2，得到指向列表选择框 IDC_COMBO1 和 IDC_COMBO2 的指针 pcom1 和 pcom2，并清空这些控制，使这些控制内不含任何项目。

② 在列表选择框 IDC_COMBO2 中增加各个运算符，在列表选择框 IDC_COMBO1 中增加

记录集合对象的各个字段名称

③ 调用 BreadAndStr 函数将过滤条件式 m _ strFilter 分解，将分解后得到的各个 AND 子式加入到列表框 IDC _ LIST1 中。

④ 如果 AND 子式的数目大于 0(m _ NumbAnd > 0)，即在列表框 IDC _ LIST1 中有项目，选择列表框 IDC _ LIST1 中的第一个项目，并调用 OnSelchangeList1 函数将 m _ strAnd[0]中存储的 AND 子式分解，将分解后的 OR 子式(基本条件子式)增加到列表框 IDC _ LIST2 中。OnSelchangeList1函数的创建和实现代码参见 16.4.6 小节。

⑤ 调用 MakeStr 函数将各个 AND 子式组织成过滤条件式 m _ strFilter，并将其字符串 m _ strFilter的内容写入到编辑框 IDC _ EDIT2 中。

16.3.5 增加条件子式

本小节在对话框类 CFilterDlg 中实现增加条件子式的功能，实现用交互操作界面组织过滤条件式。

(1) 产生基本条件子式

在 CFilterDlg 对话框中，通过如下的交互操作来组织一个基本条件子式：从列表选择框 IDC _ COMBO1 中选择记录集合中的一个字段，从列表选择框 IDC _ COMBO2 中选择运算符，从编辑框 IDC _ EDIT1 中取得操作值，利用字段、运算符和操作值可组成一个基本条件子式。为了实现这个功能，在对话框类 CFilterDlg 中定义一个成员函数：

```
public:
    BOOL MakeOrStr(CString &m _ str);
```

在实现文件 filterdlg.cpp 中，加入 MakeOrStr 函数的实现代码：

```
BOOL CFilterDlg::MakeOrStr(CString &m _ str)
{
    CString m _ str1;
    BOOL yn;
    int ncom1 = pcom1 - > GetCurSel();          //得到列表选择框 IDC _ COMBO1 中的选项 0 基序号
    if(ncom1 < 0)                               //如果选项序号小于 0,即没有在 IDC _ COMBO1 中选择
                                                //一个选项
        return FALSE;                           //函数返回 FALSE
    m _ str.Empty();                            //清空 m _ str 字符串
    m _ str + = pFieldInfo[ncom1].m _ strName;  //将字段名称加入到字符串 m _ str
    int ncom2 = pcom2 - > GetCurSel();          //得到列表选择框 IDC _ COMBO2 中的选项 0 基序号
    if(ncom2 < 0)                 //如果选项序号小于 0,即没有在 IDC _ COMBO2 中选择一个选项
        return FALSE;                           //函数返回 FALSE
    m _ str + = m _ strOperator[ncom2];         //将运算符加入到字符串 m _ str
    GetDlgItemText(IDC _ EDIT1,m _ str1);       //读出编辑框 IDC _ EDIT1 中的内容到字符串 m _ str1
    m _ str1.TrimLeft();                        //去掉字符串 m _ str1 左边的空格
    m _ str1.TrimRight();                       //去掉字符串右边的空格
```

```
    int nn = m_str1.GetLength();              //得到字符串的长度
    if(nn <= 0)          //如果字符串的长度小于等于 0,即没有在 IDC_EDIT1 中输入运算值
    {
        AfxMessageBox("没有输入字符值");
        return FALSE;
    }
    if(pFieldInfo[ncom1].m_nSQLType == 1)   //如果当前选择的字段类型为字符型
        yn = 1;
    else                    //如果当前选择的字段类型不是字符型
        yn = 0;
    if(nn > 2&&yn == 1) //如果当前选择的字段类型是字符型,且操作值字符串长度大于 2
    {
        if(m_str1.GetAt(0) == 39&&m_str1.GetAt(nn-1) == 39) //如操作值字符串前后为字符"'"
            yn = 0;
    }
    if(yn)//如需要对操作值字符串加引号(字段类型是字符型,而操作值字符串又没有加引号时)
        m_str += "'";   //在字符串中增加一个引号
    m_str += m_str1;   //增加操作值字符串到字符串 m_str
    if(yn)
        m_str += "'";   //在字符串中增加一个引号
    return TRUE;
}
```

MakeOrStr 函数的主要操作步骤如下：

① 得到列表选择框 IDC_COMBO1 中选项的序号。如果在列表选择框 IDC_COMBO1 中没有选项，即没有选择字段名称时，函数返回 FALSE 退出。如果在列表选择框 IDC_COMBO1 中选择了字段名称，则通过选项的序号来得到字段的名称，将字段名称增加到字符串 m_str。

② 得到列表选择框 IDC_COMBO2 中选项的序号。如果在列表选择框中没有选项，即没有从列表选择框 IDC_COMBO2 中选择运算符，函数返回 FLASE 退出。如果在 IDC_COMBO2 中选择了运算符，则通过选项序号来得到运算符的字符串，并将其增加到字符串 m_str。

③ 读出编辑框 IDC_EDIT1 中的内容到字符串 m_str1，去掉字符串前后的空格。如果字符串的长度小于 1，即在编辑框 IDC_EDIT1 中没有输入操作值，函数返回 FALSE 退出。

④ 当选择的字段类型是字符型时(pFieldInfo[ncom1].m_nSQLType == 1)，在字符串长度大于 2 的情况下，检查字符串的前后是否为单引号(')。如果字段类型为字符型且字符串的前后没有单引号，在产生基本条件子式时，应该在操作值字符串的前后加单引号。变量 yn 用来控制是否需要在操作值字符串前后加单引号，当 yn = 1 时需要加单引号。

⑤ 将操作值字符串加到字符串 m_str 之后，对于需要加单引号的情况(yn = 1)，在操作值字符串的前后加上单引号。

MakeOrStr 函数通过 CString& 型参数 m_str 返回产生的基本条件子式(OR 子式)。例如：当在选择列表框 IDC_COMBO1 中选择字段名称"only_id"，在选择列表框 IDC_COMBO2 中选择运算符">="，在编辑框 IDC_EDIT1 中输入操作值"12"时，MakeOrStr 函数会产生一个基本条件子式："only_id >= 12"。

又例如，当在列表选择框 IDC _ COMBO1 中选择字段“city _ name”，从列表选择框 IDC _ COMBO2 中选择运算符“ = ”，在编辑框 IDC _ EDIT1 中输入操作值“北京”时，MakeOrStr 函数会产生一个基本条件子式：“city _ name = ‘北京’”。

(2) 增加 OR 子式

当按中 CFilterDlg 对话框(见 IDD _ FILTER 对话框资源)中的“增加 OR 子式”按钮(ID 号 ID _ OR _ ADD)时，如果在列表框 IDC _ LIST1 选择了一个 AND 子式，则会根据对话框中所做的选择产生一个基本条件子式，并把这个基本条件子式通过连接符 OR 与在列表框 IDC _ LIST1 中选择的 AND 子式，组成一个包含更多基本条件子式的 AND 子式。为了实现这个功能，用 ClassWizard 为按钮“增加 OR 子式”(ID 号为 ID _ OR _ ADD)增加一个 BN _ CLICKED 控制消息处理函数，增加完毕后，在对话框类 CFilterDlg 中定义了一个消息处理函数：

```
//{{AFX_MSG(CFilterDlg)
virtual BOOL OnInitDialog();
afx_msg void OnOrAdd();
//}}AFX_MSG
```

在实现文件 filterdlg.cpp 中，增加了映射机制：

```
BEGIN_MESSAGE_MAP(CFilterDlg, CDialog)
    //{{AFX_MSG_MAP(CFilterDlg)
    ON_BN_CLICKED(ID_OR_ADD, OnOrAdd)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 filterdlg.cpp 中，增加了 OnOrAdd 函数空的实现代码，将函数实现代码进行修改，得到修改后的实现代码如下：

```
void CFilterDlg::OnOrAdd()
{
    int nlist1 = plist1 -> GetCurSel(); //得到列表框 IDC _ LIST1 中对 AND 子式的选项
    if(nlist1 < 0)                      //如果没有在 IDC _ LIST1 中选择 AND 子式，函数返回
        return;
    if(m_NumbOr > = 6)  //如当前 AND 子式已包含六个以上 OR 子式(基本条件子式)，函数退出
    {
        AfxMessageBox("最多只能有六个.OR.子式");
        return;
    }
    CString m_str1;
    CString& p_str1 = m_str1;
    BOOL yn = MakeOrStr(p_str1);              //调用函数 MakeOrStr 函数，得到基本条件子式
    if(yn)                                    //如果成功地得到了基本条件子式
    {
        m_strOr[m_NumbOr++] = m_str1;    //记录 OR 子式，并将 OR 子式的数目加 1
```

```
            plist2 - > AddString(m _ str1);          //在列表框 IDC _ LIST2 中增加新产生的 OR 子式
            plist2 - > SetCurSel(m _ NumbOr - 1);    //在列表框 IDC _ LIST2 中选择新增的 OR 子式
            MakeAndStr(p _ str1);                    //得到增加了 OR 子式后的 AND 子式
            //以下用新得到的 AND 子式修改列表框 IDC _ LIST1 中当前选择的 AND 子式
            plist1 - > DeleteString(nlist1);         //删除列表框 IDC _ LIST1 中的第 nlist1 选项
            if(nlist1 = = m _ NumbAnd - 1)           //如果删除的是最后一个项目
                plist1 - > AddString(m _ str1);      //在列表框中增加新产生的 AND 子式
            else   //如果删除的不是最后一个项目,在列表框相应位置插入新产生的 AND 子式
                plist1 - > InsertString(nlist1,m _ str1);
            plist1 - > SetCurSel(nlist1);            //仍然选择第 nlist1 个 AND 子式
            m _ strAnd[nlist1] = m _ str1;           //修改相应 AND 子式为修改后的 AND 子式
            MakeStr();                               //重新计算得到过滤条件式
        }
    }
```

OnOrAdd 函数的主要操作步骤如下：

① 得到在列表框 IDC _ LIST1 中选项的序号 nlist1。如果没有在列表框 IDC _ LIST1 中选择一个要增加 OR 子式的 AND 子式，则不能进行增加 OR 子式(基本条件子式)的操作，函数退出。

② 调用 MakeOrStr 函数来产生一个 OR 子式(基本条件子式)。如果 MakeOrStr 函数不能成功产生 OR 子式，OnOrAdd 函数退出。

③ 如果创建 OR 子式成功，将新建的 OR 子式记录在 CString 对象数组 m _ strOr 中，当前 AND 子式所包含的 OR 子式数目 m _ NumbOr 加 1，即在当前 AND 子式中增加了一个 OR 子式。将新建的 OR 子式增加到列表框 IDC _ LIST2 中，并在列表框中选择这个新增的项目。

④ 调用 MakeAndStr 函数，利用记录在 CString 对象数组 m _ strOr 中的 OR 子式重新生成 AND 子式，并用它来替换记录在 CString 对象数组 m _ strAnd 中的对应 AND 子式。在列表框 IDC _ LIST1 中修改当前选择的 AND 子式，修改的具体方法是：从列表框 IDC _ LIST 中删除这个项目，再增加或插入修改后的 AND 子式。

⑤ 调用 MakeStr 函数得到过滤条件式 m _ strFilter，并把这个过滤条件式填写到编辑框 IDC _ EDIT2 中。

(3) 增加 AND 子式

当按中 CFilterDlg 对话框(见 IDD _ FILTER 对话框资源)中的“增加 AND 子式”按钮(ID 号 ID _ AND _ ADD)时，会根据对话框中所做的选择产生一个基本条件子式，与运行“增加 OR 子式”功能时不同，不是在一个选定的 AND 子式上增加这个基本条件子式(OR 子式)，而是新增一个 AND 子式，这个 AND 子式中只包含了新增的基本条件子式。为了实现这个功能，用 ClassWizard 为“增加 AND 子式”按钮(ID _ AND _ ADD)增加一个 BN _ CLICKED 控制消息处理函数。增加完成后，在对话框类 CFilterDlg 中增加了消息处理函数的定义：

```
//{{AFX _ MSG(CFilterDlg)
virtual BOOL OnInitDialog();
```

```
afx_msg void OnOrAdd();
afx_msg void OnAndAdd();
//}}AFX_MSG
```

在实现文件 filterdlg.cpp 中,增加了映射机制:

```
BEGIN_MESSAGE_MAP(CFilterDlg, CDialog)
    //{{AFX_MSG_MAP(CFilterDlg)
    ON_BN_CLICKED(ID_OR_ADD, OnOrAdd)
    ON_BN_CLICKED(ID_AND_ADD, OnAndAdd)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 filterdlg.cpp 中,增加了函数空的实现代码。以下是经过修改后的实现代码:

```
void CFilterDlg::OnAndAdd()
{
    if(m_NumbAnd >= 14) //如果 AND 子式的数目超过 14 个,则给出提示信息并退出函数。
    {
        AfxMessageBox("最多只能有 14 个.AND.子式");
        return;
    }
    CString m_str1;
    CString& p_str1 = m_str1;
    BOOL yn = MakeOrStr(p_str1);                    //调用 MakeOrStr 函数生成基本条件子式
    if(yn)                                          //如果成功的生成了基本条件子式
    {
        m_strOr[0] = m_str1;                        //将基本条件子式记录为第 1 个 OR 子式
        m_NumbOr = 1;                               //OR 子式的数目为 1
        plist2->ResetContent();                     //清空列表框 IDC_LIST2
        plist2->AddString(m_str1);                  //将新增的 OR 子式增加到列表框 IDC_LIST2 中
        plist2->SetCurSel(0);                       //在列表框 IDC_LIST2 中选择新增加的 OR 子式
        MakeAndStr(p_str1);                         //调用函数 MakeAndStr 计算得到一个 AND 子式
        plist1->AddString(m_str1);                  //将新增的 AND 子式加到列表框 IDC_LIST1 中
        plist1->SetCurSel(m_NumbAnd); //在列表框 IDC_LIST1 中选择这个新增加的 AND 子式
        m_strAnd[m_NumbAnd++] = m_str1; //记录新增的 AND 子式,AND 子式的数目加 1
        MakeStr();                                  //调用 MakeStr 函数计算得到过滤条件式
    }
}
```

OnAndAdd 函数的主要工作步骤如下:

① 调用 MakeOrStr 函数产生一个基本条件子式,操作不成功时函数退出。

② 如果创建基本条件子式成功,将新建的基本条件子式记录在 CString 对象数组 m_strOr 的第一项(m_strOr[0]中),OR 子式的数目为 1。清空列表框 IDC_LIST2 中的所有项目,将新增的基本条件子式(OR 子式)增加到列表框中,并选择这个新增的 OR 子式。

③ 调用 MakeAndStr 函数,利用记录在 CString 对象数组 m _ strOr 中的 OR 子式生成一个新的 AND 子式,并在列表框 IDC _ LIST1 中增加并选择这个 AND 子式。将新产生的 AND 子式记录在 CString 对象数组 m _ strAnd 中,AND 子式的数目 m _ NumbAnd 加 1。

④ 调用 MakeStr 函数得到过滤条件式 m _ strFilter,并把这个过滤条件式填写到编辑框 IDC _ EDIT2 中。

16.3.6　删除和修改条件子式

在 CFilterDlg 对话框(见 IDD _ FILTER 对话框资源)中有三个按钮——"删除 OR 子式"(ID 号为 ID _ OR _ DELETE)、"删除 AND 子式"(ID 号为 ID _ AND _ DELETE)、"修改 OR 子式"(ID 号为 ID _ OR _ MODIFY)来完成对条件子式的删除和修改操作。用 ClassWizard 为这三个按钮增加 BN _ CLICKED 控制消息处理函数。增加完毕后,在对话框类 CFilterDlg 中增加函数的定义:

```
//{{AFX_MSG(CFilterDlg)
virtual BOOL OnInitDialog();
afx_msg void OnOrAdd();
afx_msg void OnAndAdd();
afx_msg void OnAndDelete();
afx_msg void OnOrDelete();
afx_msg void OnOrModify();
//}}AFX_MSG
```

在实现文件 filterdlg.cpp 中,增加映射机制:

```
BEGIN_MESSAGE_MAP(CFilterDlg, CDialog)
    //{{AFX_MSG_MAP(CFilterDlg)
    ON_BN_CLICKED(ID_OR_ADD, OnOrAdd)
    ON_BN_CLICKED(ID_AND_ADD, OnAndAdd)
    ON_BN_CLICKED(ID_AND_DEL, OnAndDelete)
    ON_BN_CLICKED(ID_OR_DEL, OnOrDelete)
    ON_BN_CLICKED(ID_OR_MODIFY, OnOrModify)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 filterdlg.cpp 中,增加了空的函数实现代码。为了实现具体的操作功能,分别对这几个消息处理函数的实现代码进行修改。

(1) OnAndDelete 函数

在按中 CFilterDlg 对话框(见 IDD _ FILTER 对话框资源)中的"删除 AND 子式"按钮时,执行消息处理函数 OnAndDelete,删除列表框 IDC _ LIST1 中选择的 AND 子式,并从 CString 对象数组 m _ strAnd 中删除对应的 AND 子式。修改后的函数实现代码如下:

```
void CFilterDlg::OnAndDelete()
```

```
{
    short nlist1 = plist1 - > GetCurSel();      //得到在列表框 IDC _ LIST1 中选项的序号
    if(nlist1 < 0)                              //如果没有在列表框 IDC _ LIST1 中选择 AND 子式,函数
                                                //返回
        return;
    plist1 - > DeleteString(nlist1);            //从列表框 IDC _ LIST1 中删除选择的 AND 子式
    m _ NumbAnd - - ;                           //AND 子式的数目减 1
    if(nlist1 = m _ NumbAnd)                    //如果删除的是 IDC _ LIST1 列表框中的最后一项
        plist1 - > SetCurSel(nlist1 - 1);       //选择列表框中的上一个选项
    else                                        //如果删除的不是 IDC _ LIST1 列表框中的最后一个选项
        plist1 - > SetCurSel(nlist1);           //选择列表框第 nlist1 个选项
    for(int i = nlist1;i < m _ NumbAnd;i++ )    //从 CString 对象数组 m _ strAnd 中删除第 nlist1
                                                //个 AND 子式
        m _ strAnd[i] = m _ strAnd[i + 1];
    m _ strAnd[m _ NumbAnd].Empty();
    OnSelchangeList1();      //分解 IDC _ LIST1 列表框中选择的 AND 子式为各个 OR 子式
    MakeStr();               //调用 MakeStr 函数,计算得到过滤条件式
}
```

OnAndDelete 函数的主要操作步骤如下：

① 从列表框 IDC _ LIST1 中得到当前选择的 AND 子式的序号,如果没有在列表框中选择 AND 子式,函数退出。

② 从列表框 IDC _ LIST1 删除选择的 AND 子式项,然后根据删除项的位置来设置当前的选项(如果删除的是 IDC _ LIST1 列表框的最后一个项目,则选择列表框的上一个项目,否则,仍然选择原来序号的选项)。

③ 将 AND 子式的数目 m _ NumbAnd 减 1,并通过对 CString 对象数组 m _ strAnd 的滚动,从对象数组 m _ strAnd 中删除要删除的 AND 子式。

④ 调用 OnSelchangeList1 函数,将列表框 IDC _ LIST1 中新选择的 AND 子式分解,将分解得到的 OR 子式(基本条件子式)增加到列表框 IDC _ LIST2 中。OnSelchangeList1 函数的实现参见 16.4.7 小节。

⑤ 调用 MakeStr 函数得到过滤条件式 m _ strFilter,并把这个过滤条件式填写到编辑框 IDC _ EDIT2 中。

(2) OnOrDelete 函数

在按中 CFilterDlg 对话框(见 IDD _ FILTER 对话框资源)中的"删除 OR 子式"按钮时,消息处理函数 OnOrDelete 被执行,用来删除组成当前 AND 子式(在列表框 IDC _ LIST1 中选择的 AND 子式)的一个 OR 子式(在列表框 IDC _ LIST2 中选择的 OR 子式)。OnOrDelete 函数的实现代码如下：

```
void CFilterDlg::OnOrDelete()
{
    int nlist1 = plist1 - > GetCurSel();  //得到在列表框 IDC _ LIST1 中的选项序号
```

```
    if(nlist1 < 0)                    //如果没有在 IDC_LIST1 中选择 AND 子式,函数返回
        return;
    int nlist2 = plist2 - > GetCurSel();  //得到在列表框 IDC_LIST2 中的选项序号
    if(nlist2 < 0)                    //如果没有在编辑框 IDC_LIST2 中选择 OR 子式,函数返回
        return;
    plist2 - > DeleteString(nlist2);      //从列表框 IDC_LIST2 中删除当前选项
    if(nlist2 = = m_NumbOr - 1) //如果删除的是列表框中最后一个项目,选择列表框上一个项目
        plist2 - > SetCurSel(nlist2 - 1);
    else              //如果删除的不是列表框中的最后一个项目,选择列表框中原来序号的项目
        plist2 - > SetCurSel(nlist2);
    m_NumbOr - - ;                    //OR 子式的数目减 1
    for(int i = nlist2;i < m_NumbOr;i++ )     //从 CString 对象数组 m_strOr 中删除相应的 OR 子式
        m_strOr[i] = m_strOr[i + 1];
    OnSelchangeList2();           //调用 OnSelchangeList2 函数,将 IDC_LIST2 中选择的 OR 子式分解
    CString m_str1;
    CString& p_str1 = m_str1;
    if(MakeAndStr(p_str1))            //如果产生 AND 子式,即删除 OR 子式后仍然存在 OR 子式
    {
        plist1 - > DeleteString(nlist1);      //从 IDC_LIST1 中删除选择的 AND 子式项
        if(nlist1 = = m_NumbAnd - 1)      //如果删除的是列表框 IDC_LIST1 中最后一个项目
            plist1 - > AddString(m_str1);//在 IDC_LIST1 中增加新产生的 AND 子式
        else                              //如果删除的不是列表框 IDC_LIST1 中的最后一个项目
            plist1 - > InsertString(nlist1,m_str1);//在相应位置插入新产生的 AND 子式
        plist1 - > SetCurSel(nlist1);         //在列表框 IDC_LIST1 中仍然选择原来的项目
        m_strAnd[nlist1] = m_str1;            //修改记录在 m_strAnd 中的 AND 子式
    }
    else                  //如果不能产生 AND 子式,即当前 AND 子式只有一个被删除的 OR 子式
        OnAndDelete();                    //调用 OnAndDelete 函数删除这个 AND 子式
    MakeStr();                            //调用 MakeStr 函数计算得到过滤条件式
}
```

OnOrDelete 函数的主要操作步骤如下：

① 从列表框 IDC_LIST1 中得到当前选项的序号，从列表框 IDC_LIST2 中得到当前选项的序号。如在任何一个列表框中没有选项，函数退出。

② 从列表框 IDC_LIST2 中删除当前的选项(OR 子式)，然后根据删除项在列表框 IDC_LIST2 中的位置来设置当前的选项(如果删除的是列表框 IDC_LIST2 的最后一个项目，则选择列表框中的前一个项目，否则，将选择原来序号的选项)。

③ 将 OR 子式的数目 m_NumbOr 减 1，并通过对 CString 对象数组 m_strOr 的滚动，从对象数组中删除要删除的 OR 子式(下标为 nlist2 的项)。

④ 调用 OnSelchangeList2 函数，将在列表框 IDC_LIST2 中选择的 OR 子式分解成字段名称、运算符、操作值的形式，并设置列表选择框 IDC_COMBO1、IDC_COMBO2 中的选项序号和编辑框 IDC_EDIT1 中的操作值。OnSelchangeList2 函数将在 16.4.7 小节介绍。

⑤ 调用 MakeAndStr 函数产生 AND 子式，如果能够得到 AND 子式(即删除 OR 子式后仍然存在 OR 子式)，修改列表框 IDC _ LIST1 中选择的 AND 子式项，并修改 CString 对象数组 m _ strAnd 中对应的 AND 子式。如果函数 MakeAndStr 函数不能产生 AND 子式(即原来的条件子式只包含一个 OR 子式，删除 OR 子式后不存在 OR 子式)，则调用函数 OnAndDelete 删除这个 AND 子式。

⑥ 调用 MakeStr 函数得到过滤条件式 m _ strFilter，并把这个过滤条件式填写到编辑框 IDC _ EDIT2 中。

(3) OnOrModify 函数

在按中 CFilterDlg 对话框(见 IDD _ FILTER 对话框资源)中的“修改 OR 子式”按钮时，消息处理函数 OnOrModify 被执行，用来修改组成当前 AND 子式(在列表框 IDC _ LIST1 中选择的 AND 子式)的一个 OR 子式(在列表框 IDC _ LIST2 中选择的 OR 子式)。OnOrModify 函数的实现代码如下：

```
void CFilterDlg::OnOrModify()
{
    int nlist1 = plist1 -> GetCurSel();    //得到在列表框 IDC _ LIST1 中的选项序号
    if(nlist1 < 0)                         //如果没有在 IDC _ LIST1 中选择 AND 子式，函数返回
        return;
    int nlist2 = plist2 -> GetCurSel();    //得到在列表框 IDC _ LIST2 中的选项序号
    if(nlist2 < 0)                         //如果没有在 IDC _ LIST2 中选择 OR 子式，函数返回
        return;
    CString m _ str1;
    CString& p _ str1 = m _ str1;
    BOOL yn = MakeOrStr(p _ str1);         //调用 MakeOrStr 函数来产生基本条件子式(OR 子式)
    if(yn)                                 //如果成功的产生了基本条件子式(OR 子式)
    {
        m _ strOr[nlist2] = m _ str1;      //修改记录在 CString 对象数组中的 OR 子式
        //以下对列表框 IDC _ LIST2 中当前选择的 OR 子式进行修改，修改为新产生的子式
        plist2 -> DeleteString(nlist2);
        if(nlist2 = = m _ NumbOr - 1
            plist2 -> AddString(m _ str1);
      else
          plist2 -> InsertString(nlist2,m _ str1);
      plist2 -> SetCurSel(nlist2);
      MakeAndStr(p _ str1);                //调用 MakeAndStr 函数产生 AND 子式
      //以下对列表框 IDC _ LIST1 中当前选择的 AND 子式进行修改，修改为刚产生的子式
      plist1 -> DelcteString(nlist1);      //从 IDC _ LIST1 中删除选择的 AND 子式项
      if(nlist1 = = m _ NumbAnd - 1)       //如果删除的是列表框 IDC _ LIST1 中最后一个项目
          plist1 -> AddString(m _ str1); //在列表框 IDC _ LIST1 中增加新产生的 AND 子式
      else                                 //如果删除的不是列表框 IDC _ LIST1 中的最后一个项目
          plist1 -> InsertString(nlist1,m _ str1);    //在相应位置插入新产生的 AND 子式
```

```
            plist1 -> SetCurSel(nlist1);          //在列表框 IDC_LIST1 仍然选择原来序号的项目
            m_strAnd[nlist1] = m_str1;            //修改记录在 m_strAnd 中的 AND 子式
            MakeStr();                            //调用 MakeStr 函数计算得到过滤条件式
        }
    }
```

OnOrModify 函数的主要操作步骤如下：

① 从列表框 IDC_LIST1 中得到当前选项的序号，从列表框 IDC_LIST2 中得到当前选项的序号。如在任何一个列表框中没有选择项目，函数退出。

② 调用 MakeOrStr 函数来产生一个基本条件子式(OR 子式)，如果不成功函数退出。

③ 如果创建基本条件子式成功，修改 CString 对象数组 m_strOr 中存储的相应 OR 子式为新产生的 OR 子式，在列表框 IDC_LIST2 中修改当前选择的 OR 子式为新产生的 OR 子式。对列表框 IDC_LIST2 当前选项的修改，是通过从列表框中删除当前选项，再增加或插入新产生的 OR 子式项来实现的。

④ 调用 MakeAndStr 函数，利用修改后的 OR 子式产生一个 AND 子式，修改列表框中选择的 AND 子式项，并修改 CString 对象数组 m_strAnd 中对应项存储的 AND 子式。

⑤ 调用 MakeStr 函数得到过滤条件式 m_strFilter，并把这个过滤条件式填写到编辑框 IDC_EDIT2 中。

16.3.7　条件子式的分解功能

在对话框类 CFilterDlg 中，除了以上讨论实现的功能之外，还应具有以下的操作功能：当在列表框 IDC_LIST1 中选择一个 AND 子式时(通过鼠标或键盘选择)，应用程序将选择的 AND 子式进行分解，将分解后得到的 OR 子式列在列表框 IDC_LIST2 中；当在列表框 IDC_LIST2 中选择一个 OR 子式时(通过鼠标或键盘选择)，应用程序将选择的 OR 子式分解成字段名称、运算符、操作值的格式，并对列表选择框 IDC_COMBO1、IDC_COMBO2 及编辑框 IDC_EDIT1 中的内容进行初始化。

为了完成以上的功能，用 ClassWizard 给列表框 IDC_LIST1、IDC_LIST2 分别增加一个 SEL_CHANGE 控制消息处理函数。增加完毕后，在对话框类 CFilterDlg 中增加了消息处理函数的定义：

```
//{{AFX_MSG(CFilterDlg)
virtual BOOL OnInitDialog();
afx_msg void OnOrAdd();
afx_msg void OnAndAdd();
afx_msg void OnAndDelete();
afx_msg void OnOrDelete();
afx_msg void OnOrModify();
afx_msg void OnSelchangeList1();
afx_msg void OnSelchangeList2();
//}}AFX_MSG
```

在实现文件 filterdlg.cpp 中，增加了映射机制：

```
BEGIN_MESSAGE_MAP(CFilterDlg, CDialog)
    //{{AFX_MSG_MAP(CFilterDlg)
    ON_BN_CLICKED(ID_OR_ADD, OnOrAdd)
    ON_BN_CLICKED(ID_AND_ADD, OnAndAdd)
    ON_BN_CLICKED(ID_AND_DEL, OnAndDelete)
    ON_BN_CLICKED(ID_OR_DEL, OnOrDelete)
    ON_BN_CLICKED(ID_OR_MODIFY, OnOrModify)
    ON_LBN_SELCHANGE(IDC_LIST1, OnSelchangeList1)
    ON_LBN_SELCHANGE(IDC_LIST2, OnSelchangeList2)
//}}AFX_MSG_MAP
```

在实现文件 filterdlg.cpp 中,增加了空的函数实现代码。分别对两个消息处理函数进行修改,得到修改后的函数实现代码。

(1) OnSelchangeList1 函数

当在 CFilterDlg 对话框(见 IDD_FILTER 对话框资源)中的列表框 IDC_LIST1 中选择一个选项时,OnSelchangeList1 函数被执行,将在列表框 IDC_LIST1 中选择 AND 子式进行分解,将分解后得到的各个 OR 子式在列表框 IDC_LIST2 中列出。以下是函数的实现代码:

```
void CFilterDlg::OnSelchangeList1()
{
    int n = plist1->GetCurSel();          //从 IDC_LIST1 列表框中得到选项的 0 基序号
    if(n<0)                               //如果没有选中列表框 IDC_LIST1 的 AND 子式
    {
        plist2->ResetContent();           //清空列表框 IDC_LIST2 中的项目
        return;                           //退出函数
    }
    BreakOrStr(m_strAnd[n]);              //调用 BreakOrStr 函数将 AND 子式分解成各个 OR 子式
    plist2->ResetContent();               //清空列表框 IDC_LIST2 中的各个项目
    for(int i=0;i<m_NumbOr;i++)           //将各 OR 子式加入到 IDC_LIST2 列表框
        plist2->AddString(m_strOr[i]);
}
```

OnSelchangeList1 函数的操作过程较为简单,可以通过函数中的注释语句来理解。这个函数除了可被 SEL_CHANGE 消息激活运行外,也可以在别的函数中被调用来完成其操作功能。

(2) OnSelchangeList2 函数

当在 CFilterDlg 对话框(见 IDD_FILTER 对话框资源)中的列表框 IDC_LIST2 中选择一个选项时(鼠标或键盘选择),OnSelchangeList2 函数被执行,将 OR 子式分解成字段名称、运算符、操作值的格式,并根据分解后的内容来初始化对话框控制 IDC_COMBO1、IDC_COMBO2、IDC_EDIT1 的状态。以下是函数的实现代码:

```
void CFilterDlg::OnSelchangeList2()
```

```
{
    short nIndex;
    int nlist2 = plist2 - > GetCurSel();  //得到列表框 IDC_LIST2 中选项的 0 基序号
    if(nList2 < 0)                        //如果没有在列表框 IDC_LIST2 选择一个 OR 子式,函数返回
        return;
    CString m_str2,m_str3;
    CString m_str1 = m_strOr[nList2]; //将选择的 OR 子式复制到字符串 m_str1
    m_str1.TrimLeft();                //去掉字符串前的空格
    m_str1.TrimRight();               //去掉字符串后的空格
    BOOL yn = 0;
    for(int i = 0;i < m_NumbOperator;i++ )      //对所有的运算符循环
    {
        n = m_str1.Find(m_strOperator[i],0); //在 OR 子式中查找第 i 个操作符
        if(n > 0)                         //如果在字符串中找到了运算符
        {
            nIndex = i;                   //找到的运算符的序号
            yn = 1;                       //标识找到了运算符
            break;                        //结束循环退出
        }
    }
    BOOL yn1 = 0;
    if(yn)                                //如果找到了运算符
    {
        pcom2 - > SetCurSel(nIndex);      //在列表选择框 IDC_COMBO2 中选择找到的运算符
        m_str2 = m_str1.Left(n);          //将 OR 子式中运算符前的字符串拷贝到 m_str2
        for(i = 0;i < m_NumbField;i++ )   //对所有的记录集合字段循环
        {
            if(! m_str2.Compare(pFieldInfo[i].m_strName))
            //如果字符串 m_str2 与记录集合对象的第 i 个字段名称相同
            {
                yn1 = 1;                  //标识找到了字段名称
                break;                    //结束循环退出
            }
        }
        if(yn1)                           //如果 OR 子式中,运算符前的字符串是某个字段的名称
            pcom1 - > SetCurSel(i);       //在列表选择框 IDC_COMBO1 中选择找到的字段名称
        else                              //如果没有找到记录集合对象的字段
            pcom1 - > SetCurSel( - 1);    //在列表选择框 IDC_COMBO1 中,不选择一个选项
        m_str3 = m_str1.Mid(n + m_OperatorLong[nIndex]);   //将运算符后字符串拷贝到 m_str3
        SetDlgItemText(IDC_EDIT1,m_str3); //将运算符后的字符串作为操作值写到 IDC_EIDT1
    }
    else          //如果在 OR 子式中没有找到运算符,将整个字符串写到编辑框 IDC_EDIT1 中
        SetDlgItemText(IDC_EDIT1,m_str1);
```

}

OnSelchangeList2 函数的主要工作步骤为：

① 得到在列表框 IDC_LIST2 中选项的 0 基序号 nlist1，如果没有在列表框 IDC_LIST2 中选择 OR 子式，函数退出。

② 根据 nlist1 序号得到 OR 子式，并将这个条件子式拷贝到字符串 m_str1，去掉字符串 m_str1 前后的空格。

③ 依次在字符串 m_str1 中寻找各个运算符，如果没有找到运算符(yn=0)，将字符串 m_str1 的内容写入编辑框 IDC_EDIT1 中并退出函数。

④ 如果从 m_str1 中找到了运算符，根据找到的运算符的顺序在列表选择框 IDC_COMBO2 中选择这个运算符。

④ 将字符串 m_str1 中运算符前的字符串拷贝到字符串 m_str2 中，并将这个字符串依次与记录集合对象的各个字段名称进行比较，如果这个字符串与某个字段名称相同，就在列表选择框 IDC_COMBO1 中选择这个字段名称，否则，不在列表选择框 IDC_COMBO1 中做选择。

⑤ 将字符串 m_str1 中运算符以后的字符串作为操作值填写到编辑框 IDC_EDIT1 中。

在 OR 子式中寻找运算符时，必须注意各个运算符的顺序，如果一个运算符是另外一个运算符的一个子串(如操作符“>”是运算符“>=”的一个子串)，进行运算符的查找操作时，这个运算符必须在另外一个运算符后进行查找(如在 OR 子式中查找运算符时，必须先查找运算符“>=”，后查找“>”运算符)，否则会引起分解错误。在初始化字符串数组 m_strOperator 中存储的各个运算符时(参见 14.2 小节)，已经考虑了各个运算符的排列顺序。

16.3.8　实现过滤操作

在退出过滤条件编辑对话框前，为了能够检查创建的过滤条件式的合法性，以及删除在对话框类 CFilterDlg 中动态分配的空间，在对话框类 CFilterDlg 中为按钮“查询”(ID 号为 IDOK)和“放弃退出”(ID 号为 IDCANCEL)增加消息处理函数。因为这两个按钮的 ID 号是 MFC 中的两个标准 ID 号，已经在 CDialog 类定义了消息处理函数，所以为这两个按钮创建消息处理函数时，实际上是在当前对话框类 CFilterDlg 中重载了 CDialog 类的消息处理函数。操作完成后，在对话框类 CFilterDlg 中增加了重载函数的定义：

```
//{{AFX_MSG(CFilterDlg)
virtual BOOL OnInitDialog();
afx_msg void OnOrAdd();
afx_msg void OnAndAdd();
afx_msg void OnAndDelete();
afx_msg void OnOrDelete();
afx_msg void OnOrModify();
afx_msg void OnSelchangeList1();
afx_msg void OnSelchangeList2();
virtual void OnOK();
virtual void OnCancel();
//}}AFX_MSG
```

在实现文件 filterdlg.cpp 中，增加了函数的实现代码。为了完成具体的操作功能，需要对实现代码进行修改。

(1) OnOK 函数

当选择运行对话框中的“查询”按钮时，OnOK 函数被执行。修改后的函数实现代码如下：

```
void CFilterDlg::OnOK()
{
    GetDlgItemText(IDC_EDIT2,m_strFilter);//从编辑框 IDC_EDIT2 读出过滤条件式到 m_strFilter
    if(m_pSet->IsOpen())                  //如果打开了记录集合对象，则关闭记录集合对象
        m_pSet->Close();
    CString m_str=m_pSet->m_strFilter; //保存记录集合对象原来的过滤条件式
    m_pSet->m_strFilter=m_strFilter;//将记录集合对象的过滤条件设置为当前组织的过滤条件
    if(m_pSet->Open())                    //如果打开记录集合对象成功
    {
        for(int i=0;i<10;i++)             //删除动态分配的字符串数组
            free(m_strOperator[i]);
        EndDialog(IDOK);                  //退出对话框
    }
    else                        //如果打开记录集合对象不成功(记录集合对象的过滤条件式有错误)
        m_pSet->m_strFilter=m_str;    //恢复记录集合对象原来的过滤条件式
}
```

OnOK 函数的主要操作步骤如下：

① 从编辑框 IDC_EDIT2 中读出组织完成的过滤条件式 m_strFilter，将记录集合对象的过滤条件设置成这个过滤条件式，在改变记录集合对象的过滤条件前，首先将记录集合对象原来的过滤条件保存在字符串 m_str 中。

② 关闭记录集合对象，然后再重新打开记录集合对象，在打开过程中完成过滤操作。如果打开记录集合对象成功，表明组织完成的过滤条件式正确，删除动态分配的字符串数组，退出对话框。

③ 如果打开记录集合对象操作不成功，表明组织完成的过滤条件式存在语法错误(一般会在屏幕上提示出错信息)，将记录集合对象中的过滤条件式恢复成原来的内容，不退出对话框，继续进行过滤条件的编辑操作。

(2) OnCancel 函数

当按中 CFilterDlg 对话框(见 IDD_FILTER 对话框资源)中的“放弃退出”按钮时，OnCancel 函数被执行，放弃在对话框中编辑的过滤条件式，释放动态分配的空间并退出对话框。以下是 OnCancel 函数的实现代码：

```
void CFilterDlg::OnCancel()
{
    for(int i=0;i<10;i++)          //删除动态分配的字符串数组
```

```
        free(m_strOperator[i]);
    EndDialog(IDCANCEL);        //退出对话框
}
```

在 OnCancel 函数中，删除在对话框类 CFilterDlg 中动态分配的字符串数组并退出对话框，m_pSet 指向的记录集合对象中的过滤条件式(m_pSet->m_strFilter)没有发生变化。

本节在对话框类 CFilterDlg 中实现了交互编辑过滤条件的各种功能。除此之外，还可以在对话框类 CFilterDlg 中增加一些函数来完成其他功能。例如，可以增加程序代码来控制对话框中各个操作按钮的状态，根据实际的运行状态来动态确定各个按钮是否能够执行(对于在某种状态下不能运行的按钮，变暗禁止操作)，提高系统的可操作性。读者可自己去实现这些功能，在此不再具体讨论实现。

16.4　在数据库浏览视图中实现过滤功能

本节将在数据库浏览视图中，利用对话框类 CFilterDlg，来具体组织过滤操作功能。

16.4.1　增加一个操作菜单

为了在应用程序 Draw 的浏览视图中完成数据库的过滤和排序功能，本小节将在应用程序 Draw 的数据库浏览视图操作界面上增加一个菜单，来激发完成针对数据库的过滤和排序操作。

(1) 增加菜单资源

首先在浏览视图采用的菜单资源 IDR_DRAWDATA 中增加一个菜单“过滤及排序”。在这个菜单中包含两个菜单项“过滤操作...”和“排序操作...”(如图 16-3)。使用 AppStudio 可视化编辑工具增加菜单后，在资源文件 Draw.rc 中可以找到增加菜单的代码描述：



图 16-3　IDR_DRAWDATA 菜单资源的过滤及排序菜单

```
IDR_DRAWDATA MENU PRELOAD DISCARDABLE
BEGIN
    ......//省略以上代码
    POPUP "过滤及排序"
    BEGIN
        MENUITEM "过滤操作...",        ID_RECORD_FILTER
        MENUITEM "排序操作...",        ID_RECORD_SORT
    END
    ......//省略以下代码
END
```

在资源 ID 号定义文件 resource.h 中，可以找到对菜单项 ID 号的定义：

```
# define ID_RECORD_FILTER        32796
# define ID_RECORD_SORT          32790
```

(2) 增加消息处理函数

用 ClassWizard 给菜单项“过滤操作...”和“排序操作...”在视图类 CDatabaseView 中增加消息处理函数。操作完毕，在视图类 CDatabaseView 中增加了消息处理函数的定义：

```
//{{AFX_MSG(CDatabaseView)
.....//省略
afx_msg void OnRecordFilter();
afx_msg void OnRecordSort();
//}}AFX_MSG
```

在实现文件 filterdlg.cpp 中，增加了消息映射机制：

```
BEGIN_MESSAGE_MAP(CDatabaseView, CScrollView)
    //{{AFX_MSG_MAP(CDatabaseView)
    .....//省略以上代码
    ON_COMMAND(ID_RECORD_FILTER, OnRecordFilter)
    ON_COMMAND(ID_RECORD_SORT, OnRecordSort)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 filterdlg.cpp 中，增加了消息处理函数 OnRecordFilter 和 OnRecordSort 空的实现代码。在下一小节，将对 OnRecordFilter 函数的实现代码进行修改，完成数据库过滤功能。在 17.4.2 小节，将对 OnRecordSort 函数的实现代码进行修改，完成数据库排序功能。

16.4.2　实现数据库过滤操作功能

在数据库浏览视图中，以如下的步骤实现过滤操作功能：当运行应用程序 Draw 中的数据库浏览视图中的“过滤及排序”菜单下的“过滤操作...”菜单项时，能够调用 CFilterDlg 对话框交互编辑过滤条件，并利用得到的过滤条件对记录集合实现过滤操作。为了完成以上功能，修改“过滤操作...”菜单项的消息处理函数 OnRecordFilter(见 16.5.1 小节)的实现代码如下：

```
void CDatabaseView::OnRecordFilter()
{
    CFilterDlg dlg(m_pSet1);      //定义一个对话框类 CFilterDlg 对象
    if(dlg.DoModal() == IDOK)  //调用对话框，当运行“过滤”按钮退出对话框时
    {
        m_pSet1->m_strFilter = dlg.m_strFilter;              //设置记录集合对象的过滤条件
        Init2();                    //调用 Init2 函数设置视图滚动区域的纵向参数
        SetScrollSizes(MM_TEXT, sizeTotal, sizePage, sizeLine);   //设置视图的滚动区域
        Invalidate();               //重画视图
```

```
    }
}
```

在 OnRecordFilter 函数中，首先定义一个对话框类 CFilterDlg 对象 dlg，m _ pSet1 是指向当前视图浏览的记录集合对象的指针，作为参数传入了 dlg 对象。启动对话框，当以 IDOK 号退出对话框(即在过滤条件编辑对话框中通过运行“过滤”按钮退出对话框)时，用对话框中组织产生的过滤条件作为记录集合对象的过滤条件，然后调用 Init2 函数对视图滚动区域的纵向参数进行初始化，得到新的记录集合对象和数据库滚动视图，并对视图进行重画。

对应用程序 Draw 编译后运行，在 CDatabaseView 浏览视图中，运行“数据过滤...”菜单进入到过滤条件编辑对话框。在对话框中编辑组织一个过滤条件，当运行对话框的“过滤”按钮退出对话框时，可以看到数据库浏览视图发生变化，在浏览视图中列出的只是符合过滤条件的记录。

16.5 增强过滤条件编辑器的功能

本章以上各节，讨论实现了一个通用的过滤条件编辑器，并在数据库浏览视图中利用这个过滤条件编辑器实现了过滤操作功能。过滤条件编辑器，为数据库应用程序过滤功能设计提供了良好的程序框架和功能基础，能够满足数据库应用程序普通过滤操作功能的需要。在实际的数据库应用程序中，根据具体情况，还需要扩充通用过滤条件编辑器的功能。

16.5.1 增强过滤条件分解能力

应该说，一个过滤条件编辑器能够处理的过滤条件的复杂程度，与应用程序能够提供的过滤条件分解能力有根本的关系。在 16.3 节中实现的过滤条件分解能力是相对较弱的，它只能分析固定层次的过滤条件。

(1) 过滤条件分解能力的局限性

本章实现的过滤条件编辑器，能够处理的过滤条件层次较少，只能处理两个固定层次的过滤条件。例如，对于以下的过滤条件式，就不能通过本章的过滤条件编辑器来实现和分解：

```
((population > 100) AND (area > 50)) OR ((population < 50) AND (area < 30))
```

其根本原因在于，本章的过滤条件编辑器只能编辑两个层次的过滤子式，而且两个层次上分别只能使用 AND 和 OR 中的一个连接符。

本章实现的过滤条件编辑器，对过滤条件的分解操作只是按照组织过滤条件的层次进行的，而不是一个完美的条件分解器。过滤条件编辑器对过滤条件进行第一个层次的代码分解时，通过查找“ AND ”字符串来将代码分解为几段，这种代码分解方法因为没有处理利用括号形成的组织层次，在某些情况下会产生错误。例如，如果过滤条件式中包含不是连接符的“AND”字符串时，如对于以下的过滤条件式：

```
"(city _ name = 'beijing AND shanghai') AND (population > 100)"
```

进行条件分解时，会将过滤条件式分解成三个 OR 子式，出现分解错误。这也是本章组织

和分解过滤条件式时，在 AND 和 OR 连接符前后都加空格的原因。在连接符 AND 和 OR 的前后增加空格，可以将发生分解错误的几率降低到较小的程度。但这样做造成的缺陷是：对于合法的过滤条件式，当在连接符 AND 和 OR 前后没有空格时，过滤条件编辑器就不能正确对这个过滤条件式进行分解。

本章实现的过滤条件编辑器，只能分解按照本身的层次和规则组成的过滤条件。所以在过滤条件编辑对话框中，将编辑控制 IDC_EDIT2 设置成只读状态，不能手工对过滤条件式进行修改，防止手工书写的过滤条件不能被过滤条件编辑器正确分解。

(2) 增强过滤条件分解能力

理想的过滤条件编辑器应该能够正确地对符合 SQL 语法规则的过滤条件式进行组织和分解。为了达到这种能力，一方面需要对过滤条件编辑器能够处理的运算符进行扩充(能够处理 SQL 函数等)，另外一个方面需要能够任意处理过滤条件式的层次。

为了能够任意处理过滤条件式的层次，需要在对过滤条件式进行分解时，考虑括号的层次以及各种运算符的优先级。这种分解操作比较复杂，实际上是在实现一个与 SQL 语言类似的解释性语言。有兴趣的读者，可以参考作者编著的《Microsoft Visual C++ 图形系统开发基础》，在该书的 15.4 节中详细介绍了代码分析的机理和具体的实现方法，在掌握这些方法的基础上，可以编制出完全分解过滤条件式的过滤条件编辑器。

16.5.2　增强交互式编辑界面

过滤条件编辑器的交互式编辑界面应该是与过滤条件的分解能力一致的。即利用这个交互式编辑界面能够简捷而准确的表示出分解后的过滤条件。只增强过滤条件的分解能力而不改善交互式编辑界面，就不能在编辑界面上正确表示分解后的过滤条件，也就不能去组织复杂的过滤条件；而不增强过滤条件的分解能力而片面的增强过滤条件交互式编辑界面，就会造成过滤条件编辑器不能正确分解交互式编辑界面能够组织的过滤条件式，也就不能对组织完成的过滤条件式进行修改，这样的过滤条件编辑器也没有太大的意义。

对于一个完成一定功能的交互式编辑界面，需要努力提高编辑界面的可操作性。对于本章实现过滤条件编辑器的对话框类 CFilterDlg，至少需要在以下两个方面对编辑界面进行扩充：

(1) 控制各按钮的操作状态

在对话框界面中编辑过滤条件时，需要根据当时的操作状态，来决定各个对话框按钮是否能够执行(不能执行的变暗)。例如，在对话框界面上，在没有选择字段名称、运算符，或者没有在编辑框 IDC_EDIT1 中填入操作值的情况下，“增加 AND 子式”、“增加 AND 子式”按钮应该处于失效不能执行状态。由于篇幅所限，本章没有列出对各个按钮进行状态控制的代码，读者可自行增加这些内容，提高程序的可操作性。

(2) 合法性检查

在对话框界面中交互组织过滤条件时，应该对产生的过滤条件基本子式进行一些合法性

检查。例如,对于字符类型的字段来说(如 city _ name),利用“ > ”运算符组成的过滤条件基本子式没有意义,如:

```
city _ name > "北京"
```

所以,在产生过滤条件基本子式的过程中,需要进行合法性检查,或加入其他预防措施,如当“字段名称”列表选择框中选择的字段是字符类型时,禁止从“操作符”列表选择框中选择“ > ”等操作符选项。

16.5.3 存储过滤条件

本章实现的过滤操作功能,在每次进行过滤操作前都需要调用过滤条件编辑器来组织过滤条件,记录集合对象只能记录当前的过滤条件。在实际的数据库应用程序设计中,对于每一个记录集合对象,建立一个 CString 对象数组来存储针对这个记录集合对象的过滤条件,提供一个交互式操作界面对这个 CString 对象数组进行增加、修改、删除等操作功能,通过调用过滤条件编辑器在 CString 对象数组中增加新的过滤条件或对既有过滤条件进行修改。这样,在进行过滤操作时,可以从这个 CString 对象数组中选择已有的过滤条件完成过滤操作,而没有必要每次过滤操作前,必须调用过滤条件编辑器组织过滤条件。

通过建立一个记录集合基类,在记录集合基类中建立 CString 对象数组和提供各种过滤条件管理功能,供所有从这个基类派生的记录集合类使用。记录集合基类的创建方法见本书的第 18 章 18.2.2 小节。

第 17 章　通用数据库排序功能

本章将在应用程序 Draw 中分析实现通用数据库排序功能。通过设计实现一个排序子式编辑器，为任意的记录集合对象组织排序子式。通过组织的排序子式，对显示在浏览视图中的记录集合进行排序。

17.1　排序操作的方式

在数据库应用程序中，对于记录集合的排序操作，一般采用两种方式：固定方式和通用方式。以下对这两种操作方式进行介绍。

(1) 固定排序方式

固定排序方式指的是预先确定要用来排序的字段和排序顺序。例如，对于应用程序 Draw 中讨论的 CData1Set 类对象，如果要以 population 字段降序来对记录集合进行排序，则在应用程序中可以通过以下代码来实现：

```
if(m_pSet->IsOpen())                    //如果打开了记录集合对象
    m_pSet->Close();                    //关闭记录集合对象
m_pSet->m_strSort="population DESC";    //设置排序字段和顺序
m_pSet->Open();                         //重新打开记录集合对象
```

以上程序代码中，m_pSet 指向要进行排序操作的记录集合对象，首先关闭记录集合对象，然后将记录集合对象的 m_strSort 成员变量设置成进行排序操作的 SQL 子式(排序子式)，当重新打开记录集合对象时，就会得到按照 population 字段降序排列的记录集合。

在数据库应用程序中，可以利用固定排序方式来实现一些固定的排序功能。在很多情况下，采用的是一种半固定的方式，即在排序操作前首先通过交互操作界面选择排序字段和排序方式，动态得到用来排序操作用的排序子式，完成排序操作。

(2) 通用排序方式

通用排序方式的含义是，只要能够得到指向记录集合对象的指针，就能够提供一个交互式操作界面来组织排序用的排序子式，实现任何方式的排序功能。同时，这个交互式操作界面能够对排序子式进行分解，用交互式状态表示分解后的排序字段和排序顺序。所谓“通用”，一方面是指这个排序子式编辑器与具体的记录集合对象无关；另一个方面是指可以用它来组织任何形式的排序子式。

本章将具体实现一个用来组织排序子式的对话框类(排序子式编辑器)，并在数据库浏览视图中调用这个对话框类来产生和修改排序子式，实现对记录集合的排序操作。

17.2　通用排序子式编辑器的实现

与第 16 章设计一个用来编辑过滤条件的对话框类 CFilterDlg 类似，在本节将在应用程序 Draw 中实现一个编辑排序子式的对话框类。

17.2.1　创建一个对话框类

本小节，将在应用程序 Draw 中创建一个用来编辑排序子式的对话框类。

(1) 创建对话框资源

为了能够创建编辑排序子式的对话框类，首先需要创建一个对话框资源。用 ResourceView 和可视编辑工具 AppStudio 编辑完成一个如图 17-1 所示的对话框资源。对话框资源的 ID 号为 IDD _ SORT，在资源文件 draw.rc 中可以找到对话框资源 IDD _ SORT 的描述代码：

图 17-1　IDD _ SORT 对话框资源

```
IDD_SORT DIALOG DISCARDABLE 0, 0, 265, 149
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON   "排序",IDOK,196,99,50,14
    PUSHBUTTON      "放弃",IDCANCEL,196,119,50,14
    LISTBOX         IDC_LIST1,12,33,69,61,LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_
                    TABSTOP
    LTEXT           "排序字段列表",IDC_STATIC,21,19,56,12
    EDITTEXT        IDC_EDIT1,16,104,160,16,ES_AUTOHSCROLL
    COMBOBOX        IDC_COMBO1,98,33,86,67,CBS_DROPDOWNLIST | WS_VSCROLL | WS_
                    TABSTOP
    LTEXT           "字段名称",IDC_STATIC,99,20,50,9
    PUSHBUTTON      "增加",IDC_SORT_ADD,99,56,47,13
```

```
    CONTROL         "降序",IDC_CHECK1,"Button",BS_AUTOCHECKBOX | WS_TABSTOP,203,
                    34,41,10
    PUSHBUTTON      "插入",IDC_SORT_INSERT,155,56,47,13
    PUSHBUTTON      "修改",IDC_SORT_MODIFY,99,76,47,13
    PUSHBUTTON      "删除",IDC_SORT_DELETE,155,76,47,13
END
```

对话框资源中的 ID 号在 resource.h 文件中被定义值。在创建 IDD_SORT 对话框资源时注意,对于对话框资源中的列表框 IDC_LIST1 和列表选择框 IDC_COMBO1,在创建时不要选择排序风格(见 16.2.1 小节)。对于列表选择框 IDC_COMBO1,在创建时要选择"DROP LIST"风格。

(2) 创建一个对话框类

利用对话框资源 IDD_SORT,应用 ClassWizard 来产生一个对话框类 CSortDlg,类的头文件在 sortdlg.h,实现文件在 sortdlg.cpp。产生对话框类的具体过程可参考第 6 章 6.4.4 小节。以下是在头文件 sortdlg.h 中的对话框类 CSortDlg 的定义代码:

```
class CSortDlg : public CDialog
{
// Construction
public:
    CSortDlg(CWnd * pParent = NULL); // standard constructor

public:
// Dialog Data
    //{{AFX_DATA(CSortDlg)
    enum { IDD = IDD_SORT };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSortDlg)
    protected:
    virtual void DoDataExchange(CDataExchange * pDX); // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CSortDlg)
    //}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
};
```

在实现文件 sortdlg.cpp 中，有对话框类 CSortDlg 的实现代码。下面列出的是实现文件 sortdlg.cpp 的代码列表：

```
#include "stdafx.h"
#include "draw.h"
#include "SortDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////
// CSortDlg dialog

CSortDlg::CSortDlg(CWnd* pParent /* = NULL*/)
    : CDialog(CSortDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CSortDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CSortDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CSortDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CSortDlg, CDialog)
    //{{AFX_MSG_MAP(CSortDlg)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////
// CSortDlg message handlers
```

(3) 对话框类 CSortDlg 中要实现的主要功能

对于对话框类 CSortDlg，在设计完成后应该能够提供如下的操作功能：在如图 17-1 所示的对话框中，在“排序字段”列表框 IDC_LIST1 中，按顺序列出各个排序用的字段，在字段名称前

面加符号“↑”或“↓”，来表示这个字段是升序还是降序排序字段。在“字段名称”列表选择框 IDC _ COMBO1 中，列出了要进行排序操作的记录集合对象所包含的所有字段，“降序”检查框 IDC _ CHECK1 用来设置一个排序字段是升序还是降序。在这个对话框界面中，能够提供如下的操作功能：

① 增加排序字段。从“字段名称”列表选择框 IDC _ COMBO1 中选择一个字段，用“降序”检查框 IDC _ CHECK1 来设置排序顺序，按“增加”按钮，可以在已经存在的各排序字段后，增加一个排序字段。

② 插入排序字段。从“字段名称”列表选择框 IDC _ COMBO1 中选择一个字段，用“降序”检查框 IDC _ CHECK1 来设置排序顺序，在“排序字段”列表框 IDC _ LIST1 中选择一个字段，按“插入”按钮，就会在 IDC _ LIST1 列表框中选择的字段前插入一个排序字段。

③ 修改排序字段。在“排序字段”列表框中选择一个已经存在的排序字段，从“字段名称”列表选择框 IDC _ COMBO1 中选择一个字段，用“降序”检查框 IDC _ CHECK1 来设置排序顺序，按“修改”按钮，就会修改在 IDC _ LIST1 选择的排序字段为当前的排序字段(及排序顺序)。

④ 删除排序字段。在“排序字段”列表框 IDC _ LIST1 中选择一个排序字段，按“删除”按钮，就会删除这个排序字段。

⑤ 产生排序子式。能够根据在对话框中选择的排序字段和排序顺序，产生一个排序子式。

⑥ 分解排序子式。能够把一个排序子式分解成各个排序字段和排序顺序的形式进行编辑。

以下各小节，将对对话框类 CSortDlg 进行修改，实现以上介绍的各项功能。

17.2.2　定义和初始化对话框类的成员变量

为了在对话框类 CSortDlg 中实现上一小节提出的各种功能，在对话框类 CSortDlg 中定义多个成员变量，并在对话框类的有关成员函数中对这些变量进行初始化。

(1) 定义成员变量

在对话框类 CSortDlg 中，定义以下的成员变量(黑体部分)：

```
class CSortDlg : public CDialog
{
// Construction
public:
    CSortDlg(CWnd * pParent = NULL); // standard constructor

public:
    CString m_strSort;
protected:
    DataISet * m_pSet;
    short m_NumbSort;
    CString strSort[10];
```

```
    BOOL b_Sort[10];
    short m_NumbField;
    CODBCFieldInfo pFieldInfo[200];
    CListBox * plist1;
    CComboBox * pcom1;
.....//省略以下代码
}
```

CString 对象 m_strSort 用来记录在对话框中组织的排序子式。

CDataISet 类指针 m_pSet 指向要进行排序的记录集合对象。为了能够在对话框类 CSortDlg 中识别 CDataISet 类,在头文件 sortdlg.h 中对话框类 CSortDlg 的定义前加引用代码:

```
class CDataISet;
```

short 型变量 m_NumbSort 用来记录排序字段的数目,CString 对象数组 strSort 用来记录各个排序字段的名称,BOOL 型数组 b_Sort 用来记录 strSort 数组中对应排序字段的排序顺序。

short 型变量 m_NumbField 用来存储记录集合对象包含的字段数目,CODBCFieldInfo 型数组 pFieldInfo 用来存储记录集合对象中各个字段的信息。

CListBox 类指针 plist1 用来记录指向列表框 IDC_LIST1 的指针,CComboBox 类指针 pcom1 用来记录指向列表选择框 IDC_COMBO1 的指针。

(2) 修改构造函数

为了将指向进行排序操作的记录集合对象的指针传入对话框类 CSortDlg 对象中,对对话框类 CSortDlg 中构造函数的定义进行修改(也可重载一个新的构造函数):

```
CSortDlg(DataISet * pSet,CWnd * pParent = NULL); // standard constructor
```

对构造函数的实现代码进行修改,修改后的实现代码如下:

```
CSortDlg::CSortDlg(DataISet * pSet,CWnd * pParent /* = NULL */)
    : CDialog(CSortDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CSortDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    m_pSet = pSet;                          //将 m_pSet 指向通过参数传来的记录集合对象
    if(! m_pSet->IsOpen())                  //如果没有打开记录集合对象,则打开记录集合对象
        m_pSet->Open();
    m_strSort = m_pSet->m_strSort;                  //得到记录集合对象的排序子式
    m_NumbField = m_pSet->GetODBCFieldCount(); //得到记录集合包含的字段数目
    CODBCFieldInfo fieldinfo1;
    CODBCFieldInfo& fieldinfo = fieldinfo1;
    for(int i = 0;i < m_NumbField;i++)    //对记录集合所有的字段循环,得到各个字段的信息
    {
```

```
        m_pSet->GetODBCFieldInfo(i,fieldinfo);      //得到第 i 个字段的信息
        pFieldInfo[i] = fieldinfo1;      //将记录集合第 i 个字段的信息保存在对象数组 pFieldInfo 中
    }
    m_NumbSort = 0;                  //排序字段的数目为 0
}
```

在以上代码中,通过对话框类 CSortDlg 构造函数的参数初始化了对话框类 CSortDlg 中的指针 m_pSet,使其指向了进行排序操作的记录集合对象,通过这个指针调用记录集合类的操作功能。在构造函数中,打开记录集合对象,将记录集合对象中的排序子式拷贝到对话框类的成员变量 m_strSort 中,得到了记录集合对象中字段的数目 m_NumbSort,并获得且记录了各个字段的信息到 pFieldInfo 数组中。

(3) 重载和修改 OnInitDialog 函数

为了对对话框操作界面进行初始化,用 ClassWizard 在对话框类 CSortDlg 中增加 WM_INITDIALOG 窗口消息的处理函数,即在对话框类 CSortDlg 中重载 CDialog 类的虚函数 OnInitDialog。操作完毕,在对话框类 CSortDlg 中增加了函数的定义:

```
//{{AFX_MSG(CSortDlg)
virtual BOOL OnInitDialog();
//}}AFX_MSG
```

在实现文件 sortdlg.cpp 中,对 OnInitDialog 函数的实现代码进行修改。以下是经过修改后的函数实现代码:

```
BOOL CSortDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    plist1 = (CListBox *)GetDlgItem(IDC_LIST1);      //获得指向列表框 IDC_LIST1 的指针
    pcom1 = (CComboBox *)GetDlgItem(IDC_COMBO1);//获得指向列表选择框 IDC_COMBO1 的指针
    plist1->ResetContent();              //清空列表框 IDC_LIST1
    pcom1->ResetContent();               //清空列表选择框 IDC_COMBO1
    BreakSortStr(m_strSort);             //将过滤子式分解
    CString m_str;
    for(int i=0;i<m_NumbSort;i++)        //对所有的排序字段循环
    {
        if(b_Sort[i])                    //如果排序字段的排序顺序是升序
            m_str="↑    ";               //增加一个上箭头字符串
        else                             //如果排序字段的排序顺序是降序
            m_str="↓    ";               //增加一个下箭头字符串
        m_str+=strSort[i];               //在字符串中增加排序字段的名称
        plist1->AddString(m_str);        //在列表框 IDC_LIST1 中增加项目
    }
    for(i=0;i<m_NumbField;i++)           //对记录集合所有字段循环
                                         //将字段名加入列表选择框 IDC_COMBO1
```

```
        pcom1 -> AddString(pFieldInfo[i].m_strName);
    MakeStr();                          //产生排序子式
    return TRUE; // return TRUE unless you set the focus to a control
}
```

OnInitDialog 函数的主要操作步骤如下：

① 得到指向对话框中列表框 IDC_LIST1 的指针 plist1，得到指向列表选择框 IDC_COMBO1 的指针 pcom1，清空这两个列表中的项目。

② 调用 BreakSortStr 函数，把排序子式 m_strSort 分解成各个排序字段及排序顺序的形式，由 CString 对象数组 strSort 和 BOOL 型数组 b_sort 存储，排序字段的数目为 m_NumbSort。BreakSortStr 函数的功能和实现代码参见 17.2.3 小节。

③ 在列表框 IDC_LIST1 中加入各排序字段。对于升序字段，在其字段名称前加字符串“↑”；对于降序字段，在字段名称前加字符串“↓”。

④ 将记录集合对象包含的所有字段的名称加入到列表选择框 IDC_COMBO1 中。

⑤ 调用 MakeStr 函数，重新得到排序子式。MakeStr 函数的功能和实现代码见 17.2.3 小节。

17.2.3 产生和分解排序子式

为了在对话框类 CSortDlg 中实现对排序子式的编辑功能，必须能够将通过交互式操作选择的排序字段(及排序顺序)生成排序子式，同时能够将排序子式分解成各个排序字段和排序顺序的形式，通过对话框界面来交互表示。为了实现这两个功能，在对话框类 CSortDlg 中定义两个函数：

```
public:
    void MakeStr();
    int BreakSortStr(CString m_str);
```

以下讨论实现这两个函数的实现代码。

(1) MakeStr 函数

MakeStr 函数实现的功能是：利用记录在 strSort 和 b_sort 数组中的关于排序字段的信息，产生排序子式。如当 m_NumbSort = 2，CString 对象数组 strSort 前两项内容为“population”和“Area”，BOOL 型数组 b_Sort 的前两项值为 0 和 1 时(即有两个排序字段 population 和 area，这两个排序字段的顺序分别为降序和升序)，通过 MakeStr 函数可得到一个排序子式：“population DESC，area”，并将这个排序子式记录在成员变量 m_strSort 中。为了完成上述功能，在实现文件 sortdlg.cpp 中，加入 MakeStr 函数的实现代码：

```
void CSortDlg::MakeStr()
{
    m_strSort.Empty();                  //清空字符串 m_strSort 的内容
    for(int i=0;i<m_NumbSort;i++)       //对所有的排序字段循环
    {
```

```
    if(i > 0)                    //如果不是第一个排序字段,在字符串 m_strSort 增加一个逗号
        m_strSort += ",";
    strSort[i].TrimLeft();       //去掉字符串前的空格
    strSort[i].TrimRight();      //去掉字符串后的空格
    m_strSort += strSort[i];     //增加排序字段名称到字符串 m_strSort
    if(! b_Sort[i])              //如果是降序字段
        m_strSort += " DESC";    //在字符串 m_strSort 中增加字符串" DESC"
}
SetDlgItemText(IDC_EDIT1,m_strSort);   //将排序子式填写到编辑框 IDC_EDIT1
}
```

MakeStr 函数的实现过程很简单,对 m_NumbSort 个排序字段循环,将各个排序字段用逗号连接成一个字符串,如果排序字段的排序顺序是降序,则在字段名称后加上字符串" DESC"(对于升序字段,也可以在其后加上字符串" ASC")。

(2) BreakSortStr 函数

BreakSortStr 函数实现的功能是:将一个排序子式分解成各个排序字段和排序顺序的方式,由 CString 对象数组 strSort 和 BOOL 型数组 b_Sort 来记录,是 MakeStr 函数的逆操作函数。例如,对于一个排序子式"population DESC, area",经 BreakSortStr 函数分解,得到结果如下:m_NumbSort = 2,strSort 对象数组的前两项内容为"population"和"Area",b_Sort 数组的前两项值为 0 和 1 时。为了实现上述操作功能,在实现文件 sortdlg.cpp 中,加入 BreakSortStr 函数的实现代码:

```
int CSortDlg::BreakSortStr(CString m_str)
{
    int nChar;
    m_str.TrimLeft();                  //去掉字符串前的空格
    m_str.TrimRight();                 //去掉字符串后的空格
    nChar = m_str.GetLength();         //得到字符串的长度
    CString m_str1 = m_str;
    CString m_str2;
    int nStart;
    nStart = 0;                        //开始查找逗号的位置为 0
    m_NumbSort = 0;                    //排序字段的数目为 0
    int n;
    do
    {
        n = m_str1.Find(",",nStart);   //在 m_str1 字符串中从第 nStart 字符开始,查找逗号
        if(n > nStart)                 //如果找到了逗号
        {
            //将字符串 m_str1 中查找逗号的开始位置至找到的逗号间的字符串拷贝到 m_str2 中
            m_str2 = m_str.Mid(nStart,n - nStart);
            m_str2.TrimLeft();                    //去掉字符串前的空格
```

```
            m_str2.TrimRight();                    //去掉字符串后的空格
            if(m_str2.GetLength()>0)               //如果字符串 m_str2 的长度大于 0,将字
                                                   //符串
                strSort[m_NumbSort++]=m_str2;      //m_str2 记录在 CString 对象数组 strSort 中
            m_str2.Empty();                        //清空 m_str2 字符串
            nStart=n+1;   //在 m_str1 中,把找到的逗号后的位置作为下次查找逗号的开始位置
        }
        else                  //如果没有在字符串中找到逗号(从 nStart 个字符开始查找)
        {
            if(nStart<nChar)   //如果从 nStart 开始到字符串结束还有一段字符串
            {
                //以下将 m_str1 中从第 nStart 位置向后的字符串拷贝到字符串 m_str2 中
                m_str2=m_str.Mid(nStart,nChar);
                m_str2.TrimLeft();                 //去掉字符串前的空格
                m_str2.TrimRight();                //去掉字符串后的空格
                if(m_str2.GetLength()>0)           //字符串的长度>0 时,将 m_str2 记录
                    strSort[m_NumbSort++]=m_str2; //在数组 strSort 中
                m_str2.Empty();                    //清空字符串 m_str2
            }
        }
    }while(n>=0);                          //直到找不到逗号,退出循环

    for(int i=0;i<m_NumbSort;i++)          //对分解得到的所有字符串循环
    {
        m_str2=strSort[i];                 //将第 i 个字符串拷贝到 m_str2 中
        int nn=m_str2.GetLength();         //得到字符串 m_str2 的长度
        n=m_str2.Find(" ");                //在字符串 m_str2 中寻找空格
        if(n>0)                            //如果找到了空格
        {
            strSort[i]=m_str2.Mid(0,n);    //将空格前的字符串记录在 strSort 数组对应项目下
            m_str1=m_str2.Right(nn-n-1);   //将空格后的字符串拷贝到字符串 m_str1
            m_str1.TrimLeft();             //去掉字符串 m_str1 前面的空格
            m_str1.MakeUpper();            //去掉字符串 m_str1 后面的空格
            if(!m_str1.Compare("ASC"))     //如果字符串 m_str1 是"ASC"
                b_Sort[i]=1;               //排序字段的排列顺序是升序
            else if(!m_str1.Compare("DESC"))  //如果字符串 m_str1 是"DESC"
                b_Sort[i]=0;               //排序字段的排列顺序是降序
            else                           //否则,排序字段的顺序是升序
                b_Sort[i]=1;
        }
        else                               //如果在字符串 m_str2 中没有找到空格
            b_Sort[i]=1;                   //排序字段的顺序是升序
    }
```

```
    retu rn m_NumbSort;                    //返回排序字段的数目
}
```

BreakSortStr 函数的主要操作步骤如下：

① 将参数传入的字符串拷贝到字符串 m_str1 中，去掉字符串 m_str1 前后的空格。

② 以 逗号(",")将字符串 m_str1 分解成 m_NumbSort 个字符串，分解后的字符串记录在 CString 对象数组 strSort 中。

③ 对分解后记录在 CString 对象数组 strSort 中的 m_NumbSort 个字符串进行循环，去掉字符串前后的空格，利用一个空字符" "将字符串分为两部分：前一部分作为字段名称记录在 strSort 对象数组中，后一部分字符串 m_str1 与字符串"DESC"和"ASC"比较，如果是字符串"DESC"，则这个排序字段的排序顺序是降序，将 b_sort 数组的相应项设置为 0；如果不是字符串"DESC"(是字符串"ASC"、为空或者其他)，则这个排序字段的排序顺序是升序，将 b_sort 数组相应项设置为 1。

④ 返回分解得到的排序字段的数目。

17.2.4　增加、插入、修改、删除排序字段

本小节中，在对话框类 CSortDlg 中讨论实现增加、插入、修改、删除排序字段的功能。为了实现这些功能，分别为对话框中的按钮"增加"、"插入"、"排序"、"删除"增加 BN_CLICKED 控制消息处理函数。另外，为了在列表框 IDC_LIST1 中选择一个排序字段时，能够利用这个选项所代表的排序字段和排序顺序，初始化列表选择框 IDC_COMBO1 和检查框 IDC_CHICK1，给列表框 IDC_LIST1 增加一个 SEL_CHANGE 控制消息处理函数。增加各个消息处理函数后，在对话框类 CSortDlg 中，增加了各个消息处理函数的定义代码：

```
//{{AFX_MSG(CSortDlg)
virtual BOOL OnInitDialog();
afx_msg void OnSortAdd();
afx_msg void OnSortDelete();
afx_msg void OnSortInsert();
afx_msg void OnSortModify();
afx_msg void OnSelchangeList1();
//}}AFX_MSG
```

在实现文件 sortdlg.cpp 中，增加了消息映射机制：

```
BEGIN_MESSAGE_MAP(CSortDlg, CDialog)
    //{{AFX_MSG_MAP(CSortDlg)
    ON_BN_CLICKED(IDC_SORT_ADD, OnSortAdd)
    ON_BN_CLICKED(IDC_SORT_DELETE, OnSortDelete)
    ON_BN_CLICKED(IDC_SORT_INSERT, OnSortInsert)
    ON_BN_CLICKED(IDC_SORT_MODIFY, OnSortModify)
    ON_LBN_SELCHANGE(IDC_LIST1, OnSelchangeList1)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 sortdlg.cpp 中,增加了消息处理函数空的实现代码。以下具体讨论和实现各个消息处理函数的实现代码,完成需要的操作功能。

(1) OnSortAdd 函数

OnSortAdd 函数是对话框中的按钮控制"增加"(ID 号为 ID_SORT_ADD)的 BN_CLICKED 消息处理函数。其主要功能是:通过在列表选择框 IDC_COMBO1 中选择的字段名称和在检查框 IDC_CHECK1 中设置的排序顺序,在现有的各个排序字段后增加一个新的排序字段。OnSrotAdd 函数的实现代码如下:

```
void CSortDlg::OnSortAdd()
{
    if(m_NumbSort >= 10)            //如果排序字段的数目已经有 10 个,则不能再增加排序字段
    {
        AfxMessageBox("最多只能有 10 个排序字段");
        return;
    }
    int ncom1 = pcom1 -> GetCurSel();   //得到在列表选择框 IDC_COMBO1 中选项的 0 基序号
    if(ncom1 < 0)              //如果没有在列表选择框 IDC_COMBO1 中选择字段名称,函数退出
    {
        AfxMessageBox("没有选择排序字段");
        return;
    }
    strSort[m_NumbSort] = pFieldInfo[ncom1].m_strName;        //得到排序字段的名称
    CButton * pcheck1 = (CButton * )GetDlgItem(IDC_CHECK1); //得到指向检查框 IDC_CHECK1 的
                                                            //指针
    b_Sort[m_NumbSort] = 1 - pcheck1 -> GetCheck();          //根据检查框的状态得到排序顺序
    CString m_str;
    if(b_Sort[m_NumbSort])              //如果是升序排序
        m_str = "↑ ";                   //在字符串中增加一个上箭头
    else                                //如果是降序排序
        m_str = "↓ ";                   //在字符串中增加一个下箭头
    m_str += strSort[m_NumbSort];       //将字段名称加在字符串后
    plist1 -> AddString(m_str);         //在列表框 IDC_LIST1 中增加排序字段
    plist1 -> SetCurSel(m_NumbSort);    //在列表框 IDC_LIST1 中选择刚增加的项目
    m_NumbSort++;                       //排序字段的数目加 1
    MakeStr();                          //调用 MakeStr 函数计算得到排序子式
}
```

OnSortAdd 函数的主要操作步骤如下:

① 检查当前的排序字段数目是否已经达到最大数目 10,如果达到了最大数目,不能再增加排序字段,函数退出。

② 得到存储字段名称的列表选择框 IDC_COMBO1 中选项的 0 基序号,如果没有在这个

列表选择框中选择一个字段名称，函数退出。

③ 通过在列表选择框 IDC_COMBO1 中选项的序号和检查框 IDC_CHECK1 的状态，得到要增加排序字段的名称和排序顺序，将字段名称和排序顺序分别记录在 CString 对象数组 strSort 和 BOOL 型数组 b_sort 中。

④ 用排序顺序和字段名称产生一个字符串 m_str，将这个字符串增加到列表框 IDC_LIST1 中，并在列表框中选择这个新加入的项目。

⑤ 调用 MakeStr 函数产生排序子式，并将产生的排序子式填写到编辑框 IDC_EDIT1 中。

(2) OnSortInsert 函数

OnSortInsert 函数是对话框中的按钮控制“插入”(ID 号为 ID_SORT_INSERT)的 BN_CLICKED 消息处理函数。它实现的功能与 OnSortAdd 函数类似，不同的是，OnSortInsert 函数不是在已有的排序字段后再增加一个排序字段，而是在某个已有的排序字段前插入一个新的排序字段。OnSortInsert 函数的实现代码如下：

```
void CSortDlg::OnSortInsert()
{
    if(m_NumbSort >= 10)              //如果排序字段的数目已经有 10 个，则不能再增加排序字段
    {
        AfxMessageBox("最多只能有 10 个排序字段");
        return;
    }
    int nlist1 = plist1->GetCurSel();   //得到在列表框 IDC_LIST1 中的 0 基序号
    if(nlist1 < 0)        //如果没有在列表框 IDC_LIST1 中选择一个已经存在的排序字段，函数退出
    {
        AfxMessageBox("没有选择要插入排序字段的位置");
        return;
    }
    int ncom1 = pcom1->GetCurSel();        //得到在列表选择框 IDC_COMBO1 中选项的 0 基序号
    if(ncom1 < 0)                 //如果没有在列表选择框 IDC_COMBO1 中选择字段名称，函数退出
    {
        AfxMessageBox("没有选择排序字段");
        return;
    }
    //以下对数组 strSort 和 b_Sort 进行滚动，在 nlist1 位置插入一个数组项
    for(int i = m_NumbSort;i > nlist1;i--)
    {
        strSort[i] = strSort[i-1];
        b_Sort[i] = b_Sort[i-1];
    }
    strSort[nlist1] = pFieldInfo[ncom1].m_strName;                    //得到并记录排序字段的名称
    CButton *pcheck1 = (CButton *)GetDlgItem(IDC_CHECK1);  //得到指向检查框 IDC_CHECK1
                                                                      //的指针
```

```
    b_Sort[nlist1] = 1 - pcheck1 - > GetCheck();        //根据检查框的状态得到并记录排序顺序
    CString m_str;
    if(b_Sort[nlist1])                                  //如果是升序排序
        m_str = "↑ ";                                   //得到一个包含上箭头的字符串
    else                                                //如果是降序排序
        m_str = "↓ ";                                   //得到一个包含下箭头的字符串
    m_str + = strSort[nlist1];                          //将字段名称加在字符串后
    plist1 - > InsertString(nlist1,m_str);              //在列表框 IDC_LIST1 第 nlist1 个项前插入一个项目
    plist1 - > SetCurSel(nlist1);                       //在列表框 IDC_LSIT1 中选择第 nlist1 个项
    m_NumbSort ++ ;                                     //排序字段的数目加 1
    MakeStr();                                          //调用 MakeStr 函数,计算得到排序子式
}
```

OnSortInsert 函数的操作步骤与 OnSortAdd 函数类似,主要操作步骤如下:

① 检查已有的排序字段数目是否已经达到最大数目 10,如果达到最大数目,不能再增加排序字段,函数退出。

② 在列出既有排序字段的列表框 IDC_LIST1 中,得到当前选项的 0 基序号 nlist1,新增的排序字段就是要插入到这个位置。如果在列表框 IDC_LIST1 中没有选项,因为没有确定插入排序字段的位置,函数退出。

③ 得到列表选择框 IDC_COMBO1 中选项的 0 基序号,如果没有在这个列表选择框中选择要插入排序字段的名称,函数退出。

④ 将 CString 对象数组 strSort 和 BOOL 型数组 b_sort 进行滚动,腾出 nlist1 项来记录要插入的排序字段。

⑤ 通过在列表选择框 IDC_COMBO1 中选项的序号和检查框 IDC_CHECK1 的状态,得到要插入排序字段的名称和排序顺序,将字段名称和排序顺序记录在 CString 对象数组 strSort 和 BOOL 型数组 b_sort 中。

⑥ 用排序顺序和字段名称产生一个字符串 m_str,将这个字符串插入到列表框 IDC_LIST1 中 nlist1 项目的前面。

⑦ 调用 MakeStr 函数产生排序子式,并将排序子式填写到编辑框 IDC_EDIT1 中。

(3) OnSortModify 函数

OnSortModify 函数是对话框中的按钮控制"修改"(ID 号为 ID_SORT_MODIFY)的 BN_CLICKED 消息处理函数。其主要功能是:通过在列表选择框 IDC_COMBO1 中选择的字段名称和在检查框 IDC_CHECK1 中设置的排序顺序得到一个排序字段,用这个排序字段替换当前已经存在的一个排序字段。OnSrotModify 函数的实现代码如下:

```
void CSortDlg::OnSortModify()
{
    int nlist1 = plist1 - > GetCurSel();        //得到在列表框 IDC_LIST1 中的 0 基序号
    if(nlist1 < 0)      //如果没有在列表框 IDC_LIST1 中选择一个已经存在的排序字段,函数退出
    {
```

```
    AfxMessageBox("没有选择要修改的排序字段");
    return;
}
int ncom1 = pcom1 - > GetCurSel();      //得到在列表选择框 IDC_COMBO1 中选项的 0 基序号
if(ncom1 < 0)                  //如果没有在列表选择框 IDC_COMBO1 中选择字段名称,函数退出
{
    AfxMessageBox("没有选择排序字段");
    return;
}
strSort[nlist1] = pFieldInfo[ncom1].m_strName;              //得到并记录排序字段的名称
CButton * pcheck1 = (CButton * )GetDlgItem(IDC_CHECK1); //得到指向检查框 IDC_CHECK1 的
                                                          //指针
b_Sort[nlist1] = 1 - pcheck1 - > GetCheck();          //根据检查框的状态得到并记录排序顺序
CString m_str;
if(b_Sort[nlist1])                    //如果是升序排序
    m_str = "↑ ";                     //得到一个包含上箭头的字符串
else                                  //如果是降序排序
    m_str = "↓ ";                     //得到一个包含下箭头的字符串
m_str + = strSort[nlist1];            //将字段名称加在字符串后
plist1 - > DeleteString(nlist1);      //在列表框 IDC_LIST1 中删除第 nlist1 个项目
if(nlist1 = = m_NumbSort - 1)         //如果删除的是列表框 IDC_LIST1 中的最后一个选项
    plist1 - > AddString(m_str);      //在列表框 IDC_LIST1 增加刚产生的排序字段字符串
else                                  //如果删除的不是列表框 IDC_LIST1 中的最后一个选项
    plist1 - > InsertString(nlist1,m_str); //在第 nlist1 个项目前插入刚产生的排序字段
plist1 - > SetCurSel(nlist1);         //在列表框 IDC_LIST1 中选择第 nlist1 选项
MakeStr();                            //调用 MakeStr 函数,计算得到排序子式
}
```

OnSortModify 函数的主要操作步骤如下：

① 在列出既有排序字段的列表框 IDC_LIST1 中,得到当前选项的 0 基序号 nlist1,也就是需要对这个序号的排序字段进行修改,如果在列表框 IDC_LIST1 中没有选择要修改的排序字段,函数退出。

② 得到列出字段名称的列表选择框 IDC_COMBO1 中选项的 0 基序号,如果没有在这个列表选择框中选择排序字段名称,函数退出。

③ 通过在列表选择框 IDC_COMBO1 中选项的序号和检查框 IDC_CHECK1 的状态,得到要修改成的排序字段的名称和排序顺序,将字段名称和排序顺序记录在 CString 对象数组 strSort 和 BOOL 型数组 b_sort 中。

④ 用排序顺序和排序字段名称产生一个字符串 m_str,用这个字符串来修改列表框 IDC_LIST1 中的当前选项。修改列表框选项采用从列表框中删除这个选项,然后再增加或插入项目的方法。

⑤ 调用 MakeStr 函数产生排序子式,并将产生的排序子式填写到编辑框 IDC_EDIT1 中。

(4) OnSortDelete 函数

OnSortDelete 函数是对话框中按钮控制“删除”(ID 号为 ID _ SORT _ DELETE)的 BN _ CLICKED 消息处理函数,完成从排序子式中删除一个既有排序字段的功能。以下是 OnSortDelete 函数的实现代码:

```
void CSortDlg::OnSortDelete()
{
    int nlist1 = plist1 -> GetCurSel(); //得到在列表框 IDC _ LIST1 中的 0 基序号
    if(nlist1 < 0)                      //如果没有在列表框 IDC _ LIST1 中选择一个排序字段,函数退出
    {
        AfxMessageBox("没有选择要删除的排序字段");
        return;
    }
    //以下对数组 strSort 和 b _ Sort 滚动,删除数组的第 nlist1 项
    for(int i = nlist1;i < m _ NumbSort - 1;i++)
    {
        strSort[i] = strSort[i + 1];
        b _ Sort[i] = b _ Sort[i + 1];
    }
    plist1 -> DeleteString(nlist1);        //从列表框 IDC _ LIST1 中删除当前选择的排序字段项
    if(nlist1 = = m _ NumbSort - 1)        //如果删除的是列表框 IDC _ LIST1 中的最后一项
        plist1 -> SetCurSel(nlist1 - 1);   //选择列表框 IDC _ LIST1 的上一个项目
    else                                   //如果删除的不是列表框 IDC _ LIST1 中的最后一项
        plist1 -> SetCurSel(nlist1);       //选择列表框 IDC _ LIST1 中同样序号的项目
    m _ NumbSort - -;                      //排序字段的数目减 1
    //以下调用 OnSelchangeList1 函数,分解在列表框 IDC _ LIST1 中选择的排序字段,并初始化列表
    //选择框 IDC _ COMBO1 和检查框 IDC _ CHECK1 的状态
    OnSelchangeList1();
    MakeStr();                             //调用 MakeStr 函数,计算得到排序子式
}
```

OnSortDelete 函数的主要操作步骤如下:

① 在列出既有排序字段的列表框 IDC _ LIST1 中得到当前选项的 0 基序号 nlist1,也就是选择需要删除的排序字段,如果在 IDC _ LIST1 中没有选择要删除的排序字段,函数退出。

② 将 CString 对象数组 strSort 和 BOOL 型数组 b _ sort 进行滚动,删除数组中下标是 nlist1 的项,排序字段的数目减 1。

③ 在列表框 IDC _ LIST1 中删除选择的项目,并调整列表框中的选项,仍然选择第 nlist1 项(如果删除的是列表框的最后一个项目,则选择第 nlist1 - 1 项)。

④ 调用 OnSelchangeList1 函数,利用在列表框 IDC _ LIST1 中选择的排序字段,初始化列表选择框 IDC _ COMBO1 和检查框 IDC _ CHECK1 的状态。

⑤ 调用 MakeStr 函数产生排序子式,并将产生的排序子式填写在编辑框 IDC _ EDIT1 中。

(5) OnSelchangeList1 函数

OnSelchangeList1 函数是列表框 IDC _ LIST1 的 SEL _ CHANGE 控制消息处理函数，当在列表框 IDC _ LIST1 中选择一个选项时，这个函数被执行。OnSelchangeList1 函数根据在列表框 IDC _ LIST1 中选择的排序字段的名称和排序顺序，来初始化列表选择框 IDC _ COMBO1 和检查框 IDC _ CHECK1 的状态。以下是 OnSelchangeList1 函数的实现代码：

```
void CSortDlg::OnSelchangeList1()
{
    int nlist1 = plist1 -> GetCurSel(); //得到在列表框 IDC _ LIST1 中的 0 基序号
    if(nlist1 < 0)                      //如果没有在列表框 IDC _ LIST1 中选择一个排序字段，函数退出
    {
        AfxMessageBox("没有选择要删除的排序字段");
        return;
    }
    CString m _ str1 = strSort[nlist1];          //得到要选择的排序字段名称
    m _ str1.TrimLeft();                         //去掉字符串前面的空格
    m _ str1.TrimRight();                        //去掉字符串后面的空格
    BOOL yn = 0;
    for(int i = 0;i < m _ NumbField;i++)         //对记录集合的所有字段循环
    {
        if(! m _ str1.Compare(pFieldInfo[i].m _ strName))
        //如果当前选择的排序字段的名称等于记录集合对象中第 i 个字段的名称
        {
            yn = 1;                              //标识找到了字段名称
            break;
        }
    }
    if(yn)                                       //如果选择的排序字段是记录集合对象的一个字段
        pcom1 -> SetCurSel(i);                   //在列表选择框 IDC _ COMBO1 中选择这个字段
    else                                         //如果选择的排序字段不是记录集合的一个字段
        pcom1 -> SetCurSel(-1);                  //不在列表选择框 IDC _ COMBO1 中设置选项
    CButton * pcheck1 = (CButton * )GetDlgItem(IDC _ CHECK1); //得到指向检查框 IDC _ CHECK1 的
                                                              //指针
    pcheck1 -> SetCheck(1 - b _ Sort[n]);        //根据排序顺序设置检查框 IDC _ CHECK1 的状态
}
```

OnSelchangeList1 函数的主要操作步骤如下：

① 在列出既有排序字段的列表框 IDC _ LIST1 中，得到当前选项的 0 基序号 nlist1，如果在列表框 IDC _ LIST1 中没有选择一个选项，函数退出。

② 将排序字段的名称与记录集合对象中的所有字段名称进行比较，得到排序字段属于记录集合对象的第几个字段。在列表选择框 IDC _ COMBO1 中选择这个字段对应的项目。

③ 根据排序字段的排序顺序设置检查框 IDC _ CHECK1 的状态。如果字段是升序排序，

去掉检查框的选择标志,否则,在检查框中设置选择标志("√")。

17.2.5 进行排序

在对话框中选择和组织了排序子式后,运行对话框的"排序"按钮,就会退出对话框并进行排序操作。为了检查排序子式的有效性,在退出对话框前进行排序操作,操作成功则退出对话框,不成功则继续在对话框中组织和修改排序子式。为了实现这个功能,在对话框类 CSortDlg 中增加"排序"按钮控制(ID 号 IDOK)的消息处理函数,因为对于 IDOK 消息已经在 CDialog 中定义了消息处理函数 OnOK,所以实际上是在当前对话框类中重载了 CDialog 类的 OnOk 函数。操作完毕,在对话框类 CSortDlg 中增加了虚函数的重载定义:

```
//{{AFX_MSG(CSortDlg)
virtual BOOL OnInitDialog();
afx_msg void OnSortAdd();
afx_msg void OnSortDelete();
afx_msg void OnSortInsert();
afx_msg void OnSortModify();
afx_msg void OnSelchangeList1();
virtual void OnOK();
//}}AFX_MSG
```

在实现文件 sortdlg.cpp 中,增加了 OnOK 函数的实现代码。以下是经过修改后的函数实现代码:

```
void CSortDlg::OnOK()
{
    if(m_pSet->IsOpen())                    //如果打开了记录集合对象,则把它关闭
        m_pSet->Close();
    CString m_str=m_pSet->m_strSort;  //存储记录集合对象原来的排序子式
    m_pSet->m_strSort=m_strSort;        //将记录集合的排序子式设成在对话框中组织的排序子式
    if(m_pSet->Open())
                                //如果打开记录集合对象成功,调用 CDialog::OnOK 函数退出对话框
        CDialog::OnOK();
    else                                    //如果打开记录集合对象不成功,即排序子式可能包含错误
        m_pSet->m_strSort=m_str;      //恢复记录集合对象原来的排序子式
}
```

OnOK 函数的主要操作步骤如下:

① 将记录集合对象的排序子式设置成对话框中组织完成的排序子式。在改变记录集合对象的排序子式前,利用 m_str 存储记录集合对象原有的排序子式。

② 关闭并重新打开记录集合对象。如果操作成功,则表明排序子式正确,调用 CDialog 类的 OnOK 函数退出对话框。

③ 如果打开记录集合对象操作不成功,则表明排序子式可能存在错误(一般情况下有出错提示信息),恢复记录集合对象原来的排序子式,不退出对话框,继续在对话框中组织排序

子式。

至此，在对话框类 CSortDlg 中完成了组织过滤子式操作的各种功能。在 17.3 节，我们将讨论在应用程序 Draw 的数据库浏览视图(CDatabaseView)中，通过调用这个对话框类组织排序子式，实现对记录集合的排序功能。

17.3　实现数据库排序功能

在 16.5.1 小节，在视图类 CDatabaseView 使用的菜单资源 IDR _ DRAWDATA 中，增加了一个菜单项“排序操作...”，并为这个菜单项建立了一个消息处理函数 OnRecordSort。本节我们对消息处理函数 OnRecordSort 进行修改，利用对话框类 CSortDlg 来组织排序子式，完成对记录集合的排序功能。以下是修改后的 OnRecordSort 函数实现代码：

```
void CDatabaseView::OnRecordSort()
{
    CSortDlg pDg1(m_pSet1);              //定义一个对话框类 CSortDlg 对象
    if(pDg1.DoModal() == IDOK)           //调用对话框组织排序子式，如果按“排序”按钮退出对话框
    {
        if(m_pSet1->IsOpen())            //如果打开了记录集合对象，将它关闭
            m_pSet1->Close();
        //以下将记录集合对象的过滤子式设置成在对话框中组织的排序子式
        m_pSet1->m_strSort = pDg1.m_strSort;
        m_pSet1->Open();                 //打开记录集合对象
        Invalidate();                    //重画视图
    }
}
```

在 OnRecordSort 函数中，首先定义一个对话框类 CSortDlg 对象 pDg1，参数 m _ pSet1 是指向浏览视图当前浏览的记录集合对象的指针。启动对话框，在对话框界面中组织排序子式，当按“排序”按钮退出对话框时，将指针 m _ pSet1 指向的记录集合对象的过滤子式设置成在对话框中组织的排序子式，打开记录集合对象完成排序操作并重画视图。

因为进行排序操作时，记录集合包含的记录数不发生变化，所以在排序操作后，不需要调用函数 Init2 重新计算浏览视图滚动区域的纵向参数。

对应用程序 Draw 编译后运行，在 CDatabaseView 浏览视图中，运行“过滤及排序”菜单下的“排序操作...”菜单项时，就会进入到排序子式编辑对话框，如果记录集合对象存在过滤子式，则会在对话框界面上交互列出分解后的排序字段。在对话框界面上交互编辑排序子式，当运行按钮“排序”退出对话框时，在重画后的数据库浏览视图中，可以看到各个记录按照在对话框中组织的排序子式进行了排序。

第 18 章　数据库管理功能通用化

本篇的第 16 章和第 17 章，在第二篇开发的矢量图形系统 Draw 基础上，创建了一个数据库浏览视图，在视图中实现了对记录集合的编辑功能（增加记录、修改记录、删除记录等），以及通用查询和排序功能。但是，所有这些功能都是针对记录集合类 CData1Set 对象建立起来的，不能在应用程序 Draw 中对其他记录集合对象完成各种操作功能，这就限制了系统的适用范围。

C++ 类提供的虚函数和多态性为解决这个问题提供了极大的方便性，本章将利用类的继承和通过虚函数实现的多态性，使应用程序 Draw 中的大多数数据库管理功能通用化。

18.1　各种数据库管理功能的可通用化分析

本篇在应用程序 Draw 中所实现的数据库管理功能中，很多功能模块是相对独立的，这些模块经过一定处理后可以形成完全独立的功能模块，供所有的记录集合对象通用；而另外的一些功能模块，是针对具体的记录集合对象创建的，与记录集合对象的具体内容直接相关，这些功能模块就不容易被通用化。本节我们对在应用程序 Draw 中实现的各种数据库管理功能模块的可通用化进行分析。

18.1.1　可通用化的数据库管理功能

在应用程序 Draw 中实现的数据库管理功能模块中，有很多是相对独立的，可以对其进行通用化。下面对这些模块进行分析。

（1）数据库浏览功能

在视图类 CDatabaseView 中实现的记录集合浏览功能，只与指向 CData1Set 类对象的指针 m_pSet1有关，只要获得了指向记录集合对象的指针，就可以通过这个指针得到记录集合中的字段信息及记录数目，通过这些信息确定视图滚动区域的范围。在视图类 CDatabaseView 中的绘制函数（OnDraw 函数）中，是直接从指针 m_pSet1 指向的记录集合对象的当前记录中读出各个字段的内容并绘制到视图中，而没有采用记录集合类 CData1Set 的成员变量。也就是说，在数据库浏览功能中，通过指针 m_pSet1 所使用的成员变量，除了一些虚函数之外，都属于 CRecordSet 类而不属于具体的记录集合类 CData1Set。所以，只要能够对记录集合类指针 m_pSet1 进行抽象化，使它能够指向任意记录集合对象，并通过多态性去执行具体记录集合对象的虚函数，就能够使数据库浏览功能完全通用化，使它能够适用于任何记录集合对象。

（2）记录删除功能

在应用程序 Draw 中实现的删除记录功能（包含删除记录和删除所有记录两项功能），与数

据库浏览功能类似，这些操作功能只与记录集合类指针 m_pSet1 有关，通过 m_pSet1 所调用的成员函数也是 CRecordSet 类的成员函数而不是 CData1Set 类的成员函数。所以，只要对指针 m_pSet1 进行抽象化，使它能够指向任意记录集合对象，就能够对记录删除功能通用化。

(3) 通用过滤功能

从第 15 章通用过滤功能的具体实现过程可以看出，在对话框类 CFilterDlg 中，通过指向具体记录集合对象的指针 m_pSet 调用记录集合类的各种功能，而所调用的这些功能与具体的记录集合对象没有关系。所以，只要对指针 m_pSet 抽象化，使它能够指向任意记录集合对象，就能够对通用过滤功能通用化。

(4) 通用排序功能

与通用过滤功能相同，在对话框类 CSortDlg 中实现的通用排序功能，也与具体的记录集合对象没有关系，只要对对话框类 CSortDlg 中的指针 m_pSet 进行抽象化，就能够对排序功能完全通用化，实现针对任何记录集合对象的排序操作。

在这一小节的讨论中，所谓各种功能模块与具体的记录集合对象没有关系，指的是在这些功能实现过程中，对记录集合对象所进行的操作功能都是从 CRecordSet 类中继承下来的标准功能，而与从 CRecordSet 类派生的具体记录集合类(如 CData1Set)没有关系。虽然在这些操作过程中也调用了具体记录集合类的一些函数(如 GetDefaultSQL 等)，但这些函数都是虚函数，可以通过多态性来抽象化。所以，在数据库应用程序设计中，通过抽象定义一个记录集合基类，具体的记录集合类从这个记录集合基类派生，定义一个记录集合基类指针，将这个指针指向要操作的记录集合对象，通过这个指针来调用记录集合类的函数实现功能模块时，可以通过多态性去分别执行各个记录集合类中的虚函数重载，去具体实现针对各个记录集合对象的操作功能，从而实现了操作功能模块的通用化。在本章 18.2 节，将具体讨论各个功能模块通用化的方法和步骤。

18.1.2　不易通用化的功能

在应用程序 Draw 中，还有一些数据库管理功能模块，是与记录集合对象相关的。在数据库浏览视图 CDatabaseView 中，进行增加记录和修改记录操作时，利用 CData1Dlg 对话框来组织对一条记录进行编辑修改的对话框界面，在对话框类 CData1Dlg 中，使用了具体的记录集合对象的成员变量。也就是说，因为不同记录集合对象所包含的字段和类型各不相同，所以对于不同的记录集合对象，必须组织不同的交互操作界面来编辑记录集合的一条记录，对于这些操作功能，不便通过抽象化来实现通用化，只能针对不同的记录集合对象来具体组织记录编辑界面。

本节对在应用程序 Draw 中与数据库管理有关的功能模块进行了分析，确定了哪些功能模块可以通过抽象化实现通用化，哪些功能模块是与具体的记录集合对象密切相关的，对于不同的记录集合对象必须进行不同的操作。在本章 18.2 节中，将对能够通用化的功能模块进行抽象化处理，实现这些功能模块与记录集合对象无关性即通用化。

18.2 实现数据库管理功能通用化

本节介绍对这些功能模块进行通用化的方法和步骤。

18.2.1 通用化功能模块的方法

在应用程序 Draw 中,通过抽象建立一个记录集合基类来通用化各操作功能模块。这一小节讨论具体实现方法。

(1) 建立记录集合基类

在数据库应用程序 Draw 中,创建一个记录集合类作为记录集合基类,应用程序中的具体记录集合类由这个基类派生。

例如,在应用程序 Draw 中,用 ClassWizard 创建一个 CRecordSet 类的派生类 CDataBaseSet(创建一个 CRecordSet 派生类的操作步骤可参考 14.2.2 小节)。在创建这个类时,可任意选择一个 ODBC 数据源和数据表,类的头文件在 dbaseset.h,实现文件在 dbaseset.cpp。因为这个记录集合类不被用来创建记录集合对象,所以可将类中的一些虚函数(如 GetDefaultSQL、GetDefaultConnect、DoFieldChange 等)从类中删除(也可以保持创建后的类不变)。

(2) 从记录集合基类派生具体的记录集合类

在应用程序 Draw 中,将具体的记录集合类由记录集合基类 CDataBaseSet 来派生,而不是由类 CRecordSet 来派生。这时,具体的记录集合类实际上是由类 CRecordSet 来间接派生,记录集合基类 CDataBaseSet 在中间起抽象作用。具体的操作步骤非常简单,按照以下的几个操作步骤,就可以把应用程序中由类 CRecordSet 派生的记录集合类改成由记录集合基类 CDataBaseSet 来派生。我们以应用程序 Draw 中的 CData1Set 类为例进行讨论。

① 在记录集合类 CData1Set 的定义中,将类的基类由类 CRecordSet 改成记录集合基类 CDataBaseSet:

```
class CData1Set : public CDataBaseSet
{
    ......//省略具体的定义代码
}
```

为了在 CData1Set 类定义时能够识别 CDataBaseSet 类,将 CDataBaseSet 类的头文件插入到 CData1Set 类的定义前:

```
# include "dbaseset.h"
```

② 在记录集合类 CData1Set 的构造函数中,将调用 CRecordSet 类的构造函数改成调用 CDataBaseSet 类的构造函数。即将如下代码:

```
CData1Set::CData1Set(CDatabase * pdb)
    : CRecordSet(pdb)
```

```
{
    ......//省略具体的实现代码
}
```

修改为：

```
CData1Set::CData1Set(CDatabase * pdb)
    : CDataBaseSet(pdb)
{
    ......//省略具体的实现代码
}
```

对记录集合类 CData1Set 经过以上两个步骤的改动，就能够保证这个类是由 CDataBaseSet 类而不是直接由 CRecordSet 类来派生。

(3) 用记录集合基类 CDataBaseSet 指针来实现可通用化功能模块

在实现了将具体记录集合类都是由记录集合基类 CDataBaseSet 来派生后，对于数据库应用程序的可通用化的功能模块，利用记录集合基类指针来代替具体的记录集合类指针，来完成对记录集合对象的操作功能，就可以使这些功能模块通用化。在视图类 CDatabaseView 中，定义一个记录集合基类指针：

```
CDataBaseSet * m_pSet;
```

在视图类 CDatabaseView 中的浏览功能、记录删除操作、通用过滤操作、通用排序操作等功能模块中，利用 CDataBaseSet 类指针 m_pSet 来代替原来的 CDdata1Set 类指针 m_pSet1，通过指针 m_pSet 来操作记录集合对象。类指针 m_pSet 在使用前必须被指向具体的记录集合对象。例如，对于以 CDataBaseSet 类为基类创建的记录集合类 CData1Set，在文档类中创建一个记录集合对象：

```
CData1Set m_data1;
```

为完成对这个记录集合对象的操作功能，在视图类 CDatabaseView 的 OnInitialUpdate 函数中，将 CDataBaseSet 类指针 m_pSet 指向这个记录集合对象：

```
void CDatabaseView::OnInitialUpdate()
{
    CDrawDoc * pDoc = GetDocument();
    m_pSet1 = &pDoc->m_data1;
    m_pSet = &pDoc->m_data1;
    Init1();
    Init2();
    CScrollView::OnInitialUpdate();
    SetScrollSizes(MM_TEXT, sizeTotal, sizePage, sizeLine);
}
```

对于以记录集合基类 CDataBaseSet 为基类创建的其他记录集合类(如 CData2Set)，在文档

类中创建相应的记录集合对象：

```
CData2Set m_data2;
```

为了对记录集合对象 m_data2 进行管理，需要在视图类 CDatabaseView 的 OnInitialUpdate 函数中，将 CDataBaseSet 类指针 m_pSet 指向这个记录集合对象：

```
void CDatabaseView::OnInitialUpdate()
{
    CDrawDoc * pDoc = GetDocument();
    m_pSet1 = &pDoc->m_data1;
    m_pSet = &pDoc->m_data2;
    Init1();
    Init2();
    CScrollView::OnInitialUpdate();
    SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);
}
```

根据虚函数和由此引发的多态性，CDataBaseSet 类指针 m_pSet 会根据指向的不同派生类对象，而分别去执行派生类中的虚函数重载。例如，当通过 m_pSet 调用函数 GetDefaultSQL 时：

```
m_pSet->GetDefaultSQL();
```

GetDefaultSQL 函数是定义在 CRecordSet 类中的虚函数，在具体的记录集合类中（如 CData1Set、CData2Set 等）有这个虚函数的重载。如果 m_pSet 是指向 m_data1 记录集合对象，则调用的是 CData1Set 类中的虚函数 GetDefaultSQL；而如果 m_pSet 指向的是 m_data2 记录集合对象，则调用的是 CData2Set 类中的虚函数 GetDefaultSQL。这样就通过一个记录集合基类指针 m_pSet，实现了对各个记录集合类的抽象调用。通过这个记录集合基类指针 m_pSet 所完成的各个数据库管理功能，与具体的记录集合对象无关即实现了通用化。

18.2.2 功能模块通用化的步骤

本小节我们将对应用程序 Draw 中实现的一些功能模块进行通用化。以下是具体的操作步骤。

(1) 创建记录集合基类

需要首先在应用程序 Draw 中创建一个记录集合基类，具体的记录集合类都要由这个基类来派生。按照 18.2.1 小节中介绍的步骤，创建一个 CRecordSet 类的派生类 CDataBaseSet 作为记录集合基类，然后将具体的记录集合类 CData1Set 改成由 CDataBaseSet 类派生。

(2) 修改视图类 CDatabaseView

在视图类 CDatabaseView 中，定义一个记录集合基类指针：

```
protected:
```

```
CDataBaseSet * m_pSet;
```

在视图类 CDatabaseView 的实现文件中，将使用指针 m_pSet1 的地方，全部由指针 m_pSet 来代替。只有以下几处例外：

① 在视图类 CDatabaseView 的 OnInitialUpdate 函数中，对 m_pSet1 指针不做修改，另外增加一行代码初始化指针 m_pSet：

```
void CDatabaseView::OnInitialUpdate()
{
    CDrawDoc * pDoc = GetDocument();
    m_pSet1 = &pDoc->m_data1;
    m_pSet = &pDoc->m_data1;
    Init1();
    Init2();
    CScrollView::OnInitialUpdate();
    SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);
}
```

② 对于视图类 CDatabaseView 的成员函数 OnRecordAdd、OnRecordModify，因为在这些函数中进行的一些操作（如调用记录编辑对话框 CData1Dlg 来编辑记录），与具体的记录集合对象有关（使用了具体记录集合类中的成员变量），只能由 CData1Set 类指针 m_pSet1 而不能使用记录集合基类指针 m_pSet（因为在 CDataBaseSet 类中不包含 CData1Set 类中的成员变量）调用完成。所以，在这些成员函数中，不能用 CDataBaseSet 类指针 m_pSet 代替 CData1Set 类指针 m_pSet1。

③ 视图类 CDatabaseView 中的成员函数 GetOnlyID，其实现代码是针对具体的记录集合对象的（利用了指向 CData1Set 类的指针，使用了 CData1Set 类的成员变量 m_only_id）。可以采用以下的两个方法来使这个成员函数适用于所有从 CDataBaseSet 类派生的记录集合类对象：

第一种方法是在记录集合基类 CDataBaseSet 中创建一个成员函数 GetID，利用这个成员函数来得到记录集合当前记录的某个字段（用来标识记录唯一值的字段）的值。将 GetID 函数定义成虚函数：

```
class CDataBaseSet : public CRecordSet
{
    ......//省略以上操作
    public:
        virtual int GetID()=0;
    ......//省略以下代码
}
```

将虚函数 GetID 设置为纯虚函数，一方面保证 CDataBaseSet 类不能直接来创建对象，另一方面，在由 CDataBaseSet 类派生的记录集合类中，必须对这个虚函数进行重载，否则，不能用这个记录集合类来创建对象。在应用程序 Draw 中，必须在 CDataBaseSet 类的派生类 CData1Set 中重载这个虚函数：

```
class CData1Set : public CDataBaseSet
{
```

```
    public:
        virtual int GetID();
    ......//省略以下代码
}
```

在实现文件 dataIset.cpp 中，加入 GetID 函数的实现代码：

```
int DataISet::GetID()
{
    return m_only_id;
}
```

这样，就可以对视图类 CDatabaseView 的成员函数 GetOnlyID 进行修改，将函数中的 CDataISet 类指针 m_pSetI 全部改为 CDataBaseSet 类指针 m_pSet，同时对代码进行改动，加入调用 GetID 函数来得到记录集合当前记录识别号的代码。以下是修改后的 GetOnlyID 函数的实现代码：

```
int CDatabaseView::GetOnlyID()
{
    int * IndexI,id;
    IndexI = GetDocument()->m_Index;              //指向文档对象中定义的数组 m_Index
    CString m_strI = m_pSetI->m_strFilter;        //存储记录集合对象的过滤子式
    m_pSet->m_strFilter="";  //将过滤子式设为空，即产生的记录集合包含数据表的所有记录
    if(m_pSet->IsOpen())                          //如果打开了记录集合对象
        m_pSet->Close();                          //关闭记录集合对象
    m_pSet->Open();                               //打开记录集合对象
    for(int i=0;i<30000;i++)                      //将整数数组 IndexI 的所有项设为 0
        IndexI[i]=0;
    while(!m_pSet->IsEOF())                       //对记录集合的所有记录遍历
    {
        IndexI[m_pSet->GetID()]=1;                //将以记录的字段值为下标的 IndexI 数组项设为 1
        m_pSetI->MoveNext();                      //记录指针指向下一条记录
    }
    //以下得到数组 IndexI 中第一个等于 0 的项，其下标号是与所有记录字段值不同的唯一值
    for(i=0;i<30000;i++)
    {
        if(IndexI[i]==0)
        {
            id=i;
            break;
        }
    }
    m_pSet->Close();                              //关闭记录集合对象
    m_pSet->m_strFilter=m_strI;                   //恢复记录集合对象原来的过滤子式
    m_pSet->Open();                               //重新打开记录集合对象
```

```
    return id;                                    //返回得到的唯一识别号
}
```

以上函数中的黑体代码，利用虚函数 GetID 来得到记录集合当前记录的唯一识别号。只要在由 CDataBaseSet 类派生的记录集合类中，重载 GetID 虚函数来返回记录集合中当前记录的唯一识别号，就可以利用 GetOnlyID 函数，计算这个派生的记录集合类定义的记录集合对象中新增记录的唯一识别号，即 GetOnlyID 函数适用于任何 CDataBaseSet 派生类对象。

第二种方法是假定所有记录集合对象都是利用第一个字段来存储唯一识别号。在这种假定下，通过 CRecordSet 类的 GetValue 函数直接得到第一个字段的值，而不是利用记录集合类的成员变量来得到字段值，从而使函数 GetOnlyID 中的所有操作都与具体的记录集合类无关，可以由 CDataBaseSet 类指针 m_pSet 代替具体记录集合类指针 m_pSet1，将 GetOnlyID 函数通用化。按照这种方式修改 GetOnlyID 函数后，得到 GetOnlyID 函数的实现代码如下：

```
int CDatabaseView::GetOnlyID()
{
    int * Index1,id;
    Index1 = GetDocument() -> m_Index;          //指向文档对象中定义的数组 m_Index
    CString m_strl = m_pSet1 -> m_strFilter;    //记录记录集合对象的过滤子式
    m_pSet -> m_strFilter = "";  //将过滤子式设为空，即产生的记录集合包含数据表的所有记录
    if(m_pSet -> IsOpen())                      //如果打开了记录集合对象
        m_pSet -> Close();                      //关闭记录集合对象
    m_pSet -> Open();                           //打开记录集合对象
    for(int i = 0;i < 30000;i++)                //将整数数组 Index1 的所有项设为 0
        Index1[i] = 0;
    CString c1;
    CString& c2 = c1;
    i = 0;
    while(! m_pSet -> IsEOF())                  //对记录集合的所有记录遍历
    {
        m_pSet -> GetFieldValue(1,c2);          //得到当前记录第 1 个字段的字符串
        Index1[atoi(c1)] = 1;                   //将以记录的字段值为下标的 Index1 数组项设为 1
        m_pSet1 -> MoveNext();                  //记录指针指向下一条记录
    }
    //以下得到数组 Index1 中第一个等于 0 的项，其下标号是与所有记录字段值不同的唯一值
    for(i = 0;i < 30000;i++)
    {
        if(Index1[i] = = 0)
        {
            id = i;
            break;
        }
    }
    m_pSet -> Close();                          //关闭记录集合对象
```

```
    m_pSet->m_strFilter = m_strl;      //恢复记录集合对象原来的过滤子式
    m_pSet->Open();                   //重新打开记录集合对象
    return id;                        //返回得到的唯一识别号
}
```

以上函数的黑体部分代码，通过得到记录集合当前记录第一个字段的字符串内容，将这个字符串转成整数作为这个记录的唯一识别值，然后比较计算得到新增记录的唯一识别值。

(3) 修改对话框类 CFilterDlg

为了在应用程序 Draw 中通用化过滤条件编辑器，供所有的记录集合对象使用，需要对对话框类 CFilterDlg 的头文件和实现文件进行修改。

■ 对于对话框类 CFilterDlg 的头文件，需要进行如下步骤的修改：

① 在对话框类 CFilterDlg 的定义前，将对类的外部引用由代码：

```
class CDatalSet;
```

改成：

```
class CDataBaseSet;
```

② 改变对话框类 CFilterDlg 中定义的记录集合类指针，由具体的记录集合类指针改为记录集合基类指针。即将以下代码：

```
CDatalSet * m_pSet
```

改为：

```
CDataBaseSet * m_pSet;
```

③ 对于对话框类 CFilterDlg 的构造函数，将其参数进行修改，修改后的构造函数定义如下：

```
CFilterDlg(CDataBaseSet * pSet,CWnd * pParent = NULL);
```

■ 对于对话框类 CFilterDlg 的实现文件 filterdlg.cpp，需要进行如下步骤的修改：

① 改变插入的头文件，将以下代码：

```
#include "datalset.h"
```

改为：

```
#include "databaseset.h"
```

② 对应于头文件中对构造函数的修改，需要在实现文件中，修改构造函数的实现代码：

```
CFilterDlg::CFilterDlg(CDataBaseSet * pSet,CWnd * pParent = NULL)
{
    ......//省略实际代码
}
```

(4) 修改对话框类 CSortDlg

对于实现通用排序子式编辑器的对话框类 CSortDlg，需要做的修改与对话框类 CFilterDlg 类似，读者可参考(3)中对对话框类 CFilterDlg 的修改，自己完成对对话框类 CSortDlg 的修改。

经过这一小节所做的修改，就实现了应用程序 Draw 中可通用化功能模块的通用化，使这些功能模块可以适用于任何由记录集合基类 CDataBaseSet 派生的记录集合类的对象，从而可以非常容易地在应用程序 Draw 中扩充应用程序能够管理的记录集合对象。

18.2.3　选择操作不同的记录集合对象

在应用程序 Draw 中，在视图类 CDatabaseView 的 OnInitialUpdate 函数中，将记录集合基类 CDataBaseSet 指针 m_pSet 指向了 CData1Set 类的记录集合对象 m_data1。一般情况下，数据库应用程序中需要建立多个而不是仅仅一个记录集合对象，这就需要在数据库应用程序中，根据需要有选择性的将指针 m_pSet 指向不同的记录集合对象，实现对多个记录集合对象的管理功能。下面介绍实现这个功能的一种方法。

(1) 创建一个管理多个记录集合对象信息的结构

在应用程序 Draw 中，将矢量图形系统和数据库管理系统相结合实现 GIS 系统时，要求每个记录集合对象要有一个唯一的识别号，这个识别号在记录集合对象创建时给定，而且一直保持不变。可考虑定义一个如下的结构来存储一个记录集合对象的信息：

```
typedef struct
{
    short only_ID;      //唯一识别号
    char Name[20];      //记录集合对象的描述名称
} DataInfoStru;
```

only_ID 用来存储一个记录集合对象的唯一识别号，Name 字符串用来存储一个记录集合对象的描述名称。在文档类 CDrawDoc 中定义一个结构数组来存储记录集合对象的信息：

```
public:
    DataInfoStru DataInfo[300];
    short m_NumbData;
```

在文档类 CDrawDoc 中，定义了一个结构数组 DataInfo 用来存储记录集合对象的信息，以上定义的结构数组可以用来存储 300 个记录集合对象(可以定义更大项数的数组或者动态分配数组项数)，m_NumbData 用来存储记录集合对象的数目。对于应用程序 Draw 中在文档类中已经定义的记录集合对象 m_data1，可以在文档类的构造函数中初始化 DataInfo 结构的值：

```
CDrawDoc::CDrawDoc()
{
    ...//省略以上代码
    strcpy(DataInfo[0].Name,"城市信息");
```

```
    DataInfo[0].only_ID=1;
    m_NumbData=1;
}
```

以上的代码表示在文档对象中有一个记录集合对象,这个记录集合对象的唯一识别号为1。

(2) 从文档对象中选择记录集合对象

为了能够从文档对象中有选择性的返回一个记录集合对象的指针,在文档类 CDrawDoc 中定义一个函数:

```
public:
    CDataBaseSet * GetDataSet(int Index);
```

GetDataSet 函数用来返回一个指向记录集合对象的 CDataBaseSet 类指针,参数 Index 是 0 基序号,表示要返回的记录集合对象在 DataInfo 数组中的序列号。对于应用程序 Draw,可以在实现文件 drawdoc.cpp 中,加入以下的 GetDataSet 函数实现代码:

```
CDataBaseSet * CDrawDoc::GetDataSet(int Index)
{
    CDataBaseSet * m_pSet;
    if(Index<0||Index>=m_NumbData)          //如果序号 Index 不在有效范围内,函数返回空
        return NULL;
    short ID=DataInfo[Index].only_ID;       //得到要返回的记录集合对象的唯一识别号
    switch(ID)                  //根据记录集合对象的唯一识别号,选择返回不同的记录集合对象
    {
        case 1:                             //如果记录集合的唯一识别号为 1
            m_pSet=&m_data1;                //得到指向记录集合对象 m_data1 的指针
            break;
        default:                            //否则,指针 m_pSet 指向空
            m_pSet=NULL;
    }
    return m_pSet;                          //返回指向记录集合对象的 CDataBaseSet 类指针
}
```

在 GetDataSet 函数中,通过记录集合对象的 ID 号来决定要返回哪个记录集合对象的指针。在 DataInfo 结构数组中给每个记录集合对象定义了一个唯一识别号 only_ID,对于每个记录集合对象,从始至终能有一个与之相联系的唯一识别号,在 GetBaseSet 函数中,通过唯一识别号来决定返回哪个记录集合对象的指针。

根据不同的识别号,有选择性的返回指向不同记录集合对象的指针,这样就会使在应用程序 Draw 中增加一个记录集合对象变得较为容易;对于在应用程序 Draw 中新建的一个记录集合对象,在结构数组 DataInfo 中增加一个项目,并对 GetBaseSet 函数的代码略做修改,就可以达到返回指向这个记录集合对象的指针,实现针对这个记录集合对象的各个通用操作功能。具体操作可参见下面 18.2.4 小节。

(3) 实现选择记录集合对象的交互操作界面

当数据库应用程序中有多个记录集合对象时，应该建立一个交互操作界面来选择需要操作的记录集合对象，具体实现方法可参考如下：在浏览视图 CDatabaseView 的操作界面上，通过建立一个控制(如建立菜单、对话框窗口、对话条等)，将当前文档对象中所具有的各个记录集合对象显示出来，即通过一个列表，把文档对象中 DataInfo 结构数组所记录的各个记录集合对象的描述名称依次列出。在交互操作界面上，通过对这个列表中项目的选择，得到要进行操作的记录集合对象在 DataInfo 结构数组的序号，以这个序号为参数调用文档类的 GetDataSet 函数，返回指向要选择的记录集合对象的 CDataBaseSet 类指针。通过这个 CDataBaseSet 类指针，调用视图类 CDatabaseView 中的成员函数 Init1、Init2 对数据库浏览视图的滚动区域进行设置，并激活视图重画，就改变了操作的记录集合对象。可以对每个记录集合对象进行诸如数据浏览、数据删除、通用查询、通用排序等通用操作功能。

读者可以采用不同的方法在 CDatabaseView 视图中建立一个交互操作界面。通过交互操作界面，得到要选择的记录集合对象在文档对象的 DataInfo 结构数组中的序号 nSelectData。得到这个序号后，在视图类 CDataBaseView 中可以通过以下的代码来改变记录集合对象：

```
if(m_pSet->IsOpen())          //如果打开了记录集合对象
    m_pSet->Close();          //关闭记录集合对象
//以下代码通过 GetDataSet 函数得到指向第 nSelectData 个记录集合对象的指针 m_pSet
m_pSet = pDoc->GetDataSet(nSelectData);
m_pSet->Open();               //打开记录集合对象
nIndex = -1;                  //将浏览视图中记录的序号设置为 -1,即不选中一条记录
Init1();                      //计算视图滚动区域的横向参数
Init2();                      //计算视图滚动区域的纵向参数
SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);   //设置视图的滚动区域
Invalidate();                 //重画视图
```

18.2.4　增加应用程序管理的记录集合对象

在应用程序 Draw 中，期望实现对多个不同的记录集合对象提供管理功能。在 18.2.3 小节中，实现了从多个记录集合对象中选择一个记录集合对象的方法。在已经实现的程序框架基础上，可以较容易地增加一个新的记录集合对象，并为这个记录对象提供各个管理功能。本小节介绍在应用程序 Draw 中增加一个可被操作的记录集合对象的步骤。

(1) 建立记录集合对象类

要在应用程序 Draw 中增加一个新的记录集合对象，一般需要首先创建一个记录集合类。*按照 14.2.2 小节介绍的步骤，在应用程序 Draw 中创建一个 CRecordSet 类的派生类。例如，可*以在应用程序 Draw 中创建一个 CRecordSet 类的派生类 CData2Set，类的头文件在 data2set.h，实现文件在 data2set.cpp。

创建记录集合类 CData2Set 后，按照 18.2.2 小节(2)中介绍的步骤，将记录集合类 CData2Set

的基类由 CRecordSet 改为 CDataBaseSet，即 CData2Set 类是由记录集合基类 CDataBaseSet 派生的。

(2) 定义记录集合对象

在应用程序 Draw 的文档类 CDrawDoc 中，为新增的记录集合类定义一个记录集合对象。例如，对于新增加的记录集合类 CDataSet，在文档类 CDrawDoc 中定义一个记录集合对象：

```
CData2Set m_data2;
```

如有必要，在文档类 CDrawDoc 的定义前插入 CData2Set 类的头文件：

```
#include "data2set.h"
```

(3) 修改文档类中存储记录集合对象的信息和记录集合对象选择函数

对文档类 CDrawDoc 中的结构数组 DataInfo 进行修改，增加一个新的数组项来存储新增的记录集合对象信息。对 CDrawDoc 类的构造函数做如下修改：

```
CDrawDoc::CDrawDoc()
{
    ......//省略以上代码
    strcpy(DataInfo[0].Name,"城市信息");
    DataInfo[0].only_ID=1;
    strcpy(DataInfo[1].Name,"资源信息");
    DataInfo[0].only_ID=2;
    m_NumbData=2;
}
```

以上黑体部分代码，在 DataInfo 结构数组中增加了新增记录集合对象的信息。新增记录集合对象的唯一识别号为 2，存储的是与资源有关的数据信息。

同时，对文档类 CDrawDoc 的函数 GetDataSet 进行修改，使它能够返回指向新增记录集合对象的指针：

```
CDataBaseSet* CDrawDoc::GetDataSet(int Index)
{
    CDataBaseSet *m_pSet;
    if(Index<0||Index>=m_NumbData)      //如果序号不在有效范围内，返回空指针
        return NULL;
    short ID=DataInfo[Index].only_ID;   //得到记录集合对象的识别号
    switch(ID)                          //根据不同的识别号，返回相应的记录集合对象指针
    {
        case 1:
            m_pSet=&m_data1;
            break;
        case 2:
```

```
            m_pSet = &m_data2;
            break;
        default:
            m_pSet = NULL;
    }
    return m_pSet;
}
```

GetDataSet 函数中的黑体部分代码，实现在记录集合对象的唯一识别号为 2 时，返回指向新增记录集合对象 m_data2 的指针。

经过(1)～(3)三个步骤，就在应用程序 Draw 中新增了一个记录集合对象，可以在数据库浏览视图中选择这个记录集合对象，并能够直接给这个记录集合对象提供了视图浏览、记录删除、过滤、查询等已通用化的操作功能。

(4) 建立针对新增记录集合对象的记录编辑对话框

当在应用程序 Draw 中新增一个记录集合对象 m_data2 后((1)～(3))，对这个记录集合对象可以直接提供了浏览、记录删除、通用过滤、通用查询等已通用化的操作功能，而对于记录增加或记录修改功能，则需要增加一个针对 CData2Set 记录集合类的数据记录编辑对话框类，通过在对话框中建立控制并与 CData2Set 类中的映射变量建立联系，来完成对数据记录的交互编辑操作功能。可以参考 15.3 节介绍的步骤，在应用程序 Draw 中增加一个对话框类 CData2Dlg，并对对话框类进行修改，完成对记录集合类 CData2Set 对象的记录编辑功能。

在建立了编辑记录集合类 CData2Set 对象记录的对话框类 CData2Dlg 后，为保证在数据库浏览视图中，能够根据不同的记录集合对象提供不同的编辑对话框，需对视图类 CDatabaseView 的成员函数 OnRecordAdd 和 OnRecordDelete 进行修改。以下是修改后的 OnRecordAdd 函数实现代码：

```
void CDatabaseView::OnRecordAdd()
{
    CDrawDoc * pDoc = GetDocument();                    //得到当前视图对象的文档对象指针
    if(m_pSet->IsOpen())                                //如果打开了记录集合对象，则关闭它
        m_pSet->Close();
    int ID = pDoc->DataInfo[nSelectData].only_ID;       //得到当前选择的记录集合对象的唯一识别号
    if(ID == 1)                                         //如果记录集合对象的唯一识别号为 1
    {
        CData1Dlg dlg(m_pSet1,1,GetOnlyID());           //定义一个对话框类 CData1Dlg 对象
        if(dlg.DoModal() == IDOK)                       //调用 dlg 对象管理的对话框，当保存退出对话框时
        {
            Init2();                                    //计算视图滚动区域的纵向参数
            SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);     //设置视图的滚动区域
            UpdateWindow();
            Invalidate();                               //重画视图
        }
```

```
        }
        else if(ID = = 2)                                    //如果记录集合对象的唯一识别号为 2
        {
            CData1Dlg dlg(m_pSet2,1,GetOnlyID()); //定义一个对话框类 CData2Dlg 对象
            if(dlg.DoModal() = = IDOK)          //调用 dlg 对象管理的对话框,当保存退出对话框时
            {
                Init2();                                     //计算视图滚动区域的纵向参数
                SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);    //设置视图的滚动区域
                UpdateWindow();
                Invalidate();                                //重画视图
            }
        }
    }
```

在以上代码中,m_pSet2 是一个 CData2Set 类指针,指向定义在文档类 CDrawDoc 中的记录集合对象 m_data2。m_pSet2 在视图类 CDatabaseView 中定义如下:

```
CData2Set * m_pSet2;
```

在视图类 CDatabaseSet 的 OnInitialUpdate 函数中,m_pSet2 被初始化指向 CData2Set 类对象 m_data2:

```
void CDatabaseView::OnInitialUpdate()
{
    nSelectData = 0;
    CDrawDoc * pDoc = GetDocument();
    m_pSet1 = &pDoc->m_data1;
    m_pSet2 = &pDoc->m_data2;
    m_pSet = pDoc->GetDataSet(nSelectData);
    Init1();
    Init2();
    CScrollView::OnInitialUpdate();
    SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);
}
```

同样,需要对视图类 CDatabaseView 的 OnRecordModify 函数进行修改,修改后的函数实现代码如下:

```
void CDatabaseView::OnRecordModify()
{
    CDrawDoc * pDoc = GetDocument();
    if(nIndex < = 0&&nIndex > nRecord) //如果记录指针没有指向记录集合的一条记录,函数返回
        return;
    int ID = pDoc->DataInfo[nSelectData].only_ID;   //得到记录集合对象的唯一识别号
    if(ID = = 1)                                    //如果记录集合对象的唯一识别号为 1
    {
```

```
            if(m_pSet1->IsOpen())          //如果打开了 m_pSet1 指向的记录集合对象,则关闭它
                m_pSet1->Close();
            m_pSet1->Open();               //重新打开记录集合
            m_pSet1->Move(nIndex,SQL_FETCH_ABSOLUTE);      //将记录指针指向第 nIndex
                                                           //条记录
            CData1Dlg dlg(m_pSet1,0,m_pSet1->m_only_id);   //定义一个对话框类对象
            if(dlg.DoModal())              //调用对话框,如果保存退出了对话框
            {
                Init2();                   //设置浏览视图的纵向参数
                SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);  //设置视图滚动区域
                Invalidate();              //重画视图
            }
        }
        else if(ID==2)                     //如果记录集合对象的唯一识别号为 2
        {
            if(m_pSet2->IsOpen())          //如果打开了 m_pSet2 指向的记录集合对象,则关闭它
                m_pSet2->Close();
            m_pSet2->Open();               //重新打开记录集合
            m_pSet2->Move(nIndex,SQL_FETCH_ABSOLUTE);      //将记录指针指向第 nIndex
                                                           //条记录
            CData2Dlg dlg(m_pSet2,0,m_pSet2->m_only_id);   //定义一个对话框类对象
            if(dlg.DoModal())              //调用对话框,如果保存退出了对话框
            {
                Init2();                   //设置浏览视图的纵向参数
                SetScrollSizes(MM_TEXT, sizeTotal,sizePage,sizeLine);  //设置视图滚动区域
                Invalidate();              //重画视图
            }
        }
    }
```

在以上代码中,根据不同的记录集合对象,调用不同的对话框来对当前记录进行修改操作。

18.2.5　增加通用数据库操作功能

在本节的讨论中,记录集合基类 CDataBaseSet 只是一个起抽象作用的空类,在类中没有本身的成员变量。在开发数据库应用程序时,可以在这个类中增加成员变量和成员函数,组织实现具体的操作功能,而供所有由 CDataBaseSet 类派生的记录集合类继承使用。

在第 16 章讨论通用过滤功能时,曾经介绍在记录集合类中提供对过滤条件的存储功能,存储和管理用过滤条件编辑器组织得到的过滤条件。可以在记录集合基类 CDataBaseSet 中增加成员变量和成员函数,完成管理过滤条件的功能。在记录集合基类 CDataBaseSet 中定义如下的成员变量和成员函数:

```
class CDataBaseSet : public CRecordSet
```

```
{
.....//省略以上代码
protected:
    CString m_strCondition[30];          //用来存储过滤条件式,可以最多存储30个查询条件
    short m_NumbStr;                     //存储过滤条件式的数目
public:
    BOOL AddStr(CString m_str);
    BOOL ModifyStr(int Index,CString m_str);
    BOOL DeleteStr(int Index);
    BOOL GetStr(int Index,CString& m_str);
}
```

在 CDataBaseSet 类中,定义了多个成员函数来完成管理过滤条件的功能。下面分析实现各个成员函数。

① AddStr 函数。AddStr 函数完成增加一个过滤条件的功能,函数的参数 m_str 记录的是要增加的过滤条件式,操作成功时函数返回 TRUE,否则函数返回 FALSE。函数的实现代码如下:

```
BOOL CDataBaseSet::AddStr(CString m_str)
{
    if(m_NumbStr>=30)                    //如果存储的过滤条件式大于等于30,不能再增加
                                         //过滤条件,函数返回FALSE
        return FALSE;
    m_strCondition[m_NumbStr++]=m_str;   //记录要增加的过滤条件,过滤条件的数目加1
    return TRUE;
}
```

② ModifyStr 函数。ModifyStr 函数用来完成修改第 Index 个过滤条件的功能。参数 Index 是要修改的过滤条件的 0 基序号,参数 m_str 是要修改成的过滤条件。函数的实现代码如下:

```
BOOL CDataBaseSet::ModifyStr(int Index,CString m_str)
{
    if(Index<0||Index>=m_NumbStr)        //如果要修改的过滤条件的序号不在合法的范围内
        return FALSE;                    //函数返回FALSE
    m_strCondition[Index]=m_str;         //修改第Index个过滤条件
    return TRUE;
}
```

③ DeleteStr 函数。DeleteStr 函数用来删除第 Index 个过滤条件,参数 Index 是要删除的过滤条件的 0 基序号。函数的实现代码如下:

```
BOOL CDataBaseSet::DeleteStr(int Index)
{
    if(Index<0||Index>=m_NumbStr)        //如果要删除的过滤条件的序号不在合法的范围内
        return FALSE;                    //函数返回FALSE
```

```
    //以下对数组 m_strCondition 滚动,删除第 Index 个项
    for(int I = Index;I < m_NumbStr - 1;I ++ )
        m_strCondition[I] = m_strCondition[I + 1];
    m_NumbStr - - ;                    //过滤条件的数目减 1
    return TRUE;
}
```

④ GetStr 函数。GetStr 函数用来返回第 Index 个过滤条件。参数 Index 是要返回的过滤条件的 0 基序号,CString& 型参数 m_str 用来返回过滤条件。函数的实现代码如下:

```
BOOL CDataBaseSet::GetStr(int Index,CString& m_str)
{
    if(Index < 0||Index > = m_NumbStr)    //如果要返回的过滤条件式的序号不在合法的范围内
        return FALSE;                     //函数返回 FALSE
    m_str = m_strCondition[Index];        //得到要返回的过滤条件式
    return TRUE;
}
```

对对话框资源 IDD_FILTER 和对话框类 CFilterDlg 进行修改,扩充通用过滤条件编辑器的功能,使过滤条件编辑器不但能够交互编辑过滤条件式,而且能够对编辑完成的过滤条件式进行存储、修改、删除等操作,通过选择这些存储的过滤条件式来完成过滤操作功能。这样就可以实现功能更强、操作性更好的通用过滤操作功能。

18.3　数据库管理功能完全通用化

(1) 数据库管理功能完全通用化需要扩充的功能

本章讨论实现的对数据库管理功能的通用化操作,并非一种完全的通用化操作。所谓"非完全",是因为在本章实现的数据库管理功能,只能对预先确定的记录集合对象进行管理,要实现数据库管理功能完全通用化,至少需要在以下两个方面实现功能扩充:

① 动态创建记录集合对象。本篇实现的数据库管理功能,对需要进行管理的记录集合对象(数据库)需要在程序设计阶段确定,通过创建 ODBC 数据源和记录集合类来达到对数据库的管理。完全通用化的数据库管理系统,需要具备动态组织记录集合对象的能力,即应用程序能够管理的记录集合对象是在应用程序的运行过程中,而非在应用程序的设计阶段中确定的。同时,在完全通用化的数据库管理系统中,需要能够提供创建数据表或修改数据表结构的能力,对创建或修改得到的数据表能够提供通用的数据库管理功能。

② 动态地组织记录编辑界面。在本篇实现的数据库管理功能中,在应用程序设计时,针对不同的记录集合对象,需要设计不同的数据记录编辑对话框类。在实现完全通用化的数据库管理功能时,针对系统能够管理的记录集合对象的任意性,需要实现动态组织的数据记录编辑界面,在程序的运行过程中,根据需要编辑的记录集合对象的数据结构,动态产生编辑记录集合数据记录的操作界面,通过这个操作界面,对记录集合对象的记录实现编辑操作(增加记录、修改记录等)。

(2) 开发通用化数据库管理系统的方法

在开发一般的数据库管理系统时,采用 MFC 类和 MFC 数据库应用程序框架具有不可比拟的优势,而在开发完全通用化的数据库管理系统时,采用 MFC 类和 MFC 数据库应用程序框架并不能完全解决问题。

作者开发完成的一套 GIS 系统工具,采用的是系统自身组织的数据库,对于这种自身组织的数据库,开发者能够达到对数据库的完全控制,实现创建和编辑数据库结构、动态创建数据库、动态组织记录编辑界面等各种功能。但这种自身组织的数据库,很难应用于网络中,开发的难度和工作量也非常大。除非开发一些产品级应用程序,一般不采用这种方法。

直接利用 ODBC API 函数,可以组织更灵活的数据库管理系统设计,MFC 中与数据库应用程序有关的类,就是通过封装 ODBC API 函数来实现的,直接利用 ODBC API 函数,能够开发出在流程和功能上更灵活的数据库应用程序,组织完成完全通用化数据库管理系统的开发。在动态组织记录集合对象方面,采用 DAO 进行数据库应用程序设计具有更大的优势,采用 DAO 能够直接利用 Access 的 .mdb 数据库,来动态组织需要管理的记录集合对象。

采用 MFC 类,能够满足通常的数据库应用程序设计的需要,如果要实现更灵活和功能更强大的通用化数据库管理系统,就需要放弃 MFC 中固有的开发框架,去灵活组织数据库应用程序的设计。

基本 GIS 系统

一个基本 GIS 系统包含三个核心方面：矢量图形系统、数据库管理系统和矢量图形系统与数据库管理系统的连接。在第二篇完成的基本矢量图形系统和第三篇完成的数据库管理系统的基础上，建立两者之间的联系，就能够实现一个基本 GIS 系统。本篇将讨论 GIS 系统基本功能的实现方法，在应用程序 Draw 的基础上，开发矢量图形系统与数据库管理系统的相互连接功能，以及 GIS 系统的信息查询、空间统计分析等基本功能。

第 19 章 矢量图形系统与数据库管理系统的连接
第 20 章 在应用程序 Draw 中实现连接功能
第 21 章 实现双向信息查询
第 22 章 图形元素的空间拓扑关系
第 23 章 GIS 系统空间信息统计和分析功能

第四篇

第 19 章　矢量图形系统与数据库管理系统的连接

本章介绍在矢量图形系统与数据库管理系统间建立连接的方法。具体地讲，就是讨论在应用程序 Draw 中的矢量图形元素与数据表的记录之间建立连接的方法及具体的实现步骤。

19.1　连接方式及连接单元

本节介绍建立矢量图形元素与数据表的记录间的连接时，一般采用的连接方式及对应于连接方式的连接单元。

19.1.1　连接方式

对于一个矢量图形元素，可以采用不同的方式去连接数据表的记录，以下是常用的三种连接方式。

(1) 连接数据表的一条记录

在这种连接方式下，一个矢量图形元素连接的是数据表的一条记录。如在全国城市图上，有 2000 多个县级以上城市的标志，建立一个数据表存储各个城市的性质数据信息，将每个城市的信息记录在数据表的一条记录中，对于城市图中的每一个城市标志(一个矢量图形元素)，在数据表中有一个记录与其连接，这个记录存储了这个城市的性质数据信息。在这种情况下，数据表的一条记录作为一个连接单元，采用的是连接数据表的一条记录的连接方式。

这是一种最基本的连接方式，在这种连接方式下，数据表的每条记录有独立的唯一识别号，并作为一个连接单元与矢量图形元素建立连接。在应用程序 Draw 中，也就是把记录集合对象的每条记录作为一个连接单元来与矢量图形元素建立连接。

(2) 连接整个数据表

在某些情况下，矢量图形元素连接的是一个数据表。例如，在开发一个工程项目建设管理系统时，对于每个重点工程(如一个大桥)，在施工组织平面图中可以用一个桥梁图例来表示，在数据库管理系统中，一个数据表存储了有关这个重点工程(一个大桥)施工过程中随着时间变化的有关工程量、施工进度、投资等各方面的信息，即这个数据表中的所有记录都是针对这个重点工程的。在这种情况下，整个数据表作为一个连接单元，与一个矢量图形元素建立联系。在应用程序 Draw 中，在这种连接方式下，一个不带过滤条件的记录集合对象(即由数据表中选择数据记录形成记录集合时，没有经过过滤操作)作为一个连接单元，与矢量图形元素建立连接。

(3) 连接数据表的一个记录集合

还有另外一种连接方式,它连接的是数据表中的一部分记录。例如,在开发一个工程项目施工管理系统时,在一个数据表存储多个桥梁的随着时间变化的有关工程量、施工进度、投资等各个方面的数据资料。在这种情况下,一个桥梁图例(矢量图形元素)连接的是一个数据表中符合某个过滤条件的记录集合,这个经过过滤操作的记录集合(即由数据表中选择记录形成记录集合时,经过过滤操作)作为一个连接单元,与矢量图形元素建立连接。

19.1.2 连接单元

连接单元指的是矢量图形系统或数据库管理系统中可以用来进行连接操作的整体。

(1) 矢量图形系统中的连接单元

矢量图形系统中的连接单元可简称图形连接单元。在矢量图形系统中,一个矢量图形元素就是一个连接单元,即一个矢量图形元素可以作为一个连接操作的整体。

在矢量图形系统中,一个图例也是一个矢量图形元素,作为一个连接单元来与数据库管理系统中的连接单元进行连接。而一个图例是由多个基本图形元素组成的,通过图例这类图形元素,可以将多个基本图形元素组织成一个整体即一个连接单元,与数据库管理系统的连接单元进行连接。

(2) 数据库管理系统中的连接单元

数据库管理系统中的连接单元可简称数据连接单元。在数据库管理系统中,针对不同的连接方式有不同的连接单元,在连接数据表的一条记录的连接方式下,数据表的一条记录就是一个连接单元;而在连接数据表中一个记录集合的连接方式下,由数据表中符合指定过滤条件的记录集合组成一个整体即一个连接单元,来与矢量图形元素进行连接。

19.1.3 连接的特性

在 GIS 系统中,图形连接单元与数据连接单元之间可以进行连接操作。两者之间建立的连接,一般具有以下的特性。

(1) 连接的双向性

所谓连接的双向性,指的是通过图形连接单元可以得到与之连接的数据连接单元,通过数据连接单元,也可以得到与之连接的图形连接单元。即在 GIS 系统中,可以得到一个矢量图形

元素所连接的性质数据信息，也可以得到一个数据连接单元所连接的图形连接单元(矢量图形元素)。

(2) 连接的多项性

所谓连接的多项性，指的是一个图形连接单元可以连接多个数据连接单元，同时一个数据连接单元可以连接多个图形连接单元。

(3) 连接的永恒性

所谓连接的永恒性，指得是一旦建立了图形连接单元与数据连接单元之间的连接，在撤消这个连接之前，这个连接一直存在，不会因为对矢量图形系统的操作(如增加、删除图形元素等操作)和对数据库管理系统的操作(如增加数据表，增加、插入数据表记录等操作)而改变。

为了实现连接的永恒性，需要每个连接单元都具有一个唯一不变的识别号，即每个连接单元从创建时起，就具有一个与别的连接单元不同的标识，并且这个标识在连接单元存在周期内不会改变。通过这个标识建立起来的连接就能够保证其永恒性。19.2 节中将讨论标识连接单元的方法。

19.2　唯一标识连接单元

所谓标识，就是给每个连接单元一个唯一的识别号，通过这个识别号将这个连接单元与别的连接单元区分开来。

19.2.1　标识图形连接单元

对于矢量图形系统来说，每个图形元素是一个连接单元。在本书第二篇矢量图形系统的开发过程中，在图形元素基类 CDraw 中定义了一个成员变量 m_only_id(参见 5.1.1 小节)，通过它来唯一标识矢量图形元素。

在 6.3.1 小节中，在增加图形元素操作时，为了操作方便和加快操作速度，成员变量 m_only_id 在同类图形元素中具有唯一性，也就是说，每类图形元素中所有图形元素的成员变量 m_only_id 的值是唯一的(如有 10000 条直线，那么这 10000 条直线对象的成员变量 m_only_id 各不相同)，而对于不同类的图形元素，则 m_only_id 可具有相同的值(如一个直线对象和一个圆对象中的成员变量 m_only_id 可以相同)。所以要唯一地标识一个矢量图形元素，需要两个参数：图形类别和图形元素的唯一识别号。给每类图形元素一个类别号，通过这个类别号来表示这类图形元素，如可以如下定义各个图形元素类的类别号：

1 —— 直线
2 —— 连续直线和多边形区域
3 —— 圆和圆形区域
4 —— 圆弧
5 —— 文本
6 —— 图例

……

将这个图形类别号与图形元素的唯一识别号(成员变量 m _ only _ id)相结合,就可以在整个矢量图形系统范围内唯一的标识一个矢量图形元素,即一个图形连接单元。例如,一个直线对象,如果它的成员变量 m _ only _ id 的值为 200,则可以将这个矢量图形元素唯一标识为(1,200)。

在 6.3.1 小节完成增加图形元素功能时,如果在所有矢量图形元素范围内,确定要增加图形元素对象的成员变量 m _ only _ id,那么就可以只通过矢量图形元素的成员变量 m _ only _ id 来唯一地标识这个矢量图形元素(图形连接单元)。

19.2.2 标识数据连接单元

在数据库管理系统中,针对不同的连接方式有不同的连接单元。本小节分别讨论不同连接方式下标识数据连接单元的方法。

(1) 数据表的一条记录作为一个连接单元

在这种连接方式下,数据表中的一条记录作为一个连接单元。在第 15 章建立记录集合对象时,在建立用于 GIS 系统的数据表 city 时,在其结构中建立了一个字段 only _ id,这个字段在不同记录中有不同的值。在数据表中增加一条记录时,为这个字段计算并确定了唯一值(参见 15.4.1 小节)。这样,在一个记录集合对象范围内(这里所说的记录集合对象范围,指的是没有经过过滤操作,包含数据表的所有记录的记录集合),通过 only _ id 字段值就可以唯一标识一条记录。

为了能够在数据库管理系统范围内唯一标识一条记录,还需要一个参数,那就是记录集合对象的识别号。通过记录集合对象的识别号来表示连接的是哪个记录集合对象,通过字段 only _ id 值来表示连接的是记录集合对象中的哪条记录。

在 18.2.3 小节中,在应用程序 Draw 中,定义了一个 DataInfoStru 结构数组 DataInfo 来存储各个记录集合对象的信息。在 DataInfoStru 结构中有一个成员变量 only _ ID,用来存储记录集合对象的唯一识别号,通过这个识别号能够唯一的确定记录集合对象。

所以,当数据表的一条记录作为一个连接单元时,数据连接单元通过两个参数来唯一确定:记录集合对象的唯一识别号和记录集合对象中字段 only _ id 的值。

(2) 数据表的部分记录作为一个连接单元

对于另外的两种连接方式,即连接整个数据表和连接一个数据表中的部分记录(符合一定过滤条件的记录集合),可以采用相同的方法来处理。在这种连接方式下,连接单元是数据表中符合某个过滤条件的记录集合,为了在数据库管理系统范围内唯一标识这个连接单元,需要以下两个参数:记录集合对象的识别号和一个过滤条件式。对于记录集合对象的唯一识别号的实现方法,在(1)中作了介绍,过滤条件式则是用来对记录集合对象进行过滤操作时采用的过滤条件。

在这种连接方式下,标识一个连接单元需要的参数,与标识数据表的一条记录作为一个连接单元时不同。为了能够统一标识数据连接单元的参数,可以考虑在记录集合基类 CDataBas-

eSet 中，建立如下的成员变量来存储过滤条件：

```
class CDataBaseSet : public CRecord
{
    ......//省略以上代码
    CString m_strLink[30];
    int only_id[30];
    short m_NumbLink;
......//省略以下代码

}
```

CString 对象数组 m_strLink 用来存储多个过滤条件式，每个过滤条件式被用来对记录集合对象进行过滤操作而得到一个连接单元，only_id 数组存储 m_strLink 数组中对应过滤条件式的唯一识别号，m_NumbLink 记录过滤条件式即连接单元的数目（这种管理过滤条件式的方法，保证在增加、删除、插入过滤条件式操作时，既有过滤条件式的识别号不变）。通过 only_id 数组中某个项的值，可以唯一的得到一个过滤条件式。这样，通过记录集合对象的识别号和过滤条件式的唯一识别号这两个参数，可以在数据库管理系统范围内唯一标识一个数据连接单元。标识用的参数与数据表的一条记录作为一个连接单元时相同。

19.3　管理连接信息的方法

本节讨论在图形连接单元和数据连接单元间建立连接的具体方法。并介绍几种存储连接信息的方式。

19.3.1　连接的参数表示

按照 19.2 节中讨论的标识连接单元的方法，要完整的记录一个图形连接单元和一个数据连接单元之间的连接，需要使用多个参数。以下结构可以用来存储一个连接的参数：

```
typedef struct
{
    unsigned char lb;           //连接方式：  1－连接数据表的一条记录
                                //            2－连接数据表的一个记录集合
    unsigned char graph_id;     //图形类别号
    int graph_element_id;       //图形元素的识别号
    short database_id;          //记录集合对象的识别号
    int record_id;              //在连接方式 1 下，表示数据记录的唯一识别号
                                //在连接方式 2 下，表示过滤条件式的唯一识别号
} LinkStru;
```

在以上结构中，成员变量 lb 用来表示连接方式，当 lb = 1 时，表示数据表的一条记录作为连接单元的连接方式；当 lb = 2 时，表示数据表中符合某过滤条件的记录集合作为连接单元的连接方式。成员变量 graph_id 和 graph_element_id 用来表示一个图形连接单元（一个矢量图

形元素)的图形类别和图形元素唯一识别号。成员变量 database _ id 用来记录一个记录集合对象的唯一识别号。成员变量 record _ id 在不同的连接方式下有不同的含义:在第一种连接方式下(数据表的一条记录作为连接单元),record _ id 用来记录数据表中一条记录的唯一识别号;在第二种连接方式下(数据表的一个记录集合作为连接单元),record _ id 用来记录过滤条件式的唯一识别号(参见 18.2.3 小节中管理过滤条件的方法)。

一个 LinkStru 结构可以用来存储一个图形连接单元和数据连接单元之间连接的完整信息。在应用程序设计中,可以有多种不同的方法存储组织连接信息。

19.3.2 存储组织连接信息的方法

本小节讨论几种存储组织连接信息的方法。

(1) 分散管理连接信息

所谓分散管理连接信息,就是把记录图形连接单元和数据连接单元一个完整连接的信息分成两部分,分别与图形连接单元和数据连接单元存储在一起。下面分别介绍在图形连接单元和数据连接单元中存储连接信息的方法。

在图形连接单元中存储连接信息

一个图形连接单元,实际上就是一个矢量图形元素。可以通过对矢量图形元素类的修改,增加成员变量来组织存储与这个图形元素连接的数据连接单元。可以建立一个如下结构来存储一个与图形元素连接的数据连接单元:

```
typedef struct
{
    unsigned char lb;        //连接的方式: 1 - 连接数据表的一条记录,
                             //            2 - 连接数据表的一个记录集合
    short database _ id;     //记录集合对象的识别号
    int record _ id;         //连接方式 1 下,表示记录的识别号,方式 2 下,表示过滤条件式的识别号
} LinkStru1;
```

可以在图形元素基类 CDraw 中定义一个 LinkStru1 结构数组,来存储与图形元素存在连接的数据连接单元信息:

```
clase CDraw : public CObject
{
    ..... //省略以上代码
protected:
    LinkStru1 link[6];
}
```

在 CDraw 类中定义了一个可以存储六个连接信息的结构数组,即对于一个矢量图形元素,最多可以连接六个数据连接单元。为节约内存,也可以把这个存储连接信息的结构数组存储到硬盘的临时文件中。通过图形元素的类别和唯一识别号来得到这些信息在临时文件中的位置,并实现存取这些连接信息的操作。

在数据连接单元中存储连接信息

在分散存储连接信息的方式下，可以建立一个如下的结构来存储一个与数据连接单元连接的图形连接单元信息：

```
typedef struct
{
    unsigned char graph_id;        //图形类别号
    int graph_element_id;          //图形元素的识别号
}linkstru2;
```

为存储连接信息，在数据表的一条记录作为一个连接单元的连接方式下，在数据表中建立一个字段，用来存储与记录相连接的图形连接单元(可以通过建立一个字符型字段，将与各个图形连接单元的连接信息转变成格式化字符串存储)。在使用数据库记录等长的数据源(如 Foxpro 的 .dbf 数据库)时，为了使数据表的每条记录能够连接多个图形连接单元，需要预留较长的字段长度。

在数据表的一个记录集合作为一个连接单元的连接方式下，可以针对组织记录集合的每个过滤条件式建立一个 linkstru2 结构数组，来存储与这个数据连接单元存在连接的图形连接单元信息。

分散存储连接信息的优缺点

分散存储连接信息的方法，最显著优点是具有较快的连接信息查找速度。因为不管是对图形连接单元，还是数据连接单元，其连接信息不需要进行查询操作而直接得到。例如，当对一个图形连接单元(如一条直线)连接的数据连接单元查询时，因为通过这个图形元素的成员变量可以直接得到与之相连接的数据连接单元，所以不论矢量图形系统和数据库管理系统多么庞大，都可以迅速得到与其连接的性质数据信息。

但是，分散存储连接信息的方法，在组织和实现上具有很多缺点：

① 浪费存储空间。这种存储方法要求对于每一个连接单元(不论是图形连接单元，还是数据连接单元)都要预留出足够的固定空间来存储连接信息(一般按照最大的操作容量进行预留)，造成较大的存储空间的浪费。采用非固定长度的连接信息存储方法(即根据连接信息的数量动态分配存储连接信息的空间)，则给编程造成较大困难。

② 连接数目有限。用预留固定空间的方法来存储连接信息时，不管预留多大的存储空间(预留存储空间越大，造成存储空间的浪费越大)，对于每个图形连接单元(或数据连接单元)能够连接的数据连接单元(或图形连接单元)的数量是有限的。

③ 组织复杂。用分散存储连接信息的方法组织连接信息时，因为一个完整连接由图形连接单元和数据连接单元共同来存储，所以每增加一个连接时，需要在图形连接单元和数据连接单元两处都增加连接信息；同样，当删除一个连接时，也需要在图形连接单元和数据连接单元两处删除连接信息，程序的组织实现比较复杂。而且，在这种存储方式下，因为一个完整连接信息在两处存放，对于维护连接的统一性也需要有较高的要求，否则容易造成连接不正确(例如，一个图形连接单元中存储着与一个数据连接单元的连接信息，而在这个数据连接单元中却没有存储与这个图形连接单元的连接信息)。

(2) 集中管理连接信息

为了克服分散存储连接信息的缺点,可以采用集中管理连接信息的方法。所谓集中管理,就是在应用程序中,把所有的图形连接单元与数据连接单元的连接信息存储在一个共同存储区内。19.3.1 小节介绍了完整表示一个连接的数据结构 LinkStru,可以在应用程序中建立一个(或多个)大的 LinkStru 结构数组,来集中存储连接信息。为了节省内存,可以把结构数组存储到外部临时文件中。

集中存储连接信息,具有明显的优点:

① 节省存储空间。用这种方法存储连接信息,不需要开辟冗余的存储空间,与分散组织连接信息相比,可节省大量存储空间(或内存)。

② 连接容量不限。所谓连接容量不限,指的是在总的连接数量没有超过允许容量的情况下,对于每一个图形连接单元(或数据连接单元)可以连接任意多个数据连接单元(或图形连接单元),连接的数量不受限制。

③ 维护较简单。因为集中存储连接信息时,在增加或删除一个连接时,只需从集中管理的连接信息中增加或删除一个连接即可。因为连接信息在一处存放,所以不存在连接信息存储的统一性问题。

虽然集中存储连接信息优点较多,但这种方法也有一个缺点:那就是操作速度慢。因为必须对所有的连接信息进行检索,才能得到与某个图形或数据连接单元存在连接的数据或图形连接单元,这在连接信息非常多的情况下,会降低系统的运行速度。

(3) 使用数据表集中存储连接信息

当采用集中存储连接信息的方法来管理连接信息时,可以采用在内存中建立 LinkStru 结构数组的方法。当将连接信息通过在内存中建立大的结构数组来管理时,管理功能(如增加连接、删除连接,查找连接等)的实现方法比较简单,但是需要较大的内存空间,在连接信息非常多的情况下,内存空间可能满足不了要求。也可以利用外部临时文件来组织连接信息,这种方法虽然可以实现较大的容量,但是组织连接信息的管理功能(如增加连接、删除连接,查找连接等)时较为复杂,需要自身去编制较复杂的程序来完成这些功能。

另外一种管理连接信息的方法是:把连接信息存储在一个数据表中,数据表的每条记录存储一个连接,通过在应用程序中建立一个记录集合对象对数据表的管理,来实现管理连接信息的功能。这种方法具有以下优点:

① 能够满足多用户化的要求。将连接信息存放在数据表中,通过 ODBC 对数据表进行管理,通过采用容易实现多用户的数据源(如 SQL Server),可以使连接操作多用户化变得非常简单。

② 存储容量大。数据表能够处理数据记录的数量非常大的,对于一些中大型的数据库系统更是如此。所以采用数据表来存储连接信息能够满足存储容量方面的要求。

③ 程序设计简单。用数据表存储连接信息,通过 CRecordSet 派生类建立记录集合对象来管理数据表中的记录集合,利用 CRecordSet 类的成员变量可以对记录集合实现诸如增加、修改、删除、查询等普通数据库操作功能,来实现对连接信息的管理。

第 20 章，将利用数据表来集中管理连接信息，在应用程序 Draw 中建立针对这个存储连接信息数据表的记录集合对象，来实现对连接信息的管理，实现矢量图形系统与数据库管理系统的连接，为实现基本 GIS 系统奠定基础。

第 20 章　在应用程序 Draw 中实现连接功能

本章在第二篇实现的矢量图形系统和第三篇实现的数据库管理系统的基础上，进一步实现两者间的连接功能。在本章的讨论中，对于连接信息将采用数据表集中存储管理的方法。

20.1　建立管理连接信息的记录集合对象

为了管理连接信息，在 Access 数据源 Info.mdb 数据库中建立一个数据表 link _ info。数据表的结构如表 20-1 所示：

表 20-1　link _ info 数据表结构

字段名称	数据类型	含　义
link _ mode	short	连接方式
database _ id	short	记录集合对象的唯一识别号
record _ id	int	记录或者过滤条件式的唯一识别号
graph _ id	short	图形元素类别的识别号
element _ id	int	图形元素的唯一识别号

在本章的讨论中，将用这个数据表来存储连接信息。

20.1.1　新建记录集合类

为了对存储在数据表 link _ info 中的连接信息进行管理，在应用程序 Draw 中建立一个派生于 CRecordSet 类的记录集合类。按照 14.2.2 小节中介绍的步骤，在应用程序 Draw 中新建一个记录集合类 CLinkDataSet，类的头文件在 linkdataset.h，实现文件在 linkdataset.cpp。在头文件中，可以找到类的定义：

```
class CLinkDataSet : public CRecordset
{
public:
    CLinkDataSet(CDatabase * pDatabase = NULL);
    DECLARE _ DYNAMIC(CLinkDataSet)

// Field/Param Data
    //{{AFX _ FIELD(CLinkDataSet, CRecordset)
    int     m _ database _ id;
    long    m _ record _ id;
    int     m _ graph _ id;
    long    m _ element _ id;
```

```
    int      m_link_mode;
    //}}AFX_FIELD

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CLinkDataSet)
    public:
    virtual CString GetDefaultConnect();    // Default connection string
    virtual CString GetDefaultSQL();    // Default SQL for Recordset
virtual void DoFieldExchange(CFieldExchange* pFX);    // RFX support
    //}}AFX_VIRTUAL

// Implementation
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
};
```

在实现文件 linkdataset.cpp 中，有一个 CLinkDataSet 类的实现代码。实现文件 linkdataset.cpp的代码列表如下：

```
#include "stdafx.h"
#include "draw.h"
#include "LinkDataSet.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////
// CLinkDataSet

IMPLEMENT_DYNAMIC(CLinkDataSet, CRecordset)

CLinkDataSet::CLinkDataSet(CDatabase* pdb)
    : CRecordset(pdb)
{
    //{{AFX_FIELD_INIT(CLinkDataSet)
    m_database_id = 0;
    m_record_id = 0;
    m_graph_id = 0;
    m_element_id = 0;
```

```
    m_link_mode = 0;
    m_nFields = 5;
    //}}AFX_FIELD_INIT
    m_nDefaultType = snapshot;
}

CString CLinkDataSet::GetDefaultConnect()
{
    return _T("ODBC;DSN=GISDATA");
}

CString CLinkDataSet::GetDefaultSQL()
{
    return _T("[Link_Info]");
}

void CLinkDataSet::DoFieldExchange(CFieldExchange* pFX)
{
    //}}AFX_FIELD_MAP(CLinkDataSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Int(pFX, _T("[database_id]"), m_database_id);
    RFX_Long(pFX, _T("[record_id]"), m_record_id);
    RFX_Int(pFX, _T("[graph_id]"), m_graph_id);
    RFX_Long(pFX, _T("[element_id]"), m_element_id);
    RFX_Int(pFX, _T("[link_mode]"), m_link_mode);
    //}}AFX_FIELD_MAP
}
/////////////////////////////////////
// CLinkDataSet diagnostics

#ifdef _DEBUG
void CLinkDataSet::AssertValid() const
{
    CRecordset::AssertValid();
}

void CLinkDataSet::Dump(CDumpContext& dc) const
{
    CRecordset::Dump(dc);
}
#endif    //_DEBUG
```

从以上代码中可以看出，记录集合类 CLinkDataSet 管理由 ODBC 数据源 GISDATA 连接的

数据表 link _ info 中的全部内容(所有字段和记录)组成的记录集合。

20.1.2　创建管理连接信息的记录集合对象

为了利用记录集合类 CLinkDataSet 具体实现对连接信息的管理功能,在文档类 CDrawDoc 中创建一个记录集合类 CLinkDataSet 对象:

```
class CDrawDoc : public COleDocument
{
    ......//省略以上代码
public:
    CLinkDataSet m_linkdata1;

......//省略以下代码
}
```

为了能够在文档类 CDrawDoc 中使用 CLinkDataSet 类,需要在文档类的定义前插入 CLinkDataSet 类的头文件:

```
#include "linkdataset.h"
```

对于在文档类中定义的 CLinkDataSet 类对象 m _ linkdata1,为了能够在数据库浏览视图中对其进行操作,在视图类 CDatabaseView 中定义一个 CLinkDataSet 类指针:

```
class CDatabaseView : public CScrollView
{
    ......//省略以上代码
    protected:
        CLinkDataSet * m_pLinkSet1;
    ......//省略以下代码
}
```

并在视图类 CDatabaseView 的 OnInitialUpdate 函数中初始化这个类指针,使它指向在当前文档对象中创建的 CLinkDataSet 对象 m _ linkdata1:

```
void CDatabaseView::OnInitialUpdate()
{
    CDrawDoc * pDoc = GetDocument();
    ......//省略以上代码
    m_pLinkSet1 = &pDoc->m_linkdata1;
}
```

为了在视图类 CDatabaseView 的实现文件 dbaseview.cpp 中能够使用 CLinkDataSet 类,需要的话,在实现文件 dbaseview.cpp 中插入 CLinkDataSet 类的头文件:

```
#include "linkdataset.h"
```

20.2 组织进行连接操作的对话框类

在应用程序 Draw 中,需要提供交互操作界面编辑图形连接单元和数据连接单元间的连接信息。本节将实现一个对话框类,通过这个对话框类提供的交互操作界面,完成管理连接信息的功能。

20.2.1 连接操作的具体步骤

在应用程序 Draw 中,采用如下的步骤来增加连接信息:

① 在显示矢量图形系统的视图中,选择要与数据连接单元连接的一个或多个图形元素。

② 在数据库浏览视图中,选择图形连接单元要连接的数据连接单元。在数据表的一条记录作为数据连接单元的连接方式中,用鼠标在浏览视图中选中要连接的数据记录。

③ 在数据库浏览视图中,通过一个命令菜单调用一个对话框,通过这个对话框,交互组织选中的矢量图形元素与当前选择的数据连接单元间的连接。

以上是将要在应用程序 Draw 中实现的管理连接信息的操作步骤。对于步骤①、②,已经在应用程序 Draw 中实现,本节将介绍步骤③的操作功能,以实现对连接信息管理。

以上介绍的步骤只能对第一种连接方式(数据表的一条记录作为一个连接单元)的连接进行管理。在数据库浏览视图中,完成对记录集合对象中用于连接操作的过滤条件式的管理,即可以将数据表中符合过滤条件的记录集合组织成一个连接单元,在步骤②选择这类连接单元时,同样可以实现对第二种连接方式(数据表中符合指定过滤条件的记录集合作为一个连接单元)的连接进行管理。

20.2.2 创建对话框类

为了实现管理连接信息的对话框操作界面,首先需要在应用程序 Draw 中创建一个对话框类(CDialog 类的派生类)。在这个对话框类的基础上,扩充各种操作功能,最终实现交互管理连接信息的对话框界面。

(1) 创建对话框资源

为了创建对话框类,首先应该在应用程序 Draw 中创建一个对话框资源,利用 ResourceView 及其可视化编辑工具 AppStudio,创建一个如图 20-1 所示的对话框资源 IDD _ LINK。

在资源文件 draw.rc 中,可以找到 IDD _ LINK 对话框资源的代码描述:

```
IDD_LINK DIALOG DISCARDABLE 0, 0, 233, 181
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "数据连接单元与矢量图形元素连接操作对话框"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON   "保存退出",IDOK,104,152,46,14
    PUSHBUTTON      "放弃退出",IDCANCEL,157,152,47,14
```

图 20-1　IDD _ LINK 对话框资源

```
    LTEXT           "连接方式",IDC _ STATIC,104,25,42,12
    LTEXT           "数据库类别",IDC _ STATIC,103,41,42,12
    LTEXT           "记录识别号",IDC _ STATIC,103,60,43,11
    LTEXT           "图形类别",IDC _ STATIC,103,75,43,11
    LTEXT           "元素识别号",IDC _ STATIC,103,93,46,15
    EDITTEXT        IDC _ EDIT1,150,24,22,12,ES _ AUTOHSCROLL | ES _ READONLY
    EDITTEXT        IDC _ EDIT2,150,39,31,12,ES _ AUTOHSCROLL | ES _ READONLY
    EDITTEXT        IDC _ EDIT3,150,58,37,12,ES _ AUTOHSCROLL | ES _ READONLY
    EDITTEXT        IDC _ EDIT4,150,75,19,11,ES _ AUTOHSCROLL | ES _ READONLY
    EDITTEXT        IDC _ EDIT5,150,93,46,12,ES _ AUTOHSCROLL | ES _ READONLY
    LTEXT           "已存在的连接",IDC _ STATIC,32,11,51,10
    LTEXT           "选中的图形元素",IDC _ STATIC,23,83,64,11
    LISTBOX         IDC _ LIST1,13,23,79,53,LBS _ MULTIPLESEL |
                    LBS _ NOINTEGRALHEIGHT | WS _ VSCROLL | WS _ TABSTOP
    LISTBOX         IDC _ LIST2,15,96,78,71,LBS _ NOINTEGRALHEIGHT | WS _ VSCROLL |
                    WS _ TABSTOP
    PUSHBUTTON      "增加连接",IDC _ LINK _ ADD,102,113,46,13
    PUSHBUTTON      "全部增加",IDC _ LINK _ ADDALL,156,113,46,13
    PUSHBUTTON      "删除连接",IDC _ LINK _ DELETE,103,133,46,13
    PUSHBUTTON      "全部删除",IDC _ LINK _ DELALL,157,133,46,13
END
```

在对话框资源 IDD _ LINK 中，列表框 IDC _ LIST1 和 IDC _ LIST2 不具有排序特性(设置方法见 15.3.1 小节)；编辑框 IDC _ EDIT1、IDC _ EDIT2、IDC _ EDIT3、IDC _ EDIT4、IDC _ EDIT5 被设置成只读特性，即在资源文件代码中，给各个编辑框设置了 ES _ READONLY 选项。在可视化编辑对话框资源 IDD _ LINK 时，通过在编辑框的 styles 属性页中选择“Read - only”选项来设置其只读特性(如图 20-2 所示)。

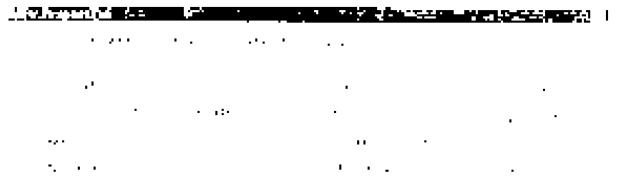


图 20-2 编辑框控制的 Styles 属性页

(2) 创建对话框类

在应用程序 Draw 中,采用第 6 章 6.4.4 小节中介绍的步骤,用 ClassWizard 创建一个对话框类 CLinkDlg,对话框类使用对话框资源 IDD _ LINK,类的头文件在 linkdlg.h,实现文件在 linkdlg.cpp。创建对话框类 CLinkDlg 后,为了能够在对话框类 CLinkDlg 中管理记录集合类 CLinkDataSet 对象,按照第 15 章 15.3.3 小节介绍的步骤,给对话框类 CLinkDlg 增加一个 CLinkDataSet 对象外部引用指针 m _ pSet。

完成操作后,可以在头文件 linkdlg.h 中找到 CLinkDlg 类的定义:

```
class CLinkDataSet;
class CLinkDlg : public CDialog
{
// Construction
public:
    CLinkDlg(CWnd * pParent = NULL); // standard constructor

// Dialog Data
    //{{AFX_DATA(CLinkDlg)
    enum { IDD = IDD_LINK };
    CLinkDataSet * m_pSet;
    //}}AFX_DATA
protected:
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CLinkDlg)
    protected:
    virtual void DoDataExchange(CDataExchange * pDX); // DDX/DDV support
    //}}AFX_VIRTUAL
// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CLinkDlg)
```

```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

在实现文件 linkdlg.cpp 中，列出了对话框类 CLinkDlg 的实现代码：

```
#include "stdafx.h"
#include "draw.h"
#include "drawdoc.h"
#include "LinkDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////
// CLinkDlg dialog

CLinkDlg::CLinkDlg(CWnd* pParent /* = NULL*/)
    : CDialog(CLinkDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CLinkDlg)
    m_pSet = NULL;
    //}}AFX_DATA_INIT
}

void CLinkDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CLinkDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CLinkDlg, CDialog)
    //{{AFX_MSG_MAP(CLinkDlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////
// CLinkDlg message handlers
```

以上列出了对话框类 CLinkDlg 的定义和实现代码，黑体部分代码是增加一个记录集合类

CLinkDataSet 对象外部引用指针 m _ pSet 而加入的代码。

下面将在这个对话框类 CLinkDalg 基础上，增加各种操作功能，实现管理连接信息的对话框界面。

(3) 对话框类中要完成的操作功能

为了完成对连接信息的管理功能，需要在对话框类 CLinkDlg 中，增加如下的操作功能：

① 启动对话框时，在列表框 IDC _ LIST1("已存在的连接"列表框)中列出数据连接单元已经具有的连接，在列表框 IDC _ LIST2("选中图形元素")中列出在矢量图形中选中的图形元素，在编辑框 IDC _ EDIT1("连接方式")中列出连接方式(1 - 数据表的一条记录作为一个连接单元，2 - 数据表的一个记录集合作为一个连接单元)，在编辑框 IDC _ EDIT2("记录集合对象类别")和 IDC _ EDIT2("记录识别号")中分别列出启动对话框前选择的数据连接单元的记录集合对象识别号和记录唯一识别号(在第二种连接方式下，是记录集合过滤条件式的唯一识别号)。

② 点中列表框 IDC _ LIST2 中的一个矢量图形元素时，会在编辑框 IDC _ EDIT4("图形类别")和 IDC _ EDIT5("图形元素的识别号")中分别列出这个矢量图形元素的图形类别号和唯一识别号。

③ 在对话框中，按中"增加连接"按钮，可以把数据连接单元与在 IDC _ LIST2 列表框中选中的矢量图形元素间建立连接；按中"全部增加"按钮，可以把数据连接单元与在列表框 IDC _ LIST2 中列出的所有图形元素建立连接；按中"删除连接"按钮，可以删除在列表框 IDC _ LIST1 中选择的连接；按中"全部删除"按钮，可以删除当前数据连接单元所有的连接(即从列表框 IDC _ LIST1 中删除所有项目)。

④ 按下"保存退出"按钮时，保存在对话框中对数据连接单元连接信息的修改并退出对话框；按下"放弃退出"按钮时，放弃对话框中对数据连接单元连接信息所做的修改并退出对话框，即可以完全放弃在对话框的编辑操作。

20.2.3 初始化对话框类

本小节在对话框类 CLinkDlg 中定义各个成员变量，并对对话框类的构造函数和初始化函数 OnInitDialog 进行修改，以初始化启动对话框时的初始状态。

(1) 定义成员变量

在对话框类 CLinkDlg 中，定义以下各成员变量：

```
class CLinkDlg : public CDialog
{
.....//省略以上代码
protected:
    short m _ DatabaseID;
    int m _ RecordID;
    short m _ LinkMode;
```

```
    CDrawDoc * m_pDoc;
    CListBox * plist1, * plist2;
    short m_GraphLB[200],m_GraphLB1[200];
    int m_GraphID[200],m_GraphID1[200];
    short m_NumbLink,m_NumbLink1;
    CString graph_Name[8];
    ......//省略以下代码
}
```

变量 m_DatabaseID 和 n_RecordID 分别用来记录数据连接单元的记录集合对象识别号和记录唯一识别号(在数据连接单元是数据表的一个记录集合时,记录的是过滤条件式的唯一识别号)。变量 m_LinkMode 用来记录连接方式。文档类 CDrawDoc 指针 m_pDoc 用来记录指向当前文档对象的指针。

plist1 和 plist2 分别用来记录指向对话框中的列表框 IDC_LIST1 和 IDC_LIST2 的指针。

数组 m_GraphLB 和 m_GraphID 分别用来记录当前数据连接单元所连接的矢量图形元素的图形类别号和唯一识别号。变量 m_NumbLink 用来记录存在连接的数目。数组 m_GraphLB1 和 m_GraphID1 同样也是用来记录数据连接单元连接的图形元素的图形类别号和唯一识别号,m_NumbLink1 记录存在连接的数目。第二套成员变量(m_GraphLB1、m_GraphID1、m_NumbLink1)用来记录数据连接单元初始状态的连接信息,供放弃操作时使用;第一套成员变量(m_GraphLB、m_GraphID、m_NumbLink)用来记录数据连接单元操作过程中随着变化的连接信息。数组项设置成 200 时,只能处理一个数据连接单元最多连接 200 个图形连接单元(矢量图形元素)的情况,可以根据实际情况将数组项数增加,以处理连接更多图形连接单元的情况(也可以动态分配数组空间)。

CString 对象数组 Graph_Name 用来存储各类图形类别的名称,用于对话框中的显示。

(2) 修改对话框类的构造函数

为了能够在启动对话框时,把各个参数传递到对话框类对象中,在对话框类 CLinkDlg 中修改构造函数的定义如下(也可以在对话框中增加一个构造函数的重载):

```
CLinkDlg(CDrawDoc * pDoc,CLinkDataSet * pSet,short LinkMode,short
    DatabaseID,int RecordID,CWnd * pParent = NULL);
```

参数 pDoc 传递启动对话框时当前文档对象的指针;参数 pSet 用来传递指向记录连接信息的记录集合类 CLinkDataSet 对象的指针;参数 LinkMode、DatabaseID 和 RecordID 传递当前数据连接单元的连接方式、记录集合对象的识别号和记录或过滤条件式的唯一识别号。在构造函数中加入对各成员变量的初始化代码,得到修改后的构造函数实现代码如下:

```
CLinkDlg::CLinkDlg(CDrawDoc * pDoc,CLinkDataSet * pSet,short LinkMode,short
    DatabaseID,int RecordID,CWnd * pParent /* = NULL */)
    : CDialog(CLinkDlg::IDD, pParent)
{
    m_pDoc = pDoc;                    //记录指向当前文档对象的指针
```

```
    m_pSet = pSet;                    //记录指向存储连接信息的记录集合对象的指针
    m_DatabaseID = DatabaseID;        //记录数据连接单元属于的记录集合对象识别号
    m_RecordID = RecordID;            //记录数据连接单元的记录识别号(或过滤条件式识别号)
    m_LinkMode = LinkMode;            //记录连接方式
    if(m_pSet->IsOpen())              //如果打开了记录集合对象,则关闭它
        m_pSet->Close();
    m_pSet->Open();                           //打开记录集合对象
    m_NumbLink = 0;                           //连接的数目为 0
    if(! m_pSet->IsBOF())                     //如果记录集合对象中包含记录
    {
        m_pSet->MoveFirst();                  //指向记录集合的第 1 条记录
        while(! m_pSet->IsEOF())              //遍历整个记录集合,记录连接的矢量图形元素的
                                              //类别和识别号
        {
            m_GraphLB[m_NumbLink] = m_GraphLB1[m_NumbLink] = m_pSet->m_graph_id;
            m_GraphID[m_NumbLink] = m_GraphID1[m_NumbLink] = m_pSet->m_element_id;
            m_NumbLink++;
            m_pSet->MoveNext();               //指向记录集合的下一条记录
        }
        m_pSet->MoveFirst();                  //指向记录集合的第一条记录
    }
    m_NumbLink1 = m_NumbLink;                 //连接数目
}
```

构造函数完成的主要功能如下:

① 对对话框类 CLinkDlg 中的成员变量进行初始化。通过构造函数的参数得到了启动对话框时的数据连接单元的连接方式、记录集合对象识别号、记录识别号,以及指向当前文档对象的指针 pDoc 和指向存储连接信息的记录集合类 CLinkDataSet 对象的指针 m_pSet。

② 通过指针 m_pSet 重新打开了记录集合对象,然后对记录集合的所有记录遍历,利用数组 m_GraphLB(及 m_GraphLB1)、m_GraphID(及 m_GraphID1)记录下了当前数据连接单元所连接的所有矢量图形元素的图形类别和唯一识别号。m_pSet 指向的记录集合对象在调用对话框前设置了过滤条件,保证打开记录集合对象时,记录集合中的所有记录存储的是当前数据连接单元的连接信息,具体实现方法参见本章 20.3 节。

(3) 增加和修改 OnInitDialog 函数

在对话框类 CLinkDlg 中,增加窗口消息 WM_INITDIALOG 的处理函数 OnInitDialog(实际上是重载了 CDialog 类中的虚函数)。在对话框类 CLinkDlg 的定义中增加了函数的定义代码:

```
//{{AFX_MSG(CLinkDlg)
virtual BOOL OnInitDialog();
//}}AFX_MSG
```

在实现文件 linkdlg.cpp 中,对 OnInitDialog 函数进行修改,得到修改后的函数实现代码

如下：

```
BOOL CLinkDlg::OnInitDialog()
{
    CString m_str;
    char p1[10];
    plist1 = (CListBox *)GetDlgItem(IDC_LIST1); //得到指向列表框 IDC_LIST1 的指针
    plist2 = (CListBox *)GetDlgItem(IDC_LIST2); //得到指向列表框 IDC_LIST2 的指针
    plist1 -> ResetContent();                   //清空列表框 IDC_LIST1
    plist2 -> ResetContent();                   //清空列表框 IDC_LIST2
    //以下定义各个图形类别的名称
    graph_Name[0] = "直线";
    graph_Name[1] = "连续直线或多边形区域";
    graph_Name[2] = "圆或圆形区域";
    graph_Name[3] = "圆弧";
    graph_Name[4] = "标注文本";
    for(int i = 0;i < m_NumbLink;i++)    //对所有连接循环,在列表框 IDC_LIST1 中列出已经
                                         //存在的连接
    {
        sprintf(p1,"%7d",m_GraphID[i]); //将第 i 个连接的矢量图形元素的识别号转成字符串
        m_str = p1;                     //字符串 m_str 设置成图形元素识别号的
                                        //字符串
        m_str += " ";                   //在字符串 m_str 后加几个空格
        m_str += graph_Name[m_GraphLB[i] - 1];     //在字符串 m_str 后加上图形类别的名称
        plist1 -> AddString(m_str);     //将字符串 m_str 增加到列表框 IDC_LIST1 中
    }
    for(i = 0;i < m_pDoc -> n_GraphSelect;i++)     //对所有选中的矢量图形元素循环
    {
        m_str.Empty();
        sprintf(p1,"%7d",m_pDoc -> GraphSelect[i].pbh);//将第 i 个图形元素识别号转成字符串
        m_str += p1;                    //字符串 m_str 设置成矢量图形元素的识别号转成的字
                                        //符串
        m_str += "  ";                  //在字符串 m_str 后加几个空格
        m_str += graph_Name[m_pDoc -> GraphSelect[i].Lb - 1]; //在字符串 m_str 后加图形类
                                                              //别名称
        plist2 -> AddString(m_str);     //将字符串 m_str 增加到列表框 IDC_LIST2 中
    }
    sprintf(p1,"%d",m_LinkMode);        //将连接方式转成字符串
    SetDlgItemText(IDC_EDIT1,p1);       //将连接方式转成的字符串写到编辑框 IDC_EDIT1
    sprintf(p1,"%d",m_DatabaseID);      //将记录集合对象的识别号转成字符串
    SetDlgItemText(IDC_EDIT2,p1);       //将字符串写到编辑框 IDC_EDIT2
    sprintf(p1,"%d",m_RecordID);        //将记录或过滤条件式的识别号转成字符串
    SetDlgItemText(IDC_EDIT3,p1) ;      //将字符串写到编辑框 IDC_EDIT3
```

```
    if(m_NumbLink > 0)                    //如果存在连接的数目大于 0
        plist1 -> SetCurSel(0);           //在列表框 IDC_LIST1 中选择第 1 个选项
    return TRUE;
}
```

OnInitDialog 函数的主要操作步骤如下：

① 得到了指向列表框 IDC_LIST1 和 IDC_LIST2 的指针 plist1 和 plist2，清空这两个列表框中的项目。在列表框 IDC_LIST1 中列出了当前数据连接单元已经连接的矢量图形元素，在列表框 IDC_LIST2 中列出了在矢量图形中选中的图形元素。在这两个列表框用以下方式来显示一个矢量图形元素：对于一个唯一识别号为 121 的直线图形元素，将在列表框中以字符串“ 121　直线”来显示。

② 将数据连接单元的连接方式、记录集合对象的识别号、记录或过滤条件式的识别号转成字符串，分别填写到编辑框 IDC_EDIT1、IDC_EDIT2、IDC_EDIT3 中。

20.2.4　实现对话框类的数据交换机制

在对话框类 CLinkDlg 中，需要建立对话框中的各个编辑框与 m_pSet 指向的记录集合对象当前记录中各个字段的数据交换机制，从而实现利用对话框来交互编辑连接信息的功能。参考 15.3.4 小节中介绍的方法，利用 ClassWizard 将对话框中的五个编辑框分别与 m_pSet 指向的 CLinkDataSet 对象的当前记录的各个字段建立数据交换机制。增加了数据交换机制后的 DoDataExchange 函数实现代码如下：

```
void CLinkDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CLinkDlg)
    DDX_FieldText(pDX, IDC_EDIT1, m_pSet->m_link_mode, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT2, m_pSet->m_database_id, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT3, m_pSet->m_record_id, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT4, m_pSet->m_graph_id, m_pSet);
    DDX_FieldText(pDX, IDC_EDIT5, m_pSet->m_element_id, m_pSet);
    //}}AFX_DATA_MAP
}
```

在 DoDataExchange 函数中，增加了 5 个 DDX_FieldText 宏，通过这些宏可以完成对话框控制与记录集合对象当前记录的各个字段间的数据交换。关于交换机制实现过程的详细讨论，可参见 12.3.4 小节。

20.2.5　实现对话框类的各操作功能

本小节将在对话框类 CLinkDlg 中增加消息处理函数，实现编辑连接信息的具体功能。

(1) 增加控制消息处理函数

对于对话框中的按钮控制“增加连接”、“全部增加”、“删除连接”、“全部删除”、“保存退

出"、"放弃退出"等(见图 20-2 所示的对话框资源),用 ClassWizard 在对话框类 CLinkDlg 中增加 BN_CLICKED 控制消息处理函数,对于列表框 IDC_LIST2,增加一个 SEL_CHANGE 控制消息处理函数。增加完毕,在对话框类 CLinkDlg 中可以找到以下消息处理函数的定义代码:

```
//{{AFX_MSG(CLinkDlg)
virtual BOOL OnInitDialog();
afx_msg void OnLinkAdd();
afx_msg void OnLinkDelete();
afx_msg void OnLinkDelAll();
afx_msg void OnLinkAddAll();
virtual void OnOK();
virtual void OnCancel();
afx_msg void OnSelchangeList2();
//}}AFX_MSG
```

在对话框类 CLinkDlg 的实现文件 linkdlg.cpp 中,增加了消息映射机制:

```
BEGIN_MESSAGE_MAP(CLinkDlg, CDialog)
    //{{AFX_MSG_MAP(CLinkDlg)
    ON_BN_CLICKED(IDC_LINK_ADD, OnLinkAdd)
    ON_BN_CLICKED(IDC_LINK_ADDALL, OnLinkAddAll)
    ON_BN_CLICKED(IDC_LINK_DELETE, OnLinkDelete)
    ON_BN_CLICKED(IDC_LINK_DELALL, OnLinkDelAll)
    ON_LBN_SELCHANGE(IDC_LIST2, OnSelchangeList2)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 linkdlg.cpp 中,增加了空的消息处理函数。下面将对这些消息处理函数进行修改,来实现具体的操作功能。

(2) 增加与一个矢量图形元素的连接

在对话框中,当在列表框 IDC_LIST2 中选择一个矢量图形元素,并按中"增加连接"按钮时,如果当前数据连接单元中不存在与这个图形元素的连接,就会在当前数据连接单元和这个矢量图形元素间建立连接关系,新增的连接被增加到列表框 IDC_LIST1 中。为了实现这个操作功能,对"增加连接"按钮的消息处理函数 OnLinkAdd 修改如下:

```
void CLinkDlg::OnLinkAdd()
{
    CString m_str;
    char p1[8];
    GetDlgItemText(IDC_EDIT4,m_str); //从编辑框 IDC_EDIT4 中读出内容到字符串 m_str 中
    m_str.TrimLeft();                //去掉字符串 m_str 左边的空格
    if(m_str.GetLength() < 1)        //如果字符串长度小于 1,即没有在 IDC_EDIT4 中填入
                                     //图形类别识别号,函数退出
```

```
        return;
    int lb = atoi(m_str);                    //得到图形类别识别号
    m_str.Empty();
    GetDlgItemText(IDC_EDIT5,m_str); //从编辑框控制 IDC_EDIT5 中读出内容到字符串 m_str 中
    m_str.TrimLeft();                        //去掉字符串 m_str 左边的空格
    if(m_str.GetLength() < 1)                //如果字符串长度小于 1,即没有在 IDC_EDIT5 中填入
                                             //图形元素识别号,函数退出
        return;
    int id = atoi(m_str);                    //得到图形元素识别号
    if(! IsExistLink(lb,id))                 //如果这个图形元素与当前数据连接单元间不存在连接
    {
        m_GraphID[m_NumbLink] = id;//记录当前数据连接单元增加连接的图形元素识别号
        m_GraphLB[m_NumbLink++] = lb; //记录当前数据连接单元增加连接的图形类别识别号
        sprintf(p1,"%7d",id);                //将图形元素识别号转成字符串
        m_str = p1;                          //字符串 m_str 设置成图形元素识别号转成的字符串
        m_str += "";                         //在字符串后增加几个空格
        m_str += graph_Name[lb-1];//在字符串后增加图形类别的名称
        plist1 -> AddString(m_str);      //在列表框 IDC_LIST1 中增加一个项目,表示新增的连接
        plist1 -> SetCurSel(m_NumbLink-1); //在列表框 IDC_LIST1 中选择新增的项目
    }
}
```

OnLinkAdd 函数的主要操作步骤如下：

① 从编辑框 IDC_EDIT4 中读出内容到字符串 m_str，然后将字符串转换成图形类别号 lb；从编辑框 IDC_EDIT5 中读出内容到字符串 m_str，然后将字符串转换成图形元素唯一识别号 id。如果编辑框 IDC_EDIT4 或 IDC_EDIT5 中的内容为空，退出函数，结束增加连接操作。

② 调用成员函数 IsExistLink，检查当前数据连接单元与选择的图形元素间是否已经存在连接，如果不存在这个连接，用数组 m_GraphLB 和 m_GraphID 记录下要连接的矢量图形元素的图形类别号和唯一识别号，连接的数目 m_NumbLink 加 1，并在列表框 IDC_LIST1 中增加一个项目来表示这个连接。

函数 IsExistLink 用来检查当前数据连接单元和选择的图形元素间是否存在连接，函数在对话框类 CLinkDlg 中定义如下：

```
BOOL IsExistLink(short lb, int id);
```

IsExistLink 函数的参数 lb 和 id 分别是要连接的矢量图形元素的图形类别号和唯一识别号，当当前数据连接单元存在这个连接时，函数返回 TRUE，否则，函数返回 FALSE。完成后的函数实现代码如下：

```
BOOL CLinkDlg::IsExistLink(short lb, int id)
{
    for(int i=0;i<m_NumbLink;i++)        //对当前数据连接单元连接的所有矢量图形元素循环
    {
        if(m_GraphID[i] == id&&m_GraphLB[i] == lb) //如果第 i 个图形元素就是要连接的图形
```

```
                                        //元素
            return TRUE;                //函数返回 TURE,表示已经与这个图形元素建立了连接
        }
        return FALSE;                   //函数返回 FALSE,表示还没有与这个图形元素建立连接
}
```

(3) 增加与全部选中图形元素的连接

当按中对话框中的"增加全部"按钮时,就会把在列表框 IDC_LIST2 中列出的所有矢量图形元素(在矢量图形中选中的图形元素)与当前数据连接单元建立连接。为了实现这个操作功能,将"增加全部"按钮的消息处理函数 OnLinkAddAll 修改如下:

```
void CLinkDlg::OnLinkAddAll()
{
    for(int i=0;i<m_pDoc->n_GraphSelect;i++)  //对所有选中的矢量图形元素循环
    {
        plist2->SetCurSel(i);   //在列表框 IDC_LIST2 中选择第 i 个图形元素
        OnSelchangeList2();     //调用 OnSelchangList2 函数将列表框 IDC_LIST2 中当前选择的矢
                                //量图形元素的参数填写到编辑框 IDC_EDIT4 和 IDC_EDIT5 中
        OnLinkAdd();            //调用 OnLinkAdd 函数增加一个连接
    }
}
```

OnLinkAddAll 函数对所有选中的矢量图形元素循环,依次在列表框 IDC_LIST2 中选择要连接的图形元素,调用 OnSelchangeList2 将选择的图形元素的图形类别号和唯一识别号填写到编辑框 IDC_EDIT4 和 IDC_EDIT5 中,然后调用 OnLinkAdd 去增加与这个矢量图形元素的连接。

OnSelchangeList2 函数将列表框控制 IDC_LIST2 中选择的矢量图形元素的图形类别号和唯一识别号,填写到编辑控制框 IDC_EDIT4 和 IDC_EDIT5 中。修改后的函数实现代码如下:

```
void CLinkDlg::OnSelchangeList2()
{
    int nlist2=plist2->GetCurSel();  //得到在列表框 IDC_LIST2 中选项的 0 基序号
    if(nlist2<0)                     //如果没有在列表框 IDC_LIST2 中选择一个项目,函数退出
        return;
    char p1[10];
    //以下将文本框 IDC_STATCI1 的内容设置成当前选择的图形元素的类别名称
    SetDlgItemText(IDC_STATIC1,graph_Name[m_pDoc->GraphSelect[nlist2].Lb-1]);
    sprintf(p1,"%d",m_pDoc->GraphSelect[nlist2].Lb);  //将图形类别号转成字符串
    SetDlgItemText(IDC_EDIT4,p1);   //将图形类别号填写到编辑框 IDC_EDIT4 中
    sprintf(p1,"%d",m_pDoc->GraphSelect[nlist2].pbh);  //将图形元素识别号转成字符串
    SetDlgItemText(IDC_EDIT5,p1);   //将图形元素识别号填写到编辑框控制 IDC_EDIT5 中
}
```

(4) 删除一个与矢量图形元素的连接

在对话框的列表框 IDC _ LIST1 中,在选择当前数据连接单元连接的一个矢量图形元素后,按中"删除连接"按钮,就能够删除当前数据连接单元与选择的矢量图形元素的连接。为了实现这个功能,将"删除连接"按钮控制的消息处理函数 OnLinkDelete 修改如下:

```
void CLinkDlg::OnLinkDelete()
{
    int nlist1 = plist1 -> GetCurSel(); //得到在列表框 IDC _ LIST1 中选项的 0 基序号
    if(nlist1 < 0)                      //如果没有在列表框 IDC _ LIST1 中选择一个选项,函数退出
        return;
    //以下对数组 m _ GraphLB 和 m _ GraphID 滚动,删除第 nlist1 项
    for(int i = nlist1;i < m _ NumbLink - 1;i++)
    {
        m _ GraphLB[i] = m _ GraphLB[i + 1];
        m _ GraphID[i] = m _ GraphID[i + 1];
    }
    m _ NumbLink - -;                   //当前数据连接单元连接的矢量图形元素数目减 1
    plist1 -> DeleteString(nlist1);     //从列表框 IDC _ LIST1 中删除当前选择的项目
    if(nlist1 = = m _ NumbLink)         //如果从列表框 IDC _ LIST1 中删除的是最后一个项目
        plist1 -> SetCurSel(nlist1 - 1); //选择列表框 IDC _ LIST1 中的上一个项目
    else                                //如果从列表框 IDC _ LIST1 中删除的不是最后一个项目
        plist1 -> SetCurSel(nlist1);    //选择列表框 IDC _ LIST1 中原来序号的项目
}
```

OnLinkDelete 函数的主要操作步骤如下:

① 从列表框 IDC _ LIST1(在列表框中列出当前数据连接单元连接的矢量图形元素)中得到选项的 0 基序号 nlist1,通过对数组 m _ GraphID 和 m _ GraphLB 进行滚动操作,将数组的第 nlist1 项删除(即删除了第 nlist1 个连接的矢量图形元素),把当前数据连接单元具有的连接数目 m _ NumbLink 减 1。

② 从列表框 IDC _ LIST1 中删除第 nlist1 项,根据是否从列表框 IDC _ LIST1 中删除了最后一项,来调整列表框中的当前选项。

(5) 删除所有与矢量图形元素的连接

当按中对话框中的"全部删除"按钮时,就会将当前数据连接单元中所有与矢量图形元素的连接删除,也就是从列表框 IDC _ LIST1 中删除了所有选项。为了实现这个功能,将"全部删除"按钮的消息处理函数 OnLinkDelAll 修改如下:

```
void CLinkDlg::OnLinkDelAll()
{
    m _ NumbLink = 0;
    plist1 -> ResetContent();
```

```
}
```

OnLinkDelAll 函数的实现代码非常简单，只需要把当前数据连接单元具有的连接数目 m_NumbLink 设置为 0，并从列表框 IDC_LIST1 中删除所有项目。

(6) 保存对连接信息所做的修改并退出对话框

在以上实现的各个操作功能中，对当前数据连接单元的连接信息修改时，只是对数组 m_GraphID 和 m_GraphLB 的内容进行了修改，并没有对存储连接信息的记录集合的内容进行修改。如果在对话框中选择“保存退出”按钮，则首先需要对存储连接信息的记录集合进行修改，来完成各个操作对连接信息的修改，然后退出对话框。为了实现这个功能，将“保存退出”按钮的消息处理函数 OnOK 修改如下：

```
void CLinkDlg::OnOK()
{
    for(int i=m_NumbLink1-1;i>=0;i--)   //对数据连接单元修改前连接的所有图形元素逆
                                        //循环
    {
        if(! IsExistLink(m_GraphLB1[i],m_GraphID1[i]))
        //如果原来与某个图形元素存在连接，而操作后这个连接不存在，即删除了这个连接
        {
            m_pSet->Move(i+1,SQL_FETCH_ABSOLUTE); //将记录指针指向记录集合的第 i
                                                  //条记录
            m_pSet->Delete();           //删除当前记录
            m_pSet->Close();            //关闭记录集合对象
            m_pSet->Open();             //重新打开记录集合对象
        }
    }

    for(i=0;i<m_NumbLink;i++)           //对修改后当前连接单元连接的所有图形元素循环
    {
        if(! IsExistLink1(m_GraphLB[i],m_GraphID[i]))
        //如果连接的这个图形元素原来不存在，即新增了与这个图形元素的连接
        {
            plist1->SetCurSel(i);   //在列表框 IDC_LIST1 中选择第 i 个选项
            OnSelchangeList1();     //调用 OnSelchangList1 函数把当前选择的矢量图形元素的图形
                                    //类别和识别号分别填写到编辑框 IDC_EDIT4 和 IDC_EDIT5 中
            m_pSet->AddNew();   //在记录集合中增加一条新记录
            UpdateData();           //以 TURE 为参数(默认参数)调用 UpdateData 函数，存储各个编
                                    //编辑框中的内容到记录集合新增记录的各个字段中
            if(m_pSet->CanUpdate()) //如果能够对数据源进行更新
                m_pSet->Update();  //更新数据源中的数据表
        }
    }
```

```
    m_pSet->Close();                    //关闭记录集合对象
    EndDialog(IDOK);                    //退出对话框
}
```

OnOK 函数的主要操作步骤如下：

① 把当前数据连接单元修改前连接的矢量图形元素(图形类别和识别号记录在数组 m_GraphLB1 和 m_GraphID1 中)分别与修改后连接的矢量图形元素(图形类别和识别号记录在数组 m_GraphLB 和 m_GraphID 中)使用函数 IsExistLink 进行比较。

② 如果某个修改前与数据连接单元连接的矢量图形元素，在修改后不再与数据连接单元连接(即在操作过程中删除了这个连接)，则在记录集合中移动记录指针到这条记录，并删除这条记录。为了保证记录定位操作的正确性，从后向前删除记录集合中需要删除的记录。

③ 为了保证正确更新数据源中的记录表，每次删除操作后都关闭并重新打开了记录集合对象。

④ 把当前数据连接单元修改后连接的所有矢量图形元素(图形类别和识别号记录在数组 m_GraphLB 和 m_GraphID 中)，分别与修改前连接的矢量图形元素(图形类别和识别号记录在数组 m_GraphLB1 和 m_GraphID1 中)利用函数 IsExistLink1 进行比较。

⑤ 如果某个矢量图形元素在修改前没有与数据连接单元连接，即在操作过程中增加与这个矢量图形元素的连接，则通过调用 CRecordSet 类的成员函数 AddNew 给 m_pSet 指向的记录集合对象增加一条记录。

⑥ 调用 OnSelchangeList1 函数，把要连接的矢量图形元素的图形类别号和唯一识别号填写到编辑框 IDC_EDIT4 和 IDC_EDIT5 中，然后以参数 TRUE(默认参数)调用 UpdateData 函数，通过 UpdateData 函数调用 DoDataExchange 函数，保存各个编辑框中的内容到记录集合的当前记录中。

⑦ 通过 Update 函数把对记录集合的修改反映到数据源，实现对数据表内容的实际修改。

⑧ 关闭 m_pSet 指向的记录集合对象，退出对话框。

在 OnOK 函数中，调用了两个成员函数 IsExistLink1 和 OnSelchangeList1，这两个函数在对话框类 CLinkDlg 中定义如下：

```
protected:
    BOOL IsExistLink1(short lb,int id);
    void OnSelchangeList1();
```

IsExistLink1 函数用来检查一个矢量图形元素在连接信息被修改前是否与当前数据连接单元连接。参数 lb 和 id 用来记录要检查的矢量图形元素的图形类别号和唯一识别号，当这个矢量图形元素与数据连接单元的连接时，函数返回 TRUE，否则，函数返回 FALSE。函数的实现代码如下：

```
BOOL CLinkDlg::IsExistLink1(short lb, int id)
{
    for(int i=0;i<m_NumbLink1;i++)  //对当前数据连接单元修改前连接的所有图形元素循环
    {
        if(m_GraphID1[i]==id&&m_GraphLB1[i]==lb)//如第 i 个图形元素与要检查的图形元
                                                //素相同
```

```
            return TRUE;                                    //返回 TRUE
        }
        return FALSE;
    }
```

OnSelchangeList1 函数用来把列表框 IDC_LIST1 中当前选项的图形类别号和唯一识别号，分别填写到编辑框 IDC_EDIT4 和 IDC_EDIT5 中。函数的实现代码如下：

```
void CLinkDlg::OnSelchangeList1()
{
    int nlist1 = plist1 -> GetCurSel();          //得到在列表框 IDC_LIST1 中选项的 0 基序号
    if(nlist1 < 0)                  //如果没有在列表框 IDC_LIST1 中选择一个图形元素,函数退出
        return;
    char p1[10];
    //以下设置文本框 IDC_STATIC1 的内容为选择图形元素的图形类别名称
    SetDlgItemText(IDC_STATIC1,graph_Name[m_GraphLB[nlist1] - 1]);
    sprintf(p1,"%d",m_GraphLB[nlist1]);  //将选择的图形元素的类别号转成字符串
    SetDlgItemText(IDC_EDIT4,p1);          //在编辑框 IDC_EDIT4 中填入图形元素类别号
    sprintf(p1,"%d",m_GraphID[nlist1]);  //将选择的图形元素的唯一识别号转成字符串
    SetDlgItemText(IDC_EDIT5,p1);          //在编辑框 IDC_EDIT4 中填入图形元素的唯一识别号
}
```

(7) 放弃对连接信息所做的修改并退出对话框

当选择运行对话框的“放弃退出”按钮时，系统会放弃在对话框中对当前数据连接单元连接信息的修改，退出对话框。为了实现这个功能，对“放弃退出”按钮控制的消息处理函数 OnCancle 修改如下：

```
void CLinkDlg::OnCancel()
{
    m_pSet -> Close(); //关闭记录集合对象
    CDialog::OnCancel();
}
```

在 OnCancel 函数中，关闭 m_pSet 指向的记录集合对象并退出对话框。在 OnCancel 中没有具体修改 m_pSet 指向的记录集合来修改连接信息，也就放弃了对话框中对连接信息的修改操作。

20.3　实现矢量图形系统和数据库管理系统的连接

在 20.2 节中，讨论实现了用来交互管理连接信息的对话框类 CLinkDlg，本节将在数据库浏览视图中利用对话框类 CLinkDlg，完成矢量图形系统和数据库管理系统间的连接功能。

20.3.1　实现连接功能

对话框类 CLinkDlg 用来管理一个数据连接单元所连接的图形连接单元(矢量图形元素)，

也就是说,是在确定了一个数据连接单元的情况下,对与这个数据连接单元连接的矢量图形元素进行管理。根据 20.1 节的介绍,有多种不同种类的数据连接单元,本节只对数据表的一条记录作为一个连接单元这种连接方式下的连接操作进行讨论。

在数据库浏览视图中实现连接功能的具体步骤如下:在数据库浏览视图的操作界面上增加一个菜单项,运行此菜单项时,将把数据库浏览视图中当前选中的记录作为当前的数据连接单元,利用对话框类 CLinkDlg 对象组织对话框,对这个数据连接单元的连接信息进行管理。

(1) 增加菜单资源

在数据库浏览视图所采用的菜单资源 IDR_DRAWDATA 中进行修改,增加一个菜单项"图形连接",为了操作方便,把菜单项增加在"记录操作"菜单中(参见 15.2.1 小节)。增加菜单项后,在资源文件 draw.rc 中增加了菜单项的定义:

```
IDR_DRAWDATA MENU PRELOAD DISCARDABLE
BEGIN
    ......//省略以上代码
    POPUP "记录操作"
    BEGIN
        MENUITEM "增加记录...", ID_RECORD_ADD
        MENUITEM "修改记录...", ID_RECORD_MODIFY
        MENUITEM "删除记录...", ID_RECORD_DELETE
        MENUITEM "删除所有记录", ID_RECORD_DELETE_ALL
        MENUITEM "图形连接...", ID_RECORD_LINK
    END
    ......//省略以下代码
END
```

菜单项的 ID 号在资源 ID 定义文件 resource.h 中定义如下:

```
# define ID_RECORD_LINK32796
```

(2) 增加和修改消息处理函数

针对增加的菜单项"图形连接...",用 ClassWizard 在视图类 CDatabaseView 中为其增加一个消息处理函数 OnRecordLink。操作完毕后,在视图类 CDatabaseView 中增加了函数的定义:

```
//{{AFX_MSG(CDatabaseView)
......//省略以上代码
afx_msg void OnRecordLink();
//}}AFX_MSG
```

在实现文件 dbaseview.cpp 中,增加了映射机制:

```
BEGIN_MESSAGE_MAP(CDatabaseView, CScrollView)
    //{{AFX_MSG_MAP(CDatabaseView)
```

```
    ......//省略以上代码
    ON_COMMAND(ID_RECORD_LINK, OnRecordLink)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 dbaseview.cpp 中，增加函数空的实现代码。对函数实现代码进行修改，得到修改后的函数实现代码如下：

```
void CDatabaseView::OnRecordLink()
{
    CDrawDoc * pDoc = GetDocument();                    //得到指向当前文档对象的指针
    m_pSet->Move(nIndex, SQL_FETCH_ABSOLUTE); //移动记录指针到记录集合的第 nIndex
                                                        //条记录
    CString m_str;
    //以下动态组织一个过滤条件式
    m_str = "(database_id = ";
    short lb = pDoc->GetDataID(nSelectData);            //得到当前记录集合对象的识别号
    if(lb < 0)                                          //如果记录集合对象识别号小于 0,函数
                                                        //退出
        return;
    char p1[9];
    sprintf(p1,"%d",lb);                                //将记录集合对象识别号转成字符串
    m_str += p1;
    m_str += ") AND (record_id = ";
    int ID = m_pSet->GetID();                           //得到当前记录的唯一识别号
    sprintf(p1,"%d",ID);                                //将记录唯一识别号转成字符串
    m_str += p1;
    m_str += ")";
    m_pLinkSet1->m_strFilter = m_str;                   //设置记录集合对象的过滤条件式
    CLinkDlg pDlg1(GetDocument(),m_pLinkSet1,1,lb,ID); //定义一个对话框类 CLinkDlg 对象
    pDlg1.DoModal();                                    //调用连接信息管理对话框
}
```

OnRecordLink 函数的实现步骤如下：

① 根据数据库浏览视图中的记录指针序号 nIndex，把 m_pSet 指向的记录集合对象的记录指针移动到第 nIndex 条记录。

② 通过调用文档类的 GetDataID 函数，得到浏览视图中当前记录集合对象的识别号 lb，调用函数 GetID 得到当前记录的唯一识别号 ID，通过记录集合对象的识别号和当前记录的唯一识别号动态产生一个过滤条件。例如，当 lb = 1 和 ID = 40 时，会产生一个如下的过滤条件：

```
"(database_id = 1) AND (record_id = 40)"
```

将 m_pLinkSet1 指向的记录集合对象（存储连接信息的记录集合对象）的过滤子式设置成过滤条件 m_str。

③ 定义一个对话框类 CLinkDlg 对象 pDg1。定义 pDlg1 对象时，第一个参数指向当前文档

对象的指针,第二个参数指向存储连接信息的记录集合对象的指针,第三个参数是连接方式,第四个参数是浏览视图中当前记录集合对象的类别号,第五个参数是当前记录的唯一识别号。通过对话框类对象 pDlg1 调用连接信息对话框,来管理数据库浏览视图中当前选中记录(一个数据连接单元)的连接信息。

20.3.2 完善连接功能

本章讨论实现了数据表的一条记录作为一个连接单元的连接方式下,交互管理记录集合对象一条记录的连接信息。在实际的 GIS 系统中,一般还需要其他几种连接操作功能。

(1) 数据表的一条记录集合作为一个连接单元来连接矢量图形元素

当数据表的一个记录集合作为一个数据连接单元时,需要对这个数据连接单元也能够提供像数据表的一条记录那样的连接信息管理功能。为了实现这个功能,首先应该对记录集合基类 CDataBaseSet 的功能进行扩充,使之能够管理多个过滤条件(通过这些过滤条件过滤得到用作数据连接单元的记录集合)。在 18.2.5 小节中介绍了管理多个过滤条件式的思路。对于数据表的一个记录集合形成的数据连接单元,也通过两个参数来描述:记录集合对象的识别号和用来产生记录集合的过滤条件式的唯一识别号,只要能够在数据库浏览视图中通过一定的方式来确定一个数据连接单元,就同样可以利用对话框类 CLinkDlg 对象来管理与这个数据连接单元连接的矢量图形元素(仅仅连接方式与数据表的一条记录作为一个连接单元时不同)。

(2) 从矢量图形元素去连接数据连接单元

本章讨论实现的连接操作,需要首先确定数据连接单元,然后对这个数据连接单元的连接信息进行管理。通过这个操作,可以建立起任何数据连接单元和图形连接单元间的连接操作,也就是说可以完成任何连接任务。但是,这种单一的连接操作方法操作并不方便,在很多情况下,希望能够从矢量图形元素的角度出发,即首先确定图形连接单元,然后对这个图形连接单元的连接信息进行管理(对于这个图形连接单元,提供增加、删除数据连接单元的操作功能)。为了完成这个操作功能,需要建立另外一个对话框类,用这个对话框类提供管理一个矢量图形元素(图形连接单元)所连接的数据连接单元,即在启动这个对话框时,应首先确定图形连接单元(即确定需要对哪个图形元素进行管理),在对话框内能够完成诸如选择记录集合对象、选择记录集合的记录或组成记录集合的过滤条件式等操作,来增加与这个图形元素连接的各个数据连接单元。

这样,就可以从另外一个角度来实现连接操作。实际上,任何一个连接操作都可以通过两种连接对话框分别来完成,只不过在不同需要的连接操作中,两种对话框提供连接操作的方便性不同。

读者可以参照对话框类 CLinkDlg 的实现方法,实现另外一个对话框类来管理一个图形连接单元(一个矢量图形元素)的连接信息,并在应用程序 Draw 的矢量图形视图中利用这个对话框类来组织实现具体的连接操作。

20.4 实现实体

本节讨论实体的概念和在 GIS 系统中组织实现实体的方法。

20.4.1 实体的概念

这里所指的实体,是矢量图形元素与性质数据信息结合在一起的整体。以下用一个具体的例子来说明实体的概念。当在一个全国城市图上组织全国城市信息管理系统,利用矢量图形系统和数据库系统相连接实现 GIS 系统时,需要以下的步骤:

① 在矢量图形中绘制表示各个城市的图形元素,形成一幅包含代表各个城市图形标志的矢量图形。

② 建立起用来存储城市信息的数据表,数据表的每条记录用来存储一个城市的信息,建立起管理这个数据表的记录集合对象,将各个城市的信息通过交互操作界面输入到数据表中。

③ 通过连接操作,将矢量图形中表示城市的矢量图形元素与数据表中对应的记录建立连接关系。

上述这个操作方法存在着以下突出的问题:

① 操作烦琐。从以上的操作步骤来看,其操作烦琐性是显而易见的,完成一个城市点需要经过几个步骤的操作,当有大量城市的情况下,建立连接是一种很麻烦的工作。

② 容易出错,可操作性差。因为建立城市点的工作是经过几个步骤交互式实现的,不可避免的会造成连接错误。作为一个实用系统来说,可操作性肯定不能满足要求。

③ 维护困难。因为对矢量图形元素、数据表的记录以及两者的连接是分别维护的,很容易造成错误连接。例如,造成数据表的记录与矢量图形元素不对应,将北京市的信息连接到了表示广州的矢量图形元素上;又例如,对于有的矢量图形元素连接了多个记录,或者没有连接记录。

为了解决这些问题,可以建立起一个特殊的整体即实体。实体一方面确定了其图形信息,即这个实体是由哪种图形元素组成的,另一方面确定了要连接的性质数据信息,即要连接哪个记录集合对象、采用哪种连接方式。

在系统中增加对实体的管理功能,将矢量图形元素和性质数据信息作为一个整体来处理。例如,在系统中建立一个叫"城市"的实体,增加一个"城市"实体时,在矢量图形上增加城市的图形信息(如一个图例)的同时,在对应的存储性质数据信息的数据表中增加一条记录,同时建立起增加的矢量图形元素与增加的记录之间的连接。同样,在删除一个"城市"实体时,将从矢量图形中删除其对应的图形元素,从数据表中删除其对应的记录,并从存储连接信息的数据表中删除两者间的连接。

利用实体可以较容易地组织一个 GIS 系统。

20.4.2 实体的数据结构

本小节对描述实体的数据结构和管理实体的类进行讨论。

(1) 实体结构

在应用程序中建立一个数据结构来描述一个实体的组成,在数据结构中对组成一个实体的矢量图形和性质数据信息两方面的组成参数进行描述。以下的数据结构可用来描述一个实体:

```
typedef struct
{
    char Name[20];              //实体的名称,作为管理实体时采用,可为任意字符串
    short only_id;              //实体的唯一识别号
    short graph_lb;             //图形元素的类别
    short m_ColorPen;           //笔色
    short m_ColorBrush;         //填充刷颜色
    short m_LineWide;           //线宽(像素)
    short m_LineType;           //线型(像素)
    short m_Layer;              //所处图层
    short tag_id;               //如果是图例的话,为图形块的唯一识别号
    float bl1;                  //如果图形类别是图例的话,为横向比例尺
    float bl2;                  //如果图形类别是图例的话,为纵向比例尺
    short LinkNumb;             //连接的性质数据库的数目
    short link_mode[6];         //连接的方式:1-数据表的一条记录,2-数据表的一记录集合
    short int database_id[6];   //连接的记录集合对象的识别号
} ObjectStruct;
```

字符串 Name 用来存储标识这个实体的名称,如可以给一个实体取名"城市点"。

graph_lb 用来表示实体所属的图形元素类别,对于不同的图形类别,可以在结构中定义不同的图形参数。当图形类别是本书讨论的一些基本图形元素时(如直线、圆等),成员变量 m_ColorPen、m_ColorBrush、m_LineWide、m_LineType 等用来记录这种类别图形元素的参数;当图形元素的类别是图例时,成员变量 bl1 和 bl2 用来确定将图形块插入到矢量图形中时横向和纵向比例(即当把图形块作为实体用的图形元素时,需要确定图形块的实际大小);成员变量 m_Layer 表示实体的图形元素在矢量图形系统中的图层号。以上这些参数描述了组成实体的图形元素参数。

LinkNumb 用来表示需要连接到矢量图形元素上的数据连接单元的数目,数组 link_mode 用来记录各个连接的连接方式,数组 database_id 用来记录各个连接需要连接的记录集合对象的识别号。在实体中,一般采用数据表的一条记录作为一个连接单元的连接方式(也可采用数据表的一个记录集合作为一个连接单元的连接方式)。

在应用程序中,建立一个 ObjectStruct 结构,并对结构中的成员变量初始化就可以建立起一个实体结构,可以通过建立 ObjectStruct 结构数组来管理多个实体结构。在应用程序中提供操作界面,来交互式的完成对实体结构的增加、删除等管理功能。

(2) 实体类

在应用程序中,与建立一个 CLine 类来管理直线类似,为了管理实体,可以建立一个如下

的类：

```
class CObject1 : public CObject
{
public:
    COob();
protected:
    ObjectStruct * pObject;    //指向当前实体属于的实体结构
    short object_id;           //当前实体属于的实体结构的识别号
    int graph_element_id;      //实体的矢量图形元素的唯一识别号
    int record_id[6];          //实体连接的数据连接单元的识别号(记录或过滤条件式的识别号)
}
```

一个 CObject1 类对象用来管理一个实体，类中的成员变量 pObject 指向当前实体所属于的实体结构，object_id 是实体所属的实体结构中的唯一识别号 only_id，用来永久与实体结构建立联系用。graph_element_id 是实体属于的矢量图形元素的识别号，数组 record_id 记录的是实体所连接的数据连接单元的识别号。这两个成员变量与实体属于的 ObjectStruct 结构相结合，就完整的描述出了一个实体的矢量图形元素和性质数据组成。

与基本图形元素类如直线类 CLine 类似，通过建立一个 CObject1 对象来管理一个实体，可以在文档类中建立一个 CObArray 对象来管理多个实体的对象指针，实现对各个实体的统计、分析等各种操作功能。

20.4.3　有关实体的操作

本小节讨论与实体有关的主要操作的实现思路。

(1) 增加实体的步骤

在应用程序中增加一个实体时，通过实体结构的描述参数，去进行以下步骤的操作：

① 根据实体结构所描述的图形元素类别，在矢量图形系统中增加一个图形元素，并得到图形元素的识别号。

② 根据实体结构所描述的数据连接单元，在指定的一个或多个记录集合对象中，增加一个数据连接单元(记录或过滤条件式)，得到数据连接单元的识别号，并建立与新增矢量图形元素的连接。

③ 创建一个新的实体对象(CObject1 类对象)，用新增矢量图形元素和数据连接单元的识别号来初始化这个实体对象的成员变量，并把新增实体对象的指针增加到文档对象中记录实体对象指针的 CObArray 对象中。

(2) 交互式绘制实体

在应用程序中交互绘制实体时，其操作步骤与交互绘制一个普通的图形元素类似，在绘制过程中，需要完成如下的操作步骤：

① 选择一个实体结构。在系统中可以有多个实体结构，在绘制一个实体前，首先应通过

系统提供的交互操作界面选择要绘制的实体属于的实体结构。

② 根据选择的实体结构得到要绘制的图形元素的类别,然后调用系统既有的交互绘制图形元素的功能进行交互绘制操作。例如,实体结构中设置的图形类别是直线时,当进行实体绘制操作时,就会在图形屏幕上进行直线的绘制操作。图形元素绘制完毕后,得到这个图形元素的识别号。

③ 根据实体结构所描述的数据连接单元,在指定的一个或多个记录集合对象中,增加一个数据连接单元(记录或过滤条件式),得到数据连接单元的识别号,并建立与新增的矢量图形元素的连接。

④ 创建一个新的实体对象(CObject1 类对象),用新增矢量图形元素和数据连接单元的识别号来初始化实体对象的成员变量,并把新增实体对象的指针增加到文档对象中记录实体对象指针的 CObArray 对象中。

(3) 删除实体

当选择一个实体进行删除时(可以对矢量图形系统的点选操作进行修改,实现能够通过得到实体在 CObArray 对象中的序号来选择实体),系统需要进行以下步骤的操作:

① 根据实体对象和这个实体属于的实体结构,得到并删除组成实体的图形元素。

② 根据删除的图形元素具有的连接,来从对应的记录集合对象中删除与矢量图形元素连接的数据连接单元,并从存储连接信息的记录集合对象中删除存储这些连接的记录。

③ 从存储实体对象指针的 CObArray 对象中删除这个实体对象的指针,并销毁要删除的实体对象。

以上是对一个实体进行不可逆删除的操作步骤。对应用程序的删除操作和恢复操作功能进行修改,可以实现删除实体操作的逆操作。

(4) 实现系统的统一性

实体实际上是一个建立在矢量图形系统、数据库系统以及两者之间的连接系统之上的一个抽象整体,通过这个抽象整体可以完成 GIS 系统中的各种统计分析工作,实体的完整性不容破坏。而如果不对应用程序的各个操作功能进行限制,就很容易破坏实体的完整性。以下是保持实体完整性的一些措施。

① 在图形元素基类 CDraw 和存储记录集合对象信息的结构 DataInfoStru 中增加一个成员变量,来识别一个矢量图形元素或一个记录集合对象是否用于实体。如可以在 CDraw 类和 DataInfoStru 中增加如下的成员变量:

```
BOOL b_IsObject;
```

b_IsObject 为 TURE 时,表示这个矢量图形元素(或记录集合对象)是用来实现实体的;当 b_IsObject 为 FALSE 时,表示这个矢量图形元素(或记录集合对象)不是用来实现实体的。

② 修改系统的操作功能,对于用于实体的矢量图形元素,不能再用交互式连接数据连接单元(连接数据连接单元的操作见本章的以上各节)。

③ 对于用于实体的记录集合对象,不能对它的数据连接单元的连接信息进行修改;取消对记录集合对象的增加记录和删除记录操作功能(允许实现记录修改操作);取消对记录集合

对象中用于实现数据连接单元的过滤条件式的增加和删除操作。

本节讨论了实体的概念和在应用程序中实现实体的方法和思路，不再对实现实体的具体操作步骤和实现代码进行讨论。读者可参考作者提供的方法和思路独立编程实现，也可参考作者提供的包含此操作功能的程序源代码(代码 Demo 见附录和随书 CD－ROM)。

第 21 章　实现双向信息查询

通过在图形连接单元和数据连接单元间建立连接，可以将存储在数据表中的性质数据信息赋予各矢量图形元素，实现基本 GIS 系统。通过连接可以在图形连接单元和数据连接单元间实现双向信息查询。所谓双向查询，是指通过数据连接单元可以得到与之相连接的图形连接单元(矢量图形元素)，通过图形连接单元(矢量图形元素)同样可以得到与其连接的各个数据连接单元。

在 20.3 节实现连接操作时，在调用对话框类 CLinkDlg 对象进行连接操作前，得到了与当前数据连接单元相连接的矢量图形元素的图形类别号和唯一识别号。只要得到了与数据连接单元相联系的图形元素的图形类别号和唯一识别号，就可以实现对图形元素进行任意的统计操作。所以，在本章将主要讨论通过矢量图形元素查找性质数据信息(连接的数据连接单元)的操作，在应用程序 Draw 中提供一个对话框界面，实现查询和修改矢量图形元素连接的性质数据信息。

21.1　建立显示和编辑数据信息的对话框类

在应用程序 Draw 中，当用鼠标在矢量图形中点中一个图形元素时，能够通过交互界面得到这个图形元素所连接的数据连接单元，并提供对数据连接单元的修改功能。本节将建立一个对话框类，用来管理一个矢量图形元素所连接的数据连接单元。

(1) 建立对话框资源

为了建立一个管理矢量图形元素连接的数据连接单元的对话框类，需要首先建立一个对话框资源。用 RecourceView 工具及其可视化编辑工具 AppStudio 新建并可视化编辑一个对话框资源，得到一个如图 21-1 所示的对话框资源 IDD _ DATA _ LINK。

图 21-1　IDD _ DATA _ LINK 对话框资源

在资源文件 draw.rc 中，可以找到对话框资源 IDD _ DATA _ LINK 的描述代码：

```
IDD_DATA_LINK DIALOG DISCARDABLE 0, 0, 201, 119
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "图形元素的性质数据信息管理窗口"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON   "退出",IDOK,132,78,50,14
    LISTBOX         IDC_LIST1,10,23,105,75,LBS_NOINTEGRALHEIGHT | WS_VSCROLL |WS_
                    TABSTOP
    LTEXT           "连接的性质数据",IDC_STATIC,32,10,60,12
    PUSHBUTTON      "编辑数据",IDC_DATA_MODIFY,132,52,49,16
    PUSHBUTTON      "删除连接",IDC_LINK_DELETE,132,27,49,16
END
```

资源中的 ID 号在 ID 号定义文件 resource.h 中定义。

(2) 创建对话框类

按照 6.4.4 小节介绍的步骤，在应用程序 Draw 中增加对话框类 CDataLinkDlg，对话框类使用 IDD _ DATA _ LINK 对话框资源，头文件在 dlinkdlg.h，实现文件在 dlinkdlg.cpp。在头文件中 CDataLinkDlg 类定义如下：

```
class CDataLinkDlg : public CDialog
{
// Construction
public:
    CDataLinkDlg(CWnd* pParent = NULL); // standard constructor
// Dialog Data
    //{{AFX_DATA(CDataLinkDlg)
    enum { IDD = IDD_DATA_LINK };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDataLinkDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(CDataLinkDlg)
    //}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
};
```

对话框类 CDataLinkDlg 的实现文件 dlinkdlg.cpp 代码如下：

```
#include "stdafx.h"
#include "draw.h"
#include "datadlg.h"
#include "DataLinkDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

CDataLinkDlg::CDataLinkDlg(CWnd* pParent /* = NULL */)
    : CDialog(CDataLinkDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDataLinkDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CDataLinkDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDataLinkDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CDataLinkDlg, CDialog)
    //{{AFX_MSG_MAP(CDataLinkDlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////
// CDataLinkDlg message handlers
```

在对话框类 CDataLinkDlg 基础上，增加具体的实现代码，完成如下的管理功能：当在矢量图形中点中一个图形元素时，如果这个图形元素连接了数据连接单元，就会进入到对话框类 CDataLinkDlg 管理的对话框，在列表框 IDC_LIST1（“连接的数据性质”列表框）中列出连接的各个记录集合对象名称，在列表框 IDC_LIST1 中选择一个记录集合对象，按“修改数据”按钮，就会调用数据记录编辑对话框对选择的数据连接单元数据显示和编辑，按“删除连接”按钮，就会

删除当前图形元素与选择的数据连接单元之间的连接。

下面具体讨论实现上述的操作功能。

21.2　实现对话框类的操作功能

本节在 21.1 节创建的对话框类 CDataLinkDlg 基础上，完成对一个矢量图形元素存在连接的数据连接单元的显示和编辑功能。

21.2.1　对话框类的初始化

本小节在对话框类 CDataLinkDlg 中定义各个成员变量，并对对话框类的构造函数和初始化函数 OnInitDialog 进行修改，完成对话框类的初始化。

(1) 定义成员变量

为了完成对话框类 CDataLinkDlg 的各种管理功能，在对话框类 CDataLinkDlg 中定义以下各成员变量：

```
protected:
    CDrawDoc * m_pDoc;              //指向当前文档对象的指针
    CLinkDataSet * m_pLinkSet;      //指向存储连接信息的记录集合对象的指针
    CData1Set * m_pSet1;            //指向实际的记录集合对象的指针
    short m_NumbLink;               //连接的数据连接单元的数目
    short link_mode[20];            //存储各个连接的连接方式
    short database_id[20];          //存储各个连接的记录集合对象识别号
    int record_id[20];              //存储各个连接的记录识别号或过滤条件式识别号
```

m_NumbLink 用来记录当前图形元素所连接的数据连接单元的数目，对于每个数据连接单元，通过数组 link_mode、database_id、record_id 分别记录了连接方式、记录集合对象的识别号、数据记录或过滤条件式的识别号。

(2) 修改构造函数

为了利用构造函数的参数传入指向存储连接信息的记录集合对象和当前文档对象的指针，在头文件 dlinkdlg.cpp 中，将对话框类 CDataLinkDlg 的构造函数定义修改如下(也可以重载一个构造函数)：

```
CDataLinkDlg(CDrawDoc * pDoc,CLinkDataSet * pLinkSet,CWnd * pParent = NULL);
```

必要的话，在对话框类 CDataLinkDlg 的定义前加入对其他类的引用：

```
class CDrawDoc;
class CLinkDataSet;
```

在实现文件 dlinkdlg.cpp 中，修改构造函数的实现代码如下：

```
CDataLinkDlg::CDataLinkDlg(CDrawDoc * pDoc,CLinkDataSet * pLinkSet,CWnd * pParent /* = NULL
```

```
*/)
    : CDialog(CDataLinkDlg::IDD, pParent)
{
    m_pDoc = pDoc;              //保存指向当前文档对象的指针
    m_pLinkSet = pLinkSet;      //保存指向存储连接信息的记录集合对象的指针
    //{{AFX_DATA_INIT(CDataLinkDlg)
    // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}
```

(3) 增加和修改 OnInitDialog 函数

用 ClassWizard 在对话框类 CDataLinkDlg 中增加窗口消息 WM_INITDIALOG 的处理函数(实际上是在当前类中重载 CDialog 类中的虚函数 OnInitDialog)。操作完毕后,在 CDataLinkDlg 类的定义中增加了函数的定义代码:

```
//{{AFX_MSG(CDataLinkDlg)
virtual BOOL OnInitDialog();
//}}AFX_MSG
```

在实现文件 dlinkdlg.cpp 中,对 OnInitDialog 函数进行修改,得到函数的实现代码如下:

```
BOOL CDataLinkDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    CString m_str;
    char p1[9];
    int nIndex
    if(! m_pLinkSet->IsOpen())          //如果打开了存储连接信息的记录集合对象,则关闭它
        m_pLinkSet->Open();
    m_NumbLink = 0;                     //连接的数目等于 0
    if(! m_pLinkSet->IsBOF())           //如果在记录集合对象中存在记录
    {
        m_pLinkSet->MoveFirst();        //将记录指针移动到记录集合的第 1 条记录
        do              //对记录集合遍历,得到并记录当前图形元素所连接的各个数据连接单元
        {
            database_id[m_NumbLink] = m_pLinkSet->m_database_id; //记录集合对象的识别号
            link_mode[m_NumbLink] = m_pLinkSet->m_link_mode;  //连接方式
            record_id[m_NumbLink] = m_pLinkSet->m_record_id;    //记录数据连接单元的
                                                                //识别号
            m_NumbLink++;               //连接的数目加 1
            m_pLinkSet->MoveNext();     //指向记录集合的下一条记录
        }while(! m_pLinkSet->IsEOF());  //直到指向记录集合的最后一条记录后
    }
```

```
    CListBox * plist1 = (CListBox * )GetDlgItem(IDC_LIST1); //得到指向列表框 IDC_LIST1 的指针
    plist1 -> ResetContent();                    //清空列表框 IDC_LIST1
    for(int i = 0;i < m_NumbLink;i++ )           //对当前图形元素连接的所有数据连接单元循环
    {
        nIndex = m_pDoc -> GetDataIndex(database_id[i]);//得到第 i 个连接的记录集合对象的序号
        if(nIndex < 0)  //如果记录集合对象序号小于 0,即在文档对象中不存在这个记录集合对象
        {
            AfxMessageBox("不能找到记录集合对象");
            return;
        }
        m_str = m_pDoc -> DataInfo[nIndex].Name; //得到记录集合对象的名称
        sprintf(p1,"%d",record_id[i]);           //将数据连接单元的记录(或过滤条件式子)
                                                 //的识别号转成字符串
        m_str + = " ";                           //在字符串后增加几个空格
        m_str + = p1;                            //在字符串后增加记录(或过滤条件式)的识别号
        plist1 -> AddString(m_str);              //在列表框 IDC_LIST1 中增加一个项目
    }
    return TRUE;
}
```

OnInitDialog 函数的主要操作步骤如下：

① 打开 m_pLinkSet 指向的存储连接信息的记录集合对象，对记录集合中的所有记录循环，将记录在各个记录上的数据连接单元的信息存储在数组 link_mode、database_id 及 record_id 中。m_NumbLink 记录了与当前图形元素连接的数据连接单元的数目。在调用 CDataLinkDlg 对话框前，通过设置 m_pLinkSet 指向的记录集合对象中的过滤条件式(m_strFilter)，使 m_pLinkSet 指向的记录集合对象中，只包含那些存储了与当前图形元素有关连接信息的记录。

② 得到指向对话框中的列表框 IDC_LIST1 的指针，清空列表框的内容，并将各个与图形元素连接的数据连接单元的信息增加到列表框中。在列表框中，每个项目记录的是如下信息："记录集合对象的名称记录或过滤条件式的识别号"。例如，当图形元素连接的是文档对象中的记录集合对象 m_data1，数据记录的识别号是 121 时，在列表框中如下表示："城市信息　　121"。

在 OnInitDialog 函数中，为了得到记录集合对象的名称，首先调用文档类的成员函数 GetDataIndex，得到识别号为 database_ID[i]的记录集合对象在文档类的结构数组 DataInfo 中的 0 基序号 nIndex。GetDataIndex 函数在文档类 CDrawDoc 中定义如下：

```
public:
    short GetDataIndex(int ID);
```

参数 ID 表示记录集合对象的识别号，函数返回记录集合对象在结构数组 DataInfo 中的 0 基序号。在实现文件 drawdoc.cpp 中，函数的实现代码如下：

```
short CDrawDoc::GetDataIndex(int ID)
{
    for(int i = 0;i < m_NumbData;i++ )       //对所有的记录集合对象循环
```

```
    {
        if(ID = = DataInfo[i].only_ID)        //如果记录集合对象的识别号与 ID 相同
            return i;                         //返回 0 基序号
    }
    return 1;                                 //如果没有找到识别号为 ID 的记录集合对象,
                                              //返回 1
}
```

21.2.2　编辑性质数据

当按下对话框中的"编辑数据"按钮(ID 号为 IDC_DATA_MODIFY)时,能够对在列表框 IDC_LIST1 中选中的数据连接单元进行编辑,以下是具体的实现步骤。

首先用 ClassWizard 为"编辑数据"按钮(ID 号 IDC_DATA_MODIFY)增加一个控制消息处理函数 OnDataModify。操作完毕,在对话框类 CDataLinkDlg 中增加了消息处理函数的定义:

```
//{{AFX_MSG(CDataLinkDlg)
virtual BOOL OnInitDialog();
afx_msg void OnDataModify();
//}}AFX_MSG
```

在实现文件 dlinkdlg.cpp 中,增加了消息映射机制:

```
BEGIN_MESSAGE_MAP(CDataLinkDlg, CDialog)
    //{{AFX_MSG_MAP(CDataLinkDlg)
    ON_BN_CLICKED(IDC_DATA_MODIFY, OnDataModify)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 dlinkdlg.cpp 中,增加了函数的实现代码。以下是修改后的函数实现代码:

```
void CDataLinkDlg::OnDataModify()
{
    char p1[9];
    CListBox * plist1 = (CListBox * )GetDlgItem(IDC_LIST1); //得到指向列表框 IDC_LIST1 的指针
    int nn = plist1 -> GetCurSel();          //得到列表框 IDC_LIST1 中选项的 0 基序号
    if(link_mode[nn] = = 1)                  //如果当前选择连接的连接方式是数据表的一条记录
                                             //为连接单元
    {
        if(database_id[nn] = = 1)            //如果记录集合对象的识别号为 1
        {
            m_pSet1 = &m_pDoc -> m_data1;    //得到指向记录集合对象 m_data1 的指针
            if(m_pSet1 -> IsOpen())          //如果打开了记录集合对象
                m_pSet1 -> Close();          //关闭记录集合对象
            //以下动态产生记录集合的过滤条件
            m_pSet1 -> m_strFilter = "only_id = ";
```

```
            sprintf(p1,"%d",record_id[nn]);
            m_pSet1->m_strFilter+=p1;
            m_pSet1->Open();                    //打开记录集合对象
            CData1Dlg pDg1(m_pSet1,0,0);        //定义一个 CData1Dlg 对话框对话
            pDg1.DoModal();                     //调用对话框,编辑记录集合的当前记录
        }
    }
}
```

OnDataModify 函数实现的功能与视图类 CDatabaseView 中的 OnRecordModify 类似。以下是函数的主要实现步骤：

① 从列表框 IDC_LIST1 中得到当前选项的 0 基序号 nn，根据序号 nn 得到当前连接的连接方式，如果采用的是数据表的一条记录作为连接单元的连接方式，函数继续运行，否则函数退出。

② 如果当前记录集合对象的识别号是 1，将 CDataLinkDig 类中定义的 CData1Set 类指针 m_pSet1指向文档对象中的 CData1Set 对象 m_data1，然后根据记录在数组 record_id 中的记录识别号(record_id[nn])动态产生一个过滤条件。例如，如果记录的识别号为 2，则得到的过滤条件为："only_id=2"。

③ 将记录集合对象的过滤条件(m_strFilter)设置成动态产生的过滤条件，重新打开记录集合对象，这时记录集合对象中应该只包含一条记录。定义一个对话框类 CData1Dlg 对象，调用针对记录集合类 CData1Set 的记录编辑对话框，实现对记录集合当前记录的编辑修改。

以上实现的 OnDataModify 函数，只对数据表的一条记录作为一个连接单元的连接方式下，一个矢量图形元素连接的数据集合对象的类别号是 1 时的情况进行处理。可以对这个函数进行扩充，使它能够对其他的记录集合对象进行处理：

```
void CDataLinkDlg::OnDataModify()
{
    char p1[9];
    CListBox * plist1=(CListBox *)GetDlgItem(IDC_LIST1); //得到指向列表框 IDC_LSIT1 的指针
    int nn=plist1->GetCurSel();                 //得到列表框 IDC_LIST1 中选项的 0 基序号
    if(link_mode[nn]==1)   //如果当前选择的连接的连接方式是数据表的一条记录为连接单元
    {
        if(database_id[nn]==1)                  //如果记录集合对象的识别号为 1
        {
            m_pSet1=&m_pDoc->m_data1;           //得到指向记录集合对象 m_data1 的指针
            if(m_pSet1->IsOpen())               //如果打开了记录集合对象
                m_pSet1->Close();               //关闭记录集合对象
            //以下动态产生记录集合的过滤条件
            m_pSet1->m_strFilter="only_id=";
            sprintf(p1,"%d",record_id[nn]);
            m_pSet1->m_strFilter+=p1;
            m_pSet1->Open();                    //打开记录集合对象
            CData1Dlg pDg1(m_pSet1,0,0);        //定义一个 CData1Dlg 对话框类
```

```
            pDg1.DoModal();                    //调用对话框编辑记录集合的当前记录
        }
        else if(database_id[nn] == 2)          //如果记录集合对象的识别号为 2
        {
            m_pSet2 = &m_pDoc->m_data2;        //得到指向记录集合对象 m_data2 的指针
            if(m_pSet2->IsOpen())              //如果打开了记录集合对象
                m_pSet2->Close();              //关闭记录集合对象
            //以下动态产生记录集合的过滤条件
            m_pSet2->m_strFilter="only_id=";
            sprintf(p1,"%d",record_id[nn]);
            m_pSet2->m_strFilter += p1;
            m_pSet2->Open();                   //打开记录集合对象
            CData2Dlg pDg1(m_pSet2,0,0);       //定义一个对话框类 CData1Dlg 对象
            pDg1.DoModal();                    //调用对话框编辑记录集合的当前记录
        }
    }
}
```

OnDataModify 函数中的黑体部分代码，实现在连接的记录集合对象的识别号是 2 的情况下，通过对话框类 CData2Dlg 对象来组织对话框界面，完成对连接的数据连接单元进行编辑。m_pSet2 是一个 CData2Set 类指针，在 CDataLinkDlg 类中定义：

```
CData2Set * m_pSet2;
```

也可以对 OnDataModify 函数进行扩充，使它能够编辑用第二种连接方式(数据表的记录集合作为连接单元)连接的数据连接单元。在这种情况下，除了需要修改 OnDataModify 函数外，还需要建立另外的对话框界面来编辑数据表的一个记录集合。

21.2.3　删除与数据连接单元的连接

当运行对话框中的“删除连接”按钮(ID 号为 IDC_LINK_DELETE)时，要求系统能够删除在列表框 IDC_LIST1 中选中的连接。以下讨论实现这个操作功能的步骤。

用 ClassWizard 为“编辑数据”按钮 IDC_LINK_DELETE 增加一个控制消息处理函数 OnLinkDelete。操作完毕后，在对话框类 CDataLinkDlg 中增加了消息处理函数的定义：

```
//{{AFX_MSG(CDataLinkDlg)
virtual BOOL OnInitDialog();
afx_msg void OnDataModify();
afx_msg void OnLinkDelete();
//}}AFX_MSG
```

在实现文件 dlinkdlg.cpp 中，增加了消息映射机制：

```
BEGIN_MESSAGE_MAP(CDataLinkDlg, CDialog)
    //{{AFX_MSG_MAP(CDataLinkDlg)
    ON_BN_CLICKED(IDC_DATA_MODIFY, OnDataModify)
```

```
    ON_BN_CLICKED(IDC_LINK_DELETE, OnLinkDelete)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 dlinkdlg.cpp 中，增加了 OnLinkDelete 函数的实现代码。以下是修改后的函数实现代码：

```
void CDataLinkDlg::OnLinkDelete()
{
    CListBox * plist1 = (CListBox * )GetDlgItem(IDC_LIST1); //得到指向列表框 IDC_LIST1 的指针
    int nn = plist1 -> GetCurSel();                    //得到列表框 IDC_LIST1 中选项的 0 基序号
    m_pLinkSet -> Move(nn + 1, SQL_FETCH_ABSOLUTE);//将记录指针指向记录集合的
                                                       //第 nn + 1 条记录
    m_pLinkSet -> Delete();              //删除当前记录
    m_pLinkSet -> Close();               //关闭记录集合对象
    SendMessage(WM_INITDIALOG);          //发送 WM_INITDIALOG 消息，以运行 OnInitDialog 函数
                                         //初始化对话框
    //以下根据删除的连接在列表框 IDC_LIST1 中的位置，设置列表框中的当前选项
    if(nn > = m_NumbLink)
        plist1 -> SetCurSel(m_NumbLink - 1);
    else
        plist1 -> SetCurSel(nn);
}
```

OnLinkDelete 函数的主要操作步骤如下：

① 从列表框 IDC_LIST1 中得到当前选项的 0 基序号 nn，将 m_pLinkSet 指向的记录集合对象的记录指针移动到第 nn + 1 条记录，这条记录就是存储要删除连接的记录。

② 调用 CRecordSet 类的 Delete 函数，删除 m_pLinkSet 指向的记录集合对象中的当前记录。

③ 发送 WM_INITDIALOG 消息，激发运行 OnInitDialog 函数，使对话框的状态和成员变量重新进行初始化。

④ 在列表框 IDC_LIST1 中选择当前选项。当删除的连接项目序号大于等于当前的项目数时，选择列表框 IDC_LIST1 的最后一个项目(序号是 m_NumbLink - 1)，否则，还是指向原来序号(nn)的项目。

21.3　实现信息查询功能

21.2 节实现了一个对话框类 CDataLinkDlg，用来显示和编辑一个矢量图形元素连接的数据信息。本节利用这个对话框类，在应用程序 Draw 中实现查询一个矢量图形元素所连接的性质数据信息的功能。

(1) 增加一个菜单项

为了激发运行查询图形元素性质数据信息的功能，需要在应用程序 Draw 的菜单资源 IDR

_DRAWTYPE(矢量图形视图所采用的菜单资源)中增加一个菜单项“信息查询”。为方便起见,将菜单项加在了“图形选中”菜单下:

```
IDR_DRAWTYPE MENU PRELOAD DISCARDABLE
BEGIN
    ......//省略以上代码
    POPUP "图形选中"
    BEGIN
        MENUITEM "&S 鼠标点中",          ID_SELECT_MOUSE
        MENUITEM "&U 放弃选中",          ID_SELECT_CLEAR
        MENUITEM "&D 图形删除",          ID_SELECT_DELETE
        MENUITEM "&F 性质查询",          ID_DATA_SEACHER
    END
    ......//省略以下代码
END
```

在资源 ID 号定义文件 resource.h 中,给菜单项的 ID 号进行定义:

```
# define ID_DATA_SEACHER          32800
```

(2) 增加消息处理函数

为了在矢量图形视图中激发运行以上定义的菜单项,用 ClassWizard 在视图类 CDrawView 中定义这个菜单项(ID 号为 ID_DATA_SEACHER)的消息处理函数。操作完成后,在视图类 CDrawView 中增加了消息处理函数的定义:

```
//{{AFX_MSG(CDrawView)
......//省略以上代码
afx_msg void OnDataSeacher();
//}}AFX_MSG
```

在实现文件 drawview.cpp 中,增加了消息映射机制:

```
BEGIN_MESSAGE_MAP(CDrawView, CView)
    //{{AFX_MSG_MAP(CDataLinkDlg)
    ......//省略以上代码
    ON_COMMAND(ID_DATA_SEACHER, OnDataSeacher)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在实现文件 drawview.cpp 中,增加了函数的实现代码。对函数实现代码修改如下:

```
void CDrawView::OnDataSeacher()
{
    m_DrawCurrent = 31;          //标识查询图形元素连接的性质数据信息
    PushNumb = 0;
```

```
}
```

在 OnDataSeacher 函数中并没有进行具体的查询操作，而只是将 m_DrawCurrent 设置为 31，标识正在进行性质数据信息查询操作，然后通过对鼠标消息处理函数 OnLButtonDown 的修改，完成具体的查询功能。

(3) 修改 OnLButtonDown 函数实现性质数据信息查询功能

当在应用程序 Draw 中运行“性质查询”菜单项后，用鼠标在矢量图形中点中一个图形元素，当选中的图形元素存在与数据连接单元的连接时，就会调用对话框类 CDataLinkDlg 产生的对话框，显示和修改这个图形元素连接的数据连接单元。为了实现这个功能，对按下鼠标左键的消息处理函数 OnLButtonDown 进行修改，加入如下的代码：

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    ...//省略以上代码
    else if(m_DrawCurrent==31)                    //进行鼠标查询图形元素的性质数据信息操作
    {
        CString m_str;
        char p1[9];
        VPtoDP(point.x,point.y,&xx1,&yy1);        //将鼠标按中点坐标转换成实际坐标
        float jl=blc*4;                           //选中范围(四个像素的实际距离)
        //以下调用 PointSelect 函数来点选一个图形元素
        BOOL pb=pDoc->PointSelect(xx1,yy1,jl,blc,&Lb,&Index,&pbh);
        if(pb)                                    //如果选中了一个图形元素
        {
            if(m_pLinkSet->IsOpen())              //如果打开了存储连接信息的记录集合对象
                m_pLinkSet->Close();              //关闭记录集合对象
            //以下动态产生一个过滤条件式
            m_str="(graph_id=";
            sprintf(p1,"%d",Lb);
            m_str+=p1;
            m_str+=") AND (element_id=";
            sprintf(p1,"%d",pbh);
            m_str+=p1;
            m_str+=")";
            m_pLinkSet->m_strFilter=m_str;        //设置记录集合的过滤条件为产生的
                                                  //过滤条件式
            m_pLinkSet->Open();                   //重新打开记录集合对象
            if(m_pLinkSet->IsBOF())               //如果记录集合中不包含任何记录
            {
                m_pLinkSet->Close();              //关闭记录集合对象
                return;                           //退出函数
```

```
            }
            CDataLinkDlg pDg1(pDoc,m_pLinkSet);      //定义一个对话框类 CDataLinkDlg 对象
            pDg1.DoModal();                          //调用对话框显示和修改图形元素连接的性质数据信息
        }
    }
}
```

以上代码的主要操作步骤如下：

① 将鼠标点中点的坐标转换成实际坐标，调用 PointSelect 函数来计算判断是否选中了图形元素，并返回选中图形元素的类别 Lb 和识别号 pbh。

② 如果选中了图形元素，则根据选中图形元素的类别和识别号，来动态产生一个过滤条件式。例如，对于选中图形元素的类别为 1(直线)、识别号为 120 时，产生一个如下的过滤条件式：

"(graph_id = 1) AND (element_id = 120)"

③ 关闭存储连接信息的记录集合对象，将动态产生的过滤条件式设置成记录集合对象的过滤条件，并重新打开记录集合对象。如果经过过滤操作的记录集合不包含任何记录(if (pLinkSet->IsBOF())，即当前选中的图形元素没有与任何数据连接单元连接，退出 OnLButtonDown 函数。

④ 如果过滤操作后产生的记录集合中包含记录，则定义一个对话框类 CDataLinkDlg 对象 pDg1，并利用 plg1 调用对话框，实现对当前选中图形元素连接的数据连接单元的显示和编辑功能。

至此，在应用程序 Draw 中完成了在矢量图形上利用鼠标对矢量图形元素的信息查询功能。编译并执行应用程序 Draw，在矢量图形视图中，运行"性质查询"菜单项后，就可以用鼠标点中图形元素来显示和编辑这个图形元素所连接的性质数据信息(数据连接单元)。

第 22 章　图形元素的空间拓扑关系

在 GIS 系统中，需要完成大量的空间统计和分析功能。如统计一个矩形区域、一个圆区域内的某种实体的数量，计算一条线路（用连续直线表示）与其他线路、行政区域、地形区域（可以用封闭多边形区域来表示各类区域）如何相交，相交的段如何？……。所有这些统计和分析功能的核心是各类矢量图形元素间的拓扑关系，即各类图形元素间的相互位置关系。

本章讨论实现各类基本图形元素间的相交关系的计算思路和具体实现函数。

22.1　直线与多边形区域的关系

本节讨论直线与多边形区域的相交关系。并得到计算直线与多边形区域、连续直线与多边形区域相交关系的计算函数。

22.1.1　直线与直线的关系

多边形区域是由多个直线段连接而成的，所以在讨论直线与多边形区域的相交关系之前，需要首先讨论计算直线与直线相交关系的算法。

（1）计算思路

怎么来解决直线与直线的相交问题呢？这里所说的直线，是一个直线段。在平面上，两条直线段存在以下两种相交关系：相交、不相交。从数学上解决这个问题的方法是：得到两条直线的方程，根据方程能得两条直线段延长线的交点（在平面上两条不平行的直线肯定有一个交点），只要再判断这个交点是否在两条直线段上，就可以得到两条直线段是否相交及交点。用计算机程序来解决这个问题时，可以把这个过程用以下的步骤来解决。

根据两条直线段的端点坐标判断这两条直线段是否为竖直线，根据直线段是否为竖直线可以分为以下四种情况：

① 两条直线段都是竖直线。在这种情况下，首先判断两条直线的横坐标是否相同，横坐标不同，两条直线肯定不相交。在两条直线的横坐标相同的情况下，再根据两条直线段的起终点的纵坐标来判断两条直线段的相交情况（不相交、一点相交还是两点相交（部分直线段重合））。

② 第一条直线段是竖直线，第二条直线段是斜线。在这种情况下，根据第一条直线的横坐标，解方程来得到两条直线的交点，然后通过判断交点是否在两条直线段上，来判断两直线段是否相交及交点坐标。

③ 第一条直线段是斜线，第二条直线段是竖直线。这种情况的处理与第②种情况相同。

④ 两条直线段都是斜线。在这种情况下，计算出两条直线的斜率，判断两条直线是否平行。如果两条直线平行，经过计算判断两条直线是不相交、一点相交，还是部分重合。如果两

条直线不平行，解方程得到两条直线的交点，然后通过判断交点是否在两条直线段上，来确定两条直线段是否相交及交点坐标。

以上是解决两条直线段相交问题的完整计算机程序实现步骤。在实际实现中，也可以在直线是竖直线时，通过给其一个非常大的斜率(如 100000)，将①、②、③步骤简化成第④种情况，只通过第④种情况来计算两条直线段的相交关系。

(2) 函数的实现

为了计算直线段与直线段相交关系，在应用程序 Draw 的直线类 CLine 中定义一个函数：

public:

```
int LineLine(CLine * line1,float * xxx1,float * yyy1,float * xxx2,float * yyy2);
```

LineLine 函数计算任意两条直线段的相交关系及交点。第一条直线的坐标由调用这个函数的 CLine 对象本身确定，第二条直线由 CLine 类指针 line1 指向的 CLine 对象确定。计算得到的交点坐标由参数(*xxx1, *yyy1)和(*xxx2, *yyy2)返回。根据不同的相交情况，函数返回不同的值：

0：两条直线段不相交；

1：两条直线段有一个交点，交点坐标通过(*xxx1, *yyy1)返回；

2：两条直线段部分重合，重合段的起终坐标分别通过(*xxx1, *yyy1)和(*xxx2, *yyy2)返回。

在实现文件 drawdoc.cpp 中，加入函数的实现代码：

```
int CLine::LineLine(CLine * line1,float * xxx1,float * yyy1,float * xxx2,float * yyy2)
{
    float xx1,xx2,yy1,yy2;
    float x1,x2,y1,y2,n1,n2,n3,n4;
    float k1,k2;
    /*
        可在此处加入检查两条直线的边界矩形是否重叠的代码，如果不重叠，
        函数直接返回 0(两条直线不相交)
    */
    //将当前直线对象中直线段的两个顶点按横坐标由小到大放在(xx1,yy1)和(xx2,yy2)
    if(m_X1 > m_X2)          //如果起点横坐标大于终点横坐标
    {
        xx1 = m_X2;
        xx2 = m_X1;
        yy1 = m_Y2;
        yy2 = m_Y1;
    }
    else                     //如果起点横坐标小于等于终点横坐标
    {
        xx1 = m_X1;
```

```
    xx2 = m_X2;
    yy1 = m_Y1;
    yy2 = m_Y2;
}
if(xx1 = = xx2&&yy1 = = yy2) //如果第一条直线是一个点,则两条直线不不相交,函数返回 0
    return 0;
//将第二条直线段的两个顶点按横坐标由小到大放在(x1,y1)和(x2,y2)
if(line1 - > m_X1 > line1 - > m_X2)        //如果起点横坐标大于终点横坐标
{
    x1 = line1 - > m_X2;
    x2 = line1 - > m_X1;
    y1 = line1 - > m_Y2;
    y2 = line1 - > m_Y1;
}
else                                      //如果起点横坐标小于等于终点横坐标
{
    x1 = line1 - > m_X1;
    x2 = line1 - > m_X2;
    y1 = line1 - > m_Y1;
    y2 = line1 - > m_Y2;
}
if(x1 = = x2&&y1 = = y2)       //如果第二条直线是一个点,则两条直线不相交,函数返回 0
    return 0;
if(x1 = = x2&&xx1 = = xx2)     //两条直线都是竖直线
{
    if(x1! = xx1)              //两条平行竖直线的横坐标不同,两条直线段不相交,函数返回 0
        return 0;
  * xxx1 = x1;                 //得到交点的横坐标
  n1 = min(yy1,yy2);           //第一条直线段的最小纵坐标
  n2 = max(yy1,yy2);           //第一条直线段的最大纵坐标
  n3 = min(y1,y2);             //第二条直线段的最小纵坐标
  n4 = max(y1,y2);             //第二条直线段的最大纵坐标
  if(n1 > n4||n2 < n3)         //两条横坐标相同的竖直线,纵坐标不交叉,两条直线不相交
      return 0;                //函数返回 0
  else if(n1 = = n4||n2 = = n3) //两条横坐标相同的竖直线纵坐标有一点相交(连接)
  {
      if(n1 = = n4)            //第一条直线的开始点与第二条直线的结束点纵坐标相同
      {
          * yyy1 = n1;         //得到交点的纵坐标
          return 1;            //两条直线段有 1 个交点,函数返回 1
      }
      if(n2 = = n3)            //第一条直线的结束点与第二条直线的开始点纵坐标相同
      {
```

```
            * yyy1 = n2;                    //得到交点的纵坐标
            return 1;                       //两条直线段有 1 个交点,函数返回 1
        }
    }
    else                                    //如果两条直线的纵坐标重叠
    {
        * yyy1 = max(n1,n3);                //重叠直线段的起点纵坐标
        * yyy2 = min(n2,n4);                //重叠直线段的结束纵坐标
        return 2;                           //两条直线段重叠,有两个交点,函数返回 2
    }
}
else if(x1 = = x2&&xx1! = xx2)              //如果第一条直线是斜线,第二条直线是竖直线
{
    * xxx1 = x1;                            //得到交点的横坐标
    if(! (x1 > = xx1&&x1 < = xx2))          //如果交点横坐标没有在第一条直线的横坐标范围内
        return 0;                           //两条直线不相交,函数返回 0
    * yyy1 = yy1 + (x1 - xx1)/(xx2 - xx1) * (yy2 - yy1);          //计算得到交点的纵坐标
    if( * yyy1 > = y1&& * yyy1 < = y2|| * yyy1 > = y2&& * yyy1 < = y1)   //交点纵坐标在第二条
                                                                  //直线纵坐标范围内
        return 1;                           //两条直线有一个交点,函数返回 1
    else                                    //交点纵坐标没有在第二直线纵坐标范围内
        return 0;                           //两条直线不相交,函数返回 0
}
else if(x1! = x2&&xx1 = = xx2)              //如果第一条直线是竖直线,第二条直线是斜线
{
    * xxx1 = xx1;                           //得到交点的横坐标
    if(! (xx1 > = x1&&xx1 < = x2))          //如果交点横坐标没有在第二条直线的横坐标范围内
        return 0;                           //两条直线不相交,函数返回 0
    * yyy1 = y1 + (xx1 - x1)/(x2 - x1) * (y2 - y1);                //计算得到交点的纵坐标
    if( * yyy1 > = yy1&& * yyy1 < = yy2|| * yyy1 > = yy2&& * yyy1 < = yy1)    //交点纵坐标在
                                                                  //第一直线范围内
        return 1;                           //两条直线有一个交点,函数返回 1
    else                                    //交点纵坐标没有在第二条直线纵坐标范围内
        return 0;                           //两条直线不相交,函数返回 0
}
else                                        //如果两条直线都是斜线
{
    k1 = (y2 - y1)/(x2 - x1);               //得到第二条直线的斜率
    k2 = (yy2 - yy1)/(xx2 - xx1);           //得到第一条直线的斜率
    if(k1 = = k2)                           //如果两条直线平行

        if(x1 > xx2||x2 < xx1)              //两条直线的横坐标不重叠
            return 0;                       //两条直线不相交,函数返回 0
```

```
            else                                    //两条直线的横坐标重叠
            {
                n1 = y1 + (0 - x1) * k1;            //横坐标为 x1 时,第二条直线的纵坐标
                n2 = yy1 + (0 - x1) * k2;           //横坐标为 x1 时,第一条直线的纵坐标
                if(n1! = n2)                        //两条直线的纵坐标不同,即两条直线平行
                    return 0;                       //两条直线不相交,函数返回 0
                else                                //如果两条直线的纵坐标相等,即两条直线重合
                {
                    n3 = max(xx1,x1);               //重叠段起点横坐标
                    n4 = min(xx2,x2);               //重叠段终点横坐标
                    * xxx1 = n3;                    //重叠段起点横坐标
                    * yyy1 = y1 + (n3 - x1) * k1;   //重叠段起点纵坐标
                    if(n3 = = n4)                   //如果重叠段是一个点
                        return 1;                   //两条直线有一个交点,函数返回 1
                    * xxx2 = n4;                    //重叠段终点横坐标
                    * yyy2 = y1 + (n4 - x1) * k1;   //重叠段终点纵坐标
                    return 2;                       //两条直线有二个交点,函数返回 2
                }
            }
        }
        else                                        //如果两条直线都是斜线,且两条直线不平行
        {
            * xxx1 = (yy1 - y1 + x1 * k1 - xx1 * k2)/(k1 - k2);     //交点的横坐标
            * yyy1 = y1 + ( * xxx1 - x1) * k1;                      //交点的纵坐标
        //如果交点横坐标在两条直线段的横坐标范围内
            if(( * xxx1 > = x1&& * xxx1 < = x2)&&( * xxx1 > = xx1&& * xxx1 < = xx2))
                return 1;                           //两条直线段有一个交点,函数返回 1
            else                                  //如果交点横坐标不全在两条直线段的横坐标范围内
                return 0;                           //两条直线段不相交,函数返回 0
        }
    }
    return 0;
}
```

在 LineLine 函数中,第二条直线的坐标是直接通过 CLine 对象 line1 得到的,如果在直线类 CLine 中,存储直线坐标的成员变量定义成保护型(protected)或私有型的(private),就不能直接使用直线对象中的顶点坐标,而需要在直线类 CLine 中定义一个成员函数来返回直线的起终坐标。如可以在直线类 CLine 中定义如下的函数:

```
public:
    void GetXY(int * x1, * y1, * x2, * y2);
```

在实现文件 drawdoc.cpp 中,加入函数的实现代码:

```
void CLine :: GetXY(int * x1, * y1, * x2, * y2)
```

```
{
    *x1 = m_X1;
    *y1 = m_Y1;
    *x2 = m_X2;
    *y2 = m_Y2;
}
```

可以通过这个成员函数来获得直线对象的顶点坐标：

```
float x1,y1,x2,y2;
line1 -> GetXY(&x1,&y1,&x2,&y2);
```

(3) 函数的测试使用

以下介绍在应用程序 Draw 中测试和使用 LineLine 函数的方法。假定在应用程序 Draw 中定义了两个 CLine 类指针：

```
CLine *line1, *line2;
```

将这两个类指针分别指向要计算相交关系的两个 CLine 对象。以下的代码可以调用 LineLine 函数计算这两个直线的相交关系：

```
float x1,y1,x2,y2;
int numb;
numb = line1 -> LineLine(line2,&x1,&y1,&x2,&y2);  //调用 LineLine 函数计算两条直线的相交关系
if(numb == 0)          //如果两条直线段没有交点
    ;
else if(numb == 1)     //两条有一个交点,交点坐标为(x1,y1)
    ;
else if(numb == 2)     //两条直线段部分重合,重合段的起点和终点坐标为(x1,y1)和(x2,y2)
    ;
```

在应用程序 Draw 中，可以参考以上的方法来灵活使用 CLine 类中的 LineLine 函数，计算得到两条直线段的相交关系。

22.1.2 直线与多边形区域的相交关系

在实现了计算两条直线段相交关系的 LineLine 函数后，可以组织计算一条直线段和一个多边形区域(即一条连续直线)的相交关系。

(1) 计算思路

在很多情况下，讨论直线段与多边形区域的相交关系时，感兴趣的往往不只是两者是否相交并得到交点，更主要的是要得到直线与多边形区域的交点中，哪些交点之间的线段在这个多边形区域内。解决这个问题，在几何学上是靠判断直线与连续直线的各个段的相交方向(左侧还是右侧)以及各直线段的序列来判断的。这个方法不太好理解，实现上也有一定的困难，现

在给读者提供一个容易理解和实现的方法。以下是这个方法的实现步骤：

① 通过直线段与直线段交点计算函数 LineLine，得到直线段与多边形区域各直线段的所有交点，再加上直线段本身的起点和终点得到所有的交点。

② 将所有的交点按照 X 轴方向或者 Y 轴方向进行排序（当直线段是竖直线时必须按照 Y 轴方向排序）。

③ 得到两个连续交点的中点，检查这个中点是否在多边形区域内，如果这个点在多边形区域内，这两个连续交点间的直线段就在多边形区域内，否则，就不在多边形区域内。

通过以上的计算步骤可以得到直线段与多边形区域的所有交点，并且得到两个交点之间的直线段是否在多边形区域内。

(2) 函数实现

为计算直线与多边形区域的相交关系，在应用程序 Draw 的连续直线类 CPline 中，定义一个函数：

```
public:
    int LineRgn(CLine * line1,int * nCross,PointStruct * Pointxy1);
```

LineRgn 函数计算一条直线段和一个多边形区域所有边的交点，并判断两个相临交点间的直线段是否在这个多边形区域内。参数 line1 指向存储直线段坐标的 CLine 对象，多边形区域的所有顶点坐标存储在调用这个函数的 CPline 对象中。直线段与多边形区域相交点的数目通过指针变量 nCross 返回，各个交点的坐标和以这个交点开始的直线段是否在多边形区域中的判断通过结构数组 Pointxy1 返回。Pointxy1 是指向 PointStruct 结构数组的指针（PointStruct 结构的定义见 5.1.3 小节）。根据直线段与多边形区域相交关系，函数返回如下值：

0：　直线段与多边形区域边不相交，且直线段在多边形区域外；

1：　直线段与多边形区域边不相交，且直线段在多边形区域内；

2：　直线段与多边形区域边相交。

在实现文件 drawdoc.cpp 中，加入函数的实现代码：

```
int CPline::LineRgn(CLine * line1,int * nCross,PointStruct * Pointxy1)
{
    int pb,sm,pb1;
    float x1,y1,x2,y2;
    float xx1,yy1,xx2,yy2;
    CLine line2;
    /*
    此处加入代码检查直线的边界矩形和多边形区域的边界矩形是否重叠，如果
    边界矩形不重叠，函数直接返回 0(直线在多边形区域外)
    */
    //得到直线段的起点坐标(xx1,yy1)和终点坐标(xx2,yy2)
    xx1 = line1 -> m_X1;
    yy1 = line1 -> m_Y1;
```

```
xx2 = line1 - > m_X2;
yy2 = line1 - > m_Y2;
//在连续直线各个顶点后增加一个顶点,形成封闭区域
m_PointList[m_Numble].x = m_PointList[0].x;
m_PointList[m_Numble].y = m_PointList[0].y;
//将直线段的起点作为第 1 个交点
Pointxy1[0].x = xx1;
Pointxy1[0].y = yy1;
sm = 1;   //交点的数目为 1,直线段的起点作为第一个交点
for(int i = 0;i < m_Numble;i++)
//对多边形区域的所有直线循环,用直线边坐标初始化直线对象 line2
{
    x1 = line2.m_X1 = m_PointList[i].x;          //直线段的起点横坐标
    y1 = line2.m_Y1 = m_PointList[i].y;          //直线段的起点纵坐标
    x2 = line2.m_X2 = m_PointList[i + 1].x;      //直线段的终点横坐标
    y2 = line2.m_Y2 = m_PointList[i + 1].y;      //直线段的终点纵坐标
    //调用 LineLine 函数,计算直线段与多边形区域直线边的交点
    pb = line1 - > LineLine(&line2,&x1,&y1,&x2,&y2);
    if(pb > = 1) //如果有交点,将第一个交点的坐标记录在结构指针 Pointxy1 指向的结构数组
    {
        Pointxy1[sm].x = x1;                     //交点的横坐标
        Pointxy1[sm++].y = y1;                   //交点的纵坐标
    }
    if(pb = = 2)   //如有两个交点,将第二个交点的坐标记录在指针 Pointxy1 指向的结构数组
    {
        Pointxy1[sm].x = x2;                     //交点的横坐标
        Pointxy1[sm++].y = y2;                   //交点的纵坐标
    }
}
//将直线段的终点作为最后一个交点
Pointxy1[sm].x = xx2;
Pointxy1[sm++].y = yy2;

//以下对交点的坐标进行排序
pb = 1;                                          //表示按横坐标进行排序
if(xx1 = = xx2)                                  //如果直线段是竖直线,则按纵坐标排序
    pb = 2;                                      //表示按纵坐标进行排序
do                                               //按横坐标或纵坐标排序
{
    pb1 = 0;
    for(i = 0;i < sm - 1;i++)                    //对所有的交点循环
    {
        if(pb = = 1&&(Pointxy1[i].x > Pointxy1[i + 1].x&&xx1 < xx2
```

```
                    ||Pointxy1[i].x < Pointxy1[i+1].x&&xx1 > xx2)
                    ||pb = = 2&&(Pointxy1[i].y > Pointxy1[i+1].y&&yy1 < yy2
                    ||Pointxy1[i].y < Pointxy1[i+1].y&&yy1 > yy2))
                //如果连续两个交点的横坐标(或纵坐标)顺序与直线段起点到终点的顺序反向,
                //则交换两个交点的位置
                {
                    x1 = Pointxy1[i].x; y1 = Pointxy1[i].y;
                    Pointxy1[i].x = Pointxy1[i+1].x; Pointxy1[i].y = Pointxy1[i+1].y;
                    Pointxy1[i+1].x = x1; Pointxy1[i+1].y = y1;
                    pb1 = 1;
                }
            }
        }while(pb1 = = 1);          //直到所有的交点都是按照横坐标或纵坐标顺序排列
        for(i=0;i<sm-1;i++)  //对所有交点循环,检查两个相邻交点的中点是否在多边形区域中
        {
            x1 = (Pointxy1[i].x + Pointxy1[i+1].x)/2;      //中点的横坐标
            y1 = (Pointxy1[i].y + Pointxy1[i+1].y)/2;      //中点的纵坐标
            if(PointRgn(x1,y1,m_Numble,m_PointList,1)) //如果中点是否在多边形区域内
                Pointxy1[i].z = 1;                         //表示中点在多边形区域内
            else
                Pointxy1[i].z = 0;                         //表示中点不在多边形区域内
        }
        *nCross = sm;                                      //得到交点的数目
        if(*nCross = = 2)                                  //如果只有两个交点
        {
            if(Pointxy1[0].z = = 1)                        //如果这两个交点的中点在多边形区域内
                return 1;                                  //直线段包含在多边形区域内,函数返回 1
            else                                           //如果中点不在多边形区域内
                return 0;                                  //直线段在多边形区域外,函数返回 0
        }
        else                                               //如果交点的数目多于 2 个
        return 2;                                          //直线段与多边形区域边相交,函数返回 2
    }
```

LineRgn 函数的实现步骤可参考函数中的详细注释,以下是函数的主要操作步骤:

① 创建一个 CLine 对象 line2,对一个多边形区域的所有直线循环,用直线边的顶点坐标来初始化 CLine 对象 line2 的。通过在 CLine 对象 line1 中调用 CLine 类的 LineLine 函数,计算出直线段与组成多边形区域各个直线边的交点,再把直线段的起始点和结束点作为第一个和最后一个交点,就得到直线段与多边形区域的所有交点。

② 对所有的交点按照横坐标进行排序(在直线是竖直线的情况下,按纵坐标排序),排序的顺序是按照从直线段起点到终点的方向。例如,当直线段的起点横坐标大于终点横坐标时,则按降序排列。

③ 得到所有交点中相临两个交点的中点,调用 CDraw 类的 PointRgn 函数,来判断中点是

否在多边形区域内，从而判定两个交点间的直线段是否在多边形区域内。

④ 判断直线段与多边形区域的相交关系，返回函数的值。

(3) 函数的测试使用

以下介绍在应用程序 Draw 中测试和使用 LineRgn 函数的方法。现有两个类指针：

```
CLine * line1;
CPline * pline1;
```

将这两个类指针分别指向要计算相交关系的直线对象和多边形区域对象。以下代码完成计算这两个图形元素间的相交关系：

```
PointStruct * Pointxy1;
int crossNumb, numb;
Pointxy1 = new PointStruct[pline1 -> GetNumb() * 2 + 2];//动态分配一个 PointStruct 结构数组 Pointxy1
//以下调用 LineRgn 函数计算直线段与多边形区域的相交关系，函数返回值为 numb
numb = pline1 -> LineRgn(line1, &crossNumb, Pointxy1);
if(numb = = 0)          //直线段在多边形区域外
    ;
else if(numb = = 1)     //直线段在多边形区域内
    ;
else if(numb = = 2)     //直线与多边形区域边相交，交点数是 crossNumb - 2，各个交点坐标以及交
                        //点后的直线段是否在多边形区域内的判断信息，存储在结构数组 Pointxy1 中
;
delete Pointxy1;        //释放动态分配的 Pointxy1 结构数组
```

在以上代码中，动态定义了一个 PointStruct 结构数组 Pointxy1，利用这个数组存储并返回直线段与多边形区域的相交信息。Pointxy1 结构数组的容量定为多边形区域边数的 2 倍加 2，因为一条直线段与一条直线段最多有两个交点，所以 Pointxy1 数组的容量能够满足任何情况。

在应用程序 Draw 中对 LineRgn 函数进行测试，结合图形绘制函数能够得到如图 22-1 所示的结果。一个直线段和一个多边形区域相交，与多边形区域相交的直线段用粗线显示。

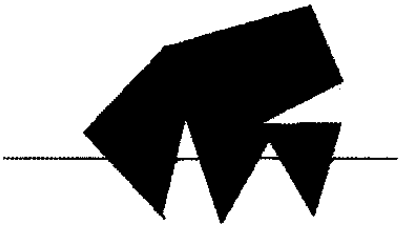


图 22-1 直线段与多边形区域相交图

22.1.3　连续直线与多边形区域的相交关系

在计算直线与多边形区域相交关系的函数 LineRgn 基础上，可以较容易地实现连续直线与多边形区域相交关系的计算函数。其实现的思路是：利用 LineRgn 函数获得连续直线中各个直线段与多边形区域的相交关系，然后经过处理得到连续直线与多边形区域的相交关系。

(1) 函数实现

为了完成连续直线与多边形区域相交关系的计算，在应用程序 Draw 的连续直线类 CPline 中定义一个函数：

```
public:
    int PLineRgn(CPline * pline1,int * nCross,PointStruct * Pointxy1);
```

多边形区域的顶点坐标存储在参数 pline1 指向的 CPline 对象中，连续直线的顶点坐标存储在调用这个函数的 CPline 对象中。指针参数 nCross 返回交点的数目，各个交点的坐标和以这个交点开始的直线段是否在多边形区域中的判断，通过结构数组 Pointxy1 返回，Pointxy1 是一个指向 PointStruct 结构数组的指针。根据连续直线和多边形区域的不同相交情况，函数返回不同的值：

0：连续直线与多边形区域不相交，连续直线在多边形区域外；
1：连续直线与多边形区域不相交，连续直线在多边形区域内；
2：连续直线与多边形区域相交。

在实现文件 drawdoc.cpp 中，加入函数的实现代码：

```
int CPline::PLineRgn(CPline * pline1,int * nCross,PointStruct * Pointxy1)
{
    int NumbAll,Numb;
    BOOL pb1,pb2;
    PointStruct * Pointxy;
    CLine line1;
    /*
        此处加入代码检查连续直线的边界矩形和多边形区域的边界矩形是否重叠，
        如果边界矩形不重叠，函数返回 0(连续直线在在多边形区域外)
    */
    Pointxy = new PointStruct[m_Numble * 2 + 2];//为 Pointxy 分配结构数组
    NumbAll = 0;
    for(int i = 0;i < m_Numble - 1;i++)          //对连续直线的各个直线段循环
    {
        line1.m_X1 = m_PointList[i].x;           //直线段的起点横坐标
        line1.m_Y1 = m_PointList[i].y;           //直线段的起点重坐标
        line1.m_X2 = m_PointList[i + 1].x;       //直线段的终点横坐标
        line1.m_Y2 = m_PointList[i + 1].y;       //直线段的终点纵坐标
```

```
        plinel -> LineRgn(&linel,&Numb,Pointxy);        //调用函数 LineRgn 计算直线段与多边形
                                                        //区域的交点
        for(int j=0;j<Numb-1;j++)          //存储直线段与多边形区域的各个交点
            Pointxy1[NumbAll++]=Pointxy[j];
    }
    Pointxy1[NumbAll++]=Pointxy[Numb-1]; //将最后一个直线段的终点坐标作为最后一个交点
    *nCross=NumbAll;                       //相交顶点的数目
    pb1=0;
    pb2=0;
    for(i=0;i<NumbAll-1;i++)               //对所有相邻顶点组成的直线段循环
    {
        if(Pointxy1[i].z==1)               //如果直线段的中点在多边形区域内
            pb1=1;
        else                               //如果直线段的中点不在多边形区域内
            pb2=1;
    }
    delete Pointxy;                        //释放分配的 PointStruct 数组
    if(pb1==1&&pb2==1)                     //在多边形区域内外都有直线段,函数返回 0
        return 2;
    else if(pb1==1&&pb2==0)                //所有的直线段都在多边形区域内,函数返回 1
        return 1;
    else                                   //所有的直线段都在多边形区域外,函数返回 0
        return 0;
}
```

PLineRgn 函数的主要操作步骤如下：

① 动态分配一个 PointStruct 结构数组 Pointxy，这个数组用来返回一个直线段与一个多边形区域的相交信息，结构数组的项数为多边形区域的顶点数目乘 2 并加 2，能够满足各种情况下的容量要求。

② 对连续直线的各个直线段循环，用直线段的坐标初始化直线对象 linel，通过指向多边形区域对象的指针 plinel 调用 CPline 类的 LineRgn 函数，得到多边形区域与直线的相交信息，并把返回的相交信息都记录在 PointStruct 结构数组 Pointxy1 中。因为在 LineRgn 函数中，直线段与多边形区域的交点是由直线段的起点到终点顺序排列的，所以很容易将连续直线中各个直线段与多边形区域的相交信息连接起来。在连接时，为防止上个直线段的终点（Pointxy 数组的最后一项）与当前直线段的起点（Poingxy 数组的第一项）之间又会产生一个直线段（这两个是同一个点），在记录 LineRgn 返回的 Pointxy 结构数组时，不记录数组的最后一项（连续直线的最后一条直线段除外）

③ PointStruct 结构数组 Pointxy1 中，记录了连续直线与多边形区域所有交点（包含了连续直线的各个顶点），以及各交点后的直线段是否在多边形区域内的判别信息。通过对各个交点后的直线段是否在多边形区域的检查，如下确定连续直线与多边形区域的相交关系：

如果连续直线与多边形区域相交后，所得到的所有直线段都在多边形区域内（pb1 == 1&& pb2 == 0），连续直线包含在多边形区域内，函数返回 1；

如果连续直线与多边形区域相交后，所得到的所有直线段都在多边形区域外（pb1 = = 0&& pb2 = = 1），连续直线在多边形区域外，函数返回 0；

如果连续直线与多边形区域相交后，所得到的直线段在多边形区域内外都有（pb1 = = 1 &&pb2 = = 1），此时连续直线与多边形区域相交，相交的详细信息通过结构数组 Pointxy1 返回，函数返回 2。

(2) 函数的测试使用

以下介绍在应用程序 Draw 中测试和使用函数 PLineRgn 的方法。在应用程序 Draw 中定义两个 CPline 类指针：

```
CPline * pline1;
CPline * pline2;
```

将这两个类指针分别指向要计算相交关系的连续直线对象和多边形区域对象。以下代码能够用来计算这两个图形元素的相交关系：

```
PointStruct * Pointxy1;
int crossNumb, numb;
//以下动态分配一个 Pointstruct 结构数组，数组的项数保证能够满足各种要求
Pointxy1 = new PointStruct[pline1 - > GetNumb() * pline2 - > GetNumb() + 2];
//以下调用 PLineRgn 函数计算连续直线与多边形区域的相交关系
numb = pline1 - > PLineRgn(pline2, &crossNumb, Pointxy1);
if(numb = = 0)      //如果连续直线在多边形区域外
    ;
else if(numb = = 1) //如果连续直线在多边形区域内
    ;
else if(numb = = 2) //如果直线与多边形区域边相交，交点数是 crossNumb - 2，各个交点坐标以及
                    //交点后的直线段是否在多边形区域内的判断信息存储在结构数组 Pointxy1 中
    ;
delete Pointxy1;    //释放动态分配的 Pointxy1 结构数组
```

在以上的测试代码中，Pointxy1 结构数组的项数为连续直线边数乘多边形区域的边数并加 2，这个容量能够满足所有正常情况的容量要求。

在应用程序 Draw 中对 PLineRgn 函数进行测试，结合图形绘制函数能够得到如图 22-2 所示的结果，一条连续直线和一个多边形区域相交，与多边形区域相交的直线段用粗线显示。

22.2　圆弧与多边形区域的关系

本节讨论圆弧与多边形区域的相交关系。因为圆是圆弧的一个特殊情况，所以对圆弧与多边形区域关系的讨论中，也就包含了对圆与多边形区域关系的讨论。

为了计算圆弧与多边形区域的关系，首先对圆与直线、圆弧与直线相交关系的计算方法进行讨论，在此基础上得到圆弧与连续多边形区域相交关系的计算方法和具体实现代码。

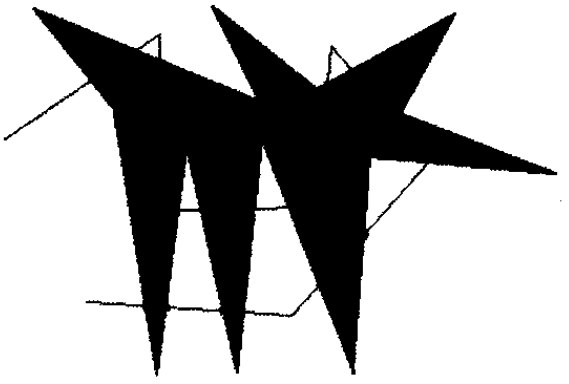


图 22-2 直线段与多边形区域相交图

22.2.1 圆与直线的关系

计算圆与直线的相交关系,最基本和最重要的是得到圆与直线的交点,从几何上,就是要得到一个圆的方程和一条直线的方程,通过解方程组得到圆与直线段的交点。

(1) 函数实现

为了计算圆与直线段的相交关系,在应用程序 Draw 的圆类 CCircle 中定义一个函数:

```
int CircleLine(CLine * line1,float * angle1,float * angle2,float * X1, float * Y1,
    float * X2, float * Y2)
```

圆的坐标存储在调用这个函数的 CCircle 对象中,直线的坐标由 CLine 类指针 line1 指向的 CLine 对象来确定。为了满足不同的需求,函数用两种方式返回了交点的坐标:交点的平面坐标和交点相对于圆心的弧度,通过参数(*X1,*Y1)和(*X2,*Y2)返回第一个和第二个交点的坐标,通过 *angle1 和 *angle2 返回第一个和第二个交点相对于圆心的弧度。根据圆和直线相交关系的不同,CircleLine 函数返回不同的值:

0: 直线段与圆不相交,直线段在圆外;
3: 直线段与圆不相交,直线段在圆内;
1-2: 直线段与圆相交,返回交点的数目。

在实现文件 drawdoc.cpp 中,加入函数的实现代码:

```
int CCircle::CircleLine(CLine * line1,float * angle1,float * angle2,float * X1, float * Y1,
    float * X2, float * Y2)
{
    float x1,y1,x2,y2;
    float xx1,yy1,xx2,yy2,k,k1,k2;
```

```
float yy,miny,maxy;
int pb;
float x,y,r;
/*
    此处加入代码检查圆的边界矩形和直线的边界矩形是否相交,如果边界矩形不
    相交,函数返回 0(直线在圆外)
*/
//得到圆的圆心坐标和半径
x = m_CircleX;
y = m_CircleY;
r = m_CircleR;
//将圆心的坐标(x,y)移到(0,0),直线的顶点坐标也做相应的改变
x1 = line1 - > m_X1 - x;
y1 = line1 - > m_Y1 - y;
x2 = line1 - > m_X2 - x;
y2 = line1 - > m_Y2 - y;
if(x1 = = x2)                    //如果直线是竖直线
{
    if(x1 * x1 > r * r)          //直线到原点(0,0)的距离大于圆的半径,直线和圆不相交
        return 0;                //函数返回 0
    else                         //直线到原点(0,0)的距离小于圆的半径
    {
        yy = (float)sqrt(r * r - x1 * x1);   //得到交点的绝对纵坐标
        yy1 = - yy; yy2 = yy;                //得到两个交点的 y 坐标
        miny = min(y1,y2);                   //得到直线段的最小纵坐标
        maxy = max(y1,y2);                   //得到直线段的最大纵坐标
        if(miny > yy1&&maxy < yy2)           //如果直线段在圆内,函数返回 3
            return 3;
        pb = 0;
        if(yy1 > = miny&&yy1 < = maxy)       //如果第一个交点在直线段上
        {
            * Y1 = yy1 + y;                  //第一个交点的纵坐标
            * angle = pi * 3/2;              //第一个交点相对圆心的弧度
            pb = 1;
        }
        //如果第二个交点在线段上,且第二个交点不等于第一个交点
        if(yy1! = yy2&&yy2 > = miny&&yy2 < = miny)
        {
            if(pb = = 1)                     //如果第一个交点已经在线段上
            {
                * Y2 = yy2 + y;              //第二个交点的纵坐标
                * angle2 = pi/2;             //第二个交点相对圆心的弧度
                pb = 2;                      //表示有两个交点
```

```
            }
            else //如果第一个交点不在直线段上,第二个交点是与直线段相交的第一个交点
            {
                * Y1 = yy2 + y;        //第一个交点的纵坐标
                * angle1 = pi/2;       //第一个交点相对圆心的弧度
                pb = 1;                //表示有一个交点
            }
        }
        return pb;                     //返回交点数目
    }
}
else                                   //如果直线段不是竖直线
{
    k = (y1 - y2)/(x1 - x2);           //得到直线段的斜率
    k1 = y1 - k * x1;
    k2 = 4 * k * k * k1 * k1 - 4 * (k * k + 1) * (k1 * k1 - r * r);    //解方程时的 b² - 4ac
    if(k2 < 0)                         //如果 b² - 4ac < 0,方程无解即没有交点,函数返回 0
        return 0;
    else          // 如果 b² - 4ac > = 0,函数有 1 个或 2 个解,直线段与圆有 1 个或 2 个交点
    {
        xx1 = ( - 2 * k * k1 - (float)sqrt(k2))/(2 * (k * k + 1));//第一个交点的横坐标
        xx2 = ( - 2 * k * k1 + (float)sqrt(k2))/(2 * (k * k + 1));//第二个交点的横坐标
        yy1 = y1 + k * (xx1 - x1);     //第一个交点的纵坐标
        yy2 = y1 + k * (xx2 - x1);     //第二个交点的纵坐标
        if(x1 > xx1&&x1 < xx2&&x2 > xx1&&x2 < xx2)      //如果直线段在两个交点内,即直
                                                        //线段在圆内
            return 3
        pb = 0;
        if(xx1 > = x1&&xx1 < = x2||xx1 < = x1&&xx1 > = x2)   //如果第一个交点是在
                                                             //直线段上
        {
            * X1 = xx1 + x;            //第一个交点的横坐标
            * Y1 = yy1 + y;            //第一个交点的纵坐标
            * angle1 = GetAngle(0,0,xx1,yy1);  //第一个交点相对圆心的弧度
            pb = 1;                    //表示有一个交点
        }
        if(k2! = 0&&xx1! = xx2&&xx2 > = x1&&xx2 < = x2||xx2 < = x1&&xx2 > = x2)
        //如果第二个交点在直线段上
        {
            if(pb = = 1)               //如果第一个交点已经在直线段上,则有两个交点
            {
                * X2 = xx2 + x;        //第二个交点的横坐标
                * Y2 = yy2 + y;        //第二个交点的纵坐标
```

```
                    * angle2 = GetAngle(0,0,xx2,yy2);  //第二个交点相对圆心的弧度
                    pb = 2;                 //表示有两个交点
                }
                else                        //如果第一个交点不在直线段上,则第二个交点
                                            //是与直线段的第一个交点
                {
                    * X1 = xx2 + x;         //第一个交点的横坐标
                    * Y1 = yy2 + y;         //第一个交点的纵坐标
                    * angle1 = GetAngle(0,0,xx2,yy2);  //第一个交点相对圆心的弧度
                    pb = 1;                 //只有一个交点
                }
            }
            return pb;                      //返回交点数目
        }
    }
    return 0;
}
```

CircleLine 函数的主要操作步骤如下：

① 将坐标原点(0,0)选在圆心上,将直线段的顶点坐标也作相应的改变。

② 判断直线段是否为竖直线。

③ 如果直线段是竖直线,解得直线与圆的交点,利用交点和直线段的坐标判断直线是否在圆内,并判断交点是否在直线段上,返回交点的数目和坐标。

④ 如果直线段不是竖直线,得到直线的斜率,通过解方程得到圆与直线的交点,判断直线是否在圆内,并判断交点是否在直线段上,返回交点的数目和坐标。

(2) 函数的测试使用

以下介绍在应用程序 Draw 中测试和使用 CircleLine 函数的方法。在应用程序 Draw 中定义两个类指针：

```
CCircle * circle1;
CLine * line1;
```

将这两个类指针分别指向要计算相交关系的圆对象和直线对象。以下代码可以完成计算圆和直线的相交关系：

```
float x1,y1,x2,y2,angle1,angle2;
//以下调用函数 CircleLine 计算圆与直线的相交关系
int numb = circle1 - > CircleLine(line1,&angle1,&angle2,&x1,&y1,&x2,&y2);
if(numb = = 0)              //如果直线段在圆外
    ;
else if(numb = = 3)         //如果直线段在圆内
    ;
else                        //直线段与圆相交,numb 为交点数目,交点的坐标为(x1,y1)和(x2,y2)
```

```
                    //交点相对于圆心的弧度为 angle1 和 angle2
    ;
```

22.2.2　圆弧与直线的关系

可以在计算圆与直线相交关系的 CirlceLine 函数基础上，得到计算圆弧与直线相交关系的函数。利用 CircleLine 函数计算得到圆弧所在的圆与直线的相交关系和交点，然后通过判断各个交点是否在圆弧上，就可以得到圆弧与直线的交点。

(1) 函数实现

为了计算圆弧与直线的相交关系，在应用程序 Draw 的圆弧类 CArc 中定义一个函数：

```
int ArcLine(CLine * line1, float * angle1, float * angle2, float * X1, float * Y1,
    float * X2, float * Y2);
```

ArcLine 函数各个参数的含义与 CircleLine 函数完全相同。唯一不同的是，函数返回的是圆弧与直线的交点数目(当直线在圆弧所在的圆内时，因为不存在直线段在圆弧内这种情况，所以 CArc 函数返回 0 而不是返回 3)。

在实现文件 drawdoc.cpp 中，加入函数的实现代码：

```
int CArc::ArcLine(CLine * line1, float * angle1, float * angle2, float * X1, float * Y1,
    float * X2, float * Y2)
{
    float x1,y1,x2,y2;
    float angl1,angl2;
    int pb,pb1;
    //以下调用函数 CircleLine 得到圆和直线段的相交情况和交点坐标
    pb = CircleLine(line1,&angl1,&angl2,&x1,&y1,&x2,&y2);
    pb1 = 0;
    if(pb = = 0 || pb = = 3)          //圆与直线段不相交，则圆弧与直线段也不相交，函数返回 0
        return 0;
    else                              //圆与直线段有交点
    {
        if(IsInArc(angl1))            //如果第一个交点在圆弧上
        {
            * X1 = x1; * Y1 = y1;     //第一个交点的坐标
            * angle1 = angl1;         //第一个交点的弧度
            pb1 = 1;                  //交点的数目
        }
        if(pb = = 2)                  //如果有两个交点
        {
            if(IsInArc(angl2))        //如果第二个交点在圆弧上
            {
```

```
                if(pb1 = = 1)          //如果已经有了一个交点
                {
                    * X2 = x2; * Y2 = y2;   //第二个交点的坐标
                    * angle2 = angl2;       //第二个交点相对于圆心的弧度
                }
                else                        //第一个交点不在圆弧上,这是圆弧与直线的
                                            //第一个交点
                {
                    * X1 = x2; * Y1 = y2;   //第一个交点的坐标
                    * angle1 = angl2;       //第一个交点相对于圆心的弧度
                }
                pb1 ++ ;                    //圆弧与直线交点的数目加 1
            }
        }
        return pb1;                         //返回交点数目
    }
    return 0;
}
```

因为 CArc 类是 CCircle 类的派生类,所以在 ArcLine 函数中可以直接调用 CCircle 类中的 CircleLine 函数,来计算圆与直线的相交关系和交点。ArcLine 函数的主要操作步骤如下:

① 调用计算圆与直线相交关系的函数 CircleLine,计算得到圆弧所在圆与直线段的相交关系和交点。

② 如果圆与直线段没有交点(pb = = 0 || pb = = 3),则圆弧与直线段也没有交点,函数返回 0。

③ 如果圆与直线段有交点,判断各个交点是否在圆弧上,把在圆弧上的交点作为圆弧与直线的交点。

④ 函数返回圆弧与直线的交点数目,交点坐标通过参数(* X1, * Y1)和(* X2, * Y2)返回,交点相对于圆心的弧度通过参数 * angl1 和 * angl2 返回。

在 ArcLine 函数中,调用了一个函数 GetAngle 来得到一个点相对于另一个基点的弧度。这个函数在应用程序 Draw 的图形元素基类 CDraw 中定义:

```
protected:
    float GetAngle(float x, float y, float xx, float yy)
```

GetAngle 函数计算得到得到并返回一个点(xx,yy)相对于另一个基点(x,y)的弧度。

在实现文件 drawdoc.cpp 中,加入这个函数的实现代码:

```
float CDraw::GetAngle(float x, float y, float xx, float yy)
{
    float r,ang;
    if(x = = xx&&y = = yy)              //如果目标点和基点相同,返回一个 0 弧度
        return 0;
    if(x = = xx)                        //如果连接两点的直线是竖直线
```

```
    {
        if(yy > y)                          //如果目标点的纵坐标大于基点的纵坐标
            ang = (float)(pi/2);            //弧度为 π/2
        else                                //如果目标点的纵坐标小于基点的纵坐标
            ang = (float)(pi * 3/2);        //弧度为 3/2π
        return ang;                         //返回相对弧度
    }
    else                                    //如果连接两点的直线不是竖直线
    {
        r = (float)sqrt((xx - x) * (xx - x) + (yy - y) * (yy - y));   //得到两点的距离
        ang = (float)acos((xx - x)/r);                                //得到弧度
        if(yy < y)                                                    //如果目标点在坐标系的
                                                                      //第三和第四个象限
            ang = (float)(pi * 2) - ang;                              //将弧度转化在 0 - 2π 范围内
        return ang;                                                   //返回相对弧度
    }
}
```

(2) 函数的测试使用

在应用程序 Draw 中，ArcLine 函数的测试和使用方法与 CircleLine 函数类似。在应用程序 Draw 中定义两个类指针：

```
CArc * arc1;
CLine * line1;
```

将这两个类指针分别指向要计算相交关系的圆弧对象和直线对象。以下的代码可以完成计算圆弧与直线的交点：

```
float x1,y1,x2,y2,angle1,angle2;
//以下调用 ArcLine 函数计算得到圆弧与直线的交点
int numb = arc1 - > ArcLine(line1,&angle1,&angle2,&x1,&y1,&x2,&y2);
if(numb = = 0)    //如果函数返回 0,圆弧与直线不相交
    ;
else              //如果函数返回非 0,圆弧与直线相交,numb 为交点数目,交点的坐标为(x1,y1)
                  //和(x2,y2),交点相对于圆心的弧度为 angle1 和 angle2
  ;
```

22.2.3　圆弧与多边形区域的关系

本小节讨论实现圆弧与多边形区域相交关系的计算。

(1) 计算思路

圆弧与多边形区域相交关系，与 22.1.2 小节讨论的直线与多边形区域的相交关系类似。

计算思路如下：

① 通过圆弧与直线关系的计算函数 ArcLine，计算得到圆弧与多边形区域所有直线段的交点(用相对于圆弧圆心的弧度值表示的交点)，将圆弧本身的起始弧度和结束弧度分别作为第一和最后一个交点，得到所有用弧度表示的交点。

② 将所有交点按照弧度值大小进行排序。

③ 得到圆弧上两个连续交点间的圆弧中点，通过检查这个点是否在多边形区域内，来确定这两个交点间的圆弧是否在多边形区域内。

④ 通过以上计算，可以得到圆弧与多边形区域边的所有交点(用相对于圆心的弧度值表示)，并且得到两个交点之间的圆弧是否在多边形区域内。

(2) 函数实现

为实现计算圆弧与多边形区域的相交关系，在应用程序 Draw 的连续直线类 CPline 中定义一个函数：

```
public:
    int ArcRgn(CArc * arc1,int * nCross,PointStruct * Pointxy1)
```

圆弧的坐标存储在 CArc 类指针 arc1 指向的圆弧对象中，连续直线的顶点坐标存储在调用这个函数的 CPline 对象中。参数 * nCross 返回圆弧与多边形区域边的交点数目(与多边形区域边的交点加上圆弧的起终点)，Pointxy1 指向的 PointStruct 结构数组返回各交点信息：PointStruct 结构的成员变量 x 用来记录各交点相对于圆心的弧度，成员变量 z 用来表示以这个顶点开始的圆弧段是否在多边形区域内。根据不同的相交情况，函数返回不同的值：

0：圆弧与多边形区域不相交，圆弧在多边形区域外；

1：圆弧在多边形区域内；

2：圆弧与多边形区域的边相交。

在实现文件 drawdoc.cpp 中，加入函数的实现代码：

```
int CPline::ArcRgn(CArc * arc1,int * nCross,PointStruct * Pointxy1)
{
    int i,numb,nn,pb;
    float ang1,ang2,x1,y1,x2,y2;
    float angle[400];
    CLine line1;
    float angl1,angl2;
    /*
        此处加入代码检查圆弧的边界矩形和多边形区域的边界矩形是否重叠，如果边界矩形不
        重叠，圆弧在多边形区域外，函数返回 0
    */
    numb = 0;                          //圆弧与多边形区域边交点的数目为 0;
    angl1 = arc1 -> m_Angle1;          //圆弧的起始弧度
    angl2 = arc1 -> m_Angle2;          //圆弧的结束弧度
```

```
angle[numb++] = angl1;                          //将圆弧的起始弧度作为第一个交点
//在多边形顶点坐标后增加一个顶点,封闭多边形区域
m_PointList[m_Numble].x = m_PointList[0].x;
m_PointList[m_Numble].y = m_PointList[0].y;
for(i = 0;i < m_Numble;i++)                     //对多边形区域所有的直线边循环
{
    line1.m_X1 = m_PointList[i].x;              //得到直线的起点横坐标
    line1.m_Y1 = m_PointList[i].y;              //得到直线的起点纵坐标
    line1.m_X2 = m_PointList[i+1].x;            //得到直线的终点横坐标
    line1.m_Y2 = m_PointList[i+1].y;            //得到直线的终点纵坐标
    nn = arc1 -> ArcLine(&line1,&angl,&ang2,&x1,&y1,&x2,&y2); //计算圆弧与直线的交点
    if(nn >= 1)                                 //如果存在交点
    {
        angle[numb++] = angl;                   //将交点的弧度存数组 angle
        if(nn == 2)                             //如果有两个交点
            angle[numb++] = ang2;               //将第二个交点的弧度存在数组 angle
    }
}
angle[numb++] = angl2;                          //圆弧的终点作为最后一个交点
if(angl1 > angl2)                               //如果圆弧是跨 X 轴正向的
{
    for(i = 0;i < numb;i++)                     //对所有的交点循环
    {
        if(angle[i] >= 0&&angle[i] <= angl2) //在 X 轴正向到圆弧结束部分的交点,弧度值
                                                //加 2π,以利于排序
            angle[i] = (float)pi * 2 + angle[i];
    }
}
//以下按弧度值由小到大对 angle 数组进行排序
do
{
    pb = 0;
    for(i = 0;i < numb - 1;i++)                 //对所有的交点循环
    {
        if(angle[i] > angle[i+1])   //如相临两个交点的弧度值顺序相反,将这两个交点互换
        {
            pb = 1;                 //表示各个交点还有存在倒序排列的
            angl = angle[i];
            angle[i] = angle[i+1];
            angle[i+1] = angl;
        }
    }
}while(pb == 1);                    //如果还有倒序的,再进行排序操作
```

```
    pb = 0;
    for(i = 0;i < numb - 1;i++ )                    //对所有的交点循环(最后一个交点除外)
    {
        Pointxy1[i].x = angle[i];                   //将交点的弧度值记录在结构数组 Pointxy1 中
        ang1 = (angle[i] + angle[i + 1])/2;         //得到两个相临交点间圆弧的中点
        x1 = arc1 - > m_ CircleX + arc1 - > m_ CircleR * (float)cos(ang1);   //得到中点的横坐标
        y1 = arc1 - > m_ CircleY + arc1 - > m_ CircleR * (float)sin(ang1);   //得到中点的纵坐标
        if(PointRgn(x1,y1,m_ Numble,m_ PointList,1))                    //如果中点在多边形区域内
            Pointxy1[i].z = 1;                      //表示两相临交点间的圆弧在多边形区域内
        else                                        //如果中点不在多边形区域内
            Pointxy1[i].z = 0;                      //表示两相临交点间的圆弧不在多边形区域内
    }
    Pointxy1[numb - 1].x = angle[numb - 1];         //最后一个交点的弧度值
    * nCross = numb;                                //交点的数目
    if(numb = = 2)      //如果只有两个顶点(圆弧的起点和终点),即圆弧与多边形区域边不相交
    {
        if(Pointxy1[0].z = = 1)                     //如果圆弧在多边形区域内
            return 1;                               //圆弧在多边形区域内,返回 1
        else                                        //如果圆弧不在多边形区域内
            return 0;                               //圆弧与多边形区域不相交,返回 0
    }
    else                                            //如果有多于二个交点
        return 2;                                   //圆弧与多边形区域边相交,返回 2
}
```

ArcRgn 函数的主要操作步骤如下：

① 对一个多边形区域边界的所有直线段循环，利用 ArcLine 函数分别计算出圆弧与各个直线段的交点，再加上圆弧的起始和结束点，得到圆弧与多边形区域的所有交点，交点的数目为 numb(numb > = 2)，将各个交点相对于圆心的弧度值记录在数组 angle。

② 当圆弧跨越 X 轴正方向时，为了把各个交点按照弧度由小到大进行排序。对存储在数组 angle 中的各个交点的弧度值进行处理，把在 X 轴正向到圆弧结束这一段圆弧上的交点的弧度值加 2π。

③ 对所有交点按照弧度值由小到大进行排序。

④ 对所有的交点循环，得到相临两个交点之间圆弧中点的坐标，调用函数 PointRgn 判断中点是否在多边形区域内，从而确定相临两个交点间的圆弧段是否在多边形区域内。

④ 判断圆弧与多边形区域的相交关系，返回函数的值。

(3) 函数的测试使用

以下介绍在应用程序 Draw 中测试和使用 ArcRgn 函数的方法。在应用程序 Draw 中定义两个类指针：

```
CArc * arc1;
```

```
CPline * pline1;
```

将两个类指针分别指向要计算相交关系的圆弧和多边形区域对象。以下代码可以计算圆弧对象与连续直线对象的相交关系：

```
PointStruct * Pointxy1;
int crossNumb,numb;
//动态分配一个 PointStruct 数组,用来返回圆弧与多边形区域的交点信息
Pointxy1 = new PointStruct[pline1 - > GetNumb() * 2 + 2];
numb = pline1 - > ArcRgn(arc1,&crossNumb,Pointxy1); //调用 ArcRgn 函数,计算圆弧与多边形区域
                                                    //的相交关系
if(numb = = 0)          //如果函数返回 0,圆弧在多边形区域外
    ;
else if(numb = = 1)     //如果函数返回 1,圆弧在多边形区域内
    ;
else if(numb = = 2)     //如果函数返回 2,圆弧与多边形区域边界相交,交点数目为 crossNumb - 2,
                        //交点相对于圆心的弧度和以这个交点开始的圆弧是否在多边形区域的判
                        //别信息,存储在结构数组 Pointxy1 中
    ;
delete Pointxy1;        //释放动态分配的结构数组
```

以上测试代码中,动态分配了一个 PointStruct 结构数组 Pointxy1,来存储圆弧与多边形区域的相交信息,数组的项数设置为多边形区域的边数乘 2 加 2,可保证满足任何相交情况的容量要求。

结合图形绘制函数可得到如图 22-3 所示的相交结果图,粗线部分表示的是圆弧相交于多边形区域的部分。



图 22-3 圆弧与多边形区域相交图

22.3 图形元素与圆形区域的关系

本节讨论各类基本图形元素与圆形区域的相交关系。在以上各节讨论各类图形元素与多边形区域的关系时,已经讨论实现了直线与圆、圆弧与多边形区域相交关系的计算函数,以这些函数为基础,可以实现直线以及连续直线与圆形区域相交关系的计算函数。所以,本节的重

点放在讨论实现圆弧与圆形区域相交关系的计算上。

22.3.1　圆与圆的关系

为了计算圆弧与圆形区域的相交关系，需要首先解决圆和圆相交关系的计算。

(1) 函数实现

两个圆之间的相交关系，可以分为如下几种类型：
① 不相交。两个圆不相交时，有下列三种情况：
第一个圆在第二个圆内；
第二个圆在第一个圆内；
两个圆相互分离。
② 相切(即只有一个交点)。两个圆相切的情况也有三种：
第一个圆内切于第二个圆；
第二个圆内切于第一个圆；
两个圆外切
③ 相交(有两个交点)。
为了计算圆与圆的相交关系，在应用程序 Draw 的圆类 CCircle 中定义一个函数：

```
protected:
    int CircleCircle(CCircle * cir1,float * angl1,float * angl2)
```

第一个圆的坐标由调用这个函数的圆对象来确定，第二个圆的坐标存储在参数 cir1 指向的圆对象中。参数 * angl1 和 * angl2 用来返回两个圆的交点相对于第一个圆圆心的弧度。对于两个圆之间的不同相交关系，函数返回以下的值：

0：　两个圆分离不相交；
10：　第一个圆在第二个圆内；
20：　第二个圆在第一个圆内；
11：　第一个圆内切于第二个圆，参数 * angle1 返回切点相对于第一个圆圆心的弧度；
21：　第二个圆内切于第一个圆，参数 * angle1 返回切点相对于第一个圆圆心的弧度；
31：　两个圆外切，参数 * angle1 返回切点相对于第 1 个圆圆心的弧度；
2：　两个圆相交，参数 * angle1 和 * angle2 返回两个交点相对于第一个圆圆心的弧度

在实现文件 drawdoc.cpp 中，加入函数的实现代码：

```
int CCircle::CircleCircle(CCircle * cir1,float * angl1,float * angl2)
{
    float jl,ang,angl;
    float x1,y1,r1,x2,y2,r2;
    x1 = m_CircleX;                    //第一个圆的圆心横坐标
    y1 = m_CircleY;                    //第一个圆的圆心纵坐标
    r1 = m_CircleR;                    //第一个圆的半径
```

```
    x2 = cir1 - > m_CircleX;          //第二个圆的圆心横坐标
    y2 = cir1 - > m_CircleY;          //第二个圆的圆心纵坐标
    r2 = cir1 - > m_CircleR;          //第二个圆的半径
    ang = GetAngle(x1,y1,x2,y2);      //第二个圆圆心相对第一个圆圆心的弧度
    jl = (float)sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));  //计算两个圆心间的距离
    if(jl < r2 - r1)                  //如果第二个圆半径减第一个圆半径大于两个圆心的距离,则
                                      //第一个圆在第二个圆内,函数返回 10
        return 10;
    else if(jl + r1 = = r2) //如果第一个圆的半径加两个圆心的距离等于第二个圆的半径,则第一个圆
                      //内切于第二个圆,得到切点相对于第一个圆心的弧度,函数返回 1
    {
        * angl1 = ang;  //切点相对于第一个圆圆心的弧度
        return 11;
    }
    else if(r1 - r2 > jl)   //如果第一个圆半径减第二个圆半径大于两个圆心的距离,则第二个圆在
                      //第一个圆内,函数返回 20
        return 20;
    else if(jl + r2 = = r1)//如果第二个圆的半径加两个圆心的距离等于第一个圆的半径,则第二个
                      //圆内切于第一个圆,得到切点相对于第一个圆心的弧度,函数返回 21
    {
        * angl1 = ang;  //返回相对于第一个圆心的弧度
        return 21;
    }
    else if(r2 + r1 < jl)   //两个圆的半径和小于两个圆心的距离,两个圆分离没有交点,函数返回 0
        return 0;
    else if(r2 + r1 = = jl)//两个圆的半径和等于两个圆心的距离,两个圆外切,函数返回 31
    {
        * angl1 = ang;  //返回切点相对于第一个圆心的弧度
        return 31;
    }
    else                  //否则,两个圆相交,返回两个交点相对于第一个圆圆心的弧度
    {
        //得到交点到第一个圆圆心的直线与两个圆心连线的交角
        angl = (float)acos((r1 * r1 + jl * jl - r2 * r2)/(2 * r1 * jl));
        * angl1 = ang - angl;             //得到一个交点相对于第一个圆圆心的弧度
        if( * angl1 < 0)                  //如果弧度小于 0,将弧度值增加 2π
            * angl1 = (float)pi * 2 + * angl1;
            * angl2 = ang + angl;         //得到第二个交点相对于第一个圆圆心的弧度
            if( * angl2 > 2 * pi)         //如果弧度小于 0,将弧度值增加 2π
                * angl2 = * angl2 - (float)pi * 2;
            return 2;                     //有两个交点,函数返回 2
    }
}
```

CircleCircle 函数的操作步骤比较简单，读者可从函数代码的详细注释中得到函数的实现步骤。

(2) 函数的测试使用

以下介绍在应用程序 Draw 中测试和使用 CircleCircle 函数的方法。在应用程序 Draw 中定义两个类指针：

```
CCircle * circle1;
CCircle * circle2;
```

将这两个类指针分别指向要计算相交关系的两个 CCircle 对象。以下代码能够完成计算圆与圆的相交关系：

```
float angle1,angle2;
//以下调用 CircleCircle 函数,计算圆与圆的相交关系
int ret = circle1 - > CircleCircle(circle2,&angle1,&angle2);
if(ret = = 0)          //如果函数返回 0,两个圆分离不相交
    ;
else if(ret = = 10)    //如果函数返回 10,第一个圆在第二个圆内
    ;
else if(ret = = 20)    //如果函数返回 20,第二个圆在第一个圆内
    ;
else if(ret = = 11)    //如果函数返回 11,第一个圆内切于第二个圆,angle1 返回切点相对
                       //第一圆心的弧度
    ;
else if(ret = = 21)    //如果函数返回 21,第二个圆内切于第一个圆,angle1 返回切点相对
                       //第一圆心的弧度
    ;
else if(ret = = 31)    //如果函数返回 31,两个圆外切,angle1 返回切点相对第一个圆圆心的弧度
    ;
else if(ret = = 2)     //如果函数返回 2,两个圆相交,angle1 和 angle2 返回交点相对
                       //第一个圆心的弧度
    ;
```

22.3.2　圆弧与圆形区域的关系

在 22.3.1 小节中实现的圆与圆相交关系的计算函数 CircleCircle 基础上，可以较容易地实现圆弧与圆形区域相交关系的计算函数。

(1) 函数实现

为了计算圆弧与圆形区域间的相交关系，在应用程序 Draw 的圆弧类 CArc 中，定义一个函数如下：

```
public:
    int ArcCircle(CCircle * cir1,int * nCross,PointStruct * Pointxy1);
```

圆弧的坐标由调用这个函数的 CArc 对象确定,圆形区域的坐标由参数 cir1 指向的 CCircle 对象来确定。参数 * nCross 返回圆弧与圆形区域交点的数目(包含了圆弧的起始和结束点),参数 Pointxy1 指向的 PointStruct 结构数组用来返回各个交点的信息,PointStruct 结构的成员变量 x 用来记录各交点相对于圆弧圆心的弧度,成员变量 z 用来判断以这个交点开始的圆弧段是否在圆形区域内。根据不同的情况,函数返回不同的值:

0: 圆弧与圆不相交,圆弧在圆形区域外;
1: 圆弧与圆不相交,圆弧在圆形区域内;
2: 圆弧与圆相交。

在实现文件 drawdoc.cpp 中,加入函数的实现代码:

```
int CArc::ArcCircle(CCircle * cir1,int * nCross,PointStruct * Pointxy1)
{
    float ang1,ang2,angle[5],x1,y1,r;
    int pb,numb,i;
    numb=0;
    angle[numb++]=m_Angle1;                    //将圆弧的起点作为第一个交点
    pb=CircleCircle(cir1,&ang1,&ang2);         //调用 CircleCircle 函数来计算圆和圆的相交关系
    if(pb==1||pb==11||pb==21)                  //如果圆弧所在的圆和圆形区域有一个交点
    {
        if(IsInArc(ang1))      //如果交点在圆弧上,记录这个交点相对于圆弧圆心的弧度
            angle[numb++]=ang1;
    }
    else if(pb==2)             //如果圆弧所在的圆和圆形区域有第二个交点
    {
        if(IsInArc(ang1))      //如果第一个交点在圆弧上,记录这个交点相对于圆弧心的弧度
            angle[numb++]=ang1;
        if(IsInArc(ang2))      //如果第二个交点在圆弧上,记录这个交点相对于圆弧圆心的弧度
            angle[numb++]=ang2;
    }
    angle[numb++]=m_Angle2;          //将圆弧的终点作为最后一个交点
    if(m_Angle1>m_Angle2) //如果圆弧的起始弧度大于结束弧度,即圆弧跨越 X 轴正方向
    {
        for(i=0;i<numb;i++)          //对所有的交点循环
        {
            if(angle[i]>=0&&angle[i]<=m_Angle2)  //如果交点在从 X 轴正向开始到圆弧
                                                 //结束段
                angle[i]=(float)pi*2+angle[i];   //将交点的弧度增加 2π
        }
    }
```

```
    pb = 0;
    for(i = 0;i < numb - 1;i++ )                    //对所有的交点循环(最后一个交点除外)
    {
        Pointxy1[i].x = angle[i];                   //记录交点相对于圆弧圆心的弧度
        angl = (angle[i] + angle[i + 1])/2;         //相临两个交点间圆弧段中点的弧度
        x1 = m_CircleX + m_CircleR * (float)cos(angl); //中点的横坐标
        y1 = m_CircleY + m_CircleR * (float)sin(angl); //中点的纵坐标
        r = (float)sqrt((x1 - cirl - > m_CircleX) * (x1 - cirl - > m_CircleX) + (y1 - cirl - > m_Cir-
            cleY) * (y1 - cirl - > m_CircleY));     //计算圆弧段中点到圆形区域圆心的距离
        if(r < = cirl - > m_CircleR)                //如果距离小于圆形区域的半径,则中点在圆形区域内
            Pointxy1[i].z = 1;                      //表示以第 i 个交点开始的圆弧段在圆形区域内
        else                                        //如果距离大于圆形区域的半径,则中点在圆形区域外
            Pointxy1[i].z = 0;                      //表示以第 i 个交点开始的圆弧段在圆形区域外
    }
    Pointxy1[numb - 1].x = angle[numb - 1];         //记录下圆弧的终点作为最后一个交点
    * nCross = numb;                                //交点的数目
    if(numb = = 2)                                  //如果只有两个顶点,即圆弧与圆不相交
    {
        if(Pointxy1[0].z = = 1)                     //如果圆弧在圆形区域内
            return 1;                               //圆弧在圆形区域内,函数返回 1
        else                                        //如果圆弧不在圆形区域内
            return 0;                               //圆弧与圆形区域不相交,函数返回 0
    }
    else                                            //如果有多于二个交点,则圆弧与圆相交,函数返回 2
        return 2;
}
```

ArcCircle 函数的主要操作步骤如下：

① 通过 CircleCircle 函数得到圆弧所在圆与圆形区域的交点，通过判断这些交点是否在圆弧上(用 CArc 类的 IsInArc 函数)来得到圆弧与圆形区域的交点，再加上圆弧的起始点和结束点，得到圆弧与圆形区域的所有交点。交点的数目为 numb(numb > =2)，相对于圆弧圆心的弧度记录在数组 angle。

② 当圆弧跨越 X 轴正方向时，对数组 angle 中存储的弧度值进行处理，对在 X 轴正向到圆弧结束这一圆弧段交点的弧度值加 2π。

③ 得到相临两个交点间的圆弧段中点的坐标，通过判断中点是否在圆形区域内，来确定这两个交点间的圆弧段是否在圆形区域内。

④ 得到圆弧与圆形区域的相交关系，返回函数的值。

(2) 函数的测试使用

以下介绍在应用程序 Draw 中测试和使用 ArcCircle 函数的方法。在程序 Draw 中定义两个类指针：

```
CArc * arc1;
CCircle * circle1;
```

将这两个类指针分别指向要计算相交关系的圆弧类 CArc 和圆形区域类 CCircle 对象。以下代码能够计算出圆弧与圆形区域的相交关系：

```
PointStruct Pointxy1[5];      //定义一个 PointStruct 结构数组
int crossNumb;
//以下调用 ArcCircle 函数,来计算圆弧与圆形区域的相交关系
int ret = arc1 - > ArcCircle(circle1,&crossNumb,Pointxy1);
if(ret = = 0)                 //如果函数返回 0,则圆弧在圆形区域外
    ;
else if(ret = = 1)            //如果函数返回 1,则圆弧在圆形区域内
    ;
else if(ret = = 2)            //如果函数返回 2,圆弧与圆相交,交点数 crossNumb - 2,各个交点相对于
                              //圆弧圆
                              //心的弧度,以及交点后的圆弧段是否在圆形区域的判别信息,存在
                              //Pointxy1 中
    ;
```

因为一个圆弧与一个圆最多有二个交点，加上圆弧的起点和终点，最多有四个顶点，所以直接定义了一个固定项数的 PointStruct 结构数组 Pointxy1 来返回相交信息。

结合图形绘制函数，可以得到如图 22-4 所示的相交结果，粗线部分是圆弧相交于圆形区域的部分。

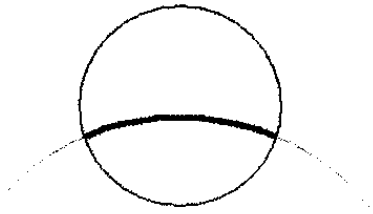


图 22-4 圆弧与圆形区域相交图

22.3.3 直线与圆形区域的关系

在 22.2 小节，在 CCircle 类中实现了计算直线和圆相交关系的 CircleLine 函数。在这个函数基础上，可以实现计算一条直线与一个圆形区域相交关系的函数，计算得到直线与圆的交点，并得到以各个交点开始的直线段是否在圆形区域内。在直线与圆形区域相交关系的函数基础上，可以完成连续直线与圆形区域相交关系的函数。受篇幅所限，不再列出这两个函数的实现代码，读者可参考 22.1 节在连续直线类 CPline 中实现的 LineRgn 和 PLineRgn 函数，自己组织实现这两个函数。

本章在讨论实现各个图形元素间相交关系的基础上，讨论实现了各类基本图形元素与多边形区域、圆形区域的相交关系计算函数。除此之外，还有其他诸如各类图形元素与椭圆区域相交关系的计算函数、其他类型曲线与各个图形元素、区域相交关系的计算函数等。对于这些相交关系，读者可根据图形元素的几何特征(坐标方程)具体分析实现，也可以用连续直线或多边形区域来近似表示这些图形元素，用本章介绍的各个函数来实现。

除了需要计算线类图形元素与区域类图形元素的相交关系外，在 GIS 系统中，还需要计算区域类图形元素间的相交关系，如两个多边形区域的相交关系(相交后得到的相交区域、不相交区域等)、圆形区域与多边形区域的相交关系等。对于区域间相交关系的计算，可根据图形元素本身的几何特征，利用 MFC 的区域类 CRgn 提供的一些成员函数去组织完成。对于这类计算，本书不再具体讨论。另外，本章所讨论的空间拓扑关系计算全部是针对二维(平面)图形元素的。

第 23 章　GIS 系统空间信息统计和分析功能

空间信息统计和分析功能是 GIS 系统的一个基本功能，其本质就是利用图形元素的空间拓扑关系，通过图形元素来实现对信息数据的统计和分析。

23.1　空间信息统计和分析的实现步骤

本节介绍在 GIS 系统中实现空间信息统计和分析功能的方法和实现步骤。

(1) 信息统计和分析实现步骤

在 GIS 系统中，空间信息统计和分析是通过以下两个步骤来完成的：

① 第一步是在矢量图形上，按照各种图形元素的拓扑关系得到需要统计和分析的矢量图形元素。也就是根据图形元素的空间位置，来确定各种条件下能够选择得到的图形元素。

② 第二步是对选择的图形元素的性质数据信息进行统计和分析。也就是通过所选择的图形元素，得到各个图形元素所连接的性质数据信息(数据连接单元)，对这些性质数据信息实现统计和分析的操作功能。

(2) 选择图形元素

为了进行空间信息统计和分析功能，首先需要得到一定条件下选择(或通过计算得到)的矢量图形元素。矢量图形元素的选择，可以采用各种不同的方法：8.1 节中实现了对图形元素的点选操作；在第 22 章中，讨论实现了各种图形元素间相交关系的计算函数，利用这些函数，可以实现诸如区域选择、线类相交选择等多种选择矢量图形元素的方法；此外，还能根据图形元素的相交关系，计算得到图形元素相交操作后得到的新图形元素，例如，连续直线与多边形区域相交后能够得到多个直线段，两个区域相交后能够得到相交区域等。

不论是采用哪种选择(或计算)矢量图形元素的方法，最后得到的都是与操作有关的各个矢量图形元素。例如，在 8.1 小节点选图形元素时，得到了选中图形元素的类别、序号及识别号，同样可以用其他方法来选择得到图形元素的类别、序号及识别号：用区域选择图形元素时，可以得到包含在某个区域内的所有图形元素；用线类相交操作选择图形元素时，得到与指定线类图形元素相交的的图形元素；在线类相交区域操作中，能够计算得到指定线类图形元素与区域相交后得到的新图形元素；而在区域相交操作时，能够得到区域与区域类图形元素相交后得到的新区域类图形元素。

GIS 系统中，在进行空间信息统计和分析操作时，首先根据具体的操作要求，采用相应的方法，从矢量图形中选择并记录与操作有关的图形元素。

(3) 信息统计和分析

GIS 系统的空间信息统计和分析功能，并不是为了对矢量图形元素进行统计和分析，而是以矢量图形元素作为操作的中介，实现对性质数据的统计和分析功能。也就是说，要统计分析的是记录在数据库管理系统的数据表中的性质数据信息。

在通过空间统计(或计算)得到与操作有关的各个图形元素后，就可以根据这些图形元素与数据连接单元的连接，得到赋予在这些图形元素上的性质数据信息。从 19.3.1 小节的讨论可知，一个连接是通过以下五个参数来描述的：

```
typedef struct
{
    unsigned char lb;          //连接的方式：1－连接数据表的一条记录，
                               //            2－连接数据表的一个记录集合
    unsigned char graph_id;    //图形类别的识别号
    int graph_element_id;      //图形元素的识别号
    short database_id;         //记录集合对象的识别号
    int record_id;             //连接方式 1 下，表示记录的识别号，在连接方式 2 下，
                               //表示过滤条件式的识别号
} LinkStru;
```

通过选择矢量图形元素，也就是说已知了图形元素的类别(graph_id)和识别号(graph_element_id)的情况下，就可以得到与一个图形元素相连接的性质数据信息。在获得对于统计和计算后得到的所有图形元素的性质数据信息后，就可以根据具体的功能要求，采用不同的方法，任意对这些数据信息进行各种统计、分析操作，实现信息统计和分析功能。更主要的是，只要得到了性质数据信息，就可以对数据信息进行任何要求的处理，实现各种实用的信息管理、辅助决策、辅助设计等系统，从而达到结合 GIS 技术，实现各类实用系统的目的。

以下各节，将讨论对矢量图形元素进行统计或计算的几种常用方法，并介绍这些方法在 GIS 系统信息统计和分析操作中的一些具体应用。

23.2　区域信息统计和分析

所谓区域信息统计和分析，就是在 GIS 系统的矢量图形上确定一个区域，得到这个区域内所包含的矢量图形元素，通过这些矢量图形元素，统计和分析这个给定区域内的性质数据信息。实现区域信息统计和分析功能的一个关键问题，是能够得到包含在一给定区域内的矢量图形元素，通过各个图形元素得到其所连接的性质数据信息，并实现通常的或有针对性的数据信息统计和分析功能。

本节讨论几种常用的区域选择图形元素的方法，以及这些方法在 GIS 系统中的一些应用。

23.2.1　区域选择图形元素的方法

对图形元素进行区域选择时，根据给定区域形状的不同，有不同的选择方法。

(1) 矩形区域选择

最简单和最常用的区域选择方法就是矩形区域选择(或称为窗口选择)。即在矢量图形中确定一个矩形区域,然后对矢量图形的所有图形元素进行比较运算,来确定包含在矩形区域内的图形元素。

矩形区域选择方法,本质上是计算各种图形元素与区域矩形(四边形)的相交关系。矩形是一个特殊的多边形区域,所以利用在 22.1 节和 22.2 节中介绍的各类图形元素与多边形区域相交关系的计算函数,就可以较容易地实现矩形区域选择操作。

(2) 多边形区域选择

多边形区域选择,就是在矢量图形中确定一个任意多边形区域,得到包含在多边形区域内的矢量图形元素。在 22.1 和 22.2 节中,讨论实现了各类图形元素与多边形区域的相交关系计算函数,利用这些计算函数对矢量图形的各个图形元素进行进行比较运算,就可以确定能够包含在多边形区域内的图形元素。

(3) 圆形区域选择

圆形区域选择,就是在矢量图形中确定一个圆形区域,得到包含在这个圆形区域内的矢量图形元素。在 22.3 节中,讨论实现了各种图形元素与圆形区域相交关系的计算函数,利用这些计算函数对矢量图形的各个图形元素进行比较运算,就可以确定包含在圆形区域内的图形元素。

(4) 圆弧区域选择

圆弧区域选择就是在矢量图形中确定一个圆弧区域(一条圆弧及其起终点的连线确定的区域),得到包含在圆弧区域内的图形元素。在第 22 章中介绍的圆弧与各类图形元素相交关系计算函数的基础上,可以编制出各类图形元素与圆弧区域相交关系的计算函数,利用这些计算函数对矢量图形的各个图形元素进行比较运算,来确定能够包含在圆弧区域内的矢量图形元素。

(5) 椭圆弧和其他曲线区域选择

除了圆弧区域外,还有椭圆弧或其他类型的曲线区域,可以利用这些曲线区域去选择图形元素。对于这些曲线区域,一方面可以根据这些曲线的几何特征,编制实现各种图形元素与这些曲线区域相交关系的计算函数,利用这些计算函数通过比较运算来选择图形元素;另一方面,也可以利用这些曲线的近似多边形,利用多边形区域来代替这些曲线区域来完成选择操作,只要给定足够多的插入点得到近似多边形,就可以达到多边形区域与这些曲线区域的高度近似,满足操作的精度要求。具体的实现过程不再讨论。

(6) 混合区域选择

所谓混合区域，指的是由直线段和曲线段连线所包含的区域。以上讨论的圆弧区域就是一个特殊的混合区域，在 AutoCAD 中的 pline 线可以包含直线段和曲线段(圆弧)，其围成的区域可以看作是一种混合区域。混合区域选择，就是在矢量图形上确定一个混合区域，通过计算得到包含在这个区域内的图形元素。

对于混合区域，可以把它分解成多个多边形区域和其他曲线区域，利用既有的多边形区域和曲线区域选择方法组织选择操作。也可以采用更简单易行的方法：将混合区域用高度近似的多边形区域来代替，利用多边形区域的选择操作来代替混合区域的选择操作。

(7) 三维区域选择

在本书的讨论中，图形元素和区域选择图形元素的方法都是二维的。实现完全的三维矢量图形，特别是完全的三维区域选择操作，实现方法非常复杂。在日常的应用程序设计中，在很多情况下可以采用在二维区域选择操作中，与图形元素的 Z 坐标相结合，实现考虑 Z 坐标因素的区域选择。考虑 Z 坐标因素的区域选择比较简单，只是对二维区域选择的简单扩充。对于用二维区域选择的图形元素，再对其 Z 坐标进行比较计算，通过检查这些图形元素的 Z 坐标是否处于一定的 Z 坐标范围内，来确定是否选择了这些图形元素。

23.2.2　区域信息统计和分析

23.2.1 小节介绍了几种常用的区域选择图形元素的方法，不管是哪种区域选择方法，最后得到的都是包含在区域内的矢量图形元素，针对这些图形元素所连接的性质数据信息，可以完成一些常用的统计和分析功能。另外，根据系统的具体要求，可以有针对性地完成数据信息的统计和分析功能。

(1) 区域信息统计

在 GIS 系统中，可以以 23.2.1 小节中介绍的区域选择方法为基础，组织实现一些具体的区域信息统计和分析功能，以下介绍一些常用的功能实例。

在一个全国基本信息 GIS 系统中，需要完成诸如“离某个点 100 公里内范围的城镇、矿产地……”等统计功能，实现这些统计功能的基础，就是圆形区域选择操作。选择矢量图形中某个圆形区域内的图形元素，并对图形元素所连接的性质数据信息进行统计。

再比如，在全国基本信息 GIS 系统中，一条河流用一条连续直线近似表示，在完成诸如“沿河两岸 30 公里内的城镇、资源、人口……”等统计功能时，实现这些功能的基础，是多边形区域选择功能，选择矢量图形中计算得到的多边形区域中的矢量图形元素，并对矢量图形元素所连接的性质数据信息进行统计。

另外，在 GIS 系统中完成统计功能时，可以把各种区域选择方法结合使用，完成较为复杂的区域信息统计功能。可以采用“与”、“或”等连接符来组合各个区域选择。例如，在全国基本信息 GIS 系统中，完成“离某个点 100 公里内，且在某条河 30 公里以内的城镇”统计时，就可以通过将圆形区域选择和多边形区域选择两种选择方式通过“与”结合起来，得到指定区域内的

矢量图形元素(也可以通过区域的"与"运算,确定出圆形区域与多边形形区域的相交区域,采用混合区域选择图形元素)。在选择得到各种矢量图形元素后,就可以对矢量图形元素所连接的性质数据信息进行统计。

(2) 区域信息分析

除了对选择的图形元素的性质数据信息进行统计操作外,可以根据具体的情况,完成一些具体的分析操作功能。

例如,在全国基本信息 GIS 系统中,可以完成诸如"某个区域内的人口密度"、"某个区域的交通网密度"、"某个区域内的人均资源情况"等分析功能,所有这些分析功能都是在区域选择的图形元素基础上,对图形元素所连接的信息进行分析计算而得到的。

针对不同的实用 GIS 系统,可以实现一些具体的区域分析功能。所有这些分析功能都是以区域选择操作为基础,对选择得到的矢量图形元素连接的性质数据信息进行分析计算得到的。

23.3 线类相交信息统计和分析

所谓线类相交信息统计和分析,就是在 GIS 系统的矢量图形中确定一条或多条线类图形元素,通过计算矢量图形中的所有图形元素与线类图形元素的相交关系,来选择或计算得到与线类图形元素相交的图形元素,然后对这些图形元素所连接的性质数据信息进行统计和分析操作。

本节介绍线类相交选择图形元素的方法,并介绍在 GIS 系统中具体实现的一些线类相交信息统计和分析功能。

23.3.1 线类相交选择图形元素的方法

线类相交选择图形元素,指的是在矢量图形中确定一个线类图形元素,通过检查矢量图形中的各个图形元素是否与这个线类图形元素相交,来实现对图形元素的选择操作。

根据选择图形元素时采用的线类图形元素不同,可以有以下几种常用的相交选择图形元素的方法。

(1) 直线相交选择

直线相交选择,指的是在图形系统中确定一条直线,通过计算矢量图形中的所有图形元素与这条直线的相交关系,得到与这条直线相交的矢量图形元素。

在第 22 章中,讨论实现了直线与各种图形元素相交关系的计算函数,通过这些函数可以很容易地得到一条直线能够选择的图形元素。

(2) 连续直线相交选择

连续直线是多条直线段的组合,通过对连续直线的每条直线段进行相交选择操作,可以得到一条连续直线相交选择的矢量图形元素。

(3) 圆和圆弧相交选择

圆和圆弧相交选择，指的是在矢量图形中确定一个圆弧(圆是圆弧的特殊情况)，通过对矢量图形中的各个图形元素与这个圆弧相交关系的计算，来选择与这个圆弧相交的图形元素。

在第 22 章中，讨论实现了各类图形元素与圆弧相交关系的计算函数，通过这些函数可以得到一个圆弧与其他图形元素的相交关系，从而确定与一个圆弧相交的矢量图形元素。

(4) 椭圆弧和其他曲线相交选择

除了圆弧外，还有椭圆弧等其他类型的各种曲线，可以通过这些曲线相交选择矢量图形中的图形元素。对于这些曲线与各类图形元素的相交关系，一方面可以根据这些曲线的几何特征，编制函数来计算得到；另一方面，也可以利用这些曲线的近似连续直线，利用连续直线与各类图形元素相交关系的计算函数来得到。对于这些曲线，在得到近似连续直线时，只要给定足够多的插入点，就可以满足用连续直线代替这些曲线时精度方面的要求。

(5) 混合线选择

所谓混合线，指的是由直线段和曲线段连接成的连续线。对于混合线，可以把它分解成多个连续直线和其他曲线，利用连续直线和其他曲线的相交选择操作来实现混合线的选择操作。也可以采用更简单易行的方法：将混合线用高度近似的连续直线来代替，利用连续直线的相交选择操作来代替混合线的相交选择操作。

23.3.2　线类相交区域计算

在 GIS 系统中，不但需要通过线类相交选择图形元素，得到线类图形元素与哪些图形元素相交。在很多情况下，还要计算得到与区域类图形元素的具体相交情况。例如，在 22.1 节中，讨论实现了各种图形元素与多边形区域的相交关系计算，当一条直线与多边形区域相交时，不但能够计算得到这条直线是否与多边形区域相交，而且能够得到各个交点，并通过各个交点将直线分成多个直线段，并计算得到了每个直线段是否在多边形区域内的信息。

在 22.1 节和 22.2 节中，分别得到了线类图形元素与多边形区域和圆形区域相交关系的计算函数，参考这些函数，可以实现线类图形元素与其他区域(如椭圆和其他曲线区域、混合区域等)相交关系的计算函数，而且也可以实现其他线类图形元素(如椭圆和其他曲线、混合线等)与区域类图形元素相交关系的计算函数。

以线类相交区域计算函数为基础，在 GIS 系统中可以用来实现具体的统计和分析功能。

23.3.3　线类相交信息统计和分析

线类相交信息统计和分析，指的是通过在矢量图形中确定一个或多个线类图形元素，通过矢量图形中的各个图形元素与这些线类图形元素的相交关系，选择得到与这些线类图形元素相交的图形元素，以及与区域类图形元素的详细相交信息，并针对这些相交选择的图形元素所连接的性质数据信息，完成统计和分析功能。利用线类图形元素与区域类图形元素的相交详

细信息况，完成具体的信息统计、分析及计算等功能，实现实用 GIS 系统的具体操作功能。

例如，在完成全国基本信息 GIS 系统时，省、市、县等行政区域在矢量图形上由多边形区域来表示，其边界就是一个连续直线，公路和河流用连续直线来表示。当进行诸如"河流经过哪些省市"、"公路经过哪些省市"及"公路跨过哪些河流"等统计时，进行统计操作的基础就是线类相交图形元素选择。

又例如，在全国基本信息 GIS 系统中，完成诸如"河流经过哪些省市，经过各个省市的长度分别为多少"、"某个城市中心方圆 50 公里内河流的长度为多少"、"某条公路经过哪些省市，经过各个省市的长度分别为多少"等统计和分析功能时，完成这些功能的基础就是连续直线与多边形区域的相交关系计算。通过计算得到了连续直线与多边形区域的各个交点，这些交点将连续直线分成了更多的直线段，并得到了各个直线段是否在多边形区域内的判别信息，通过各个直线段并结合区域类图形元素的连接信息，就可以组织完成这些操作功能。

灵活运用线类图形元素和区域类图形元素的相交关系计算函数，能够针对具体的 GIS 系统，组织实现具体的操作功能。例如，在完成一个铁路新建线方案决策系统时，可以根据地形条件、地质情况、工农业情况等各方面的因素，将国土分为不同的等级区域，给各区域赋予性质数据信息来表示这些区域的地形条件、新建铁路平均每公里造价等大量信息，当在图形中确定一条线路(混合线)时，就可以得到线路与各种地形块的相交长度，计算得到线路经过各种区域的长度，根据不同区域的平均每公里造价等数据信息，计算得到新建铁路的线路长度、估算工程造价、经济评价等各种参数，为路网规划和新建铁路建设提供辅助决策。

23.4 实用 GIS 系统开发

本篇讨论实现了矢量图形系统、数据库管理系统以及两者间的连接系统，并开发实现了一些基本功能，实现了一个基本 GIS 系统。开发这个基本 GIS 系统的目的，并不是单纯为了去实现电子地图，而是要以此为基础去进行实用 GIS 系统的开发。

基本 GIS 系统只是为实用系统开发提供了管理性质数据信息的方式和平台，以这个平台为基础，可以按照具体的要求去开发实现各种实用 GIS 系统，这是用 VC++ 开发 GIS 系统的根本优势所在。只要具备了矢量图形系统、数据库管理系统及矢量图形系统与数据库管理系统的连接等三个基础系统，具备了图形元素间的空间拓扑关系计算函数，就可以任意组织程序流程，实现各种信息统计、分析及处理等功能，实现以 GIS 技术为基础，随心所欲地开发完成各种实用的信息管理系统、决策支持系统、辅助设计系统等，实现 GIS 系统与其他系统的完美结合。在灵活性、针对性、外部接口性等方面有着其他专门 GIS 系统开发工具不可比拟的优势。

本章以上各节介绍了区域信息统计和分析、线类相交信息统计和分析等操作功能。在 GIS 系统中，除了包含这些统计和分析功能之外，一般还需要具有区域相交信息统计和分析等操作功能。所谓区域相交信息统计和分析，指的是通过实现区域类图形元素间相交关系的计算函数，得到两个或多个区域类图形元素的相交关系，即不但得到了区域类图形元素是否相交的判别信息，更重要的是通过"与"、"或"等元素方式，得到新的区域类图形元素。利用这些新区域类图形元素，以及这些区域图形元素所连接的性质数据信息，完成各种信息统计和分析功能。

本章以上各节讨论的信息统计和分析功能，都是针对二维图形元素的，在很多情况下，还需要提供三维的统计和分析功能，通过将矢量图形系统三维化，并实现三维图形元素间相交关系的计算函数，实现针对三维空间的信息统计和分析功能。

多用户 GIS 系统

本篇将介绍实现分布式多用户数据库管理系统的方法和一些主要的技术，并对 GIS 系统进行多用户化的不同方案进行讨论。

第 24 章　开发分布式多用户数据库管理系统
第 25 章　实现多用户 GIS 系统

第五篇

第 24 章　开发分布式多用户数据库管理系统

24.1　Visual C++ 开发分布式多用户数据库管理系统

所谓分布式多用户数据库管理，指的是运行在网络上的可以由多个用户同时使用的数据库管理系统系统。随着计算机网络和信息技术的发展，单机运行的数据库管理系统其局限性越来越大，没有网络连接功能的数据库管理系统其生命力是有限的。本章介绍用 Visual C++ 开发基于客户机/服务器结构的数据库管理的方法和技术。

24.1.1　服务器/客户机应用程序设计

客户机/服务器结构作为实现分布式信息管理系统的方法，已受到人们的青睐。这个结构通过由一群通过网络连接起来的计算机组成。信息系统的功能由运行在这些计算机上的应用程序来完成，并通过网络进行通信。客户机/服务器结构作为一种机制已经变的相当普遍，这种机制将集中化的数据库引到网络上的桌面 PC 上去。一个或多个服务器管理集中化的数据库，而客户机则通过服务器来存取数据。

为利用客户机/服务器结构实现分布式数据库管理系统，可以选用 UNIX、WindowsNT 等操作系统，数据库服务器可以选用 SQL Server、Oracle、Sybase 等。本节介绍较常选用的配置：Windows NT 作为操作系统、SQL Server 作为数据库服务器。通过 Windows NT 操作系统将多台计算机连成网络，在作为服务器的计算机上安装 Windows NT Server 操作系统，将作为客户的计算机上安装 Windows NT Workstartion、Windows 9X 等操作系统。在服务器上安装 SQL Server 7.0 数据库服务器，在客户机上使用 ODBC(或其他方式)来使用 SQL Server 数据库。

24.1.2　Visual C++ 开发基于服务器/客户机体系的数据库管理系统

Visual C++ 6.0 企业版为开发使用 SQL Server 数据库的应用程序提供了良好的开发环境，可以很容易地去组织分布式多用户数据库管理系统的开发。在这种开发环境下，不必去过多的了解有关网络的具体技术，就像开发普通的单机数据库管理系统类似的方法组织系统的开发。开发者需要做较少的工作就可以实现分布式的多用户数据库管理系统。

在本书的 13.3 节可以看到，在 Visual C++ 6.0 企业版开发环境中，用 DataView 可以非常方便地对 SQL 数据库进行管理。通过建立 ODBC 数据源，可以像使用其他数据库(如 Access)那样使用 SQL 数据库进行应用程序设计。应该说，开发分布式数据库管理系统的思路和方法与开发单机数据库管理系统的基本技术是相同。不同的是，多用户分布式数据库管理系统在运行时，可以有多个用户同时使用服务器上的数据库，也就是说，有多个用户可以同时对服务器上的数据库进行修改，这就要求在开发过程中，需要考虑额外的一些问题。24.2 节将介绍一些开发分布式数据库管理系统时需要考虑的一些主要问题及解决方法。

24.2 分布式多用户数据库管理系统开发技术

在多用户分布式数据库管理系统中,可以有多个用户同时使用服务器上的数据库,数据库自身从系统级上解决了多个用户同时使用数据库的技术问题,不需要开发者去考虑。而多个用户同时使用同一个数据库,则会引起数据库的一致性和完整性方面的潜在问题,造成运行结果错误或不合理,破坏应用程序的完整性和一致性。在应用程序的开发过程中,需要程序设计者考虑解决这方面的问题。本节简单介绍了分布式数据库管理系统中,需要考虑的主要问题及解决方法。

24.2.1 定期更新记录集合对象

对于本书中开发实现的数据库管理系统,如果将采用的数据源改为 SQL Server 数据库,系统的各种运行功能不变。但在有多个用户在各个客户机上运行本系统的情况下,数据库的内容可能随时被任何一个用户来修改,这样对于运行在一个客户机上的用户来说,虽然没有去修改数据库中数据表的内容,但可能数据表的内容已经发生了变化。为了能够及时的浏览到经过修改后的数据表内容,就需要对显示在数据浏览视图中的数据表的内容及时进行更新,以保证数据视图中显示的记录集合对象的准确性。

要解决这个问题,可以有多种方法,最常采用的是定期更新记录集合浏览视图的方法。所谓定期更新,就是每隔一段固定时间,自动激发程序运行,利用 SQL Server 数据源中的数据表重新生成记录集合对象,然后在浏览视图中重新绘制这个重新产生的记录集合对象。

(1) 创建计时器并定期激发 WM_TIMER 消息

实现浏览视图定期重画的关键是能够定期的得到一个消息,给这个消息增加一个消息处理函数来完成数据的定期刷新。为了能够定期发送消息,需要设置一个计时器。在视图类 CDatabaseView 的构造函数中,可以用以下的语句来创建一个计时器:

```
CDatabaseView::CDatabaseView()
{
    ......//省略以上代码
    m_TimeTag = SetTimer(ONE_MINUTE_TIMER,1000,NULL);    //设置时间控制器
    if(! m_TimeTag)
    {
        AfxMessageBox("不能正确创建计时器");
    }
}
```

在以上代码中中,m_TimeTag 用来标识创建的计时器,这个变量在视图类 CDatabaseView 中定义:

```
UINT m_TimeTag;
```

参数“1000”指定了发送 WM _ TIMER 消息的时间间隔，时间以毫秒计算，参数“1000”即设置计时器每 1 秒发送一个 WM _ TIMER 消息。ONE _ MINUTE _ TIMER 是一个 ID 号，可以手工在 ID 号定义文件 resource.h 中加入：

```
# define          ONE _ MINUTE _ TIMER          1088
```

更常用的方法是，在 VC++ 开发平台的“ResorceView”工作区中，在“Draw resource”标签下按右键弹出快捷菜单，选择运行快捷菜单中的“Resource Symbols...”菜单项，得到 ID 号管理界面，通过按“New...”按钮弹出输入界面来添加新的 ID 号。

(2) 创建 WM _ TIMER 消息处理函数

用 ClassWizard 为 CDatabaseView 类增加一个 WM _ TIMER 消息处理函数，增加完毕后，在视图类 CDatabaseView 中增加了消息处理函数的定义代码：

```
//{{AFX _ MSG(CDatabaseView)
.....//省略以上代码
afx _ msg void OnTimer(UINT nIDEvent);
//}}AFX _ MSG
```

在实现文件 dbaseview.cpp 中，增加了映射机制和函数的实现代码：

```
BEGIN _ MESSAGE _ MAP(CDatabaseView, CScrollView)
    //{{AFX _ MSG _ MAP(CDatabaseView)
    .....//省略以上代码
    ON _ WM _ TIMER()
    //}}AFX _ MSG _ MAP
END _ MESSAGE _ MAP()
```

在函数 OnTimer 的实现代码中，加入重新产生浏览视图浏览内容的代码后，得到如下代码：

```
void CDatabaseView::OnTimer(UINT nIDEvent)
{
    m _ timeCount++;
    if(m _ timeCount > m _ timeRedraw)    //如果从上次刷新后又过了 m _ timeRedraw 秒的时间
    {
        m _ tiemCount = 0;                //初始化时间计数器
        Init2();                          //初始化滚动视图的纵向参数
        SetScrollSizes(MM _ TEXT, sizeTotal, sizePage, sizeLine);    //设置滚动区域
        UpdateWindow();
        Invalidate();
    }
    CScrollView::OnTimer(nIDEvent);
}
```

以上代码完成的功能是:每经过 m _ timeRredraw 秒的时间,记录集合滚动视图重新生成并刷新一次。变量 m _ timeCount 和 m _ timeRedraw 在视图类 CDatabaseView 中定义:

```
protected:
int m _ timeCount, m _ timeRedraw;
```

并在视图类 CDrawbaseView 的构造函数中进行初始化:

```
CDatabaseView::CDatabaseView()
{
    ......//省略以上代码
    m _ timeCount = 0;
    m _ timeRedraw = 30;
}
```

将变量 m _ timeRedraw 设置成 30 时,每 30 秒记录集合滚动视图刷新一次,可以通过一个交互界面来设置这个变量的值,来实现随时修改记录集合视图刷新的时间间隔。

(3) 删除创建的计时器

对于在视图类 CDatabaseView 的构造函数中创建的计时器,可以在视图类 CDatabaseView 的析构函数中进行删除:

```
CDatabaseView::~CDatabaseView()
{
    ......//省略以上代码
    delete m _ TimeTag;
}
```

24.2.2 多用户应用程序中对记录的修改策略

与单用户系统不同,在分布式多用户数据库管理系统中,允许多个用户即多个应用程序同时对数据库中的一个数据表进行修改,如果不加入一定的处理措施,就可能造成运行的不正确甚至造成系统运行出现致命错误。应当说,运行在服务器上的数据库系统(如 SQL Server)本身完善地解决了多个应用程序同时操作数据库的情况,只需在进行数据库应用程序设计时采用一定的策略,就能够实现完善的多用户数据库管理系统。

在多用户数据库管理系统设计中,最常发生和最需要避免的情况是:一个用户(应用程序)在对服务器中数据库中的数据表的一条记录进行修改操作过程中,别的用户(应用程序)可能对这条记录进行了删除或者修改操作,在这种情况下,当前用户对这条记录进行修改操作时,就可能出现错误的运行结果。处理这种情况的方法是:如果当前应用程序在对一条记录的修改过程中别的应用程序删除或修改了,就放弃对这条记录的修改操作。以下介绍在关系数据库中如何标识一条记录,以及如何识别一条记录被删除、被修改的方法。

(1) 标识数据库的记录

在关系数据库中,在建立数据表时,一般必须在数据表中建立一个标识字段,通过这个字

段来唯一的标识一条记录。在 SQL Server 数据库中，是通过建立一个 int 型字段并给这个字段赋 identity 属性来实现。具有 identity 属性的这个字段不需要应用程序来输入，而是靠数据库自身来填写，当增加一条记录时，SQL Server 数据库自动为这个字段分配一个唯一的值（这个值是由 SQL Server 中的系统数据表记录并分配的，实际上是记录最后一次操作后的字段值，下一次增加记录时，在这个值基础上简单的增加或删除一个固定数（一般设置为 1）来得到要增加记录中这个字段的值，即这个值一直在增加（或缩小），对于删除的记录，其空出的值也决不会再被分配）；而在对记录进行修改操作时，这个字段的值保持不变，这样就可以通过这个字段来唯一标识数据表中的每一条记录。

值得注意的是，在第三篇开发实现的数据库管理系统中，在作为 GIS 系统的性质数据表中设置了一个 only _ id 字段，在增加一条记录前，通过程序来计算得到了一个唯一的识别号。应用这种方法最突出的优点是，可以适用于任何类型的数据源且识别号相对集中，但这种方法的缺点是需要花费较长的计算时间。而在多用户数据库管理系统中，应用这种方法时可能会为多条记录产生同一个识别号，从而破坏数据表中记录的唯一性。在使用这种方法时，一个用户（应用程序）在产生识别号到提交新增记录的这段时间内，如果另外一个应用程序也在同一个数据表中增加一条记录，这两条记录的 only _ id 字段就会相同。通过在提交记录前重新创建记录集合对象来计算识别号的方法，可以极大地减小程序间同时产生识别号的概率，但毕竟仍然存在着这种可能性。所以，在使用关系数据库编制多用户数据库管理系统时，必须通过数据库自身来产生唯一的识别号来标识记录。

(2) 对记录修改前从数据源中得到记录

为避免在对一条记录修改时，这条记录已经被别的用户删除而造成的错误运行结果，必须在对一条记录进行修改前检查这条记录是否存在。在 19.1 节实现的数据记录修改操作中，在实现的 CData1Dlg 对话框类中，在进入 CData1Set 对象前，已经将记录指针指向了要修改的记录，这样，在利用这个对话框对这条记录的修改过程中，数据源中的这条记录可能被别的应用程序删除，从而造成运行结果错误（不合理）。为避免这种情况的发生，在对源数据表进行修改前，重新创建针对这个数据表的记录集合对象，将记录指针指向这条记录：

```
CString m_str;
if(m_pSet->IsOpen())          //如果打开了记录集合对象，则关闭记录集合对象
    m_pSet->Close();
m_str=m_pSet->m_strFilter;  //保存记录集合对象原来的过滤条件
char p1[20];
sprintf(p1,"only_id=%d",Record_ID);   //产生过滤条件式
m_pSet->m_strFilter=p1;       //设置过滤条件式
m_pSet->Open();               //重新打开记录集合对象
if(!m_pSet->IsBOF())          //如果记录集合不为空，即源数据表中仍然存在被修改的记录
{
    m_pSet->Edit();           //设置编辑状态
    UpdateData(TRUE);         //将对话框控制中的内容映射到记录集合对象中
    if(m_pSet->CanUpdate())   //如果能够修改源数据表，则通过记录集合对象修改源数据表中
```

```
                                        //记录
        m_pSet->Update();
}
else
{
    //加入提示信息,表明此记录已经被删除
}
m_pSet->Close();                        //关闭
m_pSet->m_strFilter = m_str;
```

在以上代码中,以 only_id 字段(在利用关系数据库进行多用户数据库管理系统设计时,only_id 字段为具有 identity 属性的 int 型字段)为过滤条件,重新得到记录集合对象。如果得到的记录集合对象不包含任何记录,则说明这条记录在当前应用程序进行修改前已经被别的用户删除。在这种情况下,就不再对数据表中的这条记录进行实际修改,从而避免了错误的发生。

在以上代码中,变量 Record_ID 是对一条记录修改前记录下的这条记录的 only_id 字段值。

(3) 识别对记录的修改

上述(2)中的操作只能识别在对记录进行修改过程中,别的记录删除了这条记录。在多用户数据库应用程序中,还存在这样的情况:当前用户(应用程序)调用记录修改功能对这条记录进行编辑修改过程中,别的应用程序对这条记录进行了修改,这时应用程序并不能察觉这种情况,继续对这条记录进行操作,而此时进行的操作是在获得不正确的信息(即记录的实际信息与当前应用程序得到的信息并不相同)的情况下进行的,在某种情况下,可能会产生重大的程序设计漏洞。例如,在一个对申请单进行管理的系统中,一个数据表的一条记录来管理一份申请单,主管经理(一个用户)查看申请单并对申请单进行审批(修改了某个字段),如果在主管经理审批一个申请单的过程中,职员发现这个申请单输入有误并修改了这个申请单的内容,修改后的申请单可能并不符合审批条件(即如果主管经理看到的是这个申请单的话,不会批准它),这样就造成了对不该审批的申请单进行审批的情况,系统具有设计漏洞。

以下是一个解决这个问题的办法,除了给数据表的每条记录赋予一个标识字段外,可以给数据表增加另外一个 int 型字段(或其他数值型的字段)varible_id,这个字段没有实际的意义,而只是为了避免以上出现的对数据记录的修改隐患。实现的方法如下:

在对数据表的一条记录进行修改前,记录下要修改记录的 only_id 和 varible_id 字段的值到变量 Record_ID1 和 Record_ID2,则在记录编辑对话框中对记录中的各字段值(以上的两个字段的值不能在对话框界面中修改)进行修改后,在提交(对源数据表进行修改)前,通过以下的代码重新构造记录集合对象,并将记录指针指向这条记录并对其修改:

```
CString m_str;
if(m_pSet->IsOpen())                    //如果打开了记录集合对象,则关闭记录集合对象
    m_pSet->Close();
m_str = m_pSet->m_strFilter;            //保存记录集合对象原来的过滤条件
```

```
char p1[20];
sprintf(p1,"only_id = %d&&varible_id = %d",Record_ID1,Record_ID2);   //产生过滤条件式
m_pSet->m_strFilter = p1;                    //设置过滤条件式
m_pSet->Open();                              //重新打开记录集合对象
if(! m_pSet->IsBOF())                        //如果记录集合不为空,即源数据表中仍然存在被修改的记录
{
    m_pSet->Edit();                          //设置编辑状态
    //在修改记录前,将 varible_id 的值加 1
    UpdateData(TRUE);                        //将对话框控制中的内容映射到记录集合对象中
    if(m_pSet->CanUpdate())                  //如果能够修改源数据表,则通过记录集合对象修改源数据
                                             //表中记录
        m_pSet->Update();
}
else
{
    //加入提示信息表明此记录被删除或修改,当前不能再对其进行修改
}
m_pSet->Close();                             //关闭
m_pSet->m_strFilter = m_str;
```

问题的关键在于,如果以 only_id 和 varible_id 两个字段作为过滤条件能够重新得到记录,在对记录进行修改时,将 varible_id 字段加 1。所有对数据表进行修改的应用程序都采用这个修改策略(对记录进行修改后将 varible_id 字段加 1),则如果当前应用程序在对一条记录进行修改操作时,别的用户删除了这条记录或对这条记录进行了修改(varible_id 的字段值加 1),这时当前用户(应用程序)就不能得到包含这条记录的记录集合对象,也就无法对这条记录进行修改,从而弥补了设计上的漏洞。

利用 SQL Server 进行数据库管理系统时,也可以通过设置一个 timestamp 字段 varible_id 实现同样的功能。timestamp 型字段有一个突出特点,当对一条记录进行修改时,SQL Server 数据库自动为这个字段分配一个另外的值。通过使用这个字段和标识字段 only_id 作为过滤条件,就可以有效地识别在当前应用程序在对一条记录进行修改的过程中,其他应用程序是否对这条记录进行了修改。

第 25 章　实现多用户 GIS 系统

本书指导开发的 GIS 系统是以单机状态开发完成的。本章将介绍对这个 GIS 系统进行不同程度的多用户化方案。

25.1　部分多用户化 GIS 系统

本书介绍和实现的 GIS 系统由三个主要部分组成：矢量图形系统、数据库管理系统、矢量图形系统与数据库管理系统的连接。在存储管理上，矢量图形采用的是二进制文件存储对象的方法，性质数据库和连接信息采用数据库来组织存储，而通过 ODBC 来访问使用。本章结合这几个组成部分，简单讨论一下 GIS 系统多用户化的方案。

25.1.1　部分多用户化 GIS 系统

所谓部分多用户化 GIS 系统，指的是只对 GIS 系统中的部分功能实现多用户化。在很多情况下，GIS 系统中的矢量图形只是作为进行数据库管理操作的辅助手段，在系统构建阶段绘制完成，在系统的实际执行过程中，只是性质数据被频繁修改，矢量图形较少发生变化。即在系统的运行过程中，较少对这个矢量图形进行修改，即使需要修改，也可以通过一个特殊的用户在单用户状态下进行。在这种情况下，可以只对 GIS 系统中管理性质数据的数据库管理系统部分进行多用户化。

例如，在实现一个铁路新线建设指挥管理系统时，组织建立一个铁路新线建设施工组织平面图。在施工组织平面图上包含了铁路线、铁路上的重点工程、材料供应点、交通线、河流等等各种的标志，将这些标志与数据表建立联系来实现一个简单 GIS 系统，以矢量图形界面来实现针对铁路建设过程的信息管理和施工指挥。在这个系统中，施工组织平面图在整个系统的运行周期内变化较少，而变化的主要是数据表中的具体信息，在这种情况下，就可以采用部分多用户化 GIS 系统的解决方案。即允许系统中的数据库管理系统部分可以由多个用户共同来管理和维护，而对于系统中的矢量图形部分，不允许在多用户运行状态下进行修改，只是在多用户运行状态下运行一个副本。

25.1.2　实现数据库管理系统的多用户化

对于本书讨论实现的 GIS 系统，只对数据库管理系统部分进行多用户化，其解决方案相对比较简单。在本书中，在开发数据库管理系统时，采用 ODBC 连接使用实际的数据源，所以很容易对数据库管理系统进行多用户化，因为只需将数据源改成使用诸如 SQL Server 等数据库，只需做有限地改动即可达到多用户化数据库管理系统的目的。在第 24 章中介绍了实现分布式多用户数据库管理系统的一些主要技术和策略，只要参考这些技术，对数据库管理系统进行修改，就可以使其实现多用户使用。

25.1.3　实现连接信息管理多用户化

在基本 GIS 系统中，连接信息也是利用数据表来组织管理的，管理方法与管理性质数据的数据库管理系统类似，所以可以采用类似的方法实现连接信息的管理多用户化。

在本书的第 20 章实现了将矢量图形元素与数据记录进行连接的对话框类 CLinkDlg。这个对话框类，不能完全适用于多用户运行状态下的需要。为满足多用户状态下的要求，需要对这个完成连接操作的对话框类进行修改。

在 CLinkDlg 对话框类中，采用的是通过结构数组记录连接信息的方法，在进入对话框时，通过一个结构数组记录下一条记录所连接的图形元素，在对话框中进行的诸如增加、删除连接的操作都是对结构数组进行的，只有保存退出对话框时才能将对连接信息的实际修改反映到存储连接信息的数据表中。这种实现方式在单机状态下是十分有效和完善的，但不能适用于多用户状态，因为由这个对话框类实现的连接操作，在对话框编辑连接信息的整个过程中，如果别的应用程序对连接信息进行了修改，当前应用程序并不能察觉。

要实现在多用户状态下管理连接信息，最主要的是能够实时对存储连接信息的数据表进行修改，即在增加一个连接时，首先检查这个连接(记录)是否存在，如果这个连接不存在，则在数据表中增加一条存储连接信息的记录。检查连接信息和增加一条连接信息是在一个过程中完成的，其时间间隔极小，不可能在数据表中产生同样的连接信息(在当前应用程序检查连接信息到增加连接信息前这段时间内，另外一个应用程序检查了连接信息数据表并要增加一个同样的连接信息的概率几乎等于 0)。对于删除连接信息操作，采用同样的操作方法，在删除连接信息前，首先检查在连接信息数据表中是否存在这个连接信息，以避免删除别的应用程序已经删除的连接信息。

25.2　完全多用户化 GIS 系统

实现完全多用户化的 GIS 系统的关键，是对矢量图形系统进行多用户化，要对矢量图形系统进行多用户化，需要对矢量图形元素的存储组织方式进行修改。一般的解决方案是，将矢量图形元素记录化，即通过数据库来存储矢量图形元素，这样就可以采用多用户化数据库管理系统的方法来多用户化矢量图形系统。

25.2.1　矢量图形元素的数据库记录化

利用数据库管理系统管理矢量图形系统的关键，是将矢量图形元素数据库记录化。通过建立一个或多个数据表，来存储某一类矢量图形元素。如对于直线图形元素，可以用表 25-1 结构的数据表来表示：

表 25-1

字段名称	字段类型	含　义
only_id	int	此字段采用 identity 属性，以自动产生唯一的识别号
LineWide	int	线宽

续表

字段名称	字段类型	含 义
LineType	int	线型
Layer	int	层
IsDelete	booler	是否处于删除状态
ColorPen	int	笔色
ColorBrush	int	画刷颜色
StartX	float	起点横坐标
StartY	float	起点纵坐标
EndX	float	终点横坐标
EndY	float	终点纵坐标

对于圆、圆弧、标注矢量文本等都可以建立类似的数据表来表示，对于连续直线(或多边形区域)，可以通过两个数据表来表示，第一个数据表(表 25-2)来存储基本信息：

表 25-2

字段名称	字段类型	含 义
only_id	int	此字段采用 identity 属性，以自动产生唯一的识别号
LineWide	int	线宽
LineType	int	线型
Layer	int	层
IsDelete	booler	是否处于删除状态
ColorPen	int	笔色
ColorBrush	int	画刷颜色
pointnumb	int	顶点数目
isFill	booler	是否填充

用另外一个数据表(表 25-3)来存储连续直线各顶点的坐标，表的结构如下：

表 25-3

字段名称	字段类型	含 义
only_id	int	识别号，等于主信息表中的 only_id 字段，用来识别那些哪些记录存储的定义属于某个连续直线
cord_x	int	顶点的 x 坐标
cord_y	int	顶点的 y 坐标
cord_z	int	顶点的 z 坐标
sequence	float	用来决定各个顶带内的顺序，为插入顶点操作服务

连续直线的主信息存储在表 25-2 中，连续直线的顶点坐标存储在表 25-3 中。通过字段 only_id 来决定一条连续直线包含哪些顶点。

除了各种图形元素之外，其他的图形绘制参数(如颜色信息、图层信息等)都可以建立相应的数据表来表示。

利用服务器中数据库中的各个数据表,来存储管理与矢量图形系统有关的各种信息,为实现多用户化矢量图形系统奠定基础。然后,采用以上各节介绍的对数据库管理系统进行多用户化的思路和方法,实现矢量图形系统的多用户化,这样,就可以实现一个完全的GIS系统。

25.3.2 转换数据表的内容为图形元素

用数据库来存储矢量图形时,对矢量图形元素的修改即是对数据库记录的修改。绘制图形元素时,应用程序从数据库中取出与矢量图形有关的各种信息,利用这些信息绘制出矢量图形来。在这个实现过程中,一个关键的技术问题是,系统能够迅速的绘制图形元素。如果矢量图形的每次绘制都需要从数据表中取出数据来完成,数据库管理系统的运行速度势必受到影响。所以在实现矢量图形系统时,可以把数据库管理系统中存储的图形元素转换成客户机中的图形元素对象的形式,并保持一定的相对稳定性。在客户机中,通过定期、人为触发等方法去执行将存储在数据库管理系统中的图形元素转变成客户机中图形元素对象的操作,来刷新原来的矢量图形元素。

将数据库中数据表中的记录转换成图形元素的操作比较简单,在此不再具体讨论。下面只简单的介绍一下在客户机中刷新矢量图形时,可以采用的触发机制。

(1) 定期触发

所谓定期触发,就是在客户机应用程序中设置一个计时器,每隔一定的固定时间,就运行转换程序将存储在数据库中的信息转换成客户机上矢量图形元素对象的形式,来刷新矢量图形。

(2) 人为交互触发

人为交互触发方法更加简单,就是在客户机应用程序中建立一个触发控制(菜单或按钮等),在需要进行矢量图形刷新时,通过这个控制运行转换程序将存储在数据库中的信息转换成客户机上矢量图形元素对象的形式,来刷新矢量图形。

(3) 根据数据库更改触发

这是一种较为完善的触发机制。在应用程序中,当转换程序将数据库的记录转换成图形元素操作后,记录下各个数据表的最后修改时间。客户机应用程序在空闲时间或进行某项操作前,检查各个数据表的修改时间,根据修改时间来确定哪个数据表的内容发生了变化。对于发生变化的数据表,执行转换程序将其内容转换成客户机的矢量图形元素对象,来刷新原来的矢量图形。

在实际的应用程序设计中,可以根据自身应用程序的特点(矢量图形的大小、矢量图形的改动频率等)设计出图形刷新机制,在系统的数据统一性、运行速度等方面达到一个最好的效果。

附录 本书所附光盘内容及技术支持

(1) 本书作者实例指导开发了一个基本GIS系统。为满足读者学习的需要,作者为本书配套了一张CD-ROM光盘。光盘中包含了书中所介绍的基本GIS系统的完整源程序代码。这既节省了读者的时间,又方便读者在代码基础上进行实用系统的开发工作。光盘中包含的内容可参考光盘中的readme.txt(或readme.doc)文件。

(2) 为满足读者在较高的基础上开发实用系统的需要,作者把多年研究的一些成果,整理成下列的系列应用程序,将程序源代码供读者选用。

系统源程序盘名称	主要内容
盘A:矢量图形系统	包含本书讨论的基本矢量图形系统Draw的功能。增加了图层、颜色、线型等的设置功能;增加了窗选、圆选、多边形选择图形元素功能;增加了图像底图功能;增加了与AutoCAD的DXF文件的接口;扩充了图形块(图例)的制作、编辑和绘制功能,及完整的随机帮助功能
盘B:GIS系统	包含矢量图形系统的功能。利用ODBC数据源实现了对数据表的通用浏览功能、通用查询功能(提供了组织查询条件的编辑器)、通用排序功能以及对数据表的编辑功能(增加、修改记录)。提供了数据表的记录与矢量图形元素间的任意多项连接的功能,实现了通过矢量图形元素查询性质数据以及通过数据表的记录得到图形元素的双向查询功能。实现了空间信息统计功能、实体的组织和绘制功能,并增加了所画即所得的电子表格功能,以及将多个电子表格结合在一起的演示表格功能

以上应用程序的详细说明和盘A和盘B系统的DEMO可在本书所配的CD-ROM光盘中找到。凡需购买上述源代码磁盘的读者,请与作者直接联系。

(3) 作者联系地址如下:

通信地址:北京市北方交通大学土木建筑工程学院 陈建春收 邮编100044

联系电话:(010)63240257、62252216

E_mail:chenjianchun@263.net

参 考 书 目

[1] 陈建春. Visual C++ 高级编程技术——开发实例剖析. 北京:电子工业出版社,1999

[2] 陈建春. Microsoft Visual C++ 图形系统开发技术基础. 北京:电子工业出版社,1998

[3] Visual C++ 开发使用手册,北京:机械工业出版社,1999

[4] 程耀、宋守许、胡立. Visual C++ 5.0 程序设计教程. 北京:电子工业出版社,1998

[5] 邱仲潘等译. Visual C++ 6 从入门到精通. 北京:电子工业出版社,1999

[6] 屠强、白铮、周迎秋、程源等译. Visual C++ 开发指南(第二版). 北京:电子工业出版社,1998

[7] 王国印等译. Visual C++ 技术内幕(第四版). 北京:清华大学出版社,1999