第 5 章

程 序 设 计

使用菜单以及各类对话框可以完成一般的数据处理工作，如果仅限于此，使用 Igor 的水平是比较低的，Igor 强大的数据处理能力远远没有发挥出来。Igor 是一个基于命令行的数据处理工具，通过编写程序才能彻底发挥其强大的数据分析威力。如光电子能谱数据，一次实验少则几十张图谱，多则上百张图谱，每张图谱都由好几百条曲线组成，仅靠菜单手动处理分析这些数据，几乎是不可能的。

作为一款数据处理工具，程序设计应该满足几个基本准则：

（1）自由。程序设计的目的就是为了自由地设计各种新功能，这要求编程环境应该类似于普通的编程语言，具有最大的自由度，不仅仅能分析数据，还能访问操作计算机的系统资源，如文件、窗口、输入输出设备等。编程环境必须具有对任何对象的实时获取能力，而不是“硬编码”。

（2）简单。应和普通的程序设计语言区分开来，不能让语法的复杂性掩盖了服务数据处理需求的本质。程序设计的本质就是对各种数据对象的访问和操作，必须有一个最简单、统一的获取基本数据对象的机制，最好是按照人们对数据的直观认识去操作。

（3）高效。既然是编程，在速度上就不能太慢，至少和普通编程语言不能差太多。如果编程仅仅是对命令行的批量解释执行，这样的程序设计是不满足要求的，不能算作真正地支持程序设计。

Igor 在这 3 个方面都支持得非常好。Igor 的图形界面提供的功能，几乎都可以通过它自身的编程环境实现。对数据对象的访问也非常简单，基本的对象都是全局对象，可以利用名字直接访问，同时提供了大量的函数和命令用于获取这些对象的基本信息。在效率方面，Igor 里所有的程序都可以被编译成较底层的机器码，因此速度非常快。

Igor 下的程序设计具有以下特点：

（1）语法环境规范而完整。

（2）面向过程的程序设计语言，命令和函数被组织在一起形成一个功能相对独立的操作单位。

（3）有两种级别的程序模式：脚本程序（Macro、Proc）和函数程序（Function）。前者语法规范宽松，速度慢，但使用简单；后者语法规范严谨，编译为机器代码执行，速度极快，但

使用相对复杂。

(4) 有一个方便的编程环境和调试环境,可以方便地编写程序和调试代码。调试器可查看任意变量值,支持查看表达式。

(5) 可以利用 C/C++编程工具无缝扩展 Igor 功能,甚至控制硬件采集数据。扩充的功能作为动态运行库被 Igor 调用,和内置函数和命令没有任何区别。

(6) 提供了近千个不同的函数和命令,涉及绘图、窗口、数据操作、数据分析、矩阵、信号处理、统计、文件读写、数学函数等多个方面,以供在程序设计中使用。

(7) 程序必须在 Igor 环境下运行。不能编译脱离 Igor 运行的程序。

虽然功能很强大,但是学习曲线并不是很陡峭。即使没有受过任何编程训练的人,通过本书的介绍,也能在很短的时间之内掌握 Igor 的编程方法。

5.1 程序设计概述

5.1.1 程序窗口

程序窗口是写程序的窗口。程序窗口相当于具有 IDE 的程序设计语言中的代码编辑器,在该窗口中可以完成输入代码、编译、运行和调试等工作。程序窗口具有语法高亮的功能,如系统关键字显示为蓝色,函数显示为深橙色,预编译指令显示为紫罗兰色,命令显示为暗青色,注释显示为红色等。

除了编辑功能之外,程序窗口还可以查看帮助,查看某个函数(或程序),运行命令等。最新版本的 Igor(Igor Pro7 版本)还支持外置编辑器,如 Vim。

按 Ctrl+M 键(或者利用菜单命令【Windows】|【Procedure Windows】|【Procedure Window】)就打开了一个程序窗口,这是一个内置的程序窗口,可以在该窗口中输入代码完成程序编写,如图 5-1 所示。

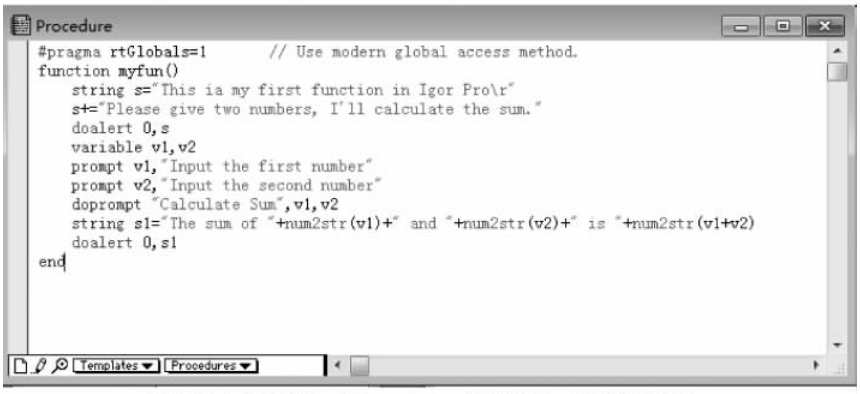
读者可以在程序窗口输入以下代码:

```
Function myfun()
    string s = "This ia my first function in Igor\r"
    s += "Please give two numbers, I'll calculate the sum."
    doalert 0,s
    Variable v1,v2
    prompt v1,"Input the first number"
    prompt v2,"Input the second number"
    doprompt "Calculate Sum",v1,v2
    string s1 = "The sum of " + num2str(v1) + " and " + num2str(v2) + " is " + num2str(v1 + v2)
    doalert 0,s1
End
```

图 5-1　内置程序窗口

输入完毕，单击程序窗口下方的【Compile】按钮编译程序（注意 Igor 下的函数必须在编译后才能运行）。【Compile】按钮在单击后会消失，对应位置显示为空白，表示程序已经编译完毕，如图 5-2 所示。如果代码里包含错误，就会无法编译，此时会显示一个编译错误对话框。

(a) 编译前编译按钮可用



(b) 输入代码后按下【Compile】，编译完毕，编译按钮消失

图 5-2　程序窗口的编译

在命令行窗口输入 myfun()，按回车键执行(注意不能省略圆括号)。还可以在程序窗口中选择 myfun()，按 Ctrl+Enter 键执行程序。程序首先显示一个对话框，然后提供一个输入对话框，输入两个数字，最后显示一个对话框，显示输出结果。

Ctrl+M 快捷键打开的程序窗口是 Igor 内置的程序窗口。Igor 内置的程序窗口默认是全局的，也就是说所有的程序都可以访问内置窗口里的程序和变量。可以通过菜单命令【Window】|【New】|【Procedures】打开一个新的程序编辑窗口，此时可以给窗口起一个有意义的名字，如 MyProc，如图 5-3 所示。

(a) 创建新程序窗口对话框

(b) 新程序窗口

图 5-3 创建新程序窗口

新程序窗口有一个名字，如这里的 MyProc，在程序设计里可以通过这个名字来访问该程序窗口，此时程序窗口相当于一个普通的窗口。

内置程序窗口是 Igor 的一部分，保存实验文件后内置程序窗口里的内容都将被保存。非内置程序窗口如果没有保存到文件，也会随实验文件一起保存，但是一旦被保存到文件里，Igor 就不会再保存整个程序文件，而只是保存程序文件的路径信息。内置程序也可以被保存在文件里，但是其内容仍然会在保存实验文件时被保存。

如果不使用内置程序编辑窗口，例如将不同的函数放在不同的文件中，可以创建新的程序编辑窗口。

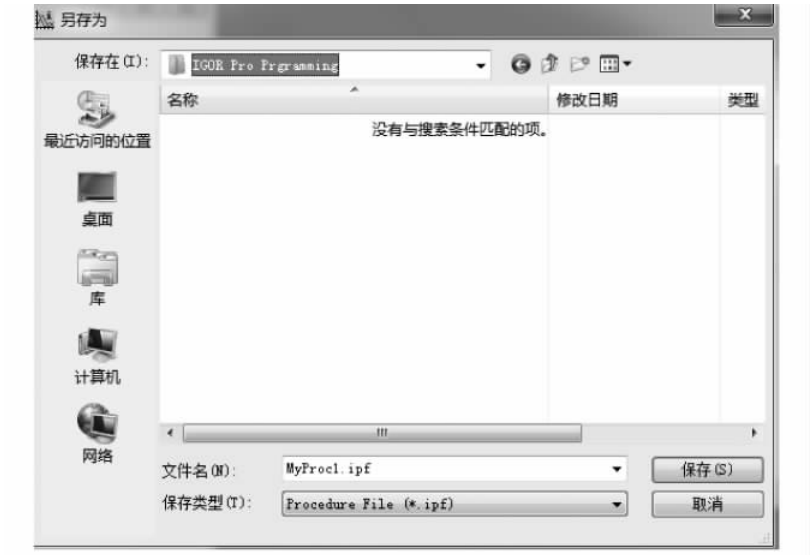
5.1.2　程序窗口说明

1. 保存程序窗口中的程序

习惯于 VS 或者其他编程环境的读者都知道，在 IDE 中创建代码时，需要首先创建一个代码文件，然后在 IDE 中修改或者编辑该文件。在 Igor 中稍稍不同，当利用快捷键 Ctrl＋M 或者 Windows 菜单创建一个新的程序编辑窗口，在其中编写代码时，代码是实验文件的一部分，并没有创建对应的文件（很好理解，因为这个过程中没有指定任何要保存的文件）。当保存实验文件时，程序窗口连同内部的代码会和数据一起保存，这样当下次打开实验文件时，里面的程序窗口及程序代码也会被同时打开。

实际情况中，程序应该是通用的，而不是依赖于某个具体的实验文件，这就需要将程序窗口中的代码以文件形式单独存放于硬盘或者其他存储介质中。

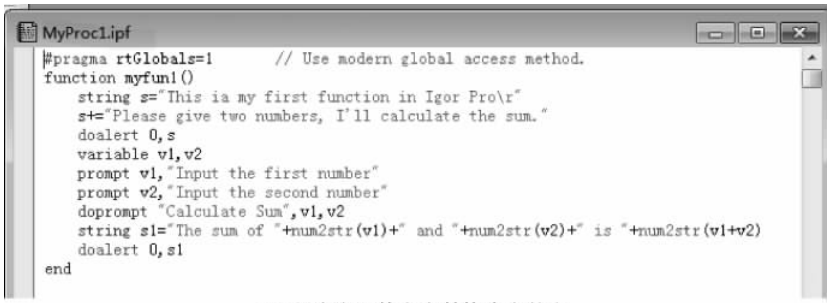
保存程序文件的方法很简单，保持需要被保存的程序窗口处于激活状态，执行菜单命令【File】|【Save Procedure As】，在弹出的保存对话框中浏览到合适位置保存即可。此时窗口中所有的内容都会被保存在一个扩展名为 ipf 的文件中，程序窗口的标题也被替换为保存时的名字。一般将经常用到的程序文件保存到 Igor 安装目录/User Procedures 或者 Igor 安装目录/Igor Procedures 下。当然可以不存放在这个文件夹里，此时使用 Include 指令包含程序文件时需要指明完整路径。保存程序文件如图 5-4 所示。



(a) 保存程序文件对话框

(b) 已保存的程序文件

图 5-4　保存程序文件



(c) 程序窗口的名字替换为文件名

图 5-4 （续）

当打开一个存放于硬盘上的程序文件(或者将程序文件保存在硬盘上)时,Igor 会建立该文件与 Igor 中对应窗口的关联,但这种关联不是实时的。如果修改代码,代码并不会实时保存到程序文件中,编程者需要通过按 Ctrl+Shift+S 键来保存程序文件的内容(或者通过菜单命令【File】|【Save Procedure】)。每次按 Ctrl+Shift+S 键只保存当前程序窗口,如果有多个程序窗口,需要选择每个窗口并执行保存命令。

当程序窗口被保存为独立文件时,Igor 仅会在实验文件里保存该程序文件的路径信息,而不再保存其内容。下次打开实验文件时,Igor 会自动根据路径信息打开该程序文件(并列出在【Windows】|【Procedures】菜单下)。如果程序文件不存在(这种情况很普遍,比如打开从别的地方复制的实验文件,程序文件被误删或者改名,程序文件的存放路径发生改变),Igor 就会报错并显示一个对话框让用户提供正确的程序文件路径。本例中,读者可以先保存实验文件(按 Ctrl+S 键)并关闭实验文件,在电脑里将 MyProc1.ipf 改为 MyProc2.ipf,然后再打开已经保存的实验文件,就会看到如图 5-5 所示的错误提示。

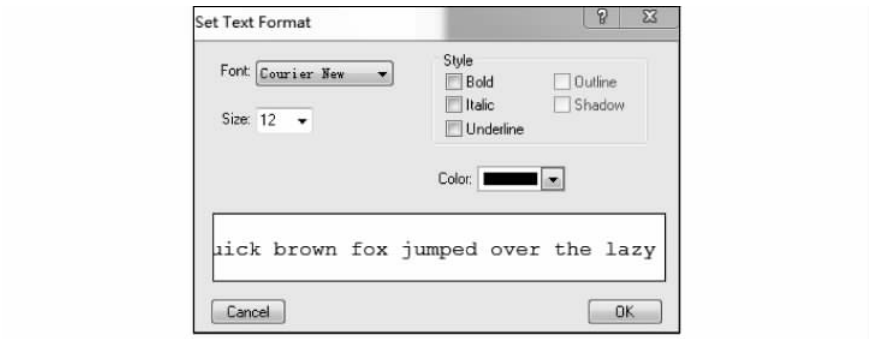
图 5-5 找不到程序文件的错误

可以单击【Skip This File】按钮忽略错误,这种情况下程序文件就不能正常打开了,但不会影响实验数据或者其他程序的打开。程序文件没有打开,自然就不能使用程序文件里的程序,可以按【Look for File】按钮提供正确的程序文件。

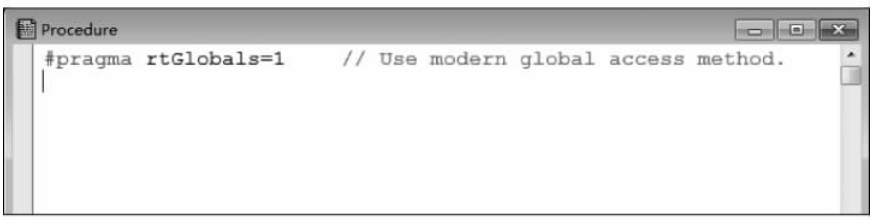
2. 设置程序字体

对于中文语言字体的计算机,程序窗口默认字体为宋体,字体尺寸也往往比较小,看起来不太舒服,可以将其修改为常见的代码字体 Courier New,并设置合适字号如 12pt。设置方法如下:

选择一个程序窗口为当前窗口，主菜单栏会动态出现一个【Procedure】的菜单（利用该菜单可以设置程序编辑窗口的字体、字号等信息），执行菜单命令【Procedure】|【Set Text Format】，打开字体设置对话框，进行如图 5-6(a)所示的设置，单击【OK】按钮，可以看到程序编辑窗口的字体变为常见的代码字体，如图 5-6(b)所示。



(a) 字体设置对话框



(b) 更改字体、字号后的效果

图 5-6　设置程序窗口字体

注意，设置好的字体需要保存，否则当下次打开 Igor 后仍然为默认字体。保存设置的方法是执行菜单命令【Procedure】|【Capture Procedure Prefs】，选择相应的项，这里选中【Text Format】复选框，然后单击【Capture Prefs】按钮，就可以为这台计算机永久保存字体设置。读者可以试着创建一个新的程序编辑窗口并将字体设置为合适的字体。

3. 打开并查看已有程序文件

打开已有程序文件的方法非常简单，可以直接将已有程序文件拖入 Igor 或者执行菜单命令【File】|【Open File】|【Procedure】打开一个程序对话框，定位到程序文件处打开即可。如果在处理实验数据时打开了已有的程序文件，则在保存实验数据时会自动保存该程序文件的路径信息，下次打开实验文件时会自动打开程序文件。这要求程序文件存在且位置不能发生变化，否则就会出现打开文件错误。

已经打开的程序文件可以修改其内容，如果修改时出现如图 5-7 所示提示则表示程序文件处于写保护状态。

观察程序文件左下角，可以发现一个上面有一条斜线的铅笔状的小图标，表示当前程序文件处于不能编辑状态，如图 5-8 所示。单击该图标，即可解除不能编辑状态，再次单击则恢复不能编辑状态。

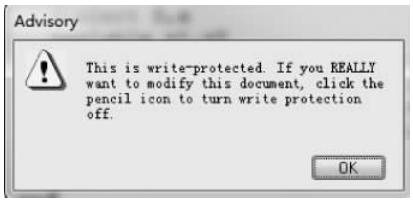


图 5-7　程序文件写保护警示

图 5-8　程序文件写保护状态

图 5-9　程序文件被其他程序打开或者被操作系统写保护，则不能修改

如果打开的程序文件已经被别的实验文件打开，或者是操作系统设置了写保护，则“小铅笔”对应位置处是一把小锁的图标，此时程序文件不能被修改，如图 5-9 所示。

单击图 5-9 最左端一个类似于文档的小图标，会显示当前程序文件的基本信息，如存放路径、当前光标所处行数等。小放大镜可以按比例对字体缩放。【Templates】下拉列表可以插入函数、命令模板或者各种流程控制语句，【Procedures】下拉列表可以查看自定义 Macro 或者函数。

所有打开的程序文件都被列入【Windows】|【Procedure Windows】菜单内，可以在这个菜单中找到并查看已经打开的程序文件。但也有例外，当程序文件中指定 Pragma 的 *hide* 参数为 1 或者指定为 IndependentModule 或者是操作系统设置为隐藏时，虽然程序文件已经打开，但【Windows】|【Procedures Windows】菜单内并不列出。隐藏文件提供了一种保护代码不被修改的方法，但一般不建议采用。因为对于普通的使用者，即使文件不隐藏也一般也不会被修改，而对于程序开发者而言隐藏文件除了给自己制造一些麻烦之外没有什么别的用处。

IndependentModule 参数很有用，它可以指定一个程序文件单独编译，即使当前程序文件出现错误无法编译，IndependentModule 中的程序仍然可以正常运行，这在使用 Igor 进行数据采集时非常重要，请参看 5.3.2 节。

可以按 Ctrl＋Alt＋M 键在已经打开的程序文件窗口中切换查看，按 Ctrl＋Alt＋Shift＋M 键隐藏当前程序编辑窗口并自动显示下一个程序文件对应的程序编辑窗口。但这两个快捷键并不好用，最好的办法还是利用【Windows】|【Procedures】菜单定位到指定的程序文件，也可以右击相应的对象（如控件、函数），然后【Go To】到指定的程序文件。

4. 关闭程序文件

单击程序编辑窗口右上角的[×]按钮或者按 Ctrl＋W 快捷键，会调出关闭程序文件对话框，如图 5-10 所示。

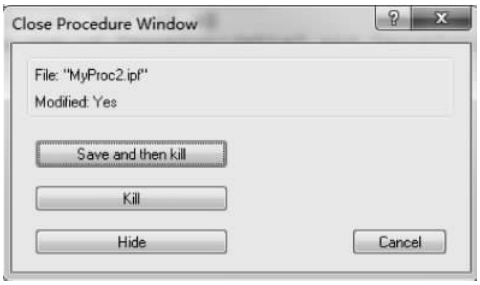


图 5-10　关闭程序窗口对话框

在这里有 3 个选项：

- 【Save and then kill】表示将程序编辑窗口中的程序保存到计算机存储设备，然

后从 Igor 中彻底关闭，此时程序文件中的所有程序都将不能再使用。

- 【Kill】表示彻底关闭且不保存。
- 【Hide】表示隐藏程序编辑窗口，但是程序文件里的内容仍然留在内存里，可以通过菜单命令【Windows】|【Procedure Windows】查看并重新打开，或者利用其他方法如通过查找对话框等重新打开。

一般在关闭程序编辑窗口时选择【Hide】按钮，既不影响程序文件中的函数的使用，同时又不会被程序编辑窗口遮挡影响数据处理。

若【Macro】菜单中的【Auto-Compile】为打开状态（默认）时，单击【Hide】按钮会自动编译程序。当打开一个写好的程序时，应该选择【Hide】按钮编译并隐藏程序文件，这样就可以像使用 Igor 内置的函数和命令一样使用程序文件中提供的程序。

5. Adopt 程序文件

Adopt 是保存程序文件的逆操作——将程序文件重新保存到实验文件里。

在保存实验文件时，如果实验文件里的程序文件是存放于硬盘上的 ipf 文件，则程序文件里的内容不会被保存在实验文件里，而是保存了该程序文件的路径信息，这样当下一次打开实验文件时，Igor 会根据路径信息自动打开该程序文件。但这样存在一个问题，如果实验文件在另外一台计算机上打开（比如与别人交流实验数据时），如果该计算机上不存在对应的程序文件，就会出现找不到程序文件的错误，实验文件也不能使用该程序文件中的程序。解决这个问题的方法是 Adopt 程序文件。

要 Adopt 一个程序文件，先使该程序文件（外部打开的）处于显示状态，然后选择菜单命令【File】|【Adopt Procedure】，Igor 会显示一个提示信息，提示用户会先复制一份程序文件放到当前实验文件，然后断开与源文件的关系。Adopt 程序文件后，程序文件的内容将会成为实验文件的一部分，随实验文件保存，这样该实验文件在任何地方都可以使用程序而无须关心程序文件是否存在。

5.1.3 编译程序

在第 5.1.1 节可看到程序只有在编译后才能运行。当然不是所有的代码都需要编译后才能运行，如果只是调用内置命令和函数的命令行，或者是 proc 和 macro，无论是在命令行窗口还是程序窗口中运行都不需要编译。但注意，如果使用自定义函数，则必须编译自定义函数所在的程序文件。

Igor 的程序分为编译阶段和运行阶段。编译阶段，编译器会检查代码中的语法错误并将代码生成较低层次的机器语言。运行阶段，调用前面编译好的代码。编译后的机器代码执行速度要比直接解释执行快得多。

Igor 下程序文件的编译非常简单，当菜单命令【Macro】|【Auto Compile】处于被选状态时，单击程序编辑窗口以外的任何地方都会自动完成编译。第 5.1.2 节中提到的关闭程序文件选择【Hide】按钮编译程序文件就是这个原理。也可以主动编译程序文件，保持待编译的程序文件处于激活显示状态，选择菜单命令【Macro】|【Compile】进行编译，或者按程序左下角的【Compile】按钮进行编译，如图 5-11 所示。

图 5-11 编译程序按钮

编译以后的程序文件【Compile】按钮处对应的地方为空白，否则会显示一个虚线状态的【Compile】按钮。

【Macro】|【Auto Compile】即自动编译默认为打开状态，以方便程序自动编译。但有时需要关闭这个功能，在编写程序时这样做很有必要。在程序开发的过程中经常需要查看文件夹、wave、全局变量等信息，如果设置为自动编译，则在查看这些信息的过程中，Igor 会自动编译还未完成的程序并报错，这会给程序的编写带来不便，此时就可以关闭自动编译。

不同于普通的程序设计，Igor 并不会生成独立的可执行程序，因此每次打开程序文件都需要执行编译过程。如果程序文件较大，编译会花费较长的时间，但一般都是很快的。如果程序文件随实验文件一起打开，则编译过程是自动的，不需要手动操作。

5.1.3 程序代码构成

程序设计就是在程序编辑窗口中输入代码。按照代码在程序中的作用不同，可以将程序文件中的内容划分以下 10 个类型。

(1) 编译器指令，关键字为 Pragma，一般位于程序文件开头，格式为 # Pragma，并紧靠程序文件最左侧，用于指定编译参数。

(2) 程序文件包含指令，关键字为 Include，一般位于程序文件开头，格式为 # Include <文件名>或者 # include “文件名”，和 C 语言程序设计类似，用于打开并包含一个已有的程序文件。

(3) 常量声明，关键字为 Constant，用于声明常量，类似于 C 语言的 # DEFINE 指令。

(4) 结构体类型定义，关键字为 Structure，定义的结构体变量类型可以在函数中使用。

(5) 图片资源，关键字为 Picture，以代码格式保存图片，类似于编程语言中的图形资源，这些图形资源可以作为图形按钮或者窗口程序的背景图片。

(6) 菜单，关键字为 Menu，定义菜单，可以向系统菜单添加菜单项，也可以创建自定义菜单。

(7) 函数，关键字为 Function，程序设计的主要内容，完成数据处理的功能单位，需要编译以后执行。

(8) 脚本，关键字为 Macro 和 Proc，功能类似于函数，但是并不需要编译，解释执行，类似于 Windows 下的批处理文件或者 Linux 下的 bash 文件。

(9) 预编译指令，用于条件编译。

(10) 注释，格式为以“//”开头，和 C 语言类似。

请读者对照图 5-12 所示的程序代码示例理解上面的内容，其中用 Include 指令包含的文件内容如图 5-13 所示。

Procedure

```
#pragma rtGlobals=1                          //编译器指令

#include "E:\IGOR Pro Prgramming\proc0" // 包含指令


#ifndef CS1                                  // 预编译指令

constant CS=1                                // 常量声明

#endif

structure myPoint                            //结构体类型
   variable xx
   variable yy
endstructure

function f1()                               // 函数
   struct myPoint pt
   pt.xx=0
   pt.yy=0
   struct myPoint shift
   shift.xx=CS
   shift.yy=CS
   f2(pt,shift)                              // 函数f2存放于Include指令打开
                                             // 的程序文件里

   print "shifted (x,y)=(",pt.xx,",",pt.yy,")"
end

Macro macro1()                               //脚本
   f1()
end

menu "mymenu"                                //菜单
   "calculate shift",f1()
end
```

Templates Procedures

图 5-12　程序代码构成类型

proc0.ipf

```
#pragma rtGlobals=1     // Use modern global access method
function f2(p1,p2)
   struct mypoint &p1
   struct myPoint &p2
   p1.xx=p1.xx+p2.xx
   p1.yy=p1.yy+p2.yy
end
```

Templates Procedures

图 5-13　被 Include 的程序文件

Proc0.ipf 存放位置如图 5-14 所示。

图 5-14　被 Include 程序文件存放于计算机硬盘上

注意观察在菜单栏的最后一项添加了一个名为 mymenu 的菜单，如图 5-15 所示。

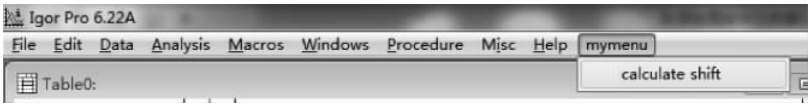


图 5-15　自定义菜单

【Macros】菜单下面也添加了一个名为 macro1 的菜单项。【macro1】和【mymenu】菜单项的功能是相同的，即执行后在历史命令行窗口打印信息，如图 5-16 所示。

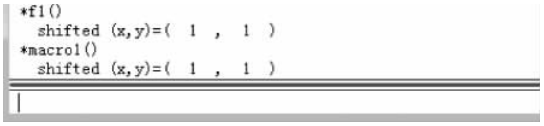


图 5-16　历史命令行窗口打印信息

上面没有包括 Picture 的例子，关于 Picture 在后面窗口程序设计时进行详细介绍。

这些代码形式是 Igor 下语法结构的一部分，这里列出来主要是给读者一个整体印象，让读者了解一个完整的程序文件的基本组成。关于每个部分的详细介绍请参看接下来的内容。

5.1.4　程序类型

Igor 下可执行的程序类型有 3 种：Macro、Proc 和 Function。这些程序内容基本一样：都包含变量声明和流程控制语句，通过调用其他程序处理数据或者对数据进行拟合，或者创建新的变量数据，实现一个具体的功能。3 种程序形式的格式可描述如下：

```
Macro macroname(parameters list)
    body
End

Proc procname(parameters list)
    body
```

```
End

Function functionname(parameters list)
    body
End
```

可以看到,3 种程序形式也非常类似,功能也几乎完全相同。一般而言,能用 Macro 和 Proc 实现的功能,Function 都可以做,反之亦然。但是它们的执行效率是不一样的,Macro 和 Proc 是解释执行,而 Function 是编译执行。

Igor 是基于命令行的实验数据处理软件,在设计之初,为了执行命令的方便,引入了 Macro 的概念和方法,将命令以脚本方式组合起来,并扩充一些功能,如变量声明和流程控制等,辅助程序设计,这有点类似于 Windows 下的批处理以及 Linux 下的 shell 程序,其本质是批量执行多条命令。但缺点也很明显,由于是解释执行,当数据量比较大或者计算量比较大时,十分耗时。因此 Igor 后来的版本引入了函数的概念,并引入了自己的编译器。不同于 Macro,函数并不是解释执行,而是先编译为低层次的机器指令,然后再执行,这就大大提高了程序运行的速度。下面比较 Macro 和 Function 二者在执行效率上的差异:

```
Macro test1()
    Variable i
    Variable t0,t1
    t0 = ticks
    i = 0
    silent 1
    do
        i += 1
    while(i < 1e5)
    t1 = ticks
    Print/D "Time used is ",(t1 - t0)/60,"s"
End
Function test2()
    Variable i
    Variable t0,t1
    t0 = ticks
    i = 0
    silent 1
    do
        i += 1
    while(i < 1e5)
    t1 = ticks
    Print/D "Time used is ",(t1 - t0)/60,"s"
End
```

在命令行输入 test1()和 test2()分别执行，结果如图 5-17 所示。

```
*test1()
  Time used is   11.1333333333333  s
*test2()
  Time used is   0  s
```

图 5-17　脚本和函数执行的时间差异

完全相同的代码，test1()即 Macro 形式，执行时间约为 11s，而 test2()即 Function 形式执行时间在计算机数据精度范围内为 0s，可见二者的效率差别之巨大。因此在编写程序时尽量使用 Function 而不使用 Macro。

Proc 其实是另外一种形式的 Macro。一般而言，使用 Macro 关键字声明的程序都会被默认添加在系统【Macro】菜单下，以方便执行，如果不希望这些程序出现在【Macro】菜单下，就可以使用 Proc 关键字声明程序。除此之外，二者完全相同。其实还可以利用 Window 关键字声明一个 Macro，其执行方式和普通 Macro 一模一样。

由于 Macro 和 Function 的这种差异，一般不建议在设计程序时使用 Macro，而应该使用 Function。Macro 更多地是被 Igor 用来做一些自动化的操作，如自动生成程序面板创建代码、图表创建代码、样式风格创建代码等。

除了执行效率和使用场合的区别之外，Macro 和 Function 还有一些语法上的区别，Macro 在使用上更自然方便一些，可以不需要声明而直接访问全局变量或者 wave，但是有些语法不能使用，如不能使用 for 循环，不能使用 switch 语句等。而 Function 则相对严谨，全局变量必须在声明引用以后才能使用，语法格式也更加丰富多样。

5.2　基本语法

作为一门完全可编程的语言，Igor 具有系统而完整的语法环境，体现在以下几个方面：

第一，高度自治，不需要引进任何外部支持，原生支持脚本级别、编译级别的程序设计，并提供了很多普通编程语言才有的高级编程技术，如预编译指令、文件包含、条件编译、名称空间等。编译器支持的功能广阔，如字符串处理、结构体变量、函数指针、对所在平台系统资源的自由访问、线程安全函数设计及多线程编程等。这使得仅仅使用 Igor，不借助任何外部专门的编程工具，就能够实现各种复杂的功能。

第二，显著区别于普通的程序设计语言，将程序设计的复杂性做了包装，针对数据处理程序的设计做了大大简化，如变量仅包括数值型和字符串型两类变量（当然也可以指定更具体的变量类型，这会提高效率，但一般这两类就够了），没有普通编程语言“强类型数据”所带来的复杂性，所有的内置函数和命令（包括用户自定义函数）无须声明即可自由使用，提供极为通用的方式自由访问 Igor 的数据对象，如 wave、变量、窗口等。

第三，提供了相当方便的在线帮助系统，可随时查询函数及命令的使用并能够将示例直接插入代码处，几乎所有的对话框操作都能转化为对应的命令行并醒目显示，这些命令行都可以直接作为程序的代码。

第四，提供了方便易用的扩展接口，可以利用 C++等编程工具任意扩展其功能。

这4个方面的特性使得Igor非常适合于处理复杂且体积较为庞大的实验数据。在需要大批量重复的操作、在多变量中任意调整某变量随时观察数据变化和需要编写某一专门类型的数据处理工具时，Igor更是极好的选择。

Igor下语法结构包括编译指令、文件包含、函数定义、变量定义，流程控制语句、注释等。本节重点介绍关于这些语法结构的基本含义。

另外注意，Igor下的程序设计是面向过程的，主要通过函数和各种结构来构建程序体系。

5.2.1 表达式和命名规则

表达式是代码的基本单位。变量声明语句、赋值语句、内置命令调用、用户自定义函数调用等都属于表达式：

```
Variable v = sin(x)
Make/N = 100 gsfun = gauss(x,50,1) + gnoise(0.1)
CurveFit gauss,data/D
myfun()
```

表达式里可以包含任何内置函数、命令，或是用户自定义函数。每一行只包含一个表达式，表达式结尾不需要使用分号（当然加上分号编译器也不报错）。另外，Igor程序设计语言不区分大小写，这和很多程序设计语言完全不同，如make、Make、MAKE表示完全相同的命令。Igor下编程语言使用ASCII字符，Unicode字符（如中文字符）只能包含在字符串中或者注释中。

Igor下的命名规则和普通编程语言类似，无论是变量、函数、wave、窗口，名字一般都由字母、数字、下画线组成，其中下画线和数字不能位于开头，不能包含空格、逗号、冒号等字符，名字不能和系统内置的关键字重名。如下面的名字都是不合法的：

```
Variable sin
Make cos
newpanel/N = sin            //系统会自动将 sin 改为 sin0
Variable 1v, _v
```

注意，wave的命名条件稍微宽松一些，可以创建包含特殊字符的wave名字，如

```
Make 'a wave'
```

这时wave名字需要用单引号括起来，Igor里把它叫作Liberal Name。这种命名方式会给wave的访问带来很大麻烦，因此不建议使用。

5.2.2 变量和常量

变量包括数值型变量和字符串型变量。常量也可以理解为一种特殊的变量，只不过其值不能修改。常量同样包括数值型常量和字符串型常量。

1. 数值型变量

数值型变量的声明方式为

```
Variable [/C/D/G] varName [ = numExpr ][, varName[ = numExpr ]]...
```

其中，Variable 是关键字，*varName* 是变量名。可以在声明时给变量赋值。如果声明后未赋值那么系统默认赋值 0。

Variable 有 3 个选项，用于限定所声明变量的类型。

(1) /C，表示声明一个复数型变量。

(2) /D，表示声明一个双精度型变量，由于 Igor 下任何数值变量都默认为双精度，因此/D 参数没有实际意义，主要是为了向前兼容(即和旧版本的 Igor 兼容)。

(3) /G，表示声明一个全局变量，此变量会出现在当前数据文件夹下，并作为实验数据的一部分随实验数据永久保存。

和一些强类型语言如 C 语言等不同，使用 Variable 声明的数值型变量没类型之分，所有的变量不管是整型还是浮点型甚至是 Char 型，其声明全部使用 Variable 关键字，在计算中全部使用双精度类型。这使得 Igor 下的变量定义和使用非常简单。

变量名的命名规则和 C 语言相同，包括字母、数字和下画线，其中数字不能位于首位，变量名不能有空格等特殊字符，变量名也不能为系统关键字、函数名或者命令。由于 Igor 编程语言不区分大小写，所以 num1 和 NUM1 表示同一个变量。

数值型变量主要用于程序设计，用来存放中间变量。全局的数值型变量主要用于保存控件状态以及在不同的程序之间传值。

下面举几个变量定义的例子：

```
Variable v1                    //声明一个变量 v1
Variable v1 = 0                //声明一个变量 v1,并赋值为 0
Variable pi                    //错误,pi 是系统函数,返回圆周率值
Variable sin                   //错误,sin 是系统数学函数
Variable/C c = cmplx(1,1)      //声明一个复数型变量,并赋值 1 + i
Variable/G v2                  //声明一个全局变量 v2
Variable _v3                   //错误,不能以下画线开头
```

2. 字符串型变量

字符串是字符的集合，在一般编程语言中经常用字符数组来表示。在 Igor 中字符串被简化为一个 String 型的变量，使用起来相当方便。

字符串变量的声明方式为

```
String [/G] strName [ = strExpr ] [,strName[ = strExpr ]...]
```

其中，String 是关键字，*strName* 表示字符串变量名，/G 选项和 Variable 含义一样，表示声明一个全局字符串。

字符串变量名的命名规则和数值型变量相同。字符串变量名在声明时可以初始化。如

果没有初始化，则系统会初始化为 null，即空字符串。空字符串是不能使用的，否则会报错。在程序中一定要在声明字符串后对字符串赋值。

字符串变量主要用于处理数据对象、图形对象以及其他 Igor 对象的名字，以方便在程序里对这些对象访问和处理。Igor 对字符串处理的支持相当完善，提供大量的函数和命令用于对字符串的操作。

下面举几个字符串声明的例子：

```
String str1 = ""                    //声明一个字符串 str1,赋空值
String str1 = "hello,world"         //声明一个字符串 str1,并赋值"hello,world"
String/G str1                       //声明一个全局型字符串 str1
```

3. 变量的作用域和寿命

变量的作用域取决于声明变量的位置和声明时的选项。如果在函数体内声明（包括 Function、Macro、Proc）且没有使用/G 选项，则变量的作用域只局限于程序体内，此时变量又称为局域变量。如果在程序体外声明或者是在命令行窗口声明，或者虽然在程序体内声明但是使用了/G 选项，则变量的作用域为全局的，所有的程序都可以访问该变量。

局域变量的寿命在程序运行终止时即结束。全局变量的寿命则是永久的，除非被删除。

4. 常量定义

常量分为数值型常量和字符串型常量。

数值型常量的定义方式为

```
Constant kName = literalNumber
```

字符串型的常量定义方式为

```
Strconstant ksName = "literal string"
```

常量一般在程序体外程序文件的开头定义，作为一个常量使用，类似于 C 语言的 #DEFINE 语句。如果没有用 static 修饰符修饰，其作用域是全局的。

注意常量和变量不同，系统并不会分配内存空间储存常量，因此不能对常量修改或者赋值。

5.2.3 Structures

Igor 支持结构体。一般在程序文件的开头、程序的外部定义结构体类型，在函数内部通过声明方式创建结构体类型的实例。结构体定义方式如下：

```
Structure structurename
    memtype memname [arraysize] [,memtype,memname [arraysize]]
…
EndStructure
```

Structure 是关键字，*structurename* 是要创建的结构体类型名字，*memtype* 是 Igor 下支

持的变量类型，包括 variable、string、wave、nvar、svar、dfrer、funcref 和结构体变量类型，除此之外，*memtype* 还包括一些其他的数据类型，这些数据类型和 C 语言非常类似，如表 5-1 所示。

表 5-1　结构体支持的类 C 数据类型

变 量 类 型	C 语言对应类型	字　节　数
Char	Signed char	1
Uchar	Unsigned char	1
Int16	Short int	2
Uint16	Unsigned short int	2
Int32	Long int	4
Uint32	Unsigned long int	4
Float	Float	4
Double	Double	8

在结构体里可以声明数组，利用 *arraysize* 指定数组的长度，这个长度必须是常数或者常量，不能是变量。注意在函数中不能声明数组。

创建好结构体类型后，可以使用如下的语法在函数中创建结构体变量：

```
Struct structurename name
```

这里 Struct 是必需的，类似于 C 语言里的 Structure 关键字，*structurename* 是结构体类型名，*name* 是要创建的变量名。

下面是一个可能的例子：

```
Structure mystruct
    Variable var1
    Variable var2[10]
EndStructure
Function myfunc()
    Struct mystruct ms
    ms.var1 = 1
    ms.var2[n] = 20
    ...
End
```

上面的例子中，首先创建了一个结构体类型 mystruct，然后在函数内利用 Struct mystruct ms 声明了该结构体类型的一个实例，即创建一个 mystruct 类型的变量 ms，然后对 ms 赋值。这里看到访问结构体变量成员的方法：变量名.成员名，这和 C 语言完全一样。

给 ms.var2 数组赋值时，利用一个变量指定要赋值的位置，这是允许的。

下面是一个更为复杂的结构体类型，当然这个结构体类型没什么实际作用，主要是为了

让读者理解结构体的概念：

```
Constant kCaSize = 5
Structure substruct
    Variable v1
    Variable v2
EndStructure
Structure mystruct
    Variable var1
    Variable var2[10]
    String s1
    Wave fred
    Nvar globVar1
    Svar globStr1
    Funcref myDefaultFunc afunc
    Struct substruct ss1[3]
    char ca[kCaSize + 1]
EndStructure
```

上面的结构体类型中几乎包含了 Igor 下所有的数据类型。当结构体类型中包括引用时，在创建结构体变量后必须对该引用赋值，指明引用的对象。

```
Struct mystruct ms
Wave ms.fred
Nvar ms.globvar1
Svar ms.globstr1
Funcref mydefaultfunc anotherfunc
```

这里假定当前文件夹下存在 fred wave、globvar1 和 globstr1 全局变量。注意为引用赋值时和在函数中声明引用的方法完全一样，这里不能省略 wave、nvar 和 svar 关键字。如不能写成如下形式：

```
ms.fred = fred
```

也不能写成如下形式：

```
Wave w = fred
Ms.fred = w
```

关于引用的详细介绍请参看 5.3.6、5.3.7、5.3.8 节内容。

结构体变量在函数中传递时是按照引用传递的，因此被调用函数可以修改调用函数中结构体变量的值，这使得结构体变量既可以作为函数的输入参数，也可以作为函数的输出参数。当使用结构体变量作为参数时，语法如下：

```
Function subfunc(s)
    Struct structname &s
    …
End
```

这里使用了 & 符号获取结构体变量的引用。

Igor 内置了一些预定义好的结构体类型，如 Button 控件对应的 WMButtonAction 结构体类型，CheckBox 控件对应的 WMCheckBoxAction 结构体类型等。这些结构体里包含了关于该控件的各种状态和运行信息，以方便对控件进行控制和对控件的状态作出反应。关于控件结构体类型将在第 7 章图形用户界面程序的程序设计中详细介绍。

5.2.4 流程控制语句

流程控制语句包括条件判断语句、分支语句和循环语句。

1. 条件判断语句

条件判断语句有两种形式：

```
if (<expression>)
    <True Part>
else
    <False Part>
endif
```

和

```
if (<expression>)
    <True Part1>
elseif
    <True Part2>
else
    <False Part>
endif
```

其中，expression 是一个数值型表达式，非 0 表示真，0 表示假。这里的数值型表达式可以是普通的数值型表达式，如 a+b，也可以是一个逻辑数值型表达式，如 a==b。else 可以省略，因此最简单的条件判断语句为

```
if (<expression>)
    <True Part>
endif
```

注意，不能忽略后面的 Endif。Igor 不使用大括号作为语句块的标识，而是使用 End 或者 End 作为前缀的关键字来作为一个语句块结束的标识，请读者注意体会这个特点。if 语句可以嵌套，即 if 内部还可以包含 if 语句。

下面是一个简单的条件判断的例子，其作用是比较两个数的大小：

```
Function example()
    Variable a = 2,b = 1
    if(a > b)
        Print a
```

```
    else
        Print b
    endif
end
```

表 5-2 列出了常见的比较运算符。

表 5-2　常见的比较运算符

运　算　符	含　　义	运　算　符	含　　义
==	相等	<=	小于或等于
!=	不等	>	大于
<	小于	>=	大于或等于

当比较结果为真时返回 1，否则返回 0，如 Print 2==2 结果为 1，Print 2>3 结果为 0。比较字符串的大小时需要使用相应的字符串函数，如 CmpStr。CmpStr 会根据字符串大小返回一个数字。总而言之，只要是运算结果为数字的都可以充当条件判断表达式，甚至是一个常量，如 if(1)这样的语句也是可以的。

表 5-3 列出了常见的逻辑运算符和位运算符。

表 5-3　逻辑运算和位运算符

运　算　符	含　　义	运　算　符	含　　义
~	按位取反	!	逻辑取反
&	按位与	&&	逻辑与
|	按位或	||	逻辑或

逻辑运算符用于连接多个条件判断表达式。位运算符则常用于对比特变量按位进行设置。一些命令或者函数如 TraceNameList 使用比特变量作为参数，使用位运算符可以方便地对这些比特参数进行置位或取消置位。运算符之间存在优先级，圆括号可以改变优先级。为避免出错，建议读者在使用运算符时多使用圆括号，而不要依赖于对优先级的记忆，这样会避免很多错误。

2. 分支语句

分支(switch)语句也有两种形式：

```
switch(<numerical expression>)
    case <literal><number constant>:
        <code>
        [break]
    case <literal><constant>:
        <code>
        [break]
    [default:
        <code>]
```

```
Endswitch
```

和

```
strswitch(<string expression>)
    case <literal><string constant>:
    <code>
        [break]
    case <literal><string constant>:
        <code>
        [break]
    [default:
        <code>]
Endswitch
```

不同于C语言，switch语句有两种格式，分别对应数值型和字符串型。switch根据numberical expression或者string expression的值选择执行相应case分支对应的代码，如果有break关键字，则执行完后跳出swtich语句，否则顺序执行其后的case的代码，直至碰到break或者到结束为止。default如果存在且没有任何case被对应，则执行default语句后面的代码。注意case标识符后面只能是一个具体的数字(双引号括起来的字符串)或者是一个数值型常量(字符串型常量)，而不能是变量。

当判断分支很多，用if语句比较麻烦时，就可以选用switch语句，如使用window hook函数侦测键盘输入，用switch语句判断键码就非常方便。下面是一个switch语句的例子，其作用是输出两个数中较大的一个：

```
switch(a>b)
    case 1:
        Print a
        break
    case 0:
        Print b
        break
    default:
        break
endswitch
```

switch语句不能用于proc或者maro形式的程序中。

3. 循环语句

循环语句有两种形式，分别是for循环和do-while循环，其格式为

```
do
    <loop body>
while(<experssion>)
```

和

```
for(<initialize>;<continue test>;<update>)
    <loop body>
endfor
```

其中，expression 是一个表达式，当表达式非 0 时执行循环，表达式为 0 时停止循环。for 循环先初始化条件，然后每次循环后先更新循环变量，接着计算循环条件表达式并进行判断，如果判断结果为真则执行循环，否则跳出循环。下面举一个例子：

```
Function example()
    Variable I = 0,sum = 0
    do
        sum += i
        i += 1
    while(i <= 100)
    Print sum
End
```

上面的例子用于计算从 1 到 100 的和，如果用 for 循环可写为

```
Function example()
    Variable I,sum = 0
    for(i = 0;i <= 100;i += 1)
        sum += i
    endfor
    Print sum
End
```

循环是可以嵌套的，但是对于一些计算量非常大的数据处理，应尽量避免将大量的计算放在最内层的循环，否则会大大增加运算的时间。

for 循环只能用于 Function 程序类型，而 do-while 循环则可以用于 Function 和 Macro 程序类型。这也是 Function 和 Macro 的区别之一。

和 C 语言类似，在循环中可以使用 break 和 continue 语句用于跳出循环或者满足某些条件时不执行循环体而进入下一个循环。break 和 continue 作用于包含它的最近循环。

如上面的例子用 break 可以写为

```
Function example()
    Variable I = 0,sum = 0
    for(;;)
        sum += i
        i += 1
        if(i > 100)
            Break
        endif
    endfor
End
```

求 1 到 100 之间所有奇数的和例子用 continue 可以写为

```
Function example()
    Variable i = 0,sum = 0
    do
        i += 1
        if(mod(i,2) == 0)
            continue
        endif
            sum += i
    while(i <= 100)
    Print sum
End
```

mod 是一个系统内置函数，用于计算一个整数被另外一个整数除时的余数，这里用来计算 i 被 2 除时的余数，当余数为 0 时表示 i 是偶数，否则为奇数。

5.2.5 函数

按照 Igor 规定的语法规则，将变量、命令、语句组合在一起，形成一个具有明确功能的代码块，就构成了函数。函数是 Igor 程序设计的功能单位，Igor 程序就是由一个个函数构成的。一些命令和操作，如数据拟合、方程求解等，也需要使用由用户指定的函数。

正如任何常见的编程语言，Igor 的函数概念同样涉及函数声明、修饰符、函数名、参数类型及参数表、返回类型和返回值、函数体等。请看下面的例子：

```
Function myFun(a,b)
    Variable a,b
    Variable c
    C = sqrt(a^2 + b^2)
    Return c
End
```

这里 Function 可以理解为修饰符，定义函数必须以 Function 关键字开头。除了 Function 之外，函数修饰符还包括 Static、Threadsafe，分别用于声明静态函数和线程安全函数。静态函数请参看本书第 5.3.2 节，线程安全函数请参看本书第 7.2.1 节。

myFun 是函数名，可以任意定义，但是必须满足 Igor 的命名规则，即字母＋数字＋下画线（数字和下画线不能位于首位）的要求。

a 和 b 是参数表，并用圆括号括起来。参数表里不指明参数的类型（Igor Pro7 里可以指明），而是紧跟函数后利用相应的关键字声明。注意，可以省略参数，即定义无参数函数，但是圆括号不能省略。

参数类型包括数值型、字符串型、wave 型、结构体类型等。

c 是在函数体内定义的一个数值变量，由于没有使用/G 选项，因此 c 是一个局部变量。

return 语句返回一个值，在这里将 c 值作为返回值返回给调用 myFun 的函数。注意，

函数可以没有显式 return 语句，但函数仍有返回值，返回的值及其类型取决 Function 关键字的选项参数。一个函数体内可以有多个 return 语句。当执行到 return 语句时函数会立即停止执行并返回 return 后面的值。

函数的返回类型是由 Function 的选项指定的，Function 选项如下：

```
Function [ /C /D /S /DF /WAVE ]
```

方括号表示选项是可选参数，其含义为

（1）/C，返回一个复数。

（2）/D，返回一个双精度浮点型数值，默认返回类型。

（3）/S，返回一个字符串。

（4）/DF，返回一个文件夹引用。

（5）/WAVE，返回一个 wave 引用。

当返回类型指定以后，return 语句后跟的返回值必须与返回类型相对应。例如返回类型是数值时，return 后面只能跟数值型结果（不可以是复数），返回类型是字符串时，return 后面只能跟字符串型结果。

函数在定义后就可以在命令行中直接执行或者在其他函数中调用。在命令行中调用函数的例子如图 5-18 所示。

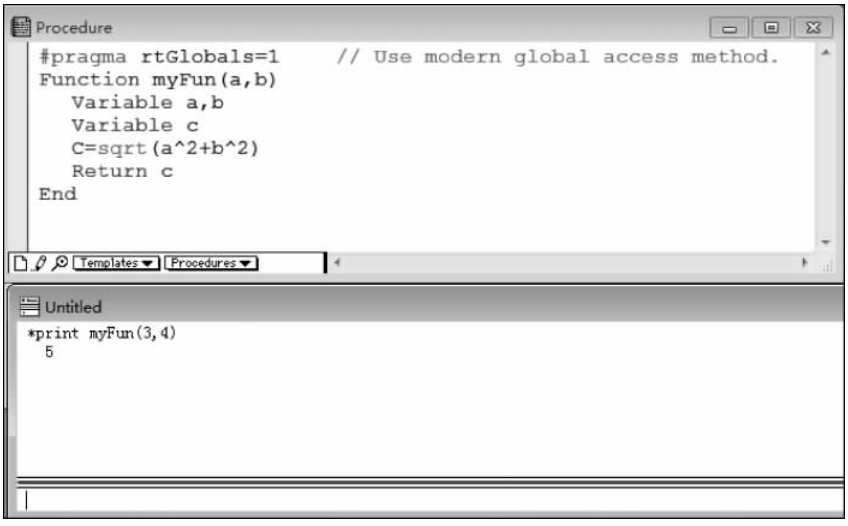


图 5-18　在命令行中调用自定义函数

在函数中调用函数的例子如图 5-19 所示。

第 2 个例子定义了一个新的函数 example，这个函数没有参数，在函数体内调用 myFun 并将返回值赋给变量 v，然后利用 print 语句将 v 输出到历史命令行窗口。同样需要在命令行窗口中执行 example()函数。注意不要忘记圆括号。

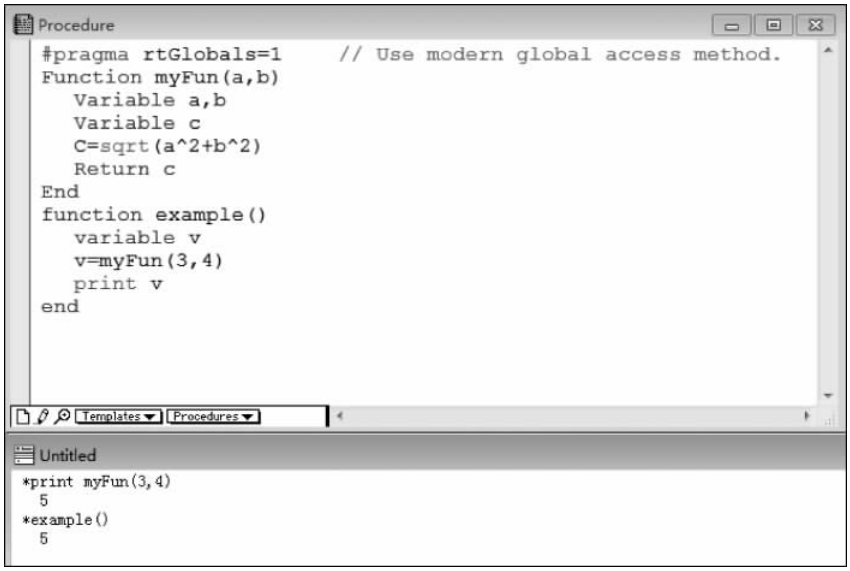


图 5-19 在函数中调用函数

再看下面的例子：

```
Function/S gettracename()
    String s,s1
    s = tracenamelist("",";",1)
    s1 = stringfromlist(0,s)
    Return s
End
```

上面的例子返回当前 Graph 中包含的曲线中第一条曲线的名字。

读者可以在命令行窗口中执行下面的命令查看上面的例子，结果如图 5-20 所示。

```
Make data1
Display data1
Print gettracename()
```

Igor 提供了大量的内置函数，如数学函数 sin()、cos()等，其功能和用户自定义函数类似，甚至完全相同，但注意用户自定义函数可以当作命令直接执行，而系统内置函数则必须位于赋值语句的右边或者是有一个变量(可能是非显式的)能接收其返回值的位置。例如直接执行 sin(1)会提示错误，而 c=sin(1)，print sin(1)等则是正确的表达式。此外，用户自定义函数必须在编译后才能使用。

5.2.6 程序子类型

为了方便管理和使用，Igor 支持对程序进行分类，方法是在程序声明后用冒号指明程序的类型：

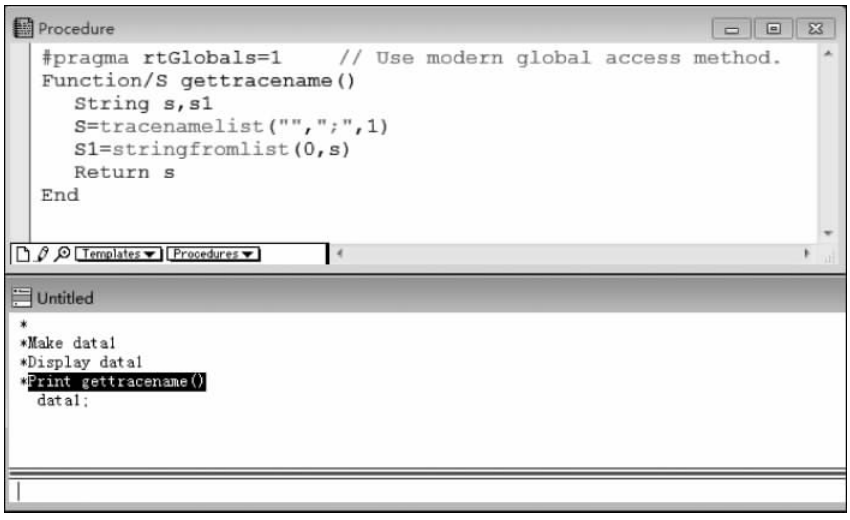


图 5-20　返回字符串的函数

```
Function functionname(<parameter list>):subtype
Window macroname(<parameterlist>):subtyp
```

Proc 和 Macro 也可以指明子类型，但常见的都是 Function 和 Window。注意，Window 本质上是一个 Macro。

subtype 是一个关键字，用于描述一个类型。当用 subtype 指明程序子类型后，编译器会记录这些类型，并在需要的场合列出这些程序供用户选择，如指定一个程序的子类型为 GraphStyle，则当用户在修改图表外观时，外观设置对话框会自动列出该程序以供用户选择。

注意，这些子类型关键字一般都是由 Igor 自动创建时加上去的，使用者一般不需要专门添加。当然如果需要，添加是可以的。手动添加和系统自动创建时添加的效果完全一样。

表 5-4 列出了子类型关键字及其含义。

表 5-4　程序子类型及其含义

子　类　型	含　　义	适用程序类型
Graph	在【Windows】|【Graph Macros】下显示	Macro
GraphStyle	在【Windows】|【Graph Macros】及其样式设置时显示	Macro
GraphMarquee	在图表选择框(Marquee)中显示	Macro/Function
Table	在【Windows】|【Table Macros】下显示	Macro
TableStyle	在【Windows】|【Table Macros】及其样式设置时显示	Macro
Layout	在【Windows】|【Layout Macros】下显示	Macro
LayoutStyle	在【Windows】|【Layout Macros】及其样式设置时显示	Macro
ListBoxControl	给 Listbox 控件指定回调函数时显示	Macro/Function

续表

子 类 型	含 义	适用程序类型
Panel	在【Windows】|【Panel Macros】下显示	Macro
FitFunc	在选择拟合函数时显示	Function
ButtonControl	给按钮控件指定回调函数时显示	Macro/Function
CheckBoxControl	给复选框控件指定回调函数时显示	Macro/Function
PopupMenuControl	给弹出式菜单控件指定回调函数时显示	Macro/Function
SetVariableControl	给文本框控件指定回调函数时显示	Macro/Function

GraphMarquee 关键字提供了一种非常好的用户交互实现方法：只需要给函数指明 graphmarquee 子类型，该函数就会出现在【Graphmarquee】菜单中。看下面的例子，结果如图 5-21 所示。

```
Function mymarqueefun():graphmarquee
    getmarquee
    Print V_left,V_right,V_top,V_bottom
End
Make/O data = x
Display data
```

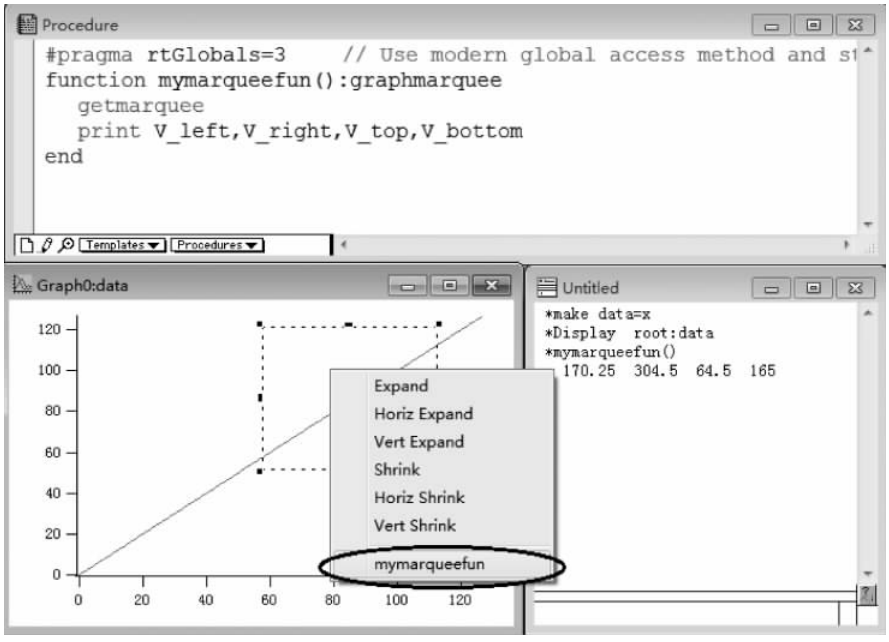


图 5-21 利用 GraphMarquee 关键字将函数添加到【Graphmarquee】菜单

当然还有给【Graphmarquee】菜单添加菜单项的其他方法，但是这种方法是最简单的。

5.2.7 参数传递

Igor 中，函数参数传递有两种方式：按值传递和按引用传递。

按值传递表示调用函数将变量的值传递给被调用函数，按引用传递表示调用函数将变量的引用传递给被调用函数。这类似于 C 语言中的传递值和传递地址，前者不能修改原变量的值，后者能修改原变量的值。

来看一个按值传递的例子：

```
Function mainFunc()
    Variable v = 1
    String s = "hello"
    subroutine(v,s)
    Print v,s
End
Function subroutine(v,s)
    Variable v
    String s
    Print v,s
    v = 2
    s = "hello,IGOR"
End
```

在上面的例子中，函数 mainFunc 调用 subroutine，将变量 v 和 s 按值传递给 subroutine 的两个参数 v 和 s，subroutine 打印 v 和 s 并修改了 v 和 s，由于是按值传递，subroutine 中的修改并不会影响 mainFunc 中的 v 和 s。在这里，v 和 s 是各自函数的局域变量，尽管名字相同，但是二者没有任何关系，读者可以将 subroutine 的参数起一个不同的名字，效果完全一样。

接下来看一个按引用传递的例子。

```
Function mainFunc()
    Variable v = 1
    String s = "hello"
    subroutine(v,2,s)
    Print s,v
End
Function subroutine(num1.num2,s)
    Variable &num1,num2
    String &s
    num1 = num1 + num2
    S = "The sum of the two num is"
    Print s,num1
End
```

在上面的例子中，函数 subroutine 有 3 个参数，其中第 1 个参数和第 3 个参数按照引用传递，由于 subroutine 获取的是被调用函数的变量的引用（地址），所以可以修改被调用函数中变量的值。在命令行中执行 mainFunc()，结果如图 5-22 所示。

```
*mainFunc()
  The sum of the two num is  3
  The sum of the two num is  3
```

图 5-22　按引用传递的结果

如果是按值传递，则应该输出

```
Hello 1
The sum of the two num is 3
```

传递引用的语法格式非常简单，只需在声明函数参数时名称前面添加“&”符号即可，和 C 语言中引用的声明方式一样。结构体变量在作为参数传递时同样适用这种声明方式。

对于使用引用变量作为参数的函数，在调用时还可以将引用变量作为参数，如

```
Function f1(v)
    Variable &v
    f2(v)                    //v 是一个变量的引用,传递给 f2
End
Function f2(v)
    Variable &v
End
```

请注意，除了用于函数参数之外，不能直接使用“&”符号定义一个变量的引用，如下面的语句是错误的，这点和 C 语言不一样。另外，也不能将全局变量作为引用传递。

```
Variable v1
Variable &v = v1
```

引用的主要目的是一次返回多个函数值。一般而言，同全局变量一样，除非必要，不要使用引用传递参数，否则会增加程序之间的耦合程度，增大由于对某一函数中变量的修改而导致错误出现的概率，通常这些错误难以预测。

如果函数的参数是 wave，则传递方式全部为按引用传递，这和 C 语言完全类似。看下面的例子：

```
Function routine()
    Make/O wave0 = x
    subroutine(wave0)
End
Function subroutine(w)
    Wave w
    W = 1234
End
```

上面的例子中，传递给 subroutine 的参数是 wave0 的引用，而非 wave0 里的值。因此在 subroutine 里对 w 修改就是对 wave0 的修改。

引用实际上也是一个变量。因此对于 wave 作为参数的情况,“引用变量”本身是按值传递,而 wave 则是按引用传递。在 subroutine 中对引用变量本身的修改不会影响调用函数中引用变量。如在 subroutine 中执行

```
Waveclear w
```

原函数中对 wave0 的引用(就是 wave0 本身)并不会被清除,请读者注意体会。

5.2.8 默认参数

很多编程语言支持默认参数。默认参数有两个好处:支持多态性,可以通过提供不同的参数调用同一个函数,方便记忆;简化代码,开发者无须为相同的功能编写重复的代码。Igor 支持默认参数,定义如下:

```
Function f(p1,p2,[p3,vout,s1,w1])
    Variable p1,p2
    Variable p3
    Variable &vout
    String s1
    Wave w1
End
```

这里 p3、vout、s1、w1 都是默认参数。默认参数需要用中括号括起来,可以是任意的数据类型,包括结构体类型,还可以是变量的引用。在调用带有默认参数的函数时,使用“变量名=值”的方式提供默认参数值,变量名就是定义中使用的名字。可以只提供部分默认参数。如上面的函数可以按照下面的方式调用:

```
f(1,2)
f(1,2,p3 = 3)
f(1,2,vout = v) //v 是一个数值变量
f(1,2,p3 = 3,s1 = s1) //s1 是一个字符串变量
f(1,2,w1 = w) //w 是一个 wave 引用
```

如果不提供默认参数值,数值型变量默认取 0。字符串变量为 null 字符串。注意 strlen 函数以 null 字符串作为参数时返回 null 而非 0。wave 引用默认为 null wave,即不指向任何 wave。

在程序中可以使用 ParamIsDefault 函数判断默认参数是否提供,如果没有提供可以在程序中提供合理的默认值。ParamIsDefault 以默认参数名为参数,当默认参数没有提供时,ParamIsDefault 返回非 0 值,否则返回 0。

```
if(paramisdefault(p1))
    p1 = 1
endif
```

看下面的例子:

```
Function maxvalue(a,b,[c])
    Variable a,b,c
    if(paramisdefault(c))
        Return max(a,b)
    else
        return max(max(a,b),c)
    endif
End
```

在命令行窗口输入：

```
Print maxvalue(1,2)
```

结果如图 5-23 所示。

图 5-23　默认参数函数调用示例

输入：

```
Print maxvalue(1,2,c = 3)//注意变量名 c 不能省略
```

结果如图 5-24 所示。

图 5-24　默认参数函数调用示例

5.2.9　注释和代码风格

良好的注释有助于程序的可读性，有助于理解程序代码，也有助于对程序的升级和维护。注释在程序调试中也很有用。Igor 下的程序注释类似于 C 语言，使用“//”作为注释符号，跟在“//”后面同一行的内容将视作注释内容。不同于 C 语言里的“/ * … * /”注释符号，Igor 里没有能注释多行的注释符，需要注释多行时可以在每一行行首打入“//”符号。

Igor 提供了一个快捷的多行注释的方法，选择要注释的多行代码，然后执行菜单命令【Edit】|【Commentize】可以自动在每一行行首添加注释符，从而实现一次注释多行代码。如果同时取消多行注释可以执行菜单命令【Edit】|【Decommentize】。

注释一定要有意义，不要为了注释而注释，比如下面的注释就完全没有必要：

```
//计算 a + b
c = a + b
```

一般应该在关键函数(如通用性非常高的函数)前面添加对函数功能的描述性注释,对函数参数的含义进行注释,如

```
//这个函数用来拟合×××格式的数据
Function lorf(x,cw):Fitfunc
    Variable x                    //待拟合数据的 x 坐标值
    Wave cw                       //拟合参数(初始参数和拟合结果都存放在这个 wave 里)
    //cw[0]:                      拟合参数说明
    //cw[1]:                      拟合参数说明
    …
    <function body>
End
```

良好的代码风格能使程序结构清晰,提高程序可读性,避免出错。特别是对于语句的多层嵌套,恰当的缩进显得尤为必要。在编写程序时应该有意识的使用缩进,如 if 语句可以缩进为如下格式:

```
if(a>b)
    Num_max=a
else
    Num_max=b
endif
```

if 里语句块的内容相对于 if 关键字缩进一个制表符,这样程序看起来结构清晰,层次分明。

【Edit】|【Adjust Indentation】可以自动调整缩进,方法是先选择要调整缩进的内容,然后执行该菜单命令。

5.3 程序设计技术

前面详细介绍了 Igor 下程序设计的基本方法,利用这些知识已经可以编写具有实用性的程序。但是要使得程序功能强大,应用范围更广,则必须利用 Igor 提供的其他编程特性和技术。这些技术有些为 Igor 所特有,有些则借鉴了其他优秀编程语言的特点。掌握这些技术是掌握 Igor 程序设计的必需步骤,和掌握基本语法一样重要。建议读者仔细阅读,并在实践中加以应用。

5.3.1 Include 指令

Include 指令用于将另外一个程序文件包含在当前程序文件之中。通常将具有类似功能或者通用功能的一组程序存放于一个程序文件之中,并将之另存为一个后缀名为 ipf 的程序文件。在后续的程序开发中,当需要已经编写好的程序时,就可以通过 Include 指令直接将包含该程序的文件包含进来,而不需要重新编写相同的代码。通过这种方法也可以很方便地使用他人已经写好的程序。Include 指令是 Igor 下程序设计最常使用的编译器指令,必须熟练掌握。

Include 指令的语法格式非常简单：

```
#Include <procedurefilename>
#Include "procedurefilename"
```

“#”号是必需的，且必须位于行首。#Include 可以位于程序文件的任何地方，但是一般都位于程序文件的开头位置。procedurefilename 是要被包含的程序文件名，该程序文件必须以 ipf 作为后缀名，但注意 procedurefilename 不包括“.ipf”，如被包含的程序文件全名为 somename.ipf，则 procedurefilename 应为 somename 而不是 somename.ipf。编译器会自动打开 Include 指令所指定的文件，一般可以在【Windows】|【Procedure Windows】里查看已经打开的程序文件。如果取消了 Include 指令，则被打开的程序文件将会在下一次编译的时候自动关闭。

尖括号、引号、procedurefilename 的具体形式表示了 Include 指令获取程序文件位置的 4 种情况：

1. #Include <filename>

这条指令用于包含一个名为 filename.ipf 的程序文件，该程序文件必须位于 Igor Folder/WaveMetrics Procedures 及其子文件夹下。这里“Igor Folder”指 Igor 安装目录（下同）。如没有专门强调，下面说的文件夹都是相对于 Igor 安装目录的。

WaveMetrics Procedures 文件夹位于 Igor 安装目录下，里面存放了大量由 WaveMetrics 提供的 ipf 程序。每个程序文件里都包含了一系列具有实用价值的的通用函数，程序文件名字描述了这些函数的大致功能，如 Image Common.ipf 里面包含了大量关于图像处理的通用函数，这些函数可以获取、创建和删除图像等。这些程序文件是在 Igor 下进行程序设计和开发的最佳范例。

当要写一个关于图像处理的程序时，就可以通过下述指令将 Image Common.ipf 包含进来，这样就不用再重复编写一些通用的函数代码，指令如下：

```
#Include <Image Common>
```

2. #Include “filename”

和第一种情况相比较，这里尖括号变成了双引号，用于包含一个名为 filename.ipf 的程序文件，该程序文件必须位于 User Procedures 目录或者 Igor User Files/User Procedures 目录及其子目录下。Igor User Files/User Procedures 位于我的文档下，是 Igor 在安装时自动创建的一个文件夹。可以通过菜单命令【Help】|【Show Igor User Files】打开该文件夹。

User Procedures 和 Igor User Files/User Procedures 专门用于存放由用户开发的程序文件。到底存放在哪个目录下，则看个人习惯，推荐存放在 Igor User Files/User Procedures 中。当然不管存放于何处，对程序文件的备份都是必需的。

假如要包含的程序文件名为 proc0.ipf，则包含指令为

```
#Include "proc0"
```

3. Include "full file path"

前面两种情况要求被包含的程序文件必须位于特定的目录下,这并不是必需的。可以用双引号指定程序文件的完整路径,此时程序文件可以位于任何地方。例如,程序文件位于E盘下的tmp文件夹中,文件名为myproc.ipf,则包含指令可以为

```
#Include "E:tmp:myproc"
```

注意,这里文件路径格式是Igor下的文件路径格式,也可以使用E:\tmp\myproc这样的格式。

4. Include "patial file name"

除了指定程序文件的绝对路径,还可以指定程序文件的相对路径,此时Igor首先从User Procedures文件夹中寻找,如果失败再从Igor User Files/User Procedures文件夹中寻找,如果再失败则从当前使用#Include指令的程序文件所在的文件夹中寻找。

假如当前程序文件位于E:\tmp\myproc.ipf,要包含的程序文件位于E:\tmp\categary1\proc1.ipf,则包含指令为

```
#Include ":categary1:proc1"
```

如果要包含的程序文件位于E:\tmp\proc1.ipf,即和主程序文件myproc.ipf位于同一个目录下,则包含指令为

```
#Include":proc1"
```

这里冒号不能省略,否则会导致错误。

最常用的Include方法是第2种,第3种和第4种。由于包含路径信息,当程序文件的路径信息不小心被改变(如程序从一台计算机复制到另外一台计算机)时就会出现找不到文件而无法编译的错误,这种情况是非常容易出现的。因此除非必要,不建议读者采用后两种方法。

使用第2种方法,只需要将程序文件放入User Procedures或Igor User Files/User Procedures文件夹中,就可以通过程序文件名方便地包含该文件。

5.3.2 Pragma参数

在每一个程序编辑窗口前几行都有一个#Pragma的参数,如图5-25所示。

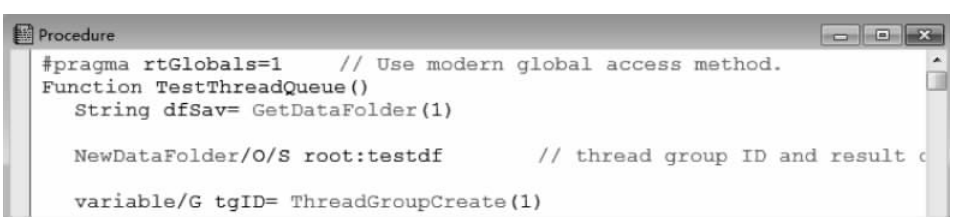


图5-25 Pragma参数

这个参数的作用是设定编译器模式或者向编译器传递一些有用的信息。

#Pragma 参数一般位于程序文件之首，且#号不能与窗口左侧存在任何空格或者是制表符，其声明形式如下：

```
#Pragma keyword [ = parameters ]
```

目前，Igor 支持以下 6 种 Pragma 参数(Igor Pro 6.37 及以前)：

```
#pragma rtGlobals = value
#pragma version = versionNumber
#pragma IgorVersion = versionNumber
#pragma ModuleName = name
#pragma hide = 1
#pragma IndependentModule = name
```

#Pragmas 参数只影响它所处的程序文件。下面介绍每一个参数的具体含义。

1. rtGlobals 参数

rtGlobals 参数用于指定编译器的行为和程序运行时对错误的处理方法。

rtGlobals 参数取值为数字 0、1、2、3，其含义与 Igor 的发展有关。早期的 Igor 版本(早于 Igor 3，本书 Igor 版本新于 6.37)，在函数中访问一个全局变量或者 wave 时，要求全局变量和 wave 必须在编译阶段就存在，否则无法编译。在 Igor 3 版本中引入了“运行时查找变量”的技术，即在编译阶段，编译器并不要求全局变量和 wave 存在，只是在运行时才将全局变量和与其引用(wave、nvar、svar)关联起来。在 Igor 6.20 以后又引入了对 wave 引用和 wave 长度的严格检查。rtGlobals 不同取值就是反映了 Igor 这种版本演化的行为，其主要目的是实现不同版本之间的兼容性。

```
#pragma rtGlobals = 0
```

这是 Igor 最早期版本的编译器模式，不应该继续使用。

```
#pragma rtGlobals = 1
```

打开“运行时查找变量”技术，使用此选项，编译器在编译时并不检查被引用的全局变量是否存在，一般是默认选项，本书中绝大多数例子都是在此模式下编写的。

```
#pragma rtGlobals = 2
```

强制使实验文件处于“非兼容模式”，也是为了与旧版本 Igor 程序设计兼容的一个编译器选项，几乎很少使用，无须理会。

```
#pragma rtGlobals = 3
```

打开“运行时查找变量”技术，使用严格 wave 引用，运行时检查 wave 的长度，越界访问报错。

当#pragma rtGlobals=3 时，对 wave 引用的检查非常严格，任何使用 wave 的场合都

必须通过 wave 引用，换言之，要使用 wave，该 wave 引用必须存在，否则编译器就会报错。如下面的例子：

```
Function test()
    Display wave1
End
```

上面的代码在 rtGlobals=3 时不能通过编译，否则会出现图 5-26 所示的错误。

图 5-26 rtGlobals=3，wave 必须声明引用后才能使用

这是由于 wave1 并没有事先声明。正确的做法是

```
Function test()
    Wave wave1
    Display wave1
End
```

即先声明 wave1 的引用，然后才能使用该 wave。

但是下面的情况编译时不会报错：

```
Function test()
    Make wave1
    Display wave1
End
```

这是因为 Make 命令在创建新 wave 时会自动创建 wave 引用，请读者参看本书第 5.3.10 节自动变量。

当 #pragma rtGlobals=1 时，编译器对 wave 引用的检查是不严格的，除非是赋值语句，否则可以不声明 wave 的引用而直接使用 wave，如下面的代码在 #pragma rtGlobals=1 是正确的：

```
Function test()
    Display wave1
End
```

如果是赋值语句，则无论是 rtGlobals=1 还是 rtGlobals=3，都必须声明 wave 的引用，如下：

```
Function test()
    Wave wave1
    Wave1 = sin(x)
    Display wave2                    // rtglobals = 1 时正确，rtglobals = 3 时错误
End
```

这里需要给 wave1 赋值，因此必须声明 wave1 引用，而 wave2 作为 Display 的参数，在 rtGlobals=1 时不需要声明 wave 引用，在 rtGlobals=3 时需要声明 wave 引用。

rtGlobals=1 和 rtGlobals=3 对 wave 越界访问的处理也不一样。rtGlobals=1 时对 wave 的越界访问并不会报错，如

```
Make data1 = {1,2,3,4,5}
data1[6] = 7
```

这里 data1 的长度为 5，显然 data1[6]并不存在，在 C 语言这会导致内存非法访问错误，产生不可预料的结果。但是在 Igor 下，上面的赋值语句会被自动截断为 data1[4]=7，即当越界访问时，只访问 wave 的最后一个元素。当然这不是绝对的，如果指定了错误的 Label，访问的是第一个元素，由于 Label 很少使用，这里就不做介绍了，感兴趣的读者可以查看 SetDimLabel 命令的帮助。

如果 rtGlobals=3，上述代码在运行时仍然会有截断赋值行为，但是会报一个越界访问的错误，以引起使用者关注。

2. version 参数

version 参数用于指明当前程序文件的版本号，如

```
#pragma version = 1.01
```

如果不指定则版本号默认为 1.00。版本号应该在程序文件的开头指定。

指明程序的版本号有助于程序的升级和维护，特别是当程序由多人开发时，版本号能方便程序的开发管理。

Igor 的 Include 指令会检查程序文件的版本号，如

```
#Include "a procedure file" version>=1.03
```

表示被打开的程序文件版本号应该不低于 1.03，否则会显示一个警告信息，提示程序需要更新(但是程序还是会被正常包含和编译的)。

3. IgorVersion 参数

IgorVersion 参数用于指明程序在哪个版本的 Igor 下运行，如

```
#pragma IgorVersion=5.03
```

上述指令表示当前程序文件要求 Igor 的版本不低于 5.03。Igor 版本更新很快，每一个版本

都会增加一些新的特性，其编程环境也会发生变化，如引入新的语法、函数等。旧版本 Igor 下的程序很有可能在新版本下无法通过编译，反之亦然。为了让程序正常运行，指定 IgorVersion 参数是必要的。

4. ModuleName 参数

ModuleName 参数类似于 C++ 或是 C# 等编程语言中的名称空间，通过指定 ModuleName 参数，相当于给程序文件起了一个名字，可以通过该名字访问程序文件里的程序和变量。ModuleName 的声明方式如下：

```
#pragma ModuleName = name
```

其中，name 是一个合法的名字，注意不要和其他的 ModuleName 相冲突。当声明了一个程序文件的 ModuleName 参数以后，就可以通过该名字访问程序文件内的对象，方法是 name#对象名，这里的#号类似于 C++或者 C#里的“.”运算符，如下：

```
#pragma ModuleName = myGreateProcedure
Static function foo(a)
    Variable a
    Return a + 100
End
```

这时，可以在命令行窗口执行：

```
myGreateProcedure#foo(0)
```

使用 ModuleName 参数的主要作用是避免命名冲突。前面介绍函数声明时只介绍了 Function 关键字，其实 Function 还有一个修饰符 static。通过 static 声明的函数是私有函数，只属于所处程序文件本身，只能在程序文件内使用，不能在程序文件外直接使用。如上面的例子，在命令行窗口直接输入 foo(0)是无法运行的，Igor 提示找不到该函数。要在程序文件外访问私有函数，必须通过 ModuleName 进行，格式为 Modulename#函数名。

不同 ModuleName 的程序文件中，私有函数是可以重名的，如图 5-27 所示。

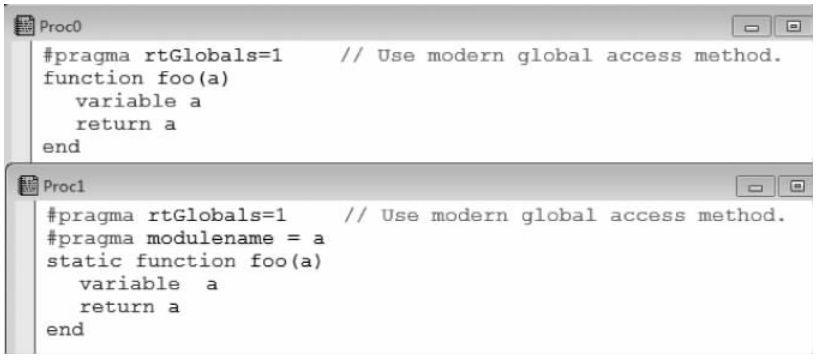


图 5-27　ModuleName 不同的程序窗口中，函数可以重名

这里 ModuleName 为 a 的程序窗口中有一个私有函数 foo，在内置程序窗口（全局程序窗口）中有一个函数也是 foo，由于 Modulename 为 a 的程序窗口中 foo 被声明为私有函数，所以它可以和内置程序窗口中的 foo 重名。

在开发过程中，如果函数和已有的函数发生名字冲突时，就可以给程序文件指定一个独立的 ModuleName，并将可能冲突的函数通过 static 声明为私有。这在多人开发或者项目比较大时非常有用。

如果不使用 static，则函数是公有的，此时所有程序窗口包括命令行都可以通过函数名访问该函数，这是本节以前所有例子的情况。所有的公有函数名字必须独一无二，即使是位于不同的程序文件夹中。

这里的 static 应该借鉴 OOP 程序设计语言中静态成员的概念，静态表示该函数属于类本身，只能通过类（这里就是程序文件 Module 名字）访问。

可以将不同的程序文件声明为相同的 ModuleName，但是没有必要这样做，因为用不同 ModuleName 的目的就是为了避免命名冲突，而不是提供一个公共的名称空间。

如果省略 ModuleName，Igor 会自动给程序指定一个 ModuleName——ProcGlobal。换句话说，没有声明 ModuleName 的所有程序文件都位于 ProcGlobal 环境下。在 ProcGlobal 环境下，也可以使用 static 函数声明私有函数，但是该私有函数将无法在程序文件外部访问，只能在程序文件内部访问。读者可能想通过“ProcGlobal＃私有函数名”访问，这也是不可以的。要从外部访问一个程序文件里的私有函数，必须给该程序文件声明一个 ModuleName。最后，请读者注意，除了函数之外，static 还可以作用于 constant、structure 对象。一旦用 static 声明，这些对象即被所处程序文件私有，在程序文件外访问时只能通过 ModuleName 加“＃”的格式访问。

5. hide 参数

＃Pragma hide＝1 可以隐藏程序文件，此时在菜单【Windows】|【Procedure Windows】下看不到对应的程序文件。

6. IndependentModule 参数

IndependentModule 参数使用比较复杂，也很有用，请看下一节 5.3.3。

5.3.3 IndependentModule

IndependentModule 参数用于将程序文件设置为独立模块（IndepentModule）。被指定为 IndependentModule 的程序文件，里面的代码将会单独编译。此时即使其他普通程序文件里的代码没有编译（如发生语法错误），独立模块中的代码仍然处于已编译状态，可以运行。利用独立模块可以编写需要在任何时候运行的代码，如数据采集程序、高通用性程序等。

将一个程序文件指定为独立模块的方式非常简单：

```
＃pragma IndependentModule = imName
```

注意，内置的程序窗口不允许声明为独立模块，即内置程序窗口不能被指定为独立模块。下面看一个独立模块的简单例子：通过菜单命令【Windows】|【New】|【Procedures】创建一个程序文件，在程序文件的开头输入独立模块预编译指令，如图 5-28 所示。

```
#pragma IndependentModule = myIM
```

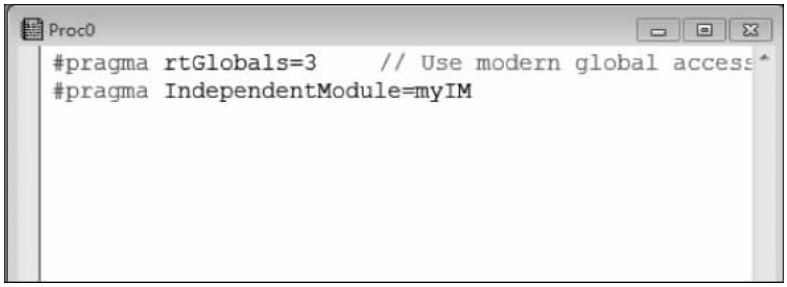


图 5-28　将程序窗口声明为独立模块

在程序窗口里输入以下代码：

```
Function test()
    Print "this function executes under an Independent Module"
End
```

在命令行窗口执行指令，如图 5-29 所示。

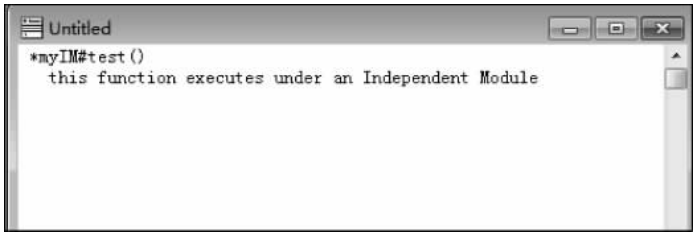


图 5-29　执行独立模块中的函数

按 Ctrl＋M 键，打开内置程序窗口，人为制造一个错误，使程序无法编译，如图 5-30 所示。

单击【Quit Compile】取消编译，然后在命令行窗口输入以下命令：

```
myIM#test()
```

可以发现尽管有程序没有编译成功，但是 myIM 中的 test 函数仍然可以正常运行。这说明 myIM 模块中的代码和其他正常模块中的代码是独立编译的。

下面对独立模块程序设计进行更加详细的介绍。

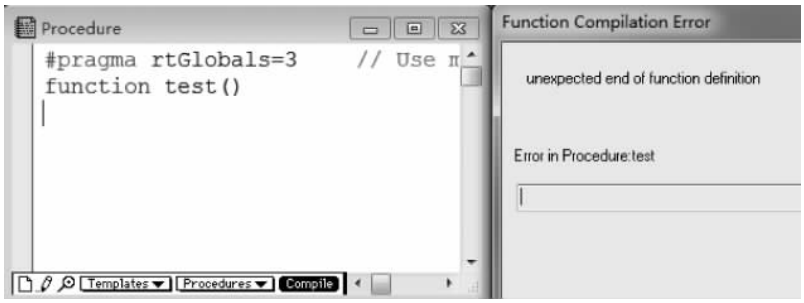


图 5-30 人为制造编译错误,独立模块仍正常编译

1. 独立模块中的程序类型

独立模块中只能定义函数,不能定义 Macro 和 Proc 类型的程序,但是可以定义 Window 类型,也就是说可以在独立模块中放置窗口生成代码(Window Recreation Macro)。

2. 函数定义和访问

独立模块中的函数同样可以定义为公有的或者私有的。公有函数的定义方法为直接使用 Function 关键字定义函数,私有函数则在 Function 前面加 static 关键字。如下:

```
#pragma IndependentModule = myIM
Function test()
    f1()
End
static Function f1()
    print "Independent Module"
End
```

独立模块的函数的访问层次比普通模块低一个级别。访问公有函数需要在该函数名字前面添加独立模块名作为前缀,如 myIM#test()。私有函数外界完全无法访问,只能由独立模块内部的代码访问:

```
myIM#test()          //可以执行
myIM#f1()            //不能执行
```

3. 程序文件不可视

独立模块的程序文件默认不可见,在【Windows】|【Procedure Windows】菜单里是看不到独立模块对应的程序文件的。因此独立模块的程序在编译并选择隐藏后,是无法通过【Windows】|【Procedure Windows】找到的,只能通过诸如查找、函数帮助、【Go To】命令等方式定位并重新打开程序文件。如果程序处于开发阶段,这种特性会带来不便,可以通过以下命令突破这个限制,结果如图 5-31 所示。

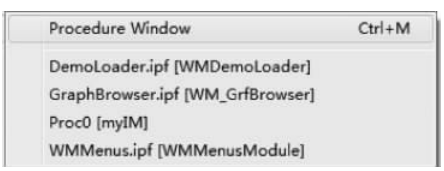


图 5-31 使 Independent Module 重新出现在【Procedure Window】菜单里

```
SetIgorOption IndependentModuleDev = 1
```

如图 5-31 所示，myIM 独立模块已经出现在【Procedure Window】列表里。注意独立模块的命名方式，在程序里访问独立模块时必须按照这个格式去访问：

```
文件名[独立模块名]     //文件名和模块名之间的空格不能省略
```

独立模块程序文件被隐藏起来是合理的，这样可以防止被误修改。因此在开发阶段可以使用 SetIgorOption IndependentModuleDev＝1 使程序文件可见，而在开发完后则使用 SetIgorOption IndependentModuleDev＝0 隐藏独立模块程序文件。

4. 文件包含

在独立模块里可以使用 Include 指令包含其他的程序文件，被包含进来的程序文件自动转换为独立模块，Igor 会在每个包含进来的程序文件头添加＃pragma IndependentModule 指令并自动将名字指定为该独立模块名字。如独立模块为 myIM，则被包含的程序文件开头会自动添加如下的预编译指令：

```
#pragma IndependentModule = myIM
```

这样就可以自由访问被包含文件里的函数了。

注意，被包含进来的程序文件为一份临时复制文件，对这些程序文件的任何修改都不会被保存。因此不要对这些程序文件做任何修改，如果做了修改，一定要以别的方式将这些内容保存到原程序文件。

独立模块不能包含独立模块程序文件，除非该程序文件具有相同的独立模块名字。

5. 窗口程序

在独立模块里同样可以设计窗口程序，但是要注意窗口程序所有的代码都必须位于独立模块内，不要调用独立模块外的自定义程序。关于窗口程序设计请参看第 6 章。

6. 菜单

在独立模块里创建菜单时，如果菜单项对应的函数位于独立模块，最好在函数前面添加独立模块名作为前缀。如果是给弹出式菜单设置菜单项，且菜单项由独立模块里的一个函数返回，则必须在该函数名前加独立模块名前缀，如图 5-32 所示。

完整的程序代码如下：

```
#pragma rtGlobals = 3
#pragma IndependentModule = myIM
menu "macros"                          //创建两个菜单
    "my menu item", myIM#f1()          //指定"my menu item"对应的函数
    "create window",myIM#f2()          //指定"create window"对应的函数
End

Function f1()
    Print 1
End
```

Proc0

```
#pragma rtGlobals=3     // Use modern global access method and strict
#pragma IndependentModule=myIM
menu "macros"
   "my menu item", myIM#f1()
   "create window",myIM#f2()

end
function f1()
   print 1
end

function f2()
   execute "panel0()"
end

Window Panel0() : Panel
   PauseUpdate; Silent 1      // building window...
   NewPanel /W=(150,77,412,164)
   PopupMenu popup0,pos={21,17},size={100,20},bodyWidth=100
   PopupMenu popup0,mode=1,popvalue="Yes"
   PopupMenu popup0,value= #GetIndependentModuleName()+"#popuplist()"
EndMacro

function/S popuplist()
   return "items;item2"
end
```

图 5-32　独立模块中菜单的设计

```
Function f2()
    execute "panel0()"
End

Window Panel0() : Panel
    PauseUpdate; Silent 1
    NewPanel /W = (150,77,412,164)
    PopupMenu popup0,pos = {21,17},size = {100,20},bodyWidth = 100
    PopupMenu popup0,mode = 1,popvalue = "Yes"
    PopupMenu popup0,value =  #GetIndependentModuleName() + "#popuplist()"
EndMacro

Function/S popuplist()                    //返回 Panel0 中下拉菜单 popupo 中的菜单项
    return "items;item2"
End
```

上面的代码在独立模块里给【Macros】添加了两个菜单项，第一个菜单项打印数字 1，第二个菜单项能打开一个窗口，该窗口包含一个弹出式菜单控件，控件的菜单项由 value 关键字指定，函数 popuplist 返回一个字符串列表给 value 关键字，以设置菜单项。注意，在

popuplist 前添加了 GetIndependentModuleName()函数以获取独立模块名，因此最后的调用是

```
myIM#popuplist()
```

5.3.4 Execute 命令

有些命令不能在函数里执行，如函数里不能直接调用 Proc 或者 Macro 程序。如果要在函数里调用 Proc 和 Macro，需要使用 Execute 命令。如下：

```
Function test()
    Execute "f1()"
    //由于 f1()是一个 proc,在函数中无法直接调用
End
Proc f1()
    Print 1
End
```

上面的例子中，f1 是一个 Proc 类型的程序，因此不能直接在函数里调用，但是可以通过 Execute 来执行。

窗口生成脚本一般都不是 Function，如果在函数里直接调用就会出错，此时可以使用 Execute 命令。有人会认为将脚本修改为 Function 就可以了，这样做没问题，但是会给以后修改窗口带来麻烦。因为生成脚本一般都是由 Igor 自动生成的，如果修改了生成脚本的名字，如将 Window 关键字改成 Function，Igor 在更新脚本时就会因为找不到该脚本，而将更新变成了重新生成。

Execute 后面还可以跟一个字符串，字符串里是待执行的命令，这个命令可以是一个函数、一个脚本，或者一个可以执行的任意合法表达式：

```
String cmd
sprintf cmd, "GBLoadWave/P = %s/S = %d \"%s\"", pathName, skipCount, fileName
Execute cmd
```

Execute 有一个/P 参数非常有用，它表示在没有任何程序或者函数运行时，再运行其后字符串指定的命令。一些命令在函数运行时没有任何效果，如 Dowindow/R 在函数运行时不能自动创建或者更新生成脚本，如果想要通过编程控制更新生成脚本，可以按照如下方式：

```
Execute/P "dowindow/R PanelName"
```

利用 Execute/P 还可以自动包含程序文件：

```
Execute/P "INSERTINCLUDE <Peak Functions>"
```

利用这个特性，可以让程序变得更加智能，如自动检查是否需要某些功能，并智能加载包含这些功能的程序文件。

5.3.5 条件编译

Igor 编译器支持条件编译，通过设定编译条件可以有选择地编译某些代码段。在编译和调试阶段利用条件编译可以将正常代码和调试代码区分开来，方便程序开发。条件编译的语法格式和说明如下：

(1) 格式 1：

```
#define symbol
#undef symbol
```

#define 用以定义一个符号标记，#undef 可取消一个符号标记定义。

```
#ifdef symbol
    <code1>
#else
    <code2>
#endif
```

如果前面利用#define 定义了 symbol，则编译代码段 code1，否则编译代码段 code2。#else 可以省略。

(2) 格式 2：

```
#if expression
    <code1>
#else
    <code2>
#endif
```

expression 是一个合法的表达式，但不能包括用户自定义的任何函数或者变量(因为这些函数和变量还没有编译)，当 expression 计算结果绝对值大于 0.5(小于 0.5，四舍五入为 0，相当于假)则编译代码段 code1，否则编译代码段 code2。#else 可以省略。

(3) 格式 3：

```
#if expression1
    <code1>
#elif expresion2
    <code2>
#else
    <code3>
#endif
```

可以利用#elif 关键字引入多种条件的判断，注意这里的关键字是#elif 而不是#elseif。

(4) 格式 4：

```
#ifndef symbol
    <code1>
#else
    <code2>
#endif
```

和#ifdef 刚好相反。

注意，#define 关键字的作用仅仅是用来声明一个用于条件编译的符号，除此之外没有任何含义，#define 并不是宏定义，Igor 里也没有宏展开的概念。这和 C 语言完全不一样，在 C 语言中利用#define 预定义的符号可以当作一个常量使用，Igor 里是不能的。

一般一个程序文件里的符号标记仅在这个程序文件里有效，但是在主程序文件里声明的符号标记所有程序文件都可见。如果一个程序文件通过#include 指令包含了一个新的程序文件，则这个程序文件对于被包含的文件就是主程序文件。

不同于#pragma 关键字，用于条件编译的符号“#”可以有缩进而无须紧靠程序窗口最左侧。

在 Igor 中预定了一些全局符号，这些符号不需要使用#define 定义就可以使用，如表 5-5 所示。

表 5-5 Igor 预定义的全局符号(#define)

符号标记	含 义
MACINTOSH	Igor 运行于苹果系统环境下
WINDOWS	Igor 运行于 Windows 环境下
IGOR64	Igor 是 64 位版本

来看下面的例子：

```
#ifdef WINDOWS
Function f()
    Print "I am running on windows."
End
#endif
#ifdef MACINTOSH
Function f()
    Print "I am running on Mac OS."
End
#endif
```

如果在 Windows 操作系统下运行 Igor，并执行函数 f()，则会输出

```
I am running on windows.
```

5.3.6 函数引用

在C语言中,可以定义指向函数的指针,通过将函数名赋值给指针变量就可以利用指针变量访问函数了。函数指针的概念使得用户在设计程序时不需要知道要执行的程序是什么,到运行时再将实际的程序指针传递给程序。如Windows程序设计里窗口过程函数就是典型的例子:窗口类维护一个指向窗口过程的函数指针,用户在创建窗口时将自定义的窗口过程指定给该指针。这给程序的设计带来了极大的灵活性。

Igor同样支持这种程序设计特性。在Igor中可以定义函数的引用,函数引用就类似于C语言中的函数指针。可以将一个函数名赋值给函数引用,之后就可以通过函数引用调用被引用的函数。

声明一个函数引用的语法如下:

```
Funcref myprotofunc f = functionname
```

Funcref是关键字,f是函数引用的名字。myprotofunc是一个用户自定义的模板函数,有两个含义:限定函数所能引用的函数类型,被引用函数的返回类型和参数必须与模板函数匹配,这和C语言类似,C语言里声明函数指针时同样需要指定返回类型和参数表;当f没有指向一个合法的函数时系统默认调用的函数,如f引用的函数不存在时系统就会调用myprotofunc函数。functionname是所要引用的函数名字。看下面的例子:

```
Function f0()
    Print "Naive!"
End
Function f1()
    Print "You get it!"
End
Function test(fref)
    Funcref f0 fref
    Function f0 fref1 = f1
    Function f0 fref2 = $"somefunc"
    Function f0 fref3 = fref
    Fref1()
    Fref2()
    Fref3()
End
```

上面的例子中,f0就是myprotofunc,它表示被引用的函数必须没有参数且返回一个数值。对函数引用赋值有3种方式:将一个函数名赋值给函数引用;利用“$”运算符解析一个字符串变量,将其内容作为函数名赋值给函数引用;将另外一个函数引用赋值给函数引用。

fref1引用了f1,因此fref1()就相当于调用f1()。fref2引用了一个名为somefunc的函数,这个函数是不存在的,因此系统会自动调用f0。fref3就是test函数参数传过来的函数

引用，fref3()将执行该参数所引用的函数。在这里可看到函数引用也可作为函数的参数。

在命令行里执行：

```
Test(f1)
```

输出结果如图 5-33 所示。

再执行

```
Test( $ "aa")
```

输出结果如图 5-34 所示。

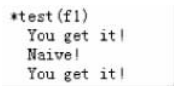


图 5-33 Test(f1)输出结果

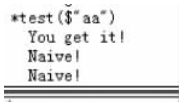


图 5-34 Test($ "aa")输出结果

请读者对照上面对函数引用的介绍分析输出结果。

在上面的程序文件里再增加一个函数 f3：

```
Function f3(v)
    Variable v
    Print "I am not the right one"
End
```

然后在函数 test 里添加：

```
Function test(fref)
    Funcref f0 fref
    Funcref f0 fref1 = f1
    Funcref f0 fref2 = $ "somefunc"
    Funcref f0 fref3 = fref
    Funcref f0 fref4 = f3              //f3 参数类型与模板函数不匹配，因此不能指定给 f0 类型
                                         的函数引用
    Fref1()
    Fref2()
    Fref3()
End
```

当编译时，会出现如下错误，如图 5-35 所示。

由于 f3 有一个参数，f0 没有参数，因此不能使用 f0 作为模板声明 f3 的函数引用。

注意，声明函数引用并赋值时必须采用下面的格式：

```
Funcref myprotofunc f = functionname
```

不能先声明后赋值，如下面的方法是错误的：

图 5-35　函数引用要求被引用的函数和模板函数必须匹配

```
Funcref myprotofunc f
f = functionname
```

函数引用在程序设计中是非常有用的。在编写数据拟合程序时，如果将拟合公式和过程硬编码在程序里，程序的通用性是很低的，以后扩展和维护起来也很不方便，如果使用函数引用，只有在运行时，才将真正的拟合函数传递给程序会大大提高程序的灵活性。这样，只需要按照指定格式编写拟合函数，原程序不需要任何修改即可利用新的公式拟合。拟合函数可以通过诸如下拉菜单等选择，利用 $ 符号将字符串转化为函数引用名。利用 ProcedureText 等函数，还可以自动获取拟合参数的信息（当然这需要拟合函数按照一定格式编写）。如此设计，程序会更加通用和简洁。在编写数据变换程序（如坐标变换）时，变换的公式往往随实验条件改变而改变，但是数据本身格式却相对固定，如源数据是两个坐标，目标数据亦是两个坐标，则可以在程序中使用函数引用，在具体的场合由用户选择或者按照指定格式编写相应的变换程序，同样可大大简化程序的结构，提高程序的通用性。

函数引用程序设计可以使用这样的技巧：如果模板函数的参数在编写程序时无法确定则可使用 wave 作为参数（当然也可以使用默认参数）。

5.3.7　访问全局对象

Igor 下能在函数中访问的全局对象包括全局数值型变量、全局字符串型变量、wave、函数、数据文件夹、符号路径、Graph 窗口、Panel 窗口、Notebook、程序窗口、坐标轴、控件等。除了全局变量和 wave 这些传统的数据对象之外，其他的对象都可以通过名字直接访问，也可以通过 $ 符号访问（这种方式更灵活，请读者参看本书第 5.3.9 节）。

对于 wave 和全局变量，Proc 和 Macro 程序类型中可以通过名字直接访问，Function 程序类型中不能通过名字直接访问。

在 Function 中访问 wave 和全局变量，如果这些全局对象是在函数体内定义的，则可以直接访问；如果是在函数体外定义的（如访问事先已有的数据），则需要先定义 wave 和变量的引用，然后才能访问。

先看一个例子。在命令行窗口中定义一个全局数值型变量并赋值为1，如图5-36所示。

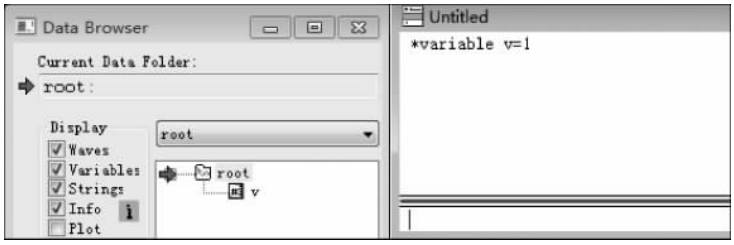


图5-36　在命令行窗口创建一个全局变量

在程序编辑窗口输入以下函数：

```
Function myFun()
    Print v
End
```

这里试图在函数里直接使用全局变量，但在编译程序时，Igor提示错误，如图5-37所示。

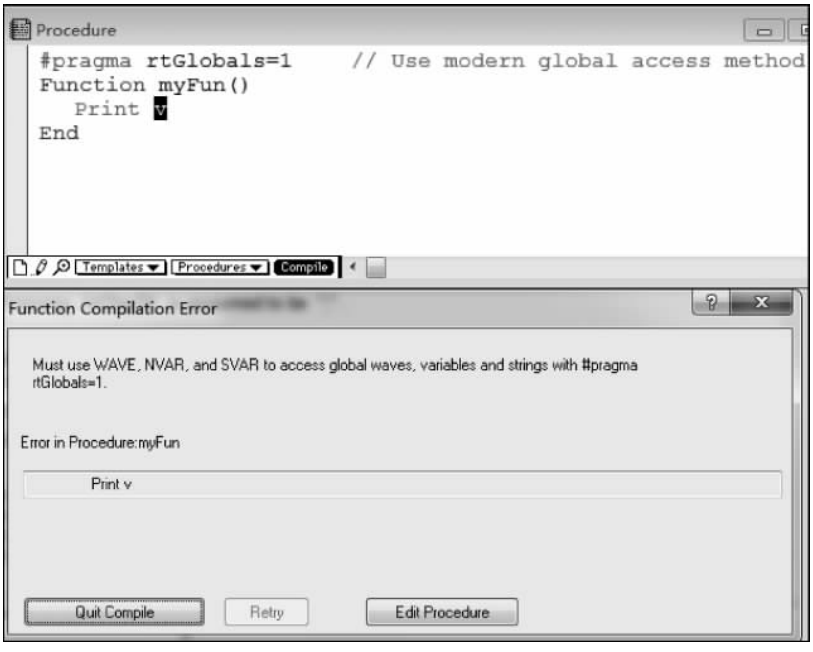


图5-37　不能在函数里利用名字直接访问全局变量

这个错误提示要使用wave、nvar和svar关键字去访问全局wave、变量或是字符串。因此对于本例，需要使用nvar创建对v的引用，如图5-38所示。此时程序就能正常编译运行，且能正常访问全局变量v。

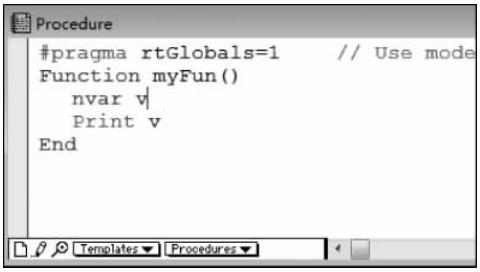


图 5-38 创建对全局变量的引用

Igor 使用下面 3 个关键字创建 wave 和全局变量的引用：

(1) wave 创建 wave 的引用。

(2) nvar 创建数值型变量的引用。

(3) svar 创建字符串型变量的引用。

这 3 个关键字用于创建一个引用，编译器会将全局变量和该引用关联起来，在运行时函数就可以使用该引用去访问全局变量。

注意，Igor 下使用引用访问全局变量(wave)，而不是通过名字去访问，不是为了使问题复杂化，恰恰相反，是为了使问题简单化。在程序设计时，并不知道有哪些数据，更不知道数据名字叫什么，因此根本无法通过名字访问数据。使用引用的技术可以先在程序里使用数据，而只有在运行时才将引用与真正的数据联接起来。

预编译参数 #Pragma rtGlobals 就是用来控制编译器的这种行为的。在最早的 Igor 版本中，创建全局变量(wave)的引用时，要求该变量(wave)必须存在，否则程序不能编译。#Pragma rtGlobals=1 或者 3 表示在创建引用时，Igor 并不检查与引用关联的变量是否存在，只有在运行时才将变量和引用真正联接起来(runtime lookup of globals)。

1. nvar 数值型变量引用

nvar 用于创建一个数值型变量引用，用于引用数值型全局变量，其语法格式为

```
NVAR [/C][/Z]localName[ = pathToVar][,localName1 [=pathToVar1 ]]...
```

其中，/C 和/Z 是可选参数，前者表示引用复数型全局变量，后者表示当被引用的全局变量不存在时不报错。*localName* 表示引用名字，在函数中将使用该名字来访问全局变量。*pathToVar* 是包含了路径的全局变量。nvar 创建非当前数据文件夹下的全局变量的引用时，需要指明全局变量的路径。

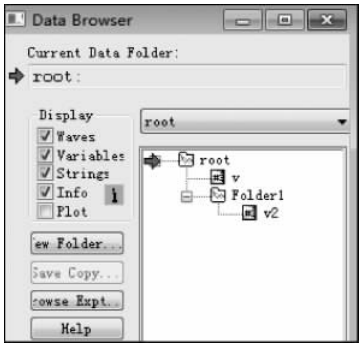


图 5-39 处于不同文件夹的 2 个全局变量

假定有以下 2 个全局变量，如图 5-39 所示，其中，v 位于 root 文件夹，v2 位于 root：Folder1 文件夹。当前文件夹是 root，则创建 v 和 v2 引用的方法如下：

```
nvar v                    //正确，此时引用名就是 v，能直接访问 v
```

```
nvar num = v                    //正确,引用名为 num,能直接访问 v
nvar num = root:v               //正确,引用名为 num,指明 v 的完整路径,能直接访问 v
nvar v2                         //错误,由于当前文件是 root,root 下不存在 v2 这样的全局变量,
                                  不能访问 v2
nvar num = root:Folder1:v2      //正确,引用名为 num,且指明了绝对路径,可以访问 v2
nvar num = :Folder1:v2          //正确,引用名为 num,且指明了相对路径,可以访问 v2
nvar v = v                      //正确,引用名和全局变量名相同,可以访问 v
```

请读者注意,访问当前文件夹下的全局变量时,可以在 nvar 后直接跟全局变量名,就可以通过变量名访问该全局变量了,如上面第一行例子。

创建复数类型全局变量的引用与创建普通数值型变量引用的方法完全相同,只需要使用/C 标记即可。

当编译选项 #Pragma rtGlobals=1 或者 3 时,编译器在编译时不会检查全局变量是否存在,只有在运行时才会检查全局变量是否存在并建立关联。

2. svar 字符串型变量引用

svar 用于创建一个字符串型引用,引用一个全局字符串变量,用法和 nvar 完全相同,本书不再赘述。

3. wave 引用

wave 关键字用于创建一个 wave 的引用,其语法格式为

```
WAVE [/C][/T][/Z] localName [ = pathToWave][,localName1 [ = pathToWave1 ]]...
```

(1) /C,创建一个复数型 wave 引用,如果被引用 wave 是复数类型时需要使用此选项。

(2) /T,创建一个文本型 wave 引用,如果被引用 wave 是文本类型时需要使用此选项。

(3) /Z,当被引用 wave 不存在时不报错(但使用时会报错)。

在函数中各种能产生 wave 的命令,除非使用/FREE 选项,都会自动在当前或者指定的文件夹下创建一个真实的 wave。这不同于普通编程语言(如 C 语言),数组在函数结束时即被释放。函数中创建的 wave 和命令行窗口中创建的 wave 一样,将永久存在。因此 wave 几乎总是全局型的,在访问 wave 时,必须用 wave 关键字创建一个引用。请看下面的例子,用到的 wave 如图 5-40 所示。

在函数中访问 root 下面的 wave,必须按照如下方式声明引用:

```
wave w = wave1              //引用名为 w
wave wave1                  //引用名为 wave1
wave w = root:wave1         //引用名为 w,指明绝对路径
wave w = :wave1             //引用名为 w,指明相对路径
```

和 nvar 一样,如果 wave 位于当前文件夹,可以直接在 wave 关键字后跟 wave 的名字创建该 wave 的引用,此时可以通过 wave 名字访问 wave。

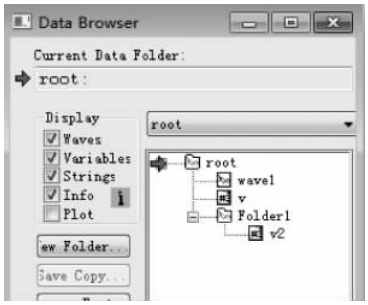


图 5-40　wave 访问的示例

需要注意的是，一些命令在执行时会自动创建 wave 的引用，这时就不需要专门创建引用了。如 Make wave1 指令在创建 wave1 的同时自动创建了一个 wave1 的引用，该引用名就是 wave1 本身，其他的命令如 Duplicate 等也是如此。

使用 Proc 和 Macro 关键字声明的代码段，可以直接访问全局变量而无须使用 nvar、svar 和 wave 声明引用。考虑下面的例子，假设当前文件夹为 root，并存在一个名为 wave1 的 wave，一个字符串变量 str 和一个数值型变量 v，则下面的程序可以直接访问这些全局变量而无须创建引用（执行之前编译器仍然不会去检查变量是否存在）。

```
Make/O wave1
Variable v
string str
proc myProc()
    wave1 = sin(x)
    v = 1
    str = "Hello,Igor"
End
Macro myMacro()
    wave1 = sin(x)
    v = 1
    str = "Hello,Igor"
End
```

全局变量会增加函数之间的相互关联度，降低程序的通用性，应该在使用过程中加以避免。但是在有些场合全局变量也是必要的，如记忆程序的执行状态，记录实验数据信息，存放 ListBox 内容等都需要使用全局变量（或 wave）。

5.3.8 wave 引用

wave 是 Igor 下最基本的概念，在 Igor 下编程必须熟练地掌握获取 wave 对象的各种技巧。第 5.3.7 介绍了如何通过 wave 关键字创建一个 wave 的引用，这是在函数中使用 wave 的一般方法。本节介绍一些特殊情况。

1. 命令自动生成 wave 引用

当执行 Make、Duplicate 或者是 FFT 等命令时，Igor 编译器会自动创建 wave 引用，如

```
Make wave1
Duplicate wave1 wave2
FFT/dest = w wave1
```

第 1 个命令在当前文件夹下创建 wave1，同时创建 wave1 的引用，这个引用名就是 wave1，可以直接使用 wave1 访问刚创建的 wave 或者是做任意运算，如 wave1＝x＊x。

第 2 个命令中 wave1 的创建与引用与第 1 个命令中含义完全相同。

第 3 个命令对 wave1 执行傅里叶变换，并创建一个新的名为 w 的 wave，将变换后的结果保存在 w 中，同时使用 w 作为新创建 wave 的引用名。

Igor 下绝大多数能创建输出 wave 的命令都具有这个特性，新生成的 wave 可以直接使用。

上面的情况也有例外，考虑下面的例子：

```
Function CreateaWave(namestrforwave)
    String namestrforwave
    Make $namestrforwave
    wave w = $namestrforwave
    w = x^2
End
```

本例中，使用 Make 命令创建一个 wave，该 wave 的名字保存在一个字符串变量 namestrforwave 中，使用 $ 运算符从该字符串解析出名字并作为 Make 参数创建该 wave。此时并没有同时创建该 wave 的引用，因此需要利用 wave 关键字创建其引用。

对于上面的例子还有一种更方便的方法：

```
Make $namestrforwave/wave = w
```

在 Make 命令中使用/wave 选项，就可以直接利用引用名 w 访问刚创建的 wave 了。

很多命令支持这种声明 wave 的方法，如下面的命令计算微分后，可以直接用 wave1 访问微分以后的结果：

```
Differentiate wave0 /D = $name/wave = wave1
```

上面的命令生成的结果 wave 名保存在 name 字符串变量中，在这里使用了 $ 运算符创建目标 wave。

在创建 wave 引用时需要指明被引用 wave 的数值类型，如果不指明则默认为被引用的 wave 是一个双精度数值型 wave。

2. 返回 wave 引用的函数

为了方便 wave 的访问，Igor 提供了能直接返回 wave 引用的函数，这些函数功能都特别强大，用好这些函数访问数据会变得很简单。表 5-6 列出了部分返回 wave 引用的函数。

表 5-6 返回 wave 引用的函数

函 数	功 能
CsrWaveRef	返回 Graph 中当前 Cursor 所处的 Y wave 引用
CsrXWaveRef	返回 Graph 中当前 Cursor 所处的 X wave 引用(XY 型 wave)
WaveRefIndexed	返回 Graph 或是 Table 或是当前文件夹下的 wave 引用
XWaveRefFromTrace	返回 Graph 中的 X wave 引用。需要指明对应曲线(trace)的名字，该名字通常可以通过 TraceNameList 函数获取
ContourNameToWaveRef	返回 Graph 中被显示为 Contour 模式的 wave 引用。需要提供 Contour 对应 wave 的名字，该名字可以通过 ContourNameList 函数获取

续表

函　　数	功　　能
ImageNameToWaveRef	返回 Graph 中被显示为 Image 模式的 wave 引用。需要提供 Image 对应 wave 的名字，该名字可以通过 ImageNameList 函数获取。
TraceNameToWaveRef	返回 Graph 中被显示为曲线模式的 wave 引用。需要提供 Trace 对应 wave 的名字，该名字可以通过 TraceNameList 获取
TagWaveRef	返回 Graph 中当前 Tag 所处的 Y wave 引用
WaveRefsEqual	比较两个 wave 的引用是否相同，是则返回 1，否则返回 0

这些函数会返回一个 wave 的引用，可以赋值给由 wave 关键字创建的 wave 引用变量。这些函数绝大多数都以 Graph 作为操作对象，要被获取引用的 wave 显示在 Graph 窗口中。这样只需要通过 Graph 就能获取 Graph 中数据对象的引用，而无须知道名字和存放的路径，这会大大降低编程的复杂性并提高数据处理的效率。

常用的返回 wave 引用的函数是 CsrWaveRef、WaveRefIndexed、TraceNameToWaveRef 和 ImageNameToWaveRef。

(1) CsrWaveRef 函数的使用方法为

```
CsrWaveRef(cursorName[,graphNameStr])
```

courseName 表示光标(按 Ctrl+I 键设置)，可以是 A 或者 B。*graphNameStr* 是图窗口的名字，为可选参数，如果提供则从 *graphNameStr* 指定的图窗口中获取引用，否则默认从当前最顶层图窗口获取引用，如

```
wave w = Csrwaveref(A)
wave w = CsrWaveRef(A,"graph0")
wave w = csrwaveref(A,namestr)
Print Csrwaveref(A)[0]
```

第 1 行返回当前最顶层窗口 Cursor A 所在的 wave 引用，并赋值给 w。

第 2 行返回一个名为 graph0 的窗口中 Cursor A 所在的 wave 引用，并赋值给 w。

第 3 行返回一个名字存放在 namestr 中的 Graph 中 Cursor A 所在的 wave 引用，并赋值给 w。

第 4 行直接打印当前最顶层窗口 Cursor A 所在的 wave 的第一个数据。

在对曲线进行拟合时，当窗口中曲线较多，且需要使用 Cursor 指定拟合范围时，利用 CsrWaveRef 函数可以获取要被拟合的曲线的引用，十分方便。

(2) WaveRefIndexed 函数的调用方法为

```
WaveRefIndexed(windowNameStr, index, type)
```

windowNameStr 是一个字符串变量，用于存放 Graph 或者 Table 的名字，如果是空字符串则表示当前顶层 Graph 或者顶层 Table 或者当前文件夹。

index 表示要返回 wave 所处的次序。

type 可取 1～4，如果取值为 4，*windowNamestr* 应为空字符串，表示获取当前数据文件夹下的 wave 引用。当前窗口为 Graph 时，1 表示获取 Y wave 引用，2 表示获取 X wave 引用，3 表示两者都获取，这也是最常见的情况，如

```
wave w = WaveRefIndexed("",0,1)
```

上述命令返回当前 Graph 窗口中的第一条曲线的引用。一般一个窗口只显示一条曲线，当需要对该曲线进行拟合、运算等操作时，这条命令提供一个非常方便地获取该曲线引用的方法：既不需要知道该曲线的名字，也不需要知道该曲线的存放路径。

（3）TraceNametoWaveRef 函数的调用方法为

```
TraceNameToWaveRef(graphNameStr, traceNameStr)
```

graphNameStr 表示 Graph 窗口的名字，如果为空字符串表示当前顶层窗口，*traceNameStr* 表示曲线 wave 的名字。可以利用 TraceNameList 获取 Graph 中所有曲线的名字列表，并将它作为 TraceNameToWaveRef 的参数，如

```
string list = TraceNameList("",";",1)
wave w = TraceNameToWaveRef("",StringFromList(0,list))
```

第一行通过 TraceNameList 函数获取当前顶层 Graph 中所有曲线的名字列表，以分号作为分隔符存放在字符串变量 list 中，第二行利用 StringFromList 取出 list 中的第一个字符串，然后将它作为 TraceNametoWaveRef 的参数，就可以获取该字符串对应曲线的 wave 引用。

ImageNameToWaveRef 函数和 TraceNameToWaveRef 函数的使用方法完全一样，表示获取 Graph 中一个 Image 的引用，唯一不同的是其名字列表需要用 ImageNameList 获取。

在 Igor 下，要访问 wave，仅仅使用名字是不够的，还需要同时知道 wave 所处的数据文件夹，因此一般的方法是要么设置当前文件夹为 wave 所处文件夹，要么使用完整路径访问 wave。利用上面介绍的返回 wave 引用的函数的优点是不需要知道目标 wave 的名字及其存放路径。

在实际中经常还有这样的需求：需要对 Graph 中显示的曲线或者 Image 进行操作。普通方法需要预先知道 Graph 或者 Image 中数据存放的位置，然后将这些信息硬编码到程序中。显然这样的程序是没有通用性的。利用上面的函数，就可以解决这个问题，只需要通过 Graph 就能直接得到目标曲线的引用。其实这也是很好理解的，Graph 既然能显示一个 wave，那么它必然包含这个 wave 的全部信息。

细心的读者可能会问，有了 wave 的引用，能否获取 wave 的名字和它所在的路径呢？答案是可以的，只需要使用下面两个函数就能达到这个目的：

```
NameOfWave(wave)
GetWavesDataFolder(wave,kind)
```

第一个函数使用非常简单，直接返回 wave 的名字，注意其参数是一个 wave 引用，如下

列命令打印 wave w 的名字：

```
Print NameOfWave(w)
```

第二个函数返回一个 wave 所处的文件夹，第 1 个参数是一个 wave 引用，第 2 个参数含义如表 5-7 所示。

表 5-7　GetWavesDataFolder 函数的 kind 参数含义

kind	含　义
0	返回包含 wave 的文件夹名
1	返回包含 wave 的完整路径，但不包括 wave 名
2	返回包含 wave 的完整路径，包括 wave 名

看下面的例子，结果如图 5-41 所示。

```
Function myFunc()
    Make/O data1
    Setscale/I x,0,2 * pi,data1
    Display data1
    wave w = waverefindexed("",0,1)
    Print nameofwave(w)
    Print getwavesdatafolder(w,2)
End
```

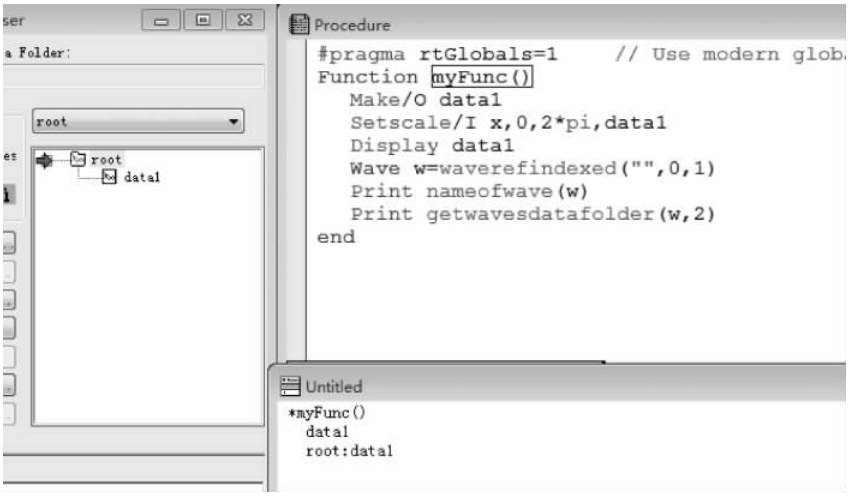


图 5-41　通过引用获取 wave 的名字和存放路径

一个被引用的 wave 不存在时，访问 wave 就会出错。当使用 #Pragma rtGlobals＝1 时，编译器在编译一个类似于 wave w＝wave1 的表达式时并不会检查 wave1 是否真正存在，只有在运行时才会将 w 和 wave1 真正联结起来。因此在使用 wave 引用之前应该检查

wave 是否存在，方法为

```
if(!waveexists(w))                          //wave w 存在
    <do somethings>
else                                        //wave w 不存在
    <coping with errors>
endif
```

5.3.9 $ 运算符

引用是 Igor 中广泛应用的一个概念，前面看到可以利用 nvar、svar 和 wave 声明变量或者 wave 的引用。实际上除了变量和 wave 之外，还可以声明或者创建窗口（Window）、路径符号（Symbolic Path，这里指操作系统文件路径，非 Igor 下文件路径）和函数的引用。

利用前面介绍的方法声明引用时，变量已经存在，如

```
nvar num = v
```

v 是一个已经存在的变量，在编写代码时就知道。但在实际中，要访问的对象经常是变化的，或者说无法预知访问的对象是谁，比如需要将一批 wave 显示出来，而在编写程序时，并不知道这些 wave 的名字，只有在运行时才能获取它们的名字，这种情况下就无法按照上面的办法来创建引用。

解决这个困难的方法是使用 $ 运算符。

利用 $ 运算符可以将一个字符串或者字符串变量转化为一个引用，引用该字符串所代表的对象。使用 $ 运算符无须担心引用的类型，Igor 会自动根据上下文环境判断被引用的对象类型，这个对象可以是变量、wave、窗口、文件夹、路径符号或者函数。首先来看利用 $ 运算符创建 wave 引用的例子。

1. $ 创建 wave 引用

```
wave w = $"wave1"
```

假定当前文件夹下存在一个名为 wave1 的 wave，则上面的命令就可以创建 wave1 的引用。使用 $ 运算符作用于“wave1”字符串，并将它赋值给引用变量 w，这样就创建了 wave1 的引用。可以简单地理解为 $ 运算符会解析字符串里的内容，并按照字符串内容（这里就是 wave 的名字）创建该 wave 的引用。

在这里也可以使用 wave w＝wave1 来声明 wave1 的引用，请读者体会两者的区别。

下面来看较为复杂一些的例子：

```
Function displaywaves()
    String s,s1
    s = WaveList("*",";","DIMS:1")
    Variable i,n
    N = ItemsInList(s)
    Display
```

```
for(i = 0;i < n;i += 1)
    s1 = StringFromList(i,s)
    AppendToGraph $ s1
endfor
End
```

这个函数用来显示当前文件夹下所有的一维 wave。首先使用 WaveList 获取所有一维 wave 的名字，以分号分隔并存放于变量 s 中，然后利用 for 循环取出每一个 wave 的名字存放在 s1 中，接着利用 $ 运算符将 s1 转化为其内容对应 wave 的引用。

显然，本例中事先无法得知要显示的 wave 有哪些，不能直接创建这些 wave 的引用，而只能在程序运行时通过 $ 运算符动态生成。

注意下面使用 $ 的方法是错误的：

```
String s = "somewave"
$ s = sin(x)
```

$ 运算符并不会直接创建引用，这是因为仅仅使用 $ 运算符，编译器无法确定 $ 运算符后面的字符串里内容所代表的对象类型。前面，通过 wave 关键字告诉编译器，$ 运算符解析出来的字符串代表一个 wave。对于第二个例子，在 AppendToGraph $ s1 命令中，编译器能够通过 AppendToGraph 确定这里的 $ s1 代表一个 wave 的引用。类似的情况还有 Display、CurveFit 等需要使用一个 wave 作为参数的情况。

对于上面的错误只要进行如下修改就可以正常使用：

```
String s = "somewave"
wave w = $ s
w = sin(x)
```

2. $ 创建变量引用

通过 $ 创建变量引用和创建 wave 引用的例子完全类似，只需要将 wave 关键字替换为 nvar 和 svar 即可，但注意被创建引用的变量必须是全局变量。看下面的例子：

```
Variable var1 = 1
String str1 = "Global string"
Function myFun()
    String vref,sref
    vref = "var1"
    sref = "str1"
    nvar v = $ vref
    svar sG = $ sref
    Print v,sg
End
```

执行上面的函数，结果如图 5-42 所示。

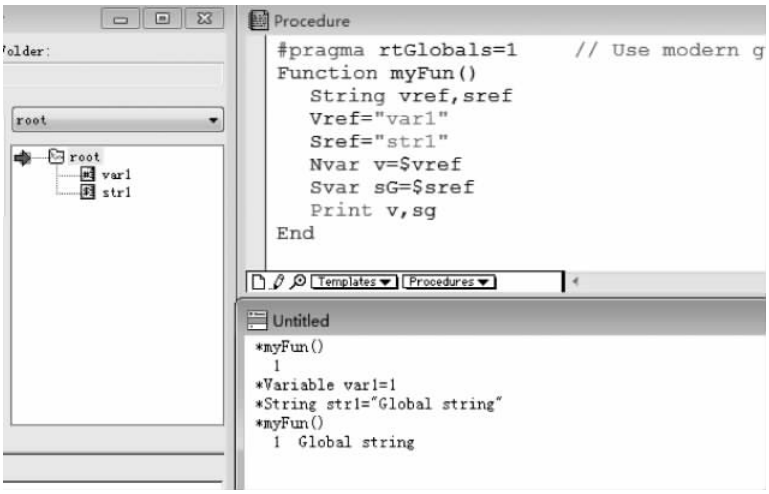


图 5-42　利用 $ 运算符创建全局变量的引用

3. $ 运算符创建窗口引用

可以利用 $ 运算符创建窗口的引用。从前面的介绍知道可以通过名字直接访问窗口，但有时窗口的名字并不知道，而是动态生成的，并且保存在一个字符串里，此时就可以通过 $ 运算符解析该字符串获取对窗口的引用。例如下面的函数判断指定名字的窗口是否存在，就使用了窗口引用的技巧。

```
Function GraphWindowExsts(s)
    String s
    Dowindow $s
    if(V_Flag == 0)
        Return 0
    else
        Return 1
    endif
End
```

在这个例子中，将 Window 的名字存放在字符串变量 s 中，然后利用 $ 运算符解析出 Window 名字，由于 $s 作为 Dowindow 的参数，编译器能够确认 $s 代表一个 Window 引用，并根据 s 里存放的名字获取该 Window。其他的命令如创建、修改或是需要指定 Window 名字的命令，都可以通过 $ 运算符来指定 Window 的名字。如

```
Display/N= $graphNameStr $ywavename vs $xwavename
Cursor/W= $graphNameStr A, $ynamestr,0
```

第一个例子，选项/N 指定了新创建的 Graph 的名字为 graphNameStr 字符串内容。第二个例子/W 选项指明了 Cursor 命令执行时的目标 Graph 窗口，窗口名称包含在 graphNameStr 内，如果不用 $ 运算符，则表示窗口名称为 graphNameStr 本身。请读者注

意体会。

4. $运算符的对其他全局对象的引用

在程序设计中，其他的全局对象有记事本(Notebook)、控件(Controls)、坐标轴(Axis)，函数(Function)、程序窗口(Procedure Window)、程序面板(Panel)、符号路径(Symbolic Path)等。Igor下有很多的命令和函数用于操作这些全局对象，一般通过名字来访问这些对象。和上面的窗口引用完全一样，如果这些名字事先无法确定，则可以利用一个字符串变量保存动态生成的对象名字，然后在对应的命令后利用$运算符＋字符串变量的方式来通过引用方式访问对象。

如一个设置图中坐标轴的程序，由于Graph的坐标轴不仅仅是left和bottom，还可能包括各种自定义坐标轴，所以程序里无法事先知道有哪些坐标轴，只能在程序运行时利用Igor提供的函数获取Graph里所有的坐标轴名字，然后通过$运算符转化为对坐标轴的引用：

```
String s1 =  AxisList("")                    //获取当前窗口所有坐标轴名字列表,保存在 s1 中
String s2 = StringFromList(0,s1)             //获取第一个坐标轴名字,保存在 s2 中
ModifyGraph log( $ s2) = 1                   //将 s2 对应的坐标轴设置为对数刻度
```

通过$引用函数的方法请参看第5.3.6节。

通过$引用符号路径请参看第7.8节。

5.3.10 自动创建变量

Igor中有很多命令在执行时会自动创建一些变量。如果是在命令行窗口或是Macro(Proc)中执行，则在当前文件夹下创建全局变量；如果是在函数中执行，则创建局域变量。在函数中使用自动变量时不需要声明，可以直接访问。

Dowindow命令用于检查一个窗口是否存在，存在则将一个变量V_flag设置为1，否则将V_flag设置为0。在命令行窗口执行下列命令：

```
Display
Dowindow/C myGraph
Dowindow myGraph
Print V_flag
```

注意观察在执行命令之前当前文件夹中并没有V_flag这样的命令，如图5-43所示。

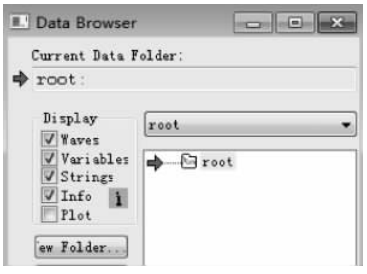


图5-43 执行命令前当前root文件夹中没有V_flag变量

执行完命令后可以看到Dowindow自动创建了一个变量V_flag，并且将其值设置为1。这里Display命令产生一个Graph窗口，Dowindow/C命令将其名字设置为myGraph，由于myGraph窗口处于显示状态，故Dowindow作用于myGraph窗口时会自动创建V_flag变量并设置为1。在程序中经常使用这个方法来

判断一个窗口是否存在。

执行下面的命令：

```
Dowindow somegraph
Print V_flag
```

输出结果为0。这是由于不存在名为somegraph这样的窗口，故V_flag被设置为0(注意，如果V_flag没有会自动创建)。

函数中使用Dowindow的例子如下：

```
Function GraphWindowExsts(s)
    String s
    Dowindow $s
    if(V_Flag == 0)
        Return 0
    else
        Return 1
    endif
End
```

上面的函数用于测试一个窗口是否存在。窗口名字存放于字符串变量中作为参数传递给函数，通过$运算符获取窗口名字并由Dowindow判断是否存在。程序随后检查V_flag变量，如果V_flag等于0，则返回0，表示窗口不存在；否则返回1，表示窗口存在。

这里可以看到在使用V_flag之前并没有创建它，也没有在其他地方声明，也不是通过参数由调用环境传递而来，那么V_flag到底是从哪儿来的？答案是由编译器自动创建。编译器在编译上述代码时，当检测到存在Dowindow这样的命令时，就会自动创建V_Flag这样的变量，换句话说，是编译器创建了这个变量。

其他的命令如ControlInfo、WaveStats、FindRoots等都会创建自动变量，这些变量一般都统一以“V_”这样的前缀开头，命令执行后可以直接使用。Igor对每个命令能创建的自动变量都有详细的说明，读者只需要在在线帮助系统中查询该命令就可以获知这些信息。

除了创建自动变量，一些命令还会创建wave，如CurveFit命令会自动创建名为w_sigma的wave，如果是在函数中，其引用名亦为w_sigma。这个wave保存了每个拟合参数的标准偏差，可以无须声明就可直接使用。

注意，无论是自动创建的变量还是wave，对其访问必须在对应的命令之后。如下面对自动创建变量的访问是错误的：

```
Function test()
    Print V_flag
    Dowindow aa
End
```

5.3.11 调试程序

编写程序重要，调试程序更重要。Igor 下有两种调试程序的方法：

(1) 使用 Print 命令。

(2) 使用自带的调试器。

Print 类似于 C 语言中的 printf 语句，通过向历史命令行窗口输出信息来调试程序中可能的错误。调试器则可以设置断点、实时观测变量值，甚至计算表达式的值，功能更加复杂、完善。

使用 Print 方法进行调试非常简单，在最可能出错的地方输出一些关键变量的值，通过审查这些变量值是否符合预期从而确定程序出错的原因。这里不再对 Print 调试方法做过多介绍，而是重点介绍调试器的使用方法。

当需要调试程序时，首先需要使程序处于可调试状态，方法是右击程序编辑窗口，在弹出的快捷菜单中选择【Enable Debugger】或者执行菜单命令【Procedure】|【Enable Debugger】(注意只有程序编辑窗口处于选中状态【Procedure】菜单才会出现)。这类似于一些集成开发环境中以 Debug 模式运行程序。打开程序调试功能的菜单如图 5-44 所示。

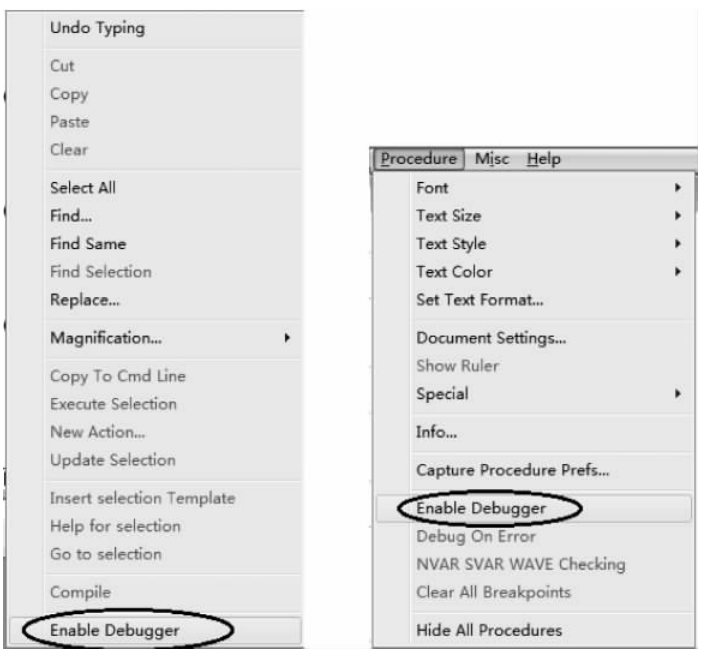


图 5-44 打开程序调试功能

Igor 程序调试器支持单步执行，可以在任意一行代码插入断点。程序运行到断点会自动停止，同时弹出调试对话框，在对话框中可实时查看变量、wave 的值，也可以计算任意合法表达式的值。

当程序处于可调试状态时，在程序编辑窗口的左侧会出现一个空白的竖条，单击竖条会在单击位置出现一个红点，表示在红点对应的行设置了一个断点，再次单击红点可清除该断点，如图 5-45 所示。

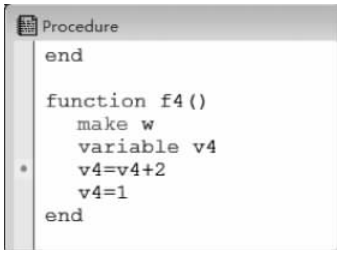


图 5-45　设置断点

当程序运行到断点位置时停止执行，并弹出调试对话框，如图 5-46 所示。

设置断点有点类似于 Print 方法，只是 Print 方法输出关键变量的值后继续执行，而设置断点之后，程序会在断点处停下来，此时就可以通过调试器观察当前函数所有变量的值，然后选择继续执行或者是终止执行，是单步执行还是正常执行等。

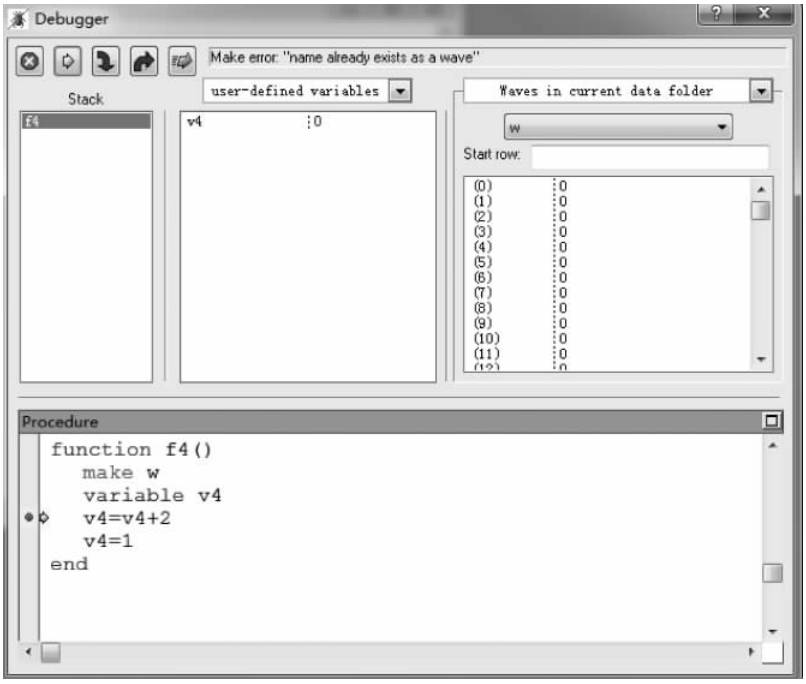


图 5-46　调试对话框

设置断点是程序调试最基本的技能，但有时仅靠设置断点是不够的。设置断点要求对程序最可能出错的地方有所估计，否则断点的设置具有一定的盲目性，有时难以找到程序的故障点或者出错的真正原因。例如，一些脆弱的程序经常在执行完后才显示某个命令出错，而这个命令什么时候出错，出错时其运行环境如何都不知道，这就很难准确地设置断点。当这个出错的命令位于一个循环之中或者是位于一个被多次调用的函数之中时，直接设置断点几乎不可能，因为无法确定在第几次循环或者在第几次被调用时出现错误。还有一种情况是如果函数中引用的对象不存在，如一个 wave 引用变量所引用的真实 wave 不存在，会跳出一个 wave 引用的错误，但是它并不提示到底是哪个 wave 引用错误，如果程

序中引用的 wave 很多，就很难确定到底是哪个 wave 引用出错，从而也无法准确设置断点。

针对这种情况 Igor 提供了另外两种会自动调出调试器的方法：运行时错误调试；nvar、svar 和 wave 引用错误调试。前者当程序运行出错时立即调出调试器，后者当引用的全局对象不存在时立即调出调试器。这两种方法结合断点几乎可以立即确定程序出错的位置，顺利完成程序调试。

启动运行时错误调试的方法是右击程序编辑窗口，在弹出的快捷菜单中选择【Debug on Error】，也可以执行菜单命令【Procedure】|【Debug on Error】。启动 nvar、svar 和 wave 引用错误调试的方法是右击程序编辑窗口，选择【NVAR SVAR WAVE Checking】，或者执行菜单命令【Procedure】|【NVAR SVAR WAVE Checking】，如图 5-47 所示。

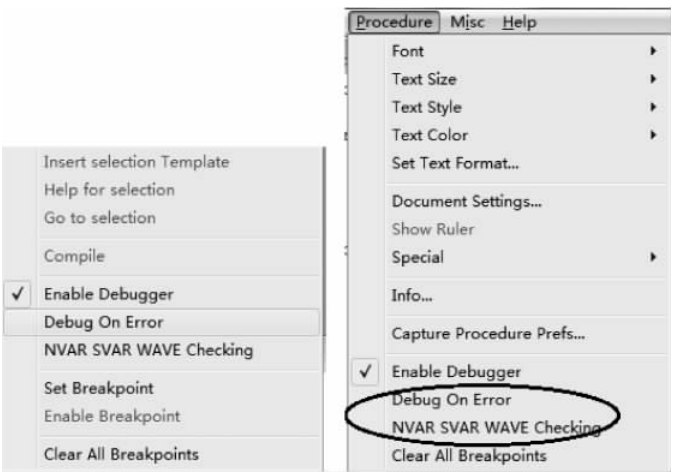


图 5-47　启动运行时错误调试和引用错误调试

看下面的例子：

```
Function testf(v1)
    Variable v1
    Make/O/N=(3,4) data
    Variable i
    for(i=0;i<v1;i+=1)              //v1 大于 4 会出错
        MatrixOP/O w=col(data,i)
    endfor
    wave w=w1                       //w1 可能不存在,此时引用会出错
    Display w1
    Print v1
End
```

上面的例子中存在一个明显的错误，一个潜在的错误。明显的错误是 w 引用了一个不存在的 wave w1，这时 Display 命令就会出错；潜在的错误是循环中的 MatrixOP 命令，当参数 v1 大于二维 wave 的最大列数（本例中最大列数为 4）时程序就会出错。首先不启用

【Debug On Error】和【NVAR SVAR Wave Checking】菜单命令，执行 testf(1)，这时程序出现如图 5-48 所示的错误提示。

图 5-48　执行 testf(1)后报错

同时历史命令行窗口输出 1，表示程序仍然正常执行完毕，但是里面有一些错误。

如果执行 test(5)则会出现如图 5-49 所示的错误提示。

图 5-49　执行 test(5)后报错

虽然上面两个错误提示了可能出错的原因，但是并没有给出到底是哪个 wave 引用出现了错误(实际的例子远比这个复杂，wave 引用可能是几十个)，特别是第二个错误，只有在 v1 大于 4 时才出错，而什么时候 v1 大于 4，是什么原因导致 v1 大于 4，单靠这个提示是无法确定的。此时，如果启用了【Debug On Error】和【NVAR SVAR Wave Checking】选项，则程序会在出错的地方立即停下来，这样就可以看到到底是哪个 wave 引用出错或者是到底 v1 等于多少导致 MatrixOP 出错。

启用 Debug On Error 和 NVAR SVAR WAVE Checking，执行 testf(1)，程序会弹出调试对话框如图 5-50 所示。

再执行 testf(5)，弹出调试对话框，如图 5-51 所示。

当被调试的程序类型是 Proc 或者 Macro 时，如果存在错误，则无论是否设置 Debug On Error，都会弹出一个类似于图 5-52 所示形式的对话框，单击【Debug】按钮，也可以调出调试器。当然如果设置了断点，则当程序执行到断点时也会自动调出调试器，这和 Function 没有区别。图 5-52 所示是 Macro 程序出现错误时的提示信息对话框。

下面介绍调试器(调试对话框)的含义和使用方法。调试器由程序运行控制按钮、出错信息、当前函数列表(Stack)、变量列表、wave 和表达式及结构体变量、程序窗口区域这 6 个区域组成，除了程序运行控制按钮和出错信息区域外，每个区域的大小都可以调节。调试对话框如图 5-53 所示。

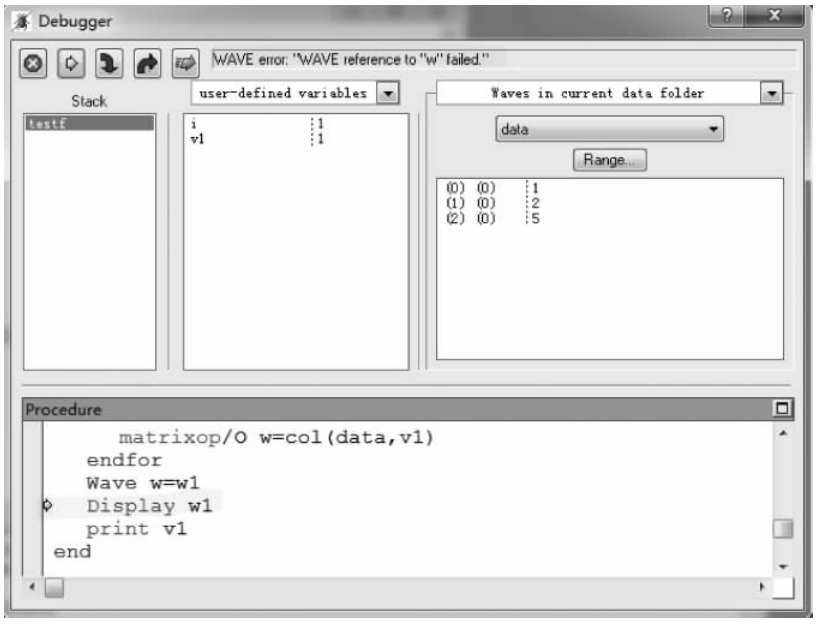


图 5-50　wave 引用错误

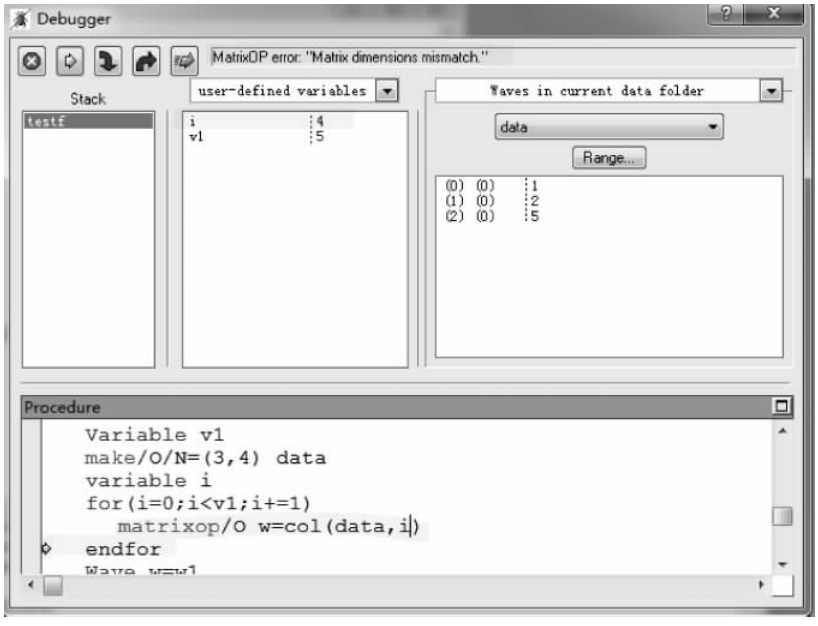


图 5-51　程序在出错的地方立即停止执行并弹出调试对话框

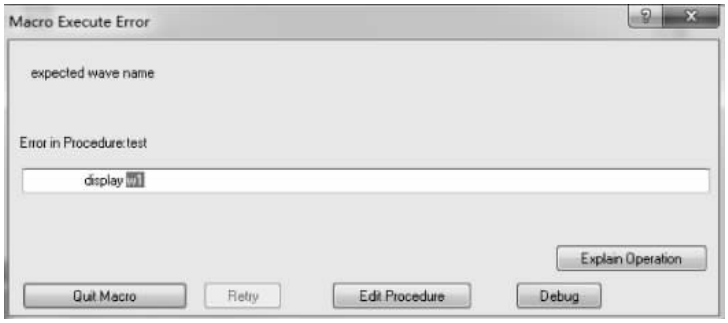


图 5-52　Macro 出错时的提示信息对话框

图 5-53　Igor 程序调试器

程序控制按钮含义如下(Igor Pro7 按钮图标变了,但含义一样):

：立即终止程序执行。

：单步执行。

：进入调用函数。当表达式中含有函数时,按此按钮会进入被调用函数。

：跳出被调用函数,返回到原函数。

：正常执行程序。

函数列表列出当前正在执行的函数以及调用该函数的函数列表，例如图 5-53 所示的调试器就是对图 5-54 的例子进行调试时的结果。

在函数 f4 中设置断点，然后执行 f1()，由于 f1 调用了 f2，f2 调用了 f3，f3 调用了 f4，这 4 个函数都会显示在函数列表中。

变量列表可以显示当前函数中所有变量的值，变量列表有 4 个选项和 1 个控制选项，如图 5-55 所示。

```
Procedure
#pragma rtGlobals=1
function f1()
   variable v1
   print 1
   f2()
end

function f2()
   variable v2
   f3()
end

function f3()
   variable v3
   f4()
end

function f4()
   make w
   variable v4
   v4=v4+2
   v4=1
end
```

图 5-54　程序代码调试示例

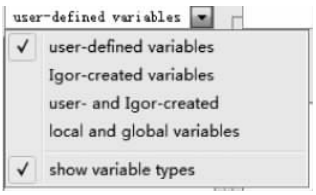


图 5-55　调试器变量列表

- user-defined variables 表示只显示局部变量。
- Igor-created variables 表示只显示由 Igor 创建的变量，如 V_flag 等。
- user-and Igor-created 表示显示局部变量和 Igor 创建的变量。
- local and global variables 表示显示局部变量和和当前函数引用的全局变量。
- show variable types 表示显示变量的类型。

变量列表显示方式如图 5-56 所示。

其中第一列为变量名，第二列为变量的值，如果选择了【Show variable types】选项，则还会显示变量的类型，如图 5-57 所示。

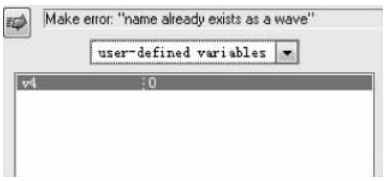


图 5-56　调试器变量列表

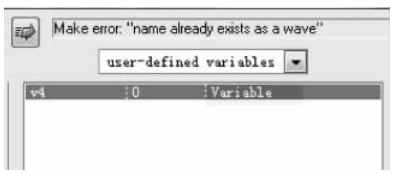


图 5-57　显示变量类型

每一列的宽度都可以调节。双击每一个变量值所在区域，可以修改变量的值。

当左边函数列表中被选择的函数发生变化时，变量列表也相应调整为对应函数中的变量。

wave 和表达式及结构体区域列出当前函数列表中所选择函数中所有的 wave、结构体变量，还可以输入表达式并计算输出结果。和变量列表类似，这里也有不同的选项，如图 5-58 所示。

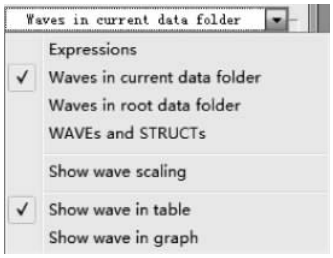


图 5-58　在调试器中查看 wave、结构体和表达式

Expressions 表示输入表达式并计算表达式的值，如当前函数中有一个 v4 的局部变量，输入下面的表达式，然后按回车键，调试器自动计算出结果，如图 5-59 所示。

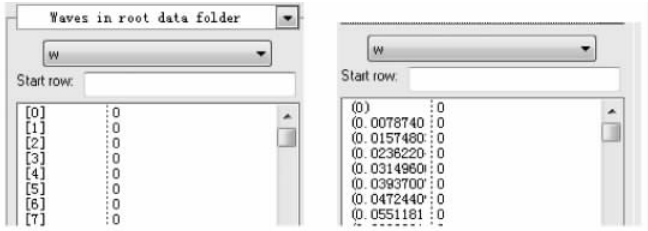
图 5-59　调试器中使用表达式

Waves in current data folder，查看当前文件夹下的所有 wave。

Waves in root data folder，查看 root 文件夹下的所有 wave。

WAVEs and STRUCTs，查看当前函数中所有引用的 wave 和结构体变量。

Show wave scaling，显示 wave 的 $x(y)$ 坐标。如果不选择此选项，wave 显示的方式为【index】：value，选择此选项，则显示方式为(坐标)：value，如图 5-60 所示。



(a) wave的显示方式为[index]: value　(b) wave的显示方式为(坐标): value

图 5-60　调试器可以查看 wave 内容

Show wave in table，设置 wave 的显示方式为表格方式，如图 5-60 所示。

Show wave in graph，设置 wave 的显示方式为图形式，如图 5-61 所示。

调试器程序窗口显示当前函数所处程序文件中的所有代码，由一个黄色的小箭头指向当前断点处，另外还有一个灰色的小箭头指向调用设置了断点的函数的表达式。程序窗口区域也可以设置断点，但无法编辑程序。通过程序窗口还可以实时地查看变量的值，方法是将鼠标指针置于一个变量之上等待一小会儿，就会自动显示该变量的值，如图 5-62 所示。

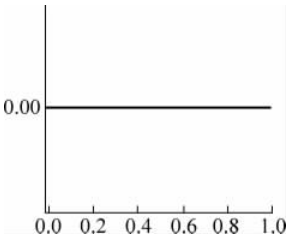


图 5-61　在调试器中以图的方式查看 wave

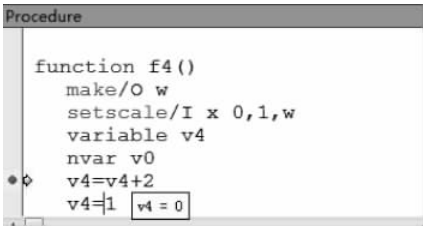


图 5-62　调试器程序窗口中查看变量实时值

除了显示变量的值以外，程序窗口还能显示表达式的值，方法是选择一个表达式，然后将鼠标指针置于表达式之上等待一小会儿，如图 5-63 所示。

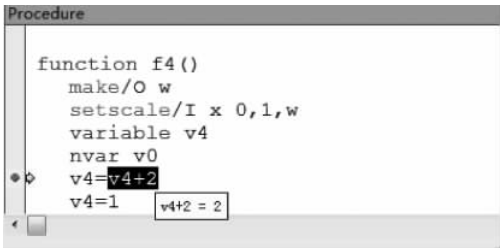


图 5-63　调试器查看表达式的值

通过调试找到程序出错的位置后，就可以按下终止程序执行按钮并返回程序文件进行修改。通过下面的方法可以迅速从调试器程序窗口中定位到程序文件出错代码处：在调试器程序窗口中右击出错函数名，在弹出的快捷菜单中选择【Go To <函数名>】；按 Esc 键或者终止执行程序或者选择正常执行程序退出调试器，也可定位到程序文件出错代码处。还有一个简便的方法是直接终止程序执行（按下调试器终止程序执行按钮），则程序文件终止执行时对应的代码处会被高亮选择。

当程序经过充分调试确认没有任何错误时，应该消除所有断点，以免干扰程序执行，这时可以右击程序文件，在弹出的快捷菜单中选择【Clear All Breakpoints】。

程序的调试是程序开发的重要组成部分，可能有 20％时间是开发程序，剩下的 80％时间都是在调试程序。因此，读者一定要充分掌握程序的调试技巧，善于发现并消除程序中隐藏的错误，提高程序的健壮性。