求是科技

Visual C++

数据库通用模块及典型系统开发

● 求是科技　编著

实例导航

附源代码光盘
CD-ROM

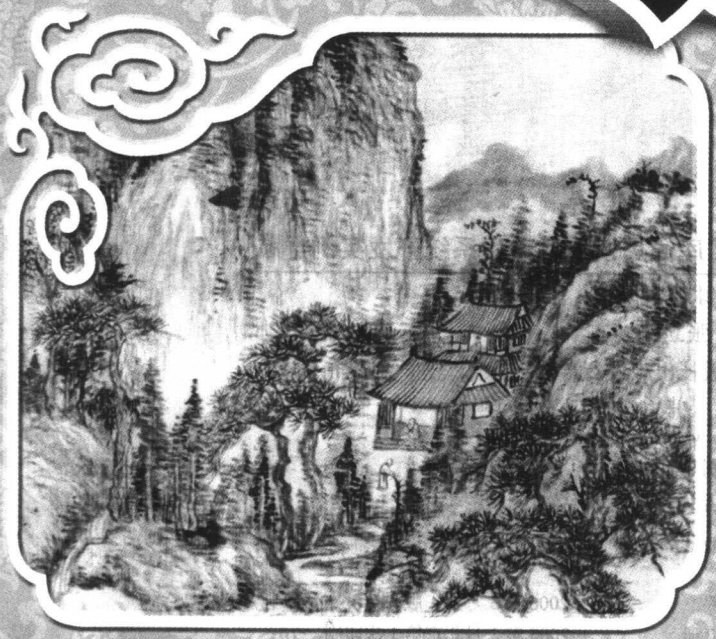
人民邮电出版社
POSTS & TELECOM PRESS

Visual C++

数据库通用模块及典型系统开发

实例导航

● 求是科技　编著



人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

Visual C++数据库通用模块及典型系统开发实例导航/求是科技编著. —北京：人民邮电出版社，2006.3

ISBN 7-115-14334-X

Ⅰ. V... Ⅱ. 求... Ⅲ. C语言—程序设计 Ⅳ. TP312

中国版本图书馆CIP数据核字（2006）第014319号

内 容 提 要

本书对Visual C++数据库通用模块及典型系统开发进行了详细的介绍，全书共分为3个部分。第一部分介绍软件设计中的基础知识和需要注意的问题，包括分层、数据访问层和ADO技术。第二部分是典型模块的设计和实现，典型模块选择的几乎是所有大型软件中都会用到的模块，例如登录模块、数据和图像显示模块、打印模块、编辑模块等。掌握这些模块的实现是开发大型应用程序的基础。第三部分是典型系统的开发，包括系统预览（相当于原始需求）、需求分析、模块设计和具体实现等内容。

本书适合大中专院校的学生和软件项目开发人员学习和参考。

Visual C++数据库通用模块及典型系统开发实例导航

◆ 编　　著　求是科技
责任编辑　张立科

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鸿佳印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：22.25
字数：540千字　　2006年3月第1版
印数：1－6 000册　　2006年3月北京第1次印刷

ISBN 7-115-14334-X/TP・5192

定价：42.00元（附光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223

前　言

Visual C++是最优秀的开发工具之一。由于其所具有的强大功能、灵活的特性和良好的对底层控制能力，使其被广泛应用于研究和开发工作中。

与快速开发工具（RAD）相比，Visual C++不具有强大的可视化界面设计功能，这就使通过 Visual C++开发应用程序带来了一定的难度。可以说，学习 Visual C++容易，但用好 Visual C++难。在本书中给出了很多涉及到界面设计方面的示例，例如“编辑模块”的字体工具条、“数据显示模块”的可编辑表格、“人事管理系统”的多视图结构等。读者可以通过这些示例学习到在 Visual C++中设计界面的基本方法，提高应用程序的开发效率。

随着计算机技术的不断进步，计算机的应用越来越广泛，应用程序的规模也与日俱增。随着程序规模的扩大，修改的成本也越来越高。降低修改成本的方法除了做好需求分析、控制好软件的开发流程外，还需要设计一个良好的程序结构。另外，在本书介绍的典型模块和典型系统中，都会根据分层的思想，对程序的结构进行合理划分。

本书中使用了大量的示例程序，这样可以使读者更好地运用 Visual C++进行程序设计和开发，而不是为读者提供现成的源代码。建议读者本着学习程序开发思想的目的来学习本书中的内容和源码。

在读本书之前，读者需要具备一定的 Visual C++和 MFC 的基础知识，这样更有助于理解书中讲到的知识点和示例。

本书具有如下特点：

（1）本书分为通用模块与典型系统，在通用模块中分别详细介绍模块的创建过程与方法，在典型系统中介绍使用通用模块创建系统的方法。

（2）案例程序的功能贴近实际，读者可以在案例的基础上添加功能，使案例实用化。

（3）光盘资料丰富，光盘中程序的代码经过严格测试，有利于读者根据源代码进行学习。

由于时间仓促和作者的水平有限，书中错误和不妥之处在所难免，敬请读者批评指正。

编　者

2006 年 2 月

目 录

第 1 章　多层结构的设计

1.1　分层的原因

首先看一个简单的示例。在这个示例中，用户输入用户名和密码，系统与 ini 文件中的用户名和密码进行比较，最后把登录的结果返回给用户。示例代码如下：

```
#include "stdafx.h"
#include <iostream.h>
#include <stdio.h>
#include <windows.h>


int main(int argc, char* argv[])
{
 char UserName_Input[20];
 char Password_Input[20];
 char UserName[20];
 char Password[20];

 cout<<"请输入用户名：";
 cin>>UserName_Input;

 cout<<"请输入密码：";
 cin>>Password_Input;

 GetPrivateProfileString("UserInfo", "UserName", "", UserName, 20, "UserInfo.ini");
 GetPrivateProfileString("UserInfo", "Password", "", Password, 20, "UserInfo.ini");

 if((strcmp(UserName_Input, UserName) != 0) || (strcmp(Password_Input, Password) != 0))
       cout<<"登录失败"<<endl;
 else
       cout<<"登录成功"<<endl;

 return 0;
}
```

这段程序把界面的显示、文件的访问，用户名和密码的比较都放到了一起。就功能而言，这样做没有一点问题，它也能很好地实现需求。但这种做法在应对需求变化方面就差了许多。

考虑下面几种情况。

- 程序要从控制台窗口程序转为 Windows 窗口程序。
- 用户名和密码不是保存在 ini 文件中而是保存在数据库中。
- 给用户名和密码加密。

当这段程序遇到上述某个或几个变化时，需要如何改动呢？以界面变化为例，程序设计人员需要在程序中找到所有的输入输出，将其改为 Windows 窗口的输入输出，并且需要根据

Windows 程序设计特点调整程序的结构。这个工作量是可想而知的。

如果在程序设计的开始就把界面显示、数据访问、业务逻辑等都封装成单独的模块呢？这是一个好的想法，这种思想就是模块编程的基本思想。模块编程把相同或相似的功能封装成模块，减少了冗余代码。模块编程的结构如图 1-1 所示。

> 注意：冗余代码是一个很可怕的东西，当程序变化时，每一处都需要修改。这会使程序修改的复杂度成倍增长。修改一旦发生遗漏，就会产生 Bug。

随着程序规模的扩大，传统的模块编程也出现了一些致命的缺点。

- 模块间调用很随便，各种模块互相牵连，独立性差，系统结构不够清晰。
- 数据大多数情况下都作为全局变量处理，软件系统中的任一模块都可能对数据进行存取和修改，从而造成了各模块间有着很隐蔽的关系。要更改一个模块是很困难的，因为要弄清各模块间的接口，按当初约定的规则处理数据。

显然，要清除传统的模块编程的缺点就必须减少模块之间毫无规则地相互调用、相互依赖的关系，特别是清除循环现象。软件分层思想就是从这点出发，它力求使模块间的调用的无序性变为有序性。

所谓的分层，就是把所有的功能模块，按照功能的调用次序划分成若干层，各层之间的模块只能单项依赖或单项调用（例如只允许上层调用下层或外层调用里层）。这样不但结构清晰，而且不会造成循环。分层结构如图 1-2 所示。

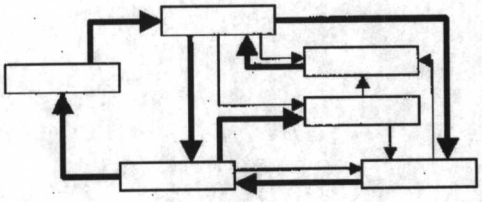


图 1-1　模块结构

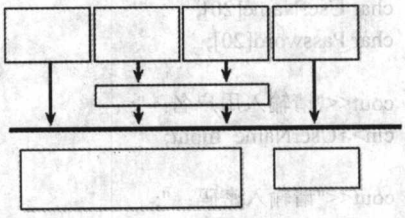


图 1-2　分层结构

1.2　典型的三层结构

1.2.1　三层结构简介

在介绍三层结构之前，首先明确一点，软件要分成几层，完全根据需要而定，不要拘泥于现有的结构。因为三层结构在当今企业应用中用的比较广泛，所以本节主要对其进行介绍。

所谓的三层是指界面层、逻辑层和数据访问层，三层的关系如图 1-3 所示。

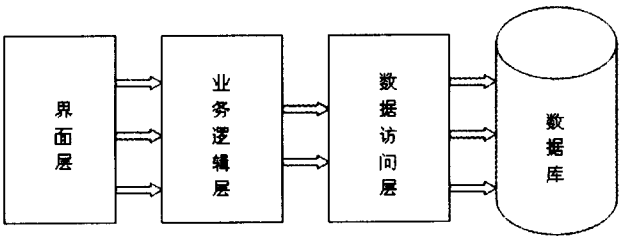


图 1-3　三层结构关系图

1．界面层

界面层是用户和应用程序交互的接口，它主要完成的功能如下。

(1) 辅助用户输入，给出必要的提示

该功能是界面层的主要功能。当今，已经很少见到批处理形式的软件了，一个好的软件，首先需要的是和用户有一个好的交互。试想，一个什么人都不会用或者只有少数几个人会用的软件，能被人们广泛接受吗？

(2) 限制用户的非法输入

这个功能是一个安全方面的保障。软件的用户可能是一个根本不懂如何使用软件的人，甚至是以破坏为目的的用户。对于这些用户，他们的输入是很危险的。著名的 SQL 注入漏洞就是对非法输入没做控制导致的。

限制用户的非法输入，常见的方式有限制用户输入的长度、限制用户只能输入某些字符或限制用户不能输入某些字符。

(3) 响应用户操作

当用户进行某个操作，应该能够调用逻辑层的某些逻辑对用户的操作进行处理，并将执行结果返回给用户。例如用户登录程序，用户输入了用户名和密码后，登录界面应该调用逻辑层的用户名、密码校验模块对用户名、密码进行校验，并将登录结果以某种形式返回给用户。

(4) 显示操作的执行结果

操作的执行结果是多种多样的，简单的可以使用一个对话框提示用户，复杂的可能需要一个表格显示大量的数据。主要有以下几种：

- 提示信息；
- 格式化数据；
- 特殊显示，例如图片；
- 将一些特殊代码翻译成用户能读懂的语言。

(5) 处理拖拽、剪贴板等特殊操作

这些操作也是软件易用性方面一个很重要的部分，良好的界面层应该支持这些操作。此外，在界面层的设计中，程序人员需要根据显示的方式选择不同类型的界面，例如控制台界面、Windows 窗口、网页等。

2．逻辑层

逻辑层全称为业务逻辑层，它位于界面层和数据访问层之间。通常情况下，业务逻辑层

由一组业务组件组成。这些业务组件主要完成以下功能。

(1) 对特定业务逻辑和内部业务流程封装

这是业务逻辑层必须完成的工作，对不同的软件，需要处理的业务逻辑各不相同，封装的方法也各不相同。本书的后续章节中会通过几个典型模块和典型案例介绍业务逻辑的封装。

(2) 支持事务

业务逻辑往往不是单个的模块，而是一个多个模块组成的一个流程，业务逻辑层支持事务就是为保证流程的完整性。

(3) 需要有一定的安全机制

安全机制是软件健壮性的保障。

(4) 为界面层、其他业务组件提供接口

能够通过调用数据访问层组件来获取或修改数据。对于这一点，将在后面讲三层之间的关系时做详细的介绍。

(5) 能够通过代理调用外部服务

有些时候，软件需要其他的软件作为支撑，这时候就要求业务逻辑层有调用外部服务的能力。

3. 数据访问层

数据访问层负责和各种数据源打交道，例如 SQL Server、Oracle 等关系型数据源、XML 数据以及其他种类的非关系型数据、Web 服务、各种特别的遗留系统等。它的主要作用是封装数据访问操作，为逻辑层提供统一的数据服务。逻辑层可以认为自己在访问一个特殊的数据库——数据访问层。

1.2.2 层与层之间的关系

通过对 1.1.1 节的学习，读者应该对为什么要分层有了一定的了解。分层最大的好处就是降低各模块之间的耦合。

> 耦合是从模块外部考察模块的独立性程度。它用来衡量多个模块间的相互联系。一般来说，耦合度应从以下三方面来考虑，即：耦合内容的数量、模块的调用方式、模块间的耦合类型。
>
> 内聚是模块内部各成份（语句或语句段）之间的联系。显然，模块内部各成份联系越紧，即其内聚越高，模块独立性就越强，系统越易理解和维护。具有良好内聚度的模块应能较好地满足信息局部化的原则，功能完整单一。同时，模块的高内聚必然导致模块的低耦合。理想的情况是：一个模块只使用局部数据变量，完成一个功能。

一个好的设计强调的是高内聚和低耦合，为了做到这一点，层与层之间的通信必须遵循一定的约束条件。这些约束条件是什么呢？归根结底一句话，每一层只能调用其下一层提供的服务，禁止跨越层调用和调用上一层。例如界面层只能调用逻辑层的服务，不能调用数据访问层的服务。逻辑层不能调用界面层的服务。

在程序设计中，尤其是大规模软件的开发中，程序人员都遵循一套严格编程规范来保证软件的可读性、易维护性。这些编程规范包括变量、常量、函数等的命名规范，甚至包括缩

进的空格数。但却很少有架构方面的约束。对于不同的软件，架构是不同的。但是对同一软件，一旦架构设计出来，就应该定义出一套相应的规范来约束程序人员严格地按照架构来编写代码。

下面将以界面层和逻辑层为例讲述各层之间的关系。

按上述约束。逻辑层需要给界面层提供一系列的逻辑操作，而不能调用界面层的逻辑。这也就要求逻辑层需要为界面层提供一组操作接口，很多时候，这些接口也会组成一个代理层，如图 1-4 所示。

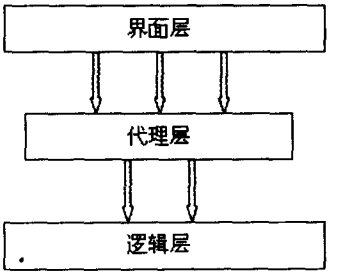


图 1-4　使用代理层来连接界面层和逻辑层

使用代理层，可以很好地隔离界面层和逻辑层，使逻辑层的接口更清晰。同时，使用代理层还可以实现懒加载（lazy loading）提高效率。

在界面层和逻辑层间通信时，常出现下面的几种错误，在这里列举出来希望能引起读者能注意。

（1）逻辑层直接提示用户

这是在该层中最容易犯的错误，例如下面这句代码在逻辑层中执行时发现了越界，直接弹出对话框通知用户。

```
if(a < 5)
{
 //……
}
else
{
 MessageBox(NULL, "数组越界", "错误", MB_OK + MB_ICONERROR);
}
```

这是一个非常不好的设计。在逻辑层提示一下的确很简单，但当面临到界面的修改、尤其是界面类型的修改时，麻烦可就大了。例如将 Windows 窗口改为网页显示，程序中所有类似 MessageBox 的函数都不能再用。如果这些函数除了界面层外，逻辑层中也存在，那么就需要在两个层中进行改动。

对于这个问题，常用的解决方式有如下两种：

①使用返回值

逻辑层增加如下方法用来检查是否越界。

```
bool CheckRange(int a)
{
 return a < 5;
}
```

原来的逻辑操作改为：

```
void Operate(int a)
{
 Assert(a < 5);
 //……
}
```

界面层进行调用，具体代码如下：

```
if(CheckRange(a))
{
 Operate(a)
}
else
{
 MessageBox(NULL, "数组越界", "错误", MB_OK + MB_ICONERROR);
}
```

②抛异常

这种方式需要界面层对异常进行处理。

异常类：

```
class MyException
{
public:
 MyException() {};
 ~MyException() {};
 char * Message()
 {
       return "数组越界";
 }
};
```

抛异常程序：

```
if(CheckRange(a))
{
 Operate(a)
}
else
{
 throw MyException();
}
```

（2）逻辑层内设置界面的显示

最常见的是设置界面中的某些控件的状态。很多逻辑操作都会影响到界面是否可用的问题，例如一个媒体播放软件，当用户单击“播放”按钮后开始播放，“播放”按钮会变成“暂停”按钮。还有一种情况也很常见，就是单击一个按钮后，界面上的某些控件变为不可用。对于类似这些控制，经常会在不好的逻辑层设计中见到。

（3）界面层直接访问数据库层

在软件设计中，之所以提出层的概念，就是为了控制模块间的胡乱调用，降低模块间的耦合。界面层如果直接调用数据访问层的服务，也就将界面层和数据访问层绑定到了一起，更换界面层或数据访问层都会很麻烦。

1.2.3　层的位置

在这里，层的位置指的是各层（例如界面层、数据访问层、逻辑层等）是在客户端运行还是在服务器端运行。对一个大型的网络系统，这是一个需要仔细考虑的问题。

一般情况下，程序分为单机和网络（包括分布式系统）两类。

对于单机程序，各层都在同一台机器上运行。例如 Windows 操作系统就是这类程序。

> **注意：虽然 Windows 使用的不是第 1.2.1 节中介绍的三层结构，但它本身也是一个很好的分层结构。上面曾经提到，分层是一种思想，不能拘泥于某种层次结构，要根据具体的需求而定。**

对于网络程序，数据访问层一般安排到服务器端。主要需要考虑的是逻辑层和界面层应该在服务器端运行还是在客户端运行。

最简单的情况是将所有的运行都在服务器端，使用一个 Web 浏览器的 HTML 前端和用户交互。这样做最大的好处是将所有的逻辑都集中在服务器端，所有的修改都只需要在服务器端进行和重新编译，不需要考虑客户端的分发及客户端版本等问题，也不需要考虑客户端已安装软件的兼容问题。

是不是上述方式就是最好的呢？也不一定，上面的架构最主要的问题就是效率问题。所有的工作都要由服务器承担。随着系统规模的不断扩大，服务器端的压力也就越来越重。不改变架构的情况下，只能通过增加系统配置来维持系统的正常运行。在这种情况下，程序的某一部分不得不放到客户端运行。

在客户端上运行程序的最大好处是系统的响应性好，甚至在网络断开的情况下也能正常运行。有些情况下，某些非常复杂的任务超出了 Web 可以处理的范围。这时，部分程序不得不运行在客户端。

对于逻辑层，它可以全部运行于服务器端，可以全部运行于客户端，也可以一分为二。和上面提到的一样，运行在服务器端有助于系统维护，运行在客户端能够提高响应速度和断网使用的需求。

将逻辑层一分为二是最差的选择，因为这样做无法确定某个业务逻辑到底运行在哪一端。如果非要这样做的话，建议读者将两部分分别封装成独立的模块，两个模块间尽量不存在任何关系。

尽量将所有的业务逻辑都放到服务器端去执行，因为这样能够对数据和操作进行统一的处理，不用考虑不同用户间的同步等问题。

如果有一部分逻辑在客户端运行又不能单独封装成模块时，可以考虑将所有的逻辑都放到客户端去执行。