

# ProE 野火版 TOOLKIT

## 二次开发

## 入门与进阶



谨以此书献给那些热爱 **PRO/E** 二次开发的人们

作者：王伟

(测试版 3.0) 20100312

# 目录

|                                |    |
|--------------------------------|----|
| 序言： .....                      | 4  |
| 第一节：环境配置 .....                 | 6  |
| 环境配置一：野火 2.0 搭配 VC++6.0 .....  | 6  |
| 环境配置二：野火 4.0 搭配 VS2005 .....   | 11 |
| 第二节：菜单 .....                   | 14 |
| 第一个普通菜单 .....                  | 14 |
| 第一个浮动菜单 .....                  | 20 |
| 第一个右键菜单 .....                  | 23 |
| 补充内容：热键指定（快捷键的一种） .....        | 28 |
| 第三节：对象的基本操作 .....              | 30 |
| 信息输入/输出 .....                  | 30 |
| 选择对象 .....                     | 34 |
| 单选 .....                       | 34 |
| 多选 .....                       | 35 |
| 框选 .....                       | 36 |
| 自动选择 .....                     | 37 |
| 载入对象 .....                     | 39 |
| 遍历对象中的元素 .....                 | 42 |
| LOG 文件 .....                   | 44 |
| 制作并调用快捷键 .....                 | 46 |
| 第四节：参数，尺寸，关系式 .....            | 50 |
| 参数操作 .....                     | 50 |
| 尺寸操作 .....                     | 53 |
| 补充知识：函数指针 .....                | 56 |
| 参数与关系式的搭配使用 .....              | 57 |
| 综合实例：齿轮的参数化设计的实现（基本原理） .....   | 61 |
| 第五节： 装配概述 .....                | 64 |
| 选择装配 .....                     | 64 |
| 深入理解 ProAsmcomppath .....      | 66 |
| 自动装配 .....                     | 69 |
| 元素的遍历与干涉 .....                 | 73 |
| 第六节：UDF 特征创建 .....             | 77 |
| 如何制作 UDF （用户自定义特征） .....       | 77 |
| PART 环境中调用 UDF 特征 .....        | 79 |
| 组立环境中调用 UDF 特征 .....           | 82 |
| 第七节：工程图概要 .....                | 85 |
| 表格 .....                       | 85 |
| 符号 .....                       | 88 |
| 将装配模型及其组件载入当前磁盘并改名（含工程图） ..... | 92 |
| 将工程图导出为其它格式 .....              | 96 |
| 第八节：界面一瞥 .....                 | 97 |

|                         |     |
|-------------------------|-----|
| ProE 系统 UI 界面一瞥.....    | 97  |
| VC 界面一瞥.....            | 105 |
| 第九节：数据库.....            | 110 |
| 用 INI 文件制作系统配置文件.....   | 110 |
| 用 TXT 文件制作微型数据库.....    | 113 |
| 用 EXCEL 文件制作微型数据库.....  | 115 |
| 用 ACCESS 制作微型数据库.....   | 119 |
| 第十节：工程专题.....           | 126 |
| 专题一：建立团队公用（DLL）函数库..... | 126 |
| 专题二：建立元件库（标准件库）.....    | 135 |
| 专题三：BOM 表系统.....        | 141 |
| 专题四：关于程序的移植性与有效期.....   | 142 |
| 专题五：螺丝系统.....           | 147 |
| 专题六：自动打印系统.....         | 147 |

# 序言：

本书的序言与其他相关书籍不太一样，因为，笔者并不打算在此千篇一律的描述二次开发是如何重要，产生于什么样的背景，将会有有什么新的发展或璀璨的将来。有如此描述的书籍可谓是多如牛毛，读者若对此感兴趣，随意在网络上搜索便是。

本书页码不多，是一本袖珍型，压缩型的学习笔记。

本书将要描述的是一项实实在在的处理问题的方法，笔者一直认为，既然是方法，尤其像这种与程序要打交道的方法，就应该就事论事，直截了当。（当然，笔者会尽力不把本书写的像高等数学那样枯燥无味）本书的特点就是：废话少，经验与实际操作比较多。本书的大部分内容都是笔者在自学过程中总结出来的，虽然不能说是极具参考价值，但是笔者能够肯定的是，本书一定能帮助初级人员少走弯路，同时也能给中级人员一些另类的解决问题的途径。不管你们目前的水平如何，本书中的一些学习技巧以及解决问题的方法，都将在你们开发 PROE 的路上起到一臂之力。

读者也许会感到疑惑，这本书真的有如此功效吗？

没错，如果你热爱 PROE 二次开发，或者打算在该方面大展拳脚，那么本书的相关内容无疑会给你带来意想不到的收获（即使是笔者讲到的最初级的部分）。笔者敢于这样说，那是因为笔者正是从 C++ 最简单的基础开始，一步一步走上 PROE 二次开发之路的。笔者从无到有的大部分经验都总结在了本书中，同时笔者相信大部分的读者在初学时都将遇到与读者相同的问题，现在社会的发展是飞速的，如果什么都自己从头来摸索那将是对大家时间的浪费。笔者在学习的过程中也得到了很多朋友的帮助，作为对他们的回报（也算是经验共享吧），笔者也在此将自己探索的结果公布于众，希望能够对大家有用。

相信很多人都知道 EMX（PROE 中的一个比较出名的外挂，中文名字叫模具专家系统），学完了本书的内容，或许其中 60%~70% 的功能你也可以实现。当然，本书绝不是要教会大家来重写 EMX，也不是吹嘘笔者有多能耐，而是借此说明本书的价值。

本书所有内容均为笔者亲自编写调试，其中有些是来自对 UserGuide（PROE 帮助文件）的资料整理，有些则是直接引用了其中相关的内容。由于精力有限，笔者并没有涉及到 PROE 的各个方面，不过其中的一些探索并解决问题的方法将适合任何模块。

本书读者：

如果你还不曾了解 C/C++ 的基础知识，或者还不曾了解 PROE 的基本指令，那么本书目前并不适合你，你可以在了解这些知识后，再学习本书中的相关内容。

如果以上两样你都具备，同时又苦于自学无门，那么本书将非常适合你。

如果你正在被某项开发所困扰，那么本书也一样适合你，其中的一些解决问题的方法或许会激发你的灵感。

如果你已经是一个老手，那么也希望你能阅读本书，同时指出书中欠妥之处，以便笔者改正后，造福更多的后来者。

#### 衷心感谢：

首先感谢 PROE 公司提供了二次开发接口，这接口使得我们可以根据实际需要进行二次开发以提高工作效率，同时也孕育了本书的诞生。

其次要感谢我的妻子安金凤女士，她给了笔者生活上，精神上无微不至的照顾，使笔者能心无旁骛的进行程序设计。同时，也是她一直在旁边鼓励笔者坚持创作，本书才能在如此短的时间内完成。

最后，感谢所有为这本书出谋划策并挥洒汗水的人，有了你们无私的帮助，本书才终于得以面世，衷心的感谢你们！

#### 其他：

本书包含的内容是笔者这几年来利用业余时间摸索帮助文件的心血，如果与某些读者的技术有所雷同，纯属荣幸。如果有热心的读者发现有人盗用了本书中的原版内容并加以收费，欢迎举报。

由于本书涉及的内容并不是什么秘密（帮助文件中都有，只是难找而已），作为经验共享的一种读物，笔者允许读者任意复印，拷贝本书，但读者不得修改本书中的内容并用于商业用途。同时，笔者保留对本书署名的权利。

读者在阅读本书的过程中有任何疑问，可致电：[pretty494@163.com](mailto:pretty494@163.com) 寻找帮助，笔者将不定时的回答你们提出的相关技术问题。

另外，笔者可以代工各种基于 PROE TOOLKIT 的开发项目，提供相关的开发资料以及二次开发培训等，欢迎联系。

王伟  
2009-4

# 第一节：环境配置

俗话说，工欲善其事，必先利其器。虽然已经有很多书籍都介绍了如何配置 proe 的二次开发环境，笔者也曾经按照那些著作的方法配置过，但是，笔者始终都没有发现一个较为完善的配置方法，配置的环境总有一些警告，或者显得非常繁琐，因此，笔者在这里向大家推荐一种零警告，零错误的配置方案。

## 环境配置一：野火 2.0 搭配 VC++6.0

首先，请打开你 Pro/E TOOLKIT 所在的根目录，例如，本机的目录在：  
E:\ProEngineer\proe2.0\proeWildfire 2.0\protoolkit 下。在该目录下有四个文件夹，其中 i486\_nt 中存放的是对外开放的 DLL 文件，includes 中存放的是开发中需要用到的一部分头文件，protk\_appls 中存放了一些范例源代码，protkdoc 中则是帮助文件。另外还有一个 Protoolkit\_Wildfire20\_RelNotes.pdf 文件用来对比了当前版本与以往版本的不同之处，tkuse.pdf 则系统的讲解了如何使用 TOOLKIT 进行二次开发。

在开始配置环境前，我们先来认识下 TOOLKIT 中的几个关键 LIB (库文件)，只有认识了这几个 LIB，我们才能明白为什么要按照笔者介绍的方法来配置环境。

打开 i486\_nt\obj 文件夹，我们看到有好几个 LIB 文件，这里，重点先向大家介绍 protk\_dll.lib 和 protoolkit.lib 两个库文件。（实际上在应用时，我们只用到了其中的一个）我们在调试二次开发的程序时，有两种方法，一种是将我们的程序编译成以 exe 结尾的可应用程序，一种是将程序编译为 DLL（动态链接库）。前者多用于开发人员的调试，后者多应用于发布已经完成的结果。

通过该 LIB 名称我们便可知晓，protoolkit.lib 对应于 EXE 文件，protk\_dll.lib 则对应于产生的 DLL 文件。因此，对于不同的情况，我们只需要在工程中添加相应的 lib 即可。这点了解以后，我们便可以开始配置开发环境了。（注意：在 PROE 安装目录下还有一个名称为 Prodevelop 的文件夹，其中的文件结构与 TOOLKIT 中基本一致，此处不再介绍）

环境配置一共分为三大步：首先要告诉 VC 你将要使用到的函数的头文件的路径，然后要告诉它，导出这些函数的库的路径，最后还要指明具体使用到了哪些库。我们现在来按照这个顺序配置适合我们自己的开发环境。

打开 VC++6.0，选择文件 —>新建，弹出如下对话框：按照图示选择 MFC AppWizard(dll) 选项，在工程名称中输入你的工程名称，并设置工程路径。如图（图 1-1）：设置完成后，点击确定。

系统弹出如图 1-2 所示的对话框，此处选择“动态链接库使用共享 MFC DLL”单选框。点击确定。（后面出现的其他对话框皆选确定便可）此时，系统便自动生成了一个基于 MFC 的动态链接库样本，接下来，我们只要在该样本中添加一些我们所要的功能便可。（当然，除了这种方法之外，还有一种更直观的方法，那就是 makefile 方法，此方法笔者将在以后的章节中描述，请各位读者切莫心急）



图 1-1

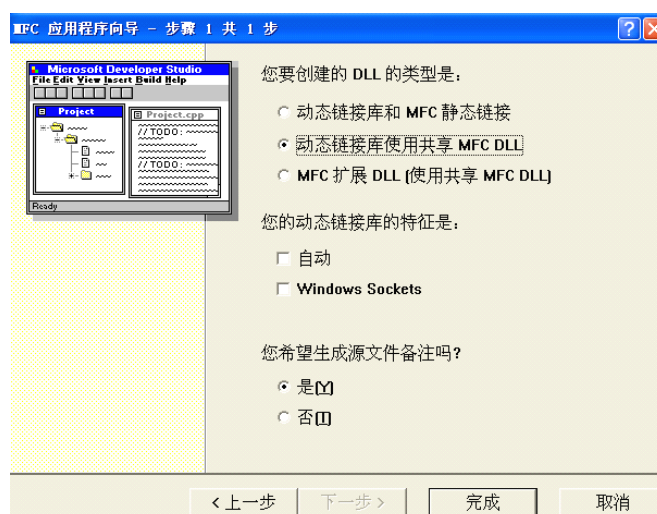


图 1-2

现在，环境配置正式开始。

首先给 VC 指定头文件路径：选择“工具”——>”选项”，系统弹出选项对话框（图 1-3）

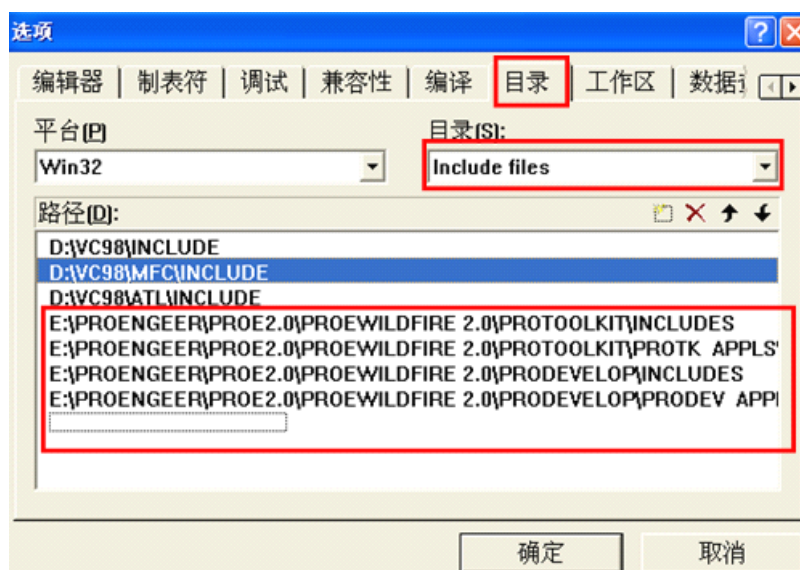


图 1-3

按照图示指定头文件路径，由于笔者的安装路径比较长，所以有部分内容被挡住了，为了让初学者明白其内容，特注明于此：（路径分别是：）

```
E:\PROENGINEER\PROE2.0\PROEWILDFIRE 2.0\PROTOOLKIT\INCLUDES
E:\PROENGINEER\PROE2.0\PROEWILDFIRE 2.0\PROTOOLKIT\PROTK_APPLS\INCLUDES
E:\PROENGINEER\PROE2.0\PROEWILDFIRE 2.0\PRODEVELOP\INCLUDES
E:\PROENGINEER\PROE2.0\PROEWILDFIRE
2.0\PRODEVELOP\PRODEV_APPLS\INCLUDES
```

这里的路径分为两块，一块是 TOOLKIT 中的路径，一块是 PRODEVELOP 中的路径。下面我们来设置该函数库的路径。

仍旧在这个界面下，我们设置这个库文件路径：（如图 1-4）

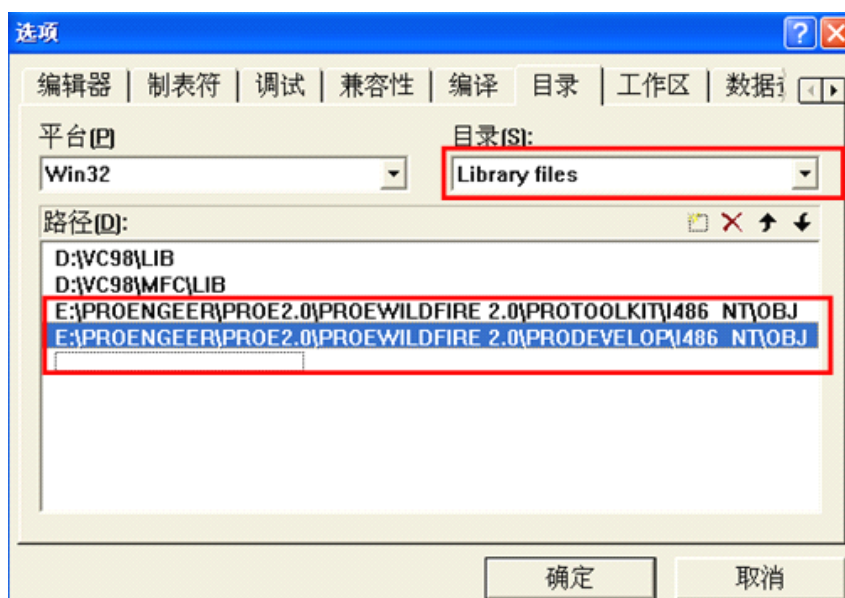




图 1-4

以上路径分别是：

E:\PROENGINEER\PROE2.0\PROEWILDFIRE 2.0\PROTOOLKIT\I486\_NT\OBJ

E:\PROENGINEER\PROE2.0\PROEWILDFIRE 2.0\PRODEVELOP\I486\_NT\OBJ

最后，我们要指定将要使用到的具体的库文件。

（注意：在使用 VC 的时候，系统有两个编译版本，默认的是 Debug 版本，也就是调试程序时所用到的版本，这个版本下编译出来的文件在别人是不能使用的。还有另外一个版本是 Release 版本，这个版本也叫发布版，发布版可以在他人的计算机上使用，当然这只是 VC 的使用界限，对于 PROE 的 TOOLKIT 来讲，一个完整的程序如果需要在其他的计算机上使用，必须使用 PROE 提供的解锁工具解锁（盗版的除外），对于这两个版本的库文件指定有些细微的差别。）

在工具栏上右键单击鼠标，选择“自定义”，弹出自定义属性页，（如图 1-5）

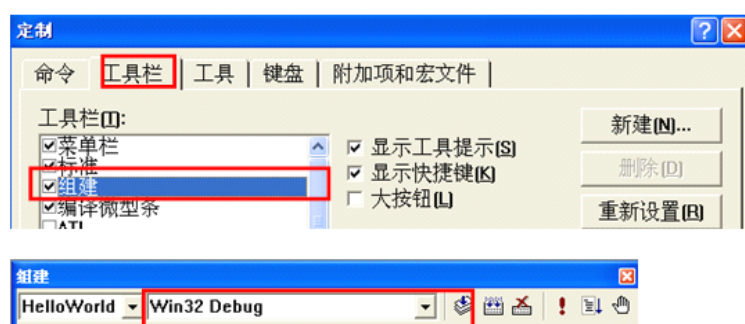


图 1-5

按照图示将“组建”工具显示于工具栏上。可以看到，当前默认的版本是 Debug 版本，如果要在该版本下编译，需要如下指定库文件：

选择“工程” —> “设置” 系统弹出如图 1-6 所示对话框

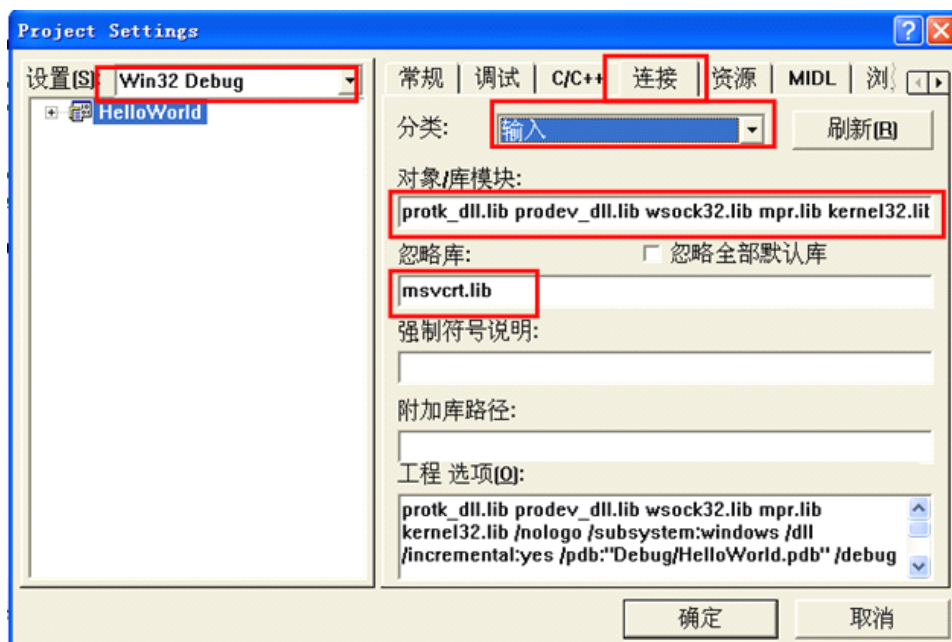


图 1-6

其中方框部分是要注意的地方。

在“对象/库模块”栏位中，笔者的配置为： protk\_dll.lib prodev\_dll.lib wsock32.lib mpr.lib kernel32.lib

仍旧在此界面下，我们将 Release 版本也做如下设置，其中方框部分是要注意的地方。

设置如图 1-7 所示：

（注意：在 Release 版本中所忽略的库为 MSVCRTD.LIB，这与 Debug 版本 MSVCRT.LIB 是不一样的，当向工程添加了 TOOLKIT 源程序后，这两个需要调换，后续会有说明）

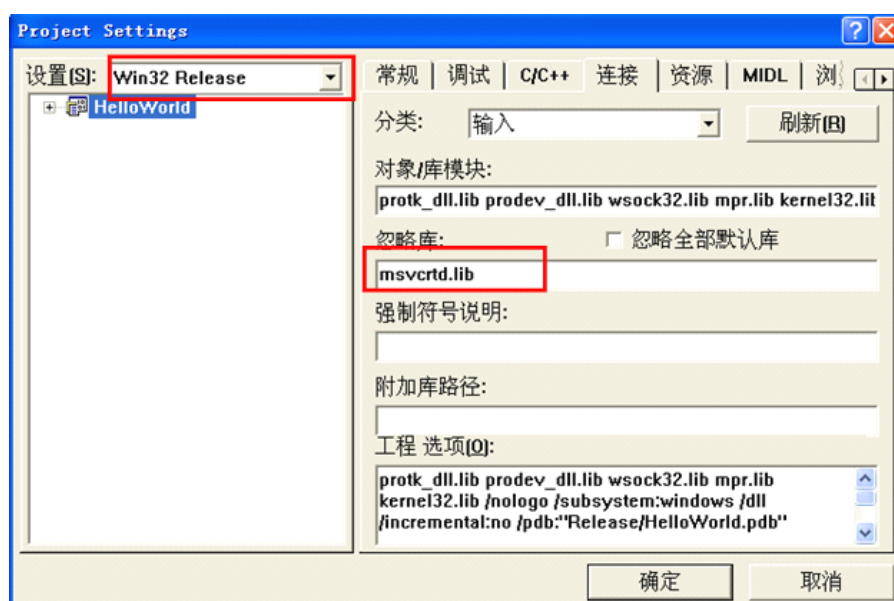


图 1-7

这样，我们的二次开发环境便设置完成了。利用刚才拖拽到工具栏上的“组建”工具，快速切换于 Release 版本与 Debug 版本之间，编译均为零警告。（此时还未添加 **toolkit** 源程序，当我们后续添加了源程序后，这两个版本的设置刚好需要互换。）

现在，将本工程保存后便可以退出了。（以后还要用到这个工程）需要说明的是，以后如果利用 VC 建立新工程，则需要按照前面的方法重新指定详细的库文件名称，其他路径 VC 将自动保存。

小结一下：配置环境共有三大步骤，首先指定头文件路径，然后指定库文件路径，最后列出详细的库文件名称。（在本书的后续章节中将要讲述的自定义函数库文件所采用的也是这个机制）。

## 环境配置二：野火 4.0 搭配 VS2005

这套组合在环境的配置上与前面的组合类似，依旧是三个步骤，只不过是界面改进了而已，由于 VS2005 中取消了在 VC++6.0 中的类向导工具，所以，在使用上会对那些已经熟悉 VC++6.0 的用户造成一些不便。

对于 PROE4.0 来讲，其文件结构与 2.0 是一致的，读者可参照前面的描述来回顾相关内容，下面请随我一起来配置这套组合下的二次开发环境。

首先打开 VS2005 中的 VC 开发环境。

依次选择 文件 —> 新建 —> 项目，系统弹出项目对话框（图 1-8）

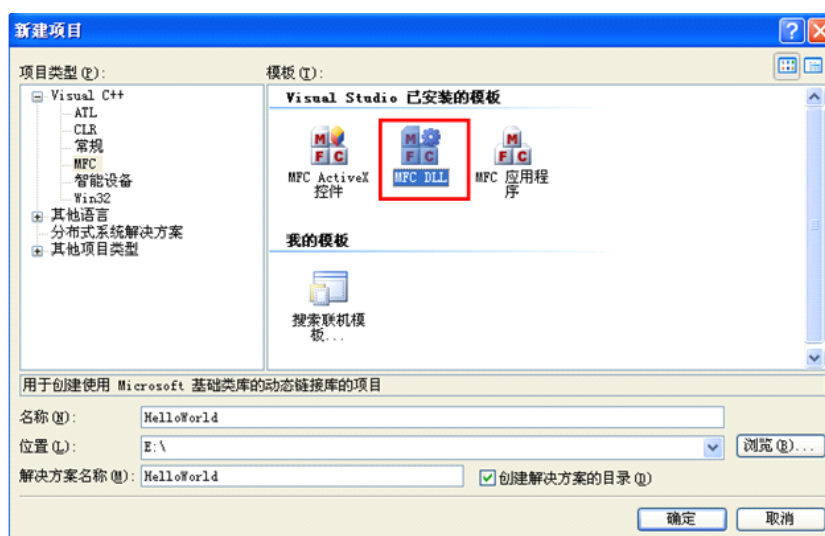


图 1-8

选择如图所示的 MFC DLL 项目，输入项目名称并设置项目路径，点击确定。

在接下来的对话框中选择“使用共享 MFC DLL 的规则 DLL”（默认的也是这个选项），点击完成后，与 VC++6.0 一样，系统自动为我们搭建了一个 DLL 的开发环境，现在，我们要在这个环境下配置 TOOLKIT 开发所需要的头文件和库文件路径。

依次在菜单中选择：工具—> 选项，系统弹出选项对话框。在该对话中选择“项目和解决方案”一栏中的“VC++ 目录”选项，并参照图 1-9 的方法设置头文件路径

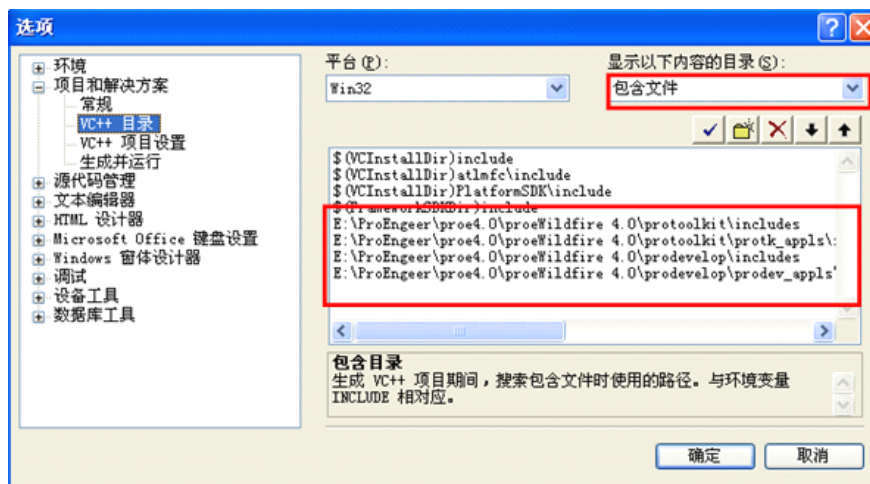


图 1-9

笔者的设置为：

E:\ProEngineer\proe4.0\proeWildfire 4.0\protokit\include  
E:\ProEngineer\proe4.0\proeWildfire 4.0\protokit\protk\_appls\includes  
E:\ProEngineer\proe4.0\proeWildfire 4.0\prodevelop\includes  
E:\ProEngineer\proe4.0\proeWildfire 4.0\prodevelop\prodev\_appls\includes

仍旧在该对话框下，设置库文件路径，如图 1-10：

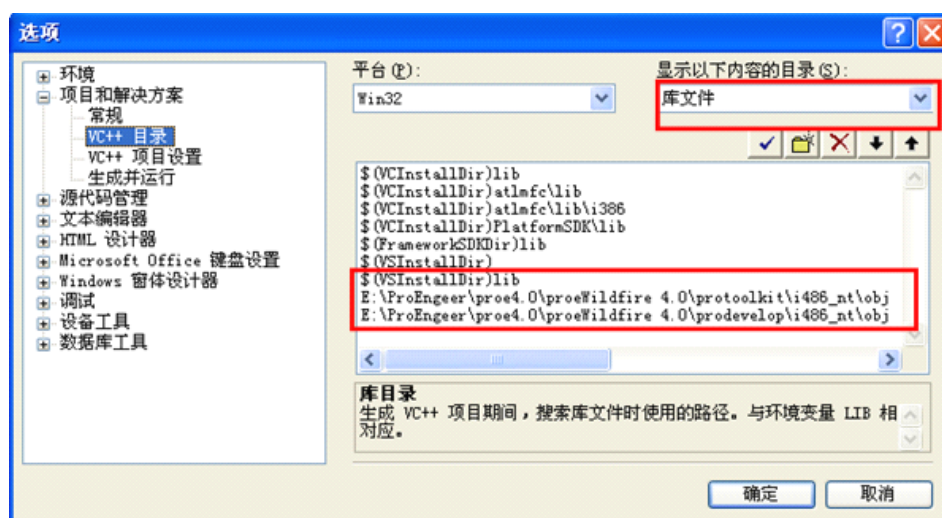


图 1-10

设置好头文件和库文件路径后，就还差最后一步了，现在我们要来指定使用中的具体库文件。

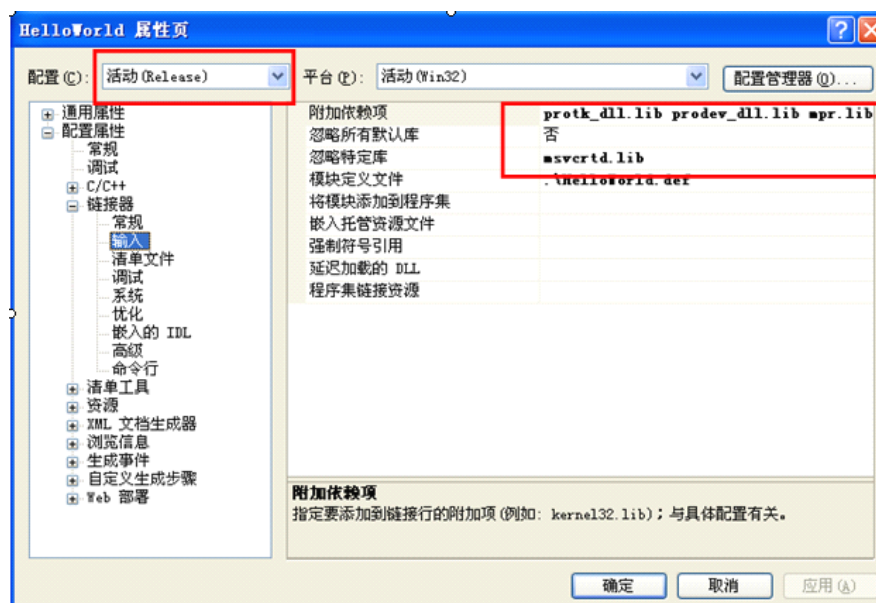


图 1-11

点击“项目” —> “属性”，系统弹出该项目的属性对话框。按照图 1-11 的方法设置工程的连接属性。

笔者的“附加依赖关系”中为：protk\_dll.lib prodev\_dll.lib mpr.lib psapi.lib（笔者这里一共用到了四个库，当然，读者可以根据自己需要，添加其他额外的库文件）

在 Release 版本下忽略的库文件为：msvcrt.lib（与 VC++6.0 中是不一样的）。设置完毕后，在 Release 版本下编译连接完全没有问题。（Debug 版本的设置于此类似，读者可自行尝试）

至此，环境设置便讲完了。顺带说一下，VC2005 也可以开发 野火 2.0，但是，VC++6.0 却不能开发野火 4.0。（环境配置非常麻烦）。当使用 VC2005 开发 野火 2.0 的时候，注意使用的 TOOLKIT 库文件为：protk\_dllmd.lib 和 prodev\_dllmd.lib 同时，由于 VC2005 中已经加入了对宽字符 (WString) 的定义，在编译的时候需要在头文件中注释掉相关的定义。有兴趣的读者可以尝试去设置这个环境。（笔者曾成功设置过该环境）

## 第二节：菜单

这一节笔者将向大家介绍如何在 PROE 中挂上自定义的菜单，笔者介绍的菜单不会仅仅是文字型菜单（目前市面上的书籍都讲的是纯文本菜单），笔者的菜单将包含：文本，自定义图标，文本快捷键，自定义工具栏，以及右键菜单（包含模型树右键菜单和图形窗口右键菜单）。

### 第一个普通菜单

菜单可以说是程序与 PROE 进行交互的重要通道，同时，一个具有独特图标的菜单不但能够让使用者很快联想到该菜单的功能，还能拖拽到工具栏上以加速工作进度。下面我们一起来建立第一个“HelloWorld”程序。

首先回到在前一章节中建立的环境中，（这里以 VC++6.0 搭配 野火 2.0 为例）在工程中添加一个名称为“MyFirstMenu.cpp”的 C++源文件。整个文件的内容如下：（读者如果对其中的某些函数不清楚，可以先不要理会）

```
// MyFirstMenu.cpp
//
#include "stdafx.h"
#include "ProUtil.h"
#include "ProMenu.h"
#include "ProMessage.h"
#include "ProMenuBar.h"
#define PRO_USE_VAR_ARGS 1
//设置菜单在不同模式下的状态
static uiCmdAccessState AccessDefault(uiCmdAccessMode)
{
    return ACCESS_AVAILABLE;
}
// 自定义函数
void Test()
{
    AfxMessageBox("Hello World");
}
//函数入口，PROE 与 VC 的接口
extern "C" int user_initialize()
{
```

```

ProError status;
ProFileName MsgFile;
ProStringToWstring(MsgFile, "IconMessage.txt");
uiCmdCmdId PushButton1_cmd_id, PushButton2_cmd_id;
status = ProMenubarMenuAdd("MainMenu", "Function", "Help", PRO_B_TRUE, MsgFile); //主菜单

ProCmdActionAdd("PushButton1_Act", (uiCmdCmdActFn)Test, // Test 是动作函数
    12, AccessDefault, PRO_B_TRUE, PRO_B_TRUE, &PushButton1_cmd_id);
ProMenubarmenuPushbuttonAdd("MainMenu", "PushButton1", "FirstButton", "this button will show a
    message", NULL, PRO_B_TRUE, PushButton1_cmd_id, MsgFile); //第一个菜单
ProCmdIconSet (PushButton1_cmd_id, "PushButton1.gif"); //设置菜单图标
ProCmdDesignate(PushButton1_cmd_id, "FirstButton", "this button will show a message", "show first
    button", MsgFile); //使菜单可以拖拽到工具栏

ProCmdActionAdd("PushButton2_Act", (uiCmdCmdActFn)Test, // Test 是动作函数
    uiCmdPrioDefault, AccessDefault, PRO_B_TRUE, PRO_B_TRUE, &PushButton2_cmd_id);
ProMenubarmenuPushbuttonAdd("MainMenu", "PushButton2", "secondbutton", "this button will show a
    message", NULL, PRO_B_TRUE, PushButton2_cmd_id, MsgFile); //第二个菜单
return status;
}

//菜单终止时所调用的函数，一般可用来做一些程序的清理工作
extern "C" void user_terminate()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
}

```

注意，为了便于说明程序功效，本源文件中没有添加错误处理机制，读者可以自行添加错误处理机制。在编译该源文件时，系统会报错，这时候需要将环境设置中“忽略库”选项中的 MSVCRTD.lib 与 MSVCRT.lib 调换。

下面我们来了解下这篇源代码都做了什么工作。

对于已经有注解的地方，笔者不再叙述，这里主要介绍入口函数都做了哪些事情。程序首先是在 PROE 菜单系统的“HELP”菜单后面添加了一个名称为“Function”的自定义菜单栏，后面的两个菜单都是在此基础上添加的。（程序中提到的 IconMessage.txt 是一个“菜单描述文件”，该文件中用来存放自定义菜单的描述）然后，程序在刚才的 Function 菜单下添加了一个子菜单：“FirstButton”，为了说明后续子菜单如何添加，笔者接着在其后右添加了"secondbutton"子菜单。当我们点击菜单时，需要激发相应的处理函数，该函数由 ProCmdActionAdd 来指定，本例中，两个子菜单所指定的都是自定义函数 Test，读者一定会问，这么多的函数，我怎么知道每一个是什么用法呢，这也是笔者马上要讲到的内容。回顾 TOOLKIT 文件夹的结构，我们打开其帮助文件，笔者的路径为：



E:\ProEngineer\proe2.0\proeWildfire 2.0\protoolkit\protkdoc\ IENoSwing.html

打开后可看见帮助文件的使用界面。如图 2-1， 该文件的左边是一个函数搜索引擎。（注意，如果当前读者的计算机上没有安装 JAVA 运行时环境， 该引擎将是不可见的， PROE 专门提供了一个默认的 JAVA 运行时环境， 用户只需要手动安装就可以了， 该安装文件的路径在： E:\ProEngineer\proe2.0\proeWildfire 2.0\i486\_nt\obj\j2re-1\_4\_2\_03-windows-i586-p.exe ， 野火 4.0 中， 此文件的安装路径为： E:\ProEngineer\proe4.0\proeWildfire 4.0\i486\_nt\obj\jre.exe）， 点击帮助文件左边函数搜索引擎中的“Find”， 系统弹出如图 2-2 的函数搜索器。按照图示在文本栏位输入想要查询的函数， 系统会将查询结果显示于下方的列表框， 通过点击列表框中的函数， 将可以看到该函数的详细说明， 包括函数定义， 函数的归属头文件， 以及一些范例。读者若想高效开发 PROE， 该工具必须要非常熟悉。

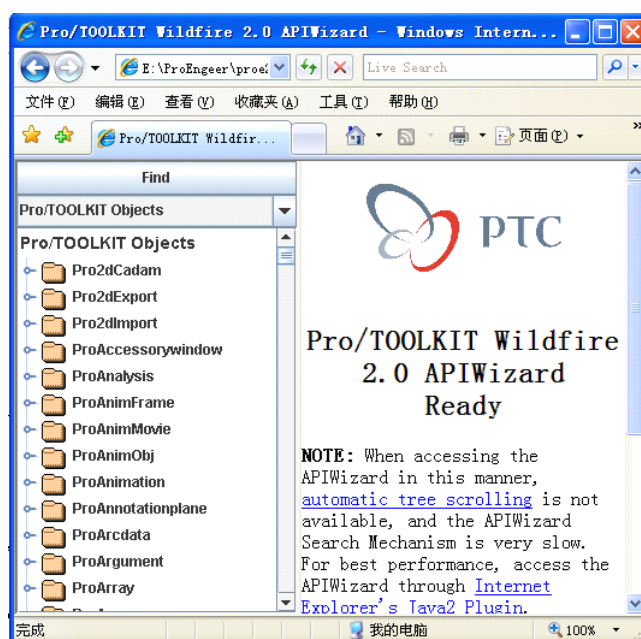


图 2-1

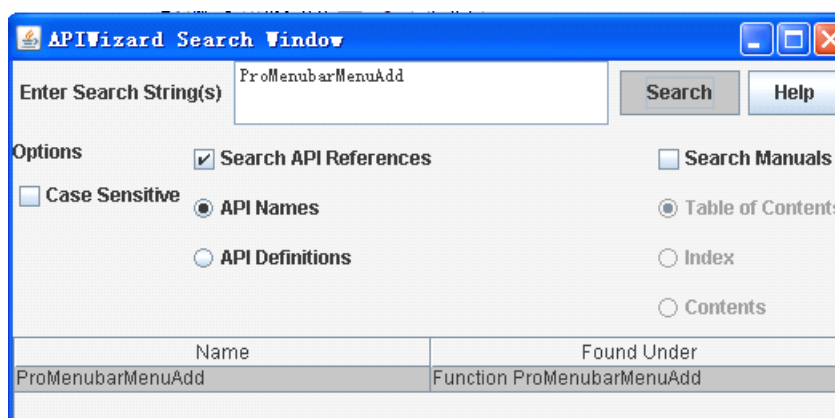


图 2-2

对于上面源文件中的系统函数，读者可以自行在这个函数搜索器中查看，笔



者就此略过。该源文件编译成功后,下面我们就来将它加载到 PROE 中去。PROE 加载菜单有两个要求,一是必须要有注册文件, 另外就是必须要有相应的菜单说明文件(菜单文件只能被加载一次,如果在没有退出 PROE 的情况下,菜单文件被改变了, 改变后的结果是不会显示在 PROE 中的,只有重新启动)。下面我们来分别建立这两个文件。

为了方便说明,我们假设在 D 盘根目录进行操作。(当然读者可以在任何有读写权限的文件夹下操作)。首先在 D 盘建立一个名称为“MyTestMenu”的文件夹,然后在该文件夹下建立一个名称为“text”的文件夹,在该“text”文件夹下再建立一个名称为“Resource”的文件夹。接下来,将我们在 VC 中编译好的 DLL 文件拷贝至“MyTestMenu”文件夹下, 同时建立一个名称为“ProTK.dat”的文件,文件内容为:

```
NAME MyTestMenu
STARTUP dll
EXEC_FILE D:\MyTestMenu\HelloWorld.dll
TEXT_DIR D:\MyTestMenu\text
ALLOW_STOP TRUE
REVISION 2009
END
```

**注意: 如果在野火2.0中开发的程序要在野火4.0中使用,则在注册文件中还需要添加如下语句:  
“unicode\_encoding FALSE” 这一点在野火4.0中的帮助文件中有详细介绍。**

现在我们需要建立一个菜单说明文件,该文件名称为: IconMessage.txt, 文件内容为:

```
Function
UserFunction
#
#
FirstButton
Demo1
#
#
secondbutton
Demo2
#
#
this button will show a message
this button will show a message
#
#
show first button
show first button
#
#
```

注意: 该文件中的内容必须与源程序中的相关内容一致, 否则加载菜单会发生失败。

将该菜单文件放置在刚才建立的“text”文件夹下。到这里还有最后一个步骤，就是添加图标的资源文件。由于我们在源程序中指定了图标文件：PushButton1.GIF，所以，必须将该文件放置在“Resource”文件夹下，读者自己在制作图标的时候注意不要太大，一般在 25X25 就可以了。至此，所有的准备工作都完成了，下面我们就来将该菜单加载到 PROE 中去。

启动 Proe， 点击“TOOLS”菜单下的 “Auxiliary Applications”，如图 2-3，系统弹出应用程序注册对话框（图 2-4）

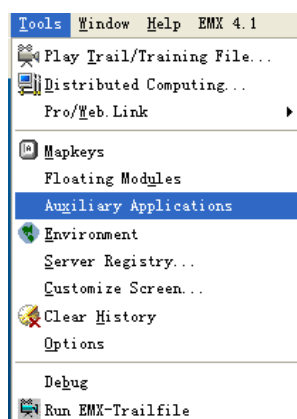


图2-3

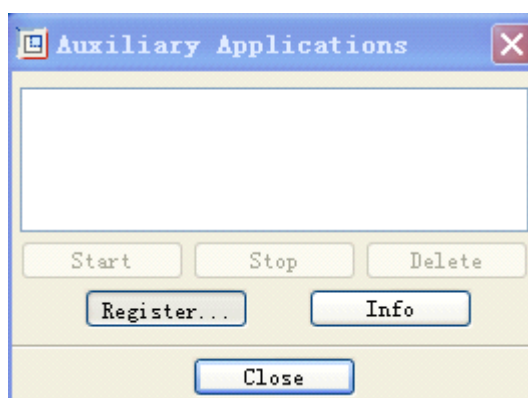


图2-4

点击“Register”，从弹出的对话框中找到我们刚才建立的：D:\MyTestMenu\ProTK.dat 文件，打开。如图 2-5，

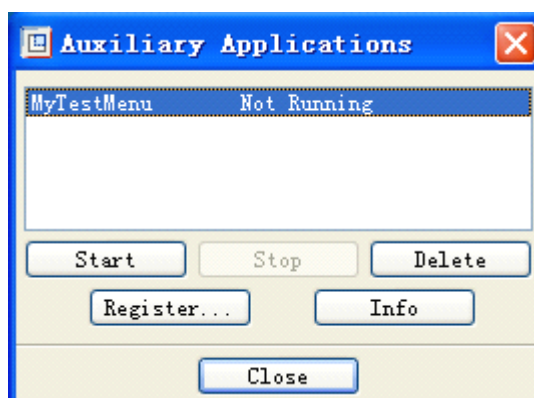


图2-5

选择我们注册的这个菜单，点击“Start”按钮，系统便加载了我们的菜单。执行效果如图 2-6:



图2-6

为了方便我们的工作，我们可以将命令“Demo1”拖拽到工具栏上以供使用，方法如下： 点击“Tools”下的“Customize Screen”命令，系统弹出客户化指令对话框，如图 2-7



图2-7

将此导入系统的自定义图标拖拽到工具栏上，点击确定按钮退出。这样，以后直接点击工具栏上的这个快捷按钮便可以执行其所对于的命令了。我们的第一个普通菜单便成功完成了。像这种形态的菜单还有级联菜单与选项菜单，这里笔者就不多讲了，有兴趣的读者可以参考其他书籍，或参考帮助文件。通常情况下，本例中介绍的这种菜单已经完全够用了。如果实在不够，PROE 还支持加载多个菜单条，读者可自行尝试。

下面介绍另外一种形态的菜单：浮动菜单。

## 第一个浮动菜单

浮动菜单在较老版本的 PROE 中比较常见，当前版本中，为了形象化指令，很多命令都已经用工具条替代，不过作为对菜单的一种补充，笔者觉得还是有必要对这种菜单加以说明。浮动菜单的形态如图 2-8 所示：



图2-8

下面请读者与我一起来创建一个浮动菜单。

首先，在上面的普通菜单的文件夹中建立一个文件夹来放置浮动菜单文本，结合上面的例子，我们应该在 D:\MyTestMenu\text 下建立一个名称为 “menus” 的子文件夹，打开这个文件夹，在里面建立一个名称为 “Sample.mnu” 的文件。（可先建立一个 Sample.txt，然后把后缀修改为 .mnu 即可），在该文件中填写如下内容：// Sample.mnu 的内容

```
User_functions
#
#
The#first
activate Menu button1
#
The#second
activate Menu button2
#
The#third
activate Menu button3
#
Done
Exit
#
```

写完后保存。然后我们在程序中去指定这个菜单， 假设我们用函数：**ShowYourMenu** 来实现这个功能。 函数体如下：

```
void First_fun()
{
    AfxMessageBox("first fun is working"); // 自定义功能函数
}

void Second_fun()
{
    AfxMessageBox("second fun is working");// 自定义功能函数
}

void Third_fun()
{
    AfxMessageBox("third fun is working");// 自定义功能函数
}

int ShowYourMenu() // 浮动菜单入口函数
{
    int menu_id;
    ProError err;
    ProMode mode;
    err = ProModeCurrentGet(&mode);
    if(mode == PRO_MODE_PART || mode == PRO_MODE_ASSEMBLY) //指定只能在PART和ASM模式下使用
    {
        ProMenuFileRegister("User_functions", "Sample.mnu", &menu_id);
        ProMenubuttonActionSet("User_functions", "The first", (ProMenubuttonAction)First_fun,
                                NULL, 0);
        ProMenubuttonActionSet("User_functions", "The second",
                                (ProMenubuttonAction)Second_fun, NULL, 0);
        ProMenubuttonActionSet("User_functions", "The third", (ProMenubuttonAction)Third_fun,
                                NULL, 0);
        if(mode == PRO_MODE_PART) // 设置在 PART 下"The first"是不可见状态
        {
            err = ProMenubuttonVisibilitySet("User_functions", "The first", PRO_B_FALSE);
        }

        ProMenuCreate(PROMENUTYPE_MAIN, "User_functions", &menu_id);
        ProMenuProcess("User_functions", &menu_id);
    }

    return true;
}
```

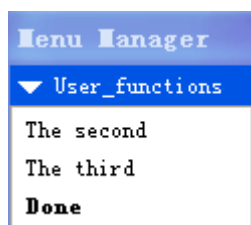
将该段代码粘贴到前面我们建立的普通菜单的源文件：MyFirstMenu.cpp 中。然后将原先的 Test 函数改写如下：

```
int ShowYourMenu(); //调用该函数前必须声明
void Test()
{
    //AfxMessageBox("Hello World");
    ShowYourMenu();
}
```

编译后得到 Release 版本的 HelloWorld.dll 文件， 将该文件覆盖之前我们在 D:\MyTestMenu 中的 HelloWorld.dll ， 完成后，按照前面的方法注册运行，结果如下：

（注意，有可能会无法编译或无法替代，如果当前 PROE 中正在使用这个文件，那么有两种方法可以结束这种状态，一种就是直接关闭 PROE 后，重新开启，另外一种就是按照前面介绍的方法调出注册程序对话框，选择要终止的对象后，点击“停止”，以退出对该文件的使用）

由于在程序中，我们指定了这个菜单的工作状态必须在 PART 或 ASM 下使用，所以，重新注册该文件后，我们必须打开或新建一个 PART 或 ASM 档案，然后点击“UserFunction”下的“Demo1”， 这时将在当前屏幕的右上角出现如下浮动菜单：（图 2-9）



Part 模式



Asm 模式

图 2-9

点击相应的命令后，将触发对应的函数功能。

至此，第二种形态的菜单介绍完毕。读者可在此基础上增加菜单长度了。

读者可能已经注意到， 笔者在介绍这些菜单的时候，并没有长篇大论，而是尽量使用了简短有效的代码来说明，基本上都是点到即止，所以，要想实现更多的功能，读者需要自己在笔者的基础上去多加练习。因为菜单是一切功能的前提，所以这部分的内容读者必须要精通。

在做了短暂的休息后，笔者将要给大家介绍第三种形态的菜单： 右键菜单。

## 第一个右键菜单

右键菜单涉及的东西稍微要比前面两种形态的菜单多点，不过大家也不用着急，请与我一起来制作这个菜单。

首先，我们要建立一个程序的源文件。 假设该文件的名称为： POPMENU .CPP

文件内容如下：

```
// POPMENU .CPP
#include "stdafx.h"

#include <ProUtil.h>
#include <ProMenu.h>
#include <ProMessage.h>
#include <ProMenuBar.h>
#include <ProNotify.h>
#include <ProPopupMenu.h>
#include <ProSelbuffer.h>
#include <ProArray.h>

#define PRO_USE_VAR_ARGS 1

ProError UserPopmenuNotification(ProMenuName name);

void popaction();

uiCmdAccessState Accessall(uiCmdCmdId command, ProAppData appdata, ProSelection* sel_buffer)
{
    return (ACCESS_AVAILABLE);
}

static uiCmdAccessState AccessDefault(uiCmdAccessMode)
{
    return ACCESS_AVAILABLE;
}

int UserPopupmenusSetup() // 模型树右键菜单入口函数
{
    ProError status;
```

```

uiCmdCmdId cmd_id;
ProFileName MsgFile;
ProStringToWstring(MsgFile, "Demo.txt");

// prepare action for right_mouse_menu
status = ProCmdActionAdd("mypopmenu1_act", (uiCmdCmdActFn)popaction,
                        uiProe2ndImmediate, AccessDefault, PRO_B_FALSE, PRO_B_FALSE, &cmd_id);
//add new menu to the model tree right_mouse_menu
status = ProMenubarmenuPushbuttonAdd("ActionMenu", //model tree menu of system
                                     "popmenu1", //your pop menu button name
                                     "mypopmenu1", // the lable(need exist in the msgfile)
                                     "help info", // help info(need exist in the msgfile)
                                     NULL, PRO_B_TRUE, cmd_id, MsgFile);

//add the popup menu notification to the session
status = ProNotificationSet(PRO_POPUPMENU_CREATE_POST, (ProFunction)UserPopmenuNotification);
return 0;
}

ProError UserPopmenuNotification(ProMenuName name) //点击右键时产生的消息
{
    ProPopupMenuId menu_id;
    uiCmdCmdId cmd_id;
    ProError status;
    ProLine lable1, lable2;
    ProLine help;

    //check if the menu being created matches the menu name we want
    ProPopupMenuIdGet(name, &menu_id);
    if(strcmp(name, "Sel Obj Menu") == 0)
    {
        // add exist command (popmenu1) to pop menu
        status = ProCmdCmdIdFind("mypopmenu1_act", &cmd_id); // "mypopmenu1_act" 在前面已声明
        ProStringToWstring(lable1, "first pop menu");
        ProStringToWstring(help, "help info");
        status = ProPopupMenuButtonAdd(menu_id, PRO_VALUE_UNUSED,
                                       "thepopmenu1", // the lable(need exist in the msgfile)
                                       lable1, help, cmd_id,
                                       (ProPopupMenuAccessFunction)Accessall,
                                       NULL);

        // add new command to pop menu
        uiCmdCmdId cmd_id_2;
        status = ProCmdActionAdd("mypopmenu2_act",
                                (uiCmdCmdActFn)popaction,

```



```

        uiProe2ndImmediate,
        AccessDefault,
        PRO_B_FALSE, PRO_B_FALSE, &cmd_id_2);
ProStringToWstring(lable2, "second pop menu");
status = ProPopupMenuButtonAdd(menu_id, PRO_VALUE_UNUSED,
                                "thepopmenu2", // the lable(need exist in the msgfile)
                                lable2, help, cmd_id_2,
                                (ProPopupMenuAccessFunction)Accessall,
                                NULL);
    }
return PRO_TK_NO_ERROR;
}

void popaction() // 定义菜单函数的动作
{
    ProError status;
    ProSelection* ret_buff;
    status = ProSelbufferSelectionsGet(&ret_buff);
    if(status != PRO_TK_NO_ERROR) return;

    int num;
    status = ProArraySizeGet((ProArray)ret_buff, &num);
    if(status != PRO_TK_NO_ERROR) return;

    if(num == 1)
    {
        ProModelitem p_modelitem;
        ProSelectionModelitemGet(ret_buff[0], &p_modelitem);
        if(p_modelitem.type != PRO_PART)
            return;
        AfxMessageBox("you choose a part");
    }
    ProArrayFree((ProArray*)&ret_buff);
}

```

这个源代码文件可以通过 VC 中“文件”——>“新建”中的“C++ Source File”选项来建立，也可以先通过其它方式建立好后，加载进 VC 的编译器，方法如图所示：

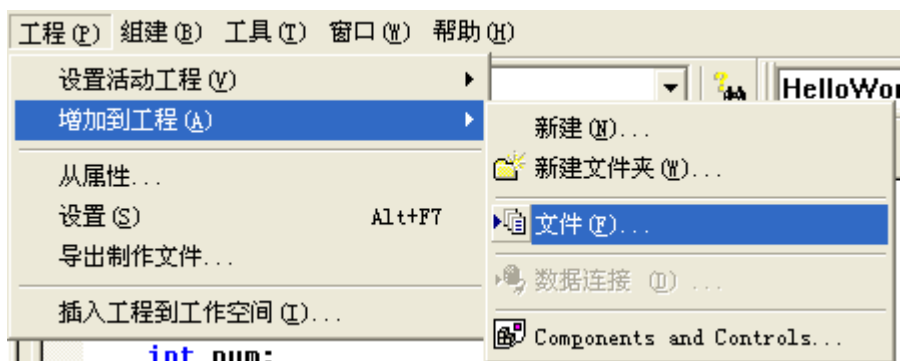


图 2-10

选择该文件后，点击确定即可。

在本例中，我们把这个文件加入我们前面建立的工程：“Hello World”中。如图 2-10

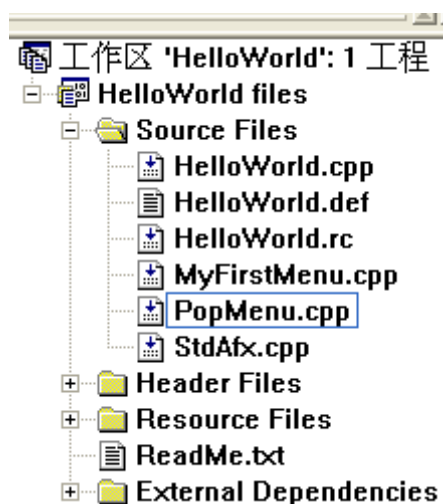


图 2-11

为了使本源程序中的函数能够与前面所建立的其他函数所关联，我们需要修改 MyFirstMenu.cpp 中的函数：user\_initialize，修改结果如下（加粗部分为添加的新内容）

```
int UserPopupmenusSetup();    //声明右键菜单入口函数
extern "C" int user_initialize()
{
    ProError status;
    ProFileName MsgFile;
    ProStringToWstring(MsgFile,"IconMessage.txt");
    uiCmdCmdId PushButton1_cmd_id, PushButton2_cmd_id;
    status=ProMenubarMenuAdd("MainMenu", "Function", "Help", PRO_B_TRUE, MsgFile);

    ProCmdActionAdd("PushButton1_Act", (uiCmdCmdActFn)Test,
                    12, AccessDefault, PRO_B_TRUE, PRO_B_TRUE, &PushButton1_cmd_id);
```

```

ProMenubarmenuPushbuttonAdd("MainMenu","PushButton1","FirstButton","this button will show a
message",NULL,PRO_B_TRUE,PushButton1_cmd_id,MsgFile);
ProCmdIconSet (PushButton1_cmd_id, "PushButton1.gif");
ProCmdDesignate(PushButton1_cmd_id, "FirstButton", "this button will show a message", "show first
button", MsgFile);
ProCmdActionAdd("PushButton2_Act",(uiCmdCmdActFn)Test,uiCmdPrioDefault,AccessDefault,PRO_B_
TRUE,PRO_B_TRUE,&PushButton2_cmd_id);
ProMenubarmenuPushbuttonAdd("MainMenu","PushButton2","secondbutton","this button will show a
message",NULL,PRO_B_TRUE,PushButton2_cmd_id,MsgFile);
ProCmdIconSet (PushButton2_cmd_id, "PushButton2.gif");

UserPopupmenusSetup();
return status;
}

```

修改完成后，点击编译，得到最新的 HelloWorld.dll 文件。将该文件替换：D:\MyTestMenu 下的同名文件。这样，从代码段，我们便完成了右键菜单的设计，接下来我们要制作于此对于的菜单描述文件。

在 D:\MyTestMenu\text 下建立一个 Demo.txt 文件，其内容如下：

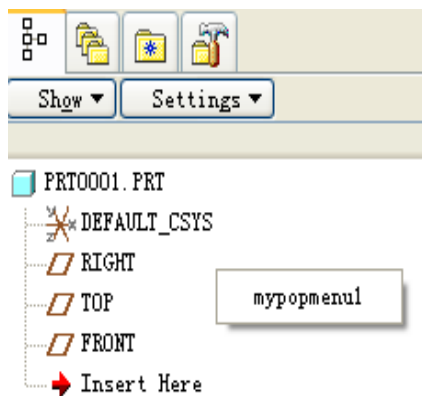
```

mypopmenu1
mypopmenu1
#
#
help info
help info
#
#
thepopmenu1
thepopmenu1
#
#
thepopmenu2
thepopmenu2
#
#

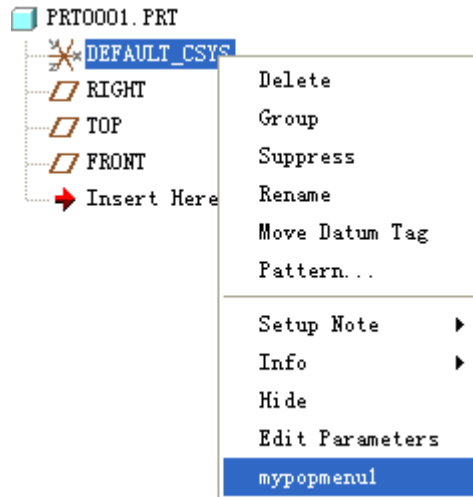
```

细心的读者会发现，该文件中的内容与程序段的内容是一致的。没错，这个文件中的内容必须与程序中的相关内容一一对应，否则菜单将无法加载成功。

到这里，我们的右键菜单便制作完毕了，打开 PROE，使用之前介绍的方法注册 HelloWorld.dll，用 PROE 打开（或新建）一个 Part 文件，在该模型的模型树窗口直接点击右键，可直接弹出该右键菜单，如果先在模型书中选择了一个对象，则该右键菜单将加载在系统右键菜单的末尾，如图 2-12 所示：



直接在模型树点击右键



选择对象后点击右键

图 2-12

当在图形窗口选择对象并点击右键后，结果将如图 2-13 所示：



2-13

## 补充内容：热键指定（快捷键的一种）

请大家仔细观察系统菜单与我们自己的菜单，细心的读者也许会发现，我们的自定义菜单是没有指定热键的，举个例子，拿 Help 来讲，细心的你会发现，在这个菜单的 H 字母下方有一条简短的横线，这就是说，如果我不使用鼠标，那么我在键盘下使用“Alt + H”便可以弹出该菜单下的子菜单。既然如此，那可不可在我们的菜单中也加入这样的元素呢？答案是可以的，而且及

其简单。回顾我们之前建立的菜单描述文件，我们只需要稍作修改就可以实现这个目的了，下面请看笔者修改后的：IconMessage.txt

内容如下：

```
Function
&UserFunction          注意此处笔者在菜单描述前面添加了一个“&”符号。
#
#
FirstButton
De&m01                 注意此处笔者在菜单描述中间添加了一个“&”符号。
#
#
secondbutton
Dem&o2                 符号“&”可添加在描述字符串的任意位置，不过对应的字母不能重复。
#
#
.....后续略.....
```

至此，对于菜单的介绍全部结束。对于笔者在介绍这三种形态的菜单时所用的那些函数，笔者会在后续的章节中有所提及，心急的读者也可以参照本章节前面介绍的 PROE 自带的函数查询工具自行查找相关函数的使用方法。由于本书默认读者已经具备初步的 C++/C 方面的知识，因此有些内容是一笔带过的。对于有疑惑的地方，一方面可以自己尝试从函数库中获取解决方案，另外，也可以通过使用本书页脚下的联系方式直接来与笔者沟通或登陆笔者网站（见序言的末尾）以获取技术支持（非诚勿扰.....^.^）。

## 第三节：对象的基本操作

前面两个章节都在介绍 TOOLKIT 的基本要素，从本章节开始，我们将进入正真的开发。请读者跟谁笔者一起进入 PROE TOOLKIT 开发的多彩世界。（从这里开始，笔者提供的功能将以函数为主，读者可自己将这些功能添加到菜单中运行）

### 信息输入/输出

信息在 PROE 中是非常重要的一个元素，在编译时它可以随时检测程序的状态，在发布后，它又起到提示用户的作用。同时，它还可以提示用户输入数据，以驱动程序的运行。下面让我们来一步一步的掌握它的用法。

首先来看下面的代码，类似代码可以在帮助文件 UserGuide 中找到。

```
#include "stdafx.h" // 此头文件是 VC 包含的，若不在 VC 下编译可以省略。
#define PRO_USE_VAR_ARGS 1 //这个定义是为了能够使用 ProMessageDisplay, 详细信//
                             息可以查询 ProMessage.h 文件。

#include <ProUtil.h>
#include <ProMessage.h>
#include <ProWstring.h>

void UserTestMessage()
{
    ProName msg_file;
    ProStringToWstring(msg_file, "TestMessage.txt"); //定义信息文件

    // 下面代码演示如何在消息窗口显示信息
    ProName value;
    ProStringToWstring(value, "this is a Wstring");
    //因为是宽字符，所以使用 " %0w "
    ProMessageDisplay(msg_file, "USER Info: %0w", value);
    //因为是普通字符所以使用 " %0s "
    ProMessageDisplay(msg_file, "USER Info: %0s", "this is a common string");
    //显示如何使用信息中的不同图标（结合结果理解）
    ProMessageDisplay(msg_file, "USER Info: %0s", "it's an Info icon");
    ProMessageDisplay(msg_file, "USER Critical: %0s", "it's a critical icon");
    ProMessageDisplay(msg_file, "USER Error: %0s", "it's an error icon");
}
```

```

ProMessageDisplay(msg_file, "USER Prompt: %0s", "it's a prompt icon");
ProMessageDisplay(msg_file, "USER Waring: %0s", "it's a waring icon");

// 下面代码演示如何从消息窗口读取信息
ProError err;
// 读取字符串
ProLine default_value;
ProStringToWstring(default_value, "default value"); //设置默认值
ProMessageDisplay(msg_file, "USER Prompt: Enter any string: |||%0w", default_value);

ProName read_val;
err = ProMessageStringRead (PRO_NAME_SIZE, read_val); // 读取输入的宽字符
if (err != PRO_TK_NO_ERROR && err != PRO_TK_GENERAL_ERROR)
    return ;
if (err == PRO_TK_GENERAL_ERROR) // 如果用户不输入则使用默认值
    ProWstringCopy (default_value, read_val, PRO_VALUE_UNUSED);
ProMessageDisplay(msg_file, "USER Info: we read a string value : %0w", read_val);

//读取整型数据
int range[2] = {0, 20};
int default_int_val = 10;
int read_int_val = 0;
ProMessageDisplay(msg_file, "USER Prompt: Enter any int value between %0d and %1d:
|||%2d", &range[0], &range[1], &default_int_val);
err = ProMessageIntegerRead (range, &read_int_val);
if (err != PRO_TK_NO_ERROR && err != PRO_TK_GENERAL_ERROR)
    return ;
if (err == PRO_TK_GENERAL_ERROR)
    read_int_val = default_int_val; //Using the default

ProMessageDisplay(msg_file, "USER Info: we read an int value : %0d", &read_int_val);

}

```

与本函数所对应的文本文件 TestMessage.txt 的内容如下：

```

%CIUSER Info: %0w
Info: %0w
#
#

%CIUSER Info: %0s
Info: %0s
#

```

#

%CCUSER Critical: %0s

Critical: %0s

#

#

%CEUSER Error: %0s

Error: %0s

#

#

%CPUSER Prompt: %0s

Prompt: %0s

#

#

%CWUSER Waring: %0s

Waring: %0s

#

#

%CPUSER Prompt: Enter any string: |||%0w

Prompt: Enter any string: |||%0w

#

#

%CIUSER Info: we read a string value : %0w

Info: we read a string value : %0w

#

#

%CPUSER Prompt: Enter any int value between %0d and %1d: |||%2d

Prompt: Enter any int value between %0d and %1d: |||%2d

#

#

%CIUSER Info: we read an int value : %0d

Info: we read an int value : %0d

#

#

本函数运行的结果如下：（图 3-1）



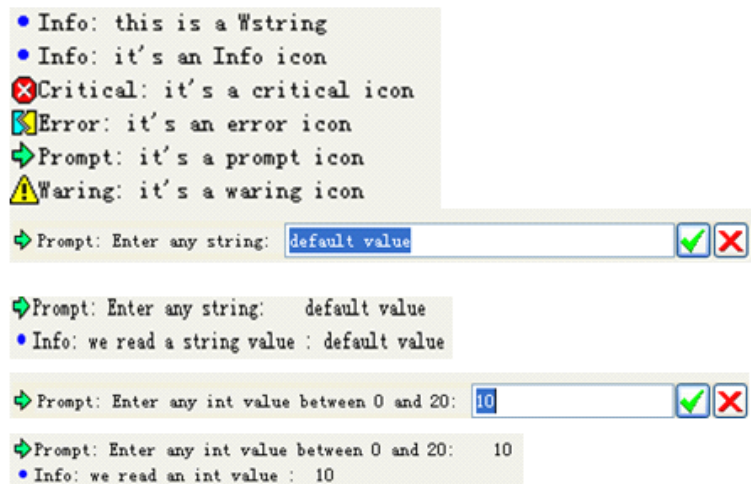


图 3-1

读者可能会被 TestMessage.txt 中的内容搞的晕头转向，不过不要紧，下面笔者结合程序运行结果简单介绍的做些介绍。

我们使用警告图标来讲解这个内容。

```
%CWUSER Warning: %0s
Warning: %0s
#
#
```

第一句中的%CW 代表警告图标，USER 表示来自用户指定。后面的 Warning: %0s 表示显示在窗口中的内容为 Warning: + 一个普通字符串。第一句相当于是标志符，在整个 TestMessage.txt 文件中必须唯一。与这句话所对应的程序为：

ProMessageDisplay(msg\_file, "**USER Warning: %0s**", "it's a warning icon"); 注意其中加粗的部分，该部分必须严格与 TestMessage.txt 中的%CW**USER Warning: %0s** 对应。

第二句是对第一句话的格式化输出，实际系统在向窗口输出信息时，所参考的便是这一句。从运行的结果看，在窗口显示的字符串为 Warning: + 一个普通字符串，其前面的“警告”图标由系统自动添加。

第三句是用于其他语言的翻译，比如你要想在某个环境下在英语与汉语之间切换，那么，这一句将起到作用。

最后一句是 PROE 为以后预留的一个接口。目前没有什么实际用途。

其它几个图标的内容于此类似。需要注意的是，在程序中指定的这些信息特征必须要有相应的 TXT 文件与其对应，否则系统会提示找不到信息。

## 选择对象

对象的选择在开发中也是非常重要的元素，若要操作一个模型，比如修改参数，尺寸，或者添加一个特征，首先你必须获得该模型的句柄。又或者，要测量两个物体的距离，那么你必须分别获取这两个物体的句柄。选择对象有多种方式，笔者下面给大家介绍两种常用的选取对象的方法：交互选择与自动选择。

首先介绍交互选择：即系统提示用户手动从图形窗口进行选择。在这个交互选择中又分别分为三种情况： 单选， 多选， 框选。（这里假设我们要在一个模型中选择面）

### 单选

```
#define PRO_USE_VAR_ARGS 1
#include <ProUtil.h>
#include <ProMessage.h>
#include <ProWstring.h>
#include <ProSelection.h>
#include <ProModelitem.h>
ProError UserTestSingleSelect()
{
    ProError status;
    int n_sel = 0;
    ProSelection *sel;
    // 下面函数的第一个参数是需要选择的对象，可以组合多个对象。具体请参照 UserGuide
    // 中的描述，第二个参数表示只选取一个对象。
    status = ProSelect ("surface", 1, NULL, NULL, NULL, NULL, &sel, &n_sel);
    if (status != PRO_TK_NO_ERROR || n_sel < 0)
        return (PRO_TK_USER_ABORT);

    ProModelitem feature;
    status = ProSelectionModelitemGet(*sel, &feature);
    if(status == PRO_TK_NO_ERROR)
    {
        CString item_id;
        item_id.Format("sel item id is : %d", feature.id);
        AfxMessageBox(item_id); // 调用 Windows 的警告对话框以快速查看信息（在实际
        // 应用中应该使用 LOG 文件以记录程序运行时状态）
    }
    return (PRO_TK_NO_ERROR);
}
```

}

为了方便说明选择功能，本函数没有进行过多的错误检查，要运行本函数，请打开或新建一个模型，运行后系统会弹出一个选择器，同时模型中只有“面”是可选状态，选择某一个面后，系统将提示出该面的 ID。如图 3-2 所示：



图 3-2

## 多选

与单选一样，只是函数 ProSelect 中的第二个参数由 ‘1’ 变成了 ‘-1’。程序如下：

```
ProError UserTestMultiSelect()
{
    ProError status;
    int n_sel = 0;
    ProSelection *sel;

    status = ProSelect ("surface", -1, NULL, NULL, NULL, NULL, &sel, &n_sel);
    if (status != PRO_TK_NO_ERROR || n_sel < 0)
        return (PRO_TK_USER_ABORT);

    ProModelitem feature;
    CString item_id;

    for(int i=0; i<n_sel; i++)
    {
        status = ProSelectionModelitemGet(sel[i], &feature);
        if(status == PRO_TK_NO_ERROR)
        {
            item_id.Format("sel item id is : %d", feature.id);
            AfxMessageBox(item_id);
        }
    }
}
```

```

    }
    return (PRO_TK_NO_ERROR);
}

```

运行该函数进行选择的时候，用户可以按住 **ctrl** 键不放，然后用鼠标左键进行选择。

## 框选

框选也是多选的一种，框选时，用户可以通过鼠标在图形窗口中画出的矩形框来选择，凡是包含在矩形框中的相关对象都将被选中。

代码如下：

```

ProError UserTestBoxSelect()
{
    ProError status ;
    ProSelection* p_sel_list;
    int p_sel_num;

    ProSelectionEnvOption    env_options[ 3 ] = {{PRO_SELECT_DONE_REQUIRED, 0},
                                                {PRO_SELECT_BY_MENU_ALLOWED, 1}
                                                , {PRO_SELECT_BY_BOX_ALLOWED, 1} };

    ProSelectionEnv sel_env;
    ProSelectionEnvAlloc( env_options, 3, &sel_env );

    ProSelect("surface", -1, NULL, NULL, sel_env, NULL, &p_sel_list, &p_sel_num);

    CString count;
    count.Format("totally choosed %d surfaces", p_sel_num);
    AfxMessageBox(count);
    return PRO_TK_NO_ERROR;
}

```

运行时，在图形窗口中按住鼠标左键不放，在窗口中画出一个矩形，松开后，在该矩形框中的所有相关元素（本例中为面）便会被选中。当然，也可使用 **CTRL+鼠标** 选择。

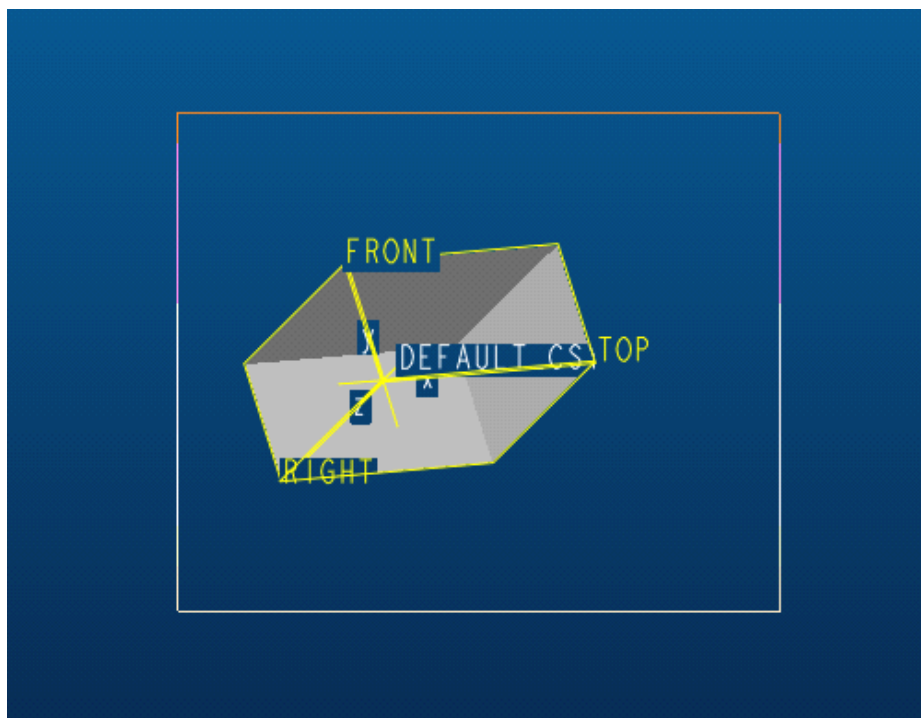


图 3-3

## 自动选择

接下来介绍自动选择，比如说模型中存在一个名称为“ABC”的元素，那么我们可以直接通过程序来自动选择这个元素以便后续操作。又或者，要将模型中所有的边都做倒角，那么也可以通过程序来自动选择所有的边。对于自动选择来讲，其在自动装配中的用途非常大，因此是一个必须熟练掌握的内容，下面笔者就以 PART 模式下自动选择一个对象作为本段的说明。（在 ASM 模式下的自动选择涉及到 compath 的相关内容，笔者将在后续章节详细描述）

首先，我们要在模型中建立一个有固定名称的对象，可以是一个几何特性（点，面，线），也可以是一个特征。为了便于说明，此处我们就以一个基准轴来说明此种选择的应用。

请看下面的程序：

```
#include <ProMdl.h>    //新添的头文件（所有头文件都可以通过函数搜索器搜索获得）
void UserTestAutoSelect()
{
    ProMdl current_mdl;
    ProMdlCurrentGet(&current_mdl);

    ProType type = PRO_AXIS;
```

```

ProName name;
ProStringToWstring(name, "TEST_AXIS");

ProModelitem sel_item;
ProModelitemByNameInit(current_mdl, type, name, &sel_item);

ProSelection output_selection;
ProSelectionAlloc(NULL, &sel_item, &output_selection);

ProModelitem check_item;
ProSelectionModelitemGet(output_selection, &check_item);

CString result;
result.Format("the selected item's id is : %d", check_item.id);
AfxMessageBox(result);
}

```

在本例中读者也可以使用函数：ProSelectionHighlight 来高亮被选中的元素。

运行本程序的时候读者需要在当前 PART 模型中建立一个名称为：“TEST\_AXIS”的基准轴，运行后，该轴将被程序自动选中，然后它的 ID 将被显示出来。要查看 PROE 系统本身分配给该轴的 ID 可参见下图介绍的流程：

首先点击模型树工具栏中的“Settings”按钮，然后选择“Tree Columns”，如图 3-4：

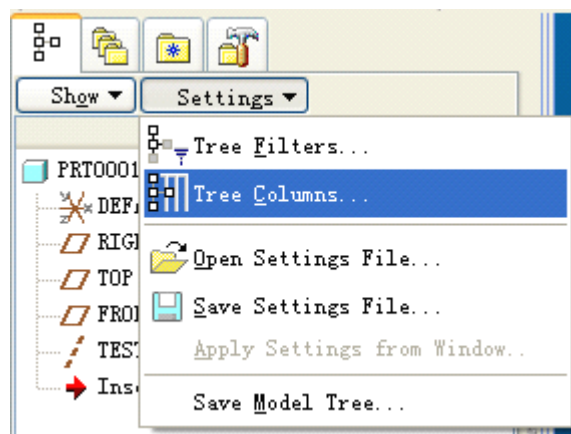


图 3-4

这时在弹出的对话框中双击“Feat ID”便可。如图 3-5

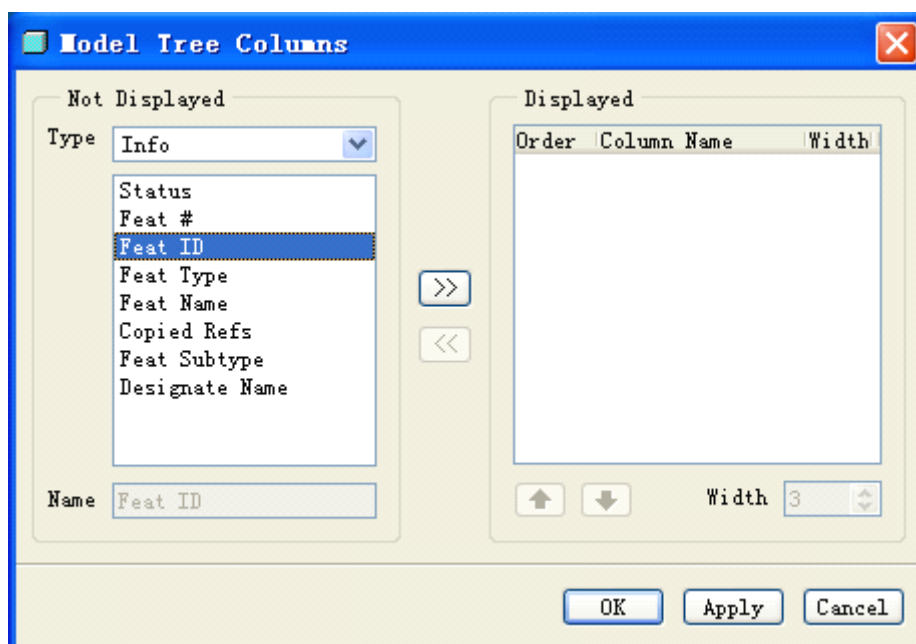


图 3-5

点击 OK 后，模型树变成图 3-6 所示状态：

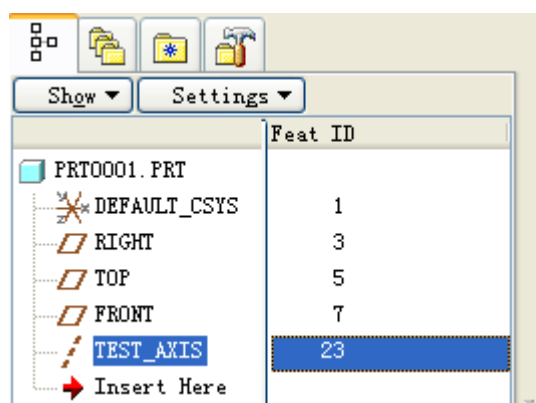


图 3-6

读者可将系统分配给对象的 ID 与 自己从程序中获取的 ID 对比以判断结果的正确性。

## 载入对象

这里给大家补充介绍下 Mdl 与 Modelitem 的区别。对于这两个对象，大家在前面的学习中应该已经多少用过几次了，那么他们到底有什么区别呢？笔者用通俗点的方法来解释下。Mdl 可以看做是 Model 的简称，也就是说，它是一个模型。而 Modelitem 可以看做是 Model 下的 item，也就是说，（理论上）它应该是 Model 的一个元素。因此，如果我们想要获得 Modelitem，则必须获取其父项：Model。比如前面我们在一个模型中去自动选择了一个基准轴，在这

个过程中,我们首先获取了当前的模型(Model),然后才是其中的轴(Modelitem)。

根据这个逻辑,我们来到组立的状态下。在组立模型中,当前的模型是大组立,此时,这个模型便是我们的 Mdl,在这个组立中的那些 Part 便是我们的 Modelitem。刚才说过,Part 在单独打开时也是一个 Mdl,为了便于转换这两种状态,Proe 提供了函数来转换这两个对象,这个函数便是: ProMdlToModelitem

。

下面我们回到正题上来。载入模型指的是将模型读取到内存中,(可以不显示), 从当前路径的磁盘中载入模型使用的函数为: ProMdlRetrieve, 从其它磁盘载入模型所使用的函数为: ProMdlLoad。下面我们用 ProMdlRetrieve 来举例。请参考下面的代码,下面代码首先从当前内存中寻找要显示的模型,如果没有找到则从当前工作目录载入该模型并显示在当前窗口。代码如下:

```
#include <ProWindows.h>    //新添的头文件
void UserTestLoadMdl()
{
    ProError status;
    ProMdlType model_type = PRO_MDL_PART;
    ProMdl *session_mdls, find_mdl;
    int mdls_counts;
    ProName target_name;
    ProStringToWstring(target_name, "TESTLOADMDL");
    ProBoolean was_found = PRO_B_FALSE;

    status = ProSessionMdlList(model_type, &session_mdls, &mdls_counts);
    if(status == PRO_TK_NO_ERROR)
    {
        ProName mdl_name;
        for(int i=0; i<mdls_counts; i++)
        {
            status = ProMdlNameGet(session_mdls[i], mdl_name);
            int compare_result = 1;
            ProWstringCompare(target_name, mdl_name, PRO_VALUE_UNUSED,
&compare_result);
            if(compare_result == 0)
            {
                ProMdlDisplay(session_mdls[i]);
                was_found = PRO_B_TRUE;
                break;
            }
        }
    }

    ProArrayFree((ProArray*)&session_mdls);
}
```



```

    }

    // if not found, try to retrieve it from disk
    if(was_found == PRO_B_FALSE)
    {
        status = ProMdlRetrieve(target_name, model_type, &find_mdl);
        if(status == PRO_TK_NO_ERROR)
        {
            ProMdlDisplay(find_mdl);
        }
    }

    int current_window;
    status = ProWindowCurrentGet(&current_window);
    if(status == PRO_TK_NO_ERROR)
    {
        ProWindowActivate(current_window);
    }
}

```

在函数搜索器中搜索 ProMdlRetrieve，点击参数 ProMdlType 可以看到 PROE 提供给我们的可载入的模型的类别有：

```

typedef enum
{
    PRO_MDL_UNUSED      = PRO_TYPE_UNUSED,
    PRO_MDL_ASSEMBLY    = PRO_ASSEMBLY,
    PRO_MDL_PART        = PRO_PART,
    PRO_MDL_DRAWING     = PRO_DRAWING,
    PRO_MDL_3DSECTION   = PRO_3DSECTION,
    PRO_MDL_2DSECTION   = PRO_2DSECTION,
    PRO_MDL_LAYOUT      = PRO_LAYOUT,
    PRO_MDL_DWGFORM     = PRO_DWGFORM,
    PRO_MDL_MFG         = PRO_MFG,
    PRO_MDL_REPORT      = PRO_REPORT,
    PRO_MDL_MARKUP      = PRO_MARKUP,
    PRO_MDL_DIAGRAM     = PRO_DIAGRAM
} ProMdlType;

```

因此要载入不同形态的模型，只需要更改枚举参数 ProMdlType 的值便可。  
（当我们遇到不熟悉的函数时，一定要善于利用函数搜索器，只有熟练掌握了这个搜索器，才称得上是“入门”）

## 遍历对象中的元素

前面已经介绍了如何在模型中去选择元素，这个过程可以手动，也可以自动。但是不管是手动也好，自动也好，我们选择对象都是有限的，同时也是一个一个去选的，如果我们要实现批量功能时，这个选择就有点吃力了。那么有没有一种方法能实现批量获得模型的对象呢？答案是有的。接下来介绍的对象的遍历便可以轻松实现这个功能。

PROE 提供了一系列的遍历函数以供我们遍历不同的对象。比如我们要遍历某个模型中所有的特征，那么该函数会自动从第一个特征遍历到最后一个特征，同时，总是把当前所遍历到的特征交给用户处理。遍历元素时，可以指定哪些元素需要过滤，哪些元素需要额外的处理。下面我们用一个例子来说明遍历函数的工作原理。

假设我们要遍历一个模型中所有的特征，遍历的时候，我们把每个特征的 ID 值记录下来，最后将这个结果输出。

代码如下：

```
#include <ProArray.h>    // 新增头文件
#include <ProSolid.h>    // 新增头文件

ProError UserFeatVisitFilt(ProFeature *p_feature, ProAppData id_array);
ProError UserFeatVisitAct(ProFeature *p_feature, ProError status, ProAppData id_array);

void UserTestVisitAllFeatruesInPart()
{
    ProError status;
    ProArray id_array = NULL;

    status = ProArrayAlloc (0, sizeof(int), 1, (ProArray*)&id_array);
    if(status != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("ProArrayAlloc error");
        return ;
    }

    ProMdl mdl;
    ProMdlCurrentGet(&mdl);
    ProSolidFeatVisit((ProSolid)mdl,
                      (ProFeatureVisitAction)UserFeatVisitAct,
                      (ProFeatureFilterAction)UserFeatVisitFilt,
                      (ProAppData)&id_array);
}
```

```

int array_size;
ProArraySizeGet((ProArray)id_array, &array_size);
if(array_size == 0)
{
    AfxMessageBox("No id found");
    return;
}

char result[255] = {0};
strcpy(result, "the ids are : ");

for(int i=0; i<array_size; i++)
{
    char temp[20] = {0};
    sprintf(temp, "%d | ", ((int*)id_array)[i]);
    strcat(result, temp);
}

AfxMessageBox(result);
ProArrayFree ((ProArray*)&id_array);
}

ProError UserFeatVisitFilt(ProFeature *p_feature, ProAppData data)
{
    if(p_feature->id < 100)
        return PRO_TK_CONTINUE; // if id < 100, skip this feature.

    else
        return PRO_TK_NO_ERROR; // if id >= 100, pass it to 'UserFeatVisitAct'
}

ProError UserFeatVisitAct(ProFeature *p_feature, ProError status, ProAppData data)
{
    int value = p_feature->id;

    status = ProArrayObjectAdd((ProArray*)data, PRO_VALUE_UNUSED, 1, &value);

    return PRO_TK_NO_ERROR;
}

```

与此所对于的程序运行结果如图 3-7 所示：

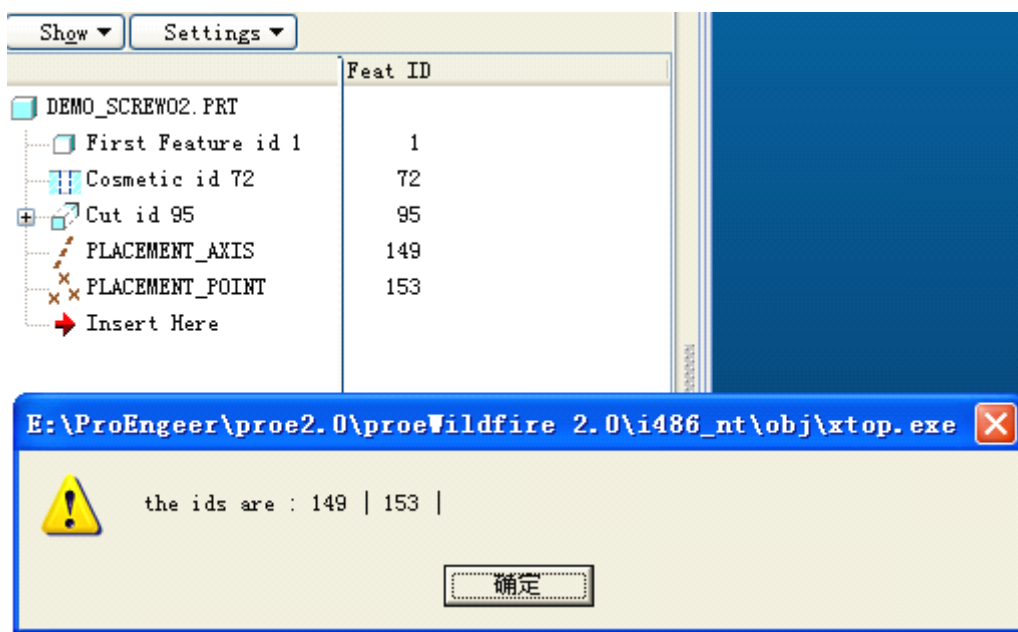


图 3-7

下面对该函数的工作过程进行简要的说明。

为了记录特征中的 ID, 程序一开始便初始化了一个数组 `id_array`, 然后程序调用了系统中的遍历函数: `ProSolidFeatVisit`, 该函数的执行过程是这样的: 函数按照模型树的顺序从上到下依次访问模型中的特征 (包含内部特征), 每当访问到一个特征, 函数首先调用过滤函数: `UserFeatVisitFilt` 对该特征进行过滤, 如果这个函数返回 `PRO_TK_CONTINUE` 则表示需要跳过此特征进而继续访问下一个特征, 如果该函数返回 `PRO_TK_NO_ERROR` 则过滤函数将该特征抛给动作函数: `UserFeatVisitAct`, 于是动作函数便在函数体内执行某种操作, 本例中, 我们执行的操作为记录特征的 ID 号码。如此循环, 一直到模型树中所有的特征都被访问完毕, 此时, 遍历函数退出。

在本例中有两个难点, 一个是 `ProArray` 的使用, 另外一个就是 传递给动作函数的 `ProAppData` 的使用。读者如果目前还不能够完全理解, 请不要心急, 因为笔者以后会对该部分做单独介绍。(这个遍历函数涉及到一个 C 语言知识: 指向函数的指针) 当然, 心急的读者也完全可以通过其他途径来自己理解这个问题。

## LOG 文件

无论是在调试程序, 还是程序发布之后, 在程序运行的过程中, 我们往往会记录程序的一些动作, 这个记录的文件通常称为 LOG 文件, 为了方便讲解, 笔者在演示程序的过程中大量使用了 `AfxMessageBox` 函数, 但是, 要知道, 这仅仅是笔者偷懒的一种方法而已, 更为正规的方法是编写 LOG 文件, 随时记录下

程序的运行状态。下面就给大家演示一段用 C++编写的简单的 LOG 文件的操作方法。程序代码如下：

```
#include <fstream>
void UserInitializeLog();
void UserInsertLog(char* value);

void UserTestInfoWindow()
{
    UserInitializeLog(); // 新建一个文件，如果已存在则清空。

    UserInsertLog("first line"); // 向文件中写入相关内容
    UserInsertLog("second line");
    UserInsertLog("third line");

    ProPath file_name;
    ProStringToWstring(file_name, "wangwei.log");
    ProInfoWindowDisplay(file_name, NULL, NULL); //调用 PROE 的 INFO 对话框
}

void UserInitializeLog()
{
    std::ofstream logfile("wangwei.log", std::ios::out);
    if(!logfile)
    {
        return;
    }
    logfile.close();
}

void UserInsertLog(char* value)
{
    std::ofstream logfile("wangwei.log", std::ios::app);
    if(!logfile)
    {
        return;
    }
    logfile << value << std::endl;
    logfile.close();
    return;
}
```

读者可能已经注意到，这其实是一个最简单的文件操作，里面甚至连错误检查都不全面，没错，笔者这里仅仅是给大家演示了一个大概框架而已，其他内容

读者可自行完成。(当然,若要偷懒,以上代码也是可以直接使用的),文件操作的方法有很多,除了 C++提供的该方法外,C 和 VC 里面也提供了相应的方法,其中 VC 中的 CFile 类则更为简单,这里就不一一叙述了。有兴趣的读者请查阅相关书籍。

参考代码如下:

```
void FileInitialize()
{
    CFile ErrorFile;
    ErrorFile.Open("error.txt", CFile::modeReadWrite|CFile::modeCreate)
}

void ErrorReport(CString errorstatus)
{
    int length = errorstatus.GetLength();
    LPCTSTR Content = errorstatus;
    ErrorFile.Write(Content, Length);
}

void FileClose()
{
    ErrorFile.Close();
}
```

## 制作并调用快捷键

ProE 中的快捷键也可以称之为“宏”,其原理就是将用户的操作记录下来,到用户激活的时候,在按照原样输出,这样就能起到节约时间,提高工作效率的作用。下面我们来一起了解该“宏”的录制过程。

这里就用我们目前最常用的一个命令来用宏实现,前面介绍了如何将我们的程序加载到 PROE 中,即:点击“工具”下的 **Auxiliary Applications**,结果是每次加载和卸载时都需要点击这个命令,而且有时候还容易点错。下面我们就用宏的方法来解决这个问题。

点击工具下的宏指令,如图 3-8:

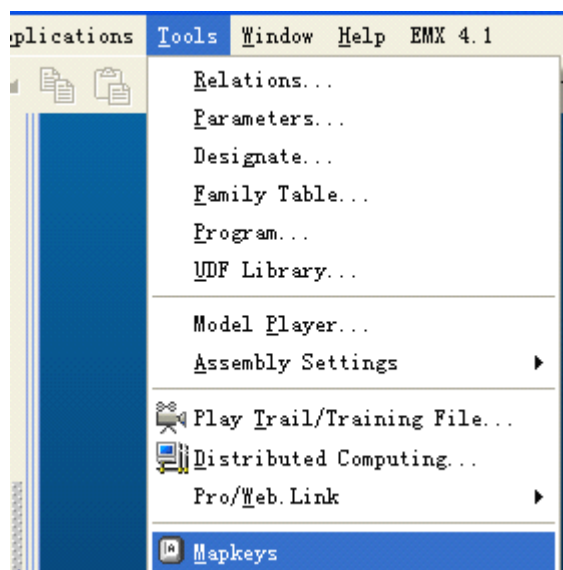


图 3-8

系统弹出录制宏的对话框：图 3-9

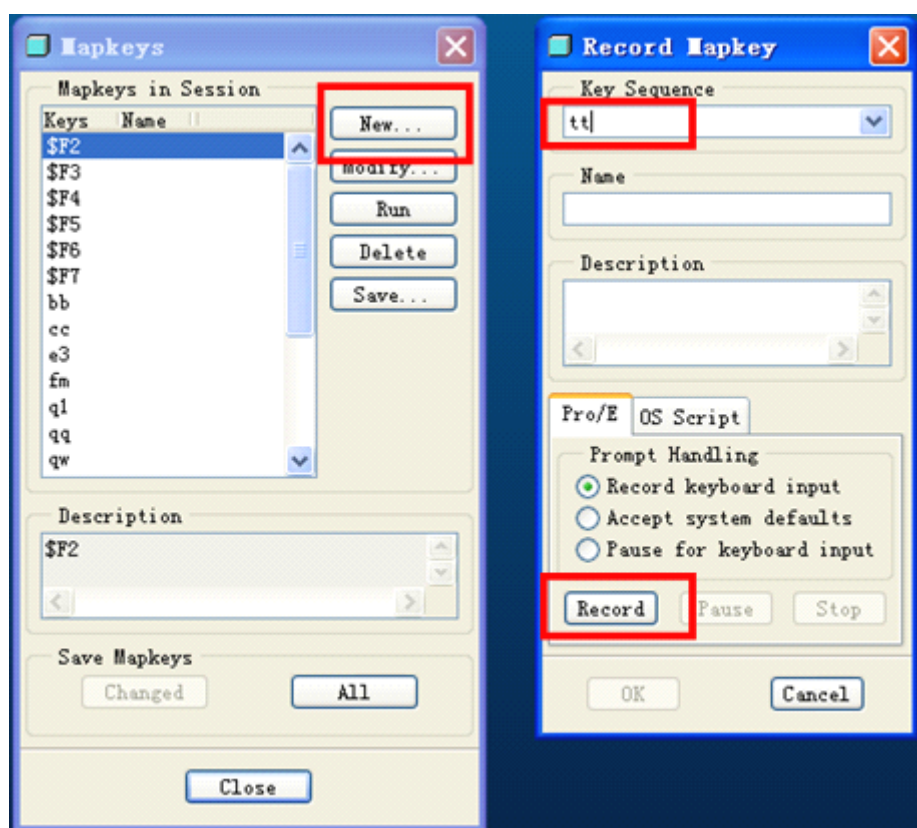


图 3-9

点击“新建”后出现“录制宏”对话框（图 3-9 右），首先在最上面的“Key Sequence”栏位输入快捷键的键顺序，本例中为 tt(不区分大小写)，按照图示选择“记录键盘输入”选项，点击“记录”按钮，此时，系统便会把你后续的操作录制下来，进而称为宏。这里，我们点击工具下的 **Auxiliary Applications**，此

时系统会弹出加载程序的加载框，现在我们的宏已经录完，点击录制宏对话框中的“停止”按钮结束宏的录制然后点击确定保存。如图：3-10

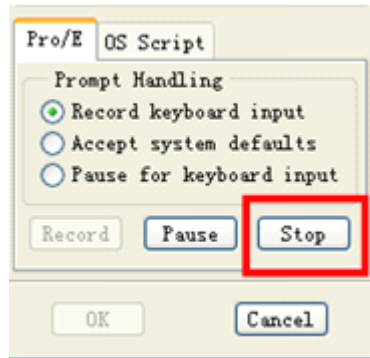


图 3-10

现在，双击键盘上的 TT，屏幕在快速闪动之后，便调出了我们所要的对话框。这个过程就好像我们自己通过菜单选择了命令一样。这便是快捷键的一般操作。现在打开刚才保存快捷键时的那个文件，本机中为：config.pro，在该文件的最后，我们能发现如下新添内容：

```
mapkey tt ~ Select `main_dlg_cur` `MenuBar1`1 `Utilities`;\  
mapkey(continued) ~ Close `main_dlg_cur` `MenuBar1`;\  
mapkey(continued) ~ Activate `main_dlg_cur` `Utilities.psh_util_aux`;
```

这便是系统为我们录制下的操作过程，虽然不怎么形象，但也能大概看得出来是先打开了一个菜单，激活了一个命令，然后又关闭了主菜单，最后激活了另外一个命令。这和我们自己手动的操作完全一致。本例中只是介绍了快捷键的一般录制方法，其他更详细的方法可参考其他书籍或登陆笔者论坛进行查询。

下面进入我们的正题，如何用程序来调用这个快捷键。

用程序来调用快捷键的方法很简单，主要用到一个函数：ProMacroLoad 我们用一段简单的代码来演示该函数的应用。

```
#include <ProMenu.h>  
  
void UserTestMacro()  
{  
    ProMacro wmacro;  
    char *macro = "~ Select `main_dlg_cur` `MenuBar1`1 `Utilities`;\  
                  ~ Close `main_dlg_cur` `MenuBar1`;\  
                  ~ Activate `main_dlg_cur` `Utilities.psh_util_aux`;";  
  
    ProStringToWstring(wmacro, macro);  
    ProMacroLoad(wmacro);  
}
```



```
}
```

相关的范例在帮助文件中也能找到。需要注意的是，函数 **ProMacroLoad** 是宏操作，所以只能在程序的末尾执行。举个例子，请看以下代码：

```
void UserTestMacroSequence()  
{  
    UserTestMacro();  
  
    int value;  
    for(int i=0; i<10; i++)  
        value += i;  
    CString temp;  
    temp.Format("the result is %d", value);  
    AfxMessageBox(temp);  
}
```

可能大家的第一印象是，程序会先执行 **UserTestMacro()** 函数，然后再执行后面的内容。可实际情况是，程序在运行 **UserTestMacro()** 函数的时候，检查到了 **ProMacroLoad** 的存在，因此，暂时跳过了这个函数，一直到其他所有内容执行完毕之后，在程序的末尾才调用 **ProMacroLoad**。所以运行以上的函数，会发现一个奇怪的现象：**AfxMessageBox** 执行完毕后，快捷键宏才生效运行。另外，读者可能发现，在调用快捷键宏的时候，屏幕总会有闪动，有没有方法能减少或者避免呢？答案是有的。当然，需要说明的是，并不是所有的快捷键都能“不闪”，这个要根据宏的大小来判断。下面介绍下让宏减少闪屏的方法。其实也非常简单，我们在录制宏的时候，生成了一些记录宏的文本描述，如前所述。这些描述中，有些是多余的。读者可以尝试将自己录制的宏里面的某些描述文字删掉（当然在保证宏能运行的情况下）。这时读者可能就会发现，屏幕减少了闪动或者是不闪动了。

宏的用途非常广泛，有时候某些开发用程序来一步一步的做比较费时，同时也是比较困难的，这时如果能够合理有效的与宏搭配的话，可以起到事半功倍的作用。这一点在本书最后的专题中可窥见一二。

## 第四节：参数，尺寸，关系式

ProE 的最主要特点就是实现了参数化建模，在实际应用当中，参数是非常重要的一个环节，同时，尺寸与关系式也相对重要。对于关系式，我们一般很少直接用程序来控制它，通常情况下，我们会利用 PROE 里面的工具将关系式写在模型中，以此建立参数与尺寸的连接。而程序要做的事情只是操作参数而已。

### 参数操作

首先介绍 PROE 系统的参数界面，在系统中，有很多对象都可以建立参数，打开一个模型后，点击：工具 → 参数，系统弹出参数对话框：

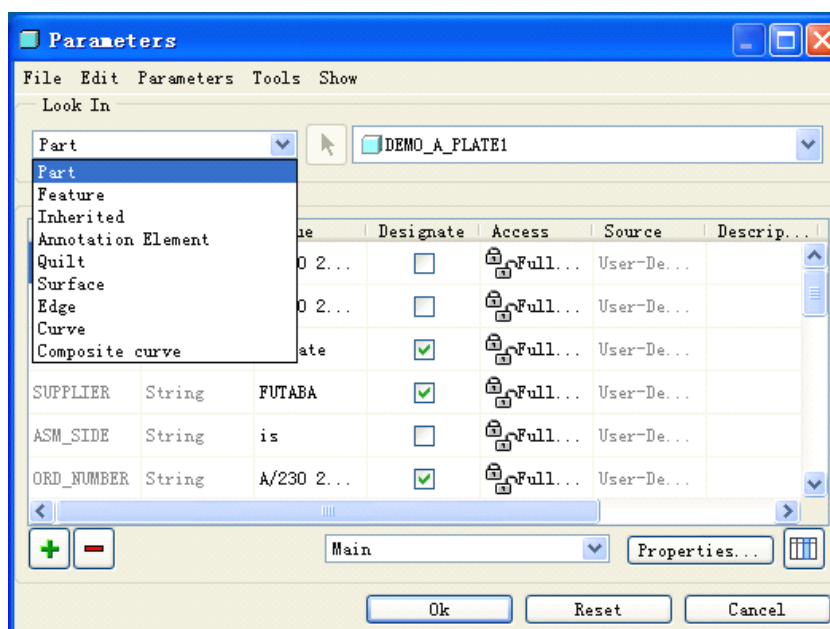


图 4-1

在如图所示的下拉列表中选择要查询的对象，然后在模型树或者图形窗口中选择后，所选对象的参数便会被罗列出来，刚打开该对话框时，系统默认为当前模型的参数列表。在这个列表中，我们可以通过左下角的“加号”来添加参数，也可以使用“减号”来删除参数。建立好参数后，只要我们对模型进行了保存，那么这些参数也将作为数据库存储在模型中。所以，参数列表实际上可以存放很多与当前模型相关的信息。这对于自动化建模来讲是非常重要的。

点击“加号”后，系统会在参数列表中新建一行：

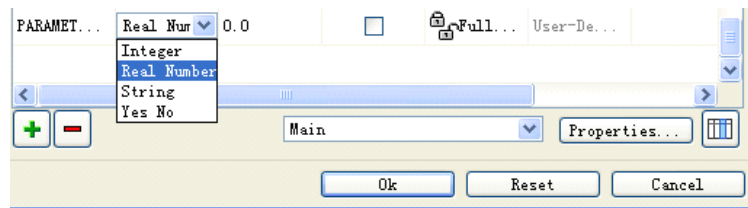


图 4-2

从左到右依次为：参数名称，参数类别（共四类，分别是整型，浮点型，字符型以及布尔型），参数值，设计标签（通常用于出 BOM 表）等。设计好的参数通常可应用在程序中，或直接与关系式挂钩（后面再讲）。下面，我们就来介绍如何使用程序来完成以上我们所做的工作。

用程序来操作参数其实很简单。PROE 给我们提供了相应的函数来控制参数的创建，查询，删除等动作。在下面的例子中，我们分别为当前模型创建了四个不同类型的参数。

代码如下：

```
#include <ProParameter.h>
```

```
void UserTestCreateParams()
```

```
{
```

```
    ProMdl cur_mdl;
```

```
    ProModelitem cur_modelitem;
```

```
    ProParameter param;
```

```
    ProName name;
```

```
    ProParamvalue proval;
```

```
    ProMdlCurrentGet(&cur_mdl);
```

```
    ProMdlToModelitem(cur_mdl, &cur_modelitem);
```

```
    //创建小数
```

```
    ProStringToWstring(name, "doubleparam");
```

```
    proval.type = PRO_PARAM_DOUBLE;
```

```
    proval.value.d_val = 12.5;
```

```
    ProParameterCreate(&cur_modelitem, name, &proval, &param);
```

```
    //创建字符串
```

```
    ProStringToWstring(name, "stringparam");
```

```
    proval.type = PRO_PARAM_STRING;
```

```
    ProStringToWstring(proval.value.s_val, "this is a string");
```

```
    ProParameterCreate(&cur_modelitem, name, &proval, &param);
```

```
    //创建整数
```

```
    ProStringToWstring(name, "intparam");
```

```

proval.type = PRO_PARAM_INTEGER;
proval.value.i_val = 10;
ProParameterCreate(&cur_modelitem, name, &proval, &param);

// 创建BOOL类型
ProStringToWstring(name, "boolparam");
proval.type = PRO_PARAM_BOOLEAN;
proval.value.i_val = 1;
ProParameterCreate(&cur_modelitem, name, &proval, &param);
}

```

程序分别创建了 double, string, int 和 bool 四个类型的参数，运行结果如下：





|            |            |                  |                          |                                                                                             |            |
|------------|------------|------------------|--------------------------|---------------------------------------------------------------------------------------------|------------|
| DOUBLEP... | Real Nu... | 12.50            | <input type="checkbox"/> |  Full... | User-De... |
| STRINGP... | String     | this is a string | <input type="checkbox"/> |  Full... | User-De... |
| INTPARAM   | Integer    | 10               | <input type="checkbox"/> |  Full... | User-De... |
| BOOLPARAM  | Yes No     | YES              | <input type="checkbox"/> |  Full... | User-De... |

图 4-3

与创建参数的过程一样，ProParameterValueGet 与 ProParameterValueSet 分别用于获取和修正参数的值。范例如下：

```

void UserTestGetParamValue()
{
    ProMdl cur_mdl;
    ProModelitem cur_modelitem;
    ProParameter param;
    ProName name;
    ProParamvalue proval;

    //获取一个小数数据
    ProMdlCurrentGet(&cur_mdl);
    ProMdlToModelitem(cur_mdl, &cur_modelitem);
    ProStringToWstring(name, "doubleparam");
    ProParameterInit(&cur_modelitem, name, &param);
    ProParameterValueGet(&param, &proval);
}

```

以上两个范例当中都是直接在模型中建立和读取参数，除此之外，我们还可以直接在特征中建立参数。函数 ProModelitemByNameInit 可以直接将模型中的特征转换为Modelitem对象，然后就可以直接利用上面两个范例中的相关方法来操作参数了，具体的代码可以在函数搜索器上得到，笔者此处不再详述，只给出伪代码。

```

void UserTestCreateParamsInFeature()
{
    ProModelitem target_feature;
    ProModelitemByNameInit(.....& target_feature)
    // 创建参数
    ProParameterCreate(&targer_feature, param_name, &proval, &param);
    // 读取参数
    ProParameterInit(&targer_feature, param_name, &param);
    ProParameterValueGet(&param, &proval);
}

```

## 尺寸操作

在参数化建模的过程中，尺寸往往是与参数不分家的，因此，这两个的搭配使用显的尤为重要，但是，尺寸中也存在一些容易混淆的地方，因此笔者在此仍对纯粹的尺寸操作做一些简单的说明。

要用程序准确获取尺寸对象，首先，我们需要规范尺寸的命名。命名方法如下： 首先选择尺寸中的属性，如图 4-4

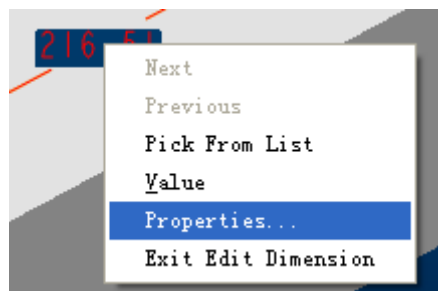


图 4-4

在随后系统弹出的尺寸属性对话框中，按照图 4-5 的方法修改尺寸名称（最好使用大写）

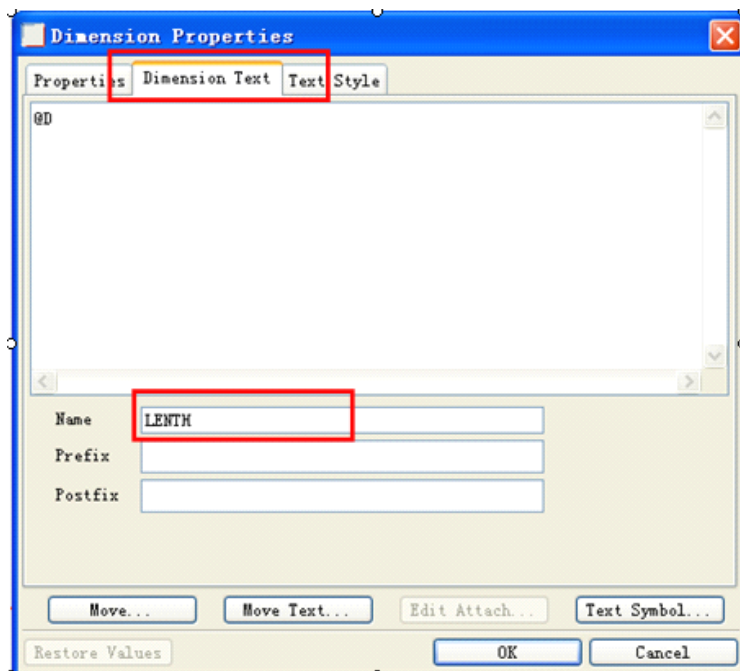


图 4-5

修改完成后，点击确定。这样准备工作就做完了。为了检查尺寸是否真的已经修改，我们可以使用 PROE 的尺寸查看器来查看尺寸名称：双击某个特征，系统将该特征所有的尺寸都显示了出来，此时只有尺寸数值，没有尺寸名称。点击“信息”菜单下的“尺寸转换”，这些尺寸便切换到其名称的状态了，如图：

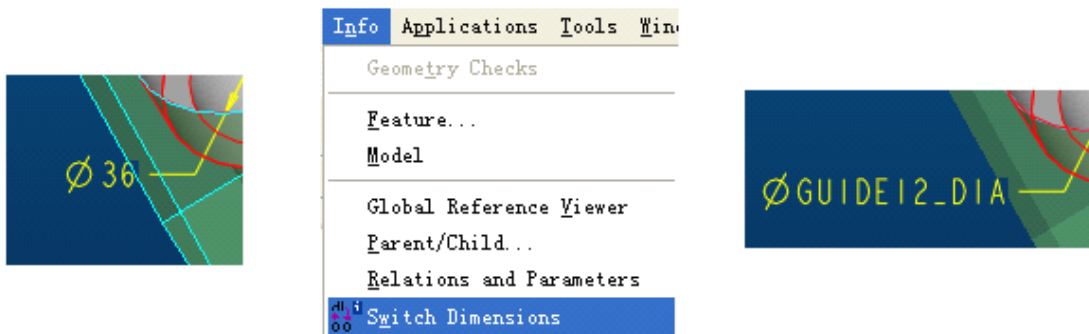


图 4-6

下面就以图 4-5 的设置（尺寸名称为 LENTH）来访问我们的尺寸。

代码如下：

```
#include <ProSolid.h> //新添的头文件

ProError DimAction(ProDimension *dimension, ProError status, ProAppData data);

void UserTestDimOperation()
{
```

```

ProMdl mdl;
ProMdlCurrentGet(&mdl);

// 依次访问每一个尺寸，并调用“DimAction”来处理这些尺寸
ProSolidDimensionVisit((ProSolid)mdl, PRO_B_FALSE, DimAction, NULL, NULL);

ProSolidRegenerate((ProSolid)mdl, PRO_REGEN_NO_FLAGS); //更新模型
}

ProError DimAction(ProDimension *dimension, ProError status, ProAppData data)
{
    ProName symbol, dim_name;
    ProStringToWstring(dim_name, "LENTH");

    ProDimensionSymbolGet(dimension, symbol); //获取传入的尺寸符号
    int result;
    ProWstringCompare(symbol, dim_name, PRO_VALUE_UNUSED, &result);

    double dim_value = 0;

    if(result == 0) //如果是我们的“LENTH”尺寸
    {
        ProDimensionValueGet(dimension, &dim_value);
        CString show;
        show.Format("current dim value is : %f", dim_value);
        AfxMessageBox(show); //输出尺寸值

        dim_value = 300;
        ProDimensionValueSet(dimension, dim_value); //设置一个新数值
    }

    return PRO_TK_NO_ERROR;
}

```

尺寸的访问与之前介绍的特征的访问是一个原理，在这个 **ProSolidDimensionVisit** 的函数中，系统会自动在当前的模型中去搜寻每个尺寸，因为模型中的尺寸特别的多，所以有必要给这个访问函数设置过滤条件，笔者在这里没有使用过滤函数是因为笔者的模型中只有一个特征，也算是偷懒吧。当程序发现与我们名称相同的尺寸时，便调用动作函数来处理这个尺寸。在这个动作函数里面，我们先获取了这个尺寸的值，然后又给这个尺寸赋予了一个 300 的新值（基本实现了尺寸的访问与读取）。

## 补充知识：函数指针

在第三节中我们讲过，访问函数涉及到一个 C 语言知识点：函数指针，为了避免大家再去查阅 C 语言的相关部分（很多初级书本上是找不到的），作为补充知识，笔者将该知识点在此讲解一下。

首先还是先看如下代码：

```
#include <iostream>
using namespace std;

//定义一个指向函数的指针，所指的函数必须包含三个整型参数并返回一个整型结果。
typedef int (*var_function) (int, int ,int);

//使用上面的指针作为Visit函数的参数
void MyVisit(var_function  variable)
{
    int a = 60;
    int b = 10;
    int c = 20;

    int result = variable(a, b, c);
    cout << "the result is " << result << endl;
}

// 普通函数
int FunctionA(int i, int j, int k)
{
    return (i + j + k);
}

// 普通函数
int FunctionB(int i, int j, int k)
{
    return (i - j - k);
}

//主函数
int main()
{
    MyVisit(FunctionA); //用函数A作为实参
    MyVisit(FunctionB); //用函数B作为实参
    return 0;
}
```



这是一个非常简单而且能够说明问题的范例，希望大家能够理解其中的原理。这也是我们在 PROE 的 VISIT 函数中所用到的。希望能够通过这个小范例加深大家对访问函数的理解。

在实际的工程应用中，纯粹的使用尺寸来驱动模型是很少的。大多都要用到参数驱动。下面将给大家演示如何将参数与尺寸通过关系式结合起来。

## 参数与关系式的搭配使用

前面已经大概的描述了如何操作参数和尺寸，这一小节中，主要向大家介绍如何用关系式来建立参数与尺寸之间的连接。

假设我们有这样一个简单的模型，该模型有长，宽，高 三个尺寸，如果我们要驱动这个模型，通过前面的方法，我们可以直接访问到尺寸，并通过修改其值来更改模型。但是大家可能会情不自禁的想到，当我的模型特征很多的时候，那相应的尺寸将是成千上万的，难道要一个一个的去过滤，去访问吗？当然不是。这时候，我们可以利用参数来做点事情。在这个模型中，假如长，宽，高有一定的关系，那么我们可以建立一个参数，同时利用关系式来建立这几个尺寸与参数的连接。

方法如下：

首先要建立一个参数，用以驱动尺寸。点击“工具”下的“参数”对话框，在该处假设我们建立一个 test 的参数，值为 10，图（4-7）

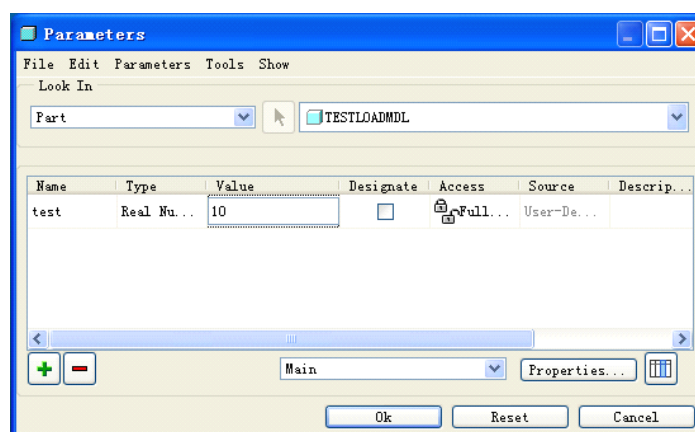


图 4-7

然后，点击“工具”菜单下的“关系”菜单，系统弹出关系对话框（图 4-8）

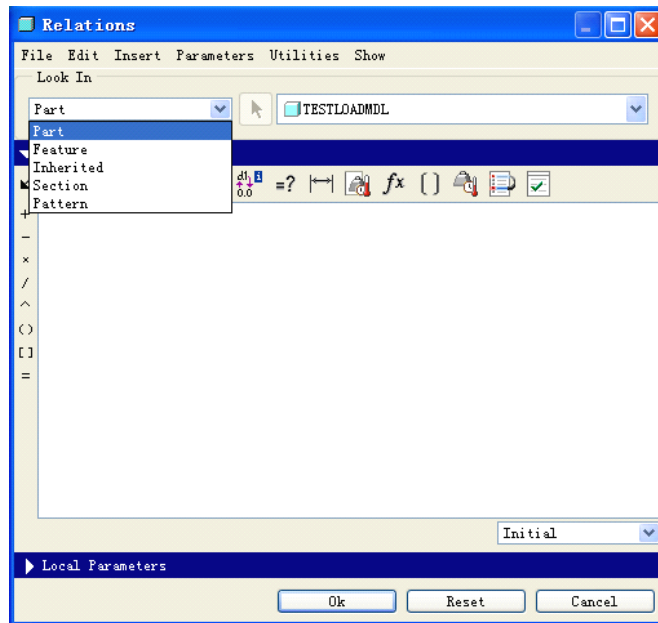


图 4-8

在这个对话框中，我们可以通过下拉菜单选择将关系建立在何种类型的对象中。PROE 一共有如下几种类型可供建立关系式。

**Part**—Access part relations both in the Part and Assembly modes.

**Assembly**—Access relations in an assembly.

**Feature**—Access relations specific to a feature in the Part or Assembly mode.

**Inherited**—Access relations both in the Part and Assembly modes.

**Section**—If a feature has a section, access section relations in Sketcher while in the Part or Assembly mode.

**Pattern**—Access relations specific to a pattern in the Part or Assembly mode.

**Skeleton**—Access relations for a skeleton model in Assembly mode.

**Component**—Access relations for an assembly component.

假设笔者选择 **Part** ，然后我们在空白区域填写关系式，如图 4-9

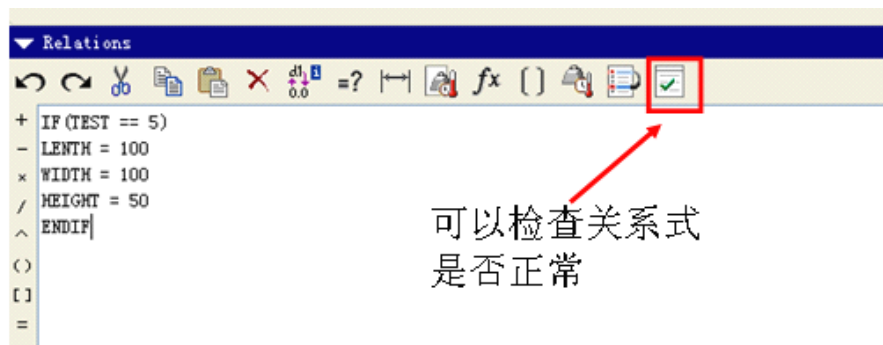


图 4-9

当然，读者可以根据参数的值来编写很多的 IF 语句，这样，我们便通过关系式使参数“test”与模型的长宽高建立了关系，这时我们只需要简单的修改参数的值，模型中的尺寸便会自动推算到相应的值，从而改变模型，这就是一个非常简单的基于参数化设计的范例。假设我们直接访问并修改尺寸，我们可能要在循环中分别对这三个尺寸进行处理，而通过修改参数，我们只需要修改这一个参数的值就行了（请参照前面的代码），当我们需要更改的尺寸很多的时候，这个方法是否更高效一点呢？

下面我们来了解一下关系式的一些基本知识。（关系式的写法与代码的写法类似，这里只是大概介绍其方法和过程以及一般性应用，并不详细描述关系式所有的写法，关系式的详细用法在 PROE 帮助文件中有详细介绍。）

为了便于查看，关系式一般是一行只写一句代码。如下：

```
/* Width is equal to 2*height    这是注解的方式
d1 = 2*d2                        这是有效关系式
```

在关系式中，其运算符与 C 语言的类似，同时，系统也提供了很多数学计算函数可供使用。这些就不多讲了，下面主要介绍字符串的应用。

The following operators and functions are supported for strings:

|                           |                                                                                                               |
|---------------------------|---------------------------------------------------------------------------------------------------------------|
| ==                        | Compares strings as equal.                                                                                    |
| !=, <>, ~=                | Compares strings as unequal.                                                                                  |
| +                         | Concatenates strings.                                                                                         |
| itos(int)                 | Converts integers to strings. Here, int can be a number or an expression. Nonintegers are rounded off.        |
| search(string, substring) | Searches for substrings. The resulting value is the position of the substring in the string (0 if not found). |

|                                                    |                             |
|----------------------------------------------------|-----------------------------|
| <code>extract(string,<br/>position, length)</code> | Extracts pieces of strings. |
|----------------------------------------------------|-----------------------------|

For example:

If `param = abcdef`, then:

- `flag = param == abcdef`—returns TRUE
- `flag = abcdef != ghi`—returns TRUE
- `new = param + ghi`—new is abcdefghi
- `new = itos(10 + 7)`—new is 17
- `new = param + itos(1.5)`—new is abcdef2
- `where = search(param, bcd)`—where is 2
- `where = search(param, bcd)`—where is 0
- `new = extract(param, 2, 3)`—new is bcd

另外，在关系式中，还可以将字符串作为参数使用。请看下列函数：

**string\_length()** —Returns the number of characters in a parameter.

**rel\_model\_name()** —Returns the current model name.

**rel\_model\_type()** —Returns the current model type.

**exists()** —Evaluates whether an item, such as a parameter or dimension, exists.

For example:

If the value for the string parameter **material** is defined as steel, **string\_length(material)** equals 5, because the word "steel" has five letters.

if you are currently working in a part called A, **rel\_model\_name()** is equal to A

If you are working in Assembly mode, **rel\_model\_type()** is equal to assembly

if **exists("d5:20")**—Checks if the model with runtime ID 20 has a dimension d5.

if **exists("par:fid\_25:cid\_12")**—Checks if the feature ID 25 in the component ID 12 has parameter par.

对于参数的操作和关系式的设定是参数化设计的重要内容，笔者这里只是带领大家认识了他们一些相关的应用，如果想灵活使用，还需对这一块进行大量的练习。（提示： 对于一些日常大量使用的函数， 如参数的操作等，读者可以考虑将其编写成自己的库函数，这样便于日后的编译与调用， 对于如何建立在 PROE 中可调用的函数库， 笔者在后续章节中会陆续讲到， 在此之前，读者可以自行尝试）

## 综合实例：齿轮的参数化设计的实现（基本原理）

本书的测试版 1.0 推出后，不少热心的网友提出希望添加齿轮的参数化设计的相关内容，因此，笔者在此处进行必要的补充。注意，由于参数化设计的核心都是一样的，其原理都是对模型参数的修改，所以本例重在演示原理，并不会将真实的齿轮的所有关系式都进行一一讲解。这点请读者们谅解。

下面展开我们的设计之旅。

首先假设齿轮模型已经建立完成（可从网络上下载，也可以自己动手制作）。为了能够进行参数化设计，所以必须建立关系式，齿轮的关系式由很多标准来控制，这里我不再详述。

下面直接讲解重点：模型中为程序保留的接口。以直齿为例，为了简便，我们通常会保留这样几个参数：齿轮模数(MOSHU)，齿轮齿数(CHISHU)，压力角(YALIJIAO)以及齿宽(CHIKUAN)等（要使模型根据这几个参数的变化而重生需要使用关系式来严格控制齿轮的相应尺寸，由于齿轮模型在网络上可以下载，因此这个不做讲解），有了接口之后，为了方便我们设计，当然一个漂亮的操作界面也是必不可少的。对于如何制作操作界面，在后续的章节中会有介绍，此处只提供参考界面。



图 4-10

根据图示不难看出，当我们点击“生成齿轮”按钮后，程序应该根据输入的数据生成一个左边所示的齿轮，并在当前窗口中显示出来。那么该如何做呢？其实很简单，由于与齿轮所对应的关系式已经完成，此处我们只需要简单的修改齿轮中的参数值便可。

请参考如下代码：

```
// 子函数： 设置参数值
ProError UserSetParamValue(ProModelitem owner
                           , char* name
                           , double value)
{
    ProError status;
    ProParameter param;
    ProName param_name;
    ProStringToWstring(param_name, name);
    status = ProParameterInit(&owner, param_name, &param);
    if(status != 0)
    {
        return status;
    }

    ProParamvalue proval;
    proval.type = PRO_PARAM_DOUBLE;
    proval.value.d_val = value;

    status = ProParameterValueSet(&param, &proval);
    if(status != 0)
    {
        return status;
    }

    return PRO_TK_NO_ERROR;
}

// 按钮函数
void Cylinder::OnOK()
{
    UpdateData(true);

    ProError status;
    ProFamilyName name;
    ProMdl p_handle;
    ProModelitem p_modelitem;

    ProStringToWstring(name, "CYLINDER");
    status = ProMdlRetrieve(name, PRO_MDL_PART, &p_handle);
    if(status != PRO_TK_NO_ERROR)
```

```

{
    return;
}

status = ProMdlToModelitem(p_handle, &p_modelitem);
UserSetParamValue(p_modelitem, "MOSHU", m_moshu);
UserSetParamValue(p_modelitem, "CHISHU", m_chishu);
UserSetParamValue(p_modelitem, "YALIJIAO", m_yalijiao);
UserSetParamValue(p_modelitem, "CHIKUAN", m_chikuan);

ProSolidRegenerate((ProSolid)p_handle, PRO_B_TRUE);    // 修改之后一定要记住重
生模型
status = ProMdlDisplay(p_handle);
int window;
ProMdlWindowGet(p_handle, &window);
ProWindowActivate(window);
}

```

由于是描述基本原理，所以本范例十分的清晰易懂，读者可在此基础上添加更多详细的功能，比如载入模型前的数据检查，模型重生后的状态，模型重生失败的补救措施，以及用户输入数据的限定等等。

到这里，参数，尺寸以及关系式的讲解全部结束，对于其中生疏的地方，读者需要加强练习（这是参数化设计中必须熟练掌握的一项基本技能），对于其中一些不懂的地方，比如关系式的编写等，读者可上网搜寻相关资源，也可使用 PROE 的帮助文件进行查询。

还是那句话，多练则熟，熟能生巧。

## 第五节： 装配概述

### 选择装配

装配在二次开发中也是一个非常重要的内容，通常我们会在元件库中使用到这种方法。其实还是一句话，如何让手动的东西变成自动化。这里假设我们已经建立了自己的元件库，当我们在设计模型的时候，如果要用到这些元件，那么常用的方法便是将这些元件手动装配到当前的组立中。在装配过程中，可能还要选择若干的参照，约束以便成功装配。这里，请大家思考一下，我们就拿最简单的装配（三个面约束）来说，在定义元件时，元件的约束已经是确定的了，因此在装配时我们应该只选择组立中的约束就可以成功安装。但是，实际情况是我们仍旧需要选择元件中的这些约束。为了提高效率，我们打算用程序来实现元件中约束的选取，而组立中约束的选取仍旧采用手动形式（本节最后将介绍全自动装配）

请看下面代码（稍微修改了帮助文件中的内容）

```
#include <ProAsmcomp.h> // 新添的头文件

ProError UserAssembleByDatums (ProAssembly asm_model,
                                ProSolid comp_model)
{
    ProError status;
    ProName comp_datums [3];
    ProMatrix identity_matrix = {{ 1.0, 0.0, 0.0, 0.0 },
                                  {0.0, 1.0, 0.0, 0.0},
                                  {0.0, 0.0, 1.0, 0.0},
                                  {0.0, 0.0, 0.0, 1.0}};

    ProAsmcomp asmcomp;
    ProAsmcompconstraint* constraints;
    ProAsmcompconstraint constraint;
    int i;
    ProBoolean interact_flag = PRO_B_FALSE;
    ProModelitem asm_datum, comp_datum;
    ProSelection asm_sel, comp_sel;
    ProAsmcomppath comp_path;
    ProIdTable c_id_table;
    c_id_table [0] = -1;

    //Set up the arrays of datum names

    ProStringToWstring (comp_datums [0], "COMP_D_FRONT");
    ProStringToWstring (comp_datums [1], "COMP_D_TOP");
```



```

ProStringToWstring (comp_datums [2], "COMP_D_RIGHT");

//Package the component initially

ProAsmcompAssemble (asm_model, comp_model, identity_matrix, &asmcomp);

//Prepare the constraints array

ProArrayAlloc (0, sizeof (ProAsmcompconstraint), 1,
               (ProArray*)&constraints);

for (i = 0; i < 3; i++)
{
    //Find the component datum

    status = ProModelitemByNameInit (comp_model, PRO_SURFACE,
                                     comp_datums [i], &comp_datum);

    if (status != PRO_TK_NO_ERROR)
    {
        interact_flag = PRO_B_TRUE;
        continue;
    }

    //Allocate the references

    ProSelectionAlloc (NULL, &comp_datum, &comp_sel);

    int n_sel = 0;
    ProSelection *sel;
    status = ProSelect ("surface", 1, NULL, NULL, NULL, NULL, &sel, &n_sel);
    if (status!= PRO_TK_NO_ERROR || n_sel < 0)
        return (PRO_TK_USER_ABORT);

    ProSelectionCopy(sel[0], &asm_sel);

    //Allocate and fill the constraint.

    ProAsmcompconstraintAlloc (&constraint);

    ProAsmcompconstraintTypeSet (constraint, PRO_ASM_ALIGN);

    ProAsmcompconstraintAsmreferenceSet (constraint, asm_sel,
                                         PRO_DATUM_SIDE_YELLOW);
}

```

```

ProAsmcompconstraintCompreferenceSet (constraint, comp_sel,
                                      PRO_DATUM_SIDE_YELLOW);

ProArrayObjectAdd ((ProArray*)&constraints, -1, 1, &constraint);

}

status = ProAsmcompConstraintsSet (NULL, &asmcomp, constraints);

ProSolidRegenerate ((ProSolid)asmcomp.owner, PRO_REGEN_CAN_FIX);

if (interact_flag)
{
    ProAsmcompConstrRedefUI (&asmcomp);
}

return (PRO_TK_NO_ERROR);
}

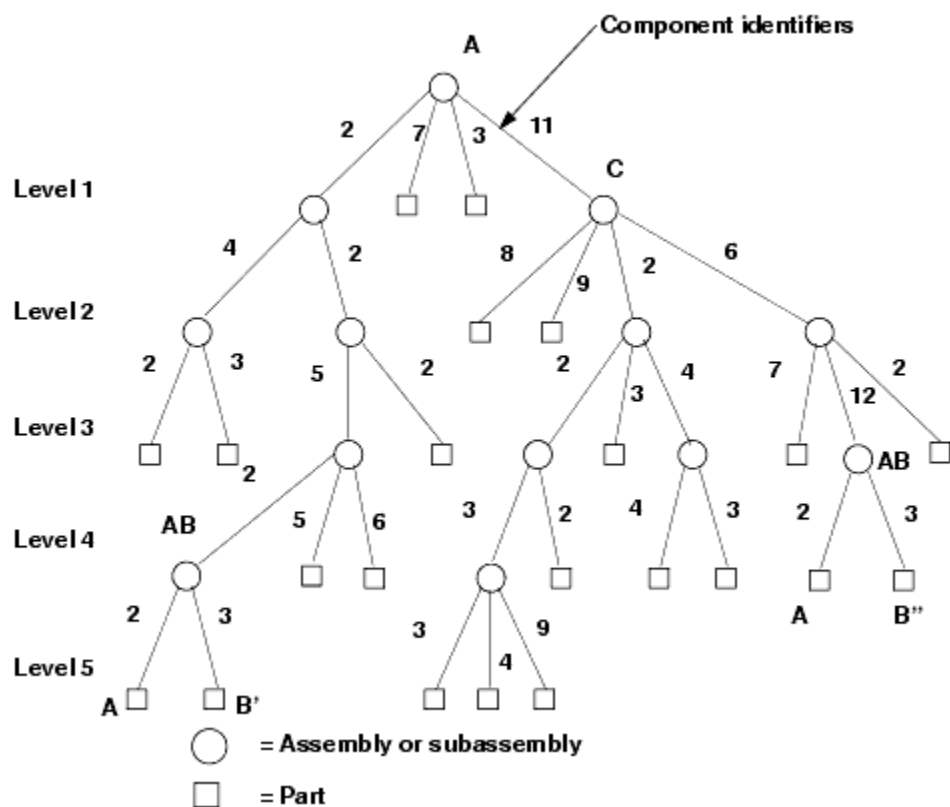
```

在上面的程序中，首先假设了元件装配的参照为三个面，他们在 `comp_datums` 中存放，因此，在将元件调入装配体中后，首先便设置了元件的约束，紧接着又用交互式选择的方式，提示用户进行选择，以确定元件准确的装配位置。这种半自动的方法比较灵活，通过用户选择，元件可以装配在组立中的任何地方。在本范例中，笔者仍旧没有添加错误处理程序，笔者请自行添加，同时，在用户进行交互选择的时候，需要在信息栏位提供提示信息，这一点请读者参照前面的章节进行补充设计。另外，本例可能会因为读者不同的建模方式而不能运行，遇到这种情况，请将 `ProAsmcompconstraintTypeSet (constraint, PRO_ASM_ALIGN)` 修改为：`ProAsmcompconstraintTypeSet (constraint, PRO_ASM_MATE)` 这种半自动的装配方式在 EMX 的元件库中有大量的使用，读者也可以将这种思维应用在任何元件库系统中。（在后续章节的专题中，笔者会讲解元件库的一般建立过程）

## 深入理解 ProAsmcomppath

装配档是一个非常重要的部份，而 `ProAsmcomppath` 又是装配档的一个非常重要又难以理解的部份。（笔者在初学时曾经非常头痛这个东西）如果要在装配档中实现自动化，就必须非常熟悉 `ProAsmcomppath` 的原理。下面我们一起来探讨这个看似难懂的问题。

我们可以这样理解，一个组立中存在很多元件，这些元件中可能还有很多同名元件，那么，我们要怎么样来标识这些元件呢？首先我们可以形象的想到路径。这是对的，对于装配档来讲，就算两个元件一模一样，但是其在该档下的路径是不一样的，这个路径由每个元件独立的 ID 来对应，组合后便成为了 `ProAsmcomppath` 的核心：`comp_id_table`，请看下图：（来自帮助文件）



Component B'

```
table_num = 5
comp_id_tab[0] = 2
comp_id_tab[1] = 2
comp_id_tab[2] = 5
comp_id_tab[3] = 2
comp_id_tab[4] = 3
```

Component B''

```
table_num = 4
comp_id_tab[0] = 11
comp_id_tab[1] = 6
comp_id_tab[2] = 12
comp_id_tab[3] = 3
```

图 5-1

如图所示，如果我们要在一个装配体中获得元件 B 的句柄，那么我们必须从装配体的顶部开始搜寻，按照其标识的顺序，一层一层的向下寻找。从图中还可以看出，系统默认的层是 5 层，所以，后续我们在建立自己的模型时，最好不要超过这个默认的层数。

下面我们用一张更为具体的图片来说明这个 PATH 的组成。

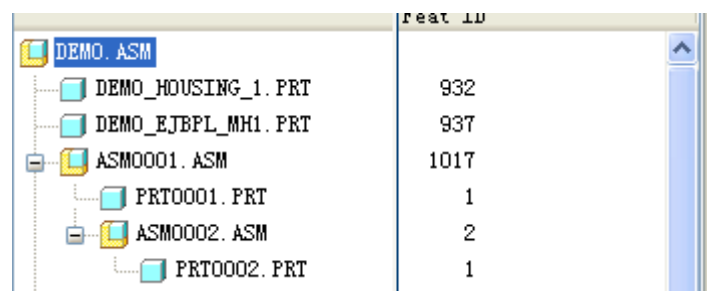


图 5-2

如图，假如现在我们要获取名称为 PRT0002.PRT 的句柄，那么应该怎么办呢？首先我们需要在总装配中找到 ASM0001，然后在它下面找到 ASM0002，最后才能找到我们的目标元件。对于程序来讲，由于所有元件在装配体中的 ID 标识都是唯一的（同一模型装配两次，其标识也是不一样的），所以，可以通过 ID 来识别。而 ProAsmcomppath 的核心便是这一组由 ID 所构成的路径表。对于上图的 PRT0002.PRT 来讲，其 ID 表为：1017, 2, 1, -1；其中，最后的-1 是一个系统自动补齐的数，表示该表已经结束。（通常情况下，这个-1 是可以省略的，但当装配档为一层的时候，该值不可省略）

下面的函数向大家演示了如何获取一个对象的 comppath

```
void UserTestGetComppath()
{
    ProError status;
    int n_sel = 0;
    ProSelection *sel;
    status = ProSelect ("part", 1, NULL, NULL, NULL, NULL, &sel, &n_sel);
    if (status!= PRO_TK_NO_ERROR || n_sel < 0)
        return ;

    ProAsmcomppath p_cmp_path;
    ProSelectionAsmcomppathGet(sel[0], &p_cmp_path);

    char result[255] = {0};
    strcpy(result, "the sel comp's comppath is: ");

    for(int i=0; i<p_cmp_path.table_num; i++)
    {
        char temp[20] = {0};
        sprintf(temp, "%d - ", p_cmp_path.comp_id_table[i]);
        strcat(result, temp);
    }

    AfxMessageBox(result);
}
```

本范例需要在组立状态下才能正常工作，通过选择组立中的某一个元件，程序将以警告框的形式输出其 ID 的路径标识。

一般来讲，通过选择来获得 PATH 是最简单的方法，在实际应用中，更过的

则是通过用户初始化（用户自己计算 ID 表）的方法来获得元件在装配体中的唯一标识。下面的程序充分说明了这点（程序将自动将一个元件高亮显示，即自动选择到该元件）

```
void UserAutoHighlightComp()
{
    ProMdl mdl;
    ProIdTable memb_id_tab;
    memb_id_tab[0] = 932;    // 将要高亮显示的首层元件的 ID
    memb_id_tab[1] = -1;    // 由于该模型在第一层，所以此处的 -1 不可省略
    ProAsmcomppath p_handle;

    ProMdlCurrentGet(&mdl);
    ProAsmcomppathInit((ProSolid)mdl, memb_id_tab, 1, &p_handle);

    ProMdl p_model;
    ProModelitem p_model_item;

    ProAsmcomppathMdlGet(&p_handle, &p_model);
    ProMdlToModelitem(p_model, &p_model_item);

    ProSelection p_selection;
    ProSelectionAlloc(&p_handle, &p_model_item, &p_selection);

    ProColorotype color = PRO_COLOR_QUILT;
    ProSelectionHighlight(p_selection, color);
}
```

读者可以将该函数通过修改后高亮自己所建立的模型。注意：当所建立的组立模型不止一层时，若要显示第二层或者更深层的元件，则需要按照前面所讲的那样，将该 ID 表填满后才能运行出正确的结果。

至此，有关于 Comppath 的相关内容就介绍完了，虽然大家刚开始可能对这个不好理解，但是通过多写多练之后，一定能加深对它的正确理解和准确运用，在后续的工程实例中，它将以很高的频率出现。

## 自动装配

前面分别介绍了选择装配与 Comppath 的内容，在以上两个内容的基础之上，我们可以进入自动装配的环节了。自动装配其实就是上面两个部分的综合，其主要运用了装配，然后运用了自动选择。在熟悉了前面内容的情况下，相信这个部分已经是水到渠成了。

下面我们直接用一段代码来说明自动装配的原理（引用于帮助文件，可与前

面的选择装配对比，以寻找不同之处)

```
ProError UserAssembleByDatums (ProAssembly asm_model, //输入顶级装配
                                ProSolid comp_model)    //输入装配元件
{
    ProError status;
    ProName asm_datums [3];
    ProName comp_datums [3];
    ProMatrix identity_matrix = {{ 1.0, 0.0, 0.0, 0.0 },
                                   {0.0, 1.0, 0.0, 0.0},
                                   {0.0, 0.0, 1.0, 0.0},
                                   {0.0, 0.0, 0.0, 1.0}};

    ProAsmcomp asmcomp;
    ProAsmcompconstraint* constraints;
    ProAsmcompconstraint constraint;
    int i;
    ProBoolean interact_flag = PRO_B_FALSE;
    ProModelitem asm_datum, comp_datum;
    ProSelection asm_sel, comp_sel;
    ProAsmcomppath comp_path;
    ProIdTable c_id_table;
    c_id_table [0] = -1;

    //装配体中的三个默认装配参照
    ProStringToWstring (asm_datums [0], "ASM_D_FRONT");
    ProStringToWstring (asm_datums [1], "ASM_D_TOP");
    ProStringToWstring (asm_datums [2], "ASM_D_RIGHT");

    //元件的三个默认装配参照
    ProStringToWstring (comp_datums [0], "COMP_D_FRONT");
    ProStringToWstring (comp_datums [1], "COMP_D_TOP");
    ProStringToWstring (comp_datums [2], "COMP_D_RIGHT");

    //将原件载入到装配体中，此时按默认位置摆放
    ProAsmcompAssemble (asm_model, comp_model, identity_matrix, &asmcomp);

    ProArrayAlloc (0, sizeof (ProAsmcompconstraint), 1,
                   (ProArray*)&constraints);

    //给这个元件设定约束条件
    for (i = 0; i < 3; i++)
    {
        status = ProModelitemByNameInit (asm_model, PRO_SURFACE, asm_datums [i],
                                           &asm_datum);
    }
}
```

```

if (status != PRO_TK_NO_ERROR)
{
    interact_flag = PRO_B_TRUE;
    continue;
}

status = ProModelitemByNameInit (comp_model, PRO_SURFACE,
                                comp_datums [i], &comp_datum);
if (status != PRO_TK_NO_ERROR)
{
    interact_flag = PRO_B_TRUE;
    continue;
}

//选中组立中的第 i 个参照，注意此处的 comp_path
ProAsmcomppathInit (asm_model, c_id_table, 0, &comp_path);
ProSelectionAlloc (&comp_path, &asm_datum, &asm_sel);

//选中元件中的第 i 个参照
ProSelectionAlloc (NULL, &comp_datum, &comp_sel);

//设定第 i 个参照组，包含有参照类型以及各自的约束对象
ProAsmcompconstraintAlloc (&constraint);
ProAsmcompconstraintTypeSet (constraint, PRO_ASM_ALIGN);
ProAsmcompconstraintAsmreferenceSet (constraint, asm_sel,
                                     PRO_DATUM_SIDE_YELLOW);
ProAsmcompconstraintCompferenceSet (constraint, comp_sel,
                                     PRO_DATUM_SIDE_YELLOW);

//将这组参照添加入参照集合中
ProArrayObjectAdd ((ProArray*)&constraints, -1, 1, &constraint);

}

//设置参照集合
status = ProAsmcompConstraintsSet (NULL, &asmcomp, constraints);

ProSolidRegenerate ((ProSolid)asmcomp.owner, PRO_REGEN_CAN_FIX);
if (interact_flag)
{
    //如果约束失败则调用系统默认的装配对话框
    ProAsmcompConstrRedefUI (&asmcomp);
}

return (PRO_TK_NO_ERROR);

```

}

在本例中，装配所使用的参照为三个面，读者可以在此基础上使用其他几何约束来进行相应的装配。为了能够实现自动装配，相应的参照几何必须具有标识，笔者常常使用名称来作为标识，读者可能要问，在刚才的装配中都是基准面，基准面作为特征来讲，其名称可以随意修改，但是如果选择一个其它的几何信息，比如一个立方体上的面，又该怎么办呢？

下面笔者就带大家来给一个普通的几何元素命名。操作如下：

打开一个模型窗口，点击“编辑”下的“setup”按钮。在随后弹出的菜单管理器中，选择“名称” 图 5-3



图 5-3

在其下方将扩展出设置名称的子菜单，这时选择“其他”按钮，如图 5-4，

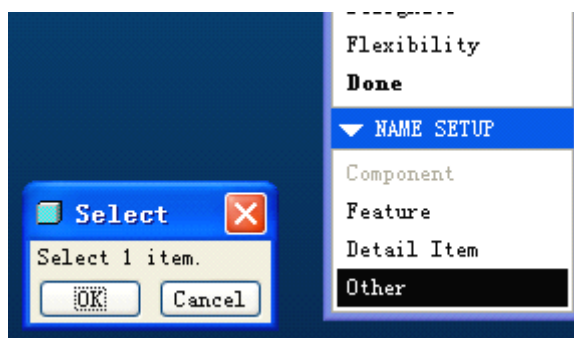


图 5-4

在图形中选择要命名的面，选择后，系统在消息栏提示输入该几何特征的名称，输入完成后点击其后的确定图标，如下图

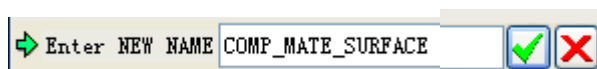


图 5-5



然后退出菜单管理器。这样，这个几何元素便具有了名称，也就可以使用上面介绍的方法来自动选择了。

自动装配的用途非常广泛，与前面的选择装配一样，在标准件的调用上有不可替代的作用，自动装配从表面上看缺乏一定的灵活性，但是这也是其优点所在，如果标准化的程度比较高，那么自动装配将是设计全自动化的基础。另外，如果在自动化装配的过程中通过程序去抓取某些关键位置的尺寸，进而驱动元件库中标准件的尺寸，那么其自动装配的效率将更为显著。

## 元素的遍历与干涉

在一个装配体中，元素的遍历是家常便饭，在遍历时主要的一个问题就是要正确的区分子装配和元件，从而保证当前总装配体下的每一个元件都能够被有效的访问到，由于该功能的利用率极高，这里笔者将代码演示如下：（只提供了访问，动作函数请读者根据需要自行添加）

//过滤函数

```
ProError filter_action(ProFeature* p_feature, ProAppData app_data)
{
    ProFeattype p_type;
    ProFeatureTypeGet(p_feature, &p_type);
    if(p_type != PRO_FEAT_COMPONENT)
    {
        return PRO_TK_CONTINUE;
    }
    return PRO_TK_NO_ERROR;
}
```

//动作函数

```
ProError visit_action(ProFeature* p_feature, ProError status, ProAppData app_data)
{
    ProMdlType type;
    ProMdl mdl;
    ProAsmcompMdlGet((ProAsmcomp*)p_feature, &mdl);
    ProMdlTypeGet(mdl, &type);
    if(type == PRO_MDL_ASSEMBLY)
    {
        ProSolidFeatVisit((ProSolid)mdl,
                           (ProFeatureVisitAction)visit_action
                           ,(ProFeatureFilterAction)filter_action
                           ,NULL);
    }
    if(type == PRO_MDL_PART)
```

```

    {
        .....自行添加对 PART 的操作
    }

    return PRO_TK_NO_ERROR ;
}

// 在主程序中使用到的访问函数：ProSolidFeatVisit

```

```

ProSolidFeatVisit((ProSolid)mdl, (ProFeatureVisitAction)visit_action,
                  (ProFeatureFilterAction)filter_action, NULL);

```

在装配体中，零件的干涉检查也是一项重要内容，虽然 PROE 提供了干涉检查的相关功能，但对于程序来讲，无法从默认功能中获得对象的句柄，为了获取与某个元件所干涉的其他元件的句柄以供后续操作，我们只能自己编写一个干涉检查的函数，笔者用了一个比较慢的方法来实现这种检查，方法如下：

```

#include "prodevelop.h"
#include <pro_interf.h>
/*=====*\
                InterferenceCheck
\=====*/

//子函数：比较两个装配路径是否一样
BOOL UserCompareAsmcomppath(const ProAsmcomppath patha, const ProAsmcomppath pathb)
{
    if(patha.table_num == pathb.table_num)
    {
        for(int i=0;i<patha.table_num;i++)
        {
            if(patha.comp_id_table[i] != pathb.comp_id_table[i])
                return 1;
        }
        return 0;
    }
    return 1;
}

// 主函数：输入一个元件，输出与该元件干涉的所有元件
ProError ProUserInterferenceCheck(const ProAsmcomppath Inputpath, ProArray* Outpath)
{
    int status;
    Prohandle p_model;

```

```

p_model = pro_get_current_object(); //include "prodevelop.h"
if(p_model == NULL)
{
    AfxMessageBox("ProHandle not found");
    return PRO_TK_E_NOT_FOUND;
}
Pro_intf_type interference_type = PROINTERF_SOLID_ONLY;
int n_interferences = 0;
Interference_parts* intf_parts;
status = pro_compute_global_interference(p_model,interference_type,

&n_interferences,&intf_parts,NULL,NULL);
if(status != PRODEV_NO_ERROR)
{
    AfxMessageBox("global_interference error");
    return PRO_TK_GENERAL_ERROR;
}
if(n_interferences == 0)
{
    AfxMessageBox("no interference found");
    return PRO_TK_E_NOT_FOUND;
}

if(n_interferences != 0)
{
    ProError err;
    err = ProArrayAlloc(0,sizeof(ProAsmcomppath),1,Outpath);
    if(err != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("array alloc error");
        return err;
    }

    CString result;
    result.Format("totally %d interferences found", n_interferences);
    AfxMessageBox(result);
    ProAsmcomppath patha, pathb;
    char temp[20] = {0};
    for(int i=0;i<n_interferences;i++)
    {
        ProSelectionAsmcomppathGet((Select3d*)&(intf_parts[i].part1_sel), &patha);
        ProSelectionAsmcomppathGet((Select3d*)&(intf_parts[i].part2_sel), &pathb);
        if(!UserCompareAsmcomppath (Inputpath, patha))
        {

```

```

err = ProArrayObjectAdd(Outpath, PRO_VALUE_UNUSED,1,&pathb);
if(err != PRO_TK_NO_ERROR)
{
    AfxMessageBox("ProArrayObjectAdd failed");
    return err;
}
}
else if(!UserCompareAsmcomppath (Inputpath, pathb))
{
    err = ProArrayObjectAdd(Outpath,PRO_VALUE_UNUSED,1,&patha);
    if(err != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("ProArrayObjectAdd failed");
        return err;
    }
}
}
}

return PRO_TK_NO_ERROR;
}

```

该函数要求输入一个元件在组立中的唯一标识：**Comppath**，结果是输出一个数组，该数组用来存放所有与这个元件有所干涉的其他元件。这样，后续便可以通过直接操作数组中的元素来控制干涉的对象了。（比如高亮显示产生干涉的元件，输出干涉的列表等等）

除了这种方法之外，还有另外一种方法也可以实现类似的功能，不过这要用到函数：**ProFeatureAsmintersectionsCollect** 和 **ProFeatintersectionDataGet** 需要说明的是，对于第一个函数而言，通过函数搜索器搜索出来的结果肯定会让大家失望：

```

#include <ProFeature.h>
ProError ProFeatureAsmintersectionsCollect ( void )

```

## Returns

|                    |                                                    |
|--------------------|----------------------------------------------------|
| PRO_TK_NO_ERROR    | The function found and returned the intersections. |
| PRO_TK_BAD_INPUTS  | One or more arguments is invalid.                  |
| PRO_TK_E_NOT_FOUND | No intersections are found.                        |

图 5-6

搜索器上什么结果都没有，如果看到这个画面，相信很多人都会要放弃了，呵呵，其实这可能是制作帮助文件人员的疏忽吧，我们不要管它，直接进入 **ProFeature.h** 头文件中查找便可。

## 第六节：UDF 特征创建

UDF 在二次开发中的重要作用不言自明，可以说是二次开发中最重要的工具之一。通常，就算在不使用程序的情况下，它也一直是二次开发中的一个开发利器。现在，我们结合程序来使用它，大家可以先预想下其功能将变得多么强大。改善之一便是可以实现图形化界面调用，改善之二是可以通过连接数据库来驱动各种可变尺寸。当然，最大的改善在于：可以通过程序判断，并自动在适当的地方来调用适当的 UDF，从而实现模型设计的自动化。（虽然使用程序也可以直接通过特征元素树来创建特征，但这种方法开销比较大，速度比较慢，这也是为什么本书中压根儿就没有“创建特征”这个章节，读者可以用特征元素树来了解特征创建的一般方法，但笔者不建议在实际应用中使用这种方法）

### 如何制作 UDF （用户自定义特征）

UDF 的创建非常简单，它的实质就是把用户创建的特征组合成一个组，调用时，只要将这个组创建在适当的地方，这个组中的所有特征便都自动创建了出来。UDF 是 Proe 提供的一个非常强大的二次开发工具，我们一定要熟练的掌握其制作和调用方法。还是老规矩，我们先来一起创建一个简单的 UDF，然后再分析其原理。（笔者这里用野火 2.0 来举例，意图在于说明 UDF 创建的一般方法，并不在于介绍什么技巧，）

首先我们在立方体的一个面上建立一个点，然后依托这个点，我们建立一个孔特征，然后用这个孔来制作我们的 UDF。按照图 6-1 点击 UDF 库菜单

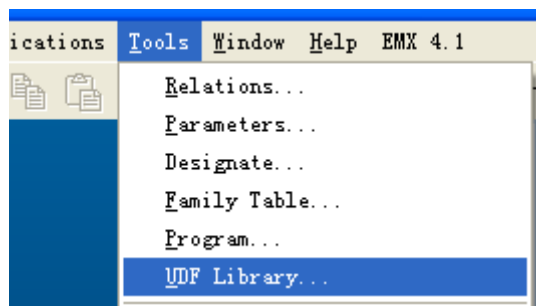


图 6-1

在随后弹出的菜单管理器中选择“创建”

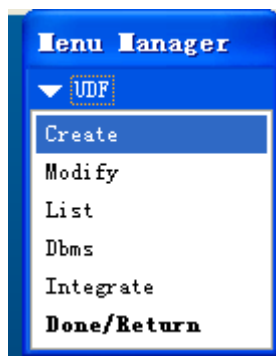


图 6-2

然后在信息提示框输入这个 UDF 的名称，这时，在原来的菜单管理器的基础上弹出了扩展菜单，参照图 6-3 进行选择：



图 6-3

然后信息提示栏会提示是否需要包含参照零件，一般来讲，如果不使用程序调用的话，这里需要选择“是”，如果用程序调用的话，一般选择“否”，这里读者可自己选择。

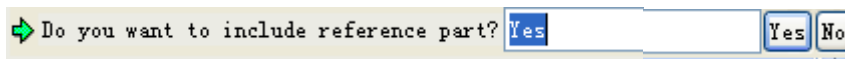


图 6-4

这时候，系统要求用户选择哪些特征来组成这个 UDF。在图形窗口选择完特征后，点击完成（这里我们选择这个孔）。这时候，系统便会将我们在制作这个孔的时候所用到的参照读取出来，并提示我们给这些参照赋予名称以便区别，也便于后续的调用。这里也是技巧比较高的一个地方，如果我们在建立这个 UDF 特征的时候，用的参照过多，那么在后续调用的时候，所需要输入的参照也是很多的，通常是一一对应关系，所以，为了便于后续的调用，在制作 UDF 的时候，我们需要详细考虑需要使用到的参照，争取用最少的参照建立我们想要的 UDF。

在参照设置完毕之后，如果不用调整尺寸，那么就可以直接点击确定按钮来完成整个 UDF 的创建过程，如果需要在 UDF 中更改尺寸或者参数的值，那么我们需要再进行额外的选择，如图 6-5

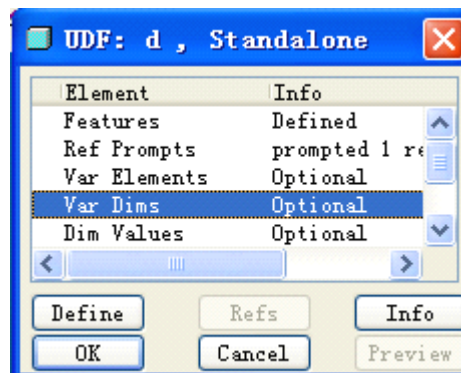


图 6-5

这时候再选择相应的尺寸或者参数。这里有一个需要注意的地方，为了方便后续的编程，拿尺寸为例如，我们应该事前给这些尺寸使用自定义名称，当在此处选取了尺寸之后，系统会让我们设定尺寸在 UDF 中的名称，这个名称要与特征中我们设定的名称一致。否则程序将无法驱动 UDF 中的尺寸。

制作完成的 UDF 特征这会自动存放到当前的工作目录中，我们可以把这个 UDF 文件拷贝至我们的 UDF 库中去。在配置选项 `pro_group_dir` 中，我们可以设置 UDF 库的默认路径。

下面我们来回顾一下整个 UDF 的制作，首先是要选择有哪些特征需要制作成 UDF，然后要给这个 UDF 特征指定必要的放置参照（貌似装配），最后需要指定在这个 UDF 特征中有哪些尺寸或参数具有可变的值。在调用 UDF 的时候，也是严格按照这个顺序的（只有细微差别）。

在制作 UDF 的时候，有几个需要注意的地方，首先是参照关系要考虑好，然后是可变尺寸需要事前将名称定义好。一般来讲，PROE 模型树中的特征都可以制作成 UDF（只有少数不行）。当在组立环境中建立 UDF 的时候，其使用环境也必须是在组立中。另外，我们在组立中也可以将若干元件作为一个 UDF，调用这种类型的 UDF 后的效果与装配类似，通过这种方法可以实现一些新奇的功能。

## PART 环境中调用 UDF 特征

这里向大家介绍如何用程序在普通的 PART 零件中调用一个 UDF 特征。

代码如下：

```
//该函数载入一个 udf 包含三个参照和两个可变尺寸
```

//该 udf 名称为: FORTEST

```
void UserInserUDF()
```

```
{
```

```
    ProError status;  
    ProUdfData data;  
    ProMdl mdl;  
    ProMdlCurrentGet(&mdl);
```

```
    //初始化 UDF 名称
```

```
    ProName udf_name;  
    ProStringToWstring(udf_name, _T"FORTEST");  
    status = ProUdfdataAlloc(&data);  
    ProUdfdataNameSet(data, udf_name, NULL);
```

```
    //初始化 UDF 参照
```

```
    ProSelection* p_in_sel;  
    int sel_n;  
    ProLine temp_ref_name;  
    ProUdfreference udf_ref;
```

```
    //选择点
```

```
    ProSelect("point", 1, NULL, NULL, NULL, NULL, &p_in_sel, &sel_n);  
    ProUdfreferenceAlloc(ProStringToWstring(temp_ref_name, "POINT"), *p_in_sel,  
                        PRO_B_FALSE, &udf_ref);  
    ProUdfdataReferenceAdd(data, udf_ref);
```

```
    //选择顶面
```

```
    ProSelect("surface", 1, NULL, NULL, NULL, NULL, &p_in_sel, &sel_n);  
    ProUdfreferenceAlloc(ProStringToWstring(temp_ref_name, "TOPSURFACE"), *p_in_sel,  
                        PRO_B_FALSE, &udf_ref);  
    ProUdfdataReferenceAdd(data, udf_ref);
```

```
    //选择底面
```

```
    ProSelect("surface", 1, NULL, NULL, NULL, NULL, &p_in_sel, &sel_n);  
    ProUdfreferenceAlloc(ProStringToWstring(temp_ref_name, "BOTTOMSURFACE"),  
                        *p_in_sel, PRO_B_FALSE, &udf_ref);  
    ProUdfdataReferenceAdd(data, udf_ref);
```

```
    //初始化可变尺寸
```

```
    ProName temp_var_dim;  
    double lenth = 30, width = 25 ; //尺寸也可从界面中读取  
    ProUdfvardim vardim;
```



```

//添加长度
ProUdfvardimAlloc(ProStringToWstring(temp_var_dim,"LEN"),lenth,
                  PROUDFVARTYPE_DIM, &vardim);
ProUdfdataUdfvardimAdd(data, vardim);

//添加宽度
ProUdfvardimAlloc(ProStringToWstring(temp_var_dim,"WID"),width,
                  PROUDFVARTYPE_DIM, &vardim);
ProUdfdataUdfvardimAdd(data, vardim);

//创建 udf
ProGroup udf;
ProUdfCreate((ProSolid)mdl, data, NULL,NULL,0,&udf);
ProUdfdataFree(data);
}

```

以上整个调用 UDF 的过程都非常明显，只是有一点笔者在这里需要特别强调一下，这就是函数 ProUdfreferenceAlloc。

```

ProError ProUdfreferenceAlloc (
    ProLine prompt
    /* (In)
       The prompt defined for the reference in the UDF
    */
    ProSelection ref_item
    /* (In)
       The item to be referenced
    */
    ProBoolean external
    /* (In)
       Whether the reference is to an item which is referenced though a parent assembly of the
       solid owning the UDF
    */
    ProUdfreference *reference
    /* (Out)
       The allocated ProUdfreference structure
    */

```

表 7-1

该函数第三个参数非常重要，在使用中经常做错，如果我们只在 PART 模式下调用 UDF，那么这个参数是：PRO\_B\_FALSE，而如果是在组立模式中调用，

那么它的值应该是：PRO\_B\_TRUE. 在我们使用的过程中，为了节省时间，会经常将代码拷贝来使用，而这个函数常常忘了修改（参数不对也可以通过编译）。这在检查错误的时候比较难发现。所以笔者有必要在此略为说明。

## 组立环境中调用 UDF 特征

与在 PART 中的调用一样，请看下面代码：

```
void UserCreateUdfInAsm(ProAsmcomppath p_cmp_path //在该 path 所指的路径中创建 udf
                        , ProSelection p_mdl_point //输入参照一
                        , ProSelection p_mdl_top   //输入参照二
                        , double value           //输入尺寸值
{
    ProError status;
    ProUdfdata data;
    ProMdl mdl;
    ProAsmcomppathMdlGet(&p_cmp_path, &mdl);

    //初始化 UDF 名称
    ProName udf_name;
    ProStringToWstring(udf_name, "screw_kaojian");
    status = ProUdfdataAlloc(&data);
    if(status != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("udf data alloc error");
        return;
    }
    status = ProUdfdataNameSet(data, udf_name, NULL);
    if(status != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("screw_kaojian name get error");
        return;
    }

    //初始化参照
    ProUdfreference udf_ref;

    //选择点参照
    ProUdfreferenceAlloc(L"PLACE_POINT", p_mdl_point, PRO_B_FALSE, &udf_ref);
    status = ProUdfdataReferenceAdd(data, udf_ref);
    if(status != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("screw_kaojian ref of point error");
    }
}
```

```

        return;
    }

    //选择面参照
    ProUdfreferenceAlloc(L"PLACE_SURFACE", p_mdl_top, PRO_B_FALSE, &udf_ref);
    status = ProUdfdataReferenceAdd(data, udf_ref);
    if(status != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("screw_kaojian ref of surface error");
        return;
    }

    //初始化尺寸
    ProUdfvardim vardim;
    status = ProUdfvardimAlloc(L"BIG_DIM", value, PROUDFVARTYPE_DIM, &vardim);
    status = ProUdfdataUdfvardimAdd(data, vardim);
    if(status != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("screw_kaojian -> big_dim error");
        return;
    }

    //创建这个 UDF
    ProGroup udf;
    status = ProUdfCreate((ProSolid)mdl, data, &p_cmp_path, NULL, 0, &udf);
    if(status != PRO_TK_NO_ERROR)
    {
        AfxMessageBox("screw_kaojian create error");
        return ;
    }

    status = ProUdfdataFree(data);
}

```

在以上的程序中，我们假设 UDF 存在两个参照，点和面，同时还存在一个可变尺寸。读者可以将该程序与上面在 PART 模式中的程序对比，对比后发现有两个比较关键的地方，他们分别是函数：ProUdfreferenceAlloc 与 ProUdfCreate，其中，前面一个函数已经讲解过了，这里重点介绍一下后者。

在函数搜索器中搜索该函数，得到如下结果：

```

ProError      ProUdfCreate      (
ProSolid solid
/* (In)

```

```

The solid which will contain the UDF
*/

ProUdfdata data
/* (In)
The data which described the placement and geometry of the UDF
*/

ProAsmcomppath *asm_reference
/* (In)
The assembly and member to reference when placing a UDF with external
references. NULL if the UDF does not require external references.
*/

ProUdfCreateOption *options
/* (In)
Array of placement options
*/

int n_options
/* (In)
The number of options
*/

ProGroup *udf
/* (Out)
The resultant UDF group
*/

```

表 7-2

该函数的所有参数在此描述得非常清楚，这里笔者主要对第三个参数进行说明。这个参数要求输入一个 **Comppath**，前面讲过，**Comppath** 是元件在装配档案中的唯一标识，此处是为了在组立中能够精确定位元件的位置。也就是说，在组立环境下，如果我们要在元件 A 中放置这个 UDF 特征，那么我们必须将 A 的 **Comppath** 传递给函数：ProUdfCreate，这一点非常重要，否则特征创建将会跑位，或者创建失败。

# 第七节：工程图概要

到目前为止，笔者所用到的工程图主要有如下两个方面：表格与符号。（之所以没有创建工程图，是因为还没有实现自动标注，自动标注目前采用的方法是先在模板中标注好一些大概尺寸，然后在调用模型时，将工程图附带一起载入）

## 表格

在工程图中，我们都会用到表格，手动创建表格的一般方法为：点击“表格”菜单，选择“插入”命令。

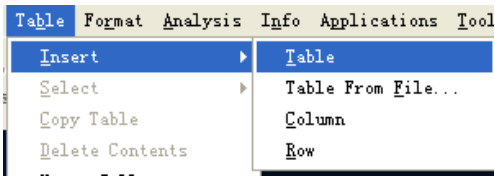


图 7-1

在空白区域点击鼠标，出现一串数字，这串数字便代表你所要建立表格的长度，在相应的数字上点击左键（本例选择的是 8），便会出现图 7-2 画面：

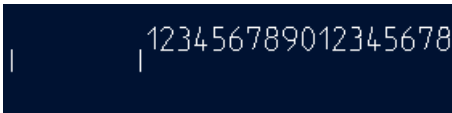


图 7-2

当表格的列数够了之后，点击中键，系统将出现宽度选择的一串数字，按照前面的方法将宽度设置好之后，表格便建立成功了。总的来说，表格的建立还是比较简单的。如果建立好的表格不符合要求，那么可以双击表格，调整其属性。

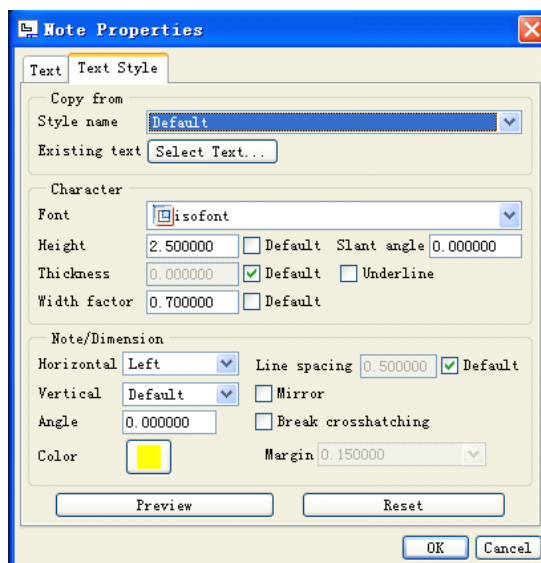


图 7-3

表格在工程图有广泛的应用，比如制作零件清单，自动填写图框等，按照老规矩，我们先来看一段代码：

```
int UserTableCreate(ProDrawing drawing, ProVector pos)    // 子函数：创建表格
{
    ProView*   views;
    int n_views;
    ProDwgtabledata tdata;
    ProDwgtable  table;
    double init_heights[] = {1, 0};
    double widths[] = {20, 10};
    ProHorzJust adjusts[] = {PROHORZJUST_CENTER, PROHORZJUST_CENTER};
    int n_rows;
    int i;
    int sheet;
    ProName v_name;
    ProCharLine name, sheet_str;
    status = ProDrawingViewsCollect(drawing, &views);

    //设置表格
    status = ProDwgtabledataAlloc(&tdata);
    status = ProDwgtabledataOriginSet(tdata, pos);
    status = ProDwgtabledataSizetypeSet(tdata, PRODWGTABLESIZE_CHARACTERS);
    status = ProDwgtabledataColumnsSet(tdata, 2, widths, adjusts);
    status = ProDwgtabledataRowsSet(tdata, 1, init_heights);

    //创建表格
    status = ProDrawingTableCreate((ProDrawing)drawing, tdata, PRO_B_FALSE, &table);
}
```

```

status = TableTextAdd(&table, 1, 1, "View Name");
status = TableTextAdd(&table, 1, 1, "sheet no");
status = ProArraySizeGet(views, &n_views);

//在所有视图中循环
for(i=1; i<=n_views; i++)
{
    //获取视图名称
    status = ProDrawingViewNameGet(drawing, views[i], v_name);

    //获取视图的页面号
    status = ProDrawingViewSheetGet(drawing, views[i], &sheet);
    if(sheet <= 0)
        continue;

    //在表格的末尾新添一行
    status = ProDwgtableRowsCount(&table, &n_rows);
    status = ProDwgtableRowAdd(&table, n_rows, PRO_B_FALSE, 1);

    //在新行中添加视图名称以及当前的页面号码
    ProWstringToString(name, v_name);
    status = TableTextAdd(&table, 1, n_rows+1, name);
    sprintf(sheet_str, "%0d", sheet);
    status = TableTextAdd(&table, 2, n_rows+1, sheet_str);
}

// 更新表格显示
status = ProDrawingTablesUpdate(drawing);
return status;
}

//子函数： 向表格添加数据
ProError TableTextAdd(ProDwgtable* table, int col, int row, char* text)
{
    ProWstring* lines;
    ProLine first_line;

    //准备要填写的内容
    status = ProArrayAlloc(1, sizeof(ProWstring), 1, (ProArray*)&lines);
    ProStringToWstring(first_line, text);
    lines[0] = (ProWstring)first_line;

    //将内容填入表格
    status = ProDwgtableTextEnter(table, col, row, lines);
    status = ProArrayFree((ProArray*)&lines);
}

```

```

        return status ;
    }

//主函数： 菜单入口
ProError MenuAction()
{
    ProMdl drawing;
    ProVector pos;
    ProMouseButton button;

    //初始化当前工程图
    status = ProMdlInit(L"shroud_drawing", PRO_MDL_DRAWING, &drawing);
    if(status != PRO_TK_NO_ERROR) return status;

    // 提示用户在图形窗口选择表格左上角的放置位置
    status = ProMessageDisplay(L"MESSAGE.txt", "user pic the table position");
    if(ProMousePickGet(PRO_ANY_BUTTON, &button, pos) != PRO_TK_NO_ERROR)
        return status;

    // 在选择的位置创建表格
    UserTableCreate((ProDrawing)drawing, pos);
    return PRO_TK_NO_ERROR;
}

```

通过注解读者应该能够很清楚整个制作过程。该程序也涵盖了大部分表格的操作，读者可以在笔者的基础上建立更为复杂的表格。或者在已知表格中添加数据。

## 符号

在上面我们介绍了表格的操作，这里，我们来进行下一个项目：符号。

首先我们来学习如何手动制作一个符号。

制作符号有两种常用的方法，一种是直接利用工程图中的画图工具绘制符号，一种是在现有的符号上加以修改。这里为了让大家更加明白符号的制作过程，我们直接用绘图工具来制作我们的符号。（也可以使用其他工具将符号制作好之后，将符号导入到图档中来）。由于 PROE 的工程图中的绘画工具不是很方便，所以我们在绘画之前必须将网格显示出来。

单击“视图”中的“绘制栅格”“显示栅格”。这时可能看不见栅格，将图面放大后便可以看见了。单击“网格参数”中的“XY 坐标单位”可以设定网格的间距值。虽然现在能够看见栅格，但是，系统还抓不到栅格上的点，我们需要点击“工具”中的“环境”菜单，然后选中“栅格对齐”复选项。这样我们便可以



抓捕到栅格上的点了。

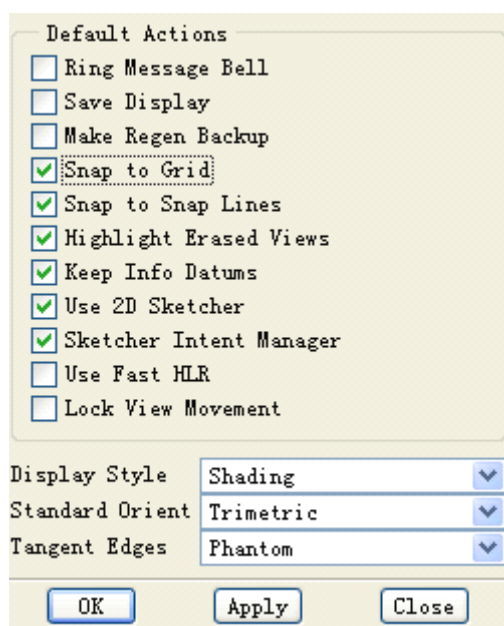


图 7-4

参照下图绘制一个几何图形。

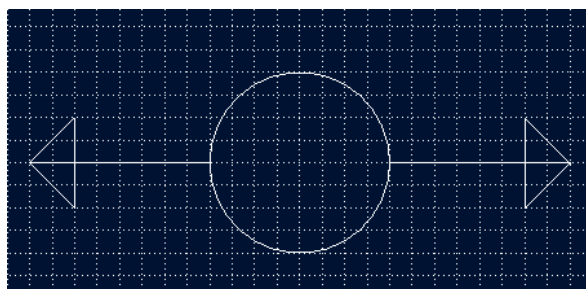


图 7-5

然后我们为这个符号添加一个注释。单击“插入”→“注释”，单击“无方向指引”（noleader）→“输入”→“水平”→“标准”→“圆心”→“制作注释”，然后选择圆形的几何中心来放置注释。输入“\num\”来作为注释文本。采用这种格式是为了能够在调用符号时修改文本内容。不加“\”的话，在符号中，文本不可改变。完成后如图：



图 7-6

准备工作做完了，我们便可以建立这个符号了。

依次单击“格式”“符号库”“定义”，在信息提示栏中输入符号的名称，在菜单管理器中单击“绘图复制”，框选刚才建立的几何信息，点击“确定”然后点击菜单管理器中的“完成”按钮。系统弹出“符号定义属性”对话框。

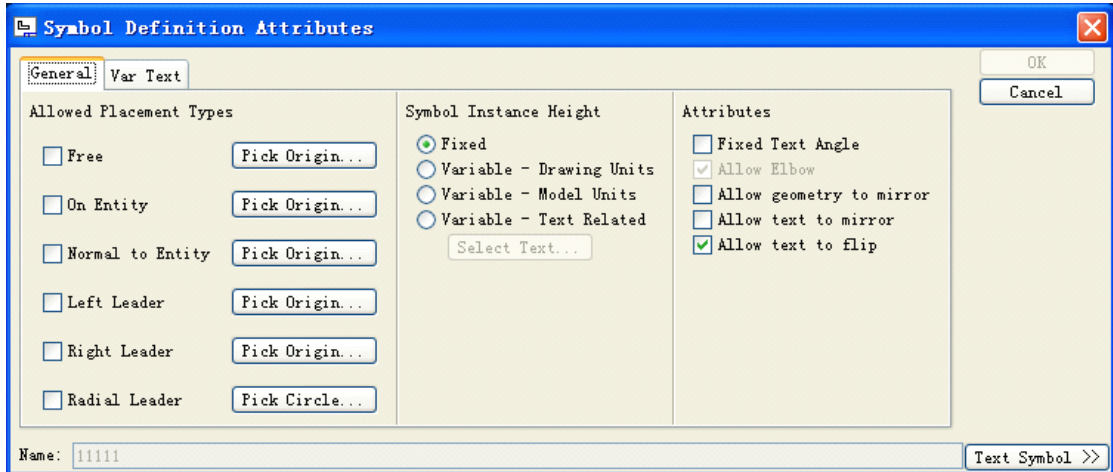


图 7-7

选中复选框“自由”，然后在符号中选择一点（将来符号的放置参照点），选择“可变—相关文本”作为“符号实例高度”的选项，然后选择符号中的注释。接下来我们要定义符号中的可变文本，在“可变文本”标签的右边空白处可以对可变文本进行预设，此处我们预设为 first, second 和 third 三个文本。单击确定完成符号属性定义。单击符号编辑菜单中的完成。单击“符号库”菜单中的“写入”，接受存储符号的默认路径。（注意：“写入”选项可将符号保存为\*.sym格式，从而使其可以在其它绘图中使用，倘若没有选择该项，则保存时符号只存储在当前绘图中。符号当前被存储在默认路径下，通过使用“格式”，“符号库”，“符号目录”来定义路径，可设置存储符号的默认位置。）

要使用该符号，可以依次点击“插入”，“工程图符号”，“客户”，便可调出该符号的使用界面，在“可变文本”标签中，我们可以通过选择预设的值来改变当前的注释。

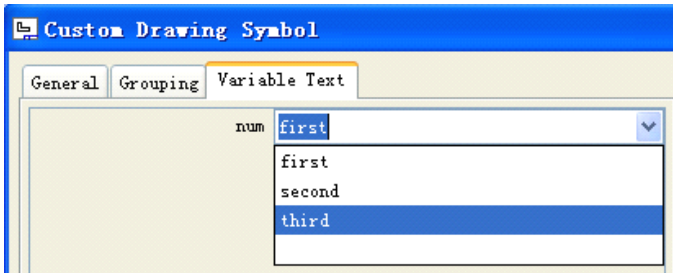


图 7-8

至此，符号的创建就为大家介绍到这里，这里仅仅是给大家大概的介绍了一

下创建的过程，对于对此有兴趣的朋友，可以查阅相关书籍以提高水平。

接下来我们介绍如何用程序来调用符号（此函数与上面我们自制的符号无关）。参见如下代码：

```
#include <ProDtlSymInst.h>    // 新添的头文件
// 子函数：在工程图的某处放置名称为 sym_name 的符号
int UserSymbolCreate(ProDrawing drawing, ProName sym_name, ProVector pos)
{
    ProDtlSymDef symdef;
    ProDtlSymDefData symdata;
    ProDtlSymInstData sym_data;
    ProDtlSymInst sym_inst;
    ProDtlAttach attach;
    ProDtlVartext vartext;
    ProPath sym_def_path;
    ProLine var_text_prompt;
    ProLine var_text_replace;

    ProStringToWstring(sym_def_path, ".");
    ProStringToWstring(var_text_prompt, "DesignChange");
    ProStringToWstring(var_text_replace, "DCG");

    ProDrawingDtlSymDefRetrieve(drawing, sym_def_path, sym_name
                                , PRO_VALUE_UNUSED, PRO_B_TRUE, &symdef);
    // 分配 Data
    ProDtlSymInstDataAlloc(drawing, &sym_data);
    ProDtlSymInstDataDefSet(sym_data, &symdef);

    //分配 Attach
    ProDtlAttachAlloc(PRO_DTLATTACHTYPE_FREE, NULL, pos, NULL, &attach);
    ProDtlSymInstDataAttachTypeSet(sym_data, PROSYMDEFATTACHTYPE_FREE);
    ProDtlSymInstDataAttachmentSet(sym_data, attach);

    // 分配 text
    ProDtlVartextAlloc(var_text_prompt, var_text_replace, &vartext);
    ProDtlSymInstDataVartextAdd(sym_data, vartext);

    // 创建该符号
    ProDtlSymInstCreate(drawing, sym_data, &sym_inst);
    ProDtlSymInstShow(&sym_inst);

    ProDtlAttachFree(attach);
    ProDtlVartextFree(vartext);
}
```

```

        ProDtlSymInstDataFree(sym_data);
        return PRO_TK_NO_ERROR;
    }

// 主函数
ProError MenuAction()
{
    ProError status;
    ProMdl drawing;
    ProVector pos;
    ProMouseButton button;
    ProName sym_name;
    ProStringToWstring(sym_name, "DesignChange");

    status = ProMdlInit(L"shroud_drawing", PRO_MDL_DRAWING, &drawing);
    if(status != PRO_TK_NO_ERROR) return status;
    status = ProMessageDisplay(L"MESSAGE.txt", "user pic the table position");
    if(ProMousePickGet(PRO_ANY_BUTTON, &button, pos) != PRO_TK_NO_ERROR)
        return status;

    UserSymbolCreate((ProDrawing)drawing, sym_name, pos);
    return PRO_TK_NO_ERROR;
}

```

注意：在本程序中所用到的其他头文件请读者从函数搜索器中获取。

在上面的程序中，我们调用了一个名称为“设计变更”的符号。从程序的结构以及注解来看，创建一个符号需要三个步骤，首先是为符号分配数据结构，然后分别填充文字与方位。这样便可以调用这个符号了。

## 将装配模型及其组件载入当前磁盘并改名（含工程图）

前面我们已经知道了如何调用表格和符号，这些操作都是在已有工程图的情况下才能使用，那么能不能自己来建立一个工程图呢？答案是可行的。我们虽然可以直接用程序来根据某个模型来建立相关的工程图，但是，当我们摆放好视图之后，却不太方便实现自动标注。因此，在这种情况下，笔者使用了一个比较笨的方法：对于标准件，我们可以先将其工程图制作好，而且相关的尺寸也标注完毕。在使用这个标准件的时候，我们只要将他们一起调过来使用就可以了，同时，由于工程图与模型之间本来就有依赖关系，当我们改变了模型中的那些尺寸之后，工程图将会自动更新。这样便能节省大量标注时间了。

下面的代码给大家演示了如何将工程图与模型一起载入当前磁盘中。（笔者虽然已经尽力将其它不相关代码剪掉了，但这段代码还是稍长，请读者不要失去耐性）

```

#include <ProFeatType.h>
ProError RenameAction (ProFeature* p_feature, ProError status ,ProAppData app_data);
ProError filter_action(ProFeature* p_feature, ProAppData app_data);

typedef struct UserData
{
    CString path;
    CString newname;
}Mydata;

void UserLoadAsm()
{
    //设置选项后，可将内存中的模型存储在当前工作路径中
    ProName option;
    ProPath option_value;
    ProStringToWstring(option, "override_store_back");
    ProStringToWstring(option_value, "yes");
    ProConfigoptSet(option, option_value);

    // 初始化数据
    ProError status ;
    ProMdl topasmhandle;
    ProMdl newasmhandle;
    Mydata data;
    data.newname = "PRINT"; //输入新前缀
    data.path = "F:\\ABC\\"; //输入路径
    CString temppath;
    temppath = data.path;
    temppath += "test.asm"; //输入原始组立名称
    ProPath asmpath;
    ProStringToWstring(asmpath,(LPTSTR)(LPCTSTR)temppath);

    //载入组立
    ProMdlLoad(asmpath, PRO_MDL_UNUSED,PRO_B_FALSE,&topasmhandle);
    ProSolidFeatVisit((ProSolid)topasmhandle
        ,(ProFeatureVisitAction)RenameAction
        ,(ProFeatureFilterAction)filter_action
        ,(ProAppData)&data);

    //更改名称后存储
    ProName newname;
    ProStringToWstring(newname,(LPTSTR)(LPCTSTR)data.newname);
    ProMdlRename(topasmhandle,newname);
    ProMdlSave(topasmhandle);
}

```

```

//还原选项值
ProStringToWstring(option_value, "no");
ProConfigoptSet(option, option_value);
}

ProError filter_action(ProFeature* p_feature, ProAppData app_data)
{
    ProFeattype p_type;
    ProFeatureTypeGet(p_feature, &p_type);
    if(p_type != PRO_FEAT_COMPONENT)
    {
        return PRO_TK_CONTINUE;
    }
    return PRO_TK_NO_ERROR;
}

ProError RenameAction(ProFeature* p_feature, ProError status, ProAppData app_data)
{
    //传输数据以备后续循环使用
    Mydata data;
    data.newname = ((Mydata*)((Mydata*)app_data)).newname;
    data.path = ((Mydata*)((Mydata*)app_data)).path;
    CString temppath;
    temppath = data.path;
    temppath += "\\";

    //获取模型的句柄 类型 以及名称
    ProMdl childmdl;
    ProAsmcompMdlGet((ProAsmcomp*)p_feature, &childmdl);
    ProMdlType type;
    ProMdlTypeGet(childmdl, &type);
    ProName oldname;
    ProMdlNameGet(childmdl, oldname);

    //判断是否已经改过名称，如果没有则执行下列操作
    wchar_t tempname[6] = {0};
    ProWstringCopy(oldname, tempname, 5);

    ProName newmdlname;
    ProStringToWstring(newmdlname, (LPTSTR)(LPCTSTR)data.newname); //获取传入的名称

    int compare;
    ProWstringCompare(tempname, newmdlname, PRO_VALUE_UNUSED, &compare);
}

```

```

if(compare)
{
    ProPath drawpath;
    ProStringToWstring(drawpath,(LPTSTR)(LPCTSTR)temppath);

    ProWstringConcatenate(oldname,drawpath,PRO_VALUE_UNUSED);
    ProWstringConcatenate(L".drw",drawpath,PRO_VALUE_UNUSED);// 连接图档完整
路径
    ProMdl drawmdl; //载入图档
    status = ProMdlLoad(drawpath, PRO_MDL_DRAWING, PRO_B_FALSE,&drawmdl);
    if(status == PRO_TK_NO_ERROR)
    {
        ProWstringConcatenate(oldname, newmdlname, PRO_VALUE_UNUSED);
        ProMdlRename(childmdl, newmdlname);
        ProMdlRename(drawmdl, newmdlname);
        ProMdlSave(drawmdl);
        ProMdlSave(childmdl);
    }

    if(status != PRO_TK_NO_ERROR)
    {
        ProWstringConcatenate(oldname, newmdlname, PRO_VALUE_UNUSED);
        ProMdlRename(childmdl, newmdlname);
        ProMdlSave(childmdl);
    }

    //如果是组立则扫描其中组件
    if(type == PRO_MDL_ASSEMBLY)
    {
        ProSolidFeatVisit((ProSolid)childmdl
                           ,(ProFeatureVisitAction)RenameAction
                           ,(ProFeatureFilterAction)filter_action
                           ,(ProAppData)&data);
    }
}

return PRO_TK_NO_ERROR;

}

```

在这段代码中，我们从远程载入了一个模型以及其工程图到当前的工作路径，在载入的过程中，我们在该模型的名称前面添加了一个前缀，如果该模型是一个装配档案，则遍历其中所有元件并统一添加新前缀，由于修改了选项值，所

以最后需要还原选项值。

这段代码所用到的功能有如下几个方面：载入模型，修改名称，修改选项值，遍历组立，存储，以及宽字符的比较。

## 将工程图导出为其它格式

在实际工作当中，可能常常需要将 PROE 的工程图转换为其它格式，针对这个需求，PROE 提供了相关的函数。

下面的代码给大家演示了转档的一般过程，读者可以查询相关的函数实现更为符合自己需要的功能。

```
int UserExportFile(ProDrawing export)
{
    ProFileName w_filename;
    ProPrintPrinterOpts printer_opts;
    ProPrintMdlOpts model_opts;
    ProPrintPlacementsOpts placement_opts;
    int window;
    ProPath pcf_filename;

    //转出为 STP 档案
    ProMdlNameGet(export, w_filename);
    ProUtilWstrStrcatToWstr(w_filename, ".stp");
    Pro2dExport(PRO_STEP_FILE, w_filename, export);

    //转出为 PCF 档案
    ProStringToWstring(pcf_filename, ".\\new_pcf.pcf");
    ProPrintPCFOptionsGet(pcf_filename, export, &printer_opts, &model_opts,
    &placement_opts);
    ProMdlNameGet(export, printer_opts.filename);
    ProUtilWstrStrcatToWstr(printer_opts.filename, ".plt");
    ProMdlWidowGet(export, &window);
    ProPrintExecute(window, &printer_opts, &model_opts, &placement_opts);
    return PRO_TK_NO_ERROR;
}
```



## 第八节：界面一瞥

在这一节中，笔者要向大家介绍界面方面的内容，该部分内容的目的就是使读者能够上手制作属于自己的界面。所以，本节重点在于引导大家入门而不会详述每一个内容。（以下两个内容都可以作为一本书介绍）

### ProE 系统 UI 界面一瞥

PROE 系统的 UI 基本上都存放于安装路径的 `text\resource` 下，例如，本机上该文件夹为：`E:\ProEngineer\proe2.0\proeWildfire 2.0\text\resource`。其对话框以 `.RES` 命名，如果了解这种 UI，大家可以查看这个文件夹下的系统资源。在本书中，笔者只对常用的几个 UI 控件进行说明。

建立这种 `RES` 文件非常简单，首先从这个文件夹中随便拷贝一个以 `.res` 命名的文件，用文本编辑器打开后，将其中的内容删除，这样我们就可以在里面写自己的 UI 界面了。另外一种方法是直接建立一个 `TXT` 文件，然后将其后缀名称修改为 `res`。但是这种方法并不是每次都行，有时候系统会提示说文件结尾格式不对。

编写好的 UI 可以用 PROE 自带的一个 UI 查看器来查看，这个查看器名称为：`prodialog_view.exe`，本机的存放目录为：`E:\ProEngineer\proe2.0\proeWildfire 2.0\protoolkit\i486_nt\obj`，查看时，将 `RES` 文件直接拖拽到这个可执行文件的图标上便可。

下面我们就来编写我们自己的 UI 对话框。

```
//第一个对话框.res
(Dialog 第一个对话框
(Components
  (PushButton OK)
  (PushButton Cancel)
  (PushButton Show)
  (Label      Label)
)

(Resources
  (OK.Label      "OK")
  (OK.TopOffset  4)
  (OK.BottomOffset 4)
  (OK.LeftOffset 4)
  (OK.RightOffset 4)
```

```

        (Cancel.Label      "Cancel")
        (Cancel.TopOffset  4)
        (Cancel.BottomOffset 4)
        (Cancel.LeftOffset 4)
        (Cancel.RightOffset 4)

        (Show.Label      "Cancel")
        (Show.TopOffset  4)
        (Show.BottomOffset 4)
        (Show.LeftOffset 4)
        (Show.RightOffset 4)

        (Label.Label      "my first dialog")

        (.Label            "第一个对话框")
        (.StartLocation    5)
        (.DefaultButton    "Show")
        (.Layout
            (Grid (Rows 1 1 1) (Cols 1)
                Label
                (Grid (Rows 1) (Cols 1 1)
                    Cancel
                    Show
                )
                OK
            )
        )
    )
)

```

从字面意思基本上就可以理解整个对话框的构造以及搭建过程。这里就不用多讲了，以上对话框的效果如图：8-1



图 8-1

结合这个对话框的界面，我们详细介绍下这几句：

```

(.Label            "第一个对话框")    // 整个界面的总属性：标题
(.StartLocation    5)                //对话框出现在屏幕中的位置,数值

```

为 1--9

```
(.DefaultButton    "Show")    // 默认为选中的控件
(.Layout            // 布局管理器
  (Grid (Rows 1 1 1) (Cols 1)  // 采用网格布局，这里为 3 行 1 列
    Label              // 第一行放置文本标签
    (Grid (Rows 1) (Cols 1 1)  // 将第二行分割为两列
      Cancel            // 第二行第一列放置 Cancel 按钮
      Show              // 第二行第二列放置 Show 按钮
    )
  )
  OK                    第三行放置 OK 按钮
)
)
```

第一个对话框就介绍完毕，接下来，我们来学习其他几个常用控件。还是直接看资源描述

```
// 第二个对话框.res
(Dialog 第二个对话框
  (Components
    (PushButton    OK)
    (Label          LabelA)
    (RadioGroup     RadioGroupA)
    (PushButton     Cancel)

    (Separator      SepA)
    (Label           LabelB)
    (CheckButton     CheckA)
    (CheckButton     CheckB)
    (CheckButton     CheckC)
  )

  (Resources
    (OK.Label        "OK")
    (OK.TopOffset     4)
    (OK.BottomOffset  4)
    (OK.LeftOffset    4)
    (OK.RightOffset   4)

    (Cancel.Label     "Cancel")
    (Cancel.TopOffset  4)
    (Cancel.BottomOffset 4)
    (Cancel.LeftOffset 4)
    (Cancel.RightOffset 4)
  )
)
```

```

(LabelA.Label      "选项组")
(LabelA.TopOffset  4)
(LabelA.BottomOffset  4)
(LabelA.LeftOffset  4)
(LabelA.RightOffset  4)

(RadioGroupA.Orientation  True)
(RadioGroupA.Alignment    True)
(RadioGroupA.Names        "n1" "n2" "n3")
(RadioGroupA.Labels       "first" "second" "third")

(SepA.TopOffset  4)
(SepA.BottomOffset  4)

(LabelB.Label      "checkgroup")
(LabelB.TopOffset  4)
(LabelB.BottomOffset  4)
(LabelB.LeftOffset  4)
(LabelB.RightOffset  4)

(CheckA.Label      "ca")
(CheckB.Label      "cb")
(CheckC.Label      "cc")
(CheckA.Set        True)

(.DefaultButton    "OK")
(.Label            "第二个对话框")
(.Layout
  (Grid (Rows 1 1 1 1) (Cols 1)
    (Grid (Rows 1) (Cols 1 1)
      LabelB
      (Grid (Rows 1 1 1) (Cols 1)
        CheckA
        CheckB
        CheckC
      )
    )
  )

  SepA

  (Grid (Rows 1) (Cols 1 1)
    LabelA
    RadioGroupA
  )

```

这段资源文件效果如图



### 第三个对话框.res

```
(Resources
  (OK.Label      "OK")
  (OK.TopOffset  4)
```

|                      |                                |
|----------------------|--------------------------------|
| (OK.BottomOffset     | 4)                             |
| (OK.LeftOffset       | 4)                             |
| (OK.RightOffset      | 4)                             |
| (Cancel.Label        | "Cancel")                      |
| (Cancel.TopOffset    | 4)                             |
| (Cancel.BottomOffset | 4)                             |
| (Cancel.LeftOffset   | 4)                             |
| (Cancel.RightOffset  | 4)                             |
| (LabelA.Label        | "inputbox:")                   |
| (LabelA.TopOffset    | 4)                             |
| (LabelA.BottomOffset | 4)                             |
| (LabelA.LeftOffset   | 4)                             |
| (LabelA.RightOffset  | 4)                             |
| (InputA.LeftOffset   | 2)                             |
| (InputA.RightOffset  | 4)                             |
| (LableB.Label        | "listbox:")                    |
| (LableB.TopOffset    | 4)                             |
| (LableB.BottomOffset | 4)                             |
| (LableB.LeftOffset   | 4)                             |
| (LableB.RightOffset  | 4)                             |
| (ListA.Names         | "li1" "li2" "li3")             |
| (ListA.Labels        | "value1" "value2" "value3")    |
| (ListA.RightOffset   | 4)                             |
| (LabelC.Label        | "optionmenu:")                 |
| (LabelC.TopOffset    | 4)                             |
| (LabelC.BottomOffset | 4)                             |
| (LabelC.LeftOffset   | 4)                             |
| (LabelC.RightOffset  | 4)                             |
| (OptionA.Names       | "opt1" "opt2" "opt3")          |
| (OptionA.Labels      | "choose1" "choose2" "choose3") |
| (OptionA.RightOffset | 4)                             |
| (LabelD.Label        | "text:")                       |
| (LabelD.TopOffset    | 4)                             |
| (LabelD.BottomOffset | 4)                             |
| (LabelD.LeftOffset   | 4)                             |
| (LabelD.RightOffset  | 4)                             |

```

        (AreaA.Rows 10)
        (AreaA.Columns 25)
        (AreaA.RightOffset 4)

        (.DefaultButton "OK")
        (.Label "第三个对话框")
        (.Layout
            (Grid (Rows 1 1 1 1 1) (Cols 1 1)
                LabelA
                InputA
                LabelB
                ListA
                LabelC
                OptionA
                LabelD
                AreaA
                OK
                Cancel
            )
        )
    )
)
)

```

这个对话框的效果如图 8-3

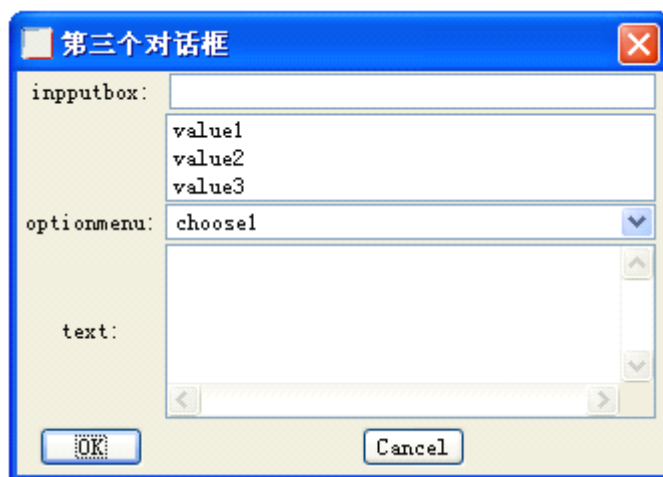


图 8-3

这个对话框中包含了大部分信息输入，输出控件，读者应该尽量的去熟悉它们的用法.

到这里，笔者常用的 UI 控件的搭建就介绍完了，读者可以自己到 PROE 系

统资源中去查看各个控件的属性（笔者目前也没有把所有属性都了解，只是做到够用）另外还有其它一些控件与界面的搭建，如滑动条，带菜单的对话框等等，都可以在那里找到，读者可以通过对它们的学习以制作出符合自己实际需要的更高级的对话框。

另外还有一点，如果我们在对话框中添加了背景图片，在对话框预览器中是看不见的，需要等对话框加载到程序中之后才能看见。

下面我们来学习如何将自己搭建的对话框加载到 ProE 系统中。为了简单起见，我们将使用前面我们制作的第一个对话框来举例。

代码如下：

```
#include <ProUIDialog.h>
#include <ProUIPushbutton.h>
#define Cancel 0

// 定义退出动作
void Exit_ACTIONIN(char* dialog, char* component, ProAppData data)
{
    ProUIDialogDestroy(dialog);
}

// 定义OK按钮的动作函数
void OK_ACTIONIN(char* dialog, char* component, ProAppData data)
{
    AfxMessageBox("ok");
}

//定义cancel按钮动作函数
void Cancel_ACTIONIN(char* dialog, char* component, ProAppData data)
{
    AfxMessageBox("cancel");
}

//定义show按钮动作函数
void Show_ACTIONIN(char* dialog, char* component, ProAppData data)
{
    AfxMessageBox("show");
}

//主函数：创建对话框
void UserTestShowUI()
{
```



```

ProError status;
char* dialog_name = "第一个对话框";
int dialog_status;
ProUIDialogCreate(dialog_name, dialog_name); // 创建对话框

// 以下设置各个动作函数
ProUIDialogCloseActionSet(dialog_name, Exit_ACTION, NULL);
ProUIPushbuttonActivateActionSet(dialog_name, "OK", OK_ACTION, NULL);
ProUIPushbuttonActivateActionSet(dialog_name, "Cancel", Cancel_ACTION, NULL);
ProUIPushbuttonActivateActionSet(dialog_name, "Show", Show_ACTION, NULL);

//激活并显示对话框
ProUIDialogActivate(dialog_name, &dialog_status);
}

```

然后，我们需要把这个对话框的资源文件“第一个对话框.RES”放置到我们工程中的 Resource 文件夹中。这样，当用户激活这个函数时，我们建立的对话框便显示在了图形窗口中。当用户点击上面的按钮时，便会激活相应的动作函数。当然，本例中只是实现了最简单的功能而已，连读取数据的动作也没有涉及，这主要是为了让代码更简洁，也便于大家理解基本的创建过程。

从代码看，要使用一个对话框必须具备这些步骤：首先在内存中创建对话框，然后设置其上的每一个控件所对应的动作函数，最后激活并显示这个对话框。对于在这段简单代码中所包含的这几个函数，大家可以查询函数搜索器，以便加深理解。当然，读者也可以添加其他与数据输入输出相关的控件，并通过函数去读取这些控件中的数据，笔者这里就不详细叙述了，本书的主要目的是让大家能够入门，并积极主动的去探索适合自己开发的途径。有关图形界面的详细内容，读者若想加深了解，一定要经常查阅帮助文件，或者直接查询相关头文件。只有这样，读者才能够在此基础上制作出优美的图形界面。

## VC 界面一瞥

在这一小节中，我们来了解如何建立 VC 中的简单对话框并将其与 PROE 中的 UI 做简要对比。

VC 界面的制作比前面我们学习的 PROE 中默认的界面要简单得多，有关 VC 界面的书籍也是多如牛毛，不过用 VC 开发界面有个小缺点，就是它的界面在转移上不是很方便，也就是移植性不强，总会少这少那的。（也可能是笔者水平不咋样），建议如果是做大规模开发，还是选择 proe 默认的 UI 比较好，如果是小型的开发，那么则可以考虑使用 VC，因为 VC 界面的确做起来非常方便，使用 VC 的 IDE，我们可以直接将控件拖拽到界面编辑器中，而不需要一个一个的去指定位置。

因为 VC 界面的内容非常多，加之刚才也讲到，笔者的水平有限，所以此处只是略为介绍其基本的操作过程，读者可以在此基础上建构自己的界面，或者干脆直接上书店买本有关 VC 界面制作的书，慢慢研究。(^^)

此处我们还是使用 VC6.0 来制作一个对话框。

在 6.0 的界面中，我们按照图中所示，选择插入菜单的资源选项：

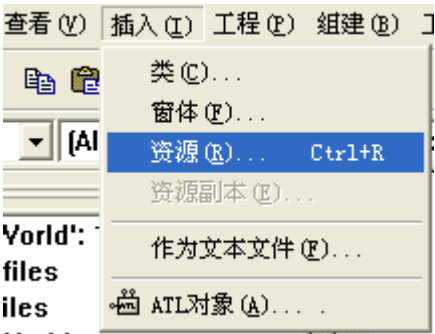


图 8-4

系统弹出资源对话框，在里面我们选择“对话框”后，点击新建：



图 8-5

于是系统弹出对话框编辑器：

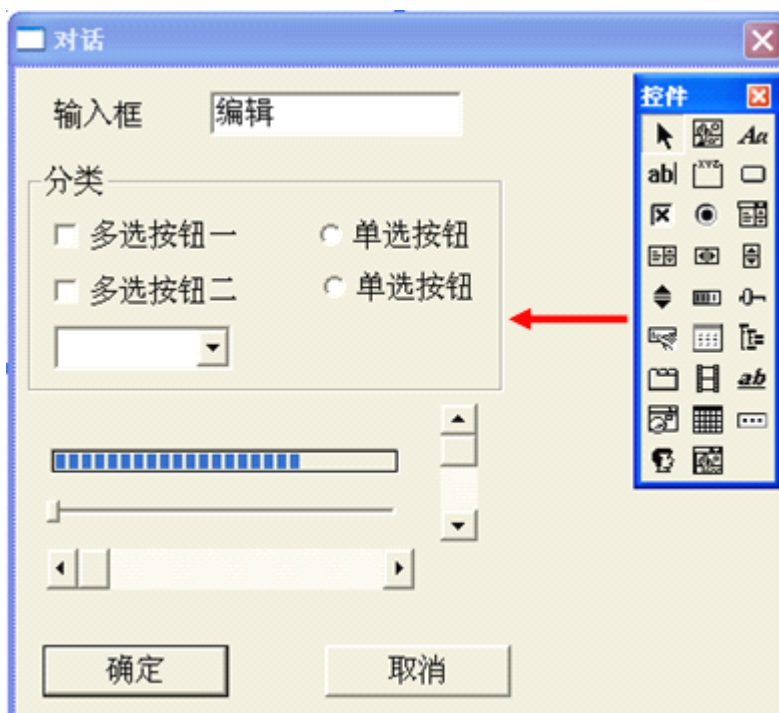


图 8-6

如图所示，右边的工具条上是我们常用的一些对话框控件，当建构更为复杂的界面的时候，这些控件也不能完全满足我们的需要，所以需要添加 ActiveX 控件，当然，我们这里并不使用 X 控件。首先，我们在工具条中选择我们所需要的控件，然后将他们拖拽到对话框编辑面板中调整尺寸，设置变量，修改 ID，改变属性，等等，在本例中，我们什么也不做，（仅仅是给大家灌输一种观念，以便与 PROE 系统的 UI 界面做对比），只是将所需要的空间拖到编辑器中便可。这一步相当于我在 PROE 中编写资源文件，在资源文件中，我们必须手工设置每一个控件的位置，还要设置控件的排版，而在 VC 中，我们只需要拖拽几下，便可轻松完成整个界面的排版过程。

下面我们来将这个对话框添加到程序中。

在 VC 中，每一个对话框都是一个类（控件也是）。对于如上对话框，我们必须让它与我们的类相联系起来，在对话框编辑器的任意空白位置双击鼠标左键，系统询问是否创建一个新类，点击 OK，后出现创建类的对话框。在该对话框中我们输入该类的名称后点击确定。这样系统便自动为我们刚才创建的界面指定了一个类。在这个类中包含了我们所有的控件的定义与动作函数（需要自己手动设定动作内容，这与 PROE 中的动作设定是一样的，此处我们仅作演示，不设置任何动作）

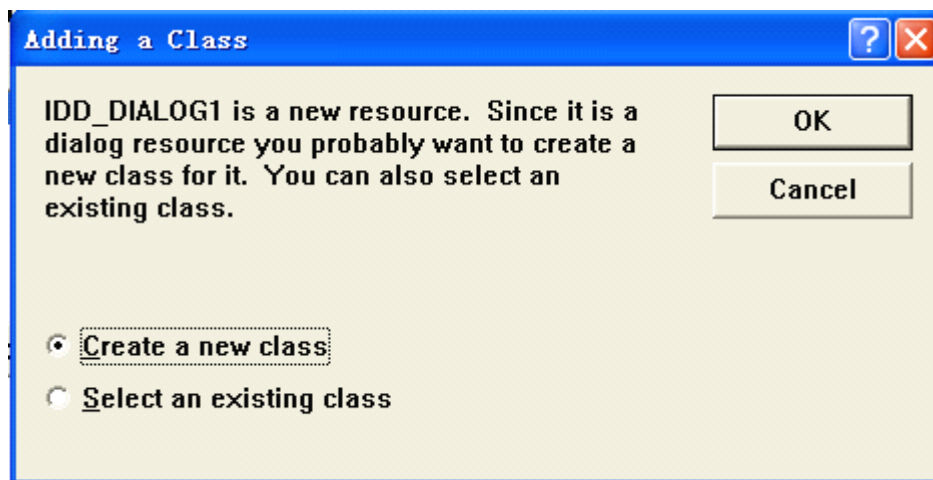


图 8-7

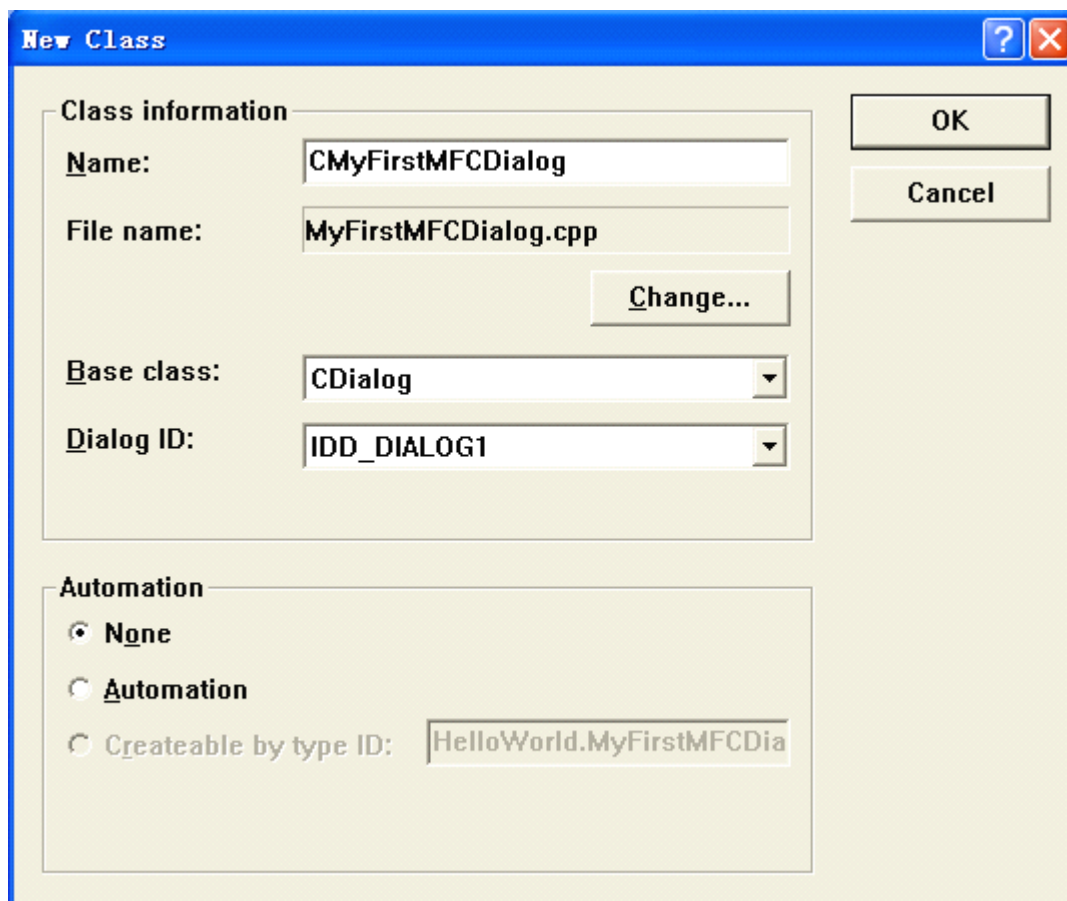


图 8-9: 定义自己的类

定义完成自己的类之后，我们只需要将其加载在菜单函数中便可以了。方法如下：

```
int MenuShowMFCDialog()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

```

CMyFirstMFCDialog* MFCDlg = new CMyFirstMFCDialog;
MFCDlg ->Create(IDD_DIALOG1, NULL);
MFCDlg ->ShowWindow(SW_SHOW);
MFCDlg = NULL ;
return 0 ;
}

```


这样，当我们在菜单中激活这个菜单命令之后，我们在上面创建的对话框便以非模态的形式出现在了图形窗口中（非模态是指用户在操作这个对话框的同时，还可以切换到其他物件上操作，与此相对的是模态对话框，模态对话框除了自己外，不允许用户操作其他物件。通常我们都会创建非模态对话框，以便对图形窗口进行操作），读者可能发现在移动这个对话框的过程中，图形窗口会有对话框的轨迹出现，为了解决这个问题，我们可以重载对话框的 `ONMOVE` 函数，在该函数中使用 `ProWindowRefresh` 去刷新窗口。

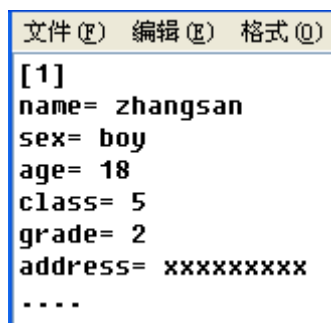
至此，我们的界面讲解全部结束，在讲解的过程中，我们只是带领读者了解了两种不同方式下的创建过程，尤其是对 VC 界面下的对话框，更是只略带的提了一下而已，还是那句话，读者若希望能够制作出做工精良的对话框，一定要查阅相关的资源，网上和书店都有相当多的书籍在专门介绍对话框的制作。笔者在此祝愿大家能早日熟练对话框的制作，以提高用户在使用时候的满意度。

## 第九节：数据库

数据库在程序的应用中有非常重要的地位,可以说,没有数据库的程序不是正真的程序。程序的主要功能便是实现与数据的交互,很难想象一个程序可以独立于数据而单独存在(小程序除外),虽然笔者在 PREO 二次开发中使用的数据相对而言比较少,但是作为一种基本手段,我们还是应该多少了解一些数据库方面的知识。为了简单起见,这里只讲解微型数据库的知识。(其它数据库其实也是这样演变而来的)

### 用 INI 文件制作系统配置文件

INI 文件通常用于设置系统的各种属性,其图标为: , 书写这种文件时有固定的格式,如图:



```
文件(F) 编辑(E) 格式(O)
[1]
name= zhangsan
sex= boy
age= 18
class= 5
grade= 2
address= xxxxxxxx
....
```

图 9-1

我们可以把它看成一个微型数据库,在如上的数据库中便记录了张三的各种信息。VC 提供了专门的函数对这种文件中的数据如字符串,整数等类型进行读写操作。比如说,对于字符串,VC 便提供了函数: `::GetPrivateProfileString` 以及 `::WritePrivateProfileString`。下面我们就来介绍一下这两个函数的具体用法。(其它函数的用法与此类同)

请看下面的代码剪辑:

```
char path[255] = {0};
GetCurrentDirectory(255,path);
strcat(path,"\\setting.ini"); // 获取当期路径下的 ini 文件
CString str;
int j=1;
str.Format("%d",j);
char restr[255]={0};
::GetPrivateProfileString(str,"name","",restr,255,path);
```

在上面的剪辑中,我们首先获取了 ini 文件的绝对路径,然后使用

**GetPrivateProfileString** 从这个文件的标识号为 “str （序号为 1）” 的内容块中，读取了名称为 **name** 的关键字后面的内容，并将其存放到了 **restr** 字符数组中，以备程序的其它部分使用。

同样，我们也可以对该文件进行写操作，如下面的程序片段：

```
char path[255] = {0};
GetCurrentDirectory(255,path);
strcat(path, "\\setting.ini");
CString S_index;
int index = 1;
S_index.Format("%d",index);

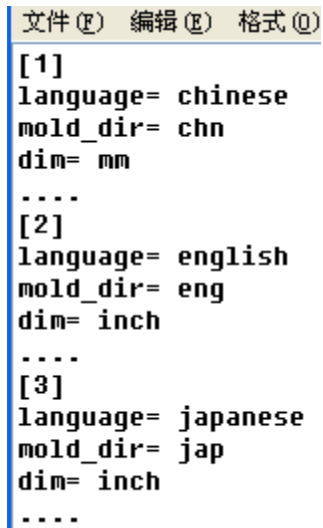
::WritePrivateProfileString(S_index, "name", "zhangsan", path);
```

可以看到，操作同样非常简单，与读取相比，写操作所需要的参数更少了。在上面的程序中，我们在当前目录的 INI 文件中，序号为 1 的数据块里，找到关键字 **NAME**，然后将其所对应的字符串的值修改为 **zhangsan**

对于 INI 文件来讲，如果使用 VC 提供的操作函数来进行读写是十分方便的，如上面的两个程序片段就非常轻松的实现了文件的读写。因此，基于这个原理，我们也可以将该文件制作为微型数据库，读者可以将自己在程序中要用到的一些数据使用一定的格式存储在该文件中，然后使用函数进行读写。不过，对于数据库的常用操作，如查询等，VC 并没有提供相应的函数，此时可以自己根据固有的一些函数来修改，扩展为所需要的函数。

由于这种文件需要固定的格式，同时，其排版只能以行为单元，也就是说，其一行只能存放一个数据。实际在使用的时候，一般都不把这种文件用来做数据库。这种文件在使用的时候，更多的是被当做整个系统的环境配置文件使用。只要按照我们自己的方式将各个环境序号设置好，在程序中，我们便可以通过读取该文件而改变不同的程序使用环境。

如图，便根据序号的不同而指定了三种不同的语言：



```

文件(F) 编辑(E) 格式(O)
[1]
language= chinese
mold_dir= chn
dim= mm
....
[2]
language= english
mold_dir= eng
dim= inch
....
[3]
language= japanese
mold_dir= jap
dim= inch
....

```

图 9-2

补充：循环读取 ini 中所有内容

```

void CFILEDlg::LIST_UP()    //根据文件内容更新控件内容
{
    char path[255]={0};
    ::GetCurrentDirectory(256,path);
    strcat(path,"\\file.ini");
    m_LIST.DeleteAllItems(); // m_LIST 是一个控件，此处为删除控件中所有元素以备写入
    新内容
    int j=1;
    CString str;
    str.Format("%d",j); // INI 文件中的访问序号

    char buf[512]={0};
    int index;
    char restr[255]={0};
    CString temp;
    int age;
    int ROW_NUMBER=0,count;
    while(GetPrivateProfileSection(str,buf,512,path)) //若没到文件末尾
    {
        index=::GetPrivateProfileInt(str,"index",0,path); //读取整数
        if(!index) return;
        ::GetPrivateProfileString(str,"name","",restr,255,path);
        count=m_LIST.InsertItem(ROW_NUMBER,restr);
        age=::GetPrivateProfileInt(str,"age",0,path);
        temp.Format("%d",age);
        m_LIST.SetItemText(count,1,temp);
        ROW_NUMBER++; //让控件中的输入序号加 1
        j++; //让 INI 文件中的访问序号加 1
    }
}

```



```

        str.Format("%d",j);
    }
}

```

在本书最后一章的工程专题中，就涉及到了自定义配置文件的读写，读者可参考其用法。此处对 INI 文件的操作就简单介绍到这里。

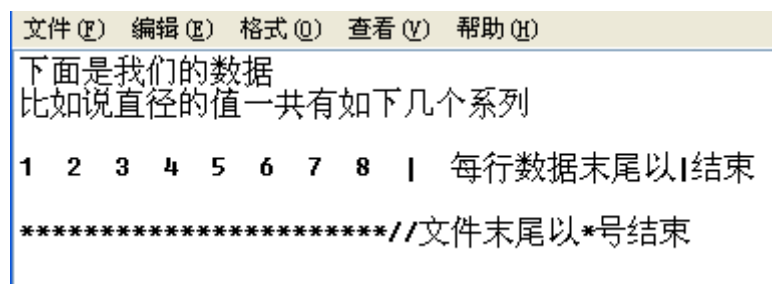
## 用 TXT 文件制作微型数据库

在上面我们简单的介绍了如何用 INI 文件制作系统的配置文件，当然，大家也完全可以将其用来作为微型数据库使用，只不过在数据制作上不太方便罢了。

下面我们来介绍下 TXT 文件的读写。（理论上讲，只要可以进行读写的对象就都可以用来制作数据库）在使用 TXT 文件来制作数据库的时候有一个问题，大家都知道 TXT 只能用来存放文本，也就是字符串。那么为了实现数据库的功能，我们必须能够读取想整数，小数等数据，怎么才能实现呢？要知道，在 TXT 中，哪怕一个小数点也是被当做一个字符来处理的。所以，如果要读取 TXT 中的非字符数据，必须使用强制转换，同时，在制作数据库的时候，对该 TXT 必须设定固定的格式。下面我们就一起来制作一个简单的基于 TXT 文件的微型数据库。

制作该数据库必须对 TXT 文件的读写操作有一定的了解。

首先我们来建立一个 TXT 文件，大致内容如下：



```

文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)
下面是我们的数据
比如说直径的值一共有如下几个系列
1  2  3  4  5  6  7  8  |  每行数据末尾以|结束
*****//文件末尾以*号结束

```

图 9-3

按照以上的格式，我们使用下面的代码来读取数据。

```

// 读取文件中第 ROW 行里的前 10 个数据，假设每个数据（字符串的长度）不能超过 15，最后将这些数据存放到数组中备用
void ReaLine(CString file_name, int row, CString Value[10])
{
    CFile openfile;
    char ch = '$'; //以该特殊符号作为基本比较对象
    int T = 0;

```

```

char str[17] = {0};
int i = 0, count = 0;
if(openfile.Open(file_name,CFile::modeRead|CFile::shareDenyNone))
{
    openfile.SeekToBegin();
    while(ch != '*')
    {
        openfile.Read(&ch,1);
        if(ch == '\n')
            count ++;
    }
    if(row > count)
        return;
//以上代码计算了这个文件的总长度，并进行了简单的错误排查。
// 下面将光标定位到指定的行
//-----
    int index = 1;
    char Text;
    openfile.SeekToBegin();
    while(index < row)
    {
        openfile.Read(&Text, 1);
        if(Text == '\n')
            index ++;
    }
// 从该行开始读取数据
//-----
    ch = '$';
    while (ch != '|')
    {
        openfile.Read(&ch, 1);
        if(ch != ' ')
        {
            if(i == 16) //如果长度太大则溢出，错误。
            {
                AfxMessageBox("err, each data must be smaller than 15 chars");
                openfile.Close();
                return;
            }
            str[i];
            i++;
        }
        if((str[0] != 0) && (ch == ' ')) //当读取到空格，则将前面的数据转移
        {

```

```

        if(T > 9)
        {
            openfile.Close();
            return;
        }
        Value[T] = (CString)str;
        T++;
        for(int temp = 0; temp < 16; temp++)
        {
            str[temp] = 0;
        }
        i = 0;
    }

    openfile.Close();
}

else
    AfxMessageBox("failed to open the file");
}

```

以上便是读取 TXT 文件的关键代码，在这段代码中实现了读取文件中某行的前 10 个数据，当然，大家可以将此代码修改为读取更多的数据。需要注意的是，读取出来的数据全部都是字符串，不过幸好我们有强制转化功能，通过 `atof`, `atoi` 等函数，可以将我们的数据转化为相应的小数或整数。另外，在制作数据库的时候，其格式可按照自己的喜好自行设置。（这需要与程序中读取方式一致）。

在以上的代码中，我们只是实现了数据的读取，要修改和维护这个数据库需要自己按照文本的方式修改。（笔者目前并没有实现通过准确定位来修改其内容），所以修改的时候都是在程序外使用普通方法修改。该数据库的最大用途就是可以给我们的模型提供一些默认的值。（用户可以通过修改默认值来达到修改模型的目的。从这个意义上来说，其实已经涉及不到对该文件的写操作了。）

下一个话题中，我们将看到可读可写的文件操作：`excel` 和 `access`。

## 用 EXCEL 文件制作微型数据库

在前面的话题中，我们看到了两种文件操作，`INI` 和 `TXT`，在这两中文件中，对于数据的操作都不是很方便，在这里我们给大家介绍一种方便读写的数据形式，这也是常用的一种数据存储方式。

`Excel` 的用途非常大，在这里，我们仅仅考虑其数据存储功能。对于 `excel` 表的操作，网上有很多的介绍，大家可以去搜索。这里给大家介绍一个比较有用

的类：CspreadSheet，由于该类内容太多，若在本书中将代码贴出来极不方便，有需要的朋友可从其网站上下载。

此处笔者只将该类的使用方法贴出来，通过查看该类的方法，大家应该也能够将该类的功能窥知一二。

```
void CExcelAccessDlg::OnOK()
{
    CSpreadSheet SS("c:\\Test.xls", "TestSheet");
    CStringArray Rows, Column;
    //清空列表框
    m_AccessList.ResetContent();
    for (int i = 1; i <= SS.GetTotalRows(); i++)
    {
        // 读取一行
        SS.ReadRow(Rows, i);
        CString strContents = "";
        for (int j = 1; j <= Rows.GetSize(); j++)
        {
            if(j == 1)
                strContents = Rows.GetAt(j-1);
            else
                strContents = strContents + " --> " + Rows.GetAt(j-1);
        }

        m_AccessList.AddString(strContents);
    }
}

void CExcelAccessDlg::OnWriteexcel()
{
    // 新建Excel文件名及路径，TestSheet为内部表名
    CSpreadSheet SS("c:\\Test.xls", "TestSheet");
    CStringArray sampleArray, testRow;
    SS.BeginTransaction();
    // 加入标题
    sampleArray.RemoveAll();
    sampleArray.Add("姓名");
    sampleArray.Add("年龄");
    SS.AddHeaders(sampleArray);
    // 加入数据
    CString strName[] = {"A", "B", "C", "D", "E"};
    CString strAge[] = {"27", "23", "28", "27", "26"};
    for(int i = 0; i < sizeof(strName)/sizeof(CString); i++)
    {
```

```

        sampleArray.RemoveAll();
        sampleArray.Add(strName[i]);
        sampleArray.Add(strAge[i]);
        SS.AddRow(sampleArray);
    }
    // 初始化测试行数据，进行添加、插入及替换数据操作演示
    for (int k = 1; k <= 2; k++)
    {
        testRow.Add("Test");
    }
    SS.AddRow(testRow);           // 添加到尾部
    SS.AddRow(testRow, 2);       // 插入新行到第二行
    SS.AddRow(testRow, 6, true);  // 替换原第四行来新的内容
    SS.Commit();
    if(m_Check.GetCheck())
        SS.Convert(";");         // 将原Excel文件转换为用分号分隔的文本，并另存为同名文本
文件

    AfxMessageBox("文件写入成功!");
}
void CExcelAccessDlg::OnQuery()
{
    CSpreadSheet SS("c:\\Test.xls", "TestSheet");

    CStringArray Rows, Column;
    CString tempString = "";

    UpdateData();

    if(m_strRow == "" && m_strColumn == "")           // 查询为空
    {
        AfxMessageBox("行号、列号不能同时为空!");
        return;
    }
    else if(m_strRow == "" && m_strColumn != "")       // 查询指定列数据
    {
        int iColumn = atoi(m_strColumn);
        int iCols = SS.GetTotalColumns();
        if(iColumn > iCols)                           // 超出表范围查询时
        {
            CString str;
            str.Format("表中总列数为: %d, ", iCols);
            AfxMessageBox(str + " 查询列数大于Excel表中总列数，请重新输入!");
            return;
        }
    }
}

```

```

    }

    // 读取一列数据，并按行读出
    if(!SS.ReadColumn(Column, iColumn))
    {
        AfxMessageBox(SS.GetLastError());
        return;
    }

    CString tmpStr;
    for (int i = 0; i < Column.GetSize(); i++)
    {
        tmpStr.Format("行号: %d, 列号: %d , 内容: %s\n", i+1, iColumn, Column.GetAt(i));
        tempString += tmpStr;
    }

    AfxMessageBox(tempString);
}

else if(m_strRow != "" && m_strColumn == "")    // 查询指定行数数据
{
    int iRow = atoi(m_strRow);
    int iRows = SS.GetTotalRows();

    if(iRow > iRows)                            // 超出表范围查询时
    {
        CString str;
        str.Format("表中总行数为: %d, ", iRows);
        AfxMessageBox(str + " 查询行数大于Excel表中总行数，请重新输入!");
        return;
    }

    // 读取指定行数据
    if(!SS.ReadRow(Rows, iRow))
    {
        AfxMessageBox(SS.GetLastError());
        return;
    }

    CString tmpStr;
    for (int i = 0; i < Rows.GetSize(); i++)
    {
        tmpStr.Format("行号: %d, 列号: %d , 内容: %s\n", iRow, i+1, Rows.GetAt(i));
        tempString += tmpStr;
    }
}

```

```

        AfxMessageBox(tempString);
    }
    else if(m_strRow != "" && m_strColumn != "")    // 查询指定单元格数据
    {
        int iRow = atoi(m_strRow), iColumn = atoi(m_strColumn);
        int iRows = SS.GetTotalRows(), iCols = SS.GetTotalColumns();

        if(iColumn > iCols)    // 超出表范围查询时
        {
            CString str;
            str.Format("表中总列数为: %d, ", iCols);
            AfxMessageBox(str + " 查询列数大于Excel表中总列数, 请重新输入!");
            return;
        }
        else if(iRow > iRows)
        {
            CString str;
            str.Format("表中总行数为: %d, ", iRows);
            AfxMessageBox(str + " 查询行数大于Excel表中总行数, 请重新输入!");
            return;
        }

        // 读取指定行、列单元格数据
        if(!SS.ReadCell(tempString, iColumn, iRow))
        {
            AfxMessageBox(SS.GetLastError());
            return;
        }

        CString str;
        str.Format("行号: %d, 列号: %d, 内容: %s", iRow, iColumn, tempString);
        AfxMessageBox(str);
    }
}

```

通过使用这个类，我们可以方便的对 EXCEL 表中的数据进行读写。读者可以自行研究研究。(该类可以从网上下载，请搜索关键字：CspreadSheet)

## 用 ACCESS 制作微型数据库

在前面我们介绍了“冒牌”数据库，之所以称为“冒牌”是因为我们只是利用它们完成了类似数据库的某些功能而已。正真的数据库绝不是简单的进行读取和修改这么简单。在提供这些功能的时候还要求能够查询，建立库与库的关系，

便于维护性，和高的安全性等等。（当然，这些数据库并不是所有人的计算机上都有安装，从通用性的角度上讲，我们的 INI, TXT, EXCEL 冒牌数据库就起到了用场）

要完善如此多的功能的数据库自然是庞大的，像 MYSQL, SQLSERVER 等都是比较出名的数据库，在这里只作为入门，我们仅仅讲 access 的使用。（笔者对数据库的了解也不深入）

首先请确认读者计算机上安装了 ACCESS 数据库（好像 OFFICE 的默认安装是不会安装这个的，如果是从外面买的那些“GHOST”版 WINDOWS 就更是不可能有这个的了，大家可以去买一个）



新建一个 ACCESS 文件，如：myfirsttest.mdb，为了方便后面的程序使用，最好全部使用英语（或拼音）。

打开后，可以看到如下画面，按照图中所示选取对象后，双击。

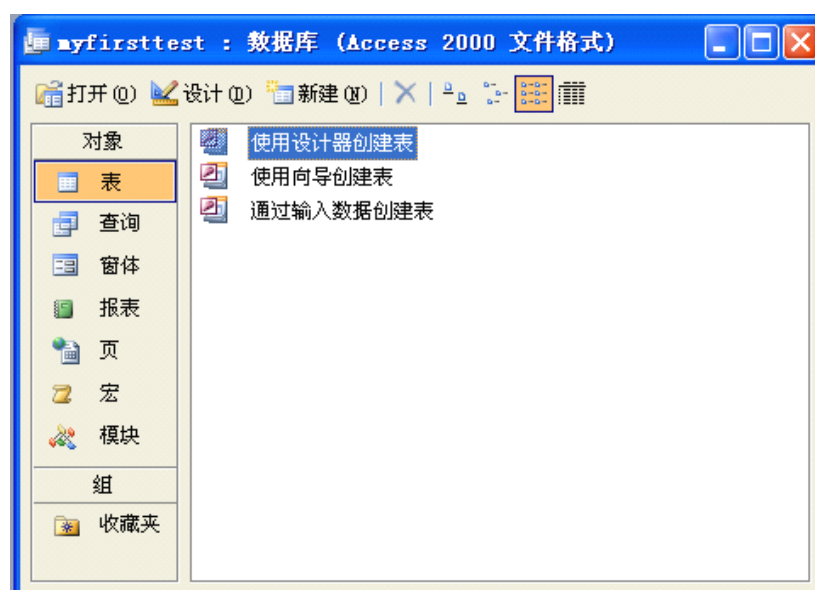


图 9-4

双击“使用设计器创建表” 系统弹出表设计器：





图 9-5

如图，字段名称为关键字，数据类型可以为其制定相关的值。设计完成后点击关闭按钮，系统提示是否保存，选择是。

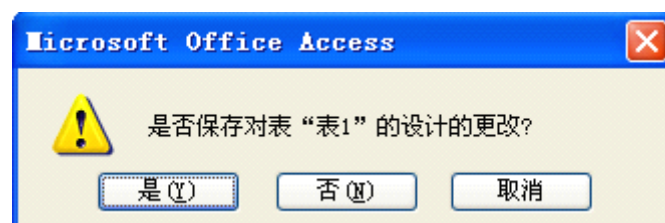


图 9-6

在接下来弹出的对话框中输入表名称：

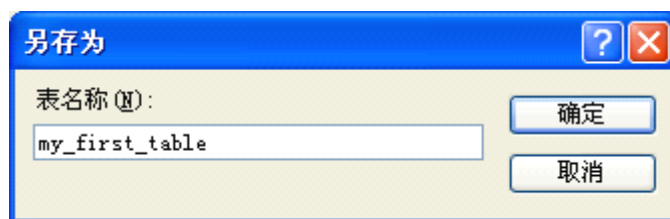


图 9-7

完成后，我们的表的格式便制作完成了，接下来对表进行数据填充：

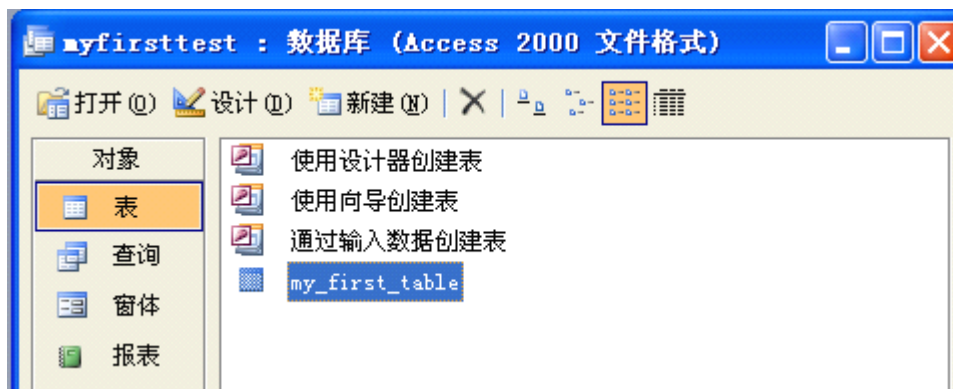


图 9-8

双击我们刚才建立的表格，填充数据：

| my_first_table : 表 |        |      |     |     |       |       |         |
|--------------------|--------|------|-----|-----|-------|-------|---------|
|                    | 编号     | name | age | sex | grade | class | address |
|                    | 1      | 张三   | 25  | 男   | 一年级   | 一班    | 桃花岛     |
|                    | 2      | 李四   | 20  | 女   | 一年级   | 二班    | 灵蛇岛     |
|                    | 3      | 王麻子  | 18  | 女   | 幼儿园   | 中班    | QQ堂     |
|                    | (自动编号) |      | 0   |     |       |       |         |

图 9-10

数据填充完毕后，直接点击退出按钮，系统将自动保存。

到这里，我们便建立了一个自己的单页数据库。对数据库的基本操作也就介绍到此处，读者可以在这里面建立若干张表，还可以在表与表之间建立关系（关系型数据库），如果想更深入的了解数据库的结构与原理，建议大家到书店去买些相关的书籍进行研读。笔者此处只是简单的给大家演示了一下建立数据库所需要的流程。对于数据库的操作语言 SQL 语句，大家也请翻阅相关书籍。

数据库建立好之后，接下来我们要用程序来对其进行读写。同样，由于这种数据库用途极其广泛，所以网络上也有不少精华处理方法，和 EXCEL 一样，笔者也从网络上找了一个笔者认为比较方便的类，然后对这个类做了一些简单的修改（略为修改了构造函数）。由于该类比较简单，因此在此贴出。

代码如下：

```
#ifndef MY_ACCESS_OPERATION_H
#define MY_ACCESS_OPERATION_H

#import "C:\Program Files\Common Files\System\ado\msado15.dll" no_namespace
rename ("EOF", "adoEOF") //黑体字在同一行

class ADOConn
{
```

```

public:
    ADOConn(char* temp_path = "defaultpath");
    virtual ~ADOConn();

    BOOL ExecuteSQL(_bstr_t bstrSQL);
    _RecordsetPtr GetRecordSet(_bstr_t bstrSQL);
    void ExitConnect();
    void OnInitADOConn();
    _ConnectionPtr m_pConnection;
    _RecordsetPtr m_pRecordset;

protected:
    char* path;
}

ADOConn::ADOConn(char* temp_path)
{
    path = temp_path;
}

ADOConn::~~ADOConn()
{
    path = NULL;
}

void ADOConn::OnInitADOConn()
{
    try
    {
        m_pConnection.CreateInstance("ADODB.Connection");
        char* temp = "Provider = Microsoft.jet.OLEDB.4.0;Data Source = ";
        m_pConnection->Open(strcat(temp,path), "", "", adModeUnknown);
    }
    catch(_com_error e)
    {
        {AfxMessageBox(e.Description());}
    }
}

void ADOConn::ExitConnect()
{
    if(m_pRecordset != NULL)
        m_pRecordset->Close();
    m_pConnection->Close();
    path = NULL;
    ::CoUninitialize();
}

```

```

}

_RecordsetPtr ADOConn::GetRecordSet(_bstr_t bstrSQL)
{
    try
    {
        if(m_pConnection == NULL)
            OnInitADOConn();
        m_pRecordset.CreateInstance(__uuidof(Recordset));
        m_pRecordset->Open(bstrSQL,m_pConnection.GetInterfacePtr(), adOpenDynamic,
adLockOptimistic, adCmdText);
    }
    catch(_com_error e)
    {e.Description();}
    return m_pRecordset;
}

BOOL ADOConn::ExecuteSQL(_bstr_t bstrSQL)
{
    _variant_t RecordsAffected;
    try
    {
        if(m_pConnection == NULL)
            OnInitADOConn();
        m_pConnection->Execute(bstrSQL,NULL,adCmdText);
        return true;
    }
    catch(_com_error e)
    {e.Description();}
    return false;
}

}

#endif

```

这个类如果要在 MFC DLL 模式下使用，请将 MFC DLL 中的 StdAfx.H 中与该数据库重复定义的部分注释掉。否则程序会提示重复定义而无法通过编译。

下面介绍下这个类的用法。

请看如下代码：

```

void UserTestAccessOperation()
{

```

```

ADODConn object;
object = ".\\data.mdb";
object.OnInitADODConn(); //初始化,包含连接数据库的过程

CString sql;
sql = "select * from my_first_table where age = “30”";

object.GetRecordSet(_bstr_t(sql)); //获取记录集并将地址传递给 m_pRecordset;

while(object.m_pRecordset->adoEOF == 0) //到了文件末尾则返回零
{
    //获得纪录中 m_pRecordset 所指各列的值
    CString name = (char*)(_bstr_t)object.m_pRecordset->GetCollect("name");

    //纪录集中的 address 列
    CString address = (char*)(_bstr_t)object.m_pRecordset->GetCollect("address");

    object.m_pRecordset->MoveNext(); //指针向后移动一位
}

object.ExitConnect();
}

```

程序首先建立了与数据库文件的连接，然后使用了 SQL 语句对其中的数据进行了查询，并对记录集中的某些数据进行了读取。最后，关闭了连接并退出数据库操作。对于程序中涉及到的 SQL 语句，以及修改，删除等操作，读者可查阅相关书籍，笔者不再详述。（由于这些操作所涉及的代码比较多，详细范例请参考本书最后的工程实例的部分）

## 第十节：工程专题

在前面的章节中，我们学习了一些开发工程所必须用到的工具或方法，在这一节中，我们将以实际的事例来进一步学习二次开发，一方面可以加深认识，另一方面也学习解决问题的方法，由于该部分比较难，所以需要对本节前面所描述的内容完全理解之后再尝试，以免打击积极性。

笔者一直认同一句话“尽信书不如无书”。对于这本书也一样，希望大家抱着一种中立的心态来看待本书中的内容。笔者也不可能保证本书一个错都没有，在遇到有疑问的地方（特别是专题系列中），一方面可以查阅相关的函数以进行考证，另一方面也可以通过邮件或登陆论坛来寻找答案。

### 专题一：建立团队公用（DLL）函数库

经常写程序的人都知道，为了提高代码的重用性，有时候我们会写一些通用的代码，这些代码包括一些常用的类，函数，数据结构体等等，一般来讲，每个团队都会有自己的类库，或者函数库，并且这种库以模块的方式存储在计算机中，以便大家都可以访问，通常情况下，我们把这些库集合到动态链接库（DLL）中。

举个很简单的例子，我们在开发 PROE 的时候，为什么要设置环境呢？比如我用到一个函数： `ProStringToWstring`，那么 VC 并不知道这个函数在哪里，于是，我们便需要告诉 VC，这个函数的头文件，LIB，以及 DLL 都在什么地方。PROE 提供给我们所有函数都存放在 DLL 中，这个在第一节中有讲到过。

废话少说，下面我们就来建立一个我们自己的 DLL 函数库。

建立 DLL 的方法与其它 PROE 工程的过程一样。我们知道，要想一个 DLL 中的函数被外界调用，需要导出该函数，一般而言有两种方法，一种是在 DEF 文件中定义，另外一种方法则是直接在声明函数时，添加宏：

```
extern "C" void __declspec(dllexport) Test(int var);
```

对此宏的理解可以参照 MSDN 上的描述，此处不做过多解释。对于只在 WINDOWS 下应用的函数来讲，通过这种方式产生的 DLL 已经可以直接在其它应用程序中调用了。（这要求 DLL 与应用程序在同一路径下，或在 WINDOWS 能够找到的路径下，大家可以将路径添加在环境变量 PATH 中，这样 WINDOWS 就会在这些路径下去寻找目标文件）

到这里我们遇到了我们的问题，PROE 怎么知道要调用的 DLL 在哪里？我们知道，在之前我们也写了很多 DLL，为了调用这些 DLL 中的功能，我们必须要通过 PROTK.DAT 文件进行注册。

对于我们的函数库来讲，其运行原理是一样的，当然也需要注册。到这里，我们需要注意四个问题： 一是为了注册成功，DLL 中必须要有标准入口函数 `extern "C" int user_initialize()` 和 `extern "C" void user_terminate()`，第二个问题是，通常我们在初始化函数运行后就会挂上菜单，由于只是通用功能库，这里我们是不需要菜单的，所以，初始化函数应当这样写：

```
extern "C" int user_initialize()
{
    Return PRO_TK_NO_ERROR; // 直接返回
}
```

第三个问题是注册文件的写法：

```
NAME SAMPLED
STARTUP dll
EXEC_FILE E:\work\firstsample\Release\firstsample.dll
TEXT_DIR E:\work\firstsample\text
ALLOW_STOP TRUE
REVISION 2008
END
```

由于我们没有菜单，所以不存在菜单描绘会文件，但是，如果我们的注册文件中不写该选项：

```
TEXT_DIR E:\work\firstsample\text
```

那么这个 DLL 将不会注册成功，那么怎么办呢？办法是有的，例如我们要在 DLLA 中调用了标准函数库 DLLB 中的内容，那么我们在写 DLLB 的注册文件时所指定的 菜单描绘 TXT 文件时可以直接指向 DLLA 中的该文件。

最后一个问题是注册文件的顺序问题，仍旧使用上面的例子，它的注册顺序应该是 DLLB 在 DLLA 的前面。

下面结合一个实际的例子来说明这些情况：

首先按照前面介绍的方法建立一个 MFC (也可以是一个 WIN32，不过需要包含 windows.h) 的 DLL，系统会自动为我们生成一些文件，不要管它，直接在工程中添加 UserLib.h 和 UserLib.cpp 文件，如下：

```
// DLL 库
// UserLib.h
extern "C" int user_initialize(); 默认入口

extern "C" void user_terminate(); 默认退出函数
```

```
extern "C" void __declspec(DLLexport) Test(); 所要导出的函数
```

```
// UserLib.cpp
extern "C" int user_initialize()
{
    Return PRO_TK_NO_ERROR; // 直接返回
}

extern "C" void user_terminate()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
}

extern "C" void __declspec(DLLexport) Test(int var)
{
    ProMdl mdl;
    ProMdlType type;
    if(ProMdlCurrentGet(&mdl) == PRO_TK_NO_ERROR)
    {
        ProMdlTypeGet(mdl, &type);
        if(type == PRO_MDL_PART || type == PRO_MDL_ASSEMBLY)
            AfxMessageBox ("model find !!");
        else
            AfxMessageBox ("model not find !!");
    }
}
```

如果环境设置正确的话，整个工程可以成功编译，并产生一个 DLL，这里假设我们的这个 DLL 名称为 UserLib.dll，与该 DLL 同样重要的文件还有：UserLib.lib 以及我们所创建的 UserLib.H，这三个文件中，UserLib.lib 和 UserLib.H 是我们在创建其它 DLL 的时候，编译所必须用到的文件，而 UserLib.dll 文件则是在 PROE 中运行所需要的文件。

现在假设我们把这三个文件存放在 E:\work\UserLib 中。

下面我们创建一个 UserTestLib.dll

由于在这个 DLL 中，我们要使用到 UserLib 中的功能，所以，必须要告诉 VC 编译器，我们所使用的函数的头文件，以及库文件的路径。此添加方法与前面的一样。

依次在编译界面中选择 工具 ---- 选项，添加头文件和库文件路径：





图 10 - 1

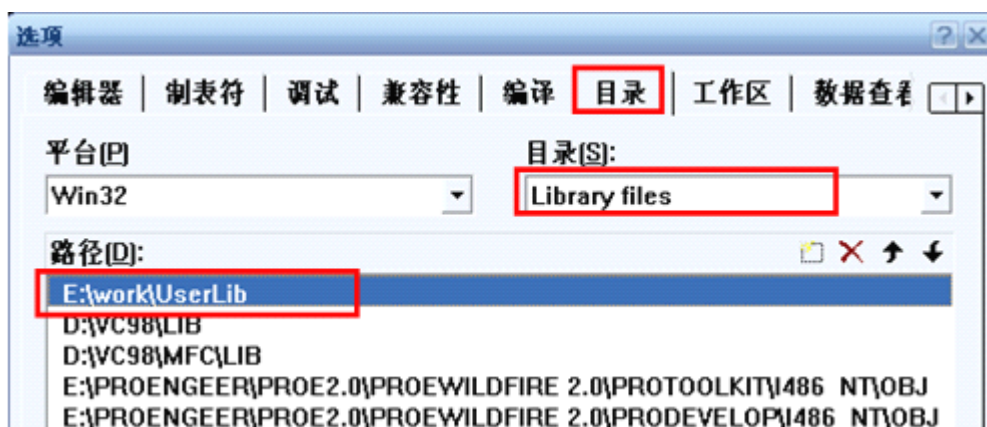
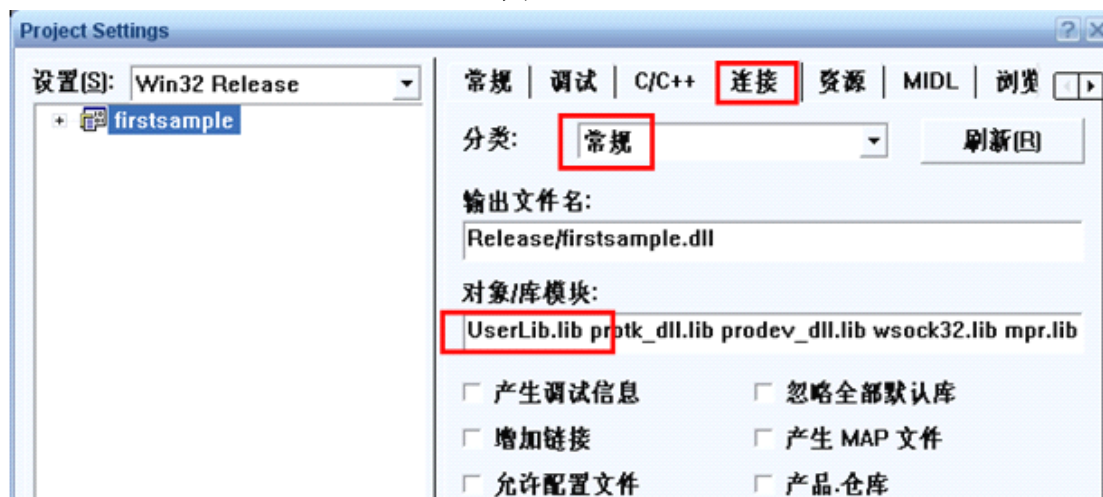


图 10- 2

然后点击 工程 – 设置 在其中设置使用到的库文件:

图 10 - 3



这样，我们便设置好了环境，大家可以看到，我们的设置方式与调用 PROE 中函数的方式是一样的。

现在，我们按照普通的方法建立一个 MFC 的 DLL 样本，然后在这个样本的菜单功能中，我们就可以直接使用 UserLib 中的函数：Test() 了。当然，在使用该函数前，必须包含 UserLib.h 头文件。

通过编译之后，我们得到了 UserTestLib.dll 文件。现在我们需要在 PROE 中注册，并运行该 DLL。注册文件按照如下方式来写：

```
NAME UserLib // 必须先注册 库
STARTUP dll
EXEC_FILE E:\work\ UserLib \ UserLib.dll
TEXT_DIR E:\work\ UserTestLib \text // 此处与 UserTestLib 一致
ALLOW_STOP TRUE
REVISION 2008
END

NAME UserTestLib // 然后注册库的调用者
STARTUP dll
EXEC_FILE E:\work\ UserTestLib \Release\ UserTestLib.dll
TEXT_DIR E:\work\ UserTestLib \text
ALLOW_STOP TRUE
REVISION 2008
END
```

这样，PROE 便会按照先后顺序将这两个 DLL 分别注册进入内存中，所以，当 UserTestLib 调用 UserLib 中的函数时，才能够正确执行。大家可以在这个基础上去建立更多的标准函数库。

随着我们的函数库一天天的庞大，函数帮助文档是必不可少的，现在有很多方可以用来建立帮助文件，笔者在这里给大家介绍一种纯手工打造的帮助文件的制作方法（当然，可能有些落后了，不过大家可以结合程序按照如下方法来自动生成所需要的文档）

下面我们就一起来用 TXT 文件制作基于 HTML 的帮助文档（所需的 JAVASCRIPT 知识和 CSS 的知识并不多，大家若有兴趣，可以参考其它书籍）假设我们做完的帮助文件外观如下：  
（为了简单起见，我们做得非常简陋）



图 10 - 4

在这个界面中，IE 浏览器所浏览的页面分成了三份，顶部用来描述该文档的用途，左侧是函数清单，右侧是函数查询界面，当用户点击左侧的某个函数后，右侧将该函数的详细说明文档显示在右侧。（为了让界面元素更加丰富，大家可以在熟悉这种结构之后在添加一些按钮之类的控件，这里就不详细说明了）

构建上面的内容一共需要 6 个文件，他们分别是：  
 主描述页面： index.html                      顶部页面： HEAD.html  
 左侧页面： LEFT\_LINK.html                  右侧页面： SHOW.HTML  
 函数说明页面： your\_function.html        背景图片： test.jpg

下面我们就来依次建立这几个文件。

为了把这些页面放在同一个文件夹中，我们最好新建一个文件夹，在其中新建一个 index.TXT 文件，在其中写下如下内容：

```
<HTML>
<HEAD>
<TITLE>
    HELP FILES
</TITLE>
</HEAD>

<FRAMESET ROWS = "12%, 88%">
```

```

    <FRAME NAME = "myhead" SRC = "HEAD.HTML" SCROLLING = "no" noresize>
    <FRAMESET COLS = "15%, 85%">
    <FRAME NAME = "myleft" SRC = "LEFT_LINK.HTML" SCROLLING = "YES">
    <FRAME NAME = "myright" SRC = "SHOW.HTML" SCROLLING = "YES">
    </FRAMESET>
</FRAMESET>
</HTML>

```

该文件描述了如何将一个页面分成三份，每份分别叫什么名字，以及都放些什么文件。该文件相信大家在理解上都不存在问题吧。然后我们把该文件保存，并修改后缀名为 HTML ， 这样我们就得到了第一个文件。

按照相同的方法，我们依次建立其它几个文件。

HEAD.html :

```

<HTML>
<HEAD>
<TITLE>
    HELP FILES
</TITLE>
</HEAD>

<BODY>
    <h2><PRE>

                                公共函数库使用帮助文档
                                -----

    ---- 王伟</h2>
</BODY>
</HTML>

```

LEFT\_LINK.html

```

<HTML>
<HEAD>
<TITLE>
    HELP FILES
</TITLE>

<SCRIPT language = javascript>

function UserLoadPage ()

```

```

{
    var the_path = arguments[0] + ".html";
    parent.myright.location.reload (the_path);
}

</SCRIPT>

</HEAD>

<BODY bgColor = gray>

<font size = 4 color = blue>
    <b>FUNCTIONS LIST</b>
</font><BR>

<A href = "#">-----
</A><BR>

<A href = "#" onclick =
"UserLoadPage('your_function')">your_function</A><BR>

</BODY>
</HTML>

```

SHOW.HTML :

```

<html>
<title>
    help files
</title>

<style type = "text/css">
body {
    color = red;
    font-weight:bold;
    background-image:url("test.jpg");
    background-repeat: repeat;
    background-attachment:fixed;
    background-position:center;}
</style>

```

```
</head>

<body bgColor = orange>
<h1>
<PRE>

您可以在此处写下关于该文档的一些描述，如：

注意： 本手册为开发人员专用

大家在使用的过程中若发现任何 BUG

请即时与手册维护人员联系。。。

</h1>
</body>
</html>
```

your\_function.html :

```
<HTML>
<HEAD>
<TITLE>
    HELP FILES
</TITLE>

<BODY>

<PRE>

    FUNCTION:  void your_function ()
    DESCRIBE:  this function is for test only
    INPUTS   :  no
    RETURN   :  no

    DEMON    :  no

</BODY>
</HTML>
```

最后是背景图片，当然这个就随便大家用什么图片了。

所有文件建立完成后，双击 index.html 文件，就能看到效果了。大家可以在此基础上添加一些其它元素或者通过程序来自动建立这些文件。笔者此处就不详述了。

## 专题二：建立元件库（标准件库）

元件库也叫做标准件，但凡是做装配设计的，基本上都要用到标准件，不管是做模具也好，做电子也好，基本上都有自己的行业标准，所以，标准件其实是一个用途很广泛的东西，作为一个 PROE 二次开发设计人员，标准件的客制化及其制作是一项基本技能。

总的来讲，标准件分为两步：建模 以及 编程。但是在做标准件之前，最好是把标准件的调用思路理清楚。如果希望自己制作的程序可以供别人客制化来用，那么还要考虑数据库的问题。（PROE EMX 用的是 TXT 数据库，NX 基本上用到的是 EXCEL 数据库）。然后就是程序的界面，一个好的使用界面能使使用者很快上手，减少培训费用。

当然，如果严格要求的话，需要注意到的地方还有很多，为了简单起见，我们这里只用一个例子，大家可以在理解了基本思路和创作流程之后，根据自己的实际需要来扩展。

模型假设：现在，假设我们要做一个标准件，这个标准件是一个 PART 档案（关于组立形式的标准件比较复杂，为了简单，我们以 PART 举例，在了解了 PART 这种形式之后，相信 ASM 档案类型的也就迎刃而解了）。那么，该 PART 档案中除了要有实体之外，还需要有一个用于“切割”的面，当该 PART 装配到一个组立中时，我们便可以通过这个“切割”面来切掉组立中与该 PART 相互干涉的其它模型。另外，一般来讲，标准件还会有相应的工程图对应，所以，这个也应该作为标准件的一部分考虑到其中。

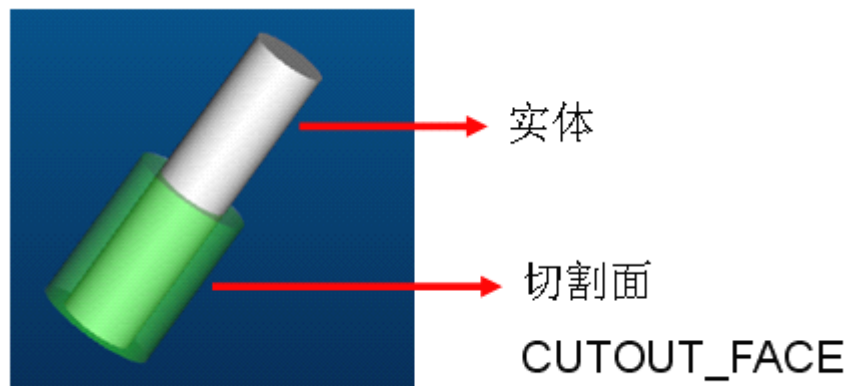
在建立模型的过程中，我们还要考虑，该标准件中那些尺寸是需要变化的，因为标准件不可能只有一个，往往有一系列，所以我们需要把经常随设计变化的那些尺寸标记下来，做到数据库中，这样，便可在调用标准件的界面中让用户根据需要来选择对应的型号。

在这一个专题中，我们将会用到本书前面的一些知识，也可以说是一个综合性的练习了，根据先后顺序，在本练习中会用到的知识有：UDF 的制作，模型的自动装配，组立中 UDF 的调用，模型与工程图的改名，读取数据库，以及模型参数化驱动。

废话少说，下面我们按照先后顺序，一步一步往下做。

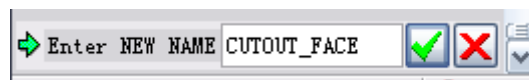
第一步：建立模型。

在 PROE 中以坐标为中心，建立一个实体模型，形状大家可以自己定，为了简单起见，本例中的模型如下：



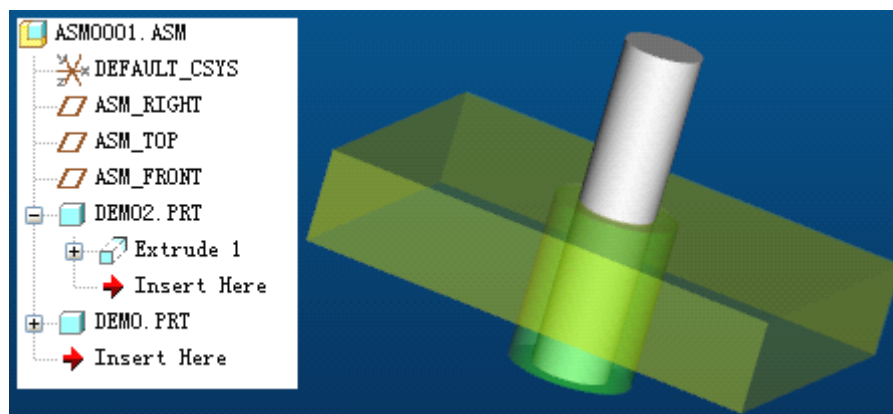
其中，白色部分为实体，半透明的部分为一个封闭的面（要用于切割所以必须是封闭面），等该实体装配好后，我们将使用该面来切割组立中的其它实体。我们将该面命名为 CUTOUT\_FACE。

命名方法如下：依次点击：EDIT → SETUP → NAME → OTHER



然后在图形窗口选择该面，在提示输入框中输入名称，然后点击右侧的小勾，这样便完成了，最后点击 DONE 退出浮动菜单。

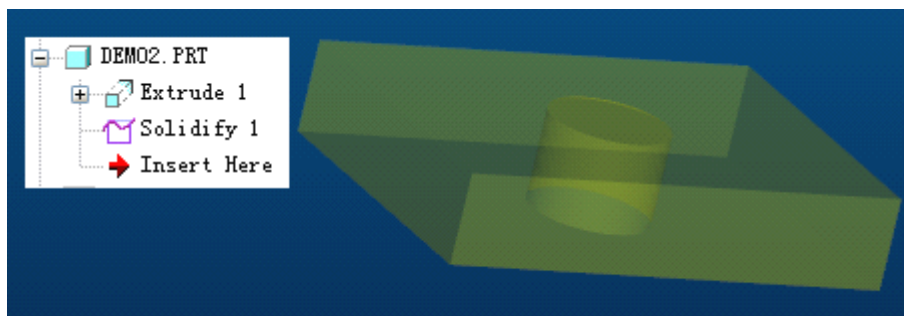
元件我们就初略的制作到这里，下面我们来建立用于切割实体的 UDF 。首先建立一个装配档案。并将该元件装配到组立中。如图：



接下来我们在 DEMO2（长方体） 中来建立切割特征。

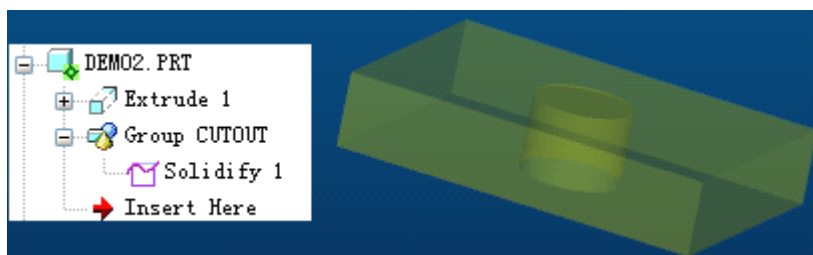
激活 DEMO2 模型，选中圆柱体的切割面，点击编辑中的 SOLIDIFY 命令，选择正确的方向之后，将自己切掉，切掉后如图：





这时，我们需要将这个 SOLIDIFY 命令制作成我们的 UDF，由于该特征是在组立中创建的，所以需要在组立中来建立这个 UDF。

仍旧保持 DEMO2 的激活状态，我们依次点击：TOOLS → UDF LIBRARY 在弹出的浮动菜单中选择创建，在提示栏输入名称：CUTOUT。然后系统询问这个 UDF 要不要将该 UDF 与该 UDF 所参照的 PART 一起绑定，我们这里选择 STAND ALONE，意思是仅仅创建这个 UDF 就可以了。然后点击 DONE。系统接下来会提示选择 UDF 所要包含的特征，选择切割特征后，我们还可以给这个特征选择可变尺寸，这里我们不选择尺寸，用于固定切割（对于可变尺寸的 UDF 切割，我在螺丝系统中会讲到）。需要注意的是，在 UDF 创建完成之后，最好自己通过 INSERT → UserDefinedFeature 手动测试一下该 UDF 是否正确。下图是本例的测试结果。



UDF 创建完成之后，我们的准备工作就做完了，下面就该代码上场了。

首先设定文件结构，我们假设组立档案在工作区 E:\work 下，而我们的元件库和 UDF 在另外一个地方：E:\work\Resource 下。那么我们要做的第一件事情就是将模型从资源区载入到工作区，同时给标准件修改名称以应用到当前的工作中来，然后通过坐标系装配到当前的组立中。

代码如下：

```
void LoadAndRename (char libpath, char oldname, char newname)
{
    //设置选项后，可将内存中的模型存储在当前工作路径中
    ProName option;
    ProPath option_value;
```

```

ProStringTowstring(option, "override_store_back");
ProStringTowstring(option_value, "yes");
ProConfigoptSet(option, option_value);

// 初始化数据
ProError status ;
ProMdl oldhandle;
ProPath w_libpath, w_oldname, w_newname;
ProStringTowstring(w_libpath, libpath);
ProStringTowstring(w_oldname, oldname);
ProStringTowstring(w_newname, newname);
ProwstringConcatenate(w_oldname, w_libpath, PRO_VALUE_UNUSED);
ProwstringConcatenate(L".prt", w_libpath, PRO_VALUE_UNUSED);

//载入模型
ProMdlLoad(w_libpath, PRO_MDL_UNUSED, PRO_B_FALSE, &oldhandle);

// 准备载入图档
ProPath drawpath;
ProStringTowstring(drawpath, libpath);
ProwstringConcatenate(w_oldname, drawpath, PRO_VALUE_UNUSED);
ProwstringConcatenate(L".drw", drawpath, PRO_VALUE_UNUSED); //连接图档完整路径

ProMdl drawmdl; //载入图档
status = ProMdlLoad(drawpath, PRO_MDL_DRAWING, PRO_B_FALSE, &drawmdl);
if(status == PRO_TK_NO_ERROR)
{
    ProMdlRename(oldhandle, w_newname); // 给模型改名
    ProMdlRename(drawmdl, w_newname); // 给图档改名
    ProMdlSave(drawmdl); // 先保存图档
    ProMdlSave(childmdl); // 再保存模型
}

//还原选项值
ProStringTowstring(option_value, "no");
ProConfigoptSet(option, option_value);
}

```

经过上面的过程，我们已经将远程资源载入到了当前工作磁盘，并且名称也做了修改，下面就可以讲这个元件装配到当前组立中了。（还有另外一种方法，就是先不把模型载入到当前磁盘，而直接从远程载入到内存中并改名，当用户点击菜单进行保存的时候才把改名后的这个模型保存到磁盘上）

装配过程比较简单，尤其是像这种以坐标系对坐标系的装配，大家可以参照

我在前面章节中关于装配的例子。

```
ProError UserAssembleByCSYS (ProAssembly asm_model, // 输入组立
                             ProSolid comp_model,   // 输入装配目标
                             ProAsmcomp asmcomp)    // 输出装配后的原件
{
    ProError status;
    ProName comp_CSYS ;
    ProMatrix identity_matrix = {{ 1.0, 0.0, 0.0, 0.0 },
                                {0.0, 1.0, 0.0, 0.0},
                                {0.0, 0.0, 1.0, 0.0},
                                {0.0, 0.0, 0.0, 1.0}};

    ProAsmcompconstraint* constraints;
    ProAsmcompconstraint constraint;
    int i;
    ProBoolean interact_flag = PRO_B_FALSE;
    ProModelitem asm_REF, comp_REF;
    ProSelection asm_sel, comp_sel;
    ProAsmcomppath comp_path;
    ProIdTable c_id_table;
    c_id_table [0] = -1;

    //Set up the CSYS names
    ProStringTowstring (comp_CSYS , "LIB_CSYS");

    //Package the component initially
    ProAsmcompAssemble (asm_model, comp_model, identity_matrix,
&asmcomp);

    //Prepare the constraints array

    ProArrayAlloc (0, sizeof (ProAsmcompconstraint), 1,
                   (ProArray*)&constraints);

    //Find the component CSYS
    status = ProModelitemByNameInit (comp_model, PRO_CSYS,
                                     comp_CSYS , &comp_REF);
    if (status != PRO_TK_NO_ERROR)
    {
        interact_flag = PRO_B_TRUE;
        continue;
    }
}
```

```

//Allocate the references
ProSelectionAlloc (NULL, &comp_REF, &comp_sel);

int n_sel = 0;
ProSelection *sel;
tatus = ProSelect ("CSYS", 1, NULL, NULL, NULL, NULL, &sel, &n_sel);
if (status!= PRO_TK_NO_ERROR || n_sel < 0)
    return (PRO_TK_USER_ABORT);
ProSelectionCopy(sel[0], &asm_sel);

//Allocate and fill the constraint.
ProAsmcompconstraintAlloc (&constraint);

ProAsmcompconstraintTypeSet (constraint, PRO_ASM_CSYS);
ProAsmcompconstraintAsmreferenceSet (constraint, asm_sel,
                                     PRO_DATUM_SIDE_YELLOW);
ProAsmcompconstraintCompreferenceSet (constraint, comp_sel,
                                     PRO_DATUM_SIDE_YELLOW);
ProArrayObjectAdd ((ProArray*)&constraints, -1, 1, &constraint);

status = ProAsmcompConstraintsSet (NULL, &asmcomp, constraints);

ProSolidRegenerate ((ProSolid)asmcomp.owner, PRO_REGEN_CAN_FIX);

if (interact_flag)
{
    ProAsmcompConstrRedefUI (&asmcomp);
}

return (PRO_TK_NO_ERROR);
}

```

在该函数中我们输出了装配后的组件， 该组件在后面可以用来轻松的获取 **COMPPATH**

装配完成后，我们接下来需要调用 **UDF** 来切掉与本原件相互干涉的所有目标原件。这里涉及到前面讲过的两个知识点，一个是找出所有与本原件相互干涉的其它元件，第二个知识点是如何在组立环境中正确的调用 **UDF** 特征。

由于这两个知识点在前面都已经讲过了，大家可以返回到前面去复习相关应用。这里我只把整个过程介绍一下。

在我们完成装配的时候，输出了装配后的元件：**asmcomp**， 根据这个元件

的 ID 我们可以使用 ProAsmcomppathInit 来初始化 asmcomppath，然后根据前面章节中讲到的干涉函数，以该 comppath 为参数，找出所有干涉的 comppath，在本例中，我们只能找到 DEMO2 这个元件。

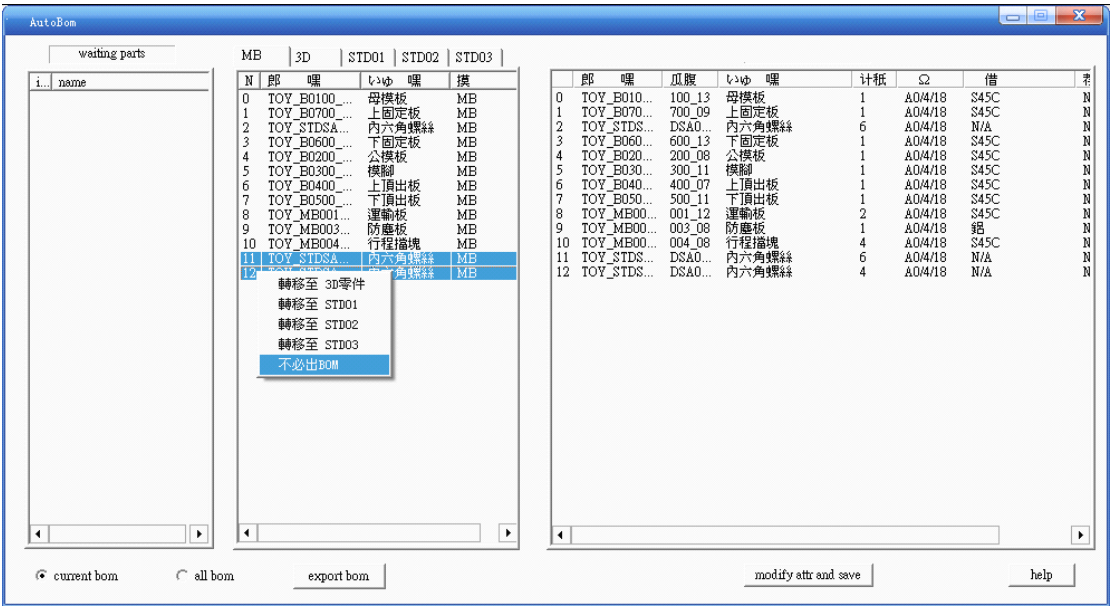
接下来对该元件调用 UDF，首先根据 comppath 获取到 DEMO2 在组立中的模型句柄 MDL，然后在该 MDL 上使用组立中调用 UDF 的方法。在调用 UDF 的时候，注意通过 comppath 来选择标准件上的切割面。具体方法请参见第六节的相关章节（82 页），此处不再详述。

最后要注意的是，对于元件库来讲，其中的元件不可能只调用一次，所以，在实际应用的时候要考虑到元件在同一组立中被多次调用的情况，这种情况又分为重复装配（名称一样），和流水号装配（名称不一样），大家可自行考虑。

### 专题三：BOM 表系统

BOM 表是一个非常重要的需求，但与其他程序相比，它的实现原理其实是比较简单的一种。只不过过程比较繁杂而已。要想写好 BOM 表程序，首先要与使用 BOM 表的客户进行比较多的沟通。其实 BOM 表本生只是简单的与 PROE 中的模型进行参数交互，以此来给元件进行分类。复杂点的也就是所谓的差异 BOM，其实只要做好前期检讨，BOM 就能够很轻松的做出来。另外，由于一个组立档案可能会很大，所以如何高效的读取数据就显得很重要。除了这两点之外剩下的就是与第三方软件交互了（比如导出 EXCEL 之类），这要看对方有什么样的需求。

例如如下的 BOM 界面：（界面中的乱码是字符集的问题，此处只讨论 BOM 原理）



如上便是一个 BOM 的基本构造，当然随着要求的不同，界面可能大相径庭，

这里只看看一般的过程。BOM 也就是清单，拿这张图片上得清单来讲，我们可以注意到中间的标签页，也就是说，该程序将 ASM 中的所有元件按照一定的规则分成了若干等分，产生出了不同需求的表单，表单中的元素可以互相流通（从右键菜单中可以看到表 A 的元素可以转移到表 B 中去），界面右边便是当前表单中所有元素的一些相关参数，当然这些参数要能够在界面中进行修改，并反映到最终的模型中去，另外，我们还需要对这些清单中的元素按照某种规则进行排序（如果有必要的话）。

BOM 表本身并不难，难的是需求的分析，即：客户要求怎样的一个 BOM 清单。当需求分析搞清楚之后，剩下的便是根据游戏规则对 ASM 中的零件进行筛选而已。一般来讲，筛选的规则就是参数设置。所以，利用前面参数的那一章的内容基本上就可以搞定这样的需求了。

由于本功能需要的代码比较多，所以详细设计流程以及代码部分就不在此处给出，有需要的朋友可以通过邮件向本人索取。

## 专题四：关于程序的移植性与有效期

对于我们开发的程序来讲，到目前为止，基本上都是在 VS 中建立的 MFC 工程，这样的工程对于小的程序没有什么问题，但如果我们忽然某天发现我们需要对所有的功能进行一个整合，问题就来了，如此多的工程，我们要怎么办才好呢？

一种方法就是，在新的工程中包含之前的工程，然后将新的工程编译一次。比如说在工程 A 中有一个对话框，但是新工程中却没有，于是便需要将该对话框资源以及相关的程序文件拷贝到新工程中，进行编译。这是一件比较麻烦的事情。

另外一种方法就是在做每一个工程的时候都将程序的移植考虑到其中，这样当最后真的需要移植的时候，就很方便了。

细心的开发人员可能会发现，在 TOOLKIT 提供的帮助程序中，其编译资源都只有源代码与头文件，（使用 MAKEFILE 进行编译的），这就是很好的一个方法，开发人员只需要将这些源文件进行修改就可以了，而且这些单独的源文件与头文件可以随意拷贝到任何地方。当然，这里我并不打算使用 MAKEFILE 来编译，也不在这里讲 MAKEFILE 到底是怎么回事（有兴趣的朋友可以从网上找相关资料进行学习）。

我们在这里使用另外一种方法，我们可以试着用纯粹的代码来制作对话框，以及其中的消息响应函数。这样，如果将来需要转移的话，我们直接将源文件与头文件拷贝到其它地方即可。再也不用担心 VC 工程中 RES 的问题了。当然，使用纯粹的代码编写对话框刚开始不是一件讨好的事情，但是随着熟练程度的提高以及对界面的理解，相信大家在熟悉之后就会喜欢上这样的方式了。

下面就是利用纯代码创建的一个对话框, 入口函数为: `main()`, 相信大家都知道该怎么用吧? (大家第一次学习 C++ 应该就是从 `main()` 开始的吧)。我先使用代码创建了一个对话框, 然后在 `main()` 中进行了调用。当然, 该对话框也可以通过别的方式来调用, 这里主要是演示这个过程。以下是代码部分, 您可以直接拷贝到您的程序中进行测试。

```
// test.cpp : Defines the entry point for the application.
//需要在设置中支持 MFC , 并将程序发布为 release 模式
#include <afxcmn.h>
// 本例演示了如何使用 类 来创建一个对话框应用程序
// 本例中使用纯代码建立了一个对话框
// 当进入程序入口后, 首先利用 模板结构体 建立了一个空对话框容器
// 然后, 在该对话框中建立了 两个按钮 和一个 列表控件
// 为了获取列表控件中的鼠标信息, 我们派生了一个控件并做了简单处理
// 最后, 建立了一个对话框类来响应该对话框中的各种消息。

// class TESTLIST : public CListCtrl
// 该类 TESTLIST 仅仅演示如果获取 子控件中的一些消息, 如鼠标消息
// 因为这类消息通常在父窗口中不好拿, (父窗口还有自己的鼠标消息)
// 所以, 一般要在子空间中自行处理, 或者额外发送通知给主窗口
class TESTLIST : public CListCtrl
{
protected:
    afx_msg void OnLButtonDown(UINT nFlag, CPoint pt);
    DECLARE_MESSAGE_MAP()
};

void TESTLIST::OnLButtonDown (UINT nFlag, CPoint pt)
{
    CListCtrl::OnLButtonDown( nFlag, pt );
    AfxMessageBox (" the left button is down in listwindow ");
    // 如果要想父窗口获取类似这种消息, 可用如下方法发送
    // GetParent()->SendMessage(WM_USER_MSG, GetDlgCtrlID(), NM_CLICK);
}

BEGIN_MESSAGE_MAP(TESTLIST, CListCtrl)
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()

// class class MYDLG : public CDialog
// DEFINE RESOURCE

#define IDD_BUTTON_1    50
```

```

#define IDD_BUTTON_2    51
#define IDD_LIST_CTRLBOX 52

// 为模板所创建的 对话框类
class MYDLG : public CDialog
{
public:
    TESTLIST m_list;    // 列表控件，使用了自定义的那个列表，从而可以获取鼠标消息

protected:

    virtual void OnCancel();    // 响应对话框退出消息
    afx_msg void OnButton1();
    afx_msg void OnButton2();
    afx_msg void OnClickList1(NMHDR* pNMHDR, LRESULT* pResult); // 响应列表框的单击消息
    DECLARE_MESSAGE_MAP()    // 声明本窗体将使用消息映射
};

BEGIN_MESSAGE_MAP(MYDLG, CDialog) // 将 CDialog 中的消息映射到 MYDLG 中
    ON_BN_CLICKED(IDD_BUTTON_1, OnButton1)    // IDD_BUTTON_1 的单击消息
    ON_BN_CLICKED(IDD_BUTTON_2, OnButton2)    // IDD_BUTTON_2 的单击消息
    ON_NOTIFY(NM_CLICK, IDD_LIST_CTRLBOX, OnClickList1) // IDD_LIST_CTRL 的单击通知消息
END_MESSAGE_MAP()

// 为列表框(子窗口) 所映射的响应函数
// 也就是说，单击本来应该由 列表框自己处理， 但这里
// 我们将该消息映射了过来，于是列表框自己不再处理，
// 而由主窗口处理
void MYDLG::OnClickList1(NMHDR* pNMHDR, LRESULT* pResult)
{
    AfxMessageBox (" click in the listctrl box");
    *pResult = 0;
}

// 退出的动作响应函数，如果没有，则右上角无法关闭窗口
void MYDLG::OnCancel()
{
    AfxMessageBox (" EXIT WINDOW ");
    EndDialog (0);
    PostQuitMessage (0) ; // 通知主体函数退出消息循环
}

```



```

void MYDLG::OnButton1()
{
    m_list.InsertItem (0, "");
    m_list.SetItemText (0, 0, "hehe");
    m_list.SetItemText (0, 1, "haha");
    m_list.SetItemText (0, 2, "heihei");
}

void MYDLG::OnButton2()
{
    m_list.InsertItem (1, "");
    m_list.SetItemText (1, 0, "hehe");
    m_list.SetItemText (1, 1, "haha");
    m_list.SetItemText (1, 2, "heihei");
}

int MyDlgTemplate ()
{
    // 定义对话框模板
    static DLGTEMPLATE TopDlg;
    TopDlg.style = DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU;
    TopDlg.x = 0;
    TopDlg.y = 0;
    TopDlg.cx = 150;
    TopDlg.cy = 150;
    // 创建空对话框（容器）
    MYDLG my_dlg;
    my_dlg.CreateIndirect (&TopDlg, NULL);
    my_dlg.ShowWindow(SW_SHOW);
    // 创建两个按钮
    HWND m_button, m_button1;
    m_button = CreateWindow (TEXT("button"), TEXT("fir"), WS_CHILD |
WS_VISIBLE , 20, 20, 60, 60, my_dlg.m_hwnd, (HMENU)IDD_BUTTON_1, NULL, NULL);
    ShowWindow (m_button, 1);

    m_button1 = CreateWindow (TEXT("button"), TEXT("sec"), WS_CHILD |
WS_VISIBLE , 20, 80, 60, 60, my_dlg.m_hwnd, (HMENU)IDD_BUTTON_2, NULL, NULL);
    ShowWindow (m_button1, 1);

    // 创建一个 LIST
    RECT list_rect;
    list_rect.top = 150;
    list_rect.bottom = 250;

```

```

list_rect.left = 20;
list_rect.right = 250;

my_dlg.m_list.Create(LVS_REPORT | WS_BORDER | WS_TABSTOP, list_rect, &my_dlg,
    IDD_LIST_CTRLBOX);
my_dlg.m_list.SetExtendedStyle(LVS_EX_FLATSB|LVS_EX_FULLROWSELECT|LVS_EX_HEA
DERDRAGDROP|LVS_EX_ONECLICKACTIVATE|LVS_EX_GRIDLINES);
    my_dlg.m_list.InsertColumn(0, "first", LVCFMT_LEFT, 70);
    my_dlg.m_list.InsertColumn(1, "second", LVCFMT_LEFT, 70);
    my_dlg.m_list.InsertColumn(2, "third", LVCFMT_LEFT, 70);
    my_dlg.m_list.ShowWindow(SW_SHOW);

    MSG msg; // 主程序的消息循环
    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

// 程序入口
void main()
{
    MyDlgTemplate();
    system ("pause");
}

```

以上的程序便创建了一个独立于 RES 的对话框，这样的对话框连同消息响应与资源一起，如果转移的话，自然是没有任何问题了。

接下来再讨论下关于如何设置程序有效期的问题。（仅仅是本人的拙见）

设置有效期肯定要以时间作为依据，一种方法就是当程序运行的时候，主动去与服务器的时间做对比，当时间过期之后，就停止程序的运行。当然，像我们做二次开发的，可能基本上使用不到服务器了，那么就只能以当前系统的时间做对比了。那么有人要问了，如果当前时间被设置为过去时间，那不是一样可以使用了吗？的确有些程序到期后，通过重新设置当前系统时间之后，又可以运行了。其实这样也是可以的，实现起来也比较简单。相信大多数人都能做出这样的程序来。

下面主要介绍两种即使修改了系统时间，但还是不能正常运行的设置有效期的方法。

方法一：为自己的程序设计一个初始文本，即，每次运行程序的时候，程序

都必须检查该文本。如同软件的 licences 一样，这样，当你的程序运行到期之后，你可以将该文本里面的某项内容修改掉，于是当程序再度运行该的时候，由于该文件已经被你秘密修改，所以初始化便不能成功，便起到了设置有效期的作用。要想继续使用，只有通过更换 licences 达到目的。

方法二：这招比较损一点，当你的程序运行到有效期之后，可以将程序中的某个文件永久删除，这样程序再次启动的时候便不能运行了。要想继续使用，只有更新你的程序才行，当然，更新程序自然是由你来提供了。

方法三：修改用户注册表。添加注册表项目，每次在程序运行的时候都去检查该项的值，如果到了有效期，就将该值修改掉，然后当程序再次启动的时候，再读取，自然走不通了。

以上三种方法都可以有效的设置使用期限，就算使用者修改了本机的当前时间也无法再次启动你的程序。当然，你也可以把如上的三条同时使用，以达到最大的限制程度。

希望大家的有效期不至于一下就被破解掉。。。 (呵呵)

## 专题五： 螺丝系统

本专题介绍的是一个基于元件库的一个综合运用，使用过程类似于 EMX 中调用螺丝的过程。用户在一个组立模型中先选择“点”特征来确定螺丝的位置（一个点特征可以包含很多点），然后再选择对象的“起始面”，和“终止面”，我们便将螺丝从“起始面”打到“终止面”，其中螺丝长度自动计算，螺丝的大小可以由用户选择，螺丝的名称也可以按照流水号修改。当然螺丝成功装配之后，还要对其穿过的模板进行切孔（切孔的 UDF 的尺寸可以由参数驱动），在最后一块板上还要放置一个带螺纹孔的 UDF。

这是一个完整的装配并切割的过程。

由于该过程涉及到比较多的源代码，同时还需要有建立模型的相关知识，所以就不在此处添加了，有需要的朋友可以用邮件向本人索取。

## 专题六： 自动打印系统

在使用 PROE 打印的时候，如果只是利用其本身提供的功能，一次就只能打印一张工程图，这有时候不能满足我们的需要，比如我们一个 ASM 下面有很多工程图，而且每张工程图里面还包含了若干的页面，如果要我们逐个的打印的话，恐怕会耗费掉不少时间，那很自然的，自动打印便提到了开发的日程上来。

自动打印还有另外一个好处，就是可以使用异步模式来做。也就是说，用户只需要在程序界面中指定要打印的工程图的路径，然后指定打印机，程序便自动

在后台启动 **PROE** ，并且将该路径下面所有的工程图都依次传到打印机去进行打印。如果之前配置好先决条件，甚至连打印机都可以不用设置，程序可以自动去根据工程图的信息配置你所需要的打印机，以及需要打印的页面大小。这样就可以节省大量时间，比如说中午下班的时候可以指定要打印的内容，然后吃饭。等到下午上班的时候，图纸已经打印完毕了。

同专题五一样，该功能的需求不是太难理解，关键还是源代码比较多，由于本书中强调的并不是代码本身，所以就不再此处列出了，如有需要，可以通过邮件向本人索取。